PMI Extensions for Scalable MPI Startup *

S. Chakraborty
Dept of Computer Science and Engineering
The Ohio State University
chakraborty.52@osu.edu

H. Subramoni
Dept of Computer Science and Engineering
The Ohio State University
subramoni.1@osu.edu

J. Perkins
Dept of Computer Science and Engineering
The Ohio State University
perkinjo@cse.ohio-state.edu

A. Moody
Lawrence Livermore National Laboratory
Livermore, California
moody20@llnl.gov

M. Arnold
Dept of Computer Science and Engineering
The Ohio State University
arnoldm@cse.ohio-state.edu

D. K. Panda
Dept of Computer Science and Engineering
The Ohio State University
panda@cse.ohio-state.edu

ABSTRACT

An efficient implementation of the Process Management Interface (PMI) is crucial to enable a scalable startup of MPI jobs. We propose three extensions to the PMI specification: a ring exchange collective, a broadcast hint to Put, and an enhanced Get. We design and evaluate several PMI implementations that reduce startup costs from scaling as $O(P)$ to $O(k)$, where k is the number of keys read by the processes on each node and P is the number of processes. Our experimental evaluations show these extensions can speed up launch time of MPI jobs by 33% at 8,192 cores.

Keywords

PMI-2, SLURM, Job Launch, MPI, InfiniBand

1. INTRODUCTION

Fast, scalable startup of MPI jobs is important for numerous reasons. For example, during application development and debugging, it is often necessary to start and restart a job multiple times. Reducing startup costs from minutes to seconds cumulatively saves developers hours of time. While testing a system or while regression testing an application, many large-scale, quick-running MPI jobs must be run in succession. In this case, MPI startup becomes the dominant cost so that improving startup dramatically speeds up testing time. As a final example, fast startup is necessary to maintain machine efficiency when using fast checkpoint/restart methods [1].

A major bottleneck in starting large MPI jobs is the cost associated with exchanging information within the MPI library that is needed to initialize high-performance communication channels between processes in the job. For portability, most job launchers provide a common "out-of-band" communication infrastructure known as the Process Management Interface (PMI) [2]. Current implementations of PMI scale poorly on today's largest systems – a problem that will only be exacerbated on next generation exaflop systems that are expected to have O(100,000) to O(1,000,000) hosts [3].

PMI defines a portable interface that MPI libraries use to initialize communication between the processes of the parallel job. PMI is typically implemented as a client-server library with the process manager acting as the server and the MPI library taking the role of the client. The core functionality of PMI is to provide a global key-value store (KVS) that the MPI processes use to exchange information as key-value pairs. The basic operations in PMI are `PMI2_KVS_Put`, `PMI2_KVS_Get`, and `PMI2_KVS_Fence`, which we refer to as *Put*, *Get*, and *Fence*, respectively. *Put* adds a new key-value pair to the store, and *Get* retrieves a value given a key. *Fence* is a synchronizing collective across all processes in the job. It ensures that any *Put* made prior to the *Fence* is visible to any process via a *Get* after the *Fence*.

Existing PMI implementations have no advance knowledge about which keys are needed by which processes; as a result they broadcast each key to every PMI server. Since each MPI process often *Put*s one or more keys, such implementations require time and memory in a manner that requires a linear increase in the number of MPI processes and will therefore scale poorly on large systems. An alternative is to implement PMI as a distributed key-value store in which each server stores a subset of keys and then sends messages to exchange values with other servers during each *Get* operation. This design, however, scales poorly for any key that must be read by every process.

To alleviate this problem, we propose three extensions to the PMI specification: a ring exchange collective, a broadcast hint to *Put*, and an enhanced *Get*. The first two enable PMI implementations to avoid algorithms with linear terms and the third eliminates unnecessary synchronization. By employing these extensions, we then implement several PMI implementations which reduce MPI startup costs from scaling as $O(P)$ to scale as low as $O(k)$, where k is the number of keys read by the processes on each node and P is the number of processes in the job. At large scale when $k \ll P$, these extensions significantly reduce MPI startup time.

