13X007: Comprehensive Analysis of Message Broadcasting Methods in MPI

Due on 18.10.2023

Parallelism

UNIVERSITÉ
DE GENÈVE

CHRISTOFOROU Anthony

Contents

1 Introduction

This report provides a comprehensive analysis of three distinct methods for broadcasting messages among multiple threads using the Message Passing Interface (MPI) in C++. The methods under study are Sequential Broadcast, Sequential Ring, and Hypercube Broadcast. The objective is to dissect the algorithms, elucidate key components of the code, elaborate on the testing methodologies, and discuss challenges and performance metrics.

2 Algorithms

2.1 Sequential Broadcast

2.1.1 Algorithm Description

Sequential Broadcast is straightforward. The root node, usually identified with a rank of zero within MPI, sends the message to all other nodes sequentially. This is a one-to-all communication pattern.

2.1.2 Code Explanation

- `int data = rank;`: The message to be broadcasted is initialized. For demonstration purposes, it's set to the rank number.
- `int size;`: This variable will hold the total number of processes or nodes involved.
- `MPI_Comm_size(MPI_COMM_WORLD, &size);`: Here, we populate `size` with the total number of processes.
- The `if (rank == 0)` block: Only the root node (with rank 0) initiates the sending process.
- `MPI_Send(&data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);`: The root node sends the message to each node identified by the index i.

2.1.3 Code Snippet

Listing 1: Key part of Sequential Broadcast

```
int data = rank;  // Initialize data with the rank of the process
int received_data;

if(rank == 0) {
    // Root process sends data to all other processes
    for(int i = 1; i < size; i++) {
        std::cout << "Broadcast: Rank " << rank <<
        " sending data to rank " << i << std::endl;
        MPI_Send(&received_data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
} else {
    // All other processes receive data from root process
    MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat);
    std::cout << "Broadcast: Rank " << rank <<
```

```
    "␣received␣data␣from␣rank␣0" << std::endl;
}
```

2.2 Sequential Ring

2.2.1 Algorithm Description

In Sequential Ring, each node forwards the received message to its immediate successor in a circular topology. The message circulates until it reaches back to the root node.

2.2.2 Code Explanation

- `int next = (rank + 1) % size;`: Identifies the next node in the ring.
- `int prev = (rank - 1 + size) % size;`: Identifies the previous node in the ring.
- The `if (rank == 0)` block: The root node initiates the message sending.
- `MPI_Recv(&data, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &stat;`: Each node waits to receive a message from its predecessor.
- `if (rank != 0)`: Non-root nodes forward the message.

2.2.3 Code Snippet

Listing 2: Key part of Sequential Ring

```
// Calculate the next process in the ring
int next = (rank + 1) % size;
// Calculate the previous process in the ring
int prev = (rank + size - 1) % size;

if(rank == 0) {
    // Root process starts the ring
    std::cout << "Ring:␣Rank␣" << rank <<
    "␣sending␣data␣to␣rank␣" << next << std::endl;
    MPI_Send(&data, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
} else {
    // Receive data from the previous process
    MPI_Recv(&received_data, 1, MPI_INT, prev, 0,
    MPI_COMM_WORLD, &stat);
    std::cout << "Ring:␣Rank␣" << rank <<
    "␣received␣data␣from␣rank␣" << prev << std::endl;

    // Send data to the next process
    std::cout << "Ring:␣Rank␣" << rank <<
    "␣sending␣data␣to␣rank␣" << next << std::endl;
    MPI_Send(&data, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
}
```

2.3 Hypercube

2.3.1 Algorithm Description

Hypercube broadcasting is a more complex yet efficient approach. Nodes are arranged in a hypercube topology, and each node forwards the message to its neighbors in this topology.

2.3.2 Code Explanation

- `int dim = std::log2(size);`: Calculates the number of dimensions in the hypercube.
- `int partner = rank ^ (1 << i);`: Bitwise XOR operation to find the partner node for each dimension.
- `MPI_Send` and `MPI_Recv`: Nodes send and receive messages to and from their partners.

2.3.3 Code Snippet

Listing 3: Key part of Hypercube

```
int dim = std::log2(size);
for (int i = 0; i < dim; ++i) {
    // Compute partner rank by XOR-ing with 2^i
    int partner = rank ^ (1 << i);
    int received_data;

    if (rank < partner) {
        // Lower-ranked process sends first, then receives
        std::cout << "Hypercube: Rank " << rank <<
        " sending data to rank " << partner << std::endl;
        MPI_Send(&data, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);

        MPI_Recv(&received_data, 1, MPI_INT, partner, 0,
        MPI_COMM_WORLD, &stat);
        std::cout << "Hypercube: Rank " << rank <<
        " received data from rank " << partner << std::endl;
    } else {
        // Higher-ranked process receives first, then sends
        MPI_Recv(&received_data, 1, MPI_INT, partner, 0,
        MPI_COMM_WORLD, &stat);
        std::cout << "Hypercube: Rank " << rank <<
        " received data from rank " << partner << std::endl;

        std::cout << "Hypercube: Rank " << rank <<
        " sending data to rank " << partner << std::endl;
        MPI_Send(&data, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
    }

    // Update the data by adding the received_data
    data += received_data;
    }
```

3 Testing and Verification

3.1 Running the Code

Listing 4: Compiling the code

```
make
```

Listing 5: Running the code

```
mpirun -np 4 ./build/communication
```

3.2 Using Baobab

To be able to experience real-world performance, the code was run on the Baobab cluster. The following commands were used to compile and run the code:

Listing 6: Compiling the code

```
module load CUDA
module load foss
make
```

In order to efficiently manage the cluster resources, an SBATCH script named `run.sbatch` was used. This script provides Slurm with the specifications required for the job. Here's a breakdown of the SBATCH script's main components:

Listing 7: SBATCH script

```
#!/bin/sh
#SBATCH --job-name broadcast
#SBATCH --error broadcast-error.e%j
#SBATCH --output broadcast-out.o%j
#SBATCH --ntasks 16
#SBATCH --cpus-per-task 1
#SBATCH --partition debug-cpu
#SBATCH --time 15:00

module load CUDA
module load foss

echo $SLURM_NODELIST

srun --mpi=pmi2 ./build/communication broadcast
srun --mpi=pmi2 ./build/communication ring
srun --mpi=pmi2 ./build/communication hypercube
```

Listing 8: Running the code

```
sbatch run.sbatch
```

3.3 Testing Methodology

Testing involved using different configurations:

1. Varying the number of processors from 2 to 16.
2. Different message sizes.

Logs were generated at each node to ensure that the correct message was received

4 Challenges and Issues

1. **Deadlock in Sequential Ring**: Careful ordering of MPI_Send and MPI_Recv was required to prevent deadlock situations.
2. **Scalability in Hypercube**: The complexity of identifying neighbors in a hypercube topology increased with the number of nodes, requiring more computational overhead.
3. **Baobab Issues**: It was pretty hard to run my code in Baobab as it was the first time I was using it. The documentation was not very clear and I had to research some things on my own. The first thing was that i had to compile my code on Baobab so that the C libraries would be compatible with the cluster. I had to load the CUDA and foss modules in my /texttt.bashrc file to be able to compile my code. The second thing was that I had to add the `-mpi=pmi2` flag to the `srun` command as the mpi version detected by the cluster was not compatible with it.

5 Performance and Scalability

Performance metrics were gathered to measure the elapsed wall-clock time for each method.

- **Sequential Broadcast**: Scales linearly, $O(N)$, where N is the number of processors.
- **Sequential Ring**: Better performance but faces issues with scalability, $O(N)$.
- **Hypercube**: Scales logarithmically, $O(\log N)$, making it the most efficient for a large number of processors.

