SHMEMPMI - Shared Memory based PMI for Improved Performance and Scalability

Sourav Chakraborty, Hari Subramoni, Jonathan Perkins and D. K. Panda
Department of Computer Science and Engineering
The Ohio State University
Email: {chakrabs, subramon, perkinjo, panda}@cse.ohio-state.edu

***Abstract*—Dense systems with large number of cores per node are becoming increasingly popular. Existing designs of the Process Management Interface (PMI) show poor scalability in terms of performance and memory consumption on such systems with large number of processes concurrently accessing the PMI interface. Our analysis shows the local socket-based communication scheme used by PMI to be a major bottleneck. While using a shared memory based channel can avoid this bottleneck and thus reduce memory consumption and improve performance, there are several challenges associated with such a design. We investigate several such alternatives and propose a novel design that is based on a hybrid socket+shared memory based communication protocol and uses multiple shared memory regions. This design can reduce the memory usage per node by a factor of $Processes\ per\ Node$. Our evaluations show that memory consumption per node can be reduced by an estimated 1 GB with 1 million MPI processes and 16 processes per node. Additionally, performance of PMI Get is improved by 1,000 times compared to the existing design. The proposed design is backward compatible, secure, and imposes negligible overhead.**

***Keywords*-PMI, Shared Memory, Hash Table, Memory Scalability, MPI**

I. Introduction

Innovations in manufacturing technology and increasing computational demands have led to a rapid emergence of multi- and many-core architectures in High Performance Computing (HPC) systems. Next generation architectures for exascale computing are expected to have even higher core-count per node. However, total memory on such architectures are not expected to increase dramatically, leading to reduced amount of available memory per core. Consequently, memory scalability of the software — programming models and supporting frameworks — is becoming more important.

Most parallel programming frameworks need to interact closely with the system's process manager to bootstrap and initialize itself. The process manager is a logically centralized entity that is responsible for launching and terminating processes, providing environment information to processes and manage information exchange among processes of the parallel application [1]. The Process Management Interface (PMI) is a portable interface between the parallel programming library and the process manager. In addition to providing various basic information and functionality required by the programming library, PMI also facilitates information exchange by exposing a globally shared key-value store and a set of functions (PMI2_KVS_Put, PMI2_KVS_Fence and PMI2_KVS_Get) to interact with it. PMI is used by many major implementations of various parallel programming frameworks like Message Passing Interface (MPI) and OpenSHMEM as well as most popular process managers including SLURM [2], Hydra [3], and mpirun_rsh [4]. While PMI itself is generic enough for almost any parallel programming framework, in this paper we primarily consider the MPI programming model. However, our proposed solutions are equally applicable to other parallel programming frameworks as well.

The PMI standard consists of two separate specifications — the PMI API proper (also known as the *client* API) exposed to the users of PMI, and the *wire protocol* for communication between the application processes and the process manager. The wire protocol defined by the PMI standard assumes a client-server model. The application processes or the MPI library work as the client and the process manager acts as the server. The communication between the client and the server is done through a stream of requests and replies. This client-server based communication model comes with certain drawbacks. Since the server can process only one or a few requests in parallel, with a large number of concurrent clients the requests get serialized at the server. Figure 1 shows the effect of increasing number of concurrent requests on the time taken for completing each request. Even with no inter-node communication, a large number of concurrent requests results in a significant increase in average turnaround time. Figure 1 shows the time taken for for a single PMI2_KVS_Get request with different number of concurrent clients. The latency goes up from approximately 20μs to 250μs as the number of clients increases from 1 to 32. The setup used for this experiment is described in Section VII-C1.

One of the major uses of PMI in high performance MPI libraries is to exchange information relevant to setting up communication channels over the high-performance networks like InfiniBand. Each communicating process publishes its own network address (a hardware address and an unique identifier for each process) via PMI Put and fetches the addresses for its peers using PMI Gets. As shown in Figure 1, the latency for a single PMI Get can be quite high compared to the typical communication latency for small message transfers observed in typical high-performance networks (around 1μs). As this request-reply based model prevents efficient bulk-sharing of data between clients and the server, MPI libraries are forced to copy this information into the process's memory. Since

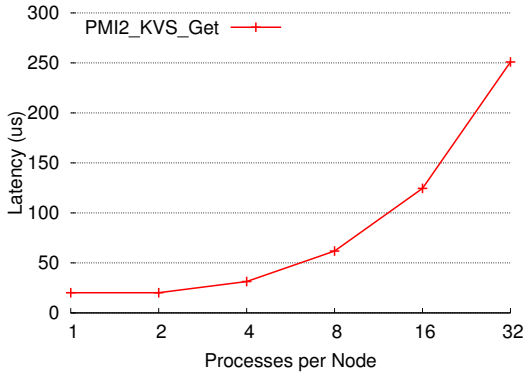


Fig. 1. Time taken for one PMI2_KVS_Get with different number of processes per node

PMI provides no delete functionality, the process manager also holds a copy of the entire key-value store for the lifetime of the job. Consequently, the information is replicated $PPN+1$ times per node where PPN is the number of MPI processes per node. Figure 2 shows the amount of memory consumed for storing the network endpoint addresses for different process counts with 16, 32, and 64 processes per node in the MVAPICH2 [5] MPI library. With a million MPI processes and 16 processes per node, more than 1 GB of memory would be consumed per node for storing this address table.

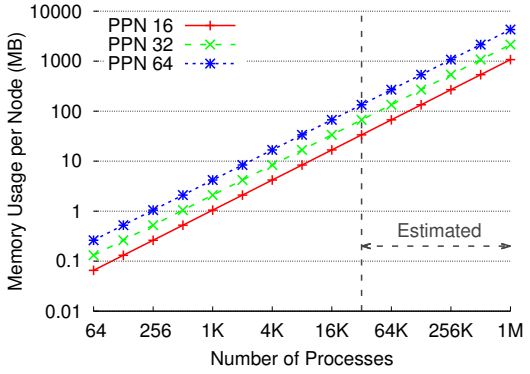


Fig. 2. Memory usage per node for different number of processes with 16, 32 and 64 processes per node

The performance and memory scalability issues are only going to be exacerbated by the next generation architectures with denser nodes and larger number of cores. To address this challenge, we propose to use shared memory regions between the PMI server and the client to reduce duplication and improve access latency. Instead of copying the information from the key-value store to each process's private memory, the information can be pushed to a shared memory region by the process manager and made available to the client processes. The client processes would then able to access the data directly without going through a costly request-response cycle involving the server. However, there are several design challenges related to which must be addressed in such a shared memory based design of the PMI protocol:

- How to synchronize concurrent writes to the shared memory region by multiple clients to maintain consistency?
- How can we avoid solutions based on memory-polling to avoid adverse impact on performance?
- Are there any security issues associated with granting client processes write permission to the shared memory regions and how can one work around them?
- Can the changes be backward compatible with the existing PMI specification?

In this paper we identify and address these challenges and evaluate our proposed solutions in terms of memory efficiency, scalability and performance.

II. Background

In this section, we describe the background information necessary for the paper.

A. Process Management Interface (PMI)

PMI defines a portable interface between the parallel application processes and the process manager. The PMI standard consists of two separate specifications - the client side API and the wire protocol. The MPI library calls the client side APIs while the wire protocol is used for communication between the client and the server. While the PMI specification does not require a PMI provider to adhere to the wire protocol, the overwhelming majority of the available process managers support the reference wire protocol for interoperability.

1) PMI Put-Fence-Get: In addition to providing support for creating, connecting with, and exiting parallel jobs, accessing information about the parallel job or the node on which a process is running, exchanging information related to the MPI Name publishing interface, PMI also provides a mechanism to exchange information used to connect processes together. To achieve this, PMI defines a common Key-Value Store (KVS) for all the parallel processes that are part of a job. The PMI API also defines the following functions to interact with the key-value store:

- PMI2_KVS_Put — Adds a new key-value pair to the key-value store. Each process can perform arbitrary number of Puts. Multiple Puts with the same key may lead to undefined behavior.
- PMI2_KVS_Get — Looks up the value for a given key from the key-value store.
- PMI2_KVS_Fence — Fence is a collective operation that must be called by all processes. Any key-value pairs added before the Fence must be visible to the Gets performed later.

2) PMI Allgather: In most implementations, the Put and Get are implemented as local (intra-node) operations while Fence performs the inter-node communication to distribute the key-value pairs to all the connected processes. In our earlier work [6] we proposed an extension named PMIX_Allgather to optimize the common use case of symmetric data movement where each process broadcasts a single key-value pair and looks up the same information from every other process. This is functionally similar to each process performing a single Put using their rank as the key, followed by a Fence and $N_{procs}-1$ Gets. We also proposed a set of non-blocking extensions to

overlap the PMI operations with initialization and computation. In this paper we use the function PMIX_Allgather which is defined as:

- PMIX_Allgather— Each process provides an input value and a buffer. The input values are gathered into the buffer ordered by their source rank. Once the operation is complete, the values can be directly accessed from the buffer.

B. MPI over InfiniBand and High Speed Ethernet

InfiniBand is a popular switched interconnect standard being used by more than 47% of the Top500 supercomputing systems. [7] according to the November '15 ranking. For two processes to communicate over InfiniBand, their network endpoint addresses must be exchanged beforehand through an out-of-band communication channel. MPI libraries for InfiniBand typically use PMI as the mechanism to implement this out-of-band exchange. In this paper, we use the MVAPICH2 [5] MPI library for our evaluations.

C. SLURM

SLURM [2] (Simple Linux Utility for Resource Management) is a popular process manager used by many HPC clusters. SLURM has a main controller daemon *slurmctld* running on the controller node and another daemon *slurmd* running on 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 initialize environment variables and launch the processes. The *slurmd*s participating in a job set up a hierarchical k-ary tree with an *srun* process as the root as shown in Figure 3.

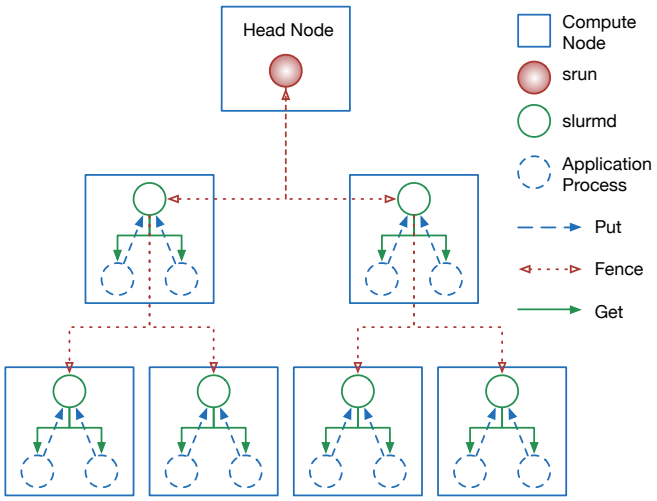


Fig. 3. Hierarchical communication scheme in SLURM

Each slurmd works as the local PMI server and maintains a copy of the key-value store. When a client process performs a Put, the key-value pair is stored in a temporary buffer. During a Fence, the key-value pairs are aggregated and sent to the parent slurmd/srun. Once the root of the tree (srun) has all the pairs, it broadcasts them through the tree to all children slurmds. The slurmds then merge the incoming pairs into the local key-value store which can be accessed by the clients by making Get requests. A similar process is followed during Allgather, except the output buffer is copied to the process's memory and the values are accessed directly from the output buffer without further interaction with the local slurmd.

III. Hybrid Shared Memory Based Communication

In the existing design, all communication between the PMI clients and the server go over local UNIX sockets. The server opens sockets to the local clients during initialization, waits on a *select* call and waits for client requests. We refer to this design as *Pure Socket Based* design. With the introduction of memory regions that are shared between the clients and the server, all or some of the client-server communication can be moved over from the sockets.

