Network-Friendly One-Sided Communication Through Multinode Cooperation on Petascale Cray XT5 Systems

Xinyu Que[†] Weikuan Yu[†‡] Vinod Tipparaju[‡] Jeffrey S. Vetter[‡] Bin Wang[†]
Department of Computer Science[†] Computer Science & Mathematics[‡]
Auburn University, AL 36849 Oak Ridge National Laboratory
{*xque,wkyu,bzw0012*}@*auburn.edu* {*tipparajuv,vetter*}@*ornl.gov*

***Abstract*—One-sided communication is important to enable asynchronous communication and data movement for Global Address Space (GAS) programming models. Such communication is typically realized through direct messages between initiator and target processes. For petascale systems with 10,000s of nodes and 100,000s of cores, these direct messages require dedicated communication buffers and/or channels, which can lead to significant scalability challenges for GAS programming models. In this paper, we describe a network-friendly communication model, *multinode cooperation*, to enable indirect one-sided communication. Compute nodes work together to handle one-side requests through (1) request forwarding in which one node can intercept a request and forward it to a target node, and (2) request aggregation in which one node can aggregate many requests to a target node. We have implemented multinode cooperation for a popular GAS runtime library, Aggregate Remote Memory Copy Interface (ARMCI). Our experimental results on a large-scale Cray XT5 system demonstrate that multinode cooperation is able to greatly increase memory scalability by reducing communication buffers required on each node. In addition, multinode cooperation improves the resiliency of GAS runtime system to network contention. Furthermore, multinode cooperation can benefit the performance of scientific applications. In one case, it reduces the total execution time of an NWChem application by 52%.**

***Keywords*- GAS; ARMCI; Multinode Cooperation; Request Aggregation;**

I. Introduction

GAS (Global Address Space) or PGAS (Partitioned Global Address Space) models support data access to local and remote memory through simple shared memory styled access. Because of the attractiveness of their simple access model, PGAS languages such as Unified Parallel C (UPC) [2] and Co-Array Fortran (CAF) [10], and GAS libraries such as Global Arrays (GA) Toolkit [1] are becoming increasingly popular. X10 [3] from IBM and UPC [14] have also pioneered a slightly different category of PGAS model, termed *Asynchronous* Partitioned Global Address Space model.

All the above mentioned GAS languages and libraries use the services of one-sided communication library (also referred to as the GAS Runtime) for their communication needs. GAS languages normally convert their data transfers through compilation techniques into one-sided communication messages on distributed memory architectures. They have a translation layer that translates memory access to various one-sided messages, with which programmers no longer have to orchestrate complicated message passing schemes among many pairs of parallel processes.

One-sided communication is typically realized through direct messages between initiator and target compute nodes. To realize efficient and asynchronous one-sided communication primitives, communication buffers and/or channels need to be preallocated for processes to initiate an operation without the involvement of the targeted process. On petascale systems with 10,000s of nodes and 100,000s of cores, these direct communication buffers and/or channels will amount to a hefty memory requirement, and impose a critical scalability challenge for GAS programming models.

Another serious challenge to GAS programming models is the potential contention that could be caused by many concurrent one-sided messages to a single target node. Because one-side messages are directly sent to the same target, network paths of these requests will converge at one or more network endpoints. In PGAS programming models, this is quite likely when thousands of processes simultaneously access one data element in the global address space. Such scenarios can cause severe network contention, placing significant burdens on the physical network. For example, the Seastar2+ network adopted by the Cray XT5 systems will resort to a throttling mechanism to handle network congestion, typically causing serious slowdown of the entire network and jeopardizing system productivity.

In this paper, we describe *multinode cooperation*, a network-friendly communication model that supports indirect one-sided communication, and overcomes the challenges caused by direct one-sided communication between any pair of initiator and target nodes. In multinode cooperation, compute nodes form a multinode group and work together to handle one-sided communication requests. Their cooperation occurs in two ways: (1) request forwarding in which one node can intercept a request and forward it to the target node, and (2) request aggregation in which one node can aggregate many requests to the same target node. With multinode cooperation, compute nodes no longer have to create one set of communication buffers for all possible pairs of peer processes. Instead, they divide the requirement of communication buffers amongst themselves in a cooperative manner. When a request reaches one node in a multinode group, it will be forwarded to the target node, and handled accordingly. Through request aggregation, multinode cooperation also exploits the presence of multiple requests to the same target node. It consolidates them together to reduce network contention, thereby alleviating the pressure to the underlying physical network.

We have implemented multinode cooperation for a popular GAS run-time library, Aggregate Remote Memory Copy Interface (ARMCI). Our experimental results on large-scale Cray XT5 systems demonstrate that multinode cooperation is able to greatly increase the memory scalability by reducing communication buffers. In addition, multinode cooperation increases the resiliency of GAS runtime system in handling network contention caused by transient and irregular communication patterns. Furthermore, multinode cooperation is shown to significantly improve the performance of scientific applications. In one case, it reduces the total execution time of an NWChem application by 52%.

The rest of the paper is organized as follows. Section II discusses background and related work. Section III describes the design and implementation of multinode cooperation. Experimental results are provided in Section IV. Finally, Section V concludes the paper.

