

Remote memory operations on Linux clusters: expressiveness and efficiency of current implementations

C. S. Guiang, A. Purkayastha & K. F. Milfeld
Texas Advanced Computing Center
The University of Texas at Austin
Austin, Texas USA

Abstract

For certain types of applications, the use of remote memory access (RMA) operations offers several advantages over message passing in terms of ease-of-use, applicability, and performance. The implementation of RMA on Linux clusters is currently limited to LAM-MPI's support of a subset of the MPI-2 one-sided communication, the ARMCI library, Global Arrays (GA) Toolkit, and GPShMEM. ARMCI is a portable communication library that supports RMA operations, including one-sided data transfer (for both contiguous and non-contiguous data formats), synchronization, and accumulate operations. This work will focus on the performance characteristics of the ARMCI one-sided communication operations versus the use of equivalent message passing calls. Early experiences with the use of the GA toolkit will also be reported, to provide insight on the feasibility of using an SMP-like programming interface on distributed memory systems.

1 Introduction

Remote memory access (RMA) is a communication model that offers different functionalities from those provided in shared memory programming and message passing. The use of one-sided communication exhibits some of the advantages of shared memory programming such as direct access to global data, without suffering from the limited scalability that characterizes SMP applications. Moreover, RMA has advantages over MPI on two important respects:

1. Communications overhead: MPI send/receive routines generate two-way communications traffic (“handshaking”) to ensure accuracy and coherency in data transfers that is often unnecessary. In some scientific applications (e.g., some finite difference and molecular dynamics codes), the communication workload involves accessing data that does not require explicit cooperation between the communicating processes (e.g., writes to non-overlapping locations in memory). Remote memory operations in ARMCI [1], GPShMEM [2] and the Global Arrays (GA) Toolkit [3] decouple the data transfer operation from handshaking and synchronization. There is no extra coordination required between processes in a one-sided put/get operation, so there is less of the enforced synchronization that is sometimes unnecessary. Additional calls to synchronization constructs may be included in the application as needed. In addition, there are known performance penalties resulting from the ordering of data transmission, introduced by the cooperative nature of message passing [4]. Some applications do not rely on the ordering of data delivery, and hence this feature of message passing is unnecessary.
2. API: The distributed data view in MPI is less intuitive to use in scientific applications that require access to a global array. In contrast, the Global Arrays API provides a global data view of the underlying distributed data that does not distinguish between accessing local versus remote array data.

Different versions of remote memory operations are included in several commercial implementations such as SHMEM for Cray and SGI platforms, and LAPI for IBM systems. The MPI-2 standard provides support for RMA using what are commonly known as one-sided communication operations, where “one-sided” refers to the use of a single function call for initiating data movement from the calling process to a remote one. In MPI-2, a process needs to set up a “memory window” to define the local memory available for RMA, as well as the group of processes which will access this data. Performing one-sided data transfer in MPI-2 is straightforward but has its drawbacks. The use of “active” targets in MPI-2 requires remote processes to call synchronization constructs in order to complete the data movement, which undermines the one-sided nature of RMA operations. Extra synchronization by the remote processes is not called for in “active target” communication, but writes to a target memory window are only allowed during “access epochs”. Only one processor at a time is allowed to perform put operations during an access epoch, resulting in a performance penalty as write access to the target memory window is effectively serialized.

Unfortunately, the implementation of MPI-2 one-sided communication is lacking on Linux platforms. To date, only the LAM-MPI [5] library contains MPI-2 RMA operations, but support is limited to “active target” RMA. Portable libraries such as ARMCI and GPShMEM provide the only alternative to MPI-2 for performing RMA operations on Linux clusters.

In this paper, we discuss the differences between message passing and RMA operations in terms of the expressiveness of the API, suitability to applications, and communication performance. A summary of the functionalities included in ARMCI, GPShMEM and Global Arrays is provided in the next section. Section 3 describes the performance measurements made on ARMCI RMA operations versus the equivalent MPI send/receive calls, as well as the transfer rate of array sections using ARMCI, MPI, and Global Arrays. Section 4 concludes this work with a summary of the experimental results, and advice on the performance benefits/disadvantages of using RMA operations as implemented in ARMCI and Global Arrays. Lastly, we include a list of interesting topics on RMA operations that warrant further study.

2 Current Implementations of RMA on Linux Clusters

The Global Arrays (GA) Toolkit provides support for RMA operations and an SMP-like development environment for applications on distributed computing systems. GA implements a global address space, which makes it possible for elements in distributed arrays to be accessed as if they were in shared memory. The details of actual data distribution are hidden, but are still within user control through utility functions included in the library. MPI and GA library calls can be combined in a single program, as GA is designed to complement rather than replace existing implementations of message passing libraries. The GA library consists of routines that implement one-sided, collective array operations, and utility and diagnostic operations (e.g., for querying available memory, data locality). As the name implies, GA implements a global address space for arrays of data, not scalars. Therefore, GA one-sided communications are designed to access arrays of data (double, integer, and double complex) of up to 7 dimensions, not arbitrary memory addresses.

One-sided operations in GA includes put, get, scatter, gather, and accumulate operations. These operations complete without involving the participation of remote processes. The GA API also frees the programmer from explicitly tracking the processes wherein a given array section resides. Collective array operations in GA may be applied to the entire array or patches of it, and consist of basic array operations (such as zeroing, scaling by a factor, or setting array elements to a given value), linear algebra operations (such as add, matrix multiplication, transpose), and interfaces to 3rd party software like scaLAPACK

and PeIGS. A new capability exists in GA for creating arrays with built-in ghost cells, as well as a single function call for updating ghost cells. This feature is extremely useful in solving partial differential equations using finite difference methods on a discretized, multidimensional grid.

GPSHMEM is a portable implementation of the Cray Research Inc. SHMEM, which is a one-sided communication library for Cray and SGI systems. Like GA, GPSHMEM implements a global address space for accessing distributed data, but GPSHMEM contains support for more data types. GPSHMEM operations are limited to symmetric data objects, objects for which local and remote addresses have a known relationship (e.g., arrays in common blocks). The library routines provide the following functionalities: 1) noncollective RMA operations including those for strided access and atomic operations; 2) collective operations; 3) parallel environment query functions. In addition to the supported Cray SHMEM routines, GPSHMEM has added block strided put/get operations for 2D arrays

All RMA operations in GA and GPSHMEM use the Aggregate Remote Memory Copy Interface (ARMCI) library. ARMCI is a portable, one-sided communication library. On Linux clusters, ARMCI relies on low-level network communication layers (e.g., Elan for Quadrics, GM for Myrinet) to implement remote memory copies, while utilizing the native MPI implementation for collective communication operations. Noncontiguous data transfers are supported in ARMCI, which facilitates access of sections of multidimensional arrays in GA and 2D arrays in GPSHMEM.

ARMCI contains support for the following RMA operations:

- data movement using put/get and update of a memory location using Accumulate
- synchronization operations using local and global fence, atomic read-modify-write, and
- utility functions for allocating/deallocating memory for data transfer

Unlike MPI-2, one-sided communication in ARMCI do not require any action to be taken by the remote processes for completion. In this sense, the data transfer operations in ARMCI are truly one-sided, whereas “active target” operations on MPI-2 are not. However, the completion of the ARMCI operations put and accumulate only signals that data has been copied out of the calling process’ local memory, and may occur before data is actually transferred to the remote process. In both these cases, the use of a global or local fence is warranted on the calling process to ensure arrival of the data to its destination. On the other hand, get does not complete until the remote data has been written to the calling process’ local memory.

The data transfer operations in ARMCI support the use of noncontiguous data, and comes in two versions, each of which utilizes a different format. The vector data format is more general, and includes the address of each block of contiguous data. The strided data format is an optimized version of the vector format, and is useful for transferring dense sections of multidimensional arrays. For the transfer of contiguous data, ARMCI put and get are implemented on top of the corresponding strided operations.

Although it is a low-level library, ARMCI can be used in applications that require one-sided communication. Unlike GA, ARMCI does not present a global view of data. Consequently, distributed data structures in ARMCI applications must be explicitly managed by the programmer. Some disadvantages of ARMCI include the absence of an API for Fortran (only the C interface is supported) as well as the restriction that access to remote memory is limited to data allocated with ARMCI_Malloc.

3 Performance Measurements

All of the experiments conducted in this study were performed on an IA64 cluster that consists of 16 dual-processor nodes interconnected with a Myrinet switch fabric. The MPI library employed in the experiments is MPICH-GM [6] version 1.2.1.7, and the MPI-2 put/get functions were taken from LAM-MPI 6.5.6. The latest versions of Global Arrays (3.2B) and ARMCI (1.0.6) were used to generate the results presented here. The GPShMEM library has not been fully ported to the IA64, and as a result was not included in our tests.

