PARALLEL COMPUTING SYSTEM ASSIGNMENT

A.RAKSHANA MALYA
66CG001
917723CS009

MPI PROGRAMMING

Matrix addition using MPI Scatter and MPI Gather

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <mpi.h>

#define MATRIX_SIZE 4

// Function to generate random values for the matrix
void generateRandomInput(int matrix[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            matrix[i][j] = rand() % 10; // Generates random values between 0 and 9
        }
    }
}

// Function for matrix addition
void matrixAddition(int matrix1[MATRIX_SIZE][MATRIX_SIZE], int
matrix2[MATRIX_SIZE][MATRIX_SIZE], int result[MATRIX_SIZE][MATRIX_SIZE], int
size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}

int main(int argc, char** argv) {
    int world_size, my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int matrix1[MATRIX_SIZE][MATRIX_SIZE];
    int matrix2[MATRIX_SIZE][MATRIX_SIZE];
    int local_matrix1[MATRIX_SIZE][MATRIX_SIZE / world_size];
    int local_matrix2[MATRIX_SIZE][MATRIX_SIZE / world_size];
    int local_result[MATRIX_SIZE][MATRIX_SIZE / world_size];

    struct timeval start, end;
    long long elapsed_time;
```

```
    if (my_rank == 0) {
        generateRandomInput(matrix1); // Generate random input on the root process
        generateRandomInput(matrix2); // Generate another random matrix

        gettimeofday(&start, NULL); // Start measuring execution time
    }

    // Scatter matrix1 and matrix2 to all processes
    MPI_Scatter(matrix1, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT,
local_matrix1, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, 0,
MPI_COMM_WORLD);
    MPI_Scatter(matrix2, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT,
local_matrix2, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Perform matrix addition locally
    matrixAddition(local_matrix1, local_matrix2, local_result, MATRIX_SIZE / world_size);

    // Gather local results back to the root process using MPI_Gather
    MPI_Gather(local_result, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT,
matrix1, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, 0,
MPI_COMM_WORLD);

    if (my_rank == 0) {
        gettimeofday(&end, NULL); // Stop measuring execution time
        elapsed_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);

        printf("Matrix Addition Result:\n");
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                printf("%d ", matrix1[i][j]); // Print the result
            }
            printf("\n");
        }
        printf("Elapsed time: %lld microseconds\n", elapsed_time); // Print execution time
    }

    MPI_Finalize(); // Finalize MPI

    return 0;
}
```

OUTPUT:

```
vboxuser@Ubuntu:~$ mpicc matrix1.c -o matrix1
vboxuser@Ubuntu:~$ mpiexec -n 2 ./matrix1
Matrix A:
6 2 2 5
0 5 6 4
9 1 0 4
5 1 6 1
Matrix B:
8 5 1 6
2 1 8 5
4 5 4 9
9 9 2 5
Matrix Result:
14 7 3 11
2 6 14 9
13 6 4 13
14 10 8 6
Elapsed time: 0.000055 seconds
```

EXPLANATION:

MPI_Scatter: Distributes data from one process (often the root) to all processes in a communicator.
MPI_Gather: Purpose: Gathers data from all processes in a communicator to one process.

This C code demonstrates matrix addition using MPI (Message Passing Interface) for parallel computing. It initializes two random matrices, scatters them to different processes, performs local matrix addition, and then gathers the results back to the root process. The code includes timing measurements to calculate the execution time. MPI functions like MPI_Init, MPI_Scatter, MPI_Gather, and MPI_Finalize are used for parallelization. The program prints the resulting matrix and the elapsed time on the root process.

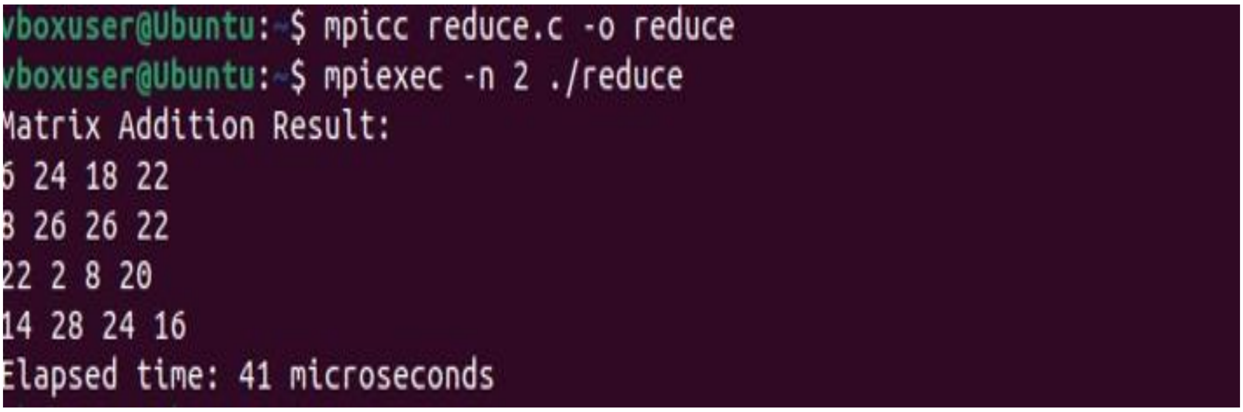
Matrix addition using MPI Reduce and Broadcast

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <mpi.h>

#define MATRIX_SIZE 4

// Function to generate random values for the matrix
void generateRandomInput(int matrix[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            matrix[i][j] = rand() % 10; // Generates random values between 0 and 9
        }
    }
```

```
}

// Function for matrix addition
void matrixAddition(int matrix1[MATRIX_SIZE][MATRIX_SIZE], int
matrix2[MATRIX_SIZE][MATRIX_SIZE], int result[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}

int main(int argc, char** argv) {
    int world_size, my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int matrix1[MATRIX_SIZE][MATRIX_SIZE];
    int matrix2[MATRIX_SIZE][MATRIX_SIZE];
    int local_result[MATRIX_SIZE][MATRIX_SIZE];
    int global_result[MATRIX_SIZE][MATRIX_SIZE];

    struct timeval start, end;
    long long elapsed_time;

    if (my_rank == 0) {
        generateRandomInput(matrix1); // Generate random input on the root process
        generateRandomInput(matrix2); // Generate another random matrix

        gettimeofday(&start, NULL); // Start measuring execution time
    }

    // Broadcast matrices to all processes
    MPI_Bcast(matrix1, MATRIX_SIZE * MATRIX_SIZE, MPI_INT, 0,
MPI_COMM_WORLD);
    MPI_Bcast(matrix2, MATRIX_SIZE * MATRIX_SIZE, MPI_INT, 0,
MPI_COMM_WORLD);
     // Perform matrix addition locally
    matrixAddition(matrix1, matrix2, local_result);
   // Sum the local results across all processes using MPI_Reduce
    MPI_Reduce(local_result, global_result, MATRIX_SIZE * MATRIX_SIZE, MPI_INT,
MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    if (my_rank == 0) {
        gettimeofday(&end, NULL); // Stop measuring execution time
        elapsed_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);

        printf("Matrix Addition Result:\n");
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                printf("%d ", global_result[i][j]); // Print the result
            }
            printf("\n");
        }
        printf("Elapsed time: %lld microseconds\n", elapsed_time); // Print execution time
    }

    MPI_Finalize(); // Finalize MPI

    return 0;
}
```

OUTPUT:



EXPLANATION:

MPI_Reduce: Combines values from all processes in a communicator, typically performing an operation
MPI_Bcast: Broadcasts data from the root process to all other processes in a communicator.

