Title: MPI Session:External Network Transport Implementation

Author(s): Pritchard, Howard Porter Jr.
Herschberg, Tom

Intended for: Report

Issued: 2020-09-29

STPR17/OMPIX/17-37

MPI Sessions:

External Network Transport Implementation

Version 1.0

September, 2020

LA-UR-19-YYZZZZ

Howard Pritchard – LANL
Tom Herschberg – Univ. Tennessee Chattanooga

CONTENTS

Page

ABSTRACT

The MPI Sessions extensions to the MPI standard have been accepted by the MPI Forum and will be included in the upcoming MPI 4 version of the standard. 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.

A prototype of MPI Sessions, based on the Open MPI implementation of the MPI standard, was developed to facilitate acceptance of the Sessions proposal by the Forum. The initial implementation had some limitations, one of the more significant ones being that it was limited in its ability to fully exploit modern network APIs such as OFI libfabric and OpenUCX and underlying network hardware. This report presents enhancements to the prototype implementation of MPI Sessions that removes this restriction for the networks to be used in the next generation of DOE exa–scale systems. Open MPI was used as the implementation vehicle, but results here are also relevant to other middleware stacks.

1. BACKGROUND

1.1 INTRODUCTION

The MPI Sessions proposal specifies well-defined extensions to the MPI Standard, and has recently been accepted for the MPI 4.0 Standard specification. A prototype has been developed to evaluate the practicality of implementing Sessions functionality, the potential impact on basic MPI performance characteristics, as well as the usability for existing, large-scale MPI applications (Hjelm, et al., 2019). This work was also reported in several previous ECP-reports: STPM13-34 , STPM13-35, and STPM13-36.

The proposed MPI Sessions extensions to the MPI API have been previously published (Holmes, 2016). 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 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. The returned 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. An MPI implementation must support at least two process sets: *mpi://world* and *mpi://self*. A process may query for additional process sets using the *MPI_Session_get_num_psets* and *MPI_Session_get_nth_pset* functions. 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.

Sessions can be used in applications making use of *MPI_Init* to initialize MPI. This allows for gradual introduction of Sessions use into existing applications still using *MPI_Init* or *MPI_Init_thread* and *MPI_Finalize* (termed the World Process Model in the Sessions additions to the MPI Standard).

The remainder of this paper is organized as follows: Section 2 offers background and motivations for this work. Section 3 describes our enhancements to the prototype to better leverage networks to be deployed as part of next generation DOE exa–scale systems. Section 4 evaluates the prototype and our findings regarding MPI Sessions; in particular, this section covers evaluation criteria, experimental setup and benchmark results. We offer conclusions and outline future work in Section 5.

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. BACKGROUND AND MOTIVATION

The MPI Sessions prototype was initially designed to use the most general-purpose component of the point-to-point messaging framework (PML): OB1. This PML was chosen because message handling (including tag matching) is done entirely within Open MPI, making it relatively easy to modify to support MPI Sessions. Although the OB1 PML component can leverage RDMA capable networks, it is not designed to take advantage of any tag matching offload capability provided by lower levels of the network stack, including possible hardware-based tag matching features in the network interface. Hardware-based tag matching allows for moving the processing of MPI messages off of the host processor and onto the network hardware. The potential benefits of offloading MPI message processing from host processors being used by MPI applications is well known (Marts, 2019), (Derradji, 2015). The CM PML, which is intended to leverage networks featuring tag matching offload, has been shown to consistently outperform OB1 in terms of latency, bandwidth, and message rates (Graham, 2007).

The CM PML makes use of the message transport layer (MTL) framework to interface to particular network stacks. Note only one MTL component can be active within a given instance of an MPI application. There are several MTL components to consider when choosing one to modify for use with MPI Sessions. Of them, the OFI MTL, which utilizes the OpenFabric Interfaces (OFI) libfabric (Grun, 2015), is the most promising. Libfabric is hardware-agnostic and compatible with several popular high-performance fabrics and networking hardware, which allows for more straightforward cooperation between applications and network hardware. As such, libfabric is currently the network interface of choice for systems designed for exascale. Libfabric will be the network API used in exascale-era systems such as Aurora at Argonne and Frontier at Oak Ridge, both of which utilize HPE's Slingshot Interconnect (Sensi, 2020). Thus, extending Open MPI's implementation of MPI Sessions to leverage the OFI MTL will lead to the greatest potential for its use on these DOE exa-scale systems.

(Marts, 2019)

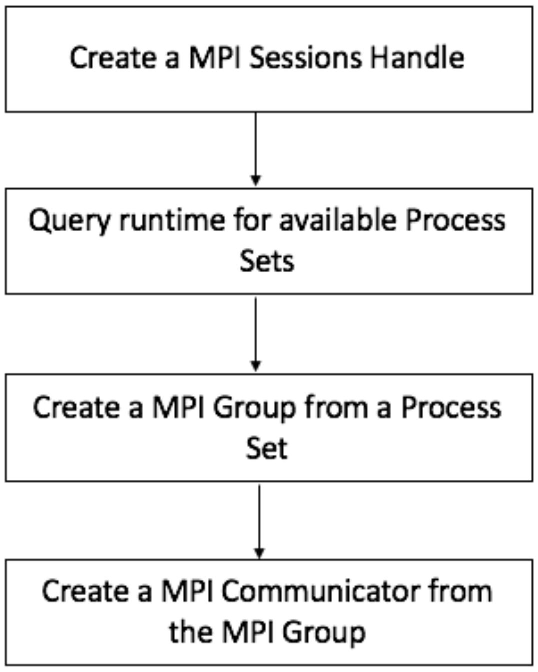
Create a MPI Sessions Handle
Query runtime for available Process Sets
Create a MPI Group from a Process Set
Create a MPI Communicator from the MPI Group

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

3. SESSION PROTOTYPE ENHANCEMENTS

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

- PMIx (Castain R. H., 2017), (Castain R. H., 2018), a reference implementation of the Process Management Interface for Exascale specification;
- PRRTE (Castain R. H., 2020), the PMIx reference runtime environment; and
- enhancements to Open MPI, an open-source MPI implementation (Gabriel E. a., 2004).

Enhancements to the first two components have been described in a previous paper (Hjelm, et al., 2019). This paper also described enhancements to Open MPI's OB1 PML to support MPI Sessions.

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 (CID) generator to support the creation of MPI communicators not derived from MPI_COMM_WORLD,
- update of the OB1 PML 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 report we focus on additional changes to Open MPI to support MPI Sessions with the OFI MTL.

While the OFI MTL has the benefit of offloading message tag matching to the network hardware, it comes with the restriction that the tags used for matching messages must be 64 bits or smaller. Thus, in order for the OFI MTL to work with the 128-bit exCID, some modifications to the MTL's message delivery methods were required.

If a process only has the exCID of the peer that it wants to send a message to, it must initiate an exchange of local CIDs that can be used for tag matching within OFI Libfabric. To do this, the sender creates an untagged control message containing:

- the exCID of the communicator that the sender is trying to send the message on,
- the rank of the sender in that communicator, and
- the sender's local CID.

The sender then posts a receive buffer and waits for a response from the receiver. Concurrently, the receiver posts a receive buffer in anticipation of receiving an untagged control message from the sender. Once the control message has been received, the receiver saves the sender's local CID to an array of local CIDs associated with the current communicator, using the sender's rank as the index into this array. The receiver then sends its own untagged control message containing its local CID and rank back to the sender. Once this response message is received by the sender, the sender saves the receiver's local CID into its own array, and the two processes can begin communicating normally using the local CIDs that were just exchanged.

Implementing this algorithm required modifications at both the MTL and PML levels. On the MTL level, several functions were added to facilitate the sending, receiving, and processing of untagged messages. Sending an untagged message is relatively straightforward because the destination rank is passed down from the send operation in the application. Receiving, on the other hand, is more difficult because MPI_ANY_SOURCE can be used instead of specifying a specific source rank. For this reason, all processes keep a receive buffer posted to receive untagged control messages. This addition guarantees progress and prevents deadlocks that might have occurred in communication patterns where all processes perform blocking send operations before posting receive buffers.

The modified prototype was tested with the PSM2, sockets, ofi rxm;verbs, and GNI providers. This turned out to be important as different providers tend to behave differently, particularly for providers sensitive to the libfabric endpoint capabilities requested. Because the original sending and receiving algorithms are left intact, the modified PML and MTL avoid use of this local CID exchange mechanism when using the World Process Model and global CIDs.

