PMIx: Process Management for Exascale Environments

Ralph H. Castain
Intel, Inc.
ralph.h.castain@intel.com

David Solt, Joshua Hursey
IBM
dsolt@us.ibm.com
jhursey@us.ibm.com

Aurelien Bouteiller
The University of Tennessee,
Knoxville
bouteill@icl.utk.edu

ABSTRACT

High-Performance Computing (HPC) applications have historically executed in static resource allocations, using programming models that ran independently from the resident system management stack (SMS). Achieving exascale performance that is both cost-effective and fits within site-level environmental constraints will, however, require that the application and SMS collaboratively orchestrate the flow of work to optimize resource utilization and compensate for on-the-fly faults. The Process Management Interface - Exascale (PMIx) community is committed to establishing scalable workflow orchestration by defining an abstract set of interfaces by which not only applications and tools can interact with the resident SMS, but also the various SMS components can interact with each other. This paper presents a high-level overview of the goals and current state of the PMIx standard, and lays out a roadmap for future directions.

ACM Reference format:

Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. 2017. PMIx: Process Management for Exascale Environments. In *Proceedings of EuroMPI/USA '17, Chicago, IL, USA, September 25–28, 2017*, 10 pages. <https://doi.org/10.1145/3127024.3127027>

1 INTRODUCTION

The drive to produce ever more powerful supercomputers is challenged by the need for maintaining a manageable power budget despite stagnating processor frequencies. High Performance Computing (HPC) systems are subsequently embracing multilevel, massive parallelism based on a steadily increasing number of computing units. Consequently, the billion-way parallelism required to achieve exascale performance within reliability and power constraints is expected to feature from 50-100k nodes, which pushes future node architectures toward many-core and accelerator based designs, thereby raising the complexity of each node by orders of magnitude. Simultaneously, cost considerations force a corresponding decrease in the memory available per-core.

Similar to the developments in the hardware area, programming models are also undergoing major changes as they strive to achieve

exascale performance. New models continue to emerge at a rapid pace, frequently placing unique added requirements on the system management stack (SMS). At their core, the models all employ a communication library to express the necessary data movement and synchronization between the application processes (typically MPI[11], but also PGAS[19] languages and libraries such as UPC[9], OpenSHMEM[8], and others) that requires wireup support during initialization. In addition, coexisting programming models increasingly need to coordinate their actions with the SMS in order to achieve optimal performance and desired behaviors.

The deployment and management of large scale HPC applications has historically been fractured in a variety of competing systems and approaches that can deploy and monitor the execution of parallel applications. Vendors and MPI implementors have each developed a complicated ecosystem of solutions to provide such services involving a varied mix of SMS components[1, 2, 10, 17] and programming library runtimes[3, 5]. These solutions each have advantages (performance, portability, unique features), but the incompatibility of their interfaces adds overhead to the design of dependent software and renders difficult the direct use of the SMS by applications and tools for workflow orchestration.

The Process Management Interface (PMI), originally developed and distributed as part of MPICH[16], has historically been used as a means of exchanging wireup information needed for interprocess communication and deployment of processes. Two versions (PMI-1 and PMI-2[13]) have been released thus far, with PMI-2 demonstrating significantly better scaling properties than its PMI-1 predecessor. However, meeting the significant orchestration challenges presented by exascale systems requires that the process-to-SMS interface evolve to permit a tighter integration between the different components of the parallel application and existing and future SMS solutions.

PMI Exascale (PMIx[7]) addresses these needs by providing an extended version of the PMI definitions specifically designed to support exascale clusters by (a) eliminating some current restrictions that impact scalability, (b) augmenting the interfaces with extended capabilities, (c) establishing a standards-like body for maintaining the definitions, and (d) providing a reference implementation of the PMIx standard that demonstrates the desired level of scalability. The PMIx effort is not aimed at a particular programming model or implementation, but instead focuses on providing a well-defined interface that abstracts common features necessary to manage large scale application deployment and monitoring, in a programming model agnostic manner. This is being done through a process involving a multipartite community that includes vendors, programming library implementors, and providers of performance, monitoring and debugging tools.

The rest of the paper is organized as follows: Section 2 presents the motivation behind the PMIx initiative and the goals of the

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.

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroMPI/USA '17, September 25–28, 2017, Chicago, IL, USA

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-4849-2/17/09...\$15.00

<https://doi.org/10.1145/3127024.3127027>

PMIx community. Section 3 describes the existing state of the standard, and Section 4 outlines an example of how the standard can be used for application resilience. Finally, Section 5 outlines areas of current community activity, and we summarize the time schedule for PMIx deployment in Section 6.

2 PMIx COMMUNITY

The PMIx community consists of a consortium of industry, government, and academia members spanning the general HPC arena, specifically focused on orchestration of application launch and execution. The community arose out of shared concerns for both issues associated with scaling of machines to ever increasing size, and the ability to support the growing wave of innovation in the HPC programming area.

