Optimized All-to-all Connection Establishment for High-Performance MPI Libraries over InfiniBand

Shulei Xu, Goutham Kalikrishna Reddy Kuncham, Mustafa Abduljabbar,
Hari Subramoni, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio, USA
{xu.2452, kuncham.2, abduljabbar.1, subramoni.1, panda.2}@osu.edu

***Abstract*—In modern multi-/many-core HPC systems, the increasing number of processor cores presents new challenges in managing parallel compute workloads across multiple nodes. One crucial aspect that significantly impacts the startup phase of parallel MPI jobs is the methodology used for connection establishment. In this paper, we investigate the limitations of existing all-to-all connection establishment designs in-depth, identify the primary sources of performance overhead, and propose an optimized all-to-all connection establishment design. This is done through an enforced ordering rank-by-rank scheme that significantly cuts down on the data exchange overhead via PMI. To address the increasing overheads associated with queue pair creation and synchronization, we explore multi-thread parallelism and incorporate CPU affinity awareness into our design. We implement our proposed design in the state-of-the-art MVAPICH2 MPI library and conduct extensive experiments on two emerging architectures. Through a comprehensive performance evaluation of these architectures, we demonstrate the efficacy of our optimized all-to-all connection establishment design. Our microbenchmark results reveal up to 20 times faster MPI_Init time, while evaluations with application kernels exhibit a 31% improvement in throughput.**

***Index Terms*—Multi-/Many-core, HPC, MPI, Job Startup, Connection Establishment, InfiniBand, RDMA**

I. Introduction

Modern High-Performance Computing (HPC) clusters have witnessed unprecedented advancements with the emergence of multi-/many-core processors and high-bandwidth, low-latency networking technologies. These rapidly developing technologies provide scientists and engineers with the ability to allocate an ever-increasing number of compute nodes and processor cores for their HPC workloads.

To effectively orchestrate the increasing number of compute resources, which include CPU, GPU, DPU, and more, programming models like Message Passing Interface (MPI) [1] are commonly employed. MPI is one of the most popular programming models for writing parallel applications in cluster computing area. MPI libraries provide basic communication support 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 underlying network dynamics and try to leverage the best communication performance on the given interconnect.

To launch a parallel MPI job, the MPI program must first invoke MPI_Init to initialize the MPI execution environment. During the initialization process, the MPI library firstly identifies each process (as MPI rank), and then establishes connections between them. State-of-the-art interconnect technologies such as InfiniBand, allow a process to use Direct Memory Access (DMA) to read or write data directly to or from other processes' application memory, bypassing the need for the intervention of operating system. These connections allow MPI processes to communicate and exchange data throughout the execution of the parallel program. There are two major connection establishment approaches commonly used in MPI: on-demand connection establishment and all-to-all connection establishment [2].

In all-to-all connection establishment, connections are made between all MPI ranks/processes, typically during the MPI_Init phase. As the example shown in Figure 1, rank 0 needs to build connections to all other ranks from rank 1 to rank N and store the connection information in local process. Same process is required for each rank. This design is advantageous in scenarios where frequent and structured communication between all ranks is expected, as it allows for efficient communication patterns and can minimize communication overhead during the program execution.

On the other hand, on-demand connection establishment involves establishing connections dynamically during program execution as needed. This approach is suitable when communication occurs sporadically or in a selective manner, such as broadcast collective communication, as it potentially reduces the overhead associated with establishing connections that may not be immediately required. One of the major limitation of on-demand connection establishment is that if the program ends up establishing nearly all-to-all connections during program execution, which is a common scenario in many applications, it can result in significant increased execution time compared to establishing the connection during MPI_Init phase. In contrast, the all-to-all connection is established during the MPI_Init phase. This allows for a more efficient setup of communication infrastructure, and reduces the overall

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.

communication overhead.

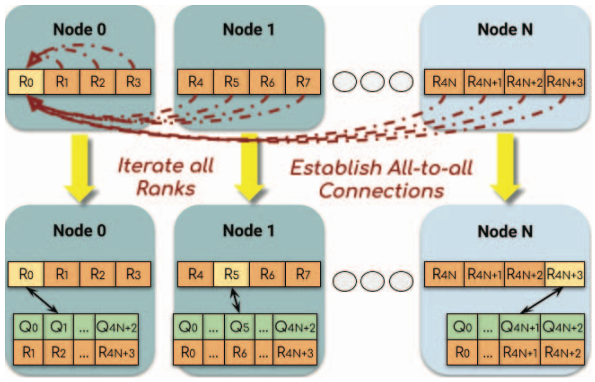


Fig. 1. Steps showing all-to-all connection establishment between all ranks

According to the usage trends of the various HPC systems observed by the US National Science Foundation (NSF) [3] funded Extreme Science and Engineering Discovery Environment (XSEDE) [4] project, the total consumed CPU hours and a significant majority of HPC jobs and are attributed to one or no more than 8 nodes. These trends are shown in Figure 3(a) and Figure 3(b). Considering the most popular job

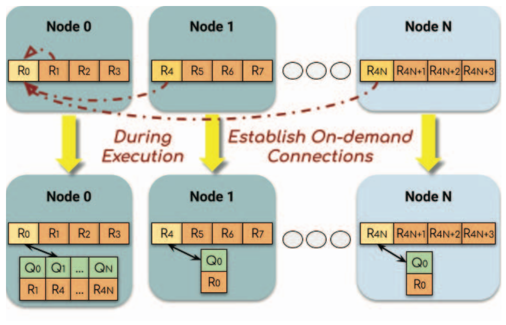
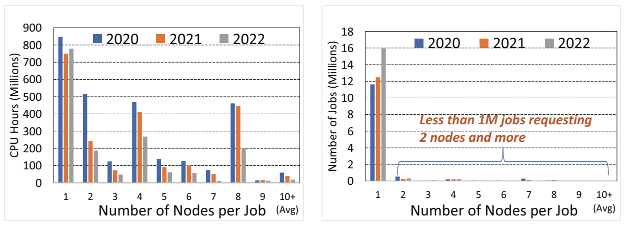


Fig. 2. Steps showing on-demand connection establishment between all ranks

size, we tackle the limitation of existing all-to-all connection establishment approach by proposing an optimized design to mitigate the bottlenecks of the existing all-to-all connection establishment design.



(a) Total CPU Hours

(b) Job Sizes

Fig. 3. Number of submitted jobs and total CPU hours consumed by jobs of different sizes over past three years in XSEDE clusters.

In the this paper, we revisit the existing all-to-all connection establishment design and analyze its limitations and primary sources of performance overhead. Based on this analysis, we propose an optimized design that streamlines the all-to-all connection establishment to take advantage of characteristics of the hardware behavior. Furthermore, we explore enhancement of proposed design by incorporating thread-level parallelism and CPU-affinity awareness into our optimized design. Then we conduct a comprehensive performance evaluation with micro-benchmarks and real applications, to validate the efficacy of our proposed design and assess the trade-offs between our design and existing on-demand design. Notably, similar approaches to the proposed optimized all-to-all connection design proposed by this paper will be applicable to other connection-oriented transport protocols available on other HPC networks, such as Omni-Path Express, Omni-Path, Slingshot, etc.

To the best of our knowledge, this is the first study that presents an optimized design of all-to-all connection establishment in InfiniBand Remote Direct Memory Access (RDMA) while leveraging thread-level parallelism and CPU-affinity awareness. This approach sets a new precedent in tackling the issue of performance overhead in MPI implementations over InfiniBand RDMA.

II. Motivations and Challenges

The motivation for an optimized RDMA all-to-all connection establishment design comes from the scalability limitations of existing MPI startup designs. As job sizes or the number of processes increase, existing designs struggle to efficiently handle the growing number of connections, leading to performance degradation.

For instance, Figure 4 presents an evaluation of MPI_Init execution time using the startup benchmark from OSU-microbenchmark [5]. The benchmark assesses the performance of MPI_Init as job sizes increase. The figure demonstrates an exponential increase in startup time, particularly evident beyond 8 nodes and 448 processes. This steep rise in time highlights the noncompetitive performance of existing all-to-all connection startup design.

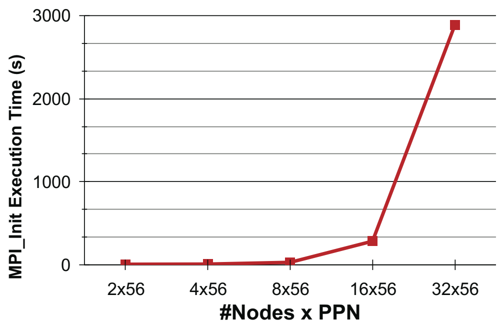


Fig. 4. MPI_Init execution time of All-to-all RDMA Connection Startup, taken by osu_init of OSU-Microbenchmark and run with up to 32 nodes 1792 processes on TACC Frontera

To tackle the limitation of current all-to-all MPI startup designs, our approach involves conducting a program profiling analysis to understand the primary cause of the high startup cost. Specifically, we perform program profiling for the 16-node 896-process startup performance degradation shown in Figure 4. We utilize the built-in run-time startup profiling option in MVAPICH2, a popular MPI library. From our profiling analysis, we identify the three most costly types of operations: Process information allocation, Queue pair creation or initialization, and Data Exchange via PMI (Process Management Interface). Figure 5 illustrates the breakdown of execution time among these operations. Notably, we observe that in the current all-to-all startup design, the data exchange operations over PMI dominate the total execution time (83.5%). Smaller and larger scale results follow the same trend here.

This finding underscores the significance of optimizing the data exchange over PMI in order to reduce the overall startup cost.

Another major limitation of the all-to-all connection establishment design is the maximum number of queue pairs supported on Host Channel Adapters (HCA) devices. For example, NVIDIA ConnectX-6 network adapters support up to 131,072 queue pairs, and Broadcom RoCEv2 adapters support 65,536 queue pairs. Although the maximum queue pair count is expected to increase in future generations of adapters, currently popular network adapters may not have sufficient queue pairs to accommodate all-to-all connection establishment when job size exceeds a few thousand processes. Given the fundamental requirement of establishing connections between every pair of processes in an all-to-all communication pattern (at least $N \times (N-1)$ connections, where N represents the total number of processes), this paper acknowledges the hardware limitation and does not specifically address it. Instead, the focus of this paper will be on optimizing the PMI data exchange process, while assuming that the necessary number of queue pairs is available for the given job size.

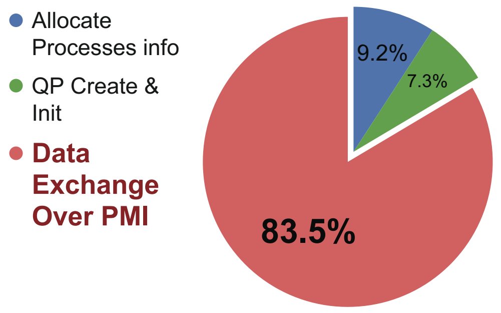


Fig. 5. Total time percentage of different operations during all-to-all RDMA connection startup, run with up to 16 nodes 56 PPN (Process Per Node) on TACC Frontera

A. Challenges

To address the high cost of data exchange via PMI, our approach focuses on reducing the time overhead and amount of data associated with exchanging data between processes via PMI. Referring to our initial investigation, there are two types of information exchanged via PMI in all-to-all connection establishment – HCA type and local ID (lid), Queue Pair number (qp_num). The HCA information is identical for the same HCA, but the queue numbers vary and serve as unique identifiers for queue pairs created by the same HCA. Given that understanding, our first challenge is ***How we can minimize the data, especially the queue pair numbers exchanged via PMI for optimizing all-to-all connection establishment***.

