LA-UR-19-29588
Approved for public release; distribution is unlimited.

Title: MPI Sessions: Second Demonstration and Evaluation of MPI Sessions Prototype

Author(s): Pritchard, Howard Porter Jr.
Gutierrez, Samuel Keith
Hjelm, Nathan
Holmes, Daniel
Castain, Ralph

Intended for: Report

Issued: 2019-09-24

EXASCALE COMPUTING PROJECT

STPM13/OMPIX/13-35

MPI Sessions:

Second Demonstration and Evaluation of MPI Sessions Prototype

Version 1.0

September, 2019
LA-UR-19-XXYYY

Samuel Gutierrez, Howard Pritchard – LANL
Nathan Hjelm - Google
Daniel Holmes, EPCC
Ralph Castain, Intel

CONTENTS

Page

ABSTRACT

MPI Sessions presents a new paradigm for applications to use MPI functionality within an application. This paradigm offers the promise of more flexible ways for applications to express the capabilities they require from MPI at a fine grain level. This report presents a second evaluation of a prototype implementation of the MPI Sessions proposal, including additional development work on the prototype and investigations using Sessions in the DASK data analytics framework.

1. BACKGROUND

1.1 INTRODUCTION

MPI Sessions addresses a number of the limitations of the current MPI programming model. Among the immediate problems MPI Sessions is intended to address are the following: 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 finalize has been called. With MPI Sessions, an application no longer needs to explicitly call *MPI_Init* to make use of MPI, but rather can use a Session to only initialize MPI resources for specific communication needs. Unless the MPI process explicitly calls *MPI_Init*, there is also no explicit *MPI_COMM_WORLD* communicator. Sessions can be created and destroyed multiple times in an MPI process.

MPI Sessions provides a compartmentalization mechanism, which can be used as a basis both for optimization of MPI behavior for separate components of an application, as well as for supporting run-through-stabilization style MPI fault tolerance schemes. The ability to initialize and finalize MPI multiple times within an MPI process can also assist in the development of fallback-style fault tolerance schemes as well.

The MPI Sessions model is backward compatible, so applications do not need to be extensively modified to start making use of it. In multi-component applications, for example, a component such as a library can make use of MPI Sessions without impacting the rest of the application.

A prototype of the MPI Sessions proposal has been developed based on Open MPI (Hjelm, et al., 2019). A description of the prototype and associated enhancements to the Process Management Interface for Exa-scale (PMIx) was described in a previous ECP report - STPM13-34. An initial evaluation of the prototype was also presented in a previous ECP report – STPM13-36. It turns out there were some mistakes in the data presented in that report. The corrected results are presented in this report. There were also some inaccuracies in the report for the story STPM13-34 concerning the tag matching algorithm. A corrected

description of the algorithm is also presented in this report.

In addition to corrections to data and algorithm descriptions, this report includes a description of the current Sessions Proposal before the forum, which differs in some aspects from the Proposal described in the ECP report for STPM13-4, additional work on the prototype undertaken since completion of STPM13-36, and results of investigations into the usability of Sessions with a variant of the DASK data analytics framework.

The rest of this report is organized as follows: a review of important elements of the current MPI Sessions proposal, a brief review of the prototype implementation focusing on corrections to the description in the report for STPM13-36 and additional work on the prototype since that report, corrected performance results, and an evaluation of the use of the prototype within DASK.

1.2 INTENDED AUDIENCE

This report is written for knowledgeable software professionals and designers. Thus, the Client will not be within the intended audience for this document, which is: (a) Project Team; (b) Project Lead; (c) ECP Auditors and Reviewers.

2. MPI SESSIONS API

2.1 OVERVIEW

In this section, a summary of the MPI Sessions API extensions to the MPI standard is presented.

