Gonzalo P. Rodrigo Álvarez

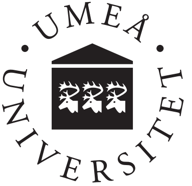
Gonzalo P. Rodrigo Álvarez

HPC Scheduling in a Brave New World

2017

HPC Scheduling in a Brave New World

Gonzalo P. Rodrigo Álvarez



PhD Thesis, March 2017
Department of Computing Science
Umeå University
Sweden

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

gonzalo@cs.umu.se

Copyright © 2017 by the authors
Except Paper II, © Springer-Verlag, 2013

ISBN 978-91-7601-693-0
ISSN 0348-0542
UMINF 17.05

Printed by Print & Media, Umeå University, 2017

Abstract

Many breakthroughs in scientific and industrial research are supported by simulations and calculations performed on high performance computing (HPC) systems. These systems typically consist of uniform, largely parallel compute resources and high bandwidth concurrent file systems interconnected by low latency synchronous networks. HPC systems are managed by batch schedulers that order the execution of application jobs to maximize utilization while steering turnaround time.

In the past, demands for greater capacity were met by building more powerful systems with more compute nodes, greater transistor densities, and higher processor operating frequencies. Unfortunately, the scope for further increases in processor frequency is restricted by the limitations of semiconductor technology. Instead, parallelism within processors and in numbers of compute nodes is increasing, while the capacity of single processing units remains unchanged. In addition, HPC systems' memory and I/O hierarchies are becoming deeper and more complex to keep up with the systems' processing power. HPC applications are also changing: the need to analyze large data sets and simulation results is increasing the importance of data processing and data-intensive applications. Moreover, composition of applications through workflows within HPC centers is becoming increasingly important.

This thesis addresses the HPC scheduling challenges created by such new systems and applications. It begins with a detailed analysis of the evolution of the workloads of three reference HPC systems at the National Energy Research Supercomputing Center (NERSC), with a focus on job heterogeneity and scheduler performance. This is followed by an analysis and improvement of a fairshare prioritization mechanism for HPC schedulers. The thesis then surveys the current state of the art and expected near-future developments in HPC hardware and applications, and identifies unaddressed scheduling challenges that they will introduce. These challenges include application diversity and issues with workflow scheduling or the scheduling of I/O resources to support applications. Next, a cloud-inspired HPC scheduling model is presented that can accommodate application diversity, takes advantage of malleable applications, and enables short wait times for applications. Finally, to support ongoing scheduling research, an open source scheduling simulation framework is proposed that allows new scheduling algorithms to be implemented and evaluated in a production scheduler using workloads modeled on those of a real system. The thesis concludes with the presentation of a workflow scheduling algorithm to minimize workflows' turnaround time without over-allocating resources.

Sammanfattning på svenska

Många genombrott i vetenskaplig och industriell forskning stöds av simuleringar och beräkningar på högpresterande datorsystem, så kallade HPC-system. Traditionellt består dessa av en stor mängd parallella och homogena datorresurser som är sammankopplade med ett synkront nätverk med hög bandbredd och låg latens. Resurserna i HPC-systemet hanteras av en schemaläggare som planerar var och när olika applikationer ska exekveras för att minimera användarnas väntetider och maximera resursnyttjandet.

Till för några år sedan skedde den normala utvecklingen av alltmer kraftfulla HPC-system genom att man ökade antalet noder i det parallella systemet samt ökade transistordensiteten och klockfrekvensen i processorerna. På senare tid kan man på grund av halvledartekniska begränsningar inte längre dra fördel av ökade klockfrekvenser. Istället ökas kapaciteten genom ökad parallellitet i processorer och i form av ökat antal datornoder. En annan del av utvecklingen är att HPC-systemens minnes- och I/O-hierarkier blir allt djupare och mer komplexa för att hålla jämna steg med utvecklingen av processorkraft. De applikationer som körs på HPC-systemen förändras också: behovet att analysera stora datamängder och simuleringsresultat ökar betydelsen av datahantering och dataintensiva applikationer. Därtill har behovet av tillämpningar i form av långa arbetsflöden istället för enskilda jobb också ökat.

Denna avhandling studerar hur denna utveckling skapar nya utmaningar för framtidens HPC-systems schemaläggningssystem. Studien inleds med en detaljerad analys av utvecklingen av tre referenssystem vid National Energy Research Supercomputing Center (NERSC) vid Lawrence Berkeley Lab, med fokus på jobbens heterogenitet och schemaläggarnas prestanda. Detta följs av en analys och förbättring av en prioriteringsmekanism (så kallad fairshare) för schemaläggare. Därefter presenteras en studie av nuläge och förväntad framtida utveckling vad gäller HPC-hårdvara och -applikationer. Studien presenterar ett antal viktiga utmaningar för framtida schemaläggare, som t.ex. applikationernas ökade heterogeneitet, behovet av att effektivt schemalägga arbetsflöden samt schedulering av I/O-resurser. Därefter presenteras en modell för schemaläggning för HPC-system som utvecklats med inspiration från schemaläggning för datormoln. Avhandlingen presenterar också en simulator, tillgänglig som öppen källkod, för forskning kring schemaläggning. Simulatorn gör det möjligt att studera nya algoritmer för schemaläggning i ett system av produktionskaraktär. Vi avslutar med att presentera en algoritm för schemaläggning av arbetsflöden med målsättning att minimera användarnas väntetid och minimera resursutnyttjandet.

Preface

This thesis contains an introduction that summarizes key concepts that the author considers useful to understand before reading research articles about HPC scheduling, including the definition of and need for high performance computing, basic batch scheduling, and the characteristics of present and future HPC systems together with the associated scheduling challenges. The thesis includes the following papers:

Paper I — Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards Understanding HPC Users and Systems: A NERSC Case Study. *Submitted*, 2017. This paper is an extended, joint version of papers IX and X.

Paper II — Gonzalo P. Rodrigo, Per-Olov Östberg, and Erik Elmroth. Priority Operators for Fairshare Scheduling. In *Proceedings of the 18th Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 70-89, Springer International Publishing, 2014.

Paper III — Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth, and Lavanya Ramakrishnan. A2L2: An Application Aware Flexible HPC Scheduling Model for Low-Latency Allocation. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC'15)*, pp. 11-19, ACM, 2015.

Paper IV — Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. ScSF: A Scheduling Simulation Framework. In *Proceedings of the 21th Workshop on Job Scheduling Strategies for Parallel Processing*, Accepted, Springer International Publishing, 2017.

Paper V — Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. Enabling workflow aware scheduling on HPC systems. *Submitted*, 2017.

In addition, the following publications were produced during the author's PhD studies but are not included in the thesis:

Paper VI Gonzalo P. Rodrigo. Proof of compliance for the relative operator on the proportional distribution of unused share in an ordering fairshare system. *Technical report UMINF-14.14*, Umeå University, 2014

Paper VII Gonzalo P. Rodrigo. Establishing the equivalence between operators: theorem to establish a sufficient condition for two operators to produce the same ordering in a Fairshare prioritization system. *Technical report UMINF-14.15*, Umeå University, 2014

Paper VIII Gonzalo P. Rodrigo. Theoretical analysis of a workflow aware scheduling algorithm. *Technical report UMINF-17.06*, Umeå University, 2017

Paper IX Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems. In *Proceedings of the 24th International ACM Symposium on High-Performance Distributed Computing (HPDC'15)*, ACM 2015

Paper X Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16)*, IEEE/ACM 2016

This material is based upon work supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) and the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE, by the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS), the European Union's Framework Programme Horizon 2020 under grant agreement 732667 (RECAP), and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

Acknowledgements

”Four years or work,
four years of hardship,
four years on a road towards enlightenment and joy.
The road was sometimes steep,
the road was sometimes deep,
but the road was never lonely, as many people always helped me.
In my falls, with patience, they consoled me,
in my struggles, with energy, they pushed me,
harsh and gently they brought me, to the place I wanted to be...” [1]

First, I would like to thank **Lavanya Ramakrishnan**, my advisor at the Berkeley Lab. We met in one of the most challenging moments of my PhD and, under her supervision and the Californian sun, we managed to steer my work towards the thesis that now sits in your hands (or on your screen). Always patient and insightful, she was (and is) a great advisor, colleague, and friend.

Second, but in no way smaller, comes my gratitude to **Erik Elmroth**, my advisor at Umeå University and suspected shareholder of Lottas Krog. Apart from patiently understanding my unconventional PhD program, his scientific, and personal support were fundamental in these years. He is probably the busiest person I have ever met, still, he always had time for matters that were important.

I also would like to thank **P-O Östberg**, co-advisor at Umeå University, who helped me settle down in Umeå, start a new life, and improve my skiing technique (the hard way).

To the *pecan cheesecake club* members: **Mina**, patient ex-office mate, confessor, and victim of my devil’s advocacy; **Jakub**, ex-house mate, PhD brother, and comrade on the same battles; and **Cristian**, source of wisdom, cardio partner-in-crime, and psyco-canoeing victim. Thanks for all those great conversations about science, life, “moment soaking”, and everything else.

I would like to express my gratitude to the rest of the distributed systems group at Umeå University: **Abel** (the HPC torch is yours now), **Ahmed** (we didn’t have enough coffees together), **Amardeep** (first room-mate), **Chanh** (so brief!), **Daniel** (fairshare advisor), **Ewnetu** (smartest coworker), **Francisco** (unexpected good advise), **Johan** (always on target), **Lars** (stuff you should know), **Luis** (a man with a red hat), **Muyi** (anomalies for all), **Peter**

[1] My apologies for this poetic crime.

(the stuff that he knows), **Petter** (I finally got a flat!), **Selome** (always power-aware), **Simon** (the Openstack master!), and **Viali** (unexpected owner of a Quebec driving license). And also to other people at the department who have been great company these years.

I should thank the members of the Data Science and Technology department (and surroundings) at the Berkeley Lab who were great co-workers, friends, donut dealers, Friday pizza-goers, and running mates during these years: **Abdelrahman Elbashandy, Dan Gunter, Deb Agarwal, Devarshi Goshal, Eugen Feller, Gilberto Pastorello, Gunther Weber, Hamdy Elgammal, Katie Antypas, Megha Sandesh, Richard Gerber, Paolo Calafiura, Sarah Poon, Shreyas Cholia, Sophia Passadis, Val Hendrix, Wenqin Chen,** and **You-Wei Cheah**.

During these years, I worked in two projects that were not strictly thesis related and I would like to thank: **Stephen Bailey**, for teaching me the true value of open-source work and showing me a different side of the Berkeley Lab; and **John Wilkes** for making me a better engineer through the fear (and joy) of Google's code-review system and the Oxford comma.

For being patient, responsive, and for helping me to harness the compute power that made this work possible, I have to specially thank the systems' gurus **Tomas Forsman** (UmU) and **Keith Beattie** (LBNL).

To my parents, **Gonzalo** and **Nieves**, whose support in the last 37 years gave a personal, deeper meaning to the quote *"If I have seen further, it is by standing on the shoulders of giants"*. And, finally, to **Katherine**, who made these last three years worth it.

¡Gracias a todos!

Contents

1 Introduction **1**

2 High performance computing **5**
- 2.1 High performance applications 5
- 2.2 HPC systems 7
- 2.3 HPC scheduling 10
  - 2.3.1 Job submission and priority 11
  - 2.3.2 Job scheduling 11
  - 2.3.3 Placement, accounting, and other functions 12

3 New scheduling challenges **13**
- 3.1 Present and future workloads 13
- 3.2 Future systems 16
  - 3.2.1 Dennard scaling break and processor parallelism 16
  - 3.2.2 The memory gap 17
  - 3.2.3 I/O gap 18
  - 3.2.4 Hardware heterogeneity 18
- 3.3 Scheduling research 19

4 Contributions **21**
- 4.1 Paper I: Characteristics and trends in HPC workloads 21
- 4.2 Paper II: Prioritization in batch schedulers, Fairshare 22
- 4.3 Paper III: A survey of current scheduling challenges and a position model to ease them 23
- 4.4 Paper IV: Building tools for scheduling research 23
- 4.5 Paper V: Workload-aware scheduling in HPC systems 24

Paper I **33**

Paper II **53**

Paper III **75**

Paper IV **87**

Paper V **109**

Chapter 1

Introduction

High performance computing (HPC) systems are designed to support the concurrent execution of performance-critical and large-scale applications at the minimum possible cost in terms of initial investment, operating costs, and energy consumption [20]. Research in scientific fields such as high-energy physics, geophysics, climate study, and bioinformatics relies on HPC systems to perform complex large-scale experiments [9]. HPC systems are managed by HPC batch schedulers, which arrange the concurrent execution of applications (jobs) to maximize utilization while offering reasonable turnaround times.

Classical HPC systems were uniform, largely parallel, and built around low latency synchronous interconnects [11]. To meet the increasing demands of computation-based science, compute capacity was increased by using processors with higher operating frequencies. However, current semiconductor technologies do not support further frequency increases, so new design strategies are required [26]. Currently, the parallelism within compute nodes is increasing but individual processing units are not becoming more powerful. Also, systems are becoming more heterogeneous and incorporating specialized resources tailored to the execution of specific application types [11]. Despite this growing raw processing capacity, current I/O and memory technologies cannot deliver matching performance improvements, creating a widening capability gap. As a consequence, HPC systems' memory and I/O hierarchies are becoming deeper and more complex.

Changes in hardware and architecture present new challenges when making scheduling decisions. For instance, the heterogeneity of modern systems' resources increases the complexity of placement and scheduling decisions: the scheduler must infer a resource allocation that will run an application efficiently while avoiding resource fragmentation that would reduce utilization. In addition, new I/O and memory systems include resources that are managed by the scheduler, such as burst buffers - high bandwidth, low access-time solid-state drives that act as caches to boost file system performance [36]. Their allocation is requested by users but the scheduler controls their assignment and coordin-

ates their use with that of the compute resources, increasing the dimensionality of the scheduling problem.

HPC applications are also changing. Classical HPC workloads were dominated by large, parallel, and tightly coupled jobs. However, data processing and high throughput applications, whose performance models differ from those of tightly coupled jobs, are more prevalent in scientific workloads [21]. Also, some applications are based on the composition of sub-applications (tasks) with different performance and resource requirements. These applications are usually structured as workflows whose composition derives from the control or data dependencies of their tasks. HPC schedulers support the execution of these increasingly common applications, but their scheduling decisions are typically not optimized for such uses. For example, workflows may suffer from long turnaround times because most schedulers cannot recognize their structure, leading to long intermediate wait times for their tasks.

This thesis proposes models and solutions for current and future scheduling challenges. The first step in solving any scheduling challenge is to understand and characterize current systems and their workloads to identify trends in their evolution. To this end, the first stage of the work presented in this thesis involved analyzing the workloads of three HPC systems at the National Energy Research Supercomputer Center (NERSC) using a novel workload analysis method that incorporates a characterization of job diversity. Many differentiated groups of jobs were detected in the workloads, indicating that their job geometry (in terms of requested resources and runtime) was very heterogeneous. Job heterogeneity also affected the predictability of job wait times. Under ideal conditions, a larger job (i.e. one requiring a greater allocation of resources) of a given priority (i.e. importance) would be expected to wait for longer than a smaller job of the same priority. Also, for a given resource allocation, a job with a higher priority should wait less than one with lower priority. The job geometry heterogeneity within priority groups in the studied workloads produced wait times that did not match this expected behavior.

Next, the mechanisms used by current HPC schedulers were analyzed. Specifically, a deep analysis of the fairshare prioritization model, common in HPC systems, was performed. Priority systems steer the wait time of jobs according to administrative policies, and this mechanism allows priority models to be expressed as organization hierarchy trees. This analysis showed how individual system rules enforce overall system behavior and add new mechanisms that improve its behavior.

The next step involved surveying new systems and applications, and identifying the scheduling challenges they present. This survey was complemented by an analysis of existing cloud scheduling solutions that address similar problems. Based on these investigations, a two-level cloud-based application-aware scheduling model was proposed. This model uses one scheduler per application type, and thus avoids the problem of comparing applications whose performance metrics have different semantics (for example, wait times have different impacts on regular jobs and workflow jobs). It also includes mechanisms to

enable dynamic scaling of malleable applications, whose resource usage can be changed during runtime to increase utilization. In addition, it supports real time jobs (jobs that require short wait times) by enabling the schedulers to borrow resources for their execution.

Unfortunately, a lack of suitable scheduling research tools made it difficult to implement or evaluate this model. A new open source *Scheduling Simulation Framework*, ScSF, was therefore developed. Built around a real instance of the Slurm scheduler (the most popular scheduler among the systems on the Top500 list), the framework includes modules for modeling systems, generating workloads based on the models, running simulations, and analyzing their results. It also includes orchestration tools to automate the concurrent execution of experiments over distributed resources. This tool was very useful in the work presented herein, and will be useful in any future HPC scheduling research involving simulation.

To conclude, this thesis describes WoAS, a *Workflow Aware Scheduling* technique for efficient workflow scheduling. Workflows in HPC systems are often run as chained jobs or pilot jobs. Chained jobs are groups of jobs interconnected by dependencies such that no job can start until all those that precede it in the workflow are complete. Pilot jobs allocate the workflow the maximum resources it will need at any point in its entire runtime, leaving resources idle at some stages. WoAS instead proposes submitting workflows as pilot jobs that include manifests describing the workflow's structure. It also modifies the way in which the scheduler's waiting queue exposes jobs to the the rest of the scheduler, keeping them as *pilot jobs*, when handled by the prioritization engine, and transforming them in *chained jobs*, when handled by scheduling algorithms (first come first serve and backfilling). WoAS was implemented as an open-source project in Slurm and evaluated in ScSF. Experiments showed that with WoAS workflow turnaround times are significantly shorter than under the chained jobs approach, giving speedups of up to 3.75x with no over-allocation of resources or significant effects on the turnaround time of non-workflow jobs.

The rest of the thesis is structured as follows. Chapter 1 provides an overview of High Performance Computing as well as HPC systems and schedulers. Current and new scheduling challenges are described in Chapter 3. The contributions of this thesis are summarized in Chapter 4. Finally, the five papers that describe the thesis work in full detail are presented.

Chapter 2

High performance computing

This chapter introduces basic concepts in High Performance Computing (HPC): its purpose, applications, systems, and scheduling. On the basis of previous work [20] and the author's own observations, HPC can be defined as follows: *A computing system should be considered high performance if it supports the execution of large-scale, performance-oriented applications, at the smallest possible cost, with the shortest possible runtime, within some time constraint.* Each part of the definition imposes certain requirements, constraints, and characteristics on HPC systems and their schedulers. These are discussed in the following sections.

2.1 High performance applications

High performance computing supports the execution of applications used in fields such as science, engineering [13], and finance [12]. Although all these fields employ similar systems and computing models, this thesis focuses on scientific applications because of the author's knowledge base and research interests. However, the results presented herein should be equally applicable to HPC for engineering and finance.

Put simply, scientific applications are applications that support the scientific process by enabling researchers to perform simulations, analyze data, or apply numerical methods to solve large computational problems. These applications are executed on HPC systems as batch jobs and have diverse characteristics [53]. The remainder of this sections describes the application characteristics with the greatest impact on the design and operation of HPC systems.

Large tightly-coupled parallel applications: Some scientific applications run many parallel processes with synchronized execution steps. Their complexity is illustrated by a classical tightly-coupled application: atmospheric

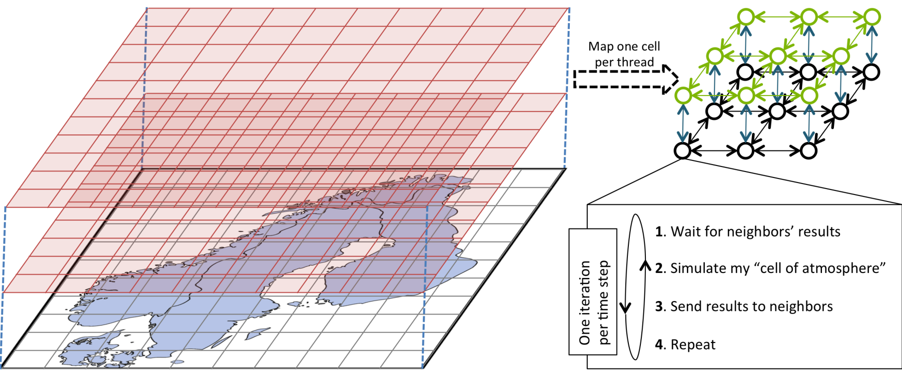


Figure 2.1: Spacial mapping of the atmosphere onto computing threads in an atmospheric simulation.

simulation [41]. Figure 2.1 depicts a computational approach used to predict the weather in Scandinavia. First, the space over Scandinavia is divided into a three dimensional grid and each cell is assigned to a single CPU core in an HPC system. Each core executes a loop to iteratively simulate the atmosphere in the corresponding cell during one time step. After the simulation of each time step, the results for the cell are propagated to the neighboring cells (and thus the corresponding cores) and used as inputs for the next iterative loop. Because each loop iteration depends on the previous iteration results from multiple cells, the communication step between cells becomes the application's repeating synchronization point. Under this execution model, a higher communication latency between processes implies a longer runtime per simulation step, a longer overall runtime, and thus worse performance and efficiency of resource use. Moreover, if any of the parallel processes run more slowly than the others, the need for synchronization delays the execution of the all processes.

Some specific hardware and software requirements must be met to achieve good performance under a synchronization model of this kind [7]. First, *the compute hardware must be homogeneous* to ensure that the execution time for a given piece of code is identical in all the system's CPU cores. Also, the system must employ a *low-jitter and low-latency interconnect* to minimize the latency and variability of the communication phase. Moreover, the *system's resources must be large enough* to run the applications. For example, the resolution of the weather simulation discussed above depends on the number of concurrent processes: a system with more CPU cores permits the execution of more concurrent processes, making it possible to perform simulations that use greater numbers of smaller cubes and thus yield more precise weather predictions. Finally, a simple *low latency communications framework* or message passing interface (MPI [49]), is required to ease the development and deployment of applications that require communication and synchronization between processes.

Data processing and high-throughput applications: High performance computing also supports scientific applications for analyzing large data sets produced by real life experiments or simulations [40]. Such applications do not adhere to the tightly-coupled model: while they may be parallel, their execution threads are not tightly synchronized, and their resource allocation may be malleable, i.e. they can tolerate dynamic resource allocation during runtime. However, such applications are typically I/O and memory bound, and require *low latency and high bandwidth memory and I/O* to achieve good performance. This requirement is amplified by the concurrent execution of applications on HPC systems: I/O requirements aggregate, multiplying the peak I/O bandwidth the system must provide.

To satisfy these memory performance requirements, the compute nodes of high performance systems have fast memory and large caches. The I/O requirements are satisfied by using large parallel file systems (PFS) that serve multiple concurrent I/O operations and divide objects' data across disks in a process known as striping [34] to achieve high bandwidth.

Workflows: Some scientific processes rely on applications that perform different computational operations in sequence. For example, a tightly-coupled simulation using a subatomic particle model may generate a dataset that is then analyzed using a data processing application to evaluate the simulation's accuracy. In such cases, the applications have multiple stages that execute different tasks with different execution models and resource allocations (e.g. different sizes, durations, or specific hardware requirements). These *applications are often composed as workflows* that map the stages onto different batch jobs, which then receive stage-specific resource allocations [51]. However, workflows require the system scheduler to at least support the expression of dependencies between jobs, i.e. statements indicating that one job cannot start until another has finished correctly. Users commonly employ workflow managers that automate the submission of workflow jobs and supervise their execution.

2.2 HPC systems

HPC systems are necessarily shaped by applications' requirements and performance-efficiency trade offs. This section describes the characteristics of such systems with reference to Edison, a Cray XC30 supercomputer at NERSC that produces 2.5 Pflops/s of raw computational power [4]. Edison was deployed in 2014, when it was ranked at #18 in the Top500 list. As of April 2017, Edison is still operational and is ranked at #60 in the Top500.

Performance and efficiency: Distributed systems running scientific and industrial applications are created to support different economic models. For example, the compute infrastructure of an online company must always deliver the performance required to process any level of its variable bursty workload. Cost reduction and control of underutilization are addressed (e.g. through overbooking, elasticity, and idle capacity lending) but are considered secondary pro-

blems [37]. However, in high performance computing, workload levels always exceed the system's capacity and a backlog of waiting work is always present [20]. Also, applications require specialized hardware, which make HPC systems significantly more expensive than other large computing systems. Therefore, producing the maximum amount of work per cost unit is a high priority. The following subsections discuss some key characteristics of HPC systems that affect their efficiency in terms of the costs of system deployment, operating costs, and power consumption.

Uniformity and a high degree of parallelism: Applications for studying large scientific problems benefit from large or extreme parallelism. In some cases, the precision of their results depends on the availability of resources, as in the case of the weather simulation discussed above or when solving linear systems. In addition, tightly-coupled applications require parallelized hardware to achieve uniform performance and avoid straggler threads that slow down execution.

The Edison system consists of 5586 identical compute nodes, each having two processors (with 12 CPU cores each) and 64 GB of RAM, giving 134064 CPU cores and 357 TB of RAM in total. The system can run parallel applications with up to 134064 threads, or 268128 threads if hyper-threading is enabled. Such a system allows large applications to be executed without adversely affecting overall system performance. For instance, during the work on Paper I we observed applications allocating tens of thousands of threads in Edison without noticeably affecting the performance of other applications or their wait times.

A low-latency, low-jitter, high-bandwidth, uniform interconnect: The need to support large, parallel, tightly-coupled applications imposes unique requirements on the interconnect of a HPC system: it must provide synchronous, low latency, high-bandwidth communications between processes running across the whole system. These requirements affect the hardware and topology of the network. For instance, the degree of connectivity is higher and the maximum number of hops is smaller than in conventional networks. This ensures that latency does not vary significantly across the system, simplifying the scheduler's placement decisions. Also, network cards and drivers must support synchronized inter-process communication across nodes. This allows the processes of tightly-coupled parallel applications to reduce their execution time by controlling the precise instant at which a message is sent through the network.

In the case of Edison, its interconnect has a Dragonfly topology [33] that guarantees uniform low latency between all processes in all system nodes. This model's degree of connectivity is superior to that of the Fat-tree interconnects [35] used in older systems, which do not provide uniform internode latency and thus cause tightly-coupled applications to run more slowly unless all their assigned nodes are leaves of the same network switch.

Power-efficient computation: HPC systems are large systems (e.g. Edison has 134064 cores) that are used at almost their full capacity without interruption throughout their lifetime. Since their power budget is large (e.g. 3748 kW

for Edison), power efficiency is a critical requirement that must be considered during both hardware design and system operation. For example, systems are built with the most recent, power-efficient semiconductor and system technologies available at design time. Moreover, systems are retired when a newer system becomes available that consumes significantly less power while offering equivalent or superior compute capabilities.

For example, the Hopper supercomputer at NERSC produced 1.28 Petaflops/sec and offered 212 terabytes of RAM while consuming 2910 kW of power (2227 kW/Petaflops) [5]. It was retired after only four years of operation and replaced by Edison, a more modern and power-efficient system. Edison produces 2.59 Petaflops/sec and offers 356 terabytes of RAM while consuming 3747kW of power (1,446 kW/Petaflops) [4].

High-bandwidth, concurrent I/O: High performance systems support concurrent execution of many applications with high individual I/O requirements. This necessitates the use of large parallel file systems (PFSs) that divide files into stripes that are spread across multiple disks. The combined performance of multiple disks when reading or writing a file provides the high bandwidth needed in HPC systems. The distribution of files across a large pool of disks also provides the capacity to serve different applications concurrently while satisfying each application's performance requirements.

Edison is served by four different file systems with a total storage capacity of 7.56 petabytes [2]. Each file system has different performance characteristics that reflect its intended use. For example, the *home folder* system is the slowest, serving a maximum bandwidth of 100 MB/s. On the other hand, the *global scratch* Lustre PFS can serve over 700 GB/s of combined bandwidth.

Software support: HPC systems' software stacks offer common interfaces and functionalities to sustain widely-used operations in scientific applications. Such software often takes the shape of software libraries or execution frameworks. Many parallel tightly-coupled applications use message passing libraries to coordinate their processes. Consequently, systems include implementations of MPI (e.g. open-MPI [6] or MVAPICH2 [3]) that are adapted to the peculiarities of their interconnects and hardware. Similarly, execution frameworks facilitate application development by abstracting the underlying system, making it possible to create applications that should need only minor modifications to run on any system that supports the execution framework in question. For example, users in NERSC are increasingly migrating their Spark [55] and R [52] applications from their private clusters to Edison because both frameworks are supported at NERSC [1],

However, deploying new applications in an HPC system is not trivial, even if the required libraries are supported. Particularities in hardware and differences in implementation alter the behavior of libraries across systems, affecting the results of overlaying applications. To facilitate portability, NERSC systems support Shifter [30], a container technology [38] for HPC systems. Using Shifter, HPC applications are deployed together with specific versions of the required libraries that override those already present in the system.

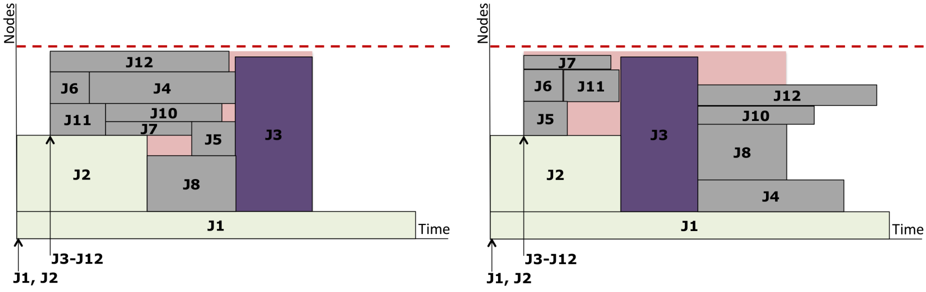


Figure 2.2: Job ordering for maximum utilization

Figure 2.3: Job ordering to avoid long individual turnaround times.

2.3 HPC scheduling

Applications in HPC systems are run as batch jobs, i.e. time-limited requests for resources to run the application binaries. To run multiple applications concurrently, HPC schedulers order the execution of batch jobs to achieve high utilization while controlling their turnaround times. However, as demonstrated by the following examples, these two objectives may oppose each other. Figures 2.2 and 2.3 present two different orderings for scheduling the same group of jobs. The job ordering in Figure 2.2 maximizes utilization and, over the measured window (indicated by the red lines), it minimizes the presence of idle resources. However, we observe that job J3 is delayed significantly more than the others. In contrast, Figure 2.3 depicts an ordering that limits the maximum per job wait time to control turnaround times. The resulting ordering equalizes the jobs' turnaround times more effectively (no job waits as long as J3 in the previous example), but yields more idle resources during the observation window. In HPC schedulers, the balance between utilization and turnaround time is controlled by the scheduler prioritization system and the scheduling algorithms, which are discussed in sections 2.3.1 and 2.3.2

Production HPC systems use different workload managers that combine scheduling and resource management. Some of the most popular are Slurm [54], LSF [15], Loadlever [31], and the combination of Moab (scheduler) with Torque (resource manager) [16]. While these tools are similar in characteristics and implementation, the work presented in this thesis was performed using Slurm because of its open-source nature and its dominant presence in the Top500 systems list.

The following subsections describe the components and mechanisms present in a generic HPC scheduler. Understanding of these descriptions may be facilitated by looking at Figure 2.4, which depicts the steps in the life of a batch job and the relationship between the scheduler's components.

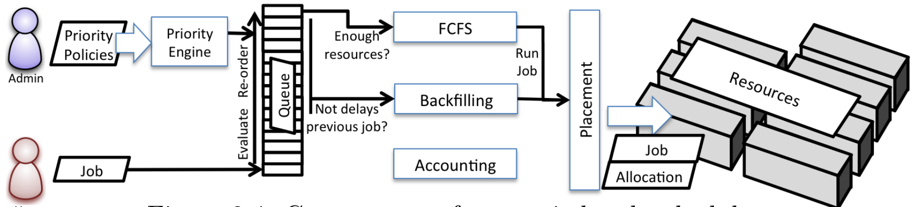


Figure 2.4: Components of a generic batch scheduler.

2.3.1 Job submission and priority

The life-cycle of a batch job starts with its **submission** to the system. The batch job submission must provide a detailed specification of the requested resources (e.g. the number of cores, minimum RAM per core, or specific compute nodes to run on), an estimate of the job's runtime, a priority request (expressing the job's importance), and, if applicable, a list of dependencies on other jobs (e.g. statements that the job should not start until some set of conditions is met).

If no other jobs are waiting and there are enough resources available, the scheduler runs the job immediately. Otherwise, it is appended to a **job waiting queue**, and will be executed when it reaches the queue's top area. The jobs in the waiting queue are initially ordered by arrival time. However, to steer the jobs' turnaround time, jobs are ranked and re-ordered by a **priority engine**. Different ranking policies define priorities based on job size (e.g. smaller jobs should run sooner), priority class (e.g. jobs in the real time class should run before any other job), fairness (i.e. priorities dictated by system quotas), or other administrator-defined criteria. This prioritization makes it possible to steer the turnaround time of jobs by assigning higher priorities to those that should have shorter turnaround times.

2.3.2 Job scheduling

Jobs progress towards the top area of the waiting queue until they are extracted by the **scheduling algorithms** and then executed. Most HPC batch schedulers include the *FCFS* (First Come First Served) and *backfilling* scheduling algorithms [50]. Under FCFS, the first jobs of the queue are run in order until there are no longer enough unallocated resources for the next job in the queue. This algorithm preserves the queue's initial order, guaranteeing a certain level of wait time equality among jobs. However, by itself FCFS produces low utilization values. Figure 2.5 presents an example of the scheduling of five jobs using FCFS alone. The jobs J1 and J2 start as soon as they are submitted. J3 starts as soon as J2 ends and enough resources become available. The jobs J4 and J5 start after J3 because they arrived later. Given the available idle resources, J4 could be started before J3 to increase utilization. However, FCFS does not support such changes in job order.

The backfilling algorithm solves the underutilization problem by starting

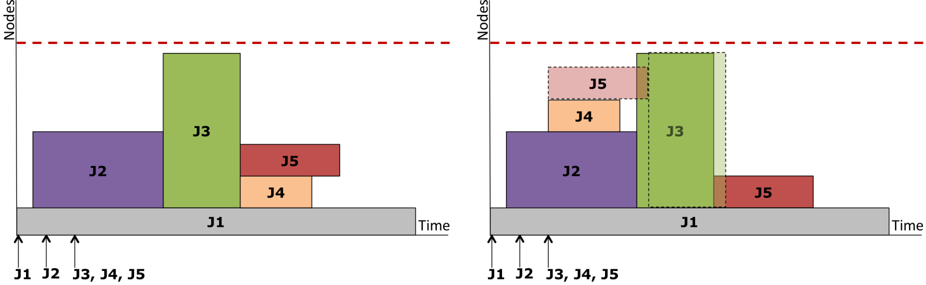


Figure 2.5: Scheduling with FCFS alone: idles resources over J2.

Figure 2.6: Scheduling with FCFS and backfilling: job J4 is backfilled.

waiting jobs whose execution would not delay previous jobs in the queue. An example of its effect is presented in Figure 2.6. The job J4 is scheduled before J3 because it does not delay the latter job's expected start time (which is after J2 ends). However, J5 is started after J3 because its backfilling (plotted with dotted lines) *would* delay the start of J3 (favoring wait time control) .

These two algorithms are the most common scheduling mechanisms present in HPC schedulers. They are effective in achieving high utilization and producing turnaround times (enforced in the waiting queue order) that reflect the jobs' priorities. However, they do not take into account any application-specific characteristics, which may lead to suboptimal scheduling, for example by generating long turnaround times for workflows as intermediate wait times accumulate. Some of the work presented in this thesis (Paper V) involved modifying these mechanisms to improve workflow scheduling.

2.3.3 Placement, accounting, and other functions

Although beyond the scope of this thesis, HPC schedulers include several mechanisms that are needed to manage an HPC system. Among other things, they incorporate placement systems that calculate which resources should be used for specific jobs. These subsystems' decisions take into account the network topology or special job requirements. For instance, in a system with a fat-tree interconnect topology [35], a tightly coupled application will run faster if all its assigned nodes are leaves pending from same network switch. Otherwise, inter-node latencies may vary, slowing down all the application's processes.

HPC systems also require accounting to register the use of compute hours and resources by user jobs. In general, HPC centers assign allocations of core-hours or system shares to users and projects (i.e. quotas). Accounting is required to prevent users from utilizing the system beyond their assigned quota (e.g. by de-prioritizing their jobs) and to encourage those who have not used it (e.g. by elevating the priority of users with little quota usage).

Finally, workload managers include functions to handle the basic operations to run an HPC system, such as managing the compute resources, staging-in jobs, controlling their execution, and staging-out resources.

Chapter 3

New scheduling challenges

The use of HPC in scientific research is increasing, introducing new application models into HPC workloads and necessitating increases in the capacity of HPC systems. This chapter outlines some of the challenges of scheduling and running these new HPC workloads. It also describes the difficulties of building larger HPC systems, many of which relate to the limitations of current semiconductor technologies, and the resulting challenges for HPC schedulers.

3.1 Present and future workloads

New scientific applications are changing the scientific workload landscape. HPC workloads were traditionally, dominated by large parallel tightly-coupled applications, but recent years have seen dramatic increases in the prevalence of other application types including significantly more data-intensive and high throughput applications, workflows, and malleable jobs. This section describes the requirements of such applications and the challenges they present for HPC systems and schedulers.

Data-intensive workloads: Many scientific discoveries are supported by data produced by scientific simulations run in HPC systems (e.g. atmospheric simulations). Also, a large corpus of science is based on analyzing data generated during real world experiments (e.g. large hadron collider experiments). Increases in sensor resolutions and the availability of compute capacity are causing the quantities of data produced by experiments and simulations to grow at unprecedented rates. This has increased the importance and complexity of data analysis in science, establishing what has been described as the "fourth paradigm" [27]: scientific discoveries resulting from the use of data-intensive and data analysis processes to interpret, group, and correlate large datasets.

However, the performance model of data intensive applications often differs from that of the parallel tightly-coupled applications (e.g., MPI) [22] that historically dominated HPC workloads and guided the design of HPC systems

and their schedulers. While the performance of MPI applications depends on execution uniformity and communication latency [7], data-intensive applications' performance typically depends on memory and I/O performance [22]. For example, the interconnect of an HPC system dominated by I/O intensive applications will be heavily utilized, increasing the possibility of network hotspot formation and congestion, and reducing the system's overall performance. Consequently, future schedulers will require placement systems that account for the system's network topology and the expected I/O traffic generated by individual applications.

Old and new characteristics: Some characteristics of applications are rarely taken into account by modern schedulers. For example, malleable applications support changes in resource allocation during their runtime [25]. Such applications were rare in traditional HPC workloads, so few HPC schedulers incorporate mechanisms for exploiting their malleability. However, data-intensive applications are often malleable and are becoming important components of HPC workloads. Therefore, the importance of managing malleability is growing. Malleability presents opportunities to increase system utilization or, as described in Paper III [47], to quickly borrow resources for other applications.

On the other hand, new applications may have new requirements. For example, the Advanced Light Source [32] is a particle accelerator that produces high quality infra-red and x-ray beams to "illuminate" experiments. These experiments produce large amounts of data that are transferred to NERSC systems for later processing and later analysis. These experiments would benefit from real time data processing - for example, scientists could use live feedback [10] to adjust their experimental conditions in real time. This could be enabled by running the experiment application using pre-allocated resources at a planned time (i.e. by making advance resource reservations). However, such practices significantly reduce overall utilization, especially because experiments' start times are hard to predict. Consequently, it is difficult to schedule quasi-real time applications without reducing utilization. To ease this problem, we propose a model (Paper III [47]) that takes advantage of the presence of malleable applications to temporarily borrow resources and allocate them to real-time applications. Another approach has been implemented by NERSC [29]: "real-time jobs" are allocated small resource sets but are given extremely high priority. They are then run in a partition of the system where jobs' runtimes are observed to be short. Because the probability of a job to end in this partition is high at any given moment, resources are often freed up for the real-time applications. This strategy enables wait times of about two minutes, which is close to real-time for a batch job.

Workflows: Scientific applications are becoming more complex, requiring the composition of various applications with different resource requirements [51] or compute models. In a very simple example, a simulation stage requiring a large parallel allocation is often followed by a data analysis job that processes the simulation's results using a long serial job code. Also, applications running on a heterogeneous system could potentially use different resource types at dif-

ferent stages of their execution. For instance, an application might use regular compute nodes supporting MPI in a first stage, followed by a machine learning classification stage that requires compute nodes with GPUs and large memory.

In both cases, the application consists of different tasks that can be allocated as different jobs, together forming a workflow. However, current schedulers' workflow support is limited to offering job dependencies. In this approach, users submit workflows as a group of jobs and dependencies, which prevent one job from starting until the jobs it depends on have successfully completed. This leads to long intermediate wait times and overall turn-around times because the scheduler does not consider whether a job belongs to a workflow or not. Consequently, users often run workflows as pilot jobs, which are allocated the maximum resources required at any point in the workflow's runtime for the entire duration of that runtime. This minimizes the job's turnaround time, but wastes resources through over-allocation.

This thesis presents new ways of solving such problems. For example, it introduces a scheduling model that includes different schedulers for batch jobs and workflows (Paper III [47]), eliminating the interference of regular jobs and reducing the workflow's intermediate wait time. It also details an algorithm for modifying the scheduler's job waiting queue (Paper V [44]), which enables regular schedulers to minimize their turnaround time without over-allocating resources.

Job diversity: MPI applications, data intensive applications, malleable jobs, real time jobs, workflows, and regular jobs co-exist in current HPC workloads [8]. This application diversity poses new challenges for the scheduling process. For example, applications may be affected differently by the scheduler's decisions - the wait time of a workflow job affects the total workflow turnaround. Also, different applications may have different performance objectives: real time jobs must be run as quickly as possible, whereas regular jobs need only achieve "reasonable" turnaround times. Moreover, some applications support specific scheduling operations, e.g., malleable jobs support online re-sizing of their resource allocations. However, current HPC schedulers treat all applications equally, and specific scheduling behaviors are enforced by tweaking the scheduler's configuration. For example, real-time applications at NERSC are run as high priority jobs on a partition containing small, fast-running jobs [29].

This thesis presents an alternative model (Paper III [47]) in which independent schedulers manage applications of different types. This makes it possible to compare similar applications using the same criteria and incorporate mechanisms for exploiting their unique characteristics or ensuring that their unique requirements are satisfied.

