Class: Final Year (Computer Science and Engineering)

Year: 2022-23 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 8

PRN No : 2019BTECS00110

Name : Pavan Shinde

Problem Statement 1:

Study and implement 2D Convolution using MPI. Use a different number of processes and analyze the performance.

Code :

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

#define MAXN 512

int rank, nprocs;
int i, j;
int chunk;
MPI_Datatype Complextype;

typedef struct { float r; float i; } complex;
static complex ctmp;

complex data1[MAXN][MAXN];
complex data2[MAXN][MAXN];
complex data3[MAXN][MAXN];



#define C_SWAP(a,b) {ctmp=(a);(a)=(b);(b)=ctmp;}

void fft1d(complex *r, int n, int isign)
```

```
{
   int     m, i, i1, j, k, i2, l, l1, l2;
   float   c1, c2, z;
   complex t, u;

   if (isign == 0) return;

   /* Do the bit reversal */
   i2 = n >> 1;
   j = 0;
   for (i = 0; i<n - 1; i++) {
       if (i < j)
           C_SWAP(r[i], r[j]);
       k = i2;
       while (k <= j) {
           j -= k;
           k >>= 1;
       }
       j += k;
   }

   /* m = (int) log2((double)n); */
   for (i = n, m = 0; i>1; m++, i /= 2);

   /* Compute the FFT */
   c1 = -1.0;
   c2 = 0.0;
   l2 = 1;
   for (l = 0; l<m; l++) {
       l1 = l2;
       l2 <<= 1;
       u.r = 1.0;
       u.i = 0.0;
       for (j = 0; j<l1; j++) {
           for (i = j; i<n; i += l2) {
               i1 = i + l1;

               /* t = u * r[i1] */
               t.r = u.r * r[i1].r - u.i * r[i1].i;
               t.i = u.r * r[i1].i + u.i * r[i1].r;

               /* r[i1] = r[i] - t */
               r[i1].r = r[i].r - t.r;
```

```
                r[i1].i = r[i].i - t.i;

                /* r[i] = r[i] + t */
                r[i].r += t.r;
                r[i].i += t.i;
            }
            z = u.r * c1 - u.i * c2;

            u.i = u.r * c2 + u.i * c1;
            u.r = z;
        }
        c2 = sqrt((1.0 - c1) / 2.0);
        if (isign == -1) /* FWD FFT */
            c2 = -c2;
        c1 = sqrt((1.0 + c1) / 2.0);
    }

    /* Scaling for inverse transform */
    if (isign == 1) {       /* IFFT*/
        for (i = 0; i<n; i++) {
            r[i].r /= n;
            r[i].i /= n;
        }
    }
}

/*Read input files*/
void file_read()
{
    FILE *f1, *f2; /*open file descriptor */
    f1 = fopen("1_im1", "r");
    f2 = fopen("1_im2", "r");
    for (i = 0; i<MAXN; i++)
    {
        for (j = 0; j<MAXN; j++)
        {
            //printf("%e", data1[i][j].r);
            fscanf(f1, "%e", &data1[i][j].r);
            fscanf(f2, "%e", &data2[i][j].r);
        }
    }
    fclose(f1);
    fclose(f2);
```

```
}

/*Write Output File*/
void file_write()
{
   FILE *f1; /*open file descriptor */
   f1 = fopen("output", "w+");
   for (i = 0; i<MAXN; i++)
   {
       for (j = 0; j<MAXN; j++)
       {
           fprintf(f1, "%1.6e  ", data3[i][j].r);
       }
       fprintf(f1, "\n");
   }
   fclose(f1);
}

int main(int argc, char* argv[])
{
   double etstart, etstop;  /* Elapsed times using MPI */
   double t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12;
   int p;

   file_read();

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

   chunk = MAXN / nprocs;

   if (rank == 0)
       t1 = MPI_Wtime();

   /*Distribution of data1 and data2*/
   if (rank == 0)
   {
       for (p = 1; p < nprocs; p++)
       {
           MPI_Send(&data1[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD);
```