Assembling a coalition across such a diverse collection is never simple, and one critical factor was an early agreement that any SMS subsystem retains the right to return a "not supported" response to any request. Thus, participants agreed in essence to move the point of competition from the API to the quality and breadth of support for the standard. Customers will then be presented with a consistent API, for which it is easy to check if the support for desired features is present in the particular implementation, which informs their selection process for system software, and in turn fosters vendors to improve support.

The remainder of this section will describe the motivation behind the formation of the community, its processes for defining the PMIx standard, and its adopted scope.

2.1 Motivation

Exascale launch times are of significant concern due to the anticipated large number of nodes and processes, generally ranging from 50-100k distinct nodes, supporting up to 1M individual, probably multi-threaded processes. Regardless of programming model, high-speed communication between these processes will almost certainly be required, thereby mandating that a scalable method for connecting processes be provided.

Properly utilizing clusters of such size requires that applications and the system management stack (SMS) work together to both coordinate evolving resource needs, and identify and respond to faults. This places an increasing burden on SMS developers to not only adequately support such endeavors, but also to provide an appropriate infrastructure easing widespread adoption of those that prove successful. This burden is distributed across all the SMS components — for the purposes of this paper, these will be defined as:

- (a) the workload manager (WLM), responsible for scheduling resource allocations for jobs;
- (b) the resource manager (RM), tasked with spawning and monitoring application processes, and tracking/managing resource utilization by those processes;
- (c) the global file system (FS) that serves as persistent storage for the cluster. The FS frequently contains multiple elements, including a parallel file system (PFS) for both input and output data files, and a network file system (NFS) that typically houses home directories, executables, and supporting libraries. In addition to large-scale backend storage, the File System may including caching elements

either in the form of "burst buffers" located on the periphery of the cluster, or *in situ* banks of non-volatile RAM (NVRAM) distributed throughout the cluster;

- (d) the fabric manager (FM) that oversees the high-speed network, defining addresses and routing patterns for inter-node communication. This includes one or more local network interface cards (NICs) on each compute node; and

- (e) the reliability, availability, and serviceability subsystem (RAS) whose primary role in HPC systems is monitoring and providing alerts for both hardware and firmware failures.

Excluding vendor-proprietary implementations, these components have historically been developed by independent groups, and therefore the resulting packages include, at best, light levels of integration. The diversity of offerings in each category further complicates integration efforts as each offering maintains its own unique interfaces and data structure definitions.

The PMIx community seeks to resolve the integration problem by providing an abstracted, standardized set of interfaces, while leaving implementation of those interfaces to others. The community has focused on four general areas: launch support, memory efficiency, support across programming models, and tools.

2.1.1 Launch support. Studies performed during the RoadRunner [14] project provided valuable insights into the breakdown of time spent during launch. While the detailed timings have of course changed since that time, the overall distribution and the sequence of events during startup has remained relatively consistent. Launch of a typical parallel application can be separated into six distinct stages, as shown in Fig. 1:

Stage 1. User submits a job (either in the form of a script or a request for an interactive session) to the WLM. The request typically will include descriptions of the type and amount of resources required for the job. Historically, resource requests have been considered immutable once allocated, but demand and support for dynamically modifiable resource allocations is growing.

Stage 2. Depending upon resource availability, relative priorities, and scheduling algorithm, at some point in the future the WLM will assign resources to the requested job and inform the RM daemons that control those resources of their assignment. The precise sequence of steps used by an RM daemon to spawn its local application processes varies, but the overall result is that the application binary executable is retrieved from the FS, instantiated on the compute node, and executed. At this time, any dynamic library dependencies will result in calls to retrieve the corresponding libraries from the FS.

Scalably dealing with the library retrieval "storm" caused by multiple processes from a large number of nodes near-simultaneously requesting a library has been the subject of significant investigation. Some systems simply utilize static builds to avoid the storm altogether, while others cache the first request on an intermediate location (e.g., a "burst buffer") and then serve all subsequent requests from the cache (cfe, SPINDLE [12]).

Stage 3. Once instantiated, each process in the application must discover local information required for interactions with its peers. This information consists of at least the process rank and communication endpoint (e.g., an IP address and TCP socket number), but typically includes any process affinity that has been applied,

Figure 1: Non-PMIx Start Sequence

local network capabilities, and other factors that might influence optimized configuration of the application (e.g. selection of communication algorithms). Once the information has been locally discovered, the process may publish the information to a local proxy server assigned to their node (usually hosted by the local RM daemon, or a daemon provided by a dedicated starter such as *mpirun*).

Stage 4. Global exchange of the published information (often referred to as the *modex*) is then executed via a collective operation normally executed over the management Ethernet, either conducted directly by all processes or by their per-node proxies. This stage is typically the largest application launch time component. Efforts to reduce launch time have therefore focused primarily on development of more efficient algorithms for this stage, and more recently on moving the collective to the high-speed fabric. While the latter is generally acceptable when nodes are allocated to a single application, this may not be compatible with future practices. In addition, while both approaches reduce the magnitude of the endpoint exchange problem, they do not eliminate it as scaling dependencies on the number of nodes and/or processes continue to exist.

