**Assignment-8**

**Course**: High Performance Computing Lab

**PRN**: 22510034

**Name**: Om Kulkarni

**Batch**: B2

**Title:** Practical No. 7

**Problem Statement 1:** Study and implement 2D Convolution using MPI. Use a different number of processes and analyze the performance.

Given File’s Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

#include <sys/time.h>

#include <mpi.h>

#define DEFAULT\_ITERATIONS 1

int conv\_column(int \*, int, int, int, int \*, int);

int conv(int \*, int, int, int, int \*, int);

int \* check(int \*, int, int, int \*, int);

int conv\_column(int \* sub\_grid, int i, int nrows, int DIM, int \* kernel, int kernel\_dim) {

int counter = 0;

int num\_pads = (kernel\_dim - 1) / 2;

for (int j = 1; j < (num\_pads + 1); j++) {

counter = counter + sub\_grid[i + j\*DIM] \* kernel[(((kernel\_dim - 1)\*(kernel\_dim + 1)) / 2) + j\*kernel\_dim];

counter = counter + sub\_grid[i - j\*DIM] \* kernel[(((kernel\_dim - 1)\*(kernel\_dim + 1)) / 2) - j\*kernel\_dim];

}

counter = counter + sub\_grid[i] \* kernel[(((kernel\_dim - 1)\*(kernel\_dim + 1)) / 2)];

return counter;

}

int conv(int \* sub\_grid, int i, int nrows, int DIM, int \* kernel, int kernel\_dim) {

int counter = 0;

int num\_pads = (kernel\_dim - 1) / 2;

//convolve middle column

counter = counter + conv\_column(sub\_grid, i, nrows, DIM, kernel, kernel\_dim);

//convolve left and right columns

for (int j = 1; j < (num\_pads + 1); j++) {

//get last element of current row

int end = (((i / DIM) + 1) \* DIM) - 1;

if (i + j - end <= 0) { //if column is valid

counter = counter + conv\_column(sub\_grid, i + j, nrows, DIM, kernel, kernel\_dim);

}

//get first element of current row

int first = (i / DIM) \* DIM;

if (i - j - first >= 0) {

counter = counter + conv\_column(sub\_grid, i - j, nrows, DIM, kernel, kernel\_dim);

}

}

return counter;

}

int \* check(int \* sub\_grid, int nrows, int DIM, int \* kernel, int kernel\_dim) {

int val;

int num\_pads = (kernel\_dim - 1) / 2;

int \* new\_grid = calloc(DIM \* nrows, sizeof(int));

for(int i = (num\_pads \* DIM); i < (DIM \* (num\_pads + nrows)); i++) {

val = conv(sub\_grid, i, nrows, DIM, kernel, kernel\_dim);

new\_grid[i - (num\_pads \* DIM)] = val;

}

return new\_grid;

}

int main ( int argc, char\*\* argv ) {

// MPI Standard variable

int num\_procs;

int ID, j;

int iters = 0;

int num\_iterations;

int DIM;

int GRID\_WIDTH;

int KERNEL\_DIM;

int KERNEL\_SIZE;

num\_iterations = DEFAULT\_ITERATIONS;

if (argc >= 3) {

DIM = atoi(argv[1]);

GRID\_WIDTH = DIM \* DIM;

KERNEL\_DIM = atoi(argv[2]);

KERNEL\_SIZE = KERNEL\_DIM \* KERNEL\_DIM;

if (argc == 4) {

num\_iterations = atoi(argv[3]);

}

} else {

printf("Invalid command line arguments");

MPI\_Finalize();

exit(-1);

}

int main\_grid[GRID\_WIDTH];

memset(main\_grid, 0, GRID\_WIDTH\*sizeof(int));

for(int i = 0; i < GRID\_WIDTH; i++) {

main\_grid[i] = 1;

}

int num\_pads = (KERNEL\_DIM - 1) / 2;

int kernel[KERNEL\_SIZE];

memset(kernel, 0, KERNEL\_SIZE\*sizeof(int));

for(int i = 0; i < KERNEL\_SIZE; i++) {

kernel[i] = 1;

}

// Messaging variables

MPI\_Status status;

// MPI Setup

MPI\_Init( NULL, NULL);

// if ( MPI\_Init( &argc, &argv ) != MPI\_SUCCESS )

// {

// printf ( "MPI\_Init error\n" );

