INDIAN INSTITUTE OF TECHNOLOGY GOA

NSM Nodal Center for Training in HPC and AI

Tutorial: MPI "SUM OF NUMBERS" PROGRAM

Authors:

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

CONTENTS:

➢ Objective
➢ Program Explanation
➢ Compilation and Execution on Local machine
➢ Steps to Run the Program on the Param Vidya HPC Cluster
➢ Experimentation, Plotting and Observation for various cases
a) Keeping the no. of elements constant and varying the no. of processors
b) Keeping the no. of processors constant and varying the no. of elements
➢ Conclusion
➢ Steps to run the program using Slurm Script file on HPC machine

MPI "Sum of Integers from 1 to n" Program on Param Vidya HPC

Objective:

This tutorial will show you how to create, compile, and run a simple MPI "Sum of Numbers" program on the Param Vidya HPC cluster. The goal is to compute the sum of an array of numbers 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 an array of numbers across multiple processes. Each process computes a partial sum of the array elements assigned to it. The partial sums are then collected by the master process to compute the final sum.

'sum_of_nums.c' code:

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

int main(int argc, char* argv[])
{
    int pid, np, elements_per_process, n_elements_received;
    int n; // number of elements
    int* a = NULL; // dynamic array
    int* a2 = NULL; // temporary array for slave process
    MPI_Status status;

    // Creation of parallel processes
    MPI_Init(&argc, &argv);

    double start_time, end_time;

    // find out process ID and how many processes were started
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    // master process
    if (pid == 0) {
        printf("Enter the number of elements: \n");
        scanf("%d", &n);

        // Start the timer
        start_time = MPI_Wtime();

        // allocate array dynamically
        a = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            a[i] = i + 1;
        }

        elements_per_process = n / np;

        // check if more than 1 process is run
        if (np > 1) {
            // distributes the portion of array
            // to child processes to calculate
            // their partial sums
            for (int i = 1; i < np - 1; i++) {
                int index = i * elements_per_process;

                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
                MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0,
MPI_COMM_WORLD);
            }

            // last process adds remaining elements
            int index = (np - 1) * elements_per_process;
            int elements_left = n - index;

            MPI_Send(&elements_left, 1, MPI_INT, np - 1, 0, MPI_COMM_WORLD);
            MPI_Send(&a[index], elements_left, MPI_INT, np - 1, 0,
MPI_COMM_WORLD);
        }

        // master process add its own sub array
        long long int sum = 0;
        for (int i = 0; i < elements_per_process; i++) {
            sum += a[i];
        }
        printf("Sum calculated by root process is %lld\n", sum);

        // collects partial sums from other processes
        for (int i = 1; i < np; i++) {
          long long int tmp;
          MPI_Recv(&tmp, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD, &status);
          printf("Partial sum returned from process %d is %lld\n", status.MPI_SOURCE,
tmp);
          sum += tmp;
       }

       // prints the final sum of array
       printf("Sum of array is : %lld\n", sum);

       // free the allocated memory
       free(a);
    }
    // slave processes
    else {
       MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);

       // allocate temporary array dynamically
       a2 = (int*)malloc(n_elements_received * sizeof(int));

       // stores the received array segment in local array a2
       MPI_Recv(a2, n_elements_received, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);

       // calculates its partial sum
       long long int partial_sum = 0;
       for (int i = 0; i < n_elements_received; i++) {
          partial_sum += a2[i];
       }


       // sends the partial sum to the root process
       MPI_Send(&partial_sum, 1, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);

       // free the allocated memory
       free(a2);
    }

    // Stop the timer
    end_time = MPI_Wtime();

    // Calculate and print the total execution time in the master process
    if (pid == 0) {
       printf("Total execution time: %f seconds\n", end_time - start_time);
    }

    // cleans up all MPI state before exit of process
    MPI_Finalize();

    return 0;
}
```

Code Explanation:

1. **Initialization:**
    - MPI_Init(&argc, &argv): Initializes the MPI environment.

2. **Process Identification:**
    - MPI_Comm_rank(MPI_COMM_WORLD, &pid): Determines the rank (ID) of the calling process.
    - MPI_Comm_size(MPI_COMM_WORLD, &np): Retrieves the total number of processes.
    - A communicator holds a group of processes that can communicate with each other. All message passing calls in MPI must have a specific communicator to use the call with. MPI_COMM_WORLD is the default communicator that contains all processes available for use.

3. **User Input (Master Process pid == 0):**
    - The master process (pid == 0) prompts the user to enter the number of elements n. This input determines the range of numbers that will be summed. For example, if the user inputs n = 1000, the program will calculate the sum of numbers from 1 to 1000.

4. **Timer:**
    - The elapsed (wall-clock) time between two points in an MPI program can be computed using MPI_Wtime():
    - We take the difference between start_time and end_time to calculate the total execution time of the program.

5. **Master Process (pid == 0):**
    - The master process prompts the user to input the number of elements n.
    - The master process dynamically allocates an array of size n and initializes it with values from 1 to n.

- The array is divided among the available processes, with each process receiving a portion of the array to calculate its partial sum.
- The master process also calculates its own portion of the sum and gathers the partial sums from other processes.
- Finally, it prints the total sum and the execution time.

6. **Slave Processes (pid != 0):**
    - Each slave process receives a portion of the array, calculates its partial sum, and sends the result back to the master process.

7. **Final Output:**
    - The master process prints the final sum of numbers from 1 to n.
    - It also prints the total execution time for the computation.

8. **Finalization:**
    - MPI_Finalize(): Cleans up the MPI environment. This should be called after all MPI-related operations are complete.

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 sum_of_nums.c
    ```

3. **Run the Program:**
    -
    ```
    mpirun -np <number_of_processes> mpi
    ```
    - For example, to run with 4 processes, we would use the below command.

```
mpirun -np 4 mpi
```

- Then the user will be prompted to enter the number of elements, say the user enters 1000. Then the output would look like this:

```
$ mpirun -np 4 mpi
Enter the number of elements:
1000
Sum calculated by root process is 31375
Partial sum returned from process 1 is 93875
Partial sum returned from process 2 is 156375
Partial sum returned from process 3 is 218875
Sum of array is : 500500
Total execution time: 0.000561 seconds
```

- 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 below command:

```
ssh username@<address of PARAM Vidya> -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/sum_of_nums.c <your
username>@paramvidya.iitgoa.ac.in:<path to directory on HPC where
to save the data>
```

   - Otherwise, you can also create a sum_of_nums.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 sum_of_nums.c
```

5. **Run the Program:**

-

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

- For example, to run with 4 processes, we would use the below command.

```
mpirun -np 4 mpi
```

- Then the user will be prompted to enter the number of elements, say the user enters 1000. Then the output would look like this:

```
Enter the number of elements:
1000
Sum calculated by root process is 31375
Partial sum returned from process 1 is 93875
Partial sum returned from process 2 is 156375
Partial sum returned from process 3 is 218875
Sum of array is : 500500
Total execution time: 0.000466 seconds
```

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

```
Enter the number of elements:
1000
Sum calculated by root process is 210
Partial sum returned from process 2 is 1010
Partial sum returned from process 1 is 610
Partial sum returned from process 3 is 1410
Partial sum returned from process 4 is 1810
Partial sum returned from process 5 is 2210
Partial sum returned from process 6 is 2610
Partial sum returned from process 8 is 3410
Partial sum returned from process 9 is 3810
Partial sum returned from process 10 is 4210
Partial sum returned from process 11 is 4610
Partial sum returned from process 12 is 5010
Partial sum returned from process 13 is 5410
Partial sum returned from process 14 is 5810
Partial sum returned from process 15 is 6210
Partial sum returned from process 16 is 6610
Partial sum returned from process 17 is 7010
Partial sum returned from process 18 is 7410
Partial sum returned from process 7 is 3010
Partial sum returned from process 21 is 8610
Partial sum returned from process 19 is 7810
Partial sum returned from process 20 is 8210
Partial sum returned from process 22 is 9010
Partial sum returned from process 23 is 9410
Partial sum returned from process 24 is 9810
Partial sum returned from process 25 is 10210
Partial sum returned from process 26 is 10610
Partial sum returned from process 27 is 11010
Partial sum returned from process 28 is 11410
Partial sum returned from process 29 is 11810
Partial sum returned from process 30 is 12210
Partial sum returned from process 31 is 12610
Partial sum returned from process 32 is 13010
Partial sum returned from process 33 is 13410
Partial sum returned from process 34 is 13810
Partial sum returned from process 35 is 14210
Partial sum returned from process 36 is 14610
Partial sum returned from process 37 is 15010
Partial sum returned from process 38 is 15410
Partial sum returned from process 40 is 16210
Partial sum returned from process 39 is 15810
Partial sum returned from process 41 is 16610
Partial sum returned from process 42 is 17010
Partial sum returned from process 43 is 17410
Partial sum returned from process 44 is 17810
Partial sum returned from process 45 is 18210
Partial sum returned from process 46 is 18610
Partial sum returned from process 47 is 58230
Sum of array is : 500500
Total execution time: 0.001059 seconds
```

Experimentation, Plotting and Observation for various cases:

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

a) Keeping the number of elements constant and varying the number of processors:

<u>CASE 1: No. of elements = 1000, No. of processors = 2, 4, 8, 16, 32, 48.</u>

1. **Modify the code:**
    - Modify the code to set the number of elements to a constant, say 1000. Also add a file handling part in the code to create a text file, say exec_times_const_elem.txt, to store the execution times run across different number of processors.

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

int main(int argc, char* argv[])
{
    int pid, np, elements_per_process, n_elements_received;
    int n = 1000; // Fixed number of elements
    int* a = NULL; // dynamic array
    int* a2 = NULL; // temporary array for slave process
    MPI_Status status;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Find out process ID and how many processes were started
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    // Master process
    if (pid == 0) {
        // Start the timer after user input
        double start_time = MPI_Wtime();
        // Allocate array dynamically
        a = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            a[i] = i + 1;
        }

        elements_per_process = n / np;

        // Distribute the portion of array to child processes
        if (np > 1) {
            for (int i = 1; i < np - 1; i++) {
                int index = i * elements_per_process;

                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
                MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0,
MPI_COMM_WORLD);
            }

            // Last process adds remaining elements
            int index = (np - 1) * elements_per_process;
            int elements_left = n - index;

            MPI_Send(&elements_left, 1, MPI_INT, np - 1, 0,
MPI_COMM_WORLD);
            MPI_Send(&a[index], elements_left, MPI_INT, np - 1, 0,
MPI_COMM_WORLD);
        }

        // Master process calculates its own partial sum
        long long int sum = 0;
        for (int i = 0; i < elements_per_process; i++) {
            sum += a[i];
        }
        printf("Sum calculated by root process is %lld\n", sum);

        // Collect partial sums from other processes
        for (int i = 1; i < np; i++) {
            long long int tmp;
            MPI_Recv(&tmp, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD, &status);
            printf("Partial sum returned from process %d is %lld\n",
status.MPI_SOURCE, tmp);
            sum += tmp;
        }
        // Print the final sum of array
        printf("Sum of array is : %lld\n", sum);

        // Stop the timer and calculate the total execution time
        double end_time = MPI_Wtime();
        double execution_time = end_time - start_time;

        // Save the execution time to a file
        FILE *f = fopen("exec_times_const_elem2.txt", "a");
        fprintf(f, "%d %f\n", np, execution_time);
        fclose(f);

        // Free the allocated memory
        free(a);
    }
    // Slave processes
    else {
        MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

        // Allocate temporary array dynamically
        a2 = (int*)malloc(n_elements_received * sizeof(int));

        // Store the received array segment in local array a2
        MPI_Recv(a2, n_elements_received, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

        // Calculate its partial sum
        long long int partial_sum = 0;
        for (int i = 0; i < n_elements_received; i++) {
            partial_sum += a2[i];
        }
        //printf("Partial sum returned from process %d is %d\n", pid,
partial_sum);

        // Send the partial sum to the root process
        MPI_Send(&partial_sum, 1, MPI_LONG_LONG, 0, 0,
MPI_COMM_WORLD);

        // Free the allocated memory
        free(a2);
    }

    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

2. **Compile and Run the program:**
    - Now let us run the code across different number of processors, say 2, 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 exec_times_const_elem.txt would look like this:

```
2 0.000232
4 0.000430
8 0.000898
16 0.000930
32 0.000837
48 0.001102
```

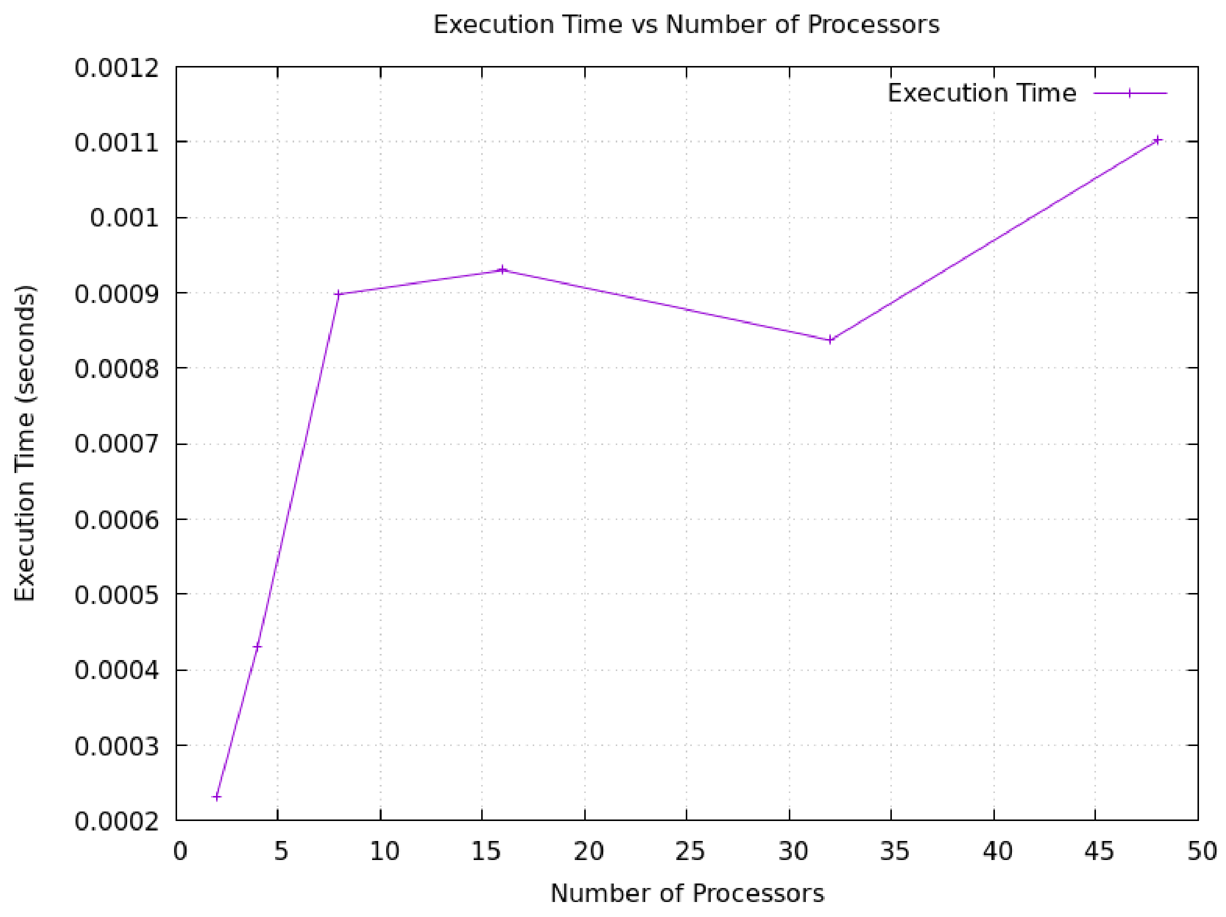
3. **Transfer the text file to your local machine:**
    - Use the scp command to send the exec_times_const_elem.txt file from the cluster to your local machine.