2. MOTIVATION AND CONTRIBUTIONS

We integrate support for PMI-2 in the widely used MVAPICH2 MPI library [4] and profile the time taken during various phases of job startup. Figure 1 shows a breakdown of the time taken during `MPI_Init` when launching a simple MPI program for different job sizes on the Stampede supercomputing system at the Texas Advanced Computing Center (TACC). We show the time spent ex-

*This research is supported in part by National Science Foundation grants #OCI-1148371, #CCF-1213084, #CNS-1347189; and a grant from Cray.
Prepared by LLNL under Contract DE-AC52-07NA27344.

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642780.

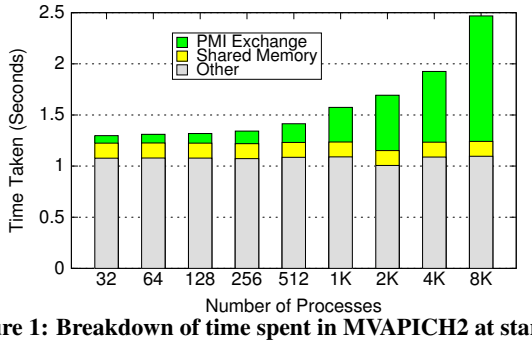


Figure 1: Breakdown of time spent in MVAPICH2 at startup

ecuting a PMI exchange, the time spent to set up shared memory communication, and the time spent in other initialization work.

During the PMI exchange, each MPI process writes its network address via a single *Put*. The processes then execute a *Fence*, and each process then issues multiple *Get* operations to lookup the addresses of all processes. We observe that as the job size increases, the PMI *Put-Fence-Get* sequence takes an increasingly larger portion of the total time and consumes the majority at larger scales. While this particular example uses MVAPICH2, it is applicable to other high-performance MPI libraries.

To analyze this cost, we profile the time taken by the individual PMI operations at various job sizes. Figure 2 indicates the amount of time spent by the MPI library in *Put*, *Fence*, and *Get* (represented by "PUT", "FENCE with PUT", and "GET", respectively) for different numbers of processes on Stampede. It is obvious that the *Fence* step is the most time consuming part of the PMI exchange.

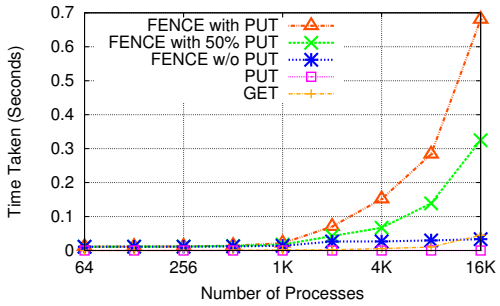


Figure 2: Time taken by different PMI-2 operations

To gain further insights into this performance trend, we explore SLURM's PMI-2 implementation of these operations. SLURM [5] (Simple Linux Utility for Resource Management) is a popular process manager used by many small and large clusters. SLURM has a main controller daemon *slurmctld* running on the controller node and another daemon *slurmd* running on each of the compute nodes. The *slurmctld* is responsible for scheduling, allocating, and managing jobs while the *slurmd* launches and cleans up processes, redirects I/O, etc. While launching a job, *slurmctld* instructs the *slurmd*s on the allocated nodes to set up environment variables and launch the processes. The *slurmd*s participating in a job set up a hierarchical k-nomial tree with an *srun* process as the root as shown in Figure 3.

In SLURM's PMI-2 implementation, the *Put* and *Get* operations are local and do not present a bottleneck as seen in Figure 2. However, the *Fence* operation synchronizes all processes through a barrier followed by an allgather of key-value pairs implemented as a gather and a broadcast over the SLURM tree. Note that this data distribution happens irrespective of whether the destination process performs a *Get* for the data or not. To further understand the over-

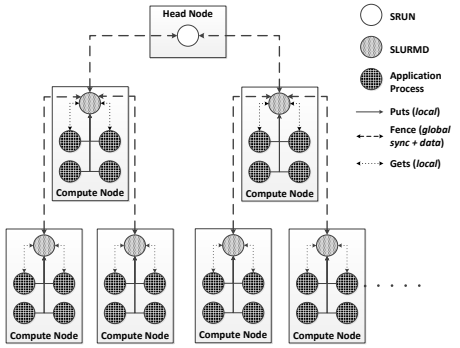


Figure 3: Hierarchical communication scheme in SLURM

heads in the *Fence* operation, we profile the time taken by *Fence* without a preceding *Put* (represented by "FENCE w/o PUT") and with a single 16-byte *Put* from processes with odd rank (represented by "FENCE with 50% PUT"). It is clear that the data movement is the dominant cost factor of the *Fence* operation.