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 THE EXTENDED PROTOTYPE

In this section, we evaluate the performance of the modified OFI MTL in the MPI Sessions prototype using microbenchmarks. Our results show that the modifications introduced to support MPI Sessions functionality do not impose a performance penalty over our baseline for MPI startup and MPI communicator construction. For the latter, the prototype shows improved performance for MPI communicator construction. However, we do observe a performance impact on latency and message throughput for communicators created via Sessions when using some libfabric providers.

4.1 EVALUATION CRITERIA

Several of the changes made in the MPI Sessions prototype could potentially affect Open MPI's performance. Large modifications to the constructor and destructor methods for multiple Open MPI subsystems could impact MPI initialization. Likewise, the changes made to MPI Communicator construction to support *MPI_Comm_create_from_group* could add overhead to creating MPI Communicators. The exCID-based tag-matching process outlined in (Hjelm, et al., 2019) also has a possibility of adding overhead, even though the exchange of local CIDs only has to occur once. Additionally, the process of generating MPI Communicators using PMIx could also incur a performance penalty.

Table 1. Hardware and software used for this evaluation.

	Grizzly	*Oneseventeen*
Model	Penguin Tundra ES	Dell R730
OS	Redhat 7.8	Redhat 7.2
CPU	2x18-core Intel E5-2695v4 @ 2.10 GHz	2x12-core Intel E5-2650v4 @ 2.20 Ghz
RAM	128 GB	128 GB
Network	Intel Omni-Path 100	EDR Infiniband
Compiler	GCC 9.3	GCC 10.2
Resource Manager	SLURM 2.0.3	OGS Grid Engin 2011.11p1

4.2 EXPERIMENTAL SETUP

Performance results were gathered from the systems detailed in Table 1. Data were collected during regular operating hours, so the systems were servicing other workloads alongside but in isolation from our performance evaluation runs.

4.3 MPI BENCHMARK RESULTS

The results reported in this section were obtained using the Sessions prototype (Hjelm N. P., n.d.). For the baseline Open MPI, the master branch at Git SHA c17968c7 was used. Libfabric 1.10.1 was used for all runs unless otherwise noted. PSM2 version 11.2.78 was used on the Grizzly cluster.

4.3.1 MPI Startup Overhead

MPI initialization times using *MPI_Init* were measured with the OSU osu_init benchmark (The Ohio State University MVAPICH Benchmarks, 2019). 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_session_pset*, and *MPI_Comm_create_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 (https://github.com/hppritcha/osu-microbenchmarks-sessions, 2019).

Figure 2 presents the timing data using both approaches obtained on the Grizzly cluster using the PSM2 OFI libfabric provider. Results for the case of one MPI process per node and 32 MPI processes per node is shown. The performance difference between using the MPI Sessions approach verses *MPI_Init* to initialize MPI shows some variation, especially as the number of MPI processes per node increases. The main difference between the two approaches to initialization MPI is how the two paths use PMIx for synchronization. In the case of *MPI_Init*, the *PMIx_fence* method is used, while for MPI Sessions the *PMIx_Group_construct* method is used.

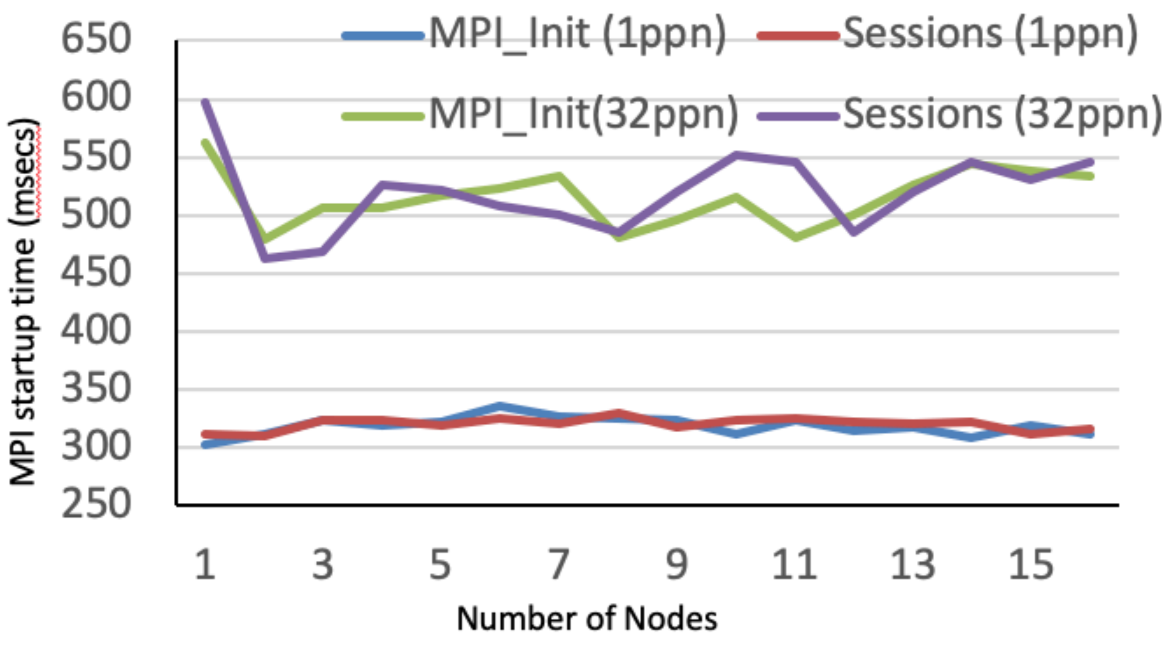

Figure 2. MPI Initialization time using MPI_Init and MPI Sessions methods. Results obtained on the Grizzly cluster using the PSM2 OFI provider.

4.3.2 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 3 compares the time for the communicator duplication operation when using the two approaches to MPI initialization when run on the Grizzly cluster using the PSM2 libfabric provider. Note the times reported are per iteration, not the time reported in the benchmark output. The data indicate that on this platform, the exCID algorithm out performs the existing CID consensus algorithm. As described in (Hjelm, et al., 2019), the new algorithm avoids the need to use a sequence of *MPI_Allreduce* operations to most of the duplicates of an existing communicator. Figure 4 shows the results obtained using the oneseventeen cluster. Note problems using the Open Grid scheduler with Open MPI prevented running the benchmark across nodes. Even on a single node, there is considerable scatter in timing data using the baseline Open MPI.

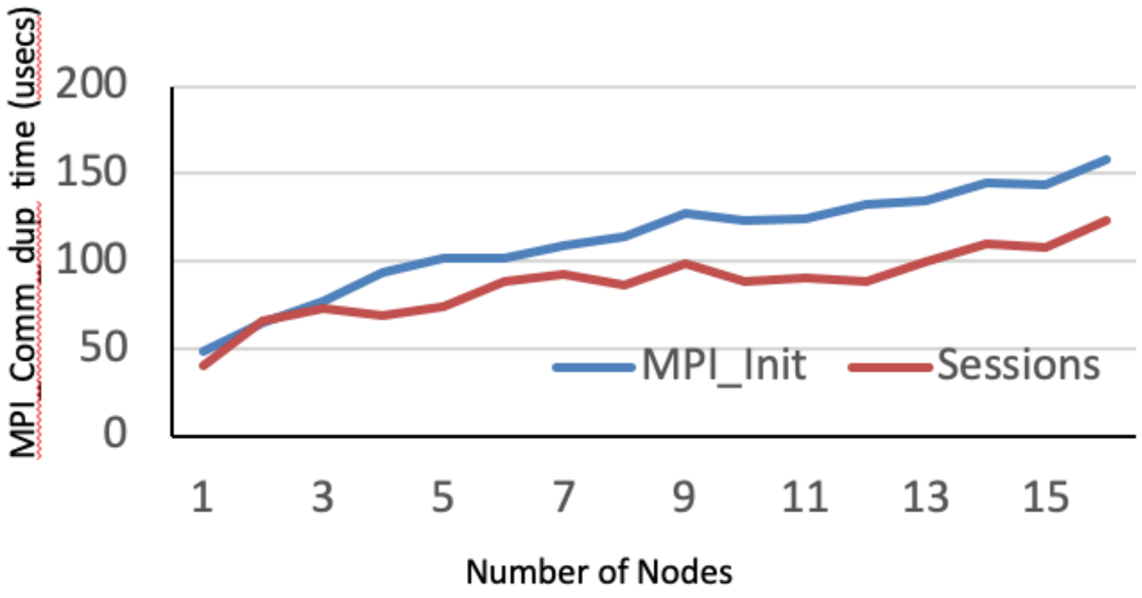

Figure 3. MPI_Comm_dup overhead obtained on the Grizzly cluster using the PSM2 OFI provider and 32ppn.

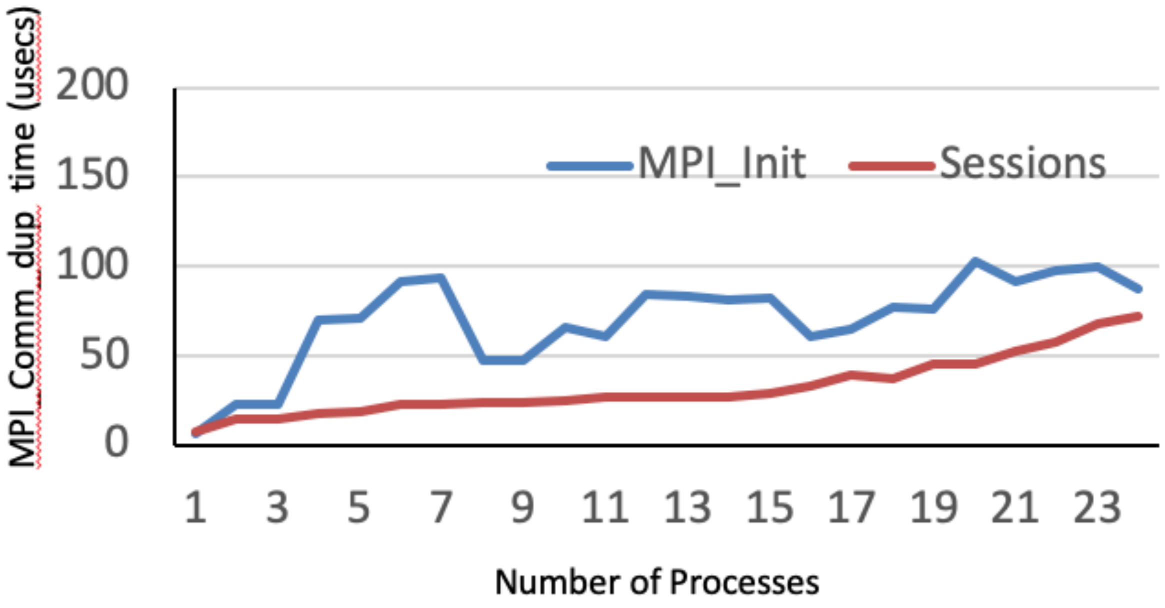

Figure 4. MPI_Comm_dup overhead obtained on the oneseventeen cluster using the RXM/verbs OFI provider.

4.3.3 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 single nodes of the two systems (Table 1), 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. Note not all OFI libfabric providers make use of shared memory for intranode data exchange.

Figure 5 presents relative MPI latency and Figure 6 presenting message throughput when using *MPI_Init* and *MPI_Session_init* to initialize MPI. As discussed in (Hjelm, et al., 2019), the use of exCIDs and local CIDs could have a performance impact on the handling of MPI messages at both the sender and receiver. For the PSM2 provider, which does make use of shared memory be default for intra-node messages, the introduction of the exCID code does impact short message latency and hence message throughput rate. On the Grizzly cluster, an 8-byte message latency of a little under 400 nanoseconds is obtained using the OFI MTL with the PSM2 provider. Measurements indicate that the exCID code adds about 50-100 nanoseconds into the code path, even after the local CID information has been exchanged. An 8-byte message throughput rate of 3.1×10^6 messages/second is measured using the baseline Open MPI, while the corresponding rate measured with the prototype is 2.8×10^6. In contrast, for the OFI RXM/verbs provider, which does not have an optimized, shared memory path for intra-node messages, the overhead introduced by the exCID path is negligible.

