MPI Sessions: Evaluation of an Implementation in Open MPI

MPI Sessions: Evaluation of an Implementation in Open MPI

Nathan Hjelm
Google Inc.
Seattle, USA
hjelmn@google.com

Howard Pritchard
Los Alamos National Laboratory
Los Alamos, USA
howardp@lanl.gov

Samuel K. Gutiérrez
Los Alamos National Laboratory
Los Alamos, USA
samuel@lanl.gov

Daniel J. Holmes
EPCC, The University of Edinburgh
Edinburgh, UK
d.holmes@epcc.ed.ac.uk

Ralph Castain
Intel
Hillsboro, USA
ralph.h.castain@intel.com

Anthony Skjellum
University of Tennessee at Chattanooga
Chattanooga, USA
Tony-Skjellum@utc.edu

***Abstract*—The recently proposed MPI Sessions extensions to the MPI standard present a new paradigm for applications to use with MPI. MPI Sessions has the potential to address several limitations of MPI's current specification: MPI cannot be initialized within an MPI process from different application components without *a priori* knowledge or coordination; MPI cannot be initialized more than once; and, MPI cannot be reinitialized after MPI finalization. MPI Sessions also offers the possibility for more flexible ways for individual components of an application to express the capabilities they require from MPI at a finer granularity than is presently possible.**

At this time, MPI Sessions has reached sufficient maturity for implementation and evaluation, which are the focuses of this paper. This paper presents a prototype implementation of MPI Sessions, discusses certain of its performance characteristics, and describes its successful use in a large-scale production MPI application. Overall, MPI Sessions is shown to be implementable, integrable with key infrastructure, and effective, but with certain overheads involving the initialization of MPI as well as communicator construction. Small impacts on message-passing latency and throughput are noted. Open MPI was used as the implementation vehicle, but results here are also relevant to other middleware stacks.

I. Introduction

The MPI Sessions proposal specifies well-defined extensions to the MPI Standard. The proposal has reached a point of sufficient maturity to warrant development and evaluation of a prototype implementation. Interesting metrics for this evaluation include the practicality of implementation, the potential impact on basic MPI performance characteristics, as well as the usability for existing, large-scale MPI applications. This work has timely implications that impact standardization, including potential changes to the specification, as well as the establishment of community best practice for MPI-Sessions-enabled applications. While the authors chose to utilize Open MPI as the implementation vehicle for this prototype (due to its widespread use and efficient implementation of MPI), lessons learned here are nonetheless applicable to other MPI middleware implementations.

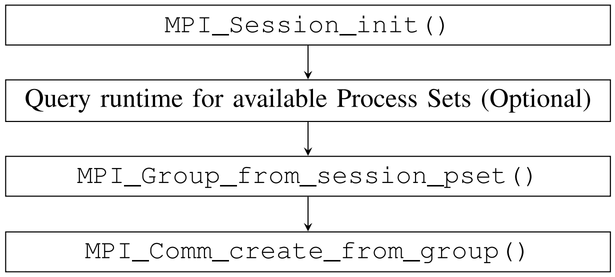

MPI_Session_init()
Query runtime for available Process Sets (Optional)
MPI_Group_from_session_pset()
MPI_Comm_create_from_group()

Fig. 1: Steps to create an MPI communicator from an MPI session handle. An application must first acquire a sessions handle, which can be used to query the runtime for available *process sets*. A group can be created using the session handle and one of the *process set names* defined by the runtime. This group can, in turn, be used to create an MPI Communicator using the `MPI_Comm_from_group()` function.

The proposed MPI Sessions extensions to the MPI API have been previously published [2]. There have been some changes to the API additions since the time of that publication, but the basic functionality of the MPI Sessions methods remain unchanged. To help in understanding the discussions in the following sections, we briefly review the key elements of the proposed MPI Sessions API here.

To use MPI Sessions, an application or component of an application must first obtain an MPI Session handle using the `MPI_Session_init()` function. This function allows the consumer software to specify the thread support level for MPI objects associated with this MPI Session, as well as the default MPI error handler to use for initialization of the Session and associated MPI objects. The MPI implementation must ensure that this method is thread-safe. Upon successful invocation of `MPI_Session_init()`, an MPI Session handle is returned. This function is intended to be local and light-weight. This MPI Session handle may be optionally used to query the runtime for available *process sets*. Process sets are identified by their corresponding *process set name*. The proposal requires MPI implementations to support two process sets: `mpi://world` and `mpi://self`. There are proposed new functions to query the runtime for additional implementation-specific or site-specific process set names. An MPI Group object is obtained using the `MPI_Group_from_session_pset()` function, which takes as inputs an MPI Session handle and process set name. This operation is also local and should be light-weight. The resulting MPI group can then be used as input to the `MPI_Comm_Create_from_group()` function to obtain an MPI communicator. This call is collective over the MPI processes in the supplied MPI Group. This sequence of steps is illustrated in Figure 1.

The remainder of this paper is organized as follows: Section II offers background and motivations for this work. Section III describes our prototype implementation; changes required for our prototype in PMIx, PRRTE, and Open MPI are described. Next, Section IV evaluates the prototype and our findings regarding MPI Sessions; in particular, this section covers evaluation criteria, experimental setup, benchmark results, and application results. Section V describes relevant related work. Finally, we offer conclusions and outline future work in Section VI.

II. Background and Motivation

Conceptually, each MPI session identifies a stream of MPI function calls, which manage a sequence of MPI operations. Associating that stream of instructions/operations with a particular execution thread, or with a particular software component, enables a new class of interface design opportunities, with implications both for communication performance and for programmer productivity.

A. Using Sessions as Parallel Regions

All current and previous versions of MPI require that MPI is initialized before (and must not be finalized before) any other MPI function calls can be made. There are a few exceptions to this general rule: mostly, MPI functions aimed at avoiding erroneous behavior when initializing MPI.

The MPI function `MPI_Init_thread()` was added in MPI-2, along with definitions of four thread support levels, to accommodate the expectation that OS processes were increasingly multi-threaded. (Note that calling `MPI_Init()` is equivalent to calling `MPI_Init_thread()` and requesting `MPI_THREAD_SINGLE`.) However, this additional initialization route was still not thread-safe itself—it cannot be called more than once in each MPI process, neither concurrently nor sequentially, even if `MPI_Finalize()` is called in between.

There is a recent trend away from static applications, with fixed allocation of resources throughout their execution, towards more dynamic applications, which can grow and shrink their resource usage depending on the computational needs of each phase of the application. The wide range of dynamic coding techniques includes task-based applications comprising of many short-running serial tasks and ensemble computations comprising of many long-running parallel sub-jobs. A typical example application is an ensemble weather simulation to investigate the effect of small perturbations in initial conditions—the European Centre for Medium-Range Weather Forecasts (ECMWF) has expressed a desire to initialize and re-initialize MPI for the Integrated Forecast System (IFS) [3].

There are also frameworks that offer higher programmer productivity by hiding the detail of how the execution is organized. For example, DASK-MPI [4] orchestrates concurrent execution of many parallel tasks and thus wants to re-initialize new MPI environments, each tailored to a different task.

MPI Sessions proposes a new function that initializes MPI: `MPI_Session_init()`. In contrast to the existing functions, this new function can be called multiple times and must always be thread-safe. These requirements solve the multi-threaded initialization problems [2] and enable succinct expression of fork-join parallel regions similar to concepts for OpenMP threads but applied to MPI processes, thereby supporting both ensemble applications and parallel-task execution models.

B. Using Sessions as Resource Isolation Domains

The MPI Sessions proposal includes a restriction that sessions must be isolated from each other. This is not strictly required to fix the multi-threaded initialization problem described in Section II-A. However, isolation offers interesting opportunities and does not appear to be overly burdensome for the intended usage models. MPI communicators already provide some isolation guarantees (akin to separate virtual communication fabrics). MPI sessions permit additional control of the granularity of that isolation.

Specifically, several related communicators can be associated together with the intention of sharing resources and, at the same time, isolating those resources from interference or contention from other MPI objects in other MPI sessions. Also, isolating MPI sessions permits additional user freedom while maintaining guarantees needed by MPI for performance optimizations. In particular, the user has the freedom that MPI objects from different sessions can be accessed concurrently while maintaining the guarantee that no MPI object will be accessed concurrently by multiple threads. That is, multiple threads can access MPI concurrently (each using MPI objects from a different session) even when the threading support level for each session is set to *funneled* (exactly one thread) or *serialized* (one thread at a time). If each session uses isolated resources (a choice made by the MPI implementation), then MPI does not need to protect against corruption caused by concurrent accesses. MPI can use a higher performance non-concurrent implementation even while the user is free to use concurrent accesses.

C. Using Sessions as Fault Isolation Domains

Exploring all possibilities for using MPI sessions to assist with fault tolerance is a work-in-progress. However, interesting topics for future research include using MPI sessions to:

- re-initialize MPI after each failure, potentially with fewer processes or including replacement processes;
- offer another possible option for limiting the scope of a failure by containing a cascading fault within a session.

a) Re-Initialization After Failure: Current proposals for MPI fault tolerance include *global restart* or *re-init* [5], [6], which attempts to return MPI to the initial state (that is, the state immediately after the call to `MPI_Init()` returns). However, in the case of a process failure, the original state cannot be recovered accurately, although the inclusion of replacement processes may be sufficient for practical purposes. MPI Sessions provide a new way to roll-forward in these circumstances—the application can re-initialize MPI after a failure and use whatever resources are available at the point of re-initialization. Redistributing application data is then entirely under user control, and the user has access to all information necessary to decide how to continue the computation.

b) Limiting Failure Scope: In general, the impact of a failure can be limited by identifying and removing anything that causes a failure to cascade or cause consequential failures in connected components.

In current MPI, the built-in `MPI_COMM_WORLD` communicator is a global-state object that exposes a single-point-of-failure for an entire MPI job. Using MPI sessions to initialize MPI avoids the creation of this built-in communicator and removes its single-point-of-failure.

In current MPI, connecting a client and server together results in a single set of connected processes. If the client fails while connected to the server, the default error handler for MPI is required to terminate all processes in both the client and the server. Other error handlers may be able to avert the automatic termination of all connected processes, but MPI may still be in a state that prevents further use as a communication library. Using MPI sessions to isolate resources used for internal coordination of server processes from resources used to manage client connections offers the possibility of a clean separation that avoids a cascade failure and permits the server to continue serving other clients.

III. Prototype Implementation

Our prototype implementation of the proposed extensions to MPI to support MPI Sessions functionality in Open MPI involved enhancements to three software components:

1) PMIx, a reference implementation of the Process Management Interface for Exascale specification;
2) PRRTE, the PMIx reference runtime environment; and
3) Open MPI, an open-source MPI implementation [7].

This section describes the enhancements to these software components required to implement our prototype of the MPI Sessions proposal.

A. PMIx Enhancements Facilitating MPI Sessions

PMIx [8], [9] defines a set of abstract interfaces that provide mechanisms to allow applications to interact with system management software (SMS) for scalable workflow orchestration. A reference implementation of the PMIx specification has been developed to validate that the defined methods are meeting the requirements of consumer applications. A reference PMIx runtime implementation (PRRTE) has also been developed to facilitate the development of PMIx-based applications on systems lacking native PMIx support.

PMIx group functionality was recently introduced into the upcoming version 4 of the PMIx specification [10]. PMIx groups are defined to be a set of processes that, having previously initialized PMIx as clients, wish to create a unique identifier for purposes of PMIx event forwarding or PMIx fence operations.

PMIx allows for both asynchronous and collective creation and destruction of groups. Asynchronous construction is based on an *invite, join* model that allows the initiator to replace processes that refuse the invitation or fail to respond within a specified time. Both construction methods provide notification of process failures, including that of the initiator so that survivors can either continue the operation, terminate the operation, or replace the failed process. Regardless of construction method, processes can depart the group at any time (with remaining participants receiving asynchronous notifications of the departure), or destroy the entire group via a collective operation. For simplicity, the MPI Sessions prototype used only the collective creation and destruction APIs shown in Figure 2.

The constructor method allows for the calling application to specify a string name for the group and the processes that are participating in it, and to request additional functionality by providing appropriate directives, including:

- specification of a leader process in the group,
- time-out duration for completion of the group constructor operation,
- request a Process Group Context Identifier (PGCID)—a unique 64-bit ID for the process group, assigned by the resource manager, which can be used by MPI as the communicator and/or session ID,
- request an event if a process in the group terminates without first leaving the group, and
- indication of whether a process terminating before joining the group is to be treated as an error.

The destructor method cleans up any PMIx/PRRTE internal state associated with the PMIx group and invalidates the use of the group identifier in any further PMIx operations. Both of these functions are blocking across the set of PMIx processes in the group and support a time-out feature to avoid deadlock due to a non-responsive participant (e.g., a process that has ceased to progress due to some internal error). Non-blocking versions of these methods are also available.

The development of PMIx group functionality in the PMIx reference implementation leveraged existing functionality including the PMIx event notification mechanism and PMIx server/client remote procedure call (RPC) mechanisms, as well as a generalized *all-to-all* data exchange mechanism in PRRTE. For scalability, the methods are implemented in a three-stage hierarchical fashion. First, the local processes managed by a single PMIx server each notify the server process that they

```
pmix_status_t PMIx_Group_construct(const char grp[],
                                   const pmix_proc_t procs[], size_t nprocs,
                                   const pmix_info_t directives[], size_t ndirs,
                                   pmix_info_t **results, size_t *nresults);

pmix_status_t PMIx_Group_destruct(const char grp[],
                                  const pmix_info_t info[], size_t ninfo);
```

Fig. 2: Collective creation and destruction PMIx APIs used in our MPI Sessions prototype.

are joining or leaving the group. Once all local processes that intend to participate in the operation have notified the server, the PMIx server initiates an all-to-all data exchange pattern with the PMIx servers managing the remaining processes in the group. Upon completion of the data exchange, each participating PMIx server notifies its local participants so they can return from the API call.

PMIx also defines two new query keys that can be used as arguments to the PMIx `PMIx_Query_info_nb()` API to discover both the number and names of existing PMIx groups, namely `PMIX_QUERY_NUM_PSETS` and `PMIX_QUERY_PSET_NAMES`. This feature proved useful in the development of supporting tools and for use in the asynchronous forms of the operations.

B. Prototype Implementation in Open MPI

The MPI Sessions prototype is based on the master branch of Open MPI available from the project's GitHub repository. To build the prototype, five major modifications and additions were made to Open MPI:

- development and implementation of a new communicator identifier (communication identifier (CID)) generator to support the creation of MPI communicators not derived from `MPI_COMM_WORLD`,
- update of point-to-point support (PML components) to accommodate changes to the CID generator,
- restructuring required to support invocation of MPI info, MPI error handling, and MPI Sessions attribute functions before the invocation of `MPI_Session_init()`,
- restructuring of MPI resource tear-down to support the ability for MPI Sessions to be initialized and finalized multiple times within a single application execution instance, and
- implementation of the interface extensions proposed for the MPI Sessions API.

In this section, we describe these changes in detail.

1) Dynamic Process Discovery Support in Open MPI: Historically, as part of the startup process in Open MPI, the implementation of `MPI_Init()` would query the runtime to discover all MPI processes in `MPI_COMM_WORLD`. This process is known as `add_procs`. This led to large overheads in memory and large startup costs. The developers of Open MPI addressed these issues by modifying the discovery procedure to call `add_procs` only for node-local processes. All other processes are discovered on first communication. This support was necessary to provide robust support for the Sessions Process Model defined by the MPI Sessions proposal.

2) Communicator Identifiers in Open MPI: The CID implementation for communicators in Open MPI uses a 16-bit integer representing the index into a local array of communicator objects. This representation was chosen to allow for fast and efficient (i.e., constant time, constant space) lookup of a communicator. The CID is used by the point-to-point messaging implementations (known as PMLs) in different ways depending on the underlying communication library. For the MPI Sessions prototype, we focused on the most general-use PML component: ob1. This component sends the CID as a part of a 14-byte matching header attached to the user data. This header is used by the receiving process to match the incoming message with the correct communicator and receive request. The header was designed to be as compact as possible to limit the overhead of messaging.

The CID representation chosen by Open MPI requires the CID to be consistent across every MPI process that participates in a communicator. To guarantee this property, Open MPI currently uses a consensus algorithm [7]. This algorithm performs a series of reduction operations on the user-supplied parent communicator. First, each MPI process attempts to store the new communicator at the lowest available local array index. An all-reduce is then performed to find the largest index across the group of participating MPI processes. If every participating process agrees on the same index, the algorithm terminates. If not, then the algorithm continues with the largest index determined in the previous round. Generally, the algorithm will finish after a small number of rounds but may end up searching the entire CID space if it becomes heavily fragmented.

As the above algorithm requires a parent communicator, it could not be used as-is to support the communicator constructors needed by the MPI Sessions prototype.

3) Changes to CID Generation to Support Sessions: One of the most significant challenges in implementing the MPI Sessions prototype was developing a fast method for generating a unique CID for each communicator created. Two primary factors motivated the development of a new CID generation algorithm:

- the lack of a parent communicator (for example, `MPI_COMM_WORLD`) to use for new CIDs generation;
- the use of the PMIx group APIs provides a robust (but potentially slow) way to create a unique 64-bit ID within an allocation.

The current MPI Sessions proposal provides a single new object constructor function named MPI_Comm_create_from_group(). Previous versions included other constructors for other MPI objects, for example, MPI_Win_allocate_from_group(). These functions provide an MPI group and a string tag instead of a parent communicator. No communicator is provided because no predefined communicator exists in the Sessions Process Model. For the prototype implementation, we chose to support these constructor functions by using the runtime group constructor support provide by PMIx. See Section III-A for more details on the PMIx groups constructor.

The PMIx group constructor returns a 64-bit PGCID that is guaranteed to be unique for the duration of an allocation (in the case of a batch managed environment). This PGCID could be used as a direct replacement for the existing CID. However, there are two major problems with taking this approach. First, if the prototype were to use this PGCID directly as the communicator CID, the associated field in the match header of ob1 would have to be expanded by at least 48 bits. This would require a reworking of the existing, optimized tag matching support, and likely lead to degradation in performance for shorter MPI messages. Second, acquiring the PGCID is a relatively expensive operation as it involves internode messaging between PMIx servers to generate the PGCID. Performance of existing MPI communicator constructors would be significantly degraded were the PGCID to be adopted as a direct replacement for the existing CID.

The prototype addresses both of these issues by introducing the concept of a 128-bit extended CID (exCID) and removing the constraint that a communicator's CID (array index) be consistent between all MPI processes in a communicator. The original CID is left intact so the optimized matching support in ob1 (and other PMLs) can be left intact.

Similar to the old CID, the exCID of a communicator is consistent between all MPI processes that participate in the communicator. We guarantee this property by careful construction of the exCID. The exCID is divided into two 64-bit fields. The first field contains the PGCID returned from PMIx when constructing a PMIx group. Since the PGCID is guaranteed to be non-zero, this field is set to 0 for the built-in World Process Model communicators. The second field is divided into eight 8-bit subfields. The subfields are used to generate exCIDs for derived communicators. The exCID structure also contains a field to keep track of the currently active subfield. When allocating an exCID from a new PGCID, this field is initialized to 7. When creating a derived communicator, for example, by calling MPI_Comm_dup(), the value in the active subfield of the parent communicator is incremented and assigned to the new communicator. This can be done 2^8 times before a new PGCID is needed. The active subfield field exCID of the derived communicator is decremented to ensure that there are no exCID collisions. If the active subfield of the parent communicator is 0, or the active subfield value is 255, or not all processes are participating in the communicator creation (MPI_Comm_create_group()), then a new PGCID is acquired and assigned to the new communicator.

For applications exclusively using the World Process Model, the prototype can use either the new exCID generator or the original consensus algorithm. The exCID generator is used exclusively when using a version of PMIx that supports group creation and the ob1 PML is in use. In all other cases, the prototype falls back to the original consensus algorithm.

4) PML Modifications: Changing the way communicator CIDs are generated required changes to the way the PML components use the CID. For the prototype, the ob1 PML was modified to support exCIDs. If a communicator has an exCID when sending the first message to a peer MPI process, an additional message header is generated and prepended to the existing match header. This header includes both the exCID and the sender's local CID for the communicator. Upon receipt of the first message, the receiver matches the exCID against the exCIDs of locally known communicators. The sending processes' CID is stored locally and the tag match field is updated to include the receiving processes' local CID for the communicator. The match is then processed normally. A response message is generated and sent back to the sender indicating the receiver's local CID for the communicator. This value is stored in existing space available in a per-process structure associated with each MPI communicator. For subsequent messages, the stored local CID for the remote process is used, and the standard optimized tag matching mechanism is employed.

The ob1 PML was chosen because matching is handled entirely within Open MPI. This is not the case for all the available PML components. In the future, we intend to implement support for exCIDs for all available PML components.

5) Restructuring to Support Dynamic Initialization: The MPI Sessions proposal allows for the creation of multiple MPI Sessions throughout the lifetime of the MPI application. In addition, it also allows additional MPI functions to be invoked before a call to one of the initialization functions: MPI_Init(), MPI_Init_thread(), or MPI_Session_init(). In particular, before initialization, the proposal allows for:

- calls related to MPI_Info objects including object creation, duplication, destruction, and the insertion and deletion of key/value pairs from an MPI info object,
- calls to create/destroy MPI_Errhandler objects, and
- calls related to session attributes creation, destruction, and value caching functions.

These functions must, additionally, all be thread safe as they may be called before the thread safety level is set. To support thread safety, the locks associated with MPI Info, error handlers, and attributes are always enabled. None of these code paths are on the critical path for MPI communication operations.

To support the additional functionality needed before initialization, the prototype modifies Open MPI to use a different approach to initializing and cleaning up different MPI subsystems. Instead of initializing the entire MPI library on initialization, as is done to support the World Process Model, and relying on a carefully ordered series of cleanup calls to various MPI subsystems as part of `MPI_Finalize()`, the prototype leverages a new cleanup callback framework provided by an Open MPI Open Platform Abstraction Layer—OPAL. As the application creates MPI objects, the subsystems needed for those objects are either initialized if not previously initialized, or an internal reference count is incremented for previously initialized subsystems. When a new subsystem is initialized, it adds its cleanup callback to the framework. As calls to `MPI_Session_finalize()` destroy MPI Sessions, these reference counts are decremented. When the last MPI Session has been finalized, the cleanup callbacks are invoked, and MPI-internal resources are released. The cycle begins again if the application creates a new MPI Session.

The legacy MPI-3 initialization and finalize functions (`MPI_Init()` and `MPI_Finalize()`) were restructured to create and finalize an internal MPI Session that also initializes the World Process Model built-in MPI objects. This removes the need for any duplicate code and allows the prototype to support the use of the new Sessions Process Model alongside the World Process Model.

6) Implementation of MPI Sessions Interfaces: The prototype implements the complete set of C interfaces that are defined in the MPI Sessions proposal. This includes the functions to create/finalize sessions, get info on process sets, create groups from process sets, and create MPI objects (communicators, windows, and files) from groups.

The implementation of the `MPI_Session_init()` function is required to be local-only. In the prototype, we initialize only the minimum set of MPI subsystems needed to support the MPI Session object. This includes initializing the Modular Component Architecture (MCA), info subsystem, point-to-point support, etcetera. The implementation of `MPI_Session_finalize()` releases all resources associated with the MPI Sessions object and tears down any resources not still in use by another MPI object.

The MPI Sessions proposal introduces the concept of an MPI process set. Process sets differ from MPI Groups in that they are simply names for lists of MPI processes. These names are either predefined (e.g., `mpi://world`, `mpi://self`) or implementation-defined. The prototype implementation defines three default process sets: `mpi://world`, which corresponds the process set in the World Process Model communicator `MPI_COMM_WORLD`; `mpi://self` (`MPI_COMM_SELF`); and `mpi://shared`, which is defined as the set of processes on the local node. Additional process sets are supported and must be provided by PMIx. When a process set is used to create an MPI Group, the prototype queries the underlying PMIx implementation to discover the associated MPI processes.

No changes were made to how Open MPI supports or represents MPI Groups. When requesting an MPI Group for `mpi://world`, the returned MPI Group is equivalent to calling `MPI_Comm_group()` on `MPI_COMM_WORLD`.

Support for creating MPI objects from MPI Groups is handled using the exCID generation algorithm. In the case of MPI Communicators, the exCID is used as the communicator identifier. In all other cases, the prototype first creates an intermediate communicator, then calls the MPI-3 object creation function with a parent communicator, and finally the intermediate communicator is freed. This was done to speed up the development of the prototype. We are actively looking at supporting MPI Window creation without the need for an intermediate communicator.

