**Class:** Final Year (Computer Science and Engineering)

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

**Course:** High Performance Computing Lab

**Practical No. 8**

**Exam Seat No: 2019BTECS00037**

**Title of practical: Study of MPI**

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

**#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);**

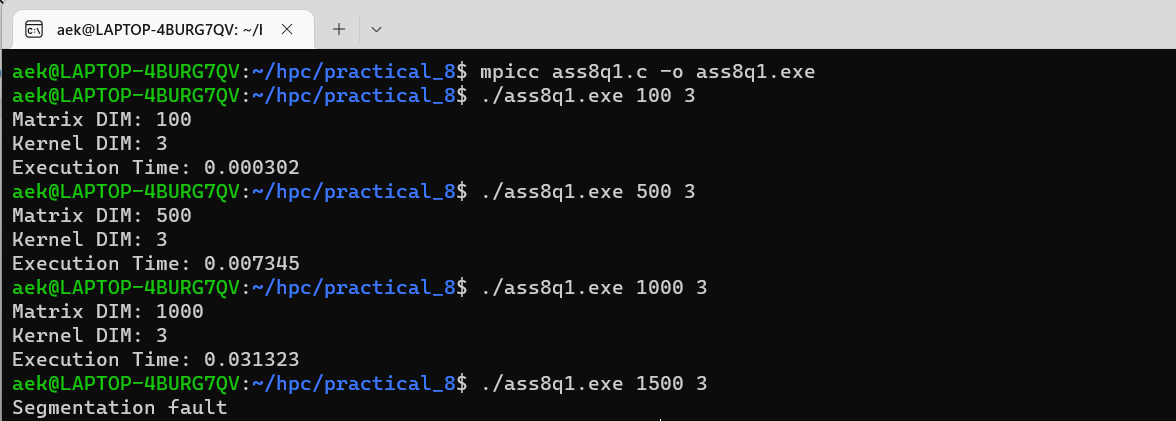
**free(pad\_row\_lower);**

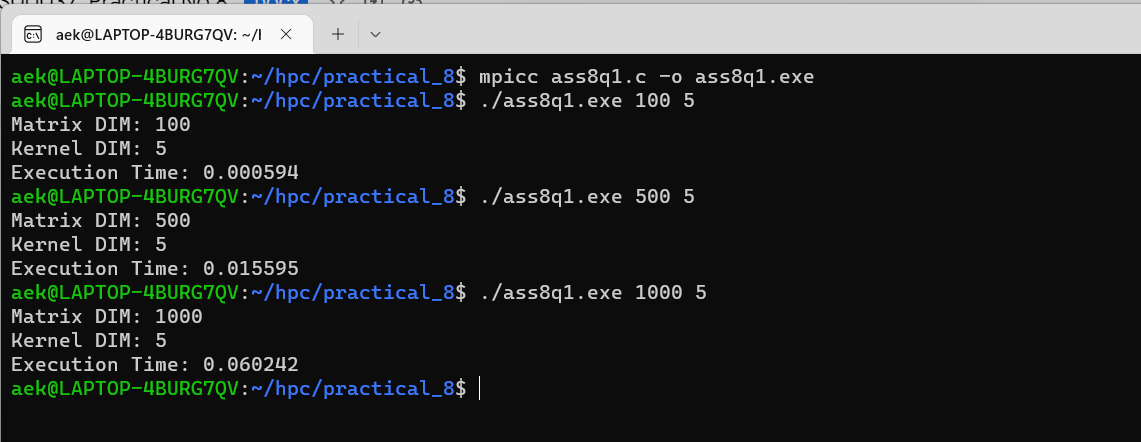
