Locality-Aware PMI Usage for Efficient MPI Startup

Ken Raffenetti*, Neelima Bayyapu*, Dmitry Durnov†, Masamichi Takagi‡, Pavan Balaji*
* Mathematics and Computer Science Division, Argonne National Laboratory
Email: {raffenet, nbayyapu, balaji}@anl.gov
†Intel Corporation
Email: dmitry.durnov@intel.com
‡RIKEN Center for Computational Science
Email: masamichi.takagi@riken.jp

***Abstract*—In this paper, we examine usage of the Process Management Interface (PMI) during `MPI_Init`. Specifically, how PMI is used to exchange address information between peer processes in an MPI job. As node and core counts continue to increase in HPC systems, so does the amount of address data processes need to exchange. We show how by applying well-established locality-awareness techniques, we can significantly reduce the time spent in `MPI_Init`. We first present the use of shared memory to reduce the total amount of information retrieved from PMI. Next, by combining shared memory with a minimally connected set of processes, we further reduce the dependence on PMI, and employ the HPC fabric to transfer the bulk of address data. Our approach is low impact, relying on functionality already provided by MPI libraries and process managers, instead of new APIs and capabilities.**

I. Introduction

MPI is the de facto standard programming model for distributed memory systems. The vast majority of applications running on high-performance computing (HPC) clusters use MPI to communicate among parallel processes. With MPI, users are provided with basic building blocks of communication. The simplicity of the interface, along with its scalability and performance contribute to its wide success in scientific computing.

Before an MPI application can do useful work on an HPC system, it must first initialize. Although a necessary step, initialization does not contribute to the overall output of a program. It only prepares the MPI library for future work. Historically, initialization has not received the same attention as common MPI communication routines. The one-time nature of initialization means users are often willing to forgive some additional cost, since it will be spread out over the execution of the program. However some initialization costs grow increasingly noticeable at scale. It stands then that initialization should strive to be as efficient as possible and avoid wasting valuable core hours.

A non-comprehensive list of initialization tasks includes gathering information about the parallel job, setting up internal library state, and preparing resources for communication with peers. Much of this work is local: a process may request resources from the host operating system, or examine its running environment for configuration. But when preparing communication, it is often necessary to obtain external information. For example, if MPI initializes a network interface, its address may not be known until creation. Peer processes that wish to communicate over the network fabric must first exchange addresses by some external service, which will be the focus of our examination.

For many MPI implementations, processes utilize the Process Management Interface (PMI) [1] to facilitate address exchange. PMI is well-suited to this task, as it provides a logically centralized service for all processes in an MPI job. As clusters grow in size, so does the amount of data exchanged to enable communication. For large-scale jobs, this can mean long initialization times dominated by PMI usage. But is PMI really that slow, or is MPI using it inefficiently? In this paper, we will examine how MPI uses the PMI interfaces today and propose ways to improve on that usage.

The main objectives of this work are as follows:

- Analyze PMI usage for address exchange in `MPI_Init`.
- Propose new techniques for address exchange and analyze their cost.
- Present performance improvements on two supercomputers with variations in node and process count.

This paper is organized as follows: Section II explores the role of a process manager for parallel applications, and how PMI came to fill a need for libraries like MPI. Section III analyzes PMI usage for address exchange. Optimized methods of address exchange are proposed utilizing well-known techniques of shared memory and collective communication from MPI. Section IV details the experimental setup and benchmark used to evaluate and analyze the proposed optimizations on two supercomputers. A variety of node and core counts are used to give a full picture of the evaluation. Section V reviews the existing literature and compares with the proposed work in this paper. Finally, the paper concludes in Section VI.

II. Background

A. Process Manager

Process managers serve several purposes for parallel applications. First and foremost, they handle the start and stop of processes. Other user-visible functionality includes std-out/in/err aggregation, environment and signal propagation. The aim is to make launching and running parallel applications as simple and straight-forward as running a sequential binary on single machine. An example usage

```
mpiexec -n 16 ./myapp
```

launches 16 instances of `myapp`. On a typical HPC cluster, nodes are allocated from the resource manager, and `mpiexec` transparently discovers those nodes and how to access them. All this without the user needing to know where or how physical launch actually takes place. A side effect to launching processes in this manner, is that the launched processes all have a singular process manager in common. A process manager can thus act as a central coordination point for parallel processes, providing valuable functionality for libraries like MPI.