On-node performance of the ARMCI and MPI-2 remote memory operations were compared with MPI send/receive calls for the transfer of both contiguous and non-contiguous data, the latter consisting of n blocks of data (in this case a double precision word), separated by a stride of 2. We defined a strided datatype in MPI, and used this datatype in the point-to-point call instead of performing message passing calls on each contiguous chunk of data. The results of our pingpong test are shown in Figure 1.

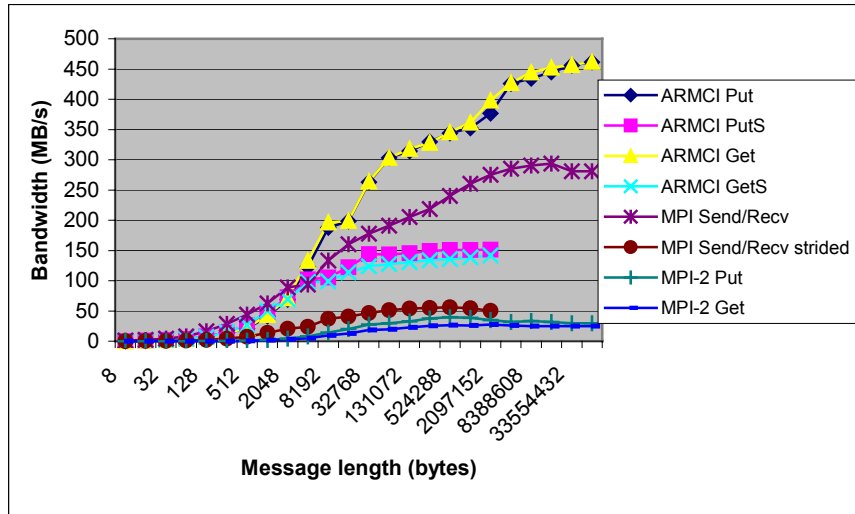


Figure 1: Measurement of pingpong bandwidth on a single IA64 node

In the case of both contiguous and noncontiguous data transfers, the ARMCI put and get operations outperform the corresponding MPI send/receive operations by a wide margin, except at message lengths at or below 2KB. This demonstrates ARMCI's efficient use of the underlying GM layer in performing the data transfers and its optimal use of the shared memory segment for communication. Moreover, the ARMCI put/get operations do not incur the extra overhead of synchronization involved in MPI send/receive. The pingpong bandwidths for ARMCI put and get are very close in value for contiguous data transfers. For the strided case, ARMCI put outperforms get by a slight margin. The MPI-2 get and put operations exhibit poor performance in the pingpong test, and were outperformed by the noncontiguous data transfers using ARMCI and MPI. This result is surprising, considering the fact that LAM-MPI contains support for the use of shared memory for on-node communication. However, one-sided communication in LAM-MPI is currently implemented over point-to-point communication. Due to LAM-MPI's limited support for the use of GM as the low-level messaging layer (all off-node communication uses TCP over Myrinet) and because of the inefficiency in the implementation of MPI-2 put/get, the MPI-2 one-sided operations were excluded from all other subsequent tests.

To measure both on- and off-node communication performance, 16 processes were placed in a ring topology, where each process sends to its two nearest neighbors. This "ring-passing" type of data movement is a simple prototype of some of the communication which occurs in applications like finite difference,

where neighboring grids exchange edge information. MD applications that employ decomposition methods based on the spatial location of particles and enforce a cutoff in the force calculation also display a similar communication pattern. The aggregate bandwidth for all 16 tasks versus message size is displayed in Figure 2, which shows the exchange of both contiguous and non-contiguous data:

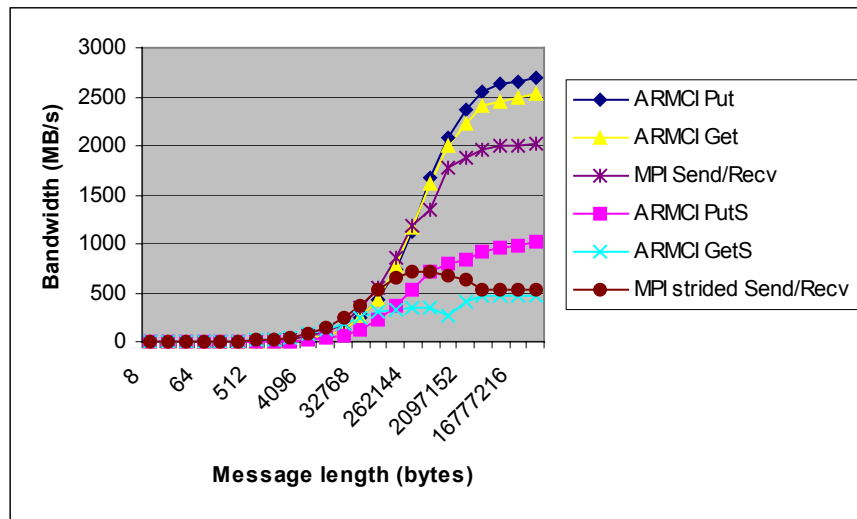


Figure 2: Aggregate bandwidth for a ring exchange of contiguous and non-contiguous data among 16 processes, two on each node.

MPI shows better performance than either ARMCI put or get for contiguous data smaller than 512 KB. At larger data sizes however, the bandwidth associated with ARMCI put and get is significantly higher. As with the pingpong test, the performance difference between ARMCI put vs. get is small. For the transfer of strided data (same format as the one used in the pingpong test), MPI outperforms ARMCI put for message sizes below 512 KB and ARMCI get for the entire range of the test. However, ARMCI get's bandwidth approaches asymptotically the same value as MPI send/receive for large message sizes.

Both ARMCI and MPICH-GM are designed to use the shared memory segment for on-node communication, as this provides greater bandwidth than sending data off the switch and back. To measure the efficiency of ARMCI one-sided operations without the support of shared memory communication on node, we ran the pingpong and ring-passing measurements on one processor per node. The results for both tests are presented in Figures 3 and 4.

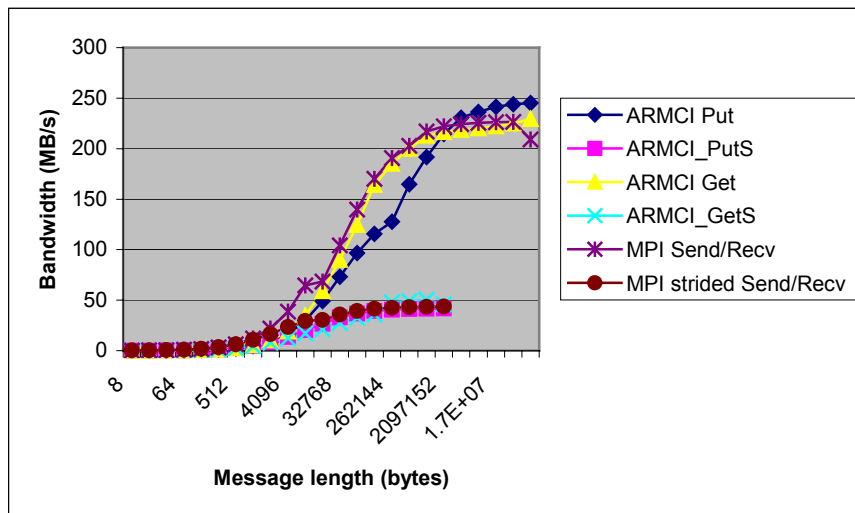


Figure 3: Measurement of pingpong bandwidth on two IA64 nodes

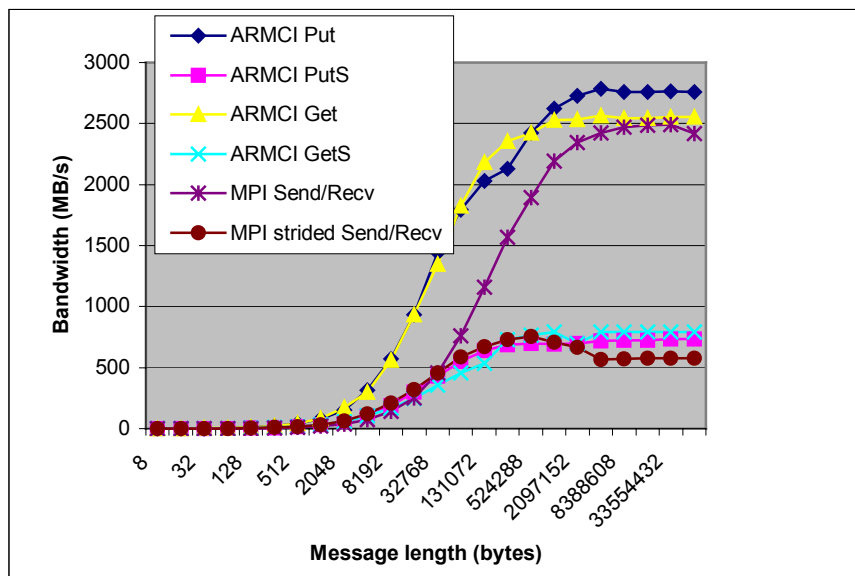


Figure 4: Aggregate bandwidth for a ring exchange of contiguous and non-contiguous data among 16 processes, one on each node.

In the case of contiguous data transfers, Figure 3 shows a peak pingpong bandwidth of 245 MB/s for ARMCI put, and 230 MB/s for both MPI send/rcv and ARMCI get. Both ARMCI and MPI attain node-to-node bandwidths that are close to the theoretical peak of 250 MB/s, although the bandwidth for ARMCI put is roughly 25% less than those from both MPI and ARMCI get for a wide range of data sizes. This simple experiment demonstrates that ARMCI makes efficient use of available bandwidth for remote memory operations, even without shared memory support for communication. For strided data transfers, the performance of ARMCI put/get is lower than MPI send/receive for message sizes below 2 MB.

The plot in Figure 4 shows that ARMCI put performance for transfer of contiguous data is not significantly affected by the use of a single process on each node. However, the aggregate bandwidth for ARMCI strided get almost doubles in the data range above 32 KB, while the performance of strided put degrades by about 25% at larger message sizes (> 256 MB). MPI exhibits noticeable performance improvement for exchange of contiguous data when all communication goes off to the switch fabric. The behavior of MPI send/receive for the transfer of strided data remains relatively unchanged relative to that in Figure 2.

In scientific and mathematical computing, it is often necessary to transfer sections of multidimensional arrays. This is most often seen in distributed array libraries which communicate array sections to solve problems in computational areas such as linear algebra or quantum chemistry. In the next two experiments, square and cubical sections of 2D and 3D arrays, respectively, were transferred between processors using the same scheme as described previously in the ring passing test. In the message passing version of our test code, we also used MPI datatypes to represent the array sections. Included in these measurements are rates obtained when using GA to move the array sections, as the GA API contains support for this task using the operations put and get on a global array. Figures 5 and 6 contain plots of the bandwidth obtained in the array transfer as the linear dimension of the array section is increased.

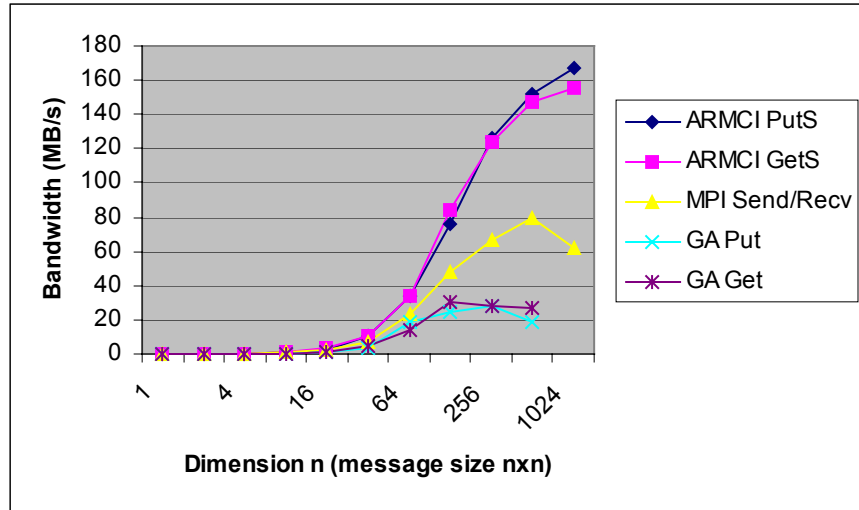


Figure 5: Bandwidth per processor for transfer of 2D array sections

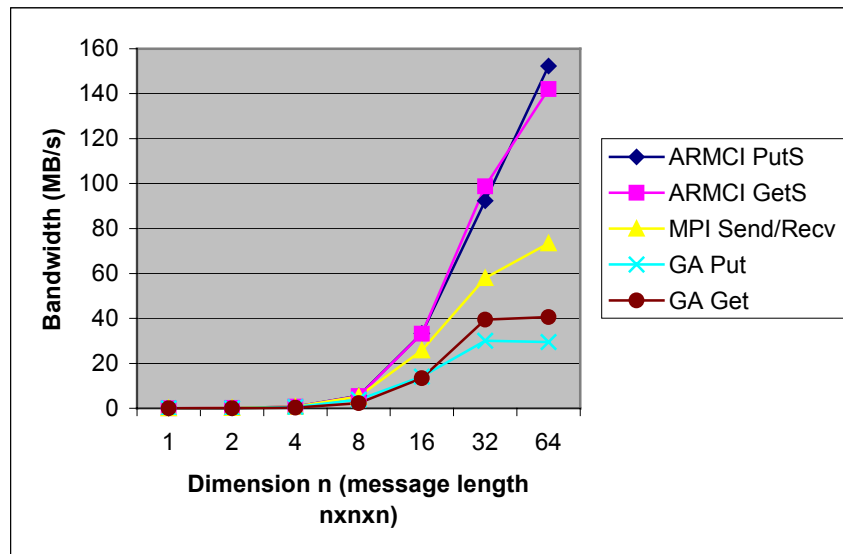


Figure 6: Bandwidth per processor for transfer of 3D array sections

For both 2D and 3D array sections, the ARMCI put and get operations exhibit superior performance to either MPI send/receive or GA put/get in the entire

range of linear dimensions considered in the experiment. GA put/get are the poorest performers, although its measured bandwidth per processor is comparable to that of MPI at small message sizes (< 32 KB for 2D sections and < 128 KB for 3D sections). At longer message sizes, GA Get performs the data transfer more efficiently than put in the transfer of 2D array sections, and outperforms put at the entire data range considered when moving 3D array sections. The gap in communication bandwidth between MPI and GA get decreases in going from 2D to 3D arrays.

Figures 7 and 8 show the bandwidth per processor vs. array size obtained for moving 2D and 3D array sections, respectively, when only one process is run per node. When shared memory support is turned off, ARMCI performance is largely unaffected whereas MPI exhibits large increases in the data transfer bandwidth (>60%) in the intermediate data range, for both 2D and 3D array transfers. The GA put and get operations are not significantly affected by turning off shared memory support. This is not surprising, as GA relies on ARMCI strided put and get for transferring noncontiguous data.

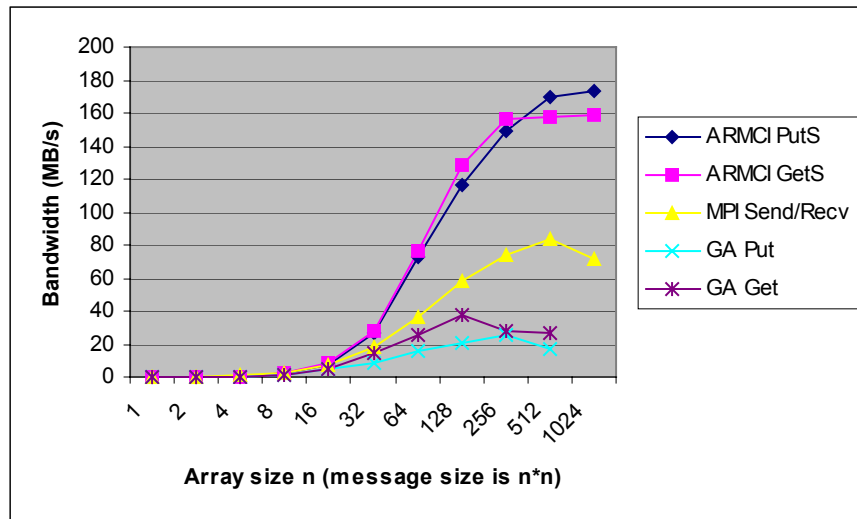


Figure 7: Bandwidth per processor for transfer of 2D array sections, run on a single process per node

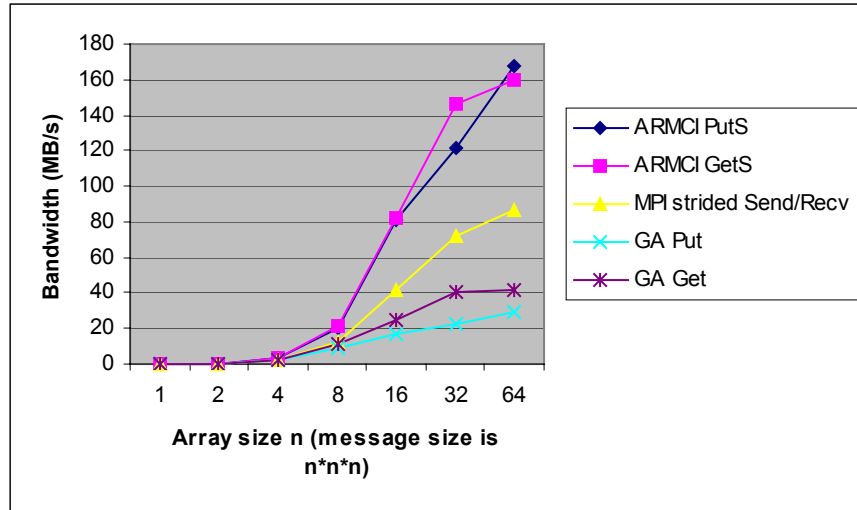


Figure 8: Bandwidth per processor for transfer of 3D array sections, run on a single process per node

Applications that involve dense, distributed arrays, and irregular communication that does not require synchronization among all participating processes are likely to benefit from using Global Arrays. A discretized representation of PDE problems, solved using a finite difference method, is one such example. Here, the entire problem domain can be represented most conveniently by a “shared”, global array in GA. In addition, the exchange of edge information can be accomplished more easily with GA’s support for one-sided communication of array data. A new capability was recently added to the Global Arrays toolkit for creating arrays with ghost cells around the visible data on each process, along with a function for updating the ghost cells owned by a processor with boundary information from its neighbors. To test the efficiency of the GA_Update_ghosts routine, three multidimensional arrays (dimensions ranging from 2-4) were created with GA_Create_ghosts, and distributed evenly among the processors. Only the communication intensive part (i.e. the call to GA_Update_ghosts) was timed during the iterative part of the code.

The MPI test code used dynamically allocated multidimensional arrays (dimensions ranging from 2-4) on each process to represent the domain, which was decomposed evenly among the processors. To simplify the transfer of ghost cells, the processes were arranged in a Cartesian topology, and each process sends messages to its $2 \cdot ndim$ neighbors. Ghost cells were defined as MPI datatypes to simplify communication and decrease the number of Sendrecv calls.

The result of timing measurements on the ghost cell exchange for the 2D, 3D, and 4D cases are shown in Figure 9.

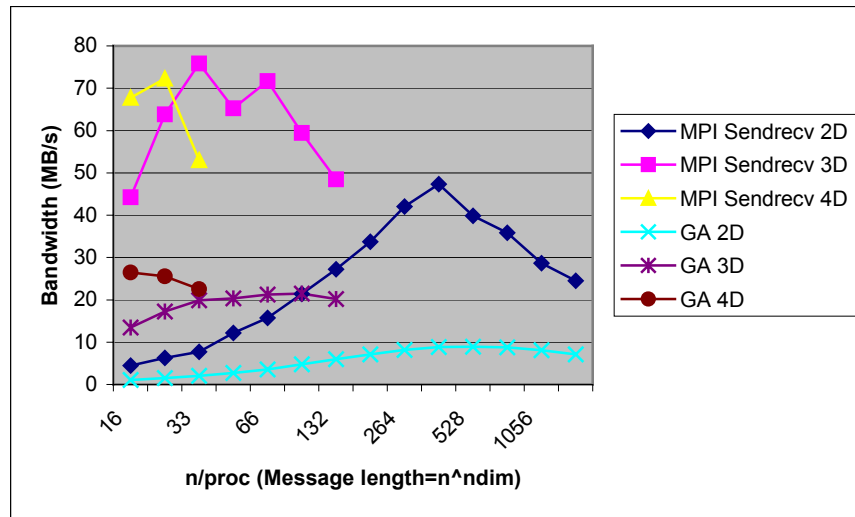


Figure 9: Performance of GA's ghost update routine versus ghost exchange using MPI Sendrecv calls. The measured bandwidth per process is plotted against the number of grid points n (along each dimension) owned by each process. The number of grid points per process is thus n^{ndim} , where $ndim$ is the number of dimensions of the domain.

The plot in Figure 10 shows the performance measurement of ghost cell exchange when run on a single processor per node.

