

Efficient On-Demand Connection Management Mechanisms with PGAS Models over InfiniBand

Abhinav Vishnu*, and Manoj Krishnan*

* High Performance Computing Group

Pacific Northwest National Lab, Richland, WA 99352

Email: {abhinav.vishnu, manoj}@pnl.gov

Abstract—In the last decade or so, clusters have observed a tremendous rise in popularity due to the excellent price to performance ratio. A variety of Interconnects have been proposed during this period, with InfiniBand leading the way due to its high performance and open standard. At the same time, multiple programming models have emerged in order to meet the requirements of various applications and their programming models. To support requirements of multiple programming models, InfiniBand provides multiple transport semantics, ranging from unreliable connectionless to reliable connected characteristics. Among them, the reliable connection (RC) semantics is being widely used due to its high performance and support for novel features like Remote Direct Memory Access (RDMA), hardware atomics and Network Fault Tolerance. However, the pairwise connection oriented nature of the RC transport semantics limits its scalability and usage at the increasing processor counts. In this paper, we design and implement on-demand connection management approaches in the context of Partitioned Global Address Space (PGAS) programming models, which provided shared memory abstraction and one-sided communication semantics, leading to the development of multiple languages (UPC, X10, Chapel) and libraries (Global Arrays, MPI-RMA). Using Global Arrays as the research vehicle, we implement this approach with Aggregate Remote Memory Copy Interface (ARMCI), the runtime system of Global Arrays. We evaluate our approach, ARMCI-On Demand Connection Management (ARMCI-ODCM) using various microbenchmarks and benchmarks (LU Factorization, RandomAccess and Lennard Jones simulation) and application (Subsurface transport over multiple phases (STOMP)). With the performance evaluation for up to 4096 processors, we are able to have a multi-fold reduction in connection memory with a negligible degradation in performance. Using STOMP at 4096 processors, reduces the overall connection memory by 66 times with no performance degradation. To the best of our knowledge, this is the first design, implementation and evaluation of on-demand connection management with InfiniBand using PGAS models.

I. INTRODUCTION

The computational needs of today’s scientific applications has led to the augmentation of high performance computing. Combining commercial off the shelf processors with commodity interconnects has led to cluster computing [1], a very effective methodology for achieving excellent price-to-performance ratio. As the commodity processors continue to grow, commodity interconnects such as Myrinet [2], Quadrics [3], and InfiniBand [4] are being introduced to combine these commodity processors. As reflected by the TOP500 [5] rankings, InfiniBand in particular has been observing wide acceptance

due to its high performance and open standard, with 28% of the systems using InfiniBand as their interconnect. The current largest InfiniBand cluster uses more than 70,000 processor cores at NUDT [6], and larger scale systems are being planned for the near future.¹

A variety of programming models have emerged at the same time, to provide scientists with different tools for writing their parallel applications. While two-sided message passing continues to dominate with Message Passing Interface (MPI) [7], [8], Partitioned Global Address Space (PGAS) programming models like Global Arrays [9] are being used in a variety of applications with dynamic computation characteristics and naturally suiting one-sided communication paradigm. To meet the needs of multiple programming models, modern interconnects like InfiniBand support multiple transport semantics, ranging from unreliable connection-less to end-to-end reliable connection characteristics. Among them, reliable connection semantics has observed most popularity due to support for novel features like Remote Direct Memory Access (RDMA), hardware atomics and Automatic Path Migration [10]. However, the pairwise connection oriented nature of the reliable connection transport semantics introduces scalability challenges. Each of the connection can consume up to 44KBytes [11] of memory, resulting in a utilization of more than 700 MBytes of connection memory for 16K processors!

While this problem has been addressed in detail with MPI for two-sided messaging semantics [12], [11], [13], [14], [15], PGAS programming models entail additional challenges. The two-sided natures of connection establishment protocol maps is a natural fit with implicit synchronization of MPI. However, this property imposes challenges for PGAS models, which use one-sided communication and exhibit no implicit synchronization. A variety of MPI collective communication patterns allow for automatic overlap in connection management, while connection establishment is always serialized for PGAS models. As a result state of the art PGAS runtime systems like Aggregate Remote Memory Copy Interface (ARMCI) [16] and GASNet [17] use static connection establishment mechanisms. To address this, we present a design for on-demand connection management for PGAS models runtime

¹This work is supported by Computational Science and Mathematics Division program of the Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, with a contract for Pacific Northwest National Lab, operated by Battelle.

systems using Aggregate Remote Memory Copy Interface (ARMCI) [16], the runtime system of Global Arrays [9]. Within the solution space, we present the overall design of ARMCI over InfiniBand, multiple overlap protocols for on-demand connection creation and establishment; simplification of the connection establishment state machine and reliability protocol due to the asymmetric nature of communication paradigm in PGAS runtime systems. We implement our design and evaluate it with micro-benchmarks and benchmarks (LU Factorization, RandomAccess, and Lennard Jones simulation) and applications (Sub-surface transport over Multiple Phases (STOMP) [18]). With the performance evaluation for up to 4096 processors for benchmarks, we are able to achieve a multi-fold reduction in connection memory without perceivable performance degradation. With evaluation of STOMP at 4096 processors, we are able to reduce the connection memory utilization by 66 times. To the best of our knowledge, this is the first design, implementation and evaluation of on-demand connection management approach with PGAS model runtime systems over InfiniBand.

The rest of the paper is organized as follows. In section II, we present the background of our work. In section III, we present the design of ARMCI-ODCM, discussing the challenges presented by the PGAS models. In section IV, we present the performance evaluation of ARMCI-ODCM using simple micro-benchmarks and a variety of benchmarks and STOMP, comparing it to the state of the art implementation. We present the related work in section V. We conclude and present our future directions in section VI. We begin with the description of the background work.