B. Process Management Interface

For users, interactions with a process manager occur at the command-line or from job scripts. Differences between implementations are not very impactful to the overall user experience. Programmatic interactions from an MPI library are a different story. Without a standard interface, MPI libraries would need specialized code for all the HPC process managers it intends to support. The APIs provided by a Cray system may be different from those on an IBM system, for example. PMI was first introduced in the MPICH [2] MPI implementation as a way to decouple process management functionality from the underlying process manager. This common abstraction allowed process managers to expose functionality in a useful, portable way. In turn, MPI libraries using the PMI API could interact with any compliant system without the need for additional code. Since its introduction, PMI has seen widespread community adoption and is supported by most HPC vendors today.

A core feature of PMI is a generic "key-value" store or KVS. The KVS acts as a centralized database to store and retrieve arbitrary data. The PMI KVS uses a "write once, read many" model, ideal for data exchanges like the one between processes during `MPI_Init`. Exposing written data requires processes to first enter into a collective barrier. This gives an implementation flexibility in how data storage and synchronization takes place. In most PMI implementations, barrier is used as an opportunity to distribute key-value pairs to process management "proxies" running on each node in the parallel job. Subsequent retrival operations are thus local, inexpensive operations. For the purposes of this paper, we do not further analyze the KVS implementation, but only the interactions made with it by MPI processes.

C. Motivation

Our work is driven by the increase in both node and core count on today's HPC machines. Machines with 100,000 or more cores are now commonplace on the Top500 [3] list of the fastest supercoputers in the world. In order to enable applications to take advantage of the full scale of these machines, an MPI library must be able to quickly and efficiently coordinate communication across ranks. The PMI interface dates back to when machines were comprised of just a few thousand cores. In the next section, we will analyze the most costly part of MPI initialization: address exchange using PMI.

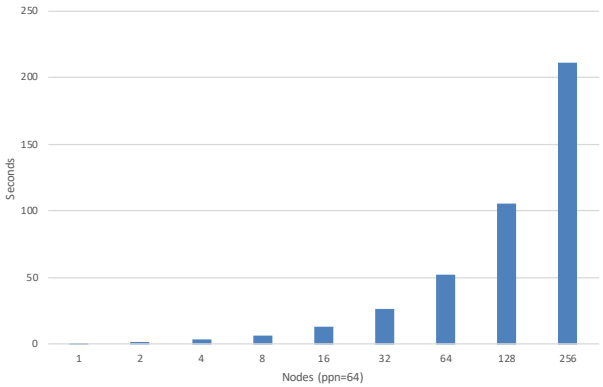

Fig. 1: Simple Address Exchange Performance

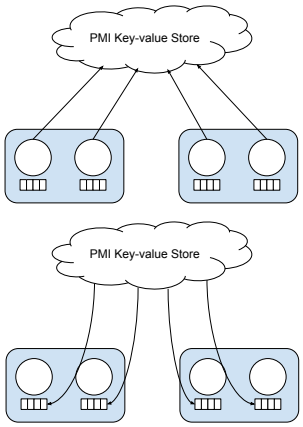

Fig. 2: Simple Address Exchange Illustrated

III. Method

We first analyze a simple address-exchange mechanism used by many PMI-based MPI implementations in Section III-A. Then, in Section III-B we present a PMI usage optimization using shared memory, and in Section III-C we present an additional optimization leveraging MPI collective communication.

A. Simple Address Exchange

A simple address exchange using PMI in psuedocode:

```
PMI_KVS_Put(rank, myaddr);
PMI_KVS_Barrier();
for (i = 0; i < size; i++)
    PMI_KVS_Get(i, &addrs[i]);
```

This method is effective and easy to understand. A process writes its address, sometimes called a business card, into the KVS, and after a barrier, retrieves business cards for all other process in the job. However, the order of this algorithm is $O(P^2)$, with each process performing P PMI_KVS_Get operations. At scale, the cost is especially noticeable, as evidenced in Figure 1.

B. Shared-Memory Optimization for Address Exchange

One of the primary shortcomings of the traditional business card exchange approach is the amount of redundant work performed on each node. Each process on the node gets the business cards for every remote process into local memory.

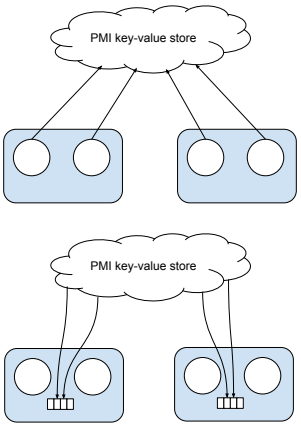

Fig. 3: Shared-Memory Address Exchange

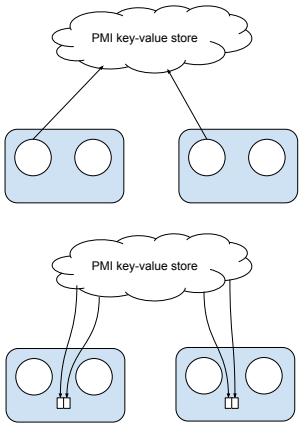

Fig. 4: Node Roots Address Exchange

With multiple processes on a node, exactly the same information is retrieved and stored multiple times. Figure 3 illustrates the redundancy.

