Exercise 2

Gaia Alessio

17 May 2024

1 Performance Comparison of MPI Broadcast Algorithms

1.1 Introduction

This report details the implementation and evaluation of broadcast algorithms on a distributed memory system using the Message Passing Interface (MPI). The focus is on efficiently and reliably delivering a message from a single root process to all other nodes in the system. Various broadcast algorithms have been explored, each with its own propagation mechanism.

- Binary Tree: Processes are organized in a binary tree structure, and the message is forwarded from the sender to the leaves of the tree.
- Chain: Processes are organized in a linear chain or sequence, where each node forwards the message to its immediate successor until it reaches the end of the chain.
- Flat: The root process directly sends the message to all other processes in the system.

For this study, the broadcast message is represented by an array.

In the strong scaling experiments, only THIN nodes were utilized, and the array size ranged from 1 to 1,000,000 elements. This approach aims to investigate the scalability of the broadcast algorithm across different array sizes and node configurations. Performance metrics include the average time taken for broadcast operations using each algorithm, providing insights into their efficiency and suitability for different scenarios.

1.2 Implementations

In this section, we delve into the implementation details of various broadcast algorithms using MPI.

1.2.1 Binary Tree

The binary tree transmission algorithm efficiently distributes the message from each process to its child processes in a binary tree structure, using blocking send and receive.

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

void my_binary_tree_broadcast(int *my_data, int my_rank, int my_root_rank, int my_num_elemnts){
    MPI_Status my status;
    int my_parent_rank=(my_rank - 1) / 2;
    int my_left_child_rank= 2 * my_rank + 1;
    int my_right_child_rank= 2 * my_rank + 2;

    if (my_rank==my_root_rank){
        if (my_left_child_rank < my_procs)
            MPI_Send(my_data, my_num_elements, MPI_INT, my_left_child_rank, 0, MPI_COMM_WORLD);
        if (my_right_child_rank < my_procs)
            MPI_Send(my_data, my_num_elements, MPI_INT, my_right_child_rank, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv (my_data, my_num_elements, MPI_INT, my_parent_rank, 0, MPI_COMM_WORLD,
        &my_status);
        if (my_left_child_rank < my_procs)
            MPI_Send(my_data, my_num_elements, MPI_INT, my_left_child_rank, 0, MPI_COMM_WORLD);
        if (my_right_child_rank < my_procs)
            MPI_Send(my_data, my_num_elements, MPI_INT, my_right_child_rank, 0, MPI_COMM_WORLD);
    }
}
```

1.2.2 Chain

The chain broadcast algorithm ensures reliable message delivery by propagating the message through the linear sequence of processes. Each process except the root forwards the message received from its predecessor.

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

void my_chain_tree_broadcast(int *my_data, int my_rank, int my_root_rank, int my_num_elemnts){
    MPI_Status my status;
    int my_parent_rank=my_rank - 1;
    int my_child_rank= my_rank + 1;

    if (my_rank==my_root_rank){
        MPI_Send(my_data, my_num_elements, MPI_INT, my_child_rank, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv (my_data, my_num_elements, MPI_INT, my_parent_rank, 0, MPI_COMM_WORLD,
        &my_status);
        if (my_child_rank < my_procs)
            MPI_Send(my_data, my_num_elements, MPI_INT, my_child_rank, 0, MPI_COMM_WORLD);
    }
}
```

1.2.3 Flat

The flat broadcast algorithm efficiently distributes, using point-to-point communication, the message from the root process to all other processes in the system.

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

void my_flat_tree_broadcast(int *my_data, int my_rank, int my_root_rank, int my_num_elemnts){
    MPI_Status my status;

