ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems

Jarek Nieplocha[1] and Bryan Carpenter[2]

[1] Pacific Northwest National Laboratory,
Richland, WA 99352
j_nieplocha@pnl.gov

[2] Northeast Parallel Architectures Center,
Syracuse University
dbc@npac.syr.edu

Abstract. This paper introduces a new portable communication library called ARMCI. ARMCI provides one-sided communication capabilities for distributed array libraries and compiler run-time systems. It supports remote memory copy, accumulate, and synchronization operations optimized for non-contiguous data transfers including strided and generalized UNIX I/O vector interfaces. The library has been employed in the Global Arrays shared memory programming toolkit and Adlib, a Parallel Compiler Run-time Consortium run-time system.

1 Introduction

A portable lightweight remote memory copy is useful in parallel distributed-array libraries and compiler run-time systems. This functionality allows any process of a MIMD program executing in a distributed memory environment to copy data between local and remote memory. In contrast to the message-passing model used predominantly in these environments, remote memory copy does not require the explicit cooperation of the remote process whose memory is accessed. By decoupling synchronization between the process that needs the data and the process that owns the data from the actual data transfer, implementation of parallel algorithms that operate on distributed and irregular data structures can be greatly simplified and performance improved. In distributed array libraries, remote memory copy can be used for several purposes, incuding implementation of the object-based shared memory programming environment of Global Arrays [13] or ABC++ [8], and optimized implementation of collective operations on sections of multidimensional distributed arrrays, like those in the PCRC Adlib run-time system [4] or the P++ [15] object oriented class library.

In scientific computing, distributed array libraries communicate sections or fragments of sparse or dense multidimensional arrays. With remote copy APIs (application programming interfaces) supporting only contiguous blocks of data, array sections are typically decomposed into contiguous blocks of data and transferred in multiple communication operations. This approach is quite inefficient on systems with high latency networks, as the communication subsystem would handle each contiguous portion of the data as a separate network message. The communication startup costs are incurred multiple times rather than just once. The problem is attributable to an inadequate API, which does not pass enough information about the intended data transfer and layout of user data to the communication subsystem. In principle, if a more descriptive communication interface is used, there are are many ways communication libraries could optimize performance. For example: 1) minimize the number of underlying network packets by packing distinct blocks of data into as few packets as possible, 2) minimize the number of interrupts in interrupt-driven message-delivery systems, and 3) take advantage of any available shared memory optimizations (prefetching, poststoring, bulk data transfer) on shared memory systems. We illustrate the performance implication of communication interfaces with an example.

Consider a *get* operation that transfers a 2×100 section of a 10×300 Fortran array of double precision numbers on the IBM SP. The native remote memory copy operation *LAPI_Get* supports only contiguous data transfers [16, 9]. Therefore, 100 calls are required to transfer data in 16-byte chunks. With 90μs latency for the first column and 30μs for the next ones (pipelining), the transfer rate is 0.5MB/s. For the same amount of contiguous data, the rate is 30MB/s. If the application used a communication interface supporting strided data layout, a communication library could send one LAPI active message specifying the entire request. At the remote end the AM handler could copy data into a contiguous buffer and send all data in a single network packet. The data would then be copied to the destination. The achieved rate is 9MB/s. We argue that platform-specific optimizations such as these should be made available through a portable communication library, rather than being reproduced in each distributed array library or run-time system that requires strided memory copy.

In this paper, we describe design, implementation and experience with the ARMCI (Aggregate Remote Memory Copy Interface), a new portable remote memory copy library we have been developing for optimized communication in the context of distributed arrays. ARMCI aims to be fully portable and compatible with message-passing libraries such as MPI or PVM. Unlike most existing similar facilities, such as Cray SHMEM [1] or IBM LAPI [9], it focuses on the noncontiguous data transfers. ARMCI offers a simpler and lower-level model of one-sided communication than MPI-2 [12] (no epochs, windows, datatypes, Fortran-77 API, rather complicated progress rules, etc.) and targets a different user audience. In particular, ARMCI is meant to be used primarily by library implementors rather than application developers. Examples of libraries that ARMCI is aimed at include Global Arrays, P++/Overture [15, 2], and the PCRC Adlib run-time system. We will discuss applications of ARMCI in Global Arrays and Adlib as particular examples. Performance benefits from using ARMCI are presented by comparing its performance to that of the native remote memory copy operation on the IBM SP.

Table 1. ARMCI operations

Operation	Comment
ARMCI_Put, *ARMCI_PutV*, *ARMCI_PutS*	contiguous, vector and strided versions of put
ARMCI_Get, *ARMCI_GetV*, *ARMCI_GetS*	contiguous, vector and strided versions of get
ARMCI_Acc, *ARMCI_AccV*, *ARMCI_AccS*	contiguous, vector and strided versions of atomic accumulate
ARMCI_Fence	blocks until outstanding operations targeting specified process complete
ARMCI_AllFence	blocks until all outstanding operations issued by calling process complete
ARMCI_Rmw	atomic read-modify-write
ARMCI_Malloc	memory allocator, returns array of addresses for memory allocated by all processes
ARMCI_Free	free memory allocated by *ARMCI_Malloc*
ARMCI_Poll	polling operation, not required for progress

2 ARMCI Model and Implementation

2.1 ARMCI Operations

ARMCI supports three classes of operation

- data transfer operations including put, get and accumulate.
- synchronization operations—local and global fence and atomic read-modify-write, and
- utility operations for allocation and deallocation of memory (as a convenience to the user) and error handling.

The currently supported operations are listed in Table 1.

The data transfer operations are available with two noncontiguous data format:

1. *Generalized I/O vector.* This is the most general format intended for multiple sets of equally-sized data segments, moved between arbitrary local and remote memory locations. It extends the format used in the UNIX *readv/writev* operations by minimizing storage requirements in cases when multiple data segments have the same size. For example, operations that would map well to this format include scatter and gather. The format would also allow to transfer a triangular section of a 2-D array in one operation.

```
typedef struct {
  void *src_ptr_ar ;
  void *dst_ptr_ar ;
  int bytes ;
  int ptr_ar_len ;
} armci_giov_t ;
```

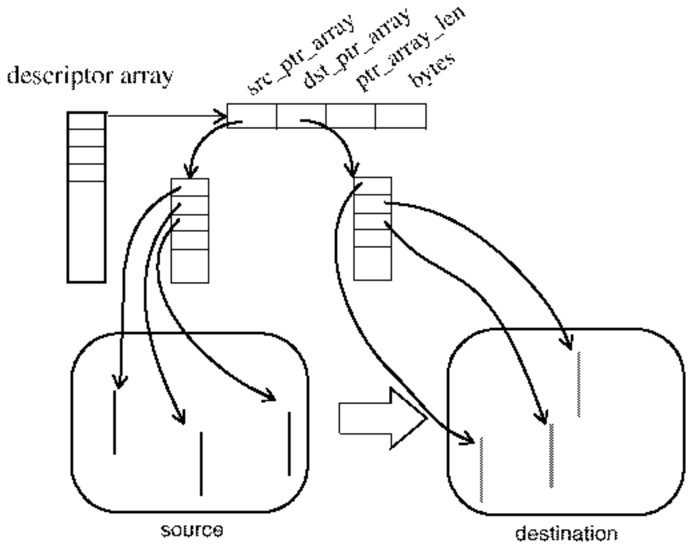

Fig. 1. Descriptor array for generalized I/O vector format. Each array element describes transfer of a set of equally-sized (*bytes*) blocks of data.

For example, with the generalized I/O vector format a *put* operation that copies data to the process(or) *proc*'s memory has the following interface:

```
int ARMCI_PutV(armci_giov_t dscr_arr[], int len, int proc)
```

The first argument is an array of size *arr_len*. Each array element specifies a set of equally-size segments of data copied from the local memory to the memory at the remote process(or) *proc*.

2. *Strided.* This format is an optimization of the generalized I/O vector format for sections of dense multidimensional arrays. Instead of including addresses for all the segments, it specifies for source and destination only the address of the first segment in the set. The addresses of the other segments are computing using the stride information:

```
void *src_ptr ;
void *dst_ptr ;
int stride_levels ;
int src_stride_arr[stride_levels] ;
int dst_stride_arr[stride_levels] ;
int count[stride_levels + 1] ;
```

For example, with the strided format a *put* operation that copies data to the process(or) *proc*'s memory has the following interface:

```
int ARMCI_PutS(src_ptr, src_stride_ar, dst_ptr,
               dst_stride_ar, count, stride_levels, proc)
```

The first argument is an array of size *arr_len*. Each array element specifies a set of equally-sized segments of data to be copied from the local memory to the memory at the remote process(or) *proc*, see Figure 2.

