RADICAL-Pilot and PMIx/PRRTE: Executing heterogeneous workloads at large scale on partitioned HPC resources

Mikhail Titov[1][0000−0003−2357−7382], Matteo Turilli[1,2][0000−0003−0527−1435], Andre Merzky[2][0000−0002−7228−4327], Thomas Naughton[3][0000−0002−3546−2382], Wael Elwasif[3][0000−0003−0554−1036], and Shantenu Jha[1,2][0000−0002−5040−026X]

[1] Brookhaven National Laboratory, Upton, NY 11973, USA
{titov,mturilli,shantenu}@bnl.gov
[2] Rutgers, The State University of New Jersey, Piscataway, NJ 08854, USA
andre@merzky.net
[3] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
{naughtont,elwasifwr}@ornl.gov

Abstract. Execution of heterogeneous workflows on high-performance computing (HPC) platforms present unprecedented resource management and execution coordination challenges for runtime systems. Task heterogeneity increases the complexity of resource and execution management, limiting the scalability and efficiency of workflow execution. Resource partitioning and distribution of tasks execution over portioned resources promises to address those problems but we lack an experimental evaluation of its performance at scale. This paper provides a performance evaluation of the Process Management Interface for Exascale (PMIx) and its reference implementation PRRTE on the leadership-class HPC platform Summit, when integrated into a pilot-based runtime system called RADICAL-Pilot. We partition resources across multiple PRRTE Distributed Virtual Machine (DVM) environments, responsible for launching tasks via the PMIx interface. We experimentally measure the workload execution performance in terms of task scheduling/launching rate and distribution of DVM task placement times, DVM startup and termination overheads on the Summit leadership-class HPC platform. Integrated solution with PMIx/PRRTE enables using an abstracted, standardized set of interfaces for orchestrating the launch process, dynamic process management and monitoring capabilities. It extends scaling capabilities allowing to overcome a limitation of other launching mechanisms (e.g., JSM/LSF). Explored different DVM setup configurations provide insights on DVM performance and a layout to leverage it. Our experimental results show that heterogeneous workload of 65,500 tasks on 2048 nodes, and partitioned across 32 DVMs, runs steady with resource utilization not lower than 52%. While having less concurrently executed tasks resource utilization is able to reach up to 85%, based on results of heterogeneous workload of 8200 tasks on 256 nodes and 2 DVMs.

Keywords: high performance computing · resource management · middleware · runtime system · runtime environment.

1 Introduction

Workflows are increasingly necessary for scientific discovery, and represent a fast growing class of applications [11] that require efficient and effective scalability on computing resources. Workflow-based applications are increasing in number, are often diverse and highly complex. This trend is driven by the coupling of traditional high performance computing (HPC) with new simulation, analysis, and data science approaches. Several workflows of the Exascale Computing Project, the winners and contestants of the special Gordon Bell Prize for COVID-19 research of the last two years, and of multiple INCITE awards exemplify this new reality [14,15,34]: a heterogeneous combination of applications, machine learning models, and "glue" code, running on heterogeneous computers, orchestrated by a scalable workflow system.

Executing workflows on leadership-class HPC platforms at scale poses unprecedented challenges in terms of capability and performance requirements. Departing from traditional monolithic MPI applications, modern workflows applications require executing tens of thousand heterogeneous tasks on heterogeneous computing supports. Tasks may have different for runtime and I/O properties, execute on CPU and/or GPU, utilize MPI across few or large amount of nodes, and run as standalone executables, functions written in diverse languages or services exposing dedicated interfaces. Workflow applications require middleware capable of prioritizing, scheduling, placing and launching heterogeneous workflow tasks across entire HPC platforms while maintaining high resource utilization and low management overheads.

Addressing those challenges requires the development of new middleware components specifically designed for modern HPC platforms. Among these, pilot systems have played a major role in enabling the execution of many tasks applications on HPC resources. By decoupling resource acquisition performance via a single job submission to the HPC platform's batch system, and task scheduling performed via a dedicated scheduler on the acquired resources, pilot systems have made possible to execute hundreds of thousand tasks on resources otherwise designed to execute a single large job. Pilot systems are relying on lower-level middleware to place and launch those tasks across the nodes of the HPC platforms. Often, that middleware is not designed specifically for high-throughput task launching and poses bottlenecks both in terms of performance and reliability. The Process Management Interface for Exascale (PMIx), focused on support exascale environments, provides abstracted and standardized interfaces used as building blocks for the implementation of distributed asynchronous runtimes. PMIx-based Reference RunTime Environment (PRRTE) is a corresponding reference implementation with capabilities to launch and monitor MPI jobs.

In this paper, we offer a performance evaluation of PMIx/PRRTE when used to execute up to 65,600 heterogeneous tasks on up to 2048 compute nodes of Summit, the leadership class HPC platform hosted at the Oak Ridge National Laboratory. We couple our pilot system—RADICAL-Pilot—with PMIx/PRRTE to enable task scheduling, placement and launching. We confirm that RADICAL-Pilot and PMIx/PRRTE can be efficiently and effectively coupled, without in-

troducing mutual bottlenecks. Further, while PMIx/PRRTE was not originally designed to support this use case, we show its ability to scale the provided workload, reaching a peak of concurrently executed tasks up to 25K with resource utilization not lower than 52%. Having workload with smaller number of concurrently executed tasks allows reaching a peak of resource utilization up to 85%.

Our analysis focuses on resource partitioning via multiple PRRTE Distributed Virtual Machines (DVMs). Resource partitioning is fundamental for scalability as the overheads of task placement and launching grows with the number of managed resources per DVM. Without partitioning, the cost of bookkeeping and concurrent communication becomes dominant at petascale [31] and unpractical at the upcoming exascale. By utilizing multiple DVMs, we partition the costs of task placement and launching across multiple concurrent and independent processes, limiting the impact of global overheads. While this approach is generally considered promising, we are still missing a detailed performance analysis on a production leadership class machine and with realistic workload parameters. This paper fills this gap.

In the next section we introduce PMIx, PRRTE, DVM and RADICAL-Pilot, detailing their architectures and integration, and explain how they support the execution of heterogeneous tasks on Summit. In §3 we review related work, outlining the gaps that this paper address in the current state of the art. In §4 we describe our experimental design and setup, show how we have parameterized our workloads to be consistent with the workflow applications that are currently supported on Summit, and discuss the results of our performance evaluation across diverse scales and configurations. Finally, in §5 we summarize the contributions of this paper and suggest some future lines of research supported by our results.

