Module:	Introduction to Parallel Programming Techniques
Module ID:	EE4107
Student Name:	Umid Narziev
Student ID:	200234832
Assignment Number:	3
Academic Year:	2020 - 2021

Contents

System Specification

Below, shown 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>

#define A  //Use followings to change the mode
/*          1. A -> task A
 *          2. B -> task B
 *          3. C -> task C
 */

void printMatrix(int matrix[], int size, int rank, char title[]);
void matrixGenerate(int matrix[], int my_rank, int size);
void matrixCommunication_A(int matrixOriginal[], int matrixGenerating[], int
size, int my_rank);
void matrixCommunication_B(int matrixOriginal[], int matrixGenerating[], int
size, int my_rank, MPI_Request *requests);
void matrixCommunication_C(int matrixOriginal[], int matrixGenerating[], int
size, int my_rank, MPI_Request *requests);

int main(){

    int my_rank, comm_sz, n, size;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Request requests[comm_sz];

    if(my_rank == 0){
        if(comm_sz != 4){
            printf("4 processes is need for communication, sorry");
            MPI_Finalize();
            return -1;

        }
        printf("Please eneter the size of the matrix(n): ");
        scanf("%d", &n);
        if(n%2 != 0){
            printf("Sorry eneter multiple of two");
            MPI_Finalize();
            return -1;

        }
            size = n/2;
    }
    MPI_Bcast(&size,1, MPI_INT, 0, MPI_COMM_WORLD);
    int A_local[size*size];
    int B_local[size*size];
    int C_local[size*size];

    matrixGenerate(A_local, my_rank, size);
    printMatrix(A_local, size, my_rank, "A matrix -> ");
#ifdef A
    matrixCommunication_A(A_local, B_local, size, my_rank); // Generating ma-
trix B
#endif

#ifdef B
    matrixCommunication_B(A_local, B_local, size, my_rank, &re-
quests[my_rank]); // Generating matrix B
#endif
```

```
#ifdef C
    matrixCommunication_C(A_local, B_local, size, my_rank, &re-
quests[my_rank]); // Generating matrix B
#endif
///-------------------------------------------------------------------------
--------------------
    printMatrix(B_local, size, my_rank, "B matrix -> ");
///-------------------------------------------------------------------------
--------------------
#ifdef A
    matrixCommunication_A(B_local, C_local, size, my_rank); // Generating ma-
trix C
#endif
#ifdef B
    matrixCommunication_B(B_local, C_local, size, my_rank, &re-
quests[my_rank]); // Generating matrix C
#endif
#ifdef C
    matrixCommunication_B(B_local, C_local, size, my_rank, &re-
quests[my_rank]); // Generating matrix C
#endif

    printMatrix(C_local, size, my_rank, "C matrix -> ");

    MPI_Finalize();
    return 1;
}

void matrixGenerate(int matrix[], int my_rank, int size){
    for (int i = 0; i < size*size; ++i) {
        matrix[i] = my_rank;
    }
}

void printMatrix(int matrix[], int size, int rank, char title[]){
    MPI_Barrier(MPI_COMM_WORLD);
    printf("my_rank -> %d, %s ", rank, title);
    for (int i = 0; i < size*size; ++i) {
        printf("%d\t", matrix[i]);
    }
    printf("\n");
}

void matrixCommunication_A(int matrixOriginal[], int matrixGenerating[], int
size, int my_rank){
    if(my_rank % 2 != 0) { //odd
        for (int i = 0; i < size*size; ++i) {
            MPI_Send(&matrixOriginal[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD);
        }
        //   printf("Send -> my_rank -> %d, my_rank - 1 -> %d\n", my_rank,
my_rank - 1);
    }else{
        for (int i = 0; i < size*size; ++i) {
            MPI_Recv(&matrixGenerating[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            matrixGenerating[i] += matrixOriginal[i];
        }
        //  printf("Recv -> my_rank -> %d, my_rank + 1 -> %d\n", my_rank,
my_rank + 1);
    }

    if(my_rank % 2 == 0) { //even
        for (int i = 0; i < size*size; ++i) {
```

```
            MPI_Send(&matrixOriginal[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD);
        }
        //  printf("Send -> my_rank -> %d, my_rank + 1 -> %d\n", my_rank,
my_rank + 1);
    }else{
        for (int i = 0; i < size*size; ++i) {
            MPI_Recv(&matrixGenerating[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            matrixGenerating[i] -= matrixOriginal[i];
        }
        //  printf("Recv -> my_rank -> %d, my_rank - 1 -> %d\n", my_rank,
my_rank - 1);
    }
}

