ARMCI: A Portable Aggregate Remote Memory Copy Interface

Version 1.1, October 30, 2000

Jarek Nieplocha and Jialin Ju
Pacific Northwest National Laboratory

Motivation and Background

A portable lightweight remote memory copy is needed in parallel distributed-array libraries and compiler run-time systems. However, a simple API for transfers of contiguous blocks of data in the style of the local memory copy operation *memcpy(addr1, addr2, nbytes)* is not optimal for this purpose. In particular, such an API 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). In most cases, the performance loss is due to the communication subsystem handling each contiguous portion of the data as a separate message. This causes the communication startup costs to be incurred multiple times rather than once. The problem is contributed to the inadequate API that does not pass the information about the intended data transfer and actual layout of the user data to the communication subsystem. Usually, there are many ways a communication library could optimize the performance if a more descriptive communication interface is used, 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 the interrupt-driven message-delivery systems, and 3) take advantage of any available shared memory optimizations (prefetching/poststoring) on the shared memory systems. In principle, 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.

Most of the existing remote memory copy interfaces do not offer interfaces sufficient to specify the noncontiguous data transfers used in the scientific computing area. The well known Cray SHMEM library [1] available on all the Cray and SGI platforms as well as clusters of PC through the HPVM package [2], supports only put/get operations for contiguous data and scatter/gather operations for the basic datatypes. The Virtual Interface Architecture (VIA) [3], a recent PC-industry interface specification, goes further in terms of supporting such transfers. However, its scatter/gather type of operations can transfer data between a noncontiguous location in memory and a contiguous buffer. For example, in scientific codes where data in a section of one array is transferred to a section of another array, the VIA interface would require additional memory copy to an intermediate buffer and therefore is not optimal. The MPI-2 standard includes one-sided communication specifications that includes remote memory copy operations such as put and get [4]. The noncontiguous data transfers are fully supported through the MPI derived data types. However, as the MPI-2 model has been derived from message-passing, its semantics including the rather complicated progress rules are closer to MPI-1 than to the existing remote memory copy interfaces such as SHMEM, Fujitsu MPlib, or IBM LAPI.

The Aggregate Remote Memory Copy Interface (ARMCI) is a new library that offers remote memory copy functionality. It aims to be fully portable and compatible with message-passing libraries such as MPI or PVM. It focuses on the noncontiguous data transfers unlike, for example, SHMEM. ARMCI targets a different user audience than MPI-2. In particular, it is meant to be used by the library rather than application developers. Example libraries that ARMCI is targeting include Global Arrays [5], P++/Overture[6], and Adlib PCRC run-time system [7]. ARMCI offers both simpler and lower-level model than the MPI-2 one-sided communication (no epochs, windows, datatypes, Fortran-77 API, complicated progress rules etc.) to streamline the implementation and improve its portable performance.

It is important for a communication library such as ARMCI to have straightforward progress rules: they simplify development and performance analysis of the applications built on top of libraries that use ARMCI, and avoid dealing with ambiguities of the platform-specific implementations. Therefore, the ARMCI remote copy operations are truly one-sided and complete regardless of the actions taken by the remote process. In particular, for performance reasons polling can be helpful (for example, to avoid the cost of interrupt processing); however, it should not be necessary to assure progress. The operations are ordered (complete in order they were issued) when referencing the same remote process. Operations issued to different processes can complete in an arbitrary order. Ordering simplifies programming model and is required in many applications such as computational chemistry. 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 the application performance on 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 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.

The ARMCI specification does not describe or assume any particular implementation model, for example threads. The implementation should exploit the most efficient mechanisms available on a given platform and might include active messages, native put/get, shared memory, and/or threads.

