A Case Study on PMIx-Usage for Dynamic Resource Management

Dominik Huber[1](✉), Martin Schreiber[1,2], and Martin Schulz[1]

[1] Technical University Munich, Boltzmannstraße 3, 85748 Garching, Germany
{domi.huber,martin.schreiber,schulzm}@tum.de

[2] Université Grenoble Alpes, 621 Av. Centrale, 38400 Saint-Martin-d'Hères, France
martin.schreiber@univ-grenoble-alpes.fr

Abstract. With the increasing scale of HPC supercomputers efficient resource utilization on such systems becomes even more important. In this context, dynamic resource management is a very active research field, as it is expected to improve several metrics of resource utilization on HPC systems, such as job throughput and energy efficiency.

However, dynamic resource management is complex and requires significant changes to various layers of the software stack including resource- and process management, programming models and applications. So far, approaches for resource management are often specific to a particular implementation of the resource management and process management software, thus hindering interoperability, composability and comparability of such approaches.

In this paper, we discuss the usage of the Process Management Interface - Exascale (PMIx) Standard for interactions between the process manager and the resource manager. We describe an architecture that allows the resource manager to connect to the process manager as PMIx Tool to have access to a set of PMIx services useful for resource management.

In a concrete case-study we connect a python- and PMIx-based resource manager to PRRTE and assess the applicability of this architecture for debugging and exploration of dynamic resource management techniques. We conclude that a PMIx-based architecture can simplify the process of exploring new dynamic and disruptive resource management mechanisms while improving composability.

Keywords: Dynamic Resource Management · Process Management · PMIx

1 Introduction

HPC supercomputers are just entering the exascale era, which comes with a myriad of challenges such as energy efficiency, scalability of system software and resiliency [17]. To tackle these challenges, hardware technology, as well as system software and HPC applications need to be rethought under these new circumstances. In the area of system software, new ways have to be explored to make this software scalable and to improve the overall efficiency of resource utilization on the system.

In this context, hierarchical resource management software and dynamic resource management (where we consider malleability to be a subset of this) are very active research areas. In the last years, a large amount of research has been done ranging from theoretical work on dynamic resource scheduling [5,18,19] and simulations of dynamic scheduling strategies [8,9,20], to concrete implementations of mechanisms for dynamic resource management software [3,7,16,22,23] as well as runtime systems, programming models and applications (see, e.g., [4] for an overview of work). In these works, various benefits of dynamic resource management have been demonstrated. However, despite this large amount of research in this area, different dynamic resource management approaches often remain isolated from each other due to a lack of interoperability between the different solutions for the components in the system software stack as well as a high specialization on just one or a few applications.

To this end, we study the usage of PMIx [2] to increase the interoperability in the system software stack. PMIx is a programming interface standard providing an abstract interface for various services required on parallel and distributed systems, such as a distributed key-value store or an event notification system. Thus, PMIx can enable implementation-independent interactions between different system components. Despite providing an extensive API for the interaction between various components in the system software stack, so far PMIx has not been widely adopted. In most resource managers/process managers its usage is restricted to the small subset of its API required for bootstrapping parallel applications. One exceptions is the PMIx Reference Runtime Environment (PRRTE) [6], which provides a fully PMIx-enabled runtime environment. In the context of containerization techniques PMIx has also been used for the separation of application and runtime containers for running unprivileged HPC applications [15] and for enabling on-node resource management for containerized HPC workloads integrated into a hierarchical setup [24].

This work illustrates how PMIx could be used for portable interactions between the *Resource Manager (RM)* and the *Process Manager (PM)*. Here, we refer to the RM as the global system software responsible for managing and scheduling resources of the system. In contrast, we refer to the PM as the software managing the lifecycle of jobs and processes on the resources assigned by the resource manager.