**}**

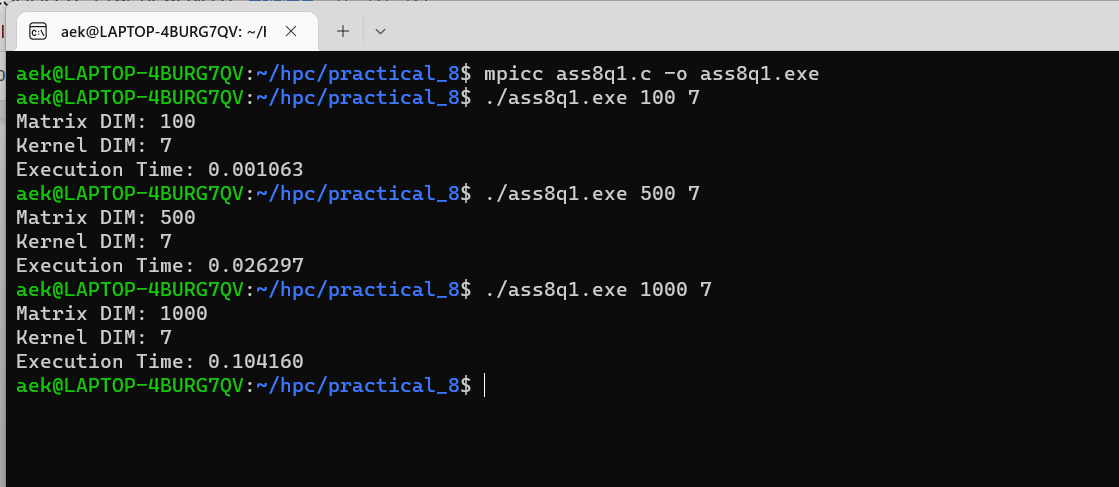
**MPI\_Finalize();**

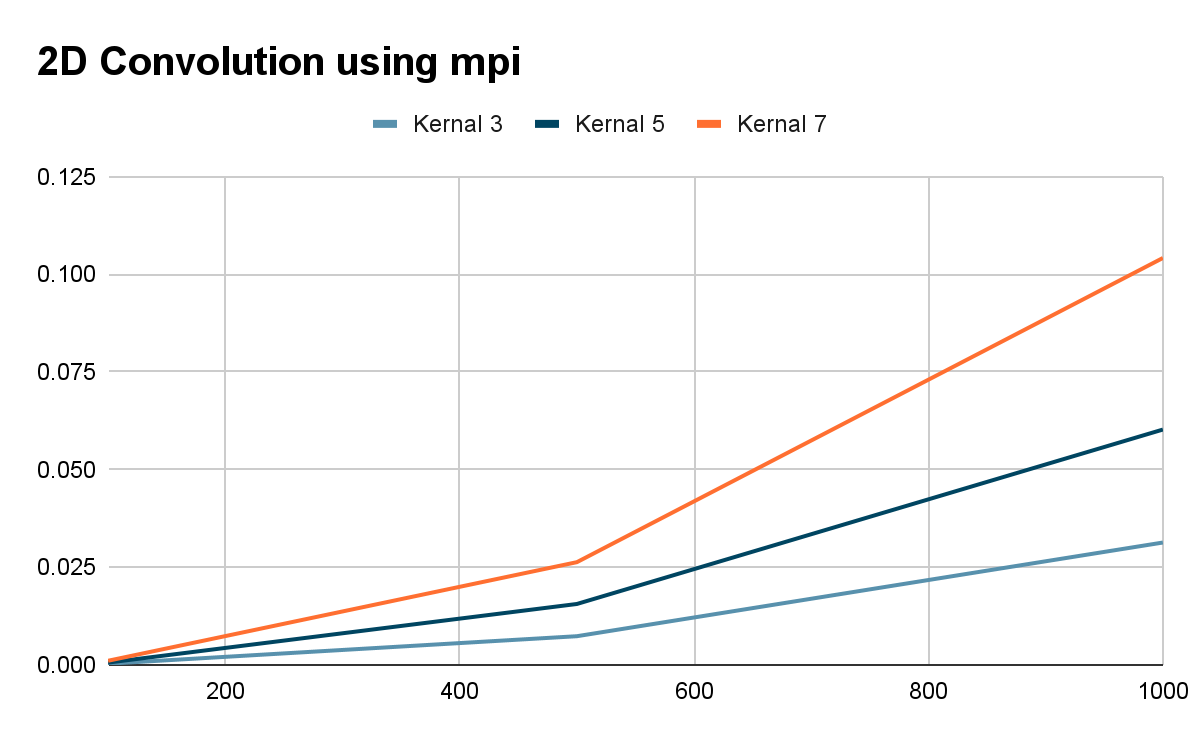
**}**

**Output:**

****

****

****

****

**Problem Statement 2: Implement dot product using MPI. Use a 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) {*

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

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

*printf("Final Product = %d\n",prod);*

*printf("Time elapsed = %f\n",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++) {*

