CITS7707 Project 2 Report

Joo Kai Tay (22489437)

2023-10-16

Project Overview

Fish School Behaviour (FSB) is a heuristic algorithm used for multi-dimensional optimization problems. This project aims to test various sequential and parallel implementations of search based on the Fish School Behavior (FSB) algorithm using MPI(Message Passing Interface) to allow the program to execute on varying number of nodes. The performance for each experiment will be measured by the time taken to execute the program given a number of fish and steps.

Fish simulation

Fish Structure

The file fish.h contains the definition of a fish in this simulation. The struct Fish contains the following parameters:

- **double euclDist**: This is the euclidean distance of the fish from the origin in the previous step
- **double x_c**: This holds the x coordinate of the fish in the current step
- **double y_c**: This holds the y coordinate of the fish in the current step
- **double weight_c**: This holds the weight of the fish in the current step
- **double weight_p**: This holds the weight of the fish in the previous step

```
typedef struct fish {
    double euclDist;
    double x_c;
    double y_c;
    double weight_c;
    double weight_p;
} Fish;
```

In order to simplify passing the Fish structure using MPI, we will create a custom MPI datatype named `MPI_FISH`. This datatype will have the same fields as the fish structure described above.

```
MPI_Datatype create_mpi_fish_datatype() {
    MPI_Datatype MPI_FISH;
    MPI_Datatype types[NUMFIELDS] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};
    int blocklengths[NUMFIELDS] = {1, 1, 1, 1, 1};
    MPI_Aint offsets[NUMFIELDS];

    offsets[0] = offsetof(Fish, euclDist);
    offsets[1] = offsetof(Fish, x_c);
    offsets[2] = offsetof(Fish, y_c);
    offsets[3] = offsetof(Fish, weight_c);
    offsets[4] = offsetof(Fish, weight_p);
```

```
    MPI_Type_create_struct(NUMFIELDS, blocklengths, offsets, types, &MPI_FISH);
    MPI_Type_commit(&MPI_FISH);
    return(MPI_FISH);
}
```

Fish methods

fish.h also contains declarations for functions that are implemented in fish.c

This function dynamically allocates memory in the heap for an array of Fish structures and initializes each fish with random coordinates within a 200x200 square which represents the pond. The parameter `numfish` is used to specify the number of fish instead of a `#define NUMFISH` C preprocessor constant to allow for experiments with different numbers of fish.

```
Fish* initializeFish(int numfish)
```

This function updates the weight of a fish during a simulation step. It takes 3 parameters:

- **Fish* fish1**: A pointer to the fish that will be updated
- **double maxObj**: This is the maximum difference in distance between the previous step and the current step
- **int step**: The current step of the function

```
void eat(Fish* fish1, double maxObj, int step)
```

This function updates the position of a fish instance if its current weight is less than two times the STARTWEIGHT. The new position is calculated randomly within -0.1 and 0.1 and is then clamped to ensure it remains within a 200 x 200 square.

```
void swim(Fish* fish1)
```

File Writing

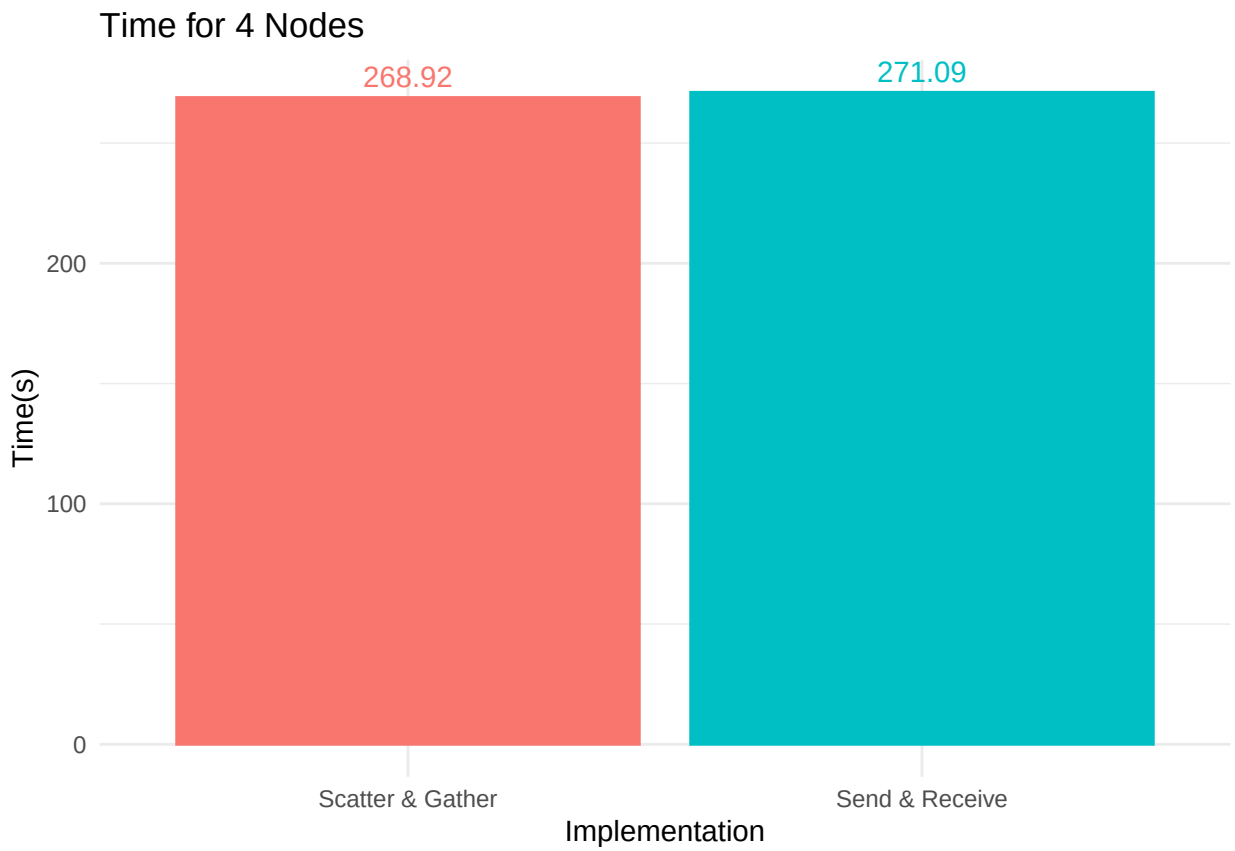
The deliverable is investigating the method to deliver effective communication between MPI processes. We want to establish the best method for communication so that the code in the following sections will run as efficiently as possible. The master process (rank == 0) generates the fish data and distributes it to the 4 nodes (including itself) using gather. Immediately after, it will write to a file named `initial_data.txt`. The master process will then gather the data back and write it into `final_data.txt`.

In this experiment, we will compare collective communication operations (like MPI_Scatter and MPI_Gather) with point-to-point operations (like MPI_Send and MPI_Recv) to determine which is the most suitable for transmitting the FISH data.

- Source file: `experiment0.c` & `experiment1.c`
- Run with: `sbatch experiment0.sh` & `sbatch experiment1.sh`
- Number of fish: 100,000,000
- Number of nodes: 4

The file experiment0.c uses `MPI_Scatter and MPI_Gather` to distribute the Fish data to the 4 processes including the master and gather to data back.

The file experiment1.c uses `MPI_Send and MPI_Receive` to distribute the Fish data to the 4 processes including the master and gather to data back.



From the plot above, we can see that using `MPI_Scatter & MPI_Gather` performed better than `MPI_Send and MPI_Receive`. There could be several reasons for this:

- Synchronization: Collective communication functions provide implicit synchronization. This means that processes participating in a collective operation will reach a known state relative to one another, which can be particularly useful when distributing data as you often want all processes to start computing on their respective data chunks simultaneously.
- Less Overhead: Using collective communication reduces the overhead of initiating and managing multiple point-to-point communication calls. When distributing a dataset, we have to send data to multiple processes, and managing these communications individually can be inefficient.

Comparing the file writing outputs

Afterwards to compare the outputs:

```
diff -s initial_data.txt final_data.txt
```

The output of this command should be: `Files initial_data.txt and final_data.txt are identical`

Experiment Methodology

The experiments will be compared against the best parallel and best sequential functions as found in project 1.

Base Case - Sequential

The base case for this simulation was implemented using the `void sequential(Fish* fishArray, int numfish, int numsteps)` function implemented in `sequential.c`. The FSB problem was tackled as such:

1. The outer loop runs for a number of iterations equal to `NUMSTEPS`.
2. The first of the inner loops iterates over all the fish present and finds the maximum difference in the position of the fish in the current and the fish in the previous round.
3. The second of the inner loops uses the maximum difference found in step 2 and performs the eat and swim operations on each fish in the array.
4. The final of the inner loops calculates the numerator and denominator variables for the barycentre.
5. The barycentre is calculated using the values from step 4.

To run this simulation use the following command:

```
sbatch experiment3.sh
```

Base Case - Parallel

The base case for parallel functions in this simulation is the `void parallelReduction(Fish* fishArray, int numfish, int numsteps)` function implemented in `parallel_functions.c`.

1. The first inner loop is parallelized using `#pragma omp for reduction(max:maxDiff)` inside the parallel region. The reduction clause ensures that the `maxDiff` variable is updated in a thread-safe manner, to avoid any race conditions when updating the variable.
2. The second inner loop is parallelized using `#pragma omp for` as the eat and swim operations are thread-safe, therefore, no special clauses need to be attached to this loop.
3. The last inner loop is also parallelized using `#pragma omp for reduction(+:sumOfProduct,sumOfDistance)` to avoid race conditions when updating the two variables.

To run this simulation use the following command:

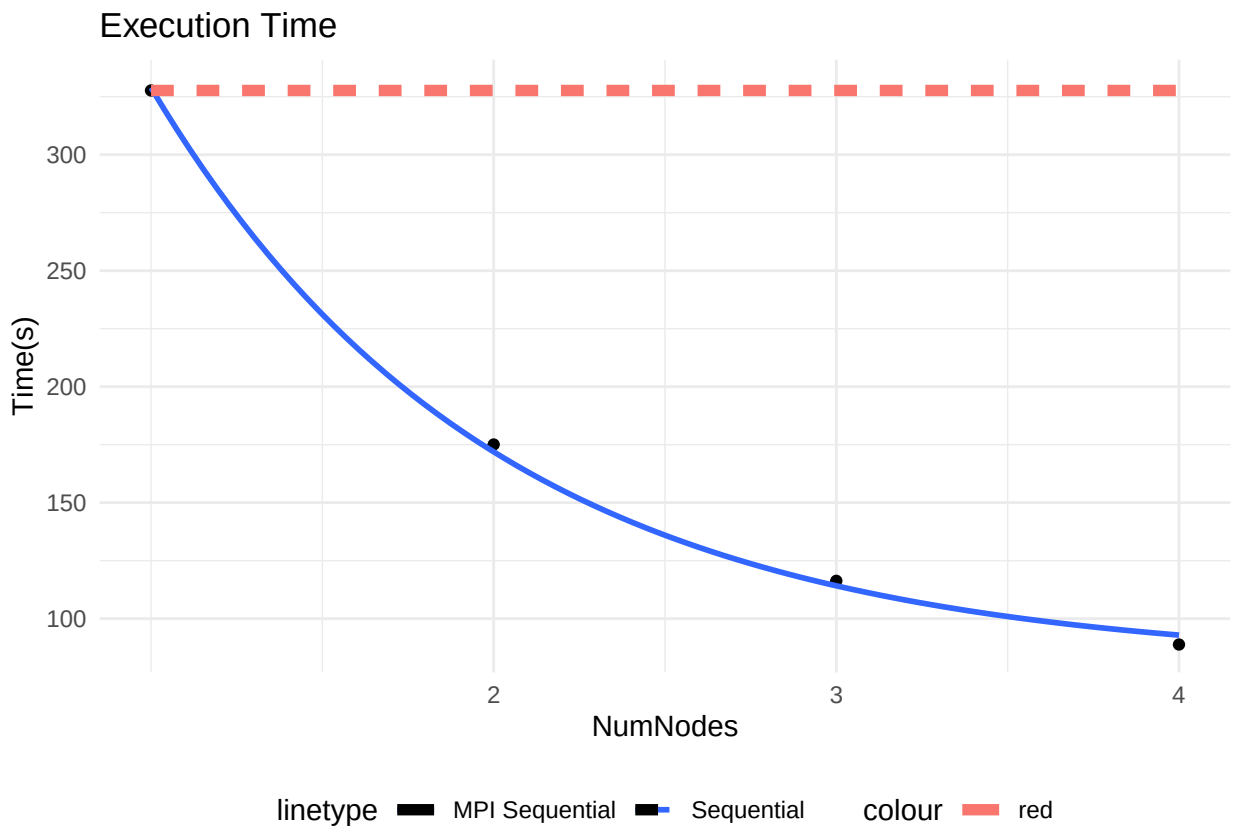
```
sbatch experiment4.sh
```

Experiments

Using MPI Only

In our initial experiment, we sought to evaluate the effect of using the MPI framework on the performance of the FSB algorithm. This required measuring the time of the MPI-only implementation, without any involvement of OpenMP threads. We will be comparing the MPI implementation with the base sequential implementation described above.

