Contributions to Computing needs in High Energy Physics Offline Activities : Towards an efficient exploitation of heterogeneous, distributed and shared Computing Resources

Alexandre Boyer

▶ **To cite this version:**

Alexandre Boyer. Contributions to Computing needs in High Energy Physics Offline Activities : Towards an efficient exploitation of heterogeneous, distributed and shared Computing Resources. Performance [cs.PF]. Université Clermont Auvergne, 2022. English. NNT : 2022UCFAC108 . tel-04462655

HAL Id: tel-04462655

https://theses.hal.science/tel-04462655v1

Submitted on 16 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Contributions to Computing needs in High Energy Physics

Offline Activities:

Towards an efficient exploitation of heterogeneous, distributed and shared Computing Resources

A Thesis
Submitted to the Graduate School of Engineering Sciences
of the Université Clermont Auvergne
In fulfillment to the requirements for the degree of
Doctor of Philosophy in Computer Science
by

Alexandre F. Boyer

Supervised by David R. C. Hill
at the LIMOS laboratory - UMR CNRS 6158
And Christophe D. Haen
at CERN - EP-LBC group
Publicly defended on November, 30^{th} 2022

Committee:

Pr. Mamadou K. Traoré	Reviewer
Université de Bordeaux	
Dr. Ziad El Bitar	Reviewer
Université de Strasbourg	
Dr. Hélène Toussaint	Examiner
Université Clermont Auvergne	
Dr. Éric Innocenti	Examiner
Université de Corse	
Pr. David R.C. Hill	Supervisor
Université Clermont Auvergne	
Dr. Christophe D. Haen	Supervisor
CERN	
Dr. Federico Stagni	Guest
CERN	

Abstract

Pushing the boundaries of sciences and providing more advanced services to individuals and communities continuously demand more sophisticated software, specialized hardware, and a growing need for computing power and storage. At the beginning of the 2020s, we are entering a heterogeneous and distributed computing era where resources will be limited and constrained. Grid communities need to adapt their approach: (i) applications need to support various architectures; (ii) workload management systems have to manage various computing paradigms and guarantee the proper execution of the applications, regardless of the constraints of the underlying systems. This thesis focuses on the latter point through the case of the LHCb experiment.

The LHCb collaboration currently relies on an infrastructure involving 170 computing centers across the world, the World LHC Computing Grid, to process a growing amount of Monte Carlo simulations, reproducing the experimental conditions of the experiment. Despite its huge size, it will be unable to handle simulations coming from the next LHC runs in a decent time. In the meantime, national science programs are consolidating computing resources and encourage using supercomputers, which provide tremendous computing power but pose higher integration challenges.

In this thesis, we propose different approaches to supply distributed and shared computing resources with LHCb tasks. We developed methods to increase the number of computing resource allocations and their duration. It resulted in an improvement of the LHCb job throughput on a grid infrastructure (+40.86%). We also designed a series of software solutions to address issues in highly-constrained environments that can be found in supercomputers, such as lack of external connectivity and software dependencies. We have applied those concepts to leverage computing power from four partitions of supercomputers ranked in the Top500.

Keywords: High-Throughput Computing, High-Performance Computing, Grid Computing, Monte-Carlo simulation, Supercomputers.

Résumé

Repousser les limites de la science et fournir des services spécifiques et performant aux particuliers et aux communautés requièrent des logiciels toujours plus sophistiqués, du matériel spécialisé et un besoin croissant en stockage et puissance de calcul. En ce début de décennie, nous entrons dans une phase informatique distribuée et hétérogène, où les ressources seront limitées et contraintes. Les communautés employant les grilles de calculs doivent adapter leur approche : (i) les applications doivent supporter diverses architectures ; (ii) les systèmes de gestion de charge de travail doivent gérer plusieurs modèles de traitement informatique et garantir la bonne exécution des applications, en dépit de contraintes liées aux systèmes sous-jacents. Cette thèse se concentre sur le dernier point évoqué au travers du cas de l'expérience LHCb.

La collaboration LHCb s'appuie sur la World LHC Computing Grid, une infrastructure impliquant 170 centres de calcul répartis dans le monde, pour traiter un nombre croissant de simulations de Monte Carlo afin de reproduire les conditions expérimentales du projet. Malgré son envergure, l'infrastructure ne sera pas en mesure de couvrir les besoins en simulation des prochaines périodes d'exploitation du LHC en un temps raisonnable. En parallèle, les programmes scientifiques nationaux encouragent les communautés à s'approprier leurs supercalculateurs, des ordinateurs centralisant une puissance de calcul significative mais impliquant des défis d'intégration de taille.

Au cours de cette thèse, nous proposons différentes approches pour approvisionner des ressources de calcul hétérogènes et distribuées en tâches LHCb. Nous avons développé des méthodes pour augmenter le débit d'exécution des programmes LHCb sur des grilles de calcul (+40.86%). Nous avons également conçu une série de solutions logicielles pour répondre aux limitations et contraintes que l'on peut retrouver dans des super calculateurs, comme le manque de connexion au réseau externe ou les dépendances des programmes par exemple. Nous avons appliqué ces solutions pour tirer profit de la puissance de calcul provenant de quatre partitions de super calculateurs classés au Top500.

Mots clefs : Calcul Haut Débit, Calcul Haute Performance, Grille de calcul, Simulation de Monte Carlo, Supercalculateurs.

Acknowledgements

Je tiens tout d'abord à remercier Mamadou K. Traoré, Ziad El Bitar, Hélène Toussaint ainsi qu'Éric Innocenti pour avoir fait partie de mon jury de thèse, et en particulier Mamadou et Ziad pour leur travail de rapporteurs.

Merci également à David, mon directeur de thèse, pour sa disponibilité malgré la distance. Tu m'as donné goût à la rédaction d'articles scientifiques et inculqué une partie de ton savoir, notamment dans le domaine de la répétabilité et reproductibilité. Merci à Christophe, mon superviseur au CERN, pour son soutien inconditionnel et la confiance qu'il m'a accordée. Merci à Federico, coordinateur technique sur DIRAC, pour sa patience ainsi que ses précieux conseils et enseignements sur le fonctionnement de DIRAC et des grilles de calcul en général. Je ne saurai jamais assez exprimer l'immense reconnaissance, le respect et l'humilité que j'ai envers vous.

Cette thèse n'aurait pu se faire sans le concours du groupe EP-LBC du CERN et des membres du projet DIRAC. Je tiens à remercier notamment Clara, Marco Ca., Christopher, Vladimir, Zoltan, Andrei, André, Sebastien, Ben, Marco Cl., Luìs, Paolo, Joel, Concezio. Vous m'avez inspiré, guidé et appris tout ce que je sais sur les besoins informatiques dans le domaine de la physique des hautes énergies. Dans le même esprit, je tiens à remercier le laboratoire LIMOS et son directeur de m'avoir accueilli.

Bien entendu, je n'en serai pas là sans le soutien indéfectible de ma famille. Je vous associe à mes succès présents et futurs.

Le sport à été le vecteur de nombreuses valeurs qui s'inscrivent dans ce travail. Laurent, tu m'as inculqué la discipline et m'as appris que chaque détail compte. Jonathan, je n'oublie pas que si j'ai pu faire quelques acrobaties sur le chemin, c'est bien parce que tu l'avais déblayé avant. Tu resteras une référence dans bien des domaines à mes yeux.

Enfin, je tiens particulièrement à remercier Marie pour m'avoir accompagné et supporté pendant ces années de thèses. Tes espièglerie sont sans limite, la vie est plutôt fun avec toi.

Une dernière pensée va à mes collègues, camarades et/ou amis de l'IUT de Clermont-Ferrand, de l'école ISIMA, du NIST et, plus généralement, à toutes les personnes que j'ai pu rencontrées et qui m'ont, délibérément ou non, de quelque manière que ce soit, façonné.

Contents

List of Figures

List of Tables

Introduction

Managing tasks on distributed and heterogeneous computing resources

Pushing the boundaries of sciences and providing more advanced services to individuals and communities continuously demand more sophisticated software and hardware, and a growing need for computing power and storage. Building complex software, operating and maintaining large-scale hardware infrastructure is expensive and require technical expertise not accessible to all (Gray, 2003). Therefore, to collect, filter, process and present an unprecedented amount of data, scientists and commercials generally rely on remote computing resources.

Workloads can involve tasks of different natures: CPU or IO-intensive, tightly or loosely coupled; leveraging a large number of specific hardware - such as GPUs, many-core nodes and high-speed networks - for a short period or commodity CPUs for months. Additionally, tasks can require specific software dependencies and data, or at least a way to access them.

According to their budget and needs, scientists and commercials can also expect a certain level of Quality of Service (QoS). In the same way, they may expect reproducible, trustable and secure computing environments. They generally provide incentives - money, recognition, crypto assets - or hardware and expertise, if any, in exchange for computing power.

Conversely, many institutes, laboratories and companies own underused and expensive computing power and storage. For recognition, financial and ecological reasons, they want to make the best use of their idle resources by sharing them with internal and external communities. To protect and keep control over their system, they usually impose constraints at various levels. They generally limit the use of hardware - in time and number of components - per individual, community or project. For security reasons, they might prevent outbound connectivity and the installation of

software. Administrators of the resources may provide QoS guarantees and mechanisms to secure the workloads. Infrastructures can rely on open standards facilitating interoperability between different systems or proprietary black-boxes.

Between computing resources suppliers and consumers, we find workload management systems relying on middleware. Middleware is defined as "a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system" (Bakken, 2001). Middleware help manage the heterogeneity and complexity of a distributed system, namely "a collection of autonomous computing elements interacting through a shared network" (Astley et al., 2001).

A Workload Management System (WMS), sometimes referred to as a workflow management system, provides a service responsible for the distribution and management of tasks across distributed computing resources (Andreetto et al., 2008). The ultimate aim of a WMS is to match the most adapted computing and storage resources with a certain task, given a list of constraints imposed by both actors. To satisfy suppliers and consumers, WMS should:

- Support relevant distributed and heterogeneous computing and storage resources for one or more communities.
- Get as many allocations as needed, as fast as possible if necessary. An allocation corresponds to a set of computing resources blocked for a given duration for a task.
- Use these allocations efficiently. Tasks should be adapted to the underlying resources and respect the allocation conditions (duration, number of cores). For instance, loosely-coupled tasks would not be adapted in a supercomputer with high-network connectivity, unless nodes remain idle.

WMS might ensure that no actor intends to cheat the systems, and that software reproducibility and privacy are guaranteed. WMS might also monitor workloads across distributed and heterogeneous computing resources, and intervene when an incident occurs. WMS are often coupled with a Data Management System (DMS) to process data-driven tasks, and feature resources discovery, matchmaking and accounting processes. The combination of these components eases the interactions between actors by delivering a simple interface hiding the complexity of the workload management.

Under the hood, they deal with (i) the workloads, their software dependencies, input and output data; (ii) diverse infrastructures and computing models involving different paradigms, heterogeneous protocols and hardware components.

A growing need for computing power in High Energy Physics

The Standard Model of particle physics - a theory describing the fundamental particles and their interactions - has successfully explained various phenomena and experimental results, but remains incomplete and leaves many questions open ("LHC Season 2 facts & figures", 2018). To validate and develop the Standard Model of particle physics, the European Organization for Nuclear Research (CERN) leverages a chain of particle accelerators that speed up a beam of particles before ending in the Large Hadron Collider (LHC). Inside the LHC, two particle beams, traveling at close to the speed of light in opposite directions, collide and provide data about constituents of matter, which are captured by four detectors corresponding to distinct experiments: ALICE, ATLAS, CMS and LHCb. Experiments capture millions of events every second that have to be filtered, processed and stored. In parallel, to better understand the impact of detector effects and experimental conditions, experiments also model events occurring in the detectors by executing Monte-Carlo simulation applications: they both reproduce the generation of events and the configuration of the detectors (Clemencic et al., 2011).

CERN does not have the financial resources to process on-site the totality of the events - simulated and real - and currently relies on the Worldwide LHC Computing Grid ("Worldwide LHC Computing Grid", 2022) to deliver nearly real-time data to physicists. This infrastructure currently involves 170 autonomous computing centers spread within 42 countries, 1 million computing cores and 1 Exabyte of storage. More than 50 Petabytes of data are distributed and analyzed every year.

This sole approach was reliable during LHC Run1, but LHC has produced a growing amount of data since then. According to the analysis of Stagni et al. on the CPU cycles used in 2016, all the LHC experiments consume more CPU hours than those officially pledged to them by WLCG (Stagni et al., 2017). Moreover, in the coming LHC Run3 and then the High-Luminosity Large Hadron Collider (HL-LHC) (Apollinari et al., 2015) era, experiments are expected to produce up to an order of magnitude

more data compared to the current phase (LHC Run2). In the meantime, computing infrastructure and funding models are changing, and national science programs are consolidating computing resources and encourage using cloud systems as well as supercomputers (Barreiro et al., 2019).

Dealing with LHCb workloads

In this thesis, we focus on the workload management of the offline activities of the LHCb experiment. We study the efforts that have to be made to exploit additional computing power in order to handle the upcoming LHCb workload from the LHC Run3 and further, from the HL-LHC infrastructure. The purpose is to provide different approaches to increase the throughput of the jobs on available computing resources, namely the number of jobs we can execute on distributed computing resources in parallel. This involve a better use of the already supported computing resources, and novel mechanisms to integrate the LHCb workload on non-adapted and heterogeneous computing resources.

We work on the workload management system of LHCbDIRAC (Stagni and Charpentier, 2012), the middleware designed by the LHCb experiment to originally interact with WLCG distributed and shared resources with tasks and data. LHCbDIRAC is an extension of the general-purpose DIRAC Interware project ("DIRAC", 2022; Tsaregorodtsev, 2014) developed since 2003. The middleware combines both a Workload Management System (WMS), to handle and orchestrate job requests among distributed and heterogeneous resources, and a DMS, which includes automated data replication with integrity checking, needed to deal with large volumes of data. DIRAC has been adopted in various contexts such as the Belle II experiment (Miyake et al., 2015), the Cherenkov Telescope Array (CTA) (Arrabito et al., 2012) and the European Grid Infrastructure ("DIRAC EGI", 2022).

Outline

The thesis is structured into four main chapters. Chapter 1 introduces the current heterogeneous computing landscape. It defines the content and features of tasks and jobs, and compares different computing paradigms. It also presents the current

hardware components and classes available, and how they can be combined to handle tasks. More particularly, it emphasizes the heterogeneity of the resources at various levels. To finish, it describes the LHCb workloads and available resources, and how the evolution of the computing landscape has influenced the development of the experiment.

Chapter 2 proposes a review of the current literature regarding the integration of embarrassingly parallel tasks with limited inputs on various distributed infrastructures. Embarrassingly parallel tasks refer to problems requiring no effort to be separated into a number of parallel tasks. This mainly implies software and middleware solutions to efficiently harness computing power under constraints. It highlights methods to (i) provision computing resources with jobs, (ii) provide a reproducible environment including the dependencies of the jobs and (iii) efficiently harness the allocated resources.

Having surveyed the constraints, the solutions developed around distributed and heterogeneous computing resources and their limitations, we emphasize various levers at our disposal to use additional computing power in the LHCb experiment context (Chapters 3 and 4). Chapter 3 focuses on improvements related to already supported computing resources, which mainly come from WLCG. This involves changes in the DIRAC Pilot-Job provisioning tool, the Site Director, and the DIRAC fast CPU benchmarking solution called DIRAC Benchmark.

Chapter 4 gathers known constraints and solutions we developed to integrate LHCb workloads on supercomputers. This also includes a general model gathering commonalities of all the efforts done by LHC experiments on supercomputers and different practical use cases allowing us to test our approaches.

We will now start by surveying the current computing landscape and how it impacts the development of the LHCb experiment.

1 Computing landscape at the beginning of the 2020s: heterogeneous solutions for complex workloads

1.1 Introduction

Artificial Intelligence (AI) is fueling a revolution in how businesses and researchers think about problems and their computational solutions (Reed et al., 2022). This new class of problems spreads fast and increasingly needs computing power and performance. Therefore, there is growing user demand for cutting-edge and expensive hardware components and infrastructures.

Large companies leading the hardware market, followed by hardware startups are surfing on the trend and shaping the computing landscape of a new decade. Primarily by means of cloud services, they provide an access to shared and distributed computing resources. They have a direct influence on manufacturers designing components, which are becoming more and more specialized. In the meantime, there are still many workloads running on commodity CPUs, such as Monte-Carlo simulations in High Energy Physics (HEP). On the one hand, developers need to reshape their workloads accordingly. On the other hand, not all applications can be ported to specialized architectures.

After decades of homogeneity through the use of x86 CPUs in WLCG grid sites, the LHCb experiment has to evolve in this new heterogeneous environment to handle a growing amount of data. This implies software and distributed computing challenges.

In this chapter, we first present the form that a problem can take (Section 1.2). Then, we propose a survey of the current hardware components and classes, and how they work together in a wide area network to provide computing power to many different communities (Section 1.3). Finally, we describe the LHCb workloads and the computing resources available (Section 1.4).

1.2 About Jobs

1.2.1 Task, Workload, Workflow and Job

Many terms related to jobs are used differently according to their context. To progress through this thesis, we need to draw on common definitions of terms associated to the jobs. In Section 1.2.1, we provide a brief summary of our conception of a task, a workload, a workflow and a job.

Task

The term "Task" is usually defined as a unit of work. Turilli et al. define a computational task - mentioned as a task in this thesis - as "a set of operations to be performed on a computing platform, alongside a description of the properties and dependencies of those operations and indications on how they should be executed and satisfied. Implementations of a task may include wrappers, scripts, or applications" (Turilli et al., 2018). Despite a clear definition, "Task" remains an ambiguous word, even in the field of computer science. Indeed, a running task can be represented as a process or a thread. In our context, we decided to refer to a running task as a process, and threads as operations within a task. A task can have (i) its own business or science value or (ii) contribute to a common effort as part of a larger scheme. Its dependencies include data, software and hardware.

The Quality of Service (QoS) of a task refers to its quantitative and qualitative features, necessary to achieve a set of initial requirements. Cardoso et al. propose a QoS model for web service that fits with our task definition (Cardoso et al., 2004). The authors present four main dimensions: time, cost, reliability and fidelity. Time - needed to transform inputs into outputs - is a basic and universal measure of performance. Cost refers to any financial, human and supply involvement during the task management and processing. Reliability is a function of the failure rate: a task has one initial state and two distinct terminating states which are "done" and "failed". A "done" task does not provide information about how good a produced result is. Fidelity is a measure of the quality of the task output. Cardoso et al. provide two additional dimensions to their model for tasks with stronger requirements: maintainability and security. Maintainability represents the mean time necessary to repair a task failure, namely to maintain it in a condition where it can perform its intended function. Security refers to mechanisms - such as authentication, access control, labels, audits, system integrity - and development techniques - formal specifications, formal proofs, tests - to ensure the confidentiality of the execution.

Workload

We define "Workload" as a set of tasks. In this paradigm, the collective outcome of the tasks is relevant. We distinguish two main types of workloads: the Bag-of-Tasks (BoT) and the workflow. A BoT is a set of independent, indistinguishable and embarrassingly

parallel tasks. BoT are generally used to perform massive searches, fractal calculations, or simulations (Cirne et al., 2003). On the contrary, a workflow, defined hereafter, represents "a workload with arbitrarily complex relationships among the tasks" (Turilli et al., 2018).

Workflow

Yu and Buyya propose a comprehensive overview of various workflow design features (Yu and Buyya, 2005). The structure defines the temporal relationships between tasks. Direct Acyclic Graph (DAG) workflows can integrate sequences - ordered series of tasks -, parallelism - tasks running concurrently -, and choice. In addition to the previous patterns, non-DAG workflows can include iterations, where one or more tasks are repeated until a certain condition is true. The model includes, or not, information about dependencies. Abstract models do not specify low-level information about resources and data movements and are, therefore, much more portable than concrete models. In contrast, concrete models might be more efficient to control workflow execution. There exist different means to assemble tasks and compose a workflow. User-directed composition systems allow users to edit workflows directly, under the form of a language (e.g. XML), or a graph (e.g. UML). Automatic composition systems - as precised in the term - automatically generate workflows for users. Workflows have similar QoS constraints to tasks. Users may define QoS constraints at the task level, and/or at the workflow level. In the latter case, the underlying system decides how fast each task has to be processed.

Job

Turilli et al. define the term "Job" as a type of container used to acquire - mostly computing - resources on a computing system (Turilli et al., 2018). They can be considered as metadata wrappers - containing dependencies requirements, QoS constraints explicitly defined - around one or more executables. A job can include a single task, a whole workflow or part of a workflow depending on user needs and system specifications. A job is a component used to communicate the wrapped task/workflow capabilities to a computing system, which aggregates information and orchestrates resources accordingly. Generally, the more a job is flexible, providing accurate information, the sooner it will acquire resources. Feitelson et al. characterize jobs

running in a multi-processor system according to their flexibility, level of preemption, knowledge available and memory allocation (Feitelson et al., 1997).

1.2.2 High-Throughput, High-Performance and Many-Task Computing

Scientific and business applications dealing with workloads have mainly relied on two distinct computing paradigms, which have largely influenced the architectures and models of most of the current computing infrastructures: High-Throughput Computing (HTC) and High-Performance Computing (HPC).

High-Performance Computing

A typical HPC workload "will operate on a single, large volume of data, in the form of initial data files or checkpoint files which are provided by the user or written by a previous task in the workload" (Huerta et al., 2019). An HPC workload is generally composed of tightly coupled parallel tasks wrapped within a single job, and uses a message passing interface (MPI), an Open multi-processing (Open MP) interface or a mix of both, to achieve inter-process communication. It is executed within a particular machine with low-latency interconnects (Foster et al., 2008): since IO is typically a synchronization point between processes, insufficient IO bandwidth becomes a significant bottleneck. Therefore, tasks of an HPC workload are not executed across widely distributed computing resources. Floating-point operations per second (FLOPS) have been the yardstick used by most HPC efforts to rank their systems (Livny et al., 1997). Twice a year since 1986, Top500.org releases a list of the 500 best general-purpose computing systems of the world based on their HPC abilities (“Top500 The List”, 2022). Performances of the machines are benchmarked with Linpack (Dongarra et al., 2003; Dongarra et al., 1979), which reflects the performance of a dedicated system for solving a dense system of linear equations.

High-Throughput Computing

Nevertheless, for many experiments, scientific progress and quality of research are tightly bound to computing throughput as Livny et al. emphasize (Livny et al., 1997). According to them, most scientists are concerned with how many floating-point

operations per week or month they can extract from their computing environment. A typical HTC workload consists of running many loosely coupled and independent tasks - BoT - requiring a large amount of computing power during a long period (Rho et al., 2012). In such a context, maximizing the number of resources - composed of commodity CPUs and memory - accessible to the users is preferred over providing high-efficiency computing resources.

Many-Task Computing

Many-Task Computing (MTC) aims to bridge the gap between HPC and HTC (Raicu, 2009; Raicu et al., 2008). MTC is a broad class of workloads that can include small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive, static or dynamic, loosely coupled or largely coupled tasks. An MTC workload tends to involve an extremely large amount of computing power, data and tasks over short periods, where primary metrics are measured in seconds (e.g. FLOPS, tasks/sec) as opposed to tasks per month in HTC. Communication within an MTC workload is generally not naturally expressed using standard MPI commonly found in HPC, drawing attention to the many computations that are heterogeneous but not "happily" parallel (Raicu et al., 2008). In Section 1.3, we present the influence of these computing paradigms on the computing resources and their components.

1.3 A variety of Computing Resources

1.3.1 Hardware components

Completing a task involves the interactions of many different components, aggregated into a computing resource, from hardware to software. Challenges arise when it comes to exporting tasks from a given computing resource to another one: combinations of heterogeneous components are infinite and outcomes tend to be hardly repeatable. In Section 1.3.1, we provide a brief summary of the main hardware components that can be found in computing resources and their evolution through time. This will be followed by a taxonomy of the existing computing classes gathering these hardware components (Section 1.3.2) and how they can interact with each other to solve large scientific and industrial problems (Section 1.3.3).

Central Processing Unit

The central processing unit (CPU), also named "processor", is the core of a computing resource. A CPU takes the form of an electronic circuitry executing instructions received from hardware and software running on the computing resource. A CPU can be composed of a certain number of cores, each of them containing three main units: the Control Unit (CU), the Arithmetic Logic Unit (ALU) and the Registers. The CU decodes the instruction and controls the operations performed with the two remaining units. The CU contains a clock to control the rate at which instructions are performed. The ALU performs arithmetic operations such as incrementation and subtraction, intermediate results related to the current operations are stored in the Registers, which are small capacity memory cells. CPU performances are affected by three main factors (Tremblay et al., 1998): (i) how fast you can crank up the clock; (ii) how much work you can do per cycle; (iii) how many instructions you need to perform a task.

Two main instruction set architectures (ISA) have significantly influenced the design of the CPUs since the 1970s: Complex Instruction Set Computer (CISC) and Reduced Instructions Set Computer (RISC). The CISC approach consists in minimizing the number of instructions per task, providing hardware flexibility. CISC processors tend to propose a large number of complex instructions able to load, evaluate and store data. This comes at a cost: instruction sets may include instructions of different sizes and clock cycles. RISC, implemented by John Cocke (Cocke and Markstein, 1990) and coined by Patterson (Patterson and Ditzel, 1980), involves a limited number of simple instructions of the same size that can be executed within one clock cycle. RISC CPUs perform much more operations than CISC CPUs to execute a given task but in less number of cycles. Blem et al., from the University of Wisconsin, revisited the RISC vs CISC debate (Blem et al., 2013). Their study suggests that whether the ISA is RISC or CISC is irrelevant, evolution has been continuous, focused on enabling specialization and agnosticism of RISC or CISC.

There exist various CPU families based on RISC and CISC. CISC ISA, mainly through Intel x86 but also AMD CPUs, has largely dominated the desktop and high-performance server market. ARM-based CPUs - RISC ISA - have led the tablet and smartphone market, and are also entering the high-performance server market (“Top500 The List”, 2022). These companies propose various models adapted to different types

of tasks and budgets, based on clock speed and transistor size.

In recent years, because of various fundamental limitations in the fabrication of integrated circuits - such as power, heat restrictions and transistor size - multi-core processors have become the norm: "Rather than solely looking to increase the performance of a single processing core, why not put more than one in a personal computer? In this way, personal computers could continue to improve in performance without the need for continuing increases in processor clock speed" (Sanders and Kandrot, 2010). This process started with the introduction of Hyper-Threading by Intel (Marr et al., 2002) followed by the introduction of logical cores. A single physical core often appear as two logical cores, also named hardware threads. Modern and high-performance CPUs such as AMD Optimized $3^r d$ generation EPYC now contain 64 physical cores seen as 128 logical cores by many operating systems. Each core has also dedicated vector units supporting Single Instruction Multiple Data (SIMD) used for vector data processing, another axis of parallel software (Flynn, 1966). This encouraged the development of multi-threaded software running on the same computing resource. Yet, even though Hyper-Threading and embedded vector processing enable CPUs to maximize the use of the cores, most software do not support all these aspects (Gramoli, 2017).

Co-Processors and Accelerators

Even as the number of cores in CPUs continues to increase, many difficulties remain in performing specific types of operations: floating-point arithmetic, graphics and cryptography, for instance. A co-processor aims to supplement the functions of the primary processor, namely the CPU. By offloading processor-intensive tasks from the CPU, co-processors can improve the whole system performances. Over time, economics and potential performances have motivated the migration of co-processor abilities inside the CPU. In 1989 for instance, Intel integrated the former 80387 co-processor, dedicated to floating-point operations, within their 80486 processors to eliminate the external communication delays (“80486 - Intel”, 2022). Accelerators have resisted that trend. Contrary to co-processors, which are highly connected to the internal of the main processor, accelerators are typically seen as independent I/O devices, programmed through an interface.

The Graphics Processing Unit (GPU) is a programmable accelerator composed of

thousands of specific processing cores running simultaneously, facilitating real-time execution and massive vector data processing. According to Fan et al., the computational power of commodity GPUs has exceeded that of PC-based CPUs, and - driven by the game industry - GPU performance has approximately doubled every six months since the mid-1990s (Fan et al., 2004). Even if they do not exactly fit all scientific applications because of their vector design, GPUs have become more general purpose and are, since more than a decade, an integral part of mainstream computing systems. GPUs are particularly efficient with applications with the following characteristics (Owens et al., 2008): computational requirements are large, SIMD parallelism is substantial, and throughput is more important than latency. Frontier, the first exascale supercomputer, is composed of AMD Instinct MI250X GPU accelerators, each of them providing up to 47.9 TFLOPs in double precision.

This performance advantage comes at a price. The programming model differs fundamentally from that of CPUs. As a consequence, existing programs cannot run directly on GPUs. This issue constitutes one of the reasons Intel introduced the Xeon Phi accelerator in 2013: a many integrated core architecture (MIC) based on x86 technology, generally composed of 61 cores, namely 244 hardware threads. Xeon Phi received a lot of attention since one could cross-compile x86 applications for this architecture, making scientific applications easier to port on such an accelerator. In 2013, it was part of Tianhe-2A, the largest supercomputer in the world at that time (Xeon Phi 31S1P). In 2016, it was embedded in 6% of the supercomputers of the Top500. It is worth mentioning that, during these years, the Laboratory of Informatics, Modeling and Optimization of the Systems (LIMOS), located in Clermont-Ferrand in France, developed several research activities around Xeon Phi accelerators and HEP. They designed a method for porting HEP software to Xeon Phi (Schweitzer et al., 2014) and studied its performances and reproducibility potential regarding stochastic simulations (Dao et al., 2014; Schweitzer et al., 2015). In June 2022, Xeon Phi is still embedded in 1% of the supercomputers of the Top500, although it was abandoned by Intel in 2018.

As computing needs have become more specialized, manufacturers started to design application-specific integrated circuits (ASIC). Because they considered GPU as too general-purpose to run specific AI workloads efficiently, Google engineers developed their custom ASIC chip: the Tensor Processing Unit (TPU) (Jouppi et al., 2017). In response, chip manufacturers and major IT companies have entered the

competition. Among them, we can mention Intel, which has developed the Nervana Neural Network Processor-T (NNP-T) with the ambition of offering direct integration with popular deep learning frameworks (Hickmann et al., 2020), unlike the Google TPU designed for their machine learning library. It is also worth citing Cerebras System, an AI-accelerator company, which introduced the largest processor in the industry, coming with 2.6 trillion transistors: CS-2. Contrary to GPUs and other ASICs, CS-2 can host large AI models, removing the need for mastering model parallelism and tensor parallelism. Reuther et al. analyzed different AI ASICs and defined CS-2 as one of the most powerful AI chips in 2021 (Reuther et al., 2021). Nevertheless, to cope with the ever-growing need for specialized hardware to run more complex and specific applications, and avoid designing a different accelerator for each application need, manufacturers and engineers have adopted Field Programmable Gate Arrays (FPGA).

An FPGA is a re-programmable accelerator (Monmasson and Cirstea, 2007), which is considered by an increasing number of designers from various fields such as telecommunications, image and signal processing and AI. FPGA is highly flexible, reduces (i) time-to-market by bypassing costly manufacturing cycles engaging a lot of manpower and (ii) the impact of design mistakes. However, programmers do not have any control over power optimization and are limited by the resources available in the FPGA. FPGA is convenient for prototyping and low quantity production.

Memory and Storage

To reach the CPU, task instructions and mainly data are moved from non-volatile memory (NVM) to volatile memory closer to a CU. On the one hand, NVM refers to long-term persistent and mass storage, retaining data even when the computer power is off. We define two types of storage: hot storage and cold storage. Hot storage refers to solutions quickly accessible, gathering frequently and actively used data, and takes the form of a solid-state drive (SSD), a hard disk drive (HDD), or read-only memory (ROM) to load a given operating system. Cold storage represents objects containing archived or rarely used data. Tapes are a popular type of cold storage. Hot storage is generally faster but more expensive and less reliable than cold storage.

On the other hand, volatile memory, also named random access memory (RAM), requires power to maintain the stored information. Volatile memory, also named main

memory, is usually faster and, therefore, more expensive than NVM. We distinguish two main types of RAM: Static RAM (SRAM) and Dynamic RAM (DRAM). DRAM consists in storing data in capacitors, which gradually discharge energy. To prevent data loss, a periodic refresh of power is required. DRAM is generally used to implement main memory. SRAM, on the contrary, consists in storing data in transistors and does not require any refresh of power to keep the data intact. It is often employed to implement cache and internal registers of the CPU. DRAM tends to provide greater memory capacities but slower access speed and higher power consumption than SRAM.

DRAM can operate either synchronously or asynchronously. Asynchronous DRAM is not coordinated with the system clock and, thus, has some latency that minimizes the speed. Asynchronous DRAM is not appropriate for modern high-speed memory systems anymore. Conversely, with synchronous DRAM (SDRAM), all operations are controlled by a system clock and synchronized with the clock speed of the CPU. It allows for much higher clock speeds than conventional DRAM. Single-data-rate SDRAM (SDR SDRAM) was one of the first synchronous memory architectures and was able to transfer one machine word - 16 bits on x86 CPUs - of data during one clock cycle. It was widely used in the 1990s but is now obsolete. Double-data-rate SDRAM (DDR SDRAM) was the next iteration of SDRAM. The main advantage of DDR SDRAM was the ability to transfer data on both the rising and falling edges of the clock. It allows sending twice the data per clock cycle. DDR SDRAM is still largely employed nowadays and was upgraded through time: DDR2, DDR3, DDR4 and DDR5 with a memory frequency of 4.8 MHz. While the underlying components and functionalities remain similar, each new version increases the clock speed. For instance, DDR2 SDRAM doubled the clock speed compared to DDR SDRAM, which allowed doubling data transfer speeds while maintaining the same bus speed - 6.4 GB/s were potentially delivered. In comparison, DDR4 SDRAM could theoretically provide 25.6 GB/s, whereas DDR5 SDRAM could reach 32 GB/s and manufacturers are already announcing a memory bandwidth of 160 GB/s for the DDR5-10000 SDRAM. DDR5 SDRAM benefits from a power-efficient design - with the transfer of the power management from the motherboard to the RAM itself - and improved reliability features (Criss et al., 2020).

On high-performance machines, data generation rates increase faster than traditional parallel file system ingestion capabilities (Khetawat et al., 2019). To cope

with this bottleneck, the traditional parallel file system is generally extended with an intermediate component: a high-bandwidth flash-based storage device called burst buffer (BBs). These BBs sit between CPUs and the parallel file system, and are designed to absorb the periodic I/O bursts of HPC tasks. On supercomputers, BBs are a multi-million dollar resource that impact the productivity of the center, the I/O performance of the workload and the scientific progress of the users.

Memory is designed as a pyramid where the most expensive and fastest components are quantitatively limited and close to the CPU. Nevertheless, the quantity and quality of memory components may vary according to the nature and needs of the tasks and the budget available.

Input/Output devices

Input/Output (I/O) refers to any peripheral designed to transmit input to and/or receive output from data to and from a computing resource. To make its way to a CU, a task has to be: (i) developed within the computing resource using I/O devices; (ii) sent to an external storage solution (ii) sent over the network. Developing a task commonly requires input peripherals such as a mouse and a keyboard to supply storage with content, while a computer monitor - an output device - provides live feedback on the content produced. Such devices are present on most desktop computers but remain uncommon on server solutions where a single Keyboard Video and Mouse (KVM) switch allows one to control multiple computers only for setup or maintenance purposes. Tasks and data can also be stored on external storage devices, such as an external HDD or USB key, manually transported to another computing resource.

Computing resources are usually equipped with a network interface card (NIC), an I/O device allowing them to remotely communicate data with each other, within a specific area network. The connection can be wireless, passing by the Wi-Fi for instance, or bound to an Ethernet or Infiniband (Pfister, 2001) cable depending on latency and bandwidth requirements of the tasks. We distinguish two main types of area networks: Local Area Network (LAN) and Wide Area Network (WAN). LANs are characterized as small, localized, fast and secure, often owned and managed in-house by the organization where they are deployed. For instance, a LAN can be represented as two computing resources connected to a single switch or a local cluster of computing resources linked by a fast Infiniband interconnect. WANs, conversely,

enable more widespread connectivity covering larger areas such as cities or countries. Internet can be considered the largest WAN.

Operating System and software

The operating system (OS) is the main software running on a computing resource. It manages the hardware and software resources and allows tasks to interact with the inner computing components. There exists a variety of OS grounded on different philosophies and targeting specific markets. MacOS and Windows have mostly focused on user experience at the expense of security and an open-source code base like GNU/Linux. Many distributions - a consistent set of software - are based on the GNU/Linux kernel: Ubuntu and Fedora are known to be user-oriented, while Red Hat Enterprise Linux or ArchLinux are server-oriented.

Most OSs perform the same basic functions such as processes, memory, I/O management and security. An OS uses a multi-tasking mechanism, via a scheduler, to swap tasks in and out of the CPU so that they appear as running simultaneously. It determines the amount of memory to allocate to each process and keeps track of the allocated portions of memory. In the same way, it controls the location of the stored data so that they can be smoothly accessed when requested. It can handle device drivers, programs controlling the operations of specific types of external devices. Once a driver is installed on the OS, this one is able to communicate with it and forward requests to and from the software on the computing resource. The OS prioritizes processes and can interrupt them - suspend the execution and swap to a more important task - based on I/O signals. Finally, an OS ensures the safety of a computing resource by providing means of creating user accounts and specific permissions on data. They can include specific utility software such as firewalls and anti-malware as well.

Software is originally developed in a human-readable language, which cannot be directly interpreted by the CPU: the source code has to be translated to a low-level and OS/CPU-specific set of instructions. There exist two kinds of translators, which largely influence the design of the computing languages: compilers and interpreters. While in practice a programming language can depend on both processes - pre-compilation and just-in-time compilation for instance -, we tend to associate them with a single one. The purpose of a compiler is to convert code from a source language to another language in order to create an executable file after a linking phase. Most

of the compilers perform general operations such as pre-processing, lexical analysis, parsing, semantic analysis, code optimization and generation. Cross compilers are a specific but largely used type of compiler able to create executable code for a variety of platforms, referred to as a combination of an OS and a CPU. Compiled programming languages such as C and C++ are commonly translated to executable code by GCC (Von Hagen, 2011) or Clang (Lattner, 2008). Aho et al. present a comprehensive analysis of compilers in their book: "Compilers: Principles, Techniques, and Tools" (Aho et al., 2007). Alternatively, the goal of an interpreter is to transform and executes the source code of the software. There are different ways of performing the translation: (i) parsing the source code and executing it directly; (ii) transforming the source code into an intermediate representation or object code and executing it; (iii) executing pre-compiled bytecode coming from a compiler using a Virtual Machine (VM). Shell programs are parsed and executed line by line, whereas Java, Python and Go are first compiled into bytecode, which is then interpreted by a VM.

The concept of the virtual machine was developed in the 1970s by IBM, originally as a method of time-sharing expensive mainframe hardware. At this time, Meyer and Seawright defined VM as a "software replica of a complete computer system", consisting of a data structure describing the memory size and the I/O configuration of the simulated system (Meyer and Seawright, 1970). VMs could support many OS, and thus provide software portability across various underlying platforms. Nowadays, there is a wide variety of VMs built by OS and compiler developers as well as language designers. We distinguish system VMs providing a complete system environment from process VMs, such as the Java Virtual Machine, capable of supporting an individual process (Smith and Nair, 2001). In the next sections, we will refer to the term VM as system VM. It is worth noting that the concept of virtualization has been largely adopted and engendered the notion of containerization, which will be reviewed in details in Section 2.3.1. Contrary to system VMs, which virtualize an entire machine down to the hardware layers, containers only virtualize software layers above the OS level. They offer a lightweight and fast-to-modify solution compared to VMs.

1.3.2 Aggregated into different computing classes

Moore's law (Moore et al., 1965) states that the number of transistors in a dense integrated circuit doubles every 18 months, though the cost of computers is halved.

Moore's law was validated over the course of the following half of the century and has driven the semiconductor industry and set targets for research and development. It has dramatically enhanced the impact of computing in nearly every segment of the world economy. There is no consensus on when Moore's law will cease to apply, but due to physical limits, it becomes more difficult to exploit its advantages and semiconductor advancements have slowed down over the last decade. It is worth noting that Moore's prediction was updated through time and also popularly widened to refer to processing power, which was the case until the 2000s. It was stated by Robert Dennard and known as Dennard's scaling (Dennard et al., 1974). Dennard's scaling - also known as MOSFET scaling - relates to Moore's law by claiming that the performance per watt of computing grows exponentially at roughly the same rate. Dennard's scaling appeared to break down in the 2001-2002 time period, circuits started to melt and since then the clock frequency of CPUs started to stabilize (Frank et al., 2001). The inability to operate within the same power envelope led the CPU industry to transition to multi-core architectures, creating significant challenges for memory technology.

Physical and economic constraints, as well as community needs and years of evolution, have shaped various classes of computing resources. Bell defines a computer class as "a set of computers in a particular price range with unique or similar programming environments that support a variety of applications that communicate with people and/or other systems" (Bell, 2008). According to him, a class may be the consequence and combination of a new platform with a new programming environment, a new interface and a new network.

According to Bell, classes evolve along three paths: (i) constant price and increasing performance of an established class; (ii) supercomputers - a race to build the largest computer of the day; (ii) and novel, lower-priced "minimal computers". Since the 2010s, microprocessors are the basis for nearly all classes from personal computers and commodity servers to scalable servers costing a few hundred million dollars. We are going to see different forms of computing resource, their origins and their features.

Mainframe

Meek defines "Mainframe" as a cabinet housing the CPU and main memory of a computing resource, separated from the I/O devices (Meek, 2003). Mainframes are

considered reliable and stable and characterized by their continuous and evolutionary improvement while maintaining backward compatibility. They are often used by IT organizations to host the most important and mission-critical workloads such as customer order processing, financial transactions, production and inventory control ("Mainframe concepts", 2010). Nevertheless, mainframe hardware and software remain large and expensive to acquire, and require high-end skills and specific training to work with.

In the 1950s, mainframes were large, bulky and expensive and reserved for a small number of privileged users, often from the same company or institution ("Evolution of computer networks", 2016). Originally, mainframes were not able to serve users interactively and concurrently. Users prepared punched cards containing data and program code and manually transferred their cards to operators that entered the cards into the mainframe. Users got the results of their program later in the form of a printout. With the size and cost reductions of the processors, new approaches based on terminals and VMs emerged.

Users now interact with a time-sharing mainframe via terminals - light computing resources composed of a display, a keyboard and a connection to the mainframe - located out of a computing center and onto their desktops. Users interact nearly simultaneously with the CPU and are able to start their program and receive the results almost immediately. Until the mid-1990s, mainframes provided the only acceptable means of handling the data processing requirements of a large business ("Mainframe concepts", 2010).

Desktop Computer

Desktop computers appeared in the 1960s and became prevalent in LANs as components became cheaper and smaller. Basic operations formerly based on mainframes were outsourced, first to minicomputers, and then to desktop computers.

Personal computers (PC) are a broad subclass, designed for the people, containing all the components seen in Section 1.3.1. They comprise commodity CPUs - from 2 to 20 cores - and memory - from 4 to 64 GB of RAM. They perform basic processing operations and store a few TB of data. In developed countries, they are now part of most of the households, mainly to write and store documents and surf the web.

Since 2003, they started to propose multi-threaded and, later, multi-core CPUs. The can also embed accelerators such as GPUs, mainly for gaming. Workstations are business-oriented desktop computers primarily designed to handle demanding tasks. It is defined as more robust and efficient than a PC and highly configurable to match specific needs and budgets. The market is dominated by Windows, which is the result of strong relationships between Microsoft and manufacturers delivering the OS with most desktop computers.

A desktop computer is usually exclusive to a physical person, even though many approaches - discussed later in Section 1.3.3 - have been built around this class to harness unused CPU cycles from thousands of hundreds of them.

Cluster

Sterling defines clustering as "a powerful concept and technique for deriving extended capabilities from existing classes of components" (T. L. Sterling, 2002). In the encyclopedia of physical science and technology (T. Sterling, 2003), he describes cluster computing as a class of parallel computer structure relying on "cooperative ensembles of independent computers integrated by means of interconnection networks to provide a coordinated system capable of processing a single workload". Bell and Gray define clusters with over 1,000 processors as massively parallel processors (MPP), and constellations as clusters made up of nodes with more than 16 processors (Bell and Gray, 2002).

Fostered by Amdahl's law (Amdahl, 1967), cluster computing was developed in the 1960s as an alternative to linking large mainframes to provide a cost-effective form of commercial parallelism (Buyya, 1999). Cluster computing did not gain momentum until the convergence of three important trends in the 1980s according to Yeo et al. (Yeo et al., 2006): high-performance microprocessors, high-speed networks, and standard tools for high-performance distributed computing. A possible fourth trend corresponds to the increasing need for computing power coupled with the low accessibility of traditional supercomputers. Academic projects such as Beowulf (T. L. Sterling, 2002), Berkeley NOW (Culler et al., 1997) and HPVM (Chien et al., 1997) have largely contributed to the emergence of cluster platforms. They promoted open source development platforms, vendor independence, low-entry costs to supercomputing-level performance and proved the advantage of clusters over other traditional platforms.

According to Yeo et al. (Yeo et al., 2006), the trend in parallel computing was to move away from traditional specialized supercomputing platforms, to cheaper and general-purpose systems consisting of loosely coupled components made from PCs and workstations.

Today, clusters remain widely used in science, engineering, commerce and industry applications. Potential user institutions have a "plethora of choices in terms of form, scale, environments, cost to meet their scalable computing requirements" (T. L. Sterling, 2002). Cluster computing resources are generally non-interactively shared among many users, mostly target non-service workflows, and are orchestrated by Local Resource Management System (LRMS).

Supercomputer

Bell refers to "supercomputers as the largest computers at a given time, coming into existence by competing and pushing technology to the limit to meet the unending demand for capability" (Bell, 2008). Supercomputers are primarily designed for academic and research purposes, and aim at processing specialized tasks requiring immense amounts of mathematical calculations such as scientific simulations, fluid dynamic calculations and animated graphics - HPC workflows.

CDC 6600, introduced as the culmination of several years of effort by Seymour Cray and his team, is often considered the first major supercomputer: it was an order of magnitude faster than any computer shipping at the time. Seymour Cray left CDC to propose "Cray style" computers that paved the way for supercomputing for 30 years with shared memory parallel functional units, which include vector processing. Nowadays, supercomputers are massively parallel clusters with large shared memory, containing many-core high-end nodes bound by fast connectivity systems such as Infiniband (Pfister, 2001). They all embed a GNU/Linux distribution and CPUs with vector processing capabilities, and many of them propose accelerators since 2010. The Top500 organization benchmarks supercomputers twice a year and provides statistics and trends in the area.

In 2022, the most powerful supercomputer, Frontier - built at the Department of Energy's Oak Ridge National Laboratory in Tennessee - have been the first of its class to break the exascale barrier with the linpack benchmark (1.1 exaFLOPS). After years

of "supercomputing monoculture" (Bell and Gray, 2002) led by Intel x86 CPUs and Nvidia GPUs, supercomputers have entered a new heterogeneous era driven by the growing needs for specific hardware able to handle AI workloads and by the arrival of Quantum computing.

Smartphone

Smartphones - and tablets by extension - appeared in the 2000s and have gained in popularity in the 2010s. They are the results of key technological advances, mostly in terms of miniaturization and networking. Progress on lithium-ion batteries has been an important factor in the emergence of this type of computing resource. They combine the abilities of a cell phone - namely calling and sending text messages - and the features of a personal computer. Smartphones embed a CPU, memory, storage as well as I/O devices such as a touch screen and camera, and can be identified as micro PC fitting within a pocket. Bell defined "smartphone" as the convergence between a personal media device (PDA), a camera and a cell phone (Bell, 2008). IBM Simon, introduced in 1992 at Las Vegas COMDEX, is often considered the first smartphone despite its limited battery life and bulky form.

Just as the PC revolution disrupted mainframe, minicomputer, and workstation markets, smartphone and cloud services are increasingly disrupting the PC market nowadays (Reed et al., 2022). The smartphone OS market is unequally shared between Android and iOS. There have been some developments to harness unused CPU cycles of smartphones (Jenviriyakul et al., 2019). However, their architecture - CPU, battery, cooling system - was not designed toward this goal.

1.3.3 Widely networked

Gray described the costs of computing (Gray, 2003): software and hardware companies sell billions of dollars of computers and software each year; the total cost of ownership is more than a trillion dollars per year. Operations cost far exceed capital costs. Due to physical and economic limitations, there has always been a large interest in sharing computing power from different computing resources, across institutions and organizations: (i) certain communities may temporarily need additional computing resources not available locally - not affordable - to process their workloads;

(ii) certain organizations may need to amortize their infrastructure costs - initial and maintenance - by providing them to several communities across the globe. Nevertheless, Gray questioned the economic issues of moving a task from one computer to another or from one place to another. According to him, "it is fine to send a GB over the network if it saves years of computation - but it is not economic to send a kilobyte question if the answer could be computed locally in a second" (Gray, 2003). The author recommended putting the computation near the data, and stated that "on-demand" computing was only viable for very CPU-intensive tasks; which is still the case but to a lesser extent since years of networking progress have passed. This factor has been essential in the expansion of several approaches to temporarily acquire/offer remote computing resources.

Grid Computing

In the mid-1990s, Foster and Kesselman, inspired by the electricity Grid, coined the same term "Grid" to denote a distributed computing infrastructure for advanced science and engineering (Foster and Kesselman, 2003). Grid computing aims at coordinating resource sharing and problem-solving in dynamic, multi-institutional Virtual Organization (VO) (Foster et al., 2001). It does not imply unrestricted access to resources, resource providers and consumers are expected to clearly and carefully define what is shared, who is allowed to share, and the conditions under which sharing occurs. Such a set of individuals and/or institutions is defined as VO.

Grid computing can be seen as a set of additional protocols and services built on the Internet to support the creation and use of computation-enriched environments. These include: (i) resource management protocols and services that support secure remote access to computing and data resources and the co-allocation of multiple resources; (ii) security solutions that support the management of credentials and policies when computations span multiple institutions; (iii) information query protocols and services that provide configuration, monitoring and status information about resources, organizations and services; (iv) data management services that locate and transport datasets between storage systems and applications (Foster et al., 2001). For Foster and Kesselman, interoperability is essential to provide fair sharing arrangement and dynamic VO formations, and should rely on standard and universal protocols and syntax. Grid computing architecture is deliberately open rather than prescriptive.

Computing resources architectures, operating systems, scheduling and usage policies are not imposed, which can lead to administrative and physical heterogeneities, and task reproducibility issues. Though, the authors recommend obtaining agreement on standard protocols. Additionally, grid environments introduce challenges that are not encountered in sequential or parallel computers: multiple administrative domains, new failure modes and large variations in performance. Foster and Kesselman designed the Globus Infrastructure Toolkit (Foster and Kesselman, 1997) to build an adaptive wide-area resource environment. It provides basic mechanisms such as communication, authentication, network information and data access, enabling high-level applications to adapt to heterogeneous and dynamically changing environments.

Grid computing enables access to millions of processors within a VO. Users have mostly access to clusters, workstations and supercomputers, via commodity hardware such as PCs and smartphones. The dynamic and unpredictable behavior of grid computing is not adapted to HPC workloads that require tightly coupled computing resources such as supercomputers, but several scientific communities dealing with HTC tasks have largely adopted the approach. In practice, grid computing tends to involve organizationally-owned resources managed by professional administrators and powered on most of the time.

The business model of Grid Computing - mainly found in academia and laboratories - is project-oriented (Foster et al., 2008): VOs typically have a certain number of CPU hours they can spend. Access to computational power requires increasingly complex proposals to be written. When a VO joins a grid infrastructure with a set of resources, it knows that others in the community can now use these resources. In the meantime, it acknowledges the fact that it gains access to other sites. Popular grid computing infrastructures involve WLCG (“Worldwide LHC Computing Grid”, 2022), which partners with the European Grid Infrastructure (EGI), the Open Science Grid (OSG) and the Nordic e-Infrastructure Collaboration (NeIC).

Cloud Computing

Cloud computing emerged in the 2000s and evolved out of grid computing to become a tremendously influential paradigm for the whole computing area. NIST denotes cloud computing as a "model for enabling ubiquitous, convenient, on-demand network

access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell, Grance, et al., 2011).

According to Mell et al., cloud computing is characterized by: (i) on-demand self-service, a consumer can unilaterally provision computing capabilities without requiring human interaction; (ii) broad network access, capabilities are available over the network and accessed through heterogeneous client platforms such as workstations, personal computers, smartphones; (iii) resource pooling, provider's computing resources, physical or virtual, can serve multiple consumers; (iv) rapid elasticity, capabilities can be elastically provisioned and released, and appeared as unlimited; (v) measured service, cloud systems control and optimize resources, and provide transparency for both the provider and consumer of the utilized service. The capability provided to the consumer may vary depending on the service model chosen by the provider, the most known are: (i) software as a service (SaaS) allows using the provider's application running on a cloud infrastructure; (ii) platform as a service (PaaS) enables deployment of consumer-created or acquired applications, supported by the provider, onto the cloud infrastructure ; (iii) infrastructure as a service (IaaS) allows provisioning processing, storage, networks to deploy and run arbitrary software, OS and applications. In the context of this thesis, we focus on IaaS. Cloud infrastructures can be private, for exclusive use by a single organization comprising multiple consumers. It may be owned, managed and operated by the organization or a third party. "Community cloud" is also widely spread, and opened to specific communities of consumers from organizations that have shared concerns. "Public cloud" is the most known form of infrastructure because it is open to the general public. Most known actors in the domain include Amazon Web Service (AWS), Google Cloud and Microsoft Azure. Lastly, "hybrid cloud" is a composition of two or more distinct cloud infrastructures that remain unique entities, but are bound together by standardized or proprietary technologies.

Foster et al. describe factors that contribute to the surge and interest in cloud computing (Foster et al., 2008): (i) the rapid decrease in hardware cost and increase in computing power and storage capacity; (ii) the exponentially growing data size; (iii) the wide-spread adoption of web 2.0 applications. The evolution of virtualization technology has also been essential in the expansion of cloud computing. Over the past

few years, processor manufacturers such as AMD and Intel have introduced hardware support for virtualization (Foster et al., 2008). Cloud providers have largely embraced virtualization (Foster et al., 2008): (i) multiple applications can be run on the same server, resources can be utilized more efficiently; (ii) resources can be dynamically configured and bundled for significantly different compute and storage needs; (iii) virtual environments can be backed up, migrated and recovered; (iv) provisioning and maintenance can be automated. Virtualization also provides better security, manageability and isolation. Computing infrastructures are much better utilized, leading to lower upfront and operational costs.

Cloud computing dramatically lowers the cost of entry for small firms and third-world countries trying to benefit from compute-intensive workflows that were only accessible to the largest of corporations (Marston et al., 2011). According to Marston et al., cloud computing also enables faster time to market in many businesses, promotes innovation, and makes it easier for enterprises to scale their services depending on client demand. Even though small businesses have been quick to adapt to cloud computing, large corporations have voiced a plethora of concerns, clearly enumerated by Armbrust et al. (Armbrust et al., 2010).

Security is one of the major concerns, as the responsibility is divided among potentially many parties including cloud users, vendors, and any third-party vendors that users rely on for security-sensitive software or configurations. Data in the cloud face security threats both from outside and inside the cloud. Virtualization is a key security lever protecting against most attempts by users to attack one another or the underlying cloud infrastructure. One of the last security concern is about protecting the cloud user against the provider. Users mainly use contracts and courts, rather than security engineering, to guard against provider malfeasance.

Data lock-in and software licensing are other obstacles. Current software licenses commonly restrict the computers on which the software can run, whereas cloud computing APIs are still essentially proprietary, which makes it difficult to extract software and data from the cloud. Additionally, users are vulnerable to price increases, reliability problems - the OVH data center fire is a recent example -, and even to providers going out of business. Interoperability of all the services and applications remains a significant challenge, as users need to tap into a federation of Clouds instead of a single Cloud provider. Standardization appears to be a good solution to address

the interoperability issue. However, the interoperability issue "has not appeared on the pressing agenda of major industry cloud vendors" (Dillon et al., 2010). None of them supported the Unified Cloud Interface project (“Unified Cloud Interface Project”, 2012) proposed by the Cloud Computing Interoperability Forum (CCIF), or the Open Cloud Manifesto - mentioned by Nelson et al. (Nelson, 2009) - supported by almost 200 companies and organizations. In 2017, the National Institute of Standards and Technology (NIST) and the Institute of Electrical and Electronic Engineers (IEEE) announced a collaboration to build consensus on creating an inter-cloud (Lee et al., 2020).

Cloud computing was first attracting web services and HTC workloads. A few years ago, Napper et al. demonstrated that cloud resources could not reach a spot in the Top500. Indeed, the performance of single nodes available on AWS EC2 was as good as nodes found in current HPC systems, but the available memory and network performance were insufficient to maintain high performance when scaling up the cluster (Napper and Bientinesi, 2009). Nevertheless, major cloud vendors are proposing HPC resources nowadays.

The rapid growth of commercial cloud services and business outsourcing has made Amazon AWS, Microsoft Azure, and Google Cloud among the fastest-growing elements of the computing services industry (Reed et al., 2022). Today, CPU manufacturers are responsive to cloud vendor requirements, as a large part of their microprocessors is purchased by those vendors. In the same way, business and technology shifts have attracted money and opportunities, and therefore, talents from academia to a small number of very large companies or creative startups.

Global Computing

The key idea of global computing - also named volunteering computing or public-resource computing - is to "harvest the idle time of Internet-connected computing resources which may be widely distributed across the world, to run a very large and distributed application" (Fedak et al., 2001). Global computing relies on volunteers, willing to offer some of their idle time to execute tasks, mostly related to academic or public projects, in exchange for incentives such as credit and screensaver graphics. The goal of global computing is to harness a large number of unused computing resources, PCs and smartphones, to build a very large parallel computer. According to

Anderson, it allows research scientists with moderate computing skills to create and operate a large computing project and also encourages public awareness of current scientific research (Anderson, 2004).

Global computing emerged in the mid-1990s with GIMPS, Distributed.net and SETI@home Korpela et al., 2001. Open-source middleware such as XtremWeb (Fedak et al., 2001) and Berkeley Open Infrastructure for Network Computing (BOINC, Anderson, 2004) have been essential in the expansion of large-scale public resource projects. They make it easy for scientists to create and operate public-resource computing projects and for PC owners to participate in multiple projects.

In contrast to grid and cloud computing, global computing involves an asymmetric relationship between projects and participants. Projects have no control over participants, and cannot prevent malicious behavior. Additionally, computers can embed various OS and CPUs, and are frequently turned off or disconnected from the Internet. Global computing middleware typically include redundant computing and cheat-resistant accounting mechanisms to address these issues. Most of the projects supported are throughput-oriented and have relatively small memory, disk and network bandwidth requirements.

Nowadays, the cloud market is dominated by few providers and has been developed without standards, which limits medium to small providers to enter the market (Romero Coronado and Altmann, 2017). With the emergence of the blockchain technology (Nakamoto, 2008) and the smart contracts (Szabo, 1996), several projects such as iExec, Golem and SONM have intended to build decentralized cloud infrastructure based on the global computing paradigm. Uriarte et al. provide a complete comparison of these projects, and emphasize the lack of generality and interoperability between them (Uriarte and DeNicola, 2018).

Through Section 1.3, we have had a brief overview of the current computing landscape and how it has evolved. These advancements have greatly enhanced data management and processing at CERN and fostered significant scientific progress such as the discovery of the Higgs Boson in 2012 (A. Collaboration, 2012). CERN signed the purchasing contract for its first mainframe computer - a Ferranti Mercury - in 1956. In 1996, it turned off the IBM 3090 for the last time and replaced them by scalable solutions based on Unix PCs to comply with LHC computing requirements. In Section

1.4, we are going to describe the current LHCb workflow and available resources and identify the computing needs of the collaboration.

1.4 LHCb workflows and available computing resources

1.4.1 LHCb workflow

At the origins of the universe, matter and antimatter were created in equal proportions. However, a growing imbalance has appeared over time that should result in the loss of antimatter. The purpose of the LHCb experiment lies in the study of beauty and charm hadron decays to provide insight into the phenomenon of matter-antimatter asymmetries (T. L. Collaboration, 2008; "LHCb - Large Hadron Collider beauty experiment", 2022). The experiment also searches for physics beyond the Standard Model through rare decays. In the following sections, we will present the LHCb detector as well as the computing model of the experiment (Section 1.4.1) and will enumerate the computing resources at the disposal of the collaboration (Section 1.4.2). It is worth mentioning that the LHC Run3 has started in 2022 and involves an upgraded version of the LHCb detector, a different computing model as well as novel applications. Since these applications currently remain unstable and experimental, we focused on the original detector components and the computing model from Run1 and Run2.

Detecting collisions

Collisions occurring in the LHC result in an abundance of quarks that quickly decay into other forms. To catch beauty quarks, LHCb has developed a series of sophisticated sub-detectors, close to the path of the beams circling in the LHC. The ability to identify different particle species is a fundamental requirement for the success of the experiment. The detector geometry is optimized to detect forward events efficiently (Figure 1.1):

- The *VErtex LOcator* (VELO), a silicon-strip vertex detector, is positioned at five millimeters from the proton-proton interactions to pick out short-lived B mesons from the multitude of other particles produced.
- The Ring Imaging Cherenkov (RICH1 and RICH2) detectors identify a range of different particles resulting from the decay of B mesons, such as charged pions,

kaons and protons. Charged particles pass through a dense gas, faster than light does, and emit a cone of light, which the RICH detectors reflect onto an array of sensors using mirrors.

- The tracking system comprises four tracking stations: TT located between RICH1 and the dipole magnet, and T1-T3 between the magnet and RICH2. They enable the trajectory of each particle passing through the detector to be recorded, and Cherenkov rings to be reconstructed.

- The magnet consists of two coils of 27 tonnes, mounted inside a 1,450-tonne steel frame. It causes the paths of charged particles to curve, with positive and negative particles moving in opposite directions. It helps scientists calculate the momentum of a particle and find its identity.

- The muon system contains five rectangular stations, gradually increasing in size (M1-M5). Each station contains chambers filled with a combination of three gases, with which passing muons - tiny, electron-like particles present in the final stages of many B-meson decays - react while wire electrodes detect the result.

- The calorimeter system (SPD, PS, ECAL, HCAL) stops particles, measuring the amount of energy lost as each one grinds to a halt. The Scintillating Pad Detector (SPD) determines whether particles hitting the calorimeter system are charged or neutral, while the Pre-Shower detector (PS) indicates the electromagnetic character of the particle. The electromagnetic calorimeter (ECAL) deals with the energy of lighter particles, such as electrons and photons, whereas the hadron calorimeter (HCAL) samples the energy of protons, neutrons and other particles containing quarks. Calorimeters rely on ultraviolet light, which is emitted proportionally to the energy of the particles. They remain the main way of identifying particles that possess no electrical charge.

Eventually, the upgraded detector should be able to support an increased luminosity coming from the LHC (Piucci, 2017). It is going to embed, once completed, a new tracking system composed of three sub-detectors: a new VELO, an Upstream Tracker (UT) and a large Scintillating-Fiber (SciFi) tracker. The upgraded VELO, already installed, is closer to the beam axis. This should improve the impact parameter resolution by a factor of about 40%, increase the VELO tracking, efficiency especially

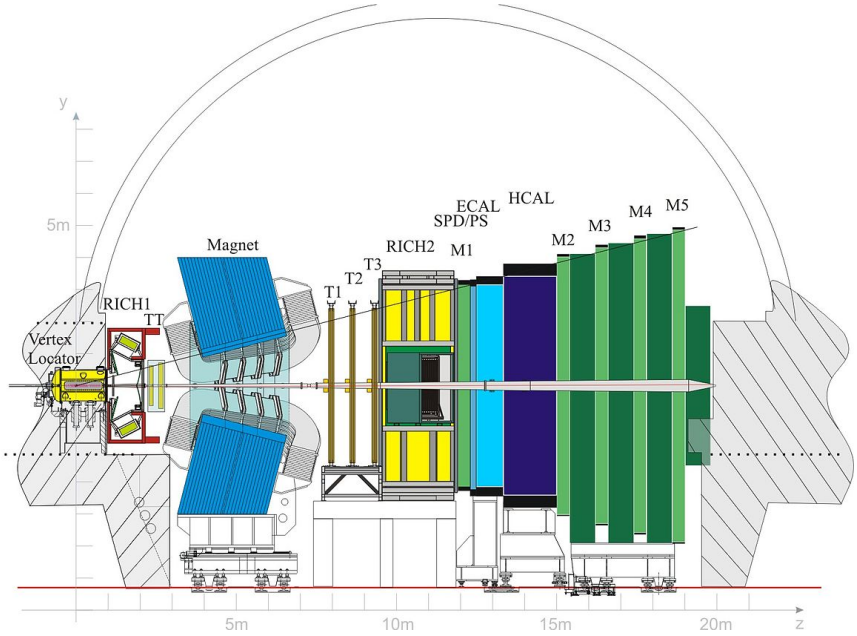

Figure 1.1 – A schematic view of the original LHCb detector.

for low momentum tracks, and provide a better decay time resolution. The UT should be fully installed in 2023. It comprises four tracking layers based on silicon strip technology that will be located between the RICH1 detector and the dipole magnet, in place of the current TT. The UT will be used for downstream reconstruction of long-lived particles decaying after the VELO and will be essential to improve the trigger timing and the momentum resolution. The SciFi tracker, already in place, is located between the dipole magnet and the RICH2 detector. It is structured in 12 detector layers and used for track reconstruction after the magnet region. The system will allow reducing the number of fake tracks reconstructed by the tracking algorithms, by a factor of 50-70%. Consequently, the trigger timing will be largely improved. In addition, the RICH detectors have been upgraded with new photo sensors and front-end electronics. The physics performance of the new RICH system will achieve similar performance to the previous detector but at a ten-fold higher luminosity. We can also mention minor upgrades in (i) the calorimeter system where the SPD and the PS have been removed and the readout electronics of the ECAL and HCAL have been replaced; (ii) the muon system where the off-detector electronics have been redesigned and the M1 has been removed (Mnich, 2022).

The original LHCb detector records 40 million collisions every second, which represents an amount of data too large to be processed and stored, namely 1 TB/s or

3.6 PB per hour. Moreover, physicists cannot directly work on raw data coming from the detector, it has to be filtered, converted into a human-readable format and stored. This requires HEP software and significant computing power.

The LHCb computing model

Filtering, processing and analysis are part of a complex workflow represented in Figure 1.2. During Run1 and Run2, LHCb used a two-level trigger system for only selecting the most interesting collisions from the LHC: the level 0 trigger (L0), and the high-level trigger (HLT). L0 is implemented in custom FPGAs and aims at reducing the rate of visible interactions from 40 MHz to 1 MHz at which the LHCb detector can be read out. L0 takes decisions in under 4 μs, exclusively based on information from the muon and the calorimeter systems, as these are the only piece of information available at such a high rate. HLT, a software implementation (Moore), further filters collision to reach an output rate of 30 kHz. HLT is a two-stage system: HLT1 performs a fast track reconstruction and makes a decision based on a track segment; HLT2 performs a high-fidelity reconstruction and makes a decision based on the full detector read-out information. The data is recorded to RAW files and transferred to tapes. These operations are defined as the *online* part of the experiment. They are followed by a second set of operations called the *offline* part of the experiment, which is mainly orchestrated by LHCbDIRAC (Hushchyn et al., 2017).

