Using Pilot Systems to Execute Many Task Workloads on Supercomputers

Andre Merzky[1], Matteo Turilli[1], Manuel Maldonado[1], Mark Santcroos[1], and Shantenu Jha[1,2]

[1]RADICAL Laboratory, Electric and Computer Engineering, Rutgers University, New Brunswick, NJ, USA
[2]Brookhaven National Laboratory, Upton, NY

Abstract. High-performance computing systems have historically been designed to support applications comprised of mostly monolithic, single-job workloads. Pilot systems decouple workload specification, resource selection, and task execution via job placeholders and late-binding. Pilot systems help to satisfy the resource requirements of workloads comprised of multiple tasks. RADICAL-Pilot (RP) is a modular and extensible Python-based pilot system. In this paper we describe RP's design, architecture and implementation, and characterize its performance. RP is capable of spawning more than 100 tasks/second and the steady-state execution of up to 16K concurrent tasks. RP can be used stand-alone, as well as integrated with other application-level tools as a runtime system.

1 Introduction

Traditionally, advances in high-performance scientific computing have focused on the scale, performance and optimization of a workload with a large but single task, and less on workloads comprised of multiple tasks. High-performance workflows and scalable computation of ensemble workloads are becoming increasingly important and are highly relevant to exploit post-Moore parallelism. As a result, the number and type of applications that can be formulated as workflows or ensembles is vast and span many scientific domains.

Applications with workloads comprised of multiple tasks impose sophisticated execution and advanced resource management requirements [1]. High-performance computing (HPC) systems have been designed to support applications comprised of mostly monolithic, single-job workloads. For example, HPC systems have been designed and operated to maximize overall system utilization, which typically entails static resource partitioning across jobs and users. Thus, there is a tension between the resource requirements of workloads comprised of many tasks, and the capabilities of the traditional HPC resource management as well as their usage policies. This tension motivates middleware that can efficiently manage the ability to support the resource requirements of many task workloads without compromising traditional capabilities of HPC systems.

Enter pilot systems. Ref. [2] defined the properties of the Pilot paradigm, and its relevance in the execution of workloads comprised of multiple tasks. A defining

element of the Pilot paradigm is the execution of a workload via multi-entity and multi-stage scheduling on resource placeholders. Systems implementing the Pilot paradigm submit job placeholders (i.e., pilots) to the scheduler of resources. Once active, each pilot accepts and executes tasks directly submitted to it by the application. In this way, pilot systems decouple workload specification, resource selection, and task execution via job placeholders and late-binding.

Pilot systems address two apparently contradictory requirements: accessing HPC resources via their centralized schedulers, and letting applications independently schedule tasks on the acquired portion of resources. Thus, pilot systems provides a simple solution to the rigid resource management model historically found in HPC systems. Not surprisingly, many workflow management systems use pilot systems. Surprisingly, in spite of the acceptance and uptake of pilot systems, to the best of our knowledge, there are no general purpose implementations capable of working in production with multiple HPC resources, including leadership class machines.

In this paper, we discuss the design, architecture and implementation of RADICAL-Pilot (RP) (§3). RP is a pilot system that fully implements the concepts and capabilities of the Pilot paradigm. The implementation of RP differs from other pilot systems mostly in terms of API, portability, and introspection. Implemented in Python, RP is a self-contained pilot system which can be used to provide a runtime system for workloads comprised of multiple tasks. In §4, we discuss how RP provides pilot capabilities on Cray systems such as *Blue Waters* and *Titan*. We experimentally characterize RP at multiple levels in §5: we study the performance of individual components of RP, followed by the integrated performance of it's Agent. We then investigate the resource utilization and performance of both the native and enhanced scheduling algorithms.

The absolute performance of the enhanced scheduler is less important than the ability to enhance performance of the scheduler via extensions and customized scheduling algorithms. This reiterates the core contribution of this paper: a careful description of the design and implementation of RP, highlighting its use of multi-level and multi-entity scheduling.

2 Related Work

Traditionally, HPC systems such as Crays have been designed to best support monolithic workloads. However, the workload of many important scientific applications is constructed out of spatially and temporally heterogeneous tasks that are often dynamically inter-related [3–5]. These workloads can benefit from being executed at scale on supercomputers (e.g., *Blue Waters* and *Titan*, both Cray systems), but a tension exists among the workloads' resource utilization requirements, the capabilities of the HPC system software, and their usage policies. Pilot systems have the potential to relieve this tension but their adoption for this class of HPC systems present several challenges that, so far, have not been fully addressed.

Since 1995, more than twenty pilot systems have been developed [2]. Most of these systems are tailored to specific workloads, resources, interfaces, or development models. Most pilot systems have been implemented to optimize the throughput of single-core (or single-node), short-lived, uncoupled tasks [2]. Some notable examples are: HTCondor with Glidein on OSG [6], one of the most widely used pilot systems for the execution of mostly single-core workloads; the pilot systems developed for the LHC communities which execute millions of jobs a week [7] and are specialized in supporting LHC workloads on specific resources like those of WLCG; the light-weight execution framework called Falkon, which represents an early stand-alone pilot system for HPC environment [8]; and Coasters, developed mostly to support the Swift workflow system [9].

One of the major challenges in developing a general-purpose pilot system, capable of executing multi-task workloads on supercomputers, is supporting multiple task launch methods, each with a specific set of limitations. For example, Cluster Compatibility Mode (CCM) [10] is designed to provide services analogous to those of Beowulf clusters but is not generally available on all Cray installations and, when present, access to it varies per system. The Application Level Placement Scheduler (ALPS) [11] system, provides launch functionality for running executables on compute nodes but limits the number of concurrent applications a user can run by default. The Open Run-Time Environment [12], a component of the OpenMPI MPI implementation, supports distributed high-performance computing applications operating in a heterogeneous environment but the degree of adoption and support varies across Cray systems.

Tools have been developed to support spatially and/or temporally heterogeneous tasks on Crays but many of these tools are built on top of CCM, ALPS, or use single MPI allocations. As such, they are not able to support task heterogeneity or reach the necessary level of execution concurrency. For example, TaskFarmer [13], a tool developed at LBNL, enables the user to execute a list of system commands from a task file, allowing single-core or single-node tasks to be run within a single `mpirun` allocation. Wraprun [14], a utility developed at ORNL, enables independent execution of multiple MPI applications under a single `aprun` call. QDO [15], a lightweight high-throughput queuing system for workflows that have many small tasks has to use the resource batch system for job submission. MySGE [16], another tool developed at LBNL that allows users to create a private Sun GridEngine cluster on large parallel systems, but is only available on NERSC resources. Python Task Farm (PTF) [17], a utility for running serial Python programs as multiple independent copies of a program over many cores, is available only on ARCHER (at EPCC).