IV. Evaluation of the Prototype

In this section, we evaluate the performance of our MPI Sessions prototype using micro-benchmarks and a real scientific application. Our results show that the modifications introduced to support MPI Sessions functionality do impose a performance penalty over our baseline for MPI startup and MPI communicator construction, but have a negligible performance impact over the baseline for message latency and throughput.

A. Evaluation Criteria

Some of the changes made to Open MPI to support MPI Sessions functionality have the potential to impact performance. Significant refactoring of constructor and destructor methods for various Open MPI subsystems was required. These changes could impact MPI initialization (either via `MPI_Init()` or using new MPI Sessions procedures). Changes to MPI Communicator construction to support `MPI_Comm_from_group()` likewise could have an effect on the overhead for creating and using MPI Communicators. The exCID-based tag matching described in Section III-B3 could also potentially affect Open MPI performance. Although the initial connection setup between two processes using the exCID-based tag matching occurs only with the initial data exchange on a new communicator using the local CID approach, it still may impact application performance. Also, the additional level of indirection in generating the local CID based on the receiving MPI process could affect MPI message rate. The new procedure for creating MPI Communicators using PMIx may also add additional overhead. These concerns drove the selection of MPI benchmarks to evaluate the performance impact of supporting MPI Sessions.

In addition to quantifying the potential overhead introduced to support MPI Sessions in an MPI implementation, the ability of the prototype to achieve some of the goals of the MPI Sessions proposal described in Section II must be assessed. As a first step in such an evaluation, a Department of Energy (DOE) production multi-physics code was modified to make use of MPI Sessions in one of its component libraries.

B. Experimental Setup

Performance results were gathered from the systems detailed in Table I. Data were collected during regular operating hours, so the systems were servicing other workloads alongside but in isolation from our performance evaluation runs.

C. MPI Benchmark Results

The results obtained in this section were obtained using PMIx [13] at Git SHA `c4e1317b` and PRRTE [14] at Git SHA `a967a246`. The sessions-enabled MPI is also available on Github [15]. Note the Sessions prototype is on the *sessions-new*

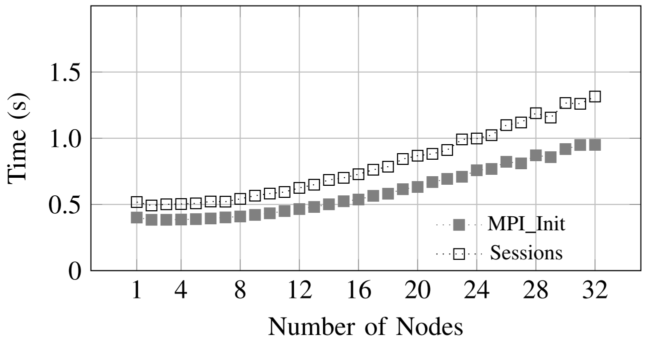
	Trinity	Jupiter
Model	Cray XC40	Cray XC30
OS	Cray Linux Environment	Cray Linux Environment
CPU	2× 16-core Intel E5-2698 v3	2× 14-core Intel E5-2690 v4
RAM	128 GB	64 GB
Network	Aries [11], [12]	Aries
Compiler	Intel 18.0.2	GCC 8.3

TABLE I: Hardware and software used for this study.

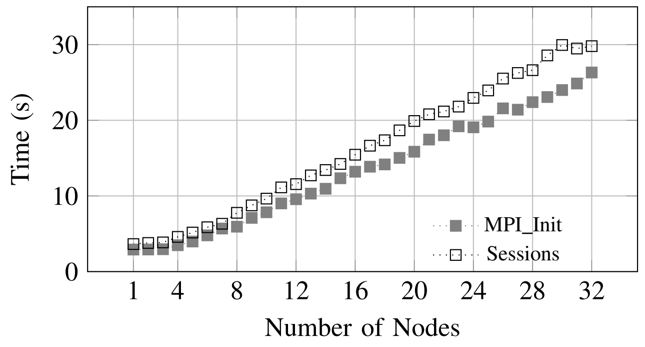
branch. For the baseline Open MPI, the master branch at Git SHA `ad29f70c` was used. Unless otherwise noted, the *prte* daemon and *prun* launcher were used to launch the applications. See the Sessions tests README [16] for instructions on using *prun*. This choice of software components and Open MPI baseline was made to reduce to a minimum the difference between the conditions under which the benchmarks were run.

1) MPI Startup Overhead: MPI initialization times using `MPI_Init()` were measured with the OSU `osu_init` benchmark [17]. Version 1.5.6 of the OSU benchmark suite was used in this evaluation. The benchmark was subsequently modified to time the `MPI_Session_init()`, `MPI_Group_from_pset()`, and `MPI_Comm_from_group()` sequence used to create a communicator equivalent to `MPI_COMM_WORLD` as depicted in Figure 1. These modified OSU MPI benchmarks and others described in this section are available on GitHub [18]. Figure 3 presents the timing data using both approaches. Results for the case of one MPI process per node and 28 MPI processes per node is shown. The MPI Sessions approach has some overhead (∼20%) compared to that used for MPI initialization with our baseline Open MPI release. An analysis of the time spent in sessions-related steps for creating an MPI communicator equivalent to `MPI_COMM_WORLD` shows that, for the case of 28 MPI ranks per node, about 30% of the time is spent in initializing MPI resources associated with the construction of the initial session handle, with the remainder spent in constructing the communicator. When using a single MPI process per node, the startup time using the sessions approach is dominated by the MPI resource initialization step which takes place when the first session is initialized. Optimizing the PMIx group constructor method should help in reducing the additional overhead currently observed in constructing an initial MPI communicator using `MPI_Comm_create_from_group()`. The relatively high overhead for MPI initialization on the system is attributable to the fact that the PMIx, PRRTE, and Open MPI components were installed on a relatively slow NFS-mounted file system. Tests with SLURM's srun launcher yielded similar times for the `osu_init` benchmark on Jupiter.

2) MPI Communicator Creation Overhead: Another area where support for MPI Sessions could potentially impact MPI performance is in overhead for MPI Communicator construction. One of the most commonly used MPI Communicator constructors is `MPI_Comm_dup()`. Timing overhead for this operation was measured. For these measurements, the `osu_init` benchmark was modified to measure the cost of `MPI_Comm_dup()` using both `MPI_Init()` and the equivalent set of operations when using MPI sessions. Figure 4 compares the time for the communicator duplication operation when using the two approaches to MPI initialization. Note the times reported are per iteration, not the time reported in the benchmark output. The data indicate that support for MPI Sessions does introduce some overhead compared to the approach taken in the current Open MPI release. This overhead is accounted for by the overhead of acquiring a PMIx group



(a) MPI initialization time with one MPI process per node.



(b) MPI initialization time with 28 MPI processes per node.

Fig. 3: MPI initialization times using `MPI_Init()` and MPI Sessions methods.

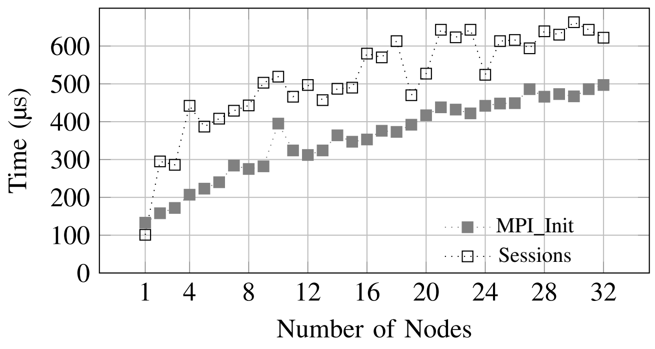


Fig. 4: `MPI_Comm_dup()` times using `MPI_Init()` and MPI Sessions methods with 28 MPI processes per node.

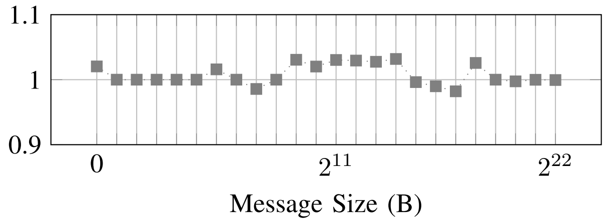
context identifier. We note that a more complex series of communicator constructor calls could take advantage of the new approach to CID generation, as more communicators could be created before needing to request a new PMIx group context identifier. Such a pattern of communicator constructor would negatively impact the performance of the current consensus algorithm in Open MPI owing to fragmentation of the CID space, while the exCID approach would not be suffer from this CID space fragmentation problem.

3) MPI Message Latency and Message Rate: The OSU `osu_latency` and `osu_mbw_mr` message rate benchmarks were also modified to use MPI Sessions for MPI initialization. These tests were carried out on a single node of Jupiter (Table I), as on-node message latency and message rate are often more sensitive to changes in the code path because the overhead for data exchange between processes using shared-memory approaches is much lower than the overhead involved for inter-node data exchange.

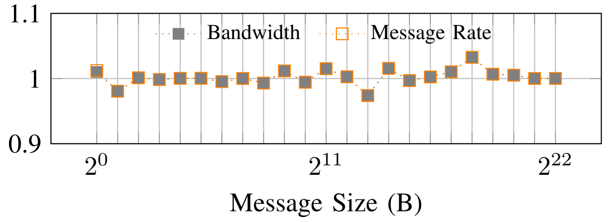
Figure 5 presents relative MPI latency (5a) and message throughput (5b, 5c) when using `MPI_Init()` and `MPI_Session_init()` to initialize MPI. As discussed in Section III-B3, the use of exCIDs and local CIDs could have a performance impact on the handling of MPI messages at both the sender and receiver. The results indicate that use of the exCID approach has a small effect on latency—in some cases showing an improvement over Open MPI's current message matching algorithm. The `osu_mbw_mr` results are more complicated. The test makes a call to `MPI_Barrier()` prior to entering the timing loop. In the case of two MPI processes, this barrier suffices to switch tag matching from using exCIDs to local CIDs prior to entering the timing loop. Thus the message throughputs presented in Figure 5b using `MPI_Init()` and `MPI_Session_init()` using only two processes are very similar. In the case of multiple MPI processes (Figure 5c), the `MPI_Barrier()` does not suffice for the paired MPI ranks to switch from exCID to the local CID tag matching scheme before entering the send/receive timing loop. This results in multiple sends being issued before the receiver's ACK is received by the sender, resulting in the switch over to using the faster local CID algorithm. When the OSU benchmark is modified to synchronize the process pairs prior to entering the timing loop, for example with a `MPI_Sendrecv()` operation, the message rates for both approaches to MPI initialization is essentially identical.

D. HPC Challenge Results

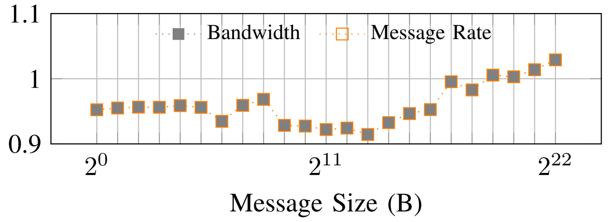
The High Performance Computing Challenge (HPCC) benchmark has a bandwidth and latency test which gives information about MPI latency when used in a more complex communication pattern than the OSU benchmarks. For this evaluation, we are particularly interested in the 8-byte Random and Natural order ring measurements. The observed latencies could be impacted by the exCID/local CID approach to MPI tag matching when using MPI Communicators derived from MPI Sessions.



(a) Relative latency by message size (2 processes).

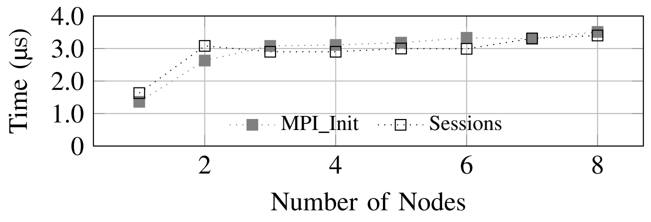


(b) Relative bandwidth and message rate by message size (2 processes).

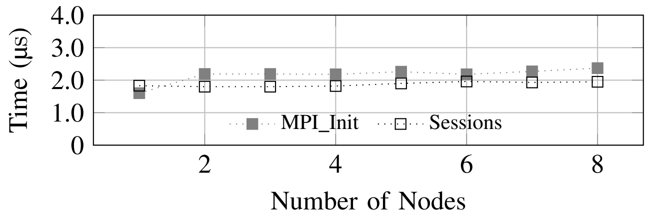


(c) Relative bandwidth and message rate by message size (16 processes).

Fig. 5: Relative performance results from the OSU latency and multiple bandwidth/multiple message rate micro-benchmarks.



(a) HPCC 8-byte random order ring latency.



(b) HPCC 8-byte natural order ring latency.

Fig. 6: HPCC 8-byte random and natural order ring latency results using `MPI_Init()` and MPI Sessions with 28 MPI processes per node.

Version 1.5.0 of the HPCC benchmark was modified to use MPI Sessions. Rather than replace the existing `MPI_Init()` and `MPI_Finalize()` usage in the benchmark's `main()` function, the `main_bench_lat_bw()` routine was modified to create its own MPI Session and use the resulting MPI Communicator for the bandwidth and latency component of the test. This serves to demonstrate the compartmentalization and backwards-compatible aspects of the MPI Sessions proposal. The rest of the benchmark could be left unmodified, yet still demonstrate the use of MPI Sessions within a subcomponent of the application.

Figures 6a and 6b present MPI 8-byte latencies for the random and natural order rings, respectively. The results reported for the modified HPC challenge uses sessions for the bandwidth and latency component of the benchmark. The baseline Open MPI was used with the unmodified application. In both cases, the latencies obtained using sessions are practically identical to what is achieved using the unmodified application and the baseline Open MPI.

E. Multi-Physics Application Results

We conclude this section with an evaluation of our prototype integrated into a production multi-physics application used at Los Alamos National Laboratory (LANL) named 2MESH. 2MESH comprises two libraries, L0 and L1. L0 simulates one type of physics on an adaptive structured mesh, and L1 simulates a different physics on a separate, structured mesh. L0 phases are MPI-everywhere and are interleaved with MPI+OpenMP phases—the parallelization strategy used by L1. For each computational phase, task schedules for MPI processes and OpenMP threads are optimized through the application's use of an open-source run-time library named QUO [19], [20].

QUO (as in *"status quo"*) is both a model and a corresponding implementation that facilitates the dynamically varying requirements of computational phases in coupled multithreaded message-passing programs. Specifically, QUO provides programmable facilities with modest overheads to dynamically reconfigure run-time environments for compute phases with differing MPI process counts, threading factors, and affinities. While the model is general, the current implementation focuses on Pthread-based MPI+X applications [20], [21].

Our MPI Sessions prototype was integrated into 2MESH through QUO, thereby obviating the need to modify the scientific application directly for our evaluation—an approach solely chosen for convenience. In particular, we modified `QUO_create()`, which is called only by L1, to include all relevant MPI session initialization logic that would otherwise be embedded directly into the target library. For our experiments, the application initializes MPI via `MPI_Init_thread()` before L1 calls `MPI_Session_init()` through `QUO_create()`.

To evaluate the performance of our prototype, we study two performance-critical operations: message-passing and process quiescence. For the former, we compare the performance of our MPI Sessions prototype implementation to that of Open MPI version 4.0.1, our performance baseline. For the latter, we

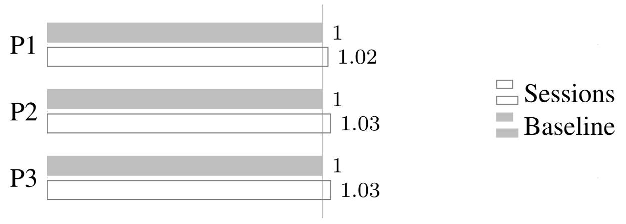


Fig. 7: Normalized 2MESH execution times.

focus on the characteristics of a performance-critical operation used to quiesce sets of MPI processes during QUO-enabled MPI+X phases: `QUO_barrier()`. In particular, we study the relative overheads of mechanisms provided by our prototype implementation (replacing those used by `QUO_barrier()` with a sessions-aware `MPI_Barrier()`) to the low-overhead ones used by QUO version 1.3 [20], which serve as our performance baseline.

Results were gathered from the Trinity system located at LANL (Table I). All experimental data were generated by executables built on top of the same software infrastructure, except for the differences already mentioned. For the three problems tested, two job sizes were chosen: 256 processes (P1, P2) and 1,024 processes (P3)—all fully subscribing the 32-core compute nodes detailed in Table I.

Our application results show for the three problems tested that our prototype imposes minimal ($\leq 3\,\%$) overhead over the baseline without MPI Sessions support. Figure 7 shows the normalized execution times calculated from averaged wall-clock times reported by the two 2MESH executables (Baseline and Sessions). The modest performance deltas observed are attributable primarily to the sub-optimal process quiescence mechanisms used in our prototype. In particular, we emulated a low-perturbation `MPI_Barrier()` by looping over alternating calls to `MPI_Ibarrier()` and `nanosleep()` until completion. Even so, our MPI Sessions prototype can replace `QUO_barrier()`—a key mechanism required to support efficient process quiescence in coupled, thread-heterogeneous MPI applications—through a standard interface requiring modest application source code perturbation ($\sim$20 source lines of code (SLOC)).

V. RELATED WORK

PMIx groups has some similarity to PVM dynamic groups [22]. As with PVM groups, a process can belong to multiple PMIx groups, and PMIx also supports the notion of processes joining and leaving groups. PMIx does not support the PVM concept of *instance numbers* in a group. It does, however, offer an *invite* operation to invite a process to join a PMIx group asynchronously. The PMIx event subsystem provides support for asynchronous notification of changes in group membership.

As discussed in Section II-C, MPI Sessions offers support for allocating and deallocating MPI resources within a process and generally strives to eliminate the notion of *global state* of MPI. Coupled with the dynamic nature of PMIx groups, MPI Sessions provides infrastructure that may simplify the implementation of MPI *global restart* [5], [6] and MPI *stages* [23] failure recovery schemes.

MPI Sessions offers the potential for MPI applications to express their resource requirements better than is available using `MPI_Init()`, thereby avoiding certain resource issues associated with MPI internal state and the implicit all-to-all connectivity required to support `MPI_COMM_WORLD`. Other approaches to improving the scalability for managing state required by `MPI_COMM_WORLD` have focused on reducing memory requirements for MPI Communicators and groups by optimizing associated data structures [24] and handling MPI processes as threads (*proclets*) within a single OS process [25] to reduce the memory requirements for MPI Communicators and MPI Groups. These approaches to MPI scalability are complementary to the MPI Sessions approach; indeed, our prototype can make use of the sparse group representation implemented in Open MPI.

Work such as [26] (older system named 'Legion') have group concepts for HPC in grid settings.

The 0MQ system [27] represents a cross-over between distributed computing and parallel computing and includes collective patterns of communication.

The Object Management Group's Data Distributed Service (DDS) [28] has aspects of dynamic groups and group-oriented communication used in distributed systems, defense-oriented applications, and also, recently, in IoT applications.

The authors are aware of an expansive literature of multicast and other group-oriented communication and group structures in distributed computing, networking, grid computing, and IoT, including fault-tolerant and dynamic approaches that do not directly relate to MPI, PVM, or PMIx-type applications and are apart in concept, performance, and functionality requirements from parallel middleware in HPC and scientific computing. For brevity, we omit references to such prior work.

VI. Conclusions and Future Work

We have presented a prototype of the MPI Sessions proposal and evaluated its performance against a baseline Open MPI release. This evaluation shows that support for MPI Sessions currently does impact the performance of MPI initialization and communicator construction, but has little impact on MPI latency or message throughput performance. The prototype also demonstrates the compartmentalization feature of MPI Sessions via its use in a multi-physics application and the HPC Challenge benchmark.

Follow-on work includes investigating some of the observed performance issues revealed by this evaluation. The Fortran Sessions API interfaces will be implemented. We also plan to expand the range of Open MPI PMLs and MTLs that can be used with the prototype's exCID approach to tag matching, in particular the OFI libfabric MTL and Open UCX PML. We also plan to explore using MPI Sessions in additional applications, including incorporating it in a task scheduling framework such as DASK and the Integrated Forecast System (IFS).

VII. Acknowledgments

This research was 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. Los Alamos National Laboratory is operated by Triad National Security, LLC for the U.S. Department of Energy under contract 89233218CNA000001. LA-UR 19-25747.

This work was performed with partial support from the National Science Foundation (NSF) under Grants Nos. CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This work was part-funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement 801039 (the EPiGRAM-HS project).

References

[1] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving MPI multi-threaded RMA communication performance," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225114

[2] D. Holmes, K. Mohror, R. E. Grant, A. Skjellum, M. Schulz, W. Bland, and J. M. Squyres, "MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 121–129. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966915

[3] O. Marsden, Private Communication, European Centre for Medium-Range Weather Forecasts (ECMWF), April 2019.

[4] Anonymous, "DASK-MPI: Deploy Dask Using mpi4py," downloaded May 21, 2019. [Online]. Available: https://pypi.org/project/dask-mpi

[5] I. Laguna, D. Richards, T. Gamblin, M. Schulz, B. De Supinski, K. Mohror, and H. Pritchard, "Evaluating and Extending User-Level Fault Tolerance in MPI Applications," *International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 305–319, 2016.

[6] M. Gamell, D. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales," vol. 2015-January, no. January, 2014, pp. 895–906, cited By 28.

[7] E. G. et al, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[8] 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

[9] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt, "PMIx: Process Management for Exascale Environments," *Parallel Computing*, vol. 79, pp. 9–29, 2018.

[10] R. Castain, "PMIx Group Extension to PMIx," https://pmix.org/pmix-standard/pmix-groups/, 2018.

[11] Alverson, Bob and Froese, Edwin and Kaplan, Larry and Roweth, Duncan, "Cray XC Series Network," *Cray Inc., White Paper*, vol. WP-Aries01-1112, 2012.

[12] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology," in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[13] "PMIx GitHub Repository," https://github.com/pmix/pmix.

[14] "PRRTE GitHub Repository," https://github.com/pmix/prrte.

[15] "GitHub Repository for Open MPI Sessions Prototype," https://github.com/hjelmn/ompi/tree/sessions_new.

[16] "GitHub Repository for MPI Sessions tests," https://github.com/hppritcha/mpi_sessions_tests.

[17] The Ohio State University, "MVAPICH Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, May 2019.

[18] https://github.com/hppritcha/osu-microbenchmarks-sessions, May 2019.

[19] S. K. Gutiérrez, "The QUO Runtime Library," https://github.com/lanl/libquo, January 2019, Los Alamos National Laboratory LA-CC-13-076.

[20] S. K. Gutiérrez, K. Davis, D. C. Arnold, R. S. Baker, R. W. Robey, P. McCormick, D. Holladay, J. A. Dahl, R. J. Zerr, F. Weik, and C. Junghans, "Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications," in *2017 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Orlando, Florida, 2017.

[21] S. K. Gutiérrez, "Adaptive Parallelism for Coupled, Multithreaded Message-Passing Programs," Ph.D. dissertation, University of New Mexico, 2018.

[22] V. Sunderam, G. Geist, J. Dongarra, and R. Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends," *Parallel Computing*, vol. 20, no. 4, pp. 531 – 545, 1994, message Passing Interfaces. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167819194900272

[23] N. Sultana, A. Skjellum, I. Laguna, M. S. Farmer, K. Mohror, and M. Emani, "MPI Stages: Checkpointing MPI State for Bulk Synchronous Applications," in *Proceedings of the 25th European MPI Users' Group Meeting*, ser. EuroMPI'18. New York, NY, USA: ACM, 2018, pp. 13:1–13:11. [Online]. Available: http://doi.acm.org/10.1145/3236367.3236385

[24] M. Chaarawi and E. Gabriel, "Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators," in *Computational Science – ICCS 2008*, M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 297–306.

[25] H. Kamal, S. M. Mirtaheri, and A. Wagner, "Scalability of Communicators and Groups in MPI," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 264–275. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851507

[26] A. S. Grimshaw and A. Natrajan, "Legion: Lessons Learned Building a Grid Operating System," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 589–603, 2005. [Online]. Available: https://doi.org/10.1109/JPROC.2004.842764

[27] P. Hintjens. (2011) 0MQ - The Guide. [Online]. Available: http://zguide.zeromq.org/page:all

[28] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ser. ICDCSW '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 200–. [Online]. Available: http://dl.acm.org/citation.cfm?id=839280.840571