Executing dynamic heterogeneous workloads on Blue Waters with RADICAL-Pilot

Mark Santcroos*, Ralph Castain†, Andre Merzky*, Iain Bethune‡ and Shantenu Jha*
* *School of Electrical and Computer Engineering, Rutgers University, New Brunswick, New Jersey, USA*
† *Intel Corporation, USA*
‡ *EPCC, The University of Edinburgh, Edinburgh, UK*

***Abstract*—Traditionally HPC systems such as Crays have been designed to support mostly 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. These workloads can benefit from being executed at scale on HPC resources, but a tension exists between the workloads' resource utilization requirements and the capabilities of the HPC system software and usage policies. Pilot systems have the potential to relieve this tension. RADICAL-Pilot is a scalable and portable pilot system that enables the execution of such diverse workloads. In this paper we describe the design and characterize the performance of its RADICAL-Pilot's scheduling and executing components on Crays, which are engineered for efficient resource utilization while maintaining the full generality of the Pilot abstraction. We will discuss four different implementations of support for RADICAL-Pilot on Cray systems and analyze and report on their performance.**

I. Introduction

Traditionally, high-performance computing (HPC) systems such as Crays have been primarily designed to support monolithic workloads, i.e. a single parallel application running across hundreds or thousands of compute nodes. However, the workload of many important scientific use-cases is instead composed of spatially and temporally heterogeneous tasks that are often dynamically inter-related [1], [2], [3]. For example, simulating the dynamics of complex macromolecules is often done using "swarms" of short molecular dynamics calculations, each running on a small number of cores. The output of these calculations is collected to determine the next set of simulations. Such workloads still benefit from execution at scale on HPC resources but a tension exists between the workload's resource utilization requirements and the capabilities and usage policies of the HPC system software.

Pilot systems have proven particularly effective in the execution of workloads comprised of multiple tasks on physically distributed resources [4]. They decouple workload specification, resource selection, and task execution via job placeholders and late-binding. Pilot systems 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. Tasks are thus executed within time and space boundaries set by the resource scheduler, yet are scheduled by the application. Per the definitions in [4], we refer to a "task" as a part of the workload and a "job" as the container used to acquire resources.

In this paper, we describe and experimentally characterize

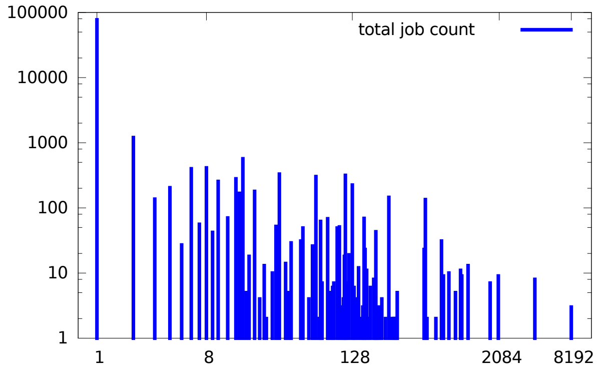


Figure 1: Distribution of job size (in number of requested nodes) submitted to *Blue Waters*. The period is one year between April 2015 and April 2016. Single node jobs outnumber other node sizes by (an) order(s) of magnitude.

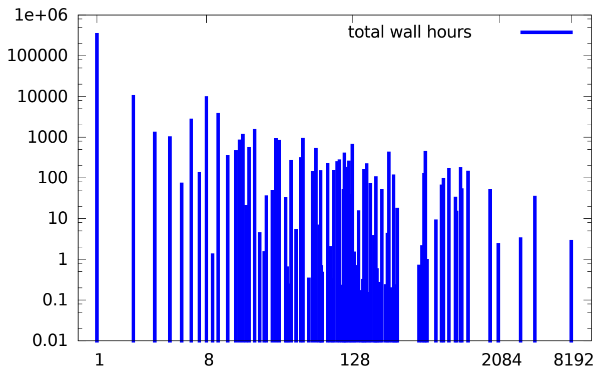


Figure 2: Total number of node hours spent per job size. The period is one year between April 2015 and April 2016. Single node jobs account for the 3rd largest consumption of node hours.

our Python based pilot system, RADICAL-Pilot (RP) [5], on the *Blue Waters* Cray XE6/XK6 at NCSA (BW). *Blue Waters* is a 26,868 node Super Computer with a peak performance of a 13.3 petaFLOPS. Like many HPC systems, it is designed for applications that would use a large portion of its thousands of processors and that would be difficult or impossible to run elsewhere.

The many-task computing [6] paradigm was introduced to bridge the gap between the high throughput computing and high performance computing paradigms. From a workload perspective, many-task computing makes no assumptions on workload duration, size, or origin (e.g. distributed scientific workflows, multi-component applications). Figure 1 shows the distribution of job size on *Blue Waters* in number of nodes during the period of one year. Small jobs outnumber larger jobs in count. Figure 2 also shows how small jobs still play a significant role also when considering total node hours.

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 provides a runtime system for applications with heterogeneous and dynamic workloads. RP exposes an application-facing API called the "Pilot API" [7] and utilizes SAGA [8] to interface to the resource layer. RP provides fine-grained profiling for each RP module, enabling a precise and detailed measurement of its overheads. RP can provide runtime capabilities when interfaced with other application-level tools [9], [10] or workflow and workload management systems such as Swift [11] or PanDA [12]. However, RP itself is not a workflow system and does not provide workload management capabilities itself.

RP supports heterogeneity by concurrently executing tasks with different properties and couplings on resources with diverse architecture and software environments. Dynamism is supported by enacting the runtime variations of the number, properties and coupling between tasks. RP's architecture is discussed in detail in [5], which also includes a characterization of RP's performance across heterogeneous HPC systems and workloads. This paper will only recapture details of the components which are specifically relevant to the efficient placement and execution of tasks on *Blue Waters*. We refer the reader to [5] for a more general overview of RP.

The remainder of this paper is as follows. In §II we position our work and discuss related work. As a background we discuss RP in some more detail in §III. In §IV we investigate the performance of RP experimentally on *Blue Waters*. We complete the paper with a conclusion in §V and look forward in §VI.

II. Related work

According to [4], around twenty systems with pilot capabilities have been implemented since 1995.

AppLeS [13] offered one of the first implementations of resource placeholders and application-level scheduling; HTCondor [14] and Glidein [15] enabled pilot-based execution on multiple and diverse resources; and DIANE [16], AliEn [17], DIRAC [18], PanDA [12], and GlideinWMS [19] brought pilot-based workload executions to the LHC and other grid communities.

In contrast to RP, the aforementioned systems are often tailored for specific workloads, resources, interfaces, or development models. They often encapsulate pilot capabilities within monolithic tools with greater functional scope. For example, HTCondor with Glidein on OSG [20] is one of the most widely used Pilot systems but serves mostly single core workloads. The Pilot systems developed for the LHC communities execute millions of jobs a week [12] but specialize on supporting LHC workloads and, in most cases, specific resources like those of WLCG.

Similar specialization can be found in systems without pilot capabilities. For example, CRAM [21] is a tool developed to execute static ensembles of MPI tasks on HPC resources, one of the workload types also supported by RP. Developed for Sequoia, an IBM BG/Q system at LLNL, CRAM parallelizes the execution of an application with many input parameters by bundling it into a single MPI executable. Compared to CRAM, RP generalizes ensemble capabilities for both MPI and non-MPI applications, and for applications for which execution plans are not known in advance. CRAM could in principle also be made to support Cray systems.

Recognizing the potential for High-Throughput Computing (HTC) on HPC resources, IBM developed an HTC mode resembling a Pilot system [22] for the series of IBM BG/L products. Unsupported by later IBM Blue Gene series, RP brings back this HTC capability generalizing it to HPC architectures beyond IBM BG/L machines, like the BG/Q. Nitro [23], is a high-throughput scheduling solution for HPC systems that works in collaboration with the Moab scheduler for TORQUE. Instead of requiring individual job scheduling, Nitro enables high-speed throughput on short computing jobs by allowing the scheduler to incur the scheduling overhead only once for a large batch of jobs.

Pilots and pilot-like capabilities are also implemented or used by various workflow management systems. Pegasus [24] uses Glidein via providers like Corral [25]; Makeflow [26] and FireWorks [27] enable users to manually start workers on HPC resources via master/worker tools called Work Queue [28] and LaunchPad [27]; and Swift [11] uses two Pilot systems called Falkon [29] and

Coasters [30]. In these systems, the pilot is not always a stand-alone capability and in those cases any innovations and advances of the pilot capability are thus confined to the encasing system. Pegasus-MPI-Cluster (PMC) [31] is an MPI-based Master/Worker framework that can be used in combination with Pegasus. In the same spirit as RP, this enables Pegasus to run large-scale workflows of small tasks on HPC resources. In constrast with RP, tasks are limited to single node execution. In addition there is a dependency on $fork()$/$exec()$ on the compute node which rules out PMC on some HPC resources.

Falkon is an early example of a Pilot system for HPC environments. Similar to RP, Falkon exposes an API that is used to develop distributed applications or to be integrated within an end-to-end system such as Swift and it has been designed to implement concurrency at multiple levels including dispatching, scheduling, and spawning of tasks across multiple compute nodes of possibly multiple resources. However, Falkon is optimized for single core applications. Coasters is similar to RP in that it supports heterogeneity at resource level. RP supports a greater variety of resources though, mainly due to the use of SAGA as its resource interoperability layer. The two systems differ in their architectures and workload heterogeneity (RP also supports multi-node MPI applications).

JETS [32] is a middleware component providing Swift and Coasters with high performance support for many-parallel-task computing (MPTC). JETS executes short-duration MPI tasks at scale using pilots managed by Coasters and workloads codified in the Swift scripting language. RP enables MPI executions natively, decoupling the implementation of application-side patterns of distributed computation like MPTC from the resource-side communication capabilities like MPI. JETS uses runtime features available in the MPICH MPI implementation [33], similar to RP using runtime features from ORTE [34], a component of the OpenMPI MPI implementation. Swift/T, the latest incarnation of Swift [35] (T of Turbine [36]), steps away from the orchestration of executables by interpreting tasks as functions. This requires tasks to be codified as functions instead of executables, for example via the main-wrap technique presented in [37].

In addition to the solutions discussed so far, that have some degree of support for heterogeneity, there is also a set of tools specifically for Crays that have been conceived at the various sites. A tool developed at LBNL is TaskFarmer [38]. TaskFarmer enables the user to execute a list of system commands from a task file one-by-one. This allows many simulations to be run within a single mpirun allocation. TaskFarmer runs as one large MPI tasks where each rank is able to spawn tasks on the node its executes on, limiting the tasks to be single core or single node. Wraprun [39] is a utility developed at ORNL that enables independent execution of multiple MPI applications under a single aprun call. It borrows from aprun MPMD syntax and also contains some wraprun specific syntax. QDO [40] is a lightweight high-throughput queuing system for workflows that have many small tasks to perform. It is designed for situations where the number of tasks to perform is much larger than the practical limits of the underlying batch job system. Its interface emphasizes simplicity while maintaining flexibility. MySGE [41] allows users to create a private Sun GridEngine cluster on large parallel systems like Hopper and Edison. Once the cluster is started, users can submit serial jobs, array jobs, and other throughput oriented workloads into the personal SGE scheduler. The jobs are then run within the user's private cluster. Python Task Farm (ptf) [42] is a utility developed at EPCC and available on ARCHER for running serial Python programs as multiple independent copies of a program over many cores.

III. 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 (see Figure 3) and characterize the performance of RP's task execution components, which are engineered for efficient resource utilization while maintaining the full generality of the Pilot abstraction. RP is supported on on Crays such as *Blue Waters* (NCSA), Titan (ORNL), Hopper & Edison (NERSC) and ARCHER (EPSRC), but also on IBM's Blue Gene/Q, many of XSEDE's HPC resources, Amazon EC2, and on the Open Science Grid (OSG).

A. Overall Architecture

RP is a runtime system designed to execute heterogeneous and dynamic workloads on 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. 3). The PilotManager launches pilots on resources via the SAGA API [8]. 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

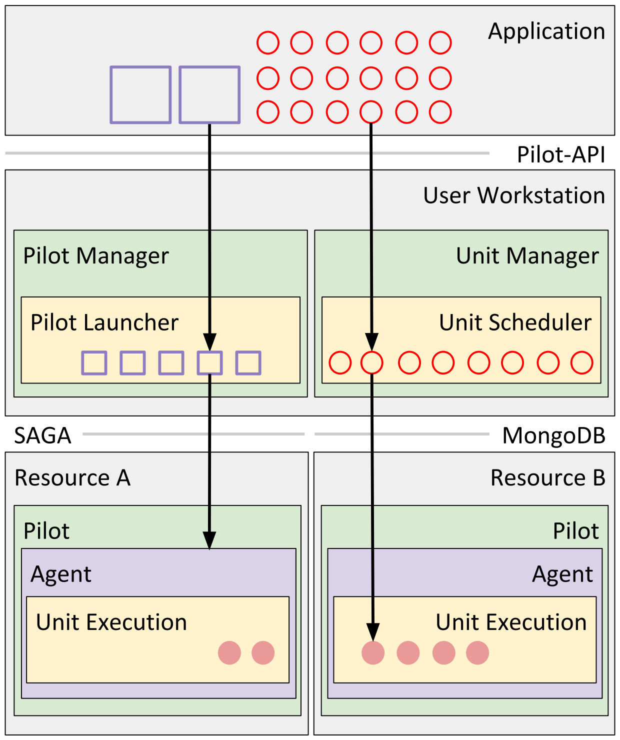


Figure 3: RADICAL-Pilot Architecture. Pilots (description and instance) in purple are for resource allocation; Units (description and instance) in red are for task execution. Applications interact with RP through the Pilot-API. Resource interoperability comes through SAGA. Unit Manager to Agent communication is via MongoDB, all other communication is via ZeroMQ.

workload between the UnitManager and Agents. For this reason, the database instance needs to be accessible both from the user's workstation and the target resources. The Agent bootstraps on a remote resource, pulls units from the MongoDB instance, and manages their execution on the cores held by the pilot. RP has a well defined component and state model which is described in detail in [5].

