LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-692709

MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale

D. Holmes, K. Mohror, R. Grant, A. Skjellum, M. Schulz

May 23, 2016

EuroMPI 2016
Edinburgh, United Kingdom
September 26, 2016 through September 28, 2016

Disclaimer

MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale

Daniel Holmes
EPCC, University of Edinburgh
dholmes@epcc.ed.ac.uk

Kathryn Mohror
Lawrence Livermore National Laboratory
kathryn@llnl.gov

Ryan E. Grant
Center for Computing Research
Sandia National Laboratories
regrant@sandia.gov

Anthony Skjellum
Auburn University
skjellum@auburn.edu

Martin Schulz
Lawrence Livermore National Laboratory
schulzm@llnl.gov

Wesley Bland
Intel Corporation
wesley.bland@intel.com

Jeffrey M. Squyres
Cisco Systems, Inc.
jsquyres@cisco.com

ABSTRACT

MPI includes all processes in MPI_COMM_WORLD; this is untenable for reasons of scale, resiliency, and overhead. This paper offers a new approach, extending MPI with a new concept called *Sessions*, which makes two key contributions: a tighter integration with the underlying runtime system; and a scalable route to communication groups. This is a fundamental change in how we organise and address MPI processes that removes well-known scalability barriers by no longer requiring the global communicator MPI_COMM_WORLD.

CCS Concepts

•**Software and its engineering** → **Message oriented middleware; Message passing;**

Keywords

Message Passing Interface; Scalable Programming Model

1. INTRODUCTION

Throughout its long history, from version 1.0 in 1994 until today's version 3.1 [2], the Message Passing Interface (MPI) has proven to be a widely successful approach for explicit parallel programming. MPI applications represent a multi-billion dollar development investment across governments, academia, and industry. Thus, the MPI Forum, the standardisation body for MPI, carefully considers proposed changes to the interface with respect to the impact the changes will have on existing MPI codes and works to ensure backwards compatibility, while standardising improvements in functionality, scalability, and performance.

However, in the twenty-two years since the first version of MPI, the landscape of high performance computing (HPC) has changed dramatically, largely in terms of the scale of HPC systems and MPI application runs. When MPI was introduced, large-scale jobs used 100's or 1,000's of processor cores [4], but today million-core jobs are not uncommon and this trend towards larger systems and larger jobs is expected to continue. The continued increase in the number of hardware cores will continue to lead to an increase in MPI process counts, even when mixing programming models in order to support the use of more than one core for each MPI process. Consequently, MPI implementations and applications are facing unprecedented scalability challenges [6].

A key challenge is the management of a potentially massive process space. MPI currently requires that all communication peers are included in the `MPI_COMM_WORLD` communicator during initialisation. In practice, active connections to peer processes are commonly made on a lazy basis. However, the amount of global state required to make these connections is more than strictly needed for many applications at massive scale. As the number of MPI processes grows, and consequently `MPI_COMM_WORLD` grows, the memory requirement and initialisation time for rank mappings and other large data structures can grow unacceptably high unless complex measures are taken to address the problem [9, 17, 19]. The requirement to include all possible communication peers in `MPI_COMM_WORLD` is an unfortunate legacy, since many applications do not incorporate communication patterns in which a given process must communicate with all other processes on a point-to-point basis. Even when third party libraries are involved, communication patterns that are truly all-to-all occur only in some applications. While applications do perform collective operations that span all MPI processes, scalable implementations of those operations rarely require any process to communicate directly with all other processes. Each process only communicates directly

DOI: 10.1145/1235

with a limited subset of the processes.

MPI applications must also deal with this massive increase in scale. One popular approach is to use hybrid programming models, such as MPI with threads, most commonly relying on OpenMP [3] to implement threading. This is attractive because OpenMP can be integrated into existing MPI application codes without refactoring the larger code base. However, it can require a large amount of developer time to choose the appropriate integration strategy and to tune the code to achieve desired speedup. Also, many of those integration strategies make use of per-thread network addressability leading to extremely large MPI process spaces.

In order to address these scalability challenges, we propose *MPI Sessions*, a fundamental change in how we organise and address MPI processes that remove the known scalability barriers by no longer requiring all possible communication peers to be included in `MPI_COMM_WORLD`. With MPI Sessions, we provide both backwards compatibility as well as a simple path forward to scalability for developers of MPI implementations and MPI applications. While scalability challenges can be addressed by intelligent, advanced MPI implementation techniques (like dynamic, lazy connection establishment on large communicators), our goal for MPI Sessions is to remove the need for "heroic developer efforts" to adapt MPI implementations and applications for the future scaling challenges. Instead, MPI Sessions facilitates these efforts with two key contributions: a tighter integration of MPI applications with the underlying runtime system; and a scalable representation of communication groups, effectively resolving the scalability issues of `MPI_COMM_WORLD`.