Stage 5. Once the information has been exchanged, the programming libraries typically assemble necessary infrastructure based on that information - e.g., mapping communication channels to peers. This again is largely a measure of the efficiency of the library and its choice of architecture. The primary point of concern here is the time spent retrieving connection and other required information from the local proxy server (if present), and the time needed for each process to configure their local NIC library for operation using that information.

Stage 6. Finally, a barrier is executed (again, typically across the management Ethernet) to ensure all processes are prepared to communicate. This ensures that all processes have completed their setup and the local high-speed network is prepared to receive messages. Release from this barrier is considered to be final approval for the application to commence operations that include communication. While the barrier involves no data and is therefore faster than the previous modex operation, it still consumes the next largest block of time in the start profile. Launch improvement efforts have included extension of modex optimizations to this stage.

The PMIx community has committed to reducing or eliminating the time spent in these stages to achieve an overall goal of launching and connecting exascale applications in under 30 seconds. For purposes of tracking this goal, the community has adopted its baseline

test as being the time required to start an application and complete MPI_Init, with all processes having all required information to communicate at that point, using an application size of 50k nodes supporting up to 1M individual processes.

2.1.2 Memory Efficiency. The process outlined in Fig. 1 results in the aggregation of a complete copy of all relevant data either in the proxy on each node (if a proxy was used), or in each individual process if the exchange was executed directly by the application. Systems that utilize the proxy approach generally still require that each application process obtain a copy of each data element that it requires. Thus, data replication can be fairly extensive. Considering that nodes are trending towards increasingly larger numbers of cores, the duplication of SMS state in each process of an exascale application may impose a heavy penalty in terms of user-available memory.

The PMIx community is addressing this emerging issue by ensuring that relevant defined interfaces are conducive to shared and/or distributed memory methods. The intent here is to ensure that the PMIx implementation has full latitude to optimize the tradeoff between access cost and memory consumption for caching the required information, in concert with the host SMS environment and the various programming libraries.

2.1.3 Programming models. Whether it be hybrid applications that combine OpenMP threading operations with MPI, new workflow paradigms based on decomposition of the problem into large numbers of small parallel tasks, or entirely new methods for parallel computations, the HPC community is witnessing a significant burst of new approaches to application architecture as researchers explore best methods for executing at exascale levels. No clear answer has yet to emerge, but all methods to-date share a common requirement for interconnection support and an increasing need for orchestration of their operations with the SMS.

Research teams considering participation in this area, however, face a significant obstacle to entry into the field. Because most SMS environments do not provide the dynamic support required by these efforts, the teams must either absorb the high cost to develop a complete "virtual" environment that hides the host SMS behind a customized abstraction layer, or suffer lack of portability by utilizing proprietary interfaces for interacting with those system management stacks that support the necessary features.

PMIx can help support and encourage such explorations by providing early researchers with an abstracted and portable capability for SMS interactions. The community is working with several

groups to ensure that the PMIx standard incorporates adequate flexibility to meet their evolving needs, and to identify additional support needed for cross-model interactions including the sharing and allocation of processor cores and network resources between the programming models.

2.2 Standards process

2.2.1 Tool support. While the primary purpose of PMIx is to provide a mechanism by which applications can interact with their host SMS, there are times when non-application processes (e.g., command-line tools) may also wish to communicate with the SMS or an application. Historically, such tools were specific to their environment due to the customized and/or proprietary nature of their interfaces.

PMIx offers the possibility for creating portable tools capable of querying multiple PMIx-enabled SMS environments without modification. Since tools are often started at the command line prompt instead of launched by the SMS, the PMIx standard defines an attribute by which the host SMS can direct the local PMIx server to accept tool connections, thus causing the PMIx server to publish the necessary rendezvous connection information in a known location. The standard includes hooks by which the SMS can validate the tool's connection request and ensure proper authorizations for tool requests.

PMIx interfaces can also be used to support shell script interactions with applications. For example, the PMIx Publish API provides the ability to post information to an SMS-hosted key-value store, controlling both the access permissions and the persistence of the provided data, including directives to persist after the application has ceased executing, and to be removed upon first access. Correspondingly, the PMIx Lookup API can be directed to return once the requested data becomes available, or upon reaching a provided timeout limit. In combination, the two interfaces provide a rich layer for asynchronous exchange of data for execution control, enabling an application to post detailed information on its completion status suitable for steering follow-on execution paths by the controlling job script.

Tools such as debuggers and performance profilers face similar problems to applications when expanding to the exascale level and are increasingly being asked to support a wider range of programming libraries and paradigms. Standardizing the SMS interaction pathway is enabling tool vendors to access a broader range of information and capabilities, while supporting greater portability and ease of extension to the exascale level. Likewise, the PMIx interfaces allow tools to interact with applications through model-independent interfaces, thus allowing researchers to use known debugging environments on new programming models without extensive integration efforts.

Prior approaches to attaining launch performance goals and providing support for tools and programming models has focused on specific programming model implementations associated with a particular team of developers. This results in a lack of portability and creates an obstacle to adoption by SMS vendors. Similarly, application integration to the SMS should not be constrained to a particular SMS environment or programming library. Users desire the ability to program to a set of "standard", agnostic interfaces

supported across the HPC community. Accordingly, the PMIx community has adopted a standards-based approach to defining the PMIx APIs that relies on submission and review of Requests For Comment (RFCs) detailing the proposed modification or extension, along with a prototype implementation. Activities within the PMIx initiative spans a number of areas, each led by a small informal working group of contributors. Final decision on approval is typically made at one of the community's weekly teleconferences.

The resulting process has proven agile enough for the community's needs, subject to the constraint that modification of existing released APIs will only be permitted under extreme circumstances. In its effort to avoid introduction of any such backward incompatibility, the community has avoided the adoption of large numbers of APIs that each focus on a narrow scope of functionality, and instead relied on the definition of fewer generic APIs that include arrays of directives for "tuning" the function's behavior.

Implementations may simultaneously support multiple versions of the specification, and users are provided with methods to identify the list of supported versions. Note that it is not mandatory for PMIx implementations to support more than one version of the specification. However, the integration of backward compatibility modules featuring legacy versions of the PMIx API, with auto-negotiation of API protocol, is strongly encouraged in implementations of the PMIx server to enable cross-operation between legacy client components and PMIx servers.

2.3 Scope and Architecture

The scope of the PMIx standard has evolved over time as driven by requests from users and members. The initial focus on reducing launch times required integration between a number of SMS subsystems to gain the necessary support — e.g., integrating the Resource Manager (RM) with the fabric manager (FM) to obtain connection endpoint information, or with the file system (FS) to prefetch and cache binaries and libraries prior to job start. This provides an opportunity for expanded utilization. In exchange for vendor support for launch time reduction, the community has defined abstractions supporting a broad range of interactions across SMS subsystems, based on the architecture illustrated in Fig. 2.

Note that this is a conceptual model used to help guide the standards process — it does not represent a design requirement on any PMIx implementation. Instead, the model is used by the PMIx community as a sounding board for evaluating proposed interfaces to avoid unintentionally imposing constraints on implementers. Built into the model are two guiding principles also reflected in the standard. First, PMIx operates in the mode of a *messenger*, and not a *doer* — i.e., the role of PMIx is to provide communication between the various participants, relaying requests and returning responses. The intent of the standard is not to suggest that PMIx itself actually perform any of the defined operations — this is left to the various SMS elements and/or the application. The only exception to this rule is when requests are consolidated for optimization purposes. For example, the PMIx server may (depending upon implementation) choose to aggregate all local PMIx_Fence calls before passing them to the SMS for global execution.

Figure 2: PMIx-SMS Interactions

Thus, as the diagram shows, the application is built against a PMIx client library that contains the client-side APIs, attribute definitions, and communication support for interacting with the local PMIx server. Cross-library interactions are supported at the client level to avoid unnecessary burdens on the server. Orchestration requests are sent to the local PMIx server, which subsequently passes them to the host SMS (here represented by an RM daemon) using the PMIx server callback functions the host registered during `PMIx_server_init`. The host can indicate its lack of support for any operation by simply providing a NULL for the associated callback function.

The conceptual model places the burden of fulfilling the request on the host SMS. This includes performing any inter-node communications, or interacting with other SMS elements. Thus, a client request for a network traffic report does not go directly from the client to the FM, but instead is relayed to the PMIx server, and then passed to the host for execution. This architecture reflects the second principle underlying the standard — namely, that connectivity is to be minimized by channeling all application interactions with the SMS through the local PMIx server.

Recognizing the burden this places on SMS vendors, the PMIx community has included interfaces by which the host can request support from local SMS elements. Once the SMS has transferred the request to an appropriate location, a PMIx interface can be used to pass the request into the target SMS subsystem. For example, a request for network traffic statistics can be sent to the node hosting the FM, where the PMIx networking abstractions can be used to retrieve the information from the FM. This transfers the burden of dealing with the interfaces of individual subsystems from the SMS to the PMIx community, which continues to work with those providers to develop the necessary support.

Tools, whether standalone or embedded in job scripts, are an exception to the communication rule and can connect to any PMIx server providing they are given adequate rendezvous information. The PMIx conceptual model views the collection of PMIx servers as a cloud-like conglomerate — i.e., orchestration and information requests can be given to any server regardless of location. However,

tools frequently execute on locations that may not house an operating PMIx server — e.g., a users notebook computer. Thus, tools need the ability to remotely connect to the PMIx server "cloud".

The scope of the PMIx standard therefore spans the range of these interactions, from client-to-SMS, and between SMS subsystems. Note again that this does not impose a requirement on any given PMIx implementation to cover the entire range — implementers are free to return *not supported* from any PMIx function.

2.4 Reference Implementation

The very breadth of PMIx's scope can present a challenge to adoption by SMS vendors and programming library developers. Accordingly, the PMIx community has developed and released a PMIx "reference implementation" containing a complete implementation of the PMIx standard, each release being tied directly to a corresponding revision level of the standard. The reference implementation itself is *not* part of the PMIx standard, nor is its use in any way required. It exists solely for the purpose of providing a quick, convenient adoption path.