In our first optimization, we exploit shared memory to completely eliminate the redundancy in both retrieving and storing address data. The method used is straightforward: each process only retrieves P/C number of keys and places them in a common shared-memory location so other processes on the same node can access them. This reduces the total number of keys retrieved by each node from $P \times C$ to P, which is significant when the number of cores on the node is large. Shared memory usage is made possible by the locality-awareness of MPI processes. Within MPI initialization, processes learn which of their peers are "local" (on-node) and which are "external" (off-node). This knowledge is primarily used for MPI to choose the best performing transport for communication later on, but since it is also known during address exchange, we may as well use it to our advantage.

There are two things to note about this approach that add complexity over the previous method. First, the creation of shared memory requires some overhead. Typically, one process creates a memory-mapped file and shares its filename with the other processes on the node. This sharing is done using PMI puts and gets, which adds some cost. Specifically, N puts (one per node), and P gets (one per process) for all processes to be able to access the shared memory region for address data. Secondly, there must be agreement on the size of address data in order to avoid clashes when writing into shared memory. For fixed-size network addresses, there is no additional work. However not all networks guarantee a fixed address size. We again use PMI to coordinate the maximum address length for a job in this case. Furthermore, we can use the already allocated shared memory to aid in this process. A local maximum is first computed in memory, then put into the PMI KVS by a single process on each node. Next, the maximum for all nodes is fetched by all processes, and a global maximum determined. This constitutes N PMI puts and $N * P$ PMI gets.

C. MPI Collective Optimization for Address Exchange

While the shared memory optimization discussed in Section III-B significantly reduces the amount of data fetched from PMI by MPI processes from $O(P^2)$ to $O(N * P)$, it remains that all address data must pass through the PMI database before processes can communicate directly with their peers. Once again we look to existing functionality in the MPI library for improvement: MPI collective communication. Modern MPI collective operations contain built-in optimizations for locality-awareness. These optimizations minimize the amount of traffic sent over network links by a set of "node root" processes versus much faster transfers over shared memory on a node. We exploit this technique to reduce the dependence on PMI for address exchange, and instead comunicate directly among peers. MPI provides the ideal operation for processes to collectively exchange data for our purposes: MPI_Allgather. The steps involved are as follows:

1) Each process locally determines the set of "node roots" using job information provided by the process manager.
2) Root processes put business cards into PMI KVS, which is $O(N)$ complexity.
3) Business cards are fetched using optimization from Section III-B. Peer-to-peer connections are created among node roots, which is $O(N^2)$ complexity.
4) Non-root processes place their address data into the shared memory region accessible by the node root.
5) Root processes perform an `MPI_Allgather` operation on the "node roots" communicator, populating address data in-place in shared memory. Remaining connections are established among peers, which is $O(logN)$ complexity.

With this method, only the root processes exchange business cards over PMI, reducing the number of puts to N. At the node-level, the processes retrieve N business cards, for a total of N^2 retrieves across the job. The remaining $(C-1) \times N$ business cards are exchanged using MPI_Allgather. Assuming a common tree-based implementation of MPI_Allgather, the cost would be logarithmic with the number of nodes.

IV. Evaluation

In this section we evaluate the performance of our modified address exchange methods. Each algorithm was implemented in the MPICH library using the CH4 communication device. The changes made were based on MPICH version 3.3b3.

A. Experimental Setup

Experiments were done using Bebop and Theta supercomputers Argonne National Laboratory, USA. Theta, a 11.69 petaflops system, is based on Intel Xeon Phi 7230 processors coupled with a Cray Aries interconnect in a Dragonfly topology. Theta is equipped with 4,392 nodes, each with 64 cores. Bebop has 1024 public nodes with 36 cores (Intel Broadwell) / 64 cores (Intel Knights Landing) per compute node, connected with Intel Omni-Path fabric.

On Bebop, we used OFI version 1.6.0 for Omni-Path network support through the PSM2 provider. The Hydra process manager provided by MPICH was used for process launch and management. MPICH was configured to use the PMI version 1.1 API supplied by the included Simple PMI implementation. Runs were done exclusively on the KNL node partition.

