MPI – Sharks and Tunas

1 The sequential ocean

An ocean is modeled by a $N \times M$ matrix of cells which are either free ('F') or occupied by either a shark ('S') or a tuna ('T'). The goal of the simulation is to make the ocean discretely evolve according to the following rules :

— At each step, a shark tries to enter a cell contiguous to its current cell. It chooses one of these four cells randomly. If it is free, it moves into it. If it is occupied by a tuna, it moves into it and eats the tuna. If it is occupied by another shark, it does not move.

— At each step, a tuna also chooses a random contiguous cell. If it is free, it enters it. Otherwise, it does not move.

Note that the ocean is actually a torus : for example, when a fish decides to move to the left of the first column, it actually moves into the same row, but in the last column.

A sequential version of one step of this simulation is provided by the function `update_ocean(fish_t *ocean, int n, int m)`. To avoid moving several times the same fish within the same step, the structured type `fish_t` was created. It includes a flag indicating if the fish was recently moved, in which case, we should avoid moving it again.

The file `ocean.h` contains utility functions for the sequential simulation of the ocean. The file `ocean.c` is an example of execution of such a simulation. Have a look on these files, notice the constants defined at the beginning of `ocean.c`, compile and execute.

2 Towards the parallel ocean

2.1 Ocean dispatching

The goal is now to parallelise the simulation. The first step is thus to scatter the ocean over the set of processes. We will assume the number of rows of the ocean can be divided by the number of processes. However, to do so, you will have to be able to make learn a new type (`MPI_FISH`) to MPI so you will be able to send parts of the ocean to the processes. To do so, you will have to use the following function :

```
MPI_Type_create_struct(n_items, fish_lengths, fish_offsets, fish_types,
&MPI_FISH);
```

where `n_items` is a constant integer giving the number of fields in the structured type, `fish_lengths` is an array of integers giving the size of each field, `fish_offsets` is an array of `MPI_Aint` containing the offsets needed to reach the beginning of each field, `fish_types` is an array of `MPI_Datatype` enumerating the types of the fields, and `MPI_FISH` is the new `MPI_Datatype` created. Before using it, you will have to commit the new type by using

```
MPI_Type_commit(&MPI_FISH);
```

Now you can use `MPI_FISH` as any other MPI type.

2.2 Ocean update

Each process is now responsible for updating its own portion of the ocean. The function `update_ocean` can be used almost "as is". However, some of the fishes in one part of the ocean may decide to move to another part of the ocean, and thus may require communication between processes, each process sending and receiving fishes to/from their two neighbors.

For the sake of simplicity, we assume the following :

— When updating its own part of the ocean, each process keeps track of fishes moving out of it, by counting the precise number of sharks and tunas that need to be sent to one of its two neighbours. To achieve this part, write a function with the signature
`void update_ocean_part(fish_t *ocean, int n, int m, int *ns_north, int *nt_north, int *ns_south, int *nt_south)`
which, besides doing the job of the `update_ocean` function, counts the number of sharks and tunas to be sent to the processes taking care of the immediate northern and southern part of the ocean, respectively.

— Once the update is done, these numbers have to be sent in order to be taken into account by the relevant nodes. More precisely, each process needs to send two messages to its two neighbors. Thus, each node has to send (asynchronously) and receive 4 messages. Use point-to-point communications to implement this part.

— Finally, the fishes newly received by nodes need to be injected in their own part of the ocean. Again, for the sake of simplicity, we will assume that they can be injected in any free cell of their new part of the ocean. Create a function `void inject_ocean(fish_t* ocean, int n, int m, int ns, int nt)` achieving this injection.

2.3 Gathering the ocean

Once the number of update iterations wanted is reached, we need to gather the parts of the updated ocean back at the root. In this part, `MPI_Gather` should be used. Finally, the master prints the final ocean.

3 Report

There is no need for a proper report. Send your properly commented files by email before `March 13, 23:59` to `cedric.tedeschi@inria.fr`. Please mention "PPAR - Lab 3" in the subject.