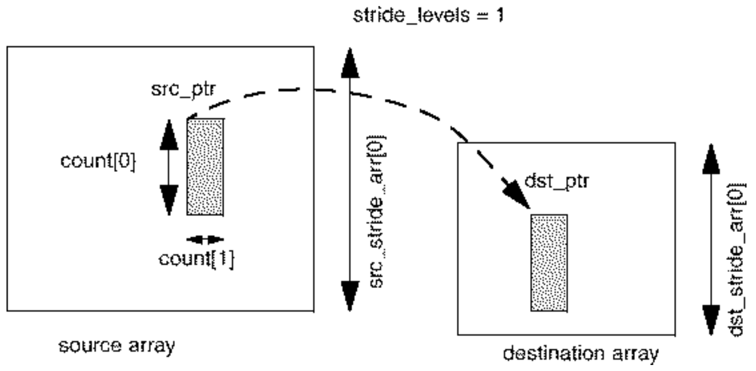

Fig. 2. Strided format for 2-dimensional array sections (assuming Fortran layout).

For contiguous data, *put* and *get* operations are available as simple macros on top of their strided counterparts.

2.2 Atomic Operations

At the present time, ARMCI offers two atomic operations: *accumulate* and *read-modify-write.*

Accumulate is an important operation in many scientific codes. It combines local and remote data atomically $x = x + a \times y$. For double precision data types, it can be thought of as an atomic version of the BLAS *DAXPY* subroutine with the x array located in remote memory. Unlike *MPI_Accumulate* in MPI-2, the ARMCI *accumulate* preserves the scale argument a available in *DAXPY.* For applications that need scaling, it has performance advantages on many modern microprocessors, where thanks to the *multiply-and-add* operation for floating point datatypes, the scaled addition is executed in the same time as the addition operation alone. The ARMCI *accumulate* is available for integer, double precision, complex and double complex datatypes.

ARMCI_Rmw is another atomic operation in the library. It can be used to implement synchronization operations or as a shared memory counter in simple dynamic load balancing applications. The operation updates atomically a remote integral variable according to the specified operator and returns the old value. There are two operators available: *fetch-and-increment* and *swap.*

2.3 Progress and ordering

It is important for a communication library such as ARMCI to have straightforward and uniform progress rules on all supported platforms. They simplify development and performance analysis of applications and free the user from dealing with ambiguities of platform-specific implementations. On all platforms, the ARMCI operations are truly one-sided and complete regardless of the actions taken by the remote process(or). In particular, there is no need for the remote process to make occasional communication calls or poll in order to assure that communication calls issued by other processes(ors) to this process(or) can complete. Although, for performance reasons, polling can be helpful (can avoid the cost of interrupt processing), it is not necessary to assure progress [10].

The ARMCI operations are ordered (complete in order they were issued) when referencing the same remote process(or). Operations issued to different processors can complete in an arbitrary order. Ordering simplifies the programming model and is required in many applications in computational chemistry (for example). Some systems allow ordering of otherwise unordered operations by providing a fence operation that blocks the calling process until the outstanding operation completes, so the next operation issued does not overtake it. This approach usually accomplishes more than applications might desire, and could have a negative performance impact on platforms where the copy operations are otherwise ordered. Usually, ordering can be accomplished with a lower overhead by using platform-specific means inside the communication library, rather than by a fence operation at the application level.

In ARMCI, when a *put* or *accumulate* operation completes, the data has been copied out of the calling process(or) memory but has not necessarily arrived at its destination. This is a local completion. A global completion of the outstanding put operations can be achieved by calling *ARMCI_Fence* or *ARMCI_AllFence*. *ARMCI_Fence* blocks the calling processor until all *put* operations issued to a specified remote process(or) complete at the destination. *ARMCI_AllFence* does the same for all outstanding *put* operations issued by the calling process(or), regardless of the destination.

2.4 Portability and implementation considerations

The ARMCI model specification does not describe or assume any particular implementation model (for example, threads). One of the primary design goals was to allow wide portability of the library. Another one was to allow an implementation to exploit the most efficient mechanisms available on a given platform, which might involve active messages [11], native put/get, shared memory, and/or threads.

For example, depending on whatever solution delivers the best performance, ARMCI *accumulate* might or might not be implemented using the "owner computes" rule. In particular, this rule is used on IBM SP where *accumulate* is executed inside the Active Message handler on the processor that owns the remote data whereas on the Cray T3E the requesting processor performs the calculation (by locking memory, copying data, accumulating, putting updated data back and unlocking) without interrupting the data owner.

