Advanced Parallel Programming Exercise 2

Tim Beringer

TECHNISCHE UNIVERSITÄT DARMSTADT

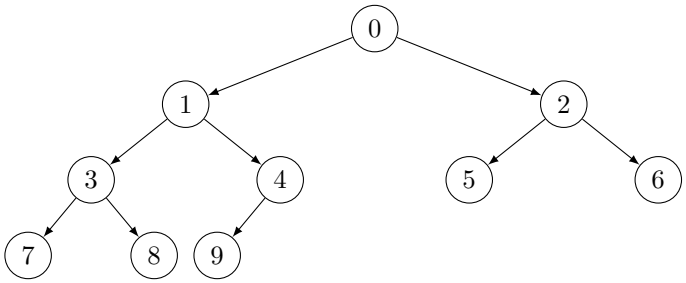
Summer term 2025
22.05.2025

Please solve the following tasks by 22.05.2025. The results are not graded, but a solution is discussed on 22.05.2025.

Task 1: Implement MPI Broadcast

A "broadcast" is a collective communication operation that sends a given message from one process to all other processes. In this task, we will re-implement the same functionality in different ways to get familiar with some important communication patterns.

- Implement a function `Bcast_simple` with the four arguments `buffer, count, datatype` and `comm` as defined for `MPI_Bcast` (no `root` argument), where rank #0 sends the given message in a simple for-loop to all other ranks.
- A "complete binary tree" is a directed acyclic graph in which each node, except possibly for the last level, which has two child nodes. Set up three formulas that calculate the number for (1) the parent of a child, (2) the left child of a parent, and (3) the right child of a parent, according to the node numbering scheme given in this example (hint: $\lfloor x \rfloor$ is the largest integer not greater than x):



- Implement the above formulas as three separate functions, and add checks for the border cases (i.e., no parent, no child) where -1 should be returned. Use the function prototypes seen below:

```
int get_parent(int child, int num_nodes);
int get_left_child(int parent, int num_nodes);
int get_right_child(int parent, int num_nodes);

```

- The directed edges can be viewed as communication between the MPI processes of the linked node numbers. Use the above functions to implement a more efficient broadcast `Bcast_tree` (same arguments as for `Bcast_simple`) where all ranks except rank #0 first receive a message from their parent using `MPI_Recv`, and then forward this message to their left as well as right children (if available!) using `MPI_Send`.

APP – Exercise 2

Solution:

- See Listing 1.

```c
#include <mpi.h>

void Bcast_simple(void *buffer, int count, MPI_Datatype datatype, MPI_Comm comm)
{
   int i, my_rank, num_procs, tag=5;
   MPI_Status status;

   MPI_Comm_size(comm, &num_procs);
   MPI_Comm_rank(comm, &my_rank);

   if (0 == my_rank) {
      for (i = 1; i < num_procs; i++) {
         MPI_Send(buffer, count, datatype, i, tag, comm);
      }
   }
   else {
      MPI_Recv(buffer, count, datatype, 0, tag, comm, &status);
   }
}
```

Listing 1: Broadcast (simple)

-

$$\text{parent}(x) = \left\lfloor \frac{x-1}{2} \right\rfloor$$

$$\text{leftc}(x) = (2 \cdot x) + 1$$

$$\text{rightc}(x) = (2 \cdot x) + 2$$

- See Listing 2.

```c
int get_parent(int child, int num_nodes)
{
   int parent;

   /* note: num_nodes is not used int this implementation */
   if (child > 0) {
      parent = (child - 1) >> 1;
   }
   else parent = -1; /* no parent */
   return parent;
}

int get_left_child(int parent, int num_nodes)
{
   int child;

   child = (parent << 1) + 1;
   if (child >= num_nodes) child = -1;
   return child;
}

int get_right_child(int parent, int num_nodes)
{
  int child;

  child = (parent << 1) + 2;
  if (child >= num_nodes) child = -1;
  return child;
}
```

Listing 2: Tree formulas

- See Listing 3.

```c
#include <mpi.h>

void Bcast_tree(void *buffer, int count, MPI_Datatype datatype, MPI_Comm comm)
{
   int my_rank, num_procs, src, dest, tag=7;
   MPI_Status status;

   MPI_Comm_size(comm, &num_procs);
   MPI_Comm_rank(comm, &my_rank);

   src = get_parent(my_rank, num_procs);
   if (src != -1) MPI_Recv(buffer, count, datatype, src, tag, comm, &status);
   dest = get_left_child(my_rank, num_procs);
   if (dest != -1) MPI_Send(buffer, count, datatype, dest, tag, comm);
   dest = get_right_child(my_rank, num_procs);
   if (dest != -1) MPI_Send(buffer, count, datatype, dest, tag, comm);
}
```

Listing 3: Broadcast (Tree)

Task 2: Benchmark and Analyze Performance of MPI Broadcast

In this task, we will use MPI performance analysis tools to profile and investigate the communication patterns of both broadcast implementations from the previous task. By collecting trace data with Score-P and visualizing it in Vampir, we will examine how each version utilizes the network and identify performance bottlenecks. This will help us better understand the efficiency of different broadcast strategies in terms of communication overhead and process idle times.

