Esteban Mocskos
Sergio Nesmachnow (Eds.)

Communications in Computer and Information Science 796

High Performance Computing

4th Latin American Conference, CARLA 2017
Buenos Aires, Argentina, and
Colonia del Sacramento, Uruguay, September 20–22, 2017
Revised Selected Papers

Communications in Computer and Information Science 796

Commenced Publication in 2007
Founding and Former Series Editors:
Alfredo Cuzzocrea, Xiaoyong Du, Orhun Kara, Ting Liu, Dominik Ślęzak,
and Xiaokang Yang

Editorial Board

Simone Diniz Junqueira Barbosa
Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

Phoebe Chen
La Trobe University, Melbourne, Australia

Joaquim Filipe
Polytechnic Institute of Setúbal, Setúbal, Portugal

Igor Kotenko
St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, St. Petersburg, Russia

Krishna M. Sivalingam
Indian Institute of Technology Madras, Chennai, India

Takashi Washio
Osaka University, Osaka, Japan

Junsong Yuan
Nanyang Technological University, Singapore, Singapore

Lizhu Zhou
Tsinghua University, Beijing, China

More information about this series at http://www.springer.com/series/7899

Esteban Mocskos · Sergio Nesmachnow (Eds.)

High Performance Computing

4th Latin American Conference, CARLA 2017
Buenos Aires, Argentina, and
Colonia del Sacramento, Uruguay, September 20–22, 2017
Revised Selected Papers

Editors
Esteban Mocskos
CSC-CONICET and Universidad de Buenos Aires
Buenos Aires
Argentina

Sergio Nesmachnow
Universidad de la República
Montevideo
Uruguay

ISSN 1865-0929 ISSN 1865-0937 (electronic)
Communications in Computer and Information Science
ISBN 978-3-319-73352-4 ISBN 978-3-319-73353-1 (eBook)
https://doi.org/10.1007/978-3-319-73353-1

Library of Congress Control Number: 2017963753

© Springer International Publishing AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

High-performance computing (HPC) is a dynamic field that combines the use of innovative computing technologies and algorithms with advances in a broad range of scientific, technical, and industrial areas. Latin America shares the global enthusiasm embracing and pushing forward HPC. New challenges coming from the use of the computing capabilities of massive multicores, accelerators, cluster platforms, cloud federations, and the new perspectives of Internet of Things resources all help to promote the research and innovation in this area.

Building on the success of the previous editions of CARLA, the High-Performance Computing Latin America Conference (and former HPCLATAM and CLCAR Conferences), the tenth edition was organized in Buenos Aires and Colonia jointly by Universidad de Buenos Aires (Argentina) and Universidad de la República (Uruguay).

The main goal of the CARLA conference is to provide a forum fostering the growth of the HPC community in Latin America, through the exchange and dissemination of new ideas, techniques, and research in HPC. In 2017, CARLA featured invited talks from academia and industry, full-paper sessions presenting mature work, and new ideas in research and industrial applications including: distributed systems, parallel algorithms and concurrency; GPU and MIC computing; mobile, grid, and cloud computing; big data, data management, and visualization; scientific computing applications; architecture, infrastructure, and HPC data center; HPC computing education and outreach; industrial solutions. Satellite events co-located with CARLA 2017 included the meeting of the Cloud Computing for Smart-City Energy Management (CC-SEM STIC-AmSud) Project, Red SCALAC (Servicios de Computación Avanzada para Latinoamérica y el Caribe), and RICAP CYTED (Red Iberoamericana de Computación de Altas Prestaciones). More than 100 researchers, students, technicians, practitioners, and representatives of industry, technology, and state companies and organizations (from more than 20 countries in Latin America, Europe, Asia, and Oceania) attended the event.

This book introduces the top contributions presented at CARLA 2017, covering all the aforementioned topics. As organizers, we think the articles are valuable contributions to the development of HPC in Latin America.

December 2017

Esteban Mocskos
Sergio Nesmachnow

Organization

Program Chairs

Gregoire Danoy	University of Luxembourg, Luxembourg
Ricardo Medel	Ascentio Technologies S.A., Argentina
Esteban Meneses	Costa Rica National High Technology Center, Costa Rica
Esteban Mocskos	CSC-CONICET and Universidad de Buenos Aires, Argentina
Sergio Nesmachnow	Universidad de la República, Uruguay
Markus Rampp	Max Planck Computing and Data Facility, Germany
Carlos Sarraute	Grandata Labs, Argentina
Luiz Angelo Steffenel	Université de Reims Champagne-Ardenne, France
Mariano Vazquez	Barcelona Supercomputing Center, Spain

Program Committee

José Pedro Aguerre	Universidad de la República, Uruguay
Hartwig Anzt	Karlsruhe Institute of Technology, Germany
Carlos J. Barrios	Universidad Industrial de Santander, Colombia
Leonardo Bautista Gomez	Centro Nacional de Supercomputación, Spain
Carlos Bederián	CONICET, Argentina
Pascal Bouvry	University of Luxembourg, Luxembourg
Carlos Buil	Universidad Técnica Federico Santa María, Chile
Harold Castro	Universidad de los Andes, Colombia
Marcio Castro	Federal University of Santa Catarina (UFSC), Brazil
Maria Clicia Stelling de Castro	Universidade do Estado do Rio de Janeiro, Brazil
Gerson Cavalheiro	Universidade Federal de Pelotas, Brazil
Germán Ceballos	Uppsala University, Sweden
Andrea Charão	Universidad Federal Santa María, Brazil
Esteban Clua	Universidade Federal Fluminense, Brazil
Flavio Colavecchia	Centro Atómico Bariloche, Comisión Nacional de Energía Atómica, Argentina
Daniel Cordeiro	Universidade de São Paulo, Brazil
Carlos Couder-Castañeda	Instituto Politecnico Nacional, Mexico
Alvaro Coutinho	Federal University of Rio de Janeiro, Brazil
Adrián Cristal	Barcelona Supercomputing Centre, Spain
Gregoire Danoy	University of Luxembourg, Luxembourg
Alvaro de la Ossa	Universidad de Costa Rica, Costa Rica
Cristian Mateos Diaz	ISISTAN-CONICET, Universidad Nacional del Centro, Argentina

Gilberto Diaz	Universidad Industrial de Santander, Colombia
Mario Jose Diván	Universidad Nacional de La Pampa, Argentina
Bernabe Dorronsoro	Universidad de Cadiz, Spain
Ernesto Dufrechou	Universidad de la República, Uruguay
Nicolás Erdödy	Open Parallel, New Zealand
Eduardo Fernandez	Universidad de la República, Uruguay
Ezequiel Ferrero	Department of Physics and Center for Complexity and Biosystems, Italy
Alejandro Flores-Méndez	CINVESTAV, Mexico
Emilio Francesquini	University of Campinas, Brazil
Joao Gazolla	Universidade Federal Fluminense, Brazil
Veronica Gil-Costa	Universidad Nacional San Luis, Argentina
Isidoro Gitler	ABACUS-CINVESTAV, Mexico
Brice Goglin	Inria, France
Leo Gonzalez	Universidad Politecnica de Madrid, Spain
José Luis Gordillo	Universidad Nacional Autónoma de México, México
Jesus Cruz Guzman	Universidad Nacional Autónoma de México, México
Elisa Heymann	Universitat Autònoma de Barcelona, Spain
Javier Iparraguirre	Universidad Tecnológica Nacional, Argentina
Santiago Iturriaga	Universidad de la República, Uruguay
Salma Jalife	Corporación Universitaria para el Desarrollo de Internet A.C., Mexico
Roberto Leon	Universidad Andres Bello, Chile
Francisco Luna	Universidad de Málaga, Spain
Renzo Massobrio	Universidad de la República, Uruguay
Rafael Mayo-Garcia	Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas, Spain
Ricardo Medel	Ascentio Technologies S.A., Argentina
Esteban Meneses	Costa Rica National High Technology Center, Costa Rica
Renato Miceli	CENAI-CIMATEC, Brazil
Barton Miller	University of Wisconsin-Madison, USA
Esteban Mocskos	CSC-CONICET and Universidad de Buenos Aires, Argentina
Philippe Navaux	Universidade Federal de Rio Grande do Sul, Brazil
Sergio Nesmachnow	Universidad de la República, Uruguay
Carla Osthoff	National Laboratory for Scientific Computing, Brazil
Alejandro Otero	Universidad de Buenos Aires and CSC-CONICET, Argentina
Horacio Paggi	Universidad Politécnica de Madrid, Spain
Jairo Panetta	Instituto Tecnológico de Aeronáutica, Brazil
Claudio J. Paz	UTN FRC, Argentina
Martín Pedemonte	Universidad de la República, Uruguay
Tomas Perez-Acle	Universidad de Chile, Chile
Laercio Lima Pilla	Universidade Federal de Santa Catarina, Brazil
Javier Principe	Universidad Politecnica de Cataluña - CIMNE, Spain

Juan Manuel Ramírez	Universidad de Colima, Mexico
Markus Rampp	Max Planck Computing and Data Facility, Germany
Vinod Rebello	Universidade Federal Fluminense, Brazil
Genghis Ríos	Pontificia Universidad Católica del Perú, Perú
Pablo Rodríguez-Bocca	Universidad de la República, Uruguay
Isaac Rudomin	Barcelona Supercomputing Center, Spain
Afonso Sales	Pontificia Universidad Católica do Rio Grande do Sul, Brazil
Carlos Sarraute	Grandata Labs, Argentina
Lucas Mello Schnorr	Universidade Federal do Rio Grande do Sul, Brazil
Hermes Senger	Universidade Federal de São Carlos, Brazil
Alejandro Soba	Comisión Nacional de Energía Atómica, Argentina
Roberto Souto	Laboratório Nacional de Computação, Brazil
Luiz Angelo Steffenel	Université de Reims Champagne-Ardenne, France
Mario Storti	Universidad Nacional del Litoral and CIMEC-CONICET, Argentina
Claude Tadonki	MINES ParisTech, PSL, France
Gonzalo Tancredi	Universidad de la República, Uruguay
Andrei Tchernykh	Centro de Investigación Científica y de Educación Superior de Ensenada, Mexico
Jamal Toutouh	Universidad de Málaga, Spain
Tram Truong-Huu	National University of Singapore, Singapore
Manuel Ujaldón	Universidad de Málaga, Spain
Gabriel Usera	Universidad de la República, Uruguay
Mariano Vazquez	Barcelona Supercomputing Center, Spain
Jesus Verduzco	Instituto Tecnológico de Colima, México
Pablo Javier Vidal	Universidad de la Patagonia Austral, Argentina
Nicolás Wolovick	Universidad Nacional de Córdoba, Argentina
Jesús Xamán	Centro Nacional de Investigación y Desarrollo Tecnológico, México
Alejandro Zunino	ISISTAN-CONICET, Argentina

Contents

HPC Infrastructures and Datacenters

A Deep Learning Mapper (DLM) for Scheduling on Heterogeneous Systems . . . 3
Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Mario Nemirovsky, Osman Unsal, Adrian Cristal, and Mateo Valero

Power Consumption Characterization of Synthetic Benchmarks in Multicores . . . 21
Jonathan Muraña, Sergio Nesmachnow, Santiago Iturriaga, and Andrei Tchernykh

Initial Experiences from TUPAC Supercomputer . . . 38
David Vinazza, Alejandro Otero, Alejandro Soba, and Esteban Mocskos

HPC Industry and Education

romeoLAB: A High Performance Training Platform for HPC, GPU and DeepLearning . . . 55
Arnaud Renard, Jean-Matthieu Etancelin, and Michael Krajecki

GPU, Multicores, Accelerators

Analysis and Characterization of GPU Benchmarks for Kernel Concurrency Efficiency . . . 71
Pablo Carvalho, Lúcia M. A. Drummond, Cristiana Bentes, Esteban Clua, Edson Cataldo, and Leandro A. J. Marzulo

Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA . . . 87
Habib Daneshpajouh, Pierre Delisle, Jean-Charles Boisson, Michael Krajecki, and Nordin Zakaria

Performance Prediction of Acoustic Wave Numerical Kernel on Intel Xeon Phi Processor . . . 101
Víctor Martínez, Matheus Serpa, Fabrice Dupros, Edson L. Padoin, and Philippe Navaux

Evaluating the NVIDIA Tegra Processor as a Low-Power Alternative for Sparse GPU Computations 111
José I. Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí

HPC Applications and Tools

Benchmarking Performance: Influence of Task Location on Cluster Throughput 125
Manuel Rodríguez-Pascual, José Antonio Moríñigo, and Rafael Mayo-García

PRIMULA: A Framework Based on Finite Elements to Address Multi Scale and Multi Physics Problems 139
Alejandro Soba

FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC 154
Josef Spillner, Cristian Mateos, and David A. Monge

AccaSim: An HPC Simulator for Workload Management 169
Cristian Galleguillos, Zeynep Kiziltan, and Alessio Netti

SherlockFog: Finding Opportunities for MPI Applications in Fog and Edge Computing 185
Maximiliano Geier and Esteban Mocskos

Big Data and Data Management

IoT Workload Distribution Impact Between Edge and Cloud Computing in a Smart Grid Application 203
Otávio Carvalho, Manuel Garcia, Eduardo Roloff, Emmanuell Diaz Carreño, and Philippe O. A. Navaux

Model-R: A Framework for Scalable and Reproducible Ecological Niche Modeling 218
Andrea Sánchez-Tapia, Marinez Ferreira de Siqueira, Rafael Oliveira Lima, Felipe Sodré M. Barros, Guilherme M. Gall, Luiz M. R. Gadelha Jr., Luís Alexandre E. da Silva, and Carla Osthoff

Parallel and Distributed Algorithms

Task Scheduling for Processing Big Graphs in Heterogeneous Commodity Clusters 235
Alejandro Corbellini, Daniela Godoy, Cristian Mateos, Silvia Schiaffino, and Alejandro Zunino

Exploring Application-Level Message-Logging in Scalable HPC Programs . . . 250
Esteban Meneses

Accelerated Numerical Optimization with Explicit Consideration of Model Constraints . . . 255
Lucia Damiani, Ariel Ivan Diaz, Javier Iparraguirre, and Aníbal M. Blanco

Parallel Processing of Intra-cranial Electroencephalogram Readings on Distributed Memory Systems . . . 262
Leonardo Piñeyro and Sergio Nesmachnow

Support Vector Machine Acceleration for Intel Xeon Phi Manycore Processors . . . 277
Renzo Massobrio, Sergio Nesmachnow, and Bernabé Dorronsoro

Performance Improvements of a Parallel Multithreading Self-gravity Algorithm . . . 291
Nestor Rocchetti, Daniel Frascarelli, Sergio Nesmachnow, and Gonzalo Tancredi

A Fast GPU Convolution/Superposition Method for Radiotherapy Dose Calculation . . . 307
Diego Carrasco, Pablo Cappagli, and Flavio D. Colavecchia

Grid, Cloud and Federations

Eeny Meeny Miny Moe: Choosing the Fault Tolerance Technique for my Cloud Workflow . . . 321
Leonardo Araújo de Jesus, Lúcia M. A. Drummond, and Daniel de Oliveira

Energy Aware Multiobjective Scheduling in a Federation of Heterogeneous Datacenters . . . 337
Santiago Iturriaga and Sergio Nesmachnow

Markov Decision Process to Dynamically Adapt Spots Instances Ratio on the Autoscaling of Scientific Workflows in the Cloud . . . 353
Yisel Garí, David A. Monge, Cristian Mateos, and Carlos García Garino

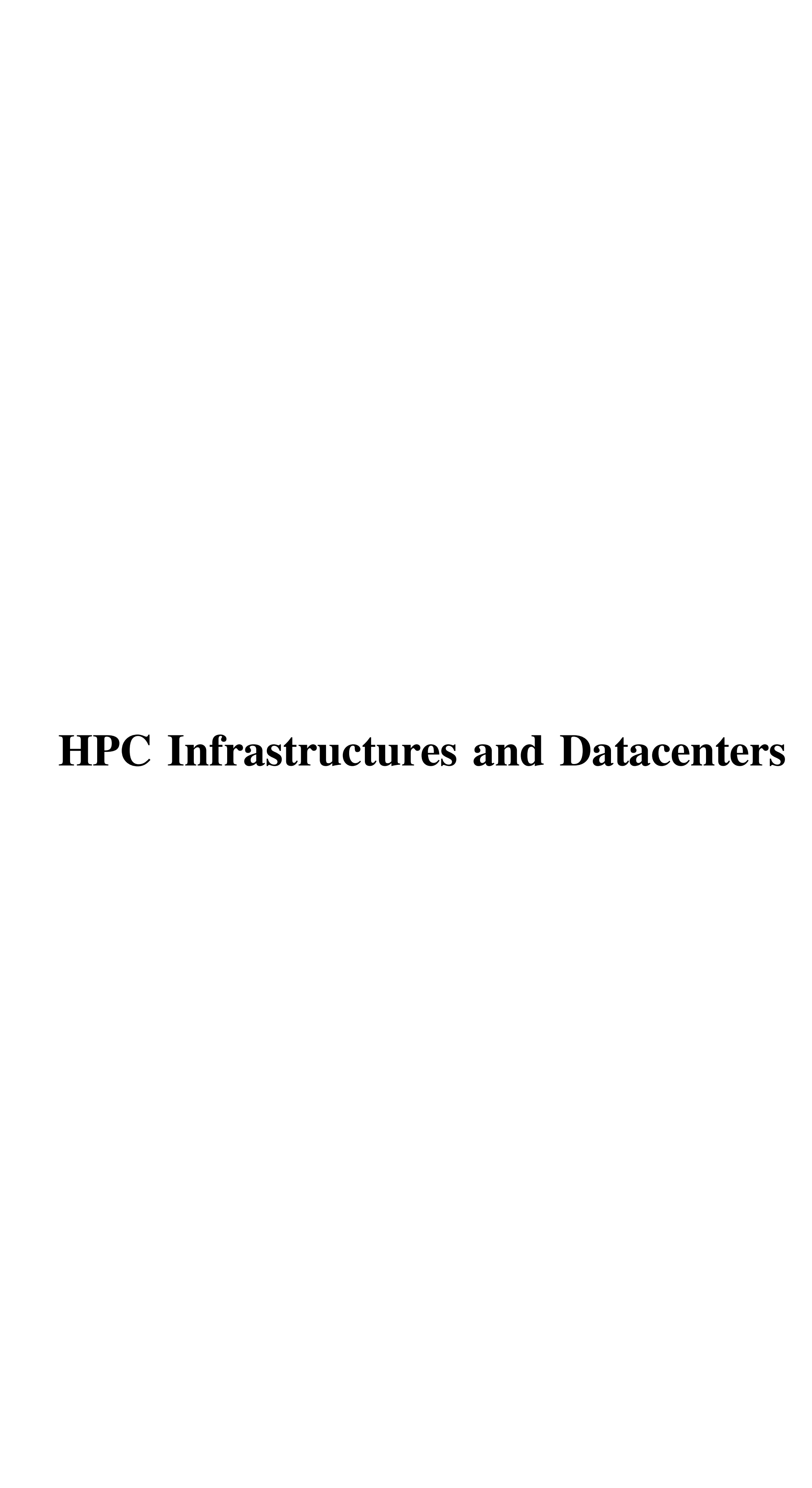
Experimental Analysis of Secret Sharing Schemes for Cloud Storage Based on RNS . . . 370
Vanessa Miranda-López, Andrei Tchernykh, Jorge M. Cortés-Mendoza, Mikhail Babenko, Gleb Radchenko, Sergio Nesmachnow, and Zhihui Du

Bi-objective Heterogeneous Consolidation in Cloud Computing 384
Luis-Angel Galaviz-Alejos, Fermín Armenta-Cano, Andrei Tchernykh, Gleb Radchenko, Alexander Yu. Drozdov, Oleg Sergiyenko, and Ramin Yahyapour

Scaling the Deployment of Virtual Machines in UnaCloud 399
Jaime Chavarriaga, César Forero-González, Jesse Padilla-Agudelo, Andrés Muñoz, Rodolfo Cáliz-Ospino, and Harold Castro

Distributed Cosmic Ray Detection Using Cloud Computing 414
Germán Schnyder, Sergio Nesmachnow, and Gonzalo Tancredi

Author Index 431



HPC Infrastructures and Datacenters

A Deep Learning Mapper (DLM) for Scheduling on Heterogeneous Systems

Daniel Nemirovsky[1(✉)], Tugberk Arkose[1], Nikola Markovic[2], Mario Nemirovsky[1,3], Osman Unsal[1], Adrian Cristal[1], and Mateo Valero[1]

[1] Barcelona Supercomputing Center, Barcelona, Spain
{daniel.nemirovsky,tugberk.arkose,mario.nemirovsky,osman.unsal, adrian.cristal,mateo.valero}@bsc.es
[2] Microsoft, Belgrade, Serbia
nimarkov@microsoft.com
[3] ICREA, Barcelona, Spain

Abstract. As heterogeneous systems become more ubiquitous, computer architects will need to develop new CPU scheduling approaches capable of exploiting the diversity of computational resources. Advances in deep learning have unlocked an exceptional opportunity of using these techniques for estimating system performance. However, as of yet no significant leaps have been taken in applying deep learning for scheduling on heterogeneous systems.

In this paper we describe a scheduling model that decouples thread selection and mapping routines. We use a conventional scheduler to select threads for execution and propose a deep learning mapper to map the threads onto a heterogeneous hardware. The validation of our preliminary study shows how a simple deep learning based mapper can effectively improve system performance for state-of-the-art schedulers by 8%–30% for CPU and memory intensive applications.

1 Introduction

Heterogeneous computational resources have allowed for effective utilization of increasing transistor densities by combining very fast and powerful cores with more energy efficient cores as well as integrated GPUs and other accelerators. Interest in heterogeneous processors within the industry has recently translated into several practical implementations including ARM's big.Little [8]. However, in order to fully utilize and exploit the opportunities that heterogeneous architectures offer, multi-program and parallel applications must be properly managed by a CPU scheduler. As a result, heterogeneous scheduling has become a popular area of research and will be essential for supporting new diverse architectures down the line.

Effective schedulers should be aware of a system's diverse computational resources, the variances in thread behaviors, and be able to identify patterns related to a thread's performance on different cores. Furthermore, since applications may perform differently on distinct core types, an efficient scheduler

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 3–20, 2018.
https://doi.org/10.1007/978-3-319-73353-1_1

should be able to estimate performances in order to identify an optimal mapping scheme. Mapping determines which thread to send to which core and is a problem that shares similarities with recommendation systems and navigation systems both of which have benefitted using machine and deep learning.

Deep learning (DL) techniques and deep neural networks (DNNs) in particular are beginning to be utilized in a wide variety of fields due to their great promise in learning relationships between input data and numerical or categorical outputs. The relationships are often hard to identify and program manually but can result in excellent prediction accuracies using DNNs. Though DL techniques have been gaining traction over the last few years, its application toward improving hardware performance remains in its earliest stages. As of yet, there has been no seminal work applying DL for predicting thread performance on heterogeneous systems and maximizing system throughput.

The objective of this work is the proof of concept of the opportunities that arise by applying DL to computer architecture designs. The novelty of this work centers on decoupling the selection and mapping mechanisms of a heterogeneous scheduler and fundamentally, the implementation of a deep learning mapper (DLM) which uses a DNN to predict system performance. The selector remains responsible for ensuring fairness and selecting the threads to execute next scheduling quantum while the mapper is charged with identifying an optimal mapping of selected threads onto available cores. Initial results of our proposal are promising, the DLM is capable of improving the performance of existing conventional schedulers (round-robin, fairness-aware, Linux CFS) by 8%, 20%, and 30% respectively for computational and memory intensive applications.

Our contributions include:

- A heterogeneous scheduling model which abstracts and decouples thread selection and mapping.
- An implementation of a deep learning mapper (DLM) that uses a deep neural network for predicting the system performance of different mapping schemes. To our knowledge this work is the first to apply deep learning to CPU scheduling for heterogeneous architectures.

The rest of this paper is structured as follows. Section 2 discusses our motivation and a brief technical overview of mapping, machine/deep learning techniques, and heterogeneous scheduling issues. Section 3 presents our proposed scheduling model with a description of a practical implementation. Validation of our implementation with experimental results is found in Sect. 4. Lastly, we discuss related work in Sect. 5 and future work and conclusion in Sect. 6.

2 Motivation

This section highlights the efficiency opportunities attainable by optimizing mapping on heterogeneous systems and also discusses the rationale for applying DL towards predicting system performance and how decoupling the thread selection and mapping mechanisms can provide model scalability while still ensuring fairness for all threads.

2.1 Mapping

Finding the optimal thread to core mapping on a heterogeneous system is no trivial feat. This is especially the case when executing diverse workloads since the performance of each application is likely to vary from quantum to quantum and core to core. These differences can vary from application to application as well as from phase to phase within an application.

Figure 1 illustrates the performance differences that result from executing SPEC2006 on a large core compared to a small core (for core details see Sect. 4.1). On average, the applications achieve about 2x better system instructions per cycle (IPC) when executing on the large core vs. the small core. Variations in IPC differences can also be observed between applications. For some applications, these IPC differences can be either very minor (mcf 29%, bzip2 33%, and hmmer 36%) or very sizable (gemsFDTD 171%, omnetpp 161%, and perlbench 153%). These variations can be partially explained by the code's structure and algorithms including loops, data dependencies, I/O and system calls, and memory access patterns among others.

The inter-application variations in core to core IPC differences also exist within the different basic blocks and phases of every application (intra-application). The more inter-application variations of core to core IPC differences there are, the harder it is for a scheduler to identify the optimal mapping scheme, but the greater opportunities for improvement.

To showcase how identifying these core to core IPC differences can translate into mapping benefits, consider the case where four applications (e.g. A, B, C, and D) are selected to run on a system with 1-large core and 3-small cores. Four mapping schemes which assign one application to the large core and the other three to the small cores can be A-BCD, B-CDA, C-DAB, D-ABC. Each mapping scheme will produce a different resulting system IPC. The overall benefits of an effective mapper will be based upon the difference between the best and worst mapping schemes. For instance if A-BCD is the best mapping scheme resulting in a system IPC of 4 and C-DAB is the worst with a system IPC of 2, then the difference in percentage terms would be 100% (i.e. $(4-2)/2$).

To demonstrate this in practical terms, we found the differences between the best and worst mapping schemes for all possible combinations of four applications from the SPEC2006 benchmark suite. The differences in system performance between the best and worst possible mapping scheme for each combination of four SPEC2006 applications range from 1%–36%. On average, identifying the most adventageous mapping scheme for a given set of four SPEC2006 applications on a 1-large 3-small core system can lead to 16% improvements in system performance. These results expose the theoretical benefits that may be gained from an effective scheduler at the application level granularity. Practical schedulers, however, work at the quantum level granularity and may additionally identify and take advantage of intra-application core to core performance differences which could expose greater opportunities for mapping optimization.

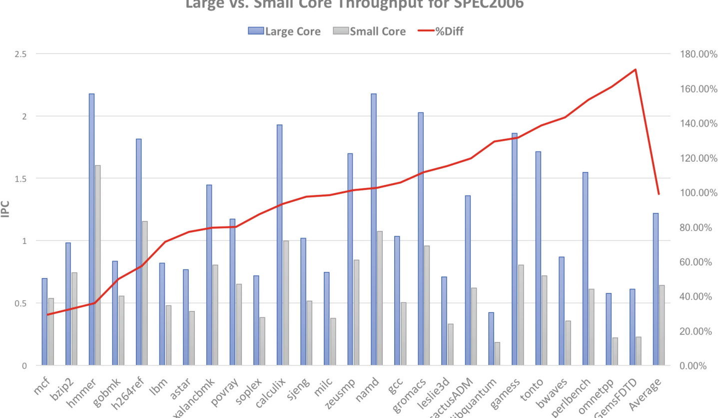


Fig. 1. The performance differences that result from executing each SPEC2006 benchmark on a large vs. small core.

In order to identify an optimal mapping scheme, a heterogeneous scheduler should be able to estimate the system performance that each individual mapping scheme would produce. Conventional schedulers such as the Linux Completely Fair Scheduler (CFS), however, typically do not make use of the mechanisms needed to exploit this potential. As we shall see, deep learning can be an effective tool for schedulers to utilize in order to help estimate system performance.

2.2 Machine/Deep Learning

Part of the attraction of machine/deep learning is the flexibility that its algorithms provide to be useful in a variety of distinct scenarios and contexts. For instance, advances in computer vision and natural language processing using convolutional neural network techniques [10,12] have led to high levels of prediction accuracy enabling the creation of remarkably capable autonomous vehicles and virtual assistants. In particular, it is our belief that the predictive power of artificial neural networks (ANNs) will be of great use for computer architects seeking to improve system performance and optimize the utilization of diverse hardware resources. Deep learning (DL) methods expand on more simplistic machine learning techniques by adding depth and complexities to existing models. Using deep ANNs (DNNs) or ensembles of different machine learning models is a typical example of DL.

DNNs consist of a set of input parameters connected to a hidden layer of artificial neurons which are then connected to other hidden layers before connecting to one or more output neurons. The inputs to the hidden and to the output neurons are each assigned a numerical weight that is multiplied with its corresponding input parameter and then added together with the result of the neuron's other incoming connections. The sum is then fed into an activation

function (usually a rectified linear, sigmoid, or similar). The output of these neurons is then fed as input to the next layer of neurons or to the output neuron(s).

A DNN can learn to produce accurate predictions by adjusting its weights using a supervised learning method and training data. This is performed via a learning algorithm such as backpropagation that adjusts the weights in order to find an optimal minima which reduces the prediction error based on an estimated output, the target output, and an error function. Advances in learning algorithms have enabled faster training times and allowed for the practical use of intricate DNN architectures. DNNs can also keep learning dynamically (often called online learning) by periodically training as new data samples are generated. Moreover, several of the training calculations may be executed in parallel for the neurons within the same layer. The latency of these calculations can be further mitigated through the use of hardware support including GPUs, FPGAs, or specialized neural network accelerators.

2.3 Program Behaviors and CPU Scheduling

Recognizing and exploiting the behavioral variations of programs is instrumental for achieving optimal scheduling schemes to maximize fairness and system performance. Behaviors represent the different characteristics of the program or thread while executing on the physical cores. These can include cache accesses and miss rates, branch prediction accuracies, and instructions per cycle (IPC). While not all programs exhibit the same behavior, studies [7,24] have revealed that the behavioral periodicity in different applications is typically consistent. In fact, the behavioral periodicity has been shown to be roughly on the order of several millions of instructions and is present in various different and even non correlated metrics stemming from looping structures inside of applications. Behavioral variations may be additionally influenced by interference effects between threads. These effects are generally due to shared data and physical resources between threads and should be taken into consideration by an optimal scheduler.

Yet, even after accounting for program behaviors, finding the optimal scheduling scheme is far from simple. CPU schedulers rely chiefly upon two mechanisms to fulfill their policy objectives: (1) thread selection and (2) thread to core mapping. The thread selection mechanism is responsible for selecting a subset of threads to run from a larger pool of available threads. It does so by using heuristics which order the threads using priorities or scores related to how critical the threads are (e.g. time constrained or system level tasks may be given a higher priority than background tasks which search for application updates) or how much execution time or progress the threads have made so far. The selection mechanism also generally ensures that no threads are continually starved of system resources thereby guaranteeing a certain level of fairness. On homogeneous systems where all cores are identical, the task of mapping individual threads to particular cores depends mainly upon keeping threads close to their data in the cache hierarchy. On heterogeneous systems, in contrast, the mapping mechanism must take into regard the different microarchitectural characteristics of the cores in order to find an optimal mapping of the threads to the

cores which is the most effective for its scheduling objective. As a result, schedulers targeted towards homogeneous systems are unable to optimally exploit the resource diversity in heterogeneous systems.

The current Linux Completely Fair Scheduler (CFS) [19] is one such example of a homogeneous scheduler. The state-of-the-art CFS selection scheme combines priorities with execution time metrics in order to select the threads to run next, however, the mapping scheme is relatively simplistic. When mapping, the CFS evenly distributes the threads onto the cores such that all cores have approximately the same number of threads to run. These threads are effectively pinned to the core because they are only swapped with threads on their assigned core and not with those of another core (i.e. threads don't move from the core they were initially assigned to).

Heterogeneous architectures, however, provide excellent environments for exploiting the behavioral diversity of concurrently executing programs and several schedulers targeting these systems have been recently proposed. The fairness-aware scheduler by Van Craeynest et al. [27] is one such scheduler which works similarly to the CFS but instead of mapping all threads evenly on all cores and pinning them there, it maps the highest priority thread (i.e. the one that has made the fewest progress) to the most powerful core. For example, in a 4 core system with 1 powerful core and 3 smaller energy efficient cores, this scheduler will send the thread with the highest priority to the large core and the next 3 highest priority threads to the other 3 small cores.

Another scheduler targeted at heterogeneous systems is the hardware round-robin scheduler by Markovic et al. [15]. Instead of using priorities for thread selection, this approach chooses which threads to run next in a round-robin manner (thereby guaranteeing fairness) and then maps the selected threads to the cores. Using the same 4 core system as described above, this scheduler will rotate the threads in a manner similar to a first in first out queue, from small core to small core to small core to large core and then back into the thread waiting pool until all threads have had a chance to execute.

Scheduling also produces overheads which may reduce the total efficiency gains due to the cost of calculations as well as context swap penalties. It is therefore imperative for effective lightweight schedulers to balance finding an optimal scheduling scheme without triggering costly context swaps.

3 Scheduling Model

In this section we present our scheduling model (shown in Fig. 2) with decoupled thread selection and mapping mechanisms. This scheduling model uses a conventional scheduler (CS) to select a subset of available threads to execute next quantum (using its prioritization scheme) and the deep learning mapper (DLM) to map the selected threads onto the diverse system resources (using a throughput maximization scheme). The scheduling quantum (the periodicity to run the scheduler) chosen is 4 ms for the CS and 1ms for the DLM which reflect typical quantum granularities of CS approaches. This difference allows

the DLM to take advantage of the finer grained variations in program behaviors and optimize the mapping on the heterogeneous system while still maintaining CS objectives. Furthermore, the context swap penalties are generally lower for the DLM since it only swaps threads which are already running and have data loaded in the caches while the CS may select to run any thread that may not have any of its data in the caches.

In addition to selecting the threads to run next, the CS is responsible for thread management, including modifying their statuses, dealing with thread stalls, and mapping for the first quantum of new threads or when the number of available threads is less than the number of available cores. When active, the DLM essentially provides a homogeneous abstraction of the underlying heterogeneous hardware to the CS since it only needs to select threads to run and not whether to execute on a large or small core.

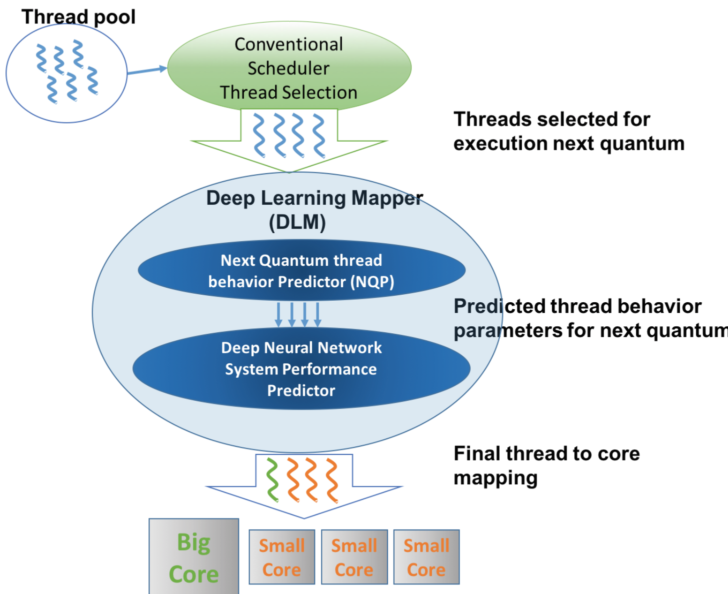


Fig. 2. The scheduling model. A conventional scheduler is used to select the threads to run next quantum and the DLM then uses the NQP and DNN predictor to find the optimal mapping to maximize system performance.

3.1 Deep Learning Mapper (DLM)

The DLM is responsible for finding a mapping of the selected threads onto the hardware cores which optimizes system throughput. This objective helps to demonstrate the significant potential that using DNN based performance predictors can have for a continuously busy system. The DLM works by firstly collecting statistical information about each selected thread pertaining to its

behavior (described in Sect. 3.1). These are gathered during the thread's previous execution quantum. These statistics are then passed along to the next quantum behavior predictor (NQP) that predicts that the thread's behavior during the next execution quantum will be the same as during its previous quantum. The NQP in essence forwards the behavioral statistics for all threads that have been selected to execute next quantum to our DNN based performance predictor. The DNN is able to estimate the system performance for a given mapping scheme of the threads selected to run next quantum. To identify the most advantageous mapping scheme to initiate for the next quantum, the DLM will utilize the DNN to make separate predictions for all possible mapping schemes given the selected threads and then choses the scheme that results in the highest estimated system performance.

Thread statistics and parameter engineering. It is important to carefully determine the appropriate set of thread statistics that characterize thread behaviors and will be used as input parameters to our system performance predictor. This process, otherwise known as parameter engineering, is critical since the accuracy of the system predictor depends upon the ability of the neural network to find causal relationships between these inputs and the expected output.

Normalizing the statistics into ratios helps to achieve parameter generalization. Using ratios instead of real values such as generating an instruction mix where each instruction type is given as a ratio of the total instructions executed during the last quantum helps to achieve this generalization. Without using this type of normalization, we would be left with inconsistent statistical input to the DNN performance predictor. For example, the number of actual executed instructions of each type depend heavily on the microarchitecture of the cores (e.g. an out-of-order core may execute more instructions than an in-order core even though the instruction mix ratios may be the same). Different forms of generalization can also be used in cases when the core types have different ISAs or cache configurations. Generalizing statistics enables our approach to be useful in systems with a variety of different architectures.

In determining the final set of statistics, we sought to balance DNN predictor accuracy while minimizing the overheads due to gathering the statistics and the arithmetic operations needed to be performed. Based upon the heterogeneous system used in our work (detailed in Sect. 4.1), we identified 12 different thread statistics that are useful in describing thread behaviors on the cores and are inclusive of thread interference effects. The statistics are collected after a thread completes an execution quantum and are composed of the accesses and misses of the different structures of the cache hierarchy as well as the instruction mix executed. These 12 thread statistics (given as ratios) are: **(1)** DL1, **(2)** L2, and **(3)** L3 data cache miss ratios, instruction mix ratios including **(4)** loads, **(5)** stores, **(6)** floating point operations, **(7)** branches, and **(8)** generic arithmetic operations, **(9)** IL1 divided by DL1 loads, **(10)** L2 divided by DL1 misses, **(11)** L3 divided by DL1 misses, and **(12)** L3 divided by L2 misses.

The 12 statistics are saved as part of a thread's context after each quantum it executes, overwriting the values from the previous quantum. Many conventional CPUs come with hardware support for collecting similar statistics and in future work we will seek to further explore the set of statistics needed in order to mitigate collection and processing overheads while maintaining or improving the accuracy of our performance predictor.

Next quantum thread behavior predictor (NQP). Several novel approaches have been proposed which predict program behavior based on various statically or dynamically collected program statistics [7,26]. However, to keep overheads low and for simplicity, we use a next quantum thread behavior predictor (NQP) that always predicts the next behavior to be the same as the immediately anterior quantum behavior. The statistics forwarded by the NQP, therefore, are based on those collected during the thread's previous execution quantum.

Figure 3 helps to visualize the behavioral periodicity which the NQP must predict for. It shows the IPC variability of the perlbench and gamess benchmarks throughout their simulated execution on an Intel Nehalem x86 using a 1ms execution quantum. There are clearly periodic behavioral phases that span tens and sometimes hundreds of quanta. It is also possible to observe that for finer granularities, the IPC variation from quantum to quantum is quite minimal, and more so on the small core.

We measured the NQP accuracy results using the mean percentage error for the SPEC2006 benchmark suite. These applications were simulated executing on an Intel Nehalem x86 configuration using a 1ms execution quantum. The errors are calculated using Eq. 1 by measuring the IPC differences from quantum to quantum.

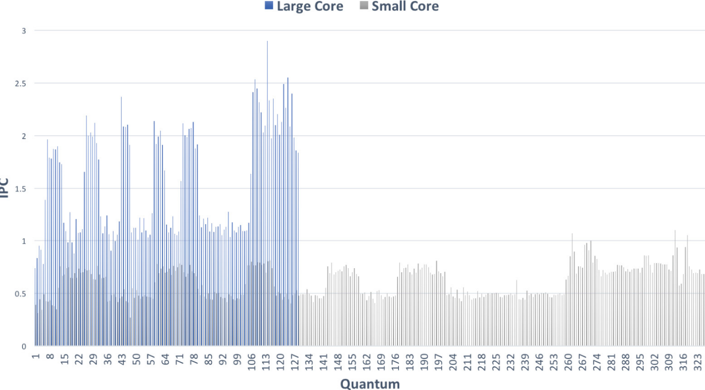
$$\begin{aligned} error_i &= \frac{|y_i - t_i|}{t_i} \\ \mu_{error} &= \frac{1}{n} \times \sum_{i=1}^{n} error_i \end{aligned} \tag{1}$$

where y is the predicted IPC and t is the target (i.e. observed) IPC value for quantum i and n is the total number of quanta (i.e. samples).

The NQP results in average errors of 10% for all SPEC2006 applications on both cores. However, the results vary between individual benchmarks with some outliers (e.g. cactusADM and soplex) exhibiting higher errors. These error variations can have a significant impact on the ability of the DNN predictor to properly predict and maximize system throughput.

DNN system performance predictor. The key component behind the DLM is a DNN system performance predictor which takes as input a set of parameters from as many individual threads as there are hardware cores and then outputs an estimated system IPC value. The system we target is a heterogeneous CPU architecture composed of 4 cores with 2 different core types (1 large core and

(a) The IPC per quantum behavior of perlbench.



(b) The IPC per quantum behavior of gamess.

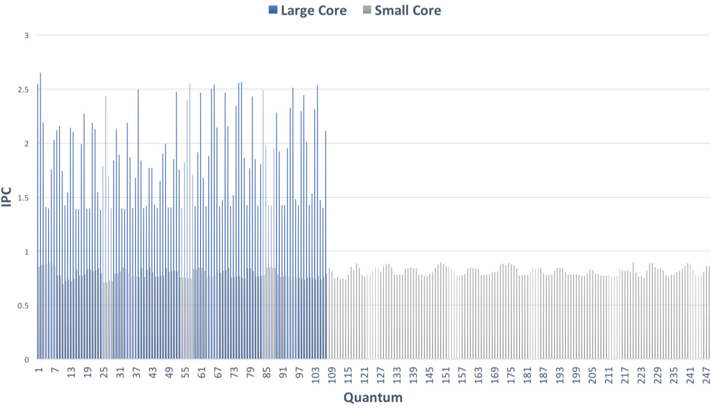


Fig. 3. The IPC per quantum behavior of four SPEC benchmarks when running on the large core compared to the small core.

3 small cores, described in Sect. 4.1). The DNN predictor takes as input the 12 normalized parameters from the 4 threads (selected for execution by the CS) for a total of 48 input parameters.

The order in which the threads are inputted to the DNN correspond to which physical core they would be mapped to with the first 12 thread parameters as corresponding to the thread mapped to the large core, and the next 36 parameters corresponding to the threads mapped to the three small cores. This way, we are able to estimate what the system IPC would be for different mapping combinations.

Mapping Combination	Big Core	Small Core 1	Small Core 2	Small Core 3	Predicted System IPC
1	A	B	C	D	3.7
2	B	A	C	D	4.8
3	C	A	B	D	3.9
4	D	A	B	C	4.5

Fig. 4. An example of how the DLM uses the DNN to predict for 4 different mapping combinations once it is passed the 4 threads selected by the CS (A, B, C, and D).

An example of this is given in Fig. 4. Here the CS has selected 4 threads (A, B, C, and D) from a larger pool of available threads to execute next quantum. There are 4 different combinations which we can map the 4 threads onto the hardware where each combination will have a different thread mapped onto the large core. The different mapping combinations represent the different ordering of the thread parameter inputs to the DNN. For instance, combination 1 will have the first 12 inputs correspond to thread A, the next 12 to thread B and so on. We can also consider all mapping permutations but since the only shared structure is the L3, there should be negligible differences in performance and interference effects. In the example, the DNN predictions for the 4 different combinations are given in the last column. Combination 2 has the highest estimated system and will be chosen as the optimal mapping scheme for the upcoming quantum.

We have implemented the DNN performance predictor using Python and the machine learning library scikit-learn [20]. An extensive exploration into the DNN architecture was conducted before settling upon the chosen design. Due to space concerns and the objective of this work being the proof of concept of the DLM, only a brief summary of the DNN design study is provided here.

Once the 12 input parameters were chosen, we evaluated numerous DNNs by modifying the hyperparameters of each including using different numbers of hidden layers, hidden units, activation functions, and training regularization techniques. We sought to balance prediction accuracy with implementation feasibility and made use of learning curves to gain insight into how many training samples the DNN needs to start predicting consistently for unseen data and how accurate these predictions are. Each training data sample consists of 48 input parameters and 1 target system IPC value. These are collected after each scheduling quantum which has resulted in the execution of 4 threads on the 4 cores. The algorithm used for training is a stochastic gradient based optimizer with L2 regularization which is readily used in machine learning models. During training, the weights of the neural network are adjusted after each full iteration of a batch of training data, always aiming to minimize the mean square error (mse) between the predicted output and the target output.

At the end of the design study, we settled upon a DNN implementation consisting of 48 total inputs, 5 hidden layers of 25 hidden units each, and a single output unit that use a rectified linear activation function. Figure 5 plots the learning curves of the training and 10-fold cross-validation results of the chosen

DNN. It highlights how, as the quantity of training data grows, so too does the accuracy and generalizability of the predictor when executing all the applications from SPEC2006. The score is measured in terms of correlation between the predicted system performance and the observed system performance using an R^2 coefficient. In particular, the figure shows that after about 15000 quanta, the correlation between the predicted performance and the observed performance on the data used to train is very high (about 0.96) and after about 35000 quanta, the correlation of stabilizes for the unseen validation data at about 0.64. The difference between the training and validation curves illustrates that the model has high variance which may indicate overfitting but can be explored in future work by adding more regularization and fine tuning the input parameters, hyperparameters, and sample data. Since our model is capable of online learning, however, the prediction errors introduced by running new applications will gradually settle at lower levels after training dynamically. Online learning works by continuing to train our DNN periodically after a certain number of new data samples are gathered. For our online DNN implementation, we have chosen to keep training our predictor every 20 execution quanta (i.e. a micro-batch of 20 samples). This requires needing to save only 20 quantum samples of data a time. The frequency of online training is related to the average number of quanta the benchmarks take to complete. A larger micro-batch could be used for longer applications or when the system is exceedingly busy in order to lower overheads.

3.2 Overheads

Schedulers typically add overheads due to the mapping calculations and resulting context swaps after each scheduling quantum. Since the DLM is triggered 4 times as often as the CS (1 ms vs 4 ms quantum), the DLM can also cause context swaps before the next CS quantum. A minimum of 0 and maximum of 4 extra context swaps can be issued by the DLM before the next CS quantum. However, the DLM will only trigger a swap if the resulting mapping is beneficial to overall system performance. The overheads due to the NQP and performance predictor amount to less than 4000 floating point operations per predicted mapping combination and less than 16000 in total. However, not only can a large quantity of these calculations be done in parallel, but this overhead is still orders of magnitude less than it costs to swap contexts and load the caches. Online training also adds overheads but is only done after every 20 quanta (or the chosen frequency of micro-batch training) and can be hidden by running it in the background when a core is idle.

Storing the 64-bit weights of the DNN requires about 21 KB of memory. The introduction of new statistical fields to save for each thread is also a minor overhead (96 bytes per thread) as is the memory needed to store the online training data (¡8 KB for 20 samples of 4 threads' worth of parameters). Lowering these overheads is a topic for future work but are still reasonable for a viable implementation of the scheduling model.

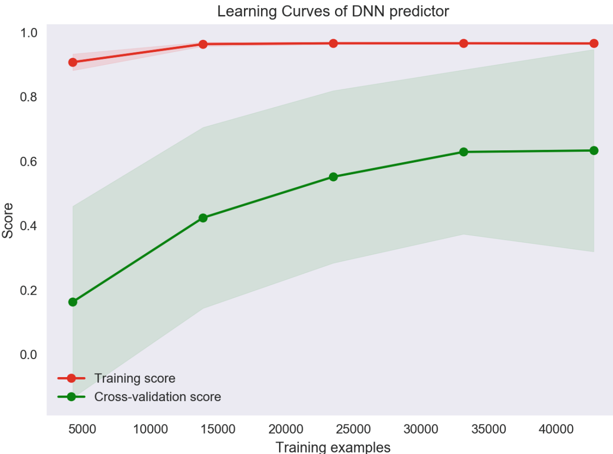


Fig. 5. The learning curve of the online DNN predictor. As the amount of training data increases the predictor becomes more generalized to account for different applications and behaviors. Higher y-axis numbers are better.

4 Evaluation

4.1 Methodology

This work uses the Sniper [3] simulation platform. Sniper is a popular hardware-validated parallel x86-64 multicore simulator capable of executing multithreaded applications as well as running multiple programs concurrently. The simulator can be configured to run both homogeneous and heterogeneous multicore architectures and uses the interval core model to obtain performance results.

The processor that is used for all experimental runs in this work is a quad-core heterogeneous asymmetric multi-core processor consisting of 1 large core and 3 identical small cores. Both core types are based on the Intel Nehalem x86 architecture running at 2.66 GHz. Each core type has a 4 wide dispatch width, but whereas the large core has 128 instruction window size, 16 cycle branch misprediction penalty (based on the Pentium M predictor), and 48 entry load/store queue, the small core has a 16 instruction window size, 8 cycle branch misprediction penalty (based on a one-bit history predictor), and a 6 entry load/store queue. The 1-large 3-small multi-core system configuration is based on the experimental framework used in previous work [15,27] that we evaluate our proposal against. These works also make use of Sniper, which unfortunately does not provide for a wide selection of different architectures such as ARM but does support hardware validated x86 core types. We believe that using the (admittedly limited) experimental setup as employed in previous work allows for the fairest comparison.

We have used the popular SPEC2006 [9] benchmark suites to evaluate and train our scheduling model. This is an industry-standardized, CPU-intensive benchmark suite, stressing a system's processor and memory subsystem. The entirety of the benchmark suite is used with the exception of some applications which did not compile in our platform (dealII, wrf, sphinx3). All 26 benchmarks are run from start to finish and the simulation ends after all the benchmarks finish. This is done to emulate a busy system which must execute a diverse set of applications. This setup is also useful in demonstrating the ability of the DLM to improve system throughput.

We evaluate the performance for three different conventional schedulers (round-robin [15], fairness-aware [27], and CFS [19]) with and without the use of a fully trained DLM. That is to say we compare how much each conventional scheduler may be improved (in terms of system throughput) by using the DLM instead of its typical mapping mechanism. To account for context switch overheads due to architectural state swapping, we apply a 1000 cycle penalty which is consistent with the value utilized in the round robin study. The additional cache effects from the context switches are captured by the simulation.

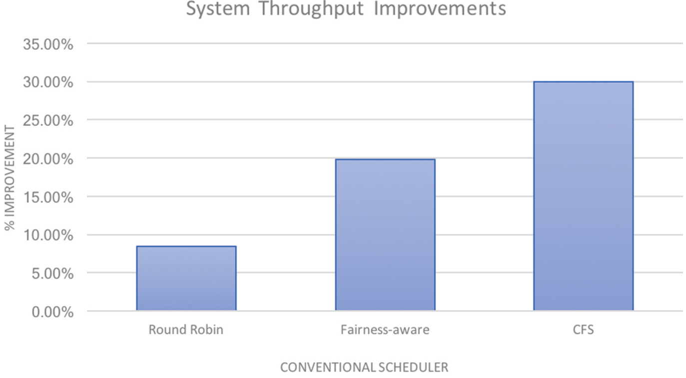


Fig. 6. Average system throughput (IPC) improvements when using the DLM for all SPEC2006. Higher numbers are better.

Figure 6 compares the system throughput improvements achieved for all 3 schedulers when using a DLM after running SPEC2006. The results show an average percentage throughput increase of 8%, 20%, and 30% for the round-robin, fairness-aware, and CFS schedulers respectfully. These improvements are significant especially for a preliminary study with a simple deep neural network predictor. They also highlight how effective the DLM is at benefitting all 3 different state-of-the-art schedulers. The improvements demonstrate the ability of the DLM to find more optimal mappings than the schedulers can by themselves. It achieves this thanks to two main factors. The DNN predictor allows the DLM to make highly accurate predictions for different mapping combinations while the 1ms quantum provides the opportunity to detect and adjust the mapping for

variations in thread behaviors. The differences in the throughput gains for the 3 schedulers are also consistent with how they perform relative to one another without the DLM. On a heterogeneous system, the round-robin scheduler has been shown to perform better than the fairness-aware scheduler, which in turn performs better than the CFS.

The average total percentage prediction error of the DLM (calculated using Eq. 1) for the experiments was 12%. The DLM errors are slightly higher than those of the DNN shown earlier (see Sect. 3.1) because the DLM includes errors from both the NQP and the DNN. This error rate is within reasonable margins and our results are notable when considering that the DLM still showed such significant throughput benefits.

5 Related Work

Much of the previous work using machine/deep learning for scheduling has been to classify applications, as well as to identify process attributes and a program's execution history. This is the approach of [18] which used decision trees to characterize whole programs and customize CPU time slices to reduce application turn around time by decreasing the amount of context swaps. The work presented in [13] studies the accuracy of SVMs and linear regression in predicting the performance of threads on two different core types. However, they do so at the granularity of 1 s, use only a handful of benchmarks, and do not implement the predictor inside of a scheduler.

The studies by Dorronsoro and Pinel [6,21] investigates using machine learning to automatically generate desired solutions for a set of problem instances and solve for new problems in a massively parallel manner. An approach that utilized machine learning for selecting whether to execute a task on a CPU or GPU based on the size of the input data is done by Shulga et al. [25]. Predicting L2 cache behavior is done using machine learning for the purpose of adapting a process scheduler for reducing shared L2 contention in [23].

In the work done by Bogdanski et al. [2], choosing parameters for task scheduling and loadbalancing is done with machine learning. However, their prediction is whether it is beneficial to run a pilot program that will characterize a financial application. They also assume that the computational parameters of the workload stay uniform over certain periods of time. Nearly all of these approaches deal with either program or process level predictions and target homogeneous systems.

Characterizing and exploiting program behavior and phases has been the subject of extensive research. Duesterwald et al. [7] and Sherwood et al. [24] showed that programs exhibit significant behavioral variation and can be categorized into basic blocks and phases which can span several millions to billions of instructions before changing. Work done in [26] has taken advantage of the compilers ability to statically estimate an applications varying level of instruction level parallelism in order to estimate IPC using monotonic dataflow analysis and simple heuristics for guiding a fetch-throttling mechanism.

A heterogeneous system containing various cores of the same ISA but of different types was proposed by Kumar et al. in [11]. Their process consists of deciding on the core that will perform in the most power efficient manner each time a new phase or program is detected using sampling techniques. Moncrieff et al. [17] and Menasce and Almeida [16] analytically examined the tradeoffs between utilizing fast and slow processors in heterogeneous processors. Their study showed that a system composed of few fast cores and many slow cores are effective in terms of cost and performance. Optimal scheduling of independent applications running on a preemptive heterogeneous CMP has been studied by Liu and Yang [14]. A separate study [1] aims to create a contention-aware scheduler that maximizes throughput by learning and mimicking the decisions of an oracle scheduler.

Chronaki et al. [4,5] propose a heterogeneous scheduler for a dataflow programming model which improves performance using a prioritization scheme and dynamic task dependency graph to assign newly created and critical tasks to fast cores. A statistical method using extreme value theory is used in [22] to determine the probabilities for optimal task assignment in massively multithreaded processors.

6 Future Work and Conclusion

In this paper we have presented a preliminary study which pioneers applying DL to heterogeneous scheduling. We outlined a scalable scheduling model that decouples thread selection and mapping routines. The thread selection mechanism of a conventional scheduler is used in conjunction with a deep learning mapper (DLM) to maintain fairness and increase system performance. The DLM uses a deep neural network to predict the system performance for different mapping options at the scheduling quantum granularity. This lightweight deep neural network can provide highly accurate predictions for a diverse set of applications while continuing to train dynamically. The validation of our approach shows that even a simple DL based mapper can significantly improve system performance for state-of-the-art schedulers by 8% to 30% for CPU and memory intensive applications.

We seek to expand the scope of our work in the future by further exploring thread behavioral statistics, alternative DL models, improving the NQP, and scalability issues. We would also like to study ensemble models could be used to further widen the scope of the DLM for dealing with irregular applications.

We hope that the novelty of this work has helped to highlight the value that using deep learning can offer towards improving system performance.

Acknowledgments. This work has been supported in part by the European Union (FEDER funds) under contract TIN2015-65316-P.

References

1. Anderson, G., Marwala, T., Nelwamondo, F.V.: Multicore scheduling based on learning from optimization models. Int. J. Innov. Comput. Inf. Control **9**(4), 1511–1522 (2013)
2. Bogdanski, M., Lewis, P.R., Becker, T., Yao, X.: Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations. In: 2011 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 496–501. IEEE (2011)
3. Carlson, T.E., Heirmant, W., Eeckhout, L.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12. IEEE (2011)
4. Chronaki, K., Rico, A., Badia, R.M., Ayguade, E., Labarta, J., Valero, M.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 329–338. ACM (2015)
5. Chronaki, K., et al.: Task scheduling techniques for asymmetric multi-core systems. IEEE Trans. Parallel Distrib. Syst. **28**(7), 2074–2087 (2017)
6. Dorronsoro, B., Pinel, F.: Combining machine learning and genetic algorithms to solve the independent tasks scheduling problem. In: 2017 3rd IEEE International Conference on Cybernetics (CYBCON), pp. 1–8. IEEE (2017)
7. Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and predicting program behavior and its variability. In: 12th International Conference on Parallel Architectures and Compilation Techniques, PACT 2003, Proceedings, pp. 220–231. IEEE (2003)
8. Greenhalgh, P.: big.little processing with arm cortex-a15 & cortex-a7 (2011). http://www.arm.com/files/downloads/bigLITTLE_Final_Final.pdf
9. Henning, J.: SPEC CPU2006 benchmark descriptions. In: Proceedings of the ACM SIGARCH Computer Architecture News, pp. 1–17 (2006)
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
11. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36, Proceedings, pp. 81–92. IEEE (2003)
12. LeCun, Y., Kavukcuoglu, K., Farabet, C.: Convolutional networks and applications in vision. In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 253–256. IEEE (2010)
13. Li, C.V., Petrucci, V., Mossé, D.: Predicting thread profiles across core types via machine learning on heterogeneous multiprocessors. In: 2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 56–62. IEEE (2016)
14. Liu, J.W., Yang, A.T.: Optimal scheduling of independent tasks on heterogeneous computing systems. In: Proceedings of the 1974 Annual Conference, vol. 1, pp. 38–45. ACM (1974)
15. Markovic, N., Nemirovsky, D., Milutinovic, V., Unsal, O., Valero, M., Cristal, A.: Hardware round-robin scheduler for single-ISA asymmetric multi-core. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 122–134. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_10

16. Menasce, D., Almeida, V.: Cost-performance analysis of heterogeneity in supercomputer architectures. In: Proceedings of Supercomputing 1990, pp. 169–177. IEEE (1990)
17. Moncrieff, D., Overill, R.E., Wilson, S.: Heterogeneous computing machines and Amdahl's law. Parallel Comput. **22**(3), 407–413 (1996)
18. Negi, A., Kumar, P.K.: Applying machine learning techniques to improve Linux process scheduling. In: TENCON 2005, 2005 IEEE Region 10, pp. 1–6. IEEE (2005)
19. Pabla, C.S.: Completely fair scheduler. Linux J. **2009**(184), 4 (2009)
20. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
21. Pinel, F., Dorronsoro, B.: Savant: automatic generation of a parallel scheduling heuristic for map-reduce. Int. J. Hybrid Intell. Syst. **11**(4), 287–302 (2014)
22. Radojković, P., Čakarević, V., Moretó, M., Verdú, J., Pajuelo, A., Cazorla, F.J., Nemirovsky, M., Valero, M.: Optimal task assignment in multithreaded processors: a statistical approach. ACM SIGARCH Comput. Architect. News **40**(1), 235–248 (2012)
23. Rai, J.K., Negi, A., Wankar, R., Nayak, K.: A machine learning based meta-scheduler for multi-core processors. In: Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts, pp. 226–238. IGI Global (2012)
24. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. IEEE Micro **23**(6), 84–93 (2003)
25. Shulga, D., Kapustin, A., Kozlov, A., Kozyrev, A., Rovnyagin, M.: The scheduling based on machine learning for heterogeneous CPU/GPU systems. In: NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), 2016 IEEE, pp. 345–348. IEEE (2016)
26. Unsal, O.S., Koren, I., Khrishna, C., Moritz, C.A.: Cool-Fetch: a compiler-enabled IPC estimation based framework for energy reduction. In: Eighth Workshop on Interaction between Compilers and Computer Architectures, INTERACT-8 2004, pp. 43–52. IEEE (2004)
27. Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A., Eeckhout, L.: Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, pp. 177–188. IEEE Press (2013)

Power Consumption Characterization of Synthetic Benchmarks in Multicores

Jonathan Muraña[1(✉)], Sergio Nesmachnow[1], Santiago Iturriaga[1], and Andrei Tchernykh[2]

[1] Universidad de la República, Montevideo, Uruguay
{jmurana,sergion,siturria}@fing.edu.uy
[2] CICESE Research Center, Ensenada, Baja California, Mexico
chernykh@cicese.mx

Abstract. This article presents an empirical evaluation of power consumption of synthetic benchmarks in multicore computing systems. The study aims at providing an insight of the main power consumption characteristics of different applications when executing over current high performance computing servers. Three types of applications are studied executing individually and simultaneously on the same server. Intel and AMD architectures are studied in an experimental setting for evaluating the overall power consumption of each application. The main results indicate the power consumption behavior has a strong dependency with the type of application. An additional performance analysis shows that the best load of the server regarding energy efficiency depends on the type of the applications, with efficiency decreasing in heavily loaded situations. These results allow characterizing applications according to power consumption, efficiency, and resource sharing, and provide useful information for resource management and scheduling policies.

Keywords: Green computing · Energy efficiency · Multicores
Computing efficiency

1 Introduction

Nowadays, data centers are key infrastructures for executing industrial and scientific applications. Data centers have become highly popular for providing storage, computing power, middleware software, and others information technology (IT) utilities, available to researchers with ubiquitous access [3]. However, their energy efficiency has become a major concern in recent years, having a significant impact on monetary cost, environment, and guarantees for service-level agreements (SLA) [4].

The main sources of power consumption in data centers are the computational resources and the cooling system [13]. When focusing on power consumption due to resource utilization, several techniques for hardware and software optimization can be applied to improve energy efficiency. Software characterization techniques [1] are used to determine features that are useful to analyze

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 21–37, 2018.
https://doi.org/10.1007/978-3-319-73353-1_2

the software behavior, including power consumption [2]. Estimation of power consumption is an important issue that can determine the quality and even the feasibility of both software products and big data center infrastructures.

In this line of work, this article focuses on the characterization of power consumption for scientific computing applications over nowadays multicore hardware used in scientific computing platforms. Such characterization is useful for designing energy efficient scheduling strategies for scientific computing platforms. Three synthetic benchmarks are studied over two physical setups from a real High Performance Computing (HPC) platform, registering their power consumption with a power meter device. Furthermore, the experimental analysis studies the power consumption of different applications sharing a computing resource via simultaneous execution. The proposed study is very relevant for nowadays data centers and HPC infrastructures, where many applications are executed at the simultaneously. This impacts on both the energy efficiency and the quality of service (QoS) offered to the users of the platform.

The article is organized as follows. Section 2 reviews related works on energy characterization in multicores. Section 3 describes the proposed methodology for energy characterization, the benchmarks, and the physical setup for experiments. The experimental results are reported and discussed in Sect. 4. The conclusions and the main lines for future work are presented in Sect. 5.

2 Related Work

The analysis of related works allows identifying two types of studies. A first group proposes the empirical evaluation of scientific applications to develop specific models or to adjust existing models for studying power and performance behavior. Other works introduces optimization techniques over existing models and presents optimized results. This section focuses on reviewing those articles about empirical measurements and evaluation of power consumption models.

Iturriaga et al. [7] studied the problem of finding schedules with appropriate trade-off between power consumption and execution time in heterogeneous computers systems, considering uncertainty. Specific versions of well-known heuristic were proposed for scheduling on realistic scenarios, applying the power consumption model in [12] and considering only CPU-bound workloads. A model for uncertainty on power consumption was determined through empirical evaluations using three CPU-bound benchmarks. Regarding scheduling results, online heuristics computed better schedules than offline approaches. Results also confirmed that uncertainly has a significant impact in the accuracy of the scheduling algorithms. The power consumption behavior of CPU-bound benchmarks shown in [7] is consistent with the one in our research. Moreover, we propose a fully empirical power consumption characterization, considering also two additional types of benchmarks: memory bound and disk bound.

Srikantaiah et al. [14] studied workload consolidation strategies for energy optimization in cloud computing systems. An empirical study of the relationship between power consumption, performance, and resource utilization was presented. The experiments were executed in four physical server connected to a

power meter to track the power consumption. The resource utilization was monitorized using the Xperf toolkit. Only two resources were considered in the study: processor and disk. The performance degraded for high levels of disk utilization, and variations in CPU usage did not result in significant performance variations. Energy results were presented in terms of power consumption per transaction (including power consumption in idle state), resources utilization, and performance degradation. On the other hand, results showed that power consumption per transaction, is more sensitive to CPU utilization than disk utilization. The authors also proposed an heuristic method to solve a modified bin packing problem where the servers are bins and the computing resources are bin dimensions. Results were reported for small scenarios where the power consumption of the solutions computed by the heuristic is about 5% more than the optimal solution. The tolerance for performance degradation was 20%.

Du Bois et al. [5] presented a framework for generating workloads with specific features, applied to compare energy efficiency in commercial systems. CPU-bound, memory-bound, and disk-bound benchmarks were executed on a power monitoring setup composed of an oscilloscope connected to the host and a logging machine to persist the data. Two commercial systems were studied: a high-end with AMD processors and a low-end with Intel processors. Benchmarks were executed independently, isolating the power consumption of each resource. Results confirmed that energy efficiency depends on the workload type. Comparatively, the high-end system had better results for the CPU-bound workload, the low-end system was better for disk-bound, and both had similar efficiency for the memory-bound workload. Our work complements this approach by including a study of the power consumption behavior when executing different types of tasks simultaneously on specific architectures for high performance computing.

Feng et al. [6] evaluated the energy efficiency of a high-end distributed system, with focus on scientific workloads. The authors proposed a power monitoring setup that allows isolating the power consumption of CPU, memory, and disk. The experimental analysis studied single node executions and distributed executions. In the single node experiments, results of executing a memory-bound benchmark showed that the total power consumption is distributed as follow: 35% corresponds to the CPU, 16% corresponds to the physical memory and 7% corresponds to the disk. The rest is consumed by power supply, fans, network, and other components. Idle state represented 66% of the total power consumption. In distributed experiments, benchmarks that are intensive in more than one computing resource were studied. Results showed that energy efficiency increased with the number of nodes used for execution.

Kurowski et al. [9] presented a data center simulator that allows specifying various energy models and management policies. Three types of theoretical energy models are proposed: (i) *static approach*, which consider a unique power value by processing unit; (ii) *dynamic approach*, which consider power levels, representing the usage of the processing unit; and (iii) *application specific approach*, which consider usage of application resources to determine the power consumption. Simulation results were compared with empirical measurements

over real hardware to validate the theoretical energy models in arbitrary scenarios. All models obtained accurate results (error was less than 10% with respect to empirical measurements), and the dynamic approach was the most precise.

Langer et al. [10] studied energy efficiency of low voltage operations in manycore chips. Two scientific applications were considered for benchmarking over a multicore simulator. The performance model considered for a chip was $S = a_k(\sum f_i) + b_k$, where S are the instructions per cycle, f is the frequency of the core i and a_k,b_k are constants that depend on k, the number of cores in the chip. A similar model is used for power consumption. Across 25 different chips, an optimization method based on integer linear programming achieved 26% in energy savings regarding to the power consumption of the faster configuration.

Several works in literature have focused on modeling and characterizing power consumption of scientific applications. However, to the best of our knowledge there is no empirical research focused on the inter-relationship between power consumption and CPU, memory, and disk utilization. Also, there is no experimental analysis of critical levels of resource utilization (close to 100%) and its impact on power consumption and performance. This article contributes in this line of research, proposing empirical analysis for both aforementioned issues.

3 Methodology for Power Consumption Evaluation

This section describes the proposed methodology for power consumption evaluation, the benchmarks and architectures studied, the power evaluation setup, and the experiments designed.

3.1 Overview of the Proposed Methodology

Experiments characterize the power consumption of the most relevant computing resources: CPU, memory, and disk [5–7]. We aim at studying holistic behaviors and analyzing the power consumption of hosts close to 100% of computing resource utilization. The analysis is complemented with performance experiments to study the trade-off between power consumption and performance degradation.

3.2 Benchmarks

The benchmarks used in the analysis are part of the Sysbench toolkit [8]. Sysbench is a cross-platform software written in C that provides CPU, memory, and disk intensive benchmarks for performance evaluation. The components used in the experiments are:

1. *CPU-bound*: an algorithm that calculates $\pi(n)$ (the prime counting function) using a backtracking method. The algorithm contains loops, square root and module operations, as described in Algorithm 1.

Algorithm 1. CPU-bound benchmark code

```
1: C ← 3
2: while C < MAX_PRIME do
3:    T ← sqrt(C)
4:    L ← 2
5:    while L < T do
6:       if C (mod T) = 0 then
7:          break
8:       end if
9:       if L > T = 0 then
10:         N ← N + 1
11:      end if
12:      L ← L + 1
13:   end while
14:   C ← C + 1
15: end while
16: return N
```

2. *Memory-bound*: a program that executes write operations in memory, as described in Algorithm 2, where the *BUF* variable is an array of integers. The cells of the array are overwritten with value of *TMP* until the last position of the array, i.e., the value of the *END* variable.

Algorithm 2. Memory-bound benchmark code

```
1: while BUF < END do
2:    *BUF ← TMP
3:    BUF ← BUF + 1
4: end while
```

3. *Disk-bound*: a program that reads/writes content in files. Read or write requests are generated randomly and executed until a given number of requests ($MAX_REQUEST$) is reached, as described in Algorithm 3.

Algorithm 3. Disk-bound benchmark code

```
1: while REQS_COUNT < MAX_REQS do
2:    REQ ← generate_rnd_request()
3:    if is_read(REQ) then
4:       read(REQ.FILE)
5:    end if
6:    if is_write(REQ) then
7:       write(REQ.FILE)
8:    end if
9:    REQS_COUNT++
10: end while
```

3.3 Multicore Hosts and Power Monitoring Setup

Experiments were performed on Cluster FING, the HPC platform from Universidad de la República, Uruguay [11]. Two hosts were chosen according to their characteristics and availability: HP Proliant DL385 G7 server (2 AMD Opteron 6172 CPUs, 12 cores each, and 72 GB RAM), and HP Proliant DL380 G9 server (2 Intel Xeon CPU E5-2680v3 CPUs, 12 cores each, and 128 GB RAM).

Figure 1 presents the power monitoring setup. Benchmarks were executed in a host connected to the power source via a Power Distribution Unit (PDU) to register the instant power consumption. In an secondary machine, a polling demon logged data for post-processing. This configuration is similar to the one used in related works [5,7].

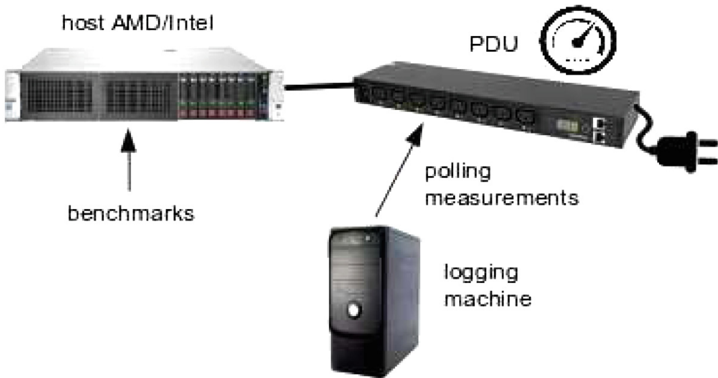


Fig. 1. Power monitoring setup

3.4 Design of Experiments

The power consumption (PC) of each host is computed as the average of 20 independent executions for each benchmark and combination of benchmarks. Idle consumption (IC), i.e., average consumption of the host without load, is registered to compute the effective consumption (EC) as $EC = PC - IC$.

In a first stage, benchmarks are evaluated independently from each other, analyzing only one resource. Utilization level (UL) is defined as the percentage of processors being used regarding the total number of processors in the host. Eight ULs were considered for single benchmark execution: 12%, 25%, 37.5%, 50%, 62.5%, 75%, 87.5% and 100%. Figure 2 shows an example of 37.5% utilization level: the host has 24 processors of which 9 execute instances of the CPU-bound benchmark. The remaining processors are in *idle* state.

In a second stage, the simultaneous execution of benchmarks is evaluated, analyzing several combinations of resource utilization at the same time. Two and three benchmarks are executed together in different ULs. In this case, UL is a vector where each entry represents the percentage of processors being used by

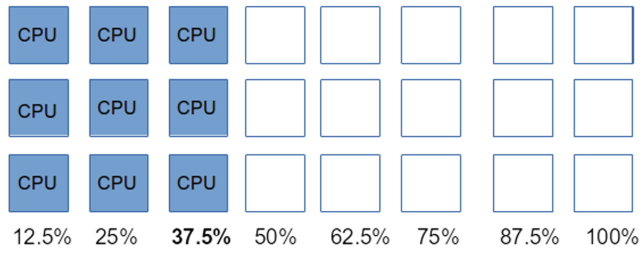


Fig. 2. CPU-bound benchmark, UL of 37.5%

each type of benchmark considered in the study (CPU-bound, memory-bound, and disk-bound, in that order). The utilization levels chosen were (25%, 25%), (25%, 50%), (25%, 75%), (50%, 25%), (50%, 50%), (75%, 25%) in pair executions and (25%, 25%, 25%), (25%, 25%, 50%), (25%, 50%, 25%), (50%, 25%, 25%) in triple executions. Figure 3 shows CPU-bound and memory-bound executing together with UL of (25%, 50%) and Fig. 4 shows CPU-bound, memory-bound and disk-bound executing combined with UL of (25%, 25%, 50%).

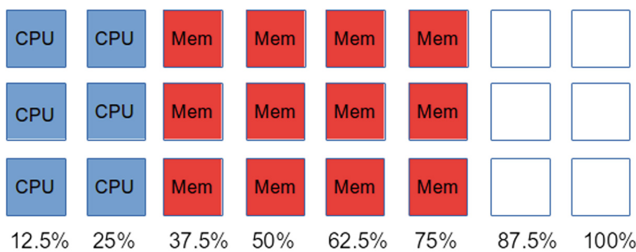


Fig. 3. CPU-bound and memory-bound benchmarks, UL of (25%, 50%).

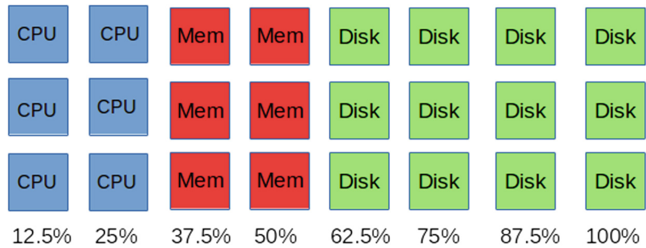


Fig. 4. CPU-, memory-, and disk-bound benchmarks in UL of (25%, 25%, 50%).

Each instance of the memory-bound benchmark is configured to use $(100/N)$ percent of the available memory, where N is the number of processors of the host (100% of the memory is used in full utilization mode). Each instance of the

disk benchmark is configured to use 4 GB of disk size in AMD experiments and 2 GB in Intel experiments. These sizes were chosen taking into account the disk size available in each host. Instances were executed and monitored for 60 s.

Finally, the impact on performance is analyzed. The *makespan* when executing multiple applications at the same time is compared with the makespan of single executions. The reported makespan values are the average computed over 20 independent executions.

4 Experimental Results

This section reports the experimental results of the power consumption and the performance evaluation. The average idle consumption was 183.4 W for the AMD host and 57.0 W for the Intel host.

4.1 Single Benchmark Executions

Figure 5 reports PC and EC values for the CPU-bound benchmark and a graphic comparison of EC values in both hosts. Results show an average difference of 56 W between the EC of the Intel host and the AMD host for all ULs. The almost linear behavior indicates that power consumption is proportional to the UL.

AMD		
UL	*PC*	*EC*
12.5%	194.2±0.6	10.8
25.0%	204.6±0.7	21.2
37.5%	214.6±0.5	31.2
50.0%	224.4±0.9	41.0
62.5%	234.6±1.6	51.2
75.0%	245.2±1.3	61.8
87.5%	253.8±1.2	70.4
100.0%	262.8±2.0	79.4
Intel		
UL	*PC*	*EC*
12.5%	121.7±1.7	64.7
25.0%	136.7±2.0	79.6
37.5%	142.7±2.1	85.6
50.0%	153.0±2.8	96.0
62.5%	163.4±2.7	106.4
75.0%	176.0±1.8	118.9
87.5%	186.0±2.2	129.0
100.0%	195.0±4.1	138.0

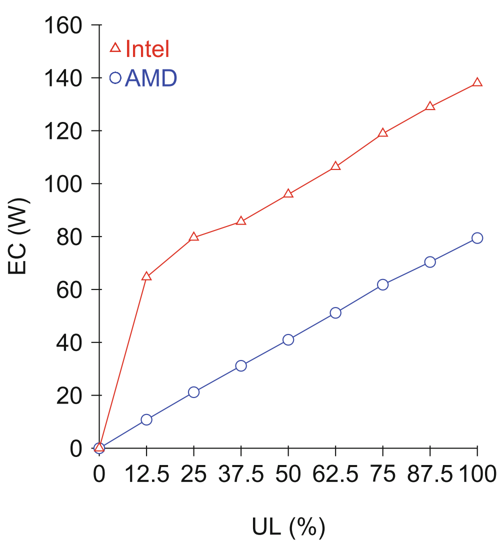


Fig. 5. CPU-bound PC and EC results

Figure 6 reports the PC and EC values for the memory-bound benchmark and a graphic comparison of the EC values in both hosts. Results show a significant increment in EC with regard to CPU-bound executions for all ULs (104% for the AMD host and 36% for the Intel host, on average).

AMD		
UL	*PC*	*EC*
12.5%	215.1±2.0	31.7
25.0%	238.0±1.1	54.6
37.5%	256.8±2.2	73.4
50.0%	271.6±2.9	88.2
62.5%	277.7±6.3	94.3
75.0%	279.0±5.9	95.6
87.5%	290.0±5.6	106.6
100.0%	290.9±13.0	107.6
Intel		
UL	*PC*	*EC*
12.5%	126.8±5.9	69.8
25.0%	158.0±4.3	100.9
37.5%	179.2±4.4	122.2
50.0%	192.0±6.7	135.0
62.5%	213.0±3.9	156.0
75.0%	225.1±5.4	168.1
87.5%	239.4±3.9	182.4
100.0%	249.4±8.9	192.4

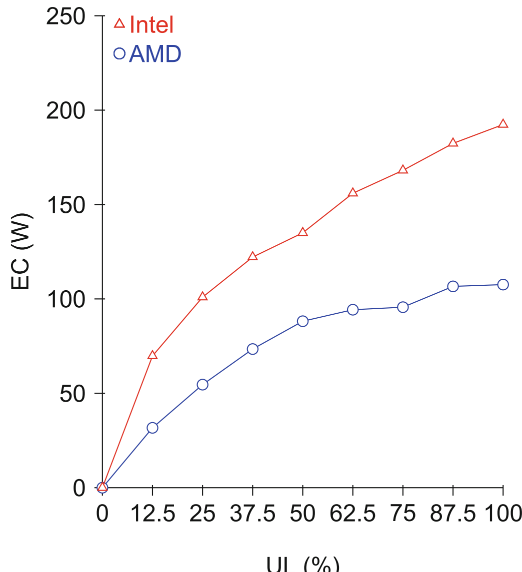


Fig. 6. Memory-bound PC and EC results

AMD		
UL	*PC*	*EC*
12.5%	188.1±0.4	4.7
25.0%	189.3±0.6	5.9
37.5%	189.5±0.6	6.1
50.0%	189.3±0.3	5.9
62.5%	189.8±0.3	6.4
75.0%	190.1±1.5	6.7
87.5%	190.7±1.5	7.3
100.0%	190.4±0.6	7.0
Intel		
UL	*PC*	*EC*
12.5%	67.5±0.9	10.5
25.0%	68.0±0.6	11.0
37.5%	67.7±0.6	10.7
50.0%	68.9±0.7	11.8
62.5%	70.5±0.8	13.5
75.0%	70.7±0.8	13.7
87.5%	71.7±0.6	14.7
100.0%	72.4±0.9	15.4

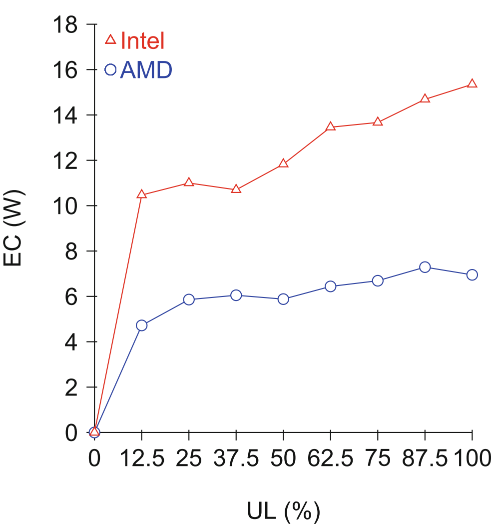


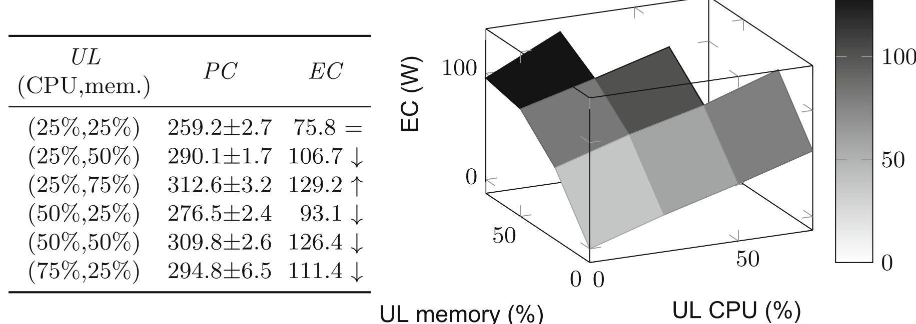
Fig. 7. Disk-bound PC and EC results

A logarithmic behavior is observed for both PC and EC, which does not occur in CPU-bound case, mainly due to the bottleneck in the access to the main memory that reduces the CPU usage. No significant increase is detected on high/critical ULs, possibly by effective resource contention by the operating system for solving conflicts over access to shared resources.

Figure 7 reports PC and EC values for the disk-bound benchmark and a comparison of EC values in both hosts. The maximum EC variation through ULs is 4 W in Intel and 2 W in AMD. These low power variation indicate that disk usage has low impact in power consumption in comparison with CPU and memory, mainly due to waits generated by bottlenecks in disk access.

4.2 Combined Benchmark Executions

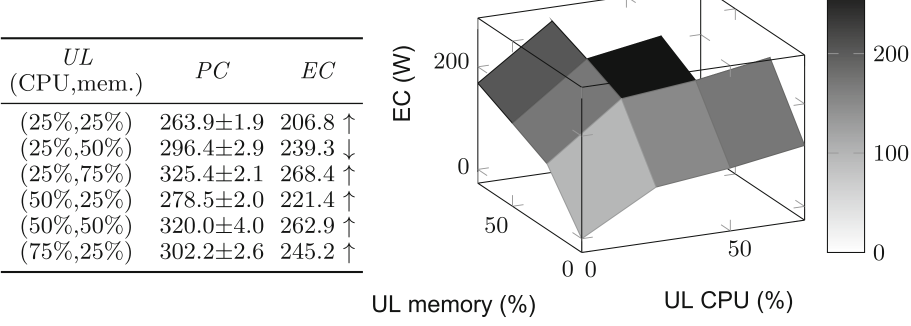
CPU and memory. Figure 8 reports PC and EC when executing CPU- and memory-bound benchmarks together and a comparison of EC values on AMD. Figure 9 reports the same analysis on Intel. Symbol ↑ indicates the EC of the combined benchmarks is higher than the sum of the ECs of each benchmark executed independently, i.e., the combined execution is less efficient than the independent execution. Symbol ↓ indicates the opposite, that is, the combined execution is more efficient. Symbol = indicates that the values are equal (less than 1 W of difference). Results show that for AMD, combined executions reduces EC compared to independent executions. On the contrary, EC of combined executions is higher than independent executions for most cases on Intel.



UL (CPU,mem.)	*PC*	*EC*
(25%,25%)	259.2±2.7	75.8 =
(25%,50%)	290.1±1.7	106.7 ↓
(25%,75%)	312.6±3.2	129.2 ↑
(50%,25%)	276.5±2.4	93.1 ↓
(50%,50%)	309.8±2.6	126.4 ↓
(75%,25%)	294.8±6.5	111.4 ↓

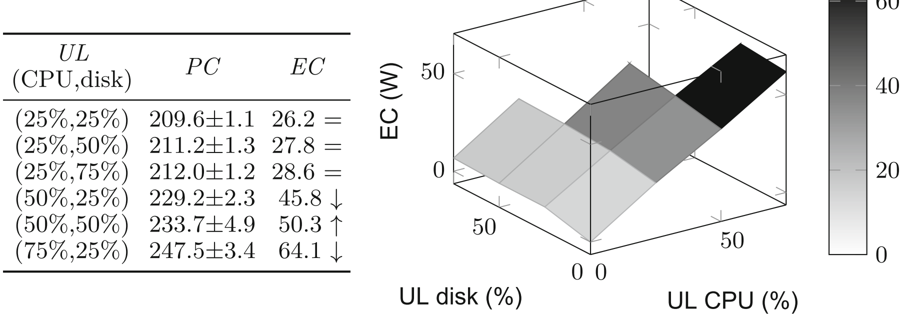
Fig. 8. Combined CPU- and memory-bound PC and EC on AMD

CPU and disk. Figure 10 (AMD) and Fig. 11 (Intel) report PC and EC when executing CPU- and disk-bound benchmarks combined, and the EC graphics on each host. Results show that the combined execution of CPU and disk benchmarks improves energy efficiency for most ULs in both hosts, with regard to EC.



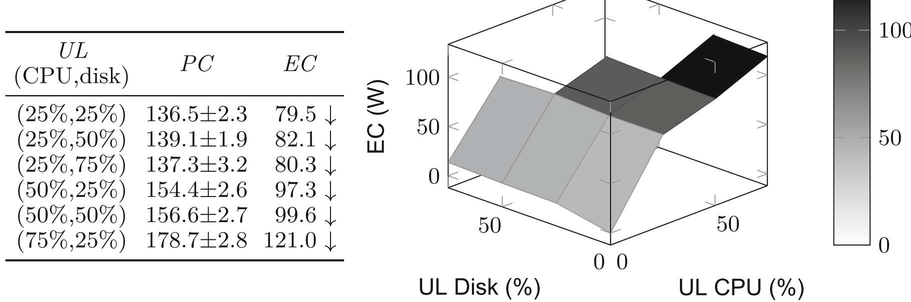
UL (CPU,mem.)	*PC*	*EC*
(25%,25%)	263.9±1.9	206.8 ↑
(25%,50%)	296.4±2.9	239.3 ↓
(25%,75%)	325.4±2.1	268.4 ↑
(50%,25%)	278.5±2.0	221.4 ↑
(50%,50%)	320.0±4.0	262.9 ↑
(75%,25%)	302.2±2.6	245.2 ↑

Fig. 9. Combined CPU- and memory-bound PC and EC on Intel



UL (CPU,disk)	*PC*	*EC*
(25%,25%)	209.6±1.1	26.2 =
(25%,50%)	211.2±1.3	27.8 =
(25%,75%)	212.0±1.2	28.6 =
(50%,25%)	229.2±2.3	45.8 ↓
(50%,50%)	233.7±4.9	50.3 ↑
(75%,25%)	247.5±3.4	64.1 ↓

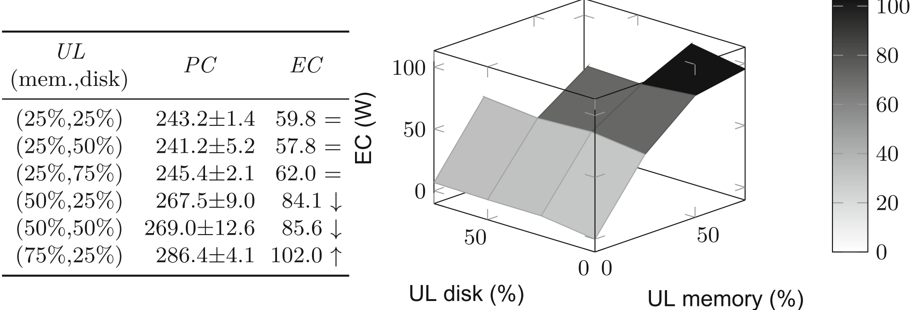
Fig. 10. Combined CPU- and disk-bound PC and EC on AMD



UL (CPU,disk)	*PC*	*EC*
(25%,25%)	136.5±2.3	79.5 ↓
(25%,50%)	139.1±1.9	82.1 ↓
(25%,75%)	137.3±3.2	80.3 ↓
(50%,25%)	154.4±2.6	97.3 ↓
(50%,50%)	156.6±2.7	99.6 ↓
(75%,25%)	178.7±2.8	121.0 ↓

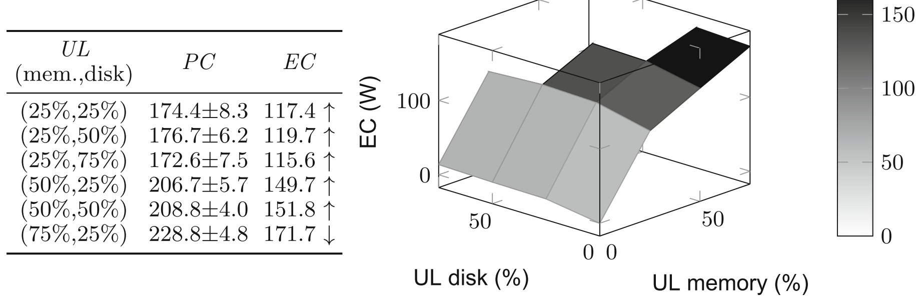
Fig. 11. Combined CPU- and disk-bound PC and EC on Intel

Memory and disk. Figure 12 (AMD) and Fig. 13 (Intel) report PC and EC for memory- and disk-bound benchmarks combined, and the EC graphics on each host. Results show that the combined execution presents higher values of EC than their independent execution, except for low ULs of the AMD host.



UL (mem.,disk)	*PC*	*EC*
(25%,25%)	243.2±1.4	59.8 =
(25%,50%)	241.2±5.2	57.8 =
(25%,75%)	245.4±2.1	62.0 =
(50%,25%)	267.5±9.0	84.1 ↓
(50%,50%)	269.0±12.6	85.6 ↓
(75%,25%)	286.4±4.1	102.0 ↑

Fig. 12. Combined memory- and disk-bound PC and EC on AMD



UL (mem.,disk)	*PC*	*EC*
(25%,25%)	174.4±8.3	117.4 ↑
(25%,50%)	176.7±6.2	119.7 ↑
(25%,75%)	172.6±7.5	115.6 ↑
(50%,25%)	206.7±5.7	149.7 ↑
(50%,50%)	208.8±4.0	151.8 ↑
(75%,25%)	228.8±4.8	171.7 ↓

Fig. 13. Combined memory- and disk-bound PC and EC on Intel

CPU, memory, and disk. Table 1 reports PC and EC of CPU-, memory- and disk-bound benchmarks executed together. Results show that the combined execution on AMD has higher EC compared to their independent execution, mainly at high ULs. However, on Intel, combined executions reduce EC compared to independent executions for all ULs.

Table 1. Combined CPU-, memory- and disk-bound PC and EC

UL	AMD		Intel	
	PC	EC	PC	EC
(25%, 25%, 25%)	265.9±4.3	82.5 =	176.9±3.8	119.9 ↓
(25%, 25%, 50%)	266.6±4.8	83.2 ↓	178.2±3.3	121.6 ↓
(25%, 50%, 25%)	303.0±3.1	119.6 ↑	221.3±5.4	164.3 ↓
(50%, 25%, 25%)	287.9±1.8	104.5 ↑	194.5±1.8	137.5 ↓

4.3 Performance Evaluation

This subsection analyzes the performance evaluation experiments.

Figure 14 reports the makespan of the CPU-bound benchmark and a graphic comparison for both hosts. Results show that increasing the UL does not impact significantly the completion time, due to the absence of resource competition. However, both hosts present a slight degradation for UL 100%, possibly due to conflicts with operating system processes.

AMD	
UL	*makespan*
12.5%	44.4±0.5
25.0%	44.4±0.1
37.5%	44.4±0.2
50.0%	44.4±0.1
62.5%	44.4±0.1
75.0%	44.4±0.1
87.5%	44.5±0.2
100.0%	50.5±5.6
Intel	
UL	*makespan*
12.5%	35.8±7.3
25.0%	30.2±2.9
37.5%	36.2±7.3
50.0%	30.7±2.3
62.5%	29.5±1.2
75.0%	29.8±1.4
87.5%	33.9±5.3
100.0%	47.3±17.8

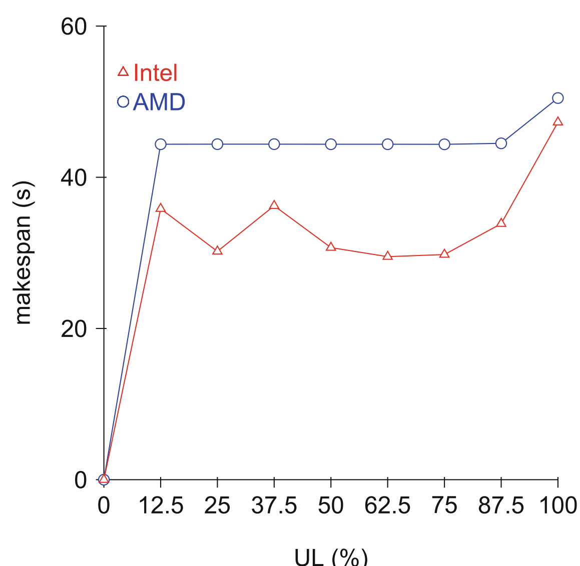


Fig. 14. CPU-bound makespan results

Figure 15 reports the makespan of the memory-bound benchmark and a graphic comparison for both hosts. Performance degrades in AMD; there is a gap of 400 s between the lowest and the highest UL. For Intel the difference is only 48 s. The difference in gaps is possibly due to specific disk features of each host, such as transfer speed.

Figure 16 reports the makespan of the disk-bound benchmark and a graphic comparison for both hosts. The disk-bound case presents a notorious degradation in performance when increasing UL when compared with other benchmarks.

4.4 Energy Efficiency Analysis

This subsection analyzes the energy efficiency from the collected measurements. The energy efficiency metric $\frac{PC \times makespan}{number\ of\ instances \times 3600}$ is defined for comparing results. The lower the metric value, the higher energy efficiency of the host.

UL	*makespan*
AMD	
12.5%	90.3±6.6
25.0%	136.8±21.4
37.5%	179.8±26.8
50.0%	220.7±33.8
62.5%	258.9±39.3
75.0%	326.7±42.6
87.5%	408.2±44.6
100.0%	490.6±58.9
Intel	
12.5%	34.7±6.4
25.0%	26.5±1.4
37.5%	41.0±4.4
50.0%	41.5±1.9
62.5%	67.5±7.2
75.0%	62.1±3.1
87.5%	74.4±5.6
100.0%	82.5±4.1

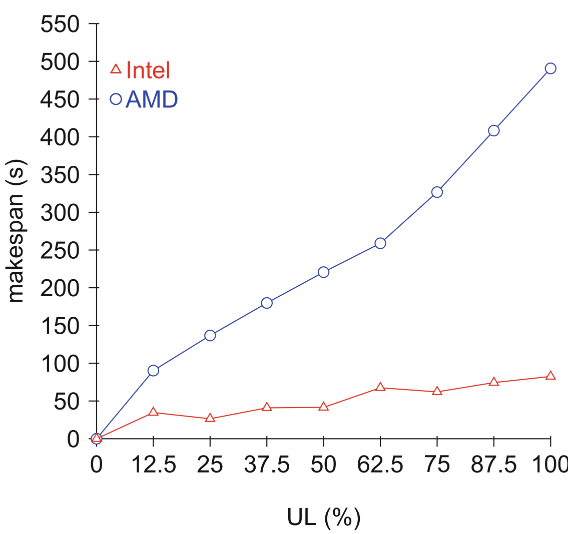


Fig. 15. Memory-bound *makespan* results

AMD	
UL	*makespan*
12.5%	234.8±8.5
25.0%	547.7±30.4
37.5%	839.4±39.5
50.0%	1397.8±144.5
62.5%	1644.0±171.1
75.0%	2006.3±169.6
87.5%	2514.8±306.1
100.0%	2571.3±217.2
Intel	
UL	*makespan*
12.5%	36.1±1.7
25.0%	73.9±1.5
37.5%	114.7±3.6
50.0%	154.6±6.0
62.5%	195.7±7.9
75.0%	238.8±10.5
87.5%	279.0±19.5
100.0%	313.5±42.9

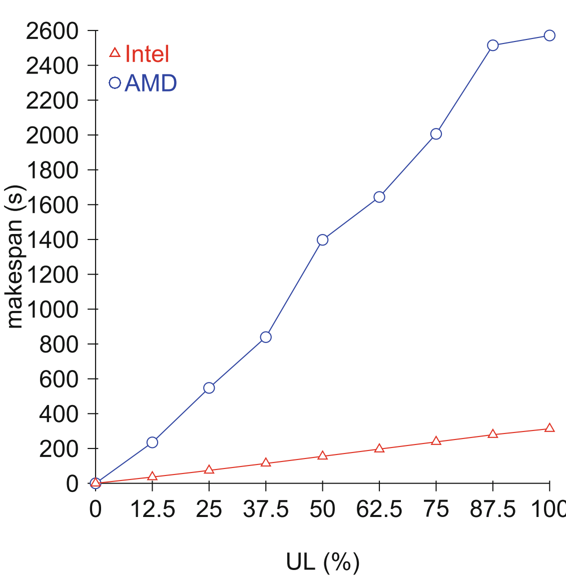


Fig. 16. Disk-bound *makespan* results

Table 2 shows the average energy efficiency of all tests for all ULs and all hosts. Best values for each host are presented in bold. The study shows that for both hosts the CPU-bound benchmark is more efficient at high ULs, memory-bound benchmark is more efficient at medium ULs and disk-bound benchmark is more efficient at low ULs. Intel host is more efficient than AMD for all ULs and all types of benchmarks. Finally, the high-critic UL (100%) is less efficient than the high-medium UL (87.5%), except for disk-bound executions.

Table 2. Efficiency ($PC \times makespan/3600$/number of instances)

UL	AMD			Intel		
	CPU	mem.	Disk	CPU	mem.	Disk
12.5%	0.801	1.804	**4.106**	0.404	0.407	**0.226**
25.0%	0.422	1.511	4.817	0.191	0.193	0.233
37.5%	0.295	1.429	4.926	0.159	0.226	0.240
50.0%	0.231	1.391	6.147	0.109	**0.185**	0.246
62.5%	0.193	**1.335**	5.801	0.089	0.266	0.255
75.0%	0.168	1.410	5.907	**0.081**	0.216	0.260
87.5%	**0.150**	1.570	6.367	0.083	0.235	0.265
100.0%	0.154	1.656	5.686	0.107	0.238	0.262

5 Conclusions and Future Work

This article presented an empirical analysis of the power consumption of synthetic benchmarks in high-end multicore systems. The main contribution is an exhaustive study of the inter-relationship among the power consumption of the main computing resources at different ULs, over AMD and Intel architectures.

The experimental methodology consisted on executing synthetic benchmarks over high-end hosts connected to a PDU, considering different ULs and combinations, aimed at characterizing the power consumption of each computing resource (CPU, memory, and disk). The operations performed by the benchmarks include mathematical functions and read/write of main memory and disk.

The study was complemented with performance experiments. A total number of 144 experiments were performed, 96 evaluating the power consumption and 48 evaluating the performance. For each experiment, 20 independent executions were performed. Results showed that in single executions, CPU utilization has a linear relation with power consumption. Memory utilization has significant impact on power consumption when compared to CPU usage, up to 157% more EC for AMD and 46% more EC for Intel. On the other hand, disk usage presented low EC variation for all ULs.

Combined executions are able to reduce EC with regard to independent executions manly for CPU and disk combined execution. Efficiency analysis showed

that different benchmarks performed more efficiently at different ULs: CPU at high ULs, memory at medium ULs and disk at low ULs. Critic UL (100%) showed worse efficiency than high-medium UL (87.5%), except for disk.

The main lines for future work are related to extend the power and performance characterization of different benchmarks (including GPU-bound, network-bound, and no-synthetic benchmarks) and other high-end hosts. We are also working on using the characterization to build energy models for evaluating energy-aware scheduling strategies on HPC infrastructures and datacenters.

References

1. Anghel, A., Vasilescu, L., Mariani, G., Jongerius, R., Dittmann, G.: An instrumentation approach for hardware-agnostic software characterization. Int. J. Parallel Prog. **44**(5), 924–948 (2016)
2. Brandolese, C., Corbetta, S., Fornaciari, W.: Software energy estimation based on statistical characterization of intermediate compilation code. In: International Symposium on Low Power Electronics and Design, pp. 333–338 (2011)
3. Buyya, R., Vecchiola, C., Selvi, S.: Mastering Cloud Computing: Foundations and Applications Programming. Morgan Kaufmann, San Francisco (2013)
4. Dayarathna, M., Wen, Y., Fan, R.: Data center energy consumption modeling: a survey. IEEE Commun. Surv. Tutorials **18**(1), 732–794 (2016)
5. Du Bois, K., Schaeps, T., Polfliet, S., Ryckbosch, F., Eeckhout, L.: Sweep: evaluating computer system energy efficiency using synthetic workloads. In: 6th International Conference on High Performance and Embedded Architectures and Compilers, pp. 159–166 (2011)
6. Feng, X., Ge, R., Cameron, K.: Power and energy profiling of scientific applications on distributed systems. In: 19th IEEE International Parallel and Distributed Processing Symposium, pp. 34–44 (2005)
7. Iturriaga, S., García, S., Nesmachnow, S.: An empirical study of the robustness of energy-aware schedulers for high performance computing systems under uncertainty. In: Hernández, G., Barrios Hernández, C.J., Díaz, G., García Garino, C., Nesmachnow, S., Pérez-Acle, T., Storti, M., Vázquez, M. (eds.) CARLA 2014. CCIS, vol. 485, pp. 143–157. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45483-1_11
8. Kopytov, A.: Sysbench repository https://github.com/akopytov/sysbench. Accessed 1 May 2017
9. Kurowski, K., Oleksiak, A., Piątek, W., Piontek, T., Przybyszewski, A., Węglarz, J.: Dcworms-a tool for simulation of energy efficiency in distributed computing infrastructures. Simul. Model. Pract. Theory **39**, 135–151 (2013)
10. Langer, A., Totoni, E., Palekar, U.S., Kalé, L.V.: Energy-efficient computing for HPC workloads on heterogeneous manycore chips. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, pp. 11–19 (2015)
11. Nesmachnow, S.: Computación científica de alto desempeño en la Facultad de Ingeniería. Universidad de la República. Revista de la Asociación de Ingenieros del Uruguay, **61**(1), pp. 12–15 (2010). Text in Spanish
12. Nesmachnow, S., Dorronsoro, B., Pecero, J., Bouvry, P.: Energy-aware scheduling on multicore heterogeneous grid computing systems. J. Grid Comput. **11**(4), 653–680 (2013)

13. Nesmachnow, S., Perfumo, C., Goiri, I.: Holistic multiobjective planning of datacenters powered by renewable energy. Cluster Comput. **18**(4), 1379–1397 (2015)
14. Srikantaiah, S., Kansal, A., Zhao, F.: Energy aware consolidation for cloud computing. In: Conference on Power Aware Computing and Systems, vol. 10, pp. 1–5 (2008)

Initial Experiences from TUPAC Supercomputer

David Vinazza[1], Alejandro Otero[1,3], Alejandro Soba[1,4],
and Esteban Mocskos[1,2](✉)

[1] Centro de Simulación Computacional p/Aplic. Tecnológicas, CSC-CONICET,
Godoy Cruz 2390, C1425FQD Buenos Aires, Argentina
{dvinazza,aotero}@csc.conicet.gov.ar

[2] Departamento de Computación, Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires, C1428EGA Buenos Aires, Argentina
emocskos@dc.uba.ar

[3] Facultad de Ingeniería, Universidad de Buenos Aires, Av. Paseo Colón 850,
C1063ACV Buenos Aires, Argentina

[4] Centro Atómico Constituyentes, Comisión Nacional de Energía Atómica,
Av. General Paz 1499, 1650 San Martín, Argentina
soba@cnea.gov.ar

Abstract. High Performance Computing centers boost the development of a wide range of disciplines in Science and Engineering. The installation of public shared facilities in a country is an effort that should be carefully used and must be as open as possible to strengthen the impact of these resources.

We describe the current status and characteristics of TUPAC supercomputer, which is hosted by a CONICET research institute. Unlike other experiences in Argentina, TUPAC is focused on supporting external scientific and technological communities. In spite of having a reduced operations staff, this machine provides computational resources and supports more than 200 external research projects.

In this work we describe the supercomputer setup, tools and policies implemented to reach the level of efficiency needed to support this amount of projects. We also characterize jobs, users and projects with special emphasis on industrial applications and large computational research initiatives hosted in TUPAC.

Keywords: High performance computing facilities
Operations · Monitoring

1 Introduction

Centro de Simulación Computacional para Aplicaciones Tecnológicas (*Computational Simulation Center for Technological Applications*) (CSC) [1] is a recently created research institute with special focus on a broad range of fields of modeling and simulation techniques in Science and Engineering. CSC's professional staff includes professionals with heterogeneous background and specializations: Information Engineering, Advanced Modeling, Energy Generation and Nanoscale

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 38–52, 2018.
https://doi.org/10.1007/978-3-319-73353-1_3

Processes. In addition to attending their own scientific projects, one of the main goals of CSC is providing technical and scientific support to public or private institutions with problems that can be addressed with techniques, processes or methodologies derived from application of advanced modeling and simulation.

Since 2015 CSC hosts TUPAC supercomputer, a 48 TFLOPS machine with 4352 `AMD Opteron` cores distributed among 68 computing nodes. Additionally, 32 `Tesla 2090 Nvidia` boards can be used for GPGPU through a vendor provided configuration which enables four boards per server. Five separate networks support the interconnection of nodes: three separated Ethernet networks for monitoring and administration, and two QDR Infiniband (very low latency) designed for message interchange during computing.

In addition to each node local storage (in total 32 TB), a storage infrastructure is implemented based on an independent optical fiber SAN. This storage is then exported to the rest of the system using two NFS servers in high availability configuration. The raw capacity is 72 TB, which can be configured in different volumes with maximum size of 15 TB. Figure 1 presents a frontal view of this machine: from left to right, the first rack is an Uninterruptible Power System (UPS), the second, fourth and fifth racks contain the computing nodes, the third and sixth are the in-row air conditioning system, and the seventh hosts the storage and monitoring and accessing servers. The air conditioning system is fed by cold water provided by a system installed in the CSC's building.

Fig. 1. TUPAC supercomputer. Seven racks hosts all the support and computing equipment, UPS and in-row water based air conditioning system.

The computing capacity of TUPAC is about 3×10^6 core hours per month. In spite of being considerably lower than other HPC centers in the world, this computing power represents one of the largest HPC capacity in Argentina. Since the beginning, TUPAC's resources are being oriented towards three main actors: in-house users, general scientific community, and industry. As one of the main

objectives of CSC is supporting the development of technology in Argentina, industrial projects have a special priority for resource allocation. The support provided for this kind of projects is not limited to computing resources, but also includes application selection, configuration, adaptation and training in the uses of HPC systems.

Several important projects related to Industry obtained their computing power from TUPAC. Unlike academic initiatives, this kind of projects has to rent the use of this machine, there are differential rates depending of the specific industry: SME's and state-owned companies have preferential discounts. The resources generated by these projects allow the buying of spare parts and hiring of specific technical services to keep the machine in working conditions. The use of TUPAC by industrial users is one of the distinctive characteristics of this machine as no other HPC system in Argentina provides a similar amount of computing resources to this community.

Another distinctive aspect of TUPAC is its external network access. From the beginning, unlike the usual setup found in Argentina, this machine was designed to serve external community. For this reason, network access was configured independently from the rest of the building, having a dedicated fiber based connection with its own firewall. In this way, we avoid any complication from setting up security policies and virtual networks inside the research institutes network. This setup also guarantees accessing TUPAC without competing with the rest of users in the building.

The academic projects are shared between internal members of CSC and the general community, there are no differences in the rules to access the computing power. To access the computational resources hosted by CSC, the interested users have to follow a simple procedure based on an online form. In this form, the users have to detail their computing requirements and the characteristics of the application that will be used. Then, the request is analyzed by a commission composed of technical specialists who evaluate the information and authorize or not the use of these valuable computational resources. If the project present some issues or if the user declares that the objective is the application development or scaling, then a test account is created to allow the user to test and improve their application. An important point is that, for security reasons, all the accounts have a limitation in time. When this period approximates to its ending, the user is informed and a renewal procedure is available to continue the use of the equipment by providing a brief usage report.

The rest of this paper is organized as follows: in Sect. 2, the users and project supported by TUPAC are presented, while in Sect. 3 we introduce some tools and details regarding the operation of this supercomputer. In Sect. 4, we provide statistical information of the usage of this equipment and in Sect. 5 we draw some general conclusions.

2 Projects and Users

Although other computer centers in Argentina were providing computational power to the scientific community, the installation of TUPAC was marked by a

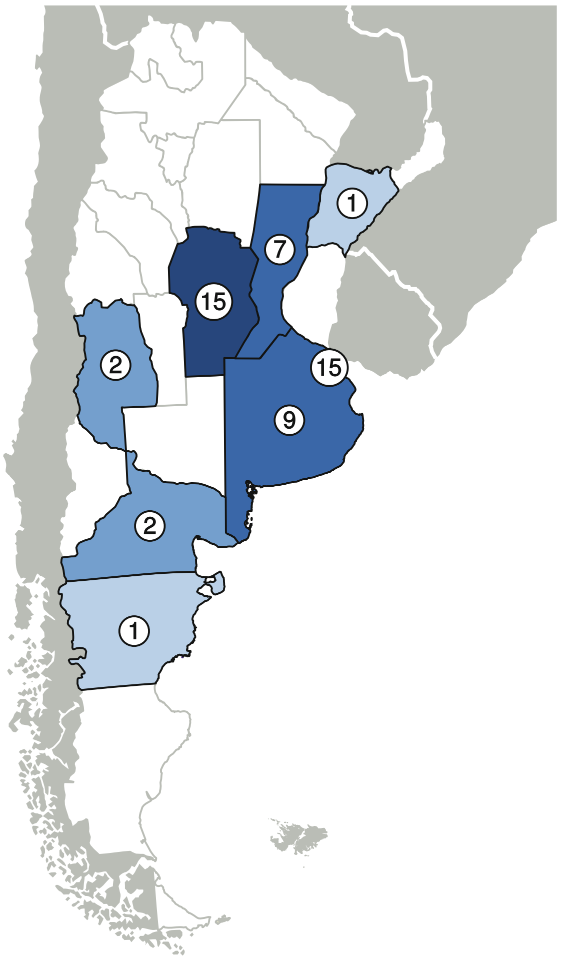


Fig. 2. Geographical distribution of received requests for using HPC resources in the context of IPAC (*Initiative for Accelerated Computational Projects*) initiative.

distinctive characteristic: it was planned to support technological development in Argentina. Unlike other HPC centers in Argentina, external users to the hosting institute should be considered a priority and the access to the computational resources should be guaranteed.

With the objective of increasing the number of users, specially for those who are limited by need of large computational resources in their research work, and in cooperation with the Sistema Nacional de Cómputo de Alto Desempeño (*National System for High Performance Computing*, SNCAD) a completely new initiative was launched: Iniciativa de Proyectos Acelerados de Cálculo (*Initiative for Accelerated Computational Projects*, IPAC), which is a call for projects with high need for HPC resources. A country wide call was sent using the diffusion

network of the national scientific system. IPAC includes resources for two type of projects: (i) Decisive Advances Projects using 1×10^6 h to be hosted in TUPAC, (ii) Computing Projects using between 300×10^3 to 500×10^3 h to be hosted in other centers which are part of the SNCAD.

The received answer for this call surpassed all expectations (Table 1). 52 projects were received in both categories. Figure 2 presents the geographical distribution of the groups who completed their presentations to IPAC call. As is expected, the distribution correlates with the places in which research has more historical support. Buenos Aires leads the list with 24 projects (almost half of total presented projects) followed by Córdoba with 15. These two places concentrate 75% of the total amount of projects.

Table 1. The first call for HPC resources in Argentina surpassed all previous expectations confirming the high need in Argentina.

Type	Requested	Maximum hours per project	Total hours requested
PADS	24	1×10^6	24×10^6
PDC	28	300×10^3 to 500×10^3	12×10^6

SNCAD technical commission evaluated the received projects and generated a ranking according to their different characteristics: justification of resource use, amount of needed storage, efficiency of application, among others. In October 2016, the first SNCAD call assigned three type (i) projects to be run in TUPAC. Additionally, six type (ii) projects were assigned to other centers. The second SNCAD of this type took place on September 2017 and, at the moment of writing, is under evaluation. Next, we briefly describe the characteristics of the projects selected to be hosted in TUPAC:

1. Hydrodynamic cosmological simulations of the formation of galaxies in the Universe (*PADS-UNIVERSE*): granted to a group of researchers of the FCEyN of the UBA. The project is part of the theoretical study of the formation process of galaxies within the paradigm that establishes the cold dark matter model with cosmological constant (ΛCDM). A study that by its characteristics of non-linearity, multi-scale and asymmetry can only be carried out by means of numerical simulations. They use `GADGET3` [12,13] which is parallelized with MPI and OpenMP. The code has good scalability up to 500 processors and uses approximately 1 GB to 2 GB of memory per processor. The volume of information generated with these simulations would reach 3 TB of storage.
2. Computer simulation of structure, electronic and magnetic properties, and reactivity of metal and organometallic species supported on surfaces (*PADS-STRUCTURE*): granted to a group of the Institute of Physics, Rosario. Centered on the search for methods to develop interfaces with specific properties for clean energy production. This project analyzes, from the theoretical point

of view, some chemical reactions on surfaces of materials under study that favors the catalytic production of H_2 from the reforming of Methane and O_2 reduction. For this purpose, they use the Vienna Ab-Initiation Simulation Package (VASP) [6] modeling atomic-scale materials from first principles by calculations of electronic structure and classical molecular dynamics for atomic nuclei. The MPI based parallelization has a fine grain level of control in task assignment which makes it a versatile tool for using in supercomputers. In particular, this project propose to use 1.7 GB of memory per core and up to 1 TB of storage.

3. Simulation of the laminar-turbulent transition in narrow channels with roughness effects and transport of a passive scalar (*PADS-TRANSPORT*): assigned to a group of the CNEA, they propose to approach the study of heat transfer in flows that goes from the laminar regime to the turbulent one by direct numerical simulation of the fluid dynamics. They solve the Navier-Stokes equations with the transport of a passive scalar in rectangular geometries. It is proposed in particular to analyze the thermo-hydraulic performance of the fuel elements of the research reactor RA6, located in Bariloche Atomic Center (Río Negro, Argentina). They propose to use INCOMPACT3D [7,8], a high order tool to solve flows for academic studies parallelized using MPI paradigm. It will use near 0.3 GB of memory per process and 2.4 TB of storage.

2.1 Scientific Projects

In addition to the projects assigned by IPAC call, since December 2015, more than 125 research projects were evaluated by the CSC's technical commission and served by TUPAC and its operations staff. 100 of these comes from the external scientific community.

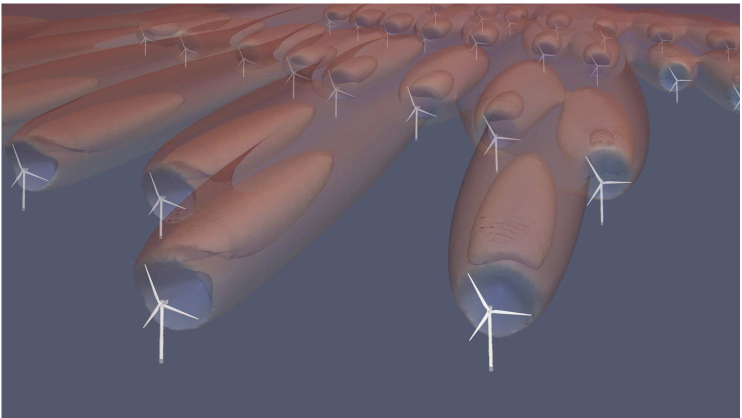


Fig. 3. Visualization of the wake field in a wind farm composed by 43 horizontal axis wind turbines.

We mention some examples next:

- *Theoretical study of organo-metallic self-assembled structure formation in surfaces.* This project is being boarded by a group in Bariloche Atomic Center (Rio Negro, Argentina). The main objective is the analysis and determination of electronic and catalytic properties of these structures, which could potentially lead to new materials.
- *Wind turbine wake interference inside large wind farms.* This project belongs to one of the research lines of CSC. The objective is to study how turbines installed in a large wind farm interact and how the power production is affected by the layout at different wind speed and direction. This will allow to predict wind energy production before construction, besides analyzing other fluid dynamic effects. They use the open source finite volume method software `OpenFOAM` [15] with an in-house implementation of the actuator disks that represent the effect of each turbine [11]. An example of the wake field in a real wind farm in the Argentinean Patagonia is shown in Fig. 3, in which the wake of each turbine is superimposed to the farm layout.

2.2 Industrial Projects

As was previously mentioned, TUPAC supports the development of several engineering projects for industrial applications. As the best of our knowledge, this is the only HPC center offering so large amount of computational resources to this community in Argentina. As an example, we include details about some of them:

- The state-owned company INVAP [2] needed to study a multiphase transient model of emptying a reflector tank. The fluid to be simulated has two phases, water-air and the time of emptying of the tank is analyzed firstly with high granularity mesh to advance later to analyze more demanding models (between 1.4×10^5 and 5×10^6 of elements) that were solved in TUPAC using 1056 cores with ANSYS Fluent software [4]. The model was validated by comparison with experimental measurements in a 1:1 scale model (see Fig. 4).
- Another use example of TUPAC computational resources in industrial projects is the analysis of the cooling rate of the Atucha I vessel. Atucha I is one of three operational nuclear power plants in Argentina. In this case, turbulence models were included in order to model the dissipation and the fluid flow resistance, which could affect the thermal mixture into the fluid. Heat transfers between the solid wall and the cooling fluid are refined using boundary layer with conduction. In Fig. 5, thermal behavior of the fluid in contact with the vessel wall and the flow lines across the entire domain are plotted. ANSYS CFX [4] using volume finite method was executed in TUPAC.

3 Cluster Operations

Keeping any supercomputer serving users demands a large amount of infrastructure and applications to detect hardware and software malfunctioning, support

developing of applications, detect bottlenecks, etc. We detail some of applications that are used in TUPAC to keep it working and serving projects.

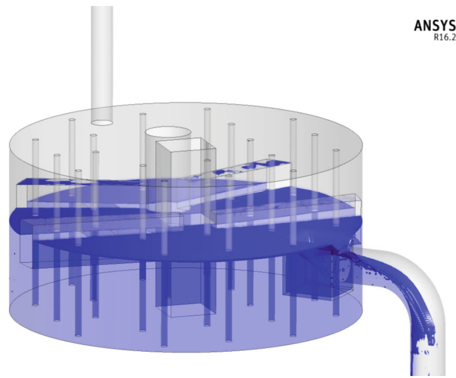


Fig. 4. Simulated Emptying reflector Tank. Case study by INVAP S.E. used 1056 of TUPAC cores to analyze a transient process.

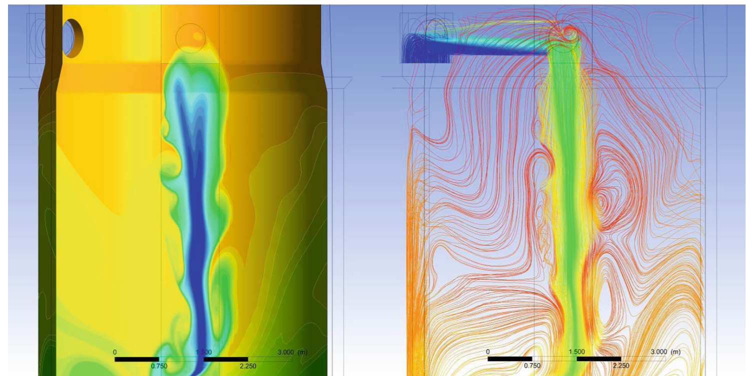


Fig. 5. Atucha I Vessel cooling behavior: thermal transfer and fluid flow lines pictures

3.1 Resource Management

As the procurement procedure of the supercomputer did not include license specifications of the needed software (i.e. if commercial or not), the equipment was installed based on a Redhat Linux distribution. The resource manager, which is a key component in every shared HPC facility was, originally, `Red Hat MRG Grid`, which is a commercial software module derived from a widely used open source grid scheduler and CPU harvesting tool (`HT Condor` [14]).

After an initial unsuccessful installation period, this component was replaced by `SLURM` [10,16], which is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. It was straightforward to configure and deploy, due to its very complete documentation and broad user community, presenting almost no computing overhead. `SLURM` allows the nodes to be assigned to partitions according to its specific hardware components and manages both user/job priorities and quota. It also includes reporting tools and API to obtain additional usage statistics and data using SQL queries.

3.2 Infrastructure Monitoring

An HPC infrastructure, specially in the case of public-funded centers, should provide clear information about its status and usage level. This information could also be consumed by active users to evaluate their own simulations and the efficiency in the use of parallel resources. Additionally, the system state is a key information for the operations staff, as any hardware or software malfunctioning should be found as soon as possible (ideally before users report a failure).

The tools used for these two needs are: `Ganglia` and `Nagios`. `Ganglia` [9] aims to provide information (both historical and current) focused on user level. This tool shows the current resources availability and its usage metrics. The information can be easily accessed by anyone (not only registered users) at TUPAC's site (http://tupac.conicet.gov.ar/ganglia/). For instance, once a job starts, the user can create an aggregate graph of a desired metric (e.g. load average) on job's assigned nodes and follow their evolution in real time. `Ganglia` also provides tools to inspect the created graph on the fly using its `inspect` function. As an example, Fig. 6 presents TUPAC's average load during the last month. Variations come from natural changes in user needs but also from monthly test of electrical emergency generators which produces a total electrical supply shortage each second Saturday of every month.

To get the detailed status of the computational infrastructure, operations staff rely on `Nagios` [5], a widely adopted open source enterprise-grade monitoring solution. `Nagios` covers monitoring most common services like processor load, networking status and disk partitions without the need of complex configuration. In spite of having a range of prebuilt tools and monitoring modules, we built some custom tests using Simple Network Management Protocol (SNMP), Intelligent Platform Management Interface (IPMI) and Nagios Remote Plugin Executor (NRPE) to cover all the details of the infrastructure. Figure 7 shows the services being monitored with `Nagios`, each square represents a different service and its color identifies its state. In this case, `Ping NRPPE` is marked as failed and needs attention from the staff.

Nowadays, based on a strong development of operations staff, all the infrastructure is monitored using `Nagios` including UPS and air conditioning system. Moreover, emergency scripts were configured, installed and tested to allow automatic shutdown when electrical or cooling troubles arise, protecting both hardware and user data sanity.

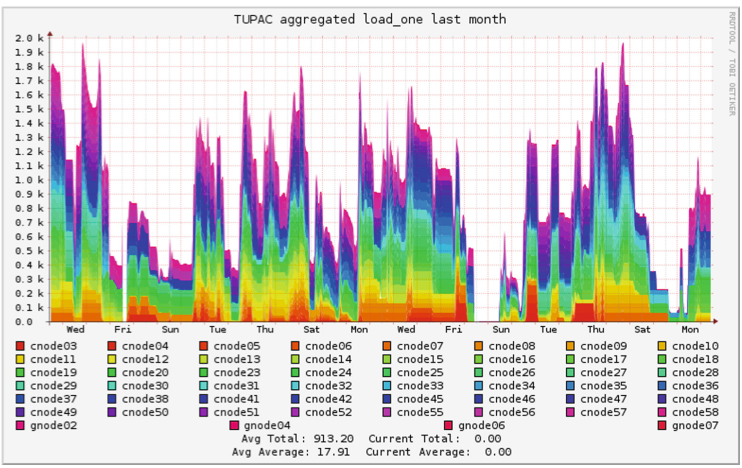


Fig. 6. Ganglia monitoring tool showing the average load during the last month (from 17/6 to 17/7/2017).