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 or newer on the workstation but the Agent has to execute different types of units on resources with very diverse architectures and software environments.

B. *Pre-configured Resources*

When a user installs RP the system comes with a large set of pre-configured resources. In §III-C we show refer to such a configuration.

```
"bw_aprun": {
  "description"    : "The NCSA Blue Waters \
                      Cray XE6/XK7 system.",
  "notes"          : "Use 'touch .hushlogin' \
                      on the login node.",
  "workdir"        : "/scratch/sciteam/\$USER",
  "valid_roots     :["/scratch/sciteam"],
  "virtenv"        : "%(global_sandbox)s/ve_bw",
  "rp_version"     :"local",
  "virtenv_mode"   :"create",
  "schemas"        :["gsissh"],
  "gsissh"         : {"job_mgr_url"  :
       "torque+gsissh://bw.ncsa.illinois.edu",
                      "file_mgr_url" :
       "gsisftp://bw.ncsa.illinois.edu/"
  },
  "stage_cacerts"       :"True",
  "default_queue"       :"normal",
  "lrms"                :"TORQUE",
  "agent_type"          :"multicore",
  "agent_scheduler"     :"CONTINUOUS",
  "agent_spawner"       :"POPEN",
  "agent_launch_method":"APRUN",
  "task_launch_method" :"APRUN",
  "mpi_launch_method"  :"APRUN",
  "pre_bootstrap_1"     :["module load bwpy"],
}
```

Listing 1: Resource configuration for ALPS on *Blue Waters*. Entries include access schema, batch queue system details, installation options, file system locations, bootstrap information and launch methods.

Besides making use of the pre-supplied resource configurations, users can add their own, or modify existing entries, either through config files or programmatically through the API. Listing 1 show the configuration for RP on *Blue Waters* with the APRUN launch method. It specifies some general description fields, locations for temporary files and Python virtual environments, which version of RP to install, the access mechanism and hostname, the queuing system, which scheduler and launch methods to use, and which commands to execute before the bootstrapper (e.g. load modules).

C. *Programming Model*

RP is a Python library that enables the user to declaratively define the resource requirements and the workload. 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 to a minimal but complete example of running a workload on *Blue Waters* using RP.

```
# create a session -- closing it will
# destroy all managers and all things
# they manage.
session = rp.Session()

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

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

Listing 2: Code example showing the declaration of Pilot Manager and Unit Manager within a Session.

In Listing 2 we show the code used to declare the respective managers for pilots and units, whose lifetime is managed by a session object.

```
# Define an 64 core pilot that
# will run for 10 minutes
pdesc = rp.ComputePilotDescription({
        'resource' : ncsa.bw,
        'cores'    : 64,
        'runtime'  : 10,
        'project'  : 'gkd',
        'queue'    : 'debug',
    })

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

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

Listing 3: Code example showing the declaration of a Compute Pilot, its subsequent submission to the Pilot Manager and the attachment to the Unit Manager.

In Listing 3 we declare a pilot, by specifying where to start it, how many cores, the walltime, and optional queuing and project details. Once the pilot is submitted to the Pilot manager, it will get passed to the queuing system asynchronously. In the last step the pilot is associated to the unit manager, which means that this pilot can be used to execute units on.

```
# 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()
```

Listing 4: Code example showing the declaration of 42 Compute Units, the subsequent submission to the Unit Manager and the statement to wait for their completion.

In Listing 4 we finally declare the workload by creating a set of compute units that specify what to run. The units are then submitted to the unit manager which schedules the unit to a pilot. Once the pilot has become active, the units may begin execution. The final wait call will block until all the units have run to completion.

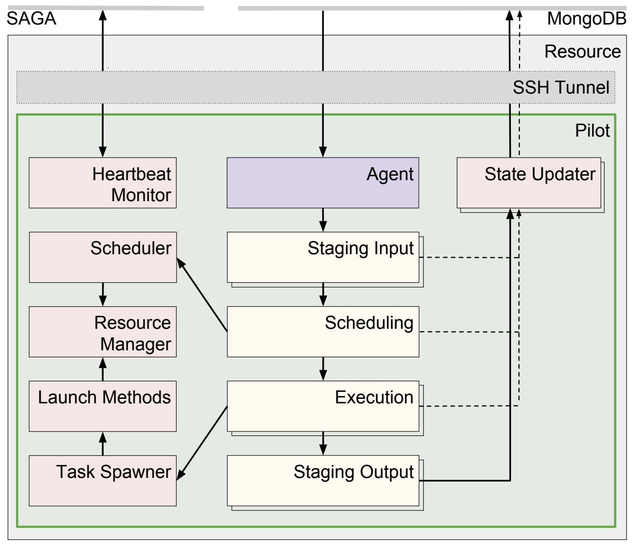


Figure 4: Agent Architecture of RP. Detailed perspective of the Agent as in Figure 3. Every unit goes through the states of Input Staging, Scheduling, Execution and Output Staging. This paper focuses on the different implementations of Launch Method and Task Spawner.

D. Agent Architecture

Depending on the architecture of the resource, the Agent's Stager, Scheduler, and Executer components (Fig. 4) can be placed on cluster head nodes, MOM nodes, compute nodes, virtual machines, or any combination thereof. Multiple instances of the Stager and Executer component can coexist in a single Agent, placed on any service node or compute node of the pilot's allocation. ZeroMQ communication bridges connect the Agent components, creating a network to support the transitions of the units through components.

E. Enabling RP on Cray systems

To enable RP on Cray systems we have developed four ways of interfacing RP and the Cray system software.

1) Application Level Placement Scheduler (ALPS): The ALPS system provides launch functionality for running executables on compute nodes in the system, interfaced 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 1000 applications concurrently within one batch job, while in practice we also see values of only 100 concurrent applications. In the Pilot use-case, these applications may run only for a very short time, which puts further strain on ALPS and the MOM node and effectively limits the throughput of these applications. ALPS also does not allow the user to easily run more than one tasks on a single compute node, which makes it unattractive to launch workloads with heterogeneous application size.

2) Cluster Compatibility Mode (CCM): Crays are effectively MPP machines and the Cray Compute Node OS does not provide a full set of the Linux services compared to typical Beowulf clusters. CCM is a software solution that provides those services when required by applications. It is not generally available on all Cray installations though. Access to CCM varies per system, requiring special flags to the job description or submitting to a special queue (RP hides those differences from the application). RP can operate in CCM with the Agent either external or internal to the created CCM cluster. When the Agent is external it uses "ccmrun" to start tasks. However, this approach still relies on ALPS and therefore has the same limitations. 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.

3) Open Run-Time Environment (OpenRTE/ORTE): The Open Run-Time Environment is a spin-off from the OpenMPI 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[43] that issue is resolved.
Figure 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"). This means that the setup of the DVM is a single ALPS interaction and that all task execution is then outside of the realm of ALPS. As RP is a Python application and ORTE is implemented in C, we choose to interface the two using the ORTE CLI. While this enabled us to push the envelope with more concurrent tasks and support the sharing of nodes between tasks, we did run into new bottlenecks. As every task requires the execution of "orte-submit", the interaction with the filesystem becomes a limiting factor for task execution. In addition, as every task requires a "orte-submit" instance that communicates independently with the "orte-dvm" we also run into network socket race conditions and system resource limits with workloads that consists of very large numbers of concurrent tasks. RP has the ability to spread the execution of tasks over multiple sub-agents (potentially running on separate compute nodes), which does alleviate the problem of having a large centralized process footprint for maintaining state about each running process this way.

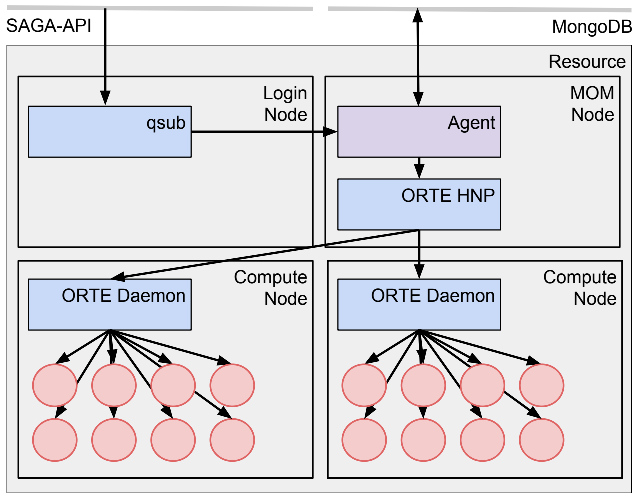


Figure 5: Architecture overview of RP with ORTE backend. The RP Client launches a Pilot using SAGA through the batch queue system. Once the job becomes active the Agent is bootstrapped on the MOM node. The Agent will use ORTE to launch a ORTE Head Node Process (HNP), and a ORTE Daemon on every Compute Node. The RP Client communicates tasks through the MongoDB to the Agent running on the MOM node. The Agent submits tasks to the HNP which forwards them to the respective ORTE Daemons running on the Compute Nodes. The ORTE Daemons are responsible for the fork() of the unitprocesses.

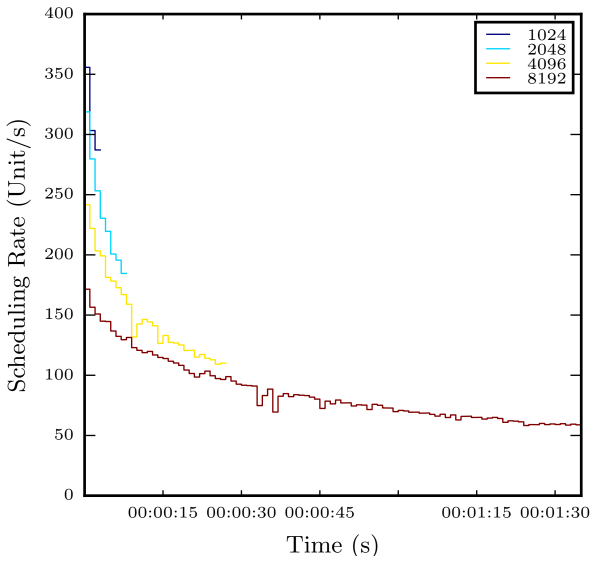
C Foreign Function Interface for Python (CFFI): CFFI [44] provides a convenient and reliable way to call compiled C code from Python using interface declarations written in C. From an ORTE perspective this mode of operation is similar to the CLI mode but differs in the way RP interfaces with ORTE. Instead of running a tool for every task to launch it only requires a library call. This also allows us to re-use the network socket, thus further decreasing the per-call overhead. The incentive for developing this approach was to overcome the limits in the CLI approach.

IV. Experiments

In the previous section we have described the general RP architecture and the specifics of various launch methods to execute units on *Blue Waters*. In this section we first look at the performance of individual components and then how these components perform in orchestration. All experiments are executed on *Blue Waters*, the XE6/XK7 system at NCSA.
It is often the case that there are more tasks than can be run concurrently. We then use the term *generation* to describe a subset of the total workload, that fits concurrently on the cores held by the pilot. For example, if we have 128 tasks of a single core that need to be executed on a 64 core Pilot, there will be twee generations of units. If each task is 42 seconds in duration, the optimal time to completion (ttc) would be 84 seconds (2 generations $\times$ 42 seconds).

A. *Micro benchmarks*

Micro-benchmarks measure the performance of individual RP components in isolation. In a micro-benchmark, RP launches a Pilot on a resource with a single unit submitted to the Agent. When the unit enters the component under investigation, it is cloned a specified number of times. All the clones are then operated on by the component and dropped once the component has completed its activity. This ensures that the downstream components remain idle. The result is that single components can be stressed in isolation, with a realistic workload, but without the influence of any other components.
Currently, RP can instantiate exactly one Scheduler component per Agent. The Scheduler is compute (and communication) bound: the algorithm searches repeatedly through the list of managed cores; core allocation and de-allocation are handled in separate, message driven threads. Figure 6 (top) shows how the component performs in allocating cores to a set of units for 4 different pilot sizes. The declining rate is explained by the algorithm and the implementation of the scheduler, as the scheduler needs to search further and further when more cores have been allocated to units. In Figure 6 (bottom) we show the same workload for the scheduler, but the results now also include the unscheduling of units and the freeing of the cores. We do not observe the slope from Figure 6 (top) anymore as the activity now becomes limited by the contention on the lock on the datastructure by both the scheduling and the unscheduling. The process of spawning and managing application tasks is central to the Agent's Executor component. Figure 7 shows



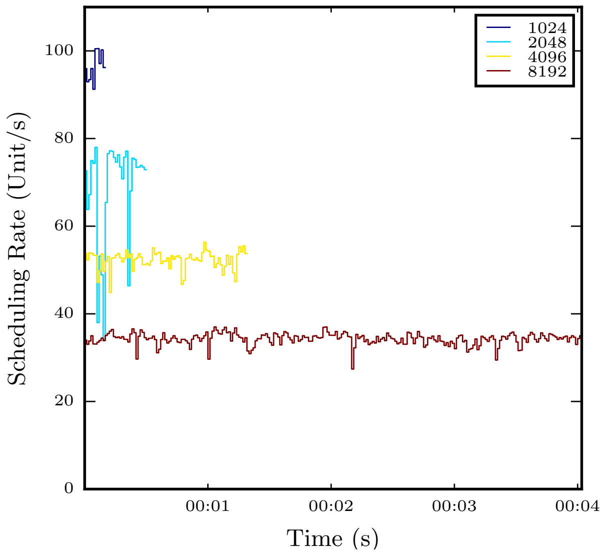


Figure 6: Micro-Benchmark performance of scheduling units on 4 pilots of different size. **(top)** Scheduling comprises allocating cores to a unit. **(bottom)** Scheduling comprises allocating cores to a unit and immediately freeing the allocated cores.

