



HiCOO: Hierarchical cooperation for scalable communication in Global Address Space programming models on Cray XT systems

Weikuan Yu^{a,*}, Xinyu Que^a, Vinod Tipparaju^b, Jeffrey S. Vetter^b

^a Department of Computer Science, Auburn University, AL 36849, USA

^b Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

ARTICLE INFO

Article history:

Received 18 July 2011

Received in revised form

3 January 2012

Accepted 30 January 2012

Available online 6 February 2012

Keywords:

GAS

ARMCI

Multicore

Multinode

Virtual Topology

Contention

ABSTRACT

Global Address Space (GAS) programming models enable a convenient, shared-memory style addressing model. Typically this is realized through one-sided operations that can enable asynchronous communication and data movement. With the size of petascale systems reaching 10,000s of nodes and 100,000s of cores, the underlying runtime systems face critical challenges in (1) scalably managing resources (such as memory for communication buffers), and (2) gracefully handling unpredictable communication patterns and any associated contention. For any solution that addresses these resource scalability challenges, equally important is the need to maintain the performance of GAS programming models. In this paper, we describe a Hierarchical COoperation (HiCOO) architecture for scalable communication in GAS programming models. HiCOO formulates a cooperative communication architecture: with inter-node cooperation amongst multiple nodes (a.k.a. multinode) and hierarchical cooperation among multinodes that are arranged in various virtual topologies. We have implemented HiCOO for a popular GAS runtime library, Aggregate Remote Memory Copy Interface (ARMCI). By extensively evaluating different virtual topologies in HiCOO in terms of their impact to memory scalability, network contention, and application performance, we identify MFCG as the most suitable virtual topology. The resulting HiCOO architecture is able to realize scalable resource management and achieve resilience to network contention, while at the same time maintaining or enhancing the performance of scientific applications. In one case, it reduces the total execution time of an NWChem application by 52%.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Several supercomputing sites have deployed systems with extreme amounts of computational power [30]. For example, the Jaguar Cray XT5 system in the US, the Tianhe-1A system in China, and the K Computer in Japan can perform to the order of 10^{15} floating point operations per second (petaflop). While supercomputing systems grow to unprecedented number of processors (with LLNL Sequoia [21] system and NCSA Blue Waters [22] system in the near future), scientific applications continue to face many challenges such as programming productivity, application scalability, and efficiency. Global Address Space (GAS) or Partitioned Global Address Space (PGAS) models are emerging as scalable alternatives because they have the ability to alleviate programming burden by supporting data access to both local and remote memory through a simple shared-memory addressing model.

PGAS languages such as Unified Parallel C (UPC) [31], Co-Array Fortran (CAF) [10], and X10 [26], and GAS libraries such as Global Arrays (GA) Toolkit [14] are becoming increasingly popular. These languages and libraries use the services of an underlying communication library (which we refer to as the GAS runtime) for serving their communication needs. They normally convert 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.

ARMCI (Aggregated Remote Memory Copy Interface) [23] is a popular runtime that has been used to implement both PGAS languages (such as Co-Array Fortran) and GAS libraries (such as GA). While some MPI applications have reached a sustained petaflop performance and beyond, NWChem [17] computation chemistry code is a GAS-based application and is one of the three applications to have crossed the petaflop barrier in terms of sustained performance [2] on Jaguar. This was made possible by the porting of Global Arrays toolkit, and more specifically, its GAS runtime, ARMCI [29].

* Corresponding author.

E-mail addresses: wkyu@auburn.edu, weikuan.yu@gmail.com (W. Yu), xque@auburn.edu (X. Que), tipparaju@gmail.com (V. Tipparaju), vetter@ornl.gov (J.S. Vetter).

Unfortunately, running a GAS model and its underlying GAS runtime in the context of a real scientific application at a scale similar to Jaguar (200,000 + cores) has brought forth a few staggering challenges. These challenges are a result of the characteristics and asynchronous one-sided features of the GAS runtime. The first is that of resource management, incurred by unpredictable communication patterns and communication resources (such as buffers) that need to be allocated to support it. The second challenge is that of network contention—allowing any process to access the address space of any other process and supporting load balancing at the same time create an environment that is prone to contention.

In this paper, we describe a Hierarchical COoperation (HiCOO) architecture for scalable GAS programming models. HiCOO formulates a cooperative communication architecture with inter-node cooperation amongst multiple nodes (a.k.a multinode) and hierarchical cooperation among multinodes that are arranged in various virtual topologies. We have implemented HiCOO for a popular GAS runtime library, Aggregate Remote Memory Copy Interface (ARMCI). It leverages the existing multicore cooperation in ARMCI and extends with *multinode cooperation* and *hierarchical cooperation*.

In multinode cooperation, compute nodes form a multinode group and work together to handle one-sided communication requests. Their cooperation is realized through (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, HiCOO no longer has to create one set of communication buffers on every node for all possible pairs of peer processes. Instead, it divides the requirement of communication buffers amongst themselves in a cooperative manner. When a request reaches one node in a multinode group, it is 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.

With hierarchical cooperation arranged in various virtual topologies, HiCOO attenuates contention and efficiently manages communication resources in ARMCI (and any GAS/PGAS runtime in general) at petascale and beyond. HiCOO represents the allocation of communication resources as directed graphs. While the original model can be depicted as a fully connected graph (FCG), two new scalable virtual topologies, Meshed FCGs (MFCG) and Cubic FCGs (CFCG), have been exploited for scalable resource management and contention attenuation in HiCOO. We have systematically examined the communication characteristics of MFCG, CFCG, and Hypercube. We have successfully implemented these virtual topologies in ARMCI on Jaguar, and conducted experiments to evaluate these topologies using microbenchmarks and real large-scale applications. We then choose MFCG as the default topology for HiCOO.

While addressing the challenges of resource scalability and network contention, equally important is the need to maintain the performance of GAS programming models. Our experimental results on a large-scale Cray XT5 system indicate that HiCOO is able to greatly increase memory scalability by reducing communication buffers required on each node. In addition, it improves the resiliency of GAS runtime system to network contention. Furthermore, HiCOO is able to maintain or 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 2 discusses background and motivation. Section 3 describes the architecture of HiCOO and its two key components: multinode cooperation and virtual topology. Experimental results are provided in Sections 4 and 5, followed by related work in Section 6. We conclude the paper in Section 7.

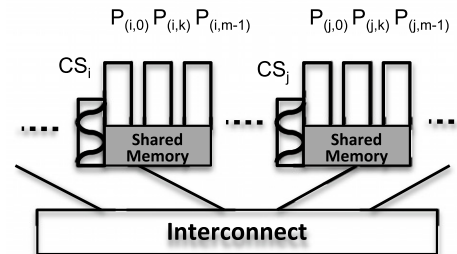


Fig. 1. ARMCI process management.

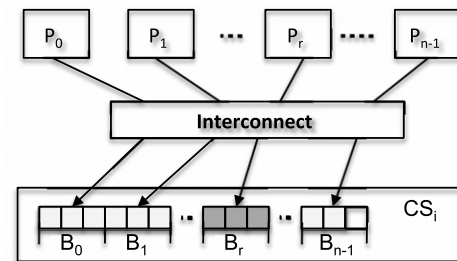


Fig. 2. ARMCI server's request buffer management.

2. Background and motivation

2.1. An overview of ARMCI

ARMCI has recently been enabled for Cray XT5 using the native portals communication library [29]. 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 (CS) 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.

