2019 IEEE/ACM Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)

On-node resource manager for containerized HPC workloads

Geoffroy Vallee
Sylabs, Inc; USA
geoffroy@sylabs.io

Carlos Eduardo Arango Gutierrez
Universidad del Valle; Colombia
carlos.arango.gutierrez@correounivalle.edu.co

Cedric Clerget
Sylabs, Inc, USA
cedric@sylabs.io

***Abstract*—This document presents a new containerized architecture to enable fine-grain control over the management of on-node resources for complex scientific high-performance workloads. Our approach is introducing a node-local, application-specific resource manager by extending a container runtime, which can coordinate with the global resource manager, i.e., the system-wide manager that assigns resources to jobs. The proposed work is based on the extension of a container runtime to interface running containers with global resource managers, as well as the implementation of advanced resource management capabilities to address all the running application's needs.**

Based on this design, the various runtimes that are required for the execution of scientific applications can interact with the container runtime under which it is running. This interaction enables the scalable and dynamic allocation of resources based on runtime requirements, in opposition to job-level requirements that are traditionally handled by the global resource manager. It also enables fine-grain control over the placement of all processes and threads running in a container on specific hardware components, which is critical to achieve performance. Our approach therefore enables an efficient, scalable, dynamic and trackable management of resources on behalf of scientific applications; bridging a gap observed with current solutions.

Our design leverages the PMIx standard [1], [2] in order to interface with the global resource manager and guarantee inter-library coordination to get resource requirements from running applications. Leveraging the PMIx standard allows us to have a generic and practical solution that can be used with most resource managers used by the HPC community, as well as some traditional runtimes such as MPI and OpenSHMEM.

***Index Terms*—containers, resource management, runtimes, MPI**

I. Introduction

With the rise of exascale, the High-Performance Computing (HPC) landscape is drastically changing, from the hardware to the programming languages. For instance, accelerators are nowadays considered as commodity hardware and applications switched from a pure MPI [3] to a hybrid model (a.k.a., *MPI+X*). These changes are increasing the complexity of the entire software stack, from the operating system to the application itself.

Fortunately, new technologies have emerged to help address these challenges: container technologies make it easier to develop applications, "package" them in a container and run them on different platforms; while low-level standards such as PMIx are used in the context of various execution runtimes (e.g., MPI) to help achieve scalability. The exascale computing project (ECP) [4], including DOE facilities, has spent tremendous amount of effort to accelerate adoption of HPC through container software technologies for HPC leadership computing platforms. However, from a resource management point-of-view, the design of HPC systems have not followed these trends. For example, it is still assumed that resources are allocated when the job is scheduled for execution, with very limited capabilities for the addition of extra resources at runtime. As a result, HPC applications are still considered to be *static* through their usage of resources. This assumption is contradicting emerging needs for flexibility, will it be for resilience purposes or because applications switch to modular architectures for which their needs in term of resources cannot be fully know at job submission time (e.g., composed and dynamic applications).

In this document, we propose a new architecture that enables reproducibility of HPC applications, a high-level of security, including for sensitive data, as well as a fine-grain advanced management of resources to fit at best the applications' needs. This is achieved by extending the Singularity container runtime to include PMIx support for both the interaction with the global resource manager and programming languages runtimes (such as MPI and OpenMP [5]). In other words, with our proposed design, the container runtime acts as an internal on-node resource manager for applications running in containers. This new role for container runtime is fairly natural since the container runtime is already in charge of managing the execution of application in containers in a safe and scalable manner.

The rest of the document is organized as follows. Section II presents the proposed architecture, while Section III presents an overview of related work. Finally, Section IV concludes.

II. Architecture

Our goal is to provide a local resource manager that act on behalf of the applications running in containers. This requires to have a software component that runs on the compute nodes that can interact with the global manager. By interacting with the global resource manager, it is possible to manage resources that are already allocated for the job, as well as request and manage new resources allocated at run time. This requires our local resource manager to interact with the runtimes of the various programming languages that are using by the applications running in the containers, such as a MPI or OpenMP.

