Parallel Programming HW4: WATOR

9961244 EE14 吳德霖

◆ Instructions

1. Compile and Execute

To compile: mpicc –o wator –lX11 wator.c

To run: mpirun –n [NUM_TASKS] ./wator config.txt outlog.txt

2. List of source code

There only one source code which is exactly the "wator.c".

◆ Design and structure

✧ Basic Design

The basic design of this version of WATOR is just same as that told during class, but I slightly added my thought toward a real WATOR. Since the neutrality told us that hunters would hunt their prey when starving, as a result, I let every sharks initiate each step before fishes do the function. So that I could ensure sharks would have higher possibility to hunt their prey, randomly though. Another thing is I used ID number and it's unchangeable no matter the creature is alive or dead, so that I can trace back to the vital information needed by the creatures' IDs.

✧ Load Balancing

Since I used the master rank as a job distributor, I can do the load balancing by taking advantage to the functionality of the master. Every time the slaves done their jobs, they will send back temporary results to the master. Upon receiving the results, the master would sense the whole wator world map to re allocate the rows to redistribute the load with respect to rows. Each slave would have some rows to do the next job, by which the number of creatures within these rows are roughly the same. The master would accumulate current sent creatures as long as the corresponding rows to decide how many rows go to the next slave. After the redistribution is done, the slaves would then start to do the jobs during the next step.

✧ Preventing Roll Back

The most vital problem of the parallelized version of this kind of task is the synchronization problem, occurring when two creatures try to occupy the same position. But since I applied the MPI library, I can make good use of its communication ability. The roll back would only have to be occurred on the border line throughout all functioning slaves, so I simply send the requiring first two rows to the previous rank each of the slaves after these two rows are done. As a result, the previous rank would have the information about the row beyond the border line and the invasion of its own last row, and since one creature could only move one step, no matter adjacent or diagonal, this would be sufficient to prevent the rollback of the last row of the slave. For each slave, they will start from the row given by the master and row by row, so it's of an increasing order. After all the row is done, this step is done. At the beginning and at the end of one step, since sharks might eat the fishes not from their hosting rank, so the communication between slaves would be generated again to ensure the right prey should be dead and eliminated.

- ✧ **Additional features**

  I added one creature to make the version funnier, which were the fisherman, and they will not breed or die, only suffer from health conditions. Each fisherman would have a temperature indicating if he is ill or health, and unfortunately he can't go fishing if the condition turns out to be ill. The fisherman have scores, when they got a shark, they score 2, while score 1if they got a fish. Just for fun!

  The basic concept of my wator is a side view of a water system, so the fisherman could only kill the upper most creatures. And one by each step!

◆ Specialty / Experiment

1. Experiments

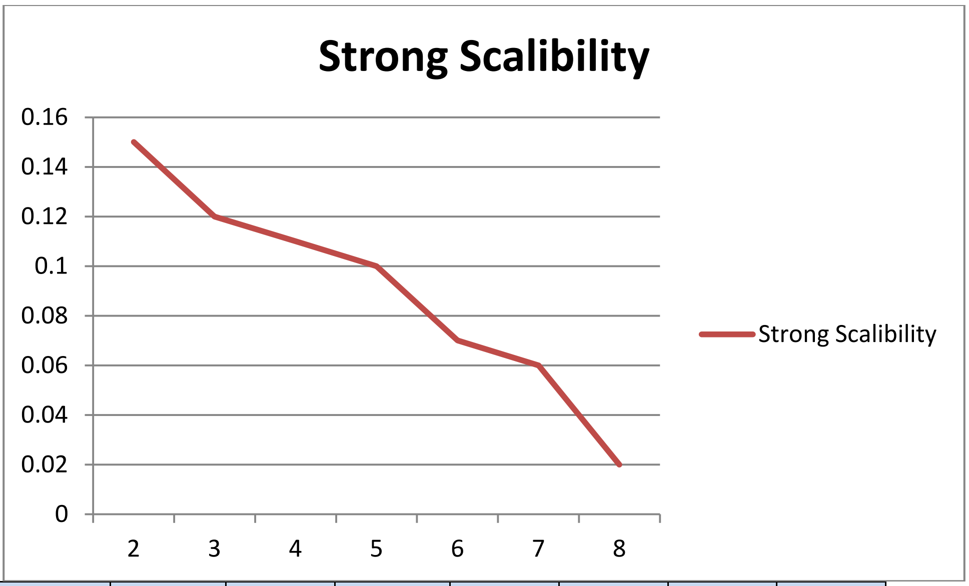
- ✧ **Strong scalability**

  Parameters:

  MAX_X = 20, MAX_Y = 20, WIDTH = HEIGHT = 20

  NUM_FISH = 100 NUM_SHARK = 30

  SIM_STEPS = 20



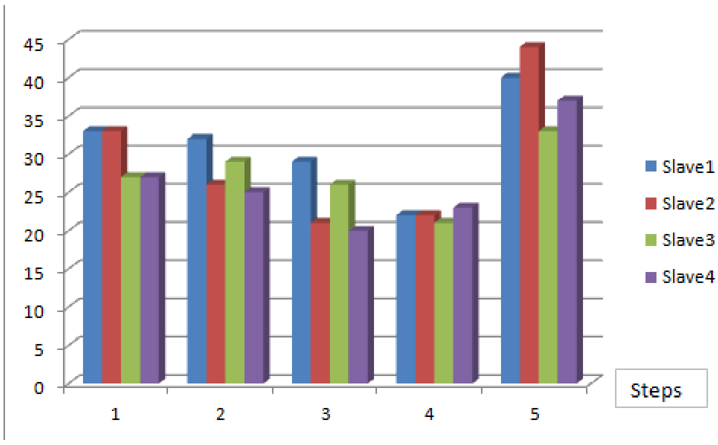
#Procs	2	3	4	5	6	7	8
Time(s)	0.15	0.12	0.11	0.1	0.07	0.06	0.02

The above result shows a quite good scalability, although not very strong. I think it's due to the parameters I applied wasn't large enough to see the great difference. But considering I used dummy array and the condition of the VM isn't sufficiently stable these days, I simply conducted such experiment to prevent some memory errors during rush hours.

✧ **Load balancing**

Using pretty much the same parameters, here I tested the load balancing efficiency to see if each task received quite the same computations.
Here I applied 5 processors, which implies 4 slaves, and recorded the fist several steps of computations among slaves.
The result is as shown below: The number is of how many creatures to deal with during such step, which is, computations. (Fishes + Sharks)

#Steps	1	2	3	4	5
Slave1	33	32	29	22	40
Slave2	33	26	21	22	44
Slave3	27	29	26	21	33
Slave4	27	25	20	23	37

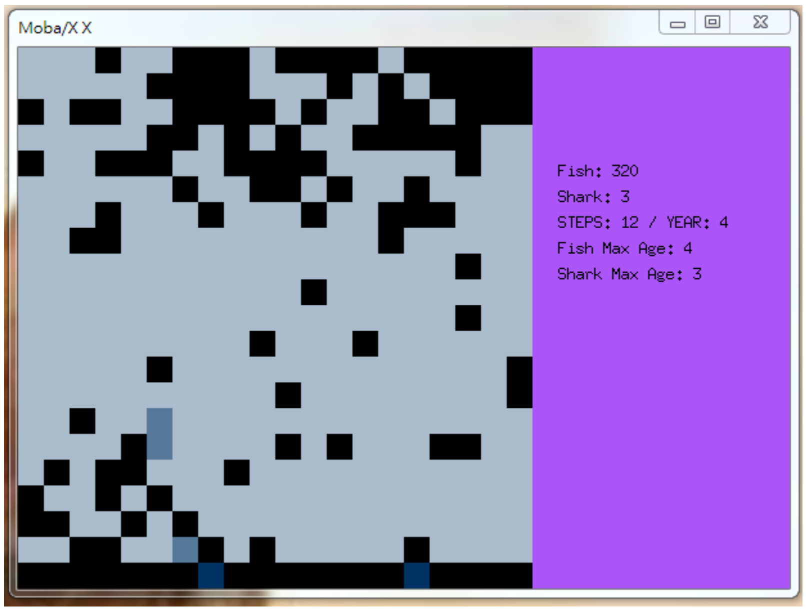


According to the above graph and format, we can tell that the load balancing method was quite as much working, even when breeding event occurs, notice that the fifth steps has a significance increment in the numbers of computations, that is, creatures.

2. Specialty

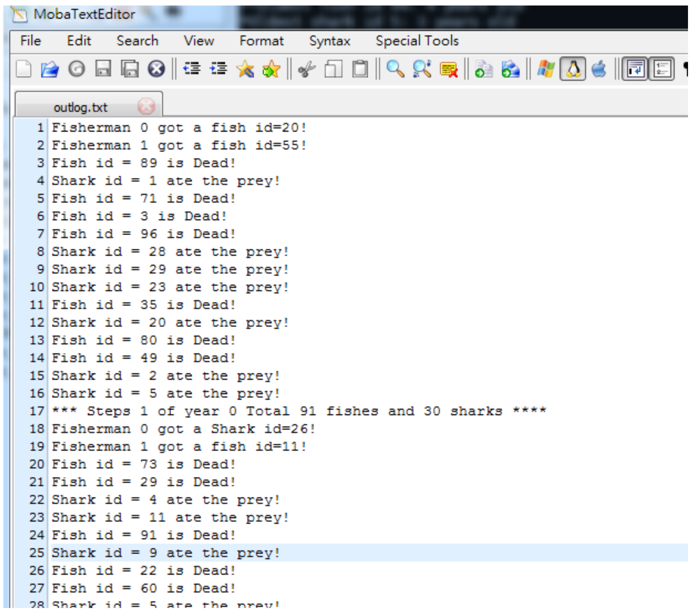
After showing the previous two experiments, we can see that my version of wator has a pretty good performance in load balancing the tasks, which in turns result in better scalability, especially using MPI. I showed much of the information in the terminal, while only printed out the vital generalized information of the creatures. Since the number of the fisherman isn't specified explicitly, I didn't print it out on the x window because it might truncate the full size of the window I declared.

The below is sample running of the output, the grey rectangles are fishes, and the light blue rectangles represent the sharks, with the dark blue in the very bottom indicating the position of the fishermans.



```
Fish: 320
Shark: 3
STEPS: 12 / YEAR: 4
Fish Max Age: 4
Shark Max Age: 3
```

The below is a sample capture of the output log file, it describes if sharks ate their prey (and which prey), if fisherman got fishes or sharks, and if both creatures breed. All the demanded trace back information is included.



```
Fisherman 0 got a fish id=20!
Fisherman 1 got a fish id=55!
Fish id = 89 is Dead!
Shark id = 1 ate the prey!
Fish id = 71 is Dead!
Fish id = 3 is Dead!
Fish id = 96 is Dead!
Shark id = 28 ate the prey!
Shark id = 29 ate the prey!
Shark id = 23 ate the prey!
Fish id = 35 is Dead!
Shark id = 20 ate the prey!
Fish id = 80 is Dead!
Fish id = 49 is Dead!
Shark id = 2 ate the prey!
Shark id = 5 ate the prey!
*** Steps 1 of year 0 Total 91 fishes and 30 sharks ****
Fisherman 0 got a Shark id=26!
Fisherman 1 got a fish id=11!
Fish id = 73 is Dead!
Fish id = 29 is Dead!
Shark id = 4 ate the prey!
Shark id = 11 ate the prey!
Fish id = 91 is Dead!
Shark id = 9 ate the prey!
Fish id = 22 is Dead!
Fish id = 60 is Dead!
Shark id = 5 ate the prey!
```

5. Experience

✧ 1. What have I learned?

During this assignment, I have learned a lot about MPI, and dealing with the memory architecture provided by such library. There were lots of works had to be done when applying MPI, since it's distributed memory, which forced me to be cautious while transmitting and receiving parameters. I have learned more about how to control the slaves and the strength of the MPI_ANY_TAG. Also I studied the PPT provided about the xwindow and found out this feature was pretty cool, if only I can master it better!

✧ 2. Any difficulties?

At first, it sound quite easy to me when hearing this task, but I found out I was extremely wrong, or maybe it was because I decided to use MPI instead of shared memory architecture. Even a tiny loss of required information could cause fatal failure, and it's hard to debug, especially when the program is as large as this one. So it took me for a long while to trace back every single detail whenever bugs occurred. It's quite annoying because sometimes I didn't even know where the problem was, and thus I thought to myself that I would definitely apply pthread next time when dealing with these kinds of problems.