Symmetric Data Objects and Remote Memory Access Communication for Fortran 95 Applications

J. Nieplocha D. Baxter V. Tipparaju

Pacific Northwest National Laboratory

C. Rassmunsen

Los Alamos National Laboratory

Robert W. Numrich

Minnesota Super Computing Institute
University of Minnesota,
Minneapoilis, MN.

Abstract. Symmetric data objects have been introduced by Cray Inc. in context of SHMEM remote memory access communication on Cray T3D/E systems and later adopted by SGI for their Origin servers. Symmetric data objects greatly simplify parallel programming by allowing programmers to reference remote instance of a data structure by specifying address of the local counterpart. The current paper describes how symmetric data objects and remote memory access communication could be implemented in Fortran 95 without requiring specialized hardware or compiler support. NAS Multi-Grid parallel benchmark was used as an application example and demonstrated competitive performance to the standard MPI implementation.

1. Introduction

Fortran is an integral part of the computing environment at major scientific institutions. It is often the language of choice for developing applications that model complex physical, chemical, and biological systems. In addition, Fortran is an evolving language [1]. The Fortran 90/95 standard introduced many new constructs, including derived-data types, new array features and operations, pointers, increased support for code modularization, and enhanced type safety. These features are advantageous to scientific applications and improve the programmer productivity.

Remote memory access (RMA) operations facilitate an intermediate programming model between message passing and shared memory. This model combines some advantages of shared memory, such as direct access to shared/global data, and the message-passing model, namely the control over locality and data distribution. Certain types of shared memory applications can be implemented using this approach. In some other cases, remote memory operations can be used as a high-performance

alternative to message passing. On many modern platforms, RMA is directly supported by hardware and is the lowest-level and often most efficient communication paradigm available. The RMA has been offered in numerous portable interfaces ranging from SHMEM [10, 11, 16], ARMCI [12], and MPI-2. Among these, Cray SHMEM has been the most widely used interface and offered by hardware vendors such as Cray, IBM, HP for their architectures. Some important characteristics of SHMEM are the ease of use, simplicity, demonstrated potential for achieving high performance.

One of the important characteristics of SHMEM is support for *symmetric data objects*. This concept allows the programmer to access remote instances of data structures through references to the local instance. In particular, the programmer is not required to keep track of addresses on remote processors as mandated by other RMA models such as LAPI[15] on the IBM SP where addresses for remote instances of the same data object can be different and thus need to be exchanged and stored on all processors. Implementation of symmetric data objects is difficult without hardware and/or OS assistance on clustered systems. This is because the virtual memory addresses allocated by the operating system for storing instances of the same data structure in a cluster can be different across the machine. Without symmetric data objects, the programmer would be required to store $O(P^2)$ addresses on the machine. In addition, the overall programming model is harder to use and the application codes become more error prone.

In this paper, we take advantage of the new Fortran 95 features to provide high-level interfaces to one-sided operations on multidimensional arrays consistent with symmetric data-object model of SHMEM. This work is inspired by Co-Array Fortran [14] with its ability to reference arbitrary sections of so called *co-arrays* using high-level array assignments. *Co-arrays* represent a special type of Fortran 95 arrays defined on all tasks in the SPMD program. The main contributions of this paper are: 1) definition of an interface that support important features of SHMEM and CAF using a library- rather than compiler-based approach: symmetric data objects of SHMEM and one-sided high-level access to multidimensional arrays that CAF offers (SHMEM does not offer such ability), 2) a description of a portable implementation of these features that do not require hardware or OS support, and 3) demonstration that the proposed approach can deliver high performance, both in context of microbenchmarks as well as the NAS NPB Multigrid (MG) benchmark [4].

The remainder of the paper is organized as follows. Section 2 describes the proposed interface and discusses its characteristics. Section 3 describes the implementation based on Chasm and the ARMCI one-sided communication library. Section 4 reports experimental results on the Linux cluster with Myrinet that demonstrate that our implementation outperforms the NAS NPB version of MG.

