Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication

James Dinan, Pavan Balaji,
Jeff R. Hammond
Argonne National Laboratory
{dinan,balaji,jhammond}@anl.gov

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
sriram@pnnl.gov

Vinod Tipparaju
IEEE Member[†]
tipparajuv@ieee.org

***Abstract*—The industry-standard Message Passing Interface (MPI) provides one-sided communication functionality and is available on virtually every parallel computing system. However, it is believed that MPI's one-sided model is not rich enough to support higher-level global address space parallel programming models. We present the first successful application of MPI one-sided communication as a runtime system for a PGAS model, Global Arrays (GA). This work has an immediate impact on users of GA applications, such as NWChem, who often must wait several months to a year or more before GA becomes available on a new architecture. We explore challenges present in the application of MPI-2 to PGAS models and motivate new features in the upcoming MPI-3 standard. The performance of our system is evaluated on several popular high-performance computing architectures through communication benchmarking and application benchmarking using the NWChem computational chemistry suite.**

***Keywords*-One-sided communication; Global address space; MPI; Global Arrays; ARMCI; NWChem**

I. Introduction

Computational chemistry applications, such as NWChem [21], operate on large data sets that exceed the capacity of a single node. Global Arrays (GA) [16] was developed to address NWChem's need for a large, asynchronously accessible data space and it is now widely used across a variety of scientific domains. GA allows the programmer to create large, multidimensional shared arrays that span the memory of multiple nodes in a distributed-memory system. The programmer interacts with a GA through asynchronous, one-sided get, put, and accumulate operations as well as through high-level parallel mathematics routines. GA provides a convenient data access interface that uses high-level array indices; it also provides a rich lower-level interface for managing data distribution, exploiting locality, and managing communication.

GA is built on top of the aggregate remote memory copy interface (ARMCI) [15], a low-level, one-sided communication runtime system. ARMCI forms GA's portability layer; and when porting GA to a new platform, ARMCI must be implemented using that platform's native communication and data management primitives. Thus, ARMCI must be implemented for a particular platform before scientists are able to run NWChem and other GA applications.

[†]*Work conducted while at Oak Ridge National Laboratory.*

Because of its sophistication, creating an efficient and scalable implementation of ARMCI for a new system is challenging. On many platforms, vendors have not provided ARMCI support or have given it low priority, leaving implementation and tuning to third parties. Because of this, the availability of a stable, scalable ARMCI implementation on many new parallel computing architectures has lagged system delivery by several months to a year or more. Recent examples include two leadership class systems: Tianhe and the K computer. The Tianhe system has been available for two years and, at the time of writing, still does not have an ARMCI implementation available. The K computer has been available for several months and a vendor-supported ARMCI implementation is under development, but not yet production-ready.

The Message Passing Interface (MPI) [13] is the industry standard communication runtime for high-performance computing. An efficient and scalable MPI implementation is provided by the system vendor for virtually every parallel computing platform. One-sided remote memory access (RMA) communication support was introduced in version 2 of the MPI standard. MPI's one-sided communication is unique in targeting broad portability, even to non-cache-coherent architectures. This design goal, however, led to complexity in the model and semantic challenges that have been an impediment to adoption. In addition, it is believed that MPI RMA's strict semantics limit its usefulness as a runtime system for partitioned global address space (PGAS) programming models [6].

In this work, we study the semantic mismatch between MPI RMA and PGAS models like ARMCI; we develop techniques for overcoming this mismatch and present a complete implementation of GA's low-level ARMCI runtime system on MPI RMA. To our knowledge, this is the first demonstration that MPI's RMA interface is a suitable substrate for such higher-level PGAS and one-sided libraries. The impact of this work is several-fold:

1) By harnessing the portability of MPI, we have created a highly portable, high-performance runtime layer for GA that extends the usability of GA and the NWChem computational chemistry suite to a wide variety of systems, including systems where a native ARMCI implementation is not available or has not yet been fully

tuned for scaling. This has the immediate impact of enabling first-year science on new architectures, using e.g. the NWChem computational chemistry suite [21].

2) Currently, although GA and MPI are commonly used together, they have separate runtime systems, which leads to consumption of extra resources and potential losses in communication performance because of separate memory registration mechanisms. By enabling GA to share MPI's runtime system, we achieve a greater degree of interoperability that increases the resources available to the application.
3) MPI RMA adoption has presented a causality dilemma: few applications use MPI RMA because it is not well supported and sufficient investment has not been made by MPI implementers to tune the performance of MPI RMA. By enabling GA to use MPI RMA, we have introduced a sizable user base for MPI RMA.
4) While we demonstrate that MPI-2 RMA is sufficient to support GA, there remains much room for improvement in MPI's one-sided interface. We identify these gaps in current MPI-2 RMA and project toward the upcoming MPI-3 RMA standard.

Our implementation of ARMCI-MPI is compared with the best-available native implementations on four popular parallel computing architectures: IBM Blue Gene/P, an InfiniBand cluster, Cray XT5, and Cray XE6. Communication benchmarking demonstrates that MPI RMA provides lower, but comparable, performance to highly tuned native ARMCI implementations. Through application-level benchmarking, we demonstrate that in spite of the measurable performance gap between moderately tuned MPI one-sided operations and aggressively tuned ARMCI one-sided operations, ARMCI-MPI provides the level of support needed to achieve competitive and portable application-level performance for the NWChem computational chemistry suite. One one system, the Cray XE6, the ARMCI-MPI implementation provides performance and scalability that is superior to the vendor supplied ARMCI implementation.

In Section II we present an overview of the NWChem and GA software stack. In Sections III and IV we discuss the MPI RMA and ARMCI communication models, respectively. In Section V we present the design of the ARMCI-MPI layer that bridges the semantic gap between ARMCI and MPI. In Section VII we present our empirical evaluation using the NWChem computational chemistry application. In Section VIII we discuss extensions to GA and ARMCI that can enhance performance and portability while maintaining backward compatibility. We also discuss the implications of this work to the ongoing MPI-3 standardization process. In Section IX we discuss related work and conclude in Section X with a brief summary.

II. Overview

We begin with an overview of the NWChem computational chemistry suite, the Global Arrays software stack, and the industry-standard MPI.

A. NWChem

NWChem [21] is one of the most widely used computational chemistry software packages, especially on large-scale supercomputers, as a result of its breadth of functionality (particularly, the density functional and quantum many-body methods) and its portability. The computational methods implemented in NWChem are often both compute- and memory-intensive. In this paper, we focus on the coupled-cluster singles and doubles with perturbative triples method, known as CCSD(T), which requires $O(N^7)$ floating-point operations and $O(N^4)$ memory, where N is a measure of the size of the molecular system. If $N = 500$, which is required for a molecule with more than a dozen atoms, the memory required exceeds 100 GB, which exceeds the memory available on today's compute nodes. As N grows larger, the data structures of CCSD(T) require many terabytes of memory. Hence, distributed-data algorithms are required.

B. The Global Arrays Software Stack

Global Arrays [16] is a partitioned global address space (PGAS) library that provides a high-level interface to distributed, shared, multidimensional arrays. In the GA model, the memory of many nodes is aggregated to store large array structures, and locality information can be leveraged to optimize for local data access. GA data is accessed through one-sided GA_Get, GA_Put, and GA_Accumulate operations that specify the desired data by using array index ranges. These operations may access data that is stored across several nodes and strided within those nodes. The resulting communication and data assembly is managed by the GA runtime.

Global Arrays is layered on top of the ARMCI one-sided communication library [15]. ARMCI is a PGAS communication substrate that provides GA with the ability to allocate shared buffers that can be accessed through one-sided get, put, and accumulate operations. In addition, ARMCI provides highly tuned strided and I/O vector operations that can be used to perform the noncontiguous remote accesses generated as a result of GA accesses to array patches. As shown in Figure 1, GA typically utilizes both ARMCI and MPI. An alternative configuration of the GA software stack does not utilize MPI; however, on almost all high-end systems, MPI is used for two-sided and collective communication, process management, and bootstrapping.

C. The Message Passing Interface

MPI [13] is an industry-standard communication library for parallel programming. The first MPI standard defined the core two-sided messaging and collective operations. With

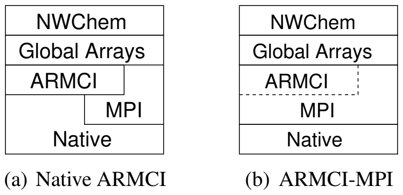


Figure 1. Global Arrays software stack: native and MPI implementations.

MPI-2, new one-sided functionality was added in response to the growing popularity of one-sided models. In contrast with one-sided models of the time, the MPI Forum created a model that uses a window concept to explicitly manage data consistency and provide portability, even to non-cache-coherent architectures. A consequence of this design was added complexity in the interface and data access semantics. For this reason, the one-sided functionality has not enjoyed the same degree of popularity as the core two-sided and collective communication functionality.

III. Using MPI RMA Effectively

MPI's one-sided Remote Memory Access programming interface supports two modes of operation: active and passive. Active mode requires synchronization among all parties involved in the RMA operation. Passive mode is asynchronous and requires participation only from the initiator, or *origin process*, of the operation. Because of the synchronization involved in active-mode communication, passive-mode RMA is more suitable for the asynchronous communication model used by GA.

MPI's passive-mode operations must be performed within an access *epoch* that is initiated through a call to MPI_Win_Lock and completed by a call to MPI_Win_Unlock. On systems where core-core and processor-network memory accesses are noncoherent, concurrent accesses to memory can overlap on data transfer units, resulting in data corruption even if they do not access the same location. Some other systems have weak core-core and core-network consistency. Calls to lock or unlock ensure that accesses are ordered and result in the expected data. All one-sided operations initiated during a passive mode epoch are nonblocking and logically occur concurrently. Thus, conflicting operations (e.g., overlapping put and get operations) within an epoch can cause unexpected results; such groupings of operations are defined to be an error by the MPI standard.

Passive-mode MPI RMA supports two data access (locking) modes: shared and exclusive. Shared locks allow multiple origins to access the same target concurrently, whereas exclusive locks prevent this type of access. If two origin processes perform concurrent conflicting operations on the same target (e.g., both use a shared lock), unexpected results can be generated, leading to data corruption. Thus, this behavior is also defined to be an error by the MPI standard.

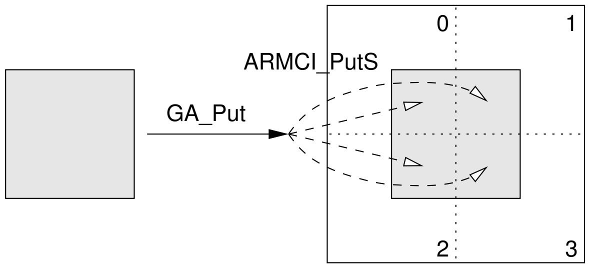


Figure 2. Put operation on a GA distributed across four processes. This operation results in four ARMCI noncontiguous put operations.

An MPI window object is logically separated into two copies: a public copy and a private copy. This separation is to accommodate systems with no coherence between the network and the host processor. Remote accesses are performed on the public copy, and local load/store accesses are performed on the private copy. When direct load/store accesses are performed, the private copy must be synchronized with the public copy; however, the MPI library has no knowledge of which data has been updated. Thus, the entire window may be copied, potentially resulting in memory corruption if a remote process is in the process of accessing the public window. Therefore, unless careful synchronization is used, direct local access should be performed only while the window is locked for exclusive access.

IV. The ARMCI Runtime System

ARMCI provides a partitioned global address space (PGAS) view of distributed shared data, where a global address takes the form $\langle process\_id, address\rangle$. Allocation and freeing of globally accessible memory are performed via the collective ARMCI_Malloc and ARMCI_Free routines. ARMCI_Malloc returns a vector of pointers to the patch of globally accessible memory on each process. The pointer for a given target process is used to calculate and specify the address component for the destination of Put or Accumulate operations or to specify the source location of a Get operation. As shown in Figure 2, GA operations can result in accesses to data stored on several processes. The array elements accessed on each process are often noncontiguous, and ARMCI provides a noncontiguous communication interface to efficiently support this mode of communication.

ARMCI also supports processor groups that are created through collective and noncollective group creation calls. ARMCI's communication operations operate on absolute process identities (ranks in the ARMCI world group) rather than group identities. Thus, group ids must be converted to absolute ids through a call to ARMCI_Absolute_id before they can be used in communication operations.

A. ARMCI Semantics

ARMCI distinguishes between local and remote completion for its communication operations. When a blocking operation returns or a nonblocking operation is completed, the operation is said to be locally complete and the local buffer is available for use by the application. In the case of a get operation, this also means that the desired data has been fetched and the communication is complete. In the case of a put or accumulate operation, it does not guarantee that the operation has completed remotely. If remote completion is desired, the programmer must perform an ARMCI_Fence operation.

ARMCI also follows a location-consistent model for overlapping data access [10]. In this model, a process observes its own operations with respect to a given target process p in the order in which they completed locally. Another process accessing the same data on p may observe a different ordering of these operations. This model provides flexibility that can be leveraged for performance while still giving the programmer the convenience of a shared-memory-like data access model.

In order to achieve a high degree of interoperability, GA and, by extension, ARMCI guarantee asynchronous progress for one-sided operations. This guarantees that communication will make progress even if the target of the communication operation is blocked in a non-ARMCI/GA (e.g., DGEMM) call, allowing blocking ARMCI operations to be interleaved with blocking MPI operations in the same program with no risk of deadlock. Most native ARMCI implementations use a communication helper thread (CHT) that ensures asynchronous progress and provides support for ARMCI operations that are not natively supported, such as double-precision accumulate and ARMCI's atomic swap and atomic fetch-and-add operations.

V. Design of ARMCI-MPI

We align MPI's restrictive, but portable, one-sided semantics with ARMCI through an intermediate global data management layer called global memory regions (GMR). In addition to translating between ARMCI and MPI data spaces, GMR manages accesses to global memory to ensure that MPI semantics are not violated.

A. Address and Rank Translation

ARMCI communication operations are performed on global addresses, whereas MPI one-sided operations are performed by using RMA windows and window displacements. GMR maintains a translation table that is used to look up the allocation corresponding to an ARMCI global address and retrieve its GMR handle. GMR handles contain all the information needed to access a global memory region, including a reference to its MPI window object and the ARMCI group on which the allocation was performed. ARMCI communication operations are performed on absolute process ids, whereas MPI's one-sided communication operations target a rank in the given window's group. Thus, GMR must also translate between ARMCI absolute process ids and ranks in the target window.

ARMCI supports two modes of process group creation: collective and noncollective. Collective group creation is implemented directly by using MPI communicators; however, noncollective group creation cannot be performed directly by using MPI's communicator creation interface. Instead, we use the recursive intercommunicator creation and merging algorithm presented in our prior work [9]. Using this technique, we can back both types of process groups using an MPI communicator.

B. Global Memory Allocation

When global memory allocation is performed, a new GMR handle is created and entered into the translation table. An MPI window is then created on the given ARMCI group by using the group's communicator and is attached to the GMR handle. All processes participating in the allocation then perform an all-to-all exchange of their local base addresses to build the base address vector that will be returned to the user. If a process requests an allocation of size 0, a base address of NULL will be used for this process. Likewise, for all processes not participating in the allocation, a base address of NULL is entered in the translation table and into the base address vector entry for that process.

In order to free an GMR, the MPI window object must be located and then collectively freed. If any processes have a zero-size slice of the allocation, they will supply NULL to the call to ARMCI_Free. The translation table may also have several NULL entries from other allocations for this process id. In order to locate the correct GMR handle, a leader process is selected by performing a reduction on process ids, where processes put forth their rank if they have been given a non-NULL global address for the free operation. The selected leader then broadcasts this address to the remaining processes, and this $\langle leader\_rank, address\rangle$ pair is used to look up the GMR handle and free the MPI window.

C. Avoiding Data Access Conflicts

MPI RMA defines any concurrent, conflicting operations to be an error because they have the potential to corrupt memory, for example, on noncoherent systems. Thus, GMR must ensure that operations issued within the same epoch are nonconflicting and that operations issued by different sources targeting the same process also do not conflict. In order to ensure that operations issued by a given process do not conflict, we issue each within its own epoch. In general, an ARMCI process has no knowledge of the operations issued by other processes. Therefore, MPI's exclusive access mode must be used to ensure that conflicts do not occur. As we discuss in Section VIII-A, however, when more application-level knowledge is available, these strict access modes can be relaxed.

D. Synchronization and Atomic Operations

In addition to process-level synchronization via ARMCI_Send, ARMCI_Recv, and ARMCI_Barrier, ARMCI provides mutexes and atomic operations that can be used to perform asynchronous, data-driven synchronization.

We have implemented the ARMCI mutexes API using the MPI RMA queueing mutex algorithm of Latham et al. [12], which is the most scalable one-sided mutual exclusion algorithm currently known for MPI RMA. In this algorithm, a byte vector B of length $nproc$ is created on the process hosting the mutex; the ith entry in this vector indicates whether process i has requested the lock. Initially $B[0 \ldots nproc - 1] = 0$. A lock operation from process i performs several nonoverlapping communication operations in a single MPI RMA exclusive access epoch: entry $B[i]$ is set to 1, and all other entries are fetched. If all other entries are 0, the lock operation has succeeded; otherwise the lock operation has effectively enqueued process i in the waiting queue for the mutex. Once enqueued, the process waits on an MPI_Recv operation from a wildcard source. Thus, when a process is blocked in the lock operation, it waits locally for a message to arrive and does not generate network traffic.

When process i performs an unlock operation, it again performs an exclusive RMA access epoch on B that sets $B[i] = 0$ and fetches all other entries. B is then scanned for an enqueued request starting at entry $i + 1$, which ensures fairness. If a request is found in the queue, a zero-byte notification message is sent to this process, forwarding the lock. If no request is found, the unlock is finished.

In addition to mutexes, ARMCI provides atomic swap atomic fetch-and-add routines that operate on a single-integer or long-integer location in memory. These operations are defined to be atomic with respect to other ARMCI RMW operations, but atomicity with respect to other operations is not guaranteed. The MPI 2.2 standard does not provide any operation that can perform atomic read-modify-write, and such a sequence of operations is forbidden within an access epoch since the read and write are necessarily conflicting. Thus, the only possible approach using the current MPI standard is to implement RMW operations using a mutex. We associate a mutex with each GMR and perform RMW operations in two epochs (read and write).

E. Direct Local Access to Global Data

Direct load/store access to memory exposed in an MPI window conflicts with all other accesses to that window region. To avoid these conflicts, we adopt a strategy of performing direct access only within exclusive mode epochs. GA provides GA_Access/Release calls that are used to acquire and release direct access to data stored within a GA; however, ARMCI does not currently provide such functions. We have extended the ARMCI API to include ARMCI_Access_begin/end calls that initiate and complete direct access to data within a GMR. This extension has required the modifications to GA described in Section VIII-A, but it requires no application-level modifications.

1) Communicating with Global Buffers: In the ARMCI model, the user can provide a memory location that is globally accessible as the local buffer in a communication operation. Several issues arise when a global buffer is used as the local source or destination in a communication operation: we may need to lock the same window more than once, which is forbidden by MPI; the local access may conflict with another access to the local window region; or, by locking multiple windows, we may induce a deadlock due to circular dependence between two processes' communication operations. In order to avoid these issues, ARMCI-MPI must check local buffers to ensure they are not exposed in an MPI window. If they are, the accesses must be managed correctly to prevent data corruption, an MPI error, or deadlock.

Several strategies are possible for managing communication where the local buffer is in global space. However, because of the possibility of deadlock from locking multiple windows, the only safe method is to stage the data through a temporary local buffer. Thus, in a put or accumulate operation, the source window is locked and the data copied into a temporary local buffer. This exclusive lock for local access is then released before requesting the lock on the target process and performing the communication operation, eliminating the possibility of deadlock. Many MPI implementations have extended the MPI standard on coherent memory systems to allow for concurrent access to shared data. On such systems, ARMCI's global buffer management mechanisms can be disabled to provide better performance.