II. Background and Related Work

A. An Overview of ARMCI

ARMCI has recently been enabled for Cray XT5 using the native portals communication library [17]. ARMCI guarantees that its one-sided operations are fully unilateral, i.e., may complete regardless of the actions taken by the remote processes. In particular, polling the application by remote processes (implicitly when making a library call, or explicitly by calling provided polling interface) is not required for communication progress. This is realized by introducing a communication helper thread (a.k.a communication server) at each compute node. This communication helper thread is created by the lowest ranked process (*master*) on a node. An area of shared memory is allocated for these processes. The communication server handles remote one-sided requests on behalf of all local processes, and exchanges data with them through the shared memory. Similar to what described earlier, the communication server pre-allocates buffers and related data structures for remote requests, in order to support direct one-sided communication for all operations (particularly for lock, unlock, accumulate, and noncontiguous data transfer operations) and allow one process to asynchronously initiate an operation without the involvement of the targeted process.

B. Related Work

The scalability of communication runtime involves a variety of complicated design issues, including process management, selection of connection models, data communication, communication buffer management, as well as flow control. The design and implementation of MPI on Portals was first described by Brightwell *et al.* [7]. This has been one of the reference implementations for other programming models on top of Portals, in which communication protocols for different size messages are elaborated. Bonachea *et al.* [6] recently ported GASNet to the Portals communication library on the Cray XT platform to support UPC and other GAS models. Generic issues such as enabling communication operations, handling requests/replies, and flow control were discussed. Tipparaju *et al.* [17] designed and implemented a scalable ARMCI communication library and demonstrated its strength in enabling GA and a real world scientific application, NWChem.

Many other efforts studied the communication scalability of programming models. Sur *et al.* [16] and Shipman *et al.* [15] studied the use of shared receive queue to increase the scalability of MPI communication resources. Koop *et al.* [13] exploited the use of message coalescing to reduce the memory requirement for MPI on InfiniBand clusters. Chen *et al.* [9] optimized communication UPC through a combination of techniques including redundancy elimination, split-phase communication, and communication coalescing. Chen *et al.* [8] investigated the use of compiler techniques that could automatically schedule data transfers for non-blocking communication, thereby achieving better computation and communication overlap. Yu *et al.* [18] proposed cooperative server clustering to improve the scalability of the ARMCI. This work is based on recent efforts from Tipparaju *et al.* [17] and Yu *et al.* [18] for a network friendly one-sided communication model with better scalability. We extend our previous work to support request aggregation and systematically evaluate the benefits of multinode cooperation in terms of memory requirement and network contention.

III. Design and Implementation of Multinode Cooperation

As described in Section II-A, ARMCI realized one-sidedness of its operations through a communication server (CS), which is a thread created by the lowest ranked process at each compute node. Every communication server pre-allocates communication buffers for potential requests from all remote peer processes. With only two 16-KB buffers per process, it would require 1024 MB to support parallel programs with 32,000 processes, and 32 GB for future programs with a million processes. Clearly, a more scalable solution is needed for petascale supercomputer and future exascale machines.

Multinode cooperation is intended to address the scalability challenge of communication buffers, as well as the associated network contention, caused by one-sided messages in ARMCI's original direct communication model. In this section, we first discuss the software architecture of multinode cooperation. Then we describe the flow of request forwarding and request aggregation of multinode cooperation in detail.

A. Software Architecture of Multinode Cooperation

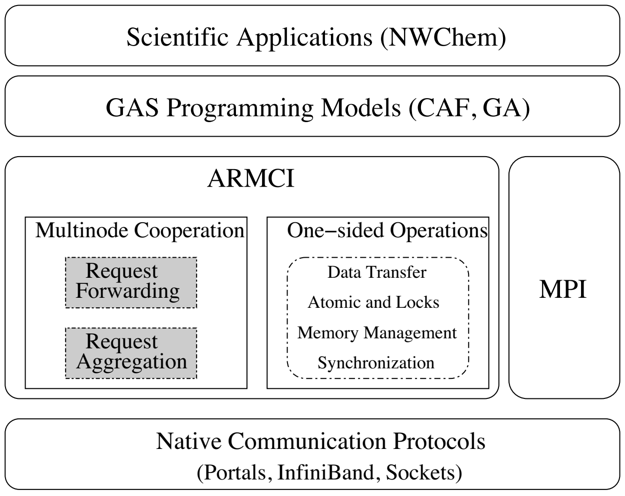


Figure 1: Software Architecture of Multinode Cooperation

Figure 1 shows the software architecture of multinode cooperation. On a system that supports scientific applications over GAS programming models, such as NWChem, ARMCI will support the required one-sided operations, including data transfer, atomic and locks, memory management, and synchronization. Multinode cooperation extends ARMCI with an indirect communication model for transmitting one-sided requests in these operations. It includes two important key techniques: request forwarding and request aggregation.

