SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Lab 5: Non-Blocking Messages

Due: 4:00 PM, Thursday, October 3, 2024

In this lab, we'll look at MPI's non-blocking point-to-point communication.

You may work alone or with a partner on this lab.

Learning goals:

1. To learn how to do error checking on MPI calls.
2. To learn about non-blocking point-to-point communication in MPI.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `nb-lab-yourgitname`, for this lab. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

You may answer the lab questions right in the `README.md` file of your repository, or use the `README.md` to provide a link to a Google document that has been shared with your instructor or the name of a PDF of your responses that you would upload to your repository.

Error Checking with MPI

The `mpimsg` directory of your repository contains a program similar to the `mpiexchange` program you wrote recently. This one demonstrates a few things beyond the earlier examples.

- All MPI calls return a status value, and it's a good idea to check it as is done in this example for the `MPI_Init` call.
  - Most class examples will not be thorough in this to keep things looking simpler.
  - For our purposes, any MPI error will cause the program to terminate with an error message, so it usually is not that important to us.
  - When developing large-scale software, especially reusable libraries, it is often very helpful return error codes rather than crash the whole program, so error checking becomes more important there.
  - The error checking includes a messy little chunk of code to print out appropriate messages, so it's probably worth putting this into your own error reporting function if you want to use it.

- `MPI_Status status` - structure which contains additional info following a receive. We often ignore it (we passed in `MPI_STATUS_IGNORE` in earlier examples), but we will see some instances where it comes in handy.

Modify the program so the `MPI_Recv` uses a different "from" parameter.

Question 1: What happens and why? (2 points)

Put that back and now modify the program so the `MPI_Recv` uses a different message tag parameter.

Question 2: What happens and why? (2 points)

Undo that change also to get back to the original version. (Tip: you can also issue the command `git checkout mpimsg.c` to revert to the last committed version).

Now, let's experiment with using `MPI_ANY_SOURCE` on our `MPI_Recv` call. That parameter allows the receive call to match an `MPI_Send` from any rank process, as long as the other parameters still match. We can also do the same for the message tag by specifying `MPI_ANY_TAG`. Replace the "from" and tag parameters in the `MPI_Recv` call with `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively, and try the program.

Question 3: What happens and why? (2 points)

In this case, we can be sure of where the message came from and what the tag was since there's only one other process and it only sent the one message. But if we are unsure, we can use the status parameter to find out. Notice that we have a variable of type `MPI_Status` declared, and that we pass a pointer to it as the last parameter to `MPI_Recv`. MPI will fill in the details of the received message in the three fields of the `MPI_Status` structure: `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`.

Practice Program: Modify the program so it uses `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, and print the source and tag from the `status` variable after the `MPI_Recv`. (4 points)

In general, it's good for efficiency purposes to avoid forcing processes to do their work in any particular order except for when there is a good reason for doing so. We will see many of those reasons soon. Let's look back at the `mpi_trap2` program from the text's examples, a copy of which is in the `mpi_trap2` directory of your repository. When the rank 0 process receives the messages with the partial answers from all of the other processes, the order in which they're received and processed doesn't really matter. As written, it must receive the messages in order by sender rank. However, suppose a that a process with a low rank, for whatever reason, takes longer than the others to finish its work. Messages that are already sent and which could be processed wait until all messages from lower-rank senders have been received. The messages and the work done to process the results received are very small here, so this doesn't matter much, but in some cases, it could.

Practice Program: Modify the `mpi_trap2` program to use `MPI_ANY_SOURCE`, and add a printout after the `MPI_Recv` to print the solution value of each message received, and the rank of the process that sent it. (4 points)

Question 4: Run with larger numbers of processes and larger n values until you see the messages being received in an order other than increasing rank of sender. Paste your output as the answer to this question. (2 points)

Non-Blocking Point-to-point Communication

Recall again the `mpiexchange.c` program you wrote for the previous lab. Suppose we wanted that program to be generalized to work for more processes. That is, each process picks and number to send to the process with the next highest rank (except that with the highest rank, which sends it to 0). Then each process will receive that value, modify it in some way, then send it back. Those messages then need to be received and printed.

The directory `mpiring` contains the programs we'll use in this section.

The program `mpiring_danger.c` does just this. We can compile and run it on any number of processes and it will likely work.

However, it is not guaranteed to work, since an `MPI_Send` call will not necessarily return until there is a corresponding `MPI_Recv` call to receive the message. If each process performs the first `MPI_Send` and those calls cannot complete until the `MPI_Recv`, we can enter a *deadlock* condition, where each process is waiting for something to happen in some **other** process before it can continue execution.

Question 5: Run this program several times on noreaster for each of 2, 8, and 32 processes. Do any fail to run successfully to completion? (2 points)

The chances of this problem increase with larger messages.

The program `mpiring_danger_large.c` sends arrays in the same pattern as the previous program sent single `int` values. The `#define` at the top of the program determines how large these messages are.

Question 6: Run the program repeatedly with 8 processes on noreaster. Start with a message size of 16. It should run to completion. Double the message size, recompile, and re-run until the program no longer produces output. That will indicate a deadlock situation. What message size is the first that leads to deadlock in this program? (5 points)

These kinds of communication patterns are very common in parallel programming, so we need a way to deal with them. MPI provides another variation on point-to-point communication that is *non-blocking*. These are also known as *immediate mode* sends and receives. These functions are named `MPI_Isend` and `MPI_Irecv`, and will always return immediately.

The program `mpiring_safe_large.c` uses non-blocking sends to remove the danger of deadlock in this program. Please see the comments there about the need to wait for the message to be

delivered (or at least to have it in progress) before the send buffer can be modified. This example uses `MPI_Wait` to accomplish that.

Question 7: Verify that this works by repeating the above experiment to see how large your messages can be before an error occurs. Eventually, you will run into trouble with the size of the arrays on the stack. At what message size does this problem occur, and what error do you see? (5 points)

Practice Program: While it is not necessary for the correctness of the program in this case, we could also use non-blocking receives (`MPI_Irecv`) in `mpiring_safe_large.c`. Modify the program to use `MPI_Irecv`, adding any other needed variables and function calls to achieve this. (5 points)

Submission

Commit and push! Make sure your answers to lab questions are provided using one of the mechanisms mentioned in the "Getting Set Up" part of the lab.

Grading

This assignment will be graded out of 35 points.

Feature	Value	Score
Question 1	2	
Question 2	2	
Question 3	2	
`mpimsg` any source/tag	4	
`mpitrap2` any source	4	
`mpitrap2` ordering	2	
Question 4	2	
Question 5	2	
Question 6	5	
Question 7	5	
`mpiring` nb receive	5	
Total	35	