On shared memory systems, ARMCI operations currently require the remote data to be located in shared memory. This restriction greatly simplifies the implementation and improves the performance by allowing the library to use simply an optimized memory copy rather than other more costly mechanisms for moving data between separate address spaces. This is acceptable in the distributed array libraries we considered since they are responsible for the memory allocation and can easily use *ARMCI_Malloc* (or other mechanisms such as using *mmap*, *shmget*, etc. directly) rather than the local memory allocation. However, the requirement can be lifted in future version of ARMCI if compelling reasons are identified.

At the present time, the library is implemented on top of existing native communication interfaces on:

- distributed-memory systems such as Cray T3E and IBM SP,
- shared memory systems such as SGI Origin and Cray J90, and Unix and Windows NT workstations.

On distributed memory platforms, ARMCI uses the native remote memory copy (SHMEM on Cray T3E and LAPI on IBM SP). In addition, on IBM SP LAPI active messages and threads are employed to optimize performance of noncontiguous data transfers for all but very small and very large messages. As an example, in Figure 3 we demonstrate the implementation of the *strided get* on the IBM SP. Implementations of *strided put* and *accumulate* are somewhat more complex due to the additional level of latency optimization for short active messages and mutual exclusion in *accumulate*. The implementation of vector operations is close to that of their strided counterparts on the IBM SP. Mutual exclusion necessary to implement atomic operations is implemented with the Pthread mutexes on the IBM SP and the atomic swap on the Cray T3E.

The ARMCI ports on shared memory systems are based on the shared memory operations (Unicos shared memory on the Cray J90, System V on other Unix systems, and memory mapped files on Windows NT) and optimized remote memory copies for noncontiguous data transfers. On most of the platforms, the best performance of the memory copy is achieved with the Fortran-77 compiler. Many implementations of the *memcpy* or *bcopy* operations in the standard C library are far from optimal even for well aligned contiguous data. Mutual exclusion is implemented using vendor-specific locks (on SGI, HP, Cray-J90), System V semaphores (Unix) or thread mutexes (NT).

The near-future porting plans for ARMCI include clusters of workstations, and Fujitsu distributed memory supercomputers on top of the Fujitsu MPlib low-level communication interface.

2.5 Performance

We demonstrate performance benefits of ARMCI by comparing it to performance of the native remote memory copy on the IBM SP in two contexts: 1) copying sections of 2-D arrays from remote to local memory and 2) gather operation for multiple double-precision elements. In both cases, we run the tests on four processors with processor 0 alternating its multiple requests between the other three processors which were sleeping at that time. This assured that the incoming requests were processed in the interrupt mode (worst case scenario). We performed these tests on a 512-node IBM SP at PNNL. The machine uses the