II. BACKGROUND

In this section, we present the background of our work. We begin with an introduction to InfiniBand [4] and the state transitions associated with a Queue Pair (QP). We also provide a brief introduction to InfiniBand transport semantics, Global Arrays [9] and Aggregate Remote Memory Copy Interface (ARMCI) [19].

ABhinav Vishnu

A. Overview of InfiniBand and QP Transition States

The InfiniBand Architecture (IBA) [4] defines a switched network fabric for interconnecting processing nodes and I/O nodes. An InfiniBand network consists of switches, adapters (called Host Channel Adapters or HCAs) and links for communication. InfiniBand supports different classes of transport services (Reliable Connection, Unreliable Connection, Reliable Datagram and Unreliable Datagram). In this paper, we focus on reliable connection and unreliable datagram transport semantics. In reliable connection model, each process-pair creates a unique entity for communication, called *queue pair*. Each queue pair consists of two queues; *send queue* and *receive queue*. Figure 1 shows the communication state transition sequence for a QP.

At the point of QP creation, its communication state is RESET. At this point, it is assigned a unique number called

qp_{num} by the InfiniBand access layer (Verbs) [4]. From this state it is transitioned to the INIT state by invoking `modify_qp` function. The `modify_qp` function is provided by the access layer of InfiniBand [4]. During the RESET-INIT transition, the QP is specified with the HCA port to use in addition to the atomic flags. Once in the INIT state, the QP is specified with the destination local identifier (LID) $DLID$ and the destination QP from which it will receive the messages. A `modify_qp` call transitions it to READY-TO-RCV (RTR) state. At this point, the QP is ready to receive the data from the destination QP. Finally, QP is transitioned to READY-TO-SEND (RTS) state by specifying associated parameters and making the `modify_qp` call. At this point, the QP is ready to send and receive data from its destination QP. Should any error(s) occur on the QP, the QP goes to the ERROR state automatically by the hardware. At this state, the QP is broken and cannot communicate with its destination QP. In order to re-use this QP, it needs to be transitioned back to the RESET state and the above-mentioned transition sequence (RESET-RTS) needs to be re-executed. The RTS-Send Queue Drained (SQD) transition is an important mechanism to ensure that the outstanding data requests have completed. After a QP is in SQD state, it can be transitioned to RTS state directly to allow messages to be sent/received from the communicating pair. We will use the term QP and connection interchangeably for the rest of the paper.

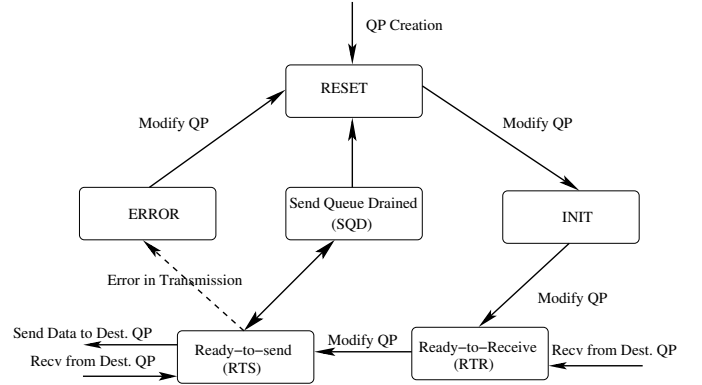


Fig. 1. QP Communication State Diagram

1) *Data Transfer Requests and Completion Queue*: The data transfer requests are initiated by posting a send/receive descriptor to the send/receive queue of the QP. For one-sided programming models like Global Arrays [9], RDMA and send/receive descriptor can be posted on the queue. Once the request is completed, an entry is generated at the completion queue. InfiniBand supports polling and blocking mechanisms to check for the completion. In this paper, we use the polling mechanism for checking the completion queue, since it results in lesser overhead compared to interrupt generation.

B. InfiniBand Transport Semantics

In this section, we present a brief introduction to the InfiniBand transport semantics. We specifically focus on the

unreliable datagram and reliable connection transport semantics.

1) *Unreliable Datagram*: The unreliable datagram (UD) model provides connection-less model for communication. Each process creates a QP (not unique) for every other process in an application. UD transport semantics guarantees at most once data delivery with checksum. The in-order data delivery is not guaranteed. Since this transport semantics does not support RDMA with InfiniBand, it is imperative to use reliable connection transport semantics for data transfer. As a result, we use the unreliable datagram as the out-of-band channel for connection establishment.

2) *Reliable Connection*: Reliable connection is the most popular transport semantics for designing runtime communication system over InfiniBand [11]. A variety of features including RDMA, Automatic Path Migration and hardware atomics are available with this semantics, which are not available with UD. In addition, it provides exact one, in-order data delivery to the destination and exact-once notification of the data delivery to the initiator, making it an attractive choice to design runtime communication systems.

However, RC requires a pairwise connection between a pair of communicating processes. Clearly, this inhibits scalability at large processor counts. Hence, it is important to design an on-demand connection management and maintain connections only to the communicating processes. While solutions exist for the two-sided message passing with MPI [11], [14], no solutions exist for the run-time systems for PGAS models like Global Arrays and ARMCI. In this paper, we design ARMCI-ODCM, which provides on-demand connection management with PGAS models over InfiniBand.

C. Global Arrays and ARMCI

Global Arrays: The Global Arrays [9] programming model provides an efficient and portable “shared-memory” programming interface for distributed-memory computers. Each process in a Multiple Instruction Multiple Data (MIMD) parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without need for an explicit co-operation by other processes. Unlike other shared-memory environments, the GA model exposes a non-uniform memory access (NUMA) characteristics of the high performance computers and acknowledges that access to a remote portion of the shared data is slower than to the local portion. The locality information for the shared data is available, and a direct access to the local portions of shared data is provided [9]. Global Arrays uses Aggregate Remote Memory Copy Interface (ARMCI) [16], as the runtime system for communication.

