THE UNIVERSITY OF WESTERN AUSTRALIA

CITS5507 - HIGH PERFORMANCE COMPUTING

SEMESTER 2, 2023

Project - 2 Report

Group Members

PRITAM SUWAL SHRESTHA (23771397)

RASPREET KHANUJA (23308425)

Part 1: MPI Communication

Initial Implementation:

Overview

The provided code implements an MPI program where a master process (`rank = 0`) initializes an array of data representing fishes and distributes this data among all available processes. After the distribution, each non-master process sends it back to the master process, which then writes the received data to an output file.

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

#define DATA_SIZE 10

typedef struct {
    double x, y;
    double distanceTraveled;
    double weight;
} Fish;

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    Fish *data = NULL;
    int data_per_process = DATA_SIZE / size;
    Fish *recv_buffer = (Fish *) malloc(data_per_process * sizeof(Fish));

    // Master process generates Fish data
    if (rank == 0) {
        data = (Fish *) malloc(DATA_SIZE * sizeof(Fish));
        for (int i = 0; i < DATA_SIZE; i++) {
            data[i].x = i * 0.5;
            data[i].y = i * 0.5;
            data[i].distanceTraveled = 0.0;
            data[i].weight = 1.0;
        }

        // Output data to a file
        FILE *file = fopen("data_before.txt", "w");
        for (int i = 0; i < DATA_SIZE; i++) {
            fprintf(file, "x: %lf, y: %lf, distance: %lf, weight: %lf\n",
                    data[i].x, data[i].y, data[i].distanceTraveled, data[i].weight);
```

```
        }
        fclose(file);
    }

    // Distribute Fish data to all processes
    if (rank == 0) {
        for (int dest = 0; dest < size; dest++) {
            // Send data to other processes as well as master
            MPI_Send(data + dest * data_per_process, data_per_process * sizeof(Fish),
MPI_BYTE, dest, 0,
                     MPI_COMM_WORLD);
        }
    } else {
        // Receive data
        MPI_Recv(recv_buffer, data_per_process * sizeof(Fish), MPI_BYTE, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // Add a barrier to ensure that all processes have received their data
    MPI_Barrier(MPI_COMM_WORLD);

    // Double the received data before sending it back for debugging purpose
    if (rank != 0) {

//        for (int i = 0; i < data_per_process; i++) {
//            recv_buffer[i].x *= 2.0;
//            recv_buffer[i].y *= 2.0;
//        }

        // Send the modified data back to the master
        MPI_Send(recv_buffer, data_per_process * sizeof(Fish), MPI_BYTE, 0, 1,
MPI_COMM_WORLD);
    } else {
        // Master receives data from workers
        for (int source = 1; source < size; source++) {
            MPI_Recv(data + source * data_per_process, data_per_process * sizeof(Fish),
MPI_BYTE, source, 1,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        // Master process outputs the received Fish data to a file
        FILE *file = fopen("data_after.txt", "w");
        for (int i = 0; i < DATA_SIZE; i++) {
            fprintf(file, "x: %lf, y: %lf, distance: %lf, weight: %lf\n",
                    data[i].x, data[i].y, data[i].distanceTraveled, data[i].weight);
        }
        fclose(file);

        free(data);
    }

    free(recv_buffer);
```

```
    MPI_Finalize();

    return 0;
}
```

The distribution of data to the processes is done based on the formula: data_per_process = DATA_SIZE / size, where DATA_SIZE is the total number of data points and size is the total number of processes. However, this approach does not evenly distribute the data among processes if DATA_SIZE is not a multiple of size.

Data Distribution

Based on the output with `DATA_SIZE = 10` and `4 processes`, the distribution of data is as follows:

Data Before (data_before.txt)	**Data After** (data_after.txt)
x: 0.000000, y: 0.000000, distance: 0.000000, weight: 1.000000 -> **Process 0**	x: 0.000000, y: 0.000000, distance: 0.000000, weight: 1.000000 -> **Original**
x: 0.500000, y: 0.500000, distance: 0.000000, weight: 1.000000 -> **Process 0**	x: 0.500000, y: 0.500000, distance: 0.000000, weight: 1.000000 -> **Original**
x: 1.000000, y: 1.000000, distance: 0.000000, weight: 1.000000 -> **Process 1**	x: 1.000000, y: 1.000000, distance: 0.000000, weight: 1.000000 -> **Received** from Process 1
x: 1.500000, y: 1.500000, distance: 0.000000, weight: 1.000000 -> **Process 1**	x: 1.500000, y: 1.500000, distance: 0.000000, weight: 1.000000 -> **Received from Process 1**
x: 2.000000, y: 2.000000, distance: 0.000000, weight: 1.000000 -> **Process 2**	x: 2.000000, y: 2.000000, distance: 0.000000, weight: 1.000000 -> **Received from Process 2**
x: 2.500000, y: 2.500000, distance: 0.000000, weight: 1.000000 -> **Process 2**	x: 2.500000, y: 2.500000, distance: 0.000000, weight: 1.000000 -> **Received from Process 2**
x: 3.000000, y: 3.000000, distance: 0.000000, weight: 1.000000 -> **Process 3**	x: 3.000000, y: 3.000000, distance: 0.000000, weight: 1.000000 -> **Received from Process 3**
x: 3.500000, y: 3.500000, distance: 0.000000, weight: 1.000000 -> **Process 3**	x: 3.500000, y: 3.500000, distance: 0.000000, weight: 1.000000 -> **Received from Process 3**
x: 4.000000, y: 4.000000, distance: 0.000000, weight: 1.000000 -> **Process 0**	x: 4.000000, y: 4.000000, distance: 0.000000, weight: 1.000000 -> **Original**
x: 4.500000, y: 4.500000, distance: 0.000000, weight: 1.000000 -> **Process 0**	x: 4.500000, y: 4.500000, distance: 0.000000, weight: 1.000000 -> **Original**

- DATA_SIZE is 10.
- There are 4 processes (size = 4).
- data_per_process is thus 2.5, but since this is a double value and indices must be integers, when used in calculations, it implicitly gets converted to 2. Therefore, each process (apart from the first one where all the remaining data will be given) will handle 2 fishes.

From the provided output:

- Process 0 handles the fishes with x: 0.0, y: 0.0 and x: 0.5, y: 0.5, and the fishes with x: 4.0, y: 4.0 and x: 4.5, y: 4.5.
- Process 1 handles the fishes with x: 1.0, y: 1.0 and x: 1.5, y: 1.5

- Process 2 handles the fishes with x: 2.0, y: 2.0 and x: 2.5, y: 2.5
- Process 3 handles the fishes with x: 3.0, y: 3.0 and x: 3.5, y: 3.5

Clearly, the distribution is uneven. Process 0 handles 4 fishes, while the rest of the processes handle only 2 fishes each.

Issue

The issue arises due to the way data is sent to processes in the loop:

```
for (int dest = 0; dest < size; dest++) {
    MPI_Send(data + dest * data_per_process, ...);
}
```

Since data_per_process is implicitly 2 for all dest, it leads to Process 0 handling the first 2 fishes and the last 2 fishes.

So, we had to think of a way to distribute the data more evenly.

Second Implementation

Solution Using Offset

To solve the problem, we used an offset approach similar to the example in the lecture. We added an offset to data_per_process based on the process's rank. The idea is to distribute the remaining data points (after dividing DATA_SIZE by size) among the first few processes.

Pseudocode for this solution is as follows:

```
int remaining_data = DATA_SIZE % size;
int offset = (rank < remaining_data) ? rank : remaining_data;