978-1-7281-6028-3/19/$31.00 ©2019 IEEE
DOI 10.1109/CANOPIE-HPC49598.2019.00011

A. Requirements

Based on this, our design constraints can be summarized as follows:

1) **Security**, we aim at executing applications in a secure manner where the data that are used can be encrypted with the application;
2) **Reproducibility**, the application and the entire software stack that is needed are provided with the application and can be executed on compatible hardware, in opposition to "port" the application to the target HPC platform;
3) **Interaction with the global resource manager**, all resources allocated for the execution of applications should be accounted for and tracked by the global resource manager;
4) **Interaction with application's runtimes**, all scientific applications rely on various programming environments (e.g., MPI, OpenMP) which allocate execution contexts (processes and threads) on designated resources; our design aims at interacting with these runtimes.

1) Security and reproducibility: In order to provide security and reproducibility capabilities, we propose to rely on containers. We propose an architecture based on the Singularity container technology since it is already the container solution that is primarily used on HPC systems. Container technologies are known to enable better reproducibility by "packaging" applications and its data into an image. By using Singularity, it is also possible to benefit from the *Singularity Image Format* (SIF), an immutable format that packages the application, its data and the entire software stack that is needed.

SIF images can also be encrypted using a RSA public key. Upon execution, the private RSA key is used to decrypt the image using the system device mapper. This is a standard approach to provide system-wide encryption because it ensures that the decrypted image is not stored on the file system, i.e., provides one of the most secure method to handle encryption.

Finally, Singularity also provides a mechanism for signing and verifying images. This capability lets users authenticate images and as a result, ensure that images that are executed can be trusted. By providing both signing and encryption of images, it is possible to implement a chain-of-trust from the creation of the image (and therefore the installation and configuration of the workload as well as its data in the the container image).

2) Global resource manager interaction: Most HPC systems rely on a global resource manager. This manager is in charge of allocating resources to applications. Traditionally, most resource managers do not allow applications to dynamically allocate resources at run time, especially outside of the initial allocation that was allocated to the job. In other words, most global resource managers perform a static and upfront allocation of resources. However, with the venue of exascale, the HPC community is switching to architectures based on the usage of various programming environments and other techniques such as application composition. These new trends make it difficult for users to know precisely what resources are necessary upon job submission and, as a result, a dynamic allocation of resources is required.

In the context of this project, we intent to provide a local resource manager that can allocate local resources on behalf of applications and still interact with the global resource manager. To all intents and purposes, this means that we privilege a hierarchical approach where the global resource manager oversees how much resources is used by applications, and a local resource manager for the dynamic allocation of resources.

To achieve that goal, we propose to rely on the PMIx standard that is developed and widely used by HPC vendors, facilities and the associated research communities.

3) Application's runtime interaction: The various runtimes used by applications, such the MPI or OpenMP runtimes, have traditionally been designed and implemented without the constraint of coordinating with other runtimes. They have been developed in silos. Runtimes usually assume that all local resources allocated on the compute node can by default be used, while the intent of the users might be for the various runtimes to share or partition the available resources. Previous work has been done for the coordination of such runtimes but to the best of our knowledge, our design is the first to integrate such coordination into a local resource manager that can interact with the global resource manager.

To achieve this goal, we propose to rely on the PMIX standard once again. Using PMIx has two main advantages: (i) it is a compatible approach with what the U.S. DOE ECP project is currently investigating; and (ii) it is already the defacto standard for the interaction between the global resource manager and runtimes.

B. Architecture overview

Figure 1 presents an overview of the proposed architecture. It highlights the PMIx integration which enables both the interaction with the global scheduler, as well as the coordination with the various runtimes running in the container. For illustration purposes, the MPI and OpenSHMEM [6] runtimes are both included. Other runtimes could be considered as well. For example, the research community, especially the OMPI-X project [7] funded by the U.S. DOE is focusing on runtime coordination using PMIx in the context of hybrid applications (especially MPI+OpenMP applications [8]).