After the replication, the RAW data is reconstructed, via the Brunel application, to transform the detector hits into objects such as tracks and clusters. Objects are stored in an output file in the form of a Data Summary Tape (DST) file. A DST file contains the full event information: raw data and reconstructed objects; which corresponds to 150 kB for each event.

This data goes through the stripping stage (DaVinci) consisting of very severe selections which correspond to different types of physics analysis that keep interesting events and generate DST or μDST files - save space by storing only the information concerning the build candidates, raw events are discarded. To save disk space and speed up access for analysts, these files are grouped into streams of 5 GB containing similar selections. Users can run their analysis tools to extract variables from DST and μDST files using the DaVinci applications. μDST files require additional calculations as some tracks are not present in the events.

Stochastic simulation applications are of major importance both in the design and construction phase of an experiment and during its operation. They allow the collaboration to understand experimental conditions and performances. LHCb production managers generate a lot of simulated events, often called Monte Carlo data to distinguish them from real data. They are processed in a very similar way to real data. Indeed, the simulated data is subject to the same deficiencies as in the processing of real data.

The simulation of proton-proton collisions, and the hadronization and decay of the resulting particles, are controlled by the Gauss application. The Bool application converts the simulated hits made in the virtual detector into signals that mimic the real detector, so that simulated data can then be passed through the usual data processing chain described above.

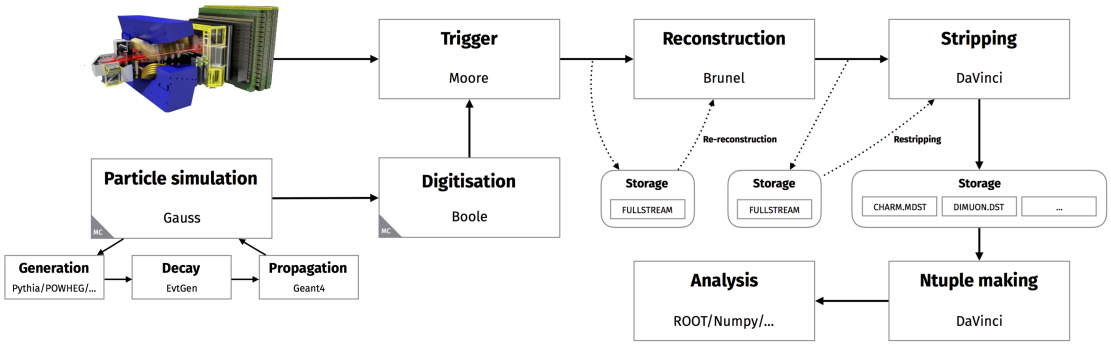

Figure 1.2 – LHCb computing workflow (Run1).

In 2015, LHCb introduced the *Turbo stream* to respond to the increase in energy coming from the Run2 of the LHC, and thus, the higher rate of interesting events. The *Turbo stream* directly provides users with the selection of candidates resulting from HLT2, with no further offline reconstruction by Brunel. The collaboration improved the reconstruction software both online (Moore) and offline (Brunel), between the first two runs, which now perform identically. However, because of limited storage, events saved to the *Turbo stream* contain only the candidates that were reconstructed in the trigger, and therefore the regular workflow is still necessary and runs in parallel. Overall, *Turbo Stream* saves a lot of time, and hence money as users do not always need to wait for the offline reconstruction to perform their analysis.

In 2022, the collaboration has fully redesigned the trigger system: L0 hardware trigger has been considered a bottleneck, in regards to the increase in luminosity

coming from the Run3, and has been removed. HLT1, which handles the track reconstruction, an inherently parallelizable task suitable for vector processing, has been reimplemented on GPUs (Allen framework). HLT1 will reduce the data rate by a factor of 30. In the same way, the collaboration is planning to increase the use of the *Turbo steam* model. For about 2/3 of data, only the data of the signal candidate will be saved and no further offline reconstruction will be possible. This should generate smaller events so that more events could be saved. In the context of this thesis, we focus on offline activities, which require a huge amount of computing power not available at CERN.

Offline and distributed computing activities

The LHCb offline and distributed computing activities include production and non-production tasks (Stagni and Charpentier, 2012). Non-production activities include user analysis, monitoring and testing, and take the form of independent tasks, not contributing to any shared outcome. Production activities comprise simulation, digitalization, reconstruction, reprocessing, stripping and analysis.

In LHCb, application managers design usable steps via the LHCbDIRAC Production Request System. A step corresponds to an activity definition: an offline application bound to a set of options and parameters, applied on the collection of events provided, or to be generated. Then, the production managers combine the available steps to create and start productions. A production can be seen as a linear meta-workflow, namely a sequence of production activities. Production managers, along with the computing shifters and the Grid Expert On Call (GEOC) follow the productions.

Once started, steps take the form of Bags of Tasks, batches of independent tasks where only the collective outcome is relevant. Each task performs an event data processing application, provided in the step definition, on a subset of events defined in the production. LHCb event data processing applications are based on the Gaudi framework (Barrand et al., 2001). Gaudi provides a common infrastructure and environment to interact with events: (i) a global *Event Loop* to process events one by one without holding them all in memory at once; (ii) a *Transient Event Store*, a per-event file system, gathering data objects - particles, vertices, tracks, hits - related to a single event; (iii) *Algorithms*, C++ class that can be inserted into the *Event Loop* to perform a

certain function for each event, such as filtering events or reconstructing particles; (iv) *Tools*, common functions shared between *Algorithms*; (v) *Options*, a script in which properties of *Algorithms* and *Tools* are specified, along with the order of execution of the *Algorithms*.

Because CERN cannot handle all the activities on-site, tasks are encapsulated into DIRAC jobs, distributed and remotely executed. The data manager creates bulk replication, deletion or archival of datasets for the jobs. A DIRAC job may provide information about the execution location and environment requirements: platform, number of processors, and minimum CPU time. A DIRAC job embeds its task in a linear workflow of connected operations to pre-process and post-process the task. A DIRAC workflow is composed of steps, each of them including a number of modules connected to Python modules.

In practice, launching a production triggers the creation of transformations in the Transformation System. A transformation handles repetitive work such as creating production jobs and data management operations. Transformation agents are responsible for inspecting the Transformation System tables, and submitting tasks either to the Workload Management System, or the Data Management System.

Figure 1.3 presents the number of offline tasks processed and the CPU days consumed by these tasks in a year. Simulation applications (Gauss), noted *Sim* and *FastSim* on the plots, represent 71.7% of the offline activities. In a year, they consume around 91.1% of the CPU time available on computing resources. They are, by far, the largest consumers of the available computing resources: 91.1% of the CPU time available is dedicated to the simulations, and should remain above 90% in Run3 (Stagni et al., 2020).

Focus on Gauss

Gauss (Belyaev et al., 2011; Clemencic et al., 2011) consists of two independent phases executed sequentially: (i) the generation of the events; (ii) the tracking of the particles through the simulated detector. The production of particles is handled with Pythia (Sjöstrand et al., 2001), a general-purpose event generator, whilst the decay and time evolution of the produced particles is delegated to EvtGen (Lange, 2001). External generator libraries - *Tools* - can be plugged to perform specific actions: generation of

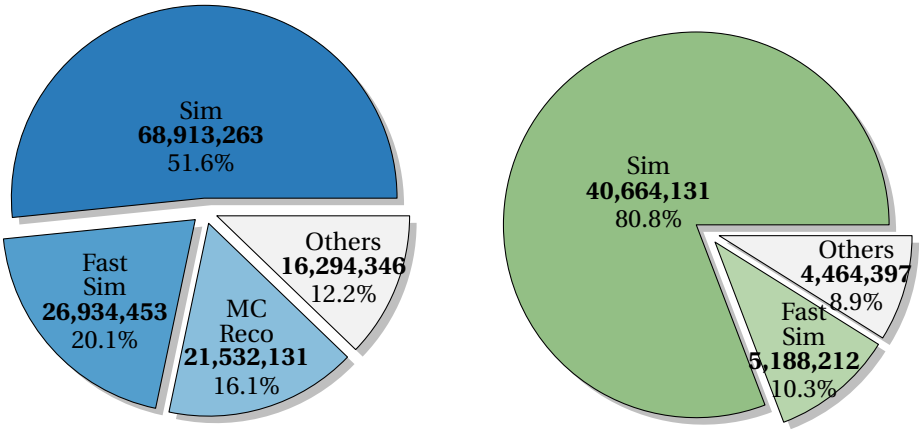

Figure 1.3 – At the left: number of tasks processed; At the right: CPU days consumed. Both are classified by type of task, from week 24 of 2021 to week 24 of 2022. Generated from the LHCbDIRAC web application.

a given event sample, decay of unstable particles, cut at generator level on the decay. The simulation of the physics processes undergone by the particles traveling through the detector is delegated to the Geant4 toolkit (Agostinelli et al., 2003). A dedicated Gauss algorithm transforms the output of the generation phase to the Geant4 input format.

Gauss does not require any data input, except an *Option* script containing the configuration and initialization variables for each phase. Gauss is highly configurable and involves different Pseudo-Random Number Generators (PRNG) in the different phases. We can pinpoint Marsaglia-Zaman-Tsang (Marsaglia et al., 1990), a PRNG from the 1990s embedded in Pythia. It is worth mentioning that Marsaglia's PRNGs were criticized by Panneton and L'ecuyer for their poor quality (Panneton and L'ecuyer, 2005), and that they fail several randomness tests from the TestU01 library, a collection of utilities for the empirical randomness testing of random number generators (L'ecuyer and Simard, 2007).

A random number seed is generated based on a run number, an event number as well as an algorithm name. The random number seed is reset for each event both at the beginning of the generation and the tracking phases. The run and event numbers are stored in the generated event and read back when initializing the tracking phase to ensure the event reproducibility. Two common ways are available to run the application: (i) a generator-only mode; (ii) a full simulation mode. In the full simulation mode, the generator output is provided along with the output of Geant4

transformed back into the LHCb event model, namely a Monte Carlo truth history as well as hits produced in the sensitive detectors.

Currently, Gauss is a compute-intensive single-process (SP), single-threaded (ST) application, with a memory footprint of 1.4 GB. From 95% to 99% of the CPU time devoted to the application is spent by Geant4. Gauss only supports CISC x86 architectures and CERN-CentOS-compatible environments ("Linux@CERN", 2022).

With the upgrade of the LHCb detector for the LHC Run3, the detector will collect a substantially larger amount of data, which will result in a bigger amount of complex simulated events. Therefore, speeding up simulations is becoming essential to meet future simulation demands. Stagni et al. created Gauss-MP, a multi-process (MP) version of Gauss (Stagni et al., 2020) based on GaudiMP (Rauschmayr and Streit, 2014) to run Gauss on multiple hardware threads. GaudiMP is a MP version of the LHCb event processing framework Gaudi, which already existed but had never been used in production. To greatly reduce the memory footprint of Gauss, the LHCb team has started to work on Gauss-on-Gaussino (Siddi and Müller, 2019), a multi-threaded (MT) version of Gauss. Gauss-on-Gaussino is still in development.

Faster alternatives to a full, Geant4-based simulation are also being pursued. LHCb developers are investigating libraries to: (i) simulate fewer particles; (ii) simulate less of the detector; (iii) simulate the subdetectors faster. For instance, Siddi has worked on the integration of the Delphes toolkit (Ovyn et al., 2009) in the simulation framework, replacing Geant4 (Siddi, 2016). Delphes is modular software designed to perform fast simulations by propagating stable particles using a parametric approach. We can also cite libraries such as ReDecay (Müller et al., 2018), RICHLess (Whitehead, 2018), and Lamarr (L. Collaboration, 2019), currently being developed. In the meantime, it remains critical to find computing resources to respond to the increasing CPU demand.

1.4.2 Computing Resources available

Figure 1.4 highlights the number of CPU days consumed by type of resources in 2019. The LHCb experiment mainly uses WLCG to perform its offline activities (57.9%), followed by the HLT farm (38.6%) and other opportunistic resources, clouds and partition of supercomputers.

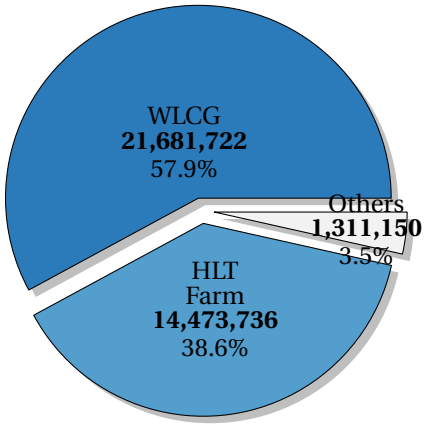

Figure 1.4 – Number of CPU days consumed, classified by type of resource, from week 0 of 2019 to week 0 of 2020. Generated from the LHCbDIRAC web application.

The Wordwide LHC Computing Grid

Created in 2002 to share the burden with computer centers around the world, WLCG is the world's largest computing grid. Coordinated by CERN, WLCG currently comprises 170 computing centers, 1 million computed cores and 1 exabyte of storage, spread within 42 countries and supported by many associated grids across the world, such as EGI and OSG as well as many regional grids. It can process 2 million tasks every day and gives a community of over 10,000 physicists near real-time access to LHC data, regardless of their physical location. WLCG is organized in different layers:

- Tier 0: the CERN Data Center located in Geneva, Switzerland. It is responsible for the safe-keeping of the raw data and the first pass reconstruction. It has to distribute the raw data and reconstruction output to the Tier 1s and reprocess data during LHC down-times as well.

- Tier 1: 13 sites responsible for the safe-keeping of proportional share of raw and reconstructed data, large-scale reprocessing and safe-keeping of corresponding output. They have to distribute the data to Tier 2s and safely keep a share of simulated data produced at these Tier 2s.

- Tier 2: universities and other scientific institutes, which can store sufficient data and provide adequate computing power for specific analysis tasks. They handle analysis requirements and proportional share of simulated event production and reconstruction.

- Tier 3: individual scientists accessing these facilities through local computing resources, which can consist of local clusters or even just PCs.

To be part of Tier 1 and Tier 2, sites need to comply with the resource and service requirements of WLCG and the experiments. Tier 1 sites have to provide a detailed plan that explains which experiments will be hosted and how they will demonstrate the expected functionality, performance and reliability. They should have excellent connectivity to the national academic network backbone. Tier 1 and Tier 2 sites should provide a significant fraction of CPU and disk capacity of that required by each experiment: typically around 10% of the global total Tier 1 requirement of the experiment for Tier 1 sites. Tier 1 sites also have to provide a significant tape archive service. In addition to resources, sites must run the set of services for the target experiments. They should provide on-call support for key services year-round, and demonstrate the appropriate level of availability and reliability of the services. They must provide the interface to the WLCG accounting services and publish accounting data on all of the work performed by the supported experiments. Computing resources of WLCG can embed various types of CISC x86 CPUs, and support CentOS, an environment adapted to LHCb software.

Each year, review boards have to compare the requests of the experiments for computing and storage resources with the actual usage of the available resources. The overall computing power comes from a variety of CPUs, where the specific hardware deployed at one site can be quite different from that at another site, both in terms of cost and computing performance (Valassi, Andrea et al., 2020). To quantify the experiment needs and the resources provided by the sites in a given year, a common unit of measurement is required. Since 2009, the HEP-SPEC06 (HS06) has been the standard CPU benchmark for the LHC experiments (“How to Run HEP-SPEC06 Benchmark”, 2017). Until now, it provides a quite good evaluation metric of a computing resource, highly correlated to HEP applications: the amount of useful "work" that the computing resource can do per unit time. Based on the actual usage of the resources of LHCb in a year, review boards allocate a certain number of CPU work, in HS06 units, for the next year. Sites also use HS06 to buy the CPU resources providing the amount of HS06 pledged to the experiments for the lowest financial cost, also taking into account electrical power efficiency (Valassi, Andrea et al., 2020).

This approach was reliable during LHC Run1, but LHC has produced a growing

amount of data since then. According to the analysis of Stagni et al. on the CPU cycles use in 2016, all the LHC experiments now consume more CPU hours than those officially pledged to them by the WLCG (Stagni et al., 2017). Experiments have found ways to exploit opportunistic resources.

The HLT Farm

Located near the LHCb experiment at the CERN Point 8, The High-Level Trigger (HLT) farm is dedicated to the HLT1 and HLT2 tasks. It is composed of 1500 PCs, distributed over 60 subfarms: each of them comprises 24, 28 or 32 PCs. The number of computers represents around 12,000 cores, idle when there is no data-taking. Recently, the LHCb collaboration decided to use these idle cycles to perform simulation tasks (Closier et al., 2019). The facility provides the environment and set of services required to run simulation tasks.

Supercomputers

Supercomputers offer a significant amount of resources and would potentially offer a more cost-effective data processing infrastructure compared to dedicated resources in the form of commodity clusters, as Sciacca emphasizes (Sciacca, 2020). Nevertheless, supercomputers are highly heterogeneous architecture, pose higher integration challenges and have not been operated continuously for LHC experiments workload processing. HEP workloads are not necessarily tailored for computing resources such as non-x86 CPUs and accelerators - GPUs and FPGAs - that we usually find in supercomputers. Furthermore, supercomputers are designed with a fast and expensive interconnect that would not be leveraged by a massive number of embarrassingly parallel tasks.

In the last few years, LHC experiments have started to integrate some of their workflows on compatible supercomputer partitions. CERN has an agreement with the Partnership for Advanced Computing in Europe (“PRACE”, 2022) and the pan-European network and services provider for research and education (“GÉANT”, 2022) to form a pioneering collaboration that will work to overcome challenges related to the use of HPC systems. This collaboration, now powered by the EuroHPC initiative, allows LHCb to access some of the best pre-exascale and petascale supercomputers

located in the European Union: Mare Nostrum 4 in Spain, Marconi in Italy and Piz Daint in Switzerland. In this context, a given number of CPU hours are reserved for the experiment. Once consumed, LHCb jobs are still allowed in the systems, but are less prioritized. LHCb also benefits from a close collaboration with the Laboratório Nacional de Computação Científica (LNCC) in Brazil, which allows the experiment to use some of the CPU resources of the Santos Dumont supercomputer.

1.5 Conclusion

This survey has covered parts of the history and latest trends of the computing industry and research. The 2020s should be marked by (i) further heterogeneous computing resources for specialized and complex workloads; (ii) a growing demand for cloud computing, especially related to HPC; (iii) a decline in Moore's law. While it is mainly LHCb-oriented, this chapter provides details about how to analyze the impact of the current trends on a computing project.

We have started this chapter by defining terms such as job, task, workload and workflow in order to set common foundations for the rest of the thesis (Section 1.2.1). Based on these, we have compared three main computing paradigms, namely HPC, HTC and MTC (Section 1.2.2). Combined with physical and economical constraints, they have influenced hardware infrastructures.

Then, we have presented a variety of hardware components to highlight a growing heterogeneity (Section 1.3.1):

- ISAs: RISC and CISC.
- Accelerators: GPUs, MICs, FPGAs and ASICs.
- Storage and memory: HDD, SSD, SRAM, DDR SDRAM.
- I/O devices and especially the network interfaces: Wi-Fi, Ethernet, Infiniband.
- OS: Windows, Mac, Linux.
- Programming languages: interpreted and compiled.

We also surveyed the main computing classes (Section 1.3.2). While mainframes are still used in large IT organizations, many of them have been replaced by clusters

of computing resources, often more affordable. Currently, cluster architectures also represent the majority of supercomputers. On the user side, the smartphone is increasingly disrupting the PC market, even though PCs are still largely employed in companies and laboratories.

Following the progress in networking, resource providers have been able to offer access to their computing power via grid computing, cloud computing and global computing (Section 1.3.3). During the last decade, cloud computing has been largely adopted. It is worth noting that, contrary to grid computing, cloud computing does not rely on any standard protocols.

We have finished with an analysis of the LHCb workload and available computing resources (Section 1.4). LHCb workloads are primarily composed of Monte-Carlo simulation tasks that represent 71.7% of the offline activities and consume 91.1% of the CPU time available. These amounts are expected to grow with the next LHC runs. While many developers are working on reducing the memory and CPU footprint of the programs, there is still a need for additional computing power. LHCb decision-makers have chosen to rely on WLCG and several supercomputers to deal with the tasks.

Thus, the aim is to efficiently execute as many Gauss tasks as possible on these computing resources, despite various integration challenges. We address the topic in Chapter 2 where we have surveyed different solutions to better use distributed and heterogeneous computing resources, and especially within grids and supercomputers, to process BoT workloads with limited inputs.

2 Running tasks on shared, time-constrained, distributed and heterogeneous computing resources

2.1 Introduction

As workloads become more complex, hardware tends to be more specific, expensive and difficult to acquire. Cutting-edge workloads require massive infrastructures that only large companies and laboratories are able to provide. Furthermore, such hardware is rare and highly demanded.

Resource providers have set up mechanisms to offer a fair share of their infrastructures to different consumers. While they grant resource computing allocations on their systems, they cannot guarantee the repeatability of programs developed on different systems. They may not supply allocations with dependencies of the programs or the required OS for instance. Thus, there exist intermediate developers creating middleware to ease the integration of workloads on heterogeneous and distributed computing resources coming from independent administrative sites.

In this chapter, we survey different middleware solutions, mostly related to grid computing and supercomputers, to run Bag-of-Tasks (BoT) with limited inputs on distributed and heterogeneous computing resources. First, we study the grid computing environment, the existing provisioning model and security mechanisms to protect the services of Workload Management System (WMS) (Section 2.2). Then, we compare VMs, containers, package managers and the CernVM-File System (CVMFS) to provide specific environments across various sites (Section 2.3). Lastly, we review tools to efficiently exploit granted allocations (Section 2.4).

2.2 Supplying computing resources with jobs

2.2.1 Environment

Temporarily acquiring shared, distributed and heterogeneous computing resources is the result of various middleware interacting with each other. Efficiently using them depends on how WMS leverage the middleware available to supply computing resources. In Section 2.2.1, we explain technical details and middleware implementations composing a computing grid, which we consider a meaningful use case to understand how standard protocols and collaboration can play a role in leveraging distributed, decentralized and heterogeneous computing systems.

Distributed Computing Resource and Infrastructure

Turilli et al. define a Distributed Computing Resource (DCR) as a set of possibly heterogeneous resources, a middleware, and an administrative domain (Turilli et al., 2018). Any system: (i) offering computing, data and network resources; (ii) proposing a middleware to ease the interactions with these resources; (iii) enforcing policies of an administrative domain; can be seen as a DCR.

A Distributed Computing Infrastructure (DCI), commonly named "Site" in this manuscript, is composed of a set of DCRs federated with a common administrative, project, or policy domain (Turilli et al., 2018).

On clusters and supercomputers, various communities continuously compete to acquire computing resources, while providers aim to maximize the use of their resources, and potentially their profit. To control access to computing resources, they generally set up a Local Resource Management System (LRMS), also defined as a batch system.

Local Resource Management System

A LRMS has three key functions: (i) allocating exclusive or non-exclusive access to resources - CPU cores, accelerators, memory - to users for some duration of time so they can perform their tasks; (ii) providing a framework to execute and monitor tasks; (iii) arbitrating contention for resources by managing a queue of pending work. It is worth noting that we find similar mechanisms in clouds. Even though resources appear infinite for the users, cloud providers still need to orchestrate their physical resources, making sure they can respond to the demand.

System administrators tend to provide different queues, or partitions, gathering similar Worker Nodes (WN), depending on user demands. A queue may accept and prioritize certain groups of users based on computing-related criteria, such as determining the best fit of jobs in memory or aiming toward maximum CPU usage (Henderson, 1995). According to Henderson et al., system administrators may also rely on policies driven by financial or political factors.

Users generally need to create jobs wrapping their tasks, inputs and meta-data. Meta-data are related to the duration of the tasks, and the number of cores and mem-

ory needed, to efficiently communicate with a LRMS. Therefore, the form of the jobs primarily depends on the underlying LRMS managing resources in the targeted system. There exists various LRMS solutions: PBS (Henderson, 1995), LSF ("IBM Platform LSF", 2022) and especially SLURM (Yoo et al., 2003) remain popular choices in the HPC community, while HTCondor (Litzkow et al., 1987) is largely employed in the HTC community, especially to harness idle CPU cycles on workstations. They have their programming syntax and configuration options, which can vary through time in the form of different versions. Communities having to interact with various infrastructures would need a deep understanding of each LRMS and its versions. Supporting each of them would result in a hard-to-maintain application. As a consequence, developers from different communities have worked on a component able to communicate with various types of LRMS: the Computing Element (CE).

Computing Element

Sfiligoi defines a CE as the "fundamental building block of any Grid environment" (Sfiligoi and Padhi, 2010). A CE provides a uniform interface across various sites, independently from the type of LRMS in place on the sites. The interface allows users to submit, monitor, cancel jobs or get their outputs. When a job is submitted, a globally unique job identifier is generally returned to further manipulate it afterward. A CE manages the burden of maintaining the code to support interactions with various versions of LRMS. In general, each site provides a CE to interact with a DCR.

Over time, different types of CE have also appeared. Similar to LRMS, types of CEs have their programming syntax and configuration options, varying through time. In 1998, Czajkowski et al. introduced the first CE (Czajkowski et al., 1998): the Globus Resource Allocation Manager (GRAM). GRAM was responsible for: (i) processing resource requests from users, under the form of Resource Specification Language (RSL) scripts; (ii) enabling remote monitoring and management of jobs created in response to a resource request; (iii) periodically updating information service with information about the current availability and capabilities of the resources managed. GRAM was originally able to communicate with HTCondor, EASY, Fork, LoadLeveler, LSF and NQE. GRAM was eventually declared deprecated and no longer used.

In 2010, the Computing Resource Execution And Management (CREAM), developed by INFN, became a popular choice among sites (Aiftimiei et al., 2008). It used a

completely different architecture than GRAM. It was based on open standards and exposed a simple, robust and lightweight interface based on Web Services. CREAM enabled a high degree of interoperability: developers could use any programming language to build client interfaces. CREAM processed Job Description Language (JDL) requests to get allocations. The JDL is a high-level, user-oriented language based on Condor classified advertisements (classads) (Litzkow et al., 1987). CREAM supported the execution of single-core and multi-core jobs, as well as MPI jobs. CREAM was no longer maintained in 2020, and thus, was declared deprecated. Instances were mostly replaced by HTCondor *schedd* instances. The LRMS proposes remote interactions with the scheduler, which can therefore be seen as a CE.

In 2002, the NorduGrid collaboration developed the Advanced Resource Connector (ARC) middleware to meet the pressing users' demands for urgent grid system deployment (Ellert et al., 2007). Because it resolved many shortcomings from the existing implementations at this time - HTCondor was not adjustable and extensible enough, while the Globus Toolkit required the number of ports opened in the firewall to be proportional to the number of potential jobs -, ARC quickly became one of the main choices for the sites. It has been the first-ever middleware to provide services to a worldwide High Energy Physics (HEP) data processing effort. ARC is portable, compact and manageable both on the server- and the client-side. It relies on an extended RSL (xRSL) syntax to handle jobs from users. Until now, ARC has embedded three main services: (i) the Computing Service, implemented as a GridFTP-Grid Manager, which provides a gateway to the computing resource; (ii) the Information System, a hierarchical database, which stores information about the list of known resources; (iii) the Brokering Client, deployed as a client part, enabling resource discovery and brokering capabilities. In ARC5 and ARC6, developers introduced A-REX services to replace both GridFTP services and the Grid Manager (Konstantinov, 2017). A-REX proposes a Web Service interface to interact with the jobs. Developers plan to drop support for GridFTP in the next release, namely ARC7.

We find similar solutions in the cloud domain, mostly to interact with APIs from different cloud providers. Apache libcloud allows users to uniformly access a variety of clouds ("Apache Libcloud", 2022). Supercomputers not supported within the Grid context are often only accessible via SSH, and sometimes through a VPN. In this case, users have to connect to the edge node of the infrastructure and interact locally with the LRMS in place. Dealing with many types and versions of CEs, supercomputers and

cloud resources would be time-consuming for the communities willing to submit their tasks on various infrastructures without having a deep understanding of middleware and CEs. For this reason, communities pass through a third layer to access remote computing resources: WMS.

WMS

The term "WMS" remains ambiguous: middleware such as ARC and HTCondor already support many computing paradigms and therefore, to some extent, could be considered as WMS. In the context of this thesis, a WMS provides an entry-point to manage jobs on various computing resources via different CEs and LRMS, across distributed sites.

Laure et al., developers of the gLite middleware, provide an overview of the main services accompanying WMS (Laure et al., 2004): (i) security services to enable identification of entities and functionalities for data confidentiality; (ii) API services to provide a convenient backend for grid services; (iii) information and monitoring services to publish and consume information related to resources and use it for monitoring purposes; (iv) data services to manage data across distributed sites through different transfer protocols if required.

There exist many different WMS. Many of them deal with a similar architecture composed of LRMS and CEs, but they focus on different kinds of computing resources or provide diverse means of handling jobs. Communities choose a WMS according to technical and emotional criteria such as: (i) the nature of their workload; (ii) the computing resources at their disposal; (iii) affinities and partnerships; (iv) the efforts deployed by developers to maintain the tool and communicate changes.

Many communities such as Belle II, CTA, EGI, iLC, JINR, Juno, and GridPP have adopted DIRAC to interact with grid resources. These communities have chosen DIRAC over similar solutions such as the Production ANd Distributed Analysis system (PanDA) WMS developed by ATLAS (Chiu and Potekhin, 2010), and the ALICE Environment (AliEN) (Bagnasco et al., 2008), because the middleware developed by LHCb quickly became generic, open-source, and self-contained. Recently, PanDA developers decided to offer PanDA features for projects outside of ATLAS and HEP experiments (Svirin et al., 2019). Contrary to other LHC experiments, the CMS ex-

periment has used an existing generic WMS solution named glideinWMS (Balcas et al., 2015; Sfiligoi, 2008). Most of the "grid" WMS mentioned above propose similar features, but depend on different funding agencies and are bound to distinct communities. They deal with BoT - rather than workflows like the Pegasus WMS (Deelman et al., 2015) - and grid resources, but remain more or less focused on integrating clouds and supercomputers. Developers of PanDA and glideinWMS started working on such opportunistic computing resources years ago, contrary to developers of DIRAC.

Communities bound to DIRAC have significantly helped to integrate cloud resources in the middleware, but a lot of work is still required for supercomputers. Contrary to the global computing area proposing self-contained and easy-to-integrate solutions such as the XtremWeb and the BOINC WMS, solutions related to supercomputers required an immediate development not grounded on any analytical understanding of underpinning abstractions. Therefore, they tend to remain specific to a WMS in a certain context. Thus, there are needs for: (i) classifying the existing solutions and proposing a first abstract model; (ii) building solutions related to supercomputers within DIRAC, generic, as far as possible. This will be treated in Chapter 4.

2.2.2 Provisioning model

Grid infrastructures tend to include various types of middleware and as many methods to interact with them. Developers of WMS have designed several approaches to provision distributed computing resources with jobs, around two main models which are going to be discussed in Section 2.2.2: push and pull. While straightforward methods consisting of directly submitting jobs are generic, they remain too often inefficient. Conversely, there exist very specific methods to provision cloud resources with jobs, non transferable to other types of computing resources.

Push model

The push-based approach consists in submitting jobs at many levels to dispatch them across the sites: (i) from a WMS to different CEs (Figure 2.1 Step 1); (ii) from a CE to a LRMS within sites geographically distributed (Figure 2.1 Step 2); (iii) from a LRMS to an available WN. In this architecture, sites are composed of single or multiple CEs,

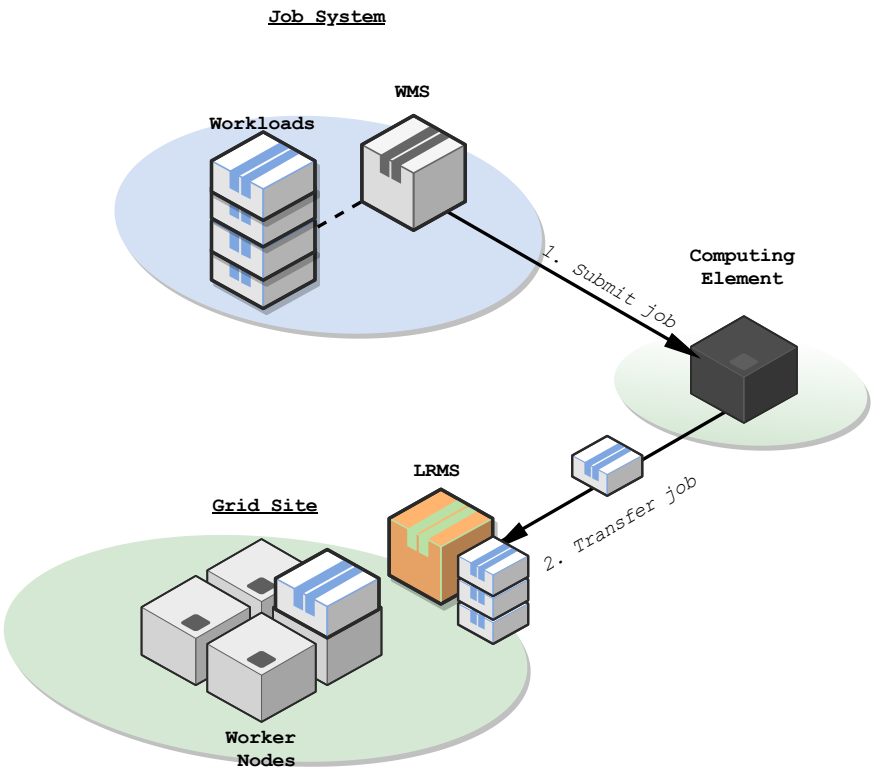

Figure 2.1 – Interactions between a Workload Management System (WMS) and a grid Site to execute a workload via the push model.

and WNs are mostly grouped and bound to LRMS queues homogeneously.

The push-based model has been shown to be inefficient and error-prone (Stagni et al., 2015). Indeed, jobs are transferred through middleware before waiting for resources within a LRMS queue. A broken site would result in failures of jobs, which would need to be submitted again. Moreover, WMS have to gather information about many sites and have to compare them with the features of the jobs, which can be a long and complex operation. Once submitted, jobs cannot be transferred elsewhere on the grid, which might put a significant load on certain sites.

Therefore, WMS developers tend to prefer pull-based approaches over the push model when possible: WNs should have external connectivity. It is the case in the WLCG context, but many supercomputers do not allow it and require jobs to be pushed. Several ATLAS teams have employed the Arc Control Tower (aCT) infrastructure to submit HEP applications on supercomputers having no outbound connectivity

(Filipčič, 2011; Nilsen et al., 2015), such as O'Brien et al. on the High-End Computing Terascale Resource (HECToR) facility (O'Brien et al., 2014), Filipčič et al. on Tody, a supercomputer located at the Swiss National Supercomputing Center (CSCS) (Filipčič et al., 2015) and Filipčič on the Chinese HPC CNGrid network to run Monte-Carlo simulation jobs (Filipčič, 2017). Installed outside of the supercomputer, aCT creates a bridge between the WMS and an ARC CE: it fetches jobs from the WMS and manages the pre-processing of the inputs and the post-processing of the outputs of the jobs, which would require an outbound connectivity. Then, it submits the jobs to an ARC CE, monitors and reports their status to the WMS.

DIRAC developers have chosen to fully support the pull model, and more specifically the Pilot-Job paradigm. Thus, we would need to reintroduce solutions to push jobs on given types of resources. Developers of aCT have designed a generic and open-source tool. In Chapter 4, we study the possibility of reusing the tool, or creating a similar utility specific to DIRAC if necessary.

Pull model: Pilot-Job paradigm

The Pilot-Job paradigm has been devised and implemented mostly to support computations across multiple distributed machines, aggregated into high-performance clusters, computing grids or virtualized in cloud infrastructures. It has been quickly adopted in the Grid Computing context as an answer to the inefficiencies of the push model. The paper, given by Casajus and his colleagues of the LHCbDIRAC team (Casajus et al., 2010), defines Pilot-Job objects, also known as pilots, as "nothing more than empty resource reservation containers that are sent to the available computing resources with the final aim of executing the most appropriate pending job in the central WMS queue" (see *Pilot* in Figure 2.2). Pilots can perform basic sanity checks of the running environment (Figure 2.2 Step 3) before any binding with a given job (Figure 2.2 Step 4) to effectively run tasks on well-behaved and well-adapted resources (Figure 2.2 Step 5). They create an overlay network that masks the central WMS components from the heterogeneity of the underlying resources.

DIRAC proposes its implementation of the Pilot-Job paradigm where pilots, considered as simple jobs, can run on WNs. A pilot performs a DIRAC installation and checks properties of the work environment such as CPU, memory and available disk space. Then, it gets the most suitable jobs from the central DIRAC WMS server. It

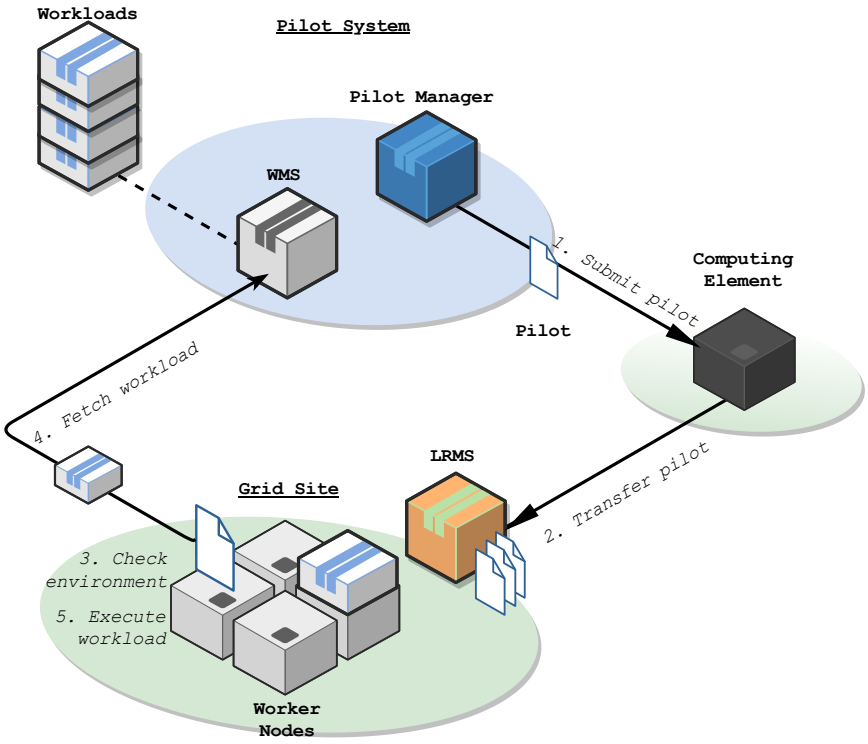

Figure 2.2 – Interactions between a Workload Management System (WMS) and a grid Site to execute a workload via the push model applying the Pilot-Job paradigm.

retrieves and checks the availability of the input data and software, executes the task, reports the success or the failure of the execution, and uploads output data if required.

Pilot provisioning tools aim at automating the submission of Pilot-Jobs to the resources, ensuring high availability and maximizing the throughput of the jobs (see *PilotManager* in Figure 2.2). A lot of WMSs integrate such tools to supply WNs with pilots. As Turilli et al. underline, the Pilot-Job paradigm appeared as a real solution for solving the inefficient push model (Turilli et al., 2018). We have seen an immediate development not grounded on any analytical understanding of underpinning abstractions, architectural patterns or computational paradigms, which led to a variety of Pilot-Job implementations and thus of Pilot-Job provisioning tools.

Condor (Bricker et al., 1992), originally designed to allow users to execute tasks on a resource pool, was one of the first software to implement the Pilot-Job paradigm, under the name of Glideins (Frey et al., 2002), to employ the grid resources via resource placeholders. It has been quickly complemented by GlideinWMS (Sfiligoi, 2008) to

automate and optimize the provisioning of the Glideins. Global computing WMS, such as XtremWeb and BOINC, rapidly adopted the concept. In this context, users install a client, which behaves as a kind of resource placeholder to pull tasks from the server when CPU cycles are available. In the meantime, WMS such as DIRAC, PanDA, AliEN, and Coaster (Hategan et al., 2011) have been developed and provided similar pilot deployment features despite slight variations.

Luckow et al. were the first to pinpoint this lack of generality and proposed a conceptual basis to compare and contrast different Pilot-Job frameworks through an abstract model called P* (Luckow et al., 2012). They also designed a generic implementation called Saga BigJob (Luckow et al., 2010), which is not maintained anymore. The P* model, represented in Figure 2.3, defines the following elements:

- *Pilot*: the entity that gets submitted and scheduled on a resource.
- *Compute Unit* (CU): encapsulates a task specified by the application that is submitted to the Pilot-Job framework.
- *Scheduling Unit* (SU): units of scheduling internal to the P* Model. Once a CU is under the control of the Pilot-Job framework, it is assigned to an SU.
- *Pilot-Manager* (PM): responsible for orchestrating the interaction between the Pilots and the different components of the model, and decisions related to internal resource assignment

Most of the Pilot-Job provisioning mechanisms aim at maximizing the throughput and minimizing the number of wasted resources by keeping a fixed number of pilots in the Grid pool and continuously instantiating them while there are jobs to process. The tools usually generate pilots that take the form of scripts, sent to WNs via the grid architecture and the push model. They also monitor pilots to identify failures and adjust the number of pilots to meet the demanding pressure. The characteristics and the priorities of the jobs are matched with the attributes of the resources to achieve the best binding. Rubio-Montero et al. (Rubio-Montero et al., 2015) and Turilli et al. (Turilli et al., 2018) emphasize the commonalities but also the differences between several WMS in further details.

LHCbDIRAC primarily relies on Pilot-Jobs to supply WLCG computing resources. In this context, an analysis of the DIRAC Pilot-Job provisioning tool, named Site

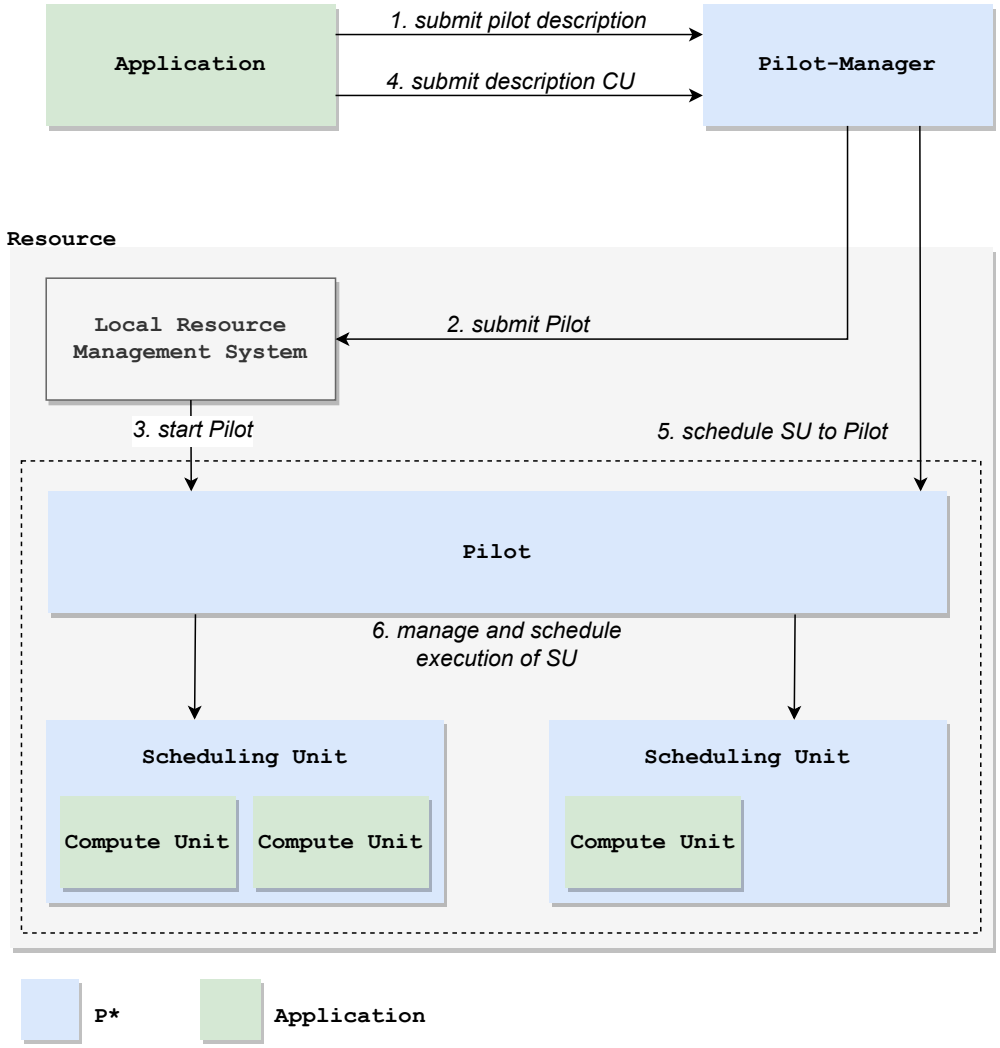

Figure 2.3 – P* Model: Elements, Characteristics and Interactions as defined by Luckow et al. Luckow et al., 2012.

Director, should be conducted to check whether the implementation is optimal. It could emphasize limitations preventing the full exploitation of the resources and allow us to develop different approaches to improve the throughput of the jobs on WLCG resources and face up to the growing amount of data coming from the LHC Run3. This will be addressed in further details in Chapter 3.

Pull model: further approaches

The emergence of clouds and other opportunistic resources has encouraged the development of new deployment methods based on the pull model. McNab et al. developed

the Vacuum model to leverage VMs (McNab et al., 2014). It consists in spontaneously producing Pilot-Jobs within VMs. Experiments supply contextualization procedures prior to launching the Vac daemon. The Vac daemon creates VMs on behalf of the resource provider directly on the hypervisor machine on which it runs. Then, it gathers information and stops VMs based on its observation, if VMs do not have any job to process for instance. The Vcycle daemon was developed following the successful deployment of Vac for production LHCb and ATLAS workloads. The Vcycle daemon extends the abilities of the Vac daemon to IaaS environments.

Some sites, especially supercomputers, propose a partial outbound connectivity from the WNs: they can exclusively communicate with Data Access Node (DAN)s, which can communicate outside the system. In such a context, Pilot-Jobs cannot fetch jobs directly from the WNs, whereas jobs cannot interact with external services. Several ATLAS teams have worked on different solutions to efficiently interact with supercomputers with partial outbound connectivity and no CE available. Oleynik et al. redefined the Pilot-Job concept to serve as a job broker on the DANs of Titan (supercomputer at the Oak Ridge National Laboratory) (Oleynik et al., 2017). In this context, PanDA submits pilots to the DANs that have both access to the external and internal networks. Similar to aCT, each pilot queries jobs, manages the pre- and post-processing steps, and then submits jobs to the LRMS of Titan.

Oleynik et al. also introduced *Next Generation Executors* (NGE), which is a run-time system to submit heterogeneous and dynamically determined workloads (Oleynik et al., 2017). It contains a Pilot Manager installed on the DANs, that submits pilots to the LRMS. Once running, a pilot instantiates a Pilot Agent and Executors on the allocated nodes. The Pilot Manager fetches jobs from PanDA and communicates with the Pilot Agent via a database installed on a DAN. Then, Executors process the jobs. Such installation is possible provided that the administrators of the supercomputer allow running services on DANs.

In the same way, the ALICE collaboration has developed a tool called ANALISA to interact with supercomputers with partial outbound connectivity such as the US Department of Energy's National Energy Research Scientific Computing Center (NERSC) resources: Edison and Cory (Fasel, 2016). ANALISA is composed of two parts: (i) the submission layer, which runs on a DAN, prepares the user software and the sandbox to contain the jobs; (ii) the worker layer, which is executed on the WNs, runs the jobs

and manages the transfer of input and output files within the local shared file system.

Maeno et al. have introduced Harvester, an attempt to provide a universal but still PanDA-specific Pilot-Job submission system on Grid and supercomputer resources (Barreiro Megino et al., 2020; Maeno et al., 2019). The service provides push and pull mechanisms, and lightweight and heavy installation procedures, on the supercomputer DANs or outside of them.

2.2.3 Authentication and Authorization

Within allocations, and especially through a pull-based model solution, jobs are expected to interact with WMS services and may request sensitive data and operations. Any malicious operation in such distributed ecosystems could lead to worldwide issues impacting a large number of communities. According to the RFC 2977 entitled *Mobile IP Authentication, Authorization, and Accounting Requirements*, "the need for service from a foreign domain requires, in many models, Authorization, which leads directly to Authentication, and of course Accounting".

WMS and site administrators must be able to verify a claimed identity and determine if a particular right, such as access to some resource, can be granted to the presenter of a particular credential. In the context of WLCG, they should maintain a collaborative collection of information on resource usage. The WLCG Authentication and Authorization Infrastructure (AAI) ensures: (i) confidentiality of processed data among distributed grid sites; (ii) traceability of user actions in WLCG resources; (iii) attribution of identifiers to determine users; (iv) isolation and suspension of compromised user's identity.

X.509 certificates

The current WLCG AAI, chosen in the early 2000s, is based on the Grid Security Infrastructure (GSI) of the Globus Toolkit (Welch et al., 2003), which relies on X.509 certificates (Cooper et al., 2008). According to Ceccanti et al. (Ceccanti et al., 2019), the approach consists in a trust fabric informing services about the certificate authorities (CAs) that can be trusted, X.509 certificates issued by trusted CAs to users and services for mutual authentication purposes, proxy certificates to delegate authentication and

authorizations from X.509 certificates and Virtual Organization Membership Service (VOMS) attribute certificates used to augment identity information with VO-issued authorization attributes that drive the authorization at services.

Grid-based WMS and middleware implement the same authentication model. Clients and services are identified by X.509 certificates and user credentials are forwarded to the final destination via proxy certificates. Authentication is done by checking the credentials received against a list of valid CAs. If the credentials presented fail to be signed by one of the valid CAs, then the connection is closed. Casajus et al. provide further details about X.509 certificates in the DIRAC context (Casajus, Graciani, et al., 2010), which is similar to other grid-based WMS. Every time a client connects to a DIRAC service, DIRAC extracts the client's credentials from the SSL handshake. The credentials identify the requester and DIRAC associates a set of properties to the credentials. To execute the requested action, the requester has to have at least one property in the set of properties required by the action.

Casajus et al. also explain how DIRAC transports users' credentials to WNs via the Pilot-Job paradigm. They define two types of Pilot-Jobs: generic and private. Private Pilot-Jobs are submitted to the grid using the user's credentials, whereas generic Pilot-Jobs are submitted with credentials from privileged users able to change their identity in the WNs to run the real user jobs. Private Pilot-Jobs were never employed in practice. The DIRAC Proxy Management service allows users to upload long-lived proxies. As proxies allow to act on behalf of users, they are considered very sensitive data and, therefore, can only be downloaded by a restricted set of users and agents, such as Pilot-Jobs once installed on a WN. Pilot-Jobs download short-lived and limited proxies from the long-lived proxies hosted in the Proxy Management service to run the jobs.

On the one hand, the GSI approach has worked well for years, "providing a secure infrastructure that has scaled to millions of jobs and hundreds of sites, has supported important scientific discoveries and has been adopted by several research communities besides HEP" (Ceccanti et al., 2019). On the other hand, experience has also exposed the main limitations of the current WLCG AAI: (i) X.509 certificates are complex, annoy most scientific users and lead to errors; (ii) Proxy certificates do not natively work in browsers, which led to complex workarounds to integrate VOMS with Science Gateways; (iii) Identity federations, such as EDUGAIN, cannot

be easily integrated within the current infrastructure; (iv) X.509 represent a barrier when integrating computing and storage resources from external partners, such as commercial providers, hybrid clouds and HPC centers. Various communities have started to work on alternatives to X.509 certificates. Token-based solutions have stood out.

Tokens

The current approach is considered outdated by today's standards regarding the protection of the privacy of user data. In the meantime, alternative AAI concepts have emerged and have been widely adopted by the industry:

- OAuth 2.0 (Hardt, 2012) has become the standard framework for delegated authorization for HTTP services. According to Ceccanti et al., it defines authorization flows targeted at service, desktop and mobile applications that describe how access tokens can be obtained from authorization servers and presented at services to be granted access to resources (Ceccanti et al., 2019).
- OpenID Connect (Sakimura et al., 2014) was designed to extend OAuth with an identity layer. It defines how authentication information is provided to services.
- JSON Web Tokens (JWTs) (M. Jones et al., 2015) provide a mechanism to express claims meant to be securely exchanged between services. Claims can describe user identity, authentication properties, attributes and capabilities.

In 2017, while multiple activities independently started to work on token-based authorization, the WLCG Authorization Working Group was formed (Bockelman et al., 2020). The group gathers experts from multiple domains aiming at charting a path towards token-based authorization for WLCG. Efforts have been focused on addressing usability and simplifying integration with third-party services and software.

The group eventually chose to adopt the INDIGO Identity and Access Management (IAM) service as the core of the future WLCG AAI. Conceptually and practically, INDIGO IAM aims to replace VOMS as the VO attribute authority, without being limited to a single authentication mechanism. INDIGO IAM provides a central VO-scoped authorization server, dealing with user authentication, registration and high-level authorization for a VO. INDIGO IAM allows users to link multiple identities to a VO account, which are exposed to services via standard OpenID Connect claims. To

enable a seamless transition, INDIGO IAM supports both X.509 certificates and tokens via a combination of JWTs, OAuth and OpenID Connect.

Existing grid middleware have started to transition from X.509 certificates to tokens as well. Developers of HTCondor have integrated SciTokens (Withers et al., 2018) to the LRMS and have planned to drop support of the GSI of the Globus Toolkit by 2023. Developers of DIRAC have started to implement token-based services and agents, but the solution is not production-ready and does not specifically focus on interactions between WMS and grid middleware. We would need to implement WMS components to accompany the grid middleware transition before the end of support of GSI.

Through Section 2.2, we have seen mechanisms to interact with distributed and heterogeneous computing resources relying on standard protocols. The section was focused on means to get allocations while preserving security within the grid infrastructure. In Section 2.3, we are going to discuss about reproducibility within the allocations.

2.3 Providing reproducible environments

2.3.1 Getting a compatible and optimized environment

In the past decade, articles in both the scientific and popular press have shed light on the reproducibility crisis. In 2016, Baker conducted a survey of 1,576 researchers who took a brief online questionnaire on reproducibility in research (Baker, 2016). It targets peer review, replication of experiments, confirmation of the results, verification and open research. According to it, more than 70% of researchers have tried and failed to reproduce another scientist's experiments. While there are recognized systemic issues such as pressure to publish and selective reporting, there are also technical limitations. Thus, there is a rising need for reproducibility of results obtained through computational research.

Hill has worked on parallel random numbers, simulation and reproducible research. According to him, being able to reproduce the results of a computing experiment on distributed computers is paramount to meet the scientific method (Hill, 2015). He distinguishes repeatability from reproducibility. He defines scientific re-

producibility in its broad sens as "means that other researchers found similar results to those published, possibly with techniques and approaches different from what is announced in a reference publication", and repeatability as a subset of reproducibility consisting in "finding the same results with the same input data and algorithm". Repeatability of software is essential to test, verify and find errors, and thus, ensures the validity of the results. Yet, it is not always guaranteed, especially in a heterogeneous and distributed context.

Hill gives two technical limitations to reproducibility (Hill, 2015): (i) it is not possible to accurately code real numbers with a limited binary representation, and thus, the order of the operations achieved with the floating-point format has a technical importance even though they are mathematically associative; (ii) it is hard to deal with implementations in different programming languages, compiling with different compilers, executions on different OS - especially relying on various modern many-core architectures or hardware accelerators (Taufer et al., 2010). Hill emphasizes that even with the same compiler, language and OS, bitwise reproducibility is not always guaranteed from one run to another in the same computing environment in case of silent errors - also known as soft errors.

The LHCb collaboration, among many researchers and companies, is aware of the issues and has set up an infrastructure to validate software before pushing it in production. In Section 2.3.1, we focus on a specific aspect of reproducibility: we want similar results of Gauss across the sites. At most, results should have slightly and insignificant variations.

Software validation

Computing resources at the disposal of the LHCb collaboration mainly involve x86 CPUs, distributions of Linux such as Scientific Linux CERN or CentOS, as well as different types and versions of compilers. To ensure simulation tasks remain reproducible on these platforms, despite perpetual code changes, the LHCb collaboration has developed a testing and verification phase (Popov, Dmitry, 2019).

The commissioning phase corresponds to a release of a major version of an application, which may require migration to a more recent version of its dependencies. It consists in compiling the software stack, executing it and finalizing it successfully,

without any technical faults. The commissioning phase relies on the LHCb nightly build system (Clemencic and Couturier, 2014; Kruzelecki et al., 2010), which performs the build and the tests on several configurations and platforms. If the build is successfully finalized, nightly tests are executed.

Nightly tests aim to verify that the built software works. They specifically check that the external libraries, frameworks and toolkits are exactly of the versions that developers intended to use. They are designed to be simple and fast to execute, about a couple of minutes to a maximum of half an hour.

Once the active development phase is achieved, the main changes have been implemented and the software stack works from a purely technical point of view, much more sophisticated tests are launched via the LHCb Performance and Regression (LHCbPR) project (Couturier et al., 2014; Mazurov et al., 2017). LHCbPR provides support to conducting systematic profiling and allows comparing the results of performance and regression tests run on the LHCb applications. Timing and hardware performance data are collected from the Gaudi framework, which includes multiple profiling measures, and from the performance monitoring unit of modern CPU architectures. Collected data form comparable, reproducible and representative profiles, allowing regression analysis. LHCbPR provides a flexible framework able to deal with various profiling tools, and perform automated and regular executions to comply with the objective of a reliable regression analysis. It centrally stores performance information, which is then available through a web application. Compared to nightly tests, PR tests require more time, about four to six hours on a single core of a modern CPU, depending on the simulated beam conditions in the case of Gauss.

According to Popov et al., some problems are spotted only on ongoing productions at runtime, such as technical problems with large productions on the grid, troubles with reconstructions or discrepancies in physics distributions (Popov, Dmitry, 2019). At this point, the LHCb Physics Performance Working Groups are involved to verify the results and explore physics processes that would be impossible to analyze on smaller data samples. To deploy LHCb software on constrained computing resources, the collaboration has to negotiate with system administrators that may be already overwhelmed by requests for specific versions of software. Alternatively, the collaboration can resort to package managers or containers.

Package managers

System administrators have struggled for years to respond to software demands from various communities: toolchains, huge scientific software stacks and different versions and variants of a given package. In the meantime, they opt for a conservative approach related to software management and aim to preserve a stable environment. To satisfy both constraints, system administrators build and install packages and make them available via environment modules (Furlani, 1991), which allows users to pick the specific packages they want.

Nevertheless, system administrators manually install software, which is a time-consuming operation. They cannot deal with many large software stacks, and to protect their system, cannot let users install their software, requiring root privileges. Hence, users cannot redeploy the same software environment on another system. Moreover, for maintaining a secure environment, system administrators have to remove installed software or upgrade it in place, which may lead to broken user environments.

To cope with these issues, various teams have developed software package managers supporting non-root users, automating package builds and facilities to create package variants: Conda ("Conda", 2022), EasyBuild ("EasyBuild: building software with ease." 2022) and Spack ("Spack", 2022). Each of them focuses on certain aspects such as usability, time to result, reproducibility or security. Conda is user-friendly and targets desktop computers, but does not build packages from source locally and thus, cannot deliver as many performances as Easybuild or Spack, which generally run on HPC systems. Contrary to the package managers mentioned above, GNU Guix ("Guix", 2022) and Nix ("NixOS", 2022) require root privileges to work. GNU Guix is a general-purpose package manager that implements the functional package management paradigm pioneered by Nix. GNU Guix extends Nix features by proposing a unified interface for package definitions and their implementations. Both focus on software reproducibility. The LHCb collaboration was mainly concerned about reproducibility of the results and portability and, therefore, has decided to employ Nix to build part of the software stack of the experiment (Burr, Chris et al., 2019).

Unfortunately, package managers are still not perfect. According to Courtès et al., some of them generally implicitly rely on tools or libraries already installed on

the system, which can also be unavailable (Courtès and Wurmus, 2015). The ad-hoc naming conventions they rely on to identify builds fail to capture the directed acyclic graph (DAG) of dependencies that led to a particular build. Alternatively, progress in virtualization has resulted in various container implementations.

Containers

As we have explained in Section 1.3.1, containers only virtualize software layers above the OS level and provide a lightweight solution to reproduce an environment compared to VMs. Containers provide isolation through the use of Linux cgroups and namespaces and embed all the dependencies required to execute the contained software application. These dependencies may include system libraries, external third-party code packages, or OS-level applications. Docker ("docker", 2022), Singularity - now Apptainer - (Kurtzer et al., 2017), Shifter ("Shifter - Containers for HPC", 2022), Podman ("podman", 2022) and Charliecloud (Priedhorsky and Randles, 2017) are popular choices among container technologies.

Docker democratized container technology and remains one of the most popular solutions, especially in the cloud computing area. Docker needs root permission to run, which generally contradicts HPC system policies. Moreover, Docker does not natively support MPI operations that are often needed for many HPC tasks. Podman is an alternative to Docker, which replaces the daemon-client architecture of Docker with individual processes that run the containers. Podman can run rootless containers through the use of user namespaces. Apptainer, Charliecloud and Shifter are designed for HPC systems. While Shifter remains relatively specific to NERSC supercomputers according to Abraham et al. (Abraham et al., 2020), Singularity/Apptainer and Charliecloud have been adopted by various HPC systems across the world. Singularity/Apptainer provides isolation for workloads while preventing privilege escalation. Charliecloud focuses on simplicity of architecture instead of portability, and provides a way to encapsulate dependencies with minimal overhead. Abraham et al. deliver further details about these implementations (Abraham et al., 2020). DIRAC supports Singularity/Apptainer within certain grid sites and opportunistic supercomputers. It is able to launch jobs within platform-compatible containers on WNs.

Courtès et al. express reserves when it comes to reproducibility aspects (Courtès and Wurmus, 2015). According to them, containers are resource-hungry, coarse-grain

and do not compose. Furthermore, configuration files used to build containers - Docker files, Definition files - suffer from being too broad to reproduce a software environment. Lastly, the tendency to rely on complete third-party system images is a security concern: part of the containers can be considered a black box. To get more control over software across many distributed sites, CERN experiments have relied on the CernVM-File System (CVMFS) for years.

2.3.2 Distributing software across distributed computing resources: another approach

Delivering a reproducible environment along with up-to-date software across thousands of heterogeneous computing resources in a user-friendly way is a major challenge: Buncic et al. designed CernVM and CVMFS to tackle it by decoupling the software from the Operating System (Buncic et al., 2010). The approach has worked well in WLCG where sites are expected to mount CVMFS on the nodes, but this is rarely the case when it comes to opportunistic resources such as supercomputers. In Section 2.3.2, we detail the main features of CVMFS and illustrate existing solutions to integrate it on constrained resources having no external connectivity.

CVMFS

CernVM ("CERNVM-FS", 2022) is a thin Virtual Software Appliance of about 150 Mb in its simplest form. It supports a variety of hypervisors and container technologies and aims to provide a complete and portable user environment for developing and running HEP applications on any end-user computer and Grid Sites, independently of the underlying Operating Systems used by the targeted platforms. CVMFS is a scalable and low-maintenance file system optimized for software distribution. CVMFS is implemented as a POSIX read-only file system in user space. Files and directories are hosted on standard web servers and mounted on the computing resources as a directory. The file system performs aggressive file-level caching: both files and file metadata are cached on local disks as well as on shared proxy servers, allowing the file system to scale to a large number of clients (Buncic et al., 2010).

This approach has been mainly adopted by the HEP community and is now getting users from various communities (Arsuaga-Rios et al., 2015). In a few years,

it has become the standard software distribution service on Grid Sites of WLCG. CVMFS developers have extended the features of the file system and have provided additional tools to support clouds (Harutyunyan et al., 2012; Segal et al., 2011) and supercomputers (Blomer et al., 2017). At the beginning of 2021, CVMFS was managing about 1 billion files delivered to more than 100,000 computing nodes by (i) 10 public data mirror servers - called *Stratum1*s - located in Europe, Asia and the United States and (ii) 400 site-local cache servers ("CernVM-FS Overview and Roadmap", 2021).

To keep the file system consistent and scalable, developers conceived CVMFS as a read-only file system. Release managers - or continuous integration workers - aiming to publish a software release have to log in to a dedicated machine - named *Stratum0* - with an attached storage volume providing an authoritative and editable copy of a given repository (Blomer, Jakob et al., 2019). Changes are written into a staging area until they are committed as a consistent changeset: new and modified files are transformed into a content-addressed object providing file-based deduplication and versioning. In 2019, Popescu et al. introduced a gateway component, a web service in front of the authoritative storage (Popescu, Radu et al., 2019), allowing release managers to perform concurrent operations on the same repository and make CVMFS more responsive (Figure 2.4.1.b and 2.4.2.b).

The transfer of files is then done lazily via HTTP connections initiated by the CVMFS clients (Figure 2.4.3.b). Clients request updates based on their Time-to-Live (TTL) value, which is generally about a few minutes. Once the TTL value expires, clients download the latest version of a manifest - a text file located in the top-level directory of a given repository composed of the current root hash, metadata and the revision number of this repository - and make the updated content available. Dykstra et al. provide additional details about data integrity and authenticity mechanisms of CVMFS to ensure that data received matches data initially sent by a trusted server (Dykstra and Blomer, 2014). This pull-based approach has been proven to be robust and efficient, according to Popescu et al. (Popescu, Radu et al., 2019), and has been widely used to distribute up-to-date software on grid sites for many years (Figure 2.4.2.a). Figure 2.4 presents a simplified schema summarizing the software distribution process on grid sites via CVMFS.