2 Background

2.1 Process Management Interface for Exascale

The Process Management Interface for Exascale (PMIx) [5] is an open source standard that extends the prior PMI v1 & v2 interfaces used to launch tasks on compute resources. PMIx provides a method for tools and applications to interact with system-level resource managers and process launch mechanisms. PMIx provides a bridge between such clients and underlying execution services, e.g., process launch, signaling, event notification. The clients communicate with PMIx enabled servers, which may support different versions of the standard. PMIx can also be used as a coordination and resource discovery mechanism for, e.g., machine topology information. An implementation of the PMIx standard is provided by the OpenPMIx project [4] as a software library that contains the programming interfaces needed to use the standard. The OpenPMIx project also provides a reference implementation of a PMIx enabled runtime: PRRTE.

2.2 PMIx Reference RunTime Environment

A reference implementation of the PMIx server-side capabilities is available via the PMIx Reference RunTime Environment (PRRTE) [16]. PRRTE leverages the modular component architecture (MCA) that was developed for Open MPI [19], which enables execution time customization of its runtime capabilities. The PRRTE implementation provides a portable runtime layer that users can leverage to launch a PMIx server.

PRRTE includes a persistent Distributed Virtual Machine (DVM), which uses system-native launch mechanisms to bootstrap an overlay runtime environment that can then be used to launch tasks via the PMIx interface. This removes the need to bootstrap the runtime layer on each individual task launch. Instead, after the launch of the DVM, a tool connects and sends a request to start a task. The task is processed and then generates a launch message that is sent to the PRRTE daemons. These daemons then launch the task. Internally, this task is referred to as a *PRRTE job*, not to be confused with the batch job managed by the system-wide scheduler. The stages of each PRRTE job are tracked from initialization through completion. DVM is teared down after the user session is completed.

The lifetime of a PRRTE job could be roughly divided into the following stages (marked by internal PRRTE state change events): (i) from `init_complete` to `pending_app_launch` — time to setup the task and prepare launch details; (ii) from `sending_launch_msg` to `running` — time to send the process launch request to PRRTE daemons and to enact them on the target nodes; and (iii) from `running` to `notify_complete` — duration of the application's execution plus time to collect task completion notification. First two stages are usually combined into a generalized metric and we will refer to it as a task setup time, i.e., the time between when the PRRTE job has started and when the job's application payload starts running.

In our experiments, we use multiple DVMs (i.e., multi-DVM) to partition available resources for heterogeneous task execution, measuring task setup time, DVM startup and termination times, and overall resource utilization. Together, our experiments allow to understand how to configure PRRTE and DVMs to support the execution of workloads with heterogeneous tasks at scale on Summit.

2.3 RADICAL-Pilot

RADICAL-Pilot (RP) [23] is a runtime system designed to decouple resource acquisition from task execution. As every pilot system, RP acquires resources by submitting a batch job, then bootstraps dedicated software components on those resources to schedule, place and launch application tasks, independent from the machine batch system [32]. Scheduling, placing and launching capabilities are specific to each HPC platform, which makes supporting diverse platforms with the same code base challenging. RP can execute single or multi core/GPU tasks within a single compute node, or across multiple nodes. RP isolates the execution of each tasks into a dedicated process, enabling concurrent and sequential execution of heterogeneous tasks by design.

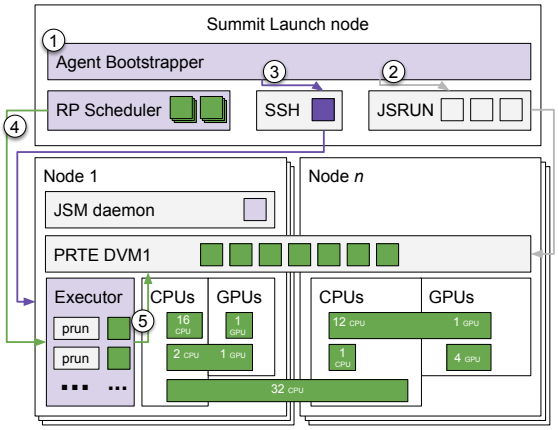

Fig. 1. Deployment of RP on Summit with PRRTE/DVM.

RP is a distributed system designed to instantiate its components across available resources, depending on the platform specifics. Each components can be individually configured so as to enable further tailoring while minimizing code refactoring. RP uses RADICAL-SAGA [24] to support all the major batch systems, including Slurm, PBSPro, Torque and LSF. RP also supports many methods to perform node and core/GPU placement, process pinning and task launching like, for example, `aprun`, `jsrun`, `srun`, `prun` (PRRTE), `mpirun`, `mpiexec` and `ssh`.

RP is composed of two main components: Client and Agent. Client executes on any machine, while Agent bootstraps on one of Summit's batch nodes. Agent is launched by a batch job submitted to Summit's LSF batch system via RADICAL-SAGA. After bootstrapping, Agent pulls bundles of tasks from Client, manages the tasks' data staging if needed, and then schedules tasks for execution via either JSM/LSF or PRRTE/DVM on Summit.

How Agent deploys on Summit depends on several configurable parameters like, for example, number of sub-agents, number of schedulers and executors per sub-agent, method of communication between agent and sub-agents, and method of placing and launching tasks for each executor of each sub-agent. A default deployment of Agent instantiates a single sub-agent, scheduler and executor on a batch node of Summit. In case of JSM/LSF, the executor calls one `jsrun` command for each task, and each `jsrun` uses the JSMD daemon to place and launch the task on work nodes resources (thread, core, GPU).

An architecture block diagram describing the integration between RP and PRRTE (Fig. 1) shows the deployment of RP/PRRTE Agent on a batch node and one sub-agent on a compute node. In this configuration, RP uses SSH to launch sub-agents on compute nodes and then PRRTE/DVM to place and launch tasks across compute nodes. This configuration enables the sub-agent to use more resources and, as shown in the next section, improves scalability and performance

of task execution. Note that, independent from the configuration and methods used, RP can concurrently place and launch different types of tasks that use different amount and types of resources.

The RP resource manager is responsible to collect and manage information about acquired nodes and to start related services if required. In case of PRRTE, RP allows to configure the number of DVMs to be started and resource manager distributes available compute nodes among all that DVMs. The command to start a corresponding PRRTE process is `prte`, which does setup a DVM environment and provides a list of compute nodes, which will be managed by the DVM. In RP, it is configured to set a flat routing tree, i.e., *a high connectivity mode* (all daemons directly connect to the DVM controller), to eliminate relay times in the tree impacting startup time. Such mode is set due to PRRTE uses a single progress thread, so communication competes with mapping and local process start, thus the notion is to take out the time a daemon spent relaying launch messages by having it directly connect to the DVM controller. Related to it, a task placement mechanism uses `prun` command to invoke each application/task as opposed to using `PMIx_Spawn`. DVM controller is responsible for handling `prun` connection requests, doing the initial mapping of each PRRTE job, etc. `prun` command is configured to allow oversubscription (i.e., running more tasks than available slots per node), but RP schedules tasks based on availability of slots without oversubscribing.

There are several tracing events produced by RP for measuring performance characteristics regarding PRRTE/DVM: DVM startup events `dvm_start` and `dvm_ready`, DVM termination event `dvm_stop`, task execution (DVM placement) events `task_exec_start` and `task_exec_stop`, and application running events `app_start` and `app_stop`. Thus, task setup time is measured from the time `prun` call is executed (`task_exec_start`) until the time application starts running (i.e., to the time when process reports in, `app_start`).

3 Related work

Pilot systems like GlideinWMS [28], PanDA [29] and DIRAC [30] are used to implement late binding and multi-level scheduling on a variety of platforms. While these systems have been successfully used on HPC machines [18, 21, 22], including on the former ORNL leadership class machine Titan [25], they are currently not available on Summit and do not support PRRTE.

PRRTE [16] relies on PMIx to place and launch processes on Summit's nodes. Many applications are actively working to directly use PMIx to interface with the system management stack to benefit from portable process and resource management capabilities [33]. While PMIx explicitly supports interfacing with command line tools, there are no other pilot systems using PMIx via PRRTE. MPICH and Hydra [10] offer capabilities similar to PRRTE but are not supported on Summit.

Pilot systems are not the only way to execute many-task applications on HPC machines. JSM and LSF natively support this capability but, as seen in [31], in

their current deployment on Summit they cannot scale beyond 1000 concurrent task executions. Flux [7] is a resource manager that provides users with private schedulers on pools of dedicated resources. This enables the task scheduling capabilities of a pilot system, including RP, but requires to be either adopted as the main job manager of the machine or be deployed as part of a pilot system.

METAQ [13] are a set of shell scripts that forms a "middle layer" between the batch scheduler and the user's computational job scripts and supports task packing. METAQ requires a separate invocation of mpirun (or equivalent) for each task. METAQ has been superseded by `mpi_jm` [12] — a Python library that is linked to applications. In addition to intelligent backfilling and task packing, `mpi_jm` allows the executable to be launched based upon an affinity with the hardware but requires the be coded into task executables and the overall workflow application.

TaskWorks—a task execution engine built using the PMIx interface—is designed as a high-level, light-weight and portable task execution engine for HPC applications [20]. It enables defining tasks and resolving their dependencies within an application, and it supports MPI tasks. PMIx is used as an interface to coordinates thread/task execution packages, such as OpenMP or MPI, and to manage resource usage.

In Ref. [23, 27] we investigated the performance of RP on ORTE—a precursor to PRRTE. Using ORTE, RP was capable of spawning more than 100 tasks/second and the steady-state execution of up to 16K concurrent tasks. Resource utilization was significant lower than with PRRTE and more sensitive to the number of tasks and tasks duration.

There is no other available solution with integration of PMIx/PRRTE using multi-DVM approach. There is an ongoing development effort to introduce resource partitioning in RP, which includes such multi-DVM approach as part of RP scaling capabilities.

4 Experiments

The performance space of RP is vast, including the execution of both homogeneous and heterogeneous tasks with resolved dependencies beforehand at both small and large scales. In Ref. [31] we presented a baseline characterization of executing homogeneous workloads on Summit, comparing the performances of `jsrun` and `prun` (PRRTE). In this paper, we build upon those results, focusing on the execution of heterogeneous workloads consistent with the requirements of the INCITE program [3]. Specifically, we consider two types of task heterogeneity: spatial and temporal. We execute workloads with multi-core tasks of different duration, requiring both CPUs and GPUs, within the boundaries of Summit's compute nodes. Further, we adjust configuration for DVMs setup, significantly extending upon our previous characterization by scaling the concurrent execution of heterogeneous tasks and optimizing baseline performance for real-life workloads.

4.1 Use cases

We consider the use cases from five DOE INCITE allocation awards [3] on Summit. We elicited the computing requirements of their workflows, deriving size and duration of each type of task alongside their I/O requirements. All workflows require to scale on Summit's CPUs and GPUs, executing a variety of workloads with MPI and single/multi-threaded tasks. We focus our experiments on PRRTE/DVM, and on how they support the placement and execution of those tasks at scale. As such, we evaluate the upper and lower boundary requirements of each workload, executing synthetic tasks consistent with those boundaries. Note that for PRRTE/DVM, it makes no different what type of executable each task launches or their I/O requirements, only how many CPUs/GPUs each task requires and for how long.

Future studies will build upon our results to evaluate the actual workflows in collaboration with the domain scientists. Without understanding the scalability of PMIx/PRRTE with heterogeneous workloads, those studies would be premature.

Overall, the workloads of the use cases we considered have the following types of tasks: (i) up to 15 million single core tasks (no GPUs) with runtime from a range 10..60 seconds for high-throughput ensemble docking to identify small molecules (MD docking scans); (ii) an ensemble of about 120 MD simulation tasks using GPUs with runtime of several hours for modeling specific binding regions and understanding mechanistic changes in drug discovery (AI-driven Molecular Dynamics); (iii) one large MPI task with many GPUs over several nodes and many CPU tasks with one core requirement (Earth Sciences domain, PrincetonU); (iv) many OpenMM [17] simulation tasks with one GPU requirement over 1000 nodes (OpenMM Ensemble, Cornell Medical Center); (v) many NAMD [26] simulation tasks with CPUs requirement only, one task per node over 1024 nodes (MDFF Error Analysis, ASU).

The first two use cases among selected are highlighted the most, due to their tasks configurations, which cover all significant heterogeneity characteristics. Both workloads are part of a multi stage campaign for a drug discovery, namely IMPECCABLE (Integrated Modeling PipelinE for COVID Cure by Assessing Better LEads) [9].

Based on the assessment of the considered workloads, we determined the following setup for experiments: (i) we group compute nodes in multiple of 256 for all experiments, and every group of nodes process 8K tasks (weak scaling up to 2048 nodes and 65K tasks); (ii) there are two types of tasks: 90% of all tasks are small and short, which represent the small CPU tasks of the use cases, and 10% of all tasks are large and long, including tasks with GPUs, which represent a dedicated group of larger tasks (e.g., simulation tasks) in the use cases. Important to note that we didn't map MPI task over several nodes into our experiment setup—larger tasks are easier to handle for RP and PRRTE and thus essentially decreasing the load we test the systems under.

4.2 Experiments Design

Our experiments measure the performance of PRRTE/DVM in addition to the performance of RP with PRRTE, when concurrently and sequentially executing workloads with heterogeneous tasks on Summit. Task execution requires assigning suitable resources to the tasks, placing them on resources (i.e., a specific compute node, core, GPU or hardware thread) and then launching the execution of those tasks. RP tracks both tasks and available resources, scheduling the former onto the latter; PRRTE enacts task placement and launching.

Combining experiments with different configurations (different number of DVMs and number of nodes each DVM manages) helps to study how well PRRTE/DVMs perform when managing nodes and tasks execution, as well as the potential interference among DVMs. The number of nodes will vary from 256 up to 2048 nodes on Summit, doubling at every experiment configuration. The experiment with 2048 nodes will demonstrate the performance of RP using multi-DVM approach with configuration setup based on gained knowledge from the previous experiments. Experiments setup parameters are collected in Tab. 2.

Our experiment tasks are self-contained executables, which carry a synthetic payload (calls for environment variables to check the correctness of allocated resources) and imitates task runtime with defined "sleep" time by suspending the calling process. All tasks are heterogeneous, regarding their runtime, type of using resources, CPU and/or GPU, and the amount of resources. For the tasks parameters setup, it is important to note that the number of *slots* per node on Summit [6] with simultaneous multithreading (SMT) level set to 4 is equal to 168 hardware threads (44 physical cores minus 2 reserved cores).

We elicit the task sizes from our use cases. First we define the min and max for every type of task to have a better understanding of their possible layouts on each compute node. Then we estimate a range of values between each min and max, so to guarantee a wide heterogeneity. The size of "small" tasks will be less than 21 slots (1/8th of compute node), while the size of "large" tasks will be in a range of 42..84 slots (from 1/4th up to half of compute node), so as to have two "large" tasks per node on average. Duration for tasks will be generated randomly as well, and "small" tasks will be twice shorter than "large" tasks with ranges 8..10 min and 16..20 min respectively.

Estimation of the number of compute nodes to be filled avoids task execution "tailing", i.e., having a small number of tasks launched after the anticipated termination time of the run (i.e., walltime), due to a not precise scheduling (since RP scheduler processes tasks as soon as a block of tasks arrived and not waiting for all submitted tasks to perform scheduling). Thus we set to use 97% of all allocated resources (i.e., provided slots) to generate task sizes, and considering 3% of resources as supplementary to avoid "tailing".

Refined ranges for task sizes are calculated during the startup of RP application as following: $N_{slots} >= (N_{tasks}/N_{gen}) \times avg(N_{task\_slots})$, where N_{slots} is a number of provided slots, N_{tasks} is a total number of tasks of a particular type, N_{gen} is a number of generations for tasks of a defined type, i.e., an approximate

number of task groups, in which all tasks could be executed concurrently, and N_{task_slots} is a number of slots per task.

For 256 nodes (including 8 "supplementary" nodes) and 8200 tasks, we have 7380 "small" tasks (4 generations) with average N_{task_slots} equals to 5 (slots range in 1..9) and 820 "large" tasks (2 generations) with average N_{task_slots} equals to 76 (slots range in 68..84). Having many "small" tasks will let us to stress the runtime for comparing RTE capabilities.

Experiments approbation As part of the experiments design, we performed an approbation stage, which confirmed the expected behavior of RP components without any overhead, and expected nodes load level. Approbation of designed experiments was made using a Docker container produced by the ExaWorks project's SDK [1, 8]. RP allows to run an application locally with an arbitrary defined resource characteristics, such as number of cores, GPUs, memory, and it also allows to imitate any number of requested nodes (`fake_resources=true`). In case of Summit, resource description includes 168 cores (SMT=4) and 6 GPUs per node. Docker file and resource description are provided in the GitHub repository [2] as part of the experiments setup.

Runs within Docker container also included usage of sub-agents (as described in §2.3) for such components as RP scheduler and executor and running each sub-agent on a dedicated node: one node for scheduler component and three nodes for three instances of executor. Usage of sub-agents on a real resource is necessary, because running all RP components on a batch node and having many tasks $O(10^4)$ hits the limitation of concurrent system processes calls per node.

4.3 Experiments on Summit

We used PRRTE release `2.0a1psrvr-v2.0.0rc1-3912-gff83b55e2e` to conduct experiments on Summit, which is referred as a master release for production. Note that a recommended delay of 10 sec was added after resource allocation and before starting any DVM to ensure that resources were ready to be mapped to DVM(s).

In this section we present key studies, which reveal limitations of examined approach and allow to determine experiment configuration for exercising scaling capabilities. We highlight metrics used to estimate the performance and evaluate results.

PRRTE/DVM size estimation PRRTE management capabilities depend on both the number of managed nodes as well as the number of launched tasks (i.e., the size of assigned workload to be executed). We experimentally determined the maximum number of nodes that can be successfully managed by a single DVM in further experiments. At first we tried to run RP with PRRTE without a payload (we submitted only one task with one core and without any `sleep` time that let RP started without providing any task to the most of acquired compute nodes), thus to confirm that DVM has started and then terminated successfully with

a list of provided nodes. Then we tried the same configurations, but with our designed workload.

Those experiments showed that, without a payload, the maximum number of managed nodes is 512, but, with a payload of 8200 tasks for every 256 nodes (as described in §4.2), the maximum number reduces to 256. Note that we disregard possible cases when the number of nodes is in between 256 and 512, since we double the number of nodes in each experiment, starting from 256, and this paper is not focused on finding an exact maximum number of nodes per DVM.

Beyond that observed numbers, DVM either wasn't able to start successfully (i.e., never reached a confirmation status `DVM ready`) or started having "connection lost" errors regarding communication with its managed nodes. We were not able to localize the cause of such limitation (e.g., high connectivity mode and/or Ethernet problems, socket timeouts, etc.), which is to be investigated, but therefore chose to constrain the following experiments to 256 nodes, and to explore the trade-offs of using multiple parallel DVMs instead of one large DVM.

PRRTE/DVM startup and termination processes The next set of experiments measured the start and termination overheads of one or more DVM, and the possible interference among concurrent DVMs. DVM startup and termination processes depend only on the number of nodes assigned to each DVM, since these processes are responsible for establishing and managing a communication between DVM and PRRTE daemons `prted` on compute nodes.

In this study, we run the three experiments with 256 nodes and 8200 tasks, and different number of DVMs (see Tab. 2). Tab. 1 shows the resulted average startup and termination times for DVMs, and the total overhead (OVH) per experiment (including DVMs OVH). All DVMs started and terminated sequentially, which is done to reduce the network load, i.e., it decreases the possibility of DVMs interference and errors related to losing connection. Increasing the number of DVMs decreases both startup and termination times for each DVM, but increases the total OVH from all DVMs on average. Thus, changing the number of DVMs from 1 to 256 decreases resource utilization (RU) from 85% to 65%. This should be considered, while planning runs with many DVMs for RU improvement.

Table 1. Average DVM startup and termination times per each instance and the total overhead for experiments with 256 nodes.

DVMs	DVM Nodes	Avg DVM Startup (s)	Avg DVM Termination (s)	Total OVH (s)
1	256	6.48	4.63	51.59
2	128	2.66	4.55	51.63
256	1	1.96	1.04	829.57

PRRTE task placement Earlier mentioned three experiments with 256 nodes also showed that decreasing the number of nodes per DVM decreases the task setup time. Fig. 2 shows the impact of the size of payload per DVM on the task setup time. Having less tasks per DVM not just improves the average or median of task setup time, but also shrinks the range of its distribution (especially interquartile range). Description of these experiments along with collected metrics, such as mean, median and third quartile are provided in Tab. 2.

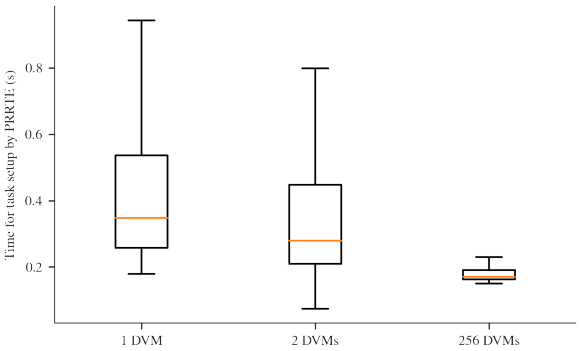

Fig. 2. Distribution of tasks setup time managed by PRRTE for experiments with 256 nodes and 8200 tasks: 1 DVM, which manages 256 nodes; 2 DVMs, each manages 128 nodes; 256 DVMs, each manages 1 node.

With a weak scaling, besides increasing the volume of resources and payload, we increase the number of DVMs per experiment in a way to decrease the load on average per each DVM. The next two experiments are as following: (1) experiment with 512 allocated compute nodes and 2 DVMs, thus each DVM manages 256 nodes; and (2) experiment with 1024 compute nodes and 8 DVMs, thus each DVM manages 128 nodes. Fig. 3 shows task setup times for these experiments, and experiment with smaller number of nodes per DVM has smaller task setup time on average. This observation helps to distinguish a pattern - while scaling a heterogeneous workload for processing, the load per DVM should be lowered.

Due to RP's scheduling algorithm to use all available slots with attempting to assign large tasks first, most of the large tasks were placed during the first half of the experiments run, and most of the small tasks were placed during the second half of the experiments run. This explains the two distinct phases visible in Fig. 3 at the begin and middle of both runs: the first peak(s) is caused by starting all large and a few small tasks, the second, larger peak, is caused by launching all remaining small tasks (see Fig. 6 for a detailed comparison).

Note that experiment with 1024 nodes has some tasks with exceptionally large setup times (up to 60 min), which were never executed due to the walltime limit, and that values fall outside of the third sigma and we don't present them on the plot.

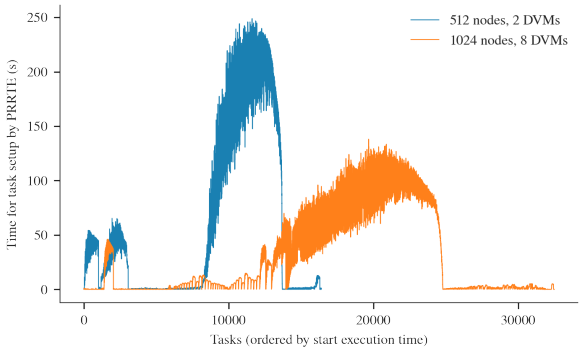

Fig. 3. Setup times per each task for experiments with 512 nodes with 2 DVMs (blue) and 1024 nodes with 8 DVMs (orange).

Study of RP performance RP performance is determined by the task processing rate at every stage of RP's task management. As such, every stage can become a performance bottleneck if the rate of task processing for that stage is lower than the one of the other stages. RP allows to scale each component to improve their performance (e.g., multiple instances of Agent's Scheduler component and use of sub-agents), but has limited options over the execution layer, which depends on third party middleware. When using PRRTE, using multiple DVMs allows to increase task execution rate and match it to the task scheduling rate, avoiding performance bottlenecks. Scheduling rate depends on the number of tasks and the number of nodes, and when using PRRTE as launching method, it is also affected by the number of DVMs. With PRRTE, when a task is assigned to a particular node, the scheduler is also responsible to map it to the DVM, which manages that node. Execution rate consists of RP launching rate and PRRTE/DVM task placement rate, including task setup time and running time.

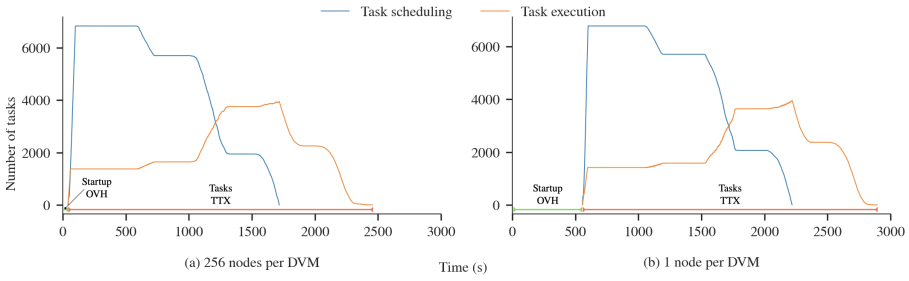

Fig. 4. Tasks concurrency for scheduling and executing processes for experiments with 256 nodes and 8200 tasks.

Experiments with 256 nodes partitioned across 1 and 256 DVMs demonstrate that the increase in amount of DVMs improves the execution rate, since it decreases task setup times, and we got 3% improvement in the total task execution time (TTX). Fig. 4 shows that using the same total number of nodes, but changing the number of DVMs, does not significantly change the scheduling and launching rates (plot slopes on a figure). That indicates the number of DVMs should be estimated with consideration of DVMs OVH and achieved tasks TTX.