2.2. ARMCI process management for one-sided communication

Fig. 1 shows the process management of ARMCI. On two arbitrary nodes, i and j , each has a set of parallel processes. All processes have a global rank. 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 communicates with all intra-node processes through the shared memory and handles all incoming inter-node one-sided communication requests on behalf of them.

Every communication server has to pre-allocate request buffers for all remote peer processes. Fig. 2 shows the request buffer management of CS_i . Each process is denoted based on its global rank P_r , $\forall r \in [0, n - 1]$. A set of request buffers are allocated for each remote process, e.g. B_r , for P_r .

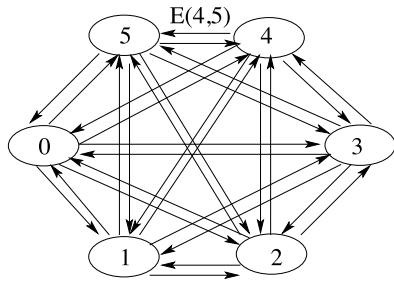


Fig. 3. A directed graph representing resource allocation for one-sided requests.

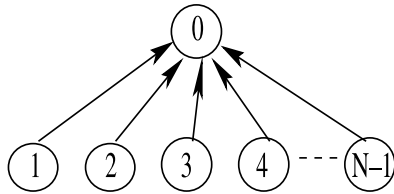


Fig. 4. A flat-tree representation of contention among communication requests.

2.3. Critical challenges for GAS runtime

To better formulate the memory resource management of ARMCI, we define *virtual topology* as a means to represent the graph of resource allocation. In the case of memory buffers for communication, a directed graph can represent the resource allocation of buffers amongst all nodes. A graph $G : (V, E)$ consists of a set of vertices V and a set of edges E . A vertex i represents all processes and the CS on a single node i . A directed edge $E(i, j)$ from i to j denotes the fact that there is a set of request buffers allocated on node i for tasks on node j . For an ARMCI application running on N nodes, this representation of buffer allocation forms a FCG with $N * (N - 1)$ directed edges. There are $(N - 1)$ outgoing edges at each vertex (node), representing $N - 1$ sets of buffers from $N - 1$ remote nodes. Fig. 3 shows the resource allocation graph for a 6-node case.

The directed graph representation of resource allocation in Fig. 3 reveals two critical challenges that a GAS model (in our case, Global Arrays) poses to its underlying GAS runtime (in our case, ARMCI).

Resource Management—The first challenge is on the allocation of resources for communication. Consider an example of the targeted systems for this work, the Cray XT5. Cray XT5 has Seastar2 + interconnect and uses the connection-less Portals messaging library as the lowest level communication protocol. To support the connection-less Portals interface, the Cray Seastar2 + allows for 256 simultaneous message streams. When additional streams need to be initiated (or in case of resource exhaustion), the Cray BEER (Basic End to End Reliability) protocol does the necessary flow control and handles reliability. This means that the resource allocation problem for ARMCI communication buffers (where a set of buffers needs to be allocated for every incoming edge as shown in Fig. 3) maps to parallel message streams in Portals but at a different scale. The total request buffer requirement in ARMCI for the FCG would be roughly $N * B * M$, where N is the total number of processes (actually slightly smaller than N due to local processes), B the buffer size, and M the set of buffers per process. With only two 16 kB buffers per process, it would require 1024 MB per CS to support parallel programs with 32,000 processes, and 32 GB per CS on an future system with a million processes.

Contention—Another challenge revealed by the FCG model is the potential contention that could be caused by many concurrent requests to a single node. Because all nodes (vertices) are directly connected, the paths for requests from all nodes to traverse a virtual FCG and reach one node can be represented as a flat

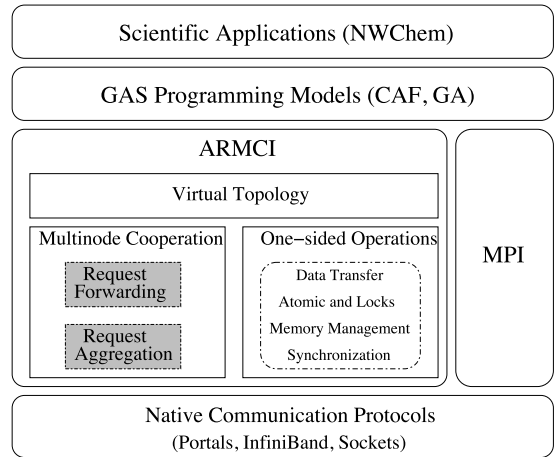


Fig. 5. Software architecture of hierarchical cooperation.

tree of depth 1. Fig. 4 shows a tree representation of request traversal paths to Node 0. Such a flat tree is very vulnerable to transient hot-spot access scenarios, such as when thousands of processes simultaneously accessing one data element in an address space. These scenarios create a severe hot-spot contention problem in addition to the resource allocation problem described above. Under such scenarios, significant burden is placed on the physical network, which will be forced to adopt some throttling mechanisms, typically causing serious slowdown of the entire communication and jeopardizing the system productivity.

3. Hierarchical COOperation (HiCOO)

Fig. 5 shows the software architecture of Hierarchical COOperation (HiCOO). On a system that supports GAS-enabled scientific applications such as NWChem, ARMCI will support the required one-sided operations, including data transfer, atomic and locks, memory management, and synchronization. HiCOO extends ARMCI with an indirect communication model for transmitting one-sided requests in these operations. It includes two key components: multinode cooperation and virtual topology. These two components are mutually dependent on each other for their functionalities. Multinode cooperation offers the fundamental communication mechanisms for different nodes and their communication servers to cooperate with each other for request handling. Virtual topology offers a formal model that defines the geometric relationship among all the nodes, and accordingly their distance in the topology hierarchy.

3.1. Multinode cooperation

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. It is supported through two communication mechanisms: request forwarding and request aggregation. We then focus on describing these two mechanisms in more detail.

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

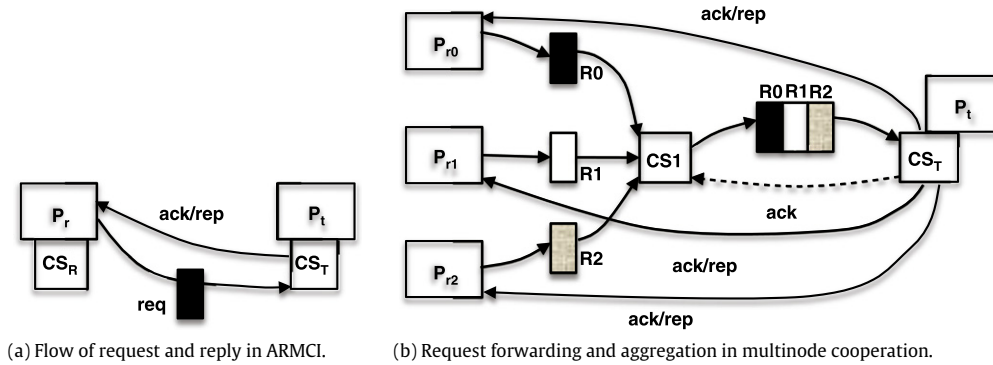


Fig. 6. Request handling in ARMCI and multinode cooperation.

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.

The original ARMCI has a very simple communication model to support direct one-sided operations. Fig. 6(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. Fig. 6(b) shows the flow of requests and replies in multinode cooperation. Three processes (P_{r0} , P_{r1} , and P_{r2}) are 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.