In the rest of this paper, we describe the motivation and reasoning behind MPI Sessions and its current status. Our goal is not to provide a finalised interface for integration into the MPI Standard. We expect that whatever is finally presented to the MPI Forum for possible inclusion into the standard will differ from what is described here. Section 2 provides background information on the problems that exist for MPI that we hope to solve with MPI Sessions. Section 3 details the current state of the MPI Sessions interface and explains how we expect it to be used. Section 4 discusses the potential impacts MPI Sessions may have on other proposals being considered for inclusion into the MPI Standard. Section 5 summarises related efforts, Section 6 discusses future directions, and Section 7 concludes this paper.

2. BACKGROUND

This section discusses some of the challenges currently faced by MPI, namely: scalability, library isolation, runtime integration, MPI's use and impact outside of HPC, fault tolerance within MPI and, finally, the code legacy of MPI with its corresponding backwards compatibility requirements.

2.1 Scalability

Since its inception, MPI has mandated the creation of a default communicator, called `MPI_COMM_WORLD`, which spans the entire process set associated with that MPI job at startup. The process space can be extended by creating intercommunicators to join together disparate MPI executions or by spawning additional sets of processes, but typically all communication is confined within the `MPI_COMM_WORLD` process set. The memory overhead that is required to store the mappings and state can be large (many GBs), which restricts the amount of memory available for computation. As systems are growing, the dense rank space mapping in this default communicator is becoming unacceptably large.

2.2 Isolation of Libraries

MPI has always been process-oriented; there is an implicit assumption that the code for each process is monolithic or, at least, that libraries are sub-programs that depend on the main program. This assumption manifests, for example, in the requirement that MPI can only be initialised once in each process and there is no thread-safe method to negotiate which thread should perform the initialisation. This means that the only option available to programmers is to have the main application initialise MPI and write all libraries such that they assume MPI will be initialised for them in a timely manner by an agent external to the library. The method of checking the assumption of timely initialisation was made thread-safe by changes in MPI 3.1, but did not address the more fundamental issue.

2.3 Interaction with Runtime Systems

Historically, the MPI Forum has been conservative about including functionality in the standard for allowing interaction with runtime systems; that is, the operating system, resource managers, job launchers, or any layer that an MPI application depends on for services beyond what is provided through the MPI API. These services include managing or querying processes as well as raising and handling signals. While multiple interfaces for interacting with runtime systems have been suggested in the past [1], they have largely been rejected.

There are exceptions to this rule in the MPI Standard, with features such as `MPI_Get_processor_name`, which returns a string indicating the name of the processor on which the MPI process is running; memory allocation routines, for potentially optimising message-passing and remote-memory-access (RMA) operations; and dynamic process creation, for extending the currently running MPI job by starting new MPI processes. The MPI Forum felt that each of these features could be added because they are sufficiently useful to application developers and only require functionality from an external runtime system that is commonly provided by all known and foreseen computing systems.

However, many have argued that a greater interaction with runtime systems could benefit many MPI applications. For example, information about the physical topology is useful for mapping MPI processes onto allocated nodes to improve communication performance [1, 15, 19]. However, this is not a straightforward process with the current MPI functionality because there is no interaction with the runtime system to get this information beyond `MPI_Get_processor_name`, which does not necessarily give any topology information.

Another example is in the area of fault tolerance. While we discuss this topic later in this section, it is worth noting here that interaction with runtime systems could be beneficial to assist with notification of failed processes and with requesting replacement hardware or processes from the resource manager.

2.4 Applicability of MPI Outside HPC

MPI has seen some use outside of the area of HPC. Some of the most notable examples are interfaces for languages, like Java, Python, and R, that are not typically used for

HPC, but are widely used for scientific computing at workstation scale. However, outside of scientific computing, MPI is not typically used, even though it has support for the connect/accept client-server model typically used in consumer applications. MPI connect/accept functions have some elements of fault tolerance, e.g., the ability to disconnect from broken connections, through the `MPI_COMM_DISCONNECT` function, without causing failure of the MPI processes. However, this only applies in cases where the processes are distinct MPI jobs; that is, they do not share the same `MPI_COMM_WORLD`. Therefore, it would be helpful to have limited connection groups to avoid unnecessarily aborting processes that are hosting working servers/clients. Such improvements should make MPI easier to leverage in non-HPC environments and allow for MPI performance benefits to be utilised by a larger portion of the computing community.

