One-sided Communication with MPI-2

Rolf Rabenseifner
rabenseifner@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Acknowledgements
This course is based on the “One-sided” chapter of the MPI-2 tutorial on the MPIDC 2000:

MPI-2: Extensions to the Message Passing Interface

Anthony Skjellum¹
Purushotham Bangalore¹, Shane Hebert¹
Rolf Rabenseifner²
Synchronization Taxonomy

Message Passing:
- explicit transfer, implicit synchronization, implicit cache operations

Access to other processes’ memory:
- **1-sided**
  - explicit transfer, explicit synchronization, implicit cache operations (problem!)
- Shared Memory
  - implicit transfer, explicit synchronization, implicit cache operations
- shmem interface
  - explicit transfer, explicit synchronization, explicit cache operations

Cooperative Communication

- MPI-1 supports cooperative or 2-sided communication
- Both sender and receiver processes must participate in the communication
One-sided Communication

- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses

Origin Process

Target Process

- put

- get

One-sided Operations

- Initialization
  - MPI_ALLOC_MEM, MPI_FREE_MEM
  - MPI_WIN_CREATE, MPI_WIN_FREE
- Remote Memory Access (RMA, nonblocking)
  - MPI_PUT
  - MPI_GET
  - MPI_ACCUMULATE
- Synchronization
  - MPI_WIN_FENCE (like a barrier)
  - MPI_WIN_POST / MPI_WIN_START / MPI_WIN_COMPLETE / MPI_WIN_WAIT
  - MPI_WIN_LOCK / MPI_WIN_UNLOCK
Window Creation

- Specifies the region in memory (already allocated) that can be accessed by remote processes
- Collective call over all processes in the intracommunicator
- Returns an opaque object of type MPI_Win which can be used to perform remote memory access (RMA) operations

```
MPI_WIN_CREATE(base_address, win_size, disp_unit, info, comm, win)
```

MPI_Put

- Performs an operation equivalent to a send by the origin process and a matching receive by the target process
- The origin process specifies the arguments for both the origin and target process
- The target buffer is at address target_addr = win_base + target_disp * disp_unit

```
MPI_PUT(origin_address, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)
```

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!
MPI_Get

- Similar to the put operation, except that data is transferred from the target memory to the origin process
- To complete the transfer a synchronization call must be made on the window involved
- The local buffer should not be accessed until the synchronization call is completed

\[
\text{MPI\_GET( origin\_address, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, win)}
\]

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

MPI_Accumulate

- Accumulates the contents of the origin buffer to the target area specified using the predefined operation \( \text{op} \)
- User-defined operations cannot be used
- Accumulate is atomic: many accumulates can be done by many origins to one target
  \( \rightarrow \) [may be very expensive]

\[
\text{MPI\_ACCUMULATE(origin\_address, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, op, win)}
\]

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!
Synchronization Calls

- **Active target communication**
  - Communication paradigm similar to message passing model
  - Target process participates only in the synchronization
  - Fence or post-start-complete-wait

- **Passive target communication**
  - Communication paradigm closer to shared memory model
  - Only the origin process is involved in the communication
  - Lock/unlock

**MPI_Win_fence**

- Synchronizes RMA operations on specified window
- Collective over the window
- Like a barrier
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation
- Used for active target communication

`MPI_WIN_FENCE(assert, win)`
Start/Complete and Post/Wait, I.

- Used for active target communication with weak synchronization

![Diagram showing one-sided communication with MPI-2]

Start/Complete and Post/Wait, II.

- RMA (put, get, accumulate) are finished
  - locally after `win_complete`
  - at the target after `win_wait`
- Local buffer must not be reused before RMA call locally finished
- Communication partners must be known
- No atomicity for overlapping “puts”
- Assertions may improve efficiency
  --> give all information you have

![Diagram showing additional one-sided communication with MPI-2]
Start/Complete and Post/Wait, III.

- symmetric communication possible, only win_start and win_wait may block

![Diagram showing communication flow between processes 0 and 1 involving win_post, win_start, put, win_complete, win_wait, and load operations.]

Lock/Unlock

- Does not guarantee a sequence
- agent may be necessary on systems without (virtual) shared memory
- Portable programs can use lock calls to windows in memory allocated only by MPI_ALLOC_MEM
- RMA completed after UNLOCK at both origin and target

![Diagram illustrating lock, unlock, and synchronization operations between Origin1, Origin2, and Target processes.]

---

12. — One-sided Communication with MPI-2 — 12. 12-8
MPI_ALLOC_MEM

MPI_ALLOC_MEM (size, info, baseptr)
MPI_FREE_MEM (base)

REAL A
POINTER (P, A(100)) ! no memory is allocated
INTEGER (KIND=MPI_ADDRESS_KIND) Size
INTEGER Lng_real, Win, IERR
CALL MPI_TYPE_EXTENT(MPI_REAL, Lng_real, IERR)
Size = 100*Lng_real
CALL MPI_ALLOC_MEM(Size, MPI_INFO_NULL, P, IERR)
CALL MPI_WIN_CREATE(A, Size, Lng_real,
  MPI_INFO_NULL, MPI_COMM_WORLD, Win, IERR)
... CALL MPI_WIN_FREE(Win, IERR)
CALL MPI_FREE_MEM(A, IERR)

Fortran Problems with 1-Sided

- Fortran register optimization
- Result ccc=999, but expected ccc=777
- How to avoid: (see MPI-2, Chap. 6.7.3)
  - window memory declared in COMMON blocks
    i.e. MPI_ALLOC_MEM cannot be used
  - declare window memory as VOLATILE
    (non-standard, disables compiler optimization)
  - Calling MPI_Address(buff, idummy_addr, ierror) after 2nd FENCE in process 2

Source of Process 1
bbbbb = 777
call MPI_WIN_FENCE
call MPI_PUT(bbbb into buff of process 2)
call MPI_WIN_FENCE

Source of Process 2
buff = 999
call MPI_WIN_FENCE
call MPI_WIN_FENCE
ccc = buff
ccc := register_A

Executed in Process 2
register_A := 999
stop application thread
buff := 777 in PUT handler
continue application thread
One-sided: Summary

- Three one-sided communication primitives provided
  - put / get / accumulate
- Several synchronization options supported
  - fence / post-start-complete-wait / lock-unlock
- User must ensure that there are no conflicting accesses
- For better performance assertions should be used with fence/start/post operations

MPI–One-sided Exercise 1: Ring communication with fence

- Copy to your local directory:
  
  ```
  cp ~/MPI/course/C/1sided/ring.c my_1sided_exa1.c
  cp ~/MPI/course/F/1sided/ring.f my_1sided_exa1.f
  ```

- Tasks:
  - Substitute the non-blocking communication by one-sided communication. Two choices:
    - either rcv_buf = window
      - MPI_Win_fence - the rcv_buf can be used to receive data
      - MPI_Put - to write the content of the local variable snd_buf into the remote window (rcv_buf)
      - MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled
    - or snd_buf = window
      - MPI_Win_fence - the snd_buf is filled
      - MPI_Get - to read the content of the remote window (snd_buf) into the local variable rcv_buf
      - MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled
  - Compile and run your my_1sided_exa1.c / .f
ring.c / .f: Rotating information around a ring

Initialization:
1. Each iteration:
   2. 3. 4. 5.

   to be substituted by 1-sided comm.

MPI–One-sided Exercise 1: additional hints

- **MPI_Win_create**:
  - base = reference to your rcv_buf or snd_buf variable
  - disp_unit = number of bytes of one int / integer, because this is the datatype of the buffer (=window)
  - size = same number of bytes, because buffer size = 1 value
  - size and disp_unit have different internal representations, therefore:
    - C: `MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL, ..., &win);`
    - Fortran: `INTEGER disp_unit`
      
      `INTEGER (KIND=MPI_ADDRESS_KIND) winsize`
      
      `CALL MPI_TYPE_EXTENT(MPI_INTEGER, disp_unit, ierr)`
      
      `winsize = disp_unit * 1`
      
      `CALL MPI_WIN_CREATE(rcv_buf, winsize, disp_unit, MPI_INFO_NULL, ..., ierr)`

- see MPI-2, page 110
MPI–One-sided Exercise 1: additional hints

- MPI_Put or MPI_Get:
  - target_disp
    - C: `MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);`
    - Fortran: `INTEGER (KIND=MPI_ADDRESS_KIND) target_disp`
      - `target_disp = 0`
      - `CALL MPI_PUT(snd_buf, 1, MPI_INTEGER, right, target_disp, 1, MPI_INTEGER, win, ierror)`
  - see MPI-2, page 116

MPI–One-sided Exercise 2: Post-start-complete-wait

- Use your result of exercise 1 or copy to your local directory:
  ```
  cp ~/MPI/course/C/1sided/ring_1sided.c my_1sided_exa2.c
  cp ~/MPI/course/F/1sided/ring_1sided.f my_1sided_exa2.f
  ```
- Tasks:
  - Substitute the two calls to MPI_Win_fence
    by calls to MPI_Win_post / _start / _complete / _wait
  - Use to group mechanism to address the neighbors:
    - MPI_COMM_GROUP(comm, group)
    - MPI_GROUP_INCL(group, n, ranks, newgroup)
    - MPI_COMM_CREATE(comm, group, newcomm)
      - do not forget ierror with Fortran!
    - Fortran: `integer comm, group, newgroup, newcomm, n, ranks(...)`
    - C: `MPI_Comm comm, newcomm; MPI_Group group, newgroup; int n, ranks[...];`
  - Compile and run your `my_1sided_exa2.c / .f`