Supporting Many Task Workloads on Frontier using PMIx and PRRTE

Wael Elwasif
Computer Science & Mathematics Division.
Oak Ridge National Laboratory
Oak Ridge, TN, USA
0000-0003-0554-1036

Thomas Naughton
Computer Science & Mathematics Division.
Oak Ridge National Laboratory
Oak Ridge, TN, USA
0000-0002-3546-2382

***Abstract*—Large scale many-tasks ensemble are increasingly used as the basic building block for scientific applications running on leadership class platforms. Workflow engines are used to coordinate the execution of such ensembles, and make use of lower level system software to manage the lifetime of processes. PMIx (Process Management Interface for Exascale) is a standard for interaction with system resource and task management system software. The OpenPMIx reference implementation provides a useful basis for workflows engines running on large scale HPC systems.**

In this paper we present early experience using PRRTE/PMIx to manage the execution of many-tasks ensemble workloads on HPE Cray XE systems, namely the early access system *Crusher* at OLCF as well as the Frontier Exascale system. We outline important considerations when using the platform for achieving performance and highlight this alternative approach for user-driven task sub-scheduling (i.e., task scheduling within an existing job allocation). We report results from experiments run on Crusher and the Frontier Exascale system based on a synthetic many-task workload.

***Index Terms*—many-task, resource management, PMIx, exascale**

I. Introduction

Large scale many-tasks ensemble are increasingly used as the basic building block for scientific applications running on leadership class platforms [2]. Workflow engines are used to coordinate the execution of such ensembles, striving to optimize overall resource utilization and time to solution for scientific applications. Towards that goal, workflow engines make use of lower level system software to manage the lifetime of computational processes that constitute such ensembles. Process management can be accomplished using native (vendor provided) system software, or third party process management tools.

PMIx (Process Management Interface for Exascale) is a standard for interaction with system resource and task management system software [8], [9], with a reference implementation provided via Open PMIx used in the PMIx Reference RunTime Environment (PRRTE) project that underpins the OpenMPI project [7]. The use of PMIx as a standard process management layer for workflow engines has been hindered by the lack of viable Python binding to the API, with Python being the de facto standard for implementing many of the workflow engines in use on large scale HPC systems. Recently, a Python binding for PMIx was developed, and its use was demonstrated on the Summit supercomputer at ORNL.

Notice: This manuscript has been authored in part by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan

The PMIx specification offers a common interface for process management that hides many of the details of the native resource management system. In the context of the new HPE Cray XE systems (e.g., Crusher/Frontier and Sunspot/Aurora), there will be two different resource managers, SLURM and PBSPro respectively. The syntax for mapping/binding of tasks differs between these two resource managers. The PMIx specification offers a common interface that can be used to abstract the local resource management environment. The PMIx interface allows for tools that request launching of both serial and parallel applications, which may use the client interface to gather and exchange information about the parallel execution environment (Figure 1).

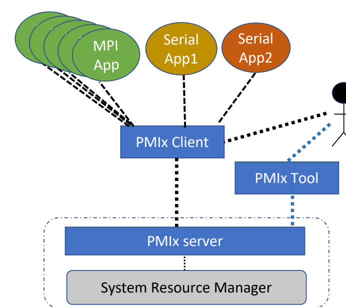

Fig. 1. PMIx interfaces (server, client, tool) that can be used to launch serial and/or parallel applications (e.g., MPI applications).

The many-task scientific application use cases require coordination among the ensemble of processes (serial and parallel) that comprise the application workload. The execution of these tasks can benefit from user-driven scheduling policies that drive the placement and order of the workload on the available resources. The emerging exascale systems at the Oak Ridge Leadership Computing Facility (OLCF) and Argonne

Leadership Computing Facilities (ALCF), Frontier and Aurora respectively, have complex compute node topologies, which require care be taken when placing tasks on the nodes. For example, on Crusher/Frontier a process on a given compute core has a higher affinity (higher bandwidth) to two of the eight GPUs and one of four network interfaces. The performance of the application will suffer if these affinity requirements are not observed when binding the resources (network, GPU, CPU) to the process. In the context of many-task workloads, the user-driven scheduling must be able to control these resource binding/mapping details.

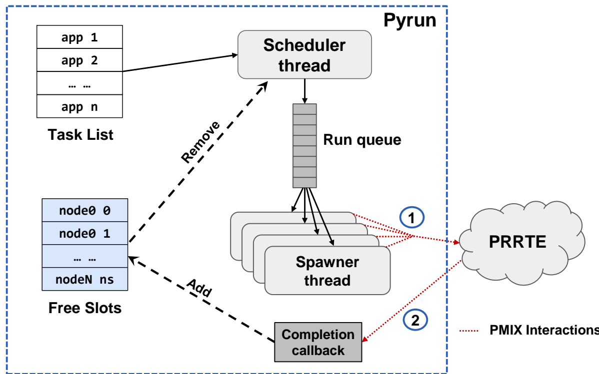

Fig. 2. Architecture of *Pyrun* driver prototype.

