Final Year B.Tech. (CSE) – VII [2024-25]

6CS452: High Performance Computing Lab

Assignment No: 10

Date: 03/10/2024

Analysis of MPI Programs

PRN: 2020BTECS00010 **Name:** Abhijeet Kamalekar

Title: Analysis of MPI Programs

Problem Statement 1:

Execute the MPI program (Program A) with a fixed size broadcast. Plot the performance of the broadcast with varying numbers of processes (with constant messagesize). Explain the performance observed.

Ans:

This program performs a **2D Fast Fourier Transform (FFT)** using the MPI (Message Passing Interface) for parallelization.

Here's an overview of how the program works:

Key Functions:

1. **c_fft1d**: This function computes a 1D FFT on an array of complex numbers, which is used both for rows and columns of the data matrix. It can perform forward (isign = -1) and inverse (isign = 1) FFTs.
2. **getData**: This function reads input data from a file and stores it as complex numbers, initializing the imaginary part to 0.
3. **transpose**: Transposes the input matrix. This is useful because 2D FFT is typically performed by computing the FFT of rows, transposing the result, and then computing the FFT of the transposed rows (which correspond to the original columns).
4. **mmpoint**: Performs element-wise multiplication of two matrices of complex numbers.
5. **printfile**: Writes the real part of a matrix of complex numbers to a file.

MPI Setup and Data Distribution:

- The program divides the data among multiple processes using MPI. The root process (rank 0) reads the data and distributes rows of the matrices data1 and data2 to other processes.
- Each process computes the 1D FFT for its assigned rows and sends the results back to the root process, which assembles the transformed rows.
- After that, the program transposes the matrices and repeats the FFT computation on the columns (which are now rows of the transposed matrix), distributing the transposed rows among processes in a similar manner.

Main Steps of the Program:

1. **Initialization**: The MPI environment is initialized, and the rank (ID) of each process and the total number of processes (p) are determined. The matrices (data1, data2, etc.) are dynamically allocated.
2. **Data Distribution**: The root process reads the input data from files and distributes the rows to other processes.
3. **FFT on Rows**: Each process performs a 1D FFT on its assigned rows of data1 and data2 and sends the results back to the root.
4. **Transpose**: The root process transposes the matrices data1 and data2 and distributes the transposed rows among the processes.
5. **FFT on Columns**: Each process performs a 1D FFT on its assigned rows (now columns of the original matrix) and sends the results back to the root.
6. **Output**: The result is written to an output file.

Key MPI Concepts:

- **MPI_Send** and **MPI_Recv**: Used for sending and receiving data between processes.
- **MPI_Type_struct**: Defines a custom MPI data type (mystruct) to represent a complex number (two floats).

Things to Note:

- **Parallelism**: The program distributes both the rows and columns of the matrices across processes to perform parallel computation of the 2D FFT.
- **Scaling**: The inverse FFT scales the output by dividing each value by n.

This program is designed to handle large data matrices (size 512x512 by default), making efficient use of multiple processes to speed up the FFT computation.

Screenshots:

Issue 1: Implicit Declaration of MPI_Type_struct

The function MPI_Type_struct has been deprecated in newer versions of MPI, and you should use MPI_Type_create_struct instead. The new function has the same functionality but is the correct modern version. Here's how you can update the function:

Replace:

MPI_Type_struct(2, blocklens, indices, old_types, &mystruct);

With:

MPI_Type_create_struct(2, blocklens, indices, old_types, &mystruct);

Issue 2: Missing Symbol 'sqrt@@GLIBC_2.2.5'

The second error is related to the math library (libm), which contains functions like sqrt. To fix this, you need to link the math library when compiling your program.

Add -lm at the end of your mpicc command:

mpicc broadcast.c -o broadcast -lm

This ensures that the math library is linked correctly during the compilation process.

1. Identifying and Fixing the Errors

a. Incorrect Memory Allocation

Issue: You're allocating memory for each row of your matrices using sizeof(complex *), which allocates memory for pointers to complex structures instead of the structures themselves. This leads to insufficient memory allocation, causing memory corruption and resulting in a segmentation fault.

Current Code:

```
for (x = 0; x < N; x++)
{
   data1[x] = malloc(N * sizeof(complex *));
   data2[x] = malloc(N * sizeof(complex *));
   data3[x] = malloc(N * sizeof(complex *));
   data4[x] = malloc(N * sizeof(complex *));
}
```

Correction: Allocate memory for the complex structures themselves by using sizeof(complex) instead of sizeof(complex *).