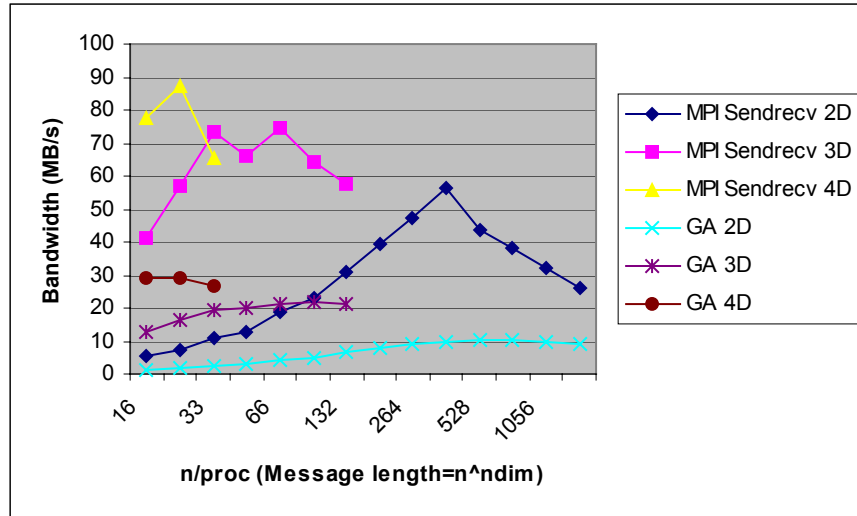


Figure 10: Performance of GA’s ghost update routine versus ghost exchange using MPI Sendrecv calls, run on a single processor per node.

Whether shared memory support is enabled or not, the GA implementation for updating ghost cells performs poorly compared to the message passing implementation across the entire range of data considered for this test. There is no clear trend showing whether efficiency of the GA ghost update function improves with increasing dimensionality of the domain, although the limited results presented here seem to show otherwise. Given the low bandwidth numbers obtained when transferring multi-dimensional array sections, it is not surprising that the ghost cell updates performed poorly because these operations rely on the efficiency of strided data transfers.

Setting aside the problematic issue of performance, the ghost cell capability in GA has an expressive and convenient semantics that has several advantages over the corresponding MPI implementation. The GA API itself has a global view of the entire domain, and consequently the user does not need to explicitly keep track of data locality, although this information is available when needed. Finite difference codes using GA “ghost” calls are easier to debug, as all array indices (whether they refer to the “visible” array or the array with ghost cells included) are under GA control. Thirdly, the exchange of ghost cells among processes is accomplished with a single call to `GA_Update_ghosts`, regardless of the dimensionality of the domain. This is not the case with the ghost cell exchange implementation using MPI, where the number of message passing calls increases linearly with $ndim$ ($2*ndim$). Defining MPI datatypes for the multidimensional

sections of the domain also becomes more progressively difficult and cumbersome as *ndim* increases.

4 Conclusion

The ARMCI library is a portable and highly efficient implementation of remote memory operations on Linux platforms. In tests of on-node and off-node communication performance, the ARMCI put operation provides higher data transfer throughput than the corresponding MPI calls. Although ARMCI is designed to take advantage of shared memory segments on SMP nodes, there is only a slight degradation in performance when ARMCI operations are run on a single process per node. ARMCI outperforms MPI by a large margin in noncontiguous data transfers involving sections of multidimensional arrays.

The Global Arrays toolkit provides a shared memory programming interface on distributed systems. The SMP-like API offers several advantages over a distributed memory programming model, such as a choice between a shared or distributed data view, simple access to data, and availability of data locality information through utility functions. In terms of performance, GA exhibited low communication throughput when transferring multidimensional array sections. Test finite difference codes which employ the “ghost cell” capability of GA likewise show poor performance relative to MPI. On the other hand, the use of this new capability greatly simplifies the development of finite difference codes especially as the dimensionality of the problem increases.

Suggestions for future work on remote memory operations include measurements of ARMCI and Global Arrays performance on large numbers of processors to determine how well these applications scale. The nonblocking nature of RMA operations like ARMCI lends itself well to the overlap of communication and computation. A study which explores the ways in which such an overlap can be efficiently executed using one-sided communication will be helpful in optimizing the performance of many scientific applications like MD and computational fluid dynamics.

References

- [1] <http://www.emsl.pnl.gov:2080/docs/parsoft/armci/>
- [2] Parzyszek, K., Nieplocha, J., and Kendall, R. A Generalized Portable SHMEM Library for High Performance Computing. *Proceedings of the Twelfth IASTED International Conference Parallel and Distributed Computing and Systems*, eds. M. Guizani & Z. Shen, Acta Press:Calgary, 401-406, 2000

[3] <http://www.emsl.pnl.gov:2080/docs/global/>

[4] Gropp, W., Lusk, E., & Thakur, R. *Using MPI-2 Advanced Features of the Message-Passing Interface*, MIT Press:Cambridge and London, pp. 119-127, 1999.

[5] <http://www.lam-mpi.org/>

[6] <http://www.myri.com/scs/>