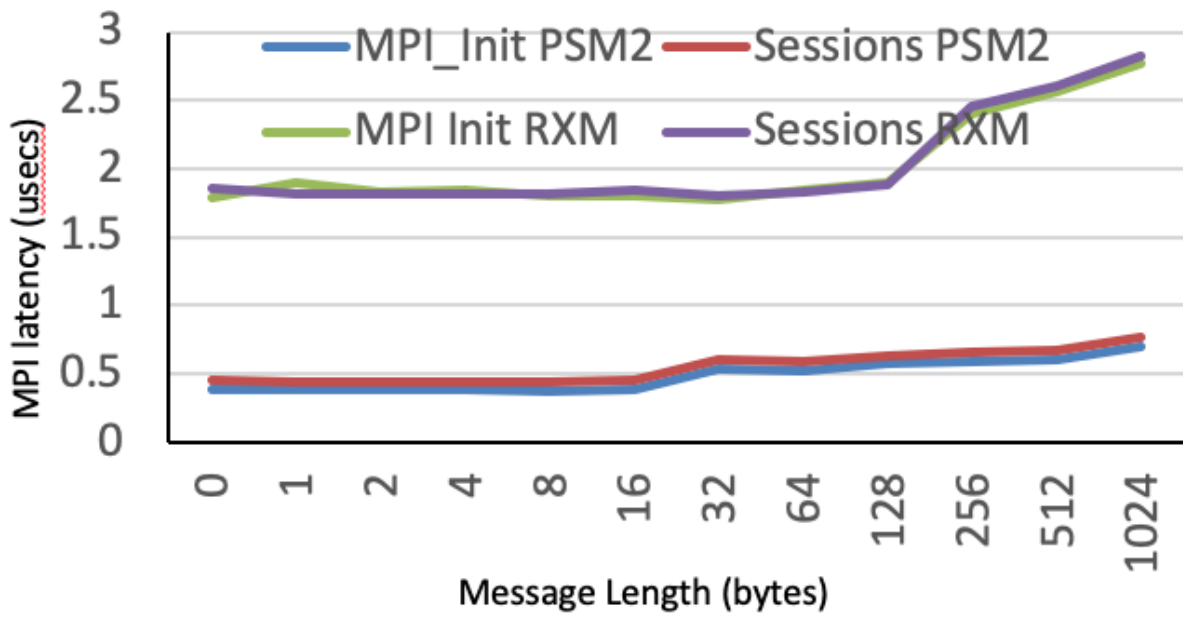

Figure 5. Comparison of MPI Latency using MPI_Init and MPI_Session_init using the PSM2 and rxm/verbs OFI providers.

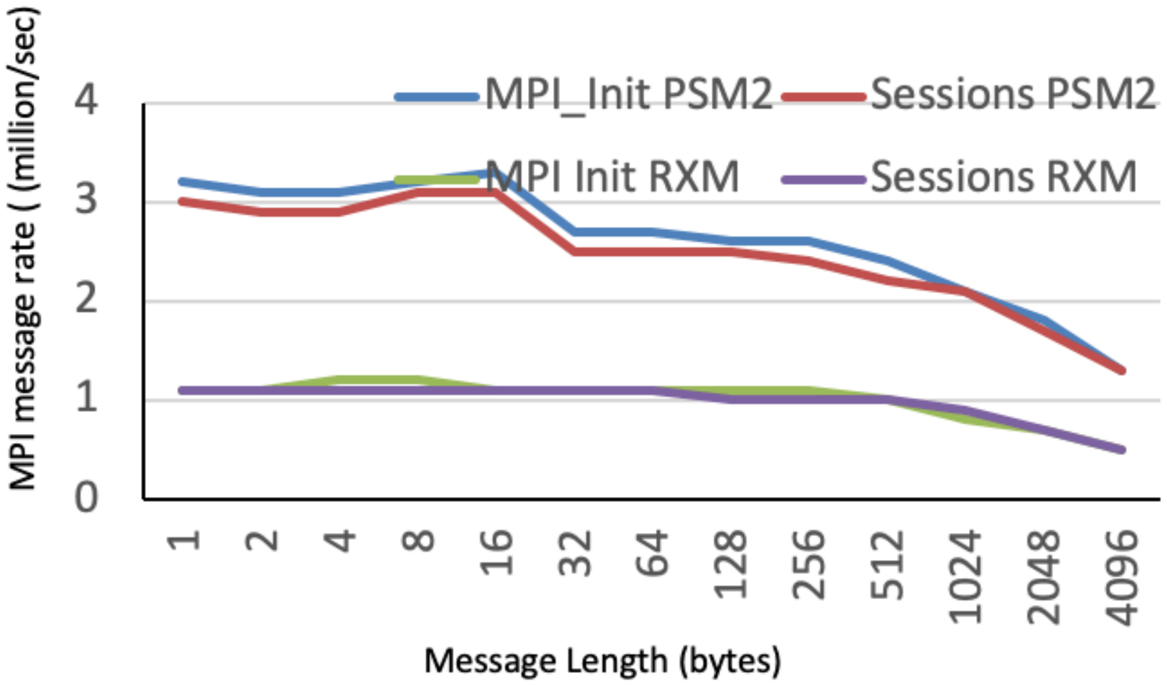

Figure 6. Comparison of MPI message rate using MPI_Init and MPI_Session_init using the PSM2 and rxm/verbs OFI providers.