```
            MPI_Send(&data2[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD);
        }
    }
    else
    {
        MPI_Recv(&data1[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&data2[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
        t2 = MPI_Wtime();

    /*1D FFT Row wise*/
    for (i = 0; i<chunk; i++)
    {
        fft1d(data1[i], MAXN, -1);
        fft1d(data2[i], MAXN, -1);
    }

    if (rank == 0)
        t3 = MPI_Wtime();

    /*Collecting data1 & data2*/
    if (rank == 0)
    {
        for (p = 1; p < nprocs; p++)
        {
            MPI_Recv(&data1[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(&data2[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    else
    {
        MPI_Send(&data1[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD);
```

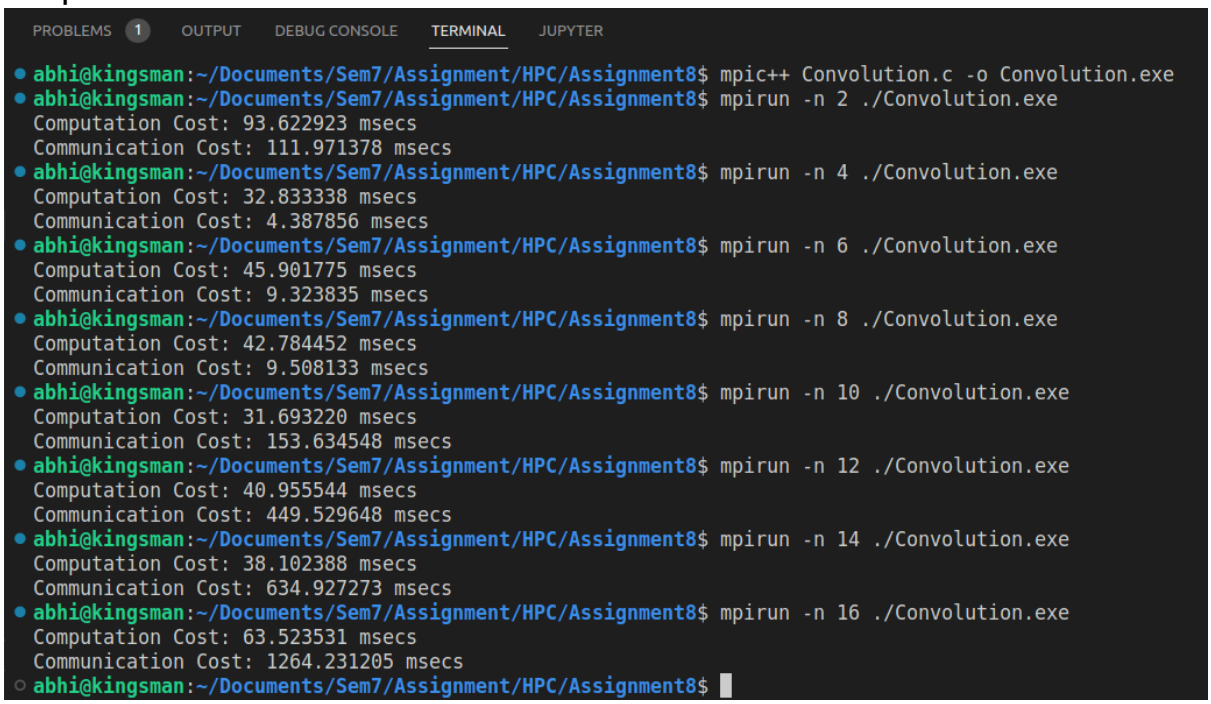
```
      MPI_Send(&data2[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD);
   }

   if (rank == 0)
       t4 = MPI_Wtime();

   /*Transpose*/
   for (j = 0; j < MAXN; j++)
   {
       for (i = j; i < MAXN; i++)
       {
           C_SWAP(data1[j][i], data1[i][j]);
           C_SWAP(data2[j][i], data2[i][j]);
       }
   }

   if (rank == 0)
       t5 = MPI_Wtime();

