CMSC416: Introduction to Parallel Computing

Topic: Advanced MPI: Non-blocking calls
Date: February 13, 2024

We began class with announcements (assignment 0, assignment 1, and scribing).

Basic MPI_Send and MPI_Recv Example Program
The memory in 0 and 2 is passed to the memory in process 1 and 3 (as shown in the illustration).

MPI communicators
Once you have the default communicator, you can create sub-communicators. Most common use is to create smaller sub-communicators.
For example, with game of life example, suppose you want to do some operations within each column or row. In this case, you can create 3 sub-communicators for the processes in each row. The 1st sub-communicator takes processes numbered 0-2, the 2nd sub-communicator takes processes numbered 3-5, and the 3rd sub-communicator takes processes numbered 6-8.

MPI datatypes
MPI datatypes are derived or user-defined. Some predefined data types are MPI_INT, MPI_CHAR, and MPI_DOUBLE. You can also send an array of elements of another datatype or custom struct data type to accommodate sending multiple data types together.

Advanced MPI
We've seen blocking MPI calls like send and receive. A downside to blocking calls is that the performance is impacted because the processes are waiting and can't run additional code until unblocked.

Non-blocking point-to-point calls:
MPI_Isend and MPI_Irecv refer to the non blocking calls of send and receive. Because we want to do other stuff while we are blocked for a message, we can run other code and then wait for results at a later point using MPI_Wait or MPI_Test to serve as the blocking calls.

MPI_Isend
MPI_Isend is for the most part similar to MPI_SEND. The main difference is there is a MPI_Request object which is the communication request object. When you send the same MPI request object with MPI_Wait, MPI will know what to wait for.

MPI_Wait
You provide the same MPI_Request object for the corresponding MPI_Isend/MPI_IRecv, and a MPI_Status object. The MPI_Status object provides information about the number of received entries and the message's source, tag, and error.

EXAMPLE: Using non-blocking send/recv
For the most part the example looks the same as the previous MPI_Send and MPI_Recv example, but using MPI_Isend and MPI_Irecv instead. If you are in a sending process and you aren't sure when you can use the data, you can use MPI_Wait. For the most part, MPI_Wait is less important for MPI_Isend because the data could already be read by the time you wait. With MPI_Irecv, you should MPI_Wait before using data.

During the example, there was a question with the request object. When using MPI_Isend and MPI_Irecv, you will have to create a request object for each call. For example, with 3 MPI_Irecv calls, you will need 3 different request objects.

When using blocking vs non-blocking send or receive requests, it should not impact the correctness of the program. Blocking vs non-blocking should only impact the performance of the program.

2D Stencil computation example
In this example, suppose we are doing a wrap-around.
Q: How many processes is each process communicating with?
A: Each process is communicating with 2 other processes. For example, in process 1, the process needs information from process 0 and 2. Process 1 has to send its top and bottom rows to Processes 0 and 2 respectively. Also, Process 1 has to receive Process 0's bottom row, and Process 2's top row.

From a performance perspective, it is better to call non-blocking receives before sends.

For sending rows in C or C++, you can point to the first element and send 16 consecutive elements because C and C++ are row-major, meaning the consecutive elements within the row are contiguous in memory. If you want to send columns, you would have to copy over the specific entries because the 4 columns are not contiguous in memory.

Before I start local computation, I need to wait for all non-blocking receives to get the data that I need.
Q: Do I need to wait for sends to complete?
A:
If you will overwrite the buffers, then YES! Otherwise, the buffers with the sends would be changed. As an alternative, you can copy the data over from sending buffers into other buffers, so you can continue computing.
Overall answer: Depends on whether you want to add extra overhead to copy over buffers to continue or use the same buffer and wait for the send to complete.

Q: In the 2D stencil example, do I need to wait for everything before computing?
A: No! You can do computation for inner rows because you have all the information you need. Then, you can wait for the top and bottom rows and do the corresponding computation afterwards.

Other MPI calls
You can use MPI_Test to check whether your sends and receives are completed. However, MPI_Test is NOT a blocking call and will return immediately.
Q: When can you use MPI_Test over MPI_Wait?
A: When you have a complex operation and can break down your computation to smaller chunks, you can use a while loop to do the smaller chunks that don't rely on the response. Then, once the send or receive is completed, we can immediately use the response. This is beneficial if we don't want to leave the send or receive to hang for too long.

You can use MPI_Waitall(...) to wait for everything and MPI_Waitsome(...) to wait for some.

Collective operations
In point to point communication, there are 2 specific points and operations you do between them.
In collective operations, there are more than 2 processes involved in the call.

MPI_Barrier is the first and simplest collective call. MPI_Barrier is a barrier or synchronizing call which blocks until all processes reach the barrier. If a process reaches the barrier early, it will wait until all processes reach the barrier.

MPI_Bcast is a broadcast routine. This is useful if you want to broadcast data from a root to all processes within a communicator. If you break MPI_COMM_WORLD into smaller groups, you can send to all processes within the communicator's group. Similar to a bunch of send and receive calls.