Nodos CPU (cnodes)

Host	Services	Actions
cnode01	/ /boot /groups /home /opt /scratch /tmp /var CPU Load IB0 IPMI Fan 1 IPMI Fan 2 IPMI Fan 3 IPMI Log Free IPMI PSU 1 IPMI PSU 2 IPMI Self-test IPMI Sensors Temperature IPMI Sensors Voltage Memory Use Ping Ping BMC Ping NRPE Processes SMART SSH Users Zombie Processes bios ib_fw ks_date munge slurm slurm-node-job-account slurm-node-job-end slurm-node-job-jobid slurm-node-job-partition slurm-node-job-runningtime slurm-node-job-size slurm-node-job-start slurm-node-job-user slurm-node-status	

Fig. 7. `Nagios` monitoring tool is used to control status of a wide variety of node and system properties. Here, a view of node-level checks are shown. (Color figure online)

3.3 User Support

Having an HPC facility oriented to serve a mainly external user community poses an additional pressure for a research institute like CSC, which only accounts with a very limited operations staff. Under such circumstances, an efficient operation is required to comply with users and keep the equipment under good working conditions. To organize the interaction with the TUPAC's increasing number of users, we adopted the use of `Redmine` [3] an online ticket system and integrated knowledge base (i.e. wiki).

Despite `Redmine` is intended mainly for application development, it also fits in an HPC environment. `Redmine` enables the user to quote their problems with text formatting, code syntax and links to other tickets. The operations and support staff can organize the tickets depending of their topic and can delegate them to specialists. Previous tickets also form a knowledge database that can be browsed by the users to help them during the search of solutions for their problems.

4 Usage of TUPAC

In this section, we present some characteristics of users and jobs during past year. The information is obtained directly from SLURM database in base to the registered usage. TUPAC has three differentiated groups of users:

(i) CSC: internal research and operation staff.
(ii) Research: includes all the accounts from external users belonging to the scientific and technological system. The usage of TUPAC for this group with has no limitations and is free of charge.
(iii) Industry: groups all the accounts involved in industrial projects. This users have to contract the usage of TUPAC following a specific CONICET administrative procedure.

Figure 8 presents the grouped use of computational resources by the three identified user groups from 7/2016 to 06/2017. Near 90% of computational resources are fairly shared with external community. Almost 30% of total core hours are destined to engineering projects generated in local companies, while 60% of these hours are consumed by researchers from different research institutes and universities of Argentina completely free of charge.

The evolution of monthly usage is presented in Fig. 9. The variations of industrial projects derive from the natural cycle of start-end of each one. Although TUPAC serves with high priority this kind of projects, the administrative process to start a new project takes some time to be finished preventing the continuous flow of projects. This lag in industrial projects can be seen during September and October, but after these months, a new set of projects started increasing the usage of computational resources. On the other hand, the variations of external scientific projects can not be associated with administrative delays, as all the account creation procedure lasts only a few days. In this case, the user experience differences produced by the cycle "simulate-analyze-refine", in which the researchers need to review obtained results before advancing or modifying their

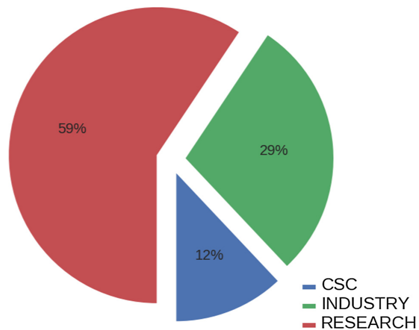


Fig. 8. Consumed CPU time (hours) per group from 7/2016 to 06/2017. Research and Industry include external accounts, while CSC represents the hours used by in-house projects.

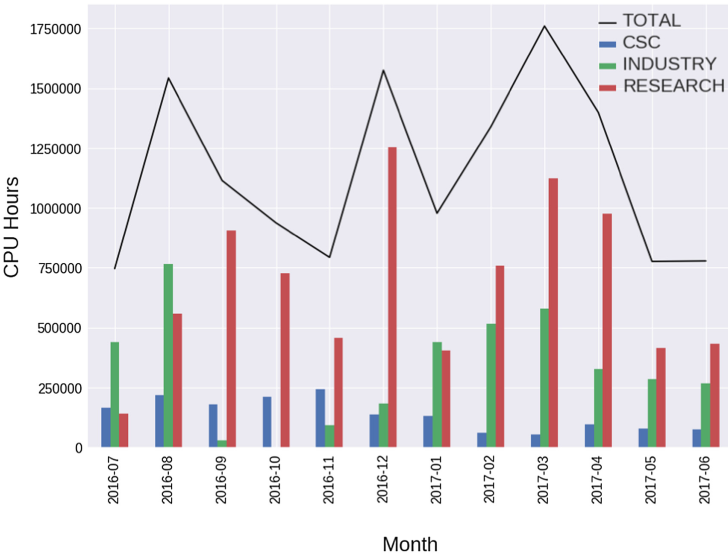


Fig. 9. Monthly evolution of consumed CPU time (hours) per group during last year.

simulation strategies. Finally, the internal demand shows a light decrease after January keeping constant until the end of analyzed period, but in all the months, its impact is lower than the other groups registered usage.

The amount of nodes used in each computational job enables understanding of the need for large scale computing facilities. Specially, to decide if a large shared computing facility could take advantage of having a low latency interconnection between nodes. If most of the jobs uses only one or two nodes, having this special network interconnection renders useless. Figure 10 presents the characterization of parallel usage by each group. As is expected the usage profile for internal and external scientific users is similar, showing near 90% of jobs using single or dual nodes. Thus, only a small fraction of this jobs can take advantage of the low latency network interconnecting the computing nodes. This situation can be related with some applications lacking scalability but also to the size of the system under study. Industrial jobs have a different usage profile, the amount of single or dual decreases to near 50%, while near 40% of the jobs use between 4 and 8 nodes. Moreover, some jobs involve more than 16 nodes which represents more than 25% of available computational resources of TUPAC used for single industrial simulations.

The projects that were selected in the context of IPAC initiative had to proceed through a competitive evaluation. One of the evaluated aspects was the experience of research group in terms of usage of parallel resources and the impact of the scientific problem to be attacked. This poses the question if the usage of TUPAC's resource by these groups of users is different from the rest of the scientific community. Figure 11 reveals that *PADS-UNIVERSE* shows near 86% of jobs use between 3 and 8 nodes, but no job use more than 8, not

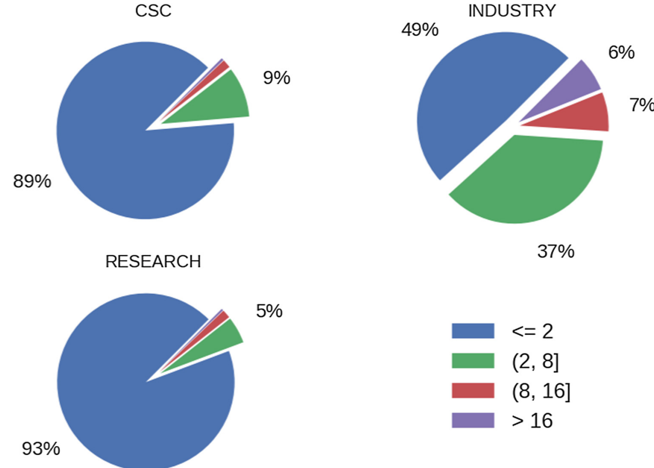


Fig. 10. Number of nodes used in each job per group from 7/2016 to 06/2017.

even a test. *PADS-TRANSPORT* presents higher usage of jobs with single or dual nodes, but still near 70% includes between 3 and 8 nodes. In this case, a few jobs with more nodes are registered which could correspond to a scalability test. Finally, *PADS-STRUCTURE* presents similar usage pattern than standard scientific user, 76% of the jobs fall in one or two nodes, while only 24% uses more than three and less than 9. Also, some test are found corresponding to jobs using larger amount of nodes. The results presented in Fig. 11 supports that selecting

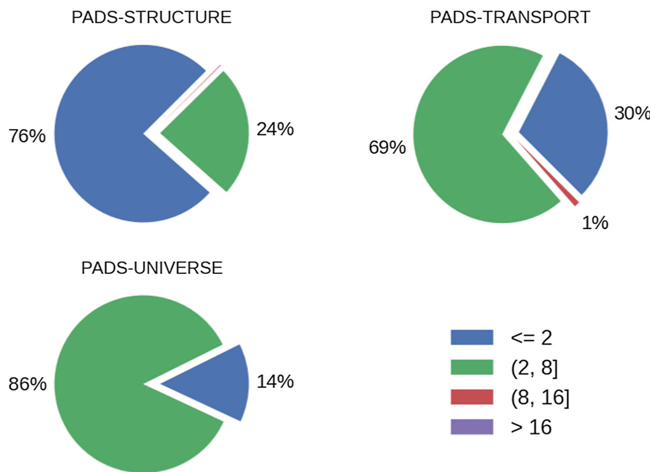


Fig. 11. Number of nodes used in each job per group in IPAC projects from 7/2016 to 06/2017

projects by a competitive process in which research groups have to state their previous experience in the use of HPC resources helps to direct the efforts to those researchers who could take real advantage of this highly valued equipments.

5 Conclusions

High Performance Computing (HPC) techniques have the potentiality of boosting development of both academic and industrial projects. Establishing a HPC center is an effort which involves great amount of resources for buying, installing and maintaining the required equipment. Moreover, additional support is needed to involve the specialized human resources needed to support operations of this kind of facilities.

TUPAC supercomputer is serving scientific and engineering projects since the end of 2015. During this time, despite having a reduced operation staff, more than 200 projects were executed in this machine. Due to a combination of monitoring tools and online communication applications, this group efficiently treats issues and questions from a community of users with very different levels of technical knowledge and projects coming from diverse fields of science and engineering.

The usage profile shows a strong component of projects coming from industrial users. The general scientific community finds in TUPAC a source of computational resources. Although most of execution uses a low number of nodes, additional training and developing of parallel application will evolve the usage profile towards massive parallelism.

Centro de Simulación Computacional p/Aplic Tecnológicas (CSC) is an example of an institute in which research continue to be its main concern, but at the same time can host a machine mainly serving a growing community of external users keeping an acceptable level of quality.

References

1. Centro de simulación computacional p/aplic tecnológicas. http://www.csc-conicet.gob.ar/. Accessed 2 Dec 2017
2. INVAP S.E.: Company devoted to the design and construction of complex technological systems. http://www.invap.com.ar/. Accessed 2 Dec 2017
3. Redmine: A flexible project management web application written using ruby on rails framework. http://www.redmine.org. Accessed 2 Dec 2017
4. ANSYS: Fluent software (2015). http://www.ansys.com/Products/Fluids/ANSYS-Fluent. Accessed 2 Dec 2017
5. Barth, W.: Nagios: System and Network Monitoring. No Starch Press, San Francisco (2006)
6. Hafner, J., Kresse, G., Vogtenhuber, D., Marsman, M.: Vienna Ab initio simulation package (2017). http://www.vasp.at/. Accessed 2 Dec 2017
7. Laizet, S., Lamballais, E.: High-order compact schemes for incompressible flows: a simple and efficient method with quasi-spectral accuracy. J. Comput. Phys. **228**(16), 5989–6015 (2009). http://www.sciencedirect.com/science/article/pii/S0021999109002587

8. Laizet, S., Li, N.: Incompact3d: a powerful tool to tackle turbulence problems with up to o(105) computational cores. Int. J. Numer. Meth. Fluids **67**(11), 1735–1757 (2011)
9. Massie, M., Li, B., Nicholes, B., Vuksan, V., Alexander, R., Buchbinder, J., Costa, F., Dean, A., Josephsen, D., Phaal, P., Pocock, D.: Monitoring with Ganglia, 1st edn. O'Reilly Media Inc., Sebastopol (2012)
10. Pascual, J.A., Navaridas, J., Miguel-Alonso, J.: Effects of topology-aware allocation policies on scheduling performance. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 138–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04633-9_8
11. Sanderse, B., van der Pijl, S., Koren, B.: Review of computational fluid dynamics for wind turbine wake aerodynamics. Wind Energy **14**(7), 799–819 (2011)
12. Springel, V.: The cosmological simulation code GADGET-2. Mon. Not. R. Astron. Soc. **364**(4), 1105–1134 (2005)
13. Springel, V., Yoshida, N., White, S.D.: GADGET: a code for collisionless and gasdynamical cosmological simulations. New Astron. **6**(2), 79–117 (2001). http://www.sciencedirect.com/science/article/pii/S1384107601000422
14. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience: research articles. Concurr. - Pract. Exp. **17**(2–4), 323–356 (2005)
15. The OpenFOAM Foundation: Open source software for computational fluid dynamics (CFD). http://www.openfoam.org. Accessed 2 Dec 2017
16. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple Linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003). https://doi.org/10.1007/10968987_3

HPC Industry and Education

romeoLAB: A High Performance Training Platform for HPC, GPU and DeepLearning

Arnaud Renard(✉), Jean-Matthieu Etancelin, and Michael Krajecki

ROMEO HPC Center and Department of Computer Science,
CReSTIC (Centre de Recherche en STIC) EA3804, University of Reims
Champagne-Ardenne, Moulin de la Housse, 51687 Reims, France
{arnaud.renard,michael.krajecki}@univ-reims.fr,
jean-matthieu.etancelin@univ-pau.fr
https://romeo.univ-reims.fr

Abstract. In this pre-exascale era, we are observing a dramatic increase of the necessity of computer science courses dedicated to parallel programming on heterogeneous architectures. The full hybrid cluster *Romeo* has been used in that purpose since a long time in order to train master students and cluster users. The main issue for trainees is the cost of accessing and exploiting a production facility in a pedagogic context. The use of some specific techniques and software (SSH, workload manager, remote file system, ...) is mandatory without being part of courses prerequisites nor pedagogic objectives. The romeoLAB platform we developed at ROMEO HPC Center is an online interactive pedagogic platform for HPC and GPU technologies courses. Its main purpose is to simplify the process of resources usage in order to focus on the taught subjects. This paper presents the romeoLAB architecture as well as its motivations, usages and future improvements.

Keywords: Programming education · Online education · HPC
GPU · Parallel programming · Web application · Teaching and learning

1 Introduction

The ROMEO HPC Center—France Grand-Est—is a High Performance Computing (HPC) platform hosted by the University of Reims Champagne-Ardenne (URCA) since 2002. Its goal is to deliver high performance computing resources for both local industrial and academic researchers, along with an entire ecosystem of services like secured storage space, innovation in processor architecture, Artificial Intelligence, specific software and support in its usage, as well as an in-depth expertise in scientific application domains. This ecosystem leads to the acquisition of *Romeo* a full hybrid CPU-GPU cluster, in 2013. ROMEO is now recognized as the regional HPC center specialized in GPU, member of the French regional centers group *Equip@meso*, with specific mission related to GPU technology: innovation, research, supporting companies and education.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 55–67, 2018.
https://doi.org/10.1007/978-3-319-73353-1_4

One of the HPC Center missions is to lead some educational activities related to HPC, GPU and hybrid computing. Since 2016, the ROMEO HPC Center is awarded as a GPU Education Center for supporting the educational activities in the specific field of GPU technologies. This label rewards and supports training and master degree courses organized by ROMEO HPC Center through *romeo-LAB* platform.

The presence of a hybrid cluster like *Romeo* is a unique opportunity for URCA students to be trained on a real and large scale HPC facility. Therefore, some HPC and computer science related courses are using the computing resources since a long time, especially in the computer science Master curriculum. From a production usage, there exists a strong need to train our users to the architecture and they are strongly interested and demanding of performances. Training and teaching is an important mission for the ROMEO HPC Center. In 2016, this romeoLAB project was started in this context not only for specific software, architecture and technologies used in HPC but also for general computer science and mathematical contents.

The rest of this paper is organized as follows. In Sect. 2, we present a discussion about related works regarding the motivations and objectives of the romeoLAB platform. Section 3 describes the solution architecture in the context of an HPC cluster. Then, we provide an overview of features and usages. Section 5 presents the practical usages and courses overview. Finally, in Sect. 6, we discuss the limits of the platform and the future works before concluding.

2 Related Work and Motivations

2.1 Online Tools for Specific Code Development

Dakkak *et al.* [1] propose a wide survey of existing solutions for online courses in the field of computer science especially in high performance computing and GPU technologies.

First of all, the *WebGPU* platform [1] is providing a scalable and online environment for GPU programming where code compiling and execution is performed on dedicated resources after web based code edition. This solution implements an advanced control interface for both student and teacher in order to manage courses, discussions and attendees graduating. WebGPU seems quite efficient in terms of access simplification for students but it focuses only on basic GPU technologies.

Some other solution implements this web-based integrated IDE, with providing code edition, compilation and execution as a black box, while targeting academic situations, but without pedagogic approach or related content. SAUCE [4] for *System for AUtomated Code Evaluation* is a free software available on GitHub and allows interactive programming exercises for the teaching of parallel programming: C++ 11 multi-threading, OpenMP, MPI and CUDA. One interesting feature of SAUCE is his compliance with Learning Tools Interoperability (LTI) specification.

The principal concept of LTI is to establish a standard way of integrating rich learning applications and *Tools* with platforms like learning management systems, portals, learning object repositories, or other educational environments that present tools to users. Moodle and Chamilo, the learning management platforms available at URCA, are part of those *Tools*.

In LTI specification, *Tools* are applications and content, often remotely hosted and provided through third-party services by *Tool Providers*: from simple communication applications like chat, to domain-specific learning environments for complex subjects like math, science or Parallel Programming. In other words, if one have an interactive assessment application or virtual chemistry lab, it can be securely connected to an educational platform in a standard way without having to develop and maintain custom integration for each platform. This LTI compliance is an objective for romeoLAB future works.

There are also current projects having a web-based editing interface and performing automated assessment of the written programs. Most of them are embedded within a classical university course structure include WebLab (TU Delft) [15], Jack (University of Duisburg-Essen) [9] (both closed source) and Praktomat (KIT) [5] (open source).

2.2 Online Tools for Educational Purposes

romeoLAB and some other solutions like modern MOOCs (Massive Open Online Courses) are build with development environments (edition, compilation and execution from web-based interface) completely integrated into pedagogical content. This kind of solution offers a level of side functionalities such as code analysis, engineering tools and debugging. Their goal is to provide the simplest interface to focus on pedagogical objectives. Nevertheless, as described later, romeoLAB can be used for dedicated software engineering tools courses such as debuggers and profilers.

Jupyter [7] is an open-source project started in 2014 as *IPython Notebooks*. It provides an interactive interface for data science and scientific computing on top of IPython—an interactive and web-based python shell. Contents are JSON data stored as *.ipynb* files that can be shared, versioned and rendered as various different formats. Notebook's strength is the ability to expose in a single document both rich text contents, documents, images or videos for courses and instructions and interactive code execution. Jupyter is also used by several other integrated solutions such as *Qwiklab* and *CoCalc*.

An integrated solution, *Qwiklab* [12], is used by NVIDIA [10] for self-placed labs and instructor-led sessions on cloud computing resources from Google Cloud Platform and Amazon Web Services. Taking advantage of on-demand resources provisioning, they are able to provide a wide range of architecture to learn about. Another cloud-based fully integrated solution is provided by *CoCalc* project [13], formerly known as *SageMathCloud*. They provide Jupyter Notebooks for many interactive mathematical frameworks and utilities for collaborative development, teaching and authoring. All these cloud-based solutions can take advantage of on-demand provisioning and resources scalability to handle a varying number

of students and users. A complete teaching environment is offered with chat, backups, assignments and grading services.

With a completely different physical resources management and access, the main purpose of romeoLAB is to provide courses on real HPC systems with a dedicated infrastructure that are not provided in the clouds systems.

2.3 romeoLAB Motivations

Learning parallel programming has become increasingly important as parallel processors are now present everywhere, from autonomous vehicle, to smartphone, workstation and supercomputer. This omnipresence is particularly true for GPUs architectures present at ROMEO. A theoretical design of efficient parallel algorithms must be completed by practical experimentations. Not only, it delivers a stronger educational message, but also permits to face some architecture and hardware specific phenomenon like deadlocks, bandwidth bottleneck or cache page issues. Furthermore, as mentioned in Sect. 1, our cluster *Romeo* is equipped with all modern hardware we need for courses. We operate large resources for multiple-nodes labs, and we can rely on our system engineer to make the software available and the whole solution works.

Unfortunately, usual tools for exploiting a high performance computing cluster are rarely part of prerequisite and pedagogic objectives of courses. In a traditional training, students and participants have to use, among others, an ssh client, the cluster workload manager and remote file systems. An introduction part dedicated to these tools is therefore mandatory and time consuming. These are obstacles to the pedagogical process efficiency and consists in the most important motivation for the romeoLAB development. Furthermore, compatibility of all the different client tools and server-side service is always an issue and we want to avoid that lab sessions to become support sessions for student heterogeneous laptop configuration.

As HTML5 (Hypertext Markup Language) is the most universal standard for creating interface and because it makes it possible to create modern and advanced interface available for nearly everyone on earth, we selected web-based technology for this platform. It will provide a simple access to computing resources without needing any client software except a recent web-browser that everyone already has. This aspect is important because ROMEO developed collaborations with universities in Africa (Cheikh-Anta-Diop in Senegal and Virtual University in Tchad).

Our platform is running on a production high performance cluster, in the same single environment for cluster users and platform users. It enables us to propose both general purpose programming courses and advanced HPC and hybrid computing specific training based on an underlying high performance architecture (Infiniband, lustre file system, GPU, ...). We can also provide site dependent training on specific softwares or architectures that we have on ROMEO.

3 A Web-Based Solution in a HPC Cluster

3.1 User View of the Lab Starting Process

Figure 1 presents the user view of accessing an interactive content in romeoLAB. First of all (1), the user creates an account, and log in to the platform https://romeolab.univ-reims.fr. Then (2) user must reach an active *Session* with an access code given by teacher or instructor. The access code can also be provided by an activation link provided by email or on a webpage. At this point, a user can list available labs and their description, to finally start (3) and reach (reach) a *lab*. At this point, the user can work in his own IPython Notebook, watch videos and documents, fill table with performance results, edit, compile and also profile code via a remote desktop.

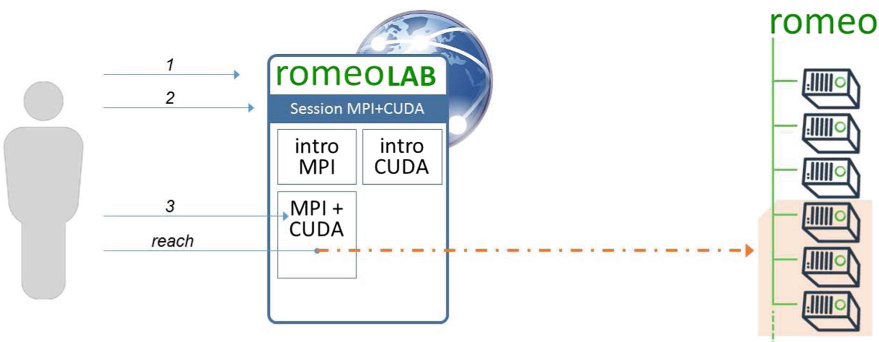


Fig. 1. Starting a lab with romeoLAB: the user view

3.2 Server Part

The romeoLAB web-server himself is written in the PHP programming language. PHP is well suited for creating web service easily: it has a simple syntax, an advanced Object programming model, a lot of plug-ins or libraries and is really specialized for web sites when combined with Apache. We developed our in-house MVC (Model-View-Controler) over RedBean as an object relational mapper (ORM) to access MariaDB database. The whole solution also relies on existing tools, mainly written in Python or NodeJs.

Before starting the lab, the server will assign a temporary cluster-user to the romeoLAB-user and dynamically load initial content of the lab from the lab repository with GIT protocol. This step is represented as (4) on Fig. 2. When starting a *lab*, the platform launches a regular job through the cluster workload scheduler (5) and possibly via reserved dedicated resources. This job consists in setting up all resources parameters and starting all services (6) while romeoLAB is probing their start (7). Once everything is started, the proxy routes are setup

on the romeoLAB gateway—also called *main proxy*—(8) in order to provide a direct access to these services to the user.

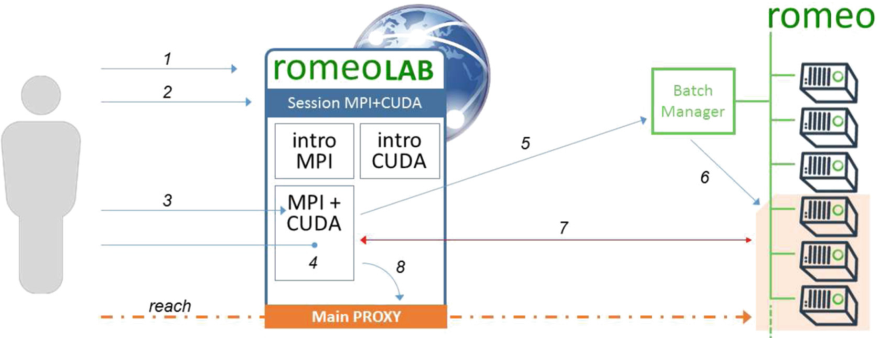


Fig. 2. Starting a lab with romeoLAB: user, server and cluster view

Git versioning system. This platform really interfaces with a *Git versioning system* that enables a decentralized collaborative working environment. In practice, we use the Git service from the ROMEO HPC Center ecosystem, which offers GIT solution for researchers of University of Reims Champagne-Ardenne. romeoLAB allow special users identified as *teachers* to update GIT repository with updated lab content as described in Sect. 4. This solution make it possible to delegate production of courses content to teachers of the university or from external collaborating institution.

Server view of the lab starting process. As the pedagogic platform is running on the production cluster, it must use the resources manager. Communication between cluster, server and user is done by websockets mainly in Python.

Dynamic proxy. The *Node.js* [14] *configurable-http-proxy* is a lightweight proxy which is configurable on runtime through a REST API. The romeoLAB server rely on this module to dynamically manage network urls and ports redirection to let users access the proper services across the cluster network. For each public url provided by the user on classical https port—and prefixed with the platform main url, the main proxy gateway redirects it to the specific compute node—selected by the job scheduler—with a specific and dynamic port—depending ports availability on this compute node.

3.3 Jupyter Resources

We use *Jupyter* as the main part for pedagogic contents. The Jupyter service is running on a compute node and executes *Jupiter Notebooks* with a direct access

to HPC resources. The access is quite simple as the entire API is exposed on a specific network port. This port is the one targeted by the main proxy and is the single entry point for users. Main part of Jupyter is made of Notebooks that gather rich text, video, images and interactive code cells, as shown on Fig. 3. Jupyter service is offering several services such as IPython kernels for notebooks interactivity, standard terminal access and remote file editor. An even richer service is available through the more recent Jupyter-Lab flavor. Finally, Jupyter-Hub [11] is a side project for a multi-user Jupyter platform. This project is not integrated in our infrastructure in order to keep our full customized entry point and a cluster-specific access and resource management.

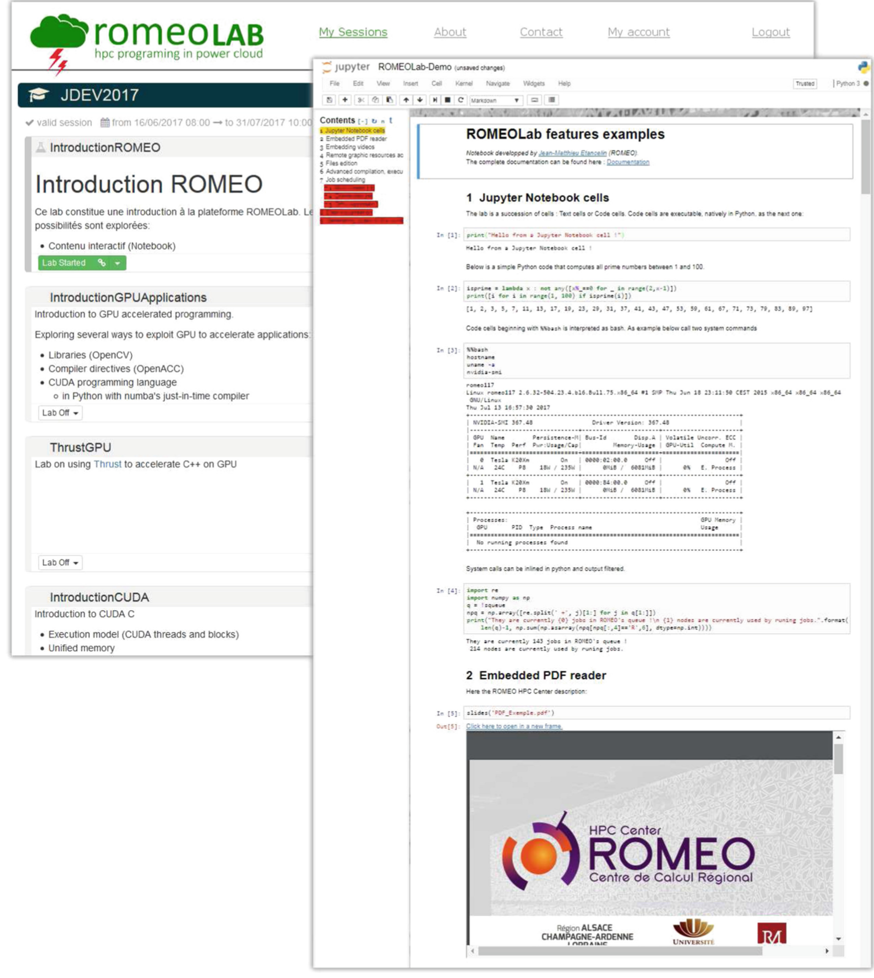


Fig. 3. Jupyter Notebook overview

3.4 Additional Tools

With a tight integration between IPython kernels and web-based interface, we gather several side services into notebook pages. These functionalities are generally provided through Python functions loaded in IPython kernels. This enables us to emphasize the interactivity and usability. Among those improvements, within the notebook, we provide high level Python functions for either side tools access or native Jupyter functionalities:

- displaying pdf files (native),
- displaying html iframes with additional resources like videos (native),
- using a remote file browser and editor, (NodeMirror and native),
- access a Linux Desktop to run graphical softwares (VNC and x11vnc),
- allow file uploading (native),
- executing cells to *bash* (native),
- executing cells through the cluster batch manager (in-house Python module),
- creation and serving *reveal.js* slideshow of the notebook (native).

External tools are described below. As their activation is described in a configuration file specific for each lab, their availability from a single entry-point exposed to the user is done again with the *Node.js configurable-http-proxy* which allow to redirect a specific URL to a specific port inside on the compute node.

NodeMirror. NodeMirror is a server-side *Node.js* file editor that is used in a integrated way within notebooks as html iframes. It enhances the pedagogic effort with close view of both explanations and source code edition with coloration and indentation. This tool is richer than the native Jupyter file editor and is a standalone application that is more suited to be embedded in iframes.

VNC Desktop. For some specific courses, we need to use softwares with graphical user interface such as profilers, scientific data visualization. In this case, we integrate the ability to reach a VNC graphical desktop on the compute nodes through a websockified *noVNC* client. Availability of VNC Desktop is specified in the lab configuration file. The size of the desktop is computed from the size of the user browser screen. The current limitation is that we do not provide graphics acceleration on the compute nodes.

Ipython-batch-execution-magics. A second level of job scheduling was introduced to enables a higher availability of computing resources and allow scalability of romeoLAB. For instance, a course on distributed parallel computing may request two or more computing nodes per student. In practice it would unnecessary lock a large amount of resources to run the notebooks. In that cases, we execute single-core jobs for notebooks and IPython kernel with the ability to execute code samples on large computing resources as another and independent jobs. Thereby, each of the 40 students of a course can run 16 nodes MPI process in their lab with only 32 nodes available in total. This functionality is quite generic and has been distributed as a IPython batch scheduler magics [2].

3.5 Security

As labs are executed on compute nodes, with the same hardware access than classical ROMEO users, security of the platform is an important issue.

First of all, we consider the security of our cluster is already managed (patchs, logs, monitoring, ...), we just have to secure the lab access. For this reason the entire communication between the user and the romeoLAB server is *https* encrypted, from the browser to the proxy, with a valid certificate created by a reliable agency. Https (HyperText Transfer Protocol Secure) also called HTTP over Transport Layer Security (TLS) is the standard to secure communication over internet. It has the advantage to be supported without any modification by all browsers and is open almost on every internet connection. As the cluster environment is under our control, communication inside the cluster is *http* only. Furthermore, nodes dedicated for labs receive a special configuration with home isolation (lab user cannot see other user homes), and network filtering.

To register, a user must provide a valid email which is checked with a confirmation email. The user must authenticate with the password he can manage. Of course, a user can start a lab only if he enters the session specific access-code provided by the teacher. That allow us to limit access to the labs to known people. Of course, multiple unsuccessful login attempt will be detected by the system, and managed with banishment of the client.

As a last security restriction, syndication of the authentication is done across all the services: the web-server, the proxy and the lab. By that way, only the user who is authenticated on the web platform can use the proxy and can start the notebook: we have the certitude that the user loading the notebook is the same who log in to the web platform.

4 Features and Usages

Within romeoLAB we can distinguish two major developments:

- The pedagogic contents which run on compute nodes without any usage restrictions of the hardware. A user can therefore upload any code and execute it. The pedagogic contents are described through the different usages of the platform in Sect. 5.
- The web-portal where a user can log in to manage, start and stop his activities and labs. As each lab is running on compute nodes, we limit extra-exploitation of the resources via restricted and time-limited sessions which are accessible only via a specific token code.

4.1 The Web-Portal Use Cases

Web-portal features are separated in three hierarchical roles: student, teacher and administrator.

As a **student**, the user can reach and leave *Session* in order to start, stop and re-initialize available *labs*. For simplicity we provide to users only the contents they are attending to run. This leads to a specific database layout managed

by teachers. These sessions are tightly linked with the resources manager reservations. Within a *lab* one can edit code samples, interactive code cells, open a Jupyter terminal, access a VNC graphic desktop, generate slides from current notebook, download all the contents and exported notebook. From code cells and terminal, user can load any *modulefile* available on the cluster and run any code or software. User can also launch jobs in the global cluster queue.

Teachers and **instructors** use the platform as a user with two additional features on specific sessions they are privileged. As romeoLAB is a fully integrated platform: pedagogic content development is possible inside the platform. Teachers have read-write access on their labs: they can save current state of each lab data into the Git versioning system in order to update courses. Diff and merge between local notebook and Git reference is also available. Secondly, teachers can administrate their sessions. It enables an instructor to schedule physical resources provision and dedicate it to run labs immediately. Teachers can also observe log information generated by student activity and manage running labs and students.

Finally, the platform **administrator** has all permissions and can manage user's roles and manage labs and sessions. Administrator has also full access to all parameters, like proxy information, system logs or users management.

5 Practical Usages

This platform has been successfully used in various contexts as shown in the Table 1 with a number of participants varying from 17 to 60. All these usages have been performed in an instructor-led course but it also can be used in a self-led format without any restrictions. A qualitative feedback from attendees and teachers is very satisfactory with the high disponibility of resources and the intuitive interface. The few negative comments have been turned into new functionalities.

Table 1. List of romeoLAB usages in chronological order

Name	Attendees	Duration	Courses
2017			
JDEV2017	24	4 days	GPU programming
Groupe calcul	38	3 days	Advanced Python for HPC
Profiler days	17	3 days	Profiling tools for parallel codes
2016			
Master courses	40	4 months	GPU programming and HPC
OpenFOAM school	20	3 days	OpenFOAM software
10th LoOPS day	60	2 days	C++ (HPX) vs Python (DSLs)
GTCEU2016	55	90 min	MPI and OpenACC
GPU spring school	36	1 week	GPU programming
GTC2016	60	90 min	Advanced tools for hybrid cluster

As shown on the Table 2, romeoLAB is addressing a wide range of technologies and levels. The contents of all these sessions came from a list of available labs, that is growing. The idea is to mutualize pedagogic efforts at ROMEO scale, regardless of context usage. We encourage teachers of the University to collaborate on courses through the usage of this platform in order to benefit from others' experiences, cluster hardware and softwares.

Table 2. List of romeoLAB current labs

Title	Level	Technologies and softwares
Introduction to Python	Beginner	Python
Introduction to OpenMP	Beginner	OpenMP
Introduction to MPI	Beginner	MPI
GPU accelerated applications	Beginner	GPU, CUDA, OpenACC, Python
GPU accelerated librairies	Beginner	GPU, cuBLAS, cuRand, cuFFT
Introduction to CUDA	Beginner	GPU, CUDA
OpenACC	Beginner	GPU, OpenACC
OpenFOAM	Intermediate	OpenFOAM
OpenCL	Intermediate	GPU, OpenCL
CUDA asynchronism	Intermediate	GPU, CUDA
Profiling with TAU	Advanced	TAU
Profiling with MAQAO	Advanced	MAQAO
Python-Cython	Advanced	Python, Cython
Python-Numba	Advanced	GPU, Python, Numba
Python-Pythran	Advanced	Python, Pythran
CUDA optimizations	Advanced	GPU, CUDA
Multi-GPU with CUDA	Advanced	GPU, MPI, CUDA
Multi-GPU with OpenACC and MPI	Advanced	GPU, MPI, OpenACC

6 Discussions and Future Work

The platform is currently in production and evolution as we are increasing use cases and pedagogic contexts. In this section, we discuss the interests of future features.

First of all, regarding gathered tools within Notebooks, VNC graphical access shows some lack of performances and usability. Therefore, we plan to enhance it with VirtualGL. For specific labs, we plan to replace it by a solution developed at URCA: USE [8]. It provides a real graphic desktop in a collaborative usage with a low latency technologies and a Windows OS access. The romeoLAB platform is

a complementary to a scientific cloud developed for the *Romeo* cluster. A tight coupling would provide a specific systems environment for running notebooks that can provide specific pedagogic contents.

Then, for new functionalities related to the user experience, we are currently investigating how to introduce some courses management, with interactions between teachers and students. We are strongly interested in a discussion functionality such as live chat, messages, work evaluation or multiple choice tests in order to reinforce pedagogic efficiency in non-teacher-led courses. A progression management by means of intermediate validation and final grading would enhance customized teaching efforts. We also plan to enhance lab list with Artificial Intelligence and Deep Learning courses.

We plan to benefit from dedicated resources usage to provide some basic computer science courses and even more in the field of introduction to programming and parallel programming to undergraduate curriculum. As preliminary step we are currently looking for an integration of the *Snap!* [3] visual programming language. We also plan to make romeoLAB accessible as public portals for scientific software, for reproducibility issues, and with a pay-per-use model.

For security reasons, we choose to separate regular cluster and romeoLAB usages. As we are able to provide courses on specific architecture and software, it becomes interesting to provide an access for regular users to the platform in their own environment (*ie.* in their home directory) in order to let them directly reuse samples codes or examples in their usage of *Romeo*. This functionality would also enable a production access to *Romeo* for managing parallel executions and experiments. For instance, users could manage their jobs in an interactive and persistent way with integrated monitoring, visualization, pre-processing and post-processing.

Finally, a software re-engineering process of the current solution is started to be able to distribute it in reusable parts. For this purpose, Web-design (for ergonomics), security (for proxy authentication) and compatibility (with LTI) are works in progress. Actually only the *IPython magics* for remote execution through a workload scheduler [2] has been distributed.

7 Conclusion

A new platform for interactive courses in computer science, HPC and GPU technologies has been presented. The originality of this solution consists in the usage of a real hybrid cluster, *Romeo*. It enables both *in-situ* and generic practical training on technologies that cannot be easily setup on desktops or mobile devices or rely on hardware support. As our computing center is specialized in leading edge architectures with GPU we started this activity with GPU technologies courses. We are able to provide a wide range of courses and pedagogic resources from basic computer science and programming languages to advanced high performance, hybrid and parallel computing. This platform main purpose is to simplify to a minimal entering point in order to dedicate all training time to specific pedagogic objectives.

The romeoLAB platform is currently in production and has been presented at GTC conference [6]. It is intended to be improved, to receive new applications—teaching purpose and not—and to be distributed to others HPC Centers who showed their interest after public sessions or presentations.

References

1. Dakkak, A., Pearson, C., Hwu, W.M.: WebGPU: a scalable online development platform for GPU programming courses. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 942–949. IEEE (2016)
2. Etancelin, J.-M.: IPython batch scheduler magic (2017). https://github.com/jmetancelin/ipython-batch-scheduler-magic
3. Feng, A., Gardner, M., Feng, W.C.: Parallel programming with pictures is a Snap!. J. Parallel Distrib. Comput. **105**, 150–162 (2017)
4. Hundt, C., Schlarb, M., Schmidt, B.: SAUCE: a web application for interactive teaching and learning of parallel programming. J. Parallel Distrib. Comput. **105**, 163–173 (2017). https://doi.org/10.1016/j.jpdc.2016.12.028
5. Breitner, J., Hecker, M., Snelting, G.: Der Grader Praktomat. Automatisierte Bewertung in der Programmierausbildung
6. Etancelin, J.M., Krajecki, M., Renard, A.: romeoLAB: turn your GPU-supercomputer to an high performance training platform (2017)
7. Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., et al.: Jupyter notebooks - a publishing format for reproducible computational workflows. In: Positioning and Power in Academic Publishing: Players, Agents and Agendas, pp. 87–90 (2016)
8. Lucas, L., Deleau, H., Battin, B., Lehuraux, J.: USE together, a WebRTC-based solution for multi-user presence desktop. In: Luo, Y. (ed.) CDVE 2017. LNCS, vol. 10451, pp. 228–235. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66805-5_29
9. Goedicke, M., Striewe, M., Balz, M.: Computer aided assessments and programming exercises with JACK. ICB Research Reports 28, University Duisburg-Essen, Institute for Computer Science and Business Information Systems (ICB) (2008). http://ideas.repec.org/p/zbw/udeicb/28.html
10. NVIDIA: qwikLABS (2017). https://nvidia.qwiklab.com
11. Project Jupyter: JupyterHub (2017). https://github.com/jupyterhub/jupyterhub
12. qwikLABS (2017). https://qwiklab.com
13. SageMath Inc.: CoCalc Collaborative Computation Online (2016). https://cocalc.com/
14. Surhone, L.M., Tennoe, M.T., Henssonow, S.F.: Node.Js. Betascript Publishing, Mauritius (2010)
15. Delft University of Technology. WebLab, Online Learning Management System (2016). https://weblab.tudelft.nl/

GPU, Multicores, Accelerators

Analysis and Characterization of GPU Benchmarks for Kernel Concurrency Efficiency

Pablo Carvalho[1], Lúcia M. A. Drummond[1], Cristiana Bentes[2](✉), Esteban Clua[1], Edson Cataldo[3], and Leandro A. J. Marzulo[4]

[1] Instituto de Computação, Universidade Federal Fluminense, Niterói, Brazil
pablocarvalho@id.uff.br, {lucia,esteban}@ic.uff.br
[2] Eng. de Sistemas e Computação, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brazil
cris@eng.uerj.br
[3] Programa de Pós-graduação em Engenharia Elétrica e de Telecomunicações, Universidade Federal Fluminense, Niterói, Brazil
ecataldo@im.uff.br
[4] Ciência da Computação, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brazil
leandro@ime.uerj.br

Abstract. Graphical Processing Units (GPUs) became an important platform to general purpose computing, thanks to their high performance and low cost when compared to CPUs. However, programming GPUs requires a different mindset and optimization techniques that take advantage of the peculiarities of the GPU architecture. Moreover, GPUs are rapidly changing, in the sense of including capabilities that can improve performance of general purpose applications, such as support for concurrent execution. Thus, benchmark suites developed to evaluate GPU performance and scalability should take those aspects into account and could be quite different from traditional CPU benchmarks. Nowadays, Rodinia, Parboil and SHOC are the main benchmark suites for evaluating GPUs. This work analyzes these benchmark suites in detail and categorizes their behavior in terms of computation type (integer or float), usage of memory hierarchy, efficiency and hardware occupancy. We also intend to evaluate similarities between the kernels of those suites. This characterization will be useful to disclosure the resource requirements of the kernels of these benchmarks that may affect further concurrent execution.

1 Introduction

Graphics Processing Units (GPUs) have been gaining prominence in general-purpose computing. Their cost/performance ratio combined with their high computational power, have attracted a broad range of users, going from the world fastest supercomputers to the shared virtual infrastructures such as cloud environments.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 71–86, 2018.
https://doi.org/10.1007/978-3-319-73353-1_5

Some benchmark packages have been created over the years to evaluate performance of GPUs in a number of real-world applications. The most important benchmark suites for GPUs are: Rodinia [1], Parboil [2], and SHOC [3]. Unlike benchmarks proposed for CPUs, these benchmarks are composed of a series of kernels, where each kernel represents a task submitted for fine-grain parallelism on the GPU and may have different needs for resources during execution.

The benchmark suites have been used over the years to evaluate aspects of the GPU architecture and its fine-grain parallelism, helping in determining the benefits of new hardware features. In this paper, we are particularly interested in analyzing the individual behavior of the kernels of these benchmarks in terms of resource usage. Our main motivation is the impact that the kernel resource requirements has in further concurrent kernel execution.

Concurrent kernel execution is a relatively recent feature in NVIDIA GPUs. The scheduling policy follows a *left-over* strategy, where the hardware assigns as many resources as possible to one kernel and then assigns the remaining resources to other kernels. In other words, the blocks of one kernel are distributed to the SMs for execution, and if there is space left for more blocks to execute, the blocks of other kernels can execute concurrently. The number of blocks which can execute concurrently on an SM is limited by: (i) the maximum number of active warps per SM imposed by the hardware, (ii) the maximum number of active thread blocks per SM imposed by the hardware, (iii) the number of thread blocks that the shared memory can accommodate given the consumption of each thread block, (iv) the number of thread blocks that the registers can accommodate given the consumption of each thread block. Therefore, a resource-hungry kernel could prevent the concurrent execution of other small kernels. According to Pai *et al.* [4], around 50% of the kernels from the Parboil and Rodinia benchmark suites consume too many resources and prevent concurrent execution of other kernels.

In this scenario, little is known about the behavior of the kernels of these benchmarks in terms of the resources requirements that affect concurrent execution. This work aims at presenting a detailed analysis of the execution of kernels from the three main benchmarks suites, considering the following resource utilization: integer, single and double precision floating point operations, SM efficiency, GPU occupancy and memory operations. Through this analysis, we intend to extend the comprehension of the kernels execution in order to guide further decisions on convenient and more efficient concurrent execution. We also propose to group the kernels with similar characteristics in terms of resource usage. For so, we use the Principal Component Analysis (PCA) statistical method for reducing information dimensionality and K-means clustering for creating the proposed groups.

Our results show that Rodinia and Parboil, although less updated than SHOC, have applications with the highest resource usage. We observed four distinct groups of kernels in the three benchmark suites. The first group concentrates kernels with low resource usage. The second group contains kernels that stress resource usage. The third and fourth groups have kernels with medium

resource utilization and relatively low occupancy. We conclude that the kernels from the third and fourth groups are good candidates for concurrent execution, since they are more likely to leave unused resources.

The rest of the paper is organized as follows. Section 2 presents previous work on GPU benchmark characterization. Section 3 discusses the benchmark suites studied. Section 4 describes the methodology used in our experiments. Section 5 discusses and analyzes our experimental results. Finally, Sect. 6 presents our conclusions and directions for future work.

2 Related Work

There are few previous effort in benchmark suite characterization for GPUs. The work by Che *et al.* [5] characterizes Rodinia applications in terms of instructions per cycle (IPC), memory instruction mix, and warp divergence. Their focus, however, is to compare with the CPU benchmark Parsec. Kerr *et al.* [6] analyze GPU applications from NVIDIA SDK and Parboil in terms of control flow, data flow, parallelism and memory behavior. This work, however, performs the study on a GPU emulator that execute the PTX code of the kernels. The work of Goswami *et al.* [7] presents a set of microarchitecture independent GPU workload characteristics that are capable of capturing five important behaviors: kernel stress, kernel characteristics, divergence characteristics, instruction mix, and coalescing characteristics. They studied Rodinia, Parboil and CUDA SDK, but their study is based on a simulation of a generic GPU, that they have heavily instrumented to extract the studied characteristics.

Burtscher *et al.* [8] study a suite of 13 irregular benchmarks in terms of control-flow irregularity and memory-access irregularity, and compare with regular kernels from CUDA SDK. In a later work, O'Neil and Burtscher [9] presents a microarchitectural workload characterization of five irregular GPU applications from the LonestarGPU benchmark suite[1]. Their study is based on simulations and they also focus on the impact of control flow and memory access irregularity. Bakhoda *et al.* [10] characterizes 12 non-graphics kernels on a GPU simulator. They study the performance impact of some microarchitecture design choices: interconnect topology, use of caches, design of memory controller, parallel workload distribution mechanisms, and memory request coalescing hardware.

Our study, on the other hand, is the first to perform the characterization on the kernels of the three main GPU benchmark suites, rather on the applications as a whole. Our analysis is based on actual executions of the benchmarks on recent GPU Maxwell architecture and rises important characteristics for further understanding concurrent kernel executions.

3 Benchmark Suites

Benchmarks are programs designed to evaluate computational systems. By analyzing the execution of these programs through well-defined metrics, it is possible to learn about the operation of a specific architecture, identify bottlenecks

[1] http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu.

in a program execution and compare different architectures [11]. Nevertheless, it is essential to choose benchmark suites that are able to stress the resource usage of the hardware being evaluated. In [12], authors recommend the use of algorithmic methods that capture a pattern of computation and communication (called "Dwarfs") to design and evaluate parallel programming models and architectures. Moreover, they recommend the use of thirteen Dwarfs instead of traditional benchmarks.

The focus of this paper is to evaluate the benchmarks developed for general purpose computing. The analyzed benchmarks, Rodinia [1,5], Parboil [2] and SHOC [3,13], provide implementations for other processors. However, our goal here is to study their behavior in the context of GPUs.

The Rodinia benchmark package, released in 2009 (now in version 3.1), focuses on the analysis of heterogeneous systems. Rodinia offers 23 applications with CUDA, OpenCL and OpenMP implementations, covering nine of the thirteen Dwarfs.

The Parboil package was developed in 2008 to test and demonstrate the capability of the first generation of GPUs with CUDA technology. According to the concept of its development, the package's composition was designed neither to deliver fully optimized and low-level versions for a particular device, nor to deliver full versions of applications that would discourage modifications. Currently, Parboil is composed of 11 applications, covering a subset of the thirteen Dwarfs. Most of these applications are only implemented in CUDA, while some have a basic CPU implementation. Therefore, this benchmark suite does not seem appropriate for comparing CPUs with GPUs nor evaluating hybrid (GPU+CPU) systems.

Scalable Heterogeneous Computing (SHOC) benchmark suite was designed for GPUs and multi-core processors. Moreover, it provides MPI+CUDA versions that allow the execution using multiple GPUs in a cluster. SHOC applications are organized in three levels: (*i*) level 0 has 6 applications that measure low-level hardware characteristics, such as memory bandwidth and peak FLOPS; (*ii*) level 1 provides 10 applications that correspond to a subset of the thirteen Dwarfs; and (*iii*) level 2 consists of 2 real applications.

4 Methodology

Our kernel characterization targets on their behavior on integer and floating-point operations, SM efficiency, GPU occupancy and memory operations. We extracted these metrics from the NVIDIA *nvprof* tool [14] as seen in Table 1.

Although *nvprof* returns the maximum, average and minimum values, for each metric, we used only the average values, since the variance is not high. From the extracted data, the sum of *shared_load_transactions* and *shared_store_transactions* provides the total transactions on shared memory. The sum of *local_load_transations* and *local_store_transactions* provides the total local transations. The sum of *gld_transactions* and *gsd_transactions* provides total transactions on global memory.

Table 1. Selected metrics and characteristics

Metric	Description
sm_efficiency	Time percentage that at least one *warp* is active in a SM in relation to all the GPU SMs
achieved_occupancy	Active *warps* rate in a single SM in relation to the maximum number of active *warps* supported by the SM
shared_load_transactions	Number of loading operations at the shared memory
shared_store_transactions	Number of writing operations at the shared memory
local_load_transactions	Number of loading operations at the local memory
local_store_transactions	Number of writing operations at the local memory
gld_transactions	Number of reading operations at the global memory
gst_transactions	Number of storage operations at the global memory
inst_fp_32	Number of single precision floating point operations
inst_fp_64	Number of double precision floating point operations
inst_integer	Number of integer operations

The analysis of the measured data follows past work in benchmark characterization [5,7,15–17] in using Principal Component Analysis (PCA) and clustering. The data is first mean-centered and normalized to make it comparable. After that, we used the PCA to reduce the data dimensionality and show the characteristics that contribute most to its variance. PCA returns a number of principal components, that are linear combinations of the original features. The first principal component (PC1) exhibits the largest variance, followed by the second principal component (PC2).

With the results of PC1 vs PC2, we group similar kernels using the K-means grouping technique. The values of K used in the K-means clustering were obtained experimentally for each analysis.

5 Experimental Results

5.1 Experimental Environment

Our experiments were conducted on a GPU GTX 980 (Maxwell architecture) with 2048 CUDA cores running at 1126 MHz in 16 SMs, with 4 GB of global memory and 2 MB of L2 cache. Each SM has 96 KB of shared memory and 256 KB of registers. To compile and run the benchmarks we used CUDA version 7.5. The statistical analysis was performed using the R language. All applications were executed with the standard input data sets.

5.2 Individual Analysis

We first analyze kernels belonging to each benchmark suite separately. In order to distinguish the kernels in the presented charts, without compromising visibility, we used a coding scheme where each kernel is identified by a letter that

corresponds to the application it belongs, and a number that distinguishes it from the other kernels of the same application.

Parboil. For the Parboil analysis, we did not use double precision floating-point operations in the metrics. This is the oldest benchmark suite from the three studied, launched before GPUs had double precision support. Parboil has 11 applications with 26 kernels. We first detect what we call *low-expressive* kernels. These kernels use a small amount of resources and run in a very short time. Thus, they were removed from our analysis. Consider that P_K is the percentage of total execution time required to run kernel K in some application and that M is the mean of the execution times for all kernels of that same application. We will remove K from the analysis if P_K is one standard deviation bellow M. Only 3 kernels were removed.

Parboil applications do not make use of concurrent execution. All kernels are executed on the same stream.

Figure 1 shows the biplot chart for the results of PC1 vs PC2 followed by the K-means clustering ($K = 5$). Each point in the chart represents a kernel (named according to our coding scheme). Arrows denote vectors that correspond to the metrics analyzed. Vector lengths are proportional to the magnitude of each metric, and the angle between two vectors represent the correlation between the corresponding metrics. Vectors with a small angle indicate that metrics are highly correlated. Vectors forming a 90° angle are uncorrelated, and vectors in opposite directions have negative correlation. The proximity of points to the vectors shows kernels with the greater values obtained in that metric.

The direction of vectors in the chart indicates a correlation between the number of shared memory transactions with the number of floating-point operations. This means that Parboil kernels normally use shared memory to store floating point data. We have also noticed a correlation between global memory transactions and SM efficiency. This means that the applications usually have a good number of warps to hide global memory latency, when global memory is intensively used.

The five groups identified by K-means are represented in different colors in Fig. 1 and specified in Table 2. Group 1 is composed of 10 kernels. This group contains Parboil's most representative kernels. These are compute-intensive kernels with high efficiency, a large number of operations with integers and single-precision floating-point, and a relatively high occupancy.

Group 2 contains 2 kernels with high efficiency, but low occupancy. In these kernels, the large number of integer and floating-point operations are enough to maintain the SM busy, and there is no need for more warps to hide latency.

Group 3 has 6 kernels. These kernels are characterized by the average resource usage. The position of its kernels in the chart suggests that the metrics present average values when compared to the more resource-consuming groups (groups 1 and 2) and the less resource-consuming groups (groups 4 and 5).

Group 4 has 3 small kernels from the *mri-gridding* application. These kernels are the less representative in terms of execution time.

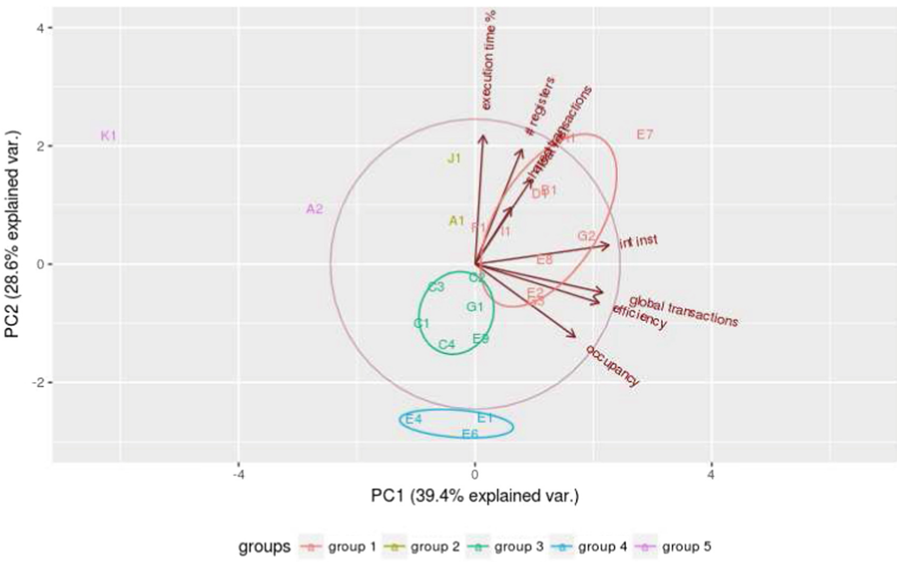


Fig. 1. Parboil results

Table 2. Parboil kernels for each group.

Group	Application	# kernels	Kernels
1	cutcp	1	cuda_cutoff_potential_lattice6overlap(B1)
	lbm	1	performStreamCollide_kernel(D1)
	mri-gridding	3	splitRearrange(E2), splitSort(E8), gridding_GPU(E7)
	mri-q	1	ComputeQ_GPU(F1)
	sad	2	mb_sad_calc(G2), larger_sad_calc_8(G3)
	spmv	1	spmv_jds(I1)
	sgemm	1	mysgemmNT(H1)
2	bfs	1	BFS_kernel_multi_blk_inGPU(A1)
	stencil	1	block2D_hybrid_coarsen_x(J1)
3	histo	4	histo_prescan_kernel(C1), histo_main_kernel(C2), histo_final_kernel(C3), histo_intermediates_kernel(C4)
	mri-gridding	1	scan_L1_kernel(E9)
	sad	1	larger_sad_calc_16(G1)
4	mri-gridding	3	reorder_kernel(E1), uniformAdd(E4), binning_kernel(E6)
5	bfs	1	BFS_in_GPU_kernel(A2)
	tpacf	1	gen_hists(K1)

Group 5 contains only 2 kernels. These are the kernels with the smallest occupancy, which make them good candidates for concurrent execution with other applications kernels. During their execution there is a higher probability of having unused resources that could be allocated to a concurrent application.

Rodinia. In Rodinia, 58 kernels from 22 applications were analyzed[2]. In the detection of *low-expressive* kernels, 14 kernels were removed.

Rodinia applications do not make use of concurrent execution. Except for the Huffman application included in version 3.0 (2015), all other Rodinia applications execute their kernels on the same stream.

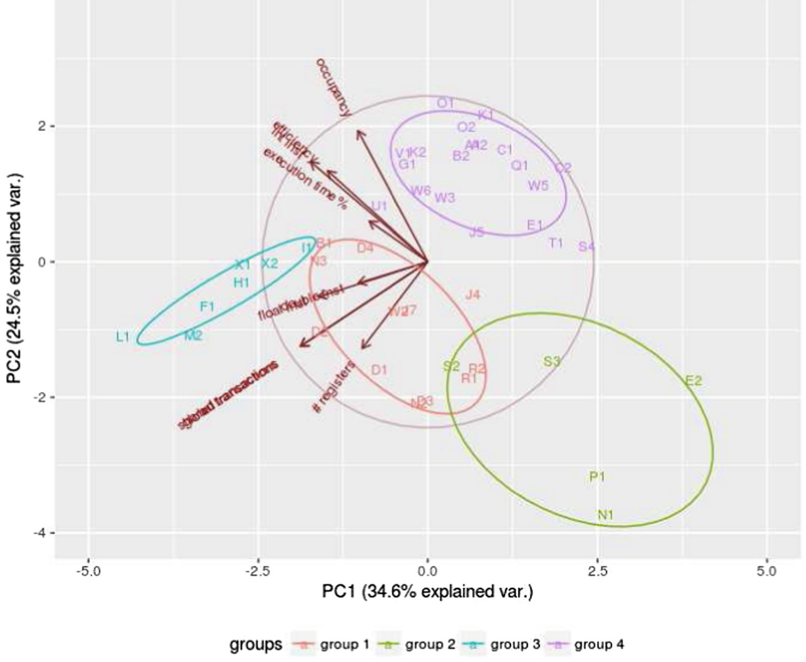


Fig. 2. Rodinia results

Figure 2 shows the biplot chart for the results of PC1 vs PC2 followed by the K-means clustering ($K = 4$). We can observe in this chart that most of the analyzed kernels that make transactions in global memory, also make transactions in shared memory, indicating that Rodinia applications are mostly optimized to take advantage of shared memory. We can also observe a high correlation between single and double precision floating-point operations, although double

[2] We did not analyze the CFD application, since nvprof was not able to correctly extract the corresponding metrics.

precision operations are less pronounced in this benchmark suite. In addition, the correlation between integer operations, SM efficiency, and occupancy indicates that integer-based applications are compute-intensive and have a great number of warps. This maintains the SM busy most of the time.

Four groups were identified and are shown in Table 3. Group 1 has memory-intensive kernels and consists of 14 kernels from 8 applications. Although the most remarkable characteristic of kernels in this group is the amount of memory operations, there are other interesting characteristics: (i) the group does not perform double-precision floating-point operations, (ii) it has variable occupancy while its efficiency is high for the majority of the members, and (iii) most of the kernels perform integer operations.

Group 2 contains 5 kernels. These kernels presented low resource usage and short execution time. All five kernels have little relevance in terms of resource usage, compared to the other groups, and are probably not recommended for assessing the GPU architecture and its parallelism. It is important to notice that all the kernels of the *Myocite* application are in this group.

Group 3 has 6 kernels. This group is characterized by intensive double-precision floating point operations, although it also performs a large number of single precision floating-point operations. These kernels are floating-point based compute intensive.

Group 4 is the one with more kernels (19). The group is characterized by high GPU occupancy and by the large number of integer operations. We can observe in the chart that kernels of this group are plotted in the opposite direction of the vectors corresponding to the memory operations metrics. This means that these kernels are not memory intensive.

SHOC. Compared to the other benchmark suites, SHOC is the suite that received updates more recently. For this reason, the applications S3D and Triad make massive use of concurrent execution, using multiple streams. SHOC contains not only real applications, but also some microbenchmarks. The SHOC level zero applications contains only microbenchmark kernels to test low-level details of the hardware. We did not include these applications in our study. Our analysis focused on 47 kernels of level one and 59 kernels of level two applications. In SHOC, we did not remove *low-expressive* kernels, since level one applications are much smaller than level two applications.

Figure 3 shows the biplot chart for the results of PC1 vs PC2 followed by the K-means clustering ($K = 3$). The chart presents a strong correlation between the percentage of the execution time and the number of operations in shared memory, which means that kernels that take more time to execute are optimized to take advantage of shared memory. Operations in global memory are highly correlated with SM efficiency, which means that kernels that need to access the global memory are able to efficiently hide memory access latency. There is also a certain proximity between operations with integers and with single precision floating point, integers are probably used for controlling loops that contain floating point operations.

Table 3. Rodinia kernels for each group.

Group	Application	# kernels	Kernels
1	backprop	1	bpnn_layerforward_CUDA(B1)
	dwt2d	4	dwt_cuda::fdwt97Kernel<int=128, int=6>(D1), dwt_cuda::fdwt97Kernel<int=192, int=8>(D2), dwt_cuda::fdwt97Kernel<int=64, int=6>(D3), c_CopySrcToComponents<float>(D4)
	huffman	1	vlc_encode_kernel_sm64huff(I1)
	hybridsort	2	bucketsort(J4) bucketcount(J7)
	lud	2	lud_perimeter(N2), lud_internal(N3)
	nw	2	needle_cuda_shared_1(R1), needle_cuda_shared_2(R2)
	pathfinder	1	dynproc_kernel(U1)
	srad-v1	1	reduce(W2)
2	gaussian	1	Fan1(E2)
	lud	1	lud_diagonal(N1)
	myocyte_10	1	kernel(P1)
	particlefilter-float	2	likelihood_kernel(S2), normalize_weights_kernel(S3)
3	heartwall	1	kernel(F1)
	hotspot	1	calculate_temp(H1)
	lavaMD	1	kernel_gpu_cuda(L1)
	leukocyte	1	IMGVF_kernel(M2)
	srad-v2	2	srad_cuda_1(X1), srad_cuda_2(X2)
4	b+tree	2	findRangeK(A1), findK(A2)
	backprop	1	bpnn_adjust_weights_cuda(B2)
	bfs	2	Kernel(C1), Kernel2(C2)
	gaussian	1	Fan2(E1)
	hotspot3d	1	hotspotOpt1(G1)
	hybridsort	1	mergeSortPass(J5)
	kmeans	2	invert_mapping(K1), kmeansPoint(K2)
	mummergpu	2	mummergpuKernel(O1), printKernel(O2)
	nn	1	euclid(Q1)
	particlefilter-float	1	find_index_kernel(S4)
	particlefilter-naive	1	kernel(T1)
	sc_gpu	1	kernel_compute_cost(V1)
	srad-v1	3	srad2(W3), prepare(W5), srad(W6)

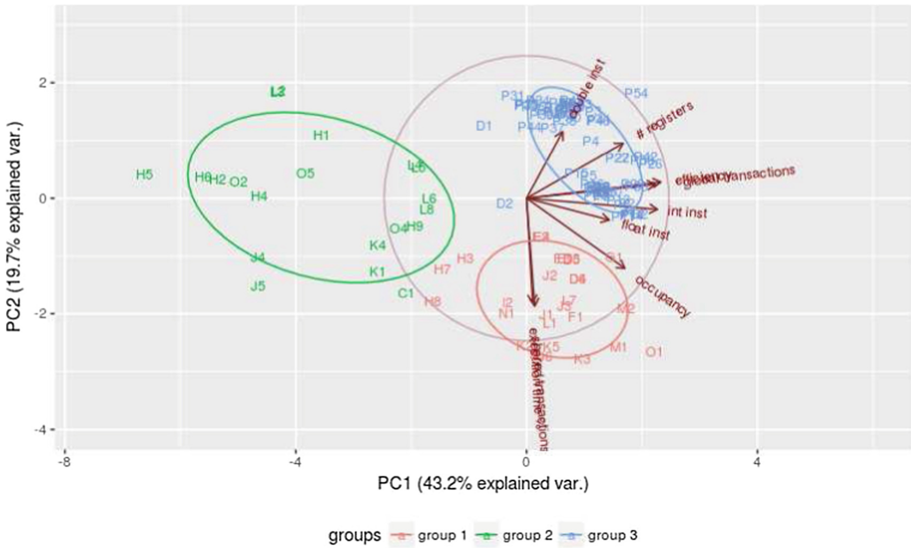


Fig. 3. SHOC results

Three groups of kernels were identified as show in Table 4. Group 1 is composed of 28 memory-intensive kernels that present the higher execution time. This group is mostly composed of level one parallel algorithms, and is the most diverse group. Kernels in this group use more operations with integers than the others.

Group 2 contains 20 kernels that have low significance. The chart shows that kernels of this group are positioned in the opposite direction of the vectors of the metrics, indicating that kernels in this group do not consume much of the analyzed resources.

Group 3 contains 58 kernels mostly from the S3D application. For these kernels, we observed short execution times and low occupancy. These characteristics confirm the capability of S3D to massively exploit concurrent execution. Kernels warps do not occupy the whole SM, and the underutilized resources can be allocated to other kernels. Most of the kernels do not take advantage of shared memory, but provide high efficiency. We can also observe two subgroups among the S3D kernels, one that performs a high number of double-precision floating-point operations, while the other subgroup does not.

5.3 Global Analysis

In this analysis, we assembled all kernels of the benchmark suites, with a total number of 173 kernels. The motivation for this analysis is to show the similarities and differences between applications of the benchmark suites.

Figure 4 shows the biplot chart for the results of PC1 vs PC2 followed by the K-means clustering ($K = 4$). The name of the kernels in this chart is formed

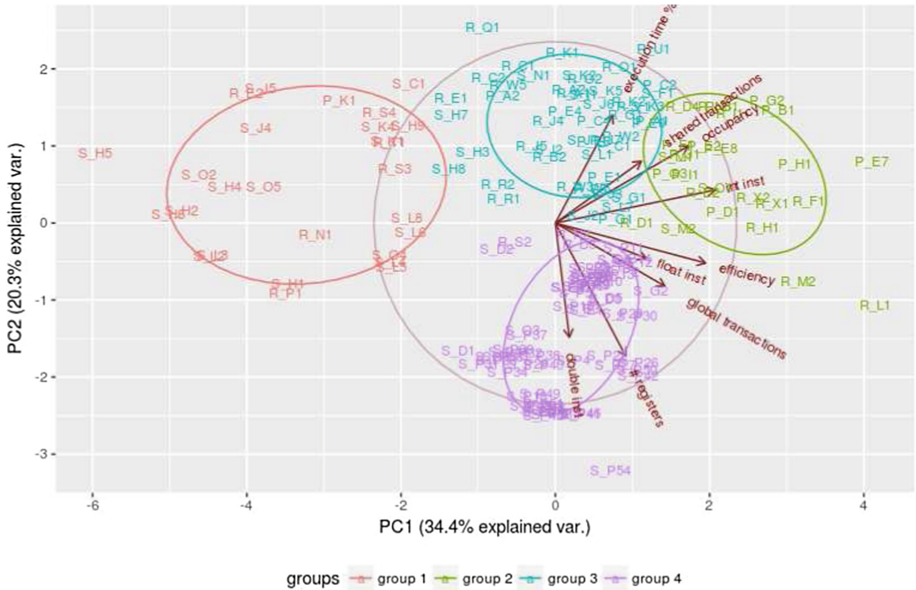


Fig. 4. Results for the kernels of all benchmarks suites.

with a similar coding scheme used in the previous analysis, but each kernel name starts with a letter indicating which benchmark suite it belongs (R, P or S). Comparing the direction of the vectors in this chart with the SHOC chart, we can observe that the large number of kernels in SHOC influenced the position of some vectors such as the percentage of execution time and the number of double-precision floating-point instructions. The position and angle between these two vectors are very similar.

Four groups of kernels were identified and are analyzed separately as follows.

Group 1 - Little Resource Usage. This group contains 27 kernels, and is a result of the combination of the kernels from benchmark suites that present low use of all the resources analyzed (integer and floating-point operations, SM efficiency, GPU occupancy and memory operations). Table 5 shows the number of kernels of each benchmark suite that comprises this group. We can observe in this table that most of the kernels in this group belongs to SHOC. Most of them are level one SHOC parallel algorithms.

Group 2 - High Resource Utilization. Group 2 consists of 26 kernels that have high efficiency, due to a great number of integer and single-precision floating-point operations. These kernels also have an average occupancy of about 70%. This is the group that present the highest resource utilization. Table 5 shows the number of kernels per benchmark suite. We can observe that Rodinia and Parboil have more kernels that intensively exploit the GPU resources than SHOC.

Table 4. SHOC kernels from each group.

Group	Application	# kernels	Kernels
1	FFT	4	FFT512_device(D3), IFFT512_device(D4), IFFT512_device(D5), FFT512_device(D6)
	GEMM	4	maxwell_sgemm_128x64_nn(E1), gemm_kernel2x2_tile_multiple_core(E2), maxwell_sgemm_128x64_nt(E3), gemm_kernel2x2_tile_multiple_core(E4)
	MD5Hash	1	FindKeyWithDigest_Kernel(F1)
	MD	1	compute_lj_force(G1)
	NeuralNet	3	axpy_kernel_val(H3), kernelBackprop1(H7), gemm_kernel1x1_core(H8)
	Reduction	2	reduce(I1), reduce(I2)
	Scan	4	reduce(J1), reduce(J2), bottom_scan(J3), bottom_scan(J6)
	Sort	3	findRadixOffsets(K2), radixSortBlocks(K3), reorderData(K5)
	Spmv	2	spmv_csr_vector_kernel(L1), spmv_csr_vector_kernel(L7)
	Stencil2D	2	StencilKernel(M1), StencilKernel(M2)
	Triad	1	Triad(N1)
	QtClustering	1	QTC_device(O1)
2	BFS	1	BFS_kernel_warp(C1)
	NeuralNet	6	kernelFeedForward3(H1), kernelBackprop3b(H2), kernelBackprop3a(H4), kernelInitNablaB(H5), kernelBackprop2(H6), kernelInitNablaW(H9)
	Scan	2	scan_single_block(J4), scan_single_block(J5)
	Sort	2	scan(K1), vectorAddUniform4(K4)
	Spmv	6	zero(L2), zero(L3), spmv_ellpackr_kernel(L4), spmv_csr_scalar_kernel(L5), spmv_ellpackr_kernel(L6), spmv_csr_scalar_kernel(L8)
	QtClustering	3	reduce_card_device(O2), trim_ungrouped_pnts_indr_array(O4), update_clustered_pnts_mask(O5)
3	FFT	2	void chk512_device(D1), chk512_device(D2)
	MD	1	compute_lj_force(G2)
	QtClustering	1	compute_degrees(O3)
	s3d	54	all kernels (P1 to P54)

Group 3 - Medium Resource Utilization. This group is composed of 51 kernels, and has some similar characteristics to group 2, high number of integer operations and average occupancy around 60%. Kernels in this group are smaller than the ones in group 2, presenting a less significant percentage of execution time. Table 5 shows the number of kernels from each suite in this group. We observe that this group contains a similar number of kernels from the three suites.

Group 4 - Low Occupancy and High Efficiency. This group is composed of 69 kernels, characterized by low occupancy, high efficiency and low percentage of execution time. This group contains mostly S3D kernels from the SHOC suite. From the 69 kernels, 44 are from S3D. This application is a computational chemistry application that solves Navier-Stokes equations for a regular 3D domain [18]. The computation is floating-point intensive, and it was parallelized by assigning each 3D grid point to one thread. The low occupancy of each of its kernels impels their concurrent implementation. Another feature of this group is the smaller number of operations with integers and the highest average use of registers than the other groups. Table 5 shows the distribution of kernels of this group in the suites. Notice that rhere are no Parboil kernels in this group, and there are only three kernels from Rodinia.

Table 5. Number of kernels for each group and benchmark suite.

Group	Patboil	Rodinia	SHOC
1	1	6	20
2	11	12	3
3	11	23	17
4	0	3	66

5.4 Discussion

Our results show that Rodinia and Parboil presented more diversity in their kernels. SHOC, on the other hand, provides less diversity but it is the only suite that exploits kernel concurrency massively. When the three suites are analyzed together, we observed four distinct groups of kernels: (1) Low significance, (2) High resource utilization, (3) Medium resource utilization and (4) Low occupancy and high efficiency. Kernels from group 1 are the ones with short execution time and low resource usage, which means that they are not appropriate for assessing the GPU hardware. Kernels from group 2 present high resource utilization, which indicates that they are not good candidates for concurrent kernel execution. Kernels from groups 3 and 4 have medium resource utilization and relatively low occupancy. These kernels are more likely to leave unused resources and provide space for concurrent execution.

6 Concluding Remarks

This work presented a detailed characterization of the three most important benchmark suites for GPUs, Rodinia, Parboil and SHOC. Our study focused on revealing the behavior of the kernels in terms of integer, single and double precision operations, SM efficiency, GPU occupancy and memory operations. We also proposed to group similar kernels in order to identify the ones with similar behavior.

The analysis and characterization of representative GPU kernels is an essential step to understand the effect of resource requirements in further concurrent execution in modern GPUs. Cross-kernel interference can drastically affect performance of applications executed in GPUs concurrently. The problem is caused by concurrent access of co-located kernels to shared resources. We believe that identifying kernels with complementary access profiles to execute concurrently can reduce interference among them. Thereby, the characterization is a first and fundamental step to be used in future strategies of kernels scheduling in GPUs.

Our results showed that the benchmarks have kernels with good diversity in terms of resource usage. We identified groups of kernels with similar behavior and distinguished the kernels that are more likely to leave unused resources. These kernels are better candidates for efficient concurrent execution.

Concurrent kernel execution is a relatively new feature in GPUs. It would be interesting that future benchmark suites for GPU exploit this feature to the full. As future work, we intend to analyze the kernels behavior in different GPU architectures. We also intend to perform a study on the performance interference of the concurrent execution of different types of kernels, and propose a intelligent strategy to benefit from all the information gathered.

References

1. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54 (2009)
2. Stratton, J.A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G.D., Hwu, W.M.W.: Parboil: a revised benchmark suite for scientific and commercial throughput computing (2012)
3. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 63–74 (2010)
4. Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving GPGPU concurrency with elastic kernels. In: ACM SIGPLAN Notices, vol. 48, pp. 407–418. ACM (2013)
5. Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L., Skadron, K.: A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In: Proceedings of the IEEE International Symposium on Workload Characterization (2010)

6. Kerr, A., Diamos, G., Yalamanchili, S.: A characterization and analysis of PTX kernels. In: IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 3–12. IEEE (2009)
7. Goswami, N., Shankar, R., Joshi, M., Li, T.: Exploring GPGPU workloads: characterization methodology, analysis and microarchitecture evaluation implications. In: 2010 IEEE International Symposium on Workload Characterization (IISWC), pp. 1–10. IEEE (2010)
8. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 141–151. IEEE (2012)
9. O'Neil, M.A., Burtscher, M.: Microarchitectural performance characterization of irregular GPU kernels. In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp. 130–139. IEEE (2014)
10. Bakhoda, A., Yuan, G.L., Fung, W.W., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, pp. 163–174. IEEE (2009)
11. Bienia, C.: Benchmarking Modern Multiprocessors. Princeton University, Princeton (2011)
12. Asanovic, K.: The landscape of parallel computing research: a view from Berkeley, Technical report UCB/EECS-2006-183, EECS Department, University of California, Berkley, CA, USA (2006)
13. SHOC (2012). https://github.com/vetter/shoc/wiki
14. NVIDIA Corporation: Profiler user's guide (2017). http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview, an optional note
15. Bienia, C.: Benchmarking modern multiprocessors, Ph.D. thesis, Princeton University (2011)
16. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.K.: Measuring benchmark similarity using inherent program characteristics. IEEE Trans. Comput. **55**(6), 769–782 (2006)
17. Che, S., Skadron, K.: Benchfriend: correlating the performance of GPU benchmarks. Int. J. High Perform. Comput. Appl. **28**(2), 238–250 (2014)
18. Spafford, K., Meredith, J., Vetter, J., Chen, J., Grout, R., Sankaran, R.: Accelerating S3D: a GPGPU case study. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 122–131. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14122-5_16

Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA

Habib Daneshpajouh[1,2(✉)], Pierre Delisle[1], Jean-Charles Boisson[1], Michael Krajecki[1], and Nordin Zakaria[2]

[1] Centre de Recherche en STIC (CReSTIC), Université de Reims Champagne-Ardenne, Reims, France
daneshpajouh.habib@gmail.com, {pierre.delisle, jean-charles.boisson,michael.krajecki}@univ-reims.fr

[2] High Performance Cloud Computing Center (HPC3), Universiti Teknologi PETRONAS, Seri Iskandar, Malaysia
nordinzakaria@utp.edu.my

Abstract. Batch Self-Organizing Map (Batch-SOM) is being successfully used for clustering and visualization of high-dimensional datasets in a wide variety of domains. Although the structure of its training algorithm has a high potential for parallelization, focus of the previous efforts has been on the original Step-wise SOM. This gap is due to the facts that Batch-SOM requires some extra precautions (specially in its initialization phase), and it took quite a while since its introduction that researchers affirmed the desirability of using it in practice over the Step-wise SOM. Hence, the purpose of this paper is to propose a GPU parallelization model and implementation for the Batch-SOM using CUDA. The most computationally expensive parts of its training algorithm (such as steps to compute distance between each data vector and neuron, and determining the Best Matching Unit based on minimum distance) are identified and mapped on GPU to be processed in parallel. The proposed implementation shown significant speedups of $11\times$ and $5\times$ compared to the sequential and parallel CPU implementations respectively.

Keywords: Self-Organizing Map · CUDA · Clustering · Parallel SOM GPGPU

1 Introduction

Self-Organizing Map (SOM) proposed by Kohonen [1] is an unsupervised neural network that provides a low-dimensional (i.e. one or two dimensional) representation of multidimensional data vectors. SOM uses a data compression technique called Vector Quantization (VQ) to perform dimensionality reduction. VQ compression works by finding local averages of the dataset (represented by centroids in K-Means algorithm and neuron weights in SOM). In contrast to classical neural networks which require an input vector together with an associated target vector to be provided beforehand, the key feature of SOM is the ability to find internal structure of data without any supervision. In a nutshell, SOM works by associating each of the input vectors to one of its neurons in an iterative process in such a way that the overall

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 87–100, 2018.
https://doi.org/10.1007/978-3-319-73353-1_6

distance between the neurons and their associated input vectors is minimized. The main goal of this process is to maintain the most important topological and/or metric relationships within the dataset in resulted low-dimensional network.

Since its introduction, SOM is being frequently used for clustering, visualization and data exploration problems in different domains such as industry, finance, natural sciences, biomedical analysis and linguistics [2]. SOM has a large application potential in engineering domain as well such as visualization of machine states, fault identification, process analysis and monitoring, and adaptive detection of quantized signals. By applying SOM to clustering problems, it will not only approximate the density function of the input samples (as most of classical clustering algorithms like K-Means do using VQ), but also provides a low-dimensional nonlinear projection of the high-dimensional datasets by topological arrangement of its neurons.

However, with the size of today's real-world datasets increasing sharply and complexity of data mining algorithms like SOM, quality of the result is not the only factor to measure the success of an algorithm, but its computational performance matters a lot too. Fortunately, just like other members of neural networks family, by having multiple computing nodes called neurons, SOM has a high potential of being parallelized. Moreover, a variation of SOM with modified training algorithm called Batch-SOM (in contrast to the original SOM with sequential step-wise training algorithm) makes it even more suitable for parallelization. According to Kohonen [3], by taking care of certain preliminaries, the result quality of the Batch-SOM is equal to (or even better than) the original SOM for majority of datatypes.

Several works have been done in the past for parallelization of SOM using different platforms on both Central Processing Unit (CPU) and Graphics Processing Unit (GPU). However, majority of the previous works emphasized on the original step-wise SOM and there is a lack of effort for parallelizing the Batch-SOM on GPU. Hence, the aim of this paper is to provide a parallelization model for the Batch-SOM using NVIDIA Compute Unified Device Architecture (CUDA) platform. The proposed GPU implementation shown significant speedups compared with the famous SOMToolbox [4] (a reference CPU implementation for the Batch-SOM provided by the Kohonen's team), and also the authors' own sequential and parallel CPU implementations.

The remaining of this paper is structured as follows. Section 2 presents the original SOM algorithm and Batch-SOM. Section 3 surveys the previous works on parallelization of SOM on GPU. The proposed GPU parallelization model of Batch-SOM is explained in Sect. 4. The performance and comparison results of the proposed model are presented in Sect. 5. Finally, Sect. 6 concludes the paper and proposes some future works.

2 SOM Algorithm

We follow Kohonen [3] to explain the two variations of SOM algorithm, and will be using the following notations for this section and also the rest of this paper:

- $x(t)$: a real n-dimensional Euclidean input vector, where integer t signifies a step in the sequence.
- X: sequence of all input vectors $\{x(t)\}$.
- m_i: a model (neuron), where i is its spatial index in SOM lattice.

- M_i: a variable sequence of all models (neurons) $\{m_i(t)\}$.
- m_c: a model (neuron) with closest distance to the input data vector passed to SOM lattice, and is located in the center of its neighbourhood. It is also called Best-Matching Unit (BMU) in SOM terminology.
- D_i: a variable set of all distances between each data vector x(t) and model (neuron) m_i.

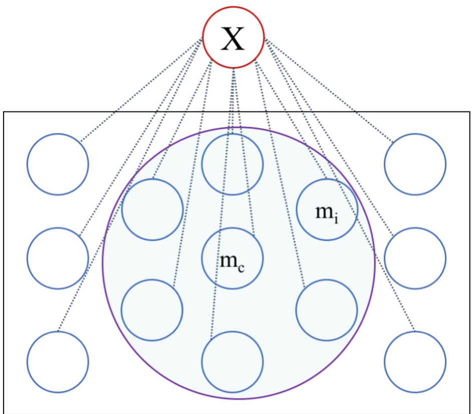


Fig. 1. Illustration of a Step-wise SOM.

Some parameters of SOM like number of neurons in each dimension of the lattice, maximum number of epochs (iterations) and learning rate (in case of Step-wise SOM) should be found using some statistical methods and heuristics that work on the basis of input data.

Learning algorithm of the original Step-wise SOM (shown in Fig. 1) known as "competitive learning" is described in the following:

1. Weights of the neurons are initialized. Initialization can be done either randomly or linearly using Principal Component Analysis (PCA) technique.
2. An input vector $x(t)$ is chosen at random from X and broadcasted to lattice.
3. The $x(t)$ is compared to each and every $m_i(t)$ to find the winner (BMU) in such a way that the neuron with index c is in the smallest distance from $x(t)$ compared to all other neurons as in the following formula:

$$c = \mathrm{argmin}_i\{\|x(t) - m_i(t)\|\} \tag{1}$$

4. The neighbourhood radius of the BMU $h_{ci}(t)$ is now calculated. The neighbourhood function has a vital role in SOM and its smooth convergence by producing large values at initial stages (typically covers the whole lattice) and decreasing overtime. This function can be defined as follows:

$$h_{ci}(t) = \alpha(t)\ e[-[x(t) - m_c]^2/2\alpha^2(t)] \tag{2}$$

where $\alpha(t) < 1$ (also called "learning rate") and $\sigma(t)$ are monotonically (e.g. exponentially) decreasing scalar functions of t, and $[x(t) - m_c]^2$ is the square distance between the neuron m_c and vector $x(t)$.

5. The weights of the BMU and each of its neighbouring neurons (found in step 4) are adjusted to be closer to the input vector. The closer a neuron is to BMU, the more its weights get altered. The formula in below calculates the new weights for each affected neuron:

$$m_i(t+1) = m_i(t) + h_{ci}(t)\,[x(t) - m_i(t)] \tag{3}$$

where $m_i(t)$ and $m_i(t+1)$ are the current and new weights of the neuron m_i respectively.

6. Steps 2 to 5 are repeated until the maximum number of epochs (i.e. iterations) is reached.

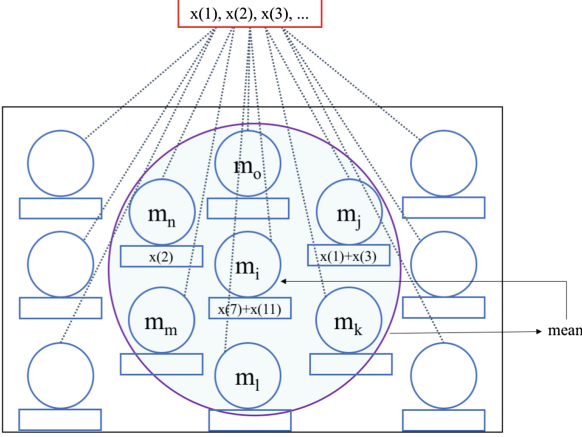


Fig. 2. Illustration of a Batch-SOM.

On the other hand, as Fig. 2 shows, the Batch-SOM works as in the following steps:

1. Weights of the neurons are initialized. Initialization can be done either randomly or linearly using PCA technique.
2. All the data vectors are passed to the lattice at once, with each neuron having its own sub-list. Each data vector is compared with each and every neuron to find its BMU. The index of each data vector goes to the sub-list of its BMU.
3. Finally, the new weights of each neuron are calculated as the weighted mean (the term "generalized median" is used by Kohonen to cover any type of data including the non-numeric types) of all the sub-lists in its neighbourhood. The mean's weight is determined by the distance of each neuron to the neuron that is getting the update.
4. Steps 2 to 3 are repeated until the maximum number of epochs is reached.

3 Related Work

Although there is a wide range of parallelization efforts for SOM, the focus of this report is only on those implemented on GPUs. The works on GPU implementations of SOM can be divided into two groups: first group includes those works that used graphics programming techniques and (vertex and fragment) shaders, while the second group belongs to those works that used CUDA and Open Computing Language (OpenCL) platforms to perform general computing on GPU. These two platforms completely revolutionized the way in which the programs are implemented on GPUs.

One of the early works on GPU parallelization of SOM is done by Zhongwen et al. [5] in which a GPU implementation of SOM is provided by converting the computational data into texture format and using the OpenGL texture buffers to perform the computation and data storage directly on GPU. The test environment for the proposed implementation contained Intel Pentium-4 2.4 GHz for CPU computation, and ATI 9550 and NVIDIA 5700 for GPU computation. By using 80 data vectors for training, the authors claimed speedups of up to 2× and 4× on the NVIDIA and ATI cards respectively.

Another work proposed at the same year as the above work is by Campbell et al. [6] in which the authors proposed a variation of SOM called Parameter-less SOM (PLSOM) which does not require to manually choose the “learning rate” parameter. They used shader functions to port SOM training computations on GPU. By trying different number of nodes for SOM lattice and a dataset including 1000 randomly generated and uniformly distributed 2-dimensional vectors, the authors claimed a speedup of up to 9× using their GPU implementation run on a GeForce 6800 card compared to a sequential CPU implementation run on a machine with 6 Intel Itanium-2 processors.

Xiao et al. [7] proposed a parallel GPU implementation for Batch-SOM using shader functions with the lattice of neurons being organized in 2D texture format. A vertex shader is responsible for finding the nearest neuron as well as the corresponding texel position for each training vector. Then, two fragment shaders are used to update the training vector and its neighbours. The authors evaluated the performance of their algorithm on an Intel Core2 Q8200 CPU and GeForce GTX 280 GPU. They claimed to achieve 15× to 40× speedup compared to a sequential CPU implementation by using different sizes of input data.

Despite the speedups that the aforementioned works achieved compared to CPU implementations, the high level of required knowledge (of graphics programming) and extra works that they were facing for converting the data structure and processing style of SOM to somewhat compatible with graphics APIs (like OpenGL) makes their approaches inconvenient in practice.

On the other hand, the major works after the introduction of CUDA are reviewed in the following. A GPU implementation of SOM is presented by Hasan et al. [8] using an open-source GPU Machine Learning Library called GPUMLib which is based on CUDA. In this work, weights initialization of neurons and subsequent update of the weights are done on host (CPU) while middle steps for finding the distance between neurons and input vectors followed by finding the BMU are done on device (GPU).

They performed an experiment on biomedical gene expression data using a NVIDIA Tesla C2075 graphics hardware and an Intel Xeon high-performance computer. The authors claimed that their GPU implementation achieved more than 3× speed up compared to the CPU implementation.

Davidson [9] proposed a parallel implementation of SOM using OpenCL on NVIDIA GPUs. In this work, Euclidean distance as the most common distance measure for SOM is replaced by Manhattan distance. Two kernels are used to find the BMU for each input vector in parallel. The performance of the OpenCL implementation is compared with a well-known SOM (CPU implementation) library called SOM_PAK [10] and shown a speedup factor of 10×.

Wang et al. [11] proposed a GPU implementation of SOM applying to travelling salesman problem using a concept called "parallel cellular matrix" that partitions the Euclidean plane of input data into a suitable number of uniform cell units. They used Nvidia GeForce GTX 570 Fermi graphics card containing 480 CUDA cores for their experiment which resulted average acceleration factors of 5.49, 12.68 and 39.74 (with respect to small, medium and large size of input data) compared to the CPU implementation.

Although the works reviewed above resulted significant speedups by their GPU implementations, one must note that the focus of these works has been mostly on the original SOM algorithm and there is a lack of effort on GPU parallelization of the Batch-SOM. While the original SOM algorithm was proposed in 1980s and the batch training algorithm a couple of years later, it took quite a while (by using it for different application areas in several research works) to confidently use the Batch-SOM as an alternative of the original SOM that might justify the lack of effort for GPU parallelization of this algorithm. As mentioned by Kohonen in [3], despite the need for taking care of some extra preliminaries (specially in the initialization phase) when working with the Batch-SOM, it is usually preferred over the original SOM in practice for several reasons:

- It does not have the time-variable learning rate parameter.
- It converges faster than the step-wise method.
- It can be generalized for the non-vectorial data too.

In addition, the Batch-SOM is more scalable to data size because of its high potential for parallelization. Therefore, the parallel-GPU model proposed in this paper focuses on the Batch-SOM by first, identifying the most computationally expensive stages of its learning algorithm, and then, using high-parallelizm potential of GPUs and also efficient processing techniques provided by CUDA platform to perform the computation of these stages.

4 Parallel Batch-SOM on CUDA

As explained by Kohonen [1], the most computationally expensive part of the Batch-SOM is the step to find the BMU for each data vector which consists of many distance comparisons. Our profiling analysis affirmed this too by showing that finding

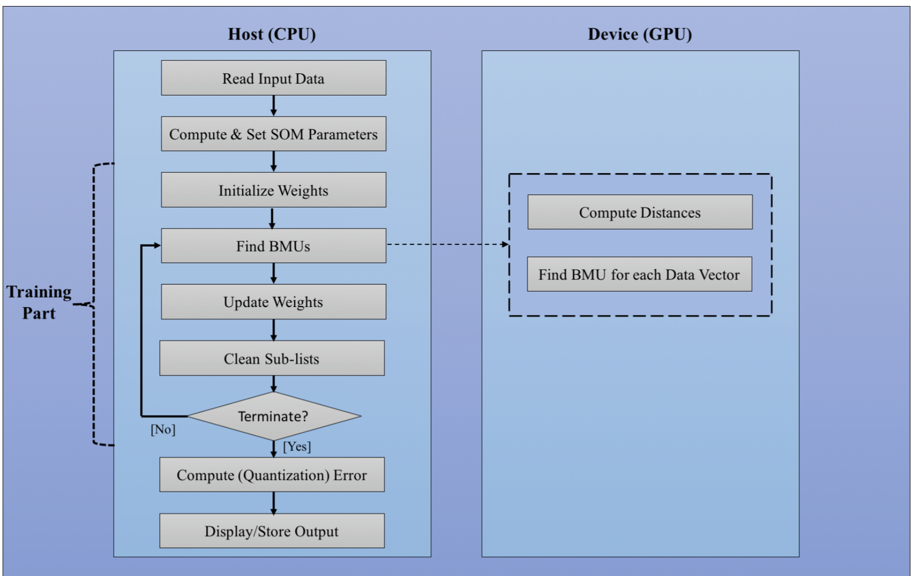


Fig. 3. Parallel Batch-SOM flowchart.

BMUs takes about 81 to 95.6% of the training's total processing time when it is run sequentially on CPU for 1000 and 7000 sizes of input data respectively. One may note that the step to update the weights takes a much smaller portion of the total processing time due to the fact that the radius of neighbourhood reduces overtime, requiring less and less computation to update the neurons. Hence, in our proposed parallelization model, computation of the step to find BMUs is ported to GPU.

Figure 3 depicts flowchart of our proposed parallel Batch-SOM. It starts with reading input data on the host side. Then, some SOM parameters such as data size, a single vector size, lattice radius etc. are computed and set. The next step (which plays a vital role in the result's quality of the Batch-SOM) is to initialize the lattice by setting the weights of its neurons to some initial values. The choice of initialization method (either randomly or linearly) depends on the data and problem types. Describing the subtleties of this choice is beyond the scope of this paper, but some useful suggestions are provided by Principe and Miikkulainen [12].

Following the initialization phase, the training loop begins by going through the procedure to find BMUs for all the input data vectors. This procedure consists of two major parts, both ported to GPU using two kernels to be processed. In its first part, distance between each data vector and neuron is computed. Then, in its second part, the BMU of each data vector is determined by finding the neuron in closest distance. The next step of the training algorithm is to update the weights of neurons based on the calculations of the previous step. The update step aims to transform the SOM lattice in such a way that it would be closer to distribution of the input dataset. At the end of the

training loop, all the sub-lists associated with neurons are cleaned and termination condition (which is usually a fixed value for maximum number of iterations) is checked. If the condition is met, Quantization Error of the resulted clustering is computed and the output is displayed/stored; otherwise, the loop continues to iterate by going back to the step of finding BMUs.

As mentioned above, computational burden of the procedure to find BMUs for each data vector is carried out by GPU. Algorithm 1 provides the pseudocode of this procedure. It begins by defining an array of CUDA streams [13] in order to be able to perform the data transfer from host to device and kernel execution in an asynchronous manner. The idea of CUDA streams comes from the fact that kernel execution in CUDA is an asynchronous action and once the host launches the kernel, it can proceed with next instruction in the program. We follow a 2-Way concurrency model (Fig. 4) for using streams in which the weights data is split into chunks to be transferred to GPU. Following the transfer of each chunk of data, our first kernel *computeDistances* (Algorithm 2) is called to operate on the respective chunk. This kernel is responsible for taking a single n-dimensional data vector and neuron, computing the Euclidean distance between them and storing the result in a global distance array. The global distance array is one-dimensional and stores distance values in a linear hierarchical structure consisting of streams, blocks and threads. After processing all the streams, the second kernel *minReduction* is launched to find the minimum distance for each data vector using the famous “parallel minimum reduction” algorithm. The *minReduction* kernel is lunched with the number of blocks equal to number of input vectors (so each block is responsible for finding the BMU for one input vector), and number of threads per block equal to half of total number of neurons. This kernel works on the global distance array that was initially stored in the device memory. Following the execution of the second kernel, the distance data is transferred from device to host. Eventually, the *findBMUs* procedure ends with calling the function *fillInSubLists* which puts the index of each data vector in its BMU’s sub-list.

```
1:  procedure findBMUs()
2:     Define an array of streams
3:     for i ← 1 to NUM_STREAMS do
4:        Create CUDA streams
5:     end for
6:     for i ← 1 to NUM_STREAMS do
7:        Calculate and set memory offset
8:        Asynchronously copy a chunk of weights from host to device memory
9:        Lunch the kernel computeDistances <<< numBlocks, numThreadsPerBlock, 0, stream[i] >>>(…)
10:    end for
11:    Lunch the kernel minReduction <<< numBlocksRed, numThreadsPerBlockRed >>>(…)
12:    Copy the distance data from device to host
13:    Call the function fillInSubLists(distances[]) to put the index of each data vector into its BMU’s sub-list
14: end procedure
```

Algorithm 1. Pseudocode of the *findBMUs* procedure

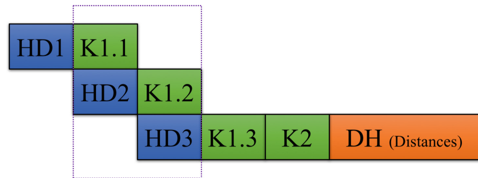


Fig. 4. 2-Way concurrency pipeline of CUDA streams used in our parallel implementation

```
1:   kernel computeDistances(…)
2:      Calculate and set vectorStartIndex
3:      Calculate and set vectorEndIndex
4:      Calculate and set weightStartIndex
5:      Calculate and set weightEndIndex
6:      Load the weights data into shared memory
7:      Synchronize the threads
8:      dist ← 0
9:      for vIdx ← vectorStartIndex to vectorEndIndex , wIdx ← weightStartIndex to weightEndIndex do
10:        dist ← dist + euclidean distance between the vector's and weight's element
11:     end for
12:     Calculate and set distanceIndex in the global distance array
13:     distArray[distanceIndex] ← dist
14:  end kernel
```

Algorithm 2. Pseudocode of the *computeDistances* kernel

To further describe the details of computations on GPU, Fig. 5 illustrates data transfers and execution of the proposed parallel implementation. There are three major sets of data in this program, each stored in a one-dimensional array. First (before the start of training loop), the set of all input data vectors X is transferred to device memory. Then, the set of weights of all neurons M_i is divided into chunks (three chunks in the case of Fig. 5) and transferred to the device memory using streams. Determining the number of streams depends on the factors such as the GPU architecture (number of connections between host and device, available registers etc.) and also data size. Although the recent GPU architectures provide the ability to have a high number of streams, the balance between data size and overhead of creating too many streams must be taken into account in order to achieve an efficient concurrency.

After transferring each chunk of the weights data, the host (CPU) launches an instance of the *computeDistances* kernel on multiple blocks of threads. Since in our implementation each thread is responsible for computing the distance between a single data vector and neuron, the total number of threads required is the total number of data vectors times the total number of neurons in the lattice. It is noteworthy that all the streams launch an almost equal number of threads and blocks. The maximum number of threads allowed in each block and the amount of available per-block resources such as shared memory determine the number of blocks and number of threads per block in each stream. The part of the weights data assigned to each block is loaded into shared memory. This is because the number of neurons is usually much smaller than the number of data vectors. So, each neuron weights are accessed many more times than

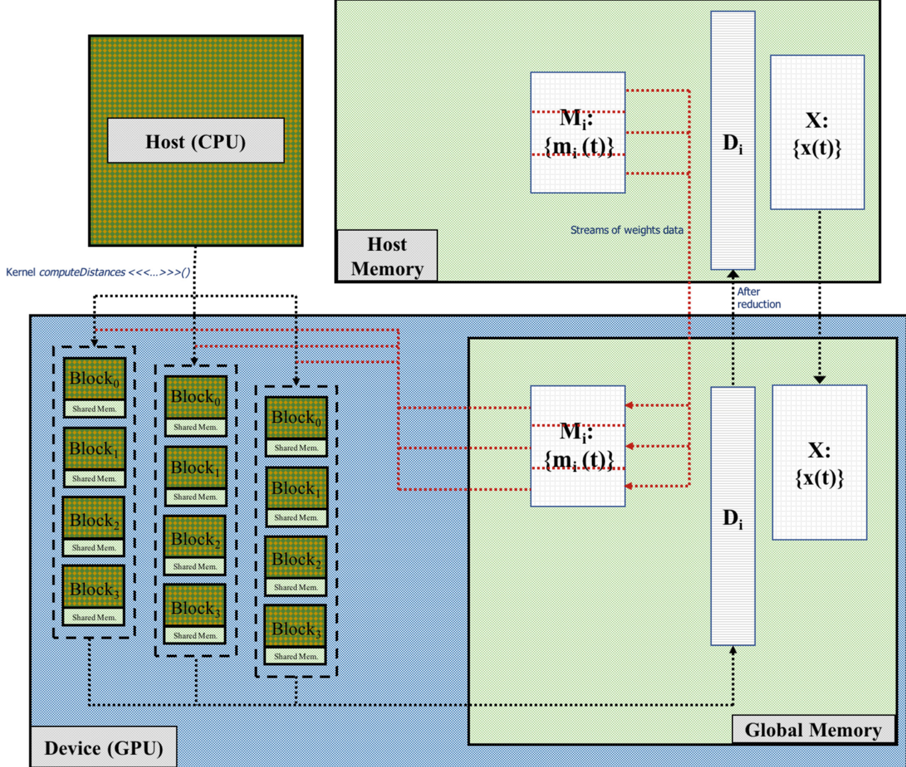


Fig. 5. Data transfers and execution of the proposed parallel implementation on GPU

the data vectors. Finally, each thread writes the result of its distance computation into the distance array that will be transferred to host memory after the reduction kernel finishes its computation.

5 Comparison and Results

In order to test the performance of the proposed parallelization model, we compared our implementation with three different implementations:

- **SOMToolbox:** A CPU-based single-thread reference implementation of the Batch-SOM, developed in MATLAB by the Kohonen's team. This package provides the ability to train a SOM network with different parameters and compute various errors, quality and measures for the SOM. It also provides visualization of the result, and correlation and cluster analysis.
- Our own sequential **CPU** implementation: This is developed in *C* by following the SOMToolbox and acts as the groundwork of our parallel implementations.

- Our own **OpenMP** implementation: This is developed in *C* in order to have an alternative parallel implementation to be compared with our CUDA implementation. Exactly like the proposed Batch-SOM on CUDA, in the OpenMP implementation, computation of the procedure *findBMUs* is done in parallel using different number of processing cores ranging from 2 to 16.

Table 1. The specifications of the machine used in our experiments

CPU (Bi-processors)	Memory	GPU	OS
Intel Xeon E5-2650V2 16 Cores (in total) 2.60 GHz	64 GB	NVIDIA Tesla K20x	Bullx Linux Server Release 6.1

All the implementations work on double precision and the necessary optimization flags are used in compilation of CPU implementations. The specifications of the machine used in our experiments are presented in Table 1. We used the dataset which was the result of Non-dominated Sorting Genetic Algorithm II (NSGA-II) runs applied to an instance of Vehicle Routing Problem with Time Windows (VRPTW) having 100 customer points with 200-dimensional integer data vectors. Each data vector occupies 800 bytes of memory. These data vectors are essentially GA chromosomes, each of which is a solution to this instance of VRPTW.

With respect to fine tuning of GA parameters for this instance of VRPTW, the population size was set to 100 with 70 as number of generations. Hence, the GA produced 7000 solutions (vectors) which we used as input data for SOM clustering. However, to see the behaviour of our parallel SOM against different sizes of input data, we extracted random subsets of 1000, 2000 and 4000 vectors from this data. The dataset being used can be retrieved from [14]. By using a heuristic provided in SOMToolbox, the maximum number of SOM iterations was set to 12 with a 15×7 lattice of neurons. Each implementation was run 50 times on different sizes of datasets.

Table 2 presents runtime comparison of the training part (as indicated in Fig. 3) of the CUDA, the OpenMP (on 2, 4, 8 and 16 cores), the sequential CPU and the SOMToolbox implementations on different sizes of input data which clearly shows that the proposed CUDA implementation is by far performing better compared to the other implementations. The speedup is more evident when the data size is increased. At its best performance when the number of input vectors is 7000, the proposed CUDA implementation provides a speedup of $5\times$, $11\times$ and $11\times$ compared to the OpenMP, the CPU and the SOMToolbox implementations respectively. Profiling information for the case of 7000 input vectors is presented by Table 3.

It is noteworthy that the slight advantage of our CPU implementation over the SOM Toolbox is because of two reasons. First, SOM Toolbox is implemented in Matlab which is typically slower than C programs. Second, unlike our implementation, SOM Toolbox includes some extra error checking and exception handling commands that is usual in all the public packages. Otherwise, our CPU implementation is the same as the one in SOM Toolbox.

Table 2. Runtime (of the training part) comparison between CUDA, OpenMP, sequential CPU and SOMToolbox implementations (time in milliseconds)

No. of Input vectors	Runtime	CUDA	OpenMP				Seq. CPU	SOM Toolbox
			2Cores	4Cores	8Cores	16Cores		
1000	Best	136	538	487	428	322	569	597
	Worst	151	549	504	433	365	584	607
	Std. dev.	142	541	495	430	341	577	601
2000	Best	215	1043	940	802	632	1137	1163
	Worst	232	1055	947	809	669	1152	1192
	Std. dev.	226	1049	942	806	651	1142	1181
4000	Best	287	1813	1571	1283	934	2012	2025
	Worst	297	1825	1593	1291	973	2029	2051
	Std. dev.	292	1817	1579	1287	959	2022	2036
7000	Best	371	3796	3288	2664	1952	4211	4262
	Worst	388	3803	3308	2685	1991	4256	4296
	Std. dev.	382	3797	3295	2672	1975	4230	4271

Table 3. Profiling information of the training part (as indicated in Fig. 3) for the case of 7000 input vectors

Section	Initialize weights ()	memcpy HtoD	Compute distances <<< >>> ()	Min reduction <<< >>> ()	memcpy DtoH	Update weights ()	Others
Time (ms)	37	9	132	24	21	141	7
Time (%)	9.97	2.43	35.58	6.47	5.66	38.01	1.88

Moreover, from the perspective of clustering quality, Table 4 provides the results of comparing our CUDA implementation with the SOMToolbox in terms of two well-known measures. The first measure is Average Quantization Error (AQE) which indicates the average of total distances of all data vectors to their respective cluster's centroids (i.e. their BMUs weights in case of SOM algorithm). The second measure is Average Silhouette Coefficient (ASC) which is one of the intrinsic methods for evaluating clustering quality by reflecting the average ratio of intra-cluster closeness to inter-cluster compactness. ASC has a range of [−1, 1]. The goal of SOM clustering is to minimize the AQE and maximize the ASC. In majority of cases, our CUDA implementation provided an equal quality to SOMToolbox while in some cases there is a slight difference (both higher and lower) between the two, which is normal because of random nature of the SOM algorithm.

Table 4. Clustering quality comparison between CUDA and SOMToolbox implementations (Average Quantization Error: lower is better – Average Silhouette Coefficient: higher is better)

No. of Input vectors	Quality	CUDA		SOMToolbox	
		Avg. quantization error	Avg. silhouette coefficient	Avg. quantization error	Avg. silhouette coefficient
1000	Best	60.8	0.9896	60.8	0.9896
	Worst	69.4	0.9854	68.6	0.9851
	Std. dev.	61.9	0.9875	62.1	0.9877
2000	Best	110.2	0.9892	110.2	0.9892
	Worst	113.7	0.9838	114.1	0.9838
	Std. dev.	110.3	0.9866	111.1	0.9869
4000	Best	195.1	0.9873	194.6	0.9873
	Worst	192.3	0.9783	191.4	0.9781
	Std. dev.	193.8	0.9823	193.5	0.9818
7000	Best	209.2	0.9625	209.2	0.9625
	Worst	211.6	0.9523	212.3	0.9566
	Std. dev.	210.7	0.9581	209.4	0.9598

6 Conclusion and Future Work

The Self-Organizing Map is a data mining algorithm being extensively used nowadays for clustering and visualization problems in different domains. SOM has a unique feature of not only providing an approximation of density function of the dataset, but also a nonlinear projection of the high-dimensional data vectors to a low-dimensional space. However, the complexity of its training algorithm and the size of today's real-world datasets makes it necessary to use some kinds of parallelization for its computation. GPUs proved to be one of the most powerful computing hardware nowadays. Hence, this paper proposed a parallelization model for the Batch-SOM, as it proved to have equal quality of the result compared to the original SOM, and it is more suitable for parallel computing. We compared our GPU implementation with other sequential and parallel CPU implementations and got significant speedups.

However, some developments can be done in future to enhance the model and implementation. From an algorithmic perspective, some useful suggestions are provided by Kohonen [3] such as multiplying the number of neurons to save computing time in constructing large SOMs, estimating the BMU location based on the previous searches, tabulating the indices of non-zero elements of sparse vectors to speed up the BMU search process and using a coarse numerical accuracy of the vectors to reduce the memory requirements of high-dimensional input data. On the other hand, the GPU architecture can benefit from future developments as well. Although the experimental case used in Sect. 5 conveniently fits a single GPU, this is not always the case. Considering the fact that the size of today's datasets (e.g. those from operations research, biology, medical image analysis etc.) might go beyond the capabilities of a

single GPU, extending the model and implementation to use multi-GPUs and also a super-computer-based implementation with multiple computing nodes, each with multiple GPUs might be useful.

Acknowledgments. This work is partially supported by Malaysia Fundamental Research Grant Scheme (FRGS) 1/2017/ICT01/UTP/02/2. The experiments reported in this work were performed on the ROMEO computational centre of Champagne-Ardenne, France (http://romeo.univreims.fr). The authors would like to thank J. Loiseau for his useful advices on the GPU implementation.

References

1. Kohonen, T.: Self-Organizing Maps. Springer, Heidelberg (2001). https://doi.org/10.1007/978-3-642-56927-2
2. Kohonen, T.: Essentials of the self-organizing map. Neural Netw. **37**, 52–65 (2013)
3. Kohonen, T.: MATLAB Implementations and Applications of the Self-Organizing Map, 201 p. Unigrafia Oy, Helsinki (2014)
4. Alhoniemi, E., Himberg, J., Parhankangas, J., Vesanto, J.: SOM Toolbox, http://www.cis.hut.fi/projects/somtoolbox. Accessed 15 Jan 2017
5. Zhongwen, L., Zhengping, Y., Xincai, W.: Self-organizing maps computing on graphic process unit. In: ESANN' 2005 - European Symposium on Artifitial Neural Networks, Bruges, Belgium, pp. 27–29 (2005)
6. Campbell, A., Berglund, E., Streit, A.: Graphics hardware implementation of the parameter-less self-organising map. In: Gallagher, M., Hogan, J.P., Maire, F. (eds.) IDEAL 2005. LNCS, vol. 3578, pp. 343–350. Springer, Heidelberg (2005). https://doi.org/10.1007/11508069_45
7. Xiao, Y., Leung, C.S., Ho, T.-Y., Lam, P.-M.: A GPU implementation for LBG and SOM training. Neural Comput. Appl. **20**, 1035–1042 (2011)
8. Hasan, S., Shamsuddin, S.M., Lopes, N.: Soft computing methods for big data problems. In: Cai, Y., See, S. (eds.) GPU Computing and Applications, pp. 235–247. Springer, Singapore (2015). https://doi.org/10.1007/978-981-287-134-3_15
9. Davidson, G.: A parallel implementation of the self organising map using OpenCL, (2015)
10. Kohonen, T.K., Hynninen, J., Kangas, J., Laaksonen, J.: SOM_PAK: The Self-Organizing Map Program Package, 27 p. (1996)
11. Wang, H., Mansouri, A., Creput, J.-C.: Massively parallel cellular matrix model for self-organizing map applications. In: 2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS), pp. 584–587. IEEE (2015)
12. Príncipe, J.C., José, C., Miikkulainen, R.: WSOM 2009. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-35230-0
13. NVIDIA: CUDA C Programming Guide (2017)
14. Daneshpajouh, H.: NSGA_II-VRP_C101 Dataset, https://drive.google.com/open?id=0ByIxLNivsQhyMVgtN0VFdTRsRms. Accessed 23 Apr 2017

Performance Prediction of Acoustic Wave Numerical Kernel on Intel Xeon Phi Processor

Víctor Martínez[1(✉)], Matheus Serpa[1], Fabrice Dupros[2], Edson L. Padoin[3], and Philippe Navaux[1]

[1] Informatics Institute (INF), Federal University of Rio Grande do Sul (UFRGS), Av. Bento Gonçalves, 9500, Campus do Vale, Porto Alegre 91501-970, Brazil
{victor.martinez,msserpa,navaux}@inf.ufrgs.br

[2] BRGM, Orléans, France
f.dupros@brgm.fr

[3] Regional University of Northwest of Rio Grande do Sul (UNIJUI), Ijuí, Brazil
padoin@unijui.edu.br

Abstract. Fast and accurate seismic processing workflow is a critical component for oil and gas exploration. In order to understand complex geological structures, the numerical kernels used mainly arise from the discretization of Partial Differential Equations (PDEs) and High Performance Computing methods play a major in seismic imaging. This leads to continuous efforts to adapt the softwares to support the new features of each architecture design and maintain performance level. In this context, predicting the performance on target processors is critical. This is particularly true regarding the high number of parameters to be tuned both at the hardware and the software levels (architectural features, compiler flags, memory policies, multithreading strategies). This paper focuses on the use of Machine Learning to predict the performance of acoustic wave numerical kernel on Intel Xeon Phi many-cores architecture. Low-level hardware counters (e.g. cache-misses and TLB misses) on a limited number of executions are used to build our predictive model. Our results show that performance can be predicted by simulations of hardware counters with high accuracy.

Keywords: Machine Learning · Geophysics applications
Many-core systems · Performance model

1 Introduction

Geophysics exploration remains fundamental to the modern world to keep up with the demand for energetic resources. This endeavor results in expensive drilling costs (100M$–200M$), with less than 50% of accuracy per drill. Thus, Oil and Gas industries rely on software focused on High-Performance Computing (HPC) as an economically viable way to reduce risks.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 101–110, 2018.
https://doi.org/10.1007/978-3-319-73353-1_7

Acoustic wave propagation approximation is the current backbone for seismic imaging tools. It has been extensively applied for imaging potential oil and gas reservoirs beneath salt domes for the last five years. Such acoustic propagation engines should be continuously ported to the newest HPC hardware available to maintain competitiveness. At the same time, on the HPC hardware front, the days of faster single core CPUs are over, and the solutions adopted are being replaced by many-core technologies [4,5].

On one hand, the trend for High Performance Computing (HPC) applications is to pay a higher cost in order to optimize the overall performance. This comes from the complexity of many interdependent factors (non-uniform memory access, vectorization, compiler optimizations, memory policies) at an architectural level that may severely influence the application's behavior. This is particularly true for stencil numerical kernels that are usually memory-bound.

On the other hand, Machine Learning (ML) is a comprehensive methodology for optimization that could be applied to find patterns on a large set of input parameters. Recently, ML algorithms have been used on HPC systems under different situations. In [16] the authors used ML algorithms to select the best job scheduling algorithm on heterogeneous platforms whereas in [2] the authors proposed an ML-based scheme to select the best I/O scheduling algorithm for different applications and input parameters. And recently, in [13] the authors used ML algorithms to predict the performance of stencil computations on multicore architectures.

In this paper, we extend the procedure to build a suitable ML-based performance model for a classical numerical model based on isotropic acoustic wave propagation for many-core architectures. The paper is organized as follows. Section 2 provides the fundamentals of the geophysical application under study. Section 3 describes the methodology of our ML-based approach. Section 4 presents configuration, simulation performance, and model accuracy. Section 5 describes related works. And finally, Sect. 6 concludes this paper.

2 Acoustic Wave Equation

In this section, we describe the application used as benchmark that simulates the propagation of a single wavelet over time over a three-dimensional domain.

We consider the model formulated by the isotropic acoustic wave propagation (Eq. 1) under Dirichlet boundary conditions over a finite 3D rectangular domain, prescribing $p = 0$ to all boundaries, and the isotropic acoustic wave propagation (Eq. 2) with variable density where $p(x, y, z, t)$ is the acoustic pressure, $V(x, y, z)$ is the propagation speed and $\rho(x, y, z)$ is the media density.

$$\frac{1}{V^2} \cdot \frac{\partial^2 p}{\partial t^2} = \nabla^2 p \tag{1}$$

$$\frac{1}{V^2} \cdot \frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{\nabla \rho}{\rho} . \nabla p \tag{2}$$

The Laplace operator is discretized by a 12th order finite differences approximation and the time derivatives are approximated by a 2nd order finite differences operator. *Petrobras*, the leading Brazilian oil company, provides a standalone mini-app of the previously described numerical method. This kernel represents the cornerstone of the classical Reverse Time Migration imaging procedure.

The code was written in standard C and leverage from OpenMP directives for shared-memory parallelism. But Indeed, the parallelization strategy relies on the decomposition of the three-dimensional domain based on OpenMP loop features. There is another implementation for GPU architectures, but it is out of scope of this work and could be analyzed in future works.

Advanced optimizations (loop interchange, vectorization, thread and data mapping) strategies have been implemented to speedup the performance on Intel Xeon Phi processors. For sake of clarity of this paper focused on the impact of machine learning methodology, we will not go into too many details regarding the implementation and interesting readers can refer to [1].

3 Testbed and Machine Learning Methodology

In this section we describe our testbed configurations and the Machine Learning (ML) model. We used Intel Xeon Phi (*Knights Landing*) many-core platform to carry out the experiments. The detailed configurations are shown in Table 1.

Table 1. Description of the Intel Xeon Phi architecture used for our experiments.

Processor	Intel Xeon Phi 7520
Clock (GHz)	1.40
Cores	68
Sockets	1
Threads	272
L2 cache size (MB)	34

Based on this platform, Table 2 details all the configurations available for our optimization categories. As it can be noted, a brute force approach would be unfeasible due to the large number of simulations required (147,968), because some of these executions can take many hours (or days).

3.1 Feature Vectors

We emphasize that the selection of the relevant feature vectors is a key ingredient of our method. In our case, we considered a classical ML model with three layers of measurements (input, hidden, and output), which are described below:

Table 2. Configurations available for our optimization procedure.

Optimization	Parameters	Total configurations
Number of threads	1	272
Scheduling policy	1	2
Chunk size	1	272
Total	**3**	**147,968**

1. **Input Layer** is defined by OpenMP implementation features such as the number of threads, the loop scheduling policy (static or dynamic), and the chunk size (which defines how many loop iterations will be assigned to each thread at a time).
2. **Hardware Counters Layer** is built on top of PAPI library to collect the most relevant metrics from the hardware counters total of last level total cache misses (measured by **PAPI_L2_TCM** event), and total of data translation lookaside buffer misses (measured by **PAPI_TLB_DM** event). We decided to use related cache values because stencils are memory bounded problems and the number of available counter is determined by the architecture.
3. **Performance Layer** represents the total elapsed time to solve the geophysical problem.

As we can see, the output depends on several parameters that create a n-dimensional problem and if we try to model it by a regression method it can not be solved by 2D or 3D classical models.

3.2 Machine Learning Model

Our ML model which is based on Support Vector Machines (SVM). This supervised ML approach has been introduced in [6] and then extended to regression problems for n-dimensional problems where support vectors are represented by kernel functions [8].

The main idea of SVMs is to expand hyperplanes through the output vector. It has been employed to classify non-linear problems with non-separable training data by a linear decision (i.e. hardware counters behavior in next section).

Our ML model was built on top of three consecutive layers, where output values of a layer are used as input values of the next layer (Fig. 1). The input layer contains the configuration values from the input vector. The hidden layer contains two SVMs that take values from the input vector to simulate the behavior of hardware counters presented in the previous section. Because hardware counters have very large values it was necessary to perform a dynamic range compression (log transformation) between the hidden layer and the output layer, this is a very common technique used in digital image processing to avoid raw data [11]. Finally, the output layer contains one SVM that takes each simulated value from the hidden layer to obtain the corresponding execution time value.

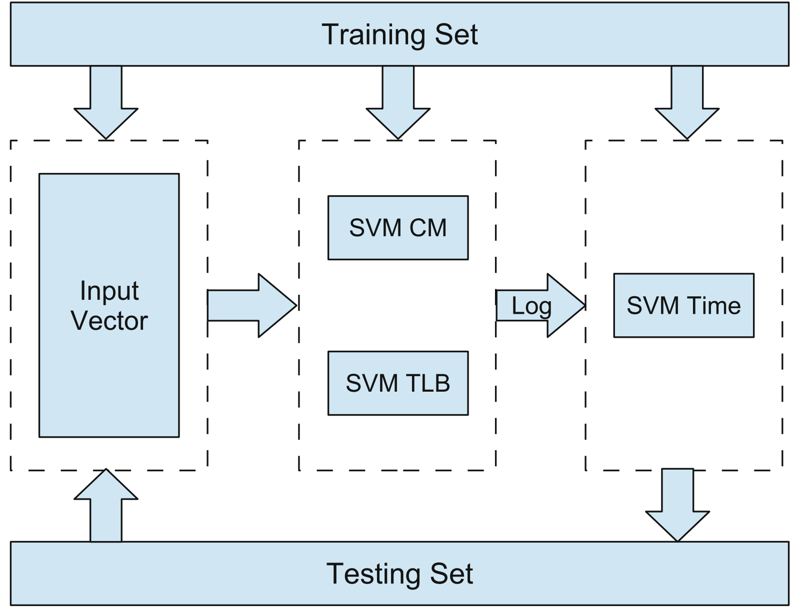


Fig. 1. Flowchart of our machine learning model.

4 Experiments

In this section we present the results of our prediction model. The use-case used as benchmark for our experiments corresponds to a three-dimensional stencil of size $1024 \times 256 \times 256$.

4.1 Preliminary Results

Hardware Counters. Figures 2 and 3 illustrate how the performance of our kernel is affected by the input variables and their relations with hardware counters. Each point represents one experiment.

For instance, Fig. 2 shows that the scheduling policy creates two separated sets when Time values are related with L2 cache misses. The same behavior is observed in Fig. 3 when chunk size create several sets if the elapsed time is observed with respect to the amount of TLB data misses.

We can resume this behavior as follows, changing input values affects the performance creating several separated areas in the graphic representation, each color represents a different value for the input value, then these areas could be separated by hyperplanes from SVM. And we note that minor cache misses are related with better performance, as expected. The sparse values are related with number of threads, chunk and scheduler changes in the input vector.

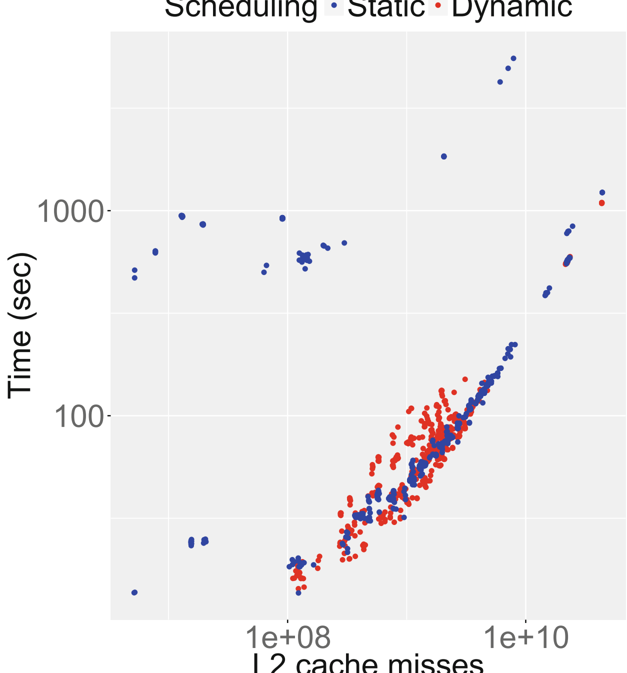


Fig. 2. L2 cache misses when varying the scheduling policy.