6 Explanation of Output Logs and Determinism

6.1 Non-Determinism in Output

The output logs of the various communication schemes, including Sequential Broadcast, Sequential Ring, and Hypercube, exhibit non-deterministic behavior. This non-determinism manifests in the unpredictable order of log messages from the receiving ranks. Such inconsistencies arise due to the asynchronous nature of the underlying message-passing mechanisms.

6.2 Role of `LRecv` in Determinism

Blocking receive operations, denoted as `LRecv` in parallel programming libraries, can introduce determinism. When a rank uses `LRecv`, it blocks until it receives the expected message, effectively serializing the communication steps. This ensures that the log messages will appear in a deterministic sequence.

6.3 Role of `LSend` in Determinism

Blocking send operations, represented as **`LSend`**, can also contribute to determinism. Unlike asynchronous send operations, **`LSend`** blocks the sending rank until the receiving rank has received the message. This ensures a more predictable flow of data and further reduces the chances of non-deterministic log output.

6.4 Combined Use of `LRecv` and `LSend`

Using **`LRecv`** and **`LSend`** together provides a robust way to achieve deterministic communication. For example, in the Hypercube scheme, replacing asynchronous sends and receives with **`LSend`** and **`LRecv`** will enforce a strict ordering of message passing, resulting in deterministic log outputs.

6.5 Example: Hypercube with `LSend` and `LRecv`

To illustrate, consider a step in the Hypercube algorithm where Rank 2 is supposed to receive a message from Rank 0 and forward it to Rank 3:

```
// Rank 2
LRecv(message, source=0);
LSend(message, destination=3);
```

Here, Rank 2 will block until it receives the message from Rank 0 (**`LRecv`**). Once the message is received, it will then block again until it confirms that Rank 3 has received the forwarded message (**`LSend`**). This strict ordering ensures that the log messages follow a deterministic sequence.

