PCSE Lecture 10

MPI One-Sided Communications

Cyrus Proctor
Victor Eijkhout

April 7, 2015

One-Sided Communications

- One-sided communication functions provide an interface to Remote Memory Access (RMA) communication methods
- RMA communications allow a single MPI process to initiate communication activity on both the sending and receiving side
- Regular send/receive communications require matching MPI_Send and MPI_Recv operations
- RMA may allow processes to avoid making costly barrier-type calls reducing the synchronization overhead
- RMA can help with system overhead as well. That is, actual time it takes to move the data from one process to another
- Allows MPI implementers to take advantage of low-latency, fast communication paths, possibly directly accessing the memory of another process

RMA Windows

RMA is provided through three different communication calls:

- MPI_Put (remote write)
- MPI_Get (remote read)
- MPI_Accumulate (remote update)

For any of these one-sided calls, we can choose among three different synchronization methods, each of which has its own syntax and requirements. RMA actions utilize the following fundamental paradigm:

1. Globally initialize a window for communication
2. Start an RMA epoch (synchronization)
3. Perform communication calls as desired
4. Stop an RMA epoch (synchronization)
5. Free window and any other resources

Window Creation

Before RMA communication can take place, all processes must agree on the areas in their local memories that remote processes can operate on. This window is created by a collective call that is executed by all processes in the given communicator.

C	int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
Fortran	MPI_WIN_CREATE(base, size, disp_unit, info, comm, win, ierr)

- **base** – initial address of window (offset)
- **size** – size of window in bytes
- **disp_unit** – local unit size for displacements, in bytes (hetero. env.)
- **info** – handle to an MPI_Info object
- **comm** – handle to a communicator
- **win** – handle to the window created by the call (out)

Simply creating the window does not automatically make its data accessible to other processes; the window must also be "opened" by a synchronization call.

Freeing a Window

- Once all communication calls for a given window are complete, processes should call MPI_Win_free to free up the window object
- A synchronization call must be made before this call may be made

C	int MPI_Win_free(MPI_Win *win)
Fortran	MPI_WIN_FREE(win, ierr)

- **win** – handle to the window (created by MPI_Win_Create)

In between the window creation and freeing operations, we can use RMA communication calls.

RMA Communication Calls

- The three RMA communication calls supported by MPI are MPI_Put, MPI_Get, and MPI_Accumulate
- All of these operations are non-blocking, which means that the call initiates the transfer, but transfer may begin or continue after the call returns
- A synchronization call is required to ensure that the transfer has completed
- The programmer needs to verify when it is safe to use or modify buffers

MPI_Get

MPI_Get allows the calling process to retrieve data from another process, as long as the desired data are contained within the target window and the copied data fits in the origin (calling side) buffer.

C	MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
Fortran	MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierr)

- **origin_addr** – address of the buffer to receive the data
- **origin_count** – the number of entries in the origin buffer
- **origin_datatype** – the datatype of each entry
- **target_rank** – the rank of target (where data will be retrieved from)
- **target_disp** – displacement from target window start to beginning of the target buffer (target offset)
- **target_count** – number of entries in the target buffer
- **target_datatype** – datatype of each entry in target buffer
- **win** – the window object

MPI_Put

MPI_Put transfers data from the caller to a target window. It has the same restrictions as MPI_Get.

C	MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
Fortran	MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierr)

- **origin_addr** – address of the send buffer
- **origin_count** – the number of entries in the origin buffer
- **origin_datatype** – the datatype of each entry
- **target_rank** – the rank of target (where data will be placed)
- **target_disp** – displacement from target window start to beginning of the target buffer (target offset)
- **target_count** – number of entries in the target buffer
- **target_datatype** – datatype of each entry in target buffer
- **win** – the window object

MPI_Accumulate

MPI_Accumulate allows the caller to combine the data moved to the target process with data already present, such as accumulation of a sum at a target process.

C	MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
Fortran	MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, ierr)

- **origin_addr** – address of the send buffer
- **origin_count** – the number of entries in the origin buffer
- **origin_datatype** – the datatype of each entry
- **target_rank** – the rank of target (where data will be placed)
- **target_disp** – displacement from target window start to beginning of the target buffer (target offset)
- **target_count** – number of entries in the target buffer
- **target_datatype** – datatype of each entry in target buffer
- **op** – predefined reduction operation
- **win** – the window object

The allowed operations are those provided by MPI_Reduce (max, min, sum, product, and the various and/or/xor operations).