The Pilot paradigm has proven sufficiently useful that resource management systems have begun to include pilot capabilities either as separate tooling, or as part of their implementation. For example, Flux [18] is described as a next-generation Scalable Resource and Job Management Software (RJMS) for HPC centers that focuses on a new paradigm of resource and job management. Within this new paradigm, Flux allows resource allocation to be dynamic (i.e., dynamic workloads), a key design principle of the Pilot paradigm [2]. This results in

jobs having the ability to scale up to a maximum requested resources (e.g., CPU cores, GPUs, etc.) during execution, or to execute workloads (i.e., workloads with different resource requirements) on a single "dynamic" allocation. Unfortunately, Flux is limited only to the HPC resources that use it as their RJMS. Further, as of the writing of this paper, Flux is still on an Alpha release.

3 RADICAL-Pilot

RADICAL-Pilot (RP) is a scalable and interoperable pilot system that implements the Pilot abstraction to support the execution of diverse workloads. We describe the design and architecture of RP, and characterize the performance of RP's task execution components. These components are engineered for efficient resource utilization while maintaining the full generality of the Pilot abstraction. RP supports several Cray machines, including *Blue Waters* (NCSA), *Titan* (ORNL), and ARCHER (EPSRC), and a whole range of other platforms.

3.1 Overall Architecture

RP is a runtime system designed to execute heterogeneous and dynamic workloads on multiple and diverse resources. Workloads and pilots are described via the Pilot-API and passed to the RP runtime system, which launches the pilots and executes the tasks of the workload on them. Internally, RP represents pilots as aggregates of resources independent from the architecture and topology of the target machines, and workloads as a set of units to be executed on the resources of the pilot. Both pilots and units are stateful entities, each with a well-defined state model and life cycle. Their states and state transitions are managed via the three modules of the RP architecture: PilotManager, UnitManager, and Agent (Fig. 1).

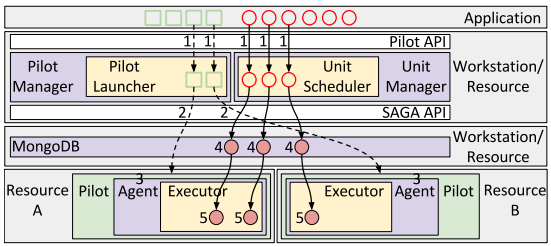


Fig. 1: RADICAL-Pilot Architecture and execution model.

The PilotManager launches pilots on resources via the RADICAL-SAGA API [19]. The SAGA API implements an adapter for each type of supported resource, exposing uniform methods for job and data management. The UnitManager schedules units to pilots for execution. A MongoDB database is used to communicate the scheduled workload between the UnitManager and one or more Agents. For this reason, the database instance needs to be accessible both from the user's workstation and the target resources. Each Agent bootstraps on a remote resource, pulls units from the MongoDB instance, and manages their execution on the cores held by the pilot.

The modules of RP are distributed between the user workstation and the target resources. The PilotManager and UnitManager are executed on the user workstation while the Agent runs on the target resources. RP requires Linux or OS X with Python 2.7 on the workstation but the Agent executes different types of units on resources with very diverse architectures and software environments.

3.2 Programming Model

RP is engineered as a Python library that enables the declarative definition of resource requirements, and of workloads to execute on them. While the Pilot-API is a well-defined interface, the application-specific relationships between resources and workload can be programmed in generic Python. In the following code snippets, we walk the reader through a minimal but complete example of running a workload on *Blue Waters* using RP.

In Listing 2(a), we show the code used to declare the respective managers for pilots and units, whose lifetime is managed by a session object. As such, closing a session destroys all its managers.

In Listing 2(b), we declare a pilot (`rp.ComputePilotDescription()`) by specifying the resource on which it should be instantiated, how many cores it should use, its runtime and, optionally, to what queue it should be submitted and to what project it should be charged. Once submitted via the PilotManager (`pmgr.submit_pilots()`), the pilot is asynchronously queued to the batch system of the indicated resource. Finally, the pilot is associated with a UnitManager (`umgr.add_pilots()`) to enable the execution of units on that pilot.

Finally, in Listing 3 we declare a workload by creating a set of compute units (`cuds`) that specify what payload should be run (`/bin/date`). Once created, the compute units are submitted to the UnitManager (`umgr.submit_units()`) which schedules the unit to a pilot. Once that pilot becomes active, it pulls the scheduled units for execution. The `umgr.wait_units()` call blocks until

```
# Create a session.
session = rp.Session()

# create a pilot manager.
pmgr = rp.PilotManager(session)

# create a unit manager.
umgr = rp.UnitManager(session)
```

(a)

```
# Declare a 64-core pilot that will
# be available for 10 minutes.
pdesc = rp.ComputePilotDescription({
        'resource' : ncsa.bw,
        'cores'    : 64,
        'runtime'  : 10,
        'queue'    : 'debug',
        'project'  : 'gkd'
    })

# Submit the pilot for launching.
pilot = pmgr.submit_pilots(pdesc)

# Make the pilot resources available
# to a unit manager.
umgr.add_pilots(pilot)
```

(b)

Fig. 2: (a) Pilot API: Declaration of PilotManager and UnitManager within a Session. (b) Pilot API: declaration of a pilot, its subsequent submission to the PilotManager and the association to a UnitManager.

all the units have run to completion. Upon its return, the session is closed (`session.close()`) indicating that the workload execution has completed.

3.3 State and Execution Models

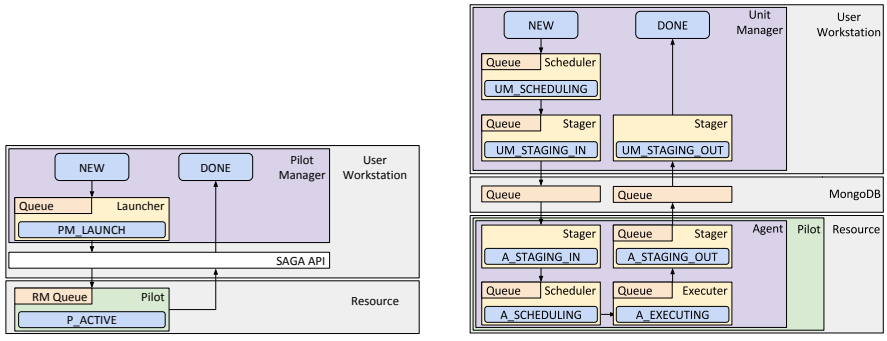
The lifespan of pilots has 4 states distributed among the PilotManager, resource, and pilot instance (Fig. 4(a)). Pilots are instantiated in the state `NEW` by the PilotManager, wait in a queue to be launched, and transition to `PM_LAUNCH` when submitted to a Resource Manager (RM) via the SAGA API. Pilots wait in the queue of the RM and, once scheduled, become `P_ACTIVE`. They remain in this state until the end of their lifetime, when they transition to `DONE`.

```
# Number of units to run.
cuds = []
for i in range(0,42):
    # create a new CU description,
    # and fill it.
    cud = rp.ComputeUnitDescription()
    cud.executable = '/bin/date'
    cuds.append(cud)

# Submit units.
umgr.submit_units(cuds)

# Wait for the completion of units.
umgr.wait_units()

# Tear down pilots and managers.
session.close()
```

Fig. 3: Pilot API: declaration and submission of compute units (CU).