We start by giving an overview of an architecture that allows PMIx-based interaction between the RM and the PM in Sect. 2. Then, in Sect. 3, we describe various possible PMIx-based interactions this architecture could provide for resource management, based on the specifications in the PMIx Standard 4.1 [21]. Subsequently, Sect. 4 presents a concrete case study of such a architecture, where we connect a Python- and PMIx-based dynamic RM to a modified version of the PMIx Reference Runtime Environment (PRRTE) to provide a testbed for dynamic resource management research. We conclude the paper by discussing and summarizing the key insights we gained from our case study in Sect. 5 and Sect. 6.

2 Overview of a PMIx-Based Architecture

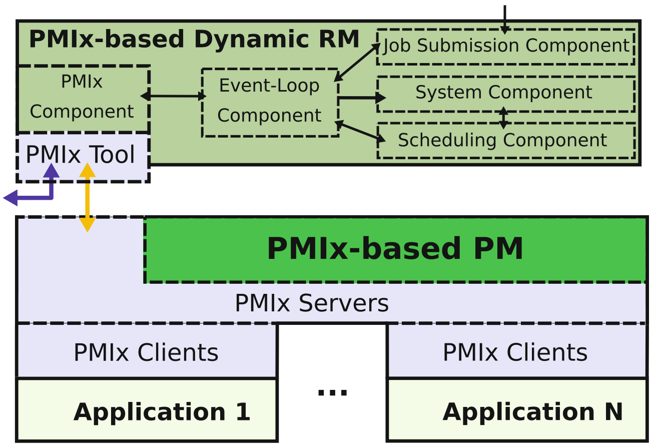


Fig. 1. Overview of an architecture for PMIx-based interactions between the Resource Manager (RM) and the Process Manager (PM). The bottom part represents the PM, which manages the execution of one or more applications. It operates PMIx servers to expose certain services to the application processes or other software components via the PMIx Client and PMIx Tool interface. The top part represents a RM manager consisting of various components. The PMIx component is responsible for managing the PMIx-based connection and interaction with the PM (yellow arrow) and possibly other system components (blue arrow) using the PMIx Tool interface. Note, that although the PM could potentially manage multiple applications across multiple nodes, the RM only needs to connect to one of the PM's PMIx servers, which facilitates the scalability of the approach. (Color figure online)

In this section we introduce a possible, basic architecture that enables PMIx-based interaction between the RM and the PM. An overview of this architecture is given in Fig. 1. The following subsections describe the three basic components in our architecture: PMIx, the PMIx-based PM and the PMIx-based RM.

2.1 PMIx

The central component in our architecture is PMIx. The PMIx standard defines PMIx as an "*application programming interface standard that provides libraries and programming models with portable and well-defined access to commonly needed services in distributed and parallel computing systems*" [21, p. 1]. This is achieved by providing an abstraction from the concrete system component using three generic roles: PMIx Server, PMIx Client and PMIx Tool. In the following we briefly introduce these PMIx roles.

PMIx Server: A process can initialize PMIx as PMIx Server using the `PMIx_server_init` function. Subsequently, PMIx Clients and PMIx Tools can connect to this PMIx Server, which gives them access to the PMIx services defined by the PMIx Standard. While some PMIx services can be serviced solely by the PMIx Server library implementation, usually interaction with the process that initialized the PMIx server library is required. To this end, during the initialization call, the process passes callback functions which enables the PMIx server library to pass on requests to be serviced by this process. The behavior of these callback functions is defined by the PMIx Standard, however, the concrete implementation needs to be provided by the hosting process. Thus, the PMIx server role provides an abstraction that can be used by system software components such as the PM to expose certain services.

PMIx Client: A process can initialize PMIx as PMIx Client using the `PMIx_Client_init` function, which establishes a connection with a local PMIx server. PMIx Clients are usually application processes launched by the process that hosts the PMIx server. Through the PMIx Client interface the launched processes have access to various PMIx services such as process synchronization and a distributed key-value store. In many MPI implementations MPI Processes use the PMIx Client role to interact with the MPI runtime environment, e.g., to exchange process wire-up information during MPI initialization.