2.5 Fault Tolerance

The `MPI_COMM_WORLD` communicator is always impacted by any non-trivial error because its functionality depends on the set of all processes involved in the job. MPI Sessions break this guaranteed impact by not requiring the existence of a communicator that spans the entire job. This means that only processes which need to communicate with the failed node are required to react to a failure. Therefore, error handling and fault tolerance are easier to accomplish as errors and faults do not necessarily have global job impact and recovery can be done only where it is needed. The MPI Forum is currently considering mechanisms to determine whether communicators have experienced a fault, to remove faulty members of a communicator, and to repair a communicator after a fault without triggering a complete application abort [8].

2.6 Backwards Compatibility

Backwards compatibility requires that `MPI_COMM_WORLD` and `MPI_COMM_SELF` are available to all MPI jobs, `MPI_Init[_thread]` enables all MPI functionality, and `MPI_Finalize` releases all resources used by MPI. As long as these objects and functions are still supported and still exhibit the same behaviour, then adding new mechanisms to access MPI functionality can be considered to be backwards compatible.

The additions to MPI to support MPI Sessions proposed in this paper do not prevent an existing standard-compliant MPI program from executing correctly without code changes. Sections 3.7 and 3.8, discuss how an existing MPI program can be modified incrementally to take advantage of MPI Sessions in some parts of the code without requiring changes in the rest of the application.

3. MPI SESSIONS

The basic intent behind MPI Sessions is to introduce a concept of isolation into MPI by relaxing the requirements for global initialisation that currently produces a global communicator. Each MPI Session creates its own isolated MPI environment, potentially with different settings, optimisation opportunities, and communication data structures.

This concept of MPI Sessions and its isolation properties can be used to overcome several limitations of the MPI interface tied to having only a single, global group of processes:

- Threading levels can be handled on a per session basis, allowing MPI libraries to optimise for different threading styles within a single application.
- When using separate sessions per thread, any thread can use MPI at any time, without the complexity and overheads of requiring `MPI_THREAD_MULTIPLE` in the main application.
- MPI-based libraries can create their own unique sessions and thereby be isolated from decisions and tuning steps taken in the main application. In the extreme case, this enables the use of multiple independent MPI-based libraries even in non-MPI-based codes.
- Several components (libraries, code modules, infrastructure elements) can make use of MPI without concern for whether MPI has already been initialised or finalised beforehand; that is, without having to determine whether MPI is in a usable state.
- Operations and events that previously had global scope can happen on a per process-set basis, avoiding global impact. This can be especially useful for error handling, since the scope of an error can now be restricted to a subset of processes.

There are two identified uses for an MPI Session: to query the runtime system (see section 3.3) and to create scalable `MPI_Group` objects (see section 3.4). A scalable `MPI_Group` can then be used to create a scalable `MPI_Communicator` (see section 3.5).

3.1 MPI Session Handles

`MPI_Session`s are local handles to the MPI library; they contain no global state. Consequently, managing session handles is intended to be light-weight, requiring no significant resources to create or maintain them. Session handles themselves are immutable, although objects created from them or linked to them can be changed. Each MPI Session forms an isolation domain; a unique identification for a sequence of future MPI function calls. A new MPI Session can be created and destroyed at any time during the execution of the program.

3.2 Creating and Destroying Sessions

The lifetime of a session begins when the constructor is called and ends when the matching destructor is called. Listing 1 shows prototypes for these two functions.

```
int MPI_Session_init(
        MPI_Info info,
        MPI_Errhandler errhandler,
        MPI_Session *session);
int MPI_Session_finalize(
        MPI_Session *session);
```

Listing 1: Session constructor and destructor

The `MPI_Session_init` function creates a new session and returns a valid handle to it. The error handling behaviour of MPI during creation of this session is controlled by the "errhandler" parameter passed in to this function. The "info" parameter allows for future expansion by offering the user a mechanism to supply information that MPI can use to guide the creation of this session. The `MPI_Session_finalize` function destroys a session and sets the session handle to `MPI_SESSION_NULL`, which represents an invalid session.

These functions are defined to be thread-safe because it is intended that they can be called from library code without reference to, or coordination with, the main application or other libraries. Multiple libraries could, for instance, create sessions concurrently so that each one has its own handle to the MPI library and can interact with MPI independently.

3.3 Named Sets of Processes

We introduce the concept of "named sets" of processes that are discoverable by querying the local runtime system. MPI processes can query which named sets exist and then use them to create a matching MPI group. An MPI group can then be used to create an MPI communicator.

