ARMCI: A Portable Aggregate Remote Memory Copy Interface

Jarek Nieplocha
<jarek.nieplocha@pnl.gov>

Motivation and Background

A lightweight portable remote memory copy is needed in parallel distributed-array libraries and compiler runtime systems. A simple API in the style of the memory copy interface *memcpy(addr1, addr2, nbytes)* can lead to poor performance on systems with high latency networks for applications that require noncontiguous data transfers (for example, sections of dense multidimensional arrays or scatter/gather operations) and prevents the communication library from exploiting the information about the actual layout of the user data.

A more general API should describe a noncontiguous layout of data in memory to allow a communication library to: 1) minimize the number of underlying network messages by packing distinct blocks of data into as few messages as possible, 2) minimize the number of interrupts in the interrupt-driven message-delivery systems, and 3) take advantage of any available shared memory optimizations (prefetching/poststoring) on the shared memory systems. Remote copy operations should map directly -- without intermediate copying of the data -- to the native high-performance memory copy operations (including bulk data transfer facilities) when shared memory is used.

A lower-level interface than the MPI-2 one-sided communication (no epochs, windows, datatypes, Fortran-77 API, complicated progress rules etc.) would streamline an implementation and could greatly improve a portable performance of such a library. Straightforward progress rules simplify the development and performance analysis of the applications and avoid ambiguities of the platform-specific implementations: remote copy operations are to be truly one-sided and complete regardless of the actions taken by the remote process(or). In particular, polling can be helpful (for performance reasons); however, it should not be necessary to assure progress.

The copy operations should be 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 programming model and is required in many applications. Some systems enforce ordering of the otherwise unordered operations by providing a fence operation that blocks the calling process until the outstanding operation completes so that the next operation issued would not overtake it. This approach accomplishes more than the applications desire and could have negative impact on performance on the platforms where the copy operations are otherwise ordered. Usually, ordering can be accomplished with a lower overhead inside the communication library by using platform-specific means rather than at the application level with a fence operation.

A compatibility with message-passing libraries (primarily MPI) is necessary for real applications that frequently use hybrid programming models. Both blocking and a non-blocking APIs are needed. The non-blocking API can be used by some applications to overlap computations and communications.

ARMCI Data Structures

ARMCI is being developed based on the assumptions presented in the previous section. It offers two formats to describe noncontiguous layouts of data in memory.

1. *Generalized I/O vector.* It 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. It uses two arrays of pointers: one for source and one for destination addresses, see Figure 1. The length of each array is equal to the number of segments. Some operations that would map well to this format include scatter and gather. It will allow to transfer the entire 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;
```

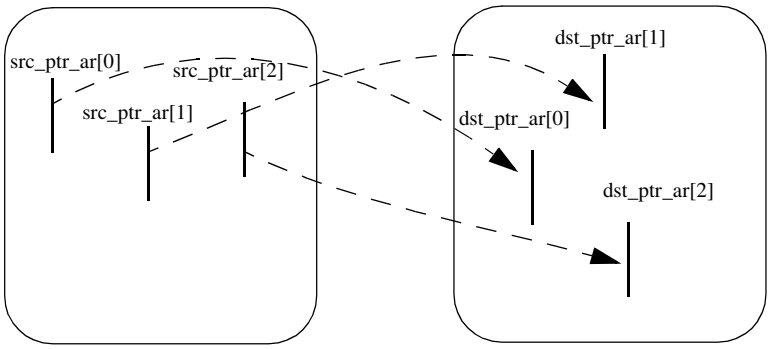

Figure1: Source and destination pointer arrays

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

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

The first argument is an array of size *arr_len*. Each array element specifies a set of equally-sized 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. It is intended to minimize storage required to describe sections of dense multidimensional arrays. Instead of including addresses for all the segments, it specifies only an address of the first segment in the set for source and destination. The addresses of the other segments can be computed 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* memory has the following interface:

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

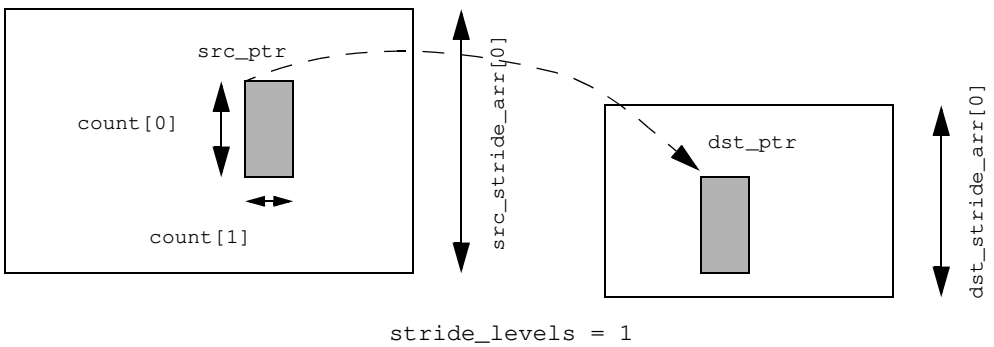

Figure 2: Strided format for 2-dimensional array sections

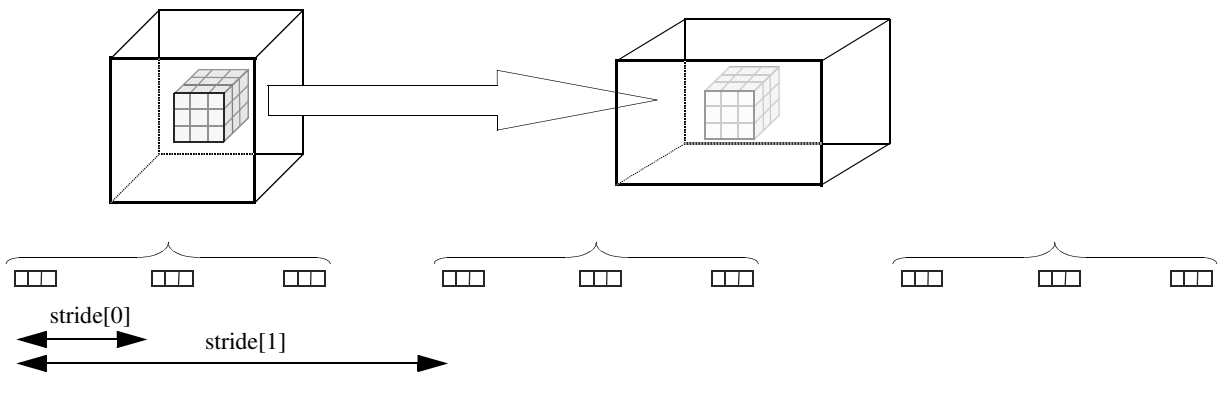

Figure 3: Memory layout and storage requirements needed to describe transfers of a 3x3x3 section of a 3-D array: The vector API uses 18 pointers (source & destination for 9 segments) and 1 +1 ints = 20 words. The strided API needs: 2 pointers (source & destination) and 2+2 +1 + 3 ints = 10 words.

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*.

A simple *put* operation for *nbytes* of continuous data in the style of *memcpy* (or *shmem_put* operation on Cray T3E) can be easily expressed in terms of either vector or strided ARMCI put operation. For example:

```
ARMCI_PutS(src_ptr, NULL, dst_ptr, NULL, &nbytes, 0, proc)
```

Figure 3 illustrates memory layout and storage requirements in the two ARMCI formats used to describe transfers of a 3x3x3 section of a 3-dimensional array. With the generalized I/O vector format, it is required to specify source and destination pointers for each segment (column) in the sequence. With the strided format only the source and destination addresses for the first have to be specified and the rest can be computed based on the stride arguments. The savings from using strided format can be quite substantial for a larger number of segments.

ARMCI Operations

A *get* operation transfers data from the remote process(or) memory (source) to the calling process(or) local memory (destination). A *put* operation transfers data from the local memory of the calling process(or) (source) to the memory of a remote process(or) (destination). The non-blocking API (with the *Nb* prefix) is derived from the blocking interface by adding a handle argument *req* that identifies an instance of the non-blocking request.

Progress and ordering

When a blocking *put* operation completes, the data has been copied out the calling process(or) memory but has not necessarily arrived to the 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 it issued to the specified remote process(or) complete at the destination. ARMCI_AllFence does the same for all the outstanding *put* operations issued by the calling process(or) regardless of the destination.

Memory allocation

On shared memory systems, ARMCI operations require the remote data to be located in shared memory. This restriction greatly simplifies an implementation and improves the performance. It can be lifted in the future versions of ARMCI if compelling reasons are identified. However, most likely the performance of ARMCI will always be more competitive if shared memory is used. As a convenience to the programmer, ARMCI provides two collective operations to allocate and free memory that can be used in the context of the ARMCI copy operations. They use local memory on systems that do not support shared memory and shared memory on those that do. The programmer can

perform memory allocation on its own using other means (e.g., with *malloc/free* on distributed memory systems and *shmget&shmat* or *mmap* on shared memory systems). ARMCI_Cleanup releases any system resources (like System V *shmem* ids) that ARMCI can be holding. It is intended to be used *before* terminating a program (e.g., by calling MPI_Abort) in case of an error. ARMCI_Error combines the functionality of ARMCI_Cleanup and MPI_Abort, and it prints (to stdout and stderr) a user specified message followed by an integer code.

List of Operations

1. Initialization/setup

```
int ARMCI_Init
int ARMCI_Finalize
```

2. Copy operations using the generalized I/O vector format

```
int ARMCI_PutV(armci_giov_t *dsrc_arr, int arr_len, int proc)
int ARMCI_NbPutV(armci_giov_t *dsrc_arr, int arr_len, int proc, int *req)
int ARMCI_GetV(armci_giov_t *dsrc_arr, int arr_len, int proc)
int ARMCI_NbGetV(armci_giov_t *dsrc_arr, int arr_len, int proc, int *req)
```

3. Copy operations using the strided format

```
int ARMCI_PutS(void* src_ptr, int src_stride_ar[], void* dst_ptr, int
    dst_stride_ar[], int count[], int stride_levels, int proc)
int ARMCI_NbPutS(void* src_ptr, int src_stride_ar[], void* dst_ptr, int
    dst_stride_ar[], int count[], int stride_levels, int proc,int *req)
int ARMCI_GetS(void* src_ptr, int src_stride_ar[], void* dst_ptr, int
    dst_stride_ar[], int count[], int stride_levels, int proc)
int ARMCI_NbGetS(void* src_ptr, int src_stride_ar[], void* dst_ptr, int
    dst_stride_ar[], int count[], int stride_levels, int proc, int *req)
```

4. Local completion of nonblocking operations

```
int ARMCI_Wait(int req)
```

5. Global completion of the outstanding operations

```
int ARMCI_Fence(int proc)
int ARMCI_AllFence()
```

6. Polling operation (not required for progress)

```
void ARMCI_Poll()
```

7. Memory allocation and release

```
int ARMCI_Malloc(void* prt[], int bytes)
int ARMCI_Free(void *address)
```

8. Cleanup and abnormal termination

```
void ARMCI_Cleanup()
void ARMCI_Error(char *message, int code)
```

The integer value returned by ARMCI operations represents an error code. The zero value is successful, other values represent a failure (described in the release notes).