CITS7707 Project 2 Report

Joo Kai Tay (22489437)

2023-10-16

```
## New names:
## * `ParBaseT` -> `ParBaseT...3`
## * `MPIParBaseT` -> `MPIParBaseT...5`
## * `ParBaseT` -> `ParBaseT...7`
## * `MPIParBaseT` -> `MPIParBaseT...8`
## * `MPIParBaseT` -> `MPIParBaseT...11`
```

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

In this experiment, we establish effective communication using MPI `scatter` and `gather`. This demonstrates that we can effectively communicate between up to 4 MPI processes. 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`.

The code to perform this experiment is stored in `experiment0.c`. Run this code using the command:

```
sbatch experiment0.sh
```

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 experiment1.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:

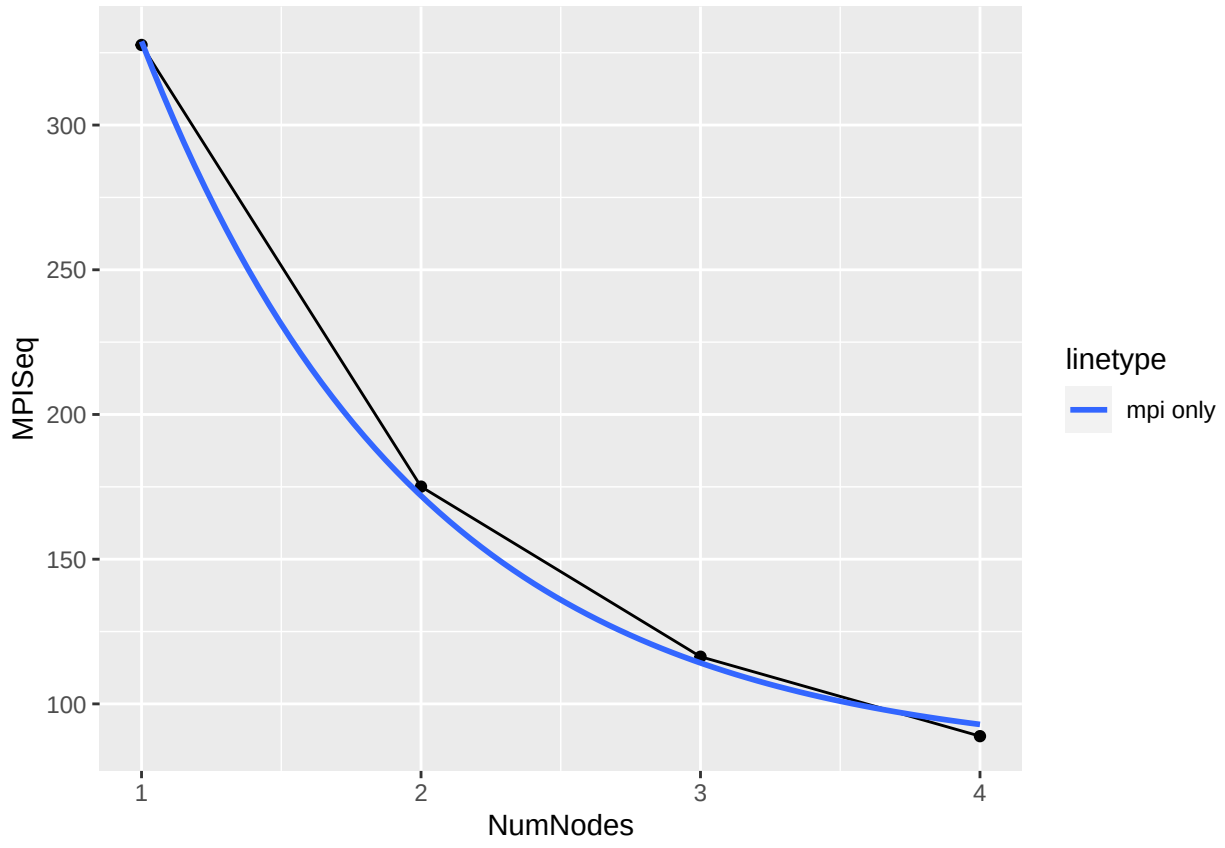
```
sbatch experiment2.sh
```

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: `experiment1.c` & `experiment3.c`
- Run with: `sbatch experiment1.sh` & `sbatch experiment3.sh`
- Number of steps: 2000
- Number of fish: 1,000,000
- Number of nodes: 1, 2, 3, 4