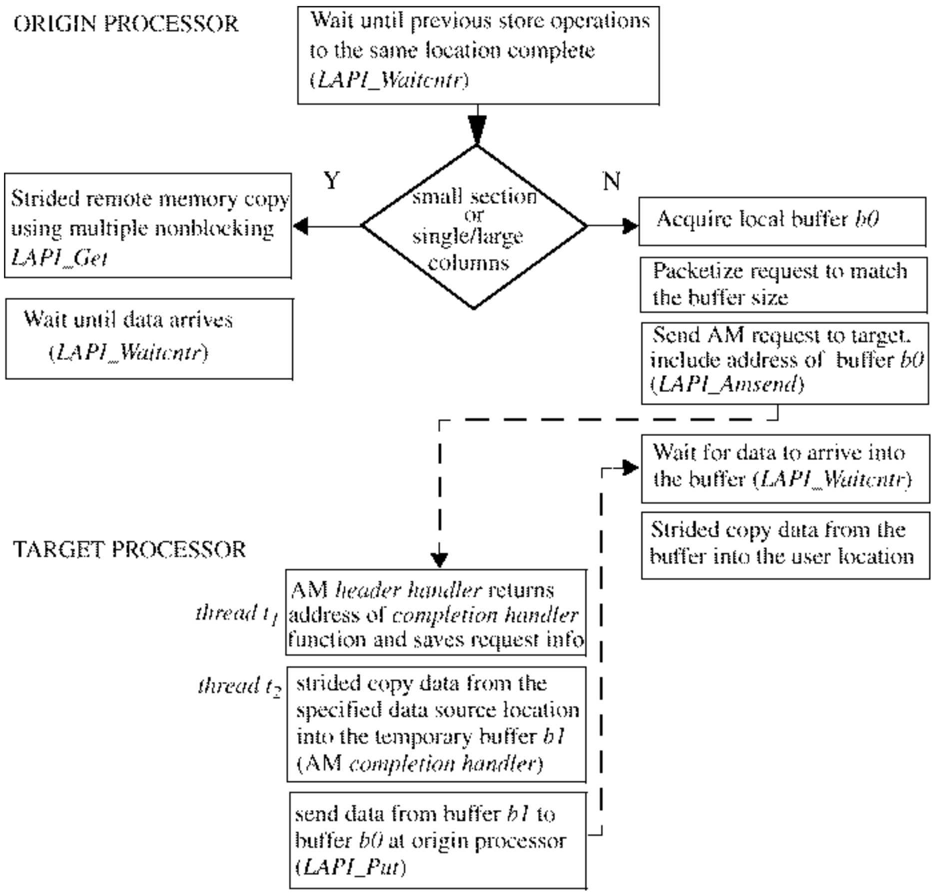

Fig. 3. Implementation of the ARMCI strided get operation on the IBM SP

120MHz Power2 Super processor and 512MB main memory on each uniprocessor node, and the IBM high performance switch (with the TB-3 adapter).

In the first benchmark, we transferred square sections of 2-dimensional arrays of double precision numbers from remote to local memory (get operation) and varied the array section dimension from 1 to 512. The array section was transferred using either a strided *ARMCI_GetS* operation or multiple *LAPI_Get* operations that transferred contiguous blocks of data—columns in the array section. The latter implementation uses nonblocking *LAPI_Get* operations which allows to issue requests for all columns in the section and then wait for all of them to complete. Had we used the blocking operation waiting for data in each column to arrive before issuing another *get* call the performance would have been much worse. Even with this optimization, Figure 4 shows that the ARMCI strided *get* operation has very significant performance advantages over the na-

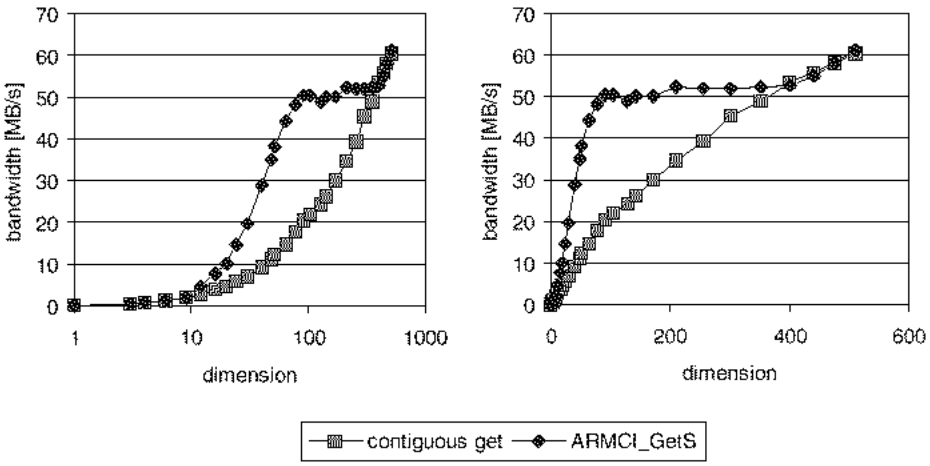

Fig. 4. The log-linear (left) and linear-linear (right) graphs represent bendwidths for a contiguous get operation and ARMCI *strided* get when moving square sections of a 2-D array.

tive contiguous *get* operation when transferring array sections. There are two exceptions where the performances of both approaches are identical:

- for small array sections where *ARMCI_GetS* uses *LAPI_Get* before switching to the Active Message (AM) protocol to optimize latency, and
- very large array sections where *ARMCI_GetS* switches again from the Active Message to *LAPI_Get* protocol to optimize bandwidth when the cost of the two extra memory copies makes the AM protocol less competitive, see Figure 3.

In Figure 5, performances of the *gather* operation for double precision elements is presented. In particular the *gather* operation in this test fetches every third element of an array on a remote processor. For comparision, the operation is implemented on top of the ARMCI vector *get* operation, *ARMCI_GetV*, and multiple native *get* operations. In this case too we used nonblocking *LAPI_Get* followed by a single wait operation to improve performance of this implementation. Despite this optimization, the performance of ARMCI is by far better for all but very small requests, for which the performances of both approaches are identical. For small requests *ARMCI_GetV* is implemented using multiple nonblocking *LAPI_Get* operaitons up to the point where the Active Message based protocol becomes more competitive. This protocol is very similar to that for the strided get shown in Figure 3. The performance of *gather* implemented on top of the nonblocking native *get* operation can be explained by the pipelining effect [16] for multiple requests in LAPI. Only the first *get* request issued to a particular remote processor is processed in the interrupt mode, the following ones are received when processing has not been completed. Regardless of pipelining, on eight bytes of user data is sent in each *LAPI_Get* message (single