A. Pure Shared Memory Based Communication

In the *Pure Shared Memory Based* design, all communication between the client and the server go over a shared memory based channel. The server polls on the shared memory channel to listen for requests and replies through the same channel. Predefined areas within the shared memory region act as the buffers required for writing and reading the messages. There are a few drawbacks associated with this design:

1) Impact on Performance: With a pure shared memory channel, the server and the clients need to use a polling or a lock based design. While the polling method achieves low latency, it is quite CPU intensive and wastes a lot of cycles. Further, this takes away valuable CPU cycles away from MPI processes, significantly hurting application performance. Using a lock based approach can reduce this cost but would require spawning an extra thread for processing inter-node message to avoid deadlocks.

2) Security: In this design, the clients (application processes) require permission to write to a memory region owned by the server (process manager). In many HPC systems, the process manager needs to perform various book-keeping and maintenance tasks and hence runs with elevated privileges (e.g. SLURM, PBS). Since the client is controlled by the application, the process manager would be susceptible to buffer overflow or privilege escalation attacks from malicious or erroneous client codes if the clients are granted write permission to the memory regions held by the privileged process manager.

B. Hybrid Socket+SHMEM Based Communication

Based on these observation, we choose a hybrid socket + shared memory based communication scheme for our design. We observe that the number of Put and Fence operations performed by each MPI process is generally constant with respect to the number of processes. Thus, moving them to a shared memory channel would neither improve nor degrade performance noticeably. On the other hand, Get calls

are responsible for bulk of the data transfer between the server and the clients. Consequently, performing Gets over the faster shared memory communication channel would lead to better performance. Thus, granting read-only access to shared memory regions to clients is sufficient to achieve high performance without sacrificing on security. Furthermore, this strategy avoids the issue of synchronizing multiple concurrent writers to the shared memory region as only the PMI server is granted write permission.

In the hybrid design, the request and response messages for most functions like PMI2_KVS_Put and PMI2_KVS_Fence still go through the socket based path. For PMI2_KVS_Get, the clients can directly read from the shared memory region to extract the value for the requested key.

IV. Shared Memory Based Put, Fence, and Get

When a client process performs a Put, the server stores the new key-value pair in a temporary buffer. During a Fence, these key-value pairs are exchanged with other remote servers. Once the Fence operation is complete and the server has received all the key-value pairs, the server opens a shared memory region, stores all the key-value pairs in it, and sends the Fence completion response to the clients. This response also contains the information required to access the shared memory region. Once the clients receive this information, they can open the shared memory region in a read-only mode and access the contents. We enhance the PMI2_KVS_Get function to only perform a lookup from the shared memory region and avoid any socket-based communication with the local PMI server. These changes do not change the existing Put-Fence-Get APIs and are transparent to the MPI library.

A. Shared Memory Storage for Key-Value Pairs

One important design choice here is which data structure to use for the key-value store and how to represent it efficiently in shared memory. The choice of the data structure is dictated by the capabilities and access patterns it needs to support. As the number of Gets performed is very large compared to any other operation and the Get can be called while initiating a message transfer, the Get operation is latency-critical. Further, the PMI standard allows only insertion or updates but no deletion on the key-value store. Also, it would be wasteful to use a data structure that cannot be packed efficiently into a shared memory region or requires an intermediate representation while being created. Based on these requirements, the chosen data structure must offer these characteristics:

- Insertion and update only
- Supports fast lookups
- Ability to grow efficiently
- Can be packed efficiently into shared memory

B. Single Shared Memory Region Based Design (SHMEM-1)

Based on these requirements, a hash table is a natural choice due to its support for fast lookups. PMI Put and Get operations directly translate to Insert and Lookup operations on a hash table. The Fence operation is equivalent to merging multiple hash tables into one. However, representing a hash table in shared memory comes with additional challenges. In a traditional non-shared-memory hash-table which resides inside a process's private address space, the buckets occupy a contiguous memory space. However, the key-value pairs can be placed in arbitrary locations in the memory in a non-contiguous fashion. In addition, offsets must be used instead of pointers as pointers obtained in the context of one process are not valid in other processes.

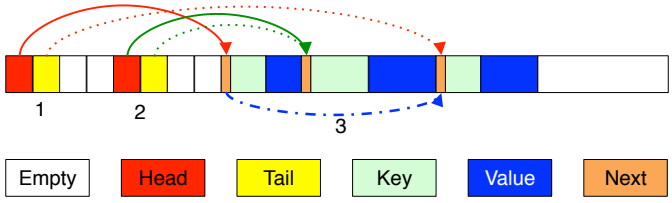


Fig. 4. Representation of a hash table in a single shared memory region

To store the hash table in a shared memory region, a chained hash scheme is used as illustrated in Figure 4. The region contains a number of buckets, which can hold an arbitrary number of key-value pairs (entries). Each bucket contains offsets to the first ($head$) and last ($tail$) entries of the chain. If there is a single key inside a bucket, both the $head$ and the $tail$ point to the same location. Each pair is accompanied by an offset which points to the next entry in the same bucket. Essentially, the region holds a number of logically connected linked-lists of key-value pairs. At initialization, both the $head$ and $tail$ of each bucket are set to an invalid value (e.g. -1). Note that $head$, $tail$, and $next$ are all offsets from the starting address of the shared memory region.

To implement the insert operation on the hash table, strings of varying length need to be allocated inside the shared memory region. Using predefined slots by padding the strings to their maximum length is space-inefficient. To avoid this, a custom memory allocator which works within a shared memory region is used. The allocator keeps track of the free memory available and returns an offset to an empty location of requested size. The allocator does not need to maintain a list of free regions as memory is never deallocated. However, as the number of inserted entries (key-value pairs) increase, the number of buckets must be increased as well to keep the number of collisions low and the allocator needs to deal with that.