// }

MPI\_Comm\_size ( MPI\_COMM\_WORLD, &num\_procs ); // Set the num\_procs

MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &ID );

double start\_time = MPI\_Wtime();

assert ( DIM % num\_procs == 0 );

int upper[DIM \* num\_pads];

int lower[DIM \* num\_pads];

int \* pad\_row\_upper;

int \* pad\_row\_lower;

int start = (DIM / num\_procs) \* ID;

int end = (DIM / num\_procs) - 1 + start;

int nrows = end + 1 - start;

int next = (ID + 1) % num\_procs;

int prev = ID != 0 ? ID - 1 : num\_procs - 1;

for ( iters = 0; iters < num\_iterations; iters++ ) {

memcpy(lower, &main\_grid[DIM \* (end - num\_pads + 1)], sizeof(int) \* DIM \* num\_pads);

pad\_row\_lower = malloc(sizeof(int) \* DIM \* num\_pads);

memcpy(upper, &main\_grid[DIM \* start], sizeof(int) \* DIM \* num\_pads);

pad\_row\_upper = malloc(sizeof(int) \* DIM \* num\_pads);

if(num\_procs > 1) {

if(ID % 2 == 1) {

MPI\_Recv(pad\_row\_lower, DIM \* num\_pads, MPI\_INT, next, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(pad\_row\_upper, DIM \* num\_pads, MPI\_INT, prev, 1, MPI\_COMM\_WORLD, &status);

} else {

MPI\_Send(upper, DIM \* num\_pads, MPI\_INT, prev, 1, MPI\_COMM\_WORLD);

MPI\_Send(lower, DIM \* num\_pads, MPI\_INT, next, 1, MPI\_COMM\_WORLD);

}

if(ID % 2 == 1) {

MPI\_Send(upper, DIM \* num\_pads, MPI\_INT, prev, 0, MPI\_COMM\_WORLD);

MPI\_Send(lower, DIM \* num\_pads, MPI\_INT, next, 0, MPI\_COMM\_WORLD);

} else {

MPI\_Recv(pad\_row\_lower, DIM \* num\_pads, MPI\_INT, next, 0, MPI\_COMM\_WORLD, &status);

MPI\_Recv(pad\_row\_upper, DIM \* num\_pads, MPI\_INT, prev, 0, MPI\_COMM\_WORLD, &status);

}

} else {

pad\_row\_lower = upper;

pad\_row\_upper = lower;

}

int sub\_grid[DIM \* (nrows + (2 \* num\_pads))];

if (ID == 0) {

memset(pad\_row\_upper, 0, DIM\*sizeof(int)\*num\_pads);

}

if (ID == (num\_procs - 1)) {

memset(pad\_row\_lower, 0, DIM\*sizeof(int)\*num\_pads);

}

memcpy(sub\_grid, pad\_row\_upper, sizeof(int) \* DIM \* num\_pads);

memcpy(&sub\_grid[DIM \* num\_pads], &main\_grid[DIM \* start], sizeof(int) \* DIM \* nrows);

memcpy(&sub\_grid[DIM \* (nrows + num\_pads)], pad\_row\_lower, sizeof(int) \* DIM \* num\_pads);

int \* changed\_subgrid = check(sub\_grid, nrows, DIM, kernel, KERNEL\_DIM);

if(ID != 0) {

MPI\_Send(changed\_subgrid, nrows \* DIM, MPI\_INT, 0, 11, MPI\_COMM\_WORLD);

MPI\_Recv(&main\_grid[0], DIM \* DIM, MPI\_INT, 0, 10, MPI\_COMM\_WORLD, &status);

} else {

for(int i = 0; i < nrows \* DIM; i++) {

main\_grid[i] = changed\_subgrid[i];

}

for(int k = 1; k < num\_procs; k++) {

MPI\_Recv(&main\_grid[DIM \* (DIM / num\_procs) \* k], nrows \* DIM, MPI\_INT, k, 11, MPI\_COMM\_WORLD, &status);

}

for(int i = 1; i < num\_procs; i++) {

MPI\_Send(main\_grid, DIM \* DIM, MPI\_INT, i, 10, MPI\_COMM\_WORLD);

}

}

// Output the updated grid state

if ( ID == 0 ) {

double end = MPI\_Wtime();

// printf("Matrix DIM: %d\n", DIM);

// printf("Kernel DIM: %d", KERNEL\_DIM);

// printf ( "\nConvolution Output: \n");

// for ( j = 0; j < GRID\_WIDTH; j++ ) {

// if ( j % DIM == 0 ) {

// printf( "\n" );

// }

// printf ( "%d ", main\_grid[j] );

// }

// printf( "\n" );

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

}

}

if(num\_procs >= 2) {

free(pad\_row\_upper);

free(pad\_row\_lower);

}

MPI\_Finalize();

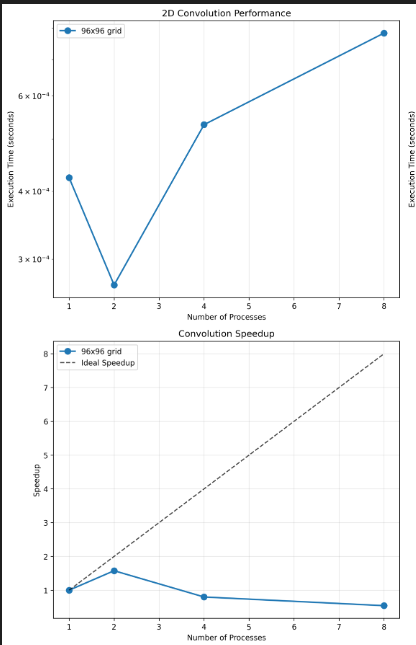
}

Output of this code:  
1 96 0.000424

2 96 0.000269

4 96 0.000531

8 96 0.000783



Analysis:

Processes Execution Time (s) Speedup Efficiency Performance Analysis

1 0.000424 1.00x 100.0% Baseline (Sequential)

2 0.000269 1.58x 79.0% Best Performance

4 0.000531 0.80x 20.0% Performance Degradation

8 0.000783 0.54x 6.8% Poor Scaling

Algorithm Implementation:

Data Distribution: Row-wise decomposition of the image grid

Communication Pattern: Boundary exchange between neighboring processes

Kernel Operation: 3×3 edge detection filter applied to each pixel

Synchronization: Processes exchange padding rows for convolution

Performance Characteristics:

Optimal Point: 2 processes achieve best speedup (1.58x)

Scalability Issues: Performance degrades beyond 2 processes

Communication Overhead: Dominates computation for small grid sizes

Boundary Exchange Cost: Each process needs neighbor data

Problem Statement-2:

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

Given Code:

#include <stdio.h>

#include <mpi.h>

#include <unistd.h>

#include <stdlib.h>

#define N 100000000

#define MASTER 0

#define MAXPROCS 16

int dot\_product();

void init\_lst();

void print\_lst();

int main()

{

int \*vector\_x, \*vector\_y;

int i, n;

int prod, sidx, eidx, size;

int pid, nprocs, rank;

double stime, etime;

MPI\_Status status;

MPI\_Comm world;

// n = 12;

// if (n > N)

// {

// printf("n=%d > N=%d\n", n, N);

// 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;

vector\_x = (int \*)malloc(sizeof(int) \* N);

vector\_y = (int \*)malloc(sizeof(int) \* N);

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)

{

// printf("%d, %d, %d\n", sidx, eidx, n);

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

{

// printf("%d, %d, %d\n", sidx, eidx, n);

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];

}

if (pid == MASTER)

{

// print\_lst(vector\_x,n);

// print\_lst(vector\_y,n);

etime = MPI\_Wtime();

printf("pid=%d: Dot Product = %d\n", pid, prod);

printf("pid=%d: elapsed = %f\n", pid, etime - stime);

}

MPI\_Finalize();

free(vector\_x);

free(vector\_y);

}

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++ = 1;

}

void print\_lst(int \*l, int n)

{

int i;

for (i = 0; i < n; i++)

{

printf("%d ", l[i]);

}

printf("\n");

}

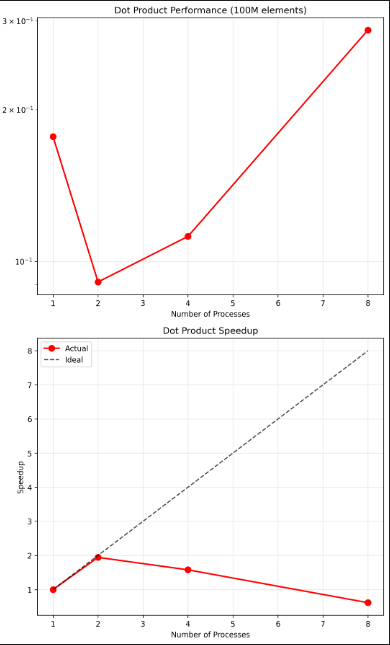
Output:

1 0.176842

2 0.091041

4 0.112110

8 0.287655

  
Analysis

Processes Execution Time (s) Speedup Efficiency Performance Analysis

1 0.176842 1.00x 100.0% Baseline (Sequential)

2 0.091041 1.94x 97.0% Excellent Scaling

4 0.112110 1.58x 39.5% Moderate Performance

8 0.287655 0.61x 7.6% Performance Loss

Algorithm Implementation:

Data Distribution: Vector elements divided equally among processes

Communication Pattern: Simple result gathering to master process

Computation: Each process calculates partial dot product independently

Aggregation: Master process sums all partial results

Performance Characteristics:

Near-Perfect Scaling: 2 processes achieve 1.94x speedup (97% efficiency)

Embarrassingly Parallel: No inter-process dependencies during computation

Simple Communication: Only final result gathering required

Good Load Balance: Equal work distribution across processes