To minimize the amount of data exchanged via PMI, we investigate techniques such as data compression and efficient data packaging. However, those techniques may create new overheads. For instance, our proposed design increases queue pair creation overhead, detailed discussion in the following section IV-B. While addressing the high cost of data exchange via PMI is crucial, it is equally important to carefully consider the trade-offs of data exchange via PMI cost and the other costs such as queue pair creation. Therefore, our second challenge is ***How to utilize the system characteristics to balance different types of overhead in the all-to-all connection establishment process.***

While the all-to-all startup design establishes connections in MPI_Init step, the on-demand startup method establishes connections during execution. Both all-to-all and on-demand connection establishment designs have their own advantage in different MPI communication patterns. As stated in section I that a significant majority of parallel jobs involve no more than 8 nodes. It is important to evaluate our optimized all-to-all startup and on-demand startup with different MPI communication patterns and applications. This brings us to the third challenge: ***How can we select the best connection establishment approach to benefit the overall HPC application performance.***

B. Contributions

We analyze the existing all-to-all connection establishment in-depth by program profiling, to identify that the most costly operations are data exchange via PMI. By highlighting the bottlenecks of existing design and leveraging an observed system feature, we proposed an optimized design for all-to-all connection establishment. Then we investigate the potential enhancement of our proposed design by incorporating thread-level parallelism and CPU-affinity awareness. This demonstrates an additional layer of optimization to further improve the performance and scalability of the proposed all-to-all connection establishment design. We integrate our designs in a popular MPI library MVAPICH2 and show the efficacy of our proposed design through a systematical performance evaluation. Our proposed optimized design is able to establish all-to-all connection **20**$\times$ faster.

To summarize, We list the following key contributions in this paper:

- Analyze limitations of the current all-to-all connection establishment design and the main reasons for the lack of scalability.
- Propose an optimized all-to-all connection establishment design with identified system features and hardware behavior.

- Propose two enhancement approaches to improve MPI startup and overall performance.
- Demonstrate the efficacy of the proposed design on real systems using micro-benchmarks and applications. Our optimized design outperforms state-of-art solutions by up to **20**$\times$ in alltoallv microbenchmark and up to **31%** in NPB NAS-FT, respectively.
- Evaluate our optimized all-to-all connection establishment design with comparison to state-of-art on-demand design and analyze the best use cases.

III. Background

In this paper, we implement our optimized design based on MVAPICH2 2.3.7 [6] for our experiments and performance evaluations. However, our observations in this context are quite general and they should be applicable to other high-performance MPI libraries as well.

A. InfiniBand

InfiniBand [7] is a high-performance, multi-purpose network architecture that provides features including high throughput, low latency, quality of service, and failover. It is commonly used in data centers, high-performance computing (HPC) environments, and enterprise-level storage systems due to its low latency and high bandwidth capabilities. A key component of the InfiniBand architecture is the Queue Pair (QP). Each QP consists of a pair of queues: one for sending and one for receiving data. InfiniBand uses a different approach to the classic network card approach, where the operating system kernel manages data transmission. Instead, InfiniBand uses Direct Memory Access (DMA) to read or write data directly to or from the application's memory, bypassing the need for the operating system's intervention. This approach minimizes latency and maximizes bandwidth. The process of creating a QP involves allocating and initializing the queue pair within the InfiniBand device. The queue pair then gets associated with a specific protection domain that provides the necessary security guarantees for the queue pair's operation.

B. RC Connection Establishment

Reliable Connection (RC) and Unreliable Datagram (UD) are two different types of transport services specified in the InfiniBand architecture. RC is a connected mode of operation that provides reliable, in-order delivery of packets. It sets up a connection between two Queue Pairs (QPs) and guarantees the delivery of packets without loss or errors. It accomplishes this using acknowledgment packets (ACKs) and packet retransmission in case of packet loss. The communication is point-to-point, i.e., between two QPs. Therefore, RC requires a connection to be established before any data can be sent. This process includes initializing Queue Pairs (QPs), transitioning them through multiple states (INIT, RTR, RTS), and the exchange of connection request and reply messages. The time and computational resources required to accomplish these steps contribute to the overhead of RC.

IV. Design

To address the challenges in section II, we propose an optimized all-to-all MPI startup design. Our design consists of 3 major components – Firstly, we propose a rank-by-rank all-to-all connection establishment method to significantly reduce the large amount of time consumed by information exchange over PMI. Secondly, we parallelize the queue pair creation step with OpenMP to mitigate the additional overheads made by the optimized all-to-all connection establishment. At last, we optimize the CPU affinity to maximize the processor resource utilization for MPI jobs.

A. Rank-by-rank All-to-all Connection Establishment

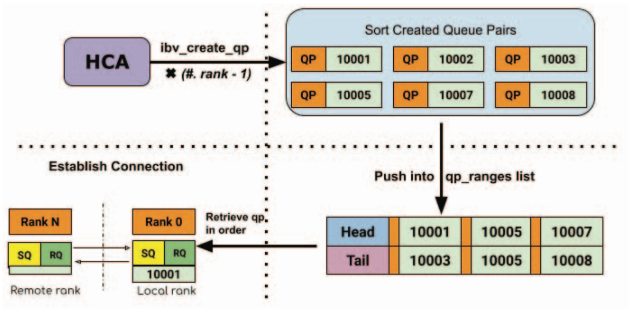


Fig. 6. Example: Steps of creating and allocating QPs to local rank for establishing connection to other ranks

In section II, we have discussed that a large amount of time is spent on exchanging queue pair, HCA local ID (lid), etc, over PMI. In our investigations on different HPC systems, we observe that the queue pair numbers return by *ibv_create_qp* calls are mostly sequential as long as one process continues to make create queue pair calls before any other process breaks in between. This behavior enlighten us to enforce ordering between different MPI ranks or processes in the same node, so that one process can create all endpoints, and then the next MPI rank or process will do it. This design will guarantee some range of sequential queue pairs for each process to establish connection to other processes. With guaranteed queue pair ranges, a process only need to exchange the initial queue pair information instead of all of them.

The Figure 6 depicts an example procedure of our proposed sequential all-to-all connection establishment. When a local process starts to establish connection to its target processes, it begins by performing necessary initialization. Subsequently, the process calls *ibv_create_qp* to create one queue pair for each target process. To ensure synchronization, we implement a local PMI barrier to the create queue pair loop, allowing only one process making *ibv_create_qp* calls on each node at any given time. The target processes count of created queue pairs' pointers will be stored in a local buffer called *qp_pool*. As Figure 6 showing, while the returned *qp_num*'s are consistently contiguous, there are instances where certain *qp_num* get skipped by HCA. To address this, we store all the contiguous *qp_num* values into a list called qp_ranges, and each entry consists of initial and tail *qp_num* of each continuous list of *qp_num*. Then the full qp_range list are exchanged to other processes over PMI, and local process can retrieve sequential list of queue pair by *qp_num* to establish connection. Figure 7 shows the example of information exchanged between processes via PMI. The data exchange via PMI consists of two primary step: First, hardware information including architecture type and HCA type is exchanged to identify if the connection can be established. Next, HCA local ID and create queue pair numbers are exchanged in order to establish actual connections.

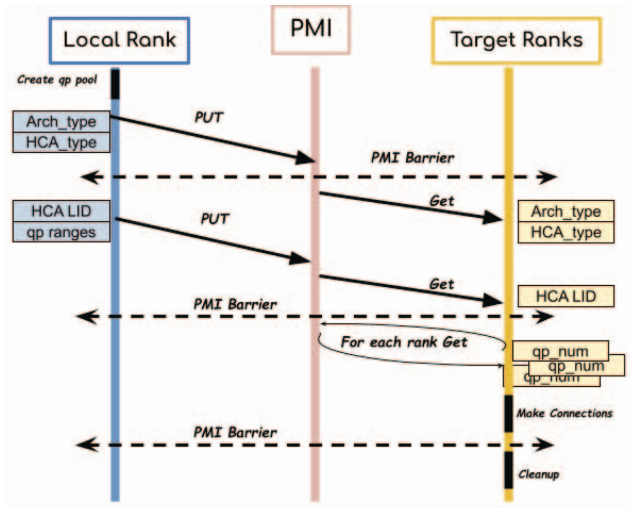


Fig. 7. The proposed design exchanges following information between MPI processes via PMI: 1. System architecure & HCA type 2. HCA LID 3. List of *qp_num* ranges PMI barriers are put between for synchronization

In Algorithm 1, we present the procedure of sequential all-to-all connection establishment design in high level.

B. Parallelized Queue Pair Creation and Initialization

Despite the significant amount of time reduction in data exchange over PMI achieved by sequential all-to-all connection establishment, the in-order queue pair creation itself can be quite costly. Figure 8 shows that up to 78% of total MPI_Init execution time is consumed by queue pair creation and initialization. This is primarily due to the restriction that only one process can operate at a time, which becomes especially problematic for emerging HPC systems with a large number of cores per processor. To address this issue, we parallelize the creating queue pair portion using OpenMP [8].

```
int qp_pool_size = GetMPIJobSize();
struct ibv_qp* qp_pool[qp_pool_size];
Initialize_attributes(qp_attr);

// Multi-threaded QP creation and initialization
#pragma omp parallel for num_threads(N)
for (int i=0; i<job_size; i++) {
    qp_pool[i]=ibv_create_qp(..., &qp_attr);
    ... ...
    // modify qp to INIT state
    ibv_modify_qp(qp_pool[i], ...);
}

// In case that qp_num out of order
Sort_by_qp_num(qp_pool, qp_pool_size);
```

Listing 1. Multi-threaded QP Creation

Algorithm 1: Rank-by-rank All-to-all Connection Establishment

```
Function OptA2aConnEst($rank$):
    PrepareConnection($gid$, $hca$, ...)
    Initialize($local\_qp\_ranges$, $local\_qp\_pool$)
    foreach $localprocess$ in $node$ do
        if $localProcess == rank$ then
            $qp\_ptr = ParallelCreateQP(rank)$
            Push_to($local\_qp\_pool$, $qp\_ptr$)
            Add_to($local\_qp\_ranges, qp\_ptr \rightarrow qp\_num$)
        PMI_LOCAL_BARRIER()
    PushToPMI($HCAInfo, rank$)
    foreach $tgtRank$ in $AllRanks$ do
        tgtHCAInfo = GetFromPMI($HCAInfo$)
    PushToPMI($qp\_ranges, rank$)
    foreach $tgtRank$ in $targetRanks$ do
        $qp\_ranges$ = GetFromPMI($HCAInfo$) $i$ = $tgtRank$
        foreach $entry$ in $qp\_ranges$ do
            if $entry.tail - entry.head \leq tgtRank$ then
                $Connect(tgtRank, entry.head + i)$
                $break$
            else
                $i$ -= $entry.tail - entry.head + 1$
    AllocateRDMABuffer($rank$, ...)
    EnableAllQueuePairConnections($rank$, ...)
    Cleanup($local\_qp\_ranges, local\_qp\_pool$)
    return
```

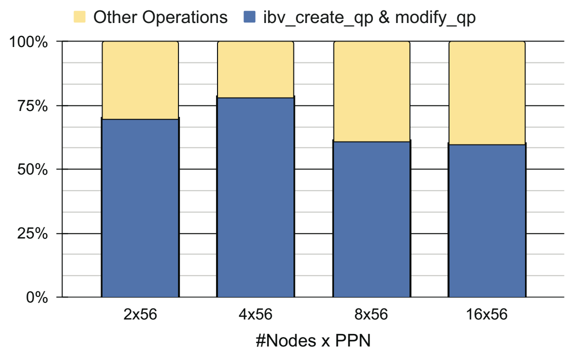


Fig. 8. Profiling the percentage of time spent on queue pair creation and initialization vs. other operation in MPI_Init with sequential all-to-all connection establishment, run with up to 16 nodes 56 PPN on TACC Frontera

The pseudo-code in Listing 1 presents the high-level implementation of parallelized queue pair creation and initialization. The queue pair *create* and *modify* (to INIT state) calls are wrapped in an OpenMP for loop, which iterates through all processes in MPI_COMM_WORLD. In case that *qp_nums* are returned out-of-order by *ibv_create_qp* in different threads, *qp_pool* will be sorted after this parallel for loop.

To explore the optimal number of OpenMP threads for multi-threaded queue pair creation, we conducted experiments, varying the number of nodes (#Nodes) and processes per node (PPN) on TACC Frontera with up to 16 compute nodes. We compare the time consumed by the multi-threaded proposed all-to-all connection design, varying from 1 to 28 (maximum number of cores in each CPU on TACC Frontera) threads. The results in Figure 9 indicate that 8 to 16 OpenMP threads yield the best performance for our multi-thread design. When compared to the serial version of the sequential all-to-all connection establishment described in section IV-A, our optimized approach demonstrates a significant reduction of up to 75% in the queue pair creation and initialization time.