The unit state model has 9 states distributed across the UnitManager, MongoDB instance, and Agent (Fig. 4(b)). Instantiated in the state `NEW` by the UnitManager, every unit is scheduled on an Agent (`UM_SCHEDULING`) via a queue on a MongoDB instance. The unit is then scheduled on the required number of cores held by the Agent's pilot (`A_SCHEDULING`), and finally executed (`A_EXECUTING`).

When required by a unit, input data are staged in by the UnitManager and Agent (`UM_STAGING_IN`, `A_STAGING_IN`), and output data are staged out (`A_STAGING_OUT`, `U_STAGING_OUT`) to a specified destination, e.g., local/shared filesystem or user workstation. Both input and output staging are optional, depending on the requirements of the units. The actual file transfers are enacted via local OS commands or RADICAL-SAGA, supporting (gsi)-scp, (gsi)-(s)ftp, and Globus Online.



(a) (b)

Fig. 4: (a) Architecture of RP Client and pilot state model. (b) Architecture of RP Agent and unit state model.

The state transitions of Fig. 4 are sequential, and every transition can fail or be canceled by the PilotManager or UnitManager. All state transitions are managed by the PilotManager, UnitManager, and Agent components. The only special case is the transition of the pilots to the state `P_ACTIVE` which is determined by the resource's RM and managed by the PilotManager.

3.4 Agent Architecture

Depending on the resource architecture, the Agent's Stager, Scheduler, and Executer components (Fig. 4(b)) can be placed on cluster head nodes, MOM nodes, compute nodes, virtual machines, or any combination thereof. Multiple Stager and Executer components can coexist in a single Agent, placed on any service node or compute node of the pilot's resource assignment. ZeroMQ communication bridges connect the Agent components, creating a network to support the transition of units through components. Every unit goes through the states of Input Staging, Scheduling, Execution & Output Staging. This paper investigates different implementations of launch methods, which are part of the Executer component, responsible for defining and managing the task execution process.

4 Enabling RP on Cray systems

As described in [20], we developed four ways of interfacing RP with the Cray system software to enable execution of distributed applications on Cray systems.

4.1 Application Level Placement Scheduler (ALPS)

The ALPS software suite provides launch functionality for running executables on compute nodes of a Cray system, by interfacing with the `aprun` command. ALPS is the native way to run applications on a Cray from the batch scheduling system. By default, ALPS limits the user to run up to 1,000 applications concurrently within one batch job but in the pilot use-case, these applications may run only for a very short time. This strains ALPS and the MOM node, effectively limiting the throughput of concurrent executions to around 100 applications. Further, ALPS does not allow the user to easily run more than one task on a single compute node, making it difficult, if not impossible, to run workloads with tasks requiring single or small amount of cores and workloads with heterogeneous task size.

4.2 Cluster Compatibility Mode (CCM)

Crays are massively parallel processing (MPP) machines and the Cray Compute Node OS does not provide the full set of Linux services typically found on Beowulf clusters. CCM is a software suite designed to reduce this gap by providing services analogous to those of Beowulf clusters when required by applications. Nonetheless, CCM is not generally available on all Cray installations

and, when present, access to CCM varies per system, requiring special flags to the job description or submitting to a special queue.

RP hides the CCM deployment differences from the application by operating the Agent either externally or internally to the CCM cluster created when submitting a job to the Cray machine. When the Agent runs outside the CCM cluster, it uses `ccmrun` to start tasks. However, this approach still relies on ALPS, inheriting all the limitations described above. When the Agent runs within the CCM cluster, only the initial startup of the Agent relies on ALPS. After that, all task launching is done within the cluster, e.g., by using SSH or MPIRUN, without further interaction with ALPS.

4.3 Open Run-Time Environment (OpenRTE/ORTE)

The Open Run-Time Environment is a spin-off from the Open-MPI project and is a critical component of the OpenMPI MPI implementation. It was developed to support distributed high-performance computing applications operating in a heterogeneous environment. The system transparently provides support for interprocess communication, resource discovery and allocation, and process launch across a variety of platforms. ORTE provides a mechanism similar to the Pilot concept—it allows the user to create a "dynamic virtual machine" (DVM) that spans multiple nodes. In regular OpenMPI usage the lifetime of the DVM is that of the application, but the DVM can also be made persistent, and we rely on this particular feature for RP. RP supports two different modes for interacting with the ORTE DVM: via orte-submit CLI calls, and via ORTE library calls. Currently we can not run applications that are linked against the Cray MPI libraries, but once Cray moves to PMIx[21] that issue is resolved.

Fig. 5 shows the layout of the RP agent, the ORTE Head Node Process that manages the DVM on the MOM Node, and the ORTE Daemons that run on the Compute Nodes.

Command Line Interface (CLI) Recently, ORTE has been extended with tools to expose the creation of the persistent DVM (`orte-dvm`) and the launching of tasks onto that DVM (`orte-submit`). The setup of the DVM requires a single ALPS interaction, after which all the tasks are executed independent of ALPS. As RP is a Python application and ORTE is implemented in C, we interface the two systems using the ORTE CLI. While this enabled concurrent task execution and sharing nodes among tasks, we did run into new bottlenecks. The interaction with the filesystem becomes a limiting factor for task

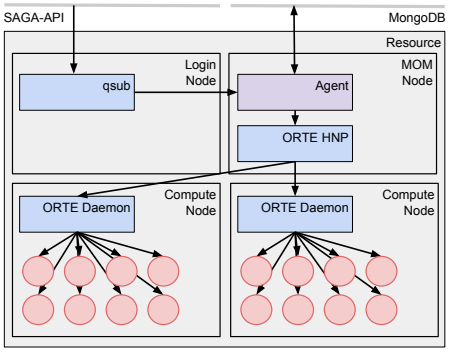


Fig. 5: Architecture overview of RP with ORTE backend.

execution as every task requires the execution of `orte-submit`. We also experience network socket race conditions and system resource limits above 16,000 concurrent tasks, as every task requires an `orte-submit` instance that communicates independently with the `orte-dvm`. RP has the ability to spread the execution management of tasks over multiple compute nodes, addressing the problem of having a large centralized process footprint for maintaining state about each running process this way.

