EPJ Web of Conferences **214**, 03057 (2019) https://doi.org/10.1051/epjconf/201921403057
CHEP 2018

PanDA and RADICAL-Pilot Integration: Enabling the Pilot Paradigm on HPC Resources

Andre Merzky[2], *Pavlo* Svirin[1], and *Matteo* Turilli[2,*]

[1]Rutgers University, New Brunswick, NJ, USA
[2]Brookhaven National Laboratory (BNL), Upton, NY, USA

Abstract. PanDA executes millions of ATLAS jobs a month on Grid systems with more than 300,000 cores. Currently, PanDA is compatible only with few high-performance computing (HPC) resources due to different edge services and operational policies; does not implement the pilot paradigm on HPC; and does not dynamically optimize resource allocation among queues. We integrated the PanDA Harvester service and the RADICAL-Pilot (RP) system to overcome these limitations and enable the execution of ATLAS, Molecular Dynamics and other workloads on HPC resources. This paper offer two main contributions: (1) introducing PanDA Harvester and RADICAL-Pilot, two systems independent developed to support high-throughput computing (HTC) on high-performance computing (HPC) infrastructures; (2) describing the integration between these two systems to produce a middleware component with unique functionalities, including the concurrent execution of heterogeneous workloads on the Titan OLCF machine. We integrated Harvester and RP by prototyping a Next Generation Executor (NGE) to expose RP capabilities and manage the execution of PanDA workloads. In this way, we minimized the reengineering of the two systems, allowing their integration while being in production.

1 Introduction

Production ANd Distributed Analysis (PanDA) [1] is the Workload Management System (WMS) [2] used by the ATLAS experiment at the Large Hadron Collider (LHC) to execute scientific applications on widely distributed resources. PanDA is designed to support the execution of distributed workloads via pilots [3]. Pilot-capable WMS enable high throughput computing (HTC) by executing tasks via multi-level scheduling while supporting interoperability across multiple sites. Special jobs (i.e., pilots) are submitted to each site and, once active, tasks are directly scheduled to each job for execution without passing through the site's batch and scheduler systems [4]. This reduces queue time, increasing task throughput. Pilots are particularly relevant for LHC experiments, where millions of tasks are executed across multiple sites every month, analyzing and producing petabytes of data.

The implementation of PanDA WMS consists of several interconnected subsystems, communicating via dedicated application program interfaces (API) or HTTP messaging, and implemented by one or more modules. Databases are used to store stateful entities like tasks,

*Corresponding author: matteo.turilli@rutgers.edu

jobs and input/output data, and to store information about sites, resources, logs, and accounting. PanDA can execute ATLAS tasks on a variety of infrastructures, including high performance computing facilities. In the past three years, PanDA was used to process around 11 million jobs on Titan, a leadership-class machine managed by the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory (ORNL).

Ref. [5] describes in detail how PanDA has been deployed on Titan. Importantly, it shows that while PanDA implements the pilot abstraction to execute tasks on grid resources, on high-performance computing (HPC) facilities PanDA submits jobs directly to the batch system of the machine. The size and duration of these jobs depend on resource availability and workload requirements. PanDA polls resource availability from Titan every ten minutes via the backfill functionality [6] of the Moab scheduler [7] and, in case of available resources, submits a job requiring that amount of resources. The workload of this job is sized so to allow its execution within the available walltime. PanDA uses Titan's resources to execute Geant4 event simulations, a specific type of workload that is amenable to execute via the machine's batch system [8].

In this paper, we present the integration between Harvester (Section 2) and Next Generation Executor (NGE) [9] (Section 3). Harvester is a new resource-facing service developed by ATLAS to enable execution of ATLAS workloads on HPC machines, bringing further coherence in the PanDA stack across the support of different types of infrastructures, like grid and HPC. NGE is a Representational State Transfer REST [10] interface developed for RADICAL-Pilot [11–13], a pilot system developed by the Research in Advanced DIstributed Cyberinfrastructure and Applications Laboratory (RADICAL) [14] and designed to support high-throughput computing on HPC infrastructures, including leadership-class machines like Titan. Different from Harvester, NGE enables to submit a pilot job via the batch system of Titan and then directly schedule tasks on the acquired resources without queuing on the machine batch system. In this way, tasks can be executed immediately while respecting the policies of the HPC machine.