For simplicity and as a proof of concept, we initially define sets to be static and immutable; that is, their memberships do not change during the lifetime of the program. Allowing dynamic sets, whose membership can change in response to a variety of events, is discussed as future work in section 6.

The names of all sets known to the runtime can be retrieved using `MPI_Session_get_names`, shown in Listing 2.

```
int MPI_Session_get_names(
        MPI_Session session,
        char **names);
```

Listing 2: Obtaining set names from the runtime via a session

The data returned in the "names" parameter is formatted as an argv-style, null-char-terminated array-of-strings. The memory for the array of strings (called "names" in the listing) is allocated by MPI and will be freed by MPI when the session is destroyed. It is strongly encouraged that each string name is formatted as a URI.

In order to smooth the transition for legacy code, two named sets are guaranteed to exist: "mpi://WORLD" and "mpi://SELF". The set with name "mpi://WORLD" refers to a set containing all processes that were started by this job execution. This set does not include dynamically created processes; those created, for example, by a call to `MPI_Spawn`. It represents the set of processes that would ordinarily be members of the default communicator, `MPI_COMM_WORLD`. The set name "mpi://SELF" refers to a set containing only the calling process. It represents the set of processes that would ordinarily be members of the communicator, `MPI_COMM_SELF`.

```
int MPI_Session_get_info(
        MPI_Session session,
        char *name,
        MPI_Info *info);
```

Listing 3: Obtaining information about a set from the runtime with via a session

The function prototyped in Listing 3 exposes information about a particular named set by providing an `MPI_Info` object. The only key that is mandated for this `MPI_Info` object by this proposal is called "size" and its value is the number of processes in the named set. Further keys could be included by particular runtime systems with implementation-dependent meanings. The information provided by the runtime about each of the sets can guide the user's decisions about which groups to create in order to gain access to the exact resources required.

3.4 Creating Scalable Groups

Any named set of processes that is exposed by a session can be converted into an `MPI_Group` using the proposed new function, shown in Listing 4. If the internal description of the set obtained by MPI from the runtime is scalable, then the internal representation of the resulting `MPI_Group` can also be scalable. It is common-place for the meta-information maintained by runtime systems about all system resources to be stored in a scalable manner. For example, the processors might be numbered sequentially according to some pre-defined pattern so that a single {start, count} tuple can describe a NUMA region, a node, or a rack.

```
int MPI_Group_create_session(
        MPI_Session session,
        char *name,
        MPI_Group *group);
```

Listing 4: Creating a group from a session

The MPI Standard already defines a rich set of functions for manipulating groups of processes. This allows any arbitrary group to be created from the members of the initial groups provided by the runtime. Some possible group manipulations may produce a non-scalable storage requirement to represent the resulting group. For example, a call to `MPI_Comm_split` with a single value for the "color" parameter and randomly chosen values for the "key" parameter would require storage of the mapping from child-rank to parent-rank. Non-scalable group operations must be avoided to achieve good scalability.

3.5 Creating Scalable Communicators

The MPI Standard already defines a function for creating a communicator from a group. However, that function requires a parent communicator that contains a super-set of the processes represented by the target group. The parent communicator is used by MPI to orchestrate the communication needed to create the new communicator. In the absence of a suitable parent communicator, such as having created a group from a session as in Section 3.4 (and ignoring the existence of `MPI_COMM_WORLD`), this existing function cannot be used. Therefore, we propose a new function that does not require a parent communicator but instead relies on the same mechanisms that MPI libraries currently use during initialisation to create and wire-up the default built-in communicators, in particular `MPI_COMM_WORLD` itself. A prototype for this new function is shown in Listing 5.

```
int MPI_Comm_create_group_X(
        MPI_Group group,
        char *tag,
        MPI_Info info,
        MPI_Errhandler errhandler,
        MPI_Comm *comm);
```

Listing 5: Creating a communicator from a group

The input "group" and output "comm" parameters are used to supply the targeted group and to return the new communicator respectively. The "tag" parameter is needed for the same reason it is needed in the existing, communicator-based, function: to disambiguate multiple concurrent operations involving overlapping groups of processes. The "info"

and "errhandler" parameters are included to provide additional useful customisation options that are not available from existing communicator creation functions in MPI. The "info" parameter provided will be applied to the new communicator and could also guide its creation method and supported functionality. The "errhandler" parameter provided will be attached to the new communicator and could be used during creation in the case of certain classes of failure.

If the internal group representation is scalable then the internal representation of the resulting `MPI_Comm` can also be scalable. These functions also provide a means to create any communicator containing any group of processes without referencing `MPI_COMM_WORLD` or requiring its existence.

