On-demand Connection Management for OpenSHMEM and OpenSHMEM+MPI

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

***Abstract*—Partitioned Global Address Space (PGAS) programming models like OpenSHMEM and hybrid models like OpenSHMEM+MPI can deliver high performance and improved programmability. However, current implementations of OpenSHMEM assume a fully-connected process model which affects their performance and scalability. We address this critical issue by designing on-demand connection management support for OpenSHMEM which significantly improves the startup performance and reduces the resource usage. We further enhance the OpenSHMEM startup performance by utilizing non-blocking out-of-band communication APIs. We evaluate our designs using a set of microbenchmarks and applications and observe 30 times reduction in OpenSHMEM initialization time and 8.3 times improvement in execution time of a Hello World application at 8,192 processes. In particular, when sufficient work can be overlapped, we show that use of non-blocking out-of-band communication APIs allow for a constant initialization cost of OpenSHMEM jobs at different core counts. We also obtain up to 90% reduction in number of network endpoints and up to 35% improvement in application execution time with NAS Parallel Benchmarks.**

***Index Terms*—On-demand Connection Management; OpenSHMEM; PGAS; Job Launch; InfiniBand**

I. Introduction and Motivation

Fast and scalable startup is an often overlooked but important aspect of High Performance Computing (HPC) jobs. 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.

While Message Passing Interface (MPI) [1] has been the dominant programming model in the HPC world, MPI does not lend itself well for writing all types of parallel applications. As an example, applications with irregular communication patterns such as Graph500 [2] are considered well suited for the Partitioned Global Address Space (PGAS) [3] programming model. OpenSHMEM [4] is a popular library based implementation of the PGAS model, which can often improve programmability while providing similar or better performance compared to MPI. As a result, PGAS and OpenSHMEM have recently seen increased adoption in the HPC community. Hybrid MPI+PGAS models are gaining popularity as they enable developers to take advantage of the PGAS model in their MPI applications, without having to rewrite the complete application [5, 6]. The Exascale roadmap identifies the hybrid model as the ‘practical’ way of programming Exascale systems [7]. Unified communication runtimes, like MVAPICH2-X [8], are enabling the efficient use of these hybrid models by consolidating resources normally used by two different runtimes, thereby providing performance, scalability, and efficient resource utilization. They also prevent deadlocks arising from independent progress of different runtimes [9].

High-performance OpenSHMEM implementations for InfiniBand [10] use the GASNet communication middleware, specifically the GASNet-ibv conduit [11] or the GASNet-mvapich2x conduit [8] to interface with the underlying network. InfiniBand is a widely used industry standard high-speed interconnect which provides low latency, high bandwidth, one-sided and atomic access to remote process’s memory. The one-sided operational model defined by OpenSHMEM can be easily mapped to the features provided by the InfiniBand network. InfiniBand supports both connection-oriented (Reliable Connected - RC) and connection-less (Unreliable Datagram - UD) transport protocols [12, 13]. However, many of the useful features like reliability and atomic access are only available in the RC protocol. A drawback of the RC protocol is that each connection requires two Queue Pairs (QP) and associated structures which can consume a significant amount of memory in large fully-connected process groups. In all existing InfiniBand based high-performance implementations of OpenSHMEM and GASNet, each process creates N IB endpoints (QPs) and connects to all N processes (including itself) taking part in the OpenSHMEM application during initialization. The total number of connections opened is $\frac{N^2}{2}$ and the number of QPs created is N^2.

Although this connection model is simple to implement, it adversely affects the scalability and performance of the application in following ways:

1) Establishment and teardown of a fully connected network is a costly operation. In fact, at large scale this becomes the dominant factor in time taken to launch and terminate an OpenSHMEM program (Startup Performance). Figure 1 shows the amount of time consumed by different steps in OpenSHMEM initialization, as

*This research is supported in part by National Science Foundation grants #OCI-1148371 and #CCF-1213084.

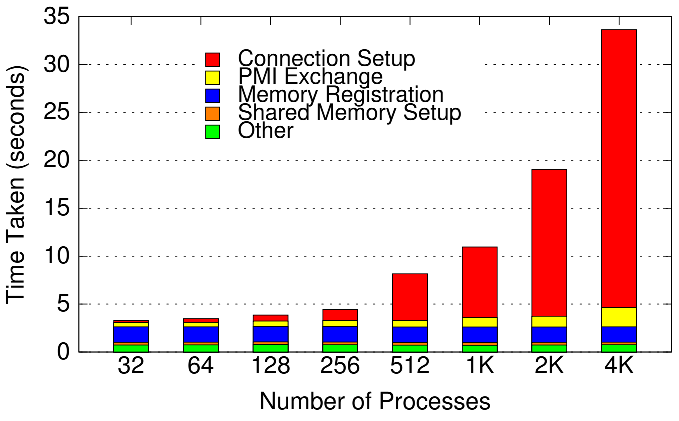


Fig. 1. Breakdown of time spent in OpenSHMEM initialization with 16 processes per node

measured on Cluster-B (described in section V-A).

2) The number of connections opened and QPs created increase quickly as the process count grows and contributes to the memory pressure. This problem is only going to be exacerbated by the newer generation hardware as number of cores per node continues to grow.
3) Most current generation network interface cards (HCA) have limited on-board memory to cache information about recently used endpoints and connections. With a large number of connections created on each HCA, their performance can be negatively impacted [12, 13].
4) As prior research shows, most HPC applications only communicate with a subset of its peer processes. Thus establishing alltoall connectivity for all HPC applications is unnecessary and wasteful. Our analysis shown in Table I also confirms this pattern for pure OpenSHMEM as well as hybrid OpenSHMEM+MPI applications.

