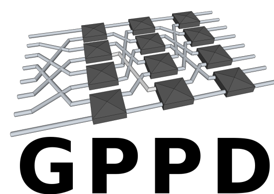
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Grupo de Processamento Paralelo e Distribuído

Proceedings

XVI Workshop de Processamento Paralelo e Distribuído

WSPPD 2018

5 de Setembro de 2018
Auditorio Centro de Eventos do Instituto de Informática
Porto Alegre, RS
ISSN: 2175-6848



List of Sessions

Session 1: Big Data and Cloud Computing **1**

1 Urban Computing Experiment by Mixing Fog Computing Simulation and Public Open Street Map Data
Lucas Alberto Santos and Claudio R. Geyer

5 Understanding and Minimizing Disk Contention Effects for Data-Intensive Processing in Virtualized Systems
Kassiano Matteussi and Claudio Geyer

Session 2: Fault Tolerance and I/O **7**

7 Estimating the Reliability of HPC Applications
Daniel Alfonso Gonçalves De Oliveira, Francis Birck Moreira, Paolo Rech and Philippe Navaux

9 A Full Year I/O Request Size Analysis of HPC Applications on Intrepid Supercomputer
Valéria Soldera Girelli, Jean Luca Bez, Francieli Z. Boito, Pablo José Pavan and Philippe O. A. Navaux

13 I/O Workload Overview of the Applications on Intrepid Supercomputer
Pablo José Pavan, Valéria Soldera Girelli, Jean L. Bez, Francieli Z. Boito and Philippe O. A. Navaux

Session 3: Performance Evaluation **17**

17 StarPU scheduler vs. the analyst: who is right?
Vinícius Garcia Pinto, Lucas Mello Schnorr and Arnaud Legrand

19 Investigating Memory Operations Performance in the StarPU Runtime
Lucas Nesi and Lucas Mello Schnorr

23 Parallel Workflow Support for StarVZ using Drake
Guilherme Rezende Alles and Lucas Mello Schnorr

27 GROUPLB: Load balancer proposal to reduce the execution time and energy consumption of parallel applications
Giovane Lizott, Vinicius Manica Mastella, Pablo José Pavan and Edson Luiz Padoin

31 **List of Authors**

Urban Computing Experiment by Mixing Fog Computing Simulation and Public Open Street Map Data

Lucas Alberto S. Santos, Claudio F. R. Geyer
Instituto de Informatica
Universidade Federal do Rio Grande do Sul (UFRGS)
Grupo de Processamento Paralelo e Distribudo
Porto Alegre, RS, Brasil
lucasa@softwarelivre.org, geyer@inf.ufrgs.br

Abstract

Fog Computing has attracted the attention of researchers in the field of Large Distributed Systems for its ability to handle the limitations of the cloud computing paradigm when applied to the reality of IoT and its massive amount of distributed sensors and wireless mobile devices at the edges of the network. Simulation of Fog computing makes possible to experiment with complex distributed processing scenarios at a very low cost compared to creating real testbeds.

In this work we extend the iFogSim Fog Computing simulation framework to more appropriate simulate a smart city experiment. We developed a experiment to help locating citys stolen cars where fog devices distributed on the city's electronic radars capture images that are processed to detect stolen car's plates. The public open GIS data of the colaborative Open Street Map project serve as a reliable basis to support Urban Computing research, this work contributes to the field as an exercise of urban computing simulation integrated with open urban data.

1. Introduction

Urban Computing seeks to optimize the processes of life at cities by intelligently acting on the massive data generated by various local sources such as people, organizations, houses, buildings, sensors, devices, vehicles, etc. This area of research in Intelligent Cities aims to analyze a large amount of heterogeneous data to extract solutions to address critical problems for cities such as pollution, energy consumption and congestion. This localized computing model connects urban sensing, data management, data analysis, and service delivery in a recurring process of continuous improvement of people's quality of life, urban control systems, and the environment [2].

According to [7], the Internet of Things can act, as well as other environments, in cities as well. The application of the IoT paradigm in the urban universe is of great interest to public managers who seek to make better use of state resources, advancing the quality of services offered to citizens while reducing the operational costs of public administration.

2. Fog Computing

Recently, has become popular in the research area of distributed systems, the Fog Computing model to deal with the limitations of the Cloud Computing paradigmn. The centralized model of the remote public data center does not scale when applied to thousands of sensors geographically distributed and connected to intelligent things at the edges of the internet [2]. Fog Computing bridges the gap in the Internet for Things (IoT) with a proposed computing and communication architecture that distributes computing, control, and storage functions to near end-user devices by introducing new devices (fog nodes) and ad-hoc networks between the central cloud and users at the edge of the Internet [5].

Researching in the field of urban computing are very laborious and costly to carry out real-scale tests, due to the complexity of this intrinsically interdisciplinary theme, the number of actors involved and the high scale of urban processes. Simulating tools and techniques are quite popular in the literature for experimental studies on distributed computing architectures. The simulation anticipates the behavior of the virtual components, helps researchers to understand and improve their modeling more efficiently [6]. For the simulation of fog computing scenarios, was recently published the iFogSim tool [4] based on the CloudSim platform [3], a popular Java framework for Discrete Event Simulation. The iFogSim framework extends the CloudSim platform adding fog computing entities such as Sensor, Actuator, Fog Device, Cloud, etc.

3. Extending the iFogSim Framework

The iFogSim architecture has been extended to include extra information properties to the entity classes of the framework. The *FogDevice class* has been extended into a new *SmartFogDevice class* to better support geographic properties and spatial range information of wireless network radio signal.

Our implementation is a prototype of a Urban Computing simulator because it incorporates spatial local context to the fog computing simulation engine. We implemented a function to import geographic locations from a *GeoJSON format* file, which is a standard format for exchanging geographic data collections.

3.1. Experiment - Locating Stolen Cars

The source code of iFogSim includes a demo source code "Case Study 2 - Intelligent Surveillance through Distributed Camera Networks". This scenario described by the authors follows the *Sense-Process-Actuate model*, where several surveillance cameras links to a network of fog devices cooperating with the Cloud to process machine learning computation task over massive data from sensors.

In this work we implemented an experiment inspired by the *Sense-Process-Actuate* scenario proposed at the iFogSim original paper. In our experiment, image sensors are located in the sixty four electronic radars of the city of Porto Alegre - RS. When a car is photographed by a radar, a plate recognition program runs over the image to locate plate signs of recently robbed cars. If car plate is identified, a notification message is then sent to the mobile device of the end-user, located at the Polices Palace. The `cloud data center` is located at the headquarters of the Municipal Data Processing Company (Procempa).

The modular application executed by the fog network has the following data flow:

- The *Motion Capture and Detection module* takes periodic photos of the traffic path as the tuples are routed if there is a perception of a moving car in the image.
- The *Plate Recognition module* looks for suspicious plates in the image. This module applies a machine learning algorithm type with high computational demand.
- If a stolen car is detected, the image is immediately forwarded to a *Post-Processing module* that envelops the suspect image and forwards it to the end-user device, including additional metadata from where and when the visual record was taken.

The main technical aspects of simulation configuration are:

- Each of the 64 electronic radars of Porto Alegre is a local Cyber-Physical System (CPS) composed by:
  - Sensor: 1 cmera 720p que captura 2 frames/seg.
  - Fog/edge device: 1 ARM hardware - Raspberry Pi 3, clock 1 GHz and 1 GB RAM.
- Cloud has 16 CPUs of 3 GHz and total of 24 GB RAM.
- The Wifi Routers are connected to the Cloud by a link of 10 MByte/seg.
- The Radars edge devices connects to Wifi Routers that are in their radio signal range (local neighbourhood). We implemented an algorithm to optimize the distribution of wireless network routers in the city, to promote that a same wifi point could be shared between multiples geographically closed Raspberry Pi devices.
- The Module Plate Recognition has an output tuple rate of 0.5% fowarded, so only a small amount of the plate images will be given as positively identified and then forwarded to the mobile device of the End User.

This smart city simulation scenario was organized with iFogSim's components abstractions of *Sensor*, *Actuator*, *Application*, *Network Links*, *Fog Devices* and *Cloud*. The georeferenced open data information of the traffic lights and radars of the City of Porto Alegre were downloaded from the Overpass API, a Open Street Map (OSM) webservice client [1]. There are many open source tools to convert the OSM XML native format to GeoJSON.

4. Smart City Simulation Results

The experiments were performed with 30 simulation execution, each execution performs 1 hour of simulation time, for each of the scenarios:

- a) execution machine learning by the fog devices;
- b) processing centralized in the cloud.

The results on the **Table 1** are the averaged values of the 30 executions.

Type	Network (MB)	Power Consumed (W)	Latency (ms)
At Cloud	362324,13	37773570,58	222,57
At Fog Devices	6427,26	37404648,77	274,91

Table 1. Results of the Simulation Scenarios

The result of analysis of the information collected from simulations show that:

- The scenario of intensive processing in the cloud floods the WAN network with image files, taking the network bandwidth as a bottleneck.

- At the distributed processing scenario, the images files are processed locally, so only images with detected cars are sent as successful notifications.
- The *energy consumption* is almost similar between the scenarios, indicating that the low-power computing hardware of the edge devices consumes almost the same as the Cloud to process the work. This experiments were copied from the *hardware characteristics* published at the iFogSim paper. We are going to is review and calibrate the energy consumption and MIPS rates of the simulated hardware, to get more realistic results.
- The average *latency time* (delay) between the car photo capture and the end-user notification was lower for the cloud scenario, indicating a expected behaviour that the simulated cloud of 16 3GHz CPUs is more powerful than the processing power of low power ARM devices.

The iFogSim simulator makes possible to collect several useful information for simulated scenario analysis and comparison, below are the information used to compare the scenarios:

- Total simulation time;
- Total memory used in the simulation;
- Total network bandwidth consumed;
- Total energy consumed (cloud + fog + networking);
- Latency between image capture and notification to the end user;

We implemented a interactive graphical UI for visualization of the urban map (Figures 1, 2 and 3) that shows the dynamics of data between the application modules that execute on the fog devices, along the passage of a virtual simulation time. The simulation information is designed as animated overlays on the actual city map, making it possible to direct focus to local information or even broadening the observation to see large urban areas.

The source code of this work with instructions for experiment reproduction can be acessed at "http://github.com/lucasa/ismartsim_project".

5. Future Work

The Java abstractions of the iFogSim framework for the Fog Computing's common entities serve as the basis for creation of a high-level architecture for simulation of smart city scenarios and their heterogeneity of IoT devices, Wifi networks, fiber optics, ad-hoc bluetooth links, urban mobility, software off-loading, etc. We are going to explore others more complex fog computing scenarios and implement

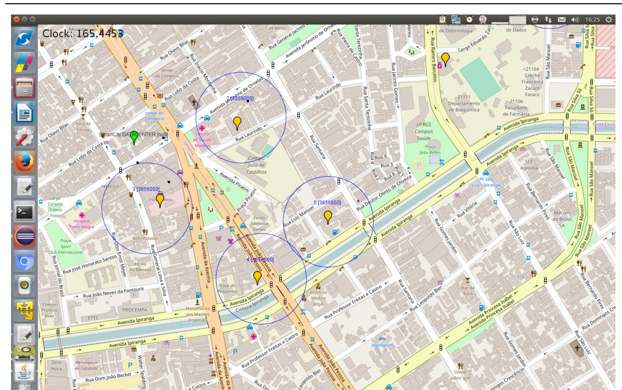


Figure 1. Screenshot of the simulation map showing the location of the fog devices (yellow marker) and the coverage radio range of the wireless network of the attached local router (blue limits).

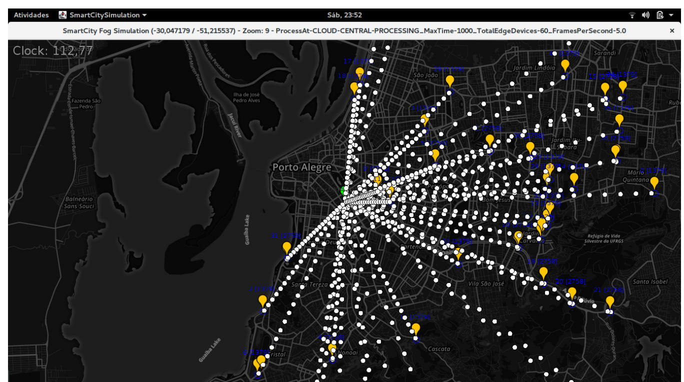


Figure 2. Animation of recorded events log: streams of tuples (raw image files) going through WAN network from the edges to be processed at central cloud.

new simulated abstractions for hardware and software components of complex smart city systems.

The use of public open data sources is a very interesting topic for smart city interdisciplinary research, going further Open Street Maps GIS data to get the simulation more realistic by using updated raw data from auditabled sources.

The user interface for editing and replaying can be rewrite as web component to run in a internet browser, opening the smart city simulation area to the world of creative Javascript and advanced usability studies. As a online tool, we get platform that potentially allows collabora-

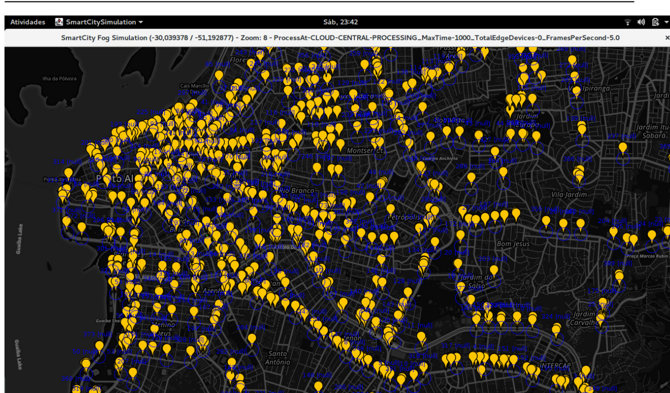


Figure 3. Visualization of a hypothetical scenario of simulation (not executed in this work) with sensors and Wifi networks distributed by all the traffic lights of Porto Alegre.

tive work in a same simulation project.

6. Conclusions

Fog Computing have been attracting the attention of researchers in the field of Large Distributed Systems for its ability to handle the limitations of the cloud computing paradigm in front of a new paradigm of location-aware data abundance by small massive distributed IoT sensors and low-power mobile computing hardwares at the edges of the network. There are still few simulators like iFogSim for the development of fog computing experiments.

In this work we implemented a support for import and visualization of georeferenced information in the backend of the simulation engine IFogSim/CloudSim. Thus, the potentialities of the iFogSim framework for cloud computing simul ation become more useful for simulating Urban Computing in smart city scenarios. The experiment implemented on the iFogSim framework, although quite simple, was useful for study a variety of configuration parameters and events generated by the simulation engine.

The simulation of Smart City experiments, when uncomplicated and high level, opens a whole space for multidisciplinary academic research aimed at the application of technologies of computation and communication to solve current and future problems in the big cities of the world.

References

[1] Openstreetmap community: Osms overpass application. Accessed April 4, 2010.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[3] R. Buyya, R. Ranjan, and R. N. Calheiros. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 1–11. IEEE, 2009.

[4] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[5] R. Mahmud, R. Kotagiri, and R. Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, Singapore, 2016.

[6] A. Sulistio, C. S. Yeo, and R. Buyya. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. *Software: Practice and Experience*, 34(7):653–673, 2004.

[7] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

Understanding and Minimizing Disk Contention Effects for Data-Intensive Processing in Virtualized Systems

Kassiano José Matteussi
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil 91509–900
Email: kjmatteussi@inf.ufrgs.br

Claudio Fernando Resin Geyer
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil 91509–900
Email: geyer@inf.ufrgs.br

***Abstract*—Distributed computing systems (e.g., clouds) have been widely employed to support an expanding range of applications. As the scale of data generation grows in regards to volume, velocity and variety (3Vs of big data), data-intensive processing became essential to extract valuable information from complex datasets. In this scenario, the infrastructure needs to meet the scaling demand of applications and must use resource management techniques to avoid interference problems. Literature review mainly focuses on CPU and memory solutions to handle resource contention problems in data-intensive processing. Complementarily, this paper further analyses and proposes techniques to minimizes disk contention effects in order to improve application performance in virtualized systems - technology that drives the cloud computing environment. For this objective, we present a general-purpose resource management strategy that adjusts dynamically disk I/O utilization rates. Results showed that the proposed approach improves application's performance by up to 26%.**

Keywords*— *Big Data, data-intensive processing, virtualization, resource contention, interference, disk contention.

I. Acknowledgment

The authors would like to thank Dell Inc and the following Brazilian Agencies: FAPERGS Projects ”GREEN-CLOUD - Computação em Cloud com Computação Sustentável” (#16/2551-0000 488-9) and ”SmartSent” (#17/2551-0001 195-3), CAPES, CNPq and PROPESQ-UFRGS-Brasil for supporting this work.

Estimating the Reliability of HPC Applications

Daniel Oliveira, Francis Birck Moreira, Paolo Rech, Philippe Navaux
Institute of Informatics, UFRGS
Porto Alegre, Brazil

***Abstract*—The error rate of current High Performance Computing (HPC) systems is already in the order of one per dozens of hours. Understanding the reliability behavior of HPC applications will be required for the next generation of supercomputers. Using the reliability behavior one can select efficient mitigation techniques for the application and fine-tune parameters such as checkpoint frequency. In this paper, we investigate the application of a machine learning model to predict the reliability behavior of HPC applications. We inject faults in more than 30 HPC applications executing in the Intel Xeon Phi Knights Landing (KNL) and use profiling information to build a predictive model with Support Vector Machines (SVM). We show that the model can predict the Program Vulnerability Factor (PVF) with an average relative error of 7% for certain classes of algorithm, such as linear algebra and sorting. The average relative error for all algorithm classes is 22%. Such a fast and straightforward prediction model can be effective as a filter to select the most unreliable applications to perform an in-depth analysis.**

A Full Year I/O Request Size Analysis of HPC Applications on the Intrepid Supercomputer

Valéria S. Girelli[1], Jean Luca Bez[1], Francieli Z. Boito[2], Pablo J. Pavan[1] and Philippe O. A. Navaux[1]
[1]Universidade Federal do Rio Grande do Sul — Instituto de Informática, Brazil
{vsgirelli, jlbez, pablo.pavan, navaux}@inf.ufrgs.br
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract

This study used data from an entire year of characterization with Darshan on the Intrepid Blue Gene/P supercomputer, at Argonne. Considering that data access is a bottleneck for several HPC applications, understanding their I/O behavior may help us apply optimization techniques at the I/O system. By analyzing the collected data, we could observe that POSIX is still widely used by the applications running on Intrepid. Additionally, some of the requests issued by the applications are very small which generally translates into poor I/O performance.

1. Introduction

In High Performance Computing systems (HPC), besides the high processing performance, the applications also demand a high storage capacity and efficiency in the data access. These applications deal with a great amount of data, and hundreds of computing nodes can access the storage system concurrently. Therefore, performing I/O operations may become a bottleneck for an increasing number of HPC applications, due to the historical gap between processing and data access speed.

Parallel File Systems (PFS) act providing an abstraction of the data on the storage system. They receive requests from the computing nodes, process these requests and access the storage devices. However, some problems may appear depending on the way these requests are performed. Occasionally, the requests sent to the storage system are too small, transferring small data amounts that hardly compensate the cost of accessing the storage devices [3, 10]. One solution to this problem is to aggregate the requests, one of the possible optimizations that can be done at the I/O system. Another situation, though is not the focus of this work, is when the access is not aligned to the stripe used by the PFS, what may further degrade performance. In this case, applications can use interfaces such as MPI-IO, that may assist in the aligned access.

However, the applications can present different access patterns. In consequence, to apply optimization techniques, we must at first understand the application I/O behavior. Tools like Darshan [4] provide a characterization of such behavior by creating profiles of the operations. Darshan was developed at Argonne Leadership Computing Facility (ALCF)[1] and captures this sort of information at the application level. Therefore, with the objective of understanding the I/O global behavior in a supercomputer, this study analyzed data collected using Darshan on the Intrepid Blue Gene/P, the ALCF supercomputer.

The remainder of this paper is organized as follows. The information about the collected data used and the methodology applied are detailed in Section 2. Analysis and results are presented in Section 3. Section 4 discusses related work. Finally, Section 5 concludes this paper and discusses future work.

2. Methodology

During the years of 2010, 2012 and 2013, data from the execution of a variety of scientific applications was collected by the Darshan I/O Characterization Tool on the supercomputer Intrepid Blue Gene/P, at ALCF. Darshan intercepts I/O function calls and records a fixed-size collection of statistics for each file that is opened by the application [1]. The collected information includes access patterns, access sizes, operation counters and time spent in I/O operations. Darshan generates a separate log file for each job, and stores this information in a compressed binary format [1].

In this study we analyzed the data collected during the year of 2012 generated by Darshan versions 1.23, 1.24 and 2.0, resulting in 36, 359 and 91.603 jobs, respectively. Until the version 2.0, Darshan only instrumented applications that successfully called `MPI_Init()` and `MPI_Finalize()`.

1 `https://www.alcf.anl.gov/`

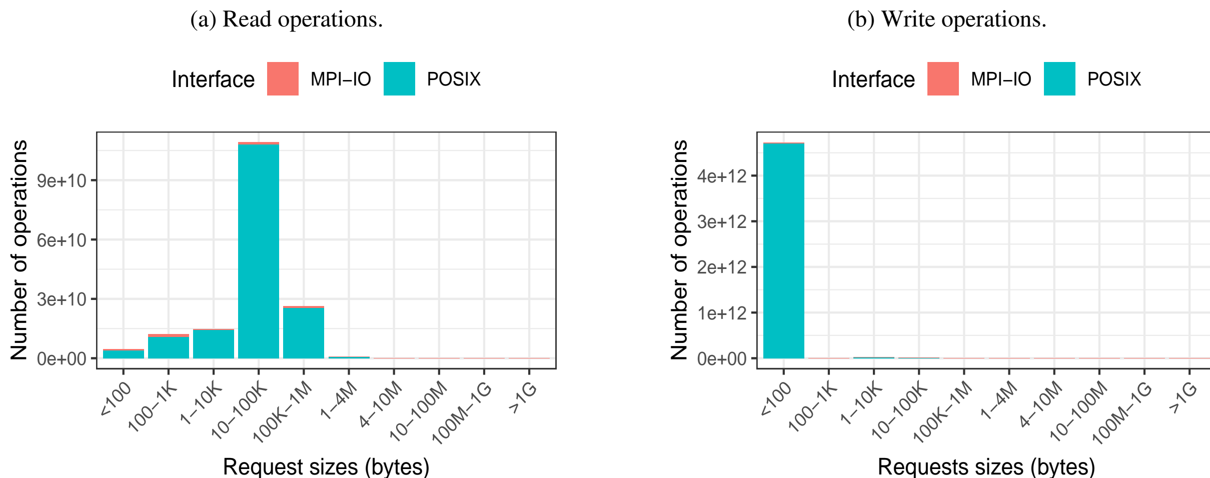


Figure 1: Access size distribution for operations by interface. The y-axis has different scales.

Therefore, although Darshan was enabled for all users, the application coverage rate varied between 20% and 80% from week to week [1]. Some of the collected information was anonymized before being publicly available. The anonymized information includes the job identifier, user identifier, and application name.

Since the Darshan logs are stored in a compressed binary format, to extract and analyze interesting information for our analysis, we used the darshan-parser tool, provided by Darshan, to generate a text file. This text file is organized in two sections[2]. The first one, at the top of the file, describes information about the executed application. The second section contains the observations of each counter captured by Darshan, in a tabular format.