F. Aligning Consistency and Progress Semantics

Given the described mapping of ARMCI to MPI operations, three dimensions of ARMCI's communication semantics must to be aligned with MPI: local/remote completion semantics, location consistent global data access, and asynchronous progress.

MPI's RMA operations do not distinguish between local and remote completion. All operations within the same epoch are logically concurrent; operations are not guaranteed to complete until the MPI_Unlock call that completes the epoch. Since ARMCI-MPI must issue each operation within its own exclusive access epoch to avoid conflicts, when the ARMCI-MPI operation returns, the operation will have completed both locally and remotely. These completion semantics guarantee that an origin process will observe its operations in the order in which they were issued, which captures ARMCI's location-consistent semantics. Since communication operations are not left in-flight, the implementation of ARMCI's Fence operation is a no-op. Further, just like under ARMCI, the MPI standard specifies that RMA operations should make asynchronous progress. On some platforms, a performance cost is associated with this requirement, and implementers have chosen to make this a runtime option that must be enabled.

VI. Noncontiguous Communication Operations

The most performance-critical component in ARMCI is its support for noncontiguous communication. As shown in Figure 2, a GA communication operation is often translated into several ARMCI communication operations. The resulting ARMCI operations may access only a part of each dimension of the array, resulting in a noncontiguous transfer. ARMCI provides two facilities for performing noncontiguous data transfers: generalized I/O vectors and a strided API.

A. Generalized I/O Vector Operations

ARMCI's generalized I/O vector (IOV) operations perform a series of data transfers of the same size. The programmer describes an IOV operation as an armci_giov_t IOV descriptor:

```
typedef struct {
  void **src_ptr_array; // Source address of each segment
  void **dst_ptr_array; // Dest. address of each segment
  int    bytes;         // Length of each segment in bytes
  int    ptr_array_len; // Number of data segments
} armci_giov_t;
```

ARMCI-MPI provides three methods for performing IOV operations: *conservative*, *batched*, and *direct*. The *conservative* method issues one RMA communication operation per segment, each in a separate RMA epoch. This method is conservative because it permits each segment to correspond to a different GMR (call to ARMCI_Malloc) and allows segments to overlap (although we are not aware of any application that performs this type of operation). The *batched* method requires that all segments in the same GMR have no overlap (the overwhelmingly common use case). It issues up to B operations per epoch, where B is a configurable parameter and has a default setting of 0, or unlimited. The *direct* method generates two MPI indexed datatypes to represent the data layout at the source and at the destination. A single communication operation is issued using these datatypes, thus allowing the MPI implementation to select the most efficient way to perform the given operation (e.g., pack/unpack of the data or batched) for the given system. This method also requires that global addresses correspond to the same GMR and not overlap.

B. Checking for IOV Errors

The batched and datatype methods of IOV transfer can generate an MPI error if segments are overlapping or if they correspond to different GMRs. It is possible (but not required) for MPI implementations to detect this and generate an MPI error; however it is possible for data to already be corrupted when this error is detected, making recovery impossible. We therefore have added an *auto* method that scans the IOV descriptor to detect these conditions and use the conservative method when needed. A naive approach requires an $O(N^2)$ scan of the segments to detect overlap. For applications such as NWChem, N can be tens to hundreds of thousands of segments for a typical GA operation.

We have developed an $O(N \cdot \log N)$ approach to overlap detection that uses a binary tree to detect conflicts. In this approach, each node represents an address range $[lo..hi]$; nodes in the tree are ordered such that, for any given node, all nodes in the left subtree are less than lo and all nodes in the right subtree are greater than hi. When considering a new range, $[lo'..hi']$, a conflict can be detected by searching the tree for a node where lo' or hi' lie in the range $[lo..hi]$ or $lo' < lo \land hi' > hi$. A full proof that such a node will be encountered in a search is outside the scope of this paper. Roughly speaking, however, a conflicting node must be encountered during a search of the tree since the tree is ordered and such a node would not lie to the left or right of the node being checked. If a conflict is not found, the range is inserted into the tree, and the next range is checked. In our implementation, we used an AVL tree [1] for its self-balancing property, and we merged the checking and insertion steps. When insertion fails, the range is not inserted into the tree, and an error is returned, indicating that the conservative transfer method should be used. The simpler behavior and organization with respect to conflict handling are the primary differences between our conflict tree structure and an interval tree [7], which supports multiple overlapping regions and queries that return all matching regions.

Table I
GA/ARMCI strided notation.

Strided Notation	Definition
src	Source pointer
dst	Destination pointer
sl	Stride level ($dimensionality - 1$)
count[ ]	Num. units in each dimension, length $sl + 1$
src_strd[ ]	Source stride array, length sl
dst_strd[ ]	Destination stride array, length sl

C. Strided Operations

Noncontiguous array operations in ARMCI can also be expressed by using ARMCI/GA strided notation. This representation, shown in Table I, is more compact than the IOV representation and lends itself readily to the pointer arithmetic needed to perform the desired operation.

In this representation, src and dst point to the first element to be transferred and the start of the target buffer. The number of units to be transferred in each dimension is specified in the count array; count[0] specifies the number of bytes in the contiguous dimension. Source and destination stride arrays are both specified to allow for different layouts of the source and destination array. The stride array specifies the displacement in bytes of the given dimension from the base address.

Strided notation can be readily translated into IOV representation by using the algorithm presented in Algorithm 1. Translation into IOV operations is a common implementation strategy for ARMCI and is supported by ARMCI-MPI; Algorithm 1 is used to construct an iterator and reduce space overheads. In addition to *IOV translation*, ARMCI-MPI provides a *direct* implementation of strided operations that translates ARMCI strided notation directly into an MPI subarray derived datatype. Similar to the IOV datatype implementation, once the MPI datatype has been generated, a single MPI RMA communication operation is issued to hand off the communication to MPI, which can optimize the operation, for example. by doing pack/unpack or by using special machine capabilities.

Algorithm 1 Algorithm to convert ARMCI strided operations to the generalized I/O vector representation.

```
In: $sl$,$src$,$dst$,$src\_strd[sl]$,$dst\_strd[sl]$,$count[sl+1]$
Out: $iov$
Let: $xfer \leftarrow 0$, $idx[0 \dots sl-1] \leftarrow 0$
while $idx[sl-1] < count[sl]$ do
  Let: $disp\_src \leftarrow 0$, $disp\_dst \leftarrow 0$

  // Calculate displacements from the base pointers
  for $i = 0 \dots sl-1$ do
    $disp\_src \leftarrow disp\_src + src\_strd[i] \cdot idx[i]$
    $disp\_dst \leftarrow disp\_dst + dst\_strd[i] \cdot idx[i]$
  end for
  $iov.src\_ptr\_array[xfer] \leftarrow src\_ptr + disp\_src$
  $iov.dst\_ptr\_array[xfer] \leftarrow dst\_ptr + disp\_dst$

  // Increment innermost index and propagate "carry"
  $idx[0] \leftarrow idx[0] + 1$
  for $i = 0 \dots sl-1$ do
    if $idx[i] \geq count[i+1]$ then
      $idx[i] \leftarrow 0$
      $idx[i+1] \leftarrow idx[i+1] + 1$
    end if
  end for
  $xfer \leftarrow xfer + 1$
end while
```

Translation from ARMCI's strided notation to an MPI subarray type involves regenerating the array and subarray dimensions by using the count and stride array information. In contrast to ARMCI's strided notation, which is convenient for pointer arithmetic, MPI's subarray types are expressed in terms of the array dimensions, subarray starting index, and subarray dimensions. Thus, we must translate backwards to get the higher-level information that is present at the GA level but not at the lower ARMCI level. Datatypes must be generated for both the source and destination. For the origin buffer the datatype is generated relative to the origin pointer, and for the target the datatype is generated relative to the beginning of the window. The subarray can be expressed as an array that originates at index 0 in each dimension, which has dimensionality $sl+1$; the size of each dimension is given as the count array. This subarray is contained within an array whose dimensions are given via the (src or dst, respectively) stride array. With MPI's C subarray dimension ordering, the parent array has outermost and innermost dimensions $dim[0] = stride[0]$ and $dim[sl] = count[sl]$, respectively. The inner dimensions, $i = sl - 1 \dots sl$, can be found as $dim[i] = count[i]/count[i-1]$.

VII. Experimental Evaluation

We have evaluated ARMCI-MPI on four platforms that capture a broad range of the high-performance computing landscape: IBM Blue Gene/P, Cray XT5, Cray XE6, and InfiniBand clusters. These systems currently have a native ARMCI implementation available and represent important execution targets for NWChem. The hardware characteristics of these systems are given in Table II. For each platform, we compare the raw communication performance of ARMCI-MPI with ARMCI-Native. We also conduct an application performance study using NWChem.

A. Communication Benchmarks

In Figure 3 we present the bandwidth achieved for contiguous ARMCI get, put, and double-precision accumulate operations over a range of message sizes. Data is presented for the native implementation of ARMCI as well as ARMCI-MPI.

From this data we can see that ARMCI-MPI's get and put performance is less than but comparable to that of Blue Gene/P and InfiniBand cluster. However, double-precision accumulate performance does not keep up, especially on the InfiniBand cluster where the bandwidth gap is more than 1.5 GB/sec. This indicates an area where a great deal of improvement is needed in MPI IS-sided implementations. On the Cray XT, performance is comparable for messages up to 32 kB; however, beyond this point, MPI achieves half of the bandwidth achieved by native ARMCI, indicating that performance tuning in the MPI implementation could benefit this transfer regime. On the Cray XE ARMCI-MPI achieves *twice* the bandwidth of native ARMCI for put and get on large messages and a 25% higher bandwidth for double precision accumulate.

In Figure 4 we present communication bandwidth measurements for strided ARMCI operations on our four test systems. In these figures, we show plots where the contiguous segment being transferred is 16 and 1,024 bytes for a number of segments ranging from 1 to 1,024.

On Blue Gene/P, the direct strided method gives the best performance for small segments as a result of the benefit of data packing done by the MPI implementation. For larger segments, however, the slow speed of BG/P's processors becomes an impediment for data packing, and we see that the batched method of transfer gives performance that is

Table II
EXPERIMENTAL PLATFORMS AND SYSTEM CHARACTERISTICS.

System	Nodes	Cores per Node	Memory per Node	Interconnect	MPI Version
IBM Blue Gene/P (Intrepid)	40,960	1 x 4	2 GB	3D Torus	IBM MPI
Cluster (Fusion)	320	2 x 4	36 GB	InfiniBand QDR	MVAPICH2 1.6
Cray XT5 (Jaguar PF)	18,688	2 x 6	16 GB	Seastar 2+	Cray MPI
Cray XE6 (Hopper II)	6,392	2 x 12	32 GB	Gemini	Cray MPI

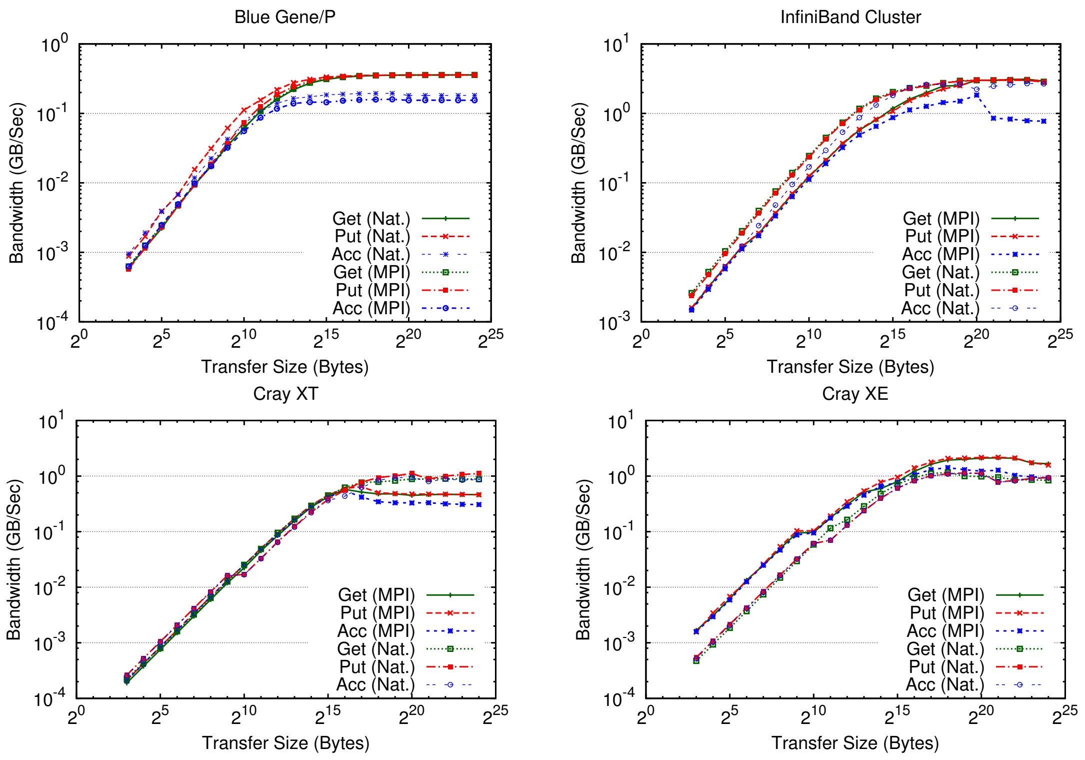


Figure 3. Bandwidth achieved for contiguous ARMCI operations using ARMCI-MPI and native ARMCI.

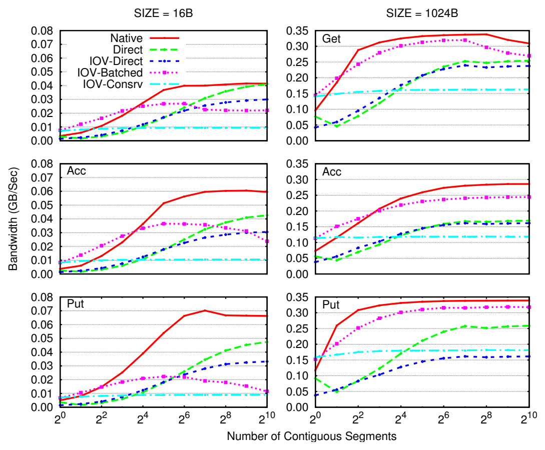
near that of the native ARMCI implementation. On the InfiniBand cluster, the direct method is again advantageous for small segments; however, for large segments the batched method offers better bandwidth. For both this and the BG/P case, these results indicates that tuning is needed within the MPI implementation to automatically switch between data packing and direct transfer modes when handling noncontiguous datatypes. For large numbers of segments on InfiniBand, performance of the batched transfer method suffers severely. This is due to a performance-related issue in MPICH-2 that has recently been fixed but not yet integrated into MVAPICH-2.

On the Cray-XT, we see that both MPI datatypes implementations of ARMCI's strided operations (IOV Dtype and Direct) outperform the batched method. For more than tens of segments, however, performance falls off to roughly half the native bandwidth, indicating some internal inefficiencies in the runtime. On the Cray XE, the story is reversed: ARMCI-MPI achieves significantly higher bandwidth for put and get transfers and matches native performance for double-precision accumulate.

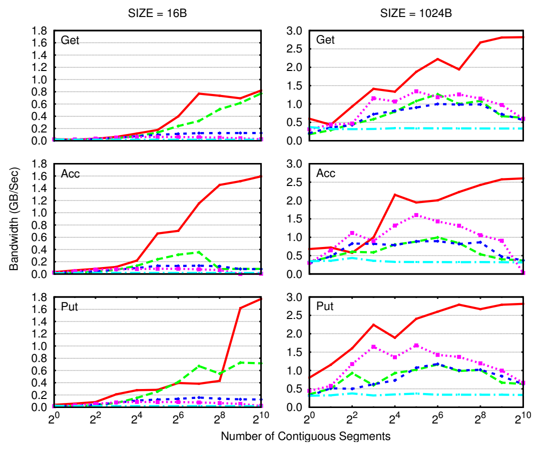
Overall, these results indicate that further tuning is needed across several important MPI implementations to better handle switching from data packing to direct transmission and to better handle large numbers of noncontiguous transfers within a single passive mode epoch. The Cray XE is a new system architecture, and its software stack continues to be improved. Even though the performance of ARMCI-MPI is well below peak for this system, MPI RMA offers better performance than the development release of ARMCI that is currently available.

B. Evaluation of Interoperability

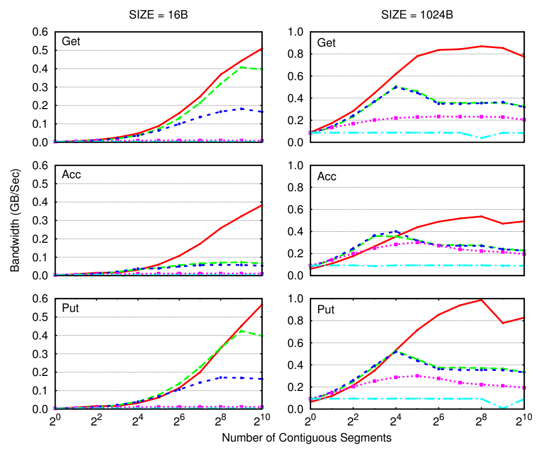
Typically, a GA application utilizes both ARMCI and MPI for its runtime support. When both runtime systems are active, additional memory and processing resources are consumed. In addition, many systems require that memory used for communication be pinned, or locked to physical frames, and registered with the network device so that DMA transfers can be performed. Registration mechanisms



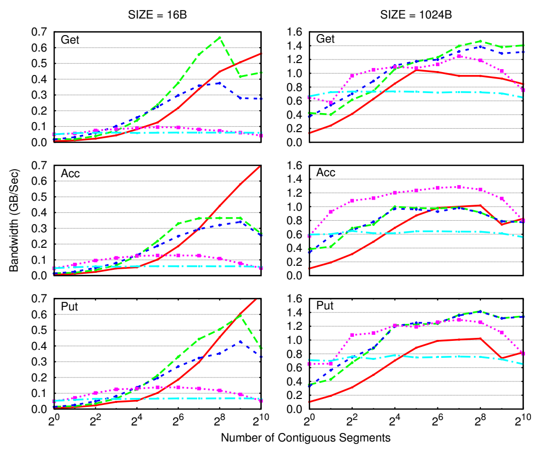
(a) Blue Gene/P



(b) InfiniBand Cluster



(c) Cray XT



(d) Cray XE

Figure 4. Bandwidth comparison for strided ARMCI operations when using several ARMCI-MPI strided methods and native ARMCI. Contiguous segment sizes of 16 and 1024 bytes are shown, and the number of segments ranges from 1 to 1,024.

typically cannot be shared between two runtime systems; thus, both ARMCI and MPI must maintain separate buffer pinning mechanisms. This strategy can lead to additional resource consumption in memory and on the network device. In addition, it can impact communication performance because each runtime system will be unaware that buffers have already been pinned by the other runtime system.

In Figure 5 we present an example of communication performance loss due to mismatched buffer registration. The figure shows the bandwidth achieved for contiguous ARMCI and MPI get operations on the InfiniBand cluster using buffers that are registered by ARMCI and MPI runtime systems. For a native ARMCI get, performance with the ARMCI allocated buffer is the best; however, there is a significant bandwidth gap when communicating using a buffer allocated by MPI forces ARMCI to switch to its nonpinned communication path. While ARMCI allocates prepinned buffers from a pinned region, MVAPICH uses on-demand memory registration. At the time of writing, MVAPICH does not support prepinned allocation; however, it is allowed by the MPI standard through the use of an info argument to the MPI_Alloc_mem routine. On-demand registration is more flexible than prepinning; however, it can consume more resources because of fragmentation, and it has a high registration cost. In Figure 5 we show the bandwidth achieved by using an MPI get when MPI has touched (i.e., registered) the buffer, versus when MPI has not touched the given buffer. For transfers smaller than 8kB, or two pages, MVAPICH copies the data into internal prepinned buffers. For transfers larger than this, MVAPICH pins the buffer and does the transfers directly; the high cost of on-demand registration can be seen for transfers in this range.

C. NWChem Experimental Setup

In order to evaluate the performance of NWChem, a water cluster was modeled by using the CCSD(T) method and the aug-cc-pVTZ basis set. Water clusters are the subject of intense scientific interest and have been previously considered in performance-oriented studies involving NWChem.

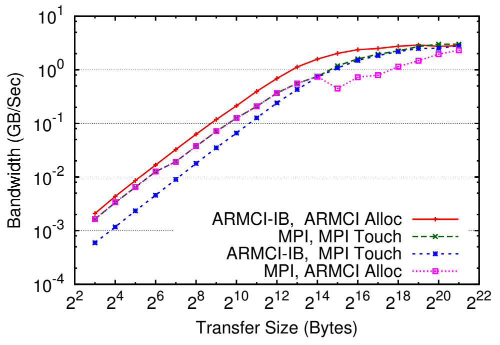


Figure 5. Bandwidth achieved for contiguous get operations for ARMCI and MPI when using ARMCI and MPI allocated local buffers.

The water pentamer (henceforth referred to as w5) has 50 electrons, which are represented by using 460 atom-centered Gaussian basis functions. For simplicity, no higher angular momentum functions were removed, in contrast to Ref. [2]. Following convention, the 10 core electrons were frozen; that is, they were not included in the CCSD(T) calculation. Thus, $n_o = 20$ and $n_v = 435$ in the $O(n_o^3 n_v^4)$ computational cost of CCSD(T) using the spin-free formalism.

D. *NWChem Performance Analysis*

In Figure 6 we present a performance comparison between NWChem running on GA using ARMCI-MPI and GA using ARMCI-Native. We show CCSD computation timings for all platforms and (T) timings for the InfiniBand cluster and XE6. For each system, we selected the best strided method using the 1 kB segments data presented in Figure 4: batched on Blue Gene and direct on InfiniBand, Cray XT, and Cray XE.

The InfiniBand target is the most aggressively tuned native implementation of ARMCI; it was written by ARMCI developers rather than system vendors and is well maintained. For this platform, there is a performance gap of roughly 2x for the CCSD and (T) calculations; however, this gap shrinks as the processor count is increased. On Blue Gene/P, we see that ARMCI-MPI's performance is comparable to the performance of the native implementation for the CCSD calculation and maintains good scaling. On the Cray XT, we see that performance is only 15%–20% less for ARMCI-MPI. On the Cray XE6 we see the impact of the increased communication performance observed in the communication benchmarks. On this system, ARMCI-MPI performs 30% better than the currently available native implementation on the CCSD calculation. ARMCI-MPI also scales much better and continues to improve execution time of the expensive (T) calculation on 5,952 processors while the native implementation's performance flattens for (T) and worsens for CCSD.

VIII. Discussion

Through this work, we have uncovered several opportunities to improve the portability and performance of GA and ARMCI as well as gaps in the current MPI standard that limit its performance and applicability as a PGAS runtime system.

A. *Extensions to GA and ARMCI*

We have made two extensions to the ARMCI API that will be included in the next GA/ARMCI software release: ARMCI direct local access (DLA) extensions and GA/ARMCI access modes. As discussed in Section V-C, DLA extensions are required for a correct implementation on top of MPI but will also help extend the GA and ARMCI models to weakly consistent and noncoherent platforms that are expected to become more prevalent as node-level architecture continues to grow larger and more complex. This extension requires changes to GA; however, it does not affect existing GA programs.

GA/ARMCI access modes are not required for correctness; they expose significant opportunities for performance optimization through application-level hints about how data will be accessed. Currently, as discussed in Section V-C, MPI exclusive mode epochs must be used to ensure correct behavior of ARMCI-MPI. In many cases, arrays will be used as read-only, accumulate-only, or will not use direct access during different phases of the program. These access patterns influence whether conflicting accesses are possible, and knowledge of the access mode can enable the use of more efficient shared locks within ARMCI-MPI. In native implementations of ARMCI, these access modes can also reduce the overheads from mechanisms used to ensure location consistency and allow ARMCI to better exploit native performance, for example by enabling adaptive routing when using relaxed consistency. In addition, they introduce the possibility for GA-level optimizations, such as caching or data replication.

B. *MPI RMA Status and Future Directions*

Most implementations of MPI-2 RMA that we have encountered are not as aggressively optimized for performance as one-sided libraries such as ARMCI. In addition, many are not well tested; in this project alone, we have worked with MPI developers to fix numerous MPI RMA bugs across a variety of MPI distributions. We hope that the open source release of our ARMCI implementation and its integration into GA will greatly increase the level of focus on testing and tuning current MPI RMA implementations.

Through this work, we have uncovered several characteristics of MPI-2 RMA that significantly limit performance. MPI's definition of conflicting accesses as fatal program errors significantly impedes our ability support communication concurrency. The need to encapsulate all communication within an epoch introduces the extra overhead of

