MA/CSSE Homework 5
Due 4/21

Directions

- **Each problem must be self-contained in a file**, named properly, and **must compile using the command** `mpicc filename.c`
- Turn in each `.c` file to the dropbox. **Do not create a `.zip` file**
- On this homework you may not use any of the standard default MPI collective operations.

Problems

Level: Easy

1. Create a function `my_scatter` with declaration

   ```
   void my_scatter(int* sendbuf, int sendcount, int* recvbuf, int recvcount, int root,
                   MPI_Comm comm)
   ```

   that re-creates some of the functionality of `MPI_Scatter`. Your code must follow the simple algorithm of repeatedly sending from the root to every non-root process. The root should not send to itself, but should rather copy the data into the correct buffer. You may not use `MPI_Scatter`. Call the file `my_scatter.c`

2. Create a function `my_broadcast` with declaration

   ```
   void my_broadcast(int* buffer, int count, int root, MPI_Comm comm)
   ```

   that re-creates some of the functionality of `MPI_Bcast`. Call the file `my_broadcast.c`. Your code may assume that the number of processors on the communicator is a power of 2. Your code may assume that the root is 0. Your code should follow the binary tree algorithm from class. You may not use `MPI_Bcast`.

3. Create a function `my_allgather` with declaration

   ```
   void my_allgather(int* sendbuf, int sendcount, int* recvbuf, int recvcount,
                     MPI_Comm comm)
   ```

   that reproduces some of the functionality of `MPI_Allgather`. You may not use `MPI_Allgather`. In this code, each process should do $p - 1$ sends and $p - 1$ receives, where p is the number of processors engaged.

Level: Difficult

1. Improve the `my_gather` function we wrote in class so that it does not assume that the number of processors is a power of 2, and it does not assume that the root is rank 0. Your algorithm should follow the binary tree algorithm discussed in class, but treating `root` as rank 0, `root+1` as rank 1, and so on. Name the file `my_gather.c`

2. Create a function `my_allplus` with declaration

```
int my_allplus(int* sendbuf, int* recvbuf, int count,
               MPI_Comm comm)
```

   that puts the sum of all the data held in each processors `sendbuf` array on every process. For example, if there are four processes p_0,p_1,p_2,p_3 and p_i holds data $d_{i,0}$,$d_{i,1}$,$d_{i,2}$ then when the processes complete the call to `my_allplus` each processor should find

$$d_{0,0} + d_{0,1} + d_{0,2} + d_{1,0} + d_{1,1} + d_{1,2} + d_{2,0} + d_{2,1} + d_{2,2} + d_{3,0} + d_{3,1} + d_{3,2}$$

   available in `recvbuf[0]`.

   No processor should do more than $\lceil \log p \rceil$ receives and $\lceil \log p \rceil$ sends, and no send or receive should be longer than one integer.

Errata

What does the testing code do, and how to interpret the results

The testing code compiles your function (`my_all_gather`, my_scatter or whatever), and compiles them along with a `main` function that I have written. It calls your function with a number of different inputs, and makes sure that your code is behaving correctly. The command that I'm using to compile your code is shown first. For example:

```
mpicc --std=c99 -g -lm -Wl,-wrap,MPI_Recv -Wl,-wrap,MPI_Send
-Wl,-wrap,main my_alltoall.c \
test_helpers.c my_alltoall_test_driver.c mpi_send_wrapper.c -o my_alltoall_exe
```

You can see that I'm just calling `mpicc` to compile your code into a bunch of code that I've written. You can compile your code exactly the same way I am, if you want. Of course, then you would be using my `main` rather than the one you wrote.

You can see the arguments that my `main` is giving your functions in each test case. For example, here is the output of one test case for `my_alltoall`

```
Using the command: mpirun -np 2 ./my_alltoall_exe --len=10
Rank 1 calls my_alltoall with:
sendbuff:[1,346,381,486,301]
sendcount:5
recvbuff:[-1939952792,32728,26006624,0,26004688,0,25434272,0,1,0]
recvcount:5

Rank 0 calls my_alltoall with:
sendbuff:[0,243,272,259,420]
sendcount:5
recvbuff:[-222065816,32616,13234448,0,13233520,0,12691280,0,1,0]
recvcount:5

Rank 0 finishes my_alltoall with:
recv_buff: [0,243,272,259,420,1,346,381,486,301]
```

```
Rank 1 finishes my_alltoall with:
recv_buff: [0,243,272,259,420,1,346,381,486,301]
```

We can see that rank 0 has an initial `sendbuf` of `[1,346,381,486,301]` and rank 1 has an initial sendbuf of `[0,243,272,259,420]`. The code completes, and after calling `my_alltoall` both ranks have `[0,243,272,259,420,1,346,381,486,301]` in their `recv_buff`.

Once the executable runs, my code examines some logs to make sure that your code executed in a reasonable fashion (the correct number of sends, receives, etc).

If you wish to debug a certain test case, you can just hard-code in the inputs into your own main, and call your function with the same inputs that my code is using.

Using the standard code

I have provided some standard implementations of the assigned problems. In order to use them, you will need to replace your calls to `foo` with `foo_standard`. For example, if you want to see how my `my_alltoall` function behaves, just replace your calls to `my_alltoall` with calls to `my_alltoall_standard`.

When you compile your code, you will need to add in the object files provided in the `obj` directory.

For example, you would switch from compiling `my_alltoall` like this:

```
mpicc my_alltoall.c -o my_alltoall
```

to like this:

```
mpicc my_alltoall.c ./objs/my_alltoall_standard.x86_64.o -o my_alltoall
```

The compiler will warn you about "implicit declaration" when you compile, but the overall compilation will succeed.

If you are using your own linux distribution, or a newer VM than the one I provided the class, or you are on grendel, use the object files with the extension `x86_64.o`. If you are using the VM provided for the class, use the object files with the extension `i686.o`.