2. Proposed Approach

We propose to support symmetric data objects and RMA for Fortran 95 applications based on Fortran 95 array pointers with special memory allocation interface and a set of remote memory access communication interfaces handling slices

(sections) of Fortran 95 arrays. These interfaces allow users to allocate/free multidimensional Fortran 95 arrays and to communicate data held in this memory using simple get/put semantics. In addition, the reference to the remote instance of arrays does not require users to keep track of addresses on remote note. A single Fortran 95 pointer is used to represent local and remote instances of a multidimensional array. A unique feature of these interfaces is that they allow users to take full advantage of Fortran 95 array mechanisms (like array-valued expressions).

The Fortran interfaces are as follows. Memory allocation is done with calls to the generic interfaces `Malloc_fa` and `Free_fa` (shown below for real, two-dimensional arrays only),

```
module Mem_F95

  interface Malloc_fa
    subroutine Malloc_2DR(a, lb, ub, rc)
      real, pointer :: a(:,:)
      integer, intent(in) :: lb(2), ub(2)
      integer, intent(out) :: rc
    end subroutine Malloc_2DR
  end interface

  interface Free_fa
    subroutine Free_2DR(a, rc)
      real, pointer :: a(:,:)
      integer, intent(out) :: rc
    end subroutine Free_2DR
  end interface

end module Mem_F95
```

In the above, the arrays `lb` and `ub` contain the lower and upper bounds of the array to be allocated and the parameter `rc` is an error code.

Similarly, the generic interfaces for RMA communication are `Put_fa` and `Get_fa`. To save space, here we only present the interface to the first one for the double precision two dimensional arrays (the get interface is similar).

```
module Types_fa
type Slice_fa
    integer :: lo(7)
    integer :: hi(7)
    integer :: stride(7)
  end type Slice
end module Types_fa

module Mov_F95
  interface Put_fa
    subroutine Put_2DR(src, src_slc,
                       dst, dst_slc, proc, rc)
      use Types_fa
      real, pointer :: src(:,:), dst(:,:)
```

```
        type(Slice_fa), intent(in) :: src_slc, dst_slc
        integer, intent(in) :: proc, rank
        integer, intent(out) :: rc
      end subroutine Put_2DR
    end interface
```

In the communication interfaces, `src` and `dst` are the source and destination arrays respectively, `src_slc` and `dst_slc` contain information about the memory portion (array section) of the source and destination arrays to be used, `proc` is the processor number of the destination array, and `rc` is an error return code. In addition to being able access sections of multidimensional arrays, to be consistent with the Fortran 95 capabilities for arrays, the user can also specify stride information.

The current implementation supports integer, floating, and complex data types of the 8- and 4-byte kinds. Array dimensions ranging from one to seven (Fortran limit) are handled. By exploiting Fortran 95 function name overloading, we can use a single name for the put operation `Put_fa` to handles all data types and array dimensions using a single interface. The defined memory allocation interfaces defined above are mandatory for allocating memory to be accessed remotely from other processors. However, they are not required for local arrays: source of data in *put*, and destination in *get*.

The semantics of the RMA operations (progress, ordering) follow closely that of the Cray SHMEM. In order to provide the application programmer with abilities to hide latency, we introduced nonblocking interfaces to put/get calls. A nonblocking call returns before the user buffer can be accessed and requires a special *wait* function to complete. This feature is not available in SHMEM. (Although the CAF standard does not offer this capability, the Rice CAF compiler adds directives that change semantics of the array assignments to nonblocking in so called non-blocking regions.)

3. Implementation

Unfortunately, the Fortran 95 standard *alone* does not provide sufficient capabilities to implement the memory management required to support symmetric data objects. However, this is made possible by the use of the Chasm array-descriptor library [9]. In addition, we use the ARMCI portable RMA library to handle communication. Our approach also relies on MPI for job startup and control. In fact, the user can use the interfaces described in the previous section in the MPI programs and take advantage of the full capabilities of MPI e.g., collective operations.