The resulting communicators can be manipulated in the same ways as other communicators, including being used to create inter-communicators with `MPI_Intercomm_create`.

3.6 Creating Scalable Topologies

Topology communicators enable the use of neighbourhood collective operations in MPI and allow MPI to optimise the internal representation and implementation of a communicator, e.g., by using knowledge of the underlying hardware topology. MPI currently requires a parent communicator when creating a topology communicator. The parent communicator must contain a superset of the MPI processes that will be contained within the topology communicator.

By using an MPI Session, it is possible to omit the intermediate step (and consequent use of resources) of creating a non-topology parent communicator. This is achieved by directly forming the `MPI_Group` representing the appropriate MPI processes and creating the topology communicator from that group. We propose four new functions that all include parameters needed for communicator creation (identical to the non-topology function from 3.5) and each add specific parameters needed for a particular type of topology;

- cartesian, shown in Listing 6;
- graph, shown in Listing 7;
- distributed graph from all known edges, shown in Listing 8; and,
- distributed graph from all adjacent edges, shown in Listing 9.

```
int MPI_Comm_cart_create_group(
        MPI_Group group,
        char *tag,
        MPI_Info info,
        MPI_Errhandler errhandler,
        int ndims,
        int dims[],
        int periods[],
        int reorder,
        MPI_Comm *comm);
```

Listing 6: Creating a cartesian communicator from a group

```
int MPI_Comm_graph_create_group(
        MPI_Group group,
        char *tag,
        MPI_Info info,
        MPI_Errhandler errhandler,
        int nnodes,
        int index[],
        int edges[],
        int reorder,
        MPI_Comm *comm);
```

Listing 7: Creating a graph communicator from a group

```
int MPI_Comm_dist_graph_create_group(
        MPI_Group group,
        char *tag,
        MPI_Info info,
        MPI_Errhandler errhandler,
        int nedges,
        int sources[],
        int degrees[],
        int destinations[],
        int weights[],
        int reorder,
        MPI_Comm *comm);
```

Listing 8: Creating a distributed graph communicator from a group by specifying all known edges

```
int MPI_Comm_dist_graph_adj_create_group(
        MPI_Group group,
        char *tag,
        MPI_Info info,
        MPI_Errhandler errhandler,
        int indegree,
        int sources[],
        int sourceweights[],
        int outdegree,
        int destinations[],
        int destweights[],
        int reorder,
        MPI_Comm *comm);
```

Listing 9: Creating a distributed graph communicator from a group by specifying all adjacent edges

Attaching a topology to a communicator during its creation is an additional step that is, semantically, entirely orthogonal to the creation itself. A simple implementation for these functions would be to create a communicator from the group that does not have a topology (using the function described in Section 3.5) and then use that communicator as the parent in a call to one of the existing topology creation functions in MPI. All the parameters needed for these two internal function calls are provided in each of the new topology-from-group functions.

3.7 Shunning `MPI_COMM_WORLD` and `MPI_COMM_SELF`

MPI sessions alleviate the problem of having a default `MPI_COMM_WORLD` by not requiring the creation of a world communicator during MPI start-up. All communicators that the application code needs for useful MPI operations, such as point-to-point, single-sided, or collective communication, can be created without requiring the existence of `MPI_COMM_WORLD`. The primary benefit of only creating communicators that require a subset of the entire rank/process mapping space is a reduction in the overhead required for enabling typical MPI communication to occur. There are other approaches that could also provide this benefit. One such

approach is to use lazy initialisation of dynamic connections. Open MPI uses such a dynamic scheme, in which it uses a sparse, rather than a dense, mapping and only fetches addresses of, and establishes connections to, processes when they are needed. MPI Sessions allow exactly the same approach but can also guide (and simplify implementation of) the heuristics that decide when to create and destroy these dynamic connections.

3.8 Downplaying `MPI_Init` and `MPI_Finalize`

Removing `MPI_Init[_thread]` and `MPI_Finalize` from the MPI Standard would immediately break backwards compatibility for every single MPI program that currently exists. However, sessions offer a semantically independent route to access all the functionality of MPI. The compromise we propose is that, in the future, these functions will still exist but it will no longer be mandatory to call `MPI_Init[_thread]` before accessing MPI functionality and it will no longer be mandatory to call `MPI_Finalize` before exiting the MPI process. MPI will initialise and finalise itself when needed, and partial initialisation or finalisation is permitted.