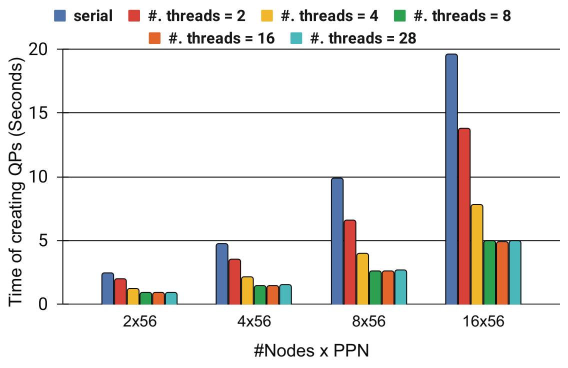


Fig. 9. Comparing performance of multi-threaded QP creation & initialization with different number of OpenMP threads, run with up to 16 nodes 896 processes on TACC Frontera

C. Enhanced Affinity-aware All-to-all Startup

The multi-threaded design we discussed in section IV-B utilizes the multi-thread parallelism on multi/many-core systems like TACC Frontera to accelerate the queue pair creation step in our proposed sequential all-to-all connection establishment design. However, modern MPI libraries typically incorporate some built-in CPU pinning policies to bind processor cores to their processes. For example, OpenMPI has cpu-pinning options like map-by-socket or -by-l3cache option, and MVAPICH2 has run-time CPU binding policy to map processes linearly or by each NUMA domain. Many of the communication algorithms, including topology-aware collective algorithms, rely on this CPU pinning. In a scenario where an MPI job fully occupies a compute node, each processor core is pinned to a specific process. Consequently, our multi-threaded queue pair creation approach would not be able to take advantage of parallelism across multiple cores. Disabling the CPU pinning step is a straightforward solution, but it would prevent MPI libraries from leveraging their built-in communication algorithms. In our multi-threaded optimized all-to-all connection establishment design, we address this challenge by configuring affinity after the inter-node startup step.

V. Performance Evaluation

To demonstrate the efficacy of the proposed rank-by-rank all-to-all connection establishment design, and identify the suitable use cases for this design, we conduct a comprehensive evaluation at both the micro-benchmark level and application level. All the experiments are executed with a maximum of 16 compute nodes since a significant majority of the HPC workloads involve no more than 8 nodes, as we show in Figure 3(b). Within this scale, we observe remarkable improvements when comparing our optimized design to the existing all-to-all connection establishment approach, which is pervasively employed in this scale. For completeness, we also evaluate the state-of-the-art on-demand design, providing a more comprehensive view of the connection establishment performance across different approaches within this scale. The evaluation is conducted using the latest HDR-100G or HDR-200G adapters, and we anticipate similar behavior on older generation adapters like EDR, FDR, QDR, etc.

A. Experimental Setup

We evaluated our designs on two different state-of-the-art architectures. Texas Advanced Supercomputing Center (TACC) Frontera [9] system and Lonestar6 [10] system. TACC Frontera system is configured with dual-socket Intel Xeon Platinum 8280 Cascade Lake CPU, while Lonestar6 system employs dual-socket AMD EPYC 7763 processor. Each compute node of Frontera consists of 56 physical cores, while each node on the Lonestar6 consists of 128 physical cores. We use the MVAPICH2-2.3.7 MPI library for all our evaluations including OSU-Microbenchmark (OMB) v5.9 for micro-benchmark evaluation. Each OMB test was run for 1 iteration after MPI_Init and the average of 5 is reported. For application-level evaluation, we use NPB-3.4.2 [11] and 3D-stencil [12] application kernels.

TABLE I
Hardware specification of different tested clusters

Specification	Frontera	Lonestar
Processor Family	Intel Cascade Lake	AMD EPYC
Processor Model	Xeon Platinum 8280	EPYC 7763
Clock Speed	2.7 GHz	2.45 GHz
Sockets	2	2
Cores Per socket	28	64
NUMA nodes	2	2
#Cores Per NUMA	28	64
RAM (DDR4)	192 GB	256 GB
Interconnect	IB-HDR(100G)	IB-HDR(200G)

B. Microbenchmark Results

In this section, we present a comprehensive performance evaluation of our proposed optimized all-to-all connection establishment design, as described in Section IV. To validate the efficacy of our design, we compare its performance against state-of-the-art on-demand and all-to-all connection establishment designs integrated into the latest MVAPICH2 MPI library.

To evaluate the performance of these connection establishment designs across various communication patterns, we execute all the MPI collective tests in OMB and particularly choose three typical collective benchmarks: osu_alltoallv, osu_allreduce, and osu_bcast. These benchmarks show the behavior of different connection establishment designs under these three typical communication patterns. The results are shown from Figure 10 to 13.

During the execution of the collective benchmarks, we collect two types of measurements: the real execution time, measured using the Linux $time$ command, and the communication time, measured by the benchmarks for a single iteration. We chose not to use the osu_init benchmark for measuring the performance of the on-demand design. The reason for this is that the on-demand design may require establishing connections after the MPI_Init function is called, which cannot be accurately measured by the osu_init benchmark alone. By collecting both the real execution time and the communication time, we obtain a comprehensive understanding of the overall execution performance as well as the specific communication overhead incurred by the connection establishment designs. This allows us to assess the effectiveness of our proposed optimized all-to-all connection establishment design and compare it to the existing on-demand and all-to-all designs.

*1) **Performance Evaluation of Connection Establishment**:* Figure 10 shows the time consumed to establish connections by the three designs. We observe in Figure 10(a) that our optimized all-to-all connection establishment design is up to $20\times$ faster than the existing all-to-all design, and reduces 8% connection establishment time comparing to on-demand design on 16 nodes 892 processes scale. The connection establishment time of existing all-to-all design exponentially steeply increases while the scale grows larger than or equal to 8 nodes 448 processes. As we discussed in Figure 4, the steep increase is primarily caused by the exponential growth in data exchange via PMI. Similarly, in Figure 10(b) and 10(c), both our optimized all-to-all connection establishment approach and the existing all-to-all design remain the same behavior during the connection establishment process. As a result, their performance follows the same trend as Figure 10(a). The proposed design is faster than the existing all-to-all design by a factor of $21\times$ in the Allreduce and Broadcast benchmark.