More detailed descriptions of the MPI Sessions proposed MPI API extensions were given in ECP milestone reports STPM13-4, STPM13-34, and a EuroMPI ’16 paper (Holmes, 2016). These proposed extensions to the MPI standard are a product of several years of work by the MPI Forum’s Sessions Working Group. A formal reading of the MPI Sessions proposal was made at the MPI Forum March 2019. Useful feedback was obtained and will be incorporated into the proposal. A second formal reading is scheduled for the May 2019 meeting.

In the Sessions Proposal and within this report we refer to two different MPI process models. The *World Model* refers to the existing model for initializing and finalizing MPI. In this model, the *MPI_COMM_WORLD* communicator is valid to use once MPI is initialized. Once MPI is finalized, MPI can no longer be used by the application. The *Sessions Model* refers to the new Sessions-based model for using MPI. A process allocates one or more MPI Sessions in order to use MPI methods. Sessions can be initialized and finalized multiple times within an application.

Note in previous reports and earlier versions of the MPI Sessions proposal, the *World Model* was termed the *World Process Model,* and the *Sessions Model* was termed the *Peer to Peer Process Model.*

As stated previously, Sessions are intended as an alternate means for an application, or component of an application, to acquire MPI resources in order to make subsequent use of MPI functionality. The general usage model for an MPI Session is illustrated in Figure 1**Error! Reference source not found.**.

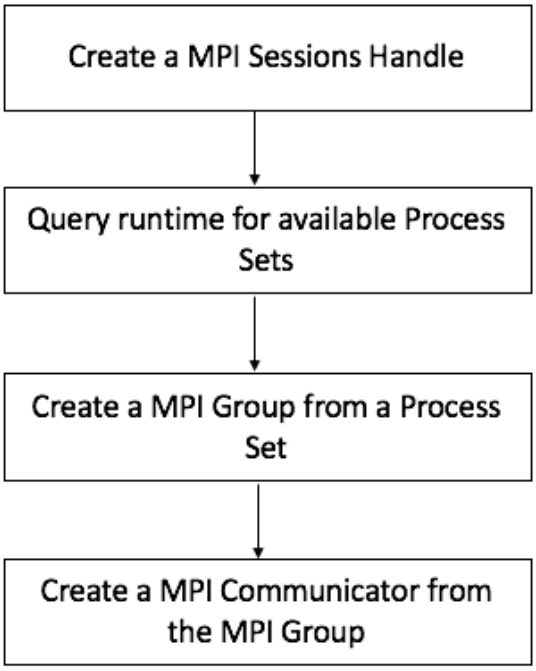

Figure 1. Steps to creating an MPI Communicator from a Session Handle

2.2 PROCESS SETS

A key element of the MPI Sessions proposal is *Process Sets.* These are the mechanism for MPI applications to query the runtime. Each process set has a unique *set name*. In the current scheme, set names have a URI format. Two process sets are mandated:

```
mpi://WORLD
mpi://SELF
```

Many additional process sets may be defined by the runtime, e.g.

```
mpi://MPI_COMM_TYPE_SHARED
mpi://UNIVERSE
location://rack/19
network://leaf-switch/37
arch://x86_64
```

application://redis-server/5

Mechanisms for defining process sets, and how system resources are assigned to these sets is currently assumed to be runtime implementation dependent. The prototype implementation, for example, includes an mpi://shared process set describing the MPI processes in a job that are local to the same node in a cluster. It was further modified to support an additional process set to use with DASK (see Section 4.2).

A process set caches key/value tuples which an application can access using *MPI_Session_get_pset_info*, and subsequent queries of the returned *info* object using existing MPI info object methods. A process may query for more information about process sets, e.g. the number of processes in the set, etc. using this method. The *size* key is mandatory for all process sets.