4.3.4 HPC Challenge

The High Performance Computing Challenge (HPCC) benchmark (Juszczek, 2005) 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.

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.

Figure 7 and Figure 8 present MPI 8-byte latencies for the random and natural order rings, respectively. The results reported for the modified HPC challenge uses MPI Sessions for the bandwidth and latency component of the benchmark. The baseline Open MPI was used with the unmodified application. As expected from the osu_latency results, the Sessions prototype yields somewhat higher latencies (20-30%), particularly at higher node counts. The very peculiar behavior observed using both the OFI MTL with an underlying PSM2 provider and the PSM2 MTL directly is not exhibited by the prototype. A cross check of the baseline Open MPI using the GNI libfabric provider on a different cluster also does not show this unusual behavior.

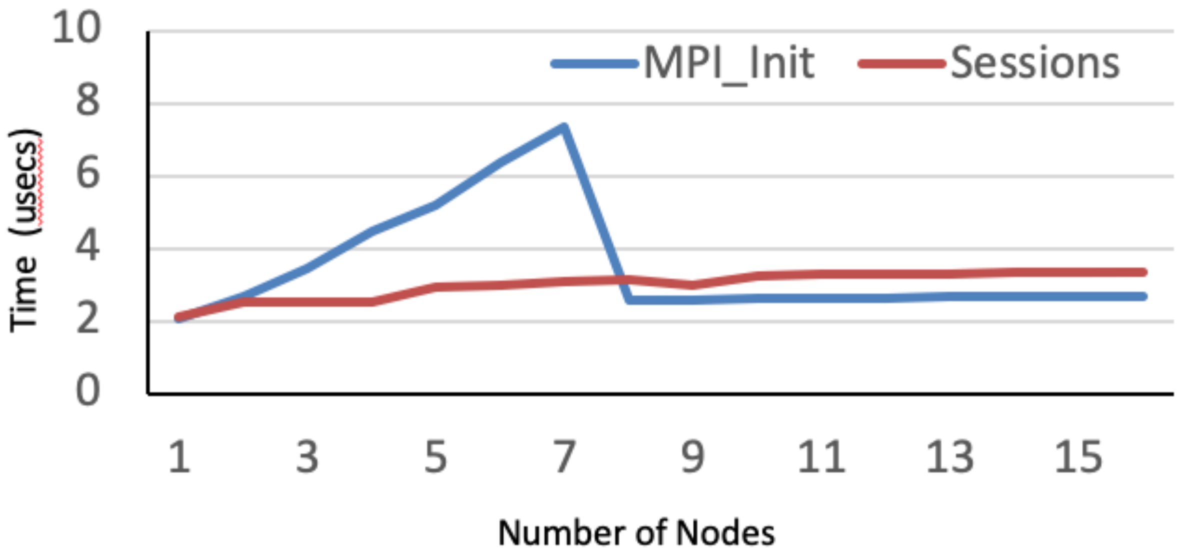

Figure 7. HPCC random order ring 8-byte latency results obtained on the Grizzly cluster using the PSM2 OFI libfabric provider and 32 ppn.