To validate the adaptability of our proposed design across different architectures, we conduct similar experiments and evaluations as Figure 10 on the TACC Lonestart6 system. The results are shown in Figure 12. One thing to be noted for Lonestar6 is that each of the compute nodes contains 128 physical cores. While the maximum queue pair number on each node is 131072 ($= 8 \times 128 \times 128$), so we will exhaust the queue pair limitation with 8 or more than 8 nodes. Therefore on Lonestar, we conduct experiments with scale up to 4 nodes 512 processes. The optimized all-to-all connection establishment design is faster than the existing all-to-all design by a factor of $2.3\times$ in Alltoallv benchmark, $2.5\times$ in Allreduce benchmark, and $2.4\times$ in Broadcast benchmark. On the other hand, the existing all-to-all and on-demand connection establishment design follow the same trend as what we observed in our tests on TACC Frontera.

*2) **Performance Evaluation of Communication Latency**:* In addition to evaluating the specific connection establishment time presented in Figure 10 and 12, we analyze the communication time of a single iteration of MPI collective communication operations after MPI_Init step. The results are shown in Figure 11 and 13. As we mentioned earlier, the on-demand startup design may need to establish connections after the MPI_Init step. This feature is demonstrated by the Alltoallv benchmark result shown in Figure 12(a). The first iteration of MPI_Alltoall operation of on-demand design spends $448\times$ longer communication time than all-to-all connection design on 16 nodes 896 processes scale. The huge gap here is attributed to the costly communication establishment of on-demand startup design during the execution time. On the other hand, our proposed design has almost no side effect communication operations after MPI_Init step, which leads to the close communication latency of our proposed design and existing all-to-all design. However, for the communication patterns with lower complexity or requirement of connection numbers, such as Allreduce and Bcast with results shown in Figure 12(b) and 12(c), we are not able to see huge communication latency difference. It is worth noting that the communication latency degradation of on-demand design in Figure 12(c) is attributed to the MPI_Bcast algorithm adjustment after a certain job size, which requires additional connections and leads to the performance overhead.

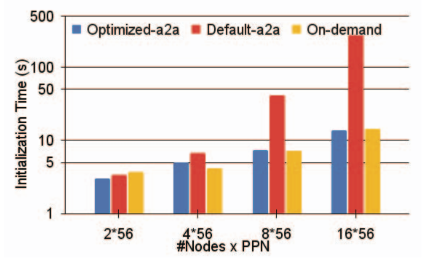
The same set of communication latency evaluations is also conducted on the Lonestar6 system. Figure 13(a) shows that the first iteration MPI_Alltoall communication latency of the proposed all-to-all connection establishment design is up to $162\times$ lower than on-demand design with 4 nodes 512 processes scale. The MPI_Allreduce requires less connection establishment than MPI_Alltoall on Lonestar6, therefore we observe optimized and existing all-to-all design have up to $3.1\times$ lower latency than on-demand design.

Observation: Based upon the analysis of the Micro-benchmark results across two different systems, we are able to identify the optimal use cases for our optimized all-to-all connection establishment design. These use cases are characterized by MPI jobs with a high level of complexity in inter-process communication (e,g, MPI_Alltoall), with a scale that does not exceed than few thousand processes or exhaust maximum queue pairs. On the other hand, for simple communication patterns or extremely large job size, on-demand connection establishment will be the optimal choice.

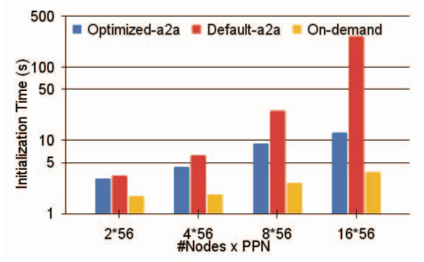
C. Application Results

To demonstrate the efficacy of our optimized connection-establishment design on real application, we evaluate the performance of the proposed design with two mini application kernels to simulate the real use cases.

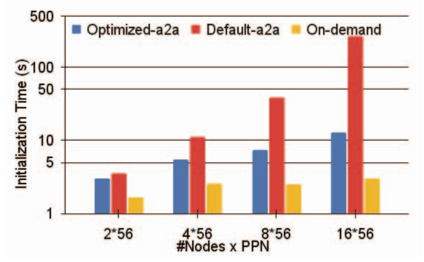
*1) **3D-Stencil (Localized neighborhood communication pattern)**:* In every iteration data is received in a predefined pattern and is used by the kernel to update the elements. 7 point stencil is used in this benchmark which involves exchanging data with six neighboring processes. To achieve this in every iteration each process posts MPI_Irecv for all the messages it is expected to receive and then posts all of its MPI_Isend calls. It uses MPI_Waitall to wait for the completion of all



(a) Alltoallv

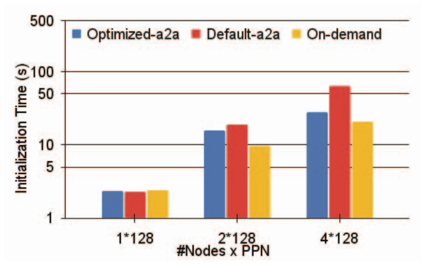


(b) Allreduce

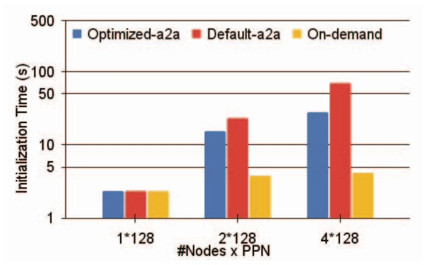


(c) Bcast

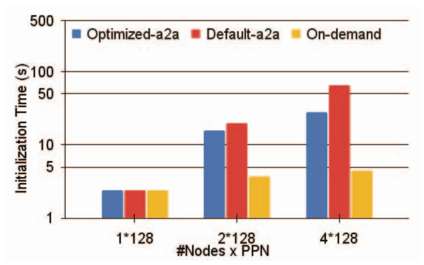
Fig. 10. Total time consumed by different approaches to establish connections for several popular MPI collective primitives, run with up to 16 nodes 896 processes on TACC Frontera



