Non-blocking PMI Extensions for Fast MPI Startup

S. Chakraborty[1], H. Subramoni[1], A. Moody[2], A. Venkatesh[1], J. Perkins[1], and D. K. Panda[1]

[1] *Department of Computer Science and Engineering, The Ohio State University*
{chakrabs, subramon, akshay, perkinjo, panda}@cse.ohio-state.edu

[2] *Lawrence Livermore National Laboratory, Livermore, California*
{moody20}@llnl.gov

***Abstract*—An efficient implementation of the Process Management Interface (PMI) is crucial to enable fast startup of MPI jobs. We propose three extensions to the PMI specification: a blocking allgather collective (PMIX_Allgather), a non-blocking allgather collective (PMIX_Iallgather), and a non-blocking fence (PMIX_KVS_Ifence). We design and evaluate several PMI implementations to demonstrate how such extensions reduce MPI startup cost. In particular, when sufficient work can be overlapped, these extensions allow for a constant initialization cost of MPI jobs at different core counts. At 16,384 cores, the designs lead to a speedup of 2.88 times over the state-of-the-art startup schemes.**

***Keywords*—Process Management Interface; Job Launch; Non-blocking; InfiniBand**

I. Introduction and Motivation

As high-performance computing clusters continue to increase in size to meet increasing computational needs, fast and scalable startup of parallel applications becomes more important. Reducing startup cost can save developers hours of time while developing and debugging an application, which often requires frequent restarts of the application. Other scenarios that involve running many large-scale, short-lived jobs in quick succession include regression testing of an application or middleware and testing of a newly configured system. In such cases, reducing startup time provides significant benefits in terms of system efficiency.

The Message Passing Interface (MPI) [1] is the de-facto standard for writing high-performance parallel applications. A 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, many job launchers implement a standard "out-of-band" communication infrastructure known as the Process Management Interface (PMI) [2]. PMI is typically implemented as a client-server library with the job launcher 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. *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*.

However, current implementations of PMI scale poorly on today's largest systems. 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). For simplicity, we only show the time spent executing a PMI exchange, and the time spent in other initialization work, e.g., memory registration, setting up shared memory channels, etc. 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. As the job size increases, the PMI *Put-Fence-Get* sequence takes a larger portion of time, and it grows to consume the majority of the startup time at larger scales.

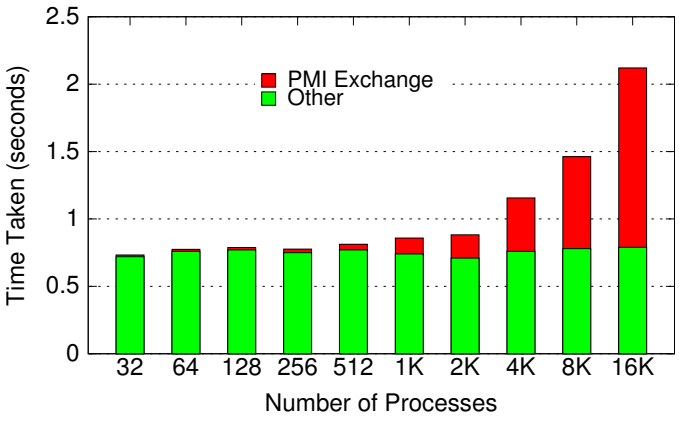

Fig. 1. Breakdown of time spent in MPI_Init inside MVAPICH2

In previous work [3], we examined SLURM's [4] implementation of PMI, and we found that the *Fence* operation is responsible for most of the cost at scale. We again consider SLURM as the baseline throughout this work, referencing SLURM version 2.6.5. In SLURM, *Fence* executes an allgather operation implemented as a hierarchical gather followed by a k-nomial broadcast over the *slurmd* and *srun* processes. Following up on this work, we did a more in-depth analysis of the time taken by the allgather operation in *Fence*. Our

*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.

TABLE I. VOLUME OF DATA TRANSFERRED AND TIME TAKEN IN DIFFERENT PHASES OF *Fence*

Number of Processes in Job	**Gather phase** process → *slurmd* → *srun*					**Broadcast phase** *srun* → *slurmd* → process		Time for Data Processing (ms)	Total Time in Fence (ms)
	Value Size (Bytes)	Key Size (Bytes)	Overhead (Bytes)	Total (Bytes)	Time (ms)	Data (KB)	Time (ms)		
4,096	18	9	8	35	236	143	218	4.28	396
8,192	18	9	8	35	284	286	386	8.74	685

analysis found that most of the time is taken by the broadcast from *srun* to the *slurmd*s.

In Table I, we show the amount of data transferred and the time taken for different phases of the *Fence* operation. The sizes of the keys and values given in the table are representative of the kind of PMI communication that happens in the startup phase in the MVAPICH2 MPI library [5]. For each key or value transferred, the PMI implementation in SLURM adds an overhead of 4 bytes to the message. This is because SLURM's PMI supports key-value pairs of arbitrary length as well as asymmetric data movement. Thus for a key of size 18 bytes and a value of size 9 bytes, each individual process sends a message of size 35 bytes to the local *slurmd*. The local *slurmd* aggregates such messages from all children (both MPI processes and other *slurmd*s) and sends it up to its parent. This process continues until the message reaches *srun*, at which point *srun* aggregates all messages and broadcasts the result to all *slurmd*s either directly or through the SLURM tree depending on the number of nodes in the job.

As shown in Table I, the volume of data transferred and the time for the data transfer in the broadcast phase increase linearly with the number of processes in the job. Note that the volume of data transferred in the gather phase is in bytes and the volume of data transferred in the broadcast phase is in KiloBytes (KB). From this assessment, it is clear that reducing the amount of total data exchanged would lead to better performance.

