Day 6:
High Performance Computing
CMT106

David W. Walker

Professor of High Performance Computing

Cardiff University

http://www.cardiff.ac.uk/people/view/118172-walker-david

Day 6

- 9:30 – 10:30am: Lecture covering dynamic communication and the molecular dynamics example (MPI).
- 10:30 – 10:50am: Break.
- 10:50am – 12:00pm: Lecture on molecular dynamics simulation with OpenMP and CUDA.
- 12:00 – 1:30pm: Lunch break.
- 1:30 – 2:00pm: Lecture, introduction to the Wator code.
- 2:00 – 3:00pm: Lab session, try out the Wator example code yourself and consider how to parallelize it (includes 15min break).
- 3:00 – 3:15pm: Review of the lab session.
- 3:15pm – 4:00pm: Lecture, parallel implementation of the Wator code and other cellular automata.
- 4:00pm – 5:00pm: Self study, do your coursework.

Topics Covered on Days 1-4

- *Day 1:* Introduction to parallelism; motivation; types of parallelism; Top500 list; classification of machines; SPMD programs; memory models; shared and distributed memory; OpenMP; example of summing numbers.
- *Day 2*: Interconnection networks; network metrics; classification of parallel algorithms; speedup and efficiency.
- *Day 3*: Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI; collective communication; integration example.
- *Day 4:* Regular computations and simple examples – the wave equation and Laplace's equation.

Topics Covered on Days 5-7

- *Day 5*: Programming GPUs with CUDA; CUDA device memory architecture; simple programming examples.
- *Day 6*: *Dynamic communication and the molecular dynamics example; irregular computations; the WaTor simulation.*
- *Day 7*: Load balancing strategies; message passing libraries; block-cyclic data distribution.

Irregular Communication

- In the wave equation and Laplace equation problems the communication is very *regular*. Once we set the number of processes and the size of the problem the communication requirements of the algorithm are fully determined.
- We shall now consider a parallel molecular dynamics simulation. In this simulation we know that data may need to be communicated between processes at a particular point in the program, but we do not know which data it will be. In this example the communication is slightly *irregular*.

Molecular Dynamics Simulations

- We have n particles in a periodic square domain.
- The particles interact in a known pairwise way. Each particle exerts a force on the other particles so that
  - if the particles are close enough they repel each other
  - if the particles are far enough apart they are attracted to each other
  - if the particles are more than some distance, r_0, apart they do not influence each other.

Molecular Dynamics Simulations 2

- This sort of interaction is typical of many molecules.
- Given initial positions and velocities for the particles, we follow the movement of the particles at a series of discrete time steps.
- Usually we are interested in the macroscopic properties of such particle systems, such as temperature, energy, etc.

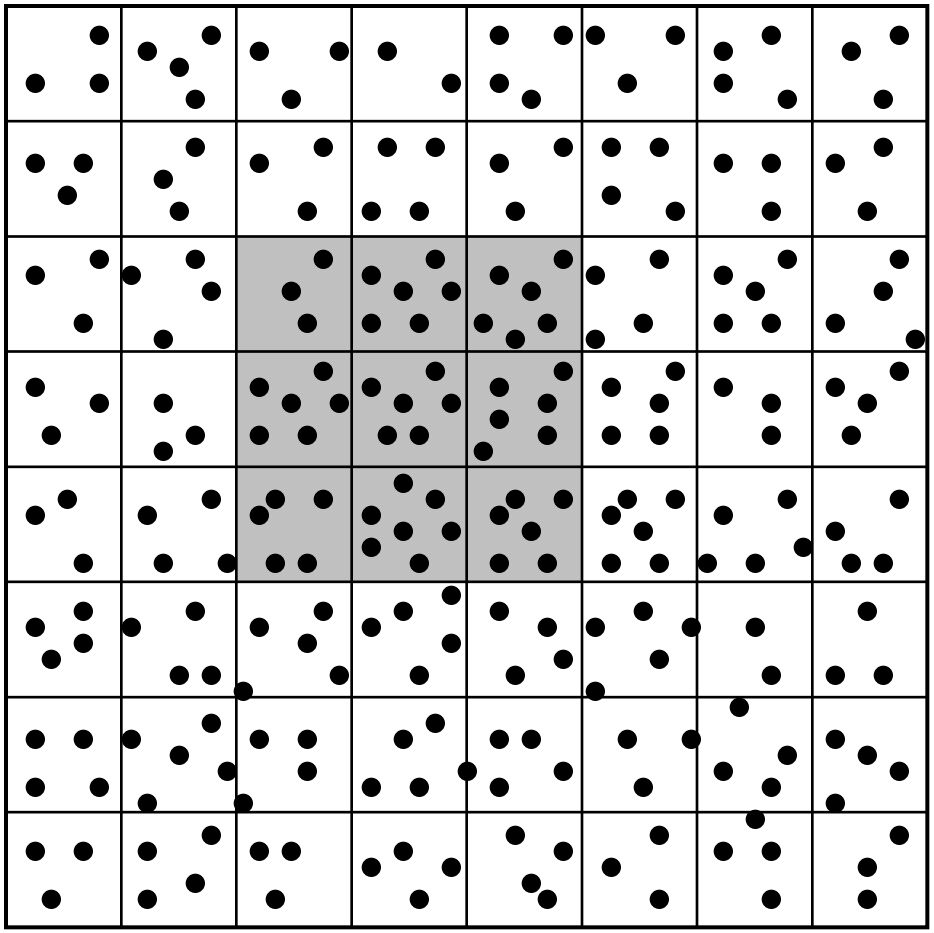
Cut-Off Distance

- We could find the force on particle i by summing over all the other particles,

$$F_i = \sum_{j=0}^{n-1} f_{ij}$$

  where $\mathbf{f}_{ij}$ is the force exerted by particle j on particle i. This results in an $O(n^2)$ algorithm.
- We can improve the running time if we make use of the fact that the force $\mathbf{f}_{ij}$ is zero for particles more than distance r_0 apart.

Cut-Off Distance 2



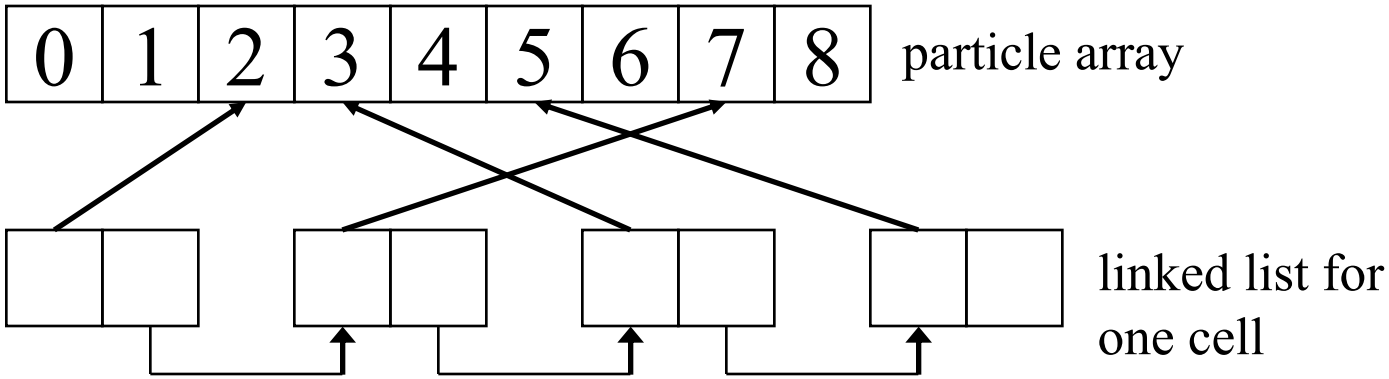
If we divide the domain of the problem into cells of size r_0 x r_0 each particle only interacts with the particles in its own cell and the 8 neighbouring cells.