Sessions provides backwards compatibility for traditional MPI applications that use `MPI_COMM_WORLD` by re-defining `MPI_Init[_thread]` and `MPI_Finalize` functions to be the constructor and destructor of the built-in communicators, `MPI_COMM_WORLD` and `MPI_COMM_SELF`. In this way, MPI can support all current function calls, with identical semantics, and can provide the same optimisations, e.g., dynamic sparse communicator data structures, as a current MPI implementation would be capable of doing.

In contrast, applications that fully adopt the Sessions model will not call `MPI_Init[_thread]` and `MPI_Finalize` at all. Instead, MPI will be initialised and finalised implicitly upon the creation and destruction routines of MPI objects (see Listing 10 for an example). The rationale behind this decision is that the creation and destruction of MPI objects is not generally considered as performance critical as other MPI functionality, such as point-to-point message passing or RMA operations, so the additional time for performing these actions will not be in the critical path. Additionally, this allows MPI implementations to initialise lazily. For example, on an MPI Datatype create, only the functionality for MPI Datatypes will need to be instantiated, if the MPI implementation chooses. The MPI implementation can use reference counting to track when the last MPI object is destroyed and can then finalise the state of MPI.

```
int main() {
    /* Create a datatype.
       This will initialise MPI */
    MPI_Type_contiguous(2, MPI_INT,
                        &mytype);
    /* Free the datatype
       This will finalise MPI if it is
       the last object */
    MPI_Type_free(&mytype);

    /* Valgrind clean at point of exit */
    return 0;
}
```

Listing 10: Initialise and finalise MPI during object create and free

3.9 Example usage

A simple example usage of session functionality is presented in Listing 11. The example provides a basic template for a halo-exchange domain-decomposition code that uses MPI I/O for reading/writing data and neighbourhood collective MPI functions for communication during an iterative calculation (not shown for brevity). MPI may, or may not, be in use by other parts of the application before, during, or after the execution of this function. The set name passed as an argument could be a pre-defined set name, such as `"mpi://WORLD"`, or a name obtained from a session query in another part of the code.

```
int haloExchange(char* setName) {
    /* Create a session */
    MPI_Session_init(info, err,
                     &mySession);
    /* Get INFO about the input set */
    MPI_Session_get_info(mySession,
                setName, &setInfo);
    /* Get size of the input set */
    MPI_Info_get(setInfo, "size", len,
                 &strSize, &flag);
    /* No longer need info handle */
    MPI_Info_free(&setInfo);
    size = atoi(strSize);
    if (size < bigEnough) return -1;
    /* create a group */
    MPI_Group_create_session(mySession,
                setName, &myGroup);
    /* No longer need session handle */
    MPI_Session_finalize(&mySession);
    /* Get size of group */
    MPI_Group_size(myGroup, &size);
    /* Calculate desired topology */
    MPI_Dims_create(size, ndims, dims);
    /* create a cart comm */
    MPI_Comm_cart_create_group(myGroup,
        tag, info, err, ndims, dims,
        periods, reorder, &myCartComm);
    /* No longer need group handle */
    MPI_Group_free(&myGroup);
    /* MPI I/O using myCartComm */
    /* neighbourhood collectives */
    MPI_Comm_free(&myCartComm);
    return 0;
}
```

Listing 11: Example usage of new sessions for a simple halo-exchange template code

The function shown in the listing initiates a session, obtains information about the set, creates a group from that set, and creates a topology communicator from that group. At each stage, resource-sensitive decisions and actions can be taken. The information about the set provides the size of the set, which determines if the set contains enough processes to proceed. The `MPI_Group` allows group maniuplation functions (not shown for brevity) to refine which processes will be used by this function. Only one communicator is created, with a topology but with no requirement for a 'parent' communicator. Once the topology communicator exists, all other MPI objects can be freed, which releases the maximum resources for the application code.

For each MPI process, the semantic correctness of this function does not depend on MPI function calls made by other functions or by other threads within that MPI process.

4. CONSEQUENCES AND PROBLEMS

The changes to MPI described in Section 3 have implications for other active proposals currently being considered by the MPI Forum. The consequences for major topics are set out in this section; MPI endpoints is discussed in Section 4.1 and fault tolerance is discussed in Section 4.2.

4.1 MPI Endpoints

In the current MPI Standard, there are no group manipulation functions that can create new members; all existing group functions select the members for the new group from the members of one or more groups provided as input parameters. The current proposal for endpoints [22, 10] provides a way to create a communicator with additional ranks that do not correspond to any ranks in other communicators. It is possible to obtain the underlying group from any communicator, so the proposed new endpoint communicator creation function gives a round-about route to create a group with new members that did not previously exist in any other group.