C Foreign Function Interface for Python (CFFI) CFFI [22] provides a convenient and reliable way to call compiled C code from Python using interface declarations written in C. This mode of operation is similar to the CLI mode, but differs in the way RP interfaces with ORTE: RP launches each task using a library call instead of the `orte-submit` tool. This also allows the re-use of network socket, thus further decreasing the per-call overhead. The incentive for developing this approach was to overcome the limits and overheads imposed by the CLI approach. We called the resulting launch method "ORTE-LIB".

5 Experiments

We characterize the performance of the RP Agent by performing experiments to benchmark individual components and integrated experiments on the Agent as a whole. The results of experiments on individual components, referred to as microbenchmarks, characterize the performance of a component in isolation, while integrated experiments characterize the performance of a pipeline of components, taking into account the communication and coordination overheads of their orchestration. Experiments were performed on two Cray systems: *Blue Waters* at NCSA, and *Titan* at ORNL.

We use two metrics to characterize the performance of individual components: throughput and concurrency. As seen in §3, the RP Agent is designed as a pipeline of distinct components with multiple instances. For each instance of a component, throughput measures the rate at which units are managed, concurrency the volume of concurrently managed units. We measure the throughput of a component as the number of units it handles per second, concurrency as the number of units it handles at a given point in time.

We use two different metrics to characterize the integrated performance of the RP Agent: total time to execution of the given workload (TTX) and resource utilization (RU). TTX is a measure of how fast a set of tasks can be executed by the RP Agent. It includes the time taken by the RP Agent to manage and spawn the units for execution and the time taken by the units to execute. RU is a measure of the percentage of available core-time spent executing the workload and/or the RP Agent. TTX and RU are relevant for HPC resources, which traditionally have been designed to execute large parallel jobs and maximize overall utilization.

Depending on the type of experiment, the number of units, number or cores per unit, duration of the unit, number of instances of a component, and number

of cores of a pilot are configurable parameters. By varying the values of these parameters, we measure the amount of units that are in a specific state as a function of time, or the time duration spent in a specific state. For example, we measure the number of units in state `A_SCHEDULING` and `A_EXECUTING` at every point in time in the RP Agent Scheduler component and derive the throughput of that component.

To capture all of the measurements mentioned above, RP is instrumented with a profiling facility to record timestamps of its operations. As the execution of a given workload proceeds (as described in §3.3), each state transition is recoded as an event. These events are written to disk for postmortem analysis via dedicated utility methods. RP's profiler is designed to be non-invasive and to have minimal effect on the runtime. We measured the temporal overhead of the profiler with a dedicated benchmark: For the same workload executed on the same resources, the overall running time of the Agent was $144.7 \pm 19.2s$ with profiling, and $157.1 \pm 8.3s$ without. Note how the standard deviation of the two measurements overlap, making the difference between the two execution times statistically insignificant.

The execution of workloads with multiple tasks on a pilot has a varying degree of concurrency, depending on the total number of cores required by the tasks and available on the pilot. When the pilot has fewer available cores than what is required by the workload, a group of tasks are executed sequentially. We call this group of tasks a 'generation'. The number of generations of a workload execution affects the theoretical minimum TTX of that execution. For example, given a workload with 128 single-core, 10 minutes-long tasks and a pilot with 64 core, the execution of that workload will require 2 generations. The theoretical minimum TTX of is 2 generations $\times$ 10 minutes, assuming 100% RU of the pilot's cores and no RP Agent overhead.

It is fundamental to understand that the executable of a unit is irrelevant to the set of experiments performed here: whether a unit runs `sleep`, `stress`, an emulator (e.g., `Synapse`), a simulation kernel (e.g. `Gromacs`) or any other executable has no effect on the measure of the throughput and concurrency of the RP Agent components, or on TTX and RU. This follows from the design and separation of scheduling, launching and execution of a process. The RP Agent schedules and launches a unit and, once launched, the unit executes its code. While the unit is executing, the Executer component of RP Agent will not interact with the unit. What code the unit is executing is completely irrelevant to the Executer and therefore to RP as a whole.

5.1 Microbenchmark Experiments

Microbenchmarks measure the performance of individual RP components *in isolation*. In a microbenchmark experiment, RP launches a pilot on a resource with a single unit scheduled onto the Agent. When the unit enters the component under investigation, it is cloned a specified number of times—10,000 for experiments in this paper. The components operate on the clones, experiencing real loading while being stressed in isolation and independent of other components.

Microbenchmark experiments are designed to isolate a component by eliminating communication, coordination and concurrency with other components. In this way, the benchmarked component does not compete for shared system resources and communication channels, and remains immune from bottlenecks in other components. Thus, the microbenchmark measures the performance *upper bound* of a component implementation, as achieved *in isolation* from all types of overhead as a consequence of interaction with other components.

We perform microbenchmark experiments for the Scheduler and Executer components of RP Agent, the two components that most affect the overall performance of the RP Agent (see Fig. 4). For the Executer, we test two launch methods: ORTE-CLI, and ORTE-LIB. Note that these methods are not used by the **executable** of the units, but instead by the RP component to launch the executable. In turn, the executable could be single/multi-thread/process or use MPI itself. Depending on the launch methods, we run microbenchmarks load-balancing among 2, 4 and 8 Executer instances, executed on 1, 2, 4, and 8 compute nodes.

We perform microbenchmark experiments on *Blue Waters* as the representative Cray system. As noted before, the executable of the units has no bearing on the microbenchmarks. Microbenchmarking of the Scheduler component require no execution, while Executer benchmarking requires actual execution of the units. We use the `sleep` command to avoid any irrelevant complication deriving from setting up specific execution environments.

A full set of microbenchmarks would span a large parameter space, making it unfeasible to present the full set of experimental results. We focus on results which expose performance and scaling differences among the RP Agent components. This enables a better characterization of the overall performance of the Agent.

Agent Scheduler Performance Currently, RP can instantiate exactly one Scheduler component per Agent. The Scheduler is compute and communication bound: the scheduling algorithm searches repeatedly through the list of managed cores, while core assignment and deassignment are handled in separate, message-driven threads.

Fig. 6(a) shows the performance of the Scheduler component in assigning cores to one generation of single-core units, for four pilot sizes. We see that the throughput is dependent on the pilot size, and that the throughput rate declines as more units are scheduled. This is explained by the chosen scheduling algorithm and its implementation: the fewer free cores remain, the more work needs to be done by the scheduling algorithm to find a suitable set of cores for the next units. This behavior is a consequence of using one scheduler to handle workloads with both homogeneous and heterogeneous units (single/multi-core, mpi, cpu/gpu, etc.). In §5.3, we show how a special-purpose scheduler drastically improves performance.