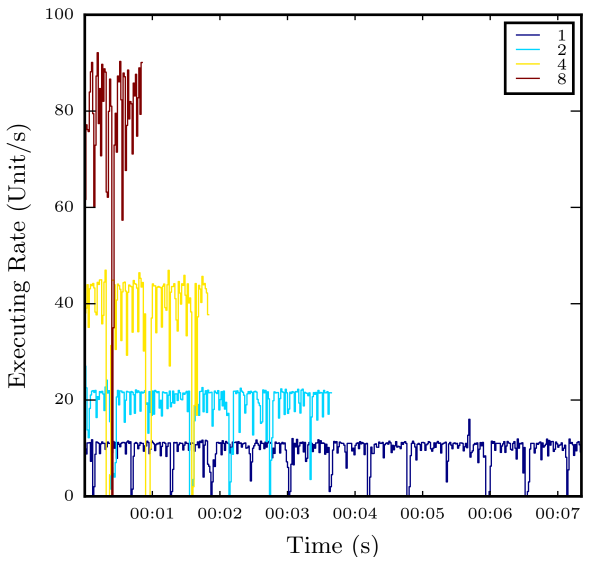
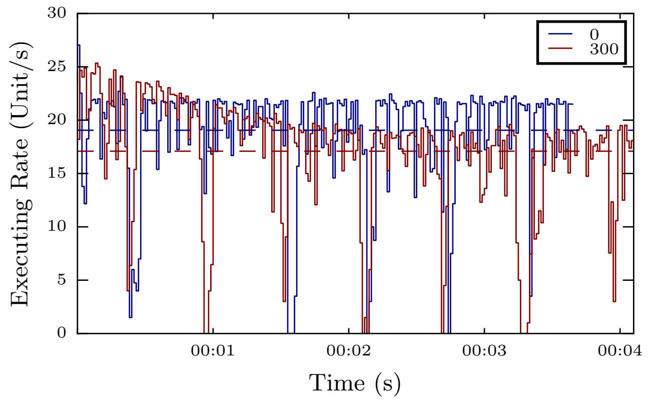


Figure 7: Micro-Benchmark Performance of executing units with ORTE-CLI. The pilot manages 4096 cores and a 4096 units are executed. Different colors show the execution for different sub-agent configurations. The respective number of sub-agents with single executor componenents are started on compute nodes.



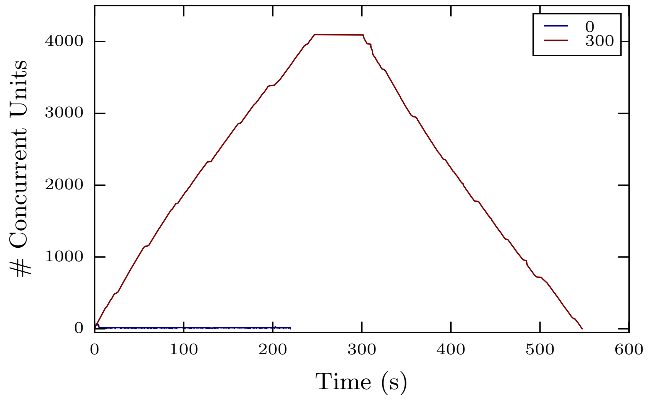


Figure 8: Micro-Benchmark executing units showing the effect of unit duration. The pilot manages 4096 cores and 4096 units are executed with ORTE-CLI and configured with 2 sub-agents. Two experiments were performed, with unit duration set to 0 and 300 seconds respectively. **(top)** The execution rate of both experiments. Horizontal dashed lines show the average execution rates for the two runs. **(bottom)** The unit concurrency that is achieved.

the scaling behavior of the ORTE-CLI launch method: the throughput scales with the number of sub-agents running on as many compute nodes. Note that this is an asynchronous process and does not mean that the overall throughput also scales linear. Not shown here is the trial with increasing the number of components per sub-agent, which caused no performance improvement, hinting that the limit is caused by RP interacting with the OS.

In Figure 8 we investigate the effect of unit duration on execution rate. The performance of the 300s run initially benefits from less contention because no units are finishing yet. Over time the rate of executing 0s units becomes higher than for the 300s run because the total number of concurrent units that are active in the latter case puts more strain on the system. As per the rate of 20 units/s from the top plot, it takes around 200s to start all 300s units. The 0s run achieves no concurrency, as the units run too short in comparion with the launch rate to build up any concurrency.

Even in the most optimal configuration the performance of ORTE-CLI is significantly less than the performance of the scheduler for a 1k pilot as seen in Figure 6, thereby creating a bottleneck at this stage.

Figure 9 shows the scaling of the ORTE-LIB launch method for different pilot sizes. For all sizes it appears stable over time, but more jittered than the scheduler micro benchmarks. This can be explained by the interaction with many external system components. In absolute terms the performance is lower than the scheduling component's, while similarly the performance decreases with increased pilot size.

While ORTE-CLI did not scale with multiple executor components per node, Figure 10 shows that for the ORTE-LIB launch method the performance does scale up to 4 executor components. Adding more sub-agents or components does not increase the performance further, as we reach the upper limit of the ORTE layer.

Figure 11 shows for varying pilot sizes the achieved unit concurrency. The initial slopes represent the launch rates. We can observe that launch rate is dependent on the pilot size. This difference is largely attributed to the ORTE layer.

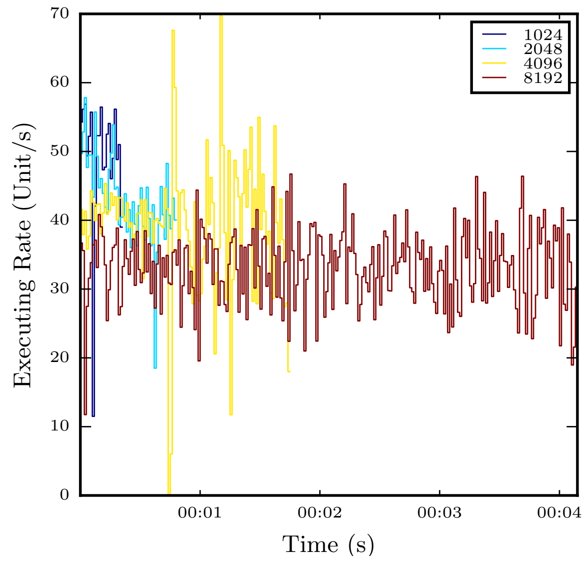


Figure 9: Micro-Benchmark of executing units with the ORTE-LIB launch method. The experiment is done for 4 different pilot sizes and a workload of one generation based on the size of the pilot. There agent is configured with a single executor.

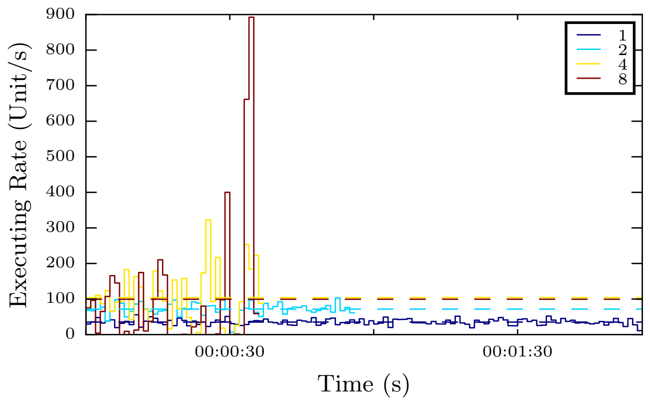


Figure 10: Micro-Benchmark Performance of executing units with ORTE-LIB. The pilot manages 4096 cores and a 4096 units are executed. Different colors show the execution for different agent configurations. One sub-agent is started on the MOM node with varying number of executor components.

