MPI Fundamentals

ONE-SIDED COMMUNICATION ROUTINES

Thanh-Dang Diep
High Performance Computing Laboratory
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

Email: dang@hcmut.edu.vn

August 16, 2016

1 Introduction

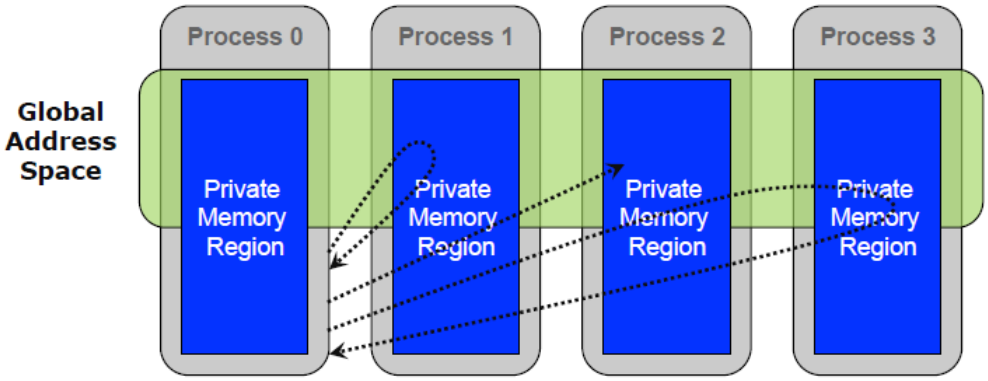

Figure 1: MPI One-sided Communication

The basic idea of one-sided communication models is to decouple data movement with process synchronization:

- Should be able to move data without requiring that the remote process synchronize.
- Each process exposes a part of its memory to other processes.
- Other processes can directly read from or write to this. memory.

2 Initialization Routines

2.1 MPI_Win_create

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
                   MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

- Expose a region of memory in an RMA window
  - Only data exposed in a window can be accessed with RMA operations
- Arguments:

  base initial address of window (choice)

  size size of window in bytes (non-negative integer)

  disp_unit local unit size for displacements, in bytes(positive integer)

  info info argument (handle)

  comm intra-communicator (handle)

  win window object returned by the call (handle)

2.2 MPI_Win_free

```
int MPI_Win_free(MPI_Win *win);
```

- Free the window object **win** and return a null handle.
- Arguments:

  win window object (handle)

Example 1:

```
int main(int argc, char **argv)
{
    int *a;
    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000 * sizeof(int), MPI_INFO_NULL, &a);
    /* use private memory like you normally would */
    a[0] = 1;
    a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000 * sizeof(int), sizeof(int),
       MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array 'a' is now accessibly by all processes in
       MPI_COMM_WORLD */

    MPI_Win_free(&win);
    MPI_Free_mem(a);
    MPI_Finalize();
    return 0;
}
```

3 Communication Routines

3.1 MPI_Put

```
int MPI_Put(const 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);
```

- Move data from origin to target
- Arguments:

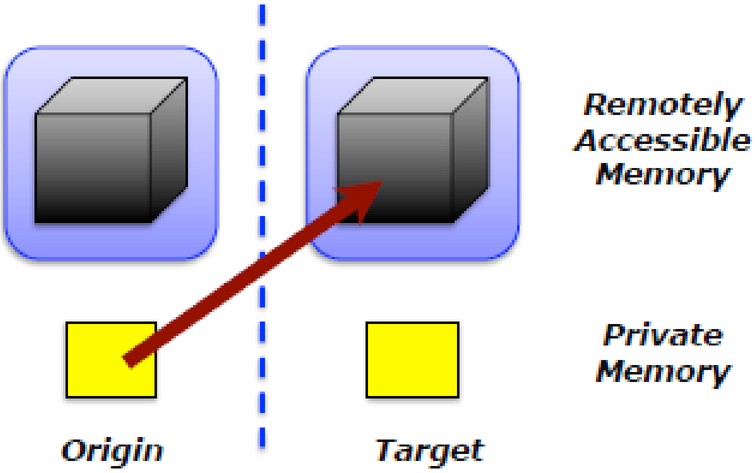

Figure 2: Communication using Put

origin_addr initial address of origin buffer (choice)

origin_count number of entries in origin buffer (non-negative integer)

origin_datatype datatype of each entry in origin buffer (handle)

target_rank rank of target (non-negative integer)

target_disp displacement from start of window to target buffer (non-negative integer)

target_count number of entries in target buffer (non-negative integer)

target_datatype datatype of each entry in target buffer (handle)

win window object used for communication (handle)