Multinode cooperation fundamentally addresses the scalability issues of direct one-sided request messages. Instead of allocating one set of buffers for all remote processes on each node, multiple nodes form a cooperative multinode group to allocate buffers. Communication servers on these nodes divide incoming requests from outside processes amongst themselves. For example, for a program with N processes, one communication server roughly has to preallocate $N-1$ sets of communication buffers in the original ARMCI. When a K-node group is formed through multinode cooperation, one communication server will only need to preallocate $(N-1)/K$ sets of communication buffers. Because of the division of requests among servers, a multinode group effectively reduces each server's communication buffer requirement by the size of the multinode group. The servers in a multinode group then cooperate and handle one-sided requests from processes outside the group. When one request reaches any server in the multinode group, it will be forwarded to the actual target server.

With multinode cooperation, most of one-sided communication requests are no longer sent directly to the destination communication server. This brings in another beneficial feature. The risk of network contention caused by many requests to a single hot-spot target node is significantly alleviated, because requests are first buffered by cooperative nodes in a multinode group, and aggregated if they arrive closely with each other in time. Request aggregation is described in more detail below.

B. Request Forwarding and aggregation

The original ARMCI has a very simple communication model to support direct one-sided operations. Figure 2(a) shows the flow of request and reply between a pair of processes (P_r and P_t). The communication server CS_T (co-located with P_t) receives the request from P_r on behalf of P_t. As the requested operation completes, CS_T returns a corresponding reply or acknowledgment (ack/rep) to P_r. This forms a direct request/reply pair and a simplified flow control scheme between P_r and CS_T.

The key of multinode cooperation is its indirect request communication model. This is achieved through request forwarding and request aggregation. Figure 2(b) shows the flow of requests and replies in multinode cooperation. Three processes (P_{r0}, P_{r1}, and P_{r2}) are

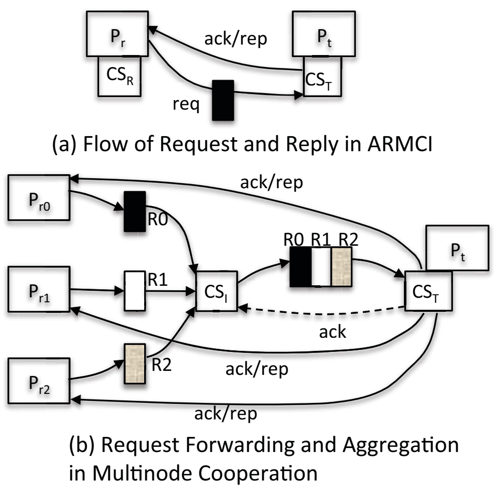

P_r
CS_R
ack/rep
P_t
CS_T
req
(a) Flow of Request and Reply in ARMCI
P_r0
R0
ack/rep
R0 R1 R2
P_t
P_r1
R1
CS_I
CS_T
ack
R2
ack/rep
P_r2
ack/rep
(b) Request Forwarding and Aggregation in Multinode Cooperation

Figure 2: Request Handling in ARMCI and Multinode Cooperation

initiating three one-sided requests (R0, R1, and R2) to a target process (P_t), through the communication server (CS_I) at the same intermediate node. CS_I receives these requests, and detects that they are targeting for the same communication server CS_T. So these requests are aggregated together into a single request and sent to CS_T. Only one acknowledgment is needed for the aggregation request. CS_T receives a combined request, and processes the embedded requests separately. In the end, it sends back individual replies or acknowledgments back to three requesting processes.

Request forwarding can be viewed as a special case of the same diagram, where requests are not allowed to be aggregated together. When a request arrives at CS_I, it is immediately forwarded to CS_T. There must be a separate acknowledgment for every request message.

Event-Driven Aggregation Window – To allow request aggregation, a communication server must hold on to one request and wait for the arrival of more requests. When requests arrive closely within each other, there are plenty of opportunities to aggregate requests. However, the communication server should not keep a request for too long when no more requests arrive in time. On the other hand, the communication server cannot busy wait for the arrival of new requests, which would consume a lot of CPU cycles. We address this issue through an event-driven aggregation window. Upon the arrival of a new request, the communication server records its timestamp. It is then blocked, waiting for the arrival of more requests. Every portals message generates an event on the communication server, and wakes up the communication server to perform possible request aggregation. A request will be forwarded when the aggregation window expires. An extra event is introduced to wake up the communication server when no portals messages are communicated. Within a multinode group, an empty message is periodically initiated by a communication server to its peer. This message will generate an event to wake up blocked communication servers, thereby breaking a potential stalemate caused by a held request.

IV. Performance Evaluation

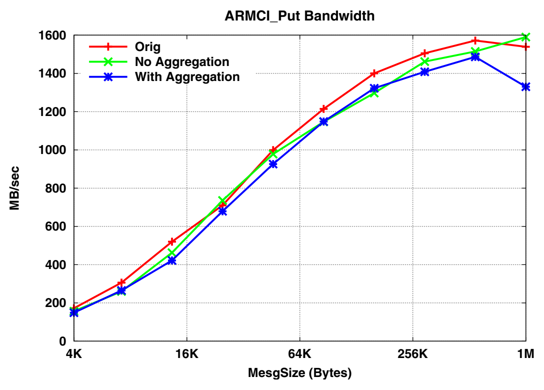
We have conducted an extensive set of experiments to evaluate multinode cooperation on Jaguar at ORNL. Our measurements include (1) the performance of ARMCI one-sided operations; (2) the amount of memory consumption; (3) the impact to network contention; and (4) the performance of application benchmarks. In all experiments, we compare the original ARMCI with the implementation of multinode cooperation. Two modes of multinode cooperation are included: one that supports only request forwarding without aggregation (*No Aggregation*), and the other that supports both request forwarding and aggregation (*With Aggregation*).