Finally, application diversity also affects the geometry of the jobs in the workload. The characterization of NERSC workloads presented in Paper I [46] includes an analysis of this phenomenon, revealing that batch jobs in the Edison, Hopper, and Carver workloads are very diverse in their geometries. Moreover, it is shown that a job's wait time may depend on the geometry diversity within its priority group (queue).

3.2 Future systems

Because Grand Challenge science is increasingly supported by simulations, data analysis, and numerical methods, larger and more powerful HPC systems are required to increase scientific research capacity. Increases in capacity were traditionally achieved by adding more compute nodes and increasing processor frequency, the amount of RAM per compute node, and the size of the parallel file system. However, the limitations of current semiconductor technologies have made some of these strategies ineffective or impractical. This section presents some of the challenges of scaling up modern HPC systems and their impact on system schedulers.

3.2.1 Dennard scaling break and processor parallelism

Dennard's scaling law is related to Moore's law, and states that the performance per watt of computing devices is growing exponentially at a constant rate [26]. This has made it possible to increase the number of transistors per chip surface unit and their operating frequencies without causing major increases in the processors' power budgets and heat dissipation. This was responsible for much of the increase in the capacity of all computational systems up until 2006, a year that has been described as the *Dennard scaling break* [26]. Since then, processors' transistor counts and parallelism have increased but their operating frequencies have not, and in some cases have even fallen slightly. Consequently, recent processor speed-ups have been much more gradual than before the break.

The primary reason for the break is that transistors have become so small that their switching frequency is limited by current leakage. Higher frequencies imply more residual heat production and power consumption, which has two side effects [24]: (1) processor frequencies cannot increase at historical rates, reducing processing speed-ups; and (2) adding more cores to the processors increases their capacity, but the higher number of active switching elements still increases overall power consumption and makes CPU heat dissipation problems more severe. These problems can be mitigated (but not eliminated) by using concepts such as the dark silicon approach [24], whereby some of the chip's circuits are powered down as necessary to keep its power consumption within safe limits. Unfortunately, this limits the chip's potential performance.

Also, even if higher core parallelization within in processors can be achieved without power constraints, this will not necessarily deliver perfect performance scaling. First, increasing the number of cores in a processor increases the likelihood of resource contention on caches and memory busses, which reduces the processor's effective performance [28]. Also, even assuming perfect hardware parallelization, Amdahl's law limits the speed-ups that can be achieved through parallelization and states that these speed-ups depend on the nature of the workload [28]. Specifically, the performance gain achievable through parallelization depends on the relative prevalence of serial and parallel code within the application or workload, which in turn depends on the usage patterns of individual users.

3.2.2 The memory gap

Memory technology has been outpaced by Moore's law, and memory performance and power-efficiency bottlenecks will become more severe in future systems. This section describes some of the problems arising from these bottlenecks and limitations.

Ideally, the amount of RAM in compute nodes should grow in parallel with the number of cores. However, this has not been possible to achieve because memory density grows more slowly than the scaling implied by Moore's law [11]. Put another way, current memory technology is less power-efficient than processor technology. Consequently, in recent years the amount of memory per core in larger systems has been getting smaller, limiting the memory footprint of applications' individual processes. Also, the gap between processors' cycle times and the memory latency has necessitated the introduction of caches to prefetch data from the slower memory before the processor requests it. However, the cache size cannot increase at the same rate as the number of cores per processor because this would require the cache to take up too much space on the processor die [24]. This reduces the efficiency of the cache mechanism and the performance of memory-related operations in code. Moreover, since memory technology is less power-efficient than CPU technology, memory movements and operations are becoming one of the largest contributors to the power consumption of modern systems [11]. This will limit the potential for increasing the size of memory hierarchies, necessitating the development of alternative mechanisms to compensate for the memory gap.

The memory gap's impact is demonstrated by the performance of the #1 super computer on the Top500 list (in March 2017): the Sunway TaihuLight system [17]. This supercomputer has a theoretical peak performance of 125.4 Pflops/s produced by 10649600 cores over 1.31PB of RAM. Its first performance measurements were impressive, with a Linpack benchmark result of 93 Pflops/s, corresponding to 74% of its theoretical peak performance [18]. However, when tested with benchmarks that model real applications' data access patterns (the HPCG benchmark [19]), it achieved only 0.3% of the theoretical peak performance. This value is much lower than those achieved by the next two systems in the Top500 (1.1% for Tianhe-2 and 1.2% for Titan), and suggests a lower efficiency when executing real applications. Initial reports on the system [18] attribute this low efficiency to the memory gap, and specifically to its low number of floating point operations per byte of data transferred from memory to the chips (22.4 Flops(DP)/Byte transfer) and the small amount of memory per core (132 MB).

To conclude, the memory gap does not present direct challenges to schedulers. However, the scheduler will be confronted by fragmented resources if the memory gap problem is solved by increasing heterogeneity because some resources may be tailored to memory-intensive applications.

3.2.3 I/O gap

As with memory, there is also a performance gap between the I/O and processing capabilities of HPC systems. Traditional parallel file systems achieve concurrent high read and write bandwidths by striping and replicating data across disks and combining their bandwidths in I/O operations. However, increasing compute capacity implies more numerous and larger applications running concurrently, and, as a consequence, higher aggregated I/O bandwidth peaks. In older systems, this problem was solved by increasing the parallelism of the PFS (i.e. adding more disks). However, this is not economically viable with modern systems: adding more spinning disks is too expensive in terms of both material costs and energy consumption. Replacing the spinning disks of PFS with higher bandwidth solid state drives (SSD) would increase overall bandwidth capacity and reduce I/O-related power consumption. However SSDs are currently too expensive to be used in the construction of a complete PFS [36].

An intermediate solution is to increase the depth of the I/O hierarchy by, for example, allocating burst buffers to applications. A burst buffer is a space allocated in a small PFS built with SSDs that can absorb peaks in the system's I/O operations [36]. For read peaks, data can be pre-fetched in a similar way as in memory caches. For write peaks, the data can be directly written to the burst buffer and then subsequently 'drained' to the main PFS. However, this poses a new challenge for HPC schedulers because the burst buffer space becomes a new constrained resource to be allocated to applications that (if poorly managed) can reduce scheduling efficiency and hence reduce utilization, leading to longer wait times. For instance, the scheduler must determine a burst buffer allocation for each application that is large enough to absorb the application's read and write bursts but not so large as to be underutilized and unduly limit the burst buffer capacity available to other applications.

3.2.4 Hardware heterogeneity

The previous sections dealt with new HPC challenges that were addressed by adding problem-specific hardware, including application diversity, the limits of processor frequency speed-ups, growing I/O requirements, and the memory gap. For example, some modern systems pair regular compute nodes with GPU-based compute nodes that offer better performance in machine learning or artificial intelligence applications [14].

The increasing heterogeneity of HPC systems poses new challenges for schedulers. For example, an application's performance may depend on the scheduler's placements more strongly than was previously the case. Also, applications' resource requests may no longer be uniform; certain applications might request specific resource types. In such contexts, the scheduler must assign resources that satisfy the applications' demands without fragmenting the resource pool and thus reducing utilization.

3.3 Scheduling research

One challenge in HPC scheduling research relates to how such research should be performed, because such research requires information, methodology, and resources that are not always available. This section explores the process of performing HPC scheduling research and the tools available to support that process.

Previous studies [48] have defined three basic methods for job scheduling research:

- **Theoretical analysis:** in which algorithm behavior is analyzed by identifying boundary cases representing the best- and worst-case performance. This method can provide insight into the algorithm's behavior but might not reflect its expected performance when supporting a real workload.
- **Real system experiments:** which provide clear measures of the algorithm's behavior with a real system and workload but require the running of many different experiments, which can be challenging for several reasons. First, allocating real system time is expensive. Also, a single experiment only provides results for one workload and system state, which is normally not enough to reason about the general case or test a general hypothesis. Finally, workload conditions are hard to control, so it can be difficult to test specific scenarios.
- **Simulation:** in which a system is emulated, a batch scheduler is run on the emulated system, and a synthetic workload is submitted to the scheduler. This method makes it possible to create different experimental scenarios and run experiments on large scale to generate data that enables induction to general conclusions. However, its results are only valid if the recreated workloads, systems, and scheduling behaviors are representative.

The works presented in this thesis used simulation and theoretical analysis to understand how the hierarchical fairshare prioritization engine works (Paper II, [43]) In addition, the Edison system was modeled after its job logs, and the resulting models were used to perform simulations and evaluate a workflow-aware scheduling method (a theoretical analysis of this method was also performed [42] but is not included in this thesis).

Research using scheduling simulations follows the cycle depicted in Figure 3.1. This process starts with system modeling, which captures the characteristics and typical workloads of the system under investigation. These characteristics are used to configure the simulator and create model workloads that are representative of the original system. Once the system has been modeled, an experiment is defined and performed in the simulator, and the scheduler's actions are recorded for later analysis. To test a scheduling hypothesis, it is necessary to define multiple experiments that must be replicated several times to ensure that representative results are obtained. For example, the experimental

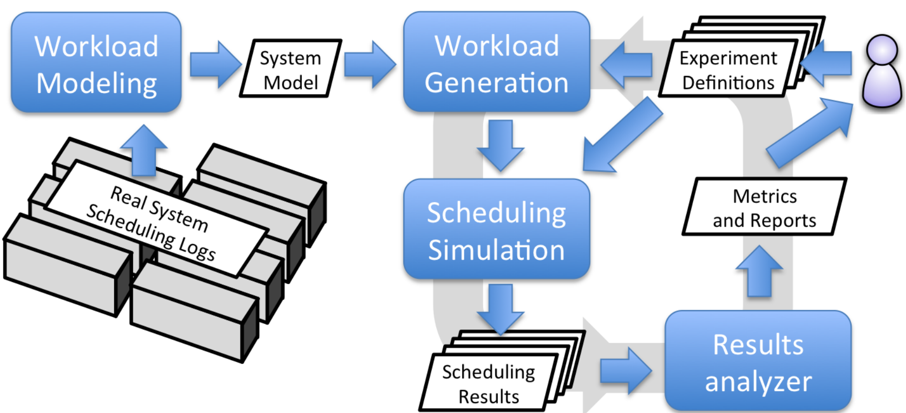


Figure 3.1: The scheduler research cycle. Experiments are defined and simulated, and then their results are analyzed. Finally, the resulting data are used to prove/disprove a hypothesis or suggest new experiments.

testing of one of the workflow scheduling algorithms presented in this thesis involved testing with six workflow types appearing in five different patterns, scheduled with three different techniques, with six experimental replicates per combination of pattern, workflow type, and technique. These 540 experiments spanned six simulated days of Edison's life, representing 3240 days of supercomputer time. Simulating them on a single instance, would take 320 days (at an estimated 10x speedup). To avoid such unrealistically long experiments, it is necessary to perform large-scale concurrent simulations.

However, no tools previously available to the the research community could be applied to every step in the scheduler research cycle or be used to perform experiments on large scales. For example, these tools lack modules for workload analysis and generation. Moreover, their scheduling simulation components are not powerful enough to run all the necessary experiments within a reasonable time frame, or they require the use of a scheduler that is unrepresentative of those used in HPC production systems. To address these issues, Paper IV [45] presents a new simulation framework that covers every step in the scheduler research cycle. It can model systems and their workloads to generate equivalent synthetic workloads. It then emulates the operation of the modeled system using a real instance of Slurm, and schedules a synthetic workload. It also provides tools for running concurrent simulations on large scale, and tools for analyzing their results.

Chapter 4

Contributions

This chapter summarizes the contributions of this thesis and the included papers as outlined in the introduction. The first contribution is a characterization of the HPC workloads of three systems at NERSC (Paper I [46]). This enabled the identification of present and future scheduling challenges such as the effects of workload diversity on the predictability of jobs' wait times. The second contribution is a characterization and improvement of a common HPC prioritization system: a hierarchical fairshare system (Paper II, [43]). The third contribution is a survey of present and future scheduling HPC problems together with a proposed scheduling model that solves some of them (Paper III, [47]). The development of that model was hindered by a lack of suitable tools for performing research on HPC scheduling. This led directly to the fourth contribution: the development of an open source scheduling simulation framework (Paper IV, [45]). This framework allows scheduling researchers to implement and evaluate scheduling algorithms in the context of the Slurm workload manager. The final contribution is a new workflow-aware scheduling method (Paper V, [44]) that minimizes workflows' turnaround times without over-allocating resources. Its implementation is open source and its performance was evaluated using our scheduling simulation framework.

The following sections of this chapter describe these contributions in more detail.

4.1 Paper I: Characteristics and trends in HPC workloads

Paper I [46] analyzes the characteristics of the jobs in the workloads of two supercomputers (Edison and Hopper) and a high performance cluster (Carver) at the National Energy Research Scientific Computing Center (NERSC) in Berkeley, California. The analysis features both a classical characterization of job variables and a trend analysis based on these variables that accounts for job diversity.

The first contribution of this paper is a novel method for characterizing diversity in job geometry (allocated CPU cores and runtime) based on $k-means$ clustering. This makes it possible to identify dominant job geometries in a workload and analyze their mapping onto priority queues. The second contribution is a characterization of the NERSC systems' workloads, providing reference datasets for the workloads of systems having similar sizes, workloads, and architectures to Edison, Hopper, and Carver. The third contribution is the observation that priority queues containing more diverse job geometries exhibited less predictable wait times. According to the expected batch scheduler behavior, for a given priority, larger jobs should have longer wait times. In addition, for a given job geometry, jobs with higher priority are expected to wait less. The analysis showed that these expectations are not fulfilled for queues with more diverse job geometries. This indicated a need to better understand the effects of job diversity on HPC systems and the performance of their schedulers.

4.2 Paper II: Prioritization in batch schedulers, Fairshare

Paper II [43] analyzes the internal mechanisms of Karma, a distributed prioritization mechanism that supports hierarchical policies. This work provides an expanded understanding of the prioritization system itself and its effect on the division of the system's compute time based on a fairshare priority factor. Karma and similar mechanisms are used in HPC sites to calculate jobs' fairshare prioritization factors. These factors account for differences between the shares of a system that are assigned to users, groups, or institutions, and the corresponding shares of actual usage. The capacity of this system has been evaluated before [39], [23], but this paper goes further than previous evaluations by explaining how certain system properties emerge from the definition of one of the model's key components: the fairshare priority operators. For example, the hierarchical model makes it possible to assign an allocation share to a project and then divide that share among the users belonging to that project. If one of those users does not consume their allocation, it would be desirable for the prioritization system to divide that allocation between the project's other users, in proportion to their prior allocations within the project. This work shows how the calculation of some of the operators in Karma produces priorities equivalent to those obtained if the idle user did not exist and the other project users received a proportional allocation of that user's share.

The second contribution of this paper is an analysis of the effect of the resolution of the scheduler's priority factor on Karma's capacity to discern priority differences in hierarchical models with many levels. This revealed that the factor resolution is divided across the the levels of the priority tree and can be used to determine the minimum priority factor resolution needed for any given hierarchical fairshare share tree.

4.3 Paper III: A survey of current scheduling challenges and a position model to ease them

Building on the understanding of current HPC workloads and scheduling mechanisms obtained in Papers I [46] and II [43]), Paper III [47] makes two contributions. The first is a a survey of current and future scheduling challenges in HPC systems and an analysis of their relationships to cloud scheduling challenges. This survey focuses particularly on the issue of schedulers applying the same policies to applications with different performance models and target objectives. For instance, stream applications that require short wait times are scheduled by the same mechanisms as workflows whose turnaround times should ideally be short but depend on the intermediate wait times between jobs. It also shows that malleable jobs are becoming increasingly common but most HPC schedulers cannot exploit their particular characteristics, i.e. the potential to change their resource allocation during their runtime. Finally, it summarizes other challenges such as the need for deeper memory and I/O hierarchies in future HPC systems.

The paper's second contribution is a description of A2L2, an *Application Aware Flexible Scheduling Model for Low Latency Allocation.* A2L2 is a cloud-inspired, two-level scheduling model that supports the use of one scheduler per application type with a smart resource manager. Each scheduler prioritizes applications of the appropriate type, and thus is not required to compare applications with different performance models and targets (Application Aware). Schedulers in such system could offer application-specific capabilities, such as dynamic resource allocation for malleable applications (Flexible). The underlying smart resource manager provides primitives to request resources with short wait times using borrow operations. During a borrow operation, the resource manager takes resources from schedulers that manage malleable applications and lends them to the requesting scheduler. This enable support for almost real time (Low Latency) jobs by allowing resources to be freed rapidly enough for such jobs to be allocated.

4.4 Paper IV: Building tools for scheduling research

The evaluation of job scheduling algorithms by simulation involves a research cycle consisting of system and workload modeling, workload generation, scheduling and simulation, and result analysis. However, the analysis and implementation of scheduling techniques reported in Papers I-III was hampered by a lack of suitable publicly available research tools to support this cycle.

The contribution of Paper IV [45] is ScSF, an open source Scheduling Simulation Framework that can be used by future scheduling researchers. ScSF includes tools for running and coordinating every step of the simulation research

cycle. It creates system models that can subsequently be used in experiments. Users can define experiments based on a system and model workloads while also incorporating experimental conditions needed to analyze a specific scheduling behavior, such as a particular configuration of the priority engine, the presence of certain types of workflows in the workload, or a workload engineered to test classic backfilling efficiency. The framework runs these experiments automatically, generating synthetic workloads that are submitted to an instance of a Slurm simulator. The Slurm simulator emulates the resources of the chosen system model, replays the job submission, and executes a real instance of the Slurm scheduler with modifications of the time routines to accelerate its execution. The framework also includes tools to analyze relevant scheduling variables in the resulting scheduling logs and perform comparisons across experiments.

The number of experiments performed to test scheduling hypotheses can be large as the system, and there are multiple degrees of freedom in terms of workload composition. Therefore, the framework can be used to run large scale experiments. The paper demonstrates this capability by running the framework over 161 VMs across a distributed infrastructure of 17 physical compute nodes at LBL and Umeå University. The final contribution of this paper is a set of lessons learned from the process of running large scale scheduling simulations that will be useful to future researchers.

All the components of the framework other than Slurm and its simulator were created by the authors. Additionally, Slurm was modified to increase its simulation speed while preserving time determinism.

4.5 Paper V: Workload-aware scheduling in HPC systems

One of the challenges identified in Paper III [47] was the scheduling of workflows within HPC systems. Workflows are often run as a group of jobs related by completion dependencies, leading to long intermediate wait times and thus long turnaround times. An alternative possibility is to run all the workflow tasks within a single pilot job that is allocated the maximum resources required within the workflow throughout its runtime, potentially causing some resources to be idle during some stages of the workflow.

The main contribution of Paper V [44] is WoAS, a new Workflow Aware Scheduling algorithm that can minimize workflows' intermediate wait times (and thus their turnaround times) without over-allocating resources. Using this technique, workflows are initially represented in the same way as in the pilot job approach. However, the pilot job includes extra information that describes the workflow's internal structure. This information is used by the scheduler to modify the representation of the workflow during different stages of its scheduling: it is treated as a pilot job during prioritization but then its internal tasks are exposed to the FCFS and backfilling algorithms.

An open-source implementation of the algorithm was incorporated into

Slurm and evaluated in the ScSF framework by simulating workloads with different types and abundances of workflows. It was found that the WoAS can run workflows with turnaround times that are significantly shorter than for the chained jobs approach (achieving speedups of up to 3.75x) and almost match those achieved using the pilot jobs approach, but without allocating extra resources.

Bibliography

[1] Data analytics (at NERSC),
http://www.nersc.gov/users/data-analytics/data-analytics/

[2] Edison file storage and I/O,
http://www.nersc.gov/users/computational-systems/edison/file-storage-and-i-o/

[3] MVAPICH, http://mvapich.cse.ohio-state.edu/

[4] NERSC's Edison, Top 500, https://www.top500.org/system/178443

[5] NERSC's Hopper, Top 500, https://www.top500.org/system/176952

[6] Open mpi, https://www.open-mpi.org/

[7] Adhianto, L., Chapman, B.: Performance modeling of communication and computation in hybrid MPI and OpenMP applications. Simulation Modelling Practice and Theory 15(4), 481–491 (2007)

[8] Ahern, S., Alam, S.R., Fahey, M.R., Hartman-Baker, R.J., Barrett, R.F., Kendall, R.A., Kothe, D.B., Mills, R.T., Sankaran, R., Tharrington, A.N., et al.: Scientific application requirements for leadership computing at the exascale. Tech. rep., Oak Ridge National Laboratory (ORNL); Center for Computational Sciences (2007)

[9] Antypas, K., Austin, B., Butler, T., Gerber, R., Whitney, C., Wright, N., Yang, W.S., Zhao, Z.: NERSC workload analysis on hopper. Tech. rep., Technical report, LBNL Report (2013)

[10] Bauer, M.A., Biem, A., McIntyre, S., Tamura, N., Xie, Y.: High-performance parallel and stream processing of x-ray microdiffraction data on multicores. In: Journal of Physics: Conference Series. vol. 341, p. 012025. IOP Publishing (2012)

[11] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al.: Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep 15 (2008)

[12] Bethel, E., Leinweber, D., Rübel, O., Wu, K.: Federal market information technology in the post flash crash era: roles for supercomputing. In: Proceedings of the fourth workshop on High performance computational finance. pp. 23–30. ACM (2011)

[13] Bhat, P.B., Lim, Y.W., Prasanna, V.K.: Issues in using heterogeneous hpc systems for embedded real time signal processing applications. In: Real-Time Computing Systems and Applications, 1995. Proceedings., Second International Workshop on. pp. 134–141. IEEE (1995)

[14] Catanzaro, B.: Deep learning with COTS HPC systems (2013)

[15] Chiang, S.H., Vernon, M.K.: Production job scheduling for parallel shared memory systems. In: Parallel and Distributed Processing Symposium., Proceedings 15th International. pp. 10–pp. IEEE (2001)

[16] Declerck, T.M., Sakrejda, I.: External Torque/Moab on an XC30 and Fairshare. Tech. rep., NERSC, Lawrence Berkeley National Lab (2013)

[17] Dongarra, J.: Report on the Sunway Taihulight system. www. netlib. org. Retrieved June 20 (2016)

[18] Dongarra, J.: Sunway TaihuLight supercomputer makes its appearance. National Science Review 3(3), 265 (2016), `http://dx.doi.org/10.1093/nsr/nww044`

[19] Dongarra, J., Heroux, M.A.: Toward a new metric for ranking high performance computing systems. Sandia Report, SAND2013-4744 312 (2013)

[20] Dongarra, J., Sterling, T., Simon, H., Strohmaier, E.: High-performance computing: clusters, constellations, mpps, and future directions. Computing in Science & Engineering 7(2), 51–59 (2005)

[21] Dongarra, J., et al.: The international exascale software project roadmap. International Journal of High Performance Computing Applications p. 1094342010391989 (2011)

[22] Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: eScience, 2008. eScience'08. IEEE Fourth International Conference on. pp. 277–284. IEEE (2008)

[23] Elmroth, E., Gardfjäll, P.: Design and evaluation of a decentralized system for Grid-wide fairshare scheduling. In: H. Stockinger et al (ed.) Proceedings of e-Science 2005. pp. 221–229. IEEE CS Press (2005)

[24] Esmaeilzadeh, H., Blem, E., St Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: ACM SIGARCH Computer Architecture News. vol. 39, pp. 365–376. ACM (2011)

[25] Feitelson, D.G., Rudolph, L.: Toward convergence in job schedulers for parallel supercomputers. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 1–26. Springer (1996)

[26] Haensch, W., Nowak, E.J., Dennard, R.H., Solomon, P.M., Bryant, A., Dokumaci, O.H., Kumar, A., Wang, X., Johnson, J.B., Fischetti, M.V.: Silicon cmos devices beyond scaling. IBM Journal of Research and Development 50(4.5), 339–361 (2006)

[27] Hey, T., Tansley, S., Tolle, K.M., et al.: The fourth paradigm: data-intensive scientific discovery, vol. 1. Microsoft research Redmond, WA (2009)

[28] Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer 41(7) (2008)

[29] Jacobsen, D.: NERSC Site Report-One year of Slurm. In: Slurm User Group 2016 (2016)

[30] Jacobsen, D.M., Canon, R.S.: Contain this, unleashing docker for hpc. Proceedings of the Cray User Group (2015)

[31] Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F.: Workload management with loadleveler. IBM Redbooks 2(2) (2001)

[32] Kilcoyne, A., Tyliszczak, T., Steele, W., Fakra, S., Hitchcock, P., Franck, K., Anderson, E., Harteneck, B., Rightor, E., Mitchell, G., et al.: Interferometer-controlled scanning transmission X-ray microscopes at the advanced light source. Journal of synchrotron radiation 10(2), 125–136 (2003)

[33] Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-driven, highly-scalable dragonfly topology. In: ACM SIGARCH Computer Architecture News. vol. 36, pp. 77–88. IEEE Computer Society (2008)

[34] Kim, Y., Gunasekaran, R.: Understanding i/o workload characteristics of a peta-scale storage system. The Journal of Supercomputing 71(3), 761–780 (2015)

[35] Leiserson, C.E.: Fat-trees: universal networks for hardware-efficient supercomputing. IEEE transactions on Computers 100(10), 892–901 (1985)

[36] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., Maltzahn, C.: On the role of burst buffers in leadership-class storage systems. In: Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on. pp. 1–11. IEEE (2012)

[37] Mell, P., Grance, T., et al.: The nist definition of cloud computing (2011)

[38] Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal 2014(239), 2 (2014)

[39] Östberg, P.O., Espling, D., Elmroth, E.: Decentralized scalable fairshare scheduling. Future Generation Computer Systems - The International Journal of Grid Computing and eScience 29 pp. 130–143 (2013)

[40] Raicu, I., Foster, I.T., Zhao, Y.: Many-task computing for grids and supercomputers. In: Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on. pp. 1–11. IEEE (2008)

[41] Rockel, B., Will, A., Hense, A.: The regional climate model COSMO-CLM (CCLM). Meteorologische Zeitschrift 17(4), 347–348 (2008)

[42] Rodrigo, G.P.: Theoretical analysis of a workflow aware scheduling algorithm. Tech. Rep. UMINF17.06, Umeå University (2017)

[43] Rodrigo, G.P., Östberg, P.O., Elmroth, E.: Priority operators for fairshare scheduling. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 70–89. Springer (2014)

[44] Rodrigo, G.P., Elmroth, E., Östberg, P.O., Ramakrishnan, L.: Enabling Workflow Aware Scheduling on HPC systems. Submitted (2017)

[45] Rodrigo, G.P., Elmroth, E., Östberg, P.O., Ramakrishnan, L.: ScSF: A Scheduling Simulation Framework. In: Workshop on Job Scheduling Strategies for Parallel Processing. Accepted, Springer (2017)

[46] Rodrigo, G.P., Östberg, P.O., Elmroth, E., Antypas, K., Gerber, R., Ramakrishnan, L.: Towards Understanding HPC Users and Systems: A NERSC Case Study. Submitted (2017)

[47] Rodrigo, G.P., Ramakrishnan, L., Östberg, P.O., Elmroth, E.: A2L2: an Application Aware flexible HPC scheduling model for low latency allocation. In: The 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC) (2015)

[48] Schwiegelshohn, U.: How to design a job scheduling algorithm. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 147–167. Springer (2014)

[49] Snir, M., Otto, S., Walker, D., Dongarra, J., Huss-Lederman, S.: MPI: The complete reference. MIT Press Cambridge, MA, USA (1995)

[50] Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Parallel Processing Workshops, 2002. Proceedings. International Conference on. pp. 514–519. IEEE (2002)

[51] Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M.: Workflows for e-Science: scientific workflows for grids. Springer Publishing Company, Incorporated (2014)

[52] Team, R.C.: R language definition. Vienna, Austria: R foundation for statistical computing (2000)

[53] Vetter, J.S., Mueller, F.: Communication characteristics of large-scale scientific applications for contemporary cluster architectures. Journal of Parallel and Distributed Computing 63(9), 853–865 (2003)

[54] Yoo, A., Jette, M., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2862, pp. 44–60. Springer Berlin / Heidelberg (2003)

[55] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud 10 (2010)

Paper I

Towards Understanding HPC Users and Systems: A NERSC Case Study

Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan

Submitted 2017 *Extends two previously published papers of the same authors:*
1) HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems published at *the 24th International ACM Symposium on High-Performance Distributed Computing (HPDC)*, 2015
2) Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study published at *the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016

Towards Understanding HPC Users and Systems: A NERSC Case Study

Gonzalo P. Rodrigo[a,1], P-O Östberg[a], Erik Elmroth[a], Katie Antypas[b], Richard Gerber[b], Lavanya Ramakrishnan[b]

[a] *Dept. Computing Science, Umeå University SE-901 87, Umeå, Sweden*
[b] *Lawrence Berkeley National Lab Berkeley, CA 94720, USA*

Abstract

The high performance computing (HPC) scheduling landscape is changing. Previously dominated by tightly coupled MPI jobs, HPC workloads are increasingly including high-throughput, data-intensive, and stream-processing applications. As a consequence, workloads are becoming more diverse at both application and job level, posing new challenges to classical HPC schedulers. There is a need to understand the current HPC workloads and their evolution towards the future in order to perform informed scheduling research and enable efficient scheduling in future HPC systems.

In this paper, we present a methodology to characterize workloads and asses their heterogeneity, both for a particular time period and as they evolve over time. We apply this methodology to the workloads of three systems (Hopper, Edison, and Carver) at the National Energy Research Scientific Computing Center (NERSC). We present the resulting characterization of jobs, queues, heterogeneity, and performance that includes detailed information of a year of workload (2014) and evolution through the systems' lifetime. Among the results, we highlight the observation of discontinuities in the jobs' wait time for priority groups with high job diversity. Finally, we conclude by summarizing our analysis to establish a reference and inform future scheduling research.

Keywords: workload analysis, supercomputer, HPC, scheduling, NERSC, heterogeneity, k-means

1. Introduction

High performance computing (HPC) supports scientific research by providing capacity to run large simulations or solve large mathematical problems. Such applications largely rely on the tightly coupled MPI model, which, as a consequence, has dominated HPC workloads. However, the workload configuration is changing as HPC systems' use evolves. For instance, some scientific fields like biology or astrophysics increasing rely on analysis of large datasets. Also, as compute capacity keeps growing, simulations produce larger datasets that require analysis. Finally, semiconductor advances enable real experiments to produce higher resolution data that requires processing. Pushed by these use-cases, workloads are becoming more diverse, increasing the importance of high-throughput, data-intensive, and stream-processing applications. These applications differ in performance models and target objectives from the classical parallel tightly coupled. As a consequence, current HPC schedulers might not produce optimal decisions since they support diverse workloads which configurations differ from what batch schedulers were designed to support (uniform and MPI dominated).

Supporting the new workload landscape and its future evolution requires new scheduling models that need to be investigated. Such research must be informed by a characterization of the state, with a focus on diversity, of current workloads in HPC centers and their evolution. However, existing work on workload modeling [1] characterizes systems that are too old, too small, or not representative of the top HPC systems. Also, previous work did not focus on the workload diversity, a new trait present in recent workloads. Thus, there is a need to investigate the workloads in current HPC centers to understand users and applications requirements and project them in the future.

In this work, we present a methodology to characterize HPC workloads in detail. It includes classical workload analysis methods such as value distribution analysis on job variables (e.g., degree of parallelism or runtime), system utilization estimation, or overall wait time analysis. However, it includes an innovative method to analyze job geometry (allocated resources and runtime) diversity. This method employs k-means clustering to identify dominating groups of jobs in the workload according to their geometry (runtime and allocated CPU cores). Under this analysis, workloads with more job geometry clusters are considered more diverse, and vice versa. Also, job groups can be mapped on the waiting queues (priority categories) to analyze jobs diversity within each queue. Consequently, an analysis of the correlation of queue diversity and wait times for jobs of different sizes is performed. This analysis verifies that jobs' wait time match the expected values according to priority and system configuration (i.e., larger jobs should wait longer, higher priority jobs should wait

Email addresses: gonzalo@cs.umu.se (Gonzalo P. Rodrigo), p-o@cs.umu.se (P-O Östberg), elmroth@cs.umu.se (Erik Elmroth), kantypas@lbl.gov (Katie Antypas), ragerber@lbl.gov (Richard Gerber), lramakrishnan@lbl.gov (Lavanya Ramakrishnan)

[1]Work performed in part at the Lawrence Berkeley National Lab.

shorter) or if they deviate due to job heterogeneity.

We apply this methodology to the workloads of three systems (Hopper, Carver, and Edison) at the National Energy Research Scientific Computing Center (NERSC) [2]. The output of these analyses is a reference of the workloads of three systems representative of others at the HPC community: Carver is a terascale IBM high performance cluster built on commodity hardware supported by an Infiniband interconnect; Hopper is an early petascale Cray supercomputer based on AMD processors; and Edison is a more modern and energy efficient petascale Cray supercomputer based on Intel processors. The results include a detailed analysis on the jobs, queues, and system behavior in each year over their lifetime for the Hopper and Carver and in 2014's for Edison. Yearly data of Hopper and Carver is compared to produce a trend analysis that allows to observe the evolution of their lifetime. These results establish a first data-point to predict future HPC workloads to designing future resource management models.

Our workload analysis methodology can also be used to support informed short-term and long-term decisions at HPC centers. Periodical workload analyses can be aggregated in a growing trend analysis that can reveal changes in the user behavior and system performance. Also, the boundary geometries (i.e. runtime and degree of parallelization) in workload's job clusters might be used as a starting template to define priority groups (queues) to avoid mixed queues and minimize discontinuities in expected job wait time behavior.

Specifically, in this paper:

- We propose analysis methods to understand and compare the workload diversity: how self similar are jobs in the workload and their mapping on the prioritization queues.
- We define a method to analyze the wait time of jobs depending on their geometry, queue priority, and diversity.
- We provide a detailed job, queue, performance, and diversity characterization of the NERSC workload, and their evolution over time. The results allow to understand the users, system behavior, and the effect of queue heterogeneity on jobs' wait time.
- We present a summary of analysis results and compare them with characterizations of other existing HPC workloads.

The rest of the paper is organized as follows. We present background on HPC systems, scheduling, and workload analysis in Section 2. A high level description of our method and the analyzed systems is presented in Section 3. The details of the methodology and its application to the NERSC workloads are described in Sections 4 to 7. Finally, we provide a summary of our results together with conclusions in Section 8.

This work includes and extends previously published work from the same authors [3], [4].

2. Background

This section describes the challenges in the HPC community that motivate this work and presents background on parallel job scheduling and workload analysis relevant to understand our methodology and results.

2.1. Challenges in HPC scheduling

The challenges of resource management in HPC are changing. New application characteristics and technological shifts are bringing new concepts and requirements to the scheduling models and system architectures. In this section, we highlight some workloads' changes that stress the importance of our analysis methods.

Stream applications are becoming more present in HPC systems. Scientists conduct experiments that would benefit from real-time processing of large amounts of data on HPC systems (e.g. X-Ray Micro-diffraction on Advanced Light Source at LBNL [5]). Real-time processing could potentially be performed by providing resources through advance reservations. Advance reservations, however, have a negative impact on the overall utilization, showing the need for real-time scheduling (i.e. low-latency allocation of resources, with no previous reservation as a response to a real-time event). As another step in application evolution, scientific experiments in fields like biology, earth sciences, or high energy physics are increasingly relying on data analysis to extract useful information from large experimental datasets, or results from large simulations [6], [7]. These applications increase the importance of data-intensive computational models in HPC workloads, or the composition of different applications through workflows (e.g., simulation followed by results analysis). These changes motivate us to analyze the workloads at supercomputers to understand their current characteristics.

The importance of stream and data intensive applications point at an increasing diversity in workloads not only dominated by large tightly coupled parallel jobs. Diversity might affect the performance of the scheduler, which governs the execution of applications in HPC systems. For example, the impact of the scheduling decisions is different across applications: e.g. delaying one job belonging to a workflow may have a significant impact on its overall run time, while delaying a stream job that has to be rapidly scheduled might render it useless. Also, schedulers are unaware of the different architecture-related constraints in applications (e.g. I/O bound performance, loosely coupled jobs, and data locality). However, information about such constraints is required for the scheduler to perform optimal placement decisions to maximize the applications' performance. Understanding the impact of the application diversity on the system motivates our workload heterogeneity analysis.

2.2. Scheduling

HPC schedulers optimize job placement to achieve the highest system utilization possible with a reasonable turn-

System	Vendor	Model	Built	Nodes	Cores/N	Cores	Memory	Network	TFlops/s	Service
Hopper	Cray	XE6	2010	6,384	24	154,216	212 TB	Gemini	1280	Jan'10
Edison	Cray	XC30	2013	5,576	24	133,824	357 TB	Aries	2570	Jan'13
Carver	IBM	iDataPlex	2010	1,120	8/12/32	9,984	147 TB	Infiniband	106.5	Apr'10

Table 1: Edison, Hopper, and Carver characteristics

around time according to the job priority. The most common base technique in schedulers is FCFS (First-Come, First-Served) [8]. With FCFS, jobs are selected in order, reserving the associated resources required for a job. However, with FCFS, the scheduler has to drain the system in order to schedule a large job, leading to resource fragmentation that reduces the overall utilization. Thus, backfilling is normally used to move jobs forward to fill resource gaps produced by the FCFS. Backfilling provides an ordered search in the waiting queue to map jobs to empty resource windows even if they are not at the head of the queue [9].

The quality of the results of the backfilling algorithm depends on the user's wall clock time estimation [8]. If a job wall clock time is overestimated, the scheduler will assign an unnecessary large resource window, reducing the opportunities to schedule a job through backfilling. On the contrary, if wall clock time is underestimated (i.e., runs over its limit), the system will kill the job resulting in lost work. These effects motivate the jobs wall clock time accuracy (relationship between estimated and actual wall clock time) characterization presented in Section 4.2.

Finally, a job's turnaround time depends on its priority (influencing its progress on the scheduler wait queue in each scheduling pass), geometry (jobs requiring more resources are harder to schedule), and requested resource load (how many jobs compete for the same resources). However, job diversity in the queues might affect this relationship. In Section 6.2 we present an analysis of the possible impact of these factors (including job diversity) on the job's wait time.

2.3. Related work on workload analysis

Previous work on scientific Grid and HPC workloads characterization is found in the Grid Workloads Archive [10] and the Parallel Workload Archive [1]. The archives contain job and performance characteristics (run time, parallelism, inter-arrival time, wait time, disk space, and memory), but their analyses overlook the workload heterogeneity. Also, analyzed systems are either at least 10 years old or significantly smaller than the current top HPC systems. Our work extends them by addressing jobs' heterogeneity and performing analyses on large, more recent systems (e.g. Edison was deployed in 2014 and still ranks 60 in the Top 500 list in February 2017).

Job heterogeneity has been observed in industrial workload diversity analysis [11], which introduces k-means as a tool for job similarity clustering. This work inspired our workload diversity analysis method for HPC workloads, which we complemented with per queue analysis and a new methodology to compare the degree of heterogeneity across systems and system states.

A previous analysis [12] on the applications run on Hopper in 2012 characterizes the importance of the different applications run on the system, with an initial insight on the jobs' geometry and memory requirements.

3. Methodology

In this section we describe the three systems analyzed (their characteristics, workload, scheduling model, and configuration), our data source (size, time span, format), analysis framework (motivation for analyzed variables), and trend analysis methodology.

3.1. System descriptions

In this work, the results of characterizing the workload of three HPC systems are presented. In this section, we describe the characteristics of these systems to enable the discussion about the applicability of our results to other systems.

3.1.1. System characteristics

NERSC is a HPC center at Lawrence Berkeley National Lab, that has the mission to provide computing infrastructure and tools for scientists performing research of relevance to the DOE (Department of Energy). Our work analyzes the various years of real jobs from three of NERSC's systems: Carver, Hopper, and Edison. These three systems were selected because their different hardware characteristics and origin in the timeline of HPC systems evolution, which can be observed in Table 1. Carver is a terascale IBM iDataPlex Linux cluster [13] deployed in April 2010. Its configuration is the closest to commodity hardware servers of the three systems and it is supported by an Infiniband interconnect. Hopper and Edison are specialized Cray supercomputers with custom interconnects [14]. Hopper is a petascale Cray XE system, based on AMD processors and a Gemini interconnect, and deployed in 2010. Edison is a newer, more power efficient petascale Cray XC30, constructed with Intel processors supported by a Aries interconnect and deployed in 2014. Thus, these systems allow us to capture the workload characteristics of high-end clusters and supercomputers, belonging to different HPC system generations and optimized for slightly different applications.

On the resource management side, all three systems use the Moab scheduler [15, 13, 16] running atop the Torque resource manager [17]. Edison's workload manager was replaced by Slurm at the end of 2015.

3.1.2. Workload

Over 5000 users and 700 distinct projects use NERSC resources [18, 12]. The workload is composed of applications from various scientific fields like Fusion, Chemistry, Material Science, Climate Research, Lattice Gauge Theory, Accelerator Physics, Astrophysics, Life Sciences, and Nuclear Physics.

In addition to serving typical MPI workloads, Carver provides a *serial* queue [19]. The serial queue allows users to submit and execute jobs with a very low degree of parallelism (i.e., one single core). Carver has 80 compute nodes allocated to serial jobs. Serial queues were added to Hopper and Edison in late 2014. The serial queue on Edison and Hopper is configured via a super-job run under the special Cluster Compatibility Mode (CCM). There are a total of 15 compute nodes each allocated to run serial jobs on Edison and on Hopper. Serial queues contain jobs running long time (limited to 48 hours) on a single core. The purpose of the serial queue is to increase resource utilization density. It serves packs jobs on the same node that do not benefit from parallelism and which performance is either not critical or rarely affected by resource sharing.

The conclusions of this work are only based on the job related information of the workload. Run time characteristics of applications, execution schema or other variables were not considered or analyzed in this study.

3.1.3. Scheduler characteristics

The configuration of a system scheduler has an impact on the system performance (i.e., utilization, wait time) and the workload shape: e.g., jobs allocation sizes will cluster around the allowed values in submission queues. In this subsection, we present the configuration of the analyze systems scheduler to provide context for later analyses.

First, node sharing is only enabled for nodes executing jobs from the *serial* queue to avoid performance degradation [20]. In order to keep the same baseline, we consider *cores* as the degree of parallelism unit in our analysis.