2.3 SESSION CREATION/DESTRUCTION FUNCTIONS

```
MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler, MPI_Session *session)
  IN        info       info object to specify thread level support, MPI implementation
                       Specific resources, etc.
  IN        errhandler specifies an error handler to invoke in the event that Session
                       initialization in the event that an error is encountered during
                       Session instantiation.
  OUT       session    handle to the created session.
```

The *info* argument can be used for specifying the level of thread safety required for the Session, and possibly other MPI implementation specific resource and functionality requirements. The *errhandler* argument specifies an error handler to associate with the Session. Session initialization is intended to be a local, lightweight operation. A single process may initialize multiple Sessions. *MPI_Session_init* is always thread safe; multiple threads within an application may invoke it concurrently.

```
MPI_Session_finalize(MPI_Session *session)
  IN        session    handle to previously created session
```

This function is the Session equivalent of *MPI_Finalize.* It can block waiting for destruction of objects derived from the Session handle. Every initialized Session must be finalized using *MPI_session_finalize.*

2.4 RUNTIME QUERY FUNCTIONS

```
MPI_Session_get_num_psets(MPI_Session session,
                          int *npset_names)
```

IN	session	handled to previously created session.
OUT	npset_names	number of available process sets

This function is used to query the runtime for the number of available process sets in which the calling MPI process is a member. The number of available process sets returned by this function may increase with subsequent calls to *MPI_Session_get_num_psets*.

```
MPI_Session_get_nth_psetlen(MPI_Session session,
                            int n,
                            int *pset_len)
```

IN	session	handled to previously created session.
IN	n	process set name number (integer)
OUT	pset_len	length of the nth process set name

This function retrieves the length of the name of the nth process set name. Valid values for n range from 0 to one minus the number of available process sets for this Session. The number of available process sets for this session can be determined by calling *MPI_Session_get_num_psets*. The length returned in C includes space for the end-of-string character.

```
MPI_Session_get_nth_pset(MPI_Session session,
                         int n,
                         int pset_len,
                         char *pset_name)
```

IN	session	handled to previously created session.
IN	n	process set name number (integer)
IN	pset_len	length of the pset_name argument
OUT	pset_name	pset_name (string)

This function returns the name of the nth process set in the supplied pset_name buffer. pset_len is the number of characters available in pset_name. If it is less than the actual size of the process set name, the value returned in pset_name is truncated. In C, pset_len should be one less than the amount of allocated space to allow for the null terminator.

```
MPI_Session_get_info(MPI_Session session,
                     MPI_Info *info)
```

IN	session	handled to previously created session.
OUT	info	info object containing information about the given process set.

MPI_Session_get_info returns a new info object containing the hints of the MPI Session associated with session. The current setting of all hints related to this MPI Session is returned in info. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by

the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info via MPI_INFO_FREE.

MPI_Session_get_pset_info(MPI_Session session, const char *set_name, MPI_Info *info)		
IN	session	handled to previously created session.
IN	set_name	name of process set to query
OUT	info	info object containing information about the given process set.

This function is used to query properties of a specific process set. The returned info object can in turn be queried with existing MPI info object query functions. One key/value pair must be defined, size. The value of the *size* key specifies the number of MPI processes in the process set. The user is responsible for freeing the returned MPI_Info object.

2.5 GROUP AND COMMUNICATOR MANAGEMENT FUNCTIONS

MPI_Group_create_from_session_pset(MPI_Session session, const char *set_name, MPI_Group *group)		
IN	session	handled to previously created session.
IN	set_name	name of process set from which to create an MPI Group
OUT	group	MPI_Group handle

The function *MPI_Group_Create_from_session_pset* creates a group newgroup using the provided session handle and process set. The process set name must be one returned from an invocation of MPI_SESSION_GET_PSET_NAME using the supplied session handle. If the pset_name does not exist, MPI_GROUP_NULL will be returned in the newgroup argument. As with other group constructors, *MPI_Group_Create_from_session_pset* is a local function.