A. ARMCI One-sided Operations

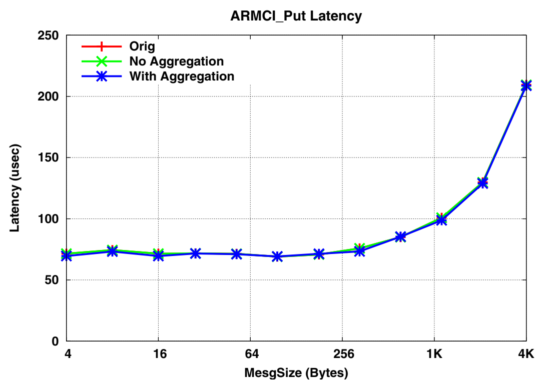
ARMCI offers a rich set of one-sided communication primitives for GAS programming models. These include (1) contiguous and noncontiguous data transfer operations, (2) atomic operations, (3) locks, and (4) synchronization operations. While multinode cooperation is intended to address challenges faced by direct one-sided communication in the original ARMCI, it is important to measure the performance impact of multinode cooperation to these one-sided operations.

1) Contiguous Data Transfer Operations: ARMCI supports contiguous data transfer operations, including direct put and direct get. On the Cray XT5, these direct put/get operations transfer contiguous data directly between source and destination memory, using native portals put and get operations on the Seastar2+ network. No one-sided requests are sent for these operations, and communication servers are not involved for these operations. We measure the performance of direct put/get operations across 16 nodes, each with 12 processes. These nodes form four groups of cooperative nodes. Figure 3 shows the latency and bandwidth performance comparison between ARMCI and multinode cooperation.

It is clear that our design of multinode cooperation has very little impact on the performance of contiguous data transfer operations. Note that, for succinctness, we only show the performance for direct put operations. The comparison is the same for direct get operations.

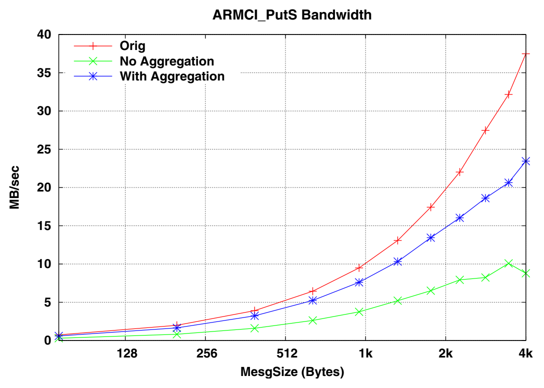


(a) ARMCI_Put Bandwidth

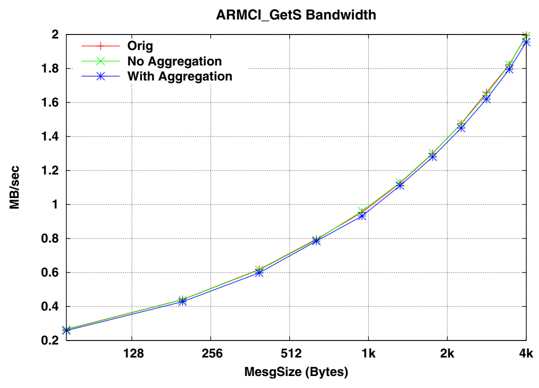


(b) ARMCI_Put Latency

Figure 3: ARMCI_Put Latency and bandwidth



(a) ARMCI_PutS



(b) ARMCI_GetS

Figure 4: Bandwidth of Noncontiguous Operations

2) Noncontiguous Data Transfer Operations: Multidimensional data arrays are commonly adopted by scientific applications for numerical analysis and matrix calculation. When such an array is decomposed into many parallel processes, each process typically owns a noncontiguous set of data elements. ARMCI supports the movement of such noncontiguous data through vectored I/O and strided I/O. The former is a generalized I/O format that describes noncontiguous data segments with a series of <addr, length> pairs; the latter is an optimization when segments are of the same length and distance from each other.

We have measured the performance of ARMCI strided data transfer. Our experiments are conducted on sixteen nodes each with 11 processes. Processes on the first node are paired with, and initiate one-sided ARMCI_PutS (and ARMCI_GetS) operations to, their counterparts on the last node. Figure 4 shows the performance results of ARMCI for short messages, with and without multinode cooperation. ARMCI_PutS requests are usually large and contain data inside. So they cannot be merged. Large requests with data need to be forwarded to the target server seperately as the size of aggregation buffer is limited. This leads to significant bandwidth degradation for ARMCI_PutS operations as shown by *No Aggregation*. On the other hand, ARMCI_GetS requests have to retrieve data separately. This leads to very low bandwidth for ARMCI_GetS operations in general. But there is no difference between the original ARMCI and multinode cooperation. These results indicate that, while the performance of noncontiguous data put operations is affected by the overhead of request forwarding, request aggregation can significantly cut down such overhead, and bring the performance close to the original ARMCI.