- Source file: `experiment3.c` & `experiment5.c`
- Run with: `sbatch experiment3.sh` & `sbatch experiment5.sh`
- Number of steps: 2000
- Number of fish: 1,000,000
- Number of nodes: 1, 2, 3, 4



The plot above shows the execution time as the number of MPI nodes increase. The sequential baseline took 327.68 seconds to execute. At 2 nodes, the MPI process took 175.07 seconds to execute which is a 46.6% reduction in time taken. At 3 nodes, the MPI code took 116.31 seconds to execute which is a 64.5% reduction in the time taken. At 4 nodes, the MPI code took 88.85 seconds to execute which is a 72.9% reduction in the time taken.

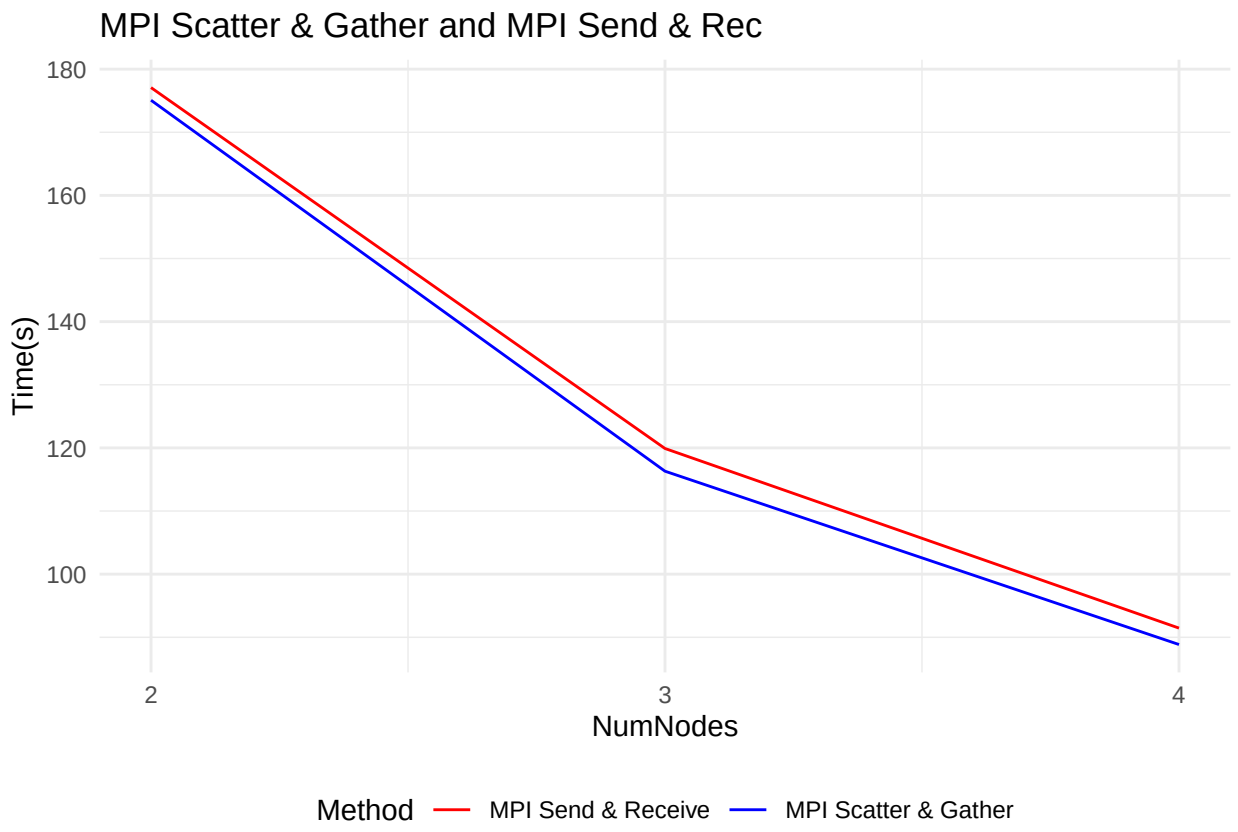
These percentages are very close to 0.5, 0.33 and 0.25 of the total time taken. There could be 2 possible reasons why they did not hit these numbers:

- The overheads from inter-process communication and synchronization caused the delay.
- Amdahl's Law, which states that the speedup of a parallel program is limited by its sequential portion. In this case, as we increase the number of nodes, the sequential portion becomes a smaller fraction of the total execution time, leading to diminishing returns in speedup.