ARMCI: The purpose of the ARMCI [16] library is to provide a general-purpose, efficient, and widely portable remote memory access (RMA) operations (one-sided communication) optimized for contiguous and non-contiguous (strided, scatter/gather, I/O vector) data transfers. In addition, ARMCI includes a set of atomic and mutual exclusion operations. ARMCI exploits native network communication interfaces and

system resources (such as shared memory) to achieve the best possible performance of the remote memory access. It exploits high-performance network protocols on clustered systems. Optimized implementations of ARMCI are available for the Portals [20], Myrinet (GM and MX) [21], Quadrics [22], and InfiniBand (using OpenFabrics and Mellanox Verbs API) [4]. It is also available for DOE leadership class machines including Cray XT4 [23] and BlueGene/P [24].

III. ARMCI-ODCM DESIGN

In this section, we present the solution space for on-demand connection management with Global Arrays using ARMCI. We begin with a description of the state of the art implementation with ARMCI over InfiniBand. We propose a new connection management protocol, which performs overlap of different states of connection management. We also discuss reliability mechanisms, implementation details and present the discussion on various aspects of the connection establishment protocol.

A. Overall Design

In this section, we present the overall design of ARMCI over InfiniBand. We begin with the description of the terminology.

1) *Terminology*: In this section, we present the terminology used for processes in a Global Arrays application. Although ARMCI processes are SPMD processes, it differentiates in the terminology between processes on the same node to facilitate the primitives of one-sided communication. The process with the lowest rank in the node is called *master* and the rest of the processes on a node are called *clients*. The master process creates a thread (which can be configured to perform polling/blocking on an event), which is referred to as the *data server*. The data server performs operations on behalf of clients on other nodes including accumulate operations, and other one-sided communication primitives (put, get, lock), which may not be efficiently implemented using memory semantics like RDMA. The master process on a node is treated as a client by the data server on another node.

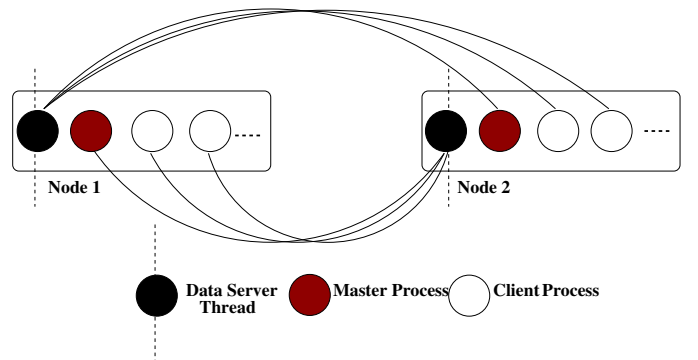


Fig. 2. Connection Establishment Pattern in ARMCI

A client establishes a connection to only the data server on another node; there are no client-client connections. While two-sided message passing creates pairwise connections between communicating processes, the fundamental nature of

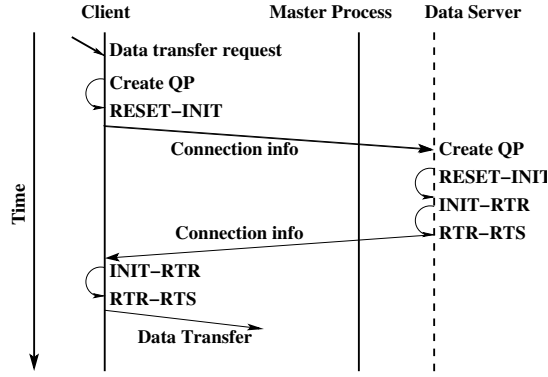


Fig. 3. Original:Connection Establishment Protocol

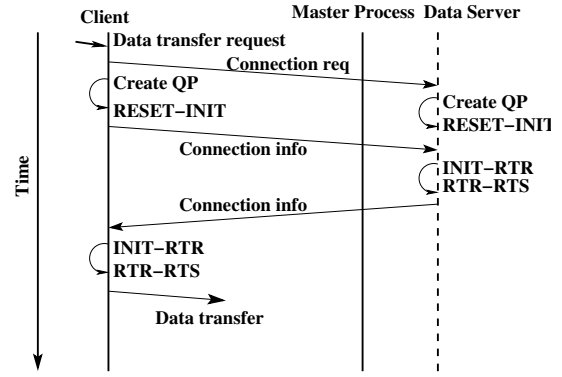


Fig. 4. ARMCI-ODCM:Connection Establishment Protocol

PGAS models like Global Arrays precludes this requirement. Each chunk of a Global Array is allocated by the master process, created as a shared memory segment, and registered with the adapter. This allows each client on a node to read and write to the chunk directly. In addition, it also allows clients on other nodes to read/write directly from/to the chunk. Hence, it is necessary only to establish connections with the data server on a node, which has access to the chunk, since it is a thread of the master process. Figure 2 shows a typical connection management scenario with ARMCI.

The state of the art ARMCI implementation establishes connections during the initialization phase of the communication runtime system. Since all connections may not be used by the application, this results in increased connection memory utilization. In addition, it also leads to overhead in overall execution time, since each connection establishment takes a significant amount of time. We present a detailed performance analysis in Section IV.

B. On-Demand Connection Establishment

In this section, we present the state of the art connection establishment protocols with traditional two-sided message passing. While these protocols are sufficient for two-sided message passing, they may not be sufficient for one-sided communication runtime systems like ARMCI. Hence, we present a new protocol for connection establishment, which leverages the phase based connection management to overlap the connection establishment as much as possible. We also present a discussion on various reliability mechanisms discussed in literature and choice of the reliability mechanism for ARMCI-ODCM.

1) *State of the Art Connection Management with Two-Sided Messaging*: Collective Communication primitives including All-to-all broadcast, All-to-all personalized exchange and Allreduce are the primary workhorses of most MPI applications, in addition to the MPI send and receive primitives. Since most of the collective communication primitives are based on MPI_Sendrecv, which results in bi-directional exchange between processes. As an example, using Ring based algorithm for MPI all-to-all broadcast results in simultaneous exchange of data between the left and the right neighbor. As a

result, in the absence of a connection with the neighbors, this results in a overlap of connection establishment. To summarize, the two-sided nature of MPI results in parallel connection establishment, resulting in the least possible overhead.

However, with one-sided communication primitives, the connection request is always initiated by a client, resulting in no overlap for connection establishment. Clearly, this results in an overhead for one-sided programming models, in comparison to the MPI two-sided communication semantics. To alleviate this, we present a novel design of a connection establishment protocol with InfiniBand, which provides overlap at multiple stages of connection establishment.

2) *Overlap Protocol for Connection Establishment*: Figure 3 shows the state of the art connection establishment protocol with ARMCI. As discussed in section II, there are multiple phases in connection establishment protocol. The state of the art connection establishment protocol performs the following algorithm:

- On request of a data transfer to target, check if a connection exists to the data_server (target)
- If a connection exists, continue with the data transfer, else *Create QP* and transition the connection from *RESET-INIT* state
- Send the *Connection info* to the data server and wait for data server's connection information
- Transition the connection from *INIT-RTR* and *RTR-RTS* states. Continue with the *Data transfer*.

From the above protocol and Figure 3, we see that the multiple phases of connection establishment protocol are completely serialized. In order to understand the overhead of each phase, we performed experiments at Verbs (Hardware access layer of InfiniBand) layer. We observed that the connection creation and transition to INIT phase is three times more expensive in comparison to the INIT-RTR, RTR-RTS phases. The connection creation and transition to INIT phase for one connection takes 367 us, while INIT-RTR and RTR-RTS phases take in the order of 105 us. Taking this observation in to account, we designed an overlap protocol for connection creation as follows:

- On a *data transfer request* to target, check if a connection exists to the data_server (target)

- If a connection exists, continue with the data transfer, else send a *Connection req* message to the data server, *Create connection* and transition the connection from *RESET-INIT* state
- Send the *Connection info* to the data server and wait for data server's Connection info
- Transition the connection from *INIT-RTR* and *RTR-RTS* states. Continue with the *Data transfer*.

Figure 4 shows this protocol in further detail. With this change, the connection creation and INIT-RTR phases of the overlap protocol are overlapped, with incurring a slight overhead of sending an extra connection creation message. Since the connection information is in the order of 16 bytes, the overall overhead incurred is lesser than 10us, while providing an overlap of multiple orders of magnitude.

The change in the connection establishment protocol also requires a change in the progress engine at the data server. With the ARMCI-ODCM implementation, the data server returns to poll for new completions, as soon as connection is created on behalf of the data request. Once the connection information from the client is received, it proceeds to perform the INIT-RTR and RTR-RTS transitions.

3) *Reliability Mechanisms*: A variety of reliability mechanisms have been discussed in literature with InfiniBand [11], [14]. Koop et al., have presented that the progress-engine based hybrid scheme presents the best case scenario, since it leads to lesser overhead from interrupts [14]. The primary problem with this mechanism is an unnecessary overhead of retransmissions, if the remote process does not respond to the connection request within a timeout. As a result, Koop et al., have suggested a hybrid mechanism based on Progress Engine and interrupts [11]. This problem is mainly observed due to the two-sided nature of message passing and potentially simultaneous initiation of data transfer.

However, the asymmetric nature of connection establishment with ARMCI-ODCM allows to prevent the occurrence of the false positives in the re-transmission of connection request. Since the data server executes as a thread, it is able to respond to the connection requests from clients, unless there is an actual data loss. Hence, we implement the progress engine based mechanism for reliability with the ARMCI-ODCM implementation.

C. Discussion

In this section, we present the discussion related to ARMCI-ODCM. During the connection establishment phase, we assume that the connections can always be created. While this assumption works for the scale of clusters considered for performance evaluation in this work, larger scale clusters are likely to have increasing failure in connection establishment. We are working to alleviate this limitation.

Koop et al., presented a connection-less approach using InfiniBand unreliable datagram [11], and presented its efficacy with the MPI applications. Unreliable datagram is attractive for message passing based programming models, since they have implicit synchronization. In addition, it does not

support RDMA, which is a key feature for providing one-sided communication. Hence, it is not suitable for one-sided programming models like Global Arrays [9].

A possible mechanism is to have an on-demand disconnection protocol, similar to on-demand connection establishment. This mechanism although attractive to alleviate the above-mentioned limitation requires careful design and evaluation in order to prevent redundant connection establishment/disconnection overhead. We are currently working to design efficient disconnection protocols with InfiniBand for one-sided communication runtime systems.

IV. PERFORMANCE EVALUATION OF ARMCI-ODCM

In this section, we present the performance evaluation of ARMCI-ODCM. We compare this with the latest release of Global Arrays, version 4.2, which we refer to as "Original" for the rest of section. We use latency and bandwidth micro-benchmarks for various one-sided communication primitives (put, get, and accumulate) to understand the overheads incurred by ARMCI-ODCM. This is followed by the performance evaluation using application kernels and benchmarks - LU Factorization, Lennard Jones Simulation and Random Access benchmark. We also present the performance evaluation with STOMP, an application to perform the ground water simulation. We have used Chinook [25], an AMD Barcelona based Supercomputer at Pacific Northwest National Lab as the experimental testbed for our evaluation.

A. Experimental Testbed

Chinook [25] is a 160 TFlops system that consists of 2310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Barcelona Processors. Each node has 32 Gbytes of memory and 365 Gbytes of local disk space. Communication between the nodes is performed using InfiniBand with Voltaire [26] Switches and Mellanox [27] Adapters. The system runs a version of Linux based on Red Hat Linux Advanced Server. A global 297 Tbyte SFS file system is available to all the nodes.

B. Performance Evaluation with Micro-benchmarks

In this section, we present the results using micro-benchmarks of various one-sided communication primitives. While most one-sided communication runtime systems focus on contiguous data transfer ARMCI is heavily used for uniform non-contiguous data transfer (strided), in addition to the contiguous data transfer.