RP with PRRTE at scale Based on the results of our previous experiments, we use 32 DVMs for our final experiment to measure weak scaling of PRRTE/DVM with up to 2048 compute nodes (see Tab. 2). We increase the number of nodes and tasks proportionally and, as a consequence, we also increase the number of concurrent system processes calls (e.g., *subprocess.Popen*) in the Agent's Scheduler and Executor components. This creates a bottleneck in those components, which run out of available processes at the operating system level. For such cases, RP allows using sub-agents to run both Scheduler and Executor on dedicated compute nodes (as discussed in §4.2). When using sub-agents, we add 4 extra nodes to the experiment resource allocation. RP does not use these service nodes for tasks execution and, as such, we do not count them as available nodes in our experiments.

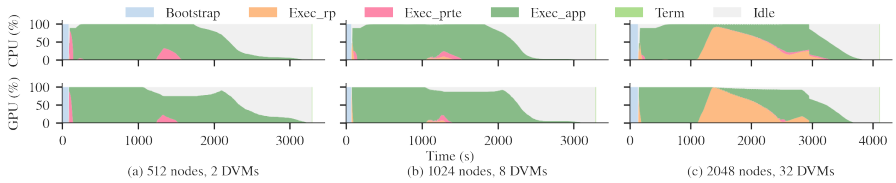

Fig. 5. Resource utilization for experiments with 512 nodes (16,400 tasks), 1024 nodes (32,800 tasks), and 2048 nodes (65,600 tasks), CPU and GPU resources per experiment. The tasks' actual execution is presented as *exec_app*.

As Fig. 5 shows that the resource utilization decreased for experiment with 2048 nodes due to overhead related with RP task preparation stage, which could be caused by shared file system (at this stage RP prepares startup scripts for each task). Our focus in Fig. 5 is on tasks being in stages *exec_prte* (DVM task setup and termination) and *exec_app* (DVM task running). For experiment with 512 nodes, there is a time period when DVM task preparation (setup and termination) slowed down the execution process (i.e., temporary decrease of resource utilization). We assume that such behavior reflects the higher load per each DVM compare to experiments with 1024 and 2048 nodes, where the RP overheads dominate.

Further, we compare the scale of concurrently executed tasks in three DVM stages (setup, running and termination) on Fig. 6. Peak values for the concurrent

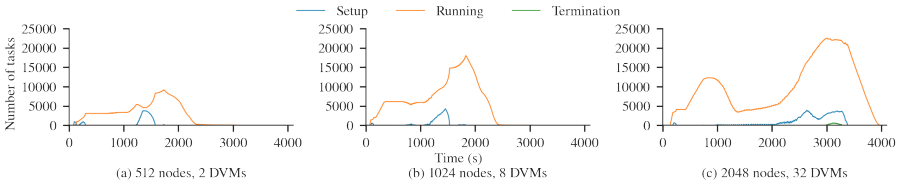

Fig. 6. Tasks concurrency for PRRTE/DVM tasks handling phases (Setup, Running, Termination) for experiments with 512 nodes (16,400 tasks), 1024 nodes (32,800 tasks), and 2048 nodes (65,600 tasks).

number of tasks for experiments with 512, 1024 and 2048 nodes are the following: (i) for the *setup* stage: 3.8K, 4.3K and 3.9K; and (ii) for the *running* stage: 9.1K, 17.9K and 22.5K. The *termination* stage is mostly passed by tasks unnoticed, and only for experiment with 2048 nodes there is one peak of $\sim$ 600 tasks finalizing their state in DVMs during the same time.

For all these cases, RP distributes tasks among nodes not equally, and each DVM processes different amount of tasks (i.e., having some fluctuation in number of tasks per DVM). Further, RP places tasks of different types not evenly in time — most large tasks are scheduled first for better resource utilization. These factors affect the number of concurrently executed tasks, which creates a load on each DVM and DVM task setup time changes accordingly. As mentioned earlier, such designed workload let us to stress the runtime, since RTE is constantly cleaning up completed tasks (*PRRTE jobs*) while trying to start new ones, causing a lot of resource contention within the RTE.

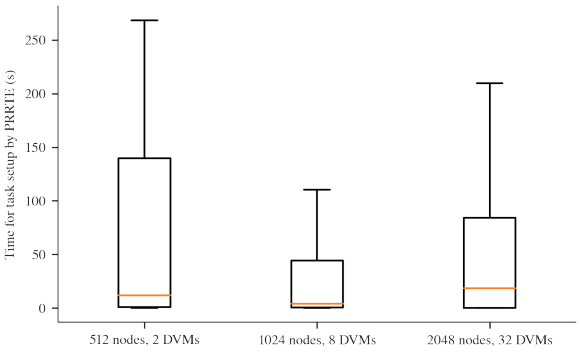

Fig. 7. Distribution of tasks setup time managed by PRRTE for experiments with 512 nodes (16,400 tasks), 1024 nodes (32,800 tasks), and 2048 nodes (65,600 tasks).

Fig. 7 allows to compare the distribution of DVM tasks setup time for experiments with 512, 1024 and 2048 nodes. For experiment with 2048 nodes distribution of task setup times is higher than for experiment with 1024 nodes. This is affected by the high load of DVMs in conjunction with task placement command `prun` as a high overhead method for invoking a PRRTE job — `prun` and the DVM have to execute a multi-step handshake to resolve security and communication protocols. The `prun` connection starts to become significant at these load levels since it flows through a single progress event thread, so each connection has to wait its turn.

Nonetheless, the experiment also shows the possibility to scale the execution of 65,600 heterogeneous tasks on 2048 nodes (+ 4 nodes for sub-agents), which demonstrates a worst-case scenario, considering the type of tasks and their distribution over DVMs.

Table 2. Descriptions and discovered metrics for conducted experiments.

