High Performance Computing
Faculty of Industrial Engineering,
Mechanical Engineering and Computer Science
University of Iceland
Advisor: Prof. Dr-Ing. Morris Riedel

UNIVERSITY OF ICELAND

Fishing simulation

Jan Detroy, Vesa Mäkitalo, Jan Schwarz

April 24, 2022

Implementing a parallel map generator and a parallel fishing simulation. Generating 'realistic' maps with connected landmasses and surrounding water elements on quadratic 2D cartesian grids. In a simplified version of the generated maps groups of fish swim around in an ocean while two fishing boats move around independently fishing for them.

1 Understanding the MPI Cartesian Communicator & Modeling

Code description

To make the code more readable we first created a custom numeration type as well as two custom print functions to print out the map correctly for any size. The customization possibilities include the maps dimensions (`DIMX/Y` & `SIZE`) and the fraction of landmass (`FRAC`). There are two functions needed because c doesn't allow overloading.

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

#define SIZE 36
#define DIMX 6
#define DIMY 6
#define FRAC 0.25
#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3

typedef enum{water,land} gridType;
//------Print Fucntions for Array--------
void printArray(const gridType map[]){
  for(int i = 0; i < SIZE; i++) {
    printf("%d",map[i]);
    if (i!=0 && i%DIMX == DIMX-1) printf("\n");
  }
}
void printArray2(const int map[]){
  for(int i = 0; i < SIZE; i++) {
    printf("%d",map[i]);
    if (i!=0 && i%DIMX == DIMX-1) printf("\n");
  }
}
```

As a first step, the root process randomly assigns land cells until the defined fraction is achieved. We also check to make sure that the border cells remain water so the island is surrounded by water. Then after reaching the defined fraction the preliminary map is printed. Then the root process broadcasts the obtained map to all processes.

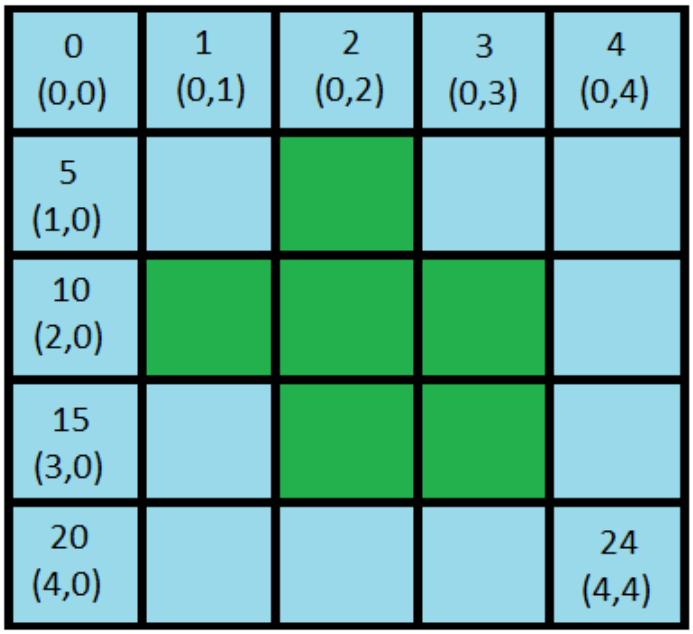
Algorithm 1 Pseudocode map generator