TABLE I
AVERAGE NUMBER OF COMMUNICATING PEERS PER PROCESS FOR DIFFERENT APPLICATIONS. THIS IS DEPENDENT ON BOTH POINT-TO-POINT AND COLLECTIVE OPERATIONS.

Application	Number of Processes	Average Number of Peers
BT	64 1024	8.7 10.6
EP	64 1024	3 5.01
MG	64 1024	9.46 11.9
SP	64 1024	8.75 10.7
2D Heat	64 1024	5.28 5.40

The second major component in the startup of OpenSHMEM jobs is the cost associated with exchanging information within the runtime that is needed to initialize high-performance IB communication channels between processes in the job. For portability, most job launchers provide a common "out-of-band" communication infrastructure known as the Process Management Interface (PMI) [14]. Current implementations of PMI scale poorly on today's largest systems – a problem that will only be exacerbated on next generation exaflop systems that are expected to have O(100,000) to O(1,000,000) hosts [15]. Prior research from the authors as well as other groups [16–18] have identified the need for and the benefits of using non-blocking PMI APIs for improving the startup of MPI-based parallel applications on large supercomputing systems. However, existing OpenSHMEM implementations do not take advantage of these APIs to enhance the startup performance of pure OpenSHMEM or hybrid OpenSHMEM+MPI applications.

These issues lead us to the following broad challenge — **Can we enhance the existing OpenSHMEM runtime design to improve the startup time and scalability of pure OpenSHMEM and hybrid OpenSHMEM+MPI parallel applications on large supercomputing systems?**

In this paper, we take up this challenge, and propose two major designs changes to the OpenSHMEM runtime to address them: 1) an on-demand connection establishment and data exchange scheme and 2) utilizing non-blocking allgather collective (PMIX_Iallgather) for "out-of-band" exchange of IB end point information at startup.

II. CONTRIBUTIONS

To summarize, this paper makes the following contributions:

- Propose, design and implement on-demand connection establishment scheme for high performance OpenSHMEM runtimes
- Utilize the non-blocking allgather collective (PMIX_Iallgather) for "out-of-band" exchange of IB endpoint information at startup scheme for high performance OpenSHMEM runtimes
- Evaluate the benefits the new designs have on performance at the microbenchmark level using OpenSHMEM microbenchmarks and at the application level using NAS parallel benchmarks

Figure 2 summarizes the benefits our proposed design brings to different aspects of job performance, namely, resource usage, startup time and application performance. The proposed designs lead to significant benefits in terms of resource usage and startup time and moderate benefits in terms of overall application performance. We also evaluate our designs using a set of microbenchmarks and applications and observe 30 times and 8.3 times improvement in initialization and startup performance respectively at 8,192 processes. In particular, when sufficient work can be overlapped, we show that the use of non-blocking out-of-band communication APIs allows for a constant initialization cost of OpenSHMEM jobs at different core counts. We also obtain up to 90% reduction in number of network endpoints and up to 35% improvement in application execution time at 1,024 processes. While we use OpenSHMEM for our evaluations in this paper, our designs are applicable to other PGAS languages such as Unified Parallel C (UPC) [19] or Co-Array Fortran (CAF) [20] as well.

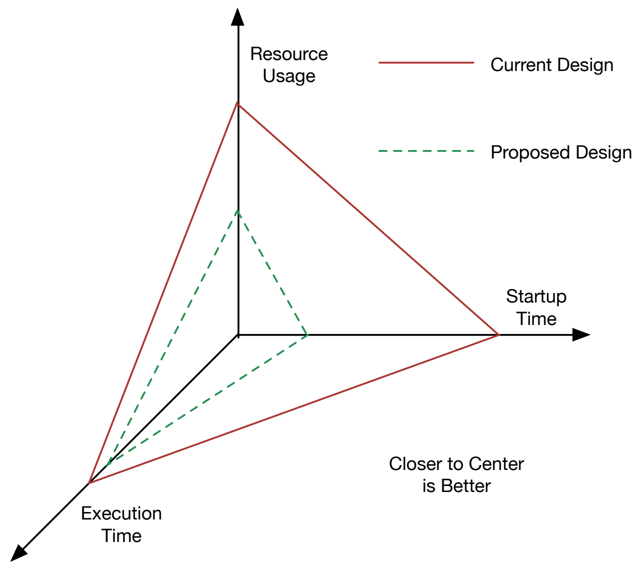


Fig. 2. Benefits of the proposed design on different performance aspects

III. Background

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

A. Partitioned Global Address Space

The Partitioned Global Address Space (PGAS) programming model brings the traditional shared memory model into a distributed memory setting. Instead of declaring all memory as shared, the address space is partitioned such that each processing element or PE (which can be a process or a thread) has affinity towards one partition. Global memory allocations are typically distributed over the partitions. Each PE can access local as well as remote memory in other partitions. The remote memory accesses are performed through library calls (in OpenSHMEM) or variable references (in languages like UPC) which get translated by the compiler. For ease of use and performance considerations, existing PGAS languages like Unified Parallel C (UPC), Co-Array Fortran (CAF), Titanium [21] also provide collective, synchronization, and bulk transfer APIs.