The existing *Put-Fence-Get* semantics in PMI-2 has another major drawback. The blocking nature of *Fence* forces the users of PMI such as high-performance middleware like MPI to wait idly while the operation completes. If the application has some operations that do not depend on the values fetched through *Get*, it can potentially overlap these operations while the *Fence* is performed in the background. Although the current model is simple to use and adequate at small scale, for large numbers of processes, it leads to a lot of wasted cycles as shown in Figure 1.

These issues lead us to the following broad challenge — **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 take up this challenge and propose three extensions to the PMI specification to address them: a blocking allgather collective (PMIX_Allgather), a non-blocking allgather collective (PMIX_Iallgather), and a non-blocking *Fence* (PMIX_KVS_Ifence).

The first extension allows efficient exchange of values between different processes. The second and third extensions enable one to overlap PMI communication with other non-communication related activities that applications and high performance middleware perform during startup. Another group has also recently proposed the addition of non-blocking PMI operations [6]. In our work, we propose a different set of extensions and illustrate their value.

By employing these extensions, we implement several PMI designs that significantly reduce MPI startup costs.

II. CONTRIBUTIONS

To summarize, this paper makes the following contributions:

- Propose, design and implement the *PMIX_Allgather* API to reduce the amount of data being transferred as well as avoid the data processing overheads in existing PMI designs
- Propose, design and implement non-blocking versions of two PMI APIs - *PMIX_KVS_Ifence* and *PMIX_Iallgather* to allow overlap of PMI communication and other activities application / middleware need to perform at startup
- Evaluate the benefits the new APIs have on performance at the microbenchmark level using PMI and MPI microbenchmarks and at application level using NAS parallel benchmarks

We design and evaluate several PMI implementations to demonstrate how such extensions reduce MPI startup cost. In particular, when sufficient work can be overlapped, these extensions allow for a constant initialization cost of MPI jobs at different core counts. At 16,384 cores, the designs lead to a speedup of 2.88 times over the state-of-the-art startup schemes.

III. BACKGROUND

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

A. SLURM

SLURM [4] (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 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 2.

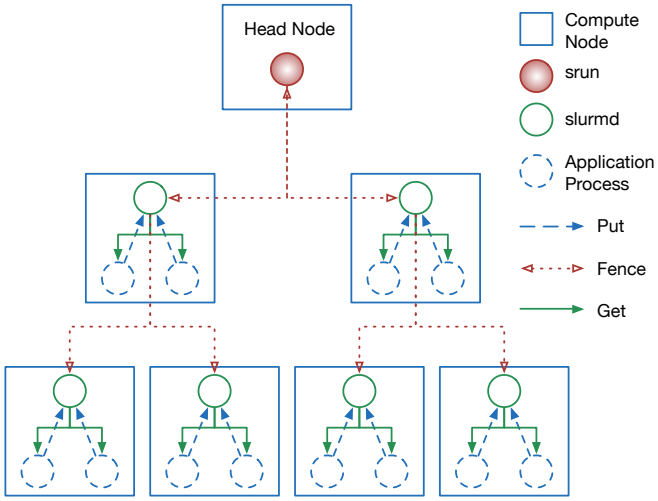

Fig. 2. Hierarchical communication scheme in SLURM

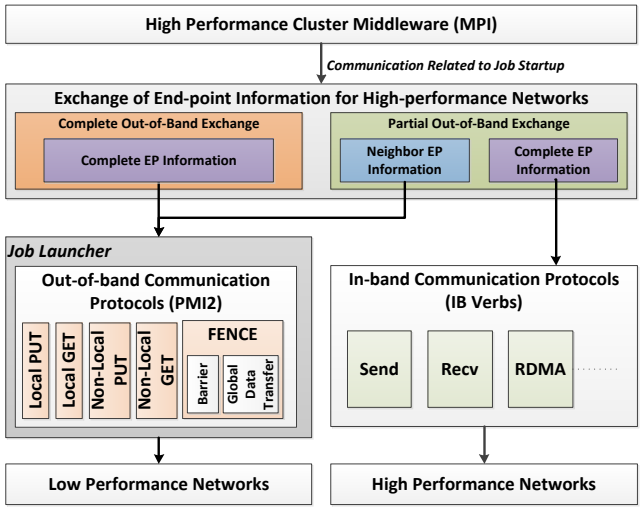

Fig. 3. State-of-the-art techniques for MPI job launch

B. MPI over InfiniBand and High Speed Ethernet

The Message Passing Interface (MPI) [1] is one of the most popular programming models for writing parallel applications in high-performance computing. MPI libraries provide optimized communication methods for a parallel computing job. In particular, several convenient point-to-point and collective communication operations are provided. High performance MPI implementations are closely tied to the dynamics of underlying high performance networks such as InfiniBand [7] and aim to achieve the best communication performance on the given interconnect. In this paper, we use MVAPICH2 [5] for our evaluations. However, our observations in this context are quite general and they can be applied to other high performance MPI libraries.

C. Current MPI Job Launch Techniques

Figure 3 represents the current state-of-the-art techniques available for launching MPI jobs on large scale supercomputing systems. Existing job launch techniques can broadly be classified into 'completely out-of-band' (left side of Figure 3) and 'partially out-of-band' (right side of Figure 3) depending on the type of communication channel being used for startup based communication. In the 'completely out-of-band' scheme, all PMI communication is routed through the out-of-band channel (typically TCP/IP). In the 'partially out-of-band' scheme, a small portion of the PMI communication happens on the out-of-band channel while the bulk of the data is transferred over the high-speed network. In our previous work [3], we designed the 'partially out-of-band' mode of job startup by proposing new extensions to the PMI standard. In this paper we propose a different set of non-blocking extensions that exclusively use the out-of-band channel.

IV. Proposed Extensions

A. PMIX_Allgather

A common use of PMI, especially when starting small or medium scale MPI jobs, is to have each MPI process *Put* its network address as a value providing its rank as the key. Then after a *Fence*, each process *Gets* the address for every other process. As an optimization for this use case, we propose PMIX_Allgather, abbreviated as *Allgather*, which combines the three separate operations of *Put*, *Fence*, and *Get* into a single collective call. The signature of the function is as follows:

```
int PMIX_Allgather (
  const char value[],
  void *buffer );
```

The caller provides a NULL terminated UTF-8 string as input and an output buffer of size $(Number\ of\ Processes * Maximum\ Allowed\ Length\ of\ Value)$. Upon completion, the output buffer holds the values provided by all processes ordered by PMI rank. All strings in the output buffer are NULL padded to the maximum allowed length, which enables a process to look up the value for a given rank by calculating an offset $(rank * MaxLength)$ and directly accessing the buffer.

As an optimization, an additional input parameter that holds the maximum length of the value strings can be introduced to reduce the size of the buffer the caller has to provide.

B. PMIX_Request

Use of non-blocking collectives to overlap communication with computation is a well-known concept in MPI [8–19]. We propose similar constructs for the PMI standard.

To support non-blocking operations, we first introduce a new type called PMIX_Request, which is an opaque handle to an outstanding non-blocking operation. The proposed non-blocking functions presented later initiate a non-blocking operation and return a request handle that is later used to wait for the completion of the operation. The lifetime of the associated request handle is managed by the PMI implementation.

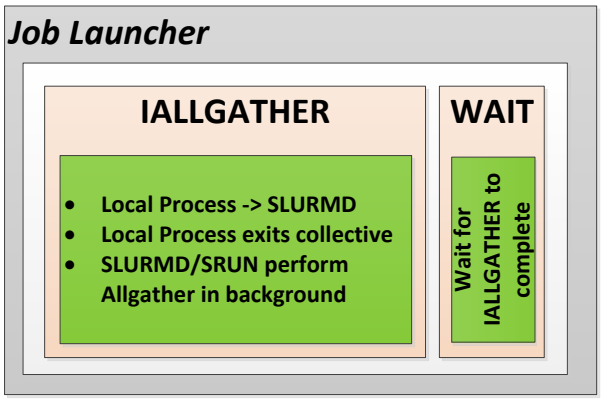

Fig. 4. High-level overview of PMIX_Iallgather

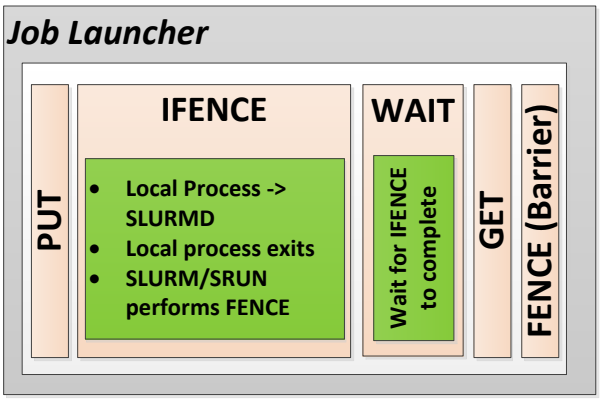

Fig. 5. High-level overview of PMIX_Ifence

C. PMIX_Wait

Each request returned by a non-blocking function must be completed by a call to PMIX_Wait, referred to as *Wait*, which we define with the following signature:

```
int PMIX_Wait (
  PMIX_Request request );
```

The *Wait* function takes a request handle obtained from an earlier call to a non-blocking function. If the operation has completed, the function deallocates the request handle and returns. Otherwise, the calling process is blocked until the associated operation completes.

D. PMIX_Iallgather

We propose a non-blocking variant of *Allgather*, called PMIX_Iallgather that we refer to as *Iallgather*, which initiates the operation and returns immediately. It takes the same parameters as PMIX_Allgather with the addition of a request parameter:

```
int PMIX_Iallgather (
  const char value[],
  void *buffer,
  PMIX_Request *request_ptr );
```

Once initiated, the caller must not access the output buffer until the associated request is completed with a call to *Wait*. Once the *Wait* call returns, the caller is allowed to reuse or dispose of the buffer. Figure 4 shows the high-level design of PMIX_Iallgather.

E. PMIX_KVS_Ifence

Fence guarantees that data for all previously performed *Put*s will be available to subsequent *Get* operations. However, the *Get*s are often not required immediately after the *Fence*. If *Fence* is implemented as a blocking operation, client processes cannot proceed while the *Fence* is in progress. In SLURM, the *Fence* is progressed by the *slurmd*s without requiring participation from the client processes. This provides an opportunity for overlap where the clients could initiate the *Fence* operation and return immediately allowing the *slurmd*s to progress the *Fence* operation in the background.

We propose a new function called PMIX_KVS_Ifence, referred to as *Ifence*, that leverages this. The function signature is:

```
int PMIX_KVS_Ifence (
  PMIX_Request *request_ptr );
```

Once initiated, the caller must not execute *Get* calls until the associated request is completed with a call to *Wait*. The high-level design of PMIX_Ifence is depicted in Figure 5.