Similarly, the PMIx community has released a "reference server" — i.e., a runtime environment containing the reference implementation and capable of operating within a host SMS. The reference server therefore provides an easy way of exploring PMIx capabilities and testing PMIx-based applications outside of a PMIx-enabled environment.

3 CURRENT FOCUS AREAS

In this section we discuss the current working groups dedicated to fleshing-out the PMIx specification, and how the interfaces they are designing help reduce overhead and offer new capabilities. Unless specified, these capabilities are present in the current v2.0 specification and supported in the corresponding v2.x reference implementation as well.

Figure 3: PMIx-Enabled Launch

3.1 Orchestrating the Launch Process

As previously discussed, the largest time costs during application startup has historically been the time required to collectively share connection endpoints, followed by the barrier prior to commencing operations. Analysis of exchanged data revealed that over 90% of it was known to the local RM daemon at the time it started an application process, but that the daemon had no mechanism for communicating that information to the client process. The remainder was dynamically-assigned endpoint connections that client processes independently discovered and then contributed to the collective operation.

The PMIx community addressed the launch sequence with a goal of completely eliminating both the global exchange of information and the subsequent global barrier, regardless of application scale. In addition, the community determined that the file system retrieval bottleneck could be effectively eliminated by pre-positioning the required information. The revised launch sequence consists of only four stages, as shown in Fig. 3:

Stage 1. User submits a job (either in the form of a script or a request for an interactive session) to the WLM. The request typically will include descriptions of the type and amount of resources required for the job. In addition to logging the request into the appropriate queue, the WLM uses a PMIx interface to pass a request to the file system for estimated retrieval times of all user-identified executables, supporting libraries, and data files. The resulting estimates can then be factored into the scheduling algorithm, especially important as hot/warm/cold storage policies become more common.

The burden on the user to identify all required libraries is often an issue. Hence, PMIx provides another interface by which the WLM can first request that a job script be analyzed to return a list of associated libraries (e.g., the equivalent to the output of a standard POSIX *ldd* command) that can be included in the FS request for retrieval time. It is expected that the development of increasingly accurate analyzers will become a research area of interest.

Stage 2. Depending upon resource availability, relative priorities, and scheduling algorithm, at some point in the future the WLM will assign resources to the requested job. This assignment typically occurs when the WLM is notified of job completion and just prior to initiation of the epilog portion of the prior job. In addition, the need to aggregate resources for larger jobs often results in at least some portion of the allocation becoming available on a rolling basis. Thus, there is some amount of time available to the SMS for pre-conditioning of the allocation prior to start of the newly scheduled job.

PMIx takes advantage of this time to accomplish two important steps. First, it provides an interface by which the WLM can notify the FS of the allocation and pass it the list of required files as determined during the initial stage. This allows appropriately enabled file systems to pre-cache the files to a location network-near the allocated compute nodes, assuming such caching elements are available.

Second, the WLM can utilize another PMIx interface to pass both the allocation and job description to subsystems such as the fabric manager, receiving back a "blob" containing all information each subsystem has determined will be required by its local resources within the allocation to support the described job. Returned values are marked to identify which are to be given directly to the local resource (e.g., network driver) on each node, and which are to be passed to application processes as environmental variables that are recognized by the resource when initialized by that specific process. For example, some networks may "preload" the local network library with the location of all processes in that application, thus allowing the library to compute the required address information for any process, and/or include a security token to protect inter-process communications.

Stage 3. Once the required files have been cached and the network configuration obtained, the WLM passes the initialization "blob" along with the job description to the RM daemons on the allocated nodes. Prior to spawning any local processes, each RM daemon can utilize a PMIx interface to deliver the configuration

information to the local resources. This allows the resources to prepare any internal state tables for the job, an operation which may require elevated privileges and therefore cannot be performed by the application processes themselves. In addition, the RM daemon *registers* the application and local processes with the local PMIx server, providing it with all relevant information pertaining to the application and local environment. The PMIx community has assembled a list of desired information to assist SMS vendors in meeting this requirement[6]. With this accomplished, the RM daemon spawns the application processes.

Stage 4. During their initialization, each process connects to its local PMIx server by calling the `PMIx_Init` function. The PMIx server notifies the host RM of the connection, and returns to the process all information (or a pointer to shared memory where the data resides) provided by the local RM daemon during registration.

At this point, the launch sequence is complete and application processes are free to communicate. Note that while PMIx provides interfaces by which application processes can globally exchange data amongst themselves, these interfaces are primarily intended for use by legacy systems. Full support for the PMIx launch sequence requires modifications to the fabric manager and NIC libraries, as well as potential hardware changes for optimized performance. The PMIx community is working with network vendors to field this support in upcoming product releases. This includes support for unexpected messages (i.e. messages received before the complete initialization of the high performance communication library) that eliminates the need for a barrier operation at the end of the initialization procedure. Similarly, the file system integration required for identifying dependent libraries and pre-positioning files has been deferred to v3.0 of the standard.

