UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Plataformes avançades en el Núvol per a la reproductibilitat d'experiments computacionals

Tesi Doctoral

Vicent Giménez Alventosa
Institut d'Instrumentació per a Imatge Molecular (I3M)
Programa de Doctorat en Informàtica
Universitat Politècnica de València

Dirigida per,
Germán Moltó Martínez
Damiàn Segrelles Quilis

Març 2022

Índex

Abreviatures

AAPM American Association of Physicists in Medicine. 100

API Application Programming Interface. 12

AWS Amazon Web Services. 15, 16, 19

BaaS Backend as a Service. 12, 13

CE Comissió Europea. 2

DAG Directed Acyclic Graph. 6

EOSC European Open Science Cloud. 2

FaaS Function as a Service. 12–14

GAD Graf Acíclic Dirigit. 6, 7, 21, 22

IaaS Infrastructure as a Service. 10, 14

IAEA International Atomic Energy Agency. 100

IM Infrastructure Manager. 24, 152

MV Màquina Virtual. 2, 15, 17, 27–33

PaaS Platform as a Service. 10, 12–14

RADL Resource and Application Description Language. 24

SaaS Software as a Service. 10

UE Unió Europea. 1

Resum de la tesi

La tesi presentada a aquest document s'emmarca dins de l'àmbit de la ciència computacional. Dintre d'aquesta, es centra en el desenvolupament d'eines per a l'execució d'experimentació científica computacional, la qual té un impacte cada vegada major en tots els àmbits de la ciència i l'enginyeria. Donada la creixent complexitat dels càlculs realitzats, cada vegada és necessari un major coneixement sobre les tècniques i eines disponibles per a dur a terme aquestes experimentacions, ja que poden requerir, en general, una gran infraestructura computacional per afrontar els alts costos de còmput. Més encara, la recent popularització del còmput en el Núvol ofereix una gran varietat de possibilitats per a configurar les nostres pròpies infraestructures amb requisits específiques. No obstant, el preu a pagar és la complexitat de configurar les esmenades infraestructures a aquest tipus d'entorns. A més, l'augment de la complexitat de configuració dels entorns de còmput no ha fet més que agreujar un problema ja existent a l'àmbit científic, i és la reproductibilitat de resultats publicats. La manca de documentació, com les versions del programari emprat per a dur a terme el còmput, o les dades requerides ocasionen que una part no negligible dels resultats d'experiments computacionals publicats no siguen reproduïbles per altres investigadors. Com a conseqüència, es produeix un malbaratament dels recursos destinats a la investigació.

Com a resposta a aquesta situació, existeixen, i continuen desenvolupant-se, diverses eines per facilitar processos com el desplegament i configuració d'infraestructura, l'accés a les dades, el disseny de fluxos de còmput, etc. amb l'objectiu de que els investigadors puguen centrar-se en el problema a abordar. Precisament, aquesta és la base dels treballs desenvolupats durant la tesi que segueix, el desenvolupar eines per a facilitar que el còmput científic es beneficiar-se d'entorns de computació en el Núvol d'una forma eficient.

El primer treball presentat comença amb un estudi exhaustiu de les prestacions d'un servei relativament nou, l'execució *serverless* de funcions. En aquest, es determinarà la conveniència d'emprar este tipus d'entorns en el càlcul científic mesurant tant les seues prestacions de forma aïllada, com velocitat de CPU i la velocitat de les comunicacions, com en conjunt a través del desenvolupament d'una aplicació de processament *MapReduce* per a entorns *serverless*.

Al següent treball, s'abordarà una problemàtica diferent, i és la reproductibilitat dels experiments computacionals. Per a aconseguir-ho, es presentarà una entorn, basat en Jupyter, on s'englobe tant el desplegament i configuració d'infraestructura computacional, com l'accés a les dades requerides i la documentació de l'experimentació. Tota aquesta informació quedarà registrada al *notebook* de Jupyter on s'executa l'experiment, permetent així a altres investigadors reproduir els resultats simplement compartint el *notebook* corresponent.

Tornant a l'estudi de les prestacions del primer treball, donades les mesurades i ben estudiades fluctuacions d'aquestes en entorns compartits, com en el còmput en el Núvol, al tercer treball es desenvoluparà un sistema de balanceig de càrrega dissenyat expressament per aquest tipus d'entorns. Com es veurà, aquest component és capaç de gestionar i corregir de forma precisa fluctuacions impredictibles en les prestacions de còmput d'entorns compartits.

Finalment, i aprofitant el desenvolupament anterior, es dissenyarà una plataforma completament *serverless* per a repartir i balancejar tasques executades en múltiples infraestructures de còmput independents. La motivació d'aquest últim treball ve donada pels alts costos computacionals de certes experimentacions, els quals forcen als investigadors a emprar múltiples infraestructures que, en general, pertanyen a diferents organitzacions. Es demostrarà com la plataforma desenvolupada és capaç de balancejar la càrrega de forma precisa alhora que parteix les tasques per acomplir cotes en el temps d'execució especificades per l'usuari. A més, assisteix en el procés d'escalat de les infraestructures de còmput emprades per tal de minimitzar el malbaratament de recursos de còmput.

Resumen

La tesis presentada se enmarca dentro del ámbito de la ciencia computacional. Dentro de esta, se centra en el desarrollo de herramientas para la ejecución de experimentación científica computacional, el impacto de la cual es cada vez mayor en todos los ámbitos de la ciencia y la ingeniería. Debido a la creciente complejidad de los cálculos realizados, cada vez es necesario un mayor conocimiento de las técnicas y herramientas disponibles para llevar a cabo este tipo de experimentos, ya que pueden requerir, en general, una gran infraestructura computacional para afrontar los altos costes de cómputo. Más aún, la reciente popularización del cómputo en la Nube ofrece una gran variedad de posibilidades para configurar nuestras propias infraestructuras con requisitos específicos. No obstante, el precio a pagar es la complejidad de configurar dichas infraestructuras en este tipo de entornos. Además, el aumento en la complejidad de configuración de los entornos en la nube no hace más que agravar un problema ya existente en el ámbito científico, y es el de la reproducibilidad de los resultados publicados. La falta de documentación,

como las versiones de software que se han usado para llevar a cabo el cómputo, o los datos requeridos, provocan que una parte significativa de los resultados de experimentos computacionales publicados no sean reproducibles por otros investigadores. Como consecuencia, se produce un derroche de recursos destinados a la investigación.

Como respuesta a esta situación, existen, y continúan desarrollándose, diferentes herramientas para facilitar procesos como el despliegue y configuración de infraestructura, el acceso a los datos, el diseño de flujos de cómputo, etc. con el objetivo de que los investigadores puedan centrarse en el problema a abordar. Precisamente, esta es la base de los trabajos desarrollados en la presente tesis, el desarrollo de herramientas para facilitar que el cómputo científico se beneficie de entornos de computación en la Nube de forma eficiente.

El primer trabajo presentado empieza con un estudio exhaustivo de las prestaciones d'un servicio relativamente nuevo, la ejecución *serverless* de funciones. En este, se determinará la conveniencia de usar este tipo de entornos en el cálculo científico midiendo tanto sus prestaciones de forma aislada, como velocidad de CPU y comunicaciones, como en conjunto mediante el desarrollo de una aplicación de procesamiento *MapReduce* para entornos *serverless*.

En el siguiente trabajo, se abordará una problemática diferente, y es la reproducibilidad de experimentos computacionales. Para conseguirlo, se presentará un entorno, basado en *Jupyter*, donde se encapsule tanto el proceso de despliegue y configuración de infraestructura computacional como el acceso a datos y la documentación de la experimentación. Toda esta información quedará registrada en el *notebook* de *Jupyter* donde se ejecuta el experimento, permitiendo así a otros investigadores reproducir los resultados simplemente compartiendo el *notebook* correspondiente.

Volviendo al estudio de las prestaciones del primer trabajo, teniendo en cuenta las medidas y bien estudiadas fluctuaciones de éstas en entornos compartidos, como el cómputo en la Nube, en el tercer trabajo se desarrollará un sistema de balanceo de carga diseñado expresamente para este tipo de entornos. Como se mostrará, este componente es capaz de gestionar y corregir de forma precisa fluctuaciones impredecibles en las prestaciones del cómputo en entornos compartidos.

Finalmente, y aprovechando el desarrollo anterior, se diseñará una plataforma completamente *serverless* encargada de repartir y balancear tareas ejecutadas en múltiples infraestructuras independientes. La motivación de este último trabajo viene dada por los altos costes computacionales de ciertos experimentos, los cuales fuerzan a los investigadores a usar múltiples infraestructuras que, en general, pertenecen a diferentes organizaciones. Se demostrará como la plataforma desarrollada es capaz de balancear la carga de forma precisa al mismo tiempo que divide las tareas automáticamente para cumplir con una cota en el tiempo de ejecución

especificada por el usuario. Además, la plataforma asiste a las infraestructuras de cómputo en el proceso de escalado de recursos, minimizando el exceso de recursos y proporcionando así un uso eficiente de los recursos de cómputo.

Abstract

This document is focused on computational science, specifically in the development of tools for executions of scientific computational experiments, whose impact has increased, and still increasing, in all scientific and engineering scopes. Considering the growing complexity of scientific calculus, it is required large and complex computational infrastructures to carry on the experimentation. However, to use this infrastructures, it is required a deep knowledge of the available tools and techniques to be handled efficiently. Moreover, the popularity of Cloud computing environments offers a wide variety of possible configurations for our computational infrastructures, thus complicating the configuration process. Furthermore, this increase in complexity has exacerbated the well known problem of reproducibility in science. The lack of documentation, as the used software versions, or the data required by the experiment, produces non reproducible results in computational experiments. This situation produce a non negligible waste of the resources invested in research.

As consequence, several tools have been developed to facilitate the deployment, usage and configuration of complex infrastructures, provide access to data, etc. with the objective to simplify the common steps of computational experiments to researchers. Moreover, the works presented in this document share the same objective, i.e. develop tools to provide an easy, efficient and reproducible usage of cloud computing environments for scientific experimentation.

The first presented work begins with an exhaustive study of the suitability of the AWS *serverless* environment for scientific calculus. In this one, the suitability of this kind of environments for scientific research will be studied. With this aim, the study will measure the CPU and network performance, both isolated and combined, via a MapReduce framework developed completely using *serverless* services.

The second one is focused on the reproducibility problem in computational experiments. To improve reproducibility, the work presents an environment, based on Jupyter, which handles and simplify the deployment, configuration and usage of complex computational infrastructures. Also, includes a straight forward procedure to provide access to data and documentation of the experimentation via the Jupyter notebooks. Therefore, the whole experiment could be reproduced sharing the corresponding notebook.

In the third work, a load balance library has been developed to address fluc-

tuations of shared infrastructure capabilities. This effect has been wide studied in the literature and affects specially to cloud computing environments. The developed load balance system, as we will see, can handle and correct accurately unpredictable fluctuations in such environments.

Finally, based on the previous work, a completely *serverless* platform is presented to split and balance job executions among several shared, heterogeneous and independent computing infrastructures. The motivation of this last work is the huge computational cost of many experiments, which forces the researchers to use multiple infrastructures belonging, in general, to different organisations. It will be shown how the developed platform is capable to balance the workload accurately. Moreover, it can fit execution time constrains specified by the user. In addition, the platform assists the computational infrastructures to scale as a function of the incoming workload, avoiding an over-provisioning or under-provisioning. Therefore, the platform provides an efficient usage of the available resources.

Capítol I

Introducció

1 Contextualització

L'augment de la complexitat computacional dels experiments científics al llarg del temps, ha forçat que cada vegada es desenvolupen infraestructures majors per tal d'afrontar aquesta demanda creixent. Un exemple clar es veu en l'increment de la potencia computacional dels superordinadors entre els anys 1992 i 2008, que va ser d'un factor 10000 [1]. D'altra banda, la computació en el núvol s'ha popularitzat ràpidament. Tant es així que la mitjana d'empreses que han contractat serveis de computació en el núvol a la Unió Europea (UE) és d'un 26% en 2018 [2], havent augmentat dit percentatge any rere any. Més encara, a països com Finlàndia, Suècia, Dinamarca o Noruega, aquest percentatge supera el 50%. Donats ambdós factors, no és d'estranyar que nombrosos treballs hagen estudiat la conveniència de traslladar el còmput científic a un entorn de computació en el núvol. Aquests estudis, que es discutiran al llarg del present document, es motiven pels ben coneguts avantatges de la computació en el núvol. Entre ells, destaquen el model de pagament per ús, la flexibilitat que proporciona un entorn virtualitzat, ja que permet configurar totes les dependències requerides sense dependre d'un administrador, i la capacitat d'escalar la infraestructura segons les necessitats de l'usuari. No obstant, l'alternativa més emprada avui en dia continua sent emprar el maquinari instal·lat en una o més institucions d'investigació, com poden ser *clusters* de maquinari o superordinadors. La forma d'emprar dit maquinari depèn de l'organització de cadascuna de les infraestructures. Pot consistir en accedir directament a un conjunt de màquines a través d'un *frontend* i un sistema de cues, a través d'una organització de tipus graella (de l'angès *Grid* [3]) que aglutine els recursos de diferents infraestructures, emprant un proveïdor de computació en el núvol privat, com OpenNebula [4] o OpenStack [5], etc.

Els fets anteriors plantegen, per tant, la necessitat d'estudiar la viabilitat,

inconvenients i avantatges de traslladar el còmput científic als nous models de computació en el núvol, ja siga de forma parcial, combinada amb infraestructures locals, o total. Un altre punt a favor dels models de còmput basats en infraestructures virtualitzades, és que proporcionen un avantatge clau en l'àmbit científic, que no és més que la capacitat de reproductibilitat de resultats computacionals. Donat que els entorns complets on s'executa l'experiment es poden encapsular, ja siga a una imatge de Màquina Virtual (MV) o un contenidor, es pot facilitar la tasca de reproducció dels resultats per part d'un investigador independent. Aquesta capacitat s'estén, inclús, a la possibilitat de detallar i reproduir el desplegament de la infraestructura necessària per executar un determinat experiment computacional, com discutirem a la secció 4. La importància d'esta característica s'emfatitza per la creixent preocupació sobre la dificultat de reproduir resultats de publicacions científiques, ja siga intencionat, frau, o per la manca de detalls sobre la metodologia, requisits de maquinari i programari, dades, etc. Tant és així que l'enquesta realitzada per Monya Baker [6] a 2016 revelava que el 90% dels investigadors reconeixien que existeix una crisi de reproductibilitat de resultats, en major o menor mesura. Un camp on dita característica és especialment crítica és l'àmbit mèdic, on la manca de reproductibilitat en els resultats impossibilita que puguen ser emprats per millorar la salut pública, malbaratant així una gran quantitat de temps i recursos. De fet, al treball de Freedman et al. [7] defenen que més de la meitat dels estudis preclínics no són reproduïbles, el que es tradueix, únicament en Estats Units, en $28,000,000,000$\$ gastats en investigació no reproduïble.

Aquesta situació ha provocat un moviment global en defensa de la investigació reproduïble, tant per evitar el frau intencionat com el malbaratament de recursos per la manca d'informació. Específicament a Europa, la Comissió Europea (CE) ha introduït la ciència oberta, de l'anglès *Open Science*, amb l'objectiu de promoure un nou model del procés científic basat en el treball cooperatiu, i en noves formes de difondre el coneixement a través de les noves tecnologies digitals i eines dissenyades per a fomentar la col·laboració [8]. Més encara, les institucions Europees han considerat la Ciència Oberta com un component necessari per a futurs programes d'investigació. Sense anar més lluny, a 2015, la CE va fixar tres objectius per al futur de la investigació, que són, *Open Science, Open Innovation and Open to the world* [9, 8]. A més, es va crear i promoure el *European Open Science Cloud (EOSC)* [10] amb l'objectiu de crear un entorn confiable per emmagatzemar i processar dades d'investigació. Cal remarcar que aquest moviment de defensa a la Ciència Oberta no es restringeix a Europa, sinó que ha sigut un moviment global adoptat per institucions arreu del món. Per exemple, als Estats Units trobem iniciatives com la *Berkeley Initiative for Transparency in the Social Sciences* [11], la *Public Library Of Science* (PLOS) [12] i el *Center for Open Science* [13].

Malauradament, la computació en el núvol te desavantatges que han d'estudiar-se per tal de determinar si una experimentació concreta pot ser duta a terme en quest tipus d'entorns de forma eficient, o, si pel contrari, encaixa millor en una infraestructura a l'ús. Un dels principals inconvenients, tot i que no és un problema exclusiu dels entorns de computació en el Núvol, és la variabilitat en les prestacions durant l'execució d'aplicacions, el qual es discutirà amb detall a la secció 3.2. Unes prestacions impredictibles poden ocasionar retards i malbaratament de recursos, sobretot en aplicacions científiques paral·leles, on diversos processos poden compartir informació de forma regular per avançar conjuntament en el càlcul. En aquest cas, una falta de sincronització entre processos podrà provocar temps d'espera comparables al temps d'execució, els quals poden involucrar costos extra. Més encara, donada la creixent diversitat de serveis oferits pels diferents proveïdors de computació en el núvol, la simple elecció del servei o combinació de serveis on realitzar l'execució ja implica un problema no trivial, ja que s'ha d'arribar a un compromís per tal de maximitzar la productibilitat i minimitzar el cost.

1.1 Objectius

Pels motius descrits anteriorment, l'objectiu global del present document, esquematitzat a la Figura I.1, consisteix en avançar en el desenvolupament de plataformes per a l'execució i reproductibilitat de còmput científic en entorns de computació en el Núvol.

Seguint l'esquema mostrat, s'abordarà, d'una banda, l'estudi de la idoneïtat i eficiència de les execucions en aquests entorns, per tal de minimitzar els costos associats i maximitzar la productivitat. Per aconseguir-ho, al primer treball es planteja un estudi exhaustiu de les prestacions d'un dels serveis *serverless* més emprats avui en dia. Aquest estudi, junt a altres que es troben a la literatura, serviran com a fonament per a identificar problemes a l'hora de realitzar experimentacions científiques i, a més, per desenvolupar eines que els mitiguen. Les esmenades eines consistiran en un mètode de distribució de la càrrega per a execucions distribuïdes, presentat al tercer treball, i un servei per combinar la capacitat de còmput de múltiples infraestructures, presentada al quart treball. L'últim dels anteriors tindrà com objectiu extra facilitar l'ús i accés a les infraestructures, ja que gestionarà els treballs a executar de forma transparent per a l'investigador. Finalment, tenint en compte la tendència a fomentar la ciència oberta, es posarà especial èmfasis en la reproductibilitat de l'entorn d'execució, tant de programari com de la infraestructura requerida. Aquest aspecte s'estudiarà al segon treball presentat, en el qual es proveeix a l'investigador d'eines per tal de desplegar infraestructures computacionals complexes i configurar-les de forma senzilla i reproduïble, facilitant al seu torn, una vegada més, l'accés a infraestructures computacionals.

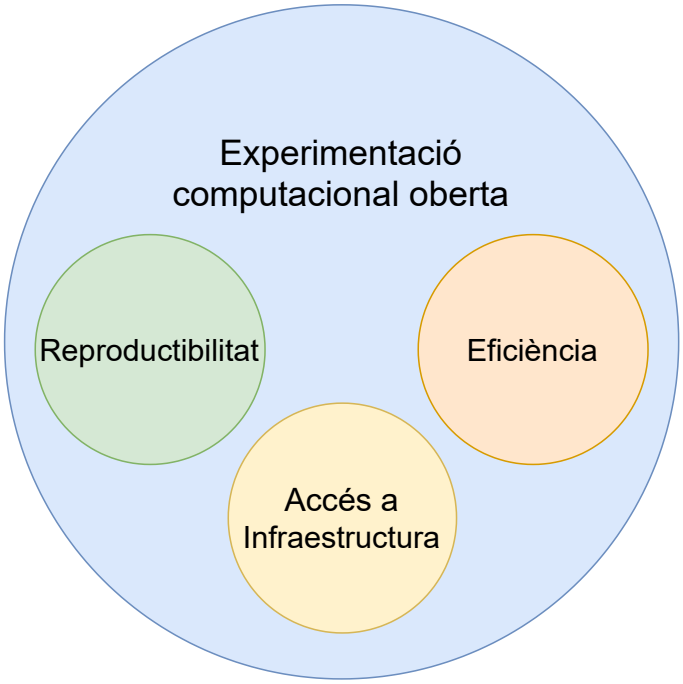

Figura I.1: Components per a la experimentació computacional oberta.

Cal remarcar que el cas a estudiar correspon a aplicacions científiques, les quals presenten característiques específiques que les diferencien d'aplicacions típiques executades en el Núvol, com, per exemple, una aplicació web. Les principals característiques a considerar es llisten a continuació,

- **Alt cost computacional**: Molts experiments científics involucren un alt cost computacional que sol traduir-se en execucions llargues i amb un ús elevat i continuat dels recursos de maquinari.
- **Alt grau de paral·lelisme**: Per afrontar els elevats costos computacionals, típicament s'empren tècniques de paral·lelisme per tal d'accelerar el còmput i maximitzar l'eficiència de l'execució. Al seu torn, dit paral·lelisme pot produir dependències entre els processos involucrats, requerint així l'intercanvi de dades entre aquests. Per tant, les prestacions de les interconnexions entre nodes poden arribar a tindre un paper determinant.
- **Requisits de maquinari**: Algunes aplicacions, com programari de simulació, requereix característiques específiques en el maquinari per a ser executades, com pot ser un alt consum de memòria o l'ús d'acceleradors de còmput com GPUs. Tot i que aquest tipus de maquinari s'ofereix en molts proveïdors, involucra costos elevats i pot estar limitat a certs serveis, pel que es requereix una anàlisi prèvia per a avaluar la seua conveniència.

1.2 Estructura de la tesi

El present document s'estructura de la forma següent: En primer lloc, s'introduiran els fonaments teòrics que han sigut necessaris per al desenvolupament dels treballs presentats i, per tant, per l'acompliment dels objectius presentats. Aquesta descripció tindrà lloc al present capítol (capítol I), on s'introduiran models de computació paral·lela, que és i com s'organitza el còmput en el núvol, així com algunes de les seues característiques i inconvenients claus a resoldre en els treballs desenvolupats, i, finalment, la reproductibilitat en el còmput científic.

A continuació, al capítol II es desenvoluparan models matemàtics amb l'objectiu d'obtindre el màxim rendiment dels models de còmput descrits al capítol I. Els models podran ser emprats conjuntament amb els treballs presentats per arribar a un compromís entre eficiència i costos d'execució depenent de les característiques de l'aplicació.

Finalment, al capítol III, es presenten tots els treballs desenvolupats i en els quals es basa aquesta tesi. El capítol acaba amb una recopilació dels resultats obtinguts i les conclusions globals de la tesi, així com una proposta per a futures línies d'investigació.

2 Models de computació paral·lela

Els models de computació paral·lela es caracteritzen principalment perquè els càlculs són processats de forma simultània per diferents processos. Aquest model de processament ha sigut extensament emprat per al còmput científic des de la popularització de les computadores per tal d'afrontar els llargs temps de còmput necessaris per a certs experiments. No obstant, per aplicar-se es necessari identificar les parts del problema que poden ser processades de forma simultània per a reestructurar el còmput de forma adient.

Avui en dia, els experiments científics requereixen una gran potencia computacional per tal d'afrontar els llargs processos involucrats. Per exemple, al cas de simulacions de factors correctors de cambres d'ionització per al seu ús clínic, el treball [14] va requerir, aproximadament, unes 13800 hores de CPU per cadascuna de les combinacions de feix i cambra simulades en l'estudi, el que es tradueix en un total de 745200 hores de CPU. Aquests requeriments no són abordables emprant simplement paral·lelisme a una única maquina, ja siga amb processadors amb múltiples cors o amb maquinari amb un nombre limitat de processadors que comparteixen una mateixa memòria, sinó que es requereix el treball conjunt de maquinari independent, conegut com a paral·lelisme amb memòria distribuïda. En aquest, múltiples nodes treballen conjuntament per a dur a terme un càlcul comú i mantenen, en general, comunicació entre ells. Dita comunicació pot ser a

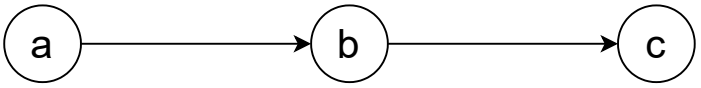

Figura I.2: GAD simple amb tres vèrtex.

través d'una xarxa d'interconnexió d'altes prestacions, com el cas de *clusters* de processament, o inclús, directament a través d'Internet per interconnectar maquinari distribuït arreu del mon. El tipus adient dependrà de la freqüència i volum de les comunicacions necessàries per al processament, ja que, en general, els nodes hauran de compartir informació per tal de dur a terme el còmput.

Donades aquestes necessitats, s'han creat diferents entorns paral·lels, escalables i eficients per al processament massiu de dades. Aquests ofereixen als usuaris un esquema o model del flux de treball on ajustar el seu còmput específic, amb el benefici de que tasques com la gestió dels recursos, treballs, resultats, etc. es realitza automàticament per part de l'entorn. Per exemple, centrat en el processament de grans volums de dades, Apache Spark [15] facilita l'execució d'aplicacions en *clusters*, que poden estar codificades en Java, Scala, Python, R, o SQL. Els sistemes de flux de treball han guanyat molta popularitat gracies a la simplificació de la gestió que ofereixen. Tant es així que actualment existeixen, i són emprats per la comunitat, una gran diversitat d'implementacions, com *Perfect* [16], *Conductor* [17], *Pegasus* [18], *MakeFlow* [19], Airavata [20] o WFE [21] entre altres.

Molts d'aquests fluxos de treball es basen en estructures de Graf Acíclic Dirigit (GAD), de l'anglès *Directed Acyclic Graph (DAG)*. Dites estructures estan formades per vèrtexs, que al seu torn s'uneixen per arestes. No obstant, les arestes no poden formar un bucle tancat. En el cas particular que ens ocupa, els fluxos de treball computacional, els vèrtex solen representar tasques a computar, mentre que les arestes són les dependències entre elles. Per tant, per a la combinació de tot vèrtex a unit a un vèrtex b posterior per l'aresta $\overline{ab}$, s'acomplirà que l'execució a serà prèvia a b, ja que el resultat d'a serà necessari per computar b. D'igual forma, si el GAD està composat, a més, per un vèrtex c unit per l'aresta $\overline{bc}$ (Figura I.2), l'ordre de les execucions serà $a < b < c$.

Noteu que aquesta propietat es manté per a GADs més complexos, on d'un mateix vèrtex poden sorgir o arribar múltiples arestes, tal i com es mostra a la Figura I.3.

L'organització de les execucions en GAD s'ha popularitzat molt. De fet, nombrosos estudis recents exploten aquesta tècnica. Alguns exemples són el treball de Juwei i Jiaheng [22], on es proposa un model de cost per optimitzar l'elecció de recursos i identificar el coll de botella de l'execució, l'article de Peerasak et al. [23], on es proposa un nou entorn de computació paral·lela on la distribució de les tasques es basa en un flux GADs, o el treball de Benjamin et al. [24], on presenten

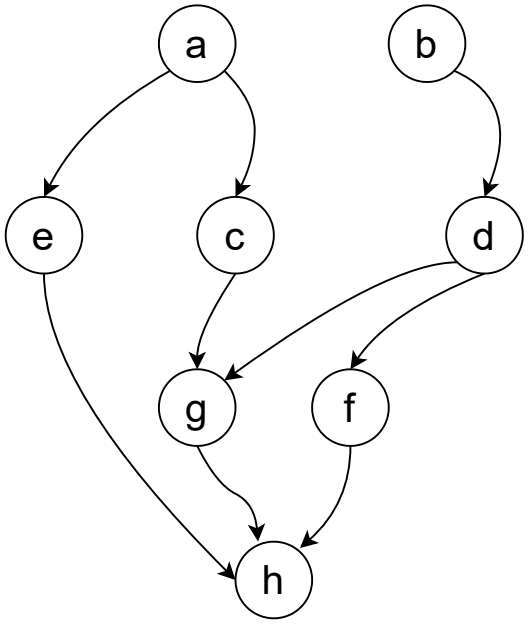

Figura I.3: GAD amb múltiples arestes per vèrtex.

wukong, un entorn *serverless* per a l'execució d'aplicacions seguint el model GAD.

Un cas particular del model GAD és el tipus de processament de dades *MapReduce*. Aquest model s'ha popularitzat molt per al processament de grans volums de dades, i es poden trobar diferents implementacions de codi obert, com Apache Hadoop [25] o MongoDB [26]. Per la seua popularitat i utilitat, el model *MapReduce* es descriurà amb més detall a la secció 2.1 i s'emprarà com a base del primer treball presentat a aquest document.

2.1 *Map Reduce*

El model de processament *Map Reduce* s'empra per processar un gran conjunt de dades de forma paral·lela i distribuïda [27]. Aquest, es composa principalment de dues parts, el mapatge de les dades i la reducció del resultat del mapatge.

El procés de mapatge consisteix en filtrar les dades i ordenar el resultat per a que, posteriorment, siguen agrupades pel procés de reducció, tal i com s'esquematitza a la Figura I.4. Per exemple, suposem que es vol processar el registre de les connexions d'entrada d'un servidor on cada línia del registre segueix el següent format,

```
dia IP connexions
```

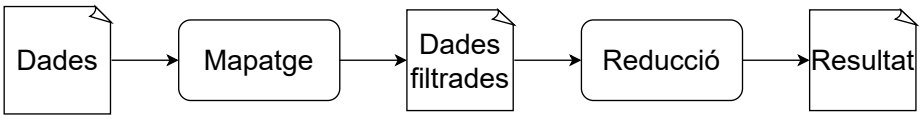

Figura I.4: Esquema de funcionament del procés *MapReduce*.

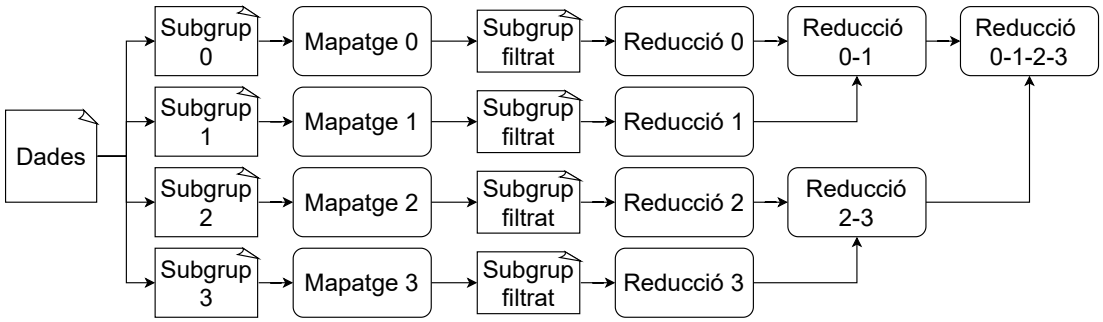

Figura I.5: *MapReduce* paral·lel amb reducció amb cost logaritmic.

on *connexions* és el nombre de connexions rebudes des de la direcció *IP* durant el dia especificat per *dia*. Suposem, a més, que el que es vol calcular és el nombre de connexions totals d'una IP determinada. Per a processar el fitxer, s'executarà primer un pas de mapatge on es descartarà tota línia la IP de la qual no coincidisca amb la seleccionada. Tot seguit, al pas de reducció, es sumaran totes les connexions de les línies resultants del mapatge.

L'avantatge d'organitzar el processament de dades amb aquesta metodologia radica en que és fàcilment paral·lelitzable. La simple divisió de les dades d'entrada en grups més menuts permet distribuir-les i ser processades entre tots els treballadors disponibles. El preu a pagar és que s'han de reunir les dades resultants de mapatge per tal d'executar la reducció. Depenent del cost del procés de reducció i el volum de dades resultant, una opció per executar la reducció pot consistir en una aproximació de cost $O(log_2(N_t))$, on N_t és el nombre total de treballadors. En aquesta, si els treballadors estan identificats numèricament com $[0, 1, ..., N_t - 1]$, els treballadors amb identificador senars enviaran les dades resultants d'una primera reducció al node anterior, el qual processarà ambdós conjunts de dades. Aquest procediment es repetirà fins que el node 0 obtinga el resultat global final de la reducció, tal i com es mostra en la Figura I.5.

No obstant, depenent del cas, és possible que siga necessari o convenient agrupar les dades resultants del mapatge per a que grups específics siguen reduïts junts, per exemple per ordre alfabètic. Si fora el cas, caldria afegir un pas de reordenació abans de les reduccions individuals de cada treballador, podent arribar a un cost de les comunicacions comparable o major que el de processament. D'altra banda, si el cost del procés de reducció és relativament baix, l'opció més simple consisteix

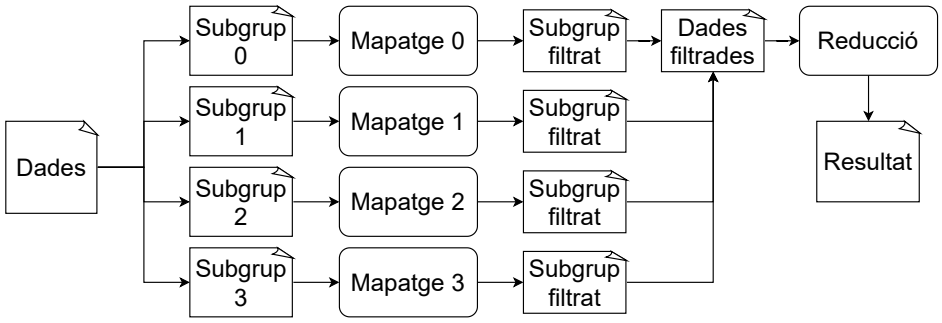

Figura I.6: *MapReduce* paral·lel amb reducció simple.

en reagrupar els resultats de tots els mapatges a un únic node i executar la reducció sobre les dades completes (Figura I.6). També es pot optar per una estratègia combinada depenent del nombre de treballadors que s'estiguen emprant per tal d'arribar al millor compromís entre minimitzar les comunicacions i el temps de processament [28].

Tal i com es conclou al treball publicat per Jeffrey [28], l'algorisme òptim per a un processament *MapReduce* dependrà del tipus de processament en sí, pel que caldrà un estudi especialitzat. Aquest estudi es discutirà a la secció 2 del capítol II.

3 Computació en el Núvol

La computació en el núvol, segons la descripció del NIST [29], és un model de computació que ofereix accés a recursos computacionals configurables que poden ser aprovisionats i alliberats ràpidament amb un mínim esforç i interacció amb el proveïdor. Segons la seua definició, aquest model de computació es composa de cinc característiques essencials que es descriuen a continuació,

- **Autoservici baix demanda**: Els recursos poden ser tant aprovisionats com alliberats per l'usuari segons les seues necessitats.
- **Accés a través de la xarxa**: L'accés als esmenats recursos i la seua gestió es realitzen a través de la xarxa.
- **Posada en comú de recursos**: Els recursos computacionals es comparteixen pels múltiples usuaris de la plataforma. La localització específica dels recursos que s'empren per part de l'usuari quedarà baix el control del proveïdor de recursos, sent, en general, desconeguda. No obstant, es podrà especificar la regió on es troben localitzats els recursos, com per exemple el país.

Aquesta implementació permet als proveïdors migrar els serveis emprats pels usuaris entre els recursos de maquinari disponibles de forma eficient, emprant diferents tècniques, com la presentada per Khan et al. [30] per tal de minimitzar el cost energètic associat i mantenir un nivell de qualitat de servei determinat.

- **Ràpida elasticitat o expansió**: L'aprovisionament i alliberament de recursos es realitza de forma ràpida. Aquesta característica permet a l'usuari adaptar el nombre de recursos contractats en funció de la seua demanda i evitar així costos associats a una infraestructura infrautilitzada.

- **Servei monitoritzat**: Els serveis aprovisionats són monitoritzats pel proveïdor de computació per diferents motius, com controlar la facturació o emprar eficientment el maquinari disponible.

D'altra banda, el NIST identifica tres tipus de models de servici,

- **Programari com a Servici**: De l'anglès *Software as a Service (SaaS)*. En aquest model, el proveïdor proporciona un servei o aplicació que serà executada en la infraestructura en el Núvol i podrà ser accedida pels usuaris a través de dispositius, com un mòbil o portàtil, i una interfície, com pot ser un explorador web. D'altra banda, l'usuari no gestiona directament els recursos de maquinari ni l'entorn de programari on s'executa l'aplicació, incloent sistema operatiu, emmagatzemament, xarxa, etc. No obstant, es possible que s'oferisquen opcions de configuració limitades.

- **Plataforma com a Servici**: De l'anglès *Platform as a Service (PaaS)*. A diferència del programari com a servici, este model permet executar aplicacions creades per l'usuari a partir de llenguatges de programació, llibreries, servicis i eines suportades pel proveïdor. No obstant, tal i com ocorre al model anterior, no es disposa de control directe sobre les característiques del maquinari ni l'entorn de programari on es desplega l'aplicació, com el sistema operatiu.

- **Infrastructura com a Servici**: De l'anglès *Infrastructure as a Service (IaaS)*. Finalment, la infraestructura com a servici permet a l'usuari aprovisionar tot tipus de recursos, com processament, emmagatzemament, xarxes de connexió, etc. A més, permet controlar completament l'entorn de programari on executar aplicacions, com el sistema operatiu, llibreries, etc. Açò s'aconsegueix, típicament, mitjançant entorns virtualitzats definits per l'usuari i que seran executats en el maquinari del proveïdor. No obstant, l'usuari no disposarà d'accés directe al maquinari i la xarxa, més enllà d'opcions de configuració limitades.

A més, es defineixen també els models de desplegament segons el tipus de proveïdor de computació,

- **Proveïdor privat**: La infraestructura és aprovisionada per una única organització. Per tant, dita infraestructura pertany i serà gestionada per dita organització. L'ús d'aquesta quedarà restringida a la pròpia organització i, possiblement, a tercers autoritzats.

- **Proveïdor comunitari**: La infraestructura es destina a l'ús exclusiu d'un conjunt d'organitzacions. Aquesta, pot pertànyer i ser gestionada per una o més de les organitzacions que conformen la comunitat, o un tercer.

- **Proveïdor públic**: En aquest cas, l'ús de la infraestructura s'obri a tot el públic, de forma que qualsevol pot emprar els seus recursos. Aquesta, pertanyerà i serà gestionada per una empresa, una institució acadèmica, una organització governamental o una combinació d'aquestes.

- **Proveïdor híbrid**: En un proveïdor híbrid la infraestructura està composada per dos o més infraestructures els proveïdors de les quals segueix un dels tipus abans descrits. Cadascuna d'aquestes opera de forma independent, però permeten la portabilitat de dades i aplicacions entre elles per mitjà d'estàndards o tecnologia pròpia.

El model de còmput en el Núvol proporciona nombrosos beneficis front a les clàssiques infraestructures científiques situades físicament a les institucions d'investigació. En primer lloc, el manteniment dels aparells. Donat que en la computació en el Núvol els recursos de maquinari són gestionats per el proveïdor, es redueix notablement l'esforç requerit per gestionar la infraestructura. D'altra banda, els costos. Sovint les infraestructures destinades a investigació pateixen períodes de temps on es troben infrautilitzades, el que provoca un cost econòmic que no s'està invertint realment en el desenvolupament científic. No obstant, la computació en el núvol ofereix sistemes de pagament per ús, podent alliberar els recursos que no s'empren per a evitar costos addicionals. Més encara, al mateix temps que permet l'alliberament de recursos, es possible aprovisionar-ne de nous, podent adaptar la infraestructura a les necessitats específiques de cada període de temps. Finalment, un dels principals avantatges és la capacitat de configuració dels recursos. Donat que aquest tipus de serveis s'ofereixen sobre entorns virtualitzats, l'usuari pot configurar completament els recursos amb el programari adient per a la seua experimentació.

Donat els avantatges presentats, l'adaptació i estudi de la viabilitat de migrar l'execució d'aplicacions científiques a entorns de computació en el Núvol ha sigut un tema d'interès als últims anys. No obstant, com es veurà a la secció 3.2,

existeixen problemes que poden limitar la seua viabilitat depenent de l'aplicació considerada.

La popularitat del model ha propiciat que els diferents proveïdors de computació en el Núvol increment el nombre de serveis oferits als usuaris. Entre aquests, destaca la computació *Serverless*, on l'usuari ni tan sols ha de preocupar-se de desplegar el recurs computacional, sinó únicament del desenvolupament de l'aplicació a executar-se. Per la seua simplicitat d'ús i altres beneficis, al present document s'ha estudiat les característiques d'aquest servei, específicament al treball de la secció 1.

3.1 *Serverless*

El model de computació *serverless* es caracteritza perquè l'aprovisionament i gestió dels recursos el realitza el proveïdor, deixant a l'usuari únicament la tasca de desenvolupar la seua aplicació. Els recursos son aprovisionats sota demanda de forma automàtica, el que proporciona un escalat automàtic i transparent a l'usuari. Donat que l'usuari no configura els recursos de la infraestructura, el que te a disposició són característiques limitades a configurar, de forma similar al model PaaS.

El principal avantatge del *serverless* és el model de cost. Donat que no hi ha un aprovisionament previ d'infraestructura per part de l'usuari, únicament es comptabilitzen costos per l'execució dels recursos *serverless*, ja siga a partir del temps d'execució, per nombre de peticions a una *Application Programming Interface (API)* , quantitat de dades emmagatzemades, etc., evitant així costos per possibles recursos infrautilitzats. D'altra banda, al no gestionar la infraestructura, l'usuari no necessita gastar temps ajustant polítiques i sistemes per escalar-la.

Dins del model *serverless*, els proveïdors ofereixen un gran nombre de serveis amb funcionalitat específica, que es poden connectar a través d'una interfície a l'aplicació de l'usuari. Aquest model es conegut típicament com *Backend as a Service (BaaS)*, i s'empra, per exemple, per a emmagatzemar informació d'una aplicació, enviar notificacions, sistemes d'autenticació, etc. L'objectiu és delegar serveis comunament presents a moltes aplicacions per a que siguen gestionats pel proveïdor, simplificant així el desenvolupament, augmentant la productivitat i, potencialment, estalviant diner. Un àmbit on s'empra molt el model BaaS, és en el desenvolupament d'aplicacions de dispositius mòbils. A la Figura I.7 vegem un esquema d'una possible aplicació on s'empren recursos BaaS per a l'autenticació d'usuaris i l'emmagatzemament de dades.

D'altra banda, trobem el model *serverless* de funció com a servici, de l'anglès *Function as a Service (FaaS)*. La diferència principal respecte al model BaaS, és que, en aquest, l'usuari defineix la funció, programada en algun llenguatge de programació suportat, que va a executar-se en l'entorn *serverless*. És a dir, es poden

Figura I.7: Esquema d'aplicació per a dispositius mòbil on s'empren recursos BaaS per a l'autenticació i l'emmagatzematge de dades.

realitzar execucions generals, dins de les limitacions de cada entorn particular, i no queda restringit a una funcionalitat específica. A l'igual que abans, els recursos on són executades dites funcions són completament gestionats pel proveïdor, tant l'aprovisionament, alliberament com l'escalat. Per tant, l'usuari pot executar el seu codi sense preocupar-se de la gestió de la infraestructura.

El proveïdor s'encarrega d'escalar el nombre d'instàncies a executar segons la demanda, que típicament ve determinada per un model de programació dirigit a esdeveniments. És a dir, les funcions definides per l'usuari són executades com a reacció a un o més esdeveniments concrets, com per exemple escriure dades a una base de dades o rebre una petició HTTP. Els esdeveniments són gestionats per serveis que també proporciona el proveïdor de computació en el Núvol, el que millora el desenvolupament del *back-end* format per l'execució *serverless*, i la seua comunicació amb el *front-end*. De fet, molts d'aquests esdeveniments configurables provenen de serveis BaaS oferits pel proveïdor, pel que és possible construir aplicacions completes emprant únicament components *serverless*. Prenent com a exemple l'emprat en el model BaaS (Figura I.7), es pot afegir una funció FaaS per executar un processament quan les dades són emmagatzemades, tal i com mostra la Figura I.8. En aquest exemple, la funció de processament de dades definida per l'usuari serà invocada o instanciada cada vegada que s'emmagatzemen dades, és a dir, s'executa com a reacció a un esdeveniment determinat. Cal remarcar que cada instància rebrà les dades corresponents que han provocat la seua execució, pel que cadascuna es considerarà un procés aïllat.

Tot i que el model *serverless* guarda moltes similituds amb el model de desplegament PaaS, ja que en ambdós casos els servidors són gestionats pel proveïdor, es tracta de models ben diferenciats. La diferència principal és que en el desplega-

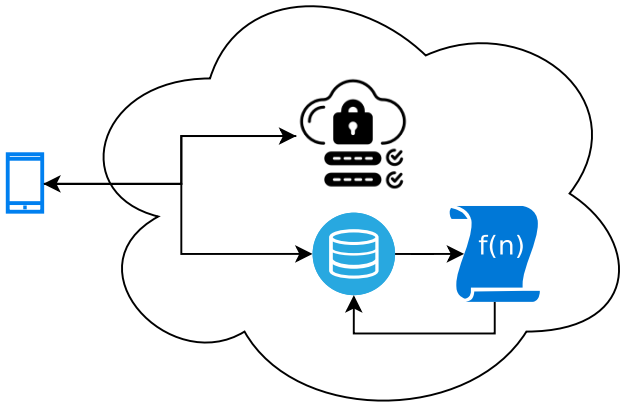

Figura I.8: Exemple de l'esquema de la Figura I.7 afegint un component FaaS per al processament de dades d'entrada.

ment *serverles* els costos van associats únicament a l'execució dels serveis, mentre que al desplegament PaaS l'usuari ha de pagar pels costos dels servidors, on, com a mínim, es requerirà un per a poder rebre les peticions. A més, l'escalabilitat en el PaaS s'aconsegueix desplegant més servidors, els quals ha de pagar l'usuari independentment del volum de treball, al contrari que al cas *serverless*, on el nombre de servidors és transparent a l'usuari i únicament produïra costos quan s'empren els serveis.

Aquestes característiques fan del model *serverless* FaaS un entorn a estudiar per a ser integrat en fluxos de treball d'aplicacions científiques. De fet, actualment s'estudia la conveniència de la seua integració [31] i ja existeixen sistemes de flux de treball que l'empren, com DEWE [32] i Wukong [33].

Tot i els avantatges del model FaaS, a l'inici de la present tesi doctoral els proveïdors aplicaven restriccions que limitaven en gran mesura el seu ús en aplicacions científiques. Per exemple, a AWS el temps execució estava limitat a 5 minuts i la memòria principal màxima que es podia emprar per instància eren 3 GB. Afortunadament, aquests límits van ampliant-se a mesura que passa el temps i la computació FaaS guanya popularitat. De fet, actualment AWS ha triplicat el temps màxim d'execució a 15 minuts i cada instància pot comptar ara amb 10 GB de memòria principal [34]. També cal destacar el servei de Microsoft *Azure Functions*, que permet, depenent del pla contractat, executar funcions amb un límit de temps d'execució de 10 minuts o un temps il·limitat [35].

A mes, depenent de factors com el cost per instància o la freqüència de les execucions FaaS, podrà ser convenient emprar un altre model de desplegament, com el IaaS, per tal de disminuir costos. Realitzar un anàlisis detallat de l'aplicació

a executar és necessari per emprar de forma eficient els recursos disponibles, i les pautes per a dur-lo a terme es descriuran al capítol II.

3.2 Heterogeneïtat i rendiment en infraestructures compartides

Tots els entorns de computació a gran escala moderns empren la compartició de recursos per tal de maximitzar l'ús de la infraestructura. Açò és un mecanisme necessari no únicament per a que els recursos computacionals siguen accessibles per a més investigadors, sinó també per reduir la contaminació associada a l'enorme consum energètic d'aquestes infraestructures. De fet, un ordinador en repòs pot arribar a consumir fins a un 60% del que consumiria a màxima capacitat [36, 37, 38], el que posa de manifest la importància de maximitzar l'ús de les infraestructures computacionals per minimitzar el malbaratament energètic.

No obstant, la compartició de recursos du associada inconvenients. El principal problema ve donat per la variabilitat de les prestacions degut a que múltiples programes pretenen emprar simultàniament els recursos del maquinari. Aquest efecte es coneix com a *veí sorollós*, de l'anglès *noisy neighbour* [39], i es manifesta com una "inexplicable" degradació de les prestacions del sistema.

Més encara, dita degradació pot arribar a produir la pèrdua de connexió i errors en transaccions [40]. La magnitud de l'efecte dependrà en gran mesura dels recursos que empre cada programa, ja siga CPU, RAM, escriptures i lectures en disc o connexions de xarxa. Tot i que l'efecte pot ser mitigat repartint estratègicament les execucions d'aplicacions en el maquinari disponible a la infraestructura, per exemple en *clusters* compartits o entorns de computació en graella on especifiquem les necessitats del programa a executar, aquesta solució no sempre és una opció. L'exemple més clar el trobem a entorns de computació en el Núvol, i, amb major grau, en proveïdors públics. El motiu es deu a que els proveïdors públics empren una estratègia de repartició de les aplicacions amb l'objectiu d'arribar a un balanç entre minimitzar el cost energètic i acomplir l'acord de nivell de servei [30]. No obstant, aquest balanç no assegura l'estabilitat de les prestacions. Actualment es desenvolupen mètodes més precisos per arribar a un balanç entre qualitat de servei, cost energètic i l'estabilitat de les prestacions, com per exemple el presentat per Hieu et al. [41], que analitza no sols el consum de la CPU, sinó també d'altres recursos, com la memòria, per a migrar les MV d'acord a prediccions. Malauradament, tot i els esforços de treballs recents, es continuen mesurant fortes variacions en les prestacions dels serveis de proveïdors de computació en el Núvol. Aquestes variacions van ser mesurades ja al 2010 al treball de Schad et al. [42], on realitzaren un estudi sobre la variabilitat de les prestacions en el servei EC2 de Amazon Web Services (AWS). L'aplicació a analitzar consistia en

un processament basat en el model *MapReduce* executat sobre un *cluster* amb 50 nodes. Els resultats reportats, mostren variacions de les prestacions d'un 11%, un ordre de magnitud superiors a les registrades en una infraestructura local. A més, realitzaren mesuraments més fins amb aplicacions específiques per tal de determinar les variacions en cada component del maquinari, trobant variacions d'un 24% en les prestacions de la CPU, d'un 20% en les lectures i escriptures en disc i d'un 19% en comunicacions a través de la xarxa. Amb un propòsit similar, Iosup et al. [43] van realitzar un estudi a llarg termini sobre dos dels proveïdors més emprats, AWS [44] i Google Cloud [45], on van detectar que les prestacions dels serveis oferits presentaven fluctuacions que seguien patrons diaris i anuals. Tot i que també detectaren períodes on les prestacions es comportaven de forma estable, les prestacions mitjanes mensuals mesurades presentaven altes variacions. Un altre exemple és el treball de Leitner i Cito [46], els quals realitzaren un estudi sobre quatre proveïdors públics de computació en el Núvol, a més d'una comparativa amb treballs previs. Al seu treball, detectaren una disminució de la heterogeneïtat del maquinari en els serveis estudiats respecte a treballs previs. A més, arriben a la conclusió de que no poden determinar un patró de variació de les prestacions en funció del temps, al contrari que al treball de Iosup. No obstant, coincideixen en que existeix una forta variabilitat de les prestacions deguda a la compartició de recursos i que aquesta depèn de forma no trivial tant de l'hora del dia, el dia de la setmana o festivitats, com del proveïdor emprat. Si s'estudien treballs més recents, aquests continuen mostrant variacions en les prestacions en proveïdors de computació en el Núvol [47], i diferències notables en dites fluctuacions entre proveïdors diferents [39]. Cal remarcar, com ja s'ha dit, que els problemes ocasionats per compartir maquinari no són exclusius de proveïdors de còmput en el núvol. De fet, es poden trobar a la literatura nombrosos estudis d'aquest efecte sobre processadors amb múltiples cors [48], [49], [50].

Fa palès, per tant, que la compartició de recursos afecta negativament les prestacions del maquinari. Més encara, donat que en un entorn altament compartit, com pot ser un entorn de computació en el Núvol, l'anàlisi i adequació dels recursos compartits és, en general, complicat i no es realitza de forma eficient avui en dia, la variabilitat pot arribar a ser completament impredictible. Si, a més, la infraestructura a emprar presenta un maquinari heterogeni, és a dir, està constituïda per diferent tipus de components de maquinari, com diferents tipus de processador, el problema s'agreuja. Aquesta variabilitat és especialment problemàtica en aplicacions paral·leles amb un alt cost computacional, on usualment el fil o procés més lent determina la velocitat de l'execució global, tal i com s'ha discutit a la secció 2. Per tal de mitigar el problema, és necessari recórrer a tècniques de balanceig de càrrega, les quals discutirem a la secció següent.

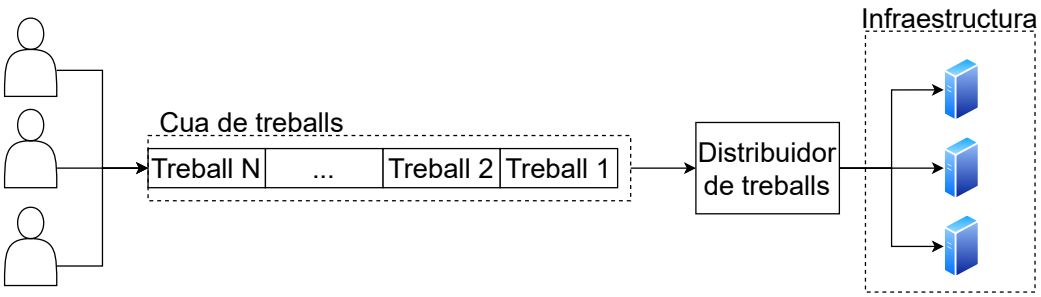

Figura I.9: Esquema de *middleware* per donar accés a múltiples usuaris a una mateixa infraestructura compartida.

3.3 Balanceig de càrrega

Per mitigar les fluctuacions de les prestacions de les infraestructures computacionals, es poden emprar diferents estratègies. D'una banda, si es té control sobre la infraestructura on s'executen les aplicacions, es pot distribuir l'execució de tasques estudiant els requeriments i ús de recursos particulars de cadascuna, intentant minimitzar així la competició per l'ús de recursos alhora que es maximitza l'eficiència de la infraestructura. Típicament, açò s'aconsegueix per mitjà d'una aplicació o *middleware* que s'encarrega de gestionar els recursos d'una o més infraestructures connectades entre si, tal i com s'esquematitza a la Figura I.9.

Probablement el cas més conegut siga la computació en graella mitjançant Globus [51], però també s'han desenvolupat altres *middlewares* de caràcter general com HTCondor [52, 53], Slurm [54] o Torque [55], o per aplicacions específiques [56], [57]. Aquesta metodologia s'estén també a les tècniques per a allotjar MVs en entorns de computació en el núvol, tema que, donat el gran consum energètic de dits proveïdors, està sent molt estudiat en els darrers anys [58], [59], [60], [61].

No obstant, una estratègia per allotjar les execucions a una infraestructura compartida no eliminarà completament les fluctuacions en les prestacions, ja que continuarà compartint-se els recursos del maquinari i cada tasca pot emprar els recursos de maquinari en major o menor mesura en instants de temps diferents. Més encara, tot i que s'executara un únic procés a cada node de la infraestructura, si aquesta és heterogènia, una aplicació paral·lela mostrarà diferències en les prestacions de cada procés involucrat. Per tant, és necessari emprar tècniques a l'interior de la pròpia aplicació per tal de distribuir el treball en funció de les prestacions individuals de cada maquinari.

Les tècniques amb aquest objectiu són conegudes com tècniques de balanceig de càrrega, les quals s'encarreguen de repartir el treball a processar de forma eficient entre els elements de còmput, comunament anomenats treballadors. Depenent del tipus d'aplicació, s'empren diferents tipus de balanceig de càrrega. Si més

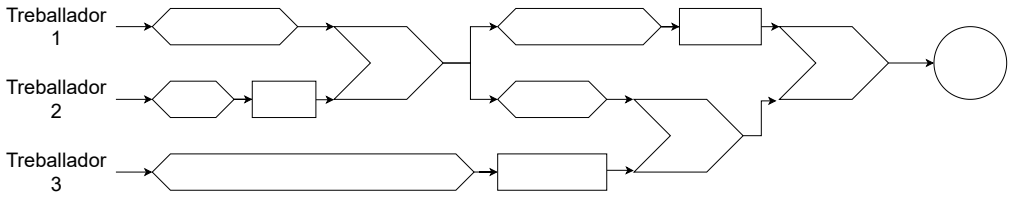

Figura I.10: Esquema de l'evolució d'una execució d'una aplicació paral·lela exemplificada amb tres treballadors. Els hexàgons representen segments de còmput independent, els quadrats temps d'espera, les figures on s'uneixen els camins de dos treballadors representen un punt de sincronització i el cercle l'obtenció del resultat final.

però, al present document ens centrarem en aquells dissenyats per a balancejar aplicacions paral·leles, les quals s'empren comunament per al càlcul científic i solen requerir llargs temps de còmput. El funcionament de les aplicacions paral·leles s'ha esquematitzat a la Figura I.10. Com vegem, l'aplicació es divideix en segments de processament independents (representats per hexàgons), segments d'espera (quadrats) i punts de sincronització on s'uneixen els camins de dos o més treballadors per compartir informació necessària abans de continuar avançant en el seu càlcul local. Donat que el càlcul local d'un o més treballadors no pot avançar sense la informació compartida als punts de sincronització, una distribució de la càrrega no adequada augmentarà els temps d'espera i, per tant, el temps global de còmput. Segons el nombre de punts de sincronització i, per tant, el requeriment de comunicacions entre processos i/o fils, és a dir, segons el seu acoblament, es poden classificar les aplicacions en dos grans grups. Aquests corresponen a aplicacions paral·leles altament acoblades i feblement acoblades, les característiques de les quals discutirem a continuació.

D'una banda estan les aplicacions paral·leles altament acoblades. En aquestes, es necessita comunicació constant entre els treballadors involucrats per tal de dur a terme el còmput, arribant a representar una part no menyspreable del temps de còmput total. Per tant, com hem vist, un mal balanceig de la càrrega provocarà que els treballadors paren de processar mentre esperen l'arribada de resultats parcials, malbaratant així els recursos de còmput. A més, unes baixes prestacions en la transferència de dades augmentarà el temps de les comunicacions, alentint també el procés.

Aleshores, aquest tipus d'aplicacions requereixen que la infraestructura on s'executen compte amb una xarxa d'interconnexió ràpida per tal de minimitzar els temps de comunicació. Malauradament, durant el desenvolupament d'aquesta tesi i als serveis estàndard proporcionats pels proveïdors públics de computació en el Núvol, la interconnexió entre nodes no presenta unes prestacions estables ni

competitives front a les característiques d'una infraestructura dedicada. Així ho constatà Keith et al. [62], on determinen que, amb proves sobre diferents aplicacions científiques en el moment de publicació, una infraestructura virtualitzada sobre el servici EC2 de AWS és de l'ordre de sis vegades més lenta que un *cluster* Linux estàndard i vint vegades més lenta que una infraestructura puntera especialitzada. Segons l'estudi, el principal motiu és la diferència de prestacions en les comunicacions i l'alta variabilitat de les prestacions en els comunicacions del servei EC2, degudes tant al compartiment de recursos, com a la virtualització i a les heterogeneïtats del maquinari subjacent. A treballs més recents, com el de Emeras et al. [63], com a molts altres, no es posa en dubte la diferència de prestacions entre una infraestructura dedicada i una virtualitzada, sinó que s'estudia des d'un punt de vista econòmic. La conclusió al comparar amb el servici EC2 de AWS és que, generalment, compensa mantenir una infraestructura local dedicada per al còmput d'aplicacions altament acoblades. Un resultat similar s'obtingué a l'estudi de Roloff et al. [64], on comparen les prestacions tant en AWS com en Microsoft Azure [65], amb una infraestructura no virtualitzada. Com a resultat, conclouen que el coll de botella en ambdós proveïdors públics està causat per les interconnexions entre nodes. Més encara, donat que contractar una instància més potent no sempre millora les prestacions de la xarxa, arriben a la conclusió que instancies més barates poden obtindre millors prestacions en relació qualitat preu. Aquest efecte també s'ha mesurat a entorns virtualitzats desplegats a infraestructures locals, com al treball de Abdallah i Ahmed [66].

En conclusió, tant els articles citats com molts altres que es poden trobar a la literatura, defenen que, durant el temps del desenvolupament de la tesi del present document, una plataforma local especialitzada per a executar aplicacions paral·leles amb acoblament alt supera tant en eficiència com en costos a un entorn virtualitzat. No obstant, aquest resultat dependrà del volum d'ús de la infraestructura, tal i com conclouen a l'estudi d'Alfonso et al. [67]. A més cal remarcar que s'estan millorant les interconnexions entre nodes en alguns proveïdors públics, com han mostrat recentment Maliszewski et al. [68] per al servei *Accelerated Networking* de Microsoft Azure. És possible que futures millores facen competitiva l'execució d'aplicacions altament acoblades en entorns de computació en el núvol.

D'altra banda, el cas d'aplicacions paral·leles amb un acoblament feble és molt diferent. Entendrem que, en aquestes, el temps invertit en comunicacions i punts de sincronització és menyspreable front al temps de còmput total o inexistent, pel que una degradació en les prestacions de les connexions no tindrà una influència notòria. No obstant, les fluctuacions en les prestacions d'altres components, com CPU o accessos a disc, sí influiran, en general, de forma significativa al temps de còmput, ja que si es desfacen els temps de processament entre treballadors, quan un arribe a un punt de sincronització, haurà d'esperar que el processament

de la resta de treballadors finalitze, tal i com s'ha esquematitzat a la Figura I.10. Per exemplificar-ho, suposem el cas més senzill, on tot el procés conté un únic punt de sincronització final per reunir els resultats parcials de n treballadors i a tots ells se'ls ha assignat la mateixa càrrega de treball. Si més no, considerem que cadascun té una velocitat constant de processament diferent s_i. En aquest escenari, menyspreant el temps de comunicació, tots els treballadors que arriben al punt de sincronització hauran d'esperar a que el treballador més lent finalitze el seu còmput local. Per tant, la velocitat de processament global s_g vindrà fixada pel treballador més lent, independentment de quina siga la velocitat de processament de la resta de treballadors,

$$s_g = min\left(s_0, s_1, s_2, ..., s_{n-1}\right). \tag{I.1}$$

Les infraestructures amb maquinari heterogeni estan molt ben representades per l'escenari anterior. Per simplificar, suposem que dita infraestructura no es comparteix i que les capacitats de còmput no fluctuen. En aquest cas, la forma més simple de dividir la càrrega entre els treballadors disponibles seria de forma proporcional a les prestacions relatives de cadascun. Sent la càrrega total a computar c_t i la velocitat de processament total del sistema s_t definida com la suma de les velocitats de tots els treballadors,

$$s_t = \sum s_i \tag{I.2}$$

la càrrega a assignar a cada treballador (c_i) vindria determinada per,

$$c_i = c_t \frac{s_i}{s_t} \tag{I.3}$$

Aquest tipus d'algorismes on la càrrega es distribueix una única vegada segons les capacitats de cada treballador es coneixen con algorismes de balanceig estàtics. El principal avantatge dels algorismes estàtics és el baix, o inexistent, sobrecost sobre el temps de còmput total de l'aplicació, ja que únicament requereixen una mesura de les prestacions de cada treballador, la qual es pot reutilitzar entre execucions d'una mateixa aplicació. A part, el temps requerit per distribuir la càrrega serà, en general, menyspreable front al temps de còmput total. No obstant, com a inconvenients, cal destacar que és necessari poder modelitzar o mesurar el cost computacional de l'aplicació, tasca que no sempre és trivial. A algunes aplicacions pot resultar molt complex determinar de forma precisa la càrrega de treball a executar per cada treballador, per exemple en simulacions de dinàmica molecular [69], on dites molècules es distribueixen de forma no uniforme per l'espai, i, per conseqüent, també la càrrega de treball. És més, les molècules es mouen durant la simulació, el que provoca canvis en la distribució de la càrrega durant l'execució.

A més dels problemes de càlcul de la càrrega, també s'ha de tenir en compte que els algorismes estàtics no seran capaços d'adaptar-se a les fluctuacions de les prestacions en infraestructures compartides. Però tot i els seus inconvenients, en certes infraestructures i aplicacions, els algorismes estàtics continuen emprant-se degut al baix sobrecost introduït. Per exemple, s'han adoptat algorismes de balanceig estàtic en llibreries de càlcul lineal com *ScaLAPACK* [70] i la *Heterogeneous PBLAS* [71] o s'ofereixen de forma opcional en la llibreria per a aplicacions paral·leles Maat [72]. Altres treballs estudien com realitzar la distribució estàtica de la càrrega de forma eficient, com el treball de Vilches et al. [73], que empra una metodologia basada en un model logarítmic per a distribuir la càrrega en sistemes combinats CPU-GPU.

Per tal d'afrontar els desavantatges dels algorismes estàtics, l'alternativa consisteix en emprar algorismes de balanceig dinàmics, és a dir, la càrrega es distribueix durant l'execució de l'aplicació. El balanceig continu durant l'execució permet adaptar la càrrega de cada treballador a les possibles fluctuacions, tant degudes a la no uniformitat de la càrrega computacional de l'aplicació com a les ocasionades per heterogeneïtats i compartició dels recursos de maquinari. Donat el benefici d'aquest mètode, s'han desenvolupat nombroses llibreries que empren balanceig dinàmic, com Charm++ [74], la presentada per Rodríguez-Gonzalo et al. [75] amb l'objectiu de maximitzar el rendiment energètic del còmput, o el treball de Garcia-Gasulla et al. [76], centrat en tècniques d'elements finits. No obstant, el preu a pagar és la necessitat d'afegir comunicació entre els treballadors per tal de recopilar la informació necessària per a realitzar el balanceig de la càrrega. Al cas d'aplicacions fortament acoblades, la informació compartida pels treballadors durant el càlcul sol ser, en general, significativament major que la requerida per balancejar la càrrega. Considerant a més que este tipus d'aplicació empra una comunicació freqüent entre els treballadors, la informació del balanceig es pot incloure als missatges, minimitzant així el sobrecost. Malauradament, els treballadors de les aplicacions dèbilment acoblades hauran de comunicar-se expressament per dur a terme el balanceig, produint un sobrecost degut a les comunicacions i punts de sincronització addicionals. Donat que, com s'ha justificat abans, l'execució d'aplicacions fortament acoblades en entorns de computació en el núvol presentava forts inconvenients durant el desenvolupament de la tesi, els treballs presentats a aquest document es centren al segon tipus d'aplicacions, les feblement acoblades.

Un altra alternativa consisteix en utilitzar una aproximació híbrida. Probablement, la més emprada consisteix en dividir el procés en tasques que són distribuïdes entre tots els recursos de còmput disponibles i usualment s'organitzen emprant una estructura GAD. En aquest cas, el punt a estudiar és la correcta distribució de les tasques, tenint en compte el cost computacional de cadascuna, la dependència entre tasques, els requeriments de maquinari, les prestacions de cada recurs, etc.

Tenint els factors pertinents en compte, cada tasca s'assigna al recurs adient, de forma similar al cas estàtic. No obstant, les tasques tenen una duració menor que el procés complet, pel qual s'espera que les prestacions del recurs romanguen constants durant el seu processament. Com avantatges cal destacar que no es requereix una monitorització constant amb comunicacions durant l'execució de les tasques, únicament a l'inici, per enviar la informació d'entrada, i en completar-se per recuperar els resultats i possibles mètriques per al balanceig. Donat que es modela el cost de cada tasca en la que s'ha subdividit el proces complet, el càlcul del cost computacional pot resultar més senzill, alhora que es tenen en compte les heterogeneïtats en el cost computacional de l'aplicació. No obstant, si les prestacions del recurs assignat canvien significativament durant l'execució de la tasca, les prediccions del balanceig no seran precises i podria provocar temps d'espera no menyspreables, malbaratant així els recursos. Més encara, cal remarcar que, en general, cada tasca s'executarà a un maquinari distint, el que pot provocar costos addicionals per transferència de dades, depenent del tipus d'aplicació. Com avantatge afegit, aquest model de balanceig permet definir un flux de treball comú per a diferents aplicacions, facilitant la seua integració. Com s'ha mencionat, molts d'ells empren GAD per representar el flux de l'aplicació, on els nodes representen tasques a computar, i les unions entre nodes comunicació per compartir dades i resultats, és a dir, dependències. A més, s'ha estudiat com adaptar aquesta metodologia de forma eficient a entorns de computació en el núvol, com el treball de Weiling et al. [77], on a més de caracteritzar les tasques es recull informació sobre la variació de les prestacions dels recursos per adaptar la distribució, o, en la mateixa línia, el de Chunlin et al. [78].

4 Reproductibilitat

Una base fonamental de l'avanç científic és la reproductibilitat, ja que és necessari que els resultats de la investigació puguen ser corroborats i emprats per altres investigadors independents. Malauradament, una enquesta realitzada a 2016 per Monya Baker [6] revela dades preocupants sobre la capacitat de reproductibilitat de les publicacions científiques. Aquesta enquesta realitzada a més de 1500 investigadors, conclou que més del 70% dels investigadors no han sigut capaços de reproduir els resultats d'algun experiment científic. Més encara, més de la meitat no eren capaços de reproduir els seus propis experiments. Per aquests motius, el 90% dels enquestats reconeixien que realment existeix una crisi en la reproductibilitat dels experiments científics.

Cal remarcar la diferència entre reproduir i replicar. La reproductibilitat computacional es pot definir com la capacitat de reproduir el mateix resultat donant un mateix conjunt de dades, metodologia i codi d'anàlisis. Al contrari, per repli-

car un resultat, un o més investigadors independents desenvoluparan el seu propi anàlisis i, possiblement, mesuraran un nou conjunt de dades per arribar a una conclusió compatible amb el primer resultat publicat.

La reproductibilitat és crucial per a que un resultat puga ser acceptat per la comunitat científica, ja que, del contrari, les conclusions presentades no podran ser degudament validades. Per tal de poder reproduir un resultat, es requerirà, per part de l'autor, informació detallada sobre les dades emprades, la metodologia, els codis emprats i l'entorn computacional. No obstant, l'estudi presentat per Hothorn i Leisch [79] conclou que, en el camp de la bioinformàtica, més del 80% dels manuscrits no proporcionen les versions del programari emprat. Donat el gran nombre de llibreries, codis d'anàlisi, simulació i processament de dades disponible a hui en dia, el coneixement detallat de l'entorn de computació és necessari per poder reproduir les conclusions d'una publicació científica. De fet, canviar de versió un únic component requerit per al processament pot resultar en incompatibilitats que impossibiliten l'execució o afecten al resultat final. Un exemple molt il·lustratiu el troben en l'entorn Python, on per a la versió 2, l'operador / s'interpreta com una divisió entre enters, mentre que a la versió 3 s'interpreta en coma flotant. Per tant, el resultat de l'operació 1/2 executada en Python 2 i 3 serà 0 i 0.5 respectivament.

Una de les formes més senzilles, avui en dia, de proporcionar l'entorn de computació per facilitar la reproductibilitat dels resultats científics és la virtualització i l'ús de contenidors [80]. A ambdós casos es pot proporcionar l'entorn complet on executar el processament de l'experiment amb totes les dependències i programes necessaris amb les versions pertinents. Inclús, es poden emprar de forma combinada per proporcionar un entorn virtualitzat on executar diferents contenidors. També es poden emprar serveis que ofereixen entorns preconfigurats per a realitzar execucions amb llenguatges de programació concrets, com runmycode [81], Code Ocean [82], Jupyter [83] o els propis entorns *serverless* discutits a la secció 3.1. El preu a pagar en aquest tipus d'entorns, es que solen tindre limitacions i molts no són aptes per a execucions amb un alt cost computacional o que requerisquen paral·lelisme.

D'altra banda, aprofitant el recurs dels contenidors, diferents treballs han presentat dissenys de fluxos de treball basats en aquesta tecnologia amb l'objectiu de proporcionar una metodologia reproduïble per als investigadors, com és el cas de *continuous analysis* [84], que empra Docker com a motor per als contenidors, i REANA [85], que és compatible tant amb Docker com Singularity [86]. Com que el flux de treball es basa amb l'execució de contenidors, els resultats seran reproduïbles en qualsevol maquinari compatible, sense importar el programari del que es dispose, ni invertir esforç en configuracions de dependències.

Com hem vist, existeixen tècniques per encapsular entorns d'execució i eines que simplifiquen el seu ús als investigadors. No obstant, si l'experimentació re-

quereix una infraestructura més complexa que un únic node per executar-se, com diversos nodes interconnectats, caldrà considerar, a més, la configuració d'aquesta. De nou, donat que difícilment investigadors diferents disposaran exactament del mateix tipus d'infraestructura amb la mateixa configuració, l'opció més adequada per a que aquesta puga ser reproduïda és mitjançant la virtualització, és a dir, desplegar la infraestructura requerida per la experimentació a un proveïdor de computació en el Núvol, ja siga públic o privat. Malauradament, en general, cada proveïdor utilitza un format diferent per descriure la infraestructura a desplegar, el que dificulta la portabilitat entre proveïdors. Per aquest motiu s'han desenvolupat diferents llenguatges per definir la configuració d'infraestructures de forma independent al proveïdor, amb l'objectiu de proporcionar una capa d'abstracció i compatibilitat entre ells, com Ansible [87] o *Resource and Application Description Language (RADL)* [88]. Al seu torn, és necessària una eina que siga capaç d'interpretar aquests formats i traslladar les descripcions d'infraestructura d'acord amb el proveïdor seleccionat. Afortunadament, aquestes eines també existeixen i permeten el desplegament d'una mateixa infraestructura definida una única vegada en múltiples proveïdors, com és el cas del *Infrastructure Manager (IM)* [89] o *Terraform* [90].

Si recopilem les eines esmentades, en la pràctica, els investigadors disposen tant de mecanismes per encapsular els entorns i dependències del còmput, com la descripció de la configuració de la infraestructura necessària. No obstant, combinar ambdues no és trivial i requereix un coneixement avançat per part del investigador, el que pot dificultar adoptar aquestes tècniques. Per tal de fer-les més accessibles als investigadors, serà convenient incloure tècniques de desplegament reproduïble d'infraestructura a entorns àmpliament emprats pels investigadors. Entre tots els disponibles, ens centrarem en l'entorn Jupyter [83] degut a la gran popularitat de la que gaudeix dintre de la comunitat científica i a la capacitat d'ampliar la seua funcionalitat.

4.1 Jupyter

Jupyter [83] és una aplicació web de codi obert basada en la creació de documents anomenats *notebooks*. Els documents, contenen tant text, codi executable, com contingut multimèdia i altres elements interactius, el que permet documentar el flux de treball d'una experimentació a mesura que s'executa. Per aconseguir-ho, el document es divideix en cel·les, a les quals se'ls pot introduir una o més línies de text o codi executable. Al segon cas, el llenguatge de programació emprat en la cel·la dependrà del kernel emprat, que no és més que el component de Jupyter que s'encarrega d'interpretar i executar el codi. A la Figura I.11 es mostra un exemple

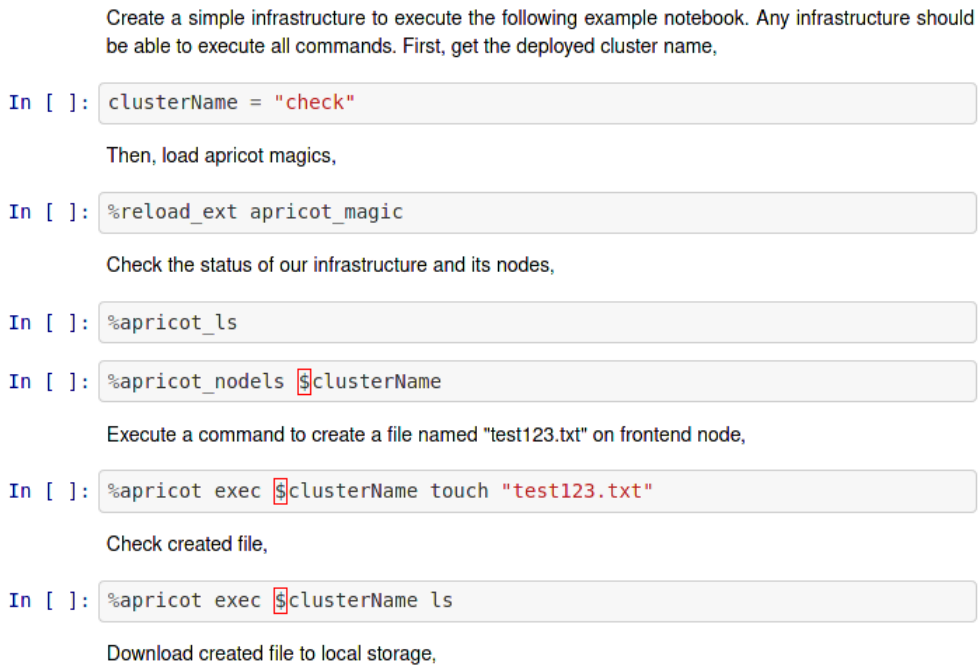

Figura I.11: Exemple d'estructura d'un *notebook* de Jupyter extret del treball d'*Apricot* presentat a aquest document.

de *notebook* extret del repositori del treball *Apricot*[1], presentat a aquest document i discutit al capítol III.

Com vegem, els *notebooks* poden emprar-se com a documents on tant la documentació com l'execució de l'experimentació es troba intercalada, permetent reproduir els resultats simplement seguint els passos i codi inclòs al document.

L'arquitectura de l'entorn Jupyter s'esquematitza a la Figura I.12. L'usuari interacciona a través de l'explorador amb el servidor de *notebooks*, que permet modificar el *notebook* i administra les comunicacions entre els diferents components. D'altra banda, els *kernels* s'encarreguen d'executar els segments de codi continguts al *notebook* i tornar el resultat al servidor, que serà mostrat a la cel·la on s'haja executat. Com s'ha dit, el llenguatge de programació interpretat dependrà del *kernel* seleccionat.

Un dels principals avantatges de l'entorn Jupyter és la gran capacitat de personalització, ja que permet estendre la seua funcionalitat de forma senzilla creant *plugins* i funcions màgiques (de l'anglès *magic functions*). També permet crear nous intèrprets per a suportar altres llenguatges a partir de la creació de kernels.

En primer lloc, les funcions màgiques no són més que funcions que són interpre-

[1] `APRICOT:https://github.com/grycap/apricot`

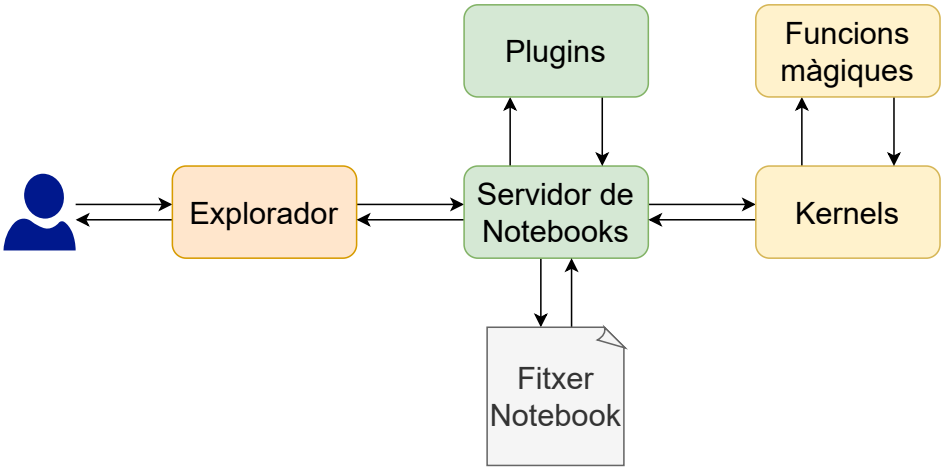

Figura I.12: Esquema de l'arquitectura emprada a Jupyter.

tades directament pel kernel de IPython però que poden ser activades a qualsevol kernel si el desenvolupador ho permet. Per tant, poden emprar-se en *notebooks* independentment del llenguatge seleccionat. Aquestes, es defineixen en Python i poden ser importades directament en el *notebook* mitjançant l'expressió següent,

```
%reload_ext nomFunc
```

que no és més que una funció màgica present per defecte a Jupyter. A l'expressió anterior, *nomFunc* correspondrà al nom assignat al paquet de funcions màgiques implementades. Prèviament, s'haurà d'instal·lar el fitxer amb la definició de les funcions mitjançant la utilitat *pip* de Python.

D'altra banda, els *plugins* són components que s'executen a través del servidor web de *notebooks*. Es poden desenvolupar amb javascript per programar interaccions entre l'usuari i el servidor, crear interfícies gràfiques, o simplement canviar l'estil de la interfície mitjançant codi CSS. A més, a través d'ells es poden executar funcions directament al kernel de Jupyter com si s'executara un segment de codi al propi *notebook*, fent-los molt versàtils.

La facilitat i eines que proporciona per a la reproductibilitat, així com la gran capacitat d'extensió, converteix Jupyter en un entorn molt favorable per a l'experimentació científica reproduïble, com així ho constaten diferents estudis [91] [92] i treballs que empren aquest entorn, tant en bioinformàtica [93], com en educació [94] entre altres camps.

Capítol II

Models de cost

Per tal d'elegir tant el model de desplegament com la configuració d'aquest de forma òptima, és necessari efectuar un estudi dels costos associats per arribar a un compromís entre velocitat, eficiència i costos. En aquest capítol, es desenvoluparan matemàticament mètodes per estimar el cost associat dels models de computació i desplegament emprats durant el desenvolupament dels treballs presentats.

1 *Serverless*

Un punt a considerar en les execucions *serverless* és l'eficiència en quant al cost del còmput, ja que, depenent de la càrrega de treball i el temps d'execució, pot ser més eficient realitzar el còmput en una MV convencional que a un entorn *serverless*. Per a cada aplicació, és convenient realitzar un anàlisis com el presentat per Mahajan et al. [95], la nomenclatura emprada pel qual es mostra a la Taula II.1. En aquest, es plantegen com minimitzar el cost en una execució mixta entre MV i computació *serverless*. Per modelitzar el cost, al treball de Mahajan et al. defineixen una sèrie de paràmetres, els quals es descriuran a continuació. En primer lloc, es defineix la capacitat de processament de treballs per unitat de temps, tant d'una MV (μ_v), com el d'una instància *serverless* (μ_s). En segon lloc, defineix el temps mitjà requerit per computar un treball per una MV i una instància *serverless* com $1/\mu_v$ i $1/\mu_s$ respectivament. Respecte a la càrrega de treball, considerada en nombre de treballs entrants per unitat de temps, es defineix com λ, de les quals $s(\lambda)$ s'assignaran a instàncies *serverless*. Per tant, s'acomplirà la restricció $s(\lambda) \leq \lambda$. Finalment, el nombre de MV llogades pel client ve denotat per $v(\lambda)$. A més, es considera que cada MV ha de mantindre un nivell d'utilització menor de ρ_t per tal de processar els treballs en un temps acceptable. Amb aquests paràmetres, el cost en funció de la càrrega de treball $c(\lambda)$ el modelitzen com,

Descripció	**Símbol**
Capacitat de processament per unitat de temps (funció *serverless* s)	μ_s
Capacitat de processament per unitat de temps (MV v)	μ_v
Treballs entrants per unitat de temps	λ
Cost monetari	$c(\lambda)$
Treballs per unitat de temps assignats a *serverless*	$s(\lambda)$
Treballs per unitat de temps assignats a MV	$v(\lambda)$
Cost monetari per unitat de temps de funció *serverless*	α_s
Cost monetari per unitat de temps de MV	α_v
Utilització màxima de MV	ρ_t
Càrrega màxima per màquina virtual	L_v
Limit d'eficiència de càrrega *serverless*	L_s

Taula II.1: Taula resum de la nomenclatura per al cas bàsic amb una única funció *serverless* i un tipus de MV i treball.

$$c(\lambda) = \alpha_s s(\lambda)/\mu_s + \alpha_v v(\lambda) \tag{II.1}$$

on els factors α_s i α_v representen el cost monetari per unitat de temps de mantindre una instància *serverless* i una MV respectivament. El terme $s(\lambda)/\mu_s$ representa el nombre mitjà d'instàncies *serverless* necessàries per a executar la càrrega $s(\lambda)$. Donat que el còmput *serverless* únicament té associat un cost mentre es tinga càrrega, serà més eficient per a càrregues suficientment baixes, ja que el cost d'una MV és independent de la càrrega que processe. No obstant, aquest cost s'igualarà al d'una MV quan,

$$s(\lambda) = \alpha_v \mu_s/\alpha_s \equiv L_s \tag{II.2}$$

Tenint en compte el factor màxim d'utilització d'una MV (ρ_t), podem definir la càrrega màxima que pot acceptar una MV com $L_v = \rho_t \mu_v$. Per tant, sempre i quan la càrrega L_s siga menor que L_v, serà més eficient mantindre una màquina virtual una vegada s'arribe a una taxa de càrrega igual a L_s. És a dir, dins de l'interval de càrrega $\lambda \in [L_s, L_v]$ s'emprarà una MV sense instancies *serverless*. Pel contrari, si $L_v \leq L_s$ mai serà més rendible llogar una MV, i el còmput es farà sempre a la infraestructura *serverless*. És fàcil comprovar que aquest comportament es repeteix a mesura que augmenta la càrrega d'entrada λ, i que els valors òptims per a $v(\lambda)$ són,

Descripció	Símbol
Conjunt de tipus de treballs	Λ
Nombre de tipus de treballs	n_λ
Nombre de tipus de funcions *serverless*	n_s
Nombre de tipus de MV	n_v
Capacitat de processament de treballs j respecte el primer (*serverless*)	β_s^{ij}
Capacitat de processament de treballs j respecte el primer (MV)	β_v^{ij}
Càrrega total suportada per unitat de temps per una MV	T_v^i
Factor d'eficiència de cost (MV)	E_v^i

Taula II.2: Taula resum de la nomenclatura per al cas generalitzat amb múltiples tipus de funció *serverless*, MV i treballs. Per simplicitat no s'han repetit els descrits a la Taula II.1, els quals són equivalents però se'ls afig un índex de tipus de funció/instància i/o treball. En tots ells, l'índex i representa el tipus de funció/instància.

$$v(\lambda) = \begin{cases} 0, & L_v \leq L_s \\ \left\lceil \frac{\lambda}{L_v} \right\rceil, & \lambda - [\lambda/L_v]\, L_v \leq L_s \\ 1 + \left\lceil \frac{\lambda}{L_v} \right\rceil, & \textit{en altre cas} \end{cases} \tag{II.3}$$

on el factor $[\lambda/L_v]$ s'ha d'interpretar com la part entera del quocient. En canvi, per a $s(\lambda)$ s'obtenen els valors òptims següents,

$$s(\lambda) = \begin{cases} \lambda, & L_v \leq L_s \\ \lambda - \left\lceil \frac{\lambda}{L_v} \right\rceil L_v, & \lambda - [\lambda/L_v]\, L_v \leq L_s \\ 0, & \textit{en altre cas} \end{cases} \tag{II.4}$$

Tot i que aquest anàlisi [95] està fet per a un únic tipus d'instància *serverless* i MV es pot generalitzar a per a múltiples tipus. Este estudi és interessant degut a que un mateix proveïdor ens permet seleccionar múltiples configuracions de les instàncies. Més encara, una mateixa aplicació pot presentar temps de còmput molt dispars depenent dels paràmetres d'entrada, pel que es poden considerar diferents càrregues de treballs. Per tant, a continuació generalitzarem el treball de [95] per a tenir en compte aquestes consideracions. La nomenclatura emprada en la nomenada generalització ve resumida en la Taula II.2.

En primer lloc, definim el conjunt de n_λ tipus de treballs diferents com $\Lambda \equiv [\lambda_1, \lambda_2, \ldots, \lambda_{n_\lambda}]$. A més, considerarem que disposem de n_s tipus d'instàncies *serverless* i de n_v tipus de MV. Reescrivim per tant l'equació II.1 com,

$$c(\Lambda) = \sum_{i}^{n_s} \sum_{j}^{n_\lambda} \alpha_s^i s^{ij}(\Lambda)/\mu_s^{ij} + \sum_{i}^{n_v} \alpha_v^i v^i(\Lambda) \tag{II.5}$$

on l'index i representa el tipus d'instància, *serverless* o MVs depenent del sumand, i l'index j el tipus de treball. A més, interpretem $s^{ij}(\Lambda)$ com el nombre d'instàncies *serverless* de tipus i emprades per a processar treballs de tipus j. Al seu torn, els coeficients $1/\mu^{ij}$ representen el temps mitjà de processament d'un treball de tipus j a una instància de tipus i. Com és d'esperar, el cost de la màquina virtual no depèn del treball que s'execute en aquesta, però si com modelem la seua capacitat d'acceptar càrrega de treball.

Per tal de calcular una quota a la càrrega de treball que pot suportar una MV de tipus i en funció dels coeficients μ_v^{ij} dels diferents tipus de treball,

$$\mu_v^i \equiv \left[\mu_v^{i1}, \mu_v^{i2}, \ldots, \mu_v^{i\lambda_n}\right] \tag{II.6}$$

els expressem en funció del coeficient corresponent al primer tipus de treball ($j = 1$),

$$\mu_v^i = \left[\mu_v^{i1}, \frac{\mu_v^{i2}}{\mu_v^{i1}}\mu_v^{i1}, \ldots, \frac{\mu_v^{i\lambda_n}}{\mu_v^{i1}}\mu_v^{i1}\right] \equiv \mu_v^{i1}\left[\beta_v^{i1}, \beta_v^{i2}, \ldots, \beta_v^{i\lambda_n}\right] \tag{II.7}$$

on $\beta_v^{i,1} = 1$ donat que es relatiu al primer tipus de treball. De forma equivalent es defineixen els coeficients $\beta_s^{i,j}$ per a les instàncies *serverless*. Amb aquesta definició, la càrrega total suportada, per unitat de temps, per una MV de tipus i (T_v^i), es pot expressar com l'equivalent a la taxa de treballs entrants de tipus 1 a partir dels factors de cost relatiu respecte al primer tipus de treball (β_v^{ij}) i la taxa de treballs de cada tipus que arriben a la MV (λ_j^i),

$$T_v^i = \sum_{j}^{n_\lambda} \lambda_j^i/\beta_v^{ij} \tag{II.8}$$

Al seu torn, al quedar en funció d'un únic valor ($\mu_v^{i,1}$), la capacitat que pot suportar una única MV, de cada tipus i, es pot expressar en funció del primer tipus de treball ($j = 1$) com,

$$L_v^i = \rho_t^i \mu_v^{i,1} \tag{II.9}$$

Amb la definició anterior, es pot calcular un factor d'eficiència de cost E_v^i, considerant que la instància es troba a màxima capacitat.

$$E_v^i = L_v^i/\alpha_v^i \tag{II.10}$$

tenint en compte que sempre s'ha d'acomplir que $T_v^i \leq L_v^i$ per tal que la MV no estiga saturada. Tornant a la part *serverless*, considerarem que cada tipus de treball s'executarà de forma òptima a un tipus d'instància determinada. Si a més, considerem que les instancies s'invoquen per a processar un treball específic i després són destruïdes, no cal considerar més d'un únic tipus d'instància *serverless* per tipus de treball. Amb aquestes suposicions, definim $s^j(\Lambda)$ com el nombre d'instàncies de tipus òptim per executar treballs de tipus j, α_s^j com el cost per unitat de temps de la instància *serverless* del tipus òptim per executar treballs de tipus j i $1/\mu_s^j$ com el temps mitjà d'execució del treball en aquest tipus d'instància. Aplicant les definicions anteriors, la funció de cost es reescriu com,

$$c(\Lambda) = \sum_{j}^{n_\lambda} \alpha_s^j s^j(\Lambda)/\mu_s^j + \sum_{i}^{n_v} \alpha_v^i v^i(\Lambda) \tag{II.11}$$

D'altra banda, suposarem que una mateixa MV pot, en general, processar múltiples treballs, pel que no s'elegirà un tipus d'instància privilegiat per a cada treball. Considerem ara el punt on els costos de l'execució en *serverless* s'equilibren amb el cost d'una MV d'un tipus determinat. El cost mitjà de processar una determinada distribució de taxa de treballs entrants, completament en l'entorn *serverless*, ve donat per,

$$c_s = \sum_{j}^{n_\lambda} \alpha_s^j \lambda_j/\mu_s^j \tag{II.12}$$

on, donat que hem forçat $v^i(\Lambda) = 0\ \forall i$, $s^j(\Lambda)$ passa a dependre únicament del treball de tipus j, podent aplicar l'equivalència de l'equació II.13.

$$s^j(\Lambda) = s^j(\lambda_j) = \lambda_j \tag{II.13}$$

Per tant, l'opció *serverless* deixarà de ser més eficient en el moment que un dels tipus de MV siga capaç d'executar la mateixa càrrega de treball (equació II.8) a un cost més baix, tal i com mostra la desigualtat de l'equació II.14 per a qualsevol tipus i de MV.

$$min(\alpha_v^i) \leq \sum_{j}^{n_\lambda} \alpha_s^j \lambda_j/\mu_s^j, \quad \forall i/L_v^i \geq \sum_{j}^{n_\lambda} \lambda_j/\beta_v^{ij} \tag{II.14}$$

Cal remarcar que en els càlculs posteriors suposarem que una MV a plena capacitat sempre serà més rendible que la càrrega equivalent processada en *serverless*, ja que, del contrari, no és necessari considerar aquest tipus de MV en el càlcul.

Si considerem una taxa de treball arbitrària, i no la mínima per a que emprar una MV siga més rendible, apareix també el problema de decidir la combinació

òptima de tipus de MV. Donat que s'ha definit l'eficiència de cada tipus d'acord amb l'equació II.10, l'opció òptima consistirà en maximitzar el tipus d'instància amb un valor major d'eficiència E_v^i, sempre i quan s'empren a màxima capacitat. Per tal d'emprar aquest resultat, hem de suposar que existeix, com a mínim, un tipus i de MV, el cost de la qual és inferior a emprar instancies *serverless* per cobrir una càrrega de treball equivalent a la seua màxima capacitat. Aquesta condició ve donada a l'equació II.15. Si no s'acomplira, emprar instancies *serverless* sempre serà més rendible.

$$\exists\, i \;/\quad \alpha_v^i < \sum_j^{n_\lambda} \alpha_s^j \lambda_j / \mu_s^j \quad \& \quad T_v^i = L_v^i \tag{II.15}$$

Amb aquesta suposició, el nombre d'instàncies necessàries del tipus més eficient, a màxima capacitat, vindrà donat per l'equació II.16,

$$v^i(\Lambda) = \frac{\sum_j^{n_\lambda} \lambda_j / \beta_v^{ij}}{L_v^i} \equiv v^E(\Lambda), \quad / \; E_v^i > E_v^l \; \forall\, l \neq i \tag{II.16}$$

on hem definit el superindex E per denotar el tipus d'instància més eficient. Els treballs restants a processar, $\Delta\Lambda$, vindran donats per l'equació II.17,

$$\Delta\Lambda = \sum_j^{n_\lambda} \lambda_j - \lambda_j^E(\Lambda) \tag{II.17}$$

on $\lambda_j^E(\Lambda)$ són el nombre de treballs de tipus j assignats a les instàncies eficients. No obstant, la carrega a afrontar per executar la resta de treballs dependrà del tipus d'instància, ja que hem suposat que, en general, tenen capacitats de còmput diferent, inclús en els coeficients β_v^{ij}. Per tant, la càrrega que ha d'afrontar una instància de tipus i per processar la taxa de treballs entrants restant, ΔT_v^i, es calcularà segons l'equació II.18.

$$\Delta T_v^i(\Lambda) = \sum_j^{n_\lambda} \frac{\lambda_j^i - \lambda_j^E(\Lambda)}{\beta_v^{ij}} \tag{II.18}$$

Amb aquest excedent calculat, es comprovarà, en primer lloc, si algun dels tipus de MV disponibles disposa d'una capacitat de còmput menor. Si s'acomplira, significarà que l'opció *serverless* ha de ser, necessàriament, menys eficient que desplegar, com a mínim, 1 MV d'aquest tipus. Per tant cal seleccionar el tipus de MV més eficient per a l'excedent, el que determinarem calculant una eficiència adaptada per a cada tipus de MV definida en l'equació II.19.

$$E_v'^i = \begin{cases} \Delta T_v^i / \alpha_s^i, & \Delta T_v^i \leq L_v^i \\ L_v^i / \alpha_s^i, & \Delta T_v^i > L_v^i \end{cases} \tag{II.19}$$

Una vegada calculada, s'afegirà una MV del tipus que maximitze $E_v'^i$. Aquest procés es repetirà de forma iterativa fins que l'excedent de treball siga nul, o fins que no siga suficient per omplir cap tipus de MV. Repetint aquest procés, l'excedent de taxa de treballs vindrà determinat per,

$$\Delta\Lambda^{'}(\Lambda) = \sum_{j}^{n_\lambda} \left(\lambda_j - \lambda_j^E(\Lambda) - \sum_{i \neq E}^{n_v} \lambda_j^i(\Lambda) \right) \tag{II.20}$$

on l'últim sumatori representa la taxa de treball assignada a instancies de tipus diferent a l'eficient (E). En aquest punt, podem assegurar que l'excés de taxa de treball, si existeix, no és suficient per mantenir una MV a màxima capacitat, pel que l'opció *serverless* podria ser més eficient. Al seu torn, este escenari és equivalent al discutit per a obtenir la condició de l'equació II.14 que determina quan una MV és més rendible que una aproximació *serverless*, pel que s'aplicarà el mateix criteri emprant únicament la distribució excedent de la taxa de treball $\Delta\Lambda'$.

Cal remarcar que durant el desenvolupament anterior s'ha suposat que qualsevol taxa de qualsevol tipus de treball, λ_j, arriba a un punt on és més rendible processar-la a una MV que a instàncies *serverless*. Si no fora el cas, l'opció més òptima per a aquest tipus de treball particular consistiria en processar-lo completament al servei *serverless* i únicament considerar la resta de treballs al mètode presentat.

A diferència del mètode presentat per Mahajan et al. [95], aquest desenvolupament permet parametritzar múltiples tipus d'instàncies, tant *serverless* com MVs. A més, es considera diferents tipus de treballs, ja que prendre un únic valor mitjà pot suposar que, aquest, presente una incertesa significativa, inclús si s'empra una única aplicació. Per exemple, en simulacions Monte-Carlo de transport de radiació, diferents simulacions poden requerir temps de còmput de l'ordre d'hores, dies, o setmanes. Més encara, efectes com les heterogeneïtats o les fluctuacions en les prestacions discutides a la secció 3.2 podrien agreujar el problema. No obstant, el preu a pagar és la necessitat de mesurar temps d'execució als diferents tipus d'instàncies disponibles per tal de calcular els paràmetres necessaris per a la selecció òptima de recursos.

En conclusió, l'entorn *serverless* mereix ser estudiat per a ser emprat en càlcul científic donat els avantatges que presenta. No obstant, si es vol minimitzar els costos, la millor opció podria ser una computació híbrida, depenent de l'aplicació i de la taxa de treballs entrants.

Descripció	**Símbol**
Temps de processament	t_p
Temps de comunicació	t_c
Quantitat de dades per subgrup de mapatge	d_m
Quantitat de dades per iteració i de reducció	d_r^i
Quantitat de dades a reduir en reducció simple	d_r^s
Quantitat de dades resultants del *MapReduce*	$d_{r_e}^s$
Quantitat de dades total a reduir	d_r^t
Cost de mapatge	$c_m(d_m)$
Cost de reducció	$c_r(d_r^i)$
Nombre de treballadors de mapatge	N_m
Nombre de treballadors que executen la reducció	N_r
Latència	l
Velocitat de transmissió de dades	s
Factor de reducció de dades per mapatge	α
Factor de reducció de dades per reducció i-èssima	β_i
Factor de reducció de dades per reducció simple	β_s
Factor màxim de reducció	β_{max}
Factor mínim de reducció	β_{min}

Taula II.3: Taula resum de la nomenclatura per a l'anàlisi *MapReduce*.

2 *MapReduce*

Per modelitzar els costos d'una execució basada en el model de computació *MapReduce*, emprarem la nomenclatura descrita a la Taula II.3. En primer lloc, modelitzem el temps de processament t_p per al paral·lelisme amb cost logarítmic com,

$$t_p = c_m(d_m) + \sum_{i=0}^{log_2(N_r)} c_r(d_r^i) \tag{II.21}$$

on d_m és la quantitat de dades per un subgrup de mapatge, c_m el cost del mapatge en funció de la quantitat de dades d_m, en unitats de temps per unitat de dades, N_r el nombre de treballadors que executen la reducció i c_r el cost de la reducció en funció de la quantitat de dades en cada iteració i de la reducció (d_r^i). Per simplicitat, s'assumeix que el cost de mapatge c_m correspon al treballador més lent i que el procés de reducció comença en acabar aquest. D'igual manera, c_r s'assumeix com el cost del treballador més lent i la següent iteració de la reducció conclourà en acabar aquest el seu processament. A més, suposarem que les dimensions de les dades a processar són, en mitjana, iguals per a tots els mapatges d_m i reduccions

d'una mateixa iteració d_r^i.

D'altra banda, considerarem que les prestacions de les comunicacions es comporten igual per a totes les combinacions de treballadors, i que aquesta es modelitza de forma lineal considerant la latència l i la velocitat de transmissió en nombre de dades per unitat de temps s,

$$t = l + d/s \tag{II.22}$$

Amb aquest model, els temps de comunicació per al cas logarítmic es poden expressar com,

$$\begin{aligned} t_c &= 2l + (d_m + d_r^0)/s + \sum_{i=1}^{log_2(N_r)} (l + d_r^i/2s) \\ t_c &= l(2 + log_2(N_r)) + \frac{1}{s}\left(d_m + d_r^0 + \frac{1}{2}\sum_{i=1}^{log_2(N_r)} d_r^i\right) \end{aligned} \tag{II.23}$$

on s'ha suposat que les dades requerides per a la primera reducció (d_r^0) es transmeten des dels treballadors de mapatge fins als de reducció. En la resta d'iteracions, en mitjana, únicament la meitat de les dades han de ser enviades, donat que l'altra meitat ja es troba al treballador de reducció que va a processar-les. Cal remarcar que s'ha considerat que tant el processament com les comunicacions s'executen de forma concurrent sense que cap treballador afecte a les prestacions de la resta. Com s'ha discutit a la secció 3.2, açò, en general, no és cert, i el sistema podria sofrir fluctuacions en les prestacions. No obstant, donat que els treballadors més lents determinen el temps d'execució total, prenent cotes inferiors i superiors de les prestacions, tant de processament com de les comunicacions, es pot obtindre un rang del temps total d'execució.

De forma equivalent es defineix el temps de processament i de comunicació per a l'execució paral·lela sense l'aproximació de reducció logarítmica,

$$t_p = c_m(d_m) + c_r(d_r^t) \tag{II.24}$$

$$t_c = l + d_m/s + l + (d_r^t)/s = 2l + (d_m + d_r^t)/s \tag{II.25}$$

on d_r^t és el conjunt total de dades a reduir.