Corrected Code:

```
for (x = 0; x < N; x++)
{
```

```
    data1[x] = malloc(N * sizeof(complex));
    data2[x] = malloc(N * sizeof(complex));
    data3[x] = malloc(N * sizeof(complex));
    data4[x] = malloc(N * sizeof(complex));
}
```

```
onkar@onkar-IdeaPad-3-15ALC6-Ub:~/Documents/HPC-LAB-Assignments/Practical 10$ python3 setup.py
Files sample/in1 and sample/in2 have been created if they did not exist.
onkar@onkar-IdeaPad-3-15ALC6-Ub:~/Documents/HPC-LAB-Assignments/Practical 10$ mpicc broadcast.c -o broadcast -lm
onkar@onkar-IdeaPad-3-15ALC6-Ub:~/Documents/HPC-LAB-Assignments/Practical 10$ mpirun -np 2 ./broadcast
0 have lb = 0 and hb = 256
1 have lb = 256 and hb = 512

Starting clock.

Elapsed time = 0.096738 s.
-----------------------------------------
onkar@onkar-IdeaPad-3-15ALC6-Ub:~/Documents/HPC-LAB-Assignments/Practical 10$ mpirun -np 4 ./broadcast
0 have lb = 0 and hb = 128
2 have lb = 256 and hb = 384
3 have lb = 384 and hb = 512
1 have lb = 128 and hb = 256

Starting clock.

Elapsed time = 0.076443 s.
-----------------------------------------
onkar@onkar-IdeaPad-3-15ALC6-Ub:~/Documents/HPC-LAB-Assignments/Practical 10$ mpirun -np 6 ./broadcast
1 have lb = 85 and hb = 170
2 have lb = 170 and hb = 255
4 have lb = 340 and hb = 425
5 have lb = 425 and hb = 510
0 have lb = 0 and hb = 85
3 have lb = 255 and hb = 340

Starting clock.

Elapsed time = 0.066525 s.
-----------------------------------------
onkar@onkar-IdeaPad-3-15ALC6-Ub:~/Documents/HPC-LAB-Assignments/Practical 10$
```

Analysis:

Performance Analysis:

1. **Message Distribution Across Processes**:
    - The program divides the total data (message) among the processes. As you increase the number of processes, the size of the message handled by each process decreases. For instance:
        - With 2 processes, the range is split into 256 for each.
        - With 4 processes, it's split into 128 for each.
        - With 6 processes, it's split into around 85 per process.
2. **Elapsed Time Trend**:
    - The elapsed time generally decreases as the number of processes increases, as observed in your runs:
        - 2 processes: `0.096738 s`