Data Structures

- The particle data structure contains a particles position, velocity, and other data such as particle type, etc.
- Particles can be stored in an array.
- The cell data structure contains a list of the particles it contains, i.e., a list of the array index for each of its particles.
- The cell data can be stored as a linked list.
- There is a 2D array of cells.

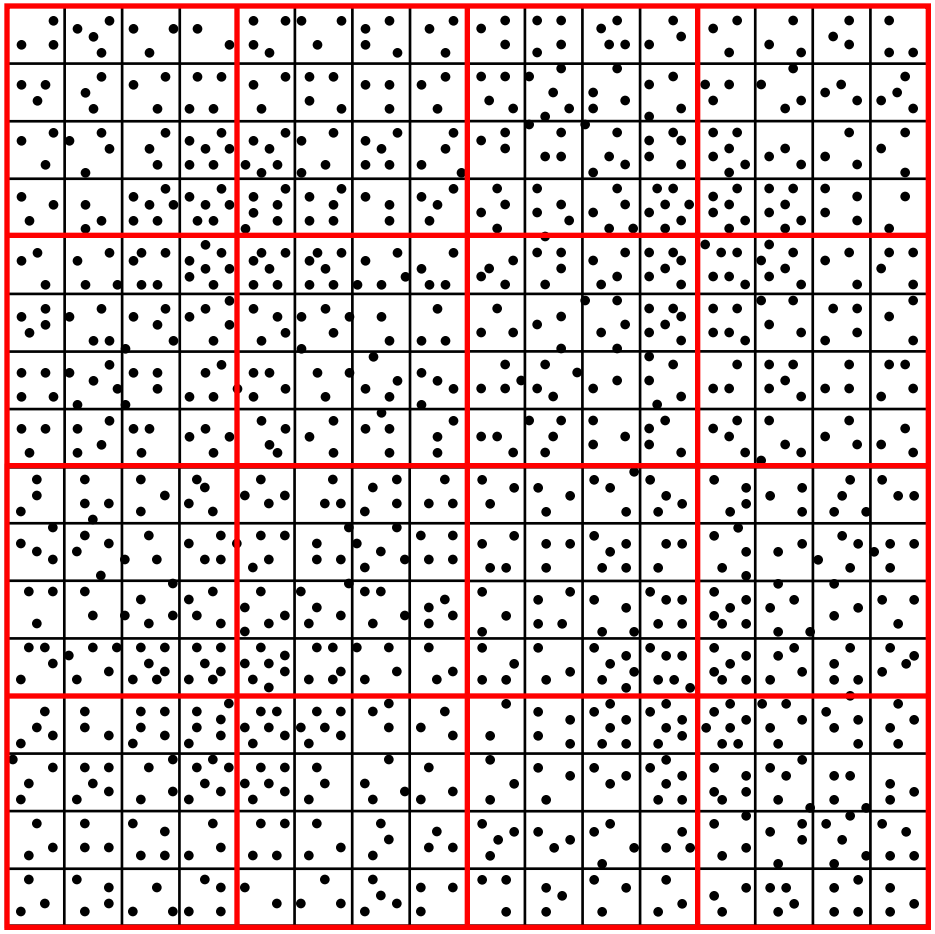
Cell Data Structure

- We can use a linked list to keep track of the particles in each cell.



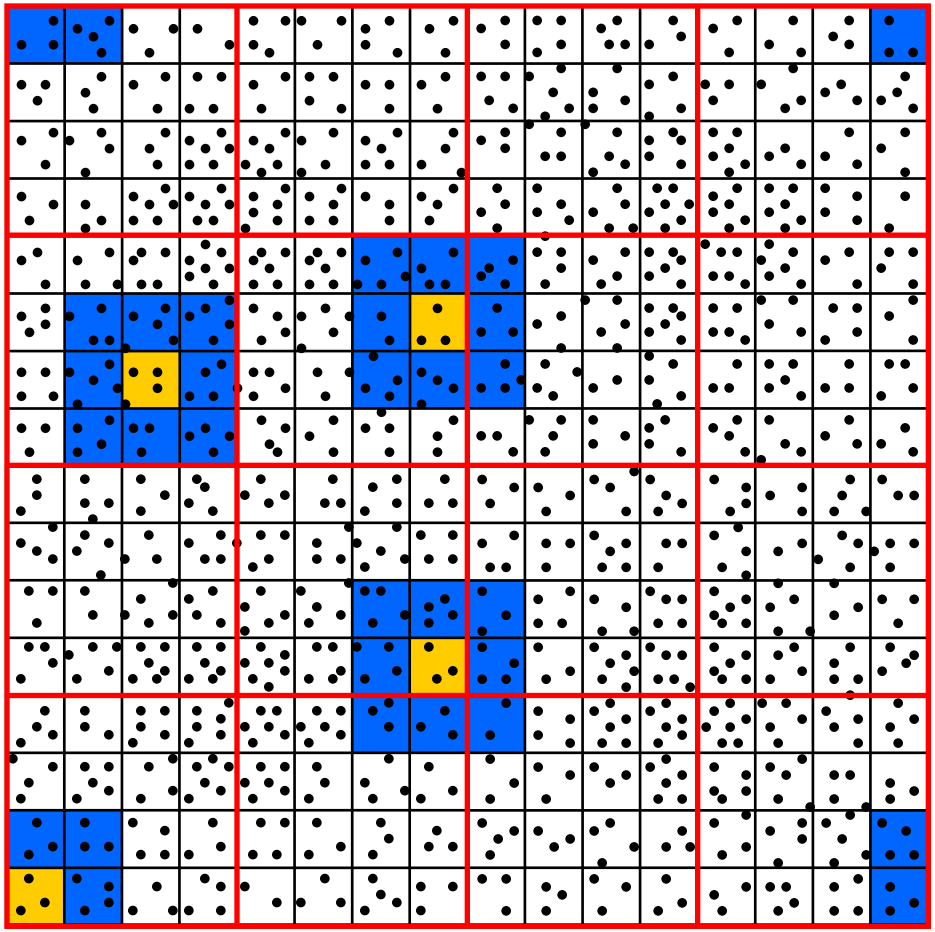
- Each cell has its own list. The starts of the lists are stored in a 2D array.
- A particle can find out which cell it is in from its position.

Data Distribution (MPI version)



- The particles are distributed to processes by assigning a rectangular block of cells to each process.
- We find out the node number, location in the process mesh, and the node numbers of the neighbouring processes as in the Laplace equation problem.