Si reescrivim la quantitat de les dades d'entrada a reduir en la primera iteració de reduccions per un treballador (d_r^0) en funció de la quantitat de dades d'entrada d'un treballador de mapatge (d_m), la ràtio entre nombre de treballadors de mapatge i de reducció (N_m/N_r) i el factor de disminució de la quantitat de dades degut al procés de mapatge (α), obtenim la relació,

$$d_r^0 = (N_m/N_r)\alpha d_m \tag{II.26}$$

on s'ha suposat que la partició de les dades entre els treballadors de reducció es realitza de forma equitativa. Si, a més, es reescriu la quantitat de dades a reduir, per treballador, en iteracions successives de reducció en funció de la iteració anterior i un coeficient de reducció de dades β_i, s'obté la quantitat de dades a reduir en la iteració i-èssima (d_r^i) com,

$$d_r^i = 2\beta_{i-1}d_r^{i-1} = 2^i(N_m/N_r)\alpha d_m \prod_{j=0}^{i-1}\beta_j \quad \forall i > 0 \tag{II.27}$$

on s'ha multiplicat per 2 degut a que en cada iteració successiva es recullen els resultats de la reducció prèvia de dos treballadors, i el productori recorre fins a $i-1$ degut a que la reducció i-èssima no s'ha executat encara. Amb aquest resultat, es pot calcular la quantitat de dades d'eixida de cada treballador després de la etapa i-èssima de reducció ($d_{r_e}^i$) com,

$$d_{r_e}^i = \begin{cases} \beta_0(N_m/N_r)\alpha d_m, & i = 0 \\ \beta_i d_r^i = 2^i(N_m/N_r)\alpha d_m \prod_{j=0}^{i}\beta_j, & i > 0 \end{cases} \tag{II.28}$$

D'altra banda, la quantitat de dades d'entrada i resultants de la reducció simple, donat que únicament s'empra un treballador per a la reducció, venen donats per les equacions II.29 i II.30 respectivament.

$$d_r^s = \alpha N_m d_m \tag{II.29}$$

$$d_{r_e}^s = \beta_s \alpha N_m d_m \tag{II.30}$$

on β_s és el coeficient de reducció de les dades aplicat com a conseqüència de l'única reducció. Donat que ambdós metodologies han de proporcionar el mateix resultat i, per tant, la quantitat de dades finals ha de ser igual en ambdós tipus de reducció, s'ha d'acomplir la condició,

$$d_{r_e}^s = d_{r_e}^{log_2(N_r)} \tag{II.31}$$

$$\beta_s \alpha N_m d_m = 2^{log_2(N_r)}(N_m/N_r)\alpha d_m \prod_{j=0}^{log_2(N_r)}\beta_j = \alpha N_m d_m \prod_{j=0}^{log_2(N_r)}\beta_j \tag{II.32}$$

de la qual podem extraure la condició,

$$\prod_{j=0}^{log_2(N_r)} \beta_j = \beta_s \tag{II.33}$$

Reescrivint les equacions de temps anteriors amb aquestes definicions, obtenim les funcions següents per a la reducció logarítmica,

$$t_p = c_m(d_m) + c_r\left((N_m/N_r)\alpha d_m\right) + \sum_{i=1}^{log_2(N_r)} c_r\left(2^i(N_m/N_r)\alpha d_m \prod_{j=0}^{i-1}\beta_j\right) \tag{II.34}$$

$$t_c = l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{N_r}\alpha\left(1 + \frac{1}{2}\sum_{i=1}^{log_2(N_r)} 2^i \prod_{j=0}^{i-1}\beta_j\right)\right) \tag{II.35}$$

i per a la reducció simple,

$$t_p = c_m(d_m) + c_r(\alpha N_m d_m) \tag{II.36}$$

$$t_c = 2l + d_m(1 + N_m\alpha)/s \tag{II.37}$$

Malauradament, obtindre una mesura precisa i genèrica dels coeficients de reducció de dades β per a cada iteració de reducció és molt complex. No obstant, podem obtenir un rang de temps considerant un coeficient global màxim (β_{max}) i un altre mínim (β_{min}) comú a totes les iteracions. Per al cas de (β_{max}), sempre s'acomplirà la següent desigualtat,

$$\prod_{j=0}^{i-1}\beta_j < \prod_{j=0}^{i-1}\beta_{max} = (\beta_{max})^i \tag{II.38}$$

on β_{max} és el valor màxim de totes les β_j. Amb aquesta desigualtat, el temps de comunicació en el processament amb reducció logarítmica queda restringit a la cota superior següent,

$$t_c < l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{N_r}\alpha\left(1 + \frac{1}{2}\sum_{i=1}^{log_2(N_r)}(2\beta_{max})^i\right)\right)$$
$$t_c < l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{N_r}\alpha\left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2}\sum_{i=1}^{log_2(N_r)}(2\beta_{max})^i\right)\right)$$
$$t_c < l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{2N_r}\alpha\left(1 + \sum_{i=0}^{log_2(N_r)}(2\beta_{max})^i\right)\right) \quad \text{(II.39)}$$

Identificant el sumatori com una sèrie geomètrica,

$$\sum_{i=0}^{n} x^i = \frac{1 - x^{n+1}}{1 - x} \quad \text{(II.40)}$$

i suposant que s'acompleix que $2\beta_{max} \neq 1$ per a evitar el zero al denominador,

$$t_c \leq l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{2N_r}\alpha\left(1 + \frac{1 - (2\beta_{max})^{log_2(N_r)+1}}{1 - 2\beta_{max}}\right)\right)$$
$$t_c \leq l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{2N_r}\alpha\left(1 + \frac{1 - 2N_r(\beta_{max})^{log_2(N_r)+1}}{1 - 2\beta_{max}}\right)\right) \quad \text{(II.41)}$$

Noteu que un càlcul anàleg fixant totes les β_j com al valor mínim de totes elles (β_{min}), ens proporcionarà una cota inferior per al temps de comunicacions,

$$t_c \geq\ l(2 + log_2(N_r)) + \frac{d_m}{s}\left(1 + \frac{N_m}{2N_r}\alpha\left(1 + \frac{1 - 2N_r(\beta_{min})^{log_2(N_r)+1}}{1 - 2\beta_{min}}\right)\right) \quad \text{(II.42)}$$

obtenint un interval de valors possible per al temps de comunicació siga quina siga la distribució de valors per als β_j.

Per estudiar com augmenta el temps de les comunicacions, calculem el limit sobre el terme dret de la desigualtat II.41 quan el nombre de treballadors de reducció augmenta. Tenint en compte que,

$$\lim_{N_r \to \infty} (\beta_{max})^{log_2(N_r)+1} = \begin{cases} 0 & \beta_{max} < 1 \\ 1 & \beta_{max} = 1 \end{cases} \quad \text{(II.43)}$$

s'arriba a que el terme de latència és l'únic que divergeix, ja que el límit del terme de transferència ve donat per,

$$\lim_{N_r\to\infty} \frac{d_m}{s}\left(1+\alpha\left(\frac{N_m}{2N_r}+\frac{N_m/2N_r - N_m(\beta_{max})^{log_2(N_r)+1}}{1-2\beta_{max}}\right)\right) =$$
$$\lim_{N_r\to\infty} \frac{d_m}{s}\left(1+\alpha\frac{-N_m(\beta_{max})^{log_2(N_r)+1}}{1-2\beta_{max}}\right) = \begin{cases} d_m/s, & \beta_{max}<1 \\ d_m/s(1+\alpha N_m), & \beta_{max}=1 \end{cases} \tag{II.44}$$

Per tant, donat que el terme independent de la latència sempre serà menor o igual que el de l'equació II.41, el temps de comunicacions creixerà com $O(log_2(N_r))$ quan N_r augmente. El temps de processament dependrà de les funcions de cost, de les quals estudiarem un cas particular a la secció següent. L'avantatge de realitzar un estudi com el presentat, i el que es desenvoluparà a la secció següent, és que ofereix una metodologia clara per a determinar la combinació òptima del nombre de nodes de mapatge i reducció per tal d'aconseguir el compromís desitjat entre temps de processament i cost. Per aconseguir-ho, s'empren paràmetres ben definits, com el volum de dades i la velocitat de processament, el valor de la qual es pot mesurar, en mitjana, realitzant proves.

2.1 Cost lineal $O(N)$

Descripció	Símbol
Velocitat de processament de mapatge	s_m
Velocitat de processament de reducció	s_r
Quocient de velocitat de mapatge front a reducció	γ

Taula II.4: Taula resum de la nomenclatura per a l'anàlisi *MapReduce* específic del cas amb cost lineal. La nomenclatura de la Taula II.3 s'empra també en aquest cas.

Per a l'anàlisi del cas amb cost lineal, emprarem la nomenclatura de la taula II.4. Si suposem que tant c_m com c_r es comporten de forma lineal amb el nombre de dades a processar, amb una constant de proporcionalitat $1/s_m$ i $1/s_r$ respectivament, obtindrem uns temps de processament per a l'execució amb reducció logarítmica i simple donats per les equacions II.45 i II.46 respectivament.

$$\begin{aligned} t_p =& \frac{d_m}{s_m} + \alpha\frac{N_m}{N_r}\frac{d_m}{s_r} + \sum_{i=1}^{log_2(N_r)} 2^i\alpha\frac{N_m}{N_r}\frac{d_m}{s_r}\prod_{j=0}^{i-1}\beta_j \\ t_p =& \frac{d_m}{s_m}\left(1+\frac{N_m}{N_r}\alpha\gamma\left(1+\sum_{i=1}^{log_2(N_r)} 2^i\prod_{j=0}^{i-1}\beta_j\right)\right) \end{aligned} \tag{II.45}$$

$$t_p = d_m/s_m + N_m \alpha d_m/s_r = \frac{d_m}{s_m}(1 + \alpha\gamma N_m) \tag{II.46}$$

on s'ha definit $\gamma \equiv s_m/s_r$, és a dir, com el quocient de velocitats de processament entre el mapatge i la reducció. Si apliquem la mateixa desigualtat que a l'equació II.38, el temps de processament per a la reducció logarítmica queda com,

$$\begin{aligned} t_p &\leq \frac{d_m}{s_m}\left(1 + \frac{N_m}{N_r}\alpha\gamma\left(1 + \sum_{i=1}^{log_2(N_r)}(2\beta_{max})^i\right)\right) \\ t_p &\leq \frac{d_m}{s_m}\left(1 + \frac{N_m}{N_r}\alpha\gamma\sum_{i=0}^{log_2(N_r)}(2\beta_{max})^i\right) \end{aligned} \tag{II.47}$$

on, tornant a aplicar el resultat per a la suma de la sèrie geomètrica de l'equació II.40, s'obté,

$$\begin{aligned} t_p &\leq \frac{d_m}{s_m}\left(1 + \frac{N_m}{N_r}\alpha\gamma\frac{1 - (2\beta_{max})^{log_2(N_r)+1}}{1 - 2\beta_{max}}\right) \\ t_p &\leq \frac{d_m}{s_m}\left(1 + \alpha\gamma\frac{(N_m/N_r) - 2N_m(\beta_{max})^{log_2(N_r)+1}}{1 - 2\beta_{max}}\right) \end{aligned} \tag{II.48}$$

Amb un càlcul anàleg a l'emprat per a calcular el límit de l'equació II.44, s'obté,

$$\lim_{N_r\to\infty} t_p \leq \begin{cases} d_m/s_m, & \beta_{max} < 1 \\ d_m/s_m(1 + 2\alpha\gamma N_m) & \beta_{max} = 1 \end{cases} \tag{II.49}$$

el que indica que arriba un punt on l'avantatge d'augmentar el nombre de treballadors de reducció satura. Com al cas de les comunicacions, un càlcul anàleg fixant totes les β_j com a β_{min} ens proporcionarà una cota inferior per al temps de processament,

$$t_p \geq \frac{d_m}{s_m}\left(1 + \frac{N_m}{N_r}\alpha\gamma\frac{1 - (2\beta_{min})^{log_2(N_r)+1}}{1 - 2\beta_{min}}\right) \tag{II.50}$$

obtenint un interval de valors possible per al temps de processament siga quina siga la distribució de valors per als β_j.

Aquest anàlisis ens permet trobar un valor òptim del nombre de treballadors a emprar donades unes condicions específiques del tipus de *MapReduce* a aplicar. Cal remarcar que l'anàlisi depèn d'un gran nombre de paràmetres, pel que no es poden extraure conclusions generals per a qualsevol aplicació.

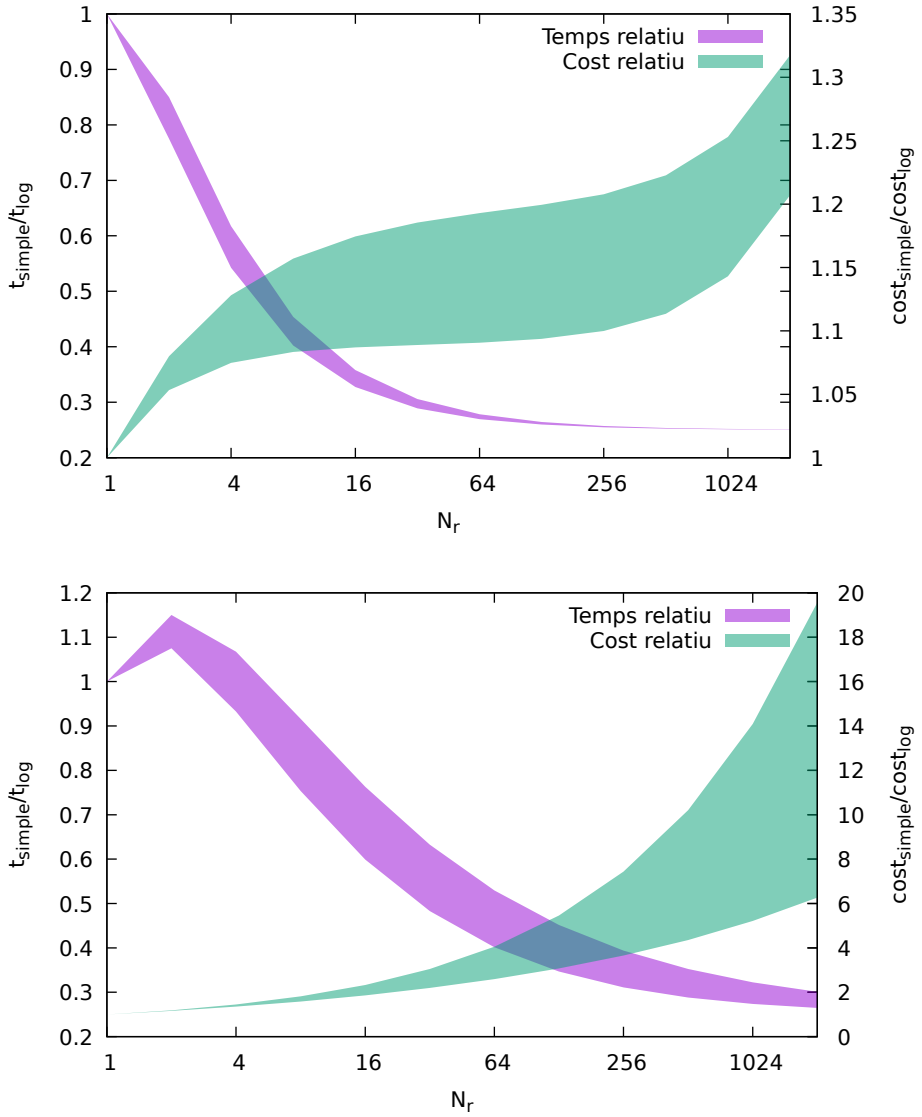

Figura II.1: Intervals de temps i cost relatiu entre l'aproximació de reducció logarítmica i simple suposant un cost lineal. Superior: $\beta_{min} = 0.2$ i $\beta_{max} = 0.3$. Inferior: $\beta_{min} = 0.6$ i $\beta_{max} = 0.8$

Per exemple, a continuació estudiem un cas particular amb 10 treballadors de mapatge ($N_m = 10$), on cadascun processa $d_m = 5GB$ de dades, amb un valor de reducció de dades pel mapatge $\alpha = 0.3$, igual velocitat per al mapatge que per a la reducció $\gamma = 1$ i una velocitat de processament per treballador de $200MB/s$. Respecte a la xarxa, s'ha fixat una latència de $20ms$ i una velocitat

de transmissió de $300Mb/s$. A la Figura II.1 s'han representat, en funció del nombre de treballadors de reducció, els intervals de temps d'execució relativa entre l'aproximació de reducció simple i logarítmica, suposant uns paràmetres de $\beta_{min} = 0.2$ i $\beta_{max} = 0.3$, i el cost relatiu, suposant que cada treballador té un cost associat de 0.5 €/h.

Amb la mateixa configuració, modificant els paràmetres β, simulem ara el cas d'una aplicació on la reducció és molt menys eficient, prenent $\beta_{min} = 0.6$ i $\beta_{max} = 0.8$. Aquesta configuració està representada a la Figura II.1 inferior.

Com vegem, en aquest cas, no sols el cost de l'aproximació logarítmica augmenta molt més ràpidament, sinó que, a més, per a un nombre reduït de treballadors de reducció és més lent que la reducció simple.

Queda patent la utilitat de realitzar un estudi previ de l'aplicació a executar amb aquest tipus de tècniques de processament, ja que, del contrari, podríem malbaratar els recursos disponibles e incrementar tant el temps com el cost del procés total, reduint així l'eficiència. Com ja s'ha dit, modelitzar els costos de processament ens permet calcular el nombre òptim de nodes de mapatge i de reducció per tal d'acomplir amb el compromís entre temps d'execució i cost desitjat.

Capítol III

Recerca científica

1 A framework and a performance assessment for serverless MapReduce on AWS Lambda

Referència: Giménez-Alventosa, V., Moltó, G., Caballer, M., 2019. A framework and a performance assessment for serverless MapReduce on AWS Lambda. Futur. Gener. Comput. Syst. 97, 259–274. https://doi.org/10.1016/j.future.2019.02.057

Categoria JCR: COMPUTER SCIENCE, THEORY & METHODS

Posició: 7/110

Quartil: Q1

Factor d'impacte (2020): 7.187

El primer treball presentat durant el desenvolupament de la tesi està enfocat en dos objectius principals. En primer lloc, el desenvolupament de MARLA[1], una aplicació de codi obert per execucions que empren el model de computació MapReduce, discutit a la secció 2.1, en una infraestructura completament *Serverless* per a beneficiar-se dels avantatges que proporciona aquest entorn (Secció 3.1). Per tal de maximitzar el rendiment de la infraestructura, serà necessari realitzar un anàlisi de costos per a determinar la rendibilitat front a un altra solució i la quantitat d'instàncies de mapatge i reducció òptimes. Aquest anàlisi es pot realitzar a partir dels desenvolupaments del capítol II. D'altra banda, s'estudia l'adequació d'aquest tipus de servei per a execucions científiques. Per aconseguir-ho, es realitza un estudi exhaustiu de les prestacions de les instàncies *Serverless*, analitzant

[1]MARLA: https://github.com/grycap/marla

tant les prestacions de les comunicacions com la velocitat de còmput. Aquest anàlisis es repeteix per a diferents tipus d'instàncies *Serverless*, classificant inclús el maquinari on són executades.

Els resultats mostren grans heterogeneïtats en el maquinari on s'executa el servei de AWS Lambda. Més encara, comparant els resultats entre instàncies amb, teòricament, les mateixes prestacions i sent executades al mateix tipus de maquinari, s'obtenen fluctuacions no negligibles tant en les prestacions de les comunicacions com en la velocitat de còmput de la CPU. En quant a les heterogeneïtats del maquinar, a l'estudi s'han registrat quatre tipus diferents de processadors on una instància *serverless* pot ser executada, independentment de la configuració d'aquesta. Al quantificar l'efecte de la heterogeneïtat del maquinari, s'han mesurat diferències en la velocitat mitjana de processament d'un 53%, entre el processador més i menys potent. Més encara, les velocitats de processament presenten fluctuacions no negligibles que s'accentuen als processadors menys potents, arribat a diferències majors d'un factor 2 entre les velocitats de processadors distints. Considerant un mateix tipus de processador, al pitjor dels casos, s'han mesurat fluctuacions de les velocitats de processament de fins a un 50%, tot i que, com s'ha dit, l'efecte disminueix quan la instància *serverless* s'executa a processadors amb models més moderns.

D'altra banda, les comunicacions ofereixen resultats, en mitjana, més estables, però amb fluctuacions molt més pronunciades. Els tests realitzats en la transferència de dades entre la instància *serverless* de AWS Lambda i el servici d'emmagatzemament S3 mostra diferències en la velocitat de transmissió de dades, d'instàncies amb la mateixa configuració, de fins a dos ordres de magnitud en les proves realitzades. Més encara, al repetir les proves variant la quantitat de dades a transferir, vegem com la dimensió de les fluctuacions s'agreuja en augmentar dita quantitat. Els resultats apunten a que l'efecte sobre la degradació de les prestacions de la xarxa es deuen a una limitació o sobrecàrrega de la xarxa quan múltiples instàncies transfereixen dades.

Aquests resultats, junt a altres treballs de la literatura, reforcen la necessitat d'adaptar les aplicacions científiques per a poder ser executades en un entorn amb prestacions impredictibles, tal i com s'ha discutit en la seccions 3.2 i 3.3.

Abstract

MapReduce is one of the most widely used programming models for analysing large-scale datasets, i.e. Big Data. In recent years, serverless computing and, in particular, Functions as a Service (FaaS) has surged as an execution model in which no explicit management of servers (e.g. virtual machines) is performed by the user. Instead, the Cloud provider dynamically allocates resources to the function invocations and fine-grained billing is introduced depending on the execution

time and allocated memory, as exemplified by AWS Lambda. In this article, a high-performant serverless architecture has been created to execute MapReduce jobs on AWS Lambda using Amazon S3 as the storage backend. In addition, a thorough assessment has been carried out to study the suitability of AWS Lambda as a platform for the execution of High Throughput Computing jobs. The results indicate that AWS Lambda provides a convenient computing platform for general-purpose applications that fit within the constraints of the service (15 minutes of maximum execution time, 3008 MB of RAM and 512 MB of disk space) but it exhibits an inhomogeneous performance behaviour that may jeopardise adoption for tightly coupled computing jobs.

1.1 Introduction

MapReduce [96] is one of the most used programming models for analysing large-scale datasets, i.e. Big Data. Apache Hadoop [25], an open-source platform for reliable, scalable and distributed computing provided the execution support to the MapReduce programming model. Indeed, the Hadoop ecosystem has flourished in the last years resulting in a myriad of tools and services for distributed programming, NoSQL databases, SQL-based databases and machine learning, among many other categories (see the Hadoop Ecosystem table [97] for further details). Apache Hadoop requires a distributed computing infrastructure to support the execution of MapReduce applications.

Infrastructure as a Service (IaaS) Clouds provide distributed computing infrastructures, offered by public Cloud providers such as Amazon Web Services (AWS) [44], Microsoft Azure [65] and Google Cloud Platform (GCP) [98]. Indeed, these providers include in their portfolios services to be able to automatically deploy Hadoop (and Spark) clusters on pay-as-you-go basis. This is the case of Amazon Elastic MapReduce (EMR) [99], for AWS, HDInsight [100] for Microsoft Azure, or Cloud Dataproc [101], for GCP. These services provide automatic deployment capabilities and, to some extent, the ability to scale the clusters in order to increase or decrease the number of worker nodes of the cluster. There are also software projects to dynamically deploy Hadoop (and Spark) clusters on on-premises Clouds. This is the case of Sahara [102], to provision data-intensive application clusters on top of the OpenStack [103] Cloud Management Platform (CMP).

In recent years serverless computing has surged as an execution model in which no explicit management of servers (e.g. virtual machines) is performed by the user. Instead, the Cloud provider dynamically allocates resources to the execution unit, which are function invocations. Users write functions in a programming language supported by the service with no assumption about the underlying computing infrastructure on which the function invocation will be run on. By creating stateless functions, a highly-parallel execution model can be used, where multiple

concurrent invocations of the same function can be triggered in response to events occurred in the infrastructure. The cost model for serverless computing really becomes pay-per-use, since no costs are involved for the user unless the function is invoked, as opposed to traditional cost models for IaaS clouds, where Virtual Machines are typically billed per-second regardless of their actual usage. Because of the advantages listed above, serverless computing is now used as an effective simple programming model for a variety of applications [104] [105] [106].

One of the pioneer services of serverless computing is AWS Lambda [107], which allows to execute stateless functions coded in one of the programming languages supported (Node.js, Java, C#, Python and Go), in response to events, on a massive scale (up to 3000 parallel invocations). As opposed to traditional IaaS Clouds, in AWS Lambda there is no explicit management of servers by the users and elasticity is automatically provided by the platform, since the functions are designed with no assumptions on the underlying infrastructure, i.e., they are designed stateless. The large degree of parallelism supported by AWS Lambda is one of the key features of the service.

Indeed, the report from the workshop and panel on the Status of Serverless Computing of the First International Workshop on Serverless Computing (WoSC) 2017 [108] indicated that a hot research topic will be its use for parallel programming and there is a challenge to extend the MapReduce use of FaaS.

To this aim, this paper studies the suitability of AWS Lambda to support the MapReduce programming model and provides two key contributions. On the one hand, an open-source framework is introduced (MARLA - MApReduce on LAmbda) in order to run MapReduce jobs in Python on AWS Lambda. On the other hand, a thorough performance assessment of this service is performed through the execution of several case studies. This highlights that AWS Lambda shows sporadic disparate performance which affects the timeliness of tightly coupled jobs, as is the case of MapReduce applications. We propose identification techniques of these issues and, to the extent that is possible, mitigation approaches to minimise the impact for the execution of these kind of jobs.

After the introduction, the remainder of the paper is structured as follows. First, section 1.2 describes the related work in the area and compares the existing approaches with the proposed framework in this paper. Next, section 1.3 introduces the MARLA framework, describes its architecture and points out the optimisation strategies required to efficiently take advantage of the computing capabilities of AWS Lambda. Then, section 1.4 provides a detailed assessment of the behaviour of AWS Lambda when executing several workloads. To the authors' knowledge this is the most comprehensive performance study carried out in AWS Lambda that identifies both the benefits and the limitations of this serverless platform as a general-purpose scalable computing platform. Finally, section 2.6

summarises the main achievements of this paper and points to future work.

1.2 Related Work

There can be found in the literature few works using serverless computing to support the execution of MapReduce jobs.

AWS has its own Serverless Reference Architecture for MapReduce (SRAM) described in [109]. In this architecture the user previously uploads to an S3 bucket the input data, that has to be previously partitioned in chunks, and write their code as Python or Node.js functions. Then, there is a Python application that the user executes locally in charge of orchestrating the execution by creating the necessary Lambda functions to act as mappers and reducers and starting the execution of the MapReduce job.

Another approach is described in the work by Jonas et al. [110] in which they describe PyWren[2], a platform that enables running Python code at massive scale via AWS Lambda. It enables to launch simple Python parallel programs but also MapReduce jobs. Since it is a general solution, the code required to create a MapReduce application demands a higher complexity compared to MARLA or the SRAM. Indeed, as it happens with the SRAM, it also requires to have the input data pre-partitioned.

Ooso [111] is a serverless MapReduce Java library based on AWS Lambda. The user defines the functions using Java and uploads the pre-partitioned data to an S3 bucket, as is the case of the aforementioned solutions. It relies on Terraform[3] to create all the related AWS resources, instead of Boto, which is commonly used by the previous solutions.

Corral [112] is a framework designed to be deployed to serverless platforms, like AWS Lambda. This framework is entirely wrote in Go. The runtime model consists of stateless, transient executors controlled by a central driver.

As opposed to previous works, MARLA improves previous solutions since it covers all the steps of the MapReduce job execution, including automated partition of the input data using the optimum size for the memory assigned to the Lambda functions. Using MARLA the user only has to define the *map* and *reduce* functions in Python and then use the framework to automatically define the functions. This results in the creation of the Lambda functions and the binding to a certain S3 bucket so that a latent MapReduce service is deployed at zero cost. Then, each time a file is uploaded to that S3 bucket all the activities related to the execution of the MapReduce job are automatically carried out and the results are also written to the output S3 bucket. This provides an effective and convenient approach

[2] `http://pywren.io`

[3] `https://www.terraform.io/`

that provides a highly-scalable MapReduce execution service that does not involve the pre-deployment of computational instances and that can be automatically triggered on-demand in response to a file upload event.

MARLA does not require a local application to orchestrate the start of the MapReduce jobs, thus providing a highly scalable multi-tenant MapReduce framework that is triggered on demand. When multiple files (from one or more users) are uploaded to the input bucket on S3, all executions will be processed concurrently, thus avoiding the bottleneck produced by using an external orchestrator.

There are also works that have studied the performance of serverless computing platforms for different Cloud providers (see [113], [114] and [115] for details). Previous studies have shown that AWS Lambda runs on an heterogeneous system and the performance of the Lambda invocations will be proportional to the allocated memory. However, additional tests will be performed in this work in order to determine the performance behaviour of AWS Lambda when running coupled job executions such as those that arise when executing a MapReduce application.

1.3 Architecture of MARLA

1.3.1 Architecture

MARLA has been designed as a lightweight framework to allow the execution of Python-based MapReduce jobs on AWS Lamba. To this aim, this section describes the proposed architecture, its components and the workflow to efficiently perform data-oriented processing on such serverless platform.

Figure III.1 shows the architecture diagram. The architecture is entirely composed by Cloud services on AWS without any external components. It consists of three groups of Lambda functions: The *coordinator*, the *mappers* and the *reducers*. In addition, it requires two S3 buckets, one for input data and another one to collect the post-processed data generated as output.

The execution workflow of the MapReduce job starts when a dataset is uploaded in the "input" S3 bucket. This causes an S3 event to activate an invocation to the coordinator Lambda function, which calculates the optimal size of the data partitions dividing the total dataset size into as many chunks as the user indicates (N) provided that the mapper functions can store in memory that amount of data in their allocated RAM. If the mappers do not have enough RAM, the coordinator Lambda function will recalculate the number of chunks to fit in each mapper.

To calculate the size of the data chunk processed by each mapper, the coordinator function does the following process. First, it calculates the possible size of each data chunk dividing the size of the file to process by the number of mappers specified by the user.

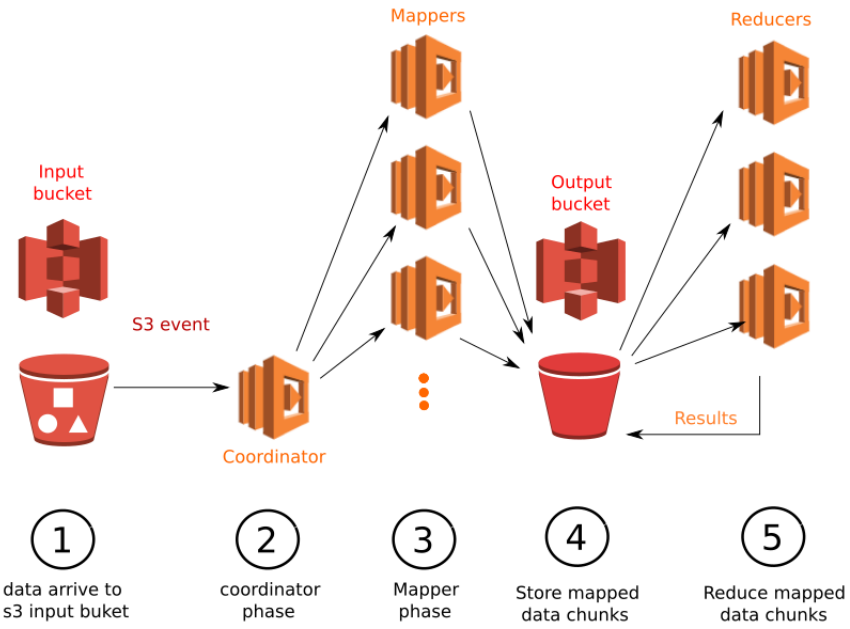

Figura III.1: Architecture of MARLA to support MapReduce on AWS Lambda.

$$chunkSize = \frac{totalDataSize}{N_{mappers}}. \tag{III.1}$$

Next, the coordinator function checks if the *chunkSize* does not satisfy the conditions listed below. These conditions are checked in the following sequence:

- *chunkSize* is smaller than the minimum block size specified by the user ($MINBLOCKSIZE$). In this case, *chunkSize* value will be set to *MINBLOCKSIZE* and the number of mappers are recalculated as follows,

$$N_{mappers} = int\left(\frac{totalDataSize}{MINBLOCKSIZE}\right) + 1 \tag{III.2}$$

- *chunkSize* is bigger than the calculated safe memory size ($safeMemorySize$). In the coordinator function, $safeMemorySize$ is calculated as a percentage of the memory assigned to the mapper functions. This ensures that the data to process fits in the mapper functions. In this case, the *chunkSize* value will be set to $safeMemorySize$ and the number of mappers recalculated as follows:

$$N_{mappers} = int\left(\frac{totalDataSize}{safeMemorySize}\right) + 1 \quad \text{(III.3)}$$

- *chunkSize* is bigger than maximum block size specified by the user (*MAXBLOCKSIZE*). In this case, the *chunkSize* value will be set to *MAXBLOCKSIZE* and the number of mappers is recalculated as follows:

$$N_{mappers} = int\left(\frac{totalDataSize}{MAXBLOCKSIZE}\right) + 1 \quad \text{(III.4)}$$

Notice that in all the previous recalculations of $N_{mappers}$, the coordinator function always adds one extra mapper, which is responsible to process the residual data chunk with size:

$$residualData = totalDataSize - (N_{mappers} - 1) \cdot chunkSize \quad \text{(III.5)}$$

Notice that the size of the residual data, in the worst case scenario, will be $chunkSize - 1$ *bytes*. This approach prevents any mapper to process its corresponding chunk plus the mentioned residual data, what could cause an additional overload in that mapper. However, depending on the dataset, the residual data can be small enough to create an under-loaded mapper that finishes its execution faster than the other mappers.

Finally, the coordinator invokes the first mapper Lambda function with an environment variable that stores the dimension of each data chunk size. The first mapper Lambda function starts a logarithmic reduction approach so that it invokes the second mapper and, then, both Lambda function invocations will invoke another two. This procedure is recursively repeated until all mappers have been invoked. Notice that this approach shows a $log_2(N_m)$ cost, where N_m is the number of mappers thus minimising, to the extent that is possible, the delays related to the invocation of the mapper function.

Next, each mapper downloads its chunk of data to perform the *map* phase in parallel to the other invocations. This *map* process produces as results a list of *key/value* pairs that will be reduced in the next phase. To process the data, the mapper Lambda function employs the user-defined mapping function.

Then, the mapper Lambda function will sort the data and divide the mapped results in chunks (mapped chunks). Each mapped chunk will be processed by an independent *reduce* Lambda function. We assume that the *reduce* process only needs all *pairs* with the same *key* to complete the data processing i.e. *pairs* with different *key* are processed independently in the reduce step. This is not a hard restriction since the user can assign any *key* to mapped *pairs* during the *map*

process. Therefore, to ensure that each *reduce* Lambda function has all the required data for post-processing, the division to create mapped chunks is performed given the first character of each *pair key* obtained from the previous mapping process. Therefore, the generated mapped chunks for each reduce will be alphabetically independent, i.e. all pair *keys* with the same first character will be stored in the same mapped chunk, ensuring that the mentioned condition for the *reduce* step will be fulfilled.

These chunks are stored in the output bucket with a name that identifies their corresponding reduce number and the mapper that produced it. In addition, a prefix for each stored mapped chunk will be added using an $md5$ hash to achieve optimal S3 performance, as described in [116]. This last step ensures a good distribution of all data chunks among S3 partitions, thus enhancing the data access performance in S3.

It is important to point out that before the execution of the mapper Lambda function corresponding to the penultimate partition ends, it will invoke the first reduce Lambda function, that will perform a tester function. This single reduce will check the uploaded chunks until all mapped partitions corresponding to the last reduce have been uploaded. Then, this function will invoke all the reducers and finish its execution. Notice that, unlike mappers, the number of used reducers on a single execution will be small (in the order of tens). Using more reducers will cause that several reducers try to process the same ASCII interval, so these "duplicate" reducers will not perform any job. In addition, as it will be explained in section 1.4, incrementing the number of reducers will be counterproductive. Therefore, there is no need to use a logarithmic approach to perform the reduction phase.

In the last step, each Lambda reduce function will execute the user's predefined reduce function on its corresponding mapped data chunks. To do this, it will download as many mapped chunks as possible per the memory size, which is chosen by the user. The reducer will download at least one entire chunk, since a single mapped data chunk will not be partitioned. Then, the reduce user function is executed on the downloaded mapped chunks, plus the previously processed ones. This download-process loop will be repeated until all corresponding mapped chunks have been processed. Therefore, all the mapped chunks are reduced. Finally, the results are stored in the output bucket. Notice that each reduce will produce a results file with its identifier (from 0 to $N_{reduces} - 1$). MARLA has been deliberately designed to not aggregate the results of the multiple reducers, since the memory of a Lambda function is limited to $3008MB$ and this might not be enough to host the final result. So, in each execution, the user will obtain as many result files as reducers used, even if all the result files fit in a single Lambda function. Notice that the aggregation of these results can be performed as an

additional post-processing on a high performant EC2 instance. In particular, this aggregation of results is not automatically performed by MARLA, since this is the same behaviour seen in the Apache Hadoop framework, where multiple reducers will generate separate output files and, therefore, the user is responsible for joining these files.

This approach allows to process huge results data files unless the *pair keys*, obtained from the mapping process, were poorly distributed alphabetically. An easy way to distribute the load across multiple reducers if the *keys* are not well distributed in the input data, is to hash the *keys* using numbers and extract the residual part of the integer division with the total number of reducers, as shown in Eq. III.6.

$$reduce = hash(key)\%N_{red}, \tag{III.6}$$

This number assigns an ASCII interval and, consequently, the reducer that must process this *pair key/value*. Next, taking into account that the printable ASCII characters interval (with no ASCII extensions) is $[32, 126]$, the total number of printable characters is $126 - 32 + 1 = 95$. Therefore, the number of characters in each reducer interval can be calculated by,

$$\Delta interval = int\left(\frac{95}{N_{red}}\right) \tag{III.7}$$

Finally, a character within the corresponding interval must be added as prefix of *pair key* as shown in Eq. III.8

$$key = chr\left(\Delta interval \cdot reduce + 33\right) + str(key). \tag{III.8}$$

Notice that adding 33 and not 32 avoids appending a *space* that will be removed by strip functions. Also, this requires to avoid an overlap with the special characters employed to separate columns in the original dataset. For example in CSV (comma-separated values) files, the comma should not be used as part of the data to avoid misleading interpretations by the framework.

1.3.2 Failure handling

This section briefly explains failure handling on MARLA. The exceptions caused by bad input from the user will terminate the Lambda function invocation, thus halting the data analysis. A failure on one or more functions (coordinator, mappers or reducers) will never produce an unlimited number of retries. Indeed, all the Lambda functions will eventually finish, since the "tester" reduce can invoke itself a limited number of times to avoid infinite recursive invocations in case one mapper fails. The same behaviour will be exhibited when a failure is detected uploading

or downloading from S3, since the current or next processing step will not find the required data. Depending on where the failure is detected, the behaviour of MARLA will be different:

- **Failure on the Coordinator**: No mapper will be invoked. The execution will end up at this point.
- **Failure on the Mapper function**: Failed mappers will not upload any mapped data to S3, so the tester reduce function will never find all the required partitions and the reducers will not be invoked. Eventually, the reducer tester will finish its invocation. However, successfully executed mappers will upload their mapped data partitions to the output bucket.
- **Failure on the Reduce function**: Even if one or more reducers fail, this only affects to the data that failed reducers must process, since reducers are independent from each other. Therefore, at the end of the execution, the data of successfully executed reducers will be uploaded to S3.

Mapper and reducer failures, in most cases, can leave partially processed data in the S3 output bucket, to be handled by the user.

1.3.3 Assumptions and Limitations

The current assumptions and limitations of MARLA are described below:

- MARLA works, at the moment, with plain text column-based data files where the column change is specified by a "," character. Future works includes extending support to binary data and other data formats.
- The input data to process does not need to be pre-partitioned. MARLA will perform the data partition step automatically considering the configuration specified by the user and the memory allocated to functions.
- The user "mapping" and "reduce" functions must be implemented in Python, which is the only supported language at the moment.
- The access credentials and IAM roles required to use AWS services must be provided by the user. MARLA will not create any role or permissions, but use the existing ones.

1.3.4 Assessment of MapReduce on AWS Lambda

This section provides a thorough assessment of the benefits and limitations of AWS Lambda as a general computing platform and, in particular, for the execution of dependent parallel jobs, as in MapReduce applications. It also aims at comparing the developed framework to other data processing frameworks, in particular the Serverless Reference Architecture for MapReduce (SRAM) developed by AWS. To achieve a fair comparison, we use the same benchmark as the SRAM [109], a subset of the Amplab benchmark [117]. The benchmark measures response time on a handful of relational queries: scans, aggregations, joins, and user-defined functions, across different data sizes. The authors extract workloads and queries from the previous study by Pavlo et al. [118]. However, since the datasets used are different, the results of both benchmarks are not directly comparable.

To be able to compare with the SRAM, the same subset of Amplab benchmark queries will be tested. The queries are:

- **Scan query**: The corresponding data set contains 90 M rows and approximately 6.36GB of data. The queries select the *pageURL* and *pageRank* from rankings where $pageRank > X$, where $X = \{1000, 100, 10\}$. The first two queries have been tested.
  - **Scan 1a**: SELECT *pageURL*, *pageRank* FROM rankings WHERE $pageRank > 1000$.
  - **Scan 1b**: SELECT *pageURL*, *pageRank* FROM rankings WHERE $pageRank > 100$.
- **Aggregation query**: Involves the *UserVisits* dataset, which contains 775M rows with approximately 127GB. Only one query from this group has been performed, defined as follows:
  - **Aggregate 2a**: SELECT SUBSTR(*sourceIP*,1,8), SUM(*adRevenue*) FROM *uservisits* GROUP BY SUBSTR(*sourceIP*,1,8).

The SRAM only run the subset of queries explained above because the benchmark is designed to increase the output size by an order of magnitude for the a, b, c iterations. Given that the output size, when the reference architecture executed the benchmark, did not fit in the Lambda memory (maximum of 1536MB at that time), they could not proceed to compute the final output.

The results obtained by the reference architecture, and other analysis frameworks, are shown in Table III.1. This results, extracted from [109], provide

the execution times spent to process the queries with different technologies. The results will be briefly explained below. A detailed explanation of the results, together with the used configurations, is available in [117].

Technology	Scan 1a	Scan 1b	Aggregate 2a
Amazon Redshift (HDD)	2.49	2.61	25.46
Impala - Disk - 1.2.3	12.015	12.015	113.72
Impala - Mem - 1.2.3	2.17	3.01	84.35
Shark - Disk - 0.8.1	6.6	7	151.4
Shark - Mem - 0.8.1	1.7	1.8	83.7
Hive - 0.12 YARN	50.49	59.93	730.62
Tez - 0.2.0	28.22	36.35	377.48
Serverless MapReduce	39	47	200

Taula III.1: Execution time in seconds of each query for different technologies. *Mem* and *Disk* refers to storage type, memmory or disc respectively. This table has been extracted from the Serverless MapReduce reference architecture repository [109].

The first queries (*Scan 1a* and *Scan 1b*) test the throughput obtained with each framework when reading and writing table data. The best performers are Impala (Mem) and Shark (Mem) which show excellent throughput by avoiding disk usage. For on-disk data, Redshift shows the best throughput, since it uses columnar compression which allows it to bypass a field that is not used in the query.

The last query (*Aggregate 2a*) applies string parsing to each input tuple, then performs a high-cardinality aggregation. Redshift's columnar storage provides greater benefit than in both aforementioned queries since several columns of the data table are unused. While Shark's in-memory tables are also columnar, the bottleneck relies on the speed at which the *SUBSTR* expression is evaluated.

For our tests we use 100 mappers Lambda functions of ($1536MB$) for *Scan* queries. However, in the third query, we use the current most performant Lambda functions ($3008MB$) to reduce the number of parallel invocations. For example, for Lambda functions with $1536MB$ we need, at least, 800 parallel invocations to fit all the data. Also, since the queries are designed to increment the results size in one order of magnitude, the number of reducers used in the *Scan* queries are, in order, 1 and 10, since the results of the first query fit in one Lambda function. For the third query (*Agregation1*) we use different configurations to find an optimal one, as we will see below.

First, we show the execution times measured for the *Scan 1a* and *Scan 1b* queries on Figure III.2. The figure shows important discrepancies in the execution time.

Therefore, each query has been repeated 30 times to provide a time distribution instead of obtaining a single value, which definitely provides a misrepresentation of the actual execution time.

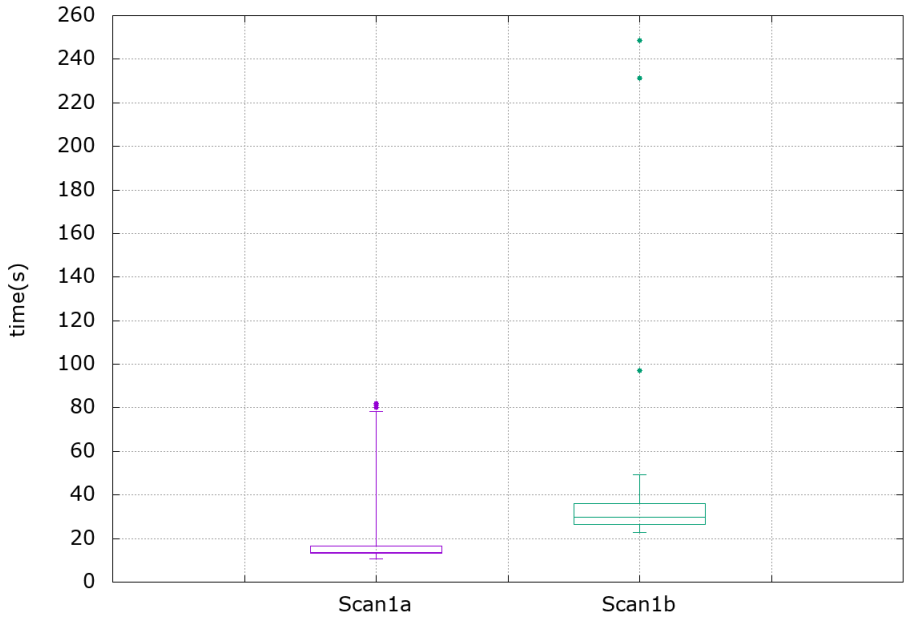

Figura III.2: Execution times distribution for *Scan 1a* and *Scan 1b*. Whiskers group the 90% of the data points. For *Scan 1a*, the execution times interval is, in seconds, $[10.699, 81.874]$, with a mean value of $13.653s$. In the same way, the interval of *Scan 1b* is $[22.758, 248.612]$ with a mean value of 29.635 s.

As we can see in Figure III.2, the measured execution times show significant fluctuations. One of the possible causes could be that some function invocations exhibit a failed S3 request and the entire system waits until this function is restarted in order to finish processing its corresponding data. This kind of errors are explained in the work by Garfinkel [119]. Other possible cause of these fluctuations is the *cold start*, which is the increased invocation latency experienced when a Lambda function is invoked for the first time or after not being used for a certain period of time. This has been extensively studied in the work by Lang et al. [120]. They measured, among other things, cold start times for Node.js 6 Lambda functions of $128MB$ and $1536MB$, shown in Table III.2.

However, according to the values in Table III.2 the slower executions when executing *Scan 1a* and *Scan 2a* cannot be produced by cold start. In order to find the causes of those fluctuations, additional insights should be obtained from the execution of the MapReduce jobs on AWS Lamba. Figures III.3 and III.4 show

Provider-Memory	Median	Min	Max	STD
AWS-128	265.21	189.87	7048.42	354.43
AWS-1536	250.07	187.97	5368.31	273.63

Taula III.2: Cold start latencies (in ms) in AWS using Node.js 6 functions. Data obtained from [120].

the distribution of the execution time for all mappers in queries *Scan 1a* and *Scan 1b* respectively, across 30 executions. These times have been measured using the execution time information provided by *CloudWatch* logs.

The whiskers were set to group 98% of the data points. Therefore, the two more divergent mappers are represented as points. Notice that the lowest execution time for each run corresponds to the last mapper, who has a residual data chunk, as we saw in section 1.3. Thus, this low execution time is not an unexpected result. However, we should expect a similar execution time among the other mappers, which process the same amount of data.

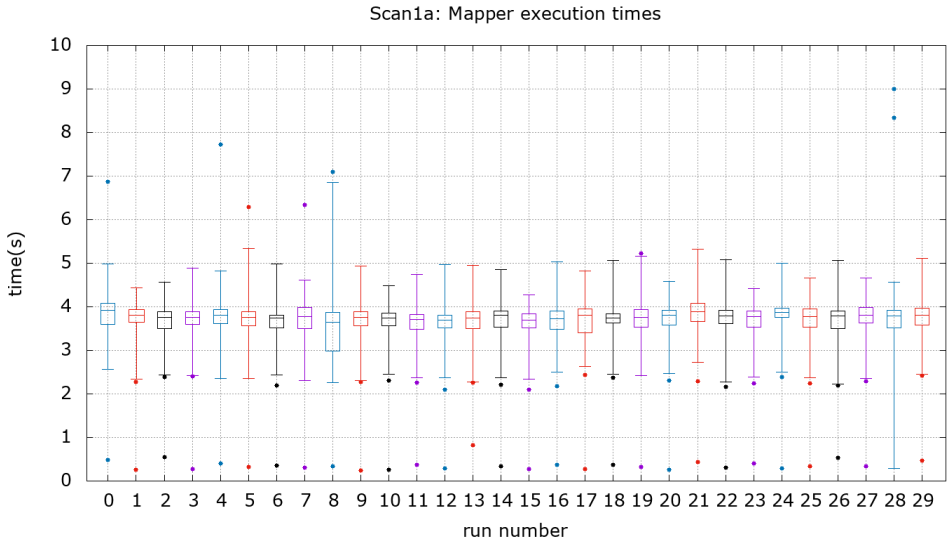

Figura III.3: Mapper execution times distribution for the "Scan 1a" query. Whiskers group 98% of total data points. Coloured lines are only included for the convenience of the reader.

The previous figures point out that the execution times of the mappers exhibit a non homogeneous behaviour, showing large fluctuations. On the one hand, *Scan 1a* has only 1 reduce, and its execution times are between 7.8 s and 15.1 s with a mean value of 11.65 s. Therefore, the executions with times of approximately 80 s cannot be exclusively attributed to the cold start. On the other hand, focusing on

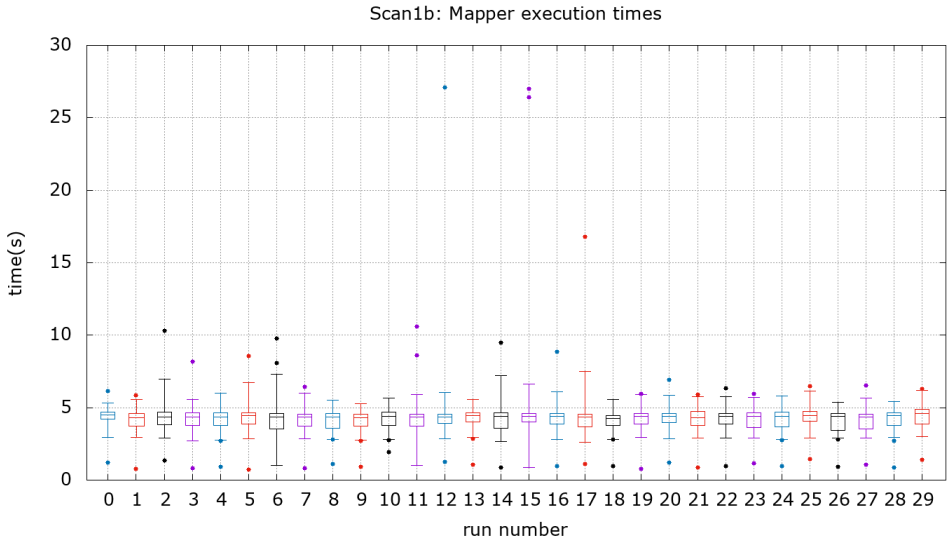

Figura III.4: Mapper execution times distribution for the "Scan 1b" query. Whiskers group 98% of total data points. Coloured lines are only included for the convenience of the reader.

the *Scan 1b* query, the cold start again cannot explain the high execution times of approximately 240 *s* shown in Figure III.2, as it happens with *Scan 1a*. We used 10 reducers in this query, thus improving the execution time of the reducers, since each one processes less data than if there was only one reducer. However, in some executions we observe that few reducers spend more than 100 *s* to download and process all data, when the mean is 14.675 *s*.

An analysis of the AWS *CloudWatch* logs revealed that these slow reducers require much longer time than the mean of reducers to download data from S3. This effect seems to be caused by the increment of the number of reducers. The more reducers are used, the more partitions needs each mapper to upload to S3 (one per reducer) and more simultaneous PUT/GET requests will be performed by the Lambda functions. Remember that we adopted a hashed prefix included on uploaded mapped data chunks in order to minimize the scalability limitations in S3 by the underlying data partitions.

Concerning the third query (*Aggregate 2a*), some preliminary tests have been carried out to determine a good configuration to perform the tests. The best configuration tested is 250 mappers and 15 reducers. To determine when a configuration is good enough, we compared the execution time with the one obtained by the SRAM, shown in Table III.1. If the first executions on same configuration give execution times worse than those from the SRAM we changed the configuration. For this reason each configuration has a different number of executions and the

best one has been repeated 20 times. This result, shown in Figure III.5 together with other configurations, seems to indicate that for these kind of applications that use S3 as temporary storage across stages of the MapReduce, AWS Lambda functions face certain scalability limitations.

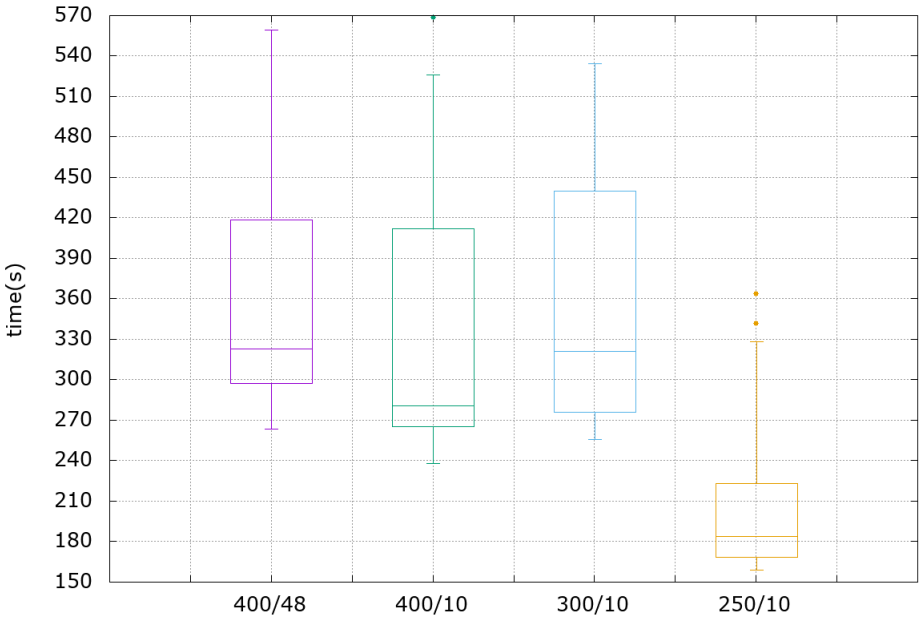

Figura III.5: Execution times distribution for "Aggregate 2a". The x axis indicate the number of mappers and reducers used, using the format $N_{mappers}/N_{reducers}$. The number of executions done for each test are, respectively, $6, 10, 5, 20$. Whiskers group the 90% of the data points..

The results shown in this section seem to indicate that AWS Lambda does not provide a homogeneous computing environment even if all Lambda functions have the same memory assigned. However, the previous tests cannot ensure that all functions have the same workload or discriminate the real effect that slow down some of the function invocations. To address these issues, the following section will perform a set of controlled tests in order to investigate the performance discrepancies in AWS Lambda.

Also, the results provided by the SRAM (Table III.1) are comparable to our execution time results. However, as we see in the current section, the results in Table III.1 should not be interpreted as a precise value. Instead, a range of execution times should be provided, as we have done in this paper, in order to facilitate comparison among researchers. The upcoming section 1.4 will study the causes of these fluctuations in terms of CPU and network performance.

1.4 AWS Lambda Performance Assessment

In this section, the suitability of Lambda functions to perform general-purpose computing will be studied. For this purpose, we select two kind of tests: CPU performance and network throughput. Even though the technical implementation details of AWS Lambda are not disclosed, we anticipate that the performance of a certain Lambda function invocation is related to the performance of the underlying computing node (i.e. Virtual Machine, or instance in AWS terminology) on which it is running. Therefore, in order to identify on which node each Lambda function invocation is running we take advantage of the trick explained in the work by Wang et al. [120], summarised as follows. The */proc/self/cgroup* file has a special entry called *instance root ID* that starts with "sandbox-root-" followed by a 6-byte random string. The authors found that this ID can be used to reliably identity a host virtual machine (VM), even outperforming the heuristic initially proposed by Lloyd et al. [115] based on the VM uptime in `/proc/stat`, which proved to be unreliable.

Another important data to extract is the memory assigned to the VM. To know this value will allow us to calculate the maximum number of Lambda functions that fit in a single VM, depending on the Lambda memory size. To extract that data, we use the command $free - t - m$. The maximum measured VM memory sizes are $3757MB$ and $3945MB$.

All the following tests have been performed on the *us-east-1* region of AWS using Python 3 as the programming language for the Lambda functions. Another factor to consider is that the time has been measured using the Python builtin library *time*, that is, times have been measured inside the Lambda executed code. This approach ensures that the cold start does not influence time measures.

In following subsection, only one Lambda function has been created for each performed test and this one will be invoked several times in multiple test executions, as will be explained then. So, when we talk about invocations, we are referring to invocations of the same test Lambda function.

1.4.1 CPU performance

For the execution of MapReduce jobs and coupled job executions such as workflows, a desirable property is performance homogeneity or, at least, to be able to anticipate the performance of each node to efficiently distribute the workload. According to AWS documentation, a Lambda function performance is proportional to its user-configured memory [121]. Previous tests in the literature [120] obtained a linear regression between CPU percentage assigned to function and Lambda function memory. However, in these tests, we aim at determining if all Lambda function invocations, with the same assigned memory, have a similar FLOPS

performance. Since all of them have the same assigned memory, theoretically, the same CPU performance should shown by each Lambda function. This way, the differences between execution times by executing the very same code must be produced by the underlying heterogeneity in the system.

The tests consist in executing a fixed number of floating point products. The number of operations depends on the Lambda function memory. Therefore, all the invocations with the same assigned memory will do exactly the same amount of computation. Notice that only the time differences among the function invocations is important, regardless of the total execution time.

The *test execution* represents the simultaneous N invocations of one test Lambda function, with the same code and characteristics (memory assigned, timeout, etc.). As we explain before, all the invocations in the same test run the same Lambda function. A test is formed by a predefined number of test executions where each one has identical configuration to each other (same Lambda function, number of invocations, code, memory, etc.). In addition, test executions will be invoked at regular time intervals that depends on the total workload, to avoid overlapping executions.

For each test execution, each function invocation has a unique numeric identifier (invocation ID or invocation number) starting from 0 to $N-1$, where N is the number of simultaneous invocations in each test execution. To minimize the delay between function invocations, the ID 0 is invoked first, in each test execution. Then, this function invocation is in charge of invoking all the others to prevent, to the extent that is possible, additional delays from our possible network fluctuations, latency and cold start. Because that, the function invocation with ID 0 is not considered in any test execution.

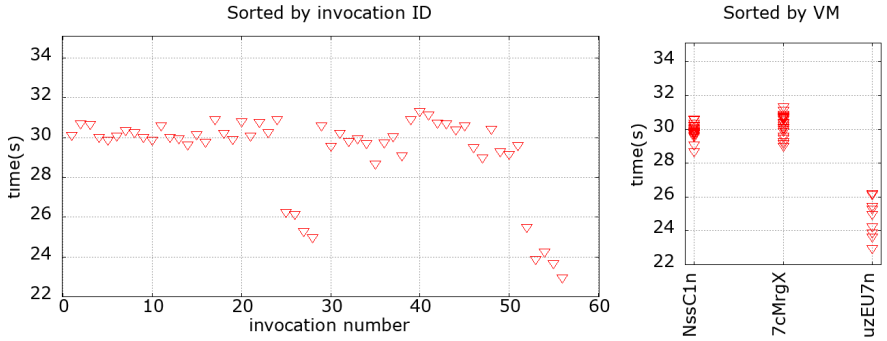

Figura III.6: Execution time distribution with 56 invocations of test Lambda function with $128MB$ of memory. Invocations sorted by ID (left) and grouped by virtual machine identifier (right).

In the first test, 56 invocations with $128MB$ of assigned memory have been

used. The selected number of Lambda invocations ensures that more than one VM will be used to allocate all functions, taking into account the maximum assigned memory to each VM. The results of one test execution are described in Figure III.6, where the execution time of each Lambda invocation has been represented sorting the Lambda invocation by number (left) and grouping invocations that run on the same VM (right). The graph clearly classifies the output results in two different groups that identify two different performance baselines. This proves that AWS Lambda features a heterogeneous computational infrastructure even if the the same memory size is allocated to the Lambda functions. In addition, there is a clear correlation between the execution time of the Lambda invocations and the underlying allocated VM.

We were able to replicate this behaviour across multiple test executions. From the outcome of tests it is possible to claim that the performance delivered by each VM does not change during successive test executions. This fact suggests that this difference of speed can be related with the CPU of the underlying VM on which the function code is being executed (encapsulated in a container). Therefore, we can infer that assigning a certain amount of RAM to a Lambda function indeed sets a certain CPU share in the underlying container run on the EC2 instance (managed by AWS and completely hidden from the user) and, therefore, this share results in greater performance on instances (VMs) with better CPU capabilities.

Additional tests were carried out with different assigned memories and the same behaviour described above was reproduced. For example, Figure III.7 shows a test execution of 40 invocations with $1216MB$ of assigned memory. In that test, the assigned VM can store only two Lambda functions each one. In Figure III.7 (down), most VM IDs have two assigned data points corresponding to two invocation measured times. However, the measured times on the same VM are so close that the points are overlapped.

As mentioned above, this behaviour could be explained if the CPUs used in each VM are not the same, and therefore, deliver different performance. Therefore, in the following test we attempt to find a correlation between CPU model and execution speed. To extract the CPU information of the corresponding underlying VM, the Lambda functions read it from "$/proc/cpuinfo$" file. This test performs 15 test executions with 200 invocations of $1216MB$ each one. Lambda execution times have been classified by VM CPU model. Like previous tests, all Lambda invocations execute exactly the same code.

Figure III.8 shows a box-whiskers plot where each box color corresponds to one CPU model. In addition, we check that all VMs have only one kind of CPU model and all CPUs of the same model work at same frequency, according to the information extracted from "$/proc/cpuinfo$".

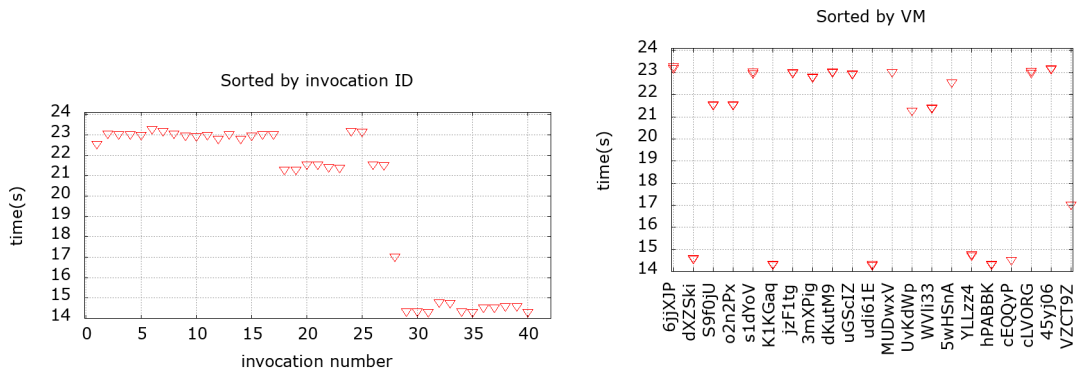

Figura III.7: Test with 40 invocations of test Lambda function with $1216MB$ of memory. Invocations sorted by ID (left) and grouped by virtual machine identifier (right).

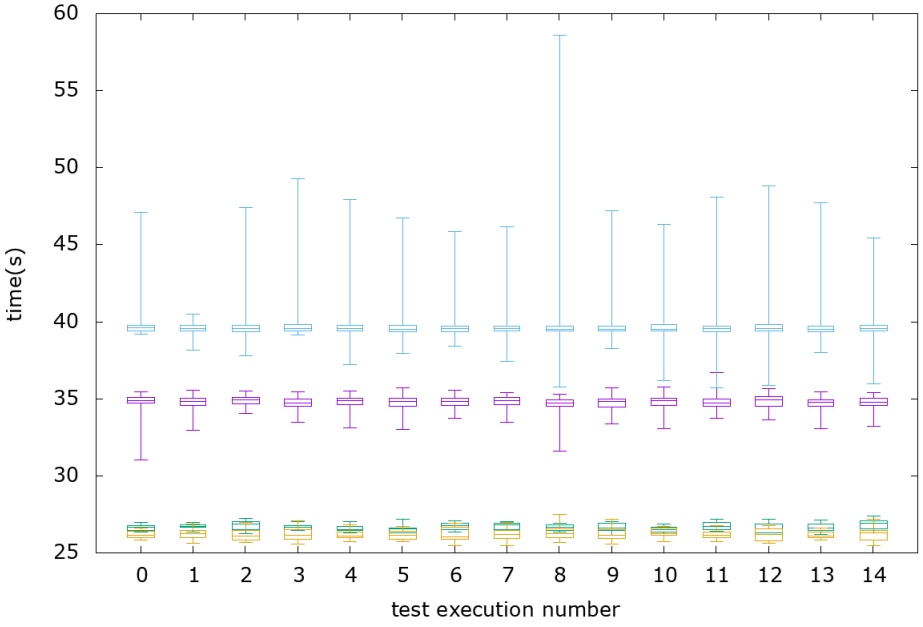

Figura III.8: Distribution of execution times for Lambda test invocations grouped by CPU model for each test execution. 200 invocations with $1216MB$ have been used in this test. Each color corresponds to one CPU model, Intel(R)-Xeon(R)-CPU-E5-2666-v3-@-2.90GHz (purple), Intel(R)-Xeon(R)-CPU-E5-2676-v3-@-2.40GHz (green), Intel(R)-Xeon(R)-CPU-E5-2680-v2-@-2.80GHz (blue), Intel(R)-Xeon(R)-CPU-E5-2686-v4-@-2.30GHz (yellow).

Even though Figure III.8 shows long whiskers on "slow" CPU models, the boxes are narrow. Therefore, the CPU model is a good indicator of VM performance and, therefore, of the Lambda function invocation performance. However, in "slow" CPU models, we must assume a small probability of big fluctuations on performance.

Notice that, far from being a homogeneous system, AWS Lambda functions show performance differences between invocations of same function ones up to a factor of two, in the worst cases.

1.4.2 Simultaneous network usage performance

Most general-purpose applications executed on AWS Lambda functions require to store partial results on a data storage facility. Since Lambda functions are allocated an ephemeral disk storage space of 512 MB, the best option is to rely on Amazon S3 to store the output data of the Lambda functions.

For this reason, we tested the network performance and fluctuations when multiple Lambda instances simultaneously access S3. Notice that, in the following tests we force Lambda invocations to upload data simultaneously to the same bucket in S3. This can be considered a worst-case scenario since, in general, not all functions will upload data at the same time. Therefore, the effects identified in this section are expected to be smaller under real-world scenarios.

The test includes also the required time to write data to Amazon S3. However, since all Lambda invocations will upload the same amount of data and use a hash as S3 prefix to ensure good distribution within S3 partitions, the time spent to write data should be similar for all the requests. Notice that adding an approximately constant value to all the measured times cannot provoke significant fluctuations on the measures. On the other hand, AWS ensures that *"your application can achieve at least 3,500 PUT/POST/DELETE and 5,500 GET requests per second per prefix in a bucket"* [122]. Our test uses a single prefix per PUT request and much less requests. Therefore, we assume that S3 is not a bottleneck and does not introduce fluctuations on upload times.

The tests consist on multiple simultaneous executions of the same Lambda function performing two steps. First, it measures the time required to upload dummy data chunks of $500KB$ each one. The number of data chunks (N_{chunks}) will change in each test to vary the workload. Then, big chunks with size of $500 \cdot N_{chunks} KB$ will be uploaded. The *big chunk* upload step forces all Lambda invocations to still use the network when they have already uploaded all chunks. Thus, this step tries to maintain the network usage while Lambda invocations perform the *multi-chunk* upload. This approach is followed to determine whether network saturation occurs. To this aim, it is desirable that all Lambda invocations simultaneously upload data. Notice that only the time dedicated to upload the

data will be measured, i.e. the cold start delay is not included in these results, as it was done in the tests performed in section 1.4.1. We adopt the same approach done in the previous section, where all instances will be invoked by the execution with identifier 0 to mitigate our network fluctuations, latency and cold start during invocations.

To ensure a good partitioning of uploaded data in S3, we followed the recommendations explained in [116]. Therefore, each chunk has a unique prefix generated using an MD5 hash to ensure that the uploaded data is well distributed in the underlying S3 partitions. We used normal upload because *multi-part* upload method requires a minimum size of $5MB$ for each partition, except the last one, as AWS explains in [123]. So, even if all chunks form a unique results file, we cannot use *multi-part* upload for small parts.

All used Lambda functions were assigned $3008MB$ of RAM, the largest value available to date, to ensure that only one invocation of test Lambda function is executed in each VM. As in previous section, for each test, all Lambda invocations run the same Lambda function with same characteristics and code. In each test, we change the number of concurrent Lambda invocations and the number of chunks to upload to modify the workload.

Figure III.9 shows the results of a test with 10 test executions using 400 function invocations and 40 chunks. The whiskers of the box plot group 98% of the data while the rest of data are represented as points. The figure shows that most of the invocations conclude the upload step fast, but a very small fraction take a lot of time to finish the task. In addition, as we can see on different test executions, the time required by the slower invocations seems to be unpredictable.

On successive test executions, we checked that the slow invocations were not executed in the same VM. Some VMs host "fast" invocations on some executions and "slow" on others. Also, seeing the upload time spent in the "big chunk" step, one can observe that some slow invocations upload data at regular speed in this step. Probably this was because most Lambda invocations have finished all their uploads. Therefore, unlike the case described in subsection 1.4.1, this increment in upload time seems unrelated to the performance of the VM but still has a major impact in the global execution time.

The previous figure shows that the network behaviour has unpredictable fluctuations in a small subset of invocations. This fact shows that the slow network effect is not caused by a slow VM, but is caused by some kind of network saturation or resources assignation to each Lambda invocation. A similar behaviour can be observed testing the download time.

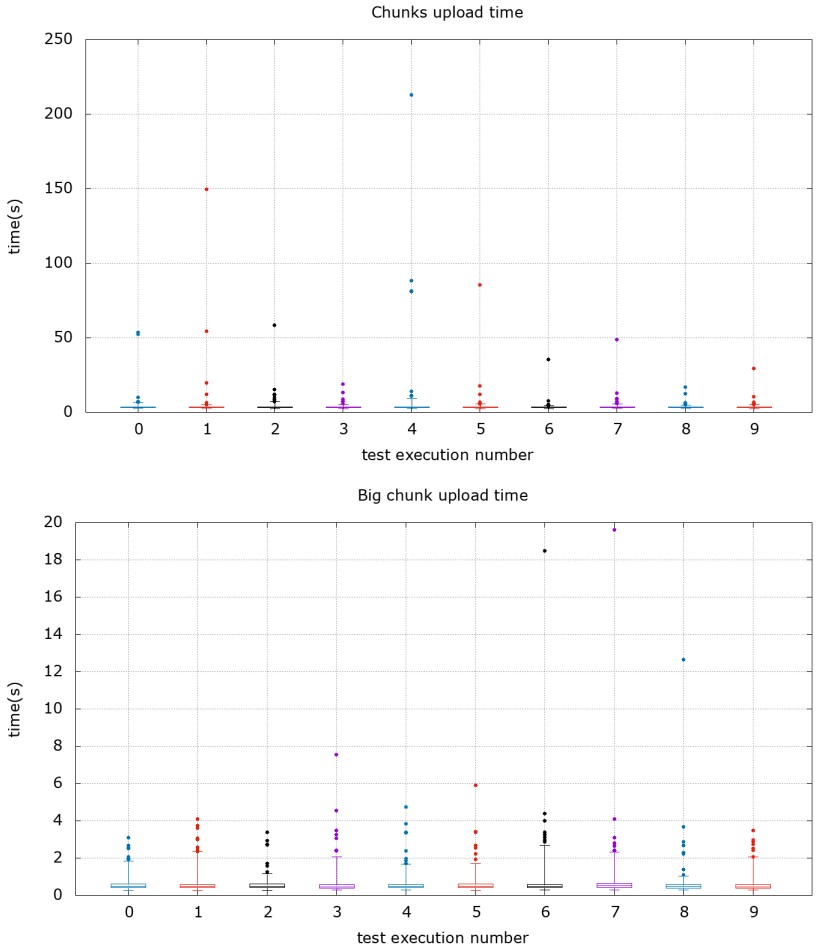

Figura III.9: Network test using 400 Lambda invocations with $3008MB$ and 40 chunks. 10 test executions were performed. The plots show distributions of time required by Lambda invocations to conclude the small chunks upload step (up) and the "big chunk" upload step (down). Whiskers group the 98% of data. Coloured lines are only included for convenience.

Several tests have been carried out using different number of concurrent invocations and chunks. Next, we show two tests: one with 100 invocations and 20 chunks (Figure III.10) and other with 20 invocations and 20 chunks (Figure III.11). Notice that test functions have similarity with the mapper functions in the MARLA architecture. The number of test function invocations would be the number of mappers and the number of chunks would be the number of reducers. So, we select the number of function invocations and chunks in each test to be reasonable values for our architecture while testing different workloads.

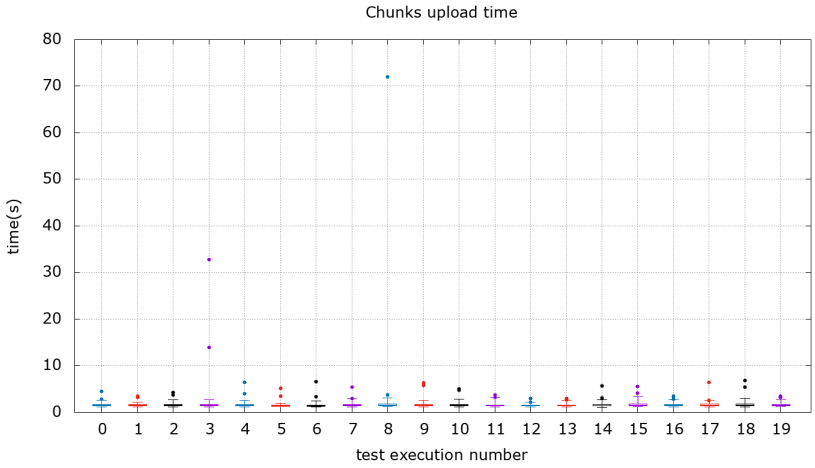

Figura III.10: Network test using 100 Lambda invocations with $3008MB$ and 20 chunks. 20 test executions have been performed for this test. Plots show distributions of time required by invocations to conclude the small chunks upload step. Whiskers group the 98% of data.

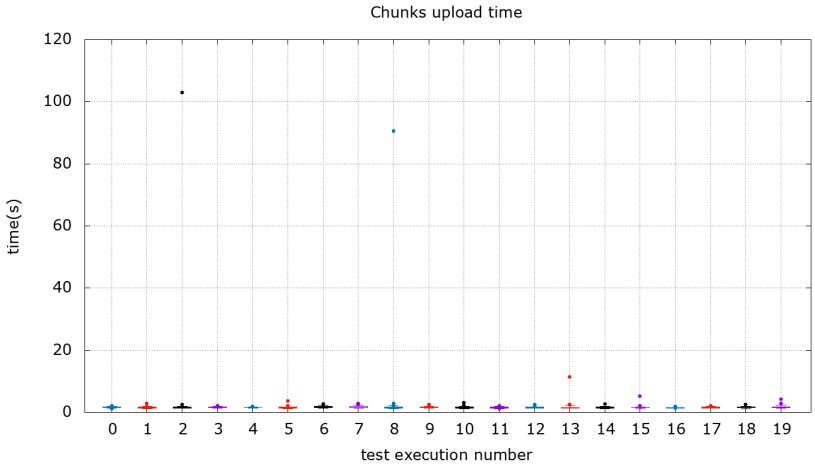

Figura III.11: Network test using 20 Lambda invocations with $3008MB$ and 20 chunks. 20 test executions have been performed for this test. Plots show distributions of time required by invocations to conclude the small chunks upload step. Whiskers group the 90% of data.

Previous tests show that the delay experimented in the upload/download speed of some invocations still appears with less Lambda invocations.

The reduce phase of a MapReduce-based application requires to wait until all mappers have finished before performing the reduction in order to obtain the

results. And this approach is also exhibited by typical job execution patterns that combine a loosely-coupled phase composed by multiple parallel independent jobs with a post-processing phase that requires the aggregation of results once they are available. Therefore, it is of paramount importance to understand and characterise the performance of opaque computing platforms such as AWS Lambda.

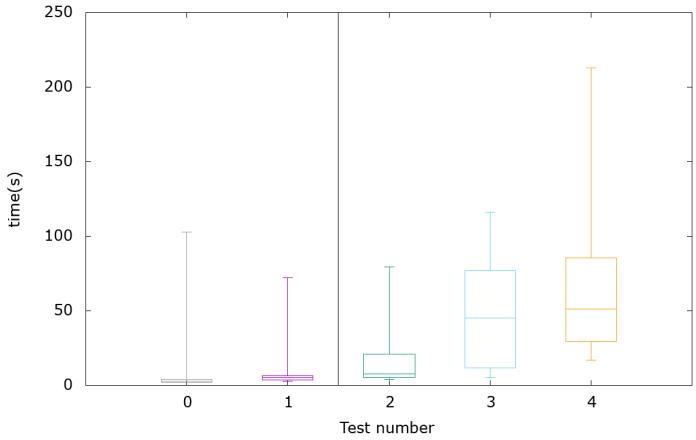

Figura III.12: Network performance assessment for Lambda function with $3008MB$. Represented tests use 20 and 40 chunks in the left and right of the vertical line, respectively. The number of simultaneous invocations is, in order, 20, 100, 100, 200 and 400. Plots show the distribution of required time, by the slowest invocation, to upload the corresponding number of chunks.

Figure III.12 shows a box-whiskers graph with the distributions of the slowest invocation upload time across five tests. The represented tests use 20 and 40 chunks in the left and right of the vertical line respectively. Each test has been executed 20 times and the number of simultaneously invocations is, in order, 20, 100, 100, 200 and 400. It can be seen that the maximum upload time, on average, will increase with the number of concurrent invocations uploading data, showing a possible saturation or limited network resources assignation on AWS Lambda function. Since many applications only need to upload a single results file per function invocation, we performed some tests with different number of concurrent invocations and only one chunk. The results are shown in Figure III.13.

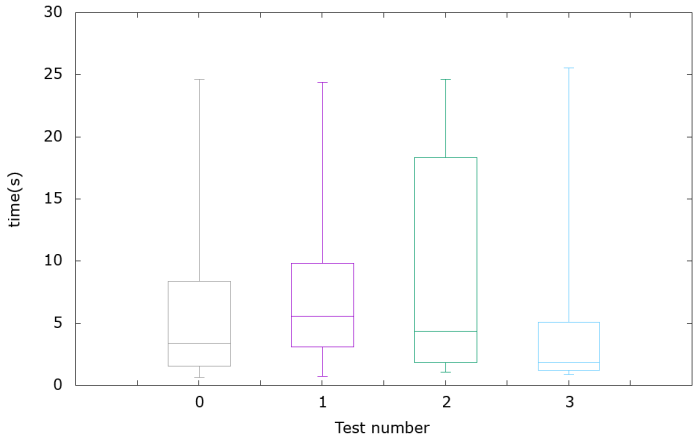

Figura III.13: Network test using 100, 200, 300 and 400 Lambda simultaneous invocations with $3008MB$ and a single chunk. Plot shows distribution of required time, by the slowest invocation, to upload 1 chunk. Each test has been repeated 20 times.

With 1 chunk tests we see that the mean of the longest upload time is not correlated with the number of invoked functions. To explain this result, we have to consider two factors, the time when Lambda invocations start to upload data and the invested time to perform upload tasks. On Figure III.14 we show the timestamps when invocations start to upload data on one test execution. The figure shows approximately a linear behaviour.

As expected, the linear behaviour of timestamps is repeated in all execution tests of all tests considered in Figure III.13, since the method to invoke Lambda functions is the same in all execution tests. To explain this results, we need to estimate the number of invocations that are using the network simultaneously in single executions. The mean of this value can be calculated knowing the time until another function invocation uses the network and the time used to perform the upload data:

$$SLI = \frac{UT}{slope} \tag{III.9}$$

where SLI is the number of simultaneous Lambda invocations using network, *slope* is the slope from linear fits and UT is the time inverted to upload data. The smaller is the *slope*, the more Lambda invocations will be using the network simultaneously. So, we use the minimum value measured of slope ($0.0498s$) to consider the worst case. Using the mean value of UT ($0.2271s$) measuring the upload times for all executions of all tests, we obtain,

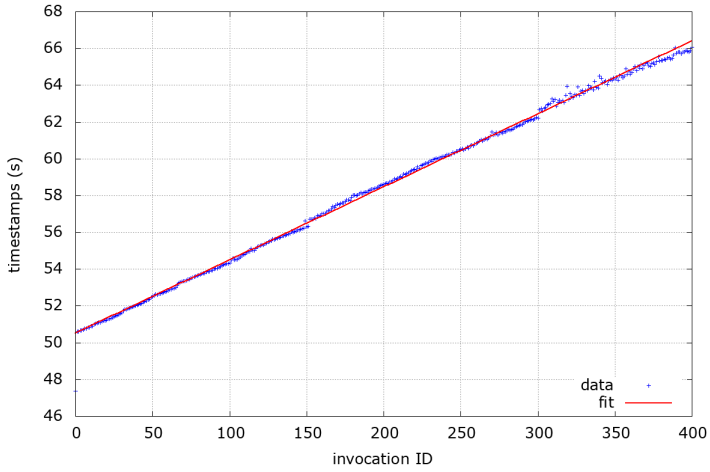

Figura III.14: Timestamps from the beginning of the upload data phase of execution test using 400 Lambda invocations with 1 chunk to upload. The timestamp origin has been shifted to avoid using values on the order of 10^9. However, only time differences between Lambda invocations are relevant. The plot represent measured data (blue) and linear fit of data (red).

$$SLI = 4.56 \tag{III.10}$$

This means that the first Lambda function invocation will finish its execution before invocation number 6 starts to upload data. So, on average only 5 invocations will be using the network simultaneously, no matter how many we use for the test. This result explains why the results of Figure III.13 seems to not depend on the number of Lambda function invocations. However, we still occasionally measure high upload times with few Lambda functions using the network simultaneously.

1.4.3 Isolated network usage performance

On the previous section we identified that fluctuations were most probably caused by simultaneous network usage of several Lambda invocations. However, for completeness, some additional tests where Lambda function invocations "sequentially" use the network will be carried out in this subsection. For this tests we have used the same Lambda function and configuration than in the previous section. However, in this tests, we ensure that only one Lambda function invocation is sending data to S3, that is, we try to avoid concurrent network usage. To this aim, two kind of tests have been performed:

- **Concurrent Execution, Sequential Upload**: All Lambda invocations will be done by the invocation with ID 0 with no delay. However, each

Lambda invocation will wait a time proportionally to its own ID using the Python *sleep* function. The time waited by each invocation is:

$$delay\,(seconds) = 6 \cdot (1 + ID) \tag{III.11}$$

where ID is the invocation identifier that is in the interval $[0, N_{invocations}]$. The +1 is used to avoid sleeps of 0 seconds. Because invocation with ID 0 spends additional time invoking all other ones, an additional delay has been set to it. That delay is,

$$delay_{ID=0} = 6 \cdot (N_{invocations} + 1) \tag{III.12}$$

With this approach, we force the invocation with $ID = 0$ to upload its data when all others have done it. The chosen delay unit (6 seconds) has been selected after testing the time required for one function to perform the upload step with 20 chunks. The total time spent by function with $ID = 1$ with previous configuration is, on average, 14 seconds, where 12 are from sleep ($6 \cdot (1 + 1) = 12s$). Therefore, approximately, only two seconds are required to execute the entire function. Tripling this value should be enough to ensure that previous invocation has finished. Thanks to the recently timeout increment on AWS Lambda functions (to 15 minutes), we can run 100 invocations safety for that test. Notice that this test forces Lambda invocations to run concurrently but avoid network sharing.

- **Sequential Execution, Sequential Write**: In that second kind of test, only one Lambda invocation will be running at once. The invocations will be done from our local machine in loop with the flag `-invocation-type RequestResponse`. So, only when a previous invocation finishes its execution, the next one will be invoked. This approach avoids any kind of concurrence and network sharing.

In any of the previous proposed tests the expected result should be small fluctuations on upload times but not significant ones. Notice also that, in these tests, the write steps on S3 are not concurrent. Both tests have been repeated using the *Reserve concurrency* option to ensure that all invocations can be done without reaching the concurrence limit. However, this appears to have no influence on the measured results.

For the "Concurrent Execution, Sequential Upload", two tests of 10 executions each one have been carried out. The results are shown in Figure III.15. The two upload steps reproduce the same behaviour, so only the "big chunk" step will be shown.

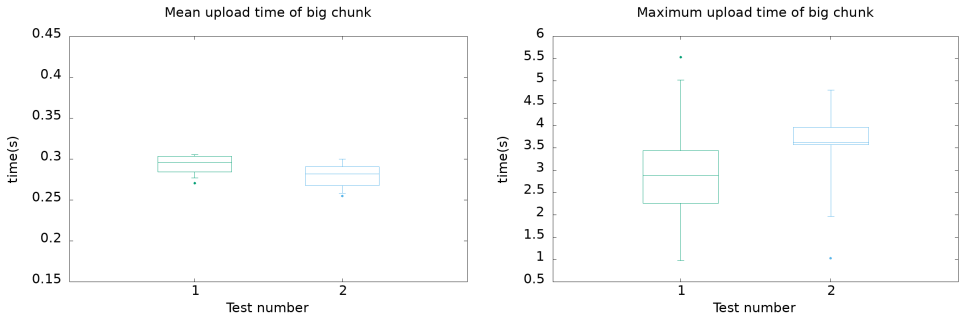

Figura III.15: Distributions of mean upload time (left) and maximum upload time (right) of big chunk step for test executions of each test. Each test has performed 10 executions with 100 Lambda invocations each one and 20 chunks. Whiskers include the 90% of all data so, one test execution is shown as a point.

As we can see, both tests show the same behaviour. The mean of upload times has small fluctuations. However, few invocations require an unusually high time to perform the upload to S3. This behaviour is repeated in both big and partitioned chunk steps. Therefore, the delay apparently produced by simultaneous uploads to S3 still appears on concurrent executions with "isolated" network usage. To compare with the results of simultaneous upload tests from previous section, Figure III.16 shows the results of Figure III.12, but only the test corresponding to 20 chunks and 100 invocations per test execution.

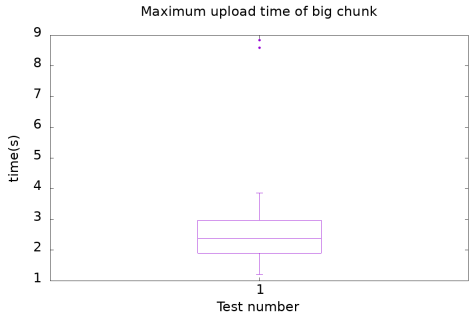

Figura III.16: Results of test 1 of figure III.12.

The tests performed in this section experiment a smaller delay than the one produced with simultaneous uploads. The behaviour, on average, of the slowest invocation is approximately the same in both simultaneous and isolated network usage tests. This results suggest that the delays are not produced by the number of invocations using the network but for the number of concurrent invocations. Notice that these results are perfectly compatible with the tests in Figure III.13.

As the invocations finish their execution before all others start, the number of concurrent invocations running is, approximately, constant. The same happens with the tests with more chunks and same number of functions or vice versa, the number of concurrent invocations is greater and the delay is greater too.

The next tests correspond to the "Sequential Execution, Sequential Write". Under these circumstances, the system cannot experiment any kind of concurrency, since all Lambda invocations will be executed sequentially. Three tests have been done with 20 chunks and 10, 100 and 100 invocations respectively and the results are shown in Figure III.17.

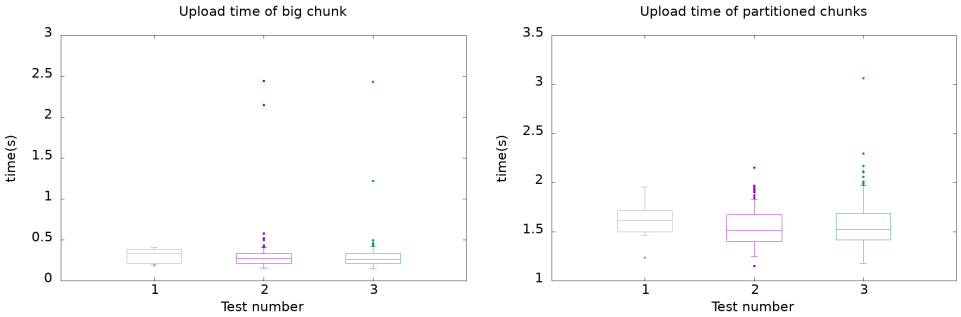

Figura III.17: Upload time distributions on network tests where only one invocation is executed at once. Each test uses 20 chunks and execute 10, 100 and 100 invocations respectively.

The results indicate that still exists this effect of “slow” uploads, even when only one function is invoked. Moreover, the time required by the slowest ones are in the range obtained in previous test (Figure III.15). Therefore, it seems that the invocation concurrency is not the cause of the experimented delays. The delays with no simultaneous network usage seems to be produced by the underlying and unknown behaviour of the infrastructure provided by AWS. However, the results indicate that the invocation concurrency accentuate this effect.

1.4.4 Mitigation Strategies

This section identifies certain usage patterns that can be adopted by applications running on AWS Lambda in order to mitigate the heterogeneous performance results obtained as a result of the performance assessment carried out in the previous section:

- **Prepare for the worst**: As seen in previous sections, VMs with very different performance can be used by AWS to automatically execute a Lambda function invocation even if the user allocated the same amount of memory.

Therefore, the user should ensure that the function invocation can run its code even when the slowest underlying VM is assigned in order to avoid time-outs.

- **Handle time-outs**: Users should assume that, eventually, a Lambda function invocation will cause a timeout provoked by a "slow network" data transfer. Therefore, the application workflow should correctly handle these errors. As an example, temporary data in S3 should not be deleted until the partial/final results have been uploaded to the corresponding permanent data storage.

- **Scalability**: According to the performance assessment carried out, applications that simultaneously upload/download data running on AWS Lambda will not scale well. However, if the application invokes Lambda functions in response to events distributed "randomly" (such as user requests) the number of functions that simultaneously use the network will be much smaller than an application where multiple functions are invoked at once, as it happens in our serverless-based MapReduce architecture. As seen before, this effect increases with the data size to transfer.

- **Avoid excessive partitioning**: Whenever possible, uploading/downloading big data chunks results in better performance that performing data transfers of a large number of small files into Amazon S3.

1.5 Conclusions and Future Work

This paper has introduced the MARLA (MApReduce on Lambda) framework in order to support serverless MapReduce on top of Amazon Web Services. The framework introduces additional optimisations with respect to the state of art regardless the automatic determination of mappers and the efficient usage of Amazon S3 for enhanced data transfers. When comparing the performance results with the serverless MapReduce reference architecture of AWS, we identified that AWS Lambda exhibits an heterogeneous performance behaviour that clearly has an impact on the execution time of MapReduce jobs. Therefore, single values for the execution time of an application in AWS Lambda cannot be used as a fair comparison mechanism. Instead, a careful analysis is required to provide statistical dispersion of the execution time.

To this aim, this paper undertook a thorough analysis of AWS Lambda in terms of CPU performance and network throughput with a special focus on identifying the reasons why this heterogeneity is exhibited. A correlation was obtained between the CPU model of the underlying EC2 instances on which the Lambda functions were running and their execution time. Therefore, the claim that the

performance of an AWS Lambda function is related to the allocated amount of RAM (chosen by the user) should be complemented with the fact that, depending on the underlying instance that actually executes the function (chosen by AWS), the function will exhibit execution times that can be up to 51% slower.

This poses a problem for coupled job executions, like MapReduce, in which a delay in just one the mappers affects the whole execution since the reduce phase cannot be started. This fluctuation in the execution can eventually cause timeouts in the Lambda functions and disrupt job executions if they have not been configured to accommodate this effect. This has also an economic impact since the pricing model of AWS Lambda depends on the amount of milliseconds executed, together with the RAM allocated to the Lambda function.

A similar effect was identified in the network performance. Multiple Lambda functions that put or get data in S3 buckets can produce, depending on the number of concurrent functions and size of data to store, a timeout on "slower" functions. However, with a good configuration of the application, the slow Lambda function will be re-invoked and, eventually, will finish its execution. Therefore, this effect can produce an increment in time and cost but the execution will be done eventually, as exemplified in our architecture.

Still, AWS Lambda provides an ideal computing platform in which automated scaling is handled by the Cloud provider and application developers can benefit from it for general purpose computing. Our presented architecture provides a general MapReduce platform completely executed in a serverless environment with a pay-per-use pricing model. This really stands out as a convenient approach to offer a MapReduce service that is pre-deployed as a set of Lambda functions at zero cost and the execution pipeline is triggered by uploading a file into a bucket, resulted in automated scaling to perform a MapReduce job tailored to the dataset to be processed in terms of the number of mappers.

Future works include extending the framework to support additional programming languages as well as other Cloud providers. In particular, we would like to study whether the fluctuations in AWS Lambda performance are present in the corresponding service offerings by other Cloud providers.

Acknowledgment

This study was supported by the program "Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d" under grant number ACIF/2018/148 from the Conselleria d'Educació of the Generalitat Valenciana. The authors would also like to thank the Spanish "Ministerio de Economía, Industria y Competitividad"for the project "BigCLOE" with reference number TIN2016-79951-R.

2 APRICOT: Advanced Platform for Reproducible Infrastructures in the Cloud via Open Tools

Referència: Giménez-Alventosa, V., Segrelles, J. D., Moltó, G., Roca-Sogorb, M. (2020). APRICOT: Advanced Platform for Reproducible Infrastructures in the Cloud via Open Tools. Methods of information in medicine, 59(S 02), e33–e45.
https://doi.org/10.1055/s-0040-1712460

Categoria JCR: COMPUTER SCIENCE, INFORMATION SYSTEMS

Posició: 105/161

Quartil: Q3

Factor d'impacte (2020): 2.176

El segon treball desenvolupat al present document pretén proporcionar eines per a la millora de la reproductibilitat en experiments computacionals, atacant així una problemàtica clau hui en dia en l'àmbit científic, tal i com s'ha discutit al capítol I, secció 4. A més, com objectiu afegit, es pretén facilitar la tasca de desplegament d'infraestructures computacionals complexes per al càlcul científic.

Per aconseguir ambdós objectius, l'estratègia seguida consisteix en desenvolupar una sèrie d'eines lligades a l'entorn *Jupyter* [83], que com s'ha discutit a la secció 4 del capítol I, proporciona, entre altres característiques, un entorn adient per a facilitar la reproductibilitat d'experimentació computacional. No obstant, per a experimentacions amb un alt cost computacional que requerisquen una infraestructura més complexa, com per exemple un *cluster* MPI, l'entorn *Jupyter* no proporciona totes les característiques necessàries per a mantindre tota la experimentació inclosa a un *notebook*, considerant el desplegament de la infraestructura, gestió de les dades, execució de l'experimentació i alliberament dels recursos.

Per aquest motiu s'han combinat diferents eines amb tasques específiques per integrar en *Jupyter* tot el procés de selecció, desplegament i ús de infraestructures computacionals complexes a l'hora que es dona suport per per compartir dades a través d'un proveïdor obert d'emmagatzemament. Els components emprats es descriuen a continuació, classificats per la seua funcionalitat.

- **Interacció amb l'usuari**: Com ja hem dit, per tal de controlar les interaccions amb l'usuari, s'emprarà l'entorn de *Jupyter*, per al qual es desenvoluparan una sèrie de *plugins* i funcions per a controlar la resta de components al mateix entorn.

- **Desplegament d'infraestructura**: El desplegament d'infraestructura serà gestionat per el programari IM [89] i EC3 [124]. El IM és un desenvolupament de còdi obert que permet el desplegament d'infraestructures complexes i virtualitzades en proveïdors de IaaS, tant públics com privats. A més del desplegament, automatitza la configuració, instal·lació de programari i monitorització i actualització de les infraestructures. D'altra banda, el programa EC3 proporciona funcionalitat extra al IM permetent crear *clusters* virtuals elàstics, adaptant el nombre de nodes a la càrrega de treball.

- **Elasticitat**: Per a que els *clusters* mencionats anteriorment escalen segons la càrrega de treball, s'emprarà CLUES [125], el qual gestiona l'aprovisionament dinàmic de recursos als *clusters* desplegats segons la càrrega de treball. A més, elimina aquests recursos quan ja no son necessaris.

- **Emmagatzemament**: Per tal de proporcionar les dades requerides per a permetre la reproductibilitat d'una experimentació, és necessari l'ús d'un proveïdor d'emmagatzemament extern. En *APRICOT*[4], s'ha elegit Onedata [126], que proporciona accés global a un sistema d'emmagatzemament distribuït destinat a l'ús científic. A més, Onedata s'empra al EGI DataHub [127], el qual conforma, actualment, la infraestructura computacional distribuïda més gran d'Europa.

L'ús dels esmenats components resulta, en general, complex i costós, el que limita el seu ús per part de la comunitat científica. L'objectiu d'*APRICOT* és el de gestionar tota aquesta funcionalitat de la forma més transparent possible a l'usuari per a que aquest únicament s'haja de preocupar de descriure i executar la seua experimentació al *notebook*. De fet, per demostrar la seua utilitat, el treball annexat a continuació inclou dos exemples d'experimentacions científiques reals completament descrites als *notebooks* proporcionats al repositori d'*APRICOT*, demostrant així que s'han acomplert els objectius plantejats.

Abstract

Background:
Scientific publications are meant to exchange knowledge among researchers but the inability to properly reproduce computational experiments limits the quality of scientific research. Furthermore, bibliography shows that irreproducible preclinical researche exceeds 50%, which produces a huge waste of resources on non profitable research at Life Sciences field. As a consequence, scientific reproducibility is being fostered in order to promote Open Science through open databases and software

[4]APRICOT: `https://github.com/grycap/apricot`

tools that are typically deployed on existing computational resources. However, some computational experiments require complex virtual infrastructures, such as elastic clusters of PCs, that can be dynamically provisioned from multiple Clouds. Obtaining these infrastructures requires not only an infrastructure provider, but also advanced knowledge in the cloud computing field.

Objectives:
The main aim of this paper is to improve reproducibility in life sciences to produce better and more cost-effective research. For that purpose, our intention is to simplify the infrastructure usage and deployment for researchers.

Methods:
This paper introduces APRICOT, an open-source extension for Jupyter to deploy deterministic virtual infrastructures across multi-Clouds for reproducible scientific computational experiments. To exemplify its utilization and how APRICOT can improve the reproduction of experiments with complex computation requirements, two examples in the field of life sciences are provided. All requirements to reproduce both experiments are disclosed within APRICOT and, therefore, can be reproduced by the users.

Results:
To shown the capabilities of APRICOT, we have processed a real magnetic resonance image to accurately characterize a prostate cancer using a MPI cluster deployed automatically with APRICOT. In addition, the second example shows how APRICOT scales the deployed infrastructure, according to the workload, using a batch cluster. This example consists of a multi-parametric study of a positron emission tomography image reconstruction.

Conclusions:
APRICOT's benefits are the integration of specific infrastructure deployment, the management and usage for Open Science, making experiments that involve specific computational infrastructures reproducible. All the experiment steps and details can be documented at the same Jupyter notebook which includes infrastructure specifications, data storage, experimentation execution, results gathering and infrastructure termination. Thus, distributing the experimentation notebook and needed data should be enough to reproduce the experiment.

2.1 Introduction

Scientific publications are intended to share knowledge to be used by other researchers on their experiments. However, most publications do not provide the necessary information to verify and reproduce its results [128]. Moreover, Freedman [129] claims that "the cumulative prevalence of irreproducible preclinical research exceeds 50%", and Monya Baker [6] shows that more than 90% of researchers consider that a reproducibility crisis exists in scientific publications. Non-reproducible

science specially affects Life Sciences research, since these results should not be reused unless they can be trusted in order to avoid consequences on people's health. This results in a huge waste of resources on non-profitable research [129].

As a result, scientific reproducibility has lately become a topic of interest for researchers and institutions which defend an open and reproducible science model. This aims at solving two old problems in scientific research: non-reproducible investigation and fraud. With that purpose, the European Commission introduced Open Science to "promote a new approach to the scientific process based on cooperative work and new ways of diffusing knowledge by using digital technologies and new collaborative tools" [8]. In fact, European institutions consider that Open Science is a necessary factor for future research programs. In 2015, the EU Commission set three main goals for future research in the EU: Open Science, Open Innovation and Open to the world [9, 8]. To promote these goals, the EU is promoting the European Open Science Cloud (EOSC), which "aims to create a trusted environment for hosting and processing research data to support EU science in its global leading role" [10]. Open Science is not only an European objective, but also for institutions around the world. For example, in USA, several institutions promote initiatives for Open Science, like the "Berkeley Initiative for Transparency in the Social Sciences" [11], the Public Library Of Science (PLOS) [12] and the "Center for Open Science" [13].

We focus in this contribution on the reproducibility of computational experiments, where the two basic components required are *data storage* and *computing infrastructure*. The former, data storage, provides a place to store the experiment input, intermediate and resulting data, analysis code and software, documentation, etc. The latter, computing infrastructure, is where calculations are carried out according to the software and hardware requirements. This requires access to some local or external storage in order to stage in the required data for experimentation.

A general computational experiment workflow is represented in Figure III.18, where the *input data production* can be experimentally measured or simulated data, configuration values for simulations, input data files, etc. Then, the *data storage* can be provided not only by local storage but also by public Cloud providers or open-source storage platforms, such as Onedata [126]. To be able to reproduce an experiment, the required input data, software/code and documentation must be accessible to other researchers via some external and persistent storage. Finally, it is at the *Computational infrastructure* where the input data is processed to produce the final result. This infrastructure largely depends on the computational experiment and, therefore, it can be composed of a single computer, a cluster of computing nodes, a set of clusters at different Cloud providers, etc. In any case, the computational infrastructure requires a data storage to get input data and save both intermediate data, if needed, and the final results.

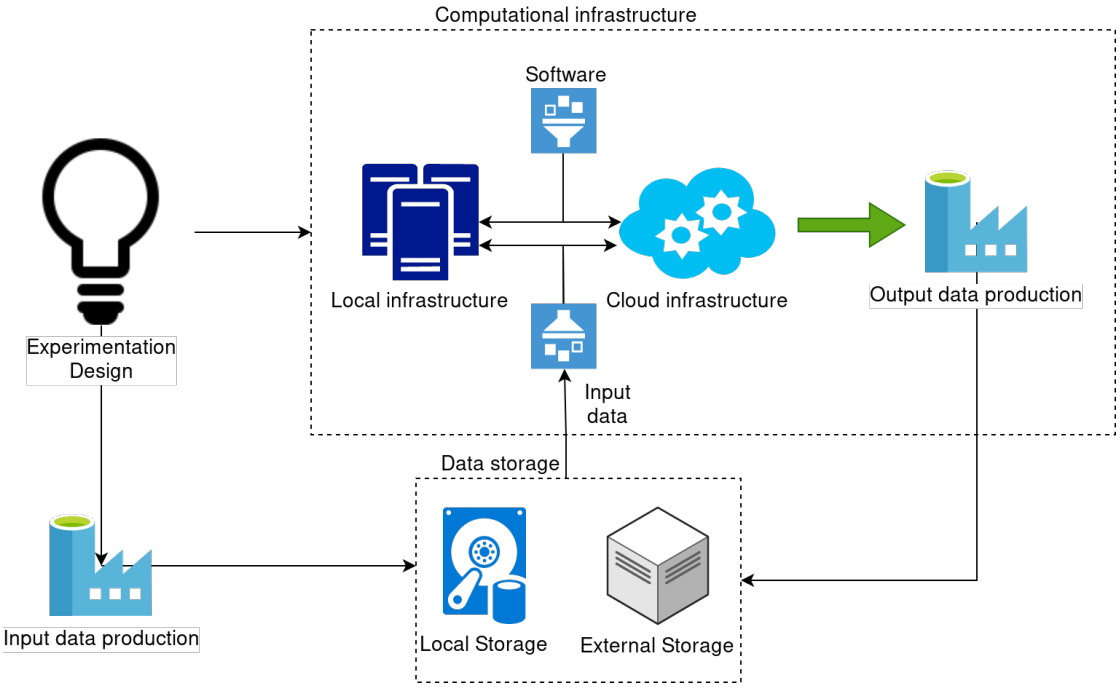

Figura III.18: Computational experimentation components.

Also, the infrastructure requires some configured computational environment to perform the analysis, e.g. specific software. Some examples of research experiments in the field of Life Sciences following this pattern are described in the work by Chillarón et al. [130], which takes Computed Tomography (CT) measured data to construct a medical image with few projections. Other examples include the work by Reader et al. [131], which presents an algorithm to reconstruct Positron Emission Tomography (PET) images with only one iteration over measured PET data and in the work by Giménez-Alventosa et al. [132], that uses radioactive sources specifications as input and perform brachytherapy simulations to discuss the error produced by approximations used at cancer treatment planning.

To achieve scientific reproducibility not only the data should adopt the FAIR principles (Findable, Accesible, Interoperable and Reusable) [133] but also the software involved should follow similar principles. Software should be available in a public online repository that can be reached by means of a web search engine (Findable). It should be available under an open, non-restrictive, software license to allow free adoption and modification to fit different purposes (Accessible). It should provide APIs (Application Programming Interfaces) and CLIs (Command-line Interfaces) using best-practices widely established in the software development communities in order to ease the integration among multiple software services (Interoperable). It should have a proper documentation that allows newcomers

to start adopting the software and find areas of applications that perhaps even the original developer team had never thought of (Reusable). The computational requirements to reproduce the experiment should also follow the same principles for scientists to self-deploy them in order to easily carry out the executions on a similar execution environment to guarantee a successful execution.

Even though some computational experiments may run on a regular computer, others may require a significant computing effort or specific customization, thus requiring a complex infrastructure. This is usually provisioned from private or public Cloud providers. However, the deployment and configuration can involve a significant effort by the researcher and advanced technical knowledge. This increases the difficulty not only for researchers to reproduce published experiments, but also for reviewers, who cannot afford to spend the required time to reproduce the whole experiment.

Recently, many new platforms have been created to promote and facilitate the adoption of Open Science for researchers. These platforms offer different services such as Cloud infrastructure to execute code in certain programming languages [81, 82], article journals and data repositories [134, 135, 126] and shareable programming environments [83, 136]. However, a lack of functionality stands out in these platforms related to the dynamic computational infrastructure deployment. Some experiments involve significant computational effort with specific distributed infrastructures or accelerated hardware devices, such as MPI clusters or GPGPUs. Current Open Science platforms cannot be used to reproduce these kind of experiments without a significant effort by the researcher and advanced technical knowledge such as infrastructure provisioning from multi-Clouds, understood as independent infrastructure deployments at different cloud providers, interconnection of nodes and configuration of applications, job submission, etc.

To address this lack of functionality, this paper introduces APRICOT (Advanced Platform for Reproducible Infrastructures in the Cloud via Open Tools), an extension for Jupyter [83] notebooks to automatize the deployment, usage and lifecycle management of virtualised computational infrastructures in multi-Clouds. We selected Jupyter because of its flexibility, ease of usage, capacity of module integration and open-source philosophy. APRICOT supports automatic deployment on multi-clouds via a Jupyter plugin that uses both EC3 (Elastic Cloud Compute Cluster) [124] and IM (Infrastructure Manager) [89] in order to automatically provision and configure resources from multiple Cloud back-ends.

2.2 Objectives

The main aim of this paper is to improve reproducibility in the life sciences field to enable the production of better and more cost-effective research. For that purpose, our intention is to facilitate the infrastructure usage and deployment for

researchers. Aimed at simplifying the researcher's technical effort, APRICOT offers a set of predefined cluster topologies such as "Batch-Cluster" or "MPI-Cluster". These are automatically configured with common utilities like a Local Resource Management System (LRMS), shared home via NFS, a set of compilers, etc. and specific ones such as MPI libraries. Thus, researchers only need to specify their (temporary) credentials to provision resources from a Cloud, the infrastructure topology, the number of nodes and their characteristics (number of CPUs, memory, etc). Data management can be achieved via SSH and it also integrated with Onedata [126] to achieve on-demand caching and access to distributed datasets across providers.

APRICOT combines the computational narrative provided by Jupyter notebooks with the dynamic provision, and integrated usage, of virtual infrastructure from a Cloud platform. This allows to create executable papers for reproducible science for research that involves specific computing requirements that exceed those available in the researcher's computer.

2.3 Methods

This section has been divided in two major parts. First, we will discuss the previous related work on reproducible science to motivate the need of our application. Then, we will discuss APRICOT's architecture and how researchers can adopt this tool to simplify and make reproducible their computational experiments.

2.3.1 Related work

The main tools for reproducible science can be divided in two big blocks: those focusing on data storage and those focusing on data processing. The latter is not restricted to provide a physical infrastructure where the application is run but also include tools that provide common and shareable computing environments.

On the one hand, storage-oriented tools focus on the persistence of different kinds of elements for reproducible science where the researcher can upload the required data to reproduce a published experimentation. Also, we can find platforms to upload scientific research papers, open source code, etc. On the other hand, processing platforms provide physical infrastructure for data processing, platforms with implemented open source algorithms to be used by researchers, workflow pattern tools and analysis workflows for specific problems, such as image recognition, etc. As we will see, most of them focus on creating easy-to-use shareable computational environments to promote reproducibility. Few of them provide simplified methods to deploy and configure virtualized computing infrastructure.

We can find in the literature code repositories for scientific reproducibility such as RunMyCode [81], an online code and data repository associated with scientific

publications (articles and working papers). In this regard, the Open Science Framework (OSF) [137] helps sharing across institutions documents, materials and data. Also, SEEK [138] is a web-based catalog for sharing scientific research datasets, models or simulations, processes and research outcomes. These platforms do not provide the ability to remotely execute the code but rather to be downloaded for local execution. This prevents applications with specific computing requirements to be executed.

Scientific journals are starting to support reproducibility of results to some extent. This is the case of IPOL [134], a research journal of image processing and image analysis. In IPOL, each article contains a description of the published algorithm and its source code, an online demonstration and a set of reproducible experiments. Text and source code are peer-reviewed, ensuring articles are truly reproducible. However, this is restricted to the topic of the journal: image processing.

There also exist open-source tools such as the Galaxy project [136], an open source workflow engine aimed at creating rapid and reproducible analyses that runs on an underlying infrastructure and can be used via a web browser. These workflows can be shared as documented experiments. This tool provides a common environment for researchers to work, facilitating software configuration and infrastructure usage. Another platform with similar features is the reusable and reproducible research data analysis platform called REANA [139]. REANA provides an environment to structure research input data and code using containerised environments and computational workflows allowing to instantiate and run the whole experiment on remote compute clouds, facilitating infrastructure usage. Experiments structured to be used with REANA can be easily reusable with the same input data, parameters and code or changing them to obtain different results. These characteristics make REANA a suitable environment for Open Science in computational experiments. However, the infrastructure where Galaxy or REANA is installed must be deployed and configured in advance.

Jupyter [83] is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations and narrative text. Documents, named "notebooks", can be easily shared and are used to create and share a complete experimentation workflow. JupyterHub provides multitenant access to Jupyter notebooks, introducing the ability to spawn notebooks, for example, on a Kubernetes cluster or on public Cloud platforms. However, this automated provision of resources is limited to the execution of the Jupyter notebook. The provision of additional complex virtual infrastructures to support the execution of the computational experiment described in the notebook is responsibility of the user.

The aforementioned tools require external computing infrastructure to run the

code. To alleviate this need, there exists completely managed reproducibility platforms, such as CodeOcean [82] a cloud-based computational platform where researchers can define a compute capsule that includes code and data together with the specification of a computational environment based on Docker. Users can execute all the published code without installing software on local computers. However, the free tier is limited and a subscription is required to unlock additional features. Managed platforms such as Stencila [140] allow researchers to create interactive, fully reproducible documents using familiar visual interfaces based on spreadsheets, thus restricting its applicability to cell-based calculations.

Existing reproducible science tools cover mainly the storage needs to provide easy and shareable environments for open science, as is the case of Jupyter notebooks or Galaxy instances. However, they do not provide a simplified method to deploy, configure and use a personalized computing infrastructure from these environments. Even though some platforms like CodeOcean offer the possibility to execute code on a per-subscription basis, this approach is unfeasible for all kind of computational experiments and requires the users to lock-in to the platform. As mentioned before, some experiments involve significant computational effort and specific infrastructure characteristics, usually provisioned from private and public Cloud providers, which require researchers to deploy a specific infrastructure and software configuration. Although there are tools to simplify the infrastructure deployment on multi-cloud platforms, they are not integrated in easy-to-use environments to encourage the use of these kind of infrastructures for reproducible science.

To address this issue we introduce APRICOT, an open-source platform that extends Jupyter notebooks with the ability to self-provision customised virtual computing infrastructure from multi-Clouds in order to execute the applications resulting from the simulation to be reproduced. We selected Jupyter notebooks as the base tool because of its flexibility, usage facility, capacity of module integration and open-source philosophy. This allows to combine in a single Jupyter notebook the description of the experiment with the ability for the user to self-provision the customised complex computing infrastructure required to reproduce the results of the computational experiment.

2.3.2 APRICOT architecture

APRICOT[5] is an open-source extension to support customised virtual infrastructure deployment and usage from Jupyter notebooks. It allows multi-cloud infrastructure provisioning using a wizard-like GUI that guides the user step by step through the deployment process. For that purpose, APRICOT integrates several

[5]APRICOT: `https://github.com/grycap/apricot`

already existing open-source components, which are summarised in Figure 2. Each one performs a specific task to facilitate the infrastructure deployment on cloud providers. However, their standalone usage still requires advanced computational knowledge. Therefore, APRICOT has been designed to handle the communication between all these components and the user, automatising the infrastructure deployment and fostering its usage among non-specialised users. Next, we will briefly explain these components and their specific role:

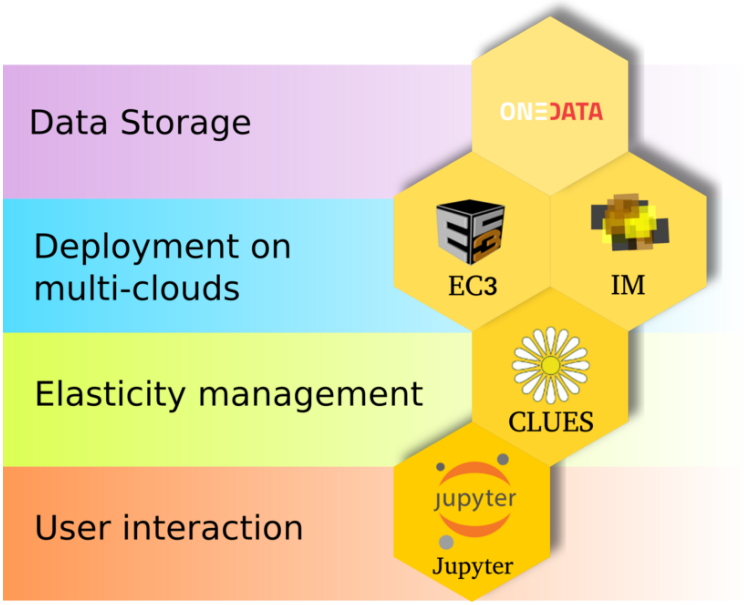

Figura III.19: Open source components used by APRICOT.

- IM (Infrastructure Manager) [89] is an open-source tool that deploys complex and customized virtual infrastructures on Infrastructure as a Service (IaaS) Cloud deployments, either public (such as Amazon Web Services, Google Cloud or Microsoft Azure) or on-premises (such as OpenNebula and OpenStack). It automates the deployment, configuration, software installation, monitoring and update of virtual infrastructures.

- CLUES (CLUster Elasticity System) [141] is an elasticity system for High Performance Computing (HPC) Clusters and Cloud infrastructures. Its main function is to deploy cluster nodes on the Cloud when they are needed via dynamic provisioning and automated integration in the LRMS and, conversely, to terminate them when they have been idle longer than a certain time.

- EC3 (Elastic Cloud Computing Cluster) [124] is a tool to create elastic virtual clusters on top of IaaS Cloud providers through the IM. To configure infrastructures, EC3 uses an infrastructure description language named RADL (Resource & Application Description Language) and, therefore, RADL files will be used by APRICOT. Multiple independent configuration files can be used in a single infrastructure deployment to create a complex configuration recipe.

- Onedata [126] is a global data access solution for science that provides access to distributed storage of scientific datasets, with automated caching and the ability to use multiple storage back-ends. It can be used to store and share the data required to reproduce the experiments.

APRICOT's architecture aligns with the vision of the EOSC (European Open Science Cloud) on the adoption of open-source tools that can interoperate with the federated Cloud infrastructures such as the one managed by EGI (European Grid Infrastructure). To this aim, both the IM and EC3 components were adopted, which are already integrated in the EOSC Marketplace [28] and are able to provision resources from the EGI Federated Cloud [142].

As shown in Figure III.19, user interaction is managed via Jupyter notebooks. Then, the infrastructure deployment uses EC3 [124] and IM [89] to support automated provisioning of computational resources from multi-clouds. APRICOT provides its own Jupyter "magics" to manage the deployed infrastructure via the EC3 client, to perform data upload and download, to execute tasks, etc. These can be used in any kernel with "magic" commands support, thus being compatible with many programming languages in the Jupyter environment.

Concerning cluster elasticity, CLUES is automatically installed at the deployed virtual clusters in order to provision additional nodes when needed. Finally, Onedata is used as the default storage provider relying on an existing Onedata provider, such as those available to support the EGI Data Hub[6]. However, the user can adopt any external storage platform by installing the required client in the cluster front-end to upload and download data.

To develop a reproducible computational experiment using APRICOT, a researcher documents in a Jupyter notebook how the experimentation will be executed and how to obtain or generate the required data. Then, the researcher specifies the computing infrastructure requirements and topology, from a set of pre-defined topologies, which can be extended via additional RADL documents. Finally, all the commands to execute the experimentation should be documented and executed in the Jupyter notebook. To allow this requirement, APRICOT implements a set of IPython magic functions so that the executed instructions are actually carried out

[6]EGI Data Hub - `https://datahub.egi.eu`

in the deployed infrastructures, not in the execution environment provided by the Jupyter notebook.

Therefore, APRICOT is composed by an open-source Jupyter notebook extension, used to deploy infrastructures, and a set of IPython magic functions for infrastructure use and management. The complete architecture schema is shown in Figure III.20.

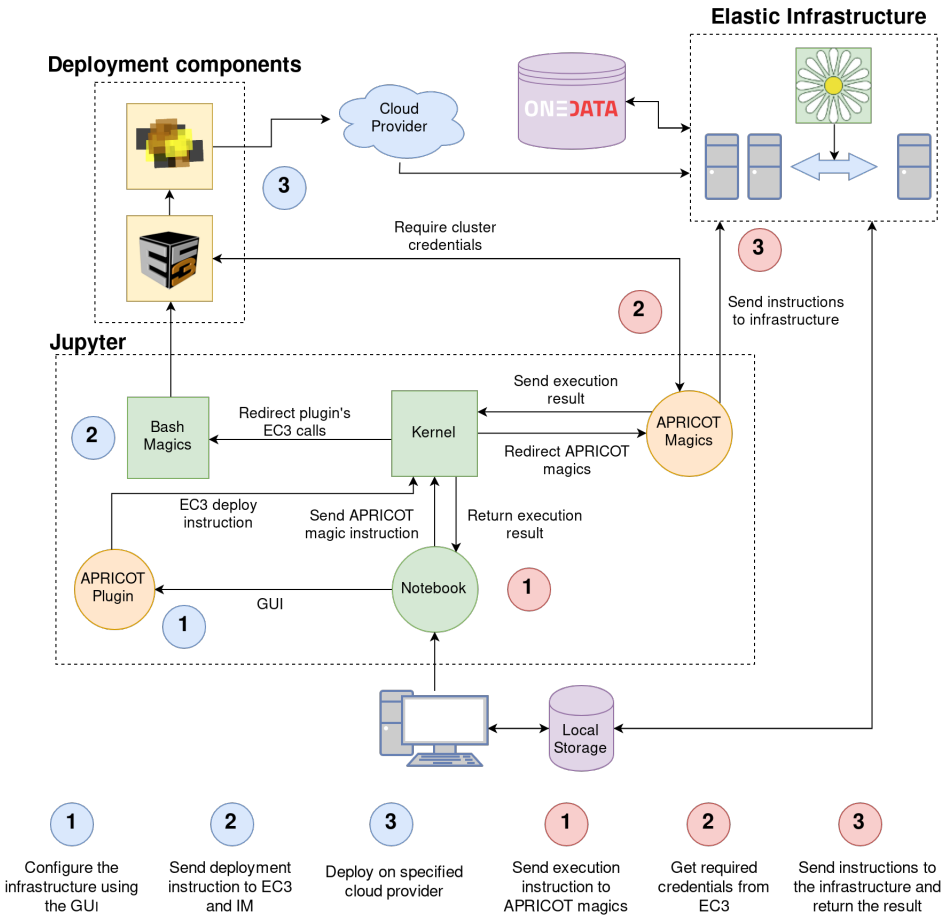

Figura III.20: APRICOT architecture.

First, APRICOT includes a graphical user interface (GUI) to guide the user on the infrastructure deployment process (Figure III.21 shows the deployment steps and Figure III.22 shows screenshots of the GUI). The configuration includes the cluster topology, cloud provider, number of workers, etc. At the end of this step, the APRICOT plugin will instruct the EC3 client, using Bash Magic functions[7] to deploy the specified infrastructure. Then, EC3 will delegate on the IM the

[7]Bash Magics - `https://ipython.readthedocs.io/en/stable/interactive/magics.html`

provisioning of the virtual machines to the specified Cloud provider to provision the infrastructure. Access credentials are automatically generated and stored by EC3 and, thus, APRICOT will contact EC3 to gather the appropriate credentials to perform remote command execution via SSH on the front-end node of the provisioned cluster.

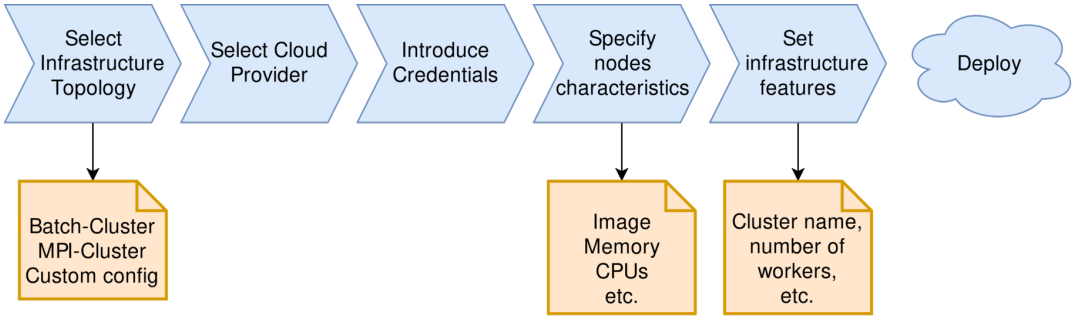

Figura III.21: Steps of the APRICOT deployment plugin.

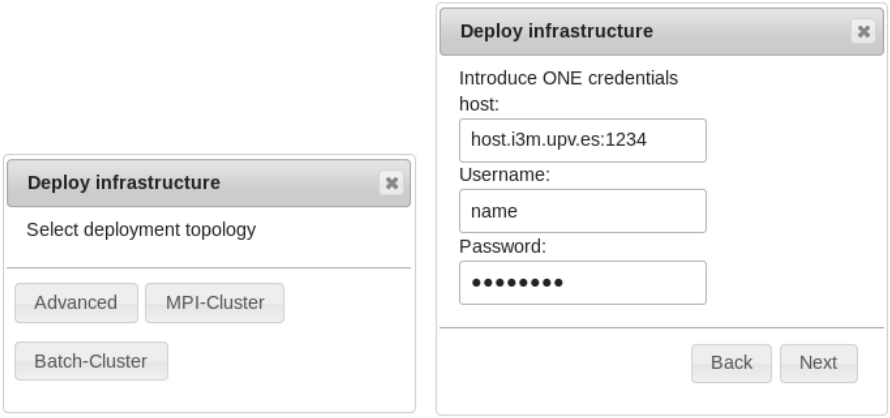

Figura III.22: Screenshots of the APRICOT deployment plugin.

To configure infrastructures, APRICOT uses a set of predefined RADL configuration files that describe the computing requirements, in terms of CPUs, RAM, disk space, etc. and include Ansible [143] roles to perform the unattended installation of software. The used RADL files depend on the selected topology by the user. However, it is possible to use the EC3 client to reconfigure an existing cluster using additional RADL files. Alternatively, users can choose the "Advanced" topology option to manually select the infrastructure configuration using a set of RADL files.

APRICOT Magics implement IPython magic functions to manage and use deployed infrastructures. Most of these instructions will be executed in the infrastructure front-end through remote SSH. Since Magics can be executed in all kernels with this support, APRICOT instructions can be executed in any language interpreted for one of these kernels. However, as it happens with all Jupyter non built-in Magics, APRICOT Magics must be loaded in each notebook before being used.

Notice that APRICOT just provides the infrastructure deployment, but the underlying computing infrastructure is not handled by APRICOT. So, if sensible data will be processed, the user must ensure that the infrastructure provider offers the required security measures. This is automatically enforced by major public Cloud providers. Additionally, the user may decide to adopt additional risk mitigation strategies such as data encryption both at rest and in transit.

Finally, a Docker image is being maintained to create a ready to use containerized Jupyter server with APRICOT installed and configured, available in Docker Hub[8].

2.4 Results

In order to assess the effectiveness of the developed platform, this section introduces two reproducible experiments which are distributed alongside this document, publicly available as Jupyter notebooks in GitHub[9]. The examples are installed in the provided Docker image for the convenience of the reader. These demonstrates the complete execution of a real magnetic resonance imaging (MRI) processing and a multiparametric analysis for medical image reconstruction using APRICOT to deploy, use and manage the required infrastructure. Therefore, this section represents a summary of the experimentation whereas further details can be obtained on the companion Jupyter notebook.

The aim of the first experimentation (MRI image processing) is to provide a real use case for our platform using a MPI cluster. On the other hand, the Positron Emission Tomography (PET) image reconstruction example uses simulated data and is intended to be used to evaluate APRICOT characteristics.

2.4.1 Infrastructure

To perform both experimentations, we used an on-premises Cloud managed by the OpenNebula [4] and the KVM hypervisor. The Storage Area Network is a Dell Equallogic PS4210 with 16 TB availables. The physical infrastructure is constituted by two type of nodes. The first one has 240 GB of Solid State Disk, 64

[8] `https://cloud.docker.com/u/grycap/repository/docker/grycap/apricot`
[9] `https://github.com/grycap/apricot/tree/master/examples`

GB of memory RAM, two Intel(R) Xeon(R) CPU E5-2683 v3 2.00GHz processors with 14 cores each one, two Ethernet network adapter of 1 Gbps and another one of 10 Gbps. The second node type has 250 GB of Solid State Disk, 128 GB of memory RAM, two Intel(R) Xeon(R) CPU E5-2660 v4 2.00GHz processors with 14 cores each one and thre Ethernet network adapters, two of 1 Gbps and a third of 10 Gbps.

On that on-premises Cloud we deployed two different topologies. For the MRI example, we used an MPI cluster formed by one front-end and three working nodes. All the nodes have been configured with the same characteristics: 1 CPU and $4GB$ of RAM with $20GB$ disk space. On the other hand, for the image reconstruction experimentation, we used a virtual elastic batch cluster with one front-end and two initial working nodes. Both the front-end and the working nodes have the same characteristics: two CPUs, 2 GB of RAM, and 20 GB of disk storage.

The OS image used on each experiment is a plain Ubuntu with version 16.04 and 18.04 LTS, configured using the "MPI-Cluster" and "Batch Cluster" option from the APRICOT deploy plugin respectively. Also, we recommend to configure the cluster so that all the working nodes are pre-provisioned at deployment time in order to avoid the delay introduced when nodes are dynamically provisioned when jobs are submitted to the LRMS.

To further illustrate the multi-Cloud features of the platform, we also reproduced the second experiment using the Amazon Web Services (AWS) public cloud provider with a cluster formed by one front-end and two working nodes. We used the "t2.small" (2 GiB and 1 vCPU) instance type to deploy the front-end node and two "t2.micro" (1 GiB and 1 vCPU) instances for working nodes. The cluster was deployed at the "us-east-1" region using an Ubuntu 16.04 AMI. Note that the selection of these instance types strictly respond to a cost-saving strategy aiming to illustrate the ability of the platform to deploy on a public Cloud. More powerful instance types would result in a significantly higher performance of the application.

All the required source codes to execute the experiment have been stored at the aforementioned GitHub repository alongside with the notebook to facilitate the reproducibility of this experiment. We stored input data in a public Amazon S3 bucket and not in Onedata to facilitate access for the readers, since no access token is required to retrieve the data.

The following section introduces both experiments and discusses the obtained results.

2.4.2 MRI image processing

Prostate cancer (PCa) is the second most frequent malignancy (after lung cancer) in men worldwide, counting $1,276,106$ new cases and causing $358,989$ deaths (3.8% of all deaths caused by cancer in men) in 2018 [144]. Early detection of

prostate cancer allows for appropriate management of the disease, and prognostic biomarkers can help clinicians make an appropriate therapeutic decision for each patient and avoid unnecessary treatment [145].

Due to recent progress in imaging, and particularly in MRI, the so-called multi-parametric MRI (mpMRI) that combines T2-weighted imaging (T2W) with functional pulse sequences such as diffusion-weighted imaging (DWI) or dynamic contrast-enhanced (DCE) imaging has shown excellent results in PCa detection and has become the standard of care to achieve accurate and reproducible diagnosis of PCa [146, 147].

Pharmacokinetic modeling of the DCE-MRI signal is used to derive estimates of factors related to blood volume and permeability that are hallmarks of the angiogenic phenotype associated with most cancers. The accuracy of DCE relies on the ability to model the pharmacokinetics of an injected tracer, or contrast agent, using the signal intensity changes on sequential magnetic resonance images.

The first pharmacokinetic model was proposed by Kety [148], who described flow-limited tracer uptake in tissue. This was followed by several pharmacokinetic models proposed by Tofts et al [149], Brix et al [150], and Larsson et al [151].

The majority of these models are based on the characterization of the contrast exchange rate between the plasma and the extracellular space through parameters such as K^{trans}, that represents the rate at which the contrast agent transfers from the blood to the interstitial space (indicating the tumor microcirculation), the reflux constant, K_{ep}, that reflects the rate at which the contrast agent transfers from the extravascular extracellular space back to the blood and the extravascular extracellular leakage volume fraction v_e, which predominantly reflects the percentage of contrast agent in the extravascular extracellular space.

The study of these parameters helps characterize prostate cancer, so estimating them accurately and robustly is a fundamental step. These parameters are calculated using the Tofts model [152], which is equivalent to the generalized kinetic model [153],

$$\frac{dC_t}{dt} = K^{trans}C_p - k_{ep}C_t \tag{III.13}$$

where interesting parameters are K^{trans}, which is the transfer coefficient between blood plasma and the compartment, and the extracellular extravascular fractional volume (EES) (v_e). Also k_{ep} is defined as $k_{ep} = K^{trans}/v_e$, and C_t is the concentration of lesson tissue defined as $C_t = C_1 v_e$, where C_1 is the leakage space concentration.

To solve equation III.13, we use the same approach as [154]. There, the model is restructured and expressed as a convolution as follows,

$$f(t) = K^{trans}\left(a(t) \otimes \frac{e^{-t/T}}{T}\right) \equiv K^{trans}\frac{1}{T}\int_0^t d\tau\, a(\tau)e^{-(t-\tau)/T} \qquad \text{(III.14)}$$

where $T = 1/k_{ep}$ and $a(t)$ function is an experimental measure, so is only available at discrete times. The previous model is evaluated for values of $T \neq 0$ by interpolating linearly between the measured values of $a(t)$. Instead, for $T = 0$ the result is $f(t) = a(t)$.

Our case study consisted on a prostate image with $256 \times 256 \times 56$ voxels and 30 time points for each one. We fitted this image using the equation III.14 implemented at the provided code in the APRICOT repository, which uses the ROOT libraries from CERN [155]. This analysis was performed using a MPI cluster with 3 working nodes, reducing the total computation time almost a factor 3 compared to the same experimentation performed on a single node. As a sample of the result, Figure III.23 shows four images that represent the resulting v_e map of four planes.

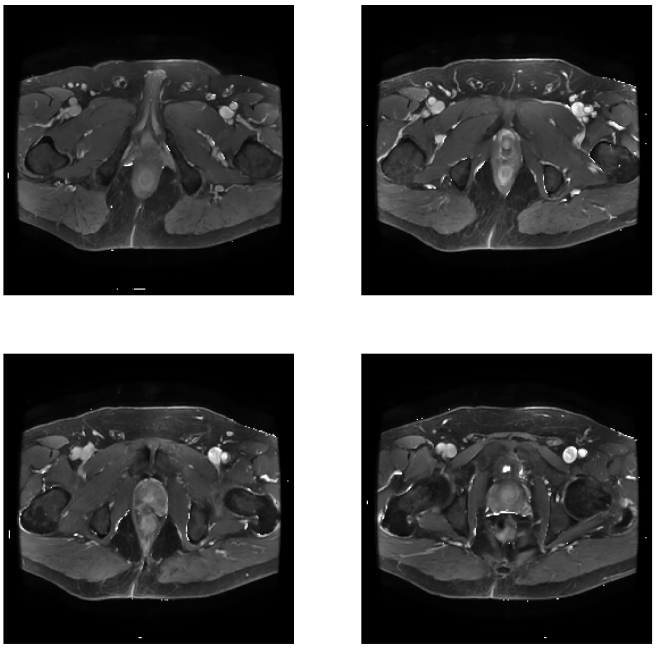

Figura III.23: MRI v_e maps for a real case of prostate image with $256 \times 256 \times 56$ planes with 30 time points on each one. Images corresponds to plane numbers 2 and 12 for top left and right images respectively, and 22 and 32 for bottom left and right images respectively.

2.4.3 PET image reconstruction

Medical scanners, like most physical detectors, measure raw data that must be post-processed to obtain an interpretable result. In particular, for medical scanners based in PET or Computed Tomography (CT), the final result is usually a patient image to be interpreted by the physician to develop a diagnostic.

Focusing on PET and CT cases, there exists a great variety of iterative and analytic image reconstruction algorithms [156, 131, 157, 130], most of them based on maximum likelihood method [158]. These reconstruction algorithms have a set of variable parameters such as number and size of voxels in the Field Of View (FOV), number of iterations, number of partitioned data chunks, weight parameters, filter iterations and weight, etc. Obviously, the final reconstruction quality and speed will depend on how accurate are the selected parameters. Furthermore, the accuracy of selected parameters depend on the scanner system (geometry, energy resolution, scanned object, etc.). Indeed, the importance of reconstruction parameters on medical image has been studied for different kind of scanners in many publications [159, 160, 161, 162].

Achieving the best parameters for our specific system and algorithm is desirable not only for medical diagnostics but to perform accurate comparison of reconstruction methods and scanner capabilities. That comparison is crucial to select and create new scanner systems using simulated data to study their theoretical performance. However, the number of possible parameters combinations grows as indicated in (3).

$$\prod_{i=1}^{n_{param}} N_i \ , \tag{III.15}$$

where N_i is the number of possible values of parameter number i and n_{param} the number of variable parameters. So, even performing a multiparametric study with few parameters requires a significant computational effort. APRICOT has been used in this experimentation to deploy and manage the required infrastructure to perform a multiparametric study on a modified implementation of "OPL-EM" reconstruction algorithm for PET systems described in the work by Reader et al. [131].

This algorithm uses a single iteration over all measured data to reconstruct the image, thus being faster than other iterative algorithms. As the example is aimed to focus on APRICOT usage, a simplified experimentation will be reproduced in the companion material, for the sake of better understanding. The provisioned infrastructure will consist on a cluster of PCs configured with CPU-based working nodes. The use of accelerated devices such GPGPUs can be achieved by provisioning the corresponding instance types in a public Cloud, provided that the

application supports using such specialised hardware. Therefore, this is agnostic to the functionality provided by APRICOT.

The workflow of these experimentation is shown in Figure III.24.

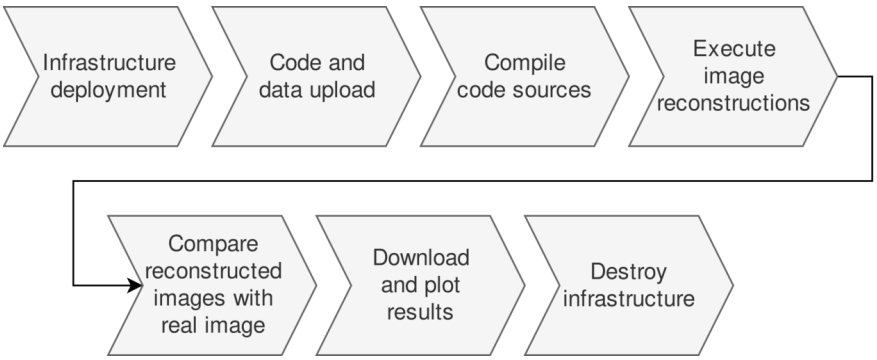

Figura III.24: Experimentation workflow.

The input data used for this experimentation has been obtained simulating a PET system formed by three rings of 20 detector modules each one. The simulations have been done using self developed routines to perform PET system simulations with the Monte-Carlo (MC) code PENELOPE [163], which accurately models photon, electron and positron interactions in an arbitrary material for the energy range of interest in this work. The simulation results include a file with all photon detection grouped by coincidences, which means that both photons have been produced by the annihilation of the same positron. This file, which is provided to perform the experimentation, will be our reconstruction input.

Once the reconstructions have been done, we extracted their execution times and image quality measures using different parameter combinations. To measure the image quality, we used the following metrics, *Root-Mean-Square Error* (RMSE), *Peak Signal to Noise Ratio* (PSNR), *Normalized Root Mean Square Distance* (NRMSD) and *Normalized Mean Absolute Distance* (NMAD), represented by equations III.16, III.19, III.17 and III.18 respectively. Regarding the notation, $v(m)$ denotes the voxel number m of the considered image and the subindex *true* indicate that is the real image.

$$RMSE = \sqrt{\frac{1}{N}\sum_{m=1}^{N}(v(m) - v_{true}(m))^2} \tag{III.16}$$

$$PSNR = 10 \cdot log_{10}\left(\frac{\max(v_{true})^2}{\frac{1}{N}\sum_{m=1}^{N}(v(m) - v_{true}(m))^2}\right) \tag{III.17}$$

$$NRMSD = \sqrt{\frac{\sum_{m=1}^{N}(v(m) - v_{true}(m))^2}{\sum_{m=1}^{N}(\bar{v}_{true} - v_{true}(m))^2}} \tag{III.18}$$

$$NMAD = \frac{\sum_{m=1}^{N}|v(m) - v_{true}(m)|}{\sum_{m=1}^{N}|v_{true}(m)|} \tag{III.19}$$

The RMSE tends to zero when the reconstructed image and the ideal one coincide, because voxels at both images are equal. So, small values should be interpreted as better image quality. Next, PSNR tends to infinity when reconstructed and ideal image become equal, because RMSE tends to zero. The NRMSD metric tends to 1.0 when the differences between images are smooth and the average intensity equal. Large differences on few voxels produce a large value of NRMSD. Finally, NMAD tends to 1.0 if the reconstructed image has negligible intensity on its voxels respect to ideal image, and tends to zero if both images are equal.

Some results of the run on our local infrastructure are shown in Figures III.25 and III.26 which represent, respectively, the time spent to reconstruct the images and the RMSE value of reconstructed images for different number of voxels in each axis using a data partitioning of 5 chunks.

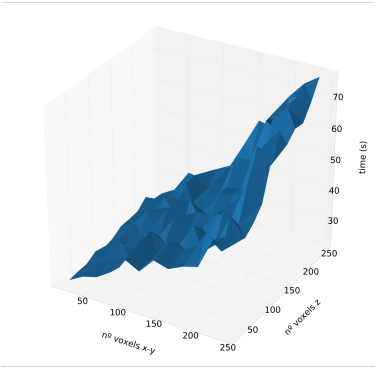

Figura III.25: Reconstruction time (s) using different number of voxels in each axis.

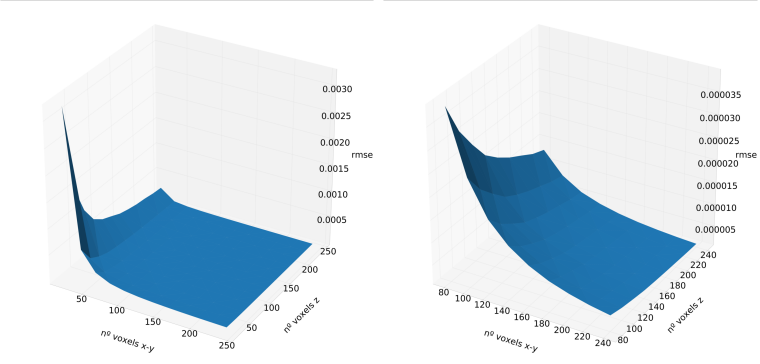

Figura III.26: Reconstruction error estimation using different number of voxels in each axis. The represented error has been obtained using the Root Mean Square Error (RMSE) method over all image voxels.

In a real case study, these results should provide the best parameters to achieve the required agreement between reconstruction speed and image quality.

2.5 Discussion

This section discusses about the platform deployment and elasticity. For that purpose, the second experimentation (image reconstruction) has been executed on two different platforms, according to the specifications in section 2.4.1, reproducing the very same results. The only difference lies on the location of the underlying resources and the time spent performing the reconstruction. The first one, deployed on our OpenNebula Cloud site, takes approximately 8 minutes to start and configure the front-end node and the same time for initial working nodes. On AWS, the deployment and configuration of the front-end node requires approximately 10 minutes and the same for the working nodes. So, both clusters require, approximately, $16 - 20$ minutes to be fully deployed and configured. Notice that working nodes are deployed concurrently. Therefore, deploying more working nodes will not cause a significant overhead on the configuration and deployment time. The main factor that affects the configuration time is the node CPU and network capabilities. Thus, using better instances should reduce the configuration time.

To show the elasticity capabilities provided by CLUES, we repeated the experiment using an infrastructure with the same specifications but configured with a minimum and maximum number of working nodes of 2 and 4 respectively. So, when the number of queued jobs exceed the number of available execution slots, CLUES deploys additional working nodes. To get statistics of the available and used slots we used CLUES reports, which monitors and extracts information about the use and state of our deployed infrastructure. Figure III.27 shows a slots usage

graph, where each node has a colour identifier and the grey colour represents idle slots. At the beginning of the graph (first grey zone) we can see when the first two nodes were configured and became ready to process jobs. Then, at the first coloured zone, both nodes were filled with the received jobs and CLUES detected more queued jobs than available slots. This caused CLUES to power on the two extra nodes (second gray zone). Once all nodes had been powered on and configured, at the second coloured zone, the number of available slots grew to 4 and the jobs execution was resumed. Finally, at the final gray zone, all the jobs had been processed and the working noes became idle. Therefore, using CLUES, our deployed infrastructures can be automatically scaled within the specified configuration parameters.

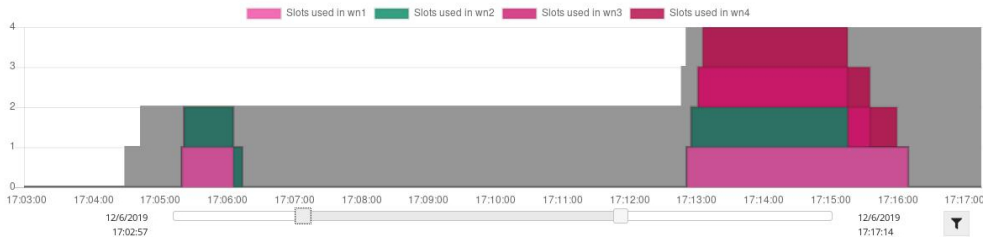

Figura III.27: Elasticity managed by CLUES. Cluster configured with 2 initial working nodes that can scale up to 4 nodes. The coloured zones indicate the number of execution slots being used while the gray zone indicates an idle slot.

As we have seen, APRICOT allows to easily deploy an scalable infrastructure to execute computationally intensive experiments. Furthermore, the same preconfigured infrastructure can be easily deployed to reproduce the whole experiment by other researchers or reviewers without knowledge on infrastructure deployment and configuration. Without APRICOT, a researcher would need to deploy and configure the required infrastructure or have access to an existing one with compatible specifications.

At the moment, APRICOT offers a limited amount of preconfigured infrastructure types that may not fit all experiments needs, but more configurations will be added in future versions. In addition, the user needs access credentials to a cloud provider supported by APRICOT.

Notice that the whole experiments have been documented in the corresponding Jupyter notebooks including the commands to execute, the required infrastructure, the data processing, the visualization, etc. The notebooks have been distributed in the aforementioned GitHub repository.

2.6 Conclusions

This paper has introduced APRICOT, an open-source extension for Jupyter that provides users with the ability to deploy complex customized virtual infrastructures across multiple Cloud providers to support the requirements of computational experiments. A set of functions have been created to simplify interaction with the virtual infrastructure for data staging as well as application execution. This facilitates the reproducibility of computational experiments on Clouds.

The benefits of this extension are the integration of specific infrastructure deployment, the management and usage for Open Science and making experiments that involve specific computational infrastructures reproducible. All the experiment steps and details can be documented at the same Jupyter notebook which includes infrastructure specifications, data storage, experimentation execution, results gathering and infrastructure termination. Thus, distributing the experimentation notebook and the needed data should be enough to reproduce the experiment.

Future works include extending APRICOT to use additional cloud providers already compatible with the IM. Also, in addition to MPI and batch clusters we plan to add more preconfigured infrastructure topologies such as Kubernetes clusters. Regarding the infrastructure usage, we will provide more *magic commands* to simplify the execution of other kind of analysis. These are special Jupyter functions that can be executed regardless of the programming language used at the notebook. Integration of additional queue systems and external storage providers will also expand the adoption of the platform by covering different use cases.

Acknowledgment

This study was supported by the program “Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d” under grant number ACIF/2018/148 from the Conselleria d’Educació of the Generalitat Valenciana and the “Fondo Social Europeo” (FSE). The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad"for the project “BigCLOE” with reference number TIN2016-79951-R and the European Commission, Horizon 2020 grant agreement No 826494 (PRIMAGE). The MRI prostate study case used in this article has been retrospectively collected from a project of prostate MRI biomarkers validation.

3 RUPER-LB: Load balancing embarrasingly parallel applications in unpredictable cloud environments

Referència: Vicent Giménez Alventosa, Germán Moltó Martínez, J. Damián Segrelles Quilis (2020). RUPER-LB: Load balancing embarrasingly parallel applications in unpredictable cloud environments. arXiv:2005.06361. `https://arxiv.org/abs/2005.06361`

Congrés: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND SIMULATION (HPCS)

Classe: CORE B

Notes: A data d'avui, els treballs presentats al congres encara no han sigut publicats.

Els inconvenients causats per la falta d'estabilitat de les prestacions en entorns de còmput compartit, especialment en entorns de còmput en el núvol (secció 3.2 de capítol I), fan patent la necessitat d'una eina que mitigue aquest efecte de forma eficient. Aquest efecte pot deteriorar en gran mesura les prestacions d'execucions paral·leles, especialment a l'àmbit científic, on són comuns els processos computacionalment costosos. Com es discuteix al treball, a la literatura es poden trobar nombrosos treballs relacionats amb el balanceig de càrrega d'aplicacions altament acoblades en entorns de computació en el núvol, les prestacions de les quals es veuen greument deteriorades degut, entre altres, a les altes fluctuacions en les prestacions de la xarxa. Malauradament, els algoritmes desenvolupats per a aquest tipus d'aplicacions no són convenients per execucions feblement acoblades, ja que introdueixen un sobrecost innecessari degut a les comunicacions i punts de sincronització involucrats. D'altra banda, els algoritmes típicament emprats per aplicacions amb acoblament feble es basen en adquirir un coneixement previ de les prestacions de la infraestructura. No obstant, si les prestacions fluctuen de forma significativa durant el temps d'execució, la predicció sobre el temps d'execució en cada maquinari no serà vàlida, donant lloc a desbalanceig i retards. És per aquest motiu que diferents treballs, com el de Weiling et al. [77] o el de Bittencourt i Madeira [164] divideixen les execucions en tasques independents relativament curtes, en comparació als canvis en les prestacions, i mesuren les prestacions del maquinari disponible de forma regular per a fer una distribució estàtica de dites tasques. Tot i que aquesta estratègia pot resultar útil per a moltes aplicacions, no es òptima per a qualsevol tipus, degut a diferents factors. Per exemple, les tasques derivades del particionatge del procés poden ser massa llargues, l'aplicació pot requerir una

inicialització amb un temps comparable a l'execució de cada tasca individual, si per exemple ha de realitzar càlculs sobre bases de dades, incorrent en un sobrecost significatiu, o si la quantitat de dades producte de les tasques individuals es relativament gran, els costos de comunicació podrien ser comparables al temps de processament.

Cal remarcar que les aplicacions amb aquestes característiques s'empren contínuament en l'àmbit científic i a enginyeria aplicada. Un exemple clar on un particionatge en tasques reduïdes no és òptim són les simulacions de Monte Carlo de transport de radiació, on, depenent de la simulació, el temps d'inicialització pot arribar a ser de l'ordre de minuts i les dades d'eixida de centenars de MBs o GBs. De fet, centrant-nos únicament a aquest tipus de simulacions, estes són considerades com l'estàndard a seguir als protocols internacionals per a tractaments oncològics basats radiació, tal i com descriuen el *Task Group report number 186* [165] de la *American Association of Physicists in Medicine (AAPM)* o el TRS 398 [166, 167] de la *International Atomic Energy Agency (IAEA)*, entre altres protocols. A més, també tenen aplicació a la industria, com el disseny d'escàners per a la industria alimentària o protecció radiològica en centrals nuclears i altres installacions radioactives. Per tant, el desenvolupament d'una tècnica per balancejar la càrrega en infraestructures amb capacitats fluctuants, on s'executen aplicacions poc acoblades i per a les quals la partició en tasques curtes no siga òptima tindrà un impacte més que notable.

Donada la necessitat plantejada, al tercer treball presentat s'ha desenvolupat un balancejador de càrrega escalable i eficient dissenyat expressament per aquest tipus d'infraestructures compartides i aplicacions, anomenat RUPER-LB. El desenvolupament de l'eina és públic, de codi obert i es pot descarregar del repositori de GitHub corresponent[10]. Als resultats es demostra com RUPER-LB és capaç de mitigar de forma precisa i introduint un sobrecost negligible les fluctuacions en les prestacions de la infraestructura. Per a dur a terme les proves, en primer lloc, s'ha emprat un entorn controlat en una infraestructura local on les fluctuacions s'han creat de forma artificial. En segon lloc, s'han realitzat proves sobre un entorn real com és el servei EC2 de AWS. Aquest últim ha servit, a més, per a evidenciar una vegada més les fluctuacions en les prestacions d'aquests entorns.

Abstract

The suitability of cloud computing has been studied by several authors to run scientific applications. However, the unpredictable performance fluctuations in these environments hinders the migration of scientific applications to cloud providers. To mitigate these effects, this work presents RUPER-LB, a load balancer for

[10]RUPER-LB: `https://github.com/PenRed/RUPER-LB`

loosely-coupled iterative parallel applications that runs on infrastructures with disparate computing capabilities. The results obtained with a real world simulation software, show the suitability of RUPER-LB to adapt this kind of applications to execution environments with variable performance and highlight the convenience of its adoption.

3.1 Introduction

Since the emerging of cloud computing, several authors have studied its suitability to run scientific applications. The motivation of these studies are the inherent benefits offered by cloud providers. First, cloud computing allows to scale the underlying infrastructure to fit the user needs, eliminating the effects of both under and over provisioning resources. Then, the pay-per-use model provides a cost-effective usage of resources, allowing the users to deploy the required infrastructure and pay for it only during the execution time. Finally, virtualisation provides increased flexibility, since Virtual Machines (VM) can be configured with all the dependencies required by the applications.

However, clouds are not widely used for all kinds of scientific applications because they also exhibit some drawbacks. First, cloud providers use a multi-tenant approach to optimise resource usage. This means that the physical processors, disk, memory, etc. where the VM is running can be shared with VMs from another user. This hardware sharing causes a variability on the CPU performance, memory bandwidth, network communications and disk I/O speed, a problem commonly known as *noisy neighbour* [39]. In addition, cloud providers typically offer instance types featuring certain characteristics, such as amount of RAM, number of virtual equivalent CPUs (vCPUs), storage, etc., but the user cannot select the specific hardware characteristics. These vCPUs are not physical cores, but a CPU equivalent unit. Unfortunately, the performance of these vCPUs are highly dependent on the underlying hardware, which produces high performance differences between instances of the same type. All these effects have been widely studied in the bibliography [43, 46, 42, 47] and even methodologies are provided to correctly measure this variability [168].

As a response to the demand of instances with predictable capabilities, some providers such as Amazon Web Services (AWS) offer the option to launch single-tenant instances [169] at the expense of additional costs. However, depending on the application this fee may not be worth. Also, these single-tenant instances ensure that the physical hardware will be used only by VMs from the account owner. However, this does not preclude from suffering noisy neighbour effects among the user's own instances.

Turning to parallel scientific applications, their execution times are usually determined by the slowest process, so an unbalanced situation will delay the entire

application. These facts highlight the need for advanced load balancing techniques to adapt scientific applications to the variable performance found on heterogeneous environments. This effort has been done for High Performance Computing (HPC) applications where authors have studied the suitability of cloud computing environments [62] [170] [171]. These studies agree that tightly coupled applications are less suitable for cloud computing, which is reasonable considering the fluctuations reported on network bandwidth. To mitigate the unbalance problem, several load balancing algorithms adapted to cloud environments have been proposed [172] [173]. In addition, we can find studies of techniques for efficient VM deployment [174] [175]. However, this unpredictable variability of the computational capabilities does not only affect tightly coupled processes, but also loosely coupled ones.

Loosely coupled applications neither require a continuous communication nor synchronisation points, like HPC applications. For instance, most of the load balancing algorithms designed for HPC involve an unnecessary overhead for these applications due to the amount of synchronisation points and communications involved. On the other hand, classic load balancing algorithms used on heterogeneous systems, which rely on previous knowledge of the underlying performance [176], are not suitable for these environments due to the unpredictable performance fluctuations.

To address these problems, we present RUPER-LB (Runtime Unpredictable Performance Load Balancer) a load balancing algorithm for loosely coupled applications running on environments with unpredictable performance variability with both multi-process and multi-thread balance. RUPER-LB is provided as open-source code[11] under the GPLv3 license. For assessment purposes, RUPER-LB was used to balance PenRed [177] simulations, which is a radiation transport simulation framework focused on medical applications with MPI and multithreading built-in parallelism.

3.2 Materials and Methods

RUPER-LB focuses on parallel iterative applications such as Monte-Carlo simulations, iterative solvers or multi-parametric analysis. These applications must comply with the following restrictions:

Firstly, the application must be split in tasks. During the execution of these tasks, the application should not require any communication or synchronisation point among the executing threads or processes. Nevertheless, if communications are required, their overhead on the task performance should be negligible. If these assumptions are not accomplished, RUPER-LB can still be used but an HPC-like

[11] RUPER-LB: `https://github.com/PenRed/RUPER-LB`

load balancing algorithm may achieve better results in terms of makespan.

Secondly, the application should measure its speed at runtime. Thus, RUPER-LB assumes that the application behaves like an iterative process, whose speed is measured in iterations per second. The number of iterations to process by each thread and process should be allowed to be changed at runtime. Notice that RUPER-LB neither requires an homogeneous computational cost for the iterations nor a previous balanced distribution among threads.

PenRed, the selected code to test the presented algorithm, satisfies these required assumptions. In this code, tasks correspond to each particle source defined by the user. Each generated primary particle and all its secondaries will be considered as a single history, which corresponds to one iteration. Finally the number of histories to simulate by each thread and process can be changed at runtime.

3.2.1 Multi-threading balance

Some multi-threading applications employ the involved threads in an unbalanced way. For example, assigning I/O operations or network communications to a specific thread. Also, the computational cost of the iterations that constitute the process could be heterogeneous, or some thread could use accelerated hardware like a GPGPU. Both situations will produce variable unbalances on thread speeds, measured in iterations per second. Also, it is not feasible in a Cloud to know which computational resources are being shared with other VMs running on the same physical hardware and, therefore, how their workload pattern will change during the execution. This fact could increase the unbalance produced by previous effects. Thus, we need to balance the workload between the threads of a single process dynamically. This section describes how this local load balancing is performed.

The workload distribution, i.e. the number of iterations assigned to each thread, is handled by two components implemented as classes in an object oriented programming (OOP) language. These are the *tasks* and the *workers*, which represent a single task and the threads executing the task respectively. Also, the execution could involve more than one task, each of them having its own workers. Figure III.28 top shows the basic balance schema for single process executions, where each thread is assigned to a single worker of the active task. The basic states of both components are listed in table III.3.

Basically, each worker reports periodically the number of completed iterations to the *task* object. This is done using the *report* method, whose code is shown in Figure III.29. In this code, and the following ones, the use of locks and the sanity checks on variable values have been omitted for simplicity. The *report* method takes as argument three values: The worker index which perform the report, the number of completed iterations and the measure timestamp. Regarding the execution, first, we use two auxiliary *worker*'s methods, *working* and *elapsed*. The first one returns

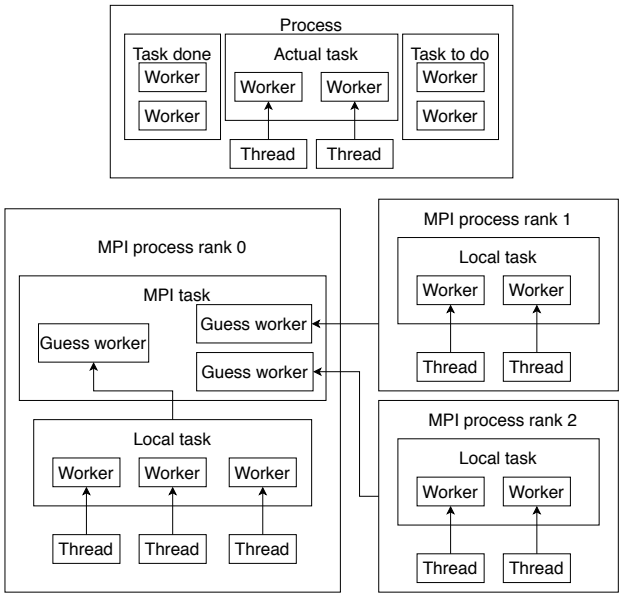

Figura III.28: Top: Thread balance system schema for a process with 3 tasks. Bottom: MPI balancing schema for 3 MPI processes and 1 task.

Taula III.3: Worker (left) and task (right) object states.

Variable	**Description**
I_n	Assigned iterations
$started$	Flags the task start
$finished$	Flags the task end
I_d	Number of finished iterations
t_r	Last report timestamp
t_i	Task start timestamp
m	Velocity measures vector

Variable	**Description**
I_n	Number of iterations to do
w	Vector of *worker* objects
t_0	Task start timestamp
t_{pc}	Last checkpoint timestamp
Δt_{pc}	Time between checkpoints
$started$	Flags task start
$finished$	Flags task finish
t_{min}	Balance time threshold
ds_{max}	Maximum speed deviation

$report(i, I_{done}, t)$
Input: $i \rightarrow$ Worker index $I_{done} \rightarrow$ Number of completed iterations $t \rightarrow$ Report timestamp **Output:** $\Delta t \rightarrow$ Suggested time until next report

if $w_i.working()$ **then**
$\Delta t \leftarrow w_i.elapsed(t)$
$dev \leftarrow w_i.addMeasure(t, I_{done})$
$dev \leftarrow ABS(dev - 1)$
if $dev > ds_{max}$ **then**
$\Delta t \leftarrow \Delta t \cdot max(1 - (dev - ds_{max}), 0.8)$
else if $dev < 0.1 \cdot ds_{max}$ **then**
$\Delta t \leftarrow \Delta t \cdot min(1 + (0.5 \cdot ds_{max} - dev), 1.2)$
end if
if $\Delta t > \Delta t_{pc}$ **then**
$\Delta t \leftarrow \Delta t_{pc} \cdot 0.8$
end if
else
$\Delta t \leftarrow -1$
end if

Figura III.29: *Task report* method.

true if the *worker* is still executing the task, otherwise returns *false*, and the second one returns the elapsed time since the last report. Following, the *worker* method *addMeasure* (Figure III.30) is used to compute and store its speed measured since the last report (t_r). In addition, that method returns the quotient s/s_l, where s is the new speed to register and s_l is the registered speed in the previous report, that is, the speed deviation from the previous report. This information will be used to calculate, in the *report* method, the suggested time interval until next report (Δt).

Each thread will compute its own reports independently, i.e. the threads do not require to synchronise to perform the report at the same time. The same goes for the *checkpoint* method, whose pseudocode is shown in Figure III.31. This *task* method, redistributes the workload among its *workers* according to the information stored by reports. First of all, the algorithm calculates three values: the total simulation speed (s_t), the total reported iterations done (I_t) and the predicted iterations done (I_{pred}). To obtain I_{pred}, we use the auxiliary *worker* method *predDone*, which returns the predicted iterations done by the worker assuming no

$addMeasure(t, I_{done})$
Input:
$I_{done} \rightarrow$ Number of completed iterations
$t \rightarrow$ Measure timestamp
Output:
$dev \rightarrow$ Speed deviation
$\Delta t \leftarrow t - t_r$
$\Delta t_m \leftarrow t - t_i$
$\Delta I \leftarrow I_{done} - I_d$
$s_l \leftarrow speed()$
$s \leftarrow \Delta I / \Delta t$
$I_d \leftarrow I_{done}$
$t_r \leftarrow t$
$dev \leftarrow s / s_l$
$m \leftarrow (\Delta t_m, s)$

Figura III.30: *Worker addMeasure* method.

changes on its speed since last report. Notice that the calculation of task speed excludes the already finished workers. Then, we check if the required iterations have been done. If that happens, the assigned iterations of each worker will be set to its reported iterations done, i.e. force workers to finish the task. On the other hand, if there are still iterations to do, we evaluate a prediction of the remaining execution time (t_{res}) according to I_{pred} and s_t. Finally, if t_{res} is greater than the threshold (t_{min}), the iterations assigned to each active worker will be recalculated according to its speed factor.

At some point of the execution, the workers will consider that they have finished the task. At this point, workers will ask to finish to the *task* object, which will allow or refuse the request to finish according to the *task* stored information. There are two reasons to deny this request. The first reason is that the *task* object has registered less iterations done by the worker than the ones assigned. In this case, a new report will be required. The second reason is that the estimated remaining execution time to complete the task is greater than t_{min}. This last case requires a new checkpoint to reassign the number of iterations for each worker. If neither of both conditions are accomplished, the worker can finish the task. Thus, the *worker* method *working* will return *false* hereinafter. Once all workers have finished, the task is considered as finished.

checkPoint()

Input:
Output:

$t_{pc} \leftarrow actualTime()$
$s_t \leftarrow 0$
$I_t \leftarrow 0$
$I_{pred} \leftarrow 0$
for each $worker$ **in** w **do**
  $I_t \leftarrow I_t + worker.I_d$
  if $worker.working()$ **then**
    $s_t \leftarrow s_t + worker.speed()$
    $I_{pred} \leftarrow I_{pred} + worker.predDone(t)$
  else
    $I_{pred} \leftarrow I_{pred} + worker.I_d$
  end if
end for
if $I_n <= I_t$ **then**
  for each $worker$ **in** w **do**
    if $worker.working()$ **then**
      $worker.I_n \leftarrow worker.I_d$
    end if
  end for
else
  $I_{res} \leftarrow I_n - I_{pred}$
  $t_{res} \leftarrow I_{res}/s_t$
  if $t_{res} > t_{min}$ **then**
    for each $worker$ **in** w **do**
      if $worker.working()$ **then**
        $s_{fact} \leftarrow worker.speed()/s_t$
        $worker.I_n \leftarrow worker.I_d +$
          $s_{fact} \cdot (I_n - I_t)$
      end if
    end for
  end if
end if

Figura III.31: Method *checkPoint* for *task* object.

3.2.2 MPI balance

If MPI load balancing is enabled, this is handled at two levels, as shown in Figure III.28 bottom. First, locally to each MPI process, where the threads are balanced using the method described in the previous section. Then, the number of iterations to do is split between MPI processes. The rank 0 will handle the assignment of iterations for each process *task*, thus the I_n value is not constant on MPI. For that purpose, both objects *worker* and *task* are extended as follows. First, since the local thread reports are performed asynchronous, the iterations done and speed registered at local tasks are, in general, outdated. To counteract that, the MPI balance procedure registers the predicted iterations done, and not the reported ones. This procedure requires a new type of worker, which has been created as a derived object of the *worker* described in section 3.2.1. That new worker object used for MPI balance has been named *guess worker*, which shares the same state as the base *worker* class (table III.3). However, notice that *guess workers* do not represent a single thread, as the workers of section 3.2.1. Instead, a *guess worker* registers the information of the whole task running on one of the MPI processes (Figure III.28). In addition, a *guess worker* object uses a different *addMeasure* method, whose pseudocode is shown in Figure III.32. This *addMeasure* method corrects the last measured speed using the deviation between the reported and the expected prediction of iterations done at the time t. Notice that this method based on speed correction could fail if 0 iterations per second is reported. To handle this situation, the *addMeasure* method of the base *worker* object (Figure III.30) will be called.

On the other hand, to adapt *task* objects to handle MPI balance, we add the variables listed in table III.4 to its state. As indicated in the following descriptions, the usage of the new variables depends on the MPI process rank. For example, as shown in Figure III.28, only the rank 0 uses the vector w^{MPI} to save the local task reports.

Taula III.4: MPI *task* state extension.

Variable	Description
w^{MPI}	Vector of *guess workers*, one for each MPI process.
$finished^{MPI}$	Flags MPI balancing finish
I_n^{MPI}	Iterations to do between all MPI processes
$finish_{req}^{MPI}$	Flags MPI finish request
$finish_{sent}^{MPI}$	Flags MPI finish request sent

With these modifications, the report and balance steps are handled by a single

$addMeasure(t, I_{done})$

Input:
$I_{done} \rightarrow$ Iterations completed prediction
$t \rightarrow$ Measure timestamp
Output:
$dev \rightarrow$ Speed deviation

if $speed() = 0$ **then**
  $dev \leftarrow worker :: addMeasure(t, I_n)$
else
  $\Delta t \leftarrow t - t_r$
  $\Delta t_m \leftarrow t - t_i$
  if $I_d > I_{done}$ **then**
    $\bar{s_1} \leftarrow I_d/(t_r - t_i)$
    $\bar{s_2} \leftarrow I_{done}/(t - t_i)$
    $dev \leftarrow \bar{s_2}/\bar{s_1}$
  else
    $\Delta I_e \leftarrow speed() \cdot \Delta t$
    $\Delta I_r \leftarrow I_{done} - I_d$
    $dev \leftarrow \Delta I_r/\Delta I_e$
  end if
  $s \leftarrow dev \cdot speed()$
  $t_r \leftarrow t$
  $m \leftarrow (\Delta t_m, s)$
end if

Figura III.32: Method *addMeasure* for *guess worker* object.

thread in each MPI process via the *monitor* method. This one has a different behaviour regarding its rank number, as shown in figures III.33 and III.34. Both are explained below.

For rank 0 (Figure III.33), Δt_i^{report} and Δt_i^{next} save, respectively, the elapsed time between reports and the time until the next report for the *guess worker* number i. Then, *receiveAny* waits until some request is received, regardless the origin rank, or until the elapsed time reaches the *timeout*. In both cases, the elapsed time will be stored at Δt. If a request is received, it is stored at *req*. After the *receiveAny* call, the time until the next report request for each MPI process will be updated according to Δt. Also, if $\Delta t >= \Delta t_i^{next}$, a report will be requested to the process with rank i. Already sent report requests are flagged with $\Delta t_i^{next} = 0$. Finally, the timeout is set to the minimum value in the Δt^{next} array.

Regarding the procedure to handle the requests, there exists three possible requests. The first one, with identifier 0, handles the workers start petitions. As response to this request, the rank 0 sends a preliminary iteration assignation that will be updated when the first report is received. This part of the code uses the auxiliary method $done^{MPI}()$, which returns the number of the predicted iterations done by all the MPI processes.

The second instruction, with identifier 1, handles the reception of the reports. For that purpose, the method *receiveReport* is used to handle the petition. The functionality of *receiveReport* is very similar to the already shown methods *report* and *checkpoint*, except that it works with predictions of the computed iterations via the *guess worker addMeasure* method. So, it stores the new measure, updates the iteration assignment for MPI workers, and sends to the rank i its new assignation together with a flag to indicate if the MPI balance continues or finishes. As local balance (section 3.2.1), this will finish when the predicted remaining time is below the threshold. When the MPI balance finishes, the number of assigned iterations for each MPI process will remain unaltered hereinafter. To save space, the pseudocode of this function is not included at this document. However, the details can be found at the provided source code repository. Finally, once the response has been sent, the corresponding time until the next report and the timeout are updated.

The last instruction, with identifier 2, handles the finish requests. Like the method used at section 3.2.1, MPI workers can request to finish the task, attaching a report to their request. The reasons to send a finish request will be explained at the *monitor* description for non zero ranks. For instance, these requests are handled by *receiveReport* too. Finally, we check if all workers have been notified that the MPI balance has finished. In this case, the monitor execution ends.

For the other ranks, which constitute the MPI workers, the monitor pseudocode is shown in Figure III.34. First of all, the monitor sends a start petition to the

monitor()

Input:
Output:

for $i = 0$ **until** $w^{MPI}.size() - 1$ **do**
  $\Delta t_i^{report} \leftarrow \Delta t_{pc}$
  $\Delta t_i^{next} \leftarrow 0$
end for
$timeout \leftarrow \Delta t_{pc}$
while *true* **do**
  $req \leftarrow receiveAny(timeout, \Delta t)$
  $timeout \leftarrow 10^9$
  for $i = 0$ **until** $w^{MPI}.size() - 1$ **do**
    if $\Delta t_i^{next} > 0$ **then**
      if $\Delta t_i^{next} <= \Delta t$ **then**
        $requireReport(i)$
        $\Delta t_i^{next} \leftarrow 0$
      else
        $\Delta t_i^{next} \leftarrow \Delta t_i^{next} - \Delta t$
        if $timeout > \Delta t_i^{next}$ **then**
          $timeout \leftarrow \Delta t_i^{next}$
        end if
      end if
    end if
  end for
  if req **then**
    if $req.instruction = 0$ **then**
      $I_{rem} = I_n^{MPI} - done^{MPI}()$
      $req.send(I_{rem}/w^{MPI}.size())$
      $\Delta t_{req.node}^{next} \leftarrow \Delta t_{req.node}^{report}$
    else if $req.instruction = 1$ **then**
      $\Delta t_{req.node}^{report} \leftarrow receiveReport(req)$
      $\Delta t_{req.node}^{next} \leftarrow \Delta t_{req.node}^{report}$
      if $timeout > \Delta t_{req.node}^{next}$ **then**
        $timeout \leftarrow \Delta t_{req.node}^{next}$
      end if
    else if $req.instruction = 2$ **then**
      $receiveReport(req)$
    end if
    if $allFinished()$ **then**
      exit
    end if
  end if
end while

Figura III.33: Method *monitor* of the object *task* for MPI rank 0.

monitor()

Input:
Output:

$I_n \leftarrow send(0)$
while *true do*
  $req \leftarrow waitAny(finish_{req}^{MPI})$
  if *req* **then**
    if $req.instruction = 1$ **or** 2 **then**
      $t \leftarrow actualTime()$
      $I_d^{pred} \leftarrow predDone(t)$
      $req.send(t, I_d^{pred})$
      $(I_n, finished^{MPI}) \leftarrow req.receive()$
      if $finished^{MPI}$ **then**
        exit
      end if
      if $req.instruction = 2$ **then**
        $finish_{sent}^{MPI} \leftarrow false$
      end if
    end if
  else
    $send(2)$
    $finish_{req}^{MPI} \leftarrow false$
    $finish_{sent}^{MPI} \leftarrow true$
  end if
end while

Figura III.34: Methods *monitor* of the object *task* for MPI rank greater than 0.

rank 0 and receives the initial assignation of iterations to do. Once inside the loop, the function *waitAny* waits to receive a petition or a response from the rank 0 or until the value of the variable $finish_{req}^{MPI}$ changes to *true*.

On the first case, whether the received instruction identifier is 1 or 2, the monitor sends the predicted computed iterations (I_d^{pred}) at time instant t. Then, it waits to receive the response of the rank 0 with the new iteration assignation and the flag to finish the MPI balance ($finished^{MPI}$). If the MPI balancing has finished, the monitor process ends. Finally, if this request is a response of a finish petition (instruction 2), the $finish_{sent}^{MPI}$ is set to $false$ to allow triggering new finish petitions.

Instead, if $finish_{req}^{MPI}$ has changed its value to *true*, the monitor sends an instruction petition 2 to ask to finish the MPI balance. Also, the values of the flags $finish_{req}^{MPI}$ and $finish_{sent}^{MPI}$ are changed to $false$ and *true*, respectively. The value of $finish_{req}^{MPI}$ can be changed to *true* by local threads when they try to finish the task. This happens when a worker satisfies the criteria to finish the local task shown in section 3.2.1. However, if the MPI balance is still active, the number of iterations to carry out could change. For instance, the local task cannot allow its workers to exit the task. Instead, the local task sends a finish petition to rank 0. In addition, the flag value could also change when a local *checkpoint* call reaches a remaining time lower than the threshold.

3.3 Results

To test the efficiency of the proposed algorithm, first, we have simulated the variable overhead caused by neighbour VMs on an on-premises cloud managed by OpenStack. Its underlying infrastructure is composed by nodes with two Skylake Gold 6130 at 2.1 GHz with 16 cores each and 768 GB RAM DDR4@2666. Then, RUPER-LB has been used to balance executions running on AWS to check its suitability on a public cloud provider. The selected infrastructure consists on 3 m5.large instances with 2 vCPUs and 8 GB RAM each one.

3.3.1 On-premises Cloud

The deployed infrastructure for our experimentation consists of two physical nodes, as shown in Figure III.35. On the first, a single VM was deployed with 64 vCPUs to ensure that the physical node is not shared with any other VM. The second one is filled with smaller VMs with 8 vCPUs each one. On the second node, only one of the small VMs will execute the PenRed simulations. Also, four of the other small VMs, will execute a dummy process whose CPU usage depends on the time of day. These overhead tasks are bash scripts which run the command *yes* followed by a *sleep*. The sleep time depends, as we said, on the time of day. With this approach,

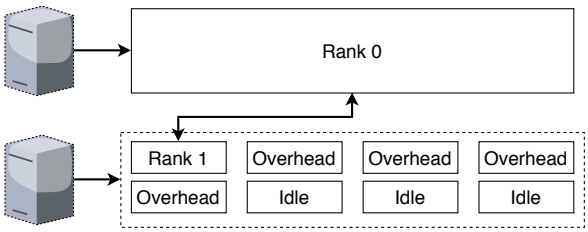

Figura III.35: Test infrastructure schema.

we simulate a variation of the CPU usage of the neighbours VMs. The other VMs remain idle, and their only purpose is to fill the physical node.

Regarding the application to balance, we have selected PenRed [177] code system, which implements the PENELOPE [163] physics in an extensible parallel engine for radiation transport in matter simulations. Some of its usages are performing simulations of clinical radiation treatments, radiological protection, or industrial applications. To test RUPER-LB we will use the PenRed simulation example *2-plane*, provided as part of the software distribution.

With that experimental setup, we have executed the very same simulation with and without load balancing. We have configured the minimum time between checkpoints (Δt_{pc}) to 300 s, which has been selected according to process execution time order. Thus, we expect to see executing times delay between ranks and threads lower than 300 s. In the following experiments, two MPI processes have been used. The process with rank 0 runs on the large VM, i.e. with no neighbour influence. Thus, the process with rank 1 is executed at the node with multiple tenants. In addition, both processes use 8 threads each.

The same simulation was repeated 4 times both with and without load balancing. Figure III.36 shows the execution time of every process by rank number, for each simulation run. As we can see, on the load balanced results, the delay between ranks is smaller than the selected Δt_{pc}. At the following test, we have increased the computational cost increasing the number of iterations (Figure III.37). As expected, maintaining the same value of Δt_{pc}, the relative differences on execution time are reduced.

For the same simulation, with load balancing enabled, Figure III.37 bottom shows the execution time for each thread of each MPI process. There, the dashed lines limit the fastest and the slowest thread for both ranks, and we can check that the corresponding delay is below Δt_{pc}.

To test how RUPER-LB can save execution time inside a single node, we have executed the same simulation using 4 MPI processes with 8 threads for each one, but all of them running on the single-tenant node. This simulation has been executed with and without load balancing. The corresponding execution times for

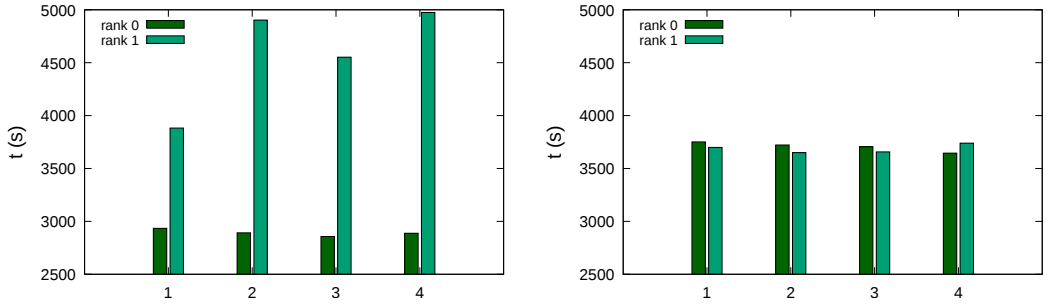

Figura III.36: Execution time using 2 MPI processes with 8 threads each one. Left: without load balance. Right: with load balance.

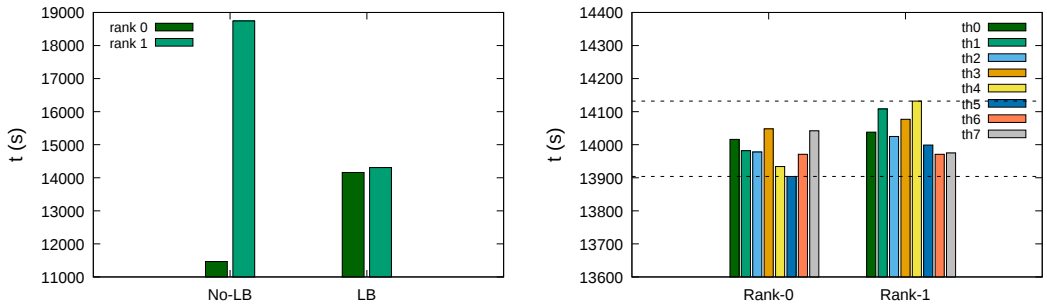

Figura III.37: Execution times for simulations with a higher number of iterations, by rank (left) and by thread with load balance (right).

each rank are shown in Figure III.38. The same simulation with load balancing enabled is about a $6-7\%$ faster. To understand the results shown in Figure III.38, we have represented the mean speed evolution of the threads of each MPI process in Figure III.39. As we can see, at the end of the execution the mean speeds present non negligible differences between the threads of the same MPI process. This fact explains why RUPER-LB achieves shorter execution times on this test. On the other hand, to explain why Figure III.38 seems to show no unbalance between ranks, notice that the execution time of each rank is determined by the slowest thread. Even if there exists unbalance between the threads, if the slowest thread of each rank requires approximately the same execution time in all of them, that gives the false appearance that the whole process is well balanced.

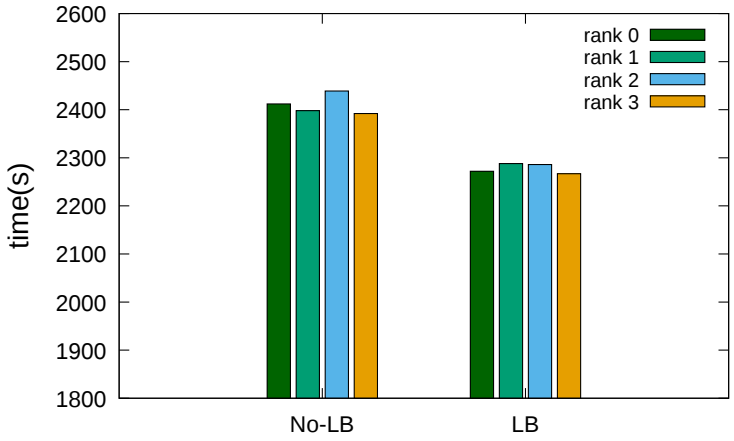

Figura III.38: Simulations executed with 4 MPI processes and 8 threads each one on the single-tenant node.

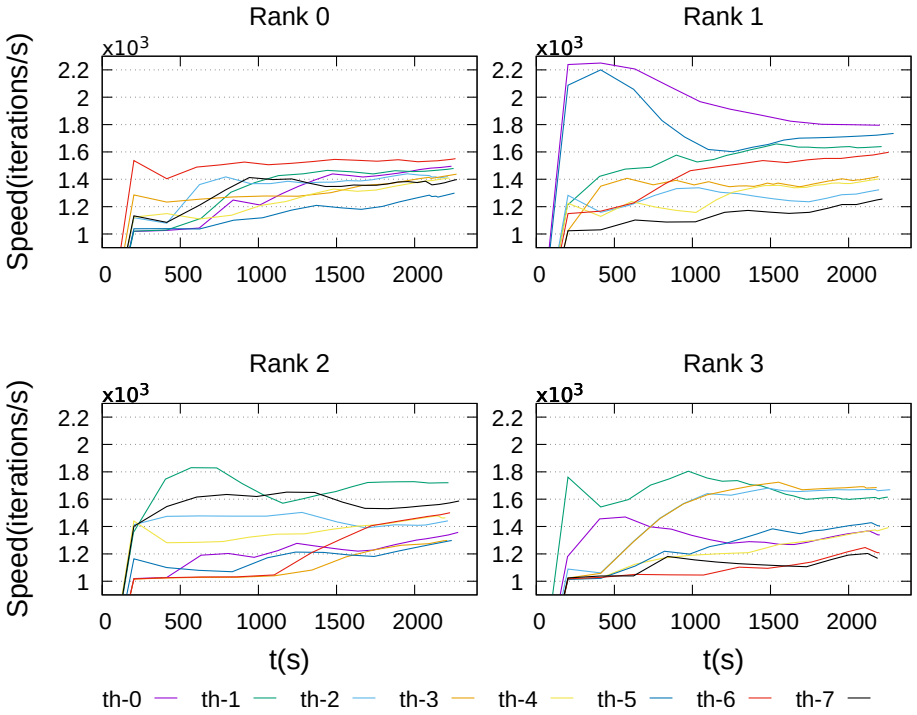

Figura III.39: Evolution of the mean speed for each thread in each MPI process.

3.3.2 Public Cloud provider

Once tested the functionality of RUPER-LB on a controlled environment (section 3.3.1), we executed several simulations on a MPI cluster deployed at AWS. This one consists on three m5.large instances. These executions show several differences between execution speeds, as we can see in Figure III.40. This Figure represents the speed of each node relative to the rank 0 speed, and shows differences up to 20% on some executions. Notice that the simulation speed does not depend on whether or not load balance is used, as we have seen in section 3.3.1.

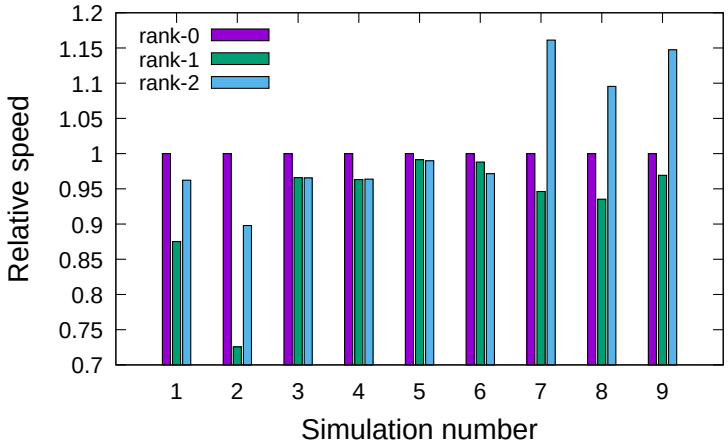

Figura III.40: Simulation speeds of each node relative to the rank 0 speed.

To compare the unbalance with and without load balance system, we have selected the first two simulations of the Figure III.40, which do not use load balance, and the 8 and 9 simulations, which use RUPER-LB. The Figure III.41 shows the relative simulation time of each node relative to the rank 0. Also, simulations 8 and 9 have been executed using a value of Δt_{pc} corresponding to the 16% and the 8% of the total execution time respectively. This selection affects the final accuracy of the balance process, as seen in Figure III.41.

Notice that, in our worst unbalanced case (simulation 2), the whole cluster remains active the 25% of the time with only a single node working and the others waiting. This waste of resources can be avoided using RUPER-LB, which provides an accurate balance of the workload with a low overhead.

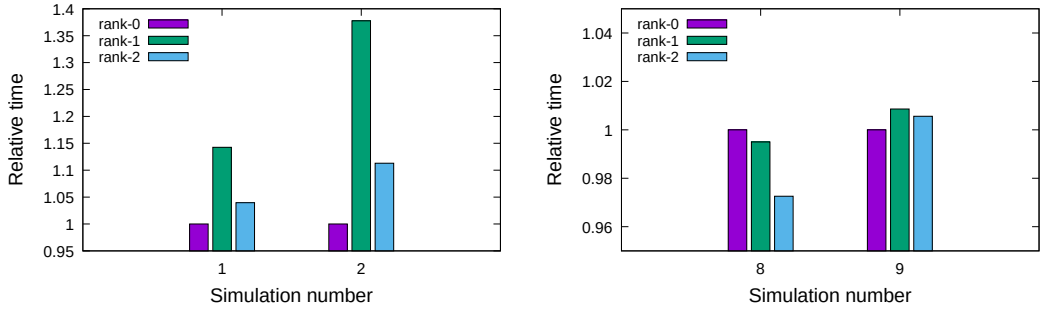

Figura III.41: Simulation time of each node relative to the rank 0 time without (left) and with (right) load balancing.

3.4 Conclusions

This work presents RUPER-LB, a load balancing system for applications with mixed MPI/multithreading parallelism support with loosely coupling. RUPER-LB focuses on iterative processes running on platforms with variable computational capabilities, such as cloud computing environments. We have shown the capabilities of RUPER-LB using a real world simulation software with MPI and multithreading capabilities. Due to its asynchronous approach, RUPER-LB introduces a negligible overhead on the processing time, making it suitable for applications with few communications. In addition, as RUPER-LB only require periodic reports of thread speeds, it is easily integrable on most applications.

Future work involves testing RUPER-LB running different kind of applications on both, public and on-premises cloud providers. Also, improving the finish request step to minimize threads waiting time. Finally, extending RUPER-LB to be able to balance executions on different cloud services, such as serverless functions.

Acknowledgment

This study was supported by the program “Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d” under grant number ACIF/2018/148 from the Conselleria d’Educació of the Generalitat Valenciana and the “Fondo Social Europeo” (FSE). The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad"for the project “BigCLOE” with reference number TIN2016-79951-R.

4 TaScaaS: A Multi-Tenant Serverless Task Scheduler and Load Balancer as a Service

Referència: V. Giménez-Alventosa, G. Moltó and J. D. Segrelles, "TaScaaS: A Multi-Tenant Serverless Task Scheduler and Load Balancer as a Service," in IEEE Access, vol. 9, pp. 125215-125228, 2021, doi: 10.1109/ACCESS.2021.3109972.

Categoria JCR: COMPUTER SCIENCE, INFORMATION SYSTEMS

Posició: 65/161

Quartil: Q2

Factor d'impacte (2020): 3.367

Arrel del treball anterior (secció 3), s'ha aconseguit un balanceig eficient en infraestructures compartides amb prestacions que sofreixen fluctuacions imprevisibles. No obstant, queda encara un problema per abordar, i és la gestió òptima i automàtica d'execucions en múltiples infraestructures independents.

Sovint, els còmputs científics requereixen un gran esforç computacional, el que força als investigadors a emprar múltiples infraestructures que, generalment, pertanyen a organitzacions diferents. Un parell d'exemples els trobem al treball de Christian et al. [178] on els autors descriuen que han requerit més de 30000 hores de CPU per a simular cada un dels casos estudiats consistents en més de $7 \cdot 10^{11}$ partícules primaries i el de Vicent et al. [14] on cadascuna de les simulacions realitzades per a cada parell de cambra d'ionització i feix de fotons considerat ha requerit unes 13800 de CPU, fent un total de 745200 hores de CPU aproximadament.

Per afrontar el problema descrit, el quart treball d'aquest document presenta TaScaaS[12], que mitjançant el balanceig de càrrega proporcionat per l'eina RUPER-LB és capaç de dividir, distribuir i balancejar la càrrega de treballs executats al maquinari de diferents infraestructures independents entre si i amb prestacions que fluctuen de forma impredictible. Més encara, es capaç de partir el processament de forma automàtica per tal d'ajustar el temps d'execució global segons les necessitats de l'usuari. Cal remarcar que les aplicacions que poden ser gestionades per TaScaaS han d'acomplir certes condicions, tal i com ocorre amb el sistema de balanceig RUPER-LB. Dites condicions venen detallades al treball presentat i permeten l'ús d'aplicacions tals com les simulacions de Monte Carlo descrites a la secció 3.

[12]TaScaaS: `https://github.com/grycap/TaScaaS`

Com es descriu al treball, al millor dels casos, és a dir, quan les fluctuacions de les prestacions són menyspreables, una aproximació de balanceig estàtic reprodueix el resultat aconseguit per TaScaaS, sent este més eficient a mesura que la magnitud de les fluctuacions augmenta. A més, TaScaaS, evita tant l'aprovisionament excessiu de recursos per tal d'acomplir amb un temps d'execució predeterminat, com els retards en l'execució degut a les fluctuacions. Aquests últims depenen de les infrastructures emprades, no obstant, als tests realitzats poden retardar l'execució d'un treball més d'un 40%.

Abstract

A combination of distributed multi-tenant infrastructures, such as public Clouds and on-premises installations belonging to different organisations, are frequently used for scientific research because of the high computational requirements involved. Although resource sharing maximises their usage, it typically causes undesirable effects such as the *noisy neighbour*, producing unpredictable variations of the infrastructure computing capabilities. These fluctuations affect execution efficiency, even of loosely coupled applications, such as many Monte Carlo based simulation programs. This highlights the need of a service capable to handle workload distribution across multiple infrastructures to mitigate these unpredictable performance fluctuations. With this aim, this work introduces TaScaaS, a highly scalable and completely serverless service deployed on AWS to distribute loosely coupled jobs among several computing infrastructures, and load balance them using a completely asynchronous approach to cope with the performance fluctuations with minimum impact in the execution time. We demonstrate how TaScaaS is not only capable of handling these fluctuations efficiently, achieving reduction in execution times up to 45% in our experiments, but also split the jobs to be computed to meet the user-defined execution time.

4.1 Introduction

The use of huge computational power is commonly required in science and engineering to be able to perform computational experiments. Many of these experiments are carried out by loosely coupled algorithms which can be easily parallelized to be executed in a distributed environment. However, the high computational power requirements typically forces the researchers to use several infrastructures belonging to different organisations. For instance, in Monte Carlo simulations of radiation transport applied to the calculus of ionisation chamber correction factors, the work presented by Christian et al. [178] required more than 30000 CPU hours to simulate a single case consisting on more than $7 \cdot 10^{11}$ primary particles, and Vicent

et al. [14] reported approximately 13800 CPU hours to simulate each combination of ionisation chamber and photon beam considered in the study, which results in a total of 745200 CPU hours. As consequence, both works have used several independent infrastructures to cope with the huge computational workload of the studies. These cases, and many others, highlights the need to efficiently handle the execution of loosely coupled applications across several computing infrastructures.

However, distributed infrastructures usually involve heterogeneous computing environments. Therefore, a single infrastructure could exhibit disparate performance among its available computing nodes due to differences in the underlying hardware. Moreover, it is common for computing infrastructures to use a multitenancy approach i.e. multiple users share the same underlying physical infrastructure in order to optimise resource usage. This technique is used both in local infrastructures and in cloud computing environments. As a consequence, physical resources such as processors, memory, disk or network bandwidth, could be shared between different users or the same user itself, either through virtualization or using a queuing system. Sharing hardware resources causes a non-negligible effect on the whole performance, commonly known as *noisy neighbour* [39]. Notice that the noisy neighbour effect unpredictably affects performance, since it depends on the tenant's workload and the resources being used.

Both effects, *hardware heterogeneity* and *noisy neighbour*, have been widely studied in the literature. For example, Alexandru et al. [43] performed a long term study of the performance variability on ten production cloud services. In the same line, Philipp and Jürgen [46] studied the impact of this variability on four cloud environments, showing a different impact in each cloud. Furthermore, in a study performed on Amazon Web Service (AWS) [44], Jörg et al. [42] concluded that the performance variability not only differs among cloud providers, but also among Availability Zones (AZs). Recent studies still confirm the existence of this performance variability in IaaS providers [47] and services such as AWS Lambda, as described in our previous work [179]. Indeed, it has even been studied how to reproduce experiments under these changing conditions [168].

To face these problems we present TaScaaS (Task Scheduler as a Service), a completely serverless and highly scalable job scheduler and load balancer service for long-running loosely coupled applications. TaScaaS can be deployed on any FaaS (Functions as a Service) solution, but we have relied on AWS Lambda [13] to exemplify its deployment in a particular cloud provider. Thus, it does not require any previously provisioned infrastructure, it can scale automatically and rapidly according to the workload and it can run at a zero cost if the usage level does not exceed AWS's free tier. In addition, it mitigates the impact on the application performance introduced by the variability in shared and heterogeneous environments.

[13] AWS Lambda - `https://aws.amazon.com/lambda`

This is done via an asynchronous load balancer system named RUPER-LB [180], which automatically splits the workload to satisfy execution time constraints specified by the user. Finally, TaScaaS has been designed to be easily deployed with the Serverless framework [181] without administrator privileges and it is provided as an open source project available in GitHub[14].

After the introduction, the remaining sections are organised as follows. First, section 4.2 discusses the related work in the area. Then, section 4.3 introduces TaScaaS and provides details about the architecture and components. Later, section 4.4 tests and discusses the capabilities of TaScaaS, which are compared to a static split approach on section 4.5. Finally, section 4.6 summarises the conclusions and introduces the future work.

The contributions of this work are, first, introducing a service to automatically distribute the workload of iterative long-running applications among several nodes of different infrastructures, each of which may belong to a different organization. Secondly, the work provides a distributed and asynchronous load balance system whose impact on the execution time is negligible. Third, TaScaaS assists the infrastructures in the scaling process to achieve an efficient usage of the available resources, avoiding both, over- and under-provisioning. Finally, an automatic system to partitioning the incoming jobs is provided to achieve execution time constrains set by the user.

4.2 Related work

Since this work considers loosely coupled applications, previous works based on task scheduler systems can be used in this regard. In this case, the entire execution could be split in tasks wrapping a portion of the computations.

Focusing on serverless schedulers, there are systems to distribute the workload according to the application characteristics and/or the underlying hardware performance. For example, focusing on applications running on serverless environments, we can find Wukong [24], which provides a serverless parallel computing framework to handle executions on AWS Lambda using a decentralised scheduler. However, this approach does not allow to combine executions from other infrastructures. On the same topic, the SDBCS algorithm, proposed by Pawlik et al. [33], plans serverless executions according to budget and time constraints, but it is restricted to serverless executions. These approaches, assume that the application can be represented as a direct acyclic graph (DAG) composed of several short fine-grained tasks. Thus, these tasks are well suited to run directly on a serverless environment. However, computing intensive tasks are less suitable to run on serverless environments, due to the execution time and storage limitations [34, 182].

[14]TaScaaS: `https://github.com/grycap/TaScaaS`

In addition, depending on the prices, the cost could be no longer competitive [95].

Another scheduler for DAG applications, but not executed on serverless environments, is described in the work by Weiling et al. [77], where the authors consider the performance fluctuations of the Virtual Machine (VM) carrying out the computation to predict its future performance and schedule the tasks accordingly. However, their approach is used on tasks with short executions of less than a minute. As the expected performance fluctuations are slower than the mean execution time of a single task, the predictions done by their scheduler are stable enough during the task execution. Nevertheless, on long tasks, this prediction will be less accurate. Although the execution could be divided in smaller tasks, this approach will require a huge amount of unnecessarily communications. Furthermore, the application could require a initialisation step whose execution time is, usually, negligible considering the whole execution. However, splitting the computation too much could force the initialisation to be repeated on each task, and produce a non-negligible overhead in the whole execution.

The same applies to other workflow DAG scheduling systems such as the ones proposed in [164, 183, 184]. These schedulers attempt to predict the performance of the computing nodes. Due to the short life of each task, these kind of algorithms could be adapted to fluctuating environments regularly measuring the performance of the system, and assuming that the performance will not fluctuate significantly during the task execution. However, for applications which are not composed of short fine-grained tasks, but by long-running independent ones, the performance can strongly fluctuate during the computation, thus rendering the performance prediction wrong. Thus, for the considered applications, a static assignment is not well suited for fluctuating environments.

HTCondor [53] is a useful tool to maximise the usage of the available resources in a distributed infrastructure, even if the user does not own the resources. However, it does not provide a system to minimise the effect of performance fluctuations on long executions or to satisfy execution time constraints.

To summarise, first, in previous works the strategy followed to distribute the jobs consists on measuring and predicting the infrastructure performance and assume that it remains stable during the mean execution time of a single task. Therefore, the tasks are assumed to be short enough. Thus, those approaches are not suitable for long-running tasks. To solve this limitation, TaScaaS will use a load balance system during the job execution, allowing to reassign the workload among the available resources during the job execution.

Secondly, most works are unable to handle executions on multiple infrastructures. Moreover, some works can handle only executions on a specific service in a specific provider, such as the work of Carver et al. [24], which is focused in executions in the AWS Lambda environment. Instead, TaScaaS is agnostic to

the infrastructures where the executions take place, allowing to combine resources from different providers and organizations.

Finally, the studied previous works use a predefined partitioning of the job to be executed. For example, in the DAG based schedulers, each task is identified by a graph node. This procedure makes the user responsible for correctly partitioning the work. Thus, it is necessary a previous knowledge about the execution cost of each task. However, the behaviour of many applications strongly depends on the input parameters, making it difficult to predict the execution time of each possible task for each resource type. Alternatively, TaScaaS handles the job partitioning automatically according to the execution time constraint specified by the user. Moreover, the partitioning is updated during the execution to be adapted to the fluctuating performance.

4.3 Architecture

As discussed in the introduction, TaScaaS provides a complete serverless service to schedule and distribute iterative jobs among the different infrastructures. To describe the TaScaaS architecture and functioning we define the following set of terms and assumptions:

First, we will define the processes to be executed in the available infrastructures, hereinafter named *jobs*. The characteristics of these *jobs* are described as follows and summarised in Table III.5. The *jobs* are supposed to be iterative processes where each iteration is independent of each other. Therefore, the job with N_i iterations can be split in N_p *job partitions* where each one processes, independently, a subset of the total number of iterations. The easiest way to perform the split consists on dividing equally the number of iterations among all *partitions*, i.e. each one processes N_i/N_p iterations. In addition, we suppose that the number of iterations assigned to each partition can be changed at runtime. Also, as the iterations are independent, the *job partitions* require no communication among them to compute their partial results. Although the assumptions seems to be restrictive, a wide variety of applications satisfy these conditions, for example, many Monte Carlo simulations algorithms. Furthermore, considering only the field of radiation transport simulations, based on Monte Carlo techniques, there are a wide variety of simulation programs, such as PENELOPE [185], GEANT4 [186], FLUKA [187], EGS [188], MCNP [189] among others. In addition, several programs have been developed based on the previous ones, such as PenEasy [190], PenRed [191] or GATE [192]. This codes are widely used for several applications, and are considered as the gold standard methods to perform calculations for clinical radiation based treatments, according to the Task Group report number 186 [165] of the American Association of Physicists in Medicine (AAPM), among other international protocols. Moreover, these characteristics are met not only by Monte Carlo simulations,

Taula III.5: Summary of the required job characteristics to be handled by TaScaaS.

Job characteristics
Iterative process
Independent iterations
Divisible in partitions
No communication required for partial results
Measurable processing speed at runtime

but also for other applications, such as multiparametric explorations, like neural network design exploration, or file processing, like image recognition processes. With these assumptions, the execution speed can be measured in iterations per second, where the meaning of *iteration* depends on each application. Following the same examples, an iteration could be, each simulated primary particle, for the radiation transport simulation case, each set of parameters that define a neural network or a set of images to process.

Second, each *job partition* will be processed by *workers*, which can be physical nodes, CPUs, vCPUs, a computing cluster, etc., since our framework is agnostic to the underlying computing infrastructure being used for processing. Notice that the number of iterations assigned to each partition will not be a fixed amount. Instead, the number of iterations assigned to each partition will be recalculated according to the processing speed of each one, assigning more iterations to the partitions which are processed faster. Therefore, TaScaaS maximises the usage of faster resources to achieve reduced *job* execution times.

Third, each *worker* belongs to a single *worker infrastructure*, which are composed of one or more *workers*, such as a computing cluster, a set of deployed nodes in a public or on-premises cloud, a single computer, etc. TaScaaS will consider each worker as a slot to compute a single *job partition*, i.e. a single *worker infrastructure* can process concurrently as many *job partitions* as the number of its workers. In addition, each *worker infrastructure* has a *frontend* process which performs periodic communications with the TaScaaS service to control the infrastructure live cycle and request more *jobs partitions* to process. TaScaaS is intended to distribute *job partitions* among several *worker infrastructures*, as shown in Figure III.42, and to balance the individual *jobs* workload among all their *partitions* according to their speeds. To achieve this purpose, each *worker* performs communications with the load balancer system during the *job partition* processing.

Once clarified the previous concepts, the TaScaaS architecture explanation follows. The provided TaScaaS service package is deployed on AWS, however, as the used services are commonly found in other public cloud providers, a equivalent approach can be followed on other providers.

TaScaaS does not require any pre-provisioned active computing infrastructure.

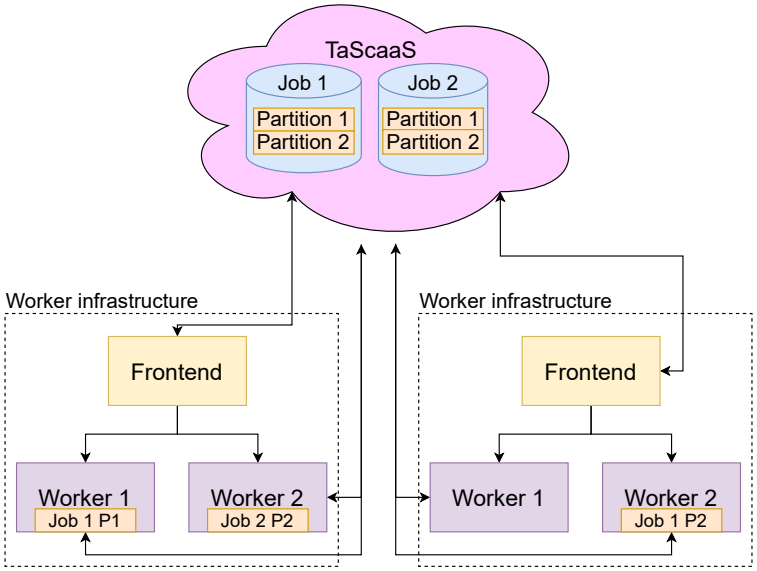

Figura III.42: TaScaaS functioning diagram.

Instead, it is executed in an event-driven approach, avoiding executing costs when is unused. In addition, the scale capabilities of all the services used by TaScaaS can be handled automatically by the cloud provider.

Notice that TaScaaS does not handle the infrastructure deployment, since there exist several tools for this purpose, such as Infrastructure Manager (IM) [89], Terraform [90] or services offered by cloud providers to automate the deployment, such as AWS Batch, Amazon EMR, etc. Therefore, resource provisioning is out of the scope of this work, as TaScaaS uses computing infrastructures already deployed by the user or where the user has access to perform computations. The motivation for this choice is to take advantage of combining resources of different organizations in which scientists have access. Nevertheles, as we will see, TaScaaS sends information to the *worker infrastructures* to assist them on the scaling process, requesting more or less slots according to the total workload.

The components and functionality of TaScaaS are described in Figure III.43, which can be split in two main parts: *jobs* and *worker infrastructures* life cycles. The implementation of TaScaaS involves four services: A high-performance object store system, which corresponds to Amazon S3 [193] in our implementation, a managed NoSQL database service based on tables, corresponding to Amazon DynamoDB [194], a Function as a Service (FaaS) serverless service, provided by AWS Lambda [195], and, finally, a REST API service to handle the communications with the *workers* and *worker infrastructures*, corresponding to Amazon API Gateway [196].

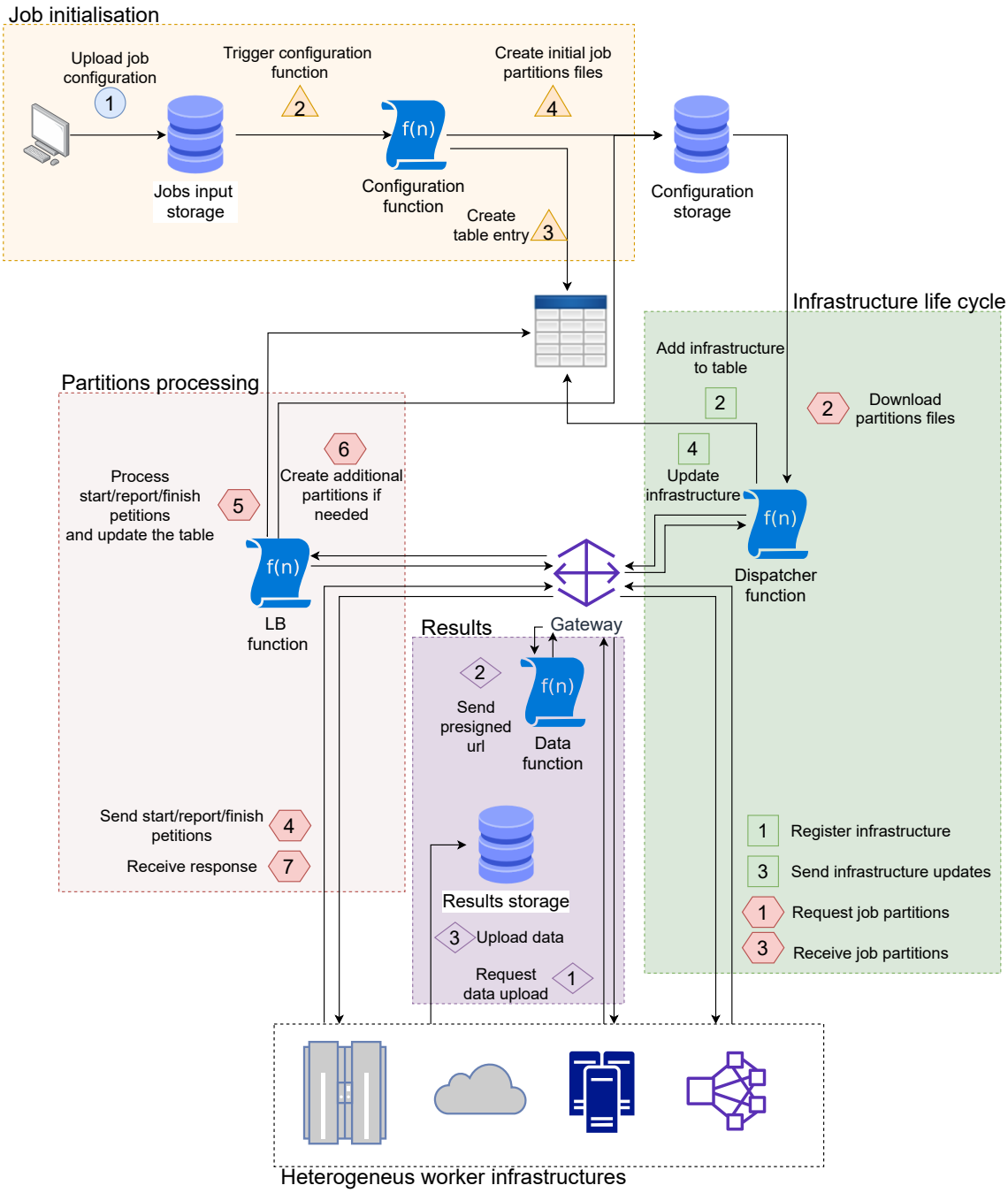

Figura III.43: TaScaaS component interaction diagram. The blue (circle) number represents the only step done by the user, the job upload. The orange numbers (triangles) correspond to the steps done to prepare the new incoming jobs from the user. On the infrastructures side, first, the green numbers (squares) describe the steps to register each infrastructure in the TaScaaS service and mantain the infrastructure information up to date. Then, the red numbers (hexagons) represent the procedure to request and process job partitions. Finally, the purple numbers (diamonds) describe how data is stored using the TaScaaS service. Also, the diagram has been split in zones to facilitate the description.

Taula III.6: Required services to implement TaScaaS in different Cloud providers.

Provider	**Storage**	**NoSQL DB**	**FaaS**	**REST API**
AWS	S3	DynamoDB	Lambda	API Gateway
Azure	Blob	CosmosDB	Functions	API Management
GCP	Storage	Datastore	Functions	API Gateway

The motivation to use a REST API is to allow both *workers* and *worker infrastructures* to communicate with TaScaaS with standard HTTPS requests through a REST API, avoiding the requirement of using a specific Software Development Kit (SDK).

Notice that the choice of AWS as the cloud platform to implement TaScaaS is a mere implementation detail, since other cloud providers such as Microsoft Azure [65] or Google Cloud Platform (GCP) [98] provide similar services. Indeed, an equivalent set of services for Microsoft Azure and GCP providers is shown in Table III.6.

The S3 service is used to store the input job files and the results of the executions in a bucket configured by the user. These buckets are the S3 storage units where the objects are stored. DynamoDB stores all the required information about running *jobs* and the available *worker infrastructures*. To handle the system workflow, four Lambda functions have been created, the *Configuration*, *Dispatcher*, *LB* and the *Data* functions. The functionality of each one will be discussed in the following sections. Finally, as mentioned, the API Gateway handles the communication between TaScaaS, the *worker infrastructures* and their *workers*, redirecting the requests to the appropriate Lambda function.

4.3.1 Jobs life cycle

To start the *jobs* life cycle, first, the job creation and initialisation must be discussed. To simplify visualisation of the description, the involved steps are summarised in the Figure III.43, which has been split in zones to facilitate the description. To create a new job, the steps involved are located in the *Job initialisation* or yellow zone. First, a job configuration file is uploaded to the jobs storage (blue or circle number 1). These configuration files specify the required *job* parameters, which include the number of iterations to perform, in how many *job partitions* the *job* must be initially split in, the required input data, and a time constrain to finish the execution.

Once uploaded, the *configuration* serverless function will be triggered (orange or triangle number 2). This will create an UUID for the *job* and assign it to all its *partitions* to be able to identify the job they belong to. Also, a configuration file for each initial *partition* (orange or triangle number 4) and the corresponding

set of entries in the TaScaaS database table (orange or triangle number 3) will be created to track all the running *jobs* information. A single entry is created for each *job partition* and another one to store the *job* configuration information. After that, the initial *job partitions* are created and ready to be requested and processed by the *worker infrastructures*.

When the initial *job partitions* have been created, they will be queued in the *configuration storage*. This is a folder in the TaScaaS file storage used to store the *partitions* configuration files until the *frontend* of a *worker infrastructure* requests them to be processed. Once requested, the partition configuration files will be dispatched following a first in first out (FIFO) approach.

First, to request *job partitions*, a *worker infrastructure* must be registered in the TaScaaS service, which steps are summarised by the 1 and 2 green *squares* in the green or *Infrastructure life cycle* zone of the Figure III.43. First, the *worker infrastructure* sends a register request (green or square number 1) to the REST API. This request must include the number of available slots, which is the actual number of *job partitions* that can process concurrently, and the maximum number of achievable slots, i.e, the maximum number of slots that the infrastructure can allocate increasing the number of nodes, CPUs, etc, with a scaling process. Then, the request is handled by the *Dispatcher function*, which will store that information in the table (green or square number 2).

Then, the *worker infrastructure* is allowed to request *job partitions* to process, which steps are summarised by the red or hexagon steps in the green or *Infrastructure life cycle* zone of the diagram. As before, the procedure begins with a request to the REST API, (red or hexagon number 1). The number of requested *partitions* is set according to the number of free slots. Then, the request is redirected by the Gateway to the *Dispatcher function*, which will process the configuration files stored in the *Configuration storage* (red or hexagon number 2) and send them to the *worker infrastructure* as response (red or hexagon number 3). The input data to perform the execution is provided via a presigned URL, which grants a limited time permission to download the file from the file storage with a simple HTTPS request. These presigned URLs expire after the specified time, and avoids the requirement to provide authentication credentials to the *workers*.

At this point, the *worker infrastructure* has received a set of *job partitions* to fill some, or all, of its available computing slots. Then, the job execution life cycle starts, which is represented described in the red or *Partitions processing* zone. When the processing starts, the *worker* must inform the TaScaaS load balancer system (red or hexagon number 4, start). This request is handled by the *LB* function and is used to inform the load balancer system that this *partition* is being processed. By default, TaScaaS uses RUPER-LB [180] as load balancer, implemented in the *LB* function, because it is precisely designed to be used on unpredictable

fluctuating environments. Nevertheless, it can be changed following the procedure explained in the GitHub repository. All the information regarding the load balancer system for the *job partition*, is stored in the corresponding TaScaaS table entry (red or hexagon number 5, start).

During the *job partition* execution, the load balancer system expects periodic reports to balance the workload of each *job partition* belonging the same job. These is done according to their relative processing speeds. These reports are handled by the *LB* function (red or hexagon number 4, report) and the period between requests depends on the specified time constrain and the TaScaaS configuration. During the report, the *job partition* information is updated in the table (red or hexagon number 5, report), and the number of assigned iterations is recalculated and sent back as response (red or hexagon number 7). Also, an estimation of the remaining time to complete the whole *job* is obtained. If the estimated execution time is greater than the specified in the *job* configuration, the *LB* function will create a set of new *partitions* to meet the time constraint (red or hexagon number 6). The number of new *job partitions* are limited by the TaScaaS configuration parameters to avoid overloading the *worker infrastructures*. The creation of new *job partitions* consists of adding the corresponding entries in the table, and create the configuration files in the *Configuration storage*. After that, the new *job partitions* could be requested to be processed by the *worker infrastructures* with the same procedure as in the *Infrastructure life cycle* zone.

To support worker failures, although the behaviour depends on the used load balancer system, RUPER-LB will check the latest communication timestamp of each *job partition* to consider the corresponding process as inactive or active. Inactive *job partitions* are not considered during the iterations distribution, and the corresponding ones will be distributed among the remaining active *job partitions*.

Finally, *workers* should send a finish request to TaScaaS to inform that this partition will not process more iterations. This request will be also handled by the *LB* function (red or hexagon number 4, finish). Although this request is intended to be sent when a *partition* has finished its assigned iterations, it could be used when the *worker* must stop the execution due to external reasons and there are remaining iterations to compute. For example, if the worker is running on a spot instance [197] it can be interrupted at any moment, depending on the used configuration. Another case is a worker running on a Lambda function, which should send a finish request when the execution time is approaching the configured timeout. When the load balancer receives this request, this *job partition* is flagged as finished in the table (red or hexagon number 5, finish), and will not be further considered to distribute the remaining iterations. Notice that if a *job partition* finishes before the assigned iterations have been reached, its remaining iterations are redistributed among the active *job partitions*.

Concerning the results handling, during, or at the end, the *job partition* processing, the *worker* could require to store results data. The procedure is represented in the purple or *Results* zone and is described following. First, the *worker* send an upload request to the REST API, which will be handled by the *Data* function (purple or diamond number 1). As response to this request, the *Data* function will send back a presigned URL to upload the results in the file storage (purple or diamond number 2). Like the procedure to obtain the input data, this approach allows to store data in the file storage without granting permissions to the *worker infrastructure* (purple or diamond number 3).

Even though TaScaaS has been designed to be used with a decentralised serverless load balancer system, it can be used as a simple job dispatcher and results storage for applications which do not support load balancing. Specifying a negative expected execution time in the *job* configuration file will disable the load balancer for this *job*. Nevertheless, TaScaaS will keep helping the *worker infrastructures* to fit the incoming workload, as is described in the next section. Further details and examples can be found in the TaScaaS repository[15].

4.3.2 Worker infrastructure life cycle

The *worker infrastructures* life cycle, as explained in section 4.3.1, begins with the registration in the TaScaaS service (green or *Infrastructure life cycle* zone of the Figure III.43). After that, each *worker infrastructure* must perform regular *update requests* (green or square number 3). The usefulness of this is twofold. First, the *update* informs TaScaaS that the *worker infrastructure* is still alive. If a *worker infrastructure* does not send any *update request* during the time specified in the TaScaaS configuration, it will be marked as inactive. Thus, it will not be considered to calculate the whole system computing capacity. Furthermore, if the *worker infrastructure* does not send an *update request* after a configurable amount of time, it will be removed from the table, requiring repeating the registration procedure to be able to receive *job partitions*. Secondly, the *update request* can be used to update the *worker infrastructure* information stored by TaScaaS (green or square number 4). This information includes the current number of slots and the maximum achievable slots. The schema of this procedure is included in the green or *Infrastructure life cycle* zone.

As mentioned, TaScaaS will assists the *worker infrastructures* to scale their slots according to the system workload. This is done in two steps. In the first one, during the time specified at the TaScaaS deployment configuration, the information about the number of dispatched and queued *job partitions* is stored. Then, combining this information with the total number of available slots of all

[15] `https://github.com/grycap/TaScaaS/tree/main/examples`

the registered *worker infrastructures*, and their maximum achievable slots, TaScaaS calculates the required percentage of achievable slots to tackle the incoming workload. This information is sent to the *worker infrastructures* frontends within the *update* and *job requests* responses. After the data measure step, TaScaaS waits for the same amount of time to let the *worker infrastructures* scale according to the received information. Notice that the *worker infrastructures* should send the new number of available slots using an *update request*, as explained before. Finally, to remove a *worker infrastructure* from the TaScaaS register, their frontend must perform a disconnect request.

4.3.3 Partitioning procedure

In this section we will discuss how jobs are partitioned in TaScaaS and which metrics are used for that purpose. First, the iterations processed during a job or partition execution can be approximated to the mean speed in the interval $[t_0, t_1]$ as,

$$n^I_{done} = \int_{t_0}^{t_1} s(t)dt \approx \Delta t \cdot \overline{s}(t_0, t_1) \tag{III.20}$$

Since the job execution is split in several partitions, the mean speed depends on the individual speeds of each partition. However, as TaScaaS uses a load balance system, we can assume that the global speed is the result of summing up the processing speeds of all individual job partitions (N^p),

$$\overline{s}^t(t_0, t_1) = \sum_{k=1}^{N^p} \overline{s}_k(t_0, t_1) \tag{III.21}$$

In addition, we can divide the speed measures in smaller time intervals to be able to obtain the processing performance behaviour with more precision in each instant,

$$\overline{s}^t(0, t) = \sum_{l=1}^{n^t} \overline{s}^t(t_l, t_{l+1}) \quad \setminus \quad t_1 = 0, t_{n^t+1} = t \tag{III.22}$$

With the previous considerations, the condition to be satisfied to achieve a job processing time restriction (t_{max}) is given by the equation III.23,

$$t_{max} \geq \frac{n^I_0}{\overline{s}^t(0, t_{max})} \tag{III.23}$$

where n^I_0 corresponds to the total number of initial iterations to be processed. However, if we want to achieve the execution time constrain at any instant t

during the execution, the equation III.23 can be rewritten as a function of the remaining time and the remaining iterations to process,

$$t_{max} - t \geq \frac{n^I(t)}{\overline{s}^t(t, t_{max})} \tag{III.24}$$

where $\overline{s}^t(t, t_{max})$ corresponds to the remaining time interval mean speed, which can differ significantly compared with the previous measured mean speeds due the fluctuating performance behaviour. The remaining iterations to process can be obtained from the number of processed iterations (equation III.20) as,

$$n^I(t) = n_0^I - t \cdot \overline{s}^t(0, t) \tag{III.25}$$

Replacing in the equation III.24 we obtain the condition to be satisfied at any instant t,

$$t_{max} - t \geq \frac{n_0^I - t \cdot \overline{s}^t(0, t)}{\overline{s}^t(t, t_{max})} \tag{III.26}$$

As we are assuming an unpredictable behaviour of the performance, we can't predict the value of $\overline{s}^t(t, t_{max})$. Instead, if at the t instant the number of speed measures is n^t, TaScaaS will approximate $\overline{s}^t(t, t_{max})$ to the latest available measure (n^t), i.e.,

$$\overline{s}^t(t, t_{max}) \approx \overline{s}^t(t_{n^t}, t_{n^t+1}) \tag{III.27}$$

as $\overline{s}^t(t_{n^t}, t_{n^t+1})$ provides the most accurate measure of the actual performance of the system. Then, if the condition of the equation III.28 is not satisfied,

$$t_{max} - t \geq \frac{n_0^I - t \cdot \overline{s}^t(0, t)}{\overline{s}^t(t_{n^t}, t_{n^t+1})} \tag{III.28}$$

TaScaaS will split the job in more partitions to satisfy the condition. To calculate the number of required new partitions, TaScaaS assumes that the new partitions will be processed by workers with an equal mean speed determined by the equation III.29,

$$\overline{s} = \frac{\overline{s}^t(t_{n^t}, t_{n^t+1})}{N^p} \tag{III.29}$$

where N^p is the number of actual partitions. This assumption is necessary because TaScaaS does not have information about where the new partitions will be executed. Thus, the new speed is expected to increase, in mean value, as,

$$\overline{s}^t(t_{n^t}, t_{n^t+1}) \rightarrow \overline{s}^t(t_{n^t}, t_{n^t+1}) \frac{N_f^p}{N_0^p} \tag{III.30}$$

where N_f^p and N_0^p corresponds to the total number of partitions after and before the split respectively. Then, replacing equation III.30 in III.28, we obtain the number of required partitions to achieve the time constraint at each instant t,

$$N_f^p = \left(\frac{N_0^p}{t_{max} - t}\right)\left(\frac{n_0^I - t \cdot \overline{s}^t(0,t)}{\overline{s}^t(t_{n^t}, t_{n^t+1})}\right) \tag{III.31}$$

Finally, the number of new partitions (N_{new}^p) will be assigned according to the maximum number of partitions constrain (N_{max}^p),

$$N_{new}^p = \begin{cases} N_f^p - N_0^p & N_f^p \leq N_{max}^p \\ N_{max}^p - N_0^p & N_f^p > N_{max}^p \end{cases} \tag{III.32}$$

Notice that the relation of the equation III.27 will be, usually, false. Therefore, this procedure is repeated during each job execution to adapt the number of partitions according to the new measured speeds. Also, although the number of partitions can be reduced according to the equation III.31, TaScaaS will not decrease that number to avoid continuous partition deletions and creations on fluctuating environments. Thus, depending on the behaviour of the performance, in some cases TaScaaS could produce and maintain an overpartitioning, causing the job to finish earlier than the specified time constraint. This effect can be caused in the specific case where the performance maintains a constant increase trend after a significant degradation measured in the previous intervals. Nevertheless, this approach ensures that the time constraint will be achieved, and ensures the synchronisation of partition processing times due the load balance system.

To compare the TaScaaS dynamic partitioning approach with a static one, consider now that no load balance system is used to balance the job partitions. Instead, the number of partitions, and their assigned iterations, is defined initially and can't be changed. In this scenario, the execution time of each partition is determined by the equation III.33,

$$t_j = \frac{n_j^I}{\overline{s}_j(0, t_j)} \tag{III.33}$$

where n_j^I is the number of iterations assigned to the partition number j. Therefore, the whole job execution time will be determined by the maximum value of t_j. In the better case, the approach used to perform the iteration partitioning will produce exactly the same execution time $\bar{t}$ for all partitions,

$$t_j = \bar{t} \quad \forall j \tag{III.34}$$

and the corresponding process speed will be,

$$s^t(0,\bar{t}) = \frac{n^I}{\bar{t}} = \sum_{j=1}^{N^p} \frac{n_j^I}{\bar{t}} = \sum_{j=1}^{N^p} \overline{s}_j(0,\bar{t}) \tag{III.35}$$

which is equivalent to the equation III.21, meaning that, in the best possible case, we can reproduce a global execution speed equivalent to the balanced approach. Also, the number of partitions cannot be adapted to compensate performance drops. Thus, normally, a static approach will achieve worst results, as we will discuss in the section 4.5 applied to the results obtained in the section 4.4.

4.4 Results

We tested the TaScaaS behaviour using three different *worker infrastructure* types simultaneously. The first *worker infrastructure* has been deployed on an on-premises cloud and consists on 32 slots running on Intel Xeon (Skylake, IBRS) processors. The second worker infrastructure has been deployed on the EGI Federated Cloud (IFCA-LCG2 site) and consists on 16 slots running on Intel(R) Xeon(R) CPU E5-26700 @ 2.60GHz processors. Finally, the third consists on a single computer with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz processor, which has been assigned a maximum of 8 slots. These three *worker infrastructures* will be used to process concurrently all the incoming *jobs* via the TaScaaS service.

To perform the tests, due the importance of the radiation transport simulations in clinical applications and the long execution times involved, we have executed Monte Carlo simulations using PenRed [191], which is a framework for radiation transport simulations using Monte Carlo techniques. Its executions are iterative-based where each iteration corresponds to the simulation of a primary particle and all the secondary particles produced by interactions with matter. Being a probabilistic process, the required computation time of each iteration usually differs. However, as the number of simulated iterations must be big enough to reduce the statistical uncertainties, the mean speed measured in iterations per second converges to a stable value. Thus, to be able to control the expected execution time and force the usage of different number of partitions for each job, we have defined a set of simulations. Each one has a different execution time constraint to be achieved by TaScaaS, but exactly the same configuration and number of iterations to simulate: 10^6 primary particles.

The executed simulation corresponds to one of the examples provided in the PenRed package, specifically the $1-disc-vr$ example, which details and description can be found in their documentation and github repository[16]. This one consists on a point source of monoenergetic electrons with a energy of $40KeV$. The beam

[16]https://github.com/PenRed/PenRed

Taula III.7: Job types with the corresponding time constraint, in seconds, and the required computational power relative to type 1.

Type	Time Constraint (s)	Relative cost (%)
1	3600	100
2	2700	133
3	2100	171
4	1500	240
5	900	400

aims to a cylindrical copper phantom. However, as TaScaaS is agnostic to the running application, the results are valid for other simulations and applications that meet the already discussed characteristics. All the required configuration, data base and geometry files are included in the example provided by PenRed, thus can be executed directly once the code has been compiled following the instructions of the corresponding documentation.

Since the number of mean operations for each simulation converges to the same value, the differences on the required computation power is determined by the execution time constraint. Furthermore, we have checked that the measured speed, in an isolated environment, requires a few minutes to stabilise, being very stable after $3 - 5$ minutes. Thus, to create jobs with different computational power requirements, a set of 4 jobs have been created, whose characteristics are summarised in the Table III.5. The simulation time constraints for each job type have been set to 3600, 2700, 2100, 1500 and 900 seconds, expecting to require more partitions for lower time constraints. Taking as a reference the computing power required by job executions of the first type, the following types require a 133%, 171%, 240% and a 400%, respectively, of the computational power required by the type 1 simulations. This selection provides a wide range of jobs with different requirements.

During the experimentation, we will measure the execution time in seconds of each submitted job and the number of created partitions to fit the time constrain. In addition, all the reports done by each job partition of each job have been registered to evaluate the performance behaviour of the system. This data includes the speed, measured in iterations per second, of each time interval, the timestamp of each report, and the evolution of the assigned iterations to each partition.

For the following analysis, we have considered that the time spent by TaScaaS to balance the execution is minimal. This is because the used balancer system has been designed to introduce a negligible overhead on the whole execution. Moreover, we have extracted the execution time information of the TaScaaS Lambda functions to ensure that assumption. The minimum execution time for all the

Lambda invocations is approximately 300 ms, the maximum 1000 ms and the mean value 400 ms. Considering that TaScaaS has been configured to perform a mean value of 20 reports during the execution of each *job partition*, and considering also synchronous blocking communications, in the worst case, the overhead introduced by the TaScaaS processing time is 20 seconds. Since the mean execution time of the lambda functions is lower than 1 second, this overhead is very overestimated. Furthermore, all Lambda functions have been configured with a capacity of 512 MB. Therefore, this overhead could be reduced increasing the computing power of the TaScaaS functions. Nevertheless, considering that the execution time of each simulation partition is on the order of hundreds or thousands of seconds, this overhead is effectively negligible. Regarding communication time, the message data required by the load balancer is lesser than 1 KB, thus, the communication time can be neglected too. Furthermore, the communication can be done asynchronously to continue the execution while waiting for the TaScaaS response.

The tests have been performed during almost three hours launching jobs with different time constraints in irregular intervals and frequencies. The results are shown in Figure III.44, where the purple line represents the number of concurrent active partitions for each timestamp and the green line represents the number of slots required by TaScaaS to process all the workload and finish the executions in the user-defined time limit. The figure shows how TaScaaS fits the incoming workload and maintains a pool of free slots to be able to process a possible peak of incoming jobs or new partitions. Then, when TaScaaS neither receives new jobs nor requires to launch more partitions, it decreases the number of required slots efficiently.

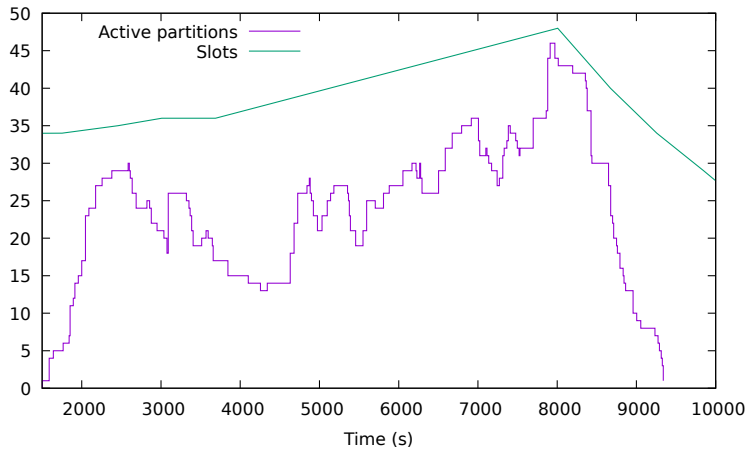

Figura III.44: Number of active partitions running on the worker infrastructures (dark) and number of required slots by TaScaaS (light) at each timestamp.

Notice that TaScaaS not only must be prepared to allocate new incoming jobs, but also new partitions of the currently running jobs. Since TaScaaS does not have any knowledge about the running applications nor the heterogeneity behaviour in the environment where the applications run, it only relies on the reported speeds during the execution to allocate new partitions to meet the configured execution time. Thus, a job that initially runs a single partition could require many more partitions to achieve the time constraint. Furthermore, our tests have been done in the worst scenario. First, we have launched all the jobs with only a single initial *partition*, relying on TaScaaS to scale the number of *partitions*. Secondly, we have fixed the scaling step time to 300 seconds, which is relatively low compared to the application execution times and forces TaScaaS to perform predictions with less statistics.

In a real application the user could estimate a mean value of the required partitions and set it during the job configuration. For example, using TaScaaS itself, the user could create sufficient jobs to fill the worker infrastructures and get an upfront distribution of the required partitions per job. As the worker infrastructures are working at full capacity, our test scenario can be considered as the slowest, providing a mean value of the number of required partitions in the worst case. This approach avoids TaScaaS to launch several new partitions during the job execution, providing a more predictable workload input.

Furthermore, the scale step time must be selected as a compromise between response speed and resources allocation accuracy. A faster scale step will provide a faster response time for peaks and troughs of new *partitions*. However, this produces larger fluctuations on the workload measures due to the lower statistic, providing a less accurate allocation of resources, like the case shown in Figure III.44. Notice that a slower scale step will provide a more accurate, and probably stable, value of the mean workload. Therefore, the allocated resources will be less underused. However, the system will be less responsive to new *partition* peaks, which may cause some jobs to take longer to start in this case. The election of these configuration parameters depends strongly on the executing applications, the execution time ranges, the worker infrastructures, the rate of incoming jobs and its variability, etc. Nonetheless, the user can obtain an occupation diagram like the shown in the Figure III.44 to be able to adjust these parameters. Nevertheless, the results show that TaScaaS can handle adverse configurations efficiently.

Turning to the ability to meet the execution time constraints, we have grouped the simulations with the same time limits. Figure III.45 represents the execution time with points, the number of used partitions is represented by the blue histogram, and the time constraint by the red line. This one corresponds to the jobs of 1500 seconds. As we can see, all the jobs satisfy the time limit. Although job number 6 slightly exceeds the execution time goal, the excess time is negligible

compared to the total execution time. Therefore, TaScaaS does not request a new worker in order to save resources.

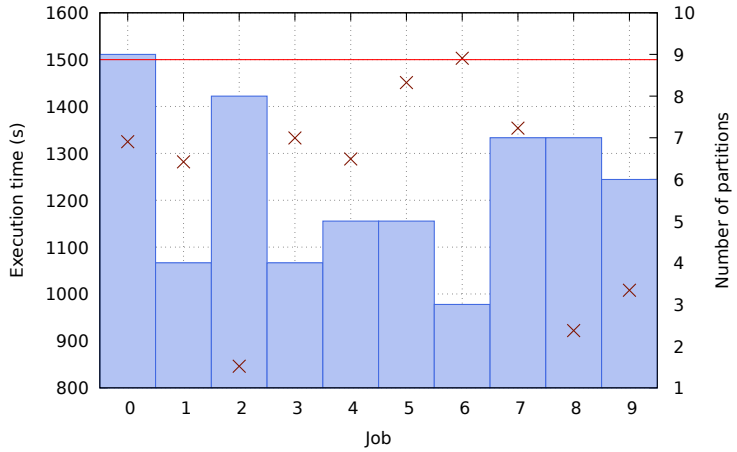

Figura III.45: Execution times (points) and number of partitions (blue histogram) required by each job with a time constraint of 1500 seconds (red line).

Notice that the execution times present huge differences among jobs which perform the same simulation. This is because each *worker infrastructure* have different capabilities and TaScaaS does not know where a job partition will be executed. This effect could be exacerbated if the *worker infrastructures* present heterogeneity on their *workers* capabilities, either because differences with the underlying hardware or because the resources sharing among both, own *partition* executions and tenants. Also, this causes important differences on the number of required partitions, even between jobs with similar execution times.

The simulations with the other execution time limits presents the same behaviour, as shown in Figures III.46 and III.47 for 2100 and 2700 seconds respectively.

An explanation for the differences in the number of used partitions for similar simulations could be the heterogeneity of the worker infrastructure processors. However, the *noisy neighbour* has also a non-negligible effect. To exemplify this fact, Figures III.48 and III.49 represents the performance behaviour evolution of the partitions of two specific jobs i.e. the time evolution of the simulation speed for each partition. In these figures, the boxes on the X axis mark the partition start, the vertical lines indicate the first speed report performed by the partition, and the remaining line shows the mean speed evolution. Notice that at the partition start TaScaaS does not have a speed measure, and this is why it is shown as 0 in the figure.

The first one, corresponds to the job number 1 from type 2 jobs set, which has a time constraint of 2700 seconds. According to the data shown in the Figure III.47,

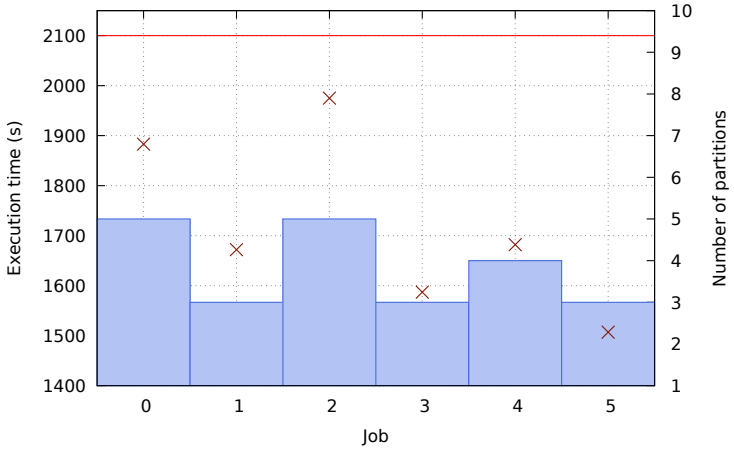

Figura III.46: Execution times (points) and number of partitions (blue histogram) required by each job with a time constraint of 2100 seconds (red line).

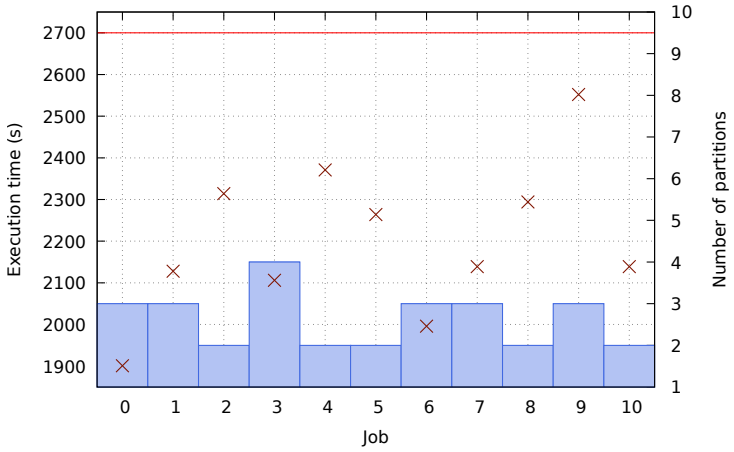

Figura III.47: Execution times (points) and number of partitions (blue histogram) required by each job with a time constraint of 2700 seconds (red line).

TaScaaS has created three partitions to fit the time constrain, whose individual speeds evolution are represented by each colour line. The figure, shows a simple behaviour where the execution of each *partition* presents an approximately constant average speed. So, in this case, the environment presents insignificant time dependent fluctuations and is only affected by a constant heterogeneity, probably caused by hardware differences. Once the first partition starts to process (Purple box), at the second box (Green) TaScaaS calculates that the *job* require one more *partition* to meet the execution time limit. However, this second *partition* has been received by a slower *worker*, and is not sufficient to meet the time. To solve it,

TaScaaS requests a new *partition*, which, this time, is processed by a faster *worker*. Since TaScaaS works with the mean speed of all partitions, after the second job begins execution, it estimates that the *job* requires a third *partition* with a speed of, approximately, 200 iterations per second. Since the third *partition* is faster, the execution time will be significantly lower than the configured limit, which can explain some of the discrepancies seen between the execution times of the same kind of simulations.

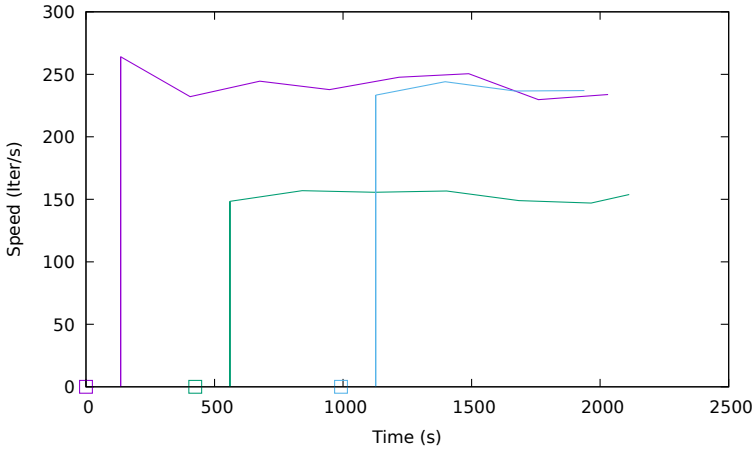

Figura III.48: Execution speed at each timestamp for the job number 1 from type 2 jobs set.

Secondly, Figure III.49 corresponds to the job number 0 from type 4 jobs set. This one has a time constraint of 1500 seconds and, according to the data in Figure III.45, TaScaaS has created 9 partitions to achieve it. The figure shows a constantly decrease of the speed of each partition. So, as opposed to the previous case, the system capabilities fluctuate during the execution. More specifically, the system decreases its performance and, therefore, this could not only be caused due to the hardware heterogeneity. To mitigate this issue, TaScaaS creates new partitions several times to compensate the continuous decreasing performance, and meet the time constraint. Notice that the execution time of this application is not expected to decrease as the execution advances, as we have explained before and proved in several executions, like the one shown in Figure III.48.

As the average speed of this application is expected to be constant if the environment conditions do not change, this behaviour is caused by the competition of several processes for the same resources. It could be caused by both the interference of our own partitions running on the same workers and by the effect of tenants (noisy neighbours). Nevertheless, TaScaaS has handled correctly both cases, creating new *partitions* only when needed. Notice that the *partitions* of the same *job*

can be executed by any *worker* of any *worker infrastructure*. Thus, increasing the number of *partitions* does not necessarily overload the same *worker*.

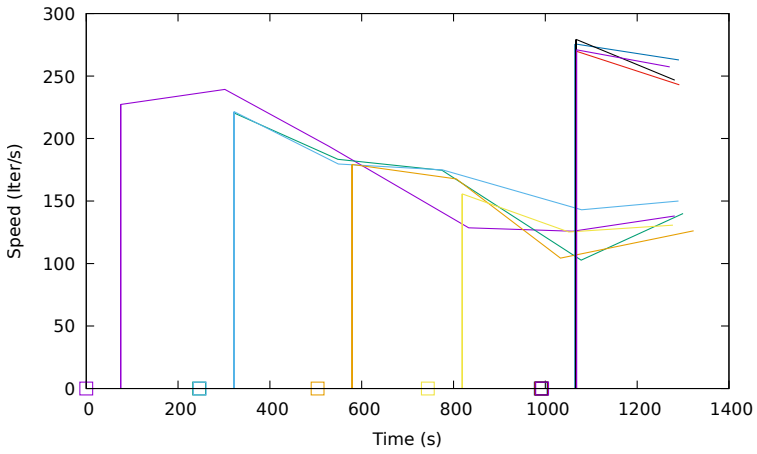

Figura III.49: Execution speed at each timestamp for the job 1 from the simulations with a time constraint of 1500 seconds.

This results demonstrate that TaScaaS can correctly handle the capability variability on heterogeneous environments correcting its effect on the execution time. Notice that TaScaaS has no information about the cause of the speed fluctuation, so it can handle also applications whose speed is not constant throughout the execution.

4.5 Discussion

To quantify the efficiency improvement achieved by TaScaaS, following, we will compare the results obtained in section 4.4 with a static partitioning approach with no balance, as many of the works discussed in section 4.2. So, for that analysis, we will assume that each job is partitioned according to the system performance information, which can be measured from previous job executions. However, the iterations cannot be reassigned once the partitions have been sent to be computed. As the possible combination of the characteristics involving a single job executions are too big, considering job types, number of partitions, resource where the execution is computed, possible performance fluctuations etc. the study will be carried out using the measured data from the previous experiment, which will allow to compare both results.

First, in Figure III.50, the mean speed of each job partition execution is represented by points. It can be seen that these points are split in two groups according to the measured speeds, which corresponds to the two different types of points in

Figure III.50. In addition, the line of each group represents the fitted evolution of the mean speed according to the measured data. The area surrounding the line represents one standard deviation of the fitted model. Although we could classify the measures according to the worker hardware where the partition has been executed, in many providers services the information about the underlying hardware where the execution is carried out is not, or only partially, accessible. Therefore, we will classify the resource types according to the observable measures. However, notice that the fits standard deviations of each group are about 15% and 2% for the slower and faster group respectively, which are significantly lower than the performance fluctuations measured in different cloud services by the works discussed in the section 4.1.

Moreover, the measured mean speeds of a single partition, whose execution is performed completely in the same worker, presents standard deviations up to 50% and 10% for the slower and faster group. Thus, considering that the speed in this kind of applications is determined by the slowest process, because the processes are not balanced, the expected real results will be worst than the predictions of our fitted models. This fact can be seen in the residuals of both fits, represented in Figures III.51 and III.52 for the slower and faster group respectively.

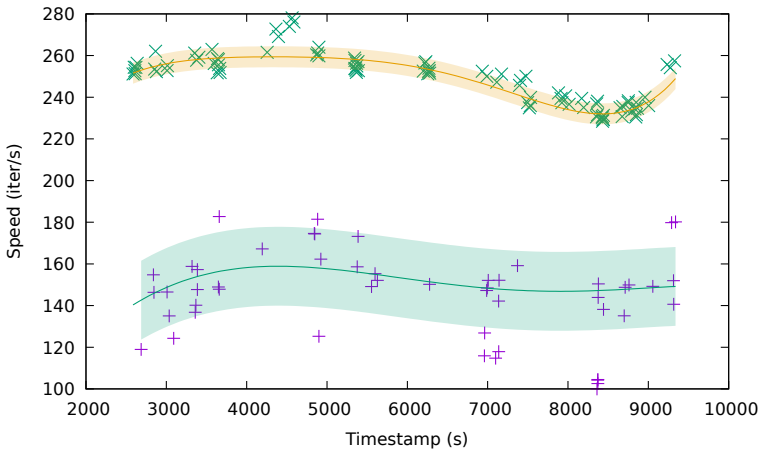

Figura III.50: Mean execution speed for each processed partition (points). The measures are divided according to two speed levels. The lines represents the fitted time dependency of the mean speed for each set of points. The area surrounding the line represents one standard deviation of the fitted model.

Then, as we discussed in the section 4.3.3, the job execution can be estimated using the equation III.33 applied to the slowest partition, whose mean speed can be predicted by the fitted models. As we cannot know the specific speed of each partition before the execution starts, we will assume that all partitions computed in

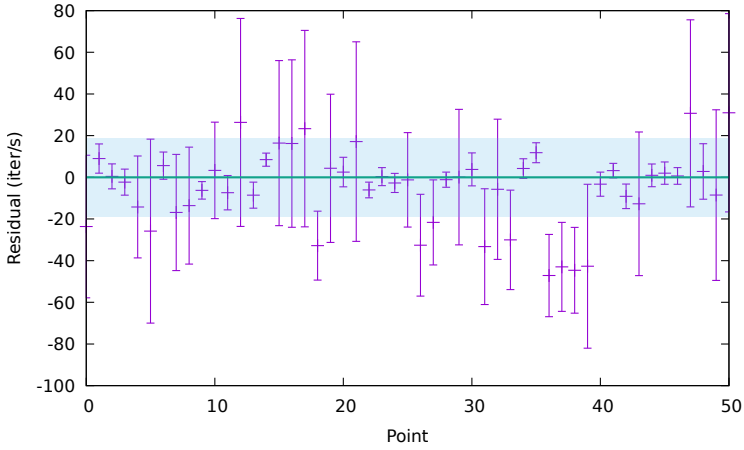

Figura III.51: Residuals of the fit from the slower group. Error bars represent one standard deviation of measured points. The area surrounding the line represents one standard deviation of the fitted model.

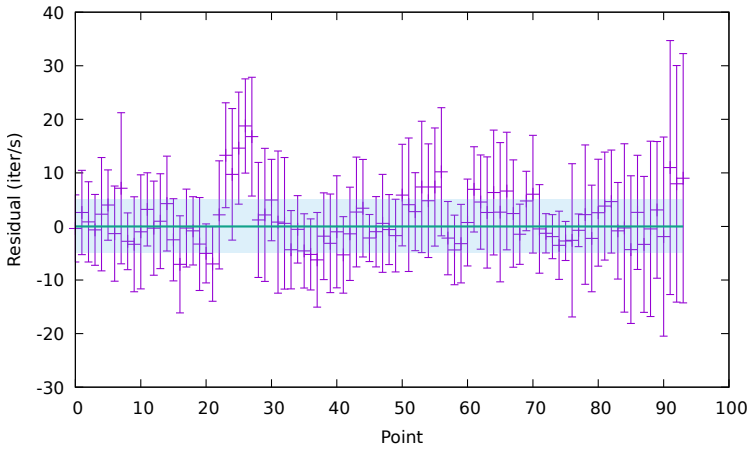

Figura III.52: Residuals of the fit from the slower group. Error bars represent one standard deviation. The area surrounding the line represents one standard deviation of the fitted model.

the same performance group have an equal iteration assignation. Also, to minimise the differences in execution time of partitions executed in different groups, the iteration assignation for the partition of each group must satisfy the relation,

$$\frac{n_f^I}{\overline{s}_f(t)} = \frac{n_s^I}{\overline{s}_s(t)} \tag{III.36}$$

where n_f^I and n_s^I are the number of iterations assigned to each partition executed

in a fast and slow worker respectively, and $\overline{s}_f(t)$ and $\overline{s}_s(t)$ the corresponding mean speeds. Then the number of assigned iterations to partitions of the slower group can be obtained as follows,

$$\begin{aligned} n^I &= n_f^I n_f + n_s^I n_s \\ n^I &= n_s^I \frac{\overline{s}_f(t)}{\overline{s}_s(t)} n_f + n_s^I n_s \\ n^I &= n_s^I \left(\frac{\overline{s}_f(t)}{\overline{s}_s(t)} n_f + n_s \right) \\ n_s^I &= \frac{n^I}{\frac{\overline{s}_f(t)}{\overline{s}_s(t)} n_f + n_s} \end{aligned} \tag{III.37}$$

and n_f^I can be calculated using the equation III.36. Also, to fit the time constraint, the condition of the equation III.38 must be satisfied,

$$t_{max} < \frac{n_i^I}{\overline{s}_i(t)} \quad \forall i \in s, f \tag{III.38}$$

where i is the partition index and n_i^I takes the n_f^I or n_s^I values depending on the group to which it belongs, the fast or the slow respectively. However, if we suppose normally distributed mean speeds around our fitted model, the partitions execution time will have a non negligible probability of not satisfying the equation III.38. As a single slow partition will delay the whole run, the probability to produce a delay is equivalent to the probability to have a single slow partition. To quantify this effect, the Table III.8 shows the delay intervals corresponding to 1σ, 2σ, 3σ and greater than 3σ for both groups, the fast and the slow. For each interval, the dependency of the probability with the number of partitions of each group is shown, and the specific probability for 5 partitions has been calculated, which is, approximately, the mean number of partitions used in our experimentation. In the worst case, the delay produced in the whole job execution time will be equal to the top limit of each delay interval, which possibly causes the condition of the equation III.38 to not be met.

Notice also that, due the symmetry of the distribution, the same probabilities can be applied to partitions which mean speed is faster than the model prediction. Thus, these analysis can be also used to calculate the probability to have underused resources. Although the mean speed of our model can be artificially decreased to increase the probability to satisfy the time constraint, for example multiplying the speed by a "security" factor $\rho \in (0, 1) \quad \setminus \quad \overline{s}'(t) = \rho \overline{s}(t)$, this will cause an increment of the required resources to perform the calculus, producing an unnecessary resources over-provisioning. Moreover, this method will not handle

Taula III.8: Expected delay probabilities caused by performance fluctuations for both groups, fast and slow. The variable n, represents the number of partitions belonging to the specific group.

Delay(fast)	Delay(slow)	Probability	Probability $n = 5$
$(0, 2]\%$	$(0, 15]\%$	$1 - (0.659)^n$	87.57%
$(2, 4]\%$	$(15, 30]\%$	$1 - (0.864)^n$	51.85%
$(4, 6]\%$	$(30, 45]\%$	$1 - (0.979)^n$	10.07%
$> 6\%$	$> 45\%$	$1 - (0.999)^n$	0.5%

the differences in partitions execution speeds, remaining faster resources potentially unused when their partial execution finishes. Furthermore, this method will produce an excess of partitions, which will increase the post-processing cost of the partial results. Depending on the application and the quantity of generated data, an excessive partitioning could produce post-processing times comparable to the execution time.

However, notice that if the fluctuations of our infrastructures are sufficiently low, like the faster performance group, the delays could be assumed and the use of a static approach may be good enough. Thus, to evaluate the suitability of using TaScaaS, could be useful to perform a benchmark following the procedure discussed in this section.

4.6 Conclusion

In this work, we presented TaScaaS an open source serverless job scheduler and load balancer service to distribute and balance jobs among multiple heterogeneous infrastructures deployed or accessed by the user. As it is deployed on AWS Lambda, it benefits from the AWS free tier, minimising the cost of its execution. Also, TaScaaS is created as a serverless application, so it produces a cost only when it is used. Moreover, AWS Lambda provides a highly scalable environment, thus TaScaaS is capable to handle a large number of simultaneous workers.

We have demonstrated how TaScaaS overcomes static partitioning approaches depending on the performance fluctuations of the available infrastructures, which affect not only public cloud providers, but also on-premises and federated cloud infrastructures. In addition, TaScaaS has proved its capabilities to handle efficiently both kinds of heterogeneity, on hardware and due to sharing resources across multiple tenants. Furthermore, TaScaaS correctly handles time constraints in the execution time in this kind of environments.

In future versions of TaScaaS we will implement improvements such as an adaptive system to select the scale step time and change it according to the incoming workload, and support to deploy the TaScaaS service on other cloud providers. As

it is accessed via HTTPS requests, the *worker infrastructures* are agnostic about where the TaScaaS back-end is running, and changing the provider require no changes on the infrastructure side. We will also investigate its behaviour in scenarios of computing continuum where resources from the edge are used in coordination with resources from on-premises and public Clouds. This will allow to achieve load balancing across highly heterogeneous computing platforms across a variety of infrastructures.

Acknowledgment

This work was supported in part by the Spanish "Ministerio de Ciencia e Innovación" for the project SERCLOCO with reference number PID2020-113126RB-I00, by the program "Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d" under grant number ACIF/2018/148 from the Conselleria d'Educació of the Generalitat Valenciana, Spain and the "Fondo Social Europeo" (FSE). This work was supported by the project AI-SPRINT "AI in Secure Privacy-Preserving Computing Continuum" that has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant 101016577. The authors would also like to thank the regional government of the Comunitat Valenciana (Spain) for the project IDIFEDER/2018/032 (High-Performance Algorithms for the Modeling, Simulation and early Detection of diseases in Personalized Medicine), co-funded by the European Union ERDF funds (European Regional Development Fund) of the Comunitat Valenciana 2014–2020.

The authors would like to thank the EGI Applications on Demand service to provide part of the resources used for this work. This one is co-funded by the EOSC-hub project (grant number 777536). The HNSciCloud project (grant number 687614) is also sponsoring the service, allowing users to access the HNSciCloud services pilot for limited scale usage using the voucher schemes provided by the two contractors: T-Systems and Exoscale.

Capítol IV

Resultats i Conclusions

1 Discussió general dels resultats

A la present secció, es discutiran els resultats obtinguts durant el desenvolupament de la tesi, tant el programari desenvolupat, com les publicacions en revistes científiques, congressos i comunicacions.

1.1 Programari desenvolupat

Durant el transcurs de la tesi del present document, s'ha desenvolupat una sèrie de programari destinat a l'ús en el còmput científic. L'esmenat programari es llista a continuació,

- **MARLA**[1]: Com a part de l'estudi de l'estabilitat i prestacions de la plataforma AWS Lambda, s'ha desenvolupat la plataforma *MARLA*, la qual proporciona un entorn automatitzat per al processament de dades basades en el model *MapReduce*. Gràcies a l'ús de components *serverless*, *MARLA* és altament escalable. A més, la discussió del Capítol II proporciona pautes clares per avaluar i maximitzar la seua eficiència.

- **APRICOT**[2]: Centrant-se ara en la reproductibilitat i facilitar l'ús d'eines de desplegament automàtic d'infraestructures, *APRICOT* proporciona una interfície gràfica per al desplegament d'infraestructures i és compatible amb diferents proveïdors de computació en el núvol. Aquesta interfície es troba integrada en l'entorn Jupyter, el qual s'empra comunament tant en l'àmbit científic com en l'acadèmic. Proporciona, a més, la possibilitat de compartir

[1] MARLA: `https://github.com/grycap/marla`
[2] APRICOT: `https://github.com/grycap/apricot`

la configuració, tant de maquinari com de programari, de les infraestructures desplegades, facilitant la tasca de reproducció per part d'altres investigadors.

- **RUPER-LB**[3]: Per tal de mitigar les fluctuacions de prestacions i heterogeneïtats de maquinari, s'ha desenvolupat la llibreria *RUPER-LB*. Aquesta, es capaç de balancejar execucions d'aplicacions iteratives dèbilment acoblades tant a nivell de fils, processos MPI, d'execucions distribuïdes a través de la xarxa o una combinació de tots tres. S'ha mostrat que el sobrecost produït en el temps d'execució és menyspreable i que és altament escalable gràcies a que s'ha seguit un model asíncron.

- **TaScaaS**[4]: A partir del desenvolupament de *RUPER-LB* i les tècniques apreses durant el desenvolupament de *MARLA*, s'ha desenvolupat la plataforma TaScaas, la qual es desplega sobre una infraestructura completament *serverless* per tal de minimitzar costos i aconseguir una alta escalabilitat. TaScaaS és capaç de gestionar els treballs a processar en un sistema constituït per diverses infraestructures de còmput independents. A més, calcula la capacitat necessària per afrontar la càrrega de treball i escalar les infraestructures de còmput de forma adient.

1.2 Publicacions

Les publicacions desenvolupades durant el transcurs de la tesi són les següents,

1. V. Giménez-Alventosa, Germán Moltó, Miguel Caballer, A framework and a performance assessment for serverless MapReduce on AWS Lambda, Future Generation Computer Systems, Volume 97, 2019, Pages 259-274, ISSN 0167-739X, https://doi.org/10.1016/j.future.2019.02.057.