void matrixCommunication_B(int matrixOriginal[], int matrixGenerating[], int
size, int my_rank, MPI_Request *requests){
    if(my_rank % 2 != 0) { //odd

        for (int i = 0; i < size*size; ++i) {

            MPI_Isend(&matrixOriginal[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD, requests);
        }
        //   printf("Send -> my_rank -> %d, my_rank - 1 -> %d\n", my_rank,
my_rank - 1);
    }else{
        for (int i = 0; i < size*size; ++i) {
            MPI_Recv(&matrixGenerating[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            matrixGenerating[i] += matrixOriginal[i];
        }
        //  printf("Recv -> my_rank -> %d, my_rank + 1 -> %d\n", my_rank,
my_rank + 1);
    }

    if(my_rank % 2 == 0) { //even

        for (int i = 0; i < size*size; ++i) {

            MPI_Isend(&matrixOriginal[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD,  requests);
        }
        //  printf("Send -> my_rank -> %d, my_rank + 1 -> %d\n", my_rank,
my_rank + 1);
    }else{
        for (int i = 0; i < size*size; ++i) {
            MPI_Recv(&matrixGenerating[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            matrixGenerating[i] -= matrixOriginal[i];
        }
        //  printf("Recv -> my_rank -> %d, my_rank - 1 -> %d\n", my_rank,
my_rank - 1);
    }
}

void matrixCommunication_C(int matrixOriginal[], int matrixGenerating[], int
size, int my_rank, MPI_Request *requests){
    if(my_rank % 2 != 0) { //odd
        for (int i = 0; i < size*size; ++i) {
           // MPI_Barrier(MPI_COMM_WORLD); -> not used
            MPI_Send(&matrixOriginal[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD);
```

```
            MPI_Barrier(MPI_COMM_WORLD);
        }
        //   printf("Send -> my_rank -> %d, my_rank - 1 -> %d\n", my_rank,
my_rank - 1);
    }else{
        for (int i = 0; i < size*size; ++i) {
          //  MPI_Barrier(MPI_COMM_WORLD);
            MPI_Irecv(&matrixGenerating[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD, requests);
            MPI_Barrier(MPI_COMM_WORLD);
            matrixGenerating[i] += matrixOriginal[i];
        }
        //  printf("Recv -> my_rank -> %d, my_rank + 1 -> %d\n", my_rank,
my_rank + 1);
    }

    if(my_rank % 2 == 0) { //even
        for (int i = 0; i < size*size; ++i) {
            //MPI_Barrier(MPI_COMM_WORLD); -> not used
            MPI_Send(&matrixOriginal[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
        }
        //  printf("Send -> my_rank -> %d, my_rank + 1 -> %d\n", my_rank,
my_rank + 1);
    }else{
        for (int i = 0; i < size*size; ++i) {
          //  MPI_Barrier(MPI_COMM_WORLD);
            MPI_Irecv(&matrixGenerating[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD, requests);
            MPI_Barrier(MPI_COMM_WORLD);
            matrixGenerating[i] -= matrixOriginal[i];
        }
        //  printf("Recv -> my_rank -> %d, my_rank - 1 -> %d\n", my_rank,
my_rank - 1);
    }
}
```

Result:

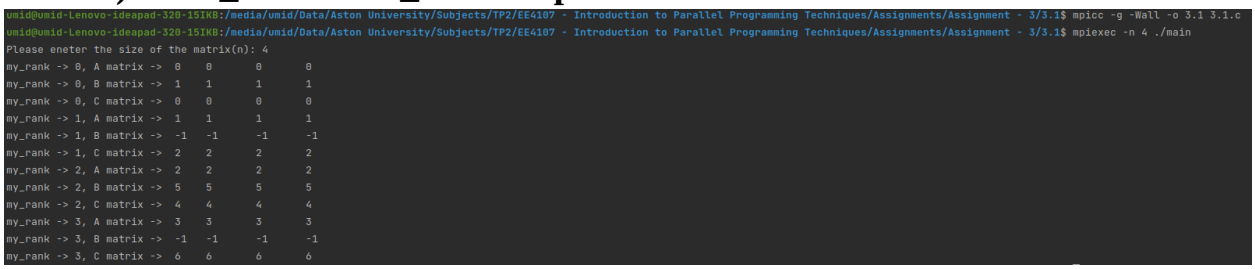
a) MPI_Send/MPI_Recv implementation

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 3/3.1$ mpiexec -n 4 ./main
Please eneter the size of the matrix(n): 4
my_rank -> 0, A matrix ->  0   0     0     0
my_rank -> 1, A matrix ->  1   1     1     1
my_rank -> 2, A matrix ->  2   2     2     2
my_rank -> 3, A matrix ->  3   3     3     3
my_rank -> 0, B matrix ->  1   1     1     1
my_rank -> 1, B matrix ->  -1  -1    -1    -1
my_rank -> 2, B matrix ->  5   5     5     5
my_rank -> 3, B matrix ->  -1  -1    -1    -1
my_rank -> 0, C matrix ->  0   0     0     0
my_rank -> 1, C matrix ->  2   2     2     2
my_rank -> 2, C matrix ->  4   4     4     4
my_rank -> 3, C matrix ->  6   6     6     6
```