An alternative way to create an endpoints communicator would be to first form the endpoints group and then to create a communicator from that group. Instead of the communicator creation function currently proposed for endpoints (*i.e.,* `MPI_Comm_create_endpoints`), this would need a group creation function; that is, `MPI_Group_create_endpoints`. An endpoints communicator could then be created directly from this group, without supplying a parent communicator, by using the function described in section 3.5.

This alternative route avoids the need to create the parent communicator in addition to the endpoints communicator, which could reduce the pressure on resource usage.

4.2 Fault Tolerance

When communicators fail, the task of rebuilding them can be difficult, if not impossible, with traditional MPI implementations. The current proposal for fault tolerance being considered by the MPI Forum (*i.e.,* ULFM [8]) would be potentially impacted by MPI Sessions. This impact results from the overlap in capabilities between the ULFM proposal and features in MPI Sessions[1].

With MPI Sessions, the task of recovering failed communicator(s) is not made easier from the application side, but rebuilding communication mechanisms is easier for a system using MPI Sessions. Replacing failed processes in MPI can be difficult to achieve, especially within the job constraints offered by the scheduler/launcher. Sessions is an interaction method with this runtime that can help ease recovery through easier expression of the needs of the MPI job in terms of it's current allocation of hardware.

5. RELATED WORK

Many efforts have been undertaken to improve the scalability of MPI. In particular, a study by Balaji et al. [6] outlined potential problems with MPI implementations and with the MPI Standard itself that prevent or limit scalability. The authors identified several facets of MPI that implementations must focus on in order to achieve scalable MPI libraries, including point-to-point communications, process-to-communicator maps, and memory usage of MPI data-structures.

Over the years, there have been multiple efforts to address the scalability of MPI implementations. Some have worked to save memory by optimising the data structures used for MPI communicators and groups [9, 17, 19]. Others have focused on developing threaded implementations of MPI, such that the MPI processes are implemented as threads instead of operating system processes [16], which can greatly reduce the amount of memory used by the MPI library [20], and also significantly improve performance [18, 21]. Additionally, there has been research toward scalable collective algorithms [5, 19] as well as point-to-point algorithms [11].

Generally, there have not been substantial changes to the MPI Standard to address scalability issues identified by the community [6]. MPI Sessions addresses several of these key scalability issues. Sessions address the scalability of a greatly increased rank space, which occurs when threads become a communication target that includes their own addressable ranks, as with the MPI endpoints proposal [10, 22], or due to threaded MPI implementations [18, 20, 21]. Additionally, Sessions introduce a method for MPI applications to interact with runtime system software, which can provide benefit in several ways. For example, process set names can be based on the structural layout of machines, which can give applications a solid basis for re-ordering application data to optimise communication performance [1, 15, 19]. Finally, Sessions can help address problems with dynamic process creation [1] and fault tolerance by enabling an application to query the runtime resources allocated to it and to establish MPI connections with processes that were dynamically created during its execution.

6. FUTURE WORK

Moving forward, there are several aspects of MPI Sessions that will need to be considered carefully. Primarily, these center around adding dynamic behavior to the generally static view offered by MPI today.

One of these in particular is the notion that the known set of MPI processes in a job could be dynamic, with MPI processes entering and leaving the MPI run environment. This might occur due to MPI process failures and subsequent replacements, or due to growing or shrinking job demands that result in adding and retiring MPI processes over time. While we can see the potential benefits of a dynamic MPI process space, this must be designed carefully. MPI was designed with a static process space primarily for performance and convenience reasons [13].

At the time MPI was being designed, there was a competing interface in use called the Parallel Virtual Machine (PVM) [7] that was dynamic in nature. In PVM, processes and hosts could come and go during a run, and PVM provided an interface with resource managers. In contrast, MPI was designed with a static process space for ease of programming and performance, and an interface agnostic to resource managers for portability [13, 14]. The dynamic nature of PVM gave it natural support for fault tolerance, and for applications with growing and shrinking computing demands. Also, PVM was implemented as a virtual machine, so hosts

[1]Other emerging approaches to MPI fault tolerance would potentially benefit from MPI Sessions too.

could be added and deleted at will. However, PVM suffered from some drawbacks due to its dynamic design, including race-conditions and issues with performance and portability [13, 14]. As we move forward with MPI Sessions and consider introducing dynamic behavior into MPI, we must look back to apply the lessons learned from the PVM design, and avoid the problems that prevented PVM's success.

Another dynamic behavior that MPI Sessions could facilitate is the easy integration of distinct MPI jobs via communicators that span multiple sets of processes. The URI naming scheme proposed in the interface for MPI Sessions could allow sets of processes to be easily identified through a publish/discover mechanism. While the current MPI Standard provides this capability via the publish/lookup and connect/accept mechanisms, the existing capability is cumbersome and may not perform well [12]. As a result, the existing functionality is not often used in real applications. We anticipate that the mechanisms can be cleaner and easier to use by applications with the Sessions semantics and potentially have better performance due to the information made available from the resource manager. However, we will need to investigate this further, especially with respect to issues of scope, security, and scalability.

7. CONCLUSIONS

A key challenge for MPI at extreme scale is the handling of large amounts of concurrency on future systems and the management of a massive number of communicating processes. Our concept, MPI Sessions, works to address this issue while also tackling issues that are legacy concerns dating back to 1994 with version 1.0 of the standard, notably the issues surrounding the predefined all-to-all virtual communication fabric and state associated with `MPI_COMM_WORLD`.

Recently, the MPI Forum created a new working group dedicated to further the development of the MPI Sessions interface. The development will be based on the work presented here, and will consider current and future application needs and use cases from industry, academia, and national laboratories. We expect the interface will change before it is presented to the MPI Forum and potentially included in the MPI Standard. However, the goals and underlying rationale for the initial design will remain relevant.

8. ACKNOWLEDGMENTS

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-692709.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly-owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This work was part-funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement 671602 (the INTERTWinE project).

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

This material is based upon work supported by the National Science Foundation under Grants Nos. 1562659 and 1229282. 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.

9. REFERENCES

[1] MPI-2 Journal of Development. http://www.mpi-forum.org/docs/mpi-jd/mpi-20-jod.ps.Z.

[2] MPI Standard 3.1. http://www.mpi-forum.org/docs/docs.html.

[3] The OpenMP API Specification for Parallel Programming. http://openmp.org.

[4] Top 500 Supercomputing Sites, Top500. http://www.top500.org/.

[5] T. Adachi, N. Shida, K. Miura, S. Sumimoto, A. Uno, M. Kurokawa, F. Shoji, and M. Yokokawa. The Design of Ultra Scalable MPI Collective Communication on the K Computer. *Computer Science-Research and Development*, 28(2-3):147–155, 2013.

[6] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traff. MPI on Millions of Cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[7] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A Users' Guide to PVM (Parallel Virtual Machine). Technical report, Oak Ridge National Lab., TN (United States), 1991.

[8] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. *Recent Advances in the Message Passing Interface: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, chapter An Evaluation of User-Level Failure Mitigation Support in MPI, pages 193–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] M. Chaarawi and E. Gabriel. Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators. In *Computational Science–ICCS 2008*, pages 297–306. Springer, 2008.

[10] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling Communication Concurrency Through Flexible MPI Endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[11] M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling MPI to Short-Memory MPPs Such As BG/L. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 209–218, 2006.

[12] E. Gabriel, G. E. Fagg, and J. J. Dongarra. Evaluating Dynamic Communicators and One-Sided Operations for Current MPI Libraries. *International Journal of High Performance Computing Applications*, 19(1):67–79, 2005.

[13] G. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[14] W. Gropp and E. Lusk. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 4th European PVM/MPI Users' Group Meeting Cracow, Poland, November 3–5, 1997 Proceedings*, chapter Why are PVM and MPI so different? 1997.

[15] F. Gygi, R. K. Yates, J. Lorenz, E. W. Draeger, F. Franchetti, C. W. Ueberhuber, B. R. d. Supinski,

S. Kral, J. A. Gunnels, and J. C. Sexton. Large-Scale First-Principles Molecular Dynamics Simulations on the BlueGene/L Platform Using the QBOX code. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[16] D. Holmes and S. Booth. Mcmpi: A managed-code mpi library in pure c#. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 25–30, New York, NY, USA, 2013. ACM.

[17] 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*, HPDC '10, 2010.

[18] H. Kamal and A. Wagner. FG-MPI: Fine-grain MPI for Multicore and Clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[19] A. Moody, D. H. Ahn, and B. R. de Supinski. Exascale Algorithms for Generalized MPI_comm_split. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11. 2011.

[20] M. Pérache, P. Carribault, and H. Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759, pages 94–103. Springer, 2009.

[21] B. V. Protopopov and A. Skjellum. A Multithreaded Message Passing Interface (MPI) Architecture: Performance and Program Issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, 2001.

[22] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling Efficient Multithreaded MPI Communication Through a Library-Based Implementation of MPI Endpoints. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 487–498. IEEE, 2014.