Comput Sci Res Dev (2010) 25: 57–64
DOI 10.1007/s00450-010-0104-6

SPECIAL ISSUE PAPER

Cooperative server clustering for a scalable GAS model on petascale cray XT5 systems

Weikuan Yu · Xinyu Que · Vinod Tipparaju · Richard L. Graham · Jeffrey S. Vetter

Published online: 8 April 2010

Abstract Global Address Space (GAS) programming models are attractive because they retain the easy-to-use addressing model that is the characteristic of shared-memory style load and store operations. The scalability of GAS models depends directly on the design and implementation of runtime libraries on the targeted platforms. In this paper, we examine the memory requirement of a popular GAS runtime library, Aggregate Remote Memory Copy Interface (ARMCI) on petascale Cray XT 5 systems. Then we describe a new technique *cooperative server clustering* that enhances the memory scalability of ARMCI communication servers. In cooperative server clustering, ARMCI servers are organized into clusters, and cooperatively process incoming communication requests among them. A request intervention scheme is also designed to expedite the return of responses to the initiating processes. Our experimental results demonstrate that, with very little impact on ARMCI communication latency and bandwidth, cooperative server clustering is able to significantly reduce the memory requirement of ARMCI communication servers, thereby enabling highly scalable scientific applications. In particular, it dramatically reduces the total execution time of a scientific application, NWChem, by 45% on 2400 processes.

Keywords PGAS · Cray XT5 · ARMCI

This work was funded in part by a UT-Battelle grant (UT-B-4000087151) to Auburn University, and in part by National Center for Computational Sciences. 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. This research was also supported by an allocation of advanced computing resources provided by the National Science Foundation. Part of the computations were performed on Kraken (a Cray XT5) at the National Institute for Computational Sciences (http://www.nics.tennessee.edu/).

W. Yu (✉) · X. Que
Department of Computer Science, Auburn University, Auburn, AL 36849, USA
e-mail: wkyu@auburn.edu

X. Que
e-mail: xque@auburn.edu

V. Tipparaju · R.L. Graham · J.S. Vetter
Computer Science & Mathematics, Oak Ridge National Laboratory, Oak Ridge, USA

V. Tipparaju
e-mail: tipparajuv@ornl.gov

R.L. Graham
e-mail: rlgraham@ornl.gov

J.S. Vetter
e-mail: vetter@ornl.gov

1 Introduction

GAS (Global Address Space) or PGAS (Partitioned Global Address Space) models support data access to local and remote memory through a simple shared memory styled access. Because of the attractiveness of its simple data accesses, PGAS languages like Unified Parallel C (UPC) [17], Co-Array Fortran (CAF) [5], and GAS libraries such as Global Arrays (GA) Toolkit [6] are becoming increasingly popular. Recently, a slightly different category of PGAS model, termed *Asynchronous* Partitioned Global Address space model, has emerged to add additional capabilities such as remote method invocations. IBM's X10 language [7] and Asynchronous Remote Methods (ARM) [12] in UPC have pioneered this new model.

All the above mentioned GAS languages and libraries use the services of an underlying communication library (which

we refer to as the GAS Runtime) for their communication needs. GAS languages normally use this runtime as a compilation target to do the data transfers on distributed memory architectures. They have a translation layer that translates a memory access to a corresponding data transfer on the underlying system. ARMCI (Aggregated Remote Memory Copy Interface) [10] is a popular GAS runtime that has been used to implement both PGAS languages (such as Co-Array Fortran) and GAS libraries (such as Global Arrays). It is highly scalable and has been ported to a variety of environments and platforms. Recently, a scalable implementation of ARMCI (described in [16]) was made available on the Jaguar Cray XT5 supercomputer at Oak Ridge National Laboratory. It enables a highly scalable Global Address Space model, Global Arrays, supporting mission critical applications such as NWChem [8]. However, overall memory usage and reliance on network flow control emerged as serious limitations of this solution.

The hallmark of ARMCI is its one-sided communication model and its simple progress rules that can be leveraged as compilation target by GAS models. To support this one-sidedness, ARMCI is designed with a communication server on each node to receive and respond to asynchronous requests from remote processes, without the involvement of the target process. The communication server has to pre-allocate one set of buffers to receive requests from every remote process (even if the remote access is for a mere handshake). The buffer requirement grows linearly with the total number of processes. For a parallel program with 64 thousand processes, it amounts to 1024 MB even if only one buffer of 16-KB is allocated per process. Thus, it creates an immense scalability challenge and limits the amount of available memory applications. Such a large memory footprint can also perturb the locality of data accesses as requests arrive in an arbitrary manner.

