Remote Memory Operations

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Materials are taken from the book, Using MPI-2: Advanced Features of the Message-Passing Interface by William Gropp, Ewing Lusk, and Rajeev Thakur.

Topic Overview

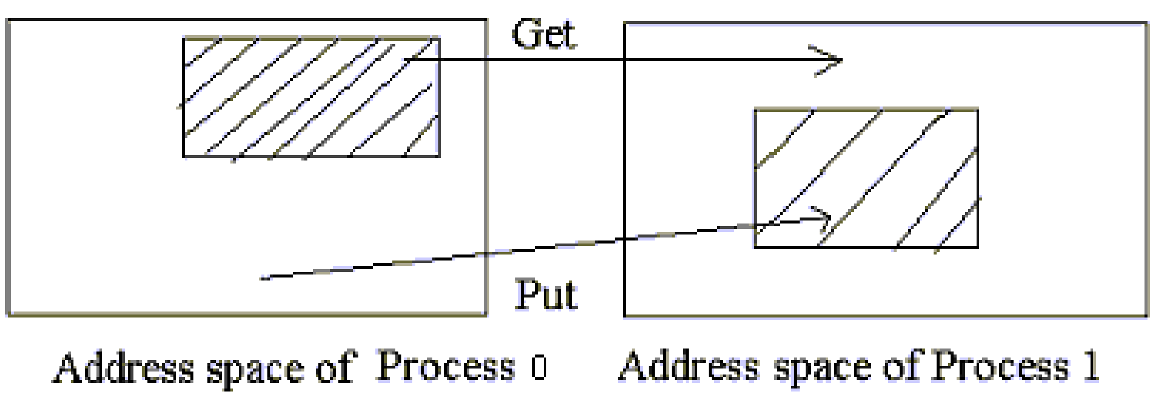
- Introduction to Remote Memory Operations
- A Non-Remote Memory Operation code
- Remote Memory Operations

Introduction to Remote Memory Operations

- Send-Recv:
  Specify what memory locations can be modified.
  Specify when this can happen.

- Advantages:
  1. Ensuring the correctness of the data.
  2. Changes in memory of the receiver can happen only when the receiver allows.

- Disadvantages:
  1. Expressiveness and low performance.
  2. What if data that is updated frequently.
  3. Communication costs might be big.



Remote Memory Access (RMA)

- One single routine that specifies both where the data is coming from and where it is going to.

- They are called one-sided communication routines.

- Three main steps when using the RMA.

  1. Define the memory of the process that can be used for RMA operations.
     `MPI_Win_create`

  2. Specify the data to be moved and where to move it.
     `MPI_Put, MPI_Get, MPI_Accumulate`

  3. Specify how we know that the data is available.
     `MPI_Win_fence`

A Non-RMA code

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx.$$

- Compute the Riemann sum.

```
h = 1.0 / (double) n;
/* n is the number of partition over [0,1] */
sum = 0.0; /* local sum */
for (i = myid + 1; i <= n; i += numprocs) {
x = h * ((double)i - 0.5);
  sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum; /* partial Riemann sum */
/* reduce all partial_sum to cpu0 */
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
if (myid == 0)
  printf("pi is approximately %.16f, Error is %.16f\n",
         pi, fabs(pi - PI25DT));
```

Remote Memory Access (RMA)

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

- `*base`: Initial address of window.
- `size`: Size of window in bytes.
- `disp_unit`: Local unit size for displacements, in bytes.
  Unit size in different processors maybe different.
- `info`: Info argument.
- `*win`: Window object returned by the call.

- Need to be called by all processors in comm which contribute to this window.

- A "window" refers to a region of memory within a single process.

- A "window object" describes the collection of windows that are the input to the MPI_Win_create call.

