Assignment 3

Title / Objective: MPI

Problem Statement: Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.

Course Outcome: C414454.1 Demonstrate knowledge of the core concepts and

techniques in distributed systems.

Requirements: openmpi-4.1.4.tar.bz2.

Input:

array.c

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


int main(int argc, char * argv[]) {
  int rank, size;
  int num[20]; //N=20, n=4
  MPI_Init( & argc, & argv);
  MPI_Comm_rank(MPI_COMM_WORLD, & rank);
  MPI_Comm_size(MPI_COMM_WORLD, & size);
  for (int i = 0; i < 20; i++)
    num[i] = i + 1;
  if (rank == 0) {
    int s[4];
    printf("Distribution at rank %d \n", rank);
    for (int i = 1; i < 4; i++)
      MPI_Send( & num[i * 5], 5, MPI_INT, i, 1, MPI_COMM_WORLD); //N/n i.e. 20/4=5
    int sum = 0, local_sum = 0;
    for (int i = 0; i < 5; i++) {
      local_sum = local_sum + num[i];
    }
    for (int i = 1; i < 4; i++) {
      MPI_Recv( & s[i], 1, MPI_INT, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    printf("local sum at rank %d is %d\n", rank, local_sum);
```

```
    sum = local_sum;
    for (int i = 1; i < 4; i++)
      sum = sum + s[i];
    printf("final sum = %d\n\n", sum);
  } else {
    int k[5];
    MPI_Recv(k, 5, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    int local_sum = 0;
    for (int i = 0; i < 5; i++) {
      local_sum = local_sum + k[i];
    }
    printf("local sum at rank %d is %d\n", rank, local_sum);
    MPI_Send( & local_sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

array_sum.c

```
#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>
 // size of array
#define n 10
int a[] = {  1,  2,  3,  4,  5,  6,  7,  8,  9,  10 };
// Temporary array for slave process
int a2[1000];
int main(int argc, char * argv[]) {
    int pid, np,
    elements_per_process,
```

```
n_elements_recieved;
// np -> no. of processes
// pid -> process id
MPI_Status status;
// Creation of parallel processes
MPI_Init( & argc, & argv);
// find out process ID,
// and how many processes were started
MPI_Comm_rank(MPI_COMM_WORLD, & pid);
MPI_Comm_size(MPI_COMM_WORLD, & np);
// master process
if (pid == 0) {
  int index, i;
  elements_per_process = n / np;
  // check if more than 1 processes are run
  if (np > 1) {
    // distributes the portion of array
    // to child processes to calculate
    // their partial sums
    for (i = 1; i < np - 1; i++) {
      index = i * elements_per_process;
      MPI_Send( & elements_per_process,
        1, MPI_INT, i, 0,
        MPI_COMM_WORLD);
      MPI_Send( & a[index],
        elements_per_process,
        MPI_INT, i, 0,
        MPI_COMM_WORLD);
    }
    // last process adds remaining elements
    index = i * elements_per_process;
    int elements_left = n - index;
    MPI_Send( & elements_left,
      1, MPI_INT,
```

```
      i, 0,
      MPI_COMM_WORLD);
    MPI_Send( & a[index],
      elements_left,
      MPI_INT, i, 0,
      MPI_COMM_WORLD);
  }
  // master process add its own sub array
  int sum = 0;
  for (i = 0; i < elements_per_process; i++)
    sum += a[i];
  // collects partial sums from other processes
  int tmp;
  for (i = 1; i < np; i++) {
    MPI_Recv( & tmp, 1, MPI_INT,
      MPI_ANY_SOURCE, 0,
      MPI_COMM_WORLD, &
      status);
    int sender = status.MPI_SOURCE;
    sum += tmp;
  }
  // prints the final sum of array
  printf("Sum of array is : %d\n", sum);
}
// slave processes
else {
  MPI_Recv( & n_elements_recieved,
    1, MPI_INT, 0, 0,
    MPI_COMM_WORLD, &
    status);
  // stores the received array segment
  // in local array a2
  MPI_Recv( & a2, n_elements_recieved,
    MPI_INT, 0, 0,
```

```
        MPI_COMM_WORLD, &
        status);
      // calculates its partial sum
      int partial_sum = 0;
      for (int i = 0; i < n_elements_recieved; i++)
        partial_sum += a2[i];
      // sends the partial sum to the root process
      MPI_Send( & partial_sum, 1, MPI_INT,
        0, 0, MPI_COMM_WORLD);
    }
    // cleans up all MPI state before exit of process
    MPI_Finalize();
    return 0;
}
```

Outputs: