Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

High Performance Computing Lab

Class: Final Year (Computer Science and Engineering) **Year:** 2022-23

PRN: 2019BTECS00089 – Piyush Pramod Mhaske **Batch**: B3

Practical 8

Github link: https://github.com/Piyush4620/2019BTECS00089HPCLab

Hosted Link : https://better-sidecar-c10.notion.site/HPC-038e2693a633408c8604841fc50f74e2

Question 1:

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

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

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

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

#define N 512

void c_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;
        }
    }
}

void getData(char fileName[15], complex **data)
{
    FILE *fp = fopen(fileName, "r");

    int i, j, result;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            result = fscanf(fp, "%g", &data[i][j].r);
            data[i][j].i = 0.00;
        }
    }

    fclose(fp);
}

void transpose(complex **data, complex **transp)
{
    int i, j;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            transp[j][i] = data[i][j];
}

void mmpoint(complex **data1, complex **data2, complex **data3)
{

    int i, j;

    float real, imag;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; 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);
        }
    }
}

void printfile(char fileName[15], complex **data)
{

    FILE *fp = fopen(fileName, "w");

    int i, j;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            fprintf(fp, "   %.7e", data[i][j].r);
```

```
        }
        fprintf(fp, "\n");
    }

    fclose(fp);
}

int main(int argc, char **argv)
{
    int my_rank, p, source = 0, dest, x;

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

    char fileName1[15] = "sample/in1";
    char fileName2[15] = "sample/in2";
    char fileName3[15] = "mpi_out_test";

    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    MPI_Comm_size(MPI_COMM_WORLD, &p);

    /* Setup description of the 4 MPI_FLOAT fields x, y, z, velocity */
    MPI_Datatype mystruct;
    int blocklens[2] = {1, 1};
    MPI_Aint indices[2] = {0, sizeof(float)};
    MPI_Datatype old_types[2] = {MPI_FLOAT, MPI_FLOAT};

    /* Make relative */
    MPI_Type_struct(2, blocklens, indices, old_types, &mystruct);
    MPI_Type_commit(&mystruct);

    int i, j;

    double startTime, stopTime;

    // Starting and send rows of data1, data2

    int offset;

    int tag = 345;

    int rows = N / p;

    int lb = my_rank * rows;
    int hb = lb + rows;
```

```
printf("%d have lb = %d and hb = %d\n", my_rank, lb, hb);

// Starting and send rows of data1, data2

if (my_rank == 0)
{
    getData(fileName1, data1);
    getData(fileName2, data2);

    /* Start Clock */
    printf("\nStarting clock.\n");
    startTime = MPI_Wtime();

    for (i = 1; i < p; i++)
    {
        offset = i * rows;
        for (j = offset; j < (offset + rows); j++)
        {
            MPI_Send(&data1[j][0], N, mystruct, i, tag, MPI_COMM_WORLD);
            MPI_Send(&data2[j][0], N, mystruct, i, tag, MPI_COMM_WORLD);
        }
    }
}
else
{

    for (j = lb; j < hb; j++)
    {
        MPI_Recv(data1[j], N, mystruct, 0, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(data2[j], N, mystruct, 0, tag, MPI_COMM_WORLD, &status);
    }
}

// Doing fft1d forward for data1 and data2 rows

vec = (complex *)malloc(N * sizeof(complex));

for (i = lb; i < hb; i++)
{
    for (j = 0; j < N; j++)
    {
        vec[j] = data1[i][j];
    }
    c_fft1d(vec, N, -1);
    for (j = 0; j < N; j++)
    {
        data1[i][j] = vec[j];
    }
}

free(vec);

vec = (complex *)malloc(N * sizeof(complex));

for (i = lb; i < hb; i++)
{
    for (j = 0; j < N; j++)
    {
        vec[j] = data2[i][j];
    }
```

```
        c_fft1d(vec, N, -1);
        for (j = 0; j < N; j++)
        {
            data2[i][j] = vec[j];
        }
    }

    free(vec);

    // Receving rows of data1, data2

    if (my_rank == 0)
    {
        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Recv(data1[j], N, mystruct, i, tag, MPI_COMM_WORLD, &status);
                MPI_Recv(data2[j], N, mystruct, i, tag, MPI_COMM_WORLD, &status);
            }
        }
    }
    else
    {

        for (j = lb; j < hb; j++)
        {
            MPI_Send(&data1[j][0], N, mystruct, 0, tag, MPI_COMM_WORLD);
            MPI_Send(&data2[j][0], N, mystruct, 0, tag, MPI_COMM_WORLD);
        }
    }