In all systems there is a distinction between the queues chosen at submission time (Torque) and the queues that the scheduler use for priority calculation (Moab). Users submit jobs to the Torque *submission queues.* Moab has its own queue configuration - the *execution queues.* Torque translates the queue into Moab's execution queues and passes the job to the scheduler. Submission queues can be mapped to a single or multiple execution queues. For example, jobs of up to 10 hours of runtime maybe submitted to the same submission queue, to be sorted into two execution queues with ranges of $[0, 5)$, $[5, 10)$ runtime hours. Table 2 shows queue properties that govern the scheduling decisions for our three systems. The properties are explained below:

Maximum wall clock time (Torque): Each queue has an upper limit for a job's estimated wall clock time specified by the user at submission time. If a job's estimated wall clock time is longer than this limit, submission fails. If a job runs longer than the user estimated wall clock time, the job is terminated.

Number of cores (Torque): Each queue has a predefined minimum and maximum limit of a job's requested number of cores. Submission of a job allocating a number of cores outside this range will fail.

Queue priority (P) (Moab): Each queue in the system is assigned a priority (represented as an integer where a higher number represent a higher priority).

Eligible jobs limit per user (E) (Moab): Only the first E jobs of the same user in the same execution queue are eligible for scheduling. This can affect a job's wait time. For example, if a user would submit 25 jobs to the serial queue on Carver, only the first 20 jobs will be considered for scheduling. The last five jobs will only be considered to be scheduled after the first five jobs have finished. This can impact wait times for the jobs where the last five jobs may have significantly higher wait times than the other 20 jobs.

The execution queues do not exist as separate data structures inside Moab. All jobs are stored in a single queue. When a job is passed to Moab, it is inserted in its job waiting queue with a job priority of zero. In every scheduling pass, the job priority is recalculated by adding a value, which depends on the associated execution queue priority. If a job is in a higher priority queue, the job priority will grow faster and it will be eligible for execution more quickly. The analysis of the impact of the queues' characteristics on jobs wait time is presented in Section 6.2.

3.1.4. Queues configuration

The analyzed system's scheduler re-calculates the jobs priority depending on the queue they are submitted to. Since the configuration of such queues affect the overall system behavior, we present their configuration in detail in this section.

Table 2 presents the execution queue configuration of Edison, Hopper, and Carver used in the analysis. It covers each queue's job maximum run time (Wall Clock Time), job allowed allocations in numbers of cores (Cores), number of eligible jobs allowed to be scheduled simultaneously (E), and the priority of the queue (P). This information allows us to understand the reasons for different wait time behaviors between queues.

The batch queue policies influence the jobs execution order. These policies changed slightly through the studied period. To simplify the analysis, we used the settings that were most common through the period of study. Also, some queues were filtered out of this study as they represented too little of the workload, or were related to system maintenance or tests.

Edison and Hopper map their queues on a single set of resources (independent for each system). However, Carver queues are mapped in sets that partly overlap: general set (1080 nodes), *matgen* set (64 nodes, subset of the general set), *xlmem* set (two nodes with large memory capacity), and *serial* set (80 nodes, not overlapping). Different

Hopper Queues	Wall clock	Cores	E.	P.	**Edison** Queues	Wall clock	Cores	E.	P.	**Carver** Queues	Wall clock	Cores	E.	P.
bigmem	24h.	1-8,856	1	0						matgen_low	Unk	1-256	66	0
ccm_int	30m.	1-12,288	2	1	cm_int	30m.	1-12,288	2	1	matgen_prior	Unk	1-256	66	10
ccm_queue	96h.	1-12,288	16	1	ccm_queue	96h.	1-16,368	16	0	matgen_reg	Unk	1-256	66	1
debug	30m.	1-12,288	2	1	debug	30m.	1-12,288	2	1	debug	30m.	1-256	1	2
low	48h.	1-16,392	6	-3	low	48h.	1-16,392	6	-3	low	24h.	1-256	3	-2
premium	48h.	1-49,152	1	2	premium	36h.	1-49,152	1	2	xlmem_sm	72h.	8	1	0
reg_1hour	1h.	Unk.	8	0	reg_1hour	1h.	Unk.	16	0	xlmem_lg	72h.	32	2	0
reg_big	36h.	49,153-98,304	2	1	reg_big	36h.	49,153-98,304	2	1	reg_big	24h.	257-512	1	0
reg_long	96h.	1-1,536	4	0						reg_long	168h.	1-128	1	0
reg_med	36h.	16,369-49,152	4	1	reg_med	36h.	16,369-49,152	8	1	reg_med	36h.	129-256	2	0
reg_short	6h.	1-16,368	16	0	reg_short	6h.	1-16,368	24	0	reg_short	4h.	1-128	4	0
reg_small	48h.	1-16,368	16	0	reg_small	48h.	1-16,368	24	0	reg_small	48h.	1-128	3	0
reg_xbig	12h.	98,305-146,400	2	0	reg_xbig	12h.	98,305-131,088	2	1	reg_ xlong	504h.	1-32	1	0
										interactive	30m.	1-64	1	2
thruput	168h.	1-48	500	0	killable	48h.	1-16,368	8	0	serial	48h.	1	20	-

Table 2: Hopper, Edison, and Carver queue characteristics. Jobs have to be within certain limits to be accepted in a queue: requested runtime upper limit (wall clock time) and accepted number cores range (Cores). Eligibility (E.): Maximum number of jobs from the same user in the same queue which are considered in jobs priority recalculation. Priority (P.): Queue priority.

queues have access to different sets: *matgen* queue jobs can only run on *matgen* resources (but jobs from other queues can use them when they are available). *xlmem* nodes can be only used by *xlmem* jobs. This implies that different queues may not present the same ratio of job core-hours requested over resources' core-hours available. How this difference may impact jobs wait time is studied in Section 6.2.

The *serial* queue jobs allocate one core per job and are executed on shared nodes (more than one job per node). Also, this queue has exclusive access to the *serial* resource set so it does not compete with any other queue. Thus, wait times of the *serial* queue should not be compared with other queues.

3.2. Data Source

All workload analysis is performed on the job summary entries from the systems' Torque logs. The data includes 1 year and 1,357,366 jobs for Edison, 4.5 years and 4,326,870 jobs for Hopper, and 4.5 years and 9,508,054 jobs for Carver. The raw data size is 45 GB, which, after filtering and parsing, is reduced to 6 GB of net data.

3.3. Analysis Framework

The analysis framework is composed of a set of scripts that express the data pipeline to process the log data. The developed data pipeline is divided into three parts. The first is the data extractor, which retrieves the log files from the NERSC repository, parses them, eliminates invalid entries and inserts them in a MySQL database. The second component is a Python API to insert, manipulate, and retrieve the data from a MySQL database. The MySQL database is indexed to facilitate the queries based on multiple fields. The third component is the analysis toolkit. It implements the logic to retrieve, analyze, and visualize the data for all the analyses. A specific plotting library was developed to support the graph generation. The code consists of 14K Python lines using the scientific libraries SciPy and NumPy combined with the plotting library MatPlotLib [21]. All analyses were run on an Intel i7 Quad core 8 GB RAM desktop computer. The database is hosted on a department server at Berkeley Lab.

Our analysis focuses on understanding the variables of the workload from the user (i.e., job) and system (i.e., queues and performance) perspectives.

The job perspective includes:

Job size: includes wall clock time, degree of parallelism, and resulting compute time allocation. These parameters define the system boundaries' requirements and job granularity.

Wall clock time accuracy: represents how accurate are the user estimations on the jobs runtime. The variable measures the quality of the information used by the scheduler in its job planning.

Inter-arrival time: models the time between the submission of two jobs. It represents the load to be managed by the scheduler and the overall wait time. For instance, for the same job sizes, a smaller inter-arrival time represents a larger job load.

Job diversity: measures how different the geometries of jobs in the workload are. It includes the analysis of dominant job geometries in the workload.

The queues and their configuration represent the mapping of the prioritization policies to the workload job mix. The study includes:

Queue significance: represents the impact of each queue on the overall system. It allows to understand how the properties of each queue contributes to the overall system behavior according to their importance.

Queue job diversity: analyzes the how self similar are

the jobs within each queue in terms of geometry. This analysis is relevant because execution queues allow the system scheduler to prioritize jobs depending on their geometry. However, the existing queues might not represent the most significant types of jobs (by geometry) present in the job mix. This analysis is focused on two objectives: understanding the diversity of the jobs across the entire workload, and similarity of jobs contained in the same queues.

The performance perspective covers the system utilization and job wait time. Additionally, the job wait time is studied from system, queue, and job geometry points of view. This study allows us to understand the how the effectiveness of the priorities for different job geometries in different queues might be affected by job diversity in the queues.

3.4. Trend analysis

In this work, the workload of Carver and Hopper was analyzed in detail for each year of their lifetime. To simplify interpreation, the detailed analysis is only presented for 2014. However, the relevant results of all years were aggregated in the trend analysis, presenting the evolution of Carver and Hopper workloads through their lifetime. Edison's trend analysis was not performed because not enough workload data was available.

In this work, we focus on understanding the evolution of the system's workload, overall performance, and user behavior. As explained in Section 3.3, workload trend covers the evolution of the job geometry (wall clock time and degree of parallelism). Overall performance is analyzed through the evolution of job wait time. User behavior is analyzed by observing the evolution of the wall clock time accuracy.

Finally, since the trend is performed by analyzing the workload in sequential time periods and aggregating the results over a time line, an adequate period had to be chosen. The size of the workload periods is calculated by detecting repeating user patters in the workloads through Fourier transform analysis on the number of tasks submitted per hour [22]. Results of such analysis are described in Section 7.1 and the dominant detected cycle was one year.

4. Job Characterization

In this section we present the workload analysis of the jobs of Edison, Hooper, and Carver in 2014. This analysis is performed with special attention to job geometry, user submission patterns, and job diversity on all three systems.

4.1. Job geometry

Job's geometry analysis allows to observe the patterns in jobs resource allocation and analyze the job mix that the scheduler manages. All variables are analyzed by calculating their value distribution and consequence Cumulative Distribution Function (CDF). This allows to understand

Job Distribution	Edison	Hopper	Carver
%Jobs Wall Clock $<$ 2 h.	88%	86%	87%
%Jobs Width $<$ 240 codes	69%	75%	99%
%Jobs Width $\leq$ 1 Node	39%	37%	92%
%Jobs Alloc. $\leq$ 1 core-h.	19%	26%	77%
%Jobs Alloc. $\geq$ 1K core-h.	7%	8%	$\sim$8%

Table 3: Detailed job characteristics distribution analysis

if jobs are dominantly small or large, if theirs sizes concentrate around certain values, or if the job mix includes enough jobs of smaller sizes to allows high system utilization. We follow to present the results for each of the job variables.

Job wall clock time. Figure 1a, shows the Cumulative Distribution Function (CDF) of the job wall clock time for the three systems in 2014. In the case of Hopper, we observe jobs running up to 160 hours, with a high concentration running under two hours. Table 3 shows an overview of the job characteristics' distribution analysis. It shows that 86-88% of the jobs on all three systems run for less than two hours. For Carver, a large number of the jobs have a wall clock time well under one hour; in fact, 60% of the jobs run for less than 13 minutes.

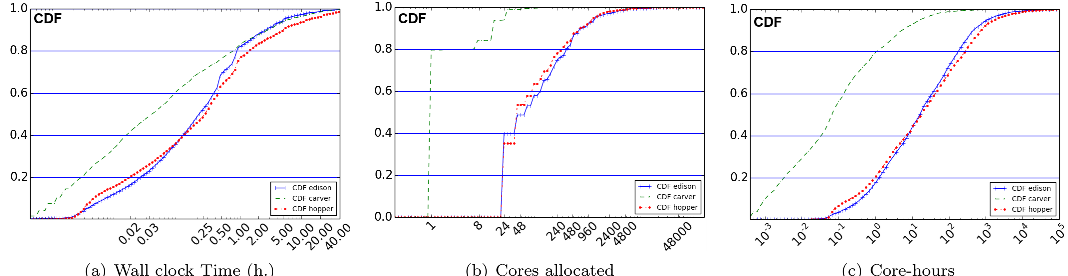
Additionally, all three CDFs present steep slopes around 30 minutes and 6, 12, 24, and 36 hours (better observed in a non-log scale version of the graphs), numbers that are similar to the queues' configured wall clock time limit. These limits are similar across the three machines (more details in Table 2).

Cores per job. Figure 1b presents the distribution of cores allocated to jobs on the three systems. It represents the number of cores requested and allocated to a certain job, and does not include any information on the actual usage of the cores. On Hopper and Edison, requests for a single job range from 24 (1 node) to over 100,000 cores (i.e. close to the full capacity of the systems). A small number of cores are requested for Hopper's jobs: 75% under 240 cores (10 nodes), and 37% of all jobs run on a single node (Table 3). Edison presents a similar pattern with 69% of the jobs running on less than 240 cores and 39% on a single node. Carver shows a different trend from Edison and Hopper. On Carver, many jobs run on a small number of cores: 99% run on 240 cores or less, and 92% of all jobs run on a single node.

Allocated core-hours per job. Figure 1c shows core-hours allocated for the jobs in the system. The figure shows that Hopper and Edison core-hour allocations are similar. Jobs on Hopper and Edison are significantly larger than those on Carver: 99% of Carver jobs individually consume less than one core-hour, in comparison with 42% on Edison and 46% on Hopper. On the other extreme, we can observe that almost 10% of Edison and Hopper jobs individually consume more than 1,000 core-hours.

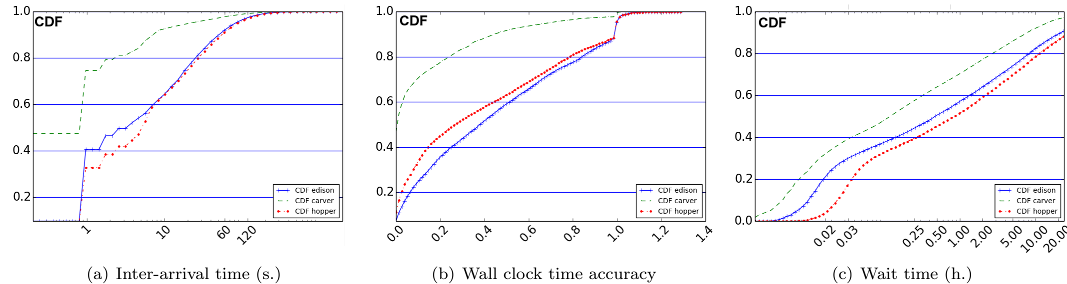
4.2. Job's characteristics

In this section we study other variables of the jobs that depend on other external agents. This includes the user's



(a) Wall clock Time (h.) (b) Cores allocated (c) Core-hours

Figure 1: Job geometry characterization on Hopper, Edison, and Carver. a) Significant percentage (Edison: 87%, Hopper: 82%, Carver: 87%) of the jobs run for 2h or less. b) 69% of Edison, 75% of Hopper and 99% of Carver jobs allocate 240 cores or less. c) Carver's jobs allocate significantly fewer core-hours.



(a) Inter-arrival time (s.) (b) Wall clock time accuracy (c) Wait time (h.)

Figure 2: Job characterization on Hopper, Edison, and Carver. a) Carver receives significantly more job submissions per time unit than the other systems: 40% of jobs are followed by another job within one second. b) 11% of Edison and 10% of Hopper jobs run over the requested time. Carver: 92% of the jobs run under 50% of requested wall clock time. c) Jobs that wait less than 3h to be executed: Edison (67%), Hopper (60%), Carver (79%).

submission pattern (inter-arrival time and wall clock time accuracy) and job's overall wait time (which depends on the scheduler configuration). A more detailed analysis of the job's wait time is presented in Section 6.2.

Inter-arrival time. Figure 2a represents the CDF for the inter-arrival times on Edison, Hopper, and Carver. The inter-arrival time measures the time elapsed between the arrival of consecutive jobs in a system, which can affect the granularity of scheduling and help to understand the load on the schedulers.

Edison and Hopper have very similar distributions: 90% of the jobs have inter-arrival times under two minutes. The remaining 10% are distributed in the 1500-2000 seconds (25-33 minutes) range. On the other hand, more than 95% of Carver's inter-arrival times are under 25 seconds. As the CDFs of the three systems are compared, Carver's inter-arrival times are shorter than those for Edison and Hopper. Thus, when compared to Edison and Hopper, we observe that more jobs are submitted to Carver queues during the studied time period.

Wall clock time accuracy. For each job we study the difference between the actual and the requested wall clock times. The accuracy is defined as $\frac{W}{W^r}$, where W is the actual wall clock time of a job and W_r is the wall clock time that the user requested for the job. The accuracy will be close to one when the estimation is good, and closer to zero when the job running time is overestimated. If the job runs over the requested time, the job will get preempted. However, this is caught during the next scheduling pass. Thus, we see values over one when jobs run over the estimated time.

As discussed in Section 2.2, the wall clock time accuracy affects the backfilling decision quality. Figure 2b presents the distribution of the wall clock time accuracy values for the three systems studied. The initial steep slope of the Carver CDF shows that it executes many jobs that use much less than the requested wall clock time. Edison and Hopper have a more linear CDF for values between zero and close to one. However, in all systems we observe jobs with an accuracy slightly above one (as they exceed their allocated run time and are terminated). We see that the percentage of jobs that run out of wall clock time is higher on Edison (11%) and Hopper (10%) than on Carver (2%). Approximately 60% of Edison and 66% of Hopper jobs run 50% or less of the requested time. On Carver, around 93% of the jobs run 50% or less of the requested

time.

Wait time. Figure 2c presents the distribution of job wait times under 24 hours (jobs with longer wait times are not included in this graph). The figure shows that Hopper has more jobs with longer wait times, followed by Edison and Carver. Considering all the jobs in the system, we see that 61% of Hopper jobs, 67% of Edison jobs, and 80% of Carver jobs have a wait time of less than three hours. Further analysis of the wait time values is presented in Section 6.2.

4.3. Job diversity

The job diversity analysis is based on a machine learning technique and extends a previous work about job grouping within clusters [11]. We construct job geometry tuples that contain job wall clock time and number of cores allocated. Before performing analysis, the tuples are normalized (whiten [23]) to reduce the effect of the value magnitudes on the clustering process. In the analysis, the target is to find the smallest number of k-means clusters [24] among the job geometry tuples where the variation coefficient (standard deviation divided by the mean) is at most 1.1. If jobs are similar, the method will group them in a small number of clusters. Jobs in more diverse workloads are grouped in larger numbers of clusters.

When invoked, k-means produces k clusters from an input dataset and a list of k centroids used as search starting points. However, k-means produce k clusters, not the minimum possible, and does not guarantee that the clusters are not disperse. As a consequence, k-means must be invoked with different k sizes and different start centroids to obtain a minimum possible number of non dispersed clusters. Figure 3 illustrates the algorithm that searches for the minimum clusters in the job geometries, by trying different k values and random centroid points.

It starts by whitening the input job tuples (line 5) to reduce the effect of the value sizes on the clustering. Then, the process to find the minimum number of k-means minimum cluster search is repeated 10 times with different starting centroids. In each trial (lines 7-37), the process starts by producing an initial random centroid set of two points. Then, starting at $k = 2$ the algorithm tries to find k non dispersed clusters, increasing the value of k in each unsuccessful trial (lines 10-37). For each k, k clusters are produced (line 14) and considered *disperse* if their variation coefficient is larger than 1.1 (lines 15-23). Each disperse cluster is divided, i.e., two centroids close to the cluster centroid are produced and added to the obtained cluster lists, increasing the resulting k size in one (line 21). In summary, the obtained clusters are reviewed, if they are not too dispersed, they are left as they are, otherwise they are split, increasing k in one per cluster split. Then, the search is repeated with the new centroids as starting points. The process is repeated until non dispersed clusters are found (lines 26, 27) or the found k is larger than the minimum size of previously observed non dispersed clusters (lines 11,12). The algorithm does not guarantee that the obtained number of clusters is the minimum possible, however it produces a local minimum.

Figure 3: Minimum number of k-means cluster search algorithm for a list of job geometries.

```
 1: (repetitions, trialsSmallerK) ← (10, 10)
 2: maxCulsterVar ← 1.1
 3: minKFound ← −1
 4: (finalClust, finalCent) ← (None, None)
 5: normJobs ← whiten(allJobs)
 6: for i ← 1, repetitions do
 7:     seed = genRandomSeed()
 8:     k ← 2
 9:     cent ← genRandomCentroids(k, seed)
10:     for j ← 1, trialsSmallerK do
11:         if k >= minKFound and minKFound ≠ −1 then
12:             break
13:         end if
14:         Clust, cent ← kMeans(normJobs, cent)
15:         cvList ← calcCVForClusters(Clust, cent)
16:         newCentroids ← []
17:         for l ← 0, len(cvList) do
18:             if cvList[l] <= maxCulsterVar then
19:                 newCentroids.append(cent[i])
20:             else
21:                 newCentroids.append(splitCentInTwo(cent[i]))
22:             end if
23:         end for
24:         if len(cent) = len(newCentriods) then
25:             (finalClust, finalCent) ← (clust, cent)
26:             if minKFound = len(finalCent) then
27:                 break
28:             end if
29:             if minKFound = −1 then
30:                 minKFound ← len(finalCent)
31:             else
32:                 minKFound ← min(k, len(finalCent))
33:             end if
34:         end if
35:         cent ← newCentroids
36:         k ← len(newCentroids)
37:     end for
38: end for
39: return finalClust, finalCent
```

Figure 4 shows the results of the clustering search method for Edison's jobs in 2014. This graph is a scatter plot of the jobs where each job is represented by a colored dot. The x-coordinate corresponds to the job's wall clock time and the y-coordinate to the number of cores allocated to the job. Note that the y-axis is in logarithmic scale. The execution queue of the job is identified by the color of the dot. The clusters' centroids are represented by black dots, while the color boxes are the boundary jobs observed in each cluster (minimum and maximum wall clock time and number of cores).

Table 5 shows the results of the clustering. Eight clusters were found for Carver, 11 for Edison and 12 for Hopper. This implies that Carver has a more homogeneous job set compared to Edison and Hopper As noted in the previous section, 70% of Carver jobs come from the *serial* queue,

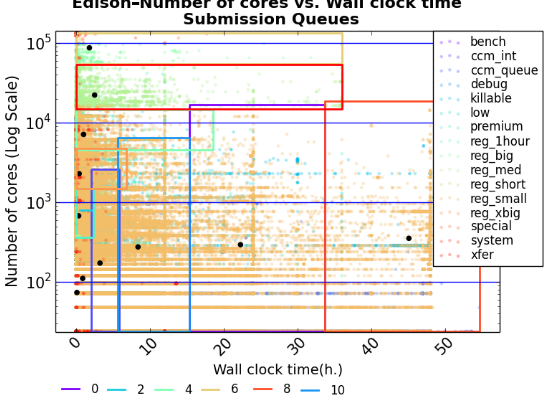


Figure 4: Result of the job clustering method for Edison 2014 with 8 clusters. Jobs are mapped on queues and clusters: Each dot is a job and dot color indicates the queue. Black dots are cluster centroids and color boxes are the surrounding jobs belonging to the same cluster. Clusters are sets of jobs with similar geometry.

defined for single core jobs with long run times.

5. Queue Characterization

In the analyzed systems, job priorities are calculated depending on the queues that they are assigned to, also different job geometries were limited to be submitted to certain queues. As a consequence, queue configuration has an effect on how user submit jobs and how job priorities are distributed. In this section, we analyze the relationship between queue configuration, job geometry, and submission behavior.

5.1. Queue significance

The first step in analyzing the queue configuration and behavior is to understand the relevance of each queue in the system. For this, Figure 5 shows the normalized view of the number of jobs and core-hours per execution queue on Edison, Hopper, and Carver. The interpretation of this data is based on the configuration presented in Table 2.

On Hopper, the *debug* queue contains approximately 20% of the total jobs. The *debug* queue is typically used for testing and has a wall clock time limit of 30 minutes. The queue *reg_small* contains 30% of the jobs and approximately 56% core-hours. The queue *reg_med* presents a lower job count ($< 5\%$) and core-hours ($\sim 15\%$). The *thruput* queue admits 168 hours jobs and multiple submissions from the same user (600). However, its contribution to the total system utilization is less than 10% of jobs and 2% of core-hours.

Edison is very similar to Hopper, with the *reg_small* queue having $\sim 40\%$ of the jobs, and $\sim 40\%$ of the core-hours contributed by the *debug* and *reg_1h* queues.

Carver shows a different pattern. The *serial* queue contains more than 70% of the jobs, which consumed less

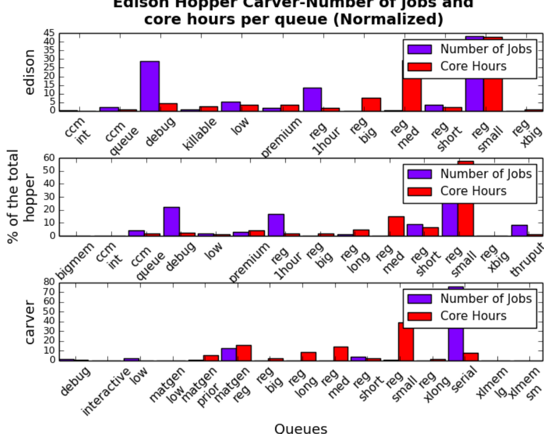


Figure 5: Normalized view of the number of jobs and core-hours per queue in Edison, Hopper, and Carver.

than 10% of core hours. This percentage matches with the fact that this queue has exclusive access to 80 out of 1,180 total nodes ($\sim 7\%$) that are used for computation. The *reg_small* queue ($\sim 40\%$) and *matgen_reg* ($\sim 15\%$) account for the majority of the remaining core-hour usage.

5.2. Queue Diversity

The minimum number of clusters and their boundaries provide an overall view on the job diversity, and a possible recommendation for queue configuration. However, it gives no information about the job mix for a queue, i.e. how similar are the jobs in each queue. Thus, we define the *queue homogeneity index* as a new metric to compare the diversity of the jobs in a queue. After the clustering process, it is possible to identify each job's original cluster. Since jobs from different clusters are significantly different, a queue which jobs largely map on a single cluster will contain more homogeneous jobs than a queue whose jobs map to many clusters. The *queue homogeneity index* is the percentage of queue jobs that are mapped to the queue's dominant cluster, i.e. the cluster to which most jobs of the queue belong. For example, a queue maps jobs to three clusters with the following shares: 20%, 30%, and 50%. The cluster that contributes the most has 50% of the jobs and, as consequence, the *queue homogeneity index* is 50%. A higher index value indicates that many jobs are mapped to the same cluster and are thus geometrically similar to each other, while a lower index value means that the queue's jobs are more heterogeneous.

Table 5 present the queue homogeneity indices for the all the queues in Edison, Hopper, and Carver. This data allows to identify which queues contain more diverse job mixes and are candidate to be reviewed and, if needed, divided into smaller better defined queues. In Edison, reg_big, reg_med, and reg_big are the more homogeneous

Edison	11 c.	Hopper	12 c.	Carver	8 c.
Queue	/1	Queue	/1	Queue	/1
ccm_queue	0.46	bigmem	0.31	debug	0.32
debug	0.63	ccm_queue	0.45	interactive	0.35
killable	0.40	debug	0.70	low	0.99
low	0.53	interactive	0.85	matgen_low	0.62
premium	0.27	killable	0.45	matgen_prior	0.66
reg_1hour	0.71	low	0.59	matgen_reg	0.68
reg_big	0.96	premium	0.40	reg_big	0.70
reg_med	0.98	reg_1hour	0.69	reg_long	0.31
reg_short	0.42	reg_big	0.66	reg_med	0.82
reg_small	0.42	reg_long	0.50	reg_short	0.62
reg_xbig	1.00	reg_med	0.86	reg_small	0.26
		reg_short	0.39	reg_xlong	0.55
		reg_small	0.36	serial	0.87
		reg_xbig	1.00	usplanck	0.54
		thruput	0.77	xlmem_lg	0.24
				xlmem_sm	0.58

Table 4: Queue homogeneity indices for each machine: share of number of jobs belonging to its dominant cluster. In light green queues with indices in $(0.50, 0.75]$ interval, in darker green queues with indices $(0.75, 1.00]$.

queues while premium is the more diverse (0.27). In intermediate values (close to 0.50), reg_small appears to be diverse (0.42) and since its impact of the system is large (40% of jobs and core-hours), it is a good candidate of further study for division. Among Hopper's queues reg_small is quite diverse (0.36) and significant in the system (50% of jobs and core-hours), becoming a good candidate for revision. In Carver, reg_small (0.26) and significant (40% of core-hours) and should also reviewed.

Systems have different queue configurations and, in order to compare different systems with different workloads, a global metric is established. It aggregates the *queue homogeneity index* of all the queues by taking into account the queues' importance relative to the entire workload in the system.

Figure 5 presents two criteria for the impact of a queue on the system: number of jobs contained and amount of core-hours contributed. The former is useful to understand scheduler behavior, while the latter represents the fraction of the machine time the queue occupies. Missing one aspect of the system may give an incomplete picture, so we define two more metrics:

Job homogeneity index is calculated as a linear combination of the queue's homogeneity index. The coefficients are the share of jobs contained by the corresponding queue. For example, Queue1 has a homogeneity index of 0.6 and contributes 30% of the system's jobs. Queue2 has a homogeneity index of 0.4 and contributes 70% of the jobs. The *Job homogeneity index* is thus calculated as: $0.6 \cdot 0.3 + 0.4 \cdot 0.7 = 0.46$

Time homogeneity index is calculated as a linear combination of a queue's homogeneity, but in the time dimension. The coefficients are the shares of core-hours contributed to the system by the corresponding queue. In the example of Queue1 and Queue2: Queue1 contributes 30% of the system's jobs, and represents 80% of the core-hours of the system. Queue2 contributes 70% of the jobs that represent 20% of the system's core-hours. The *Time homogeneity index* is thus calculated as: $0.6 \cdot 0.8 + 0.4 \cdot 0.2 = 0.56$.

	Edison	Hopper	Carver
Clusters	11	12	8
Job homogeneity idx.	0.51	0.57	0.82
Time homogeneity idx.	0.64	0.49	0.51

Table 5: Queue analysis results: for each machine, minimum number of k-mean clusters discovered in the jobs and homogeneity indices.

We understand that larger indices imply that jobs in queues are more homogeneous, and policies are able to do more precise job prioritization for certain types of jobs. A lower index implies the existence of queues that contain a diverse job mix, which probably should be divided in more narrowly defined queues. The queue homogeneity index defined above can be used to determine what and how queues should be modified. These indices are not absolute measurements, and can only be used to compare similar systems (like the ones studied) or the same system during times.

Table 5 shows the calculated *homogeneity indices* for the three NERSC systems. The job homogeneity index shows that Carver is the system in which queues contain a reasonably uniform job mix. The time homogeneity index produces a different ordering: Edison, Carver, Hopper. This implies that the uniform queues on Carver have many jobs, but they are very small in terms of the total number of core-hours. On Hopper the uniform queues contains fewer jobs, but they contribute a significant part of the system's core-hours.

This analysis can be used to improve the job sorting across queues. From the point of view of core-hours, Edison has the best sorted queues. The queue *reg_small* is the largest contributor in terms of core-hours and jobs, however, only 42% of the queue jobs are mapped to the same cluster, implying a high diversity within the queue. The *reg_small* queue allows jobs up to 48 hours long and between 1 and 16,368 cores in size. Dividing this queue into subqueues with sub ranges of wall clock times or cores could have a positive impact on the time homogeneity index of Edison, and support improved job prioritization on the system.

6. Performance Characterization

In this section we study the performance of the systems from the perspectives of both resource providers (utilization) and users (wait time). This study includes a more detailed analysis on the jobs' wait time with a focus on its relationship with queue organization and job diversity.

6.1. Utilization

Facilities such as NERSC report the utilization of their resources periodically. The 2014 NERSC report calculates

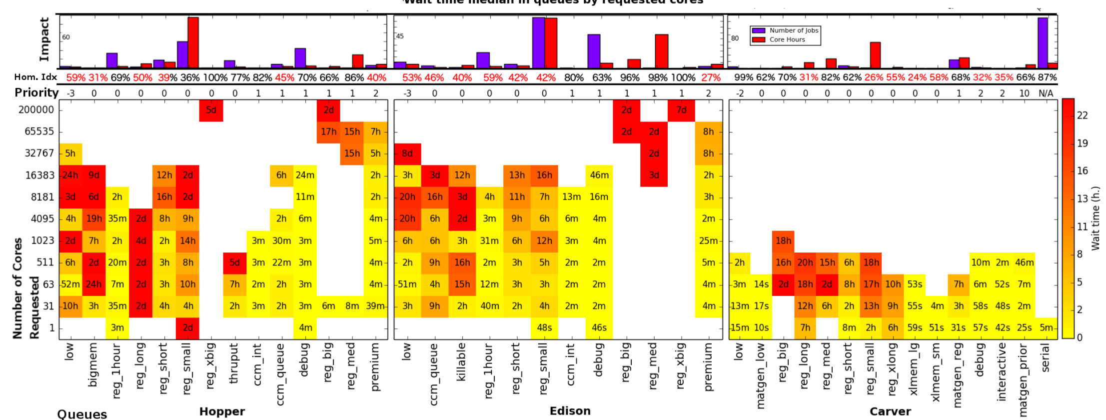


Figure 6: Job wait time median per queue depending on the requested cores. Aggregated on top: priority per queue, median of the wall clock time of queue jobs, jobs per queue (normalized), core-hours per queue (normalized).

System	Edison	Hopper	Carver
TSU	0.91	0.90	N/A
tTSU	0.87	0.80	0.88

Table 6: Total System Utilization (TSU) and theoretical Total System Utilization (tTSU). tTSU is under TSU since it does not take into account system maintenance down times.

the Total System Utilization (TSU) of Hopper and Edison as:

$$TSU = \frac{\text{core-hours used in period}}{\text{core-hours available in period}} \tag{1}$$

The available core-hours are calculated subtracting maintenance time (full and partial) and other temporal resource reductions. Down times are tracked manually and the TSU is calculated for reporting reasons. The available logs for this analysis do not contain system availability information. Thus, we calculate the *theoretical Total System Utilization* (tTSU) as:

$$tTSU = \frac{\text{core-hours used in period}}{\text{time period} * \text{maximum system capacity}} \tag{2}$$

By definition, tTSU will be less than or equal to TSU. We present the reported TSU and the tTSU in Table 6. Carver's TSU was not available for 2014.

6.2. Job wait time

Previous analysis (Section 4.2) presents a coarse grained analysis of jobs' wait time. To understand the performance of the system, it is important to also understand wait times relative to job geometry and queue priorities. In this section, we detail our wait time distribution analysis, i.e. we calculate the median wait time of jobs grouped by queues (and thus corresponding priority) and job geometry. User wall clock time estimations of the systems are inaccurate (Section 4.2), as a consequence we use the number of cores requested as job geometry metric for this analysis.

We present job wait time for the three systems as heat maps in Figure 6. For each system, queues appear on the x-axis, ordered by priority. The y-axis represents a non-linear categorization of the possible numbers of cores allocated to jobs. Each square contains the wait time median (in seconds, minutes, hours, or days) for a particular queue that was allocated the cores specified on the y-axis. White regions indicate that there were no jobs with the specific queue and cores combination. A darker tone or red represents longer wait times (24 hours or longer), and a lighter tone or yellow represents shorter wait times. The priority of each queue is specified above the heat map and below the bar graph. The bar chart on top shows the number of jobs and core-hours contributed by each queue to each system.

In all systems, we observe that the graph is darker at the top left corner and lighter at the bottom right. Thus, jobs in the same queue with a larger degree of parallelism have longer wait times. The effect follows from the fact that the wider jobs are harder to fit during scheduling. Also, jobs with similar number of cores have shorter wait times in queues with higher priorities. While these capture the general trend, we look more closely at the anomalies in the heat map.

On Hopper, the *low* queue has the lowest priority (-3) and longer wait times than most of the other queues. The queues with priority zero, *reg_1h*, *reg_xbig*, and *reg_short*, present the expected behavior: longer wait times than the ones with priority one and shorter than the one with -3. The *bigmem*, *reg_long*, *reg_small*, and *thruput* queues, present wait times significantly higher. The *big_mem* queue is the gateway for the nodes with more memory (384 large memory nodes vs. 6,000 regular nodes). The jobs in this queue may be experiencing higher wait times because they

compete to use a smaller resource set. The *reg_long* and *thruput* queues contain longer jobs than the rest with the same priority (also much longer than the ones in *low*), and thus might not be able to take advantage of backfilling. Finally, the *reg_small* long wait times may be related to its large contribution of jobs and core-hours. The queues with higher priorities than zero show shorter wait times.

Wait time for jobs allocating different number of cores but in the same queue presented unexpected values (i.e. longer wait times for larger number of cores allocated). In some cases it could be related to the jobs' wall clock time, as is the case for the *big_mem* queue. Its wait time for the 64 to 511 cores range is two days, while between 512 to 1023 cores is seven hours. We analyzed the median of wall clock times in those ranges, obtaining one day and three hours respectively. The jobs allocating 64 to 511 cores were probably harder to backfill due to their longer wall clock times, increasing the wait time.

Edison exhibits a similar behavior to Hopper. The queues *killable* and *ccm_queue* have longer wait times, because their job's run-time is longer than the jobs in other queues with similar or lower priority. The *reg_small* queue has the maximum jobs and core-hours used on Edison, resulting in possibly longer wait times of its jobs. The jobs with higher priorities behave as expected, showing shorter wait times.

Carver displays different trends compared to Hopper and Edison. The *serial* queue has exclusive resources and a median wait time of five minutes. The *matgen* queues has a pool of resources, but those might be used by other queues. The queue also has a high job count, and thus higher wait times than queues with lower priority. The *xlmem* queues are similar to Hopper's *bigmem*, and meant to serve jobs with large memory requirements. However, their resource assignment is different: On Hopper, *bigmem* jobs can only be executed on nodes with large memory capacity, but these nodes can also execute jobs from other queues. In the case of Carver, only the jobs from the *xlmem* queues can be run in the special nodes. This exclusive access, combined with *xlmem*'s low job count and core-hour contribution, explains why these queues' median job wait time is under four minutes.

Three queues (*reg_big*, *reg_long,* reg_xlong) have priority zero and longer wait times than the other queues. The *reg_small* queue jobs consume more core-hours than any other queue (apart from *serial*), which may be the reason for the long wait times in this queue. Finally, the queues with higher priorities behave as expected.

In general we observed that queues that did not display expected queue wait time patterns had low homogeneity indices (under 60%). In particular, the queues on Hopper with such indexes should be further studied. More predictable wait times could be possible by dividing these queues according to the observed job clusters.

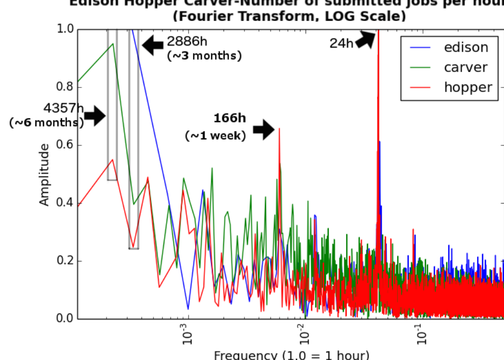


Figure 7: Fourier decomposition of the series of the jobs submitted per hour for Edison, Hopper, and Carver to detect dominant submission cycle. Note the logarithmic scale for the frequencies. Most powerful frequencies highlighted with a black arrow and its corresponding period.

7. Trend Analysis

After analyzing the system behavior during a particular period of time, this section presents a similar analysis but performed over the lifetime (Jan 2010 to June 2014) for Hopper and Carver. The purpose is to observe if there is any clear evolution pattern.

7.1. Time Patterns and analysis granularity

Choosing a time period to slice the data was the first step of this analysis. A Fourier transform analysis was performed on the number of tasks submitted per hour, since this analysis allows us to detect cycles in the user behavior [22]. The result can be observed in Figure 7: Black arrows point to the most powerful frequencies correspond to the periods of 1 day, 1 week, 3 months, 6 months. This data matches human period times (days, working weeks). Each project has a number of core-hours to be used in a year, divided in 4 allocation quarters in which the project has to consume (or forfeit) the corresponding allocated time. The strong pattern around the allocation year led us to choose one year as the trend analysis period time.

7.2. Job geometry

The evolution of the first job geometry variable is presented in Figure 8 as a box plot of job wall clock time for each system in each year. Hopper shows a significantly low wall clock time median in 2010 (< 1 minute), which might be related to the fact that it was a smaller testbed system that year. In 2011, the median increased to $\sim$ 5 minutes and subsequently increased to $\sim$ 12 minutes by 2014. Carver shows a different trend: the median, upper and lower quartile decrease effectively over the period studied. The median decreased from $\sim$ 20 minutes (2010) to $\sim$ 6 minutes (2014). However, there is some variation

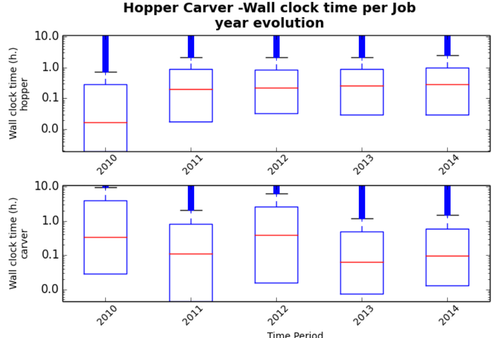


Figure 8: Job wall clock time for each each workload year. Trend: Hopper jobs become longer, Carver jobs shorter. Majority of jobs under one hour.

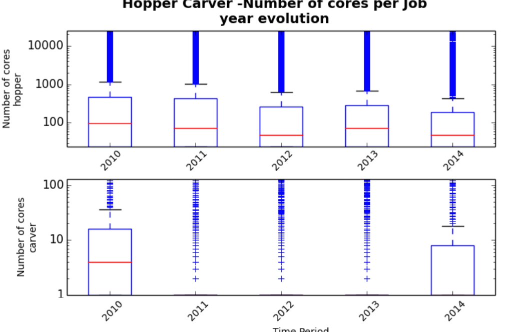


Figure 9: Allocated number of cores for each workload year. Trend: Hopper jobs allocate less cores. In 2011-2013, most Carver jobs used one core.

from year to year: It is observed that in the first year in production, Carver ran longer jobs than Hopper, a fact that slowly changed in 2014 when Hopper ran longer jobs that Carver. More generally, Hopper presents fairly short jobs, as the highest upper quartile is around the one hour value. Carver presents a similar behavior as the upper quartiles of the last years are under one hour.