B. OpenSHMEM

OpenSHMEM is a library-based implementation of the PGAS programming model that was developed in an effort to standardize and bring together various SHMEM and SHMEM-like implementations. OpenSHMEM provides one-sided point-to-point and collective operations, a shared memory view, and atomic operations. In OpenSHMEM, global variables can be allocated at runtime in a special memory area called the symmetric heap. To support Remote Direct Memory Address (RDMA) access to the heap of a remote process, the participant processes are required to exchange certain information beforehand.

C. InfiniBand

InfiniBand is a switched fabric interconnect used in many supercomputing clusters world-wide. The InfiniBand hardware (Host Channel Adapter or HCA) is accessed through an interface called *verbs*. To perform any communication using the HCA, the software needs to create at least one Queue Pair (QP). Any send/receive or read/write operations are posted as elements in the work request queue. The HCA generates events on the completion queue to notify completion of each operation. The software can poll the completion queue to detect completion of requested work items.

InfiniBand also provides multiple transport protocols with different capabilities and guarantees. The most commonly used ones are Reliable-Connected (RC) and Unreliable-Datagram (UD). RC is a connection-oriented, reliable (HCA takes care of retransmission of dropped packets) protocol which requires one QP per process per connection. UD on the other hand is a connection-less unreliable protocol and requires only one QP per process to communicate with any number of processes. One of the important features of InfiniBand is Remote Direct Memory Access (RDMA). This feature allows a local process to directly access memory of another remote process without involving the software on the remote side. Full details of the InfiniBand capabilities are available in the InfiniBand specification [22].

D. GASNet

The GASNet specification [11] describes a language and network-independent high-performance communication interface for implementing the runtime system for global address-space languages. The GASNet interface consists of two components: 1) a Core API which is based on active messages and 2) an Extended API which is a more expressive and powerful interface that provides remote-memory access and collective operations.

GASNet is available on a variety of networks and these network-specific implementations are referred to as 'conduits'. A conduit is required to implement the core API and can optionally provide the extended API. GASNet-ibv is a commonly used conduit that implements these functionalities using the InfiniBand verbs interface.

The limitation of the ibv conduit arises from the fact that a hybrid application using both MPI and PGAS programming models requires two separate runtimes. This creates additional overhead and potential deadlock conditions caused by the inter-operation of the two separate stacks. The GASNet-mvapich2x conduit [9] provides a unified common runtime for both MPI and PGAS languages and delivers higher performance with lower overhead. Figure 3 illustrates the relationships between the different middlewares.

E. Process Management Interface

Process Management Interface (PMI) defines a portable interface that many HPC middlewares like MPI libraries and GASNet use to initialize communication between the processes of the parallel job. PMI is typically implemented

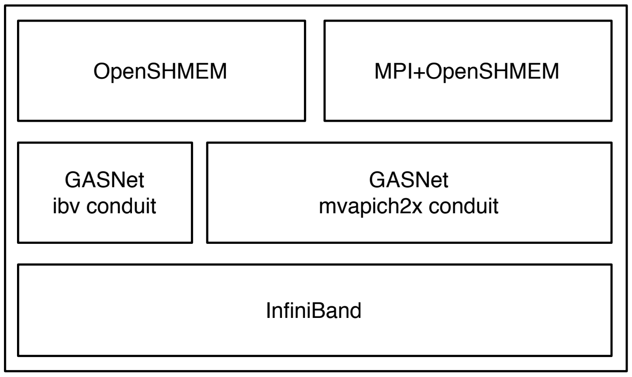


Fig. 3. OpenSHMEM implementation over InfiniBand using GASNet conduits

as a client-server library with the process manager (e.g., SLURM, mpirun_rsh, Hydra) acting as the server and the middleware taking the role of the client. The core functionality of PMI is to provide a global key-value store (KVS) that the processes can use to exchange information. The basic operations in PMI are `PMI2_KVS_Put`, `PMI2_KVS_Get`, and `PMI2_KVS_Fence`, which we refer to as *Put*, *Get*, and *Fence*, respectively. *Put* adds a new key-value pair to the store, and *Get* retrieves a value given a key. *Fence* is a synchronizing collective across all processes in the job. It ensures that any *Put* made prior to the *Fence* is visible to any process via a *Get* after the *Fence*.

Non-blocking Extensions to PMI2: Our earlier work [16] has shown that *Fence* is the most time consuming part of the *Put-Fence-Get* sequence. But as it is a blocking operation, the processes are idle and cannot perform any useful work while the *Fence* operation is progressed by the process manager. To mitigate this, we proposed a non-blocking version of the *Fence* operation where the calling process can overlap the *Fence* operation with other initialization steps [17]. As a further optimization, we also proposed a new PMI operation `PMIX_Iallgather` that combines the *Put-Fence-Get* sequence into a single operation and takes advantage of the symmetric data movement pattern commonly found in the middlewares for improved efficiency. Before the data is accessed after calling *Iallgather* or a *Get* is performed after a *Fence*, the process needs to call another function `PMIX_Wait` that ensures completion of the outstanding non-blocking operations.

IV. Design and Implementation

In this section, we describe the challenges of introducing on-demand connection management in OpenSHMEM and the designs introduced to solve them efficiently.

A. Two-Phase Connection Establishment in InfiniBand

InfiniBand provides a peer-to-peer connection model where the communicating processes need to create endpoints (QP) and perform certain operations on them to establish a connection. However, each process needs to obtain the $< lid, qpn >$ tuple (roughly equivalent to IP address and port number) for the remote QP in order to perform these operations. As InfiniBand does not provide an efficient way to obtain this information for a given remote process, an out-of-band channel (typically based on TCP) must be used in conjunction.

In both the ibv and mvapich2x conduits of GASNet, connection establishment is performed in two phases. At first, each process creates an UD endpoint and publishes the $< lid, qpn >$ tuple through an out-of-band channel called Process Management Interface (PMI). A process (referred to as client) trying to communicate with a remote process (referred to as server) needs to query the PMI key-value store using the rank of the remote process as the key.

To establish a connection, the client creates an RC endpoint and sends a $request$ message to the server through the previously created UD endpoint. The message contains the rank of the client as well as the $< lid, qpn >$ tuple of the newly created RC endpoint. The server then creates a corresponding RC endpoint and sends a $reply$ message with the same information. Now both the processes have all the necessary information to establish the connection. Since UD is not reliable, the software needs to handle dropped, out of order or duplicate messages. The processes also need to have a collision handling mechanism if both processes try to initiate the connection by sending the $request$ message at the same time. Figure 4 illustrates the message exchange protocol followed to establish a connection.

B. Key Exchange in OpenSHMEM

During initialization, each process in OpenSHMEM allocates one or more memory segments which are used as storage for global or dynamically allocated variables. The one-sided access semantics of OpenSHMEM requires each process to be able to read from and write to remote process's segments. While it is possible to have a design based on message passing and polling, the achievable performance would be poor.

InfiniBand's Remote Direct Memory Access (RDMA) semantics are well-suited for such accesses. Using RDMA, a process can read from and write to a remote process's memory without involving the remote process. For this scheme to work, the remote process must register the memory segment with the HCA and the caller process needs to obtain the triplet $< address, size, rkey >$ of the remote process. The parameters $address$ and $size$ refer to the starting address and the size of the registered segment respectively, and $rkey$ (Remote Key) is a unique identifier obtained on registering the segment with the HCA.

After allocating the segments, the OpenSHMEM process registers them and sends the $< address, size, rkey >$ triplets to every other process. OpenSHMEM uses the GASNet active message interface to send this information directly to each target process, which requires connections to be created between each pair of processes. Based on this, we identify three inefficiencies that we can eliminate:

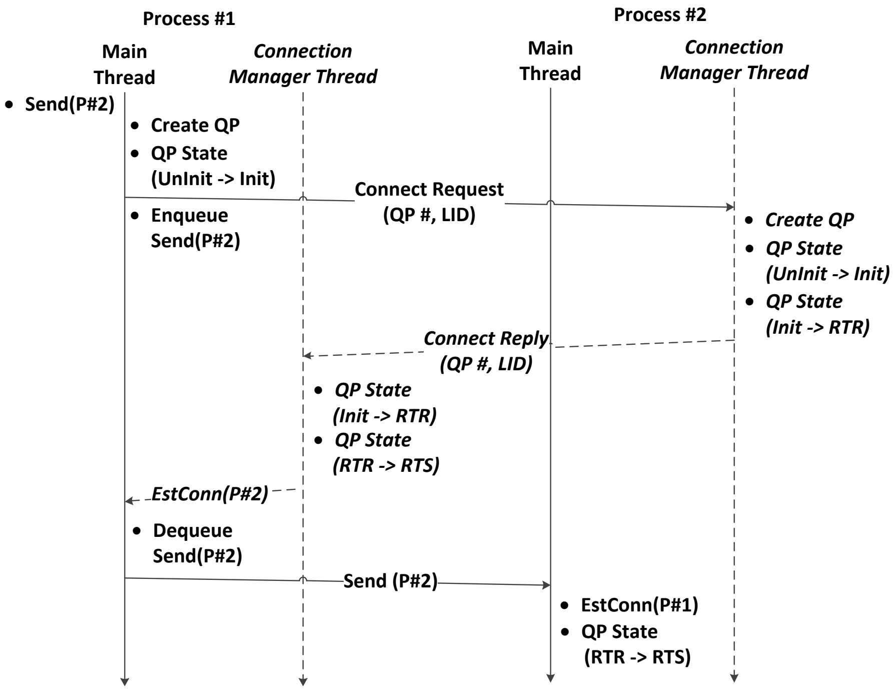


Fig. 4. Connection setup protocol in GASNet-mvapich2x conduit

1) Each process broadcasts its segment address and keys which forces establishment of all-to-all connectivity even if the underlying conduit supports on-demand connection setup.
2) After connection setup, each process sends another message containing the $< address, size, rkey >$ triplet, causing additional overhead.
3) OpenSHMEM uses a number of global barriers during initialization which results in connections being setup.

C. On-demand Connection Mechanism in OpenSHMEM

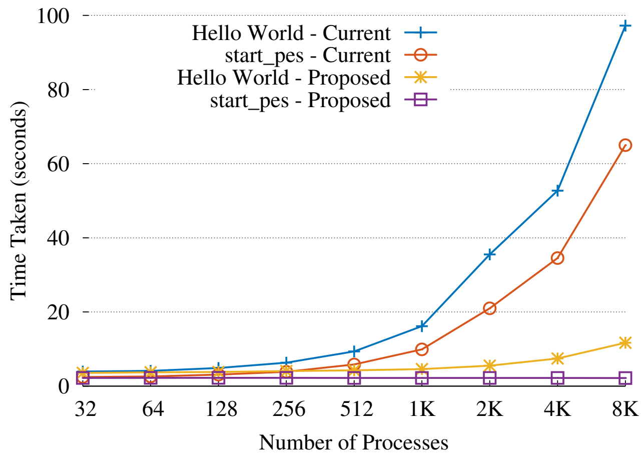
We modify the OpenSHMEM implementation to avoid the broadcast of the segment keys during initialization. Instead, the segment information is serialized and stored in a buffer.

When the client process initiates a connection, the GASNet conduit appends the data stored in the buffer with the connection $request$ message. When the server process receives this message, the conduit extracts the buffer and passes it to OpenSHMEM which can then parse the contents and populate the segment information for the client process. The same process is repeated with the $reply$ message from the server to the client. In essence, the segment information exchange packets are combined with the connection establishment packets and as a result, both the processes have the information to perform RDMA read/write operations on the connected peer as soon as the connection is established.

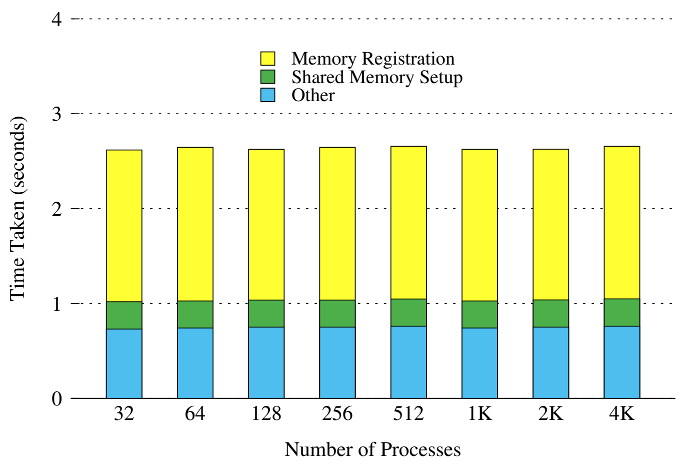
Since GASNet conduits need to support other implementations or other models such as UPC, it is not ideal for the conduit to handle the packing and unpacking of the segment information. To achieve this, the conduit layer allows the buffer to be read, written or ignored by the upper layer (OpenSHMEM/UPC) as necessary. This separation of concerns allows our design to be easily used by any similar PGAS language or implementation and also makes it backward compatible.

D. Overlapping PMI Communication with Initialization

We use the non-blocking PMI extensions described in Section III-E to accelerate the job initialization time. Instead of the *Put-Fence-Get* operations, the more efficient `PMIX_Iallgather` method is used to exchange the UD endpoint information among all the processes. While this exchange is being performed by the process manager over TCP, the processes can perform independent steps like memory registration. It should be noted that with the static connection mechanism, the *Fence* or the *Allgather* step needs to be completed at initialization, before the connection establishment phase. With the on-demand connection setup mechanism, this is no longer required and the processes can complete initialization and start the actual computation as the *Allgather* progresses in the background. A process needs to check for completion of the outstanding non-blocking operations only when it attempts to communicate with another process. Since the launching of the *Allgather* operation takes minimal amount of time, the cost of the PMI exchange can be completely or partially hidden depending on how much time the application spends in computation before entering the communication stage. The application enters the communication phase only after the PMI operations have completed, hence the PMI related traffic does not interfere with the application traffic.



(a) OpenSHMEM initialization and Hello World



(b) Breakdown of time spent in OpenSHMEM initialization

Fig. 5. Startup performance of OpenSHMEM with proposed designs on Cluster-B with 16 processes per node

E. *Shared Memory Based Intra-Node Barrier*

To synchronize the processes during initialization, OpenSHMEM uses a number of calls to the method `shmem_barrier_all()`. According to the specification this needs to be implemented as a global barrier operation across all processes and it would require at least $O(log\ P)$ connections to be established where P is the number of processes. With the introduction of the on-demand connection management, processes on different nodes no longer need to be explicitly synchronized. Processes on the same node, however, still require some synchronization to complete different phases of the initialization in tandem. We implement an intra-node barrier in the conduit and replaced the global barriers with the intra-node variant. This significantly reduces the time each process needs to spend in the initialization phase.

While launching a large scale job, the arrival pattern of the processes can vary considerably in absence of any global synchronization. This can result in a scenario where a client process can send a connection *request* before the server process is ready to handle the request, e.g., it has not yet registered its own segments. In such cases, the *reply* message is held until the server is ready. This may trigger a timeout on the client side causing a retransmission. The conduits can treat this scenario similar to a lost message and handle it accordingly.

V. Experimental Results

In this section, we compare our proposed on-demand connection mechanism with the existing static connection mechanism using a set of different microbenchmarks and applications. We focus on three major aspects — time taken for initialization and launching the job, overhead introduced by on-demand connection establishment, and resource utilization.

A. *Experimental Setup*

We used two different clusters for our evaluation.

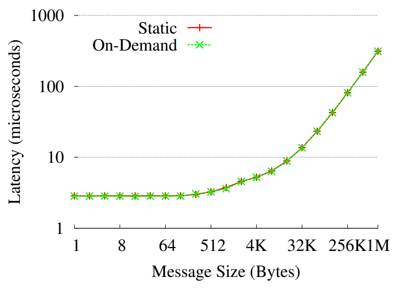
Cluster-A: This cluster has 144 compute nodes with Intel Westmere series processors, using Xeon dual quad-core sockets operating at 2.67 GHz and 12 GB of RAM. Each node is equipped with Mellanox MT26428 QDR Connect-X HCAs (32 Gbps) with PCI-Ex Gen2 interface. The operating system used is Red Hat Enterprise Linux Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3-3.

Cluster-B: We used the Stampede supercomputing system at TACC [23] to take large scale 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. The numbers reported were taken in fully subscribed mode with 16 processes per node.

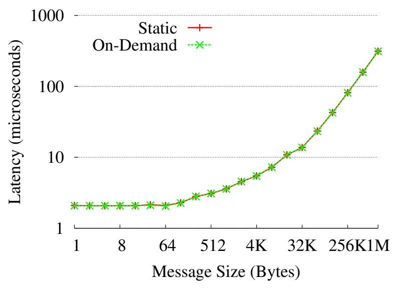
For the experiments, we used MVAPICH2-X 2.1rc1, OpenSHMEM based on OpenSHMEM version 1.0h and GASNet version 1.24.0. For microbenchmarks, we used the OSU microbenchmarks suite [24] version 4.4. The OpenSHMEM ports of the NAS parallel benchmarks suite were obtained from the official OpenSHMEM repository [25]. The performance numbers reported in Figure 1 and Figure 5 were taken on Cluster B and all other experiments were performed on Cluster A. The collective operations and applications were evaluated in a fully subscribed manner with 8 processes per node.

B. *Impact on Job Startup*

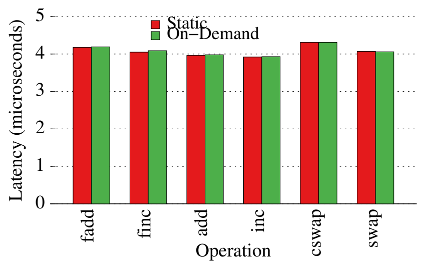
We look at two metrics — time taken for initialization and completion of a simple Hello World application. For measuring the initialization time, we introduce a new benchmark in which each process measures the time taken for the `start_pes` (OpenSHMEM equivalent of `MPI_Init`) call and reports the average. For Hello World, we just measure



(a) shmem_get

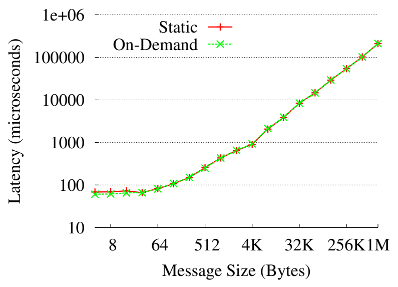


(b) shmem_put

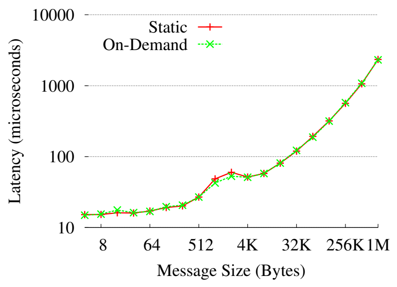


(c) shmem_atomics

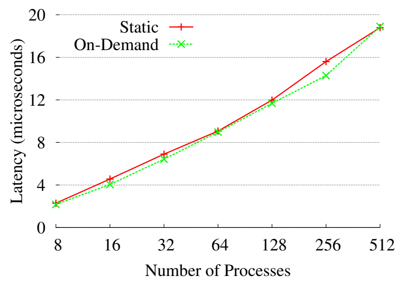
Fig. 6. Performance comparison of point-to-point and atomic operations on Cluster-A



(a) shmem_collect with 512 processes



(b) shmem_reduce with 512 processes



(c) shmem_barrier_all

Fig. 7. Performance comparison of collective operations on Cluster-A

the wall clock time as reported by the job launcher. We show average of 20 iterations to avoid effects of system noise.

Figure 5(a) shows the improvements obtained by the application of the on-demand connection setup mechanism. With the existing static connection based design, time taken for `start_pes` grows rapidly at large scale whereas with the new design it takes near-constant time at any process count. At 8,192 processes, `start_pes` completes nearly 30 times faster with the new design. Figure 5(b) shows the breakdown of the initialization procedure with on-demand connections and non-blocking PMI collectives. Compared to the static method shown in Figure 1, there is negligible time spent on PMI operations and connection setup phase.

Performance of Hello World shows a similar trend. At 8,192 processes our design performs 8.3 times better than the static connection based mechanism. The reason behind the difference in the improvement obtained is twofold: 1) The Hello World application provides minimal opportunity for overlapping the PMI communication with actual computation and 2) Even if the application does no communication, a global barrier is required at finalize to ensure proper termination of the program. This barrier requires completion of the PMI operations and establishment of some connections.

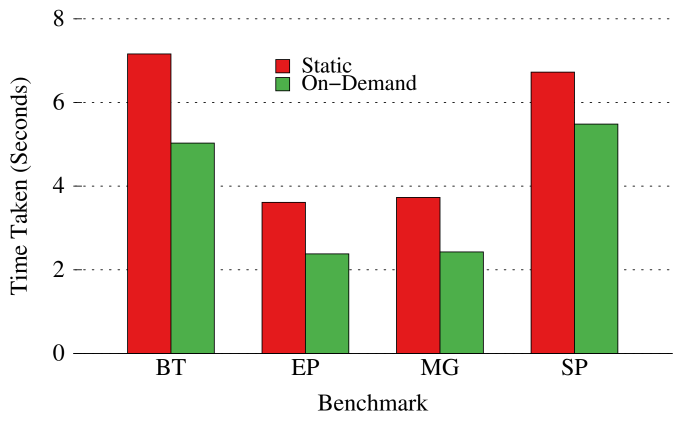
C. *Performance of Point-to-point and Collective Operations*

At a microbenchmark level, the on-demand and static connection establishment methods show identical performance. In Figure 6, we compare the performance of `shmem_put` and `shmem_get` for different message sizes and also the latency of different atomics like swap, compare and swap among others. In all cases, there is less than 3% difference between the two approaches.