If the key-value pairs require relocation during a resize operation, any entries contained in the hash table would be invalidated. In such a scenario, a larger region is allocated and the hash table is rebuilt by migrating all the new key and values into the new region. This operation becomes more expensive as the number of entries increases. As the number of entries is dictated primarily by the number of processes, this approach can negatively impact the performance of the Fence operation at scale.

C. Dual Shared Memory Region Based Design (SHMEM-2)

The key observation that allows us to avoid this limitation associated with a single shared memory region based hash table is that a hash table mapping strings (keys) to strings (values) can be decomposed into two separate entities — the keys and the values themselves, and the mapping which associates the location of a key to the location of a value. Based on this observation, two separate shared memory regions can be used — one to hold the hash table or the mapping and another to store the key/value pairs. A scheme similar to the one described in Section IV-B can be used to connect the buckets in the first region (referred to as $Table$) to the key-value pairs in the other region (referred to as KVS).

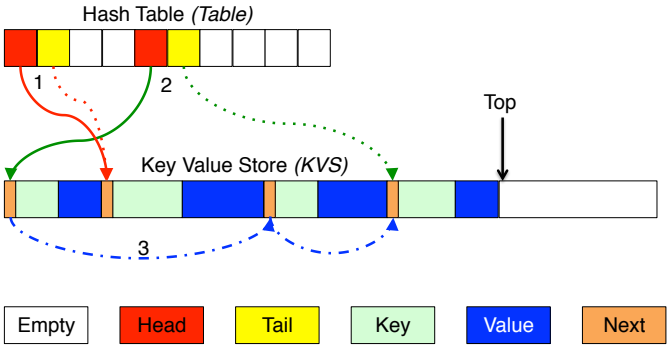


Fig. 5. Representation of a hash table in two shared memory regions

Figure 5 illustrates how a hash table can be represented by two shared memory regions with an example. The bucket marked as 1 contains a single key-value pair and both its $head$ and $tail$ points to the location of that pair in the KVS region. On the other hand, the $head$ and $tail$ of bucket 2 contains the offset to two different key-value pairs in KVS which marks the starting and ending entries of a chain. Entries in the chain point to the next element using the $next$ field (marked as 3). $KVS.top$ stores the offset to the beginning of the unused space inside KVS and is initialized to 0.

We now describe how operations like Insert, Lookup and Resize can be efficiently implemented in a hash table represented in two shared memory regions.

1) Insertion of a New Key-Value Pair: Algorithm 1 shows the procedure to insert a new key-value pair into the hash-table. During insertion, the key and value are copied to KVS, $next$ is initialized to an invalid value (-1), and $KVS.top$ is incremented by the space used by the incoming pair. The key and value strings are NULL terminated and does not require storing their length information or any additional padding. If there is not enough free space left in the KVS to accommodate the incoming key-value pair, the shared memory file must be increased in size. However, since both $Table$ and KVS only contain offsets and not actual addresses, no modification of the data is required.

After the pair is stored in KVS, the key is hashed and the destination bucket is chosen. If the $head$ or $tail$ is uninitialized, the bucket is empty and both $head$ and $tail$ are updated to the offset of the starting address of the newly stored pair. Otherwise, the bucket already contains a list of key-value pairs. The last pair is located by the $tail$ and its $next$ as well as the $tail$ itself are updated to offset of the newly stored pair. Figure 6 illustrates the insertion of 5 key-value pairs into 3 buckets and the states of the two shared memory regions after each insertion.

Algorithm 1: Algorithm to insert a Key-Value Pair

input: $Key, Value, Table, KVS$

$entry \leftarrow KVS.top$;
$entry.next \leftarrow -1$;
$entry.key \leftarrow Key$;
$entry.value \leftarrow Value$;
$KVS.top \leftarrow KVS.top + entry.length$;

$idx \leftarrow Hash(key)$;
$bucket \leftarrow Table[idx]$;
if $bucket.head \leq 0$ **then**
  $bucket.head \leftarrow bucket.tail \leftarrow entry$;
else
  $last \leftarrow KVS[bucket.tail]$;
  $last.next \leftarrow bucket.tail \leftarrow entry$;
end