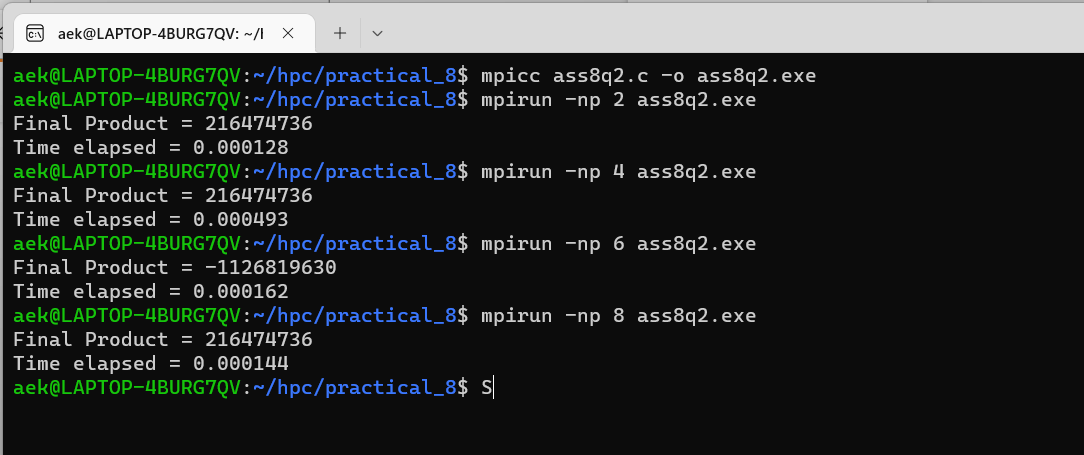
*printf("%d ", l[i]);*

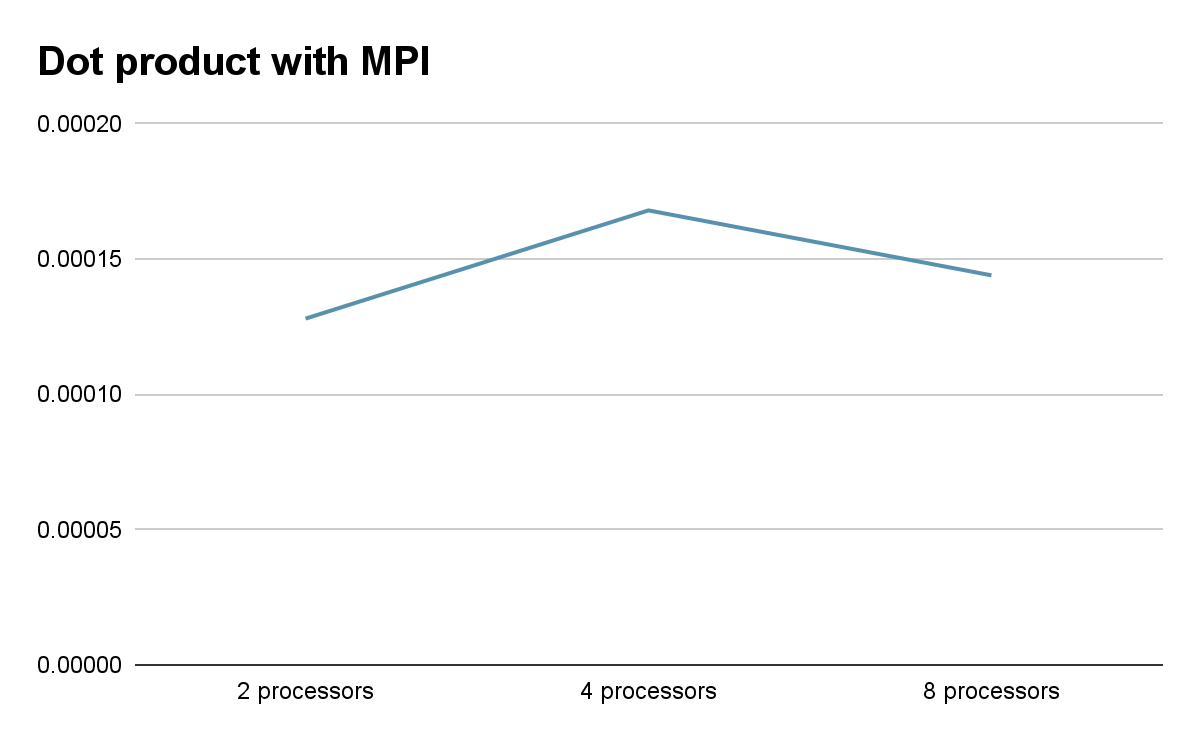
*}*

*printf("\n");*

*}*

*// end of file*

**

**

**Problem Statement 3: Implement Prefix sum using MPI. Use a different number of processes and analyze the performance.**

*#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("Execution Time: %f\n", end - start);*