Comparing different communication functions

- Source file: `experiment5.c` & `experiment8.c`
- Run with: `sbatch experiment5.sh` & `sbatch experiment8.sh`
- Number of steps: 2000
- Number of fish: 1,000,000
- Number of nodes: 2, 3, 4

The second experiment in MPI will be a follow up of the file writing experiment where we experiment with `MPI_Scatter` & `MPI_Gather` and `MPI_Send` & `MPI_Receive`. The earlier experiment tested the two different communication methods with 4 MPI nodes, and in this experiment we will vary the number of nodes to observe if changing the number of MPI nodes will affect the performance of the communication methods.



As seen from the plot above, using `MPI_Scatter` & `MPI_Gather` is more efficient than `MPI_Send` & `MPI_Receive` over all numbers of nodes. This is to be expected as collective communication functions, like MPI_Scatter and MPI_Gather, are often implemented using algorithms that are optimized for specific network topologies and hardware. They can take advantage of knowledge about the entire communication pattern to optimize data movement.

Collective communications are also more predictable since they involve a known set of processes. This predictability can be exploited by the underlying MPI implementation to further optimize communication.

Based on this knowledge, we will be using `MPI_Scatter` & `MPI_Gather` for communication between processes where we have to distribute fish in subsequent experiments.

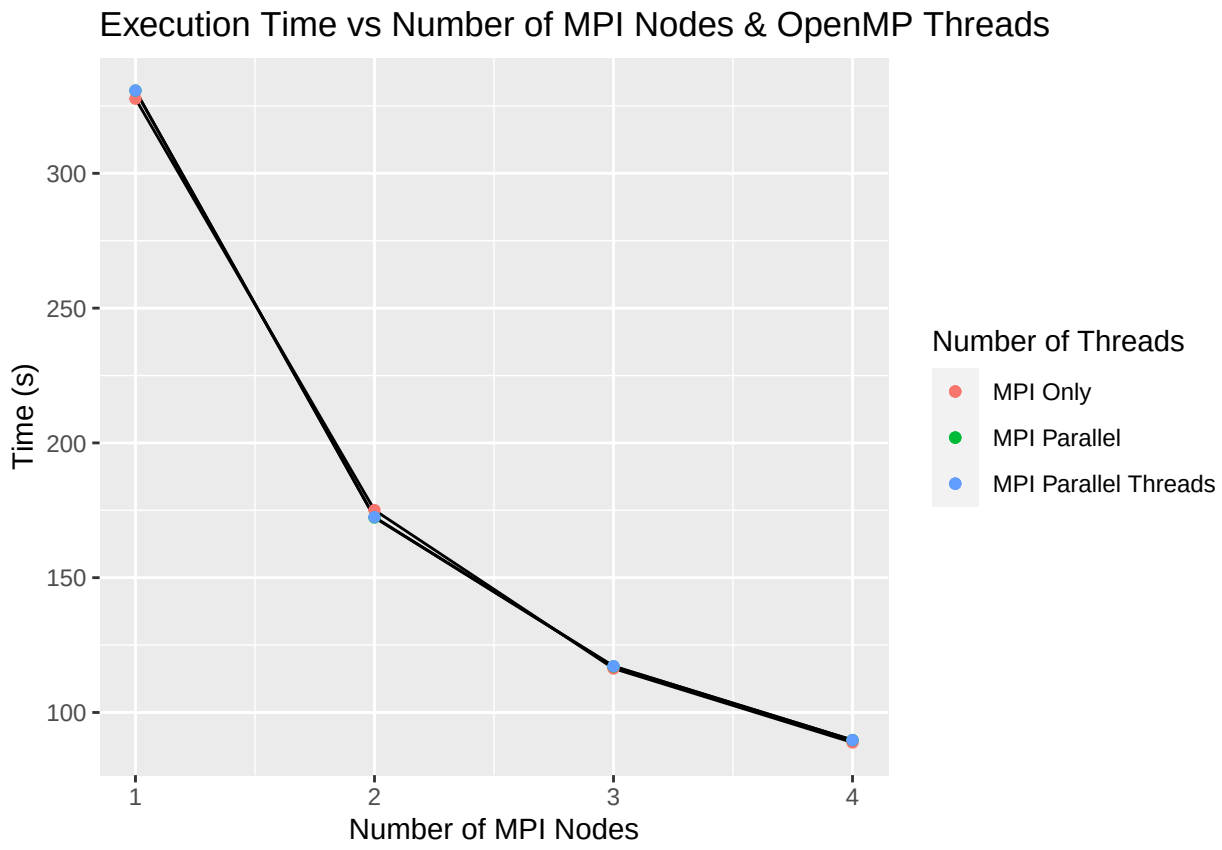
Comparing MPI Sequential and MPI Parallel