PanDA can benefit from pilot capabilities in several ways. Pilot do not require to package a workload into a batch submission script, simplifying the deployment requirement. Further, pilots enable the concurrent and sequential execution of a number of tasks, until the available walltime is exhausted: Concurrent because a pilot holds multiple nodes of an HPC machine, enabling the execution of multiple tasks at the same time; sequential because when a task completes, another task can be executed on the freed resources. In this way, tasks can be late bound to an active pilot, depending on the current and remaining availability. This is important because, in principle, ATLAS would not have to bind a specific portion of tasks to an HPC machine in advance but it could bind tasks only when the HPC resources become available. This would bring further coherence to the ATLAS software stack, as pilots and late binding are already used for grid resources.

RADICAL-Pilot offers a number of distinctive features. Among these, the most relevant for ATLAS is the possibility to run arbitrary tasks on any given pilot. Specifically, RADICAL-Pilot decouples the management of a pilot, the coordination of task executions on that pilot, and what each task executes. For example, if a task executes a Geant4 simulation and another a molecular dynamic simulation, both tasks can run at the same time on the same pilot. Further, RADICAL-Pilot also enables the concurrent execution of different tasks on Central processing units (CPU) and Graphics Processing Units (GPU), allowing for the full utilization of HPC worker nodes resources. These capabilities will enable PanDA to transition from a workload management system specifically designed to support the execution of ATLAS workloads, to a system for the execution of general purpose workloads on HPC machines.

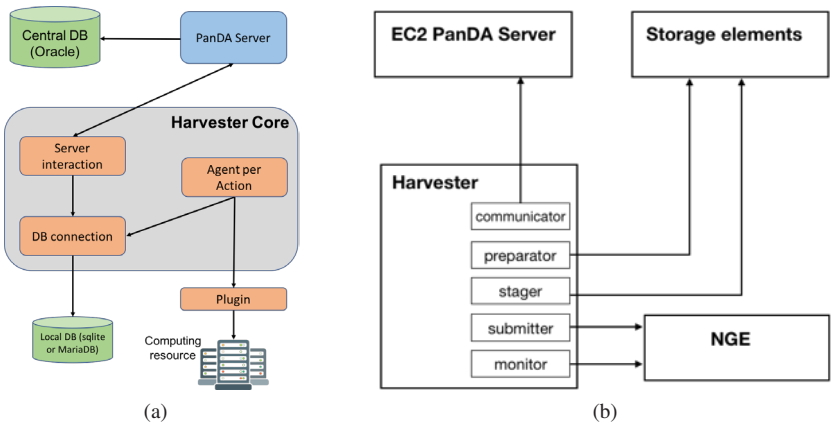

Figure 1: (a) Harvester architecture; (b) Modules used for PanDA-NGE interaction.

2 Harvester

Harvester [15] is a resource-facing service developed for the ATLAS experiment at the European Organization for Nuclear Research (CERN) since 2016 with a wide collaboration of experts. Harvester is stateless with a modular design to work with different resource types and workloads. The main objectives of Harvester are: (i) serving as a common machinery for pilot provisioning on all grid computing resources accessible to ATLAS; (ii) providing a commonality layer, bringing coherence between grid and HPC implementations; (iii) implementing capabilities to timely optimize CPU allocation among various resource types, removing batch-level static partitioning; and (iv) tightly integrating PanDA and resources to support the execution of new types of workload.

Figure 1a shows a schematic view of Harvester's architecture. Harvester is a stateless service with a local master database and a central slave database. The local database is used for real-time bookkeeping close to resources, while the central database is periodically synchronized with the local database to provide resource information to the PanDA server. The PanDA server uses this information together with a global overview of workload distribution to orchestrate a set of Harvester instances. Communication between Harvester and the PanDA server is bidirectional: Harvester fetches job descriptions that have to be executed from PanDA Server using a communicator component, and the same component is used to report back the status of these jobs.

Figure 1b shows how harvester accesses resources through different plugins. Multiple Harvester instances run on one or more edge nodes of an HPC center, and access the compute nodes of HPC resources through local HPC batch systems using specific submission plugins. The status of each batch submission is tracked using monitor plugin. Input and output data are transferred with various stager and preparator plugins, which implement connectors for Rucio [16], Globus Online [17] and other data transfer clients. Communication among the components is implemented via the local Harvester database.

In the context of the BigPanDA project, several instances of Harvester have been deployed on front nodes of BNL Institutional Cluster, NERSC (Cori), Titan and Jefferson Lab clusters. These instances are serving experiments like ATLAS, LQCD, nEDM, IceCube and LSST.

3 RADICAL-Pilot and Next Generation Executor

RADICAL-Pilot (RP) is a pilot system which enables task-level concurrency on a variety of infrastructures, including HPC and grid systems of the Extreme Science and Engineering Discovery Environment (XSEDE) [18], and the HPC machines Cheyenne at NCAR-Wyoming Supercomputing Center (NWSC) [19], Blue Waters at the National Center For supercomputing Applications (NCSA) [20], and Rhea, Titan and Summit at the Oak Ridge National Laboratory (ORNL) [21].

As a pilot system, RP allows scheduling jobs on HPC machines (or virtual machines on cloud infrastructures) to acquire computing resources [3]. Once these resources become available, RP directly schedules tasks on them, i.e., without using the machine's batch system. In this way, tasks do not wait in the queue but execute immediately, enabling high-throughput on HPC machines. It is important to stress that RP, and pilot systems in general, do not "game" the scheduling and security policies of the HPC machines: resource acquisition is performed via the batch system and resources remain available only for the given walltime. Jobs wait in the queue alongside all the other jobs and are subjected to queue, allocation and fair-use policies of the system. Pilots are own by the same user that submitted the job to the batch system and RP does not enable pilot multitenancy: only the owner of the job and therefore of the pilot can schedule tasks on that pilot.

Among pilot systems, RP has unique capabilities and architectural properties. RP offers concurrent execution of heterogeneous tasks on the same pilot, supporting both CPU and GPU. This means that tasks with diverse requirements, some using different number of cores and/or work nodes, others using one or more GPUs, can be executed at the same time on the same pilot, without having to be queued on the batch system or the HPC resource. Further, RP supports more than twelve task launching methods—e.g., ssh [22], mpirun, aprun, openmpi [23]—and all the major HPC batch systems—e.g., SLURM [24], PBS [4], or LSF [25]. In this way, RP can enable pilot capabilities on machines with different architectures and software environments.

RP isolates the execution of each tasks into a dedicated process, enabling concurrent execution of heterogeneous tasks by design. For example, RP can run a bag of 65,000 heterogeneous tasks, requiring between 1 and 384 cores, some running on CPU cores, others on GPU, some using OpenMP other MPI. These tasks may run in several 'generations', each with varying degree of concurrency and combination of types of tasks. Further RP can concurrently manage multiple pilots both on the same machine or submitted across a set of machines. Tasks can be late-bound to available resources, using different scheduling algorithms across resources and within each resource.

Architecturally, RP is designed following a so called 'building blocks' approach to be self-sufficient, interoperable, extensible and partially composable [26]. Self-sufficient because RP independently implements the necessary and sufficient set of functionalities for describing and managing pilot and task entities; interoperable in terms of type of workload, resource, and execution paradigm; and extensible as new properties can be added to the pilot, task and resource descriptions, and more functionalities can be implemented for this entities. Currently, composability is partially designed and implemented: while the pilot-API can be used by both users and other systems to describe and execute task-based workloads, RADICAL-Pilot requires RADICAL-SAGA [27–29] to interface to HPC resources. A prototype interface to cloud resources based on LibCloud [30] is available and a general-purpose resource connector component is under development.

