INDIAN INSTITUTE OF TECHNOLOGY GOA

NSM Nodal Center for Training in HPC and AI

Tutorial: MPI "MATRIX ADDITION" PROGRAM

Authors:

1. **CHODAVARAPU AISHWARYA**, **Project Associate**
2. **Prof. SHARAD SINHA**

CONTENTS:

MPI "Matrix Addition" Program on Param Vidya HPC

Objective:

This tutorial will show you how to create, compile, and run a simple MPI "Matrix Addition" program on the Param Vidya HPC cluster. The goal is to compute the sum of two square matrices using MPI (Message Passing Interface), and calculate the execution time of the program.

Program Explanation:

The program utilizes MPI to distribute the workload of summing two square matrices across multiple processes. Each process computes a partial sum of the row elements of the two matrices assigned to it. The partial sums are then collected by the master process to compute the final sum.

'matrix_addition.c' code:

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

// Function to generate a random matrix with 'rows' x 'cols' elements
void generateRandomMatrix(int* matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i * cols + j] = rand() % 100; // The matrix will be filled with random
numbers (0-99)
        }
    }
}

// Function to write a matrix to a file
void writeMatrix(char* filename, int* matrix, int rows, int cols) {
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file %s\n", filename);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    fprintf(file, "%d %d\n", rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
```

```
            fprintf(file, "%d ", matrix[i * cols + j]);
        }
        fprintf(file, "\n");
    }
    fclose(file);
}

// Function to log matrix size and execution time to 'execution_times.txt'
void logExecutionTime(int size, double execution_time) {
    FILE* file = fopen("matAdd_exec_times_const_proc.txt", "a");
    if (file == NULL) {
        printf("Error opening file matAdd_exec_times_const_proc.txt\n");
        exit(1);
    }
    fprintf(file, "%d %f\n", size, execution_time);  // Matrix size and time
    fclose(file);
}

int main(int argc, char** argv) {
    int rank, size, rows, cols;
    int *matrixA = NULL, *matrixB = NULL, *matrixC = NULL;
    double start_time, end_time, execution_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) { // Only process 0 (root process) executes this
        printf("Enter the number of rows and columns for the matrices: \n");
        scanf("%d %d", &rows, &cols);
        start_time = MPI_Wtime();  // Start timer

        printf("Root process (Process 0) is generating random matrices...\n");
        srand(time(NULL)); // To seed the random number generator with the current
time

        matrixA = (int*)malloc(rows * cols * sizeof(int));
        matrixB = (int*)malloc(rows * cols * sizeof(int));
        generateRandomMatrix(matrixA, rows, cols);
        generateRandomMatrix(matrixB, rows, cols);

        writeMatrix("matA.txt", matrixA, rows, cols);
        writeMatrix("matB.txt", matrixB, rows, cols);

        printf("Root process (Process 0) has written the matrices to matA.txt and
matB.txt.\n");
```

```
    }

    // Broadcast matrix dimensions (rows and cols) from root process to all processes
    MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int active_processes = (size < rows) ? size : rows;
    MPI_Comm active_comm;

    // Split communicator: active processes form one group, idle ones form another
    MPI_Comm_split(MPI_COMM_WORLD, rank < active_processes, rank,
&active_comm);

    if (rank < active_processes) {
        int local_rows = rows / active_processes;
        int extra_rows = rows % active_processes;
        int start_row, end_row;

        if (rank < extra_rows) {
            // First 'extra_rows' processes get an extra row
            start_row = rank * (local_rows + 1);
            end_row = start_row + local_rows;
        } else {
            start_row = rank * local_rows + extra_rows;
            end_row = start_row + local_rows - 1;
        }

        int local_size = (end_row - start_row + 1) * cols;
        int *localA = (int*)malloc(local_size * sizeof(int));
        int *localB = (int*)malloc(local_size * sizeof(int));
        int *localC = (int*)malloc(local_size * sizeof(int));

        if (rank != 0) {
            matrixA = (int*)malloc(rows * cols * sizeof(int));
            matrixB = (int*)malloc(rows * cols * sizeof(int));
        }
        matrixC = (int*)malloc(rows * cols * sizeof(int));

        int *sendcounts = (int*)malloc(active_processes * sizeof(int));
        int *displs = (int*)malloc(active_processes * sizeof(int));

        for (int i = 0; i < active_processes; i++) {
            int process_local_rows = (i < extra_rows) ? local_rows + 1 : local_rows;
            sendcounts[i] = process_local_rows * cols;
            displs[i] = (i < extra_rows) ? i * (local_rows + 1) * cols : (i * local_rows +
extra_rows) * cols;
```

```
        }

        // Scatter matrix A and B from the root process to all processes
        MPI_Scatterv(matrixA, sendcounts, displs, MPI_INT, localA, local_size,
MPI_INT, 0, active_comm);
        MPI_Scatterv(matrixB, sendcounts, displs, MPI_INT, localB, local_size,
MPI_INT, 0, active_comm);

        printf("Process %d is performing matrix addition for rows %d to %d of matA and
matB...\n",
               rank, start_row, end_row);

        // Perform matrix addition on the local chunks
        for (int i = 0; i < local_size; i++) {
            localC[i] = localA[i] + localB[i];
        }

        for (int row = start_row; row <= end_row; row++) {
            int local_row_index = row - start_row;

            printf("Process %d, row %d of matAdd is: ", rank, row);
            for (int i = 0; i < cols; i++) {
                printf("%d ", localC[local_row_index * cols + i]);
            }
            printf("\n");
        }

        // Gather the results from all processes into matrix C (resultant matrix)
        MPI_Gatherv(localC, local_size, MPI_INT, matrixC, sendcounts, displs,
MPI_INT, 0, active_comm);

        free(localA);
        free(localB);
        free(localC);
        free(sendcounts);
        free(displs);
    } else {
        printf("Process %d is idle (no rows assigned).\n", rank);
    }

    MPI_Barrier(MPI_COMM_WORLD);  // Ensure all processes finish before stopping
timer
    end_time = MPI_Wtime();  // Stop timer
    execution_time = end_time - start_time;

    if (rank == 0) {
```

```
        printf("Root process (Process 0) is gathering the results and writing to
matAdd.txt...\n");
        writeMatrix("matAdd.txt", matrixC, rows, cols);
        printf("Execution time: %f seconds\n", execution_time);