MPI_Comm_create_from_group(MPI_Group group, const char *stringtag, MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)		
IN	group	MPI group handle
IN	stringtag	character string which uniquely defines invocation of this communicator constructor using the supplied group
IN	info	info object
IN	errhandler	error handler to be attached to new intra-communicator
OUT	comm	new communicator (handle)

This function is used to create a MPI communicator from a MPI group. The *stringtag* argument allows the MPI implementation to discriminate between potentially concurrent calls by the application to create multiple MPI communicators using the same supplied group.

3. SESSION PROTOTYPE IMPLEMENTATION

3.1 PROTOTYPE DESCRIPTION

A detailed description of a MPI Sessions prototype was presented in ECP Milestone report STPM 13-34 and in (Hjelm, et al., 2019). This section provides a brief description of the prototype to provide context for later sections of the report and to clarify some elements of the prototype originally described in STPM 13-34.

The MPI Sessions prototype is based off of the *master* branch of Open MPI from the Project's GitHub repo. The prototype also relies on recent additions to PMIx (Castain, 2018). The prototype is available on a fork of the Open MPI project.

In implementing the prototype, it turned out that it was easier to implement the Sessions extensions to the standard in their entirety rather than implementing only a subset of the features identified in the previous STPM13/OMPIX/13-4 report to support QUO.

The prototype involved five major groups of modifications and additions to Open MPI:

- development and implementation of Open MPI's communicator identifier (CID) generator to support 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, error handling, and Sessions attribute functions prior to invocation of *MPI_Session_init*,
- restructuring of MPI resource teardown to support Sessions ability to be initialized and finalized multiple times within a single application execution instance,
- implementation of the Sessions API extensions interfaces,
- and additions to the PMIx component of Open MPI's internal PMIx framework to allow for the use of new PMIx group methods.

Only the first three groups of modifications are discussed in this report, as the other major modifications are covered adequately in the ECP report for STPM 13-34.

3.1.1 Communicator Identifiers in the baseline 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 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 (Gabriel & al, 2004). 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 Sessions prototype.

3.1.2 Algorithmic changes to CID generation to support Sessions

One of the most significant challenges in implementing the 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 CID 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 Sessions proposal provides two new communicator constructor functions – *MPI_Comm_create_from_group* and *MPI_Intercomm_create_from_groups.* 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 Model. For the prototype implementation, we chose to support these constructor functions by using the runtime group constructor support provide by PMIx, see (Castain, 2018).

One of the biggest challenges in implementing the prototype was developing a method for generating unique *Communicator Identifiers* (CIDs) when a communicator is created. Since the *Sessions Model* does not have the concept of a base MPI_COMM_WORLD, and hence of an implied base CID, a new method for generating CIDs is required.

The PMIx group constructor returns a 64-bit *PMIx Group Context Identifier* (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 inter-node 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 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 communicators. The second field is divided into eight 8-bit subfields. The subfields are used to generate exCID 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 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.

3.1.3 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 exCID. 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 exCID 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 exCID for all available PML components.

3.1.4 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 object management, 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 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, MPI_Init_thread,* and *MPI_Finalize* were restructured to create and finalize an internal MPI Session that also initializes the World 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 Model alongside the World Model.

3.1.5 Implementation of MPI Sessions Interfaces

The prototype implements the complete set of C interfaces that are defined in the 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 Open MPI's Multicomponent Architecture (MCA) support framework, info subsystem, point-to-point support, etc. The implementation of *MPI_Session_finalize* releases all resources associated with the Sessions object and tears down any resources not still in use by another MPI object.

The 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 to the process set in the World Model communicator MPI_COMM_WORLD; *mpi://self* (MPI_COMM_SELF); an *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.

3.2 FOLLOW-ON WORK ON THE PROTOTYPE

In the course of investigating the use of a DOE NNSA application with Sessions (described

in the report for STPM13-34), some issues were found in the prototype. The mapping of Communicator handles between C and Fortran was not implemented correctly in the prototype and had to be fixed to get the mixed C/Fortran application to work.

In addition, some issues were found with resource teardown in the implementation of *MPI_Session_finalize.* This problem actually uncovered a potential issue for implementing this function over certain types of interconnects. Since a session can, in principle, span only a subset of the entire MPI processes in the job, the usual solution of having a global barrier as part of the MPI finalization procedure does not work. The shutdown methods(s) for the communication mechanisms being used for message exchange between the MPI processes participating in the Session should ideally handle the notion of an internal connection shutdown. This was not the case for the OB1 Vader byte transport layer (BTL), so it needed to be modified slightly in order to properly handle the Session finalize method. This issue may need to be addressed by more sophisticated algorithms for a production implementation of Sessions.

The Fortran bindings (F77, F90, and F08) for the Sessions extensions to the MPI API were also added to the prototype since the STPM 13-36 story.

3.3 REFERENCE IMPLEMENTATION SOURCE AND TESTS

The prototype is available for download at https://github.com/hpc/ompi/tree/sessions_new. The code examples from the Sessions Proposals plus a more extensive test case are available at https://github.com/hppritcha/mpi_sessions_tests. Instructions on how to build and run the test cases are included in the repo's README. Note the special instructions for running the tests on Cray XC systems.

4. EVALUATION OF PROTOTYPE

4.1 UPDATE ON PERFORMANCE BENCHMARK RESULTS

In the previous STPM13-36 report, differences in the OSU MBW MR message rate results were observed when using World Model verses the Sessions Model for creating communicators to be used for message exchange. At the time of that writing, the difference in performance was attributed to the use of the exCID tag matching method rather than the optimized CID for some of the messages. Subsequent analysis revealed however that the performance difference could be attributed to the fact that for the Sessions Model, the MPI OB1 layer was being initialized to support MPI_THREAD_MULTIPLE rather than MPI_THREAD_SINGLE.

Figure 2 compares the latency for short messages using the *osu_latency* benchmark. With the correction for Sessions initialization, the difference in short message latency performance is virtually identical whether using the World or Sessions model. Similarly, Figure 3 shows the results of the *osu_mbw_mr* message rate test using two MPI processes. As with the latency measurements, the difference in message rate performance between the World and Sessions initialization approaches is minimal.

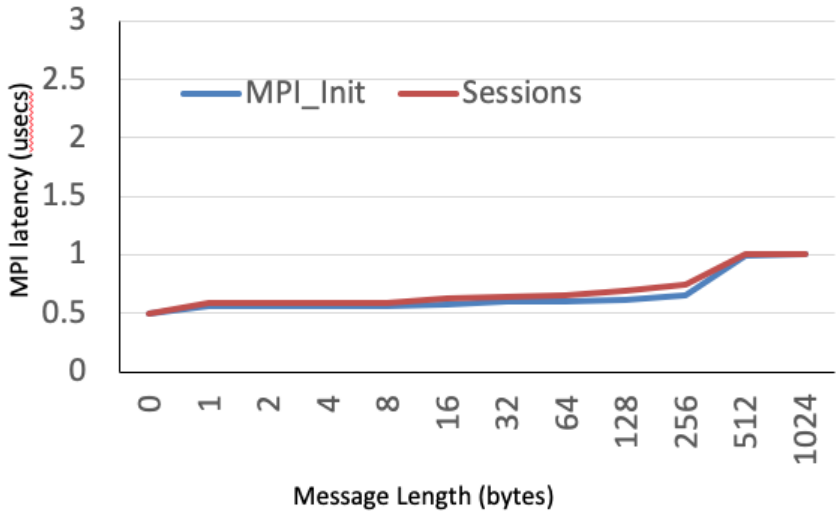

Figure 2. Comparison of MPI Latency using MPI_Init and MPI_Session_init.