For our analysis, the relevant information about the application was: application identifier, represented by the column `exec`; the job identifier, represented by `jobid`; user identifier, as `uid`; number of processes, as `nproc`, and `runtime`. Likewise, we calculated total I/O time in the same way Darshan does, using information about the slowest MPI rank. Additionally, Darshan captures request information and stores in counters separated in operation type (read or write) and interface (`AGG` for MPI-IO, and POSIX). For each operation in each interface, the access sizes are divided in the following bins: 0-100 bytes, 100 bytes - 1 KB, 1-10 KB, 10-100 KB, 100 KB - 1 MB, 1-4 MB, 4-10 MB, 10-100 MB, 100 MB - 1 GB, and 1 GB - PLUS. Thus, we used Python to extract all the useful data from the text file, storing them in CSV format, and made the analysis using R.

2 `http://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html`

3. Results and Analysis

Analyzing the number of read and write operations performed using each interface, we observed that surprisingly 97.3% of read operations were being performed by POSIX. For the write operations, the percentage of POSIX utilization reaches 99.2%. As expected and presented by previous work [2, 7, 8], POSIX is still more widely used if compared to MPI-IO. However, the data collected in 2012 further highlights the gap between the POSIX and MPI-IO utilization on the Intrepid supercomputer.

Figure 1a shows the distribution of the read access sizes observed for both POSIX and MPI-IO interfaces. The most common read size was between 10KB and 100KB, representing 64.3% of the read operations. Figure 1b shows the distribution of the write access sizes, also for both interfaces, and the most used write size was up to 100 bytes, which is a really small access size. This size represents 98.7% of the write operations.

We can notice, therefore, that the applications observed during the characterization made even smaller requests than the observed in previous work [2, 9, 6], mainly on write operations. This type of small accesses may degrade performance [3, 10], once they define the transference size between processing nodes and the storage devices.

As presented in Figure 1, the great number of requests issued using POSIX determine the overall request size distribution of the collected data. Therefore, the distribution for both read and write operations using POSIX only is very similar to the overall distribution, as shown in Figure 2.

For the requests realized with MPI-IO, we can observe a spread distribution in Figure 3. The two most common access sizes intervals for read operations were between 100 bytes and 1KB and between 10KB and 100KB. These two

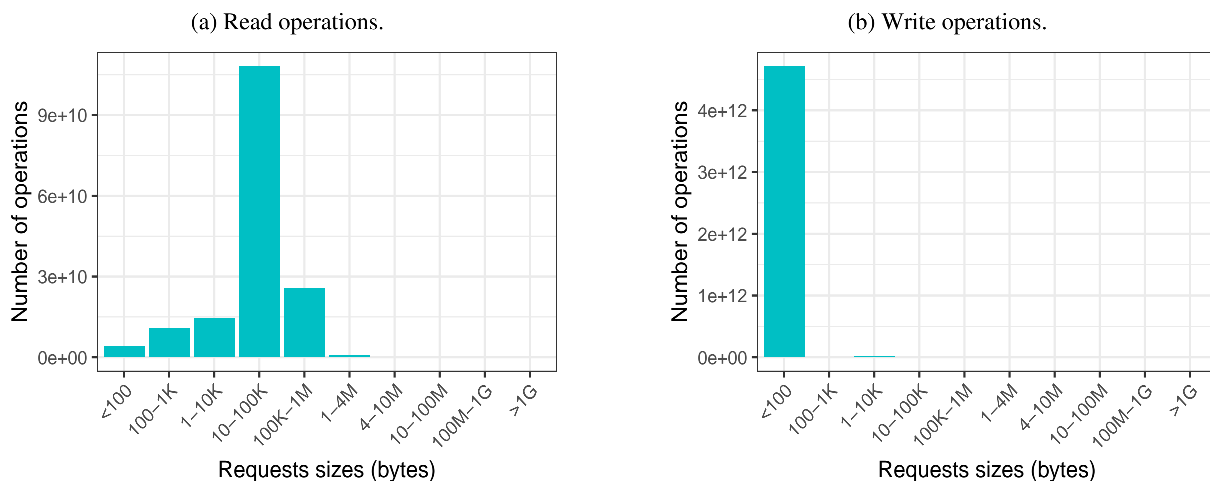


Figure 2: Access size distribution for operations using POSIX. The y-axis has different scales.

intervals represent up to 34.2% and 23.7% of the total read operations performed by MPI-IO, respectively. For the write operations, the most popular access size was also up to 100 bytes, followed by requests ranging between 10KB and 100KB. These two intervals represent up to 53.6% and 20.6% of the total write operations realized using MPI-IO, respectively.

If compared to the requests performed by POSIX, the access size distribution observed in the requests made using MPI-IO is wider. One explanation for this is the possibility to define more complex datatypes with MPI-IO. When a process wants to access small and sparse portions of a file, a datatype can be defined by aggregating this portions into a large request. Moreover, when applications use MPI-IO collective operations, usually a small number of processes do the accesses and then distribute the data to the others [5]. The number of proccess is defined following some heuristics or can be defined by the user.

The write operations presented a condensed distribution variation when compared to the read access size distribution, for both interfaces. This may be caused by the individual behavior of the applications and requires a deeper study to understand.

4. Related Work

Wang *et al.*, in 2004, analyzed two physics applications and the *ior* benchmark on a large Linux cluster with more than 800 dual processor nodes at the Lawrence Livermore National Laboratory (LLNL) [9]. Usually, each application presented only one or two typical request sizes. Large requests from several hundred KB to several MB were very common. In some cases, however, small requests accounted for more than 90% of all requests, but the greatest amount of data was still transferred by large requests.

In a work made in 2010 [6], Kim *et al.* characterized the workload of Spider, a Lustre-based storage cluster at Oak Ridge National Laboratory (ORNL). They observed three main request sizes: less than 16KB, 512KB and 1MB. This three request sizes represented more than 95% of the total requests. About 50% of the request of less than 16KB were write operations, while the read operations represented 20%. They also correlated the access sizes with the bandwidth, observing that the higher bandwidth was attained with 1MB large requests.

In a previous work published in 2011 [2], Carns *et al.* analyzed the behavior of 66 science and engineering applications. The data was collected with Darshan during two months of 2010, also on the supercomputer Intrepid, Argonne. They demonstrated that the most popular read size was between 100KiB and 1MiB, while the most popular write size was between 100 bytes and 1KiB. However, a deeper investigation revealed that a few applications influenced the access size observed. If those applications were removed from the analysis, then the most popular size for both reads and writes was 100KiB to 1MiB. The work also demonstrated that some of the applications increased their performance when using larger accesses sizes.

Although Carns *et al.* also analyzed data from the I/O workload on Intrepid, their study used a smaller dataset. On the other hand, this study was the first to investigate the entire year of 2012 in this same supercomputer. We observed the access size distributions taking into account the different I/O interfaces.

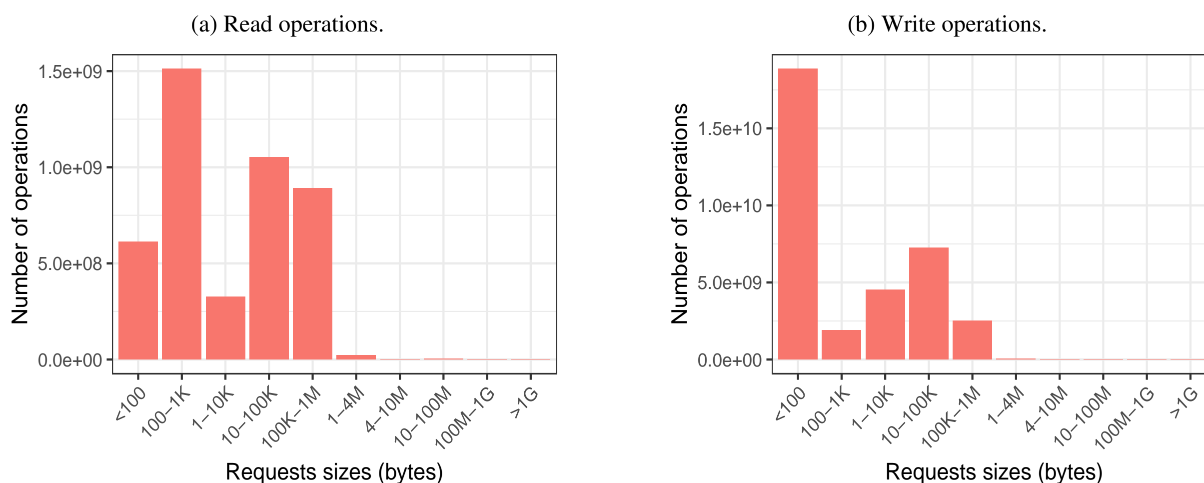


Figure 3: Access size distribution for operations using MPI-IO. The y-axis has different scales.

5. Conclusion and Future Work

This study used data collected during an entire year of characterization by Darshan. We could observe with a considerable amount of data that POSIX is still surprisingly used by the applications running on Intrepid, overcoming 97% of utilization for read operations and 99% for write operations. Applications are also performing very small write requests that do not exceed 100 bytes, what may indicates that I/O is being performed in a really inefficient way, probably accessing a few variables at a time. Therefore, this great amount of small operations using POSIX is not taking advantage of the request aggregation made by MPI-IO and neither of any other high-level interface.

In future work, we would like to investigate if this behavior is the result of a small group of applications performing a huge part of the requests, or if it represents the global behavior observed in Intrepid. We should also look into the amount of data being transferred by each access size observed, and compare which are the access sizes responsible for the largest amount of transferred data. Another possible analysis is to investigate the system time spent in the operations performed with each access size.

Acknowlegments

The research has received funding from PIBIC CNPq-UFRGS and PROBIC FAPERGS-UFRGS. It was also supported by Petrobras project, grant n. 2016/00133-9.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] P. Carns. ALCF I/O Data Repository. Technical report, Argonne Leadership Computing Facility, Feb 2013.

[2] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *Trans. Storage*, 7(3):8:1–8:26, Oct. 2011.

[3] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11, May 2009.

[4] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 Characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009.

[5] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, Dec. 1993.

[6] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer. Workload characterization of a leadership class storage cluster. pages 1–5, Nov 2010.

[7] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 33–44, New York, NY, USA, 2015. ACM.

[8] E. Smirni and D. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33(1):27 – 44, 1998.

[9] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. Mclarty. File system workload analysis for large scientific computing applications. Apr. 2004.

[10] F. Zanon Boito. Transversal I/O Scheduling: from Applications to Devices. page 171, 03 2015.

I/O Workload Overview of the Applications on Intrepid Supercomputer

Pablo J. Pavan[1], Valéria S. Girelli[1], Jean L. Bez[1], Francieli Z. Boito[2], Philippe O. A. Navaux[1],

[1]Federal University of Rio Grande do Sul (UFRGS) – Porto Alegre, RS – Brazil
{pablo.pavan, vsgirelli, jean.bez, navaux}@inf.ufrgs.br
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
francieli.zanon-boito@inria.fr

Abstract

In this work, we analyzed the I/O workload of HPC applications on Argonne's Intrepid Blue Gene/P supercomputer, using Darshan logs. For our analysis, it was necessary to perform a data treatment to filter the information regarding the execution time, I/O time, among others. Initial results show that we had 26,034 applications executed in the year 2012, of which ten applications represent 35.36% of the total number of executions observed and spend 20% of their execution time performing I/O operations.

1. Introduction

In High Performance Computing (HPC) systems, supercomputers are the most significant example, several applications require considerable computational power. These applications often use large volumes of data and I/O operations are made in Parallel File Systems (PFS), where dedicated machines act as the data servers. The servers aim to receive requests from the computational nodes and to process them, accessing the storage devices. These I/O operations are a bottleneck for a growing number of applications due to the difference between the processing speed and the data access speed [1].

