ARMCI: A portable Aggregate Remote Memory Copy Interface

Jarek Nieplocha
02.06.1998

Motivation and Background

- A lightweight portable remote memory copy interface is needed in parallel libraries and compiler run-time systems.
- A simple API that follows the memory copy interface *memcpy(addr1, addr2, nbytes)* can lead to poor performance on the high latency networks for applications that require noncontiguous data transfers (for example, sections of dense multidimensional arrays or scatter/gather operations).
- A more general API should describe a noncontiguous layout of data in memory to allow the implementation to: 1) minimize the number of underlying network messages by packing distinct blocks of data into as few messages as possible and 2) take advantage of any available shared memory optimizations (prefetching/poststoring) on the shared memory systems.
- ARMCI operations should map directly to the native high-performance memory copy operations when shared memory is used.
- Lower-level than the MPI-2 one-sided communication (no epochs, windows, datatypes, Fortran API etc.).
- Simple progress rules: ARMCI operations are 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 is not necessary to assure progress.
- Compatibility with message-passing libraries (MPI or PVM) is needed.
- 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.
- Both blocking and non-blocking API is needed.

ARMCI Data Structures

Two types of data format are offered 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.

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

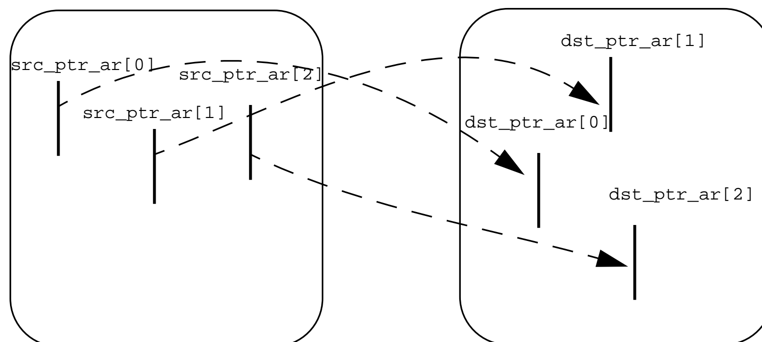

Figure1: Source and destination pointer arrays

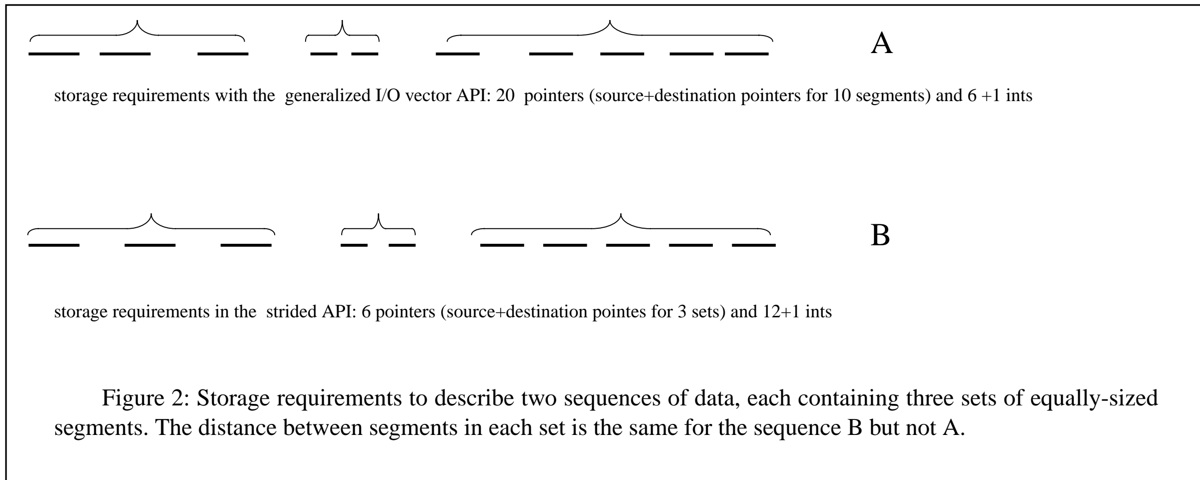

Figure 2: Storage requirements to describe two sequences of data, each containing three sets of equally-sized segments. The distance between segments in each set is the same for the sequence B but not A.

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. The addresses of the other segments can be computed using the stride parameter.

```
typedef struct{
  void *src_ptr;
  void *dst_ptr;
  int src_stride;
  int dst_stride;
  int bytes;
  int count;
}armci_strided_t;
```

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*.

Figure 2 illustrates storage requirements in the ARMCI formats used to describe two similar sequences of data. Both sequences contain three sets of equally-sized data segments. With the generalized I/O vector format, the specifications of both sequences require the same amount of storage. The difference between sequences is that the data segments in a set are evenly spaced in sequence B but not in A. Based on this fact, we can describe sequence B in a more compact strided format. Assuming a 4-byte representation for pointers and integers, the specification of sequence A requires 144 bytes while sequence B can be described with the strided format that uses only 76 bytes.

ARMCI Operations

Both blocking and non-blocking interfaces are available. The non-blocking versions (with the *Nb* prefix) add a handle argument *req* that identifies an instance of the non-blocking request. The maximum number of the outstanding nonblocking operations is given by the ARMCI_MAX_NUM_REQUESTS constant. 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 local memory of the calling process(or) (source) to the memory of a remote process(or) (destination).

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 outstanding *put* operations issued by the calling process(or) regardless of the destination.

On shared memory systems, ARMCI operations require the remote data to be located in the shared memory. This restriction greatly simplifies an implementation and improves performance. It can be lifted in the future versions of ARMCI if required by the applications. However, most likely the performance of ARMCI will 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 Sys 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.

1. Initialization
   ```
   int ARMCI_Init
   int ARMCI_Finalize
   ```
2. Copy operations using the generalized IO vector format
   ```
   int ARMCI_PutV(armci_giov_t *dscr_arr, int arr_len, int proc)
   int ARMCI_NbPutV(armci_giov_t *dscr_arr, int arr_len, int proc, int *req)
   int ARMCI_GetV(armci_giov_t *dscr_arr, int arr_len, int proc)
   int ARMCI_NbGetV(armci_giov_t *dscr_arr, int arr_len, int proc, int *req)
   ```
3. Copy operations using the strided format
   ```
   int ARMCI_PutS(armci_strided_t *dscr_arr, int arr_len, int proc)
   int ARMCI_NbPutS(armci_strided_t *dscr_arr, int arr_len, int proc,int *req)
   int ARMCI_GetS(armci_strided_t *dscr_arr, int arr_len, int proc)
   int ARMCI_NbGetS(armci_strided_t *dscr_arr, int arr_len, int proc,int *req)
   ```
4. Local completion of nonblocking operations
   ```
   int ARMCI_Wait(int req)
   ```
5. Global completion of nonblocking operations
   ```
   int ARMCI_Fence(int proc)
   int ARMCI_AllFence()
   ```
6. Polling operation (optional-- not required for progress)
   ```
   void ARMCI_Poll()
   ```
7. Memory allocation and release
   ```
   void* ARMCI_Malloc(int bytes)
   int ARMCI_Free(void *address)
   void ARMCI_Cleanup()
   ```

All operations except for ARMCI_Poll, ARMCI_Malloc and ARMCI_Cleanup return an integer error code. The zero value is successful, other values represent a failure (described in the release notes).