        // Log the size of the matrix and execution time to the file
        logExecutionTime(rows, execution_time);
    }

    if (matrixA) free(matrixA);
    if (matrixB) free(matrixB);
    if (matrixC) free(matrixC);

    MPI_Comm_free(&active_comm);
    MPI_Finalize();
    return 0;
}
```

Code Explanation:

1. **Initialization:**
    - MPI_Init(&argc, &argv): This initializes the MPI environment. It must be the first MPI function called in any MPI program.

2. **Process Identification:**
    - MPI_Comm_rank(MPI_COMM_WORLD, &rank): Retrieves the rank (ID) of the calling process. Each process in MPI is identified by its rank, which ranges from 0 to size-1. The root process (rank 0) typically handles I/O operations, while other processes perform calculations.
    - MPI_Comm_size(MPI_COMM_WORLD, &size): Gets the total number of processes in the communicator MPI_COMM_WORLD (default communicator containing all processes).

3. **User Input:**
    - (Root Process rank == 0): The root process (rank 0) prompts the user to input the number of rows and columns for the matrices. This input is then used to generate random matrices matrixA and matrixB of size rows x cols.

4. **Random Matrix Generation:**

- generateRandomMatrix(): Two matrices, matrixA and matrixB, are filled with random values between 0 and 99 using this function.

5. **Timer:**
   - MPI_Wtime() provides the elapsed (wall-clock) time. Here, the start_time and end_time are recorded around the entire computation, and the difference gives the total execution time of the program.

6. **Broadcast Matrix Size:**
   - MPI_Bcast(): The root process broadcasts the matrix dimensions (rows and cols) to all other processes using MPI_Bcast(). This ensures that every process knows the dimensions of the matrices.

7. **Dividing Rows Among Processes:**
   - The total rows are divided among the active processes. If there is a remainder (extra_rows), the first few processes will handle one extra row.

8. **Scatter Matrices to Processes:**
   - MPI_Scatterv(): The matrices matrixA and matrixB are divided and distributed across all active processes using MPI_Scatterv, with each process receiving a chunk of rows. This step allows parallel computation of matrix addition, where each process works on its assigned chunk.

9. **Local Matrix Addition (In Each Process):**
   - After receiving their respective chunks, each process performs matrix addition element-wise. The code loops through each local element of matrixA and matrixB, adds them, and stores the result in localC.

10. **Row Calculation:**
    - Each process is responsible for a specific set of rows. The start_row and end_row variables are used to calculate the row indices for each process based on its rank. These indices determine which portion of the matrices each process operates on.

11. **Gathering Results at Root:**
    - MPI_Gatherv(): The local portions of localC are gathered back at the root process (rank 0) using MPI_Gatherv, forming the complete result matrix matrixC.

12. **Final Output:**

- After gathering all portions, the root process writes the final result matrix matrixC to matAdd.txt. It also prints the total execution time for the computation.
- Also each process prints its assigned rows of the result matrix to the terminal.

13. **Memory Allocation and Deallocation:**

    - Each process dynamically allocates memory for its local portion of the matrices (localA, localB, localC). After completing their work, the memory is freed to avoid memory leaks.

14. **Finalization:**
    - MPI_Finalize(): This function cleans up the MPI environment, ensuring that all processes exit gracefully. It is the last MPI function to be called.

Compilation and Execution on Local Machine:

First you can compile and run the program on your local machine to observe how the code works. To compile and run the program on your local machine, follow these steps:

1. **Install OpenMPI:**
    - Ensure that OpenMPI is installed on your local machine.

2. **Compile the Program:**
    -
    ```
    mpicc -o mpi matrix_addition.c
    ```

3. **Run the Program:**
    -
    ```
    mpirun -np <number_of_processes> mpi
    ```
    - For example, to run with 6 processes, we would use the below command.

    ```
    mpirun -np 6 mpi
    ```

- Then the user will be prompted to enter the number of rows and columns of the input matrices, say the user enters 10 10. Then the output would look like this on the terminal:

```
$ mpirun -np 6 mpi
Enter the number of rows and columns for the matrices:
10 10
Root process (Process 0) is generating random matrices...
Root process (Process 0) has written the matrices to matA.txt and matB.txt.
Process 4 is performing matrix addition for rows 8 to 8 of matA and matB...
Process 4, row 8 of matAdd is: 102 183 61 83 95 147 144 53 111 147
Process 5 is performing matrix addition for rows 9 to 9 of matA and matB...
Process 5, row 9 of matAdd is: 156 12 44 127 145 102 111 127 65 147
Process 1 is performing matrix addition for rows 2 to 3 of matA and matB...
Process 1, row 2 of matAdd is: 105 68 137 77 124 46 26 90 89 89
Process 1, row 3 of matAdd is: 146 18 123 108 109 149 101 148 104 109
Process 2 is performing matrix addition for rows 4 to 5 of matA and matB...
Process 2, row 4 of matAdd is: 97 158 144 90 97 139 163 57 93 59
Process 2, row 5 of matAdd is: 69 51 80 160 133 157 106 160 99 99
Process 0 is performing matrix addition for rows 0 to 1 of matA and matB...
Process 0, row 0 of matAdd is: 76 34 109 191 121 92 86 103 156 144
Process 0, row 1 of matAdd is: 102 33 193 139 43 68 159 101 96 108
Root process (Process 0) is gathering the results and writing to matAdd.txt...
Process 3 is performing matrix addition for rows 6 to 7 of matA and matB...
Process 3, row 6 of matAdd is: 101 97 170 129 57 131 82 110 132 38
Process 3, row 7 of matAdd is: 21 129 49 169 123 51 60 138 112 105
Execution time: 0.008520 seconds
```

- Also the randomly generated input matrices (matA and matB) and output matrix (matAdd) in the text files would be like this:

matA.txt

```
10 10
38 29 46 92 46 49 59 20 99 84
```

```
45 7 94 76 36 44 84 28 71 50
64 53 79 71 25 38 16 13 13 46
95 4 75 41 96 74 91 55 94 90
92 91 98 86 67 86 83 52 14 54
54 31 60 86 54 85 24 71 50 89
17 45 93 45 38 89 71 29 97 17
20 89 9 70 27 28 56 62 32 22
69 87 5 29 73 60 66 49 83 68
90 0 13 84 97 51 25 68 33 74
```

matB.txt

```
10 10
38 5 63 99 75 43 27 83 57 60
57 26 99 63 7 24 75 73 25 58
41 15 58 6 99 8 10 77 76 43
51 14 48 67 13 75 10 93 10 19
5 67 46 4 30 53 80 5 79 5
15 20 20 74 79 72 82 89 49 10
84 52 77 84 19 42 11 81 35 21
1 40 40 99 96 23 4 76 80 83
33 96 56 54 22 87 78 4 28 79
66 12 31 43 48 51 86 59 32 73
```

matAdd.txt

```
10 10
76 34 109 191 121 92 86 103 156 144
102 33 193 139 43 68 159 101 96 108
105 68 137 77 124 46 26 90 89 89
146 18 123 108 109 149 101 148 104 109
97 158 144 90 97 139 163 57 93 59
69 51 80 160 133 157 106 160 99 99
101 97 170 129 57 131 82 110 132 38
21 129 49 169 123 51 60 138 112 105
102 183 61 83 95 147 144 53 111 147
156 12 44 127 145 102 111 127 65 147
```

- If you exceed the processor limit on your local machine, then the terminal might display a message similar to this:

```
$ mpirun -np 9 mpi
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 9
```

```
slots that were requested by the application:

  mpi

Either request fewer procs for your application, or make more slots
available for use.
```

Steps to Run the Program on the Param Vidya HPC Cluster:

To run the MPI program on the Param Vidya HPC cluster, follow these steps:

1. **Log in to the HPC Machine:**
    - Linux systems provide a built-in SSH client, so there is no need to install any additional package. Use SSH to log in to the HPC cluster. You'll need the appropriate credentials and network access.

      ```
      ssh username@hpc_address
      ```

    - For example, to connect to the PARAM Vidya Login Node, we use the above command.

      ```
      ssh username@paramvidya.iitgoa.ac.in -p <port number>
      ```

2. **Transfer sum_of_nums.c to the Cluster:**
    - Use scp to transfer the source code file from your local machine to the cluster:

      ```
      scp –P <port number> -r /path/to/directory/matrix_addition.c <your username>@paramvidya.iitgoa.ac.in:<path to directory on HPC where to save the data>
      ```

    - Otherwise, you can also create a matrix_addition.c file after logging in to the HPC machine using commands like nano, vim etc.

3. **Load openmpi module:**

- To list the modules available on the hpc machine, use the below command:

```
module avail
```

- Identify the correct name of the openmpi module from the list (in my case it is openmpi3/3.1.4). Then, load that module with the command:

```
module load openmpi3/3.1.4
```

This will load the openmpi module on the HPC machine.

4. **Compile the Program:**
   -

```
mpicc -o mpi matrix_addition.c
```

5. **Run the Program:**
   -

```
mpirun -np <number_of_processes> mpi
```

- For example, to run with 16 processes, we would use the below command.

```
mpirun -np 16 mpi
```

- Then the user will be prompted to enter the number of rows and columns of the input matrices, say the user enters 30 30. Then the output would look like this:

```
Enter the number of rows and columns for the matrices:
30 30
Root process (Process 0) is generating random matrices...
Root process (Process 0) has written the matrices to matA.txt and matB.txt.
Process 1 is performing matrix addition for rows 2 to 3 of matA and matB...
```

Process 1, row 2 of matAdd is: 138 88 137 26 112 136 111 34 176 147 91 107 110 172 105 117 71 97 110 128 150 141 135 108 33 91 14 175 146 125
Process 1, row 3 of matAdd is: 132 85 17 121 116 82 58 27 68 138 79 165 146 90 41 56 59 112 105 75 140 108 68 80 69 106 71 84 81 118
Process 2 is performing matrix addition for rows 4 to 5 of matA and matB...
Process 2, row 4 of matAdd is: 114 114 108 84 140 124 170 98 3 44 90 134 161 88 77 102 96 89 67 53 116 111 114 88 92 136 94 68 124 77
Process 2, row 5 of matAdd is: 91 42 95 99 131 135 75 102 139 130 98 129 117 111 21 95 114 118 88 85 75 157 96 94 97 93 82 93 162 59
Process 3 is performing matrix addition for rows 6 to 7 of matA and matB...
Process 3, row 6 of matAdd is: 122 105 53 69 104 85 109 131 92 100 113 94 29 82 106 154 181 172 176 74 61 53 83 57 99 81 103 86 74 117
Process 3, row 7 of matAdd is: 145 48 174 50 69 30 40 131 65 36 31 83 130 165 117 88 72 104 112 149 78 125 54 114 88 106 147 91 92 74
Process 4 is performing matrix addition for rows 8 to 9 of matA and matB...
Process 4, row 8 of matAdd is: 161 41 74 87 143 48 70 88 32 135 124 167 70 8 85 140 148 58 148 14 59 179 91 166 45 79 176 144 123 20
Process 4, row 9 of matAdd is: 122 136 61 149 123 108 149 145 97 133 85 26 153 107 86 191 152 135 101 100 53 64 83 97 82 80 129 110 77 56
Process 5 is performing matrix addition for rows 10 to 11 of matA and matB...
Process 5, row 10 of matAdd is: 130 51 193 95 52 69 56 154 67 106 39 104 84 97 17 123 40 169 62 141 174 116 110 158 165 92 143 146 103 21
Process 5, row 11 of matAdd is: 104 137 76 49 85 130 170 93 136 90 51 128 99 87 126 116 110 66 138 173 112 116 93 122 78 111 118 122 109 73
Process 6 is performing matrix addition for rows 12 to 13 of matA and matB...
Process 6, row 12 of matAdd is: 147 65 63 124 67 100 6 90 45 94 184 49 23 84 88 53 52 104 171 94 81 135 111 27 109 90 90 80 64 99
Process 6, row 13 of matAdd is: 105 112 70 72 137 41 124 195 131 122 142 69 123 118 157 64 171 162 120 95 57 53 83 72 132 97 115 74 129 80
Process 7 is performing matrix addition for rows 14 to 15 of matA and matB...
Process 7, row 14 of matAdd is: 75 140 197 149 112 86 90 190 133 126 64 175 47 88 45 104 56 69 171 77 117 81 130 152 105 116 153 121 90 84
Process 7, row 15 of matAdd is: 106 69 176 155 118 40 41 13 82 26 92 51 154 139 91 152 148 148 74 72 77 43 105 59 47 62 127 101 88 71

Process 8 is performing matrix addition for rows 16 to 17 of matA and matB...
Process 8, row 16 of matAdd is: 89 94 92 117 101 116 110 46 81 97 72 73 148 131 117 92 84 65 92 110 89 121 105 98 133 56 113 113 161 102
Process 8, row 17 of matAdd is: 136 151 100 81 120 154 49 83 152 131 85 25 108 85 109 125 30 97 142 122 159 136 44 68 34 78 176 100 95 138
Process 9 is performing matrix addition for rows 18 to 19 of matA and matB...
Process 9, row 18 of matAdd is: 54 132 89 106 165 115 112 67 50 117 150 87 47 162 73 108 88 156 105 82 130 68 71 127 136 158 109 112 110 5
Process 9, row 19 of matAdd is: 50 117 89 45 128 106 112 45 125 114 62 179 102 61 142 127 69 82 87 26 116 170 146 40 50 82 151 11 47 165
Process 10 is performing matrix addition for rows 20 to 21 of matA and matB...
Process 10, row 20 of matAdd is: 68 102 35 161 151 164 119 115 113 97 129 76 128 135 90 122 167 12 156 154 91 178 125 90 18 79 125 73 142 124
Process 10, row 21 of matAdd is: 139 15 31 75 176 82 143 97 98 156 146 132 136 127 120 127 54 40 192 163 46 87 141 76 177 112 107 55 85 103
Process 11 is performing matrix addition for rows 22 to 23 of matA and matB...
Process 11, row 22 of matAdd is: 132 124 118 115 103 147 51 150 96 53 107 147 185 96 126 58 127 132 150 171 96 49 111 37 130 93 101 89 152 87
Process 11, row 23 of matAdd is: 144 85 116 67 105 124 115 156 75 63 9 86 62 100 135 140 62 115 74 65 91 70 166 155 112 100 100 166 190 53
Process 12 is performing matrix addition for rows 24 to 25 of matA and matB...
Process 12, row 24 of matAdd is: 106 87 91 26 55 148 2 122 157 130 37 70 17 52 122 156 44 185 124 70 154 116 145 125 123 58 125 175 176 119
Process 12, row 25 of matAdd is: 80 86 59 123 64 166 124 68 140 33 150 81 56 119 85 131 129 83 69 105 53 75 73 51 100 96 61 78 123 90
Process 13 is performing matrix addition for rows 26 to 27 of matA and matB...
Process 13, row 26 of matAdd is: 49 157 128 161 132 145 179 61 17 72 94 119 106 103 44 96 87 73 79 156 130 137 35 104 40 136 104 54 118 80
Process 13, row 27 of matAdd is: 96 167 137 176 129 174 174 61 35 91 86 83 64 96 90 160 92 77 85 76 37 115 66 73 171 58 113 76 65 131