Elapsed Time. This work is focused on performance prediction, and we use the time as measurement to be analyzed. Firstly, we found that timing is highly affected by the input values, the standard deviation of time measure shows this variability. Some configurations take more time than the walltime available on cluster (3 h), and we can not use these input values for our experiments.

In Table 3 we present the common statistical values: mean, minimum and maximum values, and standard deviation. We can see that difference between best and worst performance is more than 400×, and standard deviation is more than 2× the mean.

Table 3. Impact of the input parameters (OpenMP scheduling policies, chunks or number of threads) on the elapsed time.

	Mean	Min	Max	Standard deviation (SD)
Time (s)	164.88	13.66	5,530.95	367.72

4.2 Performance Prediction

Training and Validation. We created a training set by randomly selecting a subset from the configuration set presented in Table 2. Then, for each experi-

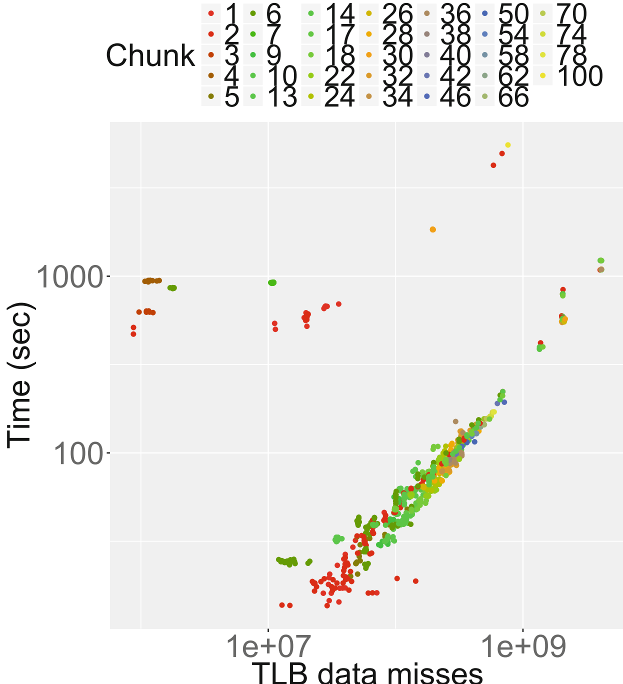


Fig. 3. TLB data misses when varying the chunk size.

ment we measured the hardware counters (L2 cache misses, and data translation lookaside buffer misses) and performance values (execution time).

A random testing set was used since all SVMs in both the hidden and the output layers are trained to calculate new execution time values. After that, we measured the accuracy of the model using statistical estimators. Table 4 presents the total number of experiments that were performed to obtain the training and validation sets.

Table 4. Number of experiments in the training and the testing sets.

Set	Number of experiments
Training	808
Testing	203
Total	**1,011**

Results and Discussions. We use the performance values from validation set and predicted values from our model to evaluate the model with two statistical estimators: root mean square error (RMSE) and the coefficient of determination (R-square). The former represents the standard deviation of the differences

between predicted values and real values whereas the latter represents how close the regression approximates the real data (R-square ranges from zero to one, equal to zero indicates regression with no one prediction and equal to one indicates a perfect fit of data prediction).

As it can be noted in Table 5, the RMSE value confirms that deviation of time value is high, and the approximation of R-square is close to 94%, then we get a highly accurate regression.

Table 5. Accuracy of our predictive modelling based on ML. We provide two statistical estimators.

Estimators	Value
RMSE	154.04
R-square	0.94

5 Related Works

Recent architectures, including accelerators and coprocessors, proved to be well suited for geophysics, magneto-hydrodynamics and flow simulations, outperforming the general purpose processors in efficiency. And some works are developed to optimize and predict the performance of these kind of applications.

Thus, in [12], the authors automatically generate a highly optimized stencil code for multiple target architectures. In [14], the authors suggest using runtime reconfiguration, and a performance model, to reduce resource consumption. In [3], the authors studied the effect of different optimizations on elastic wave propagation equations, achieving more than an order of magnitude of improvement compared with the basic OpenMP parallel version.

In [1], the authors focused on acoustic wave propagation equations, choosing the optimization techniques from systematically tuning the algorithm. The usage of collaborative thread blocking, cache blocking, register re-use, vectorization and loop redistribution. In the same way, in [9], the authors worked on target cache reuse methodologies across single and multiple stencil sweeps, examining cache-aware algorithms as well as cache-oblivious techniques in order to build robust implementations.

Other works investigated the accuracy of regression models and ML algorithms in different contexts. In [15] the authors compared ML algorithms for characterizing the shared L2 cache behavior of programs on multi-core processors. The results showed that regression models trained on a given L2 cache architecture are reasonably transferable to other L2 cache architectures. In [17] the authors proposed a dynamic scheduling policy based on a regression model that is capable of responding to the changing behaviors of threads during execution.

Finally, in [10] the authors applied ML techniques to explore stencil configurations (code transformations, compiler flags, architectural features and optimization parameters). Their approach is able to select a suitable configuration

that gives the best execution time and energy consumption. In [7], the authors improved performance of stencil computations by using a model based on cache misses. In [13], the authors proposed a ML model to predict performance of stencil computations on multicore architectures.

6 Conclusion

In this paper, we introduced a predictive performance modeling strategy for geophysical numerical kernel on many-core architectures. We showed that performance of the simplified acoustic wave equation can be predicted with a high accuracy (95%) based on hardware counters.

Moreover, the results from this work extend the based-ML strategy described in [13] for performance optimization of the elastodynamics equation on multicore architectures. Our model is not restricted to Xeon Phi platforms and can also be implemented into architectures with the available hardware counters to measure the cache-related behavior, we use the PAPI library but we believe that it don't limit our model and could be implemented with another library. Our future works can be summarized in the following lines.

Firstly, we expect to extend our methodology in order to capture complex behaviors (vectorization capabilities, data mapping). Secondly, we intend to design a model based on unsupervised ML algorithms to further improve our results. Finally, we believe that a general model can be integrated into an autotuning framework to find the best performance configuration for a given stencil kernel.

Acknowledgments. For computer time, this research partly used the resources of Colfax Research. This work has been granted by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (*CAPES*), Conselho Nacional de Desenvolvimento Científico e Tecnológico (*CNPq*), Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (*FAPERGS*). The authors thank Jairo Panetta from Aeronautics Institute of Technology (*ITA*) and *PETROBRAS* oil company for providing the acoustic wave numerical kernel code. It was also supported by *Intel Corporation* under the Modern Code Project. Research has received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the *HPC4E* Project, grant agreement n° 689772. We also thank to *RICAP*, partially funded by the Ibero-American Program of Science and Technology for Development (*CYTED*), Ref. 517RT0529.

References

1. Andreolli, C., Thierry, P., Borges, L., Skinner, G., Yount, C.: Chapter 23 - characterization and optimization methodology applied to stencil computations. In: Reinders, J., Jeffers, J. (eds.) High Performance Parallelism Pearls, pp. 377–396. Morgan Kaufmann, Boston (2015)
2. Boito, F.Z., Kassick, R.V., Navaux, P.O.A., Denneulin, Y.: Automatic I/O scheduling algorithm selection for parallel file systems. Concur. Comput. Pract. Exp. **28**(8), 2457–2472 (2016). cpe. 3606

3. Caballero, D., Farrés, A., Duran, A., Hanzich, M., Fernández, S., Martorell, X.: Optimizing Fully Anisotropic Elastic Propagation on Intel Xeon Phi Coprocessors. In: 2nd EAGE Workshop on HPC for Upstream (2015)
4. Clapp, R.G.: Seismic processing and the computer revolution(s). SEG Tech. Progr. Expanded Abs. **2015**, 4832–4837 (2015)
5. Clapp, R.G., Fu, H., Lindtjorn, O.: Selecting the right hardware for reverse time migration. Lead. Edge **29**(1), 48–58 (2010)
6. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
7. de la Cruz, R., Araya-Polo, M.: Modeling stencil computations on modern HPC architectures. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 149–171. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17248-4_8
8. Drucker, H., Burges, C.J.C., Kaufman, L., Smola, A., Vapnik, V.: Support vector regression machines. In: Advances in Neural Information Processing Systems, vol. 9, pp. 155–161. MIT Press (1997)
9. Dupros, F., Boulahya, F., Aochi, H., Thierry, P.: Communication-avoiding seismic numerical kernels on multicore processors. In: International Conference on High Performance Computing and Communications (HPCC), pp. 330–335, August 2015
10. Ganapathi, A., Datta, K., Fox, A., Patterson, D.: A case for machine learning to optimize multicore performance. In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar 2009, Berkeley, CA, USA , p. 1. USENIX Association (2009)
11. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 3rd edn. Prentice-Hall Inc., Upper Saddle River, NJ, USA (2006)
12. Kukreja, N., Louboutin, M., Vieira, F., Luporini, F., Lange, M., Gorman, G.: Devito: automated fast finite difference computation. In: Proceedings of the 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC, WOLFHPC 2016, pp. 11–19. IEEE Press (2016)
13. Martínez, V., Dupros, F., Castro, M., Navaux, P.: Performance improvement of stencil computations for multi-core architectures based on machine learning. Procedia Comput. Sci. **108**, 305–314 (2017). International Conference on Computational Science, ICCS 2017, Zurich, Switzerland, 12–14 June 2017
14. Niu, X., Jin, Q., Luk, W., Weston, S.: A self-aware tuning and self-aware evaluation method for finite-difference applications in reconfigurable systems. ACM Trans. Reconf. Technol. Syst. **7**(2), 15 (2014)
15. Rai, J.K., Negi, A., Wankar, R., Nayak, K.D.: On prediction accuracy of machine learning algorithms for characterizing shared L2 cache behavior of programs on multicore processors. In: International Conference on Computational Intelligence, Communication Systems and Networks (CICSYN), pp. 213–219, July 2009
16. Vladuic, D., Cernivec, A., Slivnik, B.: Improving job scheduling in grid environments with use of simple machine learning methods. In: International Conference on Information Technology: New Generations, pp. 177–182, April 2009
17. Weng, L., Liu, C., Gaudiot, J.L.: Scheduling optimization in multicore multithreaded microprocessors through dynamic modeling. In: Proceedings of the ACM International Conference on Computing Frontiers, CF 2013, pp. 5:1–5:10. ACM, New York (2013)

Evaluating the NVIDIA Tegra Processor as a Low-Power Alternative for Sparse GPU Computations

José I. Aliaga[1], Ernesto Dufrechou[2(✉)], Pablo Ezzatti[2], and Enrique S. Quintana-Ortí[1]

[1] Dep. de Ingeniería y Ciencia de la Computación, Universidad Jaime I, 12701 Castellón, Spain
{aliaga,quintana}@icc.uji.es

[2] Instituto de Computación, Universidad de la República, 11300 Montevideo, Uruguay
{edufrechou,pezzatti}@fing.edu.uy

Abstract. In the last years, the presence of heterogeneous hardware platforms in the HPC field increased enormously. One of the major reason for this evolution is the necessity to contemplate energy consumption restrictions. As an alternative for reducing the power consumption of large clusters, new systems that include unconventional devices have been proposed. In particular, it is now common to encounter energy-efficient hardware such as GPUs and low-power ARM processors as part of hardware platforms intended for scientific computing.

A current line of our work aims to enhance the linear system solvers of ILUPACK by leveraging the combined computational power of GPUs and distributed memory platforms. One drawback of our solution is the limited level of parallelism offered by each sub-problem in the distributed version of ILUPACK, which is insufficient to exploit the conventional GPU architecture.

This work is a first step towards exploiting the use of energy efficient hardware to compute the ILUPACK solvers. Specifically, we developed a tuned implementation of the SPD linear system solver of ILUPACK for the NVIDIA Jetson TX1 platform, and evaluated its performance in problems that are unable to fully leverage the capabilities of high end GPUs. The positive results obtained motivate us to move our solution to a cluster composed by this kind of devices in the near future.

Keywords: ILUPACK · Jetson TX1 · Sparse linear systems
High performance

1 Introduction

In a large number of scientific applications one of the most important stages, from a computational point of view, is the solution of large sparse systems of

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 111–122, 2018.
https://doi.org/10.1007/978-3-319-73353-1_8

equations. Examples are in problems related with circuit and device simulation, quantum physics, large-scale eigenvalue computations, nonlinear sparse equations, and all sorts of applications that involve the discretization of partial differential equations (PDEs) [4].

ILUPACK[1] (incomplete LU decomposition PACKage) is a well known numerical toolbox that offers highly efficient sparse linear systems solvers, and can handle large-scale application problems. The solvers are iterative procedures based on Krylov subspace methods [13], preconditioned with an inverse-based multilevel incomplete LU (ILU) factorization, which keeps a unique control of the growth of the inverse triangular factors that determines its superior performance over other preconditioners in many cases [8,14,15].

Despite the remarkable mathematical properties of ILUPACK's preconditioner, it implies a larger number of computations when it is compared with other more simple ILU preconditioners, e.g. ILU with zero fill-in. In [1] we proposed the exploitation of the task-level parallelism in ILUPACK, for distributed memory platforms, focusing on symmetric positive definite (SPD) systems, by using the preconditioned Conjugate Gradient (PCG) method. Recently, we have aimed to enhance the task-parallel approach by leveraging the data-level parallelism present in some operations by the use of graphics accelerators.

One drawback of our solution is the limited level of parallelism offered by each sub-problem in the distributed version of ILUPACK, which is insufficient to exploit the conventional GPU architecture.

On the other hand, in the last years the presence of heterogeneous hardware platforms in the HPC field increased enormously. One of the major reasons for this evolution is the necessity to contemplate energy consumption restrictions. As an alternative for reducing the power consumption of large clusters, new systems that include unconventional devices have been proposed. In particular, it is now common to encounter energy efficient hardware such as GPUs and low-power ARM processors as part of hardware platforms intended for scientific computing.

This work is a first step towards exploiting the use of energy efficient hardware to compute the ILUPACK solvers. Specifically, we developed a tuned implementation of the SPD linear system solver of ILUPACK for the NVIDIA Jetson TX1 platform, based on an NVIDIA Tegra X1 processor, and evaluated its performance in problems that are unable to fully leverage the capabilities of high end GPUs. The obtained results show that the use of this kind of lightweight devices achieves interesting runtimes, specially if it is considered that we are addressing a memory-bound problem and the gap between the memory bandwidth of a general GPU and the Jetson GPU is in the order of 8–10×.

The rest of the paper is structured as follows. In Sect. 2 we revisit the SPD solver integrated in ILUPACK and we offer a brief study about the application of the parallel preconditioner. This is followed by a description of our implementations of ILUPACK to run over the Jetson TX1 in Sect. 3, and the experimental evaluation in Sect. 4. Finally, Sect. 5 summarizes the work and makes some concluding remarks, stating the most important lines of future work derived from this effort.

[1] http://ilupack.tu-bs.de.

2 Accelerated Solution of Sparse Linear Systems with ILUPACK

ILUPACK is a software package that deals with solving the linear system $Ax = b$, where the $n \times n$ coefficient matrix A is large and sparse, and both the right-hand side vector b and the sought-after solution x contain n elements. It does so by applying Krylov subspace-based iterative solvers preconditioned with an inverse-based multilevel ILU preconditioner M, of dimension $n \times n$. The package includes numerical methods for different matrix types, precisions, and arithmetic, covering Hermitian positive definite/indefinite and general real and complex matrices. Although the preconditioner has proven to effectively boost the convergence rate in many problems, its application is usually the most computationally demanding task of each iteration of the solver.

Following, we briefly describe the main aspects related to the construction of the preconditioner in order to ease the understanding of its application in the context of iterative solvers, which is the focus of our work.

2.1 Computation of the Preconditioner

For simplicity, we base the following description on the SPD real case, where $A, M \in \mathbb{R}^{n \times n}$ are SPD, and $x, b \in \mathbb{R}^n$. The computation of ILUPACK's preconditioner is organized as follows:

1. First, a preprocessing stage scales A and reorders the result in order to reduce the fill-in in the factorization.
2. Later, $\hat{A} \approx LDL^T$ is computed by an incomplete Cholesky factorization process, where $L \in \mathbb{R}^{n \times n}$ is unit lower triangular and $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix. The partial ILU factorization obtained in this stage is of the form:

$$\hat{A} \equiv \begin{pmatrix} B & G^T \\ G & C \end{pmatrix} = LDL^T + E = \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S_c \end{pmatrix} \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} + E. \quad (1)$$

   where E contains the elements "dropped" during the ILU factorization. The factorization procedure postpones the processing of a row/column of A whenever it estimates that it would produce $\|L^{-1}\| \gtrapprox \kappa$, being κ a user-defined threshold. The postponed rows and columns are moved to the bottom right corner of the matrix and processed in a subsequent stage, yielding a multilevel structure. In the previous equation S_C represents the approximate Schur complement assembled from the "rejected" rows and columns.
3. The process is then restarted with $A = S_c$, (until S_c is void or "dense enough" to be handled by a dense solver,) yielding a multilevel approach.

At level l, the multilevel preconditioner can be expressed as

$$M_l \approx \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} \quad (2)$$

where L_B and D_B are blocks of the factors of the multilevel LDL^T preconditioner (with L_B unit lower triangular and D_B diagonal); and M_{l+1} stands for the preconditioner computed at level $l+1$.

A detailed explanation of each stage of the process can be found in [6].

2.2 Application of the Preconditioner During the Iterative Solve

The application of the preconditioner at a given level l requires solving a system of linear equation involving the preconditioner M_l and the permuted and scaled residual $\hat{r}$:

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} w = \hat{r}. \tag{3}$$

This is then solved for w in three steps,

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} y = \hat{r}, \quad \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} x = y, \quad \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} w = x, \tag{4}$$

where the recursion is defined in the second one.

Considering a partition of y and $\hat{r}$ conformable with the factors, the expressions in (4) can also be solved by

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix} \Rightarrow L_B y_B = \hat{r}_B, \; y_C := \hat{r}_C - L_G y_B. \tag{5}$$

Splitting the vectors in a similar way, the expression in the middle of (4) involves a diagonal-matrix multiplication and the effective recursion:

$$\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} x_B \\ x_C \end{pmatrix} = \begin{pmatrix} y_B \\ y_C \end{pmatrix} \Rightarrow x_B := D_B^{-1} y_B, \; x_C := M_{l+1}^{-1} y_C. \tag{6}$$

In the base step of the recursion, M_{l+1} is void and only x_B has to be computed. Finally, the expression on the right of (4) can be reformulated as

$$\begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} \begin{pmatrix} w_B \\ w_C \end{pmatrix} = \begin{pmatrix} x_B \\ x_C \end{pmatrix} \Rightarrow w_C := x_C, \; L_B^T w_B = x_B - L_G^T w_C, \tag{7}$$

where z is simply obtained from $z := \tilde{D}(\tilde{P}(\hat{P}w))$.

To save memory, ILUPACK discards the off-diagonal blocks L_G once it is done calculating the level of the preconditioner, keeping only the much sparser rectangular matrix G. Thus, (5) is changed into:

$$L_G = G^T L_B^{-T} D_B^{-1} \quad \Rightarrow \quad y_C := \hat{r}_C - G^T L_B^{-T} D_B^{-1} y_B = \hat{r}_C - G^T L^{-T} D_B^{-1} L_B^{-1} \hat{r}_B, \tag{8}$$

while the expressions related to (7) are modified to

$$L_G = G L_B^{-T} D_B^{-1} \quad \Rightarrow \quad L_B^T w_B = D_B^{-1} y_B - D_B^{-1} L_B^{-1} G^T w_C. \tag{9}$$

Operating with care, the final expressions are thus obtained,

$$\begin{array}{l} L_B D_B L_B^T s_B = \hat{r}_B \\ L_B D_B L_B^T \hat{s}_B = G^T w_C \end{array} \Rightarrow \begin{array}{l} y_C := \hat{r}_C - G s_B \\ w_B := s_B - \hat{s}_B \end{array} \tag{10}$$

In summary, the application of the preconditioner requires, at each level of the factorization, two SpMV, solving two linear systems with coefficient matrix of the form LDU, and a few vector kernels.

3 Proposal

In this section we describe the design and implementation details of the solution that will be analyzed numerically in the following sections. First, we provide a general description of our data-parallel variant of ILUPACK and then, we specify the particular strategies that were adopted for the Jetson TX1.

3.1 Exploiting the Data Parallelism in ILUPACK

Although ILUPACK is a sophisticated preconditioner that manages to significantly improve the convergence of Krylov subspace methods in many cases [5,6,16], its application is computationally expensive. We then aim to reduce the cost of the iteration by exploiting the data-level parallelism present in the operations that compose the application of the preconditioner using GPUs.

In most cases, the portion of the runtime related with the application of the preconditioner is concentrated by two main types of operation, i.e. sparse triangular system solves (SpTrSV) and the SpMV that appears in Eq. (10). NVIDIA's CUSPARSE library provides efficient implementations for these operations, so we rely on the library to offload this kernels to the GPU. The vector scalings and reorderings that are performed in the GPU via *ad-hoc* CUDA kernels, and their execution time is completely negligible when compared to that of the triangular systems or the SpMV.

The use of CUSPARSE makes necessary to make an adaptation of the structures employed by ILUPACK to store the submatrices of the multilevel preconditioner to the format accepted by CUSPARSE routines. Specifically, the modified CSR format [4] in which ILUPACK stores the L_B needs to be rearranged into plain CSR. This transformation was done only once, during the construction of each level of the preconditioner, and occurred entirely in the CPU. In devices equipped with physical Unified Memory[2], like the Jetson TX1, no transference is needed once this translation has been done, and the triangular systems involved in the preconditioner application can be solved via two consecutive calls to `cusparseDcsrsv_solve`. It is also necessary to perform the analysis phase of the CUSPARSE solver, in order to gather information about the data dependencies, generating a level structure in which variables of the same level

[2] See: JETSON TX1 DATASHEET DS-07224-010_v1.1.

can be eliminated in parallel. This is executed only once for each level of the preconditioner, and it runs asynchronously with respect to the host CPU.

Regarding the computation of the SpMV in the GPU, it is important to remember from Eq. (10) that each level of the preconditioner involves a matrix-vector multiplication with F and F^T. Although CUSPARSE provides a modifier of its SpMV routine that allows to work with the transposed matrix without storing it explicitly, this makes the routine dramatically slow. We then store both F and F^T in memory, accepting some storage overhead in order to avoid using the transposed routine.

3.2 Threshold Based Version

Our data parallel variant of ILUPACK is capable of offloading the entire application of the preconditioner to the accelerator. This strategy has the primary goal of accelerating the computations involved while minimizing the communications between the CPU and the GPU memory.

However, the multilevel structure of the preconditioner usually produces some levels of small dimension, which undermines the performance of some CUSPARSE library kernels. Specifically, the amount of data-parallelism available in the sparse triangular linear systems is severely reduced, leading to a poor performance of the whole solver.

To address this situation, in recent work [2] we have proposed the inclusion of a threshold that controls whether there is sufficient parallelism to take profit of computing a given level of the preconditioner in the GPU or if it is better to move the computations to the CPU. This has proven to boost the performance in some applications although it implies an additional CPU-GPU communication cost.

We further enhanced the procedure by moving only the triangular solves corresponding to low levels, given that it is this operation the one which dramatically degrades the performance, while in most cases we are able to take some advantage of the data parallelism present in the sparse matrix-vector products and the vector operations that remain. For the rest of the work we will consider three different implementations:

- *GPU 1 level*: computes the triangular systems of all levels but the first in the CPU while using the GPU for the rest of the operations.
- *GPU all levels*: computes the entire preconditioner application (all levels) on the GPU.
- *ARM-based*: makes all the computations in the ARM processor. This variant does not leverage any data parallelism.

We believe that the *GPU 1 level* strategy can be specially beneficial for devices like the Jetson, where each device can perform the operations for which it is better suited without adding the communication overhead necessary in other platforms.

4 Experimental Evaluation

In this section we present the results obtained in the experimental evaluation of our proposal. First we describe the hardware platform and the test cases employed in this stage, and later, we analyze the numerical and runtime results. Specifically, our primary goal is to evaluate if this kind of devices are able to solve sparse linear systems of moderate dimensions efficiently using ILUPACK. In order to do so, we start evaluating our developed variants to identify the best one. All experiments reported were obtained using IEEE single-precision arithmetic.

4.1 Experimental Setup

The evaluation was carried out in an NVIDIA Jetson TX1 that includes a 256-core Maxwell GPU and a 64-bit quad-core ARM A57 processor configured in maximum performance. The platform is also equipped with 4 GB of LPDDR4 RAM that has a theoretical bandwidth of 25.6 GB/s (see [12]).

The code was cross-compiled using the compiler `gcc 4.8.5` for `aarch64` with the `-O3` flag enabled, and the corresponding variant of CUDA Toolkit 8.0 for the Jetson, employing the appropriate libraries.

The benchmark utilized for the test is a SPD case of scalable size derived from a finite difference discretization of the 3D Laplace problem. We generated 3 instances of different dimension; see Table 1. In the linear systems, the right-hand side vector b was initialized to $A(1, 1, \ldots, 1)^T$, and the preconditioned CG iteration was started with the initial guess $x_0 \equiv 0$. For the tests, the parameter that controls the convergence of the iterative process in ILUPACK, *restol*, was set to 10^8. The drop tolerance, and the bound to the condition number of the inverse factors, that influence ILUPACK's multilevel incomplete factorization process, where set to 0.1 and 5 respectively.

Table 1. Matrices employed in the experimental evaluation.

Matrix	Dimension n	nnz	nnz/n
A126	2,000,376	7,953,876	3.98
A159	4,019,679	16,002,873	3.98
A171	5,000,211	19,913,121	3.98

It should be noted that these test cases present dimensions that are often too small to take profit of regular GPUs. These dimensions are comparable with those of each sub-problem derived from the application of the distributed ILUPACK variant on large matrices.

4.2 Results

The first experiment evaluates the performance of our 3 variants of ILUPACK developed for the Jetson TX1 platform to solve the sparse linear systems from the Laplace problem. In this line, Table 2 summarizes the runtimes implied by *ARM-based*, *GPU 1 level* and *GPU all levels* versions to solve the test cases described in Table 1. Specifically, we include the number of iterations taken by each variant to converge (*iters*), the runtimes for the application of the preconditioner (*Prec. time*), the total runtime (*Total time*), the numerical precision (i.e. the numerical error computed as $\mathcal{R}(x^*) := ||b - Ax^*||_2/||x^*||_2$) and finally, the speedup associated with both the accelerated stage with the GPU (*Prec. speedup*) and the whole method (*Total speedup*).

Table 2. Runtime (in seconds) of the three data-parallel variants of ILUPACK in Jetson TX1. *Prec. time* corresponds to the time spent applying the preconditioner during the entire solver.

Variant	Case	*Iters*	*Prec. time*	*Total time*	*Error*	*Prec. speedup*	*Total speedup*
ARM-based	A126	156	60.33	84.57	2.31E−07	-	-
GPU 1 level		156	59.00	84.85	2.37E−07	1.02	1.00
GPU all levels		156	44.36	70.30	2.45E−07	1.36	1.20
ARM-based	A159	206	161.90	228.26	3.07E−07	-	-
GPU 1 level		206	177.33	243.09	3.15E−07	0.91	0.94
GPU all levels		206	123.93	187.93	3.15E−07	1.31	1.21
ARM-based	A171	222	218.53	306.78	3.02E−07	-	
GPU 1 level		222	170.76	253.84	3.03E−07	1.28	1.21
GPU all levels		222	146.09	229.45	3.10E−07	1.50	1.34

First of all we focus on the numerical aspects of our variants. In this sense, it can be noted that all variants needed the same number of iterations to reach the convergence criteria for each of the test cases addressed. In the other hand, the residual errors attained are not exactly the same. However, the differences are not at all significant and can be explained by the use of single precision floating point arithmetic in conjunction with the parallel execution.

Considering the performance results, it should be highlighted that the *GPU all levels* version outperforms the *GPU 1 level* counterpart for all test cases. This result is not aligned with our previous experiences (see [2]) and can be explained by the elimination of the overhead caused by transferring data between both processors (ARM and GPU), allowed by the Unified Memory capabilities of the Jetson platform.

In the same line, the *GPU all levels* version implies lower runtimes than the *ARM-based* variant, but these improvements are decreasing with the dimension

of the addressed problem. This result is consistent with other experiments, and relates to the fact that GPUs requires large volumes of data to really exploit their computational power.

If we take the performance of *GPU all levels* in other kind of hardware platform into consideration, it is easy to see the benefits offered by the Jetson device. To illustrate this aspect we compared our previous results for the GPU-based ILUPACK from [3], run on an NVIDIA K20 GPU, to compare the runtimes. Table 3 summarizes these results, focusing only in the preconditioner application runtime, and contrasting it with the one obtained in the Jetson TX1 platform.

It should be recalled that ILUPACK, as a typical iterative linear system solver, is a memory-bounded algorithm. Hence, when comparing the performance allowed by the Jetson with other GPU-based general hardware platforms, it is necessary to analyze the differences between their memory bandwidth. As an example, the NVIDIA K20 GPU offers a peak memory bandwidth of 208 GB/s [11], while the NVIDIA Jetson only allows to reach 25.6 GB/s, i.e. a difference above 8×.

Table 3. Runtime (in seconds) of GPU-based ILUPACK in a K20 (from [3], using double-precision arithmetic) and the GPU-all variant of ILUPACK in Jetson TX1 (in single-precision).

Case	K20			Jetson		
	Iters	*Prec. time*	*Time by iter*	*Iters*	*Prec. time*	*Time by iter*
A126	44	11.38	.26	156	44.36	.28
A159	52	19.75	.38	206	123.93	.60
A171	-	-	-	222	146.09	.66
A200	76	28.28	.37	-	-	-

Before analyzing the results, it is important to remark that the computations in both works are not exactly the same. Note that preconditioners generated by executing ILUPACK have a different drop tolerance input parameter, and hence are distinct since this parameters affects the amount of fill-in allowed in the triangular factors. However, the runtime by iteration is an acceptable estimator for the performance of each version.

The results summarized in Table 3 show that the time per iteration for the smallest case is similar in the two platforms. Considering that the experiments in the K20 GPU were performed using double precision, it is reasonable to expect this runtimes to be reduced in half[3] if single precision is used. This means that the difference in performance is of about 2× in favour of the K20. Nevertheless, this gap is considerably smaller than the difference in the bandwidth of both devices, which is of approximately 8×. However, it can also be observed that the benefits offered by the Jetson hardware start to diminish when the dimension of

[3] Assuming a memory-bound procedure.

the test cases grow (note that in the case A159 is near to 3× if we estimate the single precision performance of the K20 as before).

This result shows that when the dimension of the addressed test case is enough to leverage the computational power of high end GPUs this kind of lightweight devices are not competitive. On the other hand, in contexts where the problem characteristics do not allow exploiting commodity GPUs efficiently, this kind of devices (e.g. the Jetson TX1) are a really good option. Additionally, the important difference in power consumption between the two devices (the K20 has a peak power consumption of 225W[4], while the Jetson only 15W[5]) should also be taken into account.

With the obtained results, our next step is to develop a distributed variant of ILUPACK specially design to run over a cluster of low power devices, as a NVIDIA Jetson TX1, and evaluate the energy consumption aspects. It should be noted that this kind of clusters are not yet widespread, but some examples are the one built in the context of the Mont-Blanc project, leaded by the Barcelona Super Computing (BSC) Spain [7], and the one constructed by the ICARUS project of the Institute for Applied Mathematics, TU Dortmund, Germany [9,10].

5 Final Remarks and Future Work

In this work we have extended the data-parallel version of ILUPACK with the aim to contemplate low power processors, as the NVIDIA Jetson TX1 hardware platform. In particular, we implemented three different versions of the solution that take into account the particular characteristics of this kind of devices, two GPU-based variants (*GPU 1 level* and *GPU all levels*) and the other centered on the use of the ARM processor (*ARM-based*).

The numerical evaluation exhibits that the *GPU all levels* variant outperforms the other options for the test cases addressed. However, when the dimension of the problems decreases the *ARM-based* version starts to be more competitive. Additionally, as the dimension of the problem grows the benefits related to the use of restrictive platforms such as the Jetson start to disappear, and the utilization of conventional GPU platforms becomes more convenient.

Given the importance of the results obtained we plan to advance in several directions, which include:

- Assessing the use of other kinds of small devices, such as the recently released Jetson TX2 (with 8 GB of memory and 59.7 GB/s of memory bandwidth).
- Developing a distributed variant of ILUPACK specially designed to run over a cluster of lightweight devices, as a NVIDIA Jetson TX1.
- Evaluate the distributed variant in a large Jetson-based cluster, such as the ones developed in the context of Mont-Blanc or ICARUS projects.
- Studying the energy consumption aspects of this distributed version of ILUPACK for small devices.

[4] TESLA K20 GPU ACCELERATOR - Board Specifications - BD-06455-001_v05.

[5] JETSON TX1 DATASHEET DS-07224-010_v1.1.

Acknowledgments. The researchers from the *Universidad Jaime I* were supported by the CICYT project TIN2014-53495R of The researchers from *UdelaR* were supported by PEDECIBA and CAP-UdelaR Grant.

References

1. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In: Jónasson, K. (ed.) PARA 2010. LNCS, vol. 7133, pp. 162–172. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28151-8_16
2. Aliaga, J.I., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Design of a task-parallel version of ILUPACK for graphics processors. In: Barrios Hernández, C.J., Gitler, I., Klapp, J. (eds.) CARLA 2016. CCIS, vol. 697, pp. 91–103. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57972-6_7
3. Aliaga, J.I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Leveraging data-parallelism in ILUPACK using graphics processors. In: Muntean, T., Rolland, R., Mugwaneza, L. (eds.) IEEE 13th International Symposium on Parallel and Distributed Computing, ISPDC 2014, Marseille, France, 24–27 June 2014, pp. 119–126. IEEE (2014)
4. Barrett, R., Berry, M.W., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, vol. 43. SIAM, Philadelphia (1994)
5. Bollhöfer, M., Grote, M.J., Schenk, O.: Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. SIAM J. Sci. Comput. **31**(5), 3781–3805 (2009)
6. Bollhöfer, M., Saad, Y.: Multilevel preconditioners constructed from inverse-based ILUs. SIAM J. Sci. Comput. **27**(5), 1627–1650 (2006)
7. Anonymous Contributors: start—mont-blanc prototype (2016). Accessed 10 July 2017
8. George, T., Gupta, A., Sarin, V.: An empirical analysis of the performance of preconditioners for SPD systems. ACM Trans. Math. Softw. **38**(4), 24:1–24:30 (2012)
9. Geveler, M., Ribbrock, D., Donner, D., Ruelmann, H., Höppke, C., Schneider, D., Tomaschewski, D., Turek, S.: The ICARUS white paper: a scalable, energy-efficient, solar-powered HPC center based on low power GPUs. In: Desprez, F., et al. (eds.) Euro-Par 2016. LNCS, vol. 10104, pp. 737–749. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58943-5_59. ISBN 978-3-319-58943-5
10. Geveler, M., Turek, S.: How applied sciences can accelerate the energy revolution-a pleading for energy awareness in scientific computing. In: Newsletter of the European Community on Computational Methods in Applied Sciences, January 2017, accepted
11. NVIDIA: TESLA K20 GPU Accelerator (2013). https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. Accessed 10 July 2017
12. NVIDIA: NVIDIA Tegra X1 NVIDIAs New Mobile Superchip (2015). http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf. Accessed 10 July 2017
13. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM Publications, Philadelphia (2003)

14. Schenk, O., Wächter, A., Weiser, M.: Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. SIAM J. Sci. Comput. **31**(2), 939–960 (2009)
15. Schenk, O., Bollhöfer, M., Römer, R.A.: On large scale diagonalization techniques for the Anderson model of localization. SIAM Rev. **50**, 91–112 (2008)
16. Schenk, O., Wächter, A., Weiser, M.: Inertia revealing preconditioning for large-scale nonconvex constrained optimization. SIAM J. Sci. Comput. **31**(2), 939–960 (2008)

HPC Applications and Tools

Benchmarking Performance: Influence of Task Location on Cluster Throughput

Manuel Rodríguez-Pascual, José Antonio Moríñigo(✉), and Rafael Mayo-García

Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas CIEMAT, Madrid, Spain
{manuel.rodriguez,josea.morinigo}@ciemat.es
http://rdgroups.ciemat.es/web/sci-track

Abstract. A variety of properties characterizes the execution of scientific applications on HPC environments (CPU, I/O or memory-bound, execution time, degree of parallelism, dedicated computational resources, strong- and weak-scaling behaviour, to cite some). This situation causes scheduling decisions to have a great influence on the performance of the applications, making difficult to achieve an optimal exploitation with cost-effective strategies of the HPC resources. In this work the NAS Parallel Benchmarks have been executed in a systematic way in a modern state-of-the-art and an older cluster, to identify dependencies between MPI tasks mapping and the speedup or resource occupation. A full characterization with micro-benchmarks has been performed. Then, an examination on how different task grouping strategies and cluster setups affect the execution time of jobs and infrastructure throughput. As a result, criteria for cluster setup arise linked to maximize performance of individual jobs, total cluster throughput or achieving better scheduling. It is expected that this work will be of interest on the design of scheduling policies and useful to HPC administrators.

Keywords: MPI application performance · Benchmarking
Cluster throughput · NAS Parallel Benchmarks

1 Introduction

The evolution in processors during the last decade has been oriented towards an increasing degree of parallelism and this fact has deeply impacted all levels of computing hardware and software. The design of clusters and supercomputers is also following this path. For example, the number of cores according to the TOP500 list [1] has grown exponentially since 1993. Current trends include the use of many-core processors, driving the number of computing units even further. An obvious way of getting the most of this is the usage of highly parallel applications. MPI and OpenMP continue being instrumental to create highly scalable applications suitable for this environment. Applications can be classified

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 125–138, 2018.
https://doi.org/10.1007/978-3-319-73353-1_9

into CPU, I/O or memory-bound depending on which factor limits its execution speed. Other common issues regarding requirements are execution time, degree of parallelism and required computational resources, to cite some. This leads to many questions and specific scheduling decisions. Probably the first is what should we do with partially-filled multi-core processors? When a given job is only using some of the CPUs of a node, or just some of the cores of a CPU, what is the most efficient decision? On one side, executing some other job at the same time would lead to a most intensive usage of the resources; on the other, sharing the resources (namely memory at different levels and I/O) may force both jobs to compete for them and slow down, having in fact a negative impact on the total execution time. The problem is complex and has extra dimensions, and the scope of this work aims to shed some light on performance issues.

2 Related Work

Impact of MPI task locality has been investigated in [2] with three kernels of NPB and application codes running in a cluster of 32 CPUs. They show that an execution time saving of up to 25% is possible. Their number of used CPUs is small and the authors plan to extend the experiments to large-scale machines since it seems necessary to be conclusive about what happens in situations of many processors. In [3] it is summarized the results of mapping MPI tasks onto sockets, taking into account the machine topology. Results show that it is beneficial to map tasks onto as many sockets per node as possible (the bigger savings in execution time, up to 30%, are obtained precisely for those cases). Similar experiments are done in [4], reporting an improvement of about 15%. In particular, research dealing with multicore architectures has been focused in the last years. To this regard, [5] presents the gain in computational efficiency of a MPI-based production application that exhibits a performance peak improvement of about 9% (with averaged performance improvement of 6%), attributed to a better use of cache-sharing at the same node and to the high intra- to internode communication ratio of the cluster. Although it is a modest speedup, it is noticed that it is obtained with minor source code modifications. The work in [6] points to the same direction by evaluating the impact of multicore architectures in a set of benchmarks; but on the contrary, they conduct a non-straightforward adaptation of the original application. Their characterization of the inter- to intranode communications ratio throws a figure of 4 to 5 in the worst case. This kind of node mappings is an area where little to moderate efforts are required for significant gains in application performance. The impact of internode and intranode latency is analyzed in [7] using a parallel scientific application with MPI tasks mapped onto the CPUs of an infiniband-based cluster of 14 nodes. With the objective of improving the computational efficiency, [8] analyzes how many cores per node should be used for applications execution. Here, both NPB and a large-scale scientific application code are executed in three single-socket-per-node clusters. They identify that task mapping is an important factor on performance degradation, being the memory bandwidth per core the primary

source of performance drop when increasing the number of cores per node that participate in the computation. Something similar concludes [9], showing a high sensitivity of the attained NAS kernels performance to the multi-core machines. In [10] it is detected that NPB exhibit high sensitivity to the cluster architecture. Also, MPI tasks mapping reveals that distributing them over the nodes is better from a computational standpoint in most cases. According to their experiments, an up to 120% speedup is attained for most of the NAS kernels. They explain this behaviour because by distributing the tasks they do not have to compete for node local resources, a scenario that seems to occur when running tasks are sharing a slot or are located in slot of the same node. In [11] a semi-empirical predictive model is formulated and tested with a large-scale scientific application code, which provides good results for weak scalability cases and show that it can lead to a 5% increase of the execution time. The study conducted in [12] on mapping MPI tasks to cores using micro-benchmarks and NPB, shows that it may affect significantly the performance of intranode communication, which is closely related to the inter- to intranode communication ratio.

These previous investigations point out to the large sensitivity of the execution time to the task mapping. The impact of grouping or not MPI tasks "outside of the box" (out of the same node), over sockets or cores within the node is high as it is seen that execution time varies significantly. Also the lack of understanding of how to proceed in a systematic manner with an application in a specific cluster remains and further clarifications are needed to improve the cluster efficiency and usage. The present investigation summarizes the results of mapping MPI tasks onto cores in two different infiniband-based clusters at CIEMAT.

Then, processor mapping combinations have been tested to explore the impact on cluster throughput and hence, to build usage criteria aiming at feeding better scheduling strategies to support the scientific groups.

Hence, the present work explores the behaviour of different scientific kernels on modern infrastructures and the impact on clusters throughput. An analysis on how task location, network traffic and resource sharing affect their execution time has been carried out to infer a generalization of their behavior. This information can then be useful to improve scheduling algorithms and cluster setups. In the text, a *job* is composed by one or more *tasks*. The assignment of those tasks to computational resources (*mapping*) determining when and where to run each job constitutes the *scheduling* process.

3 Characterization of HPC Facilities

3.1 Benchmarking

The chosen applications for systematic benchmarking can be divided in two groups. The first one is system benchmarks, to measure the raw performance of the components of our clusters. The second one is application benchmarks (NPB), to test the behavior of the infrastructure when running real applications.

STREAM benchmark [13] measures sustainable memory bandwidth and tests the communication bandwidth between the socket and its RAM. In multicore sockets an OpenMP flag is set for building one thread/core during compilation.

OSU micro-benchmark [14] measures the latency and bandwidth of MPI libraries and interconnects. It implements a set of routines with various communication patterns. It measures intra- and internode communication bandwidths.

Bonnie++ [15] is a small yet powerful benchmark to measure disk performance. It provides a number of tests of hard drive and file system performance.

Intel Memory Latency Checker 3.1 [16] allows accessing memory chunks located in the elements of the memory hierarchy, measuring the latency time.

All together, these benchmarks allow characterizing the cluster raw performance. Hence, a better understanding of the results gathered with a set of scientific kernels is intended. The NAS Parallel Benchmarks (in short NPB) [17] is developed at the Numerical Aerodynamic Simulation (NAS) program at NASA. It has evolved as a group of kernels, set for a variety of problem sizes (classes) of increasing computing cost. All together are representative of algorithmic building blocks found in the aforementioned scientific. Among them are examples of memory-bound and CPU-bound, or weak-scaling and strong-scaling kernels. The present investigation uses NPB v2.0, which includes seven portable kernels (Fortran90, MPI parallelised) whose acronyms corresponds to: BT- Block Tridiagonal solver; CG- Conjugate Gradient; EP- Embarrassingly Parallel; FT- Discrete fast Fourier Transform; IS- Integer Sort; LU- Lower-Upper Gauss-Seidel solver; MG- MultiGrid on a sequence of grids.

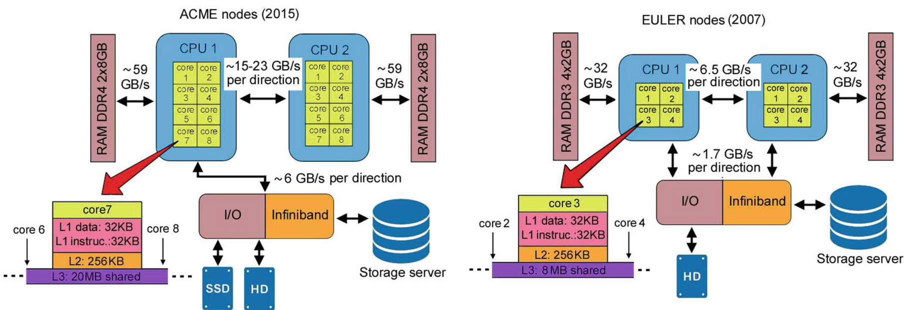


Fig. 1. Major architecture and communications for clusters ACME and EULER (bandwidths are approximate best cases sustained values).

3.2 Infrastructure Characterization

Two computing facilities of different generations based at CIEMAT have been employed. The first one, EULER, is a production HPC shared among the scientific groups. It consists in 480 Xeon®5450 quadcore@3.0 GHz, 2 GB RAM/core, mounted on Dell PowerEdge M610 blades. When EULER was installed in Autumn 2008, its performance according to LINPACK (23 TFlops peak) would

have ranked it around position 300 of TOP500. Because it is under full usage, the experiments have been done while sharing the resource and the restriction of accessing to a reserved set of 8 nodes within a limited timeframe (this matches with how the analyzed applications behave in real environments).

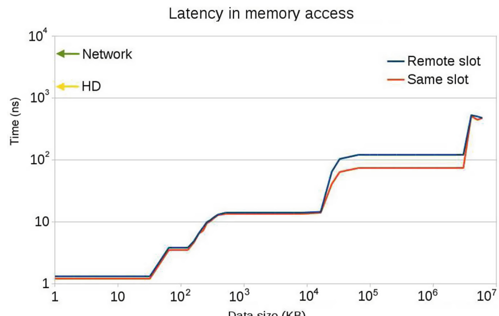


Fig. 2. Latency of a core accessing increasingly larger data blocks in ACME, corresponding to cache, RAM and disk. 'Same slot' and 'Remote slot' refer to the location of the processor to which the accessed memory is connected in the NUMA system.

The second one, ACME, is a smaller, state-of-the-art HPC for research purposes and fully devoted to this project, which counts with 10 nodes (8 of them are computing nodes): 2 Bull R424-E4 chassis with 4 nodes each, plus another two devoted to host accelerators and fast network storage. Each node consists in a Supermicro X10DRT-P motherboard with two 8-core Xeon®E5-2640 v3@2.60. Each node counts with 32 GB DDR4 RAM memory @2133 MHz, in the form of 4×8 GB modules. Two of these modules are connected to each processor (NUMA). Each core accesses to half the memory with rather smaller access time than to the other half, as show in Fig. 2. Network is provided by Infiniband FDR. The MPI library mvapich2-2.2b is installed in both. Figure 1 displays the hardware of ACME and EULER nodes. It includes some performance metrics obtained with the benchmarks and from the official documentation. It is worth noticing that most the results match their counterparts provided by the hardware vendors. ACME has higher intranode communication bandwidth (3 to 4 times higher than EULER's); 2.5 times larger shared L3 cache; and 3 times higher infiniband bandwidth. Both clusters have a ratio of intra- to internode bandwidth within 3 to 3.5.

3.3 Influence of Node Sharing on Memory Access Time

In multi-core CPUs and in multi-CPUs nodes, there is a decision concerning whether it is better or not to share the resources among pending jobs, or should a sole application be executed at the same time. It can be inferred that sharing a node between two or more jobs may increase RAM misses, as the available

memory is shared between the running jobs, so they have less space for data and executable. This same issue happens when sharing a multi-core CPU, leading to an increased number of misses in L3 cache. Hence, it is necessary to measure the access time to the different memory layers in order to quantify its influence.

Figure 2 shows latency in ACME when a given core is accessing to memory. As expected, L1 cache is the fastest one but only stores up to 32 KB of data; then comes L2 with 128 KB, L3 with 20 MB, and after that the RAM memory. In this case, as pointed out before, there is a significant difference (60%) between accessing the modules connected to the same processor and those connected to the other one in the other slot of the same motherboard. The last step corresponds to the sizes between 32 and 64 GB, where both RAM modules are accessed to store/read. The last step of the memory hierarchy is represented by the persistent storage. It counts with a HD (1 TB) for scratch and temporary files and network storage for the users' home directory and the non-OS applications (scientific codes). Measured latency is 1.5 µs for the HD and 7.35 µs for the network storage, roughly an order of magnitude larger than RAM latencies.

Summing up, results show that a miss in any cache level implies accessing an upper layer of the memory hierarchy with a penalization of about an order of magnitude in latency. Resource sharing will then have an impact on the job execution due to the influence on the access time of the different cache levels.

4 Results

4.1 Cluster Performance

The experiments with NPB have been repeated under four Slurm setups:

- Dedicated Cores: one-to-one assignment of cores to MPI tasks of the parallel job (a core executes only one MPI task of that job; set in ACME and EULER).
- Dedicated Sockets: a socket may only execute MPI tasks of the same job. Once the socket is occupied by at least one MPI task of a job, no other part of another job may be executed on it in the meanwhile (set in ACME).
- Dedicated Nodes: an entire node is assigned in exclusivity to execute MPI tasks of the same job (set in both ACME and EULER).
- Dedicated Network (reference case): the whole cluster executes only one parallel job at the same time, thus avoiding any overhead due to network congestion (set in ACME). This is a scenario devoted to obtain reference execution times.

Table 1 compares the total execution time of a set of NAS kernels. To mimic real-life workloads, all jobs corresponding to different kernel classes (sizes of computed problems) and degrees of parallelism (about 4,000 jobs sent for each cluster setup) were submitted at the same time, letting Slurm scheduler to decide where and when to execute them using 8 nodes (16 cores/node) in ACME and 16 nodes (8 cores/node) in EULER. There was no indication of any job maximum execution time, thus no preemption techniques were employed. Table 2 shows

Table 1. Total execution time of NAS kernels in ACME with different cluster setups (time ratio is referred to the Dedicated Network setup)

Setup	Execution time (s)	Time ratio (%)
Dedicated nodes	36533	32.6
Dedicated cores	19442	17.5
Dedicated sockets	28989	25.8

Table 2. Submitted jobs of all NAS kernels partitioned per degree of parallelism.

Degree of parallelism	1	2	4	8	16	32	64	128
Number of jobs	210	360	630	720	840	540	400	170
Percentage of jobs (%)	5.4	9.3	16.3	18.6	21.7	14	10.3	4.4

the jobs according to their degree of parallelism; note that 128 is the number of cores in the cluster ACME. This way, the impact of MPI tasks location inside the clusters has been analysed under the Slurm setups. It is noted that the Dedicated Sockets setup is the 2nd more efficient after the Dedicated Cores setup (about 30% of the jobs counts for 8 or even less MPI tasks), which is able to allocate more than one job at the same time.

4.2 NAS Benchmarking

The number of nodes (nN) times the number of MPI tasks per node (nT), in short $nNxnT$ (see Fig. 3), that define the configuration of each experiment (say, a definite kernel of a given class, executed under a cluster setup) has been repeated 10 times, with their average referred in what follows as a computed case. The experiments conducted under Dedicated Network setup include the kernels of class D to enlarge the population of computed cases. For the other

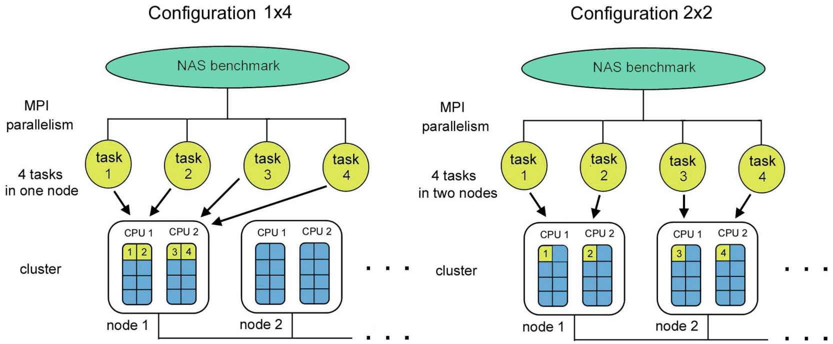


Fig. 3. Example of 1×4 and 2×2 MPI tasks mapping in cluster ACME.

cluster setups, only experiments with the A, B and C classes of the kernels have been conducted to guarantee that the running MPI processes fit into the RAM memory as well as to bound the workload computing time. All exhibit standard deviation <1%.

4.3 Dedicated Nodes Cluster Setup

Bearing in mind that the Dedicated Cores setup is a realistic scenario of production clusters, a general trend can be stated out of Fig. 4, which depict the execution time of the seven NAS kernels (sectors of the circles) for ACME. The nondimensional execution time (referred to the execution time obtained in the cluster configuration of the fewest number of nodes, which corresponds to the circle's centers) is shown in the figure. Figure 4 shows that most computed cases takes more execution time as far as more nodes are involved.

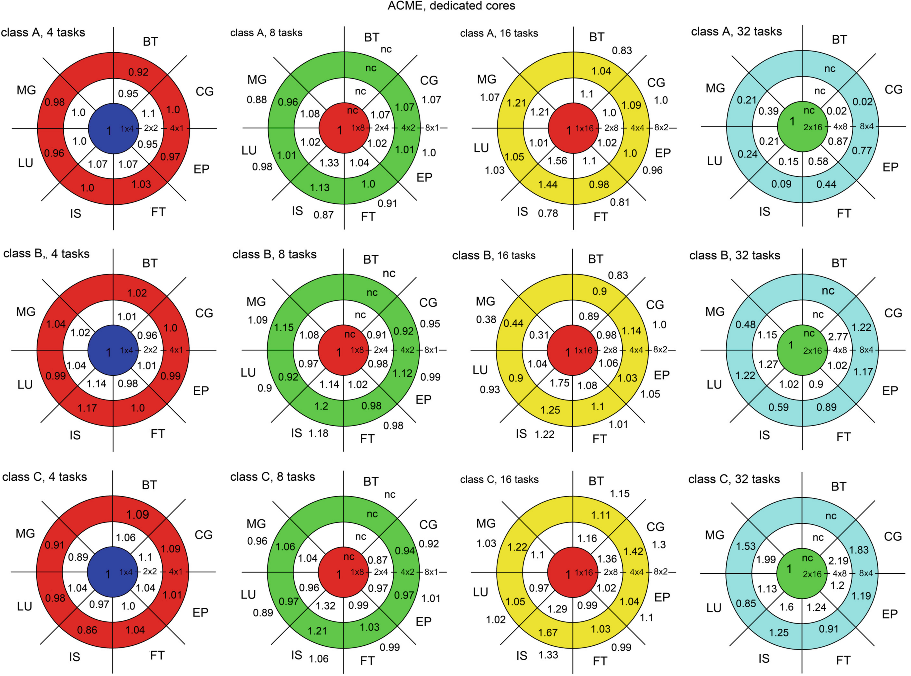


Fig. 4. Relative execution time for the Dedicated Cores setup in ACME. Nondimensional computing time is referred to the case of all processes running in one node.

Hence, it can be said that grouping MPI tasks within as few nodes as possible is good to achieve shorter execution times. This seems a general rule inferred out of the plot after examining the behaviour of the kernels as a whole. But two points must be made. On one hand, a case by case examination reveals

that there are exceptions to this rule, as it is the case of the LU kernel (class B - 8 tasks); IS kernel (class C - 4 tasks); and others. So awareness of non consistent tendencies of the kernels has to be considered. And on the other hand, non monotone behaviour occurs in several computed cases. Some of these may be explained by the kernel requirements (CPU- or memory intensive,...), but also it is suggested that the execution is affected by the MPI tasks of rather different behaving kernels located in neighboring cores by the scheduler, which compete for resources (RAM and traffic). However, it is interesting to notice such a pattern (and criterium) related to grouping MPI tasks in fewer nodes, which seems to be more effective in saving execution time as the degree of parallelism and size (class) increase. On the contrary, computed cases of low number of tasks (see column of 4 MPI tasks in Fig. 4) show a small variation of the execution time (within 5–10%) with the number of nodes. This general trend observed in ACME, it is not so definite in EULER, which shows greater sensitiveness of the execution time (the equivalent figure is not included because of space constrains). The number of computed cases in EULER with some speedup when the MPI tasks are distributed among nodes dominates. Hence, EULER shows a somehow opposite behaviour compared to ACME (due to extension restrictions, its execution time plots are not included). As a result, how to proceed in EULER to speedup kernels execution is unclear and a more in depth kernel-by-kernel analysis seems necessary. A comparison of the different behaviour found in ACME and EULER in terms of the execution time for the two MG and EP kernels (memory- and CPU-intensive, respectively) is depicted in Fig. 5 for Dedicated Nodes setup. The plots for the MG kernel with classes B and C show that the speedup increases with monotone trend as more nodes are involved in the $nNxnT$ configuration ($nT = 4$, 8, 16 and 32). And it is seen that this speedup is significantly greater in EULER, which can be explained because EULER nodes are more memory-bound than ACME's. The EP kernel in ACME exhibits also a rather small speedup when tasks are distributed over nodes. But on the contrary, EULER shows that the EP kernel runs slower when it is taken "out of the box" (that is, when the tasks are partially migrated from all being grouped in one node). It is visible in Fig. 5 that a big jump in execution time occurs as the EP kernel goes from $1xnT$ to $2xnT$ (with $nT = 4$ and 8). It is noticed that the computed cases in EULER corresponding to $nT = 16$ and $nT = 32$ start at configurations 2×8 and 4×8, respectively, thus it is not possible to have evidence of the "out of the box" effect in these cases (but it is plausible that the wavy pattern observed in these be similar to the wavy one observed for $nT = 8$ from the 2×4 configuration on). In resume, the rule derived for the EP kernel in EULER is that MPI task grouping makes sense as the execution time drops. And besides, the reversed behaviour seen in ACME for the EP kernel can be justified because of the much higher internode bandwidth, which compensate the "out of the box" effect observed in EULER.

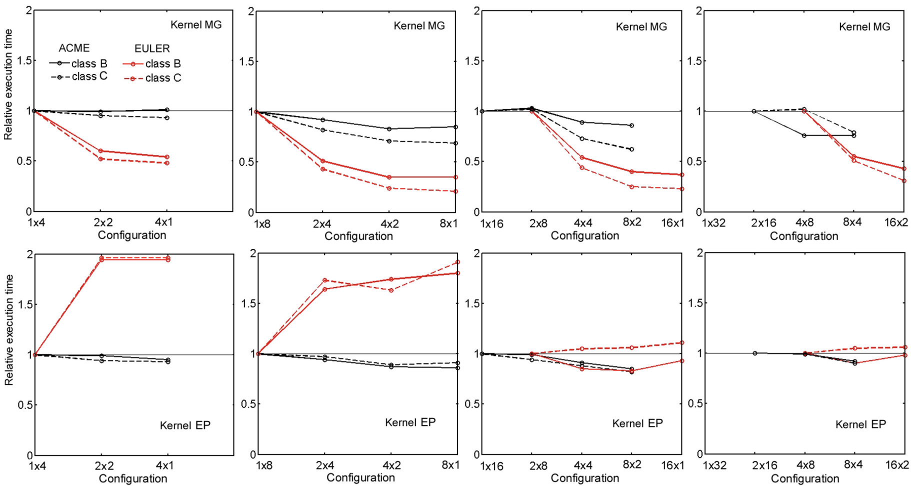


Fig. 5. Relative execution time for the Dedicated Nodes setup. NAS kernels MG (upper row) and EP (lower row) are shown for clusters ACME and EULER.

4.4 Sensitivity to the Clusters Setup

The performance of kernels MG and EP is plotted in Fig. 6 corresponding to 8 MPI tasks and the four clusters setups analysed for the classes A, B and C. This plot is relevant because it provides four $nNxnT$ configurations that start at 1×8 in both clusters, so the "out of the box" effect can be focused, if any. For both kernels, Dedicated Network and Dedicated Nodes setups provide very

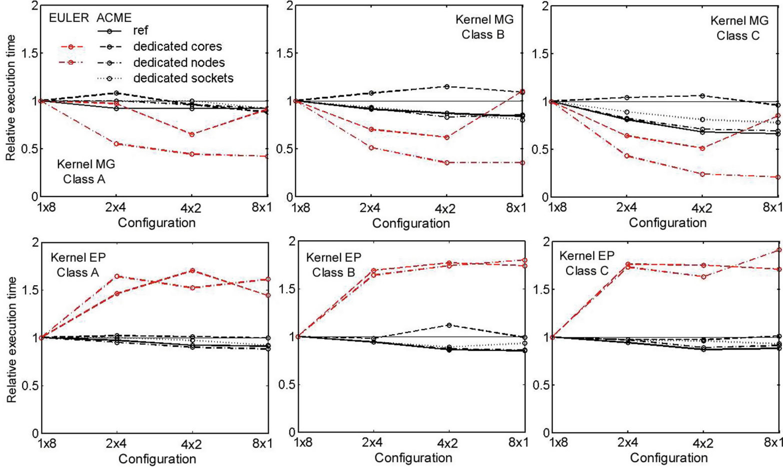


Fig. 6. Impact of cluster setup on the relative execution time for MG (memory-intensive) and EP (CPU-intensive) kernels with 8 MPI tasks in clusters.

similar execution time, showing a monotone, quasi-linear drop with the number of nodes, which suggests the benefit (but small) of adding nodes to the computation. The plot for Dedicated Sockets is similar, but even it is seen a smaller drop of the execution time. This pattern is observed for the MG and EP kernels with all classes (it is noted that Dedicated Network and Dedicated Sockets setups are included only for ACME since EULER is a production cluster and only a portion of it was assigned to this research). Again, for the MG kernel in EULER under Dedicated Nodes, it is observed a higher speedup with the number of nodes, compared to ACME. In particular, for the execution of the EP kernel in EULER, it is visible the "out-of-the-box" effect under both setups: a large increase of the execution time when the EP kernel goes from 1×8 to a 2×4 configuration, followed by a saturation of the drop when additional nodes are included. A different conclusion is derived for EULER: while for the memory-bound MG kernel is beneficial to distribute the MPI tasks over so many nodes as possible (the smaller host memory of its sockets may explain the significant improvement), the CPU-bounded EP kernel demands to group them into one node to attain the best performance. In resume, the MG and EP kernels under Dedicated cores setup in ACME, points out to the dominance of a performance drop (but not monotone) when additional nodes are added to the computation (the execution time shows a pattern of either a moderate increase of up to 20%, or a slight drop in some cases). Comparison of the speedup obtained with the Dedicated Nodes and Dedicated Cores setups for the whole set of experiments conducted in ACME and EULER is given in Figs. 7 and 8, respectively. The plotted boundary lines indicate the unused portion of the cluster due to the Slurm setup itself, which serves to build criterium about how much speedup is possible

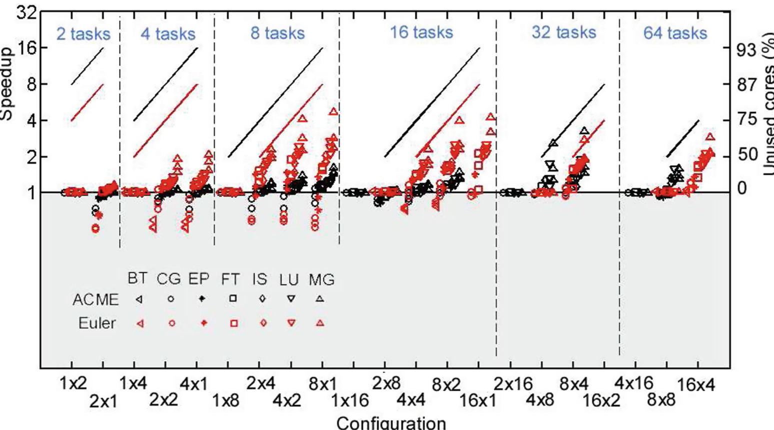


Fig. 7. Map of speedup vs. usage of computational resources for NAS under the Dedicated Nodes setup (locus of % of unused cores is plotted for each number of MPI tasks. Line color corresponds to the symbols), depicted for clusters ACME and EULER. (Color figure online)

and which is the extra cost due to not using a portion of the machine (e.g.: say an ACME 2×4 configuration with Dedicated Nodes setup. This implies 4 tasks running on a socket of 8 cores. Since each node has 2 sockets, the occupation reads 4 of a total of $8 + 8 = 16$ cores, which means a 75% of unused cores).

The vertical scales in the plots relate speedup and % of unused cores (i.e. speedup of 2 corresponds to a 50% of unused cores; speedup of 4 to a 75% of unused cores; and so on). This criterium remarks the importance of searching for a balance between significant speedups and not having too many unused cores. Obviously, it is a matter of settling a sweet point for users and cluster administrators. But under the Dedicated Nodes setup, it is seen that few points are over the boundary lines. Only in the Dedicated Cores setup, all boundary lines collapse into the 0%-unused cores situation (full occupation).

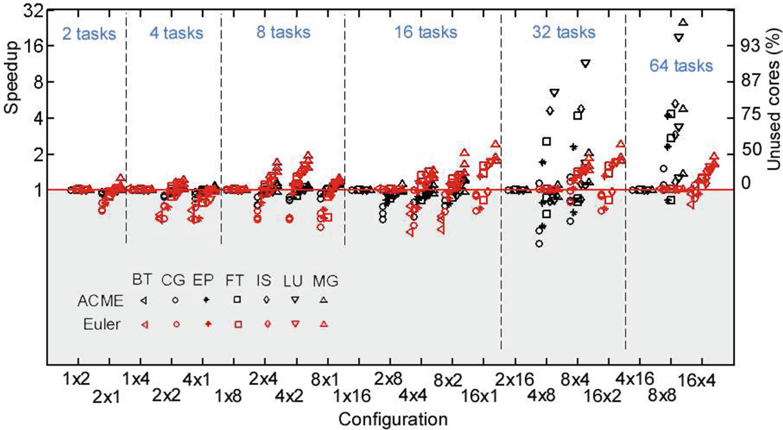


Fig. 8. Map of speedup vs. usage of computational resources for NAS under the Dedicated Cores setup depicted for clusters ACME and EULER.

5 Conclusions

The NAS Parallel Benchmarks have been executed in a systematic way on two clusters with rather different internode and intranode bandwidth properties, to identify dependencies between MPI tasks mapping and execution time speedup or resource occupation. The study comprises jobs up to 128 MPI tasks, bounded accordingly to our clusters size and usage constraints, as well as justified by the limited strong-scaling properties of NAS kernels.

The findings can be related to two scenarios. When the clusters are configured to run parallel jobs in exclusivity, the results show that in most cases the execution time drops as the MPI tasks are distributed over the nodes (this agrees with previous investigations) and it seems efficient to distribute a given parallel job over cores located in different nodes. However, the other important scenario

found in production HPC clusters corresponds to the need of sharing resources among set of jobs, such that the socket cores execute MPI tasks of different jobs. In this situation, a rather different behaviour is observed, much more sensitive to the type of cluster. In our state-of-the-art cluster ACME, many of the carried out experiments show a speedup when MPI tasks run in the fewest number of nodes. This is opposite to what is found in our older cluster EULER, where execution time trends are more sensitive to the algorithm properties and part of the experiments points out to task distribution over nodes to shorten the execution time. This major result found in ACME feeds the discussion about possible computational efficiency benefits by tailoring live tasks migration and scheduling policies in modern clusters. In production clusters which share a significant load of serial jobs while running parallel jobs (a 7-year analysis of the executed tasks in our cluster EULER showed that more than half were serial), the question of to which extent serial tasks may act as perturbations to the execution of parallel jobs arises and deserves consideration to clarify the best live task migration policies within the context of optimizing clusters occupation. Other interesting aspect is how different the results would be in the case of hybrid MPI/OpenMP tasks. These open issues are part of the ongoing research within our group.

Acknowledgment. This work was supported by the COST Action NESUS (IC1305) and partially funded by the Spanish Ministry of Economy and Competitiveness project CODEC2 (TIN2015-63562-R) and EU H2020 project HPC4E (grant agreement n 689772).

References

1. Top 500. www.top500.org
2. Jeannot, E., Mercier, G., Tessier, F.: Process placement in multicore clusters: algorithmic issues and practical techniques. IEEE Trans. Parallel Distrib. Syst. **25**(4), 993–1002 (2014)
3. Chavarría-Miranda, D., Nieplocha, J., Tipparaju, V.: Topology-aware tile mapping for clusters of SMPs. In: Proceedings of the 3rd Conference on Computing Frontiers (CF 2006), pp. 383–392. ACM (2006)
4. Smith, B.E., Bode, B.: Performance effects of node mappings on the IBM BlueGene/L machine. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 1005–1013. Springer, Heidelberg (2005). https://doi.org/10.1007/11549468_110
5. Rodrigues, E.R., Madruga, F.L., Navaux, P.O.A., Panetta, J.: Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: Proceedings of the IEEE Symposium on Computers and Communications, pp. 811–817 (2009)
6. Chai, L., Gao, Q., Panda, D.K.: Understanding the impact of multi-core architecture in cluster computing: a case study with Intel dual-core system. In: Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid, CCGrid, pp. 471–478 (2007)

7. Shainer, G., Lui, P., Liu, T., Wilde, T., Layton, J.: The impact of inter-node latency versus intra-node latency on HPC applications. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, pp. 455–460 (2011)
8. Xingfu, W., Taylor, V.: Using processor partitioning to evaluate the performance of MPI, OpenMP and hybrid parallel applications on dual- and quad-core Cray XT4 systems. In: Cray UG Proceedings (CUG 2009), Atlanta, USA, pp. 4–7 (2009)
9. Ribeiro, C.P., et al.: Evaluating CPU and memory affinity for numerical scientific multithreaded benchmarks on multi-cores. IJCSIS **7**(1), 79–93 (2012)
10. Wu, X., Taylor, V.: Processor partitioning: an experimental performance analysis of parallel applications on SMP clusters systems. In: 19th International Conference on Parallel Distributed Computing and Systems (PDCS 2007), CA, USA, pp. 13–18 (2007)
11. Wu, X., Taylor, V.: Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore. J. Comput. Syst. Sci. **79**(8), 1256–1268 (2013)
12. Zhang, C., Yuan, X., Srinivasan, A.: Processor affinity and MPI performance on SMP-CMP clusters. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and Ph.D. Forum (IPDPSW), Atlanta, USA, pp. 1–8 (2010)
13. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25 (1995)
14. OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks
15. Bonnie++. www.coker.com.au/bonnie++
16. Intel Memory Latency Checker 3.1. www.intel.com/software/mlc
17. Bailey, D., et al.: The NAS parallel benchmarks. Technical report (1994)

PRIMULA: A Framework Based on Finite Elements to Address Multi Scale and Multi Physics Problems

Alejandro Soba(✉)

CNEA - CONICET Centro Atómico Constituyentes,
Av. Gral. Paz 1499, San Martín, Argentina
soba@cnea.gov.ar

Abstract. The PRIMULA code, a multi scale and multithreads open source framework based on finite elements applicable to the numerical resolution of partial differential equations is presented. PRIMULA is portable to LINUX/UNIX, where it is compiled with gfortran, and to WINDOWS, compiled in the Visual Studio environment. It can be compiled to run in series, with shared memory under the Standard OPENMP, in a distributed environment under Standard MPI and on hybrid systems, with a compilation that combines MPI-OPENMP. The code was tested with non-linear problems in a 16 cores Intel Xeon (R) E5-2630 v3 multiprocessor of 2.4 GHz and in TUPAC, with 4 × Hexadeca core AMD Opteron 6276s processors. This paper presents results of scalability and computation times of some of the multiple tests to which it was submitted.

Keywords: PRIMULA · MPI-OPENP · FEM solver

1 Introduction

There are numerous software packages focused on problems based on differential equations in partial derivatives using the finite element method. The offer is wide for user looking for paid [13, 14], restricted [12] or free [11] licensed codes. Most of them consist of robust solvers that can be applied to the resolution of nonlinear, stationary or time-dependent problems, allowing them to approach numerical solutions of very complicated real systems. These packages usually support several parallelization modes that allow them to take advantage of the computing capacity of large computers, either distributed memory [22], shared [21] or lately with hybrid format, to profit the new generations of multithreads machines.

The geometries that these packages manipulate are also of great complexity. They are discretized using both structured and unstructured meshes with equal efficiency, to cover a wide range of problems from the spatial point of view, as well as numerical precision. In particular, this way of facing certain problems in simulation enters into what is ambiguously called multiphysics-multiscale. The multiplicity of analyzed physical problems as well as the different ranges of work scales supports this form of denomination. The method of finite elements [8], often combined with finite differences

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 139–153, 2018.
https://doi.org/10.1007/978-3-319-73353-1_10

and finite volumes, so versatile and adaptable to all kinds of geometries, has come to synergistically favour the limits of applicability of the packages mentioned in such a way that today it seems there is no physical problem that cannot be numerically dealt with.

Due not only to the cheapness of hardware but also to the growing importance of numerical simulation in various branches of science and technology, the acquisition of a computer with several thousand processors is an accessible goal even for developing countries, companies or scientific institutes at our national level. However, this supply of increasingly fast and powerful equipment is not accompanied by the development in equal measure and intensity of codes that take advantage of this availability. Rather, users often fall into the automatic use of packages already established in the market, either under paid or free license, but which show a certain robustness based on the number of groups in the world that use them or the number of publications that cite them. In some instances the scientific communities of certain countries can afford a supercomputer, but they do not seem to understand that such acquisition should be accompanied by a similar effort in time and money for the development of appropriate software, which would guarantee independence from the point of view of the generation and utilization of knowledge.

Currently in a computer like TUPAC [7], not only the administration tools are of foreign origin, but of free license [9, 10], but the 80% of the software that runs on the machine belongs to packages not developed by local groups. A great part of that time is dedicated to calculations with codes of first principle, of the so-called *ab initio* or similar, where by means of molecular dynamics or other approximate methods calculations at the atomic scale are performed. Also finite element codes like FLUENT [14], OPENFOAM [11], and the WRF-ARW [15] for climate analysis are employed in 80% of the studies. The remaining 20% uses more specific software, and only a minority of them written by local scientists.

Without diminishing merit to those groups that identify particular software and understand that this development is what they need for their research, this lack of local developments marks a trend towards dependence on the way in which scientific knowledge is manipulated. It also limits in a certain way the problems that can be faced. Due to the impossibility of adapting the code, a certain degree of simplification or idealization of the particular problem has to be assumed by the user so that the code can provide a solution.

In the Codes and Models Section of the Nuclear Fuel Cycle Management, CNEA we have started from a philosophy opposite to the one described above. Certainly, the type of institution we belong to and the character of the numerical challenges that the group needs to solve imposes that route. The confidentiality of certain problems as well as the difficulty of adapting these problems to the pre-existing packages in the market impelled us to develop our code at home, building the package that we have called DIONISIO [16–20], owned by CNEA. This code is devoted to the simulation of nuclear fuels behaviour during reactor operation, under normal and/or accident conditions. Until now it runs in desktop machines, with some computation sections parallelized under the standard Openmp [21]. DIONISIO is a multi-physic and multi-scale code, since it has models ranging from the level of a fuel grain, to the thermal-hydraulic description of heat removal by the coolant from a whole rod (typically some meters long).

The code has one, two and three-dimensional models, which operate in coupled form. However, the pretension to extend the analysis to the description of more realistic geometries, and to increase the dimension of the involved models, multiplies the computing time required. For this reason, the PRIMULA framework was designed. It is destined to solve coupled, non-linear differential equations, in two and three-dimensional geometries.

As the DIONSIO programmers and in general those of any type of application, are not experts in the writing of a distributed code, it is fundamental for the PRIMULA user that the framework structure remains hidden for him in such a way that he can parallelize his code without entering into the framework details. On the other hand, we intended to build a code that does not distinguish between operating systems (PRIMULA can be executed in a WINDOWS as well as in a LINUX environment without any further modification than, obviously, the way of compiling) or in which type of machine it is executed (it should be irrelevant for the user if the code is executed on a desktop machine or a supercomputer, so that it is transparent to serial or parallel use). Thus, through the modification of a couple of variables in the input file, the user decides whether the code will be executed serially or in parallel, and in the latter case, if the chosen parallelization will be over distributed or shared memory or a hybrid of both.

A final detail was considered in the design of PRIMULA: even for the simplest finite element packages users must be involved in programming certain details of the code to adapt it to their own particular problem. In many cases packages such as ABAQUS or OPENFOAM support writing and recompiling code segments. That is why today it is increasingly difficult to find pure users of a code, and in general, to achieve maximum software performance, the user becomes a programmer. Understanding this mutation of roles, PRIMULA also requires that the user actively intervene in the programming of certain code sectors. In the modification of the input and the output, as well as in the manipulation and creation of the variables that each user needs to introduce in his simulation, new programming is required. It should be noted that for each type of intervention, there are portions of code already written that facilitate the rapid learning of the novel programmer. Figure 1 shows a diagram of PRIMULA in which the separation between the systems to be solved and the kernel of the code is schematized. The IO of the same works through a series of input files that allow the selection of the different types of possible executions programmed so far in the kernel, as well as the manipulation of the system that the user wishes to solve.

PRIMULA is a code in development, which never seeks to consolidate in a fixed package, but is proposed as a framework that the user must adapt to his own particular problem and then continue with his own development. PRIMULA is completely versatile and has as one of its peculiarities of style (in its “to be delivered” state) to avoid FORTRAN language features that are not specific to the gfortran standard, this is, the most usual and common of the compilers of the moment. Each user, once he takes over the framework, can give him personal guidance related to the environment where he wants to execute the code under his own responsibility.

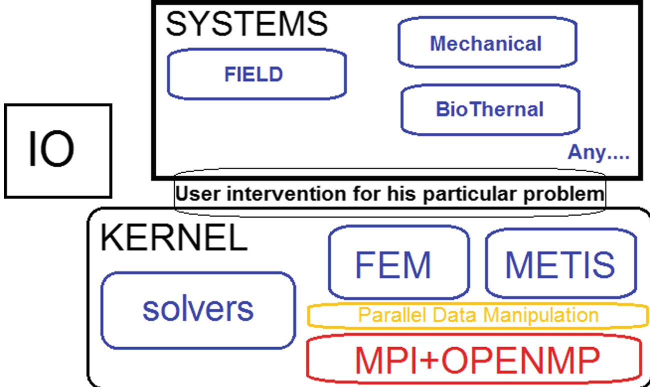


Fig. 1. Scheme of the engineering of the PRIMULA framework.

This work is proposed as a brief summary of the main characteristics of PRIMULA, in addition to a brief presentation of its capabilities, some results obtained and analysis of scalability and performance in the different environments where it was tested. Naturally, this report does not cover all the possible problems or tests to which it will be subjected in the near future, but is only intended to show the perspectives of its development.

2 PRIMULA General Features

i. **Mesh Generator:** The code has a built-in structured mesh generator, which allows meshing simple domains. The user can to modify the mesh and adapt it to others simples geometry. On the other hand the code can acquire meshes in files that agree with the preset format. The types of elements supported by the code are listed in Table 1. The integrations can be open or closed, with Gauss quadrant formulas for user defined points.

ii. **Mesh partition:** Work distribution is achieved by partitioning the original mesh into sub domains (sub meshes) that run concurrently by the corresponding MPI processes. Mesh partitioning is carried out using METIS [2]. The mesh partitioning is done sequentially by the master while the workers wait for receiving their parts of the mesh.
In order to establish the communication scheme between processes, a common graphing algorithm is used. [1, 3]: firstly the nodes-elements adjacency graph is created, then for each node the elements it belongs to are obtained. Secondly the adjacency elements graph is created, to know the neighboring elements of each element.
After that METIS uses the adjacency elements graph and the number of sub-domains and gives an array assigning the sub domain to each element.
Based on this information, the communication arrays needed to exchange data between the workers are created. Finally the communication scheduling needed to

Table 1. I: Bi and tri-dimensional elements programmed in PRIMULA.

Lagrangian bi-dimensional	Tri-dimensional	
Triangular three nodes	Tetrahedrons four nodes	Open or closed integration with triangular or tetrahedral coordinates.
Triangular six nodes	Tetrahedrons ten nodes	
Quadrangular four nodes	Hexahedrons eight nodes	Gaussian integration
Quadrangular nine nodes	Hexahedrons twenty seven nodes	
Quadrangular sixteen nodes		

determine in which order the workers have to interchange its data in pairs (through MPI_SendReceive) is computed. Once all this work is done, the distribution of the sub-meshes and corresponding element, boundary and node arrays to the corresponding workers is carried out (Fig. 2).

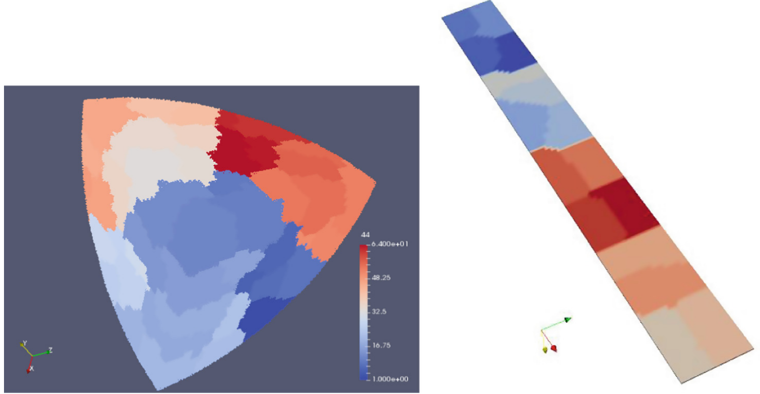


Fig. 2. Two tri-dimensional domains partitioned in 64 local regions using METIS.

iii. **Programmed systems:** PRIMULA is oriented to the resolution of stationary or time dependent equations in nonlinear partial derivatives. Field, bio-heat and mechanical equations for elasto-plasticity are presently codified (see Table 2), all types with Dirichlet, Neumann and Robin boundary conditions.
Problems can be solved in two and three dimensions. In the case of two dimensions, approximations of plane stress, plane strain and axy symmetry are possible.

Table 2. scheme of the main equations solved in PRIMULA until now

System	Unknow	Representative Equation
Field	Φ (Field variable)	$\nabla \cdot (k(\Phi))\nabla\Phi) + q = c(\Phi)\dfrac{\partial\Phi}{\partial t}$ (1)
Mechanical (Week form)	u (displacements)	$\int_{\Omega} \bar{e}^T \bar{\sigma} d\Omega = \int_{\Omega} f_{\Omega}^{Ext} \bar{u} d\Omega + \int_{\Gamma} f_{\Gamma}^{Ext} \bar{u} d\Gamma + R$ (2)
Bioheat (coupled with (1))	T (temperature)	$\nabla \cdot (k(T,\Phi)\nabla T) - a(T)(T - T_a) + + q + \lambda(\Phi, T)\lvert\nabla\Phi\rvert^2 = 0$ (3)

Where k, c, a and λ are the non linear parameters of each physical problem and q corresponds to the volumetric heat source term.

iv. **Linear Equation Systems Solution:** The solver of the linear system is a gradient conjugated with Jacobi preconditioner designed to work in hybrid systems [4, 5]. Among the iterative methods, the Conjugate Gradient method is an algorithm for the numerical solution of particular systems of linear equations, for which matrix (A) is symmetric and positive-definite. Every iteration of the algorithm requires only a single Sparse Matrix-Vector multiplication (SpMV) and a small number of dot products.

```
!$OMP PARALLEL DO                   &
!$OMP PRIVATE (jj, ...)             &
!$OMP SHARED  (glShiftRow, ... )    &
!$OMP ...
do ii = 1, lcRows
     jj = glShiftRow + ii
     r(ii) = b(ii)
     z(ii) = r(ii) * invDiag(ii)
     p(jj) = z(ii)
     inormB = inormB + b(ii)*b(ii)
     c = c + z(ii)*r(ii)
enddo                                 PARALLEL DO

scheme of comunication between processors
mpi_allreduce (inormB, c, SUM)
inormB = 1.0 / sqrt (inormB)

do while ((err > tolerance) .and. (iter < iterMax))
   call spmv (p, z)
   pz = 0.0

!$OMP PARALLEL DO                   &
!$OMP PRIVATE (jj, ...)             &
!$OMP SHARED  (glShiftRow, ... )    &
!$OMP ...
  do ii = 1, lcRows
     jj = glShiftRow + ii
     pz = pz + p(jj) * z(ii)
  enddo                               PARALLEL DO

   mpi_allreduce (pz, SUM)
   alpha = c / pz
   d = 0.0
   normR = 0.0

!$OMP PARALLEL DO                   &
!$OMP PRIVATE (jj, ...)             &
!$OMP SHARED  (glShiftRow, ...)     &
!$OMP ...
   do ii = 1, lcRows
      jj    = glShiftRow + ii
      x(ii) = x(ii) + alpha*p(jj)
      r(ii) = r(ii) - alpha*z(ii)
      z(ii) = r(ii) * invDiag(ii)
      normR = normR + r(ii)*r(ii)
      d = d + z(ii)*r(ii)
   enddo                              PARALLEL DO

   mpi_allreduce (normR,d, SUM)
   normR = sqrt (normR)
   beta = d/c

!$OMP PARALLEL DO                   &
!$OMP PRIVATE (jj, ...)             &
!$OMP SHARED  (glShiftRow, ...)
   do ii = 1, lcRows
      jj    = glShiftRow + ii
      p(jj) = z(ii) + beta*p(jj)
   enddo                              PARALLEL DO

   scheme of comunication between processors
   c = d
   iter = iter + 1
   err = normR*inormB
enddo
```

```
SUBROUTINE spmv (in b(:), out c(:))

  c = 0.0

!$OMP PARALLEL DO                        &
!$OMP ...
do ii = 1, lcRows
   c(ii) = 0.0
   do jj = ia(ii), ia(ii+1)-1
      c(ii) = c(ii) + aa(jj) * b(ja(jj))
   enddo
enddo                                      PARALLEL DO
```

Fig. 3. Hybrid algorithm of JPCG. One matrix vector operation (spmv) and two reduction of dot product are needed in each iteration. After each iteration the communication between processors are needed in order to update the values of the coefficients in shared nodes.

The storage requirements are also very modest, since vectors can be overwritten. In practice, this method often converges in far fewer than a number of iterations less than the order of the matrix A. However, the conjugate gradient can still converge very slowly if the matrix A is ill-conditioned. The convergence can often be accelerated by preconditioning. The choice of P is crucial in order to obtain a fast converging iterative method. The Jacobi preconditioner is one of the most popular forms of preconditioning, in which P is a diagonal matrix with

diagonal entries equal to those of A. The advantages of this preconditioner are the facility of its implementation and the low amount of memory it needs.

A hybrid version of the JPCG was implemented in this work (see Fig. 3), establishing the parallelization across MPI of the matrix vector multiplication and the dot product using the portion of the matrix that each processor locally has. The algorithm also allows the OPENMP strategy (highlighted in the windows) of distribution in the internal and local operations in each processor. After each iteration the communication between processors are needed in order to update the values of the coefficients in shared nodes.

v. **Post Process:** The entire post-process of the information generated in PRIMULA is adapted to the free distribution software PARAVIEW [6] under the data formats Comma Separate Values (CSV) and ENSI. ParaView is an open-source, multi-platform data analysis and visualization application. ParaView was developed to analyze extremely large datasets using distributed memory computing resources. It can be run on supercomputers to analyze datasets of petascale as well as on laptops for smaller data. The ParaView flexibility allows developers to quickly create applications that have specific functionality for a specific problem domain (Fig. 4).

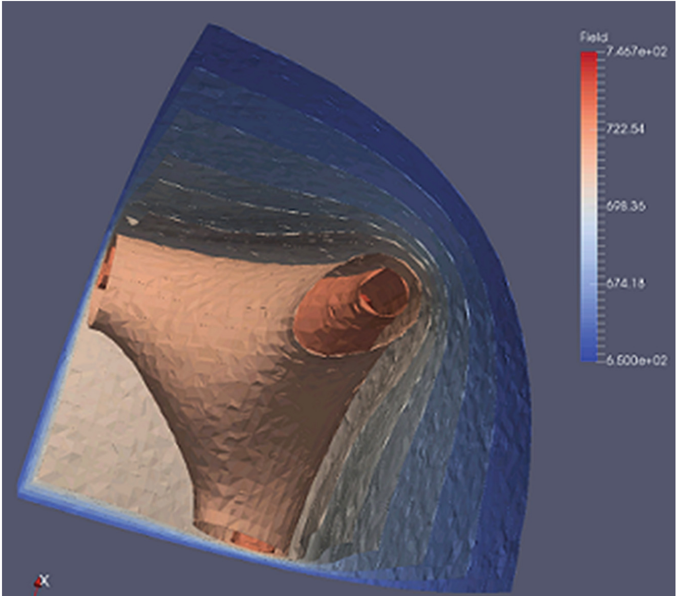


Fig. 4. A contours plot data from a field problem solved using PRIMULA.

vi. **Parallel environment:** The code is designed to be run on hybrid machines, taking advantage of the availability of multiprocessors that share their node with multiple threads. In the case of TUPAC, each node is a 4x Hexadeca core AMD Opteron

6276s, with 64 threads and 8 gigabytes of RAM. In this way a correct use of this computing capacity must have the possibility of combining algorithms that work in a distributed and shared form. To this end, the standards Message Passing Interface [22] and OPENMP [21] will be used. The user selects the parallelization mode he wants to use and the amount of threads and MPI processes needed for his problem through input files.

3 Scalability Results

Intel Xeon(R) E5-2630 v3 2.4 Hgz: Has 8 independent processing units (cores) and 2 threads per core, giving a total of 16 working threads. This allows using MPI and OPENMP or some combinations of both. In particular, to program PRIMULA in Windows OS we use FORTRAN in the VISUAL studio environment Visual Studio Pro 2012 [23]. Also we use MS-MPI [24] for distributed memory functionalities and OPENMP feature that by default contains the VS.
The main limitation that a code that handles large amounts of data possesses in the Windows environment is memory. Manipulating the variable STACK is inevitable if it is intended to work with a system of several million elements. On the other hand, there is a physical limit that can not be crossed and eventually works as a limiting problem to be established by RAM. Nevertheless, work in a desk computer with systems with several million of nodes efficiently, already is acceptable given the comfort they offer.

We have analyzed PRIMULA with a field problem (see Table 2) for domains with approximately one million variables obtaining some interesting results, plotted in Fig. 5 where the speedup for different code portions is showed. First of all, the solver of JPCG was always measured independently, since is the more demanding portion of the code. We plot too the pre-processing time, the general calculation zone (that includes the solver but also the time used in assembly the finite element matrix) and a measure of the total computing time of the code.

Analyzing these graphs, the first thing we observed is the region of super-scalability for the limit $p \leq 4$. It is interesting to note that the solver PJCG works optimally in this type of machines with a number of threads close to 4. From there the scalability is reduced to 86% with 8p and 58% with 16. The total computer time has an acceptable speedup up to 8 processors while the performance is greatly reduced to 16 where both the pre-process and the calculation saturate the computation times.

AMD OPTERON 6200: With 4 Hexadeca cores whereby each node has 64 threads. Two ways of executing the code were measured in this graph. First the application was executed in fully distributed mode, using pure MPI (plotted with dotted lines). Secondly, the code was run in hybrid form, with MPI + OPENMP, with 64 shared memory threads per node. We use an example of approximately 30 million elements for the calculation. There are several points in this comparison that are shown jointly in Figs. 6 and 7:

(i) The solver time in distributed mode has scalability similar to that of the hybrid mode. However it is remarkable how the distributed mode loses scalability in the section that we call calculus, which is the arming of matrices and calculation of all the coefficients by elements. Clearly the use of hybridization reduces the time consumed in communication between processes and accelerates the computation locally.

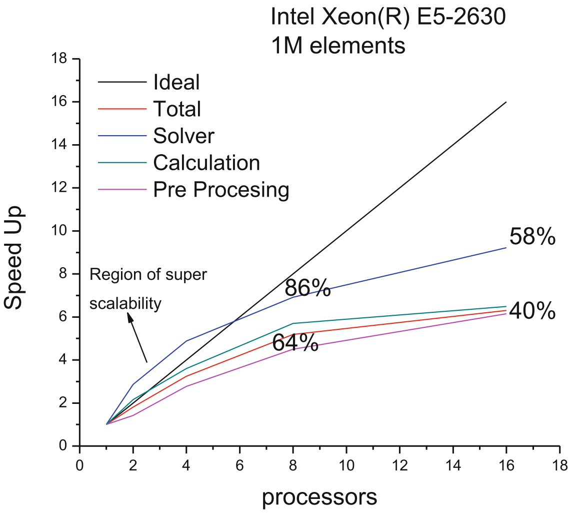


Fig. 5. Speed up for systems of 1M of elements approximately.

(ii) It is observed in the graph that for the pure MPI mode, we obtain compute times for 32 and 64 processors. However, it is not possible to obtain reasonable computing time for the hybrid case. This is because dividing the general domain (2 nodes) into two local parts, the amount of information of a mesh of 30 million degrees of freedom that must be transferred between nodes is so large that the calculation is notoriously slowing down. On the other hand, structures also grow in size considerably so memory management is also affected. In fact already for the case of 2 nodes (128 threads) the processing time increases disproportionately, (see Fig. 7). For the hybrid case, it improves the performance of the solver for 1024 processors in relation to 512. This is related to the improvement in communication that is established for 16 nodes with respect to that of 8 MPI nodes. The graph colouring algorithm shows that there is an optimal number of divisions to minimize communication in the global domain and in the case of the type of problem we are solving, this happens for that number of 16.

(iii) Figure 7 shows the pre-processing time each case consumes. We expect a similar time consumed in pre-processing for all the systems, with a grow of time consumed with the number of communication, but in case of 2 nodes plus 64 threads we highlight the fact mentioned in point (ii) about the time of post-processing in a hybrid case for low number of nodes. We have also added a point for 2048 processors. This point, although it degrades scalability

dramatically for the analyzed case, shows that PRIMULA works correctly for that number of processors. Finally, this graph tells us that the pre-processing times are lower for the hybrid case. This is for the same reason observed in item (i) in where we saw better calculation times in hybrid systems.

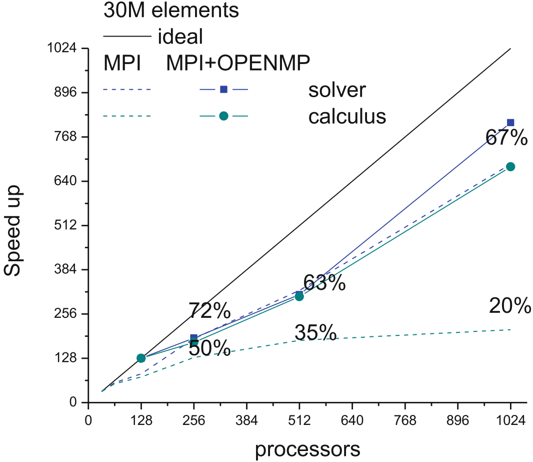


Fig. 6. Speed up for a system of 30 Millions of elements.

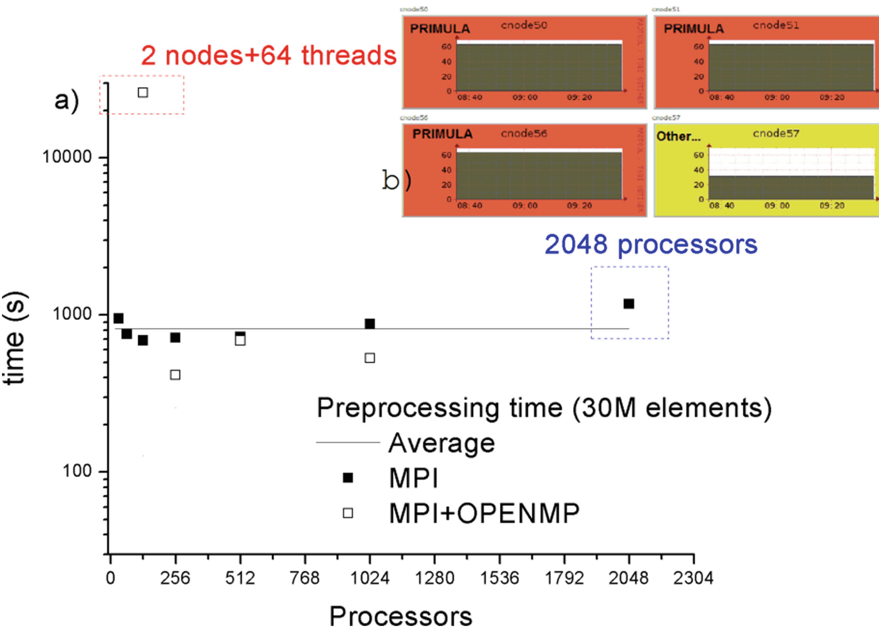


Fig. 7. (a) PRIMULA pre-processing times for a 30M system (approx). In the case 2 nodes and 64 threads the pre-processing time increases disproportionately due to the size of the generated arrays. (b) Use of processors per node in selected cores done by PRIMULA (Color figure online)

(iv) In Fig. 7(b) we presents some results performed by the monitoring system GANGLIA that analyze cluster behaviour on real time. We follow the local behaviour of PRIMULA running on the highlighted nodes. Is necessary to note that if an efficient use of the cluster is wanted, it is advisable to write applications that use the 64 threads of each node, since the access of each user to each node is individual. That's mean that if one user occupied one node alone using only 50% of the computer capacity (as in the case highlighted in yellow) is a waste of computing time with the consequent wasted cost in cash.

4 An Example of Field: PLATE Fuel

The fuels used by the vast majority of research reactors currently in operation are so-called dispersed fuels, which consist of particles of a uranium compound dispersed in a matrix of a metal (generally aluminium), which ensures a good extraction of heat of the combustible particle. This material thus formed is sheathed by co lamination between two metal plates (also generally aluminium) to be introduced into the reactor. This type of fuel is built by joining a number of plates into prismatic cavities that will occupy certain positions within the core. Other plate configurations may curve or form rings, always looking for an optimal neutron flux distribution, better dispersion of generated heat, plus adequate mechanical stability of the system. (See Fig. 8).

One of the most important codes that make up the DIONISIO package refers to the so-called PLACA3D, whose function is to simulate the behaviour of a combustible plate under irradiation in a research reactor in operation. The different phenomena that occur in each of the plate materials are motorized by temperature. It is generated inside the fissile material inside the plate and the heat is extracted through the passage of water by the outer coating, generally constructed of aluminium. Note that a combustible plate has a dimension of 1.4 mm thick, with a region of material fisil of 750 microns of thickness, 65 mm wide and 650 mm long, thus dimensionally establishing a complicated domain to solve due to the differences in scale of its geometry. On the other hand, physically interesting problems occur within the fissile region, due to being the zone of fission heat production, but in addition, the temperature on the plate must be determined through a water contour condition flowing at the rate of about 10 m per second. On the other hand, the plate is growing as time passes in the reactor a layer of oxide that functions as a thermal insulation. This complex problem with models at such dissimilar scales must be solved in more or less discrete time steps.

As the first field test of the PRIMULA framework we used the geometry of a discretized fuel plate with 1.256 million elements. The objective of this first test was to analyze the performance of the framework in a unique analysis of nonlinear temperature, to obtain the temperature distribution over the whole domain with Robin boundary conditions and a stationary generation of heat. Note that the complete history of a fuel within a reactor may range from 140 to 160 days of permanence, which is usually equivalent to a number of numerical steps of approximately 200.

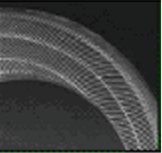


Fig. 8. Different types of fuel elements for research reactors: straight plates, curves or forming rings.

In this first estimation it was established that the calculation time by time step for a purely MPI computation with 64 processors (a TUPAC node) is approximately 36 s, so a complete history of this fuel element will oscillate between 2 and 2:30 h of computation. Figure 9 shows the temperature of the refrigerant, that obtained on the oxide layer that covers the plate and the temperature on the outside of the aluminium plate. In the inferior part the central temperature of the plate with the scale that appears in the same graph is presented. These values are correlated with knower experimental and analytical results.

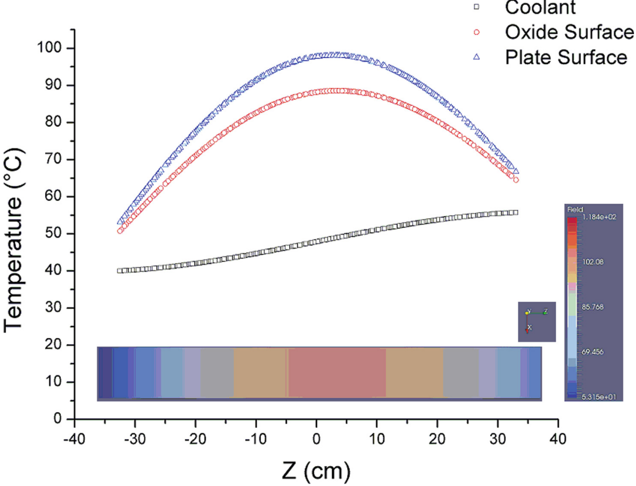


Fig. 9. Graph of temperature on the refrigerant, on the rust cover and on the aluminum cladding of the analyzed domain. The lower graph represents the central temperature of the plate.

5 Conclusions

Throughout this work we have given a first bounded description of the PRIMULA code, designed as a framework adaptable to problems of numerical resolution of differential equations in partial derivatives by the finite elements method. It responds to the usual denominations of multiscale and multiphysical, given the quality and precision with which it solves coupled problems of different dimensionalities and described physics through models of one two and three dimensions.

The code respects other interesting features such as:

(i) It works for any operating system with minimal modifications in its compilation.
(ii) It works in series or in parallel with minimal modifications in its input variables.
(iii) Works for bi and tri-dimensional geometries.
(iv) It is designed for any user to program their problem with a minimal intervention in the distribution of the particular information. That is, the user does not intervene in the distributed manipulation of its geometry or in the solver of its system.

Scalability analyzes have been performed for field problems, obtaining in different operative systems, results that justify its use given the savings in computing time that PRIMULA provides. Due to the characteristics of the problem analyzed and the type of solver used, a scalability that reaches 67% for 1024 processors is adequate.

The code is by no means finished and in fact as already warned in the introduction, it does not pretend to ever reach its final state. However there are some items in which we must continue working to make it a robust tool:

(i) Firstly, the supply of linear system solvers available in the code should be expanded. For this purpose, it is planned to include the possibility of using the Petsc package in addition to programs other solvers of the type of Krilov spaces with specific preconditioners optimized for distributed calculation.
(ii) The finite element types and the shapes functions offered in the code must be expanded. This is fundamental to more accurately encompass complicated domains.
(iii) The number of systems treated by PRIMULA should be increased considerably. We are working on the inclusion of field equations coupled with thermal phenomena and also on including equations of fluid mechanics such as the incompressible Navier Stokes equation. However the list is still short and this point should be a priority if PRIMULA is intended to compete with packages already established.

In short, the concept of PRIMULA will not be exploited in all its possibilities until the number of users grows.

Acknowledgment. I would like to thank CSC researchers and the TUPAC team of administrators for ongoing support they provided me during all of PRIMULA's writing, installation and scalability analysis tasks.

References

1. Artigues, A., Houzeaux, G.: Parallel mesh partitioning in alya. www.prace-ri.eu
2. Metis: Serial Graph Partitioning and Fill-reducing Matrix Ordering (metis documentation)
3. Coloración de Grafos. María Rosa Murga Días. Tesis de Grado. U. de Cantabria (2013)
4. Shewchuk, J.R.: An introduction to the conjugate gradient method without the agonizing pain. Carnegie Mellon University (1994)

5. Sáez, X., Soba, A., Sánchez, E., Kleiber, R., Castejón, F., Cela, J.M.: Improvements of the particle-in-cell code EUTERPE for petascaling machines. Comput. Phys. Commun. **182**(9), 2047–2051 (2011)
6. www.paraview.org
7. tupac.conicet.gov.ar/stories/home/
8. Zienkiewicz, O.C., Taylor, R.L.: The Finite Element Method, vol. 1, 2 & 3. Butterworth Heinemann, Oxford (2000)
9. slurm.schedmd.com/
10. ganglia.sourceforge.net/
11. www.openfoam.com/
12. www.bsc.es/es/computer-applications/alya-system
13. www.3ds.com/products-services/simulia/products/abaqus/
14. www.ansys.com/Products/Fluids/ANSYS-Fluent
15. www.wrf-model.org/index.php
16. Soba, A., Denis, A.: Simulation with DIONISIO 1.0 of thermal and mechanical pellet-cladding interaction in nuclear fuel rods. J. Nucl. Mater. **374**, 32–43 (2008)
17. Lemes, M., Soba, A., Daverio, H., Denis, A.: Inclusion of models to describe severe accident conditions in the fuel simulation code DIONISIO. Nucl. Eng. Design **315**, 1–10 (2017)
18. Soba, A., Denis, A., Lemes, M., González, M.E.: Modelado del comportamiento del combustible nuclear bajo irradiación mediante DIONISIO 2.0 Revista de la CNEA, Vol. 53–54 (2014)
19. Soba, A., Denis, A., Romero, L., Villarino, E., Sardella, F.: A high burnup model developed for the DIONISIO code. J. Nucl. Mater. **433**, 160–166 (2013)
20. Soba, A., Denis, A.: PLACA/DPLACA: código para la simulación de un combustible tipo placa monolítico/disperso. Rev. Int. Mét. Num. Cál. Dis. Ing. **23**(2), 205–224 (2007)
21. www.openmp.org/
22. www.open-mpi.org/
23. Visual_Studio_Pro_2013. https://www.visualstudio.com/es/ (License 62739385 COM.NAC. DE ENERGÍA ATÓMICA)
24. msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx

FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC

Josef Spillner[1(✉)], Cristian Mateos[2], and David A. Monge[3]

[1] School of Engineering - Service Prototyping Lab,
Zurich University of Applied Sciences, 8401 Winterthur, Switzerland
josef.spillner@zhaw.ch
[2] ISISTAN-UNICEN-CONICET,
Campus Universitario, Paraje Arroyo Seco, 7000 Tandil, Buenos Aires, Argentina
cristian.mateos@isistan.unicen.edu.ar
[3] ITIC Research Institute, National University of Cuyo,
Padre Jorge Contreras 1300, M5502JMA Mendoza, Argentina
dmonge@uncu.edu.ar
https://blog.zhaw.ch/icclab/

Abstract. The adoption of cloud computing facilities and programming models differs vastly between different application domains. Scalable web applications, low-latency mobile backends and on-demand provisioned databases are typical cases for which cloud services on the platform or infrastructure level exist and are convincing when considering technical and economical arguments. Applications with specific processing demands, including high-performance computing, high-throughput computing and certain flavours of scientific computing, have historically required special configurations such as compute- or memory-optimised virtual machine instances. With the rise of function-level compute instances through Function-as-a-Service (FaaS) models, the fitness of generic configurations needs to be re-evaluated for these applications. We analyse several demanding computing tasks with regards to how FaaS models compare against conventional monolithic algorithm execution. Beside the comparison, we contribute a refined FaaSification process for legacy software and provide a roadmap for future work.

1 Research Direction

The ability to turn programmed functions or methods into ready-to-use cloud services is leading to a seemingly serverless development and deployment experience for application software engineers [1]. Without the necessity to allocate resources beforehand, prototyping new features and workflows becomes faster and more convenient to application service providers. These advantages have given boost to an industry trend consequently called Serverless Computing. The more precise, almost overlapping term in accordance with Everything-as-a-Service (XaaS) cloud computing taxonomies is Function-as-a-Service (FaaS) [4]. In the FaaS layer, *functions*, either on the programming language level or

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 154–168, 2018.
https://doi.org/10.1007/978-3-319-73353-1_11

as abstract concept around binary implementations, are executed synchronously or asynchronously through multi-protocol triggers. Function instances are provisioned on demand through coldstart or warmstart of the implementation in conjunction with an associated configuration in few milliseconds, elastically scaled as needed, and charged per invocation and per product of period of time and resource usage, leading to an almost perfect pay-as-you-go utility pricing model [11]. FaaS is gaining traction primarily in three areas. First, in Internet-of-Things applications where connected devices emit data sporadically. Second, for web applications with light-weight backend tasks. Third, as glue code between other cloud computing services. In contrast to the industrial popularity, no work is known to us which explores its potential for scientific and high-performance computing applications with more demanding execution requirements.

From a cloud economics and strategy perspective, FaaS is a refinement of the platform layer (PaaS) with particular tools and interfaces. Yet from a software engineering and deployment perspective, functions are complementing other artefact types which are deployed into PaaS or underlying IaaS environments. Figure 1 explains this positioning within the layered IaaS, PaaS and SaaS service classes, where the FaaS runtime itself is subsumed under runtime stacks. Performing experimental or computational science research with FaaS implies that the two roles shown, end user and application engineer, are adopted by a single researcher or a team of researchers, which is the setting for our research.

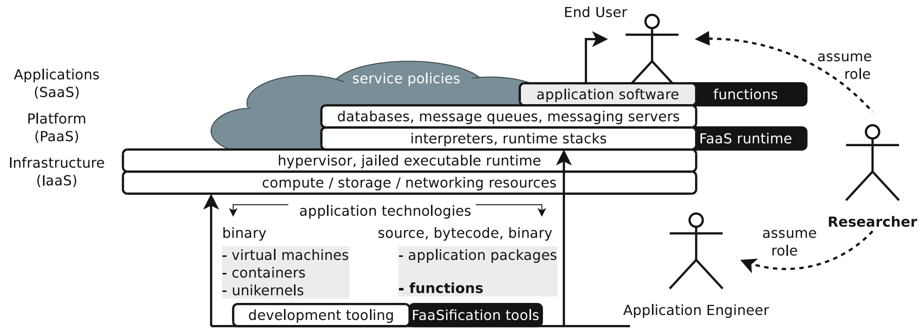


Fig. 1. Positioning of FaaS in cloud application development

The necessity to conduct research on FaaS for further application domains stems from the unique execution characteristics. Service instances are heuristically stateless, ephemeral, and furthermore limited in resource allotment and execution time. They are moreover isolated from each other and from the function management and control plane. In public commercial offerings, they are billed in subsecond intervals and terminated after few minutes, but as with any cloud application, private deployments are also possible. Hence, there is a trade-off between advantages and drawbacks which requires further analysis. For example, existing parallelisation frameworks cannot easily be used at runtime

as function instances can only, in limited ways, invoke other functions without the ability to configure their settings. Instead, any such parallelisation needs to be performed before deployment with language-specific tools such as Pydron for Python [10] or Calvert's compiler for Java [3]. For resource- and time-demanding applications, no special-purpose FaaS instances are offered by commercial cloud providers. This is a surprising observation given the multitude of options in other cloud compute services beyond general-purpose offerings, especially on the infrastructure level (IaaS). These include instance types optimised for data processing (with latest-generation processors and programmable GPUs), for memory allocation, and for non-volatile storage (with SSDs). Amazon Web Services (AWS) alone offers 57 different instance types. Our work is therefore concerned with the assessment of how current generic *one-size-fits-all* FaaS offerings handle scientific computing workloads, whether the proliferation of specialised FaaS instance types can be expected and how they would differ from commonly offered IaaS instance types. In this paper, we contribute specifically (i) a refined view on how software can be made fitting into special-purpose FaaS contexts with a high degree of automation through a process named *FaaSification*, and (ii) concepts and tools to execute such functions in constrained environments.

In the remainder of the paper, we first present background information about FaaS runtimes, including our own prototypes which allow for provider-independent evaluations. Subsequently, we present four domain-specific scientific experiments conducted using FaaS to gain broad knowledge about resource requirements beyond general-purpose instances. We summarise the findings and reason about the implications for future scientific computing infrastructures.

2 Background on Function-as-a-Service

2.1 Programming Models and Runtimes

The characteristics of function execution depend primarily on the FaaS runtime in use. There are broadly three categories of runtimes:

1. Proprietary commercial services, such as AWS Lambda, Google Cloud Functions, Azure Functions and Oracle Functions.
2. Open source alternatives with almost matching interfaces and functionality, such as Docker-LambCI, Effe, Google Cloud Functions Emulator and OpenLambda [6], some of which focus on local testing rather than operation.
3. Distinct open source implementations with unique designs, such as Apache OpenWhisk, Kubeless, IronFunctions and Fission, some of which are also available as commercial services, for instance IBM Bluemix OpenWhisk [5]. The uniqueness is a consequence of the integration with other cloud stacks (Kubernetes, OpenStack), the availability of web and command-line interfaces, the set of triggers and the level of isolation in multi-tenant operation scenarios, which is often achieved through containers.

In addition, due to the often non-trivial configuration of these services, a number of mostly service-specific abstraction frameworks have become popular among developers, such as PyWren, Chalice, Zappa, Apex and the Serverless Framework [8]. The frameworks and runtimes differ in their support for programming languages, but also in the function signatures, parameters and return values. Hence, a comparison of the entire set of offerings requires a baseline.

The research in this paper is congruously conducted with the mentioned commercial FaaS providers as well as with our open-source FaaS tool *Snafu* which allows for managing, executing and testing functions across provider-specific interfaces [14]. The service ecosystem relationship between Snafu and the commercial FaaS providers is shown in Fig. 2. Snafu is able to import services from three providers (AWS Lambda, IBM Bluemix OpenWhisk, Google Cloud Functions) and furthermore offers a compatible control plane to all three of them in its current implementation version. At its core, it contains a modular runtime environment with prototypical maturity for functions implemented in JavaScript, Java, Python and C. Most importantly, it enables repeatable research as it can be deployed as a container, in a virtual machine or on a bare metal workstation. Notably absent from the categories above are FaaS offerings in e-science infrastructures and research clouds, despite the programming model resembling widely used job submission systems. We expect our practical research contributions to overcome this restriction in a vendor-independent manner. Snafu, for instance, is already available as an alpha-version launch profile in the CloudLab testbed federated across several U.S. installations with a total capacity of almost 15000 cores [12], as well as in EGI's federated cloud across Europe.

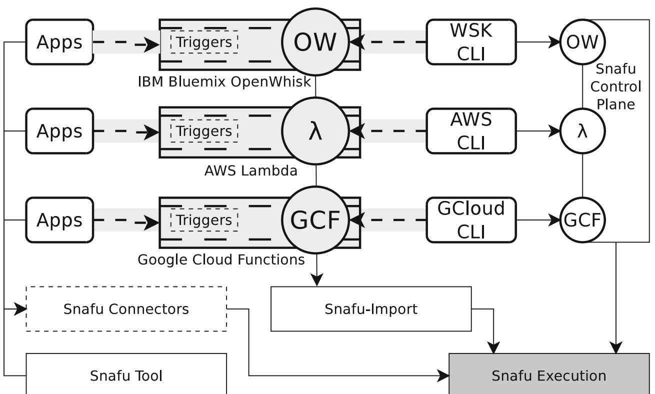


Fig. 2. Snafu and its ecosystem and tooling

Using Snafu, it is possible to adhere to the diverse programming conventions and execution conditions at commercial services while at the same time controlling and lifting the execution restrictions as necessary. In particular, it is possible to define memory-optimised, storage-optimised and compute-optimised

execution profiles which serve to conduct the anticipated research on generic (general-purpose) versus specialised (special-purpose) cloud offerings for scientific computing. Snafu can execute in single process mode as well as in a load-balancing setup where each request is forwarded by the master instance to a slave instance which in turn executes the function natively, through a language-specific interpreter or through a container. Table 1 summarises the features of selected FaaS runtimes.

Table 1. FaaS runtimes and their features

Runtime	Languages	Programming model	Import/export
AWS Lambda	JavaScript, Python, Java, C#	Lambda	–
Google Cloud Functions	JavaScript	Cloud Functions	–
IBM Bluemix OpenWhisk	JavaScript, Python, Swift, Docker	OpenWhisk	–
Fission	JavaScript, Python, Go, C#, PHP	Fission	–
Kubeless	JavaScript, Python	Kubeless	–
Snafu	JavaScript, Python, C, Java	Lambda, OpenWhisk, Cloud Functions	Lambda, OpenWhisk, Cloud Functions, Fission

2.2 Providers and Performance

Commercial FaaS offerings differ not only in their programming conventions, but also vastly by performance, cost and the combined utility defined as cost-duration product. Figure 3 informs about the benchmark of a compute-intensive function, $fib(38)$, implemented in Python as a *portable hosted function* which runs on bare metal as well as in AWS Lambda, IBM Bluemix OpenWhisk and Azure Functions. The performance in the FaaS environments will unlikely be faster than an approximate performance barrier indicated by the benchmark's runtime with the fastest widely available processor cores, exemplified by a 9.5 s execution time on a recent Intel Xeon E5-2660 v3 (Haswell) with 2.6 GHz and 8.5 s on an Intel i7-4800MQ with 2.7 GHz. Conversely, it is often much slower, and the offerings do not allow for explicitly paying more for better performance. The rather odd (although even) number 38 has thus been chosen so that on the slowest commercial FaaS offering, Lambda with 128 MB instances, the function terminates successfully before the obligatory five minutes timeout.

Lambda performs proportional to the memory assignment, thus leading to a constant price. OpenWhisk raises the price proportional to the memory assignment while keeping a constant performance. In contrast, Azure measures the

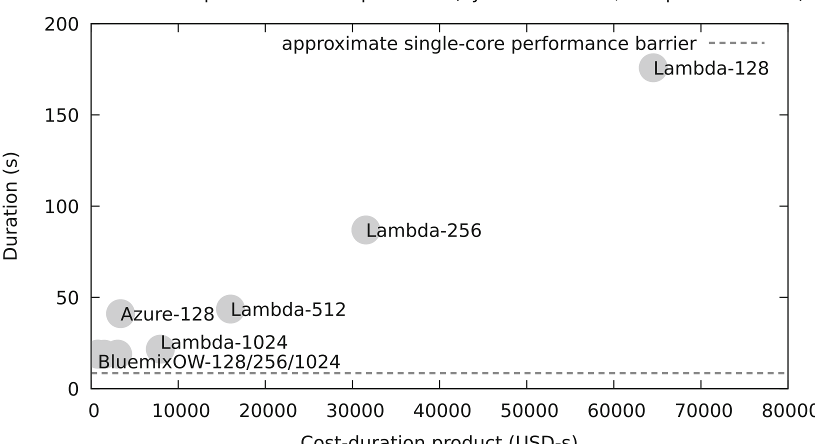


Fig. 3. CDP benchmark of FaaS providers, averaged over ten iterations, without free tier discounts

memory use but does not allow for its configuration. Finally, while Google Cloud Functions do not appear in the same diagram due to its limitation to JavaScript, its pricing model almost resembles the one of Lambda although the high-memory instances performance decreases in relation. The compute-intensive recursive benchmark implementation is shown in Listing 1.1. Functions with other characteristics will be analysed in detail in the next section.

Listing 1.1. Compute-intensive portable hosted Fibonacci function in Python

```
import time
def fib(x):
    if x in (1, 2):
        return 1
    return fib(x - 1) + fib(x - 2) # recursion
# AWS Lambda entry point
def lambda_handler(event, context):
    return fib(38)
# IBM Bluemix OpenWhisk entry point
def main(event):
    return {'ret': fib(38)}
# Microsoft Azure entry point (could be conditional)
import os
datain = open(os.environ['req']).read()
response = open(os.environ['res'], 'w')
response.write(str(fib(38)))
response.close()
```

3 Scientific Computing Experiments with Functions

In order to get a broad understanding of the feasibility and utility of FaaS models for diverse computing tasks, four experiments are conducted to compare the performance and other resource-related characteristics. The selected domains are mathematics (calculation of π), computer graphics (face detection), cryptology (password cracking) and meteorology (precipitation forecast). The first three experiments are synthetic, while the fourth one uses FaaSification to analyse an existing non-FaaS application. We will refer to the domain and function execution characteristics to infer statements about the possible and desirable degree of parallelisation without resource contention.

3.1 Mathematics: Calculation of π

A common formula to calculate arbitrary digits of π in parallel sequences for unlimited precision is Baily-Borwein-Plouffe (BBP). The implementation in Python uses the *Decimal* type explicitly due to the otherwise limited precision of built-in floating point numbers, which contrasts the unlimited digits of built-in integer numbers. The experiment is set up to calculate 2000 digits with a theoretic precision of 10000 digits.

Figure 4 compares the BBP calculation performance between the native Python 3 execution, the optimised (JiT-compiled) PyPy 3 execution, as well as the FaaS equivalents with in-process and out-of-process parallelisation, or multi-threading and multi-processing, respectively.

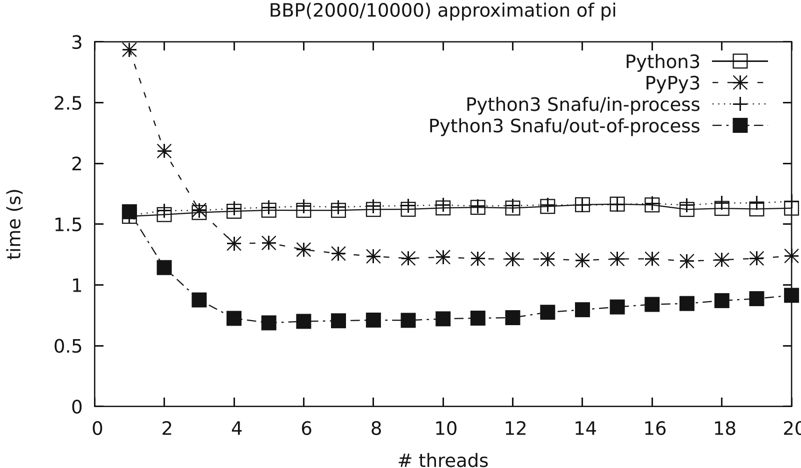


Fig. 4. Comparison of BBP(2000/10000) implementation performance with Python/PyPy 3.5

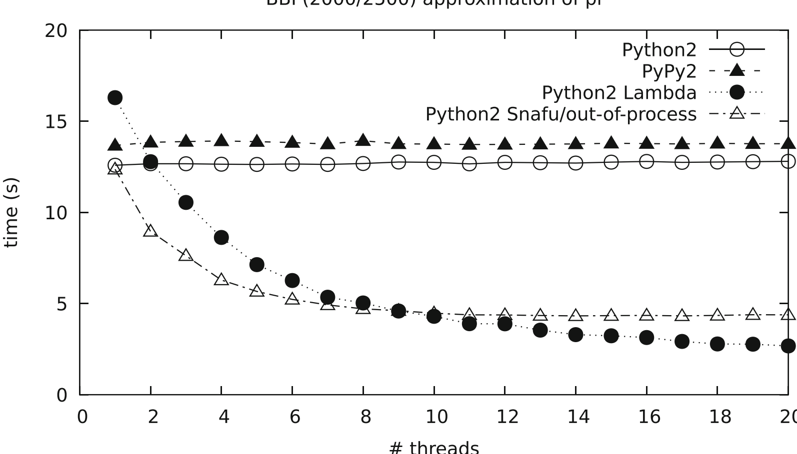


Fig. 5. Comparison of BBP(2000/2500) implementation performance with Python 2.7

Figure 5 shows the equivalent performance measurements when running all implementations with Python 2. While this version is in maintenance mode and any development of the language and libraries will cease around the year 2020, it is still widely used, most prominently in the AWS Lambda service which is one of the few commercial FaaS services offering a native Python runtime. An interesting observation is that the programming language version matters. While PyPy 3 scales much better than the corresponding CPython implementation for a number of threads equal to or larger than the number of physical cores, PyPy 2 shows no such behaviour. In both cases, an external out-of-process function execution or hosted function execution is faster despite network transmission, due to overcoming multi-threading restrictions in the Python interpreters. Consequently, using compute-optimised FaaS is beneficial from a performance point of view when multi-processing, especially at scale, is not an option.

3.2 Computer Graphics: Face Detection

Face detection and recognition have become widely used techniques in social networks, robotics and photo management applications, but also in surveillance networks. The OpenCV library is among the commonly used tools to perform face detection and mark or extract the corresponding sections in photos. Figure 6 shows an example of a person's face detected and marked by OpenCV in a lake scenery photo. This experiment performs the same detection and marking on a large number of photos.

A reference dataset with faces is provided by Faces in the Wild [2] which serves as useful input for input-output-centric file processing. The dataset contains 30281 images with a total size of 1.4 GB.

Fig. 6. Face detection and marking in photos using OpenCV with Python 2.7

As OpenCV is not yet widely available for Python 3, the implementation is based on Python 2. Figure 7 shows the performance measurements for the function of the face detection using the public dataset again using pure local execution as well as function execution in Snafu and Lambda. Due to the I/O-centric nature of the function, the Lambda performance lags significantly and only becomes competitive when 19 or more threads are used. With in-function parallelisation, this effect can be remedied so that Lambda already executes faster with 10 threads or more. Slower Lambda functions are furthermore forcibly terminated due to reaching the timeout barrier.

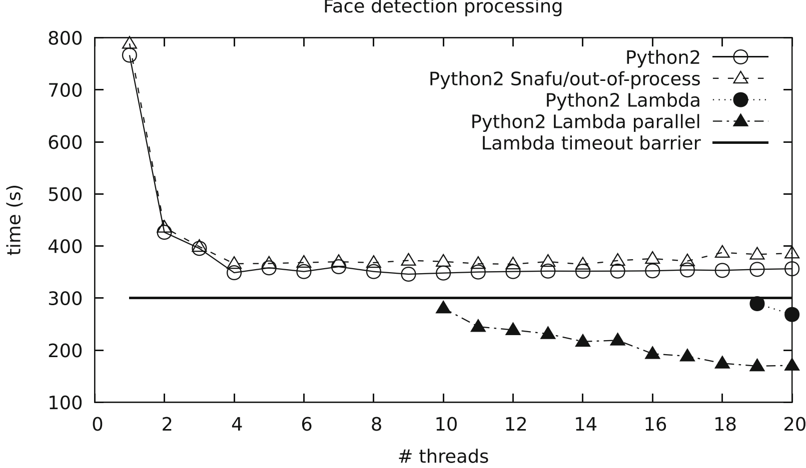


Fig. 7. Comparison of FacesDetect implementation performance with Python 2.7

The I/O lag in Lambda results from the use of the S3 object storage service compared to direct file system access in the other cases. This discrepancy makes

evident the need for research tools beyond just the FaaS execution, as functions are often interlocked with additional cloud services. First prototypes such as the Atlassian LocalStack which simulates AWS services are already available [7]. Furthermore, the latency could be reduced by deploying function instance-affine local storage, preferably in memory, a storage-optimised facility not yet offered commercially by any of the FaaS providers.

3.3 Cryptology: Password Cracking

Digital forensics tools include functionality to crack passwords based on leaked or otherwise published hashes. Research on secure hashes ensures that only a brute-force attempt to crack them will succeed. Parallelisation and map-reduce operations are helpful to speed up the process.

The associated experiment consists of 100 SHA256-hashed passwords of up to 3 characters for which all possible combinations are tried in a comparison. The implementation makes use of two layers of parallelisation: Conventional parallel processing of up to 10 workers dividing the passwords among them, always returning a single result, and map-reduce processing of up to 10 mappers dividing the character ranges per character, returning the first successful match. The implementation in Python further makes use of the built-in concurrency framework which provides a unified futures interface for multi-threading and multi-processing. Due to the compute-centric algorithm and Python's global interpreter lock, the use of multi-threading does not lead to speed-ups. Hence, multi-processing is compared with a multi-function mode called *function futures* contributed by us which outsources the function and mapping calls to Lambda similar to the processing mode of PyWren [8].

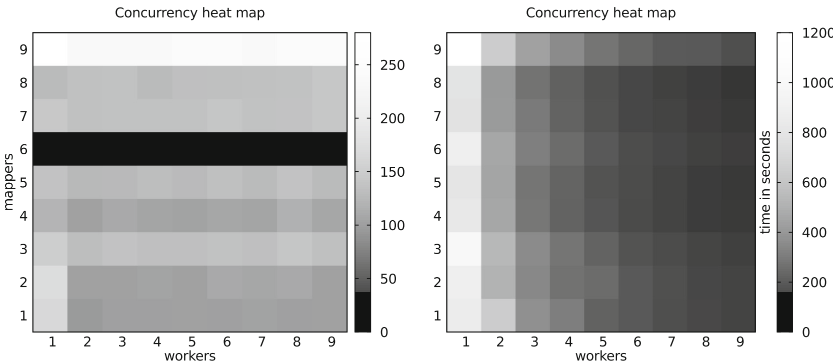


Fig. 8. Cracking of 100 passwords through a double-map-reduce process with combinations of workers and mappers; left: local multi-processing; right: multi-function on Lambda (darker is faster)

Figure 8 compares the local multi-processing execution and the multi-function execution on Lambda. The Lambda setup consists of 512 MB of

memory assignment to each function instance which affects the absolute performance. This is why the z-axis (brightness scale) has been normalised between the graphs. Comparing both results, the linear worker scaling is evidently more predictable whereas the mapper scaling is not contributing due to the overhead compared to the tiny processing spans of each mapper. Furthermore, the local system exposed a bug in Python's multi-processing code which led to some combinations fail sometimes, any with 6 mappers fail consistently, and any with 9 mappers resulting in a runtime almost twice as high as the preceding ones.

3.4 Meteorology: Precipitation Forecast

Weather forecasting is a typical use case for supercomputing environments. The forecast of precipitation in particular relies on many different models, including heuristic and fuzzy ones, which are parameterised with a multitude of variables and hyperparameters. However, while some forecasts run continuously, specialised forecasts only need to be run at certain times at full scale with high re-use factor of smaller proven functions, which could mean that FaaS is a suitable deployment model for this domain.

The experiment consists of running an implementation which confirms precipitation data from the Buenos Aires and Mendoza metropolitan regions in Argentina. It is based on neuronal nets implemented in Python, the Keras library and TensorFlow. As it is an existing application which can not readily be deployed into a FaaS environment, the research interest shifts to an earlier stage in the service provisioning lifecycle, to the pre-deployment software development. Tools are required which transform existing code into functions in conformance with the programming conventions expected by the target provider. The transformation process is consequently called FaaSification. In the experiment in question, several monolithic functions are used which need to be subdivided into several functions as part of this process to not exceed the FaaS providers' execution time limits. We have designed and implemented a tool called function splitter for evaluating the feasibility to split Python functions automatically. The tool employs the concept of *worm functions*. While a function instance may be terminated early due to a timeout, any other instance invoked by it just before the timeout will not be affected and will get its own counter. Thus, a function instance's state can be carried on to another instance of the same or another, sequentially related, partial function. The function splitter traces the use of local variables in a first partial function F_1 and adds them to the signature of the subsequent partial function F_2. The concept is explained with an example in Listing 1.2 which shows the split after line 1 of the original function F which contains a call to another function G.

Figure 9 shows the performance of the forecast function F split into two partial functions, F_1 and F_2, with an increasing number of lines in F_1 instead of F_2. One interesting observation is that independent from the division into two functions, the runtime of each successive function instance increases substantially in the long term. A second observation is the pivotal point in which F_1 almost swaps its execution time with F_2, which happens in line 18 of the code.

Listing 1.2. Example for function splitting based on worm functions

```
# original                  # partials
def F(x):                   def F1(x):
    y = x + 1                   y = x + 1
    ————————                    return F2(x, y)
    ————————                def F2(x, y):
    z = y - 2                   z = y - 2
    z += G(x)                   z += G(x)
    return z                    return z
```

This effect is a sure signal of a call to another function, either local or remote, which in turn needs to be subdivided to bring the partial function runtimes below the acceptable limits, corresponding to the call to G in the example. Currently, even the fastest partial execution time of F_2, which is 289 s, would almost reach Lambda's 300 s limit, and some even exceed Cloud Function's 540 s limit.

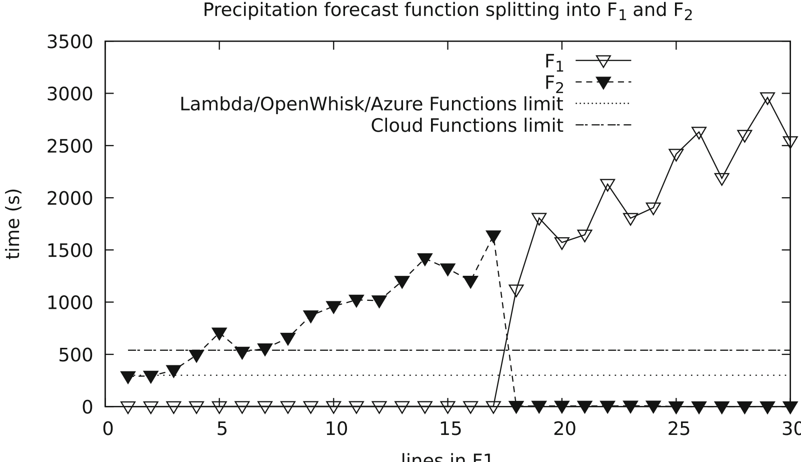


Fig. 9. Characteristics of forecast function splitting

The function splitter has been integrated with Lambada, a FaaSification tool for Python which is publicly available [13], which otherwise had performed a 1:1 translation of functions in the code to hosted functions in FaaS environments.

4 Findings

A general observation is that despite the still immature programming and deployment models for FaaS, experimental implementations from four different scientific computing domains have been successfully executed on both commercial and self-hosted FaaS runtimes.

The computing requirements of the four domains differ significantly with respect to the utilisation of compute, storage and network resources. Figure 10 gives an exemplary insight into the processing as well as disk and network input-output characteristics of the face detection function from the computer graphics domain and the password cracking function from the cryptology domain. Heavy spikes of usually 5 MB/s read and write operations, peaking in more than 9 MB/s, can be observed in the first, and low CPU use due to network I/O waiting in the second.

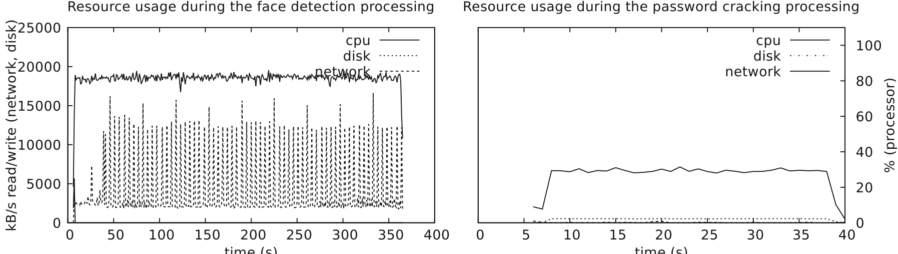


Fig. 10. Resource characteristic of two selected functions

Apart from the runtime, the findings also cover the development time. Through our work on scientific applications, we are now able to suggest the following refined classification of FaaSification process levels.

– *Shallow FaaSification*: classes or function collections divided into corresponding FaaS units. Functions or methods are the atomic units on this level.
– *Medium FaaSification*: functions divided by lines into regrouped or split FaaS units. Lines of code are the atomic units on this level.
– *Deep FaaSification*: single lines divided into multiple FaaS units or parameterised FaaS instances. Instructions are the atomic units on this level.

Existing function deployment and execution implementations cover these levels to various degrees, calling for future work towards deep FaaSification for special-purpose FaaS instances. The function execution environment Snafu performs a simple shallow FaaSification for different programming languages assuming single-file implementations. Existing transformation tools such as the triaged Lambada, but also structurally similar ones such as Podilizer or Termite, perform a thorough shallow FaaSification which also works for complex projects with multiple source files for Python and Java, respectively. Lambada furthermore now contains an implementation for medium and partial deep FaaSification, although in the case of subdivided functions, all statements are still executed serially instead of in parallel, suggesting further research to combine the work with automated parallelisation.

In order to become useful for a wider group of users in scientific computing, future research needs to concentrate on deep FaaSification, linking it to compiler

research in which optimisations, code rewrites and parallelisation take place behind the scenes from an engineering point of view, outperforming almost any manual target-specific optimisation.

Table 2 summarises means to overcome limitations in contemporary FaaS environments with the purpose to execute resource-demanding scientific computing jobs, divided into three categories. Our contributions are found within all of the categories.

Table 2. Limitations and solutions

Limitation	Solution	Status
Resources		
CPU	Compute-optimised hardware	Not commercially offered
	Manual parallelisation	*Function futures* (in this paper n)
	Automated parallelisation	Pydron [10]
Memory	Map-reduce	Reference architecture [9]
Network	Local/function-affine services	Not commercially offered
Time and cost		
Runtime	Bypassing temporal limits	*Worm functions* (in this paper n)
	Standard benchmarks	Future work
Cost	Self-hosted runtimes	Available, but lack adaptive function migration
Software		
Environment	Simulated services	Localstack [7]
Development	Function subdivision	*Deep FaaSification* (in this paper n)

5 Summary and Repeatability

The execution of resource-intensive jobs in controlled environments with requirements on repeatability is an atypical use case for serverless computing whose predominant value proposition is exactly hiding any infrastructural configuration. On the other hand, the true on-demand provisioning and billing of hosted functions makes them attractive for research tasks. Our analysis has shown that in many domains of scientific and high-performance computing, solutions can be engineered based on simple functions which are executed on commercially offered or self-hosted FaaS platforms. In this paper we have contributed novel FaaS-related concepts (worm functions, function futures, deep FaaSification) and tools (function subdivision) to improve this engineering process. Furthermore, we have argued for the usefulness of our previous tools (Snafu, Lambada) for researchers and practitioners in this context.

We still see the need for future work in standard benchmarks, tooling for debugging and autotuning, and improved transformation tools to allow for more conventional software to run natively as functions without the need to use abstraction layers such as containers.

All applications, data and scripts used in our experiments are made available for anybody interested to recompute the results and repeat the analysis through the corresponding Open Science Framework repository located at https://osf.io/8qt3j/.

References

1. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P.: Serverless Computing: Current Trends and Open Problems. arXiv:1706.03178, June 2017
2. Berg, T.L., Berg, A.C., Edwards, J., Forsyth, D.A.: Who's in the picture. In: Neural Information Processing Systems (NIPS), Vancouver, British Columbia, Canada, pp. 137–144, December 2004
3. Calvert, P.: Parallelisation of Java for graphics processors. Ph.D. thesis, Trinity College, May 2010
4. Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N.C., Hu, B.: Everything as a service (XaaS) on the cloud: origins, current and future trends. In: 8th IEEE International Conference on Cloud Computing (CLOUD), New York City, New York, USA, pp. 621–628, June 2015
5. Glikson, A., Nastic, S., Dustdar, S.: Deviceless edge computing: extending serverless computing to the edge of the network. In: 10th ACM International Systems and Storage Conference (SYSTOR), Haifa, Israel, May 2017
6. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with OpenLambda. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Denver, Colorado, USA, June 2016
7. Hummer, W.: A fully functional local AWS cloud stack, July 2017. https://github.com/localstack/localstack
8. Jonas, E., Venkataraman, S., Stoica, I., Recht, B.: Occupy the Cloud: Distributed Computing for the 99%. Preprint at arXiv:1702.04024, February 2017
9. Mallya, S., Li, H.M.: Serverless Reference Architecture: MapReduce, October 2016. https://github.com/awslabs/lambda-refarch-mapreduce
10. Müller, S.C., Alonso, G., Amara, A., Csillaghy, A.: Pydron: semi-automatic parallelization for multi-core and the cloud. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, Colorado, USA, pp. 645–659, October 2014
11. Rao, D., Ng, W.K.: Information pricing: a utility based pricing mechanism. In: 14th IEEE International Conference on Dependable, Autonomic and Secure Computing, Auckland, New Zealand, pp. 754–760, August 2016
12. Ricci, R., Eide, E.: Introducing CloudLab: scientific infrastructure for advancing cloud architectures and applications. ;login: Usenix Mag. **39**(6), 36–38 (2014)
13. Spillner, J.: Transformation of Python Applications into Function-as-a-Service Deployments. arXiv:1705.08169, May 2017
14. Spillner, J.: Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. arXiv:1703.07562, March 2017

AccaSim: An HPC Simulator for Workload Management

Cristian Galleguillos[1,2](✉), Zeynep Kiziltan[1], and Alessio Netti[1]

[1] Department of Computer Science and Engineering,
University of Bologna, Bologna, Italy
zeynep.kiziltan@unibo.it, alessio.netti@studio.unibo.it
[2] Escuela de Ing. Informática, Pontificia Universidad Católica de Valparaíso,
Valparaíso, Chile
cristian.galleguillos.m@mail.pucv.cl

Abstract. We present AccaSim, an HPC simulator for workload management. Thanks to the scalability and high customizability features of AccaSim, users can easily represent various real HPC system resources, develop dispatching methods and carry out large experiments across different workload sources. AccaSim is thus an attractive tool for conducting controlled experiments in HPC dispatching research.

1 Introduction

High Performance Computing (HPC) systems have become fundamental tools to solve complex, compute-intensive, and data-intensive problems in diverse engineering, business and scientific fields, enabling new scientific discoveries, innovation of more reliable and efficient products and services, and new insights in an increasingly data-dependent world. This can be witnessed for instance in the annual reports[1] of PRACE and the recent report[2] by ITIF which accounts for the vital importance of HPC to the global economic competitiveness.

As the demand for HPC technology continues to grow, a typical HPC system receives a large number of variable requests by its end users. This calls for the efficient management of the submitted workload and system resources. This critical task is carried out by the software component *Workload Management System* (WMS). Central to WMS is the *dispatcher* which has the key role of deciding when and on which resources to execute the individual requests by ensuring high system performance and Quality of Service (QoS), such as high utilization of resources and high throughput. An optimal dispatching decision is a hard problem [4], and yet suboptimal decisions could have severe consequences, like wasted resources and/or exceptionally delayed requests. Efficient HPC dispatching is thus an active research area, see for instance [9] for an overview.

One of the challenges of the dispatching research is the amount of experimentation necessary for evaluating and comparing various approaches in a controlled

[1] http://www.prace-ri.eu/praceannualreports/.
[2] http://www2.itif.org/2016-high-performance-computing.pdf.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 169–184, 2018.
https://doi.org/10.1007/978-3-319-73353-1_12

environment. The experiments differ under a range of conditions with respect to workload, the number and the heterogeneity of resources, and dispatching method. Using a real HPC system for experiments is not realistic for the following reasons. First, researchers may not have access to a real system. Second, it is impossible to modify the hardware components of a system, and often unlikely to access its WMS for any type of alterations. And finally, even with a real system permitting modifications in its WMS, it is inconceivable to ensure that distinct approaches process the same workloads, which hinders fair comparison. Therefore, simulating a WMS in a synthetic HPC system is essential for conducting controlled dispatching experiments. Unfortunately, currently available simulators are not flexible enough to render customization in many aspects, limiting the scope of their usage.

The contribution of this paper is the design and implementation of AccaSim, an HPC simulator for workload management. AccaSim is an open source, freely available library for Python, executable in any major operating system. AccaSim is scalable and highly customizable, allowing to carry out large experiments across different workload sources, resource settings, and dispatching methods. Moreover, AccaSim enables users to design novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power consumption and failures of the resources. The researchers can use AccaSim to mimic any real system by setting up the synthetic resources suitably, develop advanced such as power-aware, fault-resilient dispatching methods, and test them over a wide range of workloads by generating them synthetically or using real workload traces from HPC users. As such, AccaSim an attractive tool for developing dispatchers and conducting controlled experiments in HPC dispatching research.

This paper is organized as follows. After introducing the concept of WMS in Sect. 2, we present in Sect. 3 the architecture and main features of AccaSim, and recap its implementation, instantiation, and customization. In Sect. 4, we show a case study to illustrate the use of AccaSim for evaluating dispatching methods and highlight its scalability. We discuss the related work in Sect. 5 and conclude in Sect. 6.

2 Workload Management System in HPC

A WMS is an important software of an HPC system, being the main access for the users to exploit the available resources for computing. A WMS manages user requests and the system resources through critical services. A user request consists of the execution of a computational application over the system resources. Such a request is referred to as job and the set of all jobs are known as workload. The jobs are tracked by the WMS during all their states, i.e. from their submission time, to queuing, running, and completion. Once the jobs are completed, the results are communicated to the respective users. Figure 1 depicts a general scheme of a WMS.

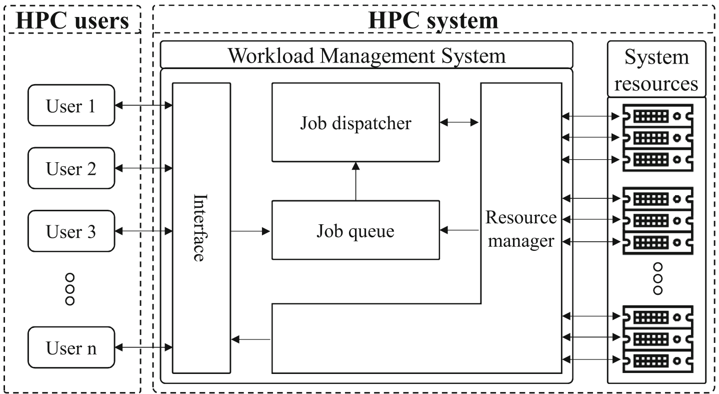


Fig. 1. HPC workload management system.

A WMS offers distinct ways to users for *job submission* such as a GUI and/or a command line interface. A submitted job includes the executable of a computational application, its respective arguments, input files, and the resource requirements. An HPC system periodically receives job submissions. Some jobs may have the same computational application with different arguments and input files, referring to the different running conditions of the application in development, debugging and production environments. When a job is submitted, it is placed in a *queue* together with the other pending jobs (if there are any). The time interval during which a job remains in the queue is known as waiting time. The queued jobs compete with each other to be executed on limited resources.

A *job dispatcher* decides which jobs waiting in the queue to run next (*scheduling*) and on which resources to run them (*allocation*) by ensuring high system performance and QoS, such as high utilization of resources and high throughput. The dispatching decision is generated according a policy using the current system status, such as the queued jobs, the running jobs and the availability of the resources. A suboptimal dispatching decision could cause resource waste and/or exceptional delays in the queue, worsening the system performance and the perception of its users. A (near-)optimal dispatching decision is thus a critical aspect in WMS.

The dispatcher periodically communicates with a *resource manager* of the WMS for obtaining the current system status. The *resource manager* updates the system status through a set of active monitors, one defined on each resource which primarily keeps track of the resource availability. The WMS systematically calls the *dispatcher* for the jobs in the queue. An answer means that a set of jobs are ready for being executed. Then the dispatching decision is processed by the *resource manager* by removing the ready jobs from the queue and sending them to their allocated resources. Once a job starts running, the *resource manager* turns its state from "queued" to "running". The *resource manager* commonly tracks the running jobs for giving to the WMS the ability to communicate their

state to their users through the interface, and in a more advanced setting to (let the users) submit again their jobs in case of resource failures. When a job is completed, the *resource manager* turns its state from "running" to "completed" and communicates its result to the interface to be retrieved by the user.

3 AccaSim

AccaSim enables to simulate the WMS of any real HPC system with minimum effort and facilitates the study of various issues related to dispatching methods, such as feasibility, behavior, and performance, accelerating the dispatching research process.

In the rest of this section, we first present the architecture and the main features of AccaSim, and then recap its implementation, instantiation and customization.

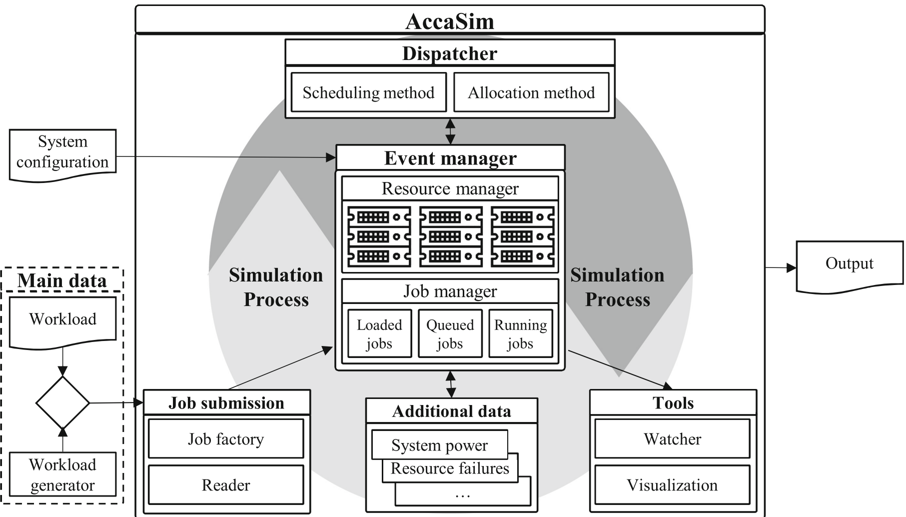


Fig. 2. AccaSim architecture.

3.1 Architecture and Main Features

AccaSim is designed as a discrete event simulator. The simulation is guided by certain events that belong to a real HPC system. These events are mainly collected from workload and correspond to the job submission, starting and completion times, referred to as T_{sb}, T_{st} and T_c, resp. The architecture of AccaSim is depicted in Fig. 2. Since there are no real users for submitting jobs nor real resources for computation during simulation, the first step for starting a simulation is to define the synthetic system with its jobs and resources.

The *job submission* component mimics the job submission of users. The main input data is the workload provided either in the form of a file, corresponding to the descriptions of the existing workloads, or in the form of an external workload generator producing synthetic workload descriptions based on statistical data. The default *reader* subcomponent reads the input from a file in Standard Workload Format (SWF)[11] and passes the parsed data to the *job factory* subcomponent for creating the synthetic jobs for simulation, keeping the information related to their identification, submission time, duration and request of system resources. The created jobs are then mapped to the *event manager* component, simulating the job submission process. The main data input is customizable in the sense that any workload description file and any synthetic workload generator can be used. The *reader* can be adapted easily for this purpose to parse any workload description file format or to read from any workload source.

Event manager is the core component of the simulator, which mimics the behaviour of the synthetic jobs and the presence of the synthetic resources, and manages the coordination between the two. Differently from a real WMS, the *job manager* subcomponent tracks the jobs during their artificial life-cycle by maintaining all their possible states "loaded", "queued", "running" and "completed" via the events handled by the *event manager*. During simulation, at each time point t:

- the *event manager* checks if $t = T_{sb}$ for some jobs. If the submission time of a job is not yet reached, the *job manager* assigns the job the "loaded" state meaning in the real context that the job has not yet been submitted. If instead the submission time of a job is reached, the *job manager* updates its status to "queued";
- the *dispatcher* component gives a dispatching decision on (the subset of) the queued jobs, assigning them an immediate starting time. The *event manager* reveals that $t = T_{st}$ for some waiting jobs and consequently the *job manager* updates their status to "running";
- the *event manager* checks if $t = T_c$ for currently running jobs. Since these jobs were dispatched in a previous time point, their starting and completion times are known (the completion time of a job is the sum of its starting time and duration). If the completion time of a job is reached, the *job manager* updates its status to "completed".

The *resource manager* subcomponent of the *event manager* defines the synthetic resources of the system using a system configuration file in input, and then mimics their allocation and release at the job starting and completion times. Hence, at a time point t, if a job starts, the *resource manager* allocates for the job the resources decided by the *dispatcher*; and if it completes, the *resource manager* releases its resources. The system configuration file can be customized according to the needed types of resources in the simulation.

AccaSim is designed to maintain a low consumption of memory for scalability concerns, therefore job loading is performed in an incremental way, loading only the jobs that are near to be submitted at the corresponding simulation time, as

opposed to loading them once and for all. Moreover, completed jobs are removed from the system so as to release space in the memory.

The *dispatcher* component responsible for generating a dispatching decision interacts with the *event manager* for retrieving the current system status regarding the queued jobs, the running jobs, and the availability of the resources. Note that the *dispatcher* is not aware of the job durations. This information is known only by the *event manager* to stop the jobs at their completion time in a simulated environment. The scheduler and the allocator subcomponents of the *dispatcher* are customizable according to the methods of interest. Currently implemented and available methods for scheduling are: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority [20]; and for allocation: First Fit (FF) which allocates to the first available resource, and Consolidate (C) which sorts the resources by their current load (busy resources are preferred first), thus trying to fit as many jobs as possible on the same resource, to decrease the fragmentation of the system.

It has been shown in the last decade that system performance can be enhanced greatly if the dispatchers are aware of additional information regarding the current system status, such as power consumption of the resources [2,5,6,24], resource failures [7,15], and the heating/cooling conditions [3,23]. The *additional data* component of AccaSim provides an interface to integrate such extra data to the system which can then be utilized to develop and experiment with advanced dispatchers which are for instance power-aware, fault-resilient and thermal-aware. The interface lets receive the necessary data externally from the user, make the necessary calculations together with some input from the *event manager*, all customizable according to the need, and pass back the result to the *event manager* so as to transfer it to the *dispatcher*.

We conclude the overview of the architecture with the *tools* component and the output data which help the users to follow the simulation process and analyze the results. The *watcher* allows tracking the current system status, such as the number of queued jobs, the running jobs, the completed jobs, the availability of the resources, etc. The *visualization* instead shows in a GUI a representation of the allocation of resources by the running jobs during the simulation. The output data are of two types: (i) the data regarding the execution of the dispatching decision for each job, and (ii) the data related to the simulation process, specifically the CPU time required by the simulation tasks like the generation of the dispatching decision, job loading etc. Such data is useful for analyzing the dispatching results and the performance of the simulation process.

To sum up the main features, AccaSim is customizable in its workload source, resource types, and dispatching methods; AccaSim enables users to design novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power consumption and failures of the resources; and Accasim provides tools and output data to follow the simulation process and analyze the results.

3.2 Implementation, Instantiation and Customization

AccaSim is implemented in Python (compatible with version 3.4 or above) which is an interpreted, object-oriented, high-level programming language, freely available for any major operating system, and is well established with a large community in academia and industry[3]. All the dependencies used by AccaSim are part of any Python distribution, except the matplotlib and psutil packages which can be easily installed using the pip management tool. The source code is available under MIT License, together with a documentation at http://accasim.readthedocs.io/en/latest/. A release version is available as a package in the PyPi repository[4].

The highly customizable characteristic of AccaSim is driven by its abstract classes and the inheritance capabilities of Python. The UML diagram of the main classes is shown in Fig. 3 where the abstract classes associated to the customizable components are highlighted in bold.

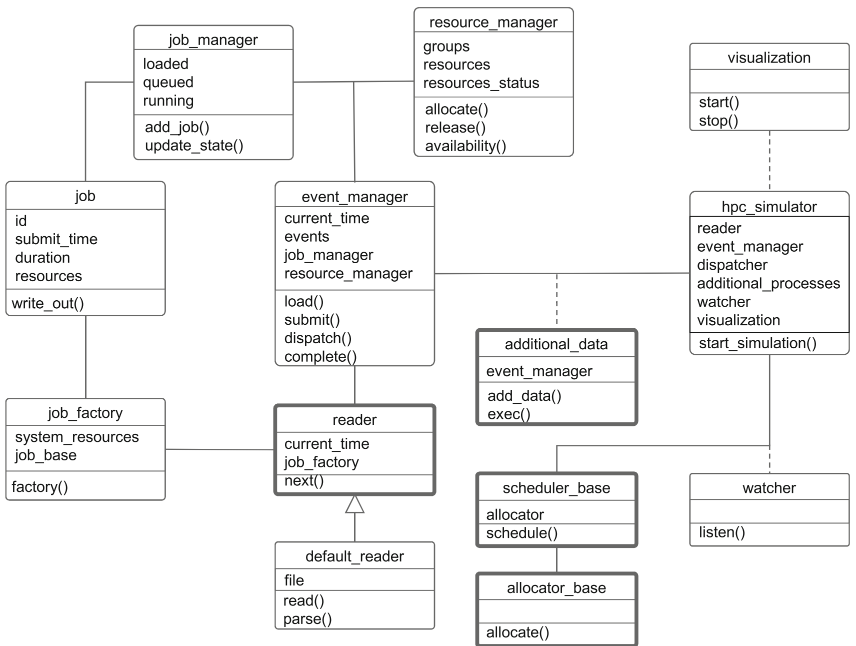


Fig. 3. AccaSim class diagram.

The starting point for launching a simulation is to instantiate the *hpc_simulator* class. It must receive as arguments at least a workload description – such as a file path in SWF format, a system configuration file path in JSON

[3] https://www.python.org/events/python-events/.
[4] https://pypi.org.

format, and a dispatcher instance, with which the synthetic system is generated and loaded with all the default features. An example instantiation is detailed in Fig. 4.

```
1  from accasim.base.simulator_class import hpc_simulator
   from accasim.base.scheduler_class import fifo_sched
3  from accasim.base.allocator_class import ff_alloc

5  workload = '/tmp/workload.swf'
   sys_cfg = '/tmp/sys_config.json'
7
   allocator = ff_alloc()
9  dispatcher = fifo_sched(allocator)
   simulator = hpc_simulator(workload, sys_cfg, dispatcher)
11 simulator.start_simulation()
```

Fig. 4. A sample AccaSim instantiation.

The workload description file is handled by an implementation of the abstract *reader* class, which is *default_reader* by default. The file is read and parsed by the *read()* and *parse()* methods. AccaSim can be customized in its workload description file format by modifying these methods suitably. AccaSim can as well be customized so as to read workloads from any source, not necessarily from a file, by implementing the abstract *reader* class appropriately.

The system configuration file, which is processed by the *resource_manager* class, defines the synthetic resources. The file has two main contents. The first specifies the resource types and their quantity in each group of nodes, which is useful for modeling heterogeneous HPC systems. The second instead determines the number of nodes of each group. See Fig. 5 for an example. The user is free to mimic any real system by customizing this configuration file suitably.

The dispatcher instance is composed by implementations of the abstract *scheduler_base* and *allocator_base* classes. Both classes must implement their main methods, *schedule()* and *allocate()* respectively, to deal with the scheduling and the allocation decisions of the dispatching. In this illustrative instantiation of the *hpc_simulator* class, *fifo_sched* implements *scheduler_base* using FIFO, whereas *ff_alloc* implements *allocator_base* using FF, and both *fifo_sched* and *ff_alloc* classes are available in the library for importing, as done in lines 2–3 of Fig. 4. AccaSim can be customized in its dispatching method by implementing the abstract *scheduler_base* and *allocator_base* classes as desired.

After instantiating the *hpc_simulator* class in line 10 of Fig. 4, the simulation process starts in line 11 with the *start_simulation()* method which has the following optional arguments:

```
simulator.start_simulation(debug=True, watcher=True,visualization=True,
    ↪ additional_data=None)
```

which serve to require the use of a debugger, as well as the *watcher*, the *visualization*, and the *additional data* components of the simulator. The additional_data

argument is an array of objects where each object is an implementation of the abstract *additional_data* class, giving the possibility to customize AccaSim in terms of the extra data that the user may want to provide to the system for dispatching purposes.

4 Case Study

In this section, we show a case study to illustrate the use of AccaSim for evaluating dispatching methods in a simulated environment and highlight AccaSim's scalability. The experiments are done on a Macbook Pro machine with Intel Dual Core i5@2.2 Ghz CPU, 8 GB of RAM, and Python 3.6.

Workload source and synthetic system configuration. We rely on a public workload trace collected from the Seth cluster[5] belonging the High Performance Computing Center North (HPC2N) of the Swedish National Infrastructure for Computing. The workload trace file includes 200,735 jobs spanning through 4 years, from July 2002 to January 2006, and is available on-line[6] in the SWF format. Seth was built in 2001 and is already retired by now. It ranked 59th in Top500 list[7], the world's 500 fastest computers. It was composed of 120 nodes, each node with two AMD Athlon MP2000 + dual core processors with 1.667 GHz and 1 GB of RAM. For high parallel performance, the system was equipped with a low latency network. Because multiple jobs can co-exist on the same node, we consider a better representation of the system, made of cores instead of processors. Therefore, we define the synthetic system in the configuration file with 120 nodes each with 4 cores and 1 GB of RAM, as depicted in Fig. 5.

```
{
    "system_name": "Seth - HPC2N",
    "groups": {
        "g0": {
            "core": 4,
            "mem": 1048576
        }
    },
    "resources": {
        "g0": 120
    }
}
```

Fig. 5. System configuration of Seth.

Dispatching Methods. As we previously mentioned in Sect. 3.1, currently implemented and available methods for scheduling are: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority (EBF); and for allocation are: First Fit (FF) and Consolidate (C). In the experiments, we consider every combination of the available

[5] https://www.hpc2n.umu.se/resources/hardware/seth.
[6] http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html.
[7] http://www.top500.org/.

scheduling and allocation methods, which gives rise to the 8 dispatching methods: FIFO-FF, FIFO-C, SJF-FF, SJF-C, LJF-FF, LJF-C, EBF-FF, and EBF-C. To employ the various dispatching methods, we modify lines 2–3 and 8–9 of Fig. 4 so as to import from the library and use the corresponding implementations of the abstract *scheduler_base* and *allocator_base* classes. Then we run AccaSim 8 times on the entire workload each time with a different dispatching method.

Scalability of AccaSim. We parallelize the experiments in two different threads. The usage of CPU time and memory of each experiment are reported in Table 1, where the time columns correspond to the total CPU time spent by the simulator and the time spent in generating the dispatching decision; whereas the memory columns give the average and the maximum amount of memory utilized over the total simulation time points. In the first thread, FIFO and LJF based experiments are completed in 110 min, while in the second, the SJF and EBF based experiments are completed in 193 min. Each of the experiments took around 30 min. The exceptions are the EBF-based experiments which require around an hour because the underlying dispatching methods are computationally more intensive. As can be observed in the table, the time spent by the simulator, other than generating the dispatching decision, is constant (around 22 min) across all the experiments. The total CPU usage is thus highly dependent on the complexity of the dispatcher.

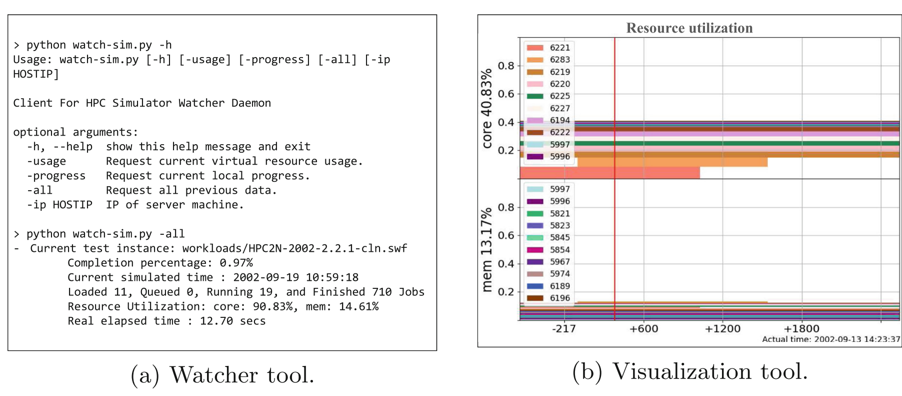
Table 1. CPU time and memory usage of the simulator.

Thread 1					Thread 2				
Simulator	Time (MM:SS)		Memory (MB)		Simulator	Time (MM:SS)		Memory (MB)	
	Total	Disp.	Avg.	Max.		Total	Disp.	Avg.	Max.
FIFO-FF	24:44	03:23	27.8	41.4	SJF-FF	25:48	05:19	32.5	54.7
FIFO-C	26:31	04:43	27.5	40.9	SJF-C	27:19	06:14	32.5	55.4
LJF-FF	29:45	07:26	32.7	54.7	EBF-FF	68:27	46:15	26.8	40.2
LJF-C	30:26	08:12	32.9	54.7	EBF-C	71:43	48:31	26.89	40.2

As for the memory usage, thanks to the incremental job loading and job removal capabilities, AccaSim consumes low memory. The average memory usage is around 30 MB with a peak at 55 MB across all the experiments. Such low memory usage makes it possible to execute experiments in parallel. Considering the large size of the workload, these numbers are very reasonable, supporting the claim that AccaSim is scalable.

Evaluation of the dispatching methods. The dispatching methods can be evaluated and compared from different perspectives thanks to AccaSim's tools and output data. In Fig. 6, sample snapshots of the *watcher* and the *visualization*

tools taken at certain time points during the FIFO-FF experiment are shown. The *watcher* receives command line queries to show a variety of information regarding the current synthetic system status, such as the queued jobs, the running jobs, the completed jobs, resource utilization, the current simulation time point, as well as the total CPU time elapsed by the simulator. The *visualization* tool summarizes the allocation of resources by the running jobs each indicated with a different color, using an estimation (such as wall-time) for job duration. The display is divided by the types of resources. In our case study, the core and memory usage are shown separately.



(a) Watcher tool.

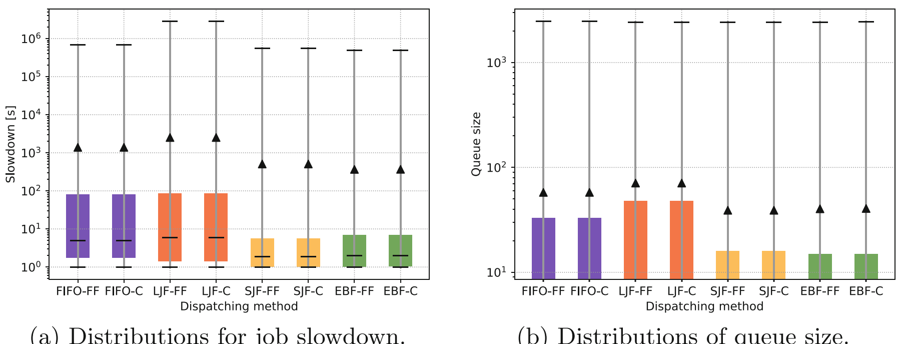
(b) Visualization tool.

Fig. 6. Following a simulation process.

The output file contains two types of data. The first regards the execution of the dispatching decision for each job, such as the starting time, the completion time and its resource allocation, which gets updated each time a job completes its execution. This type of data can be utilized to contrast the dispatcher methods in terms of their effect on system resource utilization: how many resources are used and to what extend they are consumed. Alternatively, the data can be utilized to compare them in terms of their impact on system throughput, using some metrics like the well-know job slowdown [10]. Slowdown of a job j is a normalized waiting time and is defined as $slowdown_j = (T_{w,j} + T_{r,j})/T_{r,j}$ where $T_{w,j}$ is the waiting time and $T_{r,j}$ is the duration of job j. A job waiting more than its duration has a higher slowdown than a job waiting less than its duration. Another useful metric could be the queue size, which is the number of jobs waiting in the queue at a given time point. The lower the slowdown and the queue size are, the higher the throughput is.

In Fig. 7, we show the distributions of the slowdown and the queue size for each of the 8 experiments in box-and-whisker plots. We can see that SJF and EBF based dispatching methods achieve the best results, independently of their allocation methods probably due to the homogeneous nature of the synthetic system. Their slowdown values are mainly lower than the median of the FIFO

and LJF based methods. SJF maintains overall lower slowdown values than the other methods, but a higher mean than the EBF. SJF maintains also slightly higher mean in the queue size than the EBF. The scheduling policy of EBF does not sort the jobs, like SJF, instead it tries to fit as many jobs as possible into the system, which can explain the best average results achieved in terms of slowdown and queue size.

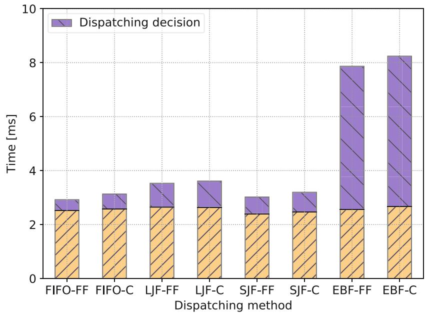


(a) Distributions for job slowdown. (b) Distributions of queue size.

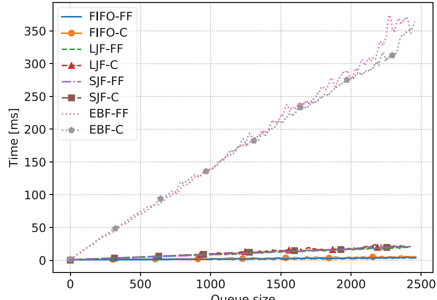
Fig. 7. QoS of the dispatchers.

The second type of output data regards the simulation process, specifically the CPU time required by the simulation tasks like job loading and generation of the dispatching decision, which gets updated at each simulation time point. This type of data can be used to evaluate the performance of the dispatchers in terms of the time they incur for generating a decision. In Fig. 8a, we report the average CPU time required at a simulation time point for each of the 8 experiments. In accordance with Table 1, the time spent in simulation, other than generating the dispatching decision, is constant (around 2 ms in this case) across all the experiments and the EBF based methods spent much more time in generating a decision than the others. In Fig. 8b, we instead analyze the scalability. Specifically, we report for each queue size the average CPU time spent at a simulation time point in generating a dispatching decision for each of the 8 experiments. While all the dispatchers scale well, the EBF based methods require more CPU time for processing bigger queue sizes, due to their scheduling policy which tries to fit as many jobs as possible into the system.

Our analysis restricted to the considered workload and resource settings reveals that, while the EBF based dispatchers give the best throughput, they are much more costly in generating a dispatching decision. Simple dispatchers based on SJF are valid alternatives with their excellent scalability and high throughput comparable to the EBF based methods. The job durations in the current workload are distributed as 56,63% short (duration under 1 h), 34,66% medium (duration between 1 and 12 h), and 8,71% long (duration over 12 h). The synthetic system has a very homogeneous structure with all the nodes having the



(a) Average CPU time at a simulation time point.



(b) Average CPU time at a simulation time point to generate a dispatching decision w.r.t. queue size.

Fig. 8. Performance of the dispatchers.

same number of the same type of resources. It would be interesting to experiment further with the dispatching methods using workloads with different job duration distributions and more heterogeneous resource structures. Thanks to AccaSim, such experiments can be conducted smoothly.

5 Related Work

HPC systems have been simulated from distinct perspectives, for instance to model their network topologies [1,14,17] or storage systems [18,21]. There also exist HPC simulators dealing with the duties of a WMS, as in our work, which are mainly focused on job submission, resource management and job dispatching.

To the best of our knowledge, the most recent WMS simulator is presented in [19]. The Scheduling Simulation Framework (ScSF) emulates a real WMS, Slurm Workload Manager[8] which is popular in many HPC systems. In [16,22] Slurm is modified to provide job submission, resource management and job dispatching through distinct daemons which run in diverse virtual machines and which communicate over RPC calls, and dedicated simulators are implemented. ScSF extends these simulators with automatic generation of synthetic workload descriptions based on statistical data, but does not give the possibility to read real workload descriptions, for instance from a file, for job submission. The dependency on a specific WMS does not render the customization of the WMS, and together with the additional dependency on virtual Machines and MySQL, the set up of ScSF is rather complex. Moreover, ScSF requires a significant amount of resources in the machines where the simulation will be executed.

In [12], an energy aware WMS simulator, called Performance and Energy Aware Scheduling (PEAS) simulator is described. With the main aim being to minimize the power consumption and to increase the throughput of the system,

[8] Slurm Workload Manager: https://slurm.schedmd.com/.

PEAS uses predefined dispatching methods and workload description file format, and the system power calculations are based on a fixed data from SPEC benchmark[9] considering the entire processor at its max load. PEAS is available only as GNU/Linux binary, therefore it is not customizable in any of these aspects.

Brennan et al. [8] define a framework for WMS simulation, called Cluster Discrete Event Simulator (CDES), which uses predefined scheduling algorithms and relies on specific resource types. Although CDES allows reading real workload descriptions for job submission, as apposed to having to generate them automatically as in ScSF, all jobs are loaded at the beginning which can hinder the performance when experimenting with a large number of jobs. Moreover, the implementation is not available which prevents any form of customization.

In [13], a WMS simulator based on a discrete event library called Omnet++[10] is introduced. Similar to ScSF, only automatically generated synthetic workload descriptions are accepted for job submission. Since Omnet++ is primarily used for building network simulators and is not devoted to workload management, there exist issues such as the inability to consider different types of resources as in CDES. Moreover, due to lack of documentation, it is hard to understand to what extend the simulator is customizable.

The main issues presented in the existing WMS simulators w.r.t. to AccaSim can be summarized as complex set up and need of many virtual machines and resources, inflexibility in the workload source, performance degrade with large workloads, no customization of the WMS, and unavailable or undocumented implementation.

6 Conclusions

In this paper, we presented AccaSim, a library for simulating WMS in HPC systems, which offers to the researchers an accessible tool to aid them in their HPC dispatching research. The library is open source, implemented in Python, which is freely available for any major operating system, and works with dependencies reachable in any distribution, making it easy to use. AccaSim is scalable and is highly customizable, allowing to carry out large experiments across different workload sources, resource settings, and dispatching methods. Moreover, AccaSim enables users to design novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power consumption and failures of the resources.

In future work, we plan to do experimental comparison to other simulators and assess further the scalability of AccaSim in terms of distinct system configurations. Besides, in order to aid the users further in evaluating dispatchers, we are currently working on showing more information in the tools regarding system utilization, such as the amount of allocation of each resource individually, and on automatically generating the performance and QoS plots used in this paper.

[9] https://www.spec.org/power_ssj2008/.
[10] http://www.omnetpp.org/.

In addition, we plan to externalize the visualization tool by executing it in an independent application to reduce the simulation resource usage.

Acknowledgments. C. Galleguillos is supported by Postgraduate Grant PUCV 2017. We thank Alina Sîrbu for fruitful discussions on the work presented here.

References

1. Acun, B., Jain, N., Bhatele, A., Mubarak, M., Carothers, C.D., Kalé, L.V.: Preliminary evaluation of a parallel trace replay tool for HPC network simulations. In: Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Weidendorfer, J., Alexander, M. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 417–429. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_34
2. Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., Wilde, T.: A case study of energy aware scheduling on SuperMUC. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 394–409. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_25
3. Banerjee, A., Mukherjee, T., Varsamopoulos, G., Gupta, S.K.: Integrating cooling awareness with thermal aware workload placement for HPC data centers. Sustain. Comput. Inf. Syst. **1**(2), 134–150 (2011)
4. Blazewicz, J., Lenstra, J.K., Kan, A.H.G.R.: Scheduling subject to resource constraints: classification and complexity. Discrete Appl. Math. **5**(1), 11–24 (1983)
5. Bodas, D., Song, J., Rajappa, M., Hoffman, A.: Simple power-aware scheduler to limit power consumption by HPC system within a budget. In: Proceedings of E2SC@SC, pp. 21–30. IEEE (2014)
6. Borghesi, A., Collina, F., Lombardi, M., Milano, M., Benini, L.: Power capping in high performance computing systems. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 524–540. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_37
7. Brandt, J.M., Debusschere, B.J., Gentile, A.C., Mayo, J., Pébay, P.P., Thompson, D.C., Wong, M.: Using probabilistic characterization to reduce runtime faults in HPC systems. In: Proceedings of CCGRID, pp. 759–764. IEEE CS (2008)
8. Brennan, J., Kureshi, I., Holmes, V.: CDES: an approach to HPC workload modelling. In: Proceedings of DS-RT, pp. 47–54. IEEE CS (2014)
9. Bridi, T., Bartolini, A., Lombardi, M., Milano, M., Benini, L.: A constraint programming scheduler for heterogeneous high-performance computing machines. IEEE Trans. Parallel Distrib. Syst. **27**(10), 2781–2794 (2016)
10. Feitelson, D.G.: Metrics for parallel job scheduling and their convergence. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 188–205. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45540-X_11
11. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. J. Parallel Distrib. Comput. **74**(10), 2967–2982 (2014)
12. Gómez-Martín, C., Vega-Rodríguez, M.A., Sánchez, J.L.G.: Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods. Concurrency Comput. Pract. Exp. **27**(17), 5436–5459 (2015)
13. Hurst, W.B., Ramaswamy, S., Lenin, R.B., Hoffman, D.: Modeling and simulation of HPC systems through job scheduling analysis. In: Conference on Applied Research in Information Technology. Acxiom Laboratory of Applied Research (2010)

14. Jain, N., Bhatele, A., White, S., Gamblin, T., Kalé, L.V.: Evaluating HPC networks via simulation of parallel workloads. In: Proceedings of SC, pp. 154–165. IEEE CS (2016)
15. Li, Y., Gujrati, P., Lan, Z., Sun, X.: Fault-driven re-scheduling for improving system-level fault resilience. In: Proceedings of ICPP, pp. 39. IEEE CS (2007)
16. Lucero, A.: Simulation of batch scheduling using real production-ready software tools. In: Proceedings of IBERGRID, pp. 345–356, Netbiblo (2011)
17. Mubarak, M., Carothers, C.D., Ross, R.B., Carns, P.H.: Enabling parallel simulation of large-scale HPC network systems. IEEE Trans. Parallel Distrib. Syst. **28**(1), 87–100 (2017)
18. Nuñez, A., Fernández, J., García, J.D., García, F., Carretero, J.: New techniques for simulating high performance MPI applications on large storage networks. J. Supercomput. **51**(1), 40–57 (2010)
19. Rodrigo, G.P., Elmroth, E., Östberg, P.-O., Lavanya, R.: ScSF: a scheduling simulation framework. To appear in the Proceedings of JSSPP. Springer (2017)
20. Skovira, J., Chan, W., Zhou, H., Lifka, D.: The EASY — LoadLeveler API project. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1996. LNCS, vol. 1162, pp. 41–47. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0022286
21. Snyder, S., Carns, P.H., Latham, R., Mubarak, M., Ross, R.B., Carothers, C.D., Behzad, B., Luu, H.V.T., Byna, S., Prabhat, S.: Techniques for modeling large-scale HPC I/O workloads. In: Proceedings of PMBS@SC, pp. 5:1–5:11. ACM (2015)
22. Stephen, T., Benini, M.: Using and modifying the BSC slurm workload simulator, Technical report, Slurm User Group Meeting (2015)
23. Tang, Q., Gupta, S.K.S., Varsamopoulos, G.: Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: a cyber-physical approach. IEEE Trans. Parallel Distrib. Syst. **19**(11), 1458–1472 (2008)
24. Zhou, Z., Lan, Z., Tang, W., Desai, N.: Reducing energy costs for IBM blue gene/P via power-aware job scheduling. In: Desai, N., Cirne, W. (eds.) JSSPP 2013. LNCS, vol. 8429, pp. 96–115. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43779-7_6

SherlockFog: Finding Opportunities for MPI Applications in Fog and Edge Computing

Maximiliano Geier[1] and Esteban Mocskos[1,2](✉)

[1] Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, C1428EGA Buenos Aires, Argentina
{mgeier,emocskos}@dc.uba.ar

[2] Centro de Simulación Computacional p/Aplic. Tecnológicas, CSC-CONICET, Godoy Cruz 2390, C1425FQD Buenos Aires, Argentina

Abstract. The Fog and Edge Computing paradigms have emerged as a solution to limitations of the Cloud Computing model to serve a huge amount of connected devices efficiently. These devices have unused computing power that could be exploited to execute parallel applications.

In this work we present SherlockFog, a tool to experiment with parallel applications in Fog and Edge network setups, specially focused on the MPI based applications. We propose a methodology to study feasibility of running parallel applications in Fog or Edge environments. We validate this tool contrasting experimental results with theoretical predictions reaching remarkable agreement between both.

We analyze the effect of worsening network conditions for several benchmarks of the MPI version of NAS Parallel Benchmarks on foglike network topologies. Our results show that this impact is sublinear in some cases, opening up opportunities to use distributed, increasingly ubiquitous computational resources.

Keywords: Distributed systems · Fog and Edge Computing Parallel applications · Benchmarks

1 Introduction

In the last years, the Cloud Computing model has emerged as an alternative to owning and managing computing infrastructure, thus reducing operation costs. These operations provide an efficient solution to serve several types of applications, such as web servers, databases, storage and batch processing. However, it is not without limitations. As the Cloud is usually farther away from the clients, latency-sensitive applications could suffer from performance degradation in this setup. The Fog Computing model is defined by Bonomi *et al.* [6] as a highly virtualized platform that provides compute, storage and networking services between end devices and the Cloud. This model allows certain services to be offloaded to end nodes, thus enabling services with lower latency requirements. Going further, the proliferation of IoT devices with increasing computing

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 185–199, 2018.
https://doi.org/10.1007/978-3-319-73353-1_13

power has given an identity to nodes at the edge of the network, defining the Edge Computing model [17]. In this model, nodes at the edge of the network cooperate among themselves to provide solutions at lower response time or bandwidth requirements, while improving data safety and privacy with respect to handling processing at the Cloud. As the number of nodes at the edge increase, it is also becoming increasingly difficult to serve all clients at the cloud with this infrastructure in terms of server load and required bandwidth. End nodes are usually slower and have worse connectivity than the main infrastructure, but both aspects have been steadily improving, enabling execution of more and more applications.

Scientific computing has gone down another path. The standard infrastructure to support this type of applications is cluster computing. It is commonplace that a cluster is homogeneous in computing power and topology (i.e. every node is connected to each other directly and runs at the same speed). This homogeneity allows developers not to worry about process placement. A variation that has appeared during the last decade is multi-core clusters. In this case, processes running on the same computer can communicate with each other faster than with processes on different nodes. Multi-cores are Symmetric Multiprocessing (SMP) machines: each node acts as an homogeneous cluster of CPU cores, which is connected to the outer world homogeneously as well. This paradigm is ubiquitous in scientific computing, driving research and improvements in simulation models and techniques.

The standard API used in scientific applications is MPI [11,12]. It has been designed to provide an abstraction to handle data exchange and synchronization among processes in the same node or in different ones. Developers of MPI implementations target their code to be used on clusters, often requiring full node connectivity and implementing optimizations for collective operations that work more efficiently in homogeneous environments. MPI is also very sensitive to network changes, bringing the computation to a halt if just one node disconnects or changes its network location.

This begets the question of whether it is possible to do scientific computing in the Edge Computing paradigm, taking advantage of unused computing power of nodes at the edge of network. The Edge/Fog paradigm poses new challenges in this regard, such as the variability on the processing power of nodes and the volatility of nodes due to the dynamic nature of the network in terms of churn, latency and bandwidth. Even though Edge Computing could provide better latency than having to offload data to do computing on the Cloud, specialized clusters are connected using a faster dedicated network. This difference could impair application performance greatly, making it unfeasible to use such an infrastructure. We approach this question by introducing a methodology to analyze distributed scientific applications in an heterogeneous environment, using different emulated network settings. It is important to note that assessing performance degradation is not easy from a theoretical standpoint. Factors such as link bandwidth and latency, communication patterns, message sizes, traffic congestion issues and implementation details of the underlying communication framework make it complex to model accurately.

Most work on leveraging MPI in heterogeneous environments has been focused on multi-core cluster architectures. This has been a natural step from traditional homogeneous clusters since the introduction of this type of processors into the market.

In this work, we present SherlockFog, a tool that aims to bridge the gap between existing scientific computing applications and Edge/Fog Computing. Our approach explores feasibility of running MPI applications in Edge Computing environments. This kind of architecture is inherently dynamic and has different topologies, performance characteristics and exploitable computing power per node than typical multi-core clusters. MPI on Edge Computing is still mostly unexplored, since computing power and communication capabilities of fog and edge nodes have only been made possible to compute High Performance Computing (HPC) applications very recently. Since implementations of existing MPI libraries have not been designed for this type of environments, providing a framework to test the library itself is also a key feature we have considered in our approach.

This work is focused on MPI as it is widely used by the scientific applications community, aiming to reuse existing code. However, our proposal provides a framework that allows the full software stack to be used unmodifed and it is therefore not limited to MPI applications.

2 Related Work

We discuss some representative examples of tools and metholodogies that can support the use of MPI applications in heterogeneous environments.

Brandfass *et al.* [7] propose a rank-reordering scheme to increase performance of unstructured Computer Fluid Dynamics (CFD) code on parallel multi-cores. This optimization produces a mapping of MPI processes to CPU cores such that main communication happens within compute nodes, exploiting the fact that intra-node communication is much faster than inter-node in this kind of architectures, using characteristics of the target application. Since load per process is not uniform in unstructured code, it makes sense to reorder processes to reduce frontier communication. Dichev *et al.* [10] show two novel algorithms for the scatter and gather MPI primitives that improve performance in multi-core heterogeneous platforms. This work focuses on optimizing broadcast trees used by most MPI implementations using a weight function that depends on topology information. However, the user can not experiment using virtual topologies, thus difficulting the study of MPI applications in edge-like environments. Mercier and Clet-Ortega [15] study a more sophisticated process placement policy that takes into account the architecture's memory structure to improve multi-core performance. This proposal is also not suitable for our purposes since the target platform is potentially dynamic and virtual topologies cannot be analyzed. Navaridas *et al.* [16] study process placement in torus and fat-tree topologies through scheduling simulation using synthetic workloads. This work relies on an execution model which would have to be adapted to study our target platform.

Simulation tools are also widely used to analyze distributed systems. This approach allows the user to explore environments that are difficult to set up in real life. In this case, the application is executed completely in a simulated environment, be it online or offline. This approach usually requires the user to modify application code in order to use the simulator. In the offline case, execution traces must be generated for a defined set of input parameters that are then fed to the simulator. A few representative examples thereof follow:

NS-3 [3] is a widely-used full-stack detailed discrete event simulator designed for network applications. However, since the simulator is not a parallel application itself, it is not possible to scale simulation of MPI applications beyond tens of nodes. It does not provide mechanisms to transform MPI applications directly into NS-3 simulations. An extension called DCE (Direct Code Execution) [2] wraps C library calls to be simulated by NS-3. However, this virtual C library is limited and does not implement all system calls required to run MPI applications.

SimGrid [8] is another widely used discrete event simulator. It is aimed at simulating large-scale distributed systems efficiently. Moreover, it provides an online mechanism to execute MPI applications by wrapping MPI calls to the simulator engine, called SMPI [9]. This API allows MPI applications to be ported to SimGrid simulations easily if the source code is available. In order to scale simulations on a single node, allowing up to thousands of simulated nodes, SimGrid implements a simplified communication model. While this simulator allows to experiment using heterogeneous network topologies, its communication models have only been validated for performance prediction accuracy on fine-tuned homogeneous environments. Moreover, this approach doesn't allow the user to experiment with different MPI libraries. The behavior of the HPC application itself, at the MPI primitives level, relies on implementation details of the simulation engine.

Dimemas [1] is a performance analysis tool for MPI applications, which is able to simulate execution offline on a given target architecture, ranging from single- or multi-core networked clusters to heterogeneous systems. Performance is calculated by means of replaying the execution trace of an application on a built-in simulator. Similarly to SimGrid, this approach relies on a particular implementation of MPI primitives and the communication model itself. We aim towards building a complete framework for analyzing distributed applications on Edge Computing, whereas simulation forces a particular model which depends deeply on the tool.

In all aforementioned simulation tools, traffic is simulated using a model that depends on the particular simulator implementation and on user-provided input parameters. Our work focuses not just on the applications themselves, but also on building a framework for analyzing and developing distributed applications and support libraries on the edge of the network.

Another approach is network emulation, which consists of building the execution environment using real nodes on a virtual network topology. Emulation allows using the same software environment as it would be used in a real system,

while providing different network conditions. This solves modelling the application and the communication framework. Several tools make use of traffic shaping facilities in modern operating systems to emulate the network and run distributed applications on it, but none of them focus on MPI applications in heterogeneous architectures. We cite a few examples:

Lantz *et al.* [14] present Mininet, a tool to emulate Software-Defined Networks (SDNs) in a single host. It leverages network namespaces and traffic shaping technologies of the Linux operating system to instantiate arbitrary network topologies. This tool requires virtual nodes to execute on a single host, impairing scalability. Moreover, as isolation occurs only at the network level, intelligent MPI implementations can determine that virtual hosts reside on the same real host and thus communicate more efficiently than expected.

Wette *et al.* [18] extend Mininet in order to span an emulated network over several real hosts. However, it does not address design limitations in the original tool that prevent it from running MPI code, such as hostname isolation on namespaces that share a filesystem.

White *et al.* [19] propose Emulab, a shared testbed that allows running MPI code on it, but it has dedicated hardware requirements that make it expensive to self-deploy. While it is possible for researchers to use the shared testbed at University of Utah, there is very little control of job allocation and bandwidth usage, and is therefore not suitable for performance analysis of CPU and network-intensive applications.

3 Methodology

We propose a novel methodology to support the analysis and porting of distributed computing applications to be executed following the paradigm of Edge Computing. Our proposal focus on the impact of different traffic patterns in applications. We have focused our work on MPI applications, as it is the most widely used API for message-passing distributed computing, but our approach is valid for other distributed programming models.

Figure 1 shows the process schematically.

1. The user selects an application and a topology and creates an experiment script for SherlockFog to deploy it on a set of physical nodes.
2. SherlockFog connects to every node and initializes network namespaces for each virtual node.
3. Virtual links are generated to match input topology.
4. Static routing is used to allow applications on each namespace to communicate to each other.

The tool allows the user to change network parameters during the run. This process can be repeated, comparing different topologies or input parameters. Application output is then analyzed, comparing behavior on different scenarios.

In the next paragraphs, we present the tool further and show the topologies we have used in our experiments. Moreover, we describe the experimental methology in detail.

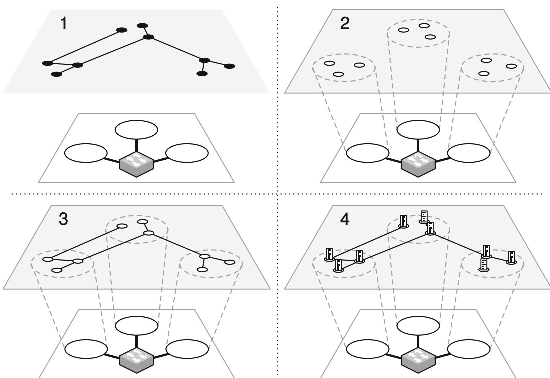


Fig. 1. SherlockFog allows the user to analyze application behavior while varying network topologies and their properties in a reproducible procedure.

3.1 SherlockFog: A Distributed Experimental Framework to Enable Fog and Edge Computing

SherlockFog is a tool that takes care of automating the deployment of a given topology and running the experiments. It makes extensive use of the `ip` tool–found on most GNU/Linux installations–to set up virtual Ethernet interfaces on Linux Network Namespaces. These interfaces are cloned using the `macvlan` feature[1]. A pair is connected by assigning IP addresses in the same point-to-point subnet[2] to both endpoints. Traffic flows through the carrier of the host network interface. This new interface has different MAC address and configuration (eg. name resolution dictionary, firewall, ARP and routing tables). Routing is also configured statically on every namespace in order to match the input topology. All virtual nodes act as routers, forwarding packets to its neighbors. Moreover, ARP is disabled on all virtual interfaces to prevent virtual nodes which are not neighbors in the input topology to find each other by sending ARP requests, thus bypassing our configuration. An SSH server is brought up on every container automatically to be able to run MPI applications. It runs on a different UTS namespace[3], whose hostname matches that of the virtual host. This feature is required for MPI applications to work on the virtual infrastructure, as some implementations check the hostname to define whether shared memory or a network transport should be used for communication. To the best of our knowledge, no other network experimentation tool takes this into account. It also allows MPI hostfiles or rankfiles to be set up more easily using consistent names, the choice of real nodes notwithstanding. Name resolution is handled by generating appropriate `/etc/hosts` files for each namespace automatically.

---

[1] https://hicu.be/bridge-vs-macvlan.
[2] A/30 network prefix.
[3] http://windsock.io/uts-namespace/.

These files are bound by the `ip netns exec` command. Finally, using the NetEm traffic control extension [13] via the `tc` tool, link parameters can be modified on a given virtual network interface's outbound port.

3.2 Features of SherlockFog

SherlockFog runs on commodity hardware, such as interconnected desktop computers in an university campus. No special interconnection technology or programmable switch is required, lowering the cost of ownership significantly with respect to similar solutions.

The scripting language allows to set up experiment parameters and runs, enabling for reproducible experimentation.

The tool can connect namespaces in the same physical computer or in different hosts, provided that the traffic that every host generates is reachable from the rest. This allows to grow experiment scale by using hosts on different interconnected switches.

Application code can be executed unmodified. The user can execute open- or closed-source programs in the same software environment as they would in a real environment.

It is also possible to experiment with changes to the MPI library, such as broadcast implementations for edge environments or features that make it more resilient to churn or changes in latency or bandwidth. As we are exploring MPI on non-standard network settings, our tool could be used as a testing framework for these use cases.

Modeling mobility is also an important aspect in Fog or Edge Computing environments. Our tool provides a mechanism to do so by changing bandwidth and packet loss for a link.

We show a sample experiment script in Fig. 2. In this example, 4 nodes named *n0* to *n3* (line 2) are initialized and connected sequentially, generating a

```
1  ### node def
2  for n in 0..4 do def n{n} {nextRealHost}
3  ### connect nodes
4  connect n0 n1
5  connect n1 n2 5ms
6  connect n2 n3
7  ### build
8  build-network
9  ### run exp
10 for m in 10..101..10 do
11   runas n0 netns n0 myuser mpirun -f h.txt ./p {m} > {m}.log
12   set-delay n1 n2 {m}ms
13 end for
```

Fig. 2. Example SherlockFog experiment script to launch a virtual topology of 4 nodes and execute an MPI application in different network conditions.

"linked-list", while setting up a 5 ms delay between nodes *n1* and *n2* (lines 4–6). Line 8 configures IP addresses for all nodes and sets up static routing tables accordingly. Finally, lines 10–13 run the actual experiment: an MPI application is repeatedly executed with argument m ranging from 10 to 100 in steps of 10, increasing latency between *n1* and *n2* on each step while saving its output for offline analysis.

3.3 Considerations When Using SherlockFog

We discuss a few usage considerations for SherlockFog:

Due to the usage of static routing, two different paths from one node to another are not allowed. Minimum Spanning Tree (MST) can be calculated on any topology to define unique paths for every pair of nodes. Real (dynamic) routing protocols on the network could give us one such path configuration.

As traffic is routed from the host carrier to the right namespace by looking up its destination MAC address, it is not possible to experiment with applications that make use of multicast messages. However, as our main focus is MPI applications and most implementations handle global communication using multiple unicast messages on some sort of virtual tree, this is not a limitation for our experiments.

Total bandwidth is shared among nodes. The user must be careful not to overflow the actual carrier. It is possible avoid this by limiting maximum bandwidth in each virtual network interface.

Finally, real link latency must be taken into account when designing the experiment. Since SherlockFog can scale on nodes on different switches, it is likely that pairwise latencies differ. They must be taken into account, as latency is increased on top of the actual link's. As it is the case for all network emulation tools, this could lead to inaccurate results if latency increments are closer to the underlying link's values.

3.4 Underlying Topology

All experiments in this work were run on an 8-node cluster of AMD Opteron 6276 processors with 128 GB of RAM. Each node has 64 cores and runs Debian GNU/Linux 8.7 amd64 with kernel version 4.9.18. MPI applications were compiled and executed using MPICH version 3.2.

Initial tests using this hardware show that no more than 48 cores can be used at the same time without incurring in performance hiccups. This is consistent with behavior of other applications used in this hardware and is related to a bottleneck of the memory bus.

4 Validation

In this section, we propose experiments to study the accuracy of our methodology in representing different network scenarios, which are defined in Table 1.

Table 1. Network topologies.

Topology name	Sizes	Description	Reference in text
Barabási-Albert	100 nodes	Random graph generated using the Barabási-Albert model for scale-free networks with preferential attachment	barabasi
Isles	16, 64 and 256 nodes	Two clusters of nodes (star topology) connected through a single path	isles

The isles topologies represent two interconnected clusters of computational resources. These clusters are connected to each other through a single distinguished link. The latency of this link indicates the distance in terms of communication. This scenario represents, for example, two sets of nodes in the edge of the network which are connected to a common infrastructure such as the Internet.

Let n be the size of the network, the process placement rules are:

1. The distinguished link connects the first node (node 0) to the last one (node $n-1$).
2. The nodes are partitioned evenly on each cluster.
   - Nodes 0 to $\lfloor \frac{n-1}{2} \rfloor$ go to the first cluster.
   - Nodes $\lceil \frac{n-1}{2} \rceil$ to $n-1$ go to the second cluster.
3. The nodes connected by the distinguished link become the exit nodes for each cluster.
4. Every other node is connected to its respective exit node.

The barabasi topology is a random graph generated using the the Barábasi-Albert model for scale-free networks with preferential attachment. It represents a connectivity model which is found on the Internet [5]. This topology was generated using model parameter $m_0 = 2$. In this case, processes are assigned randomly.

We will show that SherlockFog can emulate different network conditions by analyzing prediction output compared to the expected theoretical results.

4.1 Latency Emulation

In order to show how latency emulation works, we need to use an application with a traffic pattern for which we can obtain an analytical expression for the total communication time. By doing so, we can then compare the expected theoretical time to the output of our tool.

In particular, we have used an implementation of a passing token through a ring logical topology. Each node knows its neighbors and its order in the ring. Token size is configured to be a single integer (4 bytes) throughout this work.

The number of times the token is received by the initiator (rounds) is also a parameter of the application.

We have analyzed total number of messages on the network and execution time for this application, using two different implementations:

- **Token Ring:** implements communication using TCP sockets. This version allows us to have fine grain control of message generation and protocol.
- **MPI Token Ring:** same application, but using MPI for communications. In this case, we can test if the use of the MPI library could also be managed by our tool.

Since we know the traffic pattern, if we were to keep the topology unchanged, but increased latency of one or more links, it would be easy to estimate how much longer the application would take to complete with respect to the original network settings. This increment is calculated as follows: let N be the number of nodes in the topology, t_0 the original execution time, $c_{i,j}$ the total send count from node i to node j and $w_{i,j}$ the shortest path weight[4] from node i to node j, the expected execution time t_e is defined by:

$$t_e = t_0 + \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} c_{i,j} \cdot w_{i,j} \tag{1}$$

It is important to emphasize that Eq. 1 represents the expected execution time accurately only since **Token Ring**'s traffic pattern is sequential. Otherwise, we would have to take into account communication overlapping.

We calculated the original execution time, with a fixed latency value on all links, the estimated times for different latency settings and the actual execution times when using SherlockFog with those settings. The full description of the runs is shown in Table 2.

Table 2. Parameter configuration for validation experiments

Application	Topologies	Argument range	Latency	Comments
Token Ring	barabasi and isles	100–1000 rounds	10, 90, 170 ms	Latency increased on all edges
MPI Token Ring		100–1000 rounds	5–25 ms	Latency increased on a single edge

4.2 Token Ring

In Table 3, a partial view of the results is shown. We can observe that the predicted time differs from the measured time by less than 1% in all cases. This is also consistent with the rest of the results for all round counts, latencies and topologies in our experiment set.

[4] Each link's weight is set to how much its delay is increased with respect to the underlying platform.

Table 3. Excerpt of validation results for Token Ring, 300–500 rounds, on barabasi (100 nodes). Latency added uniformly to all edges.

Rounds	Latency (ms)	Predicted (s)	Measured (s)	Error
300	5.00	615.84	619.39	0.0057
300	15.00	1801.78	1810.63	0.0049
300	25.00	2987.72	2983.03	0.0015
300	35.00	4173.66	4177.55	0.0009
300	45.00	5359.60	5347.30	0.0022
400	5.00	820.27	826.30	0.0073
400	15.00	2400.21	2402.16	0.0008
400	25.00	3980.15	3978.04	0.0005
400	35.00	5560.09	5554.26	0.0010
400	45.00	7140.03	7130.86	0.0012
500	5.00	1024.79	1033.60	0.0085
500	15.00	2998.73	3007.53	0.0029
500	25.00	4972.67	4973.75	0.0002
500	35.00	6946.61	6943.70	0.0004
500	45.00	8920.55	8955.64	0.0039

4.3 MPI Token Ring

The MPI version produces similar results. In this case, error ranges are slightly higher, but also remain below 1% in all cases, even on a topology on which the logical order of the nodes produces a complex communication path in this application, such as barabasi. We consider that this is due to the fact that MPICH handles messaging differently than in our plain TCP implementation, though we find this difference not to be significant.

We can conclude that latency is accurately represented in our tool when executing applications that use MPI for communication on different emulated topologies.

5 Results

In this section, we show the effects of latency on different scenarios in the MPI version of NAS Parallel Benchmarks [4]. These benchmarks are derived from Computational Fluid Dynamics applications and have been throughly tested and analyzed by the community.

We have chosen three kernels (IS, CG and MG) and two pseudo applications (BT and LU) and evaluated performance loss on the isles topologies.

All benchmarks were executed using SherlockFog to increase the latency of the distinguished link up to 100 times for three different problem sizes (A, B

and C). These problem sizes are standard for this kind of applications and are defined such that going from one class to the next represents a four-fold increase. All experiments were repeated 5 times.

Our interest lies in finding out how the performance of these benchmarks–which were designed to be executed on a single cluster of nodes with low communication overhead–fares in this use case, comparing total execution time for each latency value to its no-extra-latency counterpart[5].

Each of the plots presented in this section describes the increment in total execution time as a function of the increment in latency for all network sizes. The semi-transparent patches over the curves show the standard deviation for each data series.

The results for all benchmarks are shown in Fig. 3. We can observe similar patterns for each network size.

On 16 nodes, total execution time for all network sizes grows linearly as latency is increased. In the worst case, a 100-fold latency increase results in 14 times slower total execution time. The slope of the curve is usually lower as the problem size grows: problem size C has less of an impact than the smaller sizes on most cases. We believe this to be related to the fact that each process has more work to do, reducing the impact of the overhead in communication.

On 64 nodes, the difference between problem sizes A and C is more significant. Moreover, we can also observe that the maximum increment is much lower than in the smaller network, up to 3.5 times for a 100-fold latency increase. For BT, CG and MG (problem size A), the incidence is also much more significant for smaller latency values. For example, increasing latency 10 times in CG, size A, results in the application taking 2.5 times more to complete. However, increasing latency 100 times results in it only taking 3.4 times more. LU, on the other hand, doesn't show a significant performance loss for all latency values.

On 256 nodes, we can observe similar results to 64 nodes. However, in this case, the scale is much smaller: the worst case is shown in MG, size A, which takes twice as much time to complete when subject to a 100-fold latency increment. The case of CG is also interesting, as going from no latency to 1 ms results in the benchmark taking 1.6 times more to complete. However, increasing the latency further doesn't produce a noticeable effect. This is similar to the results for 64 nodes, but the effect is more pronounced. We believe this to be related to the communication that goes through the distinguished link representing a much smaller ratio with respect to smaller networks.

Finally, in the case of IS, the increments in total execution time are much less noticeable than in the previous cases. On 16 nodes, the curves for each problem size tend to drift away from each other as the latency goes up. However, as the node count goes up, the effects of changes in latency on this topology are much less noticeable. We can conclude that this application is not greatly impaired, being the most fog-ready of all these benchmarks in this particular scenario.

[5] In the no extra latency case, the topology remains unchanged, but the latency of every link is exactly the same.

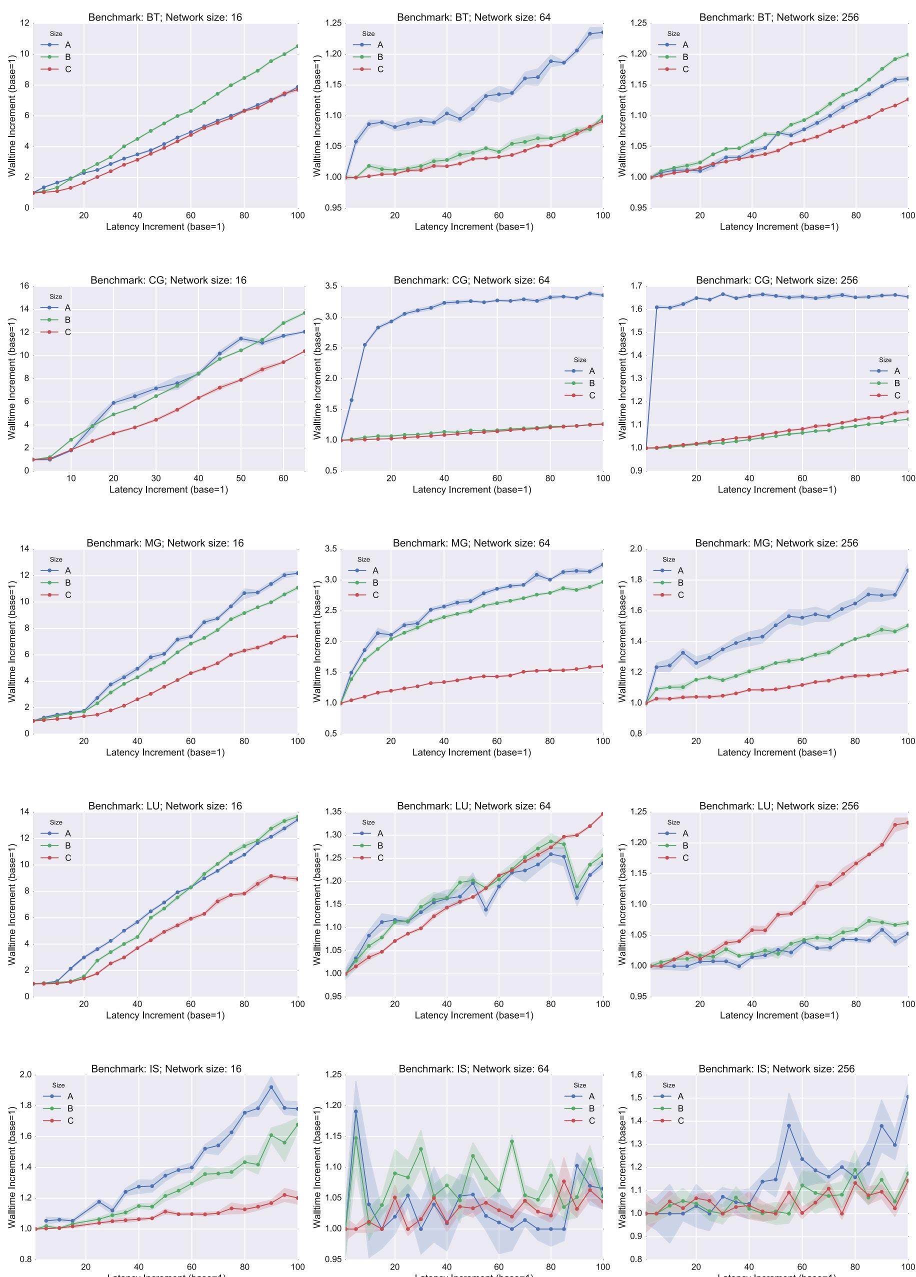


Fig. 3. Slowdown on the isles topologies as a function of the increment in latency of the distinguished link in different NAS parallel benchmarks.

6 Conclusions

In this work we introduced SherlockFog, a tool that enables experimentation with MPI applications in Fog and Edge Computing scenarios. We proposed a methodology to analyze if an MPI application can be deployed on a Fog or Edge scenario without incurring in a big performance loss, given its communication pattern and that particular network setting. Our tool also provides a testing framework to explore MPI applications and library implementations in heterogeneous scenarios.

Latency emulation in SherlockFog was validated by estimating the communication overhead in a custom application that implements a token ring. This application describes a sequential communication pattern and is therefore suitable for estimating the overhead theoretically.

We have analyzed five well-known benchmarks that use MPI to reproduce patterns in computation similar to those of CFD applications. We proposed a network topology in which two clusters are connected to each other through a single distinguished link. Using this topology, we have evaluated the impact of increasing the latency of the distinguished link on the performance of each application.

All results show a linear or sublinear impact on this particular topology, opening up opportunities to use distributed, increasingly ubiquitous computational resources.

As future work, other aspects of the Edge/Fog paradigm such as the dynamic nature of the network have to be studied. This requires adapting the MPI programming model to handle node churn and changes in logical topology. SherlockFog also models changes in bandwidth. This feature should also effect application performance but have not yet been evaluated.

Acknowledgements. The authors would like to thank D. González Márquez for his assistance with schematic drawings and the Centro de Simulación Computacional para Aplicaciones Científicas/CSC-CONICET and the Centro de Cómputos de Alto Rendimiento (CeCAR, FCEN-UBA) for providing the equipment we haved used in the experimental setup.

References

1. Dimemas. http://tools.bsc.es/dimemas. Accessed 2 Dec 2017
2. ns-3 Direct Code Execution. https://www.nsnam.org/overview/projects/direct-code-execution/. Accessed 2 Dec 2017
3. ns-3 Overview. https://www.nsnam.org/docs/ns-3-overview.pdf. Accessed 2 Dec 2017
4. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel benchmarks. Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994
5. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**, 509–512 (1999)

6. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC 2012, pp. 13–16. ACM, New York (2012). http://doi.acm.org/10.1145/2342509.2342513
7. Brandfass, B., Alrutz, T., Gerhold, T.: Rank reordering for MPI communication optimization. Comput. Fluids **80**, 372–380 (2013). Selected Contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011. http://www.sciencedirect.com/science/article/pii/S004579301200028X
8. Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. J. Parallel Distrib. Comput. **74**(10), 2899–2917 (2014). http://hal.inria.fr/hal-01017319
9. Degomme, A., Legrand, A., Markomanolis, G., Quinson, M., Stillwell, M., Suter, F.: Simulating MPI applications: the SMPI approach. IEEE Trans. Parallel Distrib. Syst. **PP**(99), 1 (2017)
10. Dichev, K., Rychkov, V., Lastovetsky, A.: Two algorithms of irregular Scatter/-Gather operations for heterogeneous platforms. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 289–293. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15646-5_31
11. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd edn. MIT Press, Cambridge (1999)
12. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface, 2nd edn. MIT Press, Cambridge (1999)
13. Hemminger, S.: Network emulation with NetEm. In: Pool, M. (ed.) LCA 2005, Australia's 6th National Linux Conference (linux.conf.au). Linux Australia, Sydney (2005). http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf
14. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, pp. 19:1–19:6. ACM, New York (2010). http://doi.acm.org/10.1145/1868447.1868466
15. Mercier, G., Clet-Ortega, J.: Towards an efficient process placement policy for MPI applications in multicore environments. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI 2009. LNCS, vol. 5759, pp. 104–115. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03770-2_17
16. Navaridas, J., Pascual, J.A., Miguel-Alonso, J.: Effects of job and task placement on parallel scientific applications performance. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 55–61, February 2009
17. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Int. Things J. **3**(5), 637–646 (2016)
18. Wette, P., Dräxler, M., Schwabe, A.: Maxinet: distributed emulation of software-defined networks. In: 2014 Networking Conference, IFIP, pp. 1–9, June 2014
19. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, pp. 255–270. USENIX Association, Boston, December 2002

Big Data and Data Management

IoT Workload Distribution Impact Between Edge and Cloud Computing in a Smart Grid Application

Otávio Carvalho(✉), Manuel Garcia, Eduardo Roloff, Emmanuell Diaz Carreño, and Philippe O. A. Navaux

Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil
{omcarvalho,magnapa,eroloff,edcarreno,navaux}@inf.ufrgs.br

Abstract. The advent of Internet of Things is now part of our reality. Increasing amounts of data are being continuously generated and monitored through widespread sensing technologies such as personal smartphones, large scale smart cities sensor deployments and smart electrical grids.

However, the ability to aggregate and act upon such data gathered by sensors is still a significant research and industrial challenge. Devices that are able to collect and act on data at network edges are bounded by the amount of data that can be sent over networks.

In this paper, we analyze the impact of workload distribution in a smart grid application, evaluating how we can increase processing rates by leveraging each time more powerful edge node processors.

Our results show that our test bed application, leveraging cloud nodes processing and processing windows, is able to achieve processing rates of approximately 800k measurements per second using four edge node processors and a single cloud node.

1 Introduction

The Internet of Things (IoT) is now a reality, and we are connecting each time more devices – such as personal consumer electronic devices, home appliances, cameras, medical devices, and all types of sensors – to the Internet environment. This ubiquity unlocks the potential to innovations that can use the data generated by those devices to enable smart cities, smart infrastructures and smart services that can improve quality of life.

By 2025, researchers estimate that the IoT will have a potential economic impact of 11 trillion per year – which would be equivalent to about 11% of the world economy. They also expect that one trillion IoT devices will be deployed by 2025. In majority of the IoT domains such as infrastructure management and healthcare, the major role of IoT is the delivery of highly complex knowledge-based and action-oriented applications in real-time [9].

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 203–217, 2018.
https://doi.org/10.1007/978-3-319-73353-1_14

Technologies and applications being created for mobile computing and the Internet of Things (IoT) are driving computing toward dispersion. Edge computing is a new paradigm in which substantial computing and storage resources – variously referred to as cloudlets, micro datacenters, or fog nodes – are placed at the Internet's edge in close proximity to mobile devices or sensors [30].

Smart grids will allow consumers to receive near real-time feedback about their energy consumption and price, enabling them to make their own informed decisions about consumption and spending. On the producer point-of-view, we can leverage home consumption data to produce energy forecasts, enabling near real-time reaction and a better scheduling of energy generation and distribution [7]. In this way, smart grids will save billions of dollars on both sides in the long run, for consumers and the generators, according to recent forecasts [26].

Since millions of end-users will be taking part into processes and information flows of smart grids, high scalability of these methods turns into an important issue. To solve these issues, cloud computing services present themselves as a viable solution, by providing reliable, distributed and redundant capabilities at global scale [10]. However, there is a large set of applications which cannot accept the delay caused by transferring data to the cloud and back, being bounded by latency. Also, it is not efficient to send a large number of small packages of data to the cloud for processing, as it would saturate network bandwidth and decrease scalability of the applications [13].

In this paper, we explore what is achievable in a realistic application for Short-Term Load Forecast (STLF), in terms of latency and bandwidth, by moving varying portions of computation from cloud to edge nodes. By building a distributed application that handles communication and processing through edge processors and cloud computing machines, we are able to distribute load between cloud and edges nodes and analyze what is possible to obtain in terms of processing speedup in comparison with a pure cloud-based application.

2 Related Work and Discussion

The state-of-the-art works on edge computing includes platforms and frameworks that aim to provide scalable processing closer to the network border, in order not only to provide low latency results, but also to better utilize resources available on the network. The approaches to solve these problems include predominantly cloud-like deployments at the edge (often described as cloudlets). The most prominent approaches and how they relate to our work are briefly described below. In the end of Sect. 2 we explain how these works relate to the proposed solution on the Sect. 3.

Femtoclouds [19], REPLISOM [1], CloudAware [24] and ParaDrop [22] are examples of applications that explore computational offloading to nearby devices. The most predominant usages either rely on offloading to edges that are underutilized or offload processing to nearby network centralizers (modified wireless network access points or specialized mobile network base station hardware). The only work that explores computation to considerably more performant nodes is EdgeIoT, which provisions virtual machines in a nearby mobile base station.

Most of the related works differ from ours by relying on hardware changes or significant modifications on the underlining communication protocols. ParaDrop applies changes to nearby wireless access points in order to provide its functionality. HomeCloud [25] relies on a Software Defined Networking (SDN) implementation on the networks in order to schedule its Network Function Virtualization (NFV) capabilities. EdgeIoT [31] relies on changes on the nearby mobile base stations in order to deploy Virtual Machines (VMs) which are used for computation offloading.

Several of the state-of-the-art works go ahead of the scope this work on the regarding of work scheduling capabilities. Cumulus [15] provide a complete framework that controls task distribution under a heterogeneous set of devices on its cloudlet. FemtoClouds and CloudAware monitor device usage on its cloudlets in order to improve device usage as part of its scheduling algorithms. CloudAware also provides a specific Application Programming Interface (API) to improve the experience of implementing applications using their framework.

Although the related works present multiple initiatives towards Edge Computing that improve computational offloading to nearby nodes, none of them explore the potential of combining the low latency edge nodes processing with scalable and more performant sets of commodity machines on public clouds.

In the next sections, we describe and evaluate our approach to this problem, applying edge processing capabilities as a complement to long running jobs on private clouds, in order to improve throughput, decrease network traffic and minimize latency.

3 Architecture and Implementation

The architectural infrastructure of our test bed application deployment can be described as a composition of three layers, as it is represented on Fig. 1: (1) Cloud layer, where we execute long running scalable jobs that can provide more performant processing at the cost of latency; (2) Edge layer, composed by edge nodes that are used to pre-process and aggregating data before sending to the Cloud; and the (3) Sensor layer, which is composed by the sensors that communicate directly with Edge layer nodes to receive actuation requests and provide measurements to the network.

3.1 Cloud Layer

In this layer, we create virtual machines to execute our application in order to aggregate data to be received from the Edge layer nodes. The Cloud layer should be composed by elements that can be able to process data as it arrives. It can be instantiated by implementing the same application logic from the layer below, but instead it should be configured to receive data from multiple edge nodes.

This layer receives data from queues and exchanges through a message hub, so that the inputs can be parallelized through multiple consumers. It can also

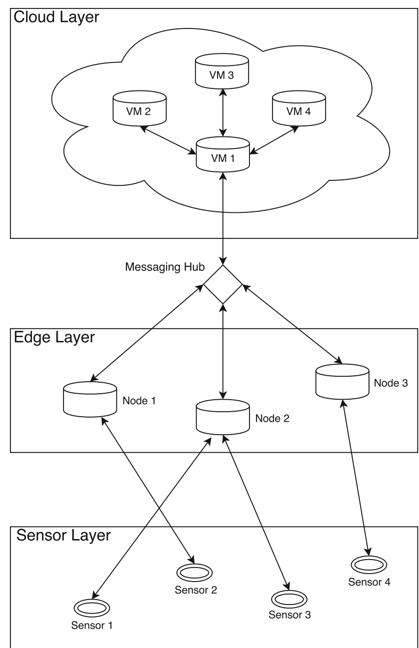


Fig. 1. The architecture is composed by 3 layers: cloud, edge and sensor

Table 1. Cloud layer configuration: virtual machine type and toolset description

Parameter	Description
Instance type	Basic_A3 (4 cores, 7 GB of RAM)
Operating system	Ubuntu 16.04 LTS
Location	Brazil south
Golang version	1.8
GRPC version	1.3.0-dev
Protocol Buffers version	3.2.0
Network interconnection	5 Mbps HFC (between edge and cloud)

be configured to support clusters of machines to execute transformations as distributed stream processing jobs over these queues and exchanges.

In our evaluation work, it is implemented as an application running in a single node inside a Linux VM at Microsoft Azure, which was chosen due to its availability at our research lab due to its cost benefits over other cloud platforms [28]. This application was written in Go programming language and receives processing request from the layer below through GRPC communication framework. The VM instance utilized was configured as it is described in Table 1.

3.2 Edge Layer

The Edge layer is composed by a set of nodes that receive sensor measurements one-at-time and executes its operators. Operators can be either transformations over results, combinations with sets of measurements received or mappings to machines on the Cloud layer above.

On this layer, the application code will be expressed to define which computation will be done inside of edge nodes and which computation will be managed by VMs on the Cloud Computing environment. The degree of control provided by this level makes possible to decrease the number of messages sent to the Cloud. In this way, it is possible to decrease the amount of data that is sent to the Cloud. Also, by processing certain amounts of data directly on the edge nodes, the latency experienced by actuator sensors is in the order of tenths of milliseconds instead of a couple of seconds of cloud processing latencies.

Actuator sensor logic can also be implemented on this layer, in such a way that when a given condition is matched by an edge node, it can trigger actuators on the Sensor layer in order to act on external applications. For example, a consumer can configure its smart grid energy meter to maintain the energy consumption below a certain level during peak cost energy hours. In this way, the smart grid meter can turn off certain machines when the average consumption reaches a certain threshold.

In our evaluation test bed, this layer was composed by a set of Raspberry Pi edge nodes connected to the internet through wireless connection. Each edge node is a complete Linux machine running our application in an ARM architecture. They were configured to communicate with the underlining sensors directly, as well as the Linux VM on the Azure Cloud service. The configuration of our edge nodes is described in details on Table 2.

Table 2. Edge layer configuration: architecture and software description

Parameter	Description
Number of edge nodes	4
Hardware	Raspberry Pi Zero W
Hardware CPU	1 GHz single core CPU
Hardware RAM	512 MB
Operating system	Raspbian Jessie Lite 4.4 (Debian Jessie based)
Golang version	1.8
GRPC version	1.3.0-dev
Protocol Buffers version	3.2.0
Wireless router	HUMAX HG100R (802.11a/b/g/n)

3.3 Sensor Layer

The Sensor layer is represented by a given set of sensors that communicate with the Edge nodes. Ideally, sensors should communicate with Edge Nodes through their available input/output hardware interconnections or lightweight wireless connection such as Bluetooth or LTE networks. However, in order to limit the analysis scope of this work, we only rely on a sensor network dataset that was previously loaded into edge nodes prior to execution of tests.

Smart grid environments rely on specific meters and plugs on households to collect data, which are provided by the energy grid provider or standardized to support only a set of accepted and verified plugs and meters types. The data types generated by these environments also need to respect a certain schema to be shared, aggregated and analyzed by the energy provider companies.

The dataset used to the evaluation of this work is based on the dataset provided by the 8th ACM International Conference on Distributed Event-Based Systems (DEBS). This conference provides competitions with problems which are relevant for the industry. In the year of 2014, the conference challenge focus was on the ability of Complex Event Processing (CEP) systems to apply on real-time predictions over a large amount of sensor data. For this purpose, household energy consumption measurements where generated, based on simulations driven by real-world energy consumption profiles, originating from smart plugs deployed in households [35]. For the purpose of this challenge, a large number of smart plugs has been deployed in households with data being collected roughly every second for each sensor in each smart plug.

3.4 Communication Protocol

Although multiple protocols for communication in IoT systems have been proposed in the recent years, the protocols in use today are still being evaluated and are subject of discussion and standardization initiatives, mainly due to advancements of internet protocols to support mobile and IoT applications. The most widely adopted protocols in use today are, respectively, MQTT [4] and CoAP [6].

One of the most prominent proposals on this area is the HTTP/2 protocol. The standard was finished in 2005 and provides several improvements over previous protocols, mainly due to the capability of multiplexing data, avoiding handshake overhead and their data compression capabilities [5,29].

As an alternative to broker centric communications protocols and synchronization costly protocols such as REST [27], Google Inc. has adopted a RPC protocol and service discovery framework Stubby/Chubby [8]. The open source version of its tool is called GRPC [18], which relies on HTTP/2 in order to avoid handshake overhead, and Protocol Buffers [16] to communicate using a binary method, which provides better data compaction by reducing the message size.

Due to the performance benefits reported from the usage of HTTP/2 protocols over standard HTTP, and their ease of usage for flexible prototyping of distributed applications, GRPC was used to build a reliable and fast communication channel for all of the communication layers implemented on this work.

GRPC as a communication platform presents several advantages over TCP only connections and communication protocols such as REST. It does not only hides complexity but also provides connections to keep alive for long periods, avoiding unnecessary handshake communication but also multiplexing several requests inside of the same channel. However, their usage is still subject of evaluation, mainly on networks with high package loss percentages, which are a limiting factor not only for HTTP/2 but also for AMQP based applications [12,17,21,32].

3.5 Measurement Algorithm

Smart grids promise to provide better control and balance of energy supply and demand through near real-time, continuous visibility into detailed energy generation and consumption patterns. Methods to extract knowledge from near real-time and accumulated observations are hence critical to the extraction of value from the infrastructure investment.

In this context, Short-Term Load Forecasting (STLF) refers to the prediction of power consumption levels in the next hour, next day, or up to a week ahead. Methods for STLF consider variables such as date (e.g., day of week and hour of the day), temperature (including weather forecasts), humidity, temperature-humidity index, wind-chill index and most importantly, historical load. Residential versus commercial or industrial uses are rarely specified.

Time series modeling for STLF has been widely used over the last 30 years and a myriad of approaches have been developed. These methods [20] can be summarized as follows:

- Regression models that represent electricity load as a linear combination of variables related to weather factors, day type, and customer class.
- Linear time series-based methods including the Autoregressive Integrated Moving Average (ARIMA) model, auto regressive moving average with external inputs model, generalized auto-regressive conditional heteroscedastic model and State-Space Models (SSMs).
- SSMs typically relying on a filtering-based (e.g., Kalman) technique and a characterization of dynamical systems.
- Nonlinear time series modeling through machine learning methods such as nonlinear regression.

Ali [2] argues that the three most accurate models for load prediction are, respectively, Multilayer Perceptron (MLP), Support Vector Machine and Least Mean Squares. Due to the model fit in relation to the distributed architecture, we decide to pursue an approach similar to the suggested by the DEBS 2014 conference committee [35], that is schematically described in Eq. (1). This approach could be interpreted as a mixed approach between MLP and ARIMA. It brings together characteristics from both Linear time series-based methods and SSMs [11].

More specifically, the set of queries provide a forecast of the load for: (1) each house, i.e., house-based and (2) for each individual plug, i.e., plug-based. The forecast for each house and plug is made based on the current load of the

connected plugs and a plug specific prediction model. The aim of these queries is not at the over the better prediction model, but at stressing the interplay between modules for model learning that operate on long-term (historic) data with components that apply the model on top of live, high velocity data.

$$L(s_{i+2}) = \frac{avgL(s_i) + median(avgL(s_j))}{2} \tag{1}$$

In the Eq. (1), $avgL(s_i)$ represents the current average load for the slice s_i. The value of $avgL(s_i)$, in case of plug-based prediction, is calculated as the average of all load values reported by the given plug with timestamps $\in s_i$. In case of a house-based prediction the $avgL(s_i)$ is calculated as a sum of average values for each plug within the house. $avgL(s_j)$ is a set of average load value for all slices s_j such that:

$$s_j = s_{i+2-n*k} \tag{2}$$

In the Eq. (2), k is the number of slices in a 24 hour period and n is a natural number with values between 1 and $floor(\frac{i+2}{k})$. The value of $avgL(s_j)$ is calculated analogously to $avgL(s_i)$ in case of plug-based and house-based (sum of averages) variants.

4 Evaluation

The evaluation of our test bed application is made in two phases: Communication evaluation and application evaluation. In each step, we evaluate aspects of the application that build on top of each other to improve our overall processing throughput, such as: Latency, message sizes, concurrency degree and message windows.

4.1 Communication Evaluation

In order to successfully evaluate our test bed middleware implementation, we first evaluate the underlining network connection. The network evaluation started by measuring the maximum amount of data that could be sent from our edge nodes to the cloud provider, using the specified network router and the given network connection described on Sect. 3.

Our measurements started by analyzing throughput through the Iperf tool [33]. Those experiments have shown us that the average throughput was slightly below the network bandwidth described by the provider, which is common on internet providers of cable connections and was expected on our measurements [14]. The results of this first measurement step are described on Table 3.

After have a glimpse about the real throughput of our underlining network, we designed a simple application in order to evaluate the performance or the GRPC middleware and the HTTP/2 protocol with varying message sizes.

The simple application designed for this task is called PingPong, which executes the following steps: (1) sends a message from the edge node to the cloud node; (2) the cloud node receives the message and sends it back to the edge node, completing a round-trip.

Table 3. Network measurements with Iperf

Connection Type	TCP window size	Interval	Transfer	Bandwidth
Edge node to edge node	43.8 KByte	60 s	388 MBytes	54.1 Mbits/sec
Edge node to cloud	43.8 KByte	60 s	7.76 MBytes	1.03 Mbits/sec

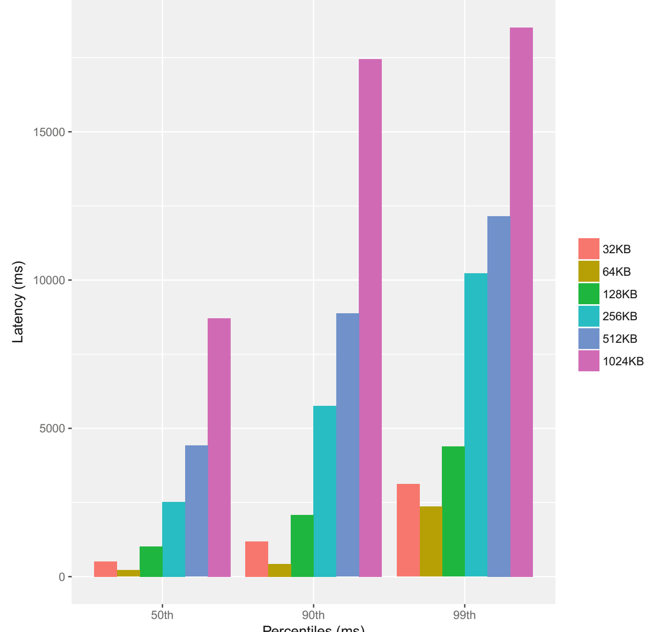


Fig. 2. PingPong: latency percentiles by message sizes (32 KB to 1 MB)

It is a known fact that distributed applications suffer from latency tails that can be several times greater than the expected average latency. On applications with multiple users that send thousands of messages per second, these fat tail latencies might be experienced by several users of the systems [3,23].

In the Fig. 2 we analyze the impact of message sizes, from 32 KB up to 1 MB, in the latency of the messages being sent over the network. As we can see, the 50th percentile (the median), is as low as a couple of milliseconds for small messages sizes, but it increases highly when we analyze the tail latencies. The impact of messages that are delayed by Garbage Collection (GC) pauses, package losses or other network failures is a limiting factor depending on the application. These measurements also serve to us a guideline to build message windows, given that we should expect, for example, messages latencies up to 2.5 s for 64 KB messages at 99th percentile.

We have also used the PingPong method to evaluate the maximum achievable network throughput under our communication middleware, which can be seen

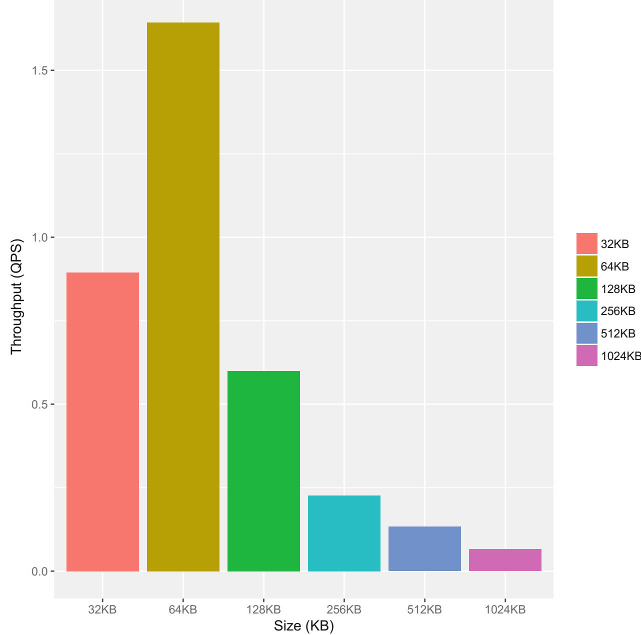


Fig. 3. PingPong: Maximum Throughput by Message Size (32 KB to 1 MB)

on Fig. 3. This evaluation was important to understand that the usage of 64 KB messages increases the throughput, probably due to a better fit on TCP windows used by the GRPC communication framework.

4.2 Application Evaluation

The application evaluation was done by distributing the aggregation step of the application processing between edge and cloud nodes. Our main objectives for this evaluation were to understand how latency affects processing throughput, as well as to analyze the performance gains obtained by aggregating data on edge nodes before sending data to the cloud.

In our evaluations, we preload our edge nodes with necessary sample sizes before running the algorithm. The schema is quite similar to the original dataset and is composed by a timestamp, the value of the energy measurement (in Watts) and an identifier id of the house/plug that is been measured. As it is compressed by the Protocol Buffers binary protocol, the final message payload size is 32 bytes.

Our evaluation can be better described in three phases: (1) Evaluation of the impact of the concurrency degree on the throughput; (2) Scalability evaluation to understand how the number of edge nodes impacts into the cloud node throughput; (3) Windowing and strategies to aggregate data on the edge node before sending data to the cloud node.

Concurrency evaluation. Given that our edge nodes execute multiple requests per second to its respective cloud nodes, it is important to explore concurrency strategies to obtain performance gains by executing multiple concurrent requests to the remote services.

In the Go programming language, the concurrency execution is done not directly through the creation of threads directly, but through Go's green threads model which are called Goroutines [34] (Fig. 4).

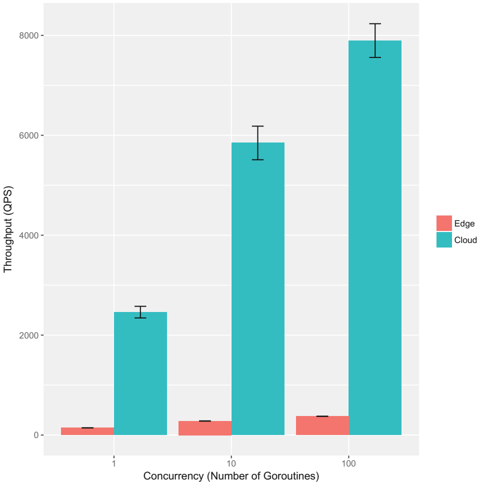


Fig. 4. Concurrency analysis: impact of Goroutines usage on throughput (edge and cloud nodes)

This experiment explored the impact of the Goroutines usage into the throughput the application. The outcome of this experiment suggests that both edge and cloud nodes are able to benefit from concurrency. Our experiments show that, in comparison with the sequential approach, it is possible to achieve a 4 times speedup on our test bed application by using 100 Goroutines.

Scalability evaluation. Another important aspect for our application is the ability of our cloud node to being able of aggregate messages from multiple edge nodes. In order to evaluate the scalability of our application, we have maintained a single cloud node and increased the number of edge nodes from one to four. As it is shown on Fig. 5, a single cloud node is able to scale linearly up to four edge nodes, each one of them maintaining an average of approximately 500 requests per second.

Impact of message windowing. Finally, in order to explore our limitations of communication bandwidth and latency, we decided to explore possibilities of aggregating multiple energy measurements into edge nodes before sending data

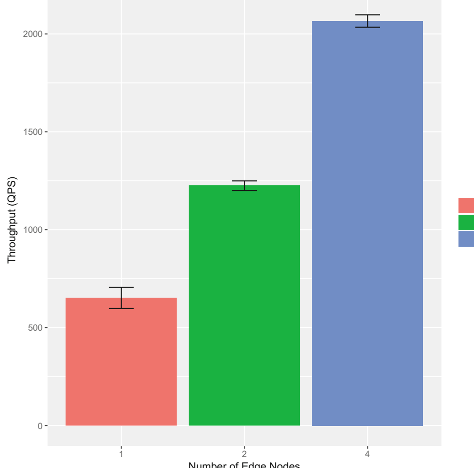


Fig. 5. Scalability analysis: throughput with multiple consumers (1 to 4 edge nodes)

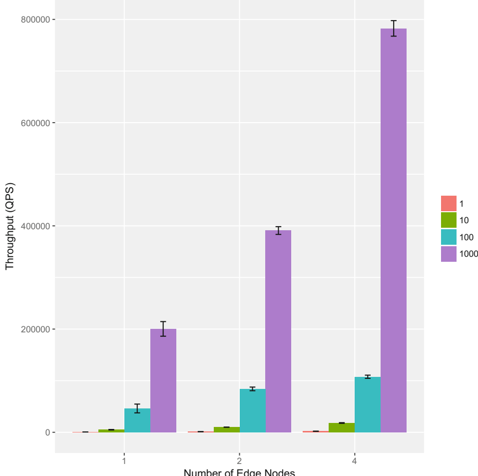


Fig. 6. Windowing analysis: windowing impact on throughput (1 to 1000 messages per request)

to cloud nodes. Prior to that, our experiments relied on the idea of receiving measurements on edge nodes, sending data to be processed on the cloud and finally receiving the updated forecast for each new measurement.

In our experiments, as it is presented on Fig. 6, we have analyzed the behavior of the test bed application when processing locally grouped sets of messages before sending to the cloud. In this way, we could evaluate our assumption that by processing more messages at the edge, and reducing the number of requests to the cloud node, we would be able to improve our overall throughput and

increase scalability (by decreasing the number of measurements being processed on the cloud node).

Our results show that the overall application (the combination of cloud and edge nodes) was able to process almost 800k messages per second by using 4 edge nodes and window sizes of 1000 combined messages.

The approach we have used to build windows of processing consists in aggregating the local measurements into a unified representation of the set, so that we could send the grouped view with the same payload size a single message of the window.

5 Conclusion and Future Works

In this work, we have studied the implementation of a smart grid application over a mixed cloud and edge processing middleware. This application was able to achieve a higher throughput by leveraging processing on edge nodes and data aggregation to reduce communication with the cloud environment.

In our future works, we would like to explore how the middleware used could be evolved into a generic framework for applications that span processing over edge and cloud. Also, it would be important to study how other communication protocols behave in this given setup.

Furthermore, we would like to explore the possibility of providing a generic method to schedule distributed tasks on this system, as well as to provide operators and potentially a query language to specify those generic data aggregations.

Acknowledgments. This research received partial funding from CYTED for the RICAP Project.

It has also received partial funding from the EU H2020 Programme and from MCTI/RNPBrazil under the HPC4E project, grant agreement no. 689772.

Additional funding was provided by FAPERGS in the context of the GreenCloud Project.

References

1. Abdelwahab, S., Hamdaoui, B., Guizani, M., Znati, T.: REPLISOM: disciplined tiny memory replication for massive IoT devices in LTE edge cloud. IEEE Internet Things J. **3**(3), 327–338 (2016)
2. Ali, A.B.M.S.: Smart Grids: Opportunities, Developments, and Trends. Springer Science & Business Media, Berlin (2013). https://doi.org/10.1007/978-1-4471-5210-1
3. Bailis, P., Kingsbury, K.: The network is reliable. Queue **12**(7), 20 (2014)
4. Banks, A., Gupta, R.: MQTT Version 3.1.1. OASIS standard (2014)
5. Belshe, M., Thomson, M., Peon, R.: Hypertext transfer protocol version 2 (HTTP/2). Internet Engineering Task Force (IETF) - RFC-7540 (2015)
6. Bormann, C., Castellani, A.P., Shelby, Z.: CoAP: an application protocol for billions of tiny internet nodes. IEEE Internet Comput. **16**(2), 62–67 (2012)

7. Brown, R.E.: Impact of smart grid on distribution system design. In: 2008 IEEE Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century, pp. 1–4. IEEE (2008)
8. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on Operating systems design and implementation, pp. 335–350. USENIX Association (2006)
9. Buyya, R., Dastjerdi, A.V.: Internet of Things: Principles and paradigms. Elsevier, Amsterdam (2016)
10. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. **25**(6), 599–616 (2009)
11. Bylander, T., Rosen, B.: A perceptron-like online algorithm for tracking the median. In: International Conference on Neural Networks, vol. 4, pp. 2219–2224. IEEE (1997)
12. Chowdhury, S.A., Sapra, V., Hindle, A.: Is HTTP/2 more energy efficient than HTTP/1.1 for mobile users? PeerJ PrePrints **3**, e1280v1 (2015)
13. Dastjerdi, A.V., Buyya, R.: Fog computing: helping the internet of things realize its potential. Computer **49**(8), 112–116 (2016)
14. Dischinger, M., Haeberlen, A., Gummadi, K.P., Saroiu, S.: Characterizing residential broadband networks. In: Internet Measurement Conference, pp. 43–56 (2007)
15. Gedawy, H., Tariq, S., Mtibaa, A., Harras, K.: Cumulus: a distributed and flexible computing testbed for edge cloud computational offloading. In: Cloudification of the Internet of Things (CIoT), pp. 1–6. IEEE (2016)
16. Gligorić, N., Dejanović, I., Krčo, S.: Performance evaluation of compact binary XML representation for constrained devices. In: 2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS), pp. 1–5. IEEE (2011)
17. Goel, U., Steiner, M., Wittie, M.P., Flack, M., Ludin, S.: HTTP/2 performance in cellular networks. In: ACM MobiCom (2016)
18. Google: gRPC Motivation and Design Principles (2015). http://www.grpc.io/blog/principles
19. Habak, K., Ammar, M., Harras, K.A., Zegura, E.: Femto clouds: leveraging mobile devices to provide cloud service at the edge. In: 2015 IEEE 8th International Conference on Cloud Computing (CLOUD), pp. 9–16. IEEE (2015)
20. Kyriakides, E., Polycarpou, M.: Short term electric load forecasting: a tutorial. In: Chen, K., Wang, L. (eds.) Trends in Neural Computation, vol. 35, pp. 391–418. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-36122-0_16
21. Lee, S., Kim, H., Hong, D.K., Ju, H.: Correlation analysis of MQTT loss and delay according to QoS level. In: 2013 International Conference on Information Networking (ICOIN), pp. 714–717. IEEE (2013)
22. Liu, P., Willis, D., Banerjee, S.: ParaDrop: enabling lightweight multi-tenancy at the network's extreme edge. In: IEEE/ACM Symposium on Edge Computing (SEC), pp. 1–13. IEEE (2016)
23. Maas, M., Harris, T., Asanovic, K., Kubiatowicz, J.: Trash day: coordinating garbage collection in distributed systems. In: HotOS (2015)
24. Orsini, G., Bade, D., Lamersdorf, W.: CloudAware: a context-adaptive middleware for mobile edge and cloud computing applications. In: IEEE International Workshops on Foundations and Applications of Self* System, pp. 216–221. IEEE (2016)

25. Pan, J., Ma, L., Ravindran, R., TalebiFard, P.: HomeCloud: an edge cloud framework and testbed for new application delivery. In: 2016 23rd International Conference on Telecommunications (ICT), pp. 1–6. IEEE (2016)
26. Reuters: U.S. Smart Grid to Cost Billions, Save Trillions (2011). http://www.reuters.com/article/2011/05/24/us-utilities-smartgrid-epri-idUSTRE74N7O420110524
27. Richardson, L., Ruby, S.: RESTful web services. O'Reilly Media, Inc., Sebastopol (2008)
28. Roloff, E., Diener, M., Carissimi, A., Navaux, P.O.A.: High performance computing in the cloud: deployment, performance and cost efficiency. In: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 371–378. IEEE (2012)
29. Ruellan, H., Peon, R.: HPACK: Header Compression for HTTP/2. Internet Engineering Task Force (IETF) - RFC-7541 (2015)
30. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1), 30–39 (2017)
31. Sun, X., Ansari, N.: EdgeIoT: mobile edge computing for the internet of things. IEEE Commun. Mag. **54**(12), 22–29 (2016)
32. Thangavel, D., Ma, X., Valera, A., Tan, H.X., Tan, C.K.Y.: Performance evaluation of MQTT and CoAP via a common middleware. In: 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), pp. 1–6. IEEE (2014)
33. Tirumala, A., Qin, F., Dugan, J., Ferguson, J., Gibbs, K.: Iperf: the TCP/UDP bandwidth measurement tool (2005). http://iperf.fr
34. Togashi, N., Klyuev, V.: Concurrency in Go and Java: performance analysis. In: 2014 4th IEEE International Conference on Information Science and Technology (ICIST), pp. 213–216. IEEE (2014)
35. Ziekow, H., Jerzak, Z.: The DEBS 2014 grand challenge. In: Proceedings of the 8th ACM International Conference on Distributed Event-based Systems, DEBS, vol. 14 (2014)

Model-R: A Framework for Scalable and Reproducible Ecological Niche Modeling

Andrea Sánchez-Tapia[1], Marinez Ferreira de Siqueira[1(✉)], Rafael Oliveira Lima[1], Felipe Sodré M. Barros[2], Guilherme M. Gall[3], Luiz M. R. Gadelha Jr.[3(✉)], Luís Alexandre E. da Silva[1], and Carla Osthoff[3]

[1] Rio de Janeiro Botanic Garden, Rio de Janeiro, Brazil
{andreasancheztapia,marinez,rafael,estevao}@jbrj.gov.br
[2] International Institute for Sustainability, Rio de Janeiro, Brazil
f.barros@iis-rio.org
[3] National Laboratory for Scientific Computing, Petrópolis, Brazil
{gmgall,lgadelha,osthoff}@lncc.br

Abstract. Spatial analysis tools and synthesis of results are key to identifying the best solutions in biodiversity conservation. The importance of process automation is associated with increased efficiency and performance both in the data pre-processing phase and in the post-analysis of the results generated by the packages and modeling programs. The Model-R framework was developed with the main objective of unifying pre-existing ecological niche modeling tools into a common framework and building a web interface that automates steps of the modeling process and occurrence data retrieval. The web interface includes RJabot, a functionality that allows for searching and retrieving occurrence data from *Jabot*, the main reference on botanical collections management system in Brazil. It returns data in a suitable format to be consumed by other components of the framework. Currently, the tools are multi-projection, they can thus be applied to different sets of temporal and spatial data. Model-R is also multi-algorithm, with seven algorithms available for modeling: BIOCLIM, Mahalanobis distance, Maxent, GLM, RandomForest, SVM, and DOMAIN. The algorithms as well as the entire modeling process may be parametrized using command-line tools or through the web interface. We hope that the use of this application, not only by modeling specialists but also as a tool for policy makers, will be a significant contribution to the continuous development of biodiversity conservation analysis. The Model-R web interface can be installed locally or on a server. A software container is provided to automate the installation.

Keywords: Species distribution modeling
Ecological niche modeling · Science gateways · Scalability · Provenance

1 Introduction

Ecological Niche Modeling (ENM) has been widely used for over a decade [1–4]. In recent years ENM approaches have become an essential tool for species conservation, ecology and evolution studies, as well for systematic conservation and

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 218–232, 2018.
https://doi.org/10.1007/978-3-319-73353-1_15

restoration planning [5]. These models use species occurrence data and predictor variables that are combined to form statistical and theoretical models resulting in projections in the geographic space that represent the potential geographic distribution of a species [6]. The environmental suitability maps [7], generated by the models inform how similar a particular area is to the area where the species occurs, thus identifying the potential area for occupation by the species, from the predictor variables selected.

Ecological niche modeling comprises several stages, which require knowledge of many concepts and techniques related to various fields of biology, such as biodiversity, biogeography, as well as climate and data processing tools, before, during and after obtaining the model [5,8]. The biotic data processing step consists of obtaining, evaluating and preparing the points of presence and, in some cases, of absence of the species to be modeled. In this process, it is fundamental to perform data cleaning with the removal of inaccurate or unreliable data. In the step of treatment and choice of environmental layers, one obtains and selects the layers to be used in the analysis. Traditionally, it is necessary to use a Geographic Information System (GIS) tools for clipping and adjusting the resolution and cropping the raster layers to the modeling extension, requiring a reasonable knowledge of the tool. This task can be even more time-consuming when dealing with a large dataset. The use of specific data types by the algorithms, and their various forms of parametrization, requires a reasonable knowledge of programming for their full use. The importance of process automation is associated with increased efficiency and performance both in the data pre-processing phase and in the post-analysis of the results generated by the packages and modeling programs, which is the main objective of this work. The elimination of external tools for data acquisition and preparation, as well as their standardization, reduces the possibility of errors, confers reproducibility and improves the speed of the modeling process, making the whole process more efficient.

The modeling process consists of many steps, as described in [8], some of which consume considerable time to be performed by traditional means. A resource available for tackling this problem is the R statistical environment, which features various possibilities of automation but does require some knowledge of programming for obtaining the desired outcomes in this process. The main objective of this work was to package modeling procedures as R functions and to create an application (Model-R) that allows, either via command-line or through a web interface, to perform ecological niche modeling, overcoming the most common barriers and providing approaches for data entry steps, data cleaning, choice of predictor variables, parametrization of algorithms, and post-analysis as well as the retrieval of the results. A list of acronyms and variable definitions is presented in Table 1.

2 Model-R Framework

The Model-R framework for ecological niche modeling is given by a set of ecological niche modeling functions (`dismo.mod`, `final.models`, `ensemble`), functions

for retrieving species occurrences records (Rjabot and Rgbif [49]) and the graphic user interface. It allows researchers to use their own data. The framework is divided in front and backend; some functions are presented at web interface that abstracts and automates the main steps involved in the ecological niche modeling process. This is a dynamic process, and our goal is that this interface will evolve and incorporate more and more aspects of the framework. All these components were implemented in R and are described in the subsections that follow.

2.1 Frontend

The main focus of the web application for Model-R is the development of an interface for the modeling process, allowing users without programming knowledge to perform the steps of the modeling process consistently, avoiding the concern with script coding and concentrating on the data and its processing workflow. To do so, we adapted the modeling functions into a Shiny application [9]. The Shiny package [9] is a web application framework for R, allowing the creation of interactive applications that can be accessed from devices with internet access, such as computers, tablets, and mobile phones. Thus, the application provides a graphical interface where users can easily choose biotic and abiotic data, perform the data cleaning on occurrence records, choose algorithms and their parameters. They can also download the results, as well as the script of the modeling process that allow its execution without the use of the Model-R web application. The use of the script as a stand-alone application allows for more precise adjustment of the parameters or adjustments that were not possible in the web interface. To make the application and process user-friendly, we separated the features by steps, following the modeling process described in [8].

The following steps of the modeling process are available in the application: biotic data entry; data cleaning; choice of abiotic variables; cutting off the geographic extension; choosing the algorithm and its parameters; visualization of results; and downloading the resulting data.

Biotic data entry. This stage represents the entry of biotic data in the system. A modeling project can be created using the "Create Project" feature, this allows for keeping track of the modeling experiments performed. Creating a project allows one to assign a name and thus organize and store the information generated. Biotic data can be given as input to the application in three ways: queries to the GBIF database, queries to the *Jabot* database, and uploading CSV files. CSV files allow for uploading occurrence records not present in GBIF and *Jabot* from other databases after conversion to this format. The RJabot package makes the query to the *Jabot* database (Fig. 1). These records are given by species name, latitude, and longitude. At the end of the biotic data entry step, a map with the occurrence records is displayed. In July 2017, GBIF contained approximately 10 million species occurrence records about Brazil, 70% of which were published by its Brazilian node, the Brazilian Biodiversity Information System (SiBBr) [10].

Fig. 1. Output of *getOccurrence* showing occurrence points obtained from Jabot.

Data cleaning. This step allows cleaning the biotic data entered into the application. It has two features: “Eliminate duplicate data” and “Delete Occurrence point”. “Eliminate Duplicate Data” removes occurrence records entered to the application that have the same value for latitude and longitude. “Delete occurrence point” eliminates points that were evaluated by the user as erroneous in their location and will not be used in modeling. Using the interface, the user clicks the button “Delete duplicate” or selects the point it wants to eliminate suspicious data. After that, the user can also save the final biotic dataset, after the data cleaning process.

Abiotic data entry. This step is responsible for the entry of abiotic variables and definition of the geographic extensions of the modeling process and its projection. The first step is to set the spatial extension of modeling process (i.e.: the extent to which the modeling will be done, also understood as study area). The extensions can be defined directly on the map, which displays the occurrence points selected in the previous steps. Regarding spatial projection, the application allows users to define a different extent to project ENM in another region. This can be useful, for instance, for checking the ability of a species to become invasive in the given region. Also, it is possible to define a projection in time, in instance, to the past (Pleistocene/Holocene) or future (2050 and 2070) using Worldclim dataset[1] and Bio-ORACLE variables [11]. Independently of the spatial and temporal projection chosen, the user might define the spatial resolution (i.e. pixel size) of the abiotic dataset. For development, we used the resolution of 10 arc minutes (for Worldclim variables) and 9.2 km (for Bio-ORACLE

[1] http://www.worldclim.org.

variables), due to storage space and processing speed reasons. The main database technologies that optimize storage and speed processing are already under study, so the application supports others resolutions, like 30 s, 2.5, 5 arc minutes.

The map, the occurrence points, and the geographic extensions are displayed using the Leaflet [12]. The package allows for zooming and interacting with the map. The application is configured to work with Wordclim and Bio-ORACLE to retrieve abiotic data and allow for other variables to be added manually to the application.