6.6 Output Logs

Listing 9: Baobab Output

```
    cpu[001-002]
    Broadcast: Rank 0 sending data to rank 1
    Broadcast: Rank 0 sending data to rank 2
    Broadcast: Rank 1 received data from rank 0
    Broadcast: Rank 0 sending data to rank 3
    Broadcast: Rank 2 received data from rank 0
    Broadcast: Rank 0 sending data to rank 4
    Broadcast: Rank 3 received data from rank 0
    Broadcast: Rank 0 sending data to rank 5
    Broadcast: Rank 4 received data from rank 0
    Broadcast: Rank 0 sending data to rank 6
    Broadcast: Rank 5 received data from rank 0
    Broadcast: Rank 0 sending data to rank 7
    Broadcast: Rank 6 received data from rank 0
    Broadcast: Rank 0 sending data to rank 8
    Broadcast: Rank 7 received data from rank 0
    Broadcast: Rank 0 sending data to rank 9
    Broadcast: Rank 8 received data from rank 0
    Broadcast: Rank 0 sending data to rank 10
    Broadcast: Rank 9 received data from rank 0
```

```
Broadcast: Rank 0 sending data to rank 11
Broadcast: Rank 10 received data from rank 0
Broadcast: Rank 0 sending data to rank 12
Broadcast: Rank 11 received data from rank 0
Broadcast: Rank 0 sending data to rank 13
Broadcast: Rank 12 received data from rank 0
Broadcast: Rank 0 sending data to rank 14
Broadcast: Rank 13 received data from rank 0
Broadcast: Rank 0 sending data to rank 15
Broadcast: Rank 14 received data from rank 0
Broadcast: Rank 15 received data from rank 0
Ring: Rank 0 sending data to rank 1
Ring: Rank 1 received data from rank 0
Ring: Rank 1 sending data to rank 2
Ring: Rank 2 received data from rank 1
Ring: Rank 2 sending data to rank 3
Ring: Rank 3 received data from rank 2
Ring: Rank 3 sending data to rank 4
Ring: Rank 4 received data from rank 3
Ring: Rank 4 sending data to rank 5
Ring: Rank 5 received data from rank 4
Ring: Rank 5 sending data to rank 6
Ring: Rank 6 received data from rank 5
Ring: Rank 6 sending data to rank 7
Ring: Rank 7 received data from rank 6
Ring: Rank 7 sending data to rank 8
Ring: Rank 8 received data from rank 7
Ring: Rank 8 sending data to rank 9
Ring: Rank 9 received data from rank 8
Ring: Rank 9 sending data to rank 10
Ring: Rank 10 received data from rank 9
Ring: Rank 10 sending data to rank 11
Ring: Rank 11 received data from rank 10
Ring: Rank 11 sending data to rank 12
Ring: Rank 12 received data from rank 11
Ring: Rank 12 sending data to rank 13
Ring: Rank 13 received data from rank 12
Ring: Rank 13 sending data to rank 14
Ring: Rank 14 received data from rank 13
Ring: Rank 14 sending data to rank 15
Ring: Rank 15 received data from rank 14
Ring: Rank 15 sending data to rank 0
Hypercube: Rank 14 sending data to rank 15
Hypercube: Rank 0 sending data to rank 1
Hypercube: Rank 15 received data from rank 14
Hypercube: Rank 15 sending data to rank 14
Hypercube: Rank 14 received data from rank 15
Hypercube: Rank 2 sending data to rank 3
Hypercube: Rank 4 sending data to rank 5
```

```
Hypercube: Rank 6 sending data to rank 7
Hypercube: Rank 8 sending data to rank 9
Hypercube: Rank 10 sending data to rank 11
Hypercube: Rank 12 sending data to rank 13
Hypercube: Rank 3 received data from rank 2
Hypercube: Rank 3 sending data to rank 2
Hypercube: Rank 2 received data from rank 3
Hypercube: Rank 13 received data from rank 12
Hypercube: Rank 13 sending data to rank 12
Hypercube: Rank 13 sending data to rank 15
Hypercube: Rank 12 received data from rank 13
Hypercube: Rank 12 sending data to rank 14
Hypercube: Rank 1 received data from rank 0
Hypercube: Rank 1 sending data to rank 0
Hypercube: Rank 1 sending data to rank 3
Hypercube: Rank 0 received data from rank 1
Hypercube: Rank 0 sending data to rank 2
Hypercube: Rank 5 received data from rank 4
Hypercube: Rank 5 sending data to rank 4
Hypercube: Rank 5 sending data to rank 7
Hypercube: Rank 4 received data from rank 5
Hypercube: Rank 4 sending data to rank 6
Hypercube: Rank 7 received data from rank 6
Hypercube: Rank 7 sending data to rank 6
Hypercube: Rank 6 received data from rank 7
Hypercube: Rank 15 received data from rank 13
Hypercube: Rank 15 sending data to rank 13