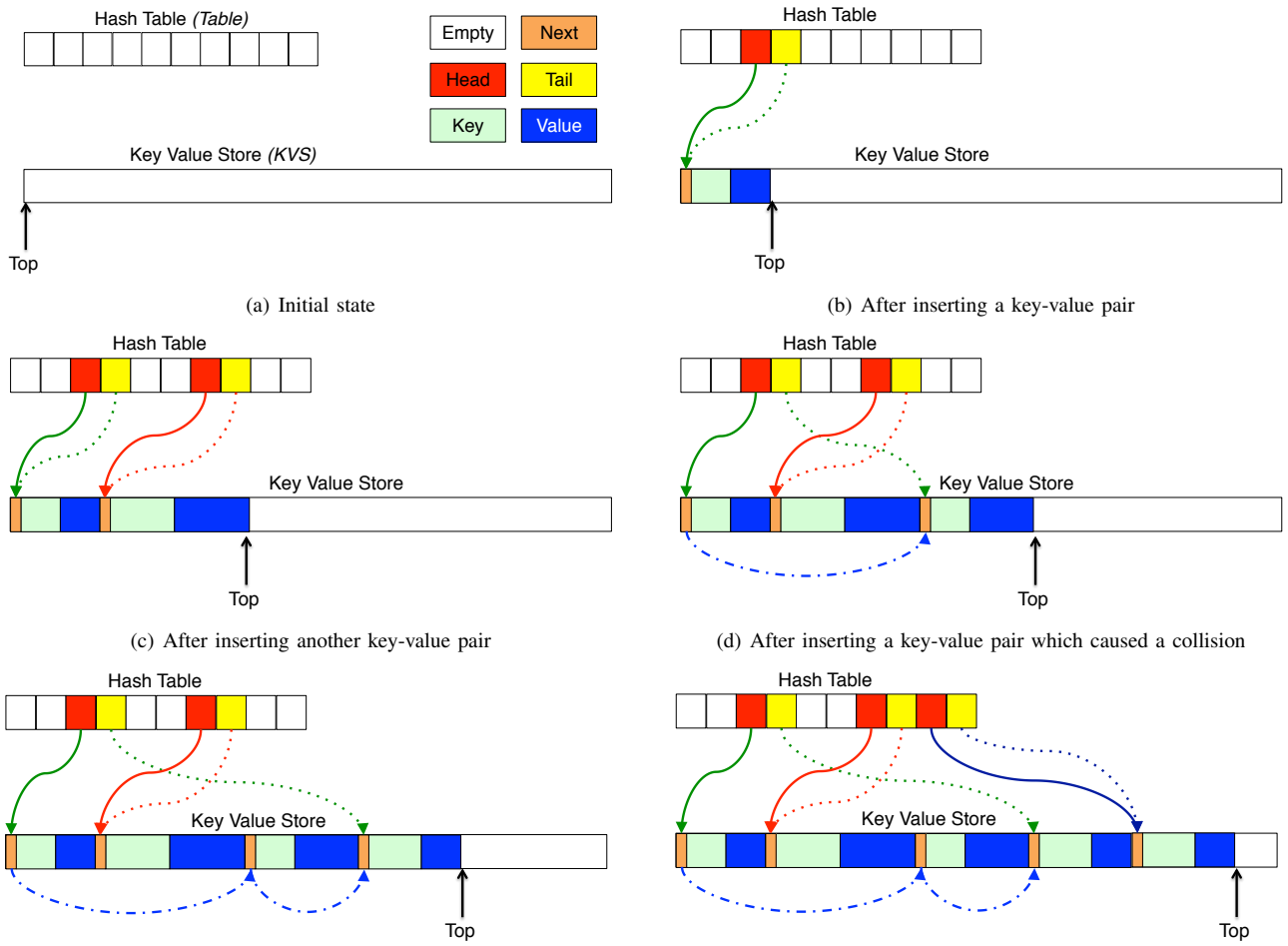
2) Lookup of a Key-Value Pair: The PMI2_KVS_Get function uses the lookup operation shown in Algorithm 2. Based on the key's hash, a bucket is identified and read from the $Table$. The bucket's $head$ points to a list of key-value pairs which is then traversed using the $next$ pointer until the intended pair is located or the end of the list is reached. If the search is successful, a pointer to the value string is returned to the caller; otherwise an error code is returned. Unlike other PMI functions, this requires no communication with the local PMI server.

Algorithm 2: Algorithm to lookup a Key-Value Pair

input : $Key, Table, KVS$
output: $Value$ associated with the Key

$idx \leftarrow Hash(key)$;
$bucket \leftarrow Table[idx]$;
$entry \leftarrow KVS[bucket.head]$;
while $entry \geq 0$ **do**
  if $entry.key = Key$ **then**
    return $entry.value$;
  else
    $entry \leftarrow KVS[entry.next]$;
  end
end
return $NotFoundError$;

It should be noted that the insertions are performed by the server while the lookups are done by the client. For this scheme to work correctly, both the clients and the server must



(a) Initial state

(b) After inserting a key-value pair

(c) After inserting another key-value pair

(d) After inserting a key-value pair which caused a collision

(e) After inserting a key-value pair which caused another collision

(f) After inserting a key-value pair without any collision

Fig. 6. Insertion of Key-Value pairs into shared memory based hash table

use the same hash function. This is not an issue if both the server and client libraries come from the same provider. In case of different providers, one possible solution is to use well known hash functions like md5 or SHA, which can be negotiated during initialization.

3) Resizing the Hash Table: Since the number of key-value pairs is not known beforehand, the hash-table must be able to grow to accommodate new pairs. Growing the KVS region is straight forward, if the available free space ($KVS.size - KVS.top$) is not enough to store the incoming key-value pair, the region is enlarged to twice its current size. If the shared memory is backed by a file, it can be enlarged in place without allocating additional temporary memory and copying the data. Compared to the single region based design where in-place enlargement is impossible, this represents a significant reduction in unnecessary data movement. If the ratio of number of keys and number of buckets grows beyond a threshold, the hash table needs to be re-sized to reduce number of collisions. To achieve this, the $Table$ is re-sized and reinitialized. Then all the pairs in the KVS is iterated through and the corresponding buckets in $Table$ are updated similar to insertion. If the target bucket of a pair is empty, it's $head$ and $tail$ are updated to the pair's offset. If the bucket is not empty, the bucket's tail and the corresponding pair's $next$ is updated to the current pair. These steps are shown in Algorithm 3. Note that since both $Table$ and KVS only store offsets, these operations preserve their validity.

Algorithm 3: Algorithm to re-size the hash table

input : $Table, KVS$

foreach $entry \in KVS$ **do**
  $entry.next \leftarrow -1$;
  $idx \leftarrow Hash(entry.key)$;
  $bucket \leftarrow Table[idx]$;
  if $bucket.head \leq 0$ **then**
    $bucket.head \leftarrow bucket.tail \leftarrow entry$;
  else
    $last \leftarrow KVS[bucket.tail]$;
    $last.next \leftarrow bucket.tail \leftarrow entry$;
  end
end

For clients performing multiple Fence operations, the dynamic resizing of the tables does not cause any additional issues. Since a client process cannot perform a Put or a Get operation while a Fence is in progress, the client can close the previously opened shared memory regions while sending the Fence request. Since the Fence response from the server always contains the updated information about the shared memory regions, the client can reopen them and perform lookup operations.

V. Shared Memory Based Allgather

In our earlier work [6] we proposed and designed a set of new PMI functions including PMIX_Allgather. This is a collective function where each process provides an input value and an output buffer. The rank of the source process is used as an implicit key and all the values are gathered and broadcasted to all the slurmds. The slurmds then sort the values by their source rank and copies them into the user provided buffer. The values are padded such that the value from rank r will be available at offset $r * MaxLength$ where $MaxLength$ is known beforehand. Once the operation is completed the clients can directly access the values from the output buffer. The Allgather function is optimized for the common scenario where data movement is symmetric. Compared to Fence, Allgather leads to less network traffic and scales better in terms of performance and memory usage.

