13X007: Assignement #2

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 message = 42;`: The message to be broadcasted is initialized. For demonstration purposes, it's set to 42.
- `int num_procs;`: This variable will hold the total number of processes or nodes involved.
- `MPI_Comm_size(MPI_COMM_WORLD, &num_procs);`: Here, we populate `num_procs` with the total number of processes.
- The `if (my_rank == 0)` block: Only the root node (with rank 0) initiates the sending process.
- `MPI_Send(&message, 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 message = 42;  // Message to be sent
int num_procs;     // Number of processes
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if (my_rank == 0) {
    for(int i = 1; i < num_procs; ++i) {
        MPI_Send(&message, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
}
else {
    MPI_Recv(&message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

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

2.2.3 Code Snippet

Listing 2: Key part of Sequential Ring

```
int next = (my_rank + 1) % num_procs;
int prev = (my_rank - 1 + num_procs) % num_procs;

if (my_rank == 0) {
    MPI_Send(&message, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
}

MPI_Recv(&message, 1, MPI_INT, prev, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

if (my_rank != 0) {
    MPI_Send(&message, 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 = 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 = log2(size);
for(int i = 0; i < dim; ++i)
    int partner = rank ^ (1 << i);
    MPI_Send(&message, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
    MPI_Recv(&message, 1, MPI_INT, partner, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

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 1024.
2. Different message sizes ranging from 1 byte to 1 megabyte.

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.

5 Performance and Scalability

Performance metrics were gathered using the MPI_Wtime function 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 Output Comments

- **Sequential Broadcast**: All nodes received the message but with an increase in latency as the number of processors increased.
- **Sequential Ring**: Reduced latency but increased complexity in preventing deadlock.
- **Hypercube**: Efficient but with increased setup overhead for identifying neighbors.