Users may need to use various versions of software on heterogeneous computing resources implying different OS and architectures. To provide a convenient and repro-

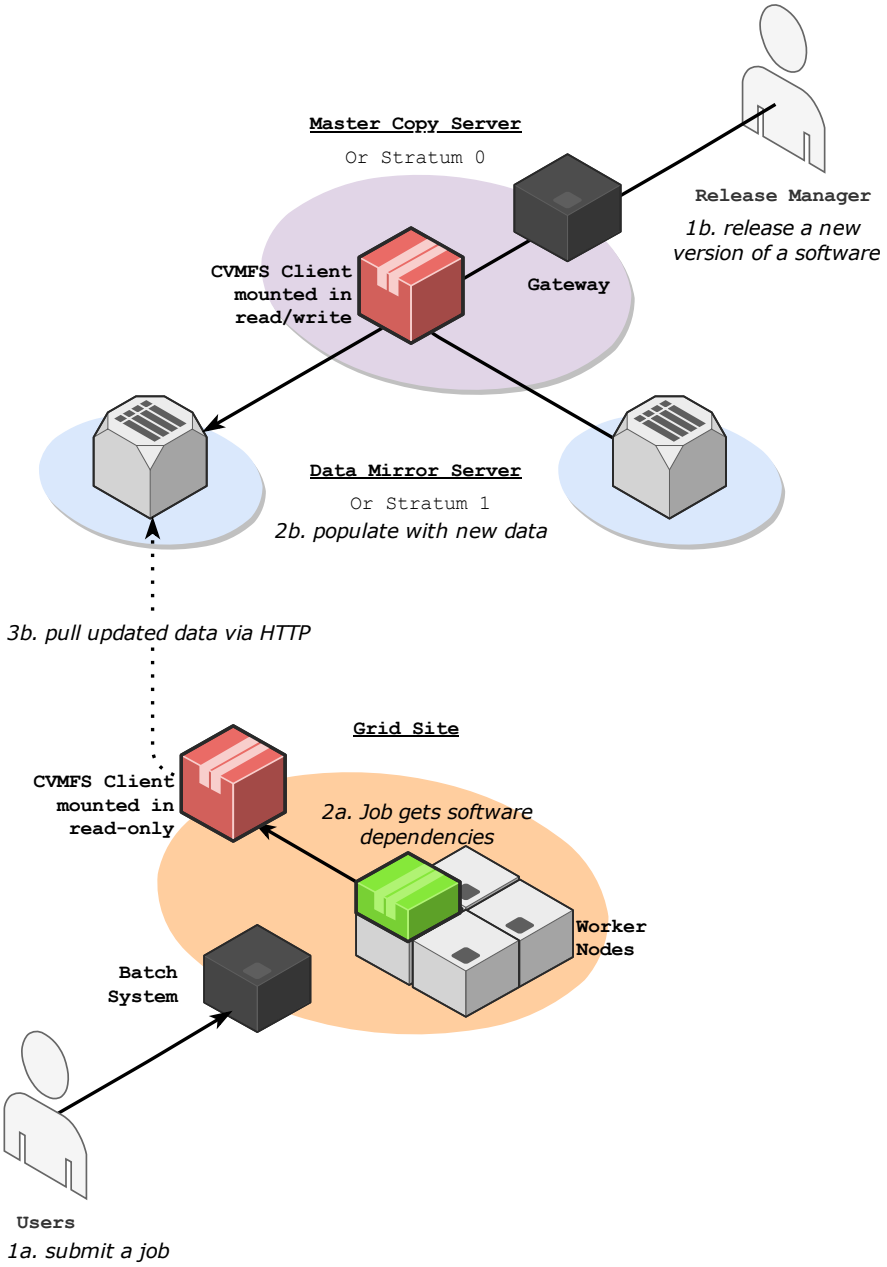

Figure 2.4 – Schema of the CVMFS workflow on Grid Sites: (a) the steps to get software dependencies from the job; (b) the steps to publish a release of a software in CVMFS.

ducible environment for the users, release managers generally provide software along with build files related to many architectures, OS and compilers. Many communities have started merging package manager systems with CVMFS (Boissonneault et al., 2019; Burr, Chris et al., 2019; Dröge et al., n.d.; Volkl, Valentin et al., 2021; Willett, 2019; Xu, Benda et al., 2020). Overall, the solutions consist of four nested layers: (i) the host-layer managing the network, CPU and co-processors; (ii) the file-system layer, CVMFS, dealing with software files across distributed computing resources; (iii) the compatibility layer, which generally take the form of a container or an OS such as a Nix or Gentoo ("Gentoo Linux", 2022), providing required OS components; (iv) the software management layer handling software and versions for various platforms (Easybuild, Spack).

Software delivery under constraints

To keep up with the computing needs, experiments have started to use supercomputers. Nevertheless, supercomputers have more restrictive security policies than Grid Sites: they do not allow CVMFS to be mounted on the nodes by default and many of them have limited or even no external connectivity. The LHC communities have developed different solutions and strategies to cope with the lack of CVMFS, which is a critical component to run their workflows.

Stagni et al. rely on a close collaboration with some supercomputer centers - Cineca in Italy and CSCS in Switzerland - to get CVMFS mounted on the worker nodes (Stagni et al., 2020). Nevertheless, their strategy is limited to a few supercomputers and their approach would be difficult to reproduce on a large number of supercomputers: most of them do not allow such collaboration.

To deal with the lack of CVMFS on supercomputers with outbound connectivity, Filipčič et al. studied two solutions: *rsync* and *Parrot* (Filipčič et al., 2015). The first solution consisted in copying the CVMFS software repository in the shared file system using *rsync*: a utility aiming to transfer and synchronize files and directories between two different systems. *rsync* added a significant load on the shared file system of the supercomputers and required changes in the repository absolute paths. The second solution was based on Parrot: a utility copied on the shared file system of the supercomputer, usable without any user privileges. Parrot is a wrapper using *ptrace* attached to a process that intercepts system calls that access the file system

and can simulate the presence of arbitrary file system mounts, CVMFS in this case. Nevertheless, the solution was "unreliable in a multi-threaded environment" (Filipčič et al., 2015) because it was unable to handle race conditions. These methods did not constitute a production-level solution but contributed to further and future advanced solutions.

In recent years, developments in the Fuse user space libraries and the Linux kernel have lifted restrictions for mounting Fuse file systems such as CVMFS. Developers of CVMFS have integrated these changes and designed a package called *cvmfsexec* ("cvmfsexec", 2022), which allows mounting the file system as an unprivileged user. The program needs a specific environment to work correctly: (i) external connectivity; (ii) the *fusermount* library or unprivileged namespace mount points or a setuid installation of Singularity/Apptainer. Blomer et al. provide additional details about the package (Blomer, Jakob et al., 2020).

Communities exploiting supercomputers that do not provide outbound connectivity cannot directly benefit from *cvmfsexec*: the package still needs to pull updated data via HTTP, which is not available in such context. We can distinguish two cases: (i) supercomputers that grant outside network or specific service access to a limited number of nodes and (ii) supercomputers that do not provide nodes with any external connectivity at all.

Tovar et al. recently worked on the first case (Tovar, Benjamin et al., 2021). They managed to build a virtual private network (VPN) client and server to redirect network traffic from the workloads running on the worker nodes to external services such as CVMFS. In this configuration, the VPN client runs on a worker node along with the job, while the VPN server is hosted on one of the specific nodes of the supercomputer and can interact with external services. Communities working on supercomputers from the second case cannot leverage the solution developed by Tovar et al.

O'Brien et al., one of the first teams to work with supercomputers in the LHC context, address the lack of external network access by copying part of it to the shared Lustre file system accessible by the WNs (O'Brien et al., 2014). The approach (i) worked because the environment of the supercomputer was similar to a grid site one, (ii) required changes in the CVMFS files and (iii) degraded the performance of the software as Angius et al. described (Oleynik et al., 2017). To tackle the latter issue on

the Titan supercomputer, Angius et al. moved the software to a read-only NFS server: this eliminated the problem of metadata contention and improved metadata read performance.

Similarly, on the Chinese HPC CNGrid, Filipčič regularly packed a part of CVMFS in a tarball. Filipčič provided a deployment script to install the software and fix the path relocation on the shared file system to the local system administrators: they were then responsible for getting and updating the CVMFS tarball on the network when requested (Filipčič, 2017).

To help communities to unpack a CVMFS repository in a file system, a team of developers designed *uncvmfs* ("uncvmfs", 2018). The utility deduplicates files of a software stack: it populates a given directory with the CVMFS files that are then hard-linked into it, if possible. The program was used, in combination with Shifter (Gerhardt et al., 2017), a container technology providing a reproducible environment, in the context of the integration of the ALICE and CMS experiments workflows on the NERSC High-Performance Computing resources (Fasel, 2016; Hufnagel, 2017). As a proof of concept, Gerhardt et al. used *uncvmfs* to deduplicate the ATLAS repository and copy it into an ext4 image - about 3.5Tb of data containing 50 million files and directories -, compressed into a 300Gb squashfs image; and Shifter to provide a software-compatible environment to run the jobs (Gerhardt et al., 2017). Despite encapsulating the files in a container reduced the startup time of the applications, the solution generated large images, long to update and deliver on time.

Teuber and the CVMFS developers conceived *cvmfs_shrinkwrap* to cope with large images (Teuber, 2019). The tool supports *uncvmfs* features with certain optimizations and delivers additional features: *cvmfs_shrinkwrap* can extract specific files and directories based on specification files, deduplicate them, making them easy to export in various formats such as squashfs or tarball. In this way, the following operations remain the responsibility of the user communities: (i) trace their applications - meaning, in this context, "capturing all their dependencies and their locations in the file system" -, (ii) call *cvmfs_shrinkwrap* to get a subset of CVMFS composed of the minimum required files, and (iii) export this subset in a certain format and deploy it on sequestered computing resources to run their jobs.

Douglas et al. already described such a project in an article (Benjamin, Douglas et

al., 2019), but the work remains specific to the ATLAS experiment. They use *uncvmfs* to produce a large image that has to be filtered afterward. We would need an open-source utility to assist various user communities in this process. It would take applications of interest in input and would output - with the help of *cvmfs_shrinkwrap* - a subset of CVMFS with the minimum required files to run the given applications, in combination with a container image if needed.

2.4 Using allocated computing resources efficiently

2.4.1 Exploiting multi-core/node allocations...

Computing resources have been acquired and the environment has been set up. From this point, it is essential to make the best use of the resources during the allocated time. Under-using them would represent a lack of workload processing for the consumers, and to some extent, a waste of financial resources. Over-using them could force the system to kill tasks before they could provide any meaningful result, which would lead to a waste of time and computing power. Therefore, in Section 2.4, we present approaches to maintain a good balance, both in terms of space (Section 2.4.1) and time (Section 2.4.2) allocated.

While it is common to get single-core allocations in grid resources - especially in WLCG -, multi-core and multi-node allocations are largely employed within the HPC community. In the context of this thesis, we define a multi-core allocation as a LRMS reservation of multiple hardware threads hosted within a single WN; and a multi-node allocation as a LRMS reservation of multiple hardware threads distributed within multiple WNs. HPC clusters tend to propose a limited number of possible allocations to users. In such a context, it is the responsibility of the users to exploit as many resources as possible within a single allocation and make the best use of them. Developers of grid WMS solutions attempting to exploit HPC clusters have worked on different solutions to exploit many-core allocations.

Multi-process/threaded tasks

For the LHC Run2, ATLAS developers chose to enable multi-processing in the reconstruction framework, AthenaMP, to exploit 8-core allocations while reducing the

memory consumption from 19 to 11 GB (Campana, 2015). As CMS and ALICE developers did (C. Jones, Collaboration, et al., 2017; Tadel and Carminati, 2010), ATLAS developers also worked on a multi-threaded version of their framework to better scale with many-core architectures (Leggett et al., 2017).

For its part, the LHCb collaboration has recently started to develop a multi-threaded framework on top of Gauss, Gauss-on-Gaussino, seen in Section 1.4.2. There was no urgent need until now, as the memory footprint of the LHCb Run2 applications was small enough to fit with all the computing resources available. To leverage multi-core allocations, the collaboration also created a method to flexibly provision jobs having different requirements on many-core nodes: the fat-node partitioning mechanism.

Fat-node partitioning

The LHCb collaboration had the opportunity to interact with the Marconi A2 partition of Cineca, a consortium joining 70 Italian universities, four national research centers and the Ministry of Universities and Research in Italy. The partition contains Xeon Phi 7250 (KNL) nodes, each of them composed of 68 physical cores and up to 4 hardware threads per core, seen as 272 logical cores. Rather than implementing a quick ad-hoc solution for the partition, DIRAC developers designed a flexible and generic fat-node partitioning mechanism to manage independent jobs simultaneously from a Pilot-Job (Stagni et al., 2020).

DIRAC submits a Pilot-Job on one many-core node per allocation. Once installed on the node, the Pilot-Job checks the number of processors available and sequentially fetches jobs accordingly to run them in parallel. The fat-node partitioning mechanism supports complex descriptions of jobs and resources. It allows users to submit single-core and multi-core jobs. Users can also specify a range of cores to use (e.g. from 1 to 32 cores), or a "Whole node" tag to use all the cores available on the node. In the same way, DIRAC can interpret and enforce constraints from resource providers. Resource providers may only accept single-core jobs, multi-core jobs, both, or only "Whole node" jobs. They may also ask the exact number of processors a job is requesting in advance.

Dealing with multi-node allocations requires further methods and libraries to

manage workloads in parallel across many nodes. From an allocation, we would need to identify and access available nodes.

Dealing with multi-node allocations

To exploit many-core and multi-node allocations, ATLAS teams developed MPI wrappers, combining multiple instances of single-node tasks simultaneously (De et al., 2015). In this context, it is worth noting that MPI is used for management of embarrassingly parallel tasks rather than for collaborative distributed computing, its main purpose. The MPI wrappers are workload-specific as they have to set up the environment, organize per-rank worker directories and input parameters, and clean up on exit. The WMS pushes MPI wrappers to LRMS, requesting multi-node allocations for each of them. At run time, a corresponding number of copies of a wrapper script is installed on each node allocated. Each copy starts a task as a subprocess and waits until its completion. A similar concept was applied by the ALICE collaboration on the NERSC supercomputer (Fasel, 2016).

Such an approach was used to exploit allocations on the Titan supercomputer. Each copy of an MPI wrapper was handling a multi-process Athena task. However, this concept makes the scheduling of multiple generations of tasks in the same allocation impossible: once a statically defined number of tasks are packaged into a wrapper, no further tasks can be added to that wrapper, which prevents efficient use of available walltime. The NGE concept introduced by ATLAS, and discussed in Section 2.2.2, addressed this issue (Oleynik et al., 2017). It consists in submitting pilots instead of MPI wrappers to: (i) avoid further job packaging and submission overheads; (ii) relax assumptions such as knowing the number of simulations, and events per simulation, before submitting; (iii) offer task-independent scheduling interface while hiding the mechanics of coordination and communication among multiple worker nodes. The pilots use the Open Run-Time Environment (ORTE), a critical component of the OpenMPI implementation, to coordinate the executors.

The DIRAC team also worked with MPI wrappers, in the context of the GISELA Latin American Grid Initiative in 2012 (Tsaregorodtsev and Hamar, 2012). Tsaregorodtsev et al. were mostly focused on supplying special services to support MPI jobs in grid sites not natively running MPI. They were able to generate dynamic allocation of virtual computer pools leveraging many single-core allocations in grid sites. Neverthe-

less, the project was abandoned. Currently, DIRAC does not embed any mechanism to support multi-node allocations. Building a layer on top of the fat-node partitioning concept is discussed in Chapter 4.

To meet growing demand, especially from the medical community and HPC users, Dotti et al. proposed a different approach: a Geant4 extension employing MPI (Dotti et al., 2015). A hybrid approach, combining MPI and multi-threads allows for simplified use of large core-count resources: users do not need anymore to write a custom script to perform job splitting, handling and merging.

2.4.2 ...As long as possible

CPU time spent on a task depends on the underlying configuration of the allocated computing resources. Site administrators may configure computing resources on their own: they can provide virtual or physical machines and enable or disable parameters such as over-clocking and hyper-threading, the latter being a potential source of errors ("[WARNING] Intel Skylake/Kaby Lake processors: broken hyper-threading", 2017). They can also involve heterogeneous computing resources having different CPU architectures, memory access time or caching capabilities. Therefore, benchmarking computing resources capabilities, regarding a given task, is critical to exploit allocations efficiently.

CPU benchmarking

Andersen et al. define benchmarking as "the process of continuously measuring and comparing one's business processes against comparable processes in leading organizations to obtain information that will help the organization identify and implement improvements" (Andersen and Pettersen, 1995). CPU benchmarking involves measurement and comparison of the performance of various processors to run applications of interest. Charpentier from the LHCb collaboration (Charpentier, 2017) and Valassi et al. (Valassi, Andrea et al., 2020) identified several reasons proving that CPU benchmarking is a critical task for the HEP community:

- Experiments keep an accounting of the consumed resources, both yearly and in the planning of long-term projects.
- Experiments know whether a given task will have enough compute resources

to complete successfully before attempting to run it. They can optimize its placement.

- Individual computing sites buy the CPU resources providing the best price-performance ratio.

- Software developers compare the performance of an application to the theoretical compute power of the machine where it is run, or to another application.

Dixit from IBM Corporation defines three types of popular CPU benchmarks: kernel, synthetic and application (Dixit, 1993). Kernel benchmarks are based on valuable code fragments representing the majority of CPU time in applications of interest. Synthetic benchmarks are custom-built to include a mix of low-level instructions resembling those found in applications of interest. Kernel and synthetic benchmarks are generally small, prone to attack and measure only the CPU performance. The best benchmark for a given application is probably the application itself. Nevertheless, due to the number of existing applications, considering that many of them are proprietary, this option is often impractical.

Results of a CPU benchmark can have various formats (Pelevanyuk, 2021). A score can take the form of (i) a number corresponding to a relative performance on some artificial workload, or (ii) absolute numbers based on different metrics such as Flops, MB/s, reads/s and writes/s.

CPU benchmarking in HEP

To cope with challenges of ever-greater complexity, developers have to design cutting-edge software relying on more specific and heterogeneous computing architectures. These always-changing requirements imply CPU benchmarking evolution. Valassi et al. propose an exhaustive state-of-the-art related to the CPU benchmarking history in the HEP domain (Valassi, Andrea et al., 2020).

In the late 1970s, the User Support Group at CERN designed the "CERN Unit", a CPU benchmark based on a set of typical FORTRAN66 programs for event simulation and event reconstruction (McIntosh, 1992). The results were normalized by comparison with an IBM 370/168. The "CERN Units" were mainly used to grant CPU quotas to users of the CERN central systems. In the early 1990s, McIntosh decided to redefine

the "CERN Unit": it was impossible to run the benchmark on newer machines. Thus, McIntosh changed the tests to make them more portable and more representative of the HEP workload of this time (McIntosh, 1992).

In his article, McIntosh also reviews other existing benchmarks outside HEP and mentioned - without knowing it at this time - the popular successor of the "CERN Unit": the SPEC benchmark suite. The Standard Performance Evaluation Corporation (SPEC) defines itself as a "non-profit corporation formed to establish, maintain and endorse standardized benchmarks and tools to evaluate performance and energy efficiency for the newest generation of computing systems" (“Standard Performance Evaluation Corporation”, 2022). SPEC aims to "provide the benchmarker with a standardized suite of source code based upon existing applications that have already been ported to a wide variety of platforms by its membership" (“Standard Performance Evaluation Corporation”, 2022). After SPEC CPU92 and SPEC CPU95, the HEP community adopted SPEC CINT2000 (SI2K), an integer benchmark suite comprised in SPEC CPU 2000 (Michelotto et al., 2010), for the Computing Technical Design Report (CTDR) of all the LHC experiments (Valassi, Andrea et al., 2020). In 2005, several presentations at HEPiX (“HEPiX Online”, 2022), an international group of HEP computing users founded in 1991 and interested in benchmarking topics, pinpointed discrepancies between the performances of HEP applications and the SI2K scores. In 2006, the HEPiX Benchmarking Working Group (BWG) was established to address these discrepancies. In 2009, a consensus emerged within HEPiX: the group suggested adopting a new HEP-specific benchmark named HEP-SPEC06 (HS06), based on the latest SPEC suite at this moment, SPEC CPU2006 (Henning, 2006).

HS06 was written in C++ and is composed of seven programs representing real applications, mostly from scientific domains, although not from the HEP domain (Valassi, Andrea et al., 2020). The difference between HS06 and SPEC CPU2006 lies in a few HEP-specific tunings. Up to now, it has been the official CPU performance metric to be used by WLCG sites: (i) HS06 score was found to be highly correlated to throughput on a large number of diverse machines replicating typical WLCG worker nodes, for several HEP applications from all the LHC experiments; (ii) HS06 CPU usage pattern was found to be quite similar to that observed on the CERN batch system used by the LHC experiments, and its memory footprint was comparable to that of typical HEP applications.

In 2017, Charpentier, from the LHCb collaboration, analyzed correlations between HS06 and LHCb reconstruction and simulation applications (Charpentier, 2017). He observed bad scaling properties with the applications. The ALICE collaboration reached the same conclusion (Giordano and Santorinaiou, 2020), and the HS06 benchmark was declared as no longer representative of WLCG software and computing (Valassi, Andrea et al., 2020). Different reasons led to this conclusion: (i) memory footprints of the applications have increased in general; (ii) 64-bit builds have replaced 32-bit builds; (iii) multi-threaded, multi-process and vectorized software solutions are becoming more common; (iv) the hardware landscape is also more and more heterogeneous, with the emergence of non-x86 architectures such as ARM, Power9 and accelerators, especially at HPC centers. To address this issue, the HEPiX BWG started further investigations to find novel benchmark solutions that could better fit with the new HEP applications.

The SPEC CPU 2017 (SC17) benchmark suite, released by SPEC in 2017, as the replacement of SPEC CPU 2006 was one of the considered candidates. SC17 is larger and has a more complex codebase, with respect to its predecessor, and is shaped for multi-core and multi-threads applications. Contrary to the initial expectations, it was proven that SC17 does not bring much benefit with respect to HS06 (Giordano and Santorinaiou, 2020). Since 2017, in conjunction with LHC experiments, the HEPiX BWG has developed the so-called HEP-Benchmarks suite (Valassi, Andrea et al., 2020) - the future successor of HS06 - based on HEP applications. The suite includes three main components:

- The HEP-workloads package: the core of the suite. It contains the code and infrastructure to build a standalone container for each of the HEP software workloads it includes. The HEP-workloads package supports a range of workloads going from single-process and single-threaded software compatible with x86 CPUs to multi-process, multi-threaded applications targeting non-x86 CPUs and accelerators. The system relies on CVMFS (“CERNVM-FS”, 2022), and Linux containers.

- The HEP-score package: combines the benchmark scores derived from the individual HEP workloads into a single number.

- The HEP-benchmark-suite package: a toolkit to coordinate the execution of several benchmarks including HEPscore along with HS06, SC17 and others.

Pelevanyuk, from the Joint Institute for Nuclear Research (JINR), explains that HS06 is based on proprietary software and, therefore, is accessible only to site administrators (Pelevanyuk, 2021). Running the whole suite of benchmarks is a time-consuming operation, so site administrators perform the tests only if the structure of their resources has changed. Thus, LHC experiments cannot leverage HS06 to get insights into whether a given task could run within the allocated time on a given worker node. They have relied on different approaches: Machine/Job Features, fast CPU benchmarking tools and fine-grained event processing systems.

Machine/Job Features

In 2016, the HEPiX virtualization group, in conjunction with the WLCG Machine/Job Features Task Force introduced the Machine/Job Features (MJF) mechanism (Alef et al., 2016). MJF was designed as a set of standard specifications, implemented by the resource providers. MJF provided running jobs with access to detailed information about their current host (Machine features) and meta-information about themselves (Job features). Machine features included the number of cores available, the CPU power (HS06) and the shutdown time if any maintenance was planned. Job features contained many details such as the number of cores allocated, the start and end dates of the job, and the maximum CPU time or wall clock time that should be spent running the job. MJF details were independent of the LRMS or VM model used, and accessible either locally via the filesystem on a worker node, or remotely via HTTPS.

In 2016, the task force proposed MJF scripts to different sites willing to test the software: a dozen of Tier1 and Tier2 sites were providing Machine/Job Features. WLCG set up a monitoring infrastructure to measure the progress of the deployment and validate the participating sites. In practice, sites did not all adopt MJF or did not update the features after changing their computing resources, which led to fragmented and erroneous information. Eventually, the project was abandoned. As a consequence, fast CPU benchmarking appeared as a more reliable approach.

Fast CPU benchmarking in HEP

Developers of DIRAC designed an open-source fast and synthetic HEP CPU benchmarking solution named DIRAC Benchmark in 2012 (DB12) (“Dirac Benchmark 12”,

2022). The project has been written in Python2, is portable, and spends less than a minute to provide an estimated HS06 score. The LHCb experiment and JINR heavily rely on DIRAC Benchmark to compute the CPU power of the worker nodes, prior to fetching a HEP task from a central broker (Korenkov et al., 2020). Combined with the CPU time left information coming from the LRMS, it provides the CPU work left in an allocation in HS06.seconds, and allows running the most appropriate HEP task on a given computing resource:

$$CPU_{work} = CPU_{power} \times CPU_{time} \tag{2.1}$$

For instance, a computing resource available for 1000 HS06.seconds would be able to run two 500-HS06.second jobs. In 2016, Charpentier, from the LHCb collaboration, finetuned DIRAC Benchmark to better fit with the evolution of the Monte-Carlo simulation applications of the experiment, which resulted in the creation of Dirac Benchmark 2016 (DB16). DB16 simply applies a factor of 1.54 to DB12, is specific to the LHCb experiment and, therefore, has not been integrated into the DIRAC Benchmark repository. To save CPU time, Stagni et al. introduced elastic MC simulation jobs based on DB16 (Stagni and Charpentier, 2015). Once introduced, a MC simulation production is first run on a test site: a small number of events are executed. Once completed, the CPU time spent on each event is averaged and combined with the CPU power:

$$CPU_{work}\ of\ 1event = CPU_{power} \times mean(CPU_{time}\ per\ event) \tag{2.2}$$

The $CPU_{work}\ of\ 1event$ is expressed in HS06.seconds. The value is stored and bound to the MC simulation production. At the beginning of its execution, a job related to a given MC simulation production computes the number of events to produce, based on the CPU work left in the allocation (Equation (2.1)), and the CPU work necessary to execute 1 event of the production - which was obtained on the test site with Equation (2.2). Because the CPU power value computed by DIRAC Benchmark is an estimation, a margin of 0.75% is applied to make sure that the job

will not be killed by the system:

$$events\ to\ produce = \frac{CPU_{power} \times CPU_{time}}{CPU_{work}\ of\ 1 event} \times 0.75. \tag{2.3}$$

DIRAC Benchmark has been successfully employed for many years by various communities, but would need to be ported to Python3: Python2 was declared deprecated in January 2020. This would potentially imply score discrepancies and would need to be cautiously handled and extensively tested. Moreover, the fast CPU benchmarking solution would need to be further analyzed to comply with new types of processors. We will tackle the issue in Chapter 3. Alternatively, fine-grained event processing system would also allow experiments to efficiently leverage CPU time within the allocations, especially from preemptible computing resources.

The ATLAS Event Service: a different approach

ATLAS teams have worked with various opportunistic and preemptible computing resources (Cameron et al., 2017; Svatos et al., 2020a, 2020b). In this context, jobs could be killed at any time without any warning if other users would require the resources.

To address this issue and fully exploit computing resources, Calafiura et al. designed a fine-grained event processing system named ATLAS Event Service (AES) (Calafiura et al., 2015a). Based on the Job Execution and Definition Interface (JEDI) (De et al., 2014), which dynamically breaks down tasks at the level of either individual events or event clusters, AES streams a small portion of the input data to the jobs in real-time. While the jobs persist, they can elastically continue to consume new inputs and stream away outputs with no need to tailor the execution time to the time left. In such as case, jobs are eventually killed by the system at any time with minimal data losses. AES is particularly adapted to opportunistic and preemptible computing resources with external connectivity.

To deal with partial outbound connectivity, Calafiura et al. developed Yoda, an MPI application submitted to the LRMS by a specialized component of the Pilot Manager running on the edge node (Calafiura et al., 2015b). The Pilot Manager is responsible for downloading input data to the shared file system, getting job defini-

tions from the PanDA server and streaming out the outputs produced by the Yoda jobs. A Yoda application consists of a leader orchestrating the entire MPI application by continuously distributing fine-grained input data to followers and collecting their outputs.

DIRAC Benchmark prevents jobs to be killed by the system and releases computing resources before the end of the allocation, which can represent waste for the community submitting jobs. Conversely, AES and Yoda let jobs run out of time to exploit allocations as long as possible, which generates a small waste per allocation: the last events processed do not provide any meaningful result. The approach is well-suited for preemptible resources, but not for pledged resources. Indeed, running out of time would imply not releasing resources that could have been better used by other communities.

2.5 Conclusion

This chapter is about getting as many allocations as needed and exploiting them efficiently while preserving reproducibility across distributing and heterogeneous sites. It should be most suitable for WMS developers aiming to manage HTC workloads with limited inputs on various independent sites, especially in mid-sized clusters and supercomputers.

First, we have described the grid environment and various implementations of middleware: LRMS, CEs, WMS (Section 2.2.1). While middleware relies on common standards, it is worth noting that the profusion of implementations makes it difficult to leverage a large number of computing resources and justifies the development of extra components. We have presented various approaches to submit jobs from WMS to WNs in such an ecosystem (Section 2.2.2). The push model remains generally adapted to any use case, especially when constraints are important, but remains inefficient. Conversely, pull model solutions are more efficient but remain specific to certain types of computing resources. The Pilot-Job paradigm has been widely adopted in the grid context for years but is not always adapted for supercomputers. Furthermore, we are not sure the Pilot-Job implementation of DIRAC is efficient enough to fully exploit grid computing resources. We have also briefly exposed security approaches to prevent malicious use of the WMS services from an allocation (Section 2.2.3). The

current approach, based on X.509 certificates is now considered outdated and should be replaced with a token-based implementation.

Then, after analyzing the LHCb software validation process, we have surveyed different approaches to bring Gauss requirements to the allocated computing resources. This mainly involve container and package manager implementations (Section 2.3.1). Package managers generally implicitly rely on tools or libraries already installed on the system, which can also be unavailable, whereas containers are considered resource-hungry and coarse-grain, their definition files being too broad to be precisely reproduced. We also described CVMFS, a shared file system developed for distributed software across many independent sites (Section 2.3.2). We have highlighted issues and existing solutions to integrate the shared file system on WNs of supercomputers. When it comes to resources with no external connectivity, most of the solutions remain too specific and do not cover the entire process to provide a functional subset of CVMFS.

Lastly, we have seen approaches to efficiently exploit allocations (Section 2.4). LHCb is equipped to manage multi-core allocations with the fat-node partitioning mechanism, but cannot deal with multi-node allocations. MPI libraries seem to be a great choice in this latter case. We have also illustrated two main approaches to run tasks as long as possible within allocations: fast CPU benchmarking tools and fine-grained event processing systems. LHCb has successfully employed DIRAC Benchmark to compute CPU work left within allocations. The benchmark has to be ported to Python 3 and be deeply analyzed in regards to Gauss executions.

This chapter comprises many solutions related to grid computing and supercomputers that we could leverage and extend to (i) better exploit WLCG computing resources (Chapter 3) and (ii) start integrating the LHCb workload - Gauss tasks in priority - on supercomputers (Chapter 4).

3 Towards a better throughput on Grid Resources

3.1 Introduction

WLCG resources only are no longer sufficient and needs additional support to process the continuously growing amount of data coming from the LHC experiments. Therefore, the WLCG communities have found ways to exploit non-reserved CPUs, often on non-formally pledged resources as stated in Section 1.4.2.

On the one hand, developers have made significant efforts to integrate non-grid and opportunistic resources such as cloud systems, supercomputers and volunteering computing middleware. They have developed novel mechanisms to provision workloads and efficiently exploit computing allocations on these new types of resources. On the other hand, there has been less focus on (i) Pilot-Job provisioning tools and (ii) fast CPU benchmarking tools dealing with shared and distributed heterogeneous clusters, such as grid resources. Yet, many virtual organizations such as LHCb still mainly depend on grid resources.

This chapter comprises two parts. In the first part (Section 3.2), we want to explore whether improving the Pilot-Job provisioning mechanism bound to the push model could speed up the Pilot-Job submission frequency and, thus, could increase the throughput of the jobs on grid resources (Boyer et al., 2022a; Boyer et al., 2020). We propose to test this hypothesis by analyzing and improving the "DIRAC Site Director" - the Pilot-Job provisioning utility used by LHCb on WLCG - and assessing the impact of the changes on WLCG over 12 months. If successful, the results of our study could be directly applied by communities involving DIRAC and could deliver insights to any community dealing with a grid architecture through the Pilot-Job paradigm in a broader sense. After the presentation of the fundamental features of the DIRAC Site Director and its current limitations in Section 3.2.1, we describe the solutions proposed to increase the Pilot-Jobs submission rate and the throughput of the jobs in Section 3.2.2. Results are finally assessed in Section 3.2.3 and discussed in Section 3.2.4.

In the second part (Section 3.3), we aim at porting DIRAC Benchmark to Python 3 and checking whether it would still be correlated with Gauss. In Section 3.3.1, we enumerate DIRAC Benchmark use cases and features. Then, in Section 3.3.2, we explain our methods to port DIRAC Benchmark to Python 3.9. The last two sections are about comparisons (i) between the Python 2 and the Python 3 versions of DIRAC

Benchmark (Section 3.3.3) and (ii) between the Python3-corrected version of DIRAC Benchmark 16 and Gauss (Section 3.3.4).

3.2 Improving Pilot-Job provisioning

3.2.1 Analysis of the DIRAC Site Director

The Site Director is a DIRAC agent performing Pilot-Job submission via the so-called push model to install pilots - mostly - on grid resources. In Section 3.2.1, we analyze its features and expose several limitations. While some of these limitations are bound to the grid architecture, some others could be addressed by performing changes within the source code and the configuration of the Site Director.

Overview of the Site Director

The Site Director works in cycles, executing the same logic at each iteration. An iteration consists in:

- Getting information about LRMS queues and the WNs bound to them from the *Configuration* service first (Figure 3.1 Step 1): OS installed, architecture of the WNs, maximum number of waiting pilots allowed in the queues, maximum number of pilots allowed in the Site. The *Configuration* service fetches information from the Berkeley Database Information Index (BDII), a service composed of multiple agents installed on the Sites, collecting data at the WN, Site, and Grid level according to Osman et al (Osman et al., 2012). The *Configuration* service also provides the Site Director with details about the CEs to connect to them: name, credentials if required.
- Querying the *Matcher* service, for each valid LRMS queue. Given a LRMS queue configuration - namely details about the architecture of the WNs and the OS installed on them - the *Matcher* service delivers a list of n_j jobs that could be executed on the underlying WNs (Figure 3.1 Step 2).
- According to the number of jobs that match the configuration n_j and the slots available in the LRMS queue S, generating a certain number of pilots n_p as scripts to run on the WNs: $n_p = \min(n_j, S)$ (Figure 3.1 Step 3). S the number of

slots available is determined by the limits set by the Site administrators minus the number of pilots submitted in previous iterations that are still waiting or running. Pilots previously submitted are registered in the *PilotsDB* database.

- Pushing these scripts through the multiple components of the grid to reach the WNs. A Site Director only submits the necessary number of pilots, according to the jobs waiting in the queues, to avoid congesting the Sites with empty pilots. To submit the scripts to a CE, the Site Director calls a DIRAC communication interface providing the necessary tool to interact with it (see Figure 3.1 Step 4).

- Registering the pilots submitted in the *PilotsDB* database on which next iterations will draw upon (Figure 3.1 Step 5).

- Monitoring pilots to spot failures and provision resources accordingly. The Site Director calls the DIRAC communication interfaces to get the status of the pilots (Figure 3.1 Step 6).

- Reporting the status to the *PilotsDB* database and the *Accounting* service (Figure 3.1 Step 7). The *Accounting* service collects and stores data about DIRAC activities that can then be used to build reports.

It is worth noting that a Site Director is highly and dynamically configurable. DIRAC administrators can set up multiple instances that can run in parallel and manage specific Sites, CEs, and types of resources to share the workload. Additionally, they can tune parameters to modify the functioning of a Site Director: execute the monitoring process every n_{update} cycle, wait n_{fail} cycles before submitting in a LRMS queue that failed, get the outputs of the pilots, etc.

By default, in LHCbDIRAC, Site Directors are configured to monitor pilots every 10 cycles. They also wait 10 cycles before submitting in a LRMS queue that failed, and do not fetch the outputs of the pilots. Additionally, one cannot control the submission operations, despite the many parameters a Site Director contains. Indeed, a Site Director stops generating and submitting pilots for 10 cycles in the LRMS queues that have no more slots available. Lastly, LHCbDIRAC administrators have configured the minimum duration of the Site Director cycles to 120 seconds.

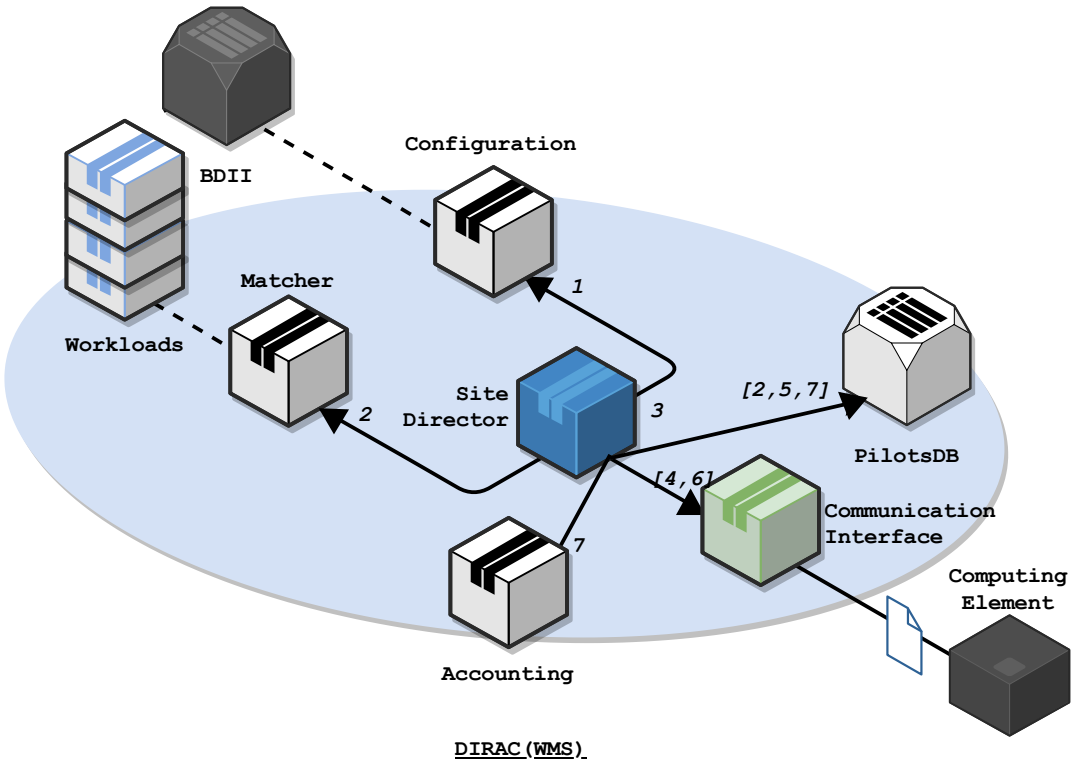

Figure 3.1 – An iteration of a Site Director: steps to manage Pilot-Jobs on grid computing resources.

Limitations due to the grid architecture

We carried out an analysis in DIRAC for the LHCb experiment to emphasize the different limitations inherent to the grid architecture that could cause latencies and prevent to submit as many pilots as needed to run jobs. Since we cannot profile the production environment, we draw on the DIRAC command-line interface, the web application, as well as the log files to get insight into the Site Directors. Raw data and results from the analysis are publicly available (Boyer, 2021a).

The *Accounting* service of the web application provides the average number of jobs processed by pilot per day and per CE during a month (Figure 3.2). In the context of LHCb, most of the pilot handles a single job, despite pilots have been designed

to fetch and run multiple jobs. Indeed, getting an accurate value of the time left allocated to a pilot is a complex operation due to the grid heterogeneity. Site managers work with various LRMS types and versions and adjust specific features differently. Therefore, LHCbDIRAC administrators prefer to limit the number of jobs that a pilot can process, to avoid aborting the jobs that would run out of time. Thus, a Site Director generally submit a pilot per waiting job.

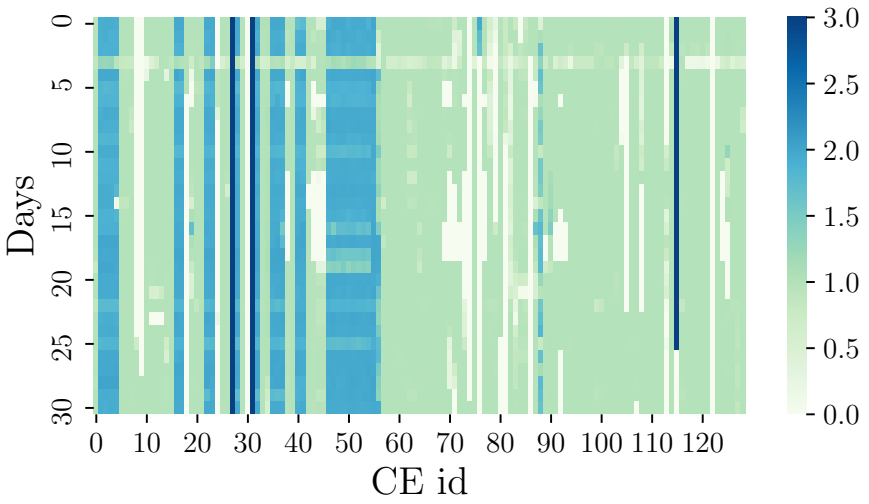

Figure 3.2 – Average number of jobs processed per pilot during a month, classified by the CE that was used to submit them

The web application also presents the time that pilots take from their submission to their installation on a WN. According to the records of 3000 pilots installed on 33 Sites, this duration is not immediate, and generally vary from 165 seconds (1-quantile) to 1719 seconds (3-quantile) (Figure 3.3). Indeed, many VOs are competing for limited computing resources on different Sites. LRMS of the Sites may put the pilots on hold when they arrive, while other VOs are using WNs. Figure 3.3 also contains the time that 3000 jobs take from their arrival in DIRAC to their installation on a WN via a pilot. The median duration to effectively bind a waiting job to a pilot is about 92 seconds, while the median duration to send and execute a pilot on a WN is 309 seconds. The medians demonstrate that jobs are rarely processed by pilots that were generated for this purpose, outlining the importance of always having waiting pilots in the LRMS queues.

The web application contains a configuration page with the parameters of LRMS queues. LRMS queues limit the number of pilots, running and waiting, by means of

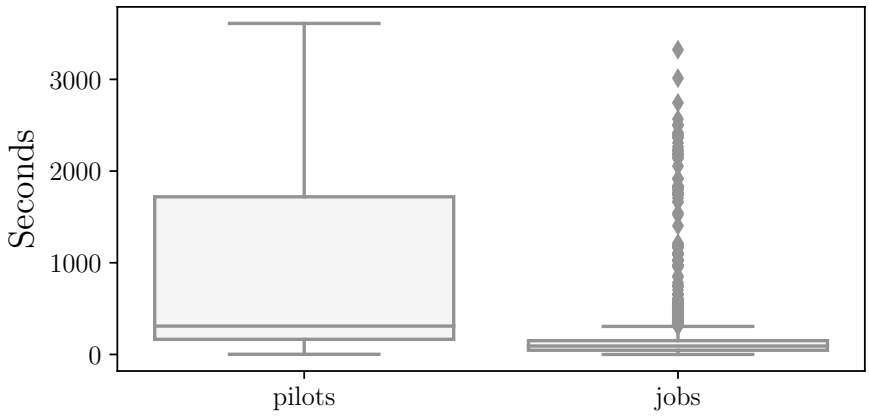

Figure 3.3 – Duration, in seconds, from the pilot generation to the pilot installation on a WN at the left; Duration, in seconds, from the job arrival to the job matching at the right

two parameters: (i) $max\ pilots$ the maximum number of pilots coming from a given queue, and bound to a VO, an LRMS can handle simultaneously; (ii) $max\ waiting\ pilots$ the maximum number of pilots, bound to a VO, that a LRMS can hold in a given queue simultaneously. The number of waiting jobs in LHCbDIRAC is often significantly superior to the $max\ pilots$ values of the LRMS queues.

Issues bound to the infrastructure remain unsolvable, as modifying the architecture in place is not a possible option. Therefore, Site Directors cannot submit as many pilots as necessary to quickly process the jobs, nor reach and maintain $max\ pilots$ in the LRMS queues. Thus, we should focus on continuously submitting pilots to maintain $max\ waiting\ pilots$ in the LRMS queues.

Limitations due to the Site Director itself

Through this part, we analyze whether the Site Director limits the production of Pilot-Jobs by itself and the reasons of such limitations if they exist. The DIRAC command-line interface allows us to get a summary of the status of the pilots, classified by CE, at a certain moment. There are keys for every active CE, and each of them contains a list of status associated with the number of pilots currently in this state. By repeating the process every 5 minutes, we get plots describing the activities of the pilots associated with a specific CE or LRMS queue through time (Figure 3.4). Plots only describe the activities of the pilots at a certain point in time, but we consider this sufficient to get

a grasp of the limitations of the Site Director. In the same way, pilots can pass from *Waiting* to *Running* in less than 5 minutes, meaning some of them can only appear as *Running* on the plots, but this should not significantly impact the plots.

We notice that $max\ waiting\ pilots$ is reached but rarely maintained in most cases. The Site Director bound to *LRMS3 queue1* did not submit any pilot for 2 hours, whereas no pilot was queued, and running pilots were decreasing through time. We can observe similar behavior in *LRMS2 queue1* and *LRMS1 queue1* even if the latter one is less noticeable. The web application can provide information about errors that could have occurred during the submission process, but nothing was reported for the studied queues during this period. Thus, the limitation must come from the execution of the Site Director.

In the LHCb context, each Site Director is bound to specific Sites and to a specific CE type to minimize the number of LRMS queues to manage per Site Director. Its execution is recorded in a distinct log file where we can extract additional information. Each file consists of a suite of logs relative to the execution of multiple cycles. Each log contains a date as well as a message that can constitute a landmark to extract information of interest. Information about the configuration such as the Sites, the types of CEs supervised, the number of jobs, and waiting pilots at a specific moment always appear first. Content about the submission and the monitoring activities can show up afterward. To study the logs, we developed an analysis tool that draws on repeated and common messages and their dates across the files. Its purpose is to extract useful data from a given log file and summarize them into different graphs such as Figure 3.5.

Figure 3.5 describes the execution's length of some Site Directors handling different CE types. The Site Director managing ARC CEs (0) can spend around 6000 seconds to make a cycle while it can take 500 seconds in (1) and (2). This difference can vary according to the number of pilots managed by the Site Directors, the type of the supervised resources, their location, and their capabilities. In this example, (1) manages slightly more pilots than (0), which indicates a potential issue in ARC resources that we are going to investigate in Section 3.2.2.

On all the plots, one cycle out of ten exceed the minimum cycle duration set to 120 seconds, despite they deal with distinct types of CEs. These specific cycles

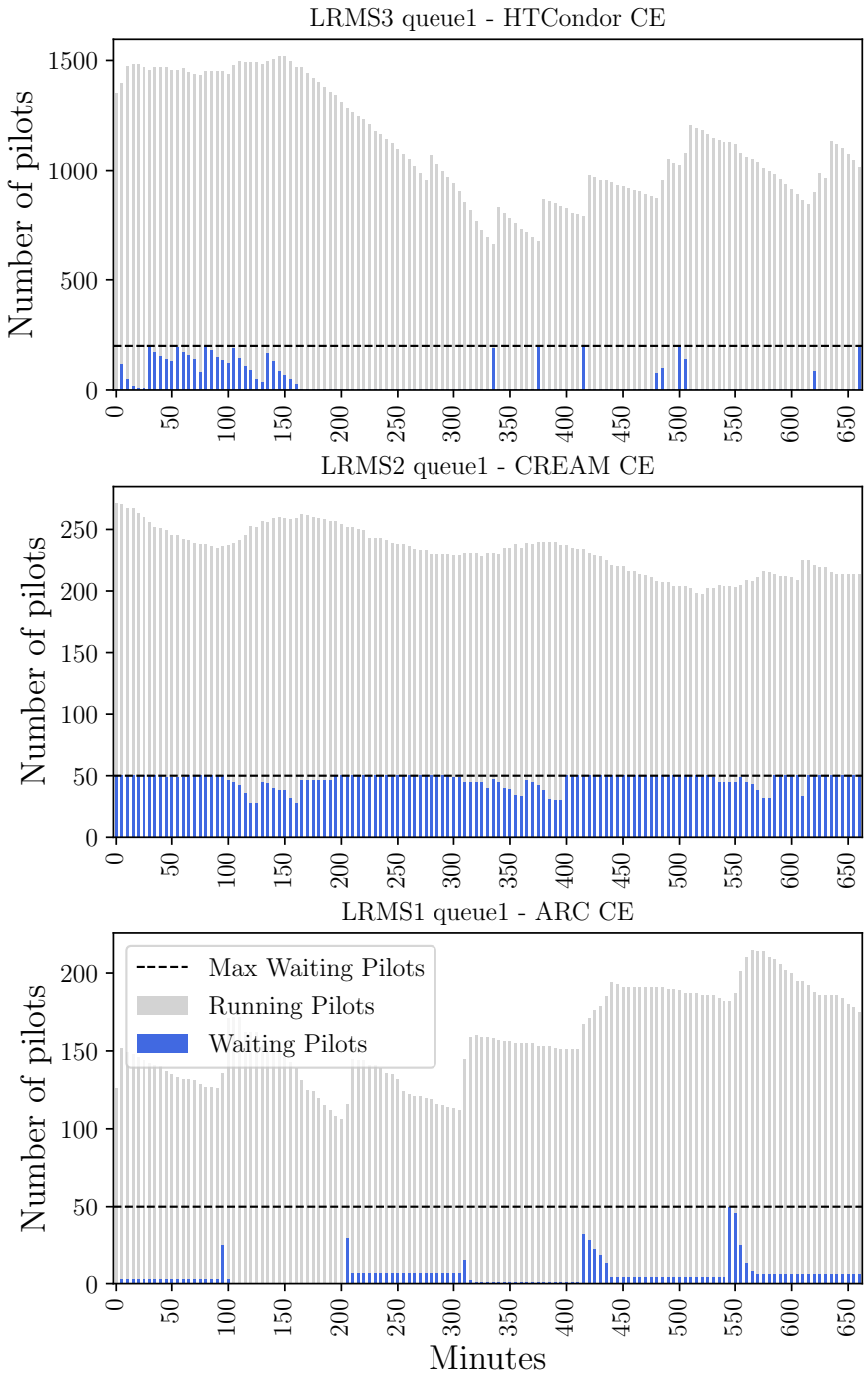

Figure 3.4 – Status of the pilots supervised by three specific LRMS queues for 12 hours

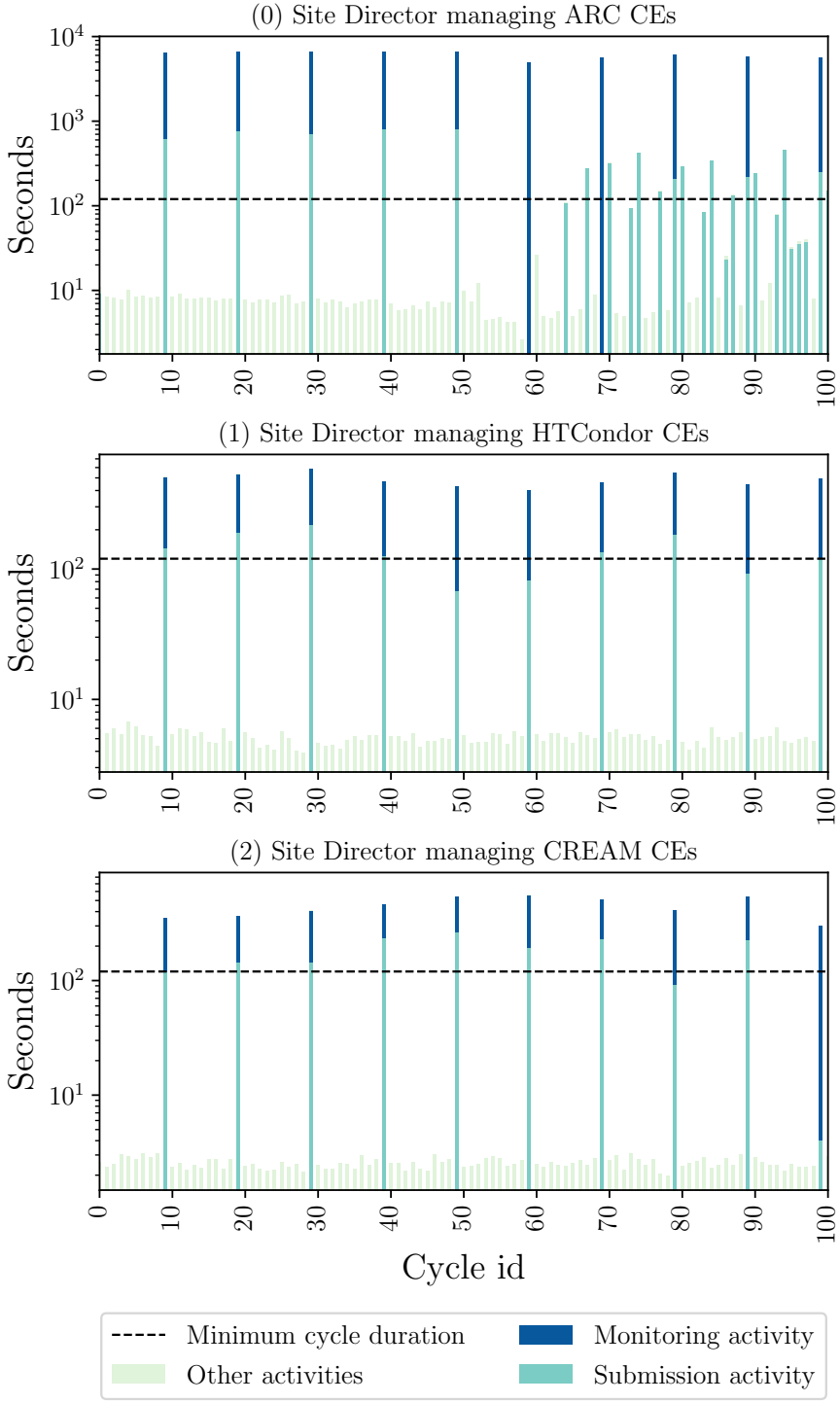

Figure 3.5 – Duration of the cycles and activities of three Site Directors (120 seconds minimum

execute the submission of the pilots followed by the monitoring step, which is time-consuming, while other cycles perform short operations. The monitoring task is the longest operation according to Figure 3.5, and would probably explain some of the Site Director limitation seen in Figure 3.4. Indeed, while monitoring the pilots, the Site Director cannot generate and submit new pilots to fill in the LRMS queues.

One could think about isolating the monitoring part of the Site Directors into a specific agent. On the one hand, this would prevent the stops occurring in the execution of the Site Director and would ease the Pilot-Job submission. On the other hand, it would only shift the monitoring issues elsewhere and would continue affecting the Site Directors. In the same way, administrators could instantiate new Site Directors to split resources across them. Having one Site Director per CE would likely provide better results, but would partially help since it would make the maintenance part difficult. Indeed, in the context of LHCb, we have hundreds of CEs, administrators would not be able to manage so many Site Directors.

By mapping the log messages with their location within the source code, we noticed that the communication between the DIRAC server and the CEs represent the longest operations. Therefore, optimizing communication methods could probably decrease the submission and monitoring duration, prevent stops in the submission of the pilots and thus, could help to maintain $max\ waiting\ pilots$ in the queues. We are going to study several approaches in Section 3.2.2.

Better sharing the workloads between cycles could also ease the submission of pilots on a more frequent basis. In (0), the submission of pilots is more frequent for a limited time, and we observe that the submission duration decreases when shared between cycles. This only occurs when $max\ waiting\ pilots$ is maintained for more than 10 cycles in some queues.

The combination of both Figure 3.4 and 3.5 suggests that the submission of pilots on a more frequent basis would help to maintain $max\ waiting\ pilots$ on the short term on the one hand. On the other hand, it would likely increase the number of running pilots and thus the monitoring period that would finally stop the submission of new pilots for a while and would decrease the number of pilots available again. The main idea would be to improve the monitoring process in order to decrease its duration and submit more frequently and, consequently, reach and maintain

$max\ waiting\ pilots$ finding a balance between the number of pilots to submit and the monitoring time.

3.2.2 Performance improvements of the DIRAC Site Director

We propose three different approaches to submit a larger number of pilots: (i) communicating with multiple CEs in parallel; (ii) better employing the interfaces of the CEs; (iii) configuring Site Directors to submit pilots more frequently.

Parallel communication with the Computing Elements

DIRAC provides communication interfaces to communicate with different CEs. Such interfaces take the form of plugins wrapping the necessary tools to connect to a specific type of CE. They allow DIRAC services, and especially Site Directors, to interact with the underlying LRMS queues and pilots. These communication interfaces include methods to submit pilots to a given CE, kill pilots, get their outputs and/or their statuses. Operations rely on communication with remote resources and require several seconds or even minutes to get responses.

A Site Director sequentially communicates with the CEs, via the communication interfaces, to submit pilots and monitor them. Moreover, the Site Director can administer tens of CEs containing hundreds of pilots, that would involve a large number of requests to remote resources. To minimize duration first, one should privilege parallel treatments and bulk operations.

Submitting pilots involves communication with the *PilotsDB* database containing the pilot identifiers and their statuses. A Site Director reads the database before generating pilots for a given LRMS queue, and updates the pilot database after finishing a submission in this LRMS queue. Because submissions are dependent from each other, we cannot simply process submission in each LRMS queue in parallel, and thus, we focus on monitoring.

Monitoring pilots from different CEs simultaneously would probably decrease the waiting time to get remote data. As various communication interfaces exist to interact with the different types of CEs and that new types often appear, we have decided to tackle the issue at the Site Director level to preserve interfaces and avoid maintaining

too many pieces of code. Classical approaches to make an application parallel include processes and threads.

Multiple processes would allow the Site Director to manage multiple CEs in parallel. However, they would mainly depend on the number of available CPUs on the DIRAC server and would not decrease the waiting time between requests. Multiple threads should, in theory, take advantage of multiple CPUs. However, as DIRAC has been written in Python, it has to deal with the Global Python Interpreter (GIL) ("Global Interpreter Lock", 2022). The GIL enables concurrency by preventing multiple threads from executing Python bytecodes at once, which does not benefit CPU-bound operations. Nevertheless, the interpreter releases the lock on I/O operations such as reading and writing in a file or connections to external resources, which is adapted to our needs. Indeed, the monitoring task would imply IO-bound threads. Connections to the CEs would create an opportunity to switch between threads and would minimize the waiting time in the program execution.

Figure 3.6 presents a Site Director requesting the status of the pilots from three different CEs, first sequentially, then using multi-threads. Each communication interface performs little CPU tasks before and after the connection, while the central part represents the waiting time due to the connection. We expect threads to switch during I/O operations to avoid the program to stop, which would result in better execution time.

Optimizations in the communication interfaces

Even though getting pilot status in each CE simultaneously would ease the monitoring of the pilots and, thus, allow the submission of a larger number of them, it remains incomplete. Indeed, CEs may interact with hundreds or even thousands of pilots, and some of the communication interfaces could be better optimized. Some of them do not exploit all the features of the underlying CEs. We have been focused on ARC, CREAM and HTCondor resources that LHCbDIRAC mainly leverages to deal with inner LRMSs.

In Section 3.2.1, we noticed an issue in the Site Directors dealing with ARC resources. Some of them were taking up to 6000 seconds to monitor a small number of pilots. The communication interface of the latter does not involve bulk methods

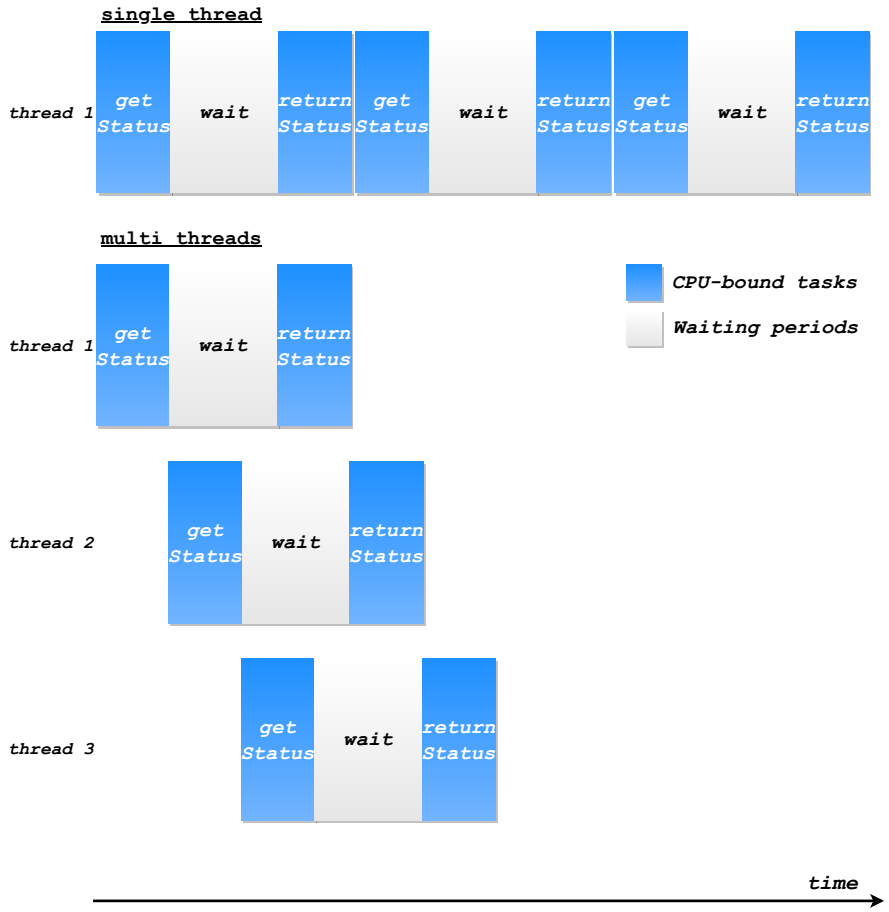

Figure 3.6 – Schema of a sequential execution of the monitoring task at the top; schema of a multi-threads execution of the monitoring task at the bottom

and creates one request per pilot, which can result in a long execution time. Yet, the ARC documentation mentions the presence of such methods grouped into a module named `JobSupervisor` ("ARC Job Supervisor", 2014), able to gather pilot identifiers and perform a single connection to cancel and clean them, renew their credentials, get their status and their outputs. While its integration within the interface would remove the overhead generated by the sum of several single requests, a too large number of pilots supervised would also generate timeouts. Thus, we split the pilots into mid-size chunks as input of the `JobSupervisor` to efficiently use it. Furthermore, as mentioned in Section 2.2.1, ARC developers are planning to drop GridFTP services, in favor to the A-REX services. Many instances have already started the transition, and some of them already dropped GridFTP services. The current communication interface does not implement necessary components to communicate with the A-REX services. Therefore, we created a second interface, inheriting from the current one, able to automatically identify the services in place in a given ARC instance thanks to a `ComputingServiceRetriever` module. The module takes an instance name in input and provides services and queues available. We redefined the submission process using it and favor A-REX services when available.

Additionally, we have worked on the CREAM communication interface and especially the proxy renewal frequency. Indeed, a pilot requires a proxy to interact with DIRAC, mainly to fetch a job to run. A proxy has a limited lifetime and can expire while the pilot waits for available resources in a LRMS queue, which can lead to its abortion. To address this issue, before getting the status of a pilot, most of the communication interfaces perform a check of the proxy lifetime left and renew it if necessary. The communication interface attached to CREAM does not perform this checking and renews the proxies of chunks of pilots in multiple requests every cycle involving the monitoring, which remains unnecessary. Renewing them every n cycles, n being larger than m the number of cycles to wait before invoking the monitoring would reduce the amount of time spent to monitor the pilots on this kind of CEs occasionally. We set n to 600 cycles by default, and we assume it would be always sufficient.

Finally, we have focused on the way DIRAC gets pilot outputs from HTCondor CEs. The communication interface can interact with local and remote HTCondor schedd. By default, and contrary to ARC and CREAM, the server hosting the communication interface automatically receives outputs once pilots end, asynchronously. This mechanism can generate too many output files on the file system. Moreover, DIRAC may

interact with multiple instances of HTCondor. To avoid erasing output files sharing the same name, DIRAC developers decided to randomize output locations at every submission and did not store these locations. Hence, the communication interface has to perform a `find` command to get pilot outputs on the server, which can be cumbersome. To address these issues, we integrated an option to use the HTCondor *spool* mechanism, which allows to manually and synchronously retrieve the outputs. Outputs are downloaded on demand only, copied to a database for a certain number of days and deleted from the file system. We also based the output location on deterministic attributes such as the CE name, the HTCondor job identifier, and a unique pilot stamp.

Pilot-Job Submission Pace

A Site Director regulates the number of pilots to submit in a given LRMS queue according to: (i) the number of running and waiting pilots, related to this queue, at a given time ($pilots$); (ii) the limit values it can support, namely $max\ pilots$ and $max\ waiting\ pilots$. Thus, we have:

$$\begin{aligned} pilots\ to\ submit = min((&max\ pilots - pilots), \\ &(max\ waiting\ pilots - waiting\ pilots)) \end{aligned} \tag{3.1}$$

The Site Director computes this number before each submission in a given queue to fill in every slot. To avoid having too many submissions of pilots that could slow down the monitoring process afterward, DIRAC developers chose to stop computing the number of slots available in a given LRMS queue for 10 cycles once slots have been filled. Thus, a Site Director waits for 10 cycles, a minimum of 1200 seconds, before submitting to the given LRMS. There are two main problems in this approach: (i) there is no mechanism to balance the submissions of pilots between different cycles and they often occur before the monitoring operation, which creates an overused cycle; (ii) LRMS queues can quickly install pilots on WNs and could get new ones in less than 1200 seconds. These problems probably explain the lack of submitted pilots sometimes, previously seen in Section 3.2.1. Therefore, we have introduced a new configuration option that intends to tune the number of cycles to wait before

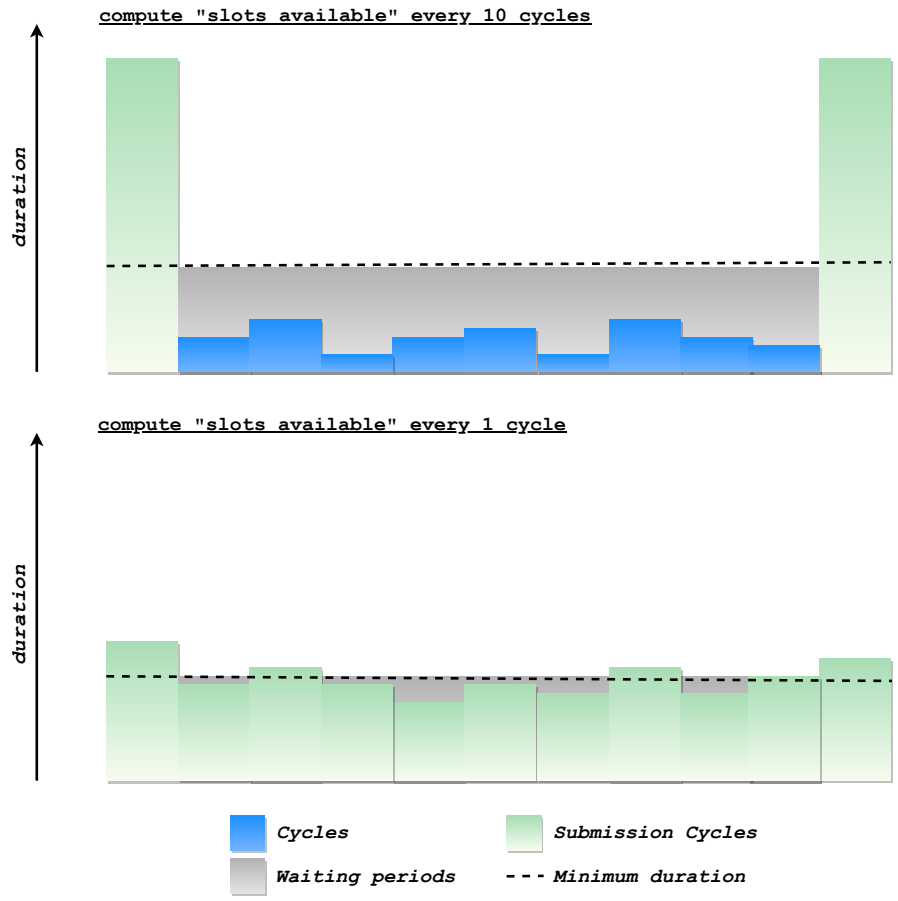

Figure 3.7 – Schema of the duration of the cycles when the number of slots available is computed every ten cycles at the top; schema of the duration of the cycles when the number of slots available is computed every cycle at the bottom

computing the number of slots available in LRMS queues. This would allow us to split the submission operation between the different cycles and, combined with the monitoring optimizations, would better meet the demanding pressure. Figure 3.7 emphasizes the benefits of such an approach.