Note that some PMI implementations may emulate the behavior of *Ifence* with their *Fence* and *Get* methods without breaking existing PMI semantics. A *Fence* can initiate a non-blocking operation and return immediately, similar to the proposed *Ifence*, and then the subsequent *Get* can block until the non-blocking *Fence* operation completes. The difference is that *Ifence* is guaranteed to be non-blocking whereas *Fence* may block, thus *Ifence* provides a stronger bound on performance to the user.

V. Design of Proposed PMI APIs

In this section, we describe and evaluate different designs for exchange of information while maintaining consistency and ease of use offered by PMI semantics.

A. Limitations of PMI2_KVS_Fence

The standard form of the *Fence* operation serves the purpose of synchronizing the key-value stores at each node to reach a consistent state across nodes. As shown in Table I, the SLURM implementation of *Fence* involves two major costs:

1. A gather operation to *srun* of all key-value pairs from each *slurmd* followed by broadcast of a *cumulative string* representing the concatenation of all key-value pairs (synchronizing operation). This can be expressed as a gather phase:

$$\forall i : 0 < i < N_{put} : srun \leftarrow (key, value)_i$$

followed by a broadcast phase:

$$\forall i : 0 < i < N_{put} :$$
$$slurmd \leftarrow (key, value)_0, \cdots, (key, value)_{N_{put}-1}$$

where N_{put} is the total number of *Put*s performed after the end of the previous *Fence* and before the current *Fence*.

N_{put} is independent of number of processes as each process can perform an arbitrary number of *Put*s before invoking *Fence*. Figure 6 illustrates the format of the packed data transferred between different processes and *slurmd*s in a single *Fence*. In the gather phase, *slurmd*s collect data from local processes and children *slurmd*s and propagate it upwards. In the broadcast phase, each *slurmd* receives the *cumulative string* containing all key-value pairs from their parent *slurmd* or *srun*. The semantics of *Fence* necessitates the inclusion of the "key" field in all of the data transfers. The "key" is an arbitrary-length string, so the format also records a "length" field to enable proper parsing at the destination process.

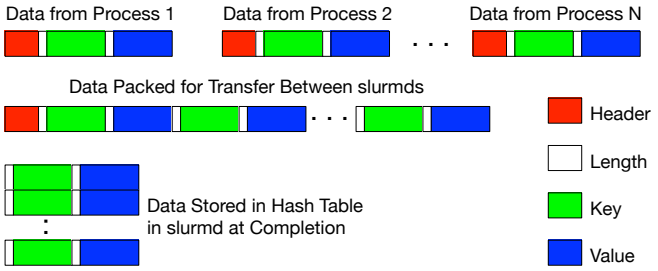

Fig. 6. Data packing format used in PMI2_KVS_Fence

2. The second step is the creation of a hash-table, which involves extracting and storing key-value pairs from the *cumulative string* into a hash table by the *slurmd*s. The *cumulative string*, which represents the set $\{(key, value)_0, \cdots, (key, value)_{N-1}\}$, may not be sorted when it is received by the *slurmd*s. Both *key* and *value* in a pair are represented as strings. The *slurmd* iterates over the *cumulative string*, reads each key-value pair, and inserts it into a chained hash table. This can be visualized as:

$$\forall\ (key, value)\ \epsilon\ cumulative\ string :$$
$$HashTable(h(key)) \leftarrow value$$

where h is the applied hash function.

Although an insertion into a hash table is an $O(1)$ operation, collisions, number of memory allocations and resizing increase with large number of keys and impose a significant overhead. As a result the hash-table creation imposes overheads that grow linearly with size of the MPI job as $O(n)$ and involves a large constant k_{hash}. While this is not optimal, maintaining a hash table is required to support fast *Get* operations.

Having identified the above two steps as the most time consuming steps in the PMI2_KVS_Fence operation, we also note that it is not possible to avoid these overheads while supporting arbitrary keys and allowing different processes to *Put* different numbers of key-value pairs. However, in practice the PMI based communication MPI libraries perform is more limited. For example in MVAPICH2, all processes make a single key-value pair available for others, and each process queries for the values provided by every other processes. This communication pattern is symmetric, and each process's rank is associated with only one value. This observation motivates our proposal for PMIX_Allgather, which executes this communication pattern with lower overhead.