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        int size, rank, localbuf, sharedbuf;
        MPI_Win win;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
        MPI_Win_create(&sharedbuf, 1, sizeof(int), MPI_INFO_NULL
            , MPI_COMM_WORLD, &win);

        localbuf = rank;

        printf("[%d] localbuf = %d\n", rank, localbuf);

        MPI_Win_fence(0, win);
        MPI_Put(&localbuf, 1, MPI_INT, (rank + 1) % size, 0, 1,
            MPI_INT, win);
        MPI_Win_fence(0, win);

        printf("[%d] sharedbuf = %d\n", rank, sharedbuf);

        MPI_Win_free(&win);

        MPI_Finalize();
        return 0;
}
```

3.2 MPI_Get

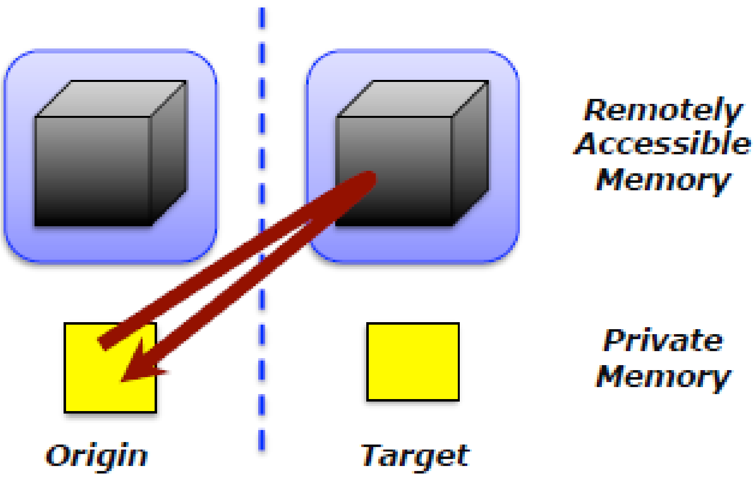

Figure 3: Communication using Get

```c
int 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);
```

- Move data from target to origin
- Arguments:

  origin_addr initial address of origin buffer (choice)

  origin_count number of entries in origin buffer (non-negative integer)

  origin_datatype datatype of each entry in origin buffer (handle)

  target_rank rank of target (non-negative integer)

  target_disp displacement from window start to the beginning of target buffer (non-negative integer)

  target_count number of entries in target buffer (non-negative integer)

  target_datatype datatype of each entry in target buffer (handle)

  win window object used for communication (handle)

```c
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        int size, rank, localbuf, sharedbuf;
        MPI_Win win;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Win_create(&sharedbuf, 1, sizeof(int), MPI_INFO_NULL
            , MPI_COMM_WORLD, &win);

        sharedbuf = rank;

        printf("[%d] sharedbuf = %d\n", rank, sharedbuf);
```

```
        MPI_Win_fence(0, win);
        MPI_Get(&localbuf, 1, MPI_INT, (rank + 1) % size, 0, 1,
           MPI_INT, win);
        MPI_Win_fence(0, win);

        printf("[%d] localbuf = %d\n", rank, localbuf);

        MPI_Win_free(&win);