By having the Singularity PMIx thread in addition of the PMIx-compliant runtimes, it is possible to coordinate all the software components involved in resource management and therefore be able to implement end-to-end resource management strategies.

The rest of this section presents details about both PMIx and important details relevant to the management of resources.

1) State of the PMIx standard: Singularity, like most solutions available, rely on kernel namespaces for the instantiation of containers. The PMIx standard also provides a concept of *namespace* and while the semantics of these two namespace capabilities are different, both aims at partitioning resources. As a result, we do not expect to have to extend the PMIx

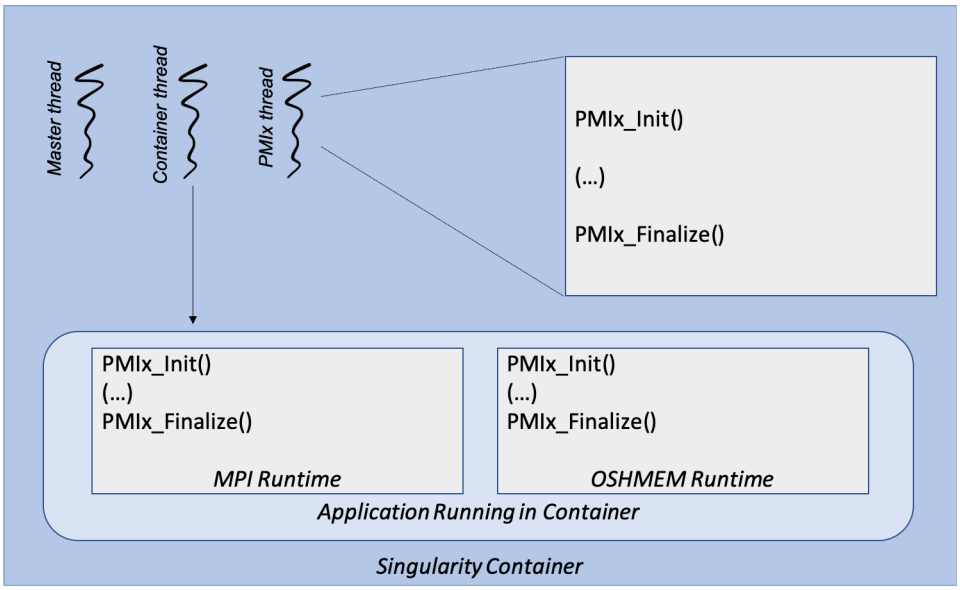

Fig. 1. Architecture overview.

namespace support for a use in the context of containers, and we expect to be able to full support containers and applications running in containers without modifications to the standard. Even if the standard does not need to be modified, an implementation of the standard will need to be extended for a full support of containers. This is mainly due to the fact that the compute entity within PMIx is based on processes and namespaces, which in a traditional environment translates into processes and other capabilities such as *cgroups*. Thus, PMIx implementations usually directly fork processes, while in our case, a native support of singularity will be required to start containers. Furthermore, this would allow the resource manager to ensure that container images are available and efficiently staged throughout the system, without creating extra burden on the users.

Similarly, PMIx provides an interface to facilitate the connect of debugging and profiling tool to running parallel applications, namely a *tool interface*. By using PMIx in the container runtime, Singularity will therefore benefit from a standard tool interface for the global debugging and profiling of processes and threads running in containers. While we do not believe that the PMIx standard needs to be extended to support Singularity, a PMIx implementation needs to be extended to fully support such a tool interface. This is mainly due to the fact that the processes running in a container are visible on the host in a way that differs from when they run outside of a container. An option to address this limitation is to extend both Singularity and a PMIx implementation to maintain a map between the process signature running on the host (all processes running in a container appears in */proc* but differently from normal processes) and the process running in the container.

We will also leverage existing PMIx interface and capabilities such as the support for both static and dynamic allocations. For example, PMIx defines *PMIX_ALLOC_NEW* and *PMIX_ALLOC_EXTEND*, which respectively specify that a new allocation has been requested and that an existing allocation is extended in terms of time or additional resources. These concepts can be reused to support advanced resource management capabilities which will rely on containers for the deployment. An intuitive example is to consider resource assigned to a container and applications running in containers as "allocations". When the required resources for a container are known in advanced, a container can be assigned a *static* allocation; while it is assigned a *dynamic* allocation when resources requirements are known only at run time. This gives an elegant way to describe the resource required by a container and ultimately partition available resources between containers, processes and threads that will be running on a single host.

In conclusion, based on the latest version of the PMIx standard, all required PMIx functionalities are already available for our purpose.

2) Extensions to Singularity: When running a container, the Singularity runtime creates two threads: (i) the *Master* thread and (ii) the *container* thread. The master thread instantiates the container runtime while the container thread becomes the application after executing *execve*. In order to implement the PMIx integration, the Singularity runtime will create an additional thread. This thread is designed to instantiate the PMIx library, i.e., call at minimum the *PMIx_Init()* function to initialize PMIx and *PMIx_Finalize()* upon termination.

The PMIx thread behaves as both a client and a server. It acts as a PMIx client in regards to the closest PMIx server running either on the node or in the context of the global resource manager. This design aims at easing the integration with most configurations that can be found while executing scientific applications. For instance, MPI implementations

such as Open MPI can create a daemon on the compute nodes that is also a node-local PMIx server. Another example is infrastructures like the IBM job and resource manager on systems such as Summit at Oak Ridge National Laboratory that create a persistent PMIx daemon on compute nodes in order to guarantee scalable start-up of workloads.

Having a PMIx-compliant container runtime enables three major capabilities: the precise mapping of resource to containers and applications running in containers; the efficient and scalable allocation of resources, and finally, inter-runtime coordination.

a) Resource mapping: The execution of HPC workloads, in order to perform, requires a careful placement of execution contexts (processes and threads) on available resources. This placement is often referred as "mapping" and in the context of MPI, traditionally done at job deployment time, through the interaction between the system's resource manager and the MPI runtime. A PMIx-compliant container runtime allows us to precisely describe and control the mapping of processes and containers on the various hosts composing the system. Our solution therefore enables the execution of use cases that are otherwise extremely difficult to fully support. Practically, our approach enables the execution of native workloads (where no container is used), container-based workloads (where only containers are used) but also "hybrid" workloads that are based on containers and processes running outside of any containers. This means that a mapper that would support containers could precisely define where processes, containers and processes within a container need to be executed, down to the granularity of cores or hardware threads.

b) Resource allocation and management: As previously stated, the PMIx thread also acts as a server for the application(s) running in the containers. This ensures that resources allocated to the job are known to the container runtime and applications mapped to the adequate resources, based on users' requirements. If applications request additional resources, the PMIx thread will either directly assign resources if resources are already allocated to the job or be relayed to the global resource manager if additional resource are required. In other terms, the PMIx thread acts as a resource management proxy between the applications and the global resource manager. This ultimately creates a hierarchical set of PMIx servers and clients, which is a preferred option in the context of HPC to guarantee scalability and performance. With this hierarchy of servers, it is possible to efficiently implement a precise mapping between applications and available resources. In fact, this extends what PMIx already does in the context of MPI. A direct result of such an approach is to enable process and thread binding throughout the entire stack and through the boundaries of containers but in a safe and secure manner. When the container's runtime forks processes and threads on behalf of the application, these processes and threads can be mapped and bounded to specific hardware resources. Similarly, devices such as accelerators can precisely be assigned to a container and eventually to a specific software component that is running in a container. This provides a level of control over resource management that is not currently possible with existing solutions. Finally, since PMIx is both a standard and an implementation that provides a programming interface, this opens the door to more advanced and complex structures where the applications running in containers, the container runtime and global resource manager could negotiate for resources. While such approach would enable the interesting concept of *internal resource manager*, this is out-of-the-scope of this document and, to the best of our knowledge, no such applications currently exist.

Figure 2 illustrates how a PMIx-aware runtime is used when executing a workload relying on containers. In the example, we can see that the hierarchy of PMIx servers is used to distribute the resource mapping assigned to the workload down to the container runtime, which can then create and bind processes to the appropriate resources.

c) Process debugging and profiling: Parallel debuggers [9], [10] and profilers are tools that are often used to analyse and optimize parallel applications. To ease the use of such tools, various "tool interfaces" have been developed and implemented over the years [11]–[13]. PMIx, as previously mentioned, also provide a tool interface and by having a PMIx thread in the Singularity runtime, it is possible to provide a standardized way to attached to processes running in containers. However, these tools usually require to parse and detect processes running on the host to know precisely to which process to attach. Tools usually rely on entries in '/proc' to do so. In the context of containers, the entries in '/proc' are different. It is still possible to see processes running in containers from the host but they appear differently and the standard process hierarchy is hidden (when creating a new container, a new namespace is created that is attached to the top process on the host, creating its own process 1) To address these differences, it is possible to create a map between the process running on the host and the processes running in the containers. That map can then be used by tools such as debuggers to attach to the processes running in a container. Of course this still requires the tool to have to required software components available in the containers, but it also gives the opportunity to have an end-to-end management of processes from the frontend node to containers running on compute nodes. As a result, it makes it possible to have a parallel tool monitor and attach to processes of a parallel workload regardless of whether the processes are running in a container.

d) Runtimes coordination: PMIx is basically a building-block for the design and implementation of asynchronous distributed system software. To achieve this, PMIx is event-based and provides a distributed key-value store for data sharing. A typical example is MPI startup: basic information about the job and hardware configuration is placed in the distributed key-value store. When a new MPI rank starts on a compute node, it retrieves all the required data to bootstrap network communications and fully initialize the MPI library. In the context of inter-library and inter-runtime coordination, the approach is similar. All runtimes call *PMIx_Init()*, which will share a *namespace*, even if called in the context of a single

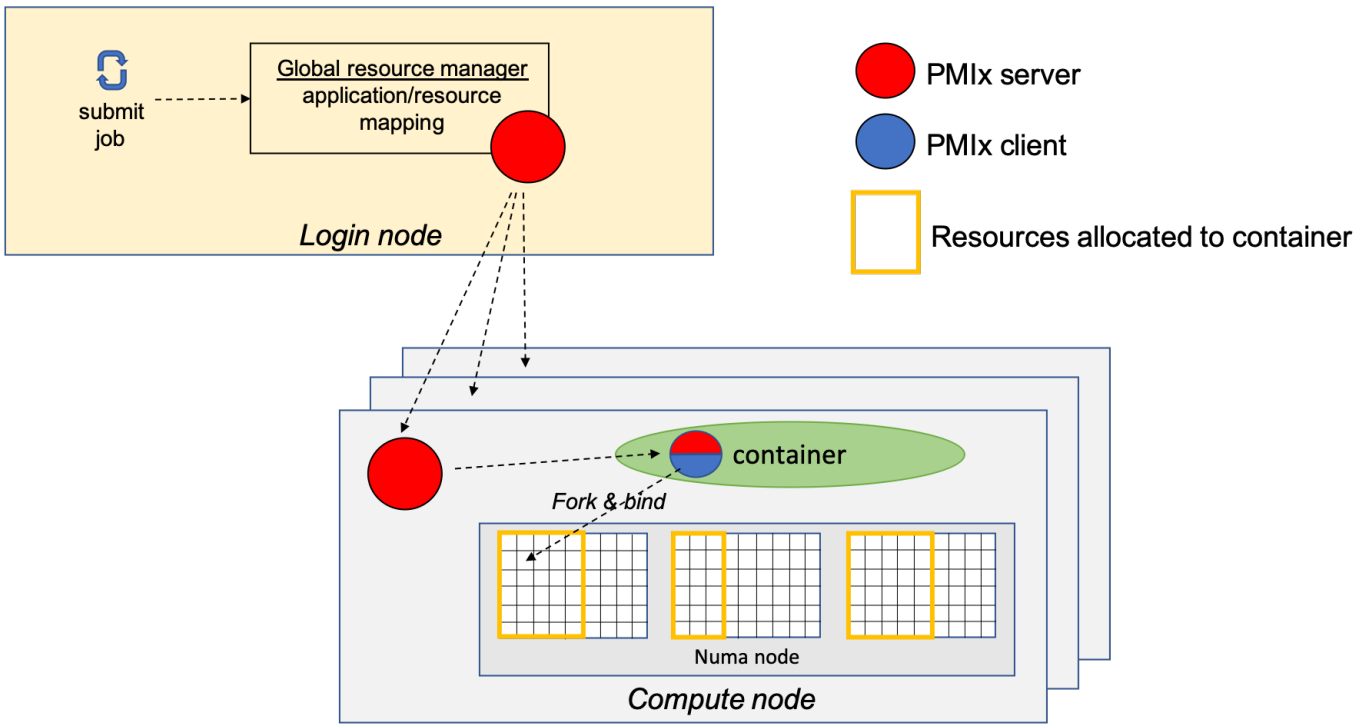

Fig. 2. Mapping and allocation of resources in the context of workloads using containers.

process (in the context of a process using various libraries in parallel). From there, Singularity and the runtimes running in the containers can exchange events (notifications) and share data. In our context, the PMIx thread acts as a server for the PMIx-compliant runtimes/libraries. The PMIx thread from the container runtime is therefore capable of coordinating with the various runtimes running in the container, exchange data about resource needs and manage the placement of threads and processes. Practically, the container's runtime is forking the various runtimes required by the application (e.g., the MPI runtime). As such, it is also possible for the container's runtime to partition or share resources between these runtimes. An example is through the use of CPU sets for resource partitioning: the container runtime can define different CPU sets to each runtime, for example through hwloc [14], and therefore ensure that each runtime has its own set of resources. This fine-grain control over resource management ensures that resources are shared only when required and otherwise properly partitioned to ensure performance and scalability.

III. Related Work

We present in this section some relevant projects both in terms of container technologies and resource management solutions for HPC.

A. Containers

Container technologies rapidly grow both in the context of enterprise and HPC computing. For enterprise computing, containers are mainly used to easily deploy and control micro-services. In fact, this is still nowadays the main use case for Docker [15]. But such solutions have been historically based on design choices that prevented them to be used on HPC systems, mainly running in privileged mode. As a result, the community saw the birth of container solutions that are designed with HPC in mind, e.g., Singularity [16] and CharlieCloud [17]. For example, Singularity supports the execution of container in user mode; container encryption to ensure that guarantee that the data stored within the container's image is securely handled, and a well documented interaction with HPC oriented programming environments such as MPI, as well as traditional HPC infrastructure components such as job schedulers.

B. Resource Managers

Resource management is a key capability for large-scale, multi-tenant, HPC systems. It ensures that the resources are adequately assigned to all applications running on the system and also ensures that these resources are charged to the appropriate projects.

Historically, resource managers have been centralized and static: upon the scheduling of a job, resources are allocated and the job is deployed. These resources are assumed to be allocated for the entire execution of the job and could not be extended (both in terms of time and additional resources). The HPC community developed various specifications, mainly driven by the needs from MPI applications, since MPI was at the time the dominating programming and execution environment for HPC applications. PMI [18] is such a specification and implementation, focusing on the scalable management of many processes running across many compute nodes. Later on,

the HPC community and the associated vendors developed the PMIx specification, which proposes a design and implementation that fits better modern architectures and requirements both in terms of performance and scalability.

