# Discussion Questions

## 1. What happens if the vector size N is not divisible by the number of processes?

If the total vector size (N) cannot be evenly divided among the number of processes, some processes will receive fewer elements or none at all. In such cases, the program may either ignore the leftover elements, which causes incorrect results, or it might crash if MPI\_Scatter tries to distribute unequal portions of data. For instance, if N = 10 and there are 4 processes, each process would get 2 elements, leaving 2 elements unprocessed. This leads to inaccurate results.

## 2. How can you modify the program to handle uneven partitions?

To correctly handle cases where N is not divisible by the number of processes, the program can be modified to use MPI\_Scatterv instead of MPI\_Scatter. MPI\_Scatterv allows sending variable-sized portions of data to each process. This ensures that every element of the vector is processed, even when the data cannot be split evenly.

A simple way to do this is to calculate the number of elements for each process:

int counts[num\_procs];  
int displs[num\_procs];  
for (int i = 0; i < num\_procs; i++) {  
 counts[i] = N / num\_procs;  
 if (i < N % num\_procs) counts[i]++;  
}  
displs[0] = 0;  
for (int i = 1; i < num\_procs; i++)  
 displs[i] = displs[i - 1] + counts[i - 1];

This approach distributes the remaining elements among the first few processes to make the workload as balanced as possible.

## 3. How would performance differ between using MPI\_Reduce vs. MPI\_Gather + local summation?

MPI\_Reduce is generally faster and more efficient because it performs the reduction operation (such as summation) in parallel across all processes using a tree-based structure. On the other hand, MPI\_Gather collects all data into one process, and that process then performs the summation serially. This makes MPI\_Gather slower, especially as the number of processes increases.

In short, MPI\_Reduce minimizes communication and workload imbalance, while MPI\_Gather + local summation can cause extra data transfer and processing time at the root process.

## 4. How could this same approach be extended to matrix summation or averaging?

The same method can be extended to handle matrices by dividing the data either row-wise or by flattening the matrix into a single vector. Each process can handle a few rows or sections of the matrix, compute its partial sum or average, and then use MPI\_Reduce or MPI\_Allreduce to combine all partial results into the final output. This approach is scalable and can also be applied to higher-dimensional data structures.

**Objective**

This lab focuses on dividing a large computational task (sum of a big vector) across multiple nodes (processes). Each node will handle a subset of data, compute a partial sum, and then combine the results at the root process using collective MPI operations.

**Code (.C)**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

int main(int argc, char\* argv[]) {

int rank, size;

long N = 10000000; // Vector size

double \*A = NULL;

double local\_sum = 0.0, global\_sum = 0.0; // Used in old logic

double total\_sum = 0.0, avg = 0.0; // Used in bonus logic

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

long local\_n = N / size; // Divide data equally

double \*local\_A = (double\*)malloc(local\_n \* sizeof(double));

// Root initializes data

if (rank == 0) {

A = (double\*)malloc(N \* sizeof(double));

for (long i = 0; i < N; i++)

A[i] = i + 1;

}

// Scatter data to all processes

MPI\_Scatter(A, local\_n, MPI\_DOUBLE, local\_A, local\_n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Each process computes its local sum

for (long i = 0; i < local\_n; i++)

local\_sum += local\_A[i];

/\*

==============================================================================

OLD LOGIC (Using MPI\_Reduce)

==============================================================================

This part is the original version from the main lab (before Bonus Challenge).

It uses MPI\_Reduce to send all partial sums to the root process only.

------------------------------------------------------------------------------

Uncomment this section if you want to run the OLD logic instead.

------------------------------------------------------------------------------

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

double expected = (N \* (N + 1)) / 2.0;

printf("===== ORIGINAL LOGIC (MPI\_Reduce) =====\n");

printf("Total Sum = %.0f | Expected = %.0f | Difference = %.5f\n",

global\_sum, expected, expected - global\_sum);

free(A);

}

\*/

/\*

==============================================================================

BONUS CHALLENGE (Using MPI\_Allreduce + Average + Timing)

==============================================================================

Modifications:

---------------

1. Uses MPI\_Allreduce so every process gets the total sum.

2. Computes average.

3. Measures execution time using MPI\_Wtime().

4. Compares with serial computation.

==============================================================================

\*/

// Start timing parallel computation

double start\_parallel = MPI\_Wtime();