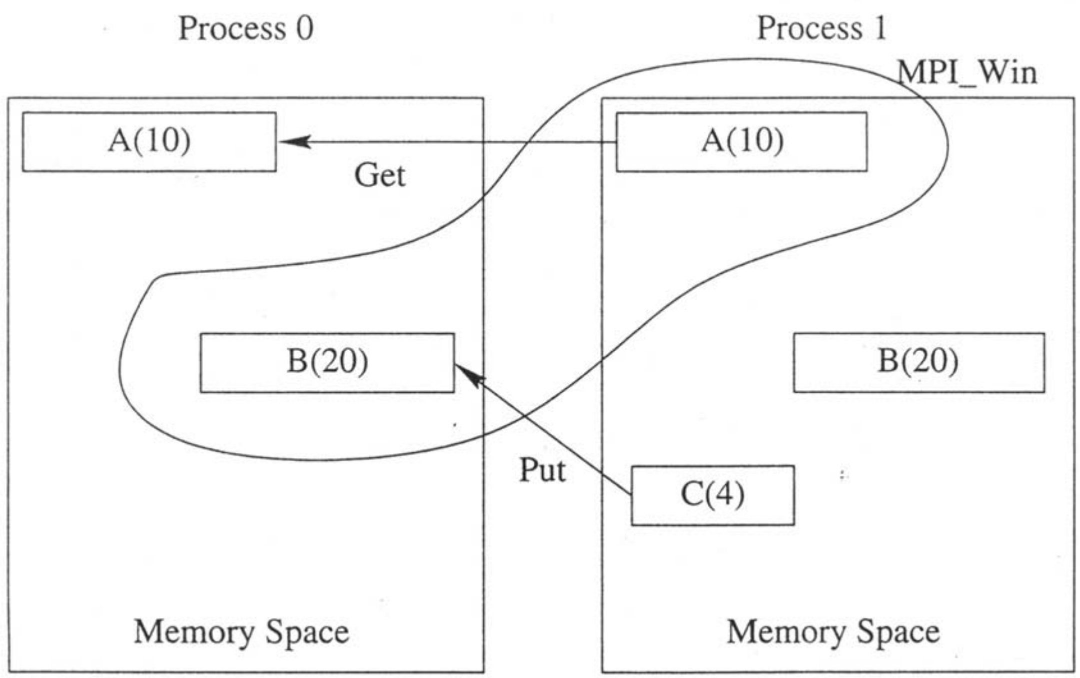
Remote Memory Access (RMA)

```
if (myid == 0){
  MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL,
                             MPI_COMM_WORLD, &nwin);
  MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL,
                             MPI_COMM_WORLD, &piwin);
}
else{
  MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                             MPI_COMM_WORLD, &nwin);
  MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                             MPI_COMM_WORLD, &piwin);
}
```

- nwin consists one integer from cpu0, none from the rest.
- piwin consists one double from cpu0, none from the rest.

Remote Memory Access (RMA)–MPI_Win_create

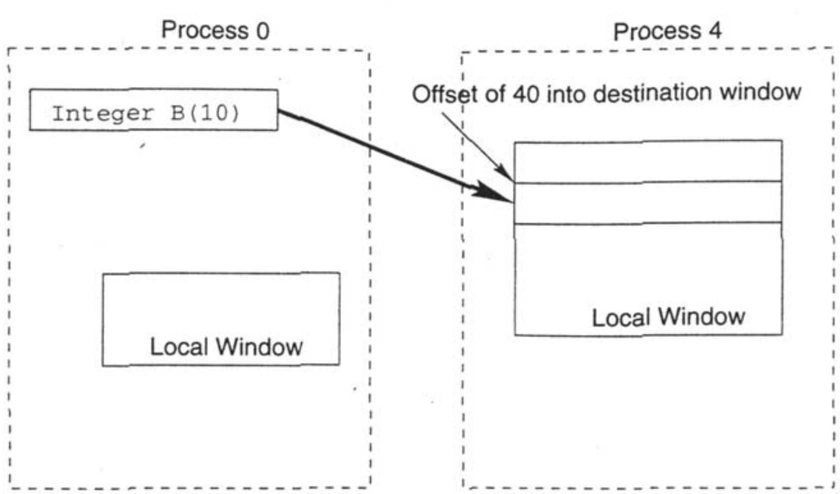
```
if (myid == 0){
    MPI_Win_create(B, 20*sizeof(int), sizeof(int), MPI_INFO_NULL,
     MPI_COMM_WORLD, &win);}
if (myid == 1){
    MPI_Win_create(A, 10*sizeof(int), sizeof(int), MPI_INFO_NULL,
     MPI_COMM_WORLD, &win);}
```