ARMCI Data Structures

ARMCI has been 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

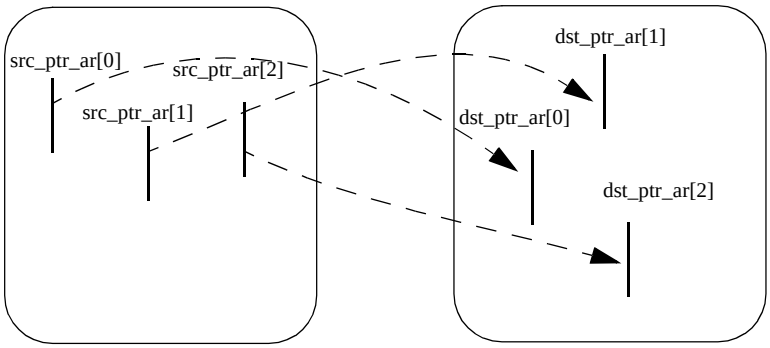

Figure1: Source and destination pointer arrays

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;
```

For example, with the generalized I/O vector format a *put* operation that copies data to the process *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 *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.

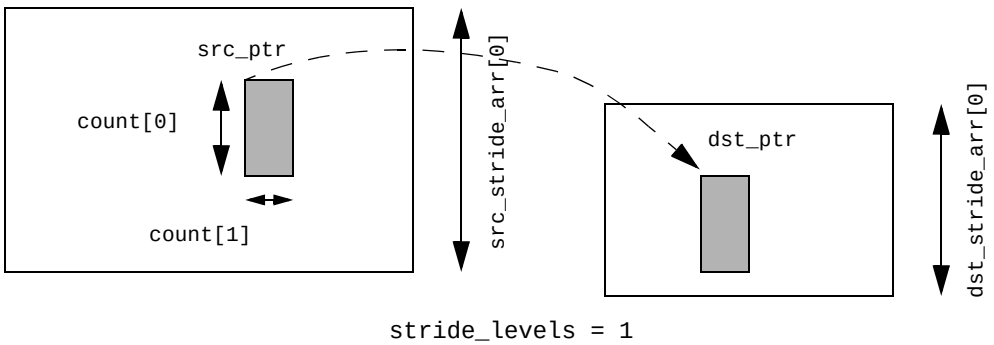

Figure 2: Strided format for 2-dimensional array sections using Fortran layout

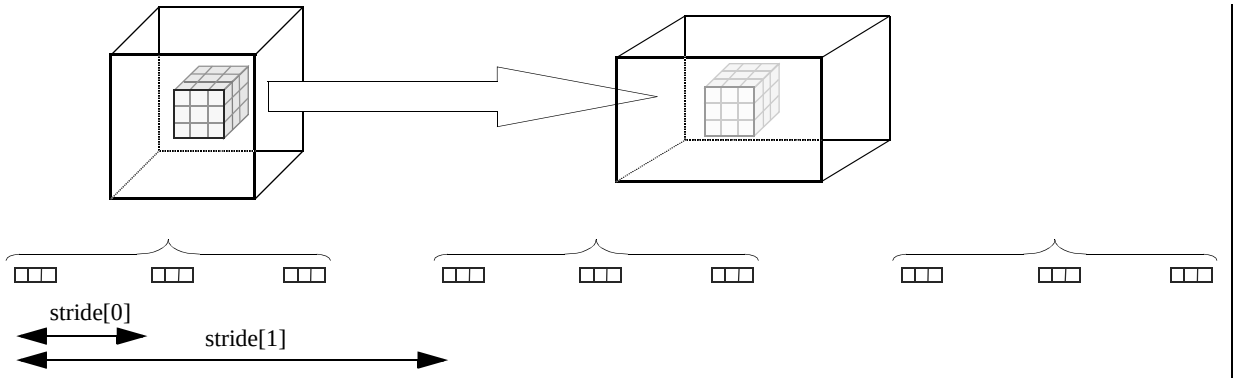

Figure 3: Memory layout and storage requirements needed to describe transfers of a 3x3x3 section of a 3-D array: The vector API uses 18 pointers (source & destination for 9 segments) and 1 +1 ints = 20 words. The strided API needs: 2 pointers (source & destination) and 2+2 +1 + 3 ints = 10 words.

```
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 *proc* 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 *proc*.

The ARMCI counterpart of the contiguous *put* operation available in other libraries (such as *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 the 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.

Example: Assume two 2-dimensional C arrays residing on different processes.

```
double A[10][20];   /* local process */
double B[20][30];   /* remote process */
```

To put a block of data, 3x6, starting at location (1, 2) in `A` to `B` in location (3, 4), the arguments of `ARMCI_PutS` can be set as following. The C data layout is assumed here.

```
src_ptr = &A[0][0] + (1 * 20 + 2);       /* row-major in C */
 src_stride_ar[0] = 20 * sizeof(double); /* number of bytes for the stride*/
dst_ptr = &B[0][0] + (3 * 30 + 4);
dst_stride_ar[0] = 30 * sizeof(double);
count[0] = 6 * sizeof(double);    /*  number of bytes of contiguous data */
count[1] = 3;                     /*  number of rows (C layout) of contiguous data */
stride_levels = 1;
proc = ID                      /* remote process ID where array B resides*/
```

ARMCI Operations

Data transfer operations

A *get* operation transfers data from the remote process memory (source) to the calling process local memory (destination). A *put* operation transfers data from the local memory of the calling process (source) to the memory of a remote process (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.

Atomic operations

In addition to the copy operations, an accumulate operation is provided. The operation is similar to put, except instead of overwriting the remote memory location it atomically updates the remote data. In particular it adds a content of the local memory (scaled a specified factor) to the data in the remote location. Accumulate works on integer and floating-point data types including complex numbers. The datatype is specified by the appropriate argument in *accumulate: ARMCI_ACC_INT, ARMCI_ACC_LNG, ARMCI_ACC_FLT, ARMCI_ACC_DBL, ARMCI_ACC_CPL, ARMCI_ACC_DCPL* for int, long, float, double, complex and double complex types. To maximize performance, ARMCI does not specify which process will perform the computations.

Another type of atomic operations available in ARMCI is *read-modify-write*. There are two types of operators for that operation supported: *fetch-and-add* and *swap*. The fetch-and-add combines the specified integer (*int* or *long*) value with the corresponding integer value at the remote memory location and returns the original value found at that location. This operation can be used to implement shared counters and other synchronization mechanisms. The datatype are specified by selection of the appropriate argument: ARMCI_FETCH_AND_ADD for *int* or ARMCI_FETCH_AND_ADD_LONG for the *long* data type. The swap operation swaps the content of remote memory location with the specified local integer value. The operation is supported for *int* and *long* datatypes.

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

Mutexes

ARMCI supports distributed mutex operations. The user can create a sets of mutexes associated with a specified process and use locking operations ARMCI_Lock/ARMCI_Unlock on individual mutexes in that set.

Progress and ordering

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

Memory allocation

For performance reasons 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. ARMCI provides two collective operations to allocate and free memory that can be used in the context of the ARMCI data transfer operations. They use local memory on systems that do not support shared memory and shared memory on those that do. 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 *stdou*t and *stderr*) a user specified message followed by an integer code.

List of Operations

1. Initialization/setup

   `int ARMCI_Init()`

   Purpose: Initializes the ARMCI. This function must be called before any ARMCI functions.

   `int ARMCI_Finalize()`

   Purpose: Finalizes the ARMCI. This function must be called after using ARMCI functions.

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

   `int ARMCI_PutV(armci_giov_t *dsrc_arr, int arr_len, int proc)`

   Purpose: Generalized I/O vector operation that transfer data from the local memory of calling process (source) to the memory of a remote process (destination).

   Data Type:

```
typedef struct {
    void *src_ptr_ar;     - Source starting addresses of each data segment.
    void *dst_ptr_ar;     - Destination starting addresses of each data segment.
    int bytes;            - The length of each segment in bytes.
    int ptr_ar_len;       - Number of data segment.
} armci_giov_t;
```

   Arguments:

```
dsrc_arr      - Array of data (type of armci_giov_t) to be put to remote process.
arr_len       - Number of elements in the dsrc_arr.
proc          - Remote process ID (destination).
```

   Return value:

```
zero          - Successful.
other value   - Error code (described in the release notes).
```

```
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)
```

Purpose: Generalized I/O vector operation that transfer data from the remote process memory (source) to the calling process local memory (destination).

Data Type:

```
typedef struct {
    void *src_ptr_ar;     - Source starting addresses of each data segment.
    void *dst_ptr_ar;     - Destination starting addresses of each data segment.
    int bytes;            - The length of each segment in bytes.
    int ptr_ar_len;       - Number of data segment.
} armci_giov_t;
```

Arguments:

`dsrc_arr` - Array of data (type of `armci_giov_t`) to get from remote process.
`arr_len` - Number of elements in the `dsrc_arr`.
`proc` - Remote process ID (source).