PMIx Tool: A process can initialize PMIx as a PMIx Tool using the `PMIx_Tool_init` function. The PMIx Tools Interface is a superset of the PMIx Client Interface, providing additional functionality for establishing connections to PMIx servers and for I/O forwarding. PMIx Tools are able to connect to PMIx Servers on local as well as on remote hosts and are usually used for HPC tools, such as debuggers and launchers. We will use this interface for the connection between the RM and PM.

2.2 PMIx-Based Process Manager

PMs for distributed applications mange the lifecycle of applications and their processes. This involves launching and monitoring of application processes as well as process termination and cleanup. For this, PMs usually create an overlay-network of daemons on the compute nodes where the applications are running on. Beyond these basic functionalities, PMs often provide a varying degree of runtime services for applications.

Figure 1 (bottom) illustrates the concept of a PMIx-based PM. A PMIx-based PM is a PM that follows the PMIx Standard to provide its process management functionalities. For this, daemon processes usually initialize PMIx as a PMIx Server, i.e., the PM runs one PMIx Server per compute node. When launching applications, the application processes connect to the PMIx Server on their local node as PMIx Clients, giving them access to services of the PM exposed via PMIx.

2.3 PMIx-Based Resource Manager

The RM is responsible for the efficient execution of jobs on the system's resources. Figure 1 (top) illustrates the concept of a PMIx-based RM. A simple, PMIx-based RM could consist of the following components:

Event Loop Component: The event loop is the central component of the RM. It allows for thread-safe, serialized processing of events from different event sources. Events can be periodic, such as periodic invocations of the scheduling component, as well as non-periodic invocations, such as events triggered by notifications from the PM, e.g., to update the system state in the system component.

Job Submission Component: This component allows the submission of jobs to be executed on the system. While on production systems jobs usually are submitted *interactively* by users, for investigating novel resource management strategies it is preferable to *simulate* job submissions, e.g., by providing a job mix file with jobs and job arrival times. Submitted jobs are included in the job queue of the system component and are considered by the scheduling component.

PMIx Component: The PMIx component manages the connections and interactions with PMIx-enabled system software components. To this end, the PMIx Tools interface is used to connect, e.g., to the PMIx-based PM (dark yellow arrow in Fig. 1). This allows for various interactions, which are described in the next section. The PMIx Standard also allows tools to be connected to multiple PMIx servers. Thus, the PMIx component could possibly manage multiple connections (indicated by the blue arrow in Fig. 1), e.g., to multiple PMIx-enabled PMs or to a parent RM instance in a hierarchical resource management setup.

System State Component: The system state component provides a *representation of the current state of the system under management*. This includes for instance information about the usage of system resources such as compute nodes as well as information about currently running jobs and jobs in the job queue. Here, the PMIx component allows for tight interaction with other system software components to collect information about the current system state.

Resource Management Component: The resource management component is responsible for scheduling job execution and dynamic resource management during a job's lifetime based on the information of the system component.

To this end, an *instance of a resource management component* can be assigned to an *instance of the system state component*. This design allows to (dynamically) change the resource management strategy of the system and to facilitate the exploration of such strategies within a Python environment.

3 PMIx-Based Interactions for Resource Management

The architecture described in the last section provides the PMIx-based RM access to various PMIx functionalities of the PMIx Tool Interface. This section summaries some of these PMIx services and describes how they can be used by such an RM to interact with the PM. However, it is important to note that a PMIx-based PM might not provide support for all of these PMIx services listed below, or it might not support all of the attributes the PMIx standard defines for these services. For our case study in Sect. 4 we used a subset of these services.

3.1 Controlling Job Execution

An important area of interaction between the RM and PM are functionalities for controlling job executions. PMIx provides several functions related to job execution.

– `PMIx_Spawn`: Launches the specified number of instances of the given executable(s) on the specified resources. PMIx specifies various attributes that can be passed to further control properties of the job, such as I/O forwarding, event generation and logging. The RM can make use of this function to start jobs based on the scheduling decisions.
– `PMIx_Abort`: Aborts the specified processes. The RM could for instance use this function to abort jobs exceeding their time limit.
– `PMIx_Job_control`: Requests job control actions. Control actions include for instance, pausing, restarting or terminating processes, checkpointing and directory cleanup. The RM could use such control actions to enforce, e.g., fault tolerance or dynamic job co-scheduling strategies.