We can characterize the application's behavior to look for new ways to improve I/O operations. This characterization can be done by generating traces or I/O operations profiling, using some tool such as Darshan[1]. Darshan was developed by the Argonne Leadership Computing Facility (USA) and aims to characterize the application's I/O operations. It shows a representation of the application behavior, reporting information such as access pattern, time and number of operations, among other information.

In this work presents an analysis of the I/O workload of the applications executed in a supercomputer, by data collected through Darshan on the Intrepid Blue Gene/P supercomputer located in Argonne.

The remaining sections of this paper are organized as follows. Section 2 discusses related work. In Section 3 we present the details of the evaluation methodology. In Section 4 we address the results obtained from our analysis. Finally, the Section 5 we show the conclusion and the future work.

2. Related Work

Zoll et al. [5] studied a set of application-side I/O traces in an HPC environment (the ASCI cluster from the LLNL) using the Lustre parallel file system for storage. Their traces were obtained in 2003 with the *strace* tool, ranged from tens of seconds to half an hour, and included two scientific applications from the physics domain and three benchmarks generated with IOR (file-per-process, shared-contiguous and shared-strided). They concluded that a Markov model could not represent the request arrival rate of applications' I/O streams present self-similarity. They presented a stochastic model to predict I/O arrival rate.

Wang et al. [4] used the same traces aiming at characterizing the HPC I/O workload. They also analyzed files size and lifetime, showing 80% of the files have a size between 512 KB and 16 MB, and these files account for 80% of data stored in the file system. 60% of the files (50% of the bytes) stay in the system from 2 to 8 weeks. Their results for request size and inter-arrival time are somewhat specific to the traced applications and show they issue large numbers of small requests (from a few bytes to 1 MB) in small time intervals.

To motivate their work on cross-application coordination, Dorier et al. [2] used data from the Parallel Workload Archive, from the period between January and September 2009. They showed half the jobs on this platform used less than 2048 cores (1.25% of the machine). Through a simple optimistic model, they used the distribution of some concur-

1 http://www.mcs.anl.gov/research/projects/darshan/

rent jobs to show there is a high probability of having multiple applications concurrently performing I/O operations, even when applications spend as little as 5% of their execution time on I/O.

Luu et al. [3] examine the I/O behavior of thousands of supercomputing applications. They analyze the Darshan logs of over a million jobs over Intrepid and Mira supercomputers, at the Argonne Leadership Computing Facility (ALCF), and Edison, at the National Energy Research Scientific Computing Center (NERSC).

The authors draw several conclusions from these observations. First, they point out that every widely adopted I/O paradigm (file per process, shared file, subsetting I/O) is represented among the best-performing and worst-performing applications. Hence, the usage of a paradigm does not provide any guarantees in terms of performance. Regarding throughput, they demonstrate that almost a third of the jobs have an aggregated throughput of no more than 256MB/s. They also point out that over a third of the jobs spend more time in metadata operations than actually transferring data. Additionally, despite the existence of high-level parallel libraries, three-quarters of the jobs use only POSIX to perform I/O.

In the same way as the works related in this section, we seek to better understand the HPC application's behavior from their I/O operations, analyzing information collected about them. Differently from Zoll et al. and Wang et al., we used as a basis for our analysis the traces obtained transparently by the Darshan profiler, an approach also used by Luu et al. Unlike the other works, we have sought to obtain an overview of the I/O workload of the applications. In this way, such information can be used to make smarter decisions in various I/O optimization techniques such as I/O scheduling, I/O forwarding and reconfiguration of the stack.

3. Methodology

In the Intrepid Blue Gene/P supercomputer data was collected from different applications using different Darshan versions in the years 2010, 2012 and 2013. For this work we only use the observations of 2012, which were generated by versions 1.23, 1.24 and 2.0, totaling 36, 359 and 91,603 observations with each, respectively. We used only data from version 2.0 since previous ones did not capture a counter needed for our analysis.

Darshan saves data collected from applications in a binary format. In order to extract and transform the necessary data for our analysis, we used the *darshan-parser* tool, available with Darshan. It transforms the binary file into a text file, which contains all the counters related to the I/O operations collected by the profiler. The text-format file generated by the *darshan-parser* is organized into two sections. At the file's beginning, stored in the key-value pattern, data is described as the executed application, such as name, job identifier, runtime and number of processes. The rest of the text file contains the observations of each counter captured by Darshan, relative to each file opened by the application (*filehandle*), where each line follows a tabular format.

We divided the collection of relevant data for our analysis and the characterization of the application's I/O workload into stages.. In order to extract the counters, it was first necessary to extract relevant data (Step 1) from the text file generated by *darshan-parser*. For this, we chose to use the Python programming language, which presents libraries that facilitate data manipulation and allow us to easily create dictionaries from the data that were later saved in compressed JSON format in order to reduce the storage space. Subsequently, to perform a specific analysis, we use an application in Python that reads the JSON files generated by Step 1, extracting the relevant data for analysis, and writes a file in CSV format (Step 2), containing a summary of the execution or the information access intervals. We chose the CSV format because it is easily manipulated by the R language, with which we perform the final analysis of the data (Step 3).

4. Results

From the data of 2012, we first look at the behavior of the applications during the whole year and find the number of different applications that were submitted to the execution and applications with the highest number of executions. For this, we used the value of the *exec* collected by Darshan.

In order to understand the annual cluster behavior, in Figure 1 we show the relation between the I/O time (y-axis) and the execution time (x-axis) of all submitted jobs, where each point represents one execution. We color these points according to the I/O percentage that each job performed, where the yellow and the blue represent smaller and higher percentage, respectively. We note that the most executions does not exceed 45,000 seconds executing and 2,500 seconds performing I/O. Within this space, we have 85,710 jobs submitted, and of these, a total of 10,050 jobs have an I/O percentage above 50% and execution time below 4,609 seconds. With this, we can conclude that executions with a high I/O percentage do not run for long periods of time.

In order to exploit these applications that have significant use of I/O, we focused the analysis on the executions whose I/O percentage was above 50% and from these we selected those with highest numbers of submitted jobs. From this filter, we chose three applications. Table 2 presents the anonymized ID of the executable (exec) and the number of times it was observed.

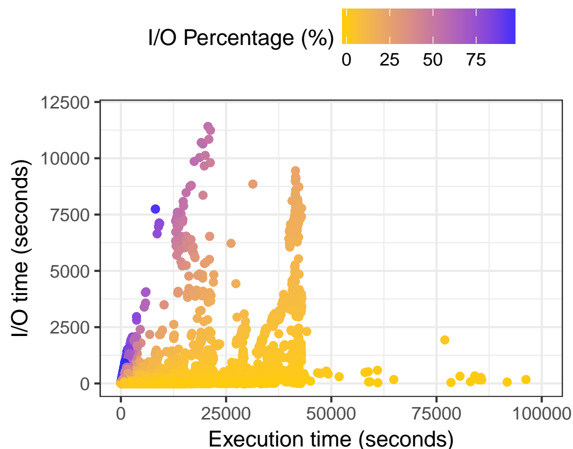


Figure 1. Annual behavior overview of the Intrepid supercomputer, relative to the percentage of time spent on I/O operations per job.

Exec (Anonymised)	Jobs with > 50% of the time in I/O operations
1176110786	980
1338247359	898
902685977	761
Total	2,639

Table 1. Applications with jobs that spend more than 50% of their execution time performing I/O operations.

Expanding the analysis for the exec *1176110786*, we noticed that it executed with 1,024 or 2048 MPI processes. This application was executed 980 times during the period from June 06, 2012 to December 18, 2012. We can see that the first executions used only 2,048 MPI processes, and it can be either the application's tests or the performance on other dates. After this period, from observation 104, dated July 9, 2012, we can observe a change in the application behavior, where the executions began to have a higher I/O operations and the use of 1,024 MPI processes. We can suppose these runs with 1,024 processes can indicate application testing with a new input set.

With 1,024 processess, we can describe one more conclusion about this application. Although transferring less data, the I/O percentage remains high. This scenario may indicate that the entry for 1,024 processes was smaller than the one used with 2,048 processes. However, with the using fewer processes, the execution time also increased, causing the same I/O percentage to be maintained.

For the application with exec *1338247359*, we had 1,051 observations during the period from March 21, 2012 to July 31, 2012. The I/O operations account for more than 50% of execution time in 898 observations. The application used different numbers of MPI processes over different executions. The highest occurrence of executions occurred between 1 and 100 processes and the processes number most used was 64, accounting for 441 observations. In cases about 1 to 100 processes, the maximum data transferred was 27.94GB and the maximum for the application was 135.88GB, using 485 MPI processes. We note that in this application the processes number reflects the data size used.

The application with exec *902685977* was submitted for execution in the period from October 23, 2012 to October 30, 2012, during which period there are 763 jobs. When analyzing this application, at the beginning of the period we had 60 executions with 64 processes, transferring a maximum of 18.05GB of data, and after that, the 701 executions with 32 processes that transferred a maximum of 9.02 GB, for the most part, behave similarly throughout the execution period.

There were also two executions using 128 processes, which transferred a maximum of 36.10GB. These executions can represent application tests since they were in smaller number, they did not repeat themselves, and they presented different behavior of the other executions.

Another analysis with the data was to find the ten most executed applications during the year. The total executions count of these ten applications was 32,535, which represents 35.36% of the total collected in the supercomputer this same year, showing that these applications are indeed significant in our analysis. With the objective to analyze these applications, which had the longest time spent in I/O operations, Table 2 presents the data on the jobs of these applications. The largest I/O time was observed in job *1538301448* relative to the application *1633035531*, with 2.27 hours of its execution time spent in I/O operations, representing 19.46% of the total execution time.

Besides, there are only three jobs out of the ten that have a low time in I/O operations, not exceeding 10% of the total execution time of each one. The remainder of the runs presents vast amounts of time spent with I/O, as can be seen from the job *2939212379* of the application *2425255765*, whose I/O time is the second largest of the realized executions. The time that this execution was performing I/O operations corresponds to almost 80% of the total execution time, totaling 49.45 minutes. The similar behavior we can be observed with job *556397787* of application *1338247359*, which has the third largest time spent on I/O operations and again corresponds to almost 80% of the ex-

Exec (Anonymised)	JOB ID	Execution time (seconds)	I/O Time (seconds)	I/O %	Data transferred (GB)
685531913	3182318857	1,370	3.12	0.22	0.003
1330277471	1330277471	1,314	536.63	40.84	5.69
931947437	2946033326	2,906	985.65	33.91	0.013
3069475893	1473997298	2,808	3.92	0.13	0.012
1074553177	1507238004	1,489	533.55	35.83	1.12
1648769576	1591638629	1,993	191.82	9.62	7.59
1633035531	1538301448	42,003	8,176.28	19.46	6,223.78
2425255765	2939212379	3,720	2,967.61	79.77	2.85
3475271559	567534836	1,512	310.14	20.51	2.25
1338247359	556397787	2,415	1,924.92	79.70	17.80

Table 2. The jobs that have spent the most time on I/O operations over of the ten applications identified as the most executed.

ecution time. In this way, it is clear to see how the ten most executed applications represent significantly the I/O pattern observed in the supercomputer.

5. Conclusion

In this work, we have analyzed the I/O workload of HPC applications executed in the Intrepid Blue Gene / P supercomputer of Argonne (USA). For this we use logs collected with the profiler Darshan, extracting those data relevant to our analysis.

When we analyzed the results of I/O workload found during the year 2012 on the Argonne supercomputer, we obtained 91,973 submitted jobs, which grouped by exec, resulted in 26,034 different applications. From this, we seek to analyze the applications following two approaches. In the first one, we looked for applications that had more than 50% of their time in I/O operations, and when we analyzed three of them, we could notice different behaviors, such as the use of different MPI process numbers, transferred data, execution period during the year and the form of access to your data. The other approach chosen was to analyze the ten most executed applications, representing 35.36% of the total number of executions observed. When we were observed these ten applications, for the most part, spend much of their execution time doing I/O operations, representing more than 20% of their execution time.

