NSDS MPI FISH PROJECT

Introduction

In this document we present main features of the program, our design and implementation choices and propose a time complexity analysis based on the number of fish present in the simulation

Data structures

We introduce the concept of *slice*. A slice is a portion of space inside the simulation. We create a number of slices equal to the number of processes and assign each slice to a different process. All slices are of the same size as our assumption is that all the processes will be placed in machines with the same compute capabilities. Moreover the shape and size of the slices is completely transparent to the rest of the program, in our program the slices are horizontal rectangles for the sake of simplicity.

In the program we use primarily one data structure to represent each fish. In turn each fish is represented by:

- Its position (x,y,z coordinates)
- Its speed, divided into x,y,z components
- Its size

Moreover, in order to ease the implementation, the *fish* data structure contains 3 additional fields:

- An ID unique in the entire simulation;
- An integer keeping track of how much the fish has eaten in the current simulation step
- A boolean indicating if the fish is has been eaten in the current simulation step

The entire program works by storing the fish data structures into double linked lists. In particular each processing node has a list containing the fishes belonging to the slice of the process and a variable number of lists that will contain fishes from other slices that may interact with the local fishes
Those double linked lists are sorted in order of decrescent size to speed up the calculation during the eating step.
In order to pass these data structures with the MPI functions we serialize the list into arrays.

The worst case complexity of the operation on the double linked lists:

- O(n) for adding as we do a sorted insert
- O(n) for lookup of an element
- O(n) for the serialization

Program functioning

The program starts out by reading the parameters and initializing the local variables needed for the simulation according to those parameters. Then each process creates an equal number of fishes randomly inside the entire space of the simulation and adds them to the local list of fishes. Each fish is spawned with a random speed.
After the initialization stage, the simulation begins. Each step of the simulation is divided into 3 substeps

- Moving step
- Eating step
- Statistics collection

Assumption: we assumed the cost of sending and receiving data with MPI as linear with regard to the amount of data that is transmitter or received

Moving step

Assumption:

- *if a fish were to move outside the bounds of the simulation, it will be placed on the border of the simulation and the speed components of the dimensions where the fish went out of bound will have the sign changed. (Ex in simulation of size 3, the fish moves to coordinates (1,4,3) with a speed of (1,2,2). It will be moved back to position (1,3,3) with a speed of(1,-2,2).*

The objective of the moving step is to move all the fishes and redistribute them between the processes according to the slice they ended up on.
Firstly each process initiates a number of arrays equal to the number of slices/processes.
Then, the process goes through the entire list of the local fishes and updates their position according to the speed of each fish. Every time the position of the fish is updated the process checks which slice is in charge of the new position: if the fish stayed in the same slice nothing happens; if the fish moved to a different slice the fish is moved into the array (implemented as a dynamic array/arraylist) corresponding to that slice.
After all the fishes have been moved each process will send each of the arrays to the corresponding process and receive all the arrays that the other processes will send to him. Once all the fishes are received, they are inserted into the local list of fishes.
Complexity of the step:

- Scan of the local list = O(n)
- Insert into arraylist = constant
- Sending/ receiving = O(n) *(assumption)*
- Insertion of all the fishes into the local list = O(n) x O(n)

Total complexity $O(n^2)$

Eating step

Assumptions:

- *If fish A is bigger than fish B and in range of eat AND fish C is bigger than fish A but NOT in size of fish B then fish A will eat B and fish C will eat A although it will increase only by one in size. This can propagate for an arbitrary long chain.*
- *In case a fish could be eaten by more than one fish, it will always be eaten by the bigger fish, in case they are tied in size, the fish from the local slice will win. If they are both from the local slice of both from a non-local slice. It is random who will eat.*

The objective of the eating step is to calculate all of the fishes that will eat/be eaten and increase the size of the fish that ate.
At the beginning of the step, each process calculates the slices that are in range of the eating distance, we will call those slices *neighbors*. Once they have done this, they send their list of local fishes to the processes corresponding to their neighbors and receive the list of fishes from the neighbors.
After this, the process will check for every fish if there is another fish from the local ones or from one of the neighbors that can eat it. If such fish is found, the fish is marked as eaten and the value of the fish that is eating which indicates how much it is going to grow at the end of the stage is increased by one. Finished this operation, the process will send back the lists of fishes to the corresponding processes. Those lists now contain the values indicating how much each fish has eaten. Once all the lists are received the processes begin a stage of cleanup: they go through all the lists received by the neighbors and update the local lists increasing the size of the fishes that ate and removing the fishes that had been marked as eaten.

Complexity of the step:

- Sending/ receiving = O(n) *(assumption)*
- For each local fish, find a fish that might eat it: worst case scenario $O(n^2)$, due to the list being sorted in decreasing size, we argue that the average case is close to O(n) as the step of finding the fish that could eat is close to constant time.
- Sending/ receiving = O(n) *(assumption)*
- Cleanup part = O(n) + O(n)

Total complexity $O(n^2)$

Statistics step

In this step the program checks for the required statistics (biggest fish and smallest fish) and checks if the simulation is over or not. To collect the statistics we have each process find its own local biggest and smallest fish then the process with the rank 0 will collect all the statistics and print on console.
Then to check if the game should proceed to the next step we check if the number of fishes in the simulation is 1 or if the remaining fishes are all of the same size, in which case we finish the game with a tie.

Complexity of the step:

- Finding the biggest fish O(1)
- Finding the smallest fish O(n)
- Sending/ receiving = O(1) *(assumption)*
- Checking if the game is tied O(1)

Total complexity O(n)

The total complexity of the simulation is the sum of the 3 steps multiplied by the number of iterations. The number of iterations is a function of the eating distance, size of the simulation and number of fishes, but due to us considering only the number of fishes as the parameters we choose to assume it as quadratic $O(n^2)$ as this cost is what better resembled what we observed in testing.
Thus the total cost of the simulation is: $O(n^2) \times [O(n^2) + O(n^2) + O(n)] = O(n^4)$

MPI design choices

Our base design philosophy is based around the fact that all the processes will be run on similarly capable machines and that no single process has the ability to store the entirety of the simulation all at once. For those reasons the slices are completely equal and at no point in time in the process any of the nodes will be required to store a number of fishes greater than what is strictly required.
We also have various synchronization points in the program placed primarily between the steps and where the data passing between processes happens

MPI communication

For all MPI communications we opted to use point-to-point communications. This choice was made because we rarely have the need to broadcast the same data to different processes.
Moreover all the communications are done in an asynchronous manner.
During both the eating and the moving steps the data passed around consists of arrays of fish structures, for which we created a custom MPI data type. When sending fishes, due to the variable dimension of the arrays, the processes first send the lengths of the arrays they are about to send, then each process allocates memory according to the lengths that it has received. Lastly it calls the asynchronous receive function and waits for the buffers to be filled before proceeding any further.