3.2. Virtual topology

As discussed in Section 2, the default resource allocation in ARMCI leads to a serious scalability challenge. More importantly,

its resource dependence relationship (irrespective of any underlying physical network topology) can cause contention when some processes become hot-spots to the communication requests. A virtual topology FCG can precisely reflect the state of resource allocation and contention. It also suggests that alternative virtual topologies may offer a solution for scalable resource management and contention attenuation. We first introduce two new virtual topologies: MFCG and CFCG, and examine various features of these two, along with a canonical topology Hypercube. Then we describe the details of request routing in realizing these topologies.

3.2.1. Comparisons of three virtual topologies

MFCG—The first virtual topology we have introduced is called Meshed Fully Connected Graphs (MFCG for short). Fig. 7(a) shows an example of MFCG, in which all nodes are virtualized as vertices in a $X \times Y$ mesh (in this case, $X = 3$ and $Y = 3$). Nodes with the same Y -offset are fully connected. That is to say, they all dedicate request buffers to each other. The same policy is applied to nodes with the same X -offset. Thus, for an arbitrary $X \times Y$ MFCG, an individual node has $(X - 1)$ outgoing edges on X -dimension and $(Y - 1)$ outgoing edges on Y -dimension. A request routing mechanism is provided to exchange requests between a pair of nodes that are not directly connected. Therefore, using MFCG, the number of request buffers on each node decreases to $O(\sqrt{N})$, instead of $O(N)$ in FCG.

MFCG is also beneficial in alleviating contention. Fig. 8(a) shows request paths for nodes in a 3×3 MFCG to reach Node 0. Two types of request paths are possible: the first type is used by the nodes that are directly connected to Node 0; and the second type is used by the nodes that are not directly connected. These paths form a tree of height 2 and rooted at Node 0. Compared to the flat tree as shown in Fig. 4, the contention is reduced to $O(\sqrt{N})$. One may rightfully argue that contention as depicted in Fig. 8(a) does not reflect the actual contention in the physical network. The purpose of scalable virtual topology is to offer a convenient tool that can cope with network contention at a software level, instead of leaving the contention issues completely to the network hardware.

CFCG—Another virtual topology we introduced is Cubic Fully Connected Graphs (CFCG). Fig. 7(b) shows an example of CFCG, in which all nodes are virtualized as vertices in a $X \times Y \times Z$ cube (in this case, $X = 3$, $Y = 3$, and $Z = 3$). The nodes with the same offsets on two dimensions are fully connected as an FCG. For an arbitrary $X \times Y \times Z$ CFCG, an individual node have $(X - 1)$ outgoing edges on X -dimension, $(Y - 1)$ outgoing edges on Y -dimension, and $(Z - 1)$ outgoing edges on Z -dimension (to clarify, not all vertices/edges are shown for CFCG). Using CFCG, the number of request buffers on one node scales in the order of $O(\sqrt[3]{N})$, instead of $O(N)$ with FCG. A request may have to be forwarded maximally two times before reaching its destination.

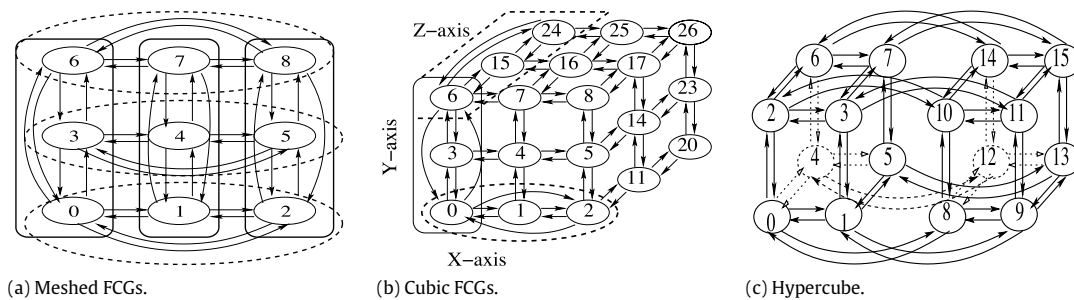


Fig. 7. Three virtual topologies. (For clarity, not all vertices/edges are shown in CFCCG.)

Fig. 8(b) shows the tree representation of request paths for nodes in a $3 \times 3 \times 3$ CFCCG to reach Node 0. These directed paths form a trinomial tree of height 3 and rooted at Node 0. For a system with N nodes, the tree of request paths rooted at an arbitrary node will be k -nomial tree where $k = \sqrt[3]{N}$. Compared to the flat tree in Fig. 4, network contention is then reduced by an order of $O(\sqrt[3]{N})$, at the expense of up to 2 forwarding steps to deliver a request.

Hypercube—As discussed above, CFCCG is more scalable in resource allocation than MFCG and FCG, despite more steps for request transmission. One may wonder if a virtual topology of even higher dimension could be a worthy solution. So we investigate the third virtual topology, Hypercube. Fig. 7(c) shows 16 nodes that are connected as a Hypercube. Each node is directly connected to $\log_2 N$ nodes (4 in this case). Fig. 8(c) provides a tree representation of request paths from all nodes to Node 0. For N nodes, it is essentially a binomial tree of depth $\log_2 N$. Using Hypercube, the number of request buffers required on one node scales in the order of $O(\log_2 N)$. Two nodes may be separated by up to $\log_2 N$ dimensions apart. Therefore, up to $(\log_2 N - 1)$ transmissions are needed for a request to reach its destination. On the other hand, at each depth of a request path tree, contention is reduced by an order of $O(\log_2 N)$.

3.2.2. Request routing in virtual topologies

We have implemented MFCG, CFCCG, and Hypercube in ARMCI on Jaguar. The support for request routing is the key to realizing these virtual topologies. Communication servers on intermediate nodes are used to transmit a request from the original process to the target server. Upon the arrival of a request, the target sends a response (or acknowledgment) directly to the original process. If an intermediate server (or the target) detects that the request is routed from an upstream server, it sends an acknowledgment to the upstream server. To support multidimensional topologies such as MFCG, CFCCG, and Hypercube, our implementation also allows a request to be transmitted multiple steps.

For correct request routing, the actual implementation of virtual topologies requires proper handling of two important issues: (a) how to determine the order of routing; and (b) how to enable virtual topologies, MFCG and CFCCG, when the number of nodes can only be configured as partially-populated topologies (mesh or cube), e.g., a prime number that cannot be evenly divided. As mentioned earlier, we include Hypercube only to examine its tradeoff in resource management and contention, compared to MFCG and CFCCG. For the investigative purpose, we only support Hypercube when the number of nodes is a power of 2.

Lowest-Dimension-First Routing—Multiple communication steps are needed for an ARMCI request to properly reach its destination, in multi-dimensional virtual topologies such as MFCG, CFCCG and Hypercube. Each step corresponds to a relationship in which an upstream node is dependent on the availability of request buffer at the downstream node. If the routing of requests were to happen arbitrarily, it would create cyclic dependences and lead to deadlocks in a multi-dimensional virtual topology.