    if (my_rank==my_root_rank){
        for(my_i; my_i< my_procs; my_i++){
            MPI_Send(my_data, my_num_elements, MPI_INT, my_i, 0, MPI_COMM_WORLD);
        }
    }
    else {
        MPI_Recv (my_data, my_num_elements, MPI_INT, my_root_rank, 0, MPI_COMM_WORLD,
        &my_status);
    }
}
```

1.3 Strong scalability

The goal is to evaluate the strong scalability of these algorithms by measuring the execution time of each transmission algorithm for different data sizes. A warm-up phase is performed to minimize the time required to set up communication, then multiple executions are then performed to ensure reliable performance measurements.

The function was called 1000 times, then the average of the times obtained by each process was calculated, and finally the maximum of the averages calculated among all processes was taken.

The MPIWtime function is used to accurately measure the elapsed time for each transmission operation.

In figure 1 it has been plotted the data collected on 2 THIN nodes using all 48 processes. The size of the input varies from 1 to 1048576.

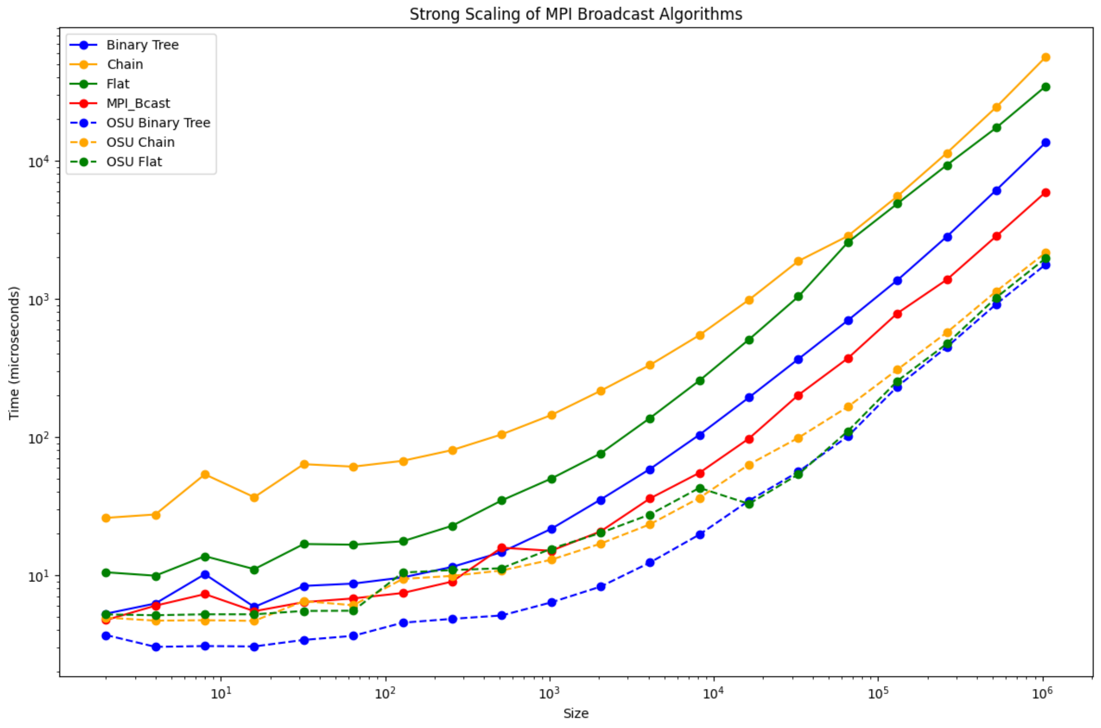

Figure 1: Strong Scaling of Broadcast Algorithms

The chain algorithm often has longer execution times than the other two algorithms because of the need to transmit the message through each node in the sequential chain. On the other hand, the binary tree algorithm can offer an advantage, particularly when dealing with a large number of processes, due to its hierarchical structure.

MPI Bcast demonstrates strong scalability, maintaining relatively constant execution times even when the number of processes increases. This indicates its ability to handle higher workloads without significant performance degradation.

While both the chain and binary tree algorithms may encounter scalability problems, especially with a larger number of processes. The sequential or hierarchical nature of communication in these algorithms can lead to congestion or network overhead, ultimately limiting their scalability.

The data obtained from the OSU benchmark have been plotted together using dashed lines in the same figure. Among the OSU implementations, the binary tree one stands out as the most efficient.

1.4 Weak scalability

The objective is to assess the weak scalability of these algorithms by measuring the execution time of each algorithm for a fixed data size (in this case equal to 1024) and a variable number of processes.