    // Starting and send columns of data1, data2

    if (my_rank == 0)
    {
        transpose(data1, data3);
        transpose(data2, data4);

        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Send(&data3[j][0], N, mystruct, i, tag, MPI_COMM_WORLD);
                MPI_Send(&data4[j][0], N, mystruct, i, tag, MPI_COMM_WORLD);
            }
        }
    }
    else
    {
        for (j = lb; j < hb; j++)
        {
            MPI_Recv(data3[j], N, mystruct, 0, tag, MPI_COMM_WORLD, &status);
            MPI_Recv(data4[j], N, mystruct, 0, tag, MPI_COMM_WORLD, &status);
        }
    }

    // Doing fft1d forward for data1 and data2 columns
```

```
    vec = (complex *)malloc(N * sizeof(complex));

    for (i = lb; i < hb; i++)
    {
        for (j = 0; j < N; j++)
        {
            vec[j] = data3[i][j];
        }
        c_fft1d(vec, N, -1);
        for (j = 0; j < N; j++)
        {
            data3[i][j] = vec[j];
        }
    }

    free(vec);

    vec = (complex *)malloc(N * sizeof(complex));

    for (i = lb; i < hb; i++)
    {
        for (j = 0; j < N; j++)
        {
            vec[j] = data4[i][j];
        }
        c_fft1d(vec, N, -1);
        for (j = 0; j < N; j++)
        {
            data4[i][j] = vec[j];
        }
    }

    free(vec);

    // Receving columns of data1, data2

    if (my_rank == 0)
    {
        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Recv(data3[j], N, mystruct, i, tag, MPI_COMM_WORLD, &status);
                MPI_Recv(data4[j], N, mystruct, i, tag, MPI_COMM_WORLD, &status);
            }
        }
    }
    else
    {
        for (j = lb; j < hb; j++)
        {
            MPI_Send(&data3[j][0], N, mystruct, 0, tag, MPI_COMM_WORLD);
            MPI_Send(&data4[j][0], N, mystruct, 0, tag, MPI_COMM_WORLD);
        }
    }

    if (my_rank == 0)
    {
        transpose(data3, data1);
        transpose(data4, data2);
```

```
        mmpoint(data1, data2, data3);
    }

    // Starting and send rows of data1, data2

    if (my_rank == 0)
    {
        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Send(&data3[j][0], N, mystruct, i, tag, MPI_COMM_WORLD);
            }
        }
    }
    else
    {

        for (j = lb; j < hb; j++)
        {
            MPI_Recv(data3[j], N, mystruct, 0, tag, MPI_COMM_WORLD, &status);
        }
    }

    // Doing fft1d forward for data1 and data2 rows

    vec = (complex *)malloc(N * sizeof(complex));

    for (i = lb; i < hb; i++)
    {
        for (j = 0; j < N; j++)
        {
            vec[j] = data3[i][j];
        }
        c_fft1d(vec, N, 1);
        for (j = 0; j < N; j++)
        {
            data3[i][j] = vec[j];
        }
    }

    free(vec);

    // Receving rows of data1, data2

    if (my_rank == 0)
    {
        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Recv(data3[j], N, mystruct, i, tag, MPI_COMM_WORLD, &status);
            }
        }
    }
    else
    {
        for (j = lb; j < hb; j++)
        {
```

```
            MPI_Send(&data3[j][0], N, mystruct, 0, tag, MPI_COMM_WORLD);
        }
    }

    // Starting and send columns of data1, data2

    if (my_rank == 0)
    {
        transpose(data3, data4);

        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Send(&data4[j][0], N, mystruct, i, tag, MPI_COMM_WORLD);
            }
        }
    }
    else
    {
        for (j = lb; j < hb; j++)
        {
            MPI_Recv(data4[j], N, mystruct, 0, tag, MPI_COMM_WORLD, &status);
        }
    }

    // Doing fft1d forward for data1 and data2 columns

    vec = (complex *)malloc(N * sizeof(complex));

    for (i = lb; i < hb; i++)
    {
        for (j = 0; j < N; j++)
        {
            vec[j] = data4[i][j];
        }
        c_fft1d(vec, N, 1);
        for (j = 0; j < N; j++)
        {
            data4[i][j] = vec[j];
        }
    }

    free(vec);