- Source file: `experiment3.c`, `experiment5.c`, `experiment6.c`
- Run with: `sbatch experiment3.sh`, `sbatch experiment5.sh`, `sbatch experiment6.sh`
- Number of steps: 2000
- Number of fish: 1,000,000
- Number of nodes: 2, 3, 4

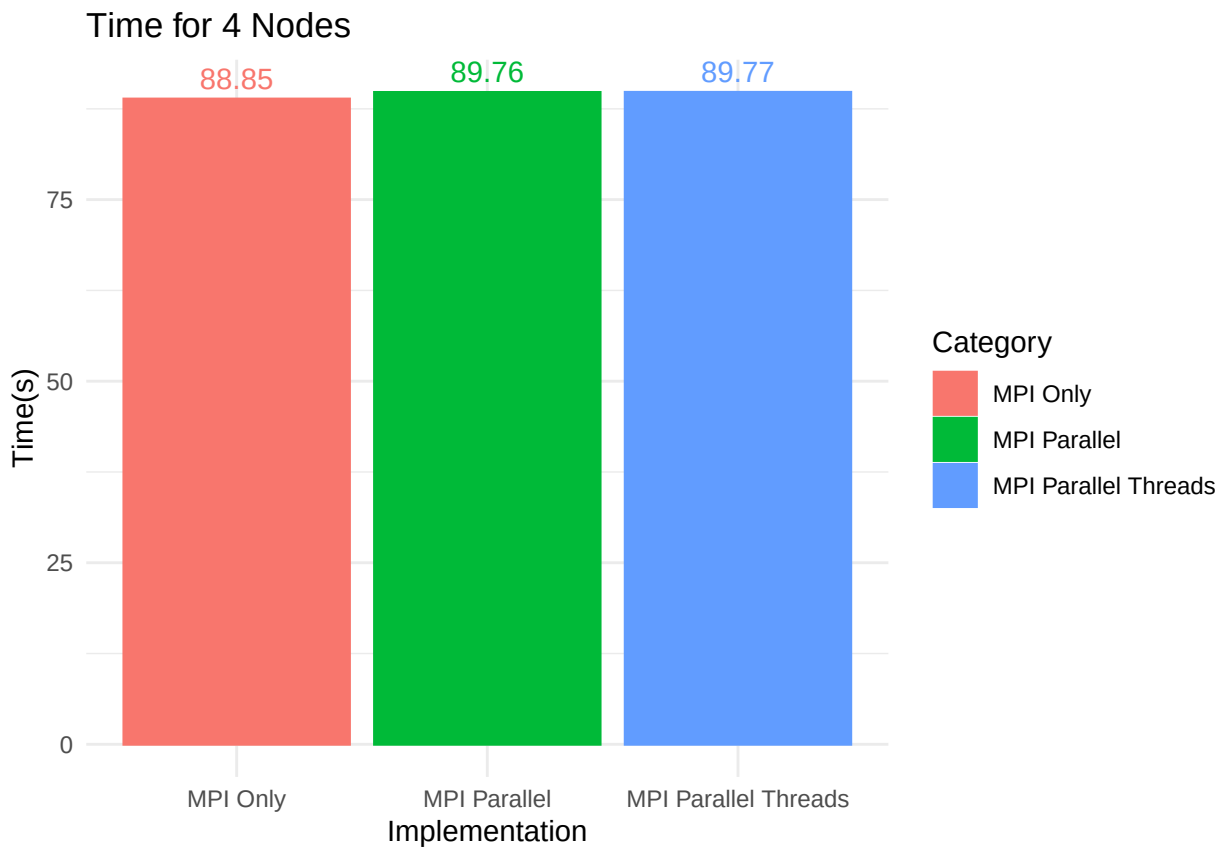
In this experiment, we will be comparing three scenarios:

1. MPI Only: This implementation makes use of just MPI with no threads.
2. MPI Parallel: This implementation makes use of MPI and OpenMP but uses MPI_THREAD_SINGLE (MPI_Init)
3. MPI Parallel Threads: This implementation makes use of MPI and OpenMP and uses MPI_Thread_FUNNELED

In theory, `MPI_Parallel` is still thread safe as it does not make MPI calls from anywhere other than the master thread, and the only difference between is using `MPI_Init` and `MPI_Init_thread`.



