MPI Send and Receive Bcast Operations

Introduction to MPI Communication

MPI (Message Passing Interface) is a standardized API for communicating between processes in parallel computing. The two fundamental operations are:

1. **MPI_Send** - sends a message to another process
2. **MPI_Recv** - receives a message from another process

MPI_Send - Sending Data

```
int MPI_Send(
    const void *buf,        // Address of the data to send
    int count,              // Number of elements to send
    MPI_Datatype datatype,  // Type of each element
    int dest,               // Rank of destination process
    int tag,                // Message tag (integer)
    MPI_Comm comm           // Communicator (usually MPI_COMM_WORLD)
);
```

Example of MPI_Send

```
int data = 42;
MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

This sends the integer value 42 to process with rank 1, with tag 0.

MPI_Recv - Receiving Data

```
int MPI_Recv(
    void *buf,              // Address to store received data
    int count,              // Maximum number of elements to receive
    MPI_Datatype datatype,  // Type of each element
    int source,             // Rank of sending process (or MPI_ANY_SOURCE)
    int tag,                // Message tag (or MPI_ANY_TAG)
    MPI_Comm comm,          // Communicator
    MPI_Status *status      // Status object with info about the message
);
```

Example of MPI_Recv

```
int received_data;
MPI_Status status;
MPI_Recv(&received_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

This receives an integer from process 0 with tag 0 and stores it in `received_data`.

Complete Example: Sending and Receiving Between Two Processes

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size < 2) {
        printf("This program requires at least 2 processes\n");
        MPI_Finalize();
        return 1;
    }

    if (world_rank == 0) {
        // Process 0 sends a message to process 1
        int number = 12345;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number);
    } else if (world_rank == 1) {
        // Process 1 receives the message from process 0
        int number;
        MPI_Status status;
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Process 1 received number %d from process 0\n", number);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

Key Points

1. **Blocking Nature**: Both `MPI_Send` and `MPI_Recv` are blocking calls - they don't return until the operation is complete.
2. **Matching Rules**:
   - The sender's `dest` must match the receiver's `source` (or use `MPI_ANY_SOURCE`)
   - The tags must match (or use `MPI_ANY_TAG`)
   - The datatypes must match
   - The communicator must match
3. **Status Object**: Contains information about the received message:
   - `status.MPI_SOURCE` - who sent the message
   - `status.MPI_TAG` - what tag was used
   - Can query message length with `MPI_Get_count`
4. **Common Pitfalls**:
   - Deadlocks (both processes waiting to receive before sending)
   - Mismatched tags/datatypes
   - Buffer overflow (receiving more data than buffer can hold)

Beyond the basic `MPI_Send` and `MPI_Recv`, MPI offers several other communication methods that are useful for different scenarios in parallel programming. Here's a comprehensive overview:

1. Non-blocking Communication

MPI_Isend (Immediate Send)

```c
int MPI_Isend(
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
    MPI_Request *request
);
```

- Returns immediately without waiting for the message to be sent
- The `request` object is used to check completion status

MPI_Irecv (Immediate Receive)

```c
int MPI_Irecv(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Request *request
);
```

- Posts a receive operation that will complete later
- Allows overlap of computation and communication

Completion Checking

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

- `MPI_Test`: Checks if operation is complete (non-blocking)
- `MPI_Wait`: Waits until operation is complete (blocking)

Complete Example

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        printf("Need at least 2 processes\n");
        MPI_Finalize();
        return 1;
    }

    int data_to_send = rank * 100;
    int received_data;
    MPI_Request send_request, recv_request;
    MPI_Status status;

    if (rank == 0) {
        // Process 0 sends to 1 and receives from 1
        MPI_Isend(&data_to_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &send_request);
        MPI_Irecv(&received_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &recv_request)
;

        // Do some computation while communication happens
        printf("Process 0 is computing while waiting...\n");
        for (int i = 0; i < 1000000; i++) {} // Simulate work