Data Dependencies



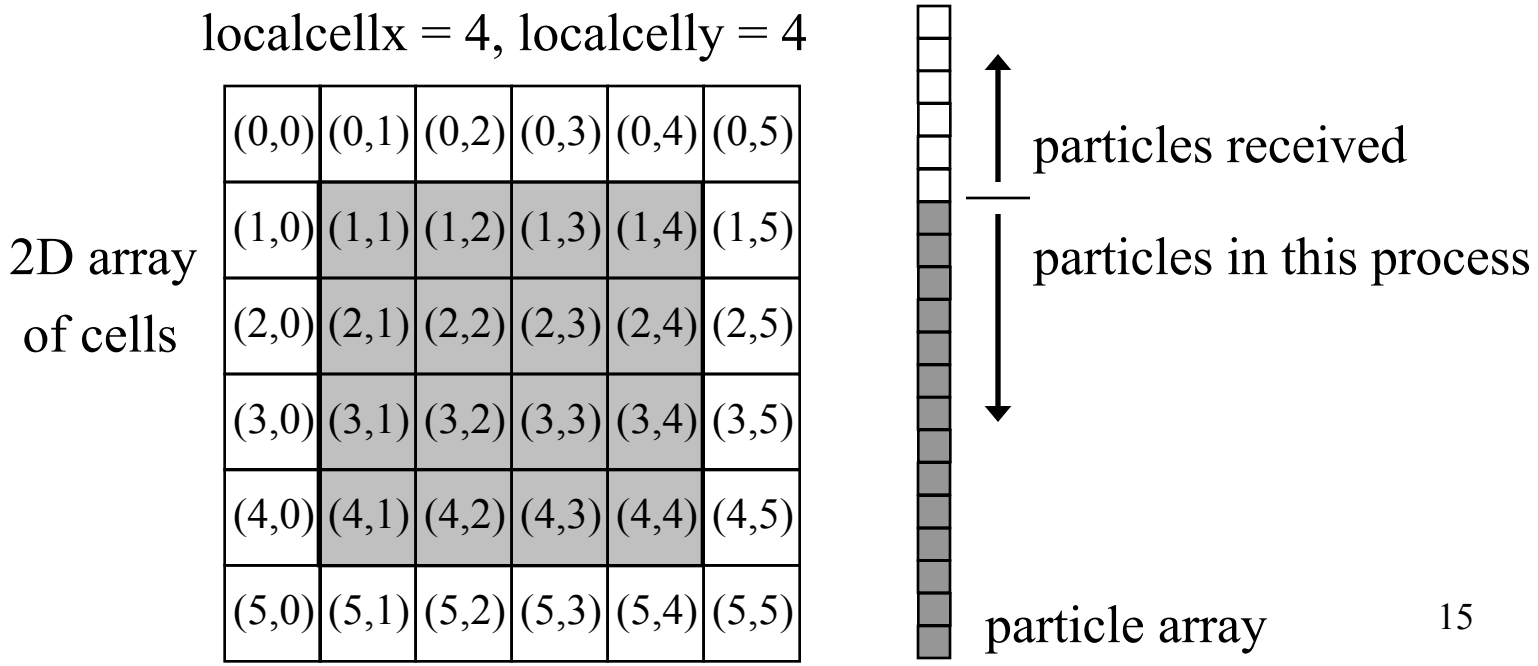
- Particles in cells along the boundary of a process need to know about particles in other processes in order to evaluate the force on them.
- Each process needs particle data from 8 neighbouring processes

Communication Requirements

- Each particle needs information about the particles in the neighbouring cells in order to determine the force on it. So we need to communicate particles lying in cells along the boundary of each process
- When particles move they may travel from the set of cells owned by one process to those of another process. This is called *particle migration* and requires communication.

Array Declarations

- We maintain a 2D array of linked lists – one for each cell in a process.
- When we receive particle information from cells lying along the boundary of adjacent processes we store the data at the end of the particle array.
- Pointers to the particles received are placed into cell lists.



Outline of Parallel Code

- **Initialise data distribution**
  - Find position of each process to determine which block of cells it handles.
  - Find out the node numbers of processes in the left, right, up, and down directions.
- **Initialise particle arrays and cell lists**
  - Generate or input initial particle positions and velocities .
  - Insert particles into cell lists.
- **Perform update**
  - Communicate boundary cell data.
  - Do update locally.
  - Communicate particles that have migrated
- **Output results**

Outline MPI Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
    MPI_Init (&argc, &argv);
                    <-- Code for initialising data distribution goes here
                    <-- Code for initialising arrays and lists goes here
                    <-- Code for update loop goes here
                    <-- Code for output phase goes here
    MPI_Finalize();
  }
}
```

Initialising the Data Distribution

```
int rank, nprocs, mypos;
int nprocx, nprocy, periods[2], dims[2], coords[2], reorder=0;
MPI_Comm new_comm;

MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

dims[0] = dims[1] = 0;
MPI_Dims_create (nprocs, 2, dims);
nprocy = dims[0];
nprocx = dims[1];
periods[0] = periods[1] = 1;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, 1,&new_comm);
MPI_Cart_coords (new_comm, rank, 2, coords);
myposy = coords[0];
myposx = coords[1];
MPI_Cart_shift  (new_comm, 0, 1, &down, &up);
MPI_Cart_shift  (new_comm, 1, 1, &left, &right);
```

Parallel Update Loop

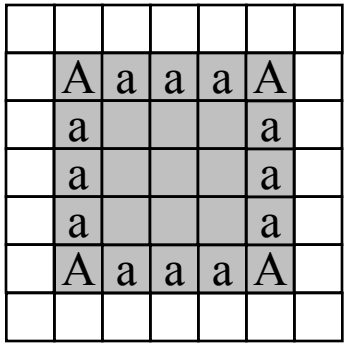
```
for (each time step){
    communicate particle data for boundary cells
    for (each particle, p, in this process){
        find out the location (i,j) of cell p is in
        force[p] = 0;
        for (cell (i,j) and the 8 neighbouring cells){
            for (each particle q in cell){
                add force of q on p to force[p]
            }
         }
    }
    for (each particle, p)
        update velocity and position of p using force[p]
    }
    migrate particles
}
```

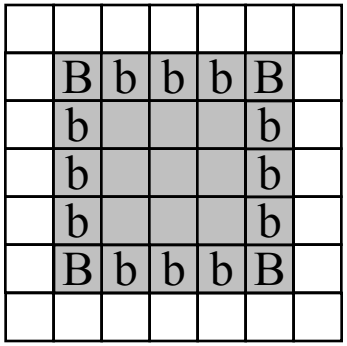
Communication of Boundary Cell Data

- The communication is similar to that for the Laplace equation problem except
  - We have to look in the cell lists to see which particles have to be sent and pack this information into a send buffer.
  - The receiving process does not know beforehand how many particles it is going to receive.
  - We have to communicate between diagonally adjacent processes.

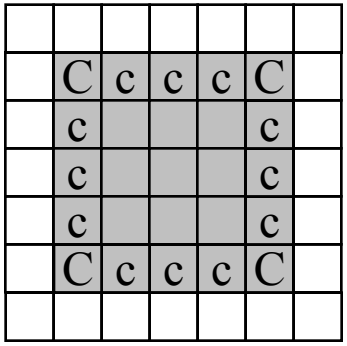
Communicating Corner Data

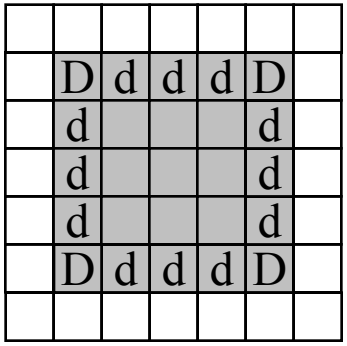
We need to communicate particles in corner cells to diagonally adjacent processes. This can be done be done in 4 shift operations.



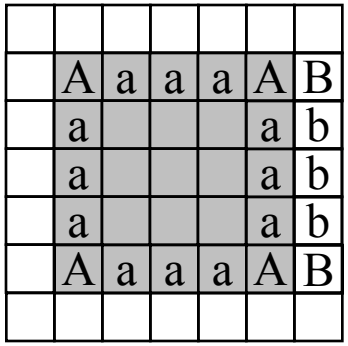


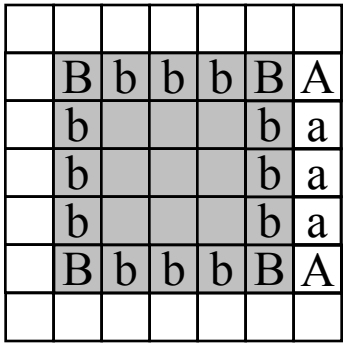
Initial state for 2x2 mesh of processes

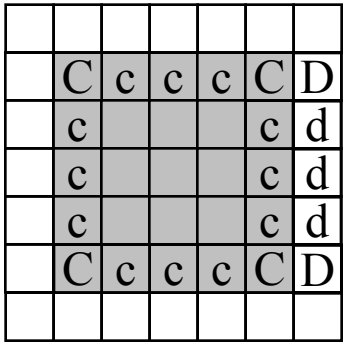


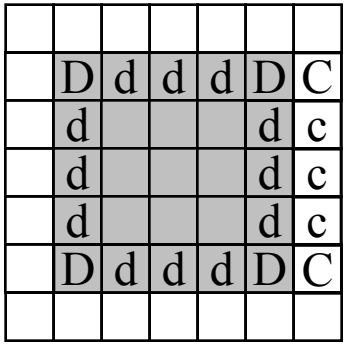


Left Shift

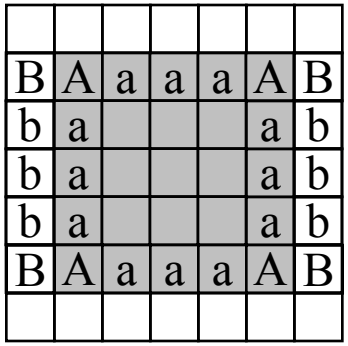

A a a a A B
a a
a a
a a
a a
A a a a A B

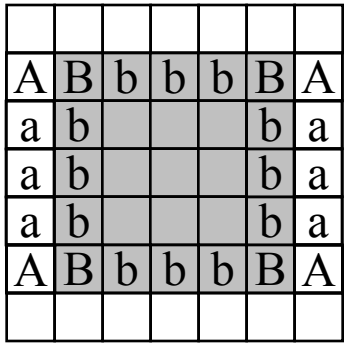

B b b b B A
b b
b b
b b
b b
B b b b B A

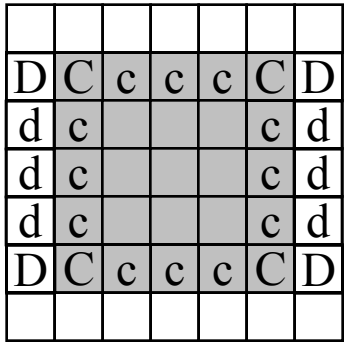

C c c c C D
c c
c c
c c
c c
C c c c C D

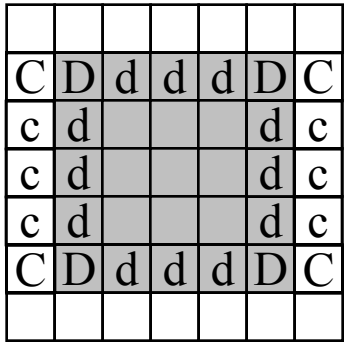

D d d d D C
d d
d d
d d
d d
D d d d D C

Right Shift

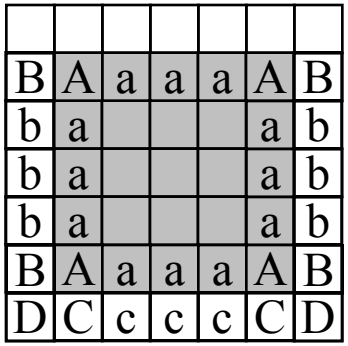

B A a a a A B
b a a
b a a
b a a
b a a
B A a a a A B


A B b b b B A
a b b
a b b
a b b
a b b
A B b b b B A

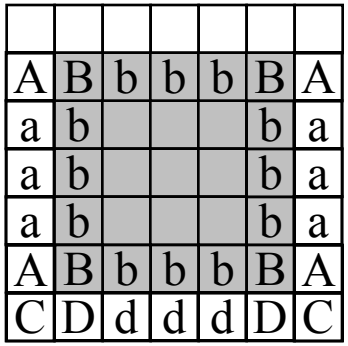

D C c c c C D
d c c
d c c
d c c
d c c
D C c c c C D


C D d d d D C
c d d
c d d
c d d
c d d
C D d d d D C

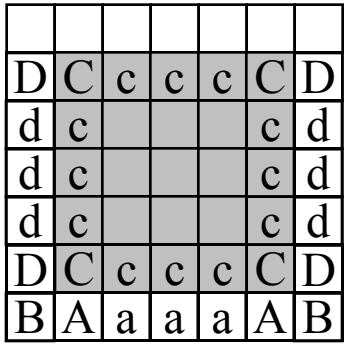
Up Shift



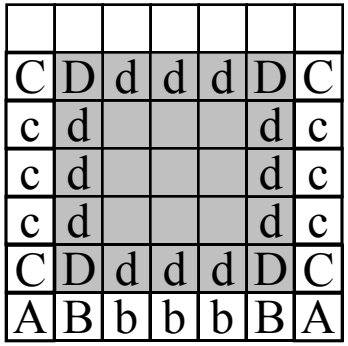
B	A	a	a	a	A	B
b	a				a	b
b	a				a	b
b	a				a	b
B	A	a	a	a	A	B
D	C	c	c	c	C	D



A	B	b	b	b	B	A
a	b				b	a
a	b				b	a
a	b				b	a
A	B	b	b	b	B	A
C	D	d	d	d	D	C

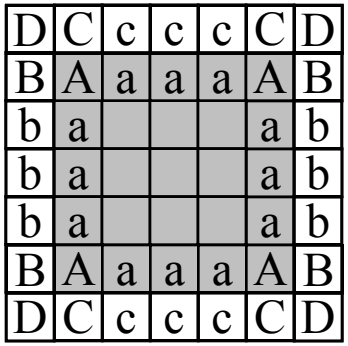


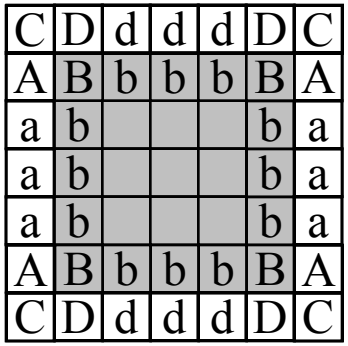
D	C	c	c	c	C	D
d	c				c	d
d	c				c	d
d	c				c	d
D	C	c	c	c	C	D
B	A	a	a	a	A	B

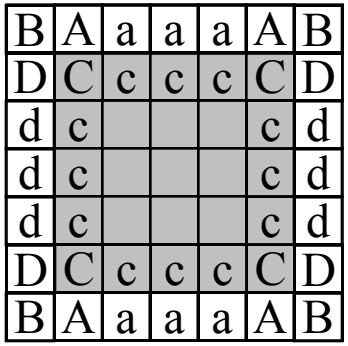


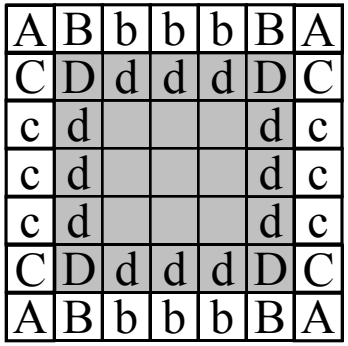
C	D	d	d	d	D	C
c	d				d	c
c	d				d	c
c	d				d	c
C	D	d	d	d	D	C
A	B	b	b	b	B	A

Down Shift


D C c c c C D
B A a a a A B
b a a A b
b a a A b
b a a A b
B A a a a A B
D C c c c C D


C D d d d D C
A B b b b B A
a b b A
a b b A
a b b A
A B b b b B A
C D d d d D C


B A a a a A B
D C c c c C D
d c c d
d c c d
d c c d
D C c c c C D
B A a a a A B


A B b b b B A
C D d d d D C
c d d c
c d d c
c d d c
C D d d d D C
A B b b b B A

Pseudocode for Left Shift