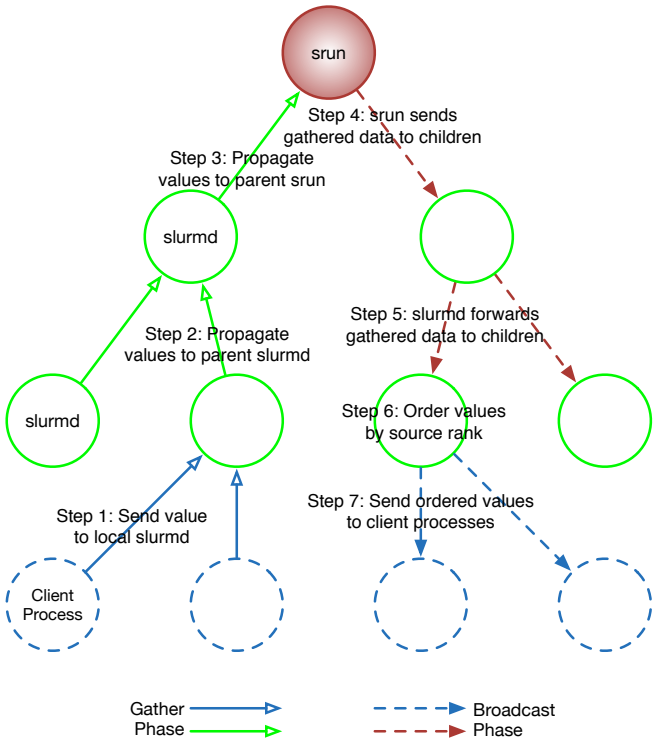

Fig. 7. Different steps involved in PMIX_Allgather

B. Design of PMIX_Allgather

Figure 7 illustrates the implementation details of our PMIX_Allgather design in SLURM. In PMIX_Allgather, we combine the three separate operations of *Put*, *Fence*, and *Get* into a single collective call. The *slurmd* receives one value from each local client process, tags the value with the rank of the client, and writes the value into a local buffer. After collecting the value from each client, it forwards the buffer up the SLURM tree to the *srun* process. The *srun* process then broadcasts the full set of values back down the tree. The gather and broadcast steps thus resemble the existing implementation of *Fence*. However, the number of rank-value pairs in this case is equal to the number of processes. Therefore the gather phase is equivalent to:

$$\forall i : 0 < i < N_{procs} : srun \leftarrow (i, value_i)$$

where N_{procs} is the number of processes.

In PMIX_Allgather, we address the two major bottlenecks observed in PMIX_KVS_Fence. First, the use of *integers* for *keys* reduces the buffer overhead that results from using string representation of MPI ranks. This improvement is achieved when $sizeof(rank\ as\ string) > sizeof(rank\ as\ integer)$, specifically when $rank \geq 1000$ if the rank is treated as a 32-bit integer. However, as noted in Section I, SLURM adds a 4 Byte overhead to any string, so the integer representation is always more compact. In practice, the keys are generally padded and prefixed, which improves this even further. This results in time reduction for both the gather and the broadcast phases of the allgather due to reduced message size.

Secondly, instead of creating the hash table, each *slurmd* allocates an array to hold N_{procs} values. Once the *slurmd* receives the *cumulative string* from *srun*, it goes through the buffer and and copies each value into the array using its rank as the index. This operation can be represented as:

$$\forall\ (rank, value)\ \epsilon\ cumulative\ string$$
$$Array(rank) \leftarrow value$$

This only requires one memory copy operation for each rank-value pair and the result is an array containing N_{procs} values sorted by their rank. This processing can be performed by either *srun* or *slurmd*. However, it is disadvantageous to do it in *srun* as the result array has all strings padded and hence would cause unnecessary data to be broadcast. Since all *slurmd*s can perform this operation in parallel, it does not add extra latency to the operation.

For accessing the keys, the clients directly read from the array as the value from the process is available at offset $rank * Maximum\ Allowed\ Length$. This gives some additional improvement over the SLURM implementation of *Fence* where each *Get* operation incurs the overhead of communication with the *slurmd* process and a hash-table lookup.

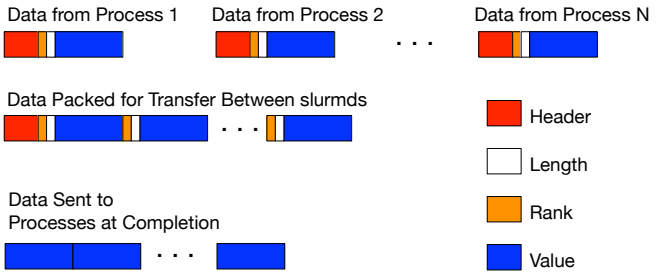

Fig. 8. Data packing format used in PMIX_Allgather

Figure 8 illustrates these two key benefits of the proposed PMIX_Allgather compared to the existing PMI2_KVS_Fence. The intermediate data packing format used for communication between the *srun* and *slurmd*s is more compact as the keys are not included. Storing the end-result in an array indexed by source rank helps avoid creation and lookup from a hash table.

C. Design of PMIX_Iallgather and PMIX_KVS_Ifence

In many typical applications or middlewares using PMI, we can observe the following pattern:

```
//Produce key and values
PMI2_KVS_Put();
PMI2_KVS_Fence();
PMI2_KVS_Get();
//Use key and values
//Do unrelated computation
```

The client process's responsibility during an ongoing *Fence* is limited to informing the local *slurmd* at the initiation and waiting for the response at completion. As described in Section V-A, only the *srun* and the *slurmd*s are involved in progressing the actual operation. By using the non-blocking variant *Ifence*, the client can overlap unrelated computation that does not depend on the key-value pairs fetched via *Get*. The modified application would look like:

```
//Produce key and values
PMI2_KVS_Put();
PMIX_KVS_Ifence();
//Do unrelated computation
PMIX_Wait();
PMI2_KVS_Get();
//Use key and values
```

A similar transformation can be made with *Iallgather*, for which the pseudo code becomes:

```
//Produce key and values
PMIX_Iallgather();
//Do unrelated computation
PMIX_Wait();
//Use key and values
```

With an MPI library using *Ifence* and *Iallgather*, there are two sources of possible overlap. First, inside MPI_Init certain initialization procedures must be performed, e.g., registering host memory with the network interface, setting up shared memory segments, allocating resources, etc. Time taken by these tasks are independent of process count and application characteristics, and thus provides a constant amount of overlap. For larger jobs, the total time required for completion of *Ifence* or *Iallgather* is higher, and these tasks are not sufficient to maximize the overlap potential, which causes the second source, the application, to come into play.

Most MPI applications do not start communication with other processes immediately after MPI_Init. Performing different computation to generate local data or performing file I/O before entering the *communication phase* is a common behavior. We define the amount of time an application spends after MPI_Init before initiating the first connection as *setup phase*. If the *setup phase* is long enough, the *Ifence* or the *Iallgather* operation can be completely overlapped, reducing the total execution time. Furthermore, the processes are no longer bound by the synchronization property of the *Fence* and can proceed without incurring delay due to process skew. In general, the case for non-blocking PMI collectives are along the lines of works that exist in the MPI context [11].

