MPI_Wait

```
/**
 * @author RookieHPC
 * @brief Original source code at https://rookiehpc.github.io/mpi/docs/mpi_wait/index.html
 **/

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

/**
 * @brief Illustrates how to wait for the completion of a non-blocking
 * operation.
 * @details This program is meant to be run with 2 processes: a sender and a
 * receiver.
 **/
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get the number of processes and check only 2 processes are used
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 2)
    {
        printf("This application is meant to be run with 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
```

```
    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if(my_rank == 0)
    {
        // The "master" MPI process sends the message.
        int buffer = 12345;
        printf("MPI process %d sends the value %d.\n", my_rank, buffer);
        MPI_Ssend(&buffer, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        // The "slave" MPI process receives the message.
        int received;
        MPI_Request request;
        MPI_Irecv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

        // Do some other things while the underlying MPI_Recv progresses.
        printf("MPI process %d issued the MPI_Irecv and moved on printing this message.\n",
my_rank);

        // Wait for the MPI_Recv to complete.
        printf("MPI process %d waits for the underlying MPI_Recv to complete.\n", my_rank);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        printf("The MPI_Wait completed, which means the underlying request (i.e: MPI_Recv)
completed too.\n");
    }
```

```
    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

MPI_Test

```
/**
 * @author RookieHPC
 * @brief Original source code at https://rookiehpc.github.io/mpi/docs/mpi_test/index.html
 **/

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

/**
 * @brief Illustrates how to test for the completion of a non-blocking
 * operation.
 * @details This application is designed to cover both cases:
 * - Issuing an MPI_Test when the operation tested is not complete
 * - Issuing an MPI_Test when the operation tested is complete
 *
 * The application execution flow can be visualised below:
 *
 *              +---------------+-----------+
 *              | Operation not | Operation |
```

```
 *              | complete  yet | complete  |
 * +-------------+---------------+-----------+
 * | MPI_Test #1 |       X       |           |
 * | MPI_Test #2 |               |     X     |
 * +-------------+---------------+-----------+
 *
 * This program is meant to be run with 2 processes: a sender and a
 * receiver.
 *
 * (Note to readers: the use of a barrier and a second message message is only
 * to guarantee that the application exposes the execution flow depicted above.)
 **/
int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);

   // Get the number of processes and check only 2 processes are used
   int size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if(size != 2)
   {
      printf("This application is meant to be run with 2 processes.\n");
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }

   // Get my rank
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
if(my_rank == 0)
{
   // The "master" MPI process sends the message.
   int first_message = 12345;
   int second_message = 67890;
   MPI_Request request;

   // Wait for the receiver to issue the MPI_Test meant to fail
   MPI_Barrier(MPI_COMM_WORLD);

   printf("[Process 0] Sends first message (%d).\n", first_message);
   MPI_Isend(&first_message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
   printf("[Process 0] Sends second message (%d).\n", second_message);
   MPI_Send(&second_message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   MPI_Wait(&request, MPI_STATUS_IGNORE);
}
else
{
   // The "slave" MPI process receives the message.
   int first_message;
   int second_message;
   int ready;
   MPI_Request request;

   MPI_Irecv(&first_message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);

   // The corresponding send has not been issued yet, this MPI_Test will "fail".
```

```
        MPI_Test(&request, &ready, MPI_STATUS_IGNORE);
        if(ready)
            printf("[Process 1] MPI_Test #1: message received (%d).\n", first_message);
        else
            printf("[Process 1] MPI_Test #1: message not received yet.\n");

        // Tell the sender that we issued the MPI_Test meant to fail, it can now send the message.
        MPI_Barrier(MPI_COMM_WORLD);

        MPI_Recv(&second_message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("[Process 1] Second message received (%d), which implies that the first message is
received too.\n", second_message);
        MPI_Test(&request, &ready, MPI_STATUS_IGNORE);
        if(ready)
            printf("[Process 1] MPI_Test #2: message received (%d).\n", first_message);
        else
            printf("[Process 1] MPI_Test #2: message not received yet.\n");
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

MPI_Reduce

```
**
 * @author RookieHPC
 * @brief Original source code at https://rookiehpc.github.io/mpi/docs/mpi_reduce/index.html
 **/

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

/**
 * @brief Illustrates how to use a reduce.
 * @details This application consists of a sum reduction; every MPI process
 * sends its rank for reduction before the sum of these ranks is stored in the
 * root MPI process. It can be visualised as follows, with MPI process 0 as
 * root:
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Determine root's rank
    int root_rank = 0;

    // Get the size of the communicator
    int size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 4)
    {
        printf("This application is meant to be run with 4 MPI processes.\n");
```

```
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Each MPI process sends its rank to reduction, root MPI process collects the result
    int reduction_result = 0;
    MPI_Reduce(&my_rank, &reduction_result, 1, MPI_INT, MPI_SUM, root_rank,
MPI_COMM_WORLD);

    if(my_rank == root_rank)
    {
        printf("The sum of all ranks is %d.\n", reduction_result);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

https://www.codingame.com/playgrounds/349/introduction-to-mpi/reductions

Q. A master process wants to send each element of an array of size N to N different processes. After completion of this operation, each process will add a random number to the received value and return it to the master process. When all values have been received by master process, it prints them using a printf statement.