The evolution of the width of jobs (number of allocated cores per job) is shown in Figure 10. For Hopper, the median decreases from 100 cores (2010) to under 30 cores (2014). Carver presents an opposite pattern. Except for 2010, the median of the rest of the years is one core, showing the predominance of single core serial jobs. In 2014, the upper quartile increased to 8 cores.

The core-time i.e., total clock time across all cores was also studied. In the case of Hopper, it remains nearly the same through time with a median of $\sim$ 20 core-hours and the upper quartile slightly under 200 core-hours in most years. In the case of Carver, it slowly decreases from a median of almost 1 core-hour to $\sim$ 6 core minutes (and a

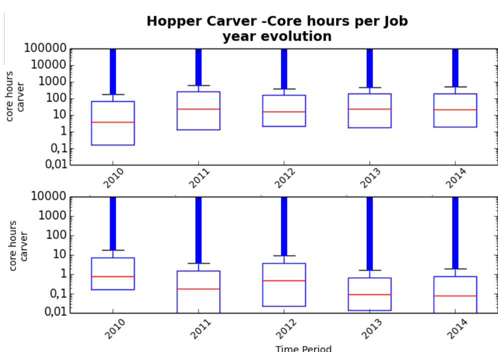


Figure 10: Allocated core-hours for each workload year. Trend: No changes on Hopper. Carver jobs become smaller.

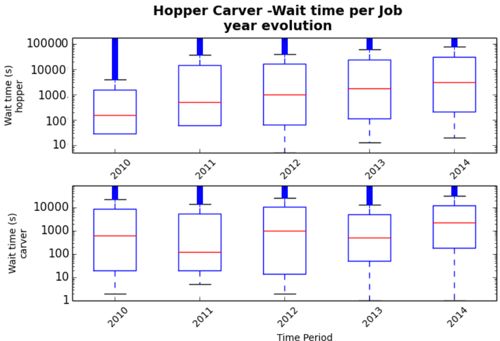


Figure 11: Jobs' wait time evolution for each workload year. Trend: All systems increase wait time. Carver lower wait time in 2011.

last upper quartile of 1 core-hour).

In summary, Hopper jobs (shorter jobs, with a higher degree of parallelism, bigger than Carver's) seem to be showing an increase in their wall clock time. As the effective job's core-hours remain the same, they must be using fewer cores. Carver jobs (longer jobs, lower degree of parallelism, fewer core-hours than Hopper's) have decreasing wall clock time and use more cores, but the increase is not sufficient to keep the job's core hours steady over the years.

7.3. Job wait time

According to Figure 11, for Hopper, the median of the wait time is steadily increasing from under 100 seconds to over 20 minutes (a pattern also present in the upper and lower quartiles). On Carver, the effective wait time increases over the four years from $\sim$ 10 minutes in 2010 to $\sim$ 20 minutes in 2014. However we notice a zigzag pattern trend in between. In 2011, Carver presented significantly shorter wait times, which could be attributed to a known increase of resources in the system. The steady increase of

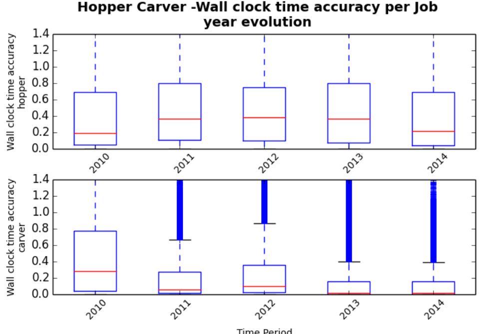
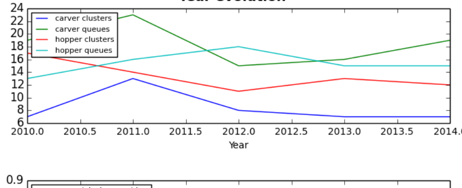


Figure 12: Jobs' wall clock time accuracy evolution for each workload year. In all systems wall clock time remains low.



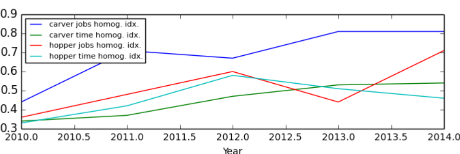


Figure 13: Workload diversity and in queue homogeneity index: Overall workload becoming less diverse. Job mix in queues becoming more uniform.

wait time over the lifetime fits with the growth of the user community of the systems.

7.4. Wall clock time accuracy

The wall clock time accuracy is calculated as *real/estimated* wall clock time. The results are shown in Figure 12. Hopper does not show a clear trend: 2011 to 2013 presents a higher accuracy than 2010 and 2014, with a median variation between 0.2 and 0.4. For Carver, the median decreases over time, with significant changes between 2010 (~ 0.25) and 2011 (<0.1). In 2014, the median is under 0.1 and the last quartile it is under 0.2. For Carver, there is a clear pattern of worse estimations as the time proceeds. In general, both systems present very low values with medians under 0.4.

Wall clock accuracy time does not show a noticeable pattern beyond the fact that accuracy is low. On Hopper, 50% of all jobs run less than 40% of the estimated time. Similarly on Carver, 50% of all jobs less than 20% of the estimated time. These values indicate that the decisions made by the backfilling algorithms are based on inaccurate user estimations.

7.5. Job and queue diversity

Following the methodology presented in Section 3.3, the diversity analysis was performed for each year and system under two different perspectives: how different are the jobs overall and how different are the jobs inside of each queue. Results can be observed in Figure 13.

Hopper shows lower numbers of clusters over time, decreasing from 17 to 12 clusters, implying a decreasing general diversity. In the case of Carver, it started with a fairly simple job mix (7 clusters), had an increase of complexity in the second year (13 clusters), to go down to the same number of clusters (7) in the last two years. In all years Carver presents a more homogeneous job mix in comparison to Hopper.

According to Figure 13, Hopper's homogeneity *job homogeneity index* increased 0.36 to 0.71, while its *time homogeneity index* increased from 0.33 to 0.46. Queues with more jobs contain more self similar jobs in 2014 than in 2010, It is interesting to note that 2012 was higher than in other years, implying that queues contributing more core-hours had more self similar jobs. In the case of carver the trend is similar, *job homogeneity index* increases from 0.44 (2010) to 0.81 (2014) and *time homogeneity index* increases from 0.34 (2010) to 0.54 (2011). As we observe all years, it can be stated that under both criteria Carver queues job diversity was lower than in Hopper.

Overall, it can be concluded that, as both systems aged, the jobs submitted by the users evolved to become more uniform. Additionally, the configured policies have evolved to build queues that classify better the jobs to contain more self similar jobs.

8. Results summary and conclusions

In this section, we summarize our results and present our conclusions on the workload analysis method and results characterization.

8.1. Summary of results, a year of workload

We present a reference of the current state of the workloads of two large scale and one mid scale high performance systems: We summarize the key results from our detailed analysis performed on the 2014's workload from Edison, Hopper, and Carver. We also compare them with preexisting analysis of similar, still older.

- The job wall clock times are short on all three systems. From 86% to 88% of the jobs run less than 2 hours.

- On Edison and Hopper, 37%, 39% of the number of jobs run on one node and 69%, 75% run on 10 nodes or less. On Carver, 92% of the jobs run on one node.
- On Carver, 77% of its jobs require one or less core-hours. Carver jobs use far fewer hours than jobs on Hopper and Edison.
- Jobs run less than 50% of their requested time: 60% of Edison jobs, 66% of Hopper's jobs and 95% of Carver jobs. Jobs run over their underestimated wall clock time: 10% Hopper's jobs, 11% of Edison's and 8% of Carver's.
- Carver has the most homogeneous workload (more similar jobs, similarity among jobs in the same queue). Hopper has a diverse workload with a complex job mix in its queues.
- Carver has the longest wait times, although its jobs allocate significantly fewer core-hours than jobs on Hopper and Edison.
- On all systems, the wait time increases as jobs allocate more resources. The wait time decreases with higher priority in most cases. In some cases anomalies are observed; e.g. larger jobs with lower priority experience shorter wait time.

Although these results represent the state of current systems, it is relevant to understand their difference to exiting work on workloads analysis of similar systems. In particular, our results were compared to analyses on Intrepid an Stampede, one current and one past HPC systems.

Intrepid was a Blue Gene/P supercomputer, with 163,840 cores, 80 TB of memory (512 MB per core), custom interconnect, peak Linpack performance of 458.6 TFLOPS, and was deployed in 2008 at the Argonne National Laboratory. From the point of view of configuration, Intrepid is more similar to Carver, as both are Teraflop systems and closer in deployment time. However, Intrepid is a Blue Gene/P system, characterized by providing compute power through smaller but more numerous CPU cores, an opposing approach to the analyzed NERSC systems' architecture, based on more powerful cores. We compare with a trace from nine months of Intrepid in 2009 [1].

Stampede is a POWEREDGE C8220 high performance cluster with 462,462 cores, an Infiniband interconnect, that can deliver up to 8 PFLOPS. It was deployed at the Texas Advanced Computing Center (Univ. of Texas) in 2012. Stampede could be compared to Edison or Hopper in terms of capacity, but its architecture differ from them as its processing units are hybrid. Stampede includes both Xeon and Phi processors in its compute nodes. For applications using its Xeon processors, Stampede performs like Edison or Hopper, in fact, Edison's processor are the next generation (Ivy Bridge) to Stampede's (Sandy Bridge). However, the Phi processors are different. They are manycore processors that include 61 light but power efficient CPU cores, from a generation before current Knightslanding (KNL) manycore Intel chips [25]. Phi processors require a regular processor to load work on them and manage their operations. We compare with an analysis over a trace of three months of Stampede in 2013 [26].

Looking into the workloads, Edison and Hopper's wall clock time distribution matches the patterns observed on Intrepid [1]. Carver's run time CDF is steeper and similar to Stampede [26]. Jobs in all three systems use fewer cores than Intrepid. Edison and Hopper jobs are similar to the jobs on Stampede in terms of cores.

The *serial* queue jobs on Carver dominate the distribution. Thus, Carver jobs are very different from Edison and Hopper and other HPC systems. Edison and Hopper share characteristics with reference systems like Intrepid or Stampede. It is possible that current DOE Leadership Computing Facilities exhibit slightly different workload characteristics [27]. This work is a first step to understand if results from NERSC may translate to such facilities.

8.2. Summary of results, systems lifetime evolution

Observing the evolution of large scientific infrastructures through their life time allows to understand systems evolution as their use mature. It also provides data to support the characteristics of future workloads.

Figure 13 presents how Hopper's workload evolved to a more uniform job mix. However, Carver presents a spike in the second year of its lifetime, to return to values similar to the beginning. Hopper results capture the nature evolution of systems, where the scheduler and other machine characteristics are refined based on the workload characteristics. Carver suffered two significant changes in 2011 that might have affected its diversity. First, Carver was expanded in 2011 [28]. Second, the serial batch queue (long running, low degree of parallelization) was added [29].

Figure 11 presents how the wait time steadily increases as the systems age, this fits with a growing scientific community using the same system and more advanced applications that require faster infrastructure. Still, there is one exception, in 2011 Carver presented significantly smaller wait times. This could be attributed to its expansion in 2011.

As Hopper and Carver are compared in Figure 8 and Figure 10, the analysis on the evolution of the job geometry reveal that Hopper jobs (which had shorter jobs but with a higher degree of parallelism than Carver) seems be increasing their wall clock time but using fewer CPU cores, while Carver jobs (which had longer jobs but with a lower degree of parallelism than Hopper) are decreasing their wall clock time and using more CPU cores.

In general, it is interesting to observe signs that would support the idea that as a systems ages, its workload variables evolve with a distinct trend, becoming more homogeneous but putting an increasing pressure on the available resources.

System	Vendor	Model	Built	Nodes	Cores/N	Cores	Memory	Network	TFlops/s	Processor
Intrepid	IBM	Blue Gene/P	2008	40,960	4	163,840	80 TB	Torus	557.1	Blue Gene
Stampede	Dell	PowerEdge	2012	6,400	16+61	462,462	192 TB	Inf. FDR	8,520	Xeon, Phi

Table 7: Intrepid and Stampede characteristics

8.3. Conclusions

Analyzing NERSC's workload offered a challenge that required new analysis tools. For this, we establish a methodology that include traditional workload analysis techniques (e.g., CDF analysis of job variables) but incorporates new methods to asses job heterogeneity. The job heterogeneity analysis includes a novel algorithm that employ's k-means clustering to detect the minimum number of dominant job geometries in an HPC workload. The method also analyzes the mapping of dominant job groups on the system prioritization schema and the resulting job wait times. This enables to asses the effect of job heterogeneity on the the scheduling performance in terms of wait time.

The results of the first application of this methodology establish a reference of the state of the workload in 2014 of three high performance systems (Edison, Hopper, and Carver). Such systems are similar size, architecture, and workload to many other current HPC systems. These results can be of use to understand the behavior in other systems, and among them we highlight: (1) The job geometries were fairly diverse including significant number of smaller jobs compared to older systems. The low per queue homogeneity indexes, show that (2) single priority policies are affecting jobs with a fairly diverse geometry. The wait time analysis shows that (3) studied queues with low homogeneity indexes present poor correlation between job's wait time and geometry. Job's submission patterns show that (4) the accuracy of users' predictions of their job's wall clock time (fundamental for the performance of backfilling functions) is very low, and does not improve over time. Finally, (5) Hopper and Carver workloads presented a clear trend in their four year lifetime: they become less diverse, their queues classify better their jobs, and they become more similar. (6) Also, they experience a heavy load that increases the overall wait times.

Our results and methodology are of use for future scheduling research and systems operations management. Scheduling research needs to address present and future workloads and our results set a first step to understand characteristics of future systems (e.g., diverse jobs, smaller jobs, or low accuracy in runtime estimations).

For system management, we highlight a result and an alternative application of our methodology. First, low values on wall clock time accuracy points to further research on how to encourage users to encourage users to provide better predictions. Better runtime accuracy will increase the quality of the backfilling in schedulers. Finally, the dominant job groups produced by the job heterogeneity analysis could be a template to define priority groups and queues. Our results show that diverse queues offered hard to predict wait times. Queues obtained by subdividing dominant job groups could show predictable wait times.

9. Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) and the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE, by the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS), the European Unions Framework Programme Horizon 2020 under grant agreement 732667 (RECAP), and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control. We would like to thank Sophia Pasadis for editing help with the paper.

References

[1] D. Feitelson, Parallel workloads archive 71 (86) (2007) 337–360, `http://www.cs.huji.ac.il/labs/parallel/workload`.

[2] NERSC, `http://www.nersc.gov`, 2015-01-18.

[3] G. P. R. Alvarez, P.-O. Östberg, E. Elmroth, K. Antypas, R. Gerber, L. Ramakrishnan, Towards understanding job heterogeneity in hpc: A nersc case study, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2016, pp. 521–526.

[4] G. Rodrigo, P.-O. Östberg, E. Elmroth, K. Antypass, R. Gerber, L. Ramakrishnan, HPC system lifetime story: Workload characterization and evolutionary analyses on NERSC systems, in: The 24th International ACM Symposium on High-Performance Distributed Computing (HPDC), 2015.

[5] M. A. Bauer, A. Biem, S. McIntyre, N. Tamura, Y. Xie, High-performance parallel and stream processing of x-ray microdiffraction data on multicores, in: Journal of Physics: Conference Series, Vol. 341, IOP Publishing, 2012, p. 012025.

[6] S. N. Srirama, P. Jakovits, E. Vainikko, Adapting scientific computing problems to clouds using mapreduce, Future Generation Computer Systems 28 (1) (2012) 184–192.

[7] T. Hey, S. Tansley, K. M. Tolle, et al., The fourth paradigm: data-intensive scientific discovery, Vol. 1, Microsoft research Redmond, WA, 2009.

[8] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, Parallel job scheduling, a status report, in: Job Scheduling Strategies for Parallel Processing, Springer, 2005, pp. 1–16.

[9] D. A. Lifka, The ANL/IBM SP scheduling system, in: Job Scheduling Strategies for Parallel Processing, Springer, 1995, pp. 295–303.

[10] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, D. Epema, The grid workloads archive, Future Generation Computer Systems 24 (7) (2008) 672–686.

[11] A. K. Mishra, J. L. Hellerstein, W. Cirne, C. R. Das, Towards characterizing cloud backend workloads: insights from google

compute clusters, ACM SIGMETRICS Performance Evaluation Review 37 (4) (2010) 34–41.
[12] K. Antypas, B. A. Austin, T. L. Butler, R. A. Gerber, NERSC workload analysis on Hopper, Tech. rep., LBNL Report: 6804E (October 2014).
[13] NERSC, Submitting batch jobs (carver), https://www.nersc.gov/users/computational-systems/carver/running-jobs/batch-jobs/, 2015.1.15.
[14] C. Vaughan, M. Rajan, R. Barrett, D. Doerfler, K. Pedretti, Investigating the impact of the Cielo Cray XE6 architecture on scientific application codes, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), IEEE, 2011, pp. 1831–1837.
[15] T. M. Declerck, I. Sakrejda, External Torque/Moab on an XC30 and Fairshare, Tech. rep., NERSC, Lawrence Berkeley National Lab (2013).
[16] Y. Etsion, D. Tsafrir, A short survey of commercial cluster batch schedulers, School of Computer Science and Engineering, The Hebrew University of Jerusalem 44221 (2005) 2005–13.
[17] G. Staples, Torque resource manager, in: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, 2006, p. 8.
[18] K. Antypas, NERSC-6 workload analysis and benchmark selection process, Lawrence Berkeley National Laboratory.
[19] NERSC, Queues and polices (carver), https://www.nersc.gov/users/computational-systems/carver/running-jobs/queues-and-policies/, 2014.1.15.
URL https://www.nersc.gov/users/computational-systems/carver/running-jobs/queues-and-policies/
[20] J. Weinberg, A. Snavely, Symbiotic space-sharing on sdsc's datastar system, in: Job Scheduling Strategies for Parallel Processing, Springer, 2007, pp. 192–209.
[21] J. D. Hunter, Matplotlib: A 2D graphics environment, Computing In Science & Engineering 9 (3) (2007) 90–95.
[22] W. W.-S. Wei, Time series analysis, Addison-Wesley publ, 1994.
[23] A. Coates, A. Y. Ng, Learning feature representations with k-means, in: Neural networks: Tricks of the trade, Springer, 2012, pp. 561–580.
[24] J. A. Hartigan, M. A. Wong, Algorithm as 136: A k-means clustering algorithm, Applied statistics (1979) 100–108.
[25] A. Sodani, Knights landing (knl): 2nd generation intel® xeon phi processor, in: Hot Chips 27 Symposium (HCS), 2015 IEEE, IEEE, 2015, pp. 1–24.
[26] J. Emeras, Workload traces analysis and replay in large scale distributed systems, Ph.D. thesis, Grenoble INP (2014).
[27] S. Ahern, S. R. Alam, M. R. Fahey, R. J. Hartman-Baker, R. F. Barrett, R. A. Kendall, D. B. Kothe, R. T. Mills, R. Sankaran, A. N. Tharrington, et al., Scientific application requirements for leadership computing at the exascale, Tech. rep., Oak Ridge National Laboratory (ORNL); Center for Computational Sciences (2007).
[28] NERSC, Magellan batch queues on carver, http://www.nersc.gov/REST/announcements/message_text.php?id=1991, 2015.01.15.
URL http://www.nersc.gov/REST/announcements/message_text.php?id=1991
[29] NERSC, Serial queue on carver/magellan, http://www.nersc.gov/REST/announcements/message_text.php?id=2007, 2015.01.15.
URL http://www.nersc.gov/REST/announcements/message_text.php?id=2007

Paper II

Priority Operators for Fairshare Scheduling

Gonzalo P. Rodrigo, Per-Olov Östberg, and Erik Elmroth

In *18th Workshop on Job Scheduling Strategies for Parallel Processing,* pp .70-89, Springer International Publishing, 2014.

Priority Operators for Fairshare Scheduling

Gonzalo P. Rodrigo, Per-Olov Östberg, and Erik Elmroth

Distributed systems group,
Department of Computing Science, Umeå Univeristy,
SE-901 87, Umeå Sweden
{gonzalo,p-o,elmroth}@cs.umu.se
www.cloudresearch.se

Abstract. Decentralized prioritization is a technique to influence job scheduling order in grid fairshare scheduling without centralized control. The technique uses an algorithm that measures individual user distances between quota allocations and historical resource usage (intended and current system state) to establish a semantic for prioritization. In this work we address design and evaluation of priority operators, mathematical functions to determine such distances. We identify desirable operator characteristics, establish a methodology for operator evaluation, and evaluate a set of proposed operators for the algorithm.

1 Introduction

Fairshare scheduling is a scheduling technique derived from an operating system task scheduler algorithm [1] that prioritizes tasks based on the historical resource usage of the task owner rather than that of the task itself. This technique defines a "fair" model of resource sharing that allows users to receive system capacity proportional to quota allocations irrespective of the number of tasks they have running on the system (i.e. preventing starvation of users with fewer tasks).

In local resource management (cluster scheduler) systems such as SLURM [2] and Maui [3], this prioritization technique is extended to job level and fairshare prioritization is used to influence scheduling order for jobs based on historical consumption of resource capacity. At this level, fairshare is typically treated as one scheduling factor among many and administrators can assign weights to configure the relative importance of fairsharing in systems.

For distributed computing environment such as compute grids [4], a model for decentralized fairshare scheduling based on distribution of hierarchical allocation policies is proposed in [5], and a prototype realization and evaluation of the model is presented in [6]. Based on this work a generalized model for *decentralized prioritization* in distributed computing is discussed in [7], and we here extend on this work and address design and evaluation of *priority operators*: mathematical functions to determine the distance between individual users' quota allocations and historical resource consumption.

2 Decentralized Prioritization

2.1 Model

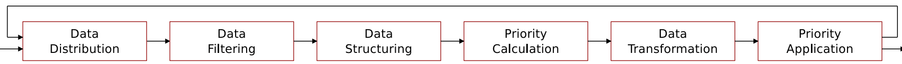


Fig. 1. A computational pipeline for decentralized prioritization. Illustration from [7].

As illustrated in Figure 1, the decentralized prioritization model defines a computational pipeline for calculation and application of prioritization. From a high level, this pipeline can be summarized in three steps: distribution of prioritization information (historical usage records and quota allocations), calculation of prioritization data, and domain-specific application of prioritization (e.g., fairshare prioritization of jobs in cluster scheduling). To model organizational hierarchies, the system expresses prioritization target functions (quota allocations or intended system behavior) in tree formats, and prioritization calculation is performed using an algorithm that uses tree structures to efficiently calculate prioritization ordering for users. The tree calculation part of the algorithm is illustrated in Figure 2, where the distance between the intended system state (target tree) and the current system state (measured tree) is calculated via node-wise application of a priority operator (in this case subtraction). The algorithm produces a priority tree - a single data structure containing all priority information needed for prioritization.

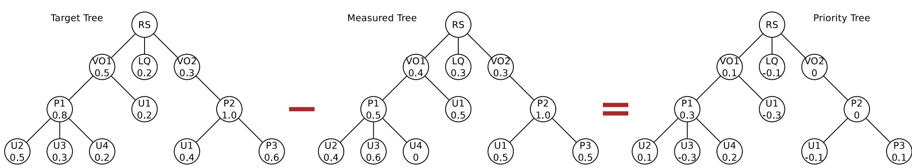


Fig. 2. A tree-based priority calculation algorithm. Tree structure models organization hierarchies (two virtual organizations and a local resource queue). Illustration from [6].

For application of prioritization (the last step in the pipeline), priority vectors are extracted from the priority tree and used to infer a prioritization order of the items to be prioritized. In fairshare prioritization of jobs in grid scheduling for example, the target tree contains quota information for users, the (measured) state tree contains a summation of historical usage information for users, and the resulting priority tree contains a measurement of how much of their respective quota each user has consumed. As each user is represented with a unique node in the trees, the values along the tree path to the node can be used to construct a priority vector for the user. Full details of the prioritization pipeline and computation algorithm are available in [6] and [7].

2.2 Challenges / Resolution Limitations

As priority operators lie at the heart of the prioritization algorithm, operator design can have great impact on the semantics of prioritization systems, and further understanding of the characteristics of the priority operator functions is needed, motivating the first part of this work.

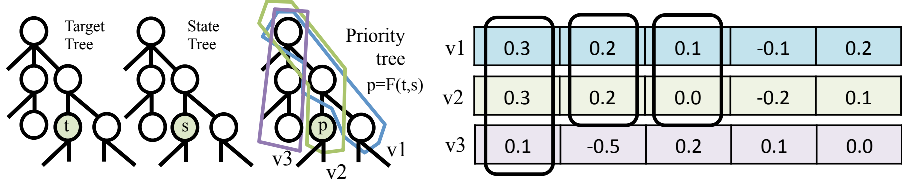
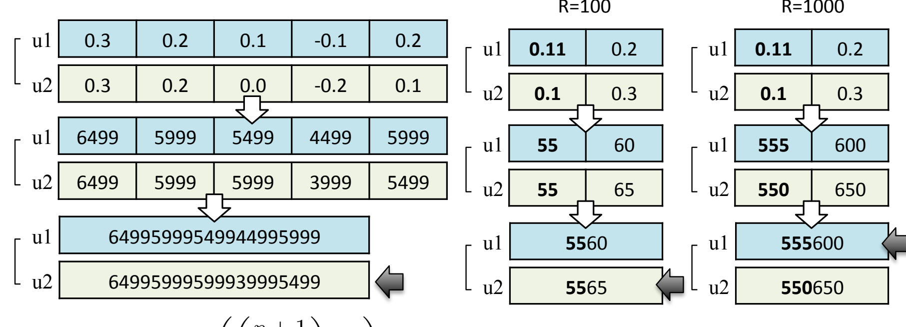


Fig. 3. Construction of priority tree and priority vectors

On the other side, the decentralized fairshare prioritization systems are meant to serve a number of resource management systems in environments created from large complex multilevel organizations. The resources access is governed by policies which structure is mapped from those organization and, as a consequence, these policies have the shape of large trees (both deep and wide). After using the priority operators to create the priority tree (as seen in Figure 2), the tree is traversed from top to bottom extracting the priority vectors (Figure 3) which have independent components representing each "level" of corresponding subgroups in the tree. The vector with the highest value at the most relevant component is the one chosen. For two or more vectors with the same component value, subsequent components are used to determine priority order.



$$scalar(p, r) = floor\left(\left(\frac{p+1}{2}\right) * r\right)$$

Fig. 4. Priority vector serialization, resolution r=10000.

Fig. 5. Resolution limitation impact on the ordering process.

However, the dimension of the overall system is translated to the size of the vector. To simplify the compare operations they are mapped on lexicographic strings. As we can see in Figure 4, for an example scalar resolution of 10000, each component of the vector is linearly mapped to a scalar in the range of 0 to 9999, where -1 is translated to 0 and 1 to 9999. Then, the values are concatenated

to construct the final string. In this example, by comparing the scalar with a simple numeric operation we can determine that u2 has a greater priority than u1 (full process described in [6]). It is important to highlight that the full final string is needed, because any transformation of the vectors that would not keep the individual element presence in it would eliminate the capacity of dividing the share in a true hierarchical way.

However, mapping the priority values to a scalar space with a limited domain has consequences: A number of priority values will map to the same scalar, which may affect the ordering process. In Figure 5 two users have two different priority vectors, when the scalar resolution is 100, the first components of both vectors map to the same scalar, although u1 has a greater priority. The result is that the u2 gets a final bigger scalar string, and thus is selected over u1. When we increase the resolution to 1000 we observe how the ordering becomes correct.

At the same time, it is also important to remember that it is desirable to use the smallest possible resolution. The overall size of the system, will bring thousands of users organized in deep trees, increasing the number of comparisons and elements in each priority vector. Any small reduction in the resolution will have a significant impact on the resources needed to compute the priority vectors.

The behavior of the operators in low resolution has to be understood and modeled, motivating the second part of this work in which we will study its impact on each operator performance and the trade-off between resolution and other characteristics of the system.

3 Operator design

3.1 Operator definition

For fairshare prioritization, we define priority operators as:

1. An operator is a function with two input variables such that:

$$t \in [0,1], s \in [0,1] \Rightarrow F(t,s) \in [-1,1] \tag{1}$$

$$\begin{aligned} F(t,s) = 0 &\iff t = s \\ F(t,s) > 0 &\iff t > s \\ F(t,s) < 0 &\iff t < s \end{aligned} \tag{2}$$

   where t represents a target value, s a (normalized) state value and $t = s$ is the ideal balance axis transecting the operator value space.

2. The function is strictly increasing on target and strictly decreasing on state:

$$\begin{aligned} \forall t_j, t_i, s_j, s_i, t, s \in &(0,1], \\ F(t_j,u) > F(t_i,u) &\iff \quad t_j > t_i \\ F(t_j,s_i) > F(t_j,s_j) &\iff \quad s_j < s_i \\ F(t_j,s) = F(t_i,s) &\iff \quad t_j = t_i \\ F(t,s_i) = F(t,s_j) &\iff \quad s_j = s_i \end{aligned} \tag{3}$$

3. Operator functions are idempotent and deterministic.

3.2 Operator characteristics

Desirable operator characteristics are dependent on application scenarios. In the context of ordering prioritization (ranking) problems it is considered desirable for operators to:

1. Have well-defined boundary behaviors:

$$\begin{aligned} \forall s \in (0,1], t = 0 &\implies \quad F(t,s) = -1 \\ \forall t \in (0,1], s = 0 &\implies \quad F(t,s) = 1 \end{aligned} \tag{4}$$

   so users with target 0 will always get the lowest possible priority (-1) and users with some target but state 0 will get the highest possible priority (1).
2. Subgroup Isolation: Depend only on target and state values of subgroup member nodes.
3. Redistribute unused resource capacity among users in the same subgroup proportionally to their allocations.
4. Be computationally efficient and have minimal memory footprint.
5. Abide by the principle of equivalence: Two operators F, F' are equivalent if they would produce the same priority ordering for a set of users knowing their target and state history. Equivalent operators will have the same characteristics for user priority ordering problems. This property is not strictly desirable for an operator but it can be used to assure that all ordering characteristics proved for an operator are automatically proved for the equivalent ones.

3.3 Operators object of study

In this section we will present the operators that are already used in the fairshare prioritization plus one new contribution (Sigmoid) and one brought from the open source cluster scheduler SLURM. The Absolute, Relative, Relative-2 and Combined operators are defined in [6]

1. Absolute: Expressed by the subtraction of the target and the state.

$$d_{Absolute} = t - s \tag{5}$$

2. Relative: Express what proportion of the user's allocation is available.

$$d_{Relative} = \begin{cases} \dfrac{t-s}{t} & s < t \\ 0 & s = t \\ -\dfrac{s-t}{s} & s > t \end{cases} \tag{6}$$

3. Relative exponential: Increases the effects of the Relative.

$$d_{Relative-n} = \begin{cases} \left(\dfrac{t-s}{t}\right)^n & s < t \\ 0 & s = t \\ -\left(\dfrac{s-t}{s}\right)^n & s > t \end{cases} \tag{7}$$

4. Sigmoid: Designed as a vertical inverse of the relative operator.

$$d_{Sigmoid} = \begin{cases} \sin\left(\frac{\pi}{2}\frac{t-s}{t}\right) & s < t \\ 0 & s = t \\ -\sin\left(\frac{\pi}{2}\frac{s-t}{s}\right) & s > t \end{cases} \tag{8}$$

5. Sigmoid Exponential: Increases the effects of the Sigmoid.

$$d_{Sigmoid-n} = \begin{cases} \sqrt[n]{\sin\left(\frac{\pi}{2}\frac{t-s}{t}\right)} & s < t \\ 0 & s = t \\ -\sqrt[n]{\sin\left(\frac{\pi}{2}\frac{s-t}{s}\right)} & s > t \end{cases} \tag{9}$$

6. Combined: Controlled aggregation of the Absolute and Relative operators.

$$d_{Combined} = k \cdot d_{Absolute} + (1-k)d_{Relative-2} \ \forall k \in [0,1] \tag{10}$$

7. SLURM: The operator used by the SLURM scheduling system [8].

$$d_{SLURMOriginal} = 2^{\left(\frac{-s}{t}\right)} \tag{11}$$

Modified to the operator output value range $[-1, 1]$

$$d_{SLURM} = 2^{\left(1+\frac{-s}{t}\right)} - 1 \tag{12}$$

4 Operator evaluation

We investigate each operator for each desirable operator characteristic.

4.1 Operator Definition

First we start with the second point of the definition. For each operator we study the sign of the first derivative when $t \neq s$. For all operators F:

$$\begin{gathered} \forall s,t \in [0,1] \wedge t \neq s : \\ \frac{d(F(t,s))}{d(p)} > 0, \frac{d(F(t,s))}{d(s)} < 0 \end{gathered} \tag{13}$$

assuring the compliance of this part of the definition. Then, as all $F(p,d)$ are strictly increasing on t and strictly decreasing on s, by studying the upper and lower bounds of the input space we can assure the compliance of the output value space:

$$\begin{gathered} F(1,0) \leq 1, F(0,0) = 0, F(0,1) \geq -1 \\ s,t \in [0,1] \wedge t = s \Leftrightarrow F(t,s) = 0 \end{gathered} \tag{14}$$

4.2 Boundary behavior

In Table 1 we observe the priority values in the boundary cases for each operator:

Operator	$t = 0$	$s = 0$
Absolute	$-s$	t
Relative	-1	1
Relative-2.	-1	1
Combined	$-0.5 - \frac{s}{2}$	$0.5 + \frac{t}{2}$
Sigmoid	-1	1
Sigmoid-2	-1	1
SLURM	-1	1

Table 1. Boundary behavior for each operator

The Absolute and Combined operator fail to comply with this property. For the Absolute, the maximum and minimum possible priority are limited in each case by the target value. The Combined operator inherits this behavior from the Absolute component in the operator.

4.3 Subgroup isolation

This property is assured by the definition of the operators: They take into account only the state and target of the corresponding nodes, which are related to the values of the nodes in the same subgroup as the first represents what share of the usage of this subgroup corresponds to this node and the latter what share of the usage should correspond to it. No data out of the subgroup is used to calculate this input values.

4.4 Proportional distribution of unused share

The situation in which a subset of users in a subgroup are not submitting jobs can be understood as an scenario with a new set of target values (virtual target): Eliminating the non submitting users and recalculating the target of the submitting users dividing the non-used share among them in proportion to their original targets. If the system would only operate with the virtual target as the input for the operator, it would converge to that new target. If an operator produces the same ordering with the virtual target (all users submitting jobs) and the old target (but with some users not submitting jobs), then we can state that the operator will bring the system to the virtual target (even if the input is the old target), spreading the unused share proportionally to the user's targets. If we define $\mathbb{T} = \{\text{set of indexes of the users submitting jobs}\}$. The condition to be complied by the operator can be expressed as:

$$\begin{gathered} i, j \in \mathbb{T} : t'_i = \frac{t_i}{\sum_{i \in \mathbb{T}} t_i}, \sum_{i \in \mathbb{T}} t_i \leq 1, t'_i \geq t_i \\ \forall t'_i, t'_j, s_i, s_j \in [0,1] : \\ 1.\ F(t'_j, s_j) > F(t'_i, s_i) \Rightarrow F(t_j, s_j) > F(t_i, s_i) \\ 2.\ F(t'_j, s_j) = F(t'_i, s_i) \Rightarrow F(t_j, s_j) = F(t_i, s_i) \end{gathered} \tag{15}$$

where t_i the target of user i, s_i the normalized state of user i and t'_i the virtual target of the user i after adding the proportional part of the unused share. Following this reasoning we proved in [9] that the Relative operator complies with this property and that any operator which would produce the same ordering would also comply with this property. As we will see in the following section, the Relative-2, Sigmoid, Sigmoid-2 and SLURM operators are equivalent to the Relative,so we can conclude that they will also distribute proportionally the unused share among the active users of the same subgroup.

4.5 Computationally efficient

Looking into the formulation of the different operators it is obvious that those ones including power, root or trigonometric operations will be more complex in math related cpu cycles. On memory requirements all should perform in a similar way. Still, the real performance of this operators will largely depend on the final implementation, as a consequence we leave this matter for the evaluation of implemented systems.

4.6 Equivalence

We formulated a theorem in [10] that allows to state that two operator are equivalent under one condition: if two operators F, F' Have the same $G(F, t_j, s_j, t_i) = s_i$ so $F(t_j, s_j) = F(t_i, s_i)$, then, they are equivalent and thus, produce the same ordering. We observe that G for the Relative operator is:

$$G(F, t_j, s_j, t_i) = s_j \frac{t_i}{t_j} \tag{16}$$

As we analyze the operators we can state that the Relative exponential, Sigmoid, Sigmoid exponential and SLURM operators share the same G and thus they are equivalent, sharing the same ordering characteristics: Proportional distribution of unused share among peering users. The Absolute and Combined have a different G between them and towards the Relative.

5 Limited output resolution

For the definition of the problem, we consider the number of possible scalar values as the resolution r. Also, under a certain resolution, each priority value p will have a resulting effective priority value $S(p, r)$, understood as the minimum priority value that will have the same corresponding scalar as p. By applying the scalar formula from Figure 4 and composing it with its own inverse function, we can calculate this effective priority value as:

$$S(p, r) = \frac{scalar(p, r)}{r} * 2 - 1 \tag{17}$$

5.1 Methodology

The output resolution problem on each operator will be studied in three steps. In the first place we will present a coarse grained study of how all the possible input pairs (t, s) are divided in sets which elements map on the same priority value. We could argue that for an operator, the bigger the set corresponding to a priory value p, the smaller resolving power (capacity to distinguish between two users) around p and vice versa. We will call this the input density study.

The second approach will be a fine grained extension of the previous. We will observe that priority operators are not defined for the full output range $[-1, 1]$ on all the possible input target values. When the input density is calculated for a priority value, it aggregates all the corresponding state values for each input target value (no matter if there is corresponding output or not), averaging the data of the resulting analysis and hiding the local behavior of the operators. For this second approach we will study each target value, analyzing the sizes of the sets of state values that map on the same output priority value. For a given operator, priority p and target t, the bigger the set of state values corresponding to the that p under t,the smaller resolving power and vice versa. We will call this the input local density study.

Finally, we will bring the input local density study to a semi-real scenario. We will define a grid scenario with a time window, resource dimension and a fictitious average job size. Then, we will determine what is the minimum output resolution required for that job to be significant for a certain user to make sure that its corresponding priority value changes. This study will be based on the previous step, as the input local density of a priority, target values can be understood as the minimum amount of normalized state that has to be added to the history of a user to assure that its corresponding priority value changes. We will call this the jobs size analysis.

In all cases the study will be focused on certain output ranges that are significant to the system: Around balance, where the state value of user is close to its target; under target, when the state is far under the target; and over target, when the state is far over the target.

5.2 Input density analysis

Calculation method For this analysis we needed to calculate the relationship between the input values corresponding to an effective priority value and the complete input range. The method follows a geometry approach when possible. In Figure 6 we can observe the effect of the discretization on the output of the Absolute operator: for each priority value p there is one horizontal surface related to the set of (t, s) that produced that effective priority value. The area of each surface will represent the relative size of that set, as a consequence, what we are looking for: the input density corresponding to p. When the geometrics of the surface were not simple enough, we used sampling to study the number of input values corresponding the the same priority value.

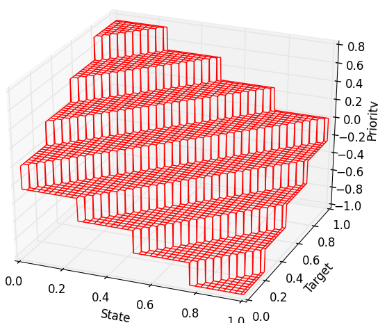


Fig. 6. 3D Representation of effective priorities for the Absolute operator, output resolution 3 bits (8 values).

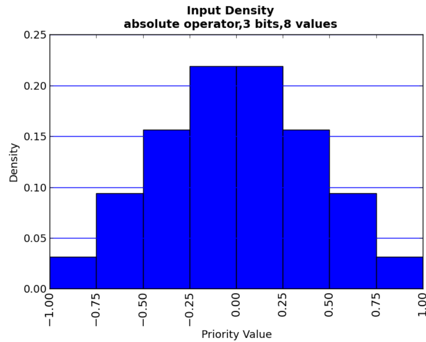


Fig. 7. Input density, Absolute operator, 3 bits, 8 values.

Input density results In order to generate all the input density maps, we run experiments for all the resolutions between 8 and 1048576, in order to ease this description we will talk from now on about the bits needed to represent a given resolution, in this case from 3 to 20 bits. Also, as we increased resolution we noticed something expected, the differences between the priority values became less and less significant. However, for the lower resolutions the operators kept a similar relationship in the same priority value areas.We have chosen 3 bits (8 values) to present the results as it is enough to show the effects of low resolution on each operator.

The results for the operators are presented in the format shown in Figure 7. The data of all operators is presented in the heat map on Figure 8. We can observe many things in this graph, in the first place there is an symmetric behavior around balance for most of the operators (except for the Combined and SLURM). Also we observe that the Relative operator presents the same input density for all its priority values. That the Sigmoid-2 presents the lowest density around balance while the highest around the over/under target cases. The Absolute operator is the one presenting the lowest density in the over/under target cases. This implies that the Sigmoid-2 operator should present the highest resolving power around balance while the lowest in the case of over usage and under usage. In the case of the Relative it presents the same resolving power along the whole output spectrum.

Considerations about this analysis The input density analysis gives a coarse grained picture of the resolution problem, it presents roughly how the whole input is mapped to the output. However, it fails to demonstrate the particularities of the operators. As we will see in the next section, the operators present different input local density distribution for different target values. As the density

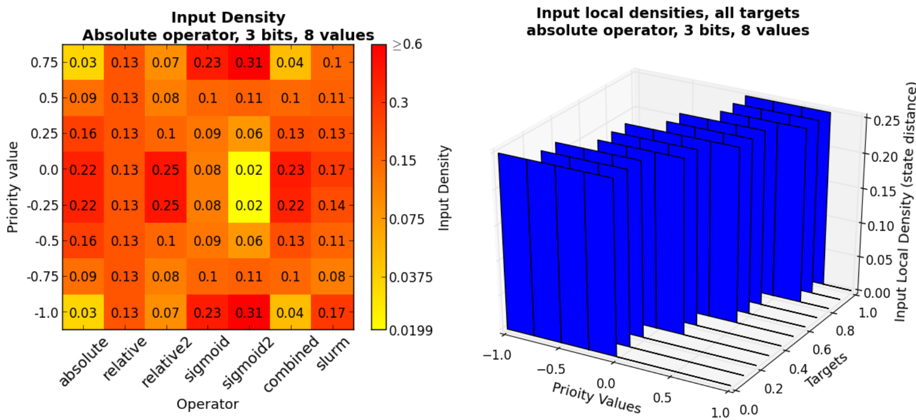


Fig. 8. Absolute Input density comparison among all operators, 3 bits, 8 values.