b) MPI_Isend/MPI_Recv implementation

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 3/3.1$ mpicc -g -Wall -o 3.1 3.1.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 3/3.1$ mpiexec -n 4 ./main
Please eneter the size of the matrix(n): 4
my_rank -> 0, A matrix ->  0   0     0     0
my_rank -> 0, B matrix ->  1   1     1     1
my_rank -> 0, C matrix ->  0   0     0     0
my_rank -> 1, A matrix ->  1   1     1     1
my_rank -> 1, B matrix ->  -1  -1    -1    -1
my_rank -> 1, C matrix ->  2   2     2     2
my_rank -> 2, A matrix ->  2   2     2     2
my_rank -> 2, B matrix ->  5   5     5     5
my_rank -> 2, C matrix ->  4   4     4     4
my_rank -> 3, A matrix ->  3   3     3     3
my_rank -> 3, B matrix ->  -1  -1    -1    -1
my_rank -> 3, C matrix ->  6   6     6     6
```

c) MPI_Send/MPI_Irecv implementation



Summary:

- The matrix 2x2 (derived from 4x4 matrix), A = [0,1,2,3] was generated in each process, and they are computed according to the given formula.
- Communication is done with a blocking MPI_SEND/RECV and a non-blocking MPI_Isend/MPI_Irecv.
- The result is obtained shown in the screenshot, for example for rank ->0
  Matrix A = [0, 0, 0, 0]
  Matrix B = [1, 1, 1, 1]
  Matrix C = [0, 0, 0, 0]
- Other part of matrix is shown in screenshot.

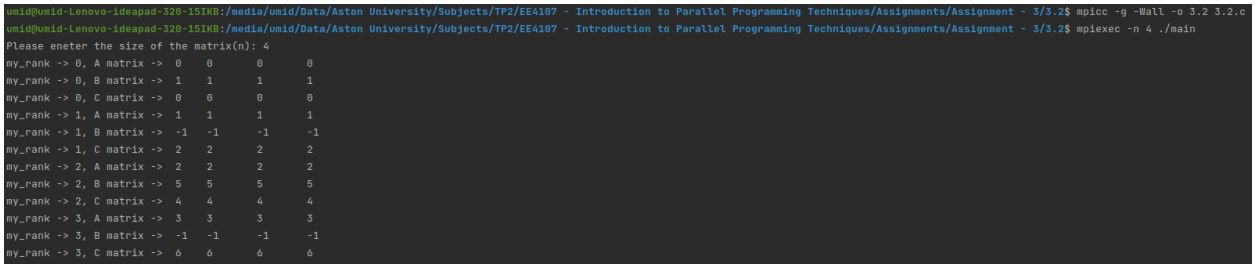
TASK 2

Code:

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


void printMatrix(int matrix[], int size, int rank, char title[]);
void matrixGenerate(int matrix[], int my_rank, int size);
void matrixCommunication(int matrixOriginal[], int vector[], int size, int
my_rank);
void computation(int matrixOriginal[], const int vector[], int size, int
my_rank);

int main(){

    int my_rank, comm_sz, n, size;

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


    if(my_rank == 0){
        if(comm_sz != 4){
            printf("4 processes is need for communication, sorry");
            MPI_Finalize();
            return -1;

        }
        printf("Please eneter the size of the matrix(n): ");
        scanf("%d", &n);
        if(n%2 != 0){
            printf("Sorry eneter multiple of two");
            MPI_Finalize();
            return -1;

        }
            size = n/2;
    }
    MPI_Bcast(&size,1, MPI_INT, 0, MPI_COMM_WORLD);
    int A_local[size*size];
    int vector[size];

    matrixGenerate(A_local, my_rank, size);
    printMatrix(A_local, size, my_rank, "A matrix -> ");

    matrixCommunication(A_local, vector, size, my_rank); // Generating matrix
B
    computation(A_local, vector, size, my_rank);
    printMatrix(A_local, size, my_rank, "B matrix -> ");

    matrixCommunication(A_local, vector, size, my_rank); // Generating matrix
C
    computation(A_local, vector, size, my_rank);
    printMatrix(A_local, size, my_rank, "C matrix -> ");



    MPI_Finalize();
    return 1;
}

void matrixGenerate(int matrix[], int my_rank, int size){
```

```
    for (int i = 0; i < size*size; ++i) {
        matrix[i] = my_rank;
    }
}

void printMatrix(int matrix[], int size, int rank, char title[]){
    MPI_Barrier(MPI_COMM_WORLD);
    printf("my_rank -> %d, %s ", rank, title);
    for (int i = 0; i < size*size; ++i) {
        printf("%d\t", matrix[i]);
    }
    printf("\n");
}

void matrixCommunication(int matrixOriginal[], int vector[], int size, int
my_rank){
    if(my_rank % 2 != 0) { //odd
        for (int i = 0; i < size*size; ++i) {
            MPI_Send(&matrixOriginal[i], 1, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD);
        }

    }else{
        for (int i = 0; i < size*size; ++i) {
            MPI_Recv(&vector[i], 1, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }

    }

    if(my_rank % 2 == 0) { //even
        for (int i = 0; i < size*size; ++i) {
            MPI_Send(&matrixOriginal[i], 1, MPI_INT, my_rank + 1, 0,
MPI_COMM_WORLD);
        }

    }else{
        for (int i = 0; i < size*size; ++i) {
            MPI_Recv(&vector[i], 1, MPI_INT, my_rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }

    }

}
void computation(int matrixOriginal[], const int vector[], int size, int
my_rank){
    MPI_Barrier(MPI_COMM_WORLD);
    if(my_rank % 2 != 0) { //odd
        for (int i = 0; i < size*size; ++i) {
            matrixOriginal[i] = vector[i] - matrixOriginal[i];
        }

    }else   if(my_rank % 2 == 0) { //even
        for (int i = 0; i < size * size; ++i) {
            matrixOriginal[i] += vector[i];
        }
    }
}
```