In this paper, we propose a technique called *cooperative server clustering* to cope with this scalability challenge. In cooperative server clustering, instead of allocating one set of buffers for all peer processes at each server, communication servers form cooperative clusters and then distribute the buffers for other peer processes among them. So long as a request reaches one server in a cooperative cluster, it will be forwarded to the target server, and receive appropriate handling and responses. Cooperative server clustering dramatically reduces the memory requirement of communication servers and increases the scalability of ARMCI. A request intervention scheme is also designed to work together with cooperative server clustering, and expedite the delivery of responses to initiating processes. An implementation of cooperative server clustering is accomplished on the Petascale Cray XT5 supercomputers, including Kraken at University of Tennessee Knoxville (UTK) and Jaguar at ORNL. Our experimental results demonstrate that, with little impact on the communication efficiency, cooperative clustering greatly improves the memory scalability of ARMCI. It also brings significant benefits to the performance of scientific applications. Our experiments show that, with cooperative server clustering, the total execution time of NWChem is reduced by 45% on 2400 processes.

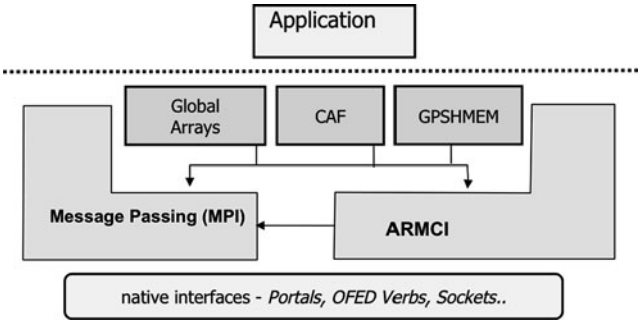


Fig. 1 Typical usage of ARMCI

The rest of the paper is organized as follows. Section 2 discusses the background and related work. Section 3 describes the design of cooperative server clusters in ARMCI. Several implementation issues are then presented in Sect. 4. Experimental results are provided in Sect. 5. Finally Sect. 6 concludes the paper.

2 Background and related work

2.1 ARMCI runtime system

ARMCI relies on a message-passing library and elements of the execution environment (job control, process creation, interaction with the resource manager). ARMCI, in addition to being the underlying communication interface for GA, has been used to implement other communication libraries and compilers [5, 11]. Typical structure of an application using ARMCI is shown in Fig. 1. ARMCI offers an extensive set of functionalities in the area of RMA communication: (1) data transfer operations (Get, Put Accumulate); (2) atomic operations; (3) memory management and synchronization operations; and (4) locks. Communication in most of the non-collective GA operations is implemented as one or more ARMCI communication operations.

ARMCI supports blocking and non-blocking versions of contiguous, strided and vector data transfer operations along with Read-Modify-Write operations. It uses the fastest available mechanism underneath to transmit data. On some platforms, native communication protocols are limited in their ability to provide all the functionalities ARMCI delivers to its users. An extra communication server is created on each node to service communication requests from its clients. ARMCI provides collective memory allocation interfaces

which allocate communicatable memory.[1] On the Cray XT5 system, ARMCI uses shared memory with-in a node and, uses Portals library for inter-node communication. The goal of this work is to optimize the communication server memory usage and its impact on the underlying network at very large scale.

2.2 Related work

The issue of scalability spans across a wide variety of aspects 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 is described in [2]. This has been a reference implementation for the communication design of other programming models on top of Portals. Communication protocols for different size messages were elaborated. Bonachea et al. [1] 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. [16] designed and implemented a scalable ARMCI communication library and demonstrated its strength in enabling GA and a real world scientific application, NWChem. Both of these efforts have laid out base line implementations of efficient GAS run-time systems using the native Portals communication library on the petascale Cray XT platforms. This work is a continued effort based on that of Tipparaju et al. to improve the communication model and resource management of ARMCI for better scalability.