D. Progressing Non-blocking Operations

An efficient implementation of non-blocking collectives requires an agent to progress the communication in the background while the caller can perform other tasks. This problem has been studied extensively in the context of MPI

TABLE II. VOLUME OF DATA TRANSFERRED AND TIME TAKEN IN DIFFERENT PHASES OF *Allgather*

Number of Processes in Job	**Gather phase** process → *slurmd* → *srun*					**Broadcast phase** *srun* → *slurmd* → process		Time for Data Processing (ms)	Total Time in Allgather (ms)
	Value Size (Bytes)	Key Size (Bytes)	Overhead (Bytes)	Total (Bytes)	Time (ms)	Data (KB)	Time (ms)		
4,096	18	4	4	26	165	104	201	0.18	275
8,192	18	4	4	26	256	208	356	0.36	471

non-blocking collectives. One solution is based on threads where the calling process creates a thread which initiates the operation and checks for completion. However, such an implementation requires one extra thread per client process which can adversely affect the performance [20].

In our design, the communication is progressed by *slurmd*, which is equivalent to an existing node-level agent. The benefit of this design is that *slurmd* is a standalone daemon process and already performs similar tasks, e.g., sending heartbeat message to the *slurmctld*. Additionally, the data exchange is performed in an out-of-band channel when the processes are still in their computation phase, so it has minimal effect on the application communication.

E. Supporting concurrent Non-blocking Operations

Supporting concurrent non-blocking operations can be useful for improving overlap in applications. However, in practice the data exchanges over PMI performed inside MPI_Init can be completed using a single *Ifence* or *Iallgather* operation. With this in mind, we do not support concurrent non-blocking operations for simplicity. Any non-blocking operation must be completed by *Wait* before initiating another. However, the proposed functions support this functionality and an implementation can provide this feature if desired.

F. Memory Footprint

For *Allgather* and *Iallgather* the processes are required to provide sufficient buffer space to hold all the values. Additionally, *slurmd* needs to allocate a temporary buffer of the same size. These two factors might suggest that the proposed extensions impose a significant memory overhead. However, in practice the MPI libraries allocate the same amount of memory to hold the addresses of the remote processes, even if they only communicate with a small subset. Furthermore, once a key-value pair is exchanged through *Fence* or *Ifence*, it is persisted inside *slurmd* till the application exits or calls PMI2_Finalize. However, the values exchanged trough *Allgather* or *Iallgather* are not persisted inside *slurmd* which can free or reclaim the allocated buffer as soon as the operation is complete. Thus, the amount of memory available to the MPI processes is actually larger for the proposed *Allgather* and *Iallgather*.

VI. EXPERIMENTAL RESULTS

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 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.

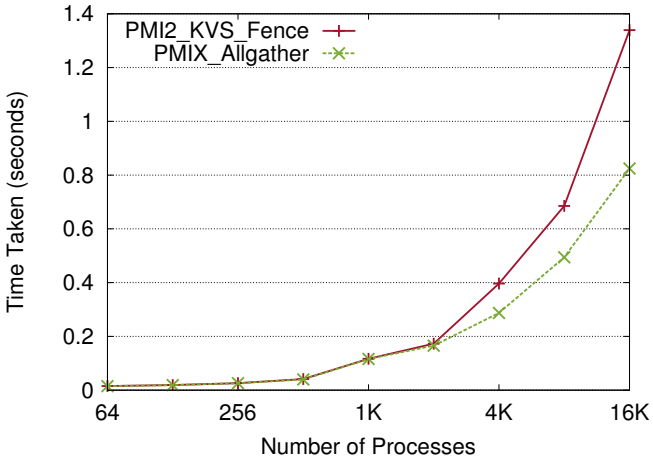

Fig. 9. Comparison of the existing *Fence* (PMI2_KVS_Fence) and the proposed *Allgather* (PMIX_Allgather)

B. PMI Microbenchmark Level Performance

To measure the performance of *Fence*, we use a small application that repeatedly performs a *Put* followed by a *Fence* operation. For fairness, we use the calling process's rank as the key and a 32-byte string as the value. For *Allgather*, we use the same value as the input. As seen from Figure 9, *Allgather* outperforms *Fence* by 38% at 16,384 processes. Once the *Allgather* operation is complete, all of the values are available in the client process's memory but after completion of *Fence*, the process must perform *Get* operations to access the keys. This additional overhead of *Fence* is not shown in Figure 9.

Table II shows the amount of data transferred and associated times for *Allgather* using the same set of key-value pairs used for *Fence* as shown in Table I. Using the integer rank as the key clearly leads to drastic reduction in data transfer time which is reflected in the total time to complete the operation as well. These lengths were chosen to mimic the key-value pairs used in the PMI exchanges in MVAPICH2.

We also measure the performance of *Ifence* and *Iallgather* followed immediately by *Wait* and compare it against their respective blocking variants (*Fence* and *Allgather*). We find that there is no additional overhead introduced by the proposed non-blocking collectives.