Fig. 9. Aggregation of the input density analysis for different target values, Absolute operator. Resolutions 3 bits.

of a priority value is the normalized aggregation of the local input densities in the whole target ranges, higher values are combined with lower ones, averaging the final result, even more significant as the operators are not covering the full output range for all the target values. Let's illustrate this with an example: the Absolute operator. According to the results in this section it presents a high resolving power around the over/under target areas and low around balance. However, lets look into what happens for each target value (results derived from next section), in Figure 9 we can see the aggregation of input local densities for 11 different targets between 0.0 and 1.0. The first observation is that, as expected, not all targets can generate the full priority range. However, what is more important is that, the resolving power is the same in all cases. This result seems contradictory to the one observed in Figure 7. This apparent divergence comes from the fact that the input density is the aggregation of the input local density along the target range, hiding the local behavior, best cases scenarios and worst case scenarios. We can conclude that the input density view gives an overall picture of the operator behavior but it is incomplete without the per-target input local density study.

5.3 Input local density analysis

Calculation methods In order to calculate the input local density for an operator, we will use an inverse method. For a operator F, a priority p, target t and resolution r, we will compute the lowest and highest state values which produce the effective priority p for target t. The local density will be the difference

between them. This can be expressed as:

$$\begin{aligned} D(F,p,t,r) = |s_j - s_i| : S(p,r) = f \quad & \wedge \\ (\forall s_i \le s \le s_j, F(t,s) = f) \quad & \wedge \\ (\forall s < s_i \wedge s_j < s, F(t,s) \neq f) & \end{aligned} \tag{18}$$

In order to calculate those s_j, s_i we will use the inverse expression of the operators on the input s and the set of possible effective priority values for resolution r, $\mathbb{P}_r$. The resulting operation is:

$$D(F,p_i,t,r) = s_{i+1} - s_i = I_s(F,p_{i+1},t) - I_s(F,p_i,t) \tag{19}$$

where:

$$\mathbb{P}_r = \{p_i : i \in \mathbb{N}_r, p_i = -1 + (i-1) \cdot \frac{2}{r}\} \tag{20}$$

(For example $\mathbb{P}_4 = \{-1, -0.5, 0, 0.5\}$) and the inverse function on s of operator F is:

$$I_s(F,p,t) = s : F(t,s) = p \tag{21}$$

Input local density results For this study we run experiments for all the resolutions between 3 and 20 bits. The target range was divided in 10 values, $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]$, as the functions are strict increasing on the target, this should be enough to appreciate the general tendency. This conforms 180 operator profiles, which had to be analyzed. One example of the aggregation of all the profiles for the Absolute operator and resolution 3 bits is the Figure 9.

As the resolution was increasing we noticed something expected, the local densities were becoming more similar for a given operator, policy and target. However, for the lower resolutions they kept a similar relationship in the same priority areas. We have chosen 3 bits for the resolution and 0.1, 0.5 as the target values (as they represent one extreme and middle point) to illustrate the behavior of the operators under low resolution.

In Figure 10 we can see one way to represent this data, the operator profile: an overlap between the input density corresponding and the operator plot for the defined target value. It will correlate graphically the priority value, input target, input state and input density, where we consider lower input local densities as something desirable since they indicate higher resolving power. This is the way to interpret them: The horizontal axis represents normalized state from 0.0 to 1.0 while the vertical axis is composed by a range of priority values from -1 to 1. On the graph we represent the overlay of different operators in different colors: The priority values and their corresponding state values in the shape of a plotted line in the corresponding color. On the vertical axis the input density on each priority value in the shape of an horizontal bar, in the corresponding color (its unit is normalized state). In this representation it is possible to observe the different operator function shapes while comparing the different density inputs. In Figure 11 we can observe the contraposition of the Sigmoid-2 and Relative-2,

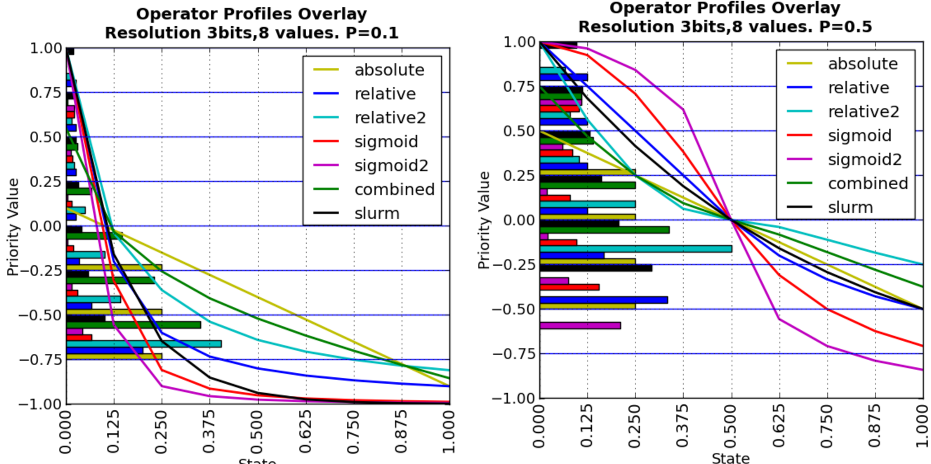


Fig. 10. Operator Profile overlay all operators. Resolution 3 bits. target value 0.1

Fig. 11. Operator Profile overlay all operators. Resolution 3 bits. target value 0.5

how the Sigmoid shape is designed to offer the bigger slope (and thus, smaller local density) around balance, and is more flat in the extremes.

As we compare the graphs in Figures 10 and 11 we observe that the range of priority values present in the graphs is smaller as the target increases. This is due to how the operator functions are built: none of them fully covering the range $[-1, 1]$. As the target value increases the most negative value possible for over-state becomes closer to 0.0 (for example, with the Absolute operator, target=0.5 and state = 1.0, the minimum possible priority value is -0.5). This has a first consequence on some of operators: as the target value increases, the same input range of $[0, 1]$ is mapped onto a smaller output range, increasing the overall input local density, increasing the average size of the bars in the graphs. One interesting point is that the Absolute operator presents a constant density in all the graphs and all the priority values, this is due to the subtraction only operation that composes it.

In order to ease the density analysis we mapped the density values on a heat map which opposes the priority values and the operators. The result are Figures 12 and 13. In which the a redder/darker color indicates a higher local density (and lower resolving power) for a pair or operator and target value while a yellower/lighter color implies a lower local density (and higher resolving power). As we observe the graphs, we confirm that, certain operators have a lower input local density behavior for different cases: In balance situations the Sigmoid and Sigmoid-2 presented smaller densities. In under target situations, the Relative-2 operators presented smaller densities. In situations of over target, again the Sigmoid and Sigmoid-2 operators presented lower densities, although not much lower. One intuitive conclusion of this section is that the Sigmoid family presents a higher resolving power for the balance cases and over usage cases, while the Relative operator presents a higher resolving power for the under usage cases.

Fig. 12. Input local density heat map. Resolution 3 bits. target value 0.1

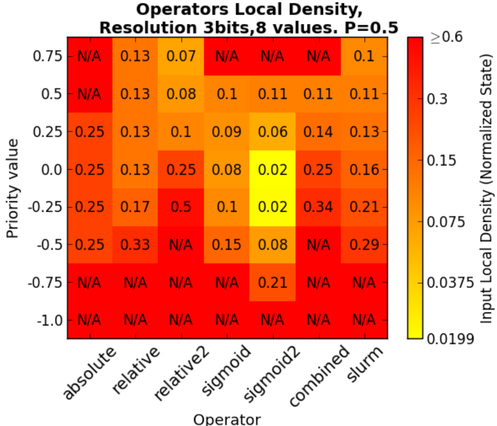


Fig. 13. Input local density heat map. Resolution 3 bits. target value 0.5

5.4 Job Size Analysis

In this study we will bring our analysis to a semi-real environment, a grid scenario with a time window (representing the total historical usage taken into account) and a set of resources. The idea is to understand how big a job has to be to be significant for a user so its effective priority value changes to the next possible one. This will show how many decisions would be made with the same priority values (although new jobs had been completed) or how many bits (minim resolution) would be required for a single job to be significant.

Calculation Method In order to do this analysis we will start from the input local density study. For a given resolution r, operator F, priority p and target t, $D(F, p, t, r)$ can be seen as the minimum amount that the state has to increase to change the output effective priority. This step can me expressed as:

$$\begin{aligned} M(F,t,s,r) = m : m \in (0,1] \quad & \wedge \\ S(F(t,s),r) \neq S(F(t,s+m),r) \quad & \wedge \\ \nexists n : 0 < n < m \wedge S(F(t,s),r) \neq S(F(t,s+n),r) & \end{aligned} \tag{22}$$

This minim step can be translated to a job size, however, it is not trivial: The impact of a job size on the normalized state will depend on the normalized state by itself, as it represents how much of the state pool corresponds to this user: Meaning that one job of a certain size will have a bigger impact for a user with less normalized state than for one with a bigger one. The current state and the state of a user after adding the length of a job can be expressed as:

$$
\begin{aligned}
s_i^d &= \frac{\sum_{0<j\leq d} J_i^j}{U^d}, J_i^{d+1} = k \cdot U^d \\
s_i^{d+1} &= \frac{\sum_{0<j\leq d} J_i^j + J_i^{d+1}}{U^d + J_i^{d+1}} = \frac{\sum_{0<j\leq d} J_i^j + k \cdot U^d}{U^d + k \cdot U^d} \\
&= \frac{(1+k)(\sum_{0<j\leq d} J_i^j + k \cdot U^d)}{U^d} = (1+k)(s_i^d + k)
\end{aligned}
\tag{23}
$$

were s_i^d is the normalized state of a user i at a time d, J_i^j is the size of a job submitted by user i in the time j, U^d is all the state recorded for all users until time d and k is the proportion between the job submitted in time d and the total state recorded until d. This equation is an expression of the new state as a function of the previous state and the proportion between the job and the time window.

Our target is to obtain a function that calculates how big that proportion has to be to jump from one two state values a user (s_i^d) to the next ($s_i^d + 1$). Taking those states as the boundaries for the local density calculation and expressing $s_i^{d+1} - s_i^d$ substituting s_i^{d+1} by the result in the Equation 23 we obtain:

$$
\begin{aligned}
D(F, p_i, t, r) = s_i^{d+1} - s_i^d = (1+k)(s_i^d + k) - s_i^d \\
k^2 + (1 + s_i^d)k - D(F, p_i, t, r) = 0
\end{aligned}
\tag{24}
$$

Which can be solved by using the quadratic equations solving formula:

$$
k = K(F, p_i, t, r, s_i^d) = \frac{-(1+s_i^d) + \sqrt{(1+s_i^d)^2 + 4 \cdot D(F, p_i, t, r)}}{2} \tag{25}
$$

This final result is an expression of k as a function of the operator, previous state, target priority value and resolution. This will be used to calculate what is the minimum job size to be submitted in order to change one user's priority value according to its current state and target and the corresponding operator and resolution. Now that we can find the job size in any case, we established a strategy for the calculations on each operator in each resolution: for each priority, among all the possible k on each target , which is the biggest one (as a bigger k means smaller job). This will allow us to calculate what would be the minimum job size that would assure a change in the effective priority value.

Job size analysis results We took the Swegrid [11] as a reference for this study to create a synthetic example in which the time window is 1 year and the resources managed 600 nodes. This implied that the total state pool of the time window was $U^d = 8760h/n \cdot 600n = 5256000h$. By using that U^d and the method to obtain k we calculated the corresponding job size. We can say that a bigger job size implies a smaller resolving power and a smaller job size implies a bigger resolving power.

For the priority value to study, we focused our calculations in 3 contexts: around balance, where the state is close to the target and the priority values are

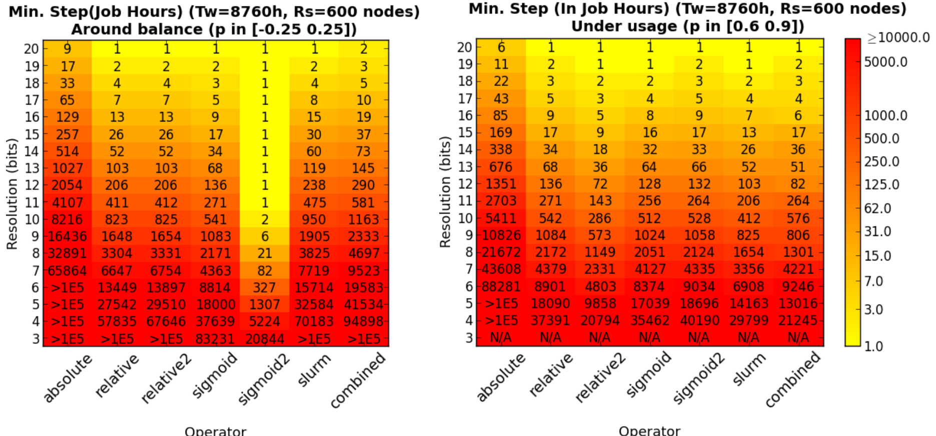


Fig. 14. Job size in hours required to alter user's priority value when state is close to target.

Fig. 15. Job size in hours required to alter user's priority value when state is under target.

in the range $[-0.25, 0.25]$; under target, where priority values are in the range $[0.6, 0.9]$; and over target, where the priorities values are in the range $[-0.9, -0.6]$.

We can observe the results of the around balance study in in Figure 14. This heat map represents the worst case (bigger) among the minimum job size required to make a difference in the priority value for each resolution with each operator. Bigger jobs (and thus smaller resolving power) are represented with darker/redder color while smaller jobs (and thus bigger resolving power) imply a lighter/yellower color. We limited the color range at jobs size 10,000h and used a logarithmic scale for the color distribution. As we can observe the results are independent of the resolution: the Absolute operator needs bigger jobs to make a difference, while the Sigmoid-2 has enough resolving power around balance to always require a smaller job than the rest operators. If we order the operators by job sizes from better to worse resolving power we obtain the following ordering: Sigmoid-2, Sigmoid, SLURM, Relative, Relative-2, Combined, Absolute. This ordering does not change as we increase resolution, although, as it was expected, the job size decreases as the resolution increases.

In the over target scenario presented in Figure 16 we observe the following ordering from better to worse resolving power in all resolutions: Sigmoid-2, Sigmoid, SLURM, Relative, Absolute, Relative-2 and Combined. In the under target scenario presented in Figure 15 we observe the following ordering in all resolutions: Relative-2, Combined, SLURM, Sigmoid, Relative, and Sigmoid-2.

At this point we had a good vision of what was the impact of a certain job for each operator in the studied cases. However we wanted to understand it from a different point of view: For a given job size, what was the minimum resolution in bits needed for an operator to make a difference? We chose an average job size

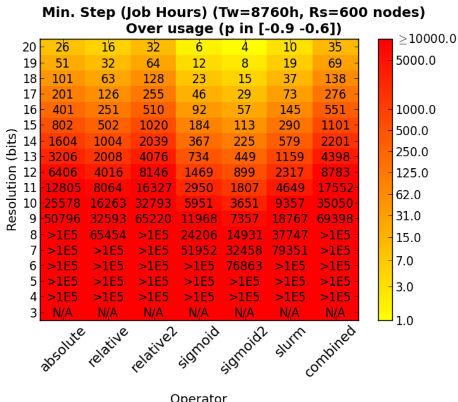


Fig. 16. Job size in hours required to alter user's priority value when state is over target.

Operator	Balance	Over t.	Under t.	Overall
Absolute	12	14	12	14
Relative	9	14	9	14
Relative-2	9	15	8	15
Sigmoid	9	12	9	12
Sigmoid-2	6	11	9	11
SLURM	9	13	8	13
Combined	10	15	8	15

Fig. 17. Output resolution bits required for a job of 2,000h to be significant for a Tw=1 year and Rs=600 nodes. Less is better.

of 2000h (200h,10 nodes). This parameter only affects to the gross value of bits obtained for each operator however, it won't change the relative relationship of the operators.

By parsing the Figures 14, 15 and 16 we produced the results for Figure 17. We can see how the Sigmoid-2 presents a clear advantage around situations of balance. The Sigmoid-2 is the one requiring less bits in the case of balance and over target. For under target, the Relative-2, SLURM and Combined perform better but only with small difference. If we look at the overall picture, the Sigmoid-2 operator is the one which requires less bits in balance and over target while for under target it is just one bit away from the best.

6 Prior and related work

The need of this work was detected in earlier efforts. In [6] and [12], the family of Relative and Combined operator were added as pointed in Section 3. The system's robustness and capabilities were tested and, as a side product, it was discovered that the absolute operator and relative operator generated different ordering when some users did not submit jobs. Also, convergence delays appeared with low resolutions (required for large scale experiments). Those findings are the motivation of this paper.

Fairshare priority is present as a decision factor in well known schedulers. SLURM ([12] shows how our system can substitute SLURM's fairshare engine), as the rest of schedulers, is not meant to deal with as deep hierarchies as Karma ([8], [7]) so the output resolution of its operator is not constrained. SLURM operator, as studied in this paper, complies with all our desired characteristics. Maui [13] presents similar characteristics but uses a version of the Relative as its operator. Other well known example is LSF used by the CERN [14], which choses the absolute operator [15].

The Karma hierarchical system is prepared for much deeper tree schemas than current schedulers and to find work related to the priority resolution we had to look into another scheduling field in which memory is limited: real time schedulers for communication and embedded systems. During the study of the monotonic scheduling algorithm in [16], this problem is named *priority granularity*: "fewer priority levels available than there are task periods", similar to the idea of a number of users with very similar priority and limited resolution around it. In [17], a solution was pointed by creating an exponential grid to distribute the priorities increasing the resolving power around a certain desired area, very similar to what we intended by designing the Sigmoid operator. Finally, [18] presents how the low output resolution can affect negatively to the utilization of resources, as it would alter the best possible decisions, for this case a logarithmic grid is presented. This studies are focused on the final priority value without taking into account the relationship of the input of the prioritization system and its output. In our work we understand how are the system states (input to the priority operator) affected by the limitation on resolution, this allows to modify the system accordingly to the desired resolving power behavior.

7 Future work

This study is a mathematical understanding of the operators in the fairshare prioritization scheduling. it allows to establish boundaries on the mathematical behavior of the functions, however, it would be desired to confront the results with the Karma [7], system: To establish a simulation environment in which all the operators ares tested in a close to real situation, with different output resolutions, tree models (different target values, target tree shapes and depths) and sources of system noise (as presented in [6]). In this context it would interesting to study a possible adaptive operator: a dynamic recommendation of the best suited operator and minimum output resolution depending on the overall state of the system.

We also understand that it would be interesting to test the Relative and Sigmoid operator family on the SLURM scheduling system. Our studies reflect that these operators may have a better resolving power than the SLURM operator, while similar general characteristics. Bringing those conclusion to SLURM, which has a different data pipeline than Karma, would bring light on the compatibility of our operators with other scheduling strategies.

In a different line of thinking, as it was presented in [7], the Karma prioritization engine can be used to deal with scenarios that are not strictly concerned about the ordering of elements, but also about the magnitude of the priority values produced by the system. It would be desirable to understand what is the impact of the different operator output value distribution on this magnitude aware systems.

8 Conclusions

We achieved a deeper understanding on the role of the priority operators in the fairshare prioritization scheduling. We established their core definition and the desired characteristics that emanate from the studied problem. This was followed by the review of the existing operators and the contribution of a new one: The Sigmoid operator.

To evaluate the desired characteristics we established a set methods and mathematical proofs that allow to test the compliance of the existing and future operators. Following this methods, all the presented operators were tested determining which ones complied with the desired characteristics. The results: While the Relative, Relative-n, Sigmoid, Sigmoid-n and SLURM operators comply with all of them, the Absolute and Combined does not on some of them.

At this point our study moved into another dimension, understanding the impact of the limitation in output resolution on the whole prioritization. The first contribution was a three step methodology to evaluate its impact on the resolving power of each operator: the study of the input density, input local density and significant job size. We reviewed the potential of the different possibilities and understood the need to go through the three steps in order to have first, an overall view; second, a local detailed understanding of the operator behavior in all its input domain; and finally, a real world quantification of the relationship between output resolution and resolving power of each operator.

By using this methodology, we were able to compare all the listed operators. We established that the Sigmoid-2 is the one which, in overall, has a better resolving power with less output resolution, being the best in situations of balance and over target. In the case of under target the Relative-2, SLURM and Combined operators are the ones with a better resolving power with low output resolution (although very close to the Sigmoid-2). We would like to highlight that the SLURM operator, taken from the SLURM scheduling system, performs average in the balance and over target situations. One interesting conclusion is that the Sigmoid family, a contribution of this paper, presents the best overall characteristics as a fairshare priority operator.

Finally, this paper establishes a set of research lines to bring these results into the Karma prioritization system.

Acknowledgments

The authors extend their gratitude to Daniel Espling for prior work and technical support, Cristian Klein for feedback and Tomas Forsman for technical assistance. Financial support for the project is provided by the Swedish Government's strategic research effort eSSENCE and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

References

1. Kay, J., Lauder, P.: A fair share scheduler. Communications of the ACM 31 (1) (1988) 44–55
2. Yoo, A., Jette, M., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In Feitelson, D., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing. Volume 2862 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2003) 44–60
3. Maui Cluster Scheduler: http://www.adaptivecomputing.com/products/open-source/maui/, January 2014.
4. Foster, I., Kesselman, C.: The grid: blueprint for a new computing infrastructure. Morgan Kaufmann (2004)
5. Elmroth, E., Gardfjäll, P.: Design and evaluation of a decentralized system for Grid-wide fairshare scheduling. In H. Stockinger et al, ed.: Proceedings of e-Science 2005, IEEE CS Press (2005) 221–229
6. Östberg, P-O. and Espling, D., Elmroth, E.: Decentralized scalable fairshare scheduling. Future Generation Computer Systems - The International Journal of Grid Computing and eScience 29 (2013) 130–143
7. Östberg, P-O. and Elmroth, E.: Decentralized prioritization-based management systems for distributed computing. In: eScience (eScience), 2013 IEEE 9th International Conference on, IEEE (2013) 228–237
8. Slurm: Multifactor priority plugin - simplified fair-share formula. `https://computing.llnl.gov/linux/slurm/priority_multifactor.html` (January 2014)
9. Rodrigo, G.P.: Proof of compliance for the relative operator on the proportional distribution of unused share in an ordering fairshare system. `http://www8.cs.umu.se/~gonzalo/ShareDemonstration.pdf` (January 2014) (To be published as a technical report).
10. Rodrigo, G.P.: Establishing the equivalence between operators. `http://www8.cs.umu.se/~gonzalo/EquivalenceDemonstration.pdf` (January 2014) (To be published as a technical report).
11. Swegrid: Swegrid organization. `http://snicdocs.nsc.liu.se/wiki/SweGrid` (January 2014)
12. Espling, D., Östberg, P-O. and Elmroth, E.: Integration and evaluation of decentralized fairshare prioritization (aequus) decentralized scalable fairshare scheduling. (2013) (Submitted for publication).
13. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Job Scheduling Strategies for Parallel Processing, Springer (2001) 87–102
14. CERN: It services - batch service. `http://information-technology.web.cern.ch/services/batch` (January 2014)
15. LSF: Fairshare scheduling. `http://www.ccs.miami.edu/hpc/lsf/7.0.6/admin/fairshare.html` (January 2014)
16. Lehoczky, J., Sha, L., Ding, Y.: The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In: Real Time Systems Symposium, 1989., Proceedings., IEEE (1989) 166–171
17. Sha, L., Lehoczky, J.P., Rajkumar, R.: Task scheduling in distributed real-time systems. In: Robotics and IECON'87 Conferences, International Society for Optics and Photonics (1987) 909–917
18. Lehoczky, J.P., Sha, L.: Performance of real-time bus scheduling algorithms. ACM SIGMETRICS Performance Evaluation Review 14 (1) (1986) 44–53

Paper III

A2L2: An Application Aware Flexible HPC Scheduling Model for Low-Latency Allocation

Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth, and Lavanya Ramakrishnan

In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing,* pp. 11-19. ACM, 2015.

A2L2: an Application Aware Flexible HPC Scheduling Model for Low-Latency Allocation

Gonzalo P. Rodrigo, Per-Olov Östberg, Erik Elmroth
Dept. Computing Science, Umeå University
SE-901 87, Umeå, Sweden
{gonzalo, p-o, elmroth}@cs.umu.se

Lavanya Ramakrishnan
Lawrence Berkeley National Lab
Berkeley, CA 94720, USA
lramakrishnan@lbl.gov

ABSTRACT

High-performance computing (HPC) is focused on providing large-scale compute capacity to scientific applications. HPC schedulers tend to be optimized for large parallel batch jobs and, as such, often overlook the requirements of other scientific applications. In this work, we propose a cloud-inspired HPC scheduling model that aims to capture application performance and requirement models (Application Aware - A2) and dynamically resize malleable application resource allocations to be able to support applications with critical performance or deadline requirements. (Low Latency allocation - L2). The proposed model incorporates measures to improve data-intensive applications performance on HPC systems and is derived from a set of cloud scheduling techniques that are identified as applicable in HPC environments. The model places special focus on dynamically malleable applications; data-intensive applications that support dynamic resource allocation without incurring severe performance penalties; which are proposed for fine-grained backfilling and dynamic resource allocation control without job preemption.

Categories and Subject Descriptors

D4.1 [**Operating Systems**]: Process Management—*Scheduling*

Keywords

Scheduling; job; HPC; malleable; applications; low-latency

1. INTRODUCTION

High-performance computing (HPC) and cloud computing are paradigms focused on large-scale resource provisioning through aggregation of resources in data centers. Although similar at high level, the paradigms have fundamental differences in system objectives and target applications that affect the design of their infrastructure and schedulers. HPC systems serve scientific applications, traditionally composed of tightly coupled parallel jobs and use batch schedulers focused on system utilization [5]. Cloud computing offers variable, on-demand compute capacity to run different types of applications that often face great variability in load and require resource and infrastructure elasticity [4]. As the data-intensive applications that are common in cloud workloads [31] are becoming increasingly common in HPC [40], a trend towards increased overlap between the paradigms can be observed.

In this work, we focus on scientific applications in HPC systems and, among these, in particular on data-intensive applications. As this application type is characterized by minimal communication between tasks and I/O centric performance models [41], application input data can often be rearranged among the processing nodes without affecting application result or performance (as in, e.g., MapReduce applications [9]). This property allows us to dynamically change application resource allocations without incurring significant performance penalties (lost work or low resource utilization). We call these applications *dynamically malleable* and observe that they provide schedulers interesting opportunities for increased scheduling performance: application malleability provides schedulers the ability to make small adjustments to the size of jobs, which allows for tighter packing of job in backfilling (improved resource utilization). In addition, the ability to downsize jobs allows schedulers to temporarily free (some of the) resources allocated to dynamically malleable applications, which can be used to make room to schedule time-critical jobs with short response times. This enables allocation of resources to run event-synchronized computations (e.g., real-time processing of data from external experiments) without advance reservations of resources (known to reduce resource utilization [1]).

However, this approach also offer challenges: in order to efficiently alter application resource assignments, schedulers require application-level control to adapt to resource changes [34]. Also, to allow schedulers to decide on resource allocations aimed at a specific performance target, users should provide performance expectations rather than resource estimations (e.g., execution deadlines instead of job runtime estimations [12]). In addition, data-intensive applications can expect high performance improvements from data locality (data on compute nodes) [34]. This is hard to realize on HPC systems that typically provide high-performance shared (distributed) file systems rather than local node storage [26]. Finally, today's batch schedulers do not capture the performance and requirement models of data-intensive

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the US Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
VTDC'15, June 15 - 16, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3573-7/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2755979.2755983 .

	HPC	Cloud
Applications	Batch, data-intensive, parallel tightly coupled jobs.	Batch jobs, data-intensive applications, services.
Platform	Bare metal.	Bare metal, virtualization, execution frameworks.
Hardware	Homogeneous (not in new systems [8]), low-latency networks, Burst Buffer or I/O nodes.	Heterogeneous, commodity, large RAM, storage on node.
Target	High Utilization, low wait times, low cost.	Availability, no wait for services, low cost.
Technique	Job ordering, FCFS, backfilling, monolithic, static partition.	Placement, feedback, elasticity, migration, monolithic, static partition, two-level, shared state.

Table 1: Applications, objectives, and scheduler techniques in HPC and cloud environments.

applications, often resulting in sub-optimal decisions in scheduling of data-intensive applications on HPC systems.

In this paper we present *A2L2, an **A**pplication **A**ware flexible HPC scheduling model for **L**ow **L**atency allocation* of resources for jobs that adapts cloud techniques to tackle HPC scheduling challenges. The model is considered *application aware* as it is built on a two-level cloud inspired model [33] in which each type of application has an individual scheduler that captures its performance and requirement models. The model is considered to be *flexible* as it uses dynamically malleable applications to backfill resource gaps (later described as *flexible backfill* technique). Finally, the model aims to support *low-latency allocations* by using resource expropriation to free resources for applications with time-sensitive or real-time needs. The main contributions of this paper are:

- A2L2, an HPC application-aware scheduling model that makes use of dynamically malleable applications to enable flexible backfilling, low-latency scheduling of jobs, and performance-oriented scheduling of data-intensive applications.
- A discussion of dynamic management and control of malleable applications in HPC environments.
- A comparative discussion of scheduling techniques applicable to both HPC and cloud computing.

The remainder of this paper is structured as follows. Section 2 presents the potential and challenges of managing dynamically malleable applications. Section 3 provides a comparative overview of cloud and HPC resource provisioning (focused on scheduling and placement mechanisms) and identifies a set of key infrastructure and application characteristics that impact design of such mechanisms. Section 4 presents A2L2, a proposed scheduling model aimed to improve the flexibility and efficiency of HPC environments by use of cloud-style placement techniques. Section 5 provides a summary of identified challenges that must be overcome to realize this model and proposes a way forward in this work. Section 6 outlines a brief summary of related work and Section 7 concludes the paper.

2. DYNAMICALLY MALLEABLE

Data-intensive applications are typically organized as workflows and characterized by significant I/O operations. They benefit from parallelism but are not necessarily tightly coupled. In this work, we focus on what we call *dynamically malleable applications*, where input data can be divided in a minimum unit (quantum) and output data only depends on a specific operation or set of operations on that input data. The processing of each quantum is independent and, as a consequence, processing n quanta will require the same amount of compute time (under the same resource circumstances), no matter whether the computations are done in parallel or serially [41]. Additionally, as each quantum is independent, runtime changes in the job geometry are possible without losing intermediate results (dynamically malleable). However, the application performance model differs from that of tightly coupled parallel batch jobs and application level knowledge is required to control application resource allocations.

2.1 Performance model

The performance of data-intensive applications is by definition I/O bound. In cloud environments, distributed file systems that divide and place data on compute nodes are commonly used to achieve high throughput for data-intensive applications [34]. As such, there are two possible performance models depending on the relationship between the size of the input data and the compute node:

- Storage-centric: Input data does not fit in node memory and is staged onto the local node storage system. Hadoop [34] is an example of an execution framework built around compute node storage based distributed file systems.
- Memory-centric: The input data of all application stages fit in node memory and no local node storage is needed. Spark [42] is an example of an execution framework specialized for this kind of applications that creates a memory-based distributed file system.

As I/O operations on memory are faster, memory-centric applications offer an overall better data processing throughput than storage-centric applications. However, the input dataset size is limited by the total available node memory, memory-centric applications may require more compute nodes to process the same input dataset. If not enough nodes are available, memory-centric applications have to load more input data on the main memory of the compute nodes, reducing system overall performance [42].

However, the key performance enhancement mechanism of data intensive clouds environments (distributed storage) cannot natively be used in HPC as compute nodes typically do not have local storage and large fast shared file systems may not offer optimal performance for an application's data access patterns [26]. In Section 4.3.2 we present how A2L2 attempts to bring cloud's performance improvement techniques to HPC.

2.2 Flexibility and management

The degree of parallelism can be changed in dynamically malleable applications, impacting the overall runtime accordingly. As we present in the next sections, this flexibility can be used to meet scheduler objectives. However, this flexibility requires applications to be aware of changes in resources state and availability, and claim and release resources as needed during runtime. In data intensive cloud frameworks (e.g. Hadoop [34]), each job has a master task controlling the rest. The frameworks communicate with the master tasks to coordinate changes in resource allocation.

The scientific community has developed similar tools for certain dynamically malleable applications in HPC, but they are not integrated with resource management system schedulers and, as a consequence, do not support graceful rescaling of applications during runtime.

3. HPC AND CLOUD SYSTEMS

This work proposes a new scheduling model that focuses on application-aware scheduling to enable low-latency (on-demand) scheduling of applications in HPC environments. The first step is to understand what cloud techniques enable these features, and compare these to the regular HPC scheduling. Table 1 summarizes the observed differences and similarities between HPC and cloud scheduling.

3.1 Batch centric HPC scheduling

HPC systems focus on computing needs of scientific applications. The traditional HPC application is often described as a large simulation that run tightly coupled parallel jobs [35], allocates large numbers of nodes for long periods of time, and requires low-latency synchronized inter-process communication across nodes. Lately however, data analysis applications have become more present on HPC systems. These applications do not have strong coupling requirements, are easier to parallelize, and may in many cases be dynamically malleable (the degree of parallelism can be adapted at runtime) [9].

HPC system architectures are deigned to support the scientific applications characteristics (e.g., provide a high enough degree of parallelism to simulate a certain model) and to offer large amounts of compute capacity to large user bases while keeping the cost per compute time-unit low. The resulting systems are typically very efficient in terms of operational, energy, and procurement costs, but the support for diverse applications models is often limited. Typically HPC infrastructures are relatively homogeneous (albeit less so in newer systems [8]), have no storage on compute nodes, and use synchronized networks and high performance parallel storage systems.

HPC schedulers use a combination of techniques aimed to satisfy the system's objectives. First Come First Serve (FCFS) algorithms execute jobs in arrival order until no more jobs are waiting, or there are not enough available resources to run the first job in the queue [15]. FCFS is often complemented with Backfilling [27], a scheduling technique aimed to increase utilization by searching in the waiting queue for jobs whose resource requirements and estimated runtimes can be met using available resources without delaying the start time of jobs found earlier in the waiting queue. Job prioritization is achieved by adding techniques such as fairshare [29] or priority queues [5]. Also, jobs can

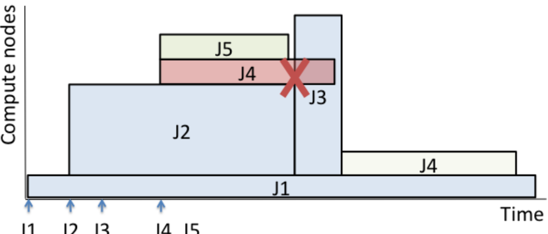


Figure 1: Scheduling of five batch jobs with FCFS and backfilling. Job J5 can be backfilled while J4 can not (as it would delay J3, which arrived before).

incorporate checkpointing techniques that can be used to restart jobs with reduced lost work after preemption.

Figure 1 illustrates scheduling of jobs using FCFS with backfilling. Each box represents a job (y-axis represents the fraction of the total nodes required by the job; x-axis represents the estimated runtime). By following FCFS, J3 starts after J2, leaving a significant amount of resources free to run J4 or J5 when they arrive. The backfilling algorithm finds that, according to its estimated runtime, J4 would end after J3's estimated start time, while J5 would end before. As a consequence J5 is backfilled. Classic batch schedulers take into account the requested resources (width) and estimated runtime (length) of jobs in a static way. This limits backfilling as it can only be performed if a match with a corresponding geometry can be fit with a resource gap. AL2L aims to address this limitation by adapting dynamically malleable jobs to the available resource windows.

3.2 Application-aware cloud placement

Cloud computing emerged as a way to sell spare compute capacity from under-utilized infrastructures [4] and has since evolved into a model that offers variable, on-demand, accountable, and instantly available resources to run different types of applications using different delivery service models [23], e.g. Infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). Services, batch jobs, and data-intensive applications are some of the types of applications that are common in cloud data centers [31]. Typically, parallel cloud applications are not as tightly coupled as in HPC environments [38] and very variable levels of utilization of the allocated resources are observed [13].

The NIST definition of cloud computing captures its objectives: no wait time to provide resources to services, dynamically increase/decrease resources allocated to an application, and support for diverse applications and platforms at the lowest cost possible. To comply with this definition cloud computing brings a series of required infrastructure characteristics to the data center: commodity hardware, nodes with large RAM capacity (to support multiple VMs on the same node) and high-speed networks (but not low-latency, like in HPC systems). Cloud applications also present various scheduling challenges. Some applications must be run immediately or at deadlines (e.g., services), allocation on-demand imposes the use of prediction methods to perform resource planning [4] and the performance of an application is hard to predict as it depends on the code itself, the supported load and the presence of other applications hosted on

the same resources [43]. However, cloud environments typically also offer opportunities and tools not present in HPC environments. For example, applications can be migrated between compute nodes [4] and applications not fully using their allocated resource capacity can sometimes be overbooked without noticeable performance degradation [36].

Scheduling in cloud datacenter starts as a placement decision. When more capacity is needed by an application it can be allocated a larger share of its host machine (vertical elasticity) or increases the number of instances of the application on other machines (horizontal elasticity). Placement, to optimize a set of target functions (e.g., energy efficiency, resource utilization, quality of service) while satisfying a set of constraints (e.g., memory or affinity requirements of a VM) by assigning resources to applications, is an NP hard problem [24].

Compared to HPC environments, the variability of cloud applications can be seen to bring different performance models and requirements to the cloud data center; e.g., batch jobs can wait to be executed, while services can not; batch job performance is measured by execution time while services focus on quality of service aspects such as response time. There are different approaches to capture applications models in a scheduler [33], with A2L2, we borrow a construct from cloud environments: two level scheduling, i.e. there is one scheduler per application type that interact with a single a resource manger which governs how the resources are assigned to each scheduler. In Section 4.1 we present the model and its underlying challenges.

3.3 HPC trends

Currently, the HPC community is striving towards Exascale, i.e. systems with Exaflop peak compute capacity. When reaching this level of capacity the number of cores per node is expected to increase significantly [6] while node memory capacity and I/O latency and bandwidth are expected to not be able grow at the same pace (due to power consumption and technology limitations respectively). Thus, the gap between CPU and I/O capacity is expected to increase, and I/O to become a performance cap for certain applications. As a consequence, a set of solutions for reducing I/O limitations are being investigated for HPC systems:

- Burst buffers on selected compute nodes: solid state memory that behaves like a cache level between main memory and the external IO systems, with larger capacity than the node main memory. The I/O latency of the burst buffer is expected to be significantly smaller than parallel file system's. Applications may decide to use this burst buffer as a local storage system [21].
- I/O dedicated nodes present a similar function to the burst buffers but out of the compute nodes, higher capacity and slightly larger latency [6].

Burst buffers can bring fast storage to (or close to) HPC nodes (i.e. dedicated analysis nodes), which can increase the performance of data-intensive applications as they are often I/O limited. As a side effect, these proposed changes reduce the differences between cloud and HPC environments as some nodes present storage and many cloud infrastructures already incorporate SSDs on their systems [16]. In Section 4.3.2 we explore the idea of using burst buffers In the context of the A2L2 model o increase data-intensive application performance in HPC environments.

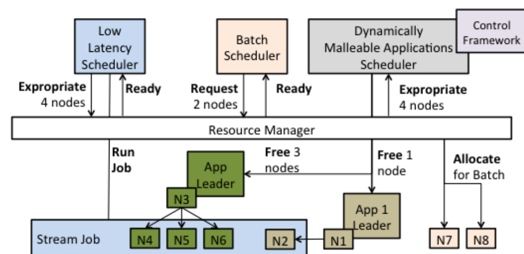


Figure 2: A2L2 with three schedulers. The low-latency scheduler requests expropriated resources, which are taken from the dynamically malleable applications.

4. A2L2 SCHEDULING MODEL

Section 3 compares HPC and cloud systems, identifying the methods used in cloud to tackle the challenges induced by the presence of data-intensive applications. In this section we discuss techniques to realize A2L2: a cloud-inspired scheduling model aimed to enable application-aware scheduling of heterogeneous workloads, that takes advantage of the presence of dynamically malleable applications to improve utilization and enable low-latency allocation of resources.

4.1 Application-aware scheduling

The core of the AL2L approach is a two level resource manager model that serves multiple schedulers at the same time. Each application is managed by a scheduler specific for that application type that captures the application's performance characteristics and requirements. The resource manager controls the resource allocation among schedulers. An example with three schedulers is presented in Figure 2. This schema is a hybrid between the two level and shared state approaches presented in [33] to schedule heterogeneous workloads (services, batch jobs, workflows) on cloud infrastructures. At the core of our approach is the resource manager, which requires the following system characteristics to enable application-aware scheduling and the other features presented in this work:

- More than one scheduler uses the resource manager.
- The resource manager offers a stable and consistent view of the state of the resources to all schedulers.
- All resource allocations follow a request and offer resource protocol jointly controlled by the resource manager and the schedulers. Periodically the resource manager allow all the schedulers to request resources at the same time. This process has two stages. First, *schedulers request* free resources to run an application and, if not all requests can be satisfied, a conflict resolving method is applied (described later). Then, the *resource borrowing* phase starts: schedulers are offered any remaining free resources to increase the allocation of already running applications. Borrowed resources are considered free in the next request phase and it is the responsibility of the allocating scheduler to deallocate borrowed resources when needed.

- Dynamically malleable application schedulers must support *resource expropriation* - de-allocation of resources at the request of the resource manager.

- Schedulers can request expropriated resources to run time-critical applications if there are not enough free resources to run it.

In our model, each scheduler is isolated from the rest, their decisions are made solely on their individual performance models and policies. The resource manager controls the resources assigned to each scheduler. This control is enforced by individual decision to solve resource request conflicts between schedulers (Conflicts arise when there are not enough resources to satisfy all the resource requests). The aggregated effect of all decisions is intended to produce a resource sharing policy. These are some of the resource sharing policies to be explored in our work:

- Weighted random: Probabilities of each scheduler are configured. Schedulers with higher probability will submit more jobs regardless of their resource time consumption.