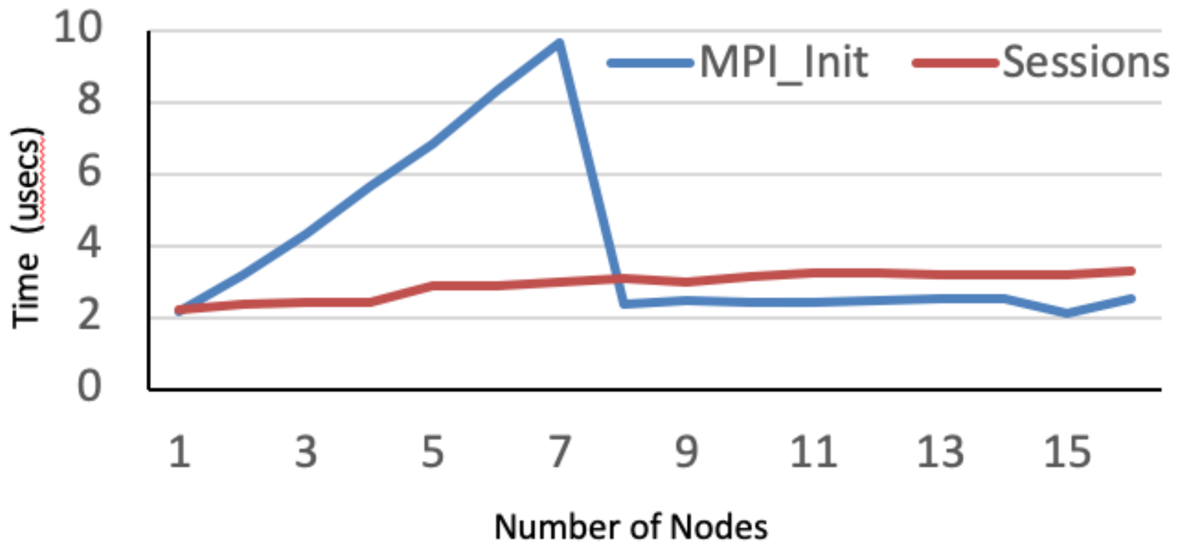

Figure 8. HPCC natural order ring 8-byte latency results obtained on the Grizzly cluster using the PSM2 OFI libfabric provider and 32 ppn.

5. 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 provides the similar or better performance for MPI initialization and communicator construction than the baseline, but in contrast to results found using the OB1 PML, there is some impact on MPI latency or message throughput performance when using optimized OFI libfabric providers. The prototype also demonstrates the compartmentalization feature of MPI Sessions via its use in the HPC Challenge benchmark.

Future work includes investigating ways to reduce the overhead of the exCID support code in the critical path for short messages, and algorithms to further optimize the process of exchanging local CID information. For example, when creating an MPI Sessions using a small process set, it may be possible to force all processes within the Session to use the same CID, similar to the current World Process Model. It may also be possible to use a global CID opportunistically if all processes within the Session can agree on a single CID to use. Other optimizations could include performing an allreduce operation once a communicator has been formed from a Session to ensure that all processes have the CID of all other processes, eliminating the need for an initial exchange of control messages. Other future work includes extending the MPI Sessions prototype to be compatible with Open UCX by enhancements to the UCX PML in Open MPI.

6. 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 20-XXXXXX.

This work was performed with partial support from the National Science Foundation (NSF) under Grants Nos. CCF- 1822191, CCF-1821431, CCF-1822191, and CCF-1562306. 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.

7. WORKS CITED

(2019). Retrieved from https://github.com/hppritcha/osu-microbenchmarks-sessions.

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. (2017). PMIx: Process Management for Exascale Environments. *Proceedings of the 24th European MPI Users' Group Meeting*.

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

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

Castain, R. H. (2020). *PRRTE GitHub Repository*. Retrieved from https://github.com/pmix/prrte

Derradji, S. P.-S. (2015). The bxi interconnect architecture. *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 18-25.

Gabriel, E. a. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97-104.

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

Graham, R. L.-G. (2007). An Evaulation of open mpi's matching transport layer on the cray xt. *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 161-169.

Grun, P. H. (2015). A brief introduction to the openfabrics interfaces- a new network api for maximizing high performance application efficiency. *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 34-39.

Hjelm, N. P. (n.d.). *GitHub Repository for Open MPI Sessions Prototype*. Retrieved from https://github.com/hpc/ompi/tree/sessions_new

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.

Juszczek, P. a. (2005). *Introduction to the hpc challenge benchmark suite*. Retrieved from https://apps.dtic.mil/sti/citations/ADA439315

Marts, W. P. (2019). MPI Tag matching performance on connectx and arm. *Proceedings of the 26th European MPI Users' Group Meeting*, 1-10.

Sensi, D. D. (2020). An in-depth analysis of the slingshot interconnect. *https://arxiv.org/pdf/2008.08886.pdf.*

The Ohio State University MVAPICH Benchmarks. (2019, May). Retrieved from http://mvapich.cse.ohio-state.edu/benchmarks