As the HPC community is moving to exascale, a growing need for dynamic resource allocation is emerging. Fortunately, specifications such as PMIx already support the dynamic management of resources. However, the rest of the ecosystem is evolving at a slower pace. Global resource managers from vendors are just starting to consider supporting application-driven dynamic resource allocations. Furthermore, schedulers, such as Slurm [19], are not designed to let jobs perform their own resource management. New projects such as Flux [20] are investigating such a capability from a scheduling point of view. Practically, they provide a hierarchical scheduler with a local scheduler running on compute nodes when necessary, as well as a well-defined interface that can be used by users. However, to the best of our knowledge, there is no integration with container solutions. This lack of integration with container solutions is creating a gap: the applications running in containers find themselves isolated from the host system, limiting the possibilities for resource management. Furthermore, the container runtime being disconnected from the resource manager, it is not possible to finely place processes and threads running in containers. This is especially limiting with the HPC community moving to highly-heterogeneous systems where processes and threads will have to be carefully placed on CPUs and accelerators.

IV. Conclusion

While previous work already has been done for runtime coordination and hierarchical scheduling, to the best of our knowledge, this is the first architecture that provides *security* and *reproducibility* since based on containers, *fine-grain user resource management* since the proposed local resource manager is an extension of the Singularity runtime and a direct interaction with the various runtimes used the running applications, and *accountability* since interacting with the global resource manager. Our approach also enables the support of parallel debuggers and profilers which are common practice in the context of high performance computing.

As a result, the proposed architecture is the next step for a HPC-focused container solution.

As future work, we plan to implement the presented architecture and evaluate its performance using hybrid applications.

References

[1] R. H. Castain, D. Solt, J. Hursey, and A. Bouteiller, "Pmix: Process management for exascale environments," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 14:1–14:10. [Online]. Available: http://doi.acm.org/10.1145/3127024.3127027

[2] "The pmix standard," 2019. [Online]. Available: https://pmix.org/pmix-standard/

[3] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[4] P. Messina, "The exascale computing project," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 63–67, 2017.

[5] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: https://doi.org/10.1109/99.660313

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 2:1–2:3. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020375

[7] D. Bernholdt, H. Prichard, I. Laguna, R. Brightwell, and G. Bosilca, "Ompi-x: Open mpi for exascale – website," 2019. [Online]. Available: https://www.exascaleproject.org/project/ompi-x-open-mpi-exascale/

[8] G. Vallee and D. Bernholdt, "Improving support of mpi+openmp applications," in *Poster in Proceedings of The EuroMPI 2018 Conference*, 2018.

[9] R. Rubin, L. Rudolph, and D. Zernik, "Debugging parallel programs in parallel," *SIGPLAN Not.*, vol. 24, no. 1, pp. 216–225, Nov. 1988. [Online]. Available: http://doi.acm.org/10.1145/69215.69236

[10] C. M. Pancake and R. H. B. Netzer, "A bibliography of parallel debuggers, 1993 edition," *SIGPLAN Not.*, vol. 28, no. 12, pp. 169–186, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/174267.168238

[11] T. Islam, K. Mohror, and M. Schulz, "Exploring the capabilities of the new mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 91:91–91:96. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642781

[12] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *IWOMP*, 2013.

[13] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, and D. K. D. Panda, "Mpi performance engineering with the mpi tool interface: The integration of mvapich and tau," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 16:1–16:11. [Online]. Available: http://doi.acm.org/10.1145/3127024.3127036

[14] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "Hwloc: A generic framework for managing hardware affinities in hpc applications," in *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 180–186. [Online]. Available: http://dx.doi.org/10.1109/PDP.2010.67

[15] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[16] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[17] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126925

[18] 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 *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 31–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894122.1894127

[19] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[20] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, J. Koning, T. Patki, T. R. Scogland, B. Springmeyer *et al.*, "Flux: overcoming scheduling challenges for exascale workflows," in *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2018, pp. 10–19.