- Fairshare: Each scheduler should be able to use a predefined share of the resource time for a given time window (e.g. 20% of the produced core-hours in one day).The resource manager keeps track of the past consumed resource time and decisions are aimed to bring the system to the target shares [29]. Schedulers' shares can be adjusted to different values on periods of time to enforce a temporary higher presence of certain types of applications.

This two-level model may produce clear benefits. Although at the core designed for a cloud type usage scenario, high utilization is enforced by the schedulers, that are built around the HPC applications performance models and the objectives of high utilization and capacity. The flexible backfilling may increase the utilization over classic backfilling. At the same time, the model separates how applications share the system from application specific policies (classic batch schedulers aggregate the application specific priorities dissolving both the effect of resource sharing policies and individual application sharing).

In addition, this model supersedes the batch scheduler as it allows application specific scheduling, i.e. introduction of new types of applications do not require a change of the whole scheduler system (merely the addition of a new scheduler for that application type). Also, it provides a level of fault-tolerance as the parts are decoupled and can easily be replicated. It aims to enforce a usage share among applications through the different policies on the resource conflict resolution. Finally, flexible backfilling and resource expropriation has great potential for increasing resource utilization and adding capabilities to the system (more details in the next sections).

4.2 Dynamic allocation of resources

Classic HPC jobs receive resource allocations that remain constant throughout the entire job make-span. In contrast, cloud systems are capable of increasing and reducing the resource allocations of applications to cope with variations in load and demand. Use of dynamically malleable applications in dynamic resource management has the potential for disrupting how HPC resources are scheduled and enabling more flexible backfilling and resource expropriation for time-critical applications.

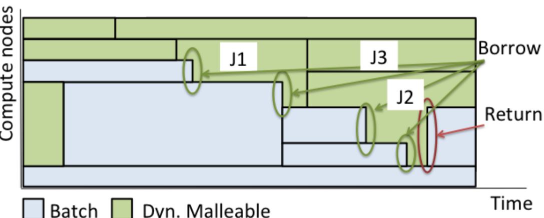


Figure 3: J1 and J2 borrow free resources (flexible backfilling). J2 returns the borrowed resources when the batch scheduler requests them.

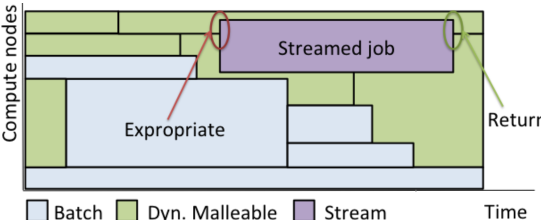


Figure 4: Resources expropriated from two jobs. After the streamed job is completed, the expropriated resources are returned.

4.2.1 *Flexible backfilling*

As presented in Section 3.1, resource fragmentation is a problem that can occur to batch schedulers. We propose the use of flexible backfilling: resource allocations for dynamically malleable jobs are increased (with little performance penalty) to use non-allocated resources without delaying other applications. At the end of the resource request phase, the resource manager re-offers unused (but not borrowed) resources to schedulers as *borrowable resources* that can be used to increase the allocations of already running applications. However, as borrowed resources are still considered available (free) for new applications allocations, the resource manager may require schedulers to de-allocate borrowed resources at any time. The theoretical result of this technique is illustrated in Figure 3, where applications with changing resource allocations can be seen in contraposition to the classic batch scheduler example of Figure 1. In this example, two dynamically malleable applications (stacked on the upper part of the figure) reduce their resource allocations, which are used to schedule a *streamed job*. Once the job finishes, one of the applications from which the resources were expropriated is still running and the resources are returned to that application. Remaining resources are given to another dynamically malleable application.

4.2.2 *Low-latency allocation*

Many scientific experiments produce large amounts of data that are stored for later analysis on HPC systems. However, some experiments would benefit from processing data while

the experiment is running. This can be achieved by aligning the experiment with an advance reservation of HPC resources. However, the use of advance reservations frequently leads to low utilization as they cannot fit the exact experiment time window [1]. Instead, a special type of job submission that aims for jobs to be run within a short time period is needed (equivalent to the on-demand nature of cloud): low-latency allocation of resources.

Low-latency allocation of resources are added to our model by using *resource expropriation* for temporary partial preemption of dynamically malleable jobs resources. A step-by-step example of this technique can be followed in Figure 2 and its corresponding effect observed in Figure 4: the low-latency scheduler requests resources from the resource manager, which expropriates resources from the dynamically malleable applications scheduler. The resource manager evaluates what resource reduction will impact application performance objectives the least and enforce a decision on the two applications. The resources are then used by the low-latency application. Once this application terminates, the expropriated resources are returned to the original scheduler. This scheduler re-assigns these resources to the dynamically malleable applications.

In batch jobs, partial preemption brings the negative effect of lost work. However, the technique is only used on dynamically malleable applications whose performance model allows their resource pool to be resized with a small performance penalty. If a processing node is stopped, its produced output data can be stored and its remaining input data transferred to another node to be processed. The overall performance of the job decreases but no work is lost. The application can continue with reduced resources, but if more resources are allocated to it, its performance may return to the pre-expropriation level.

This technique is inspired by cloud techniques such as Brownout [17] (graceful degradation of application quality of service as a means to cope with increased resource load) and overbooking [36] (packing physical hosts with VMs which do not fully utilize their allocation to better utilize the resources).

4.3 Dynamically malleable management

Dynamically malleable jobs do not have a hard constraint on the required resources. A larger resource allocation implies a shorter running time and vice versa. In classical HPC scenarios, users specify the required resource allocation and estimated runtime for jobs. This poses two challenges. First, users have to estimate the resource allocation and consequent runtime for an application whose performance is hard to predict (and thus, easy to under/overestimate). Second, it eliminates the freedom of the scheduler to adapt the job's allocation to achieve overall targets.

We propose performance-based management for dynamically malleable applications: users provide deadline by when the job should be completed and application schedulers have the freedom to dynamically allocate resources to application so deadlines are met. The benefits for the scheduling model are the following:

- Eliminate user over/under estimations: Workflows run multiple phases with different operations on resources where performance might be unknown. It is complex to estimate the required resources and runtime for each phase. However, if the user provides a deadline, the scheduler can adapt the resource allocation dynamically and avoid over/under allocation.
- Malleable applications can be used for flexible backfilling or as a source of expropriated resources.

4.3.1 *Deadline based resource management*

In the proposed A2L2 approach, an specific scheduler will exist for dynamically malleable applications. This scheduler has two goals: to control and manage dynamically malleable jobs (job manager) and calculate the resources needed for these jobs to meet their deadlines (resource calculator). The job manager is meant to be built over one of the existing cloud execution frameworks for dynamically malleable applications (e.g., Hadoop, Spark). It will have the responsibility of controlling the job workers on each allocated node, manage the distribution of input data among them, allocate new resources to a job, and de-allocate them if needed. The scheduling functions will interact with the job manager to free resources or the possibility of using borrowed ones.

The resource calculator will inform the job manager on the resources allocated to the jobs to meet the user provided deadlines. Job deadlines are a well known model for expressing an execution target [12]. Deadlines can be synchronized with real life events and, hence, easy to understand by the user. We propose to study what methodology should be used to perform these calculations and enforce them on dynamically malleable workflows. As presented in previous work [12], it implies translating the overall deadline to per-stage deadlines and estimating the resource performance for each stage. The resources performance estimation is used to calculate the required resource size to meet the deadline of a running stage workflow. Finally, a job manager uses the calculation to alter the resource set assigned to the stage.

Finally, the dynamic malleable scheduler must support resource expropriation and resource borrowing. One of the challenges will be to determine the best heuristic to calculate what resources and from which jobs should be released to minimize the impact in terms of job's execution slowdown. A similar heuristic will be investigated to determine the jobs that should receive extra resources in case the scheduler can borrow resources.

4.3.2 *Performance enhancements*

The throughput of dynamically malleable applications highly depend on the system's I/O performance. In cloud environments, data locality is often leveraged though distributed file systems [16]. However, in most HPC systems, compute nodes use remote parallel file systems and have little or no local storage on nodes. Application performance will depend on the data access patterns [26] and the network capacity of the data center. This is especially important in Exascale systems as I/O operations will be more expensive at that level [6].

Another performance concern comes from A2L2's ability to dynamically change the resource allocation of applications. In this case, although the performance penalty is smaller than that of full preemption of jobs, the time to recreate the application data on nodes may affect the overall performance.

In order to address these two challenges we propose to use memory-centric approaches when possible. For storage-centric applications, burst buffers and I/O nodes can host a filesystem, equivalent to the distributed file system used in

cloud environments. Application data has to be preloaded on nodes but, once loaded, the application read/write patterns will not impact the performance of the I/O network or the central parallel file system [19]. Also, this distributed file system can be used in a semi-persistent way: if a node is expropriated from an application, the file system is not erased. Once the node is free and the application is restored, the application can keep using the near storage system. The only performance penalty will come from checking what input data has been already processed. This penalty can be neutralized by using eventual data consistency techniques on the distributed file system.

5. CHALLENGES AND FUTURE WORK

Based on the A2L2 model outlined in this paper, further research poses three immediate challenges: further detailed modeling of the different components of the scheduling environment, implementation of the proposed model, and evaluation of the quality of scheduling produced. These challenges are embodied in the various system components for an HPC resource management schema with batch job scheduling, dynamic resource allocation on a per job basis, application level control, resource borrowing, flexible backfilling, resource expropriation, low-latency job allocation, workflow aware scheduling, and data locality enforcement for data-intensive applications on HPC.

The work and research on these challenges cannot be performed as a single line of work, but the aggregation of three parallel lines of work: the first is to characterize the workload of a set of reference systems both on HPC and cloud workloads. We plan to study traces from systems at NERSC (the US National Energy Research Computing Center), the Parallel Workload Archive [10], HPC2N (the High Performance Computing Center North, Sweden), and Google cluster data [39]. This line of investigation has two purposes: performing a comparison between HPC and cloud (to verify the feasibility of using cloud techniques in HPC) and producing a model for synthetic workloads to test our scheduler. Our previous works has initiated this line of research with the workload analysis of two NERSC systems with an special focus on its evolution throughout the systems' lifetime [32].

The second research line aims at building a resource performance model that includes node compute capacity, memory, network, and I/O capabilities. In particular, analysis of the effect on the memory and I/O models versus the presence of node burst buffers, as well as the expected Exascale increases in the gap between capacity of processing units and I/O systems, are topics for study.

The third line aims at creating and evaluating three components: a scheduling suite (implementing our model), a resource emulator (based on the resource model) and a workload emulator (that captures characteristics of workload model). The scheduling suit will be based on community open source scheduling software and will be developed for real system deployment. The resource emulator will wrap the scheduler to run it in a test environment. The workload emulator will model different types of job submissions and user behaviors. This approach will support the implementation test and algorithms evaluation processes. Each model feature will be evaluated comparing its performance (in terms of utilization, turnaround time or enabled capacity) against a reference scenario using a classical HPC scheduling algorithms. The ultimate goal of this effort to produce a software suite to run over a real HPC system.

6. RELATED WORK

Applications whose resource allocation needs are variable are not new to HPC centers. In 1996 Feitelson and Rudolph [11] did a classification that identifies three types of jobs with different degrees of flexibility: moldable, malleable, and evolving.

Moldable applications [7, 18], are those whose degree of parallelism can be chosen just before they are started but do not support any changes after that [11]. A scheduler can decide on the geometry of a moldable job by considering the current state of the resources, the geometry of waiting jobs, and the system's target functions: e.g., high utilization, or short turnaround time. However, unlike dynamically malleable applications, moldable applications don't allow any changes of their resource allocations once allocated. This disables any further scheduling decisions for them (apart from abortion of the job).

Malleable applications support changes in their resource set during execution time without stopping execution [11]. An example are MPI malleable applications [37, 22]. We define dynamic malleable applications as a subset of of the malleable applications characterized for being data intense and organized in a workflow

These applications requires scheduler support as they need to be aware of any resource offering or reduction. This support has appeared in highly distributed works such as grid scheduling [28, 3] but not in pure HPC schedulers. In this work, we propose a model exclusively for HPC systems, with a fixed size resource set and no possibility to schedule jobs to external resources.

Evolving jobs have resource requirement changes throughout their execution time [11] and the system must satisfy them or their execution will not continue. Some grid scheduling systems offer support to these type of applications [20]. HPC schedulers have limited support for evolving jobs [30] and it is based on advance resource reservations. Our model proposal for HPC offers a possibility of freeing resources from dynamically malleable applications to serve evolving jobs or other rigid jobs with higher priority without using advance reservations.

Finally, multi-level scheduling approaches have been investigated in the past in the grid [2, 25] and cloud communities [14, 33]. The resulting schedulers cannot be directly applied to a HPC system. However, the two-level architecture of A2L2 is inspired by some of the models presented in that work.

7. CONCLUSIONS

This paper presents a comparative analysis between HPC and cloud scheduling methods and extrapolates on insights from this comparison to outline A2L2: a scheduling model for HPC systems that takes application characteristics into consideration when realizing low-latency, on-demand allocation of resources in HPC environments. The first insight of this analysis is that applications with different performance models are present in both HPC and cloud environments. To address this, we propose a multi-level HPC scheduler model that separates application specific policies from how compute time is shared by application types. The second in-

sight is the proposal to use the flexible nature of dynamically malleable jobs to enable different features in HPC scheduling. The presence of jobs whose resource allocations can be scaled up and down during execution, with very low performance penalties, allows both a more flexible backfilling model (where job sizes are changed to free space for backfilling jobs as well as to improve the fit of backfilling jobs to resource allocation gaps) and use of resource expropriation in order to allow admission of time-critical application jobs on-demand.

8. ACKNOWLEDGMENTS

This work is funded by the Swedish Government's strategic effort eSSENCE, the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS), and the Office of Science, Office of Advanced Scientific Computing Research (ASCR) of the U.S. Department of Energy under Contract Number DE-AC02-05CH11231.

9. REFERENCES

[1] M. A. Bauer, A. Biem, S. McIntyre, N. Tamura, and Y. Xie. High-performance parallel and stream processing of x-ray microdiffraction data on multicores. In *Journal of Physics: Conference Series*, volume 341, page 012025. IOP Publishing, 2012.

[2] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, et al. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[3] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *2007 IEEE International Conference on Cluster Computing*, pages 372–381. IEEE, 2007.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[5] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *IEEE International Symposium on Cluster Computing and the Grid. CCGRID 2005*, volume 2, pages 776–783. IEEE, 2005.

[6] Y. Chen. Towards scalable I/O architecture for exascale systems. In *Proceedings of the 2011 ACM international workshop on many-task computing on grids and supercomputers*, pages 43–48. ACM, 2011.

[7] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *Proceedings 15th International Parallel and Distributed Processing Symposium*. IEEE, 2001.

[8] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 2011.

[9] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *IEEE Fourth International Conference on eScience*, pages 277–284. IEEE, 2008.

[10] D. Feitelson. Parallel workloads archive. 71(86):337–360, 2007. `http://www.cs.huji.ac.il/labs/parallel/workload`.

[11] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[12] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[13] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Networked Systems Design and Implementation, (NSDI)*, volume 11, pages 22–22, 2011.

[15] M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC resource management systems: Queuing vs. planning. In *Job Scheduling Strategies for Parallel Processing*, pages 1–20. Springer, 2003.

[16] S.-H. Kang, D.-H. Koo, W.-H. Kang, and S.-W. Lee. A case for flash memory ssd in hadoop applications. *International Journal of Control and Automation*, 6(1):201–210, 2013.

[17] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM, 2014.

[18] C. Klein and C. Perez. An RMS architecture for efficiently supporting complex-moldable applications. In *2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, pages 211–220. IEEE, 2011.

[19] S. Krishnan, M. Tatineni, and C. Baru. myHadoop-Hadoop-on-Demand on Traditional HPC Resources. *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, 2011.

[20] W. Lammers. *Adding support for new application types to the Koala grid scheduler*. PhD thesis, Citeseer, 2005.

[21] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[22] K. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. Dynamic malleability in iterative mpi applications. In *7th Int. Symposium on Cluster Computing and the Grid*, pages 591–598, 2007.

[23] P. Mell and T. Grance. The NIST definition of cloud computing. 2011.

[24] K. Mills, J. Filliben, and C. Dabrowski. Comparing vm-placement algorithms for on-demand clouds. In *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 91–98. IEEE, 2011.

[25] H. Mohamed and D. Epema. Koala: a co-allocating grid scheduler. *Concurrency and Computation: Practice and Experience*, 20(16):1851–1876, 2008.

[26] E. Molina-Estolano, M. Gokhale, C. Maltzahn, J. May, J. Bent, and S. Brandt. Mixing hadoop and hpc workloads on parallel filesystems. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 1–5. ACM, 2009.

[27] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, anduser runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE transactions on parallel and distributed systems*, 12(6):529–543, 2001.

[28] A. P. Nascimento, A. C. Sena, C. Boeres, and V. E. Rebello. Distributed and dynamic self-scheduling of parallel mpi grid applications. *Concurrency and Computation: Practice and Experience*, 19(14):1955–1974, 2007.

[29] P.-O. Östberg, D. Espling, and E. Elmroth. Decentralized scalable fairshare scheduling. *Future Generation Computer Systems - The International Journal of Grid Computing and eScience*, 29:130–143, 2013.

[30] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf. A batch system with fair scheduling for evolving applications. In *2014 43rd International Conference on Parallel Processing (ICPP)*, pages 351–360. IEEE, 2014.

[31] B. P. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51. IEEE, 2009.

[32] G. Rodrigo, P.-O. Östberg, E. Elmroth, K. Antypass, R. Gerber, and L. Ramakrishnan. HPC system lifetime story: Workload characterization and evolutionary analyses on NERSC systems. In *The 24th International ACM Symposium on High-Performance Distributed Computing (HPDC)*, 2015.

[33] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[35] A. Snavely, X. Gao, C. Lee, L. Carrington, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance modeling of HPC applications. *Advances in Parallel Computing*, 13:777–784, 2004.

[36] L. Tomás and J. Tordsson. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, page 5. ACM, 2013.

[37] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on MPI jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 215–224. IEEE Computer Society, 2004.

[38] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl. Scientific cloud computing: Early definition and experience. In *High Performance Computing and Communications*, volume 8, pages 825–830, 2008.

[39] J. Wilkes. More Google cluster data. Google research blog, Nov. 2011. Posted at `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`.

[40] R. Williams, I. Gorton, P. Greenfield, and A. Szalay. Guest editors' introduction: Data-intensive computing in the 21st century. *Computer*, 41(4):0030–32, 2008.

[41] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Middleware 2010*, pages 1–20. Springer, 2010.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[43] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

Paper IV

ScSF: A Scheduling Simulation Framework

Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan

In *21th Workshop on Job Scheduling Strategies for Parallel Processing,* Accepted, Springer International Publishing, 2014.

ScSF: A Scheduling Simulation Framework

Gonzalo P. Rodrigo*, Erik Elmroth, Per-Olov Östberg, and
Lavanya Ramakrishnan+

Department of Computing Science, Umeå University, SE-901 87, Umeå Sweden
Lawrence Berkeley National Lab, 94720, Berkeley, California+
{gonzalo,elmroth,p-o}@cs.umu.se,lramakrishnan@lbl.gov

Abstract. High-throughput and data-intensive applications are increasingly present, often composed as workflows, in the workloads of current HPC systems. At the same time, trends for future HPC systems point towards more heterogeneous systems with deeper I/O and memory hierarchies. However, current HPC schedulers are designed to support classical large tightly coupled parallel jobs over homogeneous systems. Therefore, There is an urgent need to investigate new scheduling algorithms that can manage the future workloads on HPC systems. However, there is a lack of appropriate models and frameworks to enable development, testing, and validation of new scheduling ideas.
In this paper, we present an open-source scheduler simulation framework (ScSF) that covers all the steps of scheduling research through simulation. ScSF provides capabilities for workload modeling, workload generation, system simulation, comparative workload analysis, and experiment orchestration. The simulator is designed to be run over a distributed computing infrastructure enabling to test at scale.
We describe in detail a use case of ScSF to develop new techniques to manage scientific workflows in a batch scheduler. In the use case, such technique was implemented in the framework scheduler. For evaluation purposes, 1728 experiments, equivalent to 33 years of simulated time, were run in a deployment of ScSF over a distributed infrastructure of 17 compute nodes during two months. Finally, the experimental results were analyzed in the framework to judge that the technique minimizes workflows' turnaround time without over-allocating resources.
Finally, we discuss lessons learned from our experiences that will help future researchers.

1 Introduction

In recent years, high-throughput and data-intensive applications are increasingly present in the workloads at HPC centers. Current trends to build larger HPC systems point towards heterogeneous systems and deeper I/O and memory hierarchies. However, HPC systems and their schedulers were designed to support large communication-intensive MPI jobs run over uniform systems.

* Work performed while working at the Lawrence Berkeley National Lab

The changes in workloads and underlying hardware have resulted in an urgent need to investigate new scheduling algorithms and models. However, there is limited availability of tools to facilitate scheduling research. Current simulator frameworks largely do not capture the complexities of a production batch scheduler. Also, they are not powerful enough to simulate large experiment sets, or they do not cover all its relevant aspects (i.e., workload modeling and generation, scheduler simulation, and result analysis).
Scheduling simulators: Schedulers are complex systems and their behavior is the result of the interaction of multiple mechanisms that rank and schedule jobs, while monitoring the system state. Many scheduler simulators, like Alea [14], include state of art implementations of some of these mechanisms, but do not capture the interaction between the components. As a consequence, hypotheses tested on such simulators might not hold in real systems.
Experimental scale: A particular scheduling behavior depends on the configurations of the scheduler and characteristics of the workload. As a consequence, the potential number of experiments needed to evaluate a scheduling improvement is high. Also, experiments have to be run for long time to be significant and have to be repeated to ensure representative results. Unfortunately, current simulation tools do not provide support to scale up and run large numbers of long experiments. Finally, workload analysis tools to correlate large scheduling result sets are not available.

In this paper, we present ScSF, a scheduling simulation framework that covers the scheduling research life-cycle. It includes a workload modeling engine, a synthetic workload generator, an instance of Slurm wrapped in a simulator, a results analyzer, and an orchestrator to coordinate experiments run over a distributed infrastructure. This framework will be published as open source, allowing user customization of modules. We also present a use case that illustrates the use of the scheduling framework for evaluating a workflow-aware scheduling algorithm. Our use case demonstrates the modeling of the workload of a peta-scale system, Edison at NERSC (National Energy Research Scientific Computing Center). We also describe the mechanics of implementing a new scheduling algorithm in Slurm and running experiments over distributed infrastructures.

Specifically, our contributions are:

- We describe the design and implementation of scalable scheduling simulator framework (ScSF) that supports and automates workload modeling and generation, Slurm simulation, and data analysis. ScSF will be published as open source.
- We detail a case study that works as a guideline to use the framework to evaluate a workflow-aware scheduling algorithm.
- We discuss the lessons learned from running scheduling experiments at scale to help scheduling researchers in the future.

The rest of the paper is organized as follows. In Section 2, we present the state of art of scheduling research tools and the previous work supporting the framework. The architecture of ScSF and the definition of its modules are presented in Section 3. In Section 4 we describe the steps to use the framework

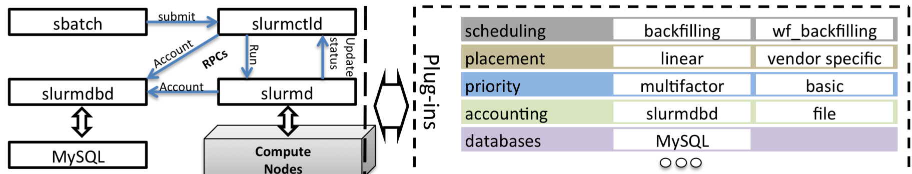


Fig. 1: Slurm is composed by three daemons: slurmctld (scheduler), slurmd (compute nodes management and supervision), and slurmdbd (accounting). A plug-in structure wraps the main functions in those daemons.

to evaluate a new scheduling algorithm. In Section 5, we present some lessons learned while using ScSF at scale. We present our conclusions in Section 6.

2 Background

In this section, we describe the state of art and challenges in scheduling research.

2.1 HPC schedulers and Slurm

The scheduling simulation framework is aimed to support research on HPC scheduling. The framework incorporates a full production scheduler and is modified to include new scheduling algorithms to be evaluated.

Different options were considered for the framework scheduler. Moab (scheduler) plus Torque (resource manager) [7], LSF [11], and LoadLeveler [13] have been quite popular in HPC centers. However, their source code is not easily available which makes extensibility difficult. The Maui cluster scheduler is an open-source precursor of Moab [12], however it does not support current system needs. Slurm is one of the most popular recent workload managers in HPC (used in 5 of top 10 HPC systems [2]). It was originally developed at Lawrence Livermore National Laboratory [20], now mantained by SchedMd [2], and it is available as open source. Also, there are publicly available projects to support simulation in it [19]. Hence, our simulator framework is based on Slurm.

As illustrated in Figure 1, Slurm is structured as a set of daemons that communicate over RPC calls:

slurmctld is the scheduling daemon. It contains the scheduling calculation functions, hosts the waiting queue, receives batch job submissions from users, and distributes work across the instances of slurmd.

slurmd is the worker daemon. Ther can be one instance per compute node or a single instance (front-end mode) managing all nodes. It places and runs work in compute nodes and reports the resources status to slurmctld. The simulator uses front-end mode.

sbatch is a command that wraps the slurm RPC API to submit jobs to slurmctld. Most commonly used by users.

Slurm has a plug-in architecture. Many of the internal functions are wrapped by C APIs loaded dynamically depending on the configuration files of Slurm (slurmctld.conf, slurmd.conf).

The Slurm simulator wraps Slurm to emulate HPC resources, emulate user's job submission, and speed up Slurm's execution. We extended previous work from the Swiss Supercomputing Center (CSCS, [19]), based on the original one by the Barcelona Supercomputing Center (BSC, [16]). Our contributions increase Slurm's speed up while maintaining determinism in the simulations, and adds workflow support. These improvements are presented in Section 3.5.

2.2 HPC workload analysis and generation

ScSF includes the capacity to model system workloads and generate synthetic ones accordingly. Workload modeling starts with flurries elimination (i.e., events that are not representative and skew the model) [10]. The generator models each job variable with the empirical distribution [15], i.e., it recreates the shape of job variable distributions by constructing a histogram and CDF from the observed values.

2.3 Related work

Previous work [18] proposes three main methods of scheduling algorithms research: theoretical analysis, execution on a real system, and simulation. The theoretical analysis is limited to produce boundary values on the algorithm, i.e. best and worst cases, but does not reason on the regular performance. Also, since continuous testing of new algorithms on large real systems is not possible, simulation is the option chosen in our work.

Available simulation tools do not cover the full cycle of modeling, generation, simulation, and analysis. Also, public up-to-date simulators or workload generators are scarce. As an example, our work is based on the most recent peer reviewed work on Slurm Simulation (CSCS, [19]), and we had to improve its synchronization to speed up its execution. For more grid-like workloads, Alea [14] is an example of a current HPC simulator. However, it does not include a production simulator in its core and does not generate workloads.

For workload modeling, function fitting and user modeling are recognized methods [9]. However, the ScSF's workload model is based on empirical distributions [15], as it produces good enough models and does not require specific information about system users. Also, this work modeling methods are based on the experience of our previous work on understanding workload evolution of HPC systems life cycle [17] and job heterogeneity in HPC workloads [6].

In workload generation, previous work compares close and open loop approaches [21], i.e. taking into account or not the scheduling decisions to calculate the jobs arrival time. ScSF is used in environments with reduced user information, which is needed to create closed-loop models. Thus, ScSF uses an open-loop workload generation model and fill and load mechanisms (Section 3.4) to avoid under and over job submission.

Finally, other workloads and models [8] are available, but are less representative than the one in the use case (Cray XC30 supercomputer, deployed in 2013 with 133,824 cores and 357 TB of RAM). Still, ScSF can run simulations on the models of such systems.

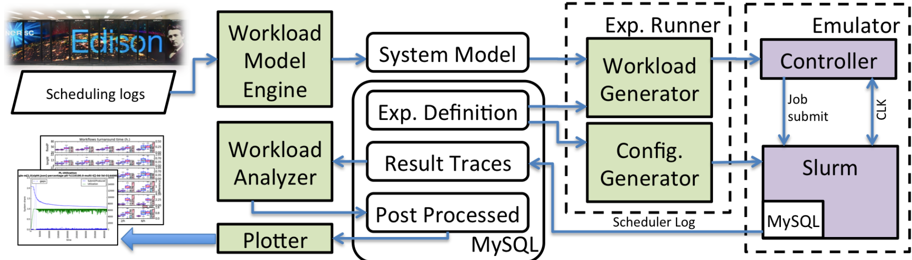


Fig. 2: ScSF schema with green color representing components developed in this work and purple representing modified and improved components.

3 Architecture and simulation process

Figure 2 shows the ScSF's architecture around a MySQL database that stores the framework's data and meta-data. Running experiments based on a reference system requires modeling its workload first by processing the system's scheduling logs in the *workload model engine*. This model is used in the experiments to generate synthetic workloads with similar characteristics to the original ones.

A simulation process starts at the description of the experimental setup in an *experiment definition*, including workload characteristics, scheduler configuration, and simulation time. The *experiment runner* processes experiment definitions and orchestrates experiments accordingly. First, it invokes the *workload generator* to produce a synthetic workload of similar job characteristics (size, inter arrival time) as the real ones in the reference system chosen. This workload may include specific jobs (e.g., workflows) according the the experiment definition. Next, the runner invokes the simulator that wraps Slurm to increase its execution pace and emulate the HPC system and users interacting with it. The simulator sets Slurm's configuration according to the experiment definition and emulates the submission of the synthetic workload jobs. Slurm schedules the jobs over the virtual resources until the last workload job is submitted. At that moment, the simulation is considered completed.

Completed simulations are processed by the *workload analyzer*. The analysis on a simulation result covers the characterization of jobs, workflows, and system. This module includes tools to compare experiments to differentiate the effects of scheduling behaviors on the workload.

In the rest of this section we present the components of the framework involved in these processes.

3.1 Workload model engine

A workload model is composed of statistical data that allows creating synthetic jobs that with characteristics similar to the original ones. To create this model, first, the workload model engine extracts batch job's variable values from Slurm or Moab scheduling logs including wait time, allocated CPU cores, requested wall clock time, actual runtime, inter-arrival time, and runtime accuracy ($\frac{runtime}{requestedWallClockTime}$). Jobs with missing information (e.g. start time), or

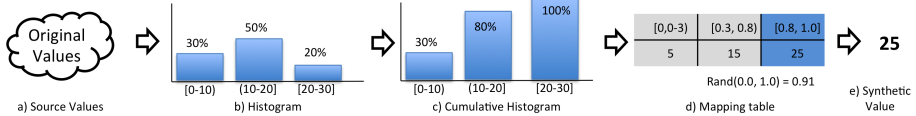


Fig. 3: Empirical distribution constructions for job variables: calculating a cumulative histogram and transforming it into a mapping table.

trace_type	"single": regular experiment."grouped": experiments aggregated.
subtraces	list of single experiments related to this grouped one.
system	name of system to model workload after
workflow_policy	"period": one workflow workflow_period_s. "percent": workflow_share corehours are workflows. "no": no workflows.
manifest_list	list of workflows to appear in the workload.
workflow_handling	workflows submission in workload. "single": pilot job."multi": chained jobs". "manifest": workflow-aware job
start_date	submit time of first job valid for analysis.
preload_time_s	time to prepend to the workload for stabilization.
workload_duration_s	workload stops at start_date+ workload_duration_s.
seed	string to init random number generators.

Table 1: Experiment definition fields

individual rare and very large jobs that would skew the model (e.g. system test jobs) are filtered out.

Next, the extracted values are used to produce the empirical distributions [15] of each job variable as illustrated in Figure 3. A normalized histogram is calculated on the source values. Then, the histogram is transformed into a cumulative histogram, i.e., each bin represents the percentage of observed values that are less or equal to the upper boundary of the bin. Finally, the cumulative histogram is transformed into a table that maps probability ranges on a value: e.g. in Figure 3, bin $(10-20]$ becomes a $[0.3, 0.8)$ probability range as its value is 80% and its left's neighboring bin value is 30%. The probability ranges map to to the mid value of the bin range that they correspond to, e.g., 15 is the mid value of $(10-20]$. This model is then ready to produce values, e.g., a random number (0.91) is mapped on the table, obtaining 25.

Each variable's histograms is calculated with specific bin sizes adapted to its resolution. By default, the bin size for the request job's wall clock time is one minute (Slurm's resolution). The corresponding bin size for inter-arrival time is one second as that is the timestamp log resolution. Finally, for the job CPU core allocation, the bin size is the number of cores per node of the reference system, as in HPC systems node sharing is usually disabled.

3.2 Experiment definition

An experiment definition governs the experimental conditions in an experiment process, configuring the scheduler, workload characteristics, and experiment duration. A definition is composed by a scheduler configuration file and an experiments database entry (Table 1) that includes:

```
{"tasks": [
  {"id":"SWide", "cmd":"./W.py", "cores":480, "rtime":360.0},
  {"id":"SLong", "cmd":"./L.py", "cores":48, "rtime":1440.0,
    "deps": ["SWide"]}]}
```

Fig. 4: WideLong workflow manifest in JSON format.

trace_type and subtraces: The tag "single" identifies those experiments which are meant to be run in the simulator. A workload will be generated and run through the simulator for later analysis. The experiments with trace_type "grouped" are definitions that list the experiments that are the different repetitions of the same experimental conditions in the "subtraces" field.

system model: selects which system model is to be used to produce the workload in the experiment.

workflow_policy: controls presence of workflows in the workload. If set to "no", workflows are not present. If set to "period" a workflow is submitted periodically once every workflow_period_s seconds. If set to "percentage", workflows contribute workflow_share of the workload core hours.

manifest_list: List of pairs (share, workflow) defining the workflows present in the workload: e.g., {(0.4 Montage.json), (0.6 Sipht.json)} indicates that 40% of the workflows will be Montage, and 60% Sipht . The workflow field points to a JSON file specifying the structure of the workflow. Figure 4 is an example of the such file. It includes two tasks, the first running for 10 minute, allocating 480 cores (wide task); and the second running for 40 minutes, allocating 48 cores (long task).The SLong task requires SWide to be completed before it starts.

workflow_handling: This parameter controls the method to submit workflows. The workload generators supports workflows submitted as chained jobs (*multi*), in which workflow tasks are submitted as independent jobs, expressing their relationship as job completion dependencies. Under this method, workflow tasks allocate exactly the resources they need, but intermediate job wait times might be long, increasing the turnaround time. Another approach supported is the pilot job (*single*), in which a workflow is submitted as a single job, allocating the maximum resource required within the workflow for its minimum possible runtime. The workflow tasks are run within the job, with no intermediate wait times, and thus, producing shorter turnaround times. However, it over-allocates resources, that are left idle at certain stages of the workflow.

start_date, preload_time_s, and workload_duration_s: defines the duration of the experiment workload. start_date sets the submit time of the first job in the analyzed section of the workload, which will span until (start_date + workload_duration_s). Before the main section, a workload of preload_time_s seconds is prepended, to cover the cold start and stabilization of the system.

Random seed: this alphanumeric string is used to initialize the random generator within the workload generator. If two experiments definitions have the same parameters, including the seed, their workloads will become identical. If two experiment definitions have the same parameters, but a different seed, their workloads will become similar in overall characteristics, but different as individual jobs (i.e. repetitions of the same experiment). In general, repetitions of the

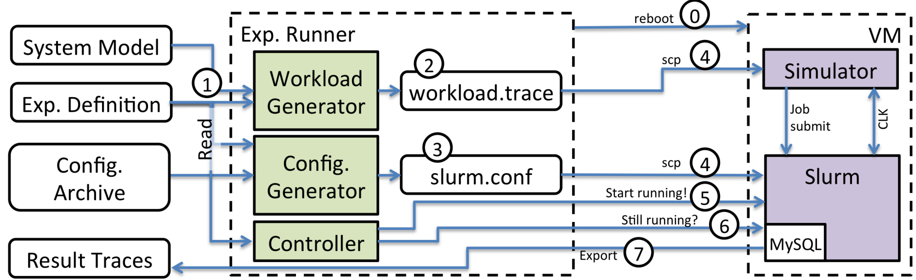


Fig. 5: Steps to run an experiment (numbers circled indicate order) taken by the experiment runner component. Once step seven is completed, the step sequence is re-started.

same experiment with different seeds are subtraces of a "grouped" type experiment.

3.3 Experiment runner

The experiment runner is an orchestration component that controls the workload generation and scheduling simulation. It invokes the workload generator and controls through SSH a VM that contains a Slurm simulator instance. Figure 5 presents the experiment runner operations after being invoked with a hostname or IP of a Slurm simulator VM. First, the runner reboots the VM (step 0) to clear processes, memory, and reset the operative system state. Next, an experiment definition is retrieved from the database (step 1) and the workload generator produces the corresponding experiment's workload file (step 2). This file is transferred to the VM (step 4) together with the corresponding Slurm configuration files (obtained in step 3). Then, the simulation inside the VM (step 5) is started, which will stop after the last job of the workload is submitted in the simulation plus extra time to avoid including the system termination noise in the results. The experiment runner monitors Slurm (step 6), and when it terminates, the resulting scheduler logs are extracted and inserted in the central database (step 7).

Only one experiment runner can start per simulator VM, however multiple runners manage multiple VMs in parallel, which enables a simple scaling mechanism to run experiments concurrently.

3.4 Workload generation

The workload generator in ScSF produces synthetic workloads representative of real system models. The workload structure is presented in Figure 6: All workloads start with a *fill* phase, which includes a group of jobs meant to fill the system. The fill job phase is followed by the stabilization phase, which include 24 hours of regular jobs controlled by a job-pressure mechanism to ensure that there are enough jobs to keep the system utilized. The stabilization phase captures the cold start of the system, and it is ignored in later analysis. The next

Fig. 6: Sections of a workload: fill, stabilization, experiment, and drain. Presented with an the associated utilization that this workload produced in the system.

stage is the experiment phase that runs for a fixed time (72 hours in the figure) and includes regular batch jobs complemented by the experiment specific jobs (in this case workflows). Although not present in the workload, after the workload is completely submitted, the simulation runs for extra time (drain period, configured in the simulator) to avoid the presence of noise from the system termination in the experiment section.

In the rest of this section, we present all the mechanisms involved in detail.

Job generation: The workload generator produces synthetic workloads according to an experiment definition. The system model set in the definition is combined with a random number generator to produce synthetic batch jobs. The system model is chosen among those produced by the workload model engine (Section 3.1). Also, the random generator is initialized with the experiment definition's seed. Finally, the workload generator also supports the inclusion of workflows according to the experiment definition (Section 3.2).

The workload generator fidelity is evaluated by modeling NERSC's Edison and comparing one year of synthetic workload with the system jobs in 2015. The characteristics of both workloads are presented in Figure 7, where the histogram and CDFs for inter arrival time, wall clock limit and allocated number of cores are almost identical. For runtime there are small differences in the histogram that barely impact the CDF.

Filling and load mechanisms: Users of a HPC system submit a job load that fills the systems and creates a backlog of jobs that induces an overall system wait time. The filling and load mechanisms steer the job generation to reproduce this phenomena.

The **load mechanism** ensures that the size of the backlog of jobs does not change significantly. It induces a job pressure (submitted over produced work) close to a configured value, usually 1.0. Every time a new job is added to the workload, the load mechanism calculates the current job pressure t as $P(t) = \frac{coreHoursSubmitted}{coreHoursProduced(t)}$ where $coreHoursProduced = t * coresInTheSystem$. If $P(t) < 1.0$ new jobs are generated and also inserted in the same submit time until $P(t) \geq 1.0$. If $P(t) \geq 1.1$, the submit time is kept as reference, but the job is discarded, to avoid overflowing the system. The effect of the load mechanism is observed in Figure 9, where the utilization raises to values close to one for the same workload parameters as in Figure 8.

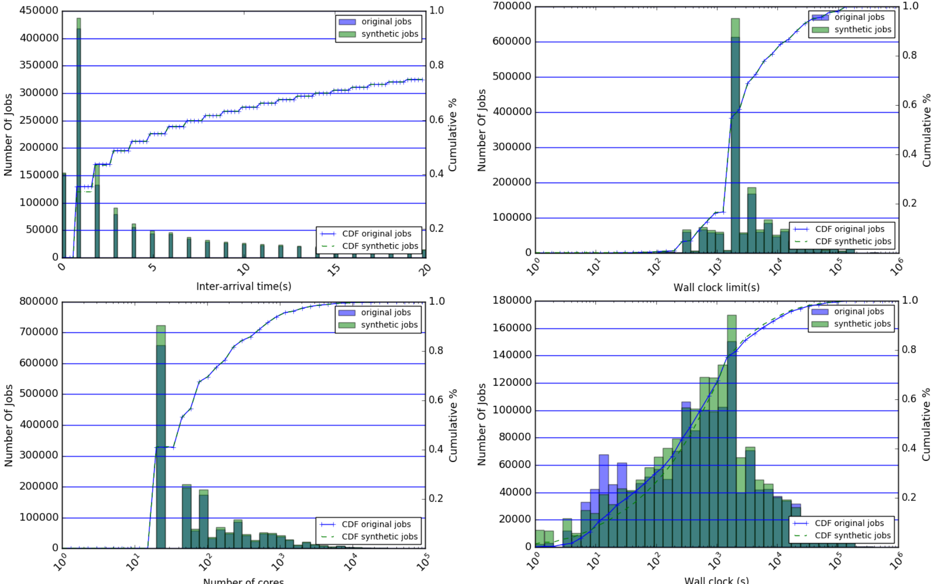


Fig. 7: Job characteristics in a year of Edison's real workload (darker) vs. a year of synthetic workload (lighter). Distributions are similar.

Increasing the job pressure raises system utilization but does not induce the backlog of jobs and associated overall wait time that is present in real systems. As an example, Figure 12a presents the median wait time of the jobs submitted in every minute of the experiment using the load mechanism of Figure 9. Here, the system is utilized but the jobs wait time is very short, only increasing to values of 15 minutes for larger jobs (over 96 core hours) at the end of the stabilization period (vs. the four hours intended).

The **fill mechanism** inserts an initial job backlog equivalent to the experiment configured overall wait time. The filling jobs characteristics guarantee that they will not end at the same time or allocate too many cores. As a consequence, the scheduler is able to fill gaps left when they end. Figure 10 shows an experi-

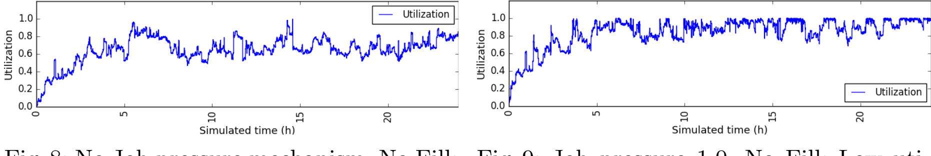


Fig. 8: No Job pressure mechanism, No Fill: Low utilization due not enough work.

Fig. 9: Job pressure 1.0, No Fill: Low utilization due to no initial filling jobs.

Fig. 10: Job pressure 1.0, Fill with large jobs: initial falling spikes.

Fig. 11: Job pressure 1.0, Fill with small jobs: Good utilization, more stable start.

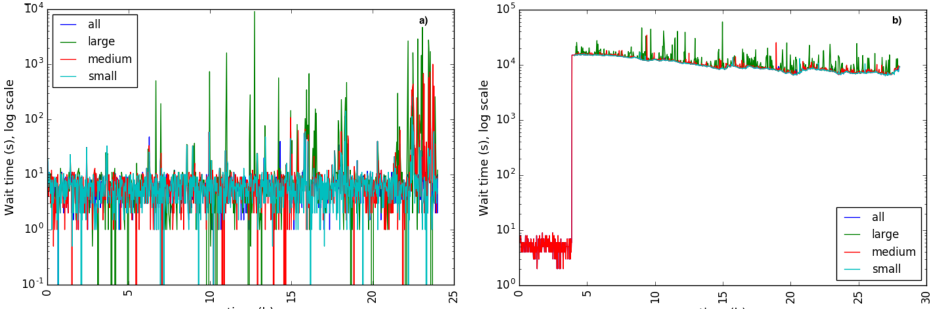


Fig. 12: Median wait time of job's submitted in each minute. a: Job pressure 1.0, not fill mechanism, and thus no wait time baseline is present. b: Job pressure 1.0, fill mechanism configured to induce four hours of wait time baseline.

ment in which the fill job allocations are too big, their allocation is 33,516 cores (1/4 of the system CPU cores count). Every time a fill job ends ($t = 8$, 9, 10, and 11h), one drop in the utilization is observed because the scheduler has to fill a too large gap with small jobs. To avoid this, the filling mechanism calculates a fill job size that induces the desired overall wait time while not producing utilization drops. Fill job size calculation is based on a fixed inter-arrival time, the capacity of the system, and the desired wait time. Figure 11 shows the utilization of a workload which fill jobs are calculated following such method. They are submitted in 10 second intervals creating the soft slope in the figure. Figure 12b shows the wait time evolution for the same workload, sustained around four hours after the fill jobs are submitted.

Customization: The workload generator includes classes to define user job submission patterns. Trigger classes define mechanisms to decide the insertion times pattern, such as: periodic, alarm (at one or multiple time stamps), reprogrammable alarm), or random. The job pattern is set as a fixed jobs sequence, or a weighted random selection between patterns. Once a generator is integrated it is selected by setting a special string in the workflow_policy field of the experiment definition.

3.5 Slurm and the Simulator

Slurm's version 14.3.8 was chosen as the scheduler of the framework: It is one of the most popular open-source workload managers in HPC. Also, as a real scheduler, it includes the effect and interaction of mechanisms such as priority engines, scheduling algorithms, node placement algorithms, compute nodes management, job submissions system, and scheduling accounting. Finally, Slurm includes a simulator to use it on top of an emulated version of an HPC system, submitting a trace of jobs to it, and accelerating its execution. This tool enables experimentation without requiring the use of a real HPC system.

In this section, we present a brief introduction to the simulator's structure and the improvements that we performed on it.

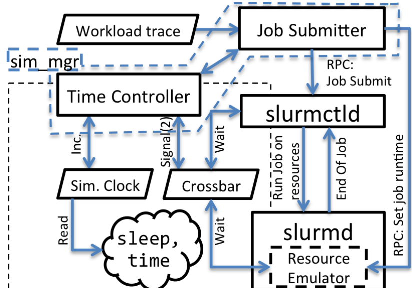


Fig. 13: Slurm simulator architecture. Slurm system calls are replaced to speed-up execution. Scheduling is synchronized. Job submission is emulated.

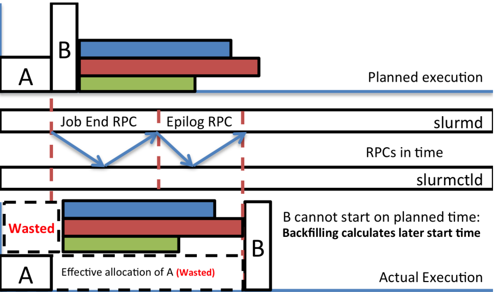


Fig. 14: Simulated time running during RPC communications delay resource de-allocation compromising backfilling's job planning and Job B start.

Architecture: The architecture of Slurm and its simulator is presented in Figure 13. The Slurm daemons (slurmctld and slurmd) are wrapped by the emulator. Both daemons are dynamically linked by the sim_func library which adds the required functions to support the acceleration of Slurm's execution. Also, slurmd is compiled including a resource and job emulator. On the simulator side, the sim_mgr controls the three core functions of the system: execution time acceleration, synchronization of the scheduling processes, and emulation of the job submission. These functions are described bellow.

Time acceleration: In order to accelerate the execution time, the simulator decouples the Slurm binaries from the real system time. Slurm binaries are dynamically linked with the *sim_func* library, replacing the *time*, *sleep*, and *wait* system calls. Replaced system calls use an epoch value controlled by the *time controller*. For example, if the time controller sets the simulated time to 1485551988, any calls to *time* will return 1485551988 regardless of the system time. This reduces the wait times within Slurm: e.g., if the scheduling is configured to run once every 30 simulated second, it may run once every 300 real time milliseconds.

Scheduling and simulation synchronization: The original simulated time pace set by CSCS produces small speed ups for large simulated systems. However, increasing the simulated time pace triggers timing problems because of the RPC nature of Slurm daemon communications. For example, RPC under second real time latency is measured by Slurm as hundreds of simulated seconds.

Increasing the simulation pace has different negative effects. First, timeouts occur triggering multiple RPC re-transmissions degrading the performance of Slurm and the simulator. Second, job timing determinism degrades. Each time a job ends, slurmd sends an RPC notification to slurmctld, and its arrival time is considered the job end time. This time is imprecise if the simulated time increases during the RPC notification propagation. As a consequence low utilization and large job (e.g. allocating 30% of the resources) starvation ocurs. Figure 14 details this effect: A large Job_B is to be executed after Job_A. However, Job_A resources are not considered free until two sequential RPC calls are completed (end of job and epilogue), lowering the utilization as they are not producing work. The later resource liberation also disables Job_B from starting but does not stop the jobs

that programmed are to start after Job_B. As the process repeats, the utilization loss accumulates and Job_B is delayed indefinitely.

The *time_controller* component of the sim_mgr was modified to control a synchronization crossbar among the Slurm functions that are relevant to the scheduling timing. This solves the described synchronization problems by controlling the simulation time and avoiding its increase while RPC calls are traveling between the Slurm daemons.

Job submission and simulation: The job submitter component of the sim_mgr emulates the submission of jobs to slurmctld following the workload trace of the simulation. Before submitting each job, it communicates the actual runtime (different from the requested one) to the resource emulator in slurmd.

Slurmctld notifies slurmd of the scheduling of a job through an RPC. The emulator uses the notification arrival time and job runtime (received from sim_mgr) to calculate the job end time. When the job end time is reached, the emulator forces slurmd to communicate that the job has ended to slurmctld. This process emulates the job execution and resource allocation.

3.6 Workload analyzer

ScSF includes analysis tools to extract relevant information across repetitions of the same experiment or to plot and compare results from multiple experimental conditions.

Value extraction and analysis: Simulation results are processed by the workload analyzer. The jobs in the fill, stabilization, and drain phases (Figure 6) are discarded to then extract (1) for all jobs: wait time, runtime, requested runtime, user accuracy (estimating the runtime), allocated CPU cores, turnaround time, and slowdown grouped by jobs sizes. (2) for all and by type of workflow: wait time, runtime, turnaround time, stretch factor. (3) overall: median job wait time and mean utilization for each minute of the experiment.

The module performs different analyses for different data types. Percentile and histograms analyze the distribution and trend of the the jobs' and workflows' variables. Integrated utilization (i.e., coreHoursProduced/coreHoursExecuted) measure the impact of the scheduling behavior on the system usage.

Finally, customized analysis modules can be implement and added to the analysis pipeline.

Repetitions and comparisons: Experiments are repeated with different random seeds to ensure that observed phenomena are not isolated occurrences. The workload analysis module analyzes all the repetitions together, merging the results to ease later analysis. Also, experiments might be grouped if they differ only in one experimental condition. The analysis module studies these groups together to analyze the effect of that experimental condition on the system. For instance, some experiments are identical except for the workflow submission

method, which affects the number of workflows that get to execute in each experiment. The module calculates compared workflow turnaround times correcting any possible results skew derived from the difference in the number of executed workflows.

Result analysis and plotting: Analysis results are stored in the database to allow review of visualization using the *plotter* component. This component includes tools to plot histograms (Figure 7), box plots, and bar charts on the median of job's and workflow's variables for one or multiple experiments (Figure 17). It also includes tools to plot the per minute utilization (Figures 8 to 11) and per minute median job's wait time in an experiment (Figures 12a and b), which allows to observe dynamic effects within the simulation. Finally, it also include tools to extract and compare utilization values from multiple experiments.

4 Scheduling simulation framework use case

In this section, we describe a use case that demonstrates the use of ScSF. The case study is to, using the simulator, implement and evaluate a workflow-aware scheduling algorithm [5]. In particular, the use case includes modeling of a real HPC system, implementing a new algorithm in the Slurm simulator, and the definition of evaluation experiments. Also, we detail a distributed deployment of ScSF and present some examples of the results to illustrate the scalability of our framework.

4.1 Tuning the model

Experiments to evaluate a scheduling algorithm require workload and system models that are representative. In this use case, NERSC's Edison is chosen as the reference system. Its workload is modeled by processing almost four years of its jobs. Next, a Slurm configuration is defined to imitate Edison's scheduler behavior, including: Edison's resource definition (number of nodes and hardware configuration) FCFS, backfilling with a depth of 50 jobs once every 30s, and a multi-factor priority model that takes into account age (older-higher) and job geometry (smaller-higher). The workload tuning is completed by running a set of experiments to explore different job pressure and filling configurations to induce a stable four hour wait time baseline (observed in Edison).

4.2 Implementing a workflow scheduling algorithm in Slurm

The evaluated algorithm concerns the method to submit workflows to an HPC system. As presented in Section 3.2, workflows are run as pilot jobs (i.e., single job over-allocation resources) or chained jobs (i.e., task jobs linked by dependencies supporting long turnaround times). However, the workflow-aware scheduling

Set	Wf. Submit	#Wfs.	Wf. Pres.	#Pres.	Sim. t.	#Reps	#Exps	Agg. Sim. t.
Set0	aware/single/multi	18	Period	1 per wf.	7d	6	324	2268d
Set1	aware/single/multi	6	Period	6	7d	6	648	4536d
Set2	aware/single/multi	6	Share	7	7d	6	756	5292d

Table 2: Summary of experiments run in ScSF.

[5] is a third method that enables per job task resource allocation, while minimizing the intermediate wait times. In this section, we present the integration of the algorithm in ScSF.

The algorithm integration requires to modify Slurm's jobs submission system, and include some actions on the job queue before and after scheduling happens. First, sbatch, Slurm's job submission RPC, and the internal job_record data structure are extended to support that jobs include a workflow manifest. This enables workflow-aware jobs to be present as pilot jobs attaching a workflow description (manifest).

Second, queue transformation actions are inserted before and after FCFS and backfilling act on it. Before they act, workflow jobs are transformed into task jobs but keeping the original job priority. When the scheduling is completed, original workflow jobs are restored. As a consequence, workflow task jobs are scheduled individually, but, as they share the same priority, the workflow intermediate waits are minimized.

4.3 Creating the experiments

The workflow-aware scheduling approach is evaluated by comparing its effect on workflow turnaround time and system utilization with the pilot and chained job ones. Three versions (one per approach) of experiments are created to compare the performance of the three approaches under different conditions.

Table 2 presents the three sets the experiments created. Workflows in $set0$, express different structure properties to study their interaction with different approaches. $Set1$ studies the effect of the approaches on isolated workflows and includes two synthetic workflows, plus four real (Montage, Sipht, Cybershake, FloodPlain [4]) submitted with different periods (0, 1/12h, 1/6h, 1/h, 2/h, 6/h). $Set2$ studies the effect of the approaches on systems increasing dominated by workflows. It includes the same workflows as $set1$ submitted with different workflow shares (1%, 5%, 10%, 25%, 50%, 75%, 100%). In total, they sum 1728 experiments equivalent to 33 years of simulated time.

Experiments are created and stored using a Python class that is initialized with all the experiment parameters. Synthetic workflows manifest files are created manually following the framework's manifest JSON format. Real workflow manifests are created using a workflow generator from the Pegasus project [4] that captures the characteristics of popular research workflows. ScSF includes a tool to transform the output of this generator into the expected JSON format.

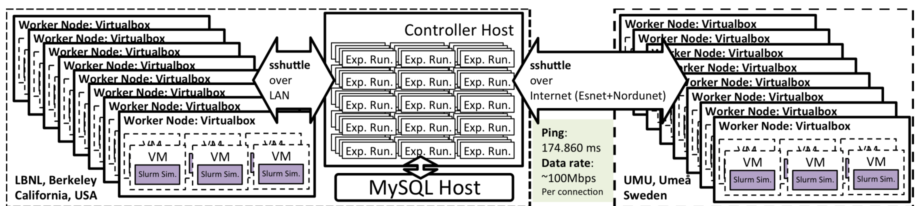


Fig. 15: Schema of the distributed execution environment: VMs containing the Slurm Simulator are distributed in hosts at LBNL and UMU. Each VM is controlled by an instance of the experiment runner in the controller host at LBNL.

4.4 Running experiments in scale

1728 individual experiments that sum 33 years of simulated time are required to run in this use case. Estimating an average speedup of 10x, experiment simulation would require more than three years of real time. In order to reduce the real time required to complete this work, simulation is parallelized to increase throughput.

As presented in Section 3, the minimum experiment worker unit is composed by an instance of the *experiment runner* component and a VM containing the Slurm simulator. As shown in Figure 15, parallelization is achieved by running multiple worker units concurrently. To configure the infrastructure, Virtualbox's hypervisor is deployed on six compute nodes at the Lawrence Berkeley National Lab (LBNL) and 17 compute nodes at Umeå University (UMU). Over them, 161 Slurm Simulator VMs are deployed. Each VM allocates two cores, four GB of RAM, and 20GB of storage. Each compute node has different configurations and thus, the number of VMs per host and their performance is not uniform, e.g., some compute nodes only host two VMs, and some host 15.

All the experiment runners run in a single compute node at LBNL (Ubuntu, 12 cores x 2.8GHz, 36GB RAM). However, VMs are not exposed directly through their host NIC and, required access from the control node over *sshuttle* [3]: a VPN over ssh software that does not required installation in the destination host. Even if both sites are distant, the network is not a significant source of problem since the connection between UMU and LBNL traverses two high performance research networks, Nordunet (Sweden) and ESnet (EU and USA). Latency is relatively low (170-200ms), data-rate is high (firewall capped ≈100Mbits/s per TCP flow), and stability consistent.

4.5 Experimental performance

The experiments wall clock time is characterized as a function of their experimental setup to understand the factors driving simulation speed-up. Figure 16 shows the experiments median runtime of one experiment set, grouped by scheduling method, workflow type, and workflow presence.

For the same simulated time, simulations run longer time under the chained job and workflow-aware approaches compared to pilot job. Also, for the chained job and aware approaches, experiments run longer time if more workflows are

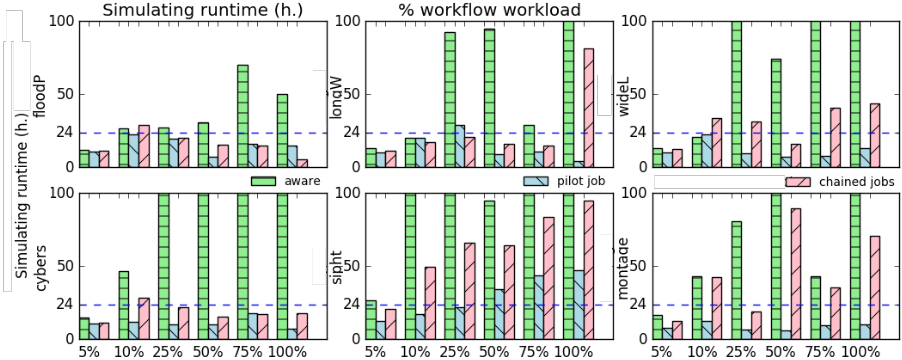


Fig. 16: Median wall clock time for a set of simulation. More complex workloads (more workflows, large workflows) present longer times. Pilot job approach presents shorter times. Simulation time is 168 hours (7 days).

present, or the workflows include more task jobs. As individual experiments are analyzed, longer runtimes, and thus smaller speed-ups, appear to be related to longer runtime of the scheduling passes because of higher numbers of jobs in the waiting queue.

In summary, simulations containing numbers of jobs similar to real system workloads present median runtimes between 10 to 12 hours for 7 days (168h) of simulated time, or 15x speedup. Speed-up degrades as experiments become more complex. Speed-ups under 1 are observed for experiments whose large job count would be hard to manage for a production scheduler (e.g., Montage-75%). To conclude, the limiting factor of the simulations speed-up is the scheduling runtime, which, in this use case, depends on the number of jobs in the waiting queue.

4.6 Analyzing at scale

The analysis of the presented use case required to synthesize the results of 1278 experiments into meaningful, understandable metrics. The tools described in Section 3.6 supported this task.

As an example, Figure 17 condenses the results of 324 experiments (six repetitions per experiment setting): median workflow runtime speed up (left) and value (right) observed for Cybershake, Sipht, and Montage, for different workflow shares and scheduling approaches. Results show that chained job workflows support much longer runtime in all cases, while aware and pilot jobs workflows show shorter and similar runtimes.

5 Lessons learned

The initial design goal of ScSF was readiness, not scale, and its first deployment included four worker VMs. As the number experiments and simulation time grew in the use case (33 years), the resource pool size had to be increased (161 VMs and 24 physical hosts), even expanding to resources in different locations. In this

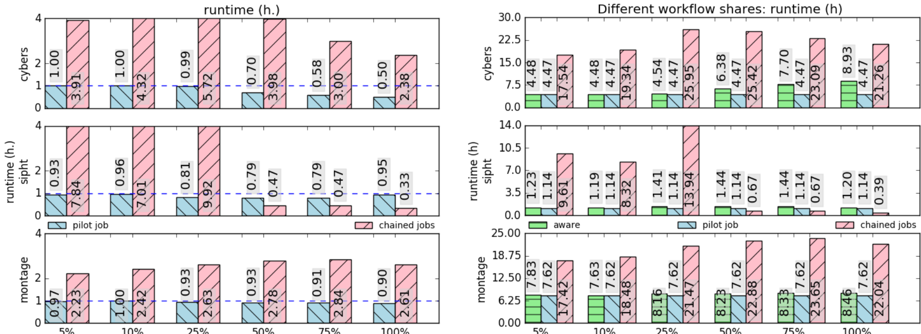


Fig. 17: Comparison of median workflow runtime on different experimental conditions as speed-up (left), and absolute numbers (right). Data of workflows in 108 experiments.

section, we describe some of the resulting problems and what we learned from them.

Loss-less experiment restart is needed: As the framework runs longer and on more nodes, the probability for node reboots becomes higher. In the months of experiments our resources required rebooting due to power cuts, hypervisor failures, VM freezes, and system updates (e.g. we had to update the whole cluster to patch the Dirty Cow exploit [1]).

Unfortunately, in ScSF, rebooting a worker host means that all simulation work in its VMs is lost. Also, if the controller host is rebooted, all the experiment runners are stopped and all the simulation work of all the cluster is lost. For some of the longest experiments, the amount of work lost accounts in days of real time.

The lesson learned is that experiments in ScSF should support graceful pause and restart so resource reboots do not imply loss of work. This would be provided by a control mechanism to pause-restart worker VMs. Also, the experiment runner functionality should be hosted in the worker VM to be paused with the VM, unaffected by any reboot.

Loaded systems network fail: Surges of experiment failures appeared occasionally: Multiple VMs would become temporarily un-responsive to ssh connections when their hypervisor was heavily loaded. Subsequently, the experiment runner would fail to connect to the VM, and the experiment was considered failed. The lesson learned is that saturated resources are unreliable. All runner-VM communications were hardened, adding re-trials, which reduced greatly the fail rate.

Monitoring is important: Many types of failure impact on experiments, such as simulator or Slurm bugs, communication problems, resource saturation in the VMs, or hypervisor configuration issues. Failures are expected, but ScSF lacked the tools and information to quickly diagnose the cause of the problems.

The lesson learned is that monitoring should register metadata that allows quick diagnosis of problems. As a consequence, details level in experiment logs was increased and a mechanism to retrieve Slurm crash debug files was added.

The system is as weak as its weakest link: All ScSF's data and metadata is stored in a MySQL database hosted in the controller host. In a first experiment run, at 80% of completed experiments the hard disk containing the database

crashed, and all the experimental data was lost. Two months of real time were lost and an article submission deadline was missed. Currently, data is subject to periodic backups and the database is replicated.

6 Conclusions

We present ScSF, a scheduling simulation framework that provides tools to support all the steps of the scheduling research cycle - modeling, generation, simulation, and result analysis. ScSF is scalable, it is deployed over distributed resources to run and manage multiple concurrent simulations and provides tools to synthesize results over large experiment sets. The framework produces representative results by relying on Slurm, which captures the behavior of the real system schedulers. ScSF is also modular and might be extended by researchers to generate customized workloads or calculate new analysis metrics over the results. Finally, we improved the Slurm simulator which now achieves up to 15x simulate over real time speed-ups while preserving its determinism and experiment repeatability,

This work provides a foundation for future scheduling research. ScSF will be liberated as open source, enabling scheduling scientists to concentrate their effort on designing scheduling techniques and evaluating them in the framework. Also, we share the experience of using ScSF to design our own scheduling algorithm and evaluating it through the simulation of a large experiment set. This experience shows that the framework is capable of simulating 33 years of real systems time in less than two months over a small distributed infrastructure. Also, it constitutes a guide for future users of the framework.

Finally, our experiences building ScSF and running it in scale might be of use of researchers who are building similar systems.

7 Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR). The National Energy Research Scientific Computing Center, a DOE Office of Science User Facility, is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE and the Swedish Research Council (VR) under contract number C0590801 (Cloud Control). Special thanks to Stephen Trofinoff and Massimo Benini from the Swiss National Supercomputing Centre, who shared with us the code base of their Slurm Simulator. Also, we would like to thank the members of the DST department at LBNL and the distributed systems group at Umeå University who administrated or gave-up the compute nodes supporting the use case.

References

1. Dirty cow (January 2017), `https://dirtycow.ninja/`

2. SchedMD (January 2017), https://www.schedmd.com/
3. shuttle (January 2017), https://github.com/apenwarr/sshuttle
4. Workflowgenerator (1 2017), https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
5. Alvarez, G.P.R., Elmroth, E., Östberg, P.O., Ramakrishnan, L.: Enabling workflow aware scheduling on hpc systems. In: Under review for HDPC 2017
6. Alvarez, G.P.R., Östberg, P.O., Elmroth, E., Antypas, K., Gerber, R., Ramakrishnan, L.: Towards understanding job heterogeneity in hpc: A nersc case study. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 521–526. IEEE (2016)
7. Declerck, T.M., Sakrejda, I.: External Torque/Moab on an XC30 and Fairshare. Tech. rep., NERSC, Lawrence Berkeley National Lab (2013)
8. Feitelson, D.: Parallel workloads archive 71(86), 337–360 (2007), http://www.cs.huji.ac.il/labs/parallel/workload
9. Feitelson, D.G.: Workload modeling for computer systems performance evaluation. Cambridge University Press (2015)
10. Feitelson, D.G., Tsafrir, D.: Workload sanitation for performance evaluation. In: 2006 IEEE international symposium on Performance analysis of systems and software. pp. 221–230. IEEE (2006)
11. IBM: Platform computing - lsf. http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/sessionscheduler.html (January 2014)
12. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Job Scheduling Strategies for Parallel Processing. pp. 87–102. Springer (2001)
13. Kannan, S., Mayes, P., Roberts, M., Brelsford, D., Skovira, J.: Workload Management with LoadLeveler. IBM Corp. (2001)
14. Klusáček, D., Rudová, H.: Alea 2 – job scheduling simulator. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010). ICST (2010)
15. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63(11), 1105–1122 (2003)
16. Lucero, A.: Simulation of batch scheduling using real production-ready software tools. Proceedings of the 5th IBERGRID (2011)
17. Rodrigo, G., Östberg, P.O., Elmroth, E., Antypass, K., Gerber, R., Ramakrishnan, L.: HPC system lifetime story: Workload characterization and evolutionary analyses on NERSC systems. In: The 24th International ACM Symposium on High-Performance Distributed Computing (HPDC) (2015)
18. Schwiegelshohn, U.: How to design a job scheduling algorithm. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 147–167. Springer (2014)
19. Stephen Trofinoff, M.B.: Using and Modifying the BSC Slurm Workload Simulator. In: Slurm User Group (2015)
20. Yoo, A., Jette, M., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2862, pp. 44–60. Springer Berlin / Heidelberg (2003)
21. Zakay, N., Feitelson, D.G.: Preserving user behavior characteristics in trace-based simulation of parallel job scheduling. In: IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014. pp. 51–60. IEEE (2014)

Paper V

Enabling Workflow Aware Scheduling on HPC Systems

Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan

Submitted *2017*

Enabling Workflow Aware Scheduling on HPC systems

Gonzalo P. Rodrigo*, Erik Elmroth, Per-Olov Östberg, Lavanya Ramakrishnan+
Dept. Computing Science, Umeå University, SE-901 87, Umeå, Sweden
Lawrence Berkeley National Lab, 94720, Berkeley, California+
{gonzalo,elmroth,p-o}@cs.umu.se
lramakrishnan@lbl.gov

ABSTRACT

Workflows from diverse scientific domains are increasingly present in the workloads of current HPC systems. However, HPC scheduling systems do not incorporate workflow specific mechanisms beyond the capacity to declare dependencies between jobs. Thus, when users run workflows as sets of batch jobs with completion dependencies, the workflows experience long turn around times. Alternatively, when they are submitted as single jobs, allocating the maximum requirementof resources for the whole runtime, they resources, reducing the HPC system utilization.

In this paper, we present a workflow aware scheduling (WoAS) system that enables pre-existing scheduling algorithms to take advantage of the fine grained workflow resource requirements and structure, without any modification to the original algorithms. The current implementation of WoAS is integrated in Slurm, a widely used HPC batch scheduler. We evaluate the system in simulation using real and synthetic workflows and a synthetic baseline workload that captures the job patterns observed over three years of the real workload data of Edison, a large supercomputer hosted at the National Energy Research Scientific Computing Center. Finally, our results show that WoAS effectively reduces workflow turnaround time and improves system utilization without a significant impact on the slowdown of traditional jobs.

1 INTRODUCTION

In recent years, we have seen an increase in the processing of large amounts of scientific data and high-throughput processing at HPC centers. These scientific workloads are changing the landscape of software ecosystems on HPC centers that have traditionally supported large communication-intensive MPI jobs. The scientific workloads are increasingly composed as scientific workflows with complex dependencies.

The HPC batch schedulers still operate with a job-centric view and do not account for the complexities and dependencies of scientific workflows. Scientific workflow tools present in HPC centers often run workflows as chained jobs (jobs with dependencies) or as a pilot job (a single job containing the entire workflow). These approaches both have their drawbacks. Workflows run as chained jobs have very long and unpredictable turnaround times as they include the intermediate wait times for each job in the critical path. Workflows run as pilot jobs are likely to have shorter turnaround time, since the intermediate tasks do not wait for resources. However, they allocate the maximum required resources over the length of the workflow for its entire runtime. Thus, resources are wasted as they are allocated but idle in parts of the workflow.

Scientific workflows also provide a unique opportunity for future HPC scheduling and resource management systems. Scientific workflows include detailed knowledge of the complex pipeline and dependencies between the tasks that can be used to gain efficiency in the system. In this work, we present the design and implementation for extending existing batch schedulers with workflow awareness. Our workflow aware scheduling system (WoAS) takes advantage of dependencies to achieve short turnaround times while not wasting resources. WoAS enables pre-existing scheduling algorithms to be aware of the fine grained workflow resource requirements and structure, without any modification to the original scheduling algorithms. WoAS uses the knowledge of the workflow graph to schedule individual tasks while minimizing the wait times for the jobs.

We implement WoAS within Slurm, a common HPC workload manager. Execution of diverse workflow workloads was simulated over a model of a real system (NERSC's Edison) [18], [1]. In our evaluation, we compare its performance with the chained and pilot job approaches. The experiment set is composed of 271 scenarios, covering different workflow types and submission patterns. Simulated time accounts for 253,484 hours (29 years) of system time and 3.8 million compute core-years.

Our experiments show that in most workloads run with WoAS, workflows show significantly shorter turnaround times than the chained job and single job approaches without wasting resources. The impact on non-workflow jobs was minimal except for workloads heavily dominated by very large workflows where performance limitations of the backfilling (queue depth limit) interfered with their scheduling.

Specifically, in this paper, our contributions are:

- We design a workflow aware scheduling system and algorithms that produce turnaround times with almost no intermediate wait times and wastage of resources.
- We present the WoAS implementation and its integration with Slurm. The WoAS system will soon to be made available open source.
- We evaluate and present the results of a detailed comparison of the workflow performance and system impact of WoAS, the pilot job, and the chained job approaches for diverse workflow workloads simulated over the model of Edison, a supercomputing system at the National Energy Research Scientific Computing Center (NERSC).

The rest of the paper is organized as follows. In Section 2, we present the life cycle of workflows and current scheduling approaches. The

* Work performed while working at the Lawrence Berkeley National Lab.

HPDC '17, Washington D.C., USA
2016. 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

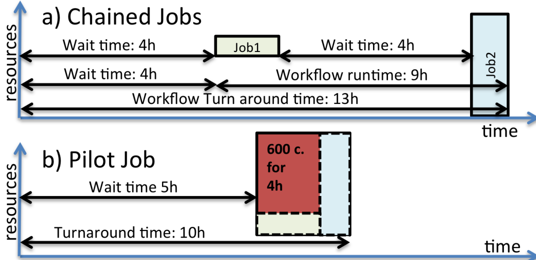


Figure 1: Cybershake workflow executed using chained and pilot job approaches. The chained jobs approach increases execution time due to the wait times. In the single job, there are no intermediate wait times but it wastes 600 cores for 4h.

Workflow Aware Scheduling technique (WoAS) is discussed in Section 3. We present the methodology to compare WoAS to the existing workflow scheduling methods in Section 4 and experimental results in Section 5. We discuss our conclusions in Section 6.

2 BACKGROUND

In this section, we describe the state-of-art and challenges of managing scientific workflows on HPC systems and discuss related work.

2.1 A workflow's life-cycle

Workflows are represented as Directed Acyclic Graphs (DAG) i.e., each vertex represents one or more work tasks, and the edges express control or data dependencies between the (vertices) tasks.

The first step to run a workflow on an HPC system is to map the DAG into one or more batch jobs, while respecting the data and control dependencies expressed by the edges. Users manually do the mapping or rely on workflow managers [26], which might also automate the submission and control of the workflow job(s). There are different mapping techniques governed by targets such as minimizing cost [27], minimizing runtime and turnaround time [4, 25], or tolerating faulty, distributed resources [17].

Once the execution plan is defined as a list of jobs and dependencies, users usually follow one of two strategies to submit a workflow. The strategies balance between lower resource consumption (as *chained jobs*) or shorter turn around time (as a *pilot job*). Figure 1 illustrates these approaches the for Cybershake workflow ([3]), which is used by to simulate geological structures to characterize earthquake hazards in a region.

2.1.1 Workflow as chained jobs. In this approach, one batch job is submitted per execution plan job. Current batch schedulers allow users to specify dependencies between batch jobs. The scheduler then forces jobs to wait to start until the completion of its dependencies. Alternatively, users or their workflow engines might submit a job when the the ones it depends upon have completed. Each job receives the exact amount of resources required to run and no allocated hardware resources are intentionally left idle. However, the workflow runtime will include the wait times endured by each of the jobs in the workflow's critical path. As described in Section 2.2, job priority systems do not consider a job until its dependencies are resolved. As a consequence, job wait times for explicit dependencies are equal to the ones observed by submitting jobs when their dependencies are solved.

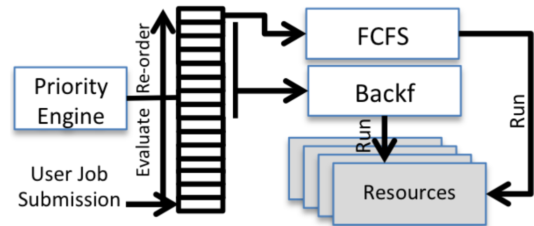


Figure 2: Classical batch scheduler with the waiting job queue in its center, which jobs are ranked by the priority engine and scheduled by FCFS and backfilling.

Figure 1a shows Cybershake's execution plan submitted as chained jobs. Even if both jobs are submitted at the same time, $Job2$ has to wait four hours from the point $Job1$ is completed, as its priority (and thus position in the waiting queue) does not increase from its initial value until its dependencies are resolved. The result is that the workflow runtime (i.e., start of first job to completion) is nine hours, wait time is four hours (44%), while the turnaround time (i.e., from job submission to completion) is 13 hours, eight hours accounts for wait time (61%).

2.1.2 Workflow as a pilot job. In this approach, the execution plan is submitted as a single job. The job's time limit is set to the expected runtime of the critical path with no intermediate job wait time. The job's resource request is the maximum resource allocation needed at any point in the workflow. As a consequence, the runtime of the workflow is the minimum possible, but some allocated resources might be idle, and thus wasted.

Figure 1b presents Cybershake's execution plan submitted as a pilot job. The workflow wait time is larger than the one faced by the chained job approach. However, the runtime is the minimum possible as there is no intermediate wait time. In this case, the wait time for the pilot job to start is smaller than the wait time for the two jobs in the chained job approach. However, during the first four hours of the workflow 600 CPU cores are allocated to the workflow but left unused, totaling 2400 idle core hours. This is the caveat of this approach, turnover is better but for a higher cost of consumed resources over the same work. This approach would work well for workflows where the difference between the minimal and maximum width of the workflow is not significant.

2.2 Classic HPC scheduling systems

Figure 2 presents the schema of a classic HPC scheduler with a queue of waiting jobs in its center: a) Jobs are inserted in a queue when users submit them. b) Scheduling algorithms select which jobs should start running and extract them from the queue. A classical HPC batch scheduler incorporates at least the following scheduling algorithms: FCFS (First-Come, First-Served) [6], running the first job of the queue if enough resources are available; and backfilling [11], scanning jobs in the queue in order to run them if enough resources are available and if they would not delay the start time of

previous jobs in the queue. Combined, they achieve high utilization and a reasonable job turnaround times in HPC systems.

Schedulers also include internal prioritization engines to manage turnaround time by ranking waiting jobs following some administrator set policy. Backfilling algorithms try to schedule sooner those jobs with a higher rank (priority).

Batch schedulers also understand job dependencies (relationships between jobs), and, among them, the most common one, enforces that a job cannot start until *n* previous jobs end successfully. Dependencies can be used to to express the structure of the jobs within a workflow. However, we observed that in existing HPC schedulers dependencies affect the job priority calculation. For example, job age priority engines consider that a job age starts when its dependencies are resolved. In such schema, the submission time of the job will be its dependency resolution time, no matter when it was submitted. In Figure 1a we show the impact of this policy on workflow's turnaround time.

2.3 Related work

Complex experiments in scientific fields like high-energy physics, geophysics, climate study, or bioinformatics, require distributed resources as computational devices, data-sets, applications, and scientific instruments. The orchestration of such processes is organized as scientific workflows: collections of tasks structured by their control and data dependencies [26]. Distributed scientific workflows have been explored in detail in the last few years. Due to location specific or large resource requirements, a large portion of the workflows are distributed [16], i.e. their tasks run and data stored in different compute centers. In such environment, their execution depends on user inputs, specific resource characteristics, and run-time resource availability variations [9].

In other cases, workflows are run within the same compute facilities (single site workflows), however their tasks might be too large, or require too different resource sets that force to run them as different entities. This work focuses on the scheduling of the jobs of single site workflows.

Scheduling, automation, and execution systems for scientific workflows has been largely studied. Pegasus [4], Askalon [5], Koala [14], and VGrADS [17] are examples of Grid workflow managers that includes different approaches to workflow mapping, meta-scheduling, execution, task management, monitoring, and fault tolerance. However, they do not propose specific solutions to schedule the jobs of a workflow inside each of the Grid site, which is responsibility of the site scheduler.

There is also work on specific Grid workflow scheduling algorithms like: *Myopic* [25], *Min-Min* [13], *Max-Min* [13], *Sufferage* [13], Heterogeneous-Earliest-Finish-Time (*HEFT*) algorithm [24], or *Hybrid* [20]. These algorithms schedule jobs within, between, or across workflows under different strategies and objective functions. However they rarely schedule regular jobs and workflow jobs together, which is the main focus of this work.

Scientific workflow management systems for high throughput application have become more popular in the last years. Fireworks [8], QueueDO (QDO) [2], Falkon [15], and Swift [29] offer tools for workflow composition and management, execution, job packing of tasks (serial, OpenMP, MPI, and hybrid), and monitoring. These systems may deploy their own execution frameworks or run their task in workers packed inside HPC jobs which, in the end are submitted as a pilot job or chained jobs.

Finally, data intensive and streaming workflows have become very important for the data processing within large IT companies. Frameworks like Hadoop [22], Spark [28], or Heron [10] offer workflow composition, management, and automation.

Clusters with batch jobs and services present the challenge of scheduling different workloads which metrics that cannot be compared. In that context, multilevel scheduling approaches have appeared allowing independent schedulers for different workloads (Mesos [7]), smart resource managers (Omega [21]), or cloud inspired two level scheduling for HPC systems (A2L2 [19]).

3 WORKFLOW AWARE SCHEDULING

Workflow Aware Scheduler (WoAS) provides an interface that allows users to submit *workflow jobs*. As illustrated in Figure 3, a user submits a *workflow job* that is a batch job that includes a manifest describing its internal workflow structure (e.g., two task jobs in the example). The workflow job is stored in the system's waiting queue.

There are three separate threads that work on the waiting queue - WoAS, the scheduler, and the priority engine. WoAS is always activated between the scheduler and priority. Thus the order of execution would be [WoAS, Scheduler, WoAS, Priority Engine]. When WoAS acts before the scheduler, it substitutes each workflow aware job by the task jobs described in its manifests, configuring the corresponding dependencies, and placing them in the same position of the queue as the original job. The resulting version of the queue is the *scheduler view* of the queue.

Once the queue is transformed, the scheduling algorithms act on it. In our example, backfilling selects and starts the first task job of of the example workflow, allocating exactly the resources that it requires. After the scheduling phase is over, WoAS transforms the waiting queue, removing the task jobs of workflows that have not started and restoring the corresponding workflow-aware jobs. The current state of the queue is the *priority view* of the queue. The priority engine periodically processes the waiting queue, calculating the priority of each job and ordering jobs accordingly.

The system continues repeating the cycle of a) recalculating the jobs priority b) transforming the queue into its scheduler view c) doing a scheduling pass (scheduling the second job of the example workflow when the first had completed) d) restoring the queue to its priority view.

In this section, we describe the steps of this process in detail.

3.1 Workflow job submission

In WoAS, users submit a workflow as a *workflow aware job*. This is an extension of the way workflows are represented in the pilot job approach. Users submit a job allocating the maximum resources required in the workflow for the minimum duration of its critical path, similar to a pilot job. However, a manifest describing the workflow is attached to the batch script.

Figure 4 is an example workflow description in JSON format for the LongWide workflow (defined in Table 2). It contains the definition of all the tasks within the workflow, including their

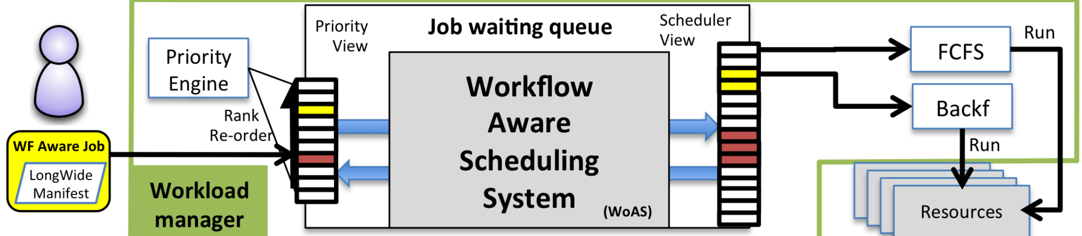


Figure 3: A workflow in WoAS scheduling model from its submission to execution start.

```
{"tasks": [
  {"id": "SLong", "cmd": "./SLong.py",
   "cores": 48, "runtime": 14400.0},
  {"id": "SWide", "cmd": "./SWide.py",
   "cores": 480, "runtime": 3600.0,
   "deps": ["SLong"]}]}
```

Figure 4: LongWide workflow manifest in JSON format.

resource allocation requirement (allocated CPU cores for an estimated runtime), command or application to be executed (cmd), and dependencies with other tasks (deps, where SWide depends on completion of SLong). This manifest information is used by WoAS to transform a workflow aware job (priority view) into its task jobs (scheduler view).

3.2 Workflow Aware Scheduling system

The Workflow Aware Scheduling system (WoAS) is a job waiting queue model to bring workflow awareness to an HPC scheduler by offering different job lists (views) depending on the scheduler element that is interacting with the queue. The dual-view enables WoAS to enforce general scheduling behaviors such as the ones in Section 3.3, without requiring to change the code of the scheduler elements interacting with the queue.

WoAS controls the access to the waiting queue, and depending on the scheduler component interacting it presents two views:

The priority view. In this view, each workflow aware job is presented as a single job (the one submitted by the user). This view is the one presented to the priority engine. As a consequence, the priority and queue position of each workflow is based on the workflow aware job characteristics (submission time, geometry). All tasks in a workflow have the same priority or start with the same priority?