As future work we intend to extend this analysis of the characterization of the intervals for all the jobs of Argonne, seeking to group applications that have a similar behavior regarding their phases. Besides, we also intend to aggregate data from multiple applications to have a global view of the use of I/O in a supercomputer.

Acknowledgment

This research received funding from the Petrobras project, grant n. 2016/00133-9. It was also supported by PIBIC CNPq-UFRGS and PROBIC FAPERGS-UFRGS. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] J. L. Bez, F. Z. Boito, L. M. Schnorr, P. O. Navaux, and J.-F. Méhaut. Twins: server access coordination in the i/o forwarding layer. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 116–123. IEEE, 2017.

[2] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 155–164. IEEE, 2014.

[3] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. ACM, 2015.

[4] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. Miller, D. Long, and T. McLarty. File system workload analysis for large scale scientific computing applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2004.

[5] Q. Zoll, Y. Zhu, and D. Feng. A study of self-similarity in parallel i/o workloads. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

StarPU scheduler vs. the analyst: who is right?

Vincius Garcia Pinto
Informatics Institute
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
vinicius.pinto@inf.ufrgs.br

Lucas Mello Schnorr
Informatics Institute
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
schnorr@inf.ufrgs.br

Arnaud Legrand
CNRS, INRIA, LIG
Univ. Grenoble Alpes
Grenoble, France
arnaud.legrand@inria.fr

***Abstract*—State-of-the-art High-Performance Computing (HPC) platforms are built with a hybrid design combining multicore processors and manycore accelerators. This design motivates the use of appropriate programming models (e.g., task parallelism) in order to reduce programming complexity, increase the performance portability and efficiently exploit such multi-level parallelism. Task-based executions are supported by dynamic runtime systems which are in charge of several critical procedures such as scheduling, load balancing, communications, and synchronizations. This way, the overall application performance depends heavily on how efficient and correct are the decisions taken by this runtime. In this work, we propose a workflow to investigate potential scheduling mistakes in the StarPU runtime system. This workflow relies on debugging and simulation strategies to enable us to rebuild the estimations computed by the scheduler at the exact moment when a new task is scheduled. We evaluate our approach by comparing the scheduler estimations and the final execution of a set of delayed tasks. Our results allow us to refute scheduling mistakes and on the other hand to identify that the real origin of the delays is related to pipeline and prefetch mechanisms.**

***Index Terms*—task parallelism, GPU, hybrid-architectures, StarPU**

Investigating Memory Operations Performance in the StarPU Runtime

Lucas Leandro Nesi, Lucas Mello Schnorr
Graduate Program in Computer Science (PPGC/UFRGS), Porto Alegre, Brazil

***Abstract*—Programming parallel applications for heterogeneous HPC platforms is much simpler using the task-based programming paradigm. The applications are modeled as a directed acyclic graph (DAG) of tasks. The simplicity exists because a runtime takes care of all activities usually carried out by the application developer, such as task mapping, process deployment, and load balancing. Besides task scheduling, the runtime is also responsible for handling memory management operations i.e. copying the necessary data to the location where a given task is schedule to execute. Correctly interleaving such memory operations with computation is crucial to achieve high performance, so a worker never wait for data. Poor memory management may be caused by bad scheduling choices or lack of appropriate applications' hints about its tasks dependencies. In this paper, we investigate the CPU-GPU memory management of the StarPU runtime, a well-known task-based middleware for HPC applications. Our results include the identification of poor data handling management when the GPU memory is saturated, ultimately leading to low application performance. Besides fixing the performance issue, we present the design of novel graphical strategies that were fundamental to identify the problem. Our experiments using the dense tiled-based Cholesky factorization show how our fix lead to performance gains of 66% and better scalability for larger input sizes.**

I. Introduction

A challenge found in High Performance Computing (HPC) area is the complexity of programming applications. Especially ones that can intelligently use all the resources' computational power in heterogeneous platforms. The task-based programming paradigm presents some benefits on this matter. It transfers some responsibilities (computation to resource mapping, data management, and communication), that previously relied on the programmers, into a runtime. The task-based applications use a Direct Acyclic Graph (DAG) of tasks as the main structure to schedule them into resources, considering the dependencies and data transfers. Among alternatives like Cilk [1] and Xkaapi [2], StarPU [3] is one example of a runtime using this paradigm. It permits the use of distinct tasks' implementations (CPU, GPU), has different tasks schedulers, and automatically transfers data between resources.

Similar to any other HPC approach, the performance analysis of task-based parallel applications is laborious due to its inherent stochastic nature. StarPU can collect execution's traces that describe the behavior of the application. Different tools can use these traces to help in the performance analysis; one example is the StarVZ workflow [4]. It uses consolidated data science tools and `R` to create trace's visualizations. Various aspects can be studied to improve the application performance, including scheduling decisions, memory transfers, and overall application behavior. In this paper, we focus on the analysis of the memory operations performance, an aspect that has a small presence on performance tools.

The contributions of this paper are the following. **(a)** We include in the StarPU runtime system extra trace information about the memory management activity. It mainly consists of blocks identification on operations, additional attributes on memory chunks, and data coherency states. **(b)** We extended the StarVZ workflow, adding new visual elements, to enable a comprehensive performance analysis over the runtime data management module. For this, we use the previously discussed new traces in StarPU and all available memory management data. **(c)** We present the methodology used with these new features in a real use case application, a dense linear algebra solver called Chameleon. We show how that led to a StarPU software inefficiency discovery, and compare the application performance after our proposed correction patch.

The paper is structured as follows. Section II provides basic concepts on the StarPU runtime system and the application we have used in our experiments: the dense linear algebra Cholesky factorization as implemented by Chameleon/MORSE. Section III presents related work on the visualization of memory management and task-based applications. We also briefly compare ourselves against the state-of-the-art. Section IV introduces the methodology we have devised to investigate memory operations performance in the StarPU runtime, employing modern data science and visualization tools. Section V detail the experiments conducted with the task-based Cholesky factorization, including how we have identified a StarPU software flaw, and the performance increase after the flaw has been corrected. Finally, Section VI concludes this paper with future work.

II. Background Concepts

To explore heterogeneous environments, StarPU allows the tasks to be implemented for different resources, like CPUs, CUDA GPUs, and OpenCL devices. It employs different scheduling heuristics to allocate tasks to resources. Classical heuristics are the `LWS` (local work stealing) and the `DM` (deque model); moreover, more sophisticated schedulers consider additional information. One example is the `DMDA` (deque model data aware); that uses data about the task's dependencies and estimated transfer time to take its decisions [5]. StarPU is responsible for transferring data between resources. For controlling the presence and the coherence of the memory, StarPU attributes a entity to each different resource memory called

memory node. It uses a `modified`, `shared` or `invalid` (MSI) memory system where each data block can assume one of the three states on each memory node [3].

The Chameleon/MORSE package [6] contains a series of solvers for linear algebra implemented on top of StarPU. From the set of available solvers, we adopt the task-based solver that implements the dense linear algebra Cholesky factorization because of its integration as computing phase in many HPC applications. An important factor for our work is that the Cholesky factorization algorithm uses a triangular matrix divided into blocks, and four different tasks: `dpotrf` (Cholesky Factorization), `dgemm` (Matrix Multiplication), `dsyrk` (Symmetric rank-k update), and `dtrsm` (Triangular Matrix Equation Solver). The Cholesky factorization begins applying tasks on lower coordinates blocks and iteratively computes all blocks from all coordinates. From an optimization perspective, the data management can free lower coordinates blocks rapidly because they are no longer required in future iterations.

III. Related Work

Data management views are absent from some performance analysis tools available for StarPU applications, like Temanejo[7] and StarVZ [4], that we are extending in this work. Another tool is Vite [8] that has a classical Gantt chart for the memory nodes' events; however, it is modest and consider asynchronous transfers events as states. The problem is that multiple asynchronous transfers can occur, so the states are misleading. In all cited cases, the tools focus on the tasks scheduling and resources states.

The performance analysis of task-based applications considering the data management or memory interference is present in [9], where the authors presented a task-oriented approach. They analyze the impact of different schedulers on data reuse by tasks on the same resource. Studying the influence on cache misses and other metrics when the sequence of scheduled tasks changes. Data operations focusing in data reuse is also studied by [10]. Where the authors propose a metric called Kernel Reuse Distance (KRD). Also, the authors of [11] analyze the workload for general HPC applications, they state a series of metrics, including data cache, reuse and prefetch for characterizing the applications.

Comparing to the others, our approach is oriented by resources, analyzing the data flow in the heterogeneous environment. We work on a high-level view of the application, using the runtime decisions to explore possible problems. Instead of using low level metrics and comparing them with multiple executions, we focus on the behavior characterization of one execution. We provide some visualization elements that facilitate the performance analysis and enrich our perception of task-based applications running over heterogeneous platforms.

IV. Analyzing the Data Management

Understanding the application's memory flow could lead to optimizations and overall performance improvement. We present our methodology to review the memory manager behavior and memory blocks locality at different resources. The StarPU's data management module is responsible for all actions involving the application's memory; some examples are the data allocation on different accelerators, transfers between inner/inter-node resources, and determine when a memory block can be freed to reopen space for another one. All these actions operate over a specific memory handle that is absent from the original StarPU's trace data. For gathering all the necessary information needed for our performance analysis, we proposed the following extensions to it. We first include the events' memory identification on all events with some extra information to create correlations between activities and to understand the decisions behind it. Second, we add trace events on memory's coherence update function, since it can't be precisely inherited using the current available information. These new events can be used to compute the presence of memory blocks on each memory node.

One classical way to analyze space/time information is applying Gantt charts. We use it to inspect the memory nodes events and do some extensions to it. Visualization present in Figure 1 memory states plot. Each element on the Y axis is a memory node, associated to a node ram or accelerator. The X axis is the time in microseconds. In the example of Figure 1, memory nodes 1 and 2 are for each one of the two GPUs. Memory node 0 is omitted here since lack events during the executions. Each state has a color associated with the action taken. Also, it is possible to visualize the memory blocks coordinates of each handle in the center of the state, useful if a time frame is selected. An example is available on Figure 2, where the allocating actions are over the memory blocks of coordinates 9×10 or 9×15. On the left side of the visualization, a percentage of the most present state is shown.

We propose a blocks' residency over memory nodes visualization. We trace the coherency `MSI` states update and derive the presence of each block on all memory nodes. On Figure 3 we can observe this new visualization for the memory blocks of coordinates 13×8, 13×9, 13×10, 13×11 of the input matrix. For each block, the X axis is the time divided into time intervals of 10 seconds. This interval is sufficiently large for the visualization; yet, small enough to show the application behavior evolution. At each time interval, the Y axis shows the percentage of time that this memory block was on each memory node. Because each block can be present on multiple memory nodes, the maximum residency percentage on Y is bigger than 100%. The maximum percentage is the number of memory nodes times 100, as the memory block can be present on all memory nodes with the `shared` state.

V. Experiments

Experiments were conducted based on preliminary tests that strongly suggested that the Cholesky Factorization, present in the dense linear algebra solver Chameleon, had performance problems, for unknown reasons, when running on certain machines. We apply the methodology of Section IV in this real case scenario to check potential problems related to memory management. First, we use a test case that has idle times problems. The input parameters are block size of

960x960 and 60x60 tiles. The machine used was `tupi`, with a `Intel Xeon CPU E5-2620, 64GB DDR4` memory ram, `2x NVIDIA GeForce GTX 1080ti`.