   /*Again Distribution of data1 & data2*/
   if (rank == 0)
   {
       for (p = 1; p < nprocs; p++)
       {
           MPI_Send(&data1[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD);
           MPI_Send(&data2[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD);
       }
   }
   else
   {
       MPI_Recv(&data1[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
       MPI_Recv(&data2[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }

   if (rank == 0)
       t6 = MPI_Wtime();
```

```
   /*1D FFT Row wise (Transposed Column)*/
   for (i = 0; i<chunk; i++)
   {
       fft1d(data1[i], MAXN, -1);
       fft1d(data2[i], MAXN, -1);
   }

   /*Point Multiplication*/
   for (i = 0; i < chunk; i++)
   {
       for (j = 0; j < MAXN; j++)
       {
           data3[i][j].r = data1[i][j].r * data2[i][j].r - data1[i]
[j].i * data2[i][j].i;
           data3[i][j].i = data1[i][j].r * data2[i][j].i + data1[i]
[j].i * data2[i][j].r;
       }
   }

   /*1D IFFT Row wise on Output Matrix*/
   for (i = 0; i<chunk; i++)
   {
       fft1d(data3[i], MAXN, 1);
   }

   if (rank == 0)
       t7 = MPI_Wtime();

   /*Collecting data3*/
   if (rank == 0)
   {
       for (p = 1; p < nprocs; p++)
       {
           MPI_Recv(&data3[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
       }
   }
   else
   {
       MPI_Send(&data3[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD);
   }
```

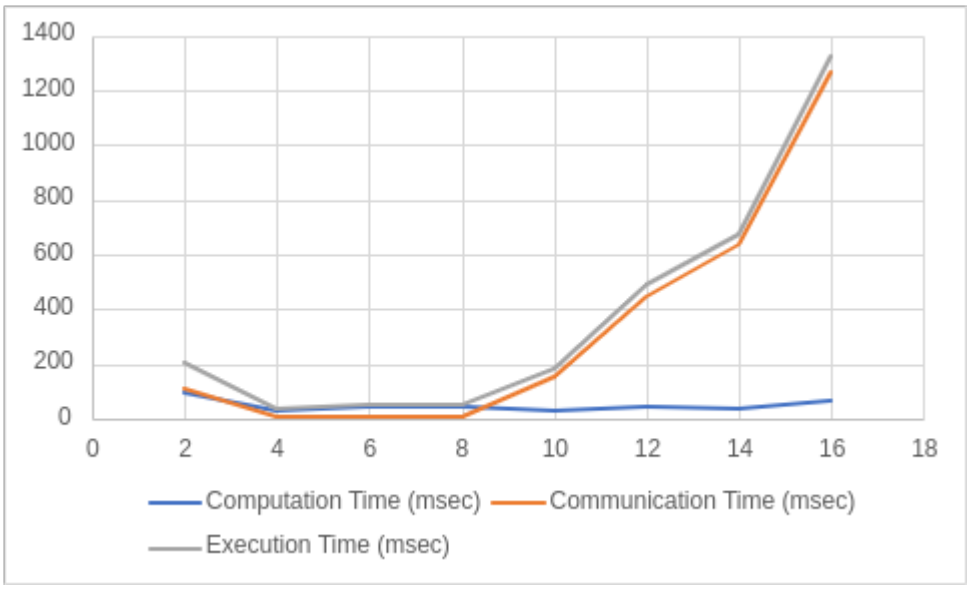
```
    if (rank == 0)
        t8 = MPI_Wtime();

    /*Transpose*/
    for (j = 0; j < MAXN; j++)
    {
        for (i = j; i < MAXN; i++)
        {
            C_SWAP(data3[i][j], data3[j][i]);
        }
    }

    if (rank == 0)
        t9 = MPI_Wtime();