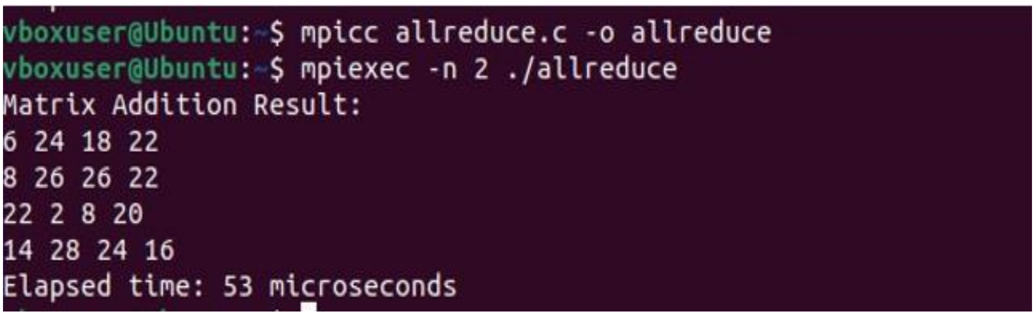
This C code demonstrates parallel matrix addition using MPI. It initializes two random matrices on the root process, broadcasts them to all processes, performs local matrix addition, and then reduces the local results to a global result using MPI_Reduce. The program measures the execution time and prints the resulting matrix and elapsed time on the root process. MPI_Bcast is employed for broadcasting, and MPI_Reduce is used for global summation. This parallel approach enhances performance by distributing the computation across multiple processes in a parallel environment, reducing the overall execution time for matrix addition.

Matrix addition using MPI AllReduce and Broadcast

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <mpi.h>

#define MATRIX_SIZE 4

// Function to generate random values for the matrix
void generateRandomInput(int matrix[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            matrix[i][j] = rand() % 10; // Generates random values between 0 and 9
        }
    }
}

// Function for matrix addition
void matrixAddition(int matrix1[MATRIX_SIZE][MATRIX_SIZE], int
matrix2[MATRIX_SIZE][MATRIX_SIZE], int result[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}

int main(int argc, char** argv) {
    int world_size, my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int matrix1[MATRIX_SIZE][MATRIX_SIZE];
    int matrix2[MATRIX_SIZE][MATRIX_SIZE];
    int local_result[MATRIX_SIZE][MATRIX_SIZE];
    int global_result[MATRIX_SIZE][MATRIX_SIZE];

    struct timeval start, end;
    long long elapsed_time;

    if (my_rank == 0) {
        generateRandomInput(matrix1); // Generate random input on the root process
        generateRandomInput(matrix2); // Generate another random matrix
```

```
        gettimeofday(&start, NULL); // Start measuring execution time
    }

    // Broadcast matrices to all processes
    MPI_Bcast(matrix1, MATRIX_SIZE * MATRIX_SIZE, MPI_INT, 0,
MPI_COMM_WORLD);
    MPI_Bcast(matrix2, MATRIX_SIZE * MATRIX_SIZE, MPI_INT, 0,
MPI_COMM_WORLD);

    // Perform matrix addition locally
    matrixAddition(matrix1, matrix2, local_result);

    // Sum the local results across all processes using MPI_Allreduce
    MPI_Allreduce(local_result, global_result, MATRIX_SIZE * MATRIX_SIZE, MPI_INT,
MPI_SUM, MPI_COMM_WORLD);

    if (my_rank == 0) {
        gettimeofday(&end, NULL); // Stop measuring execution time
        elapsed_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);

        printf("Matrix Addition Result:\n");
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                printf("%d ", global_result[i][j]); // Print the result
            }
            printf("\n");
        }
        printf("Elapsed time: %lld microseconds\n", elapsed_time); // Print execution time
    }

    MPI_Finalize(); // Finalize MPI

    return 0;
}
```

OUTPUT:



```
vboxuser@Ubuntu:~$ mpicc allreduce.c -o allreduce
vboxuser@Ubuntu:~$ mpiexec -n 2 ./allreduce
Matrix Addition Result:
6 24 18 22
8 26 26 22
22 2 8 20
14 28 24 16
Elapsed time: 53 microseconds
```

EXPLANATION:

MPI_Allreduce: An MPI collective operation that combines values from all processes in a communicator, applying a specified operation (e.g., sum, product), and distributes the result to all processes.

This C code employs MPI to parallelize matrix addition. It initializes random matrices on the root process, broadcasts them to all processes using MPI_Bcast, performs local matrix addition, and then uses MPI_Allreduce to efficiently sum the local results across all processes. The program measures and prints the resulting matrix and the elapsed time on the root process. MPI_Allreduce eliminates the need for a separate reduction step, enhancing parallel efficiency.