3.2 Retrieving System, Job and Process Information

Another requirement for the RM is to have access to system and application information relevant for the efficient management of the system resources. PMIx provides various functionalities to retrieve information about supported operations, system soft- and hardware as well as jobs, applications and processes. The RM can make use of these functions to retrieve information for its system state component, which can then be considered by its resource management component.

– `PMIx_Query_info`: Queries general information about the system. The PMIx Standard defines a rich set of query keys to be used to query system information. This includes queries for function and attribute supported by the PMIx implementation and host environment, names of running jobs, memory usage of daemons and application processes and statuses of jobs and application processes. The RM can make use of the query functionality to adjust its execution to the support level of the PM and to retrieve information for its system component.

- PMIx_Get: Retrieves key-value pairs associated with the specified process. This can be information about the process itself, as well as the session, job, application or node the process is associated with. The RM can use this functionality to retrieve information to be included in its system component, such as the mapping of a job's processes, or the memory size and CPU set of compute nodes.
- PMIx_Compute_distances: Computes relative distances from the specified process location to local devices such as GPUs, or network devices. This information could be included in the RM's system component to provide additional input to the resource management component, e.g., when dynamically assigning accelerators to jobs.
- PMIx_Fabric_(de)register: Registers access to fabric related information such as a cost matrix. The RM could include this information in its system component and use it in its scheduling decisions, e.g., to improve locality of communication.

3.3 Event Notification System and Logging

Finally, the last feature which is missing for the interaction between the RM and PM is the exchange of events. PMIx provides functionalities for event notification and logging. This allows the RM to record and react to changes of the system state and to inform the PM about reconfiguration decisions.

- PMIx_register_event/PMIx_Notify_event: Event Notifications are an important building block for the interaction between the RM and PM. Processes can use the PMIx_register_event function to register callbacks for particular event codes. The PMIx_Notify_event function can be used to send notifications of an event, thus triggering registered callbacks. The RM could for instance register callbacks for events triggered by the PM, such as job completion, node failures or reconfiguration requests. The PM could register callback for events triggered by the RM, such as job reconfiguration decisions.
- PMIx_Log: This function can be used to log data, such as events, to a data service. This can be useful to record and analyze the interactions between the RM and PM.

4 Case Study

In this section we present a case study of connecting a Python- and PMIx-based RM to a PMIx-based PM for debugging and exploration of dynamic resource management techniques. The code we develop and use in our work is included in a public repository [12]. The main objectives of this case study are a) to provide a proof-of-concept of a setup with a PMIx-based interaction between the RM and PM and b) reporting our experiences of using this setup to explore dynamic resource management techniques. We first describe our concrete setup and implementation in Sect. 4.1 followed by an evaluation based on a simple test case for dynamic resource management in Sect. 4.2.

4.1 Setup

In our setup, we closely follow the design described in Sect. 2 and we use a small subset of the possible interactions outlined in Sect. 3.

PMIx-Based Process Manager. For the PMIx-based PM we use a modified implementation of the PMIx Reference Runtime Environment (PRRTE). PRRTE is a fully PMIX-enabled PM for parallel and distributed applications and it is the native runtime environment of the Open MPI [11] implementation. It is capable of running and managing multiple jobs simultaneously, where each job can consist of multiple executables.

The PRRTE implementation we use in our setup is based on an implementation from prior work [10,13], which introduced dynamic MPI features to Open MPI, OpenPMIx and PRRTE. We extended the implementation from the prior version in two ways. First, we improve the flexibility of the dynamic MPI interface which will be described in a forthcoming publication. Second, we add support for connections with a PMIx-based RM. For this, we extend PRRTE's Resource Allocation Subsystem (RAS) framework and implement a RAS module for the interactions with our PMIx-based RM. This setup allows for single-application jobs to be reconfigured with node granularity based on the RM's resource management decisions, i.e., only full nodes can be added to or removed from applications.