    /*Distribution of data3*/
    if (rank == 0)
    {
        for (p = 1; p < nprocs; p++)
        {
            MPI_Send(&data3[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD);
        }
    }
    else
    {
        MPI_Recv(&data3[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    if (rank == 0)
        t10 = MPI_Wtime();

    /*1D IFFT Row wise(Transposed Column) */
    for (i = 0; i<chunk; i++)
    {
        fft1d(data3[i], MAXN, 1);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
```

```
        t11 = MPI_Wtime();

    /*Collecting data3*/
    if (rank == 0)
    {
        for (p = 1; p < nprocs; p++)
        {
            MPI_Recv(&data3[p*chunk][0], chunk*MAXN, MPI_COMPLEX, p, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    else
    {
        MPI_Send(&data3[0][0], chunk*MAXN, MPI_COMPLEX, 0, 0,
MPI_COMM_WORLD);
    }

    if (rank == 0)
        t12 = MPI_Wtime();

    if (rank == 0)
    {
        file_write();
        double c1 = (t2-t1)+(t4-t3)+(t6-t5)+(t8-t7)+(t10-t9)+(t12-t11);
        double c2 = (t3-t2)+(t5-t4)+(t7-t6)+(t9-t8)+(t11-t10);
        printf("Computation Cost: %f msecs\n",c2*1000);
        printf("Communication Cost: %f msecs\n",c1*1000);
    }
    MPI_Finalize();
}
```

Output:



Processors	Computation Time (msec)	Communication Time (msec)	Execution Time (msec)
2	93.622923	111.971378	205.594301
4	32.833338	4.387856	37.221194
6	45.901775	9.323835	55.22561
8	42.784452	9.508133	52.292585
10	31.69322	153.634548	185.327768
12	40.955544	449.529648	490.485192
14	38.102388	634.927273	673.029661
16	63.523531	1264.231205	1327.754736



Conclusion:

So, as we can see when we increase the number of Processors for execution of the code the execution time also increases. This observation is not because of the computation time but as we increase the number of processes the communication overhead between the processors increases and thus there is increase in execution time.

Problem Statement 2:

Implement dot products using MPI. Use a different number of processes and analyze the performance.

Code :

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

#define NELMS 100000
#define MASTER 0
#define MAXPROCS 16

int dot_product();
void init_lst();
void print_lst();

int main()
{
   int i,n,vector_x[NELMS],vector_y[NELMS];
   int prod,sidx,eidx,size;
   int pid,nprocs, rank;
   double stime,etime;

   MPI_Status status;
   MPI_Comm world;

   n = 100000;
   if (n > NELMS)
   {
       printf("n=%d > N=%d\n",n,NELMS);
       return 0;
   }

   MPI_Init(NULL, NULL);
   world = MPI_COMM_WORLD;
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
```

```
   int portion = n / nprocs;
   sidx = pid * portion;
   eidx = sidx + portion;
   init_lst(vector_x, n);
   init_lst(vector_y, n);

   int tmp_prod[nprocs];
   for (i = 0; i < nprocs; i++)
       tmp_prod[i] = 0;

   stime = MPI_Wtime();

   if (pid == MASTER)
   {
       prod = dot_product(sidx, eidx, vector_x, vector_y, n);
       for (i = 1; i < nprocs; i++)
           MPI_Recv(&tmp_prod[i-1], 1, MPI_INT, i, 123, MPI_COMM_WORLD,
&status);
   }
   else
   {
       prod = dot_product(sidx, eidx, vector_x, vector_y, n);
       MPI_Send(&prod, 1, MPI_INT, MASTER, 123, MPI_COMM_WORLD);
   }

   if (pid == MASTER)
   {
       for (i = 0; i < nprocs; i++)
           prod += tmp_prod[i];
   }

   etime = MPI_Wtime();

   if (pid == MASTER)
   {
       //print_lst(vector_x,n);
       //print_lst(vector_y,n);
       printf("pid=%d: Final Product = %d\n",pid,prod);
       printf("pid=%d: Elapsed Time = %f\n",pid,etime-stime);
   }
```

```
   MPI_Finalize();

}

int dot_product(int s,int e, int x[], int y[], int n)
{
   int i,prod=0;

   for (i = s; i < e; i++)
       prod = prod + x[i] * y[i];

   return prod;
}

void init_lst(int *l,int n)
{
   int i;
   for (i=0; i<n; i++)
       *l++ = i;
}

void print_lst(int l[],int n)
{
   int i;

   for (i=0; i<n; i++)
       printf("%d ", l[i]);
   printf("\n");
}
```

```
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpic++ dotProduct.c -o dotProduct
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 2 ./dotProduct
pid=0: Final Product = 216474736
pid=0: Elapsed Time = 0.000140
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 4 ./dotProduct
pid=0: Final Product = 216474736
pid=0: Elapsed Time = 0.000159
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 8 ./dotProduct
pid=0: Final Product = 216474736
pid=0: Elapsed Time = 0.000094
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 16 ./dotProduct
pid=0: Final Product = 216474736
pid=0: Elapsed Time = 0.000043
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$
```