- To evaluate the performance of the two broadcast implementations, write a complete MPI program that initializes MPI, allocates a data buffer, and measures the runtime of the broadcast call. Use a buffer of integers with a reasonably large size (e.g., one million elements). On rank #0, initialize the buffer with a fixed value. Repeat the broadcast operation multiple times to reduce measurement noise. Use `MPI_Barrier` before and after the timing region to synchronize processes, and use `MPI_Wtime` to measure the elapsed time. Only rank #0 should print the final result, which should be the average time per broadcast in seconds. Add a command line argument to choose between the two implementations (`Bcast_simple` or `Bcast_tree`). Compile your program with `mpic++` using optimization flags (e.g., `-O2`).
- To run your code on the Lichtenberg cluster, use a SLURM batch script that requests 32 MPI tasks across two nodes. Run both versions (simple and tree) on 32 processes with the same input size and compare the average broadcast time. Which version is faster for $n = 32$ processes? Why?
  Hint: You can inspire from the following bash script:

```bash
#!/bin/bash
#SBATCH --job-name=bcast32
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16    # total = 32
#SBATCH --time=00:05:00
#SBATCH --output=bcast32_%j.out

module load mpi

echo "Running simple broadcast with 32 procs"
srun ./bcast_test 1000000 simple

echo "Running tree broadcast with 32 procs"
srun ./bcast_test 1000000 tree

```

  More information on cluster usage can be found on the HRZ website.[1]

[1] https://www.hrz.tu-darmstadt.de/hlr/nutzung_hlr/rechenjobs_und_batchsystem_hlr/index.en.jsp

- Analyze both broadcast implementations using Score-P [2] and Vampir [3]. On the Lichtenberg cluster, load the required modules for Score-P and Vampir before compiling and running the program. Enable both profiling and tracing in Score-P by setting the appropriate environment variables and compiling the code with Score-P instrumentation. Run both versions of the program with tracing enabled. Then, use the Vampir GUI to analyze the resulting trace files. You can open the GUI remotely using SSH with the -X option for X11 forwarding. In Vampir, navigate to the timeline view to observe the communication behavior.
  - For the simple broadcast version, examine how much time the root rank spends actively sending compared to how long the non-root ranks are idle and waiting to receive. Describe the communication pattern you observe.
  - Compare the simple to the tree-based broadcast version. How does the idle time of non-root ranks differ between the two versions? Based on your analysis, explain which implementation makes better use of parallel communication and why.

  Hint: You can enable profiling and tracing with the following variables.

```
export SCOREP_ENABLE_PROFILING=true
export SCOREP_ENABLE_TRACING=true
export SCOREP_EXPERIMENT_DIRECTORY="scorep_simple_32"

```

Solution:

- See Listing 4.

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <chrono>
#include <cstring>

int main(int argc, char* argv[]) {
    int count = 1000000;
    bool use_tree = false;

    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <count> [tree]" << std::endl;
        return EXIT_FAILURE;
    }

    count = std::stoi(argv[1]);
    if (argc > 2 && std::string(argv[2]) == "tree") {
        use_tree = true;
    }

    MPI_Init(&argc, &argv);

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

    std::vector<int> data(count);

    if (rank == 0) {
        for (int i = 0; i < count; ++i) {
            data[i] = i;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
    double start = MPI_Wtime();

    for (int i = 0; i < 1000; i++) {
```

[2] https://www.vi-hps.org/projects/score-p/
[3] https://vampir.eu/

```
        if (use_tree) {
            Bcast_tree(data.data(), count, MPI_INT, MPI_COMM_WORLD);
        } else {
            Bcast_simple(data.data(), count, MPI_INT, MPI_COMM_WORLD);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
    double end = MPI_Wtime();

    bool error = false;
    for (int i = 0; i < count; ++i) {
        if (data[i] != i) {
            error = true;
            break;
        }
    }

    if (error) {
        std::cerr << "Rank " << rank << ": Broadcast error!" << std::endl;
    }

    if (rank == 0) {
        std::cout << "Broadcast (" << (use_tree ? "Tree" : "Simple")
                  << "): " << (end - start) << " seconds" << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Listing 4: Broadcast benchmark

- The tree-based broadcast is faster for 32 processes because it reduces the number of communication steps from linear (in the simple version) to logarithmic. This results in less overall waiting time and better parallel utilization, especially as the number of processes increases.
- In the simple broadcast, the root process sends messages sequentially, leading to significant idle time on non-root ranks. In contrast, the tree-based version overlaps communication across processes, reducing idle time and distributing the load more evenly. The tree implementation makes better use of parallel communication due to its logarithmic structure.