Figure 2 shows RP architecture and execution model. RP assumes a multi-task application written in Python, using RP's Pilot API. This API let users to describe tasks—called compute units (CU)—and pilots (Figure 2.1). Each compute unit description has a set of

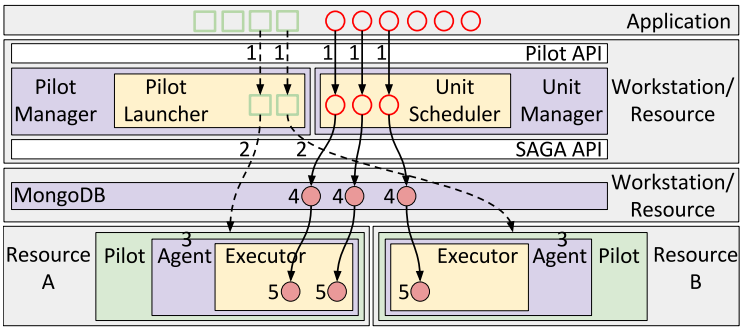

Figure 2: RADICAL-Pilot (RP) architecture and execution model. Architecture: interfaces (white), modules (purple), components (yellow), pilot entity (green), task entity called compute unit (red). Execution model: Pilot and units are described via the Pilot API (1); pilots are submitted to the indicated resource(s) (2); each pilot bootstraps an agent (3); compute units are scheduled onto available agent(s) (4) and executed (5).

properties, including the name of the executable launched by RP, the type and amount of cores this executable will need, whether it requires Message Passing Interface (MPI) [23], input files and so on. Pilot descriptions also have a set of properties, including the endpoint where the resource request should be submitted, the type and amount of resources to acquire, and the walltime of these resources. The RP API includes classes to create both pilot and unit managers to which pilot and unit descriptions are assigned for execution. Once managed, pilots are submitted to the indicated endpoint (Figure 2.2), and once scheduled pilot bootstrap an agent (Figure 2.3). Unit manager(s) schedule compute units onto available agent(s) (Figure 2.4) and each agent uses a resource-specific executor to run the units (Figure 2.5).

Note that compute units are scheduled once the agent is active and that unit descriptions need to be transferred from the client to the agent. This means that the latency of each unit transfer reduces the overall utilization of the available resources as no unit can be executed while in transit. To address this issue, RP transfers units in bulk, reducing the overall latency of unit transfer to a single round trip. Further, units are scheduled via a database instance that needs to be reachable by both the machine on which the unit managers are instantiated, and the remote machine on which the units will have to be executed.

RP supports the execution of units which require staging of input and/or output data. Units' input and output files are managed by stager components not represented in Figure 2. Data staging is performed independently of the pilot lifetime, and can thus overlap with the pilot's queue waiting time, thus reducing the impact on resource utilization.

The next generation executor (NGE) is a REST API that enables running RP as a service. The REST API is a close semantic representation of the underlying native Pilot API. The service manages the lifetime of the RP unit and pilot managers, and of the RP pilot agent, on behalf of a client. As is often the case when translating a module or library API (which is usually invoked in a process-local call stack) into a REST API (which is usually called over a network link), care has to be taken to not let even moderate network latencies impact the overall API performance. The NGE API manages to avoid that problem by adding support for bulk operations for those API methods most prone to latency impacts: unit submission and state updates.

NGE service also provides functionality that extends the scope of the Pilot API: instead of requiring the client to specify pilot size, runtime and configuration, it implements policy

driven automatisms to shape a pilot based on the received workload and on resource availability. NGE submits those auto-shaped pilots to the resource batch queue or it can inspect the target resource's backfill availability, shaping the pilots so that they fit the available backfill constraints.

NGE's pilot shaping capabilities are policy driven: they can be adapted to the configuration of the target resource, but also to the specific requirements of diverse use cases. This makes NGE deployments configurable for specific user groups and projects. In the context of the presented use case, NGE is configured to mimic properties which were available in PanDA's native execution backend, which semantically eases the integration between the two systems.