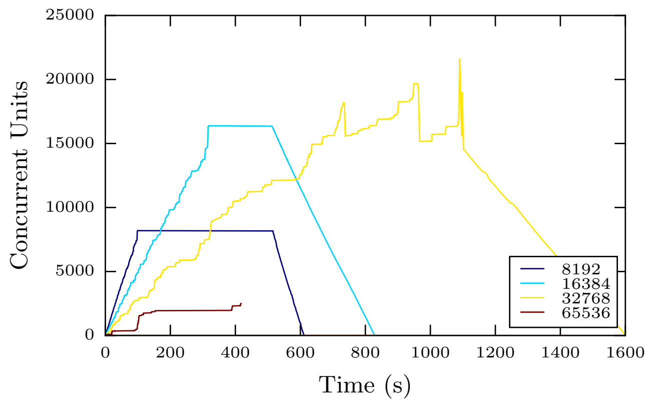


Figure 11: Micro-Benchmark observed unit execution concurrency as a function of pilot size. The workload for each experiment is constrained to one generation the size of the Pilot. All units run for 64 seconds and are executed using the ORTE-LIB launch method. The agent configuration is with one sub-agent on the MOM node with 4 executor components.

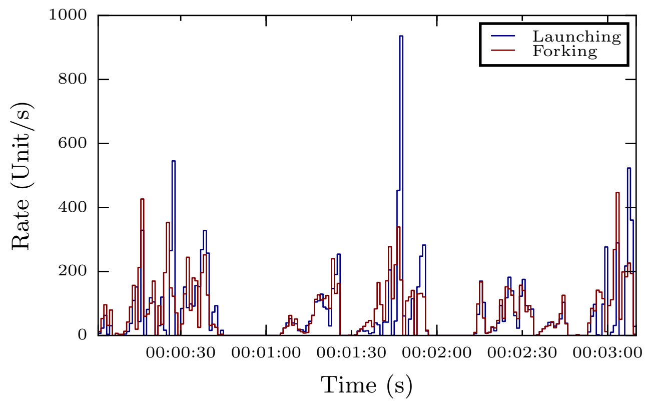


Figure 12: ORTE-only execution without RP. Workload is 3 generations of single core units on a 4k core pilot. "Launching" represents the time the callback into RP from ORTE that the unit has started and "Forking" represents the time the task is actualled forked on the compute node.

B. *ORTE-only Experiments*

In the discussion of Figure 9 we mentioned the interaction with external components. To isolate the interaction between RP and the ORTE layer we also conducted independent experiments with ORTE.

In Figure 12 we display the results from a 3 generation workload on a 4k pilot. The "Launching" (when RP is notified via a callback from the ORTE layer that the unit is started) and "Forking" (when the unit is started on the compute node) rates are a more detailed view of the "Executing" rate in Figure 9. The two are correlated, but

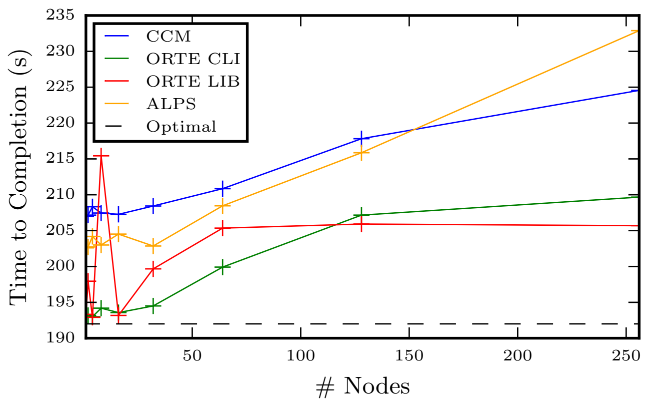


Figure 13: Time to Completion (TTC) for a 3 generation workload of full node units with a 64s duration for varying pilot sizes. The same workload is executed using RP with ALPS, CCM, ORTE-CLI and ORTE-LIB launch methods. All of the experiments use a single executer component. The theoretical optimal TTC is shown as reference.

we can see that the “Launching” is more erratic, which means that there is an offset in what RP perceives and the actual execution. This becomes especially relevant in the completion, as RP will keep the cores of a unit allocated until it is notified by ORTE that the unit is finished.

C. *Contrast to ALPS and CCM*

One of the limitations of ALPS/APRUN is that we can only run one unit per node. We also ran into SSH limitations with CCM when running more than 8 concurrent units per node. Although ALPS and CCM therefore do not satisfy all of the functional requiremens, we still want to have some baseline comparisons with ORTE. We therefore run a set of experiments where every unit consumes a full node.

Figure 13 shows the results of the full-node experiments. None of the experiments are optimized from a sub-agent and executer perspective, but still there is a large trend difference between ORTE-CLI/ORTE-LIB and ALPS/CCM. Note that these results do not include multiple runs for the same configuration and therefore we attribute some of unexpected results as outliers.

D. *Sub-Node Agent Experiments*

As stated earlier one of the limitations of ALPS/APRUN is that we can only run one unit per node, we therefore in this section do not include APRUN in the experiments. As we ran into SSH limitations with CCM when running more than 8 concurrent units per node, we also exclude CCM from further consideration and experiments. We thus focus on ORTE, specifically ORTE-LIB as that showed improvement over ORTE-CLI in section IV-A.

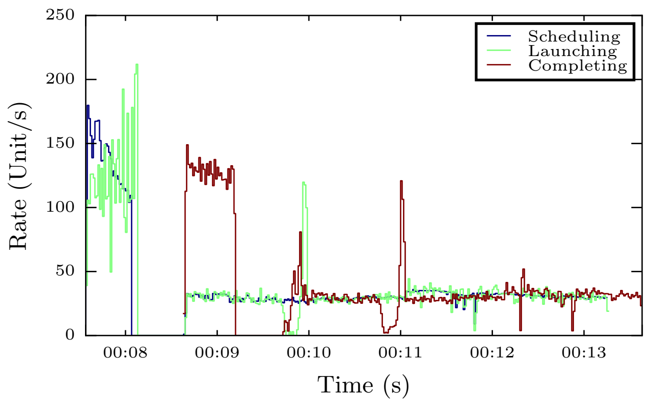


Figure 14: Agent Performance for Executing units using ORTE-LIB. Workload is 3 generations of single core units of 64s on a 4k core pilot. We display rates of Scheduling, Executing and Completing over time.

Figure 14 shows the agent performance for executing a workload of 3 generations of single core 64s units on a 4k core pilot with 4 executor components on the MOM node. It shows that the first generation of units is immediately scheduled and then the executer gets to work to launch all units. After 64s the first units are completed and we see the pattern of completion matching the execution pattern. The consecutive generations show a completely different performance characteristic though. We attribute the detoriation for later generations to the contention over the lock between the scheduling and the unscheduling that we discussed in §IV-A.

