TP5 - MPI (P2P & Collectives)

Imad Kissami

February 19, 2025

Instructions

— Use `MPI_Wtime` to measure time.

Exercise 1 : Hello World

1. Write an MPI program which prints the message `"Hello World"`.
2. Modify your program so that each process prints its rank and the total number of processes.
3. Modify your program so that only a single process (e.g. rank 0) prints a message.
4. What happens if you omit the final MPI procedure call (`MPI_Finalize`) ?

Exercise 2 : Sharing Data

Create a program that obtains an integer input from the terminal and distributes it to all the MPI processes. Each process must display its rank and the received value. Keep reading values until a negative integer is entered.

Output Example :

```
10
Process 0 got 10
Process 1 got 10
```

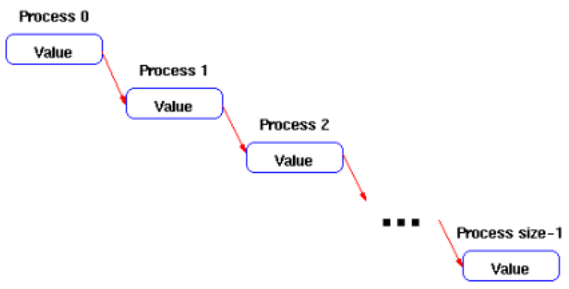
Exercise 3 : Sending in a Ring (Broadcast by Ring)

Write a program where :

— Process 0 reads a value from the user.

— It sends it to process 1, which adds its rank and sends it to process 2, and so on.

— Each process receives the data, adds its rank, prints the result, and forwards it to the next process.



Exercise 4 : Matrix Vector Product

1. Implement the MPI version of matrix-vector multiplication.
2. Plot the Speedup/Efficiency of your implementation.
3. Ensure correctness when N is not divisible by the number of processes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


void matrixVectorMult(double* A, double* b, double* x, int size) {
    for (int i = 0; i < size; ++i) {
        x[i] = 0.0;
        for (int j = 0; j < size; ++j) {
            x[i] += A[i * size + j] * b[j];
        }
    }
}
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s <matrix_size>\n", argv[0]);
        return 1;
    }
    int size = atoi(argv[1]);
    if (size <= 0) {
        printf("Matrix size must be positive.\n");
        return 1;
    }

    double* A = malloc(size * size * sizeof(double));
    double* b = malloc(size * sizeof(double));
    double* x_serial = malloc(size * sizeof(double));

    if (!A || !b || !x_serial) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    srand48(42);

    // Fill A[0][:100] with random values
    int limit = (size < 100) ? size : 100;
    for (int j = 0; j < limit; ++j)
        A[0 * size + j] = drand48();

    // Copy A[0][:100] into A[1][100:200] if possible
    if (size > 1 && size > 100) {
        int copy_len = (size - 100 < 100) ? (size - 100) : 100;
        for (int j = 0; j < copy_len; ++j)
            A[1 * size + (100 + j)] = A[0 * size + j];
    }

    // Set diagonal
    for (int i = 0; i < size; ++i)
            A[i * size + i] = drand48();

    // Fill vector b
    for (int i = 0; i < size; ++i)
        b[i] = drand48();

    // Compute x_serial = A * b manually
    matrixVectorMult(A, b, x_serial, size);

    /*
    // Compare both results
    double max_error = 0.0;
    for (int i = 0; i < size; ++i) {
        double diff = fabs(x_parallel[i] - x_serial[i]);
        if (diff > max_error)
        max_error = diff;
    }
    printf("Maximum difference between Parallel and serial result: %e\n", max_error);
    */

    free(A);
    free(b);
    free(x_serial);
    return 0;
}
```

Exercise 5 : Pi Calculation

An approximation of π is given by :

$$\pi = 4\sum_{i=0}^{N-1} \frac{1}{1+x_i^2}, \quad x_i = \frac{i+0.5}{N}$$

This expression becomes more accurate as N increases.

1. Implement the parallel version of Pi calculation using MPI.
2. Ensure correctness when N is not divisible by the number of processes.
3. Plot the Speedup/Efficiency of your implementation.