Game of Life

Courtesy: Dr. David Walker, Cardiff University

A Dynamical System - WaTor

Courtesy: Dr. David Walker, Cardiff

- Tracking evolution of life
- A 2-D ocean in which sharks and fish survive
- 2 important features

a. Potential conflicts due to updates by different processors

b. Need for dynamic load distribution

- Features shared by other advanced parallel applications

WaTor – The problem

- Ocean divided into grids
- Each grid cell can be empty or have a fish or a shark
- Grid initially populated with fishes and sharks in a random manner
- Population evolves over discrete time steps according to certain rules

WaTor - Rules

Fish:

- At each time step, a fish tries to move to a neighboring empty cell. If not empty, it stays
- If a fish reaches a breeding age, when it moves, it breeds, leaving behind a fish of age 0. Fish cannot breed if it doesn't move.
- Fish never starves

WaTor - Rules

Shark:

- At each time step, if one of the neighboring cells has a fish, the shark moves to that cell eating the fish. If not and if one of the neighboring cells is empty, the shark moves there. Otherwise, it stays.
- If a shark reaches a breeding age, when it moves, it breeds, leaving behind a shark of age 0. shark cannot breed if it doesn’t move.
- Sharks eat only fish. If a shark reaches a startvation age (time steps since last eaten), it dies.

Inputs and Data Structures

Inputs:
- Size of the grid
- Distribution of sharks and fishes
- Shark and fish breeding ages
- Shark starvation age

Data structures:

A 2-D grid of cells
```
  struct ocean{
    int type /* shark or fish or empty */
    struct swimmer* occupier;
  }ocean[MAXX][MAXY]
```
A linked list of swimmers
```
  struct swimmer{
    int type;
    int x,y;
    int age;
    int last_ate;
    int iteration;
    swimmer* prev;
    swimmer* next;
  } *List;
```

Sequential Code Logic

- Initialize ocean array and swimmers list
- In each time step, go through the swimmers in the order in which they are stored and perform updates

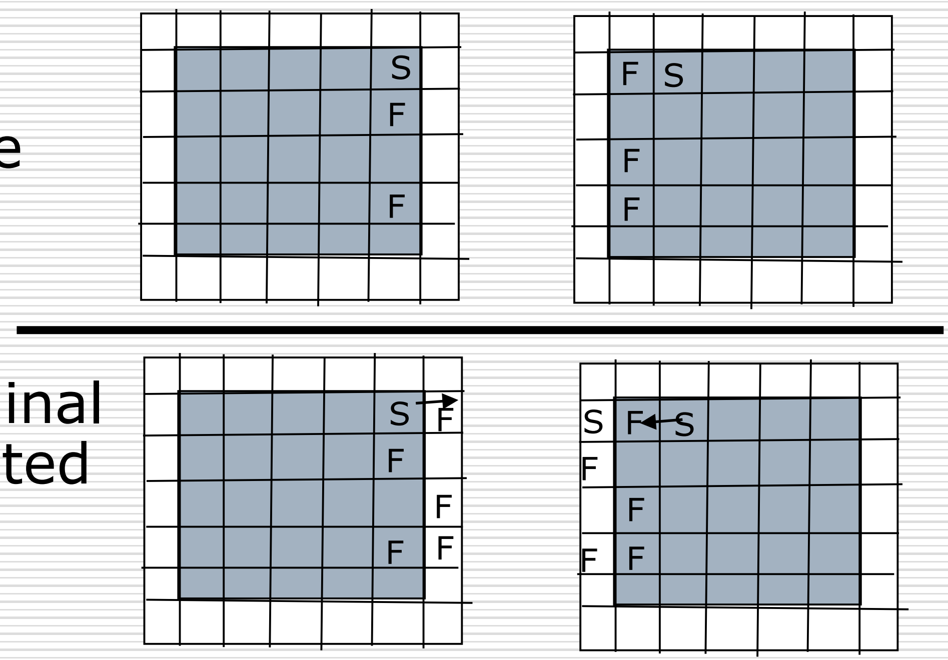
Towards a Parallel Code

- 2-D data distribution similar to Laplace and molecular dynamics is used. Each processor holds a grid of ocean cells.
- For communication, each processor needs data from 4 neighboring processors.
- 2 new challenges – potential for conflicts, load balancing

1^{st} Challenge – Potential for Conflicts

- Unlike previous problems, border cells may change during updates due to fish or shark movement
- Border cells need to be communicated back to the original processor. Hence update step involves communication
- In the meantime, the original processor may have updated the border cell. Hence potential conflicts

Time T



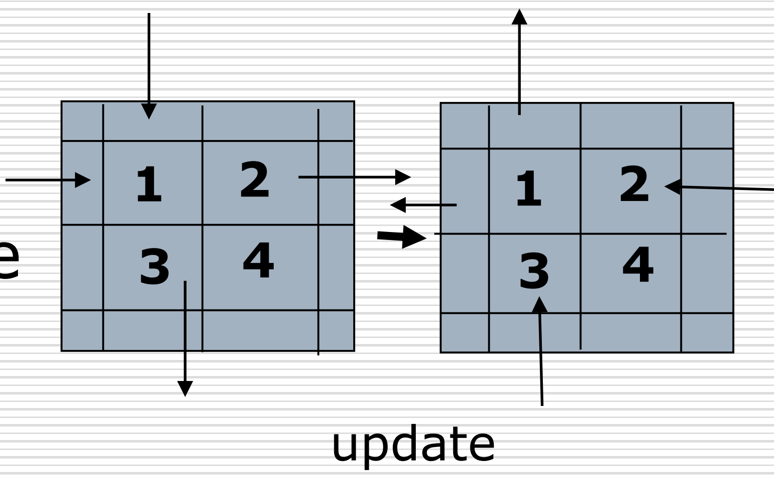
Time T+1

2 Techniques

- **Rollback** updates for those particles (fish or shark) that have crossed processor boundary and are in potential conflicts.
- May lead to several rollbacks until a free space is found.
- 2^{nd} technique is **synchronization** during updates to avoid conflicts in the first place.

2 Techniques

- During update, a processor x sends its data first to processor y, allows y to perform its updates, get the updates from y, and then performs its own updates.
- Synchronization can be done by sub-partitioning.
- Divide a grid owned by a processor into sub-grids.
- This way, some parallelism is achieved in neighbor updates



Load Imbalance

- The workload distribution changes over time
- 2-D block distribution is not optimal

Techniques:

- Static load balancing by a different data distribution
- Dynamic load balancer

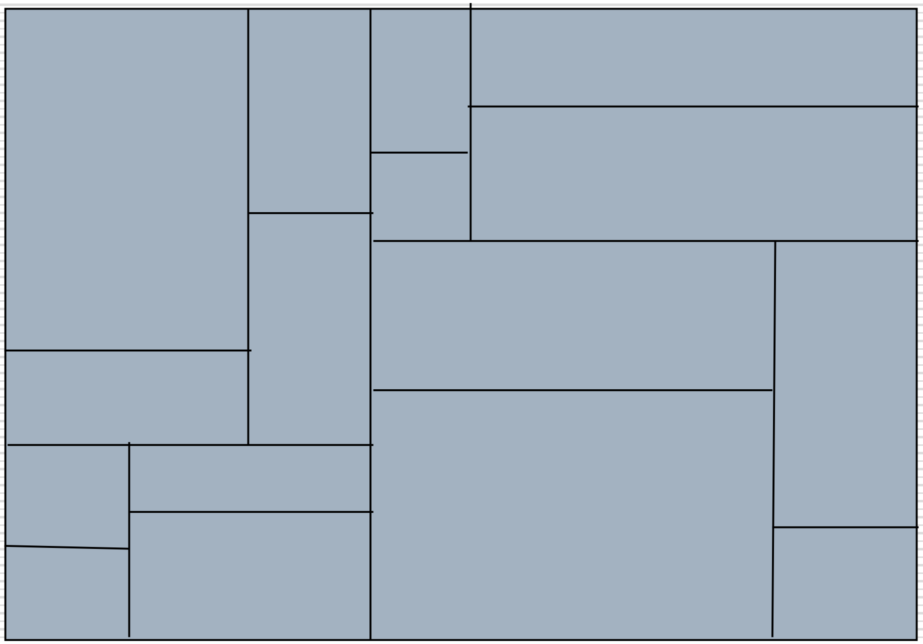
Static Data Distribution

- Using cyclic or block-cyclic

Problems: Increase in boundary data; increase in communication

Dynamic load balancing

- Performed at each time step
- Orthogonal Recursive Bisection (ORB)



Problems:

Complexity in finding the neighbors and data for communication

- END

Dynamic Load Balancing