Many other efforts studied the communication scalability for other programming models. Yu et al. [18] proposed an adaptive management model to improve the scalability of InfiniBand connections on large-scale platforms. Sur et al. [14] and Shipman et al. [13] studied the use of shared receive queue to increase the scalability of MPI communication resources. Sur et al. [15] examined the memory scalability of various MPI implementations on the InfiniBand network. Koop et al. [9] exploited the use of message coalescing to reduce the memory requirements for MPI on InfiniBand clusters. Chen et al. [3] optimized the communication for UPC applications through a combination of techniques including redundancy elimination, split-phase communication, and communication coalescing. Chen et al. [4] investigated the use of compiler techniques that can automatically schedule data transfers for non-blocking communication, thereby achieving better computation and communication overlap. Our work differ from these earlier studies by improving the scalability of ARMCI runtime system through a cooperative server clustering scheme that can reduce the memory requirements of ARMCI communication servers and alleviate the pressure from an enormous number of clients on individual servers. To our knowledge, we are not aware of any runtime that utilizes a scheme similar to cooperative server clustering.

[1] Memory is communicatable when all the necessary steps required by the communication library in order to send messages to this memory are performed at its allocation.

3 Design of cooperative server clustering

The main theme of this work is to increase the memory scalability for ARMCI communication servers. We first discuss the original process management and memory management in ARMCI communication servers. Then we describe the design of *cooperative server clustering* and a modified protocol *request intervention* that handles communication requests.

3.1 ARMCI process management for one-sided communication

As discussed earlier, GAS models provide a simple addressing model for data accesses in the form of load and store operations. These load and store operations are translated into compilation targets as one-sided communication calls by the GAS runtime communication system. ARMCI guarantees that its one-sided operations are fully unilateral (i.e., complete regardless of the actions taken by the remote process). In particular, polling the application by remote process (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 server (CS) at each compute node, along with other processes on the node.

Figure 2 shows the process management of ARMCI. On two arbitrary nodes, i and j, each has a set of processes with their global ranks. Processes on node i are also denoted as $P_{(i,k)}, \forall k \in [0, m-1]$. An area of shared memory is allocated for these m processes. The lowest ranked process $P_{(i,0)}$ creates a separate thread as a communication server CS_i. The communication server CS_i receives incoming remote one-sided communication requests and processes them

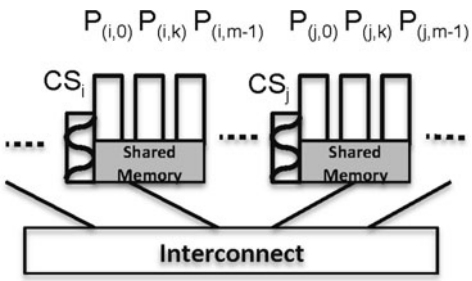


Fig. 2 ARMCI process management

on behalf of all local processes. To this purpose, it communicates with these local processes through the shared memory.

Every communication server has to pre-allocate request buffers for all remote peer processes. For a group of processes denoted as P_r, $\forall r \in [0, n-1]$, Fig. 3 shows the request buffer management of the communication server CS_i on Node i. It allocates a set of request buffers each remote process, e.g. B_r for P_r. The total buffer requirements would be roughly $N * M * B$, where N is the total number of processes (actually slightly smaller than N due to local processes), M the number of buffers for each process, and B the size of a buffer. 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.

3.2 Cooperative server clustering

To fundamentally address the scalability problem of buffer management in ARMCI servers, we design a technique called cooperative server clustering. Instead of allocating one set of buffers for all remote processes, communication servers from different nodes form cooperative clusters. Then the servers in a cluster divide peer processes amongst them.

Figure 4 shows the design of cooperative server clustering. A set of servers form a cluster C_i, which consists of $CS(C_i, k)$, $\forall k \in [0, g-1]$, where g is the number of servers in the cluster. The server $CS(C_i, k)$ will allocate request buffers for P_r, when $k = \lfloor r/g \rfloor$. For example, the

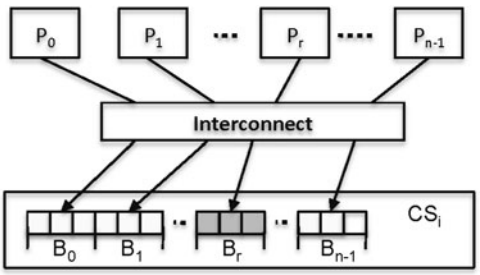


Fig. 3 ARMCI server's request buffer management

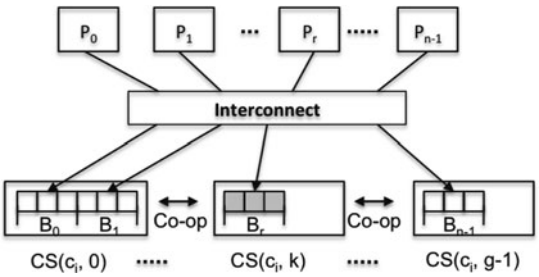


Fig. 4 Scalable ARMCI with cooperative server clustering

set of buffers B_r needed by P_r are allocated at $CS(C_i, k)$. The servers in a cluster cooperate (co-op) to handle requests from all processes. When a request reaches any server in this cluster of servers, it will be forwarded to the destined server along with all appropriate handling details and any potential response information.

Compared to the original buffer management at ARMCI servers, this new scheme requires only $N * B * M/g$ at an individual CS, where g is the number of communication servers in a cluster. N, B, and M are defined the same as discussed in Sect. 3.1. If a large-scale program can form clusters of 100 servers or more, the reduction in memory requirement can be several orders of magnitude.

3.3 Request intervention

In the original design of ARMCI, the communication server directly receives and handles one-sided communication requests from remote processes. For each request, a server returns a corresponding response (or acknowledgment) to an initiating process. This forms a direct request/response pair and a simplified flow control scheme between an initiating process and a target server. Figure 5(a) shows the direct pairing of request and response in an original ARMCI communication server. With cooperative server clustering, a technique referred to as *request intervention* is designed to process incoming requests. Figure 5(b) shows a flow diagram of request intervention in cooperative server clustering. A request from P_r will be intervened by an intermediate server CS_i and then forwarded it to the target server CS_t. Upon its arrival, CS_t communicates the response (or acknowledgment) directly to the requesting process P_r. CS_t also detects that the request is forwarded from CS_i, so it sends an acknowledgment to CS_i. The acknowledgment also serves as a means for flow control between CS_i and CS_t.

4 Implementation

We have designed and implemented a prototype of cooperative server clustering for ARMCI on the Cray XT5 systems including both Kraken at UTK and Jaguar at ORNL. Several implementation details are worth mentioning here. First, we allocate two separate pools of buffers in each communication server. One pool is reserved for requests from regular processes; the other is for the requests that are forwarded from intermediate communication servers. This separation avoids a race condition for the same resource from different requests. Second, we support request buffering at intermediate communication servers. This is needed when the response arrives at the requesting process sooner than the acknowledgment reaches the intermediate server. Third, we allocate a single portal memory descriptor for transmitting acknowledgments between servers. This avoids the need of additional memory descriptors for tracking forwarded requests

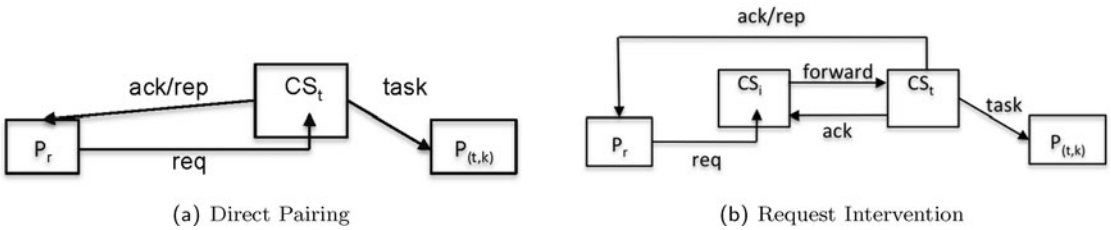


Fig. 5 One-sided request and response in cooperative server clustering

individually, and the associated communication processing, such as binding and unlinking the memory descriptors from the portals match list.

4.1 Preserving progress rules

Progress rules in ARMCI include checking for local completion and remote completion of data transfers. Our design of cooperative server clustering doesn't alter the logic or semantics of local completion. The support for remote completion in ARMCI is offered via a call to `ARMCI_Fence`. The implementation of remote completion involves a message to the communication server to verify the completion of all Put and Accumulate one-sided operations at the communication server. Our design of cooperative server clustering together with request intervention implicitly incorporates the logic to "forward" for remote completion requests as shown in Fig. 5(b). This ensures that the verification for remote completion initiated by any particular process P_r to a target t is intervened by a server CS_i and forwarded to the target server CS_t and the response to this verification sent directly to the initiator P_r.