MPI_Send(data + dest * data_per_process + offset, ...);
```

Building on this idea, we implemented the following solution:

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

#define NUM_FISHES 10

// Define the Fish structure
typedef struct {
    double x, y;
```

```c
    double distanceTraveled;
    double weight;
} Fish;

// Initialize the state of the lake with random fish positions
void initializeInitialLakeState(Fish *fishes, int numFishes) {
    for (int i = 0; i < numFishes; i++) {
        fishes[i].x = ((double) rand() / RAND_MAX) * 200.0 - 100.0;
        fishes[i].y = ((double) rand() / RAND_MAX) * 200.0 - 100.0;
        fishes[i].distanceTraveled = 0.0;
        fishes[i].weight = 1.0;
    }
}

// Write the fish data to a file
void writeFishDataToFile(const char *filename, Fish *fishes, int numFish) {
    FILE *file = fopen(filename, "w");
    if (file != NULL) {
        for (int i = 0; i < numFish; i++) {
            fprintf(file, "%.2f %.2f %.2f %.2f\n", fishes[i].x, fishes[i].y,
                    fishes[i].distanceTraveled, fishes[i].weight);
        }
        fclose(file);
    } else {
        fprintf(stderr, "Error opening file for writing: %s\n", filename);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
}

int main(int argc, char *argv[]) {
    // Initialize MPI environment
    MPI_Init(&argc, &argv);

    int numtasks, taskid;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

    Fish *fishes;
    int numFishInWorker, numFishToDistribute;
    int mtype;
    int offset = 0;

    // Master process
    if (taskid == 0) {
        srand(time(NULL));
        fishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
```

```
        if (fishes == NULL) {
            fprintf(stderr, "Memory allocation failed. Exiting...\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        // Initialize fish positions and save to file
        initializeInitialLakeState(fishes, NUM_FISHES);
        writeFishDataToFile("fish_data_initial.txt", fishes, NUM_FISHES);

        // Calculate how many fish data should be sent to each worker
        numFishToDistribute = NUM_FISHES / numtasks;
        int extra = NUM_FISHES % numtasks;

        // Distribute fish data to worker processes
        for (int dest = 0; dest < numtasks; dest++) {
            numFishInWorker = (dest < extra) ? numFishToDistribute + 1 :
numFishToDistribute;

            mtype = 1; // Message type indicating it's FROM_MASTER
            MPI_Send(&numFishInWorker, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(fishes + offset, numFishInWorker * sizeof(Fish), MPI_BYTE,
dest, mtype, MPI_COMM_WORLD);

            offset += numFishInWorker;
        }
    } else { // Worker processes
        mtype = 1; // Message type indicating it's FROM_MASTER
        MPI_Recv(&numFishInWorker, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        fishes = (Fish *) malloc(numFishInWorker * sizeof(Fish));
        MPI_Recv(fishes, numFishInWorker * sizeof(Fish), MPI_BYTE, 0, mtype,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Send received data back to the master
        mtype = 2; // Message type indicating it's FROM_WORKER
        MPI_Send(&numFishInWorker, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
        MPI_Send(fishes, numFishInWorker * sizeof(Fish), MPI_BYTE, 0, mtype,
MPI_COMM_WORLD);
        free(fishes);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (taskid == 0) {
        // Master process collects data from workers
        Fish *masterFishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
        offset = 0;
```

```
        for (int source = 1; source < numtasks; source++) {
            mtype = 2; // Message type indicating it's FROM_WORKER
            MPI_Recv(&numFishInWorker, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            fishes = (Fish *) malloc(numFishInWorker * sizeof(Fish));
            MPI_Recv(fishes, numFishInWorker * sizeof(Fish), MPI_BYTE, source,
mtype, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);

            char recvFileName[50];
            snprintf(recvFileName, sizeof(recvFileName),
"fish_data_received_%d.txt", source);
            writeFishDataToFile(recvFileName, fishes, numFishInWorker);

            // Merge received data into master data
            for (int i = 0; i < numFishInWorker; i++) {
                masterFishes[offset + i] = fishes[i];
            }

            free(fishes);
            offset += numFishInWorker;
        }

        // Save the final combined data
        writeFishDataToFile("fish_data_final.txt", masterFishes, NUM_FISHES);
        free(masterFishes);
    }

    MPI_Finalize();
    return 0;
}
```

For testing and debugging purposes, we create different files for the initial data, the data received by each worker, and the final data.

Overview

In the given program, there is a master process (with rank 0) that first generates some random data representing fishes in a lake. This master process then intends to distribute these fishes among different worker processes. Post distribution, the workers send the data back to the master. Finally, the master gathers all data received from workers and writes it to the final file.

Data Distribution

Based on the output with `NUM_FISHES = 10` and `4 processes`, the distribution of data is as follows:

In `fish_data_received_1.txt`:

fish_data_received_1.txt (Process 1)
62.74 54.32 0.00 1.00
-77.59 18.70 0.00 1.00
35.35 -85.62 0.00 1.00

In `fish_data_received_2.txt`:

fish_data_received_2.txt (Process 2)
-45.09 52.33 0.00 1.00
-36.99 -63.77 0.00 1.00

In `fish_data_received_3.txt`:

fish_data_received_3.txt (Process 3)
54.87 5.47 0.00 1.00
-17.11 2.62 0.00 1.00

The above output shows the fish data received by each of the respective worker processes after distribution by the master.

Output

Initial data (fish_data_initial.txt): Represents the state of the lake before distribution.

Final data sent back to process 0 (fish_data_final.txt): Shows the consolidated fish data received by the master from all workers.

Data Before (fish_data_initial.txt)	**Data Final (fish_data_final.txt)**
39.96 20.01 0.00 1.00 -> **Sent to Process 0**	62.74 54.32 0.00 1.00 -> **Received from Process 1**
-34.89 -3.86 0.00 1.00 -> **Sent to Process 0**	-77.59 18.70 0.00 1.00 -> **Received from Process 1**
-94.34 -4.60 0.00 1.00 -> **Sent to Process 0**	35.35 -85.62 0.00 1.00 -> **Received from Process 1**
62.74 54.32 0.00 1.00 -> **Sent to Process 1**	-45.09 52.33 0.00 1.00 -> **Received from Process 2**
-77.59 18.70 0.00 1.00 -> **Sent to Process 1**	-36.99 -63.77 0.00 1.00 -> **Received from Process 2**
35.35 -85.62 0.00 1.00 -> **Sent to Process 1**	54.87 5.47 0.00 1.00 -> **Received from Process 3**
-45.09 52.33 0.00 1.00 -> **Sent to Process 2**	-17.11 2.62 0.00 1.00 -> **Received from Process 3**
-36.99 -63.77 0.00 1.00 -> **Sent to Process 2**	0.00 0.00 0.00 0.00 -> **Not received from Process 0**
54.87 5.47 0.00 1.00 -> **Sent to Process 3**	0.00 0.00 0.00 0.00 -> **Not received from Process 0**
-17.11 2.62 0.00 1.00 -> **Sent to Process 3**	0.00 0.00 0.00 0.00 -> **Not received from Process 0**

With this approach:

- Process 0 is supposed to handle 3 fishes (2 fishes from the initial data and 1 fish from the remaining data):
- Process 1 is supposed to handle 3 fishes (2 fishes from the initial data and 1 fish from the remaining data):
- Process 2 is supposed to handle 2 fishes (2 fishes from the initial data and 0 fish from the remaining data):
- Process 3 is supposed to handle 2 fishes (2 fishes from the initial data and 0 fish from the remaining data):

This gives a more even distribution of the data.

Issue

However, the final data in `fish_data_final.txt` has 7 valid fish entities, but the last 3 entries are zeroes. This indicates that the data for the last 3 fishes was not collected properly, leading to discrepancies in the final results.

Reason for the Discrepancy:

The issue arises from how the master process `(process 0)` distributes and collects data. Specifically:

- When the master is distributing data, it also sends data to itself (as dest goes from 0 to numtasks-1). Given that MPI_Send is a blocking call, the master waits for a corresponding MPI_Recv to pick up this data.
- However, while the master is blocked on the MPI_Send, there isn't an immediate MPI_Recv to collect the data it sent to itself, causing a deadlock. So, to avoid that, we never received data sent by the master to itself.

Final Implementation

Solution using `MPI_Isend`:

To avoid the aforementioned deadlock situation, we finally figured out a way to use non-blocking sends, such as MPI_Isend. Here's how we did it:

1. Use MPI_Isend instead of MPI_Send when distributing fish data.
2. Use MPI_Request to manage the non-blocking send.
3. Add a corresponding MPI_Wait after the sends to ensure that they complete.

