MONASH University

[Template for Lab Session submission]

Applied Session Week {Week 5}

Student ID	Student Name	Student Email
32837933	Chua Sheng Xin	schu0077@student.monash.edu

Tasks

{Answers
Task 2

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

int main(int argc, char* argv[]) {
    int my_rank;
    int p;
    int val = -1;

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

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

        // Missing line: Broadcast the value to all processes
        MPI_Bcast(&val, 1, MPI_INT, 0, MPI_COMM_WORLD);

        printf("Processor %d received value: %d\n", my_rank, val);
        fflush(stdout);

    } while (val > 0);

    MPI_Finalize();
    return 0;
}
```

Task 3

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

struct valuestruct {
    int a;
```

```
    double b;
};

int main(int argc, char** argv) {
    struct valuestruct values;
    int myrank;
    MPI_Datatype Valuetype;
    MPI_Datatype type[2] = { MPI_INT, MPI_DOUBLE };
    int blocklen[2] = { 1, 1 };
    MPI_Aint disp[2];

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

    MPI_Get_address(&values.a, &disp[0]);
    MPI_Get_address(&values.b, &disp[1]);

    // Make relative
    disp[1] = disp[1] - disp[0];
    disp[0] = 0;

    // Create MPI struct
    // Insert missing line here
    MPI_Type_create_struct(2, blocklen, disp, type, &Valuetype);

    MPI_Type_commit(&Valuetype);

    do {
        if (myrank == 0) {
            printf("Enter an integer (>0) & a double-precision value: ");
            fflush(stdout);
            scanf("%d %lf", &values.a, &values.b);
        }

        // Insert missing line here
        MPI_Bcast(&values, 2, 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

```
#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 50^7 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;

    // Allocate memory to the buffer
    buffer = (char *)malloc((unsigned)buf_size);

    do {
        if (my_rank == 0) {
            nanosleep(&ts, NULL);
            printf("Enter an integer (>0) & a double-precision value: ");
            fflush(stdout);
            scanf("%d %lf", &a, &b);
            position = 0; // Reset the position in the buffer

            // Pack the integer a into the buffer
            MPI_Pack(&a, 1, MPI_INT, buffer, buf_size, &position, MPI_COMM_WORLD);

            // Pack the double b into the buffer
            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
        MPI_Unpack(buffer, buf_size, &position, &a, 1, MPI_INT, MPI_COMM_WORLD);

        // Unpack the double b from the buffer
        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

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int my_rank, comm_sz;
    long N;
    double local_sum = 0.0, total_sum = 0.0;
    double piVal;
    struct timespec start, end;
    double time_taken;

    MPI_Init(&argc, &argv);  // Initialize the MPI environment
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  // Get the rank of the process
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  // Get the number of processes

    // The root rank (rank 0) prompts the user for the value of N
    if (my_rank == 0) {
        printf("Enter the number of intervals (N): ");
        fflush(stdout);
        scanf("%ld", &N);
    }

    // Broadcast the value of N to all processes
    MPI_Bcast(&N, 1, MPI_LONG, 0, MPI_COMM_WORLD);

    // Get the start time after broadcasting N
    if (my_rank == 0) {
        clock_gettime(CLOCK_MONOTONIC, &start);
    }

    // Each process calculates its part of the sum
    long local_n = N / comm_sz;  // Number of intervals per process
    long start_idx = my_rank * local_n;
    long end_idx = (my_rank + 1) * local_n;
```

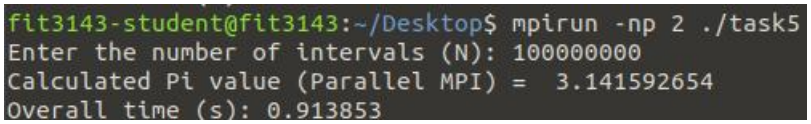
```
    for (long i = start_idx; i < end_idx; i++) {
        local_sum += 4.0 / (1 + pow((2.0 * i + 1.0) / (2.0 * N), 2));
    }

    // Reduce all local sums to a total sum in the root process
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    // The root process calculates the final value of Pi and prints the results
    if (my_rank == 0) {
        piVal = total_sum / (double)N;

        // Get the end time
        clock_gettime(CLOCK_MONOTONIC, &end);

        // Calculate the time taken in seconds
        time_taken = (end.tv_sec - start.tv_sec) * 1e9;
        time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;

        printf("Calculated Pi value (Parallel MPI) = %12.9f\n", piVal);
        printf("Overall time (s): %lf\n", time_taken);
    }

    MPI_Finalize();  // Finalize the MPI environment
    return 0;
}
```

a) The overall time taken to complete the computation is 1.828858s.
b) The overall time taken to complete the computation is 0.913853s for 100000000 as N, the number of intervals. The speed up is 2.001.
c) The observed speedup of 2.001 indicates that the parallel implementation is approximately twice as fast as the serial implementation. With only two cores in my machine, the theoretical maximum speedup is 2.0, assuming perfect parallelization and no overhead. We have observed that the speed up is actually slightly faster than expected. Achieving perfect speedup is challenging due to factors like communication overhead, load imbalance, and other conditions. Even with two cores, there is some overhead associated with setting up and managing the parallel environment. For instance, broadcasting the value of N and reducing results can add overhead, though this overhead is relatively small with only two processes. Moreover, the work should ideally be evenly distributed among all processes. However, due to factors like the way the iterations are divided or the inherent nature of the computation, some processes may finish their work earlier and remain idle while waiting for others. This imbalance leads to inefficiencies that slow down the execution. On a shared system, multiple programs or processes may compete for CPU time, memory bandwidth, or other resources. This contention can slow down the execution of the parallel program. If the parallel processes are competing for limited resources, their performance may degrade, leading to a lower speedup than expected. The fact that the observed speedup is very close to the theoretical maximum suggests that the parallel implementation is efficient. This is particularly impressive given that the program likely involves significant computation and relatively small communication overhead due to the limited number of processes.

}

{Screenshots

Task 5



}

{References
I have used ChatGPT to generate codes and answers in my work, especially in task 5.}