The rate of roughly 100 units/s in Figure 14 reflects in the slope of Figure 15. The latter figure shows the maximum concurrency thats achieved for various pilot sizes. The ceiling is caused by the rate of the launching and the duration of the units.

To build intuition into the efficiency of running a certain workload, we investigate the effect of the unit runtimes on the core utilization. The results are in Figure 16. For short unit durations, the launch overhead is relatively high, resulting in lower utilization at higher core counts. For longer running units the impact of the launching decreases, first for smaller core counts then for larger ones. All data points are singular runs and outliers are expected therefore.

E. *Discussion*

We started the experiments with the examining the RP Agent scheduler. This component is in principle easy to isolate from system dependencies. When we look at the performance of the scheduler and relate it to the execution benchmarks, we can conclude that the scheduler, or more specifically, the unscheduling, does become the bottleneck

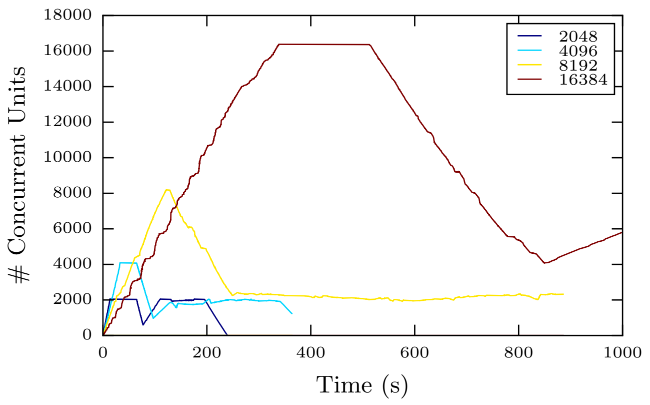


Figure 15: Observed unit concurrency as a function of pilot size and unit duration. The units are executed using the ORTE-LIB launch method with an agent configuration of 4 executor components on the MOM node. The workload for each experiment consists of 3 generations of single core units.The unit duration is tuned so that all units are started before the first one completes (respectively 64, 64, 128 and 256 seconds).

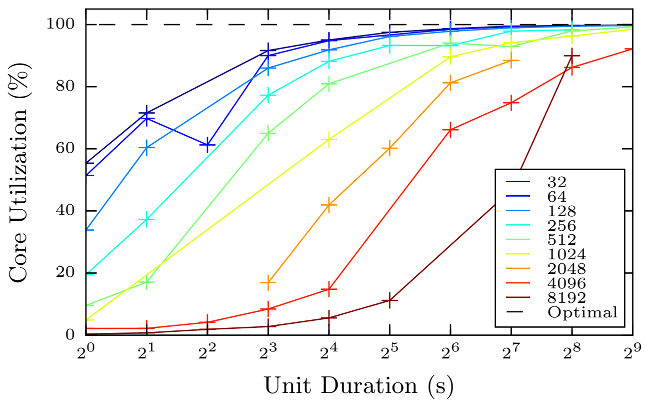


Figure 16: Core utilization as a function of task runtimes and pilot sizes. The workload for every experiment consists of three times the number of units than that would fit concurrently on the pilot. Results shown for the ORTE-LIB launch method. The configuration of the agent is with a single executor.

on pilots with higher unit counts.

In the executor micro-benchmarks we see the same scalability pattern, performance degrades when the pilot size (and thus unit count) increases.

Both ORTE-CLI and ORTE-LIB execution show scalability when adding more concurrent executor compontents. ORTE-LIB doesn't require more nodes for that, and scales by adding more compontens on the MOM node. This is explained by the fact that an execution through ORTE-LIB is only a library call that causes a network call and doesn't strain the system it is running on. In the micro-benchmarks we managed to operate 16k units concurrently; in the 32k and, especially, the 64k experiments the launch rate becomes too low, effectively throttling the maximum concurrency.

The additional ORTE-only experiments show that the executor micro-benchmarks with the optimized agent layout approaches the performance of the ORTE layer with a launch rate of about 100 units/s for a 4k pilot. In the full-node experiments the performance of CCM was unexpectedly low: RP also runs on normal clusters and we observe a higher throughput with the SSH launch method there. Because the degree of unit concurrency in the full-node experiments is relatively low, the ORTE-CLI and ORTE-LIB are quite comparable.

In [5] we provide more data and discussion of non-Cray machines and a full range of micro-benchmarks.

V. Conclusion

There is no "one size fits all" approach when it comes to HPC. While Cray systems excel at executing monolithic workloads, they are restricted in running more varied workloads. RP in combination with ORTE is a non-invasive userspace approach that enables the execution of workloads that exceeds the design objectives of Cray systems. By using RP on Crays we have overcome many of the task execution limitations. In this paper we showed running up to 16,000 concurrent tasks with a launch rate of around 100 tasks per second which exceeds native capability by orders of magnitude. Resource efficiency is largely dependent on the amount of units and unit duration and currently improvements are required to make executing more and shorter tasks viable. We are potentially trading-off some raw per-task performance as currently we can not run applications that are linked against the Cray MPI libraries. In the pre-ORTE era of RP, there was overlap with ORTE functionality. By leveraging the functionality of ORTE, RP can focus on the functionality that complements ORTE: the orchestration of tasks.

VI. Future Work

As identified the launching of tasks is currently the prime bottleneck towards higher utilization of resources via RP. For generic Agent performance improvement we have identified the following activities. The interface between RP and ORTE (for submission and notification) is currently on a per-unit basis. We intend to convert these interfaces to support bulks of units, to decrease the overhead per unit. Once we are able to pass units faster between the two layers we see two possible types of improvement within ORTE. The communication between the HNP and ORTE daemons currently runs over TCP. Work is underway to make direct use of the network fabrics for this inter-process communication. In addition to the transport of the messages, the topology of the inter-process communication in ORTE might also not be optimized for our usage mode and we want to experiment with different mechanisms.

In all the discussion in this paper resources equalled CPU cores. On systems like *Blue Waters* that have heterogeneous compute nodes, e.g. nodes with and without GPUs on them, workloads could benefit from a scheduler that is aware of this heterogeneity, and we intend to extend our scheduler to enable this.

The placement of units on *Blue Waters* currently assumes the nodes on a continuous space. For small units that is not a problem, but for relatively large units the placement might benefit from topology aware scheduling. RP's architecture supports modular schedulers and a topology-aware scheduler has been developed for IBM Blue Gene, we intent to extent this effort to other HPC resource types.

While many of the components in RP can have multiple instances, with the shown scalability improvements, the scheduler currently is the only singular component which prevents full partitioning of resources and thereby full parallel operation. We intend to also parallelize the scheduler. We are also working on wider bulk support for RP inter-component communication, which specifically is expected to releave the stress on the scheduler, as it allows for more coars grained locks on the internal data structures.

Acknowledgements

We thank Matteo Turilli for discussions and comments. This research is part of the *Blue Waters* sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. *Blue Waters* is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also part of the ”The Power of Many: Scalable Compute and Data-Intensive Science on Blue Waters”, PRAC allocation support by the National Science Foundation (NSF-1516469).