Fig. 6(b) shows the same workload of the previous microbenchmark experiment, but the measurements also include the operations of unscheduling units

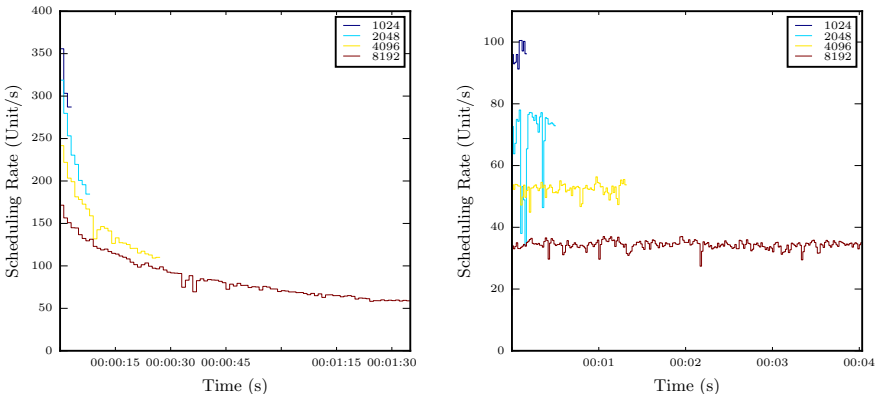


Fig. 6: RP Agent Scheduler component throughput as function of time. 1 generation of single-core units on 4 pilot sizes, including (a) allocating cores to a unit; (b) both allocating cores to a unit and deallocating those cores from the units (i.e., steady state)

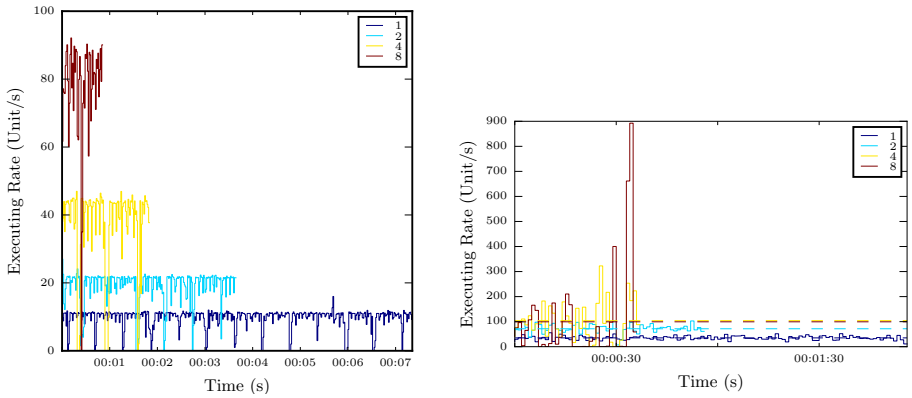
and freeing cores (i.e., steady state scheduler). We do not observe the slope of Fig. 6(a) because both the scheduling and unscheduling operations contend the lock on the Scheduler data structure. This considerably reduces the performance of the Scheduler when compared to only allocating cores to the units.

Unit Execution Performance RP can instantiate multiple Executer component instances per Agent. The Executor's performance is bound by the launch methods used to spawn the units for execution. Currently, RP supports four launch methods on Cray (ALPS, CCM, ORTE-CLI, and ORTE-LIB). Only the two ORTE-based methods enable single/multi-core units within and across compute nodes to run, at scales comparable to the size of *Blue Waters* and *Titan*.

Fig. 7(a) shows the scaling behavior of the ORTE-CLI launch method. Throughput scales with the number of Executer components, with each component running on a dedicated compute node. Data for experiments with increasing instances per node are not presented, as no performance improvements were observed. This suggests that the current performance of the Executer component using ORTE-CLI has an upper-bound due to interaction with the OS.

While ORTE-CLI did not scale with multiple instances of an Executer component on a single compute-node, Fig. 7(b) shows that with the ORTE-LIB launch method, performance scales with up to 4 instances per node. Adding more instances does not increase the performance further. This suggests that 4 Executer components on 1 compute node and the ORTE-LIB launch method reach the performance upper-bound of the ORTE layer.

Fig. 8 shows the scaling of the ORTE-LIB launch method for different pilot sizes. We launch 1,024, 2,048, 4,096 and 8,192 single-core units on pilots with 1,024, 2,048, 4,096 and 8,192 cores. Throughput is stable over time but jittery with a mean (std. dev) of 48.2 (10.2), 42.6 (7.1), 39.1 (9.8) unit/s. The jitter is explained by the interaction with many external system components which, in their totality, introduce significant noise.



(a) (b)

Fig. 7: Throughput of the RP Agent Executer component with 2 launch mechanisms. (a) ORTE-CLI, 1–8 Executer com - LIB, 1–8 instances, all run on the

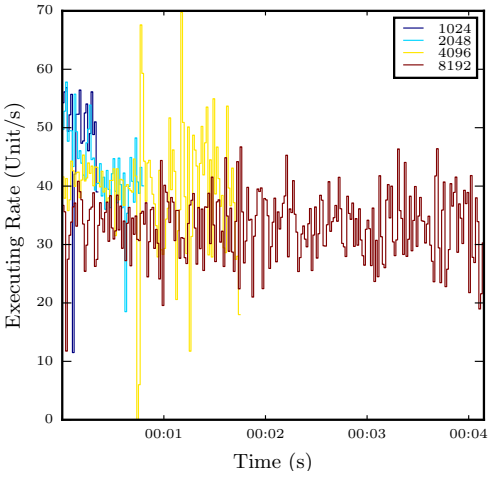


Fig. 8: Throughput of 1 Executer component with ORTE-LIB launch method; 1,024, 2,048, 4,096 and 8,192 cores/units.

The best performance of ORTE-CLI is lower than the performance of the Scheduler for a pilot with up to 1,024 cores, as seen in Fig. 6. This indicates that ORTE-CLI creates a bottleneck at the launching stage in the Agent's Executer. In absolute terms, the performance of ORTE-LIB is lower than the scheduling component's when the pilot size is less than 8,192 cores, and comparable (or at times higher) at pilot sizes over 8,192 cores. Similar to the Scheduler component, the performance decreases with increased pilot size, from an average rate of around 48 units/s for the 1,024 pilot size to an average rate of around 33 units/s for larger pilots.