```
Process 14 is performing matrix addition for rows 28 to 28 of matA and matB...
Process 14, row 28 of matAdd is: 56 65 50 98 143 84 76 21 98 112 65 36 99 181 32 89 93 129 171 78 157 60 45 75 133 69 87 146 97 156
Process 15 is performing matrix addition for rows 29 to 29 of matA and matB...
Process 15, row 29 of matAdd is: 81 158 122 85 108 69 121 136 90 171 100 60 107 52 93 143 46 39 25 170 69 35 130 116 163 69 89 102 67 39
Process 0 is performing matrix addition for rows 0 to 1 of matA and matB...
Process 0, row 0 of matAdd is: 118 99 91 61 93 145 92 151 135 35 174 71 166 139 113 42 39 106 57 81 101 52 161 93 148 98 147 53 109 83
Process 0, row 1 of matAdd is: 129 33 87 72 94 80 170 40 84 105 127 11 80 145 54 94 139 93 153 149 126 107 154 40 52 154 38 151 59 52
Root process (Process 0) is gathering the results and writing to matAdd.txt...
Execution time: 0.002164 seconds
```

- Since the limit of the number of processors in our hpc machine is 48, we can run the program up to across 48 processors.

Experimentation, Plotting and Observation for various cases:

Now let us observe the execution times for different cases and plot their graphs.
a) Keeping the size of the matrix constant and varying the number of processors.
b) Keeping the number of processors constant and varying the size of the matrix.

a) Keeping the matrix size constant and varying the number of processors:

CASE 1: Size of matrix = 100, No. of processors = 4, 8, 16, 32, 48.

1. **Modify the code:**
    - Modify the code to set the size of the matrix to a constant, say 100. Or the user can always enter the same size while executing the program for this case. Also add a file handling part in the code to create a text file, say matAdd_exec_times_const_size.txt, to store the execution times run across different number of processors.

2. **Compile and Run the program:**
    - Now let us run the code across different number of processors, say 4, 8, 16, 32, 48. Run the program at least for 10 times and take the average

execution time for each processor for better results.

- The generated matAdd_exec_times_const_size.txt would look like this:

```
4 0.008220
8 0.008355
16 0.008421
32 0.009552
48 0.120967
```

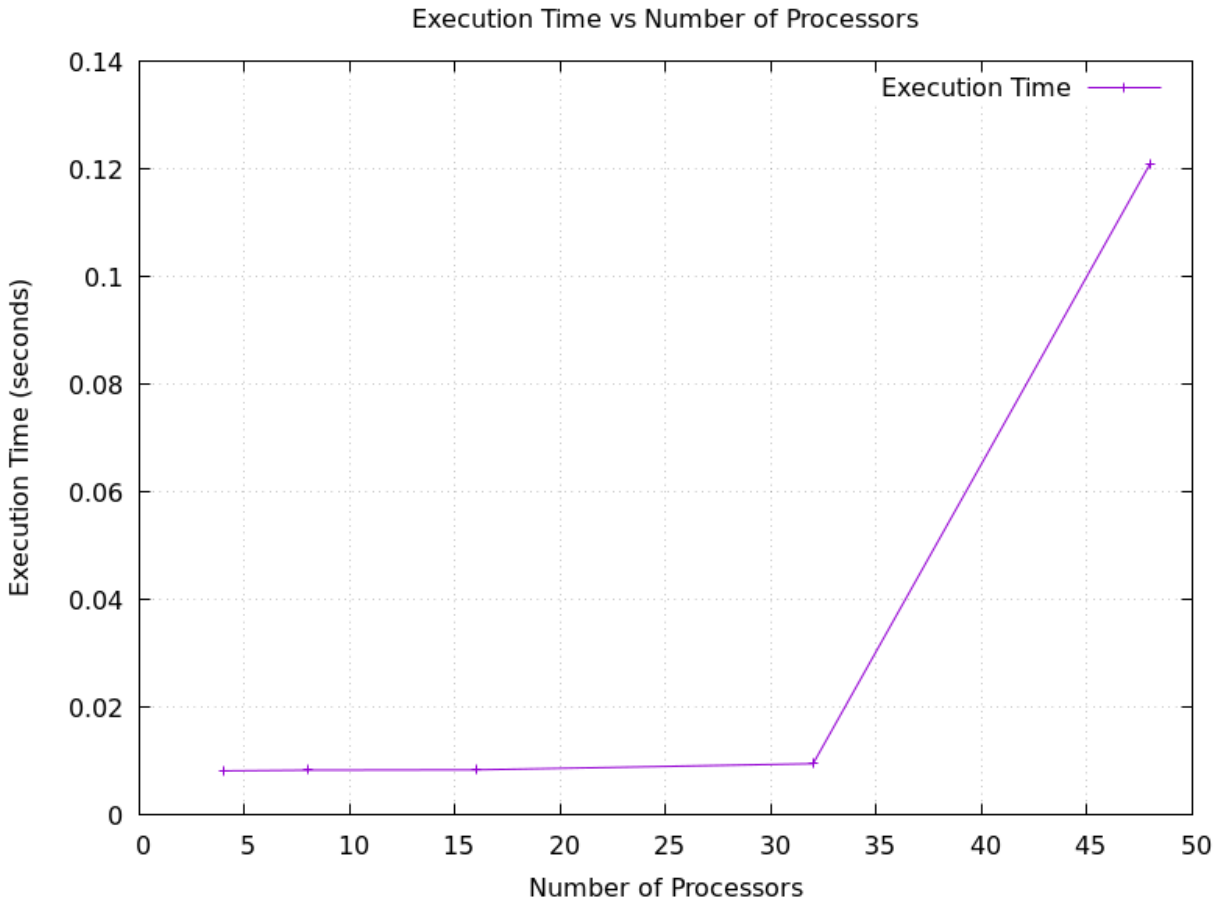
3. **Transfer the text file to your local machine:**
   - Use the scp command to send the matAdd_exec_times_const_size.txt file from the cluster to your local machine.

4. **Generate plot:**
   - Now to generate a plot, you can install gnuplot on your local machine.

     matAdd_const_size.gnuplot code:

```
set terminal pngcairo size 800,600
set output 'matAdd_exec_times_const_size.png'
set title "Execution Time vs Number of Processors"
set xlabel "Number of Processors"
set ylabel "Execution Time (seconds)"
set grid
plot "matAdd_exec_times_const_size.txt" using 1:2 with linespoints title "Execution Time"
```

   - And the corresponding plot would look like this:



- **Observation:** For a small matrix size (100 x 100), the execution time remained nearly constant regardless of the number of processors (4, 8, 16, 32). However, there was a noticeable increase in execution time when using a higher number of processors (48). This is likely due to the overhead involved in managing communication between a larger number of processors, which outweighs the computational benefits for such a small matrix.

CASE 2: Matrix Size = 1000, No. of processors = 4, 8, 16, 32, 48.

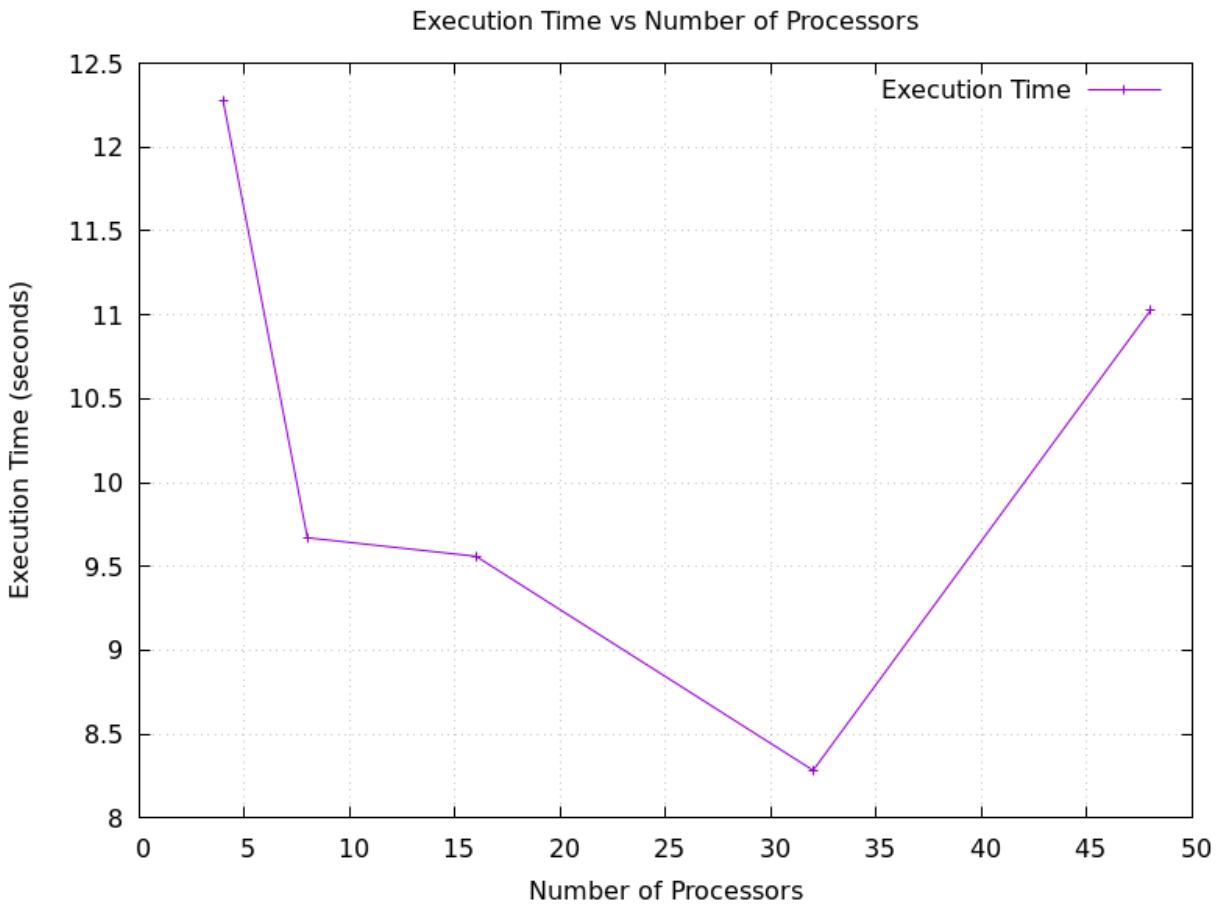
1. **Modify the code:**
   - Modify the code to set the size of the matrix to a constant, say 1000.

2. **Compile and Run the program:**
   - Now let us run the code across different number of processors, say 4, 8, 16, 32, 48.

- The generated matAdd_exec_times_const_size.txt would look like this:

```
4 12.278454
8 9.670586
16 9.560235
32 8.284315
48 11.02440
```

3. **Generate plot:**
   - The corresponding plot would look like this:



   - **Observation:** With a medium-sized matrix (1000 x 1000), the execution time decreased as the number of processors increased, which reflects more efficient parallel computation. However, in the case with 48 processors, there was a slight anomaly where the execution time increased, indicating that the communication overhead might have surpassed the gains from parallelism at this processor count.

CASE 3: No. of elements = 2500, No. of processors = 4, 8, 16, 32, 48.

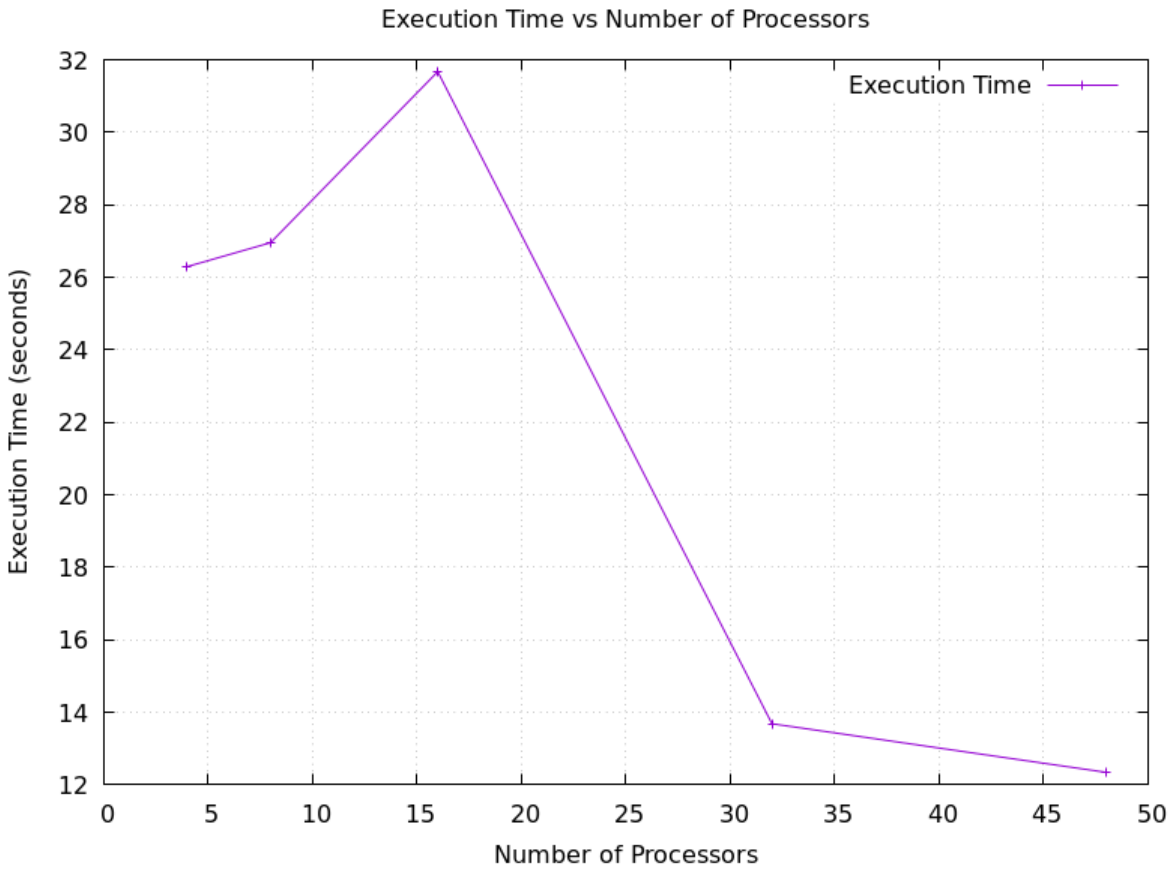
1. **Modify the code:**
   - Modify the code to set the size of the matrix to a constant, say 2500.

2. **Compile and Run the program:**
   - Now let us run the code across different number of processors, say 4, 8, 16, 32, 48.

   - The generated matAdd_exec_times_const_size.txt would look like this:

```
4 26.292243
8 26.954541
16 31.671684
32 13.682158
48 12.340749
```

3. **Generate plot:**
   - The corresponding plot would look like this:



- **Observation:** For a large matrix (2500 x 2500), the execution time remained nearly constant with 4, 8, and 16 processors. However, significant reductions in execution time were observed when running with more processors (32, 48). This suggests that for larger matrices, the benefits of distributing the computation across more processors become more pronounced, improving overall performance.

b) Keeping the number of processors constant and varying the matrix size:

CASE 1: No. of processors = 48, Matrix Size = 50, 100, 500, 1000, 2500.

1. **Modify the code:**
   - Modify the code to prompt the user to enter the size of the matrix. Also add a file handling part in the code to create a text file, say matAdd_exec_times_const_proc.txt, to store the execution times run for different sizes of the matrix.

2. **Compile and Run the program:**
   - Let us run the code across a constant number of processors, say 48. And enter a different matrix size in each run, say 1000, 2000, 3000, 4000, 5000, 6000.

   - The generated matAdd_exec_times_const_proc.txt would look like this:

```
50 0.052836
100 0.090037
500 0.220396
1000 2.828991
2500 15.822379
```

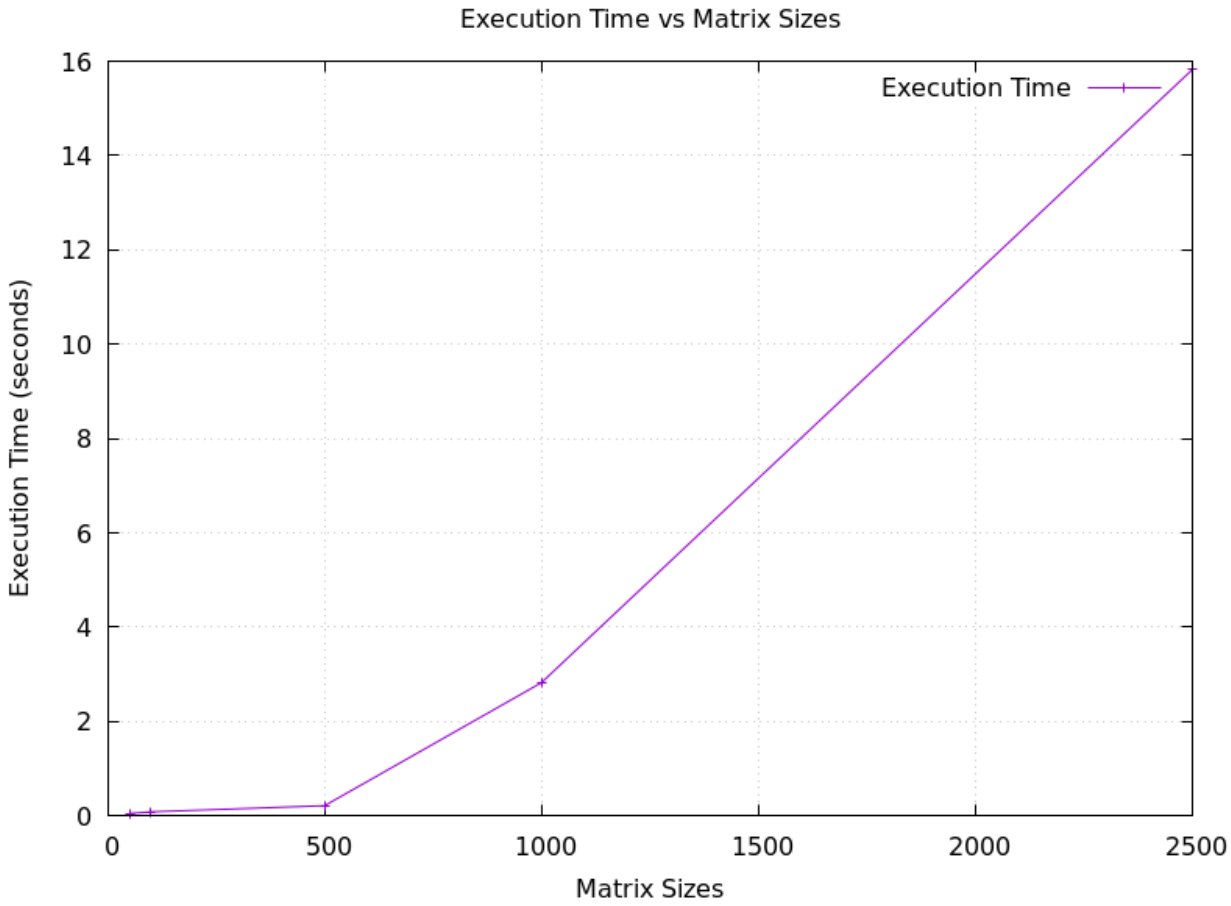
3. **Transfer the text file to your local machine:**
   - Use the scp command to send the matAdd_exec_times_const_proc.txt file from the cluster to your local machine.

4. **Generate plot:**

- matAdd_exec_times_const_proc.gnuplot code:

```
set terminal pngcairo size 800,600
set output 'matAddexec_times_const_proc.png'
set title "Execution Time vs Matrix Sizes"
set xlabel "Matrix Sizes"
set ylabel "Execution Time (seconds)"
set grid
plot "matAdd_exec_times_const_proc.txt" using 1:2 with linespoints title
"Execution Time"
```

- And the corresponding plot would look like this:



- **Observation:** Here, we observe there is an increase in execution times as the matrix size increases.

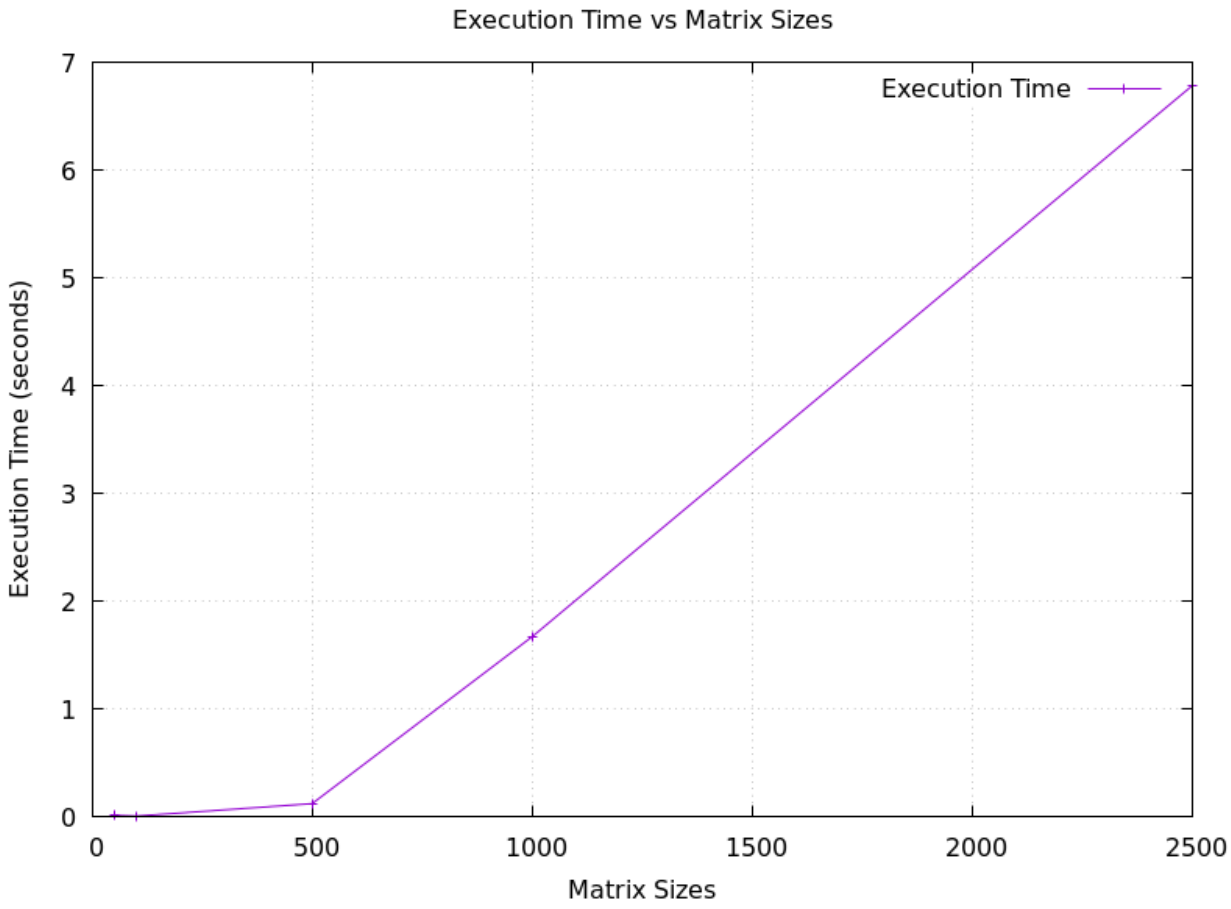
CASE 2: No. of processors = 32, Matrix Size = 50, 100, 500, 1000, 2500.

1. **Compile and Run the program:**
    - Run the code across a constant number of processors, say 32. And enter a different matrix size in each run, say 50, 100, 500, 1000, 2500.

    - The generated matAdd_exec_times_const_proc.txt would look like this:

```
50 0.016971
100 0.009197
500 0.124265
1000 1.672535
2500 6.783325
```

2. **Generate plot:**
    - The corresponding plot would like this:



    - **Observation:** Here, we observe an increase in execution time as the matrix size increases. This is not always the case, but since we have

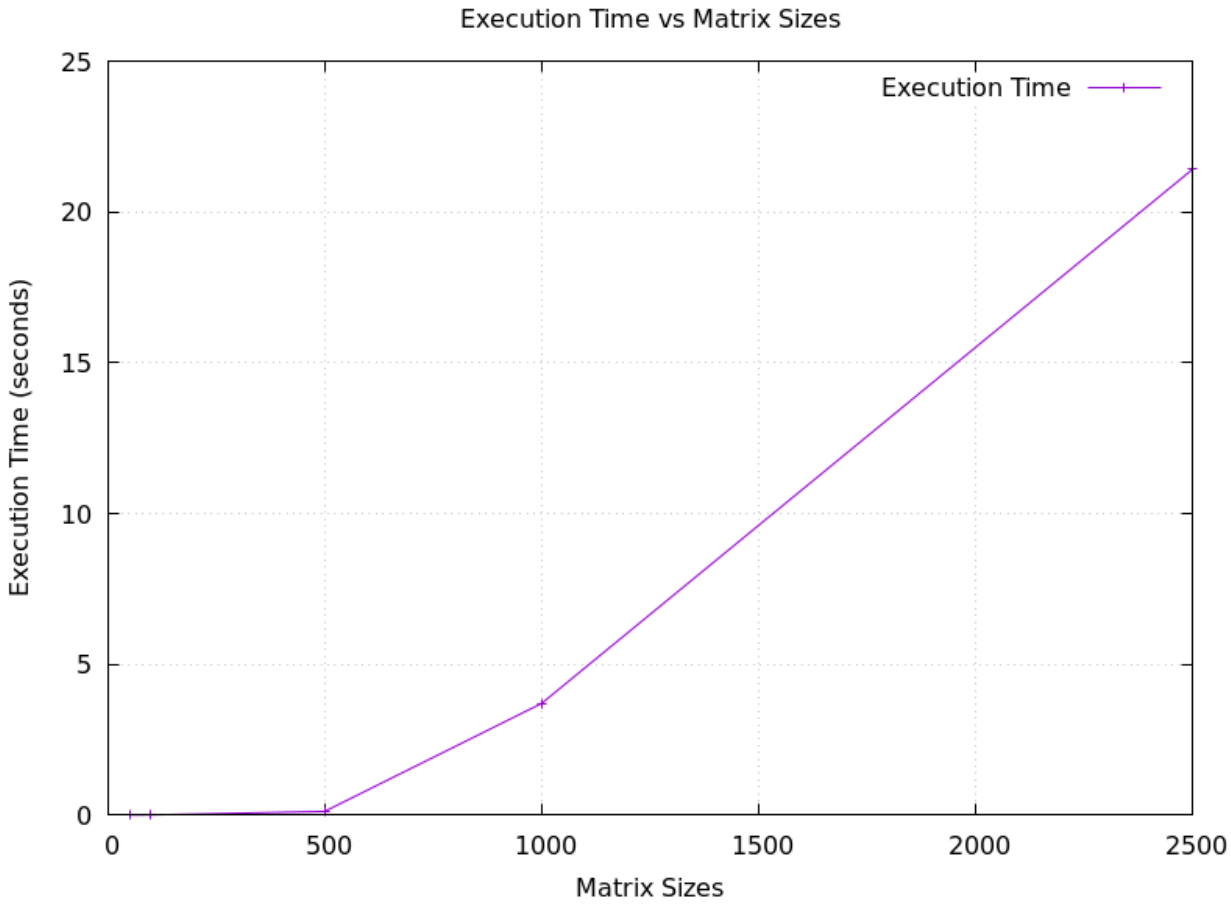
taken the average values of execution times, it seems to increase as the matrix sizes increases.

CASE 3: No. of processors = 16, Matrix Size = 50, 100, 500, 1000, 2500.

1. **Compile and Run the program:**
    - Run the code across a constant number of processors, say 16. And enter a different matrix size in each run, say 50, 100, 500, 1000, 2500.
    - The generated matAdd_exec_times_const_proc.txt would look like this:

```
50 0.003331
100 0.008179
500 0.123696
1000 3.703877
2500 21.408024
```

2. **Generate plot:**
    - The corresponding plot would like this:



- **Observation:** Here, we observe there is an increase in execution times as the matrix size increases.

Conclusion:

- **a) Keeping the matrix size constant and varying the number of processors:** As the number of processors increases, the relationship between execution time and processor count does not follow a steady pattern. While using more processors generally reduces execution time, particularly for larger matrices, too many processors can lead to inefficiencies due to increased communication overhead. This effect is especially visible when running with a higher number of processors (e.g., 48), where the communication costs can start to dominate the computation time, especially for smaller matrices.

- **b) Keeping the number of processors constant and varying the matrix size:** Across all the three cases here, there is a general increase in execution time as the size of the matrices grows. This is expected because larger matrices require more computation for each process to handle, and thus, the overall time to complete the matrix addition increases. Therefore the effect of matrix size on execution time becomes more noticeable as the size of the matrix increases.