Job Startup at ExaScale: Challenges and Solutions

Sourav Chakraborty, Advisors: Hari Subramoni, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{chakraborty.52, subramoni.1, panda.2}@osu.edu

ABSTRACT

Multi/Many-core processors and high-performance interconnects have fueled the growth of high-performance computing clusters. Launching large scale jobs efficiently on hundreds of thousands cores has been a significant challenge due to the characteristics of such interconnects and various bottlenecks present in the bootstrapping mechanisms used by modern HPC middlewares such as MPI and PGAS libraries. In this work, we identify these bottlenecks and propose scalable solutions for them. We introduce on-demand connection management for MPI and OpenSHMEM on different networks to reduce job-startup time. We also propose extensions to the PMI [4] standard to reduce the data transferred over the network as well as overlap the communication with other initialization tasks. In addition, we propose methods for efficient intra-node topology discovery and a shared memory based design for PMI geared towards many-core systems. The proposed design can reduce the memory footprint of PMI by a factor of processes per node (PPN). Our evaluation shows that with sufficient overlap, near-constant initialization time can be achieved at any process count for MPI, PGAS, and hybrid MPI+PGAS applications. Initialization time of OpenSHMEM is improved by 29.6 times at 8,192 processes. Estimated memory footprint for PMI is reduced by nearly 4GB with 1 million processes and 64 PPN. Compared to MVAPICH2-2.2 and Intel MPI-2017, time taken for MPI_Init is improved by 25 times and 14 times respectively with 4,096 processes on 256 nodes equipped with Knight Landing CPUs and Omni-Path HCAs. In addition, we are able to initialize a 65,536 process MPI job in just 5.8 seconds, a significant improvement over other state-of-the art launchers.

1 INTRODUCTION

Modern high-performance computing (HPC) systems offer sustained multi peta-flops performance and are enabling scientists to scale their parallel applications to hundreds of thousands of processors. As clusters continue to increase in size, fast and scalable startup of parallel applications is becoming more important. It is often necessary to restart an HPC job multiple times during development and debugging. Reducing startup costs from minutes to seconds can cumulatively save developers hours of time. While testing a system or while regression testing an application, many large-scale, quick-running jobs must be run in succession. In this case, startup becomes the dominant cost so that improving startup dramatically speeds up testing time. Cost of launching jobs at scale is also a major factor in efficiency of checkpoint/restart. While significant research has been done on improving communication performance of HPC middlewares, to the best of our knowledge no systematic study has been done to analyze and improve the bottlenecks involved in large-scale job startup.

The massive growth in the size and scale of supercomputing systems over the last decade has been driven by the current trends in multi-/many-core architectures and the availability of commodity, RDMA-enabled, and high-performance interconnects such as InfiniBand [1] (IB) and Omni-Path [5]. For example, Aurora [2], the next generation supercomputer announced by DOE, is expected to have more than 50,000 Knights Landing (KNL) based nodes and more than 3 million cores. These advancements in processor and interconnect technology coupled with the massive increase in scale present a set of challenges in achieving scalable job-startup that are not addressed in current HPC middlewares. In this work we identify these challanges, propose scalable solutions to address them, and evaluate them against state-of-the-art solutions at large scale.