5 Performance evaluation

Our experiments were conducted on Kraken and Jaguar supercomputers (both Cray XT5 systems). Both computers have dual hex-core Opteron processors, a total of 12 cores per node. Because of the similarity in the processor architecture and interconnection network, we will not distinguish between Kraken and Jaguar, but mention the number of processes in our experimental results.

5.1 Latency and bandwidth

Cooperative server clustering is designed to reduce the memory requirement of ARMCI communication servers. However, it is critical to minimize the performance impact to ARMCI communication operations. Particularly, ARMCI makes heavy use of put and get operations. So we have evaluated their latency and bandwidth to validate our design. These experiments were conducted across four nodes each with 12 processes. For cooperative clustering, servers are grouped into two clusters of two each. So there are two different communication patterns, one that directly occurs between a process and its targeted server, the other that requires one intermediate server to do request intervention, i.e. forwarding.

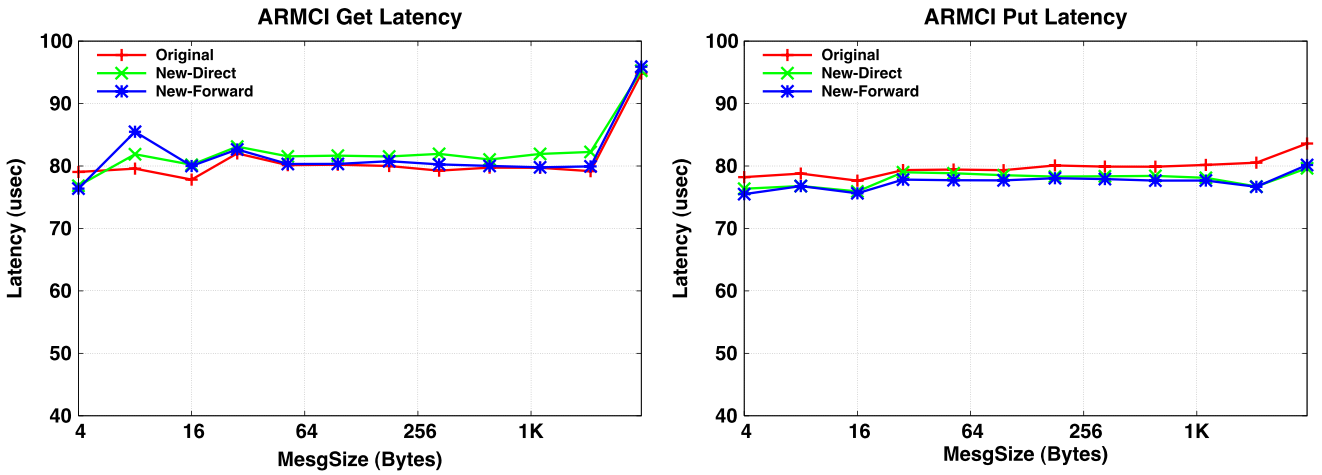
Figure 6 shows the latency comparisons between the "original" and "new" versions of ARMCI. For the new version, there are two variants, "direct" and "forward". As shown in the figure, all three have comparable latency for small messages. The original latency is slightly lower on get, marginally higher for put operations. For large messages, all three are on par with each other (data not shown due to the page limit). Figure 7 shows the bandwidth comparisons between the "original" and "new" versions of ARMCI. As shown in the figure, for small messages, the original version has slightly higher bandwidth compared to the new version, for both put and get operations. In terms of large messages, two versions are comparable to each other.

These results indicate that our design of cooperative server clustering has very little impact to the common communication operations of ARMCI. Given the significant improvement on memory scalability (as discussed below), we deem it as a worthy design, particularly when running ARMCI on a large number of processes.

5.2 Memory scalability

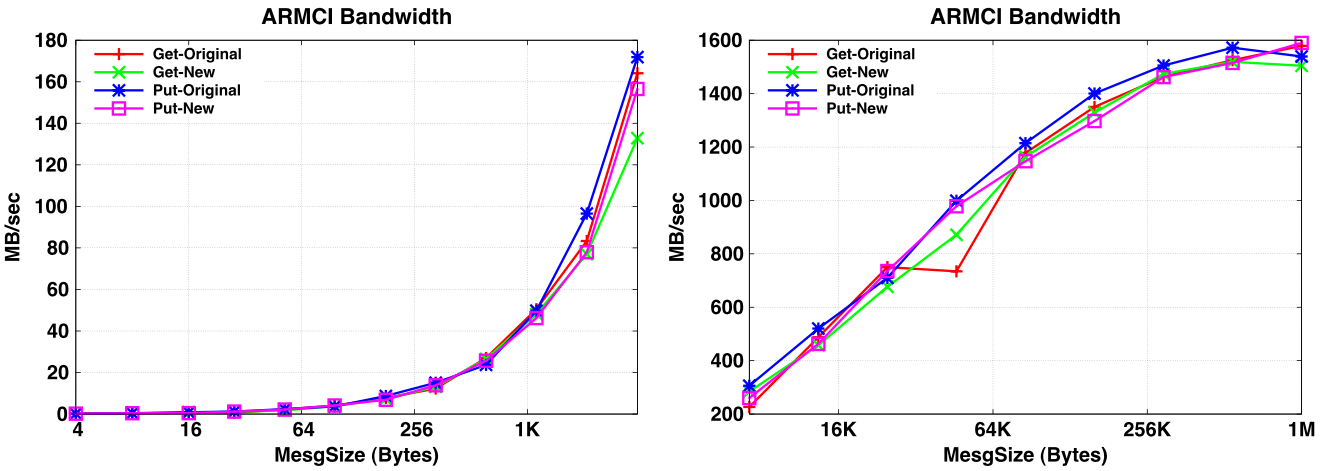
We measure the effectiveness of cooperative server clustering in reducing the memory footprint of ARMCI communication servers. The resident working set size (VmRss) in the /proc file system was taken as the memory footprint. All experiments were conducted with 12 processes per node. The servers are divided into clusters, with a cluster size close to the square root of the total number of servers. That is to say, for 1200 processes, there will be 100 ARMCI servers, which are divided into 10 clusters of 10 servers each.

Figure 8 shows the comparison of memory scalability between the "original" and "new" versions of ARMCI. With multiple processes on a single node, each ARMCI node is also allocated with a big pool of shared memory region for intranode communication. In addition, the communication server needs to allocate more request buffers for an increasing number of processes. As shown in the figure, the



(a) Get (b) Put

Fig. 6 Comparisons of ARMCI latency



(a) Small Messages (b) Large Messages

Fig. 7 Comparisons of ARMCI bandwidth

original version has a linear memory growth trend, strictly in the order of $O(N)$. The new version with cooperative server clustering reduces the memory requirement significantly. For 9600 processes on 800 nodes, the new version requires only 34.7 MB of more memory in ARMCI communication servers, while the original version needs 169.7 MB more memory. This is a factor of 4.8 improvement. With larger number of processes, the benefit is expected to be more significant.

5.3 Performance benefits to a scientific application

Density Functional Theory (DFT) is a widely used electronic structure method in NWChem. It is the workhorse of electronic structure for its balance between computational cost and accuracy (1998 Nobel prize in Chemistry). We used the DFT SiOSi3 benchmark to measure the performance benefits of cooperative server clustering. Figure 9 shows the performance of DFT SiOSi3 with a varying number of processes. As shown in the figure, ARMCI with cooperative server clustering significantly benefits the execution time of NWChem, by 45% with 2400 processes. This result demonstrates that the improvement in memory footprint benefits scientific applications such as NWChem, despite the slight degradation on microbenchmarks such as latency and bandwidth.

6 Conclusions

We have designed a scalable memory management scheme, cooperative server clustering, for ARMCI communication servers and have shown its effectiveness in enabling the