Nodes	DVMs	DVM Nodes	Tasks	Startup OVH (s)	Tasks TTX (s)	Task setup time (s)		
						mean	median	q3
256	1	256	8200	41.18	2417	9.54	0.35	0.54
	2	128		38.02	2414	9.13	0.28	0.45
	256	1		555.39	2340	0.21	0.17	0.19
512	2	256	16,400	64.1 ± 16.4	3208 ± 5	63.5	11.9	139.7
1024	8	128	32,800	69.5 ± 9.2	3169 ± 12	44.3	3.9	44.4
2048	32	64	65,600	129.5 ± 6.6	3823 ± 53	46.7	18.6	83.9

5 Conclusions

We explored the capabilities and limitations of using PMIx/PRRTE as an execution layer within a pilot-based runtime system (RADICAL-Pilot), executing heterogeneous multi-core CPU/GPU tasks on the leadership-class HPC platform Summit. We identified a set of metrics that we used to characterize the performance of PRRTE and its DVM under different configurations and payloads. Our experiments offer a quantitative understanding of the factors that impact PRRTE/DVM performance at scale.

We introduced several use cases with workflows that require the execution of heterogeneous workloads. We used cumulative characteristics of those workloads to build a synthetic workload of 8200 heterogeneous tasks per 256 compute nodes, focused on spatial and temporal heterogeneity. With many small tasks and non-uniform load on DVMs, it lets to stress the RTE.

We found no interference among DVMs while having 256 concurrently running DVMs and determined that DVM could have a limitation on the number

of nodes it can manage in conjunction with the placed payload. In case of our synthetic payload, each DVM was constrained by 256 nodes.

Examination of introduced DVM overhead, as one of the important characteristics, showed that, while changing number of DVMs from 1 to 256 (maximum tested value; DVMs were started and terminated sequentially) for 256 total allocated compute nodes, the RP total overhead has increased by 16 times (from $52s$ to $830s$), even though individual DVM overhead was decreased (from $11s$ to $3s$). This affected resource utilization, which was dropped from 85% down to 65%.

The main focus of our experiments is DVM performance evaluation, thus we conducted analysis of DVM task launching process, and particularly DVM task setup time. We investigated the case of changing only the number of DVMs for 256 total allocated nodes, which showed that it is possible to reach only 3% improvement in task TTX (increased OVH is mentioned earlier). Thus, the DVM configuration assumes to have minimal number of DVMs, which will keep OVH low, but allow to partition the payload. The case of weak scaling in amount of resources, payload and DVMs (experiments with 256/8200 and 512/16,400 nodes/tasks, and each DVM manages 256 compute nodes) leads to increase in the average DVM task setup time.

Also, weak scaling experiments showed that increasing the number of allocated compute nodes from 512 to 2048 (executing from 8200 up to 65,600 tasks respectively) requires to have at least 2^2 times more DVMs every time that the total number of compute nodes is doubled. Having a small number of concurrently executed tasks, up to 5000 tasks (experiments with 256 nodes), allowed to have $\sim$ 85% resource utilization, but with up to 25,000 tasks concurrently executed (experiment with 2048 nodes), resource utilization dropped down to $\sim$ 52%. Increased number of concurrently executed tasks affected RP overhead and along with increased DVM load has affected tasks TTX, which was increased from its minimum $2340s$ (experiment with 256 nodes and 256 DVMs) to the maximum $\sim 3823s$ (experiment with 2048 nodes and 32 DVMs).

This approach is not applicable for workloads with many one-core or small multi-core non-GPU tasks, since it will bring a large overhead and could cause execution process being unstable. Thus, for example, such workload as MD docking scans from IMPECCABLE will not benefit from it, workloads with larger tasks will.

This study gives an understanding of the PMIx/PRRTE scalability capabilities with heterogeneous workloads, and highlights corresponding characteristics. Better control over a certain processes will let to redistribute DVMs load, which will help to increase the overall performance.

Observed behaviour of PMIx/PRRTE tools and collected data can be used by the PMIx community for future development, particularly considering breaking down long times for DVM task setup and `prun` connecting states (e.g., allocation, mapping, launching, etc.) for insights, which would assist in improving the total launch time.

6 Acknowledgments

We would like to thank other members of the PMIx community, and Ralph Castain in particular, for the excellent work that we build upon. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work is also supported by the ExaWorks project (part of the Exascale Computing Project (ECP)) under DOE Contract No. DE-SC0012704 and by the DOE HEP Center for Computational Excellence at Brookhaven National Laboratory under B&R KA2401045. We also acknowledge DOE INCITE awards for allocations on Summit.

References

1. Exaworks: Software development kit, https://github.com/ExaWorks/SDK. Accessed: 2022-02-10.
2. Github repository with experiments data, https://github.com/radical-experiments/summit_prrte_multi_dvm.
3. INCITE innovative and novel computational impact on theory and experiment program, https://www.doeleadershipcomputing.org. Accessed: 2022-02-10.
4. OpenPMIx, reference implementation of the process management interface exascale (PMIx) standard, https://openpmix.github.io. Accessed: 2022-02-10.
5. Process management interface for exascale (PMIx) standard, https://pmix.github.io/pmix-standard/. Accessed: 2022-02-10.
6. User guide for leadership-class supercomputer summit at ornl oak ridge leadership computing facility, https://docs.olcf.ornl.gov/systems/summit_user_guide.html. Accessed: 2022-02-10.
7. Ahn, D.H., Bass, N., Chu, A., Garlick, J., Grondona, M., Herbein, S., Ingólfsson, H.I., Koning, J., Patki, T., Scogland, T.R., Springmeyer, B., Taufer, M.: Flux: Overcoming scheduling challenges for exascale workflows. Future Generation Computer Systems **110**, 202–213 (2020). https://doi.org/10.1016/j.future.2020.04.006
8. Al-Saadi, A., Ahn, D.H., Babuji, Y., Chard, K., Corbett, J., Hategan, M., Herbein, S., Jha, S., Laney, D., Merzky, A., Munson, T., Salim, M., Titov, M., Turilli, M., Wozniak, J.M.: Exaworks: Workflows for exascale. 16th Workshop on Workflows in Support of Large-Scale Science. SC (2021), https://arxiv.org/abs/2108.13521
9. Al-Saadi, A., Alfe, D., Babuji, Y., Bhati, A., Blaiszik, B., Brace, A., Brettin, T., Chard, K., Chard, R., Clyde, A., Coveney, P., Foster, I., Gibbs, T., Jha, S., Keipert, K., Kranzlmüller, D., Kurth, T., Lee, H., Li, Z., Ma, H., Mathias, G., Merzky, A., Partin, A., Ramanathan, A., Shah, A., Stern, A., Stevens, R., Tan, L., Titov, M., Trifan, A., Tsaris, A., Turilli, M., Van Dam, H., Wan, S., Wifling, D., Yin, J.: IMPECCABLE: Integrated modeling pipeline for covid cure by assessing better leads. In: 50th International Conference on Parallel Processing. ICPP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3472456.3473524
10. Balaji, P., Bland, W., Gropp, W., Latham, R., Lu, H., Pena, A.J., Raffenetti, K., Seo, S., Thakur, R., Zhang, J.: Mpich user's guide. Argonne National Laboratory (2014)

11. Ben-Nun, T., Gamblin, T., Hollman, D., Krishnan, H., Newburn, C.J.: Workflows are the new applications: Challenges in performance, portability, and productivity. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 57–69. IEEE (2020)
12. Berkowitz, E., Jansen, G., McElvain, K., Walker-Loud, A.: Job management with mpi_jm. In: International Conference on High Performance Computing. pp. 432–439. Springer (2018)
13. Berkowitz, E., Jansen, G.R., McElvain, K., Walker-Loud, A.: Job management and task bundling. EPJ Web Conf. **175**, 09007 (2018). https://doi.org/10.1051/epjconf/201817509007
14. Bhatia, H., Di Natale, F., Moon, J.Y., Zhang, X., Chavez, J.R., Aydin, F., Stanley, C., Oppelstrup, T., Neale, C., Schumacher, S.K., Ahn, D.H., Herbein, S., Carpenter, T.S., Gnanakaran, S., Bremer, P.T., Glosli, J.N., Lightstone, F.C., Ingólfsson, H.I.: Generalizable coordination of large multiscale workflows: Challenges and learnings at scale. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458817.3476210
15. Casalino, L., Dommer, A.C., Gaieb, Z., Barros, E.P., Sztain, T., Ahn, S.H., Trifan, A., Brace, A., Bogetti, A.T., Clyde, A., et al.: Ai-driven multiscale simulations illuminate mechanisms of sars-cov-2 spike dynamics. The International Journal of High Performance Computing Applications **35**(5), 432–451 (2021)
16. Castain, R.H., Hursey, J., Bouteiller, A., Solt, D.: PMIx: process management for exascale environments. Parallel Computing **79**, 9–29 (2018)
17. Eastman, P., Swails, J., Chodera, J.D., McGibbon, R.T., Zhao, Y., Beauchamp, K.A., Wang, L.P., Simmonett, A.C., Harrigan, M.P., Stern, C.D., Wiewiora, R.P., Brooks, B.R., Pande, V.S.: OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. PLOS Computational Biology **13**(7), 1–17 (07 2017). https://doi.org/10.1371/journal.pcbi.1005659
18. Fifield, T., Carmona, A., Casajús, A., Graciani, R., Sevior, M.: Integration of cloud, grid and local cluster resources with dirac. In: Journal of Physics: Conference Series. vol. 331, p. 062009. IOP Publishing (2011)
19. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting. pp. 97–104. Budapest, Hungary (September 2004)
20. Hou, K., Koziol, Q., Byna, S.: Taskworks: A task engine for empowering asynchronous operations in hpc applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (2020)
21. Hufnagel, D.: Cms use of allocation based hpc resources. In: J. Phys. Conf. Ser. vol. 898, p. 092050 (2017)
22. Maeno, T., et al: Evolution of the ATLAS PanDA workload management system for exascale computational science. In: Proceedings of the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013), Journal of Physics: Conference Series. vol. 513(3), p. 032062. IOP Publishing (2014)
23. Merzky, A., Turilli, M., Maldonado, M., Santcroos, M., Jha, S.: Using pilot systems to execute many task workloads on supercomputers. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 61–82. Springer (2018)

24. Merzky, A., Weidner, O., Jha, S.: SAGA: A standardized access layer to heterogeneous distributed computing infrastructure. Software-X (2015), http://dx.doi.org/10.1016/j.softx.2015.03.001
25. Oleynik, D., Panitkin, S., Turilli, M., Angius, A., Oral, S., De, K., Klimentov, A., Wells, J.C., Jha, S.: High-throughput computing on high-performance platforms: A case study. In: 2017 IEEE 13th International Conference on e-Science (e-Science). pp. 295–304. IEEE (2017)
26. Phillips, J.C., Hardy, D.J., Maia, J.D.C., Stone, J.E., Ribeiro, J.V., Bernardi, R.C., Buch, R., Fiorin, G., Hénin, J., Jiang, W., McGreevy, R., Melo, M.C.R., Radak, B.K., Skeel, R.D., Singharoy, A., Wang, Y., Roux, B., Aksimentiev, A., Luthey-Schulten, Z., Kalé, L.V., Schulten, K., Chipot, C., Tajkhorshid, E.: Scalable molecular dynamics on cpu and gpu architectures with NAMD. The Journal of Chemical Physics **153**(4), 044130 (2020). https://doi.org/10.1063/5.0014475
27. Santcroos, M., Castain, R., Merzky, A., Bethune, I., Jha, S.: Executing dynamic heterogeneous workloads on blue waters with radical-pilot. Cray User Group **2016** (2016)
28. Sfiligoi, I.: glideinWMS—a generic pilot-based workload management system. In: Proceedings of the international conference on computing in high energy and nuclear physics (CHEP2007), Journal of Physics: Conference Series. vol. 119(6), p. 062044. IOP Publishing (2008)
29. Svirin, P., De, K., Forti, A., Klimentov, A., Larsen, R., Love, P., Maeno, T., Mashinistov, R., Mukherjee, S., Nomerotski, A., Oleynik, D., Panitkin, S., Park, H.Y., Sheldon, E., Slosar, A., Wells, J., Wenaus, T.: BigPanDA: Panda workload management system and its applications beyond ATLAS. EPJ Web Conf. **214**, 03050 (2019). https://doi.org/10.1051/epjconf/201921403050
30. Tsaregorodtsev, A., Garonne, V., Stokes-Rees, I.: DIRAC: A scalable lightweight architecture for high throughput computing. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing. pp. 19–25 (2004)
31. Turilli, M., Merzky, A., Naughton, T.J., Elwasif, W., Jha, S.: Characterizing the performance of executing many-tasks on summit. In: IPDRM 2019 (Oct 2019)
32. Turilli, M., Santcroos, M., Jha, S.: A comprehensive perspective on pilot-job systems. ACM Computing Surveys (CSUR) **51**(2), 43 (2018)
33. Vallée, G.R., Bernholdt, D.: Improving support of MPI+OpenMP applications. In: Proceedings of the EuroMPI 2018 Conference (2018)
34. Ward, L., Sivaraman, G., Pauloski, J.G., Babuji, Y., Chard, R., Dandu, N., Redfern, P.C., Assary, R.S., Chard, K., Curtiss, L.A., et al.: Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing. In: 2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC). pp. 9–20. IEEE (2021)