The changes required for shared memory based Allgather is simpler compared to Fence. Due to the lack of explicit keys in Allgather, the output contains only an array of strings instead of a hash-table. This can be efficiently represented in a single shared memory region. Consequently, the clients only need to open a single shared memory region. Also, the clients no longer need to allocate a buffer to hold all the collected values. The hybrid communication scheme described in Section III-B is used for Allgather. Once PMIX_Allgather is completed, the server creates a shared memory region containing all the values sorted by the source rank and sends the name and size of shared memory region to the clients over the socket based channel. The client then opens the shared memory region and returns the starting address of the shared memory region to the caller process. The caller process (MPI library) can then directly access the values from the shared memory without making more PMI calls, by using the source rank as the index.

VI. Discussion

A. Backward Compatibility

Interoperability with the current standard is an important benefit and would ease the adoption of the proposed designs in HPC systems. We discuss any changes required in both the PMI client API and the wire protocol.

1) API Level Compatibility: One major benefit of the proposed design is that it does not change the PMI API. Thus, existing users of PMI such as MPI libraries would continue to work well and benefit from the faster lookup performance. To achieve the reduced memory footprint, the libraries would require minor changes. The only change required for an MPI library using PMIX_Allgather is to use the returned buffer instead of allocating a new output buffer. For libraries using Put-Fence-Get, the results of the Get operation should be used directly instead of being stored in an array.

2) Wire Protocol Compatibility: There is another aspect of backward compatibility - the wire protocol used for the socket based communication between the server and the client. A server with shared memory support should work well with a client without said support and vice versa. We need to consider these two cases:

New Client, Old Server: A server with shared memory support sends additional fields related to shared memory in the response to a Fence command. If a client detects a Fence response without said fields, it can conclude that the server does not support shared memory and transparently fall back to the default mode of socket based communication.

Old Client, New Server: In the reversed scenario where the server supports shared memory but the client does not, the client can simply ignore the additional fields. For such clients, a call to PMI Get would result in a lookup request to the server and the server can fulfill the request by accessing the shared memory. The only caveat is that if the client treats the additional fields as an error condition, it will likely mark the Fence call as a failure and abort. However, most prevalent PMI clients like SLURM and mpirun_rsh do not suffer from this issue.

Another possibility here is to add a completely new PMI function (for example, PMIX_SHMEM_Init) which must be explicitly called after PMI2_Init to enable the shared memory mode. However, this approach requires changes to the client-side API and by the end user (MPI library).

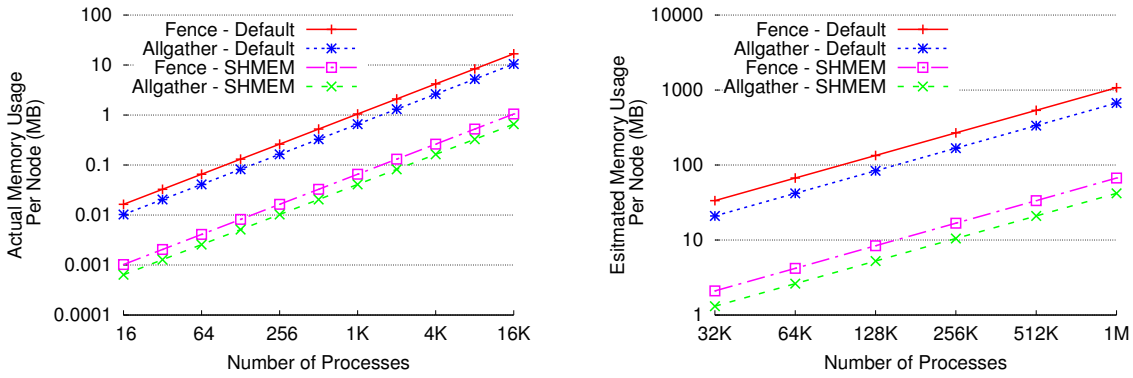
3) Compatibility with Non-blocking PMI Extensions: Non-blocking PMI functions like PMIX_KVS_Ifence and PMIX_Iallgather were proposed to allow overlap of PMI communication with MPI initialization or application computation [6]. The shared memory based designs are fully compatible with the non-blocking PMI collectives. The clients close the open shared memory regions while calling Ifence or Iallgather. When the clients call PMIX_Wait to check for completion of the outstanding PMI operations, it needs to process the response similar to the proposed design and open the shared memory regions. Essentially, a Fence or an Allgather can be transparently replaced with a Fence followed by Wait or an Allgather followed by Wait respectively.

VII. Experimental Evaluation

In this section, we describe the experimental setup used to conduct micro-benchmark and application experiments to evaluate the improvement from the proposed extensions. An in-depth analysis of the results is also provided to correlate design motivations and observed behavior.

A. 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



(a) Actual Memory usage up to 16K processes

(b) Estimated Memory usage up to 1M processes

Fig. 7. Comparison of memory used for network endpoint addresses by MPI processes

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-14.11.4 and MVAPICH2-2.1 compiled with Intel icc-13.0.1 were used to evaluate the proposed designs. For the Get level latency results shown in Figure 1 and Figure 8(a), we used a set of nodes on Stampede with identical setup except four sockets (32 cores) per node and 1 TB RAM. Since there were only 16 such nodes available, they were not used for the large scale experiments. All other numbers reported were taken in fully subscribed mode with 16 processes per node.

B. Impact on Memory Usage

With the existing design *(Default)*, PMI users like MPI libraries are required to replicate the key-value store into their private address space. Since each process maintains a copy of the data, there are $PPN + 1$ copies of the data in a node with PPN MPI processes. Figure 7(a) and Figure 7(b) show the actual and estimated memory consumption per node by MPI processes for storing this information. For MVAPICH2 running with 1 million MPI processes and 16 processes per node, that translates to approximately 960 MB on each node. With the Allgather based design, the keys are not required which slightly reduces the memory consumption to approximately 800 MB.

With the new design *(SHMEM)*, the clients do not copy the data from the server. As a result, only a single copy of the data is stored on a node irrespective of the number of processes in the job or on the node. Consequently, memory consumption for storing the information exchanged through PMI is reduced by a factor of PPN. As shown in Figure 7(b), with 1 million processes, this results in only 64 MB for the Fence and only 40 MB for the Allgather based design. The reduction in memory usage is even higher with larger scale jobs or on nodes with larger number of cores. Both the shared memory designs based on single and dual regions show nearly identical memory consumption.

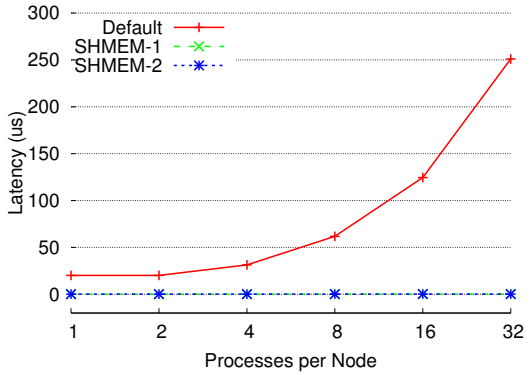
C. Performance of PMI2_KVS_Get

In the existing client-server based communication scheme, the server serializes all PMI operations. As a result, multiple clients cannot access the key-value store in a truly parallel fashion. With the proposed shared memory based design, no communication is required during the PMI2_KVS_Get operation. Each client can independently look up any key-value pair from the shared memory region. Consequently, the Get operation in this scheme requires no synchronization and shows constant latency with arbitrary number of client processes.

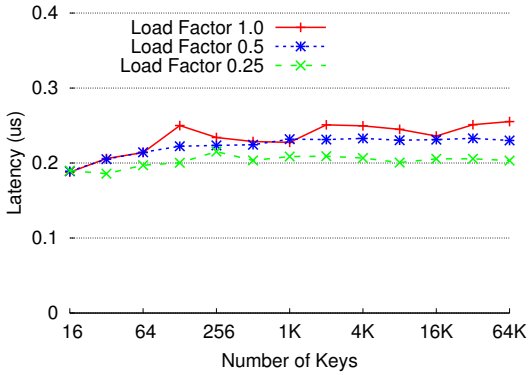
To compare the performance of PMI2_KVS_Get with and without shared memory designs, each client performs 10,000 Gets of a randomly selected key from the key value store and the average latency is reported. We show the average of 10 runs in Figure 8(a). Without shared memory *(Default)*, the latency of a single Get is 20μs for a single client but increases to 250μs for 32 client processes per node. This is expected as each Get request suffers from queuing delay and needs to wait longer before being processed by the server. Since Get is a local operation and does not involve inter-node communication, the average latency does not depend on the number of nodes.

With the dual shared memory based design *(SHMEM-2)*, the Get operation shows a near-constant latency of 0.25μs independent of the number of client processes. This represents a benefit of 1,000 times over the current design. We also evaluated the single shared memory based design *(SHMEM-1)*. Although this design requires access to a single shared memory region, Get latency shows less than 10% improvement compared to the dual region design. We believe this is due to the fact that even if the buckets and the key-value pairs are in the same region, they are not located closely enough to noticeably reduce the number of cache misses.

1) Impact of Load Factor on Get Latency: While designing a hash-table, one important design decision is the choice of the load factor. The load factor of a hash table is defined as $\frac{Number\ of\ Keys}{Number\ of\ Buckets}$. It is easy to see that with a higher load factor, the probability of collisions (multiple keys landing in the same bucket) increases. Consequently, the average time to lookup a key increases with a larger load factor. While a smaller load factor can decrease the average lookup latency, it

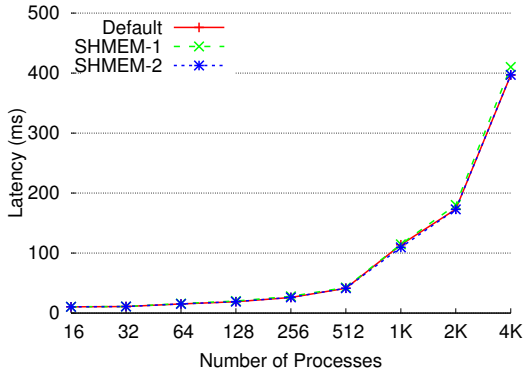


(a) Performance of default and proposed PMI2_KVS_Get

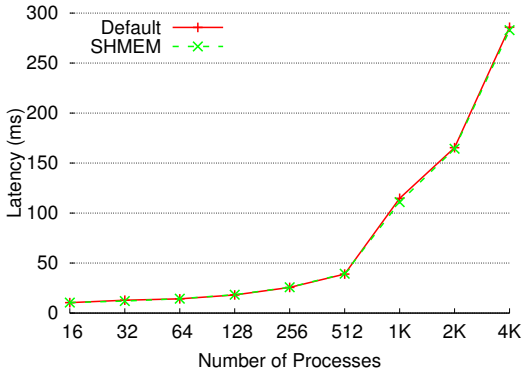


(b) Performance of proposed PMI2_KVS_Get with different load factors

Fig. 8. Performance comparison of the default and proposed PMI2_KVS_Get



(a) PMI2_KVS_Fence



(b) PMIX_Allgather

Fig. 9. Overhead of the proposed designs on PMI2_KVS_Fence and PMIX_Allgather

also requires more memory. This time-memory trade-off must be considered while designing such a scheme.

Figure 8(b) shows the effect of different load factors on the latency of dual shared memory region based Get operation. The difference is minor but noticeable with large number of keys. The average latency observed is improved by 9% and 20% respectively with load factor of 0.5 and 0.25. Load factors above 1 are possible but not recommended [8] as it can lead to a large number of collisions and poor performance. Note that with the dual region design, choosing a smaller load factor only increases the memory usage for the shared memory region which holds the hash table. The size of memory region holding the key-value pairs is not affected. We used a load factor of 1 in all other experiments.