On Theta, we used OFI version 1.6.0 for Aries network support through the user level generic network interface (uGNI) provider. The Cray Linux Environment on Theta uses the Application Level Placement Scheduler (ALPS) version 6.6.1 for process launch and management. MPICH was linked with the Cray PMI library version 5.0.14, and configured to use the PMI2 API.

For benchmarking, we instrumented MPICH to measure the time taken by address exchange in `MPI_Init`. To ensure address exchange correctness, a microbenchmark using an `MPI_Alltoallv` collective was executed on `MPI_COMM_WORLD`. The reason being that in MPICH, the `MPI_Alltoallv` algorithm relies on pairwise communication between all processes in a communicator, ensuring we exercise every addresses exchanged during initialization. Our instrumentation measured the elapsed time during each phase of the address exchange algorithms. Runs were performed five times and the results averaged.

B. Address Exchange Performance Evaluation

In this section, we show how the shared memory and allgather methods compare to the original in performance. We consider two configurations in our measurements. First, we show the performance of all three methods for fully subscribed runs (one processes per core) with increasing node count. Alternatively, we fix the number of nodes and vary the number of processes per node, starting at one and doubling until the node is fully subscribed.

C. Performance Evaluation of Address Exchange on Bebop

In Figure 5, we see that for the original address exchange method, the cost grows quickly. A few seconds may be tolerable, but at greater than 100 nodes, we see address exchange take on the order of minutes. Measurements for the optimization in Section III-B (both fixed and non-fixed size addresses) are presented which drastically cut the address exchange time over the original. At 256 nodes, the shared memory method takes just under 7 seconds to exchange addresses, including the overhead of shared memory setup and maximum address length determination. Still, a steep trend is visible. For supercomputers comprised of many thousands of nodes, this will still be unsustainable. The node roots measurements represent yet another significant reduction in address exchange time. At the largest scale (256 nodes, 16384 processes), address exchange takes just under 2 seconds for regular and irregular address length, with the dominant source of time spent in `MPI_Allgather`.

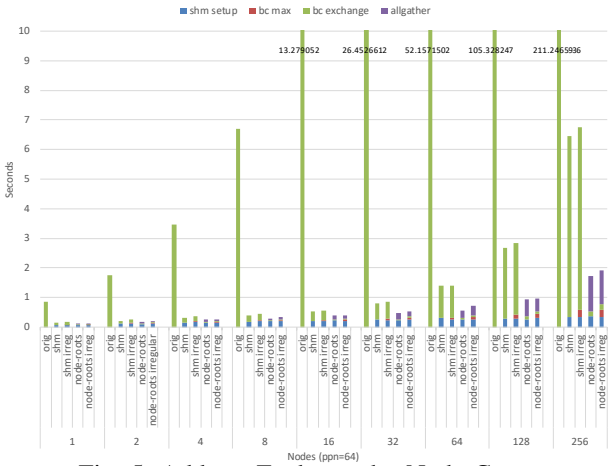

Fig. 5: Address Exchange by Node Count

Figure 6 shows the costs of address exchange for different values of processes per node. At low values (one and two), we see that the original method slightly outperforms the optimizations presented in this paper. This is expected given the additional overheads in coordinating shared memory, and relative lack of redundant work which is the advantage of our optimizations. At ppn >= 4, however, the new methods are superior. These results confirm that for applications which launch multiple MPI processes per node, our proposed exchange methods will most often shorten the address exchange time.

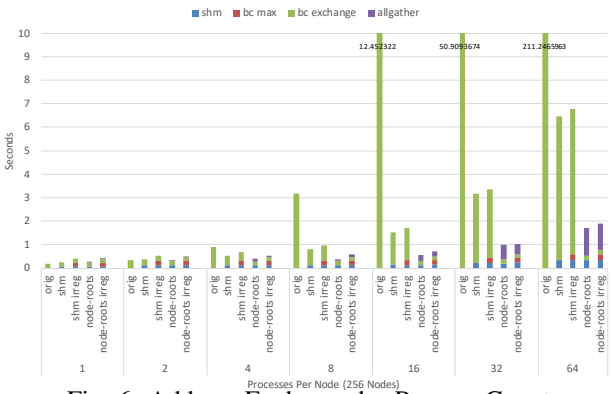

Fig. 6: Address Exchange by Process Count

D. Performance Evaluation of Address Exchange on Theta

We see similar performance improvements on the Theta supercomputer in Figure 7. Over the course of fully-subscribed production runs from 128 to 1024 nodes, our optimizations handily beat the original exchange method. For the largest runs, the allgather-based exchange completes in under three seconds. Additionally, these results on Theta illustrate the portability of our solution. Here, MPICH is linked with the Cray PMI library using the PMI2 API.