```
if root process then
    randomly assign land cells
    print preliminary map
    broadcast map to all other processes
else
    receive boradcasted map
end if
evaluate own cell status
exchange cell rank with neighbouring cells
exchange cell status with neighbouring cells
if land cell then
    count neighbours that are land cells
    if no land neighbours then
        declare self water
    end if
end if
if root process then
    gather map
    print changed map
end if
```

After receiving the preliminary map every process has to calculate its one-dimensional index in order to retrieve its state from the map array. Using the `MPI_Cart_Shift` command every cell obtains its neighbouring cells. Each process then communicates its state to their neighbours and receives the state of its neighbours. To avoid problems, all processes now wait for all other processes to finish communicating. Then every cell reports on the received information.

The rules for the making the map more realistic are the following: If a cell is a land cell and has no neighbors that are land cells, so it is a single cell island it will change its state to water. It will send out a `printf`-command about the state change and the root process then gathers all states to reconstruct the final map. With a `MPI_Barrier()` command we ensure that all processes are finished printing and sending data.

Finally we print out our final map (from the root process). In the figure below can be seen a visualization of a possible map, where blue blocks represent water and green blocks represent land, also coordinates and ranks of the cartesian communicator can be seen in the same figure.



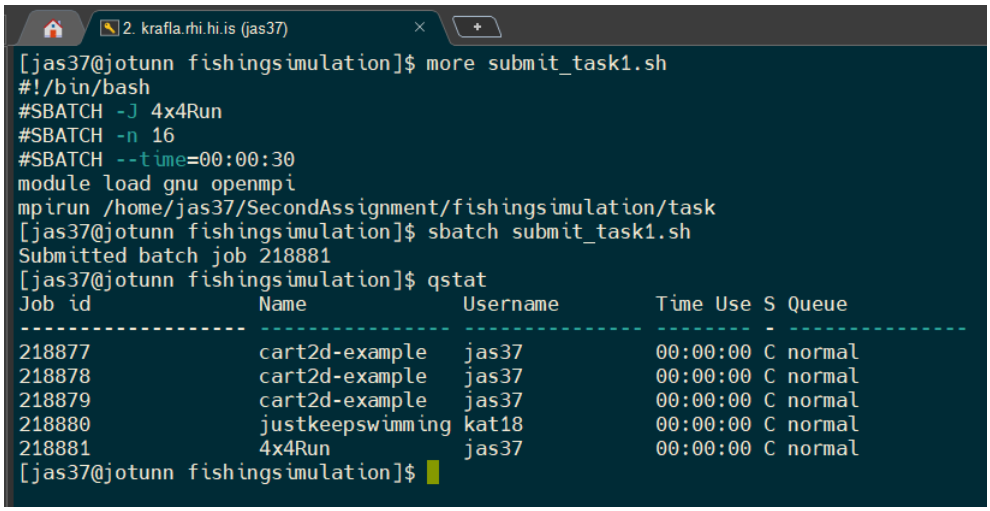
Program run reports

In order to run our code, we had to match the number of processes requested in the job script to the number of cells that we specified in the code. Here is the job script for the 6x6 grid size as an example:

```
#!/bin/bash
#SBATCH -J cart2d-example
#SBATCH -n 36
#SBATCH --time=00:00:30
module load gnu openmpi
mpirun /home/jpd5/2022-HPC-Course/Assignment-2/Task-1/task
```

4x4 run

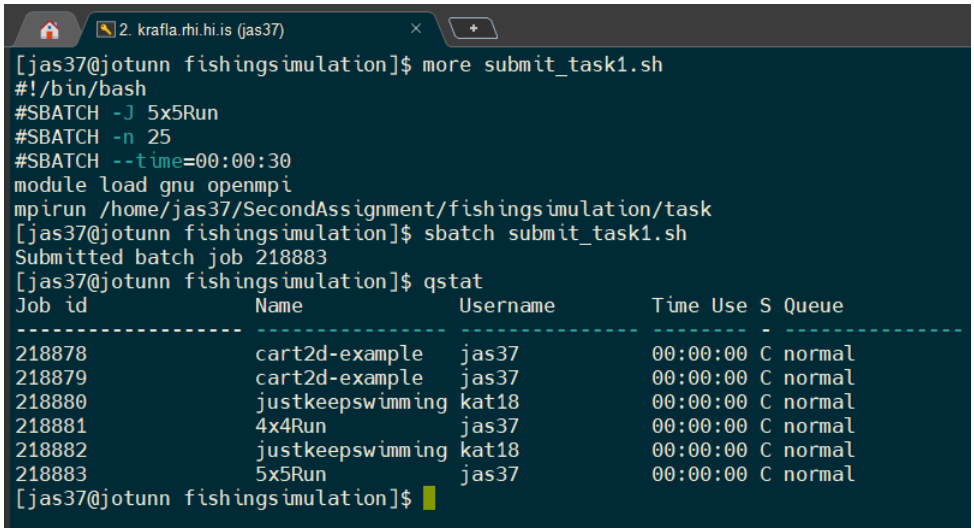
The first run is the run with a 4x4 grid. The rules that have been set up combined with the chosen volume fraction only allow one possible solution for a 4x4 grid, which can be seen in its slurm-script in the appendix. Most of the output scripts are not included in this document as the screenshots would have been far to long.