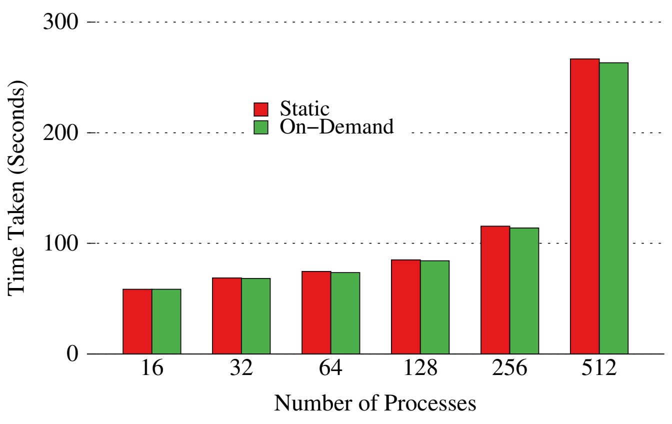
Figure 7 shows that the performance of `shmem_barrier_all` at different process counts is similar for both schemes. We also show the performance of a dense (`shmem_collect`) and a relatively sparse (`shmem_reduce`) collective operation with 512 processes. It should be noted that while static connections are pre-established, for on-demand setup the time taken to establish the connections is included in the timing loop and is amortized over multiple iterations. The reported numbers are averaged across 1,000 iterations and 5 different runs.

D. *Performance of OpenSHMEM Application*

Although the performance of point-to-point and collective operations remains unchanged, applications benefit from the shorter initialization time and perform better in terms of total execution time. The improvement observed depends on the application's communication characteristics, specifically the amount of time it spends in computation before initiating the first communication, and how many peer processes it communicates with. We use the OpenSHMEM versions [25] of NAS parallel benchmarks suite [26] for our evaluation. Unfortunately, not all of the applications have been ported to OpenSHMEM. Figure 8(a) compares the total execution time of the applications as reported by the job launcher with static and on-demand connection mechanisms using class B on 256 cores. We observe improvements ranging from 18% to 35%



(a) OpenSHMEM NAS benchmarks with 256 processes



(b) Hybrid Graph500 with different process counts

Fig. 8. Comparison of execution time of OpenSHMEM and hybrid MPI+OpenSHMEM applications on Cluster-A

in the applications.

E. Performance of Hybrid (MPI+OpenSHMEM) Application

A scalable version of the Graph500 application using hybrid MPI+OpenSHMEM programming model was introduced by Jose et al [5]. We measured the execution time of this application with up to 512 processes using static and on-demand connection mechanisms. As shown in Figure 8(b), we see negligible performance difference ($< 2\%$) between the two schemes. The graph used for this evaluation consisted of 1,024 vertices and 16,384 edges. The total execution time reported here includes time spent in generation of the graph and validation of the results.

F. Impact on Resource Usage and Scalability

With the on-demand connection setup mechanism, the number of connections, endpoints and associated resources created are limited by what the application actually requires. Consequently, the resource utilization is always 1 whereas for the static connection mechanism generally only a small subset of the endpoints created would be actually used. From Table I it is easy to see that the number of communicating peers per process does not increase linearly with the total process count. Rather, this number stays nearly constant or grows sub-linearly as the surface-to-volume ratio of the computational grid decreases.

Figure 9 illustrates how the communication pattern of a few different application varies with number of processes. The applications have fewer communicating peers per process compared to the number of total processes and benefit significantly from the on-demand connection setup scheme. The 2D-Heat kernel [27] shows the best scalability, followed by EP; while BT, MG, and SP shows very similar resource usage. The data collected with process counts 64, 256 and 1,024 is used in a linear regression to estimate the resource usage at 4,096 process. At 1,024 processes, the applications show more than 90% reduction in number of connections and endpoints which directly translates to lower memory usage

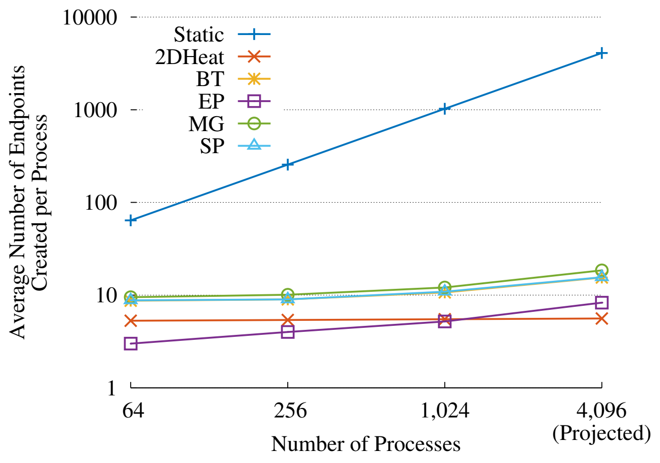


Fig. 9. Actual and projected resource usage for different applications

and higher scalability, with even higher benefits estimated at larger process counts. It should be noted that the number of peers for a given application can vary based on the runtime's implementation of various collective operations.

VI. Related Work

A large body of research work exists that focuses on improving scalability and performance of launching large-scale parallel applications. The implementation and impact of on-demand connection management in MPI over VIA-based networks was presented by Wu et al [28]. An implementation for on-demand connections for the ARMCI interface is described in [29]. While [28] was purely in the context of MPI, [29] attempted to expand the use of [28] to the context of the ARMCI programming model. However, both these works only explored on-demand method to startup jobs. In this paper we focus primarily on OpenSHMEM instead of the underlying communication library and we also explore the possibility of overlapping the "out-of-band" PMI based communication with startup related activities using non-blocking extensions to PMI.

Yu et al [30] proposed a ring based startup scheme for MPI programs on InfiniBand clusters. Our earlier work [16] introduced a novel `PMIX_Ring` collective to minimize the amount of data movement performed at initialization. We further proposed non-blocking extensions to the PMI interface in [17] where we demonstrated the use of split-phase *Fence* and *Allgather* routines to achieve near-constant startup time for MPI applications. In this work we take advantage of the proposed extensions to accelerate OpenSHMEM startup in a similar fashion.

Multiple researchers have proposed different connection schemes [12, 13, 31–33] to improve the scalability of MPI runtimes over InfiniBand. The MVAPICH-Aptus runtime [34] dynamically selects the UD or RC protocol based on the application's communication pattern.

Fundamental research on a Unified Communication Runtime (UCR) for MPI and UPC appears in [9]. It describes an integrated runtime that enables simultaneous communication for UPC and MPI over InfiniBand. The runtime was later extended to support OpenSHMEM in [35]. Our work effectively utilizes this runtime for simultaneous OpenSHMEM and MPI communication.

VII. Conclusion and Future Work

In this paper, we show the shortcomings of the existing static connection mechanism in OpenSHMEM. We implement an on-demand connection setup mechanism in OpenSHMEM that leverages the capabilities of the underlying GASNet conduits to establish connections between communicating peers only. We also take advantage of non-blocking PMI APIs and reduce synchronization by removing global barriers in favor of intra-node barriers. With these designs we are able to dramatically improve job startup time, resource utilization and scalability of pure OpenSHMEM and hybrid MPI+OpenSHMEM applications without introducing any additional overhead. Our designs show 30 times reduction in initialization time and 8.3 times improvement in execution time of a Hello World application. We also improve resource utilization by reducing the number of network endpoints by up to 90% and show up to 35% improvement in execution time of the NAS parallel benchmark suite applications.

Our designs are already available in the latest public release of MVAPICH2-X (version 2.1rc1). As part of future work, we intend to evaluate the effectiveness of our designs on different applications at a larger scale. We also plan to extend our work to other PGAS languages such as UPC and CAF.

Acknowledgments

We would like to thank Dr. Khaled Hamidouche for his valuable input and feedback.

References

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

[2] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray Users Group (CUG)*, 2010.

[3] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands *et al.*, "Productivity and Performance using Partitioned Global Address Space Languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007, pp. 24–32.

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

[5] J. Jose, S. Potluri, K. Tomko, and D. K. Panda, "Designing Scalable Graph500 Benchmark with Hybrid MPI+ OpenSHMEM Programming Models," in *Supercomputing*. Springer, 2013, pp. 109–124.

[6] J. Jose, S. Potluri, H. Subramoni, X. Lu, K. Hamidouche, K. Schulz, H. Sundar, and D. K. Panda, "Designing Scalable Out-of-core Sorting with Hybrid MPI+ PGAS Programming Models," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 7.

[7] A. Geist, "Paving the Roadmap to Exascale," *SciDAC Review*, vol. 16, 2010.

[8] MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems, http://mvapich.cse.ohio-state.edu/.

[9] J. Jose, M. Luo, S. Sur, and D. K. Panda, "Unifying UPC and MPI runtimes: Experience with MVAPICH," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 5.

[10] (2015) Infiniband Trade Association. [Online]. Available: http://www.infinibandta.org/

[11] D. Bonachea, "GASNet Specification, v1. 1," *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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

[17] 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)*, To appear in 2015.

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

[19] T. El-Ghazawi and L. Smith, "UPC: Unified Parallel C," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 27.

[20] R. W. Numrich and J. Reid, "Co-Array Fortran for Parallel Programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[21] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," in *In ACM*, 1998, pp. 10–11.

[22] InfiniBand Trade Association, http://www.infinibandta.org/.

[23] (2015) Stampede, Texas Advanced Computing Center. [Online]. Available: https://www.tacc.utexas.edu/stampede/

[24] (2015) OSU Micro-Benchmarks. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[25] (2015) NASA Parallel Benchmarks for OpenSHMEM. [Online]. Available: http://www.openshmem.org/site/Downloads/Examples

[26] 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, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks - Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[27] C. J. Palansuriya, C.-H. Lai, C. S. Ierotheou, and K. A. Pericleous, "A Domain Decomposition Based Algorithm For Non-linear 2D Inverse Heat Conduction Problems," *Contemporary mathematics*, vol. 218, pp. 515–522, 1998.

[28] J. Wu, J. Liu, P. Wyckoff, and D. Panda, "Impact of On-Demand Connection Management in MPI over VIA," in *In CLUSTER 02: Proceedings of the IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2002, pp. 152–159.

[29] A. Vishnu and M. Krishnan, "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, 2010, pp. 175–184.

[30] 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.

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

[32] M. J. Koop, T. Jones, and D. K. Panda, "Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 495–504. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2007.92

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

[34] M. J. Koop, T. Jones, and D. K. Panda, "MVAPICH-Aptus: Scalable High-performance Multi-transport MPI over InfiniBand," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[35] J. Jose, K. Kandalla, M. Luo, and D. Panda, "Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation," in *Parallel Processing (ICPP), 2012 41st International Conference on*, Sept 2012, pp. 219–228.