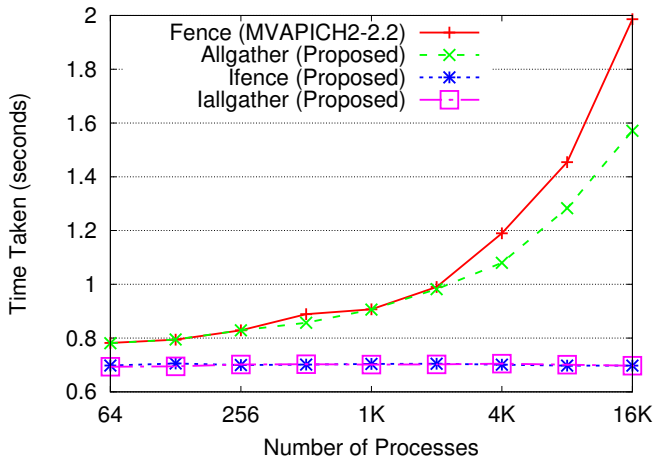

Fig. 10. Time taken by MPI_Init

C. MPI Level Performance

Figure 10 shows the impact of the proposed extensions on MPI_Init time. Replacing the *Fence* operation with *Allgather* yields 20% benefit at 16K processes. However, the most significant improvement comes from using the non-blocking PMI extensions which allows MPI_Init to complete with no communication with other processes. This results in a constant MPI_Init time independent of the number of processes. At 16,384 processes, MPI_Init with *Iallgather* exhibits speedup of 2.88 over MPI_Init based on blocking *Fence*. The predicted improvement at larger scale is even higher.

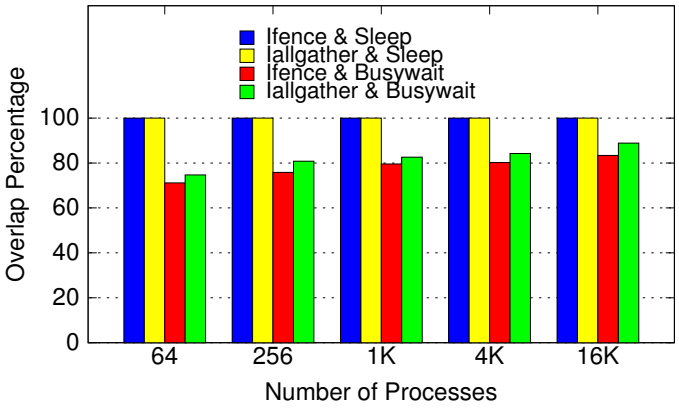

Fig. 11. Overlap percentage with different computations and number of processes

D. Overlap with Computation

To measure the overlap, two different types of computation are used. In the first, we make the client processes sleep for the amount of time the blocking *Fence* or *Allgather* takes while the *Ifence* or *Iallgather* is being performed. This allows for perfect overlap at all scales as shown in Figure 11.

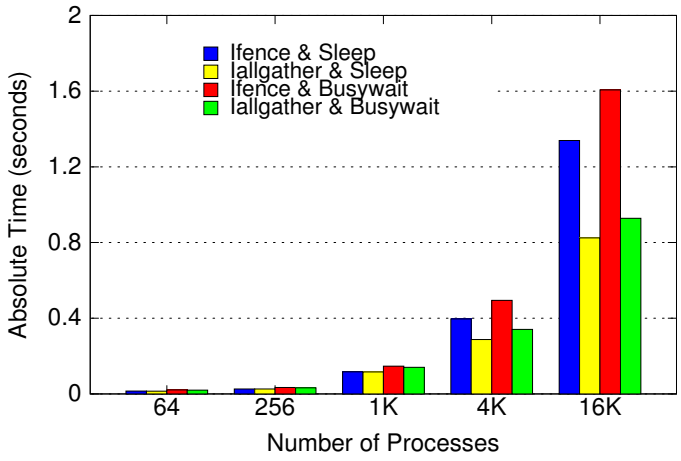

Fig. 12. Computation time required for perfect overlap

However, as *slurmd* progresses the *Ifence* or *Iallgather*, it needs to contend for CPU and memory with the client processes. To simulate the worst case scenario, we make all the client processes busy spin on the CPU for the same amount of time. As expected, this increases the wall clock time for the operation and thus reduces the observed overlap. The amount of overlap increases with process count and reaches 89% with 16K processes. The higher overlap observed in PMIX_Iallgather results from its more efficient data transfer and processing as shown in Table II.

Due to better communication performance *Iallgather* reduces the amount of time an application must spend during its *setup phase* in order to completely overlap the PMI exchange latency. At 16K processes the required time is reduced by 1.7 times by using the *Iallgather* operation, as illustrated in Figure 12. The time spent in MPI_Init already covers a portion of the total computation time required for perfect overlap, hence it represents an upper bound on how long the *setup phase* needs to be for the application to reap the full benefit.

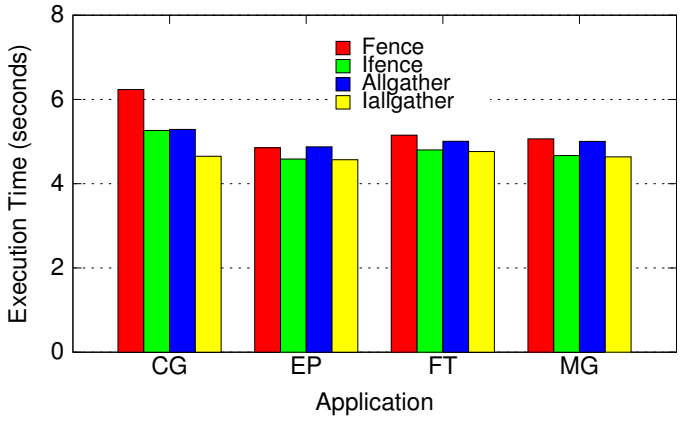

Fig. 13. Time taken by NAS Parallel Benchmarks at 4,096 processes

E. Effect of Proposed Extensions on Application Performance

We also measure the wall-clock time of some applications from the NAS Parallel Benchmarks (NPB) [21] with class B data and 4,096 processes in fully subscribed mode and observe improvements of up to 10% in total running time as shown in Figure 13. Use of non-blocking constructs like *Ifence* and *Iallgather* yields the most significant benefit. The improvement achieved depends on the application, specifically the time the application spends in the *setup phase* before the *communication phase*. In an application with a long *setup phase* the benefit of *Iallgather* over *Ifence* may not be apparent.

VII. Related Work

There has been significant work in the area of improving performance and scalability of launching parallel applications. Multiple process managers like PBS [22], MPD, Mpiexec [23], and Hydra [23] have been developed to reduce job scheduling and launch times. Wang et al [24] have proposed a multi-controller based job scheduling system called SLURM++.

Yu et al [25] explored using InfiniBand to reduce start up costs of MPI jobs. Sridhar et al proposed using a hierarchical ssh based tree structure similar to SLURM's node daemon implementation [26]. Gupta et al [27] 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 [28]. The impact of node level caching on startup performance was evaluated by Sridhar et al in [29].

The most closely related work to this paper is a project called PMIx [6]. PMIx proposes a number of new PMI functions including a non-blocking *Fence*. We extend that effort with an implementation and experimental demonstration of the value non-blocking PMI operations. We also propose new PMI allgather routines to optimize a common data exchange pattern in MPI startup.

VIII. Conclusions and Future Work

In this paper, we proposed, designed and implemented the PMIX_Allgather API to reduce the amount of data being transferred as well as avoid the data processing overheads in existing PMI designs. We also proposed, designed and implemented non-blocking versions of two PMI APIs — PMIX_KVS_Ifence and PMIX_Iallgather to allow overlap of PMI communication and other activities application / middleware need to perform at startup. We evaluated the benefits the new APIs have on performance at the microbenchmark level using PMI and MPI microbenchmarks and at application level using NAS parallel benchmarks. Our experimental evaluation demonstrated how such extensions reduce MPI startup cost. In particular, when sufficient work can be overlapped, these extensions allowed for a constant initialization cost of MPI jobs at different core counts. At 16,384 cores, the designs lead to a speedup of 2.88 times over the state-of-the-art.

As part of future work we plan to add support for multiple concurrent non-blocking collective operations. We also plan to design and evaluate the impact of high performance networks like InfiniBand on speeding up the PMI based communication.

References

[1] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, Mar 1994.

[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*. Springer, 2010, pp. 31–41.

[3] 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

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

[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] Open MPI: Open Source High Performance Computing, "PMI Exascale (PMIx)," https://github.com/open-mpi/pmix/wiki.

[7] InfiniBand Trade Association, "InfiniBand Architecture Specification, Volume 1, Release 1.0," http://www.infinibandta.com.

[8] H. Subramoni, K. Kandalla, S. Sur, and D. K. Panda, "Design and Evaluation of Generalized Collective Communication Primitives with Overlap Using Connectx-2 Offload Engine," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*. IEEE, 2010, pp. 40–49.

[9] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda, "High-performance and scalable non-blocking all-to-all with collective offload on infiniband clusters: a study with parallel 3d fft," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 237–246, 2011.

[10] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a Conjugate Gradient Solver with Non-blocking Collective Operations," *Parallel Computing*, vol. 33, no. 9, pp. 624–633, 2007.

[11] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, "A Case for Standard Non-blocking Collective Operations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2007, pp. 125–134.

[12] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-blocking Collective Operations for MPI," in *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*. IEEE, 2007, pp. 1–10.

[13] J. C. Sancho, K. J. Barker, D. Kerbyson, and K. Davis, "Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-scale Scientific Applications," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 17–17.

[14] J.C. Sancho, D.J. Kerbyson and K.J. Barker, "Efficient Offloading of Collective Communications in Large-Scale Systems," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 169–178, 2007.

[15] Rabinovitz, Ishai and Pavel Shamis and Richard L. Graham and Noam Bloch and Gilad Shainer, "Network Offloaded Hierarchical Collectives Using ConnectX-2's CORE-Direct Capabilities," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10, 2010, pp. 102–112.

[16] Venkata, Manjunath G. and Richard L. Graham and Joshua S. Ladd and Pavel Shamis and Ishai Rabinovitz and Vasily Filipov and Gilad Shainer, "ConnectX-2 CORE-Direct Enabled Asynchronous Broadcast Collective Communications," in *CASS'11*, Mar. 2011.

[17] H. Subramoni, K. Kandalla, S. Sur and D K. Panda, "Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine," in *HotI'18*, 2010.

[18] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur and D. K. Panda, "High-Performance and Scalable Non-Blocking All-to-All with Collective Offload on InfiniBand Clusters: A Study with Parallel 3D FFT ," in *ISC*, June,2011.

[19] K. Kandalla, H. Subramoni, J. Vienne, K. Tomko, S. Sur and D. K. Panda, "Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: A Case Study with HPL ," in *Hot Interconnects*, August, 2011.

[20] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - to Thread or Not to Thread?" in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 213–222.

[21] 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.

[22] R. L. Henderson, "Job Scheduling under the Portable Batch System," in *Job scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 279–294.

[23] P. Balaji, W. Bland, W. Gropp, R. Latham, H. Lu, A. J. Pena, K. Raffenetti, R. Thakur, and J. Zhang, "MPICH Users Guide," 2014.

[24] K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu, "Next Generation Job Management Systems for Extreme-scale Ensemble Computing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. ACM, 2014, pp. 111–114.

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

[26] 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.

[27] 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*. ACM, 2011, pp. 41–48.

[28] 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.

[29] 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*. Springer, 2009, pp. 230–239.