```
0000
0110
0110
0000
updated map:
0000
0110
0110
0000
```

5x5 run

The second run had the matching variables for a 5x5 grid. Here there were also no corrections due to lonely 'islands', but we can see that the one big island that has emerged contains one lonely field of water. One could argue either this is an unrealistic outcome because of a lonely field, or that this island has a very nice little lake in its middle.




```
00000
01100
01010
01110
00000
updated map:
00000
01100
01010
01110
00000
```

6x6 run

In the first 6x6 grid again there where no changes necessary.

```
2. krafla.rhi.hi.is (jas37)
[jas37@jotunn fishingsimulation]$ more submit_task1.sh
#!/bin/bash
#SBATCH -J 6x6Run
#SBATCH -n 36
#SBATCH --time=00:00:30
module load gnu openmpi
mpirun /home/jas37/SecondAssignment/fishingsimulation/task
[jas37@jotunn fishingsimulation]$ sbatch submit_task1.sh
Submitted batch job 218885
[jas37@jotunn fishingsimulation]$ qstat
Job id              Name             Username        Time Use S Queue
------------------ ---------------- --------------- -------- - ---------------
218880              justkeepswimming kat18           00:00:00 C normal
218881              4x4Run           jas37           00:00:00 C normal
218882              justkeepswimming kat18           00:00:00 C normal
218883              5x5Run           jas37           00:00:00 C normal
218884              justkeepswimming kat18           00:00:00 C normal
218885              6x6Run           jas37           00:00:00 C normal
[jas37@jotunn fishingsimulation]$
```

```
000000
001110
011000
010000
011100
000000
updated map:
000000
001110
011000
010000
011100
000000
```

However in a second one that aimed to achieve a shorter output script one lonely island became the victim of the same fate as Atlantis. Rank 25 has been changed to water.

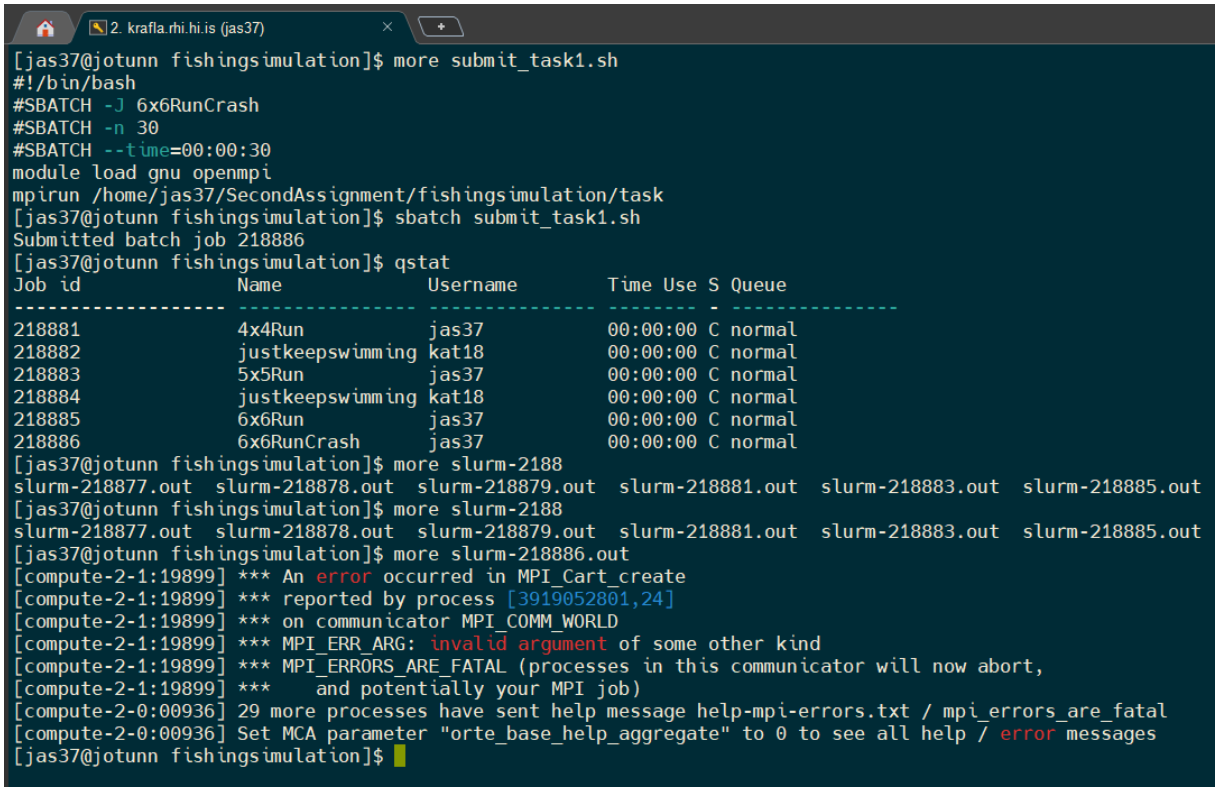
```
2. krafla.rhi.hi.is (jas37)
[jas37@jotunn fishingsimulation]$ mpicc task1.c -o task
[jas37@jotunn fishingsimulation]$ more submit_task1.sh
#!/bin/bash
#SBATCH -J 6x6RunShort
#SBATCH -n 36
#SBATCH --time=00:00:30
module load gnu openmpi
mpirun /home/jas37/SecondAssignment/fishingsimulation/task
[jas37@jotunn fishingsimulation]$ sbatch submit_task1.sh
Submitted batch job 218888
[jas37@jotunn fishingsimulation]$ qstat
Job id              Name             Username        Time Use S Queue
------------------ ---------------- --------------- -------- - ---------------
218886              6x6RunCrash      jas37           00:00:00 C normal
218887              justkeepswimming kat18           00:00:00 C normal
218888              6x6RunShort      jas37           00:00:00 C normal
[jas37@jotunn fishingsimulation]$ ls
fish          FishingSimPartlyWorking.c  slurm-218878.out  slurm-218881.out  slu
fishingSim.c  slurm-218877.out           slurm-218879.out  slurm-218883.out  slu
[jas37@jotunn fishingsimulation]$ more slurm-218888.out
000000
```

```
rank 25 changed to water
000000
001000
011110
001010
010010
000000
updated map:
000000
001000
011110
001010
000010
000000
```

All of the output scripts from above will be handed in with this report.

6x6 run with only 30 processes

Finally we submitted a 6x6 run without requesting enough processes. This resulted in the following output:


2. krafla.rhi.hi.is (jas37)
[jas37@jotunn fishingsimulation]$ more submit_task1.sh
#!/bin/bash
#SBATCH -J 6x6RunCrash
#SBATCH -n 30
#SBATCH --time=00:00:30
module load gnu openmpi
mpirun /home/jas37/SecondAssignment/fishingsimulation/task
[jas37@jotunn fishingsimulation]$ sbatch submit_task1.sh
Submitted batch job 218886
[jas37@jotunn fishingsimulation]$ qstat
Job id Name Username Time Use S Queue
218881 4x4Run jas37 00:00:00 C normal
218882 justkeepswimming kat18 00:00:00 C normal
218883 5x5Run jas37 00:00:00 C normal
218884 justkeepswimming kat18 00:00:00 C normal
218885 6x6Run jas37 00:00:00 C normal
218886 6x6RunCrash jas37 00:00:00 C normal
[jas37@jotunn fishingsimulation]$ more slurm-2188
slurm-218877.out slurm-218878.out slurm-218879.out slurm-218881.out slurm-218883.out slurm-218885.out
[jas37@jotunn fishingsimulation]$ more slurm-2188
slurm-218877.out slurm-218878.out slurm-218879.out slurm-218881.out slurm-218883.out slurm-218885.out
[jas37@jotunn fishingsimulation]$ more slurm-218886.out
[compute-2-1:19899] *** An error occurred in MPI_Cart_create
[compute-2-1:19899] *** reported by process [3919052801,24]
[compute-2-1:19899] *** on communicator MPI_COMM_WORLD
[compute-2-1:19899] *** MPI_ERR_ARG: invalid argument of some other kind
[compute-2-1:19899] *** MPI_ERRORS_ARE_FATAL (processes in this communicator will now abort,
[compute-2-1:19899] *** and potentially your MPI job)
[compute-2-0:00936] 29 more processes have sent help message help-mpi-errors.txt / mpi_errors_are_fatal
[compute-2-0:00936] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
[jas37@jotunn fishingsimulation]$

2 Fish and Ships

For this larger Assignment we decided to use Github for easier code sharing and version control. All code used can be found either in the attached files or on **Github**. The size of used cartesian communicator in this simulation is defined to be 36 and the number of iteration steps is defined to be 100. Both of those are easily changeable due to the property of HPC scalability and our coding style. To implement data passing between cores we took advantage of both blocking and non-blocking communication. Blocking communication is used to send and receive data between boats and the harbor. For example, if a boat reaches maximum fish capacity harbor will call it back and this needs to happen before the boat calculates a new step so blocking communication is necessary in this case. Another example of blocking communication is used when the Lighthouse sends the forbidden regions to the boats since the boats have to wait for that information until they can choose where to move. Non-blocking communication is used to send and receive data between neighbor cells. Full code for simulation and job script will be handed in with this report but for now let's describe the simulation steps with pseudocode.

Algorithm 2 Pseudocode fishing simulation

```
generate map with harbour, two fish and two boats in root process
broadcast map to all other processes
evaluate own cell status
if cell is harbour then
    evaluate map to get first boat pos
end if
receive neighbouring ranks
for all timesteps do
    if cell is harbour then
        calculate waves
        calculate forbidden cells
        send forbidden cells to boats
    else if cell is boat then
        receive forbidden pos
        randomly generate new dest avoiding forbidden pos
        send dest to harbour
        if boat got fish then
            increase fishcounter
        end if
    end if
    if cell is fish or boat got fish then
        r.g. dest avoiding land
    else if cell is fishfish or boat got fishfish then
        r.g. dest for first fish (avoiding boat pos)
        repeat
            r.g. different dest for second fish
        until second fis has different dest than first fish (and boat)
    end if
    enter dest into out. units array
    send out. units array to neighbour cells
    receive incoming units and combine in array
    evaluate inc. units according to Table 1
    if cell is harbour then
        print maps
    end if
end for
```

Algorithm description

In Table 1 the encoding of the different states can be seen. In the following the the pseudo-code from Algorithm 2 is shortly summarized: In a first step the root process generates a random map and broadcast that map to all processes. Subsequently the harbor gets send the boats position and calculates forbidden grid cells so that the boats can never collide. The Lighthouse in the harbor also watches out for storm cells and warns the boats about them so they don't enter those cells. Once the boats received the forbidden cells they chose a random direction to go in that is not forbidden. If a boat and a fish are in the same cell the boat increases its fish counter. Both boats and fish then get sent on to new neighbors using non blocking communication. Here all the special cases are handled so that for example both fish don't move to the same new cell. After each cell received the information about the neighbors the new situation is analyzed and the root process gathers an updated map to be printed. If the boats reach their maximum fish capacity they will return back to the harbor by calculating the direction they have to move to and then reducing the distance. All significant events during the execution are logged into a log file using parallel I/O.

Incoming units	**Cell state**
One boat	`boat1/boat2`
One fish	`fish1/fish2`
Two fish	`fishfish`
One boat & one fish	`boat1/boat2` & `gotFish=fish1/fish2`
One boat & two fish	`boat1/boat2` & `gotFish=fishfish`
Two boats	impossible

Table 1: Incoming units evaluation table

Output and visualization

To visualize the state of the simulation we used emojis. Used emojis are self-explanatory except that the duck emoji represents a second boat. Visualization of simulation is saved to log files using MPI-I/O. To optimize the MPI-I/O process we utilized independent parallel I/O and cooperative parallel I/O. The core that is allocated for harbor will write independently state map and wave map to *MapLog*-file. Every event that happens during simulations is written to *ParallelLog*-file cooperative and simultaneously by every core. This method improves the readability of log files and it is also easier to implement than writing everything in one file. In the figure below can be seen both *ParallelLog*-file and *MapLog*-file, also elapsed time in seconds is shown in figure.