Processors (Size 100000)	Execution Time (sec)
2	0.000148
4	0.000118
8	0.000402
16	0.023692

Conclusion:

So, as we can see when we increase the number of Processors for execution of the code the execution time also increases. This observation is not because of the computation time but as we increase the number of processes the communication overhead between the processors increases and thus there is increase in execution time.

Problem Statement 3:

Implement Prefix sum using MPI. Use a different number of processes and analyze the performance.

Code :

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

int main(int argc, char* argv[])
{
   int  my_rank; /* rank of process */
   int  p;       /* number of processes */

   MPI_Status status ;   /* return status for receive */
   int value;

   /* start up MPI */
   MPI_Init(&argc, &argv);

   /* find out process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* find out number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   int prefix_arr[p];

   /* getting input and scatter values */
   if(my_rank == 0)
   {
       int i;
       for(i = 0; i < p; ++i)
           prefix_arr[i] = i + 1;
   }

   double start = MPI_Wtime();

   //all call scatter
   MPI_Scatter(prefix_arr, 1, MPI_INT, &value, 1, MPI_INT, 0,
MPI_COMM_WORLD);
```

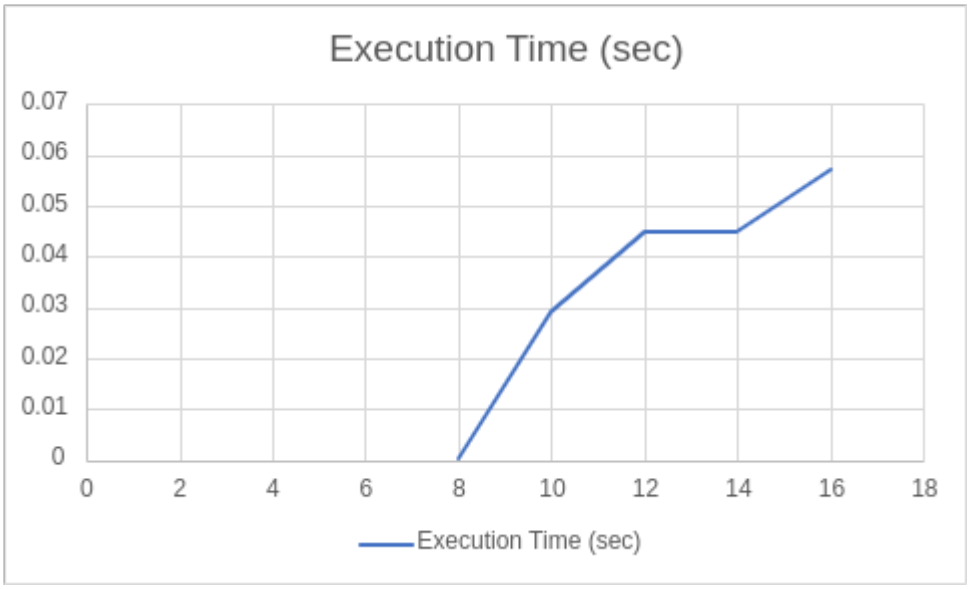
```
   /*
   prefix sum:
   repeat log n times
   each time, if we are the chosen one, we receve a value from someone
and add to ours
   otherwise, we send to the chosen one
   */
   int i;
   int logn = log2(p);
   for(i = 0; i <= logn; i++)
   {
       int lower_bound = pow(2,i);
       int upper_bound = p - lower_bound;

       if(upper_bound < lower_bound)
           upper_bound = lower_bound;

       if(my_rank < lower_bound)
       {
           int send = (int) (my_rank + pow(2,i));
           if(send >= p)
               continue;

//          printf("%d sending to %d\n", my_rank, (int)
(my_rank+pow(2,i)));
           MPI_Send(&value, 1, MPI_INT, (int) (my_rank+pow(2,i)), 0,
MPI_COMM_WORLD);
       }
       else if(my_rank >= upper_bound)
       {
           int recv = (int) (my_rank - pow(2,i));
           if(recv >= p)
               continue;

           int recv_value;
//          printf("%d receving..\n", my_rank);
           MPI_Recv(&recv_value, 1, MPI_INT, (my_rank - pow(2,i)), 0,
MPI_COMM_WORLD, &status);
           value += recv_value;
       }
       else
       {
           int send = (int) (my_rank + pow(2,i));
```

```
            int recv = (int) (my_rank - pow(2,i));
            if(send >= p || recv >= p)
                continue;

//           printf("%d sending to %d\n", my_rank, (int)
(my_rank+pow(2,i)));
            MPI_Send(&value, 1, MPI_INT, (int) (my_rank+pow(2,i)), 0,
MPI_COMM_WORLD);

//           printf("%d receving..\n", my_rank);
            int recv_value;
            MPI_Status status;
            MPI_Recv(&recv_value, 1, MPI_INT, (my_rank - pow(2,i)), 0,
MPI_COMM_WORLD, &status);
            value += recv_value;
        }
    }

    //after algorithm, each processor hols its own prefix sum
    //we gather at rank
    int gather[p];
    MPI_Gather(&value, 1, MPI_INT, gather, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if(my_rank == 0)
    {
        double end = MPI_Wtime();

        printf("\nPerfix Sum Execution Time: %f\n", end - start);
    }

    /* shut down MPI */
    MPI_Finalize();

    return 0;
}
```

Output :

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpic++ PrefixSum.c -o PrefixSum
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 2 ./PrefixSum

 Perfix Sum Execution Time: 0.000057
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 4 ./PrefixSum

 Perfix Sum Execution Time: 0.000067
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 6 ./PrefixSum

 Perfix Sum Execution Time: 0.000432
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 8 ./PrefixSum

 Perfix Sum Execution Time: 0.000081
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 10 ./PrefixSum

 Perfix Sum Execution Time: 0.029108
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 12 ./PrefixSum

 Perfix Sum Execution Time: 0.045063
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 14 ./PrefixSum

 Perfix Sum Execution Time: 0.044914
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$ mpirun -n 16 ./PrefixSum

 Perfix Sum Execution Time: 0.057190
abhi@kingsman:~/Documents/Sem7/Assignment/HPC/Assignment8$
```

Processors	Execution Time (sec)
2	0.000057
4	0.000067
6	0.000432
8	0.000081
10	0.029108
12	0.045063
14	0.044914
16	0.05719



Conclusion:

So, as we can see when we increase the number of Processors for execution of the code the execution time also increases. This observation is not because of the computation time but as we increase the number of processes the communication overhead between the processors increases and thus there is increase in execution time.

Github Link : https://github.com/pavanshinde7494/HPC-Assignment