5.2 Agent Integrated Performance

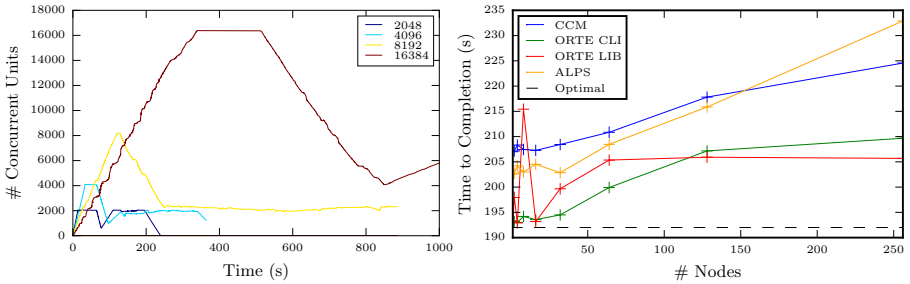
To characterize the RP Agent performance as a whole, we employ workloads with varying unit durations executed on pilots of different sizes. The size of each unit is set to 1 core, allowing experiments to measure the performance of RP with maximum pilot cores/unit ratio. Workloads with multi-core units lower the overall stress on the components of the Agents and their communication and coordination protocols, resulting in better performance.

Microbenchmarks are not sufficient to characterize the Agent performance as a whole for three reasons: (i) by definition, the microbenchmarks in §5.1 and §5.1 cannot measure the performance cost of communication among components; (ii) the concurrent operation of multiple components introduces competition for shared system resources (e.g., competing for filesystem access); and (iii) the Agent performance can be limited by components or system resources *outside* of the Agent (e.g., RP client manager components, or network latency between the Agent and MongoDB).

Accordingly, the set of integration experiments discussed in this subsection investigates the contributions of communication and concurrency to the Agent performance. To offset external overheads, we design the experiments so that the Agent operates independent of the performance of the PilotManager and UnitManager components (Fig.4): we introduce a startup barrier in the Agent to ensure that the Agent receives sufficient work to fully utilize the pilot's resources. In this way, the Agent starts to process units only when the complete workload has arrived at the Agent.

On *Blue Waters*, we measure time-dependent concurrency achieved by the RP Agent for pilots with 2,048, 4,096, 8,192, and 16,384 cores. For each pilot size, the workload is comprised of 3 generations of single-core units, resulting in workloads with 6,144, 12,288, 24,576, and 49,152 units. For each workload, the duration of each unit is 64, 64, 128 and 256 seconds, long enough for all the units of the first generation to start before the first unit is completed. In this way, the first generation can always reach maximum concurrency, saturating the number of cores available on the pilot.

Fig. 9(a) shows the maximal concurrency for each pilot size, where all cores are simultaneous used to execute units. The initial slope up to that maximum concurrency is determined by the performance of the scheduler, which, as shown in Fig. 6(a), is dependent on the pilot size. For example, with the 8,192-core pilot we see that 8,192 units are started in about 100 seconds. This is comparable to what is shown in figure Fig. 6(a), where 8,192 units are scheduled in about 90 seconds, with a throughput which starts out at 150 units/seconds and later stabilizes at about 50 units/second.



(a) (b)

Fig. 9: (a) Unit concurrency as a function of pilot size and unit duration. (b) Time to Execution (TTX) as a function of number of units, size of pilot and Executer launch method.

Fig. 9(a) shows also that once the first generation of units begins to finish execution, the scheduler enters a different mode of operation where scheduling and unscheduling threads compete (see discussion of Fig. 6(b)). This decreases the overall throughput of the Agent which is no longer able to maintain full concurrency. This effect is independent of pilot size and number of units.

Comparing ORTE to ALPS and CCM One of the limitations of ALPS/APRUN is that we can only run one unit per node. SSH based launch methods in CCM-mode on *Blue Waters* are also limited, due to connection limits when executing more than 8 concurrent units per node. ORTE does not have that limitation. In order to keep the runs comparable, i.e., to execute the same configurations for all experiments, we configure the workload used to use 32 cores per unit, so that each unit consumes a full node. This workload can be executed with all RP launch methods.

On *Blue Waters*, we run 10 workloads ranging from 3 to 768 units, where each unit consumes a full compute node (32 cores) and executes on pilots ranging form 32 cores (1 node) to 8,192 cores (256 nodes) respectively. We run the same set of 10 workloads for each launch method and compare the actual Time to Execution (TTX) against the theoretically optimal TTX (i.e., the time taken by all the units to execute without any RP overhead).

Fig. 9(b) shows that there is a large trend difference between ORTE-CLI/ORTE-LIB and ALPS/CCM. As the scale increases, the difference between ALPS/CCM to ORTE increases, with ORTE being closer to the theoretically optimal TTX and ALPS/CCM increasing somewhat linearly after around 50 nodes.

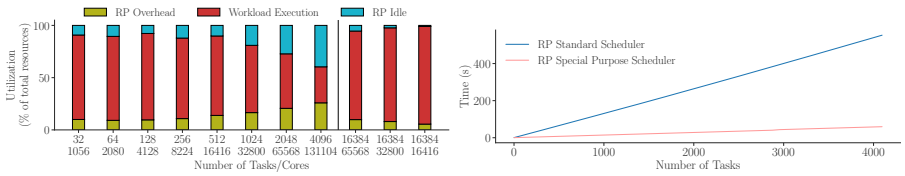
5.3 Resource Utilization and Overheads at Scale

Currently, the ORTE launch method is the one supporting the largest runs with RP on Cray machines, allowing to execute workloads with 16,384 multi-core units on more than 130,000 cores. We run two experiments to characterize the weak and strong scaling behavior of RP and its overheads up to this scale. In the weak scaling experiment, we execute workloads with a constant ratio between units and cores. In the strong scaling experiment, we execute one workload on a progressively larger amount of cores. In this way, the strong scaling experiment executes the workload with between 32 and 2 generations.

Weak and strong scaling experiments execute workloads with 32 cores, 15 minutes long MPI tasks. We perform both experiments on *Titan* for two main reasons: (i) *Titan* is very similar to *Blue Waters* in terms of architecture and scale; and (ii) these experiments required around 25 million core-hours, at the time available only on *Titan*.

Fig. 10(a) shows both the weak (first 8 bars) and strong (last 3 bars) scaling experiments. We measure resource utilization as percentage of the available core-time spent executing the workload (red), RP code (green), or idling (blue). Runs measuring weak scaling with between 32/1,024 and 128/4,096 tasks/cores have a relatively constant percentage of core-time utilization but this percentage

decreases with the growing of the number of tasks/cores. As a result, we observe that RP Agent does not weak scale with pilot larger than 8,192 cores.