```
nsend = 0
for (i=1 to localcelly){
    for (each particle, p, in cell (i,1))
        pack position of p into sendbuf
        nsend = nsend + 2;
    }
}
MPI_Sendrecv (sendbuf, nsend, MPI_DOUBLE, left, tag,
            recvbuf, rbufsize, MPI_DOUBLE, right, tag,
            new_comm, &status);
MPI_Get_count (status, MPI_DOUBLE, &nrecv);
nL = nrecv/2;
for (i=1 to nL){
    take next 2 numbers from recvbuf, store in x and y
    set position of particle npart+i-1 to (x,y)
    find out which cell (x,y) is in
    add particle npart+i-1 to list for that cell
}
```

Pseudocode for Up Shift

```
nsend = 0
for (i=0 to localcellx+1){
    for (each particle, p, in cell (1,i))
        pack position of p into sendbuf
        nsend = nsend + 2;
    }
}
MPI_Sendrecv (sendbuf, nsend, MPI_DOUBLE, up, tag,
            recvbuf, rbufsize, MPI_DOUBLE, down, tag,
            new_comm, &status);
MPI_Get_count (status, MPI_DOUBLE, &nrecv);
nUp = nrecv/2;
for (i=1 to nUp){
    take next 2 numbers from recvbuf, store in x and y
    set position of particle npart+nL+nR+i-1 to (x,y)
    find out which cell (x,y) is in
    add particle npart+nL+nR+i-1 to list for that cell
}
```

Particle Migration

- We assume that a particle stays in the same cell or moves to one of the 8 adjacent cells.
- This may involve moving to a cell in another process.
- We have to be able to handle the case where particles move to diagonally adjacent processes.

Pseudocode for Particle Migration 1

```
for (each particle, p){
    update position and velocity
    determine which cell p is in
    if (p has moved to new cell){
       delete p from list of old cell
       if (p has moved to different process){
          put p into appropriate communication buffer
          remove p from particle array
       }
       else{
          add p to list of new cell
       }
    }
 }
 shift left
 shift right
 shift up
 shift down
```

Pseudocode for Particle Migration 2

After receiving particle data from another process, a process must determine if the particle belongs to it, or if it has to be passed on to another process. For left shift:

```
MPI_Sendrecv(leftbuf, nsendleft, MPI_DOUBLE, left, tag,
             recvbuf, rbufsize, MPI_DOUBLE, right, tag,
             new_comm, &status);

MPI_Get_count(status, MPI_DOUBLE, &nrecv);
for (i=1 to nrecv in steps of 4){
    get next 4 numbers from recvbuf, store in x, y, vx, vy
    if (particle belongs in this process){
       add particle to end of particle array
       find out what cell particle is in
       add particle to list for that cell
    }
    else{
       put particle in appropriate communication buffer
    }
}
```

Notes on Parallel Molecular Dynamics Simulations

- As in previous examples we need to exchange "boundary" data between processes.
- Need to communicate with diagonally adjacent processes. This can be done with four shift operations.
- We do not know beforehand how many particles will be communicated between processes in the boundary data exchange or migration steps. The receiving process determines this with MPI_Get_count().
- In general each process holds a different number of particles and this changes over time. However, we don't expect load imbalance to be too bad because particles tend to be evenly distributed in space.

Parallel Molecular Dynamics with OpenMP

- Most computational effort is in finding the force on each particle.

```
    for (each time step){
        for (each particle, p){
            find out the location (i,j) of cell p is in
            force[p] = 0;
            for (cell (i,j) and the 8 neighbouring cells){
                for (each particle q in cell){
                    add force of q on p to force[p]
                }
             }
        }
        for (each particle, p)
            update velocity and position of p using force[p]
        }
}
```

OpenMP Code (2D)

```
    for (int k=0;k<nsteps;k++){
#pragma omp parallel default(shared) private(p,cell,row,col,i,j) num_threads(5)
{
#pragma omp for schedule(static,chunk)
        for (int p=0;p<npart;p++){
            cell = get_cell(pdata[p], r0);
            pdata[p].fx = pdata[p].fy = 0;
            col = cell.c; row = cell.r;
            for (j=row-1;j<=row+1;j++)
                for(i=col-1;i<=col+1;i++)
                    add_force(pdata, cells[j][i], p);
        }
}
        for (int p=0;p<npart;p++)){
            update_particle(pdata[p], dt);
        }
}
```

Parallel Molecular Dynamics with CUDA (2D)

- CUDA cannot dynamically allocate space in device memory, so this complicates the use of linked lists for CUDA codes.
- Implement linked list with 3 arrays:
  - cell[i][j] gives array index of first particle in list for cell at (i,j), or -1 if empty.
  - list[k] gives array index of next particle in list after particle k, or -1 if k is last particle.

Processing Particles in Cell at (i,j)

```
   p = cell[i][j];
   while(p != -1){
//       Process particle p
         :
         :
      p = list[p];
   }
```

Allocating the Device Arrays

```
float *rxd, *ryd, *vxd, *vyd, *fxd, *fyd;
int *celld, *listd;
int nsize1 = nparts*sizeof(float);
cudaMalloc((void **)&rxd, nsize1);
cudaMalloc((void **)&ryd, nsize1);
cudaMalloc((void **)&fxd, nsize1);
cudaMalloc((void **)&fyd, nsize1);
…etc….
int nsize2 = ncells*sizeof(int);
cudaMalloc((void **)&celld, nsize2);
int nsize3 = nparts*sizeof(int);
cudaMalloc((void **)&listd, nsize3);
```

Copy Arrays from Host to Device

```
cudaMemcpy(rxd, rx, nsize1, cudaMemcpyHostToDevice);
cudaMemcpy(ryd, ry, nsize1, cudaMemcpyHostToDevice);
cudaMemcpy(vxd, vx, nsize1, cudaMemcpyHostToDevice);
cudaMemcpy(vyd, vy, nsize1, cudaMemcpyHostToDevice);
```

Setting Up the Thread Blocks

```
int nthreads = 512;
int nblocks = ceil(nparts/(float)nthreads);
dim3 dimBlock(nthreads,1,1);
dim3 dimGrid(nblocks,1,1);
```

CUDA Update Loop

```
createLists(rx, ry, nparts, cell, list, ncellx, ncelly);
for(k=1;k<=nsteps;k++){
      cudaMemcpy(celld, cell, nsize2, cudaMemcpyHostToDevice);
      cudaMemcpy(listd, list, nsize3, cudaMemcpyHostToDevice);
      ForceKernel<<<dimGrid, dimBlock>>>(rxd, ryd,fxd, fyd, celld, listd,
                                    nparts,  ncellx, ncelly,…input parameters…);
      UpdateKernel<<<dimGrid, dimBlock>>>(rxd, ryd, vxd, vyd, fxd, fyd,…
                                    input parameters…);
      cudaMemcpy(rx, rxd, nsize1, cudaMemcpyDeviceToHost);
      cudaMemcpy(ry, ryd, nsize1, cudaMemcpyDeviceToHost);
      createLists(rx, ry, nparts, cell, list, ncellx, ncelly);
}
```

CUDA Force Kernel

```
__global__ void ForceKernel(float *rxd, float *ryd, float *fxd, float *fyd, int *headd,
                            int *listd, int nparts, int ncellx, int ncelly,…..)
{
    int bno= blockIdx.x;
    int pno = bno*blockDim.x+threadIdx.x;
    float fxp = fyp = 0.0;
    if (pno<nparts) {
        rxp = rxd[pno]; ryp = ryd[pno];
        ….loop over cells and accumulate force on particle pno in fxp and fyp…,
        fxd[pno] = fxp;
        fyd[pno] = fyp;
    }
}
```

CUDA Update Kernel

