CAAM 520 – Homework 3

Due: 4/3/2020

General remarks: Please submit your homework report (as a PDF file) along with any code to your personal CAAM 520 Git repository. Please provide a makefile that can be used to compile your code, and make sure that your code compiles and runs in the CAAM 520 virtual machine.

Problem 1 *(Non-blocking communication I, 20 pts.)*

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. *(5 pts.)*

b) Modify the above code so that communication and computation are overlapped. As before, only show the code executed by rank zero. *(10 pts.)*

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)? *(5 pts.)*

Problem 2 *(Non-blocking communication II, 25 pts.)*

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);
}
```

(5 pts.)

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 = 1 - rank;
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);
```

(5 pts.)

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. *(15 pts.)*

Note: Implement your function in the file `sendrecv.c` which is provided as part of this homework assignment. As always, do not modify the `main()` function. Obviously, your implementation must not use `MPI_Sendrecv()` or `MPI_Sendrecv_replace()`. If you compile and run the code in `sendrecv.c`, it will automatically test your implementation.

Problem 3 *(One-sided communication I, 20 pts.)*

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. *(5 pts.)*

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);
```

(5 pts.)

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()`. *(10 pts.)*

Problem 4 *(One-sided communication II, 35 pts.)*

In class, we discussed that one-sided MPI communication can be convenient in applications with random or very complex communication patterns. To emulate such an application, consider the following algorithm: Each MPI process creates an array `values` which contains `num_values` random integer values. While the number of random values is the same on each MPI rank, the values themselves are different.

Next, each of the `comm_size` MPI ranks creates an array `indices` of `comm_size` random indices from `0` to `num_values - 1`. Finally, each rank creates an array `result` of length `comm_size`. Using MPI communication, it then fetches the `indices[r]`th value from rank `r` and places it in `result[r]`. Figure 1 shows an example of this algorithm. Note that the random values, indices, and results are *different* on each rank.

a) Implement the above algorithm in the function

```
void exchange_standard(const int *values,
                       const int *indices,
                       int *result,
                       int num_values,
                       int comm_size)
```

in the file `random_communication.c` which is provided as part of this homework assignment. You may use two-sided communication with `MPI_Send()`, `MPI_Isend()`, etc. as well as any collective operations. Do not use one-sided communication with `MPI_Put()`, `MPI_Get()`, etc. *(15 pts.)*

Note: If you compile and run the code in `random_communication.c`, it will automatically test your implementation.

b) Using one-sided MPI communication (`MPI_Put()`, `MPI_Get()`, etc.), implement the same algorithm as in a) in the `exchange_onesided()` function in `random_communication.c`. Do not use two-sided communication with `MPI_Send()`, `MPI_Isend()`, or any collective operations other than (possibly) `MPI_Win_fence()`. *(15 pts.)*

c) In your opinion, which of the two approaches was easier to implement and why? *(5 pts.)*

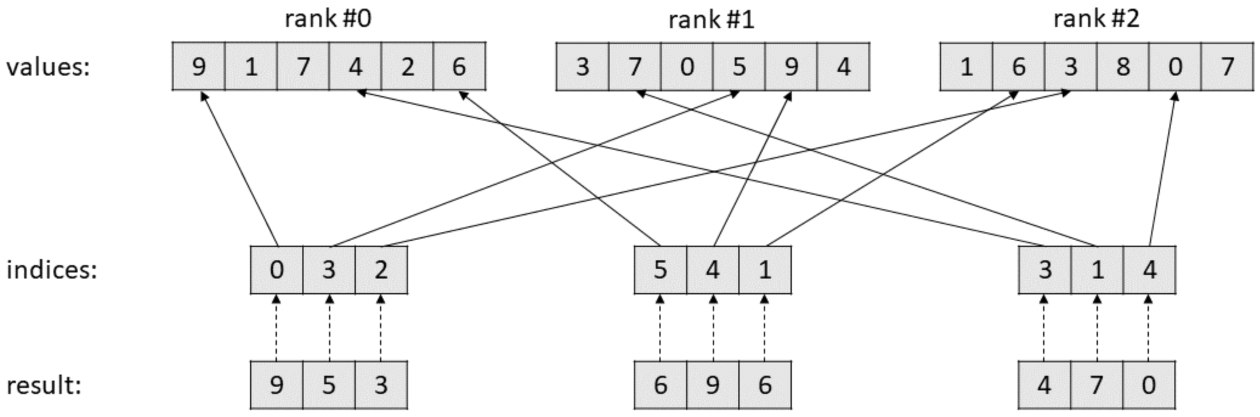

Figure 1: Example of the communication patterns in problem 4 for three MPI processes (`comm_size = 3`) and six random values per process (`num_values = 6`).