(a) Alltoallv

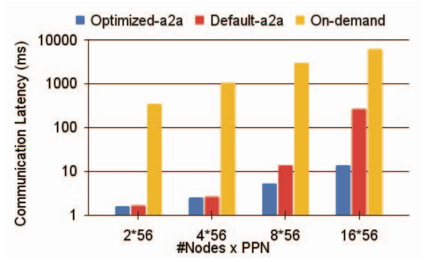


(b) Allreduce

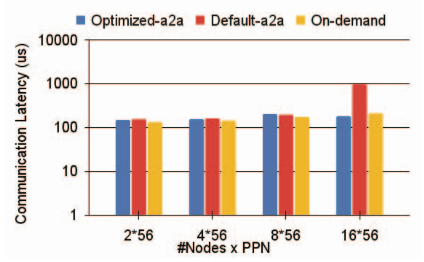


(c) Bcast

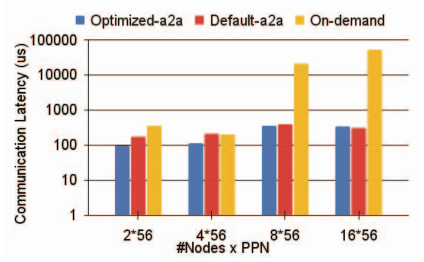
Fig. 11. Total time consumed by different approaches to establish connections for several popular MPI collective primitives, run with up to 4 nodes 512 processes on TACC Lonestar6



(a) Alltoallv

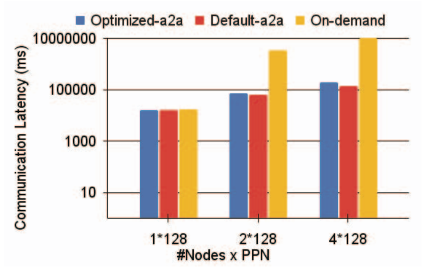


(b) Allreduce

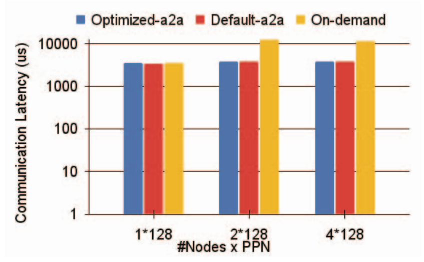


(c) Bcast

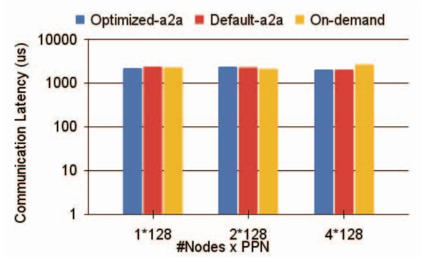
Fig. 12. First iteration communication latency of several popular MPI collective primitives after different approaches of connection establishment, run with up to 16 nodes 896 processes on TACC Frontera



(a) Alltoallv



(b) Allreduce

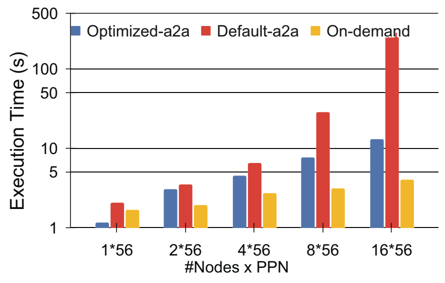


(c) Bcast

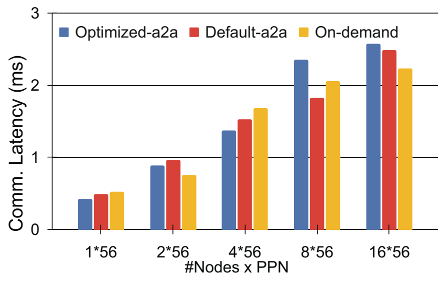
Fig. 13. First iteration communication latency of several popular MPI collective primitives after different approaches of connection establishment, run with up to 4 nodes 512 processes on TACC Lonestar6

the sends and receives and finally uses MPI_Allreduce to collect boundary information from the participating processes. We run 3D-stencil with 10 iterations of communication with measurement of program total execution time and average communication latency of each process.

Figure 14(b) shows the total execution time of 3D-stencil application on TACC Frontera with various scales up to 16 nodes 896 processes. One thing worth to be noted that the vertical axis is in logscale. We observe that the existing all-to-all connection establishment design ends up increased startup overhead with 8 and more than 8 nodes. Our optimized all-to-all connection establishment design is able to be **10**$\times$ faster than the existing design. Due to the communication pattern of 3D-stencil, the on-demand connection establishment design maintains advantage in all the scales as it selectively establish necessary connections. Figure 14(b) shows the average communication latency of those three design. Similar to the trends we observed in micro-benchmark evaluation, they have close communication latency as no additional connection needs to be built by on-demand design.



(a) Total Execution Time

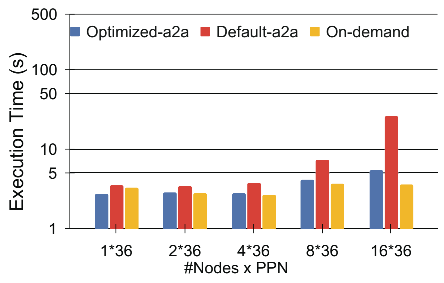


(b) Avg. Communication Latency

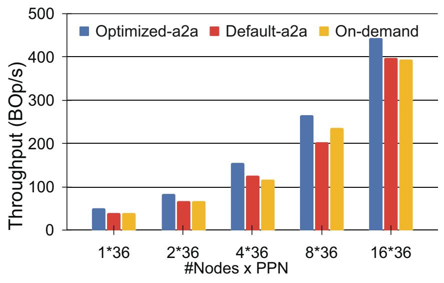
Fig. 14. Total execution time and average communication latency of 3D-stencil application kernel, run with 10 iterations, varying scales up to 16 nodes 896 processes on TACC Frontera

*2) **NPB NAS-FT (Collective MPI communication pattern)**:* NAS parallel benchmarks [13] are developed for benchmarking highly parallel systems with respect to computation, communication, memory usage, etc. NAS-FT is A 3D partial differential equation (PDE) solution using FFTs. This kernel numerically solves a PDE using both forward and inverse FFTs. Long-distance communication performance is rigorously tested by this benchmark.

Figure 15(a) shows the total execution time of NAS-FT (Class B) benchmark on TACC Frontera with up to 16 nodes 512 processes. The vertical axis is in log scale. We observe similar performance trends as 3D-stencil and OMB. Our proposed design exhibits **4.8%** faster execution times than the existing all-to-all connection establishment design. Figure 15(b) shows the program throughput by total (billion) operations of all processes per second. Different from 3D-stencil, the dominant communication pattern in NAS-FT is MPI_Alltoall. Hence we observe up to **31%** higher throughput of the proposed all-to-all connection establishment design compared to on-demand design.