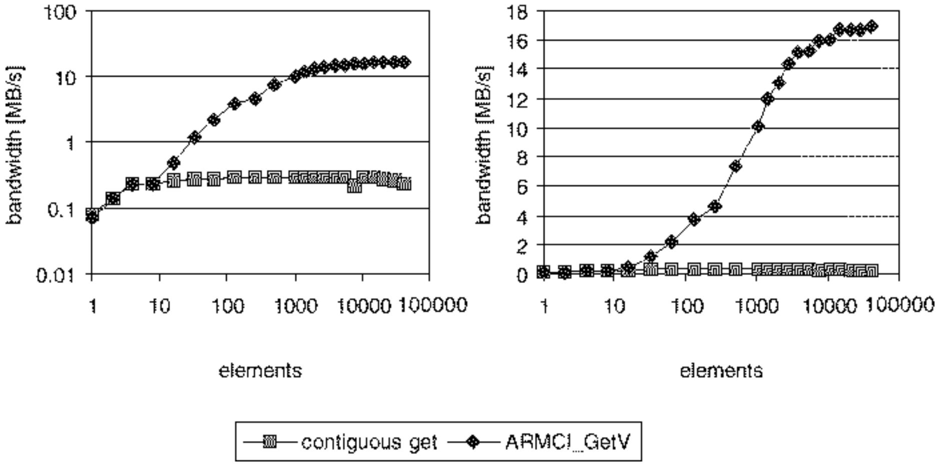

Fig. 5. The log-log (left) and linear-log (right) graphs represent bendwidth in gather operations for double precision numbers as a function of the number of elements when implemented with contiguous get operations and the ARMCI *vector* get.

network packet) whereas in the case of *ARMCI_GetV* all network packets sent by *LAPI_Put* inside the AM handler are filled with user data. With the 1024-byte packets in the IBM SP network, the network utilization and performance of the *gather* operation on top of the native *get* operation are far from optimal.

3 Experience with ARMCI

3.1 Global Arrays

The Global Arrays (GA) [13] toolkit provides a shared-memory programming model in the context of 2-dimensional distributed arrays. GA has been the primary programming model for numerous applications, some as big as 500,000 lines of code, in quantum chemistry, molecular dynamics, financial calculations and other areas. As part of the DoE-2000 Advanced Computational Testing and Simulation (ACTS) toolkit, GA is being extended to support higher dimensional-arrays. The toolkit was originally implemented directly on top of system-specific communication mechanisms (NX *hrecv*, MPL *rcvncall*, SHMEM, SGI *arena*, etc.). The original one-sided communication engine of GA had been closely tailored to the two-dimensional array semantics supported by the toolkit. This specificity hampered extensibility of the GA implementation. It became clear that a separation of the communication layer from the distributed array infrastructure is a much better approach. This was accomplished by restructuring the GA toolkit to use ARMCI hence making the GA implementation itself fully platform independent. ARMCI matches the GA model well and allows GA to support arbitrary dimensional arrays efficiently. The overhead introduced by using an additional software layer (ARMCI) is small; for example on the Cray

T3E *ga_get* latency has increased by less than 0.5μS when comparing with the original implementation of GA.

3.2 Adlib

The Adlib library was completed in the Parallel Compiler Runtime Consortium project [14] project. It is a high-level runtime library designed to support translation of data-parallel languages [4]. Initial emphasis was on High Performance Fortran, and two experimental HPF translators used the library to manage their communications [17, 11]. Currently the library is being used in the *HPspmd* project at NPAC [3]. It incorporates a built-in representation of a distributed array, and a library of communication and arithmetic operations acting on these arrays. The array model is more general than GA, supporting general HPF-like distribution formats, and arbitrary regular sections. The existing Adlib communication library emphasizes collective communication rather than one-sided communication.

The kernel Adlib library is implemented directly on top of MPI. We are reimplementing parts of the collective communication library on top of ARMCI and expect to see improved performance on shared memory platforms. The layout information must now be accompanied by a locally-held table of pointers containing base addresses at which array segments are stored in each peer process (in message-passing implementations, only the local segment address is needed). Adlib is implemented in C++, and for now the extra fields are added by using inheritance with the kernel distributed array descriptor class (DAD) as base class. A representative selection of the Adlib communication schedule classes were reimplemented in terms of ARMCI. These included the *Remap* class, which implements copying of regular sections between distributed arrays, the *Gather* class, which implements copying of a whole array from a source array indirectly subscripted by some other distributed arrays, and a few related classes.

The benchmark presented here is based on the *remap* operation. The particular example chosen abstracts the communication in the array assignment of the following HPF fragment

```
      real a(n, n), b(n, n)
!hpf$ distribute a(block, *) onto p
!hpf$ distribute b(*, block) onto p
      a = b
```

The destination array is distributed in its first dimension and the source array is distributed in its second dimension, so the assignment involves a data redistribution requiring an all-to-all communication. In practice the benchmark was coded directly in C++ (rather than Fortran) and run on four processors of an SGI Power Challenge. The timings for old and new versions are given in Figure 6. The initial implementation of the operation on top of ARMCI already shows a modest but significant 20% improvement over MPI for large arrays. For small arrays the MPI version was actually slightly faster due to extra barrier synchronizations used in the initial ARMCI implementation of *remap* (for now

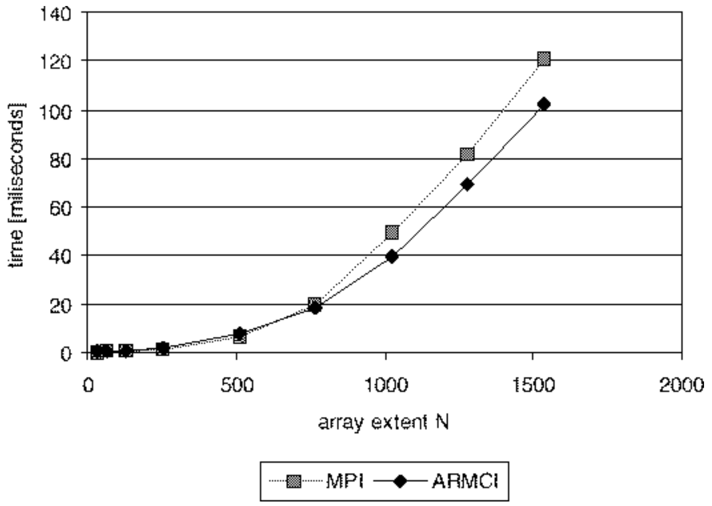

Fig. 6. Timings for original MPI vs new ARMCI implementation of *remap*. The operation is a particular redistribution of an N by N array.

these synchronizations were implemented naively in terms of message-passing). The MPI implemenation is MPICH using the shared memory device, so the underlying transport is the same in both cases.

In addition to improved performance of collective operations on shared memory platforms, adopting ARMCI as a component of the Adlib implementation will enable the next release of the library to incorporate one-sided *get* and *put* operations as part of its API—a notable gap in the existing functionality. In a slightly orthogonal but complementary development, we have produced a PCRC version of GA that internally uses a PCRC/Adlib array descriptor to maintain the global array distribution parameters. This provides a GA style programming model, but uses the Adlib array descriptor internally. The modularity of the GA implementation, enhanced by the introduction of the ARMCI layer, made changing the internal array descriptor a relatively straightforward task. An immediate benefit is that direct calls to the optimized Adlib collective communication library will be possible from the modified GA. This work was undertaken as part of an effort towards converging the two libraries.

4 Relation to Other Work

ARMCI differs from other systems in several key aspects. Unlike facilities that primarily support remote memory copies with contiguous data (for example SHMEM, LAPI, or the Active Message run-time-system for Split-C, which includes put/get operations [7]) it attempts to optimize noncontigous data transfers, such as those involving sections of multidimensional arrays, and generalized scatter/gather operations. The Cray SHMEM library [1] available on the Cray and SGI platforms, and on clusters of PCs through HPVM [5], supports only put/get operations for contiguous data and scatter/gather operations for the basic datatype elements.

The Virtual Interface Architecture (VIA) [6] is a recent network architecture and interface standard that is relevant to ARMCI as a possible implementation target. VIA offers support for noncontiguous data transfers. However, its scatter/gather Remote Direct Memory Access (RDMA) operations can transfer data only between noncontiguous locations in local memory and a contiguous remote memory area. For transfer of data between sections of local and remote multidimensional arrays, the VIA would require an additional copy to and from an intermediate contiguous buffer, which obviously has an impact on performance. This is unnecessary in ARMCI. ARMCI and VIA offer similar ordering on communication operations.