Here's a code snippet for the master process distribution using MPI_Isend:

```
MPI_Request request;
```

```
// ... previous code ...

// Distribute fish data to worker processes
for (int dest = 0; dest < numtasks; dest++) {
    numFishInWorker = (dest < extra) ? numFishToDistribute + 1 :
numFishToDistribute;

    mtype = 1; // Message type indicating it's FROM_MASTER

    // Non-blocking send
    MPI_Isend(&numFishInWorker, 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD, &request);
    MPI_Isend(fishes + offset, numFishInWorker * sizeof(Fish),
MPI_BYTE, dest, mtype, MPI_COMM_WORLD, &request);

    offset += numFishInWorker;
}

// ... remaining code ...
```

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

#define NUM_FISHES 22

typedef struct {
    double x, y;
    double distanceTraveled;
    double weight;
} Fish;

void initializeInitialLakeState(Fish *fishes, int numFishes) {
    for (int i = 0; i < numFishes; i++) {
        fishes[i].x = ((double) rand() / RAND_MAX) * 200.0 - 100.0;
        fishes[i].y = ((double) rand() / RAND_MAX) * 200.0 - 100.0;
        fishes[i].distanceTraveled = 0.0;
        fishes[i].weight = 1.0;
```

```
    }
}

void writeFishDataToFile(const char *filename, Fish *fishes, int numFish) {
    FILE *file = fopen(filename, "w");
    if (file != NULL) {
        for (int i = 0; i < numFish; i++) {
            fprintf(file, "%.2f %.2f %.2f %.2f\n", fishes[i].x, fishes[i].y,
                    fishes[i].distanceTraveled, fishes[i].weight);
        }
        fclose(file);
    } else {
        fprintf(stderr, "Error opening file for writing: %s\n", filename);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int numtasks, taskid;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

    Fish *fishes;
    int numFishInWorker, numFishToDistribute;
    int mtype;
    int offset = 0;

    // Master generates and distributes fish data
    if (taskid == 0) {
        srand(time(NULL));
        fishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));

        if (fishes == NULL) {
            fprintf(stderr, "Memory allocation failed. Exiting...\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        // Initialize the fish positions and write to an initial file
        initializeInitialLakeState(fishes, NUM_FISHES);
        writeFishDataToFile("fish_data_initial.txt", fishes, NUM_FISHES);

        // Determine how many fish to distribute to each worker
        numFishToDistribute = NUM_FISHES / numtasks;
        int extra = NUM_FISHES % numtasks;
```

```
        // Distribute fish data using non-blocking sends
        for (int dest = 0; dest < numtasks; dest++) {
            numFishInWorker = (dest < extra) ? numFishToDistribute + 1 :
numFishToDistribute;
            mtype = 1; //  // Tag for sending from master

            // Use non-blocking send
            MPI_Request request;
            MPI_Isend(&numFishInWorker, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD,
&request);
            MPI_Isend(fishes + offset, numFishInWorker * sizeof(Fish), MPI_BYTE,
dest, mtype, MPI_COMM_WORLD, &request);

            offset += numFishInWorker;
        }
    }

// All processes (master + workers) receive fish data
    mtype = 1; // // Tag for receiving from master
    MPI_Recv(&numFishInWorker, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    fishes = (Fish *) malloc(numFishInWorker * sizeof(Fish));
    MPI_Recv(fishes, numFishInWorker * sizeof(Fish), MPI_BYTE, 0, mtype,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Send the received data back to the master
    mtype = 2; // // Tag for sending to master
    MPI_Send(&numFishInWorker, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
    MPI_Send(fishes, numFishInWorker * sizeof(Fish), MPI_BYTE, 0, mtype,
MPI_COMM_WORLD);
    free(fishes);

    MPI_Barrier(MPI_COMM_WORLD); // Synchronize all processes

    // The master node receives data from all workers, including itself
    if (taskid == 0) {
        Fish *masterFishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
        offset = 0;

        for (int source = 0; source < numtasks; source++) {
            mtype = 2; // // Tag for receiving from worker
            MPI_Recv(&numFishInWorker, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            fishes = (Fish *) malloc(numFishInWorker * sizeof(Fish));
            MPI_Recv(fishes, numFishInWorker * sizeof(Fish), MPI_BYTE, source,
mtype, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
```

```
            // Write received data to separate files for each worker
            char recvFileName[50];
            snprintf(recvFileName, sizeof(recvFileName),
"fish_data_received_%d.txt", source);
            writeFishDataToFile(recvFileName, fishes, numFishInWorker);


            // Merge all received data into the master array
            for (int i = 0; i < numFishInWorker; i++) {
                masterFishes[offset + i] = fishes[i];
            }

            free(fishes);
            offset += numFishInWorker;
        }

        // Write the final merged data to a master file
        writeFishDataToFile("fish_data_final.txt", masterFishes, NUM_FISHES);
        free(masterFishes);
    }

    MPI_Finalize();
    return 0;
}
```

Overview

- The master process (taskid == 0) initializes an array of fish with random positions.
- It distributes these fish among all processes (including itself) using non-blocking send operations.
- Each process (worker and master) receives its portion of fish data.
- Each process then sends its fish data back to the master process.
- The master process receives fish data from all processes and writes them to separate files. It also consolidates all fish data into one array and writes it to a master file.

Data Distribution

Based on the output with NUM_FISHES = 10 and 4 processes, the distribution of data is as follows:

In `fish_data_received_0.txt`:

fish_data_received_0.txt (Process 0)
43.24 55.42 0.00 1.00
-89.62 95.35 0.00 1.00

fish_data_received_0.txt (Process 0)
15.47 7.92 0.00 1.00

In `fish_data_received_1.txt`:

fish_data_received_1.txt (Process 1)
-53.79 91.32 0.00 1.00
-59.97 94.20 0.00 1.00
92.62 38.19 0.00 1.00

In `fish_data_received_2.txt`:

fish_data_received_2.txt (Process 2)
30.82 -82.70 0.00 1.00
-68.64 -37.21 0.00 1.00

In `fish_data_received_3.txt`:

fish_data_received_3.txt (Process 3)
63.77 26.39 0.00 1.00
4.00 -30.59 0.00 1.00

Output

The data generated initially and the data received from all the workers is written to separate files. The final data are as follows. The data before and after distribution is shown for comparison.

fish_data_initial.txt (Process 0 - Distributed)	**fish_data_final.txt (Process 0 - Received)**
43.24 55.42 0.00 1.00	43.24 55.42 0.00 1.00
-89.62 95.35 0.00 1.00	-89.62 95.35 0.00 1.00
15.47 7.92 0.00 1.00	15.47 7.92 0.00 1.00
-53.79 91.32 0.00 1.00	-53.79 91.32 0.00 1.00
-59.97 94.20 0.00 1.00	-59.97 94.20 0.00 1.00
92.62 38.19 0.00 1.00	92.62 38.19 0.00 1.00
30.82 -82.70 0.00 1.00	30.82 -82.70 0.00 1.00
-68.64 -37.21 0.00 1.00	-68.64 -37.21 0.00 1.00
63.77 26.39 0.00 1.00	63.77 26.39 0.00 1.00
4.00 -30.59 0.00 1.00	4.00 -30.59 0.00 1.00

Conclusion

In the provided MPI program, the usage of non-blocking send operations via MPI_Isend has been implemented. This approach is pivotal for the following reasons:

1. **Avoiding Deadlocks**: Previously, using blocking sends (MPI_Send) had the potential to cause the master process to become deadlocked, especially when trying to send data to itself. With MPI_Isend, this issue is circumvented, as the master process doesn't wait indefinitely for the send operation to complete. Instead, it can proceed with other operations in the program, ensuring smooth execution and data flow.

2. **Data Integrity**: Observing the output—both the initial and final fish data—it's evident that the data remains consistent throughout the program. The initial data generated matches perfectly with the final consolidated data. This means that all the worker processes, including the master, successfully received, processed, and relayed back the fish data without any discrepancies.

3. **Effective Data Receipt by Master**: The master process being able to receive data sent to itself demonstrates the flexibility and robustness of the non-blocking send mechanism. It validates that the data distribution and collection mechanisms are functioning as expected without any potential hang-ups.

4. **Consistency and Accuracy**: The data points from the initial state and the final state are identical. This means that every process (including the master) received and then sent back the correct data. The use of non-blocking sends has not compromised the accuracy and reliability of the data transfer.

In summary, this implementation showcases the advantages of non-blocking send operations in MPI, particularly in scenarios where processes might need to communicate data to themselves. The seamless execution and data consistency validate the efficacy of this approach. By employing MPI_Isend, the program has effectively mitigated the risks associated with deadlocks, ensuring the accurate and efficient flow of data across all involved processes.

Part 2: MPI and OpenMP Simulation

We implemented a Fish simulation using two approaches. In the first approach, we used point to point communication to exchange fish data while on the second approach we used collective communication to distribute the fish data.

First Approach:

Overview

In the first approach (Source Code - below) , both MPI (Message Passing Interface) and OpenMP are utilized to parallelize the simulation of fish behavior. MPI is employed for point-to-point communication to exchange fish data among different processes (MPI ranks), facilitating the distribution of computational tasks and enhancing parallelism. The initial phase involves the root process (rank 0) initializing a group of fish with random attributes. Subsequently, these fish are distributed among MPI processes, ensuring that each process manages the simulation of the movement for a specific subset of fish.

OpenMP is integrated into the simulation steps to further exploit parallelism within each MPI process. Specifically, the simulationStep function, responsible for updating fish positions and attributes, is parallelized using OpenMP directives (#pragma omp parallel for). This allows concurrent execution of fish movement calculations, leveraging the computational resources available on each MPI process.

By combining MPI for inter-process communication and distribution of work and OpenMP for intra-process parallelism, the program achieves efficient parallelization at both levels. MPI handles the communication and coordination between processes, ensuring seamless data exchange, while OpenMP maximizes computational efficiency within each process.

Implementation using point to point communication for fish distribution:

```
#include <math.h>
#include <omp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <string.h>
#include <stddef.h>

#define LAKE_X_MIN (-100.0)
#define LAKE_X_MAX 100.0
#define LAKE_Y_MIN (-100.0)
#define LAKE_Y_MAX 100.0

#define MIN_SWIM_DISTANCE (-0.1)
#define MAX_SWIM_DISTANCE 0.1

#define MAX_FISH_WEIGHT 2.0
#define MIN_FISH_WEIGHT 0.0

#define NUM_FISHES 5
#define NUM_SIMULATION_STEPS 1

double square(double num) {
```

```
return num * num;
}

typedef struct {
double x, y;
double distanceTraveled;
double weight;
} Fish;

bool isFishOutsideLake(Fish *fish) {
return (fish->x < LAKE_X_MIN || fish->x > LAKE_X_MAX ||
fish->y < LAKE_Y_MIN || fish->y > LAKE_Y_MAX);
}

double getRandomNumberInRange(double minValue, double maxValue) {
double randomDouble = ((double) rand() / RAND_MAX);
double randomNumber = randomDouble * (maxValue - minValue) + minValue;
return randomNumber;
}

double calculateDistanceFromOrigin(double x, double y) {
return sqrt(square(x) + square(y));
}

double calculateObjectiveFunction(Fish *fishes, int numFishes) {
double sum = 0.0;

#pragma omp parallel for reduction(+ : sum)
for (int i = 0; i < numFishes; i++) {
sum += calculateDistanceFromOrigin((fishes + i)->x, (fishes + i)->y);
}
printf("Objective Function: %.2f\n", sum);
return sum;
}

void swim(Fish *fish) {
if (!isFishOutsideLake(fish)) {
fish->x += getRandomNumberInRange(MIN_SWIM_DISTANCE, MAX_SWIM_DISTANCE);
fish->y += getRandomNumberInRange(MIN_SWIM_DISTANCE, MAX_SWIM_DISTANCE);
}
}

void updateWeight(Fish *fish, double maxDistanceTraveledInFishSchool) {
double weightChange = fish->distanceTraveled / maxDistanceTraveledInFishSchool;
fish->weight += weightChange;
if (fish->weight < MIN_FISH_WEIGHT) {
fish->weight = MIN_FISH_WEIGHT;
} else if (fish->weight > MAX_FISH_WEIGHT) {
fish->weight = MAX_FISH_WEIGHT;
}
}
```

```
void simulationStep(Fish *fishes, int numFishes) {


double maxDistanceTraveledInFishSchool = 0.0;

#pragma omp parallel for reduction(max : maxDistanceTraveledInFishSchool)
for (int i = 0; i < numFishes; i++) {
double prevDistance = calculateDistanceFromOrigin((fishes + i)->x, (fishes + i)->y);
swim(fishes + i);
double nextDistance = calculateDistanceFromOrigin((fishes + i)->x, (fishes + i)->y);
(fishes + i)->distanceTraveled = nextDistance - prevDistance;
if (fabs((fishes + i)->distanceTraveled) > maxDistanceTraveledInFishSchool) {
maxDistanceTraveledInFishSchool = fabs((fishes + i)->distanceTraveled);
}
}

#pragma omp parallel for
for (int i = 0; i < numFishes; i++) {
updateWeight((fishes + i), maxDistanceTraveledInFishSchool);
printf(
"Step Fish %d: x = %.2f, y = %.2f, distanceTraveled = %.2f, weight = %.2f\n",
i, (fishes + i)->x, (fishes + i)->y, (fishes + i)->distanceTraveled,
(fishes + i)->weight
);
}
}

void calculateBarycenter(Fish *fishes, int numFishes) {
double weightSum = 0.0;
double distanceSum = 0.0;

#pragma omp parallel for reduction(+ : weightSum, distanceSum)
for (int i = 0; i < numFishes; i++) {
weightSum += (fishes + i)->weight * calculateDistanceFromOrigin((fishes + i)->x, (fishes +
i)->y);
distanceSum += calculateDistanceFromOrigin((fishes + i)->x, (fishes + i)->y);
}

if (distanceSum == 0.0) {
printf("Distance sum is zero. Cannot calculate barycenter.\n");
return;
}

double barycenter = weightSum / distanceSum;
printf("Barycenter: %.2f\n", barycenter);
}

void initializeInitialLakeState(Fish *fishes, int numFishes) {
#pragma omp parallel for
for (int i = 0; i < numFishes; i++) {
(fishes + i)->x = getRandomNumberInRange(LAKE_X_MIN, LAKE_X_MAX);
(fishes + i)->y = getRandomNumberInRange(LAKE_Y_MIN, LAKE_Y_MAX);
```

```
(fishes + i)->distanceTraveled = 0.0;
(fishes + i)->weight = 1.0;
printf(
"Initial Fish %d: x = %.2f, y = %.2f, distanceTraveled = %.2f, weight = %.2f\n",
i, (fishes + i)->x, (fishes + i)->y, (fishes + i)->distanceTraveled,
(fishes + i)->weight
);
}
}

int main() {


int provided;
MPI_Init_thread(NULL, NULL, MPI_THREAD_FUNNELED, &provided);


double start_time, end_time, elapsed_time;
start_time = MPI_Wtime(); // Start recording the elapsed time

MPI_Datatype MPI_FISH_TYPE;

// Create MPI type for Fish struct
int blocklengths[4] = {1, 1, 1, 1}; // One block for each struct element
MPI_Datatype types[4] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};

MPI_Aint offsets[4];
offsets[0] = offsetof(Fish, x);
offsets[1] = offsetof(Fish, y);
offsets[2] = offsetof(Fish, distanceTraveled);
offsets[3] = offsetof(Fish, weight);

MPI_Type_create_struct(4, blocklengths, offsets, types, &MPI_FISH_TYPE);
MPI_Type_commit(&MPI_FISH_TYPE);

if (provided != MPI_THREAD_FUNNELED) {
fprintf(stderr, "MPI does not support MPI_THREAD_FUNNEL mode.\n");
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

srand(time(NULL) + rank);

int fishPerProcess = NUM_FISHES / size;
int remainingFishes = NUM_FISHES % size;
int localFishCount = (rank < remainingFishes) ? fishPerProcess + 1 : fishPerProcess;

Fish *local_fishes = (Fish *) malloc(localFishCount * sizeof(Fish));
if (!local_fishes) {
```

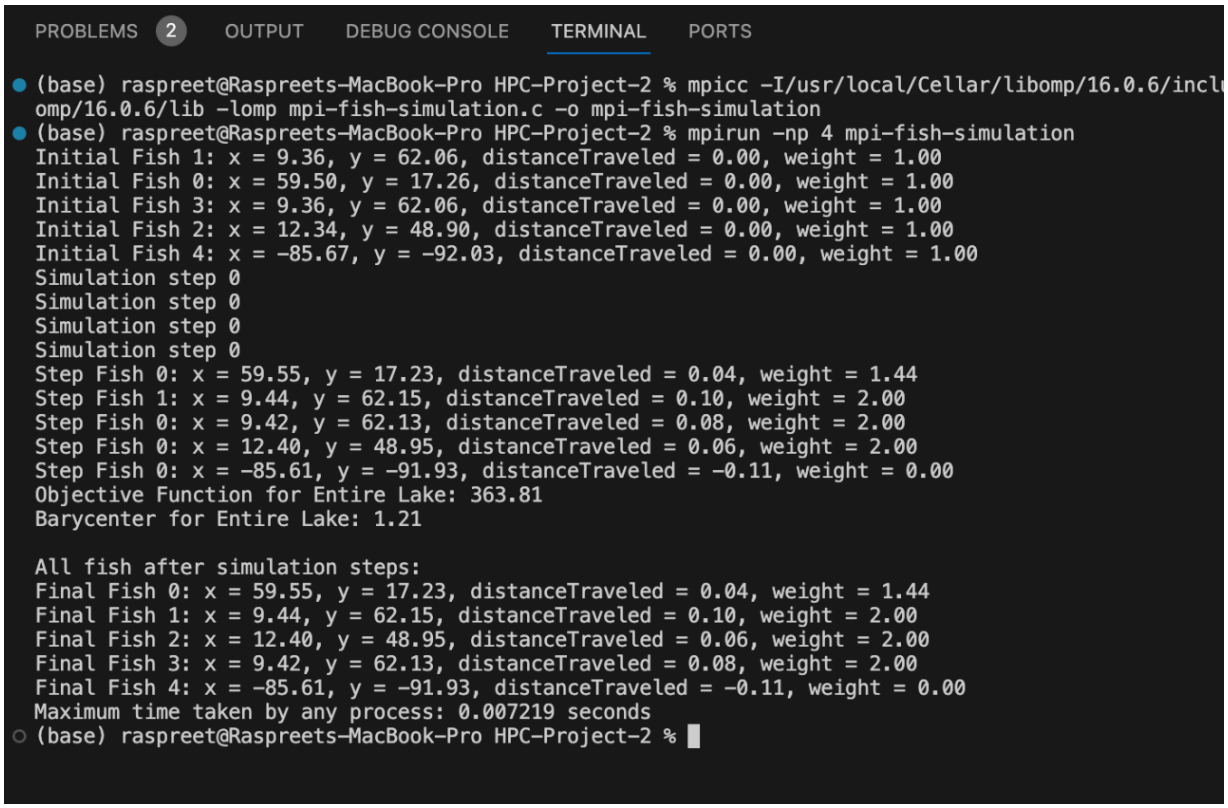
```
perror("Memory allocation failed for local_fishes");
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}

if (rank == 0) {
Fish *all_fishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
if (!all_fishes) {
perror("Memory allocation failed for all_fishes");
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}

initializeInitialLakeState(all_fishes, NUM_FISHES);

int offset = 0;
for (int i = 0; i < size; i++) {
int fishToSend = (i < remainingFishes) ? fishPerProcess + 1 : fishPerProcess;
if (i == 0) {
memcpy(local_fishes, all_fishes, fishToSend * sizeof(Fish));
} else {
MPI_Send(all_fishes + offset, fishToSend, MPI_FISH_TYPE, i, 0, MPI_COMM_WORLD);
}
offset += fishToSend;
}
free(all_fishes);
} else {
MPI_Recv(local_fishes, localFishCount, MPI_FISH_TYPE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}

for (int i = 0; i < NUM_SIMULATION_STEPS; i++) {

printf("Simulation step %d\n", i);
simulationStep(local_fishes, localFishCount);

// Objective Function for the Entire Lake
double localSum = 0.0;
double globalSum = 0.0;

#pragma omp parallel for reduction(+ : localSum)
for (int i = 0; i < localFishCount; i++) {
localSum += calculateDistanceFromOrigin(local_fishes[i].x, local_fishes[i].y);
}

MPI_Allreduce(&localSum, &globalSum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

if (rank == 0) {
printf("Objective Function for Entire Lake: %.2f\n", globalSum);
}

// Barycenter for the Entire Lake
double localWeightSum = 0.0;
double localDistanceSum = 0.0;
```

```
double globalWeightSum = 0.0;
double globalDistanceSum = 0.0;

#pragma omp parallel for reduction(+ : localWeightSum, localDistanceSum)
for (int i = 0; i < localFishCount; i++) {
localWeightSum +=
local_fishes[i].weight * calculateDistanceFromOrigin(local_fishes[i].x, local_fishes[i].y);
localDistanceSum += calculateDistanceFromOrigin(local_fishes[i].x, local_fishes[i].y);
}

MPI_Allreduce(&localWeightSum, &globalWeightSum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&localDistanceSum, &globalDistanceSum, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

if (globalDistanceSum == 0.0) {
if (rank == 0) {
printf("Distance sum is zero. Cannot calculate barycenter.\n");
}
} else {
double barycenter = globalWeightSum / globalDistanceSum;
if (rank == 0) {
printf("Barycenter for Entire Lake: %.2f\n", barycenter);
}
}

}

if (rank != 0) {
MPI_Send(local_fishes, localFishCount, MPI_FISH_TYPE, 0, 0, MPI_COMM_WORLD);
} else {
Fish *all_fishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
if (!all_fishes) {
perror("Memory allocation failed for all_fishes");
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}

int offset = 0;
for (int i = 0; i < size; i++) {
int fishToReceive = (i < remainingFishes) ? fishPerProcess + 1 : fishPerProcess;
if (i == 0) {
memcpy(all_fishes, local_fishes, fishToReceive * sizeof(Fish));
} else {
MPI_Recv(all_fishes + offset, fishToReceive, MPI_FISH_TYPE, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
offset += fishToReceive;
}

printf("\nAll fish after simulation steps:\n");
for (int i = 0; i < NUM_FISHES; i++) {
printf(
"Final Fish %d: x = %.2f, y = %.2f, distanceTraveled = %.2f, weight = %.2f\n",
```

```
i, (all_fishes + i)->x, (all_fishes + i)->y, (all_fishes + i)->distanceTraveled,
(all_fishes + i)->weight
);
}

free(all_fishes);
}

free(local_fishes);
MPI_Type_free(&MPI_FISH_TYPE);

end_time = MPI_Wtime();
elapsed_time = end_time - start_time;

double max_elapsed_time;
MPI_Reduce(&elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

if (rank == 0) {
printf("Maximum time taken by any process: %f seconds\n", max_elapsed_time);
}

MPI_Finalize();
return 0;
}
```

Output (From Local Machine)

On executing the above code in our local machine, we get the following output:



The output illustrates the execution of an MPI-OpenMP hybrid fish simulation program with 4 MPI processes (-np 4). Here's an analysis of the roles played by OpenMP and MPI:

- **Initialization (Initial Fish Positions):**

  The program initiates five fish with random positions, zero distance traveled, and an initial weight of 1.0. MPI is involved in distributing this initial state among different processes, ensuring that each process manages a subset of fish.

- **Simulation Steps (OpenMP Parallelization):**

  The simulation progresses through step 0 for each MPI process, and OpenMP parallelization is employed to enhance the efficiency of fish movement calculations.

  The swim function updates each fish's position in parallel, and the updateWeight function adjusts their weight accordingly.

  OpenMP parallel constructs are applied to parallelize these tasks, optimizing the simulation's performance within each MPI process.

- **Objective Function and Barycenter Calculation (MPI Collective Operations):**

MPI plays a crucial role in calculating the objective function and barycenter for the entire lake. MPI collective operations, specifically MPI_Allreduce, are utilized to efficiently aggregate information across all MPI processes.

The objective function represents the sum of distances of all fish from the origin, and the barycenter is computed as a weighted average position of all fish.

- **Final Fish Positions (MPI Communication):**

  After completing simulation steps, the final state of the fish is printed. In the case of MPI processes other than rank 0, the local fish state is communicated back to the root process (rank 0) using MPI point-to-point communication (MPI_Send and MPI_Recv).

- **Timing Information:**

  The program reports the maximum time taken by any MPI process. This timing information aids in evaluating the efficiency of the MPI-OpenMP parallelization strategy.

- **Overall Summary:**

  OpenMP optimizes the parallelizable tasks within each MPI process, and MPI manages the distribution, coordination, and aggregation of fish data among processes.

Output (From Setonix)

We removed the logs (printf statements) and only included maximum time for experimenting with different combination of number of process and threads while running on setonix keeping following constant.