2 BOTTLENECKS IN JOB-STARTUP

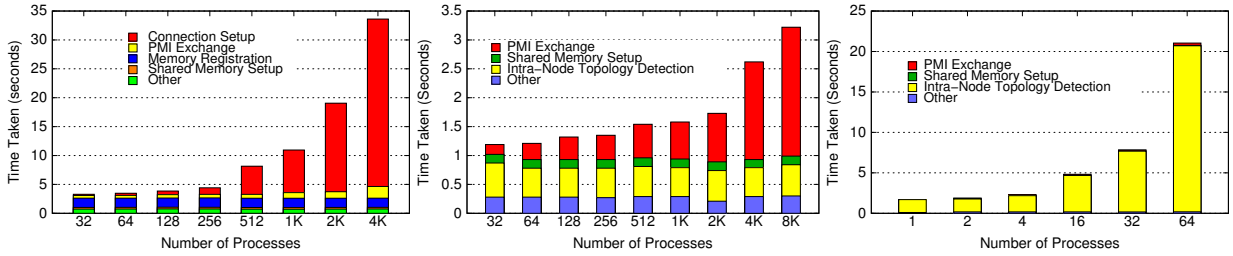
The Message Passing Interface (MPI) [17] has been the de-facto standard for programming models for developing such applications. Recently, Partitioned Global Address Space (PGAS) models such as OpenSHMEM [11] have also seen widespread adoption among application developers. To identify the bottlenecks involved in bootstrapping high-performance MPI and PGAS libraries, we study the time taken by various steps during their initialization. Time taken to return from MPI_Init() and start_pes() are treated as the initialization times for MPI and OpenSHMEM respectively. From the breakdowns shown in Figure 1(a) and 1(b), we can identify the following scalability issues:

2.1 Static Connection Setup

If there are N ranks involved in an MPI/PGAS application, it can require up to $N * (N - 1)$ individual connections to be established. A common practice is to establish these connections during initialization. However, as shown in Figure 1(a), establishing these $O(N^2)$ connections can take a significant amount of time during job-startup, especially for large scale jobs. In addition, many of these connections remain unused in a large number of applications. Table 1 lists the average number of peers each process communicates with for a few different applications at two different job sizes. As shown here, for most applications, the average number of communicating peers is much smaller compared to the total number of processes in the job.

Application	Number of Processes	Average Number of Peers
BT	64	8.70
	1024	10.6
EP	64	3.00
	1024	5.01
MG	64	9.46
	1024	11.9
SP	64	8.75
	1024	10.7
2D Heat	64	5.28
	1024	5.40

Table 1: Average number of communicating peers per process for different applications.



(a) OpenSHMEM Initialization (start_pes) on Xeon + InfiniBand (b) MPI Initialization (MPI_Init) Excluding Connection Setup on Xeon + InfiniBand (c) Single Node MPI Initialization on Knights Landing + Omni-Path

Figure 1: Breakdown of time taken for initializing OpenSHMEM and MPI libraries at different scales

2.2 Network Address Exchange over PMI

Modern MPI libraries support high-performance interconnects such as InfiniBand and Omni-Path to perform low latency and high bandwidth communication. However, to bootstrap communication on such interconnects, the MPI processes need to exchange network addresses among themselves through an out-of-band channel. Most modern MPI/PGAS libraries use the Process Management Interface (PMI) to perform this exchange [4]. The PMI interface provides a standard way for HPC middlewares such as MPI libraries to communicate with the resource manager. In addition, PMI provides a global key-value store visible to all the processes in a job and exposes the following operations - *Put* (adds a key-value pair to the store), *Get* (retrieves the value for a given key), and *Fence* (distributes the key-value pairs to all processes). As shown in Figure 1(b), this PMI exchange step is another major bottleneck in bootstrapping MPI/PGAS libraries at large scale.

2.3 Intra-node Topology Detection

The basic performance characteristics of a system can vary significantly based on the type, number, and placement of CPUs and HCAs inside a node. Furthermore, Heterogeneous systems comprising multiple different configurations of nodes inside the same cluster are also common. To extract the best performance, many MPI libraries tune various internal parameters based on the internal topology of the system. HWLOC [6] is a popular and standard way to obtain this information. However, as shown in Figure 1(c), this step can be expensive on many-core machines; and this cost is only going to be exacerbated by upcoming many-core architectures with even large number of cores per node.

While these trends were obtained from the MVAPICH2 MPI library, these scalability issues and are present in many other MPI and PGAS libraries as well.