References

[1] J. Preto and C. Clementi, “Fast recovery of free energy landscapes via diffusion-map-directed molecular dynamics,” *Physical Chemistry Chemical Physics*, vol. 16, no. 36, pp. 19 181–19 191, 2014.

[2] T. E. Cheatham III and D. R. Roe, “The impact of heterogeneous computing on workflows for biomolecular simulation and analysis,” *Computing in Science & Engineering*, vol. 17, no. 2, pp. 30–39, 2015.

[3] Y. Sugita and Y. Okamoto, “Replica-exchange molecular dynamics method for protein folding,” *Chemical physics letters*, vol. 314, no. 1, pp. 141–151, 1999.

[4] M. Turilli, M. Santcroos, and S. Jha, “A comprehensive perspective on pilot-jobs,” 2016, (under review) http://arxiv.org/abs/1508.04180.

[5] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, “Executing Dynamic and Heterogeneous Workloads on Super Computers,” 2016, (under review) http://arxiv.org/abs/1512.08194.

[6] I. Raicu, I. T. Foster, and Y. Zhao, “Many-task computing for grids and supercomputers,” in *Proceedings of the Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. IEEE, 2008, pp. 1–11.

[7] Pilot API, 2015, http://radicalpilot.readthedocs.org/.

[8] A. Merzky, O. Weidner, and S. Jha, “SAGA: A standardized access layer to heterogeneous distributed computing infrastructure,” *Software-X*, 2015, dOI: 10.1016/j.softx.2015.03.001. [Online]. Available: http://dx.doi.org/10.1016/j.softx.2015.03.001

[9] A. Treikalis, A. Merzky, D. York, and S. Jha, “RepEx: A flexible framework for scalable replica exchange molecular dynamics simulations,” 2016, (under review) http://arxiv.org/abs/1601.05439.

[10] V. Balasubramanian, A. Trekalis, O. Weidner, and S. Jha, “Ensemble toolkit: Scalable and flexible execution of ensembles of tasks,” 2016, (under review) http://arxiv.org/abs/1602.00678.

[11] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[12] T. Maeno, K. De, A. Klimentov, P. Nilsson, D. Oleynik, S. Panitkin, A. Petrosyan, J. Schovancova, A. Vaniachine, T. Wenaus *et al.*, “Evolution of the ATLAS PanDA workload management system for exascale computational science,” in *Proceedings of the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013), Journal of Physics: Conference Series*, vol. 513(3). IOP Publishing, 2014, p. 032062.

[13] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, “Application-level scheduling on distributed heterogeneous networks,” in *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE, 1996, pp. 39–39.

[14] D. Thain, T. Tannenbaum, and M. Livny, “Distributed computing in practice: The Condor experience,” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[15] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, “Condor-G: A computation management agent for multi-institutional grids,” *Cluster Computing*, vol. 5, no. 3, pp. 237–246, 2002.

[16] J. T. Mościcki, “DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data,” in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, vol. 3. IEEE, 2003, pp. 1617–1620.

[17] P. Saiz, L. Aphecetche, P. Bunčić, R. Piskač, J.-E. Revsbech, V. Šego, A. Collaboration *et al.*, “AliEn: ALICE environment on the GRID,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 502, no. 2, pp. 437–440, 2003.

[18] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev *et al.*, “DIRAC pilot framework and the DIRAC Workload Management System,” in *Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series*, vol. 219(6). IOP Publishing, 2010, p. 062049.

[19] I. Sfiligoi, D. C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Würthwein, “The pilot way to grid resources using glideinWMS,” in *Proceedings of the World Congress on Computer Science and Information Engineering*, vol. 2. IEEE, 2009, pp. 428–432.

[20] R. Pordes *et al.*, “The Open Science Grid,” *J. Phys.: Conf. Ser.*, vol. 78, no. 1, p. 012057, 2007.

[21] J. Gyllenhaal, T. Gamblin, A. Bertsch, and R. Musselman, “Enabling high job throughput for uncertainty quantification on BG/Q,” ser. IBM HPC Systems Scientific Computing User Group (SCICOMP), 2014.

[22] J. Cope, M. Oberg, H. M. Tufo, T. Voran, and M. Woitaszek, “High throughput grid computing with an IBM Blue Gene/L,” *2007 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 357–364, 2007.

[23] “Nitro web site,” http://www.adaptivecomputing.com/products/hpc-products/high-throughput-nitro/.

[24] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, “Pegasus, a workflow management system for science automation,” *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[25] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahl, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[26] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, “Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids,” in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 1.

[27] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier *et al.*, “FireWorks: a dynamic workflow system designed for high-throughput applications,” *Concurrency and Computation: Practice and Experience*, 2015.

[28] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, “Work Queue + Python: A framework for scalable scientific ensemble applications,” in *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

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

[30] M. Hategan, J. Wozniak, and K. Maheshwari, “Coasters: uniform resource provisioning and access for clouds and grids,” in *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2011, pp. 114–121.

[31] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, and P. J. Maechling, “Enabling large-scale scientific workflows on petascale resources using MPI master/worker,” in *XSEDE ’12: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, Jul. 2012.

[32] J. M. Wozniak, M. Wilde, and D. S. Katz, “JETS: Language and system support for many-parallel-task workflows,” *Journal of Grid Computing*, 2013.

[33] “Mpich web site,” http://www.mcs.anl.gov/research/projects/mpich2.

[34] R. H. Castain and J. M. Squyres, “Creating a transparent, distributed, and resilient computing environment: the OpenRTE project,” *The Journal of Supercomputing*, vol. 42, no. 1, pp. 107–123, Oct. 2007.

[35] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, “Swift/T: Large-scale application composition via distributed-memory dataflow processing,” in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 95–102.

[36] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, “Turbine: A distributed-memory dataflow engine for high performance many-task applications,” *Fundamenta Informaticae*, vol. 128, no. 3, pp. 337–366, 2013.

[37] K. Maheshwari, J. M. Wozniak, T. G. Armstrong, D. S. Katz, T. A. Binkowski, X. Zhong, O. Heinonen, D. Karpeyev, and M. Wilde, “Porting ordinary applications to Blue Gene/Q supercomputers,” in *2015 IEEE 11th International Conference on e-Science (e-Science)*. IEEE, Aug. 2015, pp. 420–428.

[38] “Taskfarmer web site,” https://www.nersc.gov/users/data-analytics/workflow-tools/taskfarmer/.

[39] “Wraprun web site,” https://www.olcf.ornl.gov/kb_articles/wraprun/.

[40] “QDO web site,” https://www.nersc.gov/users/data-analytics/workflow-tools/other-workflow-tools/qdo/.

[41] “Mysge,” http://www.nersc.gov/users/analytics-and-visualization/data-analysis-and-mining/mysge/.

[42] “Python Task Farm,” http://www.archer.ac.uk/documentation/user-guide/batch.php#sec-5.7.

[43] “PMIx web site,” https://www.open-mpi.org/projects/pmix/.

[44] “CFFI Documentation,” http://cffi.readthedocs.org.