Once the abiotic variables are defined, the Model-R application displays the variables considering the extension defined by the user, and a table with charts containing the correlation values between them (see Fig. 2, step 4), allowing to verify the correlated variables. Strongly correlated variables can impair the prediction performance and statistics of the modeling process [13,14].

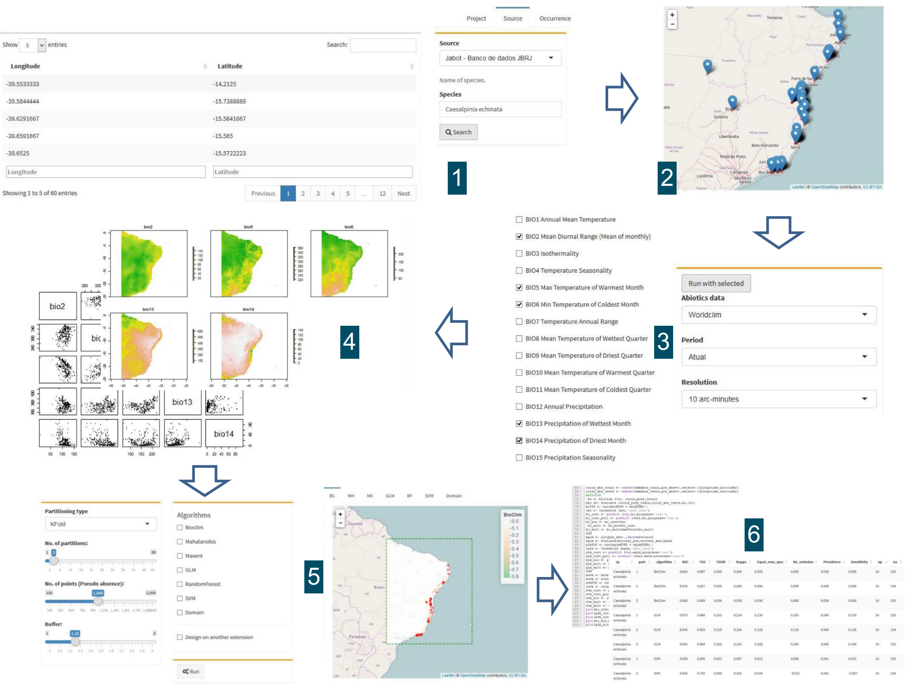


Fig. 2. Modeling steps in the web interface of Model-R.

3 Modeling Process and Backend

The next step in the web application, modeling process, is the core of the species distribution modeling workflow and was implemented as a three-step procedure, wrapped in R functions, called `dismo.mod()` (in reference to the dismo

package [15] from which it draws the main structure and functions), `final.models()` and `ensemble()`.

`dismo.mod()` takes the input data, partitions it by cross-validation, fits the models for each partition and writes the evaluation statistics tables, using function `evaluate ()` in the dismo package, with some modifications, such as the calculation of TSS for each partition. It writes the raw models, i.e. the continuous outputs, in raster and image formats. Writing to the hard disk allows keeping the main memory uncluttered. The structure of the function draws both on the dismo [15] and the biomod2 [16] tutorials.

`final.model()` joins the fitted models for each partition into a final model per species per algorithm. It can select the best partitions according to their TSS or AUC value. The default is selecting by $TSS > 0.7$, but this can be changed by the user. The function also allows choosing which algorithms will be processed. Otherwise, it will read all algorithms available from the statistics table, and to use a mask to crop the models to a subset of the fitting geographic area. Finally, it cuts the continuous models by the threshold that maximizes the TSS of the model and averages these models.

`ensemble()` computes the average of the final models, to obtain an ensemble model per species and retaining only the algorithms and partitions that were selected previously. It can also focus on the areas where the algorithms exhibited consensus. The default is 0.5, which corresponds to a Weighted Majority Rule Ensemble to reduce variability between the algorithms in final models so that the final models only retains areas predicted by at least half of the algorithms [17].

The application interface runs this framework in the background, but the user can adjust the following parameters:

Partition Number. The number of times the model will be generated for each selected algorithm and, consequently, the number of times the k-fold partitioning will be performed. (dividing the total data set in k mutually exclusive subsets of the same size, using $k-1$ for parameter estimation and algorithm training and the remaining subset to evaluate the accuracy of the model).

Number of pseudo-absences. Number of points sampled randomly in the background for use in the modeling process.

Modeling algorithms. Seven algorithms are available: BIOCLIM, Mahalanobis, Maxent, DOMAIN available in the dismo package [15] GLM ('stats'), RandomForest ('randomForest' [50]) and SVM ('kernlab' [51]). BIOCLIM, Malahanobis, and DOMAIN are based on simple statistical techniques, such as environmental distance. GLM is based on regression techniques. Lastly, Maxent, RandomForest, and SVM are based on machine learning techniques.

Buffer should be applied during the sampling of pseudo-absences. This is an inclusive buffer, it calculates the distance between the occurrence points and use the maximum or the mean geographic distance between the occurrences of the species within which pseudo-absences will be generated.

Project on another extension. The application reprojected the model to different extensions (spatial or temporal) from the modeling process obtained on the creation extension.

At the end of the execution, k continuous models, k binary models and one ensemble model are generated for each species and algorithm, as displayed in Fig. 2 (step 5). The values obtained from the validation process are stored as a table, and their values are presented in Fig. 2 (step 6). A brief description of each variable is presented in Table 1.

Table 1. Description of variables generated by the modeling process.

Variable name	Description
Sp	Species name
Part	Partition number
Algorithm	Modeling algorithm employed
AUC	Computed Area Under Curve
TSS	True skill statistic = (sensitivity + specifity) - 1
Kappa	Cohen's Kappa coefficient
No omission	Threshold where there is no omission
Prevalence	Prevalence
Sensitivity	Sensitivity
TSSth	Threshold = (sens + esp)
Np	Number of presences
Na	Number of absences

4 Reproducibility

Provenance information [18] is given by the documentation of the conception and execution of computational processes, including the activities that were executed and the respective data sets consumed and produced by them. Applications of provenance include reproducibility of computational processes, sharing and reuse of knowledge, data quality evaluation and attribution of scientific results [19]. Reproducibility is one of the important features of Model-R. The inclusion of this feature is motivated by many academic journals recommending that authors of computational studies should also provide the required data sets, tools, and workflows used to generate the results [20,21] so that reviewers and readers could better validate them. For each modeling project specified and executed in Model-R, the following information is available for download: the R script, illustrated in Fig. 2 (step 6) that allows for reproducing the steps that were performed to produce results of the modeling process and to re-execute the modeling process

without using the web interface of Model-R; a CSV file containing the resulting variables from the modeling process; the occurrence records used after data cleaning; the raster files in the GeoTIFF format generated by the application; a raster file in the GeoTIFF format with an ensemble of the models generated; raster files in the GeoTIFF format with the projection of the model into another geographic extension. These are only generated when the "Project into another extension" option is selected.

5 Case Study and Evaluation

A case study was performed with woody plants of the Brazilian Atlantic Forest and is described next.

Species occurrence data. The original plant names database (3,952 plant names and 171,144 original records) were compiled from SpeciesLink[2] and NeoTropTree[3] (List of species with number of records – appendix 1) and corrected according to the Catalog of Plants and Fungi of Brazil (CPFB)[4], using R package flora [22], which is based on the List's IPT database. The CPFB publishes the official List of the Brazilian Flora, meeting Target 1 of the Global Strategy for Plant Conservation. The catalog recognized and checked 3,910 names. The 42 names that were not found by the LSBF were looked for in The Plant List[5] (TPL) and then in the Taxonomic Name Resolution Service[6] (TNRS), as implemented in the R packages Taxonstand [23] and taxize [24,25]. The information from the CPFB, TPL, and TNRS was cross-checked, and when there were conflicts, the names from the CPFB were given priority. For each species the complete occurrence data was treated for (1) records that fell out of the Brazilian limit, (2) duplicated records, (3) non-duplicated records that fell in the same 1 km-pixel. Only species with at least 100 unique occurrences (deleting duplicated within each pixel) were maintained, and of these, to overcome bias of with marginal occurrence for Atlantic Rainforest, only species with more than 50% of occurrences in the Atlantic Rain Forest were considered. After all these procedures, a sub-sample of the 96 species (35,672 presence records) that presented the largest numbers of samples was chosen to compose the given woody plants case study (Fig. 3, left).

Environmental data. As environmental predictors, 28 variables with spatial resolution of 1 km^2 were compiled and organized. Those variables were summarized by PCA axes, from which the first ten axes (about 95% of the data variation) were used to run models. Aspect variable was edited and had its sin and cosin created to be used as variables.

[2] http://splink.cria.org.br/.
[3] http://prof.icb.ufmg.br/treeatlan/.
[4] http://floradobrasil.jbrj.gov.br/.
[5] http://www.theplantlist.org/.
[6] http://tnrs.iplantcollaborative.org/.

Environmental Niche Modeling. Environmental niche models were built for each species, using dismo.mod, final.model, and ensemble functions.

A three-fold cross validation procedure was performed. Random pseudo-absence points ($nback = 2 \times n$) were sorted within a maximum distance buffer (the radius of the buffer is the maximal geographic distance between the occurrence points) and divided into three groups, for training and testing purposes.

For each partition ($k = 3$) and algorithm, a model was built, and its performance was tested by calculating the True Skill Statistic [26]. The authors found that TSS scores were largely unaffected by prevalence and values from 0.6 to 1.0 were considered as a good adjustment of the model accuracy. Because of that, only models with $TSS > 0.7$ were retained. Selected partitions were cut by the threshold that maximizes their TSS, and the resulting binary models were averaged to generate a model per algorithm. The scale in these final models is equivalent to the number of partitions that predict the species presence (it goes from 0 to $\frac{n}{n}$ in $\frac{1}{n}$ intervals where n is the number of selected models). The ensemble model (e.g. joining models from different algorithms) was obtained by averaging the final models for each algorithm. A species potential richness map was generated by summing the binary final models, cut by the average threshold that maximizes TSS values (Fig. 3, right).

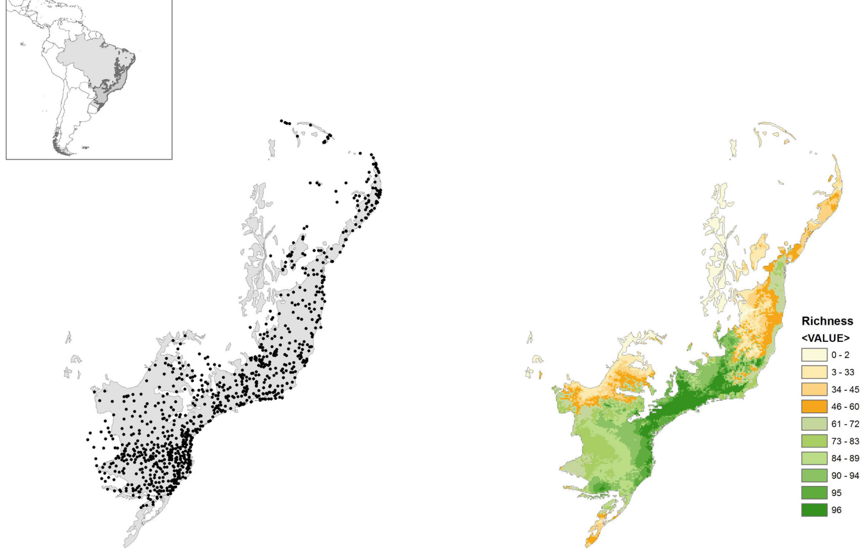


Fig. 3. Map with original occurrence records (left) and richness map generated by analyzing Model-R output data (right).

Performance and Parallelization. The `dismo.mod()` function, in which the modeling process of Model-R is based, was entirely sequential in its first version.

Models for all species of interest were generated one after another. To improve performance, parallel processing was employed. Now, if n cores are available, models for n species can be generated simultaneously. The snowfall [27] R package provided support for the parallelization. It provides functions for parallel collection processing. `sfLapply`, for instance, is the parallel version of the standard `lapply`, which applies some function to every element of an array, producing a new array with the results.

The effects of the parallelization on performance can be seen in Fig. 4. Each point in the plot is the arithmetic mean of the time elapsed to do three executions of `dismo.mod()`. Models for 96 species were generated, varying the number of cores from 1 to 64. The algorithms used were RandomForest, SVM, and Maxent. The variability in execution time for 96 species can be explained in part by the parallelization strategy used, i.e. one thread per species. The total time that it takes to apply all the modeling algorithms can be significantly different from one species to another.

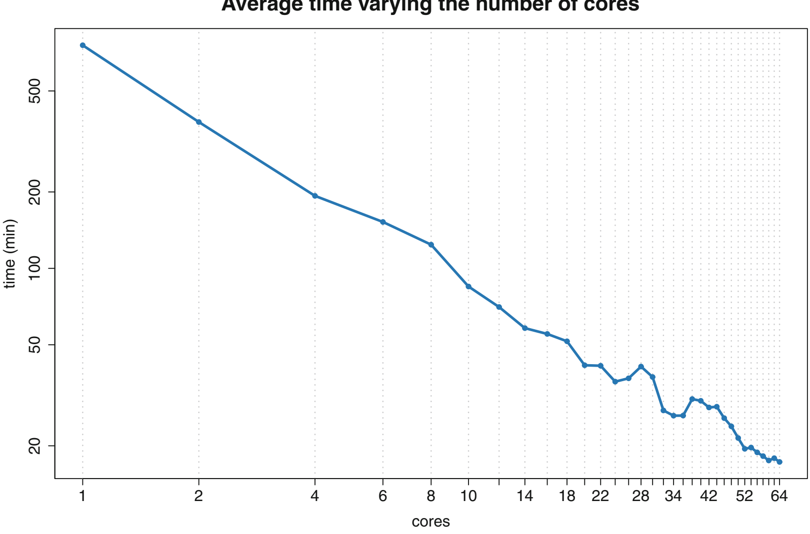


Fig. 4. Parallelization effects on performance.

The creation of separate functions for each of the modeling algorithms that `dismo.mod()` can fit was another important optimization. In its first version, all algorithms were generating models in the context of a single function. The memory allocated to the variables used by one algorithm was never released even if the referenced algorithm had finished its work. R is a programming language with garbage collector [28] meaning it releases memory when an object is no longer used. It does this by counting how many references point to each object and when the counter reaches zero, removes that object. Since `dismo.mod()` was

keeping at least one reference to the variables used by all selected algorithms for all the runtime of the function, a lot of memory was being occupied unnecessarily. The separation did not make the modeling process faster but allowed the generation of more models per node because of the smaller memory footprint. The generation of models for a single species was performed using approximately 5 GB of resident memory. Resident memory is a metric that gets closer to the actual memory budget of a process [29]. The version with separate functions for each modeling algorithm uses half of this memory.

6 Related Work

As parameters for comparison, two related services in this area were considered: the *Biomodelos* portal [30], developed by the Humboldt Institute in Colombia, and the Virtual Biodiversity e-Laboratory (BioVel) [31], an initiative supported by the European Union. These two examples were chosen because they represent two distinct efforts from the standpoint of the internal and external target audience of the system.

BioVel provides, via a web interface, a service that allows management of scientific workflows [32] for biodiversity. Several pre-defined activities can be composed to form these workflows, as an example, the following features were developed and are available in BioVel: geographic and temporal selection of occurrences; data cleaning; Taxonomic name resolution; modeling algorithms ecological niches (openModeller) [33]. Such activities can be composed freely in complex scientific workflows for performing various analyses on biodiversity. This flexibility of service and the range of applications available in its catalog, generate a plurality of results provided by the service that can be difficult to assess regarding quality and suitability, since the service is freely accessible and does not have a methodology systematic qualification of models where experts can criticize, comment and change the generated results.

The Biomodelos portal [30] is intended for species distribution modeling, which is carried out and published on the website by the Humboldt Institute modeling team. The most interesting feature of biomodelos, absent from similar portals is the existence of a network of taxonomists, who are also users of the portal, which evaluates each species distribution published on the website. The taxonomy experts have access to metadata about how the species distribution models were executed and can assign a note to the generated model, add notes to them or geographically edit the distribution map (excluding, for example, records in areas with known absences). Thus, species distributions published in Biomodelos are accompanied by information to support the decision maker in assessing their quality and fitness for the use.

Other initiatives based on R include SDM [34] and Wallace [35], and based on scientific workflow management system include Kepler [36], VisTrails [37, 38] or in the cloud computing environment [39–41]. They have some similarity with our application as well as some striking differences, especially in terms of functionality, such as the lack of scalability in the implementation of the models and the absence of provenance recording.

7 Conclusion

In this work, we presented Model-R, a framework for species distribution modeling. It abstracts away cumbersome steps usually involved in this type of modeling, such as data acquisition and cleaning, by providing a productive web interface that allows for customizing key steps of the process, including the pre-processing of biotic and abiotic data and the post-analysis of the results. The RJabot package, for instance, allows for easy retrieval of species occurrence records from the Rio de Janeiro Botanical Garden Herbarium (RB) [42], one of the most complete sources of information about the Brazilian flora. The scalable execution of the modeling process is enabled through the use of parallel programming libraries available for the R environment. Having separate functions per algorithm also presents an opportunity for further exploration of parallelism. Currently only parallelism by species is used. All models for a given species are generated by the same core even if more than one algorithm is used. Parallelism by algorithm is feasible as well. Model-R also enables reproducibility of the modeling process by providing the data sets generated and scripts in R that allow for reproducing the steps used to generate them. The application supports applying the modeling process to different sets of temporal or spatial data. Maxent, RandomForest, and all the algorithms supported by the dismo package are supported by Model-R, and their parameters can be customized through its web interface. We expect the application to become a valuable tool for scientist working with analysis and synthesis of biodiversity data, and for decision-makers in biodiversity conservation.

As future work, we plan to better automate the generation of raster files containing abiotic data by using GIS tools, such as PostGIS. These are currently generated manually for some pre-defined resolutions and copied to the Model-R application server. We also plan to further improve the scalability of the application by adapt it to run on petascale computing resources of the Brazilian National System for High-Performance Computing[7] [43] using the Swift [44] parallel scripting system, which gathers provenance information [45,46]. Additionally, we are working on porting the modeling scripts to Big Data platforms. In particular, we are adapting them to the Spark platform [47] using its R interface [48].

Model-R is available as open-source software on Github[8]. To facilitate its installation, we also built a software container that is available on Docker Hub[9]. This software container is synchronized to the Github repository, i.e. any update to the source code on Github triggers the production of an updated software container.

Acknowledgments. This work has been supported by CNPq (Grants 461572/2014-1 SiBBr - SEPED/MCTIC and 441929/2016-8 Edital MCTI/CNPQ/Universal).

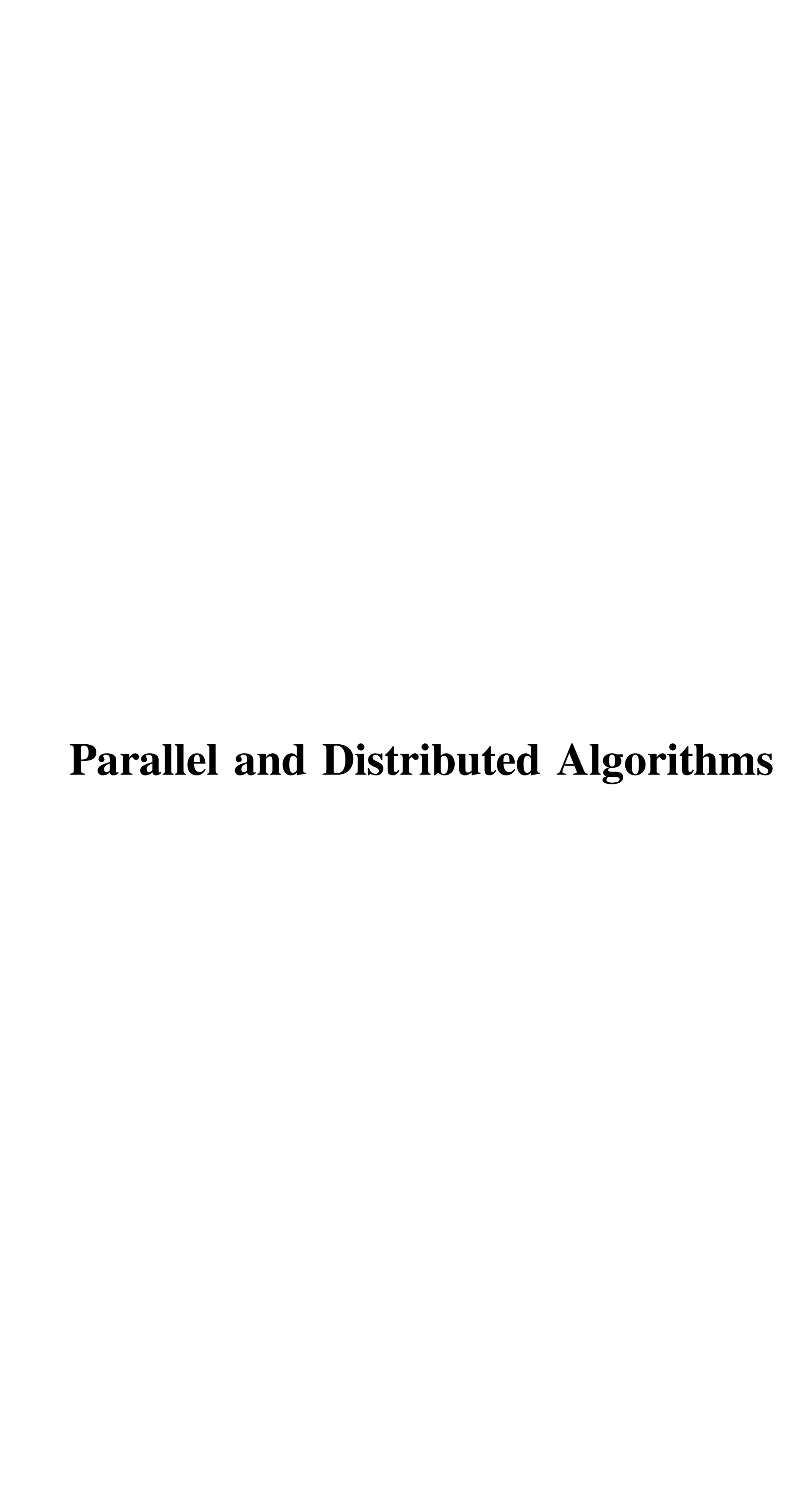
[7] http://sdumont.lncc.br.
[8] https://github.com/Model-R/Model-R.
[9] https://hub.docker.com/r/modelr/shinyapp/.

References

1. Araújo, M.B., Williams, P.H.: Selecting areas for species persistence using occurrence data. Biol. Conserv. **96**(3), 331–345 (2000)
2. Engler, R., Guisan, A., Rechsteiner, L.: An improved approach for predicting the distribution of rare and endangered species from occurrence and pseudo-absence data. J. Appl. Ecol. **41**(2), 263–274 (2004)
3. Ortega-Huerta, M.A., Peterson, A.T.: Modelling spatial patterns of biodiversity for conservation prioritization in North-Eastern Mexico. Divers. Distrib. **10**(1), 39–54 (2004)
4. Chen, Y.: Conservation biogeography of the snake family colubridae of China. North-West. J. Zool. **5**(2), 251–262 (2009)
5. Peterson, A.T., Soberón, J., Pearson, R.G., Anderson, R.P., Martínez-Meyer, E., Nakamura, M., Araújo, M.B.: Ecological Niches and Geographic Distributions. Princeton University Press, Princeton (2011)
6. Anderson, R.P., Lew, D., Peterson, A.: Evaluating predictive models of species' distributions: criteria for selecting optimal models. Ecol. Model. **162**(3), 211–232 (2003)
7. Sillero, N.: What does ecological modelling model? A proposed classification of ecological niche models based on their underlying methods. Ecol. Model. **222**(8), 1343–1346 (2011)
8. Santana, F., de Siqueira, M., Saraiva, A., Correa, P.: A reference business process for ecological niche modelling. Ecol. Inf. **3**(1), 75–86 (2008)
9. Chang, W.: Shiny: Web Application Framework for R (2016). https://cran.r-project.org/web/packages/shiny
10. Gadelha, L., Guimarães, P., Moura, A.M., Drucker, D.P., Dalcin, E., Gall, G., Tavares, J., Palazzi, D., Poltosi, M., Porto, F., Moura, F., Leo, W.V.: SiBBr: Uma Infraestrutura para Coleta, Integração e Análise de Dados sobre a Biodiversidade Brasileira. In: VIII Brazilian e-Science Workshop (BRESCI 2014). Proceedings XXXIV Congress of the Brazilian Computer Society (2014)
11. Tyberghein, L., Verbruggen, H., Pauly, K., Troupin, C., Mineur, F., De Clerck, O.: Bio-ORACLE: a global environmental dataset for marine species distribution modelling. Global Ecol. Biogeogr. **21**, 272–281 (2012)
12. Agafonkin, V.: Leaflet - a JavaScript library for interactive maps (2016). http://leafletjs.com/
13. Guisan, A., Zimmermann, N.E.: Predictive habitat distribution models in ecology. Ecol. Model. **135**(2–3), 147–186 (2000)
14. Lomba, A., Pellissier, L., Randin, C., Vicente, J., Moreira, F., Honrado, J., Guisan, A.: Overcoming the rare species modelling paradox: a novel hierarchical framework applied to an Iberian endemic plant. Biol. Conserv. **143**(11), 2647–2657 (2010)
15. Hijmans, R.J., Elith, J.: dismo: Species Distribution Modeling (2016). https://cran.r-project.org/web/packages/dismo
16. Thuiller, W., Lafourcade, B., Engler, R., Araújo, M.B.: BIOMOD - a platform for ensemble forecasting of species distributions. Ecography **32**(3), 369–373 (2009)
17. Araújo, M.B., Whittaker, R.J., Ladle, R.J., Erhard, M.: Reducing uncertainty in projections of extinction risk from climate change: uncertainty in species' range shift projections. Glob. Ecol. Biogeogr. **14**(6), 529–538 (2005)
18. Freire, J., Koop, D., Santos, E., Silva, C.: Provenance for computational tasks: a survey. Comput. Sci. Eng. **10**(3), 11–21 (2008)

19. Gadelha Jr., L.M.R., Mattoso, M.: Applying provenance to protect attribution in distributed computational scientific experiments. In: Ludäscher, B., Plale, B. (eds.) IPAW 2014. LNCS, vol. 8628, pp. 139–151. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16462-5_11
20. Sandve, G.K., Nekrutenko, A., Taylor, J., Hovig, E.: Ten simple rules for reproducible computational research. PLoS Comput. Biol. **9**(10), e1003285 (2013)
21. Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., Guy, R.T., Haddock, S.H.D., Huff, K.D., Mitchell, I.M., Plumbley, M.D., Waugh, B., White, E.P., Wilson, P.: Best practices for scientific computing. PLoS Biol. **12**(1), e1001745 (2014)
22. Carvalho, G.: Flora: tools for interacting with the Brazilian flora 2020 (2016). https://cran.r-project.org/web/packages/flora/index.html
23. Cayuela, L., Oksanen, J.: Taxonstand: taxonomic standardization of plant species names (2016). https://cran.r-project.org/web/packages/Taxonstand
24. Chamberlain, S.A., Szöcs, E.: Taxize: taxonomic search and retrieval in R. F1000Research **2**, 191 (2013)
25. Chamberlain, S., Szoecs, E., Foster, Z., Boettiger, C., Ram, K., Bartomeus, I., Baumgartner, J., O'Donnell, J.: Taxize: taxonomic information from around the web (2016). https://cran.r-project.org/web/packages/taxize
26. Allouche, O., Tsoar, A., Kadmon, R.: Assessing the accuracy of species distribution models: prevalence, kappa and the true skill statistic (TSS). J. Appl. Ecol. **43**(6), 1223–1232 (2006)
27. Knaus, J.: Snowfall: easier cluster computing (based on snow) (2016). https://cran.r-project.org/web/packages/snowfall
28. Wickham, H.: Advanced R. Chapman and Hall/CRC, Boca Raton (2014)
29. Simmonds, C.: Mastering embedded linux programming. Packt, Birmingham (2015)
30. Biomodelos: Instituto Alexander von Humboldt (2016). http://biomodelos.humboldt.org.co
31. Vicario, S., Hardisty, A., Haitas, N.: BioVeL: Biodiversity virtual e-Laboratory. EMBnet.journal **17**(2), 5 (2011)
32. Liu, J., Pacitti, E., Valduriez, P., Mattoso, M.: A survey of data-intensive scientific workflow management. J. Grid Comput. **13**(4), 457–493 (2015)
33. Souza Muñoz, M.E., Giovanni, R., Siqueira, M.F., Sutton, T., Brewer, P., Pereira, R.S., Canhos, D.A.L., Canhos, V.P.: openModeller: a generic approach to species' potential distribution modelling. GeoInformatica **15**(1), 111–135 (2009)
34. Naimi, B., Araújo, M.B.: Sdm: a reproducible and extensible R platform for species distribution modelling. Ecography **39**(4), 368–375 (2016)
35. Kass, J., Anderson, R.P., Aiello-Lammens, M., Muscarella, B., Vilela, B.: Wallace (beta v0.1): Harnessing Digital Biodiversity Data for Predictive Modeling, Fueled by R (2016). http://devpost.com/software/wallace-beta-v0-1-harnessing-digital-biodiversity-data-for-predictive-modeling-fueled-by-r
36. Pennington, D.D., Higgins, D., Peterson, A.T., Jones, M.B., Ludäscher, B., Bowers, S.: Ecological niche modeling using the kepler workflow system. In: Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M. (eds.) Workflows for e-Science, pp. 91–108. Springer, London (2007). https://doi.org/10.1007/978-1-84628-757-2_7
37. Talbert, C., Talbert, M., Morisette, J., Koop, D.: Data management challenges in species distribution modeling. IEEE Bull. Techn. Committee Data Eng. **36**(4), 31–40 (2013)

38. Morisette, J.T., Jarnevich, C.S., Holcombe, T.R., Talbert, C.B., Ignizio, D., Talbert, M.K., Silva, C., Koop, D., Swanson, A., Young, N.E.: VisTrails SAHM: visualization and workflow management for species habitat modeling. Ecography **36**(2), 129–135 (2013)
39. Candela, L., Castelli, D., Coro, G., Pagano, P., Sinibaldi, F.: Species distribution modeling in the cloud. Concurrency Comput. Pract. Exp. **28**(4), 1056–1079 (2016)
40. Candela, L., Castelli, D., Coro, G., Lelii, L., Mangiacrapa, F., Marioli, V., Pagano, P.: An infrastructure-oriented approach for supporting biodiversity research. Ecol. Inf. **26**, 162–172 (2014)
41. Amaral, R., Badia, R.M., Blanquer, I., Braga-Neto, R., Candela, L., Castelli, D., Flann, C., De Giovanni, R., Gray, W.A., Jones, A., Lezzi, D., Pagano, P., Perez-Canhos, V., Quevedo, F., Rafanell, R., Rebello, V., Sousa-Baena, M.S., Torres, E.: Supporting biodiversity studies with the EUBrazilOpenBio hybrid data infrastructure. Concurrency Comput. Pract. Exp. **27**(2), 376–394 (2015)
42. Forzza, R., Mynssen, C., Tamaio, N., Barros, C., Franco, L., Pereira, M.: As coleções do herbário. 200 anos do Jardim Botânico do Rio de Janeiro. Jardim Botânico do Rio de Janeiro, Rio de Janeiro (2008)
43. Mondelli, M.L., Galheigo, M., Medeiros, V., Bastos, B.F., Gomes, A.T.A., Vasconcelos, A.T.R., Gadelha Jr., L.M.R.: Integrating scientific workflows with scientific gateways: a bioinformatics experiment in the brazilian national high-performance computing network. In: X Brazilian e-Science Workshop. Anais do XXXVI Congresso da Sociedade Brasileira de Computação, SBC, pp. 277–284 (2016)
44. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: a language for distributed parallel scripting. Parallel Comput. **37**(9), 633–652 (2011)
45. Gadelha, L.M.R., Wilde, M., Mattoso, M., Foster, I.: Exploring provenance in high performance scientific computing. In: Proceedings of the 1st Annual Workshop on High Performance Computing meets Databases - HPCDB 2011, pp. 17–20. ACM Press (2011)
46. Mondelli, M.L., de Souza, M.T., Ocaña, K., de Vasconcelos, A.T.R., Gadelha Jr., L.M.R.: HPSW-Prof: a provenance-based framework for profiling high performance scientific workflows. In: Proceedings of Satellite Events of the 31st Brazilian Symposium on Databases (SBBD 2016), SBC, pp. 117–122 (2016)
47. Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., Zaharia, M.: Scaling spark in the real world: performance and usability. Proc. VLDB Endowment **8**(12), 1840–1843 (2015)
48. Venkataraman, S., Stoica, I., Zaharia, M., Yang, Z., Liu, D., Liang, E., Falaki, H., Meng, X., Xin, R., Ghodsi, A., Franklin, M.: SparkR: scaling R programs with spark. In: Proceedings of the 2016 International Conference on Management of Data - SIGMOD 2016, 1099–1104. ACM Press, New York, USA (2016)
49. Chamberlain, S.: rgbif: Interface to the Global 'Biodiversity' Information Facility 'API' (2017). R package version 0.9.8. https://CRAN.R-project.org/package=rgbif
50. Liaw, A., Wiener, M.: Classification and regression by randomForest. R News **2**(3), 18–22 (2002)
51. Karatzoglou, A., Smola, A., Hornik, K., Zeileis, A.: kernlab - an S4 package for kernel methods in R. J. Stat. Softw. **11**(9), 1–20 (2004). http://www.jstatsoft.org/v11/i09/



Parallel and Distributed Algorithms

Task Scheduling for Processing Big Graphs in Heterogeneous Commodity Clusters

Alejandro Corbellini(✉), Daniela Godoy, Cristian Mateos, Silvia Schiaffino, and Alejandro Zunino

ISISTAN-CONICET, UNICEN, Tandil, Buenos Aires, Argentina
alejandrocorbellini@gmail.com

Abstract. Large-scale graph processing is a challenging problem since vertices can be arbitrarily connected, reducing locality and easily expanding the solution space. As a result, in recent years, a new breed of distributed frameworks that handle graphs efficiently has emerged. In large clusters with many resources (RAM, CPUs, network connectivity), these frameworks focus on exploiting the available resources as efficiently as possible. However, on situations where the cluster hardware is unbalanced or low in computing resources, the framework must correctly allocate tasks in order to complete execution. In this work, we compare three frameworks, the generic Fork-Join framework adapted to graph processing, and the Pregel and DPM frameworks that were originally designed for computing graphs. A link-prediction algorithm was used as case study to analyze several scheduling strategies that allocate tasks to servers in a cluster of heterogeneous characteristics. The dataset used for the experiments is a snapshot from the Twitter graph, and specifically, a subset of its users that pushed the memory requirements of the algorithm.

1 Introduction

In recent years, processing large graphs in distributed environments has been a major academical and industrial concern. Graphs are structures that are ubiquitous across many fields of study. For example, they have been used to model social networks [1], protein interactions [2], or the structure of the human brain [3]. Mining data from these real-world graphs may uncover interesting information such as the most important vertices in the graph or regions that are highly connected. In the context of Social Networks (SNs), users establish friendship, commercial and/or academic ties that are represented by links in a graph. The resulting *social graph* can help to predict future interactions through link-prediction [4] or to discover relevant sub-graphs using community detection [5] or graph clustering algorithms [6].

In practice, algorithms running on large-scale, real-world graphs can easily surpass the capabilities of a single computer node. For example, a simple algorithm that collects friends of a friend in a social network may reach, in a few traversal steps, large portions of the graph or even the whole graph. A common

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 235–249, 2018.
https://doi.org/10.1007/978-3-319-73353-1_16

alternative to process these complex networks is running the algorithm in a distributed environment.

In consequence, distributed large-scale graph processing frameworks that leverage the capabilities of computer clusters have become an important tool to capitalize the vast information kept in complex networks. In part, this is an effect of the recent proliferation of processing frameworks that run on commodity hardware, which has empowered both large organizations and small teams. In some scenarios, the available resources are scarce, a common situation in small clusters of commodity hardware. Although there are cloud services that offer enormous amounts of processing capabilities, in testing environments that use large-scale datasets the cost of running an experiment multiple times in such a platform may be prohibitively high. Thus, re-using the available cluster infrastructure may lower costs, with the natural performance penalty of using a low-resource infrastructure.

The main purpose of this work is to show through a set of experiments that graph processing frameworks may benefit from using task scheduling to overcome the limitations of a very unbalanced cluster of computers, in terms of hardware capabilities. In particular, we propose two scenarios that limit the available memory of several nodes. One of those scenarios limits the amount of memory of a key component that is usually present in processing frameworks, in order to further stress the running algorithm.

An specific graph algorithm, called Twitter Followee Recommender [7] (TFR), was chosen as a case study due to its large memory usage. TFR is a simple recommendation algorithm that traverses the graph looking for vertices to recommend. On each step, this algorithm gathers vertices by following their edges, and builds a ranking according to how many times those vertices appear in the traversal. In a few steps, the amount of gathered vertices may reach the size of the graph.

This work is organized as follows. Section 2 provides insights on related literature. Section 3 gives some background in the processing frameworks used in the experiments. Section 4 describes the scheduling strategies considered for the experiments. Section 5 presents the algorithm used as case-study, the dataset, the computer cluster, the low-memory scenarios, and the results from running the experiments. Finally, in Sect. 6 we provide some conclusions.

2 Related Work

Graph processing has been supported by ad-hoc tools, which solve an specific graph algorithm, and alternatively frameworks that provide a general-purpose tool to run graph algorithms. Regarding the first graph processing technique, the authors in [8] analyzed the challenges of distributed graph processing, in general, and considered two ad-hoc implementations over different distributed memory architectures. [9] built an ad-hoc implementation of low-rank approximation of graphs and applied it to link-prediction. Unfortunately, without proper knowledge of the implementation details, ad-hoc solutions are usually hard to reuse and maintain.

Several general-purpose frameworks that have been used to process large-scale graphs can be found in the literature. Frameworks such as MapReduce [10], Fork-Join [11] or RDD (Spark) [12] have been applied to various graph-related processing problems [13–15]. Other frameworks, like Pregel [16] or GraphLab [17] were specifically designed for graph-based algorithms [18,19]. Distributed GraphLab [17] bases upon the GraphLab abstraction and extends it to distributed settings. HipG [20] is a graph framework that allows the user to model hierarchical parallel algorithms, which includes divide-and-conquer graph algorithms. Similarly, PBGL [21] and CGMGraph [22] are also open-source graph processing frameworks that allow the user to model graph algorithms. DPM [23] is a graph processing framework that provides a Fork-Join programming style, while preserving some of the benefits of the Pregel or MapReduce frameworks. In the experiments in this paper, we considered Fork-Join, Pregel and DPM for our task scheduling comparison. In order to provide a common ground of comparison, we based our efforts on an in-house developed graph processing framework, called Graphly [24], that included the implementation of the three aforementioned frameworks and provided the needed support for developing scheduling strategies. In particular, Graphly provided a flexible support to extend these frameworks and include task scheduling techniques that dynamically adjust the load balance across the cluster at runtime.

Job scheduling or *mapping* [25] is a well-studied problem in distributed computing. On heterogeneous clusters, i.e., clusters of computers with different capabilities (main memory, CPU speed, number of CPU cores, storage capacity), the allocation of jobs to computing nodes allows distributed and parallel applications to maximize a given performance criterion [26].

3 Graph Processing Frameworks

A distributed processing framework describes the set of software components and the way they must interact in order to execute an algorithm on a given dataset. This type of framework usually has an associated programming model that not only describes the way user programs must be written, but also impacts the way the components are designed for supporting such model. In the next sub-sections we describe the frameworks compared in the experiments: Fork-Join, Pregel and DPM. All their components and their programming models were implemented in the context of a more comprehensive framework, called Graphly [24].

3.1 Fork-Join for Graphs

The Fork-Join processing model uses the concept of jobs and tasks to implement a generic distributed framework. Under this framework a job may spawn many tasks, distribute them among available workers (i.e. a process that performs the tasks) and then the jobs waits for the result of the work carried out by the tasks. Adapting Fork-Join to graph processing is very simple. Figure 1 shows the Fork-Join workflow in Graphly. In the fork stage tasks are delivered

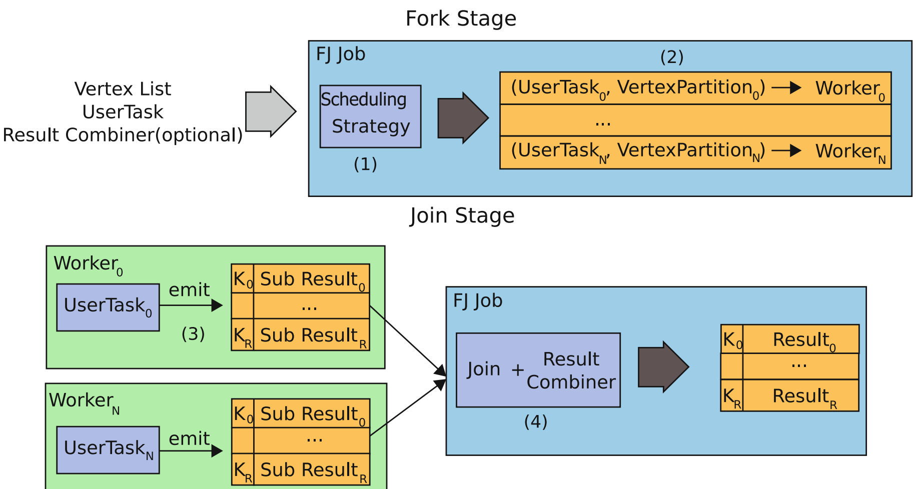


Fig. 1. The Fork-Join workflow in graphly.

to workers and in the join stage results are joined (by the parent job). A Job component executes and manages the results of the specified Tasks. Firstly, input vertices are partitioned using a Scheduling Strategy resulting in a list of $(worker, partition)$ pairs, as shown in (1). Secondly, a user-defined Task is created for each $(worker, partition)$ pair, as shown in (2) in the same figure. Each $\{K_0, K_1, \ldots, K_R\}$ emitted by the workers represent the intermediate results and have an associated value (3). Finally, as Tasks complete execution and return results, the join function merges sub-results (4).

One of the main advantages of Fork-Join is the simplicity of its fork-join programming model. As a consequence, Fork-Join is the natural choice for graph traversals. For iterative algorithms, however, Fork-Join suffers from a large network transfer overhead in comparison to other models. Fork-Join sends the data to be processed to all workers on every iteration. This back and forth of data happens on each iteration. Moreover, the bottleneck at the join stage affects each iteration, making the whole process more time-consuming than other alternatives.

3.2 Pregel

At its core, Pregel uses a logical synchronization barrier to establish the iteration boundaries and prevent access to inconsistent to results, a technique based on the well-known BSP (Bulk-Synchronous Parallel) [27] framework. In Graphly, Pregel is implemented through three main modules as shown in Fig. 2: a Pregel Client (1), a Coordinator (2) and a Pregel Worker (3). The Coordinator's main responsibility is to receive the vertex function (i.e. the user's algorithm) and a list of vertices, set up workers and manage their execution. Initially, the coordinator sends an activation message to all vertices involved in the first superstep

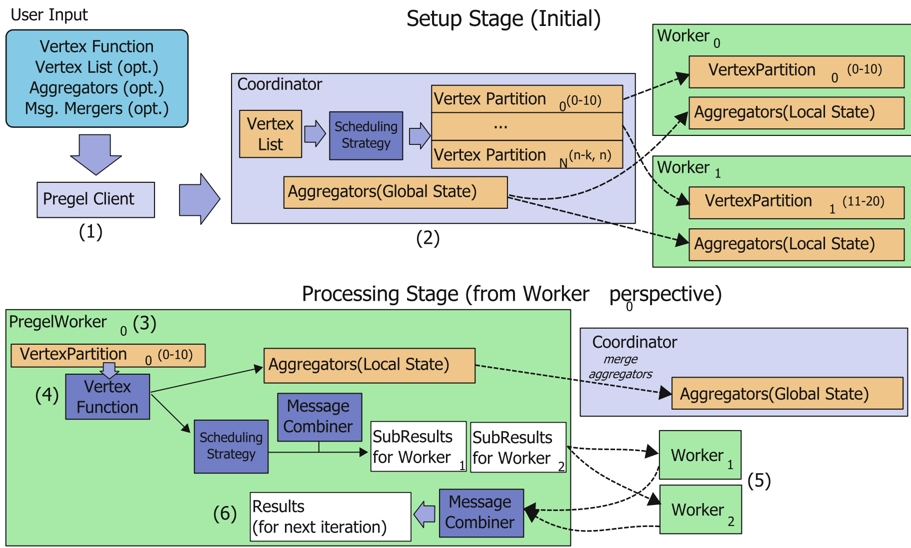


Fig. 2. The Pregel workflow in graphly.

of computation. A vertex is considered active if it has received one or more messages. Afterwards, the coordinator enters in a loop that executes until all vertices finish execution. A Pregel Worker follows a simple workflow: execute active vertices and send messages. On setup, Workers receive the initial set of vertices that must be activated, which involves putting a initial message on each vertex message queue. Then the Worker applies the user-defined vertex function to all active vertices (4). Finally, all messages generated by the execution are sent to their corresponding workers (5). Pregel Workers may receive messages from other workers at any time and must store them for the next superstep. A combiner may be applied to merge messages from different workers to reduce memory consumption (6). The user-defined vertex function can announce that it stopped computing a given vertex by voting to halt execution.

One of the major differences with Fork-Join is that Pregel distributes the merging of results across the cluster by using a vertex-to-vertex messaging mechanism. This mechanism distributes network and processing load among available workers. As a consequence, results are stored locally on each worker, instead of using a central fork-join component (the Fork-Join Job). An extra operation might be needed to collect the results from each worker but on many cases this is not necessary as the results stored on each worker are used as input to another computation.

3.3 DPM

The Distributed Partitioned Merge (DPM) framework aims at reducing the performance bottleneck observed in the Fork-Join framework while still keeping a

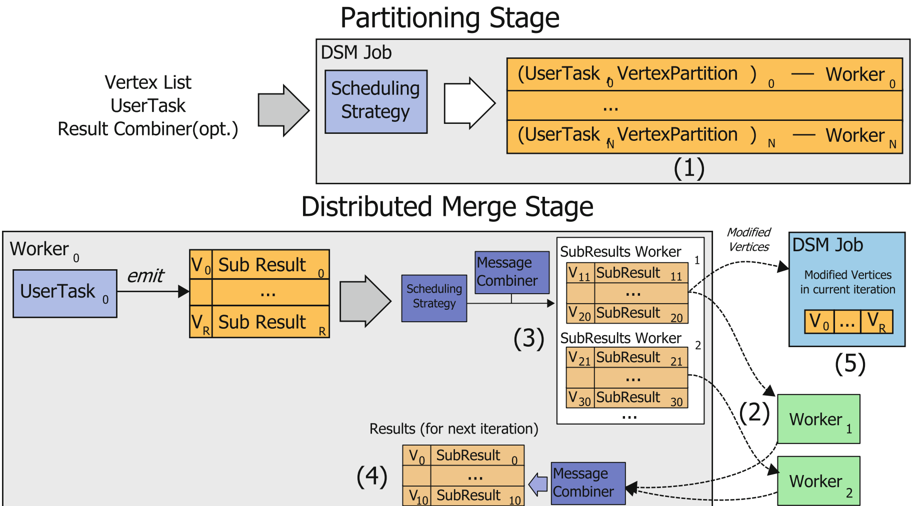


Fig. 3. Distributed partitioned merge model: Overview

fork-join programming style. In opposition to Pregel, the management of the active vertices is centralized, which produces a bottleneck as DPM needs to join the set of active vertices into one final list. Centralizing computation is often harmful in distributed systems, however, merging a set of vertex identifiers is cheap and reduces the complexity of the worker implementation. Additionally, the amount of vertices in real-world graphs is often exponentially smaller than the amount of edges [28]. Due to the fact that most graph algorithms are edge-based, the penalization from managing active vertices in a single location is often negligible in comparison to edge processing.

Figure 3 shows an example of a DPM computation step. In this simple setup, three workers (labeled $Worker_1$, $Worker_2$ and $Worker_3$) process a user-defined algorithm over a graph G of 30 vertices. The first stage of DPM is the Partitioning Stage (1), in which the DPM Job (the master component that controls the execution) splits the input vertex list $V = \{v \mid v \epsilon G\}$ into a set of partitions (from P_1 to P_3) that are assigned to each Worker, along with a copy of the user-defined algorithm. In this example, $Worker_1$ stores (in its local node) vertices from V_1 to V_{10}, $Worker_2$ from V_{11} to V_{20}, and so on. After this stage, each worker may execute the user task on its assigned partition, beginning the Distributed Merge Stage. The $(vertex, subresult)$ pairs emitted from the user-defined task are combined into a map of vertices and results (2). In this example, $Worker_1$ generates results for all vertices, from 1 to 30. Once the execution is completed, the current sub-results are partitioned (3) and sent to their corresponding Workers (4). Concurrently, the list of vertices that were part of the output, are sent to the DPM Job (5) so that they can be processed in the next step. At any given time, other workers may finish their execution and send their sub-results to other workers. In this example, $Worker_1$ receives results from $Worker_2$ and $Worker_3$, and combines them with its local sub-results table (6).

4 Scheduling Strategies

Each of the aforementioned framework implementations use Scheduling Strategies [24] to transform a list of vertices into a set of partitions grouped by worker. Using this mechanism, the user may configure each framework to distribute vertices according to the workers' nodes capabilities to achieve lower algorithm execution times, balancing vertex assignment or reducing resource usage. These strategies work as a configurable and transparent connection between the graph algorithm and the underlying infrastructure.

In particular, the cluster used in this work consisted of a set of nodes with different memory capacities. Based on this setup, the following scheduling strategies were used:

- Location Aware: This strategy takes advantage of the locality of the graph vertices. It divides the input into different lists of vertices grouped by their storage location. For example, let worker $w1$ be responsible for vertices $a1, a3, a5$ and worker $w2$ for vertices $a2, a4$. If a Location Aware strategy is used to map vertices $a1, a2, a5$, it will divide the original list into: $a1, a5 \rightarrow w1$ and $a2 \rightarrow w2$.
- Round Robin: This strategy simply divides the given list of vertices by the amount of workers available and it assigns a sublist to each node. This division of partitions among nodes makes this strategy the most fair in terms of computing load. However, it does not consider either the locality of the data nor the node's characteristics such as available memory, CPU or physical network speed.
- Available Memory: This strategy obtains the memory currently available on each worker and then divides the given list of vertices accordingly. Naturally, this strategy is dynamic, i.e., it adapts to the current status of the cluster.
- Maximum Memory: Similarly, to the Available Memory strategy, the Total Memory strategy considers the maximum amount of memory that a worker can use to divide the list of vertices. It is a fixed strategy that assigns more requests to workers that have more memory installed.

The classic Round Robin strategy was used as a baseline strategy. The Location Aware strategy is the most common technique used in processing frameworks to maintain data locality. Both memory-based strategies were proposed in [24] to tackle algorithms with large memory profiles. However, the Location Aware strategy is, in most cases, the fastest alternative. In this work, we used two low-memory scenarios to show that, in these configurations, the memory-based strategies may be the only alternative to process a memory-intensive algorithm on a large graph.

5 Experiments

As mentioned above, the experiments consisted on running the TFR link-prediction algorithm [7], on a cluster with unbalanced memory capabilities over a large dataset obtained from the Twitter graph. In this Section, we describe each part of the experiments in more detail.

5.1 Twitter Followee Recommender

The Twitter Followee Recommender [7] (TFR) algorithm is a link-prediction algorithm that performs an exploratory traversal of the graph surrounding a target user to select a set of followees to recommend. This algorithm can be represented by the equation $S = ((AA') \bullet T) A) \bullet T$, where $T = (1 - A - I)$, A is the adjacency matrix of the graph, and S is the similarity matrix. From its matrix form, it can be seen that the algorithm performs a series of traversals: first, it performs a traversal in the *out* direction (i.e. AA'), then another in the *in* direction and, finally, a final traversal in the *out* direction. On each step, direct followees of each user are removed from the sub-results (by multiplying element-by-element the matrix T). Figure 4 shows a simple example of a traversal performed by TFR. The starting vertices (users that are already friends of the target) are filtered from the second and third-level set. Second-level vertices are used to find the result vertices. Is important to note that second-level vertices can be part of the result set. It is important to note that some edges are never used in the final step because they always point to the initial set. TFR performs almost no result pruning and, as a consequence, is very memory-intensive, which makes it a good candidate for testing memory-based scheduling strategies.

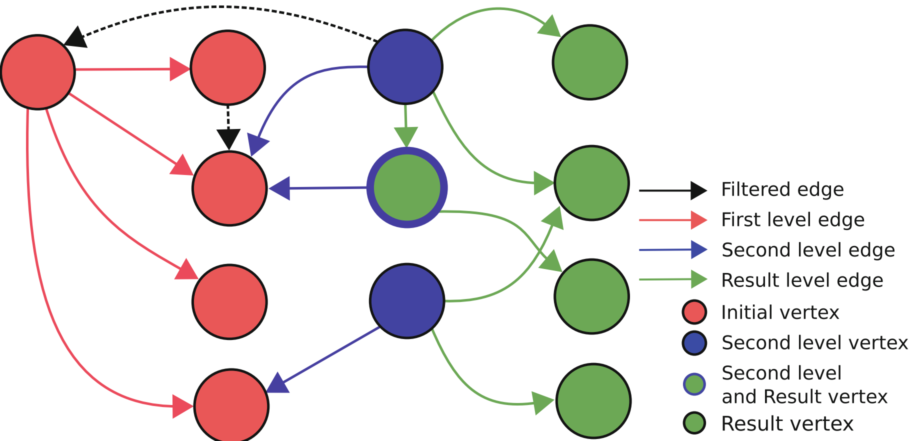


Fig. 4. TFR traversal example.

5.2 Dataset

Experiments were carried out using a Twitter dataset[1] containing the complete follower-following network as of July 2009, presented in [28]. Since Twitter data are no longer available to researchers, this remains as the largest snapshot of Twitter accessible [29]. The dataset contains approximately 1,400 million relationships between more than 41 million users, but it does not include tweets.

[1] Twitter 2010 dataset, http://an.kaist.ac.kr/traces/WWW2010.html.

Table 1. The "Followees Set" used in the experiments along with the size of the graph explored by the TFR algorithm (in millions of vertices)

	User	Followees	Followers	Size
Fee_1	@jimmyeatworld	134788	920556	35
Fee_2	@Starbucks	138045	271215	37
Fee_3	@GuyKawasaki	140903	157878	37
Fee_4	@charitywater	143408	705663	39
Fee_5	@threadless	263317	933726	38
Fee_6	@BJMendelson	283435	937627	36
Fee_7	@TheOnion	369569	1380160	38
Fee_8	@tonyhsieh	407705	1075935	36
Fee_9	@WholeFoods	498700	1112628	38
Fee_{10}	@Number10gov	505613	1105469	37

This is not a limitation since the implemented TFR algorithm is based on user relationships only. The authors in [28] conducted a study on the entire crawled graph, observing some notable properties of Twitter, such as a broken-power-law follower distribution, a short effective diameter, and low reciprocity of relationships, marking a deviation from known characteristics of human social networks [30]. Most relationships in this graph are not reciprocal: only 22.1% of edges are reciprocal (i.e. users that follow each other) and 67.6% of users are not followed by any of its followees.

To test our low-memory scenarios, a group of users was selected out of the complete dataset with the goal of stressing the cluster with high processing load. From the analysis of the dataset and the way the TFR algorithm behaves, we selected a "Followees Set", numbered from $Fee1$ to $Fee10$, that contains the top-10 users ordered by amount of followees. Since TFR starts its traversal in the outgoing direction (i.e. the user's followees), this group of users generates the largest exploration of the graph and, as a consequence, it produces the highest resource usage. The number of followees and followers of this Followees Set, can be seen in Table 1. The graph explored by TFR for each followee is almost the size of the entire dataset, as it is shown in the last column.

5.3 Scenarios

To test our approach of task scheduling using graph processing frameworks, we proposed two very unbalanced scenarios in terms of the available RAM memory of the nodes that are part of the cluster. We hypothesize that some scheduling strategies will help the algorithm run until completion in these harsh conditions, while other strategies will have difficulties in completing execution.

The cluster used in both scenarios was a heterogeneous cluster of 8 nodes, that can be further divided in three sets of machines (A, B and C). Table 2

Table 2. Cluster hardware characteristics.

	A: 3 nodes	B: 3 nodes	C: 2 nodes
CPU	AMD PII X6 1055T 2.8 Ghz	AMD FX 6100 3.3 Ghz	AMD FX 6300 3.5 Ghz
# of cores	6	6	6
RAM	8 GB	16 GB	16 GB
Physical network	1 Gbit Ethernet	1 Gbit Ethernet	1 Gbit Ethernet
Hard disk	500 GB – 7200 RPM	500 GB – 7200 RPM	500 GB – 7200 RPM

summarizes the most relevant characteristics of each set of nodes. The graph data (in this context, the Twitter dataset) is equally distributed across all nodes, as the storage capabilities of the nodes are the same.

As an example of the memory requirements of TFR, we performed a set of experiments over the user *@BarackObama*, a large vertex in the Twitter network, using the Location-Aware strategy. For this user, the algorithm gathers more than 37M users at the final traversal stage. The memory profile of the algorithm depends on the processing model used. The Pregel model uses between 2 GB and 4 GB of memory on each node. Both DPM and FJ exhibit larger memory requirements for the node where the merging of results is performed. DPM requirements on the last stage of TFR vary between 2 GB and 3 GB of memory, but requires 4, 5 GB for the node where the lists of vertices are merged. The FJ model uses between 2 GB and 4 GB of memory, but requires almost 5 GB for the node where the subresults are joined together. Taking into account these memory usage profiles, we propose the following scenarios.

First Scenario. The first scenario has the following setup: half of the machines were configured to use up to 2 GB of RAM and the rest up to 16 GB of RAM. This means that half of the cluster may not be able to cope with the computing requests generated by TFR. In this scenario, Fork-Join and DPM were configured to perform the joining of results in a 16 GB node (i.e. the node where the Fork-Join Job and the DPM Job runs).

Second Scenario. This is a variation of the first scenario. It consists in performing the merging of results, i.e. for the Fork-Join and DPM models, on a relatively low-memory node. Thus, one node was configured to use up to 4 GB of RAM, 4 nodes up to 2 GB and the rest up to 16 GB. The rationale behind this scenario is that the selected "joining node" will use increasingly more memory on each stage of the TFR algorithm, as more and more results are gathered.

5.4 Results

As shown in the results in Fig. 5 most round robin and location-aware executions failed. The only model that provided a solution on some users was the DPM model. As expected, the maximum memory strategy performed poorly but it did not fail on any of the executions. The available memory alternative

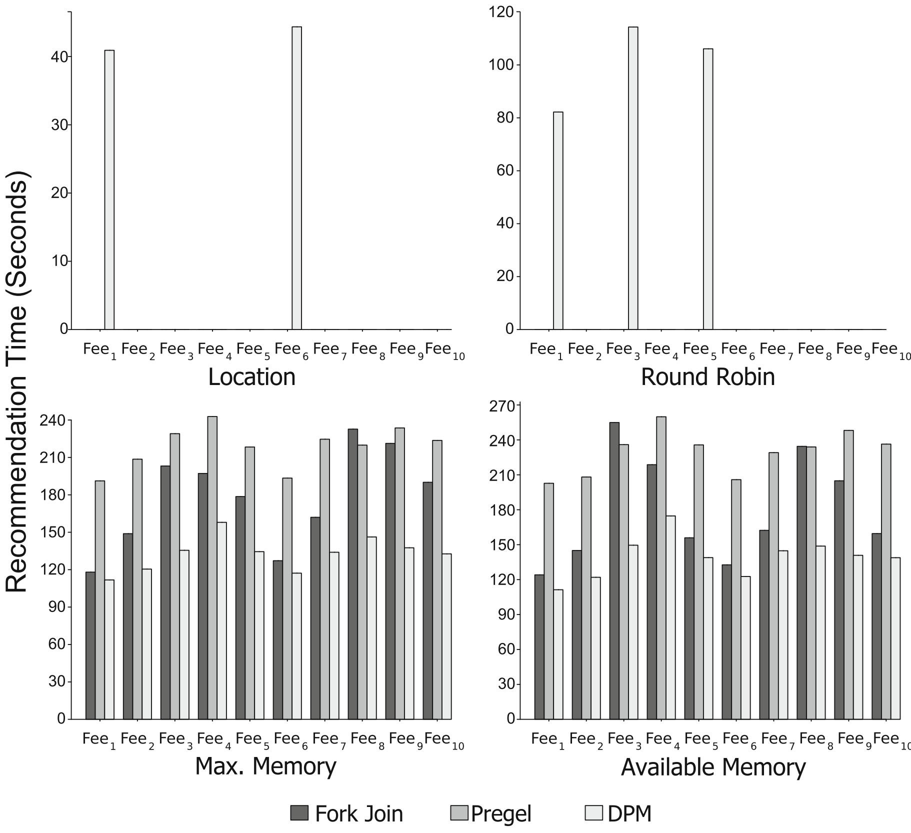


Fig. 5. TFR results using different scheduling strategies for scenario 1.

provided almost the same performance results. This scenario was very important to show that a balanced assignment of work is not always possible. Particularly, the Location Aware and Round Robin strategies suffered from out-of-memory errors and failed to provide a recommendation. However, memory-based scenarios helped to reallocate vertices that cannot be processed due to memory limitations.

Figure 6 shows the results of running TFR under the second scenario. In this scenario, both Round Robin and Location Aware strategies fail to provide results, except for two DPM executions. In fact, all balanced strategies failed, including the Maximum Memory strategy for the Fork-Join framework, which worked in the previous scenario. Restricting the joining node to 4 GB of RAM harmed Fork-Join since it needs large amounts of RAM on a central node. In consequence, many of the Fork-Join tests failed for non-dynamic strategies, such as the Maximum Memory strategy. However, the dynamic nature of the Available Memory strategy helped to avoid the out of memory errors by reallocating vertices that were once allocated to the joining node. Note that DPM and Pregel

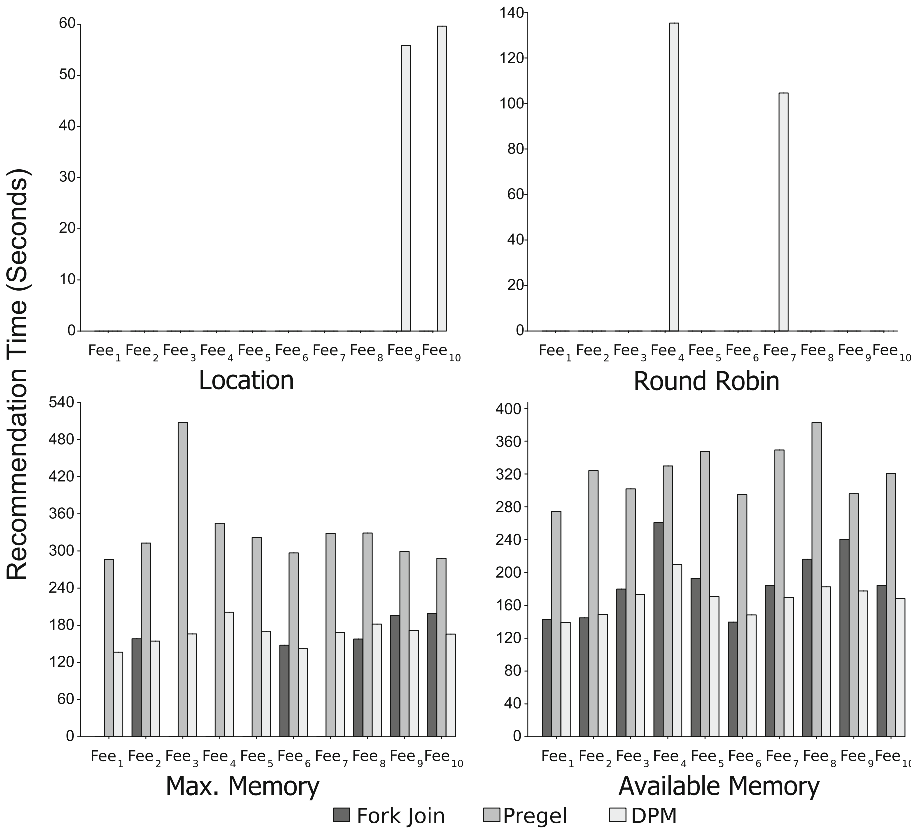


Fig. 6. TFR results using different scheduling strategies for scenario 2.

are unaffected by this situation because merging is distributed throughout the cluster. Finally, as expected, the available memory strategy helps to finish the TFR tests for all users in the Followees set.

In summary, the Location-Aware strategy fails in these scenarios due to its inability to assign processing requests based on memory availability. Round-Robin, as expected, also fails to execute the algorithm on most cases. On the other hand, memory-based strategies assign requests not only using the locality of the vertices, but also take into account memory capabilities of the nodes, and thus, provide results on most executions.

6 Conclusions

Scheduling strategies account for the customized assignment of graph tasks to nodes and their main objective of these strategies is adjusting the partition assignment according to the cluster characteristics. In this work, two scenarios

were studied to test a set of scheduling strategies designed for low-memory scenarios. Besides, for each scheduling strategy we evaluated their impact on different distributed frameworks. Naturally, the objective was to rebalance the vertex computation to nodes with more memory, instead of providing an equal assignment of partitions. On both scenarios for all frameworks, the location-aware and round robin strategies failed to deliver a result for most test users, due to out-of-memory errors. This was an expected behavior, because the location-aware strategy uses the graph storage layout (i.e. where the vertex data is stored in the cluster) to assign tasks and the round robin strategy assigns tasks equally across workers disregarding any node characteristic.

Both memory-based strategies succeeded in providing a recommendation for most frameworks, although with a sustainable time penalty. This proved that on low-memory scenarios it is critical to use a mechanism to rebalance the execution of vertices. As shown in the experiments most of the Fork-Join executions in the second scenario failed, except when the Available Memory strategy was used. Thus, this type of framework, which merge results on a central node, may benefit with a dynamic strategy.

References

1. Wang, P., Xu, B., Wu, Y., Zhou, X.: Link prediction in social networks: the state-of-the-art. Sci. China Inf. Sci. **58**(1), 1–38 (2015). http://arxiv.org/abs/1411.5118
2. Mallek, S., Boukhris, I., Elouedi, Z.: Community detection for graph-based similarity: application to protein binding pockets classification. Pattern Recognit. Lett. **62**, 49–54 (2015). http://www.sciencedirect.com/science/article/pii/S0167865515001488
3. Bullmore, E., Sporns, O.: Complex brain networks: graph theoretical analysis of structural and functional systems. Nat. Rev. Neurosci. **10**(3), 186–198 (2009)
4. Lu, L., Zhou, T.: Link prediction in complex networks: a survey. Phys. A **390**(6), 1150–1170 (2011)
5. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3–5), 75–174 (2010). http://www.sciencedirect.com/science/article/pii/S0370157309002841
6. Rausch, K., Ntoutsi, E., Stefanidis, K., Kriegel, H.-P.: Exploring subspace clustering for recommendations. In: Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM 2014), pp. 42:1–42:4, Aalborg, Denmark (2014)
7. Armentano, M., Godoy, D., Amandi, A.: Towards a followee recommender system for information seeking users in Twitter. In: Proceedings of the International Workshop on Semantic Adaptive Social Web (SASWeb 2011), ser. CEUR Workshop Proceedings, vol. 730, Girona, Spain (2011)
8. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in parallel graph processing. Parallel Proces. Lett. **17**(1), 5–20 (2007)
9. Sui, X., Lee, T.-H., Whang, J.J., Savas, B., Jain, S., Pingali, K., Dhillon, I.: Parallel clustered low-rank approximation of graphs and its application to link prediction. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 76–95. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_6
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

11. Mateos, C., Zunino, A., Campo, M.: An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. Comput. Lang. Syst. Struct. **36**(3), 288–315 (2010)
12. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: a resilient distributed graph system on spark. In: Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems (GRADES 2013), New York, USA, pp. 2:1–2:6 (2013)
13. Cao, L., Cho, B., Kim, H.D., Li, Z., Tsai, M.-H., Gupta, I.: Delta-SimRank computing on MapReduce. In: Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine 2012). ACM, Beijing, China, pp. 28–35 (2012). http://doi.acm.org/10.1145/2351316.2351321
14. Lu, H., Halappanavar, M., Kalyanaraman, A.: Parallel heuristics for scalable community detection. Parallel Comput. **47**, 19–37 (2015)
15. Buzun, N., Korshunov, A., Avanesov, V., Filonenko, I., Kozlov, I., Turdakov, D., Kim, H.: EgoLP: fast and distributed community detection in billion-node social networks. In: 2014 IEEE International Conference on Data Mining Workshop, pp. 533–540 (2014)
16. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 International Conference on Management of Data (SIGMOD 2010), Indianapolis, USA, pp. 135–146 (2010)
17. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endowment **5**(8), 716–727 (2012)
18. Han, M., Daudjee, K., Ammar, K., Özsu, M.T., Wang, X., Jin, T.: An experimental comparison of pregel-like graph processing systems. Proc. VLDB Endowment **7**(12), 1047–1058 (2014)
19. Heitmann, B.: An open framework for multi-source, cross-domain personalisation with semantic interest graphs. In: Proceedings of the Sixth ACM Conference on Recommender Systems - RecSys 2012, p. 313 (2012)
20. Krepska, E., Kielmann, T., Fokkink, W., Bal, H.: HipG: parallel processing of large-scale graphs. ACM SIGOPS Oper. Syst. Rev. **45**(2), 3–13 (2011)
21. Gregor, D., Lumsdaine, A.: The parallel BGL: a generic library for distributed graph computations. Parallel Object-Oriented Sci. Comput. (POOSC) (2005)
22. Chan, A., Dehne, F.: CGM*graph*/CGM*lib*: implementing and testing CGM graph algorithms on PC clusters. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 117–125. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39924-7_20
23. Corbellini, A., Godoy, D., Mateos, C., Schiaffino, S., Zunino, A.: DPM: a novel distributed large-scale social graph processing framework for link prediction algorithms. Future Generation Computer Systems (2017). http://www.sciencedirect.com/science/article/pii/S0167739X17302352
24. Corbellini, A., Mateos, C., Godoy, D., Zunino, A., Schiaffino, S.: An architecture and platform for developing distributed recommendation algorithms on large-scale social networks. J. Inf. Sci. **41**(5), 686–704 (2015). http://jis.sagepub.com/content/early/2015/06/06/0165551515588669.abstract
25. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002)

26. Kim, J.-K., Shivle, S., Siegel, H.J., Maciejewski, A.A., Braun, T.D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R.B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., Yellampalli, S.S.: Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France (2003)
27. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
28. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010), Raleigh, NC, USA, pp. 591–600 (2010)
29. Faralli, S., Stilo, G., Velardi, P.: Large scale homophily analysis in Twitter using a twixonomy. In: Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI 2015). AAAI Press, Buenos Aires, Argentina, pp. 2334–2340 (2015)
30. Newman, M.E., Park, J.: Why social networks are different from other types of networks. Phys. Rev. E **68**(3), 036122 (2003)

Exploring Application-Level Message-Logging in Scalable HPC Programs

Esteban Meneses(✉)

National Advanced Computing Collaboratory,
National High Technology Center and School of Computing,
Costa Rica Institute of Technology, Cartago, Costa Rica
esteban.meneses@acm.org

Abstract. The next generation of supercomputers will require HPC applications to handle failures. This paper presents, through an example application, the benefits of logging messages at the application level. The proposed method will do both, provide resilience to failures and improve performance.

Keywords: Resilience · Fault tolerance · Message logging

1 Resilience in HPC Applications

Resilience is one of the most pressing challenges the High Performance Computing (HPC) community faces for the next generation of machines [6]. The vast amount of computing components assembled into a single supercomputer at exascale makes the failure rate worryingly high. It will be crucial for HPC applications to incorporate some sort of fault-tolerance mechanism. Various resilience strategies have been devised. One family of techniques is referred to as *algorithm-based fault tolerance* (ABFT) [3], in which the original algorithm is modified to also run encoded computations that will be later used to detect and correct failures. Other, more traditional, family of techniques is known as *rollback-recovery* [1], in which the system periodically saves the state of the computation. Should a failure happen, the system rolls back to the latest saved state. The basic implementation of this idea is the well-known checkpoint/restart mechanism, where minimal changes to the application's algorithm are required. There are several extensions to checkpoint/restart. One such extension is message logging, which stores just the necessary communication to only restart the failed component, saving time and energy [5].

This paper explores the potential of a hybrid resilience strategy that combines both ABFT and message-logging philosophies. The strategy is called *application-level message-logging* and it is based on the fundamental principles of rollback-recovery with message-logging. However, it requires some intervention to the application's algorithm to store some communication at the application level. The method should tolerate the crash of one *processing entity* (PE). In this paper, a PE may refer to a core, a processor, or a node. The distinctive feature of a PE is that it runs an operating system heavyweight process.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 250–254, 2018.
https://doi.org/10.1007/978-3-319-73353-1_17

2 Application-Level Message-Logging

We will introduce our proposed method through an application called ChaNGa [2], written in the Charm++ object-oriented parallel programming language [4]. ChaNGa (Charm++ N-body GrAvity solver) is a program that computes gravitational forces among a set of particles. It performs collisionless N-body simulations. The major application of ChaNGa is to cosmological simulations with periodic boundary conditions in comoving coordinates. It has also been used for simulations of isolated stellar systems. Additionally, it can include hydrodynamics using the Smooth Particle Hydrodynamics (SPH) technique. ChaNGa relies on a Barnes-Hut tree to calculate gravity, with hexadecapole expansion of nodes and Ewald summation for periodic forces. Timestepping is achieved with a leapfrog integrator with individual timesteps for each particle.

There are four major stages in each step of ChaNGa: (*i*) domain decomposition, which creates a division of the space according to the set of particles and their positions, (*ii*) load balancing, which migrates particles to balance the load across the set of processors, (*iii*) tree building, where a Barnes-Hut tree is rebuilt for the new set of particle positions and their distribution, and (*iv*) gravity computation, which performs the actual computation on each particle.

It is the gravity computation step we will focus on. After the tree building, a set of objects called Tree Pieces will hold each a subset of the particles. During the gravity computation, each Tree Piece is responsible for computing the total gravity on its particles. In doing so, it may require the particle information from potentially many other Tree Pieces. Several Tree Pieces will reside on the same PE. To avoid multiple identical requests, ChaNGa has a receiver-side Cache object. This set of objects behave as a *group* in Charm++, meaning there is exactly one object per PE. Thus, each Tree Piece will try contacting its local Cache first to get the particles it needs. If the Cache has the particular set of particles stored, it will return them to the requesting Tree Piece. Otherwise, the Cache will contact the remote Tree Piece, cache the content of the message and forward it to the requesting Tree Piece. Figure 1 shows a diagram of the overall scheme of objects in ChaNGa where Tree Piece A requires the information from Tree Piece B and that request always goes through the local Cache on PE X.

The current implementation of ChaNGa provides (accidentally) a receiver-based message-logging mechanism [1]. However, failure of a PE will immediately make the Cache object disappear, along with all its content. Additionally, messages are cached per iteration. Therefore, not all messages would be available for recovering a Tree Piece, even if the Cache survives the crash. What is encouraging about this scenario is the relative advantage of logging the messages, at least in terms of performance. Most of the traditional message-logging techniques belong to the sender-based family, where messages are stored in the main memory of the sender. If ChaNGa were to be seen from that perspective, there would be a fertile land to implement a message-logging protocol for resilience.

To understand the potential of application-based message-logging in ChaNGa, we measured the average number of requests per iteration. We ran

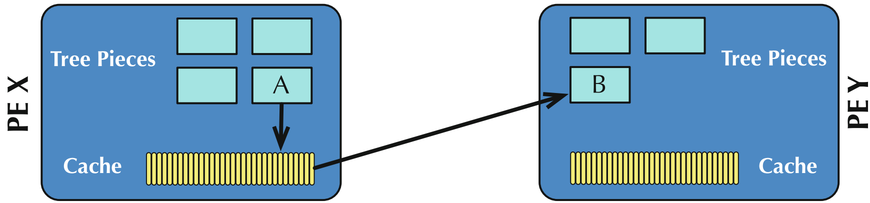
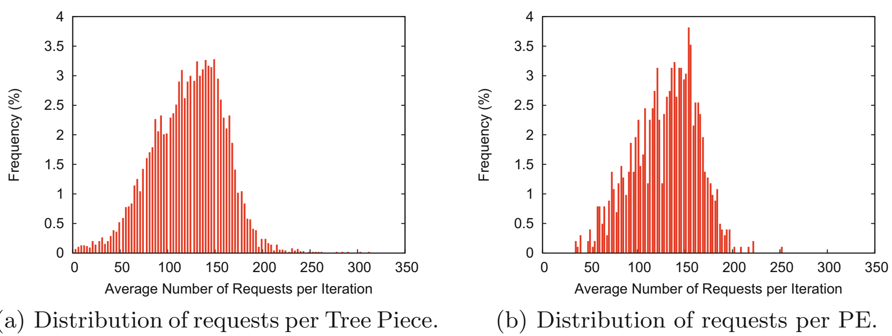


Fig. 1. Simplified diagram of objects in ChaNGa. A Tree Piece that requires the particle information of another Tree Piece will first contact its local Cache to avoid a remote transmission. If the local Cache has the particle message, it will provide the message. Otherwise, the Cache will contact the remote object.

for 10 iterations the `dwf1.2048` dataset with 8,192 Tree Pieces. The dataset contains 5 million particles. The results were gathered on Intrepid supercomputer with 1,024 cores. Figure 2 shows the distribution of requests for ChaNGa. We only counted remote requests, i.e., coming from a remote PE. In Fig. 2(a) appears the distribution of the average number of requests per TreePiece. Although most of the distribution follows a bell shape, the tail is significant with a few Tree Pieces showing a high number of requests per iteration. The story changes a little once PEs are considered. Figure 2(b) offers the distribution of average number of requests per iteration per PE. Both distributions show the immense potential for having a sender-based message-logging at the Cache objects. On average, the number of requests per iteration is higher than 100, meaning per iteration the very same message has to be built at least 100 times.



(a) Distribution of requests per Tree Piece. (b) Distribution of requests per PE.

Fig. 2. Distributions of average number of requests per iteration on ChaNGa.

Using a strong-scale approach, we ran the same experiment from 512 to 4,096 cores. Table 1 summarizes the results of this experiment. As the number of cores increases, the average number of requests raises. This can be explained by the fact that more cores, means a higher chance that the Tree Piece is not local, requiring a remote request. However, the dispersion of the data also changes

drastically. The absolute maximum of requests reports the Tree Piece with the maximum number of requests and it roughly doubles as we go from 512 to 4,096 cores. The averages in the number of requests per iteration are always higher for the Tree Piece version. This is a natural result of load balancing that will balance Tree Pieces with different request profiles on the different PEs.

Table 1. Average number of requests per iteration in ChaNGa.

Number of requests	Number of cores			
	512	1024	2048	4096
Absolute average	97.05	124.50	160.51	202.28
Absolute maximum	508.00	670.00	936.00	1154.00
Average per PE	100.99	131.45	171.66	218.95
Maximum average per PE	165.89	252.90	407.33	521.00
Average per Tree Piece	92.04	118.46	152.87	192.68
Maximum average per Tree Piece	260.00	312.40	473.40	607.30

The lesson extracted from ChaNGa is that applications may have a high potential to eliminate unnecessary creation of messages and at the same time provide a useful infrastructure for message-logging and resilience. In the particular case of ChaNGa, the Cache objects could be made sender-based too. This would avoid the cost of generating the same message multiple times and serve as the fundamental basis for message-logging protocols.

Acknowledgments. This work was partially supported by a machine allocation on Argonne Leadership Computing Facility awarded by the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. Elnozahy, E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
2. Gioachin, F., Sharma, A., Chakravorty, S., Mendes, C., Kalé, L.V., Quinn, T.: Scalable cosmological simulations on parallel machines. In: Daydé, M., Palma, J.M.L.M., Coutinho, Á.L.G.A., Pacitti, E., Lopes, J.C. (eds.) VECPAR 2006. LNCS, vol. 4395, pp. 476–489. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71351-7_37
3. Huang, K.-H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **33**(6), 518–528 (1984)
4. Kalé, L., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: Paepcke, A. (ed.) Proceedings of OOPSLA 1993, pp. 91–108. ACM Press, September 1993

5. Meneses, E., Sarood, O., Kale, L.V.: Energy profile of rollback-recovery strategies in high performance computing. Parallel Comput. **40**(9), 536–547 (2014)
6. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A.A., Coteus, P., DeBardeleben, N., Diniz, P.C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., Hensbergen, E.V.: Addressing failures in exascale computing. IJHPCA **28**(2), 129–173 (2014)

Accelerated Numerical Optimization with Explicit Consideration of Model Constraints

Lucia Damiani[1], Ariel Ivan Diaz[2], Javier Iparraguirre[2(✉)], and Aníbal M. Blanco[1]

[1] PLAPIQUI (CONICET-UNS), Bahía Blanca, Argentina
{ldamiani,ablanco}@plapiqui.edu.ar
[2] UTN-FRBB, Bahía Blanca, Argentina
arielivandiaz@gmail.com, j.iparraguirre@computer.org

Abstract. Population based metaheuristics can benefit from parallelization in order to address complex numerical optimization problems. Typical realistic problems usually involve non-linear functions, integer variables and many constraints, making the identification of optimal solutions mathematically challenging and computationally expensive. In this work, a parallelized version of the Particle Swarm Optimization technique is proposed, whose main contribution is the explicit consideration of constraints. The implementation is tested on a classic set of optimization problems. Speedups up to 101x were obtained using a single GPU on a standard PC using the Py-Cuda technology.

Keywords: Numerical optimization · Particle swarm optimization · GPU

1 Introduction

Numerical optimization has an outstanding role in science and engineering. The objective is to find the set of variables that optimizes a performance index (objective function) while verifying a set of equality and inequality constraints. A particularly challenging version arises when the variables are combined in a non-linear fashion in the involved functions. Additional complexity is introduced if some of the variables are required to be integer. A huge body of literature on numerical optimization exists from many disciplines, which addresses both theoretical and practical aspects. Regarding solution techniques, a broad classification involve deterministic and metaheuristic approaches.

Deterministic optimization relies on information of derivatives to deal with non-linearities. Additionally, branch and cut type algorithms deal with integrality [1]. This class of optimization algorithms is mathematically sophisticated and the most competitive implementations are proprietary products. Although global optimality can be guaranteed under certain conditions, deterministic algorithms are very computationally demanding. A typical case is when the execution runs out of memory or when the process reaches time limits for even medium size instances in terms of variables and equations.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 255–261, 2018.
https://doi.org/10.1007/978-3-319-73353-1_18

In contrast, metaheuristics rely on stochastic explorations of the search space [2, 3]. A particularly important family of metaheuristics is based on the evolution of an initial population of solutions through rather simple rules usually inspired on natural (biological) processes. Genetic Algorithms, Ant Colony Optimization, and Particle Swarm Optimization (PSO) are amongst the most developed techniques. In particular PSO has acquired notable popularity in recent years since it is easy to code and shows good performance in multimodal box-constrained problems such as parameter estimation studies [4]. However, its major weaknesses are related to the handling of constraints (equality and inequality) and integer variables. Moreover, PSO is computationally intensive since a large number of evaluations of the objective function and constraints are usually required to achieve convergence. Due to its inherent parallel nature, PSO can benefit from parallelization strategies to improve speed. In fact, there exist a large number of parallel implementations of PSO on many architectures. Since a thorough review of related work is beyond the scope of this contribution, the interested reader is referred to the recent survey on GPU implementations of swarm metaheuristics by Tan and Ding [5].

In this work a GPU based implementation of the PSO algorithm is presented. The major contribution regarding previous developments is the explicit consideration of a constraint handling methodology, which is unavoidable to address realistic problems arising from engineering applications. Additionally, the implementation is based on the PyCUDA platform, which allows the flexibility of an interpreted language such as Python and the efficiency provided by CUDA technology. After introducing the basics, performance is tested on a set of typical benchmark problems.

The next section briefly introduces the mathematical formulation of the PSO algorithm. Section 3 describes the implementation details of the proposal. Section 4 presents the results. Finally, the conclusions and the future work are presented in Sect. 5.

2 Particle Swarm Optimization

A general formulation of the optimization problem addressed in this work is:

$$\mathrm{Min}_{\mathbf{x}} \mathrm{f}(\mathbf{x}),\ \mathrm{st.}\mathbf{h}(\mathbf{x}) = \mathbf{0},\ \mathbf{g}(\mathbf{x}) \leq \mathbf{0},\ \mathbf{x} \in \mathbf{X} \tag{1}$$

where f(.) is an objective function and $\mathbf{h}(.)$ and $\mathbf{g}(.)$ stand for equality and inequality constraints respectively. Vector $\mathbf{x}$ is the set of optimization variables described by a lower and upper bounds on each element. Mathematically, PSO is described by two simple equations [4]:

$$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \mathbf{v}_i^{k+1} \tag{2}$$

$$\mathbf{v}_i^{k+1} = w^k \mathbf{v}_i^k + c_1 \mathbf{r}_1^k \left(\mathbf{p}_i^k - \mathbf{x}_i^k\right) + c_2 \mathbf{r}_2^k \left(\mathbf{q}^k - \mathbf{x}_i^k\right) \tag{3}$$

Vectors $\mathbf{x}_i$ and $\mathbf{v}_i$ represent position and velocity of particle i respectively. Supra index k denote iteration. Velocity is updated on the basis of a cognitive term which

involves the best position ever reached by each particle along its trajectory ($\mathbf{p}_i^k$) and a social term which considers the best position ever reached by the swarm along its evolution ($\mathbf{q}^k$). Parameters w, c_1 and c_2, are fixed or may vary with the iterations according to some strategy. Vectors $\mathbf{r}_1$ and $\mathbf{r}_2$ have random elements between 0 and 1.

The best positions ($\mathbf{p}_i^k$ and $\mathbf{q}^k$) are related with some measure of the performance of the system, based on the objective function f(.), and constraints **h**(.) and **g**(.). Constraints handling in swarm optimization has received considerable attention [6]. In this work we adopted the approach proposed by Zhang and Rangaiah [7] to deal with constraints. This approach is based on the definition of the Total Absolute Violation of each particle of the swarm (TAV_i) defined as:

$$TAV_i^k = \sum\nolimits_n \left|h_n\left(\mathbf{x}_i^k\right)\right| + \sum\nolimits_m \max\left(0, g_m\left(\mathbf{x}_i^k\right)\right) \quad i = 1, \ldots, N \tag{4}$$

If TAV_i^k is lower than a relaxation value μ, particle i is considered feasible in iteration k, otherwise it is unfeasible. In this method, parameter μ is dynamically reduced according to Eq. (5), where F_F^k represents the number of feasible particles in iteration k.

$$\mu^{k+1} = \mu^k\left(1 - F_F^k/N\right) \tag{5}$$

In each iteration the following criteria is adopted to update the swarm ($\mathbf{p}_i^k$ and $\mathbf{q}^k$): (i) a feasible solution is preferred over an unfeasible solution, (ii) between two feasible solutions, the selected one is that with the better value of objective function f(.), (iii) between two unfeasible solutions, that with the lower TAV is preferred. Although the TAV strategy was originally proposed for "Differential Evolution" algorithms in [7], it is straightforwardly implementable under PSO and any other evolutionary technique. It was preferred over other constraint handling methodologies since it gradually enforces the swarm to the feasible region while ensuring a convenient exploration of the search space.

3 PSO Implementation

The methodology described in the previous section was implemented in Python, making use of the capabilities provided by the NumPy library [8] to produce a competitive serial version of the algorithm. Then, a parallelized version was implemented using the Py-CUDA [9] programming language to exploit the potential of NVIDIA GPUs. The use of Python as a programming language allows a highly productive environment and provides a solid set of numeric libraries, which facilitates the implementation. In a similar way, the use of PyCUDA allows the use of an accelerator without losing the benefits of an interpreted programming language.

The implemented algorithm is described in the pseudo code shown in Algorithm 1. Although all parts of the algorithm admit some level of parallelization, in this work only some were actually accelerated. Specifically steps a, b, c and e of Algorithm 1 were parallelized, while reduction d was left as a serial process in the proposed kernel.

Although reductions are quite standard procedures, the identification of the global best (step d) represents a rather challenging operation to be parallelized for a general swarm. Therefore, its implementation was postponed as future work.

Following the classifications proposed by Tan and Ding [5], the implementation presented in this work can be described as an all-GPU parallel model. Except for the random number initializations, all steps involved in the PSO computation are performed on the GPU. This strategy minimizes the overhead caused by the process of data exchange between the GPU and the CPU.

In terms of resources, one particle is assigned to a CUDA block and the number of threads per block equals the dimension of the problem under study. This decision allows the use of the shared block memory. Only one GPU is involved in the computing process.

```
1. Randomly initialize x_i^0 and v_i^0 for i=1,…,N
2. For k=1,…,kmax repeat
   a. Evaluate x_i^k from Eq.(2)
   b. Evaluate f(x_i) and TAV_i^k from Eqs.(1) and (4)
   c. Identify p_i^k (local best) according to the feasibility criteria
   d. Identify q^k (global best) according to the feasibility criteria
   e. Evaluate v_i^k from Eq.(3)
```

Algorithm 1. Pseudocode for PSO

4 Results

The serial and the parallelized versions of the PSO, were tested with a set of optimization problems in order to investigate speedups. The problems were taken from a compilation of benchmark models typically used to assess the performance of metaheuristic algorithms [10]. Such compilation provides constrained models of different sizes and complexities.

In all cases the experiments were performed using the following parameterization of the algorithm: w = 0.9, c_1 = 2.0, c_2 = 2.0, N = 50 and k_{max} = 2500. It should be noted that many parallelization studies use the swarm size (N) to investigate performance since speedups sensibly increase as this parameter increases. However, it has been shown that PSO performance becomes insensitive to swarms larger than 30–60 particles for most problems [4]. Therefore, N = 50 was adopted for all purposes in this study.

As stated in related work, Speedup was selected to evaluate performance. Equation 6 defines the metric. T_{CPU} represents the time that the CPU version required to achieve a solution and T_{GPU} is the time required by the same problem executed by the accelerated version.

$$\text{Speedup} = T_{CPU}/T_{GPU} \tag{6}$$

In Table 1, a detail of the problems and the obtained speedups are reported. In column “Ref. [10]” the problem number corresponding to report [10] is provided as a reference for the complete formulation and global solution of each problem for the

interested reader. In columns "D", "n", and "m" the number of variables, equality constraints and inequality constraints of each problem are respectively informed. The models reported in this study is a rather small subset of the original collection [10] selected to perform a preliminary investigation of problems of different sizes and types of nonlinearities.

The experiments were carried out in two different systems to investigate performance. System 1 corresponds to a PC Intel Skylake Core i7 6700 3.4 GHz, 8 GB DDR4 RAM Memory, associated with a GTX480 GPU running Linux Mint 18.1. System 2 is equipped with an AMD FX-4100 Quad-Core Processor, 8 GB of DDR3 RAM memory, a GeForce GTX TITAN X GPU, running Linux Mint 18.1. Since PSO contains components that rely on random numbers, multiple independent runs are provided for each of the reported measurements. This decision provides statistical validity of the convergence of the different versions of the algorithm. In particular, column "TCPU" reports the average time of 20 independent runs of the serial PSO version. Column "Speedup" reports the average speedup obtained, also from 20 independent runs, using the parallelized version on the GPU in the corresponding system. In both cases significant speedups were obtained averaging 32x for system 1 and 51x for system 2.

In all problems, the algorithm converged to feasible solutions. Success rate, defined as the percentage of the runs that converged to the known global solution (within a tolerance), was superior to 75% in most cases. Problems were feasible but sub-optimal solutions were achieved in less than 75% are marked with an asterisk in column P#. It should be mentioned that since the focus of this study was on the speedup of the parallelized version of the algorithm, the PSO was not tuned for the different problems and a standard parameterization was adopted in all cases as described above. For this reason, success rate might not look as impressive as in other studies where the emphasis is on the convergence of the solvers to the global optimum.

Table 1. Detail of problems and speedups.

P# Ref. [10]	D	n	m	System 1		System 2	
				T_{CPU} (sec)	Speedup	T_{CPU} (sec)	Speedup
1*	13	0	9	4.76	28.12	13.13	44.30
3	10	1	0	3.27	25.94	8.99	41.86
4	5	0	6	4.24	40.39	12.21	60.30
5	4	3	2	2.93	28.92	8.70	43.34
7*	10	0	8	8.17	57.63	24.37	101.64
8	2	0	2	1.69	17.91	4.88	24.78
9	7	0	4	6.07	50.50	16.10	73.65
11	2	1	0	0.94	10.02	2.68	14.40
13*	5	3	0	2.22	22.58	6.10	32.55
14*	10	3	0	8.00	47.13	22.08	84.12
18	9	0	13	5.44	42.06	14.52	64.20
24	2	0	2	2.02	22.78	5.50	30.22

5 Conclusions and Future Work

A GPU accelerated particle swarm optimization with explicit consideration of constraints was presented. Regarding most previous implementations of swarm metaheuristics on GPUs, which mainly addresses box constrained objective functions, our implementation included the TAV methodology to handle equality and inequality constraints. Although the parallelization potential of the algorithm was not fully exploited, and the PSO can be improved as well, satisfactory performance was obtained, both in terms of speedups and success rate. These results are encouraging to address even more challenging problems and actual engineering applications.

The proposed kernel can be further developed in several ways. Specifically, future work considers the implementation of a parallel reduction to identify the best member of the swarm (step d in pseudo code of Algorithm 1). Also the optimization of the use of the shared memory will be investigated.

The PSO algorithm admits many improvements as well, in order to increase success rate and overall performance. In particular, parameter w can change dynamically with iterations in order to adjust the exploration/exploitation tradeoff of the search space. Alternative termination criteria, other than a maximum number of iterations, can be also implemented. For example, exploration time can be saved if it is detected that no improvement on the objective function occurs in a given number of consecutive iterations.

Current and improved versions will be tested with a larger set of problems and number of independent runs in order to provide results with statistical significance.

Acknowledgments. This research was partially supported by grants from Consejo Nacional de InvestigacionesCientíficas y Técnicas (CONICET) and Universidad Tecnológica Nacional (UTN) of Argentina. The authors also gratefully acknowledge the support of NVIDIA Corporation with the donation of the TITAN X GPU used in this research.

References

1. Tawarmalani, M., Sahinidis, N.: 2002: Convexification and Global Optimization in Continuous and Mixed Integer Programming. Kluwer Academic Publisher, Dordrecht (2002)
2. Boussaïd, I., Lepagnot, J., Siarry, P.: A survey on optimization metaheuristics. Inf. Sci. **237**, 82–117 (2013)
3. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. Int. Trans. Oper. Res. **20**, 1–48 (2013)
4. Marini, F., Walczak, B.: Particle swarm optimization (PSO). A tutorial. Chemom. Intell. Lab. Syst. **149**, 153–165 (2015)
5. Tan, Y., Ding, K.: A survey on GPU-based implementation of swarm intelligence algorithms. IEEE Trans. Cybern. **46**(9), 2028–2041 (2016)
6. Coello, C.A.C.: Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Comput. Meth. Appl. Mech. Eng. **191**, 1245–1287 (2002)

7. Zhang, H., Rangaiah, G.P.: An efficient constraint handling method with integrated differential evolution for numerical and engineering optimization. Comput. Chem. Eng. **37**, 74–88 (2012)
8. NumPy Homepage. http://www.numpy.org/
9. PyCuda reference. https://developer.nvidia.com/pycuda
10. Liang, J.J., Runarsson, T.P., Mezura-Montes, E., Clerc, M., Suganthan, P.N., Coello, C.A.C., Deb, K.: Problem definitions and evaluation criteria for the CEC 2006 special session on constrained real-parameter optimization. Technical report (2006)

Parallel Processing of Intra-cranial Electroencephalogram Readings on Distributed Memory Systems

Leonardo Piñeyro(✉) and Sergio Nesmachnow

Centro de Cálculo, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay
{leonardo.pineyro,sergion}@fing.edu.uy

Abstract. This article presents an approach for parallel processing of electroencephalogram readings over distributed memory systems. This is a complex problem that deals with a significantly large amount of data, especially considering that the volume of electroencephalogram readings has been growing for the last few years due to their handling in medical and health applications. Different parallelization and workload distribution techniques applied to processing intra-cranial electroencephalogram readings are studied, in order to efficiently detect whether a patient may suffer a seizure or not. More precisely, two separate approaches are presented: a first one describing a traditional Message Passing Interface implementation for cluster systems, and a second implementation using Apache Hadoop, more adapted to large-scale processing in cloud systems. The experimental evaluation performed on standard datasets demonstrates that it is possible to remarkably speedup electroencephalogram processing by applying efficient data distribution strategies. The parallel/distributed approach allows accelerating the execution time up to **22**× when compared with the sequential version.

Keywords: Feature extraction · Distributed computing
Large-scale processing

1 Introduction

Epilepsy afflicts nearly 1% of the world's population and is characterized by the occurrence of spontaneous seizures. For many patients, anticonvulsant medicine can be given at sufficiently high doses to prevent seizures, but patients frequently suffer side effects. For 20–40% of patients with epilepsy, medications are not effective. Despite the fact that seizures occur infrequently, patients with epilepsy experience persistent anxiety due to the possibility of a seizure occurring [1,2].

Intra-cranial electroencephalography (iEEG) is a type of monitoring technique that reads the brain activity of a patient by placing electrodes directly onto the brain surface. Most patients with severe epilepsy are treated with surgeries which involve the use of iEEG. Fortunately, the technology improvements in

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 262–276, 2018.
https://doi.org/10.1007/978-3-319-73353-1_19

this area make possible collecting high-quality long-term electrical brain activity from human patients, spanning from months to even years [3].

With the intention of giving the patients a faster medical attention and a better quality of life, researchers have proposed developing software packages that are capable of predicting, with the highest possible precision, the occurrence of seizure events. Most of the times it is hard for experts to find clues by just looking at the raw data of the patients. Automated large-scale analysis of iEEG data provides useful information to understand brain processes, as processing a large number of data allows obtaining more accurate and robust brain models. Real-world brain imaging using sophisticated sets of transformations (most of which belong to the signal processing area) helps experts to find more valuable and reliable information out of the raw iEEG readings. It is a fact that giving this information, instead of the raw iEEG data, to a software capable of detecting patterns and classify different kinds of brain activity increases significantly the accuracy of the detection process [4].

High Performance Computing (HPC) [5] is a paradigm that helps researchers to solve complex data processing problems such as the feature extraction problem related to iEEG data processing. Distributed and parallel programming techniques play a key role to significantly reduce the execution times of data processing algorithms. In the case of iEEG processing, the problem involves working with huge datasets of iEEG readings (likely to grow in size to several TeraBytes) and synchronizing different tasks applied to different sections of the measurements.

This article presents two different approaches for dealing with the iEEG readings processing problem using distributed memory HPC strategies: (i) a traditional implementation developed using a message passing interface library, capable of executing in cluster systems, and (ii) an implementation using the Apache Hadoop platform, taking advantage of the Map-Reduce programming paradigm to work with large-scale datasets, to be executed on cloud-based infrastructures.

The experimental evaluation on the proposed implementations is performed on real distributed memory HPC platforms and considering benchmark instances of the iEEG data processing problem. Efficiency results demonstrate that the execution time of the iEEG data processing algorithms can be remarkably reduced when compared to the sequential versions. The distributed versions achieved an acceleration of up to **22**× for the message passing implementation and 9.16× for the Hadoop implementation.

The article is organized as follows. Section 2 presents a review of related works describing different approaches for processing data from iEEG readings. Section 3 introduces the problem and the proposed approach for iEEG data processing. The specific methods proposed for applying distributed memory HPC techniques to the iEEG processing problem are described in Sect. 4. Section 5 reports the experimental evaluation of the proposed methods over a specific dataset of iEEG measurements. Finally, Sect. 6 presents the conclusions and the main lines for future work.

2 Related Work

This section reviews articles that describe different processing methods for iEEG readings.

2.1 Historic Review

Seizure prediction studies have grown significantly since the first experiments done in the 1970 decade. Studies largely improved over the 90's and the beginning of the millennium. Nevertheless, many of these studies were not sophisticated enough and had irreproducible results. Thus, solutions were not better than a random guessing algorithm when working with real-world brain activity [1].

Recent studies have proven that better results can be achieved, with up to 83% of accuracy in some scenarios. Although these results seem to be promising, they are still not good enough for real-life medical applications [1,6].

The application of signal-processing techniques has been of great importance on this field. These methods are used to extract the most valuable information out of the iEEG by performing signal-processing operations such as Fourier transformations, different kind of correlations, statical analysis like kurtosis, and skewness, among others [7].

2.2 Distributed Approaches to Process iEEG Readings

It is well known that the volume of data of these recordings can easily get to the order of Tera or Peta bytes [8]. Therefore, there is a real need for efficient algorithms that support processing a big amount of data in reasonable execution times. Distributed computing adjusts to this need, making multiple computational resources collaborate to get the results faster by performing operations on smaller sets of data. A common practice is to perform all the computations on cloud systems, especially because of the high availability of computational and storage resources and their low costs. For example, solutions applying the Map-Reduce programming paradigm have been proposed with the main goal of partitioning computationally expensive tasks into smaller and disjoint tasks that can be executed in a distributed computing environment [9].

A review of the related literature allows identifying a few publications that describe different approaches to deal with massive iEEG recordings. Among these, the approach proposed by Burns and Freund [10] used a cloud-based Map-Reduce strategy implementing in Apache Hadoop to perform the processing and analysis of EEG data. An interesting fact about the solution by Burns and Freund is the way they propose to serialize data to make possible breaking the data clips and achieve a better distribution system. This feature is discussed in detail in Sect. 4, as we considered this approach for our proposal. This strategy allowed Burns and Freund to process and analyze almost 200 GB of EEG recordings in 4.5 h using 16 computing nodes, each of them with 16 processing units, whereas running the same test using a single computing resource took approximately 56 h.

Cloudwave is another cloud-based solution presented by the Division of Medical Informatics and the Department of Neurology of the Case Western Reserve University [9]. The Cloudwave project developed a platform that uses Apache Hadoop to process a large number of EEG readings. In the case study presented, the proposed solution managed to reduce the EEG processing time from 91–177 min on a standalone system, to just 7–11 min using Cloudwave.

Summarizing, implementations for processing extensive amounts of EEG data have shown promising efficiency and scalability results when executing over distributed memory systems. Our proposal explores two different parallel implementations: a message passing implementation to execute in a cluster system, for which we did not find related work in the related literature, and an implementation using Apache Hadoop, following the main ideas from related works about the distributed processing of iEEG data in the cloud.

3 Distributed Approaches for Processing iEEG Readings

This section describes the main challenges when working with the iEEG measurements in a distributed environment.

3.1 Working with iEEG Data in a Distributed Environment

When it comes to processing iEEG recordings, applying signal processing techniques for feature extraction are of utter importance. These techniques have scientifically proved that they are able to extract real useful information out of bare brain activity recordings [7]. However, in a distributed environment, the application of these methods is not straightforward.

Several decisions are needed to be taken in order to choose the appropriate HPC/distributed computing methodologies that, in combination with feature extraction algorithms, help to build a solution to deal with a real large scale datasets of iEEG measurements. Among these decisions, the most relevant ones concern the data and functional distribution.

3.2 Data and Functional Distribution Approaches

When working with data from hours-long brain activity recordings, an important decision to make involves choosing a strategy for splitting these recordings into smaller chunks with the main goal of improving the data distribution of the algorithm. This task implies knowing the data structure and its size.

A standard definition for the structure of large datasets of iEEG recordings specifies that the brain activity recordings must be split in clips of ten minutes long. Furthermore, the structure of a clip may variate depending on the measuring equipment used. Specific parameters that change are the number of measurements per second (i.e., the recording frequency), and also the number of channels of brain activity, which is given by the number of electrodes implanted into the brain of each patient [11].

Another key decision involves how the whole processing algorithm will be executed in a distributed environment. Because of many signal processing methods are computationally expensive, it is important to take into consideration the possibility of distributing certain steps of the processing algorithm in order to make all the computational resources work balanced, and consequently, generally improve the overall performance of the algorithm.

A final issue to consider is how the extracted features are persisted for later utilization. Due to the fact that the algorithms execute in a distributed environment, it is possible for certain iEEG clips to be processed in a different order than the original order given in the dataset. This is a problem considering that the feature extraction process requires all the features of the processed clips to respect a specific order, otherwise, the extracted features lack of meaning. Thus, a specific processing is needed to get the right order for clips.

Summarizing, defining the correct data and functional distribution strategies are very important when developing a distributed iEEG processing algorithm. The following section describes how these strategies are taken into account in the proposed implementations.

4 Two Implementations for Processing iEEG Readings on Distributed Computing Systems

This section presents a feature extraction algorithm and two different approaches for a distributed iEEG processing implementation.

4.1 The Proposed Processing Algorithm

As discussed in previous sections, the use of signal processing methods and statical analysis on iEEG data is very important for the feature extraction process. The proposed processing algorithm applies a similar approach to the one presented by Brinkmann et al. [12], which proved to be effective to predict seizures in human brain activity, reaching more than 80% of accuracy. The feature extraction process consists of applying complex signal-processing and statical analysis operations, such as computing Shannon entropy [13] and spectrum correlations [14] over all brain activity channels on the recordings. Because of the high computational complexity of these operations, implementations using HPC techniques are promising approaches to reduce the execution times.

4.2 Proposed Implementation Using a Message Passing Approach

The approach using message passing is based on different tasks that implement different parts of the whole distribution process, such as the way the data is partitioned, how it is processed, as also how the results are joined and persisted. More specifically, five different tasks were defined, namely *master*, *splitter*, *mapper*, *sorter*, and *reducer*. The implementation was developed using the Message Passing Interface (MPI) library [15].

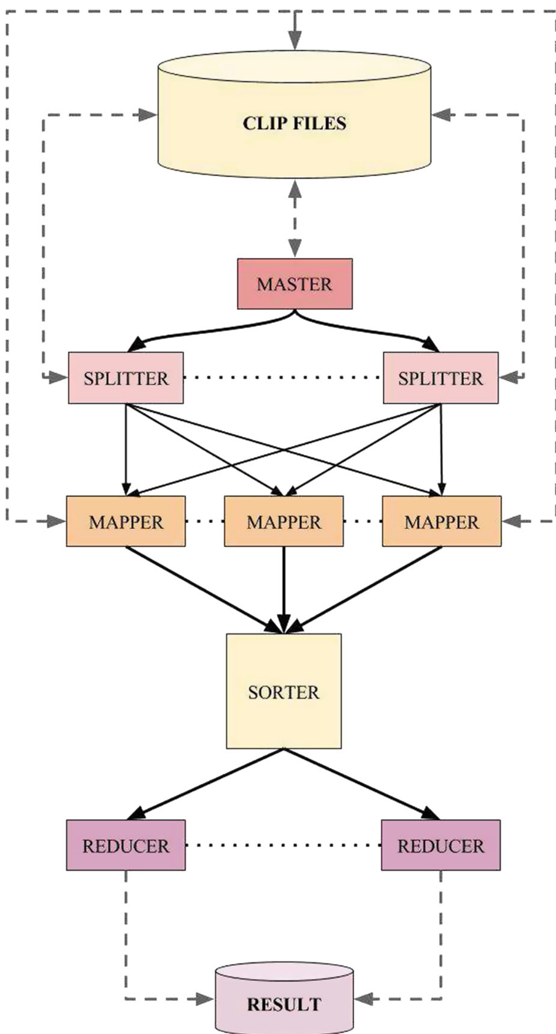


Fig. 1. Tasks communication and flow in the MPI implementation

A diagram of the MPI tasks and their relationships is shown in Fig. 1. Initially, a master task officiates as an orchestrator. It distributes the clips to be analyzed and keeps track on the processing status of each of them. This task knows the global state of the execution of the algorithm and takes actions accordingly. After that, a splitter task divides the iEEG data into smaller brain activity chunks. In the third step, a set of mapper tasks perform the processing of each iEEG chunk. Finally, a sorter task and a set of reducer tasks takes the mappers results and persist the processed clips for later utilization.

The proposed implementation needs at least one instance of each task in order to complete the whole process. Particularly, the more mapper and reducer tasks there are running, the better the algorithm distributes the iEEG data to these tasks to process.

This implementation was developed using *MPI for Python* [16], an open-source Python library that provides the standard MPI primitives and can handle and work with Python specific objects.

4.3 Map-Reduce Implementation Using Apache Hadoop

The second proposed implementation applies the Map-Reduce paradigm provided by the Apache Hadoop software. In this case, the mappers are the processes that receive the iEEG recordings, split them into smaller chunks, and perform the processing algorithm to each of these chunks. Once the processing algorithm finishes, the mappers output a tuple containing the name of the clip from which the iEEG chuck was retrieved, the index of that chunk, and finally, the processing result. These tuples are supplied to the reducers with the objective of joining all the chunk results of an iEEG clip, taking special consideration to the index to keep the correct order of the features. Once the reducers join all the expected chunks of a clip, they persist the results for later purposes.

Several decisions were taken in order to achieve a better performance when running the algorithm in Hadoop. Particularly, a preliminary efficiency analysis indicated that the format in which the iEEG chunks were received on the mappers drastically affected the performance of reading the brain activity channels. Three approaches were explored to overcome these efficiency drawbacks.

The first approach consisted of applying a solution similar to the one proposed by Burns and Freund [10], as described in Sect. 2. This approach proposes transforming the iEEG readings into serializable objects so that the raw brain activity channels could be streamed into the standard input of the mappers while they were active. A preliminary efficiency analysis of this approach demonstrated that the streaming speed of the brain activity to the mappers is not as fast as reading the same brain recordings from the file-system: in the experiments, the serializing approach took almost 2 s to retrieve all the iEEG data from a clip by applying streaming, whereas loading the same data from the file-system took less than 100 ms. On top of that, the process of making the iEEG data serializable (i.e., converting the data to the JSON format) is slow and results in a significant increase in the dataset size, which grow up to three times the original size in the preliminary experiments. Because of these efficiency results, the serializing approach was discarded.

A second approach consisted on compressing the serialized data, which reduced the size of the clips on disk. However, adding a decompression process on the mappers to read the iEEG data slows down the whole algorithm even more. Therefore, this alternative was also discarded.

Finally, a third proposal implied passing to the mappers a list of paths pointing to the clips locations in the file-system. Using this approach, preliminary efficiency experiments demonstrated that the streaming activity on the mappers

is significantly reduced and the brain activity recordings are loaded by reading directly from the iEEG data files. Consequently, the reading time of the raw iEEG readings was speeded up, taking around 90 ms instead of few seconds. Because of these results, we decided to base the proposed Hadoop implementation on this approach.

Figure 2 presents a diagram of the final Map-Reduce structure used in the Hadoop implementation. The task distribution applied in the algorithm is directly affected by the number of mappers and reducers available for concurrent execution, which can be set up from the configurations definitions of Hadoop. A sequential version of this algorithm only needs one mapper and one reducer in order to perform the whole execution.

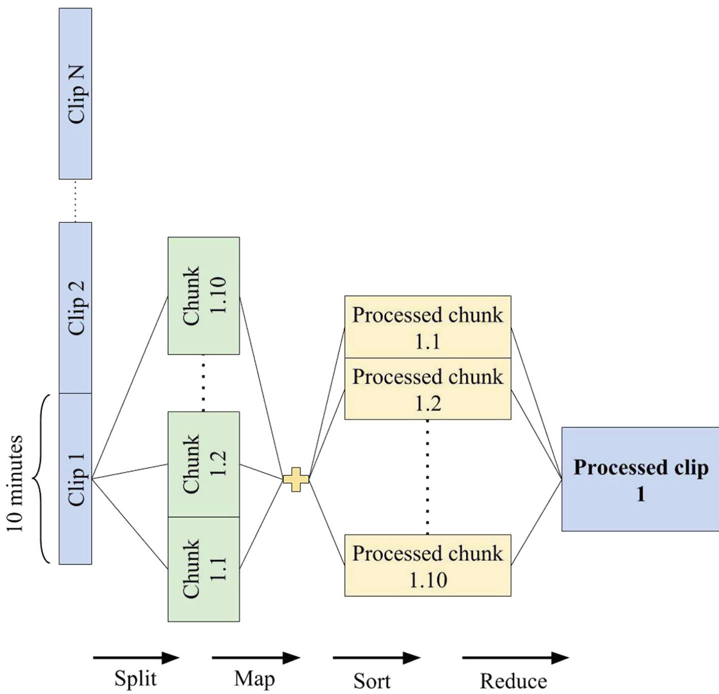


Fig. 2. Map-Reduce scheme used in the Apache Hadoop implementation

5 Experimental Evaluation

This section describes the experimental analysis of the proposed algorithms to evaluate and compare their performance. The main details about the execution environments and the working datasets are presented, and the performance results are reported and analyzed.

5.1 Execution Environments

Both proposals were evaluated in different execution environments, as well as using different dataset, according to the main features of each proposal and the environment itself.

MPI implementation. The MPI algorithm was evaluated on HP Proliant DL385 G7 servers (two AMD Opteron 6172 processors with 12 cores each, 72 GB RAM) from Cluster FING, the HPC infrastructure from Universidad de la República, Uruguay [17]. The iEEG data consisted 217 h of brain activity recorded at 400 Hz, with a total size of 5 GB of data.

Apache Hadoop implementation. The Hadoop algorithm was evaluated on a HP Proliant DL585 server (four AMD Opteron 6272 processors at 2.09 GHz, with 16 cores each, 48 GB RAM) also from Cluster FING. The working dataset consisted of 40 h of recordings at 5000 Hz, accounting for 27 GB of data.

5.2 Evaluation Metrics

All tests were executed four times to ensure the efficiency results were not affected by unpredictable factors in the execution environment, such as memory and CPU utilization levels.

The analysis considered the traditional metrics to evaluate the performance of parallel algorithms: the *speedup* and the *efficiency*. The speedup evaluates how much faster is a parallel algorithm than its sequential version. It is defined as the ratio of the execution times of the sequential algorithm (T_1) and the parallel version executed on N computing elements (T_N) (Eq. 1). The ideal case for a parallel/distributed algorithm is to achieve linear speedup ($S_N = N$). However, the common situation is to achieve sublinear speedup ($S_N < N$), due to the times required to communicate and synchronize the parallel/distributed processes or threads. The efficiency is the normalized value of the speedup, regarding the number of computing elements used for execution (Eq. 2). This metric allows comparing algorithms executed in non-identical computing platforms. The linear speedup corresponds to $E_N = 1$, and usually $E_N < 1$ [5].

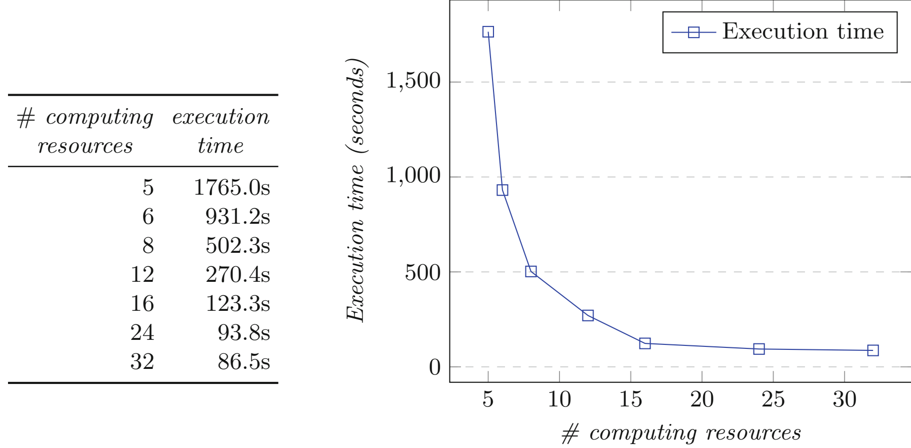
$$S_N = \frac{T_1}{T_N} \tag{1}$$

$$E_N = \frac{S_N}{N} \tag{2}$$

5.3 Computational Efficiency Analysis

This subsection reports and analyzes the performance results for both proposed implementations.

MPI implementation. The MPI algorithm was executed using 5, 6, 8, 12, 16, 24, and 32 computational resources, to process a dataset with 217 h of iEEG readings. For each number of computational resources, the implementation was executed four times. The average execution times of these executions are reported in Fig. 3. In addition, Fig. 4 reports the speedup and efficiency results.



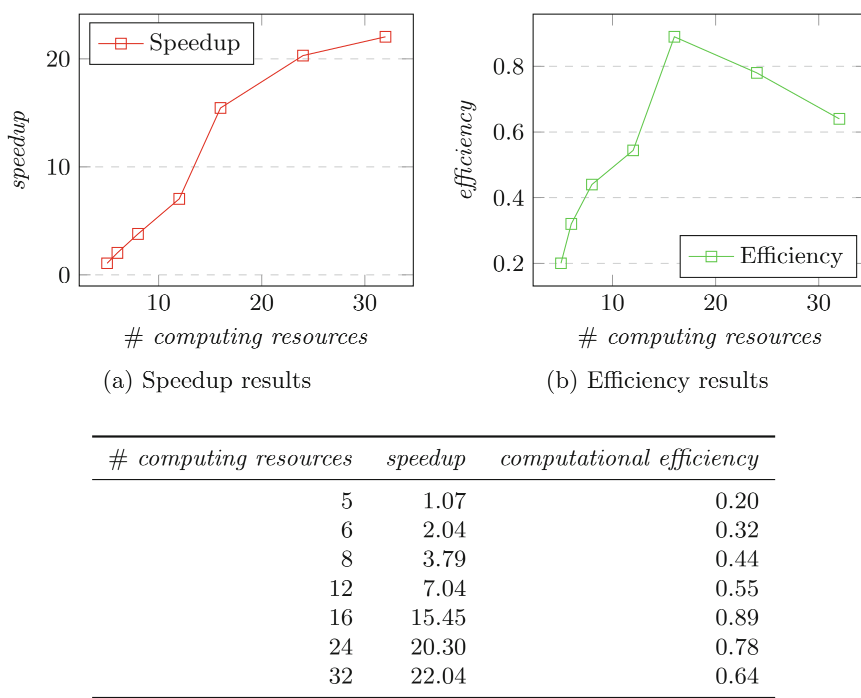
# computing resources	*execution time*
5	1765.0s
6	931.2s
8	502.3s
12	270.4s
16	123.3s
24	93.8s
32	86.5s

Fig. 3. Execution time results of the proposed implementation using MPI

The sequential version of the MPI algorithm demanded 31 min to process the studied iEEG dataset. However, using 32 computing resources in the cluster environment, the execution time was reduced to just 1 min and 27 s. Therefore, the speedup metric for the best execution time, using 32 computing resources, was 20.4 and the efficiency was of 0.64.

Efficiency results in Fig. 3 show a clear pattern; execution times rapidly reduce as the computing resources increase. However, for executions using more than 16 computing elements the execution time is slightly lower than the one using 16 resources.

The speedup results reported in Fig. 4a show a sublinear behavior, reaching a maximum of 22.4 when using 32 resources. Speedup is closer to linear when the algorithm was executed with 16 computing resources, obtaining a value of 15.45. This particular event is shown in the efficiency chart in Fig. 4b; efficiency reaches a peak of 0.89 at that exact point, meaning that the algorithm performing on 16 computing elements is more efficient when compared to the executions using 24 or 32 resources.



# computing resources	*speedup*	*computational efficiency*
5	1.07	0.20
6	2.04	0.32
8	3.79	0.44
12	7.04	0.55
16	15.45	0.89
24	20.30	0.78
32	22.04	0.64

Fig. 4. Speedup and computational efficiency results for the MPI implementation

Hadoop implementation. For the experimental evaluation, the proposed Hadoop algorithm was set up to process the whole 27 GB dataset using 8, 12, 16 and 24 computing resources concurrently. For each of these executions, only one reducer task was created. The remaining processes were assigned to mapper tasks since the mappers resulted to be the most complex tasks and they perform the most time-consuming operations.

The average execution times for all proposed executions are reported in Fig. 5. Finally, Fig. 6 reports the detailed speedup and computational efficiency metrics results.

The sequential version of the proposed Hadoop algorithm, which uses just one mapper and one reducer to process the whole working dataset, demanded more than 13 h to finish. When using all the 24 available computing resources in the execution environment, the execution time was reduced to 1 h and 23 min. This execution time reduction implied a speedup of 9.16 and a computational efficiency of 0.38 when using all the computing resources.

Overall, efficiency results indicate that the Hadoop implementation obtained lower performance improvements than the MPI implementation, probably because Hadoop is not only in charge of running the mapper and reducer tasks but also many other services to make the Hadoop ecosystem more robust (including failover mechanisms and auditing input/output interactions within the tasks). These results are consistent with situations reported in related works [18,19].

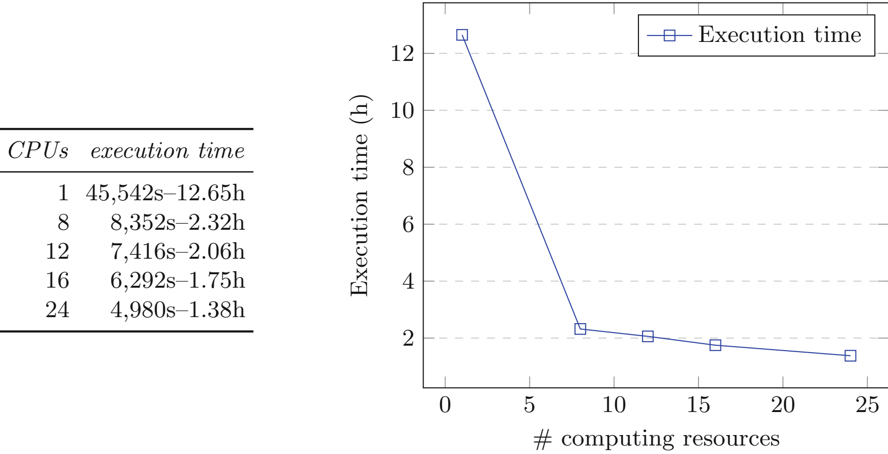
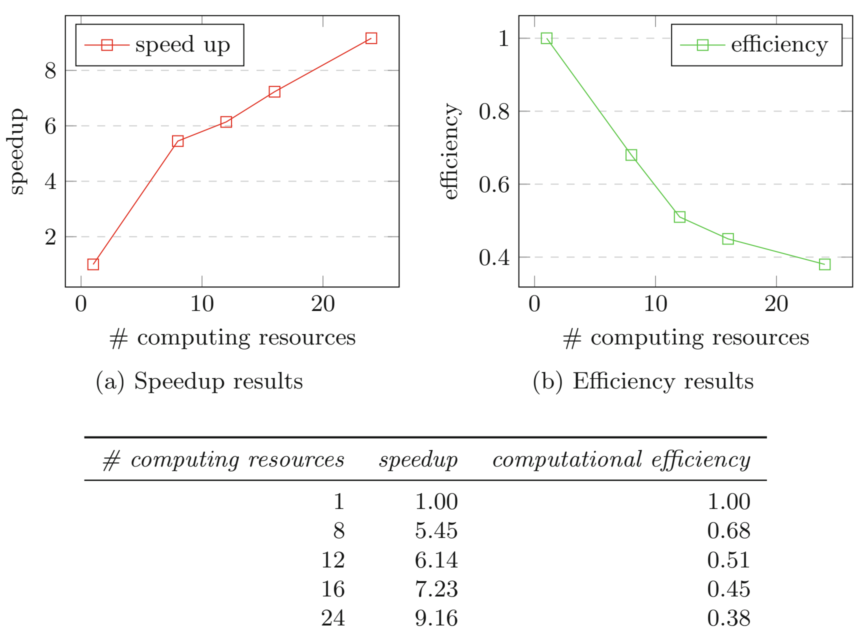

CPUs	*execution time*
1	45,542s–12.65h
8	8,352s–2.32h
12	7,416s–2.06h
16	6,292s–1.75h
24	4,980s–1.38h

Fig. 5. Execution time results of the proposed solution using Apache Hadoop



(a) Speedup results

(b) Efficiency results

# computing resources	*speedup*	*computational efficiency*
1	1.00	1.00
8	5.45	0.68
12	6.14	0.51
16	7.23	0.45
24	9.16	0.38

Fig. 6. Speedup and computational efficiency results for the implementation on Apache Hadoop

As well as in the MPI implementation, execution times reduce when using a larger number of computing elements. Even though the speed up values grow, it demands the algorithm using many computational resources to reduce the execution time to a small degree. For example, when comparing the results

obtained using 8 resources, it required 3× more resources to reduce the execution time by 40 min. This issue can be appreciated in detail in the efficiency chart in Fig. 6b, showing a clear descending behavior towards less efficient execution times when the number of computing resources increases.

6 Conclusions and Future Work

This article presented two implementations for a parallel algorithm to process iEEG readings on distributed environments. The first implementation applied a traditional approach for parallel and distributed algorithms based on message passing, adapted for executing on cluster systems. The second implementation was developed using Apache Hadoop, a software platform that uses Map-Reduce programming techniques to work with large volumes of data on cloud environments. Both strategies processed brain activity recordings by applying sophisticated and complex signal processing and statical analysis methods. Sequential versions of these algorithms demanded several hours to process standard iEEG datasets. However, parallel and distributed computing techniques can be applied with the main goal of reducing the processing time of the brain recordings.