        // Wait for both operations to complete
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
        MPI_Wait(&recv_request, &status);

        printf("Process 0 received %d from process 1\n", received_data);
    }
    else if (rank == 1) {
        // Process 1 sends to 0 and receives from 0
        MPI_Isend(&data_to_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &send_request);
        MPI_Irecv(&received_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &recv_request)
;

        // Do some computation while communication happens
        printf("Process 1 is computing while waiting...\n");
        for (int i = 0; i < 1000000; i++) {} // Simulate work

        // Wait for both operations to complete
```

```c
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
        MPI_Wait(&recv_request, &status);

        printf("Process 1 received %d from process 0\n", received_data);
    }

    MPI_Finalize();
    return 0;
}
```

Important Considerations

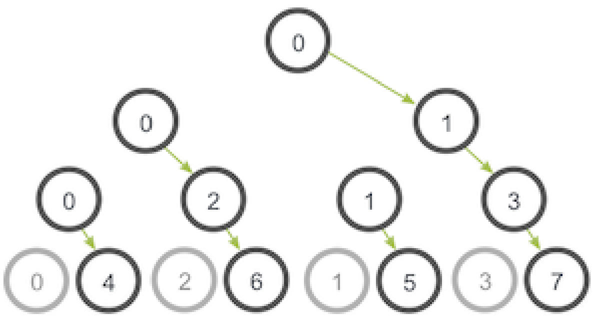
1. **Buffer Management**:
   - Don't modify send buffers until send completes
   - Don't access receive buffers until receive completes
   - Use separate buffers for overlapping communications
2. **Request Objects**:
   - Each non-blocking operation needs its own request object
   - Request objects must be matched with completion calls
   - Never reuse a request object before its operation completes
3. **Performance Benefits**:
   - Greatest when there's significant computation to overlap
   - Less beneficial for very short messages
4. **Error Handling**:
   - Always check return codes in production code
   - Use `MPI_Status` to check for message details

3. Collective Communication

Understanding and Using MPI_Bcast

What is MPI_Bcast?

MPI_Bcast (Broadcast) is a **collective communication operation** that allows one process (the root) to send the same data to all other processes in a communicator. It's one of the most commonly used MPI functions for distributing data to all processes.



Broadcast (MPI_Bcast)

```
int MPI_Bcast(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm comm
);
```

- One process (root) sends the same data to all processes

Why Use MPI_Bcast?

1. **Efficiency**:
   - More optimized than sending individually to each process
   - Uses tree-based algorithms (logarithmic complexity) rather than linear sends
2. **Synchronization**:
   - All processes reach this point together (implicit barrier)
   - Ensures all processes have the data before proceeding
3. **Simplicity**:
   - Single call replaces multiple send/receive operations

- Cleaner, more readable code

How MPI_Bcast Works

1. The **root process** provides the data in its buffer
2. All other processes provide an empty buffer of sufficient size
3. After completion, all processes have identical data in their buffers

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int ROWS = 3, COLS = 4;
    double matrix[ROWS][COLS];

    // Root process initializes the matrix
    if (rank == 0) {
        printf("Root initializing matrix:\n");
        for (int i = 0; i < ROWS; i++) {
            for (int j = 0; j < COLS; j++) {
                matrix[i][j] = i * 10 + j;
                printf("%.1f ", matrix[i][j]);
            }
            printf("\n");
        }
    }

    // Broadcast the entire matrix to all processes
    MPI_Bcast(matrix, ROWS*COLS, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
for (int p = 0; p < size; p++) {
    if (rank == p) {
        printf("Process %d received:\n", rank);
        for (int i = 0; i < ROWS; i++) {
            for (int j = 0; j < COLS; j++) {
                printf("%.1f ", matrix[i][j]);
            }
```

```
            printf("\n");
        }
        fflush(stdout);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
    MPI_Finalize();
    return 0;
}
```