3 PROPOSED SOLUTIONS AND DESIGNS

In this section, we propose different designs to address the scalability issues identified in Section 2. Figure 2 shows an overview of the set of proposed designs. Broadly, our contributions in this work are the following:

- Identify both inter- and intra-node bottlenecks in large scale job startup of MPI and PGAS libraries
- Introduce dynamic connection management for OpenSHMEM and MPI on InfiniBand and Omni-Path
- Propose PMI extensions to leverage high-performance networks during address exchange and overlap the cost
- Propose designs to reduce memory footprint of PMI
- Propose efficient methods for intra-node topology discovery using HWLOC
- Evaluate proposed designs against state-of-the art libraries on different architectures at large scale

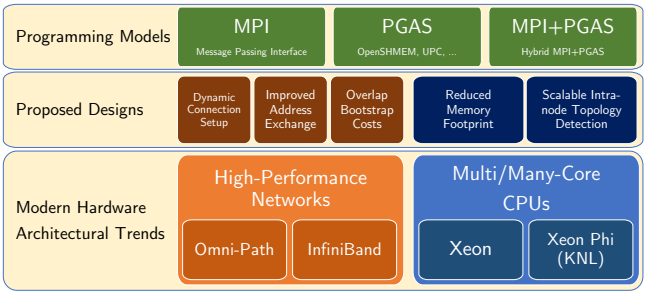


Figure 2: Overview of Proposed Designs

3.1 Dynamic Connection Setup

As discussed in the previous section, statically setting up all-to-all connectivity is wasteful and not scalable. To avoid this issue, connection between any two processes can be established only when they try to communicate for the first time. However, this step is tightly coupled with the underlying interconnect and must be carefully designed to avoid potential deadlocks.

For example, InfiniBand provides two major transport protocols - Reliable Connection (RC) and Unreliable Datagram (UD). RC is reliable and but requires one QP (Queue-Pair) per communicating peer. On the other hand, UD is unreliable but a single QP can be used to send data to any peer process [14]. At initialization, each process creates a thread that opens a UD queue-pair and publishes this address through the PMI interface. When a process P1 tries to send its first message to a peer process P2, it looks up the remote peer's address, opens a RC QP and sends its address through a connection request message to the listener thread of P2. P2 then creates a new RC QP and sends its address back through a connection reply message. In case both P1 and P2 initiates a connection at the same time, a tie-breaking mechanism based on the logical rank is used to avoid a deadlock or creation of multiple connections for a given pair. For OpenSHMEM, this design was implemented inside the underlying GASNet conduit to make it applicable to other PGAS languages such as UPC and CAF as well [8].

For the Omni-Path interconnect, the communicating processes rely on Endpoints (EP). The process of creating and publishing addresses of the EPs are similar to that of the InfiniBand. However, since Omni-Path internally handles the connection request and reply messages, a separate thread is not required. When a process P1 tries to send its first message to a peer process P2, it looks up P2's EP address through PMI and establishes a connection. Unlike InfiniBand, these connections are uni-directional, i.e., P2 needs to establish a new connection even if P1 is already connected to P2. Thus, a tie-breaking mechanism is not required for this design.

3.2 Accelerating Address Exchange

As discussed before, MPI/PGAS libraries rely heavily on the PMI interface to exchange network addresses to bootstrap their communication framework. This step is required even if connectionless protocols are used. The PMI interface exposes a global key-value store (KVS) through which MPI/PGAS libraries can perform this exchange. However, current semantics of the PMI protocol requires an expensive "Fence" operation to be performed before the KVS can be queried. This operation scales linearly with the number of key-value pairs being exchanged. Furthermore, existing PMI implementations perform all communication over TCP/IP and do not take advantage of the modern interconnects. Based on these observations, we propose a new PMI operation PMIX_Ring to establish a ring structure by exchanging key-value pairs with only the left and right neighbors. Once the ring is established, the high speed network is used to exchange bulk of the data. Since time taken by the Ring operation is nearly independent of process count [9] and the high-speed networks offer much higher bandwidth compared to TCP/IP, the total time of the address exchange is reduced significantly.