```
NUM_FISHES: 100,000

NUM_SIMULATION_STEPS: 1000
```

```
rkhanuja@setonix-01:~> cat mpi-simulation.c
#include <math.h>
#include <omp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <string.h>
#include <stddef.h>

#define LAKE_X_MIN (-100.0)
#define LAKE_X_MAX 100.0
#define LAKE_Y_MIN (-100.0)
#define LAKE_Y_MAX 100.0

#define MIN_SWIM_DISTANCE (-0.1)
#define MAX_SWIM_DISTANCE 0.1

#define MAX_FISH_WEIGHT 2.0
#define MIN_FISH_WEIGHT 0.0

#define NUM_FISHES 100000
#define NUM_SIMULATION_STEPS 1000
```

Scenario 1: NUM_NODES = 2

NUM_NODES = 2	
NUM_THREADS	EXECUTION_TIME (seconds)
2	4.704057
4	4.756785
8	4.861421
16	5.069563

Analysis for NUM_NODES = 2:

As the number of threads increases from 2 to 16, the execution time slightly increases. The execution time generally tends to increase for all configurations. This could be due to increased overhead and contention for resources as more threads are utilized.

Scenario 2: NUM_NODES = 3

NUM_NODES = 3	
NUM_THREADS	EXECUTION_TIME (seconds)
2	3.179377
4	3.219823
8	3.317687
16	3.521453

Analysis for NUM_NODES = 3:

Similar to NUM_NODES = 2, as the number of threads increases, the execution time increases. Again, this may be due to increased overhead. The benefit of parallelism seems limited, and there might be other factors influencing performance.

Scenario 3: NUM_NODES = 4

NUM_NODES = 4	
NUM_THREADS	EXECUTION_TIME (seconds)
2	2.390389
4	2.442288
8	2.542642
16	2.741692

Analysis for NUM_NODES = 4:

The trend of slightly increased execution time with higher thread counts continues, but the overall execution times are lower compared to the 2 and 3-node scenarios. This indicates improved scalability with more nodes.

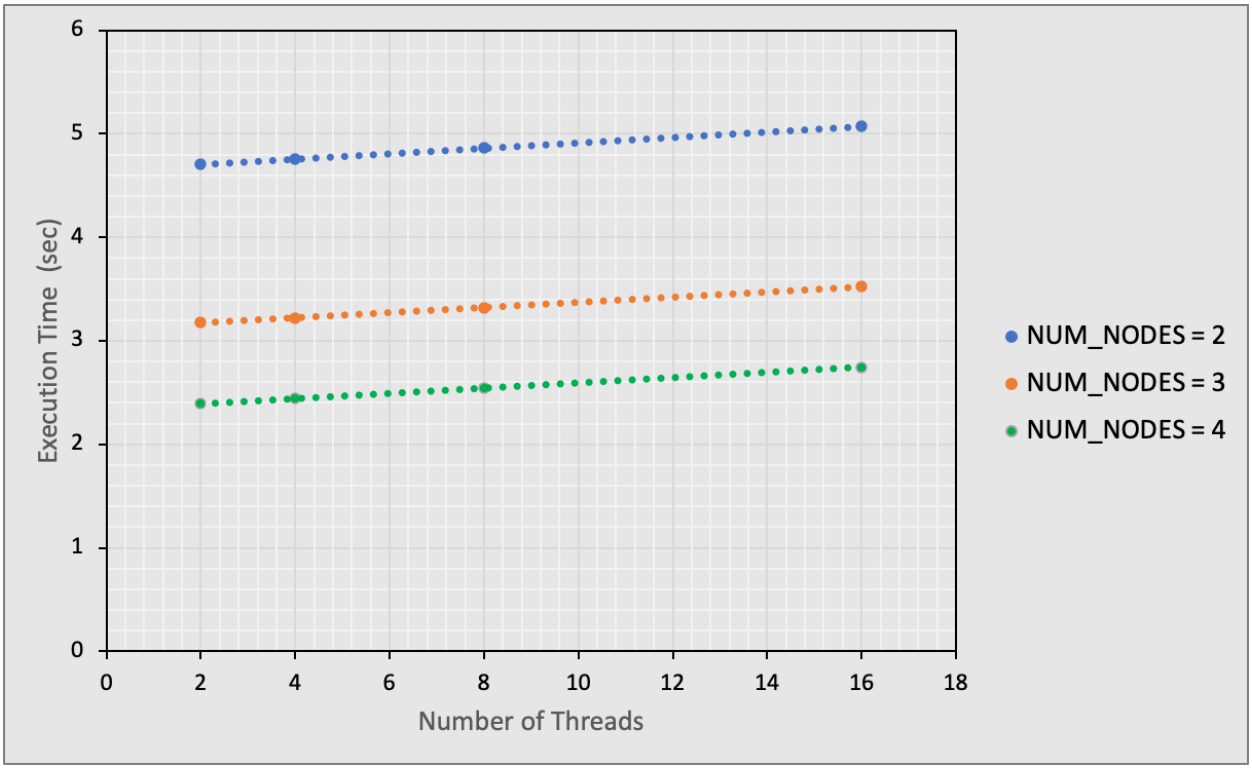


Fig: Execution Time vs. Number of Threads for Different Node Configurations

The graph above illustrate the execution time of the program with different number of nodes and threads. As, the number of process is increased (maximum limit of 4 in setonix), the execution time is decreased. However, as the number of threads were increased, the performance dropped.

Overall Observations:

- **Thread Impact:**

  Increasing the number of threads doesn't necessarily lead to better performance, and there might be a point where the overhead of managing additional threads outweighs the benefits. This implies that the workload might not be effectively parallelized with higher thread counts.

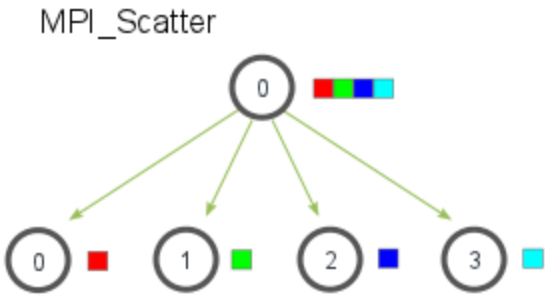
- **Node Scaling:**

  The scenario with 4 nodes consistently demonstrates better performance compared to 2 and 3 nodes. This suggests that scalability improves with an increased number of nodes, potentially due to better workload distribution.

Second Approach

We used collective communication to distribute the fish data on the second approach using MPI_Scatterv and MPI_Gatherv.

MPI_Scatter:



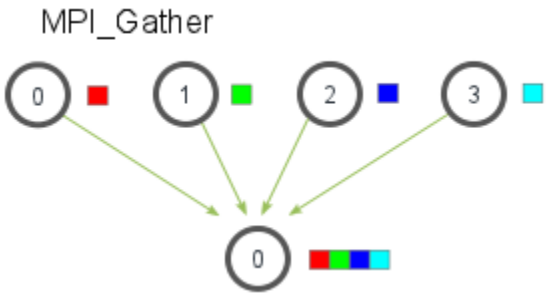
MPI_Scatter is used to distribute an array of data from one process (usually the root or master process) to all processes in a communicator (including itself).

For example, if there are `n` processes and an array of `n` elements, then the root process will send the i-th element of the array to the i-th process. If the array has `n*m` elements, then the root sends m elements to each process.

Caveats:

- **Equal Distribution:** Ensure that the data is divided equally (or can be divided equally) among the processes. For example, if you're scattering an array of 100 elements to 3 processes, it can't be evenly divided, which can result in unpredictable behavior.
- **Memory Allocation:** All processes, including the root, should allocate memory for the data they will receive. If not, there might be segmentation faults.
- **Buffer Overflows**: The send buffer should be large enough to accommodate the full data, and the receive buffer should be large enough for the chunk of data it's supposed to receive.

MPI_Gather:



MPI_Gather is the reverse of MPI_Scatter. It is used to collect data from all processes in a communicator and gather it back to one process (usually the root or master).

Each process sends a chunk of data to the root process, which then collects (or gathers) these chunks in order.

Caveats:

- **Receive Buffer Size**: The root process should allocate a receive buffer large enough to hold the combined data from all processes. If not, it might result in buffer overflows.
- **Mismatched Data:** Ensure that the data being sent from all processes is consistent in terms of type and count. Any mismatch can lead to unpredictable behavior or program crashes.
- **Ensuring Synchronization:** Before gathering, ensure that all processes have reached the gather point. Using barriers before gather (if necessary) can help synchronize processes.

In the provided code:

We used MPI_Scatterv and MPI_Gatherv (the variable versions of Scatter and Gather). The 'v' indicates that different processes might send or receive different amounts of data. In these versions, we can specify an array of counts and displacements to determine how much data each process sends or receives and where in the send or receive buffer this data should be placed.

When using MPI_Scatterv and MPI_Gatherv, additional caveats include:

- **Sendcounts and Displacements**: Ensure that the sendcounts and displs arrays are correctly initialized. Any discrepancies can lead to erroneous behavior.
- **Handling Remainders:** In scenarios where data can't be evenly divided among processes, care must be taken to ensure the remainder is handled correctly, as shown in the provided code where the remainder from dividing NUM_FISHES by the number of processes is considered.

Code