4 Integrating Harvester and NGE

We deployed Harvester, NGE, RP, and the MongoDB [31] instance needed by RP on three separate containers provided by OLCF via their OpenShift service [32]. The container used for NGE and RP can directly submit jobs to the PBS batch system [4] of Titan, while the MongoDB instance can be reached by both Titan and the NGE and RP container. This enables submission of pilot jobs to Titan and scheduling of tasks on the resources of that pilot once scheduled and bootstrapped.

A separate instance of Harvester has been set up in order to run experiments of PanDA and NGE interaction. Because of differences in the deployment environment, Harvester and NGE instances have been installed on different containers, enabling communication via a secure tunnel. A special module for job submissions to NGE and job monitoring has been developed for Harvester. The functionality of these modules have been tested with dummy jobs as well as with samples of ATLAS and Molecular Dynamics [33] workloads.

Figure 3 shows details of the coordination protocol between Harvester and NGE and of the execution model of the integrated system. RP initiates resource acquisition by submitting a configurable number of pilots to Titan (Figure 3.1). Once one or more pilots become available, RP publishes the aggregate number of cores and their time availability via NGE (Figure 3.2). Note how NGE abstracts the granularity of pilot resources into a resource overlay described by the total core availability, partitioned based on the amount of time cores are available. This partitioning is made necessary by the stacked availability of multiple pilots on the same resource.

Harvester can poll NGE to know how many cores are available and for how long they will remain available (Figure 3.3), and push a number of tasks to NGE for execution (Figure 3.4). Note that before pushing tasks to NGE, Harvester makes sure that the input data required by these tasks are available to the work nodes of Titan. This is done by copying or linking the relevant data to a shared file system, either local to the cluster or available on the OLCF network (Figure 3.5). The number of tasks pushed to NGE is calculated on the base of aggregated task requirements in terms of number of cores and walltime. Tasks are pushed in bulk so to avoid latency and other overheads associated with pushing single tasks. Once tasks are pushed into NGE, these are translated into compute unit descriptions for RP (Figure 3.6) and then executed on the available pilot resources (Figure 3.7-9) as described in Sec. 3. Once executed, RP stages out units' output to a filesystem accessible by Harvester (Figure 3.10) and Harvester collects this files, terminating the execution cycle (Figure 3.11).

NGE can be configured to use RP to submit jobs to Titan in two concurrent or single modes: standard and backfill. Standard mode supports submission of jobs to one or more of the five batch queues available on Titan. In this mode, job submissions are exactly the same as any other job submitted to Titan for execution by any other user. In backfill mode, RP uses the `showbf` [34] command of the Moab scheduler to poll the current node and walltime

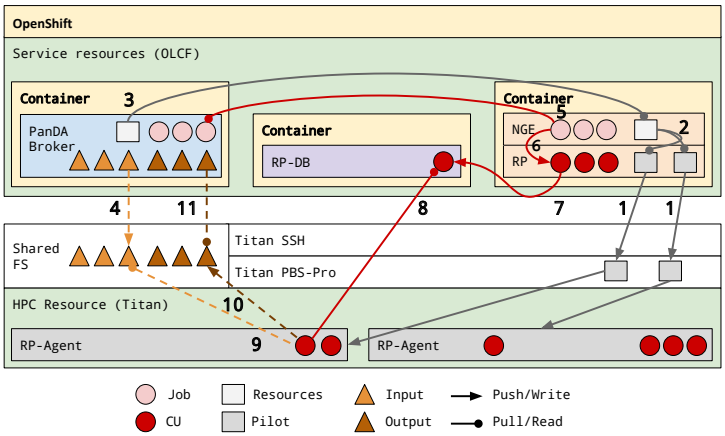

Figure 3: Integration between Harvester and NGE deployed at OLCF to manage the execution of Particle Physics and Molecular Dynamics workloads. Harvester, NGE and RP, and MongoDB are deployed on containers provided and managed via OpenShift.

availability, creates a pilot description requesting that availability and submits an equivalent job to the suitable queue. In this way, RP increases the chances for that job to be immediately scheduled on Titan, reducing the job (and therefore pilot) queue waiting time and operating as Harvester but with full-pilot capabilities. This allows for Harvester to submit multiple generations of tasks for execution on Titan, maintaining a predefined 'pressure' on Titan queues. As a result, the Havester/NGE integrated system execution model and operational modality is consistent across Grid and HPC, reducing the differences in how tasks are mapped between the two types of infrastructure.