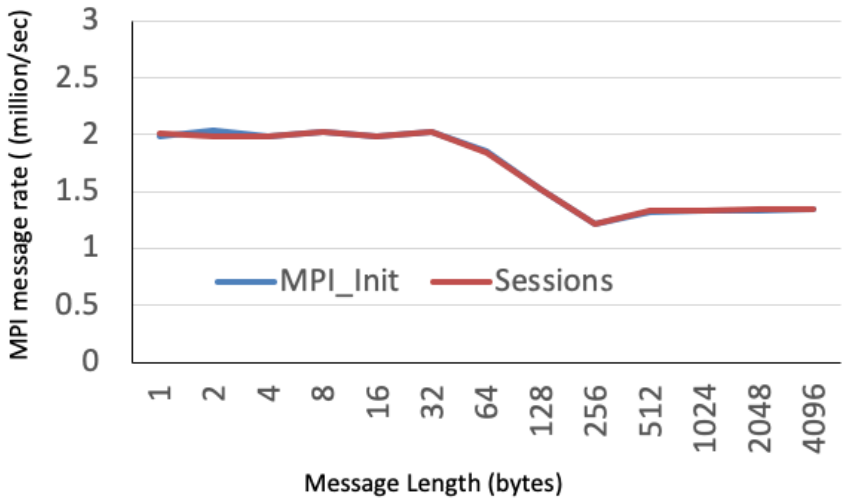

Figure 3. Comparison of MPI message rate using MPI_Init and MPI_Session_init.

4.2 EVALUATION OF MPI SESSIONS USING DASK

In the previous report, the Sessions prototype was evaluated for use in a fairly typical bulk-synchronous MPI application. The use of OpenMP in portions of this multi-physics application, and the different thread support levels required by the different modules of the application made it of interest for testing the Sessions prototype. Some modification of the application was required to use Sessions. Except for some bugs revealed testing the application with the prototype, no changes to the prototype were required.

For a second application to evaluate using the Sessions prototype, an application or framework less typical of HPC applications was desired. One of the goals of the original Sessions proposal was to expand the range of applicability of the MPI program model. Looking beyond the typical HPC application space would be more likely to reveal possible shortcomings in the Sessions proposal, as well as limitations owing to the way the Sessions prototype was implemented.

4.2.1 DASK overview

DASK is a task-based Python parallelization framework. The framework supports a scheduler API which an application writer can use to distribute tasks (typically Python functions) across a set of worker processes. DASK provides mechanisms for expressing task dependencies, allowing one to parallelize a general DAG workflow.

There a number of DASK projects for managing the backing framework. In this investigation, we focused on an MPI based version of DASK – DASK MPI as a starting point. This version of DASK uses MPI purely for launching the DASK daemons – the scheduler, the daemon to handle client applications, and the worker daemons. It uses mpi4py to initialize and finalize MPI. The DASK native messaging framework is used for any IPC between the scheduler, client, and worker processes.0

One of the main motivations for investigating the usefulness of the Sessions model within the DASK framework is the interest in running tasks within the framework which make use of MPI to accelerate tasks which, if run sequentially within a DAG could lead to a bottleneck in a workflow as problem sizes are scaled up, but which can be efficiently parallelized using MPI. Note DASK does support an IPC mechanism, but its performance is generally poor compared to that delivered by MPI implementations, especially on high performance networks typical of HPC systems. Ideally the DASK framework would not need to directly know about the fact that a set of tasks were us

As an initial first step in exploring the use of MPI tasks within DASK, we attempted to run a simple MPI task within the DASK-MPI framework. The task uses MPI Sessions to create an MPI Communicator. Figure 4 illustrates how DASK-MPI maps components of a DASK job to MPI ranks.