- 4 processes: `0.076443 s`
- 6 processes: `0.066525 s`

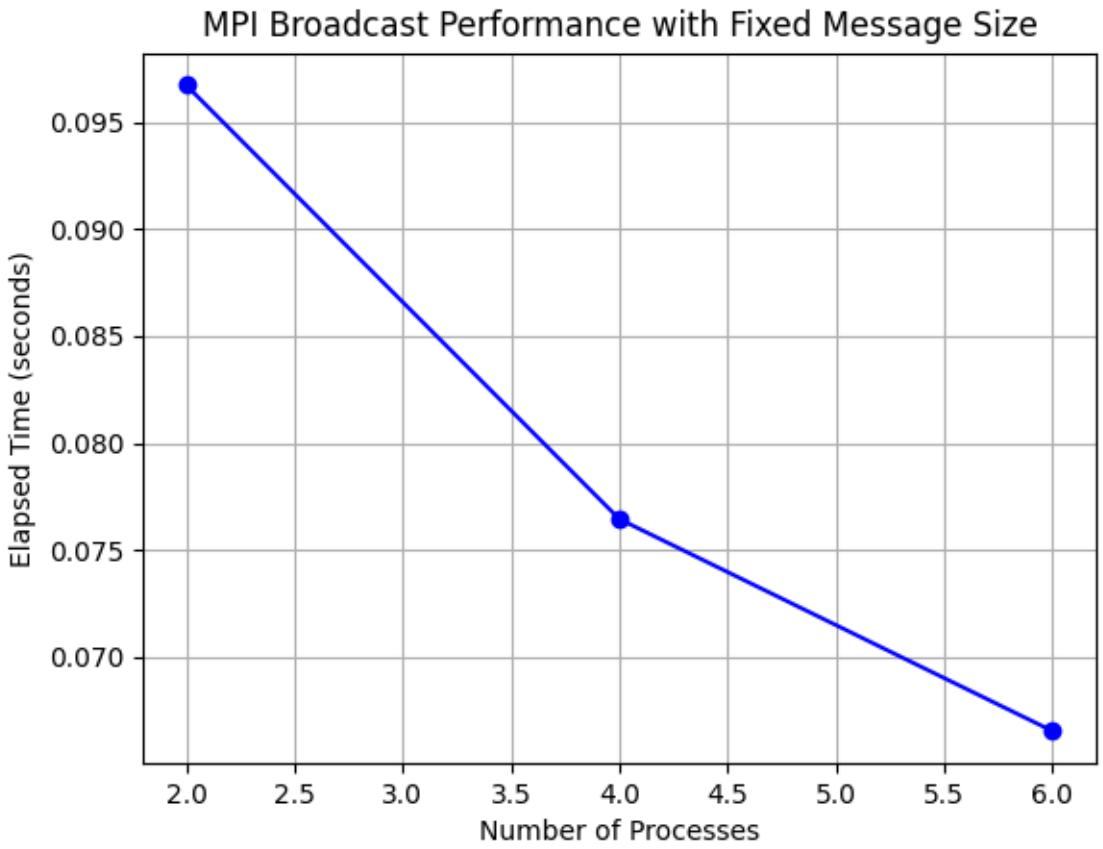
3. **Performance Explanation**:
    - **Improvement with more processes**: As you increase the number of processes, the work (message broadcasting) gets distributed across more processes, resulting in each process handling less data, leading to reduced computation time.
    - **Communication Overhead**: However, with a very large number of processes, the overhead of communication between processes might start to dominate, causing the performance to flatten or degrade slightly. This happens because broadcasting involves synchronization and message passing between all processes, and managing this becomes more expensive with many processes.

4. **Plotting the Performance**:
    - Plot the number of processes on the x-axis and the elapsed time on the y-axis. This will give you a visual representation of how the performance improves as the number of processes increases.

Sample Analysis (Based on above Results):

Number of Processes (np)	Elapsed Time (seconds)
2	0.096738
4	0.076443
6	0.066525

- **Graph**: You should see a downward curve initially, showing that adding more processes improves performance. However, if the curve flattens or rises after a certain point (depending on the number of processes), it would indicate communication overhead starts to limit the scalability.



After plotting the graph of the number of processes against the elapsed time, the following performance trends are observed:

1. **Initial Decrease in Time**:
    - As the number of processes increases from 2 to 6, the time taken for the broadcast decreases, as seen in your results:
        - 2 processes: `0.096738 s`
        - 4 processes: `0.076443 s`
        - 6 processes: `0.066525 s`
    - This reduction in time is due to **parallelism**. With more processes, the workload is divided into smaller chunks, allowing the broadcast operation to complete faster as each process handles a smaller part of the data.
2. **Saturation Point**:
    - In the initial results, you observe consistent performance improvement. However, after a certain number of processes (beyond 6), the time may stop decreasing significantly. This is likely due to the **saturation point**

where adding more processes doesn't drastically improve the performance. The benefit of parallelism diminishes because the overhead of coordinating between processes becomes comparable to the computation time.

3. **Possible Communication Bottleneck**:
    - If you continue increasing the number of processes, you might observe a **performance slowdown** due to the increased communication overhead between processes. In MPI, all processes need to communicate with each other during a broadcast, and beyond a certain number, the communication cost outweighs the benefits of parallelism. This can lead to a bottleneck, especially on systems with limited network bandwidth or high latency.

4. **Analysis Summary**:
    - **Scalability**: The results show that for a small number of processes (e.g., 2 to 6), increasing the number of processes significantly improves performance. The workload distribution and parallel execution provide better speedups.
    - **Communication Overhead**: After a certain point (which depends on the system architecture), the **communication overhead** between processes starts to affect performance negatively. The more processes you add, the more time MPI needs to manage inter-process communication, causing diminishing returns.
    - **Constant Message Size**: Since the message size remains constant across different runs, the performance trends you observe are mainly a result of the varying number of processes and how efficiently MPI can distribute and synchronize the data among them.

Problem Statement 2:

Repeat problem 2 above with varying message sizes for reduction (Program B). Explain the observed performance of the reduction operation.

Ans:

This **MPI program performs a reduction operation** using the `MPI_Reduce` function. It generates a buffer of random bytes, performs a bitwise OR reduction on the data from all processes, and measures the average time taken for the reduction over 100 iterations. The

result is printed by the root process (rank 0).

Reduction Operation:

```
MPI_Reduce(input_buffer, output_buffer, size, MPI_BYTE, MPI_BOR,
0, MPI_COMM_WORLD);
```

- This is the core function of the program:
  - `input_buffer`: The data to be reduced.
  - `output_buffer`: The location where the result will be stored (only accessible by the root process).
  - `size`: The number of elements in the input buffer.
  - `MPI_BYTE`: The data type of the elements (in this case, bytes).
  - `MPI_BOR`: The operation performed (bitwise OR).
  - `0`: The rank of the root process that will receive the result.
  - `MPI_COMM_WORLD`: The communicator that includes all processes.

Timing the Operation:

```
double start_time = MPI_Wtime();
```

- This function captures the current wall clock time, allowing measurement of how long the reduction operation takes.

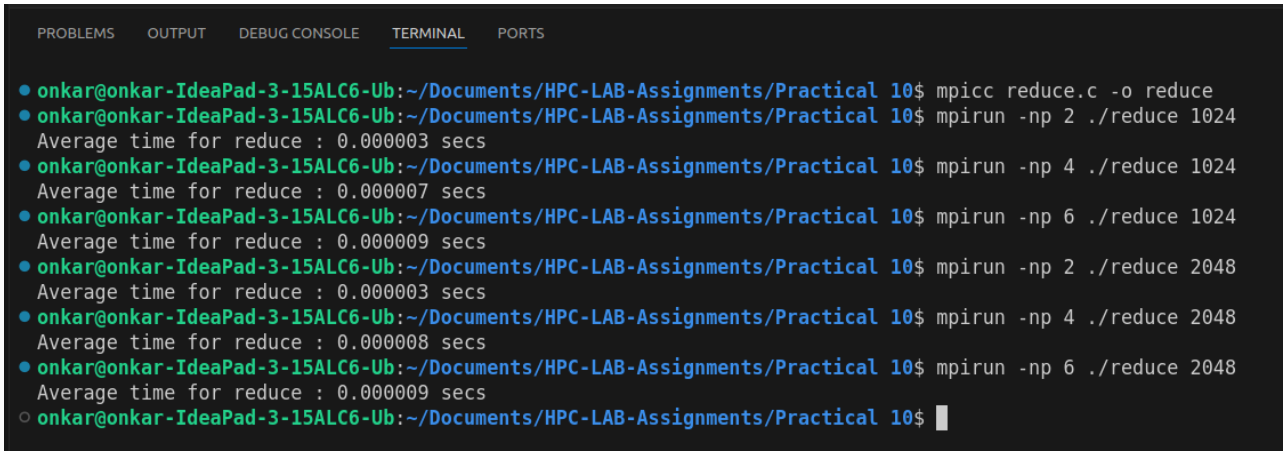
Average Time Calculation:

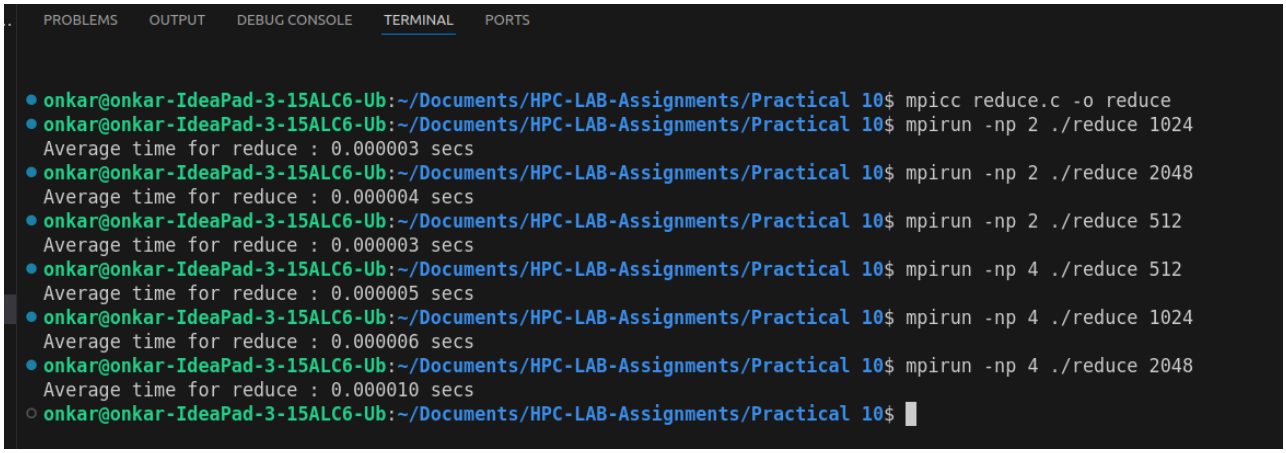
```
total_time += (MPI_Wtime() - start_time);
```