Dynamic Resource Manager. We develop the (dynamic) RM entirely in Python to lay the fundament for future work on the fast prototyping of dynamic resource managers. So far we only investigate the basics of it. The user can specify the hostnames to be included in the managed system, the interval of the resource management loop and the file containing the job mix to be executed. At startup, it starts PRRTE using the

```
prterun --host hostnames --daemonize --mca ras timex --report-pid filename
```

command, which starts PRRTE on the specified hosts, daemonizes the DVM daemons into the background, selects our new RAS module and writes the PID of the PRRTE master process into the specified file. The design of the resource manager follows the component design from Sect. 2.3:

- **Job Submission Component:** The job submission component parses an input file containing a job mix. Each line is a job description in json format, containing the name of the job, the `mpirun` command to be executed and the arrival time.
- **PMIx Component:** The PMIx component makes use of the PMIx Python package that provides Python bindings for the PMIx Interface. PMIx is initialized as PMIx Tool and the connection to PRRTE is established using the PID reported by the `prterun` command. The interaction with PRRTE is based on only a small subset of the functionalities outlined in Sect. 3:
  - `PMIx_Spawn` is used to launch new jobs from the job queue.

  - `PMIx_Query` is used to query the names of compute nodes, running jobs and process sets.
  - `PMIx_Get` is used to query the mapping of processes on nodes.
  - `PMIx_register_event` is used to register callbacks for events such as job termination, process set definition, job reconfiguration requests and finalization of job reconfigurations.
  - `PMIx_Notify_event` is used to notify PRRTE about job reconfiguration decisions of the scheduler.
- **Event Loop Component:** For the event loop component the *asyncio* Python package [1] is used. We use a periodic event that checks for new submissions based on the information from the job submission component and executes the resource management policy. Based on the results of the resource management decision, it makes use of the PMIx component to launch new jobs or to communicate job reconfiguration decisions to the PM. Here, we like to briefly mention that the periodic event could lead to delayed resource management decisions, but could be replaced with events related to updates to the system-state components. However, this is not required in the present work.
- **System State Component:** The system state component stores a small set of information about the current system state. In our setup, the system state is characterized by the nodes available in the system, the jobs executing on these nodes, the processes included in each job and the current job reconfiguration requests. Moreover, the system state component provides a resource management function, which produces resource management decisions based on the currently assigned policy instance.
- **Resource Management Component:** The resource management component provides the resource management policy class with an abstract resource management function definition, allowing for different instances of such policies. A resource management policy instance can be assigned to the system state component dynamically.

4.2 Evaluation

We evaluate the applicability of our setup based on a concrete example. This example show-cases the usage of our setup in the context of debugging and exploring dynamic job reconfiguration and in particular resource management strategies.

Test System. As a test system we use a Docker-swarm based environment [14] to simulate the nodes of a compute cluster. For our tests we simulate a 4-node system, where each node has 8 cores and runs a CentOS based docker image. Such a docker based setup is optimal for fast debugging and exploration as it allows to simulate a multi-host environment on a local machine. Thus it is a convenient and efficient environment for the functionality experiments, which are the main goal of the present work. However, this setup is obviously not suited for any kind of realistic performance measurements, as it usually leads to oversubscription to simulate multiple nodes on a single machine.

Test Case. In our test case we compare the resource management of jobs exhibiting dynamically varying resource requirements with and without dynamic reconfiguration. We use a simple loop-based test application, which executes a 50 ms sleep command followed by an MPI Barrier in every iteration. In the static case (without reconfiguration) the number of processes remains the same for each iteration. In the dynamic case (with reconfiguration), the application can request a reconfiguration to add or remove processes before entering the next loop iteration. Here, we use a blocking approach, i.e., dynamic applications block until their reconfiguration request can be fulfilled. While such a blocking approach can obviously lead to deadlocks, it is sufficient for the functionality tests of the interactions between RM and PM, which are the main objective of this test case.