5 Conclusion and Next Steps

In this paper we introduced two distinct systems—Harvester and RADICAL-Pilot—and an interface—Next Generation Executor (NGE)—that enabled the design and implementation of a coordination protocol for their integration. The main contribution is to show how systems developed by two independent teams can easily be integrated to provide new functionalities. This suggests that end-to-end models of middleware, where a single ecosystem of modules is designed to provide all the functionalities required to enable distributed computing can be replaced by a model in which systems are independently designed to be integrated. As seen in this paper, designing for integration means developing a well-defined API, making explicit entities and state models. Entities and functionalities must be chosen at the right level of abstraction so to avoid the specificity of single implementations. For example, NGE exposes Task, Core and Walltime entities. These are sufficiently abstracted to be consistent with both systems' design but specific enough to define the domain of workload management for HPC.

Currently, we deployed the integrated system at OLCF on the OpenShift platform and we have started testing and performance characterization. We measured no appreciable overhead when executing a workload via NGE or RP's native API. We have now to compare the execution of a standard ATLAS Geant4 workload with Harvester to the same execution performed with Harverser and NGE. This comparison will measure integration overheads but also workload execution performance. We expect that the execution on pilot will introduce optimizations, especially related to AthenaMP [35] setup and the total number of events

processed per walltime unit. After this characterization, we will be ready to concurrently execute heterogeneous workloads, using the same pilot to run tasks of distinct users from diverse domains. This will make PanDA a general purpose workload manager and will open the possibility to explore new resource utilization patterns on leadership-class machines.

In this context, PanDA will leverage the design of NGE and RADICAL-Pilot to gain access to Summit, the new leadership-class machine managed by OLCF at ORNL. Thanks to its design, RADICAL-Pilot has been ported to Summit in less than two months and because of the isolation between workload management and resource acquisition, no modification of Harvester is required to execute its workloads on Summit. As a matter of fact, Harvester will be able to execute general purpose workloads on both Titan and Summit, at the same time and without any modification to its code.

Following this period of testing, characterization and adoption of a new machine, the system integrating Harvester and NGE will be evaluated for production deployment.

References

[1] T. Maeno, Journal of Physics: Conference Series **119**, 062036 (2008)

[2] T. Maeno, K. De, A. Klimentov, P. Nilsson, D. Oleynik, S. Panitkin, A. Petrosyan, J. Schovancova, A. Vaniachine, T. Wenaus et al., *Evolution of the ATLAS PanDA workload management system for exascale computational science*, in *Journal of Physics: Conference Series* (IOP Publishing, 2014), Vol. 513, p. 032062

[3] M. Turilli, M. Santcroos, S. Jha, *A comprehensive perspective on pilot-job systems* (ACM, 2018), Vol. 51, p. 43

[4] R.L. Henderson, *Job scheduling under the portable batch system*, in *Workshop on Job Scheduling Strategies for Parallel Processing* (Springer, 1995), pp. 279–294

[5] D. Oleynik, S. Panitkin, M. Turilli, A. Angius, S.H. Oral, K. De, A. Klimentov, J.C. Wells, S. Jha, *High-Throughput Computing on High-Performance Platforms: A Case Study*, in *13th IEEE International Conference on e-Science* (2017), pp. 295–304

[6] *Maui scheduler: Backfill*, `http://docs.adaptivecomputing.com/maui/8.2backfill.php`

[7] A. Guide (2011)

[8] S. Agostinelli, J. Allison, K.a. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G.. Barrand et al., Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment **506**, 250 (2003)