```
__global__ void UpdateKernel(float *rxd, float *ryd, float *vx, float *vy,
                                        float *fx, float *fy, int nparts,…..)
{
      int bno= blockIdx.y*gridDim.x+blockIdx.x;
      int pno = bno*blockDim.x+threadIdx.x;
      if (pno<nparts) {
            rxp = rxd[pno]; ryp = ryd[pno]; vxp = vxd[pno]; vyp = vyd[pno];
            fxp = fxd[pno]; fyp = fyd[pno];
            ….update position and velocity for particle pno based on force on it…
            rxd[pno] = rxpnew; ryd[pno] = rypnew;
            vxd[pno] = vxpnew; vyd[pno] = vypnew;
      }
}
```

WaTor – a Dynamical System

- We shall now look at a very dynamic simulation called WaTor. WaTor is a periodic 2D ocean (hence, *Wa*tery *Tor*us) in which predators (sharks) and prey (fish) compete and survive.
- The parallel implementation of WaTor has a number of interesting features:
  - A very inhomogeneous and dynamic load distribution.
  - The need for irregular communication.
  - The possibility of conflicts between updates performed by different processes (data inconsistency).
- Other advanced parallel applications share some of these features.

The Rules of WaTor

- WaTor takes place on a periodic grid. Each grid cell either contains a fish, a shark, or is empty.
- The grid is initially populated by a specified number of fish and sharks, placed at random.
- The populations then evolve in a series of discrete time steps according to certain rules that govern how the fish and sharks move, breed, eat, and die.

Fish Rules

Moving:

- In each time step, each fish notes which of the 4 neighbouring sites are empty. One of these empty sites is chosen at random and the fish moves there. If there are no empty neighbouring sites the fish stays where it is.

Breeding:

- If a fish is past the fish breeding age, then when it moves it breeds, leaving a fish of age zero at its previous location. A fish cannot breed if it doesn't move.

Eating

- Fish eat plankton available throughout the ocean. Fish never starve.

Shark Rules

Moving