The existing design for *Fence* where all data is distributed to all processes works well if most of the processes actually require most keys. However, this strategy is sub-optimal for applications that use sparse communication patterns. For example, MVAPICH2 is able to create a ring structure after exchanging a very small number of keys and can use that ring to exchange the rest of the information. In such cases, the overhead of copying a large amount of data to uninterested processes is high. Unfortunately, the current PMI-2 specification does not provide an option for the application to specify whether a particular key is required by a majority of the processes or not. Thus, PMI applications in which each key is only read by a small number of processes are penalized by the allgather exchange in *Fence* and may instead benefit from using "on-demand" *Get* and *Put* operations.

These issues lead us to the following broad problem statement — **Can we enhance the existing PMI-2 design and specification to improve the startup time of MPI-based parallel applications on large supercomputing systems?**

In this paper, we address this challenge. We propose efficient designs for non-local *Put* and *Get* operations in PMI. We implement these designs in the SLURM process manager, and evaluate their performance. The results of our experimental evaluation show that these new designs significantly improve the launch time of MPI jobs, which leads us to propose extensions to the PMI-2 specification so that additional information can be passed to the library, which will enable more scalable PMI implementations.

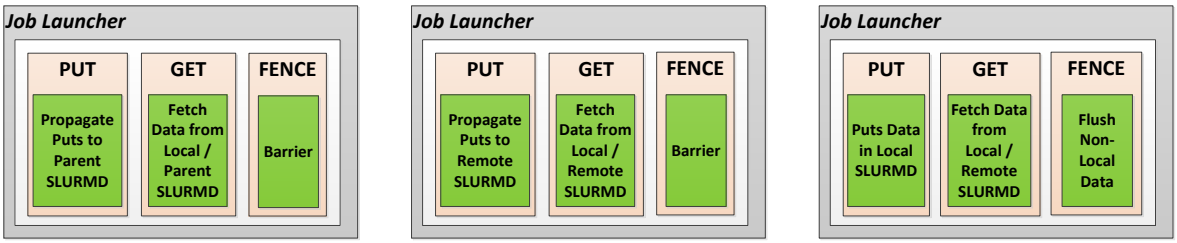
3. NON-LOCAL PUT AND GET

To eliminate the allgather from the *Fence* operation, the *Get* and *Put* operations must be able to exchange data via remote operations. In this section we describe and evaluate a few different designs for exchange of information while maintaining consistency. Figure 4 depicts the high level overview of the different PMI implementation designs.

3.1 Design 1: Hierarchical Tree

In this scheme we utilize SLURM's hierarchical tree structure for data movement. During a *Put*, the process sends the key-value pair to the local *slurmd*. The *slurmd* propagates the pair to its parent *slurmd* until it reaches the root *srun*. All of the involved *slurmd*s (i.e. the ancestors of the originating *slurmd*) also cache this key value pair locally.

The *Get* method is similar. When a *slurmd* handles a *Get* request, it first checks its local cache for that key and immediately responds



(a) Hierarchical Tree (b) Distributed Key-Value Store (c) Direct Access from Source

Figure 4: Responsibilities of PMI-2 functions in different designs

if found. If the key is not found, it forwards the request to its parent *slurmd/srun*. The request is propagated up the tree until it reaches a process that has the key, at which point, a reply with the value is sent directly back to the originating *slurmd*. The *slurmd* can then send the response to the requesting local process.

To improve the *Get* latency, each *slurmd* appends its hostname to the forwarded request and the responding *slurmd/srun* sends the reply to all of them in parallel. Figure 5 shows how the request and response messages for *Get* and *Put* propagate through the tree.