(a) Total Execution Time



(b) Total Billion Op. per Second

Fig. 15. Total execution time and throughput of total Billion of Operations of NPB NAS-FT (Class-B) application kernel, varying scales up to 16 nodes 512 processes on TACC Frontera

Observation: The analysis of optimal use cases discussed at the end of Section V-B is validated in application-level performance evaluation. The optimal use cases for the optimized all-to-all connection establishment are the MPI applications with a high level of complexity in inter-process communication (e,g, MPI_Alltoall).

VI. Related Work

Improving MPI over IB has seen significant contributions from various researchers and in several MPI implementations. Besides groundbreaking fundamental research on supporting IB RDMA communication for MPI and optimizing RC and UD communication support [14]–[17], a few notable related works are discussed here. A key contribution is the MVAPICH-Aptus, a multi-transport MPI design that uses both the RC and UD transports of IB to deliver scalability and performance higher than that of a single-transport MPI design [18]. The MVAPICH-Aptus design has shown a 12% improvement over an RC-based design and 4% better than a UD-based design for the SMG2000 application benchmark.

For the molecular dynamics application NAMD, it showed a 10% improvement over an RC-only design.

Another significant contribution in this area is the work on a high-performance UD-based MPI design over IB [19]. This design addresses the issue of increasing memory requirements in RC-based implementations as clusters continue to scale. The connection-less UD transport eliminates the need to dedicate memory for each pair of processes, making it an attractive alternative. The design was implemented and compared with the RC-based MVAPICH in terms of performance and resource usage. The evaluation showed a 60% speedup and a seven-fold reduction in memory for 4K processes for the SMG2000 benchmark. The design also estimated a 30 times reduction in memory over MVAPICH at 16K processes when all connections are created.

The work from Chakraborty et al. [20], on the other hand, focuses on the efficient implementation of the Process Management Interface (PMI), which is crucial for enabling fast start-up of MPI jobs. The authors propose three extensions to the PMI specification: a blocking all-gather collective (PMIX_Allgather), a non-blocking all-gather collective (PMIX_Iallgather), and a non-blocking fence (PMIX_KVS_Ifence). They design and evaluate several PMI implementations to demonstrate how such extensions reduce MPI start-up costs. 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 start-up schemes.

In contrast to the existing works in the literature, our study takes a unique approach to improving RDMA all-to-all connection establishment for RC in MPI. While previous works have focused on different related aspects, our work delves deeper into the pervasive scheme of all-to-all connection establishment design used in RC transports. Due to its robust performance and reliability, we recognize that RC is a default communication channel for most MPI over IB jobs with small to medium message sizes (before running out of QPs). Hence, optimizations for these workloads are of paramount importance, and our work addresses this need directly.

VII. Conclusion and Future Work

In modern multi-/many-core HPC systems, the increasing number of processor cores leads to new challenges in orchestrating parallel compute workloads across multiple nodes. Among the various aspects that affect the startup phase of parallel MPI jobs, the communication establishment methodology becomes a crucial factor that warrants in-depth investigation and research. In this paper, we investigate the bottleneck of existing state-of-the-art all-to-all connection establishment designs, identify the primary source of performance overheads and propose an optimized process-by-process connection establishment design. Furthermore, we explore multi-thread parallelism and incorporate CPU affinity awareness into our proposed design to mitigate the side effects of increasing QP creation and synchronization overheads. Finally, to demonstrate the efficacy of our proposed design, we implement it in a state-of-the-art MVAPICH2 MPI library and conduct comprehensive experiments across two different emerging architectures. Through a systematical performance evaluation across two emerging HPC architectures, comparing to existing all-to-all connection establishment design, we observe up to **20**$\times$ faster MPI_Init time in microbenchmarks level and **31**% throughput with NPB NAS-FT. In the future, we plan to extend our optimized all-to-all connection establishment design to other architectures (e.g., ARM) or network adapters (e.g., Broadcom RoCEv2).

References

[1] "Message Passing Interface (MPI)," http://www.mpi-forum.org. Accessed: November 5, 2023.

[2] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff, "Mpi at exascale," *Procceedings of SciDAC*, vol. 2, pp. 14–35, 2010.

[3] National Science Foundation (NSF), 2023. [Online]. Available: https://www.nsf.gov/

[4] XSEDE, 2023. [Online]. Available: https://www.xsede.org/

[5] D. Panda et al., "OSU microbenchmarks v5.6.3," http://mvapich.cse.ohio-state.edu/benchmarks/.

[6] Network-Based Computing Laboratory, "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE," http://mvapich.cse.ohio-state.edu/.

[7] G. Pfister, "Aspects of the InfiniBand(tm) Architecture," in *2001 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2001, p. 369.

[8] OpenMP Architecture Review Board, "OpenMP API specification," 2021. [Online]. Available: https://www.openmp.org/specifications/

[9] Texas Advanced Computing Center (TACC), "Frontera," 2023. [Online]. Available: https://www.tacc.utexas.edu/systems/frontera/

[10] TACC, "Lonestar6," 2023. [Online]. Available: https://www.tacc.utexas.edu/systems/lonestar6/

[11] "NAS Parallel Benchmarks," http://www.nas.nasa.gov.

[12] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. Panda, "High-performance and scalable non-blocking all-to-all with collective offload on infiniband clusters: A study with parallel 3d fft," June 2011.

[13] W. Yu, N. S. Rao, and J. S. Vetter, "Experimental Analysis of InfiniBand Transport Services on WAN," in *Networking, Architecture, and Storage, 2008. NAS'08. International Conference on*. IEEE, 2008, pp. 233–240.

[14] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," in *17th Annual ACM International Conference on Supercomputing*, June 2003.

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

[16] M. J. Koop, S. Sur, and D. K. Panda, "Zero-copy Protocol for MPI using InfiniBand Unreliable Datagram," in *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 179–186.

[17] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," in *Int'l Conference on Parallel Processing (ICPP '11)*, 2011.

[18] M. J. Koop, T. Jones, and D. K. Panda, "MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand," in *IPDPS'08*, 2008, pp. 1–12.

[19] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 180–189.

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