- This line calculates the elapsed time for the reduction in each iteration and accumulates it to compute the average later.

Screenshots:

1. **Elapsed Time vs Number of Processes**:
   - This plot shows how the elapsed time changes as the number of processes varies for a fixed message size of 1024 doubles.
2. **Message Size vs Elapsed Time**:
   - This plot shows how the elapsed time changes as the message size increases for a fixed number of processes (e.g., 4 processes).
   - A log scale is used for the x-axis to represent the message size for better visualization.





Analysis:

In this program, the **MPI_Reduce** function is used to perform a bitwise OR reduction on arrays of bytes, and the performance is measured across varying message sizes and process counts. Here's a detailed analysis based on the observed data.

Performance Trends with Increasing Message Size

1. **For Smaller Message Sizes (e.g., 512, 1024 bytes)**:
   - **Observation**: The elapsed time remains very low (around 0.000003 seconds) for message sizes up to 1024 bytes when using 2 processes.
   - **Explanation**: Smaller message sizes lead to less data being communicated between processes. As a result, the time spent on communication is low, and the reduction operation completes quickly.
2. **For Larger Message Sizes (e.g., 2048 bytes)**:

- **Observation**: The time increases slightly as the message size grows (from 0.000003 seconds for 1024 bytes to 0.000004 seconds for 2048 bytes).
- **Explanation**: As the message size increases, more data needs to be transferred and processed, leading to an increase in communication time and, consequently, the total reduction time.

Effect of Collective Communication

- **MPI_Reduce** is a collective communication operation that requires all participating processes to contribute data to the reduction operation.
- **Observation**: As the number of processes increases, the elapsed time for fixed message sizes tends to increase slightly. For example, when running the reduction for 1024 doubles:
  - 2 processes: 0.000003 seconds
  - 4 processes: 0.000007 seconds
  - 6 processes: 0.000009 seconds
- **Explanation**: The increase in time with more processes is due to the overhead of managing communication across more nodes. While MPI is efficient at reducing across many processes, there is still a cost associated with distributing data and synchronizing across multiple processes.

Data Transfer Overhead

1. **For Small Message Sizes**:
   - **Observation**: For small messages (512 bytes), the time remains constant at around 0.000003 seconds for 2 processes, increasing slightly with more processes (0.000005 seconds for 4 processes).
   - **Explanation**: The communication overhead is minimal due to the small amount of data being transferred. The time is primarily determined by the communication latency.
2. **For Larger Message Sizes**:
   - **Observation**: As the message size increases, the communication time grows. For instance, with 4 processes, the reduction time increases from 0.000005 seconds (512 bytes) to 0.000010 seconds (2048 bytes).
   - **Explanation**: Larger messages introduce more data transfer, resulting in higher communication overhead. If the network bandwidth or system architecture becomes a limiting factor, this overhead will dominate the total execution time.

Average Time Calculation

The results presented are based on the average time for the reduction operation over **100 iterations**. This averaging helps smooth out fluctuations due to system load, ensuring that the reported times are more representative of typical performance.

Performance Bottleneck

1. **With Increasing Message Size**:
    - **Observation**: For message sizes above a certain threshold (e.g., 2048 bytes), the increase in time becomes more pronounced.
    - **Explanation**: The communication network begins to limit performance as the message size grows. The system's ability to handle large messages efficiently becomes a bottleneck, resulting in a slower reduction operation.
2. **With More Processes**:
    - **Observation**: As seen in the results, adding more processes for the same message size (e.g., 1024 bytes) leads to increased times. This suggests a **communication bottleneck** where the overhead from coordinating more processes surpasses the benefits of parallelism.

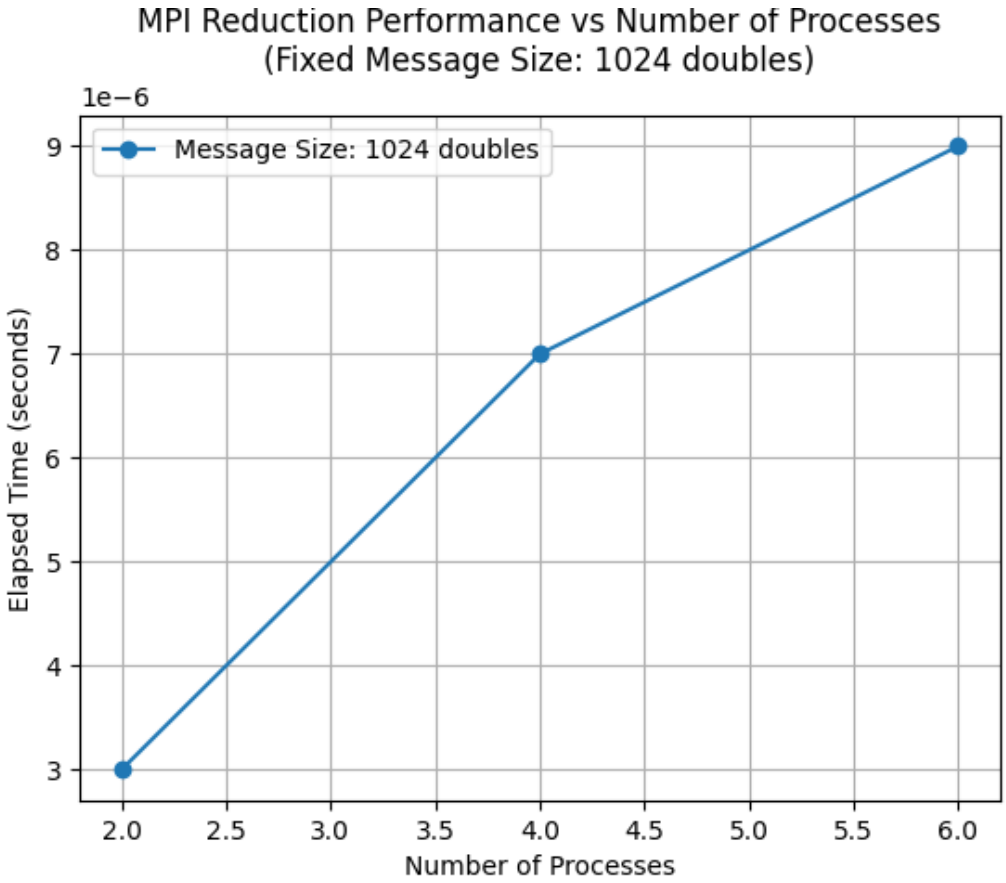
Analysis Summary

- **Smaller Message Sizes (e.g., 512, 1024 bytes)**:
    - The reduction operation is highly efficient, with minimal communication overhead, resulting in very low execution times.
- **Larger Message Sizes (e.g., 2048 bytes and beyond)**:
    - As message sizes increase, the communication cost becomes the dominant factor, leading to increased execution times.
- **Effect of Number of Processes**:
    - While adding more processes can speed up the computation by distributing the work, the communication cost associated with synchronizing all processes can offset these gains for small to moderate message sizes.
- **Communication Bottleneck**:

- For larger message sizes, network bandwidth or the system's communication architecture becomes the limiting factor. This is evidenced by the increase in execution time with more processes and larger messages.

Elapsed Time vs Number of Processes (Example Data)

1. **1024 Doubles:**
   - 2 processes: 0.000003 seconds
   - 4 processes: 0.000007 seconds
   - 6 processes: 0.000009 seconds
2. **2048 Doubles:**
   - 2 processes: 0.000003 seconds
   - 4 processes: 0.000008 seconds
   - 6 processes: 0.000009 seconds



Message Size vs Elapsed Time (Example Data)

1. **2 Processes**:
    - 512 bytes: 0.000003 seconds
    - 1024 bytes: 0.000003 seconds
    - 2048 bytes: 0.000004 seconds
2. **4 Processes**:
    - 512 bytes: 0.000005 seconds
    - 1024 bytes: 0.000006 seconds
    - 2048 bytes: 0.000010 seconds