For each of the ARMCI_Put, ARMCI_Get and ARMCI_Acc tests, we perform the latency and bandwidth benchmarks. The number of iterations are varied with size (with more number of iterations for smaller messages to a very few number of iterations for large messages). The latency is measured as the time it takes to perform each of the one-sided communication primitives. The bandwidth is reported as the inverse of the latency for the message size (in bytes).

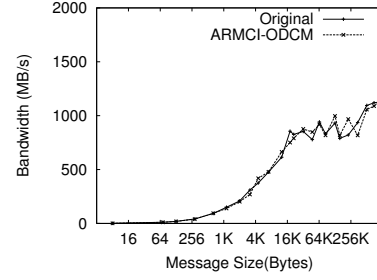
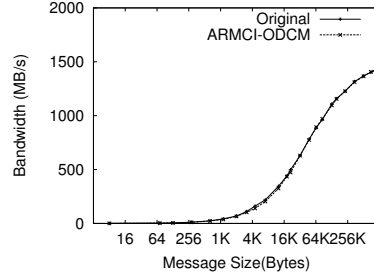
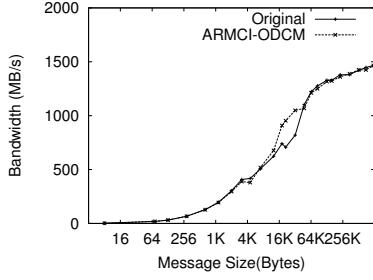


Fig. 5. ARMCI Get Unidirectional Bandwidth Fig. 6. ARMCI Put Unidirectional Bandwidth Fig. 7. ARMCI Accumulate Unidirectional Bandwidth

Figures 5, 6, and 7 show the performance comparison of the ARMCI-ODCM with the Original implementation using ARMCI_Get, ARMCI_Put and ARMCI_Acc primitives, respectively. We observe that the peak bandwidth achieved by ARMCI_Put and ARMCI_Get is 1473 MB/s. Since the contiguous data transfer uses RDMA, each of the ARMCI-ODCM and Original implementations are able to achieve the peak network bandwidth. As noticeable from the graphs, ARMCI-ODCM incurs negligible overhead in comparison to the Original implementation.

The peak bandwidth observed by ARMCI_Acc is 1327 MB/s. The primary overheads in the ARMCI_Acc compared to the ARMCI_Put and ARMCI_Get primitives is due to inability to use zero-copy for data transfer and computation at the target for the accumulate operation. However, there is no overhead incurred by the ARMCI-ODCM implementation, in comparison to the Original implementation.

Figures 8, 9, and 10 show the performance comparison of the ARMCI-ODCM with the Original implementation using ARMCI_GetS, ARMCI_PutS and ARMCI_AccS, respectively. The peak ARMCI_GetS bandwidth achieved by ARMCI-ODCM and the Original implementation is 1358 MB/s and 1337 MB/s, which is less than 2% of the performance difference. We attribute these differences to the system noise. Similarly, the peak bandwidth achieved for ARMCI_PutS for ARMCI-ODCM and the Original implementation is 1434 MB/s and 1419 MB/s, respectively. The peak ARMCI_AccS bandwidth achieved by these implementations is 1339 MB/s and 1323 MB/s, respectively. The peak ARMCI_AccS bandwidth is slightly lesser than the ARMCI_PutS bandwidth, since accumulate operation requires computation at the target node.

Figure 11 shows the performance comparison of ARMCI-ODCM with Original implementation using ARMCI_Put latency primitive. The 8 byte message latency observed is 3.74 us and 3.79 us for ARMCI-ODCM and Original implementation, respectively. Clearly, the ARMCI-ODCM implementation incurs negligible overhead compared to the Original implementation.

While there are multiple methods for measuring latency (like round trip latency with MPI), and measuring bandwidth (sending multiple outstanding messages from the origin to the target), for any of these methods, we have not observed any

overhead incurred by the ARMCI-ODCM in comparison to the Original implementation (results not included in the paper).

Figure 12 shows the overall timing of connection establishment with increasing number of processes. The micro-benchmark compares the performance of ARMCI-ODCM and Original respectively. For connecting two processes, the Original implementation takes 820 us, while it takes only 500 us for ARMCI-ODCM to connect two processes, reducing the connection time by 38%. Since the time scales linearly with the increasing number of processes, the overall connection time reduces significantly with the increasing number of processes. For 4096 processes, it takes 3.2 seconds for Original implementation, while it takes 2.1 seconds for ARMCI-ODCM. To the best of our knowledge, this is the first design and performance evaluation of parallel implementation for connection management with InfiniBand using reliable transport semantics.

For the micro-benchmarks, we do not observe an improvement in the connection memory utilization, since each of the processes are connected with every other process. In the next section, we present the performance evaluation with benchmarks, and discuss the overall number of connections in addition to comparing overall performance.

For the rest of the section, we present the results only with the overlap protocol for connection establishment, since it performs better than the non-overlap protocol for increasing number of processes, as shown above.

