PGAS Programming: The ARMCiI Approach

Manojkumar Krishnan\textsuperscript{1}, Sriram Krishnamoorthy\textsuperscript{1}, Bruce Palmer\textsuperscript{1}, P. Sadayappan\textsuperscript{2}, Daniel Chavarria\textsuperscript{1}, Abhinav Vishnu\textsuperscript{1}, Jeff Daily\textsuperscript{1}

\textsuperscript{1}Pacific Northwest National Laboratory
\textsuperscript{2}Ohio State University
**ARMCI Model**

- **Copy to local memory**
- **Get**
- **Compute/update**
- **Copy to shared object**
- **Put**

**Proc 0**
- `ptr[0]`
- `ptr[1]`
- `ptr[2]`
- `ptr[3]`
- `ptr[4]`
- `ptr[5]`
- `ptr[6]`
- `ptr[7]`

**Remote pointers**
- `ptr[0]`
- `ptr[1]`
- `ptr[2]`
- `ptr[3]`
- `ptr[4]`
- `ptr[5]`
- `ptr[6]`
- `ptr[7]`
Outline

- Writing, Building, and Running ARMCI Programs
- Basic Calls
- Intermediate Calls
- Advanced Calls
Writing, Building and Running ARMCI programs

- Installing ARMCI
- Writing ARMCI programs
- Compiling and linking
- Running ARMCI programs
- For detailed information
  - ARMCI Webpage (or search: ARMCI)
    - ARMCI software, papers, docs, APIs, etc.
    - http://www.emsl.pnl.gov/docs/parsoft/armci/
  - ARMCI API Documentation
    - http://www.emsl.pnl.gov/docs/parsoft/armci/documentation.htm
  - ARMCI Support/Help Mailing list
    - hpctools@pnl.gov
Installing ARMCI

- **configure; make; make install**
  - Refer to README/INSTALL for detailed installed instructions

- **Specify the underlying network communication protocol**
  - By default ARMCI builds over sockets
  - For high-performance interconnects, it is strongly recommended to configure ARMCI as in table below
  - If you are not sure about the network, then use the ARMCI auto-detect feature to build ARMCI with the best available network in your system*

<table>
<thead>
<tr>
<th>Configure option</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>configure --with-openib</code></td>
<td>Infiniband OpenIB</td>
</tr>
<tr>
<td><code>configure --with-dcmf --host=powerpc-bgp-linux --build=powerpc64-bgp-linux</code></td>
<td>IBM BG/P</td>
</tr>
<tr>
<td><code>configure --with-bgml[=ARG]</code></td>
<td>IBM BG/L</td>
</tr>
<tr>
<td><code>configure --with-lapi</code></td>
<td>IBM LAPI</td>
</tr>
<tr>
<td><code>configure --with-portals</code></td>
<td>Cray XT Seastar (portals)</td>
</tr>
<tr>
<td><code>configure --with-cray-shmem</code></td>
<td>Cray XT shmem</td>
</tr>
<tr>
<td><code>configure --with-mpi-spawn</code></td>
<td>MPI-2 dynamic process mgmt</td>
</tr>
<tr>
<td><code>configure --with-sockets</code></td>
<td>Ethernet (TCP/IP Sockets)</td>
</tr>
<tr>
<td><code>configure --enable-autodetect</code></td>
<td>* Attempts to locate HPC interconnects besides SOCKETS</td>
</tr>
</tbody>
</table>
Writing ARMCI Programs

- ARMCI Definitions and Data types
  - `#include "armci.h"
  - `ARMCI_Init, ARMCI_Finalize` --> initializes and terminates ARMCI library

```c
#include <stdio.h>
#include "mpi.h"
#include "armci.h"

int main( int argc, char **argv ) {
    MPI_Init( &argc, &argv );
    ARMCI_Init(); /* (or) ARMCI_Init_args(&argc, &argv); */

    printf( "Hello world\n" );

    ARMCI_Finalize();
    MPI_Finalize();
    return 0;
}
```
Writing ARMCI Programs

