MPI-2

Remote memory access

Introduction

So far in the course, all exchange of data between processes was done via MPI calls on both sides of the communication.

Example:

```
Process 0           Data flow          Process 1

MPI_Send(...)         ------->         MPI_Recv(...)

MPI_Allreduce()     <------->          MPI_Allreduce()
```

Having to put two calls in the code for every communication can be inconvenient in some situations. It adds complexity to the code.

On the other hand, managing communication explicitly like this increases reliability, so most of MPI code out there still uses this paradigm.

In some situations, doing communications via just one call will offer advantages. With some careful preparation, this is possible, via functions in MPI-2.

Remote memory access (RMA)

MPI-2 offers functions which allow for one-sided communication, or remote memory access

Example:

```
Process 0                    Data flow       Process 1

(collective preparation)                     (collective preparation)

MPI_Get(...)                 <-------

MPI_Put(...)                 ------->

                             ------->        MPI_Get(...)

                             <-------        MPI_Put(...)


(collective termination)                     (collective termination)
```

Why is this useful?

Consider case of 2 processes, process 0 needs A[i] entry of array A[N] on process 1, where i generated randomly

Process 0	Process 1
int Ai;	int A[N];
i=rand()%N;	(fill A[N] with some data)
/* get Ai via standard MPI_Send and MPI_Recv */	
MPI_Send(&i,...);	MPI_Recv(&i,...);
MPI_Recv(Ai,...);	MPI_Send(A[i],...);
/* now get Ai via RMA */	
(preparation)	(preparation: make A[N] visible)
MPI_Get(Ai,...)	(do some computation)
(termination)	(termination: make A[N] invisible)

Conclusion: we can get Ai via one function call with RMA, vs. 4 function calls via standard method. RMA is simpler! Also, process 1 can do something else while process 0 is getting data.

Preparation stage

In order to allow Remote Memory Access, a process must select a contiguous region of memory and open it to the other processes. This accessible region is then called a window.

Other processes (in the communicator) must know about the window.

MPI accomplishes this by a collective function MPI_Win_create (create window)

As it is a collective call, per implementation every process involved must open a window. However, if a process does not need to share any memory, its window may be defined to be of size zero. (size of the window may be different on each process calling MPI_Win_create)

The window object that is the output of this function may be used for Remote Memory Access operations.

MPI_Win_create

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

base - initial address of window
size - size of window in bytes
disp_unit - local unit size for displacements, in bytes
info - info argument
comm - communicator
win - window object returned by call (output)
```

Remote Memory Access stage

A process can get data from remote memory via MPI_Get

It can write data to remote memory via MPI_Put

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
origin_datatype - datatype of each entry 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
target_datatype - datatype of each entry in target buffer
win - window object used for communication
```

MPI_Put

```
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 - initial address of origin buffer
origin_count - number of entries in origin buffer
origin_datatype - datatype of each entry 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
target_datatype - datatype of each entry in target buffer
win - window object used for communication

Ensure completion

MPI_Put and MPI_Get are non-blocking operations so we will need a mechanism to ensure they complete before using the data.

Also, need a mechanism to ensure all the processes are done with the window before the process that owns it closes it or modifies the data in it.

We must be sure the data transfers of MPI_Get and MPI_Put complete before the data is used or modified.

This is accomplished via function MPI_Win_Fence

MPI_Win_fence

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

assert - (input) program assertion (integer)

win - Window object
```

This function synchronizes RMA calls on win. The function is collective on the group of win.

All RMA operations on win originating at process calling this function will complete before function returns.

They will be completed at target before function returns at target.

RMA operations started by a process after MPI_Win_fence call will actually access target process’s memory only after the target completed corresponding MPI_Win_fence

MPI_Win_free

```
int MPI_Win_create(MPI_Win *win)

win - window object (handle).  Object is freed by this call.
```

```
#include "mpi.h"
#include "stdio.h"

/* This does a transpose with a get operation, fence, and derived
    datatypes. Uses vector and hvector.  Run on 2 processes */

#define NROWS 100
#define NCOLS 100

int main(int argc, char *argv[])
{
    int rank, nprocs, A[NROWS][NCOLS], i, j;
    MPI_Win win;
    MPI_Datatype column, xpose;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (nprocs != 2) {
        printf("Run this program with 2 processes\n");fflush(stdout);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
```

```
if (rank == 0)
{
    for (i=0; i<NROWS; i++)
        for (j=0; j<NCOLS; j++)
            A[i][j] = -1;

    /* create datatype for one column */
    MPI_Type_vector(NROWS, 1, NCOLS, MPI_INT, &column);
    /* create datatype for matrix in column-major order */
    MPI_Type_hvector(NCOLS, 1, sizeof(int), column, &xpose);
    MPI_Type_commit(&xpose);
```

```
    MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    MPI_Win_fence(0, win);

    MPI_Get(A, NROWS*NCOLS, MPI_INT, 1, 0, 1, xpose, win);

    MPI_Type_free(&column);
    MPI_Type_free(&xpose);

    MPI_Win_fence(0, win);
```

```
/* check data transferred correctly */
     for (j=0; j<NCOLS; j++){
         for (i=0; i<NROWS; i++){
             if (A[j][i] != i*NCOLS + j){

  printf("Error: A[%d][%d]=%d ne %d\n", j, i, A[j][i], i*NCOLS + j);
  fflush(stdout);
             }
         }
     }
}
```

```
    else
    { /* rank = 1 */
        for (i=0; i<NROWS; i++)
            for (j=0; j<NCOLS; j++)
                A[i][j] = i*NCOLS + j;

MPI_Win_create(A, NROWS*NCOLS*sizeof(int), sizeof(int), MPI_INFO_NULL,
MPI_COMM_WORLD, &win);

        MPI_Win_fence(0, win);
        MPI_Win_fence(0, win);
    }

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

One more useful RMA function

Quite often, the need is to send the value to some remote memory location, add it to the value already stored there, and then store the resulting sum at the remote location

Useful in a reduce operation when all processes are sending a value to process 0 via RMA and the goal is to sum up all these values (or do another reduce operation: min, max etc.) .

This functionality is provided by MPI_Accumulate

Almost like MPI_Put, except with reduce operation provided as additional argument

MPI_Accumulate

```
int 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)

origin_addr - initial address of origin buffer
origin_count - number of entries in origin buffer
origin_datatype - datatype of each entry 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
target_datatype - datatype of each entry in target buffer
op - predefined reduce operation
win - window object used for communication
```

What if multiple processes are trying to access the same region of shared memory?

Must then use locks:

```
MPI_Win_lock(...)
MPI_Get(...)
MPI_Win_unlock(...)
...

MPI_Win_lock(...)
MPI_Put(...)
MPI_Win_unlock(...)
```

To avoid problems like this, can choose approach were for each open window, only one process can access. Even then, may have to use locks if a sequence of non-independent MPI_Put and MPI_Get operations is performed on it