Hypercube: Rank 9 received data from rank 8
Hypercube: Rank 9 sending data to rank 8
Hypercube: Rank 9 sending data to rank 11
Hypercube: Rank 8 received data from rank 9
Hypercube: Rank 8 sending data to rank 10
Hypercube: Rank 13 received data from rank 15
Hypercube: Rank 14 received data from rank 12
Hypercube: Rank 14 sending data to rank 12
Hypercube: Rank 11 received data from rank 10
Hypercube: Rank 11 sending data to rank 10
Hypercube: Rank 10 received data from rank 11
Hypercube: Rank 12 received data from rank 14
Hypercube: Rank 2 received data from rank 0
Hypercube: Rank 2 sending data to rank 0
Hypercube: Rank 0 received data from rank 2
Hypercube: Rank 0 sending data to rank 4
Hypercube: Rank 2 sending data to rank 6
Hypercube: Rank 3 received data from rank 1
Hypercube: Rank 3 sending data to rank 1
Hypercube: Rank 3 sending data to rank 7
Hypercube: Rank 1 received data from rank 3
Hypercube: Rank 1 sending data to rank 5
```

```
Hypercube: Rank 7 received data from rank 5
Hypercube: Rank 7 sending data to rank 5
Hypercube: Rank 1 received data from rank 5
Hypercube: Rank 1 sending data to rank 9
Hypercube: Rank 5 received data from rank 7
Hypercube: Rank 5 received data from rank 1
Hypercube: Rank 5 sending data to rank 1
Hypercube: Rank 5 sending data to rank 13
Hypercube: Rank 6 received data from rank 4
Hypercube: Rank 6 sending data to rank 4
Hypercube: Rank 0 received data from rank 4
Hypercube: Rank 0 sending data to rank 8
Hypercube: Rank 4 received data from rank 6
Hypercube: Rank 4 received data from rank 0
Hypercube: Rank 4 sending data to rank 0
Hypercube: Rank 4 sending data to rank 12
Hypercube: Rank 10 received data from rank 8
Hypercube: Rank 10 sending data to rank 8
Hypercube: Rank 10 sending data to rank 14
Hypercube: Rank 8 received data from rank 10
Hypercube: Rank 8 sending data to rank 12
Hypercube: Rank 7 received data from rank 3
Hypercube: Rank 7 sending data to rank 3
Hypercube: Rank 7 sending data to rank 15
Hypercube: Rank 3 received data from rank 7
Hypercube: Rank 3 sending data to rank 11
Hypercube: Rank 6 received data from rank 2
Hypercube: Rank 6 sending data to rank 2
Hypercube: Rank 6 sending data to rank 14
Hypercube: Rank 2 received data from rank 6
Hypercube: Rank 2 sending data to rank 10
Hypercube: Rank 14 received data from rank 10
Hypercube: Rank 14 sending data to rank 10
Hypercube: Rank 10 received data from rank 14
Hypercube: Rank 10 received data from rank 2
Hypercube: Rank 10 sending data to rank 2
Hypercube: Rank 14 received data from rank 6
Hypercube: Rank 14 sending data to rank 6
Hypercube: Rank 2 received data from rank 10
Hypercube: Rank 9 received data from rank 11
Hypercube: Rank 9 sending data to rank 13
Hypercube: Rank 11 received data from rank 9
Hypercube: Rank 11 sending data to rank 9
Hypercube: Rank 11 sending data to rank 15
Hypercube: Rank 6 received data from rank 14
Hypercube: Rank 12 received data from rank 8
Hypercube: Rank 12 sending data to rank 8
Hypercube: Rank 8 received data from rank 12
Hypercube: Rank 8 received data from rank 0
```

```
Hypercube: Rank 8 sending data to rank 0
Hypercube: Rank 0 received data from rank 8
Hypercube: Rank 12 received data from rank 4
Hypercube: Rank 12 sending data to rank 4
Hypercube: Rank 4 received data from rank 12
Hypercube: Rank 15 received data from rank 11
Hypercube: Rank 15 sending data to rank 11
Hypercube: Rank 11 received data from rank 15
Hypercube: Rank 15 received data from rank 7
Hypercube: Rank 15 sending data to rank 7
Hypercube: Rank 11 received data from rank 3
Hypercube: Rank 11 sending data to rank 3
Hypercube: Rank 3 received data from rank 11
Hypercube: Rank 7 received data from rank 15
Hypercube: Rank 13 received data from rank 9
Hypercube: Rank 13 sending data to rank 9
Hypercube: Rank 13 received data from rank 5
Hypercube: Rank 13 sending data to rank 5
Hypercube: Rank 5 received data from rank 13
Hypercube: Rank 9 received data from rank 13
Hypercube: Rank 9 received data from rank 1
Hypercube: Rank 9 sending data to rank 1
Hypercube: Rank 1 received data from rank 9
```