        MPI_Finalize();
        return 0;
}
```

3.3 MPI_Accumulate

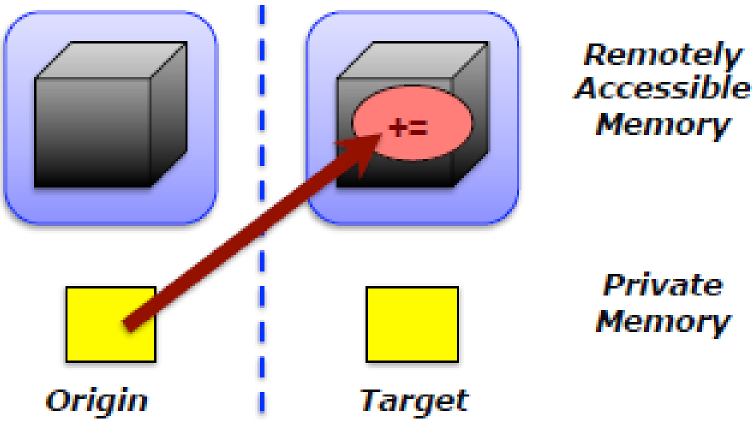

Figure 4: Communication using Accumulate

```
int MPI_Accumulate(const 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);
```

- Element-wise atomic update operation, similar to a put
    - Reduces origin and target data into target buffer using **op** argument as combiner

  - Predefined operations only, no user-defined operations

- Arguments:

  origin_addr initial address of origin buffer (choice)

  origin_count number of entries in origin buffer (non-negative integer)

  origin_datatype datatype of each entry (handle)

  target_rank rank of target (non-negative integer)

  target_disp displacement from start of window to the beginning of target buffer (non-negative integer)

  target_count number of entries in target buffer (non-negative integer)

  target_datatype datatype of each entry in target buffer (handle)

  op reduce operation (handle)

  win window object (handle)

4 Synchronization Routines

Access epoch: contain a set of operations issued by an origin process

Exposure epoch: enable remote processes to access and/or update a target's window

4.1 Fence (Active target)

```
int MPI_Win_fence(int assert, MPI_Win win);
```

- Collective synchronization model
- Starts and ends access and exposure epochs on all processes in the window
- All processes in group of "win" do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All operations complete at the second fence synchronization

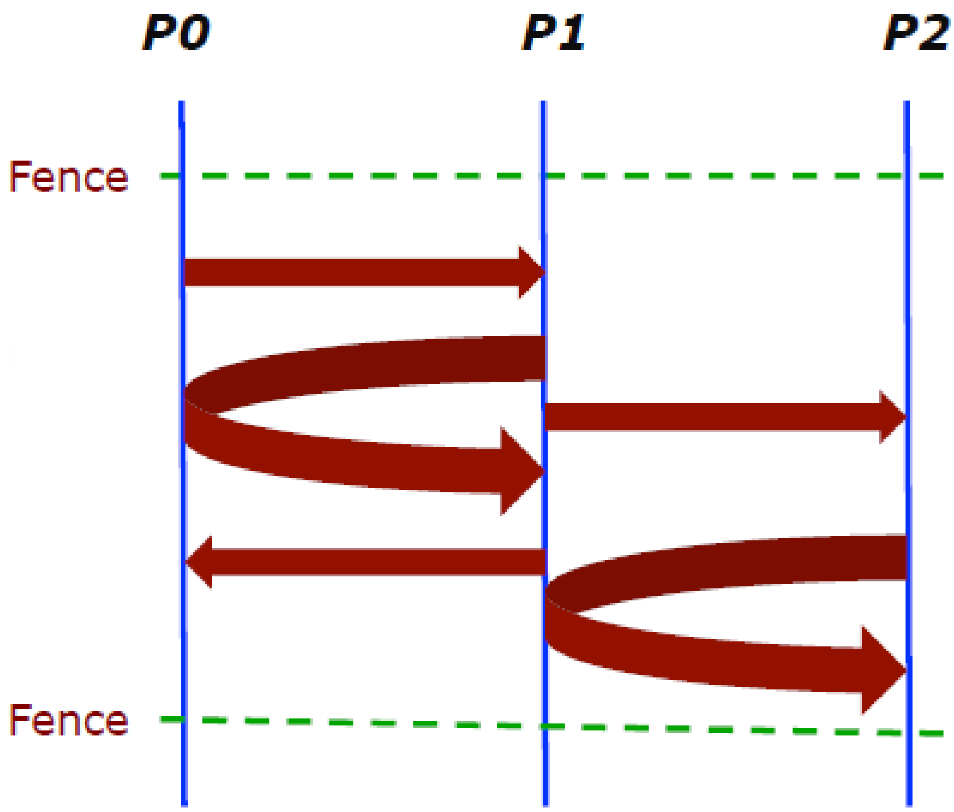

Figure 5: Synchronization using Fence

- Arguments:

  assert program assertion (integer)

  win window object (handle)

4.2 Post-start-complete-wait (Generalized active target)

4.2.1 MPI_Win_start

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win);
```