3) Atomic and Synchronization Operations: ARMCI supports a number of atomic and synchronization operations for GAS models. These include lock, accumulate, and fetch-&-add. The lock operation acquires a specified mutex on the target process on behalf of an initiating process. The accumulate operation atomically updates one or more variables on the target process. The fetch-&-add operation retrieves an integer variable at a remote location, and at the same time atomically updates the value by an integer.

We measure the performance of these operations across 16 nodes. These nodes are grouped into four sets of cooperative nodes. All processes are paired with each other for atomic and synchronization operations. In order to evaluate the performance of multinode cooperation, we tested different numbers of processes (4, 8, and 11) per node. For example, a process on Node 0 initiates lock, accumulate, or fetch-&-add operations 1000 times (after the first 50 warm-up operations) to its counterpart on Node 15. The average time is calculated as the time for an operation.

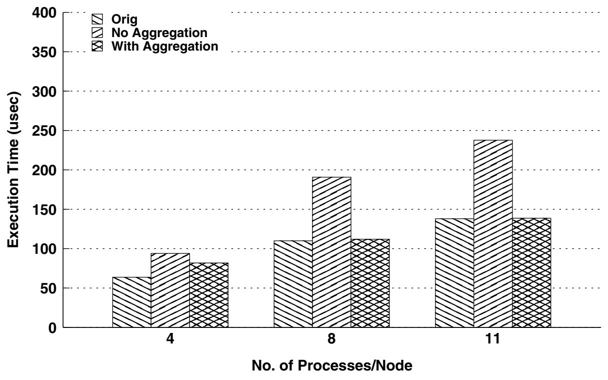
Figure 5 shows the performance results for all three operations. Again, without request aggregation, multinode cooperation does not deliver the same performance compared to the original ARMCI. However, request aggregation significantly reduces the overhead of request forwarding by combining the forwarding costs of multiple requests into one.

Taken together, our microbenchmark evaluation results indicate that, while indirect one-sided communication from multinode cooperation causes performance overhead to atomic and synchronization operations, request aggregation helps reduce the overhead of request forwarding. The resulting performance for all three operations in multinode cooperation is very close to that of the original ARMCI.

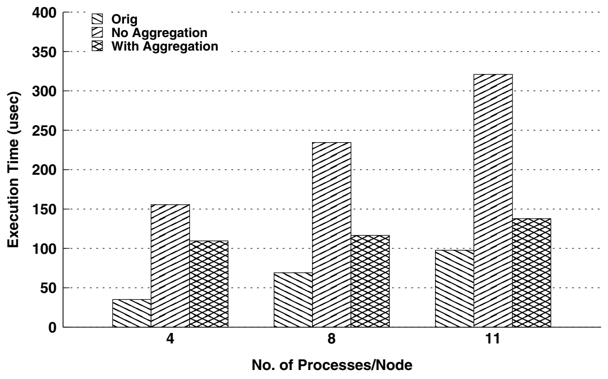
B. Memory Consumption

Jaguar runs the Compute Node Linux operating system. On each node, the /proc file system reports the memory footprint of all processes as the resident working set size (VmRSS). We create a simple ARMCI program that reports VmRSS from all processes. This number represents the total memory consumed by an ARMCI process at runtime before any additional memory usage at the application-level.

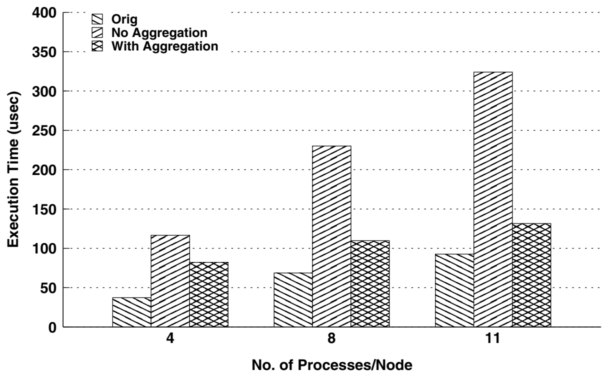
We measure the memory footprint of all ARMCI processes. Our experiments are conducted with 12 processes per node. All processes start with a memory consumption of about 612 MBytes. However, due to the allocation of request buffers by the internal communication server, each master process requires more memory for an increasing number of remote processes. The size of each buffer is 16KB; and the number of buffers per process is 4. Figure 6 shows the memory consumption of master processes. The memory consumption of the original ARMCI increases linearly. On 12,288 processes, it reaches 1,424 MBytes, an increment of 812 MBytes. On the other hand, the memory consumption of multinode cooperation reaches 720 MBytes and 725 MBytes, from a base of 604.3 MBytes under the *No Aggregation* mode, and a base of 604.8 MBytes under the *With Aggregation* mode, respectively. Compared to the original ARMCI, multinode cooperation cuts down memory consumption significantly by about seven times. To enable request aggregation, multinode cooperation consumes a little more memory, that is used as a small number of additional aggregation buffers. This test demonstrates that, compared to the original ARMCI, multinode cooperation can dramatically reduce communication buffers, thereby improving the memory scalability of ARMCI.