Remote Memory Access (RMA)–MPI_Put

```
int 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)
-------------------------------------------------------------------
if (myrank == 0) then
  MPI_Put(B, 10, MPI_INT, 4, 40, 10, MPI_INT, win);
```

- `*origin_addr`: Initial address of origin buffer.
- `origin_count`: Number of entries in origin buffer.
- `target_rank`: Rank of target.
- `target_disp`: Displacement from window start to the beginning of the target buffer.
- `target_count`: Number of entries in target buffer.



Remote Memory Access (RMA)–MPI_Get

```
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)
```

- `*origin_addr`: Address of the buffer in which to receive the data.
- `origin_count`: Number of entries in origin buffer.
- `target_rank`: Rank of target.
- `target_disp`: Displacement from window start to the beginning of the target buffer.
- `target_count`: Number of entries in target buffer.

Remote Memory Access (RMA)–MPI_Win_fence

- What if two processors try to access the same data at the same time?

- Answer: Use MPI_Win_fence.

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

- `assert`: Program assertion. Provide information about the fence that can be used by some MPI implementation to provide better performance.
- `win`: Window object.

- MPI_Win_fence completes any RMA operations that started since the last call to MPI_Win_fence.

- MPI_Win_fence is collective over all processes in the group associated with the window object.

Remote Memory Access (RMA)

- The easy rules:
  1. Do not overlap accesses on window.
  2. Separate non-RMA access from RMA access with MPI_Win_fence.

- It is possible to have several MPI window objects whose local windows overlap.

- BUT, because of the performance issues and the complexities of the rules for correct use, we recommend avoiding the use of overlapping window.

- MPI_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)
```

Remote Memory Access (RMA)

```
if (myid == 0){
  printf("Enter the number of intervals: (0 quits) ");
  printf("\n");
  scanf("%d",&n);
  pi = 0.0;
}
MPI_Win_fence(0, nwin);
```

```
if (myid != 0)
   MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
MPI_Win_fence(0, nwin);

if (n == 0)
   break;
else{
   h = 1.0 / (double) n;
   sum = 0.0;
   for (i = myid + 1; i <= n; i += numprocs){
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));}

   mypi = h * sum;

   MPI_Win_fence( 0, piwin);
   MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE,
                        MPI_SUM, piwin);
   MPI_Win_fence(0, piwin);
}
```

Remote Memory Access (RMA)–Example

```
MPI-Win_create( A, ..., &win);
MPI_Win_fence( 0, win);

if (rank == 0)
* Process 0 puts data into many local windows */
   MPI_Put( ... , win );
/* This fence completes the MPI_Put operations initiated
   by process 0 */
MPI_Win_fence( 0, win );
/* All processes initiate access to some window to extract data */

MPI_Get( ... , win );
/* The following fence completes the MPI_Get operations */
MPI_Win_fence( 0, win );

/* After the fence, processes can load and store
into A, the local window */
A[rank] = 4;
printf( "A[%d] = %d\n", 0, A[0] );
MPI_Win_fence( 0, win );

/* We need a fence between stores and RMA operations */
MPI_Put( ... , win );
/* The following fence completes the preceding Put */
MPI_Win_fence( 0, win );
```

Remote Memory Access (RMA)

- Homework: Use RMA to compute variance of n numbers.
- Homework: Use RMA to implement the Jacobi method for solving linear system.