As in the case of high scalability, a heating phase is performed, different executions are performed, the average of the times obtained by each process is calculated and finally the maximum of the averages calculated between all processes is taken.

The results shown in figure 2 were obtained by performing the test on 2 THIN nodes, where the processes vary from 2 to 48.

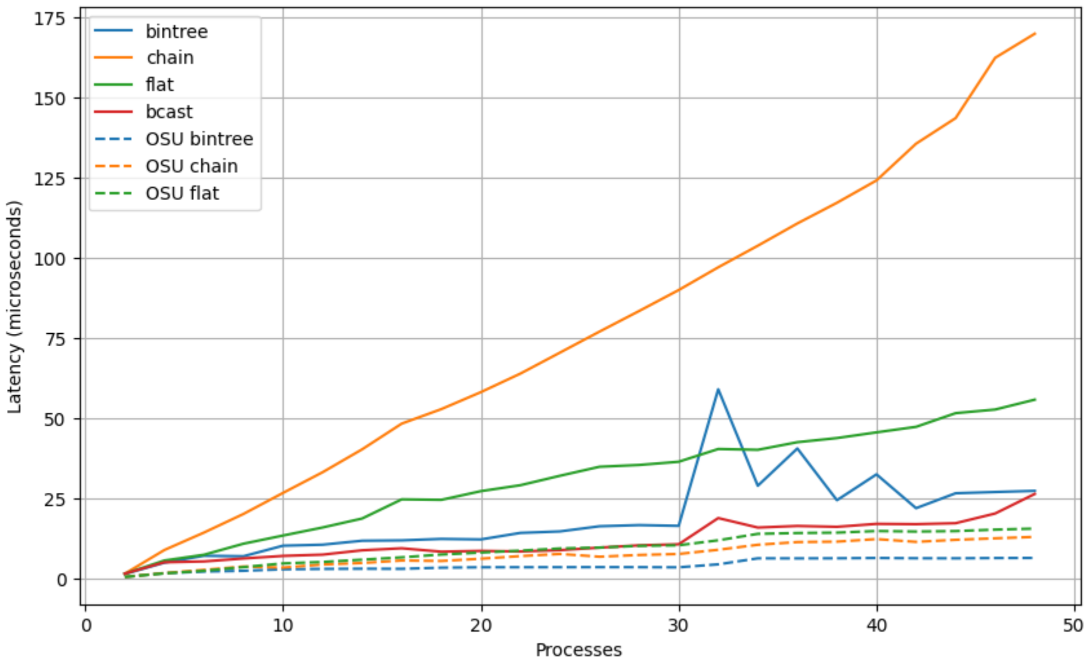

Figure 2: Weak Scaling of Broadcast Algorithms

The binary tree algorithm has an almost flat curve that shows good scalability and almost constant performance as the number of processes increases. The growth of the algorithm relative to the flat curve is probably due to the increased workload.

You can see clearly from the figure that the algorithm that scales worse is the chain algorithm.

This is because the binary tree algorithm efficiently distributes the workload through processes while the chain algorithm takes longer to send the message through all processes.

The values obtained with the MPI Bcast function are shown in red.

Also in this case, the data obtained from the OSU benchmark have been plotted together using dashed lines in the same figure.

2 The Mandelbrot set

2.1 Introduction

The Mandelbrot set is a complex geometric structure generated on the complex plane through the iterative application of a specific mathematical function. This function produces a sequence of values for each point within the plane. Mathematically, the Mandelbrot set is defined as the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$. A crucial property of this set is that if an element i of the series exceeds a distance of 2 from the origin, the series will diverge.

To determine whether a given complex number c belongs to the Mandelbrot set, we repeatedly apply the function $f_c(z)$ until the series either diverges or we reach a maximum number of iterations. If the series diverges, the point is not considered to belong to the Mandelbrot set; otherwise, it is considered part of the set.

The goal of this project is to develop an efficient and scalable code for generating the Mandelbrot set using parallel programming paradigms, specifically OpenMP.

2.2 Implementation

The main code for generating the Mandelbrot set is implemented in the file mandelbrot.c.

The mandelbrot function implements the algorithm to determine the value of a point in the Mandelbrot set. It takes as input the real and imaginary parts of a point in the complex plane, along with the maximum number of iterations allowed. The function returns the number of iterations performed before the sequence diverges, or the maximum number of iterations if the point is considered to belong to the set.