(a) ARMCI_Lock



(b) ARMCI_Accumulate



(c) ARMCI_Fetch & Add

Figure 5: Performance of Atomic and Synchronization Operations

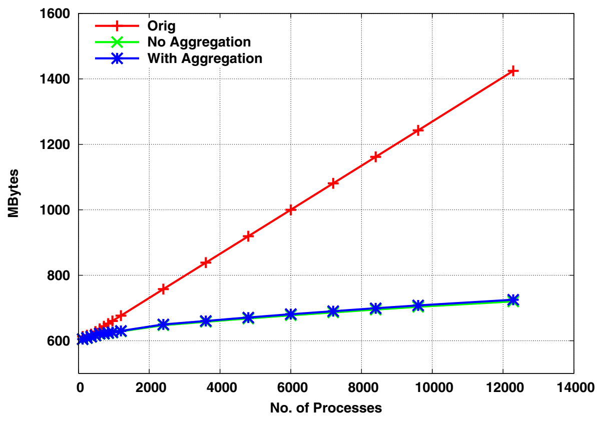


Figure 6: Benefits to Memory Consumption

C. Benefits of Multinode Cooperation to Network Contention

Multinode cooperation is also designed to address network contention in the GAS runtime. We evaluate contention for all one-sided ARMCI operations, and observe that multinode cooperation is beneficial to the contention caused by noncontiguous data transfer and atomic operations. Herein presented are results for two representative operations, atomic Fetch-&-Add and noncontiguous strided data transfer operations.

1) Contention Resilience Experiments: We define contention as the percentage of processes in a program that are contending for communication to a single process, or for access to a single data element. It is understood that contention can arise from sources outside of a program, e.g., from other programs or system services. But, for practical purposes, we consider those beyond the scope of this study, and focus on contention within one program.

We use programs with 1,024 processes for contention evaluation, 4 processes per node across 256 nodes. These numbers provide a reasonable balance between the need of many nodes to exhibit contention and the need of clarity in visualizing all data points of the results. In these programs, each process (except those on the same node with Rank 0), prepares its data as needed (vectored or strided data in the case of noncontiguous data transfer operations), and then performs one or more one-sided operations to Rank 0. This is then repeated for 20 iterations. The average time for these iterations is taken as the time to complete the operations between these processes and Rank 0.

Measurements are taken under three different contention scenarios. In the first scenario, each process sequentially performs its own one-sided operations to Rank 0, repeats for 20 iterations, and records the time. At the same time, all other processes are idle in a barrier. This effectively measures the performance of one-sided operations between Rank 0 and all other processes, without any contention. In the second scenario, each process sequentially performs the same number of operations to Rank 0, for the same number of iterations. However, in the meantime, one in every five processes performs the same operations to Rank 0, while remaining processes are idle in a barrier. Therefore this corresponds to 20% contention.

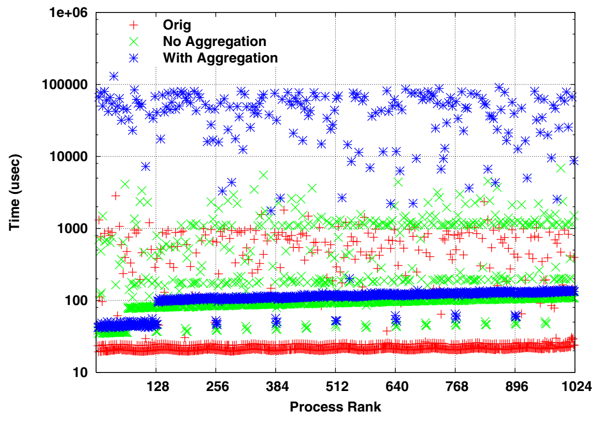
2) Atomic Fetch-&-Add: We measure the performance of fetch-&-add as a representative of atomic operations. Figure 7 shows the time for fetch-&-add operations from all remote processes to Rank 0. Comparisons are provided between ARMCI and multinode cooperation (with and without aggregation).

Figure 7(a) shows the comparison under no contention. A couple of observations can be made from this figure. First, multinode cooperation increases the time to complete fetch-&-add operations between Rank 0 and other processes. Second, the time to complete fetch-&-add gradually increases with the process rank. This increment of time is magnified with multinode cooperation due to request forwarding.

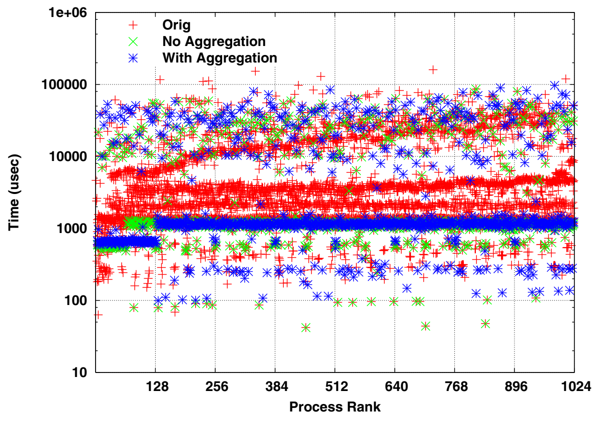
Figures 7(b) shows the comparison under 20% contention. While contention increases the time to complete atomic fetch-&-add operations for all cases, better contention resilience can be observed for both modes of multinode cooperation. Without multinode cooperation, the performance of fetch-&-add is degraded by nearly two orders of magnitude due to contention caused by direct one-sided request messages. Under 20% contention, it becomes faster to complete fetch-&-add operations for nearly all processes with multinode cooperation. It is also interesting to note that multinode cooperation reduces the variation among all processes at 20% contention. These results demonstrate that multinode cooperation can mitigate the converging pressure from many atomic operations to a single process, and lead to graceful contention resilience.

3) Strided Data Transfer: We conduct experiments to measure the performance of strided put as a representative of noncontiguous data transfer operations. Comparisons are provided for strided put operations with different size messages.

Figure 8 shows the time to complete strided put in ARMCI with and without multinode cooperation. Similar to Figure 7, several observations can be made from these results. First, multinode cooperation exhibits better contention resilience compared to the original ARMCI. Second, with request aggregation, ARMCI shows much better resilience compared to the mode with request forwarding only. Third, comparing (a) and

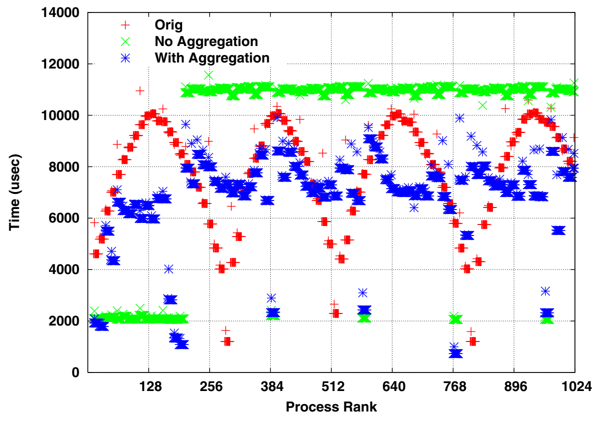


(a) Contention = 0%

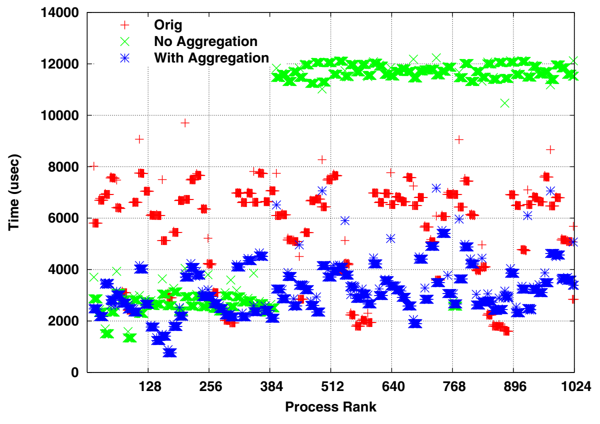


(b) Contention = 20%

Figure 7: Contention of Fetch-&-Add Operations



(a) Message size = 72 bytes



(b) Message size = 512 bytes

Figure 8: The Impact of Contention to ARMCI_PutS Operations

(b) in the figure, we conclude that the benefit is more significant for larger size one-sided requests.

D. NAS LU Application Benchmark

The LU application in the NAS parallel benchmark suite [5] has been ported to the ARMCI runtime communication protocol. It can scale to hundreds or a couple of thousand processes. We also evaluate the performance impact of multinode cooperation on NAS applications at this scale. Figure 9 shows the performance of LU on a varying number of processes. As shown in the figure, the performance results of multinode cooperation are generally on par with that of the original ARMCI. On the other hand, there is a small amount of extra overhead to enable request aggregation.

We evaluate multinode cooperation using an electronic structure method in a large-scale application NWChem [12], the SiOSi3 method for Density Functional Theory (DFT). Figure 10(a) shows the performance of SiOSi3 on 100 to 12,800 processes. Multinode

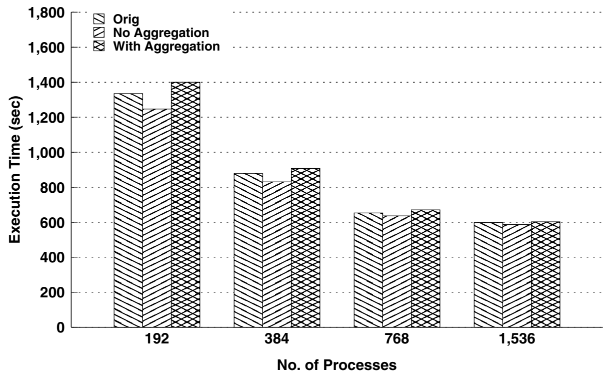


Figure 9: Performance of NAS LU

cooperation clearly performs better than the original ARMCI.