This paper builds on previous work [5], where we studied the behavior of the PMIx/PRRTE environment on Summit and the use of the PMIx Python client API to drive many-task workloads on that platform compared to the platform native LSF tools. In this work we present early experience deploying the PMIx/PRRTE software stack on Crusher/Frontier as a foundational layer for executing workloads based on many-task ensembles. We use a light weight user level scheduler prototype (Figure 2) to orchestrate the execution of many-tasks ensembles using the PMIx Python tools interface. We use a set of synthetic workloads including the NAS parallel benchmark to study the execution profile of this prototype. In particular, we explore the unique process placement challenges posed by the Frontier/Crusher node architecture, and the controls exposed to clients of both PMIx/PRRTE and SLURM that enable fine control of process placement and affinity that may be needed for workflows with complex task and data dependencies that may not be easily met by commonly used process binding and mapping patterns. The experiments will use the PMIx Python interface, which offers more fine-grained control and greater insight into the stages of the process lifecycle.

The contributions of this paper include a detailed description of the challenges many-tasks workloads may encounter when running on the new system. We present an overview of the new system and discuss binding/mapping challenges that will be important to make efficient use of resources. We present results from a synthetic workload containing mixed, scale, and duration to exercise the Pyrun/PRRTE approach for running ensembles of scientific applications on the system. We detail the setup and configuration for using the PMIx/PRRTE solution on Frontier and leverage current work (presented in separate paper) to enable Open MPI to take advantage of the shared memory (SHM) and Slingshot 11 (CXI) providers when running parallel applications.

II. PMIx

Process Management Interface for Exascale (PMIx) is a community driven standard [8], [9] that offers an application programming interface (API) to mask differences in native workload management systems. The PMIx standard extends the past PMI-1 and PMI-2 definitions [4]. The reference implementation of PMIx is provided by the OpenPMIx [7] project, which provides a library to access the client, tool and server interfaces defined by the PMIx standard. The OpenPMIx project also maintains a reference runtime environment that offers a portable PMIx server implementation, PRRTE (PMIx Reference RunTime Environment). The PRRTE implementation is the most feature complete version of a PMIx server, but other vendors have support for portions of the standard (e.g., SLURM, PBS Pro).

The primary purpose of PMIx is to offer a common programming interface for managing processes in parallel/distributed systems. There are three "roles" within PMIx [9]: client, server, tool. A tool is a supplementary utility that can be used to request new processes be created by a server, which may include co-location of processes to support the tool itself, e.g., parallel debuggers. The client and server interfaces provide the generic connection layer between the application (client) and workload management system (server). A PMIx application may be serial (single process) or parallel (coordinating process comprising a single application). A unique *job identifier (jobid)* is assigned to each application and is used to track its lifecycle (e.g., assigning resources, starting, terminating). A PMIx *namespace* is a string used to uniquely identify jobs and their constituent processes. The PMIx interface relies heavily on asynchronous events, which are used to signal new actions (e.g., launch job), supporting capabilities (e.g., input/output forwarding, signal delivery), and manage process lifecycle (e.g., process termination, process failures).

All of the PMIx APIs use a key-value approach for setting directives and qualifiers (attributes) that are used to tailor the behaviour of requested actions. For example, when launching tasks via `spawn`, the `PMIX_BINDTO` directive can be used to specify the appropriate resource binding when starting the processes (i.e., "*core*" to bind process to compute cores). The PMIx standard details the set of supported keys and attributes for the items in the interface. This key-value approach makes the interface highly flexible. This enables variations in the underlying resource management systems as to what features they choose to support. The callers can probe to determine supported features using a query interface or use a try/catch approach to backoff and retry. In this paper we focus on the PMIx Python bindings, which were introduced in v4.0 of the PMIx standard.

A. Pyrun and the PMIX Python client

In this paper, we use a Python driver prototype (Pyrun) to exercise the Python client API for PMIX, demonstrating its use to manage the execution of large ensemble of independent tasks on the Crusher early access system, and the Frontier CrayEX supercomputer. The PMIx Python API presents a thin wrapping layer on top of the native C API for PMIx, while maintaining the callback-based execution model for PMIx. The Python PMIx client supports multithreading, providing proper management for the Python GIL on entry into an exit from the OpenPMIx C library layer.

A client driver code using the PMIx Python API presents itself as a tool to the PMIx ecosystem. Listing 1 shows the initialization phase, where a tool instance (the Python driver) is initialized and connects to the server specified in a file (*dvm_file*); a callback handler (`done_cb`) is registered that is invoked when a job ends. The callback uses a condition variable `done_var` to notify other threads that maybe blocked waiting for tasks to complete (e.g. the scheduling/queuing engine).

```
1  def done_cb(evhdlr:int, status:int, source:dict, info:list, results:list):
2      with done_var:
3          done_var.notify()
4      return pmix.PMIX_EVENT_ACTION_COMPLETE,None
5
6  tool = pmix.PMIxTool()
7  rc,my_proc = tool.init( [{'key': pmix.PMIX_SERVER_URI,
8                            'value': "file:{}".format(dvm_file),
9                            'val_type':pmix.PMIX_STRING}] )
10
11 rc,handle = tool.register_event_handler( [pmix.PMIX_EVENT_JOB_END],
       None, done_cb)
```

Listing 1. Example of tool initialization and registration for job termination to trigger a `done_cb` callback that uses a condition variable to notify the Python program.

Instantiating new processes is done via the `spawn` API. This API can be used to create a single (serial) process, or multiple processes that can form a parallel cohort (e.g. by calling `MPI_init()`). Information that fully describe the code to be executed in the newly created processes is represented in an `app` data structure, while the desired placement, mapping, and binding behavior is expressed via the `info` data structure. The Python client API uses Python dictionaries (and list of dictionaries) as the main data type to represent such info, simplifying the implementation of application logic by avoiding the need for low level APIs used in the C interface.

```
1  job_info = [ {"key": pmix.PMIX_MAPBY,
2                "value": "core",
3                "val_type": pmix.PMIX_STRING},
4               {"key": pmix.PMIX_BINDTO,
5                "value": "core",
6                "val_type": pmix.PMIX_STRING} ]
7  exe = Path("/home/sgrundy/bin/sleeper").resolve()
8  app = { "cmd": str(exe),
9          "argv": [str(exe), "-n", "180"],
10         "maxprocs": 8,
11         "my_id": 1 }
12 rc, nspace = tool.spawn(job_info, [app])
```

Listing 2. Example of tool requesting to spawn an 8 process job that takes a command-line argument, with all processes mapped & bound to by cores.

Listing 2 shows an example of `tool.spawn()` that creates a job with 8 processes and *sleeper -n 180* as the command-line argument. The 8 processes in the job are mapped to the available resources based on compute cores, and each instance (MPI rank) is bound to a single compute core.

```
1  def iof_cb(iofhdlr:int, channel:int, source:dict, payload:dict, info:list):
2      buf = payload['bytes'][:int(payload['size'])].decode('UTF-8').
           strip()
3      # ...process buffer in Python as appropriate...
4
5  tool.iof_pull( [{'nspace':nspace,
6                   'rank': pmix.PMIX_RANK_WILDCARD}],
7                 pmix.PMIX_FWD_STDOUT_CHANNEL |
8                 pmix.PMIX_FWD_STDERR_CHANNEL,
9                 [], iof_cb)
```

Listing 3. Example of a tool requesting to have all `stdout` and `stderr` from a remote set of processes routed to the callback handler `iof_cb`.

Listing 3 shows an example of the tool requesting to receive the output from remote processes. When data from the remote processes is generated on the standard output and error file descriptors, the contents will be delivered to the registered callback handler (`iof_cb()`) where the data may be extracted from the buffer and processed accordingly by the calling Python process. The source of the data is also provided with the callback, so the data can be recorded on a per-job basis, which might be printed at the end or written to a common file in order to reduce the number of file descriptors. This facility can be particularly useful when managing the execution of a large number of tasks, by using a single file to store output from all tasks that can be processed postmortem into separate files per task and/or rank, alleviating the potential load on the underlying parallel file system from dealing with too many open files during job execution.

III. PMIx Server

A. PRRTE

The PRRTE package includes a PMIx server and tools to launch and manage PMIx jobs. PRRTE is the default runtime that will be included with Open MPI v5.0.0, which now requires PMIx for parallel process initialization. In Open MPI v5, the `mpirun` command is a thin wrapper around the PRRTE launch utility (`prterun`). There are two startup modes for PRRTE: one-time and persistent. The one-time mode has PRRTE daemons start and stop for each launch of a parallel application (i.e., each `mpirun a.out`). The persistent mode decouples daemon startup and allows a set of PRRTE daemons to be reused for multiple application launches. This persistent functionality is referred to as *Distributed Virtual Machine (DVM) mode* in PRRTE.

PRRTE uses a component infrastructure inherited from Open MPI to support a variety of capabilities. The process launch mechanism (PLM) framework is used to startup the runtime daemons PRRTE uses to manage the parallel execution environment. We use the `slurm` PLM for all tests.

B. Mapping/Binding

With PMIx, the placement of processes on computing resources generally involves two phases, a) logically mapping processes to available resource, and b) actually binding

the process to the hardware resources. These relate to the `PMIX_MAPBY` and `PMIX_BINDTO` a show in Listing 2.

We use the `ppr` format with PRRTE to express the desired mapping requirements. The syntax is `ppr:X:type:options`, where X is an integer value for number of instances for a given resource *type*. The *options* are a colon-separated list of qualifiers that can be used to tailor for specific uses cases. For example, `ppr:1:l3cache` states to map 1 process to each L3 cache on each available node, by default filling nodes in order. In this paper, we use the `:PE` option to further qualify the mapping with how many resources to assign to a given process when mapping. For example, `ppr:2:l3cache:PE=4` indicates that 2 processes are mapped to each l3cache, and assigned 4 cores each. This additional option is especially helpful when overlapping many tasks to the same nodes. This allows the mapping/binding to properly separate the processes on the compute nodes.

C. Affinity on Frontier

As compute nodes become more heterogeneous, it becomes even more important to pay attention to proper placement of resources. For example, on Frontier (Figure 3) there are four AMD MI250 accelerators per node, and each accelerator has two Graphic Compute Dies (GCDs) that present as eight GPU devices per node [6]. The bandwidth between different sets of GPUs and CPUs differs, with some have higher affinity (higher bandwidth). Additionally, there are four Slingshot 11 network interfaces per node, which have similar affinity characteristics to the CPUs and GPUs. This makes process mapping/binding for node resources more challenging and naive selection can result in poor performance.

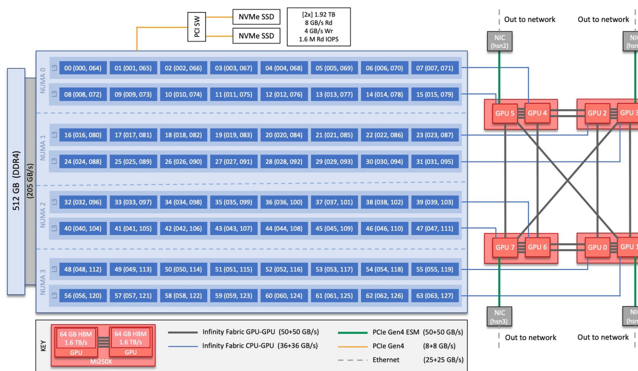

Fig. 3. Frontier compute node topology (source: OLCF)

Resource binding is often expressed in terms of CPU placement. As more heterogeneous systems like Frontier arrive, proximity of processes to certain devices becomes more important. A common method for setting the GPU affinity for a process is to use a "wrapper" script like in Listing 4, which sets the visible devices based on the process' CPU binding.

When using SLURM utilities directly to launch processes, the `srun` utility supports binding a task to the closest GPU (`--accel-bind=g`) or closest NIC (`--accel-bind=n`). There is not currently a command-line option exposed via

```
#!/usr/bin/bash
# Usage: mpirun ... ./gpuwrapper.sh ./a.out
declare -A gpumap
declare -A visible

function map_gpu() {
    local c0=$1
    local c1=$2
    local c2=$3
    local c3=$4
    local gpu=$5
    for c in $(
        seq $c0 $c1
        seq $c2 $c3
    ); do
        gpumap[$c]=$gpu
    done
}

#NUMA 0:
map_gpu 0 7 64 71 4
map_gpu 8 15 72 79 5
#NUMA 1:
map_gpu 16 23 80 87 2
map_gpu 24 31 88 95 3
#NUMA 2:
map_gpu 32 39 96 103 6
map_gpu 40 47 104 111 7
#NUMA 3:
map_gpu 48 55 112 119 0
map_gpu 56 63 120 127 1

echo -n "$(hostname) "
taskset -c -p $$

corelist=$(taskset -c -p $$ | awk '{print $NF}')
readarray -d , -t strarr <<<$(echo "$corelist")

unset ROCR_VISIBLE_DEVICES

length=${#strarr[*]}
for ((n = 0; n < $length; n++)); do
    entry="${strarr[$n]/$'\n'/}"
    readarray -d - -t cpus <<<"$entry"
    ntokens=${#cpus[*]}
    if [ $ntokens -eq 2 ]; then
        first=${cpus[0]}
        last=${cpus[1]/$'\n'/}
        for c in $(seq $first $last); do
            visible[${gpumap[$c]}]=1
        done
    else
        visible[${gpumap[$entry]}]=1
    fi
done

devices="${!visible[@]}"
export ROCR_VISIBLE_DEVICES=${devices// /,}
export HIP_VISIBLE_DEVICES=$ROCR_VISIBLE_DEVICES

echo "Use ROCR_VISIBLE_DEVICES=${ROCR_VISIBLE_DEVICES}"
exec "$@"
```

Listing 4. Example wrapper script to set the GPU visible devices based on the node layout of Frontier.

PRRTE to support binding in terms of the GPU or NIC. PRRTE does support the PMIx interface to query for hardware topology information, which could be used to create custom routines to select "closer" devices. Absent this support from the PMIx server, applications must determine the appropriate affinity based on CPU binding. For the many-task scenario, the "wrapper" script is the most straightforward way to insert affinity logic for GPUs and NICs based implicitly on CPU binding. This wrapper does not require any runtime library modifications and supports both serial processes and processes that are part of a parallel cohort.

D. Abstraction Layer

The PMIx interface can help to decouple the user from the native resource manager. This has an added benefit in the face of many-task workloads where the application mix can be rather varied (i.e., not a single parallel application) and the details for map/bind, I/O redirection, process lifecycle events, etc. is reduced to the that of the PMIx interface. This programmatic interface can improve productivity and reduce inefficiencies like ad hoc delays when using command-line only utilities that lack the higher level of system introspection (e.g., process startup & termination events).

IV. Experimental Results

In this section we present results of using the Pyrun driver prototype to execute many task workloads on the Crusher early access system as well as initial runs on Frontier. Two sets of workloads are used in these experiments, a synthetic workload used to characterize the ability of the Pyrun prototype and the PRRTE backend to orchestrate the execution of a large ensemble of independent tasks, and a workload based on the NAS parallel benchmark [1], [3] used to exercise the Pyrun/PRRTE prototype's ability to execute realistic MPI-based workloads. These workloads are explicitly configured to represent ensembles of sub-node independent jobs, a scenario that is becoming more common in HPC workloads as the trend towards *fat* more powerful nodes continue, and many codes are not able to scale up and make effective use of all node resources in a single instance.

The synthetic workload is based on a simple MPI ring buffer communication probe, where participating processes form a ring and pass a token around, then the tasks sleep for a random length of time passed as a command line argument. This exercises the ability of the PRRTE and OpenMPI runtimes to instantiate and execute basic MPI startup, shutdown and point-to-point communications for a large ensemble of independent tasks.

The workload based on the NAS benchmark is made up of an ensemble of 8000 randomly selected class B and class C tests from the MPI version of the NAS Parallel Benchmarks (NPB) 3.4.2 release. All tasks were run using 4 ranks for each job. This configuration was chosen to allow for maximum coverage as many of the tests in the suite have specific requirements for the size of the executing MPI program, while allowing for placing a meaningful number of jobs on the same node.

A. Crusher experiments: Synthetic workload

In these experiments, an ensemble of the synthetic *mpi sleeper* workload tasks was executed on different number of nodes. For each run on N nodes, the workload consisted of $ntasks$ jobs that sleep for a time interval uniformly distributed in the interval $[450, 550]$ seconds. The size of the ensemble depends on the size of individual jobs ($nranks$) and the number of nodes in the allocation. These experiments used the hardware threads on each node to allow for more tasks to be dispatched on the allocated nodes. The ensemble size ($ntasks$) was chosen to be 3X the number of tasks needed to completely fill the allocated nodes to test the ability of the Pyrun/PRRTE prototype to sustain task execution for a meaningful period of time. With 128 hardware threads per node, ensemble size $ntasks = 3 \times N \times 168/nranks$. So for an ensemble running on $N = 128$ nodes and dispatching $nranks = 2$ tasks, a total of $24,576$ tasks would be dispatched. In this set of experiments, the values of $1, 2, 4, 8, 16$ were used for $nranks$, resulting in $128, 64, 32, 16$ and 8 tasks per node, respectively.

Figure 4 shows the number of concurrent jobs executing for a subset of the node configurations, while Figure 5 shows the corresponding utilization history for the allocated nodes. The figures show that as the number of tasks in the workload and the size of the allocation grows, the Pyrun/PRRTE prototype struggles to maintain full utilization as it endeavors to process a high rate of task completion call backs while dispatching replacement tasks into the PRRTE DVM. This can be seen in the reported results, starting with $N = 76$ nodes and $nranks = 1$, utilization drops as the first *batch* of tasks terminate and the Pyrun driver attempts to replace them with queued tasks, eventually returning to full utilization before the pattern repeats before the ensemble enters the draw down phase where there are no queued tasks to submit. This pattern appears again for $N = 100$ where the Pyrun/PRRTE prototype is unable to achieve full utilization with $nranks = 1$, and again for $N = 128$, where for $nranks = 1$, the prototype achieves a maximum utilization of 77.35% before falling due to tasks completing. The inability to maintain full utilization prevents the prototype from completing all tasks in the ensemble before the batch allocation time expires, completing $39,135$ out of an ensemble of $49,152$ tasks within the 40 minutes allocation time, or 79.6%.

It should be however noted that despite the inability of the Pyrun/PRRTE prototype to sustain full utilization as the node count and ensemble size increase, it is able to successfully reach a maximum of $12,674$ concurrently executing tasks (for $N = 128$ and $nranks = 1$) before falling behind. Further improvements in the prototype and the PMIx/PRRTE backend should further improve the task dispatch performance and allow for even larger ensembles to be executed successfully.

B. Crusher Experiments: NAS Benchmarks

In this set of experiments, we use the Pyrun/PRRTE prototype to drive the execution of a workload composed of a 8000 4-rank tasks from the NASA Parallel Benchmark (NAS), class B and C problems. All experiments used a fixed mix of tasks as their input. These experiments illustrate the trade-off between increased utilization and optimal performance of individual tasks in an ensemble workload, and the impact on the execution time of the overall ensemble. We also demonstrate the use of the PMIx python client to control task binding and mapping behavior in PRRTE.

Figure 6 shows the time evolution of the number of active tasks for different task binding and mapping configuration, while Figure 7 shows total cores utilized over time for each configuration. Plots labeled `core` represent tasks that are

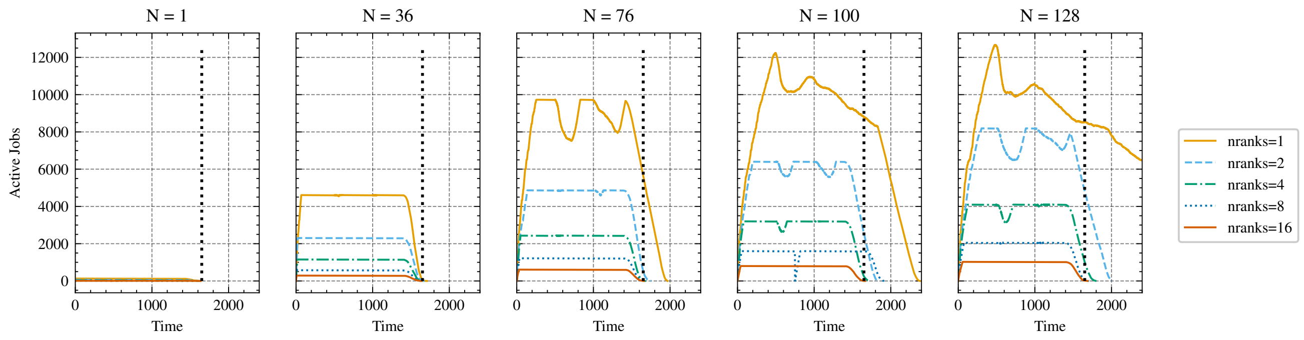

Fig. 4. Number of active jobs in Experiments on Crusher.

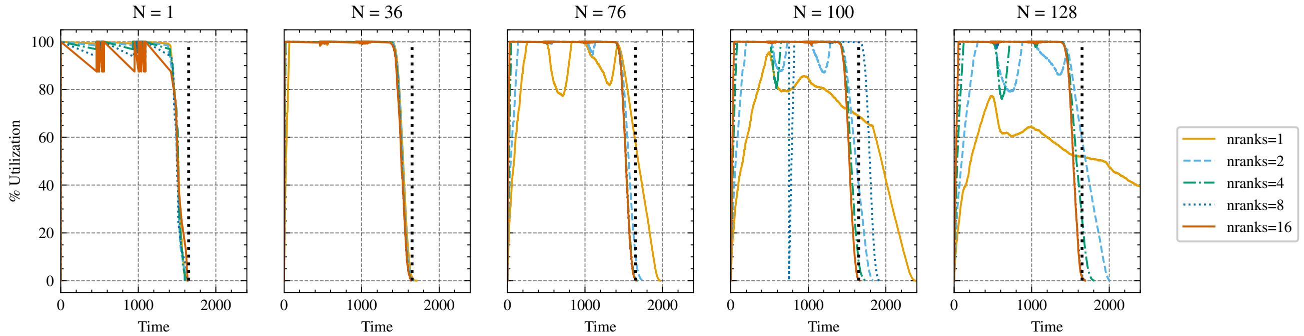

Fig. 5. Utilization percentages for Experiments on Crusher.

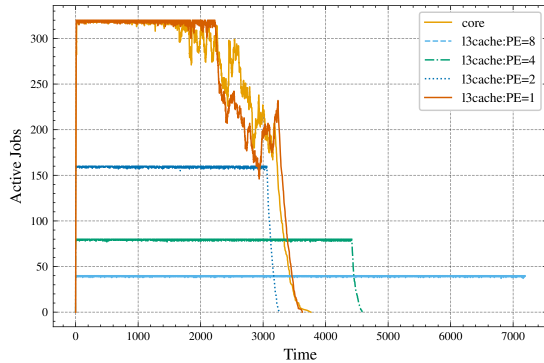

Fig. 6. NAS Benchmark, 20 Nodes, Active tasks

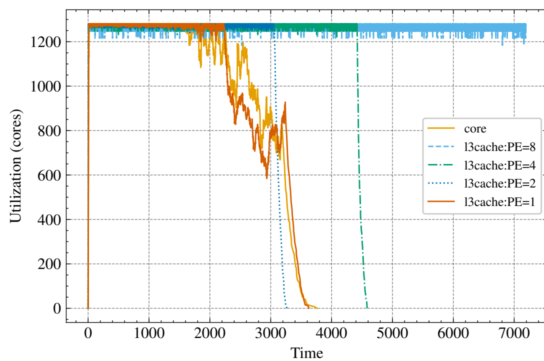

Fig. 7. NAS Benchmark, 20 Nodes, Active cores

mapped and bound to successive cores on each node. So the first 4-rank task to land on a node would occupy cores $0 - 8$ (as hardware threads are represented as physical cores), while subsequent tasks on the same node would occupy the following 8 cores. This results in 8 processes sharing each L3 cache on the node, 4 from each task. This is accomplished by setting the entries `PMIX_MAPBY` and `PMIX_BINDTO` to `"core"` in the `info` Python dictionary used to spawn a new PMIX task.

Plots labeled `l3cache:PE=N` correspond to experiments where each process in a task is mapped to `N` cores, and successive ranks are spread across L3 cache domains. So for `N=8`, each rank has dedicated access to all cores in an L3 cache domain, while for `N=2` each L3 cache domain is shared between 4 processes from 4 different tasks, each using 2 cores. This corresponds to setting `PMIX_MAPBY` to `"ppr:1:l3cache:PE=2"` and `PMIX_BINDTO` to `"core"` in the `info` Python dictionary used to spawn a new PMIX/PRRTE task.

Sharing the L3 cache between processes adversely impacts applications performance due to well known cache collision and eviction effects. This can be seen in Table I where the average execution time (μ), standard deviation (σ) and the coefficient of variation ($cv = \sigma/\mu$) for the individual benchmarks executed using `PE=8` and `PE=2`. The table shows that, as expected, sharing the L3 cache increases the average execution time for all tasks. The standard deviation and coefficient of variation also increase due to the unpredictable timing and exact task mix on any particular node. This increase however is balanced by the increased utilization of computational

TABLE I
L3 CACHE SHARING IMPACT ON NAS BENCHMARK WORKLOAD

Code	Ntasks		μ		σ		σ/μ	
	PE=8	PE=2	PE=8	PE=2	PE=8	PE=2	PE=8	PE=2
bt.B.x	470	477	37.37	61.45	0.94	8.25	0.03	0.13
bt.C.x	511	525	150.53	242.94	2.10	26.59	0.01	0.11
cg.B.x	533	536	8.03	18.20	0.21	5.22	0.03	0.29
cg.C.x	490	495	22.28	50.66	0.46	12.14	0.02	0.24
ep.B.x	508	514	9.68	10.83	0.14	0.31	0.01	0.03
ep.C.x	514	518	34.95	39.25	0.57	0.20	0.02	0.01
ft.B.x	476	481	8.00	12.37	0.12	1.53	0.02	0.12
ft.C.x	483	492	28.95	46.12	0.30	5.79	0.01	0.13
is.B.x	490	492	2.19	2.83	0.07	0.31	0.03	0.11
is.C.x	503	508	4.99	7.32	0.07	1.00	0.01	0.14
lu.B.x	491	492	22.47	32.89	0.32	2.84	0.01	0.09
lu.C.x	498	510	93.60	129.83	1.18	9.64	0.01	0.07
mg.B.x	489	492	2.17	3.20	0.08	0.46	0.04	0.14
mg.C.x	479	483	8.46	16.74	0.13	3.10	0.02	0.19
sp.B.x	472	474	23.66	54.89	0.31	11.99	0.01	0.22
sp.C.x	501	511	115.71	255.39	0.86	41.84	0.01	0.16

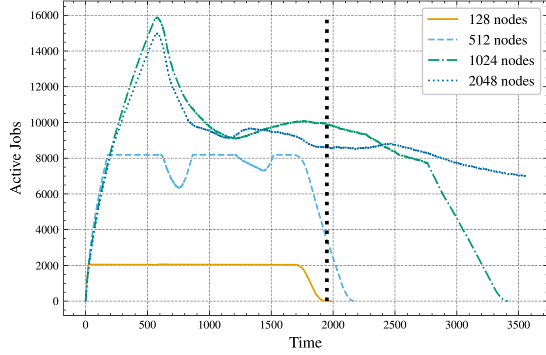

Fig. 8. 8 Active jobs on Frontier PE4.

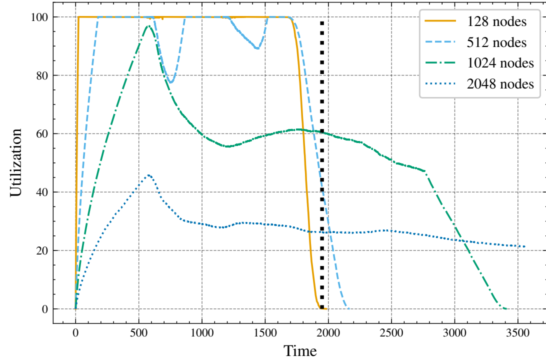

Fig. 9. 9 Utilization on Frontier PE4.

resources, leading to an overall reduction in the execution time of the entire ensemble. As a result, using PE=2 leads to the shortest overall execution time for the ensemble, while using PE=8 results in the simulation exhausting its allocation time before completing all tasks (resulting in different number of tasks in I).

C. *Frontier Experiments : Synthetic Workflow Scalability*

In this set of experiments, the synthetic MPI sleeper benchmark described in Sec. IV-A is deployed on Frontier to test the scalability of the Pyrun/PRRTE prototype when deployed at scale. For this set of runs, we chose a representative workload that assigns one or two tasks per GPU on each node in the allocation. This is done using an ensemble of single rank tasks, with sleep time uniformly distributed in the interval $[550, 650]$ seconds, with L3 cache binding and mapping as discussed in Sec. IV-A. With PE=4 two tasks map to the same L3 cache (and indirectly to the same logical GPU on the node), while using PE=8 results in single task per L3 cache/logical GPU on the node. For PE=8, Figures 10 and 11 shows the time evolution of the number of active tasks for different allocation sizes and the corresponding percentage utilization respectively. Figures 8 and 9 show the active tasks and utilization for PE=4.

The plots show the same pattern discussed earlier for runs on Crusher, with task launch overhead impeding the ability of the Pyrun/PRRTE prototype to achieve and maintain full utilization as allocation size and number of tasks in the ensemble grow. The plots also demonstrate the ability of the PRRTE runtime to scale up to 2048 Frontier nodes (or 21.7% of the machine size). At this scale, the Pyrun/PRRTE is able to sustain task man management and dispatch without error. When running on 2048 nodes, the prototype reaches a maximum of $15,126$ concurrent task (or 92.3% utilization) using PE=8. When running using PE=4, the prototype reaches a maximum of $14,978$ concurrent tasks (or 45.71% utilization). Further work to improve the task dispatch overhead is needed to allow better utilization at scale.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented our experience using the PMIx/PRRTE runtime to manage the execution of large

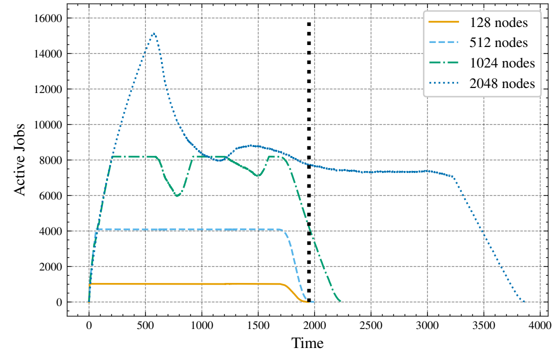

Fig. 10. 10 Active jobs on Frontier - PE8.

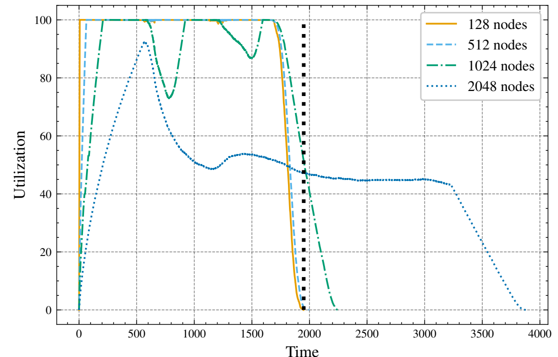

Fig. 11. 11 Utilization on Frontier - PE8.

ensembles of independent tasks on the Frontier Cray EX Exascale computer system and the Crusher early access system at OLCF. We used the PMIx Python client API via a prototype driver (Pyrun) to study the viability of PRRTE/PMIx as the runtime backend for the execution of massive ensembles of independent serial and parallel tasks. We evaluated the ability of the prototype and its scalability in launching massive ensembles and maintaining full utilization on a large size node allocations. We successfully deployed the Pyrun/PRRTE prototype to run task ensembles on 2048 Frontier nodes, reaching a maximum of 92.3% utilization with 8 tasks per node, or $15,126$ concurrent tasks. Experiments also revealed the scalability challenges of the prototype, as task dispatch overhead in the Python driver and the PRRTE backend prevents task ensembles on large node allocations from maintaining high utilization.

We plan future work to reduce the overhead and speed up the execution of large task ensembles in both the Python driver, the PMIx Python binding, and the PRRTE backend. Extending the Python driver to support dispatching tasks to multiple DVMs that partition a large allocation into smaller segments would allow faster overall task dispatch and improve overall utilization. Support for non-blocking task spawn in the PMIx Python binding would allow the driver to more efficiently submit more tasks into the PMIx/PRRTE backend. Finally, the PRRTE backend could be extended to reduce the overhead of executing tasks on a client-provided set of hosts, where it is up to the client to keep track of the occupancy state of each node in the allocation, and use that information to submit tasks to hosts with available resources, avoiding the need for the PRRTE backend to search for available slots with each new task submission.

While this work targeted the PRRTE implementation of the PMIx standard via the OpenPMIx library Python binding, the underlying principles and approach can be applied to any PMIx compliant resource manager/runtime that exports the standard PMIx interface. To the best of our knowledge, no other PMIx implementation exports the process management API used in the Pyrun prototype. Wider support for this API would provide a portable programmable abstraction to the process management functionality embedded in system resource managers, affording applications and workflow engines more flexibility and control beyond what is currently possible using a command line tool based approach. We hope this work motivates implementers to support and export a more complete subset of the PMIx specification.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] NAS parallel benchmarks, 2023. https://www.nas.nasa.gov/software/npb.html.

[2] Aymen Al-Saadi, Dong H. Ahn, Yadu Babuji, Kyle Chard, James Corbett, Mihael Hategan, Stephen Herbein, Shantenu Jha, Daniel Laney, Andre Merzky, Todd Munson, Michael Salim, Mikhail Titov, Matteo Turilli, Thomas D. Uram, and Justin M. Wozniak. ExaWorks: Workflows for exascale. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 50–57, 2021.

[3] David H. Bailey. NAS parallel benchmarks. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1254–1259, Boston, MA, 2011. Springer US.

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. PMI: A scalable parallel process-management interface for extreme-scale systems. In *Recent Advances in the Message Passing Interface*, EuroMPI. Springer, 2010.

[5] Wael Elwasif, Thomas Naughton, and Matthew Baker. Towards a standard process management infrastructure for workflows using Python. In *Parallel and Distributed Computing, Applications and Technologies*, pages 523–534, Cham, 2023. Springer Nature Switzerland.

[6] Frontier User Guide. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html.

[7] OpenPMIx: Reference Implementation of the Process Management Interface for Exascale. https://openpmix.org.

[8] PMIx: Process Management Interface for Exascale, 2022. https://pmix.org.

[9] PMIx-ASC. PMIx: Process Management Interface for Exascale Standard v4.1, 2021.