The Figure 1 presents, from top to bottom, the plots: **(a)** Application Workers, **(b)** Mem Nodes, **(c)** StarPU Workers, **(d)** Ready Tasks, and **(e)** Used memory for this specific condition generated by StarVZ. In **(a)** and **(c)** the Y axes are the workers, in **(b)** the memory nodes, in **(d)** the number of ready tasks, and in **(e)** the overall memory utilization in MB. In all plots, the X axis is the time in milliseconds. Each state has a different color associated with its task or action. The red vertical line, manually added, crossing all plots presents the moment where the used memory reaches a plateau with its maximum value. At that moment, it is possible to check that the GPUs have a lot of idle times and the memory nodes are doing a lot of allocating actions until the end of the application. The GPU idle times of 33% and 32%, present on the left side of the plot **(a)**, are impairing the overall application performance.

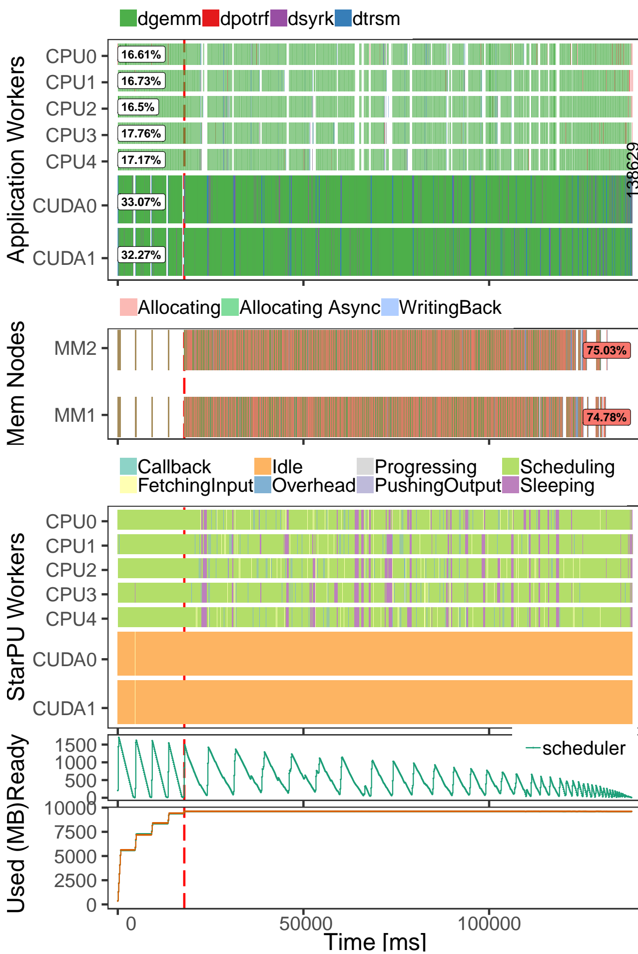


Fig. 1. Multiple Performance Analysis plots.

The possible correlation between idle times and allocation states led us to investigate memory nodes actions after the maximum memory utilization. We select a arbitrary time frame since their behavior is similar after the memory utilization peak. The Figure 2 gives a zoom on the Figure 1 **(b)** plot X axis, and shows for each action the associated memory block coordinates of the input matrix. There are many allocation's states occurring over the same memory blocks, which is a unexpected behavior (repeated allocations for the same memory block). Inspecting the StarPU source code we were able to determine that this happens if allocations fail. Using the GPU resources monitor, we checked that the GPUs are using all the memory. Our hypothesis at this point is that the devices don't have enough memory, but this shouldn't be a problem; as StarPU could free multiple memory blocks, especially those that would no longer be used by the Cholesky factorization.

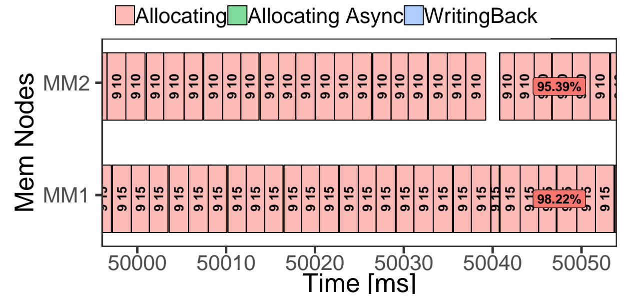


Fig. 2. Memory States on time frame of 50000 ms to 50050 ms.

We use the second visualization presented in Section IV, the blocks' residency, to understand this previously described behavior. First, we select early used blocks, with lower coordinates, that would be more appropriated cases for being free. Since, As previously discussed, the Cholesky algorithm only uses these tiles in the earlier iterations. On Figure 3, we can see that the blocks became present in all memory nodes at some point, block 13×8 at 50s for example, and remains there until the end of the execution. The presence in all memory nodes indicates that StarPU is deciding not to free these blocks.

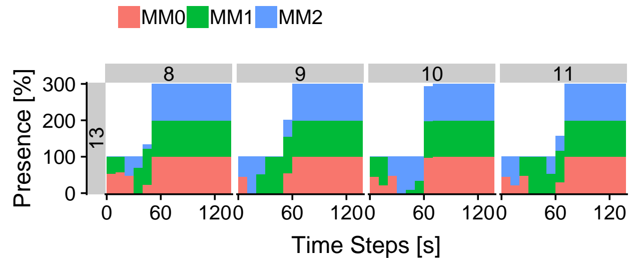


Fig. 3. Time presence (%) of the blocks (13, 8-11) in each memory node.

All these insights gave us enough information to inspect decisions directly. We use `gdb` to check StarPU's functions that free unused memory blocks. We found out that StarPU believed that it had free space on the GPUs. Also, we detected a huge difference when comparing the internal StarPU's used memory values to the ones given by the GPU resources monitor. We discovered that the CUDA function `cudaMalloc` could allocate more memory than the requested size. The function rounds the demanded memory to a device dependent page size. In the case of the `GTX 1080ti`, it is 2MB. It makes a block with 7200 KB to use 8192 KB; over-allocating

992 KB per block and 1800 MB per matrix (in the 60x60 blocks case). StarPU was wrongly calculating the used size on the resources and kept calling the expensive `cudaMalloc` function even with the GPU memory full. We then proposed a correction patch for StarPU, and compare the performance of the Cholesky application before and after it.

We executed 10 experiments for each configuration, to tackle experimental variability, using different matrices sizes and patch's versions. The input size distribution has more points around and after the memory limit when the real matrix size don't fit on GPU. Also, we use the following parameters: block size of 960x960, `DMDA` scheduler. The Figure 4 presents the performance comparison between StarPU's versions. The Y axis is the GFlops performance as reported by the Cholesky application with a 99% confidence interval. The X axis is the matrix width in cells. The gray line is the threshold where the matrix size fits on the GPU memory; considering the rounding behavior, number of blocks and the CUDA driver used memory. Two different StarPU versions are used. The red line is the **original** version (commit `be5815e`), and the blue line is our **corrected** version (commit `ca3afe9`). The **original** version has a performance drop after the memory threshold, failing from the relatively constant ~690 GFLOPs to lower values depending on the matrix size. However, the **corrected** version constantly keeps its performance on the ~690 GFLOPs mark. Demonstrating the effectiveness of our fix, keeping the program scalable as the input size increases.

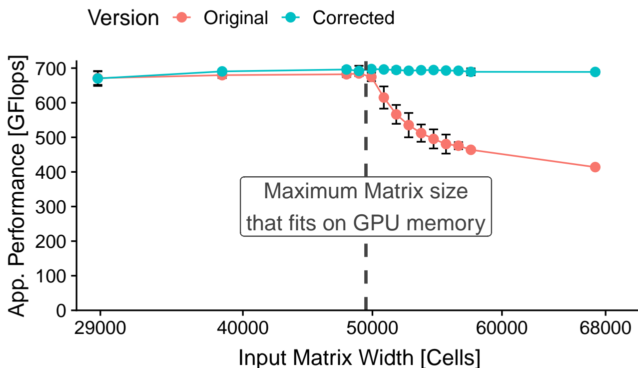


Fig. 4. Application performance (GFLOPs) before/after the patch.

VI. Conclusion

In this paper, we presented a visual performance analysis of heterogeneous task-based applications' memory management running over StarPU. We add new trace events on StarPU and use it in the new methodology incorporated into the StarVZ workflow. Since all methods rely on the runtime's features, any StarPU application can use this memory management analysis. The examination of memory management could lead to useful insights and possibly discovery of performance problems. We presented a real use case scenario using the dense linear algebra solver Chameleon. Using this methodology, we discover a performance problem (GPUs with high idle times) when the input matrix's size was higher than the GPUs' total memory.

The new visualizations allowed to verify that several slow allocations' states were occurring in the memory nodes; along with the unnecessary presence of memory blocks in limited memory resources. The problem was that StarPU computes the memory used by each resource only considering the size request for allocation; yet, in the case of CUDA GPUs, the `cudaMalloc` function can reserve more memory than the requested. In a case study, it led to a deviation of 1.8 GB between the real used memory and the runtime's registered, causing misleading decisions. Our proposed solution, to solve this StarPU's problem, resulted in a performance sustain independent of the input matrix size and if it fit in GPU's memory. For future work, we consider the analysis of the memory of other applications, trying to find optimizations on both the runtime and application.

Acknowledgements

We would like to thank Samuel Thibault and Luka Stanisic for the insights and the discussions about this work. We also thank these projects for supporting this investigation: FAPERGS GreenCloud (16/488-9), the FAPERGS MultiGPU (16/354-8), the CNPq 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, 1996.

[2] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IEEE Intl. Symposium on Parallel and Distributed Processing*, 2013.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Conc. and Comp.: Pract. and Exp., SI: Euro-Par 2009*, vol. 23, pp. 187–198, 2011.

[4] V. G. Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task-based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, 2018.

[5] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *16th International Conference on Parallel and Distributed Systems*, Shangai, China, Dec. 2010. [Online]. Available: https://hal.inria.fr/inria-00523937

[6] E. Agullo, G. Bosilca, B. Bramas, C. Castagnede, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, X. Lacoste, J. Langou, H. Ltaief, M. Messner, R. Namyst, P. Ramet, T. Takahashi, S. Thibault, S. Tomov, and I. Yamazaki, "Poster: Matrices over runtime systems at exascale," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, H. Wasserman, Ed., Nov 2012.

[7] R. Keller, S. Brinkmann, J. Gracia, and C. Niethammer, "Temanejo: Debugging of thread-based task-parallel programs in starss," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 131–137.

[8] C. Kevin, F. Mathieu, and J. Johnny., "Visual trace explorer (ViTE)."

[9] G. Ceballos, T. Grass, A. Hugo, and D. Black-Schaffer, "Analyzing performance variation of task schedulers with taskinsight," *Parallel Computing*, vol. 75, pp. 11 – 27, 2018.

[10] M. Pericàs, A. Amer, K. Taura, and S. Matsuoka, "Analysis of data reuse in task-parallel runtimes," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2014, pp. 73–87.

[11] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07. New York, NY, USA: ACM, 2007, pp. 73–82.

Parallel Workflow Support for StarVZ using Drake

Guilherme Rezende Alles, Lucas Mello Schnorr
Graduate Program in Computer Science (PPGC/UFRGS), Porto Alegre, Brazil

***Abstract*—This paper presents an approach to programming data analysis workflows using Directed Acyclic Graphs (DAGs) as the mean of separating computation stages. In this work, we analyze the programming structure of StarVZ – a framework written in the R programming language for data visualization - and rearrange it in order to form a graph that represents the data manipulation involved in the processing stage of the data analysis pipeline. The creation of the DAG is done with the help of Drake, an R package for managing workflows and usually applied in data science contexts. Our objective with this work is to increase the amount of data that can be analyzed with StarVZ, and we approach this problem by leveraging Drake features that can potentially speed-up a data analysis workflow, such as the parallel execution of independent tasks and the caching of results of previous computationally intensive steps (also known as memoizing). Even though we were able to explore these features, the results obtained are not satisfactory because of the way Drake implements communication between parallel processes.**

I. Introduction

Data science workflows are often referred to as a series of steps that can be represented as a pipeline. The most common steps in this pipeline are reading, manipulating (tidying), visualizing, and analyzing data, and data manipulation frameworks such as the Tidyverse [1], for the R programming language, enforce this pattern with its APIs. During a typical data science workflow, this pipeline is executed many times with incremental changes to the process (especially in the manipulation step), and this iterative development is important because it allows for the scientist to improve the quality of its work based on previous observations.