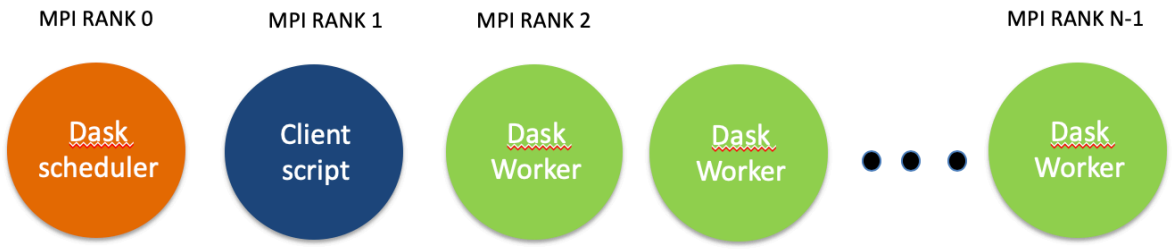

Figure 4. Mapping of MPI Ranks to components of a DASK-MPI batch job. Rank 0 runs the DASK scheduler, Rank 1 runs the client Python script, and the tasks submitted by the client

script to the DASK scheduler are run on the DASK Worker processes. The Worker processes are scheduled on the remaining MPI ranks in the job.

Although DASK-MPI was used as a starting point for evaluating the use of Sessions for more non-traditional HPC applications, we would expect it to be usable within the DASK framework as well. This would require extensions to DASK to know how to launch a task set using the PRTE launcher.

4.2.2 Challenges Using DASK with the Sessions Prototype

The evaluation of the Sessions prototype using DASK-MPI revealed a significant limitation in the prototype. Namely, there is no way currently to specify arbitrary process sets within a given MPI job. For the demo, the prototype was enhanced to pre-define a *dask:://world* process set which spans MPI ranks 2 through N-1, for an N-way MPI job. The prototype does support querying the PMIx runtime for process sets, but PMIx currently does not offer support for defining additional process sets. Were DASK-MPI to be enhanced to allow for the use of MPI tasks which employ the Sessions approach to MPI initialization, it would be necessary to enhance PMIx to be able to define additional process sets. This would probably need to be fairly dynamic. DASK-MPI would need to define client-side extensions to the underlying DASK client API to allow for specification of at least the size of an MPI accelerated task, as well as hooks into PMIx to create the necessary process sets to support these tasks.

Future work will involve enhancing PMIx to provide mechanisms for defining process sets in a dynamic way, and providing hooks to this functionality for DASK-MPI such that it can schedule MPI accelerated functions within the context of a general DAG-defined workflow.

5. CONCLUSIONS

This follow-on evaluation of the prototype implementation of the MPI Sessions API shows that in the area of functionality, it is already able to demonstrate some of the important aspects of MPI Sessions. Namely, tasks running within a framework such as DASK can initialize MPI communicators without needing for the framework to be aware of the use of MPI within the tasks. However, the evaluation also showed that the prototype and the supporting PMIx infrastructure currently lack functionality required for the proto-type to be of general use within DASK, and likely similar frameworks for supporting parallelization of DAG based workflows.

6. WORKS CITED

Bernholdt, D., Boehm, S., Bosilca, G., Venkata, M., Grant, R., Naughton, T., . . . Vallee, G. (2017, June 14). *A Survey of MPI Usage in the U. S. Exascale Computing Project.* Oak Ridge National Lab: Exascale Initiative, U.S. DOE. Retrieved June 14, 2018, from https://www.exascaleproject.org/

Castain, R. H. (2018). PIMx: Process Management for Exascale Environments. *Parallel Computing, 79*, 9-29.

Gabriel, E., & al, e. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *11th European PVM/MPI User's Group.* Budapest.

Goglin, B. (2018, 1 1). *HWLOC*. Retrieved 6 24, 2018, from HWLOC: https://github.com/open-mpi/hwloc

Hjelm, N., Pritchard, H., Gutierrez, S., Holmes, D., Castain, R., & Skjellum, A. (2019). MPI Sessions: Evaluation of an Implementation in Open MPI. *2019 IEEE International Conference on Cluster Computing.*

Holmes, D. M. (2016). MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. *EuroMPI*, 121-129.