We execute test runs for two different job mix files containing two dynamic and two static jobs respectively. Table 1 lists the number of processes in each iteration for the dynamic and static jobs. In both cases, the arrival time for *job1* and *job2* is after 3.6 and 5.7 s, respectively. For the resource management we use a simple approach, which first attempts to start jobs from the job queue whenever there are enough system resources available. If no jobs can be launched, it checks if any job reconfiguration request can be fulfilled. The duration of the resource management loop is 0.3 s. We plan to replace this by events triggered by changes of the system state.

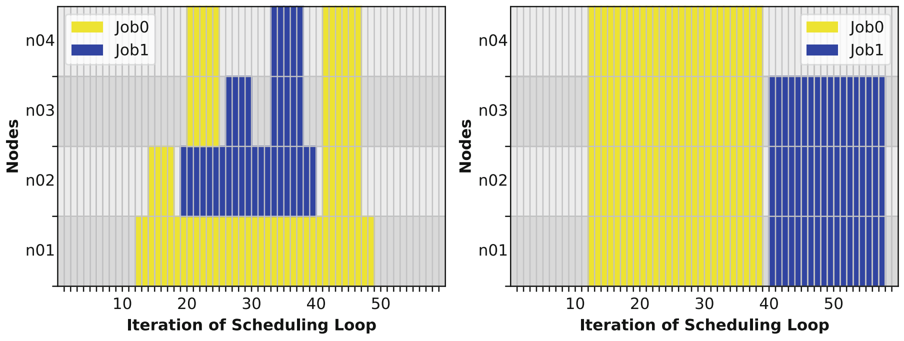
Table 1. Number of processes in each iteration.

Iteration	0–10	11–20	21–30	31–40	41–50	51–60	61–70
Job1 (dynamic)	8	16	8	24	8	32	8
Job2 (dynamic)	8	16	8	24	8	–	–
Job1 (static)	32	32	32	32	32	32	32
Job2 (static)	24	24	24	24	24	–	–

Results. Figure 2 shows a side-by-side comparison of the node occupation in the system at the beginning of each iteration of the resource management loop for the dynamic and static test run. In the static case *job1* occupies all nodes of the system during its entire runtime, thus preventing any other jobs to be started. In the dynamic case, *job1* grows step wise, which allows *job2* to be launched in iteration 19 and to be executed simultaneously. This allows for a higher job throughput in the dynamic case (37 resource management iterations vs. 46 resource management iterations). However, the individual time-to-completion of the jobs is higher in the dynamic case with 37 vs. 27 and 21 vs. 18 for *job1* and *job2* respectively. This is mainly due to the dynamic jobs blocking until resources are available for the requested reconfiguration. When comparing the node-hours (in terms of resource management iterations), the dynamic setup still provides benefits from the application perspective, with 69 vs. 108 and 35 vs. 72 iterations for *job1* and *job2*, respectively.

5 Discussion

Our case study demonstrated the applicability of the architecture described in Sect. 2 to enable PMIx-based interaction between the RM and PM and provides examples of concrete interactions for dynamic resource management using a subset of the functionalities described in Sect. 3. Based on the specifications in the PMIx standard, this architecture, in future work, can be integrated into a hierarchical resource management strategy, as described in Sect. 2.3.



(a) Test run with dynamic jobs. (b) Test run with static jobs.

Fig. 2. Node occupation during each iteration of the resource management loop using dynamic jobs (left) and static jobs (right).

We have shown how connecting a Python- and PMIx-based RM to a PMIx-based PM facilitates fast prototyping, debugging and exploration of dynamic resource management strategies. To this end, PMIx provides various benefits: First, it already provides a rich interface for managing and controlling resources and job executions, where we assessed its usability for at-least a basic PMIx-based setup for resource management. It maintains a separation-of-concerns between resource management and process management by acting as a messenger between these components. Thus, it reduces the time spent for implementing the communication protocol and allows to focus on exploring novel dynamic resource management strategies where we are convinced that these novel methods are required for convincing cases to make dynamic resource utilization a success. Second, it defines python bindings which allows for fast development and for the direct integration of python package for visualization, data analysis and mathematical models. This facilitates the debugging and comparison when exploring new techniques for dynamic resource management. Third, the usage of PMIx potentially allows for interchangeability of different RMs and PM that support PMIx, which could simplify the transfer from exploratory setups to production environments.

Our case study provided some valuable insight into various aspects of the usage of PMIx for resource management, however, it is also limited in some aspects:

Our tests were run inside of a docker environment and based on small-scale examples. While this is a convenient setup for development, debugging and prototyping, it can only provide limited insight into performance behavior and hardware related aspects. Thus, not all results from this setup are fully transferable to real HPC systems and large-scale experiments.

Moreover, while the specifications in the PMIx standard indicate that this architecture could be easily integrated into hierarchical resource management, in our case study we so far only provided a proof-of-concept for the case where the RM manages the resources assigned to the connected PM. A setup with deeper hierarchies remains to be demonstrated in future work.

6 Summary

The increase of the scale of HPC supercomputers introduces new challenges for system hardware and software. One of these challenges is to ensure efficient usage of the system's resources. To this end, scalable, dynamic resource management has gained significant interest over the last years. While there has been a large amount of work towards enabling dynamic resource management in different components of the system software stack, these attempts are often tailored towards particular implementations of these components and are often isolated from each other.

In this paper we investigated the usage of PMIx to facilitate portable interaction between the Resource Manager and Process Manager and to simplify the exploration of new resource management techniques. We first described a generic setup to enable a PMIx-based connection between the resource and process manager and describe possible interactions for resource management based on the PMIx Standard. We then described a case study based on a concrete implementation of the described setup and demonstrated its applicability to debug and explore dynamic resource management mechanisms and strategies. We see this as a proof of concept of a PMIx-based interaction between the resource manager and process manager which we expect to be an important building block for research on dynamic resource management.

In future work we will work on extending our implementation to support a hierarchical setup and use it to explore more sophisticated interactions and resource management strategies on real HPC systems.

Acknowledgements. We like to gratefully acknowledge Ralph Castain and Isaías Comprés for the discussions as part of the meetings of the PMIx Tools working group.

This project has received funding from the Federal Ministry of Education and Research and the European HPC Joint Undertaking (JU) under grant agreement No. 955701, Time-X and No. 955606, DEEP-SEA. The JU receives support from the European Union's Horizon 2020 research and innovation program and Belgium, France, Germany, Switzerland.

References

1. asyncio - asynchronous i/o. https://docs.python.org/3/library/asyncio.html
2. Castain, R.H., Solt, D., Hursey, J., Bouteiller, A.: PMIx: process management for exascale environments. Parallel Comput. **79**, 9–29 (2018). https://doi.org/10.1016/j.parco.2018.08.002
3. Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: a next-generation resource management framework for large HPC centers. In: 2014 43rd International Conference on Parallel Processing Workshops, pp. 9–17 (2014). https://doi.org/10.1109/ICPPW.2014.15
4. Aliaga, J., Castillo, M., Iserte, S., Martin-Alvarez, I., Mayo, R.: A survey on malleability solutions for high-performance distributed computing. Appl. Sci. **12**, 5231 (2022). https://doi.org/10.3390/app12105231
5. Bampis, E., Dogeas, K., Kononov, A., Lucarelli, G., Pascual, F.: Scheduling malleable jobs under topological constraints. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Los Alamitos, CA, USA, pp. 316–325. IEEE Computer Society (2020). https://doi.org/10.1109/IPDPS47924.2020.00041
6. PMIx Administrative Steering Committee: PMIx-based reference runtime environment (PRRTE) (2023). https://github.com/pmix/prrte. Accessed 21 Apr 2023
7. Dai, Y., et al.: Towards scalable resource management for supercomputers. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Los Alamitos, CA, USA, pp. 324–338. IEEE Computer Society (2022). https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://ieeexplore.ieee.org/document/10046103/&ved=2ahUKEwjb7PnelsGAAxXEOewKHfDhBpkQFnoECA4QAQ&usg=AOvVaw2gVV4Lq8_DZ9_6NCvwbPNC
8. Dupont, B., Mejri, N., Da Costa, G.: Energy-aware scheduling of malleable HPC applications using a Particle Swarm optimised greedy algorithm. Sustain. Comput. Inform. Syst. **28**, 100447 (2020). https://doi.org/10.1016/j.suscom.2020.100447
9. Fan, Y., Rich, P., Allcock, W.E., Papka, M.E., Lan, Z.: Hybrid workload scheduling on HPC systems. CoRR abs/2109.05412 (2021)
10. Fecht, J., Schreiber, M., Schulz, M., Pritchard, H., Holmes, D.J.: An emulation layer for dynamic resources with MPI sessions. In: HPCMALL 2022 - Malleability Techniques Applications in High-Performance Computing, Hambourg, Germany (2022). https://hal.science/hal-03856702
11. Graham, R.L., Woodall, T.S., Squyres, J.M.: Open MPI: a flexible high performance MPI. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 228–239. Springer, Heidelberg (2006). https://doi.org/10.1007/11752578_29

12. Huber, D.: Dynamic processes development repository. https://gitlab.inria.fr/dynres/dyn-procs
13. Huber, D., Streubel, M., Comprés, I., Schulz, M., Schreiber, M., Pritchard, H.: Towards dynamic resource management with MPI sessions and PMIx. In: Proceedings of the 29th European MPI Users' Group Meeting, EuroMPI/USA 2022, pp. 57–67. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3555819.3555856
14. Hursey, J.: PMIx docker swarm toy box (2021). https://github.com/jjhursey/pmix-swarm-toy-box
15. Hursey, J.: A separated model for running rootless, unprivileged PMIx-enabled HPC applications in Kubernetes. In: 2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 36–44 (2022). https://doi.org/10.1109/CANOPIE-HPC56864.2022.00009
16. Iserte, S., Mayo, R., Quintana-Ortí, E.S., Beltran, V., Peña, A.J.: DMR API: improving cluster productivity by turning applications into malleable. Parallel Comput. **78**, 54–66 (2018). https://doi.org/10.1016/j.parco.2018.07.006
17. Lucas, R., et al.: DOE advanced scientific computing advisory subcommittee (ASCAC) report: top ten exascale research challenges (2014). https://doi.org/10.2172/1222713
18. Marchal, L., Simon, B., Sinnen, O., Vivien, F.: Malleable task-graph scheduling with a practical speed-up model. IEEE Trans. Parallel Distrib. Syst. **29**(6), 1357–1370 (2018). https://doi.org/10.1109/TPDS.2018.2793886
19. Nagarajan, V., Wolf, J., Balmin, A., Hildrum, K.: Malleable scheduling for flows of jobs and applications to MapReduce. J. Sched. **22**(4), 393–411 (2018). https://doi.org/10.1007/s10951-018-0576-y
20. Özden, T., Beringer, T., Mazaheri, A., Fard, H.M., Wolf, F.: ElastiSim: a batch-system simulator for malleable workloads. In: Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022. Association for Computing Machinery, New York (2023). https://doi.org/10.1145/3545008.3545046
21. PMIx Administrative Steering Committee (ASC): Process management interface for exascale (PMIx) standard version 4.1 (2021). https://pmix.github.io/uploads/2021/10/pmix-standard-v4.1.pdf
22. Prabhakaran, S.: Dynamic resource management and job scheduling for high performance computing. Ph.D. thesis, Technische Universität Darmstadt, Darmstadt (2016). http://tuprints.ulb.tu-darmstadt.de/5720/
23. Schreiber, M., Riesinger, C., Neckel, T., Bungartz, H.-J., Breuer, A.: Invasive compute balancing for applications with shared and hybrid parallelization. Int. J. Parallel Prog. **43**(6), 1004–1027 (2014). https://doi.org/10.1007/s10766-014-0336-3
24. Vallee, G., Gutierrez, C.E.A., Clerget, C.: On-node resource manager for containerized HPC workloads. In: 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp. 43–48 (2019). https://doi.org/10.1109/CANOPIE-HPC49598.2019.00011