```c
#include <math.h>
#include <omp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <string.h>

#define LAKE_X_MIN (-100.0)
#define LAKE_X_MAX 100.0
#define LAKE_Y_MIN (-100.0)
#define LAKE_Y_MAX 100.0

#define MIN_SWIM_DISTANCE (-0.1)
#define MAX_SWIM_DISTANCE 0.1

#define MAX_FISH_WEIGHT 2.0
#define MIN_FISH_WEIGHT 0.0

#define NUM_FISHES 5
#define NUM_SIMULATION_STEPS 1

double square(double num) {
   return num * num;
}

typedef struct {
   double x, y;
   double distanceTraveled;
   double weight;
} Fish;

bool isFishOutsideLake(Fish *fish) {
   return (fish->x < LAKE_X_MIN || fish->x > LAKE_X_MAX ||
           fish->y < LAKE_Y_MIN || fish->y > LAKE_Y_MAX);
}

double getRandomNumberInRange(double minValue, double maxValue) {
   double randomDouble = ((double) rand() / RAND_MAX);
   double randomNumber = randomDouble * (maxValue - minValue) + minValue;
   return randomNumber;
```

```
}

double calculateDistanceFromOrigin(double x, double y) {
   return sqrt(square(x) + square(y));
}

double calculateObjectiveFunction(Fish *fishes, int numFishes) {
   double sum = 0.0;

#pragma omp parallel for reduction(+ : sum)
   for (int i = 0; i < numFishes; i++) {
       sum += calculateDistanceFromOrigin((fishes + i)->x, (fishes + i)->y);
   }
   printf("Objective Function: %.2f\n", sum);
   return sum;
}

void swim(Fish *fish) {
   if (!isFishOutsideLake(fish)) {
       fish->x += getRandomNumberInRange(MIN_SWIM_DISTANCE, MAX_SWIM_DISTANCE);
       fish->y += getRandomNumberInRange(MIN_SWIM_DISTANCE, MAX_SWIM_DISTANCE);
   }
}

void updateWeight(Fish *fish, double maxDistanceTraveledInFishSchool) {
   double weightChange = fish->distanceTraveled / maxDistanceTraveledInFishSchool;
   fish->weight += weightChange;
   if (fish->weight < MIN_FISH_WEIGHT) {
       fish->weight = MIN_FISH_WEIGHT;
   } else if (fish->weight > MAX_FISH_WEIGHT) {
       fish->weight = MAX_FISH_WEIGHT;
   }
}

void simulationStep(Fish *fishes, int numFishes) {


   double maxDistanceTraveledInFishSchool = 0.0;

#pragma omp parallel for reduction(max : maxDistanceTraveledInFishSchool)
   for (int i = 0; i < numFishes; i++) {
       double prevDistance = calculateDistanceFromOrigin((fishes + i)->x, (fishes +
i)->y);
       swim(fishes + i);
       double nextDistance = calculateDistanceFromOrigin((fishes + i)->x, (fishes +
i)->y);
       (fishes + i)->distanceTraveled = nextDistance - prevDistance;
```

```
        if (fabs((fishes + i)->distanceTraveled) > maxDistanceTraveledInFishSchool)
{
            maxDistanceTraveledInFishSchool = fabs((fishes + i)->distanceTraveled);
        }
    }

#pragma omp parallel for
    for (int i = 0; i < numFishes; i++) {
        updateWeight((fishes + i), maxDistanceTraveledInFishSchool);
        printf(
                "Step Fish %d: x = %.2f, y = %.2f, distanceTraveled = %.2f, weight =
%.2f\n",
                i, (fishes + i)->x, (fishes + i)->y, (fishes + i)->distanceTraveled,
                (fishes + i)->weight
        );
    }
}

void calculateBarycenter(Fish *fishes, int numFishes) {
    double weightSum = 0.0;
    double distanceSum = 0.0;

#pragma omp parallel for reduction(+ : weightSum, distanceSum)
    for (int i = 0; i < numFishes; i++) {
        weightSum += (fishes + i)->weight * calculateDistanceFromOrigin((fishes +
i)->x, (fishes + i)->y);
        distanceSum += calculateDistanceFromOrigin((fishes + i)->x, (fishes +
i)->y);
    }

    if (distanceSum == 0.0) {
        printf("Distance sum is zero. Cannot calculate barycenter.\n");
        return;
    }

    double barycenter = weightSum / distanceSum;
    printf("Barycenter: %.2f\n", barycenter);
}

void initializeInitialLakeState(Fish *fishes, int numFishes) {
#pragma omp parallel for
    for (int i = 0; i < numFishes; i++) {
        (fishes + i)->x = getRandomNumberInRange(LAKE_X_MIN, LAKE_X_MAX);
        (fishes + i)->y = getRandomNumberInRange(LAKE_Y_MIN, LAKE_Y_MAX);
        (fishes + i)->distanceTraveled = 0.0;
        (fishes + i)->weight = 1.0;
        printf(
```

```
                "Initial Fish %d: x = %.2f, y = %.2f, distanceTraveled = %.2f,
weight = %.2f\n",
                i, (fishes + i)->x, (fishes + i)->y, (fishes + i)->distanceTraveled,
                (fishes + i)->weight
        );
    }
}

int main() {
    int provided;
    MPI_Init_thread(NULL, NULL, MPI_THREAD_FUNNELED, &provided);

    double start_time, end_time, elapsed_time;
    start_time = MPI_Wtime();  // Start recording the elapsed time

    MPI_Datatype MPI_FISH_TYPE;

    // Create MPI type for Fish struct
    int blocklengths[4] = {1, 1, 1, 1};
    MPI_Datatype types[4] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};

    MPI_Aint offsets[4];
    offsets[0] = offsetof(Fish, x);
    offsets[1] = offsetof(Fish, y);
    offsets[2] = offsetof(Fish, distanceTraveled);
    offsets[3] = offsetof(Fish, weight);

    MPI_Type_create_struct(4, blocklengths, offsets, types, &MPI_FISH_TYPE);
    MPI_Type_commit(&MPI_FISH_TYPE);

    if (provided != MPI_THREAD_FUNNELED) {
        fprintf(stderr, "MPI does not support MPI_THREAD_FUNNEL mode.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    srand(time(NULL) + rank);

    int *sendcounts = malloc(size * sizeof(int));
    int *displs = malloc(size * sizeof(int));
    int offset = 0;

    for (int i = 0; i < size; i++) {
        sendcounts[i] = NUM_FISHES / size + (i < (NUM_FISHES % size));
```

```
        displs[i] = offset;
        offset += sendcounts[i];
    }

    Fish *all_fishes = NULL;
    if (rank == 0) {
        all_fishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
        if (!all_fishes) {
            perror("Memory allocation failed for all_fishes");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        initializeInitialLakeState(all_fishes, NUM_FISHES);
    }

    Fish *local_fishes = (Fish *) malloc(sendcounts[rank] * sizeof(Fish));

    MPI_Scatterv(all_fishes, sendcounts, displs, MPI_FISH_TYPE, local_fishes,
sendcounts[rank], MPI_FISH_TYPE, 0,
                 MPI_COMM_WORLD);

    if (rank == 0) {
        free(all_fishes);
    }

    // ... [rest of the logic for the simulation, objective function, barycenter,
etc.]
    for (int step = 0; step < NUM_SIMULATION_STEPS; step++) {

        printf("Simulation step %d\n", step);
        simulationStep(local_fishes, sendcounts[rank]); // Use sendcounts[rank] as
localFishCount

        // Objective Function for the Entire Lake
        double localSum = 0.0;
        double globalSum = 0.0;

#pragma omp parallel for reduction(+ : localSum)
        for (int j = 0; j < sendcounts[rank]; j++) {  // Changed iterator name to j
            localSum += calculateDistanceFromOrigin(local_fishes[j].x,
local_fishes[j].y);
        }

        MPI_Allreduce(&localSum, &globalSum, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

        if (rank == 0) {
            printf("Objective Function for Entire Lake: %.2f\n", globalSum);
```

```
       }

       // Barycenter for the Entire Lake
       double localWeightSum = 0.0;
       double localDistanceSum = 0.0;
       double globalWeightSum = 0.0;
       double globalDistanceSum = 0.0;

#pragma omp parallel for reduction(+ : localWeightSum, localDistanceSum)
       for (int j = 0; j < sendcounts[rank]; j++) { // Changed iterator name to j
           localWeightSum +=
                   local_fishes[j].weight *
calculateDistanceFromOrigin(local_fishes[j].x, local_fishes[j].y);
           localDistanceSum += calculateDistanceFromOrigin(local_fishes[j].x,
local_fishes[j].y);
       }

       MPI_Allreduce(&localWeightSum, &globalWeightSum, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
       MPI_Allreduce(&localDistanceSum, &globalDistanceSum, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

       if (globalDistanceSum == 0.0) {
           if (rank == 0) {
               printf("Distance sum is zero. Cannot calculate barycenter.\n");
           }
       } else {
           double barycenter = globalWeightSum / globalDistanceSum;
           if (rank == 0) {
               printf("Barycenter for Entire Lake: %.2f\n", barycenter);
           }
       }

   }


   if (rank == 0) {
       all_fishes = (Fish *) malloc(NUM_FISHES * sizeof(Fish));
   }

   MPI_Gatherv(local_fishes, sendcounts[rank], MPI_FISH_TYPE, all_fishes,
sendcounts, displs, MPI_FISH_TYPE, 0,
               MPI_COMM_WORLD);

   if (rank == 0) {
       printf("\nAll fish after simulation steps:\n");
       for (int i = 0; i < NUM_FISHES; i++) {
```

```
            printf(
                    "Final Fish %d: x = %.2f, y = %.2f, distanceTraveled = %.2f,
weight = %.2f\n",
                    i, (all_fishes + i)->x, (all_fishes + i)->y, (all_fishes +
i)->distanceTraveled,
                    (all_fishes + i)->weight
            );
        }
        free(all_fishes);
    }

    free(local_fishes);
    free(sendcounts);
    free(displs);
    MPI_Type_free(&MPI_FISH_TYPE);


    end_time = MPI_Wtime();
    elapsed_time = end_time - start_time;

    double max_elapsed_time;
    MPI_Reduce(&elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Maximum time taken by any process: %f seconds\n", max_elapsed_time);
    }


    MPI_Finalize();
    return 0;
}
```

Overview

The provided code is a parallel fish simulation using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing). The program simulates the movement and behavior of a group of fish in a 2D lake. Each fish is represented by a structure (Fish) containing information about its position, distance traveled, and weight.

Here's an overview of the scatter and gather approach in the code:

- **Initialization**:

  The program starts by initializing the MPI environment and creating a custom MPI datatype (MPI_FISH_TYPE) for the Fish structure. The total number of processes (size) and the rank of each process (rank) are obtained.

- **Fish Distribution:**

  The main process (rank 0) initializes the initial state of the fish in the entire lake (all_fishes array). The MPI_Scatterv function is then used to distribute portions of the fish array to each process. The sendcounts and displs arrays determine how many fish each process receives and from which position in the original array.

- **Simulation Steps:**

  Each process independently simulates the movement of its local fish using OpenMP parallelization (simulationStep function). After each simulation step, the program computes the local objective function (sum of distances) and the local barycenter for the fish within each process.

- **Objective Function and Barycenter Calculation:**

  The local results (objective function and barycenter) are then combined across all processes using MPI collective communication functions (MPI_Allreduce). The global objective function and barycenter for the entire lake are printed by the root process (rank 0).

- **Gathering Results:**

  After the specified number of simulation steps, the local fish states are gathered back to the root process using MPI_Gatherv. The all_fishes array is then printed to display the final state of all fish.

- **Timing:**

  The program records the elapsed time for the entire simulation and prints the maximum time taken by any process using MPI reduction (MPI_Reduce).

Output (From Local Machine only)

```
$ mpiexec -n 4 ./cmake-build-debug/Project_2
Initial Fish 0: x = 25.82, y = -46.90, distanceTraveled = 0.00, weight = 1.00
Initial Fish 1: x = -70.82, y = -72.26, distanceTraveled = 0.00, weight = 1.00
Initial Fish 2: x = -44.54, y = -5.64, distanceTraveled = 0.00, weight = 1.00
Initial Fish 3: x = 68.88, y = -2.16, distanceTraveled = 0.00, weight = 1.00
Initial Fish 4: x = -82.40, y = -81.35, distanceTraveled = 0.00, weight = 1.00
Simulation step 0
Simulation step 0
Simulation step 0
Simulation step 0
Step Fish 1: x = -70.85, y = -72.35, distanceTraveled = 0.09, weight = 2.00
Step Fish 0: x = 25.77, y = -46.84, distanceTraveled = -0.08, weight = 0.11
Step Fish 0: x = -44.52, y = -5.66, distanceTraveled = -0.02, weight = 0.00
```

```
Step Fish 0: x = -82.37, y = -81.32, distanceTraveled = -0.04, weight = 0.00
Step Fish 0: x = 68.90, y = -2.15, distanceTraveled = 0.03, weight = 2.00
Objective Function for Entire Lake: 384.28
Barycenter for Entire Lake: 0.90

All fish after simulation steps:
Final Fish 0: x = 25.77, y = -46.84, distanceTraveled = -0.08, weight = 0.11
Final Fish 1: x = -70.85, y = -72.35, distanceTraveled = 0.09, weight = 2.00
Final Fish 2: x = -44.52, y = -5.66, distanceTraveled = -0.02, weight = 0.00
Final Fish 3: x = 68.90, y = -2.15, distanceTraveled = 0.03, weight = 2.00
Final Fish 4: x = -82.37, y = -81.32, distanceTraveled = -0.04, weight = 0.00
Maximum time taken by any process: 0.005428 seconds
```