Algorithm 1 Lowest Dimension First Routing

```

1: {Dimension:  $k$ }
2: {Current Node:  $S = (s_0, s_1, \dots, s_{k-1})$ }
3: {Destination Node:  $T = (t_0, t_1, \dots, t_{k-1})$ }
4:  $D \leftarrow S$  {Initialize  $D$  as the next node}
5:  $i \leftarrow 0$ 
6: while ( $D \neq T$ ) do
7:   if  $s_i \neq t_i$  then
8:      $D \leftarrow (s_0, s_1, \dots, s_{i-1}, t_i, s_{i+1}, \dots, s_{k-1})$ 
9:     {Forward the request to the next node,  $D$ }
10:  end if
11:   $i \leftarrow i + 1$ 
12: end while

```

We develop a lowest-dimension-first (LDF) protocol to ensure deadlock-free routing in virtual topologies. Algorithm 1 illustrates the selection of next node for request routing in LDF. For two nodes $S = (s_0, s_1, \dots, s_{k-1})$ and $T = (t_0, t_1, \dots, t_{k-1})$ on a virtual topology with k dimensions, LDF always chooses the lowest dimension i on which S and T differ. A request is then forwarded to the next destination D , which is a number derived by replacing s_i of S with t_i . Since the order of routing is established in an monotonic dimension order, breaking any cyclic dependence. Therefore LDF is deadlock-free. When the number of nodes allows virtual topologies to be fully populated as meshes, cubes, or Hypercubes, LDF as shown in Algorithm 1 works perfectly.

Routing on Virtual Topologies with Any Number of Nodes—Routing in a virtual topology is similar to routing in a physical interconnect. In the case of a fully populated two-dimensional MFCG, LDF can be reduced to the classic turn model [13] that was designed for 2-D meshes. However, the key difference is that a virtual topology is very dynamic and frequently partially populated. For this reason, each node frequently changes its position from one topology to another. It is important that deadlock-free routing be enabled on virtual topologies (MFCG and CFCCG) with any number of nodes.

We achieve that by strictly ordering all nodes in a lowest dimension first manner. For a virtual topology G with dimension k , the lower order dimensions are first populated with available nodes. Only the highest dimension, $k - 1$, is allowed to be partially populated. Assume that a virtual topology G has M as its highest ranked node, where $M = (M_0, M_1, \dots, M_{k-1})$. With all nodes ordered this way, we extend the LDF algorithm slightly. It allows routing only when the next destination D is a number smaller than or equal to M . An extra condition, “if ($D \leq M$)”, is introduced to Algorithm 1 before a request is forwarded. With this extension, if routing paths of a set of requests did not violate this extra condition, there would not be a deadlock because their routing paths are determined by Algorithm 1. For a possible deadlock to occur, one request must have violated this condition once in its path. This is not possible because the nodes are strictly ordered and no node can have a rank higher than M (by definition). Therefore, it prevents any circle in request routing. The listing of the extended LDF algorithm is not included here, due to the simplicity of this addition.

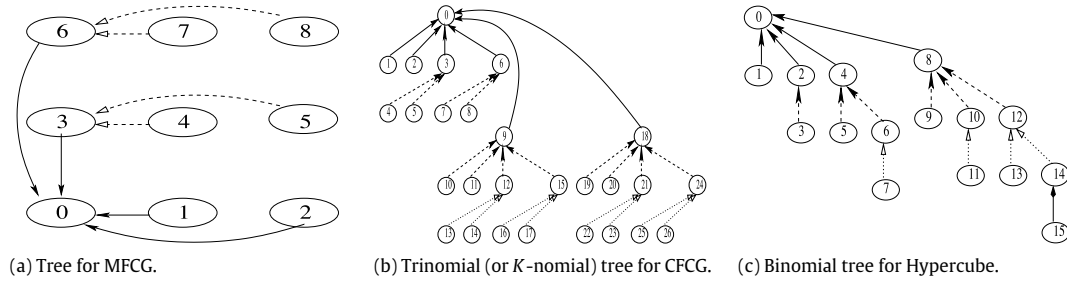


Fig. 8. Tree representations of request paths in virtual topologies.

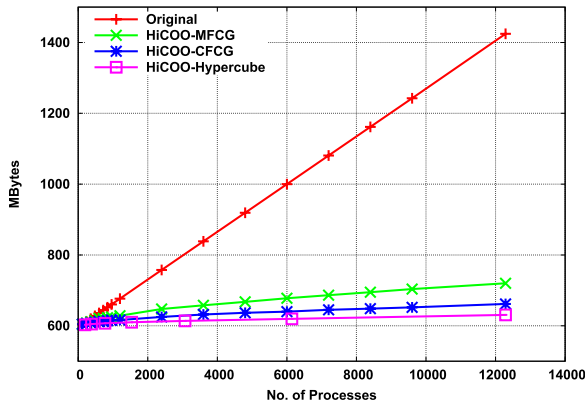


Fig. 9. Scalability virtual topologies for memory management.

4. Analysis of memory management and contention attenuation

In this section, we describe our experiments that evaluate the impact of different virtual topologies on memory management and contention attenuation. Performance results from these topologies are compared to the original ARMCI that uses the FCG pattern for request buffer allocation.

4.1. Scalable memory management

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 an ARMCI program that reports VmRSS from all processes. This number represents the total memory consumed by an ARMCI process at runtime before any additional application-level memory consumption.

We measure the impact of virtual topologies on memory resources. Our experiments are conducted with 12 processes per node. All processes start with a memory consumption of about 612 MB. However, due to the allocation of request buffers by the internal CS, a master process requires more memory for an increasing number of remote processes. The size of each buffer in CS is 16 kB; and the number of buffers per process is 4. Fig. 9 shows the memory consumption of master processes, using different virtual topologies. As expected, the memory requirement of the original ARMCI increases linearly. On 12,288 processes, the original has a memory consumption of 1424 MB, an increment of 812 MB, on top of 612 MB that is needed to run a few processes. The other three virtual topologies provide much better scalability in terms of memory resources. Compared to the original, HiCOO-MFCG, HiCOO-CFCG, and HiCOO-Hypercube cut down the increment in memory consumption significantly, by 7.5, 16.6, and 45 times, respectively.

4.2. Contention attenuation

Virtual topologies are also designed to address the other critical challenge, hot-spot contention in the GAS runtime. We evaluate contention for all one-sided ARMCI operations, and observe that virtual topologies are beneficial to the contention caused by lock, accumulate, noncontiguous data transfer, and atomic operations. Herein presented are results for two representative operations, noncontiguous vector data transfer and atomic Fetch-&Add operations.

4.2.1. Description of contention experiments

We define hot-spot contention as the percentage of processes in a program that are contending for communication to a single process, or access to a single data element. It is understood that such 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 hot-spot contention within a program.

We use programs with 1024 processes for contention assessment, 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 (vectorized 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 an operation between the respective process and Rank 0.

Measurements are collected 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 nine processes performs the same operations to Rank 0, while the remaining processes are idle in a barrier. Therefore this corresponds to 11% contention. The third scenario is very similar to the second one, except that one in every five processes concurrently invokes one-sided operations to Rank 0. This then corresponds to 20% contention.

4.2.2. Noncontiguous data transfer operations

We conduct experiments to measure the performance of vectored put and get operations as representatives of noncontiguous data transfer functions. Fig. 10 shows the time of vectored put operations from all remote processes to Rank 0. Comparisons are provided among varying levels of contention (no contention, 11% contention, and 20% contention).

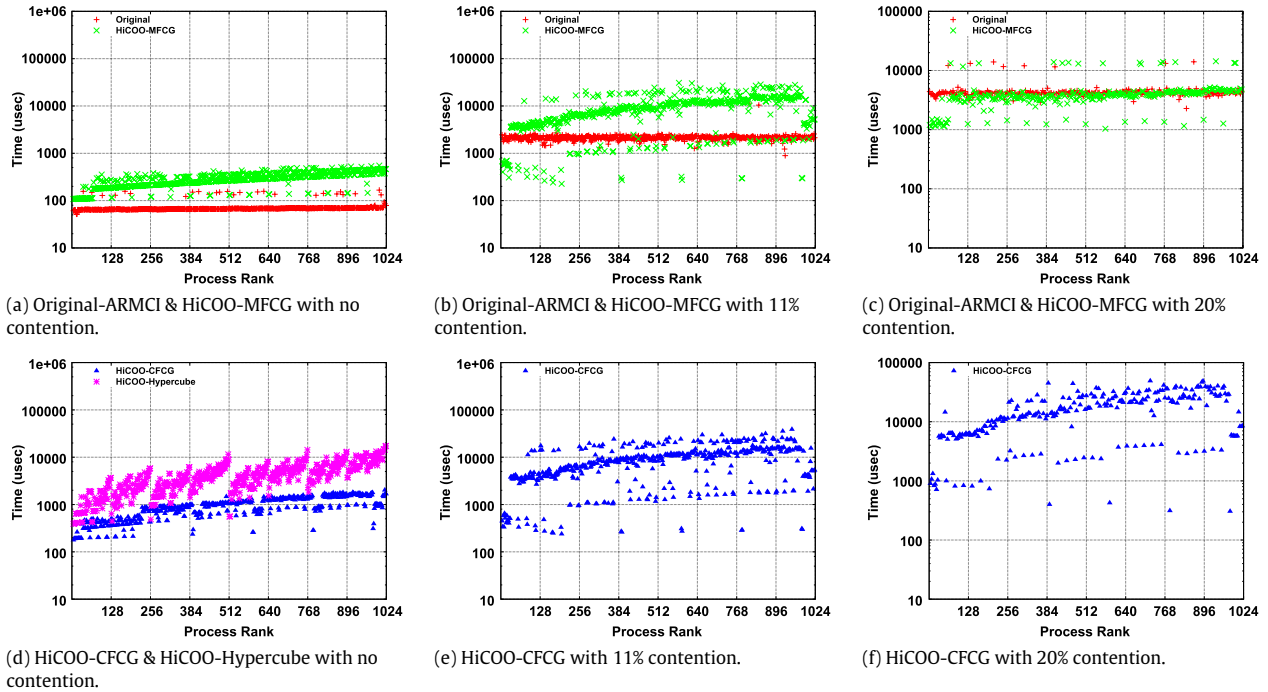


Fig. 10. Vectored data transfer operations under different contention.

Fig. 10(a) and (d) show the comparisons under no contention. Several behaviors are revealed by this figure. First, the use of MFCG, CFCG and Hypercube increases the time to complete noncontiguous data transfer operations between Rank 0 and other processes. Second, even though all processes are one step away from Rank 0 in the original ARMCI, the time to complete noncontiguous data transfers gradually increases with process rank. This suggests that the distance between a processes and Rank 0 in the underlying physical topology would play a role and contribute to the increased performance. This increment of time is magnified by the use of MFCG, CFCG and Hypercube. In particular, the results from HiCOO-Hypercube indicate that using a topology with very high dimensions for minimal memory consumption does not provide a good tradeoff to the performance. Third, with MFCG, the performance numbers from all processes form several distinct curves, representing differences in their (virtual-) topological relationship with respect to Rank 0. The same can be observed for HiCOO-CFCG and HiCOO-Hypercube as shown in Fig. 10(d).

Fig. 10(b), (c), (e), and (f) show performance comparisons with increased contention. HiCOO-Hypercube is not included in (e) and (f) because it takes too long to get a complete set of numbers. While contention increases the time to complete noncontiguous data transfer operations for all cases, it is evident that all virtual topologies exhibit contention resilience. While the performance of vectored put operations is degraded by nearly two orders of magnitude due to contention in the original ARMCI. With 20% contention, it becomes faster to complete noncontiguous data transfer operations for nearly all processes in the case of HiCOO-MFCG, compared to the original ARMCI. Comparing Fig. 10(b) and (c) it is interesting to note that HiCOO-MFCG also reduces the variations among all processes at higher hot-spot contention. The operation time for the group of processes in the middle has been brought down. This counterintuitive observation is because of the execution behavior of ARMCI communication server. When more processes are actively forwarding requests, they stay in the polling mode for handling requests and therefore have better response time in average. In summary, these results demonstrate that virtual topologies, such as MFCG and CFCG, can attenuate the pressure of

many contending noncontiguous data transfer operations, and lead to graceful resilience to contention.

4.2.3. Atomic fetch-&add operations

We measure the performance of fetch-&add as a representative of atomic operations. Fig. 11 shows the time for fetch-&add operations from all remote processes to Rank 0. Comparisons are provided among different virtual topologies, and among varying levels of contention (no contention, 11% contention, and 20% contention).

Fig. 11(a) and (d) show the comparisons under no contention. Similar observations can be made for atomic operations as revealed by Fig. 10(a) and (d). To be brief, these include (1) the use of MFCG, CFCG and Hypercube topologies increases the time to finish atomic operations under no contention; (2) the time of an atomic operation increases with a higher ranked process, suggesting a correspondence to the distance between the process and Rank 0 in the underlying physical topology; and (3) the performance numbers of atomic operations from all processes form several distinct groups, representing their relationship in the virtual topologies.

Fig. 11(b), (c), (e), and (f) show comparisons with increased contention. Again, Hypercube was not included in (e) and (f). While contention increases the time to complete atomic operations for all cases, it is also evident that all virtual topologies exhibit contention resilience. With 20% contention, it becomes faster to complete atomic operations for nearly all processes using HiCOO-MFCG than the original ARMCI. Under the same level of contention, even with HiCOO-CFCG, the time for fetch-&add is shorter for a majority of processes compared to the same with the original ARMCI. These results again demonstrate that virtual topologies, such as HiCOO-MFCG and HiCOO-CFCG, can greatly attenuate the pressure of contending atomic operations.

Furthermore, we investigate the benefits of HiCOO-MFCG for fetch-&add operations under 100% contention, i.e., all processes concurrently performs atomic fetch-&add operations to Rank 0. In this test, within two consecutive barrier (and ARMCI AllFence) operations, all process concurrently invoke 10 fetch-&add operations, record the time, and measure the average