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

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)
```

Purpose: Optimized strided I/O vector operation that transfer data from the local memory of calling process (source) to the memory of a remote process (destination).

Arguments:

`src_ptr` - Source starting address of the data block to put.
`src_stride_ar` - Source array of stride distances in bytes.
`dst_ptr` - Destination starting address to put data.
`dst_stride_ar` - Destination array of stride distances in bytes.
`count` - Block size in each dimension. `count[0]` should be the number of bytes of contiguous data in leading dimension.
`stride_levels` - The level of strides.
`proc` - Remote process ID (destination).

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

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

Purpose: Optimized strided I/O vector operation that transfer data from the remote process memory (source) to the calling process local memory (destination).

Arguments:

`src_ptr` - Source starting address of the data block to get.
`src_stride_ar` - Source array of stride distances in bytes.
`dst_ptr` - Destination starting address to get data.
`dst_stride_ar` - Destination array of stride distances in bytes.
`count` - Block size in each dimension. `count[0]` should be the number of bytes of contiguous data in leading dimension.
`stride_levels` - The level of strides.
`proc` - Remote process ID (source).

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

4. Copy operations for contiguous data

```
int ARMCI_Put(src, dst, bytes, proc)
```

Purpose: Transfer contiguous data from the local process memory (source) to remote process memory (destination).

Arguments:

`src` - Source starting address of the data block to put.
`dst` - Destination starting address to put data.
`bytes` - amount of data to transfer in bytes.
`proc` - Remote process ID (destination).

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

```
int ARMCI_Get(src, dst, bytes, proc)
```

Purpose: Transfer contiguous data from the remote process memory (source) to the calling process memory (destination).

Arguments:

`src` - Source starting address of the data block to put.
`dst` - Destination starting address to put data.
`bytes` - amount of data to transfer in bytes.
`proc` - Remote process ID (destination).

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

5. Accumulate operation using the generalized I/O vector format

```
int ARMCI_AccV(int datatype, void *scale, armci_giov_t *dsrc_arr,
               int arr_len, int proc)
```

Purpose: Generalized I/O vector operation that atomically updates the memory of a remote process (destination).

Data Type:

```
typedef struct {
    void *src_ptr_ar;     - Source starting addresses of each data segment.
    void *dst_ptr_ar;     - Destination starting addresses of each data segment.
    int bytes;            - The length of each segment in bytes.
    int ptr_ar_len;       - Number of data segment.
} armci_giov_t;
```

Arguments:

`datatype` - Available data types are:
ARMCI_ACC_INT -> int, ARMCI_ACC_LNG -> long,
ARMCI_ACC_FLT -> float, ARMCI_ACC_DBL-> double,
ARMCI_ACC_CPL-> complex, ARMCI_ACC_DCPL -> double complex.
`scale` - Scale for the data (dest = dest + scale * src).
`dsrc_arr` - Array of data (type of `armci_giov_t`) to be accumulated to the remote process.
`arr_len` - Number of elements in the `dsrc_arr.`
`proc` - Remote process ID.

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

```
int ARMCI_NbAccV(int datatype, void *scale, armci_giov_t *dsrc_arr,
                 int arr_len, int proc, int *req)
```

6. Accumulate operation using the strided format

```
int ARMCI_AccS(int datatype, void *scale, void* src_ptr, int
    src_stride_ar[], void* dst_ptr, int dst_stride_ar[], int count[], int
    stride_levels, int proc)
```

Purpose: Optimized strided I/O vector operation that atomicaly updates the memory of a remote process (destination).

Arguments:

`datatype` - Available data types are:
ARMCI_ACC_INT -> int, ARMCI_ACC_LNG -> long,
ARMCI_ACC_FLT -> float, ARMCI_ACC_DBL-> double,
ARMCI_ACC_CPL-> complex, ARMCI_ACC_DCPL -> double complex.
`scale` - Scale for the data (dest = dest + scale * src).
`src_ptr` - Source starting address of the data block to put.
`src_stride_ar` - Source array of stride distances in bytes.
`dst_ptr` - Destination starting address to put data.
`dst_stride_ar` - Destination array stride distances in bytes.
`count` - Block size in each dimension. `count[0]` should be the number of bytes of contiguous data in leading dimension.
`stride_levels` - The level of strides.
`proc` - Remote process ID (destination).

Return value:

`zero` - Successful.
`other value` - Error code (described in the release notes).

7. Read-modify-write

```
int ARMCI_Rmw(int op, void *ploc, void *prem, int value, proc)
```

Purpose: Combines atomically the specified integer value with the corresponding integer value (*int* or *long*) at the remote memory location and returns the original value found at that location.

Arguments:

op - Available operations are:
ARMCI_FETCH_AND_ADD -> int
ARMCI_FETCH_AND_ADD_LONG -> long
ARMCI_SWAP -> int
ARMCI_SWAP_LONG ->long
`ploc` - Pointer to the local memory.
`prem` - Pointer to the remote memory.
`value` - Value to be added to the remote memory.
`proc` - Remote process ID.

8. Global completion of the outstanding operations

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

Purpose: Blocks the calling process until all *put* or *accumulate* operations it issued to the specified remote process complete at the destination.

Arguments:

`proc` - Remote process ID.

`int ARMCI_AllFence()`

Purpose: Blocks the calling process until all the outstanding *put* or *accumulate* operations complete regardless of the destinations.

9. Mutex operations

`int ARMCI_Create_mutexes(int count)`

Purpose: Collective operation to create sets of mutexes on individual processes. Eacg process specifies the number of mutexes associated with that process. The total number of mutexes allocate will be a sum of the values specified on each process.

Arguments:

`count`	- number of mutexes allocated on calling process count=0 means that no mutexes will be associated with that process.

Return value:

`zero`	- Successful.
`other value`	- Error code (described in the release notes).

`int ARMCI_Destroy_mutexes(void)`

Purpose: Collective operation to destroy mutex sets allocated by `ARMCI_Create_mutexes`.

Arguments: none

Return value:

`zero`	- Successful.
`other value`	- Error code (described in the release notes).

`void ARMCI_Lock(int mutex, int proc)`

Purpose: Acquire the specified mutex on he specified process on behalf of the calling process.

Arguments:

`mutex`	- Mutex number (0..count-1)
`proc`	- Remote process ID

`void ARMCI_Unlock(int mutex, int proc)`

Purpose: `Releas` the specified mutex on he specified process on behalf of the calling process. The mutex must have been acquired with `ARMCI_Lock`.

Arguments:

`mutex`	- Mutex number (0..count-1)
`proc`	- Remote process ID

10. Memory allocation and release

`int ARMCI_Malloc(void* ptr[], int bytes)`

Purpose: Collective operation to allocate memory that can be used in the context of ARMCI copy operations.

Arguments:

`ptr`	- Pointer array. Each pointer points to the allocated memory of one process.
`bytes`	- The size of allocated memory in bytes.

Return value:

`zero`	- Successful.
`other value`	- Error code (described in the release notes).

`int ARMCI_Free(void *address)`

Purpose: Collective operation to free memory which was allocated by `ARMCI_Malloc`.

Arguments:

`address`	- pointer to the allocated memory.

Return value:

`zero`	- Successful.
`other value`	- Error code (described in the release notes).

11. Cleanup and abnormal termination

`void ARMCI_Cleanup()`

Purpose: 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 error.

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

Purpose: Combines the functionality of `ARMCI_Cleanup` and `MPI_Abort`, and it prints (to the `stdout` and `stderr`) a user specified message followed by an integer code.

Arguments:

message	- Message to print out
code	- Error code.

References

1. R. Barriuso, Allan Knies, SHMEM User's Guide, Cray Research Inc., SN-2516, 1994.
2. 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 *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
3. Compaq Computer Corp., Intel Corp., Microsoft Corp., Virtual Interface Architecture Specification, Dec., 1997.
4. MPI. Forum. MPI-2: Extension to message passing interface, U. Tennessee, July 18, 1997.
5. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomputing*, 10:197–220, 1996.
6. Overture: Los Alamos Overlapping Grid, http://www.c3.lanl.gov/%7Ehenshaw/Overture/Overture.html, 1998.
7. D.B. Carpenter, Adlib: A distributed array library to support HPF translation, 5th Int. Workshop. Compilers for Parallel Computers, 1995. Adlib homepage http://www.npac.syr.edu/projects/pcrc/doc.