Despite its relevance, the process of constantly iterating over the same set of steps can add up to a considerable productivity overhead. The data analysis pipeline can take anywhere from a few seconds to several minutes (or even hours) to complete, depending on the complexity and the amount of data that it involves. Additionally, because these pipeline operations often change in small increments, most of the computation time is wasted executing redundant code that was exactly the same as the previous iteration.

As an alternative for data science pipelines, this paper's objective is to present the advantages of using a directed acyclic graph (DAG) to represent a data science workflow. We use a workflow manager, Drake [2], to create such graphs for a data analysis framework called StarVZ [3]. We also present how the visualization of the graph and features like parallelism support and memoization have the potential to both increase the understanding of the data flow in a data analysis process and also speed up the execution of these workflows. Additionally, by leveraging such features, we intend to tackle the problem of execution time on StarVZ's data analysis process [3], which would eventually allow for larger data sets to be studied in a reasonable time.

II. Background

A. Drake

Drake is a library written for the R programming language that strives for managing workflows in the form of directed acyclic graphs of tasks. Despite being used in the context of data science, drake can be used as a general purpose workflow manager.

Drake allows its users to describe a set of tasks and its dependencies, and it uses this information to generate the DAG that represents the workflow. The three key features offered by Drake are:

- a DAG visualization and profiling tool, which allows the user to graphically visualize the tasks that are executed and their execution time;
- parallelism support, which, in conjunction with the explicit tasks dependencies, allows for parallel execution of multiple R sessions that compute independent tasks;
- memoization, which caches the result of previously executed tasks to avoid unnecessary computations and speed-up future executions.

B. StarVZ

StarVZ is a data analysis framework written in R for the visualization of traces in high-performance computing applications. It works in a two-phase process, in which the first phase is responsible for filtering, manipulating and tidying the data collected from the traces and the second phase is responsible for processing the visualization of the data.

In this work, we foscused on working with the first phase of the StarVZ processing, which involves a considerable number of operations on multiple data frames. The original source code implements a batch of operations on relatively independent data sets, some of which are merged together at given points of the execution pipeline. This pattern of operations can be suitably ported to a DAG representation.

III. Implementation

In order to port StarVZ to a new version that supports a workflow manager, some changes to the source code were needed. These changes reflect the effort to make task dependencies explicit and to postpone operations that do involve merging data from multiple data frames.

The first effort consists of separating the step of reading data from that of data manipulation. This separation is necessary if we want to maximize the number of in-memory operations

in further manipulation steps, which can be slowed down considerably if data needs to be read from the disk when demanded.

The next effort was to separate and take note of all the transformations that took place in the data frames during the data processing step. This step is relatively complex when the source code has not been originally written with task separation in mind, which was the case for StarVZ. Ideally, a data science workflow can be written from the ground up with clear, distinct operations for each data set that is involved in the process, such that extracting a DAG that represents separate tasks and identifying each task's dependencies is trivial.

After identifying independent tasks in the workflow, we used the Drake package to create a plan. A Drake plan is a recipe containing a description of the tasks in the workflow, and each task is represented by an R function to be called. When creating a plan, Drake inspects all the function definitions (i.e. tasks) identifying their inputs, which are interpreted as task dependencies. The outputs of these functions are called targets. Because Drake (and R, in general), enforce the functional programming paradigm, global state is not supported as task dependencies. Figure 1 shows the DAG created by drake with the tasks and dependencies that were described in the Drake plan. In the DAG, nodes in purple are the initial resources needed by each target (inputs and manipulation functions) and nodes in black are the targets to be computed during the execution of the workflow.

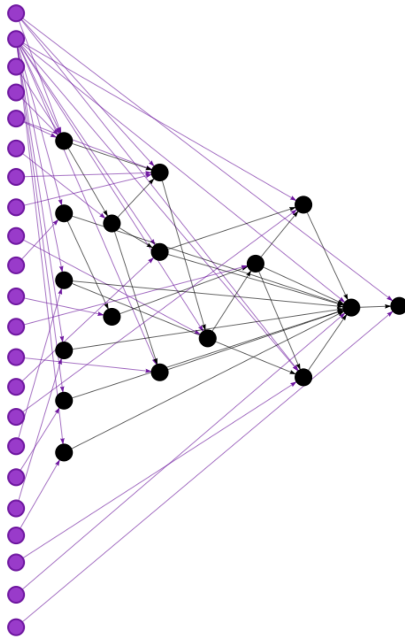


Fig. 1. Dependencies graph generated by Drake to execute the data manipulation of the first phase of StarVZ

As mentioned previously, Drake does not support the usage of global state as inputs for targets, meaning that functions should be pure (with no side effects) and every dependency should be explicitly defined. This allows for Drake to assume that, given a fixed set of inputs to a task, its output will always be the same. Consequently, Drake can cache the results of previous stages of the workflow to speed-up future executions by only calculating targets whose inputs have changed. Additionally, Figure 1 makes it explicit that every task in each column is independent from one another, meaning that they can be executed in parallel without concurrency drawbacks such as race conditions.

IV. Results

Even though Drake provides features that can potentially speedup code execution on data science pipelines, we could not observe the benefits of such features when porting the StarVZ code to a workflow.

The caching feature, for instance, saves the result of previous computations to disk in order to avoid recalculating the same steps repeatedly. However, because StarVZ's data frames are considerably large, writing them to disk in every stage of computation imposes a fairly heavy overhead in execution time.

With respect to parallel execution, Drake is limited by R's single-threaded implementation. In order to overcome this implementation decision, Drake starts multiple R sessions to compute parallel stages of the workflow. However, because these R sessions are separate processes, they are not able to share the same address space, and thus the data exchange between workers is done through the same caching system that writes the targets outputs to disk, once again imposing a fairly heavy overhead.

V. Conclusion and Future Work

In this study we explored the usage of Drake, a workflow manager for data science applications in R, to describe the data manipulation that take place in StarVZ. We conclude that using a DAG for workflow representation is helpful because it allows for the scientist to better understand the data flow of its application. Additionally, this representation also makes data dependencies explicit, and such information can be used to leverage parallel execution more easily. Caching of previous results is also possible, as long as the functions provided to the workflow manager are pure and there is no global state.

Unfortunately, StarVZ's data manipulation stage could not leverage the features of Drake that can positively impact performance. We believe that a combination of factors, such as large data frames being written to disk and the load imbalance between tasks that generate such large data frames are responsible for affecting the performance in this specific case.

As a solution to the previously mentioned drawbacks and as suggested future work, we can leverage the insights drawn by the DAG visualization of the workflow with respect to tasks dependencies to port StarVZ to a more robust and scalable data science framework, such as Apache Spark. In this case, Apache Spark can be used as a backend for the computations involved in StarVZ, and R frontends such as sparkR and sparklyr can be used as an interface.

REFERENCES

[1] H. Wickham, *tidyverse: Easily Install and Load the 'Tidyverse'*, 2017, r package version 1.2.1. [Online]. Available: https://CRAN.R-project.org/package=tidyverse

[2] W. Landau, "The drake r package: a pipeline toolkit for reproducibility and high-performance computing," vol. 3, p. 550, 01 2018.

[3] V. Garcia Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters," *Concurrency and Computation: Practice and Experience*, Apr. 2018. [Online]. Available: https://hal.inria.fr/hal-01616632

GROUPLB: Load Balancer Proposal to Reduce the Execution Time of Parallel Applications

Giovane da Rosa Lizot[1], Vinicius Manica Mastella[1], Pablo Jose Pavan[2], Edson Luiz Padoin[1]

Regional University of Northwest Rio Grande do Sul (UNIJUI) – Ijui, RS – Brazil
{giovane.lizot, vinicius.mastella, padoin}@unijui.edu.br

Federal University of Rio Grande do Sul (UFRGS) – Porto Alegre, RS – Brazil
pablo.pavan@inf.ufrgs.br

Abstract

In this article is presented the proposal of a new load balancer developed for the CHARM++ *programming model. The main purpose is to provide load balancing in order to reduce the migration of tasks, reducing the total execution time of the application. Initial test results demonstrate reductions in overall run time compared to other state of the art load balancers.*

1. Introduction

Currently, computational simulations are increasingly complex, continuously demanding more processing power from High Performance Computing (HPC) systems. When parallelized, they present excessive communication between tasks and unbalanced load, preventing the efficient use of its potential. Therefore, some cores can receive tasks with less load or remain idle while others execute the applications, causing, as a consequence, inefficiency in the use of the systems [9].

To solve such problem, load balancers have been proposed, aiming to increase the utilization of the potential of computational systems in processing level and execution time. In this context, many parallel applications that involve simulation with dynamic behaviors or calculations based on complex formulas have used such resources due to the imbalance of load and the amount of communications [10].

Many researches seek to improve the execution time of such computational systems. Thus, different research centers aim at the development of performance-enhancement techniques for scientific applications, improving the processing and execution time, and the second is the challenge, of decreasing the application execution time to be studied and proposed in this work.

Parallel computing has grown from few an processing units to systems with multiple processing units. It can be analyzed how the parallelization of tasks assists in the performance of processors through task migrations, in the same way that the parallelization of the applications exploited the processment competition in order to improve the efficiency in parallel processing.

A divide in groups as loads for control as migrations strategy contributes to more precise load balancing because task migration is a costly process for the system. GroupLB contributes by adapting better with the task of balancing processor loads without having to use more different BCs for this, dividing the loads according to their size. The rest of the paper is organized as follow. Section 2 positions our work against related work. Section 3 presents our proposed load balancer. Section 4 details the methodology used. Section 5 has the evaluation of the proposed load balancer. Conclusion and future work appears in Section 6.

2. Related Work

Parallel programming are developed to platforms with shared or distributed memory. Some scientific applications have regular designs that lead to well balanced load distributions, while others are more imbalanced due to the fact that they have tasks with different processing demands. So, different approaches can be used to deal with imbalanced workloads such as CHARM++, Kaapi and UPC [8]. These runtime systems usually focus on reducing the execution time by applying clever task orchestration strategies and by improving the load distribution among cores.

Different approaches have been employed to make load balancing and reduced runtime of applications. Among them, centralized [7, 3] and distributed [4, 2, 11]. strategies are more employed. However, new hierarchical ap-

proaches are being proposed reduce the overhead these approaches [11].

Once that in each one, a different amount of cores are selected to make decision of load balancing where tasks are migrated to different cores of the parallel systems.

Load balancers like GREEDYLB and REFINELB were implemented and are available in the CHARM++ programming environment.REFINELB is an approach based on load refinement. This load balancer moves tasks of the most overloaded cores to the least loaded ones until reaching an average[1]. The REFINELB moves objects from the most overloaded processors to the least loaded, aiming to reach an average, with the number of objects migrated limited [6].

Balancer AVERAGELB [1] and SMARTLB [5], uses acentralized approach in its strategy, which takes decisions in a single process.

Differently, GREEDYLB uses a greedy approach algorithm that implementing practice of combinatorial optimization. Its algorithm, to migrate heavy objects to the processor with less load. This is repeated until the load of all processors reaches a close proximity to the average load.