The plot above shows the execution time for the three implementations. It can be observed that the time differential between them is very small. In order to provide a better understanding, a bar chart of the times taken at 4 nodes is plotted below.

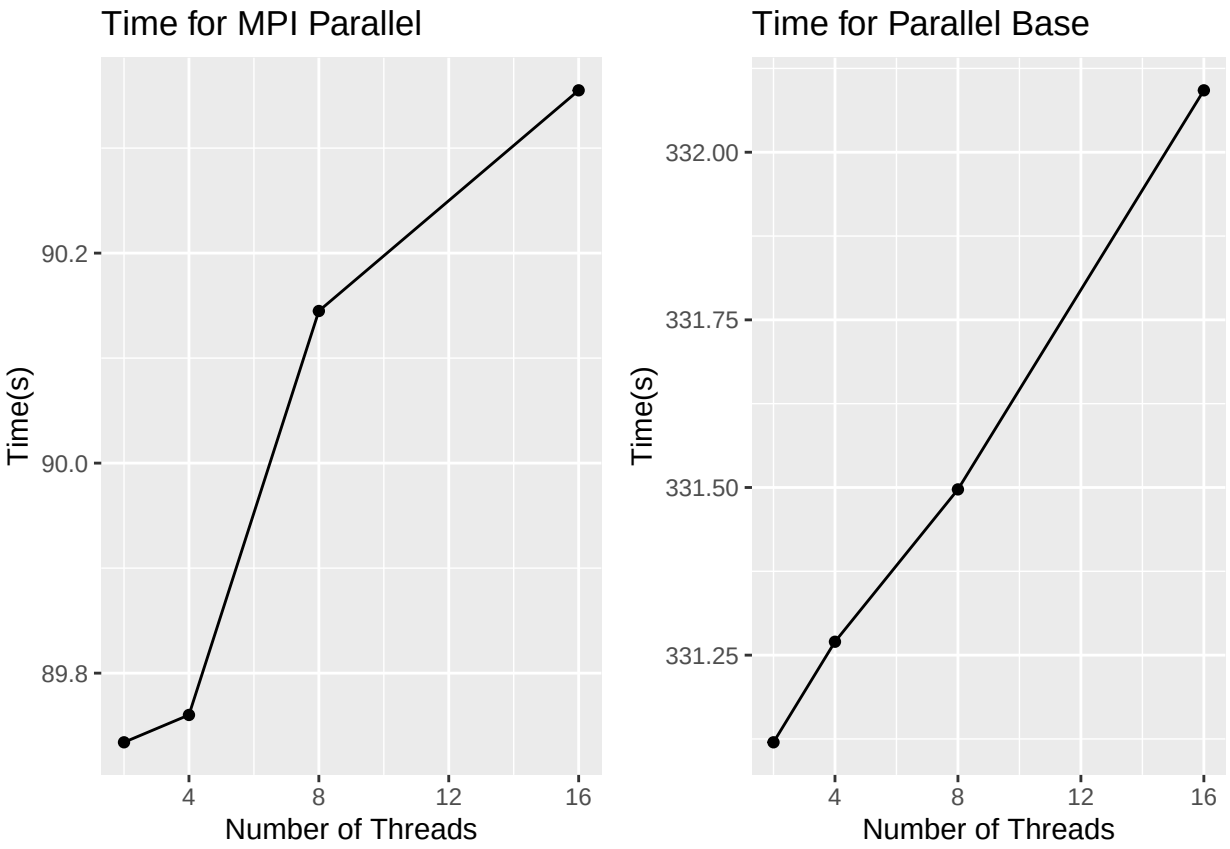


The plot above gives us a more detailed look at the timing of the three implementations. As previously stated, they are all very close in time taken. This is different from the results from project 1 where the implementations that made use of OpenMP were significantly slower than their sequential counterpart.