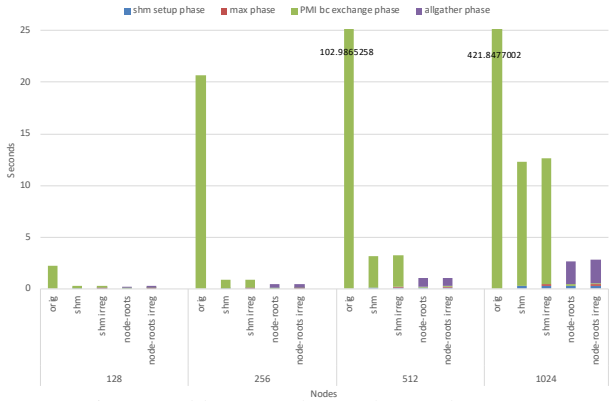

Fig. 7: Address Exchange by Node Count

V. Related Work

The literature related to PMI can be divided into three areas. One is focused on defining standard APIs. Another is optimizations to exisitng PMI library functionaliy. Lastly, there are proposed extensions to PMI for better scalability. The features and capabilities of PMI used in MPICH and its derivatives is described in [1]. A high-level overview of the goals and status of PMIx (PMI for Exascale) is given in [4], as well as a roadmap for future directions.

Others have written on the excessive time spent using PMI while initializing MPI, with address exchange rightly being the main focus. Studies often conclude that in order to support address exchange at large-scale, PMI must be extended or the implementation dramatically changed in order to eliminate bottlenecks. Proposals include new PMI collective APIs [5] that enable PMI data exchange over the HPC fabric, as opposed to TCP and Unix sockets. Nonblocking API [6] proposals allow issuing multiple PMI requests at the same time. Reimplementing PMI proxy communication over shared memory [7] instead of Unix sockets eliminates a bottleneck identified in traditional address exchange. These methods are effective at improving initialization performance, but each adds complexity for the process management system. They also operate on an assumption that traditional address exchange as shown in III-A is otherwise the best an MPI library can do using the current PMI interfaces.

Our approach achieves some of the same goals without the need to change existing PMI APIs or implementations. Storing business cards in shared memory using MPI infrastructure cuts down on PMI communication and eliminates redundant work. By first connecting a subset of processes, we were also able to use the HPC fabric for fast data movement all within MPI.

VI. Conclusions

As HPC systems grow, efficient startup time becomes more important. Taking minutes to initialize means wasting valuable core hours without contributing to program output. In this paper, we looked at the most expensive part of MPI initialization – address exchange using the Process Management Interface. PMI has been called unsuitable for extreme-scale systems due to its interface design, but this is a simplistic characterization. In this paper, we showed that by exploiting locality information, address exchange performance can be greatly improved. By using shared memory, we can eliminate on redunant work. By incorporating MPI collective communication infrastructure, we enabled the high-speed fabric without reinventing or duplicating the code needed to do so. We have shown that by working more efficiently inside MPI, we can achieve fast address exchange with minimal impact to the software running on production HPC systems.

Acknowledgments

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357 and by the JSPS KAKENHI Grant Number 23220003. This used Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.

References

[1] P. Balaji, D. Buntinas, D. Goodell, W. D. Gropp, J. Krishna, E. L. Lusk, and R. Thakur, "Pmi: A scalable parallel process-management interface for extreme-scale systems," in *17th EuroMPI Conference, Lecture Notes in Computer Science, Springer*, 11/2009 2009.

[2] "MPICH," https://www.mpich.org/, 2018.

[3] "Top500," https://www.top500.org/, 2018.

[4] R. H. Castain, D. Solt, J. Hursey, and A. Bouteiller, "Pmix: Process management for exascale environments," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17, 2017, pp. 14:1–14:10.

[5] S. Chakraborty, H. Subramoni, J. Perkins, A. Moody, M. Arnold, and D. Panda, "Pmi extensions for scalable mpi startup," in *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*, September 2014.

[6] S. Chakraborty, H. Subramoni, A. Moody, A. Venkatesh, J. Perkins, and D. Panda, "Non-blocking pmi extensions for fast mpi startup," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, May 2015.

[7] S. Chakraborty, H. Subramoni, J. L. Perkins, and D. K. Panda, "Shmempmi – shared memory based pmi for improved performance and scalability," *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 60–69, 2016.