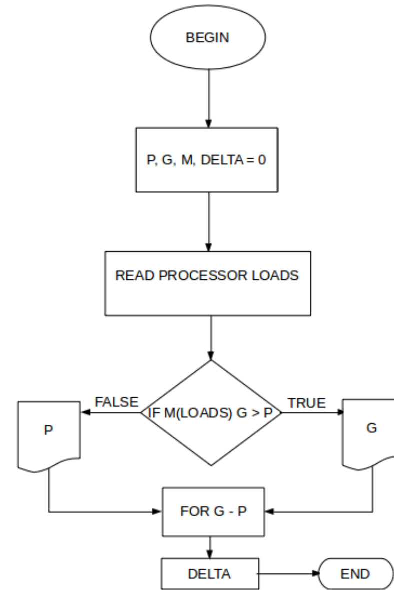


Figure 1. Algorithm Strategy

3. GROUPLB Load Balancer

Our proposed strategy, called GROUPLB, take into account the AVERAGELB [1] and SMARTLB [5] mechanisms to make decisions. Similar to AVERAGELB, it also uses a centralized approach in its strategy, which takes decisions in a single process. The strategy of the algorithm takes into account the arithmetic mean of each processor, calculating its loads, in order to reduce the number of migrations, seeking a balance between the loads [6].

The algorithms collect system and application information at run-time to dynamically use them in load balancing decision,to reduce execution time. Our strategy divides the processes according to the computational loads into three groups, called *Small* (P), *Medium* (M) and *Large* (G), which respectively store the load processes considered relatively small, medium and large.

In this way, when the load balancer is applied, the loads of the cores is verified, and, from this information, the difference of loads between them will be analyzed. Subsequently, the tasks with hight loads are divided into the respective groups. In the algorithm there will be the control variables, where the variable ”less” represents the sum of the loads in the P array, the variable ”bigger” the sum of the loads in the G array, and the variable ”delta” the difference between the variables G and P. The groups P, M, G are created and tasks are placed there based on their initial value and delta receives the difference of the loads of group M.

Figure 1 shows the run of the algorithm, how is the case of the random as load flow non processor and how divide with agreement with their dimensions, when compared to the group M, are as large will be shifted to the group G, if not to the group P, and the difference of them is displaced to the delta.

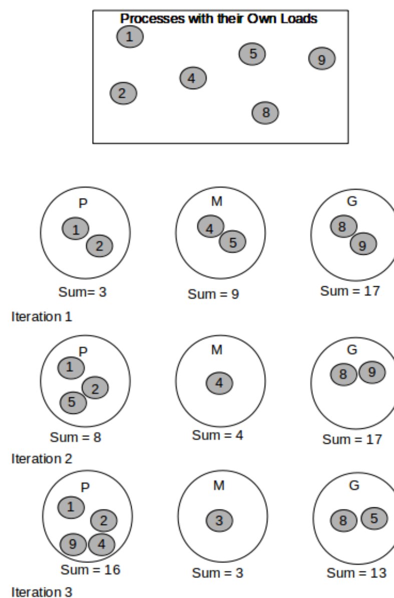


Figure 2. Load Balancer Strategy

Figure 2 shows the adopted iteration strategy. In it, it can be seen the division of the loads in the *Small*, *Medium* and *Large* groups, as well as to analyze how the difference of the

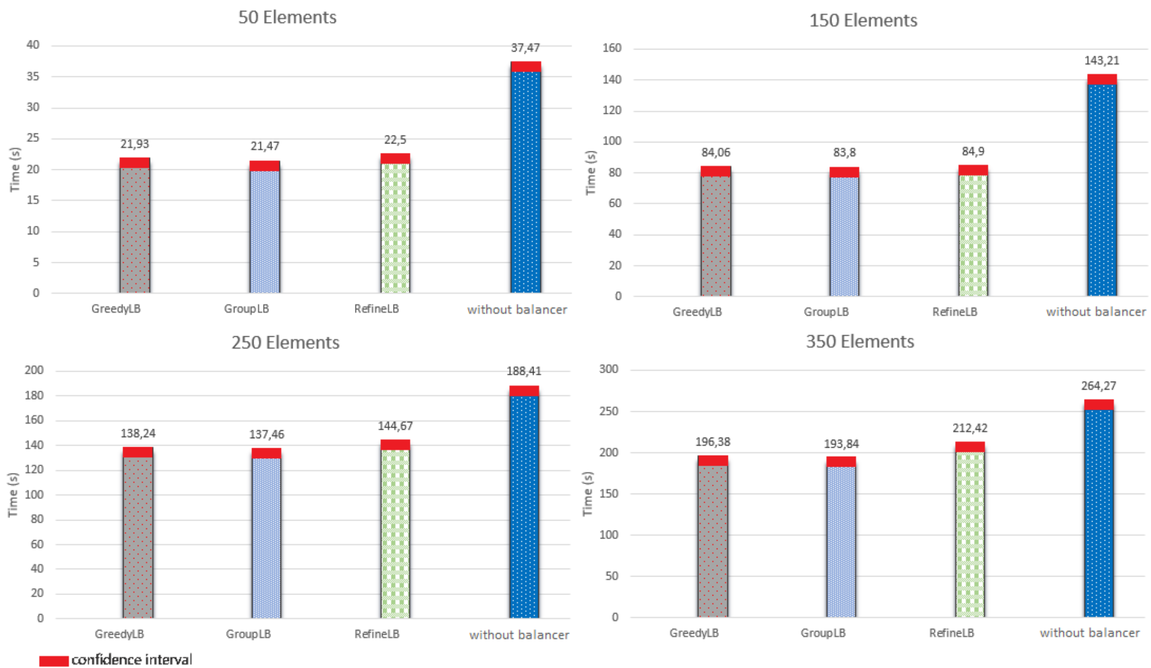


Figure 3. Load Balancer Strategy

Small and Large groups, the "delta" variable, decreases with each iteration. In the end the loads are balanced according to the adopted strategy, reducing the unbalance, the unnecessary migrations and the runtime of the application, the values are compared to the CHARM++ standard without load balancer. The results follow a normal distribution because related and inspired works are developed on the CHARM++ platform.

To implement the proposed load balancer was selected the CHARM++ environment. This environment is supported by several platforms, allowing the programs developed in this model to run in both shared and distributed memory environments.

4. Methodology and Evaluation

In this section, we present our methodology to validate the load balancer proposal (GROUPLB), and we present ours evaluates about this load balancer comparing it with other state-of-the-art balancers.

To validate our proposal strategy was used an equipment with Intel Core i7M processor wich 8 physical cores. For the tests, the Linux operating system Ubuntu 16.04 with kernel version 4.4.33-1 was used. The version of Charm++ used was 6.5.1 and the g++ compiler in version 6.2.1.

Each of the tests performed in this work was repeated 10 times, the results follow a normal distribution to achieve a relative error less than 5% and 95% statistical confidence for a Student's t-distribution.

In order to evaluate the performance of proposed load balancer, it was applied in the execution of the lb_test benchmark and compared to the other two balancers, GREEDYLB and REFINELB, which are provided by the CHARM++ programming environment. Tests were performed with 50, 150, 50 and 350 tasks, these being with computational loads varying between 1500ms and 150000ms. Synchronizations for balance call load was defined every 10 iterations.

In Figure 3 is shown the results of the tests performed with the lb_test benchmark in the platform described.

In the performed tests, the GROUPLB load balancer achieved the best performance with the benchmark tested. Using 50 tasks, the time was reduced by 37.47% when compared to the same test without any load balancer. In comparison to the REFINELB, it was lower by 4.58% and 2.09% compared to GREEDYLB. With 150 tasks, also presented a reduction compared to the test without any balancer.

With a percentage of 41.50%, though compared to the REFINELB, obtained a percentage of 1.30%, and finally, obtained 0.30% in relation to the GREEDYLB. Thus, in the tests with 250 tasks,GROUPLB we also obtained gains in the results.

It reduced the time by 27.05% in relation to the execution without any balancer, 4.98% in relation to the REFINELB balancer and 0.56% in relation to the GREEDYLB.As well as 350 task, GREEDYLB also made gains. It reduced the

time by 24.65% in relation to the execution without any balancer, 8.75% in relation to the REFINELB balancer and 1.30% in relation to the GREEDYLB balancer.

5. Conclusions

This paper presents a new load balancer proposal, which was developed for the CHARM++ programming model. Its objective is reduce the total execution time of the unbalanced applications, as well as reducing the number of process migrations. In the tests performed with the lb_test benchmark, our proposed strategy present results superior to REFINELB and GREEDYLB load balancers.

This benchmark was chosen because it is easily configurable to present different levels of unbalance of load, allowing the computational load of each task be configured in different load patterns, both of irregularity, of communication, of being disprovided by the program environment itself. As future works, it is intended to perform tests in parallel systems, using other benchmarks and real problems of scientific computation.

6. Acknowledgment

This work has been granted by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and PETROBRAS company. It was also supported by Intel Corporation under the PIBIC UNIJUI Project.

References

[1] G. Arruda, E. L. Padoin, L. L. Pilla, P. O. A. Navaux, and J.-F. Mehaut. Proposta de balanceamento de carga para reduo de migrao de processos em ambientes multiprogramados. In *XVI Simpsio de Sistemas Computacionais (WSCAD-WIC)*, pages 1–8, 2015.

[2] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *Parallel and Distributed Systems, IEEE Transactions on*, 16(4):289–299, April 2005.

[3] A. Bhatelé, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kalé. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings...*, pages 1–12. International Symposium on Parallel and Distributed Processing (IPDPS), IEEE, April 2008.

[4] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[5] V. R. S. dos Santos, E. L. Padoin, P. O. A. Navaux, and J.-F. Mhaut. Smartlb: Proposta de um balanceador de carga para reduo de tempo de execuo de aplicaes em ambientes paralelos. In *Congresso da Sociedade Brasileira de Computao (CSBC) - Workshop em Desempenho de Sistemas Computacionais e de Comunicao (WPERFORMANCE)*, jul 2018.

[6] G. Freytag, G. Arruda, R. S. M. Martins, and E. L. Padoin. Anlise de desempenho da paralelizao do problema de caixeiro viajante. In *XV Escola Regional de Alto Desempenho (ERAD)*, pages 1–4, Gramado, RS, 2015. SBC.

[7] S. Ichikawa and S. Yamashita. Static load balancing of parallel pde solver for distributed computing environment. In *Proceedings...*, pages 399–405. International Conf. Parallel and Distributed Computing Systems (PDCS), ISCA Press, 2000.

[8] E. Padoin, M. Castro, L. Pilla, P. Navaux, and J.-F. Mehaut. Saving energy by exploiting residual imbalances on iterative applications. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, Dec 2014.

[9] E. L. Padoin, M. Castro, L. L. Pilla, P. O. Navaux, and J.-F. Méhaut. Saving energy by exploiting residual imbalances on iterative applications. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[10] L. L. Pilla and E. Meneses. Programao paralela em charm++. pages 1–20, 2015.

[11] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé. Periodic hierarchical load balancing for large supercomputers. *International Journal of High Performance Computing Applications*, 25(4):371–385, 2011.

List of Authors

——/ A /——

Alles, Guilherme . . . 23

——/ B /——

Bez, Jean . . . 9, 13
Boito, Francieli . . . 9, 13

——/ G /——

Geyer, Claudio . . . 1, 5
Girelli, Valéria . . . 9, 13

——/ L /——

Legrand, Arnaud . . . 17
Lizott, Giovane . . . 27

——/ M /——

Mastella, Vinicius . . . 27
Matteussi, Kassiano . . . 5
Moreira, Francis . . . 7

——/ N /——

Navaux, Philippe . . . 7, 9, 13
Nesi, Lucas . . . 19

——/ O /——

Oliveira, Daniel . . . 7

——/ P /——

Padoin, Edson . . . 27
Pavan, Pablo . . . 9, 13, 27
Pinto, Vinícius . . . 17

——/ R /——

Rech, Paolo . . . 7

——/ S /——

Santos, Lucas . . . 1
Schnorr, Lucas . . . 17, 19, 23