(a) (b)

Fig. 10: (a) Resource utilization of RADICAL-Pilot. (b) Scheduling overheads: Standard and special purpose schedulers.

Runs measuring strong scaling show values of RP overhead and idling inversely proportional to the number of generations: the more generations, the less overhead and idling. This is explained by noting that, when tasks of one generation terminate, those of the following generation immediately starts executing. This eliminates the idling of cores for all generations but the last one. We presume that the increase of RP overhead depends, at least to some extent, on the proportional relation between the communication required to coordinate an execution and the size of the pilot used.

Reducing RP overhead We explore the decrease in resource utilization measured in the weak scaling experiment (Fig. 10(a), first 8 bars) by looking at the results of the microbenchmark shown in Fig. 6, and focusing on the relation between scheduling performance and size of the pilot used for the execution.

As described in §5.1, the larger the pilot, the larger is the resource pool managed by the scheduler. Currently, the scheduler is implemented to repeatedly search a Python data structure for available cores. This approach is effective for a general purpose scheduler that needs to handle many types of workload—e.g., homogeneous/heterogeneous, MPI/OpenMP/Scalar, or single-node/multi-node. However, for homogeneous workloads, a more efficient single-purpose scheduler can be implemented.

Leveraging the flexibility and extensibility of RP (as also used for the Executer and its multiple launch methods), we implemented a scheduler algorithm which specifically handles homogeneous, multi-node tasks of workloads used in weak and strong scaling experiments. The behavior of this special purpose scheduler is shown in Fig. 10(b): the scheduler manages each task in constant time, at a much lower time per task compared to the general-purpose scheduler.

When the special purpose scheduler encounters the first unit to schedule, it immediately divides the total set of cores into partitions which are all of the same size as the number of cores required by the first unit. In this way, the scheduling algorithm is reduced to the procedure of assigning equally-sized partitions to the units as they arrive. Crucially, this avoid the need for any search on a (Python) data structure representing the cores managed by the pilot. Instead, partition lookup and assignment can be performed in constant time.

It should be noted that there still remain limitations for when the second generations of units gets scheduled, i.e., when the scheduling and unscheduling

processes compete. Nonetheless, the throughput of this scheduler is consistently higher than for the general-purpose scheduler: the lock contention reduces due to the reduced time for which the scheduler algorithm needs to lock the data structures. Full details on the homogeneous bag of task scheduler and more detailed measurements are discussed in [23].

5.4 Discussion

Microbenchmark experiments provide insight on how the Agent's Scheduler and Executer components perform for different Agent configurations and pilot sizes (§5.1). These experiments provide an upper-bound of the throughput (i.e., units handled per second) of each of the two components and show which component could be the rate-determining factor in the overall agent integrated performance scales. Microbenchmark experiments for the Agent's Scheduler component show that the scheduling throughput is dependent on the pilot size, and that the throughput rate declines as more units are scheduled. Further, we show that when the component is doing both the scheduling and unscheduling operations (i.e., scheduler reaches steady state) the scheduler's throughput is primarily dependent on the pilot size as seen in Fig. 6(b), viz., as the pilot size increases, the scheduler's throughput decreases.

The microbenchmark experiments with the Executer component show that its throughput decreases when the pilot size and unit counts increase in the same proportion, similar to what was observed for the Scheduler component. For both ORTE-CLI and ORTE-LIB Executer launch methods we show increased throughput when an increasing number of concurrent executers are used. ORTE-LIB allows multiple executers on the MOM node, while ORTE-CLI requires a compute node for each executer. As described in §4.3, this is explained by observing that an execution through ORTE-LIB is only a library call that causes a network call and doesn't strain the system on which it is running. Conversely, each execution call done through ORTE-CLI requires interactions with the filesystem and network resources to communicate with the `orte-dvm`. Thus an agent using ORTE-CLI reaches the resource limits of *Blue Waters* and Titan with workloads that consists of very large numbers of concurrent tasks or when running multiple components on the same MOM node.

§5.2 describes experiments that characterize the integrated performance of RP Agent. We show that the maximal concurrency achievable for multiple pilot sizes, where all cores are simultaneously used to execute units, is approximately 16,000 units. We also compare the performance of the ALPS, CCM and ORTE Executer launch methods and found that ORTE-LIB and ORTE-CLI launch methods out-perform ALPS and CCM for the performance metric of TTX. We also show that the performance of ORTE-CLI and ORTE-LIB launch methods are comparable when the number of units is between 3 and 768 units, but this is likely to change with a higher number of units, where configurations with multiple executers and the lower impact of ORTE-LIB on resource utilization would make the ORTE-LIB launch method perform better than ORTE-CLI.

Further, we note that the performance of CCM on *Blue Waters* is low compared to other launch methods available on non-Cray HPC systems like Stampede [24].

Finally, we measured the resource utilization of RP Agent at highest scale currently supported, both in terms of number of units concurrently executed and number of cores of a single pilot. We moved from a single-core units to units of 32-cores, and we ran weak scaling experiments with workloads ranging from 32 to 4,096 units on pilot sizes ranging from 32 to 131,104 cores respectively, and strong scaling experiments with workloads of 16,384 units on pilot sizes ranging from 16,416 to 65,568 cores (§5.3). Our experiments show that resource utilization of the RP Agent for the weak scaling experiments with pilot sizes between of 1,024 and 4,096 cores have a relatively constant percentage of core-time utilization, but this percentage significantly decreases with pilots larger than 8,192 cores. We attributed RP Agent's poor weak scaling property with pilot sizes over 8,192 cores to the performance of the Agent Scheduler component.

We addressed the decrease in resource utilization measured in the weak scaling experiment and showed the flexibility and extensibility or RP, by implementing a special-purpose scheduler, specific to the experimental workload—i.e., homogeneous, multi-node tasks. We then showed that the overhead added by the special-purpose scheduler significantly decreases compared to the one of the special-purpose scheduler used for the experiment.

6 Conclusion

Prima facie, a system implementing the Pilot abstraction [2, 25] provides the conceptual and functional capabilities needed to support the scalable execution of many task workloads. The impact of an abstraction is limited to its best implementation. Whereas there are several existent pilot systems, they are either geared towards specific functionality or platforms. This paper describes the architecture and implementation of RP (§3.1), and characterizes the performance of its Agent module on Cray platforms (§5).

In molecular sciences, there is a demonstrated need [26] to be able to support up to 10^5 MPI tasks as part of a single "ensemble simulation". Similar scales are anticipated across multiple domains. Several parts of RP will need to be re-engineered to efficiently execute workloads at this scale. Most of the benefits will come from improving the Agent, as discussed in §3.1 and consistent with results shown in §5. To this end, we are planning to: (i) develop a set of specialized, lock-free schedulers; (ii) partition the pilot resources and operate multiple agents in parallel on these partitions; (iii) explore new launch methods; and (iv) aggregate units depending on their application provenance and duration to optimize Scheduler and Executor throughput.