2. Vicent Giménez-Alventosa, José Damián Segrelles, Germán Moltó, Mar Roca-Sogorb, APRICOT: Advanced Platform for Reproducible Infrastructures in the Cloud via Open Tools, Methods Inf Med, Volume 59, doi:10.1055/s-0040-1712460

3. **Preprint associat a congrés:** V. Giménez-Alventosa, G. Moltó and J. Damian Segrelles, RUPER-LB: Load balancing embarrasingly parallel applications in unpredictable cloud environments, 2020, url: https://arxiv.org/abs/2005.06361

4. V. Giménez-Alventosa, G. Moltó and J. Damian Segrelles, TaScaaS: A Multi-Tenant Serverless Task Scheduler and Load Balancer as a Service, in IEEE Access, doi: 10.1109/ACCESS.2021.3109972.

[3]RUPER-LB: `https://github.com/PenRed/RUPER-LB`
[4]TaScaaS: `https://github.com/grycap/TaScaaS`

1.3 Presentacions orals

A més de les publicacions, el treball realitzat durant la tesi a donat lloc a les següents comunicacions orals en congressos,

1. RUPER-LB: Load Balancing Embarrasingly Parallel Applications in Unpredictable Cloud Environments, HPCS 2020.

2. PenRed: Un motor de Monte-Carlo extensible y paralelo para el transporte de radiación basado en PENELOPE, 7º Congreso Conjunto Sociedad Española de Física Médica y Sociedad Española de Protección Radiológica (23 SEFM - 18 SEPR)

2 Conclusions

Tots els desenvolupaments realitzats durant la tesi han seguit l'objectiu principal plantejat, el desenvolupament i adaptació d'execucions científiques a entorns de computació en el núvol. En primer lloc, s'ha realitzat un estudi exhaustiu de les característiques i prestacions dels serveis proporcionats per proveïdors de computació en el núvol i la seua adequació al còmput científic. A més, s'ha ampliat la informació disponible a la literatura sobre aquest tema a partir de l'estudi del treball de la secció 1, el qual es centra en la plataforma *serverless* de AWS Lambda. En aquest, s'han estudiat les variacions de les prestacions separant les causades per heterogeneïtats del maquinari de les causades per la gestió subjacent de la infraestructura on s'executa dita plataforma. S'ha arribat a la conclusió que ambdós factors produeixen un efecte no menyspreable tant en la velocitat de processament com en les comunicacions. De fet, s'han mesurat diferències en el temps de processament de fins a un 50% per a un mateix tipus de processador, agreujant-se si consideren mes heterogeneïtats del maquinari. No obstant, cal tenir en compte que la variabilitat depèn fortament del processador on s'executa la instància, sent molt menor en processadors més nous, per motius que no podem conèixer. A més, al cas de les comunicacions, al pitjor dels casos mesurats, una mateixa instància pot ser fins a dos ordres de magnitud més lenta que la mitjana. Tot i que a la majoria d'instàncies la velocitat de la transferència de dades és manté prop d'un valor mitjà, a una aplicació científica, una única instància pot enrellentir l'execució completa. Per tant, ambdós efectes són factors a tindre en compte si es volen realitzar execucions en entorns *serverless*.

Els esmenats estudis donaren lloc a dos problemàtiques principals. D'una banda, al assumir que els proveïdors de computació en el núvol ofereixen prestacions fluctuants i que, aquestes, són impredictibles, es fa patent la necessitat de desenvolupar un sistema per a mitigar aquest efecte. D'altra banda, per assegurar la

reproductibilitat dels resultats d'experiments computacionals, cal un sistema per facilitar el desplegament d'infraestructures en proveïdors de còmput en el núvol, i que aquestes puguen ser descrites i reproduïdes a la pròpia experimentació.

En primer lloc, per afrontar el problema de la reproductibilitat, es va desenvolupar *APRICOT*, proporcionant així una metodologia clara per a definir el desplegament i ús d'infraestructures en proveïdors de computació en el núvol. Actualment, *APRICOT* suporta el desplegament en els proveïdors privats OpenStack, OpenNebula i el proveïdor públic AWS directament a través de la interfície d'usuari i, a més, la resta de proveïdors suportats per el IM (Azure, OTC, Orange, etc.) emprant-lo des de la consola. A més, *APRICOT* ha sigut desenvolupat dins de l'entorn Jupyter, aprofitant així les seues característiques i popularitat en l'àmbit científic i acadèmic. A través dels *notebooks* de Jupyter, una experimentació pot ser completament descrita a un mateix document, incloent les característiques de la infraestructura necessària a desplegar amb el *plugin* d'*APRICOT*. Més encara, s'ha incorporat suport per a emprar el sistema d'emmagatzematge de dades Onedata [126] a través del propi *notebook* de Jupyter, brindant així la possibilitat de definir l'accés a les dades al mateix document. Totes aquestes característiques fan d'*APRICOT* un entorn adient per a la creació de Ciència Oberta per a l'experimentació amb infraestructures computacionals complexes, ja que, tots els passos i detalls de l'experimentació, des de la creació de la infraestructura, passant per l'accés a les dades i finalitzant amb el seu processament, es troben integrades a un mateix document interactiu.

En segon lloc, per tal d'afrontar el problema de les fluctuacions en les prestacions, s'ha dissenyat la llibreria *RUPER-LB*, durant el desenvolupament de la qual s'han mesurat fluctuacions en la velocitat de processament del servei EC2 de AWS majors del 35%, reforçant així la hipòtesi inicial i la justificació del desenvolupament de l'esmenada llibreria. A més, s'ha mostrat la capacitat de la llibreria per compensar aquests efectes i maximitzar l'eficiència de les infraestructures desplegades, tant en proveïdors públics com privats. Més encara, s'ha comprovat que es pot augmentar l'eficiència d'una execució balancejant els fils executats en un mateix processador amb múltiples cors en un entorn virtualitzat.

Tot i que la restricció de *RUPER-LB* a processos iteratius dèbilment acoblats pot semblar, a primer vista, limitant, aquests són molt comuns en l'àmbit científic. Per exemple, molts tipus de simulacions de Monte Carlo s'identifiquen en aquest model d'execució i, en específic, les simulacions de transport de radiació a través de la matèria, que són una eina clau tant en el desenvolupament de la física mèdica actual, o en els estudis de protecció radiològica.

RUPER-LB implementa el balanceig de càrrega a dos nivells, tant entre múltiples fils, entre processos MPI, com en execucions combinades amb ambdós models. A més, s'ha dissenyat per a processar el balanceig de forma asíncrona, de forma

que es minimitzen els punts de sincronització entre fils o processos i, al seu torn, es minimitza el sobrecost introduït pel balanceig de càrrega en el temps d'execució. Aquestes característiques el fan idoni per a aplicacions paral·leles amb poca comunicació. A més, s'ha dissenyat per a ser executat en entorns heterogenis sense suposar cap informació prèvia de les prestacions del maquinari, el que, com s'ha demostrat, li permet adaptar-se correctament a entorns amb prestacions que fluctuen constantment i de forma impredictible.

Una vegada desenvolupada, per tal de preparar la llibreria *RUPER-LB* per a ser executada en una plataforma *serverless*, es va extendre la seua funcionalitat per a poder balancejar processos executats de forma distribuïda, que es comuniquen amb un servidor a través de la xarxa. Donat que les aplicacions objectiu d'aquesta llibreria empren poques comunicacions, i donada l'aproximació asíncrona del balanceig, les prestacions de la xarxa no són un factor significatiu per al bon funcionament de la llibreria, obtenint també una alta escalabilitat i correcte funcionament en balanceig de còmput distribuït.

Els desenvolupaments anteriors formaren el fonament per a l'últim treball. Com s'ha discutit al llarg del present document, a l'àmbit científic és molt comú l'ús de múltiples infraestructures independents per tal d'afrontar l'alt cost computacional d'alguns experiments. Per aquest motiu, i amb l'adaptació de *RUPER-LB* al còmput distribuït, s'ha desenvolupat la plataforma *TaScaaS*, la qual permet el balanceig i distribució de treballs entre diferents infraestructures independents i, en general, heterogènies. Aquestes infraestructures poden haver sigut desplegades per l'usuari, o que simplement siguen accessibles a aquest, donat que no requereixen instal·lar cap dependència addicional. El propi procés executant-se en la infraestructura és el que es comunicarà amb la plataforma *TaScaaS* mitjançant peticions HTTP, per tal de dur a terme el balanç de la càrrega. *TaScaaS* automatitza el processament distribuït entre múltiples infraestructures a l'hora que calcula la capacitat de processament requerida per aquestes en funció de la càrrega de treball entrant. Amb aquest càlcul, informa a les infraestructures per a que, de ser possible, escalen els seus recursos en conseqüència, evitant així el malbaratament i subaprovisionament de recursos.

En definitiva, amb els desenvolupaments realitzats durant la tesi, s'han abordat les diferents problemàtiques existents per a la l'experimentació científica en entorns de computació en el núvol. Aquests desenvolupaments s'han centrat en un tipus concret d'aplicacions, que són aquelles amb un paral·lelisme poc acoblat i que, per tant, requereixen un nombre reduït de comunicacions. L'elecció d'aquesta limitació, com s'ha justificat al capítol I secció 3.2, es deu a les característiques de les prestacions en entorns de computació en el núvol en el moment del desenvolupament de la tesi. Queda, per tant, l'adequació d'aplicacions paral·leles fortament acoblades per a treballs futurs.

3 Treball futur

Els treballs realitzats i discutits al present document donen peu a continuar diferents vies d'estudi. En primer lloc, per tal de maximitzar l'eficiència dels serveis dels proveïdors de còmput en el núvol, desenvolupar models de cost més complets i precisos on es consideren diferents serveis, proveïdors i tipus d'instàncies. Com s'ha vist al capítol II, un estudi previ de costos és essencial per evitar el malbaratament dels recursos i maximitzar la productivitat.

En segon lloc, estendre la funcionalitat de la plataforma TaScaaS per a que s'analitze l'adequació dels treballs a les diferents infraestructures disponibles per a realitzar les execucions. A partir dels paràmetres de la configuració dels treballs, seria possible emmagatzemar informació dels temps d'execució total de les particions dels treballs en funció de la plataforma on s'executen i de diferents paràmetres, com la càrrega de treball de dita plataforma, l'hora i dia de la setmana, l'aplicació que està executant-se etc. A partir d'aquestes dades, es podria crear un model per elegir la infraestructura òptima i millorar així l'eficiència conjunta de totes les infraestructures.

Finalment, i possiblement el més extens, consistiria en realitzar desenvolupaments similars als presentats a aquesta tesi però considerant aplicacions paral·leles altament acoblades. Com s'ha vist al capítol I secció 3.2, els ràpids avanços per part dels proveïdors de còmput estan aconseguint que aquest tipus d'entorns siguen cada vegada més competitius per a l'execució d'aplicacions altament acoblades amb un ús freqüent de la xarxa entre els nodes involucrats. Cal remarcar que, probablement, els resultats que s'obtingueren en aquesta via d'estudi resultarien poc adequats per aplicacions feblement acoblades, ja que el sobrecost involucrat serà major que en els treballs presentats. Per tant, l'objectiu no és substituir els resultats presentats, sinó realitzar nous desenvolupaments per a un cas d'ús diferent. Al seu torn, serà interessant analitzar els costos a partir de models per maximitzar la rendibilitat i eficiència de les execucions en aquest cas.

4 Finançament

Els treballs desenvolupats a la tesi presentada, han sigut finançats per les següents entitats,

- Subvencions per a la contractació de personal investigador de caràcter predoctoral, Generalitat Valenciana, Fons Social Europeu (FSE):
  Nº ref: ACIF/2018/148

- Ministerio de Economía, Industria y Competitividad: Projecte: *Computa-*

ción Big Data y de Altas Prestaciones sobre Multi-Clouds Elásticos (BigCLOE), Nº ref: TIN2016-79951-R

- European Commission, Horizon 2020: Projecte: *PRedictive In-silico Multiscale Analytics to support cancer personalized diaGnosis and prognosis, Empowered by imaging biomarkers* (PRIMAGE), Nº: 826494

Agraïments

Agrair, en primer lloc, als meus directors, Germán i Damiàn, pel seu suport durant el desenvolupament de la tesi i per la confiança que han dipositat en mi durant aquests anys. Al seu torn, agrair també a la resta de membres del grup GRyCAP per haver-me acompanyat i recolzat durant el transcurs d'aquesta tesi, especialment amb aquells amb qui he col·laborat directament. Espere que puguem continuar treballant conjuntament en el futur.

Seguint amb agraïments anteriors[5], de nou, agrair a la meua família per continuar, com han fet sempre, recolzant i respectant les meues decisions, i aconsellarme en aquestes. Una vegada més, vull agrair a Sandra estar sempre al meu costat, per dur a terme els nostres projectes, tant personals com professionals.

Per fi toca donar fi a aquest capítol.

[5]Tesi doctoral, Mètodes de Monte Carlo avançats en dosimetria, Vicent Giménez Alventosa, 2021

Bibliografia

[1] W.-c. Feng, K. Cameron, The green500 list: Encouraging sustainable supercomputing, Computer 40 (12) (2007) 50–55. doi:10.1109/MC.2007.445.

[2] R. Machuga, Factors determining the use of cloud computing in enterprise management in the eu (considering the type of economic activity), Problems and Perspectives in Management 18(3) (2020) 93–105. doi:doi:10.21511/ppm.18(3).2020.08.

[3] C. Kesselman, I. Foster, Grid: Blueprint for a New Computing Infrastructure (Elsevier Series in Grid Computing), Morgan Kaufmann Publishers, 2004.

[4] OpenNebula, https://opennebula.org/, accessed: 2021-05-10.

[5] A. Shrivastwa, S. Sarat, K. Jackson, C. Bunch, E. Sigler, T. Campbell, OpenStack: Building a Cloud Environment, Packt Publishing, 2016.

[6] Monya Baker, Is there a reproducibility crisis? a nature survey lifts the lid on how researchers view the 'crisis' rocking science and what they think will help, Nature (2016).

[7] L. P. Freedman, I. M. Cockburn, T. S. Simcoe, The economics of reproducibility in preclinical research, PLOS Biology 13 (6) (2015) 1–9. doi:10.1371/journal.pbio.1002165.
URL https://doi.org/10.1371/journal.pbio.1002165

[8] EUROPEAN COMMISSION, Open Innovation Open Science Open to the World, European Commission, 2016.

[9] Goals of research and innovation policy. European Commission, https://ec.europa.eu/info/research-and-innovation/strategy/goals-research-and-innovation-policy, accessed: 2021-06-01.

[10] European Open Science Cloud (EOSC). European Commission, https://ec.europa.eu/research/openscience/index.cfm?pg=open-science-cloud, accessed: 2021-06-01.

[11] BITSS, https://www.bitss.org/, accessed: 2021-06-01.

[12] Public Library of Science, https://www.plos.org/, accessed: 2021-06-01.

[13] COS, https://cos.io/, accessed: 2021-06-01.

[14] V. Giménez-Alventosa, V. Giménez, F. Ballester, J. Vijande, P. Andreo, Monte carlo calculation of beam quality correction factors for PTW cylindrical ionization chambers in photon beams, Physics in Medicine & Biology 65 (20) (2020) 205005. doi:10.1088/1361-6560/ab9501.
URL https://doi.org/10.1088/1361-6560/ab9501

[15] Apache Spark, https://spark.apache.org/, accessed: 2021-06-16.

[16] Perfect, https://www.prefect.io/, accessed: 2021-09-23.

[17] Conductor, https://netflix.github.io/conductor/, accessed: 2021-09-23.

[18] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus: a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35, funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575. doi:10.1016/j.future.2014.10.008.
URL http://pegasus.isi.edu/publications/2014/2014-fgcs-deelman.pdf

[19] Makeflow, http://ccl.cse.nd.edu/software/makeflow/, accessed: 2021-09-23.

[20] Airavata, http://airavata.apache.org/, accessed: 2021-09-23.

[21] J. Yu, R. Buyya, A novel architecture for realizing grid workflow using tuple spaces, in: Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 119–128. doi:10.1109/GRID.2004.3.

[22] J. Shi, J. Lu, Performance models of data parallel dag workflows for large scale data analytics, in: 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW), 2021, pp. 104–111. doi:10.1109/ICDEW53142.2021.00026.

[23] P. Wangsom, K. Lavangnananda, P. Bouvry, Multi-objective scientific-workflow scheduling with data movement awareness in cloud, IEEE Access 7 (2019) 177063–177081. doi:10.1109/ACCESS.2019.2957998.

[24] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, Y. Cheng, Wukong: A scalable and locality-enhanced framework for serverless parallel computing, in: Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1–15. doi:10.1145/3419111.3421286.
URL https://doi.org/10.1145/3419111.3421286

[25] Apache Hadoop, https://hadoop.apache.org/, accessed: 2021-06-16.

[26] MongoDB, https://www.mongodb.com/, accessed: 2021-06-16.

[27] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.
URL https://doi.org/10.1145/1327452.1327492

[28] J. D. Ullman, Designing good mapreduce algorithms, XRDS 19 (1) (2012) 30–34. doi:10.1145/2331042.2331053.
URL https://doi.org/10.1145/2331042.2331053

[29] NIST - Computació en el Núvol, https://csrc.nist.gov/publications/detail/sp/800-145/final, accessed: 2021-06-09.

[30] M. A. Khan, A. Paplinski, A. M. Khan, M. Murshed, R. Buyya, Dynamic virtual machine consolidation algorithms for energy-efficient cloud resource management: a review, Sustainable cloud and energy services (2018) 135–165.

[31] J. Spillner, C. Mateos, D. A. Monge, Faaster, better, cheaper: The prospect of serverless scientific computing and hpc, in: E. Mocskos, S. Nesmachnow (Eds.), High Performance Computing, Springer International Publishing, Cham, 2018, pp. 154–168.

[32] Q. Jiang, Y. C. Lee, A. Y. Zomaya, Serverless execution of scientific workflows, in: M. Maximilien, A. Vallecillo, J. Wang, M. Oriol (Eds.), Service-Oriented Computing, Springer International Publishing, Cham, 2017, pp. 706–721.

[33] M. Pawlik, P. Banach, M. Malawski, Adaptation of workflow application scheduling algorithm to serverless infrastructure, in: U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, S. L. Scott (Eds.), Euro-Par 2019: Parallel Processing Workshops, Springer International Publishing, Cham, 2020, pp. 345–356.

[34] Limits AWS Lambda, https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, accessed: 2021-05-20.

[35] Azure Functions, https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale, accessed: 2021-05-20.

[36] L. A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37. doi:10.1109/MC.2007.443.

[37] X. Fan, W.-D. Weber, L. A. Barroso, Power provisioning for a warehouse-sized computer, SIGARCH Comput. Archit. News 35 (2) (2007) 13–23. doi:10.1145/1273440.1250665.
URL https://doi.org/10.1145/1273440.1250665

[38] C. Lefurgy, X. Wang, M. Ware, Server-level power control, in: Fourth International Conference on Autonomic Computing (ICAC'07), 2007, pp. 4–4. doi:10.1109/ICAC.2007.35.

[39] J. Ericson, M. Mohammadian, F. Santana, Analysis of performance variability in public cloud computing, in: 2017 IEEE International Conference on Information Reuse and Integration, 2017, pp. 308–314.

[40] P. Wayner, Benchmarking amazon ec2: The wacky world of cloud performance (2013).
URL http://www.infoworld.com/article/2613784/cloud-computing/benchmarking-amazon-ec2--the-wacky-world-of-cloud-performance.html

[41] N. T. Hieu, M. D. Francesco, A. Ylä-Jääski, Virtual machine consolidation with multiple usage prediction for energy-efficient cloud data centers, IEEE Transactions on Services Computing 13 (1) (2020) 186–199. doi:10.1109/TSC.2017.2648791.

[42] J. Schad, J. Dittrich, J.-A. Quiané-Ruiz, Runtime measurements in the cloud: Observing, analyzing, and reducing variance, Proc. VLDB Endow. 3 (1–2) (2010) 460–471.

[43] A. Iosup, N. Yigitbasi, D. Epema, On the performance variability of production cloud services, in: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2011, pp. 104–113.

[44] Amazon Web Services, https://aws.amazon.com/, accessed: 2021-05-17.

[45] Google Cloud, https://cloud.google.com/, accessed: 2021-05-17.

[46] P. Leitner, J. Cito, Patterns in the chaos-a study of performance variation and predictability in public iaas clouds, ACM Trans. Internet Technol. 16 (3) (2016).

[47] S. Shankar, J. M. Acken, N. K. Sehgal, Measuring performance variability in the clouds, IETE Technical Review 35 (6) (2018) 656–660.

[48] N. E. Jerger, D. Vantrease, M. Lipasti, An evaluation of server consolidation workloads for multi-core designs, in: 2007 IEEE 10th International Symposium on Workload Characterization, 2007, pp. 47–56. doi:10.1109/IISWC.2007.4362180.

[49] T. Dey, W. Wang, J. W. Davidson, M. L. Soffa, Characterizing multi-threaded applications based on shared-resource contention, in: (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, 2011, pp. 76–86. doi:10.1109/ISPASS.2011.5762717.

[50] P. Apparao, R. Iyer, X. Zhang, D. Newell, T. Adelmeyer, Characterization & analysis of a server consolidation benchmark, in: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 21–30. doi:10.1145/1346256.1346260.
URL https://doi.org/10.1145/1346256.1346260

[51] I. Foster, C. Kesselman, Globus: a metacomputing infrastructure toolkit, The International Journal of Supercomputer Applications and High Performance Computing 11 (2) (1997) 115–128. arXiv:https://doi.org/10.1177/109434209701100205, doi:10.1177/109434209701100205.
URL https://doi.org/10.1177/109434209701100205

[52] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience., Concurrency - Practice and Experience 17 (2-4) (2005) 323–356.

[53] HTCondor, https://research.cs.wisc.edu/htcondor/index.html, accessed: 2021-2-17.

[54] A. B. Yoo, M. A. Jette, M. Grondona, Slurm: Simple linux utility for resource management, in: D. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 44–60.

[55] G. Staples, Torque resource manager, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 8–es. doi:10.1145/1188455.1188464. URL https://doi.org/10.1145/1188455.1188464

[56] R. J. Keizer, M. van Benten, J. H. Beijnen, J. H. Schellens, A. D. Huitema, Piraña and pcluster: A modeling environment and cluster infrastructure for nonmem, Computer Methods and Programs in Biomedicine 101 (1) (2011) 72–79. doi:https://doi.org/10.1016/j.cmpb.2010.04.018. URL https://www.sciencedirect.com/science/article/pii/S0169260710001161

[57] D. Kliazovich, S. T. Arzo, F. Granelli, P. Bouvry, S. U. Khan, e-stab: Energy-efficient scheduling for cloud computing applications with traffic load balancing, in: 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, 2013, pp. 7–13. doi:10.1109/GreenCom-iThings-CPSCom.2013.28.

[58] C. Ghribi, M. Hadji, D. Zeghlache, Energy efficient vm scheduling for cloud data centers: Exact allocation and migration algorithms, in: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 671–678. doi:10.1109/CCGrid.2013.89.

[59] W. Li, J. Tordsson, E. Elmroth, Modeling for dynamic cloud scheduling via migration of virtual machines, in: 2011 IEEE Third International Conference on Cloud Computing Technology and Science, 2011, pp. 163–171. doi:10.1109/CloudCom.2011.31.

[60] V. Kherbache, E. Madelaine, F. Hermenier, Scheduling live migration of virtual machines, IEEE Transactions on Cloud Computing 8 (1) (2020) 282–296. doi:10.1109/TCC.2017.2754279.

[61] H. Lu, C. Xu, C. Cheng, R. Kompella, D. Xu, vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications, in: 2015 IEEE 8th International Conference on Cloud Computing, 2015, pp. 453–460. doi:10.1109/CLOUD.2015.67.

[62] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, N. J. Wright, Performance analysis of high performance computing applications on the amazon web services cloud, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010, pp. 159–168. doi:10.1109/CloudCom.2010.69.

[63] J. Emeras, S. Varrette, V. Plugaru, P. Bouvry, Amazon elastic compute cloud (ec2) versus in-house hpc platform: A cost analysis, IEEE Transactions on Cloud Computing 7 (2) (2019) 456–468. doi:10.1109/TCC.2016.2628371.

[64] E. Roloff, M. Diener, L. P. Gaspary, P. O. A. Navaux, Hpc application performance and cost efficiency in the cloud, in: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2017, pp. 473–477. doi:10.1109/PDP.2017.59.

[65] Microsoft Azure, https://azure.microsoft.com/, accessed: 2021-05-17.

[66] A. Saad, A. El-Mahdy, Hpccloud seer: A performance model based predictor for parallel applications on the cloud, IEEE Access 8 (2020) 87978–87993. doi:10.1109/ACCESS.2020.2992880.

[67] C. de Alfonso, M. Caballer, F. Alvarruiz, G. Moltó, An economic and energy-aware analysis of the viability of outsourcing cluster computing to a cloud, Future Generation Computer Systems 29 (3) (2013) 704–712, special Section: Recent Developments in High Performance Computing and Security. doi:https://doi.org/10.1016/j.future.2012.08.014.
URL https://www.sciencedirect.com/science/article/pii/S0167739X12001720

[68] A. M. Maliszewski, E. Roloff, E. D. Carreño, D. Griebler, L. P. Gaspary, P. O. A. Navaux, Performance and cost-aware hpc in clouds: A network interconnection assessment, in: 2020 IEEE Symposium on Computers and Communications (ISCC), 2020, pp. 1–6. doi:10.1109/ISCC50000.2020.9219554.

[69] L. V. Kalé, M. Bhandarkar, R. Brunner, Load balancing in parallel molecular dynamics, in: A. Ferreira, J. Rolim, H. Simon, S.-H. Teng (Eds.), Solving Irregularly Structured Problems in Parallel, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 251–261.

[70] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, Y. Robert, A proposal for a heterogeneous cluster scalapack (dense linear solvers), IEEE Transactions on Computers 50 (10) (2001) 1052–1070. doi:10.1109/12.956091.

[71] R. R. Manumachu, A. Lastovetsky, P. Alonso, Heterogeneous pblas: Optimization of pblas for heterogeneous computational clusters, in: 2008 International Symposium on Parallel and Distributed Computing, 2008, pp. 73–80. doi:10.1109/ISPDC.2008.9.

[72] B. Pérez, J. L. Bosque, R. Beivide, Simplifying programming and load balancing of data parallel applications on heterogeneous systems, in: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, GPGPU '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 42–51. doi:10.1145/2884045.2884051.
URL https://doi.org/10.1145/2884045.2884051

[73] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran, M. Garzarán, Adaptive partitioning for irregular applications on heterogeneous cpu-gpu chips, Procedia Computer Science 51 (2015) 140–149, international Conference On Computational Science, ICCS 2015. doi:https://doi.org/10.1016/j.procs.2015.05.213.
URL https://www.sciencedirect.com/science/article/pii/S1877050915010212

[74] L. V. Kale, A. Bhatele, Parallel science and engineering applications: The Charm++ approach, CRC Press, 2019.

[75] M. Rodríguez-Gonzalo, D. E. Singh, J. G. Blas, J. Carretero, Improving the energy efficiency of mpi applications by means of malleability, in: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016, pp. 627–634. doi:10.1109/PDP.2016.98.

[76] M. Garcia-Gasulla, G. Houzeaux, R. Ferrer, A. Artigues, V. López, J. Labarta, M. Vázquez, Mpi+x: task-based parallelisation and dynamic load balance of finite element assembly, International Journal of Computational Fluid Dynamics 33 (3) (2019) 115–136. arXiv:https://doi.org/10.1080/10618562.2019.1617856, doi:10.1080/10618562.2019.1617856.
URL https://doi.org/10.1080/10618562.2019.1617856

[77] W. Li, Y. Xia, M. Zhou, X. Sun, Q. Zhu, Fluctuation-aware and predictive workflow scheduling in cost-effective infrastructure-as-a-service clouds, IEEE Access 6 (2018) 61488–61502. doi:10.1109/ACCESS.2018.2869827.

[78] C. Li, J. Tang, T. Ma, X. Yang, Y. Luo, Load balance based workflow job scheduling algorithm in distributed cloud, Journal of Network and Computer Applications 152 (2020) 102518. doi:https://doi.org/10.1016/j.jnca.2019.102518.
URL https://www.sciencedirect.com/science/article/pii/S1084804519303789

[79] T. Hothorn, F. Leisch, Case studies in reproducibility, Briefings in Bioinformatics 12 (3) (2011) 288–300. arXiv:https://academic.oup.com/bib/

article-pdf/12/3/288/717452/bbq084.pdf, doi:10.1093/bib/bbq084.
URL https://doi.org/10.1093/bib/bbq084

[80] C. Boettiger, An introduction to docker for reproducible research, SIGOPS Oper. Syst. Rev. 49 (1) (2015) 71–79. doi:10.1145/2723872.2723882.
URL https://doi.org/10.1145/2723872.2723882

[81] runmycode.org, http://www.runmycode.org/, accessed: 2021-05-24.

[82] Codeocean, https://codeocean.com/, accessed: 2021-05-24.

[83] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, Jupyter Development Team, Jupyter notebooks - a publishing format for reproducible computational workflows, Concurrency and Computation (2016).

[84] B. K. Beaulieu-Jones, C. S. Greene, Reproducibility of computational workflows is automated using continuous analysis, Nature Biotechnology 35 (4) (2017) 342–346. doi:10.1038/nbt.3780.
URL https://doi.org/10.1038/nbt.3780

[85] T. Šimko, L. Heinrich, H. Hirvonsalo, D. Kousidis, D. Rodríguez, Reana: A system for reusable research data analyses, in: EPJ web of conferences, Vol. 214, EDP Sciences, 2019, p. 06034.

[86] G. M. Kurtzer, V. Sochat, M. W. Bauer, Singularity: Scientific containers for mobility of compute, PloS one 12 (5) (2017) e0177459.

[87] M. Mohaan, R. Raithatha, Learning Ansible, Packt Publishing Ltd, 2014.

[88] RADL, https://imdocs.readthedocs.io/en/latest/radl.html, accessed: 2021-05-24.

[89] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso, Dynamic management of virtual infrastructures, J. Grid Comput 13, no 1 (2015) 53–70. doi:10.1007/s10723-014-9296-5.

[90] Terraform, https://www.terraform.io/, accessed: 2021-2-18.

[91] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, et al., Jupyter Notebooks-a publishing format for reproducible computational workflows., Vol. 2016, 2016.

[92] J. F. Pimentel, L. Murta, V. Braganholo, J. Freire, A large-scale study about quality and reproducibility of jupyter notebooks, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 507–517. doi:10.1109/MSR.2019.00077.

[93] H. Nguyen, D. A. Case, A. S. Rose, NGLview–interactive molecular graphics for Jupyter notebooks, Bioinformatics 34 (7) (2017) 1241–1242. arXiv:https://academic.oup.com/bioinformatics/article-pdf/34/7/1241/25119547/btx789.pdf, doi:10.1093/bioinformatics/btx789.
URL https://doi.org/10.1093/bioinformatics/btx789

[94] J. Reades, Teaching on jupyter, REGION 7 (1) (2020) 21–34. doi:10.18335/region.v7i1.282.
URL https://openjournals.wu-wien.ac.at/ojs/index.php/region/article/view/282

[95] K. Mahajan, D. Figueiredo, V. Misra, D. Rubenstein, Optimal pricing for serverless computing, in: 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1–6. doi:10.1109/GLOBECOM38437.2019.9013156.

[96] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[97] HadoopTable, https://hadoopecosystemtable.github.io/, accessed: 2022-03-23.

[98] GCP, https://cloud.google.com/, accessed: 2022-03-23.

[99] EMR, https://aws.amazon.com/emr/, accessed: 2022-03-23.

[100] HDInsight, https://azure.microsoft.com/en-us/services/hdinsight/, accessed: 2022-03-23.

[101] Cloud Dataproc, https://cloud.google.com/dataproc/, accessed: 2022-03-23.

[102] OpenStack Sahara, https://docs.openstack.org/sahara/latest/, accessed: 2022-03-23.

[103] OpenStack, https://www.openstack.org/, accessed: 2022-03-23.

[104] Alex Glikson, Stefan Nastic, Schahram Dustdar, Deviceless edge computing: extending serverless computing to the edge of the network, Conference: the 10th ACM International Systems and Storage Conference (2017).

[105] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, Occupy the cloud: Distributed computing for the 99%, Proceedings of the 2017 Symposium on Cloud Computing (2017) 445–451.

[106] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian, Building a chatbot with serverless computing, MOTA '16 Proceedings of the 1st International Workshop on Mashups of Things and APIs (2016).

[107] AWS Lambda, `https://aws.amazon.com/lambda/`, accessed: 2022-03-23.

[108] WoSC, `https://www.serverlesscomputing.org/wosc17/`, accessed: 2022-03-23.

[109] Lambda MapReduce, `https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/`, accessed: 2022-03-23.

[110] PyWren, `https://github.com/pywren/pywren`, accessed: 2022-03-23.

[111] Ooso, `https://github.com/d2si-oss/ooso`, accessed: 2022-03-23.

[112] Corral, `https://github.com/bcongdon/corral`, accessed: 2022-03-23.

[113] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, Maciej Malawski, Performance evaluation of heterogeneous cloud functions, Concurrency and Computation (2018). `doi:https://doi.org/10.1002/cpe.4792`.

[114] Hyungro Lee, Kumar Satyam, Geoffrey Fox, Evaluation of production serverless computing environments, 2018 IEEE 11th International Conference on Cloud Computing (CLOUD) (2018). `doi:doi:10.1109/CLOUD.2018.00062`.

[115] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara, Serverless computing: An investigation of factors influencing microservice performance, in: Cloud Engineering (IC2E), 2018 IEEE International Conference on, IEEE, 2018, pp. 159–169.

[116] S3 Scalability, `https://aws.amazon.com/blogs/aws/amazon-s3-performance-tips-tricks-seattle-hiring-event/`, accessed: 2022-03-23.

[117] Amplab benchmark, `https://amplab.cs.berkeley.edu/benchmark/`, accessed: 2022-03-23.

[118] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Michael Stonebraker, A comparison of approaches to large-scale data analysis, SIGMOD 2009 (2009).

[119] Garfinkel, Simson L., An evaluation of amazon's grid computing services: Ec2, s3, and sqs, Harvard Computer Science Group Technical Report TR-08-07 (2007).

[120] Laing Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, Michael Swift., Peeking behind the curtains of serverless platforms, Annual Technical Conference (2018).

[121] Configuring Lambda functions, https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, accessed: 2022-03-23.

[122] Amazon S3 request rate, https://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html, accessed: 2022-03-23.

[123] S3 Multipart Upload, https://docs.aws.amazon.com/AmazonS3/latest/dev/qfacts.html, accessed: 2022-03-23.

[124] Calatrava A., Romero E., Caballer M., Moltó G. and Alonso J.M., Self-managed cost-efficient virtual elastic clusters on hybrid cloud infrastructures, Future Generation Computer Systems (2016). doi:doi:10.1016/j.future.2016.01.018.

[125] Carlos de Alfonso Laguna, Miguel Caballer and Vicente Hernández, Efficient power management in high performance computer clusters, International Conference on Green Computing 2010 (ICGreen 2010) (2010).

[126] OneData, https://onedata.org/#/home, accessed: 2022-03-23.

[127] EGI DataHub, https://www.egi.eu/services/datahub/, accessed: 2021-09-23.

[128] Donoho, D.L., Maleki, A., Rahman, I.U., Shahram, M., Stodden, V., Reproducible research in computational harmonic analysis, Comput. Sci Eng. (2009).

[129] Freedman, L.P., Cockburn, I.M., Simcoe, T.S., The economics of reproducibility in preclinical research, PLoS Biol. (2015).

[130] Chillarón, Mónica., Vidal, Vicent, Verdú, Gumersindo, Ct image reconstruction withsuitesparseqr factorization package, Radiation Physics and Chemistry (2019). doi:doi:https://doi.org/10.1016/j.radphyschem.2019.04.039.

[131] Andrew J. Reader, Stijn Ally, Filippos Bakatselos, Roido Manavaki, Richard J. Walledge, Alan P. Jeavons, Peter J. Julyan, Sha Zhao, David L. Hastings, and Jamal Zweit, One-pass list-mode em algorithm for high-resolution 3-d pet image reconstruction into large arrays, IEEE TRANSACTIONS ON NUCLEAR SCIENCE (2002).

[132] V. Giménez-Alventosa, P. C. G. Antunes, J. Vijande, F. Ballester, J. Pérez-Calatayud, P. Andreo, Collision-kerma conversion between dose-to-tissue and dose-to-water by photon energy-fluence corrections in low-energy brachytherapy, Physics in Medicine and Biology 62 (1) (2016) 146–164. doi:10.1088/1361-6560/aa4f6a.
URL https://doi.org/10.1088%2F1361-6560%2Faa4f6a

[133] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al., The fair guiding principles for scientific data management and stewardship, Scientific data 3 (2016).

[134] IPOL, https://www.ipol.im/, accessed: 2022-03-23.

[135] Daniel Nüst, Markus Konkol, Marc Schutzeichel, Edzer Pebesma, Christian Kray, Holger Przibytzin, Jörg Lorenz, Opening the publication process with executable research compendia, D-Lib Magazine (2017).

[136] Galaxy, https://galaxyproject.org, accessed: 2022-03-23.

[137] Open Science Framework, https://osf.io, accessed: 2022-03-23.

[138] K. Wolstencroft, S. Owen, O. Krebs, Q. Nguyen, N. J. Stanford, M. Golebiewski, A. Weidemann, M. Bittkowski, L. An, D. Shockley, J. L. Snoep, W. Mueller, C. Goble, SEEK: A systems biology data and model management platform, BMC Systems Biology 9 (1) (2015) 33. doi:10.1186/s12918-015-0174-y.
URL http://www.biomedcentral.com/1752-0509/9/33

[139] REANA, http://www.reanahub.io/.

[140] Stencila, https://stenci.la, accessed: 2022-03-23.

[141] C. de Alfonso, M. Caballer, A. Calatrava, G. Moltó, I. Blanquer, Multi-elastic Datacenters: Auto-scaled Virtual Clusters on Energy-Aware Physical Infrastructures, Journal of Grid Computing 17 (1) (2019) 191–204. doi:10.1007/s10723-018-9449-z.
URL http://link.springer.com/10.1007/s10723-018-9449-z

[142] EOSC portal, https://marketplace.eosc-portal.eu/services/.

[143] Ansible, https://www.ansible.com, accessed: 2022-03-23.

[144] P. Rawla, Epidemiology of prostate cancer, World Journal of Oncology 10 (2) (2019).
URL https://www.wjon.org/index.php/wjon/article/view/1191

[145] Wu X, Reinikainen P, Kapanen M, Vierikko T, Ryymin P, Kellokumpu-Lehtinen PL, Dynamic contrast-enhanced imaging as a prognostic tool in early diagnosis of prostate cancer: Correlation with psa and clinical stage, Contrast Media Mol Imaging (2018). doi:doi:10.1155/2018/3181258.

[146] F. Bratan, E. Niaf, C. Melodelima, A. L. Chesnais, R. Souchon, F. Mège-Lechevallier, M. Colombel, O. Rouvière, Influence of imaging and histological factors on prostate cancer detection and localisation on multiparametric mri: a prospective study, European Radiology 23 (7) (2013) 2019–2029. doi:10.1007/s00330-013-2795-0.
URL https://doi.org/10.1007/s00330-013-2795-0

[147] J. D. Le, N. Tan, E. Shkolyar, D. Y. Lu, L. Kwan, L. S. Marks, J. Huang, D. J. Margolis, S. S. Raman, R. E. Reiter, Multifocality and prostate cancer detection by multiparametric magnetic resonance imaging: Correlation with whole-mount histopathology, European Urology 67 (3) (2015) 569 – 576. doi:https://doi.org/10.1016/j.eururo.2014.08.079.
URL http://www.sciencedirect.com/science/article/pii/S0302283814008914

[148] S. S. KETY, The theory and applications of the exchange of inert gas at the lungs and tissues, Pharmacological Reviews 3 (1) (1951) 1–41. arXiv:http://pharmrev.aspetjournals.org/content/3/1/1.full.pdf.
URL http://pharmrev.aspetjournals.org/content/3/1/1

[149] B. G. Tofts PS, Wicks DA, The mri measurement of nmr and physiological parameters in tissue to study disease process, Prog Clin Biol Res (1991).

[150] Gunnar Brix;Wolfhard Semmler;Rüdiger Port;Lothar Schad;Günter Layer;Walter Lorenz, Pharmacokinetic parameters in cns gd-dtpa enhanced mr imaging, Journal of Computer Assisted Tomography (1991).

[151] H. B. W. Larsson, M. Stubgaard, J. L. Frederiksen, M. Jensen, O. Henriksen, O. B. Paulson, Quantitation of blood-brain barrier defect by magnetic resonance imaging and gadolinium-dtpa in patients with multiple sclerosis and brain tumors, Magnetic Resonance in Medicine 16 (1) (1990)

117–131. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.1910160111, doi:10.1002/mrm.1910160111.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.1910160111

[152] P. S. Tofts, A. G. Kermode, Measurement of the blood-brain barrier permeability and leakage space using dynamic mr imaging. 1. fundamental concepts, Magnetic Resonance in Medicine 17 (2) (1991) 357–367. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.1910170208, doi:10.1002/mrm.1910170208.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.1910170208

[153] K. M. Donahue, R. M. Weisskoff, D. Burstein, Water diffusion and exchange as they influence contrast enhancement, Journal of Magnetic Resonance Imaging 7 (1) (1997) 102–110. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jmri.1880070114, doi:10.1002/jmri.1880070114.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jmri.1880070114

[154] D. Flouri, D. Lesnic, S. P. Sourbron, Fitting the two-compartment model in dce-mri by linear inversion, Magnetic Resonance in Medicine 76 (3) (2016) 998–1006. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.25991, doi:10.1002/mrm.25991.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.25991

[155] R. Brun, F. Rademakers, Root — an object oriented data analysis framework, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 389 (1) (1997) 81 – 86, new Computing Techniques in Physics Research V. doi:https://doi.org/10.1016/S0168-9002(97)00048-X.
URL http://www.sciencedirect.com/science/article/pii/S016890029700048X

[156] Xuan Liu, Claude Comtat, Christian Michel, Paul Kinahan, Michel Defrise, and David Townsend, Comparison of 3-d reconstruction with 3d-osem and with fore + osem for pet, IEEE TRANSACTIONS ON MEDICAL IMAGING (2001).

[157] Sarabjeet Singh, Mannudeep K. Kalra, Jiang Hsieh, Paul E. Licato, Synho Do, Homer H. Pien, Michael A. Blake, Abdominal ct: Comparison of adapti-

ve statistical iterative and filtered back projection reconstruction techniques, Radiology (2010).

[158] L.A. Sheep and Y.Vardi, Maximum likelihood reconstruction for emission tomography, IEEE TRANSACTIONS ON MEDICAL IMAGING (1982).

[159] Jin Mo Goo, Trongtum Tongdee, Ranista Tongdee, Kwangjae Yeo, Charles F. Hildebolt, Kyongtae T. Bae, Volumetric measurement of synthetic lung nodules with multi–detector row ct: Effect of various image reconstruction parameters and segmentation thresholds on measurement accuracy, Radiology (2005). doi:https://doi.org/10.1148/radiol.2353040737.

[160] James G. Ravenel, William M. Leue, Paul J. Nietert, James V. Miller, Katherine K. Taylor, Gerard A. Silvestri, Pulmonary nodule volume: Effects of reconstruction parameters on automated measurements—a phantom study, Radiology (2008). doi:https://doi.org/10.1148/radiol.2472070868.

[161] Yue-Houng Hu, Bo Zhao, Wei Zhao, Image artifacts in digital breast tomosynthesis: Investigation of the effects of system geometry and reconstruction parameters using a linear system approach, Medical Physics (2008). doi:https://doi.org/10.1118/1.2996110.

[162] Maria Lyra, Agapi Ploussi, Filtering in spect image reconstruction, Journal of Biomedical Imaging (2011).

[163] Salvat F., Penelope. a code system for monte carlo simulation of electron and photon transport, Issy-Les-Moulineaux: OECD Nuclear Energy Agengy (2014).

[164] L. F. Bittencourt, E. R. M. Madeira, HCOC: a cost optimization algorithm for workflow scheduling in hybrid clouds, Journal of Internet Services and Applications 2 (3) (2011) 207–227.

[165] L. Beaulieu, Å. Carlsson Tedgren, J.-F. Carrier, S. D. Davis, F. Mourtada, M. J. Rivard, R. M. Thomson, F. Verhaegen, T. A. Wareing, J. F. Williamson, Report of the Task Group 186 on model-based dose calculation methods in brachytherapy beyond the TG-43 formalism: Current status and recommendations for clinical implementation, Medical Physics 39 (10) (2012) 6208–6236. doi:10.1118/1.4747264.

[166] P. Andreo et al., Absorbed Dose Determination in External Beam Radiotherapy: an International Code of Practice for Dosimetry Based on Standards of Absorbed Dose to Water , IAEA Technical Reports Series 398, https://www-pub.iaea.org/MTCD/publications/PDF/TRS398_scr.pdf (2000).

[167] Andreo, P., Burns, D. T., Kapsch, R.-P., McEwen, M. and Vatnitsky, S, Status of the update of the IAEA TRS-398 Code of Practice, in: Book of Extended Synopses, International Symposium "Standards, Applications and QA in Medical Radiation Dosimetry (IDOS 2019)", International Atomic Energy Agency, Vienna, 2019, pp. 71–76, available at `https://www.iaea.org/events/idos2019`.

[168] A. Abedi, T. Brecht, Conducting repeatable experiments in highly variable cloud computing environments, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 287–292.

[169] AWS single-tenant, `https://aws.amazon.com/ec2/pricing/dedicated-instances/`, accessed: 2022-03-23.

[170] A. Gupta, D. Milojicic, Evaluation of hpc applications on cloud, in: 2011 Sixth Open Cirrus Summit, 2011, pp. 22–26.

[171] A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B. Lee, P. Faraboschi, R. Kaufmann, D. Milojicic, The who, what, why, and how of high performance computing in the cloud, in: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, Vol. 1, 2013, pp. 306–314.

[172] O. Sarood, A. Gupta, L. V. Kalé, Cloud friendly load balancing for hpc applications: Preliminary work, in: 2012 41st International Conference on Parallel Processing Workshops, 2012, pp. 200–205.

[173] A. Gupta, O. Sarood, L. V. Kale, D. Milojicic, Improving hpc application performance in cloud through dynamic load balancing, in: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 402–409.

[174] F. Xu, F. Liu, H. Jin, Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud, IEEE Transactions on Computers 65 (8) (2016) 2470–2483.

[175] P. Fan, Z. Chen, J. Wang, Z. Zheng, M. R. Lyu, Topology-aware deployment of scientific applications in cloud computing, in: 2012 IEEE Fifth International Conference on Cloud Computing, 2012, pp. 319–326.

[176] M. Cierniak, M. J. Zaki, W. Li, Compile-Time Scheduling Algorithms for a Heterogeneous Network of Workstations, The Computer Journal 40 (6) (1997) 356–372.

[177] V. Giménez-Alventosa, V. G. Gómez, S. O. Gil, Penred: An extensible and parallel monte-carlo framework for radiation transport based on penelope (2020). arXiv:2003.00796.

[178] C. Valdes-Cortez, F. Ballester, J. Vijande, V. Gimenez, V. Gimenez-Alventosa, J. Perez-Calatayud, Y. Niatsetski, P. Andreo, Depth-dose measurement corrections for the surface electronic brachytherapy beams of an esteya® unit: a monte carlo study, Physics in Medicine & Biology 65 (24) (2020) 245026. doi:10.1088/1361-6560/ab9773.
URL https://doi.org/10.1088/1361-6560/ab9773

[179] V. Giménez-Alventosa, G. Moltó, M. Caballer, A framework and a performance assessment for serverless mapreduce on aws lambda, Future Generation Computer Systems 97 (2019) 259 – 274. doi:https://doi.org/10.1016/j.future.2019.02.057.

[180] V. G. Alventosa, G. M. Martínez, J. D. S. Quilis, Ruper-lb: Load balancing embarrasingly parallel applications in unpredictable cloud environments (2020). arXiv:2005.06361.

[181] Serverless framework, https://www.serverless.com/, accessed: 2021-1-25.

[182] Google Cloud Functions limits, https://cloud.google.com/functions/quotas, accessed: 2021-2-18.

[183] M. Taufer, A. Rosenberg, Scheduling DAG-based workflows on single cloud instances: High-performance and cost effectiveness with a static scheduler, International Journal of High Performance Computing Applications 31 (07 2015). doi:10.1177/1094342015594518.

[184] Haijun Cao, Hai Jin, Xiaoxin Wu, Song Wu, Xuanhua Shi, Dagmap: Efficient scheduling for dag grid workflow job, in: 2008 9th IEEE/ACM International Conference on Grid Computing, 2008, pp. 17–24. doi:10.1109/GRID.2008.4662778.

[185] F. Salvat, PENELOPE-2018: A code System for Monte Carlo Simulation of Electron and Photon Transport, OECD/NEA Data Bank, Issy-les-Moulineaux, France, 2019, available from http://www.nea.fr/lists/penelope.html.

[186] GEANT4 Collaboration (Agostinelli. S. et al.), Geant4—a simulation toolkit, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 506 (3) (2003) 250–303. doi:10.1016/S0168-9002(03)01368-8.

[187] A. Ferrari, P. R. Sala, A. Fassò, J. Ranft, FLUKA: A multi-particle transport code (program version 2005), CERN Yellow Reports: Monographs, CERN, Geneva, 2005. doi:10.5170/CERN-2005-010.

[188] I. Kawrakow, E. Mainegra-Hing, D. W. O. Rogers, F. Tessier, B. R. B. Walters, The EGSnrc Code System: Monte Carlo Simulation of Electron and Photon Transport, NRCC Report PIRS-701, National Research Council Canada, Ottawa, CAN, 2019.

[189] Los Alamos Scientific Laboratory. Group X-6., MCNP : a General Monte Carlo Code for Neutron and Photon Transport. Los Alamos, N.M., Dept. of Energy, Los Alamos Scientific Laboratory (1979). doi:10.1007/BFb0049033.

[190] J. Sempau, A. Badal, L. Brualla, A PENELOPE-based system for the automated Monte Carlo simulation of clinacs and voxelized geometries–application to far-from-axis fields, Med. Phys. 38 (2011) 5887 – 5895. doi:10.1118/1.3643029.

[191] V. Giménez-Alventosa, V. Giménez Gómez, S. Oliver, Penred: An extensible and parallel monte-carlo framework for radiation transport based on penelope, Computer Physics Communications (2021) 108065doi:https://doi.org/10.1016/j.cpc.2021.108065.
URL https://www.sciencedirect.com/science/article/pii/S0010465521001776

[192] S. Jan, D. Benoit, E. Becheva, T. Carlier, F. Cassol, P. Descourt, T. Frisson, L. Grevillot, L. Guigues, L. Maigne, C. Morel, Y. Perrot, N. Rehfeld, D. Sarrut, D. R. Schaart, S. Stute, U. Pietrzyk, D. Visvikis, N. Zahra, I. Buvat, GATE V6: a major enhancement of the GATE simulation platform enabling modelling of CT and radiotherapy, Physics in Medicine and Biology 56 (4) (2011) 881–901. doi:10.1088/0031-9155/56/4/001.

[193] Amazon Simple Storage Service (Amazon S3), https://aws.amazon.com/s3/, accessed: 2021-2-18.

[194] Amazon DynamoDB, https://aws.amazon.com/dynamodb/, accessed: 2021-2-18.

[195] AWS Lambda, https://aws.amazon.com/lambda/, accessed: 2021-2-18.

[196] Amazon API Gateway, https://aws.amazon.com/api-gateway/, accessed: 2021-2-18.

[197] AWS Spot Instances, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html, accessed: 2021-3-15.