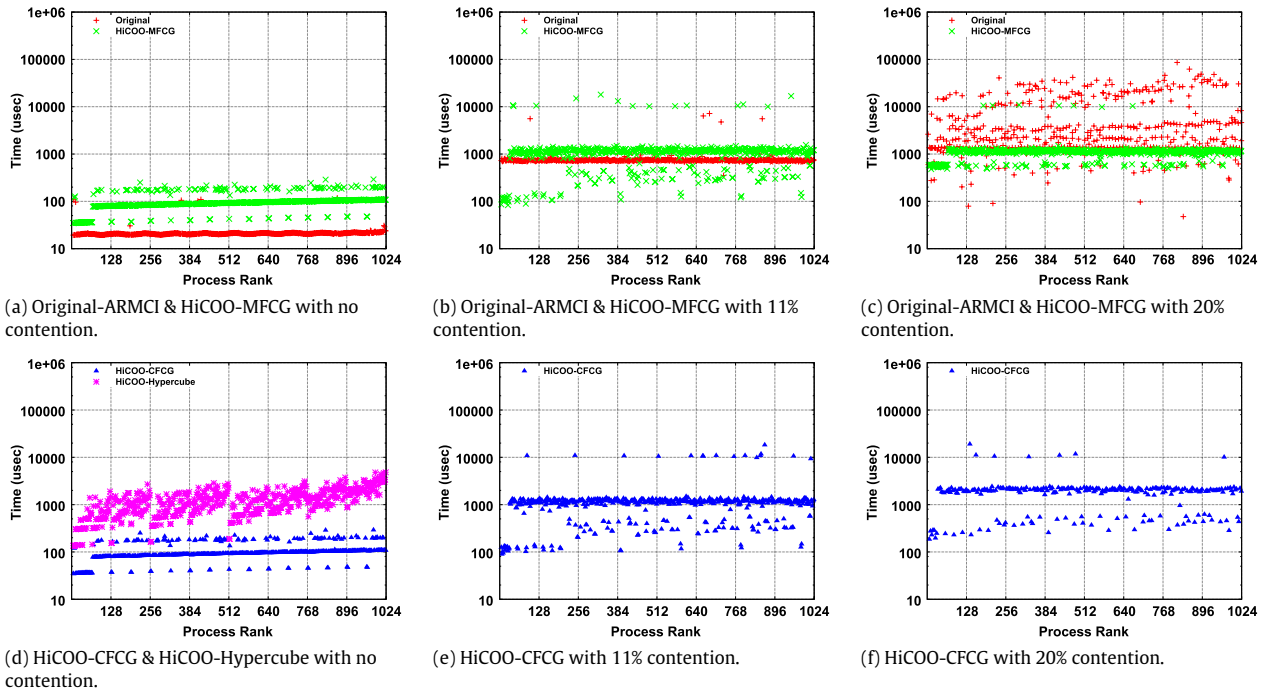


Fig. 11. Fetch-&-add operations under different contention.

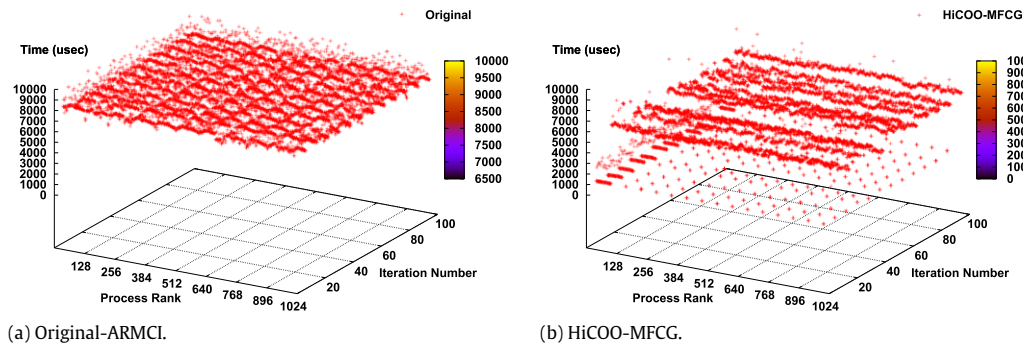


Fig. 12. Fetch-&-add operations under 100% contention.

for these 10 operations on their own. This is repeated for 100 iterations. Fig. 12 shows the performance comparison between the original ARMCI and HiCOO-MFCG, with 1024 processes and 100 iterations. With 100% contention, it takes $8000 + \mu s$ in average for a process to complete an atomic operation when using the original ARMCI. In contrast, when using HiCOO-MFCG, contention is dramatically reduced. Many processes finish within $2000 \mu s$; nearly all processes complete in $6000 \mu s$. This proves that the virtual topology HiCOO-MFCG is especially useful in attenuating the impacts of heavy contention.

5. Performance of communication operations and scientific applications

We have shown that virtual topologies can be very beneficial to reduce memory footprint and attenuate contention that would occur to hot-spot processes. It is important to find out how HiCOO will impact the ARMCI communication operations and what benefits they have to real applications. In the rest of experiments, we focus on HiCOO using the default topology MFCG.

5.1. 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.

5.1.1. 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

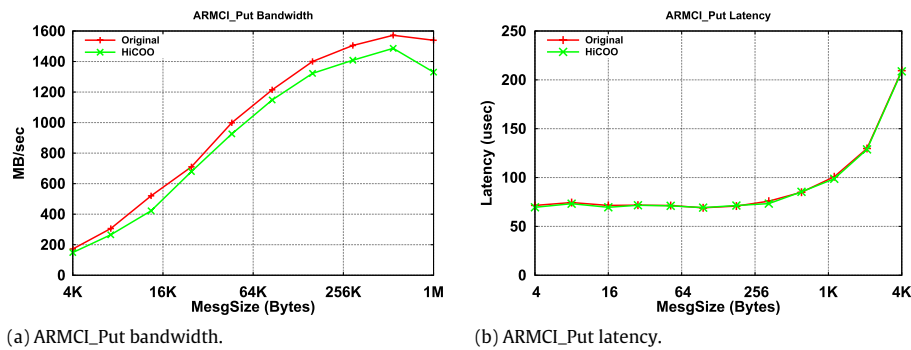


Fig. 13. ARMCI_Put latency and bandwidth.

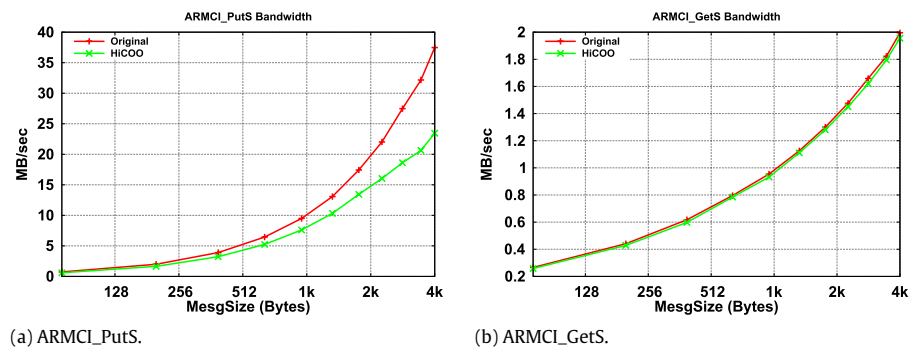


Fig. 14. Bandwidth of noncontiguous operations.

nodes form four groups of cooperative nodes. Our latency and bandwidth tests are different from the conventional ping-pong latency and stream-based bandwidth tests. 16 nodes are used to mimic the presence of message forwarding and compare the performance between the original ARMCI and HiCOO. Since there are 12 processes on each node, the latency and bandwidth numbers are measured when each node (and its network card) is loaded with communication generated by 12 processes. Fig. 13 shows the latency and bandwidth performance comparisons between the original ARMCI and HiCOO.

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.

5.1.2. 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 $\langle \text{addr}, \text{length} \rangle$ 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. Fig. 14 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 separately as the size of aggregation buffer is limited. 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 little difference between the original ARMCI and HiCOO. These results indicate that, while the performance of noncontiguous data put operations can be affected by the additional overhead of request forwarding, HiCOO is effective in minimizing such overhead with its request aggregation and hierarchical cooperation mechanisms, resulting in close performance to the original ARMCI.

5.1.3. 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.

Fig. 15 shows the performance results for all three operations. HiCOO achieves performance comparable to the original ARMCI for atomic lock operations, but it causes performance degradation for the accumulate and fetch-&-add operations. This performance difference is due to the underlying communication characteristics of these operations. Atomic lock operations do not transmit actual data between processes, but accumulate and fetch-&-add operations do.

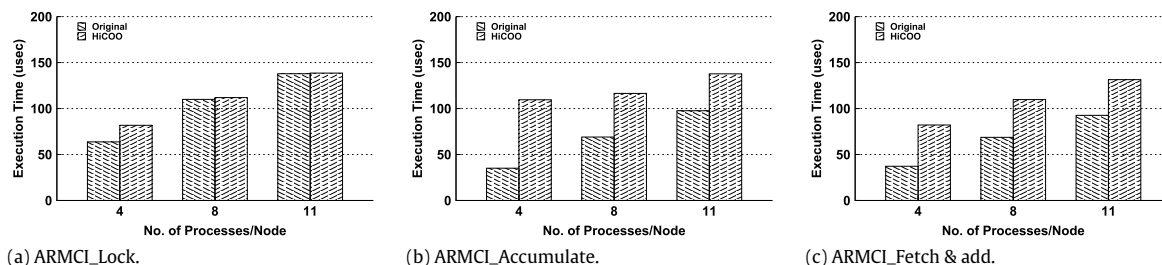


Fig. 15. Performance of atomic and synchronization operations.

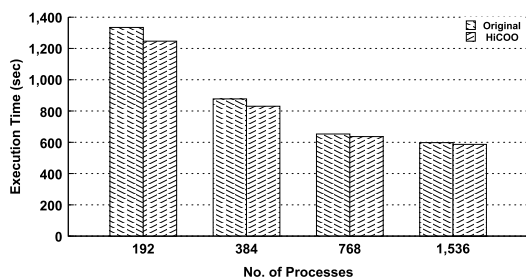


Fig. 16. The performance of NAS LU.

Taken together, our microbenchmark evaluation results indicate that while HiCOO strives to minimize memory consumption, its indirect one-sided communication through multinode cooperation causes performance overhead to atomic and synchronization operations. Care must be taken to achieve a good tradeoff between the memory consumption and the cost of atomic and synchronization operations for applications that frequently use such operations.

5.2. NAS LU application

The LU application in the NAS parallel benchmark suite [3] has been ported to the ARMCI runtime. It can scale to hundreds or a couple of thousand processes. We evaluate the performance impact of HiCOO to LU at this scale. Fig. 16 shows the performance of LU using HiCOO on a varying number of processes. As shown in the figure, HiCOO performs better or similar to the original ARMCI. At a lower number of processes, the benefit of HiCOO is slightly higher. Two observations can be made about these results. First, the LU application does not suffer much from hot-spot contention. Second, the reduction in memory footprint does not directly lead to the reduction in execution time, which is quite reasonable. On the other hand, these results are encouraging because they demonstrate that, despite the additional forwarding steps on ARMCI operations such as non-contiguous data transfer and atomic accumulation, HiCOO still brings comparable or better performance for applications such as LU.

5.3. A large-scale application: NWChem

We evaluate HiCOO using two electronic structure methods in a large-scale application NWChem [17]: the SiOSi3 method for Density Functional Theory (DFT) and the water model of Coupled Cluster (CC) in its CCSD(T) incarnation. Fig. 17 shows the performance of NWChem with the original ARMCI and HiCOO. The performance of SiOSi3 is shown in Fig. 17(a). HiCOO clearly performs better than the original ARMCI. HiCOO reduces the total execution time by as much as 52%. These results suggest that SiOSi3 is very prone to hot-spot contention, in which case HiCOO is very effective in mitigating the impact of contention.

Fig. 17(b) shows the performance of CCSD(T) water model when using the original ARMCI and HiCOO. ARMCI generally performs

better than HiCOO, except in one case at 10,000 cores. This result suggests that the total execution time for the water model does not benefit from HiCOO. The primary benefit of HiCOO is the ability to significantly reduce memory consumption of ARMCI low-level runtime library (as detailed in Section 4.1). This spares much more memory to be used by applications and help them achieve better scaling.

These application evaluation results demonstrate that HiCOO can hit the best balance of memory consumption, the need of request forwarding, and contention attenuation for the GAS runtime. With much reduced memory consumption at the runtime level, HiCOO in general performs comparably to the original ARMCI. Particularly when an application is experiencing hot-spot contention, HiCOO can mitigate the impact of contention and lead to significantly reduced total execution time.

6. Related work

The scalability of communication runtime involves a number 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. [6]. 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. Huang et al. [15] studied the scalability of communication for MPI on multicore clusters. Sun et al. [27] re-evaluated the impact of Amdahl's law in the multicore era. Our work investigates the scalability of runtime communication through multinode cooperation. Bonachea et al. [5] 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. [29] designed and implemented a scalable ARMCI communication library and demonstrated its strength in enabling GA and a real world scientific application, NWChem.

Topologies for communication networks have been well documented in the textbooks [19,12]. Exploiting scalable topologies for high performance communication networks has also been studied extensively in the literature, such as those in [8,24,25]. Our research represents an innovative use of classic topologies. By imposing mesh and cube topologies on top of small fully connected graphs (FCG), we introduced meshed FCGs (MFCG) and cubic FCGs (CFCG) to formalize challenging issues faced by today's petascale programming models.

Numerous algorithms were investigated to support deadlock-free message routing in interconnection networks. In their classic paper, Dally et al. [9] proposed deadlock-free message routing algorithms, such as dimension-order routing, for multiprocessor interconnection networks using the concept of virtual channels. Duato et al. [11] investigated deadlock-free adaptive multicast routing algorithms on worm-hole networks using a path-based routing model. Lin and Lionel [20] compared different multicast

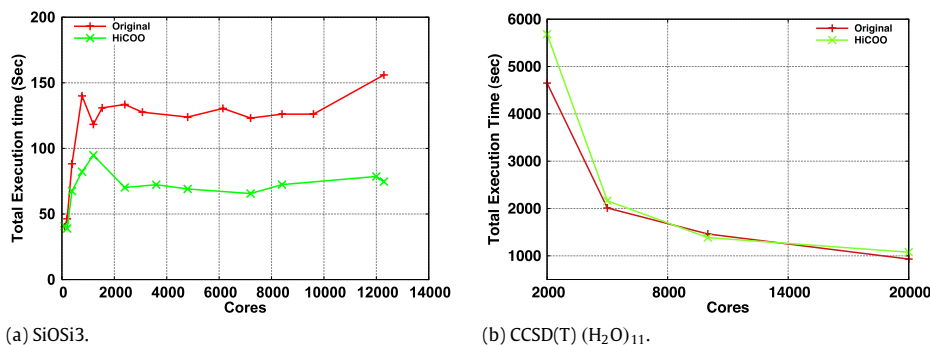


Fig. 17. NWChem execution time.

worm-hole routing algorithms, such as dual-path routing and multi-path routing, for multicomputers with 2D-mesh and Hypercube topologies. Our work builds on top of the dimension-order routing algorithm, and proposes the deadlock-free LDF (lowest dimension first) algorithm. LDF only needs to forward an ARMCI request once per dimension in MFCG, CFCG, and Hypercube. In addition, it allows partially populated MFCG and CFCG on any number of network nodes.

Many efforts studied the scalability of resource management for other contemporary programming models. Sur et al. [28] examined the memory scalability of various MPI implementations on the InfiniBand network. Koop et al. [18] exploited the use of message coalescing to reduce the memory requirements for MPI on InfiniBand clusters. Chen et al. [7] optimized the communication for UPC applications through a combination of techniques including redundancy elimination, split-phase communication, and communication coalescing. Our work differs from these earlier studies by introducing new virtual topologies to reveal the challenges of resource management and contention in the ARMCI Global Address Space runtime system. To the best of our knowledge, this is the first in literature to exploit the concept of virtual topology for systematic investigation of scalability and contention issues in Global Address Space programming models.

