CAAM 520, Spring 2020 - Problem Set 3

Eappen Nelluvelil

April 3, 2020

1. *Non-blocking communication I*

   Suppose that an application uses two MPI processes in the following way: rank zero updates data which it then sends to rank one. Rank one then uses the data to perform additional work with it. In this case, rank zero might execute code similar to that shown below:

```
update_data(data1);
MPI_Send(data1, data_size, MPI_BYTE, 1, 999, MPI_COMM_WORLD);
update_data(data2);
MPI_Send(data2, data_size, MPI_BYTE, 1, 999, MPI_COMM_WORLD);
```

   (a) In your own words, describe what it means to overlap communication and computation and how the above code could benefit from it.

   A call to `MPI_Send` does not complete until the send buffer can be used again, and a call to `MPI_Recv` does not complete until the message has been received. In some circumstances, we do need our code to run with such strong guarantees. For example, in the following code,

```
do_work(data_A);
MPI_Send(data_A, ...);
do_work(data_B);
```

   we see that we do not re-use `data_A` before we call `do_work(data_B)`, so there is not a reason to wait until the `MPI_Send` call finishes until we can call `do_work(data_B)`. Another example can be seen in the following code:

```
MPI_Recv(data_B, ...);
do_work(data_A);
do_work(data_B);
```

   In the above code, we have to wait for the `MPI_Recv` call has to finish before `do_work(data_A)` is executed, but this call is not dependent on `data_B`. This means we can execute `do_work(data_A)` first before executing the `MPI_Recv` call.

   In the first example, we can overlap communication and computation by replacing the `MPI_Send` call with an `MPI_Isend` call. This allows us to execute `do_work(data_B)` without having to wait on `data_A` being re-usable again.

   Similarly, in the second example, we can overlap communication and computation by replacing the `MPI_Recv` call with an `MPI_Irecv`. These changes have the effect of improving the efficiency of the overall code.

   The original code updates `data1` on rank 0 and sends it to rank 1, then updates `data2` and sends it to rank 1. We see that `data1` is not re-used before we update `data2`, so we can overlap communication and computation.

(b) Modify the above code so that communication and computation are overlapped. As before, only show the code executed by rank zero.

We can overlap communication and computation (on rank 0) in the following manner:

```
MPI_Request request;
update_data(data1);
MPI_Isend(data1, data_size, MPI_BYTE, 1, 999, MPI_COMM_WORLD,
          &request);
update_data(data2);
MPI_Send(data2, data_size, MPI_BYTE, 1, 999, MPI_COMM_WORLD);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

(c) Now suppose that rank zero updates and sends the same data repeatedly, as shown below.

```
update_data(data);
MPI_Send(data, data_size, MPI_BYTE, 1, 999, MPI_COMM_WORLD);
update_data(data);
MPI_Send(data, data_size, MPI_BYTE, 1, 999, MPI_COMM_WORLD);
```

Is it still possible to overlap communication and computation using your code from part (b)?

No, it is not possible to overlap communication and computation using the code from part (b). This is because we are no longer sending and updating two separate data buffers, but the same data buffer.

2. *Non-blocking communication, II*

Recall that we introduced the function MPI_Sendrecv to avoid deadlocks in situations where MPI processes both send and receive a message.

(a) If executed using two MPI processes, can the code below result in a deadlock? Explain your answer.

```
int send, recv, rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
const int other = 1 - rank;
if (rank == 0) {
        MPI_Send(&send, 1, MPI_INT, other, 999, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, other, 999, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
}
else {
        MPI_Recv(&recv, 1, MPI_INT, other, 999, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Send(&send, 1, MPI_INT, other, 999, MPI_COMM_WORLD);
}
```

The above code cannot result in a deadlock. This is because the send and receive calls on the respective ranks are appropriately paired.

(b) If executed using two MPI processes, can the code below result in a deadlock? Explain your answer.