Figure 4(a) depicts the functions of *Put*, *Get*, and *Fence* under this design. The major benefit of this approach is that all data transfers happen over existing connections between *slurmd*s. Due to the lack of connection setup and tear-down overhead, this design exhibits low latency if the key is cached by a close ancestor.

However, the tree based approach suffers from two major bottlenecks. The number of messages exchanged grows quickly as number of *Puts* and *Gets* increase. Also, all of the key-value pair go through the root *srun* at least once, which scales linearly with the total number of keys.

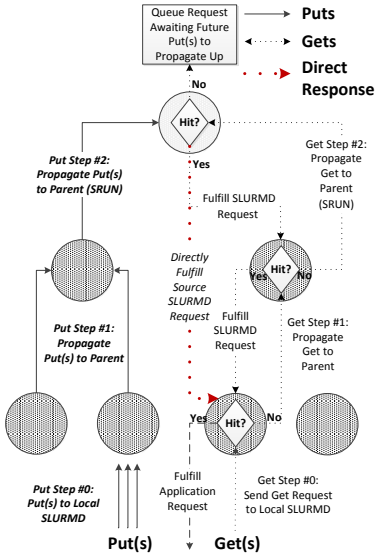


Figure 5: Proposed hierarchical tree-based scheme

3.2 Design 2: Distributed Key-Value Store

In this design all nodes can act as the primary source for a subset of the keys. Each key is hashed to a unique host id. In a *Put* operation, the local *slurmd* hashes the key to identify the owner node and sends the key-value pair to that node. For *Get* requests, the *slurmd* checks the local cache first and sends a request to the owner node in case of a miss. Only two message exchanges are required for a *Put* and a *Get* in this approach, as shown in Figure 6. Compared to the tree-based design which potentially requires $O(logN)$ exchanges for each *Get* or *Put* operation, where N is the number of nodes, the distributed key-value store can provide better latency. The downside is that each *Put* or *Get* may need to create a new connection, whereas existing connections are reused in the tree-based design.

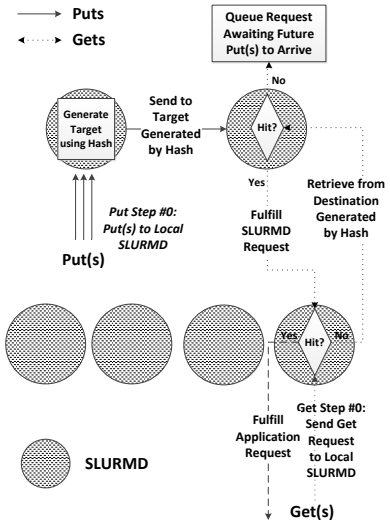


Figure 6: Proposed distributed key-value store based scheme

For this scheme to work, all nodes need to be assigned a unique node id and all processes must know how to map a node id to a hostname in order to send a message. Since the process manager already knows this mapping, it can pass this information to the processes through PMI-2 Job Attributes or environment variables. The functions of *Put*, *Get*, and *Fence* operations in this scheme are shown in Figure 4(b).

3.3 Design 3: Direct Access from Source

In many cases the process issuing *Get* knows the rank of the process that executed the corresponding *Put*. In these cases, it might be beneficial to bypass the hierarchical structure or the distributed key-value store and directly access the key from the source. The PMI-2 standard includes an optional parameter in *Get* for the caller to provide the source rank, but current implementations ignore this. To look up the hostname from the process rank, the PMI library must have the process mapping available.

Figure 4(c) depicts the changes made to the PMI functions to effect this change. In this approach the transfer can be orchestrated with just a single message exchange.

3.4 Maintaining Consistency

The PMI-2 specification mandates that each *Put* executed before a *Fence* must be visible to a *Get* after the *Fence*. Also, with multiple *Put*s and *Fence*s, the latest value for the key must be returned to the *Get*. In the existing design where *Fence* also propagates the data,

this is easily achieved. To ensure consistency in the new design, each *slurmd* maintains a global sequence number which is incremented each time *Fence* is called. We also introduced a sequence number associated with each key-value pair which reflects the sequence number when it was put into the cache. While serving the request for a non-local key, we compare the key's sequence id with the global sequence number to check for staleness. If the value is not fresh, *slurmd* purges the entry from the cache, treats it like a miss, and forwards the request to the appropriate owner which can respond with the updated value.