3.2.3 Performance assessment of the DIRAC Site Director

To provide a complete assessment of the changes we introduced within the Site Director, we performed a series of analysis. We studied individual changes first and then a group of Site Director over a long period.

Assessment of the individual changes

We measured the changes that we described in Section 3.2.2 to assess their distinct contribution. From a DIRAC client, we wrote programs involving multi-threads and communication interface developments to get the pilot statuses.

First, we study the benefit of multi-threads integrated within the Site Directors. A program computes the monitoring process, both in parallel and sequentially. It supervises different types of CEs, handling a diverse number of pilots. Figure 3.8 summarizes 10 program executions in a plot, both sequentially and in parallel. The more CEs the program monitors, the larger the gap between both methods from what we can observe. However, the duration does not increase linearly.

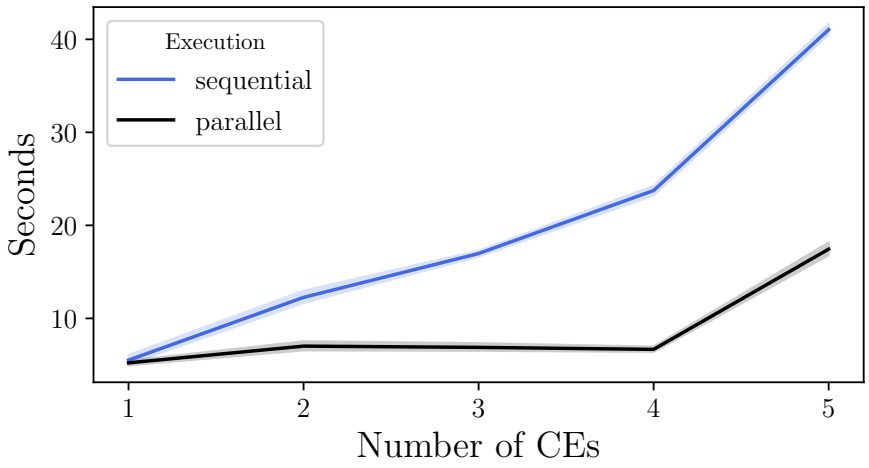

Figure 3.8 – Mean duration, in seconds, that a Site Director spends to monitor tens of pilots managed by a range of CEs: from 1 to 5; along with error bars representing the standard deviation

Figure 3.9 provides an average of the 10 program runs and details about each CE involved. The length of the sequential execution, $total_{seq}$, corresponds to the sum of every CE_n, defined as the time spent by a CE to get the pilot statuses. On the other hand, the parallel version duration, $total_{par}$, is close to CE_1, which is the longest one. The standard deviation remains low and results demonstrate the efficiency of the threads and confirm this choice in this context.

We also measured changes brought to the communication interfaces. We evaluated the `JobSupervisor` integration into the ARC interface. First, the evaluation program performs single requests as it was originally the case, and then bulk requests,

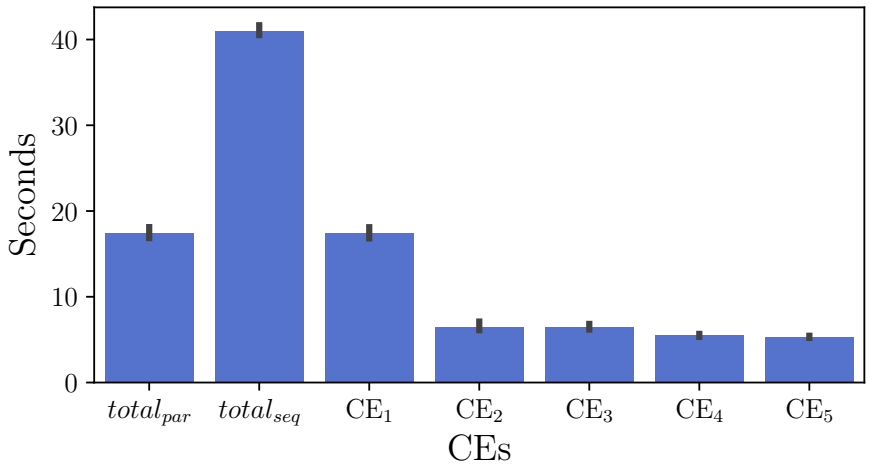

Figure 3.9 – Mean duration, in seconds, that a Site Director spends to monitor pilots in different Sites, managed by different CEs; along with error bars representing the standard deviation

leveraging the `JobSupervisor`, to get the pilot statuses. The program execution was launched three times to get an average of the results as well as a standard deviation. Three CEs from distinct Sites, each of them handling 47 pilots, were available during the experiment. The results appear in Figure 3.10. We can observe a significant improvement in these CEs. Indeed, in all of these cases, it takes less than a second to monitor the pilots using the `JobSupervisor` while it can reach 17 seconds for the same treatment employing a request per pilot. The higher the number of pilots, the larger the gap between the processes. By computing a linear regression on a CE, we can obtain a theoretic time to process 500 pilots. For instance, based on the available data, CE_1 would spend 82 seconds to monitor such a number of pilots using a request per pilot while a bulk operation would take 3.7 seconds.

Changes related to CREAM depend on the number of pilots and the time spent to renew a proxy. However, Figure 3.11 estimates the time that a Site Director would spend to renew the proxies of a certain number of pilots through CREAM resources, and offers a brief idea of the time that could be saved in some cycles. The program renews the proxies of 1 to 10 pilots on two CEs from different Sites five times. The duration scale between the CEs is varying because of their location and their capabilities. We could set up better and more accurate means to renew the proxies of the pilots, but the support of CREAM has ended and resources of this kind should progressively disappear. Furthermore, they would not be so worthwhile in terms of running time.

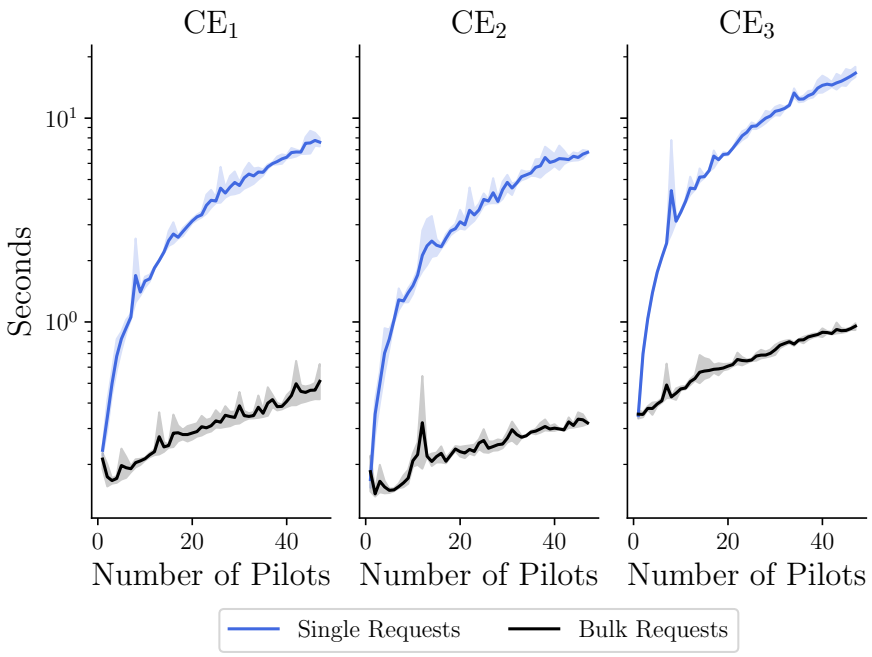

Figure 3.10 – Mean duration, in seconds, of the single and bulk requests in ARC resources along with error bars representing the standard deviation

Getting the benefits that the deterministic path setup to retrieve HTCondor pilot outputs could bring is a complex operation that would be meaningless. LHCbDIRAC administrators disabled the parameter to get the pilot outputs from the Site Directors in production. Similarly, evaluating the gain of the option to finetune the submission pace individually would depend on too many external factors such as the underlying occupancy of the LRMS queues.

The next sections of our work assessment aim at providing insights: (i) about the evolution of the throughput of the jobs and the pilot submission frequency over time; (ii) about the involvement of the changes. Such points are necessary to answer our initial research question: Does the improvement of the Pilot-Job provisioning tool speed up the Pilot-Job submission frequency and, by extension, the throughput of the jobs on grid resources? This will be plainly answered and discussed in Section 3.2.4, after the presentation of our results.

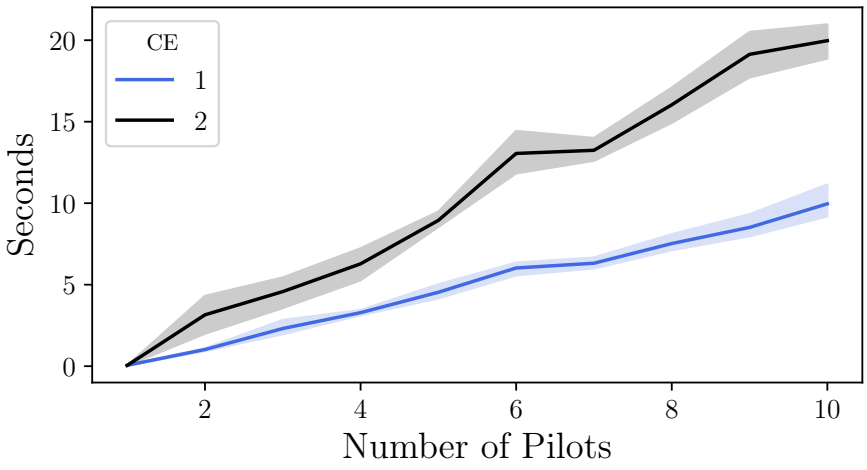

Figure 3.11 – Mean duration, in seconds, of the proxy renewal in CREAM resources along with error bars representing the standard deviation

Evaluation of the LHCbDIRAC production environment: experimental conditions

We analyzed the Site Directors of the LHCbDIRAC production environment for 12 months to assess the contributions mentioned in this chapter in a real use case. Raw data, results and figures are publicly available (Boyer, 2021a). We introduced three different phases:

- Phase1: does not include any of the change (from week 1 to week 17).
- Phase2: include changes related to the monitoring task (from week 18 to week 41).
- Phase3: include changes related to the submission pace control (from week 42 to week 56).

Getting the overall benefit of the changes is a complex operation. Indeed, Site administrators can add, replace, remove computing resources, LRMS queues and CEs over time. They can also modify the scheduling policies; grant a varying number of slots to VOs. Besides, DIRAC administrators can tweak parameters related to the Site Directors such as the Sites and CEs they manage.

Over the analysis period, Sites, as well as all the Site Directors, were modified. We removed data related to the Site Directors instantiated, largely modified, or deleted

during the three phases, as it would skew the study (Table 3.1). This mainly concerns Site Directors managing deprecated CREAM CEs. We also removed Site Directors managing pilots via SSH, as we did not observe such a use case through this chapter, and they deal with a minor part of computing resources.

Site Dir.	Reason
SD_2	Added during Phase3
SD_5	Changes of CEs (ARC to HTCondor) since Phase3
SD_8	Changes of CEs (CREAM to HTCondor) since Phase2
SD_9	Changes of CEs (CREAM to HTCondor) since Phase2
SD_{11}	Removed during Phase3
SD_{13}	Removed during Phase2
SD_{15}	Changes of CEs (CREAM to ARC) since Phase2
SD_{16}	Removed during Phase3
SD_{18}	Removed during Phase2
SD_{23}	Changes of CEs (CREAM to HTCondor) since Phase2
SD_{25}	Manage SSH CEs
SD_{26}	Manage SSH CEs
SD_{27}	Manage SSH CEs
SD_{28}	Manage SSH CEs
SD_{29}	Manage SSH CEs
SD_{30}	Added and removed during Phase2
SD_{31}	Added during Phase3

Table 3.1 – Site Directors from the LHCbDIRAC production environment removed from the study

Thus, the study includes 13 out of 31 Site Directors managing pilots within a total of 65 out of 77 Sites: five of them manage ARC CEs, five others interact with HTCondor CEs, and the last ones supervise CREAM CEs. We configured the pilot submission pace according to the type of CEs that Site Directors deal with: ARC Site Directors submit every cycle while CREAM and HTCondor Site Directors submit every 5-6 cycles. Selected Site Directors were present during the three phases and received small adjustments over time such as LRMS queues added or removed. Table 3.2 classifies the number of LRMS queues managed by the Site Directors and the changes that occurred during the different phases.

Site Directors	Phase1	Phase2		Phase3	
		Added queues	Removed queues	Added queues	Removed queues
SD_1	17	6	0	0	0
SD_3	5	9	0	8	0
SD_4	2	1	0	0	1
SD_6	3	0	0	0	0
SD_7	2	1	1	0	0
SD_{10}	1	0	0	0	0
SD_{12}	9	0	0	0	0
SD_{17}	29	4	0	0	0
SD_{14}	2	0	0	0	0
SD_{19}	4	0	0	0	0
SD_{20}	5	0	0	0	0
SD_{21}	4	0	0	0	0
SD_{22}	8	10	0	3	1

Table 3.2 – Selected Site Directors from the LHCbDIRAC production environment and their evolution over the different phases

Evaluation of the LHCbDIRAC production environment: evolution of the throughput of the jobs and the Pilot-Job submission frequency

First, we assessed the number of CPU seconds processed per second (Figure 3.12). The metric corresponds to the number of jobs running simultaneously within the Sites observed. It also represents the number of CPUs that the LHCb VO can exploit in parallel on grid resources to process the workload.

In Figure 3.12, we averaged values per week. Dashed and dotted lines designate the limits between the phases. We also grouped and averaged the values by phase. LHCbDIRAC processed 40.86% more CPU seconds per second in Phase3 (80306) than in Phase1 (57010). The largest increase occurs between Phase2 and Phase3 (21.64%) but the gap between Phase1 and Phase2 remains meaningful (15.81%). Yet, we cannot notice a clear distinction between the different phases. While the standard deviation is relatively small in Phase1 (5917) and Phase3 (3675), it is larger in Phase2 (9121). Values from Phase2 almost linearly increase from about 55,000 CPU seconds/second in the first weeks to about 70,000 in the last ones.

Moreover, we studied the evolution of the number of pilots submitted - by the selected Site Directors - per hour, averaged per week. Figure 3.13 illustrates the num-

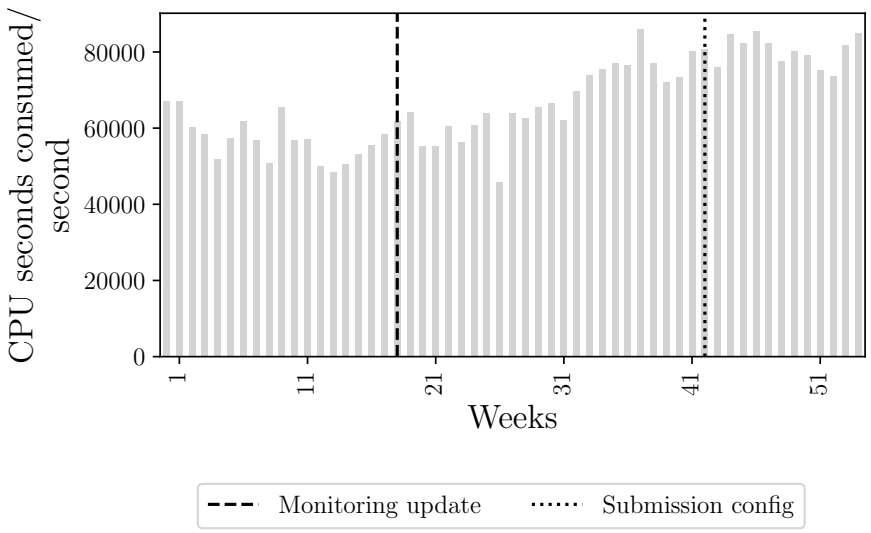

Figure 3.12 – CPU seconds processed per second by LHCb jobs on selected Sites over 12 months, averaged per week

ber of successfully submitted pilots per hour along with the pilots that Site Directors failed to submit.

Dashed and dotted lines delimit the phases. We grouped and averaged the values by phase. Site Directors intended to submit 60.23% more pilots per hour in Phase3 than in Phase1, with an increase of 33.10% between Phase2 and Phase3, and 20.38% between Phase1 and Phase2. Nevertheless, the evolution of the number of successfully submitted pilots per hour is much lower: values remained constant between Phase1 and Phase2 (+1.29%) and rose between Phase2 and Phase3 (16.90%). We can also see peaks in Phase2 and Phase3 that were not reached in Phase1. The evolution of failed submission per hour is much more noticeable in the figure (+671.41% between Phase1 and Phase3), but remains highly variable within the phases: the standard deviation is about 725 in Phase1, 1325 in Phase2 and 2693 in Phase3. As we can observe, there is no clear association between errors and phases, and most errors seem concentrated at the end of Phase2 and the beginning of Phase3.

Additionally, to get a more accurate idea of the involvement of the changes, we focus on the status of the pilots for small periods during the phases. Using the command-line interface, we got the status of every pilot on all the observed Sites between 144 and 432 times per phase. Figure 3.14 presents the distribution of waiting pilots per Site Director over time, classified per phase.

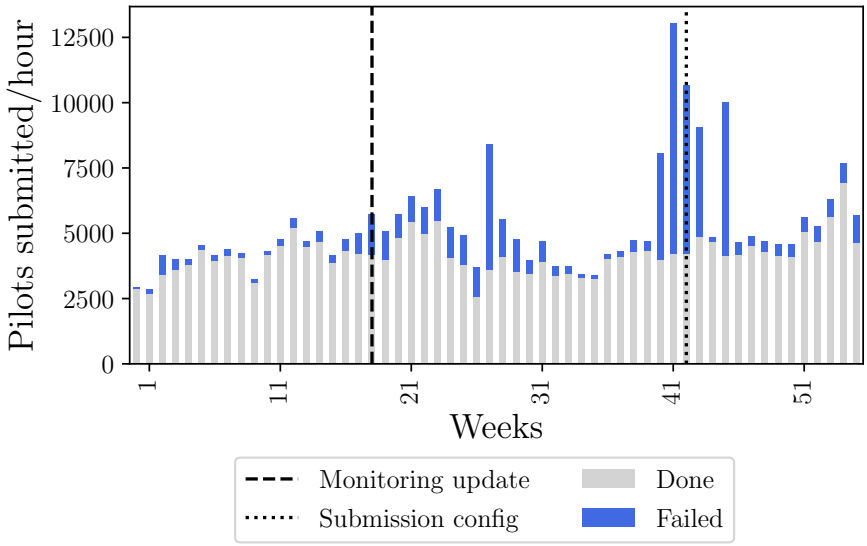

Figure 3.13 – Number of pilots submitted per hour, averaged per week

We summed the median value of each Site Director per phase. LRMS queues contain 15% more waiting pilots between Phase1 and Phase2, and 53% more between Phase2 and Phase3. We can also observe less variability in Phase3 than in Phase1, indicating that values are relatively stable in general. Most of the ARC Site Directors in (0) manage many more waiting pilots in Phase3 than in Phase1 (between +144% to +805%) except SD_4 (-76%). HTCondor Site Directors in (2) propose similar results: more waiting pilots in Phase3 than in Phase1 with less variability (between +14.60% and 62.88%). SD_{22}, which has handled a growing number of queues in Phase2 and Phase3 according to Table 3.2, monitors many more waiting pilots since the beginning of the Phase3 (+584% compared to Phase1). At the same time, the number of waiting pilots coming from SD_{20} declined in Phase3. On the contrary, in (1), CREAM Site Directors interact with a more stable number of waiting pilots, but we can notice a global decrease in Phase3 compared to Phase2. Results from SD_{10} remained low and almost identical through the different phases. To complete these data, we also analyzed the distribution of running pilots per Site Director over time, classified per phase in Figure 3.15.

We also summed the median value of each Site Director per phase. LRMS queues contain 21% more running pilots between Phase1 and Phase2, and 68% more between Phase2 and Phase3. In (0), there were many more running pilots in resources managed by SD_1, SD_3 and SD_4 in Phase3 than in Phase1 and Phase2 (between +43% and +172%).

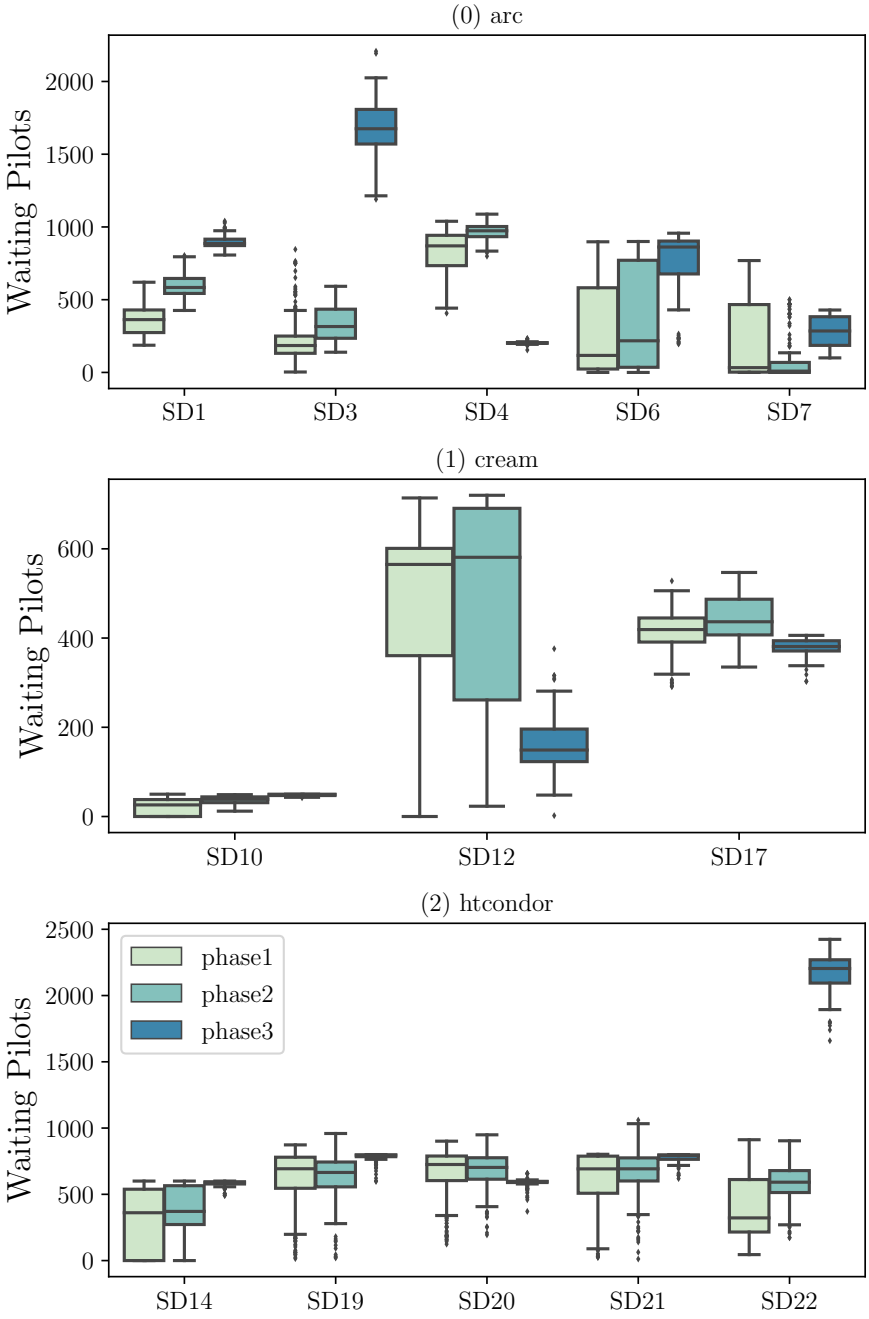

Figure 3.14 – Distribution of waiting pilots per phase, classified by Site Director. (0) gathers Site Directors managing ARC CEs; (1) gathers Site Directors dealing with CREAM CEs; (2) gathers Site Directors interacting with HTCondor CEs

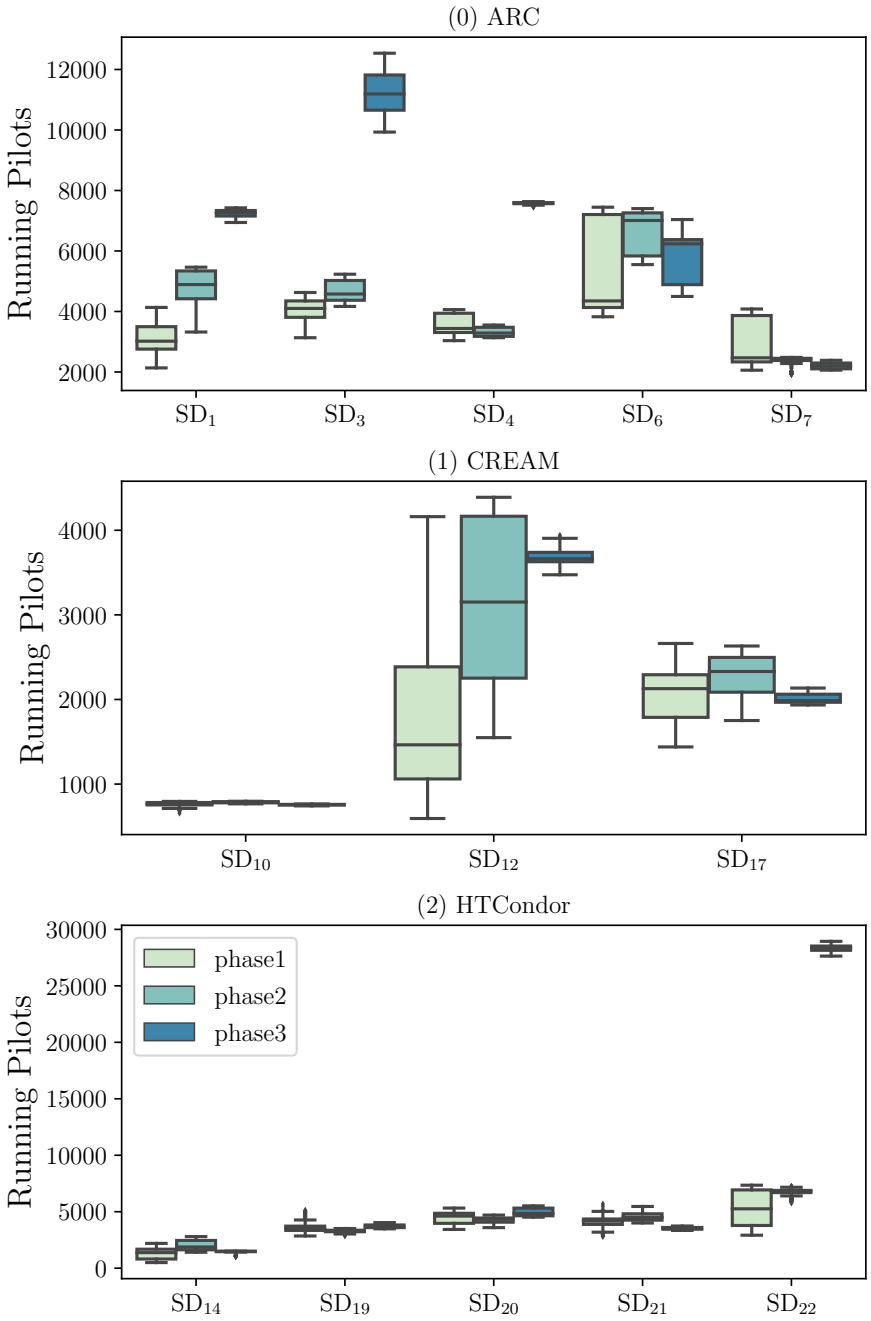

Figure 3.15 – Distribution of running pilots per phase, classified by Site Director. (0) gathers Site Directors managing ARC CEs; (1) gathers Site Directors dealing with CREAM CEs; (2) gathers Site Directors interacting with HTCondor CEs

Yet we noticed in Figure 3.14 that SD_4 monitored fewer waiting pilots in Phase3. Similarly, SD_7 monitored less running pilots in Phase3 than in Phase1, while it had more waiting pilots in Phase3. Most of the results in (2) rose between 4% and 8% between Phase1 and Phase3, which remains significant according to the large number of pilots these Site Directors manage. As in Figure 3.14, SD_{22} monitored a larger number of running pilots in Phase3 than in Phase1: more than 25,000 running pilots at the same time, which represents the highest number of pilots monitored at the same time by a Site Director. SD_{20} have also less running pilots. To finish, in (1), SD_{10} results did not change through time, while results from SD_{12} grew (+150%) and the ones from SD_{17} went down (-6.53%).

While general results provide meaningful data about the evolution of the LHCb-DIRAC production environment through time, they do not furnish any information about monitoring duration and submission pace. After some details dealing with failed submissions, involvements of the chapter contributions are analyzed in depth in Section 3.2.3.

Evaluation of the LHCbDIRAC production environment: details about failed submissions

The changes made could have potentially generated new errors: in this part, we provide additional information about errors at the Site Director level to strengthen the answer to our initial research question in Section 3.2.4.

Table 3.3 provides details about the number of errors that occurred at submission within each Site Director. Errors mainly concern 8 out of the 13 Site Directors of the study. We grouped Site Directors that got a small number of errors in the *Others* category. Most of the Site Directors encountered more issues in Phase2 than in Phase1, but the number fell in Phase3. Among them, SD_1 got even fewer errors in Phase3 (400) than in Phase1 (663). Only three Site Directors got more errors in Phase3 than in Phase2: SD_{19}, SD_{20}, and SD_{21} that interact with a common Grid Site. During Phase2, 65% of the errors were bound to three Site Directors: SD_1, SD_3 and SD_{21}; while they were only bound to 10% of the total number of errors in Phase3. Indeed, in Phase3, SD_{19} was bound to 54% of the errors.

	Phase1	Phase2	Phase3
SD_1	663	5536	400
SD_3	87	5529	171
SD_4	52	1862	667
SD_{17}	654	1582	4220
SD_{19}	958	2166	16976
SD_{20}	872	1743	5567
SD_{21}	596	5973	2637
SD_{22}	505	1232	613
Others	218	325	72
Total	4605	25948	31323

Table 3.3 – Number of failed submission per Site Director, classified per phase

Evaluation of the LHCbDIRAC production environment: involvement of the contributions

Decreasing the monitoring period is the primary way that we choose, in this chapter, to go towards an efficient Pilot-Job provisioning on grid resources. We investigated the evolution of the monitoring through the different phases and especially between Phase1 and Phase2. We extracted the monitoring duration from the logs of the Site Directors - each log file contains around 150 values - and we computed the mean for each Site Director and each phase. We coupled monitoring values with successfully submitted pilots from Figure 3.13 and, thus, we obtained Figure 3.16.

Monitoring duration dropped down from Phase2 (-49% on average). Indeed, in Phase2, SD_{22} spent the longest time monitoring its pilots: 245 seconds on average; while 6 Site Directors spent more than 850 seconds monitoring their pilots on average during Phase1. In Phase3, monitoring duration slightly went up (+64% compared to Phase2 on average, but -22% compared to Phase1): SD_{22} spent 423 seconds, on average, which remained the longest monitoring duration. SD_4 was the only Site Director to increase its monitoring duration through time (from 30 seconds in Phase1 to 85 seconds in Phase2 and 112 seconds in Phase3). In Phase3, 7 out of 13 Site Directors spend more than 120 seconds, the minimum cycle duration by default, against 12 during Phase1.

Significant results on the monitoring duration do not always involve an increase in the number of pilots successfully submitted per hour, according to Figure 3.16. Indeed, 7 Site Directors submitted fewer pilots per hour from Phase2 despite spending

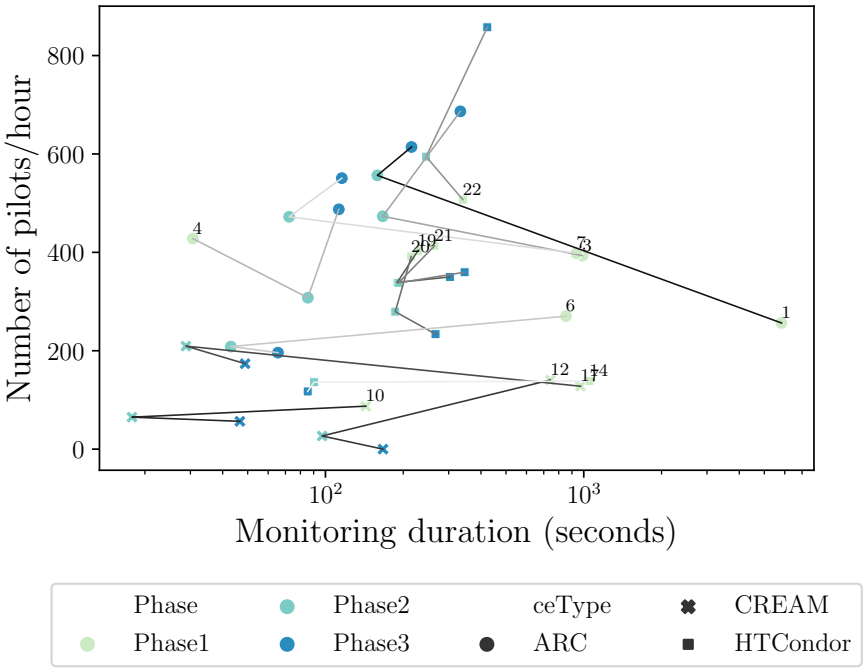

Figure 3.16 – Evolution of the number of pilots successfully submitted per hour (mean), function of the monitoring duration of the Site Directors through the different phases. Each point represents a Site Director; its shape, a type of CE managed; and its color, a certain phase. Phases of a same Site Director are associated via a line

less time on the monitoring operations. It involves 2 of the 3 CREAM Site Directors, one ARC Site Director and 4 of the 5 HTCondor Site Directors. Changes have a more significant impact on ARC Site Directors than on CREAM and HTCondor Site Directors. HTCondor Site Directors provide diverse results: (i) SD_{14} submitted almost the same number of pilots over time; (ii) SD_{17} submitted more pilots per hour in Phase2, but the value went down from Phase3; (iii) SD_{22} submitted many more pilots in Phase2 and especially in Phase3; (iv) SD_{19}, SD_{20} and SD_{21}, that work on the same Site, submitted fewer pilots over time but handled many errors according to Table 3.3.

Increasing the number of pilots submitted per cycle is the second way - and inherent to the monitoring duration - that we choose to go towards an efficient Pilot-Job provisioning on grid resources. After configuring the pilot submission pace for each Site Director, we investigated the number of pilots submitted per cycle per Site Director and per phase (Figure 3.17). We extracted the number of pilots submitted per cycle from the logs, which contain around 1500 values per log file.

In Phase1, Site Director submitted a median value of 0 pilot per cycle as submission occurred every 10 cycles, which correspond to outliers on the plot. ARC Site Directors, after Phase3, submitted every cycle: they were all able to submit between 10 and 20 pilots per cycle (median value) and outliers were, in general, smaller than in Phase1. Results from CREAM and HTCondor Site Directors are less meaningful, which was expected as they have been configured to submit pilots every 5 or 6 cycles. Only SD_{10}, SD_{14} and SD_{22} submitted pilots more often.

3.2.4 Discussions

Does the improvement of the Pilot-Job provisioning tool speed up the Pilot-Job submission frequency and, by extension, the throughput of the jobs on grid resources?

The capacity of LHCb to leverage Grid Site resources considerably rose over a year, and is probably the result of a combination of numerous factors. According to Figure 3.16, changes applied on the monitoring step have, overall, considerably decreased the monitoring duration of the Site Directors, and especially the ones managing ARC CEs and the ones having a large number of queues. Indeed, contributions seem to have a greater impact on Site Directors sharing these characteristics according to Figure 3.10

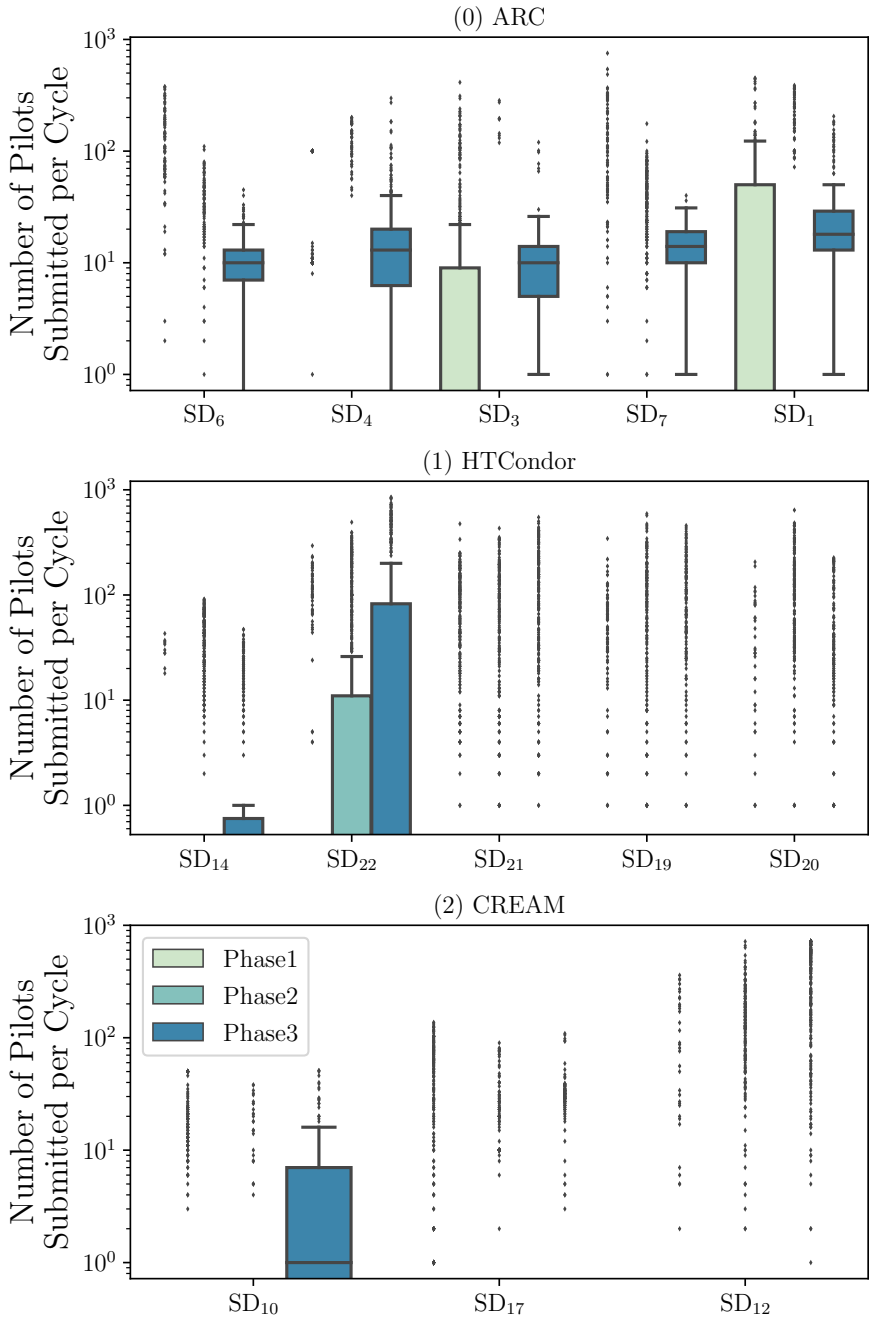

Figure 3.17 – Evolution of the number of pilot submitted per cycle (median), classified by Site Directors and phases

and 3.8. Effects on the monitoring seem to last: Site Directors managing additional queues and hundreds or even thousands more pilots have spent less or almost the same time monitoring the pilots (see SD_1, SD_3 and SD_{22} on Figure 3.16). Yet, the changes have had almost no visible impact on the number of pilots submitted per hour. Indeed, changes have decreased the duration of the cycles to a value close to 120 seconds, the minimum duration of a cycle, but did not change the submission pace. Thus, changes have only affected Site Directors that spent a long time monitoring pilots, blocking the submission process, such as SD_1.

Tweaking the submission pace after decreasing the monitoring duration has been substantial. Site Directors submitting pilots more often, such as the ARC ones, have better shared the workload between their cycles (Figure 3.17). They have filled LRMS queues with waiting pilots more rapidly as we can observe in Figure 3.14. Thus, Grid Sites have had more pilots at their disposal for available resources, which probably explains the rise of running pilots (Figure 3.15).

Some external factors have complemented the results of the solutions. Many Site Directors have managed a various number of LRMS queues through time according to Table 3.2. Overall, there was an increase of LRMS queues: 42 added against 3 removed. Site administrators have also tweaked $max\ pilots$ and $max\ waiting\ pilots$ over time according to the log files, but we did not keep track of the fluctuations. Therefore, we cannot know with accuracy whether Site Directors have maintained $max\ waiting\ pilots$ in the different LRMS queues. Such variations have probably significantly modified the monitoring duration of the Site Directors along with the number of pilots that they have supervised (see SD_3 and SD_{22} in Figure 3.14, 3.15 and 3.16). Our changes have helped Site Directors to support a growing number of LRMS queues that better handle the generation of new pilots and maximize the use of new computing resources.

Errors have diluted the effect of the changes. Indeed, some Site Directors have failed to submit a large number of pilots. Many failures have been likely independent from our contributions: SD_{19}, SD_{20} and SD_{21} have failed to submit plenty of pilots, but errors occurred for a limited period and concerned the same Grid Site (Table 3.3). Overall, we noted a larger number of submission failures after the changes. Further investigations in the logs suggest errors within the CEs and queues but remain unclear. These errors were already existing in Phase1: changes have probably eased

the submission process, even within these queues, which highlights them.

The role of the following components remains unclear, and their effects are difficult to measure. First, the number of jobs per pilot has changed over time (Figure 3.2). In general, the more jobs a pilot handles, the longer it remains on a WN. In practice, it also depends on the execution duration of the jobs, which leads to the second point. Jobs and pilots have had a varying execution duration. A short job triggers: (i) the generation of a pilot that can take several minutes before running; (ii) the allocation of a WN for a limited time - a few seconds for instance -; which is not efficient. In theory, the repetition of a large number of long jobs would reduce the risk of having unused resources: Site Directors would have more time to generate pilots and they would be replaced less often in the WNs. We did not keep track of such information over time as it would represent a massive amount of data.

Some Site Directors - that have been positively affected by our contribution and not significantly impacted by external factors that we can measure - have not produced and submitted plenty of pilots: SD_4, SD_{10} and SD_{12} for instance. Some Site Directors were probably already submitting $max\ waiting\ pilots$ in the LRMS queues, such as SD_{10}, which had the same number of waiting and running pilots over time (Figure 3.14 and 3.15) Other Site Directors were likely supervising a considerable number of running pilots, which modified $max\ waiting\ pilots$. Indeed, the value of $max\ waiting\ pilots$ depends from $max\ pilots$ such as:

$$\begin{aligned} max\ waiting\ pilots = min(&max\ waiting\ pilots,\\ &max\ pilots - running\ pilots) \end{aligned} \tag{3.2}$$

This is probably the case for SD_4 and SD_{12} that have a low and steady number of waiting pilots in Phase3 (Figure 3.14) while having a large and steady number of running pilots (Figure 3.15).

The combination of all these component have likely increased and stabilized the number of waiting pilots in the LRMS queues (Figure 3.14) and have even allowed LHCb to exploit a larger number of allocations on Grid Sites (Figure 3.15 and 3.12).

This study has mainly focused on grid resources, especially in the context of the

LHCb experiment, but remains appropriate for any pool of distributed and shared computing resources aggregated into high-performance clusters and clouds. VOs relying on supercomputers providing external connectivity, or cloud resources orchestrated by LRMS, could reuse parts of the presented content.

Future Directions and Challenges

Here, we provide further recommendations on topics not covered by this study, to help VOs depending on the distributed architecture to better exploit shared computing resources.

CEs may have different numbers of pilots to handle, as well as heterogeneous performances to execute a similar operation, as we can observe in Figure 3.9, 3.10 and 3.11. To better leverage the multi-threads integration, DIRAC administrators can bind Site Directors to multiple CEs, and especially CEs having similar performances. Indeed, the time spent in each thread should be equally distributed to avoid having one thread spending more time than all the other together, which would result in a duration close to the sequential one. One could propose an automated way to balance the number of Site Directors and the number of CEs per Site Director to minimize the duration of the cycles while maximizing the submission frequency.

Again for DIRAC administrators, in Section 3.2.1, we have demonstrated that jobs were rarely processed by pilots generated for this purpose, which may call into question the need to check the presence of jobs, before instantiating the pilots. Indeed, a Site Director only generates a limited number of pilots according to the jobs available that could run in a given resource, as well as the number of free slots. When pilots are finally running, this limited number may not reflect the number of jobs previously available as other pilots from different Site Directors may have already processed the jobs. This case happens when the number of waiting jobs is inferior to the number of free slots in the resources, which is rarely the case in production but can occur in specific Sites. Thus, one could imagine a Site Director strategy consisting in continuously sending pilots in the queues, which would slow down production rates in the case that pilots do not fetch any job. The challenge in this approach lies in the various scheduling policies of the Sites: while some Sites can prioritize VOs submitting the most pilots, others can favor pilots that effectively run for a long time.

DIRAC developers should maintain discussions with CE developers to integrate features that could ease some WMS operations. For instance, HTCondor clients are able to submit multiple pilots in a single command, whereas ARC clients have to generate n requests to send n pilots. After discussing with VOs, ARC developers have recently started to work on a small extension to support such a feature.

We have explained in Section 3.2.1 that getting an accurate value of the time left allocated to a pilot is a complex operation due to the various batch systems composing the grids: different types, versions and configurations. In combination with the time left value, pilots need to get the "power" of the CPU, namely how efficient is a CPU to run an application of interest. Indeed, two different processors will likely not spend the same time running the same application. Solutions such as DIRAC Benchmark have been developed to provide an estimation of the CPU power for Monte-Carlo simulations in the LHC context, and will be studied in Section 3.3. Designing efficient ways of getting accurate CPU power and time left values would help better exploiting the allocations by fetching the most adapted jobs.

Lastly, we encourage VOs that would like to conduct similar research with other systems to record and collect data about pilots, jobs and also about configuration changes in the Sites in order to get a clear overview of the impact of the external factors. The task remains challenging because (i) information from BDII is not always reliable and (ii) the grid architecture involves many operations that are hard to follow: there are external and internal issues, maintenances, upgrades involving various actors every day.

3.2.5 Summary

Through Section 3.2, we have demonstrated the importance of continuously improving Pilot-Job provisioning mechanisms to better exploit shared and distributed heterogeneous computing resources. After exposing the advantages and limitations of the Pilot-Job paradigm (Section 3.2.1), we explored one of the main Pilot-Job provisioning tools: the DIRAC Site Director in the context of the LHCb experiment (Section 3.2.1). For 12 months, we analyzed 13 Site Directors managing 65 grid sites on WLCG, dealing with 57,000 LHCb jobs simultaneously. By introducing multi-threading within the Site Directors and including CE-specific performance improvements, we speeded

up the monitoring mechanism: the duration of the activity dropped down (-22%). Additionally, we better shared the workloads between the cycles of the Site Directors to generate a fewer number of pilots more frequently (Section 3.2.2).

We conducted performance studies, repeated multiple times over one year, to prove the efficiency of every change made (Boyer, 2021a). We measured an overall gain of 18.41% of the number of pilots successfully submitted per hour, which represents 728 additional pilots per hour. We also recorded an increase of 40.86% of the number of jobs processed simultaneously per second, which means that WLCG is simultaneously in charge of 80,300 LHCb jobs (Section 3.2.3). Thus, this study enables the generation of more pilots to meet the increasing demand for computing power. In this context, computing power is essential to refine the analysis and increase the statistics and the confidence that we can place in the discoveries made thanks to the LHC, which will affect our understanding of the universe.

Future studies should focus on further increasing the Pilot-Job submission frequency (Section 3.2.4). Automatically fine-tuning the parameters of the Site Directors - the number and the nature of the queues that they supervise - depending on the load on the Sites would be a solution. A complementary solution would consists in adapting the submission rate according to the scheduling policies of the Sites to optimize the priority of the Pilot-Jobs within the queues. Working on a CPU benchmarking solution providing accurate CPU power estimations for various processors would constitute another approach to maximize the use of the allocated resources, and will be discussed in Section 3.3.

3.3 DIRAC Benchmark

3.3.1 Presentation of DIRAC Benchmark

As reported in Section 2.4.2, DIRAC Benchmark is an open-source fast and synthetic HEP CPU benchmarking solution created in 2012 (DB12). In 2016, Charpentier designed an LHCb-specific version of DIRAC Benchmark: DB16. In Section 3.3.1, we first describe how DIRAC Benchmark is used within DIRAC, and then we present its main features and limitations.

DIRAC Benchmark use cases

DIRAC Benchmark is mostly used to (i) fetch jobs appropriate to the underlying computing resources and their constraints; (ii) determine the number of Monte-Carlo simulation events a time-constrained worker node can run.

Three steps are necessary to efficiently run workloads on shared and time-constrained computing resources. First, a Pilot-Job is submitted to the LRMS of a site (Figure 3.18 Step 2). Second, the LRMS, which orchestrates worker nodes, puts a Pilot-Job on hold until a worker node is available (Figure 3.18 Step 3). At this moment, it provides a slot, namely a WN for a certain duration, and executes the Pilot-Job in this slot. Third, the Pilot-Job evaluates the WN before fetching a job from the central WMS of the experiment. This evaluation implies (i) running DIRAC Benchmark to obtain an estimation of the CPU power of the worker node (Figure 3.18 Step 4.1) and (ii) interrogating the LRMS to get the time left (Figure 3.18 Step 4.2). The product of both variables is equal to the CPU work, namely the amount of work in HS06seconds allowed in the slot. Because the CPU work relies on an estimation of the CPU power, a margin of 25% is usually applied to avoid jobs running out of time. In single-core allocations, a Pilot-Job fetches and executes jobs sequentially until 75% of the CPU work has been consumed. In multi-core allocations, jobs are fetched and executed in parallel on the logical cores available.

Main features and issues

DIRAC Benchmark comprises three layers: a kernel, basic features based upon it and more advanced features. The kernel is the computation of the CPU power of a single core. It involves 12,500,000 iterations corresponding to 250 HS06 seconds. During an iteration, the function generates a random variable X following the normal distribution, such as $X \sim \mathcal{N}(10,1)$, and performs several basic operations with it, such as additions and multiplications: the purpose is to artificially replicate the functioning of a Monte-Carlo simulation job. To get a better estimation, the process can be repeated multiple times. The score is then based on the CPU time needed to perform the operations, the number of repetitions done and a constant to normalize the result,

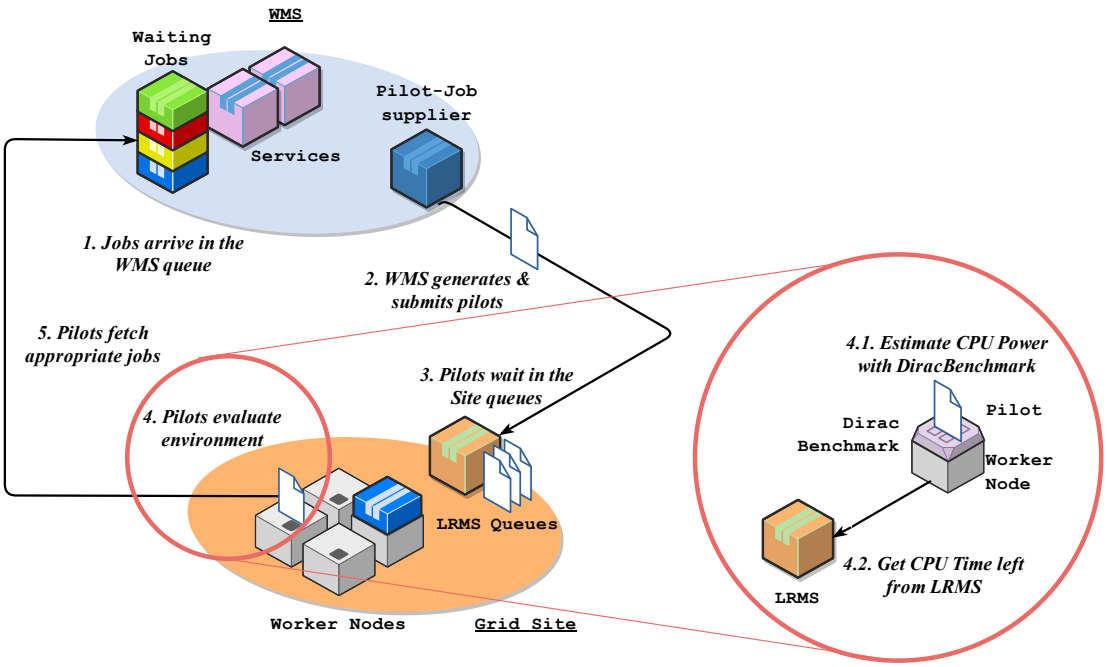

Figure 3.18 – Using DIRAC Benchmark to fetch appropriate jobs on shared and time-constrained computing resources

such as:

$$score = \frac{250 \times repetitions}{CPU_{time}} \tag{3.3}$$

Basic features solely involve the computation of the CPU power on multiple cores in parallel. The function takes a given number of copies that have to be run in parallel m as input and performs a single-core benchmark in m distinct processes. Finally, advanced features rely on the external variables - coming from the WN configuration or from the Machine Job Features (MJF) mechanism introduced by HEPiX in 2016 and now discontinued - to run copies of the benchmark on a whole node or the cores allocated by the LRMS. Figure 3.19 presents a schema representing the features enumerated above.

DIRAC Benchmark has not been updated for years while software and hardware are quickly evolving. The code itself is written in Python 2, deprecated since January 2020. We would need to update the code without creating discrepancies. In addition,

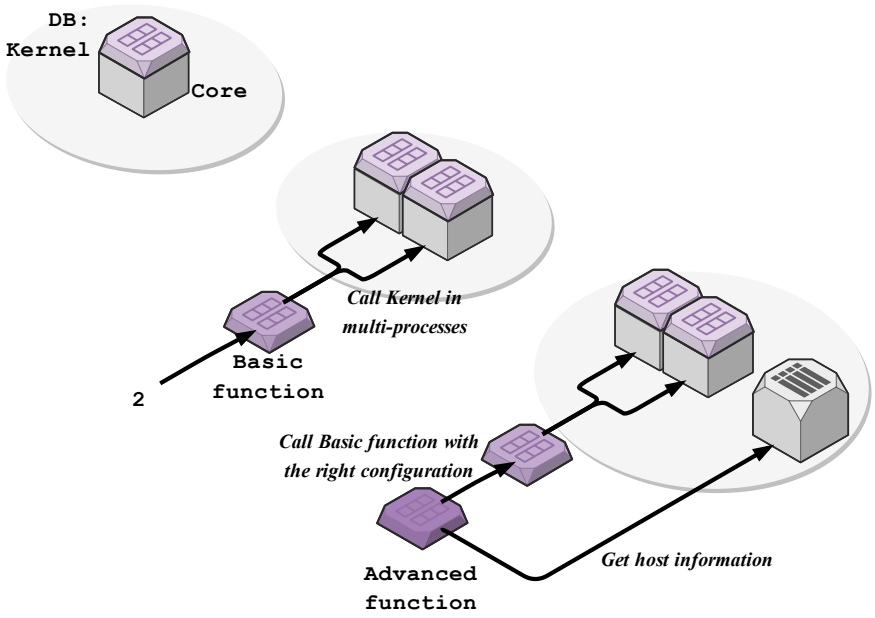

Figure 3.19 – Composition and features of the DIRAC Benchmark

the code consists of a single file, not tested and not formatted in a standard approach, which cannot be imported into other packages. DIRAC developers have decided to copy-paste the code in their WMS to use it. We would need to generate an importable Python package. In Section 3.3.2, we explain how we operated the transition of DIRAC Benchmark to Python 3.

3.3.2 Maintenance and improvement of DIRAC Benchmark

The main challenge of this transition is to make sure that the benchmark will still provide valuable and accurate - reproducible - scores in Python 3 and beyond. Indeed, developers of Python 3 have included numerous optimizations and changes compared to Python 2. The second objective is to provide a portable package so that users can directly import the benchmark into their applications. The initial transition plan includes three steps: (i) providing a tested and formatted code; (ii) rewriting the code and assessing the changes; (iii) generating a portable package.

Setting up a Continuous Integration pipeline

The first step consists in providing tools to maintain a clean and well-tested code before the transition. First, we removed features related to MJF as the project was abandoned. Then, we reformatted the code using Black, a style guide auto-formatter for Python Code compliant with Python Enhancement Proposals 8 (PEP8) ("Black", 2022). We also developed unit tests with pytest ("pytest: helps you write better programs", 2021) in the repository to cover the program and make sure that:

- Each function produces a similar output when given the same inputs multiple times on the same host. As the score relies on variable parameters, a difference smaller than 20% is accepted.
- Each function generates a similar output regardless of the number of repetitions passed as input.
- The score is positive and not unreasonably high, between 0 HS06 and 100 HS06.

Finally, we set up a continuous integration (CI) pipeline based on GitHub Actions ("GitHub Actions", 2022) to better handle future changes in the code. Each commit triggers the CI pipeline, which (i) sets up a consistent environment with Conda ("Conda", 2022); (ii) runs Pylint ("Pylint", 2022) to help enforce a coding standard and highlight programming errors; (iii) runs the new code against the unit tests. To arrive in production, the introduced code has to pass every step of the CI pipeline. To help developers to produce quality code, we also included a pre-commit ("pre-commit", 2022) module in the repository. It automatically formats the code using Black and performs several checks before pushing the code to the repository.

Porting DIRAC Benchmark to Python 3

The second step introduces the transition to Python 3.9. The program only relies on standard Python libraries and contains about 300 lines of code. Changes are minimal and include:

- `print` statements has been followed by parenthesis in Python 3.
- Python 2 `xrange` has been redefined as `range` in Python 3 and has integrated new features. Python 2 `range` has no equivalent in Python 3.

- Python 2 `long` type has been renamed as `int` in Python 3. Python 2 `int` has no equivalent in Python 3.

Nevertheless, such minimal changes generated discrepancies in the results. Indeed, Python 3 has included various optimizations under the hoods. Even though operations integrating Python 3 `int` are slower than the ones employing Python 2 `int`, Python 3 is generally faster than Python 2. Therefore, the Python 3.9 DIRAC Benchmark is faster than the original one, and thus, generally provides a better score for a given machine. To assess the impact of the discrepancies, we originally attempted to set up integration tests within the CI pipeline, to compare the results obtained with different versions of Python. In practice, it was hard to guarantee a repeatable environment for all the executions and the pipeline often failed because the discrepancies were too large between the scores: beyond 20%. In Section 3.3.3, we will present the experiment we conducted to better assess the impact of the discrepancies using various CPU models.

Creating a Python package

Finally, the last step of the transition consisted in creating a Python package to embed the benchmark and make it portable and easy to import in other Python projects. We completely revisited the structure of the project by splitting the command line interface options from the code related to the benchmark and adding configuration files to set up a package. We added a continuous delivery (CD) module to the Github Actions pipeline of the project. The CD module, triggered only when a GitHub tag is created, generates a binary wheel as well as a source tarball from the repository and publishes the files to PyPI ("PyPI", 2022), a recognized repository of Python software. We also added the package to Conda-Forge ("Conda-Forge", 2022), a repository of Conda recipes, to ease the integration of the DIRAC Benchmark in other projects.

In the next sections, we want to make sure that the DIRAC Benchmark scores computed with Python 3.9 will remain correlated to LHCb Gauss tasks. As a first step (Section 3.3.3), we are going to assess the potential impact of the discrepancies observed in Section 3.3.2. We want to correct the discrepancies so that the Python 3.9 version of DIRAC Benchmark would still provide an approximation of the CPU power in HS06, mainly for historical and compatibility reasons. As a second step (Section 3.3.4), we are going to perform a new analysis of DIRAC Benchmark regarding

Monte-Carlo simulation jobs of the LHCb experiment. We will compare the jobs to the corrected Python 3.9 version of DB16, which should correspond to the Python 2.7 version of DB16 at this point. This may result in the creation of a new version of DIRAC Benchmark that would better fit with Gauss than with the HEPSPEC06 benchmark.

3.3.3 Assessment of the DIRAC Benchmark scores: Python 3 versus Python 2 executions

Conditions of the experiment

We conducted an experiment in the LHCb production environment to compare Python 3.9 and Python 2.7 executions of DIRAC Benchmark. The experiment involved computing resources of WLCG, namely 102 CEs involving 83 different CPU models, as well as workloads containing both versions of the DIRAC Benchmark. On average, each grid site was in charge of 10 jobs. A job consists in executing sequentially the Python 2.7 and the Python 3.9 versions of the DIRAC Benchmark 11 times. To be consistent, all jobs run in the same Conda environment and leverage Python binaries coming from CVMFS instead of the local Python binaries. Indeed, for some reason, some grid sites propose custom versions of Python. We collected data related to the benchmark and the host environment. Programs, resources and results are publicly available to facilitate the reproducibility of this work (Boyer and Stagni, 2021). For the sake of clarity in the following figures, we established Table 3.4 providing short identifiers for every CPU model.

Figure 3.20 shows the distribution of the DIRAC Benchmark executions on the available grid sites - we assume that a CE corresponds to a site - per Python version used. The executions are heterogeneously shared between the sites. Indeed, we can observe a median of 99 executions - from 4 to 5 jobs - per site. The top quartile grid sites were in charge of 198 executions while the bottom quartile handled 66 executions. We submitted the same quantity of jobs on every grid site, but some of them - mainly Tier2 and Tier3 sites - spent too much time executing the jobs or jobs failed for external reasons. It is worth mentioning that we can find a similar distribution of the LHCb production jobs among the sites: the experiment is representative of the LHCb production environment.

ID	CPU model	ID	CPU model
A0	AMD EPYC 7282 16-Core Processor	A1	AMD EPYC 7302 16-Core Processor
A2	AMD EPYC 7352 24-Core Processor	A3	AMD EPYC 7451 24-Core Processor
A4	AMD EPYC 7452 32-Core Processor	A5	AMD EPYC 7551 32-Core Processor
A6	AMD EPYC 7551P 32-Core Processor	A7	AMD EPYC 7702 64-Core Processor
A8	AMD EPYC 7702P 64-Core Processor	A9	AMD EPYC 7742 64-Core Processor
A10	AMD Opteron(TM) Processor 6276	A11	AMD Opteron(tm) Processor 4386
A12	AMD Opteron(tm) Processor 6128	A13	AMD Opteron(tm) Processor 6168
A14	AMD Opteron(tm) Processor 6172	A15	AMD Opteron(tm) Processor 6320
I0	Intel Xeon Processor (Skylake & IBRS)	I1	Intel(R) Core(TM) i7 CPU 860
I2	Intel(R) Core(TM) i7-5960X CPU	I3	Intel(R) Xeon Phi(TM) CPU 7210
I4	Intel(R) Xeon(R) CPU 5150	I5	Intel(R) Xeon(R) CPU E5420
I6	Intel(R) Xeon(R) CPU E5520	I7	Intel(R) Xeon(R) CPU E5620
I8	Intel(R) Xeon(R) CPU E5640	I9	Intel(R) Xeon(R) CPU E5645
I10	Intel(R) Xeon(R) CPU L5630	I11	Intel(R) Xeon(R) CPU L5640
I12	Intel(R) Xeon(R) CPU X5650	I13	Intel(R) Xeon(R) CPU X5660
I14	Intel(R) Xeon(R) CPU E5-2420 0	I15	Intel(R) Xeon(R) CPU E5-2450 v2
I16	Intel(R) Xeon(R) CPU E5-2618L v4	I17	Intel(R) Xeon(R) CPU E5-2620 v3
I18	Intel(R) Xeon(R) CPU E5-2620 v4	I19	Intel(R) Xeon(R) CPU E5-2630 0
I20	Intel(R) Xeon(R) CPU E5-2630 v2	I21	Intel(R) Xeon(R) CPU E5-2630 v3
I22	Intel(R) Xeon(R) CPU E5-2630 v4	I23	Intel(R) Xeon(R) CPU E5-2640 v3
I24	Intel(R) Xeon(R) CPU E5-2640 v4	I25	Intel(R) Xeon(R) CPU E5-2650 v2
I26	Intel(R) Xeon(R) CPU E5-2650 v3	I27	Intel(R) Xeon(R) CPU E5-2650 v4
I28	Intel(R) Xeon(R) CPU E5-2650L v4	I29	Intel(R) Xeon(R) CPU E5-2660 v2
I30	Intel(R) Xeon(R) CPU E5-2660 v3	I31	Intel(R) Xeon(R) CPU E5-2670 0
I32	Intel(R) Xeon(R) CPU E5-2670 v2	I33	Intel(R) Xeon(R) CPU E5-2670 v3
I34	Intel(R) Xeon(R) CPU E5-2680 v2	I35	Intel(R) Xeon(R) CPU E5-2680 v3
I36	Intel(R) Xeon(R) CPU E5-2680 v4	I37	Intel(R) Xeon(R) CPU E5-2683 v4
I38	Intel(R) Xeon(R) CPU E5-2695 v4	I39	Intel(R) Xeon(R) CPU E5-2697 v3
I40	Intel(R) Xeon(R) CPU E5-2697 v4	I41	Intel(R) Xeon(R) CPU E5-2698 v3
I42	Intel(R) Xeon(R) CPU E5-2698 v4	I43	Intel(R) Xeon(R) CPU E5-4610 0
I44	Intel(R) Xeon(R) Gold 5115 CPU	I45	Intel(R) Xeon(R) Gold 5118 CPU
I46	Intel(R) Xeon(R) Gold 5120 CPU	I47	Intel(R) Xeon(R) Gold 5218 CPU
I48	Intel(R) Xeon(R) Gold 5220 CPU	I49	Intel(R) Xeon(R) Gold 6126 CPU
I50	Intel(R) Xeon(R) Gold 6130 CPU	I51	Intel(R) Xeon(R) Gold 6132 CPU
I52	Intel(R) Xeon(R) Gold 6140 CPU	I53	Intel(R) Xeon(R) Gold 6148 CPU
I54	Intel(R) Xeon(R) Gold 6226R CPU	I55	Intel(R) Xeon(R) Gold 6230 CPU
I56	Intel(R) Xeon(R) Gold 6238R CPU	I57	Intel(R) Xeon(R) Gold 6248 CPU
I58	Intel(R) Xeon(R) Gold 6248R CPU	I59	Intel(R) Xeon(R) Gold 6252 CPU
I60	Intel(R) Xeon(R) Gold 6354 CPU	I61	Intel(R) Xeon(R) Silver 4110 CPU
I62	Intel(R) Xeon(R) Silver 4114 CPU	I63	Intel(R) Xeon(R) Silver 4116 CPU
I64	Intel(R) Xeon(R) Silver 4210 CPU	I65	Intel(R) Xeon(R) Silver 4214R CPU
I66	Intel(R) Xeon(R) Silver 4216 CPU		

Table 3.4 – CPU models and their identifiers

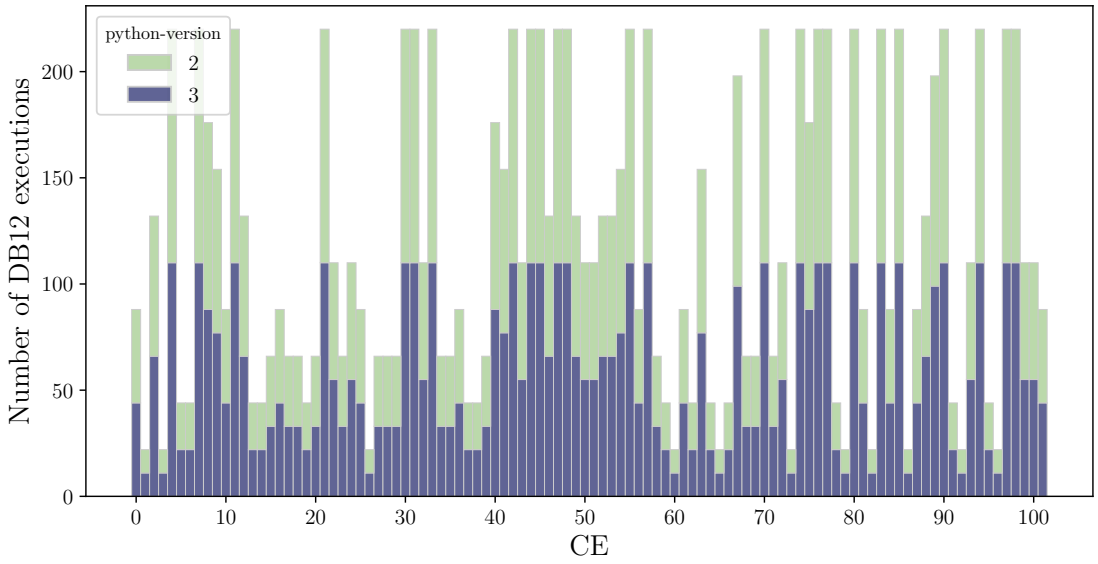

Figure 3.20 – Distribution of the DIRAC Benchmark executions among the grid sites grouped by Python version used: Python3 at the bottom, Python2 at the top.