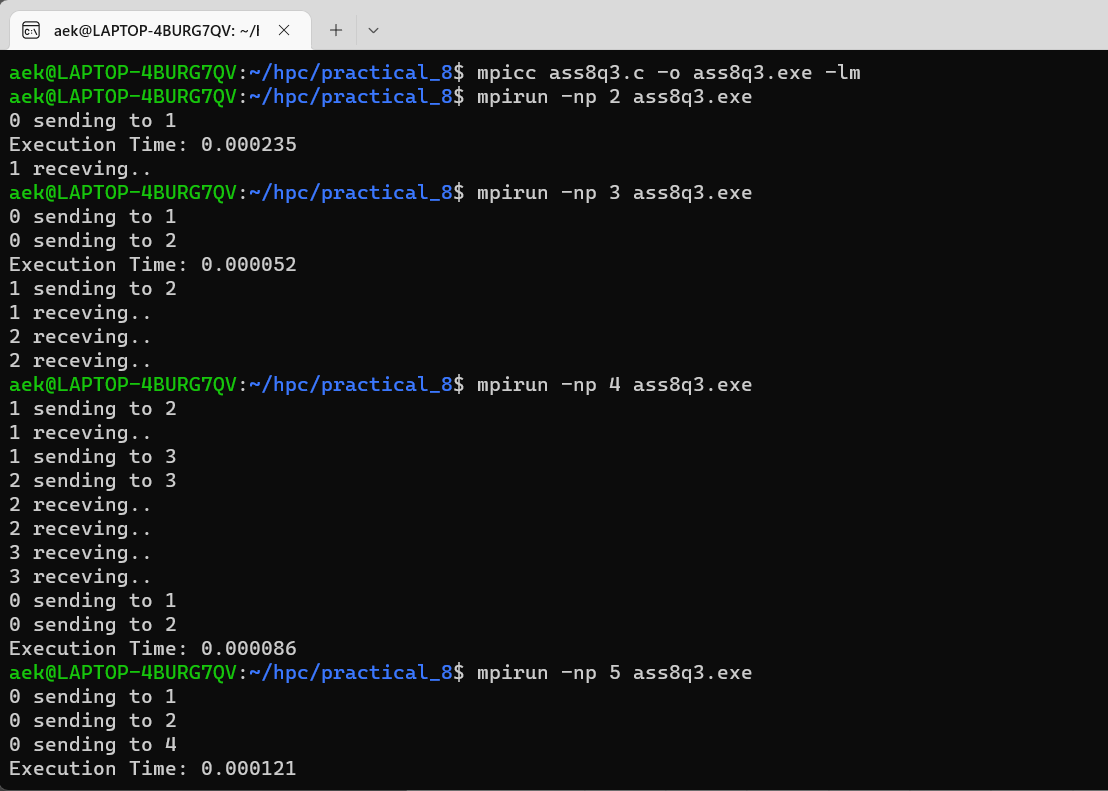
*}*

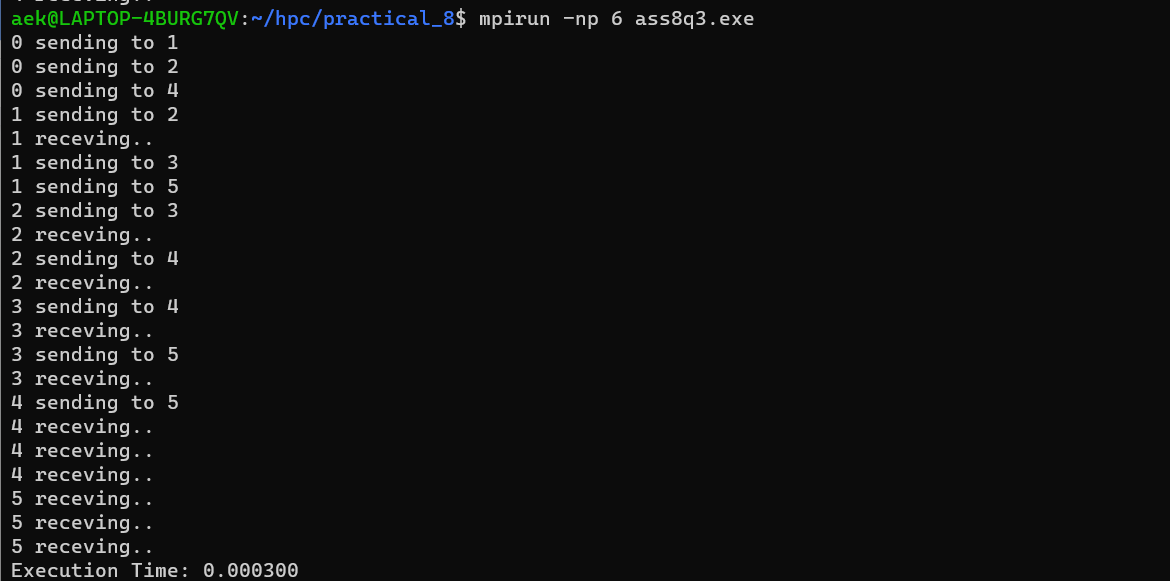
*/\* shut down MPI \*/*

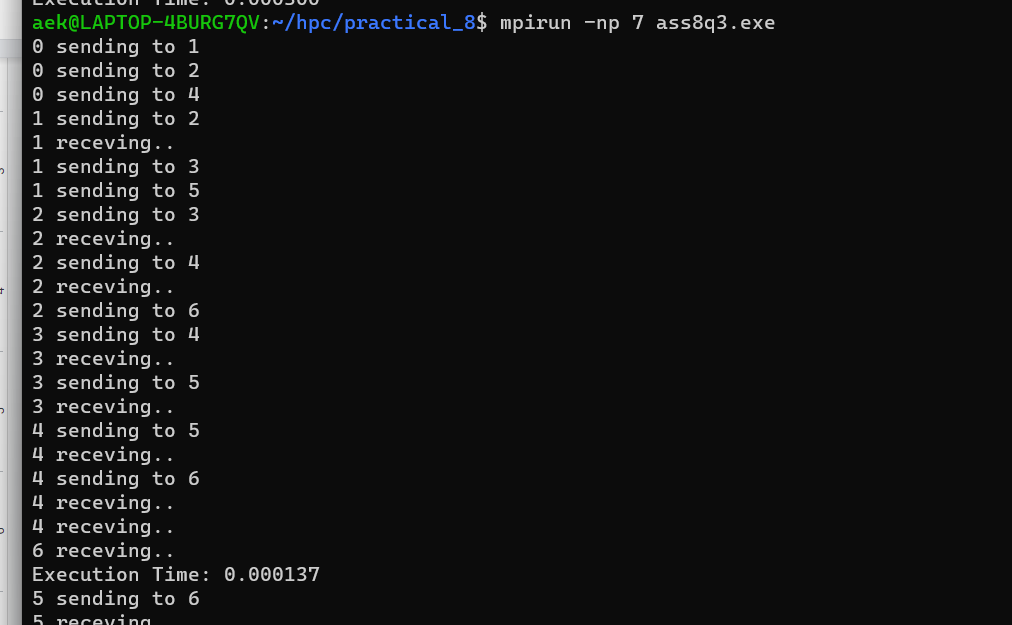
*MPI\_Finalize();*

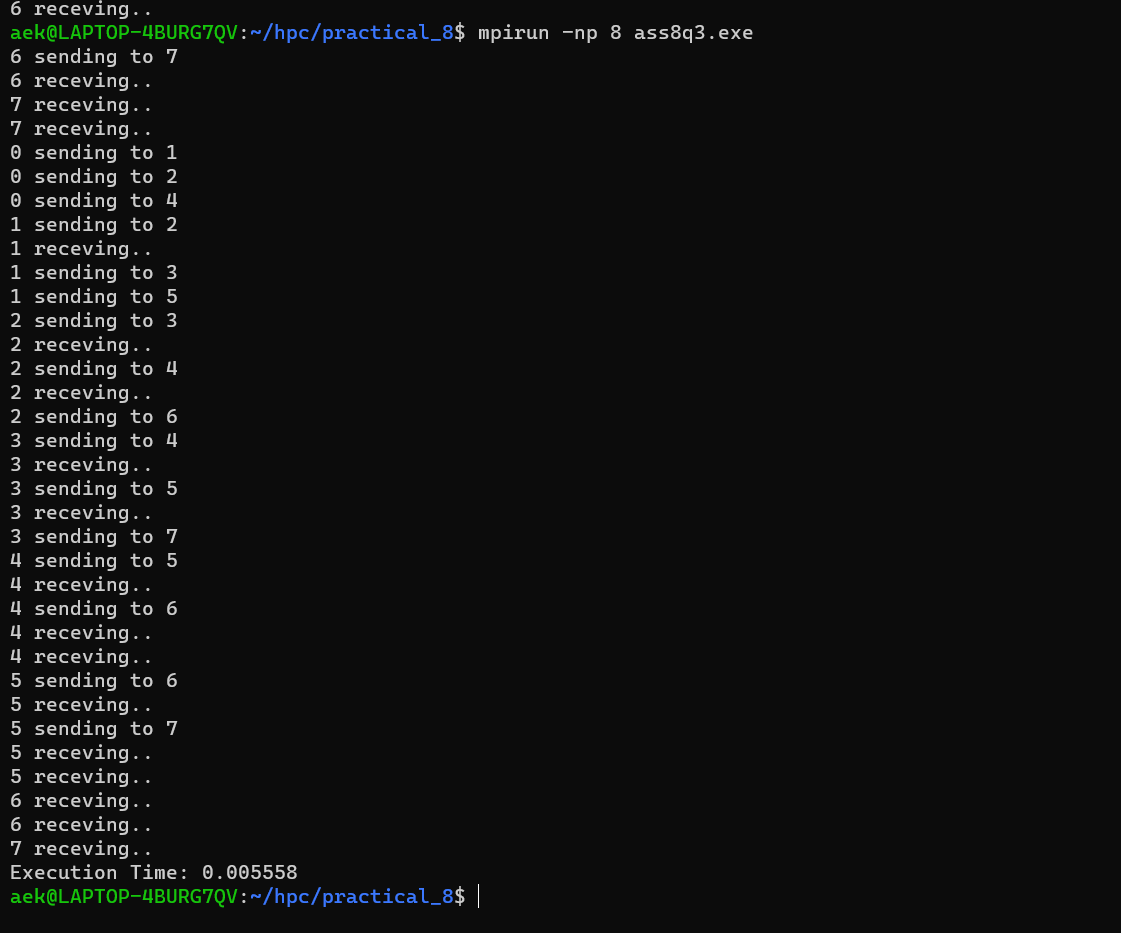
*return 0;*

*}*

**

**

**

**

| **Processors** | **Time** |
| --- | --- |
| **2** | **0.000120** |
| **3** | **0.000052** |
| **4** | **0.000086** |
| **5** | **0.000121** |
| **6** | **0.000300** |
| **7** | **0.000137** |
| **8** | **0.005558** |

**Github Link:**

[**https://github.com/OnkarGavali/HPC\_Lab/tree/main/Practical\_No8**](https://github.com/OnkarGavali/HPC_Lab/tree/main/Practical_No8)