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

    plot1.gnuplot code:

```
set terminal pngcairo size 800,600
set output 'exec_times_const_elem.png'
set title "Execution Time vs Number of Processors"
set xlabel "Number of Processors"
set ylabel "Execution Time (seconds)"
set grid
plot "exec_times_const_elem2.txt" using 1:2 with linespoints title
"Execution Time"
```

    - And the corresponding plot would look like this:



<u>CASE 2: No. of elements = 10000, No. of processors = 2, 4, 8, 16, 32, 48.</u>

1. **Modify the code:**
   - Modify the code to set the number of elements to a constant, say 10000.

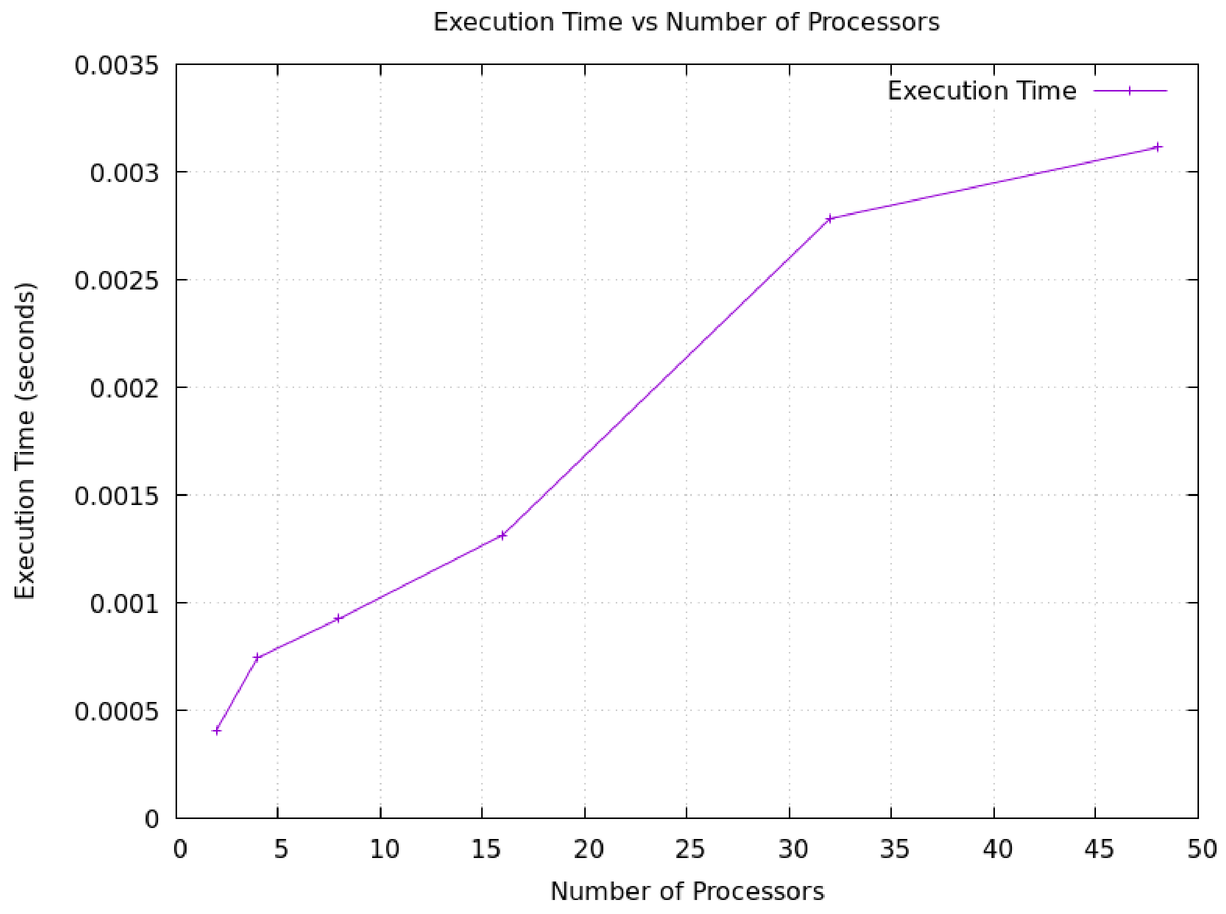
2. **Compile and Run the program:**
   - Now let us run the code across different number of processors, say 2, 4, 8, 16, 32, 48.

   - The generated exec_times_const_elem.txt would look like this:

```
2 0.000411
4 0.000746
8 0.000928
16 0.001316
```

```
32 0.002784
48 0.003114
```

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



    -

<u>CASE 3: No. of elements = 100000, No. of processors = 2, 4, 8, 16, 32, 48.</u>

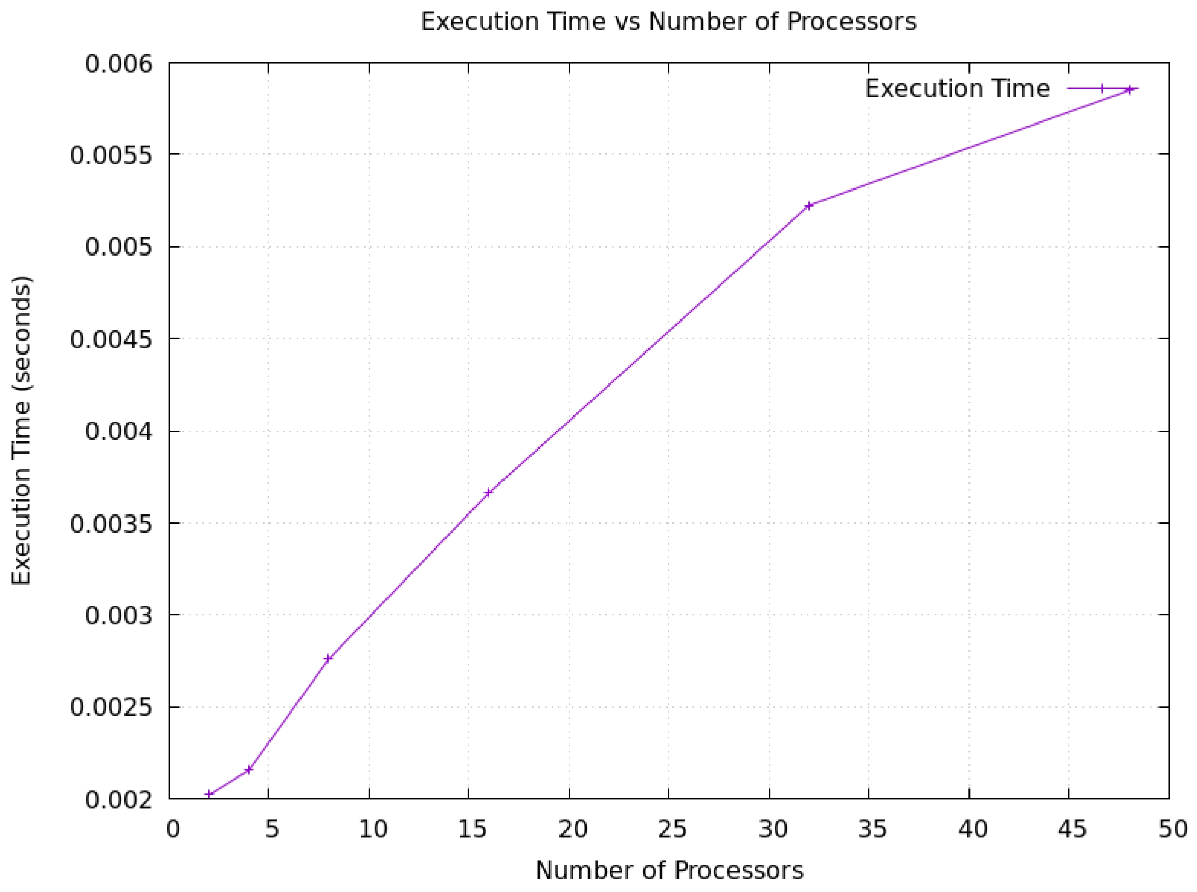
1. **Modify the code:**
    - Modify the code to set the number of elements to a constant, say 100000.