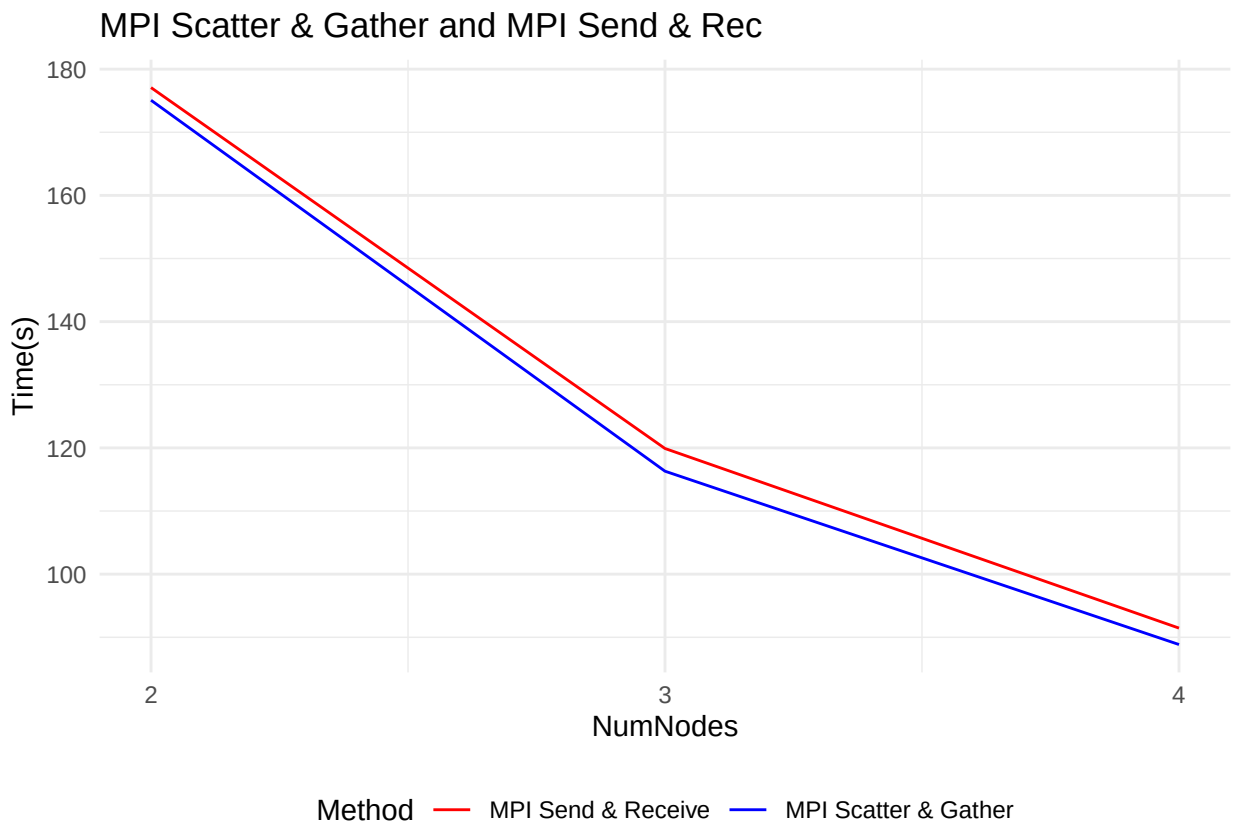



```
## NULL
```

Comparing different communication functions

- Source file: `experiment3.c` & `experiment4.c`
- Run with: `sbatch experiment3.sh` & `sbatch experiment4.sh`
- Number of steps: 2000
- Number of fish: 1,000,000
- Number of nodes: 2, 3, 4

The second experiment in MPI will be to establish which is a more effective method of communicating the fish data to the various processes. We will be comparing using `MPI_Scatter` & `MPI_Gather` with manually sending the fish using `MPI_Send` and `MPI_Receive`.



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.

Experiment: Comparing MPI Sequential and MPI Parallel

- MPISeq, MPIPar, MPIThreadSafe over 4 nodes
- 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 using just MPI with MPI and openMP.

Experiment 5: MPI parallel with different threads

Experiment 6: Compare all functions