```
int send, recv, rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
const int other = rank - 1;
MPI_Send(&send, 1, MPI_INT, other, 999, MPI_COMM_WORLD);
MPI_Recv(&recv, 1, MPI_INT, other, 999, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

Yes, the above code can result in a deadlock if called with two MPI processes. Assuming the `MPI_Send` calls block until the send buffer is re-usable, the `MPI_Send` calls are never completed on their respective ranks. To avoid the issue, we can perform a send call then receive call on rank 0 and a receive call then send call on rank 1 (or vice versa), or use non-blocking sends on both ranks.

(c) Using non-blocking MPI communication, implement a function

```
void my_sendrecv(const void *sendbuf, int sendcount,
                                 MPI_Datatype sendtype, int dest,
                                 int sendtag, void *recvbuf,
                                 int recvcount, MPI_Datatype recvtype,
                                 int source, int recvtag,
                                 MPI_Comm comm, MPI_Status *status)
```

which acts like `MPI_Sendrecv` except that it does not return an error code. Make sure that your implementation returns the correct status object for the receive operation.

3. *One-sided communication I*

(a) In your own words, explain the need for MPI windows for one-sided communication. Specifically, explain why `MPI_Put()` and `MPI_Get()` describe memory locations using pointers to on the origin rank but integer displacements within a window on the target rank.

One-sided communication allows ranks to passively communicate with other ranks, e.g., instead of sending messages between rank `a` and `b`, we can read and write to rank `b`'s memory directly from rank `a`. To perform one-sided communication, we need to understand how memory is laid out on other ranks, and MPI windows allow us to do this in a convenient manner.

The function `MPI_Put` (analogous to an `MPI_Send`) allows us to write information from a memory location on the origin rank's memory (through a pointer on the origin rank's memory) to a memory location on another rank's memory via integer displacements. This makes it convenient to write to another rank's memory because we do not have to deal with byte sizes on another rank's memory, for example. This idea also applies to the function `MPI_Get` (analogous to `MPI_Recv`).

(b) Explain why the following code is incorrect and how to fix it.

```
MPI_Win_fence(0, win);
MPI_Get(&data, 1, MPI_INT, remote_rank, displacement, 1, MPI_INT, win);
do_work(data);
MPI_Win_fence(0, win);
```

The above code is incorrect because `MPI_Get` is called on all ranks, and each rank immediately works on the data. Each call to `MPI_Get` might not finish at the same time, so each rank needs to call `MPI_Win_fence` first before working on the data. A simple fix would be to interchange the last two lines of code.

(c) Suppose that the following code is executed using 1,000 MPI processes.

```
MPI_Win_fence(0, win);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
        // Get data from rank 123
        MPI_Get(&data, 1, MPI_INT, 123, displacement, 1, MPI_INT, win);
}
MPI_Win_fence(0, win);
```

Explain why the use of `MPI_Win_fence()` is inefficient in this example and propose a solution. Specifically, modify the code so that rank zero still makes the same call to `MPI_Get()` without calling `MPI_Win_fence()`.

The above code is inefficient because MPI_Win_fence forces a synchronization between all ranks even if some ranks did not participate in the communication, but only ranks 0 and 123 participate in the communication, so the MPI_Get call is not a collective.

To avoid this inefficiency, we can modify the above code in the following manner:

```
MPI_Group comm_group, group;
MPI_Comm_group(MPI_COMM_WORLD, &comm_group)

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
        // Form target group (consisting only of rank 123)
        MPI_Group_incl(comm_group, 1, {123}, &group);

        // Begin access epoch
        MPI_Win_start(group, 0, win);

        // Get data from rank 123
        MPI_Get(&data, 1, MPI_INT, 123, displacement, 1, MPI_INT, win);

        // End access epoch
        MPI_Win_complete(win)
}
```

4. *One-sided communication II*

   (a) See code.

   (b) See code.

   (c) In my opinion, it was easier to reason about the one-sided exchange than the standard exchange. Both approaches relied on every rank knowing every other rank's values array. On any rank, the standard exchange required boilerplate code to access the values array on every other rank, whereas the one-sided exchange made it very easy to access the values array on every other rank.