The combination of request forwarding and request aggregation can reduce the total execution time by as much as 52%.

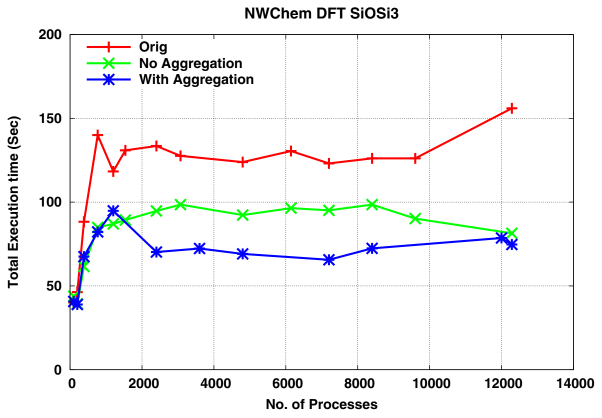


Figure 10: NWChem SiOSi3 Benchmark

V. Conclusion

We have introduced multinode cooperation as a network-friendly model that supports one-sided communication requests indirectly. Instead of allocating communication buffers to prepare for requests from all potential peer processes, multinode cooperation allows multiple nodes to form a cooperative group and work together to process one-sided communication requests. We have implemented multiple cooperation, and evaluated its benefits in terms of memory consumption and contention resilience, as well as its impact to various ARMCI one-side operations on the petascale Jaguar Cray XT5 system at ORNL.

Our experimental results have demonstrated that multinode cooperation can significantly reduce memory consumption. It also improves ARMCI's resilience to network contention caused by transient and irregular communication patterns. Moreover, multinode cooperation can improve the performance of scientific applications, e.g., reducing the execution time of a NWChem DFT method by 52%.

In the future, we look forward to further optimization of ARMCI on petascale systems. We intend to study the applicability of multinode cooperation to other one-sided communication runtime such as GASNet [6], and the performance of ARMCI multinode cooperation on petascale platforms with different physical network topologies, such as BlueGene/P [11], [4].

Acknowledgment

This work was funded in part by NSF Awards CNS-1059376 and CNS-0917137, and by UT-Battelle Grants UT-B-4000087151 and UT-B-4000086682. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Global arrays toolkit. http://www.emsl.pnl.gov/docs/global.

[2] Upc specifications, v1.2. http://www.gwu.edu/∼upc/publications/LBNL-59208.pdf.

[3] Report on experimental language X10, 2008. http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf.

[4] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of ibm bluegene/p. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008. IEEE Press.

[5] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. Nas parallel benchmark results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[6] Dan Bonachea, Paul Hargrove, Welcome M., and Katherine Yelick. Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt. In *CUG '09: Cray User Group Meeting*, 2009.

[7] Ron Brightwell, Rolf Riesen, and Arthur B. Maccabe. Design, implementation, and performance of mpi on portals 3.0. *The International Journal of High Performance Computing Applications*, 17(1), 2003.

[8] Wei-Yu Chen, Dan Bonachea, Costin Iancu, and Katherine Yelick. Automatic nonblocking communication for partitioned global address space programs. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 158–167, New York, NY, USA, 2007. ACM.

[9] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained upc applications. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. pages 29–40, Sept.-3 Oct. 2004.

[11] IBM BG/P Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2):199–220, January 2008.

[12] Ricky A. Kendall, Edoardo Aprà, David E. Bernholdt, Eric J. Bylaska, Michel Dupuis, George I. Fann, Robert J. Harrison, Jialin Ju, Jeffrey A. Nichols, Jarek Nieplocha, T. P. Straatsma, Theresa L. Windus, and Adrian T. Wong. High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Physics Communications*, 128(1-2):260–283, June 2000.

[13] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. Reducing connection memory requirements of mpi for infiniband clusters: A message coalescing approach. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 495–504, Washington, DC, USA, 2007. IEEE Computer Society.

[14] A. Shet, V. Tipparaju, and R. Harrison. Asynchronous programming in upc: A case study and potential for improvement. In *Workshop on Asynchrony in the PGAS Programming Model Collocated with ICS 2009*, Sept. 2009.

[15] G.M. Shipman, T.S. Woodall, R.L. Graham, A.B. Maccabe, and P.G. Bridges. Infiniband scalability in open mpi. pages 10 pp.–, April 2006.

[16] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K. Panda. Shared receive queue based scalable mpi design for infiniband clusters. In *IPDPS*, 2006.

[17] Vinod Tipparaju, Edoardo Apra, Weikuan Yu, and Jeffrey S. Vetter. Enabling a highly-scalable global address space model for petascale computing. In *Computing Frontiers '09*, 2010.

[18] Weikuan Yu, Xinyu Que, Vinod Tipparaju, R.L. Graham, and J.S. Vetter. Cooperative server clustering for a scalable gas model on the cray xt5. In *Proceedings of International Supercomputing Conference*, Hamburg, Germany, May 2010.