3.3 Overlapping Address Exchange Costs

PMI operations that require communication with remote nodes such as "Fence" and "Ring" are progressed by the process manager (e.g., srun, mpirun_rsh). Since all current PMI calls are blocking, the MPI/PGAS libraries cannot perform anything useful during this phase. To alleviate this, we introduce a non-blocking collective called *Ifence* which allows processes to perform other actions while the PMI operations progress in the background. We also introduce *Allgather*, and its non-blocking variant, *Iallgather*, which combine the functionality of *Put*, *Fence* and *Get* and reduce data movement by using the rank as the implicit key[7]. Using these new non-blocking collectives, the MPI/PGAS libraries can overlap the PMI communication with other steps such as opening shared memory regions. In addition, with dynamic connection setup, the non-blocking PMI operations are not required to be completed for MPI_Init()/star_pes() to return. Instead, they can continue until the application actually tries to perform a remote communication. Consequently, for many applications that perform non-communication tasks such as reading input files, setting up the problem space etc., the amount of overlap available can be higher than what is available from the MPI library.

3.4 Reducing Memory Footprint

In existing designs of PMI, there exists a "Node Level Agent" (NLA) on each node that is responsible for locally caching the key-value store and processing the commands (Put/Get/Fence etc.) issued by the MPI/PGAS libraries. This intra-node PMI communication is performed over UNIX sockets which exhibits poor latency as number of concurrent clients increases. Consequently, the key-value store is replicated by the processes which causes a total of PPN+1 copies of the address information to be stored on each node. We propose a design where the key-value store is exposed using shared memory regions. The key-value store is implemented as an insert/lookup-only hash-table that uses two shared memory regions to grow efficiently. PMI Put and Get operations are translated to direct insertion and lookup operations on the hash table. This improves Get performance and reduces PMI memory footprint per node by PPN times [10]. As an additional benefit, this approach does not require any change to the PMI interface itself. Thus, it remains backward-compatible and can be used by any MPI library.

3.5 Enhanced Intra-node Topology Detection

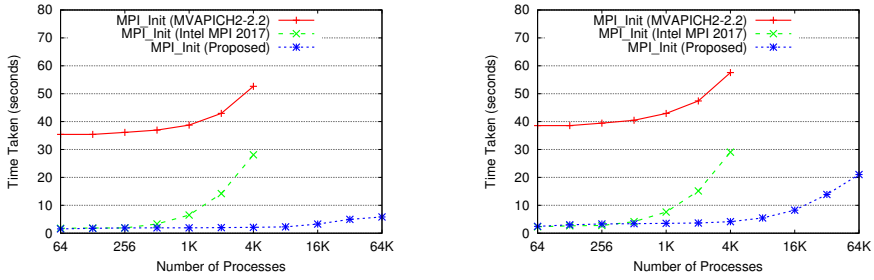
Many modern MPI libraries including MVAPICH2 and Open-MPI depend on HWLOC to discover the architecture and topology of the compute nodes and tune their internal parameters accordingly. By default, each process on the node invokes the HWLOC topology discovery functions independently. These functions read a large number of pseudo files from the /proc filesystem to populate the topology. This approach leads to very poor performance on many-core architectures, due to three factors: 1) lower clock speed of the CPU cores, 2) larger number of pseudo files that need to be read, and 3) larger number of concurrent reads. To avoid this, only one process on each node performs the topology discovery using HWLOC and serializes the topology into a file. The other processes then directly reads the file and deserializes the information instead of interacting with HWLOC.

4 EXPERIMENTAL EVALUATION

The TACC Stampede and Oakforest-PACS systems were used for experimental evaluation. In Stampede, each compute node is equipped with two Intel SandyBridge eight-core sockets @2.70 GHz, 32 GB RAM and FDR ConnectX-3 HCAs (56 Gbps). Oakforest-PACS has Intel Xeon Phi 7250 KNL compute nodes (68 cores per node, 4 hardware threads per core, 1.4 GHz) with 96 GB RAM that are connected with 100Gb/sec OmniPath network. The KNL nodes were configured in "cache" mode where the 16 GB High-Bandwidth memory (MCDRAM) is used as a direct-mapped L3 cache. The maximum number of processes used on KNL is set to 64 to avoid oversubscribing the physical cores. SLURM-15.08.1 and MVAPICH2-2.2 were used to implement the proposed designs. The results were compared against the default configurations of MVAPICH2 and Intel MPI 2017.1.132. All numbers reported are average of at least 10 iterations.

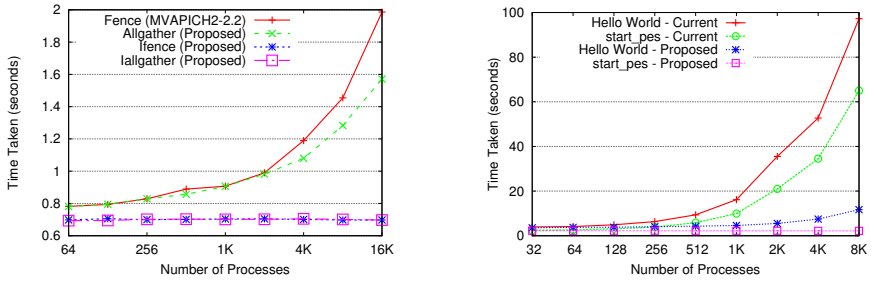
4.1 Startup Performance on KNL + Omni-Path

Figure 3(a) shows the time taken by MPI_Init() at different process counts on a Knights Landing + Omni-Path system at 64 processes per node. Compared to the latest MVAPICH2 and Intel MPI, the proposed designs offer significantly reduced startup time over state-of-the art. With 4,096 processes on 64 nodes, the proposed design performs 24 times better than MVAPICH2 and 14 times better than Intel MPI and is also more scalable. With 65,536 processes on 1,024 nodes, MPI_Init took only 5.8 seconds. Similar trends can be seen for execution time of Hello World in Figure 3(b). Total execution



(a) Time taken by MPI_Init (b) Execution time of MPI Hello World

Figure 3: MPI Initialization and Hello World execution times on Knights Landing + Omni-Path Architecture



(a) Time taken by MPI_Init (b) OpenSHMEM initialization and Hello World

Figure 4: MPI and OpenSHMEM Startup performance on Xeon + InfiniBand Architecture

time of the Hello World application with 65,536 processes on 1,024 nodes was only 21.02 seconds. It should be noted that Intel MPI does not use HWLOC for topology detection by default, which explains its performance difference with MVAPICH2-2.2.

At small scale, majority of the improvement is obtained from the improved topology detection and shared memory based PMI. At larger scale, most of the benefit is derived from dynamic connection setup and improvements in the PMI address exchange.

4.2 Startup Performance on Xeon + InfiniBand

Figure 4(a) compares the impact of different PMI-level designs on startup performance. Compared to the default "Fence"-based design, "Allgather" improves the startup performance by 20% at 16,384 processes and 16 processes per node. The largest benefit is obtained from the non-blocking variants "Ifence" and "Iallgather". With these extensions, MPI_Init can complete in near-constant time at any scale since most of the PMI exchange cost is overlapped with other operations. At 16,384 processes and 1,024 nodes, the "Iallgather" design is almost 3 times faster than the default "Fence" based design due to the reduction in total data movement.

We also compare the startup performance of OpenSHMEM using start_pes() and a Hello World program with the default and the proposed designs in Figure 4(b). Compared to the default design, the proposed design improves initialization time by up to 30 times with 8,192 processes on 512 nodes. Most of these improvements are obtained from dynamic connection setup and efficient exchange of addresses through PMI.

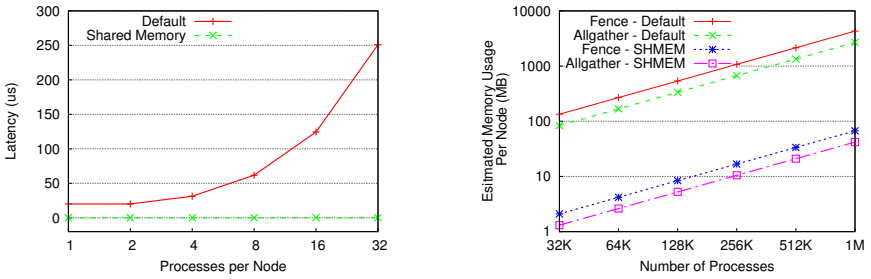
4.3 Impact on Intra-node Performance and Memory Footprint

Figure 5(a) shows that with the shared-memory design, PMI Gets take 0.25us compared to the 250us with the default design with 32 processes. This represents an improvement of 1,000 times over the current socket-based design. Figure 5(b) shows memory consumption of PMI with different schemes with 64 processes per node projected from smaller scale runs. The Allgather design can save up to 60% compared to Fence due to its optimized data representation. With the Shared memory based design, memory consumption is reduced by a factor of PPN. On a system similar to Aurora, the proposed design can save nearly 4 GB memory per node for a 1 million process job by removing the redundant information.

5 RELATED WORK

There has been significant work in the area of improving performance and scalability of launching parallel applications. Multiple process managers like PBS [13], MPD, Mpiexec [3], and Hydra [3] have been developed to reduce job scheduling and launch times. The implementation and impact of on-demand connection management in MPI over VIA-based networks was presented by Wu et al [23]. An implementation for on-demand connections for the ARMCI interface is described in [22]. Multiple researchers have proposed different connection schemes [14, 16, 21, 24] to improve the scalability of MPI runtimes over InfiniBand. The MVAPICH-Aptus runtime [15] dynamically selects the UD or RC protocol based on the application's communication pattern.

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 [19]. Goehner et al analyzed the effect of different tree configurations and proposed a framework called LIBI [12]. The impact of node level caching on startup performance was evaluated by Sridhar et al in [20]. A closely related work to this paper is a project called PMIx [18] which has been working in parallel to improve the PMI interface. We extend that effort with an implementation



(a) Performance of PMI Get

(b) Memory footprint with Shared-memory based PMI

Figure 5: Impact of the Shared-memory based design of PMI

and experimental demonstration of the value non-blocking PMI operations. We also propose new PMI routines to optimize common data exchange patterns in MPI startup an take advantage of high-performance networks to speed up the data exchange.

6 CONCLUSION AND FUTURE WORK

In this work, we analyzed the job-startup performance of MPI and PGAS libraries on modern multi/many-core systems equipped with high-performance interconnects and identified the various bottlenecks. We proposed scalable solutions for these bottlenecks and evaluated them against state-of-the art MPI libraries including MVAPICH2-2.2 and Intel MPI 2017 on large scale clusters and different hardware setups. With the proposed designs, we were able to show more than 25 times and 29 times improvement for MPI and OpenSHMEM initialization times respectively. Memory footprint for address storage was also reduced by a factor of PPN, leading to an estimated savings of 4GB with 1 million processes and 64 PPN. Our designs showed improvement of more than 14 times compared to Intel MPI on KNL + Omni-Path systems. We were also able to initialize a 65,536 process MPI job in just 5.8 seconds, a significant improvement over other state-of-the art MPI libraries.

The proposed designs are publicly available in the latest releases of MVAPICH2 and MVAPICH2-X and are being used in several production clusters. Some of the designs have already been adopted into the public release of SLURM as well. Going forward, we plan to adopt the proposed designs to UPC, CAF and other PGAS languages and investigate avenues to improve the performance and scalability of large scale job startup.

REFERENCES

[1] 2017. Infiniband Trade Association. (2017). http://www.infinibandta.org/

[2] Argonne National Laboratory. 2017. Aurora Supercomputing Systems. (2017). http://aurora.alcf.anl.gov/

[3] Pavan Balaji, Wesley Bland, William Gropp, Rob Latham, Huiwei Lu, Antonio J Pena, Ken Raffenetti, Rajeev Thakur, and Junchao Zhang. 2014. MPICH User's Guide. (2014).

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

[5] M S Birrittella, M Debbage, R Huggahalli, J Kunz, T Lovett, T Rimmer, K D Underwood, and Robert C Zak. 2015. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, 1–9.

[6] F Broquedis, J Clet-Ortega, S Moreaud, N Furmento, B Goglin, G Mercier, S Thibault, and R Namyst. 2010. HWLOC: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 180–186.

[7] S Chakraborty, H Subramoni, A Moody, A Venkatesh, J Perkins, and D K Panda. 2015. Non-Blocking PMI Extensions for Fast MPI Startup. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 131–140.

[8] S Chakraborty, H Subramoni, J Perkins, A A Awan, and D K Panda. 2015. On-demand Connection Management for OpenSHMEM and OpenSHMEM+MPI. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015 IEEE International*. IEEE.

[9] S. Chakraborty, H. Subramoni, J. Perkins, A. Moody, M. Arnold, and D. K. Panda. 2014. PMI Extensions for Scalable MPI Startup. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. ACM, New York, NY, USA, Article 21, 6 pages. DOI: http://dx.doi.org/10.1145/2642769.2642780

[10] S Chakraborty, H Subramoni, J Perkins, and D K Panda. 2016. SHMEMPMI: Shared Memory Based PMI for Improved Performance and Scalability. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE.

[11] B Chapman, T Curtis, S Pophale, S Poole, J Kuehn, C Koelbel, and L Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2.

[12] 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. 2013. LIBI: A Framework for Bootstrapping Extreme Scale Software Systems. *Parallel Comput.* 39, 3 (2013), 167–176.

[13] Robert L Henderson. 1995. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*. Springer, 279–294.

[14] M. Koop, J. Sridhar, and D. K. Panda. 2008. Scalable MPI Design over InfiniBand using eXtended Reliable Connection. *IEEE Int'l Conference on Cluster Computing (Cluster 2008)* (September 2008).

[15] M J Koop, T Jones, and D K Panda. 2008. MVAPICH-Aptus: Scalable High-performance Multi-transport MPI over InfiniBand. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE.

[16] M J Koop, S Sur, Q Gao, and D K Panda. 2007. High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. ACM, New York, NY, USA, 180–189. DOI: http://dx.doi.org/10.1145/1274971.1274997

[17] Message Passing Interface Forum 1994. *MPI: A Message-Passing Interface Standard*. Message Passing Interface Forum.

[18] Open MPI: Open Source High Performance Computing. 2017. PMI Exascale (PMIx). (2017). https://github.com/open-mpi/pmix/wiki

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

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

[21] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda. 2006. Shared Receive Queue based Scalable MPI Design for InfiniBand Clusters. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[22] A Vishnu and M Krishnan. 2010. Efficient On-Demand Connection Management Mechanisms with PGAS Models over InfiniBand. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 175–184.

[23] J Wu, J Liu, P Wyckoff, and D K Panda. 2002. Impact of On-Demand Connection Management in MPI over VIA. In *In CLUSTER fi02: Proceedings of the IEEE International Conference on Cluster Computing*. IEEE Computer Society, 152–159.

[24] W. Yu, Qi Gao, and D. K. Panda. 2006. Adaptive Connection Management for Scalable MPI over InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

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