```c
unsigned char mandelbrot(double real, double imag, int max_iter) {
    double z_real = real;
    double z_imag = imag;
    for (int n = 0; n < max_iter; n++) {
        double r2 = z_real * z_real;
        double i2 = z_imag * z_imag;
        if (r2 + i2 > 4.0) return n;
        z_imag = 2.0 * z_real * z_imag + imag;
        z_real = r2 - i2 + real;
    }
    return max_iter;
}
```

The main program main accepts command-line arguments to specify the dimensions of the image, the limits of the complex plane, the maximum number of iterations, and the number of OpenMP threads to use. OpenMP library is utilized to parallelize the computation of pixel values for the Mandelbrot set image.

The main section of the code utilizes an OpenMP directive #pragma omp parallel for collapse(2) to parallelize the computation of pixel values for the image. This allows the program to distribute the workload across multiple threads, each calculating pixel values for a portion of the Mandelbrot set image. This parallel programming approach leverages system resources to accelerate the computation of pixel values for the set's image.

```
#pragma omp parallel for collapse(2)
    for (int j = 0; j < height; j++) {
        for (int i = 0; i < width; i++) {
            double x = x_left + i * (x_right - x_left) / width;
            double y = y_lower + j * (y_upper - y_lower) / height;
            unsigned char value = mandelbrot(x, y, max_iterations);
            image_buffer[j * width + i] = value;
        }
    }
```

Once all pixel values are computed, the program writes the image data to a file using the PGM format. This file represents a grayscale image of the Mandelbrot set, where lower values correspond to points within the set and higher values represent points outside the set.

2.3 Strong scalability

To test the strong scalability, an analysis was conducted on the THIN Nodes of the ORPHEUS cluster, each of which has 24 cores.

To evaluate program Performance, the test is performed by iterating over an OpenMP thread range from 1 to 24.

For each iteration, it executes the Mandelbrot program using mpirun with a Single Process and the specified number of OpenMP threads. The program execution time is measured and stored in a variable.

After each iteration, the script adds the number of threads and the run time.

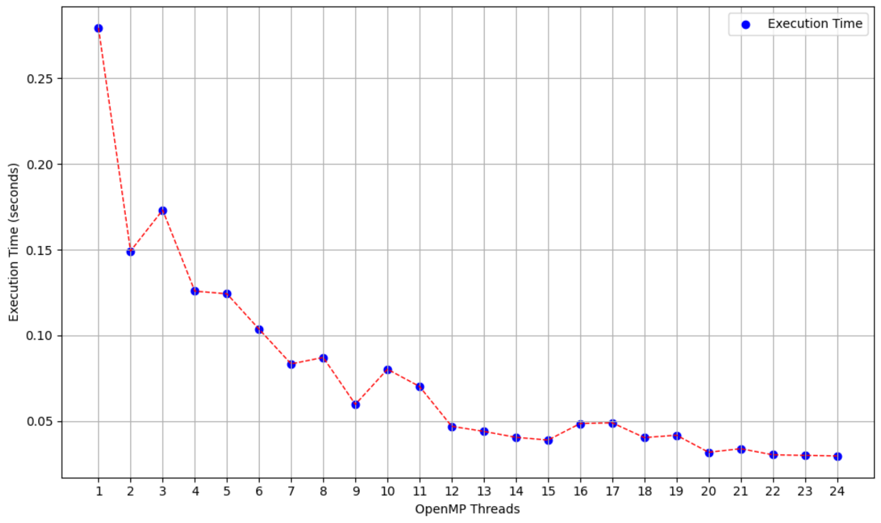

Figure 3: Execution Time vs. Number of OpenMP Threads

2.4 Weak scalability

To check for weak scalability, a script was used that automates the process of running multiple instances of the program with different problem sizes proportional to the number of OpenMP threads and registration run time.

Then, the script enters a loop to run the weak resize test. Iterate over a 1 to 24 OpenMP thread range. For each iteration, calculate the problem size (number of rows and columns) based on the base size (BASE_COLS and BASE_ROWS) multiplied by the number of OpenMP threads.

After each iteration, the script runs the Mandelbrot program using mpirun with a single process and calculated problem size. The execution time of the program is measured and stored along with the number of OpenMP threads and the size of the problem.

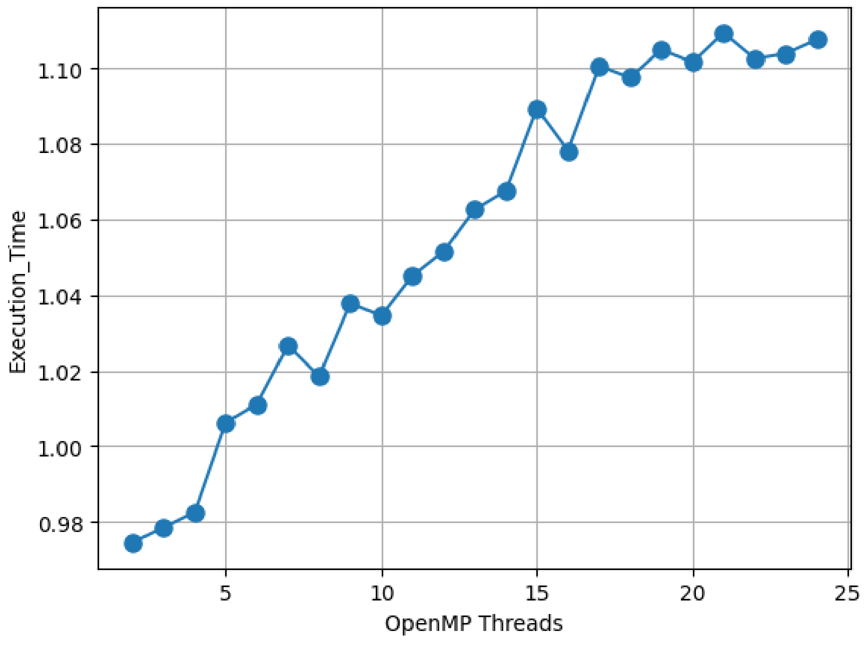

Figure 4: Execution Time vs. Number of OpenMP Threads

2.5 Image