    // Receving columns of data1, data2

    if (my_rank == 0)
    {
        for (i = 1; i < p; i++)
        {
            offset = i * rows;
            for (j = offset; j < (offset + rows); j++)
            {
                MPI_Recv(data4[j], N, mystruct, i, tag, MPI_COMM_WORLD, &status);
            }
        }
    }
    else
    {
```

```
        for (j = lb; j < hb; j++)
        {
            MPI_Send(&data4[j][0], N, mystruct, 0, tag, MPI_COMM_WORLD);
        }
    }

    if (my_rank == 0)
    {
        transpose(data4, data3);
        /* Stop Clock */
        stopTime = MPI_Wtime();

        printf("\nElapsed time = %lf s.\n", (stopTime - startTime));
        printf("---------------------------------------------\n");
    }

    MPI_Finalize();

    if (my_rank == 0)
    {
        printfile(fileName3, data3);
    }

    free(data1);
    free(data2);
    free(data3);
    free(data4);

    return 0;
}
```

Output:

```
PS D:\Academics\Fourth Year\HPC Lab\Practical 8>
 *  History restored

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
4 have lb = 144 and hb = 180
12 have lb = 432 and hb = 468
8 have lb = 288 and hb = 324
2 have lb = 72 and hb = 108
7 have lb = 252 and hb = 288
13 have lb = 468 and hb = 504
6 have lb = 216 and hb = 252
10 have lb = 360 and hb = 396
0 have lb = 0 and hb = 36

Starting clock.

Elapsed time = 0.075651 s.
-----------------------------------------
PS D:\Academics\Fourth Year\HPC Lab\Practical 8>
```

Problem Statement 2:

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

```
#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);
        exit(1);
    }

    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)
    {
        printf("pid=%d: final prod=%d\n", pid, prod);
        printf("pid=%d: elapsed=%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");
}
```

Output:

```
PS D:\Academics\Fourth Year\HPC Lab\Practical 8> mpiexec -np 10 .\dotproduct.exe
pid=0: final prod=216474736
pid=0: elapsed=0.009680
PS D:\Academics\Fourth Year\HPC Lab\Practical 8>
```

Problem Statement 3:

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

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

#define ARRAY_SIZE 1048576

int main(int argc, char *argv[])
{
    int rank;
    int size;

    if (MPI_Init(&argc, &argv) != MPI_SUCCESS)
    {
        printf("Unable to initialize MPI!\n");
        return -1;
    }
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (ARRAY_SIZE % size != 0 && rank == 0)
    {
        printf("Array size must be multiple of mpi job size.\n");
```

```
        return -1;
    }
    MPI_Status status;
    int *array = (int *)malloc(sizeof(int) * ARRAY_SIZE);
    int *chunk = (int *)malloc(sizeof(int) * ARRAY_SIZE / size);
    int i = 0;

    int total_sum = 0;
    for (i = 0; i < ARRAY_SIZE; i++)
    {
        array[i] = rand() % 1024;
        total_sum += array[i];
    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Scatter(array, ARRAY_SIZE / size, MPI_INT, chunk, ARRAY_SIZE / size, MPI_INT, 0, MPI_COMM_WORLD);
    int sum = 0;
    int temp = 0;
    int key = 1;
    for (i = 0; i < ARRAY_SIZE / size; i++)
        sum += chunk[i];

    /* Fancy stuff to keep the indexes correct */
    /* Number of processes participating halves each time */
    while (key <= size / 2)
    {
        if ((rank + 1) % key == 0)
            if (rank / key % 2 == 0)
            {
                MPI_Send(&sum, 1, MPI_INT, rank + key, 0, MPI_COMM_WORLD);
            }
            else
            {
                MPI_Recv(&temp, 1, MPI_INT, rank - key, 0, MPI_COMM_WORLD, &status);
                sum += temp;
            }
        key = 2 * key;
        MPI_Barrier(MPI_COMM_WORLD);
    }
    if (rank == size - 1)
    {
        printf("Total: %d\n", sum);
        printf("Correct Sum: %d\n", total_sum);
    }
    free(array);
    free(chunk);
    MPI_Finalize();
    return 0;
}
```

Output: