MONASH University

Lab Session Week 5

Student ID	Student Name	Student Email
32844700	Teh Jia Xuan	jteh0015@student.monash.edu

Task 2 - Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
    int my_rank;
    int p;

    //initialise MPI
    MPI_Init(&argc, &argv);
    //get current processor
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    //get all processor
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    int val = -1;

    do{
    if(my_rank == 0)
    {
    printf("Enter a round number (> 0 ): ");
    fflush(stdout);
    scanf("%d", &val);
    }

    //broadcast to all the processors
    // Please insert one line of code here
    MPI_Bcast(&val, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Processors: %d. Received Value: %d\n", my_rank, val);
    fflush(stdout);
    }while(val > 0);

    MPI_Finalize();
    return 0;
}
```

Task 3 - Code

MONASH University

Week5Lab > C Task3.c > main(int, char **)

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

//structure definition
struct valuestruct {
    int a;
    double b;
};

int main(int argc, char** argv)
{
    //variable declarations
    struct valuestruct values;
    int myrank;
    MPI_Datatype Valuetype; //custom mpi datatype
    MPI_Datatype type[2] = { MPI_INT, MPI_DOUBLE };
    int blocklen[2] = { 1, 1};
    MPI_Aint disp[2]; //array to store offset of each member

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    //get memory addresses of value.a and b and store in disp
    MPI_Get_address(&values.a, &disp[0]);
    MPI_Get_address(&values.b, &disp[1]);
    //Make relative, reflect the offsets within the structure
    //know how many step to jump from first address to second address
    //so store the displacement
    disp[1]=disp[1]-disp[0];
    disp[0]=0;

    // Create MPI struct
    // Please insert one line of code here (Hint: MPI_Type_create_struct)
    //create custom MPI datatype
    //element in struct, specifiying number of elem of each type,
    MPI_Type_create_struct(2, blocklen, disp, type, &Valuetype);

    //commit the custom datatype
    MPI_Type_commit(&Valuetype);

    do{
    if (myrank == 0){
    printf("Enter an round number (>0) & a real number: ");
    fflush(stdout);
    scanf("%d%lf", &values.a, &values.b);
    }
    // Please insert one line of code here (Hint: MPI_Bcast)
    //broadcast the value array to other processor
    MPI_Bcast(&values, 1, Valuetype, 0, MPI_COMM_WORLD);

    printf("Rank: %d. values.a = %d. values.b = %lf\n",
    myrank, values.a, values.b);
    fflush(stdout);
    }while(values.a > 0);
    /* Clean up the type */
    MPI_Type_free(&Valuetype);
    MPI_Finalize();

    return 0;
}
```

Task 4 - Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
int main(){
    int my_rank;
    struct timespec ts = {0, 50000000L}; /* wait 0 sec and 5^8 nanosec */
    int a; double b;
    char *buffer; int buf_size, buf_size_int, buf_size_double, position = 0;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // Determine buffer size
    MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &buf_size_int);
    MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &buf_size_double);
    buf_size = buf_size_int + buf_size_double; // Increase the buffer size
    // Allocate memory to the buffer
    buffer = (char *) malloc((unsigned) buf_size);
    do{
        if(my_rank == 0){
            nanosleep (&ts, NULL);
            printf("Enter an round number (>0) & a real number: ");
            fflush(stdout);
            scanf("%d %lf", &a, &b);
            position = 0; // Reset the position in buffer
            // Pack the integer a into the buffer
            // Please insert one line of code here (Hint: a MPI_Pack function call)
            MPI_Pack(&a, 1, MPI_INT, buffer, buf_size, &position, MPI_COMM_WORLD);
            // Pack the double b into the buffer
            // Please insert one line of code here (Hint: another MPI_Pack function call)
            MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buf_size, &position, MPI_COMM_WORLD);
        }

        // Broadcast the buffer to all processes
        MPI_Bcast(buffer, buf_size, MPI_PACKED, 0, MPI_COMM_WORLD);
        position = 0; // Reset the position in buffer in each iteration
        // Unpack the integer a from the buffer
        // Please insert one line of code here (Hint: a MPI_Unpack function call)
        MPI_Unpack(buffer, buf_size, &position, &a, 1, MPI_INT, MPI_COMM_WORLD);
        // Unpack the double b from the buffer
        // Please insert one line of code here (Hint: another MPI_Unpack function call)
        MPI_Unpack(buffer, buf_size, &position, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
        printf("[Process %d] Received values: values.a = %d, values.b = %lf\n", my_rank, a, b);
        fflush(stdout);
        MPI_Barrier(MPI_COMM_WORLD);
    }while(a > 0);
    /* Clean up */
    free(buffer);
    MPI_Finalize();
    return 0;

}
```

Task 5(a) – Code which includes comments

```
student@634a63424fa5:~/project/Week5Lab$ ./Task5Serial
Calculated Pi value (Serial-AlgoI) =  3.141592654
Overall time (s): 1.548100
student@634a63424fa5:~/project/Week5Lab$
```

Task 5(b) – Code which includes comments

```
student@634a63424fa5:~/project/Week5Lab$ mpirun -np 8 a.out
Enter a N value: 100000000
Calculated Pi value (Parallel-AlgoI) =  3.141592654
Overall time (s): 0.306057
student@634a63424fa5:~/project/Week5Lab$
```

Task 5(c) – Performance analysis (serial vs parallel), explanation about the speed up and behaviour of MPI functions.

Serial code overall time = 1.5481s
Parallel code overall time = 0.306057s

Speed up = 1.5481/0.306057 = 5.058 times

The performance of parallel code is better than serial code. It is faster than the serial code approximately 5 times. The ideal speed up should be 8 as I using 8 processors. There is a difference between actual and theoretical speed up is due to:

Communication Overhead:

- In parallel code there is overhead due to the communication between processors like broadcasting the N value and reducing the sum. This communication takes time and thus reduces the actual speedup.

Load Imbalance:

- if the workload is not perfectly balanced among the processors. Some of the processors will finish their tasks earlier than the other processors and remain idle. Thus, the program needs to wait for the other processors to finish their tasks. This lead to decrease in the actual speedup