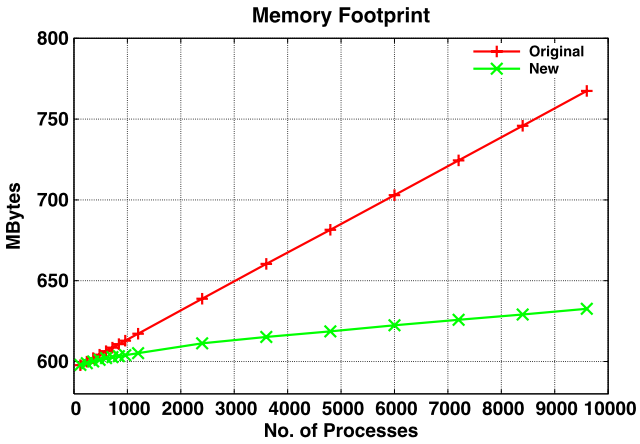


Fig. 8 Memory scalability

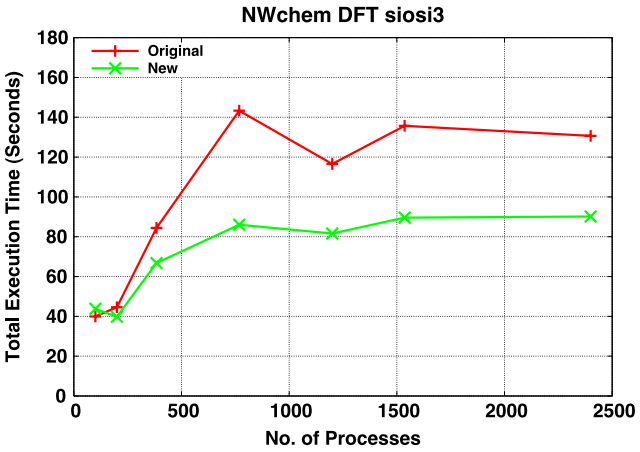


Fig. 9 Benefits to DFT siosi3 execution time

Global Arrays (GA) GAS programming model [6]. By constructing communication servers into cooperative clusters, and dividing incoming requests among them, cooperative server clustering can significantly reduce the memory requirement of large-scale GAS programs, thereby improving the scalability of GA on petascale machines such as Jaguar. A request intervention scheme is also designed to expedite the delivery of responses to requesting processes. In realizing cooperative server clustering and request intervention, we have also dealt with various implementation issues including race conditions and preserving progress rules.

Our experimental results have demonstrated that the memory requirement at ARMCI communication servers has been significantly reduced. With little impact on communication bandwidth. our optimization can significantly improve scientific applications. Particularly, we have demonstrated that ARMCI with cooperative sever clustering can reduce the total execution time of a scientific application NWChem by 45% on 2400 processes.

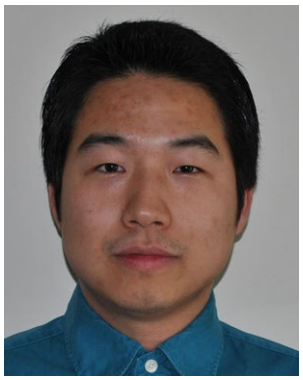
In future, we look forward to further optimization of the GA model and ARMCI on petascale systems. We also plan to study the applicability of cooperative server clustering to GA and ARMCI on other petascale platforms such as BlueGene /P.

References

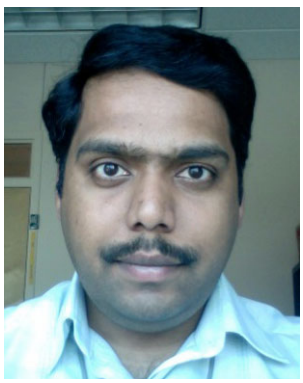
1. Bonachea D, Hargrove PMW, Yelick K (2009) Porting gasnet to portals: partitioned global address space (pgas) language support for the cray xt. In: CUG '09: cray user group meeting
2. Brightwell R, Riesen R, Maccabe AB (2003) Design, implementation, and performance of mpi on portals 3.0. Int J High Perform Comput Appl 17(1)
3. Chen WY, Iancu C, Yelick K (2005) Communication optimizations for fine-grained upc applications. In: PACT '05: proceedings of the 14th international conference on parallel architectures and compilation techniques. IEEE Computer Society, Washington, pp 267–278
4. Chen WY, Bonachea D, Iancu C, Yelick K (2007) Automatic nonblocking communication for partitioned global address space programs. In: ICS '07: proceedings of the 21st annual international conference on supercomputing. ACM, New York, pp 158–167
5. Dotsenko Y, Coarfa C, Mellor-Crummey J (2004) A multiplatform co-array Fortran compiler. In: Proceedings of parallel architecture and compilation techniques
6. Global Arrays Toolkit (2009) http://www.emsl.pnl.gov/docs/global
7. IBM (2008) Report on experimental language X10. http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf
8. Kendall RA, Aprà E, Bernholdt DE, Bylaska EJ, Dupuis M, Fann GI, Harrison RJ, Ju J, Nichols JA, Nieplocha J, Straatsma TP, Windus TL, Wong AT (2000) High performance computational chemistry: an overview of NWChem a distributed parallel application. Comput Phys Commun 128(1):260–283 -2
9. Koop MJ, Jones T, Panda DK (2007) 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, Washington, DC, USA
10. Nieplocha J, Tipparaju V, Krishnan M, Panda DK (2006) High performance remote memory access communication: the armci approach. Int J High Perform Comput Appl 20(2):233–253
11. Parzyszek K (2003) Generalized portable shmem library for high performance computing. PhD thesis, Ames, IA, USA, co-Major professor-Kendall, Ricky A and Co-Major professor-Lutz, Robyn R
12. Shet A, Tipparaju V, Harrison R (2009) Asynchronous programming in upc: a case study and potential for improvement. In: Workshop on asynchrony in the PGAS programming model collocated with ICS 2009
13. Shipman G, Woodall T, Graham R, Maccabe A, Bridges P (2006) Infiniband scalability in open mpi. In: International parallel and distributed processing symposium
14. Sur S, Chai L, Jin HW, Panda DK (2006) Shared receive queue based scalable mpi design for infiniband clusters. In: International parallel and distributed processing symposium
15. Sur S, Koop MJ, Panda DK (2006) High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In: SC '06: proceedings of the 2006 ACM/IEEE conference on supercomputing. ACM, New York, p 105
16. Tipparaju V, Apra E, Yu W, Vetter JS (2010) Enabling a highly-scalable global address space model for petascale computing. In: Computing frontiers

17. UPC Specifications, v12 (2009) http://www.gwu.edu/~upc/publications/LBNL-59208.pdf
18. Yu W, Gao Q, Panda D (2006) Adaptive connection management for scalable mpi over infiniband. In: International parallel and distributed processing symposium, Greece

Weikuan Yu is currently an Assistant Professor in the Department of Computer Science and Software Engineering at Auburn University. Prior to joining Auburn, he served as a Research Scientist for two and a half years at Oak Ridge National Laboratory (ORNL) until January 2009. Yu is also a Joint Faculty at ORNL. He earned his PhD in Computer Science from the Ohio State University in 2006. Yu also holds a master's degree in Developmental Biology from the Ohio State University and a Bachelor degree in Genetics from Wuhan University, China. At Auburn University, Yu leads the Parallel Architecture and System Laboratory (PASL) for research and development on high-end computing, parallel and distributing networking, storage and file systems, as well as interdisciplinary topics on computational biology. Yu is a member of AAAS, ACM, and IEEE.



Xinyu Que is a Ph.D. student of Parallel Architecture and System Laboratory (PASL) in the Department of Computer Science at Auburn University. Que earned his master's degree in Computer Science from University of Connecticut in 2009. His research interests include High Performance Computing, High Speed Networking, Network and Grid Computing.



Vinod Tipparaju is a Research Staff Member in the Computer Science and Mathematics Division (CSM) of Oak Ridge National Laboratory (ORNL), where he is a member in the Future Technologies Group and a matrix-ed member in the NCCS Scientific Computing and Technology Integration Groups. He is one of the main developers of Global Arrays toolkit and the ARMCI communication library. He joined ORNL in 2008, after over six years at Pacific Northwest National Laboratory. Tipparaju's interests span several areas of high-end computing including Programming Models for High Performance Computing, Network Interconnects and Collective Communication Algorithms.

Richard L. Graham is a Distinguished Research Staff Member, and Group Leader of the Application Performance Tools group, in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL). He is currently chairing the MPI Forum. Before joining ORNL he was the Advanced Computing Laboratory Acting Group Leader at the Los Alamos National Laboratory (LANL). He joined LANL's Advanced Computing Laboratory (ACL) as a technical staff member in 1999 and as Team Leader for the Resilient Technologies Team, started the LA-MPI project and co-founded the Open MPI project. Prior to joining ACL, he spent seven years working for Cray Research and SGI. Graham earned his PhD in Theoretical Chemistry from Texas A&M University in 1990 and received post-doctoral training at the James Franck Institute of University of Chicago. He has a BS in chemistry from Seattle Pacific University.

Jeffrey S. Vetter is a computer scientist in the Computer Science and Mathematics Division (CSM) of Oak Ridge National Laboratory (ORNL), where he leads the Future Technologies Group and directs the Experimental Computing Laboratory. Dr. Vetter is also a Joint Professor in the College of Computing at the Georgia Institute of Technology, where he earlier earned his PhD. He joined ORNL in 2003, after four years at Lawrence Livermore National Laboratory. Vetter's interests span several areas of high-end computing—encompassing architectures, system software, and tools for performance and correctness analysis of applications.