The focus of this paper has been on the direct execution of workloads on HPC machines, but RP also serves as the runtime system for a range of other tools and libraries [27–30], many already used in production. The requirements of these tools and libraries will also motivate future research and development.

Software and Data: RP is available for immediate use on many platforms [31]. RP source is accompanied with extensive documentation and an active developer-user community. Source code, raw data and analysis scripts to reproduce experiments can be found at:
RADICAL-Pilot: `https://github.com/radical-cybertools/radical.pilot`
RADICAL-Analytics: `https://github.com/radical-cybertools/radical.analytics`
Experiment data and scripts: `https://github.com/radical-experiments/jsspp18`

Acknowledgments: This work is supported by NSF "CAREER" ACI-1253644, NSF ACI-1440677 "RADICAL-Cybertools" and DOE Award DE-SC0016280. We acknowledge access to computational facilities: XSEDE resources (TG-MCB090174) and Blue Waters (NSF-1713749).

Bibliography

[1] Hwang, E., Kim, S., k. Yoo, T., Kim, J.S., Hwang, S., r. Choi, Y.: Resource allocation policies for loosely coupled applications in heterogeneous computing systems. IEEE Transactions on Parallel and Distributed Systems **PP**(99) (2015) 1–1

[2] Turilli, M., Santcroos, M., Jha, S.: A comprehensive perspective on Pilot-Jobs. ACM Computing Surveys (accepted, in press) (2017) `http://arxiv.org/abs/1508.04180`.

[3] Preto, J., Clementi, C.: Fast recovery of free energy landscapes via diffusion-map-directed molecular dynamics. Physical Chemistry Chemical Physics **16**(36) (2014) 19181–19191

[4] Cheatham III, T.E., Roe, D.R.: The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. Computing in Science & Engineering **17**(2) (2015) 30–39

[5] Sugita, Y., Okamoto, Y.: Replica-exchange molecular dynamics method for protein folding. Chemical physics letters **314**(1) (1999) 141–151

[6] Pordes, R., et al.: The Open Science Grid. J. Phys.: Conf. Ser. **78**(1) (2007)

[7] Maeno, T., De, K., Klimentov, A., Nilsson, P., Oleynik, D., Panitkin, S., Petrosyan, A., Schovancova, J., Vaniachine, A., Wenaus, 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. Volume 513(3)., IOP Publishing (2014) 032062

[8] Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., Wilde, M.: Falkon: a Fast and Light-weight tasK executiON framework. In: Proceedings of the 8th ACM/IEEE conference on Supercomputing, ACM (2007) 43

[9] Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A language for distributed parallel scripting. Parallel Computing **37**(9) (2011) 633–652

[10] CCM: `http://bit.ly/cray_ccm` (accessed January 2018).

[11] Karo, M., Lagerstrom, R., Kohnke, M., Albing, C.: The application level placement scheduler. (2006)

[12] Castain, R.H., Squyres, J.M.: Creating a transparent, distributed, and resilient computing environment: the OpenRTE project. The Journal of Supercomputing **42**(1) (October 2007) 107–123
[13] TaskFarmer: `http://bit.ly/taskfarmer`.
[14] Wraprun: `https://www.olcf.ornl.gov/kb_articles/wraprun/`.
[15] QDO: `http://bit.ly/nersc_qdo`.
[16] Canon, R.S., Ramakrishnan, L., Srinivasan, J.: My Cray can do that? Supporting diverse workloads on the Cray XE-6. Cray User Group (2012)
[17] Python Task Farm: `http://www.archer.ac.uk/documentation/user-guide/batch.php#sec-5.7`.
[18] Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: A next-generation resource management framework for large hpc centers. In: Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on, IEEE (2014) 9–17
[19] Merzky, A., Weidner, O., Jha, S.: SAGA: A standardized access layer to heterogeneous distributed computing infrastructure. Software-X (2015) DOI: 10.1016/j.softx.2015.03.001.
[20] Santcroos, M., Castain, R., Merzky, A., Bethune, I., Jha, S.: Executing dynamic heterogeneous workloads on blue waters with radical-pilot. In: Cray User Group 2016. (2016)
[21] PMIx web site: `https://www.open-mpi.org/projects/pmix/`.
[22] CFFI Documentation: `http://cffi.readthedocs.org`.
[23] Merzky, A., Turilli, M., Maldonado, M., Jha, S.: Design and performance characterization of RADICAL-pilot on titan. arXiv preprint arXiv:1801.01843 (2018)
[24] Merzky, A., Santcroos, M., Turilli, M., Jha, S.: Executing Dynamic and Heterogeneous Workloads on Super Computers (2016) (under review) `http://arxiv.org/abs/1512.08194`.
[25] Luckow, A., Santcroos, M., Merzky, A., Weidner, O., Mantha, P., Jha, S.: P*: A model of pilot-abstractions. IEEE 8th International Conference on e-Science (2012) 1–10 `http://dx.doi.org/10.1109/eScience.2012.6404423`.
[26] Shantenu Jha and Peter M. Kasson: High-level software frameworks to surmount the challenge of 100x scaling for biomolecul ar simulation science. White Paper submitted to NIH-NSF Request for Information (2015) `http://dx.doi.org/10.5281/zenodo.44377`.
[27] Balasubramanian, V., Treikalis, A., Weidner, O., Jha, S.: Ensemble toolkit: Scalable and flexible execution of ensembles of tasks. In: 2016 45th International Conference on Parallel Processing (ICPP). Volume 00. (Aug. 2016) 458–463
[28] Treikalis, A., Merzky, A., Chen, H., Lee, T.S., York, D.M., Jha, S.: Repex: A flexible framework for scalable replica exchange molecular dynamics simulations. In: 2016 45th International Conference on Parallel Processing (ICPP). (Aug 2016)
[29] Balasubramanian, V., Turilli, M., Hu, W., Lefebvre, M., Lei, W., Cervone, G., Tromp, J., Jha, S.: Harnessing the Power of Many: Extensible Toolkit for Scalable Ensemble Applications. (2017) `https://arxiv.org/abs/1710.08491`.
[30] Balasubramanian, V., Bethune, I., Shkurti, A., Breitmoser, E., Hruska, E., Clementi, C., Laughton, C., Jha, S.: Extasy: Scalable and flexible coupling of md simulations and advanced sampling techniques. In: 2016 IEEE 12th International Conference on e-Science (e-Science). (Oct 2016) 361–370
[31] RADICAL-Pilot: `https://github.com/radical-cybertools/radical.pilot`.