MPI parallel with different threads

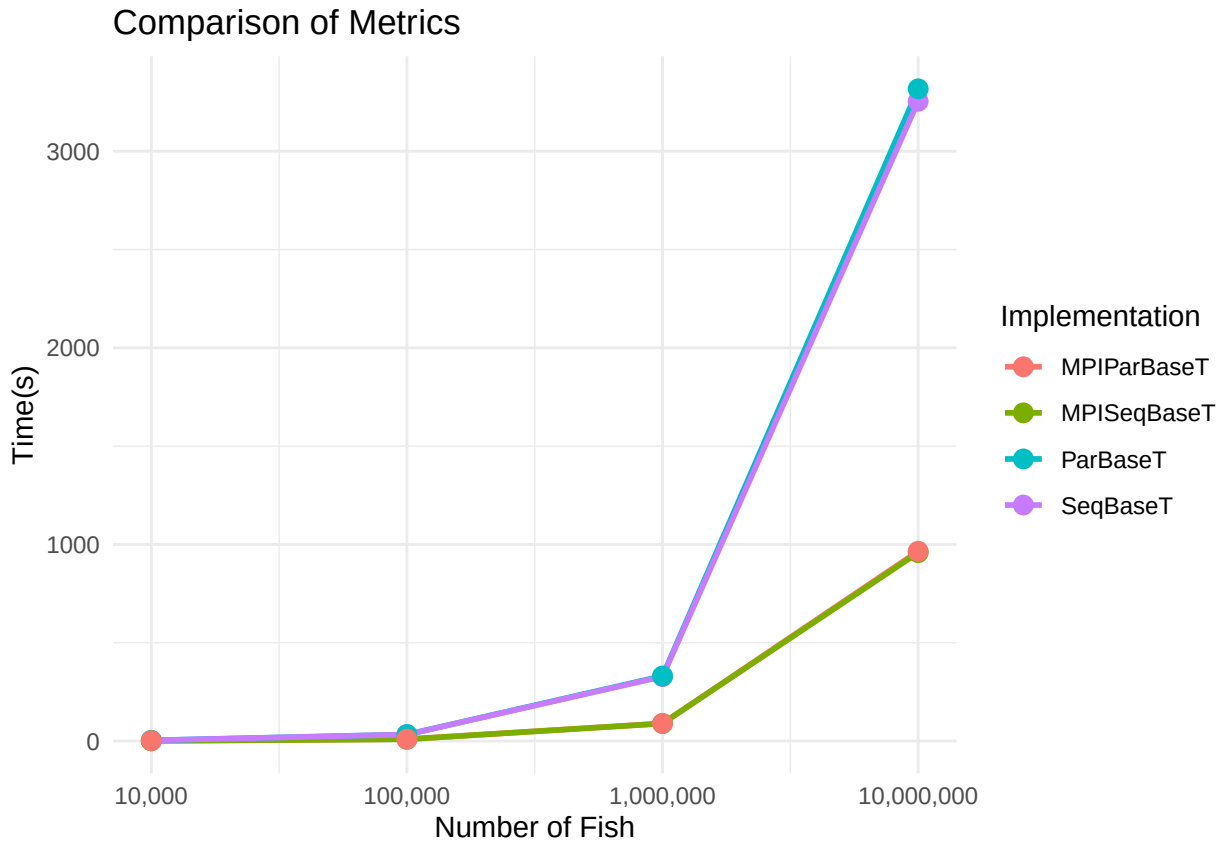
- Source file: `experiment2.c`, `experiment5.c`
- Run with: `sbatch experiment3.sh`, `sbatch experiment5.sh`
- Number of steps: 2000
- Number of fish: 1,000,000
- Number of nodes: 4
- Number of threads: 2, 4, 8, 16

In this experiment, we will investigate the increase in time as the number of threads increase and compare it to the base parallel implementation.



The plot above shows the increase in time taken as the number of threads increase for MPI + OpenMP and just OpenMP. For just OpenMP, the increase in time as the number of threads increase is relatively linear. However for MPI + OpenMP, the increase from 2-4 threads is significantly smaller than the increase from 4-8 threads.

Conclusion



The plot above shows the comparison of the functions with MPI and without MPI. It can be observed that as the size of the input grows, the benefits of using MPI increases. However, as seen in the results from both project 1 and project 2, the benefits of using multiple threads per process does not seem to benefit the Fish School Behavior (FSB) algorithm, as increasing the number of threads degrades the performance even for large input sizes (100,000,000 fish). Therefore, if we were to be implementing this algorithm in a research environment, we would focus on increasing the amount of resources for MPI, such as the ability to request more nodes. This would allow us to get the best possible performance.