- ARMCI is compatible with MPI
- ARMCI requires the following functionalities from a message passing library (MPI/TCGMSG)
  - initialization and termination of processes
  - Broadcast, Barrier
- The message-passing library has to be
  - initialized before the ARMCI library
  - terminated after the ARMCI library is terminated

```c
#include <stdio.h>
#include "mpi.h"
#include "armci.h"

int main( int argc, char **argv ) {
    MPI_Init( &argc, &argv );
    ARMCI_Init();

    printf( "Hello world\n" );

    ARMCI_Finalize();
    MPI_Finalize();
    return 0;
}
```
Compiling and Linking ARMCI Programs

For example, compile and link ARMCI test program (armci_test.c) as follows.

```bash
ARMCI_INCLUDE = /home/manoj/armci-1.5/include
ARMCI_LIB = -L/home/manoj/armci-1.5/lib -larmci

mpicc -I$(ARMCI_INCLUDE) -o armci_test armci_test.c $(ARMCI_LIB) -lm
```
Running ARMCI Programs

- Example: Running a test program “armci_test” on 2 processes
  - mpirun -np 2 armci_test
- Running an ARMCI program is same as MPI
Outline

- Writing, Building, and Running ARMCI Programs
  - Basic Calls
  - Intermediate Calls
  - Advanced Calls
ARMCI Basic Operations

- ARMCI programming model is very simple.
- Most of a parallel program can be written with these basic calls
  - ARMCI_Init, ARMCI_Finalize
  - ARMCI_Malloc, ARMCI_Free
  - ARMCI_Put, ARMCI_Get
  - ARMCI_Barrier
ARMCI Initialization/Termination

There are two functions to initialize ARMCI:

- int ARMCI_Init()
- int ARMCI_Init_args(int *argc, char ***argv)

To terminate ARMCI program:

- void ARMCI_Finalize()

```c
#include <stdio.h>
#include "mpi.h"
#include "armci.h"

int main( int argc, char **argv ) {
    MPI_Init( &argc, &argv );
    ARMCI_Init();

    printf( "Hello world\n" );

    ARMCI_Finalize();
    MPI_Finalize();
    return 0;
}
```
Parallel Environment - Process Information

- **Parallel Environment:**
  - how many processes are working together (*size*)
  - what their IDs are (ranges from 0 to *size*-1)
  - Since ARMCI is compatible with MPI, you can get these from `MPI_Comm_rank/MPI_Comm_size`
Memory Allocation

- Collective operation to allocate memory that can be used in the context of ARMCI operations (e.g. put, get, accumulate, rmw, etc)
  - Remote pointers
  - Pinned/registered memory
  - Non-symmetric

```c
int ARMCI_Malloc(void* ptr[], armci_size_t bytes)
int ARMCI_Free(void *address)
```

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

RETURN VALUE:
- `zero` - Successful; other value - Error code (described in release notes).
Memory Allocation

Physically distributed data

Global Address Space

Proc 0
ptr[0]
ptr[1]
ptr[2]
ptr[3]
ptr[4]
ptr[5]
ptr[6]
ptr[7]

Proc 1
ptr[0]
ptr[1]
ptr[2]
ptr[3]
ptr[4]
ptr[5]
ptr[6]
ptr[7]

Remote pointers

int ARMCIMalloc(void* ptr[], armci_size_t bytes)
Memory Allocation (Contd.)

- ARMCI_Malloc_local()
- Non Collective Memory allocation
- Pinned/registered memory
  - Enables zero-copy

```c
void* ARMCI_Malloc_local(armci_size_t bytes)
int ARMCI_Free_local(void *address)
```
Data Transfer Operations

- Contiguous data transfer (put, get, acc)
- Non-Contiguous data transfer
  - Strided API (puts, gets, accs)
  - Vector API (putv, getv, accv)
int **ARMCI_Put**(void* src, void* dst, int bytes, int proc)

PURPOSE: Blocking transfer of 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(void* src, void* dst, int bytes, int proc)

PURPOSE: Blocking transfer of contiguous data from the remote process memory (source) to the calling process memory (destination).

ARGUMENTS:
  src  - Source starting address of the data block to get.
  dst  - Destination starting address to get the 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_Acc(int datatype, void *scale, void* src, void* dst, int bytes, int proc)

PURPOSE: Blocking operation that atomically updates the memory of a remote process (destination).

ARGUMENTS:
- datatype - Supported 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 data (dest = dest + scale * src)
- src - Source starting address of data to transfer
- dst - Destination starting address to add incoming data
- bytes - amount of data to transfer in bytes
- proc - Remote process ID (destination)
Strided API

- Strided API to handle non-contiguous data transfer
  - ARMCI_PutS, ARMCI_GetS, ARMCI_AccS
- Can handle arbitrary N-dimensional array sections

```c
int ARMCI_PutS(src_ptr, src_stride_arr, dst_ptr, dst_stride_arr, count, stride_levels, proc)
```
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.

```c
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); /* 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*/
```
Vector API

Most general API for non-contiguous data transfer

- ARMCI_PutV, ARMCI_GetV, ARMCI_AccV
- based on the I/O vector API (Unix `readv/writev`)
- Vector descriptor specifies sets of equally-sized data segments

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

```c
int ARMCI_PutV(armci_giov_t descr_arr[], int arr_len, int proc)
```
Synchronization

- Fence
- Barrier
- Data consistency
Fence

```c
void ARMCI_Fence(int proc)

PURPOSE: Blocks the calling process until all put or accumulate operations issued to the specified remote process complete at the destination.
ARGUMENTS:
    proc    - Remote process ID.
```

```c
void ARMCI_AllFence()

PURPOSE: Blocks the calling process until all the outstanding put or accumulate operations complete remotely regardless of the destination processor.
```
ARMCI Barrier

void ARMCI_Barrier()

- Synchronize processors and memory. This operation combines functionality of MPI_Barrier and ARMCI_AllFence
Completion and Ordering

- **Put/Accumulate**
  - Blocks the calling process until put/accumulate operation is completed locally and the buffer is safe to reuse (local completion)

- **Fence**
  - Ensures put/accumulate operations, issued to the specified remote process, complete at the destination (remote completion)

- **All blocking calls (e.g. ARMCI_Put) are ordered**
  - Non-blocking operations (e.g. ARMCI_Nbput) are NOT ordered
Illustration: Creating a 2-D Global Array

g_a = GA_Create(type, dims[2], name, chunk[2])

• Blocked distribution
• Locate free index in array of handles
• Store meta-data
• Compute bytes to allocate on this process
• Allocate memory

```c
static struct {
    int active;
    int type, dims[2], blocks[2];
    void *ptr_arr[MAXPROC];
} hndl[MAX_HANDLES];

int NGA_Create(type, dims[2], name, chunk[2]) {
    int ga, bytes;
    ga = /*ga s.t. hndl[ga] == 0*/
        hndl[ga].active = 1;


    /* compute size of blocking */
    compute_blocking(chunk, blocks);

    bytes = size(type) * blocks[0] * blocks[1];
    ARMCI_Malloc(hndl[ga].ptr_arr, bytes);
}
```
void NGA_Put(ga, lo[2], hi[2], buf, ld) {
    /* determine list of blocks to obtain and which procs own them */
    foreach block {
        src_ptr = buf + /* offset corresponding to this block */;
        src_stride_arr = ld;
        dst_ptr = hndl[ga].ptr_arr[proc] + /* offset at remote end */;
        dst_stride_arr = blocks;
        stride_levels = 2;
        seg_count[0] = sizeof(type)*(hi[0] – lo[0]+1);
        proc = /* (local/remote) process owns this block */