2. **Compile and Run the program:**

- Now let us run the code across different number of processors, say 2, 4, 8, 16, 32, 48.
- The generated exec_times_const_elem.txt would look like this:

```
2 0.002024
4 0.002157
8 0.002762
16 0.003664
32 0.005226
48 0.005850
```

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



-

b) Keeping the number of processors constant and varying the number of elements:

<u>CASE 1: A) No. of processors = 48, No. of elements = 1000, 2000, 3000, 4000, 5000, 6000.</u>

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

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

int main(int argc, char* argv[])
{
   int pid, np, elements_per_process, n_elements_received;
   int n; // Number of elements
   int* a = NULL; // dynamic array
   int* a2 = NULL; // temporary array for slave process
   MPI_Status status;

   // Initialize the MPI environment
   MPI_Init(&argc, &argv);

   // Find out process ID and how many processes were started
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
   MPI_Comm_size(MPI_COMM_WORLD, &np);

   // Master process
   if (pid == 0) {
      // User input for number of elements
      fprintf(stderr, "Enter the number of elements: \n");
      scanf("%d", &n);

      // Start the timer after user input
      double start_time = MPI_Wtime();

      // Allocate array dynamically
      a = (int*)malloc(n * sizeof(int));
      for (int i = 0; i < n; i++) {
            a[i] = i + 1;
        }

        elements_per_process = n / np;

        // Distribute the portion of array to child processes
        if (np > 1) {
            for (int i = 1; i < np - 1; i++) {
                int index = i * elements_per_process;

                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
                MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0,
MPI_COMM_WORLD);
            }

            // Last process adds remaining elements
            int index = (np - 1) * elements_per_process;
            int elements_left = n - index;

            MPI_Send(&elements_left, 1, MPI_INT, np - 1, 0,
MPI_COMM_WORLD);
            MPI_Send(&a[index], elements_left, MPI_INT, np - 1, 0,
MPI_COMM_WORLD);
        }

        // Master process calculates its own partial sum
        long long int sum = 0;
        for (int i = 0; i < elements_per_process; i++) {
            sum += a[i];
        }
        printf("Sum calculated by root process is %lld\n", sum);

        // Collect partial sums from other processes
        for (int i = 1; i < np; i++) {
            long long int tmp;
            MPI_Recv(&tmp, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD, &status);
            printf("Partial sum returned from process %d is %lld\n",
status.MPI_SOURCE, tmp);
            sum += tmp;
        }

        // Print the final sum of array
        printf("Sum of array is : %lld\n", sum);
        // Stop the timer and calculate the total execution time
        double end_time = MPI_Wtime();
        double execution_time = end_time - start_time;

        // Save the number of elements and execution time to a file
        FILE *f = fopen("exec_times_const_proc2.txt", "a");
        fprintf(f, "%d %f\n", n, execution_time);
        fclose(f);

        // Free the allocated memory
        free(a);
    }
    // Slave processes
    else {
        MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

        // Allocate temporary array dynamically
        a2 = (int*)malloc(n_elements_received * sizeof(int));

        // Store the received array segment in local array a2
        MPI_Recv(a2, n_elements_received, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

        // Calculate its partial sum
        long long int partial_sum = 0;
        for (int i = 0; i < n_elements_received; i++) {
            partial_sum += a2[i];
        }
        //printf("Partial sum returned from process %d is %d\n", pid,
partial_sum);

        // Send the partial sum to the root process
        MPI_Send(&partial_sum, 1, MPI_LONG_LONG, 0, 0,
MPI_COMM_WORLD);

        // Free the allocated memory
        free(a2);
    }

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
}
```

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

    - The generated exec_times_const_proc.txt would look like this:

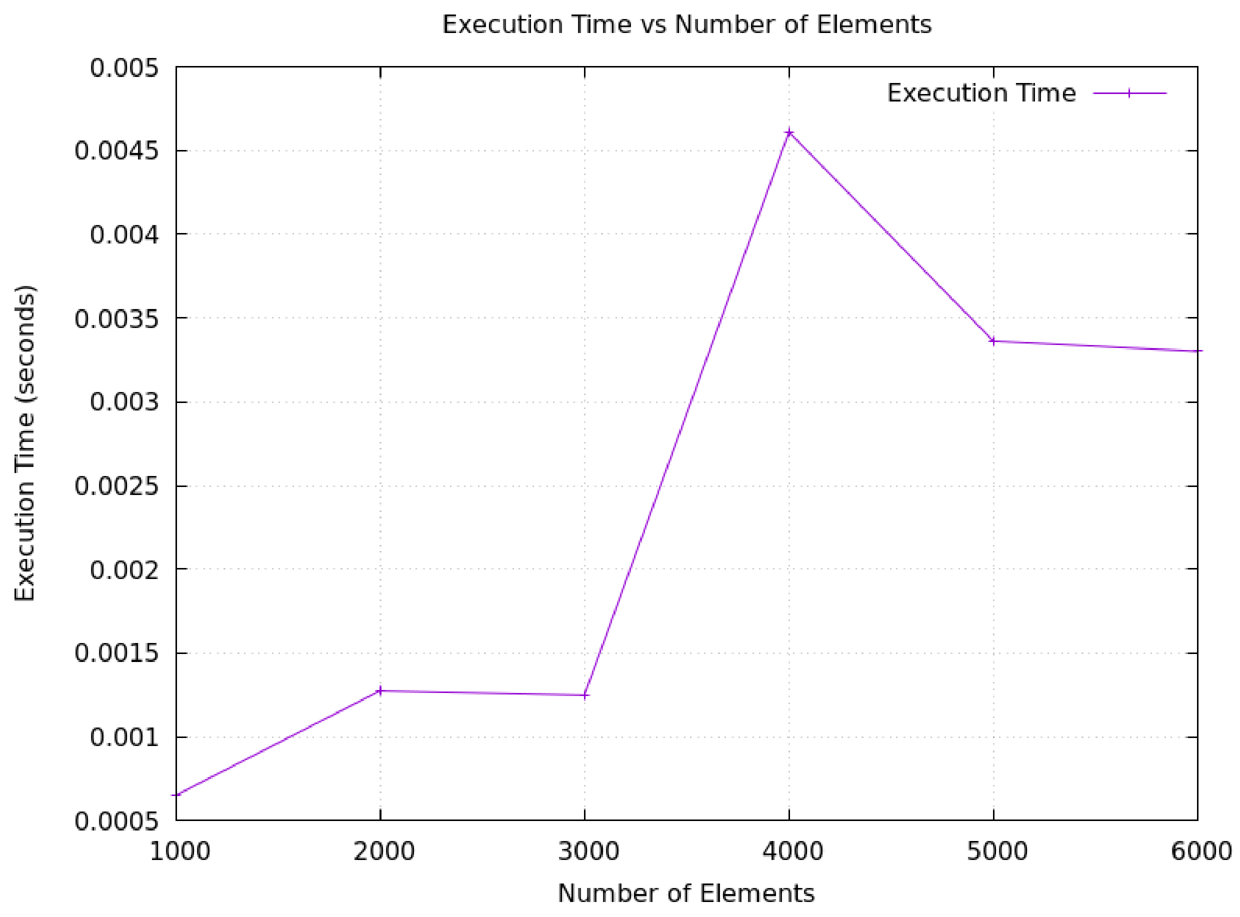
```
1000 0.000651
2000 0.001275
3000 0.001249
4000 0.004609
5000 0.003362
6000 0.003301
```

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

4. **Generate plot:**
    - plot2.gnuplot code:

```
set terminal pngcairo size 800,600
set output 'exec_times_const_proc.png'
set title "Execution Time vs Number of Elements"
set xlabel "Number of Elements"
set ylabel "Execution Time (seconds)"
set grid
plot "exec_times_const_proc2.txt" using 1:2 with linespoints title
"Execution Time"
```

    - And the corresponding plot would look like this:



- **Observation:** The execution times do not show a consistent trend. The times fluctuate without a steady increase or decrease as the number of elements increases.

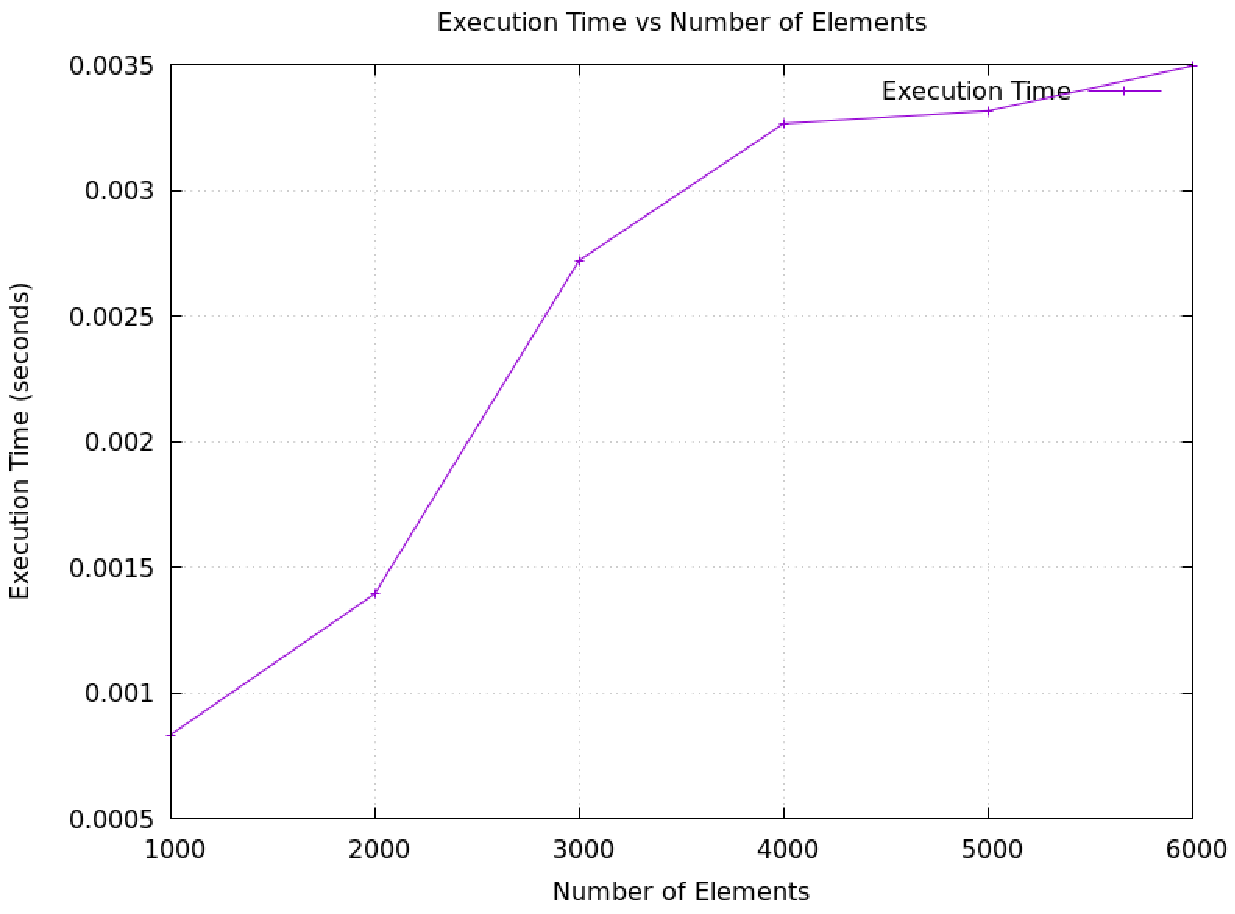
<u>CASE 1: B) No. of processors = 32, No. of elements = 1000, 2000, 3000, 4000, 5000, 6000.</u>

1. **Compile and Run the program:**
    - Run the code across a constant number of processors, say 32. And enter a different number of elements in each run, say 1000, 2000, 3000, 4000, 5000, 6000.

    - The generated exec_times_const_proc.txt would look like this:

```
1000 0.000834
2000 0.001395
3000 0.002721
4000 0.003265
5000 0.003315
6000 0.003494
```

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



   - **Observation:** Here, we observe an increase in execution time as the number of elements increases. This is not always the case, but since we have taken the average values of execution times, it seems to increase as the number of elements increases.

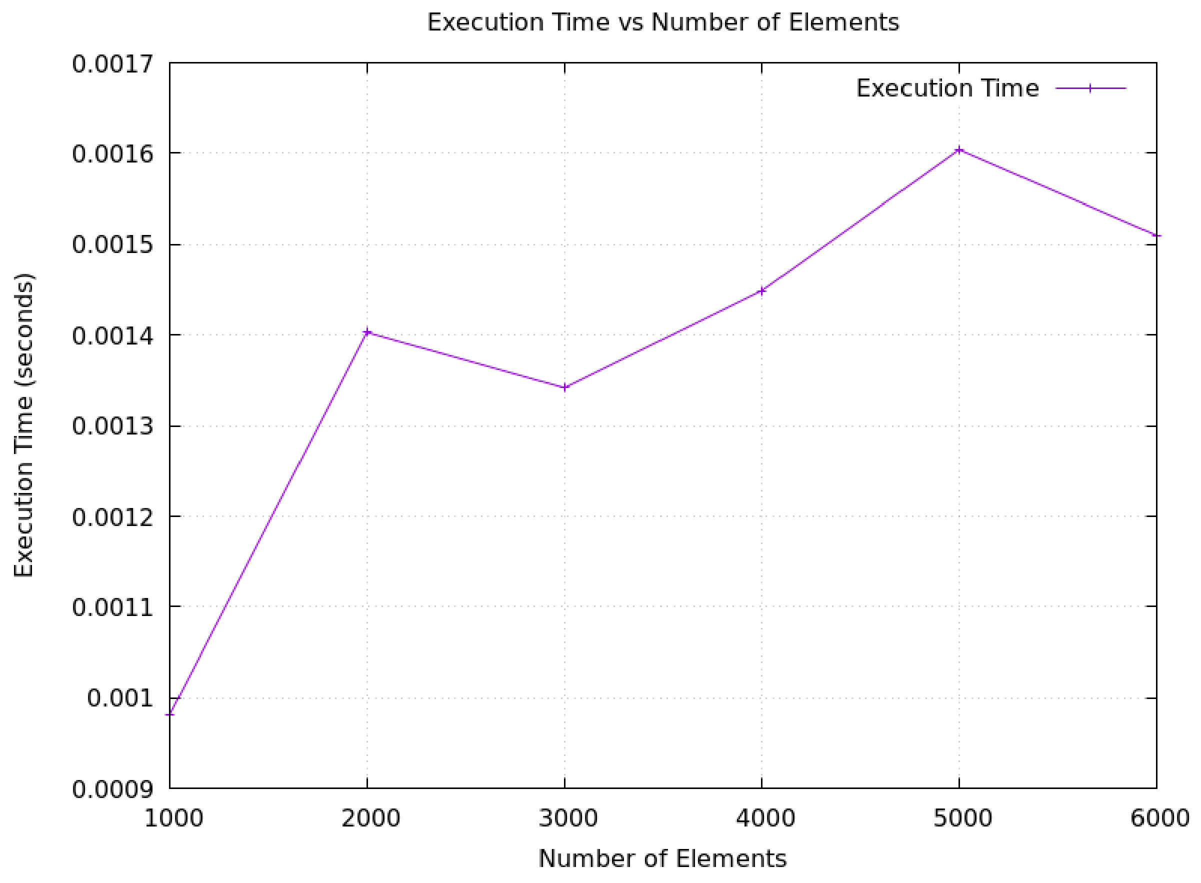
CASE 1: C) No. of processors = 16, No. of elements = 1000, 2000, 3000, 4000, 5000, 6000.

1. **Compile and Run the program:**
    - Run the code across a constant number of processors, say 16. And enter a different number of elements in each run, say 1000, 2000, 3000, 4000, 5000, 6000.

    - The generated exec_times_const_proc.txt would look like this:

```
1000 0.000981
2000 0.001403
3000 0.001342
4000 0.001449
5000 0.001604
6000 0.001510
```

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



- **Observation:** Again the execution times do not show a consistent trend. The times fluctuate without a steady increase or decrease as the number of elements increases.

<u>CASE 2: A) No. of processors = 48, No. of elements = 10, 100, 1000, 10000, 100000.</u>

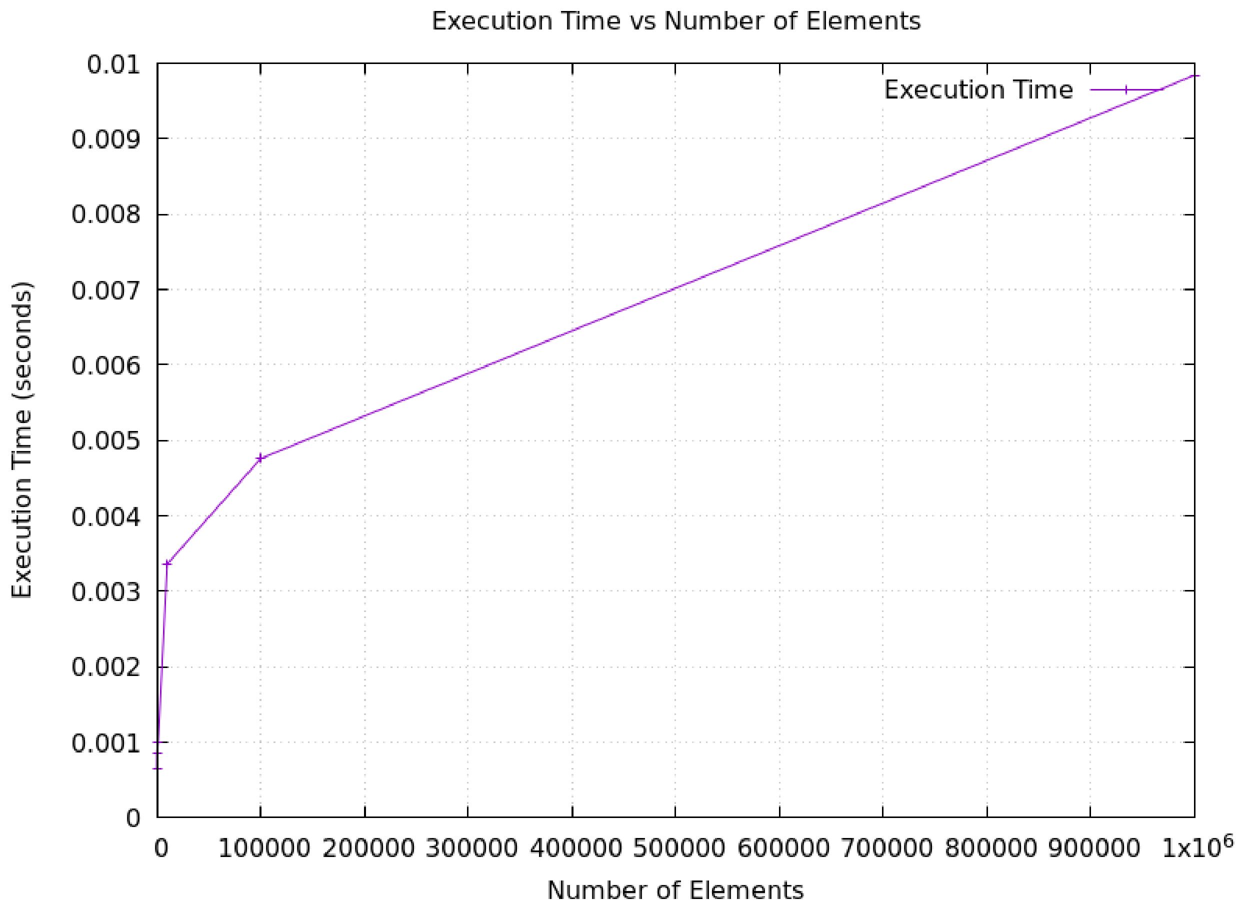
- Run the code across a constant number of processors, say 48. And enter a different number of elements in each run, say 10, 100, 1000, 10000, 100000.

- The generated exec_times_const_proc.txt would look like this:

```
10 0.000655
100 0.000848
1000 0.001001
10000 0.003364
```

```
100000 0.004759
1000000 0.009843
```

- And the corresponding plot would look like this:



- **Observation:** Here, we observe a clear increase in execution time as the number of elements increases, especially when the number of elements becomes significantly large.

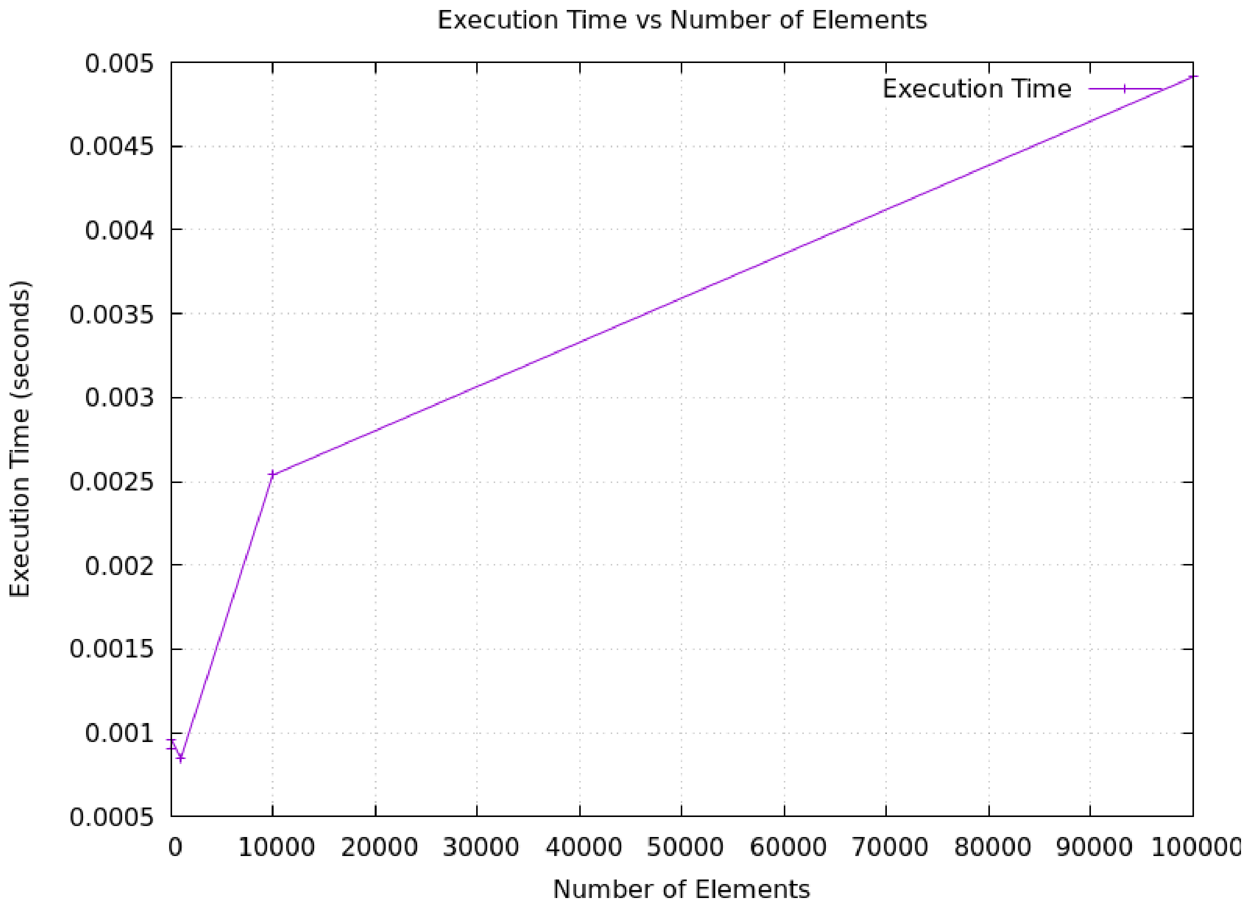
<u>CASE 2: B) No. of processors = 32, No. of elements = 10, 100, 1000, 10000, 100000.</u>

- Run the code across a constant number of processors, say 32. And enter a different number of elements in each run, say 10, 100, 1000, 10000, 100000.

- The generated exec_times_const_proc.txt would look like this:

```
10 0.000908
100 0.000956
1000 0.000850
10000 0.002538
100000 0.004913
```

- And the corresponding plot would look like this:



- **Observation:** Again the execution times do not show a consistent trend. The times fluctuate without a steady increase or decrease as the number of elements increases.

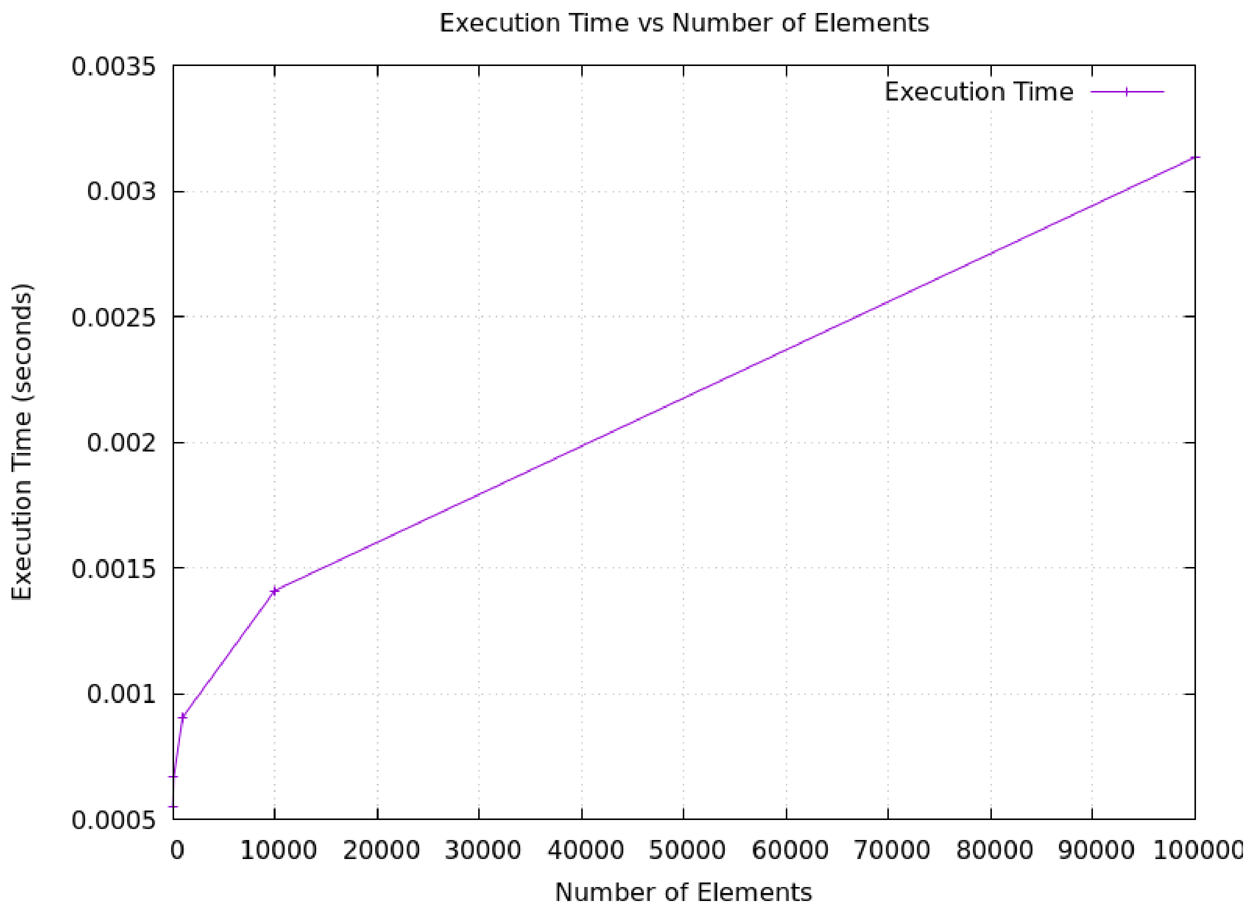
CASE 2: C) No. of processors = 16, No. of elements = 10, 100, 1000, 10000, 100000.

- Run the code across a constant number of processors, say 16. And enter a different number of elements in each run, say 10, 100, 1000, 10000, 100000.

- The generated exec_times_const_proc.txt would look like this:

```
10 0.000551
100 0.000667
1000 0.000908
10000 0.001411
100000 0.003134
```

- And the corresponding plot would look like this:



- **Observation:** Here, we observe a clear increase in execution time as the number of elements increases, especially when the number of elements becomes significantly large.

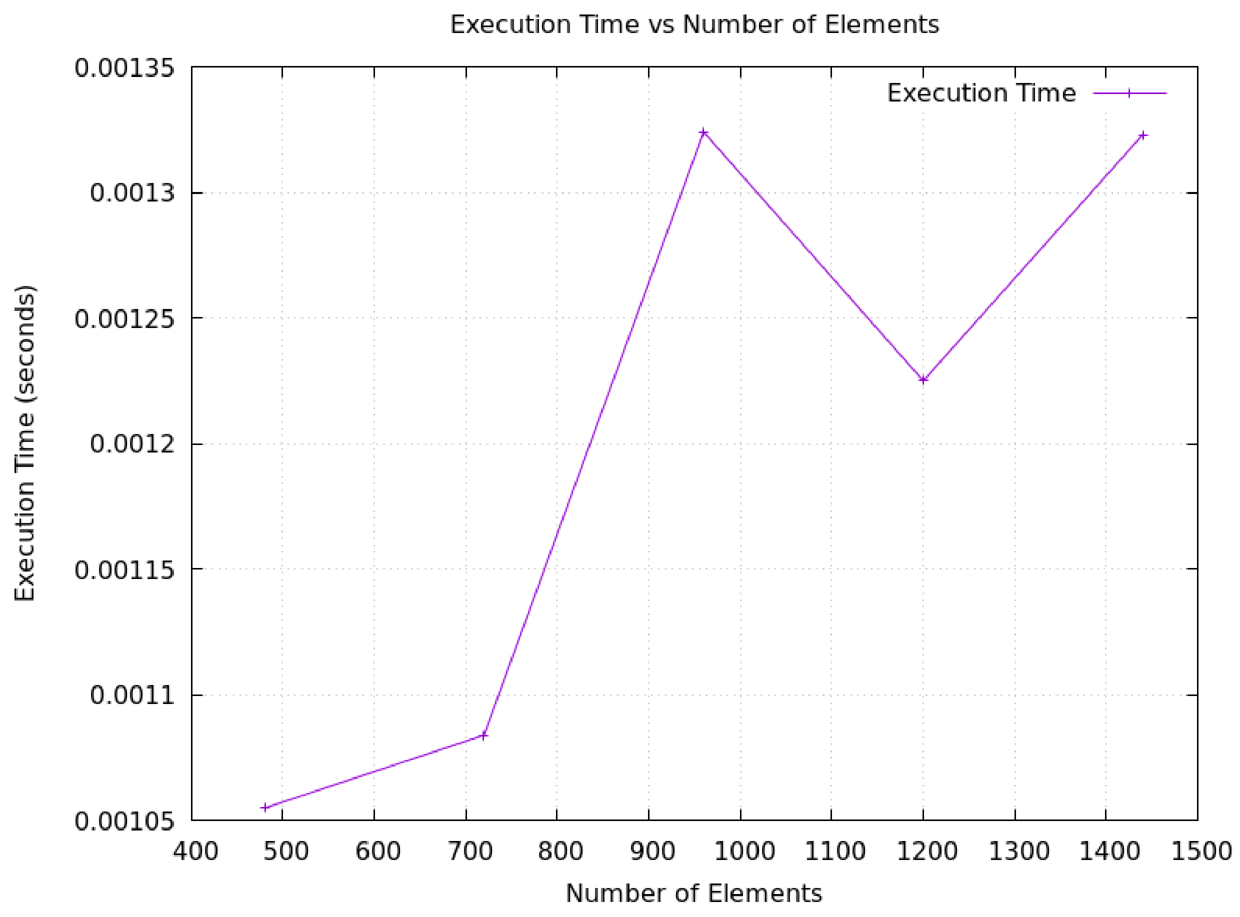
CASE 3: Multiples of 48 Elements with 48 Processors

- Keep the number of processor as 48 and change the number of elements to multiples of 48, i.e., 48x10=480, 48x15=720, 48x20=960, 48x25=1200, 48x30=1440

- The generated exec_times_const_proc.txt would look like this:

```
480 0.001055
720 0.001084
960 0.001324
1200 0.001225
1440 0.001323
```

- And the corresponding plot would look like this:



- **Observation:** In this case also, the execution times do not show a consistent trend. But the execution times in this case are relatively stable and do not show significant fluctuations. Although minor variations exist, they are not substantial.

CASE 4: Larger Multiples of 48 Elements with 48 Processors

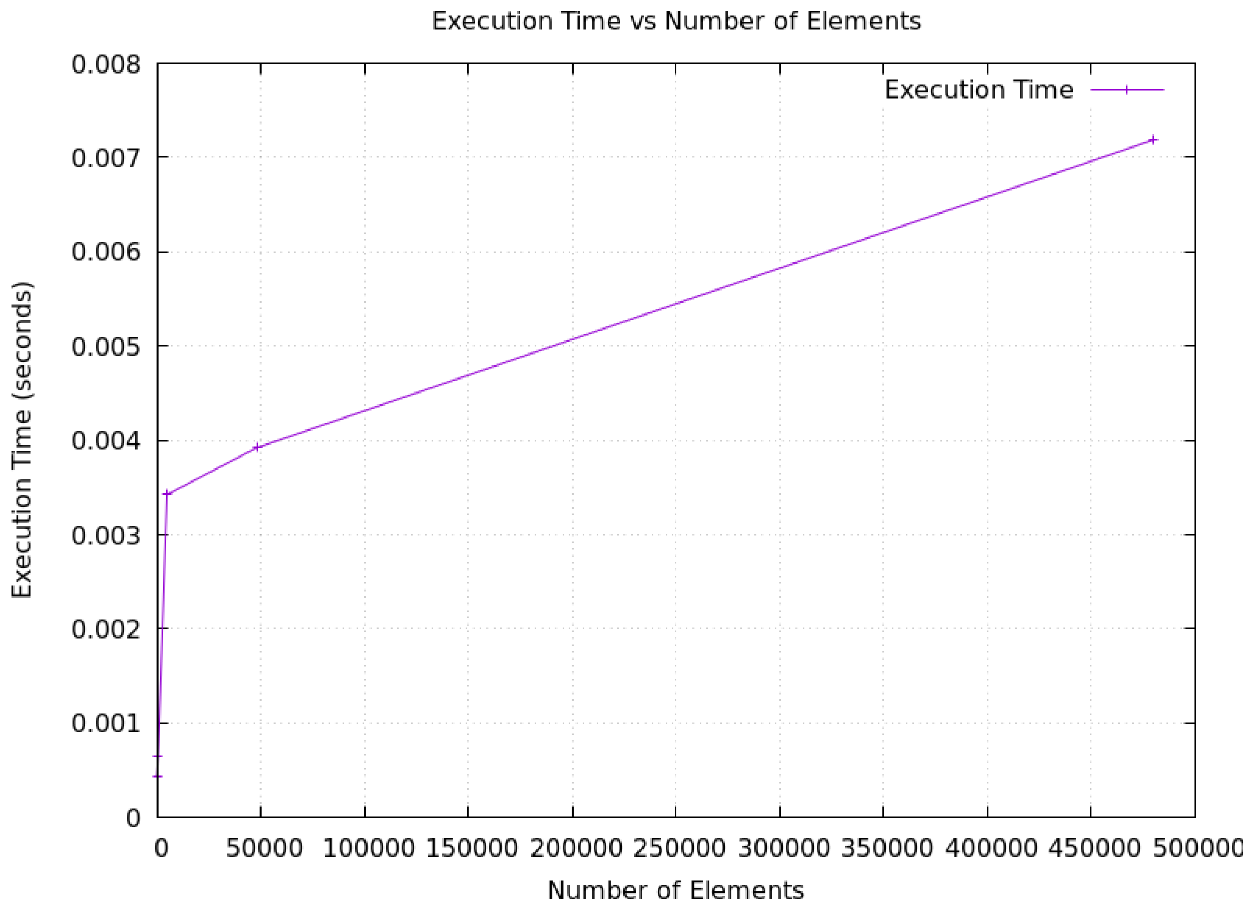
- Keep the number of processor as 48 and change the number of elements to multiples of 48, i.e., 48x1=48, 48x10=480, 48x100=4800, 48x1000=48000, 48x10000=480000

- The generated exec_times_const_proc.txt would look like this:

```
48 0.000442
480 0.000657
4800 0.003425
```

```
48000 0.003922
480000 0.007189
```

- And the corresponding plot would look like this:



- **Observation:** Similar to Case 2, this case shows a steady increase in execution time as the number of elements increases, particularly when the number of elements is significantly larger.

Conclusion:

- **Case 1 and 3:** Execution times did not follow a steady pattern, likely due to smaller variations in the problem size. This suggests that when the workload is relatively small or medium-sized, other factors such as communication overhead might impact the performance.

- **Case 2 and 4:** A clear and steady increase in execution time was observed as the number of elements increased significantly. This demonstrates that as the workload becomes larger, the execution time scales in a more predictable manner. In practical applications, this scalability indicates that larger problem sizes will require proportionately more computation time. But again this might not always be the case, as we have taken the average of values of the execution times, it seems to increase as the number of elements increase. Even in case 2B, we can observe that the execution times did not follow a steady pattern.

- Overall, the experiment shows that execution time increases with problem size, especially when there is a large difference in the number of elements being processed. For smaller or less varied problem sizes, the execution time may not follow a predictable pattern.

Steps to run the program using Slurm Script file on HPC machine:

1. **Create a SLURM Job Script:**
    - On the cluster, create a job script file named test_sum_of_nums.sh. This script will automate the job submission process.

```
vim test_sum_of_nums.sh
```

- Insert the following content into the script file:

```
#!/usr/bin/bash

#SBATCH -N 1               # Specifies number of nodes
#SBATCH --job-name=testSumOfNums   # Name the job as
'testSumOfNums'
#SBATCH --output=job.%J.out_node_4
#SBATCH --error=job.%J.err_node_4
#SBATCH --ntasks-per-node=4 # Request n cores or task per node
#SBATCH --mem-per-cpu=4GB   # Request 4GB RAM per core
#SBATCH --time=06:50:20 # maximum duration of the run
#SBATCH --partition=standard # specifies queue name

module load openmpi3/3.1.4
module list                # will list modules loaded by default. In our case, it
will be GNU8 compilers and OpenMPI3 MPI libraries

pwd                        # prints current working directory
date                       # prints the date and time

mpicc -o mpi sum_of_nums.c
mpirun mpi        # run the MPI job
```

2. **Submit the Job Script:**
   - Use the sbatch command to submit the job script to the SLURM scheduler:

```
sbatch test_sum_of_nums.sh
```

3. **Check Job Status:**
   - Use squeue to check the status of your job. This will show the current jobs in the queue for your user.

```
squeue -u username
```

4. **View Output:**
    - Once the job completes, view the output files to see the results. The output file will be named according to the #SBATCH --output directive, with %J replaced by the job ID.