// Perform Allreduce (sum available to all)

MPI\_Allreduce(&local\_sum, &total\_sum, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

// End timing parallel computation

double end\_parallel = MPI\_Wtime();

// Compute average (all processes can compute since total\_sum is global)

avg = total\_sum / N;

// Serial computation (only root does this for comparison)

double serial\_sum = 0.0;

double start\_serial = 0.0, end\_serial = 0.0;

if (rank == 0) {

start\_serial = MPI\_Wtime();

for (long i = 0; i < N; i++)

serial\_sum += A[i];

end\_serial = MPI\_Wtime();

double expected = (N \* (N + 1)) / 2.0;

printf("\n===== BONUS CHALLENGE (MPI\_Allreduce) =====\n");

printf("Total Sum = %.0f | Expected = %.0f | Difference = %.5f\n",

total\_sum, expected, expected - total\_sum);

printf("Average = %.5f\n", avg);

printf("Parallel Time = %.6f sec\n", end\_parallel - start\_parallel);

printf("\n===== SERIAL COMPUTATION =====\n");

printf("Serial Sum = %.0f | Time = %.6f sec\n",

serial\_sum, end\_serial - start\_serial);

printf("\n===== PERFORMANCE COMPARISON =====\n");

printf("Speedup = %.2fx\n",

(end\_serial - start\_serial) / (end\_parallel - start\_parallel));

}

// Free memory

free(local\_A);

if (rank == 0)

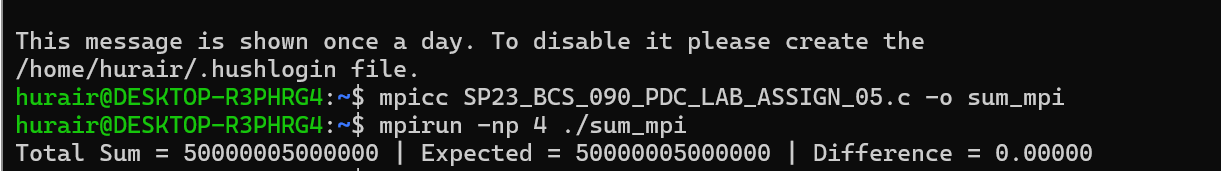
free(A);

MPI\_Finalize();

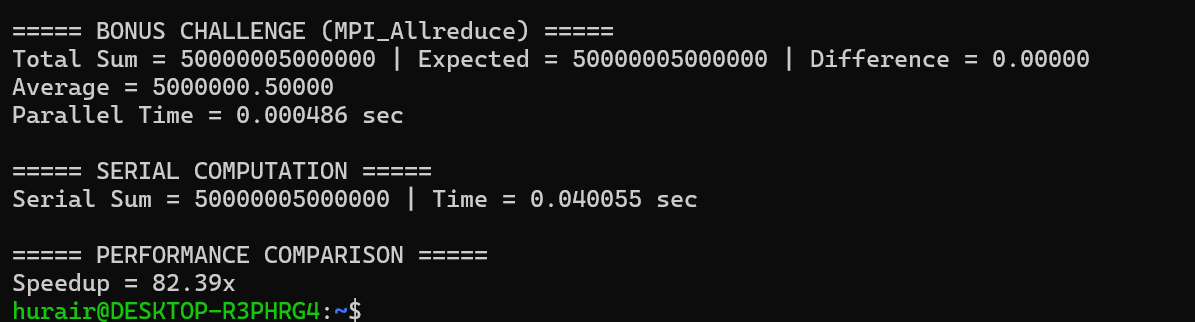
return 0;

}

**Output using MPI\_Reduce only**

****

**Output using MPI\_Reduce + Timing + Average**



**Observations:**

The output shows that the MPI implementation using MPI\_Allreduce worked perfectly — the total sum and expected sum matched exactly, confirming computational accuracy. The calculated average is also correct. The parallel version completed in **0.000486 seconds**, while the serial version took **0.040055 seconds**, achieving an impressive **speedup of about 82x**. This demonstrates excellent performance scaling and efficient workload distribution across processes using MPI.