The experimental evaluation of the proposed algorithms demonstrated that applying HPC techniques in distributed systems allows reducing the execution time of processing brain activity recordings. When compared with proposals from the related literature, the implemented algorithms obtained similar results with regard to the reduction of the execution times in distributed environments. The results showed that the proposed MPI implementation is able to process in 1 min and 27 s a 270 h dataset of brain recordings that demands 31 min to be processed with a sequential version of the algorithm. On the other hand, the distributed Hadoop implementation performed the processing on a 27 GB dataset in 1 h and 23 min, whereas a sequential version of the same algorithm demanded more than 13 h to conclude.

Regarding the efficiency of the algorithms, the MPI implementation performed better than the proposed Hadoop implementation. The main reason for this performance behavior is that the MPI implementation is much lighter than the one using Hadoop. More precisely, additional actions performed by the Hadoop ecosystem causes a significant overhead on the tasks, thus the MPI implementation resulted more efficient when using more computing resources. The MPI algorithm achieved a maximum speedup of 22.4 and an efficiency of 0.64 when using 32 computing elements. However, the MPI algorithm was more efficient when using 16 computing resources achieving an efficiency of **0.89** and a speedup of **15.45**. On the other hand, the Hadoop implementation reached a speedup of 9.16 and an efficiency of 0.38 when using 24 computing elements and the best efficiency result of **0.68** when using 8 resources.

The main lines for future work are related to extending the capabilities and the computational efficiency of the proposed algorithms. The first issue involves including in the distributed algorithms a classification system capable of detecting if a patient may or not have a seizure, advancing on building a complete

solution for the seizure prediction problem. The classification system can also be speeded up by using HPC techniques. Regarding the computational efficiency, specific improvements can be achieved for processing larger iEEG datasets by using the computing power available in nowadays distributed computing platforms (Azure, AWS, etc.). Studying the scalability of the proposed approaches over these realistic platforms is another promising line for future work.

References

1. Mormann, F., Andrzejak, R., Elger, C., Lehnertz, K.: Seizure prediction: the long and winding road. Brain **130**(2), 314–333 (2006)
2. Gadhoumi, K., Lina, J., Mormann, F., Gotman, J.: Seizure prediction for therapeutic devices: a review. J. Neurosci. Methods **260**, 270–282 (2016)
3. Davis, K., Sturges, B., Vite, C., Ruedebusch, V., Worrell, G., Gardner, A., Leyde, K., Sheffield, W., Litt, B.: A novel implanted device to wirelessly record and analyze continuous intracranial canine EEG. Epilepsy Res. **96**(1), 116–122 (2011)
4. Feldwisch-Drentrup, H., Schelter, B., Jachan, M., Nawrath, J., Timmer, J., Schulze-Bonhage, A.: Joining the benefits: combining epileptic seizure prediction methods. Epilepsia **51**(8), 1598–1606 (2010)
5. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
6. Ramgopal, S., Thome-Souza, S., Jackson, M., Kadish, N., Fernández, I., Klehm, J., Bosl, W., Reinsberger, C., Schachter, S., Loddenkemper, T.: Seizure detection, seizure prediction, and closed-loop warning systems in epilepsy. Epilepsy Behav. **37**, 291–307 (2014)
7. Ting, W., Guo-zheng, Y., Bang-hua, Y., Hong, S.: EEG feature extraction based on wavelet packet decomposition for brain computer interface. Measurement **41**(6), 618–625 (2008)
8. El Kassabi, H., Serhani, M., Dssouli, R.: Neurodegenerative diseases monitoring (NDM) main challenges, tendencies, and enabling big data technologies: a survey. Neurodegenerative Dis. **5**(01), 158–173 (2016)
9. Jayapandian, C., Chen, C., Bozorgi, A., Lhatoo, S., Zhang, G., Sahoo, S.: Cloudwave: distributed processing of "big data" from electrophysiological recordings for epilepsy clinical research using Hadoop. In: AMIA Annual Symposium Proceedings, pp. 691–700 (2013)
10. Burns, M., Freund, Y.: Large scale electroencephalography processing with Hadoop (2013). https://github.com/mattb243/EEGLAB2Hadoop. Accessed July 2017
11. Nuwer, M., Comi, G., Emerson, R., Fuglsang-Frederiksen, A., Guerit, J., Hinrichs, H., Ikeda, A., Luccas, F., Rappelsberger, P.: IFCN standards for digital recording of clinical EEG. Electroencephalogr. Clin. Neurophysiol. **106**(3), 259–261 (1998)
12. Brinkmann, B., Wagenaar, J., Abbot, D., Adkins, P., Bosshard, S., Chen, M., Tieng, Q., He, J., Muñoz-Almaraz, F., Botella-Rocamora, P., Pardo, J.: Crowdsourcing reproducible seizure forecasting in human and canine epilepsy. Brain **139**(6), 1713–1722 (2016)
13. Bruhn, J., Lehmann, L., Röpcke, H., Bouillon, T., Hoeft, A.: Shannon entropy applied to the measurement of the electroencephalographic effects of desflurane. Anesthesiol.: J. Am. Soc. Anesthesiol. **95**(1), 30–35 (2001)

14. Bendat, J., Piersol, A.: Engineering Applications of Correlation and Spectral Analysis. Wiley-Interscience, New York (1980)
15. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press, Cambridge (1999)
16. Dalcin, L., Paz, R., Kler, P., Cosimo, A.: Parallel distributed computing using Python. Adv. Water Resour. **34**(9), 1124–1139 (2011)
17. Nesmachnow, S.: Computación científica de alto desempeño en la facultad de ingeniería, universidad de la república. Revista de la Asociación de Ingenieros del Uruguay **61**(1), 12–15 (2010). (Text in Spanish)
18. White, T.: How MapReduce works (Chap. 6). In: Hadoop: The Definitive Guide. O'Reilly Media Inc. (2012)
19. Dittrich, J., Quiané-Ruiz, J.: Efficient big data processing in Hadoop MapReduce. Proc. VLDB Endow. **5**(12), 2014–2015 (2012)

Support Vector Machine Acceleration for Intel Xeon Phi Manycore Processors

Renzo Massobrio[1,2(✉)], Sergio Nesmachnow[1], and Bernabé Dorronsoro[2]

[1] Universidad de la República, Montevideo, Uruguay
{renzom,sergion}@fing.edu.uy
[2] Universidad de Cádiz, Cádiz, Spain
bernabe.dorronsoro@uca.es

Abstract. Support vector machines are widely used for classification and regression tasks. However, sequential implementations for support vector machines are usually unable to deal with the increasing size of current real-world learning problems. In this context, Intel®Xeon Phi™ processors allow easily incorporating high performance computing strategies to improve execution times. This article proposes a parallel implementation of the popular LIBSVM library, specially adapted to the Intel®Xeon Phi™ architecture. The proposed implementation is evaluated using publicly available datasets corresponding to classification and regression tasks. Results show that the proposed parallel version computes the same results than the original LIBSVM while reducing the time needed for training by up to a factor of 4.81.

1 Introduction

Support vector machines (SVMs) are supervised learning models which are used for classification and regression analysis [2]. SVMs have been widely applied to solve various real world problems, e.g., text categorization, image classification, hand-written text recognition, protein classification.

In their simplest form, SVMs are non-probabilistic binary linear classifiers. SVMs build a model given a set of training samples, each marked as members of one of two possible classes. This model is a representation of the training samples as points in space that aims at separating samples from different classes by the widest possible gap. Figure 1 shows an example of a linear binary SVM classifier, with three possible classifications, where H1 does not separate the two classes, H2 separates them, but H3 separates the classes creating the widest gap between both groups. Once the model is built, new, unknown samples can be mapped into that same space in order to predict the class to which they belong.

More generally, a SVM builds a hyperplane (or set of hyperplanes) in a high-dimensional space that has the largest distance to the nearest training-data point of any class (known as the functional margin). In most cases, the points to be classified are not linearly separable in the input dimensional space. Therefore, the input points can be mapped into a higher-dimensional space, to

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 277–290, 2018.
https://doi.org/10.1007/978-3-319-73353-1_20

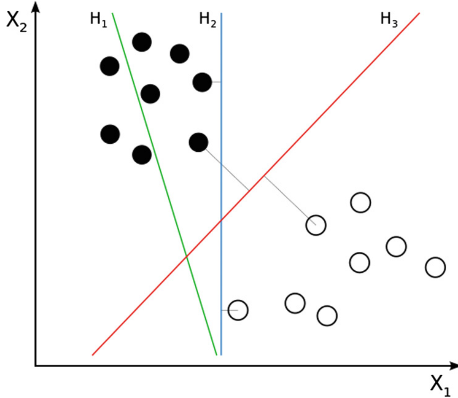


Fig. 1. Example of a linear binary SVM classifier

make the separation task easier. For this purpose, several kernel functions have been proposed, which project the original points into higher dimensional spaces. Then, the SVM finds a linear hyperplane that separates the points with the maximal margin in this higher dimensional space.

Classification and regression tasks demand increasing computational resources due to the growing size of datasets and input vectors used in real applications nowadays. Therefore, sequential SVM implementations are not able to efficiently cope with large learning tasks. Parallel implementations are necessary to be able to handle large learning problems with SVMs. In this context, many-core processors, such as Intel®Xeon Phi™, are an interesting option to exploit parallelism using a standard CPU form factor, without the need for large and expensive HPC infrastructure.

This article proposes a parallel implementation of the popular library for support vector machines LIBSVM [3], specifically adapted to the latest Intel®Xeon Phi™ architecture. LIBSVM is an easy-to-use research tool that supports vector classification, regression, and distribution estimation problems, includes several SVM formulations, and various kernel functions. The proposed implementation is evaluated using datasets of classification and regression problems. Results show that the proposed implementation is able to significantly reduce training times, which are the most demanding in most learning problems, while computing exactly the same results than the original LIBSVM. For the evaluated datasets, the best results indicate that the proposed implementation accelerates LIBSVM in up to 4.81x.

The remainder of the manuscript is organized as follows. Section 2 introduces manycore processors, the Intel®Xeon Phi™ architecture, and the software used in the parallel implementation of LIBSVM. A brief review of related works is presented in Sect. 3. Then, Sect. 4 presents in detail the proposed LIBSVM implementation and Sect. 5 presents the experimental results. Finally, Sect. 6 presents the conclusions and main lines of future work.

2 Hardware and Software Platform

This section introduces manycore processors and presents the main characteristics of the Intel®Xeon Phi™ architecture. Then, a brief introduction to the software used for the parallel implementation is presented.

2.1 Manycore Processors and Intel®Xeon Phi™

Manycore processors are multi-core processors specially designed for a high degree of parallelism, consisting of tens or thousands of simpler independent cores. The use of manycore processors has been increasing in the past years, with extensive applications in embedded systems and high-performance computing platforms. Sunway TaihuLight is the fastest supercomputer as of June 2017 according to the TOP500 ranking [15]. This Chinese supercomputer, installed at the National Supercomputing Center in Wuxi, consists of 40,960 manycore processors with 260 cores each, totalling 10,649,600 cores [5].

Xeon Phi™ is a brand name given to a series of manycore processors commercialized by Intel®. This family of processors was initially designed as an add-on PCIe card which could be connected to a standard CPU and used for computing intensive tasks. A second generation of Xeon Phi™products, with codename Knights Landing, was announced on June 2013. The main difference with its prior generation is that Knights Landing are stand-alone processors that can boot an off-the-shelf operating system. Therefore, Knights Landing avoids the bottlenecks in PCIe communications —which are inherent in coprocessors— and provides a powerful HPC platform in a standard CPU form factor. Sodani et al. [14] presents an interesting overview of the Knights Landing architecture. A brief description of the main characteristics of the Knights Landing architecture are described next.

The Knights Landing architecture consists of 38 physical tiles: at most 36 are active and the remaining two are used for recovery purposes. Each tile has two cores, two vector processing units (VPUs) per core, and a shared 1 MB L2 cache. The processing cores derive from the Intel®Atom™ core microarchitecture, but incorporate several modifications specially designed to suit HPC workloads, e.g., support for four threads per core, larger and faster caches, and larger translation look-aside buffers (TLBs). Each core supports up to four hardware contexts or threads by means of hyperthreading techniques. Additionally, Knights Landing incorporates a new instruction set named AVX-512, which supports 512 bit vector instructions and a 2D mesh that interconnects the tiles with other components of the chip such as memory and I/O controllers. Knights Landing introduces an innovative memory architecture comprising two types of memory: Multichannel DRAM (MCDRAM), which provides high bandwidth, and double data rate (DDR) memory for larger capacity. These two types of memory can be used in three different memory modes: (i) *cache mode*, where MCDRAM acts as cache for DDR; (ii) *flat mode*, where MCDRAM is used as standard memory sharing the same address space as DDR; and *hybrid mode*, where a portion of

MCDRAM is in cache mode and the remainder is in flat mode. All these features make the Knights Landing architecture a good candidate for HPC tasks, without requiring any special way of programming other than the standard CPU programming model, and even having decent support for serial legacy code.

2.2 Intel®C++ Compiler

The Intel®C++ compiler is part of the Intel®Parallel Studio XE suite. The compiler incorporates many optimizations to take advantage of specific processor features, such as the number of available cores and wider vector registers, to speed up computations. Intel®C++ compiler has broad support for current and previous C and C++ standards, including full support for C++11 and C99. Furthermore, it supports integration with OpenMP for parallel implementations.

2.3 Intel®Math Kernel Library

Intel®Math Kernel Library (MKL) supports a series of optimized and threaded math functions to take advantage of the architecture of Intel®processors to solve large computational problems. MKL performs a hardware check on runtime and selects suitable functions to improve execution time by means of instruction-level and register-level SIMD parallelism [16]. Intel®MKL also incorporates thread-safe functions to speedup computations using OpenMP. Although Intel®MKL is optimized for the latest Intel®processors, including processors with multiple cores, it can also be used in non-Intel®CPUs. The library provides Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) routines, fast Fourier transforms, vectorized math functions, random number generation functions, and many other features. In this work, we use specific functions in the BLAS Level 1 suite, which includes routines and functions that perform vector to vector math operations.

3 Related Work

There are multiple libraries and frameworks that implement SVMs in a variety of programming languages. In 2007, Ivanciuc [11] provided a thorough list and comparison of different SVM implementations.

LIBSVM (Library for Support Vector Machines) was developed by Chang and Lin [3] and supports vector classification, regression, and distribution estimation problems. It was developed with the goal of helping users from other fields to easily use SVM as a research tool. LIBSVM provides a simple interface for users to link with their own programs. The main features of this framework include: several SVM formulations, efficient multi-class classification, cross validation, probability estimates, various kernels (e.g., linear, polynomial, radial basis function (RBF), and sigmoid), and weighted SVM for unbalanced data. It is implemented in both C++ and Java, and also has interfaces in the following languages: Python, R, MATLAB, Perl, Ruby, Weka, Common LISP, CLISP, Haskell, OCaml, LabVIEW, and PHP.

GPU-accelerated LIBSVM [1] is a modified version of the original LIBSVM code that takes advantage of the CUDA framework to significantly reduce processing time while producing identical results to the original framework. This GPU version does not currently supports all the features of the original LIBSVM and is only compatible with C-SVC classification mode using the RBF kernel. From the users perspective, the functionality and interface of LIBSVM remains the same. However, the kernel computation is performed using a GPU. The authors propose a combined CPU + GPU approach where the computation of the kernel matrix elements is offloaded to the GPU, to decrease the processing time for the training phase. The experimental evaluation was performed using a training dataset consisting of high level features in video shots. Unfortunately, the datasets used are not publicly available for comparison. In small datasets, the difference between using only CPU and the combined CPU + GPU approach is hardly noticeable. However, for larger instances, the execution time when exploiting the GPU is up to one order of magnitude lower in comparison to using the CPU alone. The authors mention that it is important to notice the memory issues that can arise in very large instances due to the limited memory resources in GPU cards.

You et al. [17] presented a custom implementation of binary classification SVM for multi-core and many-core architectures. The proposed framework is evaluated using Intel®Ivy Bridge CPUs and Intel®Xeon Phi™ co-processor (MIC) from the previous generation to the one used in our proposal (Knights Landing). Several strategies are presented to improve the parallelism and optimize the implementation to the underlying architectures, including: parallel kernel evaluation, affinity models for thread-to-core assignment, vectorization, and memory alignment. The authors suggest using both "sparse" and "dense" formats to represent training vectors, depending on the characteristics of the training dataset, as it will be discussed in Sect. 4.3. An interesting approach is suggested to choose the format automatically. The authors propose training different datasets using both "sparse" and "dense" formats and measuring execution times. Afterwards, they propose building a classifier with features of each trained dataset (e.g., number of vectors, size of vectors, density) and labels indicating whether it was faster to use the "sparse" or the "dense" format. With this trained classifier, a user could predict whether their dataset would benefit from using a specific representation format. This is an interesting approach that could also be applied to our proposal. However, the authors do not provide specific details about the accuracy of the implemented format predictor. The proposed SVM implementation is able to improve LIBSVM by 4.4x–84x on MIC and 18x–47x on Ivy Bridge CPU. The training results are not exactly the same as those computed by LIBSVM due to implementations nuances. However, the authors show that the differences in accuracy for the studied datasets are minimal. Unfortunately, the proposed library does not have an official release and the only code available online has not been updated for two years as of 2017.

The analysis of related works shows that there is an interest in the research community for optimizing SVMs due to the growing size of learning problems.

However, the most popular SVM libraries are serial and do not exploit the parallelism offered by modern manycore and multi-core architectures. Some efforts have been made either to adapt existing libraries or to implement new ones, that take advantage of highly parallel computing architectures. There is still room to contribute in adapting popular SVM implementations, such as LIBSVM, to modern manycore architectures like Intel®Xeon Phi™.

4 LIBSVM Implementation for Intel®Xeon Phi™

This section presents the specific modifications performed to the LIBSVM code to adapt it to the Intel®Xeon Phi™ architecture. The proposed changes were implemented over version 3.22 of the LIBSVM code.

4.1 Coarse-Grain Parallelism Using OpenMP

The original LIBSVM code is sequential. Therefore, it does not take advantage of the multiple cores present in Intel®Xeon Phi™ machines. A simple way to exploit the availability of multiple cores is by using OpenMP to parallelize the loop that processes each training vector, as suggested in the LIBSVM FAQ [4]. This approach involves two simple modifications to the original LIBSVM code: (i) adding two OpenMP pragmas: one before the for loop in function `get_Q` of class `SVC_Q` and one before the for loop in function `svm_predict_values`; (ii) adding the `-fopenmp` flag to the LIBSVM Makefile. This modification applies to classification tasks, the approach is similar for regression tasks.

The pragma line added to the LIBSVM code is the following, where `j` is the loop variable:

```
#pragma omp parallel for private(j) schedule(guided).
```

The `guided` option uses a work queue to give each thread a chunk-sized block of loop iterations. When a thread finishes processing the assigned block, it retrieves the next block of loop iterations from the top of the work queue. Thus, an automatic load balancing scheme is implemented. The size of the chunk is initially large and decreases over time to better handle load imbalance between iterations. By default, the starting chunk size is approximately equal to the ratio between the number of iterations of the loop and the number of threads.

The number of threads to be used can be set on runtime by setting the appropriate value in the environment variable `OMP_NUM_THREADS`.

4.2 Compiling with Intel®C++ Compiler

The previous modification works with any C++ compiler that supports OpenMP, without the need of compiling using the Intel®C++ compiler. However, in order to exploit the specific characteristics of the Intel®Xeon Phi™ architecture, and to be able to use the Intel®MKL, we propose changing the compiler used in the original LIBSVM code (g++) for the Intel®C++ compiler.

The compiler options were set according to the recommendations of the Intel®Math Kernel Library Link Line Advisor [10]. This allows to optimize the compiled code to the specific hardware architecture where the program will execute. In addition to the recommended optimization flags, it was necessary to include the `-fp-model precise` option. This flag instructs the compiler to avoid performing several optimizations targeted to floating point arithmetic. While this prevents from further optimization, it guarantees returning the same results as the original LIBSVM library. If the user allows some float precision differences with the original LIBSVM, this flag can be removed and achieve an extra improvement in performance.

4.3 Integration with Intel®MKL

The previous modifications apply to all learning tasks, independently of the chosen kernel function. However, in the rest of the work we focus on accelerating the training time for classification and regression tasks using the RBF kernel function. The rationale behind this decision is that the RBF kernel is, in general, a reasonable choice for many learning tasks, as recommended by the LIBSVM Practical Guide [8]. The RBF kernel function maps training samples into a higher dimensional space, so it can handle non-linear relations between attributes and class labels. Additionally, it has fewer hyperparameters than other kernels (e.g., polynomial kernel) and presents fewer numerical difficulties than other kernels. The RBF kernel on two samples x and x', represented as feature vectors in some input space, is defined by Eq. 1, where $\gamma > 0$ is a free parameter.

$$K(x, x') = e^{-\gamma||x-x'||^2} \tag{1}$$

An initial profiling experiment of the `svm-train` program was performed to identify bottleneck functions. The profiling was performed using *gprof* [6] and training on the *connect-4* dataset [13] available at the LIBSVM dataset repository [3]. This dataset is not used in the experimental evaluation of the proposed solution in order to avoid bias and to demonstrate that the implementation scales well on different datasets.

The profiling of the `svm-train` program revealed that 87.2% of the total execution time was spent in the `dot` function of class `Kernel`. This function performs the dot product of two vectors. The dot product is used when computing the RBF kernel, since the LIBSVM implementation expands Eq. 1 into the form presented in Eq. 2.

$$K(x, x') = e^{-\gamma\left((x\boldsymbol{\cdot}x) + \left(x'\boldsymbol{\cdot}x'\right) - 2\left(x\boldsymbol{\cdot}x'\right)\right)} \tag{2}$$

The profiling results suggest that an improvement in the dot product calculation would greatly impact the overall training time. To improve the `dot` function we propose using the `cblas_ddot` routine available in the BLAS Level 1 group of functions and routines of Intel®MKL, which has the following header:

```
double cblas_ddot (const MKL_INT n, const double *x, const
MKL_INT incx, const double *y, const MKL_INT incy);
```

where the parameters are:

- n, the number of elements in vectors x and y.
- x, an array with size at least $(1 + (n - 1) \times abs(\texttt{incx}))$.
- incx, the increment for the elements of x.
- y, an array with size at least $(1 + (n - 1) \times abs(\texttt{incy}))$.
- incy, the increment for the elements of y.

Using the cblas_ddot function instead of the original code in LIBSVM is not as straightforward as interchanging only that portion of the code, due to the format in which LIBSVM stores the training vectors. LIBSVM uses a "sparse" format, in which zero values are not stored. Instead, training vectors are stored as <index:value> pairs. For instance, the training vector <0,1,0,3> is internally represented as (2:1 4:3). This format does not allow using the cblas_ddot function directly. Therefore, it was necessary to modify other sections of the LIBSVM code to implement a "dense" format, in which vectors are stored directly as arrays, including zero values. As it will be presented in the experimental analysis discussion in Sect. 5, this design decision may achieve better or worse performance depending on the specific characteristics of the training set used. Therefore, we implemented this modification in a way that the user can decide at compiling time whether to use the original 'sparse' format or the proposed 'dense' format, depending on the specific characteristics of the training dataset.

The cblas_ddot implementation is threaded. The number of threads to be used can be set on runtime by setting the environment variable MKL_NUM_THREADS.

5 Experimental Analysis

This section presents and discusses the experimental results of the proposed parallel implementation of LIBSVM for Intel®Xeon Phi™ Knights Landing systems.

5.1 Execution Platform

The experimental evaluation was performed on an Intel®Xeon Phi™ 7250 processor, with 68 cores, and 64 GB of RAM. The server ran Linux Ubuntu 16.04 and had version 17.0.1 of the Intel®C++ compiler. The server was not shared with other users or performed any other intensive tasks during the experiments, in order to accurately measure the execution times.

5.2 Problem Instances

Three learning datasets were used for the experimental evaluation of the proposed implementation. These datasets were obtained from the LIBSVM dataset repository [3]. The datasets, which correspond to classification and regression problems, are:

– *gisette*, a dataset corresponding to a handwritten digit recognition problem with the goal of separating the digits '4' and '9'. This dataset was one of five datasets of the NIPS 2003 feature selection challenge [7].
– *E2006*, a dataset with reports from thousands of publicly traded U.S. companies, published in 1996–2006, and stock return volatility measurements in the twelve-month period before and the twelve-month period after each report [12].
– *usps*, a database for handwritten text recognition research, consisting of digitalized images at 300 pixels/in. in 8-bit gray scale, corresponding to U.S. post codes scanned from real mail [9].

The datasets *gisette* and *E2006* were sub-sampled using the `subset.py` script included in LIBSVM, which allows making a stratified selection of training samples, keeping the rate of appearance of each label in the dataset. We used a subset of 1000 samples of each dataset. Therefore, we will refer to them as *gisette_1000* and *E2006_1000* for the remainder of the manuscript. Table 1 shows the main characteristics of each of the datasets used in the experimental evaluation, including the type of problem (classification or regression), the number of training vectors (# samples), the length of each training vector (# features) and, for classification problems, the number of classes (# labels).

Table 1. Datasets used for the experimental evaluation

	Problem	# samples	# features	# labels
gisette_1000	Classification	1000	5000	2
E2006_1000	Regression	1000	150360	-
usps	Classification	7291	256	10

5.3 Coarse-Grain Parallelization

Initially, we report the results achieved when using the Intel®C++ Compiler and OpenMP for coarse-grain parallelism of the outer loop that performs the kernel evaluations. These results correspond to the modifications described in Sects. 4.1 and 4.2, i.e., without changing the vector representation format and without using Intel®MKL. Figure 2 shows the average execution time in seconds for the three studied instances, when varying the number of threads (`OMP_NUM_THREADS` environment variable) assigned to the loop. The results correspond to 30 independent executions of each instance with each studied number of threads.

Results in Fig. 2 show that acceptable execution time improvements are achieved when using more than one core on all studied instances. However, execution times do not improve when using more than 64 cores for both *gisette_1000* and *E2006_1000* datasets, and there is even a significant negative impact when using large number of cores with the *usps* dataset. This could be explained due to the fact that the Intel®Xeon Phi™ processor used has 68 physical cores. Therefore, when using more threads, some of the CPU resources are shared among the threads, incurring in a noticeable overhead.

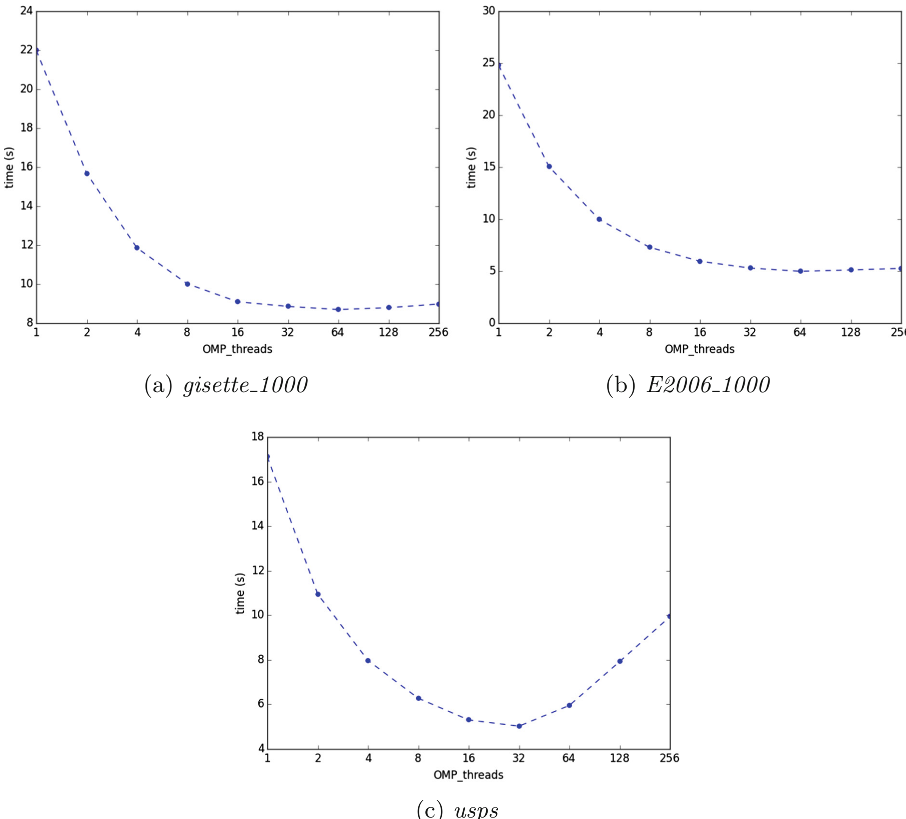


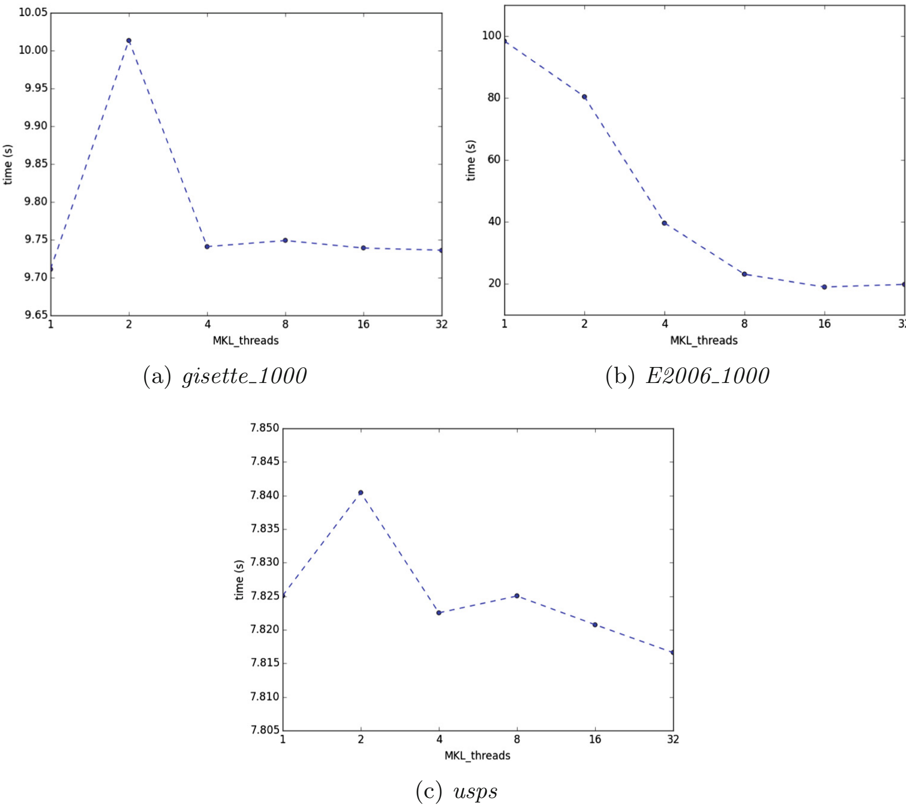
Fig. 2. Mean execution time with different number of OMP threads

5.4 Vectorized Dot Product Computation

In a second stage, we evaluated the results of implementing the modifications described in Sect. 4.3, i.e., changing from a "sparse" to a "dense" vector representation and including Intel®MKL for the dot product calculation. Figure 3 shows the average execution time in seconds for the three studied instances, when varying the number of threads (`MKL_NUM_THREADS` environment variable) assigned to the dot product calculation. There is no coarse-grain parallelization in these executions (i.e., `OMP_NUM_THREADS` = 1). The results correspond to 30 independent executions of each instance with each studied number of threads.

Results in Fig. 3 give information on two aspects. Firstly, on the convenience (or not) of using the "dense" format and including Intel®MKL for the dot product calculation. Secondly, to discuss the usefulness of adding parallelism at the vector level when computing the dot product.

To discuss the first aspect, we should compare the execution times when using only one OMP thread in Fig. 2 vs using one MKL thread in Fig. 3. It can be seen that execution times significantly improve when running a sequential version



(a) *gisette_1000* (b) *E2006_1000*

(c) *usps*

Fig. 3. Mean execution time with different number of MKL threads

with the "dense" representation format and the MKL dot product calculation for both *gisette_1000* and *usps* instances. However, for *E2006_1000* instance, since the training vectors are much larger, using the "dense" format and only one MKL thread negatively impacts the execution time. In this case, the effects of the "dense" representation are only mitigated when adding more MKL threads to reduce the execution times.

Regarding the second aspect, results show that when using the "dense" format, the improvements achieved by using a larger number of threads for the dot product computation are only noticeable for very large vectors. For *usps* instance, with vectors of size 256, the improvements when using more than one thread are marginal. For *gisette_1000*, with vectors of size 5000, there is even some minor performance decline when using more than one thread. Additionally, there is a strange behaviour when using exactly two threads, possibly due to the overhead of creating the pool of threads. However, for instance *E2006_1000*, with training vectors of size 150360, there is a noticeable improvement when using more threads for the dot product calculation.

In conclusion, dense vectors benefit from changing the original LIBSVM representation and using Intel®MKL for the dot product calculation, but only dense and large vectors benefit from using multiple threads when computing each dot product. The proposed implementation allows the user to control both the vector representation and the number of outer (OMP) and inner (MKL) threads, thus, enabling the user to tune the library to the specific needs or their learning task. A tool like the one suggested by You et al. (2014) [17] would be interesting to develop, in order to suggest users the values for these parameters that best fit their dataset.

5.5 Two-Level Parallelization Approach

Taking into account the results discussed in the previous sections, we performed 30 independent executions of each training dataset, using the configuration of threads that achieved best results. The selected configurations are reported in Table 2, where `OMP_NUM_THREADS` indicates the number of threads used for the outer loop of kernel evaluations (i.e., coarse-grain parallelism) and `MKL_NUM_THREADS` indicates the number of threads assigned to compute each vector dot product, when the "dense" format is used. Additionally, 30 independent executions of the original LIBSVM library were performed over each dataset.

Table 2. Thread configurations used for each problem instance

	Format	OMP_NUM_THREADS	MKL_NUM_THREADS
gisette_1000	dense	64	1
E2006_1000	sparse	64	-
usps	dense	32	32

Table 3 presents the execution times achieved by the original LIBSVM code and the proposed implementation using the best configuration for each problem instance. For each instance the minimum (best), average, and standard deviation are presented with the following format: mean ± std (min). All times are expressed in seconds. Additionally, the average acceleration achieved is presented for each instance. The average acceleration is computed as the ratio between the average execution time of LIBSVM and the average execution time of the proposed implementation.

Results in Table 3 show that the proposed implementation is able to efficiently improve the training time while computing exactly the same results than the original LIBSVM. The proposed implementation achieves an acceleration of up to 4.81x on average on the *usps* instance. These training time improvements are significant, specially when considering larger training datasets that would otherwise be intractable for sequential SVM implementations.

Table 3. Execution time in seconds (mean ± std (min)) and average acceleration of the proposed approach against the original LIBSVM

	LIBSVM	Best configuration	$\overline{\text{acceleration}}$
gisette_1000	22.66 ± 0.06 (22.59)	7.07 ± 0.02 (7.03)	3.21x
E2006_1000	20.59 ± 0.03 (20.56)	4.98 ± 0.02 (4.96)	4.13x
usps	18.46 ± 0.06 (18.24)	3.84 ± 0.01 (3.81)	4.81x

6 Conclusions and Future Work

This article presented a parallel implementation of the popular LIBSVM software, specifically adapted to the Intel®Xeon Phi™ architecture. The proposed implementation allow reducing the training time while computing the exact same results than the original LIBSVM. The modifications proposed include: coarse-grain parallelization of kernel evaluations, a new format for representing training vectors, the integration with Intel®MKL for kernel computation, and optimizations at compiling time to exploit the underlying architecture. The experimental evaluation was performed using three publicly available datasets, corresponding to different classification and regression problems, and with different characteristics in terms of number, size, and density of the training vectors.

The main lines of future work include exploring more techniques to further improve the optimization of LIBSVM code. Some of the possible techniques to explore include: aligning vectors in memory to make a better use of caches in Intel®Xeon Phi™ cores, improve affinity in the thread-to-core assignment, and optimize other parts of the LIBSVM code that could benefit from the manycore architecture. Additionally, it will be interesting to generalize the proposed approach to other kernels and evaluate the results over larger learning problems, with bigger and denser training vectors. Finally, a tool to help users decide the best vector representation format and number of threads to assign to each level of parallelism should be developed.

Acknowledgement. The work of R. Massobrio and S. Nesmachnow was partly supported by PEDECIBA and ANII, Uruguay. R. Massobrio would like to thank ANII, Uruguay and Fundación Carolina, Spain. B. Dorronsoro would like to acknowledge the Spanish MINECO-FEDER for the support provided under contracts TIN2014-60844-R (the SAVANT project) and RYC-2013-13355.

References

1. Athanasopoulos, A., Dimou, A., Mezaris, V., Kompatsiaris, I.: GPU acceleration for support vector machines. In: Proceedings of the 12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS) (2011)
2. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT 1992, pp. 144–152. ACM, New York (1992)

3. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. ACM Trans. Intell. Syst. Technol. **2**, 27:1–27:27 (2011). http://www.csie.ntu.edu.tw/~cjlin/libsvm
4. Chang, C.C., Lin, C.J.: LIBSVM FAQ (2015). Accessed 14 July 2017. http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html#f432
5. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The Sunway TaihuLight supercomputer: system and applications. Sci. China Inf. Sci. **59**(7), 072001 (2016)
6. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. SIGPLAN Not. **17**(6), 120–126 (1982)
7. Guyon, I., Gunn, S., Hur, A.B., Dror, G.: Result analysis of the NIPS 2003 feature selection challenge. In: Proceedings of the 17th International Conference on Neural Information Processing Systems, NIPS 2004, pp. 545–552. MIT Press, Cambridge (2004)
8. Hsu, C.W., Chang, C.C., Lin, C.J.: A practical guide to support vector classification (2003). Accessed 14 July 2017. https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf
9. Hull, J.J.: A database for handwritten text recognition research. IEEE Trans. Pattern Anal. Mach. Intell. **16**(5), 550–554 (1994)
10. Intel®Software: Intel®Math Kernel Library Link Line Advisor (2017). Accessed 14 July 2017. https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor
11. Ivanciuc, O.: Applications of Support Vector Machines in Chemistry, pp. 291–400. Wiley, Hoboken (2007)
12. Kogan, S., Levin, D., Routledge, B.R., Sagi, J.S., Smith, N.A.: Predicting risk from financial reports with regression. In: Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, NAACL 2009, pp. 272–280. Association for Computational Linguistics, Stroudsburg (2009)
13. Lichman, M.: UCI machine learning repository (2013). Accessed 14 July 2017. http://archive.ics.uci.edu/ml
14. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights landing: second-generation Intel Xeon Phi product. IEEE Micro **36**(2), 34–46 (2016)
15. TOP500.org: Top500 List - June 2017 (2017). Accessed 14 July 2017. https://www.top500.org/list/2017/06/
16. Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel math kernel library. High-Performance Computing on the Intel® Xeon Phi™, pp. 167–188. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06486-4_7
17. You, Y., Song, S.L., Fu, H., Marquez, A., Dehnavi, M.M., Barker, K., Cameron, K.W., Randles, A.P., Yang, G.: MIC-SVM: designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 809–818 (2014)

Performance Improvements of a Parallel Multithreading Self-gravity Algorithm

Nestor Rocchetti[1(✉)], Daniel Frascarelli[1], Sergio Nesmachnow[1], and Gonzalo Tancredi[2]

[1] Facultad de Ingeniería, Universidad de la República, Herrera y Reissig 565, 11300 Montevideo, Uruguay
{nrocchetti,sergion}@fing.edu.uy, dsanfra@gmail.com

[2] Facultad de Ciencias, Universidad de la República, Iguá 4225, 11400 Montevideo, Uruguay
gonzalo@fisica.edu.uy

Abstract. This article presents the application of performance optimization techniques to improve the computational efficiency of a parallel multithreading algorithm for self-gravity calculation on agglomerates. The studied algorithm applies the Discrete Element Method to simulate an ensemble of interacting particles under several contact and body forces. Based on the time scales of the process involved in the problem, we used a computation algorithm that speed up the self-gravity calculation based on defining a mesh over the simulated space. Specific performance improvements are presented, including the update of the occupied regions of the space, profiling and reimplementation of the most time consuming routines. Results indicate that the proposed implementation scale appropriately (almost-linear behavior) with the number of computational resources and the number of particles. The proposed improvements allow accelerating up to 50× the execution times over the previous version of the self-gravity algorithm in the studied scenarios.

1 Introduction

Gravitational potential is the main force that holds together astronomical objects, including agglomerates of small particles such as comets and asteroids [1]. Small particles that conform the agglomerates (called *grains*) are affected by short range interactions (e.g., contact forces) and also by long range interactions (e.g., self-gravity) [2,3]. Self-gravity interactions keep the particles together, creating a so-called rubble-pile asteroid or comet. Also, self-gravity gives shape to this type of celestial objects [4]. In addition, long-range interactions allow space bodies to interact causing mutual attraction and deformations.

The motion and interactions of rubble-pile asteroids and comets is a matter of interest for astronomers. In order to study the evolution of these systems, researchers have to solve numerically the equations of motion of the particles in the agglomerate. Given an agglomerate of particles, if the interaction between every pair of particles must be calculated, the computational cost is $O(N^2)$.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 291–306, 2018.
https://doi.org/10.1007/978-3-319-73353-1_21

Besides, if the agglomerate has millions of particles it turns computationally difficult to study the behavior of those big systems for extended periods of time.

High Performance Computing (HPC) is a paradigm that helps researchers to solve complex problems and perform simulations on big domains. HPC allows dealing with complex problems that demand high computer power, while complying with the time restrictions imposed by todays scientific standards. Instead of using a single computing resource, HPC proposes using multiple resources simultaneously applying a coordinated approach. This way, a cooperation strategy is implemented, allowing the workload to be divided between the computational units available to solve a complex problem in reasonable execution times [5].

Discrete Element Method (DEM) [6] is a numerical method used to simulate the interaction of particles. It was created to simulate short-range interactions between particles. The idea was to simulate physical phenomena by dividing the problem in particles and defining their properties and interactions between them. Given the nature of the problem, it is possible to improve its performance applying parallel programming techniques. ESyS-Particle [7] is a software for simulating geological phenomena (e.g.: rock fragmentation, failure opening, and granular flows) that implements the DEM method. ESyS-Particle includes parallel programming techniques that allow its execution on parallel/distributed high end environments, such as clusters or grids. Given that the contact forces can be calculated separately for each particle, the system is parallelized by a static spatial domain division. The first applications of ESyS-Particle in planetary sciences was presented by our research group [8]. Several surface processes in low-gravity environments, such as on the surface of asteroids and comets, were studied, and new models to simulate contact forces were introduced.

A typical shortcoming of most DEM codes, in particular ESyS-Particle, is the lack of models to simulate long-range forces like gravity among particles (i.e., self-gravity). This is a relevant problem in the case of asteroids and comets, which are in many cases agglomerates of rocks with a certain size distribution. In order to incorporate a self-gravity module into ESyS-Particle, we developed a program to compute the gravity interactions between the grains of the agglomerate [9]. It was implemented using parallel programming, allowing to perform simulations of thousands of particles efficiently, by exploiting the resources available at HPC infrastructures. The main details of the computational approach were presented in [9]. Later [10], we described several strategies for designing an efficient and accurate parallel multithreading algorithm for the self-gravity computation.

This article proposes specific performance improvements of the self-gravity algorithm introduced in [10]. The improvements consist of updating the self-gravity specifically on the sectors of the simulated space that are occupied by particles. Also, a profiling of the implementation was performed after the improvements were included to help identifying the most time consuming routines. Then, appropriate changes were implemented to mitigate the impact of those time consuming routines. The main goal of the improvements is to achieve lower execution times compared to the previous version, in which the update was calculated on every node of the simulated space, therefore allowing

ESyS-Particle simulations with self-gravity forces between the particles to scale to a higher number of particles.

The article is organized as follows. Section 2 reviews related works on domain decomposition for particle simulators and describes the parallel self-gravity implementation on ESyS-Particle. Section 3 describes the proposed modifications to improve the performance of the self-gravity simulations. Section 4 reports the experimental analysis focusing on the computational efficiency of the proposed implementation. Finally, the conclusions are presented in Sect. 5.

2 Spatial Domain Decomposition Techniques and the Parallel Self-gravity Implementation on ESyS-Particle

This section introduces static and dynamic spatial domain decomposition techniques and the parallel self-gravity implementation on ESyS-Particle. A review of related work on spatial domain decomposition techniques is also presented.

2.1 Spatial Domain Decomposition Techniques

Domain decomposition techniques are applied to speed up the calculation of interactions between particles, in order to reduce the $O(N^2)$ complexity order of a straightforward particle-by-particle algorithm.

Static domain decomposition techniques are based on a decomposition that remains invariant during the simulation. A typical example is implemented in the DEM simulator for soft spheres by Sanchez and Scheeres [11]. The same static domain decomposition technique is used to calculate short range interactions (e.g., contact forces) as well as long range interactions (e.g., gravitational forces). The space is divided in cubical cells that are bigger than the particles simulated. Then, to calculate the gravitational potential, instead of considering individual particles, the whole cell is considered as a particle. The calculations for N particles are still $O(N^2)$, but the authors claimed that the amount of calculations required decreased in one order of magnitude. The short range interactions and the long range interactions are computed concurrently. However, the authors did not propose a parallel implementation. Thus, simulations of systems with only up to $8,000$ particles were executed.

Dynamic domain decomposition techniques use adaptable structures or reconstructed from scratch during the simulation. A classic technique in this field is the one proposed by Barnes and Hut [12]. In this method, the simulated domain is divided in a hierarchical tree to accelerate the calculation of the gravitational potential in a N-Body simulation. The root node of the tree represents the complete space of the simulation. If the space that a node represents has more than one particle, the space is divided into smaller pieces. In a two-dimension simulation, the space is divided in four smaller pieces. However, in a three-dimension simulation, the space is divided into eight pieces. Applying this domain decomposition the authors affirm that the long-range interactions are calculated in $O(N \log N)$, being N the number of particles in the domain.

2.2 A Parallel Algorithm for Self-gravity Calculation

The gravitational potential vector of a particle must be computed on each timestep of a simulation. The mathematical model for computing the gravitational potential as the gravitational interaction with the rest of the particles in the studied system [9] is expressed in Eq. 1, where G is the gravitational constant, $\|\overrightarrow{r}\|$ is the norm of vector $\overrightarrow{r}$, M_i is the mass of the particle, and V_j is the gravitational potential of particle j.

$$V_j = \sum_{i \neq j} \frac{GM_i}{\|\overrightarrow{r_j} - \overrightarrow{r_i}\|} \tag{1}$$

A straightforward implementation of the gravitational potential calculation according to Eq. 1 results in a computational cost of $O(N^2)$ for each particle in each time step. This approach turns to be inefficient when the simulation scenario scales to hundreds of thousands of particles.

In order to overcome that efficiency problem, our previous work [9] proposed a state-of-the-art multi-threading algorithm for self-gravitating agglomerates. A hierarchical grouping approximation method (*Mass Approximation Distance Algorithm*, MADA) was introduced to improve the performance of the calculation of the gravitational potential of a particle in a given timestep, by considering a group of distant particles as a single big particle located in the center of mass of the group. Our proposed parallel algorithm calculates the self-gravity of million-particle objects using MADA and a pool of worker threads that execute the most compute-intensive tasks in parallel sections. The algorithm scales linearly with the number of particles in the system, and it scales with an inverse power law (exponent 0.87) with the number of threads used in the computation. The observed speedup is close to linear for systems containing up to 2×10^5 particles.

2.3 Self-gravity Implementation on ESyS-Particle

A detailed description of the self-gravity implementation into ESyS-Particle is out of the scope of this paper. The main implementation issues are outlined next.

In a DEM simulation that includes self-gravity, two type of forces acting on the particles have to be considered: short-range contact forces and the long-range gravity force. The typical time scales for these interactions are generally very different; the contact forces have a timescale on the order of the duration of the contact, while the variation of the gravity force depends on the velocity of the particles. For low-velocity displacements, the difference in timescales could be many order of magnitudes. Taking into account these considerations, the computation scheme in the self-gravity module implemented into ESyS-Particle applies the following steps: (1) compute the gravity acceleration field in a grid of nodes enclosing the ensemble of particles; (2) for every particle at each time step, compute the contact forces over the particle and interpolate the value of the acceleration for the location of the particle using the values of the acceleration on the surrounding nodes; (3) apply the forces and advance the systems; (4) if a

large displacement of the particles is found, the gravity field is updated; if not, the previous gravity field is used for the next time step and (2) is executed again.

The previously commented procedure requires an efficient method to compute the acceleration on the nodes, because the number of operations for N particles and M nodes is of the order of $O(N \times M)$. ESyS-Particle implements spatial domain decomposition using a master-worker model implemented on Message Passing Interface (MPI) [13]. The master process provides a high level of control: the workers execute the activities assigned by the manager, and the manager node is in charge of the communications between them. Regarding the workload distribution, ESyS-Particle requires a vector (d_x, d_y, d_z) to determine the way the domain is divided among the worker processes. When executing a simulation, $d_x \times d_y \times d_z$ worker process are created, and each coordinate d_i in the vector indicates the number of subdivisions in the i-direction. Weatherley et al. [14] presented a benchmarking of ESyS-Particle to analyze the scalability and the accuracy of the calculations. Regarding the self-gravity algorithm implementation, it also applies a master-worker model but includes a two-level parallelization scheme implemented with multithreading. Thus, two different implementations (distributed memory and shared memory) are included in the proposed model.

Before starting a simulation, the self-gravity module builds and overlays a grid to divide the spatial domain of the simulation. The grid is composed of boxes, whose vertexes are called *nodes*. The number of nodes and their location are defined depending on the spatial domain and the size of the boxes. The acceleration along the x-axis on a node located at position (x, y, z) due to an ensemble of N particles of individual mass m_j and positions (x_j, y_j, z_j) is given by Eq. 2. Similar equations are formulated for the acceleration along y and z-axis.

$$a_x(x, y, z; x_j, y_j, z_j, j = 1...N) = \sum_{j=1,N} Gm_j \frac{x_j - x}{\|\overrightarrow{r_j} - \overrightarrow{r}\|^3} \tag{2}$$

In order to get the value of the acceleration on a particle in a future time step, the closest eight nodes surrounding the particle (see Fig. 1) must be determined. Once the values of the acceleration in those nodes is known, interpolation can be applied to obtain the acceleration on the particle position.

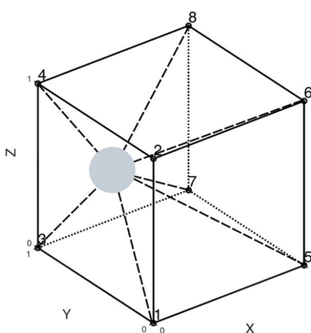


Fig. 1. Box of the self-gravity grid with nodes numbered.

3 Improvements of the Baseline Implementation

This section describes the strategies to speed up the self-gravity calculation.

3.1 Reducing the Execution Time of the Self-gravity Computation

The previous implementation [9] updated the gravity acceleration field on every node of the self-gravity grid. However, computing the acceleration of each particle only requires the values of the acceleration on the occupied nodes. Therefore, a scheme in which only the occupied nodes are eligible to be distributed over the resources available to compute the self-gravity on particles is proposed. The new scheme improves the utilization of the computational resources available, thus accelerating the self-gravity computations and the simulations.

The self-gravity field is updated not on each ESyS-Particle time step, but when the average displacement of all particles (from the last self-gravity update) is larger than a predefined distance. Thus, a particle can migrate to a different box to where it was when the acceleration on the nodes was previously updated. On this scenario, the eight surrounding nodes needed to interpolate the acceleration in the particle position are different from the initial node. In order to guarantee that the acceleration is updated on all the nodes involved in the interpolation, the surrounding 64 nodes of the grid are considered for the update. Figure 2 shows the 64-node surrounding box for a particle.

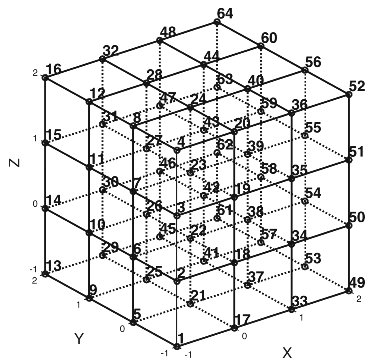


Fig. 2. The expanded 64 surrounding nodes.

In order to build the list of nodes where the self-gravity needs to be updated, the first step is retrieving from the ESyS-Particle context the list of occupied nodes for the current step of simulation. After that, for each node in this list, a 64-node expansion is performed, as shown in Fig. 2. The acceleration is only updated on the nodes of the list plus the nodes resulting from the expansion.

After that, each node in the list is assigned in a first-come first-serve manner to each self-gravity thread. Load balancing is implemented by assigning nodes on demand, thus accounting for different execution times of each calculation.

Finally, the updated acceleration value in each node is stored in memory to be available when the gravitational potential in the particles' position is calculated.

3.2 Profiling the Self-gravity Calculation

In order to analyze and identify bottlenecks in the performance of the self-gravity implementation (after including the improvements described in the previous subsection), a profiling was performed using the VTune Amplifier tool by Intel [15].

Figure 3 reports the results of the profiling performed to the self-gravity code before including the proposed improvements. The hot spots detected by the tool are considered to prioritize the modules to be improved. The efforts were focused on the nine routines with the larger execution times.

Routine	Time
BoxCoords::getZ	2269.250s
SharedMemory::getBox	1669.090s
BoxCoords::compare	1551.303s
Point::getX	875.999s
OnlyOccupiedCellsProcessingStrategy::getNextOrigin	704.426s
Box::getParticlesCount	648.715s
Particle::getCenter	570.394s
Calculus::calculateDeltaForceVectorUsingDifEcuations	508.503s
SharedMemoryManager::getBox	450.763s
std::vector<BoxCoords*, std::allocator<BoxCoords*>>::size	369.614s
__distance<std::_List_const_iterator<int> >	363.513s
[Import thunk BoxCoords::getZ]	304.461s
Calculus::getDistance	233.565s
StaticConfig::getInstance	207.853s
Calculus::getDistance	177.251s
Calculus::calculateForceVectorUsingDifEcuations	162.204s
[Import thunk BoxCoords::compare]	151.462s
SelfGravityProcessor::processOrigin	104.909s
std::_List_const_iterator<int>::operator++	98.790s
Point::getY	82.166s
ThreadExecutionManager::getProcessableBoxes	67.783s
[Import thunk StaticConfig::getInstance]	63.363s
Calculus::calculateForceVector	57.665s

Fig. 3. Profile report of the code.

According to Fig. 3, the most time consuming routine is `BoxCoords::getZ`, the routine that retrieves the z axis of the position for each particle. This routine is time consuming because it is called in every update of the acceleration for each node. In addition, the routine is also affected by memory management. Due to the number of particles involved, the total memory required to maintain these structures is usually larger than the CPU cache. This requires the CPU cache to be cleared and filled as different nodes are processed. Thus, this is the first routine executed (in the self-gravity context) when accessing a node to process. The time required to transfer data from the main memory also increases the total time spent by the routine. A similar behavior is detected for routines `Point::getX`, `SharedMemory::getBox`, `BoxCoords::compare`, `Box::getParticleCount`, and `Particle::getCenter`. The same arguments hold in these cases.

The fifth routine identified to consume the most execution time is `OnlyOccupiedCellsProcessingStrategy::getNextOrigin`. This routine is used to find the coordinates of the next node where the acceleration needs to be updated, by iterating over all the nodes of the simulation domain. Finally, `calculateDeltaForceVectorUsingDifEcuations` was identified as a time consuming routine. However, the routine implements a simple mathematical equation, with less space to achieve optimizations than the other routines analyzed. Because of this, it was not selected as a routine to improve.

After identifying the hot spots in a simulation, specific changes were introduced on the core components of the self-gravity module to improve performance, specifically in the component that updates the value of the acceleration of the nodes. In the acceleration update, instead of iterating over all the nodes of the self-gravity grid, routines were adapted to process the list of occupied nodes and the 64-node surrounding box. This way, the self-gravity acceleration update is optimized by reducing the number of times that each routine is called. This change affects all the routines that were chosen to be optimized.

4 Experimental Evaluation

This section reports the performance evaluation of the self-gravity module after implementing the performance improvements. A description of the test scenario and instances is presented, and efficiency results are reported and discussed.

4.1 Description of the Test Scenario and Instances

The test scenario consists of two identical agglomerates of particles. One agglomerate is created using the GenGeo package included in ESyS-Particle and the other is identical but the positions of the particles are central mirrored. The initial speed vector for all particles in the agglomerate is perpendicular to the radius vector of the each center of mass, with the same magnitude but in opposite direction. The contact forces among the particles are pure elastic forces. Under such conditions, both agglomerates move in a circular orbit with respect to the center of mass of them. The initial velocity of the agglomerates for every instance is 5 m/s, and should remain invariant during the simulation. The density of the individual particles is $3000\,\mathrm{g/cm^3}$.

Three instances of the two-agglomerate scenario were defined to study the computational efficiency of the proposed implementation. A small instance with 3,866 particles and radii from 50 m to 100 m, a medium instance of 11,100 particles and radii from 35 m to 70 m, and a big instance of 38,358 particles and radii from of 20 m to 60 m. The total mass of the agglomerates is not equal for all the instances, it goes from 1.2×10^{12} kg to 1.7×10^{12} kg. Nevertheless, the masses of the agglomerates have the same order of magnitude.

All instances were simulated for 100,000 time steps. To evaluate speedup and scalability, simulations were executed with different number of computational resources assigned to perform the computations. In all instances, particles

move at the same initial velocity. So, instances with smaller particles have more updates of the self-gravity than instances with bigger particles. For this reason, simulations with more particles take longer to end.

The experimental evaluation was performed on a Dell PowerEdge M620 Server (Intel Xeon E5-2680 processor at 2.50 GHz, 24 cores and 32 GB RAM) from Cluster FING, the High Performance Computing platform from Universidad de la República, Uruguay [16].

4.2 Profiling of the Optimized Version

Figure 4 shows the results of the profiling of the self-gravity calculation after the improvements were implemented and Table 1 reports the execution time of the most time consuming routines before and after the improvements.

Routine	Time
Point::getX	109.754s
Calculus::calculateDeltaForceVectorUsingDifEcuations	91.308s
Particle::getCenter	79.985s
__distance<std::_List_const_iterator<int> >	72.431s
Calculus::getDistance	32.890s
Calculus::calculateForceVectorUsingDifEcuations	31.211s
Calculus::getDistance	27.228s
std::_List_const_iterator<int>::operator++	19.717s
Point::getY	16.248s
StaticConfig::getInstance	11.134s
Point::getZ	9.483s
SharedMemoryManager::getBox	8.568s

Fig. 4. Profiling of ESyS-Particle with self-gravity, after the improvements.

Table 1. Comparison of the most time consuming routines before and after implementing the performance improvements.

Routine	Execution time (s)	
	Before	After
`BoxCoords::getZ`	2269	Negligible
`SharedMemory::getBox`	1669	Negligible
`BoxCoords::compare`	1551	Negligible
`Point::getX`	876	109
`:: getNextOrigin`	704	Negligible
`Box::getParticlesCount`	649	Negligible
`Particle::getCenter`	570	80
`::calculateDeltaForceVectorUsingDifEcuations`	509	91
`SharedMemoryManager::getBox`	451	9
`std::vector<>::size`	370	Negligible

Results in Table 1 indicate that the method `BoxCoords::getZ`, which consumed 2269 s in the original version, has a negligible contribution to the execution time in the improved version. A similar behavior is identified for routines `SharedMemory::getBox`, `BoxCoords::compare`, `SharedMemoryManager::getBox`, and `Box::getParticlesCount`. The modifications in the routine that searches for the next node to process (`::getNextOrigin`) improves the execution time from 704 s to a negligible time compared to the other routines. Finally, improvements on routine `Point:getX` allows it to execute 8 times faster (from 876 s to 109 s).

4.3 Performance Evaluation Results

Small size instance. Table 2 shows the results of the performance evaluation tests for the small instance with 3866 particles. The table reports the configuration of processes and threads used for execution, the execution time in seconds, the percentage of the overall execution time spent on calculating self-gravity, the number of self-gravity updates performed, and the average time spent in self-gravity calculations. The lowest execution time was 2.57×10^3 s, using two process for ESyS-Particle and two threads for self-gravity calculation. In the small instance, the lowest percentage of time calculating self-gravity was 74%, when using a configuration with one process assigned to ESyS-Particle and two threads assigned to the self-gravity module. These values indicate that there is more space for improving the performance in the self-gravity module than in the ESyS-Particle core. Finally, all configurations performed 456 updates.

Table 2. Performance results for the two agglomerate scenario with 3,866 particles (small instance).

# particle processes	# gravity threads	Execution time (s)	Time computing self-gravity	# self-gravity updates	Avg. self-gravity time (s)
1	1	3.98×10^3	82%	456	7.22
1	2	2.79×10^3	74%	456	4.57
2	1	3.71×10^3	84%	456	6.87
2	2	2.57×10^3	76%	456	4.33

Medium size instance. Table 3 shows the experimental results of the execution for the medium size instance with 11,100 particles. The configuration with two processes and four threads had the lowest execution time with 1.47×10^4 s. In this instance, the lower value for the percentage of time spent on self-gravity calculation is 84%, which is 10% higher than on the small instance. The number of gravitational force interactions grows in a upper-linear manner compared to the linear growth of the contact forces, which causes the increase in the time calculating the self-gravity. The best average update of self-gravity is of 17.79 s, using the configuration with two processes and four threads.

Table 3. Performance results for the two agglomerate scenario with 11,100 particles (medium instance).

# particle processes	# gravity threads	Execution time (s)	Time computing self-gravity	# self-gravity updates	Avg. self-gravity time (s)
1	1	3.20×10^4	93	703	42.61
1	2	2.19×10^4	89	703	27.88
1	4	1.53×10^4	84	703	18.35
2	1	3.39×10^4	94	705	45.19
2	2	2.15×10^4	90	705	27.68
2	4	1.47×10^4	85	705	17.79

Large size instance. Table 4 reports the performance results for the large instance with 38,358 particles. The lowest execution time was 5.26×10^4 s, using the configuration with 8 processes and 16 threads. The percentage of time spent on gravity calculations varies from 89% to 98%, which means that most of the execution time is spent on self-gravity calculations rather than on contact forces calculation.

Table 4. Performance results for the two-agglomerate scenario with 38,538 particles (large instance).

# particle processes	# gravity threads	Execution time (s)	Time computing self-gravity	# self-gravity updates	Avg. self-gravity time (s)
1	1	2.58×10^5	95%	1293	191.25
1	2	1.85×10^5	93%	1412	121.53
1	4	1.19×10^5	89%	1412	74.98
2	1	2.46×10^5	96%	1417	168.01
2	2	1.86×10^5	95%	1417	124.94
2	4	1.25×10^5	91%	1417	80.69
4	1	3.20×10^5	98%	1434	218.33
4	2	2.24×10^5	97%	1434	152.15
4	4	1.39×10^5	95%	1434	92.34
4	8	9.10×10^4	94%	1434	59.94
4	16	5.93×10^4	90%	1435	37.30
8	4	1.28×10^5	96%	1411	86.88
8	8	7.67×10^4	92%	1411	50.28
8	16	5.26×10^4	92%	1432	33.96
16	4	1.72×10^5	97%	1411	118.19
16	8	6.63×10^4	94%	1411	44.09

Overall comparison. In the small instance, self-gravity was updated 456 times, whereas on the medium instance it was updated from 703 to 705 times and on the large instance it was updated between 1293 and 1435 times. The number of gravity updates is larger when the size of the boxes of the mesh increases.

Execution times results indicate that only varying the number of processes assigned to ESyS-Particle, without varying the number of threads assigned to the self-gravity calculation, allows achieving a small improving the overall execution time of simulations. Doubling the number of processes only accounts for an execution time reduction of 7.34% for the small instance, while for the medium instance is 1.92%, and for the large instance is 5.88%.

The most significant improvements are obtained when increasing the number of threads assigned to the self-gravity calculation. Improvements of 31% are obtained for the small instance, ranging from 52% to 57% for the medium instance, and 54% to 61% for the large instance. This situation is directly related to the time spent in self-gravity updates: 74% to 84% of the time for the small instance, 84% to 94% for the medium instance and 89% to 98% for the large instance. Thus, the larger performance improvements are obtained when increasing the number of threads to perform the most time consuming operations.

In order to further analyze the self-gravity execution time improvements, Table 5 reports the acceleration values when using different configurations (number of parallel processes and threads) for the three problem instances studied. The acceleration metric is defined as the ratio of the execution time of a simulation when using a baseline configuration with a single process for ESyS-Particle and a single thread for self-gravity calculation and the execution time of the configuration using n_P processes for ESyS-Particle and n_T threads for self-gravity.

The best acceleration for the small instance was 1.55, obtained with the configuration using 2 processes and 2 threads. For the medium instance, the best acceleration was 2.17, using 2 processes and 4 threads. Finally, the best acceleration for the large instance was 4.90, computed with the configuration using 8 processes and 16 threads. This was the largest acceleration value obtained when varying the number of processes and threads for the studied problem instances.

Results show that in every instance, when increasing the number of processes for a fixed number of threads, the acceleration stagnates. The reason of this stagnation is the time spent on ESyS-Particle during a simulation, which is in average 21.0% for the small instance, 10.8% for the medium instance, and 5.9% for the large instance. Thus, the parallel algorithm cannot take advantage of using more computing resources due to the limited computing demands of those tasks not related with self-gravity computation.

Figure 5 presents a graphical comparison of the relative execution time reduction over the baseline configuration (one process, one thread) for the studied configurations on the large instance. Results are grouped by the number of processes defined for those configurations, which are shown in different colors. The growth of the execution time reduction is clearly shown for the configurations that use different number of threads for the same number of processes. Results show that execution times reduces following a linear tendency. The higher execution time reduction was 79%, using the configuration with 8 processes and 16

Table 5. Acceleration values for the two agglomerate scenario

# particle module processes	# gravity module threads	Acceleration
Small instance (3,866 particles)		
1	1	1.000
1	2	1.429
2	1	1.072
2	2	1.549
Medium instance (11,100 particles)		
1	1	1.000
1	2	1.462
1	4	2.095
2	1	0.944
2	2	1.486
2	4	2.170
Large instance (38,358 particles)		
1	1	1.000
1	2	1.395
1	4	2.168
2	1	1.049
2	2	1.389
2	4	2.059
4	1	0.806
4	2	1.150
4	4	1.856
4	8	2.837
4	16	4.356
8	4	2.020
8	8	3.366
8	16	**4.906**
16	4	1.501
16	8	3.891

threads. Nevertheless, similar results were achieved with the configuration using 4 processes and 16 threads, which accounted for an execution time reduction of 77%. On the other hand, Fig. 5 also clearly shows that increasing the number of processes does not improve the performance significantly: the configuration using 2 processes and 1 thread had 5% of reduction, and for the configuration using 4 processes and 1 thread no reduction was computed.

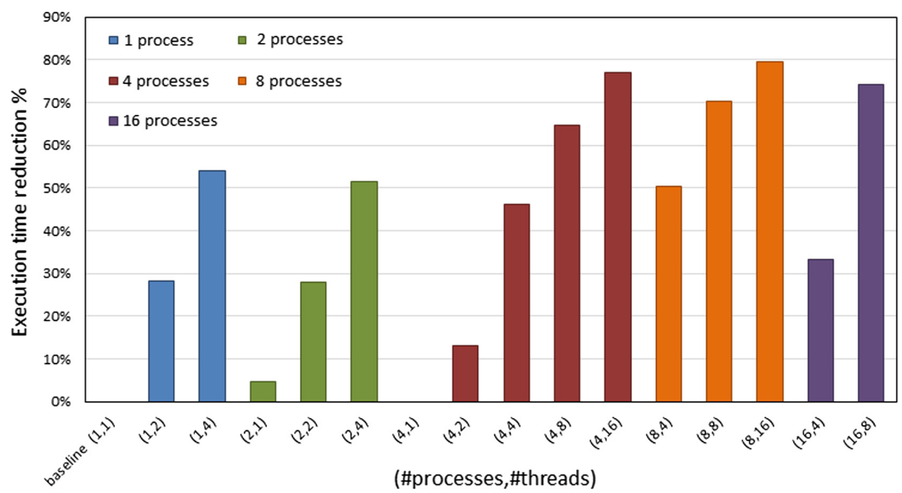


Fig. 5. Execution time reductions for a self-gravity simulation on the large instance.

Overall, after including all the proposed performance improvements, the optimized ESyS+self-gravity version achieved an acceleration factor of more than **50**× when compared with the original (non-optimized version), on simulations performed on all studied scenarios. This performance improvement allows scaling up to perform realistic simulations with a large number of particles (from 100,000 to one million) is reasonable execution times.

5 Conclusions

This article presented strategies for performance improvement of a self-gravity algorithm for agglomerates, implemented over ESyS-Particle. The main modifications consisted of updating the acceleration of the occupied nodes of the grid, plus a buffering structure where nodes can move to in future time steps. After including that improvement, a profiling tool was used to identify hot spots in the implemented code and specific modifications were included in order to reduce the number of time each time consuming routine is called.

The experimental evaluation of the proposed modifications was performed over a realistic scenario consisting of two identical agglomerates orbiting each other. Three problem instances were defined considering 3,866 (small), 11,100 (medium), and 38,538 (large) particles. The two-level parallel model was studied, by assigning different computing resources to ESyS calculation processes and self-gravity calculation threads.

Experimental results show that, depending on the size of the problem instance, 74% to 98% of the execution time of the simulations is spent on self-gravity calculation. As a consequence, the best performance improvements are obtained when assigning more computational resources to the threads that

compute the self-gravity in parallel than to the processes that perform the ESyS-Particle calculations.

Regarding the performance of the self-gravity simulations, the higher acceleration with respect to a baseline configuration using only one process and one thread was 4.9×, achieved with the configuration using 8 processes for ESyS-Particle and 16 threads for self-gravity calculations. Overall, the optimized version of ESyS including self-gravity allowed improving the performance in a factor of 50×, when compared to the previous version without optimizations. This performance improvements are obtained without affecting the numerical results of the simulations. Thus, they allow scaling up to perform simulations with larger numbers of particles in realistic execution times.

The main lines for future work include extending the performance evaluation of the implemented self-gravity simulations to consider larger problem instances and different scenarios. In addition, we are currently working on developing a comprehensive benchmark for performance comparison with other self-gravity simulators proposed in related works.

Acknowledgments. The work of Néstor Rocchetti, Sergio Nesmachnow, and Gonzalo Tancredi has been partly supported by CSIC, ANII, and PEDECIBA (Uruguay).

References

1. Sánchez, P., Scheeres, D.: The strength of regolith and rubble pile asteroids. Meteorit. Planet. Sci. **49**(5), 788–811 (2014)
2. Harris, A., Fahnestock, E., Pravec, P.: On the shapes and spins of "rubble pile" asteroids. Icarus **199**(2), 310–318 (2009)
3. Fujiwara, A., Kawaguchi, J., Yeomans, D., Abe, M., Mukai, T., Okada, T., Saito, J., Yano, H., Yoshikawa, M., Scheeres, D., et al.: The rubble-pile asteroid Itokawa as observed by Hayabusa. Science **312**(5778), 1330–1334 (2006)
4. Rozitis, B., MacLennan, E., Emery, J.: Cohesive forces prevent the rotational breakup of rubble-pile asteroid (29075) 1950 DA. Nature **512**(7513), 174–176 (2014)
5. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2010)
6. Cundall, P., Strack, O.: A discrete numerical model for granular assemblies. Geotechnique **29**(1), 47–65 (1979)
7. Abe, S., Altinay, C., Boros, V., Hancock, W., Latham, S., Mora, P., Place, D., Petterson, W., Wang, Y., Weatherley, D.: ESyS-Particle: HPC Discrete Element Modeling Software. Open Software License version 3 (2009)
8. Tancredi, G., Maciel, A., Heredia, L., Richeri, P., Nesmachnow, S.: Granular physics in low-gravity environments using discrete element method. MNRAS **420**, 3368–3380 (2012)
9. Frascarelli, D., Nesmachnow, S., Tancredi, G.: High-performance computing of self-gravity for small solar system bodies. Computer **47**(9), 34–39 (2014)
10. Nesmachnow, S., Frascarelli, D., Tancredi, G.: A parallel multithreading algorithm for self-gravity calculation on agglomerates. In: Gitler, I., Klapp, J. (eds.) ISUM 2015. CCIS, vol. 595, pp. 311–325. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32243-8_22

11. Sánchez, P., Scheeres, D.: Dem simulation of rotation-induced reshaping and disruption of rubble-pile asteroids. Icarus **218**(2), 876–894 (2012)
12. Barnes, J., Hut, P.: A hierarchical O(N log N) force-calculation algorithm. Nature **324**(6096), 446–449 (1986)
13. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press, Cambridge (1999)
14. Weatherley, D., Boros, V., Hancock, W., Abe, S.: Scaling benchmark of ESyS-particle for elastic wave propagation simulations. In: IEEE Sixth International Conference on e-Science, pp. 277–283. IEEE (2010)
15. Intel® vtune™ amplifier 2017 (2006). https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed July 2017
16. Nesmachnow, S.: Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República. Revista de la Asociación de Ingenieros del Uruguay **61**(1), 12–15 (2010)

A Fast GPU Convolution/Superposition Method for Radiotherapy Dose Calculation

Diego Carrasco[1(✉)], Pablo Cappagli[2], and Flavio D. Colavecchia[1,3]

[1] Laboratorio de Física Médica Computacional, Centro Integral de Medicina Nuclear y Radioterapia de Bariloche, Comisión Nacional de Energía Atómica, Buenos Aires, Argentina
diego.carrasco@cab.cnea.gov.ar

[2] Centro Atómico Bariloche, Comisión Nacional de Energía Atómica, Buenos Aires, Argentina

[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Buenos Aires, Argentina

Abstract. An algorithm based on Convolution/Superposition with collapsed cone approximation was developed for radiotherapy dose calculation, reducing numerical complexity and enabling a high accuracy computation in a dense grid. By analyzing the specific integrals and reducing them into a ray tracing problem, we show that both calculation and data evaluations can be mapped to specific and optimized memories types in the GPU. Using constant memory and texture fetches in the algorithm, an 144X speedup is obtained compared to an equivalent multi-threaded CPU code, without precision loss. The developed software is the foundation for a high performance calculation system with a fidelity equivalent to commercial planning systems and with a few seconds of execution.

Keywords: GPU · Convolution · Superposition · Dose · Radiotherapy

1 Introduction

There is a variety of energy transport problems that can be cast into a convolution equation. Given a source of energy and a point distribution function (or *kernel*), the convolution equation describes the deposition of energy in any region of space as it spreads out from the source [1]. The convolution equation can be solved in the three-dimensional space by means of a Fourier transform, resulting in a complexity proportional to the space discretization. The problem arises when the kernel function of the Fourier transform is not invariant under translations, increasing the complexity in several orders. One of such cases is the calculation of energy deposition in external radiotherapy. In this situation, a linear accelerator generates a photon beam that interacts inside the body depositing energy within it [2]. Therefore, the photon beam represents the main source while the local particle transportation is described by a kernel [3,4]. Several types of algorithms have been developed to deal with this problem, which in turn gave rise to

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 307–318, 2018.
https://doi.org/10.1007/978-3-319-73353-1_22

many commercial software, allowing medical physicists to optimize and obtain with high precision the parameters needed to treat an oncological patient [5].

Among all these methods to obtain the therapeutic energy distribution, the Convolution/Superposition based algorithms account for the physical principles of ionization radiation by dividing the calculation in two terms. The first term is an energy transport function called TERMA (acronym for Total Energy Released per unit Mass) that represents the primary fluence generated from the linear accelerator that is attenuated inside the patient. This primary fluence consists of photons that have not yet interacted. The second term is a function that transports the energy spread by particles generated after interactions of the primary fluence with the tissues [6]. This second term is usually referred as secondary fluence, and is represented by a convolution kernel.

The complete calculation region is well defined by a dense three-dimensional partition depicting the mass density or the relative electron density. Such array is called tomography and represents a patient volumetric treatment section with a typical voxel size of 0.1 cm. The resulting calculation is represented as energy per unit mass and named dose.

2 Theory

Given that particle transportation depends strongly on the medium density, every calculation point or voxel generates a different energy propagation, i.e. a different kernel. To take into account these density variations in humans tissues, the energy transportation is subject to a transformation ruled by the *radiological path* [7]. Convolution/Superposition methods can therefore be defined as a three dimensional spatial convolution between the TERMA and a kernel, both corrected by means of the radiological path, allowing dose calculation to be employed in irregular fields, complex and heterogeneous materials, with a precision comparable to those of Monte Carlo solutions but reducing the computation time [8].

2.1 Radiological Path

The heterogeneities present in the volume of irradiation affect the primary and secondary fluence. The effect on the former is dependent only on the heterogeneity traversed, while on the later depends on the position and lateral variations [2].

Using the O'Connor theorem, a simple transformation that removes the heterogeneous dependence from the TERMA and kernel can be accomplished [2]:

$$d(\boldsymbol{r}, \boldsymbol{r_o}) = \int_{\boldsymbol{r_o}}^{\boldsymbol{r}} \eta(\boldsymbol{x}) d\boldsymbol{x} \quad (1)$$

where $\eta(\boldsymbol{x})$ is the relative electron density, and $d(\boldsymbol{r}, \boldsymbol{r_o})$ is the radiological path from position $\boldsymbol{r_o}$ to $\boldsymbol{r}$, defined as the relative electron density integral over such line. Equation (1) implies the construction of an optimized ray tracing algorithm for the complete energy calculation.

2.2 Convolution/Superposition

A dose calculation equation in a volume with mass density $\rho(\boldsymbol{r})$ is defined as [2]:

$$D(\boldsymbol{r}) = \frac{\eta(\boldsymbol{r})}{\rho(\boldsymbol{r})} \int T(\boldsymbol{s})\rho(\boldsymbol{s})\eta^2 h(d(\boldsymbol{r},\boldsymbol{s}))dV_s \tag{2}$$

In this representation, $\boldsymbol{s}$ is a point where free energy is distributed as secondary fluence (scattered photons, electrons and positrons) and $\boldsymbol{r}$ is the point of energy deposition. The TERMA function $T(\boldsymbol{s})$ is the free energy generated from the primary fluence (photons originated in the linear accelerator) interacting in a volume element dV_s with mass density $\rho(\boldsymbol{s})$. The kernel h represents the spatial energy deposition in water, scaled by the radiological path $d(\boldsymbol{r},\boldsymbol{s})$ and the electron relative density $\eta(\boldsymbol{r})$ to account for the heterogeneous media [9].

This dose equation is based on the primary fluence releasing energy at any point in the tomographic volume, and distributing it through a kernel that represents the transport and energy deposition of the secondary fluence in the media. This mode of abstraction allows the calculation to be based on a sense of convolution (see Fig. 1). However, given the heterogeneities present in the medium, the kernel varies point to point with the radiological path, making the superposition necessary. The numerical complexity can be estimated as $O(N^7)$, where a typical number of voxels in the tomography is $N^3 = 256^3$, resulting in an unfeasible time for clinical needs. Although GPU implementations of this dose calculation method were presented in the literature [10], opaque numerical approximation were included.

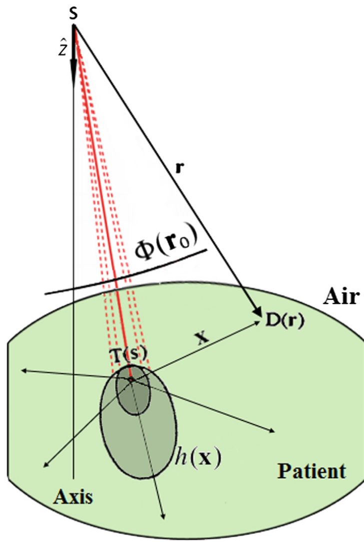


Fig. 1. Dose calculation representation. The TERMA $T(\boldsymbol{s})$ releases energy in the patient by a fluence $\Phi(\boldsymbol{r})$. The kernel $k(\boldsymbol{x})$ represents the spatial distribution of such energy.

2.3 Collapsed Cone Approximation

Although spherical coordinates can generate the best representation of a convolution kernel, the attenuation medium and irradiation volumes are characterized by a cartesian system defined in the tomography. Any transformations between both representations will certainly generate an error propagation, but maintaining separate systems would produce a slow and complex algorithm.

The energy transportation of a kernel can be simplified by supposing that all energy passing through consecutive spherical planes $x^2 d\Omega$ can be assigned to the direction $d\boldsymbol{\Omega}$ (see Fig. 2a and b). Therefore, all the energy associated with these divergent spherical sectors is collapsed and mapped to the corresponding cylinder's central axis (see Fig. 2c). This kind of transformation introduced by Ahnesjö removes all natural divergences from the secondary fluence and conserves energy [6]. With the collapsed cone approximation, (2) is simplified to a sum of unidimensional integrals accounting for every collapsed cone direction.

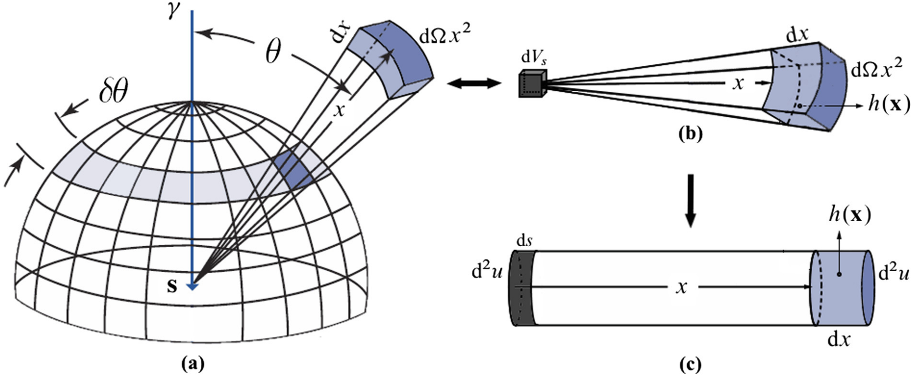


Fig. 2. Schematics of collapsed cone approximation. (a) Spatial distribution generated by the primary fluence interacting in ($\boldsymbol{s}$). (b) Energy transportation in specific cone direction $d\Omega$. (c) Spherical cone transportation is approximated and collapsed to a cylinder.

Therefore, the energy deposition kernel is collapsed to a discrete number of directions (θ_m, ϕ_n) that define the $\Omega^{m,n}$ solid angle [9]:

$$k^{m,n}(x) = \int_{\Omega^{m,n}} x^2 h(x) d\Omega. \tag{3}$$

With such transformation, all natural divergences are removed, and the calculation is transformed to a finite number of unidimensional transport problems, whose solution can be estimated with a convolution integral scaled by the radiological path. Although this approach seems to underestimate the energy deposition to small distances and overestimate it at great distances (comparable to the electron average free path), the conservation of radiant energy is compensated

by the adjacent dV_s volumes, as long as the variations of the local density are small enough [6].

With the collapsed cone approximation and the radiological path transformation, the dose equation represented by (2) is simplified to a linear problem [9]:

$$D^{m,n}(x) = \int_{s_o}^{x} T(s)\eta(s)k^{m,n}(d(x,s))ds \tag{4}$$

This means that every point in the tomography where dose needs to be computed can be resolved by means of a finite number of line integrals covering all the volume.

The discretization of such line integrals allows the generation of an efficient GPU code given a defined step sizes and finite number of them. Several optimization efforts have been made to reduce the complexity in (4), namely by reducing the amount of integrals used in the collapsed cone approximation [10] or by simplifying $k^{m,n}$ to a sum of exponentials [11]. Such simplifications result in faster calculations, but with increasing precision loss. In this paper the main effort is to improve the integral in (4) by means of an ray tracing algorithm mapped with specific GPU memory, without degrading precision.

3 The Parallel Collapsed Cone Kernel Algorithm

In this section we present the main steps to compute the dose with the Collapsed Cone Kernel (CCK) algorithm, and comment the details of the parallel implementation in CUDA capable GPUs.

3.1 Ray Tracing

As mentioned in the precedent sections, the relative electron density η and mass density ρ in the tomography are discretized in voxels of finite size defined by Δx, Δy and Δz. For every calculation line located in a partitioned space, the radiological path should be approximated by the sum of segments belonging to each voxel traversed, weighed by the local relative density.

To simplify the derivation of the algorithm, let us consider a two-dimensional tomography slice along the Z plane. We consider a ray l inside this plane and define the entering points (x_o, y_o) and the exit point (x_p, y_p) in a volume of interest (see Fig. 3). To compute the radiological path (1), it is necessary to compute each segment l_k and obtain the relative density [12].

To calculate every density section, the intersection between the transport ray l and the tomography default partition in $\boldsymbol{X}, \boldsymbol{Y}$ direction need to be accounted for (black points in Fig. 3). Let us define the coefficients (α_x, α_y) as:

$$\alpha_x = \left(\frac{\sqrt{(x_p - x_o)^2 + (y_p - y_o)^2}}{|x_p - x_o|} \right) \tag{5}$$

$$\alpha_y = \left(\frac{\sqrt{(x_p - x_o)^2 + (y_p - y_o)^2}}{|y_p - y_o|} \right) \tag{6}$$

From the starting point (x_o, y_o), the transportation line will traverse the tomography, intersecting with every $\boldsymbol{X}, \boldsymbol{Y}$ partition line in I or J steps, generating the sequence:

$$\Delta x_i = I * \Delta x, \qquad \Delta y_j = J * \Delta y \tag{7}$$

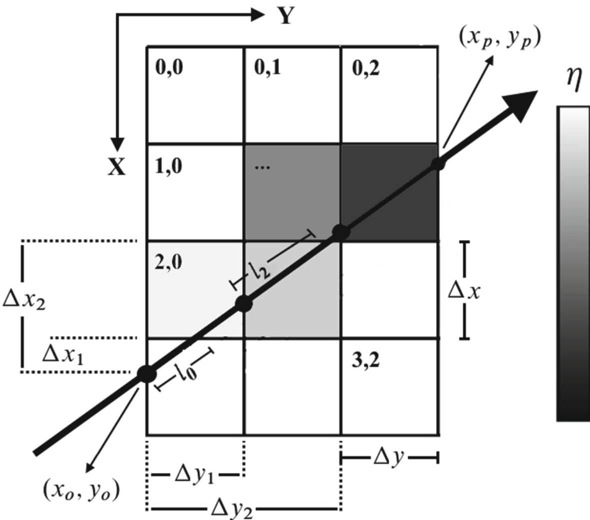


Fig. 3. Indexed density matrix associated with a patient tomography. A line l defined by entering (x_o, y_o) and exiting (x_p, y_p) points passes through different density values represented by the η gray bar. Each l_k segments is defined between two consecutive intersection between the l and $\boldsymbol{X}, \boldsymbol{Y}$ partitions. For every Δx increase in $\boldsymbol{X}$ direction, the line length increases by $\alpha_x \Delta x$.

We define the functions $f : \mathbb{N}^2 \to \mathbb{N}^2$ and $g : \mathbb{N}^2 \to \mathbb{R}$ according to the increasing (I, J) indexes as:

$$f(I,J) = \begin{cases} (I+1, J) & \text{if} \quad \alpha_x \Delta x_{i+1} < \alpha_y \Delta y_{j+1} \\ (I, J+1) & \text{if} \quad \alpha_x \Delta x_{i+1} > \alpha_y \Delta y_{j+1} \\ (I+1, J+1) & \text{if} \quad \alpha_x \Delta x_{i+1} = \alpha_y \Delta y_{j+1} \end{cases} \tag{8}$$

and

$$g(I,J) = \min(\alpha_x \Delta x_{i+1}, \alpha_y \Delta y_{j+1}). \tag{9}$$

The total length traversed until an interaction with a $\boldsymbol{X}, \boldsymbol{Y}$ partition line is by definition the corresponding $g(I,J)$ evaluation. For every step in the traverse iteration, the local length l_k inside every voxel was calculated by following subtraction:

$$l_k = g\left(\underbrace{f \circ \cdots \circ f \circ f}_{k}\right) - g\left(\underbrace{f \circ \cdots \circ f}_{k-1}\right) \tag{10}$$

where $\circ$ represents function composition. This implementation of ray tracing algorithm is completely iterative and avoids branching, allowing the full algorithm to reduce thread divergence during execution to a minimum.

3.2 Discrete CCK Algorithm

The final ingredient of the algorithm is the dose calculation. Let us consider now a ray with a (θ_m, ϕ_n) collapsed cone direction from a starting point in the patient tomography.

An efficient discretization can be defined in terms of a double integration of the collapsed kernel $k^{m,n}$ by expanding the line convolution represented in (4) and using the Stieltjes integral definition. To this end, we define q_k as the distance between the borders of segment l_k and l_o, as seen in Fig. 4. the dose calculation for point x was transformed into (12) using a double integration of the collapsed kernel defined in (11):

$$c_k^{m,n} = \int_{q_k}^{q_k+p} \left(\int_0^t k^{m,n}(u)du \right) dt \tag{11}$$

With this definition, the η is no longer explicitly present and the dose calculation in every voxel point ends up being the sum of a Stieltjes integral for every different defined direction. A discretization is obtained by using the CCK (Collapsed Cone Kernel) method defined by Lu *et al.* where the collapsed kernel is allowed to vary inside each voxel [9]:

$$D^{m,n} = \frac{1}{p} \left(T_o c_o^{m,n} + \sum_k T_k \left(c_{k+1}^{m,n} - c_k^{m,n} \right) \right) \tag{12}$$

This discretization allows a complete variation of the collapsed cone kernel, defining the radiological distances between the ends of each voxel and integrating in the corresponding segments (see Fig. 4).

3.3 GPU Implementation

The approximations presented enables one to cast the multidimensional convolution integral into a complex, ray tracing problem, which is *embarrassingly parallel*, since every line of integration and voxel are independent from the others. Therefore, a CUDA code to be deployed in a GPU is an obvious solution. However, one of the main issues when resolving this type of convolutions is the amount of reading operations needed in every iteration, as the kernel contribution is always relevant and memory access could easily be non coalesced. Hence, all access to global memory must be optimized to ensure the best performance [13].

Most of the optimization work was devoted to the ray tracing algorithm, because there are at least three reads from memory and performs 17 single clock

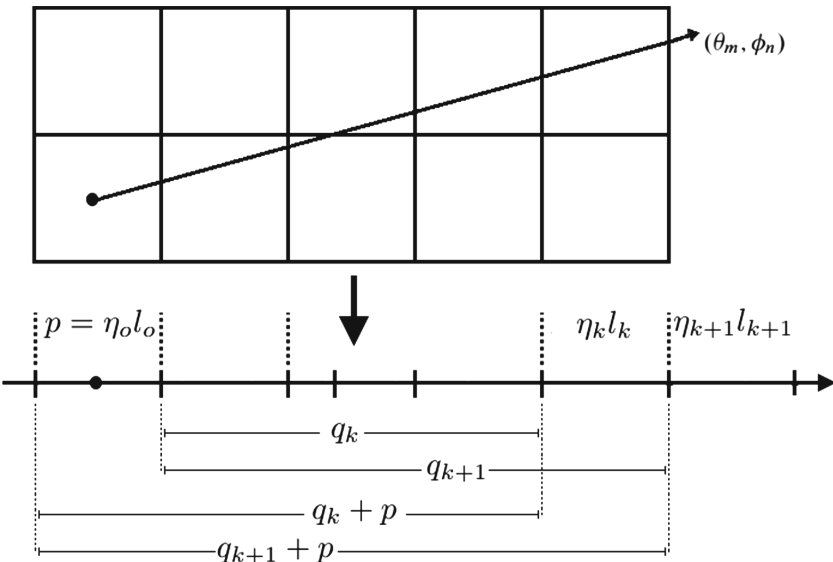


Fig. 4. Schematics of radiological paths needed to numerically resolve the dose calculation and account for kernel variation inside each voxel. A line l traverse a group of voxels defining each l_k segment and radiological segments $\eta_k l_k$ as well as radiological distances q_k.

operations in every iteration step (Algorithm 1). As every transport line is independent from each other, they cover all tomography and start from independent voxels. Therefore, a fully coalesced memory approach was unfeasible. We developed two different implementations of the CCK algorithm, using global memory and texture memory to store the density and TERMA information, as both need to be read in every iteration (see Eq. (12)).

The parallelization relies in the independence of every thread: each one starts the calculation in one voxel using the `threadIdx` variable, and loops accounting for the dose coming from every collapsed cone direction. In such way, all threads move in the same direction and access memory in the same neighborhood. With this implementation, the texture memory should be the best option to optimize local memory access [13]. Therefore, each thread must resolve the sum in (12) and fetch all $c_k^{m,n}$ values, increasing memory overhead but reducing precision loss.

The double integral over the collapsed cone represented in (11) was stored in texture memory, as its hardware interpolation capabilities allow to develop a simpler code and increase performance when calculating the specific $c_k^{m,n}$ values. Although there are cases where such array can be condensed into shared memory, this are usually rare and depend strongly on the amount of line integrals used. Optimization tests consisted in four function implementations and were compared with an equivalent multi-threaded CPU code, executed and profiled on a system with a NVIDIA GTX 980 GPU, Intel Core i7 4790 CPU and 32 Gb of 1333 MHz memory.

Algorithm 1. Ray tracing implementation

```
1: function RAY_MOVING(...)
2:     (test_x, test_y) = (Δx · |α_x|, Δy · |α_y|)        ▷ test vector
3:     step_x = (test_x ≤ test_y) ? 1:0                     ▷ f selection
4:     step_y = (test_y ≤ test_x) ? 1:0                     ▷ f selection
5:     l_k = sqrt((test · step))/(step_x + step_y) − l      ▷ local distance
6:     l = sqrt((test · step))/(step_x + step_y)            ▷ g function application
7:     (Δx, Δy) += (step_x, step_y)
8:     (I, J) += sign(α) · step                             ▷ f function application
9: end function
```

The code was divided in a `ray_moving` function whose parameters are the local index (I, J), total cartesian distance and the ray parameters $(\alpha_x, \alpha_y, \alpha_z)$. This function uses the (f, g) pair to represent the equations described from (5) to (10) and resolve together with global/texture memory access, the ray tracing algorithm. Another function called `getCCK` fetches the texture on a `cuArray` and calculates the local energy deposition of the current ray, resolving (11).

The `terma` data was pre-calculated with a Monte Carlo simulation of the linear accelerator complete system [4], as is customary in radiotherapy. With such implementation, as long as there are no structural changes, the photon beam remains unchanged. Finally, the dose calculation depicted on (12) was implemented on a different function, using the `ray_tracing` in conjunction with global or texture memory. The four algorithms are summarized in Table 1.

Table 1. Main CCK functions with memory access and arithmetic operations.

Function	Memory	Arithmetic	Optimization	Resolution	Eq.
`ray_moving`	0	17	No flux control	Thread moving	(5)–(10)
`getCCK`	4	8	Texture	Texture fetching	(11)
`terma`	4	4	Pre calculated	Dose preparation	(12)
`dose`	6	18	Global/Texture	Dose calculation	(12)

4 Results and Discussion

4.1 Dose Accuracy

A complete comparison of CCK algorithm was made against experimental and numerical data provided by FUESMEN (Fundación Medicina Nuclear, Mendoza, Argentina). The simulation used a $128 \times 128 \times 128$ virtual tomography composed from different materials and 0.2 cm width voxels. To validate the percentage depth dose data (PDD) and profiles in different directions we used the maximum relative error existing between the two sets of data. Depth curves for

homogeneous and heterogeneous completed the acceptance test with a maximum percent error under 3%, compared with the experimental data provided [14]. In Fig. 5 both numerical calculation and experimental data are shown. This is an example of the good physical behavior of the collapsed cone approximation, and has been thoroughly tested for different experimental conditions.

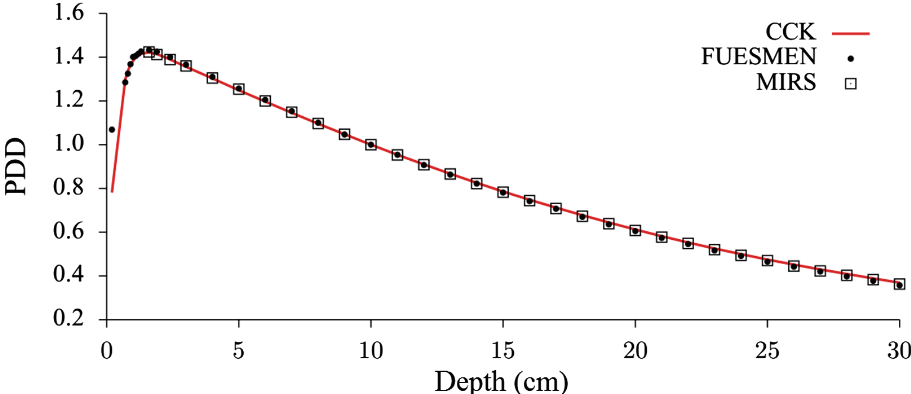


Fig. 5. Depth dose calculation with CCK algorithm compared with experimental data from FUESMEN and numerical solutions from commercial planning system. Simulation in full water tomography.

4.2 Calculation Performance

Performance comparisons were made between the two main implementations of the CCK algorithm schemed in Table 1, and the equivalent multi-threaded CPU code. The main difference in time execution was the memory used to store the density and TERMA arrays, resulting in different speedups compared to those of the CPU code. As every thread started from a one-to-one mapping in the density array, each iteration needed a memory read of near spatial location, making texture memory a better option overall. Although global memory contains a local cache implementation for reducing memory access, the efficiency observed from texture fetching from `cuArrays` improved the global implementation by a 1.4X and the CPU code by an 144X.

With the texture implementation, the memory access bottleneck was reduce given the 3D locality of `cuArrays` type, while the allocation cost increased by the texture binding, the arithmetic time remains almost the same (see Table 2).

Different grid sizes where computed to quantify the GPU dependency with tomography size. We determine that the overall most efficient block size corresponded to a threads distributed in a $(4, 4, 4)$ array. Bigger grid blocks resulted in blocks not executing simultaneously at the same time because of GPU occupancy, meanwhile smaller grid blocks reduced the total occupancy and increased the warp execution and corresponding overhead.

Table 2. Final calculation time and speedup for 128 grid size and different GPU and CPU implementations.

Implementation	Block size	Time (s)	Speedup	Calculation fraction		
				Allocation	Access	Arithmetics
CPU version	8 cores	620	1	0.0	0.91	0.09
GPU global memory	$(4, 4, 4)$	6.02	103	0.10	0.73	0.17
GPU texture memory	$(4, 4, 4)$	4.32	144	0.13	0.62	0.25
	$(8, 8, 8)$	6.12	101	0.13	0.59	0.27
	$(4, 4, 2)$	5.41	115	0.13	0.66	0.22

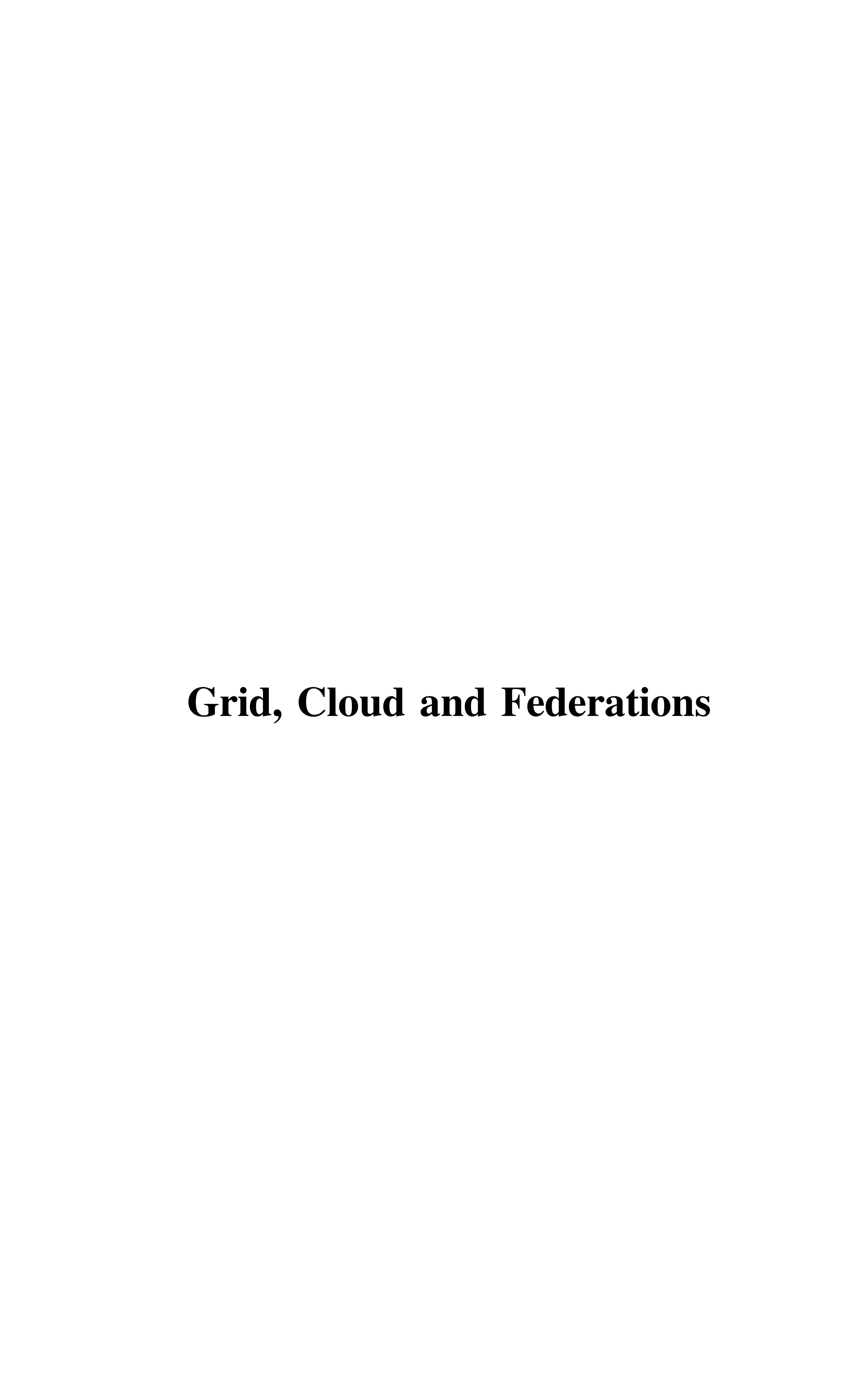
5 Conclusions

The purpose of radiotherapy planning treatment is to obtain the spatial dose distributions to exactly predict quantitatively the delivered dose. However, the continuous development of new calculation tools for advanced implementation in clinical practice, has managed to reduce the uncertainties to acceptable levels. Nowadays, algorithms based on Convolution/Superposition models of a point kernel have the highest accuracy if one takes into account the time required in clinical practice These are the algorithms mostly present in commercial treatment planning systems. In the present contribution, a Convolution/Superposition algorithm was developed, provided that energy fluence generated by a linear accelerator is known, as is the case in clinical practice. Besides, the algorithm was implemented in CUDA capable GPUs. The dose calculation was obtained using the collapsed cone approximation, that reduces the numerical complexity and enables a high resolution grid of calculation points. Dose profiles and depth dose rates were calculated and validated with experimental data and Monte Carlo gold standard. The full GPU implementation of the present CCK algorithm, using ray tracing and texture memory brings a 144X speedup in comparison with equivalent multicore CPU codes. A block size of $(4, 4, 4)$ threads was found to be the perfect match for speed increase as it represents a good balance between warp execution count and GPU register occupancy. Future work includes the use of managed memory and warp-wise operations and porting the code to multiGPU architectures.

References

1. Cowley, J.: Diffraction Physics. Elsevier, Amsterdam (1995)
2. Ahnesjö, A., Aspradakis, M.: Dose calculations for external photon beams in radiotherapy. Phys. Med. Biol. **44**(11), R99 (1999)
3. Ahnesjö, A., Andreo, P., Brahme, A.: Calculation and application of point spread functions for treatment planning with high energy photon beams. Acta Oncol. **26**, 49–56 (1997)

4. Fippel, M., Haryanto, F., Dohm, O., Nüsslin, F., Kriesen, S.: A virtual photon energy fluence model for Monte Carlo dose calculation. Med. Phys. **30**, 301–311 (2003)
5. Childress, N., Stephens, E., Eklund, D., Zhang, M.: Mobius3D white paper: dose calculation algorithm. Mobius Med. Syst. (2012)
6. Ahnesjö, A.: Collapsed cone convolution of radiant energy for photon dose calculation in heterogeneous media. Med. Phys. **16**, 577–592 (1989)
7. Batho, H.: Lung corrections in cobalt 60 beam therapy. J. Can. Assoc. Radiol. **15**, 79 (1964)
8. Vanderstraeten, B., Reynaert, N., Paelinck, L., Madani, I., De Wagter, C., De Gersem, W., De Neve, W., Thierens, H.: Accuracy of patient dose calculation for lung IMRT: a comparison of Monte Carlo, convolution/superposition, and pencil beam computations. Med. Phys. **33**, 3149–3158 (2006)
9. Lu, W., Olivera, H., Chen, M., Reckwerdt, P., Mackie, T.: Accurate convolution/superposition for multi-resolution dose calculation using cumulative tabulated kernels. Phys. Med. Biol. **50**, 655 (2005)
10. Jacques, R., Wong, J., Taylor, R., McNutt, T.: Real-time dose computation: GPU-accelerated source modeling and superposition/convolution. Med. Phys. **38**, 294–305 (2011)
11. Chen, Q., Chen, M., Lu, W.: Ultrafast convolution/superposition using tabulated and exponential kernels on GPU. Med. Phys. **38**, 1150–1161 (2011)
12. Xiao, K., Chen, D., Hu, X., Zhou, B.: Efficient implementation of the 3D-DDA ray traversal algorithm on GPU and its application in radiation dose calculation. Med. Phys. **39**, 7619–7625 (2012)
13. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**, 40–53 (2008)
14. Low, D., Harms, W., Mutic, S., Purdy, J.: A technique for the quantitative evaluation of dose distributions. Med. Phys. **25**, 656–661 (1998)



Grid, Cloud and Federations

Eeny Meeny Miny Moe: Choosing the Fault Tolerance Technique for my Cloud Workflow

Leonardo Araújo de Jesus, Lúcia M. A. Drummond, and Daniel de Oliveira(✉)

Instituto de Computação, Universidade Federal Fluminense (UFF), Niterói, Brazil
leonardoaraujo@id.uff.br, {lucia,danielcmo}@ic.uff.br

Abstract. Scientific workflows are models composed of activities, data and dependencies whose objective is to represent a computer simulation. Workflows are managed by Scientific Workflow Management System (SWfMS). Such workflows commonly demand for many computational resources once their executions may involve a number of different programs processing a huge volume of data. Thus, the use of High Performance Computing (HPC) environments allied to parallelization techniques provides the support for the execution of such experiments. Some resources provided by clouds can be used to build HPC environments. Although clouds offer advantages such as elasticity and availability, failures are a reality rather than a possibility. Thus, SWfMS must be fault-tolerant. There are several types of fault tolerance techniques used in SWfMS such as checkpoint-restart and replication, but which fault tolerance technique best fits with a specific workflow? This work aims at analyzing several fault tolerance techniques in SWfMSs and recommending the suitable one for the user's workflow using machine learning techniques and provenance data, thus improving resiliency.

1 Introduction

Scientific Workflows can be defined as abstractions used to define sequences of activities and data dependencies among them [1]. Each activity in a workflow represents the invocation of a program. Workflows are modeled, executed and monitored by complex engines named Scientific Workflow Management Systems (SWfMS). Many domains of science use scientific workflows and SWfMSs in their daily duties such as biology, chemistry and astronomy [2]. Several existing workflows are large-scale, which means they produce and consume large amounts of data and are usually executed repeatedly until a hypothesis can be confirmed or refuted. Due to the high volume of data involved in such executions, as well as the high demand for processing, many workflows need to be executed in High Performance Computing (HPC) environments, such as well-known clusters and supercomputers [2].

However, in the last decade, clouds [3] have been used as HPC environments, since they offer HPC resources in their (commonly huge) pool of resources. Clouds offer a wide variety of on demand resources (*e.g.* Virtual Machines - VMs and storage). In addition, clouds offer elasticity, which may be valuable for

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 321–336, 2018.
https://doi.org/10.1007/978-3-319-73353-1_23

executing large-scale experiments. This means the scientist deploys (or undeploys) resources on demand and one pays according to the quantum of time the resources were used. Many SWfMS already allow for running workflows in clouds such as Pegasus [4] and SciCumulus [5].

However, executing workflows in clouds raises some challenges. Even when using HPC and dedicated VMs, activities are usually executed several times and each execution may last for several hours, which increases the possibility of a failure. Especially in clouds, data centers that host VMs are traditionally composed of thousands (or even millions) of computers, and failures are a reality rather than a possibility. In fact, some papers have investigated and discussed the failure growth when the amount of resources involved in the execution increases [6]. Even if a VM does not fail, its performance is subject to variations due to providers' procedures such as live migrations [7]. When one or more failures indeed occur, the workflow execution must be able to continue.

Many programs implement their own application-level fault tolerance mechanisms, *i.e.* the program has a specific strategy to be adopted to recover from failures. However, this strategy cannot be adopted for scientific workflows. In many existing workflows, the programs associated to the activities are black-boxes *i.e.* users do not have access to their source code. This way, the SWfMS must be fault-tolerant instead of the invoked programs. Unfortunately, adding fault tolerance mechanisms into cloud-based SWfMS brings an additional overhead. The SWfMS is no longer responsible only for managing the workflow (*e.g.* scheduling activities on VMs, collecting and storing provenance [8] data) but also for monitoring the entire environment, aiming at minimizing costs and maximizing the overall quality of the results. It is important to stress that these mechanisms are no longer optional since failures are rather a rule than an exception in clouds.

Many existing fault tolerance techniques (FTT) [9–15] can be coupled to existing SWfMS. These techniques, such as checkpointing, re-executions and replication, can be applied in many scenarios. However, different workflows present different characteristics, and thus choosing the best suitable fault tolerance technique may vary according to the each specific workflow. It is commonly unfeasible to apply all possible fault-tolerance techniques in a single workflow execution, since the imposed overhead would be non-negligible when compared to the total execution time of the workflow. Thus, scientists (or the SWfMS) must choose the suitable (which will introduce a negligible overhead and guarantees that the worflow will run) technique for their own workflow (or even for a specific activity of the workflow). It is worth mentioning that there is no "one-size-fits-all" technique, *i.e.* there is no fault tolerance technique that will introduce the smallest overhead for any workflow.

This paper aims at exploring a series of fault tolerance techniques in cloud-based scientific workflows and chooses the best one for a specific workflow execution using machine learning techniques (decision trees [16] and CN2 rule inducer [17]). Such algorithms produce a set of predictive rules that can be implemented within SWfMSs in order to determine the fault-tolerance technique for a workflow. In this paper, we evaluated 3 types of fault tolerance mechanisms in the Sci-

Cumulus SWfMS: Checkpoint/Restart (C/R), Retrying, and Replication. Thus, the main contributions of this paper are: (i) a monitoring mechanism implemented in the SciCumulus SWfMS for the VMs and network in order to detect slow running tasks, VMs that fail, slow network, *etc.*; (ii) a checkpoint-restart mechanism implemented in SciCumulus SWfMS that does not require modifying any program source code; (iii) an activity replication mechanism implemented in SciCumulus; (iv) an experimental evaluation using machine learning algorithms that supports the decision making process to choose the FTT that is best suitable for each activity in a given workflow; and (v) a decision module that evaluates the workflow class and decides which of the available FTT is best suitable for it.

2 Related Work

As previously discussed, fault tolerance is a key issue in cloud-based scientific workflow execution. Thus, many approaches and algorithms have been proposed in the last years. These approaches and algorithms aim both at preventing the occurrence of faults and at recovering once the failure has already occurred. Hu *et al.* [9] proposes a scheduling heuristic that may choose to replicate tasks in certain situations such as when idle resources are detected. The schedule of replicated tasks, specially the ones in the critical path of the workflow, is interesting in case of a slowdown in one of the VMs involved in the execution, as the replicated task may finish earlier than the original one, or when a failure occurs thus allowing the execution of its successors. Gu *et al.* [10] aims at solving the resource mapping problem for a scientific workflow while minimizing failure occurrences and maximizing throughput. Although it mentions well-known fault tolerance techniques such as retrying, replication and checkpoint-restart, it relies on mapping tasks into resources whose failure rate is as low as possible in order to minimize failures. Costa *et al.* [15] proposes a fault tolerance heuristic for cloud-based workflows. It is based on re-executions, instead of trying to recover some partial result of the previous execution. Re-executions can be effective when the workflow is composed of many short-term activities, *i.e.* if one activity fails, the re-execution will be short-term as well. However, when the workflow is composed of long-term activities, re-executions can impose a non-negligible overhead.

Bala and Chana [11] propose a scheduling approach which takes into account the possibility of having a failure due to overusage of resources. It prioritizes the schedule of the activity with the higher number of data dependencies, thus reducing the makespan of the workflow. To avoid failures, VMs can be migrated to less used hosts. However, in a practical situation, public cloud providers as Amazon AWS, Google Cloud or Microsoft Azure do not allow for this sort of control by their customers. Jain *et al.* [12] present a framework named FireWorks that aims at providing a fault-tolerant platform for scientific workflows. It has several features, allowing for gathering provenance data, executing concurrent activations, etc. In terms of fault tolerance, authors detect 3 types of failures: "soft" (script throws an error), "hard" (machine stops working) and queue failures.

To the best of authors' knowledge, existing SWfMS grounds its resiliency into some of the aforementioned techniques. Retrying (re-submission) and Replication seem to be more popular as in [10–12,15]. Although C/R techniques are also common, it is usually applied at workflow level [18,19]. When using C/R, the SWfMS stores information regarding the execution progress in terms of finished activities. If a failure occurs and a workflow needs to be restarted, it is possible to continue the execution from the last completely finished activities onwards. However, this technique is not able to recover the partial work of a failed activity. If this is a long-term activity, this loss may increase the workflow makespan. In terms of activity level C/R, SWfMS usually relies on the inclusion of specific C/R libraries in the activity source code [13,14]. This solution is not feasible when the workflow comprises the execution of black-box programs, provided by third parties developers, and this is rather the rule than an exception for most scientific workflows. In addition, it is worth mentioning that no approach tries do identify the best fault tolerance technique or mechanism to use. They rely on the scientists' choice and configurations, which may be not the best one to use.

3 Background Knowledge

3.1 Failure Detection

In general, faults are detected by monitoring the environment and the workflow execution. Most existing SWfMS provide monitoring mechanisms, but to the best of authors' knowledge, fault-detection mechanisms are reactive, *i.e.* the SWfMS does not try to prevent from failures. In addition there is no fault-detection mechanism that is coupled to cloud APIs in order to evaluate the status of VMs, etc. This way, a proactive algorithm was developed in order to detect failures as soon as possible in activities executing in clouds. This algorithm is called FTAdaptation (Algorithm 1). It tries to detect potential failures before an activity actually fails. It also tries to detect degraded VMs since they are more error-prone and, thus, avoid the occurrence of new activity failures. This algorithm encompasses 2 main methods: *VerifyClusterHealth* and *SelectBadMachine*. *VerifyClusterHealth* is responsible for fetching information about the VMs and network. In this paper we implemented Algorithm 1 in SciCumulus SWfMS using the Amazon API[1] in order to obtain data. Amazon provides information about connectivity, VM status (*executing*, *initializing* or *stopped* for example), system status (*ok* or *impaired*), etc. This information is stored in the SWfMS's provenance database that contains information about the workflow structure, past executions and performance data. *SelectBadMachine* queries information collected by *VerifyClusterHealth* and combines it with the execution history of the workflow so it is possible to discover which VMs are in error state or where a high number of activity failures are occurring - the implementation implies a VM is problematic if the number of failures exceeds a given threshold. Once

[1] http://docs.aws.amazon.com/cli/latest/reference/ec2/describe-instance-status.html.

these VMs are identified, it is possible to replace them by new ones or to upgrade them. Unlike the proposed algorithm, most of the SWfMS is reactive, *i.e.*, after each activity execution, the returned information (the process exit status for example) is checked and once the failure is detected, the activity is resubmitted for execution.

Algorithm 1. FTAdaptation()

```
1: while workflow is not finished do
2:     VerifyClusterHealth()
3:     $machines = SelectBadMachine()
4:     if $machines is not empty then
5:         Stop SWfMS
6:         RequestMachines($machines)
7:         RequestMachineTermination($machines)
8:         Start SWfMS
9:     Sleep($param)
```

3.2 Failure Handling

As previously discussed, this paper aims at helping scientists to choose the best FTT for their cloud-based workflows. In addition, we have investigated that C/R, Retrying and Replication are the most common techniques used in workflows. Since Retrying technique is pretty obvious to understand, this subsection presents how we implemented the C/R and Replication in SciCumulus SWfMS [5][2] (SWfMS used in the experiments). Figure 1 pictures SciCumulus executing a workflow in the cloud computing using 4 VMs. Each VM executes an instance of the SciCumulus' Core [5], a component responsible for data transfer, scheduling *etc*. Communication between VMs is performed through MPI message exchange. The Master component has the responsibility of scheduling activities while the other components (Workers) request activities to process, acting as a Master/Worker model.

To implement C/R in SciCumulus we had to add a special feature to the Master VM. This feature is described by Algorithm 2, which determines the activities that are eligible for checkpointing. This is determined by considering the execution time for each currently executing activity. This information is available at the SciCumulus provenance database through a simple SQL query - we assume the used SWfMS has a complete register of the previously and currently executing activities, as we find in SciCumulus. For each activity$_i$, whose execution time since the last checkpoint t$_i$ is greater than δ, where δ is an input parameter, a checkpoint request is sent to the SCCore instance responsible for running activity$_i$. Each Worker instance runs Algorithm 3, waiting for checkpoint requests, and execute Algorithm 4.

[2] http://scicumulusc2.wordpress.com/.

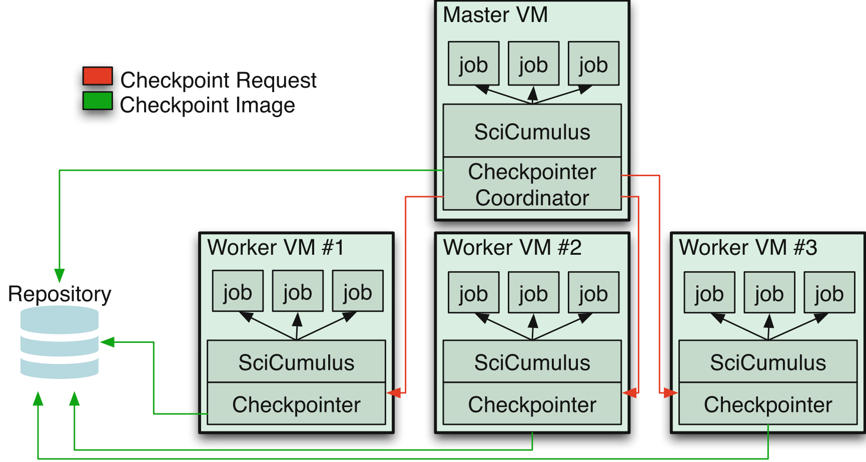


Fig. 1. SciCumulus's distributed checkpoint-restart architecture

Algorithm 2. CheckpointCoordinator()

1: **while** *workflow is not finished* **do**
2: *$infoList = GetCheckpointInformation()*
3: **for each** *$info* **in** *$infoList* **do**
4: *SendCheckpointRequest($info.workspace, $info.rank)*
5: *Sleep($param)*

Algorithm 3. CheckpointListener()

1: **while** *workflow is not finished* **do**
2: *Receive($msg)*
3: *Call Checkpoint($msg.workspace)*

Algorithm 4. Checkpoint(*$workspace*)

1: *$pid = pgrep($workspace) * Queries the process's PID based on its workspace **
2: *Dump($pid,$dest) * Stores the process image and leaves it in STOPPED state **
3: *Compress($workspace)*
4: *kill -SIGCONT $pid * Restarts the process **
5: *Compress($dest)*
6: *Move($compressedFiles,$repository)*
7: *Delete($oldCheckpoint)*

A checkpoint request contains the activity's workspace - an operating system folder - where the activity is executing. Given that each activity is associated with a unique workspace, it is trivial to obtain the activity's process id (PID) through OS commands. The activity is then paused, its image is stored in disk using, input and partial output files are copied and included to the image files and this dataset is sent to a safe and scalable repository (in our case, Amazon S3). Lastly, the activity is restarted.

Whenever a failure occurs in any activity, the SWfMS scheduler tries to execute it once again (Retrying technique). However, if a checkpoint image for this activity exists, instead of executing it from the beginning, it has to fetch and extract the activity's checkpoint image data from the repository to the activity's workspace and restores the process through Algorithm 5. C/R is also applicable to scenarios where an activity has been executing for a very long period of time at a certain VM. Once this situation is detected, the scheduler may decide to migrate it to a VM with more processing power and, instead of restarting the activity from the beginning in the new VM, it is possible to take advantage of some partial execution previously made at the original VM.

Algorithm 5. Restart($\$path$)

1: *$ckpt = Copy($Path)*
2: *Extract($ckpt,$dest)*
3: *Restart($dest)*

Algorithms 4 and 5 rely on external Checkpoint-Restart tools. This work has adopted CRIU[3] as C/R tool due to its advantages in comparison to some similar solutions such as BLCR[4]. Its advantages include: (i) no need to load any libraries to the activity's program, (ii) possibility of having C/R without having to modify or prepare any programs and (iii) storage of open files. These characteristics are important in the context of scientific workflows since they usually encompass the execution of multiple third-party programs, which are often black-box and statically linked.

Besides C/R, we also implemented Replication mechanisms in SciCumulus. Replication is a technique where a unit of work is copied at least once enabling some degree of resiliency due to redundancy [20]. Even if a copy of an activity does not finish properly (due to a failure), the remaining copies may do. Thus, the execution of the workflow can continue by using the output generated by any of the successfully finished activities. This implementation includes some extra attributes to each activity in SciCumulus. These attribute control the priority of a given activity and include a group identification, which is important in a replication scheme because activities have to be associated with their replicas. The priority attribute allows for some flexibility in activity scheduling. It is possible

[3] http://criu.org/.
[4] crd.lbl.gov.

to give priority to "primary" copies or give copies equal priorities. Giving priority to primary copies would speed the execution up as redundant copies would be scheduled after primary ones and perhaps wait for them to finish. However, in case of a failure, if no activity duplicate is executing at that moment, it would demand a retry from the beginning. This would be interesting in scenarios where the possibility of failures is low and activities are short-term. On the other hand, equal priority would result in activities and their n copies running at "the same time", thus tolerating *n-1* failures. So, this paper focuses on the priority mode where activities and their copies are executed relatively at the same time, either in the same VM in different cores or completely distinct VMs.

Algorithm 6. ExecuteActivity()

```
1: $activity = SendRequestActivity()
2: Execute($activity)
3: if $activity.finish_status == 0 then
4:     SendNotification($m)
5: else
6:     if $activity.finish_status == FINISHED_REP then
7:         FinishNormally()
```

Algorithm 6 runs at each SciCumulus Worker instance. It requests activities from the Master node and executes them. It was extended in order to notify the Master node about the successful execution of an activity and supporting the interruption of activations after some replica from its group finishes executing. *SendNotification* is a method that simply sends a message containing the activity id, group number and finished activity id to the Master node. *FINISHED_REP* is a special finish status created to characterize activities that were finished not normally but due to a request after the successful execution of some replica of a given activity. Algorithm 7 runs at Master node. It waits for messages from finished activities in order to terminate their remaining replicas.

Algorithm 7. WaitForNotifications()

```
1: $msg = ReceiveNotification()
2: cancelledGroups.add($msg.group_id)
3: $machines = Query($msg.activation_id, $msg.group_id)
4: for each $machine in $machines do
5:     SendStopActivations($m.workspace, $m.rank)
```

4 Experimental Results

This section presents the experiments performed in order to identify the suitable fault tolerance technique for a specific workflow. Firstly we present the workflows

chosen as case study, then we discuss about the executions performed to collect performance data, and finally we propose the mechanism based on machine learning algorithms.

4.1 Preliminary Analysis of FTT in Montage and SciPhy Workflows

Montage Workflow. Montage is an astronomy workflow that collects data through telescopes and creates mosaic images from objects in the sky. As presented in Fig. 2, it consists of 9 activities. Figure 2 does not necessarily show the exact number of executions of each activity, but provides information about how each activity works regarding input and output data. Since we need input data for the machine learning algorithms, we executed Montage several times. We have induced errors at the end of the execution of the activities. Figure 3 shows the execution times of the Montage's activities for the 3 available FTT (Retry, C/R and Replication) with and without faults (NF, at the graph legend, denotes No Fault).

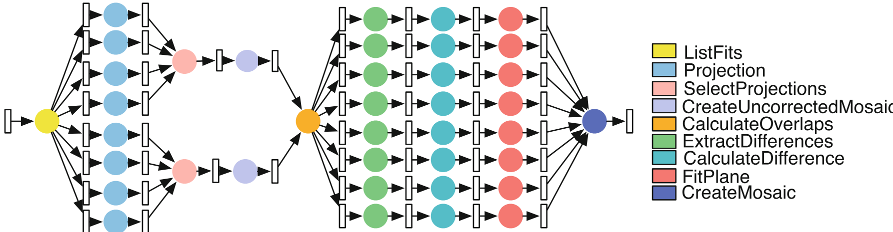


Fig. 2. Montage workflow

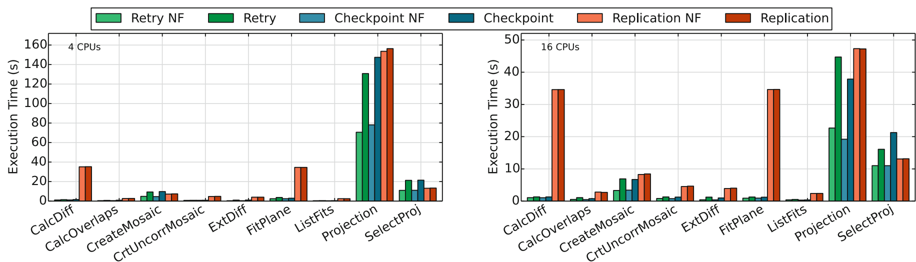


Fig. 3. FTTs performance for Montage with 9 activities

Most activities executed are short-ones - they executed in less than 20 s - when using Retry or C/R as FTT. The only exception is the activity *Projection*. Among all these short-term activities, it is interesting to notice 2 in special due to a much lower performance while using Replication technique: CalculateDifferences and FitPlane. Table 1 helps to clarify the reasons behind it. It is worth

noticing that these 2 activities are the ones which has to process the biggest datasets. Replication for these activities would hence result in a high number of processing and parallel executions, increasing queuing time and wait time in order to the correct termination of the redundant replica. As a consequence we observe a worse performance.

Table 1. Activities in the execution of Montage workflow

Name	# of processed files	Avg time retry	Avg time replication
CalculateDifference	17	0:01.121	0:34.808
CalculateOverlaps	1	0:00.523	0:02.995
CreateMosaic	1	0:04.266	0:08.206
CreateUncorrectedMosaic	1	0:00.809	0:04.549
ExtractDifferences	1	0:00.466	0:03.842
FitPlane	17	0:01.609	0:34.628
ListFits	1	0:00.392	0:02.427
Projection	10	0:45.918	1:22.152
SelectProjections	1	0:11.954	0:13.153

In contrast, activities that process small amounts of data have similar performance for any of the considered FTT. As previously mentioned, replication of activities carries additional work in order to correctly finish the replicas. If the activity is a short-term one, the additional work carried by the replication technique may be grater than the activity execution time. This is seen by the comparison between the CreateMosaic and SelectProjection to any other short-term activity. It is worth noticing from Fig. 3 with 4 CPUs that, for the SelectProjections activity, Replication performs similarly in non-failure scenarios and better than the other 2 FTT in failure ones. As the performance of the Retry technique improves (activities execute faster due to the increase in the resource amount), Replication turns into a not so good technique for the CreateMosaic activity, but still performs as the best one for SelectProjections.

It is also worth mentioning that no activity was able to restore a checkpoint during this experiment since all activities executed in less than 30s (the checkpointing interval). Thus, even though Checkpoint-Restart was set for the experiment, as no valid checkpoint was available at the restore moment, recovery was performed through Retrying. Hence the results show basically similar results for Retry and C/R.

SciPhy Workflow. SciPhy is a workflow from the biology domain that receive a dataset of DNA, RNA and aminoacid sequences and creates evolutionary models in order to determine common ancestors between organisms [21]. As presented

in Fig. 4, it basically consists of a pipeline containing 3 sequential activities followed by 2 activities that can be executed in parallel. This experiment was carried similarly to the previously presented for Montage. It involved the repeated execution of the SciPhy workflow varying the applied fault tolerance technique, the faulty activity, the number of CPUs involved in the execution and, differently from the previous experiments, the number of input files (that contains entire genomes). Thus, Figs. 5, 6 and 7 represent the average execution time of each activity belonging to the SciPhy workflow with 1, 4 and 8 input genomes, respectively, running in 3 different computational environments: 2 CPUs/4 GB RAM, 4 CPUs/8 GB RAM and 8 CPUs/16 GB RAM, except for SciPhy-8 which was not executed with only 2 CPUs as execution time would boost considerably due to queuing.

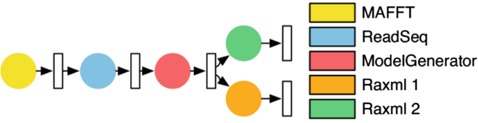


Fig. 4. SciPhy workflow

Similarly to the Montage experiment, most activities do not spend much time on processing, except for ModelGenerator and RAxML activities. As RAxML makes use of parallel processing with threads, queuing activities deteriorates the performance considerably, due to increased cache degradation [22]. RAxML was set to use 2 threads, the minimum accepted value.

By analyzing Figs. 5, 6 and 7 it is possible to observe the effect of increasing the number of input genomes on the FTTs behavior. Figure 6 shows that Replication is no longer suitable for ModelGenerator for any of the computational environments as its performance in both best and worst case scenarios is comparable to the worst case scenario using the Retry technique. C/R has the best performance for ModelGenerator in the 3 considered VMs. It is worth noticing that the Checkpointer developed at this work does not run in parallel

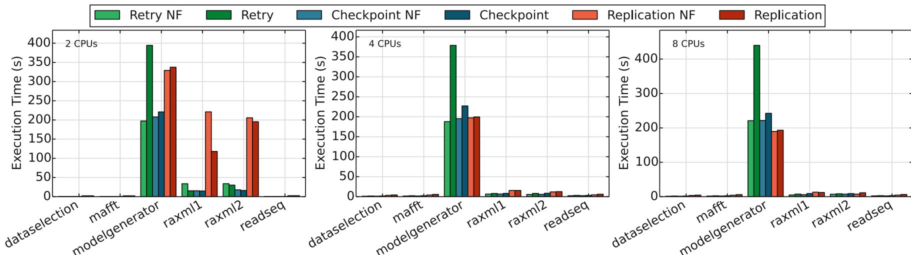


Fig. 5. FTTs performance for SciPhy with 1 input genome

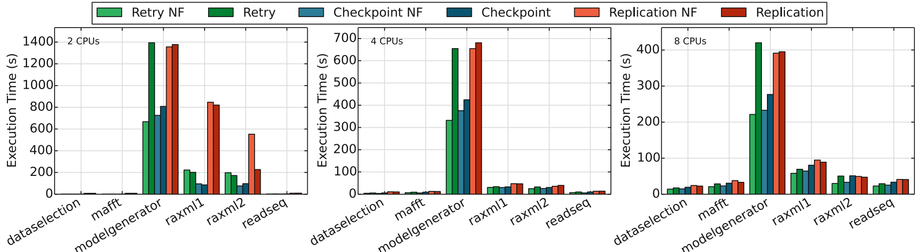


Fig. 6. FTTs performance for SciPhy with 4 input genomes

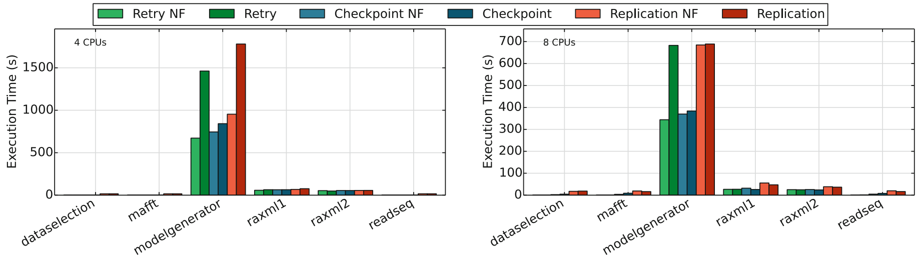


Fig. 7. FTTs performance for SciPhy with 8 input genomes

at this version due to CRIU limitations. However, as this implementation has a quite low overhead this does not impact on the results, as it is seen in Figs. 6 and 7. Comparing the execution results from ModelGenerator using Retry and Checkpoint in different VMs for the best scenarios (with no failures) it is possible to notice that Checkpointing tends to have a slightly worse performance (around 4% at these experiments, using the interval parameter as 30 s). This is observed because while the Retry technique does not need any extra work to be done, Checkpointing demands message exchanging, monitoring the processes in execution and their periodical stop and resume in order to store their images. The use of Checkpointing is, however, justified by the guarantee of the upper bound loss threshold - related to the checkpoint interval parameter - it gives to the executions. This benefit comes at a reasonably low cost, while the other techniques would not deliver such performance in terms of loss limit (Retry) or incurred overhead (Replication).

4.2 Developing a Predictive Model for FTT in Scientific Workflows

Although the previous qualitative analysis provides some interesting insights about the best FTT technique to use with each activity of both workflows, it should be performed in larger scale. Instead of analyzing a few workflow executions traces, we should consider hundreds or even thousands of traces and then try to extract useful patterns that help choosing the suitable FTT for a specific

workflow. It may be very tedious and error-prone (if not unfeasible) to manually analyze such volume of data.

Thus, we developed predictive models using machine learning algorithms to help determining the most suitable FTT for a workflow. Those models were coupled to SciCumulus SWfMS in these experiments. To develop this predictive model we considered as input a database containing 13,628 execution traces of both Montage and SciPhy workflows. To infer knowledge about these traces, machine learning methods are used to discover interesting patterns in such data. In this specific analysis, we used Decision Trees [23] and the CN2 Induction Algorithm [17]. The goal of both algorithms is to create a model that predicts the value of a target variable based on several input variables. The input variables are represented in the input workflow traces. Each tuple of the input dataset contains the name of the activity, its type (following the algebraic approach proposed by Ogasawara *et al.* [24]), the type of FTT used, the number of parallel tasks associated with the activity, number of replicas and the checkpoint interval (if applicable). The target variable is the execution time of the activity since we are looking for a FTT that provides the best makespan. Systems for inducing knowledge from examples are valuable tools for assisting in the task of knowledge acquisition for expert systems, which is our case.

We used Orange Data Mining Framework[5] to execute the algorithms on the input data. The following steps were executed: (i) Attribute Statistic Analysis, (ii) Discretization, (iii) Data Sample, (iv) Classification Tree and CN2 execution, and (v) Test Learners. In the Attribute Statistic Analysis we want to check the distribution of data for each attribute. Before generating the decision tree and the rules with CN2, we have to evaluate the statistics about each attribute used in the model. For example, we can state that most of the failures occurred with activities that follow the MAP type. MAP activities are those receive one input and produce one output (*i.e.* similarly to the MapReduce model). On the other hand, activities that follow the SPLIT-MAP type presents no failures (and probably do not need to use an FTT). SPLIT-MAP activities receives one input data and produces several output. They represent a small number of activities in Montage and they are not cpu-intensive activities, so errors are - predictably - less frequent.

A discretization step is necessary because the goal attribute is the total execution time. Since this attribute is continuous, we have to discretize it. We have chosen to perform an equal-frequency discretization which produced 4 categories of activities according to the total execution time: SHORT-TERM, MEDIUM-TERM, LONG-TERM and XLONG-TERM. A common way to organize data for classifiers (in our case decision trees and CN2) is to split the available data into 2 sets, a training set and a test set. The predictive model is then built on the training set and evaluated with the test set. The test set has never been seen by the model so the resulting performance will be a good guide to what will be seen when the model is applied to unseen data. Thus, in the Data Sample step we used a traditional proportion for training/test data, having 70% for the training

[5] https://orange.biolab.si/.

set and 30% for the test. With the defined training set we executed the machine learning algorithms in step (iv). Both decision tree and CN2 produce a set of rules that can be used by the SWfMS to determine the best FTT to use for an activity. For example, the following rule was produced by both algorithms:

```
IF activity_name=sciphy.dataselection AND
    ft_type=retry AND
    n_tasks=>7.00
THEN D_class=SHORT-TERM
```

By analyzing this rule the SWfMS can conclude that if the *Retry* technique is chosen as FTT for the *dataselecion* activity of *SciPhy* when this activity comprises more than 7 jobs, the total execution time will be classified as SHORT-TERM, **i.e.** which is desirable. On the other hand, if we choose *Replication* FTT in this case, the total execution time is classified as MEDIUM-TERM, which is a worse scenario. One advantage of decision trees and CN2 is that both algorithms generate human-comprehensible rules that can be easily implemented within a SWfMS, in this case SciCumulus. Although the generated rules seem to make sense, we have to evaluate both predictive models using the test set generated in step (iii). Thus, in step (v) we evaluate both algorithms using the test set.

Results are presented in Table 2. Table 2 presents the Precision, Recall and F-Measure metrics for decision trees and CN2. It is clear that both models presents values higher than 0.8 for all metrics, which is a promising result. High Precision means both algorithms correctly classified most of the tuples in the test dataset, while high Recall means that both algorithms returned most of the relevant results (>80%). We also used the F-Measure to evaluate the results. F-Measure is the weighted average precision and recall. F-Measure reaches its best value at 1 and worst at 0. Since both F-Measures were higher than 0.80, we can conclude that both predictive models are correctly predicting the total execution time of a workflow according to the input variables in more than 80% of the cases. Although these results are promising, more experiments are needed with different types of the workflows in order to check if the generated rules are in fact useful. All predictive models and datasets used in this experiment can be downloaded at https://github.com/UFFeScience/FTT.

Table 2. Metrics obtained from predictive models

Algorithm	Precision	Recall	F-Measure
Classification tree	0.8502	0.8842	0.8669
CN2	0.8510	0.8779	0.8642

5 Conclusions and Final Remarks

In summary, this paper has the following main contributions: (i) the development of proactive failure detection approach in SciCumulus, (ii) the development of a

C/R FTT in SciCumulus that does not require changes into programs, (iii) the development of a Replication FTT in SciCumulus and (iv) the development of 2 predictive models that can be used by existing SWfMS to choose the suitable FTT for a specific workflow before its execution.

It is important to stress that some parameters in the experiments such as checkpointing interval were chosen in a purely empirical manner although some studies as Di *et al.* [25] and Young [26] derive an optimal checkpoint interval from failure probability and expected activation makespan. This work, though, is rather concerned about applicability and potential gains that could be obtained through the use of the different FTT than about minor performance degradation due to a poor choice of a parameter value. Thus, the interval was set as 30 s so it was possible to observe that, even for a not carefully chosen interval parameter it is possible to achieve good performance in the presence of faults.

As future work we plan to consider the execution traces of other workflows and to evaluate new classification algorithms such as SVM, Random Forest and Naive Bayes. Finally, we expect the results provided by this paper can help scientists in achieving their research goals in a reliable and fast way.

References

1. Mattoso, M., Werner, C., Travassos, G.H., Braganholo, V., Ogasawara, E., de Oliveira, D., et al.: Towards supporting the life cycle of large scale scientific experiments. IJBPIM **5**(1), 79+ (2010)
2. Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B., Good, J.: On the use of cloud computing for scientific workflows. In: eScience 2008, pp. 640–645 (2008)
3. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. SIGCOMM Rev. **39**(1), 50–55 (2008)
4. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., da Silva, R.F., Livny, M., et al.: Pegasus, a workflow management system for science automation. FGCS **46**, 17–35 (2015)
5. de Oliveira, D., Ogasawara, E., Baião, F., Mattoso, M.: Scicumulus: a lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In: 3rd International Conference on Cloud Computing, pp. 378–385 (2010)
6. Jackson, K.R., Ramakrishnan, L., Runge, K.J., Thomas, R.C.: Seeking supernovae in the clouds: a performance study. In: HPDC 2010, pp. 421–429. ACM, New York (2010)
7. Lee, K.-H., Lai, I.-C., Lee, C.-R.: Optimizing back-and-forth live migration. In: Proceedings of the 9th UCC, UCC 2016, pp. 49–54. ACM, New York (2016). https://doi.org/10.1145/2996890.2996909
8. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for computational tasks: a survey. Comput. Sci. Eng. **10**(3), 11–21 (2008)
9. Hu, M., Luo, J., Wang, Y., Veeravalli, B.: Adaptive scheduling of task graphs with dynamic resilience. IEEE Trans. Comput. **66**(1), 17–23 (2017)
10. Gu, Y., Wu, C.Q., Liu, X., Yu, D.: Distributed throughput optimization for large-scale scientific workflows under fault-tolerance constraint. J. Grid Comput. **11**(3), 361–379 (2013)

11. Bala, A., Chana, I.: Autonomic fault tolerant scheduling approach for scientific workflows in cloud computing. Concurr. Eng. **23**(1), 27–39 (2015)
12. Jain, A., Ong, S.P., Chen, W., Medasani, B., Qu, X., Kocher, M., Brafman, M., Petretto, G., Rignanese, G.-M., Hautier, G., et al.: Fireworks: a dynamic workflow system designed for high-throughput applications. Concurr. Comput. **27**(17), 5037–5059 (2015)
13. Elmroth, E., Hernández, F., Tordsson, J.: A light-weight grid workflow execution engine enabling client and middleware independence. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 754–761. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68111-3_79
14. von Laszewski, G., Hategan, M.: Java cog kit karajan/gridant workflow guide. Technical report, Argonne National Laboratory, Argonne, IL, USA (2005)
15. Costa, F., de Oliveira, D., Ocaña, K.A.C.S., Ogasawara, E., Mattoso, M.: Enabling re-executions of parallel scientific workflows using runtime provenance data. In: Groth, P., Frew, J. (eds.) IPAW 2012. LNCS, vol. 7525, pp. 229–232. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34222-6_22
16. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
17. Clark, P., Niblett, T.: The CN2 induction algorithm. Mach. Learn. **3**(4), 261–283 (1989)
18. Zhang, Y., Mandal, A., Koelbel, C., Cooper, K.: Combined fault tolerance and scheduling techniques for workflow applications on computational grids. In: CCGrid 2009, pp. 244–251. IEEE Computer Society (2009)
19. Hoheisel, A.: Grid workflow execution service-dynamic and interactive execution and visualization of distributed workflows. In: Proceedings of the Cracow Grid Workshop, vol. 2, pp. 13–24. Citeseer (2006)
20. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM CSUR **31**(1), 1–26 (1999)
21. Ocaña, K.A.C.S., de Oliveira, D., Ogasawara, E., Dávila, A.M.R., Lima, A.A.B., Mattoso, M.: SciPhy: a cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes. In: Norberto de Souza, O., Telles, G.P., Palakal, M. (eds.) BSB 2011. LNCS, vol. 6832, pp. 66–70. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22825-4_9
22. Saavedra-Barrera, R., Culler, D., Von Eicken, T.: Analysis of multithreaded architectures for parallel computing. In: SPAAACM 1990, pp. 169–178. ACM (1990)
23. Quinlan, J.R.: Simplifying decision trees. Int. J. Man-Mach. Stud. **27**(3), 221–234 (1987)
24. Ogasawara, E., Dias, J., Silva, V., Chirigati, F., de Oliveira, D., Porto, F., Valduriez, P., Mattoso, M.: Chiron: a parallel engine for algebraic scientific workflows. Concurr. Comput. **25**(16), 2327–2341 (2013)
25. Di, S., Robert, Y., Vivien, F., Kondo, D., Wang, C.-L., Cappello, F.: Optimization of cloud task processing with checkpoint-restart mechanism. In: 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12. IEEE (2013)
26. Young, J.W.: A first order approximation to the optimum checkpoint interval. Commun. ACM **17**(9), 530–531 (1974)

Energy Aware Multiobjective Scheduling in a Federation of Heterogeneous Datacenters

Santiago Iturriaga(✉) and Sergio Nesmachnow

Universidad de la República, Montevideo, Uruguay
{siturria,sergion}@fing.edu.uy

Abstract. Energy efficiency is key for datacenters, however nowadays datacenters are far from being energy efficient. This article proposes a multiobjective evolutionary approach for energy aware scheduling in a federation of heterogeneous datacenters. The proposed algorithm schedules workflows of tasks aiming at optimizing infrastructure usage, quality of service and energy consumption. We perform an extensive experimental evaluation with 100 problem instances, considering a diverse set of workflows and different size of scenarios. Results show the proposed approach is able to compute accurate schedules, outperforming traditional heuristic schedulers such as round robin or load balancing algorithm.

1 Introduction

Power consumption has become a critical issue in current high performance computing facilities [1]. This issue is even more problematic when considering large distributed platforms built by interconnecting several local resources. Datacenters are a common infrastructure for scientific high performance computing nowadays, as they provide computing power, networking, and storage for developing and execute complex applications [28]. A possible way for scaling up the number of resources is to build a federation of distributed datacenters, usually by integrating heterogeneous resources to deliver a huge amount of computing power and benefit end users.

When dealing with large heterogeneous computing systems, intelligent methods must be applied to guarantee a correct operation, from both the points of view of users and administrators. In this line of work, different techniques haver been proposed for reducing power consumption in datacenters and high performance computing facilities [5,9,20,28].

This article presents the application of a two-level scheduling approach that combines a MultiObjective Evolutionary Algorithm (MOEA) and specific ad-hoc backfilling heuristics for energy-aware planning of workloads into a federation of heterogeneous distributed datacenters, taking into account task's dependencies and quality of service (QoS) provided by the datacenter. The reported research extends our previous approach [8] by considering a fully heterogeneous approach for both workloads and computing elements, a realistic assumption for nowadays high performance computing infrastructures [23].

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 337–352, 2018.
https://doi.org/10.1007/978-3-319-73353-1_24

The experimental evaluation is performed over a set of 100 realistic problem instances considering both small- and large-sized scenarios. These problem instances are comprised of five different types of computing workflows, sampling a wide range of realistic high-performance applications. Each problem instance, disregarding its type, is comprised of 1000 computing workflows, each of which is comprised of a number of tasks ranging from 1 up to 132. The proposed MOEA scheduler is compared on an objective basis with a set of accurate greedy heuristics.

The main results indicate that the proposed MOEA outperforms the most accurate greedy heuristics with a makespan improvement of 33%, energy consumption improvement of 6%, and QoS improvement of 60%.

The article is organized as follows. Section 2 presents the problem formulation and reviews related works. The scheduling approach and the proposed MOEA to solve the problem is described in Sect. 3. The experimental evaluation is reported in Sect. 4, including a comparative analysis of the proposed methods. Finally, Sect. 5 presents the conclusions and the main lines for future work.

2 Energy-Aware Scheduling in a Federation of Datacenters

This section introduces the problem model and formulation, and reviews related works.

2.1 Problem Model

The energy-aware scheduling problem proposes allocating resources to parallel tasks to be executed on a federation of heterogeneous datacenters. The problem model is presented next.

Let us consider the following elements:

- A distributed federation formed by k heterogeneous datacenters $DC = \{dc_1, \ldots, dc_k\}$. Each datacenter dc_r is comprised of a set of heterogeneous multi-core servers $S_r = \{s_1, \ldots, s_s\}$ organized in racks. All servers inside a rack are identical but may be different among racks. Each server s_j is characterized by its number of cores c_j, its performance in FLOPS ops_j, and its power consumption at idle e_j^{idle} and peak e_j^{max} usage. Servers in each rack are networked using a top-of-the-rack (TOR) switch. A TOR switch networks together all servers in its rack with communication speed of rs_j, while an aggregation switch networks all the TOR switches with communication speed of as_j. Communication between tasks in the same server are considered to be instantaneous similar to the approaches proposed by [2,26,27].
- A set of n independent heterogeneous workflows $Q = \{q_1, \ldots, q_n\}$. Each workflow q has an associated soft deadline d_q before it should be accomplished and is comprised of a set of tasks $WT_q = \{wt_1, \ldots wt_m\}$ with dependencies. Each task has different computing requirements. After task wt_α finishes its execution in server s_β, it produces an output dataset that must be transferred to all

its successor tasks. Data is transferred instantaneously to successors executing in the same server s_β. However, for successors executing in other servers, data must be transferred over the network and is bound to communication speeds rs_j and as_j. Furthermore, task $\hat{wt_\alpha}$ is eligible for execution only after all its predecessors have finished executing and the dataset generated by them has been transferred.
- A number p of owners $O = \{o_1, \ldots, o_p\}$ of workflows to be executed and a SLA agreement that determines the percentage of applications that should be finished before their deadlines.
- Each task wt_α is characterized by the tuple $(o_\alpha, nc_\alpha, d_\alpha)$ defining its length (number of operations), the number of resources (cores) required for the parallel execution, and the networking time required for transferring its output dataset.

Two system-related objectives are used to take into account the point of view of the computing system, while a QoS-related objective takes into account the point of view of the users. The problem formulation proposes the simultaneous optimization of three functions:

- *makespan* evaluates the total time to execute a set of workflows, according to the expression in Eq. (1), where $\boldsymbol{x}$ represents a schedule and CT_r represents the completion time of datacenter dc_r.

$$f_M(\boldsymbol{x}) = \max_{dc_r \in DC} CT_r \tag{1}$$

  CT_r represents the time required for data center r to finish all its assigned tasks, this considers the execution time and networking time required for transferring the input data for each task.
- *energy consumption* for a set of workflows executed in a certain datacenter, defined by Eq. (2), using the energy model for multi-core architectures proposed in [19], where f_1 is the higher-level scheduling function, and f_2 is the lower-level scheduling function. The total energy consumption takes into account both the energy required to execute the tasks assigned to each computing resource within a datacenter, and the energy that each resource consumes in idle state.

$$f_E(\boldsymbol{x}) = \sum_{r \in DC} \sum_{\substack{q \in Q: \\ f_1(q)=r}} \sum_{\substack{wt_\alpha \in WT_q: \\ f_2(wt_\alpha)=s_j}} \frac{o_\alpha}{ops(s_j)} \times e_{s_j}^{max} + \sum_{s_j \in S_r} e_{s_j}^{idle} \tag{2}$$

- *SLA violations* is defined as the number of workflows that do not finish before their deadline, over the allowed limit specified by the SLA, according to Eq. (3), where $V(q)$ is 1 when the deadline of workflow q is violated and 0 otherwise, and $W(u_i)$ is the number of workflows submitted by user u_i.

$$f_S(\boldsymbol{x}) = \sum_{u_i \in U} \max\left(0, \sum_{q \in wo(u_i)} V(q) - (1 - SLA_{u_i}) \times W(u_i)\right) \tag{3}$$

2.2 Related Work

Two approaches are widely considered when dealing with energy-aware scheduling in computing systems: the independent and the simultaneous approach. The first, optimizes objectives as separate goals, not taking into account their relationships explicitly. The second, optimizes objectives at the same time considering a multi-constrained multi-objective optimization problem. Our proposed formulation follows the simultaneous approach. The main related works are reviewed next.

Some works focus on finding only one trade-off solution instead of a Pareto set. Following this approach, Khan and Ahmad [11] applied the cooperative game theory to find optimal trade-off schedules for independent tasks, simultaneously minimizing makespan and power consumption on a DVS-enabled grid system. Lee and Zomaya [13] studied DVS-based heuristics to minimize the weighted sum of makespan and energy, including a local search to modify the schedules and enhance makespan, if it does not imply increasing the power consumption. Mezmaz et al. [16] proposed a parallel bi-objective hybrid genetic algorithm (GA) for the same problem, using the cooperative island/multi-start farmer-worker model to reduce the execution time. A two-phase bi-objective algorithm based on Greedy Randomized Adaptive Search Procedure (GRASP) and applying a DVS bi-objective local search was proposed by Pecero et al. [21]. Li et al. [14] introduced a MinMin online dynamic power management strategy with multiple power-saving states to reduce energy consumption of scheduling algorithms. The approach by Pinel et al. [22] for scheduling independent tasks on grids with energy considerations, first applies MinMin to optimize makespan, and then a local search to minimize power consumption. Lindberg et al. [15] proposed six greedy algorithms and two GAs for solving the makespan-energy scheduling problem subject to deadline and memory requirements.

The energy consumption model for multi-cores in [19] is based on the power required: (*i*) to execute tasks at full capacity, (*ii*) when multi-core machines are partially used, and (*iii*) to keep machines in idle state. Fast scheduling methods for the bi-objective problem of minimizing the time and power consumption for Bag-of-Tasks applications were proposed. A similar approach was presented in [10] applying a parallel multiobjective local search that outperformed previous algorithms. DAG-based workflows were studied in [5] applying deterministic schedules to optimize makespan, power consumption, and QoS. Our previous work [8] studied evolutionary strategies and deterministic lower bounds for the scheduling problem on a federated cloud infrastructure composed by homogeneous processors and taking into account SLA agreements. This article extends the previous work [8] by considering a fully heterogeneous approach for both workloads and computing elements. To the best of our knowledge, no other work has addressed the scheduling of DAG-based workflows for simultaneously optimizing makespan, energy consumption and SLA objectives in heterogeneous datacenters.

3 The Proposed Two-Level Multiobjective Evolutionary Scheduler

This section describes the proposed methods for energy aware scheduling in a federation of heterogeneous datacenters.

3.1 Two-Level Scheduling Approach

The proposed two-level scheduling approach works by dividing the scheduling method into a high-level and a low-level scheduling method. In this approach, the high-level scheduling method deals with the scheduling of workflows to datacenters. This method considers each workflow as a single computing task and each datacenter as a single supercomputer. It does not deal with the scheduling of the actual tasks of each workflow, nor the individual servers in each datacenter. The low-level scheduling method, however, addresses the scheduling of the tasks of each workflow to the servers of its assigned datacenter. Next, we present the considered low-level scheduling heuristic and high-level scheduling MOEA.

3.2 Low-Level Scheduling Heuristic

We consider the Earliest Finishing Time Hole (EFTH) algorithm for addressing the low-level scheduling problem. EFTH is presented in our previous work and proved to be an efficient and accurate method [8]. It works by scheduling the workflows in the order established by the first level scheduler. That is, first it schedules all the tasks in the first workflow, then all the tasks in the second workflow, and so on. The scheduling of the tasks in the first workflow is performed exactly as HEFT would do. The next workflows are scheduled following the same strategy of HEFT, but applying a backfilling strategy. This backfilling strategy allows the tasks to be scheduled in the gaps (or holes) where processors are idle due to dependencies among tasks.

3.3 High-Level MOEA Scheduler

Evolutionary Algorithms (EAs) are nondeterministic methods based on the evolution of the species in nature. These algorithm have been successfully applied for addressing many optimization problems [18]. Multiobjective Evolutionary Algorithms (MOEAs) are EAs for solving multiobjective optimization problems [4]. MOEA explore multiple solutions simultaneously, thus are able to compute several trade-off solutions approximating the Pareto front of the problem in a single execution.

We designed the high-level MOEA scheduler based on the *Non-dominated Sorting Genetic Algorithm, version II* (NSGA-II) [4]. NSGA-II is a popular state-of-the-art MOEA, successfully applied to solve optimization problems in many application areas. Next, we present the main features of the proposed high-level MOEA scheduler.

Solution encoding. Solutions are encoded as a vector of integers ranging from 0 to $n + k - 1$, with n the number of workflows and k the number of datacenters. Integers $[0, n-1]$ represent workflows while integers $[n, n+k-1]$ act as a separators of workflows assigned to each datacenter. Figure 1 presents an example for this enconding where workflows 4, 5, and 2 are assigned to datacenter 1; workflows 0, 3, and 1 to datacenter 2, and so on. This encoding represents the workflows assigned to each datacenter and the order in which they must be scheduled by the low-level scheduler. Furthermore, it allows a simple implementation of variation operators such as reassigning workflows from one datacenter to another, changing the scheduling order of workflows, etc.

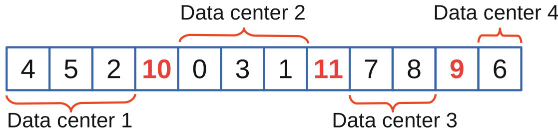


Fig. 1. Solution encoding ($n = 9$ workflows, $k = 4$ datacenters).

Fitness values. We define three objective functions, exactly as defined in Sect. 2.1. The quality of a solution is evaluated according to the NSGA-II fitness ranking based on non-domination sorting [4].

Population initialization. The initial population is created applying the high-level heuristics defined in Sect. 4.2. These heuristics are applied in random order until all initial solutions are created.

Selection. Binary tournament is applied for the selection operator, taking into account Pareto dominance and crowding distance.

Variation operators. Partially Matched Crossover (PMX) and Exchange Mutation (EM) are applied for combining and mutating solutions. PMX works by selecting two random positions in the selected solutions and swapping all values between them. The remaining values are rearranged using position wise exchanges, maintaining the ordering information. EM operator is much simpler than PMX and works by randomly selecting two values in a solution vector and swapping them.

Repair operator. It is used to transform non-feasible solutions, resulting from the variation operators, into feasible ones. The operator checks for every workflow if its assigned datacenter is capable of executing all of its tasks. If not, then the solution is repaired by reassigning unfeasible workflows to a random datacenters that can execute them.

Parameter configuration. NSGA-II is configured with population size of 100, stopping criterion of 25,000 evaluations, crossover probability of 0.9, and mutation probability of $1/n$ (with n number of workflows). These values were determined by an informal parameter setting analysis.

4 Experimental Evaluation

This section presents the experimental evaluation of the proposed scheduling methods.

4.1 Problem Instances

Each problem instance is defined by a *workload* of workflows and a *scenario* of available computing resources. This work considers the workflow models presented in [8]. In total five different models are considered: (a) *Series-Parallel* (b) *Heterogeneous-Parallel*, (c) *Homogeneous-Parallel*, (d) *Single-Task*, and (e) *Mix*. The Series-Parallel model represents workflows that can be split into concurrent processes. Heterogeneous-Parallel represent a generic workflow composed of non-identical computational tasks with arbitrary precedences. Homogeneous-Parallel represents workflows composed of identical computational blocks. Single-Task represents workflows comprised by a single task. Finally, the Mix model combines workflows from all other models with the following proportion: 30% of Series-Parallel, 30% of Heterogeneous-Parallel, 30% of Homogeneous-Parallel, and 10% of Single-Task workflows.

Workflow instances were generated using SchMng [24]. The deadline d_q of each workflow q is randomly generated by extending the completion time of the critical path of q by a ratio between $[0.05, 0.30]$. This is a realistic ratio according to [7,25]. Finally, networking communication time d_α for each task wt_α is randomly generated as a ratio between $[0.05, 0.50]$ of the execution time of wt_α, similar to [2,25].

We consider each data center to be organized in racks of servers, each rack containing 18–42 servers. We consider all processors in the same rack to be homogeneous. However, different racks may contain different types of processors. Processors in each rack are randomly chosen from a set of modern Intel processors ranging from 1 to 6 cores each. The candidate processors are presented in Table 1.

We define three realistic service level agreements, $SLA = \{90\%, 94\%, 98\%\}$. Each SLA represent the minimum ratio of workflows per user that must meet their deadlines. These are realistic values provided by current datacenter and cloud computing facilities, similar to the ones considered in the related literature [3,12,17].

A total of 50 workflow batches were created for the experimental evaluation, 10 for each of the 5 workflow types. Each batch is comprised of a total of 1000 workflows, with the number of tasks in each workflow ranging from 3 to 132, except for the Single-Task workflows that are comprised of just one task. In total, 50,000 workflows are studied in the experimental analysis. For the datacenter

Table 1. Processors considered for the datacenter scenario

Processor	Frequency	Cores	GFLOPS	E_{IDLE}	E_{MAX}
Intel Celeron 430	1.80 GHz	1	7.20	75.0 W	94.0 W
Intel Pentium E5300	2.60 GHz	2	20.80	68.0 W	109.0 W
Intel Core i7 870	2.93 GHz	4	46.88	76.0 W	214.0 W
Intel Core i5 661	3.33 GHz	2	26.64	74.0 W	131.0 W
Intel Core i7 980 XE	3.33 GHz	6	107.60	102.0 W	210.0 W

scenario we consider two instances: a small- and a large-sized scenario. Both involving a federation of five datacenters with up to 3 racks each. The small-sized scenario is comprised by an average of 150 processors per datacenter, and the large-sized instance by an average of 325 processors. Racks are communicated by 1 GB or 10 GB ethernet networks.

Overall, 100 problem instances were evaluated, considering all workflows and scenarios. The benchmark set of workflows, scenarios, and SLA levels is publicly available, it can be accessed/downloaded by contacting the authors.

4.2 High-Level Scheduling Heuristics

We consider seven different heuristic scheduling algorithms. These algorithms work by iteratively applying greedy decisions until a full schedule is constructed. Each iteration a workflow is selected and scheduled following some heuristic knowledge that varies for each heuristic. This process is repeated until all workflows are scheduled. A short description of the scheduling algorithms is presented next.

1. *Round Robin* (RR): Given a list of workflows and datacenters, this algorithm assigns the first workflow to the first datacenter, the second workflow to the second datacenter, and so on. Once the last datacenter is reached, the assignment continues with the first datacenter, following a circular strategy. If a datacenter is unable to satisfy a workflow requirements, then it is skipped and the workflow is assigned to the next datacenter suitable for it.
2. *Load Balance* (LB): This algorithm aims for a balanced workflow assignment. To accomplish this, workflows are first sorted according to the maximum number of cores required by any of their tasks. This way, workflows requiring more cores are prioritized for scheduling. Workflows are scheduled one at a time in the sorting order to the datacenter with the lowest number of assigned workflows.
3. *MaxMin*: This algorithm assigns each workflow to the datacenter that can execute it faster. In each iteration, the workflow with the maximum estimated completion time is selected and assigned to a datacenter that can finish it earlier, considering its core requirements. In this case, the completion time of each workflow is estimated with the sum of the execution time of the tasks in

its critical path. This procedure continues until all workflows are assigned. As a result, longer workflows are scheduled before shorter workflows, following the heuristic knowledge that scheduling longer workflows first produce a more balanced schedule.

4. *MaxMIN*: Just like MaxMin, this algorithm schedules first the workflows with longest completion estimation time, considering the completion estimation time of all the other workflows scheduled in the datacenters. However, workflows are scheduled to the datacenter that minimizes their overall energy consumption estimation. That is, MaxMIN schedules the longest workflows first to the datacenter consuming the minimum amount of energy. Again, energy consumption estimation and execution time estimation in this heuristic is based on the total execution time of the tasks in its critical path.
5. *MinMIN*: Similar to MaxMIN, but in this case in each iteration of the algorithm, the workflow that requires the overall minimum completion time is selected first and assigned to a datacenter with the lowest energy consumption.
6. *Core-Aware MaxMin* (CA-MaxMin): The CA-MaxMin works exactly as MaxMin, but estimates the overall completion time with the execution time of the tasks in its critical path multiplied by the number of cores required by all of its tasks. This new estimator prioritizes the scheduling of workflows with heavy processing requirements.
7. *Longest First* (LF): Finally, the LF is a simple algorithm based on the LB heuristic. Both algorithm are identical, except initial workflow sorting is performed differently. The LF heuristic sort workflows are sorted considering the product of their critical path execution time, the sum of the execution time of all of the tasks in the workflow, and the sum of the total number of cores required by all of its tasks. This estimator schedules first the most computing demanding workflows, balancing these workflows adequately among the data centers.

The RR, LB, MaxMin, MaxMIN, and MinMIN algorithm were all introduced in [8], while CA-MaxMin and LF were newly designed for this work. Furthermore, because the EFTH low-level heuristic does not deal with workflow ordering, we propose to combine the proposed high-level heuristics with a workflow sorting algorithm. That is, after the scheduling algorithm assigns each workflow to a datacenter, a sorting algorithm optimizes the ordering of workflows in each datacenter. The rationale for this is that the best ordering for the assignment of workflows to datacenters may not be the same as the best ordering for the execution of the workflows in each datacenter. The considered sorting criteria are presented next.

1. *Unsorted* (U): Applies no sorting algorithm. Workflows remain in the order the workflow assignment process produced.
2. *Average Cores per level* (AC): Sorts workflows according to the average number of total cores per level of the workflow graph.

3. *Maximum Cores* (MC): Sorts workflows according to the number of cores required by the task that requires the most cores. Sorting is untied by considering the length of the critical path of the workflow.
4. *Computing Load* (L): Sorting is performed based on the total execution time of the tasks in the critical path of the workflow multiplied by the average number of cores per level of the workflow graph.

Overall, we considered a total of 56 high-level scheduling heuristics, considering the combination of each scheduling algorithm with each sorting criterion in ascending (ASC) and descending (DSC) direction. For simplicity, from now on heuristics will be referred using the nomenclature: *scheduling algorithm* + *sorting criterion* + *sorting direction*. For example, the heuristic comprised by the RR algorithm, with AC sorting in ascending direction is referred as: RR+AC+ASC.

4.3 Development and Execution Platform

The proposed high-level heuristics were implemented Java. The high-level MOEA was also implemented in Java using the jMetal framework [6]. The low-level heuristic was implemented in C and compiled using the GNU C compiler. All the experiments were performed in ClusterFING, the HPC facility of Universidad de la República, Uruguay (platform website: https://www.fing.edu.uy/cluster).

4.4 Numerical Results

This section reports the results computed by the proposed methods.

High-level heuristic scheduler. This subsection presents a comparison between the proposed high-level scheduling heuristics. Since these schedulers are single-objective algorithms, the comparison is performed separately for each problem objective using the gap metric.

The gap metric of a given objective for some scheduler is defined as the unity-based normalization of the objective value. Equation 4 presents the gap metric for scheduler s, with v_s the value for the objective computed by s, and v_w the worst value and v_b the best value computed by any scheduler. Because all considered objectives are minimization objectives, the smaller the gap value the better the result. That is, when $gap = 0$ then s is the scheduler computing the best schedule for that objective. We define the gap_M, gap_E, and gap_S for measuring the gap of the makespan, energy consumption, and SLA violations objectives respectively.

$$gap = \frac{v_s - v_b}{v_w - v_b} \tag{4}$$

We perform a statistical analysis over the results of the proposed scheduling heuristics for determining the most accurate heuristics for each objective. First, normality of the computed values is rejected with $p - value \leq 0.0077$

after applying a Kormogorov-Smirnov test for normality. After discarding normality, the Kruskal-Wallis test is applied to test the equality of the medians of the results. The Kruskal-Wallis test shows significant differences between the medians with a $p - value \leq 0.0001$. Hence, there are significant differences in the accuracy of at least one of the heuristics. The Dunn's post hoc test is used for pinpointing pairwise differences between the results of the heuristics. The pairwise comparison shows no heuristic is significantly more accurate than all the rest for any objective. However, a ranking of the pairwise differences (with $p - value \leq 0.05$) shows the overall best performing scheduling algorithms are CA-MaxMin and LF.

Table 2 presents the average and standard deviation gap computed by the best performing scheduling heuristics for each problem objective, with the most accurate results presented in gray. The Dunn's test shows CA-MaxMin+U, CA-MaxMin+AC+ASC, CA-MaxMin+MC+ASC, and CA-MaxMin+L+ASC are all significantly more accurate than around 84% of the remaining heuristics when optimizing makespan. LF+AC+DSC and LF+MC+DSC are significantly more accurate than around 87% of the remaining heuristics when optimizing the energy consumption objective. And finally, CA-MaxMin+L+DSC is significantly more accurate than around 70% of the remaining heuristics when optimizing the SLA violations.

Table 2. Average and standard deviation gap values of the most accurate high-level scheduling heuristics for all the problem instances.

scheduling algorithm	*sorting criterion*	*sorting direction*	gap_M	gap_E	gap_S
CA-MaxMin	U	none	0.03±0.04	0.63±0.06	0.76±0.16
	AC	ASC	0.03±0.04	0.63±0.06	0.76±0.16
		DSC	0.09±0.04	0.65±0.06	0.23±0.07
	MC	ASC	0.03±0.04	0.63±0.06	0.76±0.16
		DSC	0.14±0.05	0.68±0.07	0.31±0.08
	L	ASC	0.03±0.04	0.63±0.06	0.76±0.16
		DSC	0.12±0.04	0.71±0.06	0.05±0.04
LF	U	none	0.90±0.08	0.16±0.08	0.90±0.06
	AC	ASC	0.90±0.08	0.16±0.08	0.90±0.06
		DSC	0.89±0.07	0.01±0.02	0.71±0.10
	MC	ASC	0.90±0.08	0.16±0.08	0.90±0.06
		DSC	0.97±0.04	0.03±0.02	0.78±0.11
	L	ASC	0.90±0.08	0.16±0.08	0.90±0.06
		DSC	0.92±0.05	0.08±0.04	0.65±0.07

Best results for each objective are marked in gray ($p - value \leq 0.05$).

The most accurate heuristics are selected for comparison with the high-level NSGA-II scheduler. Several heuristics showed to be equally accurate for

optimizing the makespan objective. Hence, following the Occam's razor principle we select CA-MaxMin+U for the comparison. Again, LF+AC+DSC and LF+MC+DSC showed no significant difference between them for optimizing energy consumption. However, LF+AC+DSC shows a better average gap metric than LF+MC+DSC. Hence, LF+AC+DSC is selected for comparison. Finally, CA-MaxMin+L+DSC is selected for comparing of the SLA violations objective. For simplicity, from now on these heuristics will be simply referred as *Makespan heuristic*, *Energy heuristic* and *SLA heuristic* respectively.

High-level NSGA-II scheduler. This subsection summarizes the comparison between the NSGA-II scheduler and the most accurate heuristic schedulers. For the comparison, a total of 30 independent NSGA-II executions were performed for each problem instance.

Table 3 shows the average improvement and standard deviation for the best solution obtained by NSGA-II for each objective comparing with the solution obtained by the best heuristic for that objective. Overall results show the NSGA-II outperforms the best heuristics, improving–in average–the best heuristics by up to 33% in makespan, 6% in energy consumption and 30% in SLA violations. Nevertheless, there is significant variations in the accuracy of the NSGA-II results depending on the instance type and scenario size.

Table 3. Average and standard deviation of improvement of the NSGA-II scheduler for each objective when compared to the best heuristic scheduler

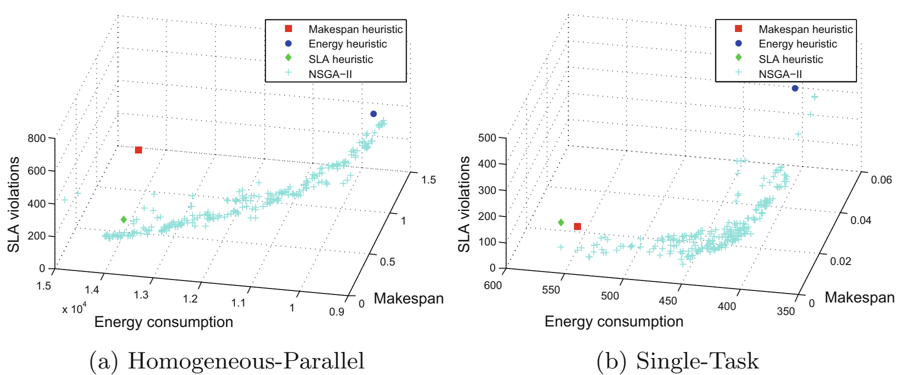
Instance type	Improvement over the best heuristic		
	f_M	f_E	f_S
Small-sized scenarios			
Heterogeneous-Parallel	$9.4 \pm 4.3\%$	$9.8 \pm 0.6\%$	$0.0 \pm 1.6\%$
Homogeneous-Parallel	$16.0 \pm 4.2\%$	$13.0 \pm 0.5\%$	$-1.7 \pm 2.0\%$
Serial-Parallel	$8.3 \pm 4.2\%$	$9.4 \pm 0.3\%$	$1.0 \pm 0.8\%$
Single-Task	$57.0 \pm 5.9\%$	$8.4 \pm 0.7\%$	$90.0 \pm 2.5\%$
Mix	$11.0 \pm 3.2\%$	$12.0 \pm 3.1\%$	$-0.7 \pm 3.0\%$
Large-sized scenarios			
Heterogeneous-Parallel	$44.0 \pm 2.9\%$	$2.9 \pm 0.9\%$	$5.9 \pm 2.2\%$
Homogeneous-Parallel	$42.0 \pm 8.6\%$	$1.4 \pm 0.5\%$	$63.0 \pm 9.3\%$
Serial-Parallel	$48.0 \pm 2.9\%$	$2.4 \pm 0.7\%$	$4.8 \pm 2.4\%$
Single-Task	$57.0 \pm 10.0\%$	$2.3 \pm 0.8\%$	$99.0 \pm 0.9\%$
Mix	$40.0 \pm 3.3\%$	$4.0 \pm 1.2\%$	$41.0 \pm 23.0\%$

For the small-sized scenarios, NSGA-II produces the best makespan and SLA violations improvements when dealing with Single-Task instances, and produces the best energy consumption improvements when dealing with

Homogeneous-Parallel instances. Furthermore, NSGA-II is able to consistently improve makespan and energy consumption objectives for all instances. However, although NSGA-II improves SLA violations by up to 90% when dealing with Single-Task instances, it is unable to improve or even worsens SLA violations for the remaining type of instances. This is because small-sized instances are heavily loaded instances with a large number of workflows and a small number of available computing resources. This makes it difficult for NSGA-II to improve SLA violations while also improving makespan and energy consumption.

NSGA-II behaves differently when dealing with large-sized scenarios. For these scenarios, NSGA-II is able to improve makespan consistently, with an average improvement of 40%–57% for all instances. Furthermore, SLA violations are also consistently improved on all instances, no longer worsening the heuristics results, and largely improving SLA violations for the Homogeneous-Parallel instances when compared to small-sized scenarios. However, improvement on energy consumption drops from 10% in average for small-sized scenarios to 2% in average for large-sized scenarios.

Overall, results show NSGA-II consistently improves heuristics results for the Single-Task instances in both scenarios and for all objectives, while the remaining type of instances present diverse results depending on the objective. On the one hand, NSGA-II consistently improves makespan in all scenarios. On the other hand, improvements of energy consumption and SLA violations are not consistent. Energy consumption is most improved for small-sized scenarios, with little improvement for large-sized scenarios. While SLA violations are most improved for large-sized scenarios, with negligible improvements for small-sized scenarios except when dealing with Single-Task instances. Figure 2 present samples of Pareto fronts computed by NSGA-II and schedules computed by each of the best heuristics.



(a) Homogeneous-Parallel (b) Single-Task

Fig. 2. Sample results computed by the best heuristics and NSGA-II for large-sized scenarios

5 Conclusions

In this paper we propose a multiobjective formulation for modeling the scheduling of a large number of workflows in a federation of datacenters to simultaneously minimize three objectives: makespan, energy consumption, and number of jobs violating a SLA threshold. This formulation extends the formulation proposed in [8] by considering heterogeneous datacenters and networking communication. This new formulation provides a more realistic modeling for nowadays datacenters.

We consider a two-level hierarchical scheduling approach to address the proposed problem. The high-level algorithm schedules workflows to datacenters, while the low-level algorithm schedules the tasks of the workflows assigned to each datacenter to the servers in that datacenter. This two-level strategy simplifies the scheduling problem by dividing the whole problem into two sub-problems.

Regarding the high-level scheduler, we proposed a total of 56 heuristic algorithms for online scheduling, and a MOEA based on NSGA-II for offline scheduling. We studied and compared the accuracy of all the proposed high-level schedulers considering a set of 100 diverse and realistic problem instances.

The analysis of the proposed high-level heuristics shows CA-MaxMin+U, CA-MaxMin+AC+ASC, CA-MaxMin+MC+ASC, and CA-MaxMin+L+ASC are the most accurate heuristics for makespan optimization. LF+AC+DSC and LF+MC+DSC are the most accurate heuristics for energy consumption optimization, and CA-MaxMin+L+DSC is the most accurate for SLA violations optimization.

The proposed NSGA-II proved to be an accurate method for addressing the proposed scheduling problem. In average, NSGA-II is able to improve makespan by 20%, energy consumption by 10% and SLA violations by 17% over the best heuristics for the small-sized scenarios. For the large-sized scenarios, NSGA-II improves makespan by 46%, energy consumption by 2% and SLA violations by 42% in average. Furthermore, NSGA-II is able to compute a diverse set of trade-off schedules with different levels of compromise between all three objectives.

The main line of future work consists in comparing the NSGA-II scheduler with other MOEA to analyze the accuracy of its computed Pareto front. On top of that, we propose to develop an mixed integer programming solution for computing exact lower-bounds for studying the optimality gap for each objective. Finally, we propose to evaluate our proposed methods with real-world workflows and datacenters to further study the behavior of the computed results.

References

1. Ahmad, I., Ranka, S.: Handbook of Energy-Aware and Green Computing. Chapman & Hall/CRC, Boca Raton (2012)
2. Chen, S., Li, Z., Yang, B., Rudolph, G.: Quantum-inspired hyper-heuristics for energy-aware scheduling on heterogeneous computing systems. IEEE Trans. Parallel Distrib. Syst. **27**(6), 1796–1810 (2016)

3. de Assuncao, M., di Costanzo, A., Buyya, R.: Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters, pp. 141–150 (2009)
4. Deb, K.: Multi-objective Optimization Using Evolutionary Algorithms. Wiley, Chichester (2001)
5. Dorronsoro, B., Nesmachnow, S., Taheri, J., Zomaya, A.Y., Talbi, E.-G., Bouvry, P.: A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. Sustain. Comput.: Inform. Syst. **4**(4), 252–261 (2014)
6. Durillo, J., Nebro, A.: jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011)
7. Garg, R., Kumar Singh, A.: Energy-aware workflow scheduling in grid under QoS constraints. Arab. J. Sci. Eng. **41**(2), 495–511 (2016)
8. Iturriaga, S., Dorronsoro, B., Nesmachnow, S.: Multiobjective evolutionary algorithms for energy and service level scheduling in a federation of distributed datacenters. Int. Trans. Oper. Res. **24**(1–2), 199–228 (2017)
9. Iturriaga, S., Nesmachnow, S.: Multiobjective scheduling of green-powered datacenters considering QoS and budget objectives. In: IEEE Innovative Smart Grid Technologies Latin America, pp. 570–573 (2015)
10. Iturriaga, S., Nesmachnow, S., Dorronsoro, B., Bouvry, P.: Energy efficient scheduling in heterogeneous systems with a parallel multiobjective local search. Comput. Inform. J. **32**(2), 273–294 (2013)
11. Khan, S., Ahmad, I.: A cooperative game theoretical technique for joint optimization of energy consumption and response time in computational grids. IEEE Trans. Parallel Distrib. Syst. **20**, 346–360 (2009)
12. Kim, K., Buyya, R., Kim, J.: Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters. In: 7th IEEE International Symposium on Cluster Computing and the Grid, pp. 541–548 (2007)
13. Lee, Y., Zomaya, A.: Energy conscious scheduling for distributed computing systems under different operating conditions. IEEE Trans. Parallel Distrib. Syst. **22**, 1374–1381 (2011)
14. Li, Y., Liu, Y., Qian, D.: A heuristic energy-aware scheduling algorithm for heterogeneous clusters. In: 15th International Conference on Parallel and Distributed Systems, pp. 407–413 (2009)
15. Lindberg, P., Leingang, J., Lysaker, D., Khan, S., Li, J.: Comparison and analysis of eight scheduling heuristics for the optimization of energy consumption and makespan in large-scale distributed systems. J. Supercomput. **59**(1), 323–360 (2012)
16. Mezmaz, M., Melab, N., Kessaci, Y., Lee, Y., Talbi, E.G., Zomaya, A., Tuyttens, D.: A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. J. Parallel Distrib. Comput. **71**, 1497–1508 (2011)
17. Moon, H., Chi, Y., Hacigümüş, H.: Performance evaluation of scheduling algorithms for database services with soft and hard SLAs. In: 2nd International Workshop on Data Intensive Computing in the Clouds, pp. 81–90 (2011)
18. Nesmachnow, S.: An overview of metaheuristics: accurate and efficient methods for optimisation. Int. J. Metaheuristics **3**(4), 320–347 (2014)
19. Nesmachnow, S., Dorronsoro, B., Pecero, J.E., Bouvry, P.: Energy-aware scheduling on multicore heterogeneous grid computing systems. J. Grid Comput. **11**(4), 653–680 (2013)
20. Nesmachnow, S., Perfumo, C., Goiri, Í.: Holistic multiobjective planning of datacenters powered by renewable energy. Cluster Comput. **18**(4), 1379–1397 (2015)

21. Pecero, J., Bouvry, P., Fraire, H., Khan, S.: A multi-objective grasp algorithm for joint optimization of energy consumption and schedule length of precedence-constrained applications. In: International Conference on Cloud and Green Computing, pp. 1–8 (2011)
22. Pinel, F., Dorronsoro, B., Pecero, J., Bouvry, P., Khan, S.: A two-phase heuristic for the energy-efficient scheduling of independent tasks on computational grids. Cluster Comput. **16**(3), 421–433 (2013)
23. Ren, Z., Zhang, X., Shi, W.: Resource scheduling in data-centric systems. In: Khan, S.U., Zomaya, A.Y. (eds.) Handbook on Data Centers, pp. 1307–1330. Springer, New York (2015). https://doi.org/10.1007/978-1-4939-2092-1_46
24. Taheri, J., Zomaya, A., Khan, S.: Grid simulation tools for job scheduling and datafile replication. In: Scalable Computing and Communications: Theory and Practice (Chap. 35), pp. 777–797. Wiley, Hoboken (2013)
25. Tang, Z., Qi, L., Cheng, Z., Li, K., Khan, S., Li, K.: An energy-efficient task scheduling algorithm in DVFS-enabled cloud environment. J. Grid Comput. **14**, 55–74 (2016)
26. Wang, Y.-R., Huang, K.-C., Wang, F.-J.: Scheduling online mixed-parallel workflows of rigid tasks in heterogeneous multi-cluster environments. Future Gener. Comput. Syst. **60**, 35–47 (2016)
27. Zhu, Z., Zhang, G., Li, M., Liu, X.: Evolutionary multi-objective workflow scheduling in cloud. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1344–1357 (2016)
28. Zomaya, A., Khan, S.: Handbook on Data Centers. Springer, New York (2014). https://doi.org/10.1007/978-1-4939-2092-1

Markov Decision Process to Dynamically Adapt Spots Instances Ratio on the Autoscaling of Scientific Workflows in the Cloud

Yisel Garí[1(✉)], David A. Monge[1,2], Cristian Mateos[4], and Carlos García Garino[1,3]

[1] ITIC-CONICET, Universidad Nacional de Cuyo (UNCuyo), Mendoza, Argentina
ygari@uncu.edu.ar
[2] Facultad de Ciencias Exactas y Naturales, UNCuyo, Mendoza, Argentina
[3] Facultad de Ingeniería, UNCuyo, Mendoza, Argentina
[4] ISISTAN-UNICEN-CONICET, Tandil, Buenos Aires, Argentina

Abstract. Spot instances are extensively used to take advantage of large-scale Cloud infrastructures at lower prices than traditional on-demand instances. Autoscaling scientific workflows in the Cloud considering both spot and on-demand instances presents a major challenge as the autoscalers have to determine the proper amount and type of virtual machine instances to acquire, dynamically adjusting the number of instances under each pricing model (spots or on-demand) depending on the workflow needs. Under budget constraints, this adjustment is performed by an assignment policy that determines the suitable proportion of the available budget intended for each model. We propose an approach to derive an adaptive budget assignment policy able to reassign the budget at any point in the workflow execution. Given the inherent variability of the resources in a Cloud, we formalize the described problem as a Markov Decision Process and derive adaptive policies based on other baseline policies. Experiments demonstrate that our policies outperform all the baseline policies in terms of makespan and most of them in terms of cost. These promising results encourage the future study of new strategies aiming to find optimal budget policies applied to the execution of workflows on the Cloud.

1 Introduction

Scientific workflows have been widely used to model complex experiments in many science disciplines such as Geosciences, Astronomy and Bioinformatics. Therefore, they often involve experiments with a large number of tasks and many hours of computation. The Cloud Computing paradigm facilitates the acquisition of computing infrastructures based on virtualization technologies [1]. In public Clouds, users can access a wide spectrum of hardware and software configurations offered by several types of Virtual Machine (VM) instances under a pay-per-use scheme.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 353–369, 2018.
https://doi.org/10.1007/978-3-319-73353-1_25

Public Cloud providers commonly offer at least two pricing models: on-demand instances and spot instances. In the first model, *on-demand instances* can be purchased for a fixed price, typically charged by the number of used hours. In the second model, the prices of *spot instances* fluctuate –i.e. decreasing during low demand periods– and the user must bid the highest price that he is willing to pay for each instance. Spot prices can show reductions of more than 50% with respect to the on-demand prices[1]. Thus, spot instances seem very attractive to optimize the trade-off between cost and performance of an infrastructure, but they in turn compromise cost and reliability. If the spot price overcomes the user's bid, an out-of-bid error is produced terminating the affected instances and interrupting the tasks that may be running.

Autoscaling strategies [2,3] dynamically scale the infrastructure to fulfill the workflows variable workload patterns and also efficiently schedule the workflow tasks in the available instances. Autoscaling using spot instances requires to determine, on each scaling step, the right proportion of spot and on-demand instances. Under budget constraints, this decision is crucial as we can exploit the huge parallelism potential offered by spot instances without incurring on costs that violate the constraints. In this context, we need to endow autoscalers with a policy that properly assigns the available budget for the acquisition of on-demand or spot instances according to the variable workflow computational requirements. Deriving such a policy is not trivial due to the difficulty of estimating tasks running-times [4]. Even more, spot prices are also very difficult to predict making very hard to anticipate the occurrence of out-of-bid errors [5].

Although many scientific workflows can be benchmarked [6,7] to provide accurate information about the tasks workload, it would not be feasible to benchmark each new workflow that may appear due to the effort and time implied in a benchmarking process. Then, we considered the workload as a source of uncertainty, according to the inherent variability of the resources in a Cloud and the final objective of being able to perform well with completely unknown workflows.

This problem can be seen as a problem of *decision making under uncertainty*, which can be defined as a Markov Decision Process (MDP). By defining the problem in such way we are able to derive new budget assignment policies from simpler policies, which can decide the most convenient budget assignment based on the current execution progress and resources availability minimizing workflow makespan under budget constraints. The contributions of this paper are:

– We formalize (Sect. 3) the problem of obtaining an adaptive budget assignment policy by casting such problem to an MDP. We describe the key elements required to define an MDP and how we define them for the problem at hand.
– We incorporated different fixed, random and adaptive policies in a start-of-the-art autoscaler called Spot Instances Aware Autoscaling (SIAA) that pursues workflow makespan minimization subject to budget constraints by combining spot and on-demand instances. We proposed this autoscaler in our previous works [3,8], being the first of them published in a earlier edition of this conference.

[1] Amazon EC2 spot instances. https://aws.amazon.com/ec2/spot/pricing/.

– We assessed the performance of the MDP adaptive policies in comparison with other simpler policies used as baseline, in terms of makespan and cost for a well-known workflow benchmark from the area of Geosciences: CyberShake [6]. Details of the experimental settings and results are discussed on Sect. 5.

Then, Sect. 6 exposes the most relevant related works in the area of MDPs and the execution of Cloud-based applications. Finally, Sect. 7 concludes this work and delineates future research lines.

2 Workflow Autoscaling

Workflow applications include a set of reusable software components denominated *tasks* which have dependency relationships. A widely-used way to model workflows is through a Directed Acyclic Graph (DAG), where tasks are represented by nodes, and task dependencies are represented by edges in the graph. In particular, scientific workflows, usually comprise hundred or thousands of tasks, whose type and duration may vary, involving several calculation hours. In this work we assume that there is a central storage for data and that transfer operations are carried out as part of the execution of tasks. Also, there is a wide spectrum of VM instances types and pricing models available.

Given an input workflow, autoscaling strategies regard two interrelated subproblems: (a) determine the proper number and type of VM instances, and (b) scheduling the workflow tasks on the available instances. Both sub-problems are NP-hard and therefore the solutions proposed to date are based on heuristics [2, 8]. To adapt the execution to the workflow requirements and mitigate the effect of the discrepancies between the estimations and the actual progression of the execution, autoscaling strategies are executed periodically [3].

The workflow autoscaling problem using a mixed infrastructure, namely comprising both on-demand and spot instances, poses a new important issue on each scaling step: how to split the budget between both pricing models and what is the effect on makespan? Usually, this balance is governed in autoscalers by the spots ratio parameter ($\rho \in [0, 1]$), which given the original budget B sets the maximum amount of money available for on-demand and spot instances, respectively. Then, the budget for on-demand instances is $B^{\text{od}} = (1 - \rho) \cdot B$ and the budget for spot instances is computed as $B^{\text{s}} = \rho \cdot B$. Periodically, the autoscaling strategy addresses a makespan minimization problem subject to these budget constraints.

Given T (the set of tasks in the workflow), I^{type} (the set of available VM types offered by the Cloud provider) and I^{scheme} (the two pricing models, i.e. spot instances and on-demand instances), such optimization problem is defined as:

$$\begin{aligned} &\min\{\text{makespan}(X^{\text{sca}}, X^{\text{sch}}, S)\} \qquad (1)\\ &\text{s.t.}: \quad \text{cost}_{\text{od}}(X^{\text{sca}}) \leq B^{\text{od}}, \text{cost}_{\text{s}}(X^{\text{sca}}, U^{\text{bid}}) \leq B^{\text{s}}, \end{aligned}$$

where $X^{\text{sca}} = \{I^{\text{type}} \times I^{\text{scheme}} \rightarrow \mathbb{N}\}$ define a *scaling plan* that indicates the number of necessary on-demand and spot instances of each VM type for the

next hour of computation, $X^{\text{sch}} = \{T \rightarrow I^{\text{type}}\}$ is a *scheduling plan* that maps each task $t \in T$ to a type of instance where it will execute, S represents the current status of the infrastructure and the workflow. The $U^{\text{bid}} = \{I^{\text{type}} \rightarrow \mathbb{R}^{+}\}$ is the selected bid price for each VM type; bid prices are provided by the user or by a bidding prediction method [9,10]. The cost_{od} and cost_{s} functions return the cost of the on-demand and spots instances respectively, according with the scaling plan. Finally, $B^{\text{od}}$and B^{s} represent the split budget for the next hour of computation.

In this way, autoscaling strategies aim to adapt the number of required instances of each pricing model considering the split budget constraint and minimize workflow makespan by periodically solving the optimization problem presented on Eq. (1).

2.1 Problem Definition

Scientific workflows present different dependency structures and workload patterns. Thus, during the execution of a workflow, there are stages with many tasks that can be executed in parallel and others in which execution must be performed in a sequential way. Spot instances can help us to increase the infrastructure during the high workload stages reducing makespan with low costs, but such instances involve a compromise between cost and reliability. Depending on the context, it might be convenient to use more spot instances than on-demand instances or viceversa.

A fixed budget assignment policy, which keep spots ratio parameter constant along the entire workflow execution, does not allow to exploit the entire potential of a mixed infrastructure, reducing the explored makespan-cost balance combinations. Moreover, a random policy could help us to explore the solutions space in a broader range, but lacks a sense of optimality because of its random nature. Then, in this paper, we address the problem of finding an optimal adaptive budget assignment policy that improves the adaptability of autoscaling strategies. Our approach is able to derive an adaptive policy from the feedback obtained from previous executions using other policies used as baseline, i.e. fixed and random. The feedback represents the state of the workflow execution and the infrastructure on each autoscaling period.

Figure 1 shows an example of an adaptive and a fixed budget assignment policy taken from the experiments presented in Sect. 5.2. The workflow execution steps are represented starting on the initial state and ending in the terminal state. Circles represent periodical scaling steps and links the spots ratio values taken. Inside the circles, it can be seen the number of spot instances (on top), on-demand instances (bottom) and between them the accumulated execution cost on each transition. As can be observed the adaptive policy almost doubles the number of spot instances acquired compared to the fixed policy, allowing it to execute more tasks in parallel. Notice that the adaptive policy causes the workflow execution to finish one step earlier (means makespan reduction) than the fixed policy.

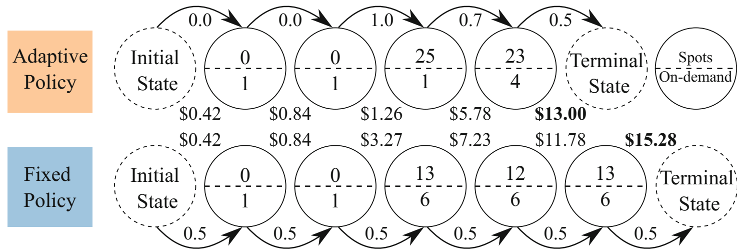


Fig. 1. Example of an adaptive (on top) and a fixed budget assignment policy (bottom). The cost of the entire execution is highlighted in bold font.

3 Budget Assignment as an MDP

Markov Decision Process [11] is an intuitive and fundamental formalism for problems of decision making under uncertainty. An MDP comprises a set of states and a set of possible actions to take on each state, with the goal of determining a sequence of actions that minimize/maximize some performance criterion. MDPs have become the de facto standard formalism for learning sequential decision making [12] and it has been applied to Cloud Computing problems [10,13,14].

Finding an optimal adaptive budget assignment policy in the context of Cloud workflow autoscaling is similar to an intelligent agent making decisions in a stochastic environment. On each scaling step the agent has to choose the proper budget assignment action based on the current state of the workflow execution and the infrastructure. The agent utility depends on a sequence of decisions and from each one it achieves a specific local progress related to the amount of work done. This progress can be seen as a reward. Finally, the agent goal is to produce an optimal behavior that balances the risks and rewards of acting under uncertainty. The uncertainty arises from the impossibility of the performance models to exactly estimate the execution time of workflow tasks, as well as the impact of the failures of the spots instances on the expected performance of the infrastructure. Thereby, the problem addressed in this paper is a sequential decision making problem in an stochastic domain and the objective is to derive an optimal adaptive budget assignment policy. Thus, we propose a solution approach using an MDP model that allows us to obtain this policy automatically.

3.1 Theoretical Foundations

An MDP can typically be represented as a 5-tuple $(S, A, P.(\cdot,\cdot), R.(\cdot,\cdot), \gamma)$ where:

- S represents the environmental state space;
- A represents the total action space;
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ represents the probability that action a in state s at time t will lead to state s' at time $t+1$;

- $R_a(s, s')$ represents the (expected) immediate reward received after transitioning from state s to state s' due to action a;
- $\gamma \in [0, 1]$ (or discount factor) is the difference in importance between future and immediate rewards. When γ is close to 0, rewards in the distant future are viewed as insignificant. When γ is 1 all rewards are equally important (i.e. additive rewards).

A solution to this problem is the sequence of actions that denote the agent behavior and is represented as policy function $\Pi(s) \in A$ that defines which action must be taken on each state. An optimal policy is a policy that yields the highest expected utility.

Figure 2 depicts a simple MDP model. Choosing Action1 on State1 will lead to State2 with a transition probability of 0.8 or will stay on State1 with a probability of 0.2. Otherwise, choosing Action2 on State1 will lead to State2 with a transition probability of 0.6 or will stay on State1 with a probability of 0.4. Thereby, the best chance to get State2 starting on State1 seems to be by taking Action1 because it provides a greater probability of success. Moreover, to obtain an optimal policy it is necessary to compute the utility of each state based on the probabilities and rewards of all possible outcoming transitions and the utility of the reached states.

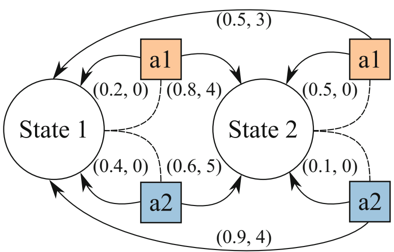


Fig. 2. MDP with two states and two actions (adapted from [14]). The label under each transition shows the corresponding probability and reward values.

3.2 Deriving Adaptive Budget Assignment Policies

We propose to derive an adaptive budget assignment policy by casting the problem to an MDP. We define the *state space* $S = \{\text{EW}, \text{RTR}\}$ as the conjunction of two state variables, *expected workload* (EW) and *running task ratio* (RTR), which describe the environment on a certain execution moment.

EW represents the workload degree that could be present in the next execution period, and it is estimated according to the workload and dependencies present in the sub-workflow formed by the tasks that are likely to be running. In order to estimate a representative value, we select the sub-workflow level[2] with the highest workload, considering the total duration of the tasks based on

[2] The task dependencies structure defines workflow levels.

the amount required machine hours for its completion. Also, we weigh up the workload of each level, according to the minimum estimated start time of the tasks, giving more importance to the first levels, since they are more prone to be executed. Given Υ the set of task belonging to the next period sub-workflow and $\Upsilon^{(i)}$ the set of the tasks in the level i where $\Upsilon^{(i)} \subseteq \Upsilon \subseteq T$, EW is defined as $\mathrm{EW} = {}^{\max\{\mathrm{workload}(\Upsilon^{(i)})\Phi^{(i)}\}}/_{\mathrm{workload}(T)}$.

The factor $\Phi^{(i)}$ weigh up the workload value of each level and it is calculated as $\mathrm{EW} = {}^{\Delta - \min_{k \epsilon \Upsilon^{(i)}}\{\mathrm{startTime}_k - \mathrm{currentTime}\}}/_{\Delta}$, where Δ is the amount of time that determines the frequency of the autoscaling steps, $\mathrm{startTime}_k$ is the estimated start time of the task k in level i and currentTime is the current execution time. Higher workloads indicate that it might be necessary to increase the infrastructure size and thus reduce the waiting time of tasks and thus the workflow makespan. Because budget is limited, EW is closely related with the amount of spot instances, as they maximize parallelism at lower costs. Conversely, a low EW value could indicate a sequential stage of critical tasks, suitable for a reduced and reliable infrastructure. Critical tasks are those that if delayed, will produce an increment of the application makespan (see [3]).

The RTR variable represents the relation between the number of tasks running and all the tasks that will be executed in the next period. This variable give us a picture of the current infrastructure state in terms of the number of running tasks and it represents the availability of computational power for all the new tasks that will be executed in the next period. Then, $\mathrm{RTR} = {}^{|R|}/_{|\Upsilon|}$ where $R \subseteq \Upsilon$ is the set of running tasks. Assuming that the values taken by the variables might not follow a uniform distribution, we discretized the variables by considering intervals defined by percentiles instead of a regular domain subdivision, which would be the natural choice for uniformly distributed data. Finally, we select 10 percentiles samples over each variable resulting in at most 100 possible states.

Each *action* $a \in A$ has the form "Assign a ρ percent of the budget B to acquire spot instances" and exactly matches with the value of the spots ratio parameter (e.g., 0.1, 0.2, ..., 1.0). We define 11 possible actions in the interval $[0, 1]$.

The *probability function* $P_a(s, s')$ returns the probability of a transition $t_{a,s,s'}$ from state s to a state s' by taking the action a based on the number of observed transitions occurrences on previous workflows executions. Let $T_{a,s,s'}$ be the set of observed $t_{a,s,s'}$ transitions and $T_{a,s,\bar{s}}$ the set of observed transitions that depart from state s and reach any other state $\bar{s}$ through the action a. Then, $P_a(s, s') = {}^{|T_{a,s,s'}|}/_{|T_{a,s,\bar{s}}|}$.

The *reward function* $R_a(s, s')$ is the expected work progress through the transition $t_{a,s,s'}$ and is based on the local progress of each transition (i.e. the difference of workflow progress between the target and the source state) considering the duration of the tasks executed in the period. Note that as we are rewarding those actions that lead to greater time reductions, the derived policies will help the autoscaler in finding scaling and scheduling plans that tend to reduce makespan, i.e. the optimization objective of the problem defined in

Eq. (1). Note also that execution costs are not considered in the rewards as they are constrained by the available budget. The autoscaler watches over cost to avoid budget-constraint violations.

Since the same transition may occur several times, as a measure of generality, the mean value is finally taken as the transition reward. For the MDP model we use the feedback produced by workflow executions using different fixed and random policies. In our context such feedback is generated by the autoscaler on each scaling step by reporting the actual state (EW and RTR values), the reward for such state (the actual local progress) and the taken action (the spots ratio used).

Lastly, we use Value Iteration, a dynamic programming algorithm often used to find an optimal policy on an MDP. Note that the policy is derived off-line which avoids producing an overhead in the autoscaler. Algorithm 1 outlines the steps involved in Value Iteration. Firstly, the algorithm starts with an arbitrary value function V_0 over all states, then iteratively updates the value of each state according to:

$$V_k(s) = \max_a \sum_{s'} P_a(s, s')[R_a(s, s') + \gamma V_{k-1}(s')], \quad (2)$$

to get the next value function $V_k (k = 1, 2, \ldots)$. It produces the following sequence of value functions $V_0 \to V_1 \to V_2 \to \ldots \to V^*$. Value Iteration is guaranteed to converge in the limit towards V^*, based on *Bellman optimality equation,* which states that the value of a state under an optimal policy must be equal to the expected return for the best action in the state. When the difference between successive value approximations is lower than a threshold Θ, we assume convergence and the algorithm approximates an optimal policy by:

$$\Pi(s) \leftarrow \arg\max_a \sum_{s'} P_a(s, s')[R_a(s, s') + \gamma V_k(s')]. \quad (3)$$

Algorithm 1. Value Iteration

1: **procedure** ValueIteration($S, A, P, R, \gamma, \Theta$):
2: assign $V_0[S]$ arbitrarily; $k \leftarrow 0$
3: **repeat**:
4: $k \leftarrow k + 1$
5: **for each** state s **do:** $V_k(s) \leftarrow$ update state values according to Eq. (2)
6: **until** $\forall s |V_k[s] - V_{k-1}[s]| < \Theta$
7: **for each** state s **do:** $\Pi(s) \leftarrow$ update policy according to Eq. (3)
8: **return** Π, V_k

4 SIAA Strategy Overview

In previous work [3,8], we proposed a novel autoscaling strategy called Spot Instances Aware Autoscaling (SIAA), which aims for makespan minimization

subject to budget constraints (established per hour of execution). The distinctive feature of SIAA is to take advantage of spot instances, without losing sight of the risks they involve. SIAA is executed periodically generating scaling and scheduling plans for the dynamic adaptation of the infrastructure to the computational demands of the workflow.

The *scaling plan* generated by SIAA, indicates the amount of instances to acquire for each combination of instance type and pricing model. First, the scaling algorithm determines, the total amount of instances of each type needed for executing the tasks that will be running during the next hour of computation. Based on that, it calculates the amount of on-demand and spot instances that can be purchased under the split budget constraints, B^{od} and B^{s} respectively. This process has linear time complexity to the number of running tasks. Then, all the VM instances on the scaling plan are acquired. At this point, the infrastructure has been fitted and the workflow must be executed efficiently considering the available instances.

The *scheduling plan* of SIAA pursues the objective of makespan minimization considering two premises: executing the tasks as fast as possible and mitigating the negative effects of instance failures in the overall makespan. The scheduling algorithm sorts the tasks according to their margin for delays, thus critical tasks are prioritized for execution. Also, critical tasks are allocated to on-demand instances whenever possible. This policy favors that failures, if occur, affect non-critical tasks, which can be rescheduled without a large impact on the overall makespan. The scheduling algorithm also has linear time complexity to the number of tasks.

In our previous studies [3,8] the value of ρ was set to 0.5 to maintain a balance between on-demand and spot instances. Such value was kept fixed during the entire workflow execution. The main issue with this fixed parameter is that restricts the parallelism potential of spot instances in cases of many independent tasks. This approach represents a limitation of SIAA. With the objective of overcoming such limitation, we propose to derive a budget assignment policy able to dynamically adapt the value of the parameter ρ during the workflow execution considering the workload variations. In the next section we use SIAA as the target autoscaler to evaluate the performance of different fixed, random and adaptive policies in terms of makespan reductions and cost savings.

5 Experimental Settings and Results

We evaluated the performance of the MDP-based policies against different fixed and random budget assignment policies. These latter provide in turn the feedback necessary to obtain the MDP-based policies. In all cases we used SIAA as autoscaler, because its novelty and promising performance [3]. We aim to evaluate the impact of using an adaptive budget assignment policy versus a fixed policy, by measuring their performance in terms of makespan and execution cost. Following the SIAA evaluation [3], we used the actual characteristics of five different types of VM instances of Amazon Elastic Compute Cloud (EC2), belonging to the US-west (Oregon) region. These instances types provide a diverse

spectrum of performance and price configurations. Spot instances have the same characteristics that on-demand instances, except that their prices vary over time. We use a history of Amazon EC2 spot prices between March 7th and June 7th of 2016 for the US-west region.

5.1 Experimental Settings

Case study. For the evaluation of the proposal under real load patterns, we used a well-known scientific workflow from the area of Geosciences called Cyber-Shake [6]. The workflow is composed of 1000 tasks belonging to 5 different types and very different durations for each type. We chose this workflow because its structure alternates stages of high and low workload suitable for different budget assignments allowing us to assess the impact of the adaptive budget assignment policies in makespan and task failures. Figure 3 presents the median durations and amount of tasks for each type.

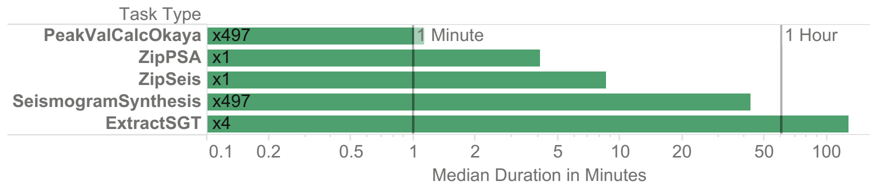


Fig. 3. CyberShake: task median durations for a CyberShake workflow with 1000 tasks. Labels on each bar represent the number of tasks for each type. Duration axis is presented in logarithmic scale.

Budget assignment policies. For the experiments we evaluate 3 families of budget assignment policies. The two first families are the baseline policies used to validate our proposal and the last family comprise different configurations of our MDP policies:

- Eleven Fixed-value policies for which the spot ratios take the values in the set of possible spot ratios $R = \{0.0, 0.1, \ldots, 1.0\}$, these policies represent the strategies that might be set a priori by a user or pre-computed for autoscaling a workflow as in [3]. Note that this kind of policy lacks dynamism, therefore we can expect a wrong fit of the infrastructure for variable workload scenarios.
- One Random-value policy. This strategy is included as an example of a simple dynamic baseline policy. Although intuitively not very effective, it is totally exploratory in nature and therefore suitable as a source of feedback for obtaining more sophisticated policies like in our proposal. Random values are constrained to those in the set of possible spots ratios R.
- Three groups of MDP-based policies resulting from considering the feedback combinations of the two baseline strategies described above. The MDP-F policies are derived from the feedback provided by fixed-value policies only, MDP-R are derived from the random policy only, and MDP-F+R are derived from

all the baseline strategies. These three configurations are chosen to evaluate the influence of each type of feedback on the performance of the autoscaler. We configured the Value Iteration algorithm with $\Theta = 0.001$ (threshold parameter). Also we define a set of values $\Gamma = \{0.1, 0.5, 0.9\}$ for the discount factor γ to explore the impact of long-term rewards on each group, resulting in a total of nine MDP policies.

Experimental scenarios. The 21 policies described were evaluated using SIAA with a base budget of 3.55 USD, which represents the 80% of the hourly budget required to execute a CyberShake workflow with the VM types considered in this study, and bid prices for spot instances that promise a probability of failure $P < 0.1$ to produce scenarios with a moderate amount of task failures. Each experimental scenario was simulated 30 times (i.e. 630 simulations in total) to ensure the statistical robustness of results using the CloudSim simulator [15] version 3.0. In all cases, task durations were affected by a 20% error to offer a more realistic environment according to the performance variability of the Cloud.

5.2 Results and Discussion

Table 1 summarizes the results obtained per policy including the mean value for the metrics of the number of acquired instances, percent of spot instances, the number of out-of-bid errors (OOB) and tasks failures, execution cost and makespan. Considering the mean values of makespan and execution cost, there are six Pareto-optimal policies: three of the Fixed-value policies (Fixed-0.8, Fixed-0.9 and Fixed-1.0) –let us call them Fixed-ManySpots– and the MDP-based policies with $\gamma = 0.5$, whose results are highlighted in the table.

Regarding the Fixed-value policies it can be seen that as the spots ratio grows always allow the acquisition of a larger amount of instances, tripling the infrastructure size on the extreme cases (Fixed-0.0 all on-demand and Fixed-1.0 all spots). Also, in presence of a higher percent of spots, the number of out-of-bid errors increases and therefore more tasks failures occur. Besides, along with the increase of spots ratio we can appreciate higher makespan reductions and cost savings. Note that makespan behaves slightly different (i.e. close but with fluctuating values) in the policies with the spots ratio greater than 0.5. This is because critical tasks are more exposed to the risk that entails the increment of the out-of-bid errors.

The Random policy outperforms all fixed policies with spots ratio below 0.4. We must remember that Random compared with the fixed policies discussed before, has the advantage of being dynamic, but it lacks an optimality sense. Then, the adaptive policies using the MDP model present different results according to the γ value: the policies with $\gamma = 0.1$ achieve the lowest cost, the policies with $\gamma = 0.9$ perform better in terms of makespan but with higher cost and the policies with $\gamma = 0.5$ are Pareto-optimal with the best makespan results (even when costs tend to be larger the scaling plans do not violate the budget constraints). From now on, due to these results, the MDP-based policies will refer to the case with $\gamma = 0.5$. Comparing with the fixed an random policies,

Table 1. Summary of results. Columns present the mean values per policy for each studied metric.

Policy	Instances	% Spots	OOB	Task failures	Makespan [s]	Cost [USD]
Fixed-0.0	9.0	0	0.0	0.0	29321.9	23.1
Fixed-0.1	11.9	32	0.9	7.2	26531.9	19.8
Fixed-0.2	14.8	51	1.9	14.4	24250.1	17.6
Fixed-0.3	18.0	60	3.2	18.9	21721.6	15.9
Fixed-0.4	19.9	69	3.7	21.3	21068.5	14.1
Fixed-0.5	21.8	77	4.5	19.9	20148.9	12.5
Fixed-0.6	24.3	85	4.2	19.9	22190.0	11.1
Fixed-0.7	26.2	91	4.5	23.0	21631.1	10.4
Fixed-0.8	29.3	92	4.6	23.7	**21294.6**	**10.5**
Fixed-0.9	30.8	96	4.9	26.3	**21520.8**	**9.7**
Fixed-1.0	32.0	100	5.8	27.8	**21620.3**	**9.1**
Random	28.0	74	4.2	23.8	20712.9	15.3
MDP-F ($\gamma = 0.1$)	31.9	90	4.9	21.2	21694.5	11.5
MDP-R ($\gamma = 0.1$)	33.6	86	4.6	19.5	21519.0	13.4
MDP-F+R ($\gamma = 0.1$)	35.0	86	5.3	24.9	21368.0	12.4
MDP-F ($\gamma = 0.5$)	31.0	90	4.9	26.5	**19116.7**	**12.0**
MDP-R ($\gamma = 0.5$)	33.1	88	5.6	30.0	**18822.8**	**12.5**
MDP-F+R ($\gamma = 0.5$)	35.0	84	5.2	29.6	**18598.6**	**13.7**
MDP-F ($\gamma = 0.9$)	29.8	75	2.6	10.6	19259.5	16.1
MDP-R ($\gamma = 0.9$)	26.3	80	4.2	21.7	19494.4	13.1
MDP-F+R ($\gamma = 0.9$)	31.2	74	3.1	20.8	19419.7	16.7

the MDP-based policies acquire the largest infrastructure with a high percent of spots instances. Also, even with many out-of-bid errors and task failures, these adaptive policies achieve better results than the most of fixed policies. Moreover, the fact that the MDP-F+R policy achieves the best makespan result, shows the potential of partially taking random actions when exploring the solutions space.

Figure 4 presents the mean values and standard deviations for the makespan and execution cost of the Pareto-optimal policies (Fixed-ManySpots and MDP-based) and the Random policy. The MDP-based policies present the shortest makespans and this is the metric that we are optimizing through transition rewards of the MDP model. Fixed-ManySpots present largest makespans and largest standard deviations because of the larger impact of out-of-bid errors on infrastructures comprising a high percent of spot instances. The Fixed-ManySpots policies also show the best result in terms of cost savings because of the low prices of the spots instances. In summary, each policy groups optimize a different objective: MDP-based policies achieve the minimum makespan and the Fixed-ManySpots policies acquire the cheapest but less reliable infrastructure.

Figure 5 shows the values distributions of makespan and execution cost from the Pareto-optimal and random policies discussed before. The Fixed-ManySpots

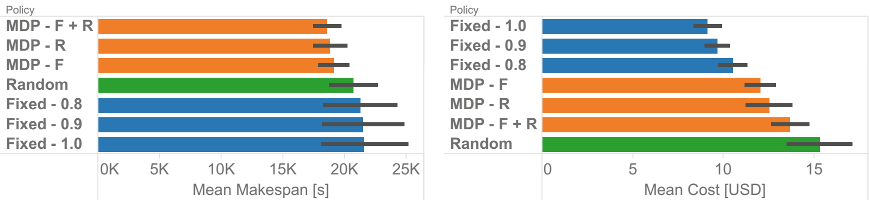


Fig. 4. Mean makespan (left) and cost comparison (right) for each policy.

policies show a higher dispersion on makespan values indicating the unreliability inherent to an infrastructure with a high percent of spot instances, doubling the maximum makespan obtained by an adaptive policy. In terms of cost the Fixed-ManySpots policies achieves the best values with the smaller median, which is due to the low prices of the instances acquired. Besides, the MDP-based policies, present lower makespan values than all of the fixed policies and with much less dispersion. The Random-value policy shows the worst cost results even when it outperforms all of the fixed policies w.r.t. makespan.

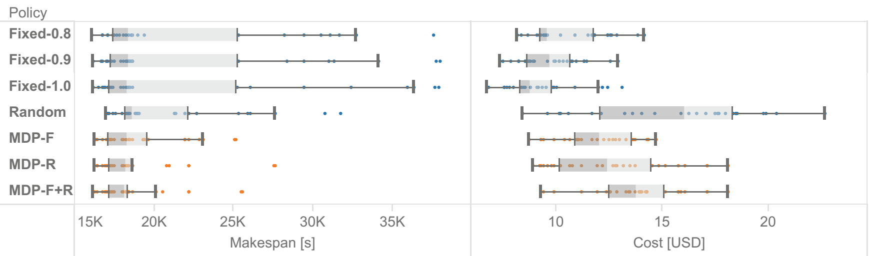


Fig. 5. Box-plot comparing the workflow makespan in seconds (left) and the execution cost in USD (right) values distributions for each policy.

To ensure the statistical significance of our results we grouped the results of the best-performing MDP and Fixed policies (i.e. MDP-based vs. Fixed-ManySpots). By applying the Kolmogorov-Smirnov test we found that the makespan and cost results do not follow a normal distribution. Therefore, we applied the Mann–Whitney U test, a non-parametric test for median value comparison. We applied the test with a significance level of $\alpha = 0.1$. In terms of makespan we got the statistic $U = 4631$ and the p-value$=0.096$. In terms of cost, we got the statistic $U = 6910$ and the p-value ≤ 0.001. In both cases, the p-value is less than the significance level, which indicates that our results are statistically significant with a 90% of confidence.

In summary, the MDP-based policies on states of high workload, usually choose an action that implies using a higher spots ratio value. This increases the infrastructure allowing the parallel execution of many tasks. Then, on states of

sequential tasks, the policies reduce the spots ratio value looking for a reliable infrastructure, ensuring that the critical tasks will not be delayed by the possible spot instances failures. As explained in Sect. 3.2, rewarding actions that will potentially reduce execution times produce MDP policies that lead to makespan reductions when compared with the baseline policies. This observation is consistent with the fact that the autoscaler aims for makespan minimization subject to budget constraints. Also note that although costs tend to be larger for MDP than for the baseline policies, SIAA produces scaling plans that do not violate such budget constraints. These promising results encourage the study of new strategies aiming to find optimal budget policies applied to the execution of workflows on the Cloud.

6 Related Work

Decision making under uncertainty is a common scenario in many optimization problems, thus the MDP has become an interesting approach in the study of a wide range of problems solved through dynamic programming or reinforcement learning. Although there have been extensive studies on workflow scheduling and resource provisioning in Grid and Cloud, to the best of our knowledge, there are few proposals using MDP to address similar problems.

Barrett et al. [13] proposed a Cloud workflow scheduling approach which employs an MDP to optimally guide the workflow execution process depending on environmental state. In addition, the system employs a genetic algorithm to evolve workflow schedules. In a similar way, Yu et al. [16] proposed a cost-based workflow scheduling approach based on using MDPs to schedule tasks on utility Grids. The main objective is to minimize execution cost while meeting the overall deadline. Their approach partitions the workflow into branches containing sequential tasks with sub-deadlines. Then the branches are mapped into resources through the solution of an MDP model via the Value Iteration algorithm. These works address the problem of scheduling workflows through an MDP, but our proposal is different because it focuses on the problem of budget assignment. We use an MDP model to dynamically determine the optimal spots ratio on each scaling step, while the scaling and scheduling problems are delegated to an autoscaler (SIAA in this paper).

Later on, Barrett et al. [14] proposed a method based on MDP and Q-learning for the autoscaling of independent user requests arriving according to a Poisson distribution. They model the entire autoscaling problem as deriving policies that permit request, maintain or kill on-demand VMs. The main differences with our work are (i) we focus on a policy for the proper assignment of the budget for on-demand and spot instances, (ii) we focus on scientific workflows instead of independent user requests.

Tang et al. [10] presented a set of bidding strategies to minimize the cost and volatility of resource provisioning with the spot instances scheme of Amazon EC2. Essentially, to derive an optimal bidding strategy, the authors formulate the problem as a Constrained Markov Decision Process (CMDP). Although

the authors deal with the problem of spot instances acquisition they focus on the obtainment of an optimal bidding strategy while our proposal is centered on the obtainment of an adaptive budget assignment policy for autoscalers, which is independent of the bidding strategy used. None of the surveyed works addressed the problem dealt in this paper, which indicates the novelty of our approach.

7 Concluding Remarks

This paper formalizes the problem of finding an adaptive budget assignment policy as an MDP problem. The approach instantiating the model permits deriving new policies from the feedback (information about the state of the infrastructure and the workflow execution) of other policies executed as part of an autoscaling strategy. Obtained policies were compared against baseline policies, fixed and random. The evaluation was carried out by comparing makespan and execution cost of a state-of-the-art autoscaler called SIAA using all the studied policies. Experiments were carried out on the well-known benchmark workflow (i.e. CyberShake), in the near future, we will extend our analyses by considering a wider spectrum of workflows applications, e.g. LIGO's Inspiral, Montage and SIPHT.

We corroborated that MDP-based policies outperform all the competitor policies in terms of makespan and most of them in terms of cost. Regarding makespan, the adaptive policies present important reductions ranging between 2503.6 s and 2696 s (11.6% to 12.7%) without violating the initial budget. A downside is that the approach incurs in overall cost increases of around 3 USD in average in comparison with the other Pareto-optimal policies, i.e. fixed policies with intensive use of spot instances. These are very encouraging results considering that the MDP policies were obtained only rewarding larger time reductions and disregarding cost savings. Notice that in our context, the makespan is of utmost importance because underlying there is a problem of makespan minimization under budget constraints. Based on these results and following the intuition that the MDP-based policies are more adaptable and capable of performing better even facing unknown situations, we will continue improving our proposal in different ways cited below.

First, we will incorporate a notion of cost saving along with time reduction into rewards to obtain more balanced policies. Second, we are interested in studying the applicability of reinforcement learning techniques to update the policies over time taking advantage of exploration/exploitation approach of these techniques. Finally, we are interested on redefining our MDP model in order to capture generic characteristics of the workflow and status of the environment. The ultimate goal is to be able to obtain policies capable of performing well in workflows that were not seen previously. This could be a distinctive and valuable feature for these policies that would really make the MDP approach worth the effort. All of these issues could have a great impact finding an optimal adaptive budget assignment policy and in a general sense would be an important step for achieving a real dynamism and adaptability in autoscaling strategies and Cloud infrastructure provisioning.

Acknowledgements. This research is supported by the ANPCyT projects No. PICT-2012-2731 and PICT-2014-1430; and by the UNCuyo project No. SeCTyP-M041. The authors want to thank the anonymous reviewers for their valuable comments and suggestions that helped to improve the quality of this paper.

References

1. Buyya, R., Yeo, C., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. **25**(6), 599–616 (2009)
2. Mao, M., Humphrey, M.: Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 67–78. IEEE (2013)
3. Monge, D.A., Yisel, G., Mateos, C., García Garino, C.: Autoscaling scientific workflows on the cloud by combining on-demand and spot instances. Int. J. Comput. Syst. Sci. Eng. **32**(4 Special Issue on Elastic Data Management in Cloud Systems), 291–306 (2017)
4. Expósito, R.R., Taboada, G.L., Ramos, S., Touriño, J., Doallo, R.: Performance analysis of HPC applications in the cloud. Future Gener. Comput. Syst. **29**(1), 218–229 (2013)
5. Ben-Yehuda, O.A., Ben-Yehuda, M., Schuster, A., Tsafrir, D.: Deconstructing Amazon EC2 spot instance pricing. ACM Trans. Econ. Comput. **1**(3), 16:1–16:20 (2013)
6. Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., Vahi, K.: Characterizing and profiling scientific workflows. Future Gener. Comput. Syst. **29**(3), 682–692 (2013)
7. Huu, T.T., Koslovski, G., Anhalt, F., Montagnat, J., Vicat-Blanc Primet, P.: Joint elastic cloud and virtual network framework for application performance-cost optimization. J. Grid Comput. **9**(1), 27–47 (2011)
8. Monge, D.A., García Garino, C.: Adaptive spot-instances aware autoscaling for scientific workflows on the cloud. In: Hernández, G., Barrios Hernández, C.J., Díaz, G., García Garino, C., Nesmachnow, S., Pérez-Acle, T., Storti, M., Vázquez, M. (eds.) CARLA 2014. CCIS, vol. 485, pp. 13–27. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45483-1_2
9. Turchenko, V., Shultz, V., Turchenko, I., Wallace, R.M., Sheikhalishahi, M., Vazquez-Poletti, J.L., Grandinetti, L.: Spot price prediction for cloud computing using neural networks. Int. J. Comput. **12**(4), 348–359 (2013)
10. Tang, S., Yuan, J., Li, X.Y.: Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In: Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012, pp. 91–98 (2012)
11. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
12. Van Otterlo, M.: The Logic of Adaptive Behavior. Frontiers in Artificial Intelligence and Applications, vol. 192. IOS Press, Amsterdam (2009)
13. Barrett, E., Howley, E., Duggan, J.; A learning architecture for scheduling workflow applications in the cloud. In: Proceedings of the 9th IEEE European Conference on Web Services, ECOWS 2011, pp. 83–90 (2011)
14. Barrett, E., Howley, E., Duggan, J.: Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurr. Comput. Pract. Exp. **25**, 1656–1674 (2012)

15. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw.: Pract. Exp. **41**(1), 23–50 (2011)
16. Jia, Y., Buyya, R., Tham, C.K.: Cost-based scheduling of scientific workflow applications on utility grids. In: Proceedings of the First International Conference on e-Science and Grid Computing, e-Science 2005, pp. 140–147 (2005)

Experimental Analysis of Secret Sharing Schemes for Cloud Storage Based on RNS

Vanessa Miranda-López[1], Andrei Tchernykh[1(✉)], Jorge M. Cortés-Mendoza[1], Mikhail Babenko[2], Gleb Radchenko[3], Sergio Nesmachnow[4], and Zhihui Du[5]

[1] CICESE Research Center, Ensenada, BC, Mexico
{vmiranda, chernykh, jcortes}@cicese.edu.mx
[2] North-Caucasus Federal University, Stavropol, Russia
mgbabenko@ncfu.ru
[3] South Ural State University, Chelyabinsk, Russia
gleb.radchenko@susu.ru
[4] Universidad de la República, Montevideo, Uruguay
sergion@fing.edu.uy
[5] Tsinghua University, Beijing, China
duzh@tsinghua.edu.cn

Abstract. In this paper, we address the application of Redundant Residue Number System (RRNS) to improve the security of public data storage, reduce storage space, and process encrypted data. We provide a comprehensive experimental analysis of Asmuth-Bloom [14] and Mignotte [15] schemes that use RRNS and Secret Sharing Scheme (SSS) to design reliable and secure storage systems. These schemes are studied in real multi-cloud environment to find compromise between performance, redundancy, and data security. We analyze and compare the speeds of encoding/decoding and upload/download of these algorithms for different RRNS settings with 11 well-known cloud storage providers. We also provide a mathematical analysis of the expected system behavior.

Keywords: Cloud computing · Storage · Security · Reliability
Residue Number System · Homomorphic encryption

1 Introduction

Cloud computing provides many benefits in terms of low costs, accessibility of data and efficient computing. In the Cloud, the resources are provided to the users based on demands from the pool of shared resources. Distributed storage systems play a key role in the new era of cloud computing and big data.

In the past few years, the concept of multi-clouds, inter-cloud or cloud-of-clouds has been used as a solution to overcome drawbacks of dealing with a single cloud. Multi-cloud can be defined as the use of multiple cloud services as a single heterogeneous architecture [1].

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 370–383, 2018.
https://doi.org/10.1007/978-3-319-73353-1_26

As cloud storages become common, more and more users send sensitive information into the cloud due to the increasing development of cloud storage technology. As a result, transmitted data are increased up to 1.1 exabytes (EB) in 2016. It is expected that in 2020 data volume will be increased up to 2.3 EB [2].

Despite advantages of the cloud storage, it brings high risks of confidentiality, integrity, and availability associated with the loss of information, denial of access for a long time, information leakage, and conspiracy that need to be cope. One of the main problems in the design of a reliable cloud storage system is to deal with the uncertainty of the occurrence of technical failures, data security breaches, collusion, etc. [3–5].

It is important to develop techniques to store and process such a data ensuring their reliability and security. One possible direction is to rely on multiple clouds as a distributed storage, where data are divided into pieces to be stored in different clouds. With this approach, availability of data can be ensured, but failures may cause inconsistency among copies of the same data [6, 7].

A solution that can improve data privacy and confidentiality is to encrypt the data before outsourcing. However, sometimes, users need to perform calculations over their data, and a drawback of encryption is that it does not allow data processing.

In [8], authors proposed to use fully homomorphic encryption. Nevertheless, this framework leads to significant data redundancy, computational complexity of data encryption algorithms, and low reliability. In [9], a cloud storage system based on RRNS is proposed to overcome these disadvantages. An approach that considers the cost implication of using real CSPs was introduced in [10].

Methods like Data Replication, Secret Sharing Schemes (SSS), Redundant Residue Number System (RRNS), Erasure Codes (EC), Regenerating Codes (RC), Homomorphic Encryption (HE), etc. have been used in order to minimize the threats of corruption or/and data loss, ensuring security and confidentiality [3, 11–13].

Modular SSS was independently introduced by Asmuth-Bloom [14] and Mignotte [15]. The basis of the both approaches is the Chinese Remainder Theorem (CRT).

In this paper, we present an analysis of the SSS proposed by Asmuth-Bloom and Mignotte in a multi-cloud environment based on RRNS for different system parameters. We also consider information upload and download into/from real cloud storage providers (CSPs).

The paper is structured as follows. The next Sect. 2 briefly reviews related works. Section 3 presents the basis of storage scheme in a multi-cloud, describes our experimental setup, and presents results their analysis. Section 4 highlights the conclusions of the paper and future work.

2 Related Work

Motivated by the safety issues and risks of technical failures of a distributed storage, several scientific contributions have been proposed in order to build a reliable, secure data storage. In one hand, SSS splits data into n segments, and each is uploaded onto a different clouds. The advantage of using SSS is that only k or more of the n participants

can determine the data without sharing information. SSS proposed by [14, 15] ensure the security and confidentiality of data. However, this approach can lead to high redundancy.

On the other hand, classical coding theory focuses on the tradeoff between redundancy and error tolerance. EC [23, 24] and RC [25] aim to reduce the amount of downloads during repair, while retaining the storage efficiency. Nevertheless, EC and RC lack of the capability of performing calculations on encrypted data. An alternative approach studied in [16], named HE, allows to carry out computations on the ciphertext generating an encrypted result. When decrypted, it matches the result of operations performed on the original numbers. Later, [17] proposed the first fully HE scheme capable of performing addition, subtraction and multiplication operations, but, with a high cost of data redundancy.

Since encryption techniques and security protocols are not sufficient to protect data in the cloud, researchers have turned their attention to alternative number system representations as an effort to further boost up cryptosystem performance.

Residue Number System (RNS) is a well-known number theory paradigm [18] and has emerged as a key-player in this endeavor. RNS allows controlling results of data processing and reduces data redundancy. In [19] authors proposed a method to reduce the complexity of modular multiplication, they substitute expensive modular operations, by fast bit right shift operations and taking low bits. In order to apply RNS for encryption, the issues of confidentiality, integrity, and cloud collusion need to be addressed [22].

A particular distributed data storage based on RNS was proposed in [20], which assures the safety, confidentiality, homomorphism, reliability, and scalability of data. However, they use the moduli set as the secret key leading to high redundancy and resource intensive decoding.

Taking advantage of RRNS properties, [9] proposed a reliable and scalable storage system in a multi-cloud environment. This system allows to minimize the data redundancy, without loss of reliability, and perform arithmetic operations.

In cloud computing, there exist many types of uncertainties that there should be considered in order to evaluate the performance, reliability, and security of a storage system. Uncertainty occurs in data processing, storage, and transmission. In a scenario, with uncertainties, the authors in [3–5] studied the risks associated with the appearance of cloud collusion under uncertain conditions. They use a modified threshold Asmuth-Bloom and weighted Mignotte SSS to mitigate this type of uncertainty, while minimizing the damages caused by cloud collusion.

To design a configurable, reliable and secure data storage based on cloud-of-clouds technology, data storage has the following properties: homomorphic cipher, weighted SSS, and error correction codes. To determine the configuration parameters, the following optimization criteria should be taken into account: accuracy, scalability, and reliability, confidentiality, security, and performance.

3 Experimental Analysis for Reliable and Secure Storage Schemes

In this section, we present the basis for the storage scheme based on a multi-cloud approach. For the experimental analysis, in order to obtain realistic results, we use real upload and download speed access to/from eleven CSPs.

3.1 Storage Model

We consider the scenario where we have a set of n clouds, and users' data. We use the properties of RRNS, which can be seen as a SSS and provided an error detection and correction capability. Data are divided into a set of smaller encrypted chunks, giving each cloud provider its own unique chunk. To reconstruct the data, some of the chunks or all of them are needed.

RRNS is an extension of RNS, which represents an integer as a set of its residues according to a moduli set. We can form RRNS by adding redundant moduli into an existing moduli set to extend the legitimate range of the original information moduli. The extended range is called the illegitimate range. Therefore, RRNS represents integers by means of a $(k + r)$-tuple of their residue moduli that consists of a set of pairwise coprime numbers $\{p_1, p_2, \ldots, p_n\}$, where $n = k + r$, k are the original moduli set, and r are the redundant moduli.

The range of RRNS is $P = \prod_{i=1}^{k} p_i$. Any integer X can be represented by its residues in Z_P with a n-tuple $X \overset{RRNS}{\rightarrow} (x_1, x_2, \ldots, x_n)$, where $x_i = X \bmod p_i$ or $\mathrm{x_i} = |\mathrm{X}|_{\mathrm{p_i}}$. For every n-tuple, the corresponding integer $X \in [0, P - 1)$ can be recover by means of the CRT:

$$X = \left(\sum_{i=1}^{k} x_i P_i b_i\right) \bmod P \tag{1}$$

$\forall i = \overline{1, k}$, where $P_i = P/p_i$ and b_i is the multiplicative inverse of P_i modulo p_i.

RRNS can be seen as homomorphic cipher, because, it meets the following property:

$$X * Y = (x_1, x_2, \ldots, x_n) * (y_1, y_2, \ldots, y_n) = \left(|x_1 * y_1|_{p_1},\ |x_2 * y_2|_{p_2}, \ldots,\ |x_n * y_n|_{p_n}\right) \tag{2}$$

where $X \overset{RRNS}{\rightarrow} (x_1, x_2, \ldots, x_n)$, $Y \overset{RRNS}{\rightarrow} (y_1, y_2, \ldots, y_n)$, $*$ denotes one of the operations: addition, multiplication, subtraction, and $|X|_{p_i}$ denotes the remainder of division of X by p_i.

In RRNS settings (k, n), using data from any k remainders from n, we can recover $(n - k)$ data and use the r remainders to verify the correctness of recover data. A unique and powerful capability of RRNS is error detection and correction accomplished as a result of the extra residues [9, 21]. Operations can be achieved in parallel since residues are independent of each other. For error detection, if RRNS has r

redundant moduli, then it is capable of detect r and correct $r-1$ errors. In [13], it is shown that the reliability of a system depends on r. A bigger value of r, more reliable the system is, with the cost of growing redundancy. As result, if we take into account the volume of the data, the redundancy becomes the key factor in terms of distributed storage of big data.

In the storage model, each CSP receives a chunk of data that consists of a chunk identifier, chunk properties, projection of the original data, simplified digital signature, and moduli RNS. To compute the unique secret key, we use hash function based on SHA-3 algorithm [26].

Let i-th cloud corresponds to RNS module p_i of the form $2^b - \alpha_i$, if i is odd, and $2^b + \alpha_i$, if i is even. The values of b and α_i are chosen according to available computational resources of i-th cloud, demanded level of security, and the reliability of the data storage. For binary conversion method, [13] proposed to use an approximation of the rank of RNS number based on replacement computation. The rank is calculated using a modification of Eq. (1). Now, an integer X can be calculated by the Eq. (3):

$$X = \sum_{i=1}^{k} P_i \left| P_i^{-1} \right| x_i - r_x \cdot P, \tag{3}$$

where $r_x = \left\lfloor \sum_{i=1}^{k} \frac{\left|P_i^{-1}\right|_{p_i}}{p_i} x_i \right\rfloor$, and $P = \prod_{i=1}^{k} p_i, P_i = P/p_i$. For all $i = \overline{1,k}$, $\mathrm{x_i} = |\mathrm{X}|_{\mathrm{p_i}}$ is an integer smaller than $\mathrm{x_i} = |\mathrm{X}|_{\mathrm{p_i}}$. r_x determined how many times the RRNS range is increased.

As depicted in Fig. 1, the system uses RRNS properties to send the segments of the data across different CSPs. Depending on users' requirements, the system configures its parameters, then, data are split with a given degree of redundancy.

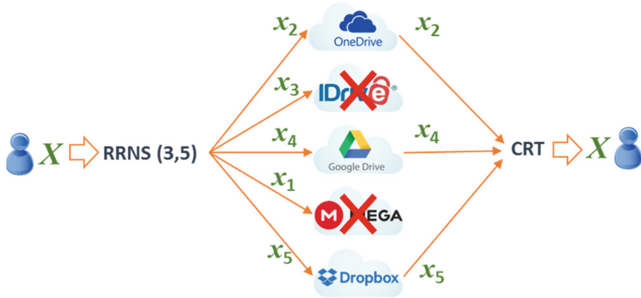


Fig. 1. Encoding and decoding for RRNS settings (3, 5).

The residue-tuples (segments $(x_1, \ldots, x_5)$) will be copied and stored on different CSPs. Since data are not stored on a single cloud, but over several CSPs, it leads to increased confidentiality. On the other hand, due to the redundancy property of RRNS, user´s data might still be reconstructed from the remaining segments, even if $(n-k)$ CSPs are temporary unavailable (clouds 2 and 4 in Fig. 1), or if a provider loss users' data.

3.2 Experimental Setup

We simulate two classical schemes used to design storage systems, SSS Asmuth-Bloom and Mignotte. The system is developed considering the Java programming language.

To determine the speed of data access, we use a video file divided into chunks of 50, 150, 250, 350 and 450 MB [3]. Experiments were performed on the CPU Intel (R) Core (TM) i7-6700 CPU @ 3.40 GHz, RAM 16 GB, HDD 1 TB, and Chrome browser version 58.0.3029.81 with Ethernet connection, which typically averages 143.71 Mbps download and 144.85 Mbps upload on Speedtest.net. Operating system is Windows 8.1 professional x64.

The chunks are uploaded to the providers, then, downloaded during three days, each 4 h. Finally, 30 measurements are obtained for corresponding speeds (see Table 1). We take, as a reference, the average speeds to download/upload from the CSPs presented in Table 1.

Table 1. Speed of access to data, and storage for cloud storage providers.

I	Provider	Storage (GB)		Download (MB/s)			Upload (MB/s)		
		Free	Max	Low speed	High speed	Average speed	Low speed	High speed	Average speed
1	Elephantdrive	2	1,000	2.50	4.55	3.43	8.65	13.63	10.42
2	Box	10	100	1.04	1.67	1.29	7.53	12.50	10.28
3	Dropbox	2	2,000	2.50	8.92	4.93	5.05	13.63	9.70
4	Justcloud	1	75	10.00	14.29	11.80	6.08	13.63	9.65
5	Cloudmail	25	512	5.49	10.71	7.89	5.35	16.66	9.58
6	Googledrive	15	100	4.55	12.50	7.87	8.33	10.71	9.32
7	Yandexdisk	20	100	3.80	9.78	6.21	7.14	10.00	9.02
8	Idrive	5	1,000	0.02	0.07	0.03	5.31	12.50	8.62
9	Sync	5	2,000	1.61	4.41	2.78	4.36	6.57	5.33
10	Onedrive	5	1,000	1.67	4.78	2.44	4.45	5.83	5.23
11	Icloud	5	200	1.65	5.10	3.32	0.32	0.46	0.40

In order to understand the behavior of the system in a real scenario, we considered files with different sizes: 50 MB, 100 MB, and 200 MB. We analyze two scenarios for CSPs selection. First, the CSPs are ordered by their average upload speed in a descending order (see Table 2). In the second scenario, we order the CSPs by their average download speed in a descending order (see Table 3). For both scenarios, we send/receive chunks into the different CSPs depending on the sorting criteria used.

In [27], it is shown that the level of protection of secret information processed using RNS depends on the threshold value used in the SSS and the size of the modules. For the experiments, we generate our moduli set with a sequence of primes $p_1, p_2, \ldots, p_n$ such that Eq. 4 holds.

$$\prod_{i=1}^{k} p_i > p_0 \cdot \prod_{i=1}^{k-1} p_{n-k+1+i}. \tag{4}$$

We choose $p_1, p_2, \ldots, p_k$ as large as possible but $p_{n-k+2}, p_{n-k+3}, \ldots, p_n$ as small as possible, considering that $\prod_{i=2}^{k} p_i < \prod_{i=1}^{k-1} p_{n-k+1+i}$ should also be held. The algorithm generates n consecutive prime numbers starting from the smallest prime larger than p_0. Next, the algorithm checks if Eq. 4 holds. If this condition is not satisfied, it generates the next prime, and again checks if the largest n primes satisfy the condition.

Table 2. CSP ranks based on average upload speed

Sorted by upload velocity			
I	Provider	Average speed (MB/s)	
		Upload	Download
1	Elephantdrive	10.42	3.43
2	Box	10.28	1.29
3	Dropbox	9.70	4.93
4	Justcloud	9.65	11.8
5	Cloudmail	9.58	7.89
6	Google	9.32	7.87
7	Yandex	9.02	6.21
8	Idrive	8.62	0.03
9	Sync	5.33	2.78
10	Onedrive	5.23	2.44
11	Icloud	0.40	3.32

Table 3. CSP ranks based on average download speed

Sorted by download velocity			
I	Provider	Average speed (MB/s)	
		Upload	Download
1	Justcloud	9.65	11.8
2	Cloudmail	9.58	7.89
3	Google	9.32	7.87
4	Yandex	9.02	6.21
5	Dropbox	9.7	4.93
6	Elephantdrive	10.42	3.43
7	Icloud	0.40	3.32
8	Sync	5.33	2.78
9	Onedrive	5.23	2.44
10	Box	10.28	1.29
11	Idrive	8.62	0.03

3.3 Results and Analysis

In this section, we provide the analysis of the threshold schemes varying the number of segments and threshold values.

Figure 2 shows the graph of encoding velocity by Asmuth-Bloom and Mignotte schemes. We see that as we generate more chunks (n is increased), additional time to encode data is required resulting in a decrease of the speed. Mignotte scheme has an encoding speed 4 times greater than Asmuth-Bloom with values of 5.649 MB/s and 1.495 MB/s, respectively. The lowest speeds in both schemes are 1.284 MB/s for Mignotte and 0.463 MB/s for Asmuth-Bloom. We observe that Asmuth-Bloom scheme has more computational overhead for data splitting. Besides, it adds a noise factor into the data in order to increase security in the system, thus, its encoding velocity is slower than one of Mignotte.

The decoding performance for both schemes is shown in Fig. 3. Asmuth-Bloom and Mignotte use CRT to recover the original data. When $k = 2$, the maximum speed is 6.047 MB/s for Mignotte and 3.17 MB/s for Asmuth-Bloom. For the minimum speed, Mignotte outperforms 8 times Asmuth-Bloom with a minimum value of 3.42 MB/s.

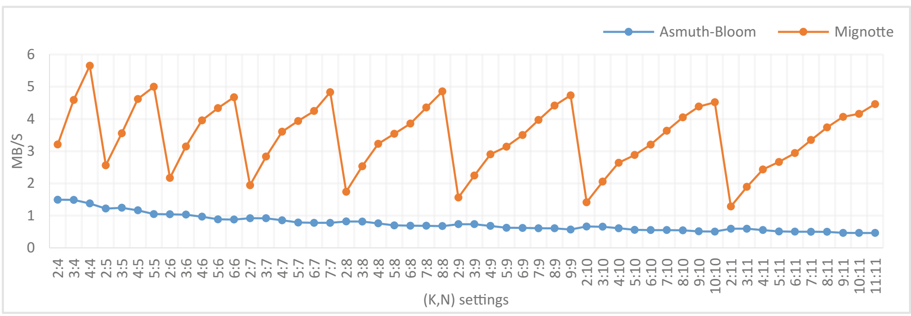


Fig. 2. The speed of encoding data (MB/s) versus RRNS settings (k, n), with b = 8.

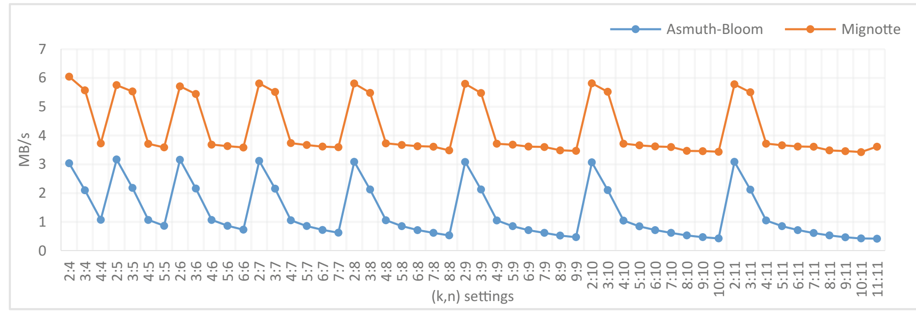


Fig. 3. The speed of decoding data (MB/s) versus RRNS settings (k, n), with b = 8.

We are focusing on a model of multi-cloud environment, so it is important to analyze impact of the expected speed of sending data to the CSPs on the overall performance. As we mentioned before, we use the average speeds of Table 1 to obtain the upload/download of the chunks into the CSPs. We introduce the following notation to describe the mathematical analysis of the access to the clouds:

D - size of original data,
d_i - size of i-th chunk,
v_{dow_i} - download speed of i-th chunk. Without loss of generality, we assume that $v_{dow_1} \geq v_{dow_2} \geq \ldots \geq v_{dow_n}$ (downloading average speed, Table 1),
v_{up_i} - upload speed of i-th chunk (uploading average speed, Table 1),
T_E - data encryption time (Fig. 2),
T_D - data decryption time (Fig. 3),
t_{dow_i} - downloading time of one chunk from i-th cloud,
t_{up_i} - uploading time of one chunk to i-th cloud,
p - probability of deny of access of one cloud provider, $q = 1 - p$.

The data upload speed is calculated with the following formula:

$$V_u = \frac{D}{T_E + \sum_{i=1}^{n} \frac{d_i}{v_{up_i}}}. \quad (5)$$

Using the data from Tables 2 and 3, and results of encryption in the Fig. 2, we obtain Figs. 4 and 5 that show access speed depending on the SSS and its settings.

Figure 4 depicts the maximum values for both schemes considering that encoding/uploading is 2.653 MB/s for Mignotte and 0.681 MB/s for Asmuth-Bloom. If we consider the rankings presented on Table 3, the CSP with the lowest upload speed has a rank of 7, affecting the access speeds from settings with $n \geq 7$. The upload speed is decreased up to 2.05 MB/s, for Mignotte, and up to 0.28 MB/s, for Asmuth-Bloom, as we can see in Fig. 5.

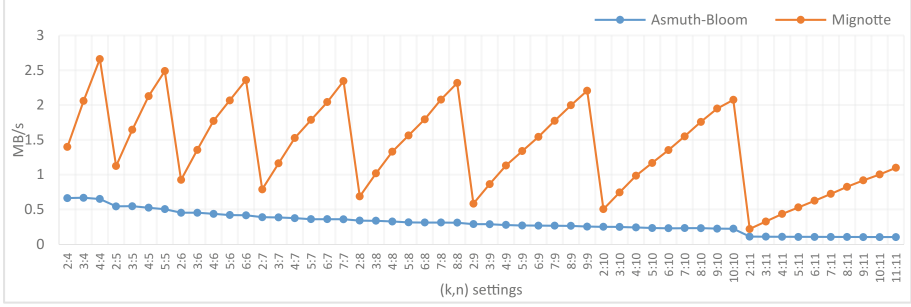


Fig. 4. Velocity of encoding and uploading n chunks based on Table 2.

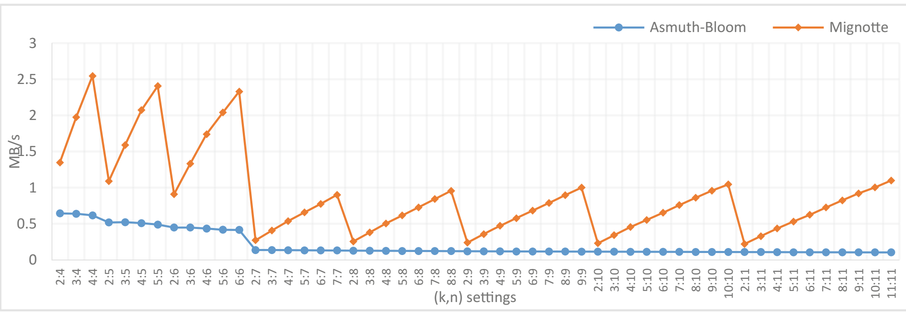


Fig. 5. Velocity of encoding and uploading n chunks based on Table 3.

Using the notation describe above, we calculate the characteristics of the system. The time of downloading the i-th chunk from the i-th cloud is $t_{dow_i} = d_i / v_{dow_i}$.

Under the conditions of limited internet bandwidth, we assume that the chunks are downloaded from the clouds sequentially. If the first cloud is available, then the first chunk is downloaded from it, otherwise, the second cloud availability is checked, etc. The process is terminated when k chunks are downloaded, or when we tried to access all n clouds. If, in either case, we did not download k chunks, then we do not have access to the data at that time.

The time of query whether the cloud is available is assumed to be zero, since, we consider that there is an independent service that makes the queries, so at each point of time, the system knows, which clouds are available.

In the best case scenario, we download k chunks for $t_{down}^{best} = \sum_{i=1}^{k} d_i / v_{dow_i}$, and, in the worst case scenario, we have $t_{down}^{worst} = \sum_{n-k+1}^{n} d_i / v_{dow_i}$.

We consider a set of j clouds. According to the above scheme, the user can access to the first j cloud providers, if k of them are available, or if from the first $j-1$ cloud providers, if $k-1$ are available. We calculate the expectation of download time of k chunks as follows:

$$t_{down}^{expect} = \sum_{j=k}^{n} \frac{\frac{(j-1)!}{(j-k)!(k-1)!} p^{j-k} q^k}{\sum_{r=k}^{n} \frac{(r-1)!}{(r-k)!(k-1)!} p^{j-k} q^k} \sum_{i=1}^{j} \frac{d_i}{v_{dow_i}}$$

$$= \sum_{j=k}^{n} \frac{(j-1)! \cdot p^{j-k}}{(j-k)! \sum_{r=k}^{n} \frac{(r-1)!}{(r-k)!} p^{r-k}} \sum_{i=1}^{j} \frac{d_i}{v_{dow_i}}$$

The access time to the data, in the best case scenario, is calculated by $T_D + \sum_{i=1}^{k} d_i / v_{dow_i}$, and, in the worst case, $T + \sum_{n-k+1}^{n} d_i / v_{dow_i}$.

Finally, the expectation of time access to data is:

$$T_D + \sum_{j=k}^{n} \frac{(j-1)! \cdot p^{j-k}}{(j-k)! \sum_{r=k}^{n} \frac{(r-1)!}{(r-k)!} p^{r-k}} \sum_{i=1}^{j} \frac{d_i}{v_{dow_i}}$$

Consequently, the data access speeds are calculated as follows:
In the best case:

$$Vd_{max} = \frac{D}{T_D + \sum_{i=1}^{k} \frac{d_i}{v_{dow_i}}}. \tag{6}$$

In the worst case:

$$Vd_{min} = \frac{D}{T_D + \sum_{n-k+1}^{n} \frac{d_i}{v_{dow_i}}}. \tag{7}$$

The expectation of downloading speed:

$$Vd_E = \frac{D}{T_D + \sum_{j=k}^{n} \frac{(j-1)! \cdot p^{j-k}}{(j-k)! \sum_{r=k}^{n} \frac{(r-1)!}{(r-k)!} p^{r-k}} \sum_{i=1}^{j} \frac{d_i}{v_{dow_i}}} \tag{8}$$

In [13], it is shown that p ≈ 0.01. Hence, using the data from Tables 2 and 3 and results from the simulation of decryption in the Fig. 3, we obtain the following graphs of access speed depending on the SSS and its settings (Figs. 6 and 7).

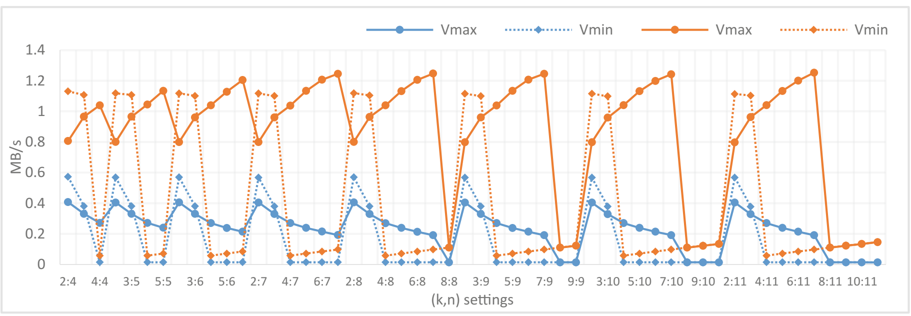


Fig. 6. Velocity of downloading k chunks and decoding based on Table 2.

Figure 6 depicts the behavior of downloading the chunks, when the CSPs are sorted by their upload speeds. From settings with $k \geq 8$, the access speed is decreased to 1.13 MB/s for Mignotte, and to 0.18 MB/s for Asmuth-Bloom, in the best case scenario. For the worst case scenario, when $k = 2$ and $k = 3$, the best velocities are obtained with 0.53 MB/s for Asmuth-Bloom and 1.12 MB/s for Mignotte. On the other hand, considering the values of Table 2, downloading segments from the CSPs, in the worst case scenario, showed a speed reduction of 1.75 MB/s for Mignotte and 0.544 MB/s for Asmuth-Bloom, in average, as it is shown in Fig. 7.

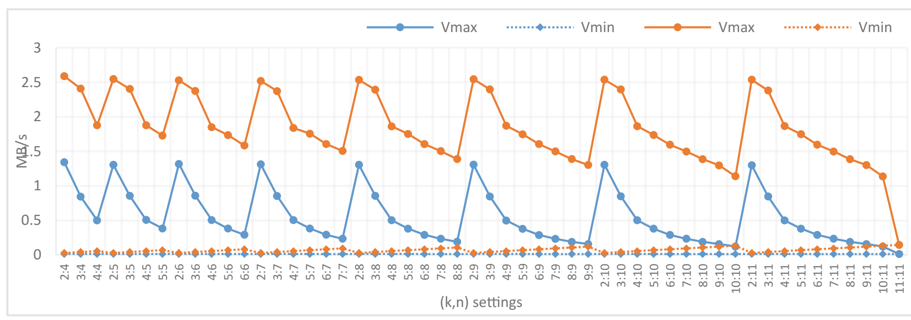


Fig. 7. Velocity of downloading k chunks and decoding based on Table 3.

The results show that Asmuth-Bloom has slower performance than Mignotte, especially, if the number of CSPs increases. However, Mignotte is computationally stable [28], while Asmuth-Bloom is the perfect and asymptotically ideal scheme [29]. Consequently, the user should make a choice between performance and data security. The advantage of a weighted Mignotte SSS is the optimal load balancing between CSPs [30]. In this paper, we do not analyze the data redundancy. In [31], it is shown that to minimize data redundancy of Asmuth-Bloom, it is recommended to use compact sequences of co-primes moduli.

4 Conclusions

In this paper, we address a multi-cloud secured storage environment comparing Asmuth-Bloom and Mignotte schemes, which rely on RRNS properties. We analyze the velocities of encoding/uploading/downloading/decoding for both schemes with 11 well-known cloud storage providers and different system parameters.

Asmuth-Bloom is a perfect and asymptotically ideal scheme. However, the simulations show that, in comparison with Mignotte, it has the slowest speeds for encoding/decoding with values of 0.463 MB/s and 0.420 MB/s, respectively. While, Mignotte shows speed 1.284 MB/s, for encoding, and 3.426 MB/s, for decoding. Hence, Mignotte is 8 times faster for decoding than Asmuth-Bloom.

The velocities of uploading/downloading from the cloud are obtained by two different scenarios of ordering the CSPs. In the first one, we upload data to the clouds with faster uploading access speed, trying to minimize uploading time, and, in the second scenario, with faster downloading speed trying to minimize downloading time.

Based on experimental analysis, we conclude that Mignotte outperforms 3.8 times Asmuth-Bloom for encoding and uploading data. If the cloud providers with the highest speed of access are unavailable, we show that it causes a velocity degradation for downloading-decoding. Here, the difference between the maximum and minimum velocities are 1.33 MB/s for Asmuth-Bloom and 2.50 MB/s for Mignotte, when CSPs are ordered by their download speeds. The behavior depicted in Figs. 5 and 6 show that important bottlenecks could reduce dramatically the access speed for uploading/downloading to the CSPs.

Hence, further study of variety of algorithms and multicriteria analysis are required to assess their actual efficiency and effectiveness. This will be subject of future work for better understanding of security, redundancy and speed in cloud storages.

References

1. AlZain, M.A., Pardede, E., Soh, B., Thom, J.A.: Cloud computing security: from single to multi-clouds. In: 2012 45th Hawaii International Conference on System Science (HICSS), pp. 5490–5499 (2012)
2. OpenFog Reference Architecture for Fog Computing. https://www.openfogconsortium.org
3. Tchernykh, A., Schwiegelsohn, U., Talbi, E., Babenko, M.: Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability. J. Comput. Sci. (2016). https://doi.org/10.1016/j.jocs.2016.11.011
4. Tchernykh, A., Schwiegelsohn, U., Alexandrov, V., Talbi, E.: Towards understanding uncertainty in cloud computing resource provisioning. Procedia Comput. Sci. **51**, 1772–1781 (2015). https://doi.org/10.1016/j.procs.2015.05.387
5. Tchernykh, A., Babenko, M., Chervyakov, N., Cortes-Mendoza, J., Kucherov, N., Miranda-Lopez, V., Deryabin, M., Dvoryaninova, I., Radchenko, G.: Towards mitigating uncertainty of data security breaches and collusion in cloud computing. In: Proceedings of UCC 2017, pp. 137–141. IEEE Press, Lyon (2017)
6. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 29–43. ACM, New York (2003)
7. Ganesan, A., Alagappan, R., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Redundancy does not imply fault tolerance: analysis of distributed storage reactions to single errors and corruptions. In: Proceedings of the 15th Usenix Conference on File and Storage Technologies, pp. 149–165. USENIX Association, Berkeley (2017)
8. Chen, X., Huang, Q.: The data protection of MapReduce using homomorphic encryption. In: 2013 IEEE 4th International Conference on Software Engineering and Service Science, pp. 419–421 (2013)
9. Celesti, A., Fazio, M., Villari, M., Puliafito, A.: Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. J. Netw. Comput. Appl. **59**, 208–218 (2016). https://doi.org/10.1016/j.jnca.2014.09.021
10. Chervyakov, N., Babenko, M., Tchenykh, A., Dvoryaninova, I., Kucherov, N.: Towards reliable low cost distributed storage in multi-clouds. In: 2017 International Siberian Conference on Control and Communications (SIBCON), pp. 1–6 (2017)
11. Dimakis, A.G., Godfrey, P.B., Wu, Y., Wainwright, M.J., Ramchandran, K.: Network coding for distributed storage systems. IEEE Trans. Inf. Theory **56**, 4539–4551 (2010). https://doi.org/10.1109/TIT.2010.2054295
12. Ateniese, G., Fu, K., Green, M., Hohenberger, S.: Improved proxy re-encryption schemes with applications to secure distributed storage. ACM Trans. Inf. Syst. Secur. **9**, 1–30 (2006). https://doi.org/10.1145/1127345.1127346
13. Chervyakov, N., Babenko, M., Tchernykh, A., Kucherov, N., Miranda-López, V., Cortés-Mendoza, J.M.: AR-RRNS: configurable, scalable and reliable systems for internet of things to ensure security. Future Gener. Comput. Syst. (2017). Elsevier. https://doi.org/10.1016/j.future.2017.09.061
14. Asmuth, C., Bloom, J.: A modular approach to key safeguarding. IEEE Trans. Inf. Theory **29**, 208–210 (1983). https://doi.org/10.1109/TIT.1983.1056651

15. Mignotte, M.: How to share a secret. In: Beth, T. (ed.) EUROCRYPT 1982. LNCS, vol. 149, pp. 371–375. Springer, Heidelberg (1983). https://doi.org/10.1007/3-540-39466-4_27
16. Rivest, R., Adleman, L., Dertouzos, M.: On data banks and privacy homomorphisms. In: Foundations of Secure Computation, pp. 169–177. Academic Press (1978)
17. Gentry, C.: A Fully Homomorphic Encryption Scheme (2009)
18. Soderstrand, M.A., Jenkins, W.K., Jullien, G.A., Taylor, F.J. (eds.): Residue Number System Arithmetic: Modern Applications in Digital Signal Processing. IEEE Press, Piscataway (1986)
19. Chervyakov, N., Babenko, M., Tchernykh, A., Nazarov, A., Garianina, A.: The fast algorithm for number comparing in three-modular RNS. In: 2016 International Conference on Engineering and Telecommunication (EnT), pp. 26–28 (2016)
20. Gomathisankaran, M., Tyagi, A., Namuduri, K.: HORNS: a homomorphic encryption scheme for Cloud Computing using Residue Number System. In: 2011 45th Annual Conference on Information Sciences and Systems (CISS), pp. 1–5 (2011)
21. Chessa, S., Maestrini, P.: Dependable and secure data storage and retrieval in mobile, wireless networks. In: Proceedings of the 2003 International Conference on Dependable Systems and Networks, pp. 207–216 (2003)
22. Chang, C.H., Molahosseini, A.S., Zarandi, A.A.E., Tay, T.F.: Residue number systems: a new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. IEEE Circuits Syst. Mag. **15**, 26–44 (2015). https://doi.org/10.1109/MCAS.2015.2484118
23. Lin, S.J., Chung, W.H., Han, Y.S.: Novel polynomial basis and its application to reed-solomon erasure codes. In: 2014 IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS), pp. 316–325 (2014)
24. Babenko, M., Chervyakov, N., Tchernykh, A., Kucherov, N., Shabalina, M., Vashchenko, I., Radchenko, G., Murga, D.: Unfairness correction in P2P grids based on residue number system of a special form. In: Proceedings of UCC 2017, pp. 147–151. IEEE, Lyon (2017)
25. Chen, H.C.H., Lee, P.P.C.: enabling data integrity protection in regenerating-coding-based cloud storage: theory and implementation. IEEE Trans. Parallel Distrib. Syst. **25**, 407–416 (2014). https://doi.org/10.1109/TPDS.2013.164
26. Pritzker, P., Gallagher, P.: SHA-3 standard: permutation-based hash and extendable-output functions (2014). National Institute of Standards and Technology. http://dx.doi.org/10.6028/NIST.FIPS.202
27. Chervyakov, N., Babenko, M., Deryabin, M., Garianina, A.: Development of information security's theoretical aspects in cloud technology with the use of threshold structures. In: 2014 International Conference on Engineering and Telecommunication, pp. 38–42 (2014)
28. Quisquater, M., Preneel, B., Vandewalle, J.: On the security of the threshold scheme based on the Chinese remainder theorem. In: Public Key Cryptography, pp. 199–210 (2002)
29. Kaya, K., Selçuk, A.A.: Threshold cryptography based on Asmuth-Bloom secret sharing. Inf. Sci. **177**, 4148–4160 (2007). https://doi.org/10.1016/j.ins.2007.04.008
30. Drăgan, C.C., Ţiplea, F.L.: Distributive weighted threshold secret sharing schemes. Inf. Sci. **339**, 85–97 (2016). https://doi.org/10.1016/j.ins.2016.01.019
31. Barzu, M., Ţiplea, F.L., Drăgan, C.C.: Compact sequences of co-primes and their applications to the security of CRT-based threshold schemes. Inf. Sci. **240**, 161–172 (2013). https://doi.org/10.1016/j.ins.2013.03.062

Bi-objective Heterogeneous Consolidation in Cloud Computing

Luis-Angel Galaviz-Alejos[1], Fermín Armenta-Cano[1], Andrei Tchernykh[1(✉)], Gleb Radchenko[2], Alexander Yu. Drozdov[3], Oleg Sergiyenko[4], and Ramin Yahyapour[5]

[1] CICESE Research Center, Ensenada, Baja California, Mexico
{lgalaviz,armentac}@cicese.edu.mx, chernykh@cicese.mx
[2] South Ural State University, Chelyabinsk, Russia
gleb.radchenko@susu.ru
[3] Moscow Institute of Physics and Technology, State University, Moscow, Russia
alexander.y.drozdov@gmail.com
[4] Universidad Autónoma de Baja California, Mexicali, Baja California, Mexico
srgnk@uabc.edu.mx
[5] University of Göttingen, Göttingen, Germany
ramin.yahyapour@gwdg.de

Abstract. In this paper, we address the problem of power-aware Virtual Machines (VMs) consolidation considering resource contention. Deployment of VMs can greatly influence host performance, especially, if they compete for resources on insufficient hardware. Performance can be drastically reduced and energy consumption increased. We focus on a bi-objective experimental evaluation of scheduling strategies for CPU and memory intensive jobs regarding the quality of service (QoS) and energy consumption objectives. We analyze energy consumption of the IBM System x3650 M4 server, with optimized performance for business-critical applications and cloud deployments built on IBM X-Architecture. We create power profiles for different types of applications and their combinations using SysBench benchmark. We evaluate algorithms with workload traces from Parallel Workloads and Grid Workload Archives and compare their non-dominated Pareto optimal solutions using set coverage and hyper volume metrics. Based on the presented case study, we show that our algorithms can provide the best energy and QoS trade-offs.

Keywords: Virtual machine · Consolidation · Energy aware scheduling SLA violations · Green cloud

1 Introduction

Facebook, Amazon, Apple, Microsoft, Google, Yahoo, and others IT companies are transforming the way in which we work, communicate, watch movies or TV, listen to music, and share files. The growth and scale of investment in the clouds are truly

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 384–398, 2018.
https://doi.org/10.1007/978-3-319-73353-1_27

mind-blowing with estimates of a 50-fold increase in the amount of digital information by 2020 [1]. In the report “Make IT Green: Cloud Computing and its Contribution to Climate Change” [2] published in March 2010, the authors studied the electricity demand of the Internet/Clouds. According to it, the combined electricity demand of the data centers and telecommunications network in 2007 was approximately 623 billion kWh. If we consider the clouds as a country, it would have the fifth largest electricity demand in the world. The growing demand for Cloud resources highlights the importance of advance techniques that help to save energy of data centers.

On a typical cloud environment, there is a group of hosts each one running a Virtual Machine Monitor (VMM), which enables the creation, management, and monitoring of Virtual Machines (VMs) and manages the operation of a virtualized environment. Each VM runs an application. The monitoring engine gathers processor, network interface, and memory usage, and other data for each host. This information is usually used by consolidation algorithms [3]. Server consolidation techniques pack a number of VMs on a fewer number of hosts to optimize resource utilization and reduce energy consumption. However, a naive consolidation can create resource contention, which results in a poor performance and high energy consumption. Cloud computing parameters and main sources of their uncertainty are discussed in [4].

Deployment of VMs can greatly influence host performance, especially, if many of them run on insufficient hardware. They can start competing for same hardware resources: CPU, memory, communication system, etc. Performance can be drastically reduced and energy consumption increased.

In our previous work [11], we proposed an energy model that consider different types of tasks. We propose Minimum Concentration (*Min_c)* policy that takes into to account application types to avoid resource contention and minimize energy consumption.

In this paper, we focus on a bi-objective experimental evaluation of this model for CPU and memory intensive jobs regarding QoS violations and energy consumption objectives. We use MOCell metaheuristic and compare it with multi-objective evolutionary algorithm NSGA-II, for different test cases.

The paper is structured as follows. Section 2 discusses related work. Section 3 presents the problem definition and proposed energy model, while the algorithms and operators are described in Sect. 4. Section 5 provides the experimental setup. Section 6 present methodology used to analyze results. Section 7 discusses simulation results. Finally, Sect. 8 concludes the paper by presenting main contribution and future work.

2 Related Work

There are several algorithms in the literature that study consolidation considering distinct parameters (Table 1).

In **MHCPESJ** [11], the authors propose an energy model that takes into account the effect on energy consumption of allocating applications that use distinct hardware resources. They propose an allocation heuristic method called “Minimal Concentration” (Min_c) that takes into to account the application types and their combinations.

In **UACSCV** [5], the authors present an algorithm based on ant colony system with an objective to reduce energy consumption considering VMs migrations and QoS. Their model considers CPU, memory, and network.

In **AESPP** [6], the authors consider a scheduling with task replications to overcome possible bad resource allocation in presence of uncertainty. They analyze speed-aware and power-aware scheduling algorithms with different grids and workloads considering two objectives: approximation factor and energy consumption. They show that by combining objective preferences and replication thresholds a compromise between better execution time and increased energy consumption can be achieved.

Table 1. Consolidation algorithms

Works	Parameters						Strategy	
	CPU	Memory	Net	Job types	Migration cost	Contention	Migration	Allocation
MHCPESJ	●	●		●		●		●
NAVMC		●	●				●	
HVCUIGG	●	●	●		●	●	●	
PACMan						●		●
NASVWIM	●		●	●	●	●		●
ESWCT	●	●	●	●		●	●	●
WCVCA	●		●	●				●
UACSCV	●	●	●				●	
AESPP			●	●				●
TNAVMECM		●	●		●		●	
DCB3T	●		●	●				●
MEWS	●							●

An excessive number of migrations could lead to network congestion in data centers and increment of energy consumption. The following consolidation algorithms consider VM migrations costs.

In **NAVMC** [7], and **TNAVMECM** [8], the authors evaluate VMPatrol and Remedy in a GENI testbed on a WAN network with realistic traffic. Both of them use models of migrations cost on a WAN.

In **HVCUIGG** [9], the authors use a genetic algorithm to consolidate VMs. They focus on reducing migration costs and saving energy. The parameters used for migration are the available bandwidth, the memory size of VM, the rate of dirty pages, and CPU utilization.

In **MEWS** [10], the authors present a multi-objective approach for scheduling large workflows in distributed datacenters to minimize makespan, energy consumption, and deadline violations.

Several authors consider consolidating of a distinct type of workloads:

In **ESWCT** [12], two algorithms are proposed: Workload-aware Consolidation Technique (ESWCT) and energy aware Live Migration Algorithm using Workload-aware Consolidation Technique (ELMWCT). These algorithms are based on the fact that cloud users share resources and distinct workloads have different energy consumption characteristics. The algorithms work with heterogeneous hosts and VMs with distinct requirements of CPU, memory, and network. However, this work does not consider the effect that the combination of distinct VM types to the same server has to energy consumption.

In **WCVCA** [13], two dynamic programming algorithms and an approximation algorithm for consolidation are presented. They consider data intensive and CPU intensive workloads.

In **DCB3T** [14], the authors propose a model of power and network-aware scheduling that can be tuned to achieve energy-savings, through job consolidation and traffic load balancing. They describe a methodology to find the best tuning of the adjustable scheduler for three types of workloads, computation-intensive, communication-intensive and balanced.

In **PACMan** [15], the authors present an algorithm that assigns VMs in a way that the performance is maintained between thresholds defined by the user while minimizing the used resources.

In **NASVWIM** [16], the authors present an interference model based on the characteristics of workloads used in a scheduler. Their model considers CPU, cache, metrics of disks and networks.

In this paper, we present workload-aware consolidation algorithms based on the energy model that takes into account the consumption profile of each application type (CPU Intensive and memory intensive), and their combination.

3 Problem Definition

We follow the system model presented in [11]. We consider a set of m heterogeneous hosts $(h_1, h_2, \ldots, h_m)$. Each host m_i is defined by its velocity in MIPS $\{s_i\}$. The set of n virtual machines $(VM_1, VM_2, \ldots, VM_n)$ executes independent tasks of a specific type, from the set $D = \{CI, MI\}$ that contains CPU Intensive (CI) and Memory Intensive (MI) jobs. The $k = |D|$ is the quantity of distinct VM types. Each VM_j is defined by the tuple $\{type_j, v_j, p_j\}$, where v_j is the requested speed of the VM_j in MIPS; $type_j \in D$ is the type of workload the VM_j executes, and p_j is the time required by the VM_j to finish its task.

The contributed utilization $u_{i,j}(t)$ of the VM_j to m_i at a time t is defined as $u_{i,j}(t) = v_j/s_i$. Let VMs of one type l are executed on m_i. The aggregated utilization $U_{i,l}(t)$ at time t is defined as $U_{i,l}(t) = \sum_{j=1}^{p} u_{i,j}(t)$, which is the sum of contributions to utilization of the p VMs of type l executing on m_i.

Our objective is to allocate VMs to hosts while reducing the energy consumption and the number of SLA violations.

In order to calculate the energy consumption, we consider the fraction of power consumption contributed by each type of VMs at time t, as a function of the host utilization at t. Hence, we have an independent function for each application type.

$$f_1\left(U_{i,1}(t)\right), f_2\left(U_{i,2}(t)\right), \ldots, f_k\left(U_{i,k}(t)\right), \tag{1}$$

where $f_l\left(U_{i,l}(t)\right) \rightarrow \{CI, MI\}$ represents the independent fraction of power consumption of a VM of type l on m_i.

We calculate $F_{i,T}(t)$ as the sum of all the power consumption fractions of each VM type, taking into account idle energy only once:

$$F_{i,T}(t) = \sum\nolimits_{i=1}^{k} f_l\left(U_{i,l}(t)\right), 0 \leq F_{i,T}(t) \leq 1 \tag{2}$$

In our first analysis, the quantity of distinct VM types (CI and MI) is limited to 2. We analyze a server energy consumption in a server Express x3650 M4, with two processors E5-2650v2, and default clock speed of 2.6 GHz. Each processor has 8 Cores and two threads per core (16 with hyperthreading), level 1 memory of 32 kB, level 2 of 256 kB, level 3 of 20 MB and a total memory of 64 GB (NUMA domain).

When executing different application types, we need to define realistic parameters for the model to take into account types of applications and their combinations.

In Fig. 1, we show the energy profile obtained with a specialized Power Distribution Unit (PDU) outlet metered when executing CI and MI applications versus CPU utilization. Based on this profile we calculate the fraction of power consumption $F_{i,T}(t)$.

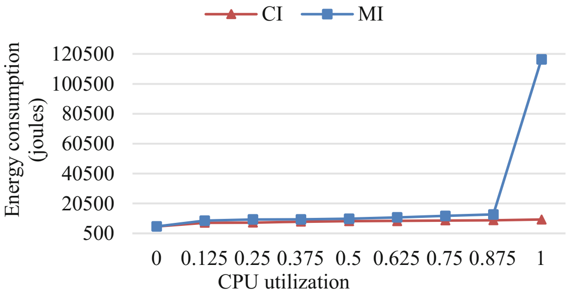


Fig. 1. Energy consumption versus CPU utilization.

In Fig. 2, we present the coefficient $g_{CI}\left(a_{i,l}(t)\right)$ that correct energy consumption due to combinations of types of applications.

If it equals to 1, each server has only one type of applications. If the server utilization reaches 100%, the server memory may be overcommitted, which results in degradation of the system performance and increasing energy consumption (due to thrashing the system spends more time in paging instead of their execution).

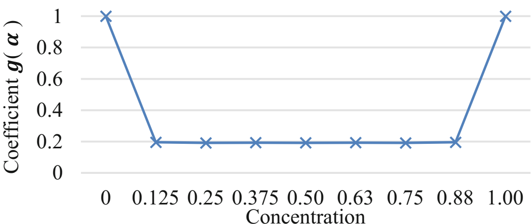


Fig. 2. Combination coefficient.

The allocation of distinct types of applications on the same host could help to reduce energy consumption, even less than the sum of the independent consumptions [11]. Besides, it has an impact on the performance by avoiding bottlenecks on hardware (CPU, memory, hard drive, etc.). Our energy model considers the effect of the execution of a combination of distinct types of VMs on a single host.

The total utilization of m_i is defined as $U_{i,T}(t) = \sum_{l=1}^{k} U_{i,l}(t)$. We define $a_{i,l}(t) = U_{i,l}(t)/U_{i,T}(t)$ as the concentration of VMs of type l in m_i at a time t.

As a first approach, we simplify our model considering only two types of applications. Based on the combination coefficient presented in Fig. 2, we consider a function $g_{CI}(a_{i,l}(t))$, where $g_{CI}(a_{i,l}(t))$ is a coefficient $(0 \le g_{CI}(a_{i,l}(t)) \le 1)$ that represent the effect of VM combination on a host. The energy consumption on host m_i at time t is given by:

$$e_i^{host}(t) = o_i(t)\left[e_{idle_i}^{host} + \left(e_{max_i}^{host} - e_{idle_i}^{host}\right) * F_{i,T}(t) * g_{CI}\left(a_{i,l}(t)\right)\right], \tag{3}$$

where $e_{idle_i}^{host}$ is the energy consumed when the host is idle, $e_{max_i}^{host}$ is the max energy of the host, and $o_i(t)$ is a function that gives the value 1, if the host is active, or 0, otherwise. If combinations are not considering, then $g_{CI}(a_{i,l}(t)) = 1$.

4 MOCell Algorithm (MO)

To minimize our objective functions, we use a variant of a Genetic Algorithm (GA) known as Multi-Objective Cellular Genetic Algorithm (MOCell) [17].

The main difference between MOCell and a classical GA is that in MOCell all the individuals in the population are placed on a toroidal grid (Fig. 3). The selection, crossover and mutation operators of the GA are executed only with their neighbors.

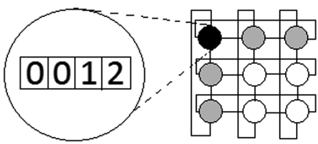


Fig. 3. Each individual on the population is placed on a toroidal grid. The GA operators are only done between an individual (black point) and his neighbors (gray point).

The individual is represented by an array of integers, where each slot represents a VM, and the value of the slot represents the host, where the VM is allocated. Figure 4 shows an example of an individual with five VMs and two hosts. Here, VM_1 is assigned to host one, VM_2 and VM_3 are assigned to host two, and so on.

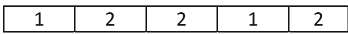


Fig. 4. An example of an individual chromosome.

4.1 GA Operators

An operator is a mechanism used by a GA to generate, combine and select new solutions. A typical GA has the operators: parent selection, crossover, mutation, and survival selection. Each one of these could greatly change the results obtained by the algorithm. We implemented and compared the behavior of MOCell using distinct operators with a multi-objective evolutionary algorithm, NSGA-II (NS) [18] with fixed operators. We want to find the MOCell operators that perform better to our problem.

We tested only one literature classical operator for parent and survival selection. The parent selection operator is called Binary Tournament, this randomly selects two parents, one from the neighborhood and one from the archive, it works by selecting two individuals and returns the non-dominated one, if both of them are not dominated, it returns one randomly. Survival selection mechanism is the same used on NSGA-II and presented in [18], it consists in dividing the population on non-dominated sub fronts where the best has rank 1, a solution is better than the others if it has a smaller rank. If the two solutions have the same rank, then the one with bigger crowding distance is selected.

For crossover operator, we use Uniform Crossover (UN), Single Point Crossover (SP), and the proposed operator called Uniform SLA Crossover (UNSLA). This operator combines a classical crossover schema UN with a repair step that limits the quantity of SLA violations on the offspring. The offspring has the same or smaller amount of SLA violations in comparison with its parents. It ignores exchanges that increase SLA violations. An example is presented in Fig. 5.

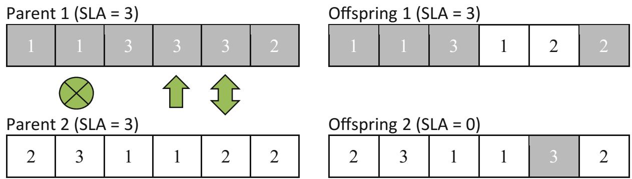


Fig. 5. An example of Uniform SLA Crossover. Darkly shaded genes correspond to parent 1, non-shaded genes to parent 2. In gene 2 the cross represents no exchange of value is done, on gene 4 the value is moved from parent 2 to 1, and on gene 5 both values are exchanged.

For mutation, we use two classic operators: BitFlip Mutation (BF) and Swap Mutation (SW). In addition, we propose new mutation operators: Balanced Concentration Mutation (BC), Swap VM Mutation (VMSW) and Local Search SLA (LS). These operators combine a classic mutation scheme with a repair step in order to obtain feasible solutions that substitute a host with SLA violation for one that does not has SLA violation.

BC selects two hosts randomly and allocates all the VMs evenly for each type between the hosts (Fig. 6). VMSW randomly selects a VM with SLA violation and moves it to a host with low utilization (Fig. 7). LS finds the overloaded hosts (the hosts that have the maximum number of VMs allocated) and allocates their VMs to the hosts with the lowest utilization (Fig. 8).

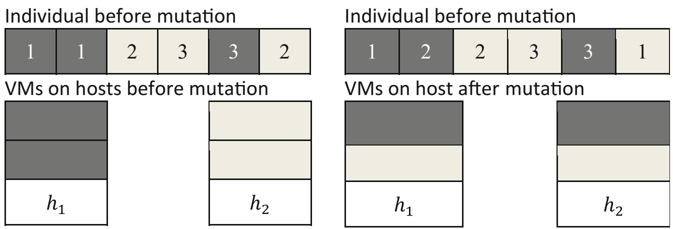


Fig. 6. An example of Balance Concentration Mutation. The lightly shaded cells represent CI, dark shaded cells represent MI.

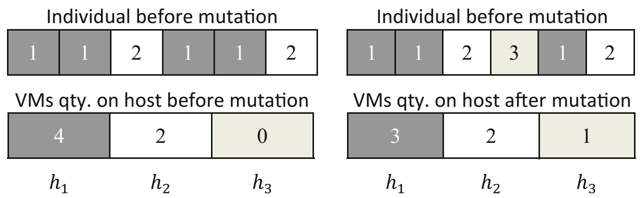


Fig. 7. An example of Swap VM Mutation. Darkly shaded cells represent the overloaded hosts. Lightly shaded cells represent the hosts with lower utilization. White cells represent the hosts with the lowest utilization.

Individual before mutation

1	3	3	3	3	1

Hosts Util. before mutation

	VMs Qty.	Utilization
h_1	2	100%
h_2	0	0%
h_3	4	200%

Individual after mutation

1	2	3	2	3	1

Hosts Util. after mutation

	VMs Qty.	Utilization
h_1	2	100%
h_2	2	100%
h_3	2	100%

Fig. 8. An example of LocalSearchSLA. Dark shaded cells represent the VMs with SLA violation. The host h_3 has the biggest utilization and their VMs are moved to the host with the lowest utilization (h_2) until all VMs of the host h_3 do not violate SLA.

5 Experimental Setup

Our 28 days workload is based on traces of real HPC from Parallel Workloads Archive, and Grid Workload Archive [19–21]. The workloads include nine traces from DAS2-University of Amsterdam, DAS2–Delft University of Technology, DAS2–Utrecht University, DAS2–Leiden University, KHT, DAS2–Vrije University Amsterdam, HPC2 N, CTC, and LANL.

We consider every task on the log as a VM with 1 MIPS. In Fig. 9, we can observe the distribution of VM tasks for 28 days, where CI represents CPU intensive VM, and MI represents memory-intensive VM.

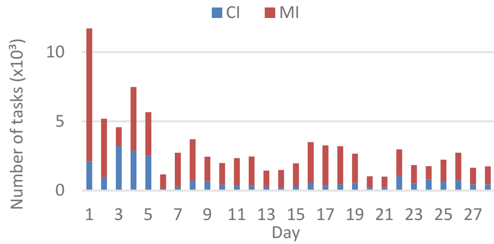


Fig. 9. VM types per day

In our environment, we consider that we have enough hosts to ensure no SLA violation. In a first test case, one host with a speed of 2 MIPS can have two VMs of 1 MIPS. The idle energy e^{host}_{idle} and the maximum energy e^{host}_{max} correspond to the values of Xeon IvyBridge processors E5-2650v2.

We use Java language (JDK 8) and JMetal Optimization framework 4.5.2 [22, 23], which includes metaheuristics such as NSGA-II, MOCell, SPEAII, etc.

We execute algorithms under parameters presented in Table 2. We study 14 algorithms for MOCell and 1 for NSGA-II described in Table 3.

Table 2. Configuration of GA operators

Population	100
Iterations	20,000
Crossover probability	0.9
Mutation probability	$1/n$

As a reference point, we calculate a lower bound for the test cases. These bound include two points: solution without SLA and solution with SLA equals to the number of VMs.

To calculate lower bound with no SLA, we order the CI and MI VMs decreasingly on its execution time, then, we allocate one VM of each kind on the host until it is full. The last step is to bound runtimes, so all the runtimes of the VMs on a host are set as the minimum runtime of all the VMs allocated on the host.

The solution with a maximum number of SLA violations is created by allocating all VMs on a single host and then bounding the runtimes with the same procedure for no SLA solution.

We obtain an approximation of Pareto front for each day, 28 fronts for each algorithm. We normalize data by a lower bound. We built a unique front for each algorithm by taking the nondominated solutions from all the fronts for each day.

Table 3. Tested algorithms

Algorithm	GA	Operators	
		Crossover	Mutation
MO-SP-BC	MOCell	Single point	Balance concentration
MO-SP-BF	MOCell	Single point	Bit flip
MO-SP-SW	MOCell	Single point	Swap
MO-SP-VMSW	MOCell	Single point	Swap VM
MO-UN-BC	MOCell	Uniform	Balance concentration
MO-UN-BF	MOCell	Uniform	Bit flip
MO-UN-SW	MOCell	Uniform	Swap
MO-UN-VMSW	MOCell	Uniform	Swap VM
MO-UN-LS	MOCell	Uniform	Local search SLA
MO-UNSLA-BC	MOCell	Uniform SLA	Balance concentration
MO-UNSLA-BF	MOCell	Uniform SLA	Bit flip
MO-UNSLA-SW	MOCell	Uniform SLA	Swap
MO-UNSLA-VMSW	MOCell	Uniform SLA	Swap VM
NS-UN-BF	NSGA II	Uniform	Bit flip

6 Methodology Used for the Analysis

In a multi-objective problem, in order to evaluate each algorithm, we need a method to measure the quality of the fronts. On this paper, we use two quality indicators: Set Coverage and Hyper volume. Both indicators were proposed on [24]. For hyper volume we use performance degradation [25] to measure how much each algorithm differs from the best found, this is calculated as follows: $\frac{best\ volume}{algorithm\ volume} - 1$. We present two degradations, absolute and relative. The first degradation it is the degradation taking in account the lower bound.

6.1 Set Cover

We use Set Coverage *SC(A, B)* to analyze the bi-objective problem. It is a formal and statistical metric that calculates the proportion of solutions in B, which are dominated by solutions in A:

$$SC(A,B) = \frac{|\{b \in B; |\exists a \in A; a \leq b\}}{|B|} \tag{4}$$

A metric value *SC(A, B)* = 1 means that all solutions of *B* are dominated by *A*, whereas *SC(A, B) = 0* means that no member of *B* is dominated by *A*. This way, the larger the value of *SC(A, B),* the better the Pareto front *A* with respect to *B*. Since the dominance operator is not symmetric, *SC(A, B)* is not necessarily equal to *1 − SC(A, B)*, and both *SC(A, B)* and *SC(B, A)* have to be computed for understanding how many solutions of *A* are covered by *B* and vice versa.

6.2 Hyper Volume

This indicator calculates the volume of the multi-dimensional region enclosed between a reference point and the points in the front. This metric has the advantage being independent (is a property of the front). Its concept is intuitive and allows front ranking according to this value. In Fig. 10, we observe an example of hyper volume indicator.

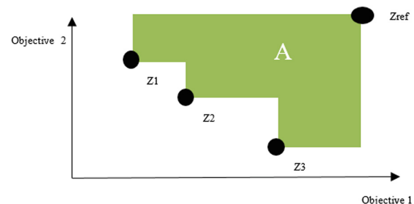


Fig. 10. Hyper volume for a bi-objective front Z1, Z2, Z3 and a reference point Zref.

7 Experimental Results

We address a bi-objective consolidation problem of minimizing power consumption and SLA violations.

The non-dominated fronts obtained for each algorithm are presented in Fig. 11. MO-UNSLA-BF, MO-UNSLA-SW, MO-UNSLA-VMSW and, MO-UNSLA-BC strategies are in the lower-left corner, being among the best solutions in terms of SLA violations.

For SLA violation degradation, MO-UNSLA-BF obtains values between 6 and 131, MO-UNSLA-SW between 11 and 93, MO-UNSLA-VMSW between 0 and 159 and MO-UNSLA-BC between 14 and 454.

On the other hand, the best algorithms for energy minimization objective are NS-UN-BF, MO-UN-BF, MO-SP-BF. NS-UN-BF. Their degradation of energy from the best found are: for NS-UN-BF between 0 and 3.48, MO-UN-BF between 0.05 and 2.56, and MO-SP-BF between 0.39 and 3.02.

Table 4 reports the SC results for each of the 14 Pareto fronts. The rows of the table show the values SC(A, B) for the dominance of strategy A over strategy B. The columns indicate SC(B, A), that is, dominance of B over A. The last two columns show the average of SC(A, B) for row A over column B, and ranking based on the average dominance.

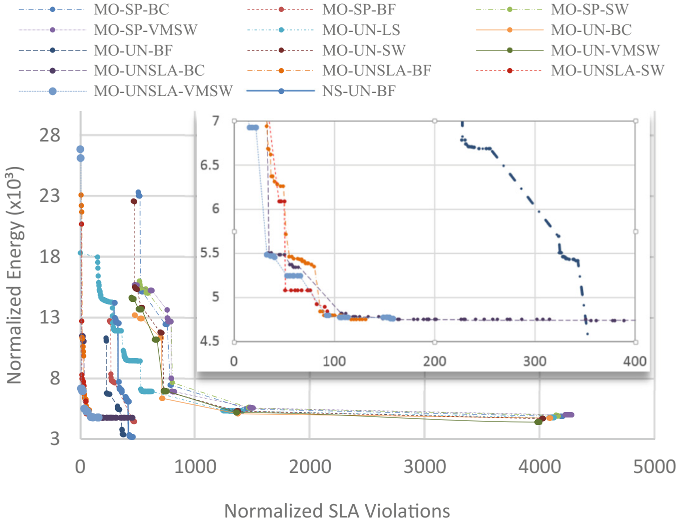


Fig. 11. The non-dominated fronts obtained for each algorithm.

We see that SC (MO-UNSLA-VMSW, B) dominates the fronts of the other strategies in the range between 44% and 100% and dominates by 82% in average. SC (A, MO-UNSLA-VMSW) shows that MO-UNSLA-VMSW is dominated by the fronts of other strategies in 11% in average.

The ranking of strategies is based on the percentage of coverage. The higher mean dominance of rows implies that the fronts are better. The rank in columns shows that the smaller the average dominance, the better the strategy. According to the set coverage metric table, the strategy that has the best compromise between minimizing SLA violation and energy consumption is MO-UNSLA-VMSW, followed by MO-UNSLA-BC, MO-UNSLA-BF, and MO-UNSLA-SW. MO-UNSLA-VMSW scores 0.82 of mean dominance and mean dominated of 0.11, MO-UNSLA-BC scores 0.75 and 0.15, MO-UNSLA-SW scores 0.75 and 0.18 and MO-UNSLA-SW scores 0.75 and 0.13.

Table 5 presents the hyper volume and rank in dominance over other algorithms. According to the hyper volume metric, the best algorithms are MO-UN-BF, MO-UNSLA-BC, MO-UNSLA-SW, MO-UNSLA-BF, and MO-UNSLA-VMSW. These algorithms show a performance difference less than 1.2% according to hyper volume relative degradation.

According to both metrics, MO-UN-BF is the best strategy in energy minimization with lowest absolute degradation value of 20.79%. On the other hand, the strategies MO-UNSLA-BC, MO-UNSLA-SW, MO-UNSLA-BF, and MO-UNSLA-VMSW are the best for SLA minimization with differences of relative hyper volume less than 0.5%. The worst algorithms by both metrics are MO-UN-SW, MO-SP-SW, MO-SP-BC, and MO-SP-VMSW with a relative hyper volume degradation greater than 20% and rank greater than 11. With the exception of MO-UN-SW, the worst algorithms use SP crossover.

Table 4. Set coverage.

	MO-SP-BC	MO-SP-BF	MO-SP-SW	MO-SP-VMSW	MO-UN-BC	MO-UN-BF	MO-UN-LS	MO-UN-SW	MO-UN-VMSW	MO-UNSLA-BC	MO-UNSLA-BF	MO-UNSLA-SW	MO-UNSLA-VMSW	NS-UN-BF	Mean	Rank
MO-SP-BC	1.00	0.00	0.44	0.74	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.16	**12**
MO-SP-BF	1.00	1.00	1.00	1.00	1.00	0.00	0.89	1.00	0.83	0.00	0.00	0.00	0.00	0.27	0.57	**7**
MO-SP-SW	0.07	0.00	1.00	0.55	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	**13**
MO-SP-VMSW	0.22	0.00	0.11	1.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.10	**14**
MO-UN-BC	1.00	0.00	1.00	1.00	1.00	0.00	0.23	0.69	0.44	0.00	0.00	0.00	0.00	0.00	0.38	**10**
MO-UN-BF	1.00	1.00	1.00	1.00	1.00	1.00	0.91	1.00	1.00	0.13	0.00	0.00	0.00	0.73	0.70	**5**
MO-UN-LS	1.00	0.00	1.00	0.90	0.52	0.00	1.00	0.76	0.76	0.00	0.05	0.04	0.06	0.24	0.45	**8**
MO-UN-SW	0.96	0.00	1.00	0.86	0.28	0.00	0.25	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.31	**11**
MO-UN-VMSW	1.00	0.00	1.00	0.87	0.43	0.00	0.25	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.40	**9**
MO-UNSLA-BC	1.00	0.74	1.00	1.00	0.72	0.81	1.00	0.90	0.83	1.00	0.36	0.16	0.21	0.73	0.75	**3**
MO-UNSLA-BF	1.00	0.74	1.00	1.00	0.72	0.81	1.00	0.90	0.83	0.33	1.00	0.16	0.24	0.73	0.75	**2**
MO-UNSLA-SW	1.00	0.74	1.00	1.00	0.72	0.81	0.86	0.90	0.83	0.22	0.45	1.00	0.18	0.73	0.75	**4**
MO-UNSLA-VMSW	1.00	0.74	1.00	1.00	0.72	0.81	0.86	0.90	0.83	0.44	0.79	0.60	1.00	0.73	0.82	**1**
NS-UN-BF	1.00	0.26	1.00	1.00	1.00	0.00	0.76	1.00	1.00	0.03	0.00	0.00	0.00	1.00	0.58	**6**
Mean	0.88	0.37	0.90	0.93	0.61	0.28	0.58	0.74	0.62	0.15	0.18	0.13	0.11	0.41		
Rank	**12**	**6**	**13**	**14**	**9**	**5**	**8**	**11**	**10**	**3**	**4**	**2**	**1**	**7**		

Table 5. Comparison results.

Algorithm	Hyper volume			Rank SC
	($x10^7$)	Absolute degradation (%)	Relative degradation (%)	
Lower bound	11.50		–	–
MO-UN-BF	9.49	20.79	0.00%	5
MO-UNSLA-VMSW	9.43	21.56	0.64%	1
MO-UNSLA-BF	9.42	21.74	0.78%	2
MO-UNSLA-BC	9.42	21.75	0.79%	3
MO-UNSLA-SW	9.39	22.14	1.12%	4
NS-UN-BF	9.36	22.44	1.37%	6
MO-SP-BF	8.96	27.95	5.93%	7
MO-UN-LS	8.68	32.11	9.38%	8
MO-UN-BC	8.03	42.71	18.15%	10
MO-UN-VMSW	8.01	43.22	18.57%	9
MO-UN-SW	7.91	44.96	20.01%	11
MO-SP-VMSW	7.72	48.49	22.94%	14
MO-SP-BC	7.68	49.25	23.56%	12
MO-SP-SW	7.63	50.24	24.39%	13

8 Conclusion

In this paper, we conduct a bi-objective analysis for the consolidation problem of 15 algorithms with the objectives of SLA violations minimization and energy optimization. With the power profiles of CPU and memory intensive jobs and their combination, obtained on real 16 core server, we design a heterogeneous power-aware model for energy consumption estimation.

Simulation results reveal that MO-UN-BF outperforms NS-UN-BF in terms of hyper volume by 1.37% and SC by 12% (see Tables 4 and 5). These strategies are MOCell and NSGA-II using the same crossover and mutation operators.

The best strategies in terms of SLA violation are MO-UNSLA-VMSW, MO-UNSLA-BC, MO-UNSLA-BF, and MO-UNSLA-SW. However, the strategy MO-UNSLA-VMSW has the best absolute values. In SLA violation degradation, it obtains values between 0 and 159. In SC, it dominates the fronts of the other strategies in the range between 44% and 100%, and dominates in 82%, in average. The relative hyper volume degradation is less than 1%.

For the energy optimization point of view, the strategy MO-UN-BF is the best in hyper volume analysis. There are only 0.05 and 2.56 times worse than the lowest possible bound. According to SC, it dominates the fronts of the other strategies in the range between 0% and 100%, and dominates 70%, on average.

References

1. Cook, G.: How clean is your cloud. Catal. Energy Revolut. **52**, 1–52 (2012)
2. Greenpeace International: Make IT Green: Cloud Computing and Its Contribution to Climate Change, pp. 1–12. Greenpeace International, Amsterdam (2010)
3. Varasteh, A., Goudarzi, M.: Server consolidation techniques in virtualized data centers: a survey. IEEE Syst. J. **11**(2), 772–783 (2015). https://doi.org/10.1109/JSYST.2015.2458273
4. Tchernykh, A., Schwiegelsohn, U., Talbi, E., Babenko, M.: Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability. J. Comput. Sci. (2016). https://doi.org/10.1016/j.jocs.2016.11.011
5. Farahnakian, F., Ashraf, A., Pahikkala, T., Liljeberg, P., Plosila, J., Porres, I., Tenhunen, H.: Using ant colony system to consolidate VMs for green cloud computing. IEEE Trans. Serv. Comput. **8**, 187–198 (2015). https://doi.org/10.1109/TSC.2014.2382555
6. Tchernykh, A., Pecero, J.E., Barrondo, A., Schaeffer, E.: Adaptive energy efficient scheduling in Peer-to-Peer desktop grids. Futur. Gener. Comput. Syst. **36**, 209–220 (2014). https://doi.org/10.1016/j.future.2013.07.011
7. Maziku, H., Shetty, S.: Network aware VM migration in cloud data centers. In: 2014 3rd GENI Research and Educational Experiment Workshop, pp. 25–28 (2014). https://doi.org/10.1109/GREE.2014.18
8. Maziku, H., Shetty, S.: Towards a network aware VM migration: evaluating the cost of VM migration in cloud data centers. In: 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet). pp. 114–119. IEEE (2014)

9. Wu, Q., Ishikawa, F.: Heterogeneous virtual machine consolidation using an improved grouping genetic algorithm. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pp. 397–404. IEEE (2015)
10. Nesmachnow, S., Iturriaga, S., Dorronsoro, B., Tchernykh, A.: Multiobjective energy-aware workflow scheduling in distributed datacenters. In: Gitler, I., Klapp, J. (eds.) ISUM 2015. CCIS, vol. 595, pp. 79–93. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32243-8_5
11. Armenta-Cano, F.A., Tchernykh, A., Cortes-Mendoza, J.M., Yahyapour, R., Drozdov, A.Y., Bouvry, P., Kliazovich, D., Avetisyan, A., Nesmachnow, S.: Min_c: heterogeneous concentration policy for energy-aware scheduling of jobs with resource contention. Program. Comput. Softw. **43**, 204–215 (2017). https://doi.org/10.1134/S0361768817030021
12. Hongyou, L., Jiangyong, W., Jian, P., Junfeng, W., Tang, L.: Energy-aware scheduling scheme using workload-aware consolidation technique in cloud data centres. China Commun. **10**, 114–124 (2013). https://doi.org/10.1109/CC.2013.6723884
13. Yang, J.S., Liu, P., Wu, J.J.: Workload characteristics-aware virtual machine consolidation algorithms. In: CloudCom 2012 – Proceedings of 2012 4th IEEE International Conference on Cloud Computing Technology and Science, pp. 42–49 (2012). https://doi.org/10.1109/CloudCom.2012.6427540
14. Combarro, M., Tchernykh, A., Kliazovich, D., Drozdov, A., Radchenko, G.: Energy-aware scheduling with computing and data consolidation balance in 3-tier data center. In: 2016 International Conference on Engineering and Telecommunication (EnT), pp. 29–33. IEEE (2016)
15. Nath, A.R., Kansal, A., Govindan, S., Liu, J., Suman, N.: PACMan: performance aware virtual machine consolidation. In: 10th International Conference on Autonomic Computing, ICAC 2013, San Jose, CA, USA, pp. 83–94, 26–28 June 2013
16. Verboven, S., Vanmechelen, K., Broeckhove, J.: Network aware scheduling for virtual machine workloads with interference models. IEEE Trans. Serv. Comput. **8**, 617–629 (2015). https://doi.org/10.1109/TSC.2014.2312912
17. Nebro, A.J., Durillo, J.J., Luna, F., Dorronsoro, B., Alba, E.: A cellular genetic algorithm for multiobjective optimization. In: Proceedings of Workshop on Nature inspired cooperative strategies for optimization, NICSO 2006, pp. 25–36 (2006)
18. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002). https://doi.org/10.1109/4235.996017
19. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. J. Parallel Distrib. Comput. **74**(10), 2967–2982 (2014). https://doi.org/10.1016/j.jpdc.2014.06.013
20. Parallel Workload Archive. http://www.cs.huji.ac.il/labs/parallel/workload/
21. Tchernykh, A., Lozano, L., Schwiegelshohn, U., Bouvry, P., Pecero, J.E., Nesmachnow, S., Drozdov, A.Y.: Online bi-objective scheduling for IaaS clouds ensuring quality of service. J. Grid Comput. **14**, 5–22 (2016). https://doi.org/10.1007/s10723-015-9340-0
22. Durillo, J.J., Nebro, A.J., Alba, E.: The jMetal framework for multi-objective optimization: design and architecture. Evol. Comput. **5467**, 18–23 (2010). https://doi.org/10.1109/CEC.2010.5586354
23. Durillo, J.J., Nebro, A.J.: jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011). https://doi.org/10.1016/j.advengsoft.2011.05.014
24. Zitzler, E.: Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. http://www.tik.ee.ethz.ch/sop/publicationListFiles/zitz1999a.pdf, (1999)
25. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. Ser. B. **91**, 201–213 (2002). https://doi.org/10.1007/s101070100263

Scaling the Deployment of Virtual Machines in UnaCloud

Jaime Chavarriaga[1(✉)], César Forero-González[1], Jesse Padilla-Agudelo[1], Andrés Muñoz[1], Rodolfo Cáliz-Ospino[2], and Harold Castro[1]

[1] COMIT Research Group, Universidad de los Andes, Bogotá, Colombia
{ja.chavarriaga908,ca.forero10,pa.jesse10, af.munoz1477,hcastro}@uniandes.edu.co

[2] Centro de Computación de Alto Desempeño (CECAD), Universidad Distrital Francisco José de Caldas, Bogotá, Colombia
cecad@udistrital.edu.co

Abstract. UnaCloud is an Opportunistic Cloud Platform that allows to create virtual clusters in non-dedicated hardware by harvesting idle resources in computer rooms across a campus. To launch a virtual cluster, the platform first determines which desktops can run the virtual machines and copies the corresponding images to these computers, mostly located in the same room. Regretfully, UnaCloud uses a TCP-based protocol to copy those images that results in large transmission times. Our diagnostics shows that the main cause for errors when deploying multiple machines is reaching a timeout. This paper reports our efforts to scale the deployment in UnaCloud to support clusters with a large number of nodes. We have implemented and evaluated multiple protocols for transferring virtual machine images. Our tests showed that BitTorrent, a P2P file transfer protocol, outperforms copying a single image using other protocols. Using it, we can deploy up to 100 virtual machines, one per desktop, in less than 10 min. Although this time is twice the offered by Amazon EC2, it is better than the exhibited by dedicated private clouds using software such as OpenStack and VMWare vCloud.

Keywords: Cloud provisioning · Virtual machine images · BitTorrent

1 Introduction

UnaCloud[1] is an Opportunistic Cloud Platform [11]. It harvests idle resources of computers in a campus to offer Infrastructure as a Service (IaaS). Using this platform, *Cloud users* can define and use virtual clusters that run on non-dedicated computers. For instance, in a university, researchers may use UnaCloud to execute a cluster of virtual machines using desktop computers in the academic campus that are idle or are being used below their capacity.

[1] https://sistemasproyectos.uniandes.edu.co/iniciativas/unacloud/es/inicio/.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 399–413, 2018.
https://doi.org/10.1007/978-3-319-73353-1_28

As in other IaaS platforms, Cloud users must first define the images of the virtual machines for the clusters they want to run on UnaCloud. When these users decide to use them, the platform copies these images to the hardware that run the virtual machines and starts the corresponding instances. The time spent copying the images is key to initialize smoothly the virtual infrastructure requested by the users.

Nowadays, UnaCloud creates copies of the images of virtual machines using a custom TCP-based protocol. The images are stored permanently in a central server and copied on-demand to the machines where the virtual machines will run. Regretfully, this process may spent a lot of time, specially when the same image must be copied to a large number of machines and when the size of the image is significant. This paper presents our efforts to scale the process to deploy the virtual machines to multiple desktops.

Considering that the typical installations of UnaCloud use computer rooms and labs located in an university, we implemented and evaluated diverse protocols that may reduce the transfer time in these environments. According to our evaluations, BitTorrent exploits the presence of multiple computers in the same room to transfer the image files in less time. As a result, we have developed UnaCloud P2P, an extension that transfers the images using BitTorrent and supports the deployment of tens of virtual machines in less than 10 min. This time is a big improvement over the current implementation and it is comparable to the offered by other platforms. This paper presents the evaluations we have done and the extensions we made to UnaCloud in order to scale the deployment process.

The rest of this paper is organized as follows: Sect. 2 presents the process implemented in UnaCloud to deploy virtual machines and the problems caused by delays in the transmission. Section 3 presents diverse protocols that may be used to reduce the time spent copying the images. Section 4 compares these protocols to determine which one offers the best option for transferring large images to multiple nodes. Section 5 evaluates the performance of the UnaCloud deployment process using the protocol that exhibited the best performance and Sect. 6 compares it with existing public and private cloud platforms. Finally, Sect. 8 concludes the paper.

2 Image Provisioning in UnaCloud

Reducing the time spent copying virtual machine images has been a challenge during the development of UnaCloud. The main reason for failures while deploying virtual machines is reaching a timeout in this task. This section presents the current approach and the problems we have detected.

2.1 UnaCloud Architecture

UnaCloud harvests idle computing resources from selected desktop workstations in an organization. The platform deploys in these desktops virtual machines

that run in parallel to the tasks performed by an end user. In order to work, it requires that some software components must be installed in the organization.

Figure 1 shows an overview of the UnaCloud Architecture. It comprises three elements: (1) A *UnaCloud server* that maintains information of all the virtual machines and services running in the platform and interacts with the cloud users by a web-based interface, and (2) A *UnaCloud File server* that stores the images for the virtual images, and (3) a set of *UnaCloud agents* that run on each participating desktop receiving commands from the server, controlling the virtual machines running in that node and reporting information to the UnaCloud Server.

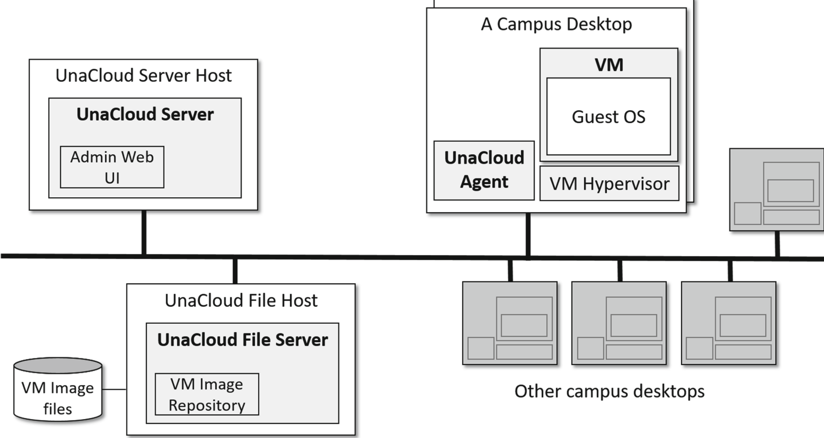


Fig. 1. Overview of the UnaCloud Architecture

Cloud users interact with the UnaCloud servers to upload the images of the virtual machines they want to use, to deploy a number of virtual machines with these images and to access and manage those virtual machines while they are running.

2.2 Deployment of Virtual Machines

When a Cloud user requests to deploy a number of machines, the *UnaCloud server* must locate the image for the virtual machines and instruct the corresponding *UnaCloud agents* to obtain a copy of the image from the *UnaCloud File Server* and start a virtual machine using that image.

In UnaCloud, the *deployment process* starts when a Cloud user requests a virtual machine and ends when that machine is running in a participating desktop or when an error occurs. During that process, each virtual machine has one of the 10 states described in the Fig. 2. First, the virtual machine is QUEUED until the UnaCloud Server has determined which desktop (a.k.a. node) has the

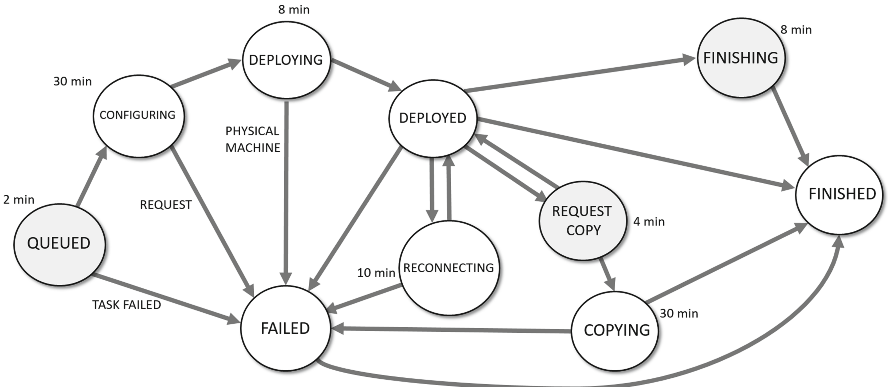


Fig. 2. States for deploying a virtual machine in UnaCloud

resources to run it. Then, it starts tasks for CONFIGURING the virtualization software on that node and DEPLOYING the virtual machine. If that desktop has a copy of the image of the virtual machine, e.g. if that node has run it before, the virtual machine can be DEPLOYED after starting its execution. If the image is not available in that desktop, CONFIGURING the machine requires to transfer the image from the server before get DEPLOYED.

Each one of the states during the deployment implies a set of tasks that must be accomplished in a predefined time. If the tasks are not performed on time, the process is halted and the virtual machine turns FAILED. The process may halt by a request of the user, an error or a timeout.

Deployment time ($T_{deployment}$), i.e. the time for deploying a virtual machine, is measured by the time that it takes to go from QUEUED to DEPLOYED. When multiple virtual machines are requested, the deployment time is determined by the maximum $T_{deployment}$ recorded across all the machines.

Transfer time ($T_{transfer}$), i.e. the time for copying the image of a virtual machine, is measured by the agent taking a time-stamp when the images are requested and another time-stamp when the files are completely received. When requesting multiple virtual machines, the transfer time is determined by the maximum $T_{transfer}$ of these machines.

2.3 Errors in the Deployment Process

In UnaCloud, the deployment of a virtual machine may fail for many reasons: (1) *An Error in the Image* that prevents the virtual machine to start, e.g. when the VM has configuration errors; (2) *An Error in the Network* that prevents the image from being copied to the destination machine, e.g. when a switch fails; (3) *An Error in the Desktop* that prevents the virtual machine to run, e.g. when the desktop is blocked or is rebooted; and (4) *A Delay in the Transmission* that causes a timeout in the deployment process.

While the errors in the configuration of the virtual machines can be attributed to the users and the problems with the hardware can be fortuitous, we consider that the failures caused by delays in the transmission must be handled by the platform.

Table 1. UnaCloud behaviour when deploying a large number of instances

# ins.	time	reliab.	% channel	payload (GB)	overhead (GB)	% overhead
1	330,33	100%	19,60%	1,620	0,061	3,8%
5	340,53	100%	71,20%	8,100	1,567	19,3%
10	394,30	100%	72,00%	16,200	9,679	59,7%
25	605,61	100%	72,10%	40,500	21,529	53,2%
50	959,73	98%	72,00%	81,000	41,618	51,4%
75	1331,16	99%	72,10%	119,880	61,091	51,0%
100	2908,37	99%	72,00%	162,000	80,917	49,9%

time: deployment time for all the n instances, given in seconds.
reliab.: reliability, i.e. number of correctly deployed instances over the total.
% channel: peek of channel utilization.
payload: total size of the images transmitted.
overhead: total size of data transmitted.
% overhead: ratio of the overhead over the payload.

Table 1 describes the behaviour of the deployment process when the number of requested machines increases. It presents the time for deploying a number of virtual machines from a single image file of 3.1 GB that, after compressed, is transmitted as a file of 1.62 GB. These values represent the average of three executions of each deployment in real infrastructure of the Universidad de los Andes, where the participant desktops were not used by final users and the communication problems were the only possible source of deployment errors.

Note that the reliability of the process, i.e. the number of correctly deployed virtual machines over the total number of requested ones, decreases as the number of instances grew. For more than 50 instances, there was 2 or 3 failed deployments every time, caused by reaching a timeout in the TCP communication. Also note that the level of utilization of the channels of the servers increased. Deployments with more than 5 instances consumed more than 70% of the channel. The deployment time increases too. For 100 instances, the spent time is more than 2900 s. Considering that starting the virtual machines takes 244 to 250 s, that means a transfer time of more than 2600 s. In fact, that is a reason because we cannot deploy that number of machines in the University, because that time is higher than the 30 min (1800 s) that we have configured as the limit for timeout in UnaCloud.

Our evaluations showed that deploying a large number of instances in the current implementation of UnaCloud produces a high congestion in the network and, therefore, takes a lot of time. In order to scale the process and be able to

deploy hundreds of virtual machines, it is necessary to reduce the congestion and the transfer time.

3 Alternative Protocols that May Improve Image Transferring

To scale the deployment process in UnaCloud, we started to consider alternative protocols that may reduce the transfer time and the congestion in the network segments. This section presents the diverse alternatives we considered.

3.1 Transfer Protocol in the Current Implementation

The current implementation of UnaCloud relies on a *TCP-based file transfer* that we will name **UnaCloud TCP** for this paper. This protocol takes advantage of the reliability of the TCP-transport to copy the files without including complex processes to check the result at the target machines.

UnaCloud TCP is our current implementation. It (1) creates a socket-based communication between the server and each client, and (2) transfers a stream of bytes obtained by reading the file. The socket is configured to use the TCP transport that guarantees completeness and data integrity.

3.2 Alternative Transfer Protocols

Existing literature describes many protocols aimed to transfer files between computers that may produce better results than our implementation. On the one hand, there are some modern protocols that have been reported as more-performing. For instance, there are protocols such as SMB that may use different transports, e.g. UDP or TCP, trying to reduce the overhead and transfer files in less time. On the other hand, there are protocols where the clients requesting the same file cooperate to reduce the transfer time. Protocols such as BitTorrent defines Peer-to-Peer (P2P) overlay networks where each node can obtain and serve content and can coordinate with the others to reduce the transfer time.

Considering the alternatives, we evaluated three different protocols: (1) **FTP**, a traditional protocol based on TCP, (2) **SMB**, a modern disk sharing protocol that may use TCP or UDP in local networks, and (3) **BitTorrent**, a widely used protocol for sharing files in P2P.

File Transfer Protocol (FTP) [10] is probably the most known protocol for transferring files between a client and a server. It was defined at the beginning of the Internet [2] and has been updated constantly since then. Nowadays, command-line FTP clients are shipped with most Microsoft Windows, Unix and Linux operating systems. In addition, web browsers such as Internet Explorer and Chrome supports fetching files using this protocol.

Server Message Block (SMB) [8] is an application-layer protocol well known for providing shared access to files, printers and serial ports for networks based on Microsoft Windows. It is supported by the Samba project[2] in other

[2] https://www.samba.org.

operating systems. SMB runs on top of diverse transport protocols using a thin layer known as NetBIOS [9].

BitTorrent [3] is a protocol widely used to share large files in the Internet[3]. In this protocol, files are transmitted by chunks from multiple machines that hold copies of the file. Each node can request for a chunk, and later transmit to others that may need it. One of the nodes is a *tracker* that maintains information of which nodes have which chunks. The other nodes are *clients* that interact with the tracker and with the other nodes trying to obtain the chunks from a near-location machine and reduce the transfer time [12].

4 Evaluating the Protocols for Transferring Files

To improve the deployment process our first step aimed to determine which alternative protocol can be used to transfer image files. We defined a test suite to compare the performance of the protocols mentioned above. The best performing protocol in our tests,i.e. the one that minimizes transfer errors and transfer time, will be used to replace the current transfer protocol in the platform.

4.1 Test Suite

Our tests for evaluating protocols for transferring files consisted on measuring the *transfer time* ($T_{transfer}$) of a virtual machine image of 3.1 GB that results in a file of 1.7 GB after ZIP compression. The same image was transferred from a single UnaCloud server to a varying number of desktops running a UnaCloud agent in each one. We defined seven scenarios by varying the number of desktop machines between 1, 5, 10, 25, 50, 75 and 100. For each scenario, the tests for each protocol was repeated three times to reduce random errors obtaining the time.

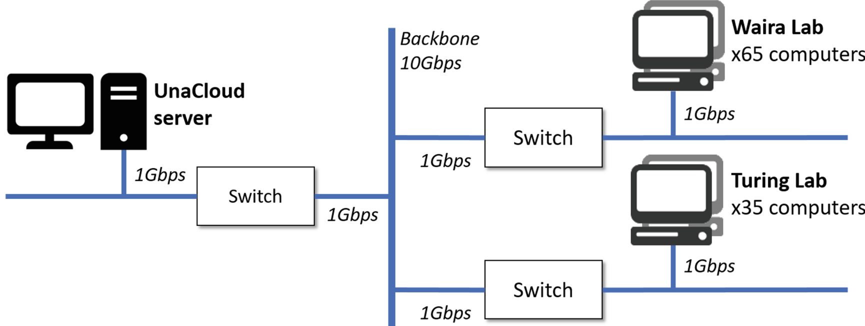


Fig. 3. Network elements used in the deployment of virtual machines

[3] http://www.bittorrent.org.

The tests were performed in two laboratories (the *Waira* and the *Turing* labs) in the Universidad de los Andes. Figure 3 describes the network elements. There are 100 computers located in two rooms with different network segments: 65 machines are in one VLAN and 35 in the other one. Both network segments and the UnaCloud server are interconnected by a 10 Gbps backbone. All these machines have 1 Gbps connections to the switches.

Each file transfer protocol was tested using a customized version of UnaCloud that allows to execute command line tools to transfer the images. The only exception was the test for BitTorrent where we created an UnaCloud extension. The following is a description of the tests we made for each protocol.

UnaCloud TCP: We tested the same protocol currently used in the UnaCloud to establish the baseline for further comparisons. Basically, we took the existing implementation and removed the time limit to prevent errors by timeout.

FTP: We tested the FTP protocol by using a *FileZilla* server[4] in a Windows host and a web browser as the FTP client in the agents. The UnaCloud agents were modified to execute the FTP transfer without starting the corresponding virtual machine. We collect the transfer time by using the logs normally collected by the agents. Table 2 reports the results.

SMB: We tested the file transfer with SMB using the tools included in the Windows Operating System. We defined a shared disk in the server and used the command line tools to access the files. The UnaCloud agent executed commands such as NET USE and XCOPY to obtain the files from the server. The time was collected using the same logs mentioned above. Table 2 reports the results of our evaluation of SMB.

BitTorrent: We evaluated the BitTorrent protocol by using an extension to UnaCloud based on the Turn's BitTorrent library[5], a pure-Java library supporting both tracker and client roles. We had to install the extension into the UnaCloud server and agents to perform the tests. Table 2 reports the results.

4.2 Evaluation

We were able to perform all our tests without problems except for those copying an image to 100 desktops using FTP. In such protocol, the time spent by transferring the images grew exponentially and we got communication errors in more than 20 images each time. Although we configured the FileZilla FTP server allowing more than 100 connections and a timeout of 1 h, we were not able to deploy these machines without errors. Considering that we had results for deploying 1 to 75 machines, we decided to omit this test for the evaluation.

Table 2 summarizes the results of our tests[6]. It reports the average time (in seconds) to copy the 1.7 GB image file to each number of instances using each protocol. It also presents the ratio of the time spent by each protocol over the time spent by our custom TCP-based implementation.

[4] https://filezilla-project.org/.

[5] http://mpetazzoni.github.io/ttorrent/.

[6] Datasets available at https://goo.gl/NXJVZF.

Table 2. Transfer times obtained copying images using FTP, SMB and BitTorrent

# inst.	UnaCloud TCP (baseline) time	UnaCloud TCP (baseline) reliab.	FTP time	FTP ratio	FTP reliab.	SMB time	SMB ratio	SMB reliab.	BitTorrent time	BitTorrent ratio	BitTorrent reliab.
1	74,00	100%	35,33	47,7%	100%	34,00	45,9%	100%	227,00	306,8%	100%
5	92,47	100%	83,93	90,8%	100%	78,67	85,1%	100%	142,80	154,4%	100%
10	150,30	100%	134,00	89,2%	100%	149,27	99,3%	100%	105,66	70,3%	100%
25	359,07	100%	340,50	94,8%	96%	373,40	104,0%	100%	104,44	29,1%	100%
50	702,62	98%	715,20	101,8%	99%	703,35	100,1%	98%	128,04	18,2%	100%
75	1084,52	99%	1020,99	94,1%	96%	1034,99	95,4%	99%	184,79	17,0%	100%
100	2658,21	99%	Failure		76%	1382,39	52,0%	99%	229,01	8,6%	100%

time: average transfer time, in seconds, for each of the n copies of a 1.7 GB file.
ratio: transfer time using a protocol over the transfer time using UnaCloud TCP.
reliab. (reliability): number of correctly transferred files over the total requested.

Note that all the evaluated protocols perform better than our original implementation. SMB performs better than BitTorrent when the same image is copied to less than ten desktops, and FTP when the image is copied to less than five. BitTorrent, in contrast, performs better than all the others when the image is copied to ten or more machines.

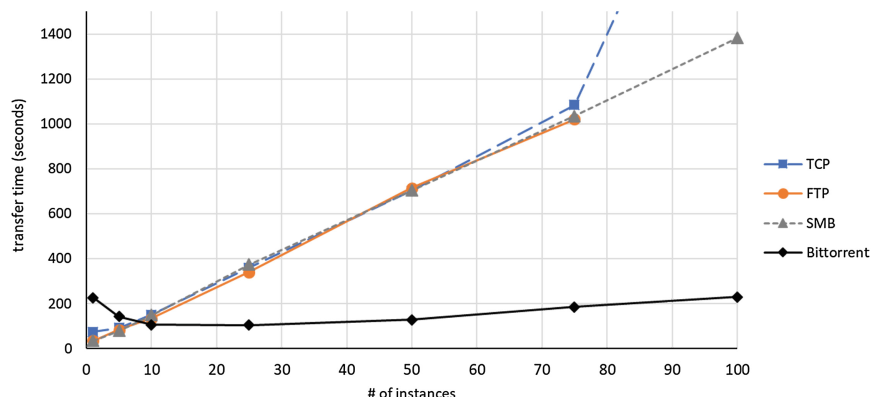


Fig. 4. Transmission time using the evaluated protocols

Figure 4 compares the transfer time of the evaluated protocols. Note that the time spent by the UnaCloud TCP, FTP and SMB protocols increases as the number of machines grows. In contrast, the time spent by BitTorrent is under 230 s between 5 and 100 machines. FTP, SMB and our original TCP-based protocol exceeds 200 s before 20 machines.

Table 3 compares the use of channel and the overhead during the transfer of the image using the evaluated protocols. This values were measured in the network interfaces of the UnaCloud File Server that transmitted the images during all the experiments. In Table 3, the overhead represents the amount of bytes

Table 3. Channel use and overhead copying images using FTP, SMB and BitTorrent

# inst.	UnaCloud TCP % channel	UnaCloud TCP % overhead	FTP % channel	FTP ratio	FTP % overhead	SMB % channel	SMB ratio	SMB % overhead	BitTorrent % channel	BitTorrent ratio	BitTorrent % overhead
1	19.60%	3.8%	21.7%	110.5%	2%	35.7%	182.1%	5.10%	17.9%	91.1%	77%
5	71.20%	19.3%	71.0%	99.8%	8%	71.2%	99.9%	1.40%	47.3%	66.4%	-31%
10	72.00%	59.7%	71.5%	99.3%	8%	72.0%	100.0%	1.39%	52.8%	73.3%	-59%
25	72.10%	53.2%	71.6%	99.3%	7%	72.1%	100.0%	1.39%	67.7%	93.8%	-81%
50	72.00%	51.4%	71.9%	99.9%	8%	72.0%	100.0%	1.39%	68.1%	94.5%	-85%
75	72.10%	51.0%	71.9%	99.7%	5%	72.1%	99.9%	1.39%	59.9%	83.1%	-87%
100	72.00%	49.9%	Failure			72.0%	100.0%	1.39%	60.0%	83.3%	-89%

% channel: peek of channel utilization.
ratio: % use of channel of a protocol over the % of channel using UnaCloud TCP.
% overhead: ratio of the overhead over the payload.

transmitted over the total size of the images transmitted. A larger overhead represents a larger number of re-transmitted data and control packages. Note that UnaCloud TCP, our custom implementation, exhibits the largest overhead. In contrast, note that BitTorrent has a negative overhead, i.e. the UnaCloud File server transmitted less than the total size of the images because part of the communications were performed among the desktops without server intervention. In addition, Note that use of the channel of the UnaCloud TCP, FTP and SMB protocols goes higher than 70% after 5 instances while the BitTorrent remains less than that value even for 100 instances. That means that BitTorrent produces less network congestion than the other protocols.

5 Evaluating the Deployment of Virtual Machines Using BitTorrent

Considering that BitTorrent outperformed the other protocols while transferring many copies of the same image, we performed further evaluations using this protocol to deploy the virtual machines in UnaCloud. This section compares the time spent deploying virtual machines using our custom UnaCloud TCP file transfer and an alternative BitTorrent implementation.

5.1 Test Suite

We performed tests comparing *the deployment time* ($T_{deployment}$) in an opportunistic desktop cloud running UnaCloud using our TCP-based protocol and using BitTorrent. We modified *UnaCloud* to ignore the predefined time for each state and avoid the related timeouts during deployment, and developed a new *UnaCloud P2P* that supports BitTorrent for the transmission of the virtual machine images.

To compare both UnaCloud implementations, we choose an image for a Debian 6.0 virtual machine with a 3.1 GB disk, 1 GB in RAM and a single core. Considering that UnaCloud transfers compressed images, the file to transmit is 1.7 GB. Note that the time required to deploy a virtual machine includes

the time for transferring and decompressing the file in addition to starting the virtualization.

We used that image to deploy 1, 5, 10, 25, 50, 75 and 100 virtual machines (a.k.a. instances). In such tests, the UnaCloud server was able to allocate each virtual machine in a different desktop and the image file was transferred to an equal number of different computers.

5.2 Evaluation

Table 4 summarizes the results of our tests. It reports the time spent to deploy multiple virtual machines using our TCP-based protocol and using BitTorrent.

Table 4. Deployment times using UnaCloud TCP and UnaCloud BitTorrent

#	**UnaCloud TCP (baseline)**		**UnaCloud BitTorrent (1VM x HW)**			
inst.	time	reliab.	deploy	time	ratio	reliab.
1	330,33	100%	1x1	476,33	144,2%	100%
5	340,53	100%	1x5	389,00	114,2%	100%
10	394,30	100%	1x10	351,52	89,2%	100%
25	605,61	100%	1x25	351,64	58,1%	100%
50	959,73	98%	1x50	381,63	39,8%	100%
75	1331,16	99%	1x75	433,19	32,5%	100%
100	2908,37	99%	1x100	489,21	16,8%	100%

time: time for deploying all the n virtual machines, in seconds.
deploy: number of virtual machines per desktop.
ratio: deployment time using UnaCloud BitTorrent over the time of UnaCloud TCP.
reliab. (reliability): number of correctly deployed instances over the total.

Note that the UnaCloud BitTorrent extension outperforms the existing implementation for deploying multiple virtual machines. While the deployment time of the existing implementation increases as the number of instances grows, the time spent by our BitTorrent-based implementation remains almost constant (410 s aprox. after 5 machines).

6 Comparison to Other Cloud Platforms

Based on our results, we performed additional tests to compare the time spent deploying virtual machines on UnaCloud using BitTorrent to the required in other existing public and private cloud platforms. Basically, these tests aim to determine if the resulting time to deploy is acceptable and competitive with respect to other existing platforms. This section presents the results of our tests.

6.1 Test Suite

We performed the test suite described in Sect. 5, deploying the same virtual machine image into multiple instances on (1) a *public cloud*, the *Amazon Elastic Computing (EC2)* service[7], and (2) two *private clouds*, one running OpenStack[8] and another running VMware vCloud Suite[9].

For each platform, we created the corresponding image and deployed it into 1, 25, 50, 75 and 100 instances. Each test was repeated three times. The deployment time was measured using the tools provided by each platform.

Amazon Elastic Compute Cloud (EC2): To deploy the virtual machines, we first created an Amazon Machine Image (AMI) with the same specifications of the previous tests. An AMI can be created from scratch using the command-line *EC2 API Tools* or by modifying an existing AMI in the web. We created the AMI from the image we used in UnaCloud and started multiple instances from it. Because we were using an academic license for EC2, we were not allowed to start 100 instances without requesting first an authorization. The results of our tests are presented in Table 5.

An OpenStack-based private cloud: In addition, we executed the tests on *Caldas I*, an existing OpenStack-based private cloud in the CECAD, the High Performance Computing Center of the Universidad Distrital Francisco Jose de Caldas[10]. To do it, we created first a QCOW2 image for the virtual machine using a VirtualBox's VDI hard disk image with the same specifications we used in the previous tests. The resulting image had a size of 3.05 GB. We used that image to start the multiple instances. Table 5 presents the results of our tests.

A VMWare vCloud-based private cloud: Finally, we executed the tests on *ISCloud*, an existing VMWare vCloud-based private cloud in the Universidad de los Andes. We created an image with the same specification of the used in the other tests. The resulting image of 3.0 GB was uploaded to the system and used to deploy the virtual machines using two types of storage, one based on hybrid HDD/SSD and other on HDD drives. Table 5 reports the average time spent by deploying these instances.

6.2 Comparison

Table 5 reports the results of our tests. It compares the total time spent by our Bit-Torrent extension to UnaCloud to the time reported by Amazon EC2 and the Caldas I and ISCloud private clouds.

Note that the time exhibited by Amazon EC2 remains almost constant independently of the number of instances requested. Although the behaviour of our BitTorrent extension is similar and the time remains almost constant, the time

[7] https://aws.amazon.com/ec2.
[8] https://www.openstack.org/.
[9] https://www.vmware.com/products/vcloud-suite.html.
[10] http://cecad.udistrital.edu.co/.

Table 5. Deployment times using UnaCloud P2P and other public and private clouds

# ins.	UnaCloud BitTorrent total time	Amazon EC2 total time	ratio	Caldas I (OpenStack) total time	ratio	ISCloud (vCloud) total time	ratio
1	476,33	250,00	52,5%	186,00	39,0%	70,50	14,8%
25	351,64	248,00	70,5%	982,00	279,3%	538,00	153,0%
50	381,63	241,00	63,2%	1737,00	455,2%	1074,50	281,6%
75	433,19	265,00	61,2%	2421,00	558,9%	1678,00	387,4%
100	489,21	251,00	51,3%	3402,00	695,4%	2200,00	449,7%

time: deploying time for all the n instances, given in seconds.
ratio: deploying time on a platform over the deployment time on the UnaCloud BitTorrent.

exhibited by Amazon EC2 is a few more than a half. While the time in Amazon EC2 is approximately 250 s, the time of UnaCloud P2P is close to 410 s.

UnaCloud P2P exhibits a better deployment time than the Caldas I. Although Caldas I is widely used, it is optimized to support high loads of computing power but not to deploy a large number of instances simultaneously. This OpenStack-based private cloud offers better performance for deploying a single virtual machine (186 s vs. 410 s on UnaCloud), but lacks behind for deploying 25 or more instances.

After some preliminary tests on the VMWare vCloud-based ISCloud, we created an image using a suspended virtual machine to optimize the time for deploying the instances. However, in order to reduce the impact on the other virtual machines running on the platform, the deployments were enqueued and performed four machines at a time. The deployment times result comparable to the obtained in the Caldas I private cloud. Note that it deploys a single machine in less time than UnaCloud P2P but takes more time for 25 or more instances.

7 Discussion

Related Work. Our research confirms other works comparing BitTorrent to other protocols for provisioning data and virtual machines. BitTorrent has many features that reduce the transfer time when a single file is transmitted to multiple machines [12]. Commonly used to share music and movies, it has been recently used to support provisioning for Grid and Cloud computing [1,4,7]. In contrast to these works focused on traditional cloud platforms [5,6], our research is focused on applying BitTorrent for deploying virtual machines in desktop clouds. In addition, we have performed some comparisons to existing cloud platforms that showed that it may be applied to improve not only opportunistic platforms but also private cloud platforms.

Deployment time in private clouds. Our findings show that public clouds as Amazon EC2 offer better deployment times that the exhibited in the evaluated private clouds[11]. There are many reasons for these differences. On the one hand, while Amazon EC2 is surely designed to support peaks where hundreds of new instances are required, the private clouds are often designed to introduce few instances at a time. We evaluated two academic private-clouds that support scientific research and class-related activities. The instances can be deployed at the beginning of a period without a rush. On the other hand, the private cloud we reviewed use the storage for the virtual machine images also for other applications, e.g. to store the virtual disks used by the instances and the databases used by the scientific software. In addition, all the instances runs virtualized on the same bare-metal. This may cause disk and CPU contention at lower that may not occurs in public cloud with a larger number of computer nodes.

8 Conclusions

The protocol used to transfer the virtual machine images may affect the time for deploying instances in IaaS platforms, specially when a large number of instances is requested or the image files are large. In UnaCloud the time spent transferring the images was the main cause of delays and errors during deployment.

We evaluated several protocols, namely FTP, SMB and BitTorrent, to replace the currently used in UnaCloud. According to our tests, SMB is the more performant when one to five machines were requested. BitTorrent offers the best performance when 10 or more instances are requested. Furthermore, we were able to transfer 50 images using it and the transfer time remains almost constant after 5 (410 s in average for an image file of 1.7 GB).

We created an UnaCloud extension to deploy virtual machines using BitTorrent. It not only outperforms the current implementation but also exhibits deployment times comparable to the achieved in public and private clouds. The times were compared with the offered by Amazon EC2 and other two private clouds. Amazon EC2 deploys multiple instances in an almost constant time, just like our extension, but its time is a few more than a half (250 vs 410 s approx.). Caldas I, an OpenStack-based private cloud, exhibits better deployment time when a single instance is requested but lacks behind when 5 or more instances are requested. ISCloud, a VMWare vCloud-based cloud, exhibits a similar behaviour than the exposed by the Caldas I.

Based on our findings, we are planning some new enhancements and evaluations. First, we have detected a bottleneck when multiple virtual machines are deployed in a single desktop. Apparently, the time spent by cloning the virtual machine in the desktop is almost a half of the time spent by copying it. We are considering a modification to the UnaCloud agents to use the same stream of BitTorrent to create multiple copies in the machine that will run them. In addition, we are interested in further evaluations of the impact of the protocols for transferring the images on the work that end-users perform on the desktops that

[11] More information of the evaluated private clouds at https://goo.gl/NXJVZF.

run the virtual machines. We want to confirm if the use of BitTorrent reduces the network congestion as perceived by these users. Finally, our comparisons with other cloud platforms showed some differences in the way which them manage and transfer the images. We are interested on exploring techniques to improve the deployment time on private clouds using software such as OpenStack.

Acknowledgments. This research was performed by the Center of Excellence and Appropriation in Big Data and Data Analytics (CAOBA), financed by the Ministerio de Tecnologías de la Información y Telecomunicaciones de la República de Colombia (MinTIC) through the Departamento Administrativo de Ciencia, Tecnología e Innovación (COLCIENCIAS), contract N° FP44842-anexo46-2015.

References

1. Babaoglu, O., Marzolla, M., Tamburini, M.: Design and implementation of a P2P cloud system. In: 27th Annual ACM Symposium on Applied Computing (SAC 2012), pp. 412–417. ACM (2012)
2. Bhushan, A.: A File Transfer Protocol. RFC 114. IETF Network Working Group (1971). https://tools.ietf.org/html/rfc114
3. Cohen, B.: The BitTorrent Protocol Specification (2008). http://www.bittorrent.org/beps/bep_0003.html
4. Costa, F., Silva, L., Fedak, G., Kelley, I.: Optimizing the data distribution layer of BOINC with BitTorrent. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8, April 2008
5. Laurikainen, R., Laitinen, J., Lehtovuori, P., Nurminen, J.K.: Improving the efficiency of deploying virtual machines in a cloud environment. In: 2012 International Conference on Cloud and Service Computing, pp. 232–239, November 2012
6. Lopez-Garcia, A., del Castillo, E.F.: Efficient image deployment in cloud environments. J. Netw. Comput. Appl. **63**, 140–149 (2016)
7. Mantoro, T., Ali, H.S.: BitTorrent: extra-locality P2P approach for grid content distribution networks. In: 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2009), pp. 406–411. ACM (2009)
8. Microsoft: Microsoft SMB Protocol and CIFS Protocol Overview (2017). https://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx. Accessed June 2017
9. NetBIOS Working Group: Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods. RFC 1001. IETF (1987). http://ubiqx.org/cifs/rfc-draft/rfc1001.html
10. Postel, J., Reynolds, J.: File Transfer Protocol (FTP). RFC 959. IETF (1971). https://tools.ietf.org/html/rfc114
11. Rosales, E., Castro, H., Villamizar, M.: UnaCloud: opportunistic cloud computing infrastructure as a service. In: 2nd International Conference on Cloud Computing GRIDs and Virtualization (CLOUD COMPUTING 2011), pp. 187–194 (2011)
12. Sharma, P., Bhakuni, A., Kaushal, R.: Performance analysis of BitTorrent protocol. In: 2013 National Conference on Communications (NCC), pp. 1–5 (2013)

Distributed Cosmic Ray Detection Using Cloud Computing

Germán Schnyder(✉), Sergio Nesmachnow, and Gonzalo Tancredi

Universidad de la República, Montevideo, Uruguay
{german.schnyder,sergion}@fing.edu.uy, gonzalo@fisica.edu.uy

Abstract. This article presents a distributed computing approach to detect cosmic rays in images taken by the Hubble Space Telescope (HST). A cloud computing implementation is developed to improve the overall processing time for the available images dataset (15 TB), containing dark images from several HST instruments. A specific architecture is presented where images are stored in a replicated and highly available storage system. Image processing is performed on virtual machines from the Azure Batch framework using a developed Python application. The experimental evaluation shows that the architecture accomplished the purpose of processing the complete dataset based on scaling computing resources in terms of processing nodes. Speedup improved in a factor of 6.57× over a previous implementation using Apache Mesos. The overall computation took 10 days to complete and results are stored on a non-relational database available to astronomers and researchers.

Keywords: Cosmic rays · HST · Azure · Python

1 Introduction

Cosmic rays are energy-charged particles whose origin can be associated with solar storms and radiation from the confines of the universe [2]. Studying cosmic rays allows understanding several phenomena, such as winds and solar eruptions. Cosmic rays are a problem for scientific instruments used in astronomy, especially for those sent to outer space. However, this problem can also be beneficial. Instruments sent to outer space are able to provide information about the impact of cosmic rays to estimate the magnetic field. Hubble Space Telescope (HST) is a remarkable device appropriate for this task. HST has several instruments on board and orbits the planet on regions for which cosmic rays information is not available. In this way, images captured periodically by HST instruments can determine the impact of the cosmic rays in the Earth's atmosphere [10].

HST dark frames are suitable for cosmic ray studies because they are acquired with closed shutters. Thus only cosmic ray events are recorded. Cosmic ray flux information from HST instruments can be used to compare the magnetic field strength, the gamma ray flux, and other geophysical data measured by geomagnetic observatories in the Earth. HST results certainly complement that of the existing cosmic rays detectors, contributing to understand those phenomena.

© Springer International Publishing AG 2018
E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 414–429, 2018.
https://doi.org/10.1007/978-3-319-73353-1_29

This work proposes studying the set of dark images from the entire period of HST activity to study the cosmic rays traces that impacted on their instruments [11]. The volume of data corresponding to this set of images is around 15 TB, so High Performance Computing comes to help in the information processing. Considering the potential of information available through the HST darks, the scientific contribution of the approach is high. An enormous data source is available with a very important characteristic: it extends over time, allowing analysis of the fluctuation of the density of the magnetosphere.

A distributed computing approach is presented for executing in the Microsoft Azure cloud platform. A specific architecture is proposed for image processing. Efficiency analysis is performed using real HST data over virtual machines from the Microsoft Azure cloud infrastructure. Accurate efficiency results are reported for both data storage and processing, improving over a previously proposed implementation using Apache Mesos in a locally distributed infrastructure. Results of the processing are made available in a data structure designed to be consulted by researchers. Two tables are built to store specific information on each processed image and on each cosmic ray that impacted each image. The design of tables takes into account different options for users interaction (e.g., consulting results by instrument, by image, by date range, and other attributes).

The article is organized as follows. Next section describes the approach for cosmic rays detection. The proposed distributed implementation on Microsoft Azure is described in Sect. 3. Section 4 describes how images were acquired and manipulated in the analysis. Section 5 reports the experimental evaluation of the proposed system, focused on the efficiency results. Finally, Sect. 6 presents the conclusions and formulates the main lines for future work.

2 Cosmic Ray Detection

This section introduces cosmic rays and describes instruments on HST and specific software for image processing.

2.1 Cosmic Rays

Cosmic rays are energetic particles that travel through the universe [3]. Cosmic radiation affects the operation of electronic devices in a testable and measurable way. In particular, HST instruments are exposed to this phenomenon and therefore they are a very rich source of detection of low energy cosmic rays, since they are not under the protection of the Earth's atmosphere.

Figure 1 shows the impact of energy particles in form of cosmic rays on the lens of the Advanced Camera for Surveys, from HST. The affected pixels in the image reveal important information about the cosmic rays that hit. Using specific image formats, such as Flexible Image Transport System (FITS), and applying processing software, such as Image Reduction and Analysis Facility (IRAF), the main features of impacting cosmic rays can be extracted. The information gathered in the image processing determines the geographic location of cosmic ray impacts, allowing to build a map of cosmic rays incidence on Earth.

Fig. 1. Dark image with cosmic rays impacts (j6mf16lhq_raw.fits)

2.2 HST Instruments

HST includes camera-type instruments, spectrographs, and various types of sensors. This work uses images of those instruments that are capable of recording the impact of cosmic rays. HST instruments operate synchronously; observations of different instruments are performed simultaneously and on the same beam of light reflected by the main mirror. Each instrument analyzes a range of the specific light spectrum.

Instruments that captured the images analyzed in this work include:

- *Advanced Camera for Surveys* (ACS), an instrument composed of three cameras: wide-field camera (WFC), high resolution camera (HRC), and solar blind channel (SBC). Following a flaw in January 2007, the WFC and HRC ceased operations for approximately two years. During the fourth HST maintenance mission the WFC was recovered but HRC was not, so it is not available for use thereafter. Images captured by ACS have a size of 2048×4096 pixels. The camera exposure data is in file with 'flt' extension and the positioning data of the instrument is in file with 'spt' extension.
- *Space Telescope Imaging Spectrograph* (STIS), a spectrograph containing a camera. Installed in 1997, operated until 2004, was repaired in 2009, and continues in use to date. STIS analyzes the ultraviolet spectrum and performs important tasks, such as the analysis of atmosphere on extrasolar planets and the search for black holes. STIS has three detectors that generate images of 1024×1024 pixels, stored in a 'ftl' file, and positioning data is in a 'spt' file.
- *Cosmic Origins Spectrograph* (COS), an ultraviolet spectrograph installed in 2009 to complement STIS, designed to observe bright spots like stars and quasars. COS has two channels to capture waves: far ultraviolet (FUV) and near ultraviolet (NUV). COS images have 16384×1024 pixels and come in

two separate files with extensions ‘flt_a’ and ‘flt_b’ for FUV and a single ‘flt’ file for NUV. The positioning data of the instrument is in a ‘spt’ file.
- *Near Infrared Camera and Multi-Object Spectrometer* (NICMOS), an instrument that captures infrared images and spectroscopic observation of astronomical objects. It operated in two periods: from 1997 to 1999, and from 2002 to 2008. NICMOS images have 256 × 256 pixels and are consolidated into a single image generated by an internal pipeline.
- *Wide Field Planetary Camera 2* (WFPC2), an instrument for image capture that was installed in the HST in a period of about 26 years. It was supplanted by WFC3. WPFC2 took the most famous photos of the HST, such as Pillars of Creation and Hubble Deep Field. It consisted of three wide-field (WFC) sensors and a high-resolution camera (PC). A typical WFPC2 image has four components of 800 × 800 pixels, due to the 3 WFCs and the PC. The latter usually appears smaller in images because of its smaller visual field.
- *Wide Field Camera 3* (WFC3), an instrument that captures images in infrared (IR) and ultraviolet (UVIS), installed in 2009 to replace WFPC2. A channel selector in the observation parameters indicates the detector to use (IR/UVIS). WFC3 images have 2051 × 4096 pixels. The camera exposure data is in the ‘flt’ file and the positioning data is in the ‘spt’ file.

Table 1 summarizes the details of each instrument, namely: the extensions for raw data and instrument positioning data, the image size in pixels (width × height); and the period in which data was made available (it represents a subset of the total period of operation of each instrument).

Table 1. Summary of data from HST instruments

Instrument	Files		Image size (pixels)	Operation period
	Data	Position		
ACS	flt	spt	2048 × 4096	2002–2014
COS	flt_a, flt_b	spt	16384 × 1024	2009–2017
NICMOS	raw, mos	spt	256 × 256	1997–2009
STIS	flt	spt	1024 × 1024	1997–2017
WFC3	flt	spt	2051 × 4096	2009–2014
WFPC2	c0m	shm	800 × 800	1993–2009

2.3 The FITS Format and the IRAF and Astropy Processing Software

FITS. FITS is a universal format defined for exchange of astronomical images [7]. Each FITS image is made up of one or more Header Data Units (HDUs). Each HDU specifies a special type of information. The first part of an HDU declares entries of type <key, value> that determine the structure of the binary content that follows in the FITS file. These entries contain diverse information, from

the configuration of the instrument used to capture the image to data of the investigator who requested the capture. A FITS image with a single HDU is called 'Basic FITS' and constitutes the minimum expression of the format. From this basic form, extensions can be added to enrich the information related to the captured image. FITS allows each instrument to establish how the different format variants are used and what type of headers and extensions are included as a result of an observation. Some types of files are used by all instruments to preserve information regarding the spatial positioning of the instrument, the type of observation, and the manipulations that the dataset suffered when processed with specific software. Support, Planning and Telemetry (SPT) files contain the information of the researcher who requested the original observation and the proposed observation (dates, telemetry, location, etc.). Trailer (TRL) files contain information about the type of processing performed on the dataset and are used as system output for internal tuning and analysis algorithms.

IRAF. Specific software is needed to process FITS images. IRAF is a general purpose software designed by developers from National Optical Astronomical Observatory to process images and astronomical data sets [13]. IRAF allows building scripts and executable files using the IRAF Common Language (IRAF CL) development and execution environment. The proof of concept from which this work starts is based on IRAF CL, which interacts with the Linux file system and a set of images, in order to obtain relevant information about the cosmic rays (see Algorithm 1). First, images are searched in the current directory and discard filters are applied (e.g., to assure the image is dark). Finally, xzap tasks of the dimsum package are applied to clean the cosmic rays. Once the clean image is obtained, it is subtracted from the original. Thus an image composed only of cosmic rays is obtained. Several parameters can be used to configure a xzap task for image analysis using IRAF. Parameters indicate different aspects of the algorithm, such as the median value of the filters, the number of pixels to be used as buffer around the image and the cosmic rays, the maximum number of iterations to be executed, etc. Table 2 shows the parameters used to configure the xzap task on the executions.

Algorithm 1. Obtaining cosmic rays using xzap

```
begin
    foreach image in the current dir do
        if image IS DARK and CHINJECT != NONE and FLASHCUR !=
        LOW then
            headers := getHeaders(image);
            cleanImage := dimsum.xzap(image, ... );
            crs := image - cleanImage;
    end
end
```

Table 2. Parameters used on xzap task configuration

Parameter	Value	Description
zmin	2000	Minimum data value for fmedian filter
zmax	4000	Maximum data value for fmedian filter
zboxsz	10	Box size for fmedian filter
skyfiltsize	15	Median filter size for local sky evaluation
subsample	0	Block averaging factor before median filtering
ngrowobj	3	Number of pixels to flag as buffer around objects
nrings	0	Number of pixels to flag around CRs
nsigneg	0	Number of sky sigma for negative zapping
nsigobj	1.5	Number of sky sigma for object identification
nsigrej	2.0	The n-sigma sky rejection parameter
maxiter	20	The maximum number of iterations
statsec	“”	Image section to use for computing sky sigma
nsigzap	1	Zapping threshold in number of sky sigma

Astropy. Astropy is a package of common features and tools for astronomical research and astrophysics in python [12]. This work uses Astropy for the manipulation of FITS files. IRAF CL allows working with FITS files in the context of IRAF, but in the last years scientists used general purpose programming languages more often. Special packages have been developed to solve common issues in scientific research, such as Astropy. Astropy contains several utilities to process files with scientific formats, among them astropy.io.fits. This utility allows reading files that comply with FITS format as well as load extensions and file headers as native python data types. Using the data representation in Astropy makes easier to construct abstractions for manipulating files from HST instruments, since there are classes that allow to map HDUs, extensions, and headers. In addition, astropy.io.fits contains utilities that facilitate the implementation of recurring tasks such as value parsing, characters removal, that appear frequently in FITS files. Regarding the spatial positioning of HST instruments, Astropy contains utilities for calculating coordinates in different reference systems through the astropy.utils.iers package [1].

3 Distributed Cosmic Rays Detection in Microsoft Azure

The basic procedure for detecting the cosmic rays impacts on HST images consist of extracting the image noise and then distinguish the cosmic ray trace in the noise. As described in Sect. 2.3, IRAF includes tools for manipulating standard FITS images, such as the ones published by HST and other instruments from Space Telescope Science Institute (STScI). The first stage, described in Algorithm 1, allows obtaining a mask with the pixels identified as noise. The second

stage consists on applying a basic algorithm for finding connected components on the image and establish if it is a cosmic ray impact. From the trace of the cosmic ray impact it is possible to estimate how strong the incident ray was. This impact information is added up to the instrument positioning to constitute the core of the statistical information to be gathered. For example, for every image, several features are obtained, including: the mean cosmic ray flux, the mean cosmic ray length, the n-th percentile of flux and length, etc.

This article reports our implementation for distributed cosmic rays detection using Microsoft Azure. This platform was selected because of an economic sponsorship that was obtained from Microsoft based on a preview of our work sent to them.

There are several aspects of the desired architecture that must take into account specific usage requisites. For example, images need to be replicated in order to allow the processing nodes to access the information at any time and from anywhere. Also, bottlenecks need to be avoided, so the tasks can perform immediately after started and hardware resources are available. Finally, the results of the processing has to be available on a partitioned storage to the queries can be optimized by date and by instrument name.

Therefore, the architecture must comprise four fundamental aspects that can be mapped to specific technologies on Microsoft Azure platform:

1. *Replicated data storage for images*, mapped to Azure Storage Blobs.
2. *Scheduling*, mapped to Azure Batch.
3. *Computing nodes*, mapped to Azure Virtual Machines.
4. *Partitioned and replicated results*, mapped to Azure Storage Tables.

A diagram of the basic operations of Microsoft Batch components is presented in Fig. 2. Client application uploads the bootstrap scripts and the task binaries to the storage account. After these initial steps, the client application triggers the jobs to run the processing tasks in virtual servers. The output from the processing tasks is also stored on the storage account.

Regarding the programming language, the choice must consider both code maintenance and extensibility. Also, IRAF Common Language is not suitable for binaries distribution to the computing nodes, so it must be replaced. The first alternative explored was rewriting any existing code, and adding all the new code, using Python. This programming language is widely used on the scientific community in research projects that require intensive processing and also is the one chosen by JPL itself for building the astronomic tools they distribute. Having the code in Python presents two advantages: its understandable for most of the programmers that may intend to improve, extend or either fix it, and its immediately compatible with the whole IRAF analysis tools.

The developed software holds two main responsibilities. First, it computes the cosmic rays incidence over the assigned image on each computing node. Second, it calculates the exact instrument position at the time the image was captured, in terms of terrestrial coordinates. The model built is presented on Fig. 3.

For each step in the algorithm a specific module was designed plus other custom libraries. There is no need to follow a specific interface or architecture in

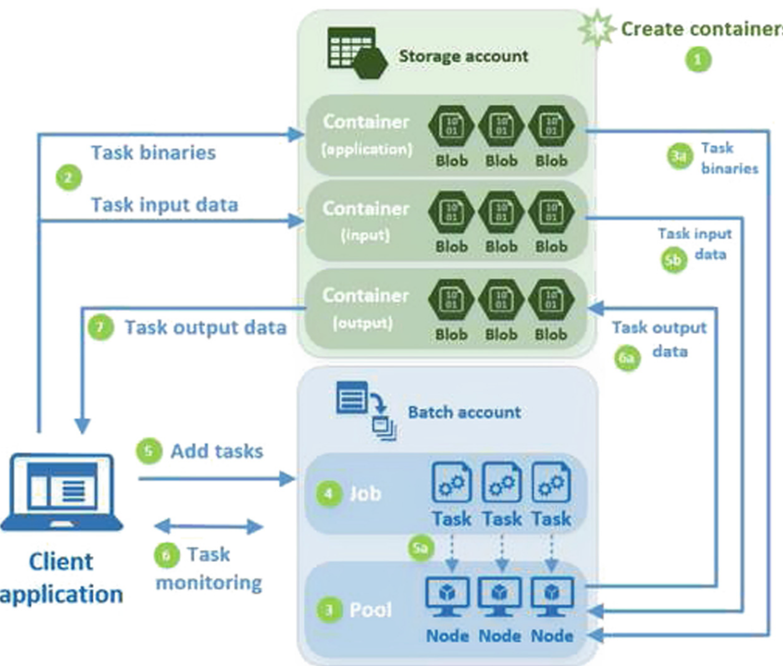


Fig. 2. Primary operations performed by the client and task scripts on Azure Batch platform (public domain image, taken from [5])

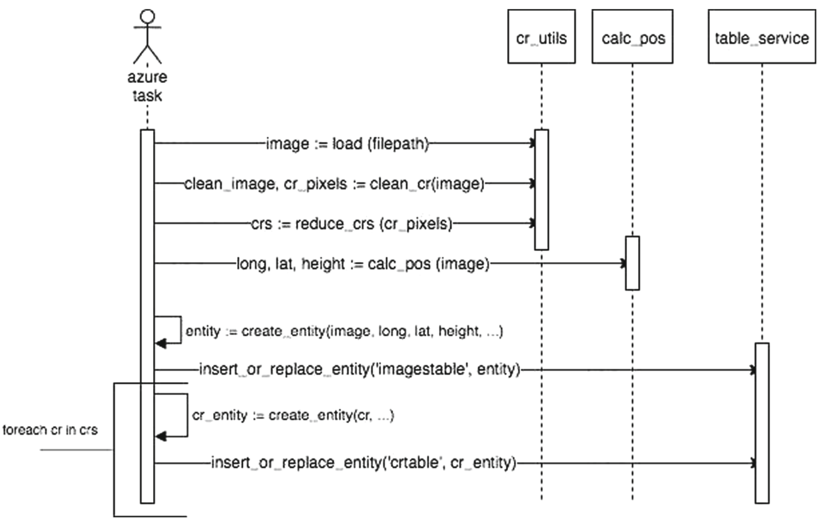


Fig. 3. Sequence diagram for the application executed on Azure

order to deploy an application in Azure Batch processing nodes but as usual the code needs to be maintainable and extensible. Also, Azure Batch provides access control signatures to blobs and databases for a specific period of time. Here the list of classes and utilities developed for interacting with Azure platform in order to accomplish the astronomical images processing task:

– `image.py`. This class represents IRAF format and provides utilities for accessing individual headers and extensions. It works as a wrapper over astropy.iraf libraries.
– `instruments.py`. This class represents the HST instruments considering each instrument particularities (e.g. file extensions containing the data, file extensions containing the positioning information, image size, etc.).
– `cosmics.py`. This class implements the LA Cosmic algorithm for cosmic ray detection (created by Malte Tewes, credited in the source file).
– `crutils.py`. This class provides methods to load images from a file into a custom model (image.py), to apply LA Cosmic algorithm (cosmics.py) and to compute metrics over it.
– `calc_pos.py`. This class is a direct translation from the class written in CL (present on the original proof of concept). It provides utilities to calculate the exact position of the image in terms of the terrestrial atmosphere (based on the instrument positioning information preserved in the image).
– `tests`. These classes provides tests and validations over the complete set of features developed. It includes several testing cases for the different instruments, from loading images to computing the cosmic rays and terrestrial coordinates for every kind of image.
– `azure_task.py`. This class contains the code for instantiate the complete pipeline in an Azure Batch processing node. It contains all the invocations for the different methods and the code for persist the results on the result tables.
– `azure_client.py`. This class works as an orchestrator for the whole process. It manages the echosystem lifecycle though managing the instance pool and the jobs scheduling. It also manages every job lifecycle considering timeouts and objects expirations.

4 Data Manipulation

Images from HST instruments were obtained from STScI through a specific data access request with research purpose. In general, all images are available to download from the STScI web platform, but the size of the complete dataset (15 TB) demanded applying a different approach. As a result, we obtained three HDD containing the images, which were connected to Cluster FING, the High Performance Computing platform from Universidad de la República [6], in order to start the evaluation of the proposed solution. First tests were run over small datasets until the code was verified. Then, the content of the three HDD was uploaded to the Azure cloud platform. In particular, we choose the Azure Blob storage because it allowed storing all the images with guarantee of replication

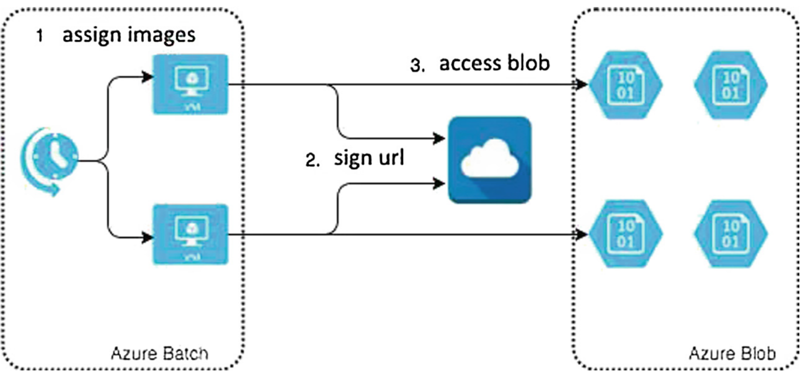


Fig. 4. Accessing the Blob storage content

and high availability. Therefore, all the processing nodes would be able to access the images avoiding any kind of bottleneck or hardware failures. Each processing node has to request access for the image that will be processed to the security authority, so it gets a time based token. After getting the token the node can download the image and run the code. This interaction between the processing nodes and storage blobs is shown in Fig. 4.

We also needed to consider how to store the analysis results taking into account two aspects: partitioning and availability. In the original proposal, and in the first steps of this work, an output file was used for presenting the results. This file strategy is not suitable for outputting the complete dataset results because the amount of files becomes hard to track and we would need to reprocess them for every query. Thus, we choose to use database tables to persist the output from the processing and take advantage of database managers optimization's (such us indexing). Future researchers may need to query by specific instruments or specific dates and therefore the data partitioning strategy became key on the data architecture side. An example of how the files are processed is shown in Fig. 5.

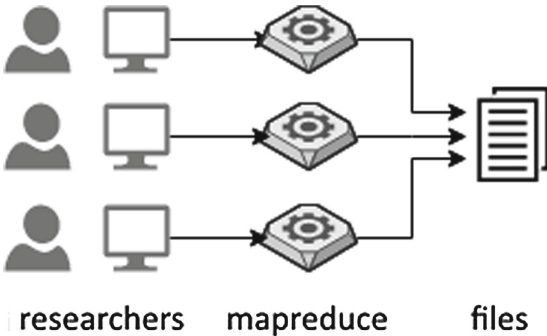


Fig. 5. Example architecture for batch processing the output files

We decided to use a database approach because the main goal of the research is obtaining the data for providing astronomers and researchers with quality information. Each processed image is analyzed and its details are stored as a new row in the table *images*. As a result, this table contains the positional information as well as basic cosmic rays statistics (as cosmic ray count). On a second table named *cosmicrays*, the specific details of each cosmic ray impact are preserved. These tables were designed following a non-relational approach, where no primary keys are defined. The rationale behind this decision is that no joins are intended and only partitioning and range keys are needed. Partition key (PK) enables the possibility of optimizing the queries based on instrument name (since this attribute was chosen as PK). Row keys (RK) determine how to order the rows inside the server who stores that particular partition of the table. As a consequence, it optimizes the date filter (since date of the image was chosen as the RK).

Figure 6 shows the basic structure of both tables used to store the data. As it was mentioned, the PK determines in which server of Azure Table that row will be stored. This condition enforces the data locality principle, where its reasonable to suppose that similar rows will be queried together. Also, the RK is the one that determines the row ordering inside the same Azure Table server. The combination of PK and RK optimizes the searches intended for the future research over this work results. Additionally, Azure Table includes a secondary RK known as timestamp that let us know when that row was created (especially interesting for obtaining the image processing metrics) [4].

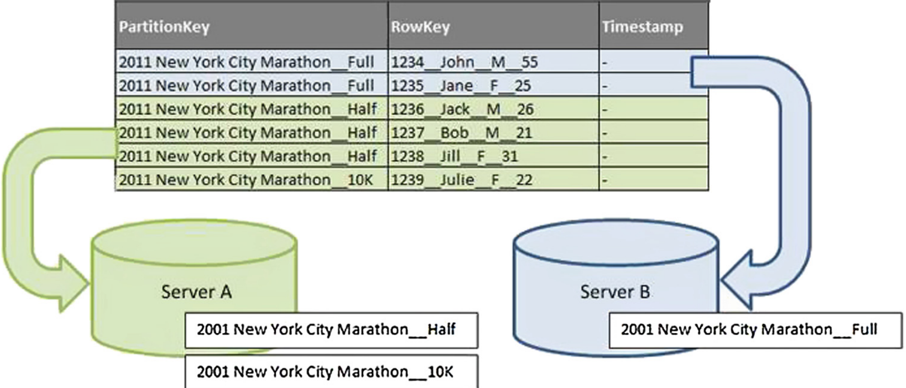
	images
PK	**instrument**
RK	**image_name**
RK	**timestamp**
	cr_count
	width
	height
	exposition_duration
	longitude
	latitude
	< other position parameters >

	cosmicrays
PK	**image_name**
RK	**cr_idx**
RK	**timestamp**
	area
	orientation
	< other math attributes >

Fig. 6. Tables used to store the images processing results

Figure 7 shows how different rows are distributed among servers. Rows with the same PK are stored in the same server. This allow the researchers to query by instrument name and specific time range. This data architecture strategy

and the related application design based on cloud computing establishes the foundations for similar processing pipelines (e.g. regular observations instead of darks).



PartitionKey	RowKey	Timestamp
2011 New York City Marathon__Full	1234__John__M__55	-
2011 New York City Marathon__Full	1235__Jane__F__25	-
2011 New York City Marathon__Half	1236__Jack__M__26	-
2011 New York City Marathon__Half	1237__Bob__M__21	-
2011 New York City Marathon__Half	1238__Jill__F__31	-
2011 New York City Marathon__10K	1239__Julie__F__22	-

Fig. 7. Azure Table partitioning schema (public domain image, taken from [4])

The proposed architecture that enables the complete image dataset processing is shown on Fig. 8.

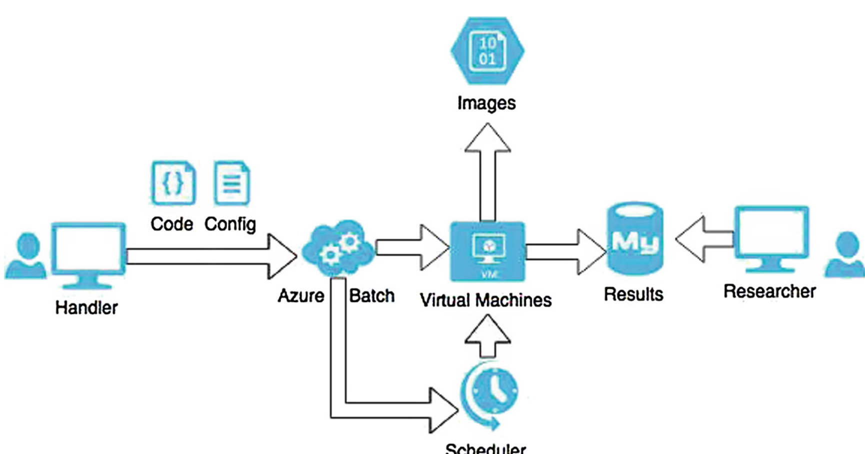


Fig. 8. Proposed architecture

5 Experimental Evaluation

For executing the application on Microsoft Azure platform, the whole available computing capacity was used. According to the sponsorship subscription terms the maximum amount of cores running at the same time was 20 [8]. For other resources (e.g. disk usage, network traffic, etc.) the cap was established in terms of costs. Since the cheapest suitable machines were double core, the total virtual servers to run the application was 10. Each server had Ubuntu 14.04 already installed, with an Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40 GHz processor and 3 GB of RAM. As part of the application bootstrapping several packages were installed (such as Python v3, Astropy, etc.). The execution times were obtained directly from the Microsoft Azure portal statistics.

Table 3 presents the execution results by different time periods. Some observations can be made from the results. For example, performance varies depending on the time period being considered. These variations may respond to resources availability in the cloud platform. This is a common issue since the same hardware is being requested and used by different users on this kind of shared platform. Anyway, an overall throughput improvement is observed as the times period being considered increases. This improvement has direct effect on the overall time improvement for the processing. On a 5 days period the average of tasks executed by minute is 15.12.

Table 3. Tasks execution metrics in Azure Batch

Time period (min)	Completed tasks	Tasks/min	Tasks mean time (min)
60	772	12.66	0.078
1440	18540	12.87	0.078
7200	82960	11.52	0.087
14400	217690	15.12	0.066

Table 4 compares the execution times of the Azure implementation with those from the implementation previously developed over the Mesos architecture [9]. Since the statistics querying on Azure Portal is based on time periods, we needed to extrapolate the results to get some comparable results. The comparison is made between the combined cosmic ray detection scheduling algorithm (CCRD) variant (since it was the algorithm that performed the best over the Mesos architecture) and the mean flow time for the 14400 min period (corresponding to the Azure best execution scenario).

From the results reported in Table 4, we conclude that the Azure execution improved substantially the numbers obtained on Mesos platform. Speaking in numbers, a speedup of 6.57 was achieved on the 1000 images dataset. Since the number of nodes used is 10 this means an efficiency of 0.66. For the datasets of 100 and 500 images the speedups obtained are 0.86 and 0.84, respectively. This means an efficiency of ∼0.85 in both cases. Despite the architectures comparison

Table 4. Flow time and efficiency for execution in Azure platform compared to CCRD on different images datasets

# images	Execution time (s)		Acceleration	Speedup	Efficiency
	CCRD	Azure			
100	533	396.89	1.343	5.57	0.56
500	1710	1984.47	0.86	5.51	0.55
1000	3326	3968.95	0.84	6.57	0.66

is not fair based on the different hardware being used, it can be outlined that the results on Azure platform are remarkable in terms of performance. On Azure platform it is possible to assign the needed resources for processing the whole dataset in times considerably smaller than the ones estimated on the Mesos platform.

Table 5 reports the mean processing time for each instrument as well as the mean number of cosmic rays found. A direct relationship is perceived between the image size and the time needed to process it. Also, there is a direct relationship between the image size and the amount of cosmic rays detected on it. But, ACS images show more cosmic rays impacts than the WFC3, despite the images sizes are similar. The reasons for this difference may be studied on future works analyzing these results.

Table 5. Mean processing time and mean cosmic rays detected by instrument

Instrument	Image size (pixels)	Mean processing time (s)	Mean cr count found
ACS	2048 × 4096	252.19	375283.00
NICMOS	256 × 256	16.91	2368.515
STIS	1024 × 1024	23.27	20638.76
WFC3	2051 × 4096	108.24	54422.68

The first attempts to run the complete pipeline over Azure platform resulted on unexpected high processing times. After some basic debugging and analysis we realized that the main reason of the slow processing was related to the step where the cosmic rays are individually inserted into the corresponding table. On most of the images the impact of cosmic rays determines several thousand of strikes. Therefore, several thousand of inserts are needed for each image. Due to an Azure platform restriction, only 100 entries can be inserted at a time. As a result, processing an image took around 10 min per file in average, with some cases where the processing of a single image took over 30 min. For this reason, all reported results do not include the individual cosmic ray insertion time but aggregated statistics for the cosmic rays list (e.g. mean flux, mean length, flux percentiles, etc.). This decision does not affect the execution times comparison reported, since the individual cosmic rays information storage was not developed nor executed on the Apache Mesos architecture.

6 Conclusions and Future Work

In this work, we extend and improve an existent distributing computing architecture using Microsoft Azure for processing images within the project "Geophysics using Hubble Space Telescope". We proposed a pipeline where the images to be processed are stored as blobs on the cloud platform for providing data replication and high availability. The computation was performed using Azure virtual machines running the developed application code using Python. The main results indicate that the complete image dataset was processed on a time period around 10 days and the speedup was improved in a factor of 6.57× in comparison with the previously developed Mesos architecture for the 1000 images testing dataset using 10 processing nodes.

The main contribution of this article is that a distributed architecture running on a cloud platform can perform better regarding makespan and flowspan metrics. This performance improvement contributes to reduce times in astronomical images processing tasks. As a result, we obtained a result set containing information about cosmic rays impact on HST instruments in the last decades.

The main lines of current and future work are related to analyzing the results in terms of statistical measurements regarding cosmic rays impact on Earth's atmosphere. These future lines of work are intended to be pursued by astronomical and geological researchers.

Acknowledgments. This work has been partly supported by CSIC, ANII, and PEDECIBA (Uruguay).

The developed architecture contributes to project "Geophysics using Hubble Space Telescope" [11], to exploit the HST capabilities as a cosmic ray detector for analyzing the magnetosphere current strength. Using our results the researches will combine HST results with measurements of solar activity, cosmic ray flux on Earth's surface, and geomagnetic data to understand external field variations.

Computing and storage resources were provided by Cluster FING and the "Microsoft Azure Sponsorship" program intended for researchers. This sponsorship included a cost-based usage of all the services provided by Microsoft Azure.

References

1. Astroplan: What are the IERS tables and how do I update them? http://astroplan.readthedocs.io/en/latest/faq/iers.html. Accessed July 2017
2. Christian, E.: Cosmic rays. https://helios.gsfc.nasa.gov/cosmic.html. Accessed July 2017
3. Mewaldt, R., Cummings, A., Stone, E.: Anomalous cosmic rays: interstellar interlopers in the heliosphere and magnetosphere. Eos Trans. Am. Geophys. Union **75**(16), 185–193 (1994)
4. Myers, T.: Designing a scalable partitioning strategy for Azure table storage. https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/designing-a-scalable-partitioning-strategy-for-azure-table-storage. Accessed July 2017
5. Myers, T., Macy, M., Squillace, R.: Get started with the Azure Batch Python client. https://docs.microsoft.com/en-us/azure/batch/batch-python-tutorial. Accessed July 2017

6. Nesmachnow, S.: Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República. Revista de la Asociación de Ingenieros del Uruguay **61**(1), 12–15 (2010)
7. Pence, W., Chiappetti, L., Page, C., Shaw, R., Stobie, E.: Definition of the Flexible Image Transport System (FITS), version 3.0. Astron. Astrophys. **524**, A42 (2010)
8. Roth, J., FitzMacken, T., Lian, J., Kemnetz, J., Squillace, R.: Azure subscription and service limits, quotas, and constraints. https://docs.microsoft.com/en-us/azure/azure-subscription-service-limits. Accessed July 2017
9. Schnyder, G., Nesmachnow, S.: Improving the performance of cosmic ray detection using Apache Mesos. In: International Supercomputing Conference in México, pp. 1–15 (2016)
10. Smith, A., McDonald, R., Hurley, D., Holland, S., Groom, D., Berkeley, L., Brown, W., Gilmore, D., Stover, R., Wei, M.: Radiation events in astronomical CCD images, vol. 183, pp. 172–183 (2002)
11. Tancredi, G., Cromwell, G., Deustua, S., Gonzalez, G., Nesmachnow, S., Schnyder, G.: Geophysics using Hubble Space Telescope, Hubble Space Telescope Cycle 24 approved proposal (2016)
12. The Astropy Collaboration, Robitaille, T., Tollerud, E., Greenfield, P., Droettboom, M., Bray, E., Aldcroft, T., Davis, M., Ginsburg, A., Price, A., Kerzendorf, W., Conley, A., Crighton, N., Barbary, K., Muna, D., Ferguson, H., Grollier, F., Parikh, M., Nair, P., Gnther, H., Deil, C., Woillez, J., Conseil, S., Kramer, R., Turner, J., Singer, L., Fox, R., Weaver, B., Zabalza, V., Edwards, Z., Azalee, K., Burke, D., Casey, A., Crawford, S., Dencheva, N., Ely, J., Jenness, T., Labrie, K., Lim, P., Pierfederici, F., Pontzen, A., Ptak, A., Refsdal, B., Servillat, M., Streicher, O.: Astropy: a community python package for astronomy. Astron. Astrophys. **558**, A33 (2013)
13. Tody, D.: The IRAF Data Reduction and Analysis System, pp. 1–20 (1986)

Author Index

Aliaga, José I. 111
Arkose, Tugberk 3
Armenta-Cano, Fermín 384

Babenko, Mikhail 370
Barros, Felipe Sodré M. 218
Bentes, Cristiana 71
Blanco, Aníbal M. 255
Boisson, Jean-Charles 87

Cáliz-Ospino, Rodolfo 399
Cappagli, Pablo 307
Carrasco, Diego 307
Carreño, Emmanuell Diaz 203
Carvalho, Otávio 203
Carvalho, Pablo 71
Castro, Harold 399
Cataldo, Edson 71
Chavarriaga, Jaime 399
Clua, Esteban 71
Colavecchia, Flavio D. 307
Corbellini, Alejandro 235
Cortés-Mendoza, Jorge M. 370
Cristal, Adrian 3

da Silva, Luís Alexandre E. 218
Damiani, Lucia 255
Daneshpajouh, Habib 87
de Jesus, Leonardo Araújo 321
de Oliveira, Daniel 321
de Siqueira, Marinez Ferreira 218
Delisle, Pierre 87
Diaz, Ariel Ivan 255
Dorronsoro, Bernabé 277
Drozdov, Alexander Yu. 384
Drummond, Lúcia M. A. 71, 321
Du, Zhihui 370
Dufrechou, Ernesto 111
Dupros, Fabrice 101

Etancelin, Jean-Matthieu 55
Ezzatti, Pablo 111

Forero-González, César 399
Frascarelli, Daniel 291

Gadelha Jr., Luiz M. R. 218
Galaviz-Alejos, Luis-Angel 384
Gall, Guilherme M. 218
Galleguillos, Cristian 169
García Garino, Carlos 353
Garcia, Manuel 203
Garí, Yisel 353
Geier, Maximiliano 185
Godoy, Daniela 235

Iparraguirre, Javier 255
Iturriaga, Santiago 21, 337

Kiziltan, Zeynep 169
Krajecki, Michael 55, 87

Lima, Rafael Oliveira 218

Markovic, Nikola 3
Martínez, Víctor 101
Marzulo, Leandro A. J. 71
Massobrio, Renzo 277
Mateos, Cristian 154, 235, 353
Mayo-García, Rafael 125
Meneses, Esteban 250
Miranda-López, Vanessa 370
Mocskos, Esteban 38, 185
Monge, David A. 154, 353
Moríñigo, José Antonio 125
Muñoz, Andrés 399
Muraña, Jonathan 21

Navaux, Philippe O. A. 101, 203
Nemirovsky, Daniel 3
Nemirovsky, Mario 3
Nesmachnow, Sergio 21, 262, 277, 291, 337, 370, 414
Netti, Alessio 169

Osthoff, Carla 218
Otero, Alejandro 38

Padilla-Agudelo, Jesse 399
Padoin, Edson L. 101
Piñeyro, Leonardo 262

Quintana-Ortí, Enrique S. 111

Radchenko, Gleb 370, 384
Renard, Arnaud 55
Rocchetti, Nestor 291
Rodríguez-Pascual, Manuel 125
Roloff, Eduardo 203

Sánchez-Tapia, Andrea 218
Schiaffino, Silvia 235
Schnyder, Germán 414
Sergiyenko, Oleg 384
Serpa, Matheus 101
Soba, Alejandro 38, 139
Spillner, Josef 154

Tancredi, Gonzalo 291, 414
Tchernykh, Andrei 21, 370, 384

Unsal, Osman 3

Valero, Mateo 3
Vinazza, David 38

Yahyapour, Ramin 384

Zakaria, Nordin 87
Zunino, Alejandro 235