Overview

Aggregate Remote Memory Copy Interface (ARMCI) is a widely portable high-performance Remote Memory Access (RMA) communication library. It's major capabilities and characteristics include:

- blocking and nonblocking data transfer operations
- aggregation of small data transfers into larger messages to reduce sensitivity to network latency
- register originated data transfers
- atomic and synchronization operations
- memory managment operations
- Processor Groups
- Global proceduce calls (prototype)
- weakly consistent memory model

In addition to the RMA interfaces, ARMCI includes a small set of collective message-passing operations including broadcast, reduce, allreduce, barrier. These operations might be implemented as wrappers to MPI but also on some platforms when implemented independently can deliver performance competitive to MPI.

ARMCI relies on a message-passing library (e.g., MPI) for process creation and managment of the execution environment. The programmer can use message-passing calls along with ARMCI calls.

Blocking and Non-blocking 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, is derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request.

All the non-blocking transfer functions are prototyped to work as transfers with both "explicit" and "implicit handle". It stores important information about the initiated data transfer. The descriptor is implemented as an abstract data type. This is motivated by a simpler implementation so that a data transfer descriptor can be stored and managed in the application rather in the ARMCI library space. If a NULL value is passed to the argument representing a handle (thus representing "implicit handle"), the function does an implicit handle non-blocking transfer. A request data structure embedded in the handle should not be copied in the application. Upon completion of the data transfer, handle can be reused. A handle can be used to represent multiple operations of the same type (i.e., all puts or all gets). Such handle is an aggregate handle. Underneath, ARMCI combines multiple requests and processes them as a single message (actually by calling ARMCI_PutV/GetV/AccV). An explict handle should be initialized using the following macro, before it is used in any non-blocking operation. It is initialized as follows:

```
ARMCI_INIT_HANDLE(armci_hdl_t* nb_handle)
```

Nonblocking operations in ARMCI allow user ot initiate a one-sided call and then return control to the user program. The data transfer is completed locally by calling a wait operation. Waiting on a nonblocking put operation assures was injected into the network and the user buffer can be now reused.
Both in case of blocking and nonblocking store operations, to access the modified data safely from other nodes programmer has to call an ARMCI_Fence call first. ARMCI_Fence completes data transfers on the remote side. Unlike the blocking operation, the nonblocking operations are NOT ordered.

Coalescing small messages (trading latency for bandwidth)

Aggregation of requests is another mechanism for improving latency tolerance. Multiple nonblocking data transfers (put/get) requests can be aggregated into a single data transfer operation in order to improve the data transfer rate. Especially, if there are multiple data transfer requests of small message sizes, aggregating those requests into a single large request reduces the latency, thus improving performance. This technique is unique in its ability to sustain high bandwidth utilization and enables high throughput. Each of these requests can be of different size and independent of data type. The aggregate data transfer operation is also independent of the type of put/get operation i.e. it can be a combination of regular, strided or vector put/get operations. There are two types of aggregation available:

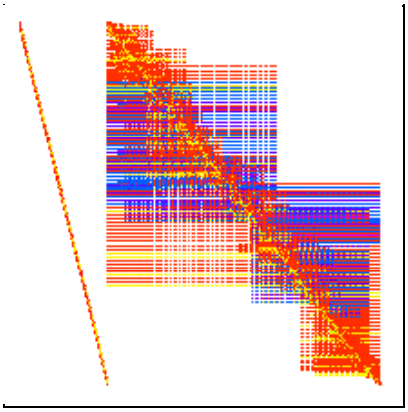
1) **explicit aggregation**, where the multiple requests are combined by the user through the use of strided or generalized I/O vector data descriptor, and

2) **implicit aggregation**, where the combining individual requests is performed by ARMCI.

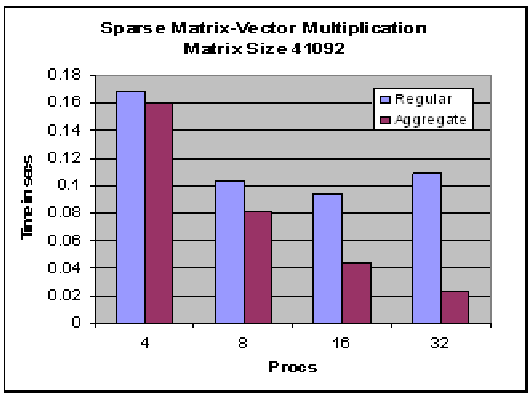
the use of strided or generalized I/O vector data descriptor, and

2) **implicit aggregation**, where the combining individual requests is performed by ARMCI.

Sparse matrix-vector multiplication benchmark illustrates this light-weight capability in ARMCI. In this benchmark, one of the sparse matrices from Harwell-Boeing collection is used. Instead of gathering the entire vector, each process caches the vector elements corresponding to the non-zero element columns of its locally owned sparse matrix. When aggregation is enabled, all the get calls corresponding to a single processor is aggregated into a single request, thus reducing the overall latency and improving the data transfer rate.



Harwell-Boeing Sparse Matrix



Performance in IA64 Linux cluster - Myrinet Interconnect

Register Originated Data Transfer

Register-memory operations (value_put/value_get) transfer a value stored in a register of local process to remote process memory (destination), by avoiding the overhead of passing through the buffer management layer (local memory sub-system).

Atomic Operations

In addition to the data transfer operations, two types of atomic operations are provided: **accumulate** and **read-modify_write**. The accumulate 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 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.

Mutexes and Locks

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.

Memory allocation

For performance reasons, ARMCI operations require the remote data to be allocated using the provided memory allocation function, **ARMCI_Malloc**. This requirement allows ARMCI to use the type of memory that allows fastest access (e.g., shared memory on SMP clusters). In addition, the library provides a memory allocator for local memory **ARMCI_Malloc_local**. Although it is not required to use that interface for allocating communication buffers, on some platforms (e.g., Myrinet or Infiniband clusters) there could be substantial performance benefits achieved. This is because ARMCI_Malloc_local attempts to allocate "best type of memory" for the interprocessor communication. ARMCI_Malloc is a collective memory allocator, whereas ARMCI_Malloc_local works like the standard malloc call. Another important difference between these calls is that only memory allocated with ARMCI_Malloc is accessible from other processors. Memory allocated by ARMCI_Malloc_local is private to the given processor.
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. 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.