Exercise 10

Task 1 (`ring_nonblocking.c`)

Write an **MPI program** `ring_nonblocking.c` that implements a ring communication pattern using nonblocking send operations.

1. Each process should send its rank to the **right neighbor** (with wraparound: process $n-1$ sends to process 0).
2. Implement the ring communication using the **nonblocking send pattern** from the lecture:
   (a) Initiate a nonblocking send (`MPI_Isend`) to the right neighbor
   (b) Perform a blocking receive (`MPI_Recv`) from the left neighbor
   (c) Wait for the nonblocking send to complete (`MPI_Wait`)
3. After receiving, each process should print:

   ```
   Rank X received value Y from Rank Z
   ```

4. Verify that each process receives the correct value (the rank of its left neighbor).
5. Measure the total communication time using `MPI_Wtime()` and print it from rank 0.

Questions:

- Why does using `MPI_Isend` prevent deadlock in this ring pattern?
- What would happen if all processes used blocking `MPI_Send` instead?

Task 2 (`halo_exchange.c`)

Write an **MPI program** `halo_exchange.c` that implements a 1D halo exchange for a distributed array.

Background: In many parallel simulations (e.g., stencil computations), each process holds a local portion of a larger array. To compute boundary values, processes must exchange "halo" or "ghost" cells with their neighbors.

1. Each process owns a local array of size `LOCAL_SIZE = 10`. Initialize the array such that all elements contain the process rank (e.g., rank 2 has all elements set to 2).
2. Add one **halo cell** on each side of the local array:
   - `left_halo`: will receive data from the left neighbor
   - `right_halo`: will receive data from the right neighbor
3. Implement a **bidirectional halo exchange** with cyclic boundary conditions:
   (a) Send the **rightmost** local element to the right neighbor's `left_halo`
   (b) Send the **leftmost** local element to the left neighbor's `right_halo`
   (c) Receive into `left_halo` from the left neighbor

(d) Receive into `right_halo` from the right neighbor

4. Use `MPI_Sendrecv` for safe, deadlock-free communication.
5. After the exchange, each process prints its halo values:

```
Rank X: left_halo=L, right_halo=R
```

6. Verify correctness: each process's `left_halo` should equal its left neighbor's rank, and `right_halo` should equal its right neighbor's rank.

Bonus: Implement the same exchange using `MPI_Irecv` + `MPI_Isend` + `MPI_Waitall` and compare the code complexity.

Task 3 (`comm_benchmark.c`)

Write an **MPI program** `comm_benchmark.c` that benchmarks different communication strategies for neighbor exchange.

Goal: Compare the performance of blocking vs. nonblocking communication patterns.

1. Define a message size `MSG_SIZE = 1000000` (1 million integers) and number of iterations `ITERATIONS = 100`.
2. Implement **three different communication patterns**:

   Pattern A: Blocking Send/Recv (*risky*)

```
MPI_Send(..., right_neighbor, ...);
MPI_Recv(..., left_neighbor, ...);
```

   Pattern B: Nonblocking Send + Blocking Recv

```
MPI_Isend(..., right_neighbor, ..., &request);
MPI_Recv(..., left_neighbor, ...);
MPI_Wait(&request, ...);
```

   Pattern C: MPI_Sendrecv

```
MPI_Sendrecv(send_buf, ..., right_neighbor, ...,
             recv_buf, ..., left_neighbor, ..., ...);
```

3. For each pattern:
   (a) Use `MPI_Barrier` before timing to synchronize all processes
   (b) Measure the time for `ITERATIONS` exchanges using `MPI_Wtime()`
   (c) Calculate and print the average time per exchange from rank 0
4. Compare the results and discuss:
   - Which pattern performed best?
   - Did Pattern A complete successfully, or did it deadlock?
   - How does performance change with different process counts?

Hint: *Pattern A may deadlock depending on message size and MPI implementation. If it hangs, note this observation and continue with Patterns B and C.*

Task 4 (`deadlock_fix.c`)

Deadlock Analysis: The following MPI code contains a deadlock. Analyze the problem and fix it.

```
#include <mpi.h>
#include <stdio.h>

#define MSG_SIZE 1000000

int main(int argc, char *argv[]) {
    int rank, size;
    int send_buf[MSG_SIZE], recv_buf[MSG_SIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size != 2) {
        if (rank == 0) printf("Run with exactly 2 processes\n");
        MPI_Finalize();
        return 1;
    }

    for (int i = 0; i < MSG_SIZE; i++) send_buf[i] = rank;

    int partner = (rank == 0) ? 1 : 0;

    MPI_Ssend(send_buf, MSG_SIZE, MPI_INT, partner, 0, MPI_COMM_WORLD);
    MPI_Recv(recv_buf, MSG_SIZE, MPI_INT, partner, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Rank %d received data from rank %d\n", rank, partner);

    MPI_Finalize();
    return 0;
}
```

1. **Explain** why this code deadlocks. Which MPI communication mode is used and what are its semantics?
2. Create `deadlock_fix.c` with **three different fixes** for the deadlock:

   Fix A: Use nonblocking send (`MPI_Isend`) with `MPI_Wait`.

   Fix B: Use `MPI_Sendrecv`.

   Fix C: Reorder operations so that one process sends first while the other receives first (based on rank).
3. For each fix, verify that the program completes successfully and produces correct output.
4. **Discussion questions:**
   - Why does `MPI_Ssend` (synchronous send) guarantee a deadlock here, while `MPI_Send` might work for small messages?

- Which fix do you consider most elegant and why?
- What is the advantage of `MPI_Sendrecv` over manually ordering send/receive operations?