[9] *Next generation executor (NGE)*, `https://github.com/radical-cybertools/radical.nge`

[10] R. Battle, E. Benson, Web Semantics: Science, Services and Agents on the World Wide Web **6**, 61 (2008)

[11] *RADICAL-Pilot*, `https://radicalpilot.readthedocs.io/en/latest/`

[12] A. Merzky, M. Turilli, M. Maldonado, S. Jha, CoRR **abs/1801.01843** (2018)

[13] A. Merzky, M. Turilli, M. Maldonado, M. Santcroos, S. Jha, *Using Pilot Systems to Execute Many Task Workloads on Supercomputers*, in *JSSPP 2018 (in conjunction with IPDPS'18)* (2018), pp. 61–82

[14] *Research in advanced distributed cyberinfrastructure and applications laboratory (radical)*, `http://radical.rutgers.edu/`

[15] F. Megino, K. De, A. Klimentov, T. Maeno, P. Nilsson, D. Oleynik, S. Padolski, S. Panitkin, T. Wenaus, Journal of Physics: Conference Series **898**, 052002 (2017)

[16] V. Garonne, R. Vigne, G. Stewart, M. Barisits, M. Lassnig, C. Serfon, L. Goossens, A. Nairz, A. Collaboration et al., *Rucio–The next generation of large scale distributed system for ATLAS Data Management*, in *Journal of Physics: Conference Series* (IOP Publishing, 2014), Vol. 513-4, p. 042021

[17] I. Foster, *Globus Online: Accelerating and democratizing science through cloud-based services* (IEEE, 2011), Vol. 15, pp. 70–73

[18] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G.D. Peterson et al., Computing in Science & Engineering **16**, 62 (2014)

[19] *Cheyenne*, `https://www2.cisl.ucar.edu/resources/computational-systems/cheyenne/cheyenne`

[20] *About blue waters*, `http://www.ncsa.illinois.edu/enabling/bluewaters`

[21] *OLCF resources: Compute systems*, `https://www.olcf.ornl.gov/olcf-resources/compute-systems/`

[22] *The SSH protocol*, `https://www.ssh.com/ssh/#sec-The-SSH-protocol`

[23] W.D. Gropp, W. Gropp, E. Lusk, A. Skjellum, A.D.F.E.E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*, Vol. 1 (MIT press, 1999)

[24] A.B. Yoo, M.A. Jette, M. Grondona, *SLURM: Simple linux utility for resource management*, in *Workshop on Job Scheduling Strategies for Parallel Processing* (Springer, 2003), pp. 44–60

[25] S. Zhou, *LSF: Load sharing in large heterogeneous distributed systems*, in *I Workshop on Cluster Computing* (1992), Vol. 136

[26] M. Turilli, A. Merzky, V. Balasubramanian, S. Jha, *Building Blocks for Workflow System Middleware*, in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (IEEE, 2018), pp. 348–349

[27] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, J. Shalf, Computational Methods in Science and Technology **12**, 7 (2006)

[28] A. Merzky, O. Weidner, S. Jha, SoftwareX **1**, 3 (2015)

[29] *RADICAL-SAGA*, `https://saga-python.readthedocs.io/en/latest/`

[30] *Libcloud*, `https://libcloud.readthedocs.io/en/latest/compute/drivers/ec2.html`

[31] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage* (O'Reilly Media, Inc., 2013)

[32] J. Hines, *OLCF testing new platform for scientific workflows*, `https://www.olcf.ornl.gov/2017/06/05/olcf-testing-new-platform\-for-scientific-workflows/`

[33] B. Brooks, C. Brooks, A. Mackerell et al., Journal of Computational Chemistry **30**, 1545 (2009)

[34] *Running jobs on titan*, `https://www.olcf.ornl.gov/for-users/system-user-guides/titan/running-jobs/`

[35] D. Crooks, P. Calafiura, R. Harrington, M. Jha, T. Maeno, S. Purdie, H. Severini, S. Skipsey, V. Tsulaia, R. Walker et al., Journal of Physics: Conference Series **396**, 032115 (2012)