RMA Synchronization Calls

For a given window, two synchronization calls are generally necessary:

- One to begin a communication epoch
- One to resolve all the communications that were initiated after the first call, ending the epoch

Notice that epochs are defined **per window**:

- Any individual process may participate in several access epochs simultaneously
- Provided that each epoch is associated with a different window

In the next two sections, we will introduce two classes of RMA Synchronization:

- Active Target Synchronization
- Passive Target Synchronization

Active Target Synchronization – Fence

- Fence synchronization is the simplest RMA synchronization pattern
- All processes call MPI_Win_fence at the start and end of epoch
- When MPI_Win_fence is called to terminate an epoch, the call will block until all RMA operations originating at that process complete

C	int MPI_Win_fence(int assert, MPI_Win win)
Fortran	MPI_WIN_FENCE(assert, win, ierr)

- **assert** – a way for the programmer to provide context to the call
  - MPI_MODE_NOSTORE – local window was not updated by local stores since last sync
  - MPI_MODE_NOPUT – local window will not be updated by put or accumulate calls after the fence call
  - MPI_MODE_NOPRECEDE – fence does not complete any sequence of locally issued RMA calls
  - MPI_MODE_NOSUCCEED – fence does not start any sequence of locally issued RMA calls
- **win** – handle to the window created by MPI_Win_create

Active Target Synchronization Example

- Populate a buffer on process 0
- Open a window
- Start with a fence call to sync
- Have every other process use MPI_Get to read buffer on process 0
- End with a fence call to sync
- Free the window

Active Target Synchronization Example

C Version – MPI_Win_Fence

```
  ...
  if(rank == 0){
    // Everyone will retrieve from a buffer on process 0
    buf = 42; size = intsize; displacement = intsize;
    MPI_Info_create(&info);
  }
  else {
    // Others only retrieve, so these windows can be size 0
    buf = 0; size = 0; displacement = intsize; info = MPI_INFO_NULL;
  }
  // Print buf before RMA calls
  // Create a Window for RMA calls
  MPI_Win_create(&buf,size,displacement,info,MPI_COMM_WORLD,&win);
  // No local operations prior to this epoch, so give an assertion
  MPI_Win_fence(MPI_MODE_NOPRECEDE,win);
  if(rank != 0){
    target_rank = 0; target_disp = 0; target_cnt = intsize;
    // Inside the fence, make RMA calls to GET from rank 0
    MPI_Get(&buf,1,MPI_INT,
            target_rank,target_disp,target_cnt,
            MPI_INT,win);
  }
  // Complete the epoch - this will block until MPI_Get is complete
  MPI_Win_fence(0,win); // Using no assertions
  // Do more work?
  // All done with the window - tell MPI there are no more epochs
  MPI_Win_fence(MPI_MODE_NOSUCCEED,win);
  // Free up our window
  MPI_Win_free(&win);
  // Print buf after RMA calls
  MPI_Finalize();
}
```

Active Target Synchronization Example

Fortran Version – MPI_Win_Fence

```
...
! Determine the size of an integer
call MPI_TYPE_EXTENT(MPI_INTEGER, intsize, ierr)
if(rank == 0)then
  ! Everyone will retrieve from a buffer on root
  buf = 42; my_size = intsize; displacement = intsize
  call MPI_INFO_CREATE(info, ierr)
else
  ! Others only retrieve, so these windows can be size 0
  buf = 0; my_size = 0; displacement = intsize
  info = MPI_INFO_NULL
end if
! Print buf before RMA calls
! Create a Window for RMA calls
call MPI_WIN_CREATE(buf,my_size,displacement,info, &
                    MPI_COMM_WORLD,win,ierr)
! No local operations prior to this epoch, so give an assertion
call MPI_WIN_FENCE(MPI_MODE_NOPRECEDE,win,ierr)
if(rank .ne. 0)then
  target_rank = 0; target_disp = 0; target_cnt = intsize
  ! Inside the fence, make RMA call to GET from rank 0
  call MPI_GET(buf,1,MPI_INTEGER, &
               target_rank,target_disp,target_cnt, &
               MPI_INTEGER,win,ierr)
end if
! Complete the epoch - this will block until MPI_GET is complete
call MPI_WIN_FENCE(0,win,ierr) ! Using no assertions
! All done with the window - tell MPI there are no more epochs
call MPI_WIN_FENCE(MPI_MODE_NOSUCCEED,win,ierr)
! Free up our window
call MPI_WIN_FREE(win,ierr)
...
```