7. Conclusion

In conclusion, we have described HiCOO as a hierarchical Cooperation architecture for scalable communication in GAS programming models. HiCOO formulates a cooperative communication architecture with inter-node cooperation amongst multiple nodes (a.k.a multinode) and hierarchical cooperation among multinodes that are arranged in different virtual topologies. With HiCOO, we have systematically studied the resource management and contention issues in a GAS run-time system, ARMCI, on the petascale Jaguar Cray XT5 system at ORNL. We use several different virtual topologies to represent the management of communication resources in ARMCI as directed graphs, and substantiate it with two new virtual topologies, MFCG and CFCG, as well as a canonical topology Hypercube. Our extensive evaluation of all three virtual topologies demonstrates that MFCG is the best choice for HiCOO in accomplishing scalable memory usage and contention attenuation. While addressing the challenges of resource scalability and network contention, equally important is the need to maintain the performance of GAS programming models. We show that HiCOO improves ARMCI's resilience to network contention caused by transient and irregular communication patterns. At the same time, it can maintain or improve the performance of scientific applications.

In the future, we look forward to further optimization of the GA model and ARMCI on petascale systems. We are investigating large-scale applications that can leverage more memory at the application level for better total execution time. We intend to explore the benefits of virtual topologies to other GAS runtime systems such as GASNet [4]. We also plan to study the applicability of virtual topologies on other petascale platforms with different physical topologies, e.g., BlueGene/P [16,1]. Furthermore, we plan to investigate the benefits of virtual topologies in the context of PGAS languages such as UPC [31] and Co-Array Fortran [10].

Acknowledgments

This work was funded in part by NSF award CNS-1059376, UT-Battelle grant (UT-B-4000087151), and 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 US Department of Energy under Contract No. DE-AC05-00OR22725.

References

- [1] S. Alam, R. Barrett, M. Bast, M.R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J.S. Vetter, P. Worley, W. Yu, Early evaluation of ibm bluegene/p, in: SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, doi:10.1145/1413370.1413394.
- [2] E. Apra, R.J. Harrison, W. de Jong, A. Rendell, S. Tipparaju, V. Xantheas, R. Olsen, Liquid water: obtaining the right answer for the right reasons, in: Supercomputing, 2009. SC '09. Proceedings of the ACM/IEEE SC 2009 Conference, 2009.
- [3] D.H. Bailey, L. Dagum, E. Barszcz, H.D. Simon, Nas parallel benchmark results, in: Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, pp. 386–393.
- [4] D. Bonachea, C. Bell, P. Hargrove, M. Welcome, GASNet 2: An Alternative High-Performance Communication Interface, November 2004.
- [5] D. Bonachea, P. Hargrove, W.M., K. Yelick, Porting gasnet to portals: partitioned global address space (Pgas) language support for the cray XT, in: CUG'09: Cray User Group Meeting, 2009.
- [6] R. Brightwell, R. Riesen, A.B. Maccabe, Design, implementation, and performance of mpi on portals 3.0, Int. J. High Perform. Comput. Appl. 17 (1) (2003) 7–19.
- [7] W.-Y. Chen, C. Iancu, K. Yelick, 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, DC, USA, 2005, pp. 267–278. doi:http://dx.doi.org/10.1109/PACT.2005.13.
- [8] W.J. Dally, Performance analysis of k -ary n -cube interconnection networks, IEEE Trans. Comput. 39 (6) (1990) 775–785. doi:http://dx.doi.org/10.1109/12.53599.
- [9] W.J. Dally, C.L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, IEEE Trans. Comput. 36 (5) (1987) 547–553. doi:http://dx.doi.org/10.1109/TC.1987.1676939.
- [10] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, A multi-platform co-array fortran compiler, 2004, pp. 29–40. doi:10.1109/PACT.2004.1342539.
- [11] J. Duato, A theory of deadlock-free adaptive multicast routing in worm-hole networks, IEEE Trans. Parallel Distrib. Syst. 6 (9) (1995) 976–987. doi:http://dx.doi.org/10.1109/71.466634.

- [12] J. Duato, S. Yalamanchili, L. Ni, *Interconnection Networks: An Engineering Approach*, The IEEE Computer Society Press, 1997.
- [13] C.J. Glass, L.M. Ni, The turn model for adaptive routing, *SIGARCH Comput. Archit. News* 20 (2) (1992) 278–287. doi:<http://doi.acm.org/10.1145/146628.140384>.
- [14] Global arrays toolkit. <http://www.emsl.pnl.gov/docs/global>.
- [15] W. Huang, M.J. Koop, D.K. Panda, Efficient one-copy MPI shared memory communication in virtual machines, in: *Proceedings of the International Conference on Cluster Computing*, 2008.
- [16] IBM BG/P team, overview of the IBM blue gene/P project, *IBM J. Res. Dev.* 52 (1–2) (2008) 199–220.
- [17] R.A. Kendall, E. Aprà, D.E. Bernholdt, E.J. Bylaska, M. Dupuis, G.I. Fann, R.J. Harrison, J. Ju, J.A. Nichols, J. Nieplocha, T.P. Straatsma, T.L. Windus, A.T. Wong, High performance computational chemistry: an overview of NWChem a distributed parallel application, *Comput. Phys. Comm.* 128 (1–2) (2000) 260–283. doi:10.1016/S0010-4655(00)00065-5. URL: [http://dx.doi.org/10.1016/S0010-4655\(00\)00065-5](http://dx.doi.org/10.1016/S0010-4655(00)00065-5).
- [18] M.J. Koop, T. Jones, D.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*, IEEE Computer Society, Washington, DC, USA, 2007, pp. 495–504. doi:10.1109/CCGRID.2007.92.
- [19] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, first ed., Morgan Kaufmann Publishers, Inc., 1991.
- [20] X. Lin, L.M. Ni, Deadlock-free multicast wormhole routing in multicomputer networks, *SIGARCH Comput. Archit. News* 19 (3) (1991) 116–125. doi:10.1145/115953.115965.
- [21] LLNL, ASC Sequoia. http://www.llnl.gov/computing_resources/sequoia/.
- [22] NCSA, Blue waters: sustained petascale computing. <http://www.ncsa.illinois.edu/BlueWaters/>.
- [23] J. Nieplocha, V. Tipparaju, M. Krishnan, D.K. Panda, High performance remote memory access communication: the armci approach, *Int. J. High Perform. Comput. Appl.* 20 (2) (2006) 233–253. URL: <http://hpc.sagepub.com/cgi/content/abstract/20/2/233>.
- [24] D.K. Panda, Fast barrier synchronization in wormhole k -ary n -cube networks with multideestination worms, in: *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, IEEE Computer Society, Washington, DC, USA, 1995, p. 200.
- [25] F. Petrini, M. Vanneschi, k -ary n -trees: high performance networks for massively parallel architectures, in: *Parallel Processing Symposium, International*, 1997, p. 87. doi:10.1109/IPPS.1997.580853.
- [26] Report on experimental language X10. <http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf>, 2008.
- [27] X.-H. Sun, Y. Chen, Reevaluating amdahl's law in the multicore era, *J. Parallel Distrib. Comput.* 70 (2) (2010) 183–188.
- [28] S. Sur, M.J. Koop, D.K. Panda, 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, NY, USA, 2006, p. 105. doi:10.1145/1188455.1188565.
- [29] V. Tipparaju, E. Apra, W. Yu, J.S. Vetter, Enabling a highly-scalable global address space model for petascale computing, in: *Computing Frontiers'09*, 2010.
- [30] Top 500 supercomputing sites. <http://www.top500.org>.
- [31] UPC specifications, v1.2. <http://www.gwu.edu/~upc/publications/LBNL-59208.pdf>.



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 Ph.D. 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.



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 Ph.D. 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.