The scheduler view. In this view, each workflow aware job is present in the waiting queue through instances of its internal task jobs (and corresponding dependencies) placed in the same position of the queue where the original workflow aware job was. This view is the one presented to the scheduler algorithms, so they can schedule the workflow task jobs individually.

3.3 Workflow awareness in WoAS

In this section, we discuss the impact of the views model on the workflow awareness in the scheduler. Specifically, there are three behaviors. First, workflow task job level scheduling results in the allocated resources are the minimum possible. Second, the intermediate wait times are minimized to avoid the ones observed in the chained job approach. Finally, this minimizes system gaming where users don't have to ask for strange resource requests to make sure the tasks in their workflows get the correct priority.

Workflow awareness is consequence of the interaction of the scheduler with the views and the way job's priority information is transferred when the queue is transformed between views.

3.3.1 Workflow task level job scheduling. In WoAS, the scheduling algorithms schedule workflow task jobs, assigning the precise required resource allocation to each step of the workflow.

This is possible because the scheduling algorithms act on the scheduler view provided by WoAS. In opposition to the pilot job approach, even if workflows are submitted as a single job, WoAS ensures that the scheduling algorithms will see the workflows as their task jobs.

This characteristic is what allows WoAS not to waste resources to run workflows, even if they are submitted as single job.

3.3.2 Minimization of the intermediate wait times. With WoAS, when the first task job of a workflow is started during the scheduling view, the rest of the workflow task jobs remain related by their dependencies in the waiting queue. This situation is similar to how task jobs stay in the queue for the chained job approach.

However, in the chained job approach, intermediate wait times might be very long. Classical schedulers consider jobs with non resolved dependencies as not "submitted", so their priority does not increase as they wait. From the moment that jobs they depend on are completed, task jobs have to wait as they had been submitted, even it had been waiting in the waiting job queue for much longer time.

WoAS reduces intermediate wait times by propagating the priority attributes of the original workflow aware job to all its tasks. All the tasks have the same geometry priority factor and submit time as the original workflow aware job. As a consequence, under the scheduling view, all the tasks in a workflow are positioned in adjacent positions in the waiting queue. If the first task job starts, its queue position should be close to the top. As all the workflow task jobs have similar queue positions, once the first task job is completed, the following one will still be in a good queue position to be started. In such situations, the intermediate wait time should be close to the time until adequate resources for that task job are available. This time can be significantly shorter than the time that it would take for that task job to progress from the bottom to the top of the priority queue in a highly utilized system.

Propagation of the priority information is performed by a combination of the views and operations within WoAS.

A workflow aware job's priority information is set during the priority calculation under the priority view. In our system, the priority of a job depends on two factors:

1) Job's geometry factor, (smaller job, higher priority). It is calculated only once in the life of a job, in the first priority calculation process that considers it.

2) Job's age factor, (older job, higher priority) is recalculated in every priority calculation process. It depends on the time the job was submitted.

Algorithm 1 Show scheduler view actions.

```
1  def woas_show_scheduler_view():
2    global waiting_queue
3    for job in list(waiting_queue):
4      if is_workflow_aware_job(job):
5        remove_job(waiting_queue, job)
6        for task_desc in job.manifest["tasks"]:
7          new_job = create_job(task_desc)
8          new_job.prio.geometry = job.prio.geometry
9          new_job.prio.age = job.prio.age
10         new_job.submit_time = job.submit_time
11         new_job.copy_wf_job = job
12         insert_job(waiting_queue, new_job)
```

The priority information of the workflow aware job is propagated to its task jobs through the operation woas_show_scheduler_view, which transforms the waiting queue into scheduler view. The detailed actions of this operation can be followed in Algorithm 1, where each task job receives the workflow aware job geometry factor, age factor, and submit time.

This propagation has three consequences for future priority calculations of all the task jobs of the same workflow. First, Future task job age factor calculations will be based on a workflow's submit time. Second, the geometry factor of a job is set, so the priority engine does not recalculate it. Thus, future task job priority calculations will be based on the workflow aware job's geometry, not its own. Finally, all task jobs of the same workflow will have the same priority (and the same the workflow aware job would have) since it has the same geometry factor and same submit time.

This ensures that all task jobs of the workflow will have the same priority and will occupy a similar position in the waiting queue, which leads to the minimization the intermediate wait time,

As a final note, the priority propagation of non started workflows is closed by the woas_show_priority_view operation. As it transforms the queue in its priority view, it enforces that if a workflow has not started, it becomes again the same workflow aware job, with the same priority factors.

3.3.3 Minimize system gaming. The priority propagation mechanism described in Section 3.3.2 has another side effect. Since task job's priority factors are the same as the ones of the workflow aware job, the waiting time of the first task job is equivalent to the waiting time of a job of the geometry and submission time of the workflow aware job.

This is a desired effect to stop users from gaming the system, i.e. users submit workflows where first job is very small, expecting a short wait time to then run larger task jobs. This takes advantage of the short wait time minimization of WoAS: As the workflow wait time depends on the workflow aware geometry, such schema would only produce longer wait times.

3.4 Batch Scheduler integration

Algorithm 2 Simplified classical scheduler algorithm with WoAS calls to enable the views model.

```
1  def scheduling_loop_with_WoAS():
2    while True:
3      if time_to_check_priority():
4        do_priority_calculations()
5      if (time_to_do_fifo() or
6          time_to_do_backfilling()):
7        woas_show_scheduler_view()   // WoAS specific
8        if time_to_do_fifo():
9          do_fifo_scheduling()
10       if time_to_do_backfilling():
11         do_backfilling_scheduling()
12       woas_show_priority_view()    // WoAS specific
```

WoAS was incorporated to the scheduler by modifying the core batch scheduling loop. Algorithm 2 describes a simplified representation of such process, in which a phase where jobs priority is recalculated (line 4) alternates with another in which the scheduling algorithms act (lines 8-11). In such a model, introducing WoAS does not require changing the priority or schedulers behavior. It requires adding just two actions to the loop, as listed below.

1) adding woas_show_scheduler_view before the scheduling phase starts (line 7). This function transforms the waiting queue for the following code (scheduling phase) to see the waiting queue through the scheduler view.

2) adding woas_show_priority_view after the scheduling phase starts (line 12). This function restores the waiting queue into the priority view, so the priority actions (if executed), act over that view.

In Slurm, the priority and scheduling components run concurrently, and the queue exclusive access is enforced through a lock. WoAS is integrated by adding the woas_show_scheduler_view call just after the scheduling code acquires the queue lock. Next, it adds woas_show_priority_view just after the scheduling code frees the queue lock. This ensures a behavior equivalent to Algorithm 2.

4 METHODOLOGY

We evaluate the workflow aware system using a Slurm simulator and analyze the resulting scheduling logs. In this section, we describe our simulator setup, metrics, and experiment definitions.

4.1 System

NERSC's Edison is the reference system chosen to be emulated and to model the baseline workload. Edison is a Cray XC30 supercomputer, with 6,384 nodes, 24 cores per node, and a total of 133,824 cores and 357 TB of RAM, installed in 2014. It uses an Aries interconnect and can produce a peak of 2.57 PFLOPS/s. Edison's hardware and workload are representative of systems and applications present in the high performance scientific community.

4.2 Simulation framework

We implemented WoAS in Slurm, since it is increasingly used in high performance systems. This functional implementation of a WoAS enabled Slurm will be distributed as open source. Also, previous work by the Barcelona Supercomputing Center (BSC) [12] and the Swiss Supercomputing Center (CSCS) [23] provided a Slurm simulator base code. We extended the simulator for our experiments. The simulator allows us to run experiments up to 20x faster than real time and run multiple simulations in parallel (up to 200).

The Slurm scheduler is configured similar to current HPC systems and uses FIFO, backfilling, and it gives higher priority to smaller jobs. However, to reduce complexity of the experiments and ease analysis, differentiated queues or QoS levels are not configured in our simulator. These features provide user-level conveniences and will translate to the workflow awareness and are not central to the focus of our experiments.

The core of our simulator is the Slurm scheduler. Slurm is configured to use the desired scheduling method (chained jobs, pilot job, or workflow aware). After configuration, the simulator starts Slurm and submits the workload to it, emulating the user behavior. The scheduling process is run for a configured simulation time (5 days plus an extra for cold start stabilization) and the scheduler logs are registered in a MySQL database for later analysis.

An experiment is defined by its workload characteristics, a scheduling method choice, a simulated system configuration, a target simulated time, and a random seed. To run an experiment, the workload is generated according to the workload characteristics. Workflow characteristics include the characteristics of the real HPC system workload after regular jobs are modeled, a list of specific workflows present in the workload, and their submission patterns.

Finally, each experiment is repeated using six different random seeds (producing different workloads) and their results aggregated to ensure that analyses are not based on single non representative experiments.

4.2.1 Workload generation. Each experiment has a workload composed of regular (non workflow) synthetic jobs and workflow jobs modeled after the experiment configuration. The regular jobs in our workload traces are modeled after the historical traces from three years of NERSC's Edison system, [18] and [1].

The experiment configuration defines the specific workflows present in the workload, the job format for the workflow (a pilot job, chained jobs, or a job including a workflow manifest), and the submission pattern. Our simulator supports two workflow submission patterns - workflow periodic and workflow share. In the periodic one, a workflow is submitted once every configured time period. In the share model, workflows are submitted at a uniform pace so the number of allocated core hours to workflows represents a desired share of the total core hours of the workload.

The workload generator also includes a mechanism to pre-fill the system to capture a typical state of a supercomputer system. The lengths of the jobs for pre-fill stage are configured to obtain a job wait time baseline of four hours. Also, a job pressure control mechanism adjusts the job and workflow submissions so the workload job pressure (submitted core hours over system capacity in a time period), is slightly over 1.0. This ensures that simulated system will have have enough pending work to support the wait time baseline, but with not to much to significantly increase the job wait time as the workload scheduling progresses. Also, the simulator uses a system cold-start stabilization period of one day. This workload is not representative of a regular day systems operation and is discarded for the analyses.

Finally, the workload generator uses a random number generator that can be initialized with a seed. The same seed always produces the exact same regular jobs and workflow submission times, independently of the workflow scheduling system chosen (as long as the same workload configuration is used). This is used to do a fair comparison between different scheduling techniques for the same experiment configuration.

The described workload analysis and modeling tools; the workload generator; the framework to define and run experiments; the tools to process and analyze experiment results; and the improvements on the Slurm simulator, were developed in the context of this work.

4.3 Evaluation metrics

In this section, we present the metrics used to compare experiment results and the method to calculate them.

4.3.1 Performance metrics. In our analyses, we use three workflow (wait time, run time and turnaround time) and two system (system utilization, job slowdown) performance metrics to compare the pilot job, chained job, and WoAS approaches.

Workflow wait time (w^W) is the time between the submission of the first job of the workflow and its execution start. Smaller wait times are preferred. It depends on the load in the system (waiting work vs. compute capacity, with higher loads implying overall longer waiting times), the geometry (smaller jobs tend to wait less due to backfill), and priority (higher tends to imply shorter wait time).

Workflow runtime (r^W) is the time between the execution start of the first job and the execution completion of the last job of the workflow. It includes the runtime of the jobs in the critical path of the plan and the wait time between them. Smaller runtimes indicate lesser waste between the tasks of the job. Minimum possible workflow runtime is the sum of the runtimes of the jobs in the critical path (as they run back to back).

Workflow turnaround time (t^W) is time between the submission of the first job and the execution completion of the last job of the plan. Smaller values are better. It is obtained as the sum of w^W and r^W, thus it depends on the factors the previous two depend on.

Actual utilization during a time period (t) is $\frac{\sum corehours_i^J - \sum waste_i^W}{cores^S * t}$, where $corehours_i^J$ are the core hours allocated by jobs and workflows that are executed, $waste_i^W$ is the number of core hours allocated by a workflow that are not assigned to an internal task or job where $cores^S$ are the number of cores of the compute system. This metric is a variation of the classical utilization that takes into account that workflows might allocate resources but not use them through all their runtime. It measures the actual work done over the system capacity, not just allocation. This is relevant to measure since pilot job workloads might show a

Group	A: Workflow critical path length.	B: Allocated cores, overall vs 1st job.	C: Alloc. cores and rtime, overall vs first job.
Geometry	n jobs/rtime:n h/max 240 core	2jobs/rtime:2n h/max 240n cores	2jobs/rtime:2n h/max 240n cores
Usage/Waste	240n core-h / 0 core-h.	$240 + (n) * 240$ core-h. / $(n-1) * 240$ core-h	$240 + n(2n-1) * 240$ core-h. / $n(n-1) * 240$ core-h.
Profile	240c. J_1 J_2 ooo J_n 1h	240c. J_1 J_2 nx240c. 1h 1h	240c. J_1 J_2 nx240c. 1h n-1x1h

Table 1: Workflows characteristics for workflow groups: critical path size, pilot job geometry (rtime and max cores), workflow tasks usage (usage), potential wasted resources (waste), and a profile of the allocated resources in time if the critical path is run with no intermediate waits.

high theoretical classical utilization and hide the fact that resources might be allocated but not used.

Job's slowdown is measured as $\frac{r^J + w^J}{r^J}$, i.e. job's turnaround divided by its runtime. This metric allows us to compare the wait time from jobs with different runtimes. We use this metric to measure the impact of different workflow scheduling techniques on the non workflow jobs. We calculate this metric for non workflow jobs grouped in three different sizes ($[0, 48)$, $[48, 960)$, $[960, \infty)$ core hours). The median values of this metric for each job group are used in comparisons.

4.3.2 Metrics calculation. All the metrics of this work are obtained over the aggregation of the results of multiple repetitions of the same experiment. To keep the meaning of each metric, the aggregation method is different.

For the workflow performance metrics, the performance values of m_i first workflows of each repetition i were aggregated and then the percentile metrics calculated. m_i of a repetition of an experiment is the minimum number of workflows completed in the three versions (WoAS, pilot job, and chained job) of that repetition i. This pre-selection is required to compare similar datasets and these metrics cannot be calculated for incomplete workflows.

Actual utilization for an experiment is calculated as the mean of the observed actual utilization in the six repetitions. This is equivalent to calculating the utilization of an experiment which was the concatenation of the six repetitions. For the aggregated calculation of the job's slowdown, all the non workflow job slowdown values in the repetitions are read, and the percentile analysis is performed on them.

4.4 Experiment sets

Two experiments sets are analyzed in this work, studying the scheduling techniques from a more analytical and real point of view, resulting in 271 experiment configurations. Each individual experiment consists of five days of simulated scheduling of the workload plus an extra initial one for the system cold start.

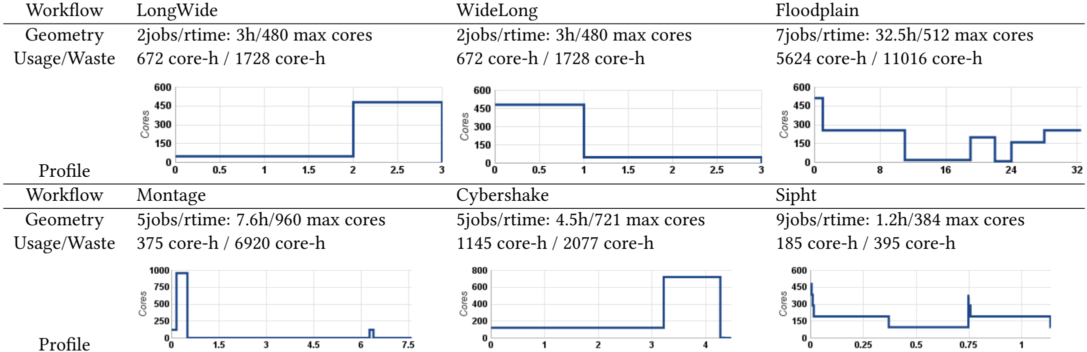
4.4.1 Workflow characteristics study. In this experiment set, we analyzed the effect on the workflow metrics of using different scheduling techniques to run workflows with different internal characteristics. There is a workflow group for each workflow characteristic, and inside each group, a workflow is defined by n: a knob that controls the effect of the workflow characteristic, where a larger n implies a larger effect.

These experiments allow creating a base knowledge on the expected wait time, runtime, and turnaround times for some basic workflow characteristics. These are the workflow groups as presented in Table 1:

Workflow critical path length, Group A: The goal with this group of workflows is to study the effect on the workflow metrics of the number of tasks in the workflow critical path (n defines the number of those tasks). All workflows in this group are chained lists of n tasks of the same size (240 CPU cores and 1h runtime), e.g. if $n = 3$ the resulting workflow has three tasks, in which the second depends on the first and the third of the second. Workflows with a longer critical path should suffer: a) larger difference between runtime of the pilot job and first job in chained job and WoAS approaches. b) more intermediate wait time periods between the tasks in the chained job and WoAS approaches (workflow runtime related). c) lower priority for the pilot job and workflow aware job vs the priority of the first job in the chained job approach (workflow wait time related).

Allocated CPU cores: First job vs. workflow's maximum, Group B: Group B is used to study the effect on the workflow metrics of the difference between allocated cores for the first job and the workflow maximum (n is the difference multiplier controlling the breadth of the workflow). All workflows in this group are composed of two jobs, the first allocates 240 cores for one hour and the second allocates $n * 240$ cores for one hour. When combined with the workflow scheduling approaches, a higher number of n will induce two workflow wait time related effects: a) larger difference between the allocated cores by the pilot job and the first job in the chained jobs and WoAS approaches. b) lower priority for the pilot job and workflow aware job vs the priority of the first job in the chained job approach (workflow wait time related). The difference in priority is induced by the difference in resource allocation in opposition to the workflow runtime (Group A).

Allocated CPU cores and runtime: First job vs. workflow's maximum, Group C: This group is used to study the effect of the difference in allocated cores of the first job and the workflow maximum combined with the difference in runtime between the first job and the minimum critical path runtime on the workflow metrics (n is the difference multiplier controlling the length and breadth of the workflow). All workflows in this group are composed by two jobs: The first allocates 240 cores for one hour. The second allocates $n * 240$ cores for $2n - 1$ hours. When combined with workflow scheduling techniques, a higher number of n will induce two workflow wait time related effects: a) larger difference between



Workflow	LongWide	WideLong	Floodplain
Geometry	2jobs/rtime: 3h/480 max cores	2jobs/rtime: 3h/480 max cores	7jobs/rtime: 32.5h/512 max cores
Usage/Waste	672 core-h / 1728 core-h	672 core-h / 1728 core-h	5624 core-h / 11016 core-h
Profile			
Workflow	**Montage**	**Cybershake**	**Sipht**
Geometry	5jobs/rtime: 7.6h/960 max cores	5jobs/rtime: 4.5h/721 max cores	9jobs/rtime: 1.2h/384 max cores
Usage/Waste	375 core-h / 6920 core-h	1145 core-h / 2077 core-h	185 core-h / 395 core-h
Profile			

Table 2: Workflows characteristics for individual workflows including: critical path size, pilot job geometry (rtime and max cores), workflow tasks usage (usage), potential wasted resources (waste), and a profile of the allocated resources in time if the critical path is run with no intermediate waits.

the allocated cores and runtime for the pilot job and the first job in the chained jobs and WoAS approaches. b) lower priority for the pilot job and workflow aware job vs the priority of the first job in the chained job approach (workflow wait time related). The difference in priority is induced by the difference in resource allocation and workflow runtime.

In this experiment set, six workflows of each workflow group are defined ($n \in \{1, 2, 4, 8, 16, 32\}$, group C was not analyzed for $n > 8$, resulting workflows were too big and would overflow the system). For each individual workflow (16 in total), we create a workload in which a workflow is submitted with a fixed inter-workflow time. Each experiment is run using the pilot job, chained jobs, and WoAS techniques to compare the resulting metrics across techniques and values of n.

4.4.2 Performance comparison. In these experiment set we compared the performance of the different workflow scheduling techniques for two synthetic and four real workflows, which are presented in Table 2. The synthetic one (LongWide and WideLong) are the minimum building units of any workflow (a serial phase followed by a parallel one and vice-versa). The real ones allow testing our technique against more realistic workloads with has a particular characteristic: fixed jobs with a complex profile (Floodplain), many small grouped tasks (Montage), large workflow with two large parallel stages (Cybershake), and a small workflow with a complex profile shape and many small jobs (Sipht).

In the experiments, workflows are submitted using the workflow share approach with seven percentages: %1, %5 %10, %25, %50, %75, %100. The experiments with lower ones (%1 to %25) allow understanding the performance of the techniques of realistic scenarios with increasing workflow importance. The larger values (%50, %75, %100) allow understanding what happens in a system when the workload is dominated by workflows over regular jobs. The resulting 42 experiments are run using the pilot job, chained jobs, and WoAS techniques.

Similar experiments were run using the workflow period submission (periods 1/12h, 1/2h, 1/h, 2/h, 6/h). These allow comparing the workflow metrics in cases in which the workflow presence is not important enough to influence in the whole system behavior.

5 RESULTS AND ANALYSIS

This section presents the results and analyses of our simulation experiments. Our evaluation focuses on: a) A study of the impact of workflow characteristics on the workflow metrics obtained with different workflow scheduling techniques (Section 5.1). b) A performance comparison for the different scheduling techniques (Section 5.2).

5.1 Workflow characteristics study

Figures 5, 6, and 7 present the observed median workflow wait times, runtime, and turnaround time for the experiments with workflows from Groups A, B, and C (described in Section 4.4.1). Each horizontal block corresponds to a different workflow group. Inside each block, adjacent bars represent the measured median value for the same experiment configuration but run with different scheduling approaches (pilot job, chained jobs, and WoAS). The x-axis corresponds to n, a value that defines the actual workflow used in each workflow group (Defined in Section 4.4.1). In each group a higher value of n indicates that the special characteristic of the workflow group is more present.

5.1.1 Workflow wait time. For **group A workflows** (top block of Figure 5), we observe that the relationship between the median wait times observed for the pilot job and WoAS approaches are similar, with slightly shorter wait times for WoAS at all the workflow path sizes (n). Any difference in workflow wait time are related to differences in backfilling eligibility since priority and CPU cores allocation of the pilot job and the first job are the same.

In contrast, the chained job workflow has the same FIFO and backfilling eligibility as WoAS (both first jobs have the same geometry), but higher priority (job size used for priority is bigger in WoAS). Hence, workflows run as chained jobs show much shorter wait time (almost half at $n = 32$) as the critical path and workflow aware job sizes increases and the priority gap increases.

Figure 5: Wait time evolution as a dimension (one per horizontal group) of the workflow group is increased.

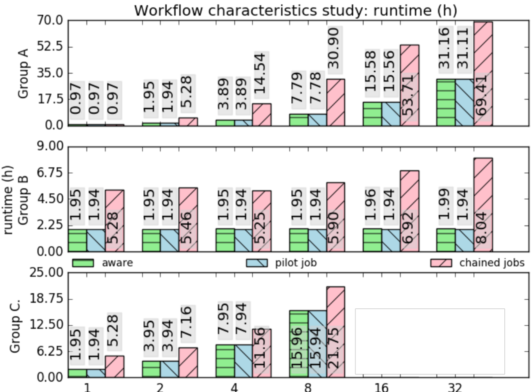


Figure 6: Runtime evolution as a dimension (one per horizontal group) of the workflow group is increased.

Similarly, workflows in groups B and C (second and third block of Figure 5) exhibit the shortest wait times when run as chained jobs, intermediate as WoAS, and much longer (specially for $n \geq 16$), as pilot jobs. The differences are due to the priority and backfilling eligibility of the workflow starting job in each group: higher priority, smaller job (more eligible) in chained job; lower priority, smaller job in WoAS; and lower priority, larger job (less eligible) in pilot job.

5.1.2 Workflow runtime. In Figure 6, we observe that all workflows run as pilot jobs or under WoAS present a very similar median workflow runtime, close to the expected minimum runtime for each value of n. This is expected for the pilot job, since all the tasks are run within a job with no internal wait times. We see that WoAS is able to perform as well; inter-job wait times between jobs when using WoAS is close to 0, e.g workflows in group A, $n = 32$, are

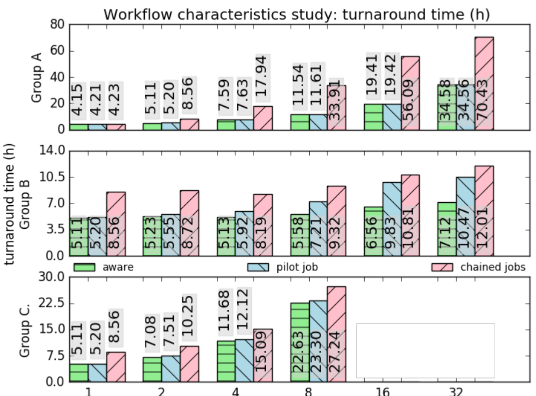


Figure 7: Turnaround time evolution as a dimension (one per horizontal group) of the workflow group is increased.

composed of 32 jobs (see Table 1) and the median of the accumulated 31 intermediate wait times accounts only for three minutes which constitutes 0.1% of the total runtime.

When run as chained jobs, group A workflows show longer accumulated inter-job wait time as the number of jobs in the workflow critical path increases. This matches the observation that most schedulers do not consider a job as truly submitted until its dependencies are resolved. Each extra job in the critical path adds an extra wait time to the runtime,.

Similarly, runtime of workflows in groups B and C are the minimum possible when run as pilot jobs, and close to minimum when using WoAS. When run as chained jobs, the increasing runtimes show the effect of the wait time of the second job on the runtime: As n increases, the geometry of the second job grows (slower in B than in C) and its wait time becomes longer.

5.1.3 Turnaround time. In Figure 7, we observe that for all workflow groups the chained jobs approach presents the longest turnaround times; followed by the pilot job and WoAS approaches, which shows the shortest (or equal to pilot job).

Workflows run as a workflow aware job present bigger gains in turnaround time over the pilot job approach as they face significantly shorter wait times (in our experiments group B and $n \geq 4$).

5.1.4 Summary. We observed that running a workflow as chained jobs results in the shortest wait time but longest runtime. Running it as a single pilot job, produces the shortest runtime but longest wait time. WoAS produces intermediate wait times and close to shortest runtimes.

Also, results show that WoAS produces the best turnaround times in all the scenarios. Finally, a workflow aware job can be considered significantly better performing than the pilot job approach even if the turnaround time is similar, since the former does not waste resources for workflows with varied resource requirements (more in the next section).

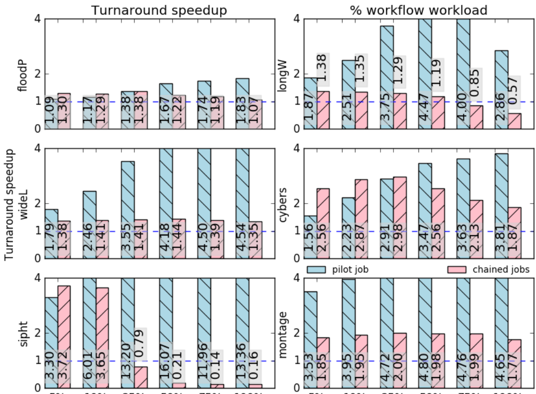


Figure 8: Workflows turnaround time speed up when scheduled as WoAS over pilot (blue) and chained (pink). Six workflows, six different workflow shares. The dashed line is speedup=1, Value > 1 implies better performance of WoAS.

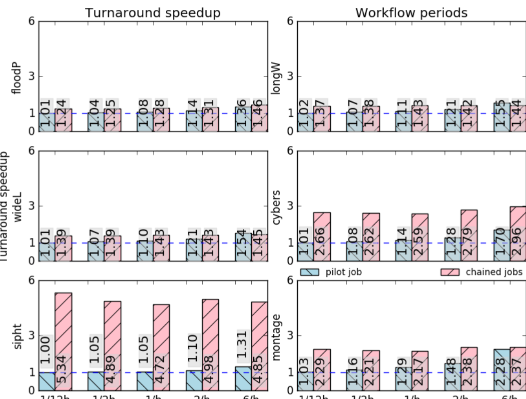


Figure 9: Workflows turnaround time speed up when scheduled with WoAS over the pilot (blue) and chained job (pink). Six workflows, six different workflow periods.

5.2 Performance comparison

In this section, we extend analyses in Section 5.1 to four real and two synthetic workflows. In this section we focus on workflow turnaround time, and the impact of the scheduling techniques over the system (actual utilization) and non-workflow job (slowdown) performance.

In this experiment set workflows are submitted using workflow share (described in Section 4.2.1) to set the percentage of workload core hours corresponding to workflows.

5.2.1 Workflow performance. Figure 8 presents the median workflow turnaround time speedup of WoAS relative to the chained jobs and pilot job approaches. The axis shows percentage of workload core hours contributed by workflows. A bar value $X = 1$ means that the median turnaround time for WoAS and the corresponding method are the same. A bar value $X > 1$, means that the median turnaround time median for the corresponding approach is X times the one observed with WoAS.

Compared to the chained job approach, WoAS showed very large speed ups for long complex workflows like Cybershake (≈2x) and Sipht (≈3s). For the rest of the workflows (shorter critical path), showed smaller but clear speedups in most cases (e.g ≈1.4x for WideLong, ≈1.9x in Montage). Floodplain has relatively small jobs reducing the effect of the intermediate wait times (1.2x-1.3x speedup).

Also, LongWide workflows showed shorter turnaround times when run as chained jobs in the 75% and 100% scenarios (< 1.0 speed ups). After analyzing system's actual utilization and overall wait time, it was observed that, when using WoAS in the scenarios, the wait time baseline is more elevated, and utilization is lower (20% and 30% less). This is likely related to the workflow shape. The workflow consists of a long job (48 cores, 4 hours runtime) and a wide job (480 cores running, 1 hour) where the long job can become a barrier that cannot start until a number of previous jobs end. The long job also stops other jobs from being backfilled since they would delay its start. This wait creates unused free resources gap that results in low utilization.

For workflow shares over 25%, Sipht experiments show turnaround speed-ups under one and get smaller as the workflow share increases. In these scenarios, the chained jobs approach achieved lower utilization values (≈10-20% less) and less workflows would complete than WoAS. Sipht is the workflow with more jobs, but very small resource allocation. In a workflow saturated scenario the scheduler manages a large number of active jobs, affecting its performance, and thus capacity to utilize the system. Since WoAS represents all workflows that have not started yet as a single job, the scheduler requirements to deal with such workload should be smaller.

Both situations are very unlikely in a real system, where the workflow includes different types of workflows and regular jobs, that can be used in efficient backfilling.

Compared to the pilot job approach, In Figure 8, WoAS shows turnaround times orders of magnitude shorter than the pilot job.

The long pilot job turnaround times are due to their long wait times. This is an artifact of the workload generation that is designed to contain the same amount of work i.e., workloads contain the exact same jobs and workflows, submitted at the same time. When run with the chained job technique, that work is meant to produce a job pressure of ≈1.0 over the system. However, when run with pilot jobs, the same work allocates more core hours because of the potentially wasted resources, increasing the job pressure. For example, in a 100% workflow share workload, the job pressure when workflows are pilot jobs is $1.0 + k$, being k the percentage of wasted core hours in the selected workflow).

In summary, for the same amount of work, the single job approach presents much longer turnaround times and loads the system significantly more than the other approaches (more in Section 5.2.3).

Finally, we asses if the previously observed short turnaround times for isolated workflows run as pilot jobs are possible for the workflows in this section. A set of experiments were performed, reducing the workflow presence in the workload by submitting

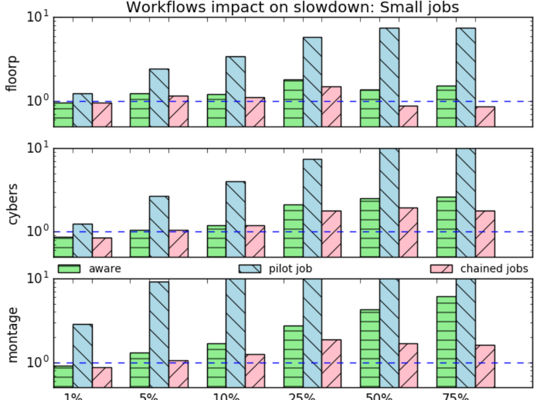


Figure 10: Relative difference on slowdown ($withWorkflowSlow/noWorkflowSlow$) for jobs allocating $[0, 48)$ core hours. Tested for three real workflows and different workflow shares.

one workflow in every time period. In Figure 9, WoAS presents similar or shorter turnaround times than the pilot job approach, confirming that for isolated workflows, the pilot job approach also shows short turnaround times.

5.2.2 Job fairness. Figure 10 presents the observed median slowdown for small jobs (under 96 core hours) over different workflow shares (x-axis). A value of one means that the median slowdown for non workflow jobs is the same as in the case where no workflows are present. A number of two represents that the observed median slowdown is two times the one observed when no workflows are present. It is important to note that adding workflows changes the workload composition and small variations in the slowdown are considered normal.

Figure 10 shows that the presence of workflows affects the regular job's slowdown. All experiments using the pilot job approach showed the biggest increases in slowdown (up to 10x), followed by WoAS (up to 8x), and the chained job ones (up to 2x). This difference is specially significant for Montage workflows run as pilot jobs, where just a 1% workflow share induces a three times bigger median slowdown, and almost 10 times slowdown with a 5% workflow share. The observed slowdown is due to the elevation of the wait time baseline due to the > 1.0 job pressure from the pilot jobs. In opposition, when the workload contained 1% of Montage workflows but were scheduled using WoAS there was no effect on non-workflow jobs.

The experiments run with chained job approach show the smallest changes in slowdown with maximum variations of 2x for over 50% workflow share scenarios. The chained job scenarios are the best possible fairness scenario for each workflow and workflow share since workflows are handled as regular jobs. It is significant that regular job's slowdown seems to stop growing after workflow shares of 25%, even decreasing for floodplain. This effect can be explained by the lower priority of the floodplain jobs, that are larger than a good share of the workload jobs.

Gain(%)	1%	5%	10%	25%	50%	75%	100%
floodP	1.80	5.22	14.46	29.29	44.53	51.64	64.47
longW	2.30	8.33	18.93	30.84	40.25	31.99	27.18
wideL	0.33	10.64	19.74	32.35	48.22	57.19	66.16
cybers	1.66	7.72	13.92	25.58	36.72	44.45	52.83
sipht	2.55	11.41	18.16	34.85	42.77	37.27	35.83
montage	12.36	44.90	60.30	72.34	80.13	82.14	85.26

Table 3: Difference of actual utilization of WoAS over the pilot job approach for different workflow shares.

In the case of the workflow aware approach, in experiments with smaller shares ($\leq 10\%$) jobs slowdown is almost the same as the case with zero workflows. For the rest of experiments (except Montage over 25%), WoAS shows only slightly bigger slowdowns ($< 2x$) than the chained job approach. Big increases on slowdown (over 4x) on Montage and large workflow shares ($> 25\%$) point that, for workloads heavily dominated by workflows with large resource allocations, WoAS might have a large impact on smaller jobs. In system heavily dominated by workflows, regular jobs might not be as important. This is an effect of the backfilling limited queue processing depth (common performance optimization practice) where non-workflow jobs have to climb up the queue by waiting longer. A technique filtering non ready queue jobs before the scheduling processes are started, could alleviate this problem.

Finally, slowdown analysis was performed for all the workflows in Table 2 and for medium ($[96, 480)$ core hours) and large ($[480, \infty)$ core hours) jobs. Since results were similar, the data was not included due to space limitations.

5.2.3 System utilization. We compare the actual utilization between running the same experiment using workflow aware scheduling, pilot jobs, and chained jobs.

Data shows that experiments using WoAS and chained jobs presente utilization over 90% and differences $\leq 5\%$. As we already analyzed in Section 5.2.1, the only exception are the experiments with the LongWide workflows and workflow shares of 75% and 100% - chained jobs show over 90% utilization, but the WoAS ones show $\approx$70% and $\approx$60%.

Compared with the pilot job approach, WoAS has much higher levels of actual utilization as the workflow share increases.

Thus, we see that WoAS does not waste resources while producing good turnaround times (Section 5.2.1), which indicates its suitability as a workflow scheduling technique.

5.2.4 Summary. For workloads with moderate workflow presence ($<$ %50 core hours) WoAS presents the shortest workflow turnaround times, keeps high system utilization (over 90%), wastes no resources, and regular jobs slowdown is equivalent to the best case scenario.

For workflow dominated workloads, WoAS showed the shortest turnaround times and high utilization except for the LongWide experiments. However, slowdown of regular jobs higher than the chained job cases.

As a final note, it is possible that WoAS could perform better in the dominated workflow scenarios if some queue filtering was added (not considering jobs with dependencies for queue construction) and with workloads with more workflows diversity. However that is subject of future work.

6 CONCLUSIONS

We propose Workflow Aware Scheduling (WoAS), a new model for a batch queue scheduler that enables unmodified pre-existing scheduling algorithms to take advantage of the fine grained resource requirements to produce short turnaround times without wasting resources. We implemented WoAS and integrated it in Slurm, and our implementation will be available as open source. We evaluated WoAS by simulating NERSC's Edison supercomputer and its workload, modeled after the study of three years of its real job traces.

Our results show that with WoAS, workflows show significantly shorter turnaround times than the chained job and single job approaches, and no wasted resources. In traces that have moderate workflow presence (< 50% core hours), using WoAS, FCFS and backfilling achieves turnaround times as short or shorter than submitting workflows as single jobs and much shorter than as chained jobs (up to 3.75x speedup), while keeping the system highly utilized (over 90% and *no allocated idle resources*). It also produces utilization gains over the single job approach (e.g. 60% for Montage workflows, 10% workflow share) and, has no or negligible impact on the slowdown on non workflow jobs.

Similarly, in workloads dominated by workflows (≥ %50) experiments show that workflow performance is similar to the one stated above. However, other scheduling mechanisms, like the queue depth limits in the backfilling algorithm, may affect WoAS increasing the non workflow job slowdown, specially in the presence of large workflows (e.g. 7x in Montage, 75% share). This effect could be eased by filtering not ready jobs in the waiting queue, which is future work.

We conclude that WoAS workflow scheduling performs significantly better than current approaches to executing workflows on HPC systems while posing no significant drawbacks.

7 ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR). The National Energy Research Scientific Computing Center, a DOE Office of Science User Facility, is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE and the Swedish Research Council (VR) under contract number C0590801 (Cloud Control).

REFERENCES

[1] Gonzalo Pedro Rodrigo Alvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. 2016. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 521–526.

[2] Stephen Bailey. 2016. (01 2016). https://bitbucket.org/berkeleylab/qdo

[3] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. 2008. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*. IEEE, 1–10.

[4] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. 2004. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*. Springer, 11–20.

[5] T. Fahringer, R. Prodan, R.Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. 2007. ASKALON: A Development and Grid Computing Environment for Scientific Workflows. In *Workflows for e-Science*, I. Taylor and others (Eds.). Springer-Verlag, 450–471.

[6] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. 2005. Parallel job scheduling, a status report. In *Job Scheduling Strategies for Parallel Processing*. Springer, 1–16.

[7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11.

[8] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, and others. 2015. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5037–5059.

[9] William TC Kramer and Clint Ryan. 2003. Performance variability of highly parallel architectures. In *International Conference on Computational Science*. Springer, 560–569.

[10] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.

[11] David A Lifka. 1995. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*. Springer, 295–303.

[12] Alejandro Lucero. 2011. Simulation of batch scheduling using real production-ready software tools. *Proceedings of the 5th IBERGRID* (2011).

[13] Muthucumaru Maheswaran, Shoukat Ali, HJ Siegal, Debra Hensgen, and Richard F Freund. 1999. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*. IEEE, 30–44.

[14] Hashim H Mohamed and Dick HJ Epema. 2005. The design and implementation of the KOALA co-allocating grid scheduler. In *European Grid Conference*. Springer, 640–650.

[15] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. 2007. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 43.

[16] Lavanya Ramakrishnan and Dennis Gannon. 2008. A survey of distributed workflow characteristics and resource requirements. *Indiana University* (2008), 1–23.

[17] Lavanya Ramakrishnan, Charles Koelbel, Yang-Suk Kee, Rich Wolski, Daniel Nurmi, Dennis Gannon, Graziano Obertelli, Asim YarKhan, Anirban Mandal, T Mark Huang, and others. 2009. VGrADS: enabling e-Science workflows on grids and clouds with fault tolerance. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.

[18] Gonzalo Rodrigo, P-O Östberg, Erik Elmroth, Katie Antypass, Richard Gerber, and Lavanya Ramakrishnan. 2015. HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems. In *The 24th International ACM Symposium on High-Performance Distributed Computing (HPDC)*.

[19] Gonzalo Rodrigo, Lavanya Ramakrishnan, P-O Östberg, and Erik Elmroth. 2015. A2L2: an Application Aware flexible HPC scheduling model for Low Latency allocation. In *The 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*.

[20] Rizos Sakellariou and Henan Zhao. 2004. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 111.

[21] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 351–364.

[22] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–10.

[23] Massimo Benini Stephen Trofinoff. 2015. Using and Modifying the BSC Slurm Workload Simulator. In *Slurm User Group*.

[24] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[25] Marek Wieczorek, Radu Prodan, and Thomas Fahringer. 2005. Scheduling of scientific workflows in the ASKALON grid environment. *ACM SIGMOD Record* 34, 3 (2005), 56–62.

[26] Jia Yu and Rajkumar Buyya. 2005. A taxonomy of scientific workflow systems for grid computing. *ACM Sigmod Record* 34, 3 (2005), 44–49.

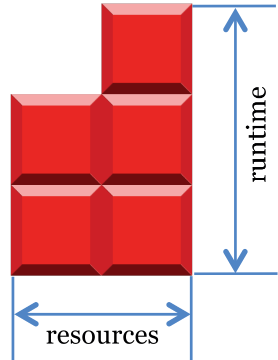
[27] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. 2005. Cost-based scheduling of scientific workflow applications on utility grids. In *First International Conference on e-Science and Grid Computing (e-Science'05)*. IEEE.

[28] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* 10 (2010).

[29] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. 2007. Swift: Fast, reliable, loosely coupled parallel computation. In *2007 IEEE Congress on Services*. IEEE, 199–206.

Department of Computing Science
Umeå University, SE-901 87 Umeå
www.cs.umu.se/english

Batch scheduling is like playing Tetris with jobs instead of pieces...
...only, you cannot rotate the pieces.



ISBN 978-91-7601-693-0
ISSN 0348-0542
UMINF 17.05