Result:



Summary:

- The same program is designed with one matrix and one vector for each process.
- The same result is obtained, by overusing given buffers several times.
- The result can be seen from the screenshot for 4x4 matrix size;

TASK 3

Code:

a) Blocking communication:

```
/*
* 3.3.1
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include "mpi/mpi.h"

#define DEBUG 0
#define THRESHOLD (5e-3)

void init_phi(double *phi, int n, int end, int my_rank, int comm_sz);

void update(double *cur, double *next, int n, int end, int my_rank, int
comm_sz);

int converged(double *cur, double *next, int n, int colEnd);

//#define PRINT // -> unncomment it to print the result

int main(int argc, char *argv[]) {
    int my_rank, comm_sz;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    int n, niters, conv;
    double *phi_cur, *phi_next, *tmp;
    if (my_rank == 0) {
        printf("\nEnter the n:");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    int count = n / comm_sz;
    int remainder = n % comm_sz;
    int start, stop;
    if (my_rank < remainder) {
// The first 'remainder' ranks get 'count + 1' tasks each
        start = my_rank * (count + 1);
        stop = start + count;
    } else {
// The remaining 'size - remainder' ranks get 'count' task each
        start = my_rank * count + remainder;
        stop = start + (count - 1);
    }

    int endCol = stop - start + 1;
    phi_cur = (double *) malloc(endCol * n * sizeof(double));
    phi_next = (double *) malloc(endCol * n * sizeof(double));

    init_phi(phi_cur, n, endCol, my_rank, comm_sz);
    init_phi(phi_next, n, endCol, my_rank, comm_sz);

    niters = 0;
    while (1) {
        niters++;
```

```
#if DEBUG
        printf("Iteration %d\n", niters );
        int i,j;
        for ( i = 0; i < endCol; i++ )
        {
            printf("[" );
            for ( j = 0; j < n; j++ )
                printf(" %10.6f ", phi_cur[j*endCol + i] );
            printf("]\n" );
        }
        sleep(1);
#endif
// Compute next (new) phi from current (old) phi
        update(phi_cur, phi_next, n, endCol, my_rank, comm_sz);
// If converged, we are done
        conv = converged(phi_cur, phi_next, n, endCol);

        int *rec = NULL;
        if (my_rank == 0) {
            rec = malloc(comm_sz * sizeof(int));
        }
        MPI_Gather(&conv, 1, MPI_INT, rec, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (my_rank == 0) {
            for (int j = 0; j < comm_sz; ++j) {
                conv = rec[j];
            }
        }
        MPI_Bcast(&conv, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORLD);
        if (conv)
            break;
        // Otherwise, swap pointers and continue
        tmp = phi_cur;
        phi_cur = phi_next;
        phi_next = tmp;
    }


    MPI_Barrier(MPI_COMM_WORLD);
#ifdef PRINT
    for (int i = 0; i < endCol; i++) {

        for (int j = 0; j < n; j++) {
            printf(" %10.3f ", phi_next[j * endCol + i]);
        }
        printf("\n");
    }
#endif
    free(phi_cur);
    free(phi_next);

    printf("Converged after %d iterations\n", niters);

    MPI_Finalize();
    return 0;
}

void init_phi(double *phi, int n, int end, int my_rank, int comm_sz) {

    if (my_rank == 0) {
        for (int col = 1; col < end; col++)
            for (int row = 1; row < n - 1; row++)
                phi[row * end + col] = 50.0;
```

```
        for (int col = 0; col < end; col++) {
            for (int row = 0; row < n - 1; row++) {
                phi[0 * end + col] = 100.0;
                phi[(n - 1) * end + col] = 100.0;
                phi[row * end + 0] = 100.0;
            }
        }

    } else if (my_rank == comm_sz - 1) {
        for (int col = 0; col < end - 1; col++)
            for (int row = 1; row < n - 1; row++)
                phi[row * end + col] = 50.0;
        for (int col = 0; col < end - 1; col++) {
            for (int row = 0; row < n - 1; row++) {
                phi[0 * end + col] = 100.0;
                phi[(n - 1) * end + col] = 100.0;
            }
        }
        for (int col = 0; col < end; col++) {
            for (int row = 0; row < n; row++) {
                phi[row * end + end - 1] = 0.0;
            }
        }
    } else {
        for (int col = 0; col < end; col++)
            for (int row = 1; row < n - 1; row++)
                phi[row * end + col] = 50.0;

        for (int col = 0; col < end; col++) {
            for (int row = 0; row < n - 1; row++) {
                phi[0 * end + col] = 100.0;
                phi[(n - 1) * end + col] = 100.0;
            }
        }
    }
}
void update(double *cur, double *next, int n, int colEnd, int my_rank, int
comm_sz) {

    double tempRow[n];
    double tempRowBelow[n];
    if (my_rank == 0) {
        for (int i = 0; i < n; ++i) {
            MPI_Sendrecv(&cur[i * colEnd + (colEnd - 1)], 1, MPI_DOUBLE,
my_rank + 1, 0, &tempRow[i], 1, MPI_DOUBLE,
                         my_rank + 1, 0, MPI_COMM_WORLD, MPI_STATUSES_IG-
NORE);
        }
    } else if (my_rank == comm_sz - 1) {
        for (int i = 0; i < n; ++i) {
            MPI_Sendrecv(&cur[i * colEnd + 0], 1, MPI_DOUBLE, my_rank - 1, 0,
&tempRow[i], 1, MPI_DOUBLE, my_rank - 1,
                         0, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
        }
    } else {
        for (int i = 0; i < n; ++i) {
            MPI_Sendrecv(&cur[i * colEnd + 0], 1, MPI_DOUBLE, my_rank - 1, 0,
&tempRow[i], 1, MPI_DOUBLE, my_rank - 1,
                         0, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
            MPI_Sendrecv(&cur[i * colEnd + (colEnd - 1)], 1, MPI_DOUBLE,
my_rank + 1, 0, &tempRowBelow[i], 1,
                         MPI_DOUBLE, my_rank + 1, 0, MPI_COMM_WORLD, MPI_STA-
TUSES_IGNORE);
```

```
        }
    }

    if (my_rank == 0) {
        for (int col = 1; col < colEnd; col++) {
            for (int row = 1; row < n - 1; row++) {
                if (col == colEnd - 1) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + cur[row * colEnd
+ (col - 1)] + tempRow[row] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else {
                    next[row * colEnd + col] = (cur[(row - 1) * colEnd + col]
+ cur[row * colEnd + (col - 1)] +
                                                cur[row * colEnd + (col + 1)]
+ cur[(row + 1) * colEnd + col]) / 4;
                }
            }
        }
    } else if (my_rank == comm_sz - 1) {
        for (int col = 0; col < colEnd - 1; col++) {
            for (int row = 1; row < n - 1; row++) {
                if (col == 0) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + tempRow[row] +
cur[row * colEnd + (col + 1)] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else {
                    next[row * colEnd + col] = (cur[(row - 1) * colEnd + col]
+ cur[row * colEnd + (col - 1)] +
                                                cur[row * colEnd + (col + 1)]
+ cur[(row + 1) * colEnd + col]) / 4;
                }
            }
        }
    } else {
        for (int col = 0; col < colEnd; col++) {
            for (int row = 1; row < n - 1; row++) {
                if (col == 0) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + cur[row * colEnd
+ (col + 1)] + tempRow[row] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else if (col == colEnd - 1) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + tempRow[row] +
cur[row * colEnd + (col - 1)] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else {
                    next[row * colEnd + col] = (cur[(row - 1) * colEnd + col]
+ cur[row * colEnd + (col - 1)] +
                                                cur[row * colEnd + (col + 1)]
+ cur[(row + 1) * colEnd + col]) / 4;
                }
            }
        }
    }
}
int converged(double *cur, double *next, int n, int colEnd) {
    int i, j;

    for (j = 1; j < colEnd - 1; j++)
        for (i = 1; i < n - 1; i++)
            if (fabs(next[j * n + i] - cur[j * n + i]) > THRESHOLD)
```

```
                return 0;
    return 1;
}
```

b) Non-blocking communication

```
/*
* 3.3.2
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include "mpi/mpi.h"

#define DEBUG 0
#define THRESHOLD (5e-3)
//#define PRINT -> uncomment to print the result
void init_phi(double *phi, int n, int end, int my_rank, int comm_sz);

void update(double *cur, double *next, int n, int end, int my_rank, int
comm_sz);

int converged(double *cur, double *next, int n, int colEnd);


int main(int argc, char *argv[]) {
    int my_rank, comm_sz;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    int n, niters, conv;
    double *phi_cur, *phi_next, *tmp;
    if (my_rank == 0) {
        printf("\nEnter the n:");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    int count = n / comm_sz;
    int remainder = n % comm_sz;
    int start, stop;
    if (my_rank < remainder) {
// The first 'remainder' ranks get 'count + 1' tasks each
        start = my_rank * (count + 1);
        stop = start + count;
    } else {
// The remaining 'size - remainder' ranks get 'count' task each
        start = my_rank * count + remainder;
        stop = start + (count - 1);
    }

    int endCol = stop - start + 1;
    phi_cur = (double *) malloc(endCol * n * sizeof(double));
    phi_next = (double *) malloc(endCol * n * sizeof(double));

    init_phi(phi_cur, n, endCol, my_rank, comm_sz);
    init_phi(phi_next, n, endCol, my_rank, comm_sz);

    niters = 0;
    while (1) {
        niters++;
```

```
#if DEBUG
        printf("Iteration %d\n", niters );
        int i,j;
        for ( i = 0; i < endCol; i++ )
        {
            printf("[" );
            for ( j = 0; j < n; j++ )
                printf(" %10.6f ", phi_cur[j*endCol + i] );
            printf("]\n" );
        }
        sleep(1);
#endif
// Compute next (new) phi from current (old) phi
        update(phi_cur, phi_next, n, endCol, my_rank, comm_sz);
// If converged, we are done
        conv = converged(phi_cur, phi_next, n, endCol);

        int *rec = NULL;
        if (my_rank == 0) {
            rec = malloc(comm_sz * sizeof(int));
        }
        MPI_Gather(&conv, 1, MPI_INT, rec, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (my_rank == 0) {
            for (int j = 0; j < comm_sz; ++j) {
                conv = rec[j];
            }
        }
        MPI_Bcast(&conv, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORLD);
        if (conv)
            break;
        // Otherwise, swap pointers and continue
        tmp = phi_cur;
        phi_cur = phi_next;
        phi_next = tmp;
    }


    MPI_Barrier(MPI_COMM_WORLD);
#ifdef PRINT
    for (int i = 0; i < endCol; i++) {

        for (int j = 0; j < n; j++) {
            printf(" %10.3f ", phi_next[j * endCol + i]);
        }
        printf("\n");
    }
#endif


    free(phi_cur);
    free(phi_next);

    printf("Converged after %d iterations\n", niters);

    MPI_Finalize();
    return 0;
}


void init_phi(double *phi, int n, int end, int my_rank, int comm_sz) {

    if (my_rank == 0) {
```

```
        for (int col = 1; col < end; col++)
            for (int row = 1; row < n - 1; row++)
                phi[row * end + col] = 50.0;

        for (int col = 0; col < end; col++) {
            for (int row = 0; row < n - 1; row++) {
                phi[0 * end + col] = 100.0;
                phi[(n - 1) * end + col] = 100.0;
                phi[row * end + 0] = 100.0;
            }
        }

    } else if (my_rank == comm_sz - 1) {
        for (int col = 0; col < end - 1; col++)
            for (int row = 1; row < n - 1; row++)
                phi[row * end + col] = 50.0;
        for (int col = 0; col < end - 1; col++) {
            for (int row = 0; row < n - 1; row++) {
                phi[0 * end + col] = 100.0;
                phi[(n - 1) * end + col] = 100.0;
            }
        }
        for (int col = 0; col < end; col++) {
            for (int row = 0; row < n; row++) {
                phi[row * end + end - 1] = 0.0;
            }
        }
    } else {
        for (int col = 0; col < end; col++)
            for (int row = 1; row < n - 1; row++)
                phi[row * end + col] = 50.0;

        for (int col = 0; col < end; col++) {
            for (int row = 0; row < n - 1; row++) {
                phi[0 * end + col] = 100.0;
                phi[(n - 1) * end + col] = 100.0;
            }
        }
    }

}

void update(double *cur, double *next, int n, int colEnd, int my_rank, int
comm_sz) {

    double tempRow[n];
    double tempRowBelow[n];
    MPI_Request request = MPI_REQUEST_NULL;
    if (my_rank == 0) {
        for (int i = 0; i < n; ++i) {
//            MPI_Sendrecv(&cur[i * colEnd + (colEnd - 1)], 1, MPI_DOUBLE,
my_rank + 1, 0, &tempRow[i], 1, MPI_DOUBLE,
//                         my_rank + 1, 0, MPI_COMM_WORLD, MPI_STATUSES_IG-
NORE);

            MPI_Isend(&cur[i * colEnd + (colEnd - 1)], 1, MPI_DOUBLE, my_rank
+ 1, 0, MPI_COMM_WORLD, &request);
            MPI_Irecv(&tempRow[i], 1, MPI_DOUBLE, my_rank + 1, 0,
MPI_COMM_WORLD, &request);

        }
    } else if (my_rank == comm_sz - 1) {
        for (int i = 0; i < n; ++i) {
            MPI_Isend(&cur[i * colEnd + 0], 1, MPI_DOUBLE, my_rank - 1, 0,
```

```
MPI_COMM_WORLD, &request);
            MPI_Irecv(&tempRow[i], 1, MPI_DOUBLE, my_rank - 1, 0,
MPI_COMM_WORLD, &request);

        }
    } else {
        for (int i = 0; i < n; ++i) {
            MPI_Isend(&cur[i * colEnd + 0], 1, MPI_DOUBLE, my_rank - 1, 0,
MPI_COMM_WORLD, &request);
            MPI_Irecv(&tempRow[i], 1, MPI_DOUBLE, my_rank - 1, 0,
MPI_COMM_WORLD, &request);

            MPI_Isend(&cur[i * colEnd + (colEnd - 1)], 1, MPI_DOUBLE, my_rank
+ 1, 0, MPI_COMM_WORLD, &request);
            MPI_Irecv(&tempRowBelow[i], 1, MPI_DOUBLE, my_rank + 1, 0,
MPI_COMM_WORLD, &request);

        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if (my_rank == 0) {
        for (int col = 1; col < colEnd; col++) {
            for (int row = 1; row < n - 1; row++) {
                if (col == colEnd - 1) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + cur[row * colEnd
+ (col - 1)] + tempRow[row] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else {
                    next[row * colEnd + col] = (cur[(row - 1) * colEnd + col]
+ cur[row * colEnd + (col - 1)] +
                                                cur[row * colEnd + (col + 1)]
+ cur[(row + 1) * colEnd + col]) / 4;
                }
            }
        }
    } else if (my_rank == comm_sz - 1) {
        for (int col = 0; col < colEnd - 1; col++) {
            for (int row = 1; row < n - 1; row++) {
                if (col == 0) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + tempRow[row] +
cur[row * colEnd + (col + 1)] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else {
                    next[row * colEnd + col] = (cur[(row - 1) * colEnd + col]
+ cur[row * colEnd + (col - 1)] +
                                                cur[row * colEnd + (col + 1)]
+ cur[(row + 1) * colEnd + col]) / 4;
                }
            }
        }
    } else {
        for (int col = 0; col < colEnd; col++) {
            for (int row = 1; row < n - 1; row++) {
                if (col == 0) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + cur[row * colEnd
+ (col + 1)] + tempRow[row] +
                             cur[(row + 1) * colEnd + col]) / 4;
                } else if (col == colEnd - 1) {
                    next[row * colEnd + col] =
                            (cur[(row - 1) * colEnd + col] + tempRow[row] +
cur[row * colEnd + (col - 1)] +
```

```
                                cur[(row + 1) * colEnd + col]) / 4;
                } else {
                    next[row * colEnd + col] = (cur[(row - 1) * colEnd + col]
+ cur[row * colEnd + (col - 1)] +
                                                cur[row * colEnd + (col + 1)]
+ cur[(row + 1) * colEnd + col]) / 4;
                }
            }
        }
    }
}

int converged(double *cur, double *next, int n, int colEnd) {
    int i, j;

    for (j = 1; j < colEnd - 1; j++)
        for (i = 1; i < n - 1; i++)
            if (fabs(next[j * n + i] - cur[j * n + i]) > THRESHOLD)
                return 0;
    return 1;
}
```