        ARMCI_Put(src_ptr, src_stride_arr, dst_ptr,
                  dst_stride_arr, stride_levels,
                  seg_count, proc);
    }
}
Locks/Mutexes

- Mutual exclusion primitive
- Locks identified by integers
- Creation: Each process specifies the number of locks on itself

```
ARMCI_Create_mutexes(nmutexes)
ARMCI_Lock(lockid, proc)
ARMCI_Unlock(lockid, proc)
```
Non-Blocking Communication

- Allows overlapping of data transfers and computations
  - Technique for latency hiding
- Non-blocking operations initiate a communication call and then return control to the application immediately
- Operation completed locally by making a call to the `wait` routine
- Non-blocking for all variants of Put, Get, and Acc

```c
ARMCI_Nbget(void* src, void* dst, int bytes, int proc,
            armci_nbhdl_t* nbhandle)

ARMCI_Wait(armci_nbhdl_t* nbhandle)
```
Implicit Handles

- Simplifies non-blocking handle management
- Call non-blocking calls with NULL handle
  - An implicit handle is allocated
- Wait on all implicit handles, or those to a specific process

```
ARMCI_Nbget(src, dst, bytes, proc, NULL)
ARMCI_Waitproc(proc)
ARMCI.Waitall()
```
Aggregate handles

- Group many small communications into a large one
  - Improves bandwidth when non-latency sensitive

- Approach:
  - Set a non-blocking handles as aggregate
  - Perform a number of non-blocking calls with that handle
  - Communication is completed upon wait().

```c
ARMCI_SET_AGGREGATE_HANDLE(nbhandle)
ARMCI_Nbget(src, dst, bytes, proc, nbhandle)
...
ARMCI_Wait(nbhandle)
ARMCI_UNSET_AGGREGATE_HANDLE(nbhandle)
```
Rmw

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.

Atomic operations
- ARMCI_FETCH_AND_ADD -> int
- ARMCI_FETCH_AND_ADD_LONG -> long
- ARMCI_SWAP -> int
- ARMCI_SWAP_LONG -> long

Remote pointer allocated with ARMCl_Malloc()

```c
int ARMCl_Rmw(int op, void *ploc, void *prem,
               int value, proc)
```
ARMCI Processor Groups

- Collections of processors (e.g. the world group) can be decomposed into processor groups
- Processor groups only affect global (collective) operations
- RDMA operations (Put/Get/Acc variants) are always on world process ids (same as ranks in MPI_COMM_WORLD)
ARMCI Group Operations

Basic operations

- ARMCI_Group_create, ARMCI_Group_destroy
- ARMCI_Malloc_group, ARMCI_Free_group
- ARMCI_Group_rank, ARMCI_Group_size
- ARMCI_Absolute_id
Group Management

- Create and destroy groups
- Provide list of process ranks (from world group)

```c
ARMCI_Group_create(int n, int *rank_list, ARMCI_Group *gout)
ARMCI_Group_free(ARMCI_Group *g)
```
Group Memory Allocation

- Create and destroy RDMA memory in a group
- Similar to calls to (de-)allocate on all processes
  - Additional group argument

```c
ARMCIMalloc_group(void *ptr_arr[], armci_size_t bytes,
                   ARMCI_Group *g)

ARMCFree_group(void *ptr, ARMCI_Group *g)
```
Managing Group Ranks

- A processes rank and size in a group
  - ARMCI_Group_rank(ARMCI_Group *g, int *rank)
  - ARMCI_Group_size(ARMCI_Group *g, int *size)

- Translate group rank of a process into rank in world group (same as MPI ranks)
  - ARMCI_Absolute_id(ARMCI_Group *g, int rank)
Summary

- Version 1.5 available
- Supported platforms
  - Linux clusters with Infiniband, GigE, Quadrics, Myrinet networks
  - IBM Bluegene/P, Bluegene/L, SP
  - Cray XT
  - Fujitsu
  - NEC
  - Windows
- Ongoing development
  - IBM Bluewaters
  - Cray Baker (Gemini interconnect)