# INDIAN INSTITUTE OF TECHNOLOGY GOA

## NSM Nodal Center for Training in HPC and AI

# Tutorial: MPI "MATRIX MULTIPLICATION"

**Authors:**

1. **POBBATI KOUSHIKI SAI MEGHANA**, **Project Associate**
2. **Prof. SHARAD SINHA**

**CONTENTS:**

# MPI "Matrix Multiplication" Program on Param Vidya HPC

## Objective:

This tutorial will show you how to create, compile, and run MPI "Matrix Multiplication" programs on the Param Vidya HPC cluster. The goal is to compute the multiplication of two square matrices using MPI (Message Passing Interface) on HPC, and calculate the execution time of the program and understand the power of parallel computing. We write program for matrix multiplication using MPI_Send and MPI_Recv

## MPI “Matrix multiplication” using MPI_Send and MPI_Recv:

- **Program Explanation:**
  The program utilizes MPI_Send and MPI_Recv to distribute the workload of multiplying two matrices to multiple processes. These processes are of two types namely the root process and the slave processes. The root process distributes the input matrices and slave processes perform the multiplication of the partial matrices assigned to them.The results of all the slave processes are then collected by the root process and the root processes displays the output.

- **Matrix multiplication code using MPI_Send and MPI_Recv:**

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

#define OUTPUT_FILE "timing_data.txt"

MPI_Status status;
int main(int argc, char **argv)
{
  int rank, num_procs; //process id, number of processes
  int num_slaves; //number of processes excluding root process
  int source, dest; //message passing source and destination
  int num_rows, offset; //num of rows given to each process, starting row index

  double start_time1, end_time1, t1; //t1 : calculation time of the root
process
  double start_time2, end_time2, t2; //t2 : calculation time of the each
process
```

```
double start_time, end_time, t; //t : execution time of the program

int N = atoi(argv[1]); //matrix size passed as an argument in command line


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

double tarr [num_procs]; //array of calculation times of all processes
int idx = 0;

// Ensure there are enough processors
if (rank == 0 && num_procs < 2) {
fprintf(stderr, "At least 2 processors are required\n");
MPI_Abort(MPI_COMM_WORLD, 1);
}

num_slaves = num_procs - 1;

// Allocate matrices dynamically
int (*A)[N] = malloc(N * N * sizeof(int));
int (*B)[N] = malloc(N * N * sizeof(int));
int (*C)[N] = malloc(N * N * sizeof(int)); //output matrix C = A X B

// Root process
if (rank == 0)
{
  srand(time(NULL)); //to get different inputs on different runs

  // Fill the input matrix A
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      A[i][j] = 1 + rand() % 9;
    }
  }
  // Fill the input matrix B
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      B[i][j] = 1 + rand() % 9;
```

```
      }
    }

    // Print the input matrix A
    printf("\nMatrix A\n\n");
    for (int i = 0; i < N; i++)
    {
      for (int j = 0; j < N; j++)
      {
        printf("%d  ", A[i][j]); //comment this statement to save screen space
      }
      printf("\n");
    }
    // Print the input matrix B
    printf("\nMatrix B\n\n");
    for (int i = 0; i < N; i++)
    {
      for (int j = 0; j < N; j++)
      {
        printf("%d  ", B[i][j]); //comment this statement to save screen space
      }
      printf("\n");
    }

    start_time = MPI_Wtime();
    // Calculate the total number of rows each process should receive
    num_rows = N / num_procs; //each process gets an equal share of total rows
    offset = 0; //distribution starts from row 0

    // Allotting calculation tasks to slaves
    printf("\n");
    for (dest = 1; dest <= num_slaves; dest++)
    {
      MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
      MPI_Send(&num_rows, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
      MPI_Send(&A[offset][0], num_rows * N, MPI_INT, dest, 1, MPI_COMM_WORLD);
      printf("Rows from %d to %d are allotted to process %d\n", offset,
offset+num_rows-1,
      dest);
      MPI_Send(&B[0][0], N * N, MPI_INT, dest, 1, MPI_COMM_WORLD);
      offset += num_rows;
    }

    printf("Rows from %d to %d are allotted to process 0.\n", offset,
```

```
offset+num_rows-1);

    // Receiving task outputs
    printf("\n");
    for (int i = 1; i <= num_slaves; i++)
    {
      source = i;
      MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
      MPI_Recv(&num_rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
      MPI_Recv(&C[offset][0], num_rows * N, MPI_INT, source, 2, MPI_COMM_WORLD,
&status);
      MPI_Recv(&t2, 1, MPI_DOUBLE, source, 2, MPI_COMM_WORLD, &status);
      tarr[idx] = t2;
      idx++;
      printf("Process %d sent task outputs %d to %d\n", source, C[offset][0],
      C[offset][num_rows*N - 1]);
    }


    //calculating its own part of output
    start_time1 = MPI_Wtime();
    for (int k = 0; k < N; k++)
    {
      for(int i=offset+1; i<N;i++)
      {
        C[i][k] = 0;
        for (int j = 0; j < N; j++)
        {
          C[i][k] += A[i][j] * B[j][k];
        }
      }
    }
    end_time1 = MPI_Wtime();
    t1 = end_time1 - start_time1;
    tarr[idx] = t1;

    //finding the max of the calculation times
    double max = tarr[0];
    for(int z = 1; z < num_procs; z++)
    {
      if(tarr[z] >= max) max = tarr[z];
    }

    end_time = MPI_Wtime();
```

```
    t = end_time - start_time;
    //Printing final output
    printf("\nResult Matrix C = Matrix A * Matrix B:\n\n");
    for (int i = 0; i < N; i++)
    {
      for (int j = 0; j < N; j++)
        printf("%d\t", C[i][j]);
      printf("\n");
    }
    printf("\n");

    //Copy the timing data into a file
    FILE *fp = fopen(OUTPUT_FILE, "a");
    if(fp == NULL)
    {
      fprintf(stderr, "Failed to open file for writing\n");
      MPI_Abort(MPI_COMM_WORLD, 1);
    }
    fprintf(fp, "np = %d,   n = %d,    et = %f, ct = %f\n", num_procs, N, t,
max);
    fclose(fp);

    free(A);
    free(B);
    free(C);
  }

  // Slave processes
  else
  {
    // Receiving tasks from master
    source = 0;
    MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&num_rows, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&A[0][0], num_rows * N, MPI_INT, source, 1, MPI_COMM_WORLD,
&status);
    MPI_Recv(&B[0][0], N * N, MPI_INT, source, 1, MPI_COMM_WORLD, &status);

    start_time2 = MPI_Wtime();
    // Calculating task outputs
    for (int k = 0; k < N; k++)
    {
      for (int i = 0; i < num_rows; i++)
      {
```