Result:

a) **Blocking communication**

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 3/3.3$ mpiexec -n 3 ./3.3.a

Enter the n:11
   100.000   100.000   100.000   100.000   100.000   100.000   100.000   100.000   100.000   100.000   100.000
   100.000    99.986    99.974    99.964    99.958    99.955    99.958    99.964    99.974    99.986   100.000
   100.000    99.974    99.950    99.931    99.919    99.914    99.919    99.931    99.950    99.974   100.000
   100.000    99.963    99.930    99.903    99.886    99.880    99.886    99.903    99.930    99.963   100.000
Converged after 136 iterations
   100.000    99.955    99.915    99.883    99.863    99.856    99.863    99.883    99.915    99.955   100.000
   100.000    99.952    99.908    99.873    99.851    99.843    99.851    99.873    99.908    99.952   100.000
   100.000    99.952    99.908    99.873    99.851    99.843    99.851    99.873    99.908    99.952   100.000
   100.000    99.955    99.915    99.883    99.863    99.856    99.863    99.883    99.915    99.955   100.000
Converged after 136 iterations
   100.000    82.262    73.197    69.277    67.755    67.363    67.755    69.277    73.197    82.262   100.000
   100.000    55.901    41.345    36.287    34.535    34.106    34.535    36.287    41.345    55.901   100.000
     0.000     0.000     0.000     0.000     0.000     0.000     0.000     0.000     0.000     0.000     0.000
Converged after 136 iterations
```

b) **Non-blocking communication**

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 3/3.3$ mpiexec -n 3 ./3.3.b

Enter the n:11
Converged after 136 iterations
Converged after 136 iterations
Converged after 136 iterations
```

Summary:

- The algorithm of the program is as follows:
  1. The program divides the matrix into portions.
  2. Each process sends a lower or higher portion of the matrix to neighbour processes.
  3. Then elements are stored in a separate buffer and used in the computation.
  4. This repeated till a steady-state is found.
- The result is shown on screens.
- One drawback is that parallel execution takes a bit longer iteration to find steady-state condition, it is due to the fact, MPI has it is own data type that might be changing the precision of original data. For the 11x11 matrix and p = 3 proceses, original program gives 120 iterations and my implementation gives 136.