Active Target Synchronization Example

C Output

```
(Rank 0): buf: 42
(Rank 1): buf: 0
(Rank 2): buf: 0
(Rank 3): buf: 0

Using MPI_Win_create and MPI_Get
     to fill buf from 0...

(Rank 0): buf: 42
(Rank 1): buf: 42
(Rank 2): buf: 42
(Rank 3): buf: 42
```

Fortran Output

```
 (Rank           0 ): buf:           42
 (Rank           1 ): buf:            0
 (Rank           2 ): buf:            0
 (Rank           3 ): buf:            0

Using MPI_WIN_CREATE and MPI_GET
     to fill buf from 0

 (Rank           0 ): buf:           42
 (Rank           1 ): buf:           42
 (Rank           2 ): buf:           42
 (Rank           3 ): buf:           42
```

Passive Target Synchronization – Lock/Unlock Request

- Permit access to a target by only one process at a time
- caller (origin process) obtains a lock (which may be shared or exclusive) to the window on a specific target
- Lock call may not block while the lock is actually being acquired
- Unlock only returns when all communication calls have completed

C	int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
Fortran	MPI_WIN_LOCK(lock_type, rank, assert, win, ierr)
C	int MPI_Win_unlock(int rank, MPI_Win win)
Fortran	MPI_WIN_UNLOCK(rank, win, ierr)

- **lock_type** – whether other processes may access the target window at the same time
  - MPI_LOCK_SHARED – Multiple processes (as long as none hold MPI_LOCK_EXCLUSIVE)
  - MPI_LOCK_EXCLUSIVE – One process at a time may access
- **rank** – rank of target window to acquire lock for
- **assert** – assertions for optimization
- **win** – the window context for the calls

Passive Target Synchronization Example

- Populate a buffer on process 0
- Open a window
- Start with a lock call to target process 1
- Process 0 uses MPI_Put to write buffer to process 1
- End with a unlock call
- Free the window

Passive Target Synchronization Example

C Version – MPI_Win_lock/unlock

```
// Start up MPI...
if(rank == 0){
    // Rank 0 will be the caller
    MPI_Win_create(buf,0,1,
        MPI_INFO_NULL,MPI_COMM_WORLD,&win);
    // Request lock of process 1
    lock_type = MPI_LOCK_SHARED;
    target_rank = 1; target_disp = 0; target_cnt = 1;
    MPI_Win_lock(lock_type,target_rank,0,win);
    MPI_Put(buf,1,MPI_INT,target_rank,target_disp,target_cnt,
            MPI_INT,win);
    // Block until put succeeds
    MPI_Win_unlock(target_rank,win);
    // Free the window
    MPI_Win_free(&win);
}
else{
    // Rank 1 is the target process
    MPI_Win_create(buf,2*sizeof(int),sizeof(int),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win);
    // No sync calls on the target process!
    MPI_Win_free(&win);
}
```

Passive Target Synchronization Example

Fortran Version – MPI_Win_Fence

```
! Start up MPI...
if(rank == 0)then
    ! Rank 0 will be the caller
    call MPI_WIN_CREATE(buf,0,1, &
        MPI_INFO_NULL,MPI_COMM_WORLD,win,ierr)
    ! Request lock of process 1
    lock_type = MPI_LOCK_SHARED
    target_rank = 1; target_disp = 0; target_cnt = 1
    call MPI_WIN_LOCK(lock_type,target_rank,0,win,ierr)
    call MPI_PUT(buf,1,MPI_INT,target_rank,target_disp,target_cnt, &
                 MPI_INT,win,err)
    ! Block until put succeeds
    call MPI_WIN_UNLOCK(target_rank,win,ierr)
    ! Free the window
    call MPI_WIN_FREE(win,ierr)
else
    ! Rank 1 is the target process
    call MPI_WIN_CREATE(buf,2*intsize,intsize, &
        MPI_INFO_NULL, MPI_COMM_WORLD, win,ierr)
    ! No sync calls on the target process!
    call MPI_WIN_FREE(win,ierr)
end if
```

References

- Victor Eijkhout, “Introduction to High Performance Scientific Computing”
- Victor Eijkhout, “Parallel Computing for Science and Engineering”
- Steve Lantz, “MPI One-Sided Communication”
- Mark Lubin, “Introduction into new features of MPI-3.0 Standard”
- “MPI: A Message-Passing Interface Standard Version-3.0”