Aston University

BIRMINGHAM UK

Module:	Introduction to Parallel Programming Techniques
Module ID:	EE4107
Student Name:	Umid Narziev
Student ID:	200234832
Assignment Number:	2
Academic Year:	2020 - 2021

Contents

System Specification

Below, show the system which was used to run and get the result of the simulation:

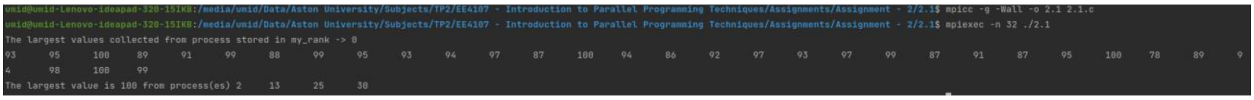
CPU:	Intel i5-7200U
Architecture:	Kaby Lake
Segment:	Mobile Processors
The number of cores:	2
Number of threads	4
Clock Frequency	2.50-3.10GHz (Turbo Boost)
Cache levels:	3
Cache level 1 size:	128KBytes
Cache level 2 size:	512Kbytes
Cache level 3 size:	3MBytes
RAM	12 GB
SSD:	250 GB
Operating System:	Ubuntu 20.04.2 LTS
Compiler:	Gcc and its libraries
IDE:	Clion (2020.03)

TASK 1

Code:

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

#define sizeOfArray 20
struct valueRecord{
    int value;
    int *index;
    int count;
};

void swap(int *xp, int *yp);
void bubbleSort(int arr[], int n);
void printArray(int arr[], int n, int my_rank);
void generateArray(int arr[], int n, int my_rank);
struct valueRecord linearSearch(int *arr, int n);

int main() {

    int my_rank, comm_sz, collection[sizeOfArray];

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    generateArray(collection, sizeOfArray, my_rank);
    bubbleSort(collection, sizeOfArray);

//    printArray(collection, sizeOfArray, my_rank); // ---------> uncomment it to print the result

    MPI_Barrier(MPI_COMM_WORLD);

    int *finalArray = NULL;
    if(my_rank == 0){
        finalArray = malloc(sizeof(int) * comm_sz);
    }
    MPI_Gather(&collection[0], 1, MPI_INT, finalArray, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    if(my_rank == 0){
        printf("The largest values collected from process stored in ");
        printArray(finalArray, comm_sz, my_rank);
        struct valueRecord record;
        record = linearSearch(finalArray, comm_sz);
        printf("The largest value is %d from process(es) ", record.value);
        for (int i = 0; i < record.count; ++i) {
            printf("%d\t", record.index[i]);
        }
        printf("\n");
    }


    MPI_Finalize();
    return 0;
}

void swap(int *xp, int *yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n){
    for (int i = 0; i < n-1; ++i) {
        for (int j = 0; j < n-i-1; ++j) {
            if (arr[j] < arr[j+1])
            swap(&arr[j], &arr[j+1]);
        }
    }
}

void printArray(int arr[], int n, int my_rank){
    printf("my_rank -> %d\n", my_rank);
    for (int i = 0; i < n; ++i) {
        printf("%d\t", arr[i]);
    }
    printf("\n");
}

void generateArray(int arr[], int n, int my_rank){
    for (int i = 0; i < n; ++i) {
        srand(my_rank+i+i*i); //may be use may_rank + timer() to generate more uniqe numbers but I want repeated larges number so I used this combination
        arr[i] = (rand() % (100 - (-100) + 1)) + (-100);
    }
}

struct valueRecord linearSearch(int *arr, int n) {
    int theLargest = arr[0];
    for (int i = 0; i < n-1; ++i) {
        if(theLargest < arr[i+1]){
            theLargest = arr[i+1];
        }
    }
    int count = 0;

    struct valueRecord rec;
    rec.value = theLargest;

    for (int i = 0; i < n; ++i) {
        if(rec.value == arr[i]){
            count++;
        }
    }

    rec.count = count;
    rec.index = malloc(sizeof(int) * count);

    int j = 0;
    for (int i = 0; i < n; ++i) {
        if(theLargest == arr[i]){
            rec.index[j] = i;
            j++;
        }
    }

    return rec;
}
```

Result:



```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 2/2.1$ mpicc -g -Wall -o 2.1 2.1.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 2/2.1$ mpiexec -n 32 ./2.1
The largest values collected from process stored in my_rank -> 0
93      95      100     89      91      99      88      99      95      93      94      97      87      100     94      86      92      97      93      97      99      87      91      87      95      100     78      89      9
4       98      100     99
The largest value is 100 from process(es) 2     13      25      30
```

Summary:

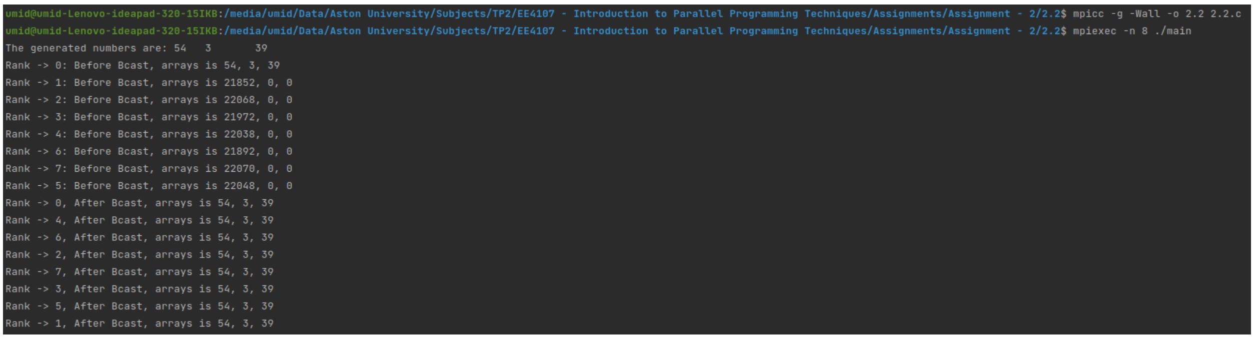
- The program designed to find the largest value among the processes. It uses linear search, and a bubble sort, to find the largest value among the processes.
- Each process generates its values, then bubble-sort sorts elements in descending order, they send the first element to processes with rank 0,
- The processes with rank – 0 find from which process it is received and prints the largests values.

TASK 2

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi/mpi.h"

void myMPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
int generateRandom(int upper, int lower);

int main(int argc, char** argv) {
   int my_rank, comm_sz;
   int numbers[3];

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   if(my_rank == 0) {
      printf("The generated numbers are: ");
      for (int i = 0; i < 3; ++i) {
         srand(rand() + time(0)); // :-)))
         numbers[i] = generateRandom(100, -100);
         printf("%d\t", numbers[i]);
      }
      printf("\n");
   }
   MPI_Barrier(MPI_COMM_WORLD) ;// used to give clarity at the output as processes were fighting for i/o buffer
   printf("Rank -> %d: Before Bcast, arrays is %d, %d, %d \n", my_rank, numbers[0], numbers[1], numbers[2]);
   MPI_Barrier(MPI_COMM_WORLD);
   myMPI_Bcast(&numbers, 3, MPI_INT, 0, MPI_COMM_WORLD);

   printf("Rank -> %d, After Bcast, arrays is %d, %d, %d \n", my_rank, numbers[0], numbers[1], numbers[2]);

   MPI_Finalize();
   return 0;
}

void myMPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm){
   int my_rank, comm_sz;
   MPI_Comm_rank(comm, &my_rank);
   MPI_Comm_size(comm, &comm_sz);

   if(my_rank == 0) {
      for (int i = 1; i < comm_sz; ++i) {
         MPI_Send(buffer, count, datatype, i, 0, comm);
      }
   }else{
      MPI_Recv(buffer, count, datatype, root, 0, comm, MPI_STATUSES_IGNORE);
   }

}

int generateRandom(int upper, int lower){

   return (rand() % (upper - lower + 1)) + lower;
}
```

Result:



```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 2/2.2$ mpicc -g -Wall -o 2.2 2.2.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 2/2.2$ mpiexec -n 8 ./main
The generated numbers are: 54   3       39
Rank -> 0: Before Bcast, arrays is 54, 3, 39
Rank -> 1: Before Bcast, arrays is 21852, 0, 0
Rank -> 2: Before Bcast, arrays is 22068, 0, 0
Rank -> 3: Before Bcast, arrays is 21972, 0, 0
Rank -> 4: Before Bcast, arrays is 22038, 0, 0
Rank -> 6: Before Bcast, arrays is 21892, 0, 0
Rank -> 7: Before Bcast, arrays is 22070, 0, 0
Rank -> 5: Before Bcast, arrays is 22048, 0, 0
Rank -> 0, After Bcast, arrays is 54, 3, 39
Rank -> 4, After Bcast, arrays is 54, 3, 39
Rank -> 6, After Bcast, arrays is 54, 3, 39
Rank -> 2, After Bcast, arrays is 54, 3, 39
Rank -> 7, After Bcast, arrays is 54, 3, 39
Rank -> 3, After Bcast, arrays is 54, 3, 39
Rank -> 5, After Bcast, arrays is 54, 3, 39
Rank -> 1, After Bcast, arrays is 54, 3, 39
```

Summary:

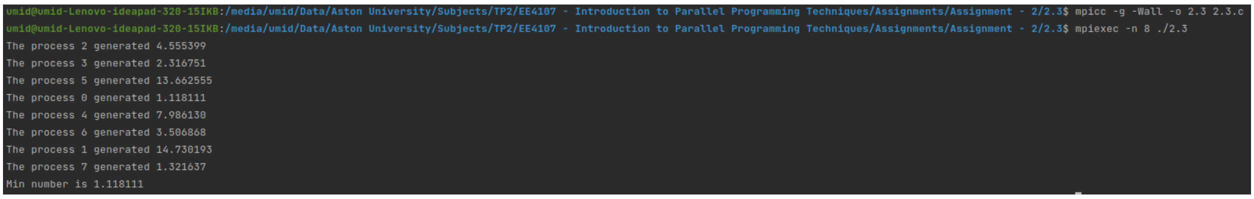
- From the screenshot, it is seen that three random numbers generated, and it is broadcasted to other processes.
- From the result, the generated buffer elements are printed. Also, buffers of each process are shown before and after broadcast.

TASK 3

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi/mpi.h"

void myMPI_Reduce(const void *sendbuf, void *recvbuf, int count, int root, MPI_Comm comm);
double pseudorand(double max);

int main() {

    int my_rank, comm_sz;
    double rec = 0;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    srand(my_rank+ time(0));
    double a = pseudorand(15.8);
    printf("The process %d generated %f\n", my_rank, a);
    myMPI_Reduce(&a, &rec, 1, 0, MPI_COMM_WORLD);
    if(my_rank == 0){
        printf("Min number is %f\n", rec);
    }


    MPI_Finalize();
    return 0;
}

void myMPI_Reduce(const void *sendbuf, void *recvbuf, int count, int root, MPI_Comm comm){

    int rank, comm_sz;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &comm_sz);


    if(rank != 0){
        MPI_Send(sendbuf, count, MPI_DOUBLE, 0, 0, comm);
    }else{
        double numbers[comm_sz];
        numbers[0] = *(double *)sendbuf;
        for (int i = 1; i < comm_sz; ++i) {
            MPI_Recv(&numbers[i], count, MPI_DOUBLE, i, 0, comm, MPI_STATUSES_IGNORE);

        }
        double min = numbers[1];
        for (int i = 0; i < comm_sz; ++i) {
            if(min >= numbers[i]){
                min = numbers[i];
            }
        }
        *(double *) recvbuf = min;
    }

}

double pseudorand(double max){
    return (max / RAND_MAX) * rand();
}
```

Result:



```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 2/2.3$ mpicc -g -Wall -o 2.3 2.3.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 2/2.3$ mpiexec -n 8 ./2.3
The process 2 generated 4.555399
The process 3 generated 2.316751
The process 5 generated 13.662555
The process 0 generated 1.118111
The process 4 generated 7.986130
The process 6 generated 3.506868
The process 1 generated 14.730193
The process 7 generated 1.321637
Min number is 1.118111
```

Summary:

- The MPI_Red with MPI_MIN function is implemented in the given, program.
- Each process generates it is own number (type of double). And with the help of MPI_Send/Recv is gathered in one process, the linear search is used to find the smallest number.