If the PMI application does not perform multiple *Put* operations on the same key, the *Fence* operation can be completely omitted. However, in this scenario *Get* operations can be performed before the corresponding *Puts*. This can be solved by two alternatives: polling and callback. With polling, the requesting *slurmd* periodically polls the remote host until the *Put* happens and the remote process responds or a timeout occurs. Clearly this approach would generate a lot of redundant network activity and does not give the lowest latency possible. In the callback approach, the primary source (*srun* in the tree-based design, hash based owner in the DKVS design, and the source in the direct access design) maintains a list of pending requests that could not be served. Whenever a *Put* happens, it scans the list for pending requests for the same key and dispatches appropriate responses. This method does not generate any additional network traffic and the requests are satisfied as early as possible.

The callback approach is applied at the node level as well to prevent multiple processes sending out redundant messages requesting the same key. A queue is maintained for each outstanding request; if another process requests for the same key before the response has been recieved, the local rank of the process is recorded in the queue. When the response from the remote host arrives, the associated queue is processed and the waiting processes are served.

4. PROPOSED PMI-2 EXTENSIONS

In the following sections, we propose three extensions to the PMI standard that would allow PMI implementations to support MPI libraries in a more scalable fashion.

4.1 New Ring Exchange Collective

First, we propose the addition of a new "ring exchange" operation. This routine is collective across all processes like *Fence* and it distributes data amongst processes in a shift pattern. The prototype for the proposed method is:

```
int PMI2_Ring(
  const char value[],
  int* size,
  int* rank,
  char left[],
  char right[]
);
```

Each process enters its input data in the `value` parameter. As output, a process acquires the number of processes in the ring in `size`, and it acquires its rank within the ring in `rank`. The rank of a process within the ring may be different from its global MPI rank, i.e., the rank returned by `PMI2_Init`. This feature is critical to enable efficient implementations. Also as output, the calling process receives copies of the data input by its neighbor processes in the ring. The `left` and the `right` buffers hold the values entered by the two neighbor processes of the caller. String buffers are NULL-terminated and they are limited to a maximum length of `PMI2_MAX_VALLEN` bytes defined by the implementation.

Given such a function, most MPI libraries can interconnect a group of processes after executing a single `PMI2_Ring` call. Each process can enter its network address as input and receive the network addresses of its left and right neighbors as output. With this information, one can then implement numerous collectives such as barrier, broadcast, and allreduce in $O(logP)$ time where P is the size of the ring [6]. Additionally, we believe this function can be implemented efficiently in most existing PMI implementations. For example, in SLURM, one could overlay the ring on the SLURM tree and implement the function as a prefix scan in $O(p \cdot logN)$ time, where N is the number of nodes and p is the number of MPI processes per node. This allows the MPI library to avoid *Put*, *Get*, and *Fence* completely in most cases and therefore bypass the inherent complexities and inefficiencies in maintaining a global key-value store within PMI.

We implement such a ring in SLURM's PMI library. Instead of overlaying the ring on the existing SLURM tree as described above, we opt for a more direct approach and used SLURM's point-to-point message capability instead. The efficiency of this implementation stems from the fact that each node executes exactly two inter-node exchanges while the majority of the information is exchanged intra-node.

4.2 Broadcast Hint for Put Operations

A PMI application can have different keys that are accessed in different patterns. For example, to implement a broadcast operation a single process can perform a *Put* operation and all other processes can perform a *Get*. We classify these type of keys as `DENSE`. At the same time, there can be keys which are read by only a small number of processes. For example, an MPI library may form a ring or a tree based overlay network which requires each key to be read by a small number of processes independent of the number of total processes in the job. The keys are classified as `SPARSE`. In case of MVAPICH2, all the keys used to bootstrap the library falls under the `SPARSE` category.

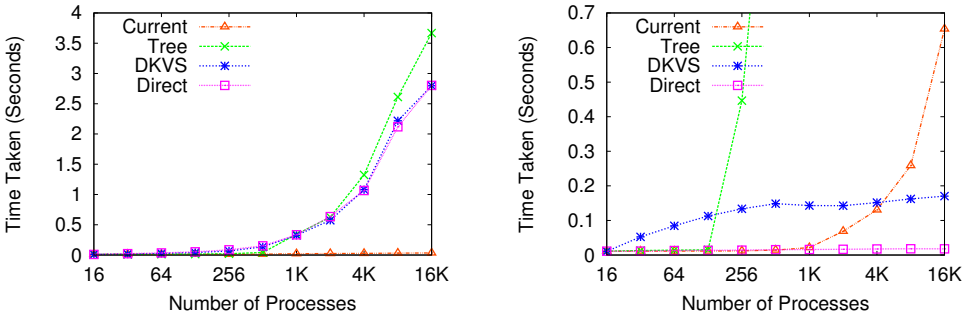
It is difficult for a PMI implementation to efficiently support both class of keys without prior knowledge about their access patterns. Broadcasting a `SPARSE` key to each process is inefficient, similarly looking up a `DENSE` key through large number of remote operations is not desirable. To eliminate this issue we propose to enhance the `PMI2_KVS_Put` method to accept an additional parameter which will provide this information. The key can be flagged as `SPARSE` or `DENSE`. The `DENSE` keys can be propagated through the existing `PMI2_KVS_Fence` method whereas the `SPARSE` keys are looked up on-demand. The PMI library is free to choose an appropriate implementation for the keys which are not flagged either way.

4.3 Enhanced Get Operation

Currently `PMI2_KVS_Get` is defined as a local operation which fails if the requested key is not available locally (recieved through previous *Fence* operation). We enhance `PMI2_KVS_Get` to search for the key at the remote host in this scenario and wait until the remote node performs the corresponding `PMI2_KVS_Put` operation. This unifies the behavior of `PMI2_KVS_Get` with the local `PMI2_Info_GetNodeAttr`, which can wait indefinitely until the requested key is found.

5. EXPERIMENTAL RESULTS

In this section, we describe the experimental setup used to conduct micro-benchmark and application experiments to evaluate the efficacy of existing startup mechanisms and that of the proposed designs. An in-depth analysis of the results is also provided to correlate design motivations and observed behavior. All results reported



(a) Dense Key: one Get on same key per process (b) Sparse Key: one Put + two Gets per process

Figure 7: Time taken for different access patterns under each design

here are averages of twenty runs to discard the effect of system noise.

5.1 Experimental Setup

We used the Stampede supercomputing system at TACC to take all performance numbers. Each compute node is equipped with Intel SandyBridge series of processors, using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (56 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. SLURM-2.6.5 and MVAPICH2-2.0b were used to implement the proposed designs. All numbers reported were taken in fully subscribed mode with 16 processes per node.

5.2 Comparison of Different Designs

In Figure 7 we compare the efficacy of the different designs with different class of keys. The baseline design is the default PMI-2 implementation provided by SLURM, denoted as "Current". "Tree" refers to Design 1 (Hierarchical Tree), "DKVS" refers to Design 2 (Distributed Key Value Store) and "Direct" denotes Design 3 (Direct Access from Source).

In these experiments, each *Put* submits a 16-byte value. "Current" executes a sequence of *Put*, *Fence*, and *Get*. In the other schemes, only *Put* and *Get* operations are needed and *Fence* is omitted. We show the time taken for all processes to complete these operations.

In Figure 7(a), a single process executes a single *Put* and all processes execute a *Get* on this key, emulating a broadcast operation using a key flagged as `DENSE`. For this type of keys, the "Current" approach with *Fence* performs better.

In Figure 7(b) each process submits a single key with *Put* and every process executes two *Get* operations to lookup the keys submitted by its neighbors in rank space. This effectively implements a ring exchange and shows the use of a `SPARSE` key. For this type of keys, "DKVS" and "Direct" are more suitable, as they impose a constant cost with process count while "Current" scales linearly.

In general across all experiments, performance of the Current scheme closely mimics the performance of the *Fence* operation. Limited scalability of the "Tree" based scheme causes it to perform poorly at larger scale. The difference of one extra exchange between the "DKVS" and the "Direct" schemes matches expected behavior.

The critical point is that different strategies are required to transport different key types, and a simple hint to *Put* is sufficient for the PMI implementation to distinguish between these types.

Note also that while Figure 7(b) emulates a ring exchange using *Put* and *Get* calls, there is still value in defining `PMI2_Ring`. First, if the processes within a node do not have consecutive PMI ranks, `PMI2_Ring` can still be implemented using as few as two inter-node messages per node, whereas emulating the ring exchange with *Put* and *Get* requires as many as $2 \cdot p$ inter-node messages per node, where p is the number of processes per node. As a result, `PMI2_Ring` can be significantly faster, and its advantage is expected to increase on future systems as the number of cores available on each node increases. Second, we believe `PMI2_Ring` can be naturally implemented in most PMI implementations whereas adding support for a distributed key-value store is likely more intrusive. Thus, even PMI implementations that force all keys to be distributed via an allgather may still be able to efficiently support `PMI2_Ring`.

5.3 Impact on MPI Startup

With the proposed `PMI2_Ring` method, MPI processes need to do minimal communication over the out-of-band PMI channel. Once the ring is established, the rest of the startup information can be exchanged over the high performance InfiniBand network. The time spent by the MPI library in different phases of communication in this strategy is shown in Figure 9.

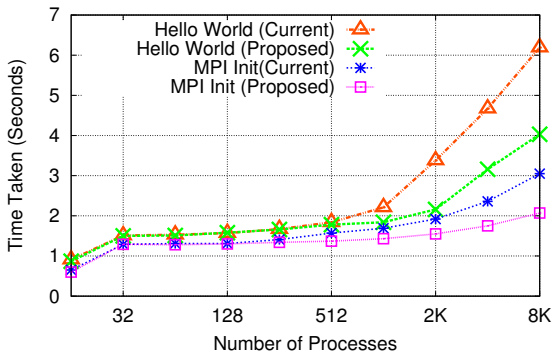


Figure 8: Time taken by MPI_Init and Hello World

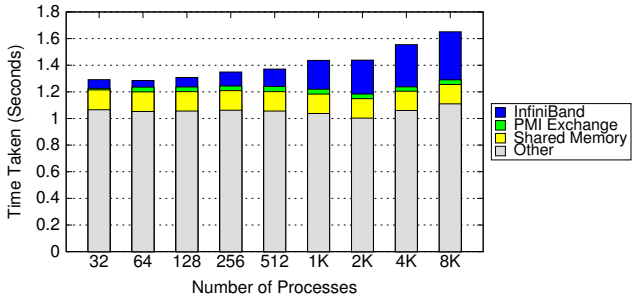


Figure 9: Breakdown of time spent by MVAPICH2 in various phases with the proposed designs

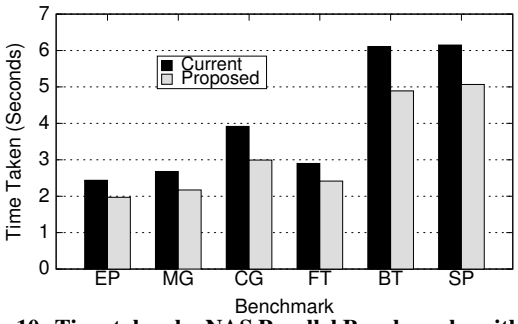


Figure 10: Time taken by NAS Parallel Benchmarks with 1,024 processes

With the new design, time spent in PMI exchanges is independent of process count. In our current implementation, we use linear-scaling algorithms to transfer data over InfiniBand, which we plan to improve in the future. Despite this inefficiency, the hybrid approach is faster than exchanging all information over PMI.

In Figure 8, we show the time taken to execute a simple Hello World program with the modified SLURM and MVAPICH2 and compare against the unmodified version. Compared to the existing PMI-2 implementation, the new designs improved performance of `MPI_Init` by up to 34% and reduced the time taken by Hello World by up to 33%. From the trends, we expect further improvement at larger scales.

We also measure the running time of some applications from the NAS Parallel Benchmarks (NPB) [7] with class B data and 1,024 processes in fully subscribed mode and observe improvements of up to 20% in total running time as shown in Figure 10.

6. RELATED WORK

There has been significant work in the area of improving performance and scalability of launching parallel applications. Multiple process managers like PBS, MPD, Mpiexec, and Hydra have been developed to reduce job scheduling and launch times.

Yu et al [8] explored using InfiniBand to reduce start up costs of MPI jobs. We furthered this concept with the introduction of the proposed `PMI2_Ring` collective, thus reducing the communication overhead even further. Sridhar et al proposed using a hierarchical ssh based tree structure similar to SLURM's node daemon implementation [9]. Gupta et al [10] proposed a smp-aware multi level startup scheme with batching of remote shells. Goehner et al analyzed the effect of different tree configurations and proposed a framework called LIBI [11]. The impact of node level caching on startup performance was evaluated by Sridhar et al in [12]. However, none of these works take advantage of faster communication over high performance networks like InfiniBand.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we explored the limitations of the PMI-2 API regarding sparse communication patterns. We evaluated various alternative designs to decouple data transfer from *Fence* and enhance *Put* and *Get* operations. We also proposed `PMI2_Ring`, a new data exchange collective that can be highly optimized. Using these, we showed how MPI libraries can reduce communication cost during start-up by avoiding nonessential data movement and taking advantage of high performance networks. We were able to significantly improve performance and scalability of job startup in MPI libraries. At 8,192 processes, launch time of MPI jobs was reduced by up to 33%, with further gains expected at larger scales.

We conclude that the proposed three extensions to the PMI specification, namely a ring exchange collective, a broadcast hint to *Put*, and an enhanced *Get*, provide dramatic benefits when starting up large MPI jobs.

Going forward, we plan to explore more efficient methods to exchange data over the ring or tree-based overlays to remove other bottlenecks and further improve the performance and scalability of MPI startup.

References

[1] J. Daly, "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps," in *International Conference on Computational Science*, vol. 2660 of *Lecture Notes in Computer Science*, pp. 3–12, 2003.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A Scalable Parallel Process-management Interface for Extreme-scale Systems," in *Recent Advances in the Message Passing Interface*, pp. 31–41, Springer, 2010.

[3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale Computing Study: Technology Challenges in Achieving Exascale Systems," *DARPA IPTO, Tech. Rep*, vol. 15, 2008.

[4] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC," in *Int'l Workshop on Sustainable Software for Science: Practice and Experiences, Held in Conjunction with Int'l Conference on Supercomputing, SC*, 2013.

[5] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *JSSPP 2003*, pp. 44–60, Springer, 2003.

[6] A. Moody, D. H. Ahn, and B. R. de Supinski, "Exascale Algorithms for Generalized MPI_Comm_split," in *Recent Advances in the Message Passing Interface*, vol. 6960 of *Lecture Notes in Computer Science*, pp. 9–18, Springer, 2011.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, "The NAS Parallel Benchmarks," *IJHPCA*, vol. 5, no. 3, pp. 63–73, 1991.

[8] W. Yu, J. Wu, and D. K. Panda, "Fast and Scalable Startup of MPI Programs in InfiniBand Clusters," in *HiPC 2004*, pp. 440–449, Springer, 2005.

[9] J. K. Sridhar, M. J. Koop, J. L. Perkins, and D. K. Panda, "ScELA: Scalable and Extensible Launching Architecture for Clusters," in *HiPC 2008*, pp. 323–335, Springer, 2008.

[10] A. Gupta, G. Zheng, and L. V. Kalé, "A Multi-level Scalable Startup for Parallel Applications," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, pp. 41–48, ACM, 2011.

[11] J. D. Goehner, D. C. Arnold, D. H. Ahn, G. L. Lee, B. R. de Supinski, M. P. LeGendre, B. P. Miller, and M. Schulz, "LIBI: A Framework for Bootstrapping Extreme Scale Software Systems," *Parallel Computing*, vol. 39, no. 3, pp. 167–176, 2013.

[12] J. K. Sridhar and D. K. Panda, "Impact of Node Level Caching in MPI Job Launch Mechanisms," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 230–239, Springer, 2009.