3.2 Event notification

Another PMIx working group is focused on dealing with faults. For the community's purposes, a *fault* was defined as anything negatively impacting the operation of the application. Thus, an application that has *stalled* (i.e., is no longer making progress on its work) is considered as experiencing a fault, as is network congestion and the failure of a hardware component. PMIx enables fault mitigation by allowing processes to register event handlers by which:

- (a) the resource manager can notify the application of system problems, both detected and anticipated, including the intent to preempt the application. PMIx does not generate events itself—it only publishes them and collects responses that might arise from them. Thus, PMIx provides an abstraction layer to help facilitate RM integration to subsystems that provide such warnings (e.g., the monitoring system);

- (b) application processes can notify their peers and/or the SMS of issues they have detected - e.g., the loss of communication to a peer. Once an event is received, an application can use PMIx to interact with the RM to identify the desired response to an event, including requesting replacement nodes, restart of specific processes from a given checkpoint, or any other operation supported by the RM; and

- (c) SMS subsystems, such as the fabric manager, can notify the resource manager of traffic congestion or rerouting operations in response to failures.

PMIx event handlers are serviced in a chained ordering, beginning with handlers registered to receive a specific event and progressing thru handlers registered to receive multiple events and ending with default event handlers. Users can specify the order of a handler within those groupings at the time of registration. PMIx automatically aggregates the result of each handler (as provided upon handler return) so that each step in the chain has full knowledge of what preceded it.

Event registration can be divided into two distinct classes. Job-specific events directly relate to a job executing within the session, such as a debugger attachment or process failure within a related job. Events in this category are to be immediately delivered to the PMIx server library for delivery to the related processes. In contrast, environment events include ECC errors, temperature excursions, and other environmental events directly affecting the session's resources. Note that although these do impact the session's jobs, they are not directly referencing those jobs - i.e., the event is generated without specifying a particular target. Thus, events in this category are to be delivered to the PMIx server library only upon request.

Both SMS elements and applications can register for events of either type. Note that race conditions can cause the registration to come after events of possible interest (e.g., a memory ECC event that occurs after start of execution but prior to registration). SMS vendors are *requested* to cache events in this category for some time to mitigate this situation, but are not *required* to do so. Thus, applications must be aware that they may not receive environment events that occur prior to registration, depending upon the capabilities of the host SMS.

3.3 Flexible resource allocations

Interest in dynamic resource management has increased, driven by new programming models and concerns over higher failure rates in extreme scale systems. In response, another PMIx working group is focused on enabling application-directed on-the-fly resource changes such as:

- (a) request allocation of additional resources. This is accomplished in a non-blocking manner so that the application can continue to progress while waiting for resources to become available. Note that the new allocation will be disjoint from (i.e., not affiliated with) the allocation of the requester - thus the termination of one allocation will not necessarily impact the other;

- (b) extend the reservation on currently allocated resources, subject to scheduling availability and priorities. This includes extending the time limit on current resources, and/or requesting additional resources be allocated to the requesting job. Any additional allocated resources will be considered as part of the current allocation, and thus will be released at the same time;

- (c) release currently allocated resources that are no longer required. This is intended to support partial release of resources since all resources are normally released upon termination of the job. The identified use-cases include resource variations across discrete steps of a workflow, as well as applications that spawn sub-jobs and/or dynamically grow/shrink over time; and

- (d) "lend" resources back to the scheduler with an expectation of getting them back at some later time in the job. This can be a proactive operation (e.g., to save on computing costs when resources are

temporarily not required), or in response to scheduler requests in lieu of preemption. A corresponding ability to "reacquire" resources previously released is included.

Each allocation request can include a user-created "allocation ID" string. This allows the initiator to issue follow-on requests (e.g., for status updates via the `PMIx_Query` function), or to issue a subsequent call to cancel the request. A call to cancel allocation requests that does not specify an allocation ID will result in cancellation of all outstanding requests associated with the requester. Requests can be labeled as *required* to indicate that the request is to be rejected if the specified resources are not available, or *optional* to indicate that the specification is provisional.

Allocation requests are complete when the callback function is executed. However, this will only occur in the process that initiated the request. Other processes can poll the status of the request using the `PMIx_Query` function, or can register for event notification upon completion of the allocation operation. In either case, the requesting process can specify the application request ID. The lack of a specific ID will result in notification for all allocation requests associated with the application to which the process belongs.

3.4 Job control and monitoring

In addition to external faults, a common problem encountered in HPC applications is a failure to make progress due to some internal conflict in the computation. These situations can result in a significant waste of resources as the SMS is unaware of the problem, and thus cannot terminate the job. Various watchdog methods have been developed for detecting this situation, including requiring a periodic "heartbeat" from the application and monitoring a specified file for changes in size[15, 18].

At the request of SMS vendors and members, a monitoring support interface has been included in the `PMIx v2.0` standard. The defined API allows applications to request monitoring, directing what is to be monitored, the frequency of the associated check, whether or not the application is to be notified (via the event notification subsystem) of an associated event, and other characteristics of the operation. In addition, the heartbeat and file monitoring methods has been included in the `PMIx` reference implementation, active only when requested.

Detecting stalled processes and receiving notification of external faults is, of course, only part of the problem. The application must then work with the SMS to actually do something about the situation. Typical current practice is to simply terminate the impacted application, but future systems require less draconian solutions.

The `PMIx` job control API enables the application and SMS to coordinate the response to failures and other events. This can include requesting termination of the entire job or a subset of processes within a job, but can also be used in combination with other `PMIx` capabilities (e.g., allocation support) to obtain replacement resources and restart processes on them, register for checkpoint support and trigger checkpoint events, pause and restart processes using standard POSIX signals, or even request that specific nodes be provisioned using a given image.

The job control API can also be used by an application to register itself as available for preemption. Jobs that do so will receive an event warning them of an impending preemption so they have an

opportunity to save their work. Once the application is ready, it calls the event completion callback function to indicate that the SMS is free to suspend or terminate it.

3.5 Queries and Logging

As the level of interaction between applications and the host SMS grows, so too does the need for the application to query the SMS regarding its capabilities and state information. `PMIx` provides a generalized query interface for this purpose, along with a set of standardized attribute keys to support a range of requests. This includes requests to determine the time remaining in the allocation, status of scheduling queues, the scope of API and attribute support offered by the SMS, location and information about job processes, and information regarding available resources.

The `PMIx_Log` interface is provided to support posting information such as computation progress reports to the SMS job log. This has since been generalized to allow posting to a variety of channels to support use-cases such as remote monitoring of application progress (e.g., via email), and error reporting for post-mortem analysis.

4 USAGE EXAMPLE

In this section, we present an actual use case of the `PMIx` interfaces to support fault tolerance in Open MPI based on the User Level Failure Mitigation (ULFM) specification [4] that consolidates detection and coordinates response to failure events.

Responding to failures is a complex problem as events may be reported by a variety of components in the system, e.g. network drivers, in-band detection from failed MPI communication, MPI process resource environment, system-wide resource manager RAS service, switches and SNMP traps. Similarly, multiple components may be interested and react to fault events, creating a need to coordinate their response.

Many Open MPI components already had an error management subsystem, but the behavior of each component was hardcoded to abort the application process. These components have now been modified to publish detected events using the `PMIx` event notification framework. Once published by a client component, the `PMIx` server notifies its local child processes and passes the event to its host RM for dissemination of the event to other `PMIx` servers within the scope of the event (by default the job scope, i.e. the MPI universe). Each `PMIx` server then passes the event to its own local clients that have subscribed for matching events. Thus, not only are application processes informed of the fault event detected by the MPI library, but other entities (like the RM) are also informed. Conversely, faults detected out-of-band by the RM or a RAS system can be published and thereby captured in a coherent fashion by application processes.

In the Open MPI architecture, the main client is the Open RTE daemon in charge of managing the MPI processes lifecycle on a node. The daemon subscribes to a variety of fault related events (unexpected process disconnection, process terminated by signal, impossible to deliver messages to the process, etc.), and consolidates these events to produce a process fault event denoting the fault nature and which MPI processes have been affected. This event is

then published to the local clients scope (i.e. the node local processes), with the effect of triggering an appropriate fault handler in the client MPI library.

Within the MPI application processes, the mechanism of priority and state propagation between components through PMIx permits coordinating the fault response. Resilient components register for fault event with high priority, thereby allowing events to default to components that abort the process only in the cases when the fault cannot be corrected in a satisfactory manner. In the non-resilient version of Open MPI, only the default handler (which aborts the application) is registered. In the ULFM version, that same handler remains registered, but a resilient MPI error handler is installed on top, which is then in charge of triggering the MPI exceptions for ongoing communication to report the fault.

5 FUTURE DIRECTIONS

PMIx working groups continue to define new capabilities based on identified needs for application-SMS coordination. Current efforts target the PMIx v3.0 standard, expected to be released in late 2017. One group is investigating use of PMIx to coordinate inter-library operations in hybrid applications (i.e., applications that utilize more than one programming model, such as an MPI application that also uses OpenMP). Initial efforts focused on providing a reliable mechanism for a programming model to learn when another model is being employed, without burdening the user with requirements to declare such scenarios in advance of execution. The PMIx event notification system is well suited for this role as it is asynchronous and supports cached events. The same mechanism also allows libraries to share knowledge of each other's resources and intended resource utilization. Thus, the working group is experimenting with having both OpenMP and MPI libraries declare their presence and register for events for inter-library coordination.

A second group is organizing to explore improvements to system throughput by pre-positioning binaries, libraries, and data to network-near caches prior to starting an application. This is being accomplished through several new PMIx features, including an interface by which the workload manager can obtain a list of required executables and their dependencies from a job submission script, and enabling the workload manager to request that the file system pre-position the files and provides notification when ready for execution.

Finally, another PMIx working group is looking at a generalized data store framework capable of supporting multiple storage implementations through a common abstraction interface. This would not be used for storing computational output, but rather provides an abstracted interface for research into various methods of storing and disseminating operational information across the SMS and applications. This might include any launch-related data, but also could be extended to provide backend support for persistent state storage, the PMIx_Log function, and the PMIx publish/lookup interfaces. The intent is primarily to provide a platform for researchers to investigate different storage topologies such as distributed hash tables (DHTs) and hierarchical caches (cold/warm/hot), as they pertain to uses of interest to the SMS and application communities. For example, researchers are looking at the use of DHTs instead

of each PMIx server locally caching job-related information as a means of reducing the PMIx server's on-node footprint.

6 SUMMARY

Version 1 of the PMIx standard was reviewed and approved in 2015. Support for that standard was subsequently released in SLURM (beginning with v15.06), IBM's Job Step Manager (for CORAL deliverables), and Open MPI (beginning with release v2.0). Since that time, APIs for fabric integration, event notification, flexible resource allocation, job control and monitoring, queries and logging have been reviewed and approved as part of the PMIx v2 standard released in June 2017. A range of SMS vendors have initiated work to support this version of the standard.

Starting from this very strong core, the PMIx community is reaching out for all interested parties to manifest their use cases, help identify existing and foreseeable job management scalability and performance issues, and are welcome to participate in the community's working groups. Information on PMIx can be found at the community web site (<http://pmix.github.io/pmix/>).

7 ACKNOWLEDGEMENTS

The authors would like to acknowledge the contribution of the overall PMIx community to both the formulation of the standard and development of the PMIx reference implementation. Special thanks go to Artem Polyakov, Boris Karasev, and Joshua Ladd of Mellanox; Adam Moody of Lawrence Livermore National Laboratory; and George Bosilca of University of Tennessee, Knoxville.

REFERENCES

- [1] Adaptive Computing Inc. 2017. Moab HPC Suite. <http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/>. (2017). [Online; accessed 14-Jul-2017].
- [2] Altair Inc. 2017. PBS Professional. <http://www.pbsworks.com/PBSProduct.aspx?n=PBS-Professional&c=Overview-and-Capabilities>. (2017). [Online; accessed 26-Apr-2017].
- [3] Argonne National Laboratory. 2014. Hydra Process Management Framework. https://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework. (2014). [Online; accessed 26-Apr-2017].
- [4] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-failure recovery of MPI communication capability. *The International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254. <https://doi.org/10.1177/1094342013488238> arXiv:<http://dx.doi.org/10.1177/1094342013488238>
- [5] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. 2005. The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*. Springer-Verlag, Sorrento, Italy, 225–232. https://doi.org/10.1007/11557265_31
- [6] PMIx Consortium. 2016. PMIx Server Data Requirements. <https://github.com/pmix/pmix/wiki/2.8-Pmix-Server-Data-Requirements>. (2016). [Online; accessed 24-Apr-2017].
- [7] PMIx Consortium. 2016. Process Management Interface - Exascale. <https://pmix.github.io/pmix>. (2016). [Online; accessed 26-Apr-2017].
- [8] OpenSHMEM Consortium. 2016. OpenSHMEM Specification 1.3 Final. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf. (2016). [Online; accessed 26-Apr-2017].
- [9] UPC Consortium. 2013. UPC Language Specifications. <https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf>. (2013). [Online; accessed 26-Apr-2017].
- [10] Cray Inc. 2011. Workload Management and Application Placement for the Cray Linux Environment. <http://docs.cray.com/books/S-2496-4001/S-2496-4001.pdf>. (2011). [Online; accessed 26-Apr-2017].
- [11] Message Passing Interface Forum. 2015. MPI-3.1: Official document. <http://www.mpi-forum.org/docs>. (June 2015).
- [12] Wolfgang Frings, Dong Ahn, Matthew LeGendre, Todd Gamblin, Bronis de Supinski, and Felix Wolf. 2013. Massively Parallel Loading. In *Proceedings*

- of the 27th International Conference on Supercomputing*. ACM, 389–398. <https://doi.org/10.1145/2464996.2465020>
- [13] Argonne National Laboratory. 2012. PMI v2 API. https://wiki.mpich.org/mpich/index.php/PMI_v2_API. (2012). [Online; accessed 26-Apr-2017].
- [14] DOE/Los Alamos National laboratory. 2009. Science At The Petascale: Roadrunner Results Unveiled. <https://www.sciencedaily.com/releases/2009/10/091026125535.htm>. (October 2009). [Online; accessed 15-May-2016].
- [15] Mark Grondona. 2012. io-watchdog. <https://github.com/grondo/io-watchdog>. (2012). [Online; accessed 17-May-2017].
- [16] Mathematics and Argonne National Laboratory Computer Science Division. 2006. MPICH-2, implementation of MPI 2 standard. <http://www-unix.mcs.anl.gov/mpi/mpich2/>. (2006). [Online; accessed 26-Apr-2017].
- [17] SchedMD, LLC. 2017. SLURM Workload Manager. <https://slurm.schedmd.com/>. (2017). [Online; accessed 26-Apr-2017].
- [18] Hadi Sharifi, Omar Aaziz, and Jonathan Cook. 2015. Monitoring HPC Applications in the Production Environment. In *Proceedings of the 2nd Workshop on Parallel Programming for Analytics Applications (PPAA 2015)*. ACM, New York, NY, USA, 39–47. <https://doi.org/10.1145/2726935.2726944>
- [19] PGAS working group. 2016. Partitioned Global Address Space. <http://www.pgas.org/>. (2016). [Online; accessed 26-Apr-2017].