- Origin opens access epoch
- Arguments:

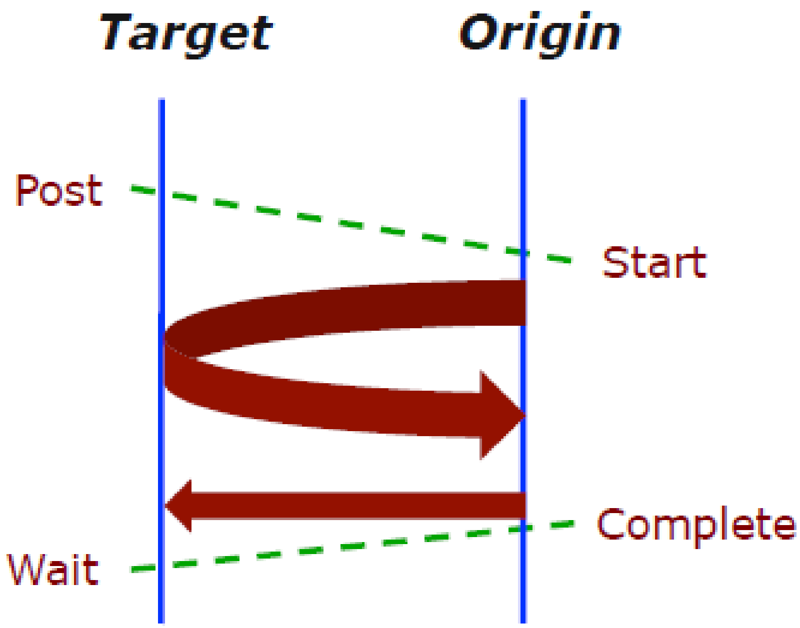

Figure 6: Synchronization using PSCW

group group of target processes (handle)

assert program assertion (integer)

win window object (handle)

4.2.2 MPI_Win_complete

```
int MPI_Win_complete(MPI_Win win);
```

- Origin closes access epoch
- Arguments:

  win window object (handle)

4.2.3 MPI_Win_post

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win);
```

- Target open expose epoch

- Arguments:

  group group of origin processes (handle)

  assert program assertion (integer)

  win window object (handle)

4.2.4 MPI_Win_wait

```
int MPI_Win_wait(MPI_Win win);
```

- Target closes exposure epoch
- Arguments:

  win window object (handle)

4.3 Lock/unlock (Passive Target)

4.3.1 MPI_Win_lock

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);
```

- Begin an RMA access epoch at the target process
- Arguments:

  lock_type either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)

  rank rank of locked window (non-negative integer)

  assert program assertion (integer)

  win window object (handle)

4.3.2 MPI_Win_unlock

```
int MPI_Win_unlock(int rank, MPI_Win win);
```

- Complete an RMA access epoch at the target process
- Arguments:

  rank rank of window (non-negative integer)

  win window object (handle)

Exercises

1. Write a program that prints out prime numbers in the first one billion of positive integers.

2. Write a program that sums up prime numbers in the first one billion of positive integers.

3. Write a program that computes the value of Pi using Monte Carlo simulation. The program samples points inside the rectangle delimited by (0,0) and (1,1) and counts how many of these are within a circle with a radius of 1. The ratio between the number of points inside the circle and the total number of samples is Pi/4.

4. Write a program multiplying two square matrices whose sizes of each are 1000x1000 and 10000x10000.