```
        C[offset + i][k] = 0;
        for (int j = 0; j < N; j++)
        {
          C[offset + i][k] += A[i][j] * B[j][k];
        }
      }
    }
    end_time2 = MPI_Wtime();
    t2 = end_time2 - start_time2;

    // Sending task outputs to root process
    MPI_Send(&offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&num_rows, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&C[offset][0], num_rows * N, MPI_INT, 0, 2, MPI_COMM_WORLD);

    //sending calculation time of each process to the root process.
    MPI_Send(&t2, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

- **Code Explanation:**

1. **Initialization:**
    - MPI_Init() is the function used to initialize the MPI environment.
    - MPI_Comm_rank is the function that determines the rank of the calling process in the MPI_COMM_WORLD.
    - MPI_Comm_size is the function that determines the number of processes in the MPI_COMM_WORLD.
    - MPI_COMM_WORLD is a universe for a group of communicating processes.
2. **Memory allocation:**
    - Space for input matrices A and B and output matrix C is dynamically allocated.
3. **Tasks of root process :**
    - Filling and printing the input matrices A and B.
    - Finding the number of rows of A that each slave process should get.

- MPI_Send() is used to send the slave processes the address of chunk of matrix A and address of matrix B.
- MPI_Recv() is used to receive the partial matrix multiplication results.
- Performing its own part of matrix multiplication.
- Printing out the final output and execution time.

4. **Tasks of slave process :**
   - Receive the allotted chunks of matrix A using MPI_Recv() function.
   - Performing matrix multiplication on the received partial A and matrix B.
   - Sending the results to the root process using MPI_Send().
5. **Memory deallocation :**
   - Deallocating the memory allocated for matrices using free() function in C.
6. **MPI_Finalize() :**
   - MPI_Finalize() is the function used to terminate the MPI communication.

## 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 matMul matrix_multiplication.c
   ```

3. **Run the Program:**
   -
   ```
   mpirun -np <number_of_processes> matMul <matrix_size>
   ```

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

   ```
   $ mpirun -np 10 matMul
   --------------------------------------------------------------------------
   There are not enough slots available in the system to satisfy the 9
   slots that were requested by the application:
   ```

```
matMul

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 matrix_multiplication.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_multiplication.c <your
username>@paramvidya.iitgoa.ac.in:<path to directory on HPC where
to save the data>
```

    - Otherwise, you can also create a matrix_multiplication.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_multiplication.c
```

5. **Run the Program:**
   - 

```
mpirun -np <number_of_processes> matMul <matrix_size>
```

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

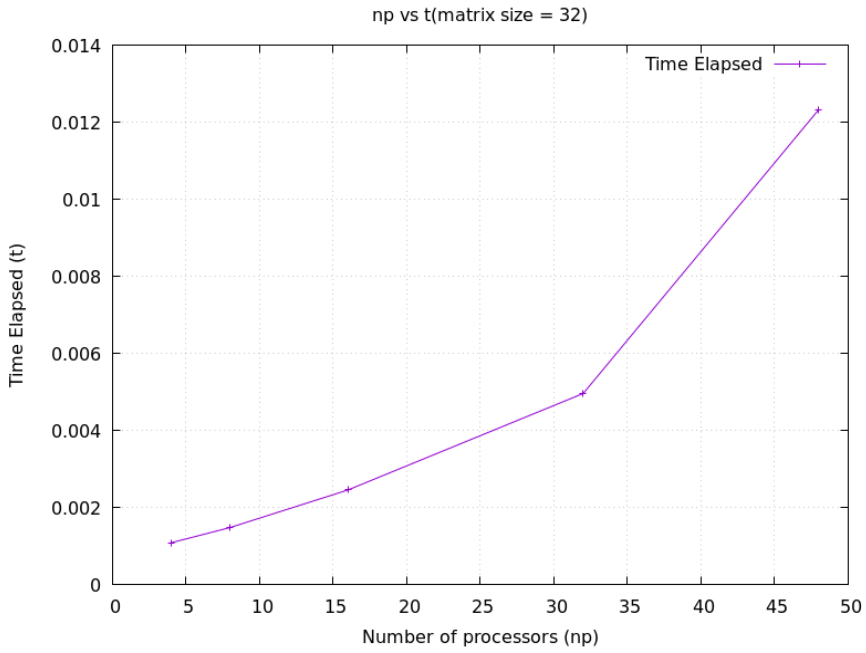
```
mpirun -np 16 matMul 128
```

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

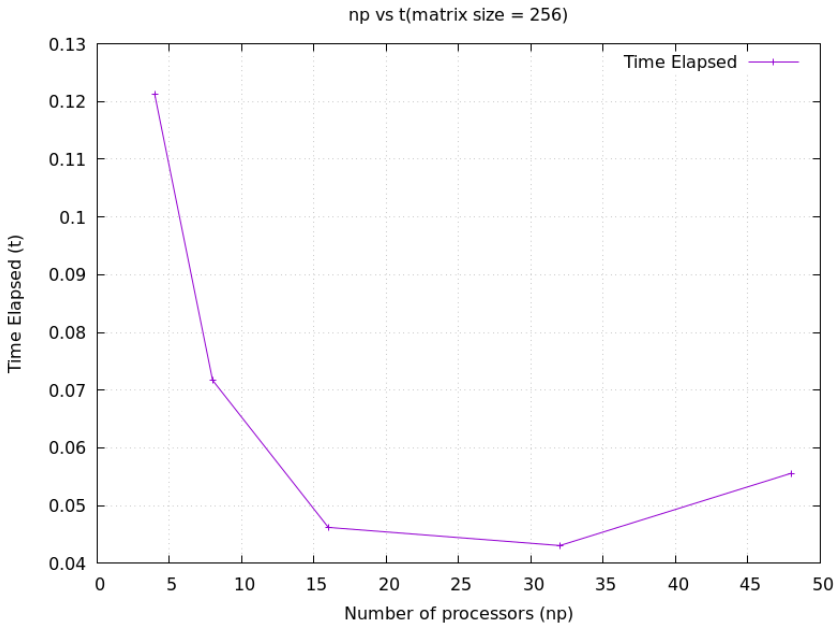
## Experimentation, Plotting and Observations of execution time :

### a. Keeping the size of the matrix(n), plotting number of processors(np) vs time(t)

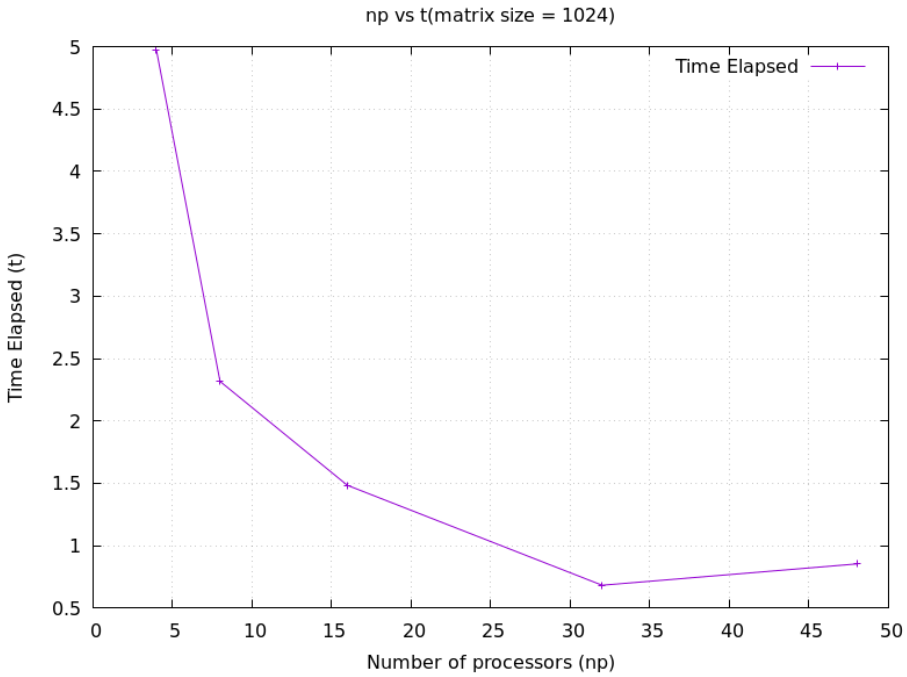
- **Case 1 : matrix size = 32**



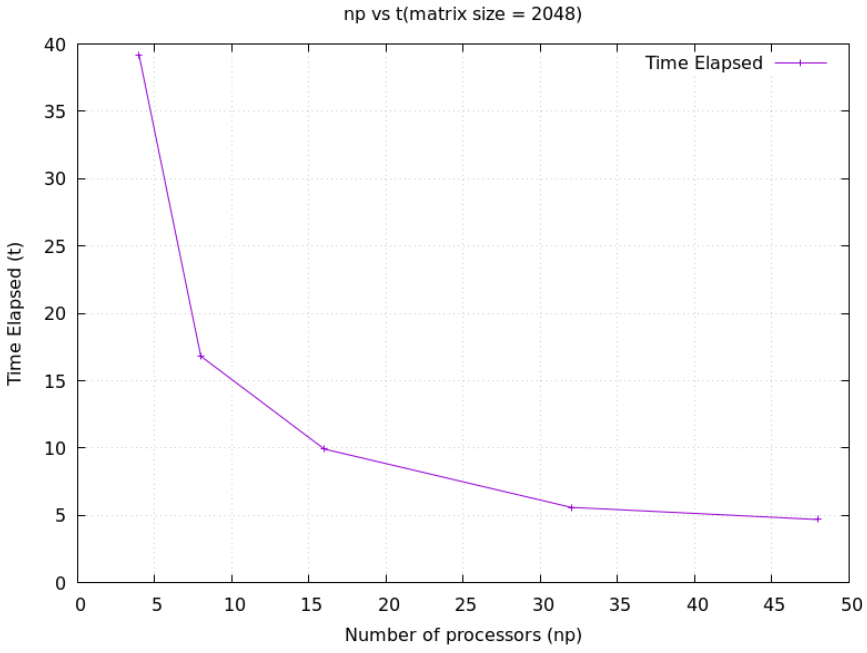

- **Case 2 : matrix size = 256**



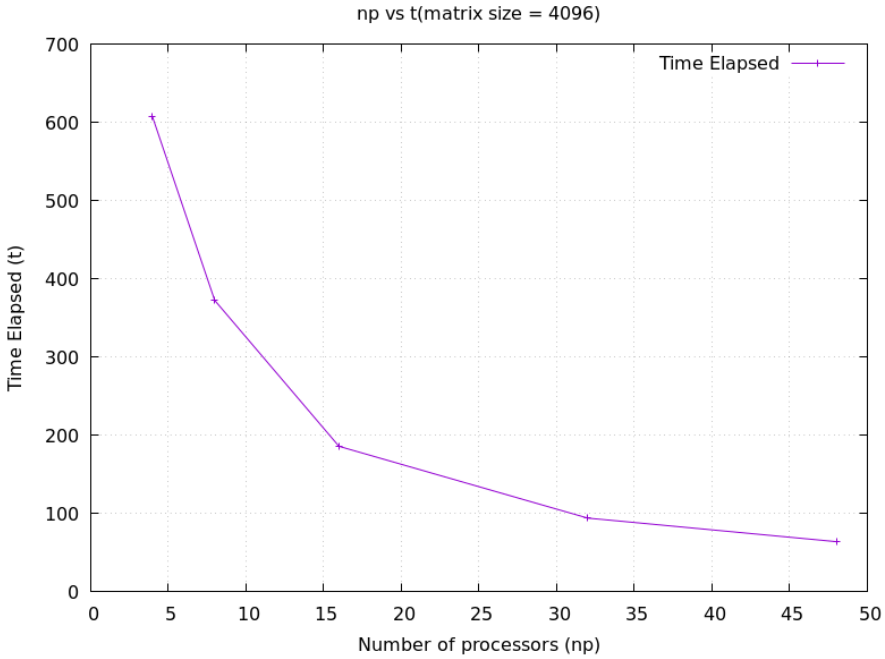
- **Case 3 : matrix size = 1024**



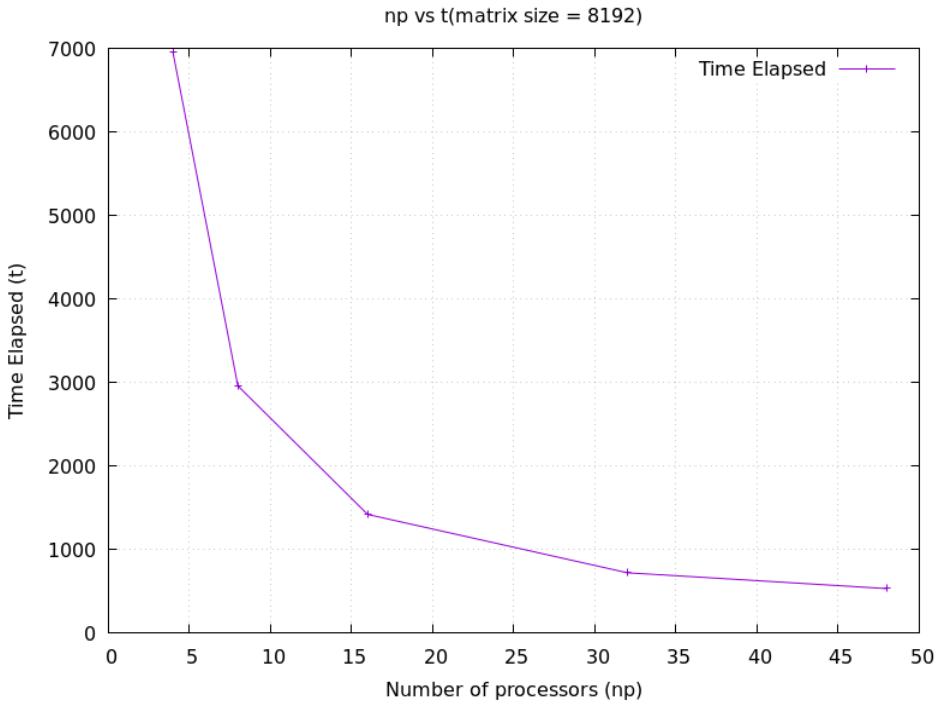

- **Case 4 : matrix size = 2048**




- **Case 5 : matrix size = 4096**




- **Case 6 : matrix size = 8192**

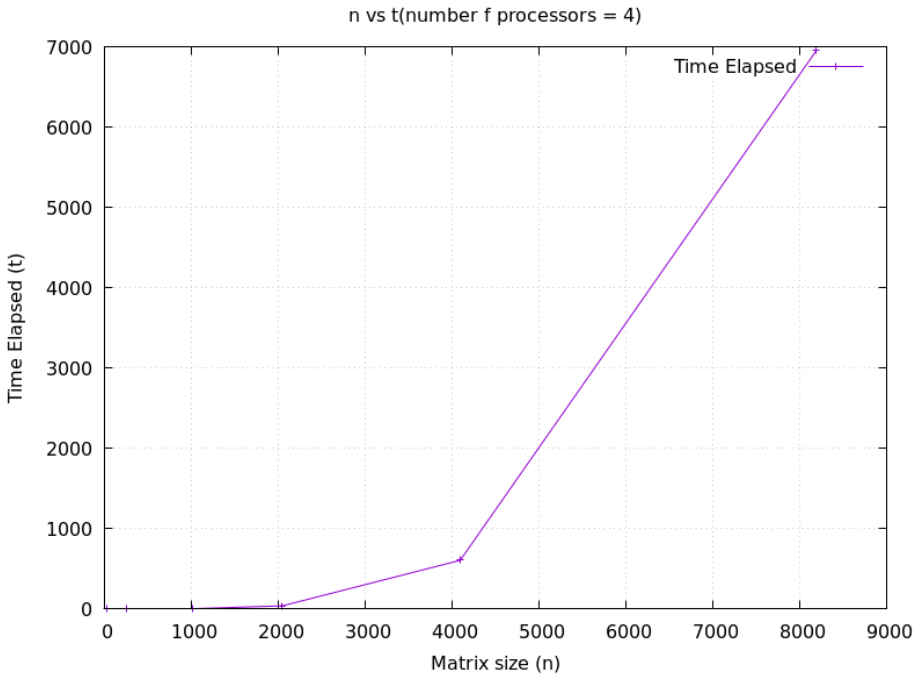



**Observation :** When the number of processes increases, the execution time in parallel computation must decrease. But in some cases, like case 1,2 and 3, the trend was not

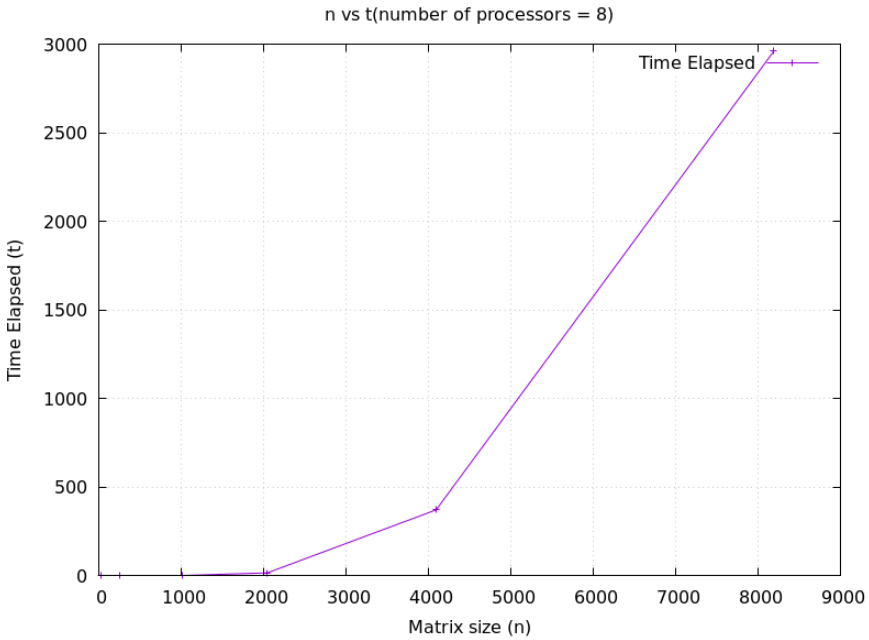
as expected. This is because of the communication delay between the processes, which is an overhead.

**b. Keeping the number of processes(np) constant and plotting matrix size(n) vs time(t) :**

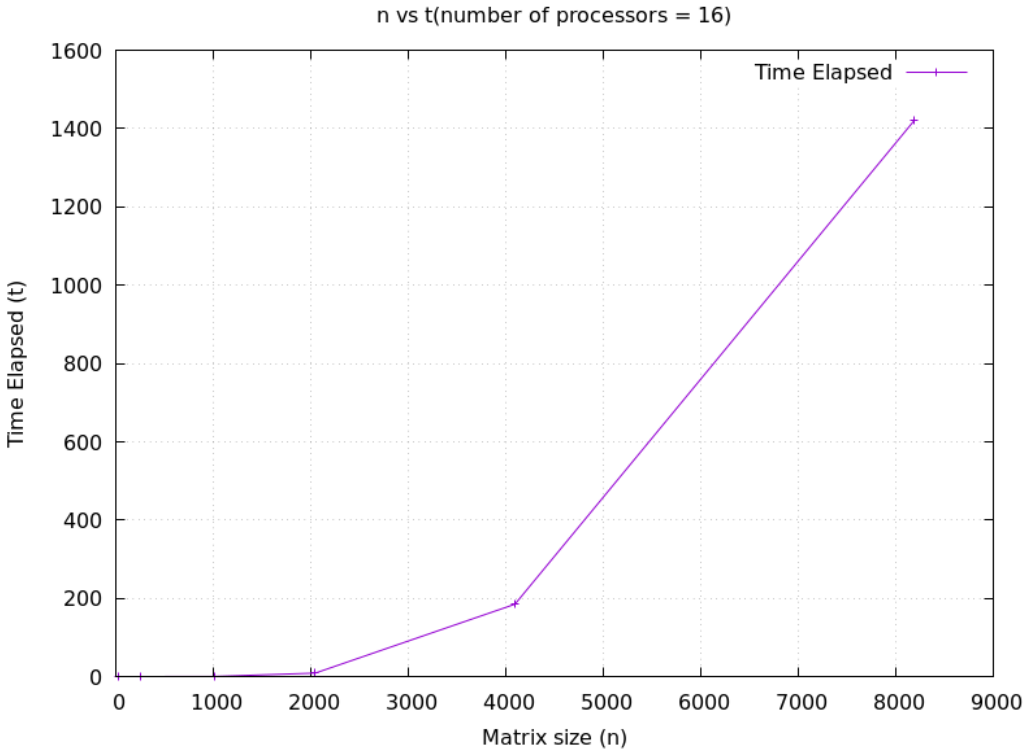
- **Case 1 : number of processes = 4**



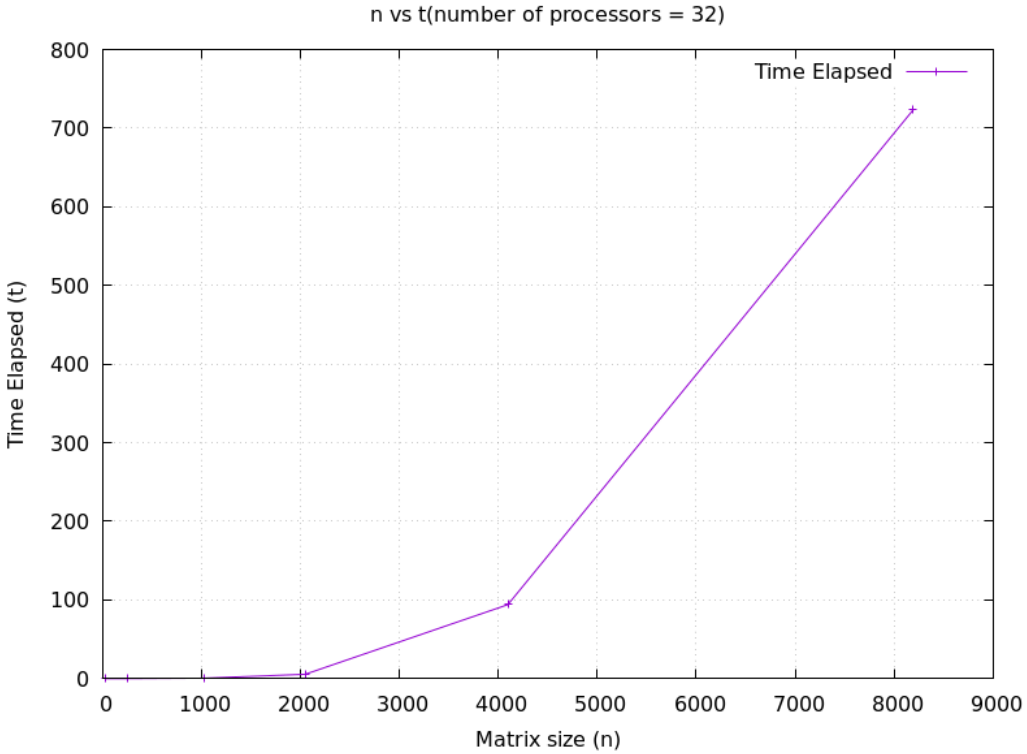

- **Case 2 : number of processes = 8**



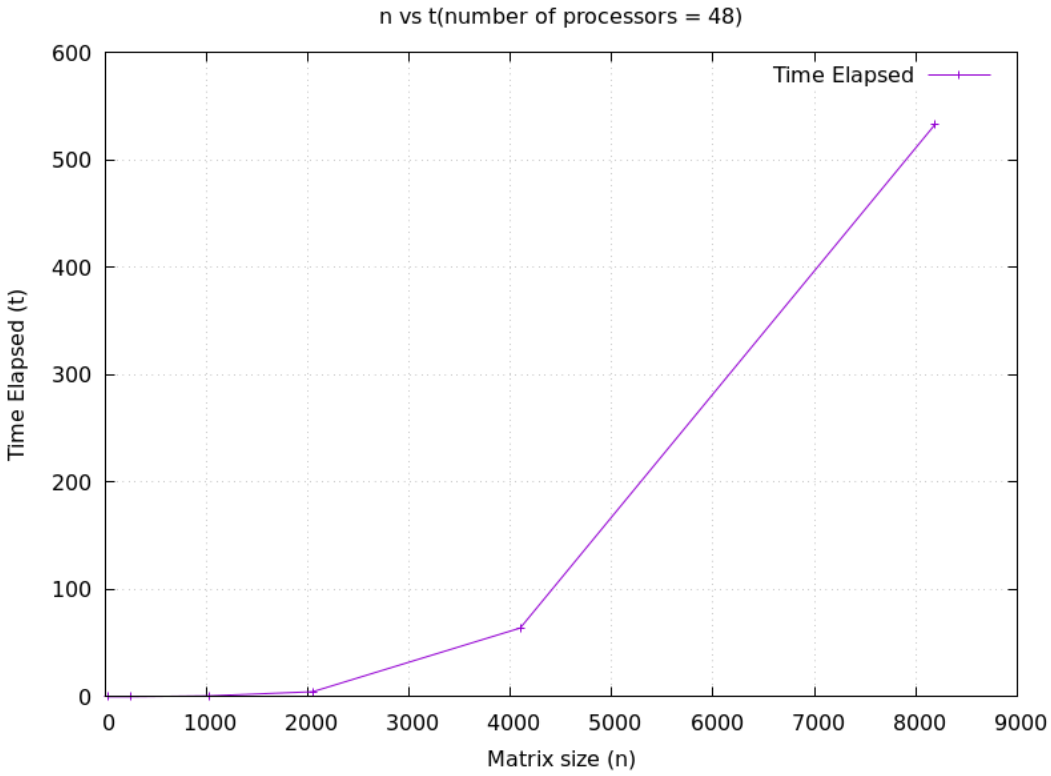
- **Case 3 : number of processes = 16**




- **Case 4 : number of processes = 32**



- **Case 5 : number of processes = 48**

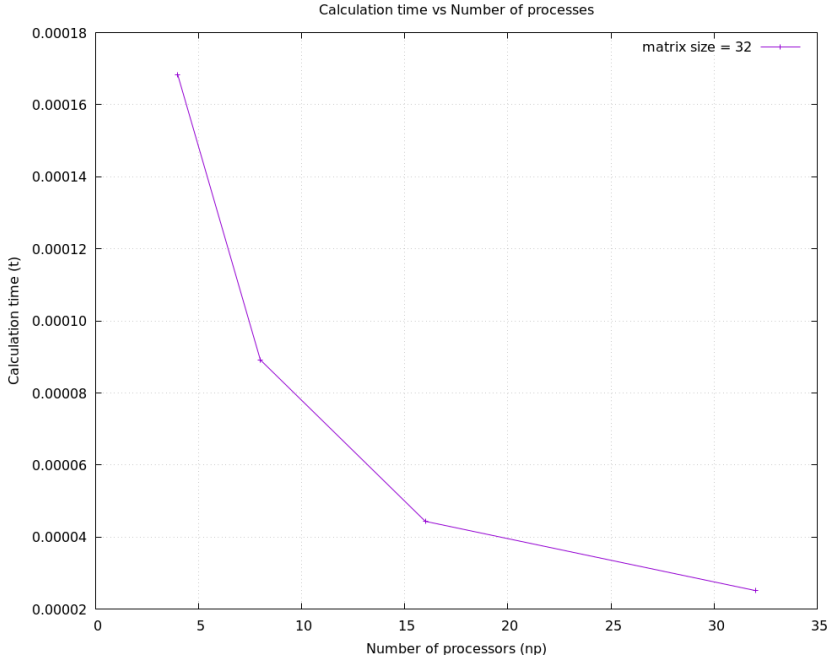



**Observation :** As the size of the matrix increases, the execution time increases in all cases.

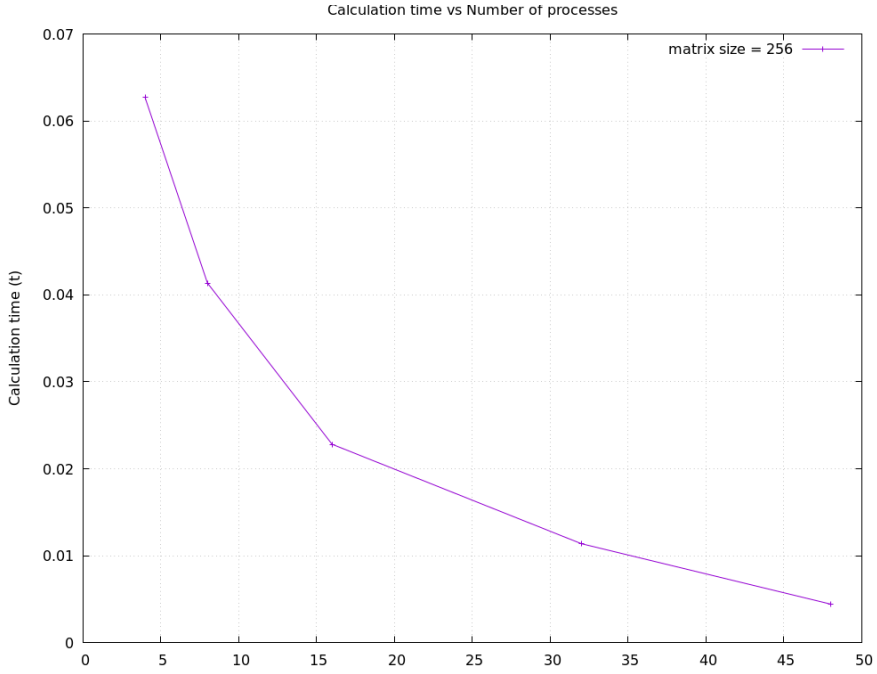
**Experimentation, Plotting and Observations of calculation time :**

a. **Keeping the matrix size(n) constant and plotting number of processes(np) vs calculation time(t) :** We have observed in the plots of np vs execution time that the communication delay was an overhead and the expected trend was not seen. Now let us plot only the calculation time ignoring the communication delays.

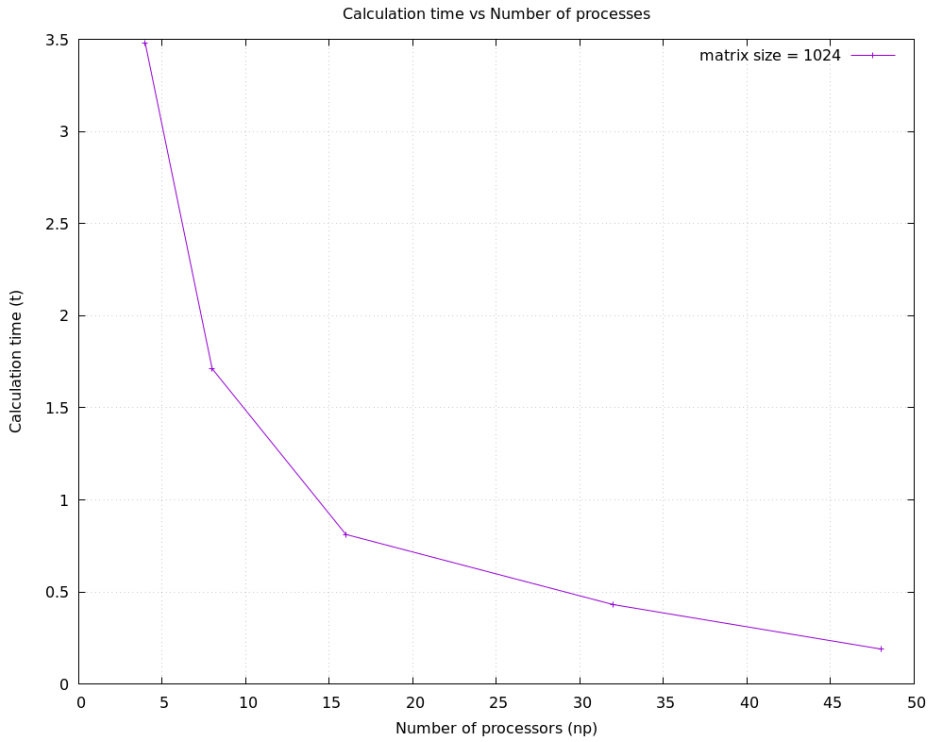
- **Case 1 : matrix size = 32**



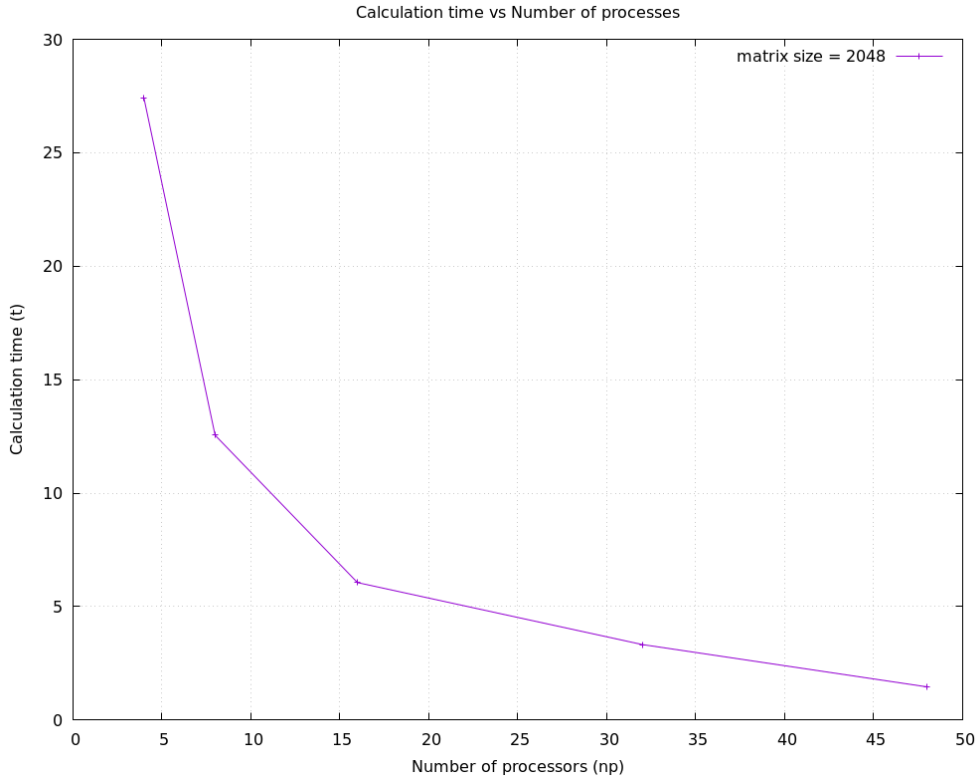

- **Case 2 : matrix size = 256**



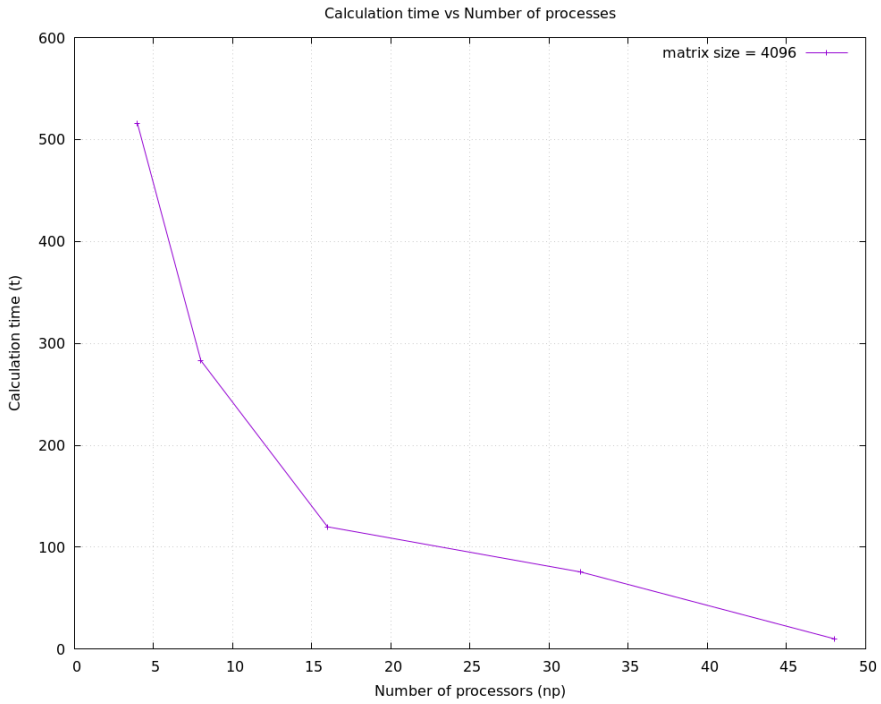

- **Case 3 : matrix size = 1024**



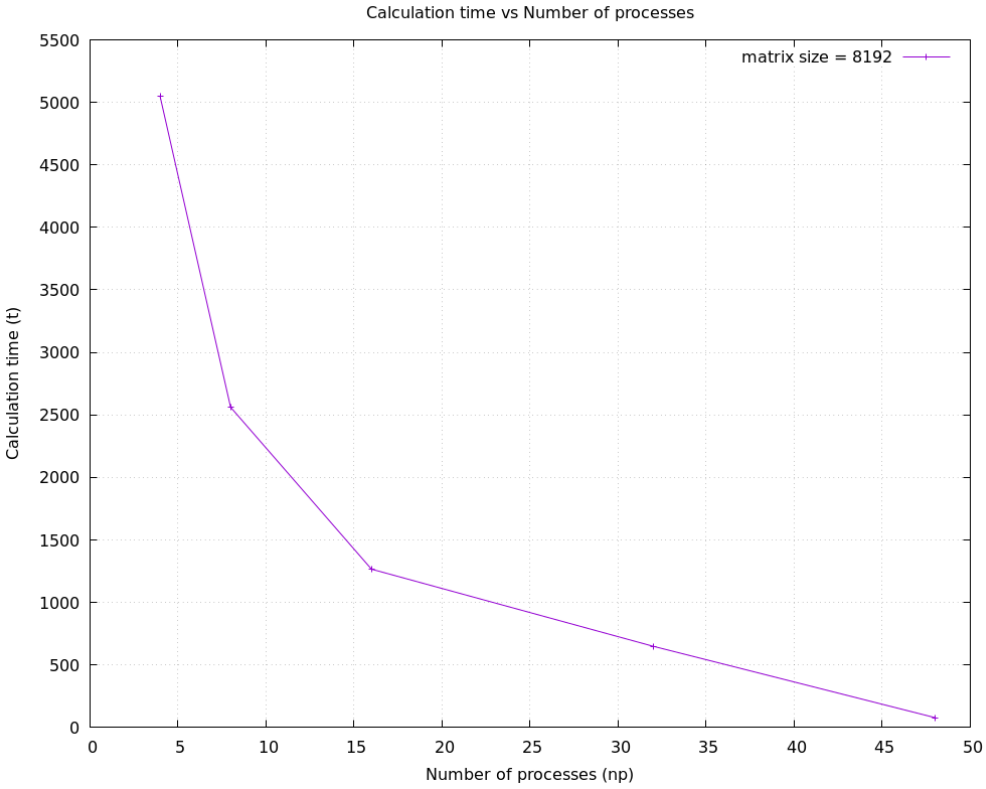

- **Case 4 : matrix size = 2048**



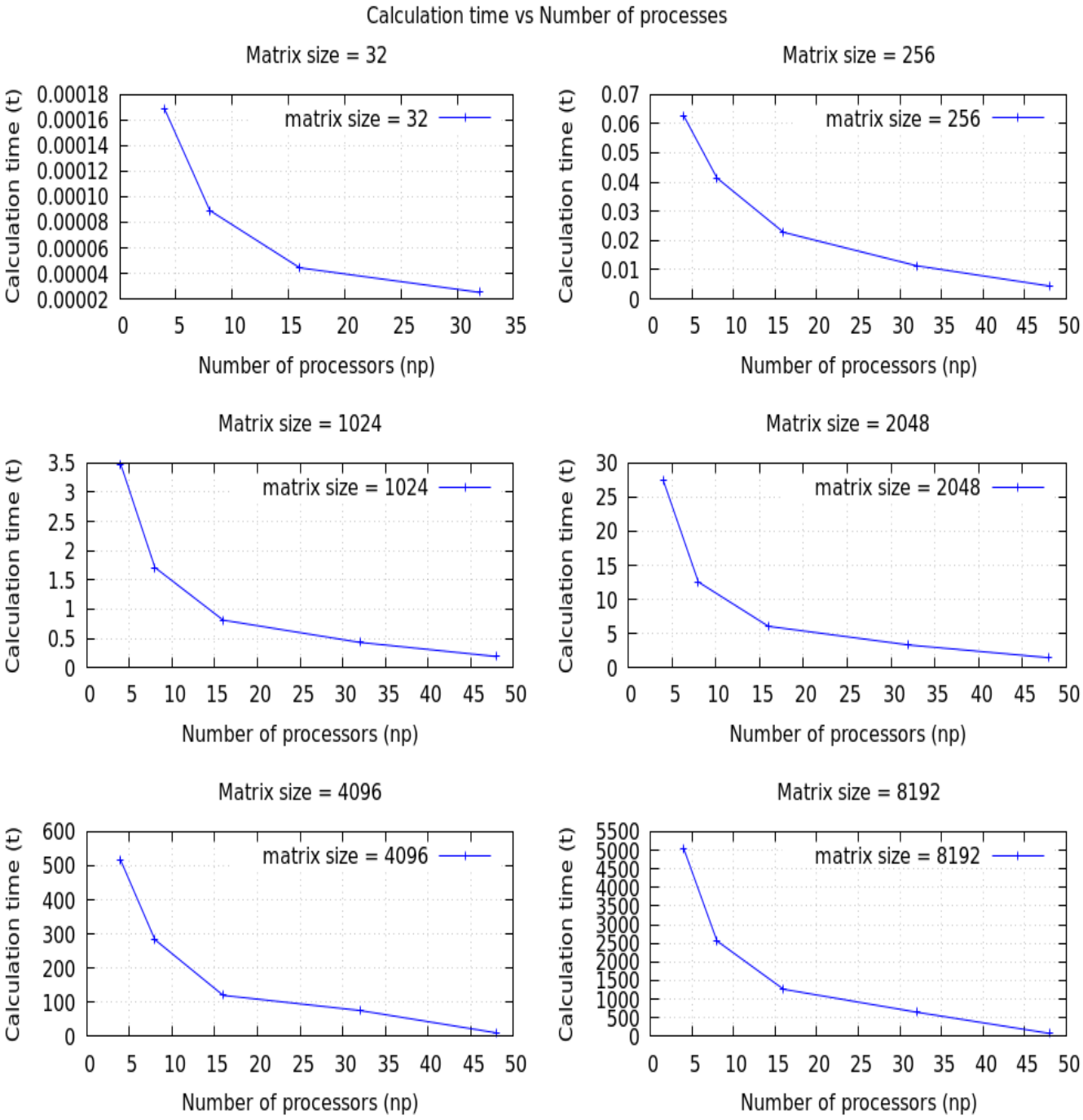

- **Case 5 : matrix size = 4096**




- **Case 6 : matrix size = 8192**



**Observations :** As expected, when the number of processes increased, the calculation time decreased in all the cases.



**b. Keeping the number of processes (np) constant and plotting the matrix size (n) vs calculation time(t) :**