Figure 3.21 presents the distribution of the DIRAC Benchmark executions on the available CPUs. Sites mainly involve two brands of x86 processors: Intel and AMD. Our studied sample comprises a majority of Intel processors: 80.7% against 19.3% AMD processors. We observe similar proportions with the distribution of the executions: 81% (9614) happened on Intel processors while 19% (2266) occurred on AMD processors. However, executions were not shared equally on every CPU model. Indeed, the median is about 110 executions - namely 5 jobs per CPU model - but the bottom quartile CPU models handled 44 executions while the top quartile was in charge of 220 executions. The `I23-Intel(R) Xeon(R) CPU E5-2640 v3` is way above with 594 executions (27 jobs), probably because it is more present in the grid sites and proposes a better price/performance ratio. We can also notice that both versions of the DIRAC Benchmark are equally shared for a given CPU model.

It is worth noting that even within the same allocation, consecutive executions of the benchmark can produce different results, as illustrated in Figure 3.22. In this example, most of the consecutive executions remain relatively similar - Python 2 executions of DIRAC Benchmark in the job running in `A3` have a standard deviation of 0.3. Nevertheless, some of them are varying significantly - Python 2 executions of DIRAC Benchmark in the job running in `I40` have a standard deviation of 3.22. The exact cause remains unknown but it is likely due to the fluctuating load in the shared

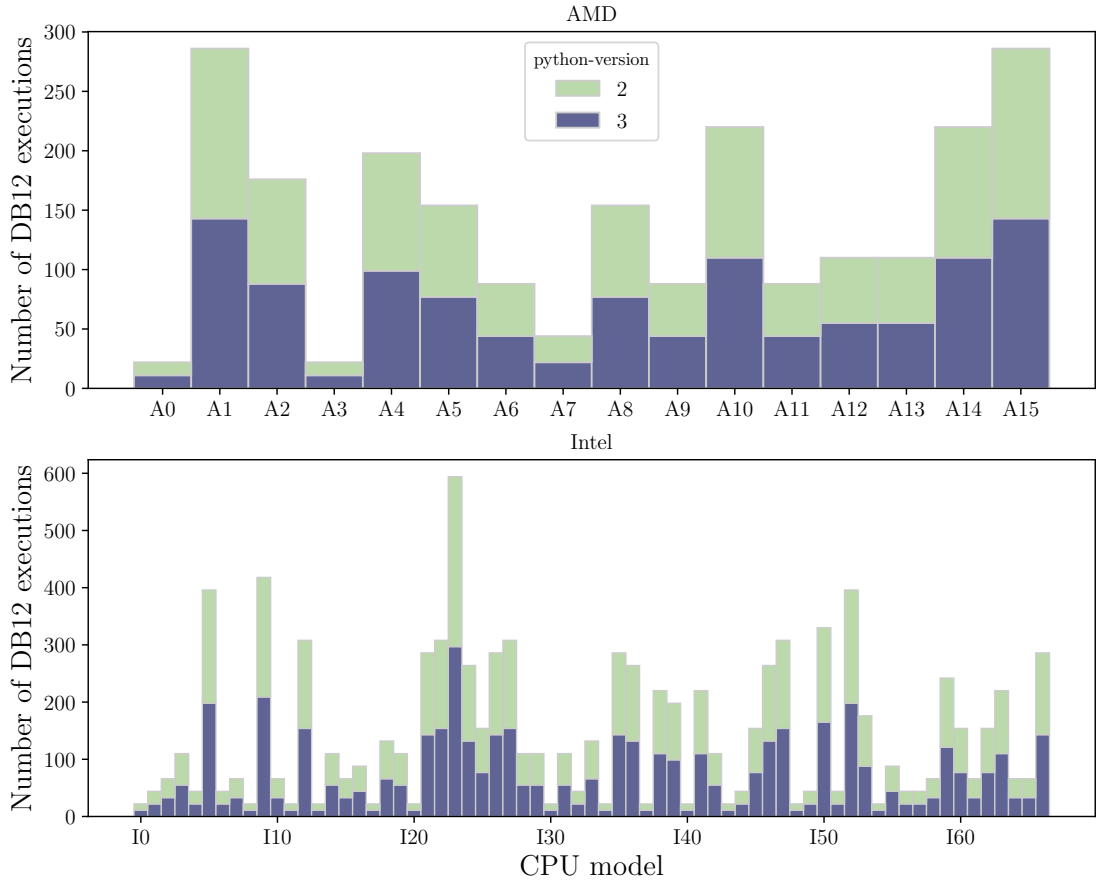

Figure 3.21 – Distribution of the DIRAC Benchmark executions among the CPU models grouped by Python version used: Python3 at the bottom, Python 2 at the top.

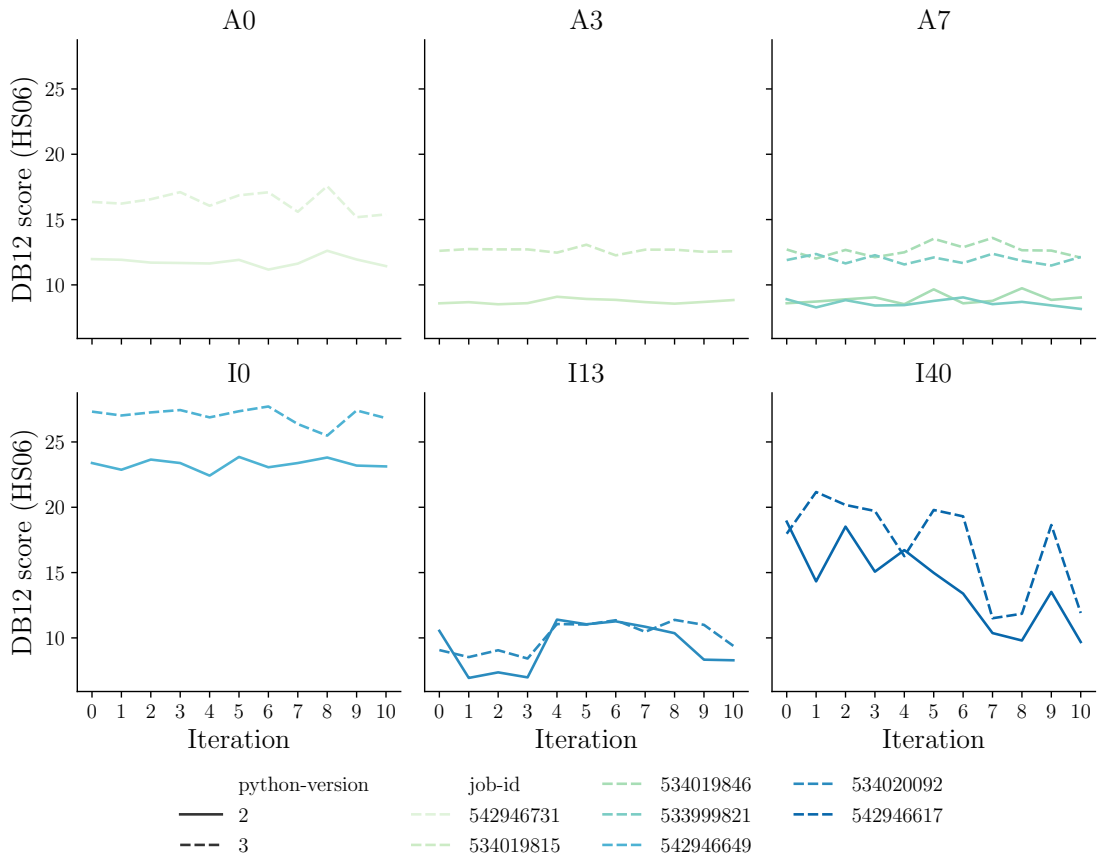

Figure 3.22 – Consecutive executions of DIRAC Benchmark within a same allocation. Each subplot represent executions of jobs on a specific CPU model. Each color represents a job execution, whereas the style of the line indicates whether DIRAC Benchmark was executed with Python 2 or Python 3.

nodes.

Results

In a job, we executed 11 times both versions of the DIRAC Benchmark. To emphasize the discrepancies, we paired scores coming from the same iteration - the Python 2.7 and the Python 3.9 scores bound to an iteration - to generate coordinates. From these coordinates, we designed Figure 3.23 to compare the gap between Python 2.7 and Python 3.9 scores. We can notice that Python 3.9 scores are mostly higher than the Python 2.7 scores. The gap seems to be almost constant across the paired scores. Yet we can still distinguish two different cases: scores coming from AMD processors would need a stronger correction than the scores coming from Intel processors. The

root mean square error of the Python 3.9 scores with respect to the Python 2.7 scores is about 2.86. Next, we will present the considered solutions to correct the discrepancies.

Solutions

To reduce the discrepancies, we designed three solutions that would be simple to set up and easy to update. The first solution consists in applying a scaling factor to the Python 3.9 scores. To define a scaling factor, we computed the ratio Python 2 score/Python 3 score for each pair and selected the median value: 0.85. The root mean square error of the Python 3.9 scores $\times$ 0.85 with respect to the Python 2.7 scores is about 1.52. The solution remains simple in theory, developers would need to update a single value over time. In practice, it would be probably more complex. Indeed, in Section 3.3.3 we distinguished two different cases: Python 3.9 scores on AMD processors were, in general, higher than Python 3.9 scores on Intel processors. As hardware and Python evolve through time, we might need to handle many different cases over time. Such a solution is still not accurate enough, and thus, could not allow managing more complex cases.

To provide better estimations and handle future cases, we produced a second solution: applying a scaling factor based on the underlying architecture and Python version. We split the dataset into two parts: one is gathering the pairs related to AMD processors and the other is bound to the Intel processors. As in the first solution, we computed the ratio Python 2 score/Python 3 score for each pair of each part of the dataset. By selecting the median values, we obtained two scaling factors: 0.86 for Intel processors and 0.71 for AMD processors. The root mean square error of the transformed Python 3.9 scores with respect to the Python 2.7 scores is about 1.19. The second solution is significantly more accurate than the first solution but remains complex to maintain. Indeed, developers would need to maintain a table within the code and would need to conduct a similar experiment each time a new CPU architecture or Python version is introduced.

The last solution consists in performing a linear regression to adjust the Python 3.9 scores. Based on inputs such as the Python 3.9 scores, the CPU model, the CPU MHz, the number of cores available, the average load and the OS, we could determine an approximate value of the Python 2.7 scores. We split the dataset into (i) a training set (30% of the dataset) to train the model; (ii) a cross-validation set to plot the learning

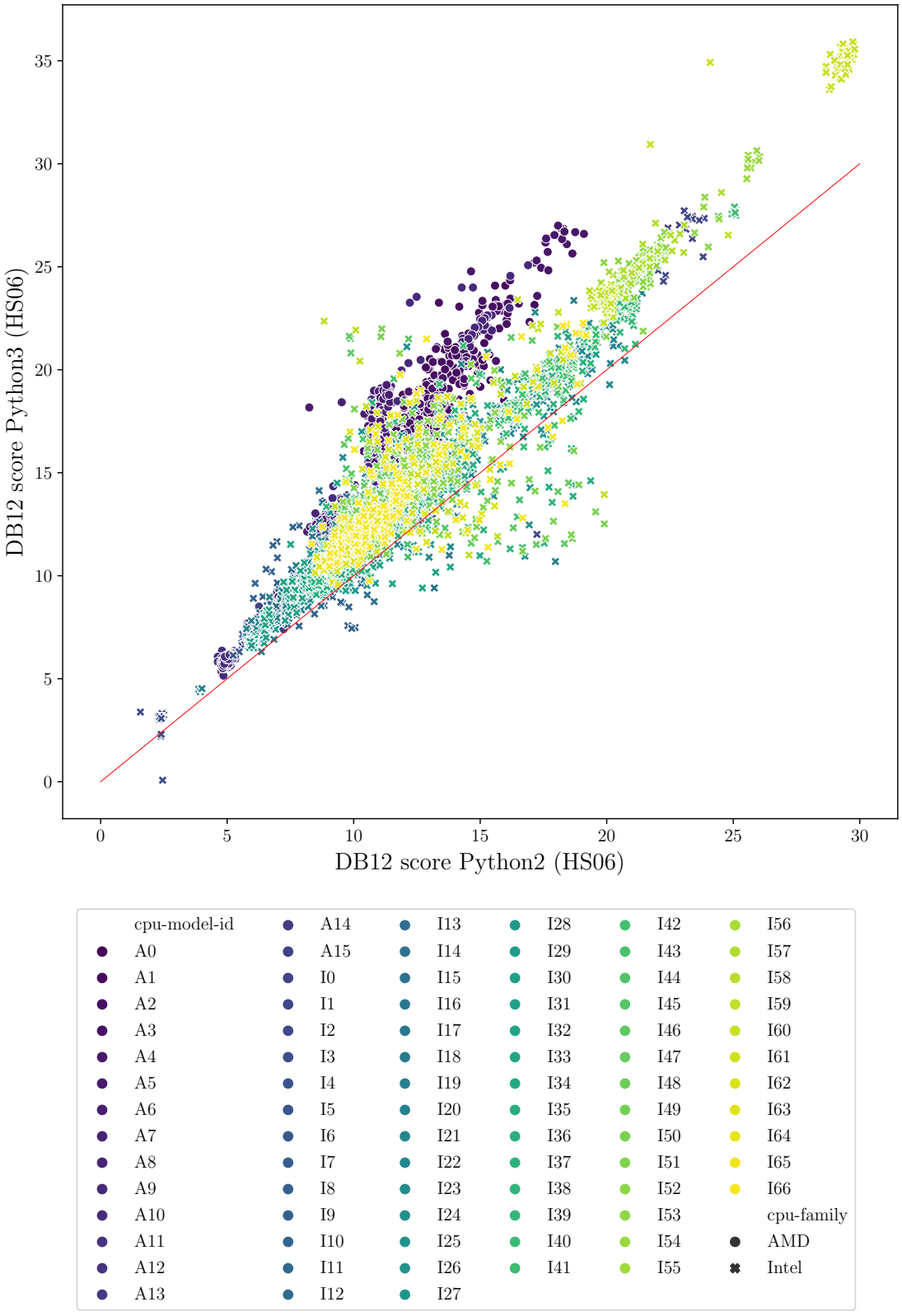

Figure 3.23 – Comparison of the benchmark scores depending on the Python version and the CPU model used

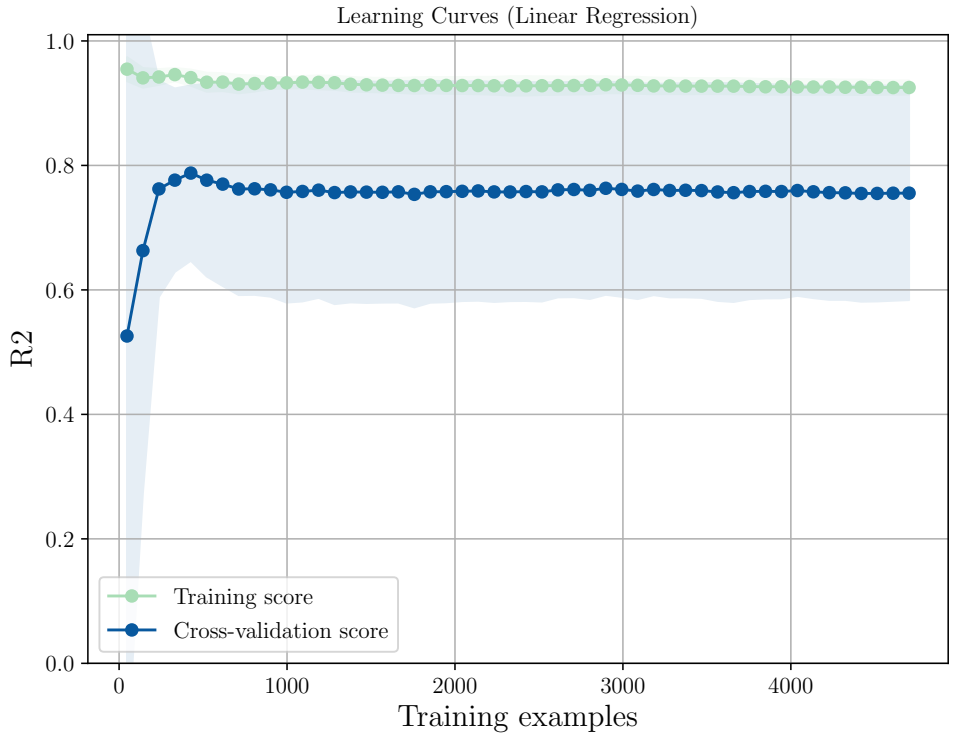

Figure 3.24 – Learning curves of the model for both training and validation. The shaded region denotes the uncertainty of that curve measured as the standard deviation. The model is scored using R2, the coefficient of determination

curves and adjust the training phase. As we have a small sample and the solution is experimental, we did not create a test dataset to validate the model. The root mean square error of the solution is about 1.25. The learning curve of the model presented in Figure 3.24 shows a high variance, which seems to indicate that additional data or a higher regularization would have provided a better model. The main advantage of this solution lies in the ease of maintenance: developers would just have to run the model with additional data regularly and integrate the model into the DIRAC Benchmark code. However, developers would need a huge amount of data.

Finally, we chose to adopt the second solution for its accuracy and because it allows developers to understand the problems. The solution takes the form of a new function in the code, which analyzes the underlying processor and Python version used, searches for a constant value to apply in a table, applies it if available, or sends a warning to the user if not. Figure 3.25 presents a comparison between the transformed

Python 3.9 scores and the Python 2.7 scores as Figure 3.23 did in Section 3.3.3.

3.3.4 Assessment of the DIRAC Benchmark scores: DIRAC Benchmark scores versus LHCb Gauss executions

As mentioned in Section 2.4.2, DIRAC Benchmark was created in 2012 based on the execution of single-thread and single-process Monte-Carlo simulation workloads on WLCG CPU resources. DIRAC Benchmark needed an update in 2016 to better fit with the evolution of the CPU resources and workloads of the LHCb experiment. Until now, there has been no further analysis following this evolution. Therefore, we perform a new analysis of DIRAC Benchmark regarding Monte-Carlo simulation jobs of the LHCb experiment.

Conditions of the experiment

We want to compare DB16 scores with Gauss jobs running on WLCG. It is worth mentioning that this experiment took place after the introduction of Python 3.9 in the LHCb production environment, Python 2.7 being not used anymore. Thus, we now define DB16 as the corrected Python 3.9 DIRAC Benchmark with a factor of 1.54. The objective is to ensure DB16 is still adapted to the current workload or to correct it if need be. We experimented by getting the list of the active Monte-Carlo productions. We extracted the attributes and parameters of 4955 Gauss jobs bound to 75 productions. Jobs ran on 98 distinct CPU models across 55 sites and provide data such as the number of events processed, the CPU power of the WN involved computed by DB16, the CPU time spent in the allocation as well as the estimated value of the CPU work spent per event - $CPU_{work}\ of\ 1event$ introduced in Section 2.4.2 -, which is related to the production.

Figure 3.26 shows the distribution of the job executions across the Sites and CPU brands. The executions are heterogeneously shared between the sites and CPUs. 70% of the executions happened on Intel CPUs against 30% on AMD CPUs. In the same way, the four most productive sites handled 54% of the jobs. This distribution corresponds to the LHCb production environment with Tier0 and Tier1 sites processing more jobs than Tier2 and Tier3 sites.

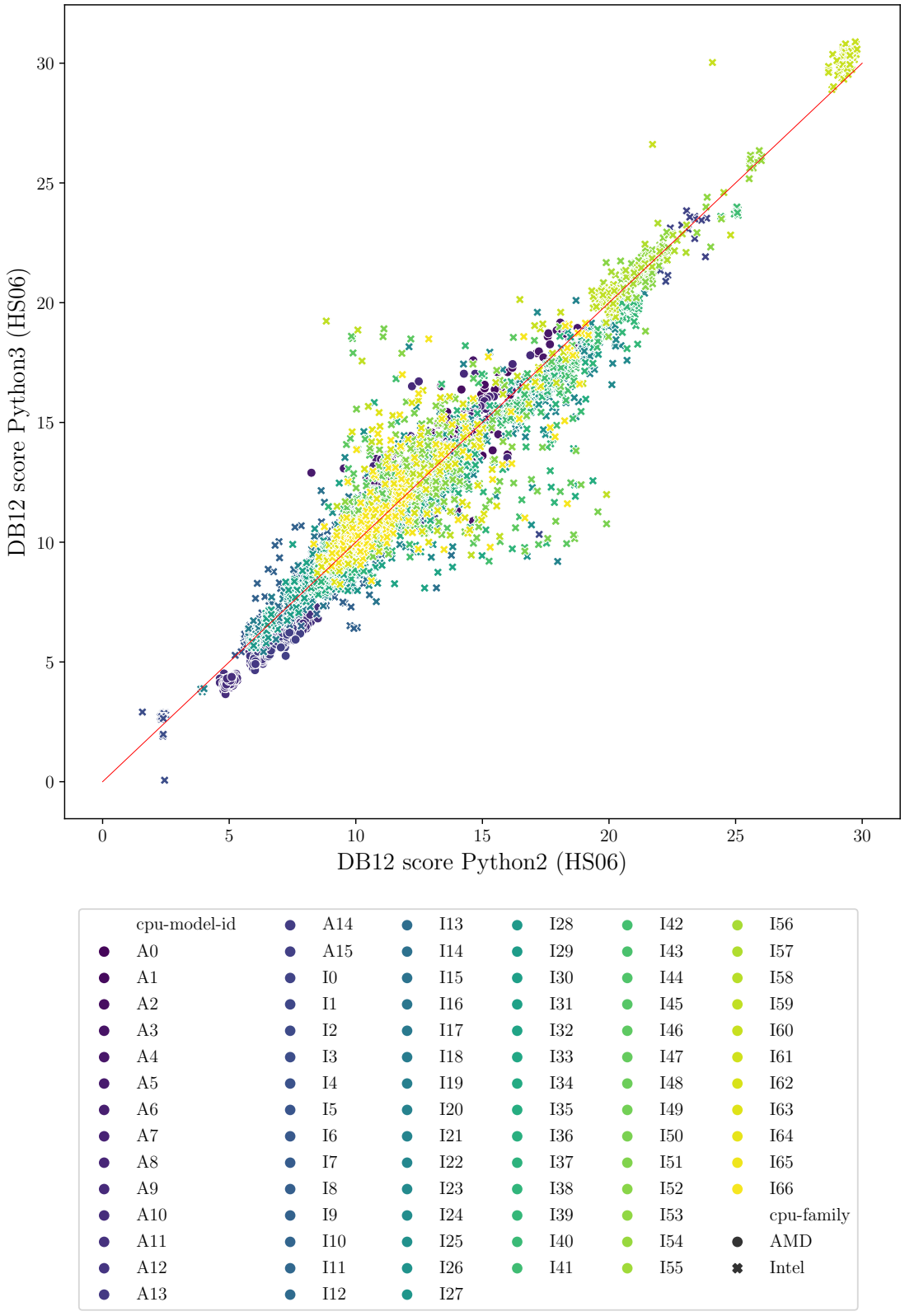

Figure 3.25 – Comparison of the benchmark scores depending on the Python version and the CPU model used after applying constant values to Python 3.9 scores

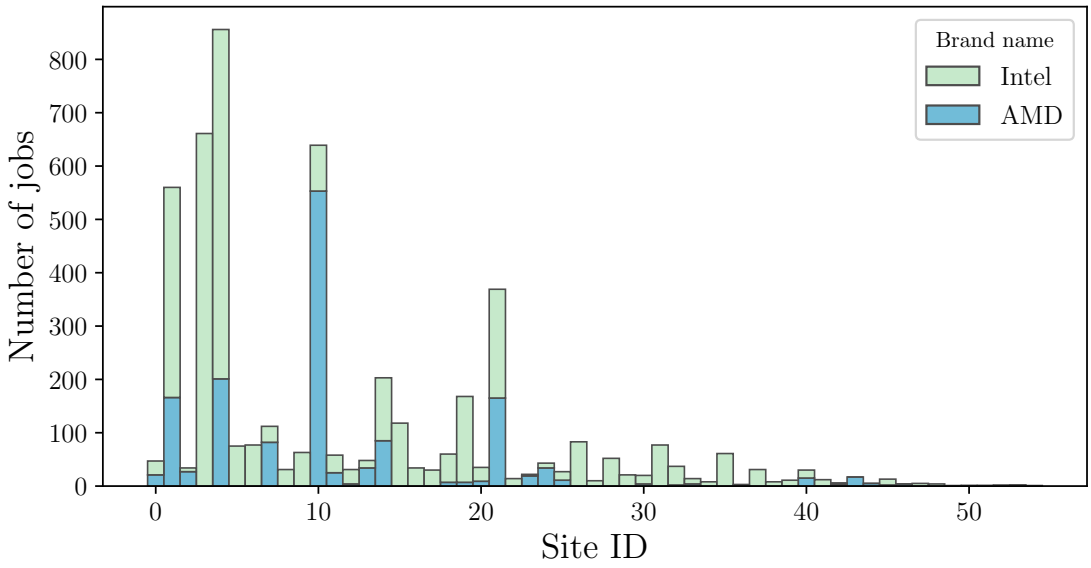

Figure 3.26 – Distribution of the job executions among the Sites and CPU models

Results

We compared the number of events produced with the CPU time spent in the allocations in Figure 3.27. Each point represents a Gauss task. We can notice that AMD CPUs generally process more events in a shorter time than Intel CPUs available. Results on AMD CPUs seem less scattered than on Intel CPUs, which could be due to the imbalance between CPU brands across the studied grid sites.

We checked whether such behavior was taken into account by DIRAC Benchmark. We compared the job power to the CPU power computed using DB16. We define the job power as the event rate, namely the inverse of the CPU time spent per event: the number of events we can process in a second. We obtained the job power by dividing the number of events processed by the CPU time spent in the allocation. From this definition, we designate $jobpower/CPUpower$ as the number of events we can process in a HS06.second. In theory, for a given production, the $jobpower/CPUpower$ value should be almost similar across various heterogeneous grid sites. In practice, it depends on the accuracy of the scores computed by DB16.

Therefore, we standardized the $jobpower/CPUpower$ values of each production - computing their Z-scores. It allows comparing the variations of the scores across many grid sites, regardless of the production parameters that could influence the job

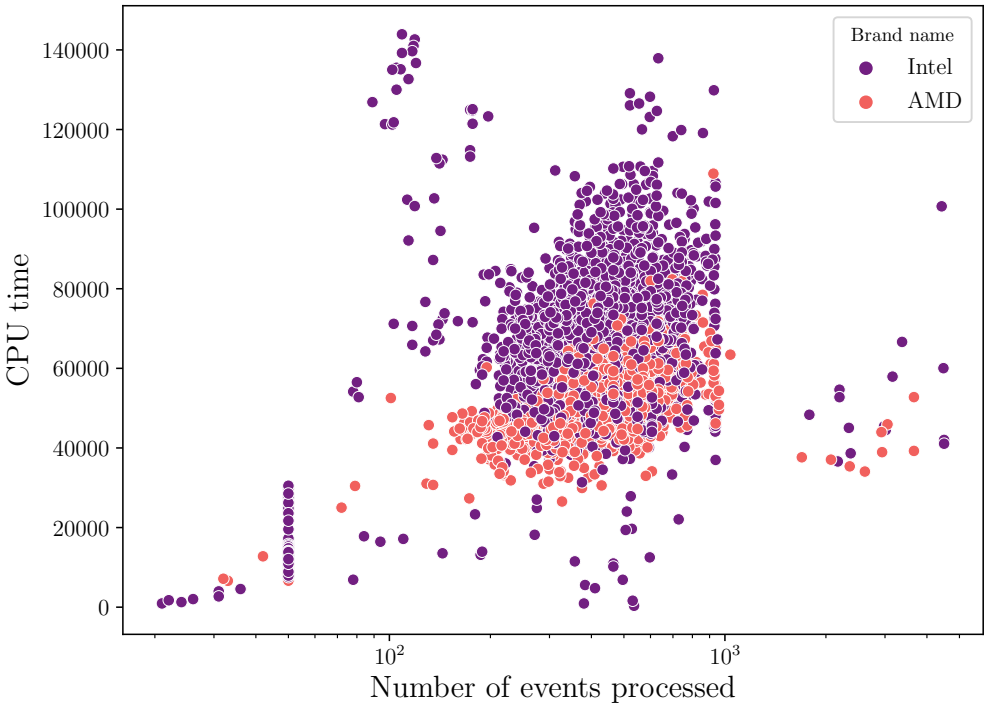

Figure 3.27 – Comparison of the number of events produced with the CPU time spent in the allocations, classified by CPU brand

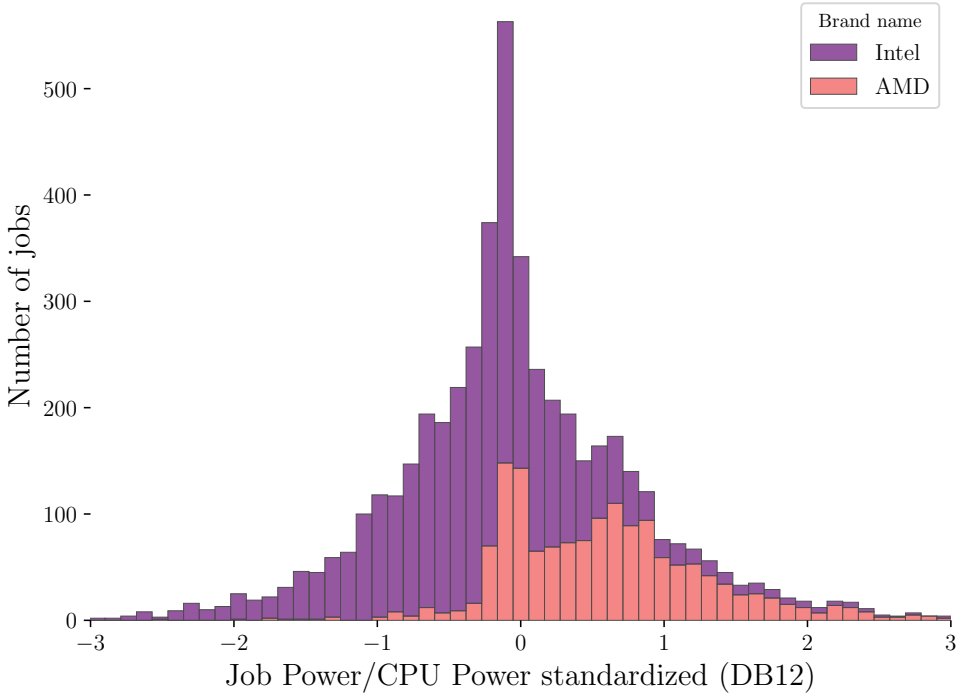

Figure 3.28 – Standardized distribution of the $jobpower/CPUpower$ values of the jobs across different grid sites, classified by CPU brands.

power. A standardized value inferior to 0 would represent a $jobpower/CPUpower$ value inferior to the mean, which would indicate an overestimation of the CPU power. Conversely, a standardized value superior to 0 would imply an underestimation of the CPU power. The more values are centered, the more DB16 is accurate. Figure 3.28 presents the standardized distribution of the $jobpower/CPUpower$ of all the productions across the sites at our disposal.

We can observe higher standardized values on AMD CPUs than on Intel CPUs, which indicates that DB16 underestimates capabilities of AMD processors to process Gauss tasks. In the same way, we can also notice that results on Intel CPUs are significantly scattered and more overestimated in general. Next, we evaluate an easy-to-implement solution to better fit with Gauss tasks.

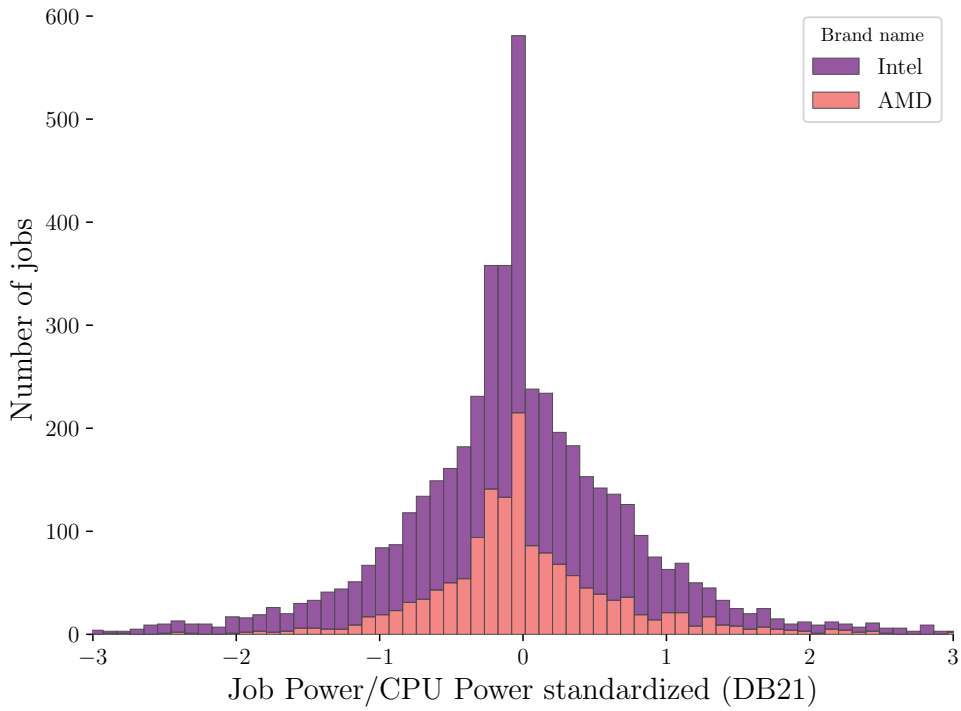

Figure 3.29 – Standardized distribution of the $jobpower/CPUpower$-not-corrected values of the jobs across different grid sites, classified by CPU brands.

DIRAC Benchmark 21

In Section 3.3.3, we noticed discrepancies between Python 2 and Python3 implementations of DIRAC Benchmark: scores computed on AMD CPUs were higher than on Intel CPUs. Thus, disabling corrections applied to the DIRAC Benchmark scores could provide results that would better fit with the current Gauss tasks. We standardized raw results of the Python 3.9 implementation the same way as above to obtain Figure 3.29.

Raw scores computed on AMD CPUs are considerably more aligned and centered around the mean, which implies a better accuracy (87% of the samples bound to AMD CPUs lie 1 standard deviation around the mean against 74% with DB16). Conversely, raw scores on Intel CPUs are slightly less grouped around the mean (78% of the samples bound to Intel CPUs lie 1 standard deviation around the mean against 81% with DB16). Even though raw scores are globally less accurate, this should not impact the LHCb production environment as LHCbDIRAC maintains a margin of 0.75% of

the CPU work available when computing the number of events to process (see Section 2.4.2). Such an approach breaks one of the key principles of the original DIRAC Benchmark 12: providing an estimated value of the HEPSPEC06 benchmark. We define the Python3.9 implementation of DIRAC Benchmark with no correction as DIRAC Benchmark 21 (DB21).

In Section 2.4.2, we have addressed the Gauss elasticity mechanism: tasks process a certain number of events depending on the CPU work left in the allocation, which depends on the CPU power and the CPU time. We completed our study by comparing the number of events processed with DIRAC Benchmark 16 with the estimated number of events that we would handle in the same allocation with DIRAC Benchmark 21. To compute the latter, we needed to use DIRAC Benchmark 21 CPU power and an approximation of the $CPU_{workDB21}\ of\ 1event$ value. $CPU_{workDB21}\ of\ 1event$ depends on the test site, which comprises 77% of Intel CPUs and 23% of AMD CPUs. Thus, $CPU_{workDB21}\ of\ 1event$ was established as:

$$CPU_{workDB21}\ of\ 1event = \frac{CPU_{workDB16}\ of\ 1event}{AMD\ factor \times 0.77 + Intel\ factor \times 0.23} \tag{3.4}$$

$AMD\ factor$ and $Intel\ factor$ refer to factors applied to port DIRAC Benchmark 12 to Python 3.9. $CPU_{workDB21}\ of\ 1event$ is expressed in HS06.seconds In this way, we obtain an estimation of the number of events processed, such as:

$$events\ to\ produce = \frac{CPU_{powerDB21} \times CPU_{time}}{CPU_{workDB21}\ of\ 1event} \times 0.75. \tag{3.5}$$

Figure 3.30 presents the results of the comparison. We can observe a clear distinction between AMD and Intel CPUs: LHCbDIRAC would process 1.15 times more events using AMD CPUs, while it would manage 0.95 times fewer events on Intel CPUs.

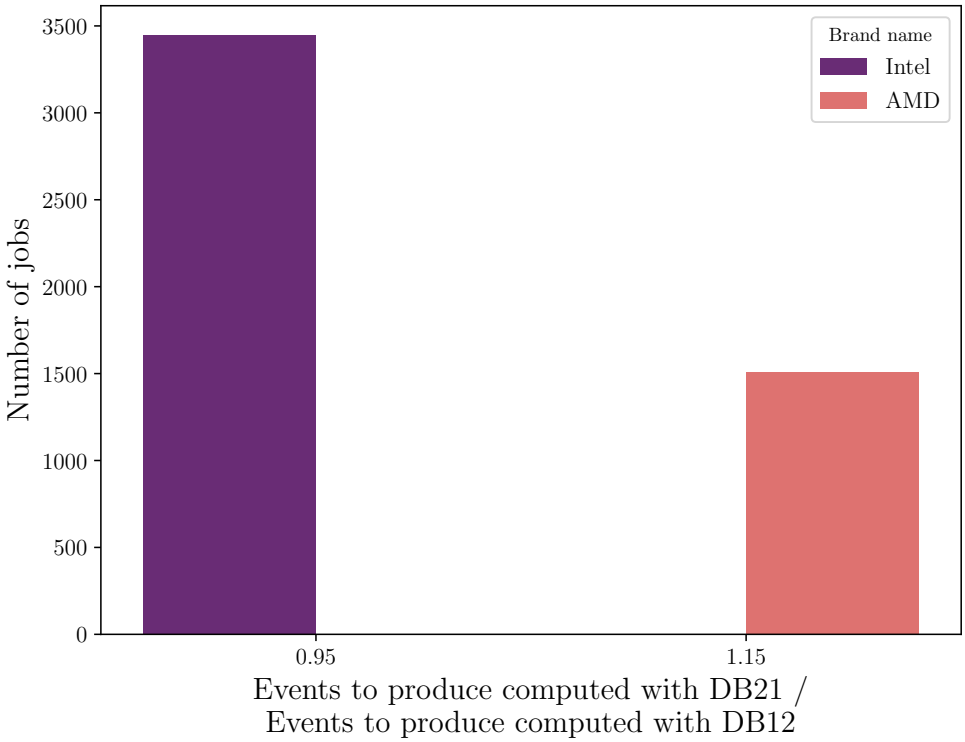

Figure 3.30 – Comparison between the number of events processed with DIRAC Benchmark 16 and the number of events that would be handled with DIRAC Benchmark 21.

3.3.5 Discussions

Does the improvement of DIRAC Benchmark allow exploiting allocations longer and, by extension, the throughput of the jobs on grid resources?

LHCbDIRAC developers drop the Python 2.7 implementation of the WMS a few months after porting DIRAC Benchmark to Python 3.9. The initial work was mandatory to preserve the current behavior in grid computing resources. According to Figure 3.25, we have been able to replicate the results of the Python 2.7 implementation and, by extension, of the HS06 benchmark, by applying corrections.

By comparing DIRAC Benchmark scores with LHCb Gauss tasks, we came to the same conclusion as Valassi et al.: the HEPSPEC06 benchmark is no longer representative of the LHC workload (Valassi, Andrea et al., 2020). DIRAC Benchmark 16 significantly underestimates AMD CPUs capabilities (Figure 3.28). DIRAC Benchmark 21 provides better results on AMD CPUs but slightly less accurate results on Intel CPUs (Figure 3.29). Given the current state of WLCG, which is mainly composed of Intel CPUs, the new version of the benchmark would not allow the WMS to longer exploit allocations, and thus, we chose to keep employing DB16 in the LHCb production environment. Nevertheless, AMD x86 CPUs are gaining ground, especially in the HPC area. The situation could change over time and DIRAC Benchmark 21 could gain in importance.

In the meantime, LHC experiments are exploring supercomputers and have started to employ accelerators such as GPUs and FPGAs to speed up parts of their workflows, and many-core allocations: DIRAC Benchmark has not been designed for such use cases. Other approaches should be considered.

Despite this study has focused on x86 CPUs in the WLCG context and remains specific to the LHCb experiment, the method could be reused in a variety of contexts implying CPU benchmarking tools bound to community-specific applications. Further efforts would be needed to include components such as co-processors and accelerators.

Future Directions and Challenges

Here, we provide further recommendations on topics not covered by this study, to help VOs employ fast CPU benchmarking tools and elastic jobs to get accurate results.

Elastic job completion largely depends on the $CPU_{work}\ of\ 1event$, which is usually computed on a test site. To get accurate results on many distributed and heterogeneous computing resources, the test site should contain a large part of the components found in the targeted grid: CPUs and accelerators. The test site should be representative of the grid state, and therefore, should be continuously checked. If none of the sites could provide such a representation, then multiple sites should be used as a reference. Even though many sites could be employed, there is currently no mechanism in place in DIRAC to make sure that the $CPU_{work}\ of\ 1event$ computed was averaged using every needed component. Such a concept could be useful in the short term.

In the long term, one could work on a new fast CPU benchmarking solution based on the HEP Benchmark suite seen in Section 2.4.2. It would potentially provide much more homogeneous results across a variety of computing resources, but might require a deep understanding of the benchmark and the applications.

One could also extend the Machine Learning benchmarking solution introduced in Section 3.3.4. It would take computing resource features as input - such as the CPU model, the load on the machine, the number of cores, the CPU time available - and would output the number of events to manage in a given allocation. This approach would likely involve many sites to train the model and would need frequent updates to remain up-to-date with the advancement of the components. This would represent a CPU-intensive task, that could otherwise be used to compute the actual workload. A study of the efforts to deploy to maintain such a model could be meaningful.

Finally, one could adopt the AES approach studied in Section 2.4.2. Designing preemptible tasks would help exploit computing resources as long as possible. It would not involve any benchmarking tool and would easily follow the evolutions. Nevertheless, it would require a lot of changes in the code and would not be adapted to grid resources: jobs running out of time would likely not be able to save the latest events processed, which would represent a waste of computing resources that any other community could have better used.

3.3.6 Summary

Through Section 3.3, we have demonstrated the importance of maintaining a fast CPU benchmarking solution up-to-date to longer exploit shared and distributed heterogeneous computing resources. After exposing the main features of DIRAC Benchmark (Section 3.3.1), we proposed a plan to continuously test the code and deploy it as a package, and more importantly, to identify discrepancies in the results after porting the code to Python 3.9 (Section 3.3.2).

We analyzed more than 4950 LHCb tasks, running on tens of sites and specifically on x86 CPUs. By defining correction factors relying on the CPU brand and the Python version used, we have been able to reproduce original scores with a Python 3.9 implementation of the tool (Section 3.3.3). We also consider a new implementation that would better fit with the LHCb Gauss tasks. We designed DIRAC Benchmark 21, which significantly corrects the underestimation of the capabilities of AMD CPUs (Section 3.3.3). Despite it would allow running 1.15 times more events on AMD CPUs, we decided not to employ it in the LHCb context as WLCG is still largely bound to Intel CPUs.

Future studies should focus on more accurate fast CPU benchmarking solutions, integrating a diversity of components such as non-x86 CPUs and accelerators, as well as many-core allocations. This will become essential in the near future and the growing adoption of systems such as supercomputers.

3.4 Conclusion

This work primarily supports research efforts conducted by the LHC experiments - and especially LHCb -, which mainly run Monte Carlo simulation workloads to replicate experimental conditions and performance of the detectors. In the context of the constant improvement of the LHC and the arrival of the High-Luminosity LHC, it is critical to make the best use of the computing power available in order to increase the quality of the analysis of the acquired data. More generally, this chapter should assist any community working with distributed and shared computing resources - even aggregated into High-Performance Computers or clouds - in processing a growing amount of data.

The approaches that we elaborate mainly target grid resources and specifically WLCG. Improving tooling (i) to provision computing resources with additional Pilot-Jobs and (ii) to maintain efficient and accurate CPU power estimations is the first part of a large plan consisting in off-loading a significant part of the LHCb workload to a few large supercomputers. Chapter 4 illustrates the second part and emphasizes several solutions to integrate the LHCb workload in constrained environments such as supercomputers.

4 LHCb workflow integration into Supercomputers

4.1 Introduction

In Chapter 3, we stated that WLCG resources were no longer sufficient to continuously process the growing amount of data coming from the LHC experiments. We emphasized the significant efforts made by LHC experiments to integrate their workloads in highly constrained environments such as supercomputers. Contrary to other LHC experiments, the LHCb collaboration has recently started and benefits from years of experience in this domain. Developers can rely on substantial literature but not mature enough to provide abstract models to help similar communities conduct similar operations in different contexts - implying other WMS, workloads and/or constrained supercomputers.

Through this chapter, we aim to offer a first draft plan to guide communities in this process. We focus on a single application: Gauss, a Monte-Carlo simulation application described in Section 1.4.1. Gauss tasks represent 71.7% of the offline activities and consume 91.1% of the CPU time available. As a CPU-intensive and limited-input application, the integration of Gauss tasks into supercomputers would represent a significant starting point.

We first describe LHCb requirements and known constraints related to supercomputers (Section 4.2). Then, we introduce the outcome of our literature review related to the integration of some LHC workload in supercomputers (Section 4.3.1). We also describe solutions that we implemented within DIRAC WMS to extend its capabilities (Section 4.3.2). Most of them are based on existing but experiment-specific approaches. Lastly, we present different use cases consisting in applying the solutions we developed to different supercomputers (Section 4.4).

4.2 Analysis of constraints

4.2.1 LHCb requirements

We need to make sure that we have the right tools before offloading the Gauss software from WLCG to supercomputers. In order to proceed, we should have a clear understanding of Gauss requirements and DIRAC capabilities (Section 4.2.1), as well as existing constraints related to supercomputers (Section 4.2.2).

Reminder: Gauss needs

In Section 1.4.1, we saw that many ongoing efforts focused on optimizing the application. Developers intend to reduce the memory footprint of the application or improve its efficiency. These approaches leverage technologies from supercomputers such as many-core nodes and accelerators. Yet, the original Gauss application based on GEANT4 still largely dominates the LHCb distributed computing activities. Gauss tasks require a huge amount of computing power to provide significant results.

Furthermore, it is worth reminding that Gauss is a CPU-intensive single-process (SP), single-threaded (ST) application, with a memory footprint of 1.4 GB. It only supports CISC x86 architectures and CERN-CentOS-compatible environments.

DIRAC capabilities

In Chapter 2, we emphasized DIRAC features and limitations. DIRAC can interact with different types of CEs and LRMS - via ssh - to supply grid resources with Pilot-Jobs (Section 2.2.1). Developers chose to entirely rely on the Pilot-Job paradigm, which is much more efficient than pushing jobs to distributed computing resources but requires ubiquitous external connectivity (Section 2.2.2). It is needed to: (i) fetch Pilot-Job code; (ii) request jobs from the DIRAC matcher service; (iii) handle input and output data before and after the execution of the jobs.

DIRAC administrators can access the configuration service to specify the number of cores available per allocation. Through the configuration, DIRAC administrators can also set up tags related to the capabilities of the resources such as memory or storage. These tags are generally matched with tags bound to jobs, to make efficient scheduling decisions. While there exist mechanisms to regulate the number of cores used within an allocation at run time, there is no solution for preventing jobs from running out of memory or checking storage availability at run time.

DIRAC supports Singularity/Apptainer to instantiate tasks within a controlled and compatible environment (Section 2.3.1). LHCbDIRAC, as well as many other VO-DIRAC implementations, relies on CVMFS to provide dependencies of the jobs (Section 2.3.2). CVMFS has to be mounted on the WNs by the site administrators.

DIRAC mainly requests single-core allocations but embeds a fat-node partitioning

mechanism to exploit multi-core allocations(Section 2.4.1). DIRAC does not support all the features provided by LRMS, which may prevent the fat-node partitioning solution from working properly: if DIRAC cannot get CPU time or wallclock time left within the allocations, then running jobs might run out of time and be killed by the system.

4.2.2 Supercomputer: many challenges to address

As described in Section 1.4.2, supercomputers offer a significant amount of computing power but remain highly heterogeneous. In practice, this poses two main integration challenges (Stagni et al., 2020): (i) a distributed computing challenge, and (ii) a software architecture challenge.

Software architecture challenge

The software architecture challenges refer to the capability of an application to efficiently exploit the computing power of a range of different processor architectures available. Nowadays supercomputers generally include multi-core and many-core x86 CPUs as well as non-x86 and RISC CPUs (ARM, Power9) that are gaining ground. Most of them integrate accelerators such as GPUs. Besides, supercomputers are mainly made for parallel processing involving inter-process communications and fast inter-node connectivity. They usually tend to favor a small number of multi-node allocations rather than many single-core allocations. Many-core nodes generally cannot support an instance of Gauss per core due to the memory footprint of the application. Overall, Gauss cannot exploit such resources by itself and needs support from DIRAC.

Distributed computing challenge

The distributed computing challenges refer to the capacity of DIRAC of providing an allocation and a proper environment including all the dependencies - data and software - to successfully execute a given task in a supercomputer. For security reasons, supercomputers may prohibit outbound connections from the WNs. DIRAC does not support such a use case and would need to be adapted accordingly. Moreover, despite all recent supercomputers leveraging Linux to administer the activities of the system, there exist various distributions of this OS. To get a compatible OS,

DIRAC administrators can set up the Apptainer interface to instantiate jobs within containers, as long as the technology is supported. In the same way, CVMFS is not installed on such infrastructures. DIRAC administrators can intend to collaborate with site administrators to cope with such an obstacle, but it is not guaranteed. To finish, supercomputer administrators may require users to connect to a VPN to access computing resources. Theoretically, DIRAC administrators should be able to install a Site Director on the edge node to communicate with DIRAC services and submit jobs to the local LRMS. In practice, the use case has not been tested and depends on security policies: running a program from an edge node might be prohibited or external connectivity might be disabled.

While WLCG Grid Sites provide a relatively uniform computing environment, supercomputers may differ significantly from one another. The singularity of supercomputer resources demands specific solutions for each of them, generally expensive in terms of manpower and operations. Thus, the strategy of LHCb consists in (i) exploiting x86 CPU supercomputer partitions to limit software changes; (ii) collaborating with the local system administrators and performance experts, which has proved to be mutually beneficial and has helped to address many specific issues. Adapting LHCb workloads and DIRAC to supercomputers is essential regarding the substantial amount of computing power they offer and the growing computational needs to process future LHC data. In Section 4.3, we are going to propose a general plan and implementations to fine-tune the DIRAC WMS according to known constraints.

4.3 Design of software blocks to integrate workloads

4.3.1 General plan

In Chapter 2, we have analyzed various use cases involving teams intending to integrate their HTC workloads on supercomputers. Most solutions required an immediate development not grounded on any analytical understanding of underpinning abstractions. In Section 4.3, we aim to guide communities that would start this process. Our guide takes the form of an activity diagram appearing in Figure 4.1. It remains relatively bound to LHC experiments and does not treat data movement problems but could serve as a basis to further projects, especially ones interacting with grid resources.

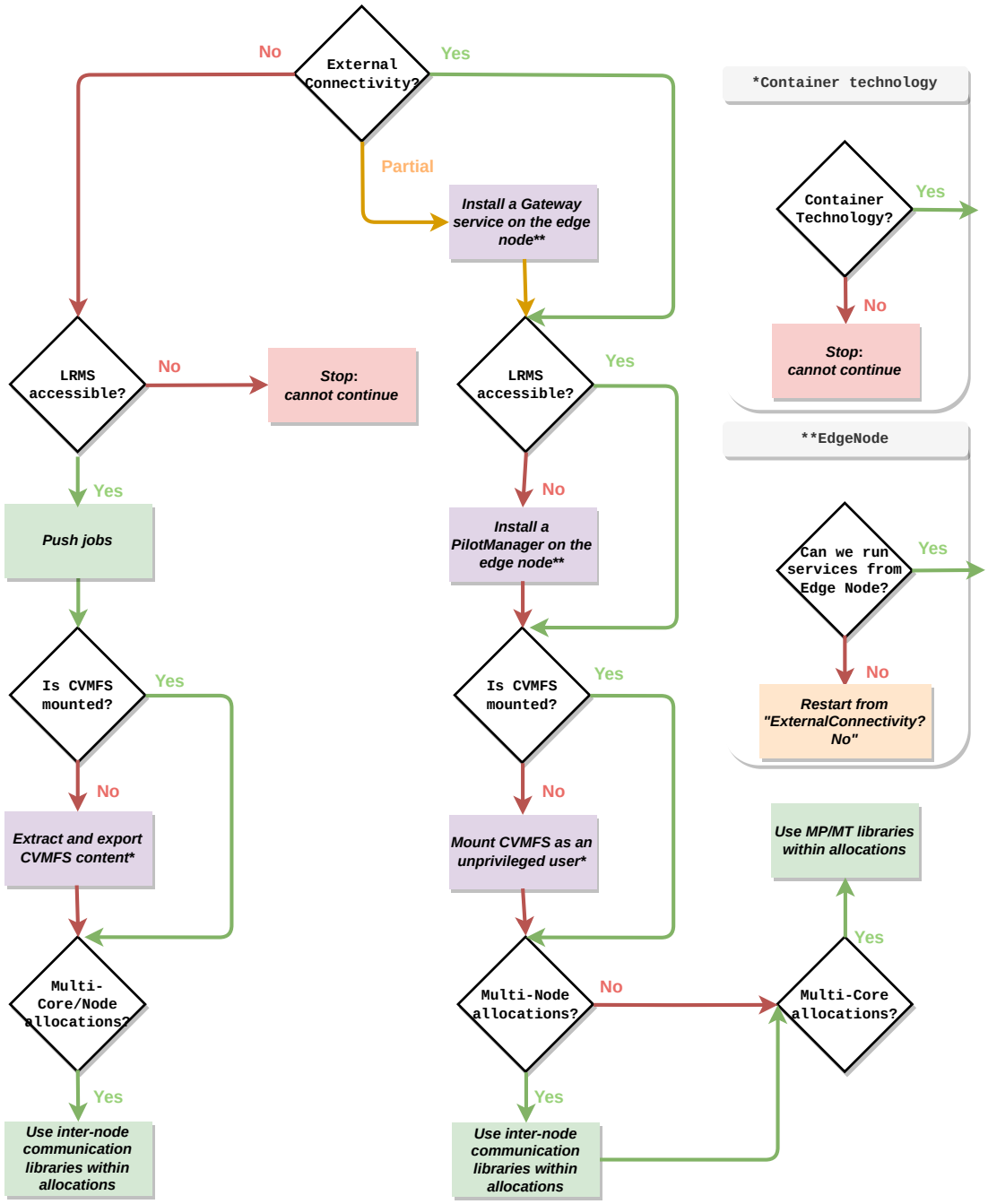

Figure 4.1 – Activity diagram to better understand capabilities of a system and existing solutions to cope with integration obstacles.

Our guide is designed to interrogate communities about the constraints of a given supercomputer. Following paths allow communities to test the capabilities of the system and set up the required components to cope with the obstacles that could hamper the integration of their workloads.

The first question implies a choice of provisioning model. Getting external connectivity from the WNs allows contacting external services, and thus, employing the Pilot-Job paradigm. Getting partial connectivity - WNs having only access to the edge nodes, which can contact services outside - is similar but requires a gateway service. The role of a gateway service is to redirect network traffic from Pilot-Jobs and tasks running on WNs to external services. A gateway service is composed of a client installed within the allocation on a WN, and a service hosted on an edge node. Tovar et al. built a VPN client and server acting as a gateway (Section 2.3.2). We could study the utility and, potentially, use it in the context of the LHCb experiment, even though we have no such a use case at the moment. The service works provided system administrators allow communities to install their components on an edge node, which is not always the case. Otherwise, jobs need to be pushed to the supercomputer. WMS have to embed or communicate with a Job Manager, installed outside of the machine, functioning similarly to a Pilot-Job provisioning tool. A Job Manager pre- and post-processes jobs - including interactions with external services -, and solely submits the tasks to a remote middleware. It also monitors jobs and controls the throughput. The most known example is the ARC Control Tower (aCT), which intends to offer a generic interface to WMS. Because DIRAC jobs embed a linear workflow of connected operations, developers would have to develop their utility based on the foundations of aCT. The first response of the activity diagram leads to significant changes in the approach to adopt and appears as two distinct paths in the diagram. Currently, DIRAC does not support any solution to push jobs directly to a LRMS.

The second question is related to LRMS accessibility. In some cases, LRMS are only reachable locally, because of an unstable ssh connection for instance, or hidden behind a VPN. WMS should implement a module to automatically connect to VPN and submit jobs via ssh. If there is no outbound connectivity within the system and the connection is unstable, then it is impossible to deal with the supercomputer. In the other case, communities can install a Pilot Manager directly on an edge node to locally submit to the LRMS without passing through an ssh connection and/or a VPN. As for the gateway service, the solution requires system administrators to allow

communities to run their services on edge nodes. The DIRAC Site Director should be able to run on an edge node if allowed, but the option would need to be studied.

The third question is specific to communities managing their dependencies through CVMFS. In Section 2.3.2 we discussed various approaches to use CVMFS from unprivileged and/or non-connected environments. So far, DIRAC administrators have been able to collaborate with system administrators proposing outbound connectivity. They have been allowed to mount CVMFS on the WNs, but this is not always the case. To execute certain applications, container technologies are mandatory but not always present on supercomputers. The exploitation is declared impossible if this technology is not present.

The last questions concern the size of the allocations. Teams that work on the subject leverage inter-node communication libraries such as MPI and OpenMP to develop wrappers around Pilots and/or jobs in order to exploit many-core allocations. In the case WNs have no access to the external network, wrappers embed a fixed number of jobs, which means they should have a similar duration to avoid waste of resources. Otherwise, wrappers can embed Pilot-Jobs and better control the allocated space. Overall, solutions present on the left side of Figure 4.1 are more generic but less efficient than approaches on the right side. In Sections 4.3.2, 4.3.3 and 4.3.4, we will study how we integrated or developed the solutions mentioned above to exploit a larger number of supercomputers and better leverage allocations.

4.3.2 Reintroducing the Push model

EuroHPC and PRACE offer access to different European supercomputers. Many of them embed a significant but highly restricted computing power: nodes have no outbound connectivity. We need to define a strategy and implement solutions within DIRAC to leverage such supercomputers. In Section 4.3.2, we first analyze the ARC Control Tower, one of the most used tools to supply nodes having no external connectivity with jobs. Then, we emphasize the limitations of the current DIRAC Pilot-Job workflow and, finally, introduce a solution to push jobs, based on the ARC Control Tower principles.

Analysis of ARC Control Tower

Nilsen et al. provides many details about aCT implementation (Nilsen et al., 2015). aCT is a middleware aiming to connect an external job provider, such as a WMS, with many ARC CE. The purpose is to ease the management of jobs pushed to an ARC CE (Figure 4.2). The *App engine* pulls job descriptions from the external job provider, converts them into XRSL and sends them to the *ARC table*. The *ARC table* stores data related to jobs and their states and is used to feed the *ARC engine* with new jobs. Then, the *ARC engine* submits the XRSL descriptions to ARC CEs and monitors the state of the jobs until finished. Once done, the *ARC engine* fetches the outputs of the jobs and sends them to the *ARC table*. They are verified by the *App engine* before the *ARC engine* cleans them from the system. Finally, the *App engine* sends the result back to the job provider.

It is worth noting that during the run time, the *App engine* communicates with the job provider to update the job heartbeats and check whether a job should be canceled. The *ARC engine* automatically resubmits jobs when they fail due to a known error coming from the resource side. Also, the *ARC engine* can interact with other CE types such as HTCondor.

aCT allows communities to design their *App engine* and use their database as *App table* to define how specific job descriptions should be translated to XRSL. While the solution is highly-configurable and extensible, it would not bring so many benefits to DIRAC, which already embeds communication interfaces to interact with CEs. We would still need to develop components to push jobs to aCT. We decided to rely on DIRAC components to develop a first push service prototype.

Analysis of the LHCbDIRAC Pilot-Job workflow

DIRAC developers develop *Pilot* (“Pilot”, 2022), a repository containing the programs used during the Pilot-Job execution once installed on a WN. Developers implemented *Pilot* using a command pattern. Each command performs an atomic operation and the repository already embeds various commands specific to DIRAC. Communities can extend *Pilot* and develop their commands. By default, the DIRAC pilot installs and configures the full DIRAC environment on the WN and analyzes the environment. It computes the CPU power using DIRAC Benchmark before invoking a special agent

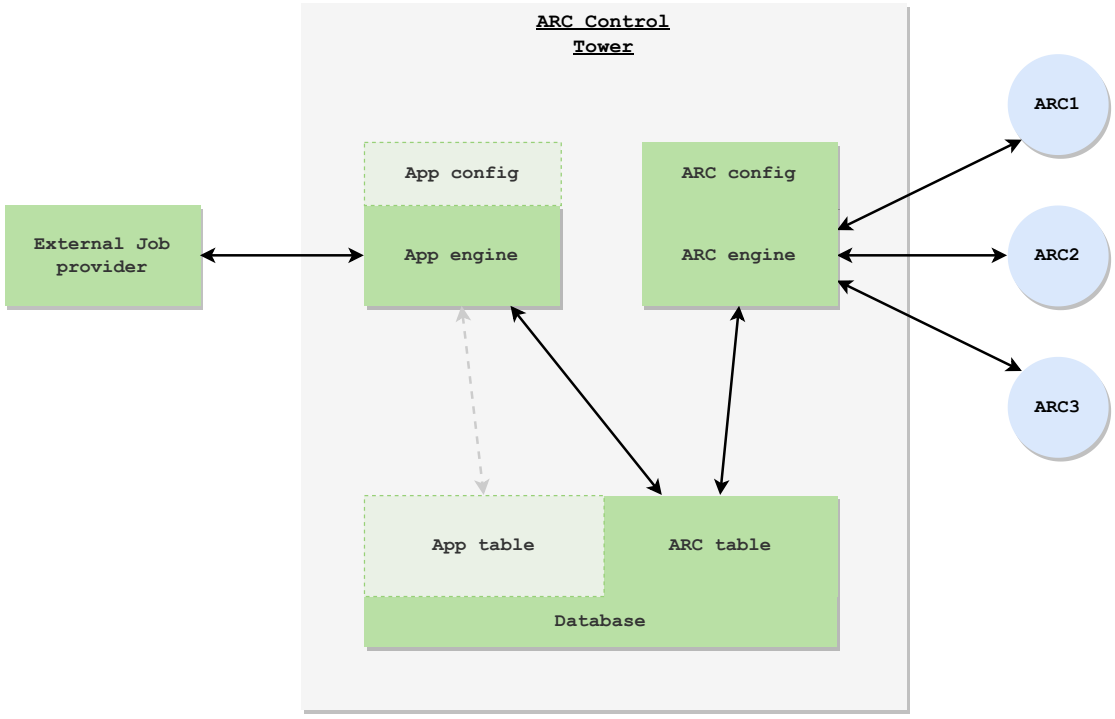

Figure 4.2 – ARC Control Tower workflow.

named *JobAgent*.

JobAgent runs multiple cycles. Each cycle, it: (i) checks whether there are still cores available; (ii) computes the CPU work left; (iii) requests one job to the DIRAC Matcher service (Figure 4.3 step 3); (iv) generates a job wrapper (Figure 4.3 step 4) and submits it to an inner CE (Figure 4.3 step 5). Inner CEs correspond to interfaces to manage jobs within an allocation. DIRAC embeds 3 base inner CEs:

- `InProcessCE`: executes the job in the same process as the caller.
- `SudoCE`: executes the job in a spawned process with a different user namespace. It allows isolating the caller environment from the user job and is used on VMs.
- `SingularityCE`: executes the job within a Singularity/Apptainer container.

In multi-core allocations, DIRAC calls the `PoolCE`, which manages a pool of separate processes. When a `PoolCE` receives a job, it checks whether a core is available. If it is the case, it instantiates a process and transfers the job to a base inner CE. The `PoolCE` is a major component of the fat-node partitioning mechanism seen in Section 2.4.1.

The job wrapper is a composition of 3 files: a *JobWrapper* executable, a *JobWrapper* template and a configuration file. The *JobWrapper* template is a Python module using the configuration file to get job arguments. It initializes a *JobWrapper* object, downloads the input sandbox and resolves‘ the input data related to the job, executes the *JobWrapper* object and uploads some of the outputs to an output sandbox (Figure 4.3 step 7, 8, 9 and 12). If an error occurs, it reschedules the job. The *JobWrapper* executable is a bash script invoking the *JobWrapper* template with additional parameters such as the logging level.

The *JobWrapper* object launches two components simultaneously. The first one is the job executor, `dirac-jobexec` by default (Figure 4.3 step 10). It is a Python script decoding an XML file containing the workflow of operations and executing the modules sequentially. The second one is a `Watchdog` object monitoring the system resource consumption and returning information via a heart-beat mechanism (Figure 4.3 step 11).

We define a workflow of sequential operations as a DIRAC workflow. A DIRAC workflow contains steps, which are independent of each other. The output of a given step serves as the input to the next one. Steps are composed of operations called modules in this context. The default DIRAC workflow comprises one step of one module, which simply executes the task. LHCbDIRAC substantially extends the concept of DIRAC workflow. The WMS proposes different DIRAC workflows depending on the type of jobs. Gauss production tasks are wrapped into a DIRAC workflow comprising two steps: *Gaudi_App_Step* and *Job_Finalization*. *Gaudi_App_Step* is related to the execution of Gauss (1 module) and the verification and formatting of the results and errors (4 modules). *Job_Finalization* intends to upload outputs to external services (4 modules).

It is worth noting that communities may choose not to use the DIRAC workflow. In this case, the *JobWrapper* object can launch a simple task, generally taking the form of a bash script. In place of `dirac-jobexec`, one could also provide another type of job executor, to interpret different workflows for instance.

Figure 4.3 describes the Pilot-Job implementation of DIRAC and emphasizes the components and external communications involved in the job execution. It starts with the Site Director submitting the Pilot to finish with the DIRAC workflow execution.

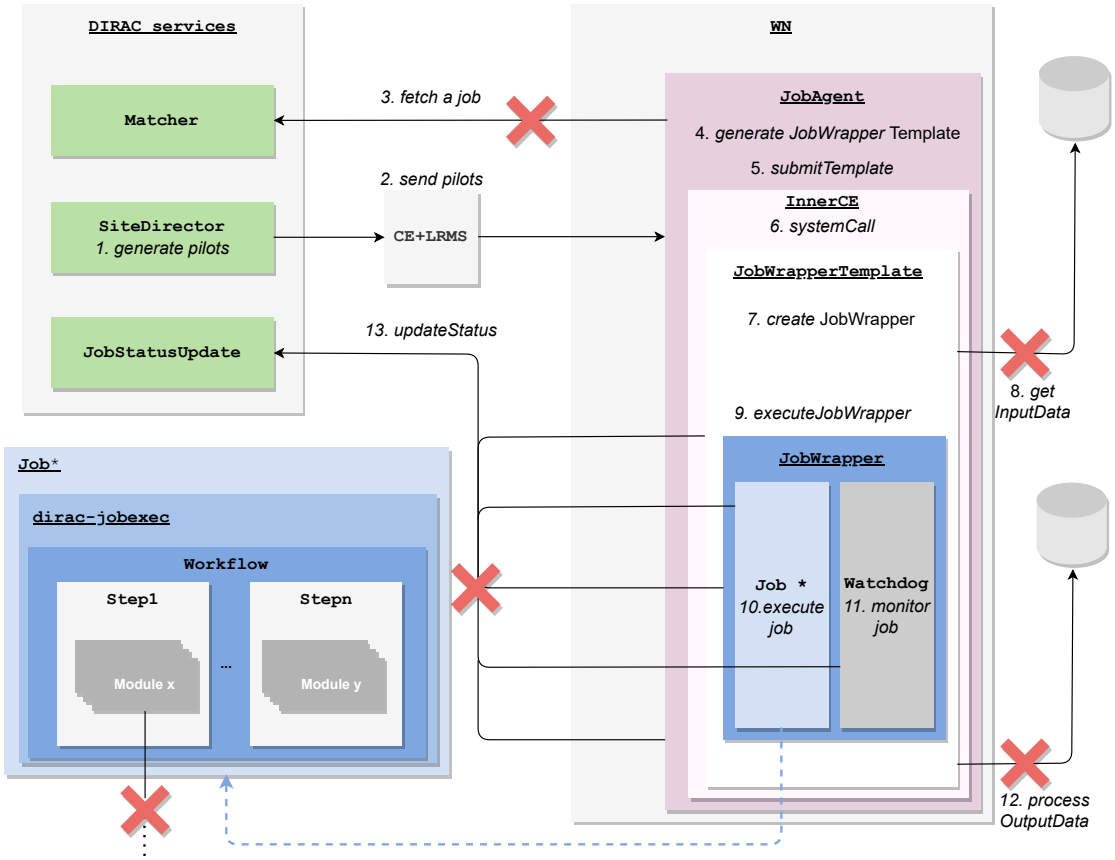

Figure 4.3 – Schema of the Pilot-Job implementation of DIRAC, focused on the job execution. Red crosses indicates external communications impossible within supercomputers with no outbound connectivity.

The execution of a job implies many connections to external services: first, we need to request the job, then to get its input data, to update its status and to upload results and output data. In constrained environments with no external connectivity, a Pilot could not even fetch a job. Because this workflow is intensively used in production, and that constrained environments currently represent an insignificant fraction of the available computing resources, we aim to design a simple and seamless solution by minimizing the changes in the architecture.

Introduction of the Push Job Agent

To cope with external connectivity issues while preserving the current architecture, we introduced two components: *PushJobAgent* and *RemoteRunner*. The main idea consists in solely submitting the tasks - Gauss in this context - and executing pre- and post-processing steps requiring external connectivity outside the targeted system. *PushJobAgent* acts similarly to the *App engine* of the ARC Control Tower, namely fetching jobs, whereas *RemoteRunner* behaves as the *ARC engine* of the ARC Control Tower. The latter submits the tasks to a CE and monitors them using the communication interfaces of DIRAC. Figure 4.4 presents the integration of these new components within the Pilot-Job implementation of DIRAC.

PushJobAgent is a DIRAC agent. It inherits from *JobAgent* and proposes a similar structure: it checks whether there are still slots available, requests jobs to the DIRAC Matcher service, generates job wrappers and submits them to a `PoolCE`. While not inheriting from *SiteDirector*, it borrows a few features from it. It can supervise multiple sites, CEs and queues and puts queues on hold for a certain number of cycles when an error occurs.

Contrary to *JobAgent*, instances of *PushJobAgent* are installed outside of the targeted system and are permanently running. Because an instance can manage multiple sites, it sequentially fetches jobs and submits them until the targeted LRMS queues of the sites are fully loaded, in a single cycle. To get the most adapted jobs, the *PushJobAgent* instance provides the DIRAC Matcher service with attributes of the targeted sites, CEs and queues, instead of the attributes of its host. Then, to guide the jobs to a *RemoteRunner* instance, the agent adds a special tag to the attributes of the targeted computing resources to inform the next components that the embedded task should be pushed. It is worth mentioning that, as some modules of the DIRAC work-

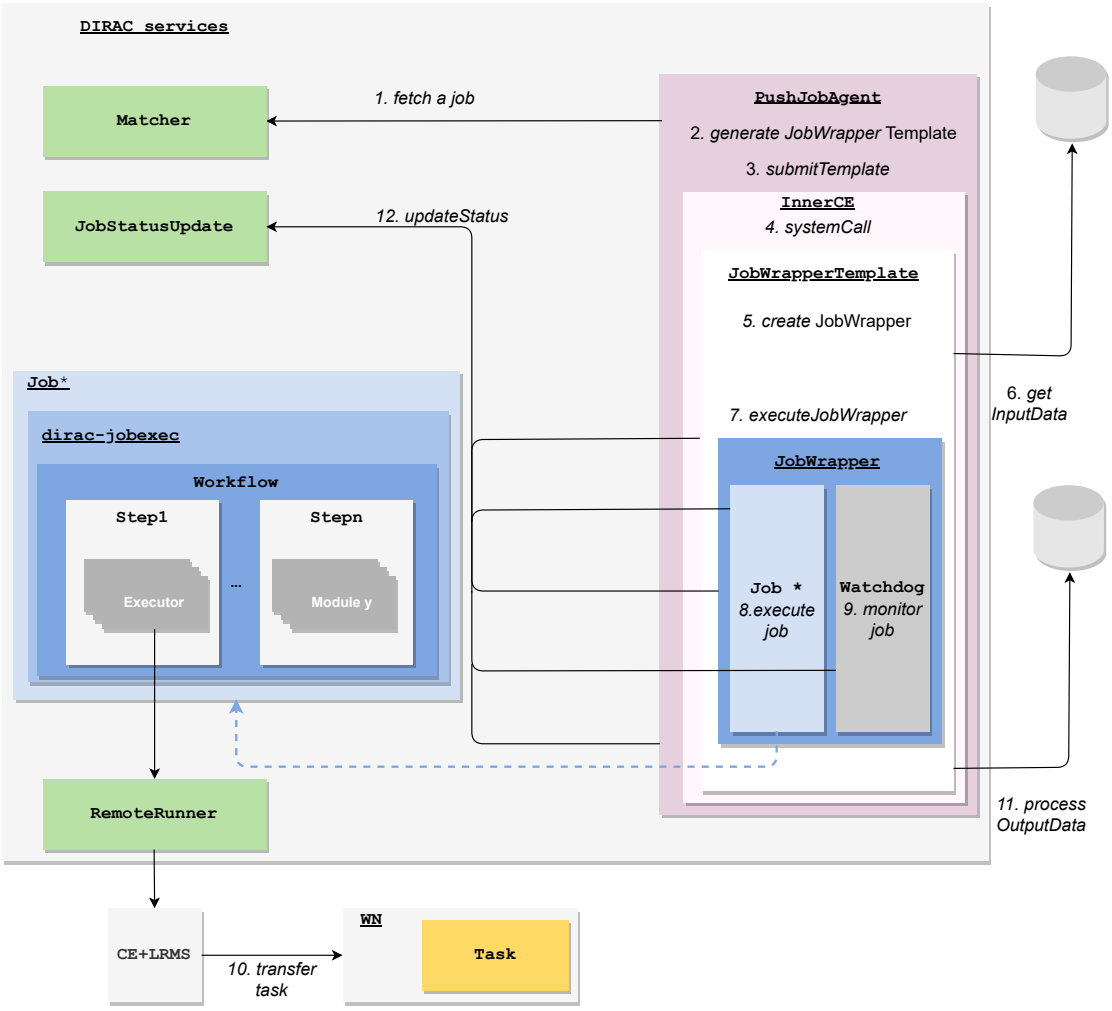

Figure 4.4 – Schema of the *PushJobAgent* implementation of DIRAC, focused on the job execution.

flows are executed in the host of the *PushJobAgent* instance, DIRAC administrators can configure the maximum number of jobs the instance can support in parallel to avoid reaching the limits of the host. The *dirac-jobexec* executor is thus run outside the targeted system.

RemoteRunner is a Python module initializing a communication interface to interact with the targeted CE. It is instantiated and employed by a DIRAC workflow, providing that the targeted computing resources have been tagged by a *PushJobAgent* instance. *RemoteRunner* submits the embedded task with its input files, monitors its status and gets its outputs once it is finished. It formats the result to return it to the caller module as if it was executed locally.

With this feature, DIRAC can partially support supercomputers with no external connectivity. It can push tasks to the system but cannot provide its dependencies. The current implementation solely allows exploiting single-core allocations. We have imagined a `BundleCE` that would be a service between *RemoteRunner* and a communication interface. It would receive many tasks and wrap them in an MPI script as a bundle that would be submitted to the targeted CE. In Section 4.3.3, we focus on the implementation of a generic pipeline to deploy a subset of CVMFS in a constrained environment.

4.3.3 SubCVMFS: providing job dependencies in no-external connectivity environments

As stated in Section 2.3.2, we developed an open-source pipeline to identify dependencies of a given application, extract them, test them and deploy them on constrained computing resources (Boyer et al., 2022c).

Input and output data

The utility takes a directory as input that should contain: (i) a list of applications of interest (`apps`): a command along with its input data in a separate sub-directory for each application to trace; and/or (ii) a list of files composed of paths to include in the subset of CVMFS (`namelists`). Additionally, user communities can embed a (iii) container image compatible with Singularity/Apptainer to get a specific environment to trace and test the applications; (iv) and a configuration file to fine-tune the utility with variables related to the deployment process, or information about repositories. A schema of the inputs is available in Figure 4.5.

The expected output can take different forms depending on the utility configuration:

- The subset of CVMFS, generated as a standalone. In this case, administrators representing their user communities need to provide the right environment by themselves, which might also involve discussions with the system administrators.
- The subset of CVMFS embedded within the given Singularity/Apptainer con-

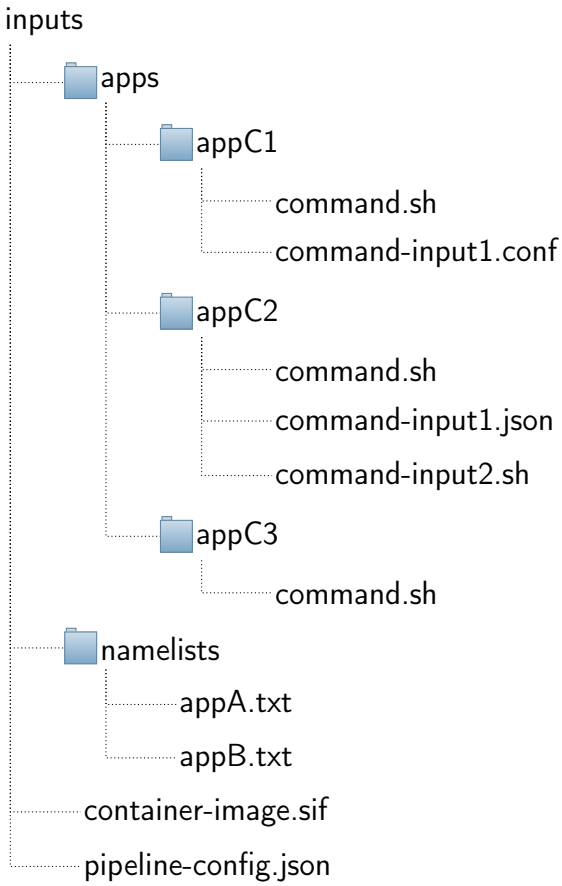

Figure 4.5 – Schema of the input structure given to the utility.

tainer image. The utility merges both elements and submits the resulting image, which can be long to generate and deploy but may limit manual operations on the remote location.

Features

We break down the process into four main steps, namely:

- *Trace*: consists in running applications contained in `apps` and trapping their system calls at runtime, using *Parrot*, to identify and extract the paths of their dependencies. Applications can run in a Singularity/Apptainer container when provided, which delivers further software dependencies and a reproducible environment. Dependencies are then saved in a specific file `namelist.txt`. In this context, *Parrot* is only used to capture system calls and, thus, is not impacted by the issues mentioned in Section 2.3.2. If the step detects an error during the execution of an application, then the program is stopped. The step is particularly helpful for users of the utility having no technical knowledge of the applications of interest.

- *Build*: builds a subset of CVMFS based on the paths coming from *Trace* and the `namelists` directory. First, the step merges the namelist files to remove duplicated or non-existent path references, and then separates the paths in different specification files related to repositories. Finally, the step calls *cvmfs_shrinkwrap* to generate the subset of CVMFS. Figures 4.6 and 4.7.3 illustrate an example. The utility deduplicates the files, and hard-link data to populate a directory, ready to be exported in various formats as shown in Figure 4.7.3.

- *Test*: consists in testing certain applications - in the given Singularity/Apptainer container environment when provided - using the subset of CVMFS obtained during the *Build* step (see Figure 4.7.4). By default, applications from `apps` are used but further tests can also be provided by modifying the utility configuration. All the applications have to complete their execution to go to the next step.

- *Deploy*: deploys the subset of CVMFS (Figure 4.7.5) embedded or not within the container image depending on the configuration options. If such is the case, then the utility (i) generates a new container definition file that includes the files with the container image, (ii) executes it to produce a new read-only

```
in namelist1.txt:
/cvmfs/repoA/path/to/file
/cvmfs/repoB/path/to/another/file
in namelist2.txt:
/cvmfs/repoA/path/to/file
/cvmfs/repoB/path/to/yet/another/file

in repoA.spec:
/path/to/file
in repoB.spec:
/path/to/another/file
/path/to/yet/another/file
```

Figure 4.6 – Transformation process occurring during the *Trace* step: CVMFS dependencies are extracted from `namelist.txt` and moved to specification files.

container image. The utility supports ssh deployment via *rsync*, provided the right credentials in the configuration.

Implementation

The utility is built as a 2-layer system. The first layer, *subcvmfs-builder* (Boyer, 2022b), is the core of the system and is self-contained. It takes the form of a Python package, which embeds the steps described in Section 4.3.3, and provides a command-line interface to call and execute steps independently from each other. The first layer is, and should remain, simple and generic to be easily managed by developers and used by various communities.

The second layer is the glue code: it consists of a workflow executing - all, or some of - the steps of the first layer. It contains the complexity required to generate and deliver a subset of dependencies according to the needs of its users. Unlike the first layer, the second one can take several forms and each community can tailor it for its software stack.

We propose a first, simple and generic layer-2 implementation calling each step one after the other: *subcvmfs-builder-pipeline* (Boyer, 2022c). This layer-2 implementation is executed from a GitLab CI/CD (“Gitlab CI/CD”, 2022), which provides a runner and a docker executor bound to a CVMFS client to execute the code (see Figure 4.8) GitLab includes features such as log preservation to help debug the implemen-

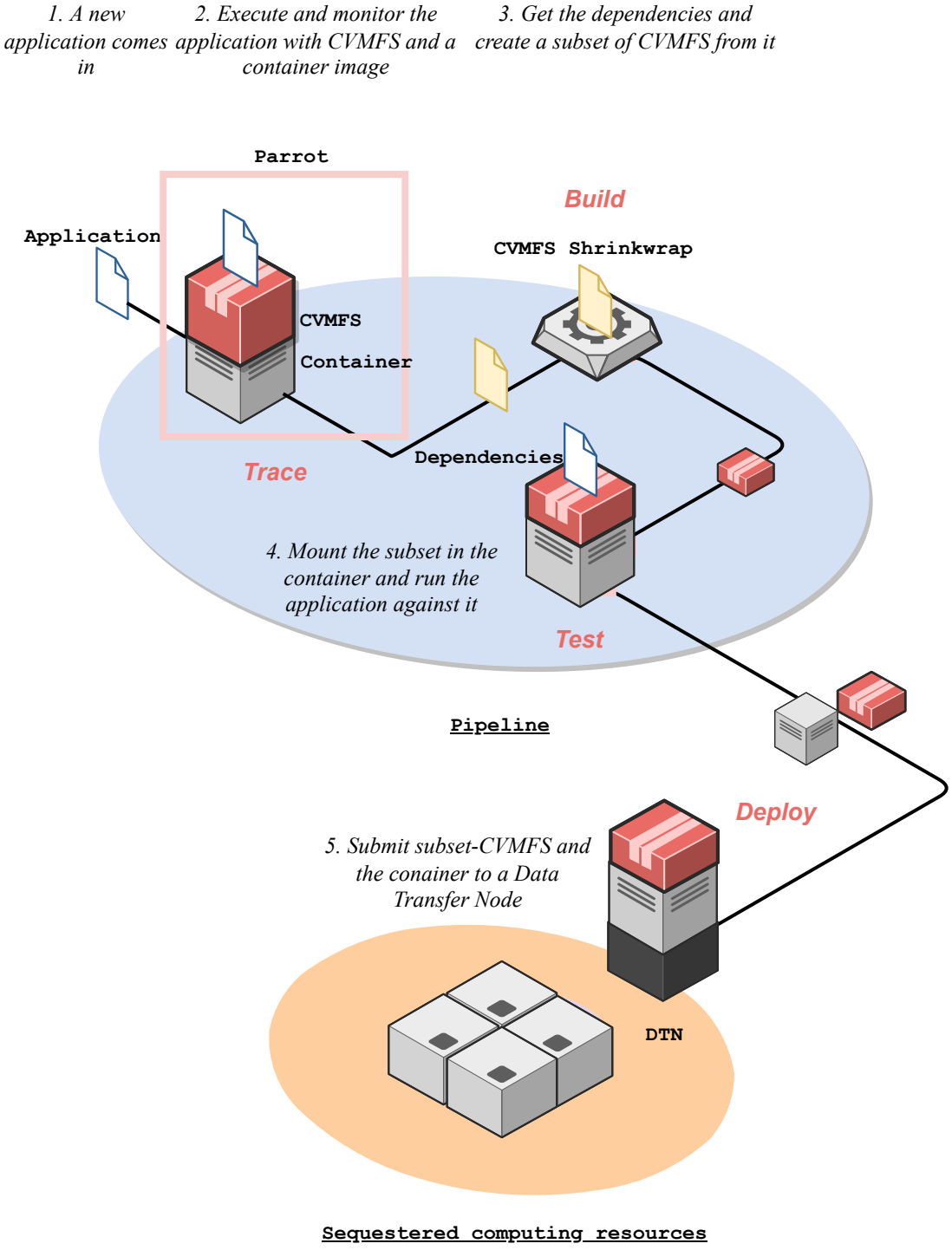

Figure 4.7 – Schema of the utility workflow: from getting an application to trace to a subset of CVMFS on the Data Transfer Node (DTN) of a High-Performance Computing cluster.

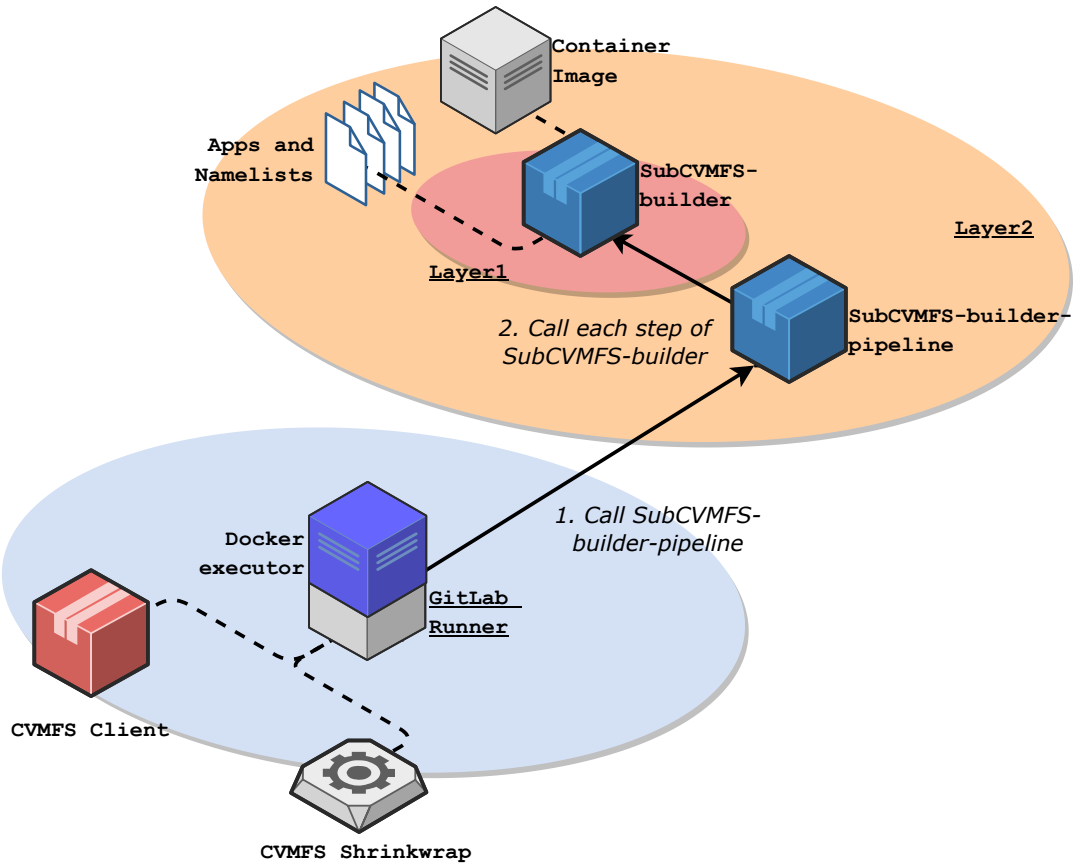

Figure 4.8 – Schema of a layer-2 implementation within GitLab CI.

tation and integrates a pipeline scheduling mechanism to regularly update a subset of dependencies. Even though this layer-2 solution is adapted for basic examples - implying a few commands to trace and test, having a small number of dependencies -, it might require further fine-tuning for more advanced use cases.

Indeed, this generic layer-2 implementation is not scalable as it (i) is a single-threaded and single-process program, and (ii) requires manual operations to insert additional inputs in the process. This is not adapted to communities having to trace and test hundreds of various applications to generate large subsets of CVMFS. Two possibilities for such communities: building a new layer-2 implementation - able to automatically fetch applications and trace/test them in parallel - based on *subcvmfs-builder-pipeline* or creating one from scratch.

Next, we are going to study how the LHCb experiment leverages *subcvmfs-builder* to deliver Gauss on WNs of a supercomputer with no external connectivity.

LHCb implementation

Running embarrassingly parallel applications such as Gauss on a supercomputer can be seen as counterproductive. While it is true that the interconnect of the supercomputer partitions has not been designed for millions of small Monte-Carlo runs, it is better to use available, otherwise unused, cycles in agreement with the management of the supercomputer sites. In the meantime, developers are adapting software, but it remains a long process, requiring deep and technical software inputs.

To deliver Gauss on Mare Nostrum, LHCb can rely on (i) *subcvmfs-builder* to produce a subset of CVMFS containing the required files; (ii) a CernVM Singularity/Apptainer container to provide a Gauss-compatible environment and to mount the subset of CVMFS as if it was a CVMFS client.

Nevertheless, as we explained in Section 1.4.1, Gauss execution can involve different packages, extra packages, options, data and versions. Encapsulating its ecosystem requires a good understanding of the application and/or a large amount of storage to encapsulate the right dependencies. Therefore, different options are available:

- Include the whole LHCb CVMFS repository: would not require any specific knowledge about Gauss and would involve all the necessary files to run any Gauss instance. However, this option would imply a tremendous quantity of storage - the full LHCb repository needs 5.2 Terabytes -, long periods to update the subset and many unnecessary files.

- Include the dependencies of various Gauss runs: as the first option, would not need any specific knowledge about Gauss and would include a few gigabytes of data. Nevertheless, such an option would not guarantee the presence of all needed files and would require a tremendous amount of computing resources to trace Gauss workloads continuously.

- Include all the known dependencies of Gauss: would require a deep understanding of Gauss and its dependencies to include all the required files in a subset of CVMFS. While this option would not involve many computing or storage resources, it would include human resources to update the content of the subset of CVMFS according to the releases of Gauss and its extra packages.

As the size of the LHCb repository is important, we decided to reject the first option. LHCb has access to tremendous computing power: it interacts with hundreds

of WLCG Sites to run Gauss workloads and could theoretically trace them and extract their requirements. In practice, tracing Gauss workloads in production could slow down the applications and their execution, which is not an option. Similarly, LHCb does not have human resources to update the subset of CVMFS according to the changes done. Thus, we chose to combine the second and the third options to propose a light and easy-to-update and maintain solution. The process consists in getting insights into the structure of the Gauss dependencies by running and tracing a small set of Gauss workloads and analyzing the system calls before including the structure in *subcvmfs-builder-pipeline*.

After analyzing 500 commands calling Gauss from the LHCb production environment and tracing 3 Gauss applications using *subcvmfs-builder* (Boyer, 2022a), we noticed that:

- 97% of the workloads studied were running the same Gauss versions (v49r20) with the same extra packages and versions. The versions of Gauss and its extra packages seem related to the underlying architecture.
- 846 Mb of files were needed to run 3 Gauss (v49r20) workloads. About 95% of the size is related to the Gauss version and the underlying architecture, and is common to the Gauss workloads traced, while the 5% left is bound to the options and Geant4 data used that are specific to a given Gauss workload.
- Integrating all the options and Geant4 data related to Gauss v49r20 would correspond to 1.8 Gb of files.

Based on these assumptions, we designed an LHCb-custom layer 2 implementation based on the generic one. As the generic layer 2, it takes the form of a GitLab CI pipeline, which consists in: (i) getting the latest Gauss definitions from the active LHCb production; (ii) extracting the versions and plugins involved; (iii) comparing the results to the previous CI run. The CI runs once per day. If the results are similar to the ones of the previous run, then the pipeline is stopped. In this case, the process lasts around 10 minutes. Else, the not-known Gauss definitions are transformed as inputs for the pipeline and the steps are launched. The input tasks are configured to execute a single event to avoid consuming too much CPU time. The whole execution is variable, but currently takes about 1 hour and 30 minutes.

The CernVM container is deployed manually and is not merged with the subset of CVMFS. The CernVM container allows providing a reproducible environment for the workload but does not require regular updates. Merging it with the subset of CVMFS is also a time-consuming operation, which might take up to 24 hours.

This results in a CernVM container occupying 6.4 Gb combined with a subset of CVMFS covering 6 Gb: dependencies occupy 3.2 Gb of space while 2.8 Gb are required for the *cvmfs_shrinkwrap* metadata. Thus, 12.4 Gb of space on the shared file system of a supercomputer is currently sufficient to run most of the Gauss workloads: 0.24% of the LHCb repository.

To avoid duplicating executions of tasks, one could also integrate the *trace* command of *subcvmfs-builder* within the LHCb production test phase, which consists in running a few events of upcoming Gauss workloads on a given Grid Site. LHCb developers could trace some of them during the process and store the traces in a database. An LHCb-specific *subcvmfs-builder-pipeline* could then periodically fetch the content of the database to build, test and deploy a new subset of dependencies to Mare Nostrum.

Pushing jobs and providing a subset of CVMFS allows the experiment to exploit a large number of supercomputers, but remains heavy. There are a significant number of supercomputers proposing external connectivity and more adapted solutions could be set up to exploit them. In Section 4.3.4, we study different approaches to integrate Gauss tasks on such supercomputers.

4.3.4 Exploiting multi-core/node allocations in environments with external connectivity

Supercomputers embed many-core nodes and generally propose a limited number of allocations. This approach encourage users to exploit multiple cores and even multiple nodes within a same allocation. Thus, we need to improve the fat-node partitioning mechanism, develop mechanisms to leverage multiple nodes in parallel, while making sure that DIRAC Benchmark will keep proposing valuable and accurate CPU power estimations.

Improving the fat-node partitioning mechanism

We added configuration parameters to the DIRAC fat-node partitioning mechanism. In the initial configuration, the `JobAgent` instances embedded in the Pilot-Jobs are able to fetch a single job every cycle, until available cores are all busy. A cycle, by default, lasts a minimum of 120 seconds even if operations are completed. In this context, it would take 96 minutes to load 48 logical cores and 512 minutes for 256 logical cores: most of the cores would remain idle for hours and would represent a waste of CPU time. We changed the default minimum value to 5 seconds, while the operations of a cycle usually needs 15 seconds. It allows to fully load a multi-core allocation 87.5% faster. Parallelizing `JobAgent` cycles would not bring significant benefits as the matching processes and the submissions of the jobs to the `PoolCE` depend on each other and could not be executed simultaneously.

By default, `JobAgent` instances stop when all the logical cores are used or when there is no more job to process. Short-lived and unreliable jobs might create a waste of resources in this configuration. Within an allocation composed of one job running for 12 hours and 47 jobs running for 1 hour, a `JobAgent` instance would stop and would not be able to fetch further jobs to load idle logical cores. Therefore, we prevented `JobAgent` to stop for such reasons. While it allows filling in idle logical cores until almost the end of the allocation, it is efficient in allocations mixing different types of jobs, especially Gauss and user jobs. The combination of these options maximizes the use of single-node allocations in terms of space. Nevertheless, the current approach does not support multi-node allocations, which are essential to leverage the resources of the supercomputer.

Supporting multi-node allocations in environments with external connectivity

Administrators of supercomputers generally limit the number of allocations per user or project. At the same time, users can request multiple nodes per allocation. Getting multiple nodes per allocation would allow using a larger number of nodes, and by extension, to process more pilots and jobs. Moreover, some partitions of nodes only accept allocations of $n > m$ nodes, m being superior to one.

We have imagined several solutions to support multi-node allocations. The first one consists in binding a pilot identifier to a `JobAgent` and a WN (Figure 4.9.a). In

this context, the Site Director submits a pilot to get a fixed, or a range of, number of nodes in the allocation. The LRMS provides the requested nodes and the pilot is installed, either on one of them or on all the nodes in parallel. Each pilot registers as a pilot and runs a `JobAgent` instance. Interacting with the pilots depends on the underlying CE and LRMS. On the one hand, the solution allows to identify every pilot, their status and outputs, and by extension, every node and job associated with them. On the other hand, the number of pilot identifiers would not correspond to the number of allocations and would break the functioning of the Site Director that does not currently support such as use case, especially when the number of nodes allocated is not known in advance. Additionally, we cannot get the status of the pilots until they start running.

The second solution consists in binding a pilot identifier to an allocation, with one `JobAgent` per WN (Figure 4.9.b). In this context, one of the pilots is considered a pilot whereas the other ones are sub-pilots. The Site Director submits a pilot to get a fixed, or a range of, number of nodes in the allocation. The LRMS provides the requested nodes and the pilot is installed, either on one of them or on all the nodes in parallel. Only one pilot is registered in the DIRAC server by the Site Director and all the sub-pilots are identified with the same pilot identifier. Each sub-pilot can then run a `JobAgent` instance, fetches and runs many jobs in parallel. We can interact with the pilot identifier but not the underlying sub-pilots. The solution does not require any change on the Site Director as the pilot identifier is still bound to an allocation. However, it might lead to debugging issues as the access to sub-pilots would not be straightforward. Figure 4.9 offers a schema of the solutions.

We chose to implement the second solution to minimize the changes in the code. As we have a limited number of use cases and they rely on the same LRMS, we currently draw on a specific solution based on the SLURM LRMS and `srun`, which allows running MPI jobs involving many nodes on a cluster managed by SLURM. Within the SLURM communication interface, we implemented a method to use `srun` when the number of nodes requested is superior to 1. It is possible to specify a range of nodes and let the LRMS choose the exact number of nodes at run time according to the load in the cluster. It copies the pilot content and executes it on all the nodes in parallel, under the same pilot identifier. It gets the status of the allocation, which corresponds to the status of the sub-pilots alive. We parameterized the allocation to keep running even if one of the sub-pilots or nodes fails to avoid killing a large number

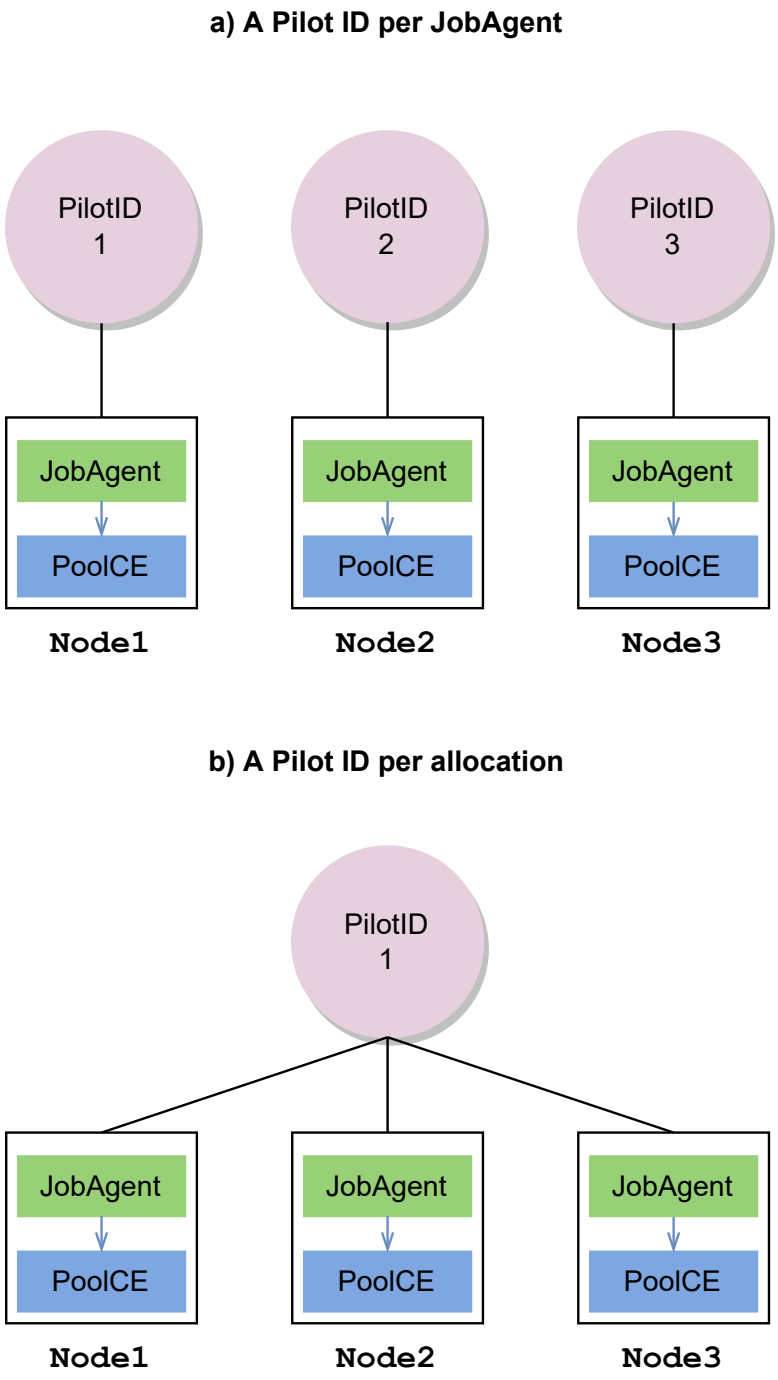

Figure 4.9 – At the top, a schema of three independent pilots running on three WNs in the same allocation; at the bottom, a schema of three pilots bound to the same identifier running on three WNs in the same allocation

of jobs for one error. It automatically extracts the outputs of the pilots on the nodes and orders them by date. In the SLURM communication interface, we reorder the outputs per sub-pilot and per date to provide a clear text to the DIRAC administrators.

Better exploiting CPU work in multi-core allocations

As explained in Section 3.3.1, the Pilot-Job computes DIRAC Benchmark prior to fetching a job. DIRAC Benchmark execution is performed by DIRAC via a script called `dirac-cpu-normalization`. The script calls a single-core execution of DIRAC Benchmark 12 and applies a correction factor to the result in order to construct a community-specific score: a DIRAC Benchmark 16 score in the context of the LHCb experiment for instance. While this approach has worked well within single-core allocation, it might not be adapted to more complex allocations.

In Figure 4.10, we studied DB16 results in a multi-core environment: a node composed of an Intel Xeon E5-2695 v2 Ivy Bridge processor with 24 hardware threads. In the same allocation, we executed DIRAC Benchmark on a range of hardware threads in parallel (1, 2, 5, 10, 15, 20, 24). We can notice a decrease of 12% on average when more than 10 hardware threads are involved in the calculation, whereas we could have expected a linear decrease proportional to the number of hardware threads used. Such variations seem to indicate that (i) the load in an allocation significantly impacts the performance of the processor, and thus, the DIRAC Benchmark scores; (ii) DIRAC Benchmark is not adapted to multi-core environments and would need to be extensively analyzed in this context.

To reduce the impact of such a gap, we introduced a parameter within `dirac-cpu--normalization`. The script is now able to find the number of processors available from the DIRAC configuration and executes as many DIRAC Benchmark copies as logical cores within the allocation. From that, the minimum value computed is selected as the CPU power estimation. This should provide better CPU Power, and thus, CPU work left estimations in multi-core allocations.

To maximize the use of the nodes depending on the SLURM LRMS, we also added code to support SLURM features known as `min-time` and `time`. It lets the system choose the duration of the allocation, between `min-time` and `time`, at run time. When the supercomputer is overused, the LRMS allocates resources for about `min-time`,

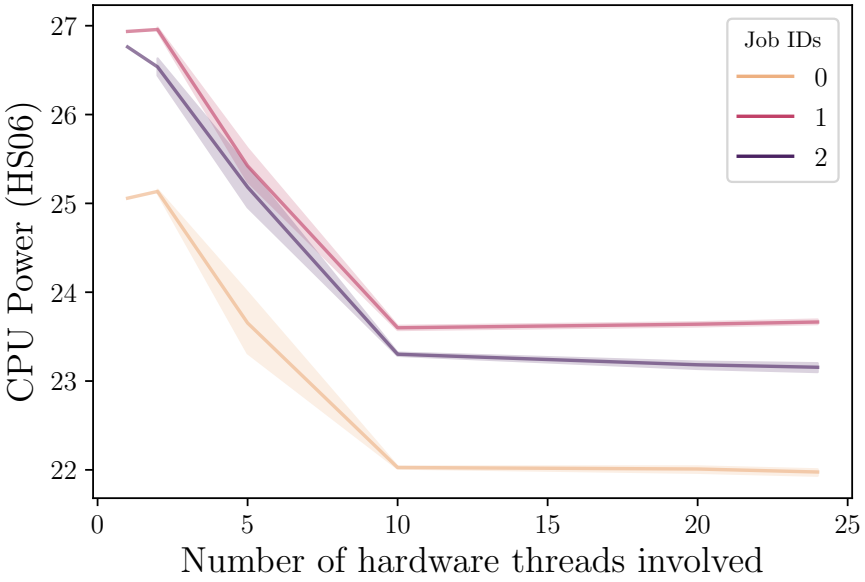

Figure 4.10 – CPU Work estimated by DB16 according to the number of hardware threads involved in the calculation.

while it gives us resources for a duration between `min-time` and `time` when the system is underused. In this way, we quickly get any available resources.

Through Section 4.3, we have strengthened the DIRAC WMS by proposing further mechanisms to support more constraints, mostly within supercomputers. The DIRAC WMS is now able to manage single-core allocations in environment with no external connectivity and better support many-core allocations. Next, we are going to apply these mechanisms on real supercomputers. We study two of them in depth: Mare Nostrum from Spain (Section 4.4.1) and Santos Dumont from Brazil (Section 4.4.2).

4.4 Work on supercomputers: use cases

4.4.1 Mare Nostrum 4

To start integrating their workflows on High-Performance computing resources, LHC experiments can benefit from a collaboration with PRACE and GÉANT (“CERN, SKAO, GÉANT and PRACE to collaborate on high-performance computing”, 2020). This collaboration gives them access to several European supercomputers such as Marconi in Italy and Mare Nostrum in Spain.

Computing resources available

Managed by the Barcelona Supercomputing Center (BSC), Mare Nostrum is the most powerful and emblematic supercomputer in Spain (Vicente and Bartolome, 2010). Mare Nostrum was built in 2004 (Mare Nostrum 1), has been updated 3 times since then (Mare Nostrum 2,3 and 4) and was ranked 82^{th} in the June 2022 Top500 list ("Top500 The List", 2022). The general purpose partition of Mare Nostrum comprises 153,216 cores distributed in 3,456 computational nodes and has a peak performance of 11.15 Petaflops. Each node composing the general-purpose block is equipped with two Intel Xeon Platinum 8160 24 cores at 2.1 GHz chips, and at least 2GB of DDR4-2667 RAM: this configuration matches Gauss requirements. Nodes and users share access to a shared file system, where they have access to 5 TB of storage. Nodes also host a non-CERN compatible version of Linux: SUSE Linux Entreprise Server 12. Administrators manage the nodes using the SLURM LRMS.

As mentioned in Section 1.4.2, a certain number of CPU hours are reserved for the experiments. Allocations are renewed every six months and tailored for the needs of the experiments. For its first round, the LHCb experiment receives an allocation of 50,000 CPU hours for testing purposes. Currently, the collaboration has access to a 6-month allocation of 500,000 CPU hours and has been allowed to manage a maximum of 366 allocations in parallel. Once reached, the experiment can still exploit the nodes of Mare Nostrum via a special queue. In this case, jobs are not prioritized in the system.

Nevertheless, Mare Nostrum is more restrictive than a traditional grid site on WLCG. By default, CVMFS is not mounted on the nodes and administrators would not allow it. Worker nodes and edge nodes have no external connectivity and no service can be installed on the edge nodes. Thus, it is impossible to supply nodes with Gauss tasks as we generally do on a WLCG grid site. Next, we are going to focus on solutions we set up to exploit the current EuroHPC and PRACE allocation and pinpoint the limitations of our approach.

Software blocks leveraged

To provide tasks with Gauss dependencies and a reproducible environment, we first created and transferred a CernVM Singularity/Apptainer container to the shared

file system of Mare Nostrum. Then, we set up the LHCb *subcvmfs-builder-pipeline*, introduced in Section 4.3.3, to push the latest Gauss dependencies to the shared file system of the supercomputer.

To ease the management of the workloads on Mare Nostrum, the team of the Port d'Informació Científica (PIC) installed an ARC instance on their infrastructure. We installed and configured a *PushJobAgent* to push Gauss tasks to Mare Nostrum via the ARC instance hosted by PIC. With the help of the PIC administrators, we fine-tuned the ARC instance to leverage RunTime Environments (RTE). RTEs allow users to flexibly contextualize a job execution environment. We set up an RTE to launch Gauss tasks within a CernVM container mounting a subset of CVMFS as CVMFS.

Results

We have run Gauss jobs on Mare Nostrum for 1 month. We have measured the number of jobs processed per hour over this period using the accounting service of the LHCbDIRAC production instance. Figure 4.11 shows the results. We configured the *PushJobAgent* instance to manage 200 jobs in parallel first and we increase the limit to 300 jobs from the 7^{th} day. We also modified the site parameters so that Gauss tasks run for about 12 hours.

During this month, 96.3% of the jobs ran successfully, while 3.7% of them failed. Failed jobs occurred after issues within the supercomputer. The last errors were caused by an unexpected issue with the shared file system causing the blocking of I/O operations and about 84% of the job failures. We can notice a gap towards the middle of the month. BSC requires the LHCb collaboration to manually and electronically submit a report about the status of the project every two weeks. Any oversight results in the blocking of the allocation of the CPU hours until the report is submitted. We can also observe that the *PushJobAgent* rarely process the maximum number of jobs allowed: in average, 179 jobs are managed in parallel. This will be investigated below.

According to BSC data, our approach is sufficient to exploit an allocation of 750,000 CPU hours for 6 months (Figure 4.12). In a month, we have consumed 133,630 (17.82%) CPU hours available. Consumption of the CPU hours remains significantly variable through the weeks, which can be explained by the configuration changes, issues on the supercomputers and the forgotten report mentioned above.

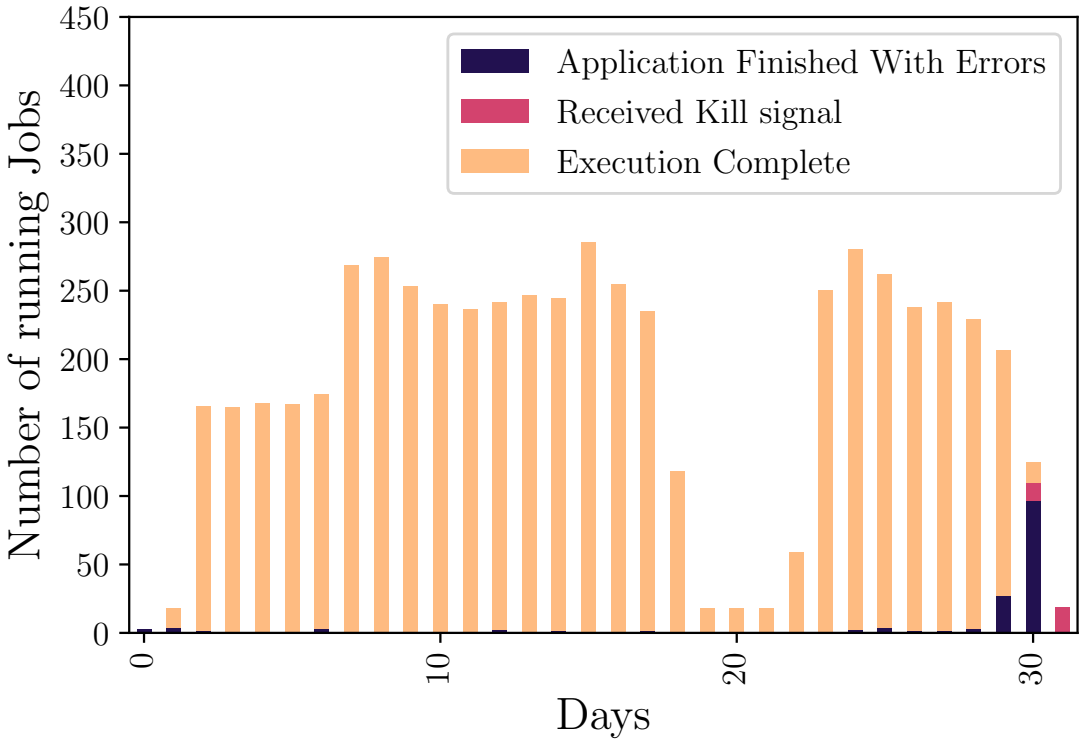

Figure 4.11 – Number of jobs processed in parallel averaged per day on Mare Nostrum during a month. *Execution Complete* corresponds to the jobs done; *Application finished with errors* represents jobs that failed because of an error during the event processing; *Received Kill signal* refers to jobs that have to be killed by the system. Sometimes, killed jobs are defined as *Application finished with errors*.

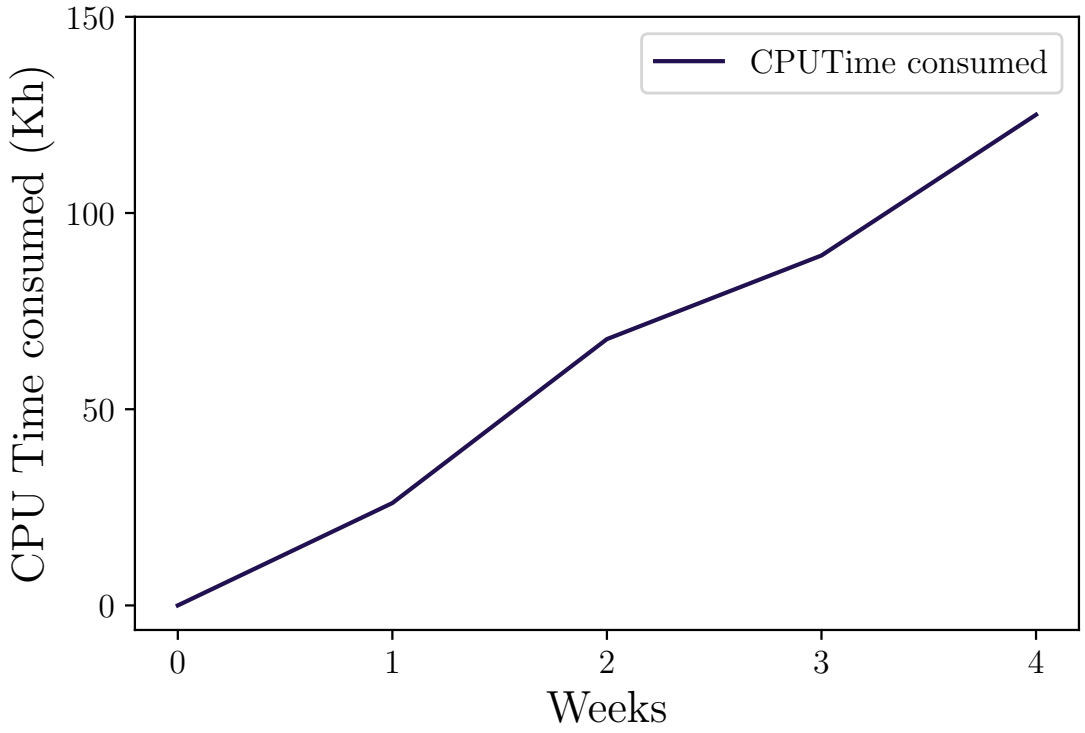

Figure 4.12 – CPU hours consumed in K hours on Mare Nostrum for one month averaged per week.

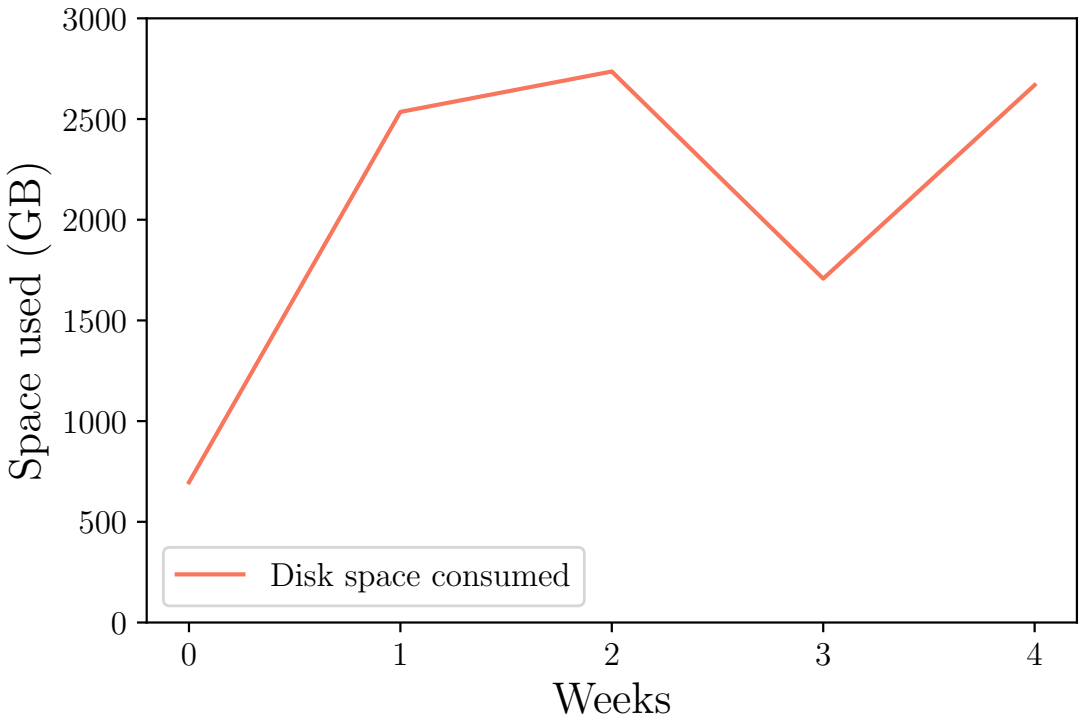

Figure 4.13 – Disk storage consumed in GB on Mare Nostrum for one month averaged per week.

BSC data also provide insights about the storage usage (Figure 4.13). We have a limit of 5 TB and we let the system clean the inputs and outputs of the LHCb jobs from time to time. The current approach generates around 3 TB of data on the shared file system. The CernVM container represents 5 GB while the subset of CVMFS currently weighs 40 GB. To avoid exporting the whole subset of CVMFS every time an update is required, we keep the existing files on the system, which explains the size of the subset after 3 months of updates. 300 jobs running simultaneously for 12 hours are responsible for 98.5% of the storage used, which represents 60% of the available storage. To exploit further jobs on the supercomputer, we would need to clean inputs and outputs once jobs are done. The *PushJobAgent* instance should also be able to manage a larger number of jobs.

We monitored the computing resource usage of the machine hosting the *PushJobAgent* instance interacting with Mare Nostrum for one week (Figure 4.14). The machine comprises an Intel Xeon Skylake CPU of 16 cores (32 hardware threads) and 64 GB of RAM, and 48 GB of storage are reserved for the DIRAC installation and data. It is

worth noting that the *PushJobAgent* instance is the only DIRAC component running on the machine.

Our approach has a minimal impact on CPU usage: 98% of the hardware threads remain idle most of the time (Figure 4.14.a). In the same way, the space used observed in Figure 4.14.b is the share of the DIRAC installation compared to the whole system (about 40%). Peaks correspond to input and output data related to jobs managed by the *PushJobAgent* instance. It represent a few MB of storage. We implemented a cleaning mechanism to prevent overloading the machine. Conversely to the previous metrics, the memory of the machine is significantly impacted by the solution. An analysis of the running processes emphasizes the association between running jobs and memory consumption. Managing 300 jobs with the current approach requires about 54 GB of RAM, namely around 180 MB per job.

As we have seen in Section 4.3.2, the *PushJobAgent* instantiates a *JobWrapper* template and a *dirac-jobexec* processes for each job. While the execution happens on the supercomputer, each process remains running to interrogate the ARC instance every 2 minutes and gets the status and outputs of the jobs. Even though the size of the processes is not important, the sum of many of them results in a heavy memory load on the machine. Furthermore, as the Site Director, the *PushJobAgent* instance executes 500 cycles of 120 seconds by default, before restarting. Because it spawns processes, they remain bound to it until the end of their execution, forcing it to stay active until the end of all the processes. As a consequence, a single *JobWrapper* template started towards the end of the *PushJobAgent* instance execution can force it to stay active whereas no other *JobWrapper* templates is running. In practice, this blocks the submission of further jobs for many hours and explains why the instance rarely processes the maximum number of jobs in parallel.

The current approach constitutes a simple solution, minimizing the changes in the code. To manage further jobs, short-term solutions could consist in tweaking the parameters of the machine and the configuration of the *PushJobAgent* instance. Dumping and reloading the memory of the *dirac-jobexec* processes could free up some space but would likely stress the storage while scaling up at some point. As jobs are all very similar, deduplicating memory could help until we start mixing Gauss tasks with other types of jobs. In addition, configuring the *PushJobAgent* instance to run a larger number of cycles would minimize issues related to processes blocking the

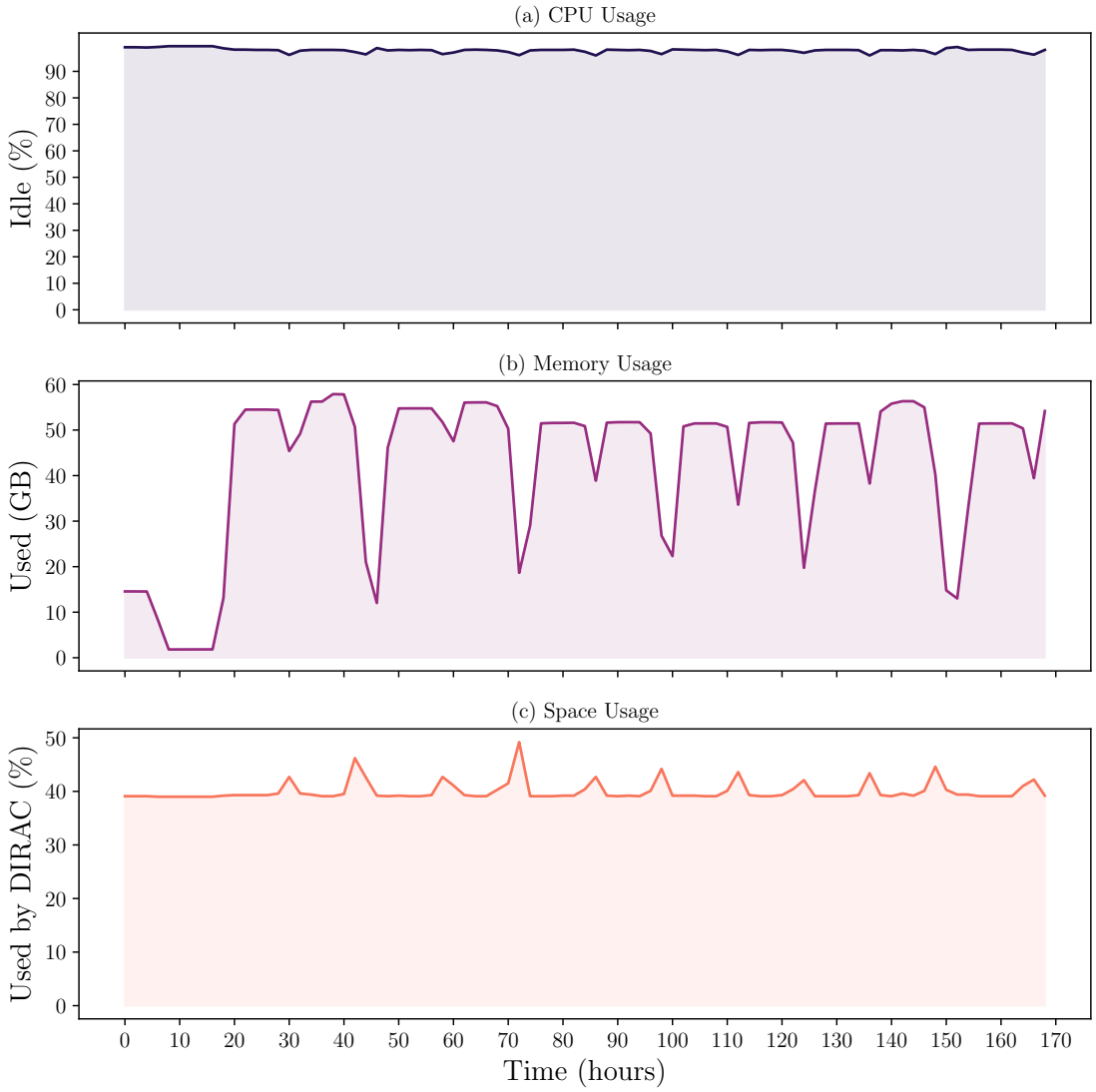

Figure 4.14 – CPU and memory usage of a *PushJobAgent* instance running in one of the production servers of LHCbDIRAC, as well as disk occupancy of the DIRAC installation. The instance solely targets Mare Nostrum.

end of the agent execution.

In the long term, we should design the WMS workflow differently. The *PushJobAgent* instance could work similarly to a Site Director. It would execute directly the *JobWrapper* template and a modified version of *dirac-jobexec* that would split pre-processing and post-processing steps and modules. In this context, the *PushJobAgent* instance would fetch jobs and execute each *JobWrapper* template in parallel. It would instantiate the first part of the DIRAC workflows containing the pre-processing steps in parallel. After that, the *PushJobAgent* submits the tasks to a remote CE and monitor multiple tasks with a single command using the same communication interface. Finally, it would execute the second and last part of the DIRAC workflows containing the post-processing steps in parallel.

4.4.2 Santos Dumont

LHCb also benefits from a close collaboration with the Laboratório Nacional de Computação Científica (LNCC) in Brazil, which provides opportunistic allocations to the collaboration.

Computing resources available

The LNCC is a public, interdisciplinary research center focused on the simulation and computational modeling of complex problems (Gitler et al., 2020). The center coordinates SINAPAD, a network of several supercomputers funded by the Brazilian Ministry of Science, Technology and Innovations (MCTI) including Santos Dumont, also known as SDumont.

In 2015, the French company Atos installed Santos Dumont (SDumont), the first Petascale supercomputer in Brazil, designed by Bull ("Archive for Santos Dumont supercomputer", 2019). Updated in 2019, SDumont remained the largest supercomputer dedicated to research in Latin America until 2020. Ranked 424^{th} of the Top500 in June 2022, the supercomputer still acts as a central node of SINAPAD, with a processing capacity in the order of 5.1 Petaflops and a total of 36,472 CPU cores, distributed in 1,134 computational nodes, mainly composed of CPUs with multi-core architecture ("SDumont", 2021). Table 4.1 presents the configuration of SDumont in further detail.

Table 4.1 – Configuration of SDumont

Number of Nodes (hardware threads)	Name	Processor	Co-Processor	Memory
504 (12.096)	B710	2x Intel Xeon E5-2695v2 Ivy Bridge		64GB DDR3
198 (4.752)	B715	2x Intel Xeon E5-2695v2 Ivy Bridge	2x NVIDIA K40 GPU	64GB DDR3
54 (1.296)	B715	2x Intel Xeon E5-2695v2 Ivy Bridge	2x Xeon Phi 7120	64GB DDR3
1 (240)	MESCA 2	16x Intel Xeon Ivy Bridge		6TB
246 (11.808)	BS X1120	2x Intel Xeon Cascade Lake Gold 6252		384GB
36 (1.728)	BS X1120	2x Intel Xeon Cascade Lake Gold 6252		768GB
94 (4.512)	BS X1120	2x Intel Xeon Cascade Lake Gold 6252	4x NVIDIA Volta V100 GPU	384GB
1 (40)	BS	2x Intel Xeon Skylake Gold 6148	8x NVIDIA Tesla V100-16GB	384GB DDR4

Table 4.2 – Configuration of the *cpu* partitions

Partition	Min. Number of Nodes	Max. Number of Nodes	Max. Wall-Clock	Max. Number of Waiting Jobs	Max. Number of Running Jobs
cpu	21	50	96h	24	4
cpu-dev	1	4	20m	1	1
cpu-small	1	20	72h	96	16
cpu-long	1	10	31d	18	3
cpu-shared	1	20	72h	96	16
cpu-scal	51	128	18h	8	1

Nodes are interconnected by an Infiniband network offering low latency and high throughput for access to the file system and communication between processes. In combination with Infiniband, SDumont also integrates a parallel Lustre file system with a gross storage capacity of 1.7 PB. Local system administrators have installed RedHat Linux 7.6 - Gauss-compatible - on the nodes and manage computational resources and their use with SLURM. They also allow external connectivity from the WN. SDumont is only available through SSH via the LNCC VPN.

Contrary to Mare Nostrum, the LHCb experiment does not benefit from reserved CPU hours on SDumont. Computing resources available have to be used opportunistically. We have access to 504 B710 nodes, which corresponds to a total of 12,096 hardware threads, spread through 5 partitions. Their configuration and the number of resources granted to the users are available in Table 4.2. The *cpu-shared* partition allows core allocation, so we allocate 20 hardware cores per allocation instead of the whole node. We mainly focus on *cpu-small* and *cpu-long*, where we allocate one node of 24 hardware threads per allocation. With the current configuration, we have the possibility to run a maximum of $16 \times 24 + 3 \times 24 + 16 \times 20 = 776$ jobs at the same time.

Software blocks leveraged

Thanks to a close collaboration with LNCC, system administrators installed and mounted CVMFS on the nodes, and created an SSH bridge from an LHCbDIRAC host to an edge node of the supercomputer so that LHCb administrators and developers do not need to deal with the presence of a VPN. We have leveraged Pilot-Jobs and the fat-node partitioning to exploit the available resources opportunistically.

Results

We have run Gauss jobs on *cpu-small*, *cpu-long* and *cpu-shared* for 3 months. We have measured the CPU usage of the resources over this period using the accounting service of the LHCbDIRAC production instance (Boyer, 2021b). Figure 4.15 shows the results. On average, about 196 CPU seconds are consumed per second, which corresponds to 196 jobs running simultaneously. This represents 1.6% of the B710 nodes available, and 25% of the theoretic maximum number of jobs (776) that we could run on the 3 partitions that we are using. Besides, the theoretic maximum number has not even been reached, the maximum being 514 CPU seconds consumed per second. The amount of CPU seconds consumed per second is significantly variable over time. Furthermore, jobs that fail before their completion consume 15% of the CPU seconds used. We are going to study the source of the jobs failures firstly, and the usage of the resources secondly.

The LHCbDIRAC production instance produces plots of the number of jobs per job status. We extracted the error statuses in Figure 4.16. We observe that 99.4% of the failures occur because pilots were not running during the execution of the jobs, which means that pilots end unexpectedly before the end of the jobs (Job stalled: pilot not running). To avoid many job failures, we did not leverage multi-node allocations and we limited the number of jobs to the number of hardware threads available on a node, which prevents replacing a job once finished before the end of the allocation.

To get further information about this issue, we extracted details about 1840 pilots by interrogating the SLURM instance of SDumont. Figure 4.17 presents the number of pilots that SDumont processed, classified by partition and status. 46.8% of the pilots failed because they run out of the time that SLURM allocated for them, which explains the large amount of stalled jobs (Timeout). In *cpu-long* and *cpu-small*, there

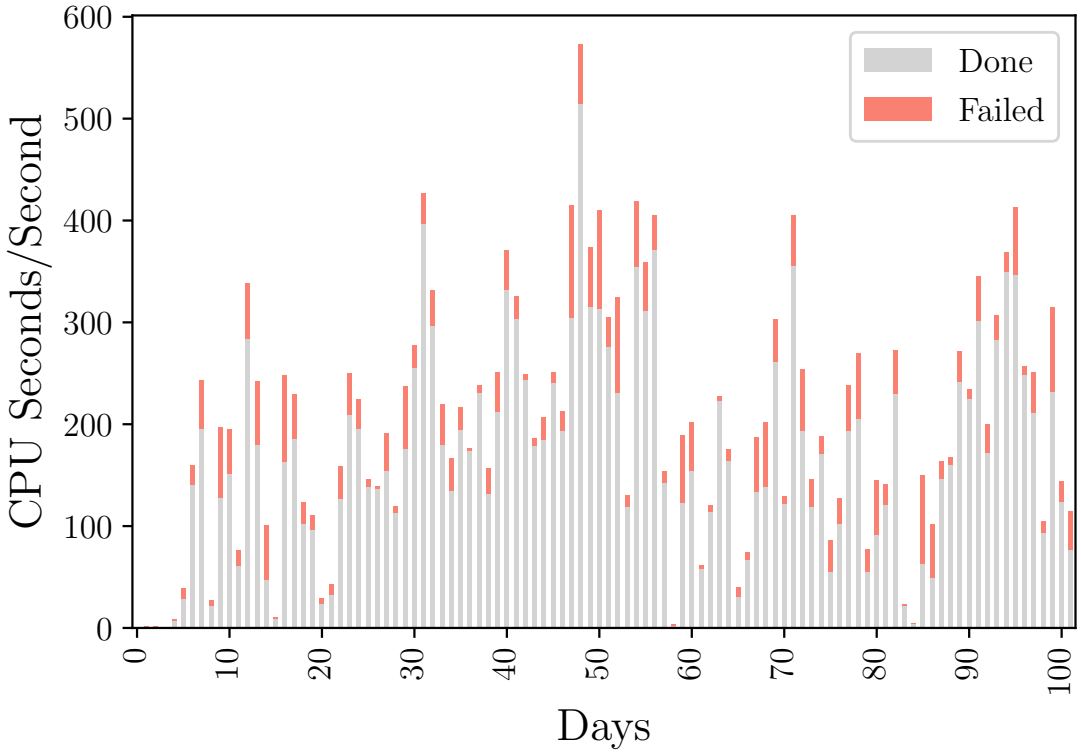

Figure 4.15 – CPU seconds used on SDumont per real second. *Done* corresponds to the CPU usage of the jobs done; *Failed* to the CPU usage of the jobs that failed.

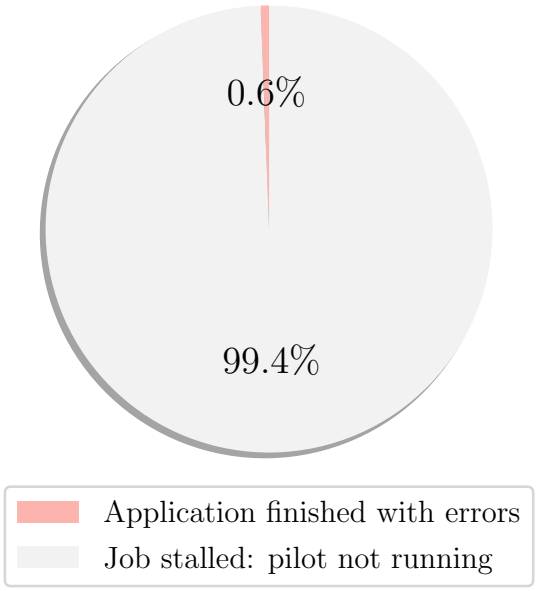

Figure 4.16 – Percentage of jobs per error status.

were even more *Timeout* pilots than *Completed* pilots, namely the pilots managing to finish their execution properly. The *cpu-shared* has a better ratio of *Completed* pilots. Contrary to *cpu-small* and *cpu-long* that share a large part of their nodes with each other, *cpu-shared* has a dedicated set of nodes that cannot be used by any other partition. This could partially explain the difference in ratios between the partitions. Despite it has the worst ratio of *Completed* pilots (34.72%), *cpu-small* is also the most used partition for LHCb workloads (61.4%). The SLURM instance seems to favor the usage of *cpu-small* over *cpu-long* and *cpu-shared*.

To better understand the *Timeout* issues, we focus on the CPU work computation. Thus, we studied the status of the pilots based on the available CPU time, and the CPU power resulting from DB16. Figure 4.18 shows the results in the form of a scatter plot. Outputs of the *Timeout* are empty, thus we do not have information about the number of jobs they manage to process, and this explains the fact that all *Timeout* pilots have a total of 0 job on the scatter plot. We distinguish 4 main clusters of pilots:

- in the top-left corner (Cluster1): only *Completed* pilots from *cpu-shared* that

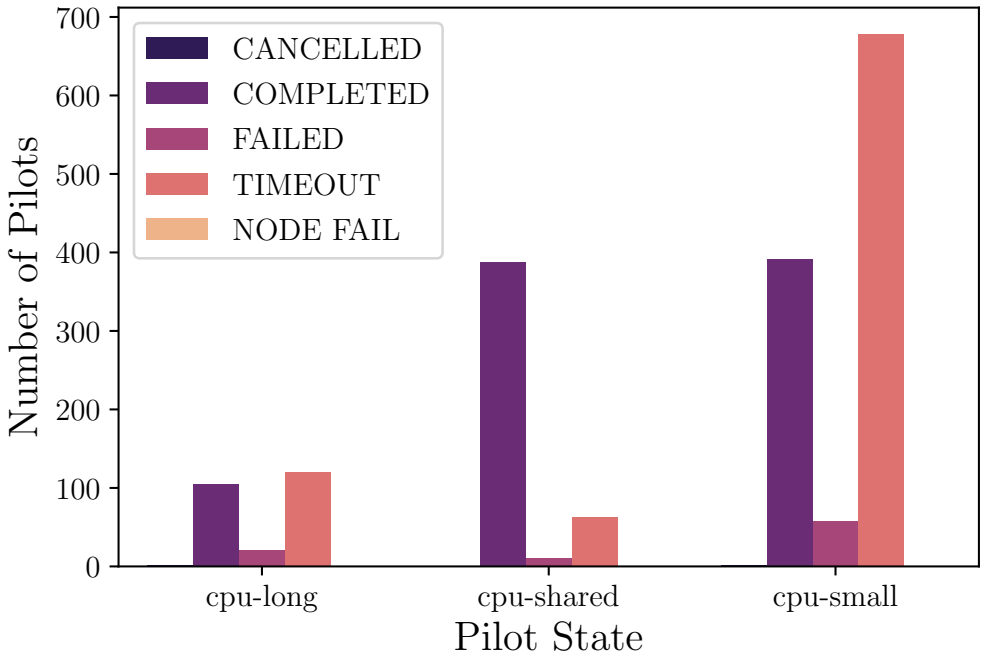

Figure 4.17 – Number of pilots processed per partition, classified according to their Slurm status.

processed 20 jobs. Pilots had 250,000 seconds at their disposal, and the CPU power was estimated between 16 and 19 HS06.

- in the top-right corner: same features as *Cluster1*, except that CPU power was estimated between 20 and 24 in this case. As both clusters are close, we will group them and call them *Cluster1*.
- in the bottom-left corner (Cluster2): a combination of *Completed* and *Timeout* pilots from all the partitions but especially *cpu-small*, that processed between 0 and 24 jobs. Pilots had between 0 and 100,000 seconds at their disposal, and the CPU power was estimated between 16 and 19 HS06. We can see that the higher the CPU power, the more we have *Timeout* issues. Nevertheless, there is no clear boundary.
- in the bottom-right corner (Cluster3): mostly *Timeout* pilots from all partitions, but especially *cpu-small*. Pilots had between 0 and 100,000 seconds at their disposal, and the CPU power was estimated between 21 and 24 in this case, which is the major difference with *Cluster2*.

The CPU power seems to have a significant impact on the failures. To get an insight into the accuracy of the CPU power estimations, we compared the $CPU_{work}\ of\ 1event$ values of several productions computed on the test site (further explanation in Section 2.4.2) with the $CPU_{work}\ of\ 1event$ values computed from several allocations on the supercomputer (Figure 4.19). A total of 2754 completed jobs were analyzed. Failed jobs were not taken into account as they did not include information such as the number of events successfully processed. Results suggest that DB16 overestimates the CPU power of the Intel Xeon E5-2695v2 Ivy Bridge processors on SDumont by 27% on average, which is critical. This issue echoes results obtained in Section 3.3 and also puts into question the viability of DB16 in multi-core environments, which has never been tested in a real use-case beforehand. We would need to run an extensive analysis of DIRAC Benchmark in this context to identify issues, associate them with the results from Section 3.3 and propose a reliable solution to improve it.

On the one hand, there are external factors that we cannot control that could explain the variability observed in Figure 4.15. For instance, we probably never reached the theoretical maximum number of jobs because of the competition for the resources within the supercomputer. Indeed, many other VOs use these resources at the same

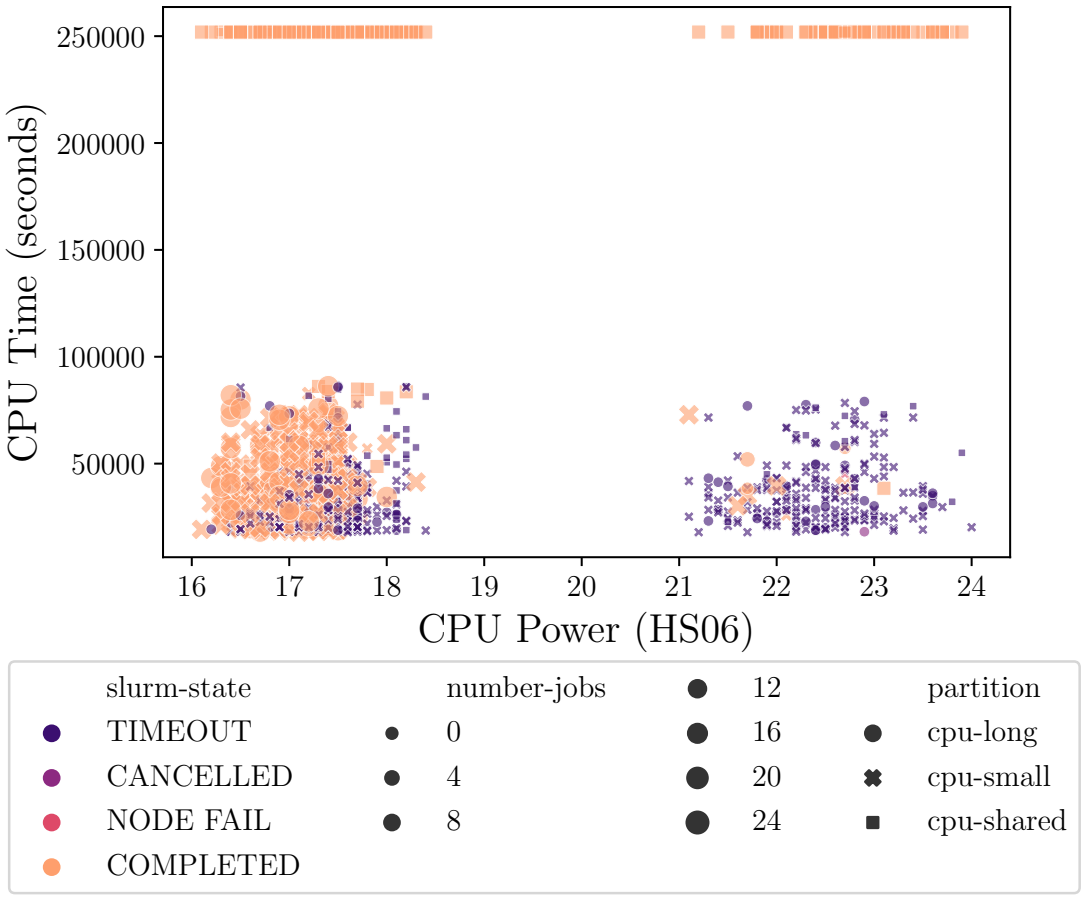

Figure 4.18 – Status of the pilots, the number of jobs they processed and the partition they used depending on the CPU time they had and the CPU power they computed.

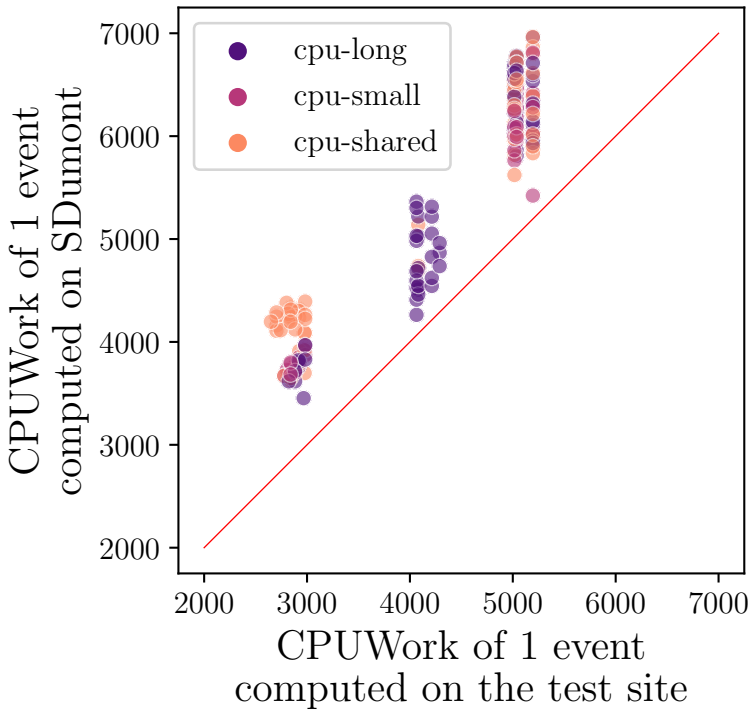

Figure 4.19 – $CPU_{work}\ of\ 1\ event$ computed on the test site compared to the $CPU_{work}\ of\ 1\ event$ computed on a SDumont WN including an Intel Xeon E5-2695v2 Ivy Bridge processor.

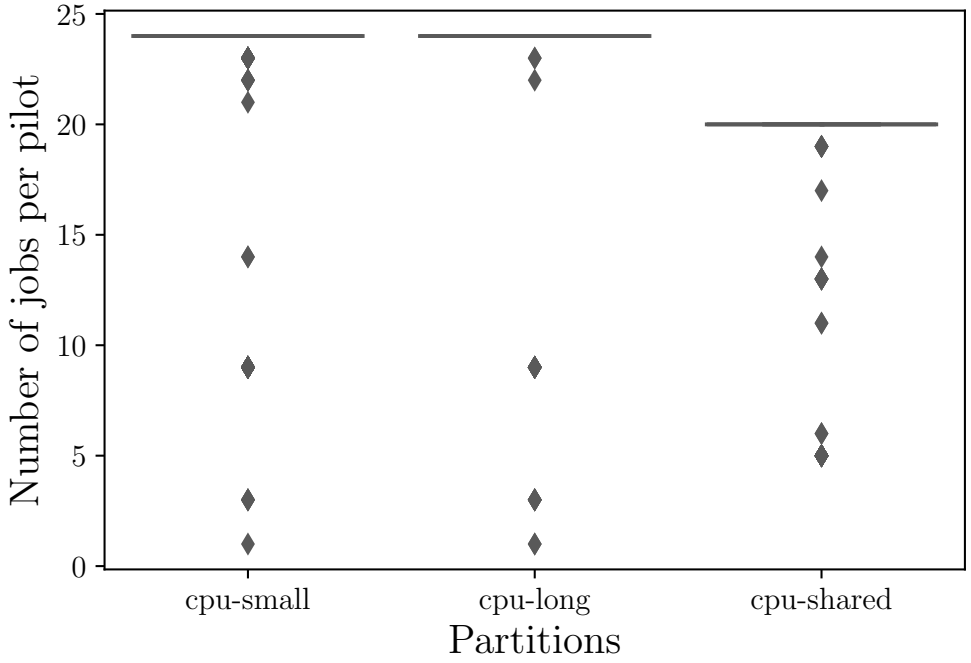

Figure 4.20 – Number of jobs processed per pilot, classified by partition.

time and some of them even have more priority compared to LHCb. We also got some issues with the CVMFS installation and the SSH access that was blocked from time to time. During these moments, we had no means of running jobs on SDumont, and this would likely explain the values close to 0 or some of the decreases in Figure 4.15. However, we have many waiting pilots in the partitions in case we cannot submit newer pilots: the LRMS can continue to execute waiting pilots while we deal with the issues.

On the other hand, we can monitor some of these factors to identify problems that we could resolve. For instance, we analyzed the outputs of the *Completed* pilots to determine the number of jobs that they usually process. According to Figure 4.20, pilots generally use all the hardware threads, available for them, on the WNs.

Figure 4.21 presents the CPU time that *Completed* pilots effectively use, compared to the CPU time that SLURM allocates for them. They are classified by cluster - according to the CPU time allocated and the CPU power computed -, which are defined above. Pilots from *Cluster1* - which is characterized by a high CPU time - use generally 35% (median) of the CPU work allocated, which is rather low compared to

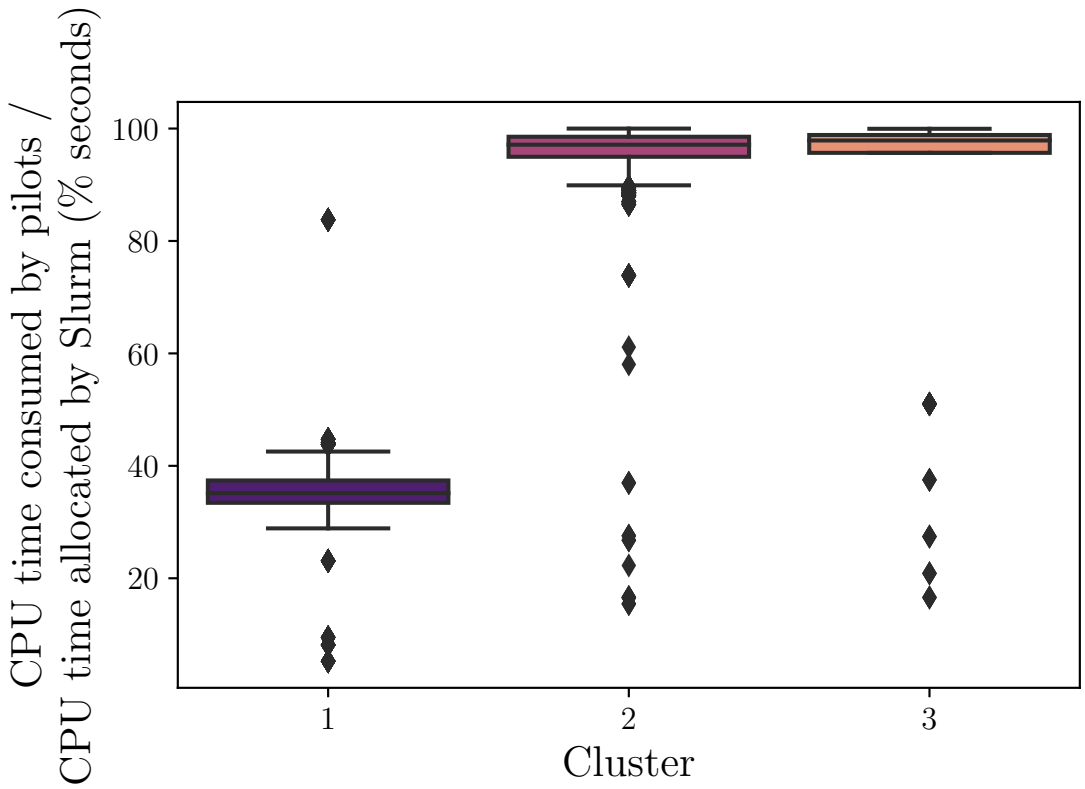

Figure 4.21 – Percentage of CPU time, that SLURM allocates to the pilot, effectively used, classified by cluster. *Cluster1* includes all the pilots with a CPU time available superior to 200,000; *Cluster2* comprises all the pilots with a CPU time available inferior to 200,000 and a CPU power approximation inferior to 19; *Cluster3* contains all the pilots with a CPU time available inferior to 200,000 and a CPU power approximation superior to 19.

Cluster2 and *Cluster3* that use 97% (median) of the CPU time available. Even though the number of Gauss events is defined at run time, there is a maximum number of events that each job can process, which seems to be reached in *Cluster1*. As jobs are not replaced once they are finished, we do not maximize the usage of allocated resources. Pilots from *Cluster2* and *Cluster3* use much more CPU time than allowed (the maximum being 75% of the total CPU time allocated).

We also analyzed the CPU time effectively used by jobs compared to the CPU time that SLURM allocates to the pilots (Figure 4.22). Jobs from *Cluster1* run during 31% (median) of the allocated time. The standard deviation is relatively small, values are close to the mean, which demonstrates that 34% (3-quantiles) of 250,000 seconds - 72,500 seconds - are sufficient to run 75% of the Gauss jobs in their current form.

The maximum job duration represents 47% of the allocated CPU time, which shows that pilots having around 250,000 seconds available could run 48 (2×24, the number of hardware threads on the B710 nodes) Gauss jobs per node. Jobs from *Cluster2* consume generally between 83% (1-quantile) and 91% (3-quantiles) of the time at their disposal, which shows that jobs exceed the CPU time margin (75% of the CPU time allocated) and supports the fact that the CPU power computation is not accurate enough on this platform, and thus should be revised. *Cluster3* has a larger standard deviation (26), probably because there are fewer jobs in *Cluster3* (148) compared to *Cluster1* and *Cluster2* (respectively 6679 and 11449). Nevertheless, we can note that the median of *Cluster3* is much smaller (48%) than the one of *Cluster2*, for almost the same CPU time available. The difference likely explains the fact that these pilots, having a higher CPU power value, managed to complete their execution: jobs run less long than in *Cluster2*.

Pilot-Manager on the edge node

The SSH connection provided by the SDumont administrators is unique and only accessible to some members of the LHCbDIRAC team, and thus, might not run in the long term. Additionally, the SDumont team wants us to renew our access passwords frequently. Trying to automatically submit pilots many times at these moments blocks our accounts for many hours or even days.

To be less dependent on this connection, we have built a DIRAC prototype installed on an edge node of SDumont, based on the Harvester principle. The prototype consists of a Site Director that directly submits pilots from the edge node to SLURM. Because system administrators do not allow services running from the edge nodes, we set up a cron job instantiating a Site Director every 10 minutes and running a single cycle. The Site Director communicates with the LHCbDIRAC production server via HTTPS, to get waiting jobs as well as registered pilots.

This approach raises two issues:

- LHCbDIRAC administrators would need to set up a delivery and update management system. Indeed, LHCbDIRAC administrators deploy releases every two weeks on the dedicated servers. While the operation could be done manually on SDumont, this would be cumbersome.

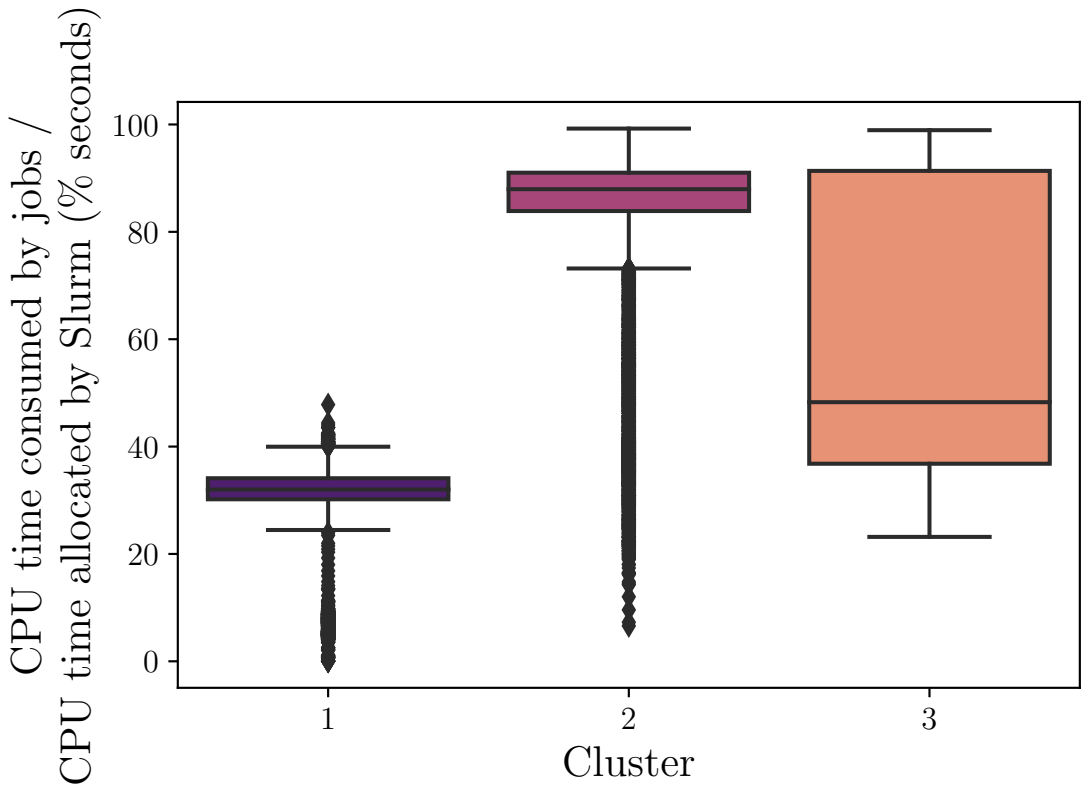

Figure 4.22 – Percentage of CPU time, that Slurm allocates to the job, effectively used, classified by cluster. *Cluster1* includes all the pilots with a CPU time available superior to 200,000; *Cluster2* comprises all the pilots with a CPU time available inferior to 200,000 and a CPU power approximation inferior to 19; *Cluster3* contains all the pilots with a CPU time available inferior to 200,000 and a CPU power approximation superior to 19.

- The Site Director would need the host X.509 certificate of the edge node of SDumont to securely communicate with LHCbDIRAC services. Local system administrators do not provide such a certificate.

The solution would need to be expanded and accepted by the local system administrators to be used on additional supercomputers. We have not developed the idea on SDumont as the SSH connection is sufficient to run Gauss jobs on the supercomputer despite the issues that we mentioned above. Figure 4.23 presents a schema of the SSH delivery system and the DIRAC extension on the edge node.

4.5 Conclusion

This work promotes and relies on research efforts conducted by LHC experiments which keep developing tools to exploit supercomputers. We mainly highlighted solutions related to Monte Carlo simulations as they are, in general, the easiest tasks to offload from WLCG. This chapter should assist any community working with distributed, shared and, especially, constrained computing resources in processing a growing amount of data. The sections are relatively bound to LHCb but they could be transposed to any other similar projects.

We have started by specifying our requirements and abilities to deal with constrained environments (Section 4.2). Next, we have presented the plan to guide any communities in this process: it raises questions about the constrained environment and prescribes generic answers to overcome the issues (Section 4.3.1). The plan is followed by implementations of these generic answers, specific or not, to our project. In this case, we added support for resources with no external connectivity: (i) *PushJobAgent*, a DIRAC agent to perform pre and post operations from LHCb servers and only submit the tasks to supercomputers (Section 4.3.2); (ii) *subcvmfs-builder-pipeline*, a generic CI pipeline to automatically generate and deploy subset of CVMFS (Section 4.3.3). We also improve abilities of DIRAC to manage many-core allocations in supercomputers with external connectivity (Section 4.3.4).

Finally, we have presented supercomputers and followed the logic of the plan to integrate Gauss tasks on their computing resources (Section 4.4). It is worth noting that the examples that we have are significantly different from each other in their architectures and policies. It raised two main limitations in our solutions: *PushJobAgent*

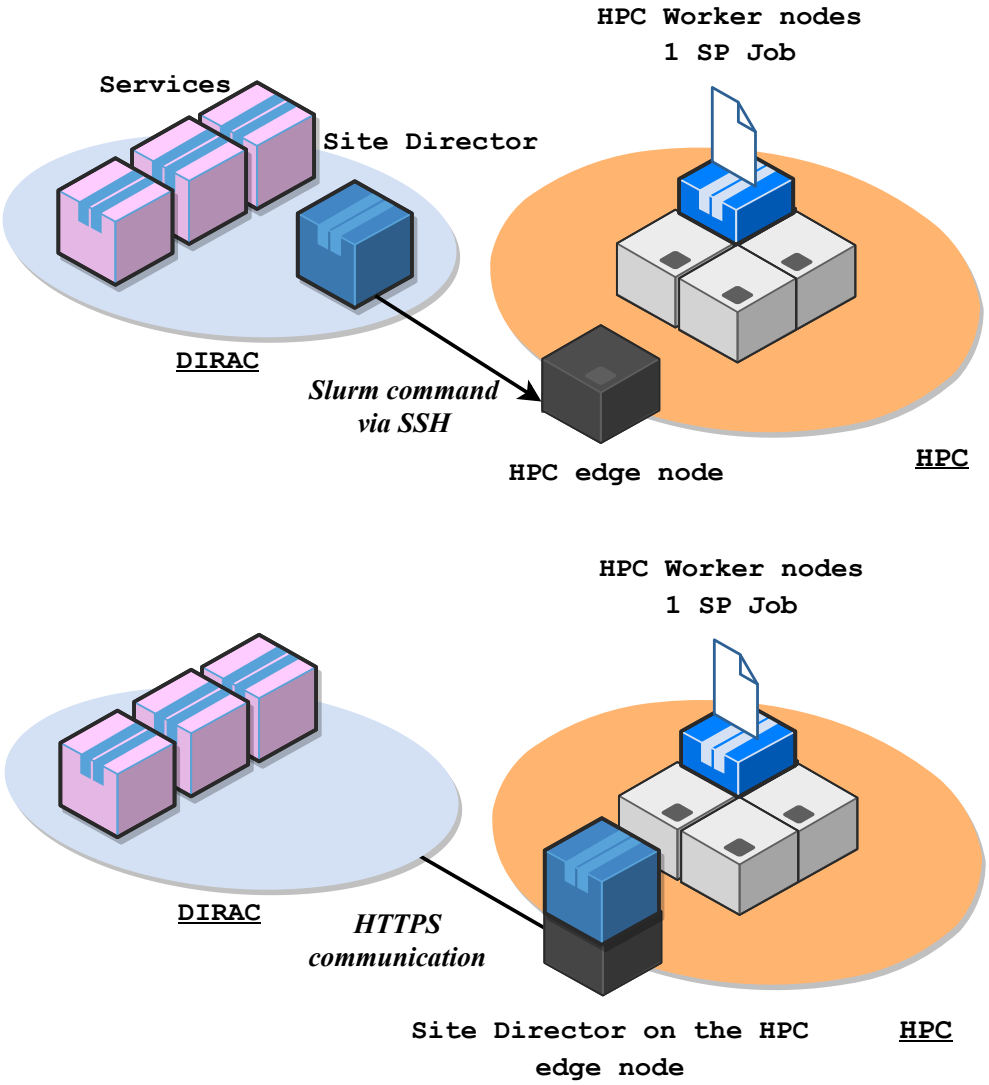

Figure 4.23 – At the top, a pilot delivery system based on SSH; at the bottom, a pilot delivery prototype based on HTTPS and the edge node

cannot scale to manage more than 300 Gauss tasks in parallel and DIRAC Benchmark may not be adapted to certain processors in supercomputers. Nevertheless, we have been able to exploit an allocation of 750,000 CPU hours on Mare Nostrum and opportunistic allocations on Santos Dumont, which represents around 500 jobs in parallel. While this is still insignificant compared to the resources that we can exploit on WLCG, it lays the foundations for further and larger upcoming allocations of computing resources.

Conclusion

This thesis is made in the context of the LHCb experiment and focuses on the efforts that have to be made in the DIRAC WMS to provide additional computing power in order to manage the upcoming LHCb workload from the LHC Run3 and HL-LHC infrastructure. Through the 4 chapters of this manuscript, we have: (i) presented the nature of the LHCb tasks and computing resources at our disposal; (ii) studied how to efficiently leverage them; (iii) implemented and evaluated solutions to better exploit already available distributed grid resources and integrate LHCb workload on supercomputers.

Summary

First, we have presented consumer needs and supplier offers at the dawn of the 2020s (Chapter 1). After having defined tasks and workloads, their structures and requirements, we have surveyed the evolution of computing components, classes and paradigms used to widely share computing resources. AI is currently influencing large companies leading the hardware market and shaping the computing landscape of a new decade: specialized and heterogeneous components aggregated in large infrastructures and mostly offered as cloud computing services. A focus has been made on the difficulty to integrate tasks on many distributed computing resources, especially when owned by independent institutions. The LHCb collaboration is following the trend to handle a growing amount of data. Many developments focus on Monte-Carlo simulation tasks that represent 71.7% of the offline activities and consume 91.1% of the CPU time available. Developers have started to integrate AI in fast simulations to speed up the process but the experiment still primarily depends on full simulation tasks that remain CPU-intensive and require a large number of computing resources to provide meaningful results. Decision-makers are encouraging experiments to integrate their workloads on supercomputers, which represents a huge but highly-heterogeneous computing power. Thus, we established a strategy consisting in integrating Monte-Carlo simulation tasks on supercomputers and improving

the use of WLCG computing resources to process other types of tasks.

Then, we have surveyed different approaches to integrate CPU-intensive and embarrassingly parallel tasks with limited inputs on distributed and shared computing resources, mainly related to grid computing clusters and supercomputers (Chapter 2). We have defined the grid architecture and enumerated existing provisioning models. It is worth noting that the Pilot-Job paradigm has had a great impact in the High-Throughput Computing community for two decades as it addresses issues of the push model, but remains impossible to set up in constrained environments with no external connectivity as we often find in supercomputers. We have also provided solutions to cope with result reproducibility in distributed and heterogeneous computing resources. Both HPC and HTC communities have been concerned about the subject and worked on containers, package managers as well as a distributed file system to export environments across many sites. CVMFS, combined with package managers, has been largely used for years in WLCG. Many teams have worked on project-specific solutions to integrate the model in constrained resources with no external connectivity, but there is still a lack of genericity. We have reviewed current approaches to efficiently exploit computing allocations in terms of space and time. DIRAC developers are able to leverage multi-core allocations but there is still a lack of support for multi-node allocations, common in HPC clusters. When it comes to running jobs as long as possible, DIRAC depends on a fast CPU benchmarking solution named DIRAC Benchmark. Coded in Python 2 in 2012, the utility had to be ported to Python 3 and should still provide accurate results about CPU power across many different architectures.

In Chapter 3, we have focused on the efforts to make better use of WLCG resources. In this part, we have aimed at increasing the job throughput with the same number of computing resources. This includes an analysis of the Site Director, the Pilot-Job provisioning mechanism involved in DIRAC and their interactions with distributed CEs, as well as an analysis of DIRAC Benchmark, making sure it still provides valuable results. By parallelizing operations of the Site Director, using efficiently the API to communicate with CEs and finetuning several configuration options within DIRAC, we have been able to increase the number of pilots successfully submitted per hour, from an average of 3955 to 4683 (18.41%). This represents an increase of 40.86% of the number of jobs processed simultaneously per second across the sites studied (from an average of 57,000 jobs to 80,300).

Porting DIRAC Benchmark to Python3 generated discrepancies: Python 3 results were generally higher than Python 2 scores, especially on AMD CPUs. By applying factors depending on the CPU brand and the Python version used, we have been able to provide results close to the original DB12 implementation. Comparing DB16 to the LHCb Monte-Carlo simulation jobs emphasizes further discrepancies: Python 3 results were more representative of the workload without corrections on AMD CPUs. It resulted in a new benchmark called DB21, which would allow the experiment to process 1.15 times more events on AMD CPUs. DB21 was not applied in production as LHCb still largely depends on Intel CPUs.

In Chapter 4, we have worked on the integration of Monte-Carlo simulation jobs on supercomputers. In this part, we have aimed at increasing the job throughput from resources with new types of constraints. We have designed a plan aggregating all the raised issues and developments made by LHC experiments on supercomputers and provided guidance to any HTC community that would like to start exploiting such resources. We have implemented solutions to exploit supercomputers with no external connectivity: (i) we have integrated the *PushJobAgent* component within DIRAC, which is based on the ARC Control Tower concept from ATLAS; (ii) we have created *subcvmfs-builder* and *subcvmfs-builder-pipeline* to assist any community requiring a subset of CVMFS on constrained nodes. We have also made efforts on supporting multi-core and multi-node allocations in supercomputers with external connectivity. This allows the LHCb collaboration to exploit an allocation of 750,000 CPU hours on Mare Nostrum (82^{th} of the June 2022 Top500) and opportunistic allocations on Santos Dumont (424^{th} of the June 2022 Top500), which represents around 500 jobs in parallel. While this proof of concept is still insignificant compared to the resources that we can exploit on WLCG, it lays the foundations for further and larger upcoming allocations of computing resources.

Research prospects

Revisiting the computing infrastructure choices

Choosing the infrastructures that would fit the most with LHCb workloads implies complex decisions and operations. The current strategy of the experiment is to use all the available computing resources. Nevertheless, through this manuscript, we

have been able to observe how heterogeneous the computing landscape is nowadays. Therefore, in practice, there is a trade-off between computing power theoretically available and efforts required to support and maintain applications and middleware in order to exploit this computing power. Efforts required also depend on the manpower available for the experiment.

LHCb decision-makers have favored supercomputers to extend the computing power of the experiment. Supercomputers are more and more powered with GPUs, and ARM CPUs are gaining ground. On the one hand, it represents the opportunity of pushing forward to adapt certain applications and middleware to novel architectures and platforms, and makes the experiment more resilient to computing model changes. On the other hand, not all applications can be ported to such platforms, and there is still a high demand for CISC x86 CPUs. It also requires significant manpower to adapt applications and middleware to such heterogeneous and unique systems. The LHCb experiment, like many other projects requiring computing power and leveraging EuroHPC calls, needs a lot of commodity CISC x86 CPUs, which becomes rare and demanded in such infrastructures.

Also, running LHCb tasks on a few large partitions of supercomputers instead of many medium-sized distributed clusters would be easier to manage for WMS administrators that would have to deal with less distributed sites. In this case, however, any incident on infrastructure would result in a more significant loss of computing power theoretically. From what we have observed in Section 4.4, temperature and shared file system issues, as well as maintenance of the machines often happen and block the submission and the execution of the jobs on a whole partition.

When it comes to sharing computing resources, cloud computing will likely become the reference for both HTC and HPC communities in the upcoming years. Most cloud infrastructures provide flexibility and virtualization and represent an asset for businesses and academics. Commodity CPUs should remain one of the most requested resources by businesses, despite the growing demand for specialized hardware. Nevertheless, there are still many parameters to take into account. Famous public cloud providers are efficient and reliable but remain centralized and proprietary. For instance, most of them are from the USA and, therefore, are bound to the Clarifying Lawful Overseas Use of Data Act (Cloud Act) which allows the US government to access any data on the servers of a US company, wherever their location.

While there exist regulation limiting their influence, such as the General Data Protection Regulation (GDPR) in Europe, it is recommended to encrypt sensitive data on such infrastructures.

Concomitantly to EuroHPC, Europe is also building a federated cloud named European Open Science Cloud (EOSC) to host and process research data in order to support European science. It provides European researchers, innovators, companies and citizens with an open multi-disciplinary environment where they can publish, find and re-use data, tools and services for research, innovation and educational purposes ("EOSC Portal - A gateway to information and resources in EOSC", 2022). A federated cloud aims at connecting cloud environments from various cloud providers using a common standard. According to Honorato et al., EGI was one of the main actors and the coordinator of EOSC-Hub, a three-year project, aimed at starting the design and implementation of EOSC (Honorato et al., 2021). In this way, EGI contributes to EOSC by offering services such as HTC and cloud computing resources, workload managers and security mechanisms based on tokens (Check-in). EGI developers have started to work on cloud resources in 2014. They have designed and implemented EGI Federated Cloud, a standards-based open cloud system that offers a scalable and flexible e-infrastructure to the European research community. EGI Federated Cloud extends the EGI computational offer beyond the traditional High Throughput Computing of the grid platform with new models like long-lived services and on-demand computation (Fernández-del-Castillo et al., 2015). LHCb would have constant needs for computing power and would not need flexibility features, but cloud resources might remain interesting to test novel applications on specific platforms not available at CERN for instance.

Global computing remains a great source of computing power in 2022. CERN made a BOINC server available, common to all LHC experiments (Barranco et al., 2017). In July 2022, the ATLAS experiment ran 23,000 jobs simultaneously using BOINC: about 10,000 on volunteer PCs and the rest on volunteer grid sites. LHC developers intended to develop mechanisms to leverage BOINC resources and were concerned about security issues related to the deployment of X.509 proxies on trustless computing resources. To cope with security issues, ATLAS has deployed an aCT instance to only deploy tasks and their inputs to the volunteer PCs (Adam-Bourdarios et al., 2015). Interactions with services are done on the aCT instance.

LHCbDIRAC developers designed a secure gateway service running on a trusted machine having a valid certificate and accepting a dummy Grid certificate signed by a dummy certification authority (CA) (Barranco et al., 2017). The service receives all calls coming from the jobs and transfers them to the DIRAC services. Before the real storage upload is performed, the output data produced by the volunteer machines are uploaded on the gateway machine where a check has to be performed to avoid storing wrong data on LHCb storage resources. The solution was declared difficult to support and maintain and was abandoned from the production environment. The BelleII experiment, which is also relying on DIRAC, implemented an aCT-like solution, which is close to the *PushJobAgent* solution (Wu et al., 2017) but was not included in the generic DIRAC code base. LHCb could benefit from the development of the *PushJobAgent* to investigate the use of BOINC computing resources. Though, the solution would need to be improved to avoid overloading the host. It would mostly work with Gauss tasks that do not require input files. In this case, we would also need to implement mechanisms to handle preemptible resources in DIRAC.

In the end, the LHCb experiment has to follow the trend and invest efforts in the integration of its applications to HPC resources. Because of limited manpower, the decision-makers have to carefully choose where the focus should be.

Supporting further computing resources and use cases

The plan presented in Section 4.3.1 contains problems related to the integration of CPU-intensive tasks with limited input to supercomputers and generic ideas to solve them. While there exist many project-specific implementations to solve these problems, they are not all adapted to DIRAC and, furthermore, to the LHCb context. In the context of this thesis, we have not been dealing with supercomputers with partial outbound connectivity. Therefore, we did not develop a gateway service to transfer calls from Pilot-Jobs to external services, or wrappers to use CVMFS as an unprivileged user.

Conversely, we developed solutions that would need to be expanded and further tested. For instance, we implemented a mechanism to support multi-node allocations. While we had the opportunity of testing it in SDumont, there was limited interest as the supercomputer was already overloaded. As explained in Section 4.4.1, improving

the memory consumption of the *PushJobAgent* approach and expanding it to allocate multi-core/node allocations would likely be significant for the future.

Furthermore, the plan we designed only gathers information related to practical use cases, and thus, is incomplete. LHC experiments are just starting to exploit supercomputers. Because these infrastructures are distinct from each other, there is probably still room for further minor problems and solutions.

DIRAC developers are also working on different approaches to integrate cloud resources. They recently implemented `CloudComputingElement`, a communication interface based on *Apache Libcloud* to interact with various cloud providers. Solutions related to BOINC were implemented years ago and would need to be updated or built from *PushJobAgent*. In the same way, DIRAC developers would have to support preemptible computing resources, which would require non-trivial changes to the code.

Towards an efficient exploitation of already supported computing resources

Remaining work in already supported computing resources is more about maintaining the DIRAC WMS than optimizing its services, at least in the short term. As we have seen in Section 3.2.5, there is still room for improvements in the Site Directors. We could adapt the submission rate according to the number of jobs processed by the Pilots and automatically fine-tune the load on the Sites.

Otherwise, it is a matter of maintaining DIRAC Benchmark so that it still provides accurate CPU power estimations. We could set up a more reliable method to compute the number of seconds to process an event of a given production. For instance, by relying on many different CPU architectures. This might prevent failures observed in SDumont in Section 4.4.2. In the long-term, we would probably need to use or create a new fast CPU benchmarking solution based on HEP-Benchmarks suite seen in Section 2.4.2.

Finally, transitioning from X.509 certificates to tokens also becomes an urgent matter. In Section 2.2.3, we have explained that middleware developers were planning to drop support of the GSI of the Globus Toolkit by 2023. We would need to implement

WMS components to follow the changes or we will be unable to distribute tasks to a large number of sites.

Bibliography

[warning] intel skylake/kaby lake processors: broken hyper-threading. (2017). Retrieved August 24, 2022, from https://lists.debian.org/debian-devel/2017/06/msg00308.html

80486 - intel. (2022). Retrieved September 19, 2022, from https://en.wikichip.org/wiki/intel/80486/

Abraham, S., Paul, A. K., Khan, R. I. S., & Butt, A. R. (2020). On the use of containers in high performance computing environments. *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 284–293. https://doi.org/10.1109/CLOUD49709.2020.00048

Adam-Bourdarios, C., Cameron, D., Filipčič, A., Lancon, E., Wu, W., Collaboration, A., et al. (2015). Atlas@ home: harnessing volunteer computing for hep. *Journal of Physics: Conference Series*, *664*(2), 022009.

Agostinelli, S., Allison, J., Amako, K., Apostolakis, J., Araujo, H., Arce, P., Asai, M., Axen, D., Banerjee, S., Barrand, G., Behner, F., Bellagamba, L., Boudreau, J., Broglia, L., Brunengo, A., Burkhardt, H., Chauvie, S., Chuma, J., Chytracek, R., ... Zschiesche, D. (2003). Geant4—a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, *506*(3), 250–303. https://doi.org/10.1016/S0168-9002(03)01368-8

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: principles, techniques, & tools*. Pearson Education India.

Aiftimiei, C., Andreetto, P., Bertocco, S., Fina, S., Ronco, S., Dorigo, A., Gianelle, A., Marzolla, M., Mazzucato, M., Sgaravatto, M., et al. (2008). Job submission and management through web services: the experience with the cream service. *Journal of Physics: Conference Series*, *119*(6), 062004.

Alef, M., Cass, T., Keijser, J., McNab, A., Roiser, S., Schwickerath, U., & Sfiligoi, I. (2016). Machine/job features. *HEP Software Foundation note HSF-TN-2016-02*.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*, 483–485.

Andersen, B., & Pettersen, P.-G. (1995). *Benchmarking handbook*. Springer Science & Business Media.

Anderson, D. (2004). Boinc: a system for public-resource computing and storage. *Fifth IEEE/ACM International Workshop on Grid Computing*, 4–10. https://doi.org/10.1109/GRID.2004.14

Andreetto, P., Andreozzi, S., Avellino, G., Beco, S., Cavallini, A., Cecchi, M., Ciaschini, V., Dorise, A., Giacomini, F., Gianelle, A., Grandinetti, U., Guarise, A., Krop, A., Lops, R., Maraschini, A., Martelli, V., Marzolla, M., Mezzadri, M., Molinari, E., ... Zangrando, L. (2008). The gLite workload management system. *Journal of Physics: Conference Series*, *119*(6), 062007. https://doi.org/10.1088/1742-6596/119/6/062007

Apache libcloud. (2022). Retrieved August 24, 2022, from https://libcloud.apache.org/

Apollinari, G., Béjar Alonso, I., Brüning, O., Lamont, M., & Rossi, L. (2015). *High-luminosity large hadron collider (hl-lhc) : preliminary design report*. CERN. https://doi.org/10.5170/CERN-2015-005

Arc job supervisor. (2014). Retrieved August 24, 2022, from http://www.nordugrid.org/documents/code/sdk/classArc_1_1JobSupervisor.html

Archive for santos dumont supercomputer. (2019). Retrieved August 24, 2022, from https://insidehpc.com/tag/santos-dumont-supercomputer/

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, *53*(4), 50–58.

Arrabito, L., Barbier, C., Diaz, R. G., Khélifi, B., Komin, N., Lamanna, G., Lavalley, C., Flour, T. L., Lenain, J., Lorca, A., Renaud, M., Sterzel, M., Szepieniec, T., Vasileiadis, G., & Vuerli, C. (2012). Application of the DIRAC framework to CTA: first evaluation. *Journal of Physics: Conference Series*, *396*(3), 032007. https://doi.org/10.1088/1742-6596/396/3/032007

Arsuaga-Rios, M., Heikkilä, S. S., Duellmann, D., Meusel, R., Blomer, J., & Couturier, B. (2015). Using s3 cloud storage with ROOT and CvmFS. *Journal of Physics: Conference Series*, *664*(2), 022001. https://doi.org/10.1088/1742-6596/664/2/022001

Astley, M., Sturman, D. C., & Agha, G. A. (2001). Customizable middleware for modular distributed software. *Commun. ACM, 44*(5), 99–107. https://doi.org/10.1145/374308.374365

Bagnasco, S., Betev, L., Buncic, P., Carminati, F., Cirstoiu, C., Grigoras, C., Hayrapetyan, A., Harutyunyan, A., Peters, A. J., & Saiz, P. (2008). Alien: alice environment on the grid. *Journal of Physics: Conference Series, 119*(6), 062012. https://doi.org/10.1088/1742-6596/119/6/062012

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature, 533*(7604).

Bakken, D. (2001). Middleware. *Encyclopedia of Distributed Computing, 11*.

Balcas, J., Belforte, S., Bockelman, B., Colling, D., Gutsche, O., Hufnagel, D., Khan, F., Larson, K., Letts, J., Mascheroni, M., Mason, D., McCrea, A., Piperov, S., Saiz-Santos, M., Sfiligoi, I., Tanasijczuk, A., & Wissing, C. (2015). Using the glideinWMS system as a common resource provisioning layer in CMS. *Journal of Physics: Conference Series, 664*(6), 062031. https://doi.org/10.1088/1742-6596/664/6/062031

Barranco, J., Cai, Y., Cameron, D., Crouch, M., Maria, R. D., Field, L., Giovannozzi, M., Hermes, P., Høimyr, N., Kaltchev, D., Karastathis, N., Luzzi, C., Maclean, E., McIntosh, E., Mereghetti, A., Molson, J., Nosochkov, Y., Pieloni, T., Reid, I. D., … Zacharov, I. (2017). Lhc@home: a boinc-based volunteer computing infrastructure for physics studies at cern. *Open Engineering, 7*(1), 379–393. https://doi.org/doi:10.1515/eng-2017-0042

Barrand, G., Belyaev, I., Binko, P., Cattaneo, M., Chytracek, R., Corti, G., Frank, M., Gracia, G., Harvey, J., Herwijnen, E., Maley, P., Mato, P., Probst, S., & Ranjard, F. (2001). Gaudi — a software architecture and framework for building hep data processing applications [CHEP2000]. *Computer Physics Communications, 140*(1), 45–55. https://doi.org/https://doi.org/10.1016/S0010-4655(01)00254-5

Barreiro, F., Benjamin, D., Childers, T., De, K., Elmsheuser, J., Filipčič, A., Klimentov, A., Lassnig, M., Maeno, T., Oleynik, D., & et al. (2019). The future of distributed computing systems in atlas: boldly venturing beyond grids. *EPJ Web of Conferences, 214*, 03047. https://doi.org/10.1051/epjconf/201921403047

Barreiro Megino, F. H., Alekseev, A., Berghaus, F., Cameron, D., De, K., Filipcic, A., Glushkov, I., Lin, F., Maeno, T., & Magini, N. (2020). *Managing the ATLAS Grid through Harvester* (tech. rep. ATL-SOFT-PROC-2020-006). CERN. Geneva. https://cds.cern.ch/record/2709178

Bell, G. (2008). Bell's law for the birth and death of computer classes: a theory of the computer's evolution. *IEEE Solid-State Circuits Society Newsletter*, *13*(4), 8–19. https://doi.org/10.1109/N-SSC.2008.4785818

Bell, G., & Gray, J. (2002). What's next in high-performance computing? *Commun. ACM*, *45*(2), 91–95. https://doi.org/10.1145/503124.503129

Belyaev, I., Brambach, T., Brook, N. H., Gauvin, N., Corti, G., Harrison, K., Harrison, P. F., He, J., Jones, C. R., Lieng, M., Manca, G., Miglioranzi, S., Robbe, P., Vagnoni, V., Whitehead, M., & and, J. W. (2011). Handling of the generation of primary events in gauss, the LHCb simulation framework. *Journal of Physics: Conference Series*, *331*(3), 032047. https://doi.org/10.1088/1742-6596/331/3/032047

Benjamin, Douglas, Childers, Taylor, Lesny, David, Oleynik, Danila, Panitkin, Sergey, Tsulaia, Vakho, Yang, Wei, & Zhao, Xin. (2019). Building and using containers at hpc centres for the atlas experiment. *EPJ Web Conf.*, *214*, 07005. https://doi.org/10.1051/epjconf/201921407005

Black. (2022). Retrieved August 24, 2022, from https://github.com/psf/black

Blem, E., Menon, J., & Sankaralingam, K. (2013). Power struggles: revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 1–12.

Blomer, J., Ganis, G., Hardi, N., & Popescu, R. (2017). Delivering lhc software to hpc compute elements with cernvm-fs. In J. M. Kunkel, R. Yokota, M. Taufer, & J. Shalf (Eds.), *High performance computing* (pp. 724–730). Springer International Publishing.

Blomer, Jakob, Dykstra, Dave, Ganis, Gerardo, Mosciatti, Simone, & Priessnitz, Jan. (2020). A fully unprivileged cernvm-fs. *EPJ Web Conf.*, *245*, 07012. https://doi.org/10.1051/epjconf/202024507012

Blomer, Jakob, Ganis, Gerardo, Mosciatti, Simone, & Popescu, Radu. (2019). Towards a serverless cernvm-fs. *EPJ Web Conf.*, *214*, 09007. https://doi.org/10.1051/epjconf/201921409007

Bockelman, B., Ceccanti, A., Collier, I., Cornwall, L., Dack, T., Guenther, J., Lassnig, M., Litmaath, M., Millar, P., Sallé, M., et al. (2020). Wlcg authorisation from x. 509 to tokens. *EPJ Web of Conferences*, *245*, 03001.

Boissonneault, M., Oldeman, B. E., & Taylor, R. P. (2019). Providing a unified software environment for canada's national advanced computing centers. *Proceedings*

of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning). https://doi.org/10.1145/3332186.3332210

Boyer, A. F., Haen, C., Stagni, F., & Hill, D. R. C. (2022a). Dirac site director: improving pilot-job provisioning on grid resources. *Future Generation Computer Systems*, *133*, 23–38. https://doi.org/https://doi.org/10.1016/j.future.2022.03.002

Boyer, A. F., Haen, C., Stagni, F., & Hill, D. R. C. (2022b). Pilot-job provisioning on grid resources: collecting analysis and performance evaluation data. *Data in Brief*, *42*, 108104. https://doi.org/https://doi.org/10.1016/j.dib.2022.108104

Boyer, A. F., Haen, C., Stagni, F., & Hill, D. R. C. (2022c). A subset of the cern virtual machine file system: fast delivering of complex software stacks for supercomputing resources. In A.-L. Varbanescu, A. Bhatele, P. Luszczek, & B. Marc (Eds.), *High performance computing* (pp. 354–371). Springer International Publishing.

Boyer, A. F., Hill, D. R. C., Haen, C., & Stagni, F. (2020). Pilot-job provisioning through cream computing elements on the worldwide lhc computing grid. *34th European Simulation and Modelling Conference (ESM), 34*, 33–38.

Boyer, A. F. (2021a). [dataset] dirac site director: analysis and performance evaluation. https://doi.org/10.17632/6r388827fz.1

Boyer, A. F. (2021b). [dataset] integration of lhcb workflows on the santos dumont supercomputer. https://doi.org/10.17632/c7w3cgfzgt.1

Boyer, A. F. (2022a). [dataset] subcvmfs: gauss analysis. https://doi.org/10.5281/zenodo.6337297

Boyer, A. F. (2022b). [program] subcvmfs-builder. https://doi.org/10.5281/zenodo.6335367

Boyer, A. F. (2022c). [program] subcvmfs-builder-pipeline. https://doi.org/10.5281/zenodo.6335512

Boyer, A. F., & Stagni, F. (2021). [dataset] porting db12 to python3: analysis of the scores. https://doi.org/10.5281/zenodo.7031029

Bricker, A., Litzkow, M., & Livny, M. (1992). *Condor technical summary* (tech. rep.). University of Wisconsin-Madison Department of Computer Sciences.

Buncic, P., Sanchez, C. A., Blomer, J., Franco, L., Harutyunian, A., Mato, P., & Yao, Y. (2010). CernVM – a virtual software appliance for LHC applications. *Journal of Physics: Conference Series*, *219*(4), 042003. https://doi.org/10.1088/1742-6596/219/4/042003

Burr, Chris, Clemencic, Marco, & Couturier, Ben. (2019). Software packaging and distribution for lhcb using nix. *EPJ Web Conf., 214*, 05005. https://doi.org/10.1051/epjconf/201921405005

Buyya, R. (1999). *High performance cluster computing: architectures and systems* (Vol. 1). Citeseer.

Calafiura, P., De, K., Guan, W., Maeno, T., Nilsson, P., Oleynik, D., Panitkin, S., Tsulaia, V., Gemmeren, P. V., & Wenaus, T. (2015a). The atlas event service: a new approach to event processing. *Journal of Physics: Conference Series, 664*(6), 062065. https://doi.org/10.1088/1742-6596/664/6/062065

Calafiura, P., De, K., Guan, W., Maeno, T., Nilsson, P., Oleynik, D., Panitkin, S., Tsulaia, V., Gemmeren, P. V., & Wenaus, T. (2015b). Fine grained event processing on hpcs with the atlas yoda system. *Journal of Physics: Conference Series, 664*(9), 092025. https://doi.org/10.1088/1742-6596/664/9/092025

Cameron, D., Filipčič, A., Guan, W., Tsulaia, V., Walker, R., & and, T. W. (2017). Exploiting opportunistic resources for atlas with arc ce and the event service. *Journal of Physics: Conference Series, 898*, 052010. https://doi.org/10.1088/1742-6596/898/5/052010

Campana, S. (2015). ATLAS distributed computing in LHC run2. *Journal of Physics: Conference Series, 664*(3), 032004. https://doi.org/10.1088/1742-6596/664/3/032004

Cardoso, J., Sheth, A., Miller, J., Arnold, J., & Kochut, K. (2004). Quality of service for workflows and web service processes. *Journal of Web Semantics, 1*(3), 281–308. https://doi.org/https://doi.org/10.1016/j.websem.2004.03.001

Casajus, A., Graciani, R. et al. (2010). Dirac distributed secure framework. *Journal of Physics: Conference Series, 219*(4), 042033.

Casajus, A., Graciani, R., Paterson, S., Tsaregorodtsev, A., & Team, t. L. D. (2010). Dirac pilot framework and the dirac workload management system. *Journal of Physics: Conference Series, 219*(6), 062049. https://doi.org/10.1088/1742-6596/219/6/062049

Ceccanti, A., Vianello, E., Caberletti, M., & Giacomini, F. (2019). Beyond x. 509: token-based authentication and authorization for hep. *EPJ Web of Conferences, 214*, 09002.

Cern, skao, géant and prace to collaborate on high-performance computing. (2020). Retrieved August 24, 2022, from https://home.cern/news/news/computing/cern-skao-geant-and-prace-collaborate-high-performance-computing

Cernvm-fs. (2022). Retrieved August 24, 2022, from https://cernvm.cern.ch/

Cernvm-fs overview and roadmap. (2021). Retrieved August 24, 2022, from https://easybuild.io/eum21/002_eum21_cvmfs.pdf

Charpentier, P. (2017). Benchmarking worker nodes using LHCb productions and comparing with HEPSpec06. *Journal of Physics: Conference Series, 898*, 082011. https://doi.org/10.1088/1742-6596/898/8/082011

Chien, A. A., Pakin, S., Lauria, M., Buchanan, M., Hane, K., Giannini, L. A., & Prusakova, J. (1997). High performance virtual machines (hpvm's): clusters with supercomputing api's and performance. *PPSC*.

Chiu, P.-H., & Potekhin, M. (2010). Pilot factory–a condor-based system for scalable pilot job generation in the panda wms framework. *Journal of Physics: Conference Series*, *219*(6), 062041.

Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauve, J., Silva, F., Barros, C., & Silveira, C. (2003). Running bag-of-tasks applications on computational grids: the mygrid approach. *2003 International Conference on Parallel Processing, 2003. Proceedings.*, 407–416. https://doi.org/10.1109/ICPP.2003.1240605

Clemencic, M., Corti, G., Easo, S., Jones, C. R., Miglioranzi, S., Pappagallo, M., & and, P. R. (2011). The LHCb simulation application, gauss: design, evolution and experience. *Journal of Physics: Conference Series*, *331*(3), 032023. https://doi.org/10.1088/1742-6596/331/3/032023

Clemencic, M., & Couturier, B. (2014). A new nightly build system for LHCb. *Journal of Physics: Conference Series*, *513*(5), 052007. https://doi.org/10.1088/1742-6596/513/5/052007

Closier, J., Gaspar, C., Cardoso, L. G., Haen, C., Jost, B., & Neufeld, N. (2019). Simultaneous usage of the lhcb hlt farm for online and offline processing workflows. *EPJ Web of Conferences*, *214*, 01001.

Cocke, J., & Markstein, V. (1990). The evolution of risc technology at ibm. *IBM Journal of Research and Development*, *34*(1), 4–11. https://doi.org/10.1147/rd.341.0004

Collaboration, A. (2012). Observation of a new particle in the search for the standard model higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, *716*(1), 1–29. https://doi.org/10.1016/j.physletb.2012.08.020

Collaboration, L. (2019). Performance of the Lamarr Prototype: the ultra-fast simulation option integrated in the LHCb simulation framework. https://cds.cern.ch/record/2696310

Collaboration, T. L. (2008). The LHCb detector at the LHC. *Journal of Instrumentation*, *3*(08), S08005–S08005. https://doi.org/10.1088/1748-0221/3/08/s08005

Conda. (2022). Retrieved August 24, 2022, from https://docs.conda.io/en/latest/

Conda-forge. (2022). Retrieved August 24, 2022, from https://conda-forge.org/

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., & Polk, W. (2008). *Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile* (tech. rep.).

Courtès, L., & Wurmus, R. (2015). Reproducible and user-controlled software environments in hpc with guix. In S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, & M. Alexander (Eds.), *Euro-par 2015: parallel processing workshops* (pp. 579–591). Springer International Publishing.

Couturier, B., Kiagias, E., & Lohn, S. B. (2014). Systematic profiling to monitor and specify the software refactoring process of the LHCb experiment. *Journal of Physics: Conference Series*, *513*(5), 052020. https://doi.org/10.1088/1742-6596/513/5/052020

Criss, K., Bains, K., Agarwal, R., Bennett, T., Grunzke, T., Kim, J. K., Chung, H., & Jang, M. (2020). Improving memory reliability by bounding dram faults: ddr5 improved reliability features. *The International Symposium on Memory Systems*, 317–322. https://doi.org/10.1145/3422575.3422803

Culler, D. E., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C., & Wong, F. (1997). Parallel computing on the berkeley now. *9th Joint Symposium on Parallel Processing*.

Cvmfsexec. (2022). Retrieved August 24, 2022, from https://github.com/cvmfs/cvmfsexec

Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W., & Tuecke, S. (1998). A resource management architecture for metacomputing systems. *Workshop on Job Scheduling Strategies for Parallel Processing*, 62–82.

Dao, V., Maigne, L., Breton, V., Nguyen, H., & Hill, D. R. (2014). Numerical Reproducibility, Portability And Performance Of Modern Pseudo Random Number Generators : Preliminary study for parallel stochastic simulations using hybrid Xeon Phi computing processors. *European Simulation And Modelling Conference*, 80–87. https://hal.archives-ouvertes.fr/hal-02077185

De, K., Golubkov, D., Klimentov, A., Potekhin, M., Vaniachine, A., Collaboration, A., et al. (2014). Task management in the new atlas production system. *Journal of Physics: Conference Series*, *513*(3), 032078.

De, K., Klimentov, A., Oleynik, D., Panitkin, S., Petrosyan, A., Schovancova, J., Vaniachine, A., Wenaus, T., Collaboration, A., et al. (2015). Integration of panda workload management system with titan supercomputer at olcf. *Journal of Physics: Conference Series*, *664*(9), 092020.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., & Wenger, K. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, *46*, 17–35. https://doi.org/https://doi.org/10.1016/j.future.2014.10.008

Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., & LeBlanc, A. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, *9*(5), 256–268. https://doi.org/10.1109/JSSC.1974.1050511

Dillon, T., Wu, C., & Chang, E. (2010). Cloud computing: issues and challenges. *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 27–33. https://doi.org/10.1109/AINA.2010.187

Dirac. (2022). Retrieved August 24, 2022, from https://github.com/DIRACGrid/DIRAC

Dirac benchmark 12. (2022). Retrieved August 24, 2022, from https://github.com/DIRACGrid/DB12

Dirac egi. (2022). Retrieved August 24, 2022, from https://dirac.egi.eu/DIRAC/

Dixit, K. M. (1993). Overview of the spec benchmarks.

Docker. (2022). Retrieved August 24, 2022, from https://www.docker.com/

Dongarra, J. J., Luszczek, P., & Petitet, A. (2003). The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, *15*(9), 803–820.

Dongarra, J. J., Moler, C. B., Bunch, J. R., & Stewart, G. W. (1979). *Linpack users' guide*. SIAM.

Dotti, A., Asai, M., Barrand, G., Hrivnacova, I., & Murakami, K. (2015). Extending geant4 parallelism with external libraries (mpi, tbb) and its use on hpc resources. *2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 1–2.

Dröge, B., Holanda Rusu, V., Hoste, K., van Leeuwen, C., O'Cais, A., & Röblitz, T. (n.d.). Eessi: a cross-platform ready-to-use optimised scientific software stack. *Software: Practice and Experience*, *n/a*(n/a). https://doi.org/https://doi.org/10.1002/spe.3075

Dykstra, D., & Blomer, J. (2014). Security in the CernVM File System and the Frontier Distributed Database Caching System. *Journal of Physics Conference Series*, *513*, Article 042015, 042015. https://doi.org/10.1088/1742-6596/513/4/042015

Easybuild: building software with ease. (2022). Retrieved August 24, 2022, from https://easybuild.io/

Ellert, M., Grønager, M., Konstantinov, A., Kónya, B., Lindemann, J., Livenson, I., Nielsen, J. L., Niinimäki, M., Smirnova, O., & Wäänänen, A. (2007). Advanced resource connector middleware for lightweight computational grids. *Future Generation Computer Systems*, *23*(2), 219–240. https://doi.org/10.1016/j.future.2006.05.008

Eosc portal - a gateway to information and resources in eosc. (2022). Retrieved August 24, 2022, from https://eosc-portal.eu/

Evolution of computer networks. (2016). Retrieved August 24, 2022, from https://www3.nd.edu/~dwang5/courses/fall16/pdf/evolution.pdf

Fan, Z., Qiu, F., Kaufman, A., & Yoakum-Stover, S. (2004). Gpu cluster for high performance computing. *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 47–47. https://doi.org/10.1109/SC.2004.26

Fasel, M. (2016). Using nersc high-performance computing (hpc) systems for high-energy nuclear physics applications with alice. *Journal of Physics: Conference Series*, *762*, 012031. https://doi.org/10.1088/1742-6596/762/1/012031

Fedak, G., Germain, C., Neri, V., & Cappello, F. (2001). Xtremweb: a generic global computing system. *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 582–587.

Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., & Wong, P. (1997). Theory and practice in parallel job scheduling. In D. G. Feitelson & L. Rudolph (Eds.), *Job scheduling strategies for parallel processing* (pp. 1–34). Springer Berlin Heidelberg.

Fernández-del-Castillo, E., Scardaci, D., & García, Á. L. (2015). The egi federated cloud e-infrastructure [1st International Conference on Cloud Forward: From Distributed to Complete Computing]. *Procedia Computer Science*, *68*, 196–205. https://doi.org/https://doi.org/10.1016/j.procs.2015.09.235

Filipčič, A. (2011). Arccontroltower: the system for atlas production and analysis on arc. *Journal of Physics: Conference Series*, *331*(7), 072013. https://doi.org/10.1088/1742-6596/331/7/072013

Filipčič, A. (2017). Integration of the chinese hpc grid in atlas distributed computing. *Journal of Physics: Conference Series*, *898*, 082008. https://doi.org/10.1088/1742-6596/898/8/082008

Filipčič, A., Haug, S., Hostettler, M., Walker, R., & Weber, M. (2015). Atlas computing on cscs hpc. *Journal of Physics: Conference Series*, *664*(9), 092011. https://doi.org/10.1088/1742-6596/664/9/092011

Flynn, M. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, *54*(12), 1901–1909. https://doi.org/10.1109/PROC.1966.5273

Foster, I., & Kesselman, C. (1997). Globus: a metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, *11*(2), 115–128.

Foster, I., & Kesselman, C. (2003). *The grid 2: blueprint for a new computing infrastructure*. Elsevier.

Foster, I., Kesselman, C., & Tuecke, S. (2001). The anatomy of the grid: enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, *15*(3), 200–222.

Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008). Cloud computing and grid computing 360-degree compared. *2008 Grid Computing Environments Workshop*, 1–10. https://doi.org/10.1109/GCE.2008.4738445

Frank, D., Dennard, R., Nowak, E., Solomon, P., Taur, Y., & Wong, H.-S. P. (2001). Device scaling limits of si mosfets and their application dependencies. *Proceedings of the IEEE*, *89*(3), 259–288. https://doi.org/10.1109/5.915374

Frey, J., Tannenbaum, T., Livny, M., Foster, I., & Tuecke, S. (2002). Condor-g: a computation management agent for multi-institutional grids. *Cluster Computing*, *5*(3), 237–246. https://doi.org/10.1023/A:1015617019423

Furlani, J. L. (1991). Modules: providing a flexible user environment. *Proceedings of the fifth large installation systems administration conference (LISA V)*, 141–152.

Géant. (2022). Retrieved August 24, 2022, from https://www.geant.org/

Gentoo linux. (2022). Retrieved August 24, 2022, from https://www.gentoo.org/

Gerhardt, L., Bhimji, W., Canon, S., Fasel, M., Jacobsen, D., Mustafa, M., Porter, J., & Tsulaia, V. (2017). Shifter: containers for hpc. *Journal of Physics: Conference Series*, *898*, 082021. https://doi.org/10.1088/1742-6596/898/8/082021

Giordano, D., & Santorinaiou, E. (2020). Next generation of HEP CPU benchmarks. *1525*, 012073. https://doi.org/10.1088/1742-6596/1525/1/012073

Github actions. (2022). Retrieved August 24, 2022, from https://docs.github.com/en/actions

Gitlab ci/cd. (2022). Retrieved August 24, 2022, from https://docs.gitlab.com/ee/ci/

Gitler, I., Gomes, A. T. A., & Nesmachnow, S. (2020). The latin american supercomputing ecosystem for science. *Commun. ACM*, *63*(11), 66–71. https://doi.org/10.1145/3419977

Global interpreter lock. (2022). Retrieved August 24, 2022, from https://wiki.python.org/moin/GlobalInterpreterLock

Gramoli, V. (2017). The information needed for reproducing shared memory experiments. In F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, & J. Weidendorfer (Eds.), *Euro-par 2016: parallel processing workshops* (pp. 596–608). Springer International Publishing.

Gray, J. (2003). *Distributed computing economics* (tech. rep. MSR-TR-2003-24). https://www.microsoft.com/en-us/research/publication/distributed-computing-economics/

Guix. (2022). Retrieved August 24, 2022, from https://guix.gnu.org/

Hardt, D. (2012). *The oauth 2.0 authorization framework* (tech. rep.).

Harutyunyan, A., Blomer, J., Buncic, P., Charalampidis, I., Grey, F., Karneyeu, A., Larsen, D., González, D. L., Lisec, J., Segal, B., & Skands, P. (2012). CernVM co-pilot: an extensible framework for building scalable computing infrastructures on the cloud. *Journal of Physics: Conference Series*, *396*(3), 032054. https://doi.org/10.1088/1742-6596/396/3/032054

Hategan, M., Wozniak, J., & Maheshwari, K. (2011). Coasters: uniform resource provisioning and access for clouds and grids. *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, 114–121. https://doi.org/10.1109/UCC.2011.25

Henderson, R. L. (1995). Job scheduling under the portable batch system. In D. G. Feitelson & L. Rudolph (Eds.), *Job scheduling strategies for parallel processing* (pp. 279–294). Springer Berlin Heidelberg.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, *34*(4), 1–17. https://doi.org/10.1145/1186736.1186737

Hepix online. (2022). Retrieved August 24, 2022, from https://www.hepix.org/

Hickmann, B., Chen, J., Rotzin, M., Yang, A., Urbanski, M., & Avancha, S. (2020). Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product. *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, 133–136. https://doi.org/10.1109/ARITH48897.2020.00029

Hill, D. R. (2015). Parallel random numbers, simulation, and reproducible research. *Computing in Science & Engineering*, *17*(4), 66–71. https://doi.org/10.1109/MCSE.2015.79

Honorato, R. V., Koukos, P. I., Jiménez-García, B., Tsaregorodtsev, A., Verlato, M., Giachetti, A., Rosato, A., & Bonvin, A. M. J. J. (2021). Structural biology in the clouds: the wenmr-eosc ecosystem. *Frontiers in Molecular Biosciences*, *8*. https://doi.org/10.3389/fmolb.2021.729513

How to run hep-spec06 benchmark. (2017). Retrieved August 24, 2022, from https://w3.hepix.org/benchmarking/how_to_run_hs06.html

Huerta, E. A., Haas, R., Jha, S., Neubauer, M., & Katz, D. S. (2019). Supporting High-Performance and High-Throughput computing for experimental science. *Computing and Software for Big Science*, *3*(1), 5.

Hufnagel, D. (2017). Cms use of allocation based hpc resources. *Journal of Physics: Conference Series*, *898*, 092050. https://doi.org/10.1088/1742-6596/898/9/092050

Hushchyn, M., Ustyuzhanin, A., Arzymatov, K., Roiser, S., & Baranov, A. (2017). The lhcb grid simulation: proof of concept. *Journal of Physics: Conference Series*, *898*, 052020. https://doi.org/10.1088/1742-6596/898/5/052020

Ibm platform lsf. (2022). Retrieved August 24, 2022, from https://www.ibm.com/products/hpc-workload-management

Jenviriyakul, P., Chalumporn, G., Achalakul, T., Costa, F., & Akkarajitsakul, K. (2019). Alice connex: a volunteer computing platform for the time-of-flight calibration of the alice experiment. an opportunistic use of cpu cycles on android devices. *Future Generation Computer Systems*, *94*, 510–523. https://doi.org/https://doi.org/10.1016/j.future.2018.11.057

Jones, C., Collaboration, C. et al. (2017). Cms event processing multi-core efficiency status. *Journal of Physics: Conference Series*, *898*(4), 042008.

Jones, M., Bradley, J., & Sakimura, N. (2015). *Json web token (jwt)* (tech. rep.).

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., … Yoon, D. H. (2017). In-datacenter

performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, *45*(2), 1–12. https://doi.org/10.1145/3140659.3080246

Khetawat, H., Zimmer, C., Mueller, F., Atchley, S., Vazhkudai, S. S., & Mubarak, M. (2019). Evaluating burst buffer placement in hpc systems. *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 1–11.

Konstantinov, A. (2017). Arc computational job management component–a-rex.

Korenkov, V., Pelevanyuk, I., & Tsaregorodtsev, A. (2020). Integration of the jinr hybrid computing resources with the dirac interware for data intensive applications. In A. Elizarov, B. Novikov, & S. Stupnikov (Eds.), *Data analytics and management in data intensive domains* (pp. 31–46). Springer International Publishing.

Korpela, E., Werthimer, D., Anderson, D., Cobb, J., & Leboisky, M. (2001). Seti@home-massively distributed computing for seti. *Computing in Science Engineering*, *3*(1), 78–83. https://doi.org/10.1109/5992.895191

Kruzelecki, K., Roiser, S., & Degaudenzi, H. (2010). The nightly build and test system for LCG AA and LHCb software. *Journal of Physics: Conference Series*, *219*(4), 042042. https://doi.org/10.1088/1742-6596/219/4/042042

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: scientific containers for mobility of compute. *PloS one*, *12*(5), e0177459.

Lange, D. J. (2001). The evtgen particle decay simulation package. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, *462*(1-2), 152–155.

Lattner, C. (2008). Llvm and clang: next generation compiler technology. *The BSD conference*, *5*.

Laure, E., Edlund, A., Pacini, F., Buncic, P., Beco, S., Prelz, F., Di Meglio, A., Mulmo, O., Barroso, M., Kunszt, P. Z., et al. (2004). *Middleware for the next generation grid infrastructure* (tech. rep.). CERN.

L'ecuyer, P., & Simard, R. (2007). Testu01: ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, *33*(4), 1–40.

Lee, C. A., Bohn, R. B., Michel, M., Delaitre, A., Stivalet, B., & Black, P. E. (2020). The nist cloud federation reference architecture 5. *NIST Special Publication*, *500*, 332.

Leggett, C., Baines, J., Bold, T., Calafiura, P., Farrell, S., van Gemmeren, P., Malon, D., Ritsch, E., Stewart, G., Snyder, S., Tsulaia, V., & and, B. W. (2017). AthenaMT: upgrading the ATLAS software framework for the many-core world with multi-

threading. *Journal of Physics: Conference Series, 898*, 042009. https://doi.org/10.1088/1742-6596/898/4/042009

Lhc season 2 facts & figures. (2018). Retrieved August 24, 2022, from https://home.cern/sites/home.web.cern.ch/files/2018-07/factsandfigures-en_0.pdf

Lhcb - large hadron collider beauty experiment. (2022). Retrieved August 24, 2022, from https://lhcb-outreach.web.cern.ch/

Linux@cern. (2022). Retrieved August 24, 2022, from https://linux.web.cern.ch/

Litzkow, M. J., Livny, M., & Mutka, M. W. (1987). *Condor-a hunter of idle workstations* (tech. rep.). University of Wisconsin-Madison Department of Computer Sciences.

Livny, M., Basney, J., Raman, R., Tannenbaum, T., et al. (1997). Mechanisms for high throughput computing. *SPEEDUP journal*, *11*(1), 36–40.

Luckow, A., Santcroos, M., Weidner, O., Merzky, A., Maddineni, S., & Jha, S. (2012). Towards a common model for pilot-jobs. *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, 123–124. https://doi.org/10.1145/2287076.2287094

Luckow, A., Lacinski, L., & Jha, S. (2010). Saga bigjob: an extensible and interoperable pilot-job abstraction for distributed applications and systems. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 135–144. https://doi.org/10.1109/CCGRID.2010.91

Maeno, T., Megino, F. H. B., Benjamin, D., Cameron, D., Childers, J. T., De, K., Salvo, A. D., Filipčič, A., Hover, J., Lin, F., & et al. (2019). Harvester: an edge service harvesting heterogeneous resources for atlas. *EPJ Web of Conferences, 214*, 03030. https://doi.org/10.1051/epjconf/201921403030

Mainframe concepts. (2010). Retrieved August 24, 2022, from https://www.ibm.com/docs/en/zos-basic-skills?topic=vmt-who-uses-mainframes-why-do-they-do-it

Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., & Upton, M. (2002). Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, *6*(1).

Marsaglia, G., Zaman, A., & Wan Tsang, W. (1990). Toward a universal random number generator. *Statistics & Probability Letters*, *9*(1), 35–39. https://doi.org/https://doi.org/10.1016/0167-7152(90)90092-L

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., & Ghalsasi, A. (2011). Cloud computing—the business perspective. *Decision support systems*, *51*(1), 176–189.

Mazurov, A., Couturier, B., Popov, D., & Farley, N. (2017). Microservices for systematic profiling and monitoring of the refactoring process at the LHCb experiment. *Journal of Physics: Conference Series*, *898*, 072037. https://doi.org/10.1088/1742-6596/898/7/072037

McIntosh, E. (1992). Benchmarking computers for HEP. https://doi.org/10.5170/CERN-1993-003.186

McNab, A., Stagni, F., & Garcia, M. U. (2014). Running jobs in the vacuum. *Journal of Physics: Conference Series*, *513*(3), 032065. https://doi.org/10.1088/1742-6596/513/3/032065

Meek, C. L. (2003). Mainframe. *Encyclopedia of computer science* (p. 1068). John Wiley; Sons Ltd.

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

Meyer, R. A., & Seawright, L. H. (1970). A virtual machine time-sharing system. *IBM Systems Journal*, *9*(3), 199–218. https://doi.org/10.1147/sj.93.0199

Michelotto, M., Alef, M., Iribarren, A., Meinhard, H., Wegner, P., Bly, M., Benelli, G., Brasolin, F., Degaudenzi, H., Salvo, A. D., Gable, I., Hirstius, A., & Hristov, P. (2010). A comparison of HEP code with SPEC1benchmarks on multi-core worker nodes. *219*(5), 052009. https://doi.org/10.1088/1742-6596/219/5/052009

Miyake, H., Grzymkowski, R., Ludacka, R., & Schram, M. (2015). Belle ii production system. *Journal of Physics: Conference Series*, *664*, 052028. https://doi.org/10.1088/1742-6596/664/5/052028

Mnich, J. (2022). *Agenda 48th meeting of the lhcb rrb* (tech. rep.).

Monmasson, E., & Cirstea, M. N. (2007). Fpga design methodology for industrial control systems—a review. *IEEE Transactions on Industrial Electronics*, *54*(4), 1824–1842. https://doi.org/10.1109/TIE.2007.898281

Moore, G. E. et al. (1965). Cramming more components onto integrated circuits.

Müller, D., Clemencic, M., Corti, G., & Gersabeck, M. (2018). Redecay: a novel approach to speed up the simulation at lhcb. *The European Physical Journal C*, *78*(12), 1–5.

Nakamoto, S. (2008). Bitcoin: a peer-to-peer electronic cash system. *Decentralized Business Review*, 21260.

Napper, J., & Bientinesi, P. (2009). Can cloud computing reach the top500? *Proceedings of the Combined Workshops on UnConventional High Performance Computing*

Workshop plus Memory Access Workshop, 17–20. https://doi.org/10.1145/1531666.1531671

Nelson, M. R. (2009). Building an open cloud. *Science*, *324*(5935), 1656–1657. https://doi.org/10.1126/science.1174225

Nilsen, J., Cameron, D., & Filipčič, A. (2015). Arc control tower: a flexible generic distributed job management framework. *Journal of Physics: Conference Series*, *664*(6), 062042. https://doi.org/10.1088/1742-6596/664/6/062042

Nixos. (2022). Retrieved August 24, 2022, from https://nixos.org/

O'Brien, B., Walker, R., & Washbrook, A. (2014). Leveraging hpc resources for high energy physics. *Journal of Physics: Conference Series*, *513*(3), 032104. https://doi.org/10.1088/1742-6596/513/3/032104

Oleynik, D., Panitkin, S., Turilli, M., Angius, A., Oral, S., De, K., Klimentov, A., Wells, J. C., & Jha, S. (2017). High-throughput computing on high-performance platforms: a case study. *2017 IEEE 13th International Conference on e-Science (e-Science)*, 295–304. https://doi.org/10.1109/eScience.2017.43

Osman, A., Anjum, A., Batool, N., & McClatchey, R. (2012). A fault tolerant, dynamic and low latency bdii architecture for grids [arXiv: 1202.5512]. *arXiv:1202.5512 [cs]*. http://arxiv.org/abs/1202.5512

Ovyn, S., Rouby, X., & Lemaitre, V. (2009). Delphes, a framework for fast simulation of a generic collider experiment. *arXiv preprint arXiv:0903.2225*.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, *96*(5), 879–899.

Panneton, F., & L'ecuyer, P. (2005). On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *15*(4), 346–361.

Patterson, D. A., & Ditzel, D. R. (1980). The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, *8*(6), 25–33. https://doi.org/10.1145/641914.641917

Pelevanyuk, I. (2021). Performance evaluation of computing resources with dirac interware. *AIP Conference Proceedings*, *2377*(1), 040006. https://doi.org/10.1063/5.0064778

Pfister, G. F. (2001). An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, *42*(617-632), 102.

Pilot. (2022). Retrieved August 24, 2022, from https://github.com/DIRACGrid/Pilot

Piucci, A. (2017). The LHCb upgrade. *Journal of Physics: Conference Series*, *878*, 012012. https://doi.org/10.1088/1742-6596/878/1/012012

Podman. (2022). Retrieved August 24, 2022, from https://podman.io/

Popescu, Radu, Blomer, Jakob, & Ganis, Gerardo. (2019). Towards a responsive cernvm-fs architecture. *EPJ Web Conf.*, *214*, 03036. https://doi.org/10.1051/epjconf/201921403036

Popov, Dmitry. (2019). Testing and verification of the lhcb simulation. *EPJ Web Conf.*, *214*, 02043. https://doi.org/10.1051/epjconf/201921402043

Prace. (2022). Retrieved August 24, 2022, from https://prace-ri.eu/

Pre-commit. (2022). Retrieved August 24, 2022, from https://pre-commit.com/

Priedhorsky, R., & Randles, T. (2017). Charliecloud: unprivileged containers for user-defined software stacks in hpc. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3126908.3126925

Pylint. (2022). Retrieved August 24, 2022, from https://pylint.pycqa.org/en/latest/

Pypi. (2022). Retrieved August 24, 2022, from https://pypi.org/

Pytest: helps you write better programs. (2021). Retrieved August 24, 2022, from https://docs.pytest.org/

Raicu, I. (2009). *Many-task computing: bridging the gap between high-throughput computing and high-performance computing*. The University of Chicago.

Raicu, I., Foster, I. T., & Zhao, Y. (2008). Many-task computing for grids and supercomputers. *2008 Workshop on Many-Task Computing on Grids and Supercomputers*, 1–11. https://doi.org/10.1109/MTAGS.2008.4777912

Rauschmayr, N., & Streit, A. (2014). Preparing the gaudi framework and the DIRAC WMS for multicore job submission. *Journal of Physics: Conference Series*, *513*(5), 052029. https://doi.org/10.1088/1742-6596/513/5/052029

Reed, D., Gannon, D., & Dongarra, J. (2022). Reinventing high performance computing: challenges and opportunities. https://doi.org/10.48550/ARXIV.2203.02544

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., & Kepner, J. (2021). Ai accelerator survey and trends. *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–9. https://doi.org/10.1109/HPEC49654.2021.9622867

Rho, S., Kim, S., Kim, S., Kim, S., Kim, J.-S., & Hwang, S. (2012). Abstract: htcaas: a large-scale high-throughput computing by leveraging grids, supercomputers and cloud. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 1341–1342. https://doi.org/10.1109/SC.Companion.2012.175

Romero Coronado, J. P., & Altmann, J. (2017). Model for incentivizing cloud service federation. *International Conference on the Economics of Grids, Clouds, Systems, and Services*, 233–246.

Rubio-Montero, A. J., Huedo, E., Castejón, F., & Mayo-García, R. (2015). Gwpilot: enabling multi-level scheduling in distributed infrastructures with gridway and pilot jobs. *Future Generation Computer Systems*, *45*, 25–52. https://doi.org/10.1016/j.future.2014.10.003

Sakimura, N., Bradley, J., Jones, M., De Medeiros, B., & Mortimore, C. (2014). Openid connect core 1.0. *The OpenID Foundation*, S3.

Sanders, J., & Kandrot, E. (2010). *Cuda by example: an introduction to general-purpose gpu programming*. Addison-Wesley Professional.

Schweitzer, P., Mazel, C., Carloganu, C., & Hill, D. R. (2014). A method for porting HEP software Geant4 and ROOT to Intel Xeon Phi hardware accelerator. *European Simulation and Modelling Conference*. https://hal.archives-ouvertes.fr/hal-02077184

Schweitzer, P., Mazel, C., Hill, D. R., & Cârloganu, C. (2015). Performance analysis with a memory-bound monte carlo simulation on xeon phi. *2015 International Conference on High Performance Computing & Simulation (HPCS)*, 444–452. https://doi.org/10.1109/HPCSim.2015.7237074

Sciacca, F. G. (2020). Enabling atlas big data processing on piz daint at cscs. *EPJ Web Conf.*, *245*, 09005. https://doi.org/10.1051/epjconf/202024509005

Sdumont. (2021). Retrieved August 24, 2022, from https://sdumont.lncc.br/

Segal, B., Buncic, P., Sanchez, C. A., Blomer, J., Quintas, D. G., Harutyunian, A., Mato, P., Rantala, J., Weir, D., & Yao, Y. (2011). Lhc cloud computing with cernvm. *13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT2010)*, *093*(4), 042003. https://doi.org/10.22323/1.093.0004

Sfiligoi, I., & Padhi, S. (2010). Evaluation of new compute element software for the open science grid: gram5 and cream. *OSG Document*, *1006*.

Sfiligoi, I. (2008). Glideinwms—a generic pilot-based workload management system. *Journal of Physics: Conference Series*, *119*(6), 062044. https://doi.org/10.1088/1742-6596/119/6/062044

Shifter - containers for hpc. (2022). Retrieved August 24, 2022, from https://github.com/NERSC/shifter

Siddi, B. G. (2016). Development and deployment of a fully parameterized fast monte carlo simulation in lhcb. *2016 IEEE Nuclear Science Symposium, Medical Imaging Conference and Room-Temperature Semiconductor Detector Workshop (NSS/MIC/RTSD)*, 1–4. https://doi.org/10.1109/NSSMIC.2016.8069894

Siddi, B. G., & Müller, D. (2019). Gaussino - a gaudi-based core simulation framework. *2019 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 1–4. https://doi.org/10.1109/NSS/MIC42101.2019.9060074

Sjöstrand, T., Edén, P., Friberg, C., Lönnblad, L., Miu, G., Mrenna, S., & Norrbin, E. (2001). High-energy-physics event generation with pythia 6.1. *Computer Physics Communications*, *135*(2), 238–259. https://doi.org/10.1016/s0010-4655(00)00236-8

Smith, J. E., & Nair, R. (2001). An overview of virtual machine architectures. *Oct*, *26*, 1–20.

Spack. (2022). Retrieved August 24, 2022, from https://spack.readthedocs.io/en/latest/

Stagni, F., & Charpentier, P. (2012). The LHCb DIRAC-based production and data management operations systems. *Journal of Physics: Conference Series*, *368*, 012010. https://doi.org/10.1088/1742-6596/368/1/012010

Stagni, F., & Charpentier, P. (2015). Jobs masonry in lhcb with elastic grid jobs. *Journal of Physics: Conference Series*, *664*, 062060. https://doi.org/10.1088/1742-6596/664/6/062060

Stagni, F., McNab, A., Luzzi, C., Krzemien, W., & Consortium, D. (2017). Dirac universal pilots. *Journal of Physics: Conference Series*, *898*(9), 092024. https://doi.org/10.1088/1742-6596/898/9/092024

Stagni, F., Tsaregorodtsev, A., McNab, A., & Luzzi, C. (2015). Pilots 2.0: dirac pilots for all the skies. *Journal of Physics: Conference Series*, *664*(6), 062061. https://doi.org/10.1088/1742-6596/664/6/062061

Stagni, F., Valassi, A., & Romanovskiy, V. (2020). Integrating lhcb workflows on hpc resources: status and strategies (C. Doglioni, D. Kim, G. Stewart, L. Silvestris, P. Jackson, & W. Kamleh, Eds.). *EPJ Web of Conferences*, *245*, 09002. https://doi.org/10.1051/epjconf/202024509002

Standard performance evaluation corporation. (2022). Retrieved August 24, 2022, from https://spec.org/

Sterling, T. (2003). Cluster computing. In R. A. Meyers (Ed.), *Encyclopedia of physical science and technology (third edition)* (Third Edition, pp. 33–43). Academic Press. https://doi.org/https://doi.org/10.1016/B0-12-227410-5/00109-5

Sterling, T. L. (2002). *Beowulf cluster computing with linux*. MIT press.

Svatos, M., Chudoba, J., & Vokác, P. (2020a). *ATLAS job submission system for Salomon HPC based on ARC-CE* (tech. rep. ATL-SOFT-PROC-2020-001). CERN. Geneva. https://cds.cern.ch/record/2707263

Svatos, M., Chudoba, J., & Vokác, P. (2020b). Improvements in utilisation of the czech national hpc center. *EPJ Web Conf.*, *245*, 09010. https://doi.org/10.1051/epjconf/202024509010

Svirin, P., De, K., Forti, A., Klimentov, A., Larsen, R., Love, P., Maeno, T., Mashinistov, R., Mukherjee, S., Nomerotski, A., et al. (2019). Bigpanda: panda workload management system and its applications beyond atlas. *EPJ Web of Conferences*, *214*, 03050.

Szabo, N. (1996). Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, *18*(2), 28.

Tadel, M., & Carminati, F. (2010). Parallelization of alice simulation-a jump through the looking-glass. *Journal of Physics: Conference Series*, *219*(3), 032024.

Taufer, M., Padron, O., Saponaro, P., & Patel, S. (2010). Improving numerical reproducibility and stability in large-scale numerical simulations on gpus. *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 1–9. https://doi.org/10.1109/IPDPS.2010.5470481

Teuber, S. (2019). Efficient unpacking of required software from CERNVM-FS. https://doi.org/10.5281/zenodo.2574462

Top500 the list. (2022). Retrieved August 24, 2022, from https://www.top500.org/

Tovar, Benjamin, Bockelman, Brian, Hildreth, Michael, Lannon, Kevin, & Thain, Douglas. (2021). Harnessing hpc resources for cms jobs using a virtual private network. *EPJ Web Conf.*, *251*, 02032. https://doi.org/10.1051/epjconf/202125102032

Tremblay, M., Grohoski, G., Burgess, B., Killian, E., Colwell, R., & Rubinfeld, P. (1998). Challenges and trends in processor design. *Computer*, *31*(1), 39–48. https://doi.org/10.1109/2.641976

Tsaregorodtsev, A., & Hamar, V. (2012). Mpi support in the dirac pilot job workload management system. *Journal of Physics: Conference Series*, *396*(3), 032109.

Tsaregorodtsev, A. (2014). Dirac distributed computing services. *Journal of Physics: Conference Series*, *513*(3), 032096. https://doi.org/10.1088/1742-6596/513/3/032096

Turilli, M., Santcroos, M., & Jha, S. (2018). A comprehensive perspective on pilot-job systems. *ACM Comput. Surv.*, *51*(2), 43:1–43:32. https://doi.org/10.1145/3177851

Uncvmfs. (2018). Retrieved August 24, 2022, from https://github.com/ic-hep/uncvmfs

Unified cloud interface project. (2012). Retrieved August 24, 2022, from https://code.google.com/archive/p/unifiedcloud/

Uriarte, R. B., & DeNicola, R. (2018). Blockchain-based decentralized cloud/fog solutions: challenges, opportunities, and standards. *IEEE Communications Standards Magazine*, *2*(3), 22–28.

Valassi, Andrea, Alef, Manfred, Barbet, Jean-Michel, Datskova, Olga, De Maria, Riccardo, Fontes Medeiros, Miguel, Giordano, Domenico, Grigoras, Costin, Hollowell, Christopher, Javurkova, Martina, Khristenko, Viktor, Lange, David, Michelotto, Michele, Rinaldi, Lorenzo, Sciabà, Andrea, & Van Der Laan, Cas. (2020). Using hep experiment workflows for the benchmarking and accounting of wlcg computing resources. *EPJ Web Conf.*, *245*, 07035. https://doi.org/10.1051/epjconf/202024507035

Vicente, D., & Bartolome, J. (2010). Bsc-cns research and supercomputing resources. In M. Resch, S. Roller, K. Benkert, M. Galle, W. Bez, & H. Kobayashi (Eds.), *High performance computing on vector systems 2009* (pp. 23–30). Springer Berlin Heidelberg.

Volkl, Valentin, Madlener, Thomas, Lin, Tao, Wang, Joseph, Konstantinov, Dmitri, Razumov, Ivan, Sailer, Andre, & Ganis, Gerardo. (2021). Building hep software with spack: experiences from pilot builds for key4hep and outlook for lcg releases. *EPJ Web Conf.*, *251*, 03056. https://doi.org/10.1051/epjconf/202125103056

Von Hagen, W. (2011). *The definitive guide to gcc*. Apress.

Welch, V., Siebenlist, F., Foster, I., Bresnahan, J., Czajkowski, K., Gawor, J., Kesselman, C., Meder, S., Pearlman, L., & Tuecke, S. (2003). Security for grid services. *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, 48–57.

Whitehead, M. P. (2018). *A palette of fast simulations in lhcb* (tech. rep.).

Willett, H. A. (2019). Building Key4HEP with Spack. https://cds.cern.ch/record/2687469

Withers, A., Bockelman, B., Weitzel, D., Brown, D., Gaynor, J., Basney, J., Tannenbaum, T., & Miller, Z. (2018). Scitokens: capability-based secure access to remote

scientific data. *Proceedings of the Practice and Experience on Advanced Research Computing*. https://doi.org/10.1145/3219104.3219135

Worldwide lhc computing grid. (2022). *CERN*. Retrieved August 24, 2022, from https://wlcg.web.cern.ch/

Wu, W., Hara, T., Miyake, H., Ueda, I., Kan, W., & Urquijo, P. (2017). BelleII@home: integrate volunteer computing resources into DIRAC in a secure way. *Journal of Physics: Conference Series, 898*, 102003. https://doi.org/10.1088/1742-6596/898/10/102003

Xu, Benda, Amadio, Guilherme, Gro, Fabian, & Haubenwallner, Michael. (2020). Gentoo prefix as a physics software manager. *EPJ Web Conf., 245*, 05036. https://doi.org/10.1051/epjconf/202024505036

Yeo, C. S., Buyya, R., Pourreza, H., Eskicioglu, R., Graham, P., & Sommers, F. (2006). Cluster computing: high-performance, high-availability, and high-throughput processing on a network of computers. *Handbook of nature-inspired and innovative computing* (pp. 521–551). Springer.

Yoo, A. B., Jette, M. A., & Grondona, M. (2003). Slurm: simple linux utility for resource management. In D. Feitelson, L. Rudolph, & U. Schwiegelshohn (Eds.), *Job scheduling strategies for parallel processing* (pp. 44–60). Springer Berlin Heidelberg. https://doi.org/10.1007/10968987_3

Yu, J., & Buyya, R. (2005). A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing, 3*(3), 171–200. https://doi.org/10.1007/s10723-005-9010-8

A Pilot-Job provisioning on Grid Resources: Collecting Analysis and Performance Evaluation Data

This appendix was published in the Data In Brief journal (Boyer et al., 2022b) and was developed as a companion paper describing data related to an article published in the Future Generation Computer Systems (FGCS) journal: "DIRAC Site Director: Improving Pilot-Job provisioning on grid resources" (Boyer et al., 2022a).

A.1 Introduction

To take advantage of the computing power offered by grid and opportunistic resources, the CERN Large Hadron Collider (LHC) experiments have adopted the Pilot-Job paradigm. In this work, we study the DIRAC Site Director, one of the existing Pilot-Job provisioning solutions, mainly developed and used by the beauty experiment (LHCb). The purpose is to improve the Pilot-Job submission rates and the throughput of the jobs on grid resources. To analyze the DIRAC Site Director mechanisms and assess our contributions, we collected data over 12 months from the LHCbDIRAC instance. We extracted data from the DIRAC databases and the logs. Data include (i) evolution of the number of Pilot-Jobs/jobs over time; (ii) slots available in grid Sites; (iii) number of jobs processed per Pilot-Job.

A.2 Specifications Table

Subject	Software Engineering
Specific subject area	Analysis and evaluation of the Pilot-Job provisioning on grid resources
Type of data	Text, Comma Separated Files (CSV), JavaScript Object Notation (JSON), Figures
How data were acquired	A Local Resource Management System (LRMS) of a grid site provides a state of the computing resources and their usage. Computing Elements (CEs) aggregate data from one or multiple LRMS and record information about jobs and Pilot-Jobs. They transfer data between grid sites and Workload Management System (WMS). A Site Director generates Pilot-Jobs and stores related data in a *PilotAgentsDB* database and an accounting service (DIRAC Accounting service). It supplements Pilot-Jobs data with information coming from CEs and produces logs containing details about the operations performed.
Data format	Raw, Filtered, Analyzed
Description of data collection	Data collected provided insights about the limits of the Site Director and the impact of our contributions to improve the throughput of the jobs on grid resources. Data collection was performed for 12 months and targeted a specific group of Site Directors. It was split into three different phases: (i) the first phase (4 months) represents the original state of the group; (ii) the second phase (4 months) is related to a contribution introduced in production; (iii) in the same way, the last phase (4 months) corresponds to another contribution. The evolution of the number of jobs processed in parallel and the number of Pilot-Jobs submitted per hour was collected at the end of the analysis from the DIRAC web interface, which interacts with the Accounting service. Further details about the operations of the Site Directors and the activities of the Pilot-Job were extracted, from the log files and from client interfaces interacting with the *PilotAgentsDB* database and the CEs, for a short period (a few hours), multiple times per phase.
Data source location	Institution: European Organization for Nuclear Research (CERN) City: Meyrin Country: Switzerland Latitude and longitude: 46.2338702,6.0469869
Data accessibility	Repository name: DIRAC Site Director: Analysis and Performance Evaluation Data identification number: 10.17632/6r388827fz.2 Direct URL to data: https://data.mendeley.com/datasets/6r388827fz/2
Related research article	Alexandre F. Boyer, Christophe Haen, Federico Stagni, David R.C. Hill, DIRAC Site Director: Improving Pilot-Job provisioning on grid resources, Future Generation Computer Systems, Volume 133, 2022, Pages 23-38, ISSN 0167-739X, https://doi.org/10.1016/j.future.2022.03.002.

A.3 Value of the Data

- This dataset provides metrics and directions to analyze and assess the efficiency of a Pilot-Job provisioning system on grid resources.
- DIRAC is an open-source interware used by various experiments such as LHCb, Belle II and CTA; in different contexts: WLCG, EGI. DIRAC administrators could directly reuse this work. This dataset could also provide insights and guidance to any virtual organization applying the Pilot-Job paradigm on grid resources.
- These data might be used to analyze the limits of a Pilot-Job provisioning tool, to assess the scaling capabilities of performance improvement contributions.
- These data could potentially help virtual organizations to better understand the functioning of grid computing resources, and thus to better exploit them.

A.4 Data Description

The dataset is split into two parts:

- `Resources`: raw data coming from LHCbDIRAC, divided into subsections according to the nature of the data. Each subsection contains data along with a short markdown description (`README.md`) and a set of Python programs used to extract and filter data (`tools`). They are proposed and designed in order to facilitate the reproducibility of the experiments.
- `Results`: processed data including figures and tables. They are produced by a Jupyter notebook program (`DIRACSiteDirector.ipynb`) exploiting raw data.

A.4.1 Raw data

Grid computing resources are heterogeneous and volatile. They are owned by many different institutes across the world and shared by several virtual organizations. Thus, an exhaustive analysis of a Pilot-Job provisioning system requires data from many remote sources. In the `Resources` directory, information is sorted into subsections according to its source and nature:

- `configInfo`: gives information about associations between Site Directors, sites and CEs: (i) `cesPerSite.json` is a dictionary where the Site Director name is the key and names of the CEs bound to it is the value. The CE identifier contains the grid name, the site name and the CE name such as $grid_{ID}.site_{ID}.ce_{ID}$. (ii) `siteDirectortypeCE.csv` is another dictionary bounding a type of CE to a Site Director. Indeed, in LHCbDIRAC, administrators chose to associate a Site Director to a single type of CE. Information comes from the DIRAC configuration service relying on BDII, a database centralizing data from grid sites and CEs.

- `jobsPerPilot`: provides a table representing the average number of jobs fetched and processed per Pilot-Job, grouped by CE, in a month (`jobsPerPilot.csv`). Each line corresponds to a day, each column to a CE, and the values to the average number of jobs processed per Pilot-Job. Data come from the Accounting service and were extracted from the DIRAC web interface.

- `pilot-jobsSubmitted`: represents the evolution of (i) the number of Pilot-Jobs submitted per hour (`pilotsSubmitted`); (ii) the number of jobs processed in parallel (`jobs Processed`). `pilotsSubmitted` contains two files: `failed-0101-20-190121.csv` the evolution of the failed submission of pilots and `success-01-0120-190121.csv` the progression of the successfully submitted pilots. In both files, each line represents a week, each column a Site Director and values the average number of pilots submitted by a given Site Director for a given week. `jobsProcessed` has one file (`010120-190121.csv`): each line corresponds to a week, each column to a grid site, and the values to the average number of jobs processed by a given site for a given week. Files were downloaded from the Accounting service through the DIRAC web interface.

- `submission-matchingTime`: is composed of two tables: (i) `pilotDuration.csv` and `jobDuration.csv`. `pilotDuration.csv` represents the time a Pilot-Job spends from its submission to its execution on a computing resource. Each line represents a Pilot-Job. Columns 1, 4 and 5 provide information about sites and CEs involved and columns 2 and 3 are the installation date and the submission date, respectively. `jobDuration.csv` is relatively similar but contains fewer details. In this file, each line describes a job as two dates: the moment when the job arrives in DIRAC, the moment when the job is fetched by a Pilot-Job running in a grid site. To produce the files, one has to (i) get job and Pilot-Job identifiers

from the DIRAC web interface; (ii) call DIRAC client interfaces interrogating the DIRAC databases to get further details about the chosen entities.

- `pilotsActivities`: contains information about the statuses of the Pilot-Jobs under the form of JSON files. The directory contains several files named `result_sorted_<date>`, where *date* corresponds to the moment where data were extracted. Each file contains the number of Pilot-Jobs grouped by status and by date. Values were extracted every 5 minutes for 12 hours using a DIRAC client interface interacting with the *PilotAgentsDB* database.

- `individualEvaluation`: used to assess individual contributions brought to the Pilot-Job provisioning tool. `arcEvalOutput.csv` is the result of 3 executions of a script that compared two methods to get the status of a variable number of Pilot-Jobs on 3 ARC CEs. `creamEvalOutput.csv` was generated by a script, executed 5 times, comparing two methods to renew the proxy of 2 CREAM CEs. `parallelEvalOutput.csv` was built by a program comparing the monitoring operations of a Site Director interacting with a variable number of CEs - from 1 to 5 - with (i) one thread and (ii) multiple threads.

- `logs`: embeds several directories named `<date>` corresponding to the date of extractions of the underlying log files. Each directory contains log files bound to Site Directors. Logs have information of interests that have to be extracted, such as the number of slots available in the grid sites, the number of pilots submitted and the duration of the operations.

A.4.2 Processed data

Once collected from different sources, data have to be combined and easily readable to provide insights to the developers. `DIRACSiteDirector.ipynb` was used to process raw data from `Resources` in order to generate figures and tables in `Results`:

- `jobsPerPilot.pdf`: generated from `Resources/jobsPerPilot` under the form of a heat map.

- `submission-matching.pdf`: box plot combining data from `Resources/submission-matchingTime`. It compares the duration from the pilot generation to

the pilot installation on a worker node to the duration from the job arrival to the job matching.

- `pilotActivities.pdf`: combines information coming from `Resources/pilotsActivities` and `Resources/logs` to highlight the variations Pilot-Jobs in certain sites.

- `SDsMonitoring.pdf`: compares duration of the operations using data from the logs (`Resources/logs`).

- `individualEval<contribution>.pdf`: highlight the results of a given contribution.

- `cpuTimeUsedPerSecond.pdf` and `pilotsSubmittedPerHour.pdf`: represent the evolution of the number of jobs processed in parallel and the number of pilots submitted, respectively, in bar plots. They both use data from `Resources/pilot-jobsSubmitted`

- `runningPilots.pdf` and `scheduledPilots.pdf`: box plots using Pilot-Jobs activities information located in `Resources/pilotsActivities`. It represents the evolution of the number of running and waiting pilots through different phases.

- `monitoringNumberPilotsSubmitted.pdf`: plot describing the evolution of the monitoring time and the number of Pilot-Jobs submitted per hour through different phases. Data comes from logs and raw data from various subsections of `Resources` and involves 13 Site Directors.

- `errorsPerSD.tex`: is a table containing the number of failed submissions observed in Site Directors grouped by phase. Data comes from `Resources/pilot-jobsSubmitted`.

- `numberOfPilotsSubmittedEvolution.pdf`: box plot relying on logs showing the evolution of the number of pilots submitted per cycle of Site Director, through different phases.

A.5 Experimental Design, Materials and Methods

A.5.1 Getting data from grid resources

Getting data from a large number of heterogeneous and remote computing resources require centralization mechanisms at some point:

- Administrators can install BDII agents on grid sites, which are able to collect LRMS configuration data (Figure A.1.1.1 and A.1.1.2). These data are centralized and can directly be used by WMS.
- LRMS orchestrate computing resources of grid sites and gather data about the state and the use of the resources to schedule jobs. To ease the interactions with a large number of grid sites with various types and versions of LRMS, WMS deal with entry points called CEs. CEs receive jobs and Pilot-Jobs from WMS and transfer them to a LRMS of a grid site. They generally embed a service to record and send information about the Pilot-Jobs and their status. A Pilot-Job provisioning tool - such as the DIRAC Site Director - generally acts as a centralization point: (i) it generates and submits Pilot-Jobs to different CEs bound to several grid sites; (ii) it gets information about the submitted Pilot-Jobs and records information in one or more databases (Figure A.1.2.1 and A.1.2.2).
- Pilot-Jobs, once installed, communicate information about worker nodes to the WMS services. They are also stored in one or more databases (Figure A.1.3.1).

A.5.2 Collecting Pilot-Jobs provisioning related data

The DIRAC interware provides different means to interact with WMS-related data:

- A web interface: an administrator can generate plots and CSV files about Pilot-Jobs and jobs, for a given period. The web interface is linked to (i) the Accounting service aggregating data from many DIRAC services; (ii) a *JobManager* service to interact with the *JobDB* database; (iii) a *PilotManager* service to get information from the *PilotAgentsDB*.

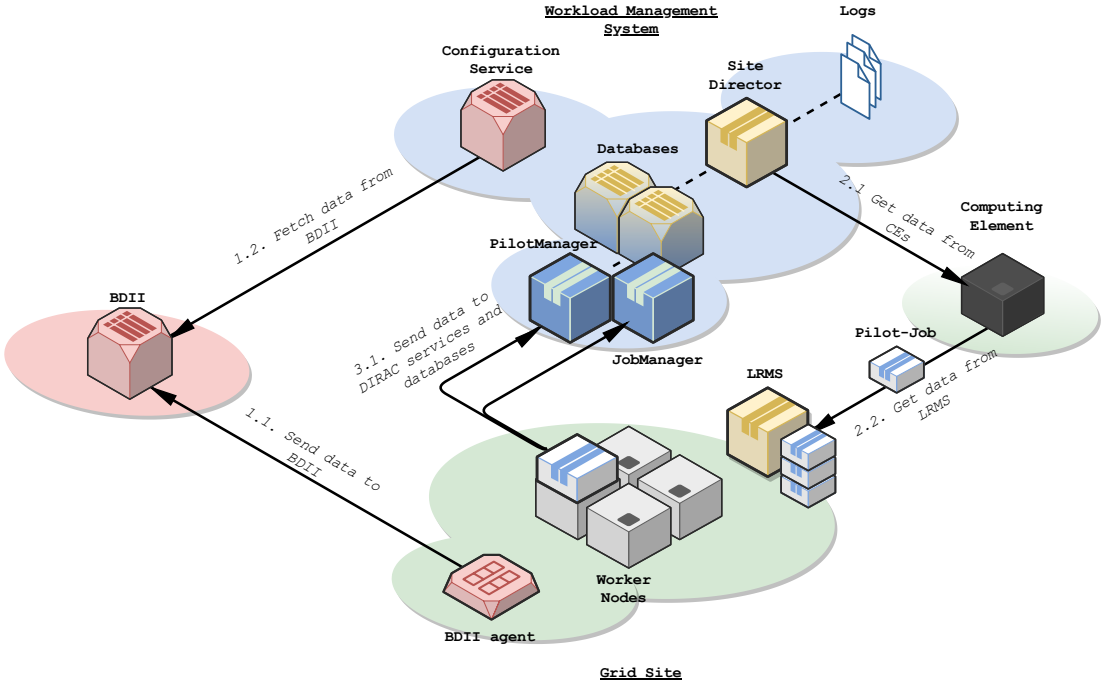

Figure A.1 – Interactions between grid components to centralize data.

- Command-line interfaces: in the same way, an administrator can use command-line interfaces from a terminal to interrogate DIRAC databases.
- Logs: DIRAC services produces logs that are stored in files. Access to the DIRAC server is generally required.

We collected data about Pilot-Jobs and jobs related to the LHCb experiment offline activities on WLCG for 12 months split into 3 phases. The experiment involved a group of 13 Site Directors supplying 65 sites with Pilot-Jobs. They were interacting with different types of CEs and LRMS. During the first phase (4 months), we analyzed the Site Directors to find their limits. The second phase (4 months) started after we introduced changes: we decreased the number of communications and their duration with CEs. Finally, the third phase (4 months) began after we configured Site Directors to submit Pilot-Jobs more frequently.

The process was similar for the 3 phases. We extracted the logs of the Site Director that contained 2 to 3 days of information, and we accessed database information, multiple times for short periods, to produce average results (Figure A.2.1.1). Getting average values limits the bias that could be introduced if grid sites are in maintenance

for instance. We also got CSV files from the web application to perform an analysis in the long term and at a large scale (Figure A.2.1.2).

A.5.3 Extracting knowledge from raw data

We ended up with data from different sources, and we wanted to combine them to build knowledge that would help to improve the Pilot-Job provisioning tool (Figure A.2.2). First, we designed programs to filter information from raw and processed data: site and CE names were replaced by identifiers. Then, we created a Jupyter notebook to import log, CSV and JSON files - filtered - and we built programs to combine data and generate figures and tables A.2.3).

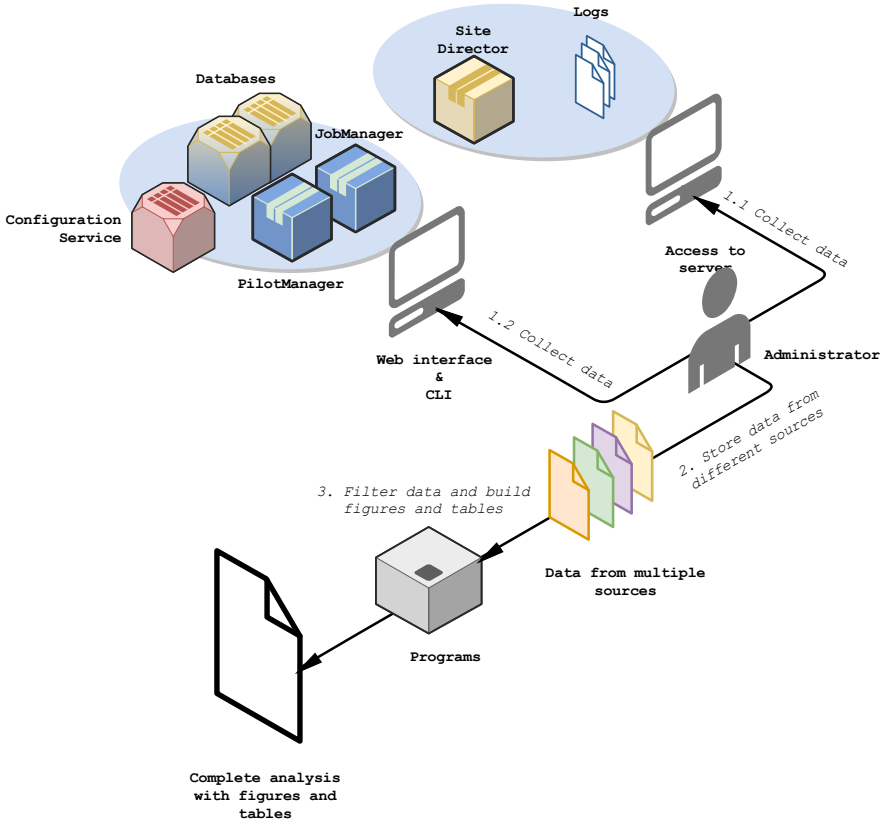

Figure A.2 – Workflow: collecting and processing WMS data.

B Porting DB12 to Python3: Collecting CPU power estimations

This appendix was partly developed as a Zenodo repository. It aims at complementing an article presented at the International Conference on High Energy Physics (ICHEP) 2022, not yet published: "Porting DIRAC Benchmark to Python3: impact of the discrepancies and solution".

B.1 Introduction

DB12 has been originally conceived with Python2 to estimate the power of a given CPU to run HEP applications. However, since January 2020, Python2 is no longer maintained and we decided to port the code to Python3, which contains several optimizations. In October 2021, we effectively ported DB12 to Python3.9, but the optimizations brought by the language generated discrepancies in the norm score, which is a critical component to evaluate the power of CPUs. The purpose is to evaluate the impact of these discrepancies and study approaches to mitigate them. We extracted data from thousands of jobs across 102 distributed grid sites. Data include (i) the results of DB12 executed with Python2 and Python3; (ii) the environments of the nodes.

B.2 Specifications Table

Subject	Software Engineering
Specific subject area	Analysis and evaluation of the DIRAC Benchmark scores using Python3
Type of data	JavaScript Object Notation (JSON), Figures
How data were acquired	Jobs are executed across distributed grid sites and produce JSON files. At the end of their executions, jobs upload outputs in a sandbox, accessible from outside. The DIRAC Workload Management System (WMS) provides scripts to interact with jobs and their outputs.
Data format	Raw, Analyzed
Description of data collection	Data collected provide insights into the impact of the discrepancies between the Python2 and Python3 executions of the DIRAC Benchmark. Analyzed data also highlight experiments conducted in order to correct the scores. Data collection was performed for 2 hours and targeted 102 sites.
Data source location	Institution: European Organization for Nuclear Research (CERN) City: Meyrin Country: Switzerland Latitude and longitude: 46.2338702,6.0469869
Data accessibility	Repository name: Porting DB12 to Python3: Analysis of the scores Data identification number: 10.5281/zenodo.5647834 Direct URL to data: https://zenodo.org/record/5647834
Related research article	Alexandre F. Boyer, Imane Iraoui, Christophe Haen, Federico Stagni, David R.C. Hill, Porting DIRAC Benchmark to Python3: impact of the discrepancies and solution, International Conference on High Energy Physics, 2022, *to be published*

B.3 Value of the Data

- This dataset provides metrics and directions to analyze and assess the changes brought to a CPU benchmarking tool used across distributed and heterogeneous computing resources.
- DIRAC is an open-source interware used by various experiments such as LHCb, Belle II and CTA; in different contexts: WLCG, EGI. DIRAC administrators could directly reuse this work. This dataset could also provide insights and guidance to any virtual organization employing DIRAC Benchmark on distributed computing resources.
- These data might be used to provide insights to mitigate involved discrepancies.

- These data could potentially help virtual organizations to better exploit heterogeneous and time-limited computing resources, and thus improve the throughput of the jobs.

B.4 Data Description

The dataset is split into two parts. `Resources` contains raw data coming from LHCbDIRAC as well as the scripts to generate them, whereas `Results` is composed of processed data taking the form of plots. A Jupyter notebook program (`DB12Analysis.ipynb`) exploits raw data contained in `Resources` to produce plots in `Results` allowing us to better understand the differences between Python 2 and Python 3 executions. It also covers several aspects of the experiment conducted.

B.4.1 Raw data

The `Resources` directory is split into two subsections:

- `Tools`: shell and python scripts used to: (i) submit jobs executing both DB12 with Python2 and Python3; (ii) collect results and generate JSON files in `Resources/Results`. They are proposed and designed in order to facilitate the reproducibility of the experiments, even though they rely on dynamic components such as CVMFS, and therefore, might not work as expected in the future.
- `Results`: processed data including JSON files ordered by creation date. They are produced by the scripts mentioned above.

Figure B.1 describes the structure of a JSON file resulting from the experiment conducted. As we can observe, the JSON file is composed of several job identifiers that were part of the process. Each job identifier refers to technical details about:

- The environment of the host: computed by the scripts present in `Tools` that aim at extracting data related to the WN such as its name, the CPU model involved and its frequency, the OS installed and the amount of memory available.
- The parameters of the job: mostly coming from the Pilot-Job which also evaluates the environment of the host. The Pilot-Job also computes DB16 to get an

estimation of the CPU power and interrogates the batch system to get CPU time left within the allocation.

- The scores: computed by many iterations of DB12 executions, both with Python 2 and Python 3. Each score is designated by a combination of an iteration identifier and a Python version. *NORM* denotes the CPU power computed by DB12.

B.4.2 Processed data

Once collected, raw data have to be combined to provide insights to the developers. `DB12Analysis.ipynb` was used to process raw data from `Resources` in order to generate figures and tables in `Results`:

- `cpu-model.csv`: table associating an identifier to a CPU model name. We decided to use identifiers instead of CPU model names within the plots to make them clear and readable.
- `distribCPU.pdf`: distribution of the DB12 executions among the CPU models.
- `distribSites.pdf`: distribution of the DB12 executions among the sites.
- `distribIterations.pdf`: line plots highlighting variations of the DB12 results after consecutive executions in the same environment.
- `py3vspy2_<original/sol>.pdf`: scatter plot comparing Python3 and Python2 executions. The `sol` suffixes indicate that the Python3 results to better match Python2 results.
- `learning_curves.pdf`: line plot bound to `py3vspy2_sol3.pdf` emphasizing the learning curves produced during the training phase.

B.5 Experimental Design, Materials and Methods

The DIRAC interware combined with a grid certificate allows us to interact with a large number of distributed and heterogeneous computing resources embedding various

```
{
  "<JobID>": {
    "environment": {
      "<Details about the host machine>": [
            "name, CPU model, OS, memory..."
        ]
    },
    "jobparams": {
      "<Details about the parameters of the job>": [
            "CE and queue names",
            "DB16 result",
            "CPU time",
            "Pilot-Job reference and version"
        ]
    },
    "results": {
      "<iteration>_<python version>": {
        "PythonVersion": {
          "Major": "X",
          "Minor": "Y",
          "Micro": "Z",
        },
        "scores": [
          {
            "UNIT": "DB12"
            "TYPE": "single",
            "COPIES": "<number of copies>",
            "ITER": "<iteration ID>",
            "NORM": 29.342723004694836,
            "CPU": 8.52,
            "WALL": 8.53999999910593,
          }
        ]
      },
      ...
    }
  }
  ...
}
```

Figure B.1 – Structure of the JSON file resulting from the execution of the jobs.

types of CPU models. In the context of the LHCb experiment, we have access to WLCG computing resources that mainly contain Intel and AMD CPUs.

We designed `submit.sh` to generate and submit a given number of jobs to given sites. The script relies on the DIRAC API to create job objects. Each job object is composed of `executable.sh`, which aims at (i) downloading DIRAC Benchmark from GitHub, (ii) enhancing it with further commands to collect data within a JSON file and (iii) executing it using Python2 and Python3 multiple times.

Once launched, jobs take about 55 minutes to complete and we get their results using `getDB12Scores.py`. It extracts data from the outputs of the Pilot-Jobs and the jobs involved and gets the JSON files generated during the execution of the jobs. It creates a JSON file gathering these data, following the pattern presented in Figure B.1.

We end up with one JSON file containing all the raw data needed to perform an analysis. We designed a Jupyter notebook to import the JSON file and transform it into a data table object to manipulate data. From this object, we developed programs to produce the plots stored in (`Results`).

C Integrating LHCb workload on the Santos Dumont supercomputer: Collecting Pilot-Jobs and Jobs Outputs

This appendix was partly developed as a Mendeley Data repository (Boyer, 2021b). It presents data related to the integration of the LHCb workload on SDumont and aims at providing the source of many results described in chapter 4.

C.1 Introduction

To handle the growing amount of data coming from the LHC Run3 and then the High-Luminosity LHC phases, LHCb aims at integrating Monte-Carlo simulation workflows into supercomputers such as Santos Dumont, hosted in LNCC, in Brazil. Data focus on the use of Santos Dumont CPU resources for three months, in the context of LHCb. Data mainly include CSV files related to the CPU usage of the resources (CPU benchmark results, CPU seconds processed per second, statuses of the jobs, wallclock time allocated and used), as well as a Jupyter Notebook to present plots based on data. Data show that providing a more accurate CPU power value and leveraging multi-node allocations would ease the exploitation of a larger number of resources.

C.2 Specifications Table

Subject	Software Engineering
Specific subject area	Analysis and evaluation of the jobs within the Santos Dumont supercomputer
Type of data	Text, Comma Separated Files (CSV), JavaScript Object Notation (JSON), Figures
How data were acquired	Pilot-Jobs are executed on Santos Dumont. They collect data related to their environment and fetch jobs from the DIRAC WMS. Jobs are executed on the worker nodes and produce further data: mainly JSON and text files. Pilot-Job outputs are accessible from the shared file system of the supercomputer while job outputs are available via output sandboxes located outside of the cluster. The DIRAC Workload Management System (WMS) provides scripts to interact with jobs and their outputs.
Data format	Raw, Analyzed
Description of data collection	Data collected provide insights into the executions of the LHCb jobs within the Santos Dumont supercomputer. Data collection targeted more than 19700 Pilot-Jobs executed in the supercomputer.
Data source location	Institution: European Organization for Nuclear Research (CERN) City: Meyrin Country: Switzerland Latitude and longitude: 46.2338702,6.0469869
Data accessibility	Repository name: Integration of LHCb workflows on the Santos Dumont supercomputer Data identification number: 10.17632/c7w3cgfzgt.1 Direct URL to data: https://data.mendeley.com/datasets/c7w3cgfzgt/1
Related research article	

C.3 Value of the Data

- This dataset provides metrics and directions to analyze and assess some of the obstacles to the integration of LHCb workload on supercomputers.

- DIRAC is an open-source interware used by various experiments such as LHCb, Belle II and CTA; in different contexts: WLCG, EGI. DIRAC administrators could directly reuse this work. This dataset could also provide insights and guidance to any virtual organization intending to integrate similar tasks on supercomputers with outbound connectivity.

- These data could potentially help virtual organizations exploit further heterogeneous and highly-constrained computing resources, getting access to additional CPU power and thus improving the throughput of the jobs.

C.4 Data Description

The dataset is split into two parts. `data` contains raw data coming from LHCbDIRAC and the supercomputer, whereas `res` is composed of processed data taking the form of plots. A Jupyter notebook program (`SDumontAnalysis.ipynb`) exploits raw data contained in `data` to produce plots in `res` allowing us to better understand the usage of SDumont.

C.4.1 Raw data

The `data` directory is split into two subsections:

- `hpc`: gathers data extracted directly from the supercomputer (`DB12` and `Pilot`).
- `webapp`: contains information coming from the DIRAC web application and the CLI (`Accounting` and `Jobparams`).

Each directory within these subsections have a similar structure:

- `README.md`: a text file describing data and how to get them.
- `Tools`: shell and python scripts used to extract, anonymize and enhance raw data. They are proposed and designed in order to facilitate the reproducibility of the experiments.
- `<result>`: one or multiple file presenting data to analyze. It generally takes the form of a CSV or a JSON file. They are produced by the scripts mentioned above.

C.4.2 Processed data

Once collected, raw data have to be combined to provide insights to the developers. `SDumontAnalysis.ipynb` was used to process raw data from `data` in order to

generate figures and tables in `res`:

- `benchmarkResult.pdf`: line plots emphasizing DB16 scores depending on the number of hardware threads occupied.
- `cpuTimeCpuPower.pdf`: scatter plot comparing the CPU Power computed, the CPU time consumed and the status of the pilots.
- `cpuTimeJobs.pdf`: distribution of the CPU time consumed by jobs compared to the CPU time allocated by Slurm.
- `cpuTimePerSecond.pdf`: number of CPU seconds processed during a second for 3 months, classified by job status.
- `cpuTimePilots.pdf`: distribution of the CPU time consumed by pilots compared to the CPU time allocated by Slurm.
- `CPUWorkeventsResults.pdf`: scatter plot comparing the CPU Work per event computed during the test phase for a given production, to the CPU Work per event computed on SDumont, classified by partition used.
- `jobsPerHour.pdf`: number of jobs processed per hour for 3 months, classified by job status.
- `jobsPerPilot.pdf`: distribution of the number of jobs processed per pilot, classified by partition.
- `jobsStatus.pdf`: pie plot presenting the percentage of jobs according to their final status.
- `nbPilots_per_status.pdf`: number of pilots per status.
- `nbPilotsPerStatusPerPartition.pdf` number of pilots per status and partition.

C.5 Experimental Design, Materials and Methods

The DIRAC WMS combined with a grid certificate allows us to interact with a large number of distributed and heterogeneous computing resources. It aggregates data

coming from various sources of information within an Accounting database accessible via the web application or the command-line interface. It also retrieves parameters and attributes related to Pilot-Jobs and jobs. To get additional data about the LRMS used, we can also extract data directly from the Site if we have access to it.

C.5.1 Leveraging the LHCbDIRAC Accounting service to get aggregated data

The web interface allows DIRAC users to generate plots and CSV files about Pilot-Jobs and jobs, for a given period. The web interface is linked to (i) the Accounting service aggregating data from many DIRAC services; (ii) a *JobManager* service to interact with the *JobDB* database; (iii) a *PilotManager* service to get information from the *PilotAgentsDB*. Leveraging the web application of LHCbDIRAC, we requested CSV files of the number of jobs processed per hour and the number of CPU seconds consumed per second, classified per job status, for 3 months.

C.5.2 Analyzing DIRAC Benchmark in SDumont

We also used the web interface to get job identifiers that we passed to the DIRAC command line interface to extract the parameters of the jobs, namely (i) the number of Monte-Carlo simulation events consumed; (ii) the CPU work consumed; (iii) the CPU work per event computed during the test phase; and (iv) the partition used on the supercomputer. Results are stored in a JSON file and converted to a CSV file to be manipulated easily.

To get further information about DIRAC Benchmark in such a context, we also accessed the HPC system and executed DIRAC Benchmark in multi-core allocations multiple times. The script ran DIRAC Benchmark on 1, 2, 5, 10, 15, 20 and 24 cores and stored the result in a JSON file. The JSON files generated are then combined within a CSV file.

C.5.3 Getting further details about Pilot-Jobs within SDumont

SDumont hosts pilot outputs and allows users to get details about the executions by interrogating the LRMS. We designed `getPilotDetails.py` to extract data from the pilot outputs and the LRMS such as the partition used, the wall clock time limit, the CPU time consumed, the state of the pilots and the jobs, the execution dates of each pilot and job. These data are aggregated into a JSON file, which is then converted to a CSV file.

Lastly, we designed a Jupyter notebook to import the CSV files and transform them into data table objects to manipulate data. From these objects, we developed programs to produce the plots stored in (`res`).

Acronyms

AAI Authentication and Authorization Infrastructure. 60–62

AI Artificial Intelligence. 8, 15, 16, 25, 201

ALICE A Large Ion Collider Experiment. 3, 59, 73, 75, 76, 80

AliEN ALICE Environment. 52, 57

ARC Advanced Resource Connector. 51, 52, 55, 94, 99, 101, 104, 108, 111, 117, 119, 122, 157, 178, 182

ATLAS A Toroidal LHC ApparatuS. 3, 52, 54, 59, 73–76, 83, 203, 205

BDII Berkeley Database Information Index. 89, 122

BoT Bag-of-Tasks. 9, 10, 12, 48, 53

CE Computing Element. 50–53, 55, 59, 84, 89–91, 93, 94, 97–99, 101, 102, 104, 105, 107, 108, 117, 119, 121, 122, 151, 157, 158, 161–163, 173, 184, 202

CERN European Organization for Nuclear Research. 3, 31, 37, 38, 41, 43, 68, 78, 79, 205

CMS Compact Muon Solenoid. 3, 52, 73, 75

CREAM Computing Resource Execution And Management. 50, 51, 99, 101, 105, 108, 111, 117

CTA Cherenkov Telescope Array. 4, 52

CVMFS CernVM-File System. 48, 68, 69, 71–74, 80, 85, 129, 151, 153, 156, 163, 165, 166, 168–171, 177, 178, 181, 186, 193, 197, 202, 203, 206

DAN Data Access Node. 59, 60

DCI Distributed Computing Infrastructure. 49

DCR Distributed Computing Resource. 49, 50

DIRAC Distributed Infrastructure with Remote Agent Control. 4, 5, 38, 52, 53, 55, 57, 61, 63, 67, 75–77, 81, 84, 88–93, 97–99, 101, 102, 104, 107, 121, 122, 126, 146, 150–153, 155–159, 161, 163, 172, 173, 175, 176, 181, 182, 184, 195, 197, 201–203, 206, 207

DMS Data Management System. 2, 4

DST Data Summary Tape. 35

EGI European Grid Infrastructure. 4, 27, 41, 52, 205

GIL Global Python Interpreter. 99

HEP High Energy Physics. 8, 15, 35, 42, 43, 51, 52, 54, 61, 68, 77–82

HPC High-Performance Computing. 11, 12, 18, 24, 27, 30, 43, 44, 50, 55, 62, 66, 67, 73, 74, 77, 80, 145, 202, 204, 206

HTC High-Throughput Computing. 11, 12, 27, 30, 44, 50, 84, 153, 202–205

LHC Large Hadron Collider. 3–5, 31–33, 35, 36, 40–43, 45, 52, 58, 71, 72, 74, 79–81, 88, 122, 123, 145, 147, 150, 153, 176, 197, 201, 203, 205, 207

LHCb Large Hadron Collider beauty. 3–5, 8, 32, 34–37, 40, 42–45, 52, 59, 64–66, 75, 77, 80, 82, 85, 88, 91, 94, 97, 109, 117, 120–123, 129, 137, 142, 145, 147, 148, 150, 151, 153, 155, 168–171, 175, 177, 178, 181, 184–186, 188, 193, 197, 201, 203–206

LHCbPR LHCb Performance and Regression. 65

LRMS Local Resource Management System. 24, 49–54, 59, 63, 74, 76, 81–84, 89, 90, 92–94, 97–99, 101–103, 106–108, 111, 119–121, 124, 125, 151–153, 155, 161, 173, 175, 177, 193

LSF Load Sharing Facility. 50

NeIC Nordic e-Infrastructure Collaboration. 27

OSG Open Science Grid. 27, 41

PanDA Production ANd Distributed Analysis system. 52, 53, 57, 59, 84

QoS Quality of Service. 1, 2, 9, 10

SLURM Simple Linux Utility for Resource Management. 50, 173, 175, 177, 185, 186, 188, 193–195

VM Virtual Machine. 20, 22, 48, 59, 67, 81, 158

VO Virtual Organization. 26, 27, 62, 92, 93, 107, 109, 121, 122, 146, 190

VOMS Virtual Organization Membership Service. 61, 62

WLCG Worldwide LHC Computing Grid. 3–5, 8, 27, 40–43, 45, 54, 57, 58, 60–62, 68, 69, 74, 79–81, 85, 88, 122, 123, 129, 137, 145, 147, 148, 150, 153, 170, 177, 197, 199, 202, 203

WMS Workload Management System. 2, 4, 48, 52–57, 60, 61, 63, 74, 76, 84, 122, 124, 126, 145, 150, 153, 155, 157, 159, 176, 184, 201, 204, 207, 208

WN Worker Node. 53–57, 59, 61, 67, 72, 74, 84, 85, 89, 90, 92, 102, 120, 124, 125, 137, 151, 152, 155–157, 168, 172, 173, 185, 193