Chasm

Chasm [12, 3] is language transformation system providing language interoperability between Fortran and C/C++. Language interoperability is provided by stub and skeleton interfaces. This code is generated by language transformation programs taking as input existing user C, C++ or Fortran source code and generating the stub and skeleton interfaces to the input code as output [3].

One of the challenges of language interoperability with Fortran is that Fortran assumed-shape array arguments are passed by an array descriptor, rather than as a simple memory address. Array descriptors contain meta data about the array, including the base address of the array, the lower and upper bounds for each dimension of the array, and *sometimes*, the rank of the array and the type of an array element. The key point is that the format of the array descriptor is not specified by the language standard, but is left to be specified by the vendor of the Fortran compiler. Chasm provides generic C interfaces to the Fortran, vendor-specific array descriptors. Without the Chasm array-descriptor library, there would be no way to call the ARMCI library from Fortran and allocate Fortran 95 arrays using the special ARMCI memory necessary for remote communication.

It should be noted that the need for the Chasm array-descriptor library will be reduced somewhat once compiler vendors have implemented the Fortran 2003 standard [8]. Fortran 2003 contains standard mechanisms for interoperating with C that allow Fortran array pointers to be associated with memory allocated from C. In addition, a modified version of the Chasm, array-descriptor interface has been accepted by the Fortran J3 committee for possible inclusion in the next Fortran standard. This would then allow the Fortran interfaces, introduced in the previous section, to be used in a language standard way, with no additional stub or skeleton code needed. Until this time, either Chasm or Fortran 2003 compilers (with slightly modified Fortran stub code) will be needed.

ARMCI

The Aggregate Remote Memory Copy Interface (ARMCI) [6] is a portable RMA communication library. It has been used for implementing distributed array libraries such as Global Arrays, other communication libraries such as Generalized Portable SHMEM [10], and compiler run-time systems such as PCRC Adlib [13] or the portable Co-Array Fortran compiler at Rice University [5]. ARMCI offers an extensive set of functionality in the area of RMA communication: 1) data transfer operations; 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. In scientific computing, applications often require transfers of noncontiguous data that corresponds to fragments of multidimensional arrays, sparse matrices, or other more complex data structures. With remote memory communication APIs that support only contiguous data transfers, it is necessary to transfer noncontiguous data using multiple communication operations. This often leads to inefficient network utilization and involves increased overhead. ARMCI, however, offers explicit noncontiguous data interfaces: strided and generalized I/O vector that allow description of the data layout so that it could, in principle, be transferred in a single message. Of course, the effectiveness of actual transfers depends on the ability of underlying networks to deal with noncontiguous data (e.g., scatter/gather operations). However, even when scatter/gather operations are not supported by the network, the ARMCI strided and vector operations take advantage of the information -- for example, at the level of data packing/unpacking -- so that the overall number of messages and network packets is reduced. The strided interfaces are important for Fortran 95 applications that use multidimensional arrays.

Fortran 95 Interfaces

The C side of the implementation is composed of 10 functions. Four of the functions are administrative functions for initializing Fortran array descriptor information, cleaning up, terminating, and synchronizing that take no arguments. The other six are functions for allocating Fortran 95 arrays that the ARMCI data movement routines can handle, blocking *put* and *get* operations for the data movement, their nonblocking analogs, and a function to free the allocated deferred shape arrays.

These functions assume the following Fortran calling convention. When calling routines with the deferred shape arrays (allocatable) as arguments, each deferred shape array argument contributes two addresses to the actual argument list. The first is the data address of the first element of the array and is in order specified in the arguments of the Fortran call/function reference. The second is the address of the dope vector describing the deferred shape array and is placed after the end of the arguments listed in the Fortran call or function reference. Routines with more than one deferred shape array have all of the addresses of the dope vectors concatenated at the end of the argument list appearing in the same relative order as the corresponding deferred shape array in the Fortran argument list. To support symmetric data objects even on clusters with virtual memory nodes, we allocate extra array memory (in addition to the user specific portion) to store array pointers on the remote nodes. When user specifies pointer to the local instance of the Fortran 95 array, we access the appropriate pointer for the specified processor and pass the required information to ARMCI put/get calls.

On the Fortran 95 side of the interface there are corresponding routines to allocate, put, get (blocking/nonblocking) and free array memory. Fortran 95 does not have the notion of a generic pointer type, the equivalent of a (void *) in C. Each pointer in fortran must point to an array of specified type and dimension (number of indices used to reference elements in the array). Module procedures are used to overload the C functionality of void *, giving a similar interface on the Fortran side where the user does not have to use a different function name for using ARMCI routines on different data types. Six types of elementary data are supported for one to seven dimensions yielding 42 Fortran routines for each corresponding C function (*allocate*, *free*, *put*, *get* and nonblocking *put* and *get*). The six Fortran data types supported are four (I4) and eight- (I8) byte integers, four- (R4) and eight- (R8) byte floating point numbers and eight (C4) and sixteen (C8) byte complex numbers. The following parameters provide a portable shorthand for defining these types and are found in the definekind.Fortran 95 file:

```
module definekind
   integer, parameter :: I4 = SELECTED_INT_KIND(9)
   integer, parameter :: I8 = SELECTED_INT_KIND(16)
   integer, parameter :: R4 = SELECTED_REAL_KIND(5)
   integer, parameter :: R8 = SELECTED_REAL_KIND(12)
   integer, parameter :: C4 = SELECTED_REAL_KIND(5)
   integer, parameter :: C8 = SELECTED_REAL_KIND(12)
end module definekind
```

For each operation in each of the 42 flavors, the `definekind` module is included and the appropriate type and dimension arguments are declared in an interface block to the generic C routine.

Sample RMA code using Fortran 95 interfaces

Below is a sample code snippet that allocates a couple of 50X50 arrays of integers, `src_arr` and `dst_arr`, and does a put operation from one to another. These arrays are first allocated with Malloc_fa interface and then `src` and `dst` slice information is filled up before doing the put communication.

```
integer(kind=4),pointer::src_arr(:,:),dst_arr(:,:)
type(Slice_fa) :: src_sl,dst_sl
integer :: lb(2), ub(2), ierr
lb(:) = 1
ub(:) = 50
call Malloc_fa(src_arr,lb,ub,ierr)
if (ierr .ne. 0) call myerror()
call Malloc_fa(dst_arr,lb,ub,ierr)
if (ierr .ne. 0) call myerror()
src_sl%lo(:) = 1
src_sl%hi(:) = 25
src_sl%stride(:) = 2
dst_sl%lo(:) = 25
dst_sl%hi(:) = 50
dst_sl%stride(:) = 2
Put_fa(src_arr,drc_sl,dst_arr,dst_sl,dst_proc,ierr)
```

3.Experimental Evaluation

We measured the latency and bandwidth of the Fortarn 95 RMA calls with microbenchmarks. We also ported the NAS MG benchmark to use Fortran 95. The experimental evaluation was carried out on a 24-node dual processor Intel Itanium2 1GHz cluster interconnected with Myrcom's GM interconnect [7]. The cluster was running Linux version 2.4.20 operating system. We used the GM dual port E cards, GM 2.1.4 and MPICH 1.2.5..12. For this test, we used Intel IFC Fortran 7.0 compilers and the 2.96 version of the GNU C compiler. We also used ARMCI 1.1 and Chasm 1.1.0 for the implementation.

Microbenchmarks

We measured the latency and bandwidth of Fortran 95 RMA interfaces with a microbenchmark that does consecutive Put and Get operations from different memory locations and averages the time taken for each operation. This is a simple microbenchmark that shows the bandwidth and latency of the Fortran 95 RMA

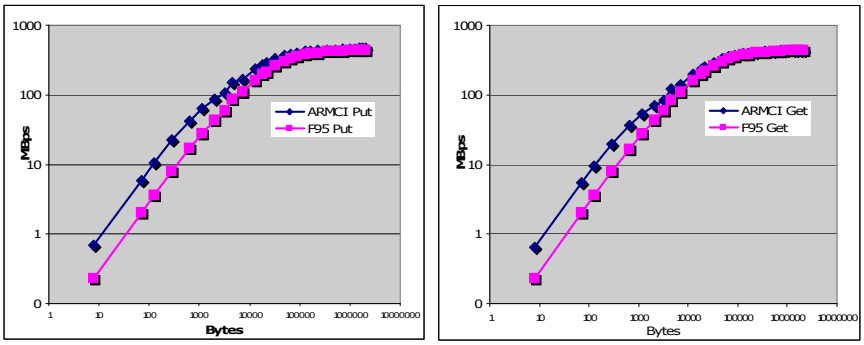

Figure 1- Left: Bandwidth of a contiguous Fortran 95 (labled F95 in the figure) Put_fa compared to ARMCI put. Right: Bandwidth of the Fortran 95 Get_fa compared to ARMCI Get operation.

interfaces. In addition we also used a similar microbenchmark to measure the bandwidth and latency of ARMCI Get and Put operations in order to measure the overhead from using Fortran 95 interfaces that involved interface mapping and all the dope vector manipulations. The bandwidth of the Get and Put operations for the Fortran 95 RMA interfaces is shown in Figure 1. The figure also includes the bandwidth of the corresponding ARMCI Put and Get calls. The overhead is independent of the message size and it is related to the cost of duplicating dope vector through Chasm that includes calloc system call. Based on these findings, the next version of Chasm will include an alternative mechanism for accessing some of the information stored in the dope vector that will be based on portable macros rather than duplication of the dope vector. The asymptotic b/w in the above microbenchmarks is consistent with bw numbers of Myricom GM [7].

NAS MG Benchmark

The Numerical Aerodynamic Simulation (NAS) parallel benchmarks (NPB) are a set of programs designed at NASA. Our starting point was NPB 2.4 [4] implementation written in MPI and distributed by NASA, we modified it to be compiled as a Fortran 95 file. We replaced MPI calls with the Fortran 95 non-blocking RMA interfaces. In addition to the mere replacement of the point-to-point message passing communications part of the current message-passing version of MG NAS kernels, an additional set of communication buffers were used to better utilize the one-sided nature of the RMA interfaces. Figure 2 shows the performance of NAS Fortran 95 MG version written with Fortran 95 RMA interface and is compared to the original MPI implementation of NAS which has been compiled with Intel Fortran 95 compiler as an Fortran 95 file. Despite the overhead Fortran 95 interfaces involve,, the RMA Fortran 95 RMA interface version of the MG benchmark outperforms the MPI version of NAS MG benchmark for Class B and Class C and performs in par with the MPI version for the Class A version of the benchmarks. The performance gains are contributed to the increased asynchronicity of the RMA model as compared to the two-sided message passing implementation of the NAS NPB MG benchmark. Table 1 shows the percentage improvement shown by the Fortran 95 RMA interface

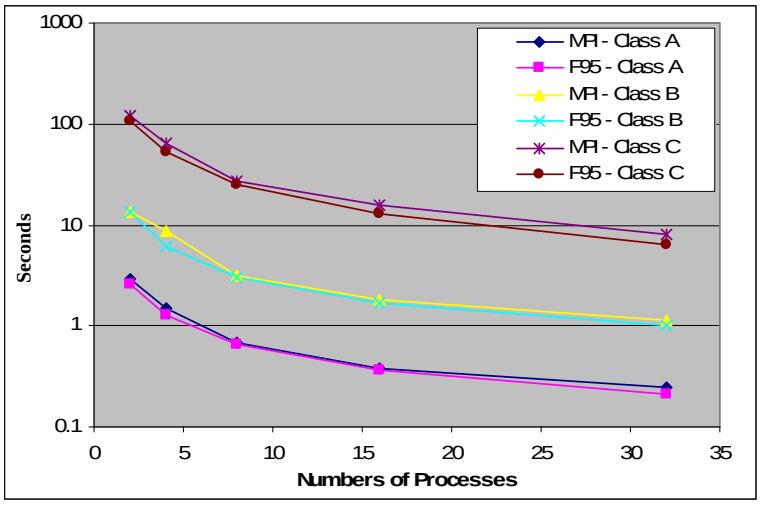

Figure 2- Fortran 95 (labled as F95 in the figure) RMA interfaces vs. MPI implementation of NAS MG benchmark for Class A, B and C

implementation of NAS MG over the standard MPI implementation of NAS MG. Up to 30% improvement can be seen.

4. Conclusions and Future Work

The current paper described how symmetric data objects and high-level array oriented RMA interfaces can be implemented for Fortran 95 applications. The proposed approach leads to simple yet efficient code, as demonstrated in the context of the NAS NPG Multi-Grid benchmark. In the process of developing the interface we identified sources of overhead involved in accessing elements of the dope vector through Chasm. The next version of Chasm will address them by providing macros for direct access to the information stored in the dope vector required by these interfaces. Our future work in addition to these performance optimizations will include performance comparisons with the Co-Array Fortran code on the Cray X1 where the native Co-Array compiler is available as well as to the Rice compiler on Linux clusters.
The microbenchmarks show the bandwidth of the Fortran 95 interfaces. The implementation of NAS MG using these Fortran 95 RMA interfaces outperforms the

NPROC	%improvement over MPI-Class B	%improvement over MPI-Class C
2	2.0	10.8
4	30.1	19.9
8	5.6	8.3
16	4.4	18.1
32	12.2	21.4

Table -1 percentage improvement over the MPI version of NAS MG of the MG implementation using Fortran 95 RMA interface

MPI version of the benchmark demonstrating that the non-blocking one-sided nature of RMA is preserved and utilized despite the overhead involved in pointer calculations and dope vector manipulations.

References

1. *http://www.j3-fortran.org*
2. Rasmussen, C.E., K.A. Lindlan, B. Mohr, and J. Striegnitz, CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability, Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium (CDROM), Santa Fe, NM, 2004.
3. Rasmussen, C.E., M.J. Sottile, S. Shende, and A.D. Malony, Bridging the Language Gap in Scientific Computing: The Chasm Approach, *Concurrency and Computation: Practice and Experience*, 2005.
4. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, The NAS parallel benchmarks, RNR-94-007, NASA 1994.
5. C. Coarfa, Y. Dotsenko, J. Eckhardt, J. Mellor-Crummey, Co-Array Fortran Performance and Potential: An NPB Experimental Study,16th International Workshop on Languages and Compilers for Parallel Computing. 2003.
6. J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems", Proc. RTSPP IPPS/SDP, 1999.
7. *http://www.myri.com*
8. *http://j3-fortran.org/doc/year/04/04-007.pdf*
9. *http://sourceforge.net/projects/chasm-interop/*
10. K. Parzyszek, J. Nieplocha, and R.A. Kendall, "A generalized portable SHMEM library for high performance computing", in *Proc. Parallel and Distributed Computing and Systems PDCS*, 2000
11. F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie "Performance Evaluation of the Quadrics Interconnection Network", *Journal of Cluster Computing*, 6(2): 125-142, April 2003
12. CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability *Rasmussen, C.E.; Lindlan, K.A.; Mohr, B.; Striegnitz, J. (2001)* Proceedings of the 2nd Annual Los Alamos Computer Science Symposium 2001
13. D. B. Carpenter. Adlib: A distributed array library to support HPF translation, 1995. 5th International Workshop on Compilers for Parallel Computers.
14. Robert W. Numrich and John K. Reid, Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 2, 1-31, 1998.
15. 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.
16. R. Bariuso and A. Knies, *SHMEM's User's Guide*, Cray Research, Inc., SN-2516, rev. 2.2, 1994.