```
[vvo2@jotunn FishingSimulation]$ sbatch submit_fishingSim.sh
Submitted batch job 219936
[vvo2@jotunn FishingSimulation]$ more MapLog
Iteration step: 000, time: 00.23s   |              wavemap
~~~~~~~~~~🚢                        |  0.00  1.01  0.00  0.00  0.00  0.00
~~~~~~~~🏝 ~~                       |  0.00  0.74  0.00  0.00  0.00  0.00
~~~~~~🦀 ~~~~                       |  0.00  0.49  0.00  0.00  0.00  0.00
~~~~~~~~~~~~                        |  0.00  0.34  0.00  0.00  0.00  0.00
⚓ ~~~~~~~~~~                       |  0.00  0.14  0.00  0.00  0.00  0.00
~~~~~~~~🐟 ~~                       |  0.00  0.12  0.00  0.00  0.00  0.00


Iteration step: 001, time: 00.25s   |              wavemap
~~~~~~~~~~~~                        |  0.00  1.98  1.01  0.00  0.00  0.00
~~~~~~~~~~🚢                        |  0.00  1.83  0.74  0.00  0.00  0.00
~~~~🦀 ~~🏝 ~~                      |  0.00  1.50  0.49  0.00  0.00  0.00
~~~~~~~~~~~~                        |  0.00  1.31  0.34  0.00  0.00  0.00
⚓ ~~~~~~🐟 ~~                      |  0.00  1.05  0.14  0.00  0.00  0.00
~~~~~~~~~~~~                        |  0.00  0.76  0.12  0.00  0.00  0.00


Iteration step: 002, time: 00.28s   |              wavemap
~~~~~~~~~~🚢                        |  0.00  2.06  1.98  1.01  0.00  0.00
~~~~~~~~~~~~                        |  0.00  2.11  1.83  0.74  0.00  0.00
~~🦀 ~~~~~~🏝                       |  0.00  2.04  1.50  0.49  0.00  0.00
~~~~~~~~🐟 ~~                       |  0.00  2.09  1.31  0.34  0.00  0.00
⚓ ~~~~~~~~~~                       |  0.00  1.90  1.05  0.14  0.00  0.00
~~~~~~~~~~~~                        |  0.00  1.66  0.76  0.12  0.00  0.00


Iteration step: 003, time: 00.30s   |              wavemap
~~~~~~~~🚢 ~~                       |  0.00  1.20  2.06  1.98  1.01  0.00
~~~~~~~~~~~~                        |  0.00  1.45  2.11  1.83  0.74  0.00
🦀 ~~~~~~~~~~                       |  0.00  1.67  2.04  1.50  0.49  0.00
~~~~~~~~~~🐟                        |  0.00  1.86  2.09  1.31  0.34  0.00
⚓ ~~~~~~~~~~                       |  0.00  1.96  1.90  1.05  0.14  0.00
~~~~~~~~~~~~                        |  0.00  2.06  1.66  0.76  0.12  0.00


Iteration step: 004, time: 00.33s   |              wavemap
~~~~~~~~~~~~                        |  0.00  0.31  1.20  2.06  1.98  1.01
~~~~~~~~~~~~                        |  0.00  0.43  1.45  2.11  1.83  0.74
~~🦀 ~~~~~~🏝                       |  0.00  0.54  1.67  2.04  1.50  0.49
🐟 ~~~~~~~~~~                       |  0.00  0.98  1.86  2.09  1.31  0.34
⚓ ~~~~~~~~~~                       |  0.00  1.22  1.96  1.90  1.05  0.14
~~~~~~~~🚢 ~~                       |  0.00  1.36  2.06  1.66  0.76  0.12
```

```
[vvo2@jotunn FishingSimulation]$ more ParallelLog
Boat 2 here, we caught 1 fish. Iteration step: 004.
Boat 2 here, we caught 3 fish. Iteration step: 006.
Moving boat 2 to harbor.       Iteration step: 007.
Boat arrived to harbor.        Iteration step: 007.
Boat 2 here, we caught 1 fish. Iteration step: 014.
Boat 2 here, we caught 2 fish. Iteration step: 022.
Storm detected!                Iteration step: 028.
Storm detected!                Iteration step: 029.
Storm detected!                Iteration step: 030.
Storm detected!                Iteration step: 031.
Boat 1 here, we caught 1 fish. Iteration step: 031.
Storm detected!                Iteration step: 032.
Storm detected!                Iteration step: 033.
Boat 1 here, we caught 2 fish. Iteration step: 038.
Storm detected!                Iteration step: 040.
Storm detected!                Iteration step: 041.
Boat 1 here, we caught 3 fish. Iteration step: 041.
Storm detected!                Iteration step: 042.
Moving boat 1 to harbor.       Iteration step: 042.
Storm detected!                Iteration step: 043.
Moving boat 1 to harbor.       Iteration step: 043.
Storm detected!                Iteration step: 044.
Moving boat 1 to harbor.       Iteration step: 044.
Storm detected!                Iteration step: 045.
Boat 2 here, we caught 3 fish. Iteration step: 045.
Moving boat 1 to harbor.       Iteration step: 045.
Moving boat 1 to harbor.       Iteration step: 046.
Moving boat 2 to harbor.       Iteration step: 046.
Moving boat 2 to harbor.       Iteration step: 047.
Boat arrived to harbor.        Iteration step: 047.
Moving boat 1 to harbor.       Iteration step: 047.
Boat arrived to harbor.        Iteration step: 047.
Boat 1 here, we caught 1 fish. Iteration step: 047.
Boat 2 here, we caught 1 fish. Iteration step: 049.
Storm detected!                Iteration step: 051.
Storm detected!                Iteration step: 052.
Storm detected!                Iteration step: 052.
Boat 2 here, we caught 2 fish. Iteration step: 052.
Storm detected!                Iteration step: 053.
Storm detected!                Iteration step: 053.
Storm detected!                Iteration step: 054.
Storm detected!                Iteration step: 054.
Boat 2 here, we caught 3 fish. Iteration step: 054.
Storm detected!                Iteration step: 055.
Storm detected!                Iteration step: 055.
Moving boat 2 to harbor.       Iteration step: 055.
```

Wave propagation

Numbers that can be seen in the wave map are generated by *generateWave()*-function which is shown in the figure below. This function is called in the left most column and the wave propagates to right. A normal distributed random variable is added to the wave function to create a more realistic wave. If the height of the wave is more than 2.15, that specific cell will be defined as a storm cell. This function uses math operations so when the c-file is compiled there need to be added command '-lm' at end of that line.

```c
double generateWave(const int timestep,const int rank){
    double height;
    double a = 1;
    double PI2 = 2*M_PI;
    double k = PI2/DIMX;
    double omega = PI2/DIMX;
    height = a*sin(k*timestep+rank)+a+randfrom(0,0.2);
    return height;
}
```