C. Performance Evaluation with Benchmarks

In this section, we present the performance evaluation with benchmarks. We use a variety of benchmarks for performance evaluation. Besides, LU factorization, we use an ARMCI version of the RandomAccess Benchmark, which creates a global table for updates, allowing different processes to update different locations of the global table in a one-sided manner using Accumulate operation. As allowed by the MPI version of the RandomAccess benchmark, a total of 1024 updates are coalesced before the update vector is actually transferred. Lennard Jones simulation uses Global Arrays as the programming model. The simulation uses force decomposition and the entire force matrix is divided into multiple blocks for dynamic load balancing. The force between two atoms/particles can be approximated by Lennard Jones potential energy function. Us-

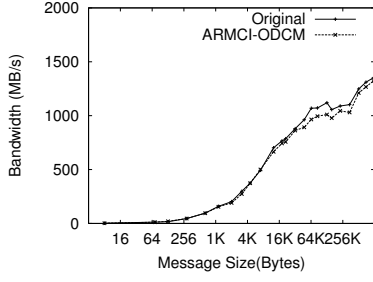


Fig. 8. ARMCI Get Strided Unidirectional Bandwidth

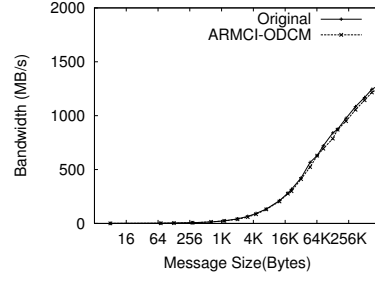


Fig. 9. ARMCI Put Strided Unidirectional Bandwidth

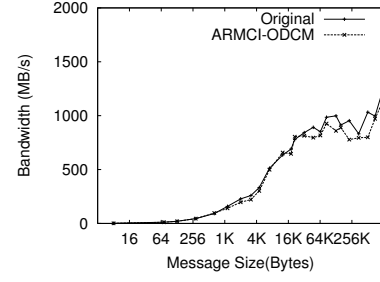


Fig. 10. ARMCI Accumulate Strided Unidirectional Bandwidth

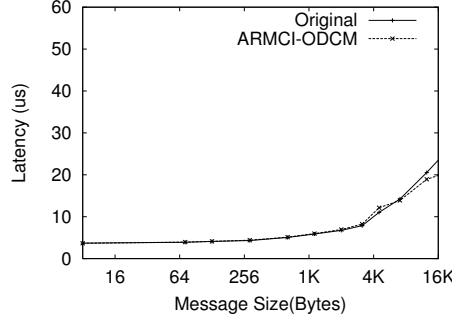


Fig. 11. ARMCI Put Latency

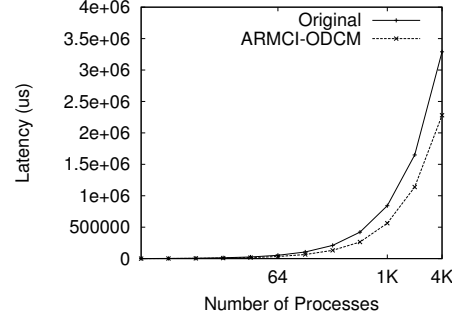


Fig. 12. Connection Establishment Latency

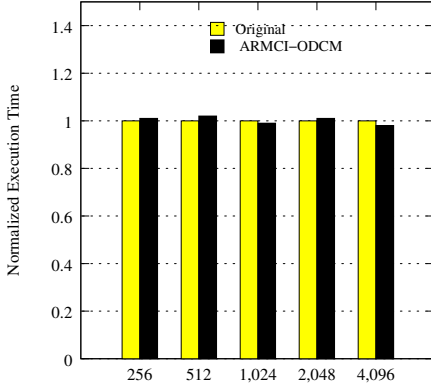


Fig. 13. LU Factorization

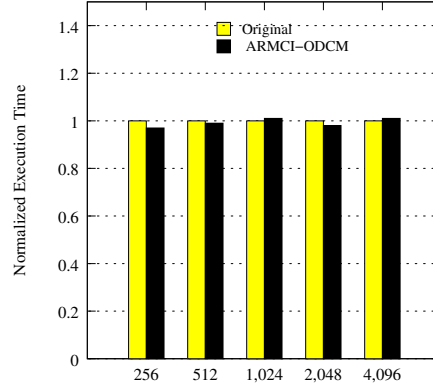


Fig. 14. RandomAccess Benchmark

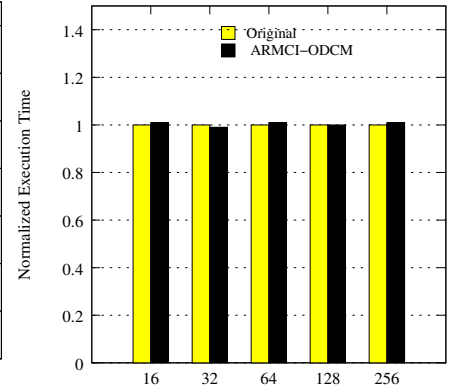


Fig. 15. Lennard Jones Simulation

ing Newton's laws of equation and Velocity-Verlet algorithm, the velocities and coordinates are updated for the next time step.

Figure 13 compares the performance of LU factorization for up to 1024 processes using ARMCI-ODCM and Original implementations. We see that the relative times of execution are very similar, showing the efficacy of the ARMCI-ODCM design. Figure 14 shows the performance of ARMCI-ODCM and Original implementation using RandomAccess Benchmark. We observe that the Giga Updates per Second (GUPS) observed under both implementations show a difference ranging from 2%-3%, which can be attributed to the system noise. Similar results are observed for the Lennard Jones simulation for increasing number of processes with Figure 15.

Tables I and II show the average number of client-server

and server-client connections created for increasing number of processes using various benchmarks. We observe that for Lennard-Jones simulation, the overall number of client-to-server connections and server-to-client connections required are equal to the number of nodes and processes respectively. Due to this communication pattern of the benchmark, we do not observe any improvement in the overall connection memory footprint. However, we do not observe any overhead incurred by ARMCI-ODCM implementation. A similar trend is also observed for RandomAccess benchmark, the communication pattern requires creation of almost all client-server and server-client connections. However, we do not observe any performance degradation here with ARMCI-ODCM in comparison to the Original implementation.

However, the LU factorization uses significantly lesser

number of connections in comparison to the total number of processes. As shown in the table, with 1024 processes, the average number of client-server and server-client connections are 31 and 263, respectively. This leads to reduction of connection memory by 65% and 73% for client-server and server-client connections respectively, without incurring any performance loss by using ARMCI-ODCM in comparison to the Original implementation.

TABLE I
AVERAGE NUMBER OF CLIENT-SERVER CONNECTIONS - BENCHMARKS

	128	256	512	1024
LU Factorization	12	19	25	31
Lennard Jones	22	44	88	176
RandomAccess	21	43	87	175

TABLE II
AVERAGE NUMBER OF SERVER-CLIENT CONNECTIONS - BENCHMARKS

	128	256	512	1024
LU Factorization	71	112	197	263
Lennard Jones	128	256	512	1024
RandomAccess	123	251	507	1021

D. Performance Evaluation with STOMP

In this section, we present the performance evaluation of STOMP with ARMCI-ODCM comparing its performance with the original implementation. The STOMP simulator solves the partial-differential equations that describe the conservation of mass or energy quantities by employing integrated-volume finite-difference discretization to the physical domain and backward Euler discretization to the time domain. The resulting equations are nonlinear coupled algebraic equations, which are solved using Newton-Raphson iteration. The simulator has been written with a variable source code that allows the user to choose the solved governing equations (e.g., water mass, air mass, dissolved-oil mass, oil mass, salt mass, thermal energy) [18].

Figures 16, 17 and 18 show the performance evaluation of STOMP for 200x200x15, 400x400x15 and 800x800x15 grid size respectively. We simulate the application for a period of twenty minutes for the evaluation varying the number of processes ranging from 128 to 4096. Using strong scaling, we evaluate a problem to a number of processors, till the overhead of communication prevents the problem to scale to a higher number of processors. In each of the figures, we observe that ARMCI-ODCM does not incur any overhead in comparison to the original implementation. During our experimentation, we observe a variance of up to 3%, which we attribute to the system noise.

1) *Connection Memory Utilization with STOMP*: Tables III and IV show the average number of client-to-server and server-to-client connections with STOMP while increasing the number of processes. Due to the limitations of the memory bandwidth, we use six processes per node and dedicate the

remaining two cores for performing computations and data transfers on behalf of the processes. As a result, the number of nodes used for running 128 to 4096 process jobs is 22, 44, 88, 176, 352 and 704 nodes, respectively. Since there is one data-server per node, and clients need to connect only to the data server on a node, the average number of client-server connections reflect trends with increasing number of nodes. We observe that for 128 processes, the number of client-server connections is 3, while the number of client-server connections for 4096 processes is 10. Compared to the original implantation, this leads to reduction in connection memory utilization by 90% to 70 times reduction for 4096 processes. We observe that the number of actual connections increase slowly while doubling the number of processes. Clearly, we expect the memory utilization of ARMCI-ODCM to be even higher for larger processor counts.

TABLE III
AVERAGE NUMBER OF CLIENT-SERVER CONNECTIONS - STOMP

	128	256	512	1024	2048	4096
200x200x15	3	3				
400x400x15		3	3	6		
800x800x15				6	10	10

TABLE IV
AVERAGE NUMBER OF SERVER-CLIENT CONNECTIONS - STOMP

	128	256	512	1024	2048	4096
200x200x15	21	22				
400x400x15		22	23	35		
800x800x15				35	61	62

Table IV shows the average number of server-to-client connections. Since the Original implementation maintains a pairwise connection between each client and server, the average number of server-to-client connections are equal to the number of processes in the job. For 128 processes, the average number of connections reduce by 6 times, while almost a 66 times improvement is observed for 4096 processes. Since STOMP performs clique based communication, multi-fold improvement is expected at larger processor counts.

Clearly, ARMCI-ODCM is able to reduce the connection memory footprint significantly for Global Arrays applications like STOMP, without incurring an overhead for overall execution time. For other Global Arrays and ARMCI benchmarks, depending up on the kernel computation and communication pattern, ARMCI-ODCM is able to show overall reduction in connection memory, without performance degradation in all cases.

V. RELATED WORK

On demand connection management with InfiniBand for MPI two-sided communications based programming models has been studied by multiple researchers [28], [12], [11], [13], [14], [15].

Wu et al., have presented the impact of on-demand connection management with VIA [28]. However, the overall

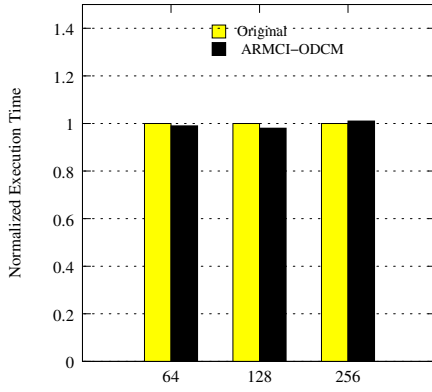


Fig. 16. STOMP-200x200x15

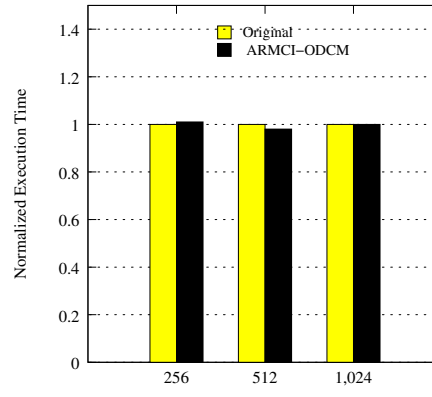


Fig. 17. STOMP-400x400x15

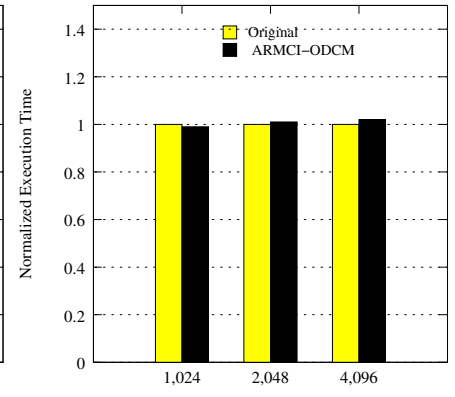


Fig. 18. STOMP-800x800x15

memory utilization is much lesser compared to the connection memory with InfiniBand. Yu et al., have presented a design for on-demand connection management using RDMA connection manager and Openfabrics interface with InfiniBand [12]. However, the reliability protocol is not clear in the design. To alleviate the limitations of the work presented by Yu et al., [12], Koop et al., presented designs using multiple transport semantics with InfiniBand [4]. Using unreliable datagram connection-less transport semantics with InfiniBand, Koop et al., presented designs for copy based approaches [11]. Copy based approaches are applicable for MPI, since it has implicit synchronization. Using zero-copy based approaches with InfiniBand, Koop et al also presented design and performance evaluation for using zero copy based approach using send/receive mechanism provided by InfiniBand, since the unreliable datagram transport semantics do not support RDMA [13]. Koop et al., have also presented multi-transport InfiniBand semantics using unreliable datagram and reliable connection transport semantics [14]. Recently, work related to extended reliable connection semantics has also been presented. Using this transport, multiple processes on the same node can share the data transfer queue, allowing memory requirements to increase corresponding to the number of nodes, rather than the number of processes [15]. This has an important consequence for upcoming large scale multi-core systems (more than 8/16 cores per node).

However, none of the above work has focused on designing on-demand connection management protocols for one-sided communication models like Global arrays [9], which fundamentally differ from the two-sided message passing as discussed in the paper. To the best of our knowledge, this is the first study of the on-demand connection management with one-sided communication runtime system like ARMCI [16]. Although, we have used ARMCI [16] as the communication runtime system, similar design is applicable to runtime systems of other runtime systems like GASNet [17].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a design for on-demand connection management for PGAS model's runtime

systems using Aggregate Remote Memory Copy Interface (ARMCI) [16], the runtime system of Global Arrays [9]. Within the solution space, we have presented the overall design of ARMCI over InfiniBand, multiple approaches for on-demand connection creation and establishment protocol; simplification of the connection establishment state machine and reliability protocol due to the asymmetric nature of communication in PGAS runtime systems. We have implemented our design and evaluated it with micro-benchmarks and benchmarks (LU Factorization, RandomAccess, Lennard Jones simulation) and application (Sub-surface Transport over Multiple Phases (STOMP) [18]). With the performance evaluation for up to 4096 processors, we are able to have a multi-fold reduction in connection memory with a negligible degradation in performance for the benchmarks. Our performance evaluation with STOMP improves a connection memory utilization by 66 times with no performance degradation. To the best of our knowledge, this is the first design, implementation and evaluation of on-demand connection management approach with PGAS programming models,

We plan to continue our research on impact of multiple InfiniBand transport semantics on PGAS models. Specifically, we plan to explore the impact of unreliable datagram semantics, and support for multi-transport runtime systems using Extended Reliable Connection (XRC) transport semantics in near future. We will also consider on-demand connection termination, which is of impending interest, as a result of the scalable work stealing primitives being designed using PGAS programming models.

REFERENCES

- [1] D. A. Patterson, D. E. Culler, and T. E. Anderson, "A Case for NOW (Networks of Workstations)," in *Principles of Distributed Computing*, 1995, pp. 17–28.
- [2] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, February 1995.
- [3] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, 2002.
- [4] InfiniBand Trade Association, "InfiniBand Architecture Specification, Release 1.2," October 2004.
- [5] "TOP 500 Supercomputer Sites," <http://www.top500.org>.

- [6] "Texas Advanced Computing Center," <http://www.tacc.utexas.edu/>.
- [7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," vol. 22, no. 6, 1996, pp. 789–828.
- [8] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the Message-Passing Interface," in *Euro-Par, Vol. 1*, 1996, pp. 128–135.
- [9] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers," 1994, pp. 340–349.
- [10] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences," in *Proceedings of Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, March 2007.
- [11] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters," in *International Conference on Supercomputing*, 2007, pp. 180–189.
- [12] W. Yu, Q. Gao, and D. K. Panda, "Adaptive Connection Management for Scalable MPI over InfiniBand," in *IPDPS*, 2006.
- [13] M. J. Koop, S. Sur, and D. K. Panda, "Zero-copy Protocol for MPI using Infiniband Unreliable Datagram," in *International Conference on Cluster Computing*, 2007, pp. 179–186.
- [14] M. J. Koop, T. Jones, and D. K. Panda, "MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand," in *International Symposium on Parallel and Distributed Computing*, 2008, pp. 1–12.
- [15] M. J. Koop, J. K. Sridhar, and D. K. Panda, "Scalable MPI design over InfiniBand using eXtended Reliable Connection," in *International Conference on Cluster Computing*, 2008, pp. 203–212.
- [16] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," vol. 1586, 1999.
- [17] D. Bonachea, "GASNet Specification, v1.1," October 2002.
- [18] Subsurface Transport over Multiple Phases, "STOMP," <http://stomp.pnl.gov/>.
- [19] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," in *Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 533–546.
- [20] Sandia Portals, "Portals Message Passing Interface," <http://www.cs.sandia.gov/portals>.
- [21] "Myricom Corporation," <http://www.myri.com/>.
- [22] "Quadrics Corporation," <http://www.quadrics.com/>.
- [23] Cray XT4 Jaguar, "National Center for Computational Sciences," <http://www.nccs.gov/jaguar>.
- [24] IBM BlueGene/P Intrepid, "Advanced Leadership Computing Facility," <http://www.alcf.anl.gov>.
- [25] "Chinook SuperComputer, Environmental Molecular Science Lab, PNNL," <http://emsl.pnl.gov>.
- [26] "Voltaire Technologies," <http://www.voltaire.com/>.
- [27] "Mellanox Technologies," <http://www.mellanox.com/>.
- [28] J. Wu, J. Liu, P. Wyckoff, and D. K. Panda, "Impact of On-Demand Connection Management in MPI over VIA," in *International Conference on Cluster Computing*, 2002, pp. 152–159.