The MPI-2 one-sided communication specifications include remote memory copy operations such as *put* and *get* [12]. Noncontiguous data transfers are fully supported through the MPI derived data types. There are two models of one-sided communication in MPI-2: "active-target" and "passive-target". The MPI-2 one-sided communication and in particular its "active-target" version have been derived from message-passing, and its semantics include rather restrictive (for a remote memory copy) progress rules closer to MPI-1 than to existing remote memory copy interfaces like Cray SHMEM, IBM LAPI or Fujitsu MPlib. A version of MPI-2 one-sided communication called "passive-target" offers more relaxed progress rules and a simpler to use model than "active-target". However, it also introduces potential performance penalties by requiring locking before access to the remote memory ("window") and forbidding concurrent accesses to non-overlapping locations in a "window". ARMCI does not have similar restrictions and offers a simpler programming model that MPI-2. This makes it more appropriate for libraries and tools supporting some application domains, including computational chemistry.

5 Conclusions and Future Work

We introduced a new portable communication library targetting parallel distributed array libraries and compiler run-time systems. By supporting communication interfaces for noncontiguous data transfers ARMCI offers potential performance advantages over existing systems for communication patterns used in many scientific applications. We presented its performance for the strided get and gather-type data transfers on the IBM SP where the ARMCI implementation outperformed by a large margin the native remote memory copy operations. ARMCI already demonstrated its value: we employed ARMCI to implement the Global Arrays library and it new higher-dimensional array capabilities, and used it to optimize collective communication in the Adlib run-time system. The porting and tuning efforts will be continued. The future ports will include networks of workstations and other scalable systems. In addition, future extensions of the existing functionality will include nonblocking interfaces to *get* oeprations and new operations such as locks.

Acknowledgements

This work was supported through the U.S. Department of Energy by the Mathematical, Information, and Computational Science Division the Office of Computational and Technology Research DOE 2000 ACTS project, and used the Molecular Science Computing Facility in the Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

References

1. R. Barriuso and Allan Knies. *SHMEM User's Guide. SN-2516.* Cray Research Inc, 1994.
2. F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan. OVERTURE: An object-oriented framework for high performance scientific computing. In *Proceedings of SC98: High Performance Networking and Computing.* IEEE Computer Society, 1998.
3. B. Carpenter, G. Fox, D. Leskiw, X. Li, Y. Wen, and G. Zhang. Language bindings for a data-parallel runtime. In Michael Gerndt and Hermann Hellwagner, editors, *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments.* IEEE Computer Society Press, 1998.
4. B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. http://www.npac.syr.edu/projects/pcrc/kernel.html.
5. A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High performance virtual machines HPVM: Clusters with supercomputing APIs and performance. In *Proceedings 8th SIAM Conference on Parallel Processing in Scientific Computing*, 1997.
6. Compaq Computer Corp., Intel Corp., and Microsoft Corp. Virtual interface architecture specification, December 1997.
7. D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93.* IEEE Computer Society Press, 1993.
8. F.C. Eigler, W. Farell, S.D. Pullara, and G.V. Wilson. ABC++. In G. Wilson and P. Lu, editors, *Parallel Programming using C++.* MIT Press, 1996.
9. IBM Corp. Understanding and using the communication Low-level Application Programming Interface (LAPI). In *IBM Parallel System Support Programs for AIX Administration Guide, GC23-3897-04.* 1997. (Available at http://ppdbooks.pok.ibm.com:80/cgi-bin/bookmgr/bookmgr.cmd/BOOKS/sspad230/9.1).
10. K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts and thread management. *Proc. Frontiers*, 96:13–22, 1996.
11. J. Merlin, B. Carpenter, and A. Hey. shpf: a subset High Performance Fortran compilation system. *Fortran Journal*, pages 2–6, March 1996.

12. MPI Forum. MPI-2: Extension to message passing interface. Technical report, University of Tennessee, July 1997.
13. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10:197–220, 1996.
14. Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.
15. D. Quinlan and R. Parsons. A++/P++ array classes for architecture independent finite difference calculations. In *Proceedings of the Second Annual Object Oriented Numerics Conference*, April 1994.
16. G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R.K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with LAPI: a new high-performance communication library for the IBM RS/6000 SP. In *Proceedings of the International Parallel Processing Symposium, IPPS '98*, pages 260–266, 1998.
17. G. Zhang, B. Carpenter, G. Fox, X. Li, X. Li, and Y. Wen. PCRC-based HPF compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997.