D. Overhead of the Proposed Shared Memory Based Designs

In the proposed designs, both the server and the client needs to perform a few additional tasks like opening shared memory regions. To measure any overhead introduced by these operations, we compare the average of 1,000 iterations of PMI2_KVS_Fence and PMIX_Allgather at different scales with 16 processes per node and 10 runs each. Each process performed a single Put with a 16 Byte key and a 32 Byte value before the Fence. As shown in Figure 9(a), the performance of the dual shared memory region based *(SHMEM-2)* Fence operation is identical to the default design and there is no noticeable overhead. The single region based design *(SHMEM-1)*, however, shows a small overhead due to the extra copy operations.

We compare the existing Allgather operation *(Default)* with the shared memory based design *(SHMEM)* in Figure 9(b). The allgather was performed with a 32 byte value. At small and medium scale, time taken by the shared memory based design is same as the default design. However, at large scale the shared memory based design performs slightly better. This is explained by the fact that in the output buffer is copied PPN times over local socket from the server to the client in the default scheme. But in the shared memory based design, this copying is avoided and only the location and size of the relevant shared memory region is passed to the client.

VIII. Related Work

There has been significant work in the area of improving scalability of MPI libraries. Balaji et al [9] investigated the scalability of the MPI specification and implementations at large scale.

The design and implementation of the PMI interface was described by Balaji et al [1]. In our earlier work [10] we explored the use of high-speed networks like InfiniBand to speed up the data exchange in PMI and proposed a new PMI collective called PMIX_Ring to minimize data movement at initialization. We further proposed non-blocking extensions

to the PMI interface in [6] where we demonstrated the use of split-phase Fence and Allgather routines to achieve near-constant startup time for MPI applications. Some of these designs are currently available in popular process managers including SLURM and mpirun_rsh. Similar designs have been proposed by other groups, such as the PMIx project [11] as well.

Design and implementation of efficient and fast hash tables have been a significant area of research for a long time. A survey of hash table techniques appears in Maurer et al [12]. Dynamic resizing of hash tables was first proposed by Larson et al [8]. Litwin et al introduced and evaluated the linear hashing scheme in [13]. Bennett et al [14] shows a methods of reorganizing collisions in buckets to improve performance. However, there has been relatively little work on representing hash tables in shared memory. A few instances of lock-based and lock-free implementations of hash tables on shared memory can be found online [15], [16], [17]. To the best of our knowledge this is the first such study of using multiple shared memory regions for efficient storage and lookup of hash table.

IX. Conclusion

In this paper, we explored different aspects of a shared memory based design for the PMI protocol. We considered different channels and communication schemes between the PMI clients and the server. We also explored different designs for efficiently exposing the PMI key-value store through single and multiple shared memory regions and evaluated them in terms of different metrics like performance and security. We designed a hash table that segregates its buckets and the key-value pairs into two separate shared memory regions to allow fast lookups and efficient resizing. This hash table was used to implement PMI's global key-value store and allow direct read access from the clients. With this design, we were able to reduce memory usage by avoiding the data duplication between the PMI server and the clients. This allows PMI users (e.g. MPI libraries) to avoid copying the values to their private address space, which can reduce the memory consumption for a node by a factor of $Processes\ per\ Node$. We also improved performance of PMI Get by eliminating the inherent serialization present in the existing current client-server model. Our evaluations with SLURM and MVAPICH2 showed projected savings of nearly 1 GB in memory usage per node with 1 million MPI processes and 16 processes per node. In addition, the latency of PMI Get with 32 processes per node was reduced from 250μs to 0.25μs, representing an improvement of 1,000 times compared to the default design. The proposed designs will be available with the upcoming MVAPICH2 release.

Acknowledgments

We would like to thank Adam Moody and Akshay Venkatesh for their valuable input and feedback. This research is supported in part by National Science Foundation grants #OCI-1148371, #CCF-1213084, #CNS-1419123, #IIS-1447804, and #ACI-1450440.

References

[1] 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*. Springer, 2010, pp. 31–41.

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

[3] Argonne National Laboratory, "Hydra Process Management Framework," https://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework.

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

[5] 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.

[6] S. Chakraborty, H. Subramoni, A. Moody, A. Venkatesh, J. Perkins, and D. K. Panda, "Non-blocking PMI Extensions for Fast MPI Startup," in *Proceedings of the Int'l Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2015)*, 2015.

[7] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "TOP 500 Supercomputer Sites," http://www.top500.org.

[8] P.-A. Larson, "Dynamic Hash Tables," *Communications of the ACM*, vol. 31, no. 4, pp. 446–457, 1988.

[9] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on a Million Processors," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2009, pp. 20–30.

[10] S. Chakraborty, H. Subramoni, J. Perkins, A. Moody, M. Arnold, and D. K. Panda, "PMI Extensions for Scalable MPI Startup," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 21:21–21:26. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642780

[11] "PMI Exascale (PMIx)," https://www.open-mpi.org/projects/pmix/.

[12] W. D. Maurer and T. G. Lewis, "Hash Table Methods," *ACM Computing Surveys (CSUR)*, vol. 7, no. 1, pp. 5–19, 1975.

[13] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing." in *VLDB*, vol. 80, 1980, pp. 1–3.

[14] J. G. Bennett and R. Krishnaswamy, "Reorganization of Collisions in a Hash Bucket of a Hash Table to Improve System Performance," Apr. 18 2000, uS Patent 6,052,697.

[15] S. Hardy-Francis, "SharedHashFile: Share Hash Tables Stored In Memory Mapped Files Between Arbitrary Processes & Threads," https://github.com/simonhf/sharedhashfile, 2012.

[16] V. Handa, "Shared Memory Hash Table," http://vhanda.in/blog/2012/07/shared-memory-hash-table/, 2012.

[17] J. Preshing, "The World's Simplest Lock-Free Hash Table," http://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/, 2012.