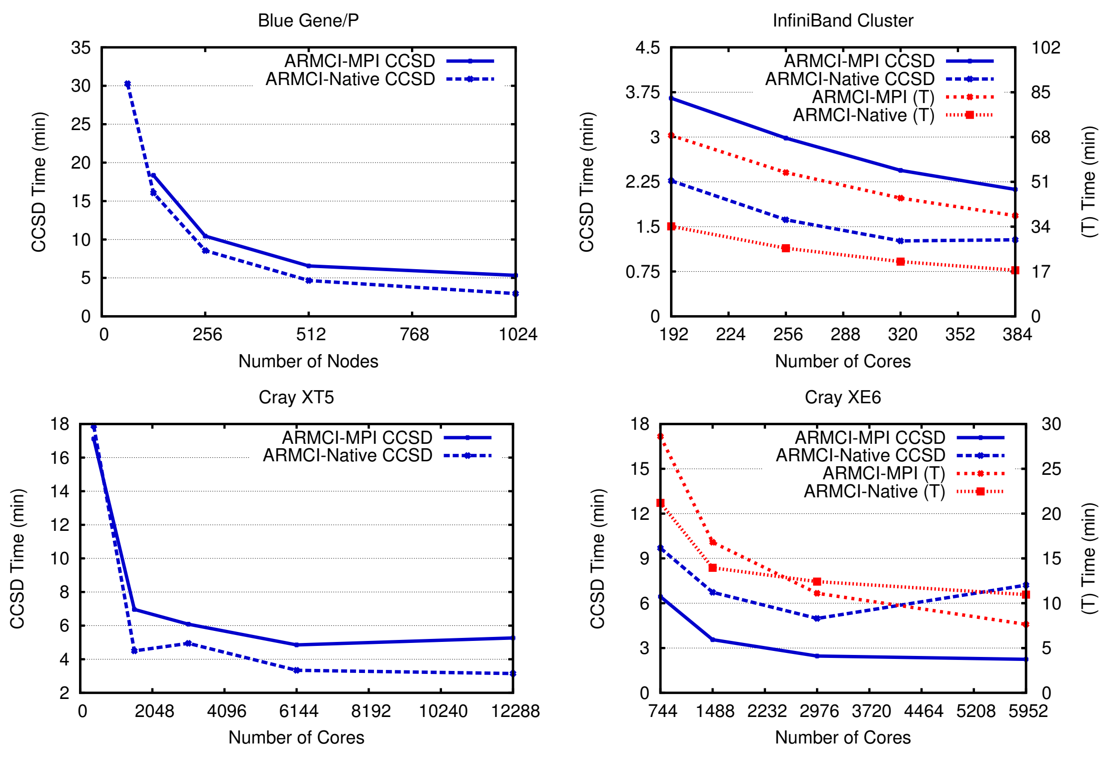


Figure 6. NWChem CCSD and (T) execution time for ARMCI-Native and ARMCI-MPI.

remote completion that is not required by ARMCI. The lack of persistent nonblocking operations has prevented us from providing efficient support for ARMCI's nonblocking operations that overlap communication with computation. Further, the lack of atomic read-modify-write operations has forced us to resort to a high-latency implementation using mutexes.

Currently, the MPI Forum is working to overhaul the MPI RMA functionality. We are excited to report that the MPI-3 RMA proposal introduces several important extensions to passive mode RMA that will address all the above challenges: (1) a better definition of how conflicting operations will interact with each other and relaxation from erroneous to undefined, (2) an epochless passive communication mode, (3) nonblocking request-based communication operations that allow overlap of computation and communication, and (4) new atomic read-modify-write operations.

IX. Related Work

Global Arrays [16] and, more specifically, its runtime system, ARMCI [15], have been implemented for a wide variety of high-performance architectures and interconnects [14], [17], [20]. However, this work is the first effort to utilize MPI's one-sided interface as an implementation vehicle for ARMCI. An implementation of ARMCI on MPI two-sided messaging has been included in the ARMCI distribution for several years. This implementation implements ARMCI's one-sided operations by running a data server process on each node. The data server maps shared memory that is shared with all processes on the node and services requests to read from and write to this data. However, this approach does not utilize MPI's one-sided functionality and has several overheads, including consumption of a core, bottlenecking on the data server, and two-sided messaging overheads such as tag matching. In comparison, the approach presented in this paper uses MPI RMA operations, which provide access to the RDMA hardware capabilities of modern high-performance interconnects.

GASNet [5] is a runtime system used by several PGAS models, including the Berkeley implementation of Unified Parallel C (UPC) [3]. GASNet also provides an MPI implementation that does not use MPI RMA because of a semantic mismatch between MPI and UPC that cannot be overcome at the runtime level [6]. Instead, GASNet-MPI is based on the AMMPI [4] active messages runtime system that uses MPI's two-sided communication.

MPI RMA has been standardized for over a decade, and significant effort has been invested in improving its performance [11], [18] and in building higher-level libraries using MPI RMA [12]. In spite of these efforts, adoption of MPI RMA by users has been slow. Nevertheless, MPI RMA has been demonstrated to be effective in several applications including earthquake modeling simulations [8] and cosmological simulations [19].

X. Conclusion

We have presented an implementation of the Global Arrays parallel programming model using MPI's RMA functionality. This implementation was achieved by porting GA's low-level ARMCI PGAS runtime system to MPI's one-sided API. The impact of this implementation is several-fold: enhanced portability for GA and GA applications, especially to new platforms where a production-capable native ARMCI implementation is not available; improved resource utilization and interoperability for applications using both GA and MPI; and an expanded user base to drive development of MPI one-sided features. Through an evaluation using the NWChem computational chemistry suite, we have demonstrated that performance is competitive with that of native implementations of ARMCI. Furthermore, we have investigated the effectiveness of the MPI one-sided model and provided important motivation to drive the inclusion of new features in the next version of the MPI standard.

Acknowledgement

This work was supported through resource grants from ALCF, Argonne LCRC, NCCS, and NERSC and by the U.S. Department of Energy under contracts DE-AC02-06CH11357, DE-AC05-00OR22725, and DE-ACO6-76RL01830.

References

[1] G. M. Adelson-Velskiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathmatics*, 3(5):1259–1262, 1962. Translated from Russian *Doklady, Akademiĭ Nauk SSSR* **146**:263–266.

[2] Edoardo Aprà, Alistair P. Rendell, Robert J. Harrison, Vinod Tipparaju, Wibe A. de Jong, and Sotiris S. Xantheas. Liquid water: obtaining the right answer for the right reasons. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–7, New York, NY, USA, 2009. ACM.

[3] Berkeley UPC. Berkeley UPC user's guide version 2.8.0, 2009.

[4] Dan Bonachea. AMMPI: Active messages over MPI. Website. http://www.cs.berkeley.edu/~bonachea/ammpi/.

[5] Dan Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1:91–99, August 2004.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[8] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D.K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling. Scalable Earthquake Simulation on Petascale Supercomputers. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, Nov 2010.

[9] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in mpi. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting*, EuroMPI '11, 2011.

[10] G.R. Gao and V. Sarkar. Location consistency – a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798 –813, August 2000.

[11] P. Lai, S. Sur, and D. K. Panda. Designing Truly One-Sided MPI-2 RMA Intra-node Communication on Multi-core Systems. In *International Supercomputing Conference (ISC)*, June 2010.

[12] Robert Latham, Robert Ross, and Rajeev Thakur. Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. *International Journal of High Performance Computing Applications*, 21(2):132–143, 2007.

[13] MPI Forum. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.

[14] Jarek Nieplocha, Edoardo Apra, Jialin Ju, and Vinod Tipparaju. One-sided communication on clusters with Myrinet. *Cluster Computing*, 6(2):115–124, 2003. TY - JOUR.

[15] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[16] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, 2006.

[17] Jarek Nieplocha, Vinod Tipparaju, and Manoj Krishnan. Optimizing strided remote memory access operations on the quadrics qsnetii network interconnect. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 28, Washington, DC, USA, 2005. IEEE Computer Society.

[18] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda. Natively Supporting True One-sided Communication in MPI on Multi-core Systems with InfiniBand. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Shanghai, China, May 18–21 2009.

[19] Thacker R. J., Pringle, G., Couchman H. M. P and Booth, S. HYDRA-MPI: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures. *High Performance Computing Systems and Applications*, 2003.

[20] V. Tipparaju, E. Apra, W. Yu, and J. Vetter. Enabling a highly-scalable global address space model for petascale computing. In *CF '10: Proceedings of the 7th ACM conference on Computing frontiers*, New York, NY, USA, 2010. ACM.

[21] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.