- In each time step, each shark notes which of the 4 neighbouring sites are occupied by fish. One of these sites is chosen at random and the shark moves there, eating the fish. If there are no neighbouring sites containing fish the shark notes which of the 4 neighbouring sites are empty and moves to one of these sites at random. If all 4 neighbouring sites are already occupied by sharks, the shark stays where it is.

Breeding

- If a shark is past the shark breeding age, then when it moves it breeds, leaving a shark of age zero at its previous location. A shark cannot breed if it doesn't move.

Eating

- Sharks eat only fish. If a shark does not eat for more than a certain number of time steps (known as the shark starvation age) then it dies.

WaTor Inputs

The inputs to the simulation are:

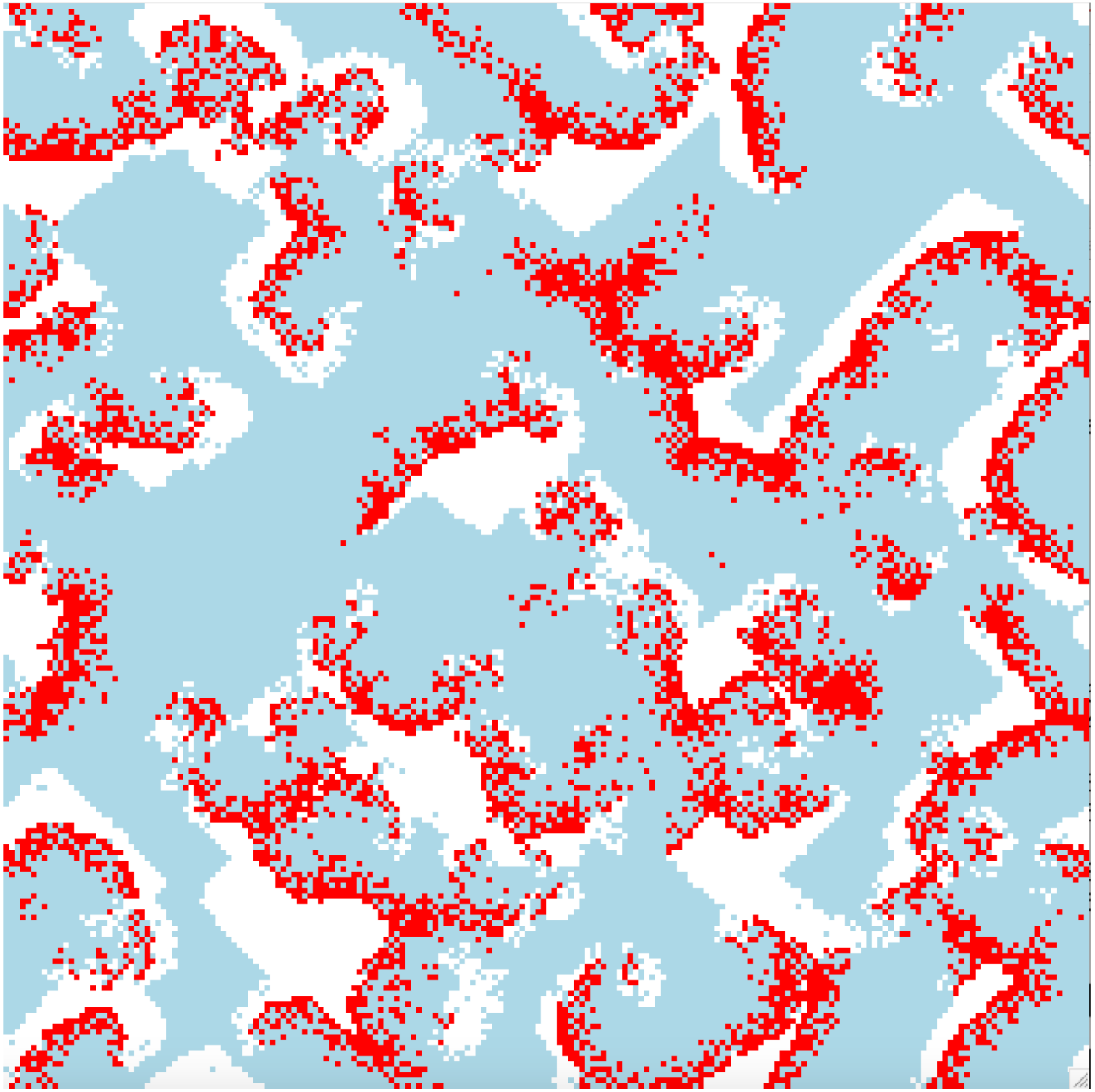
- The size of the grid.
- The initial number of sharks and fish (these are placed at random).
- The shark and fish breeding ages.
- The shark starvation age.

WaTor Data Structures

- The two fundamental data structures are the 2D grid of cells and a list of sharks and fish.
- Each cell in the grid is an object with an occupation status (empty, occupied by fish, or occupied by shark), and a reference to the fish or shark object it contains (if not empty).
- Each shark/fish object contains its type, age, cell location, and time since it last ate (if shark).

Update Order

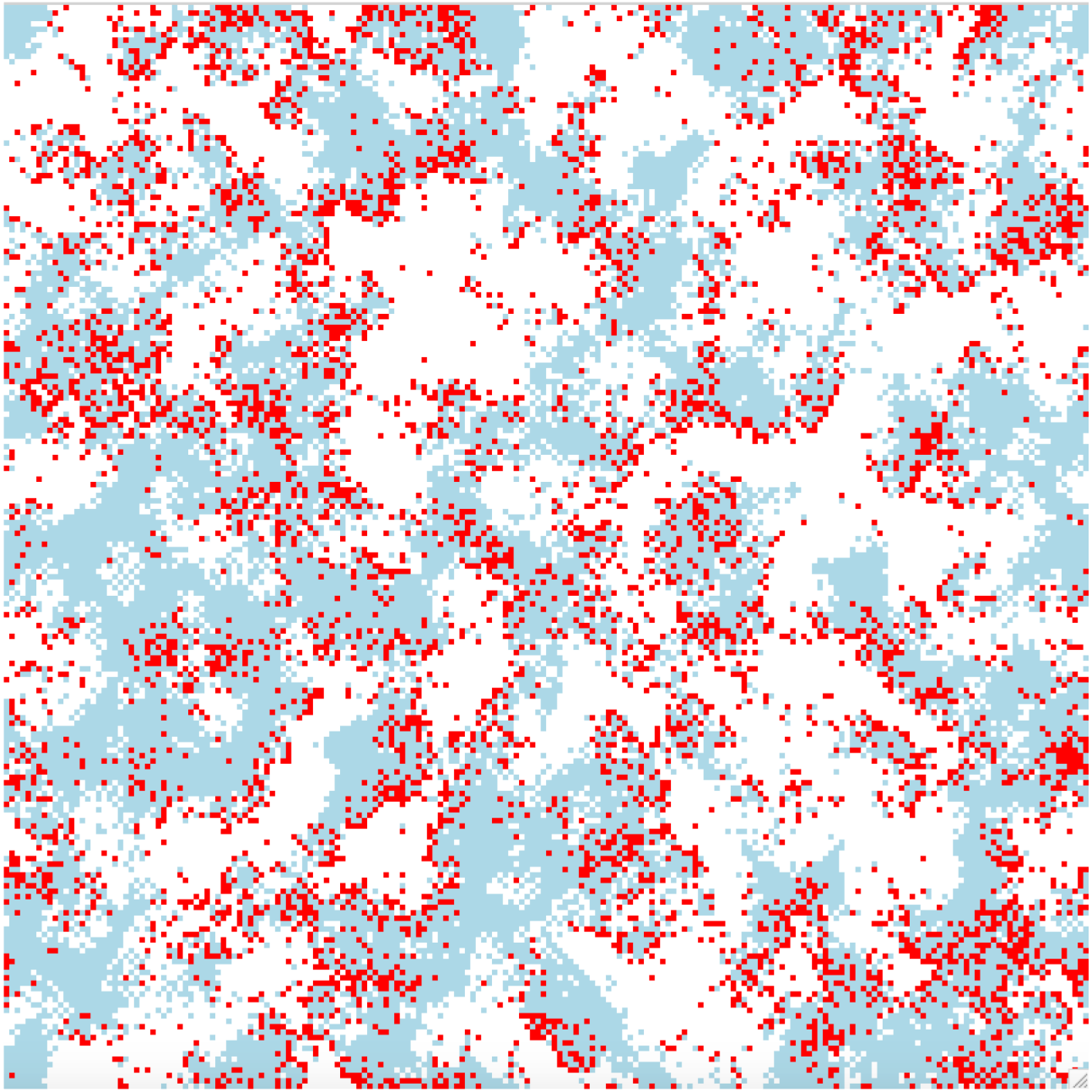
- The order of updating fish and sharks is not specified in the WaTor rules.
- Could update by looping over ocean cell locations. This is inefficient if a significant fraction of the ocean is empty.
- Alternative approach is to process the fish/shark list. In this case only active objects are processed.
- Must ensure that newly-born fish/sharks are not processed until the next time step.



Wator 1

200x200 ocean

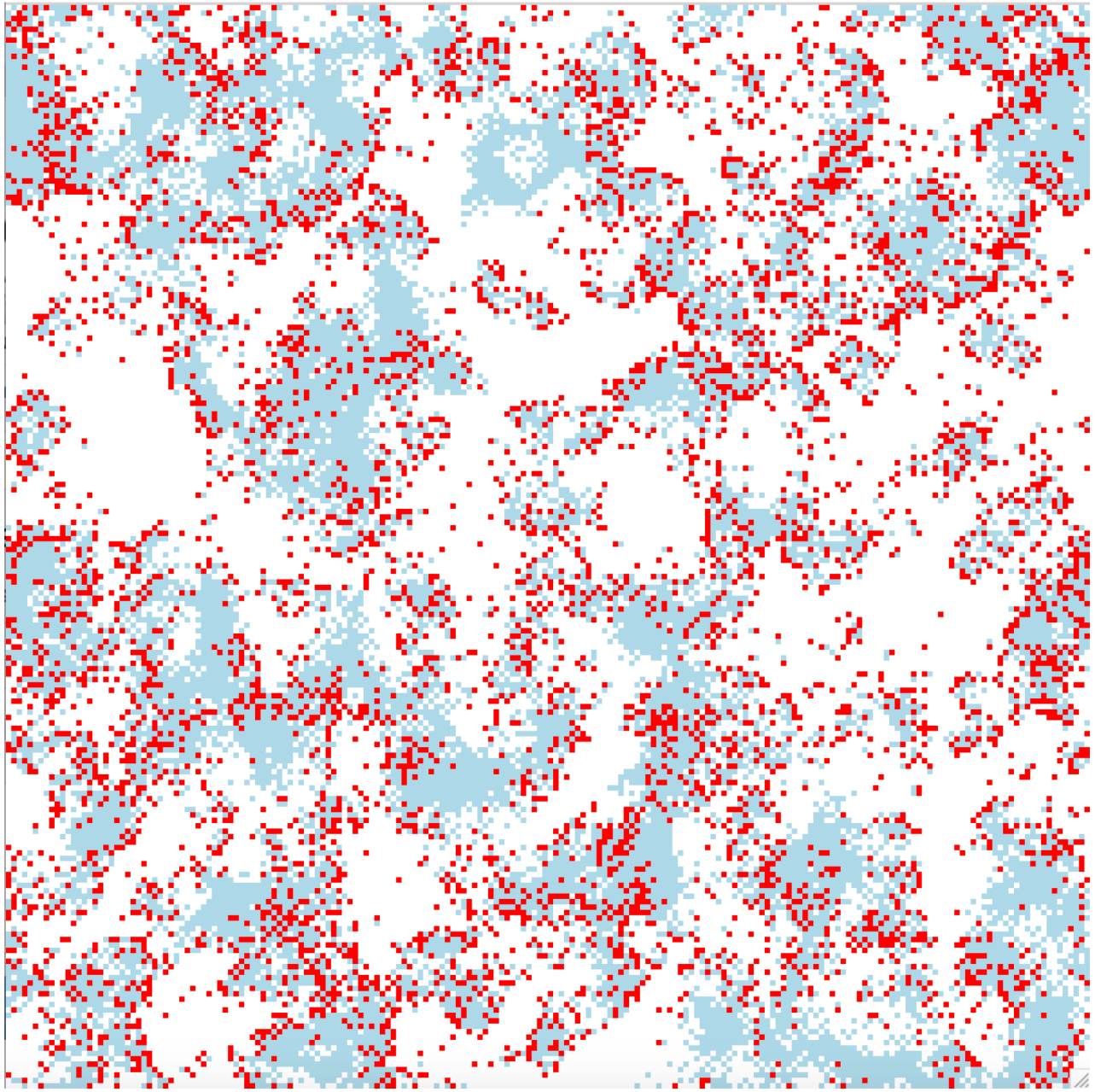
nfish	3500
nshark	10
Fish breeding age	4
Shark breeding age	5
Shark starvation age	4



Wator 2

200x200 ocean

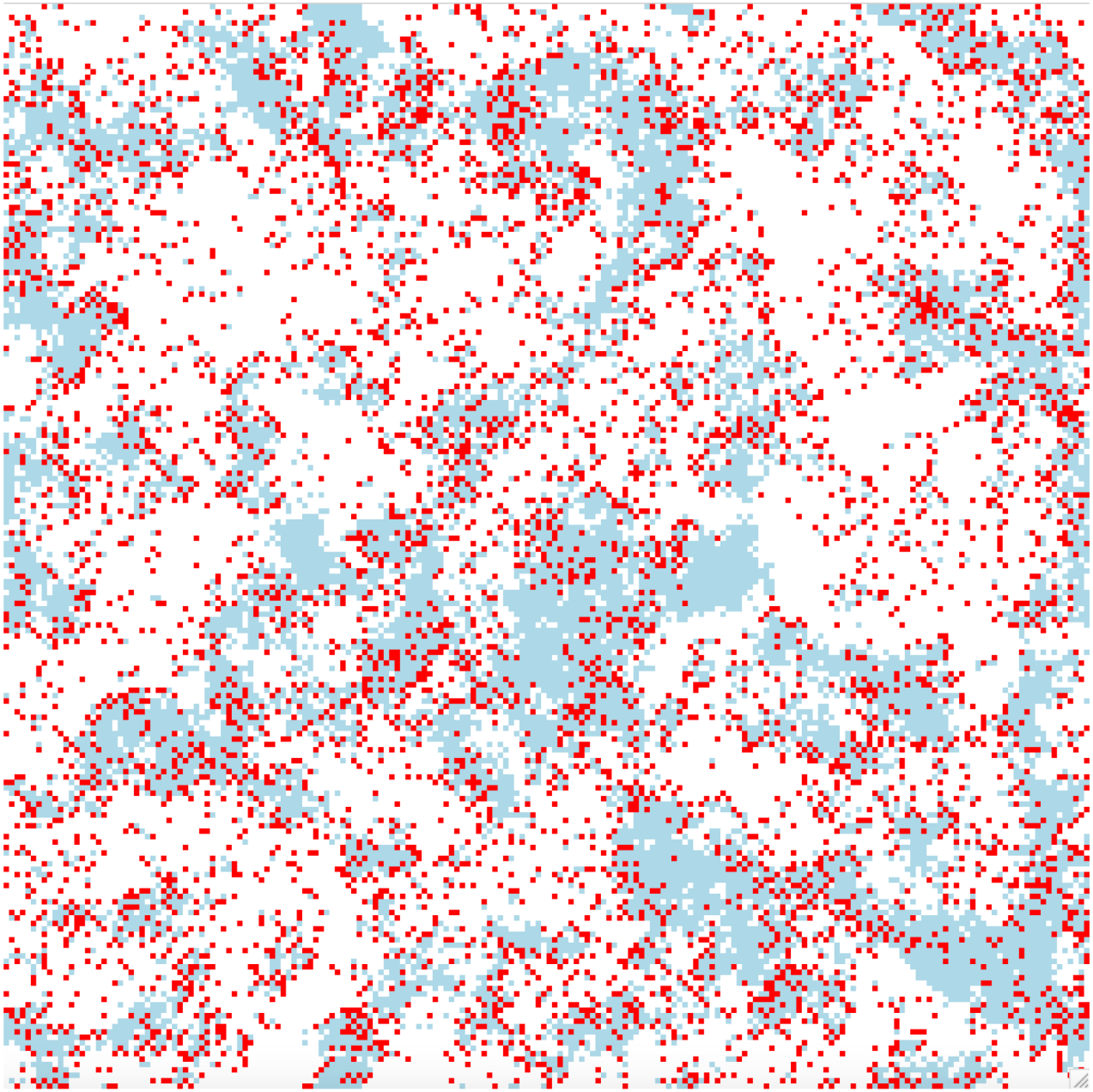
nfish	3500
nshark	10
Fish breeding age	4
Shark breeding age	30
Shark starvation age	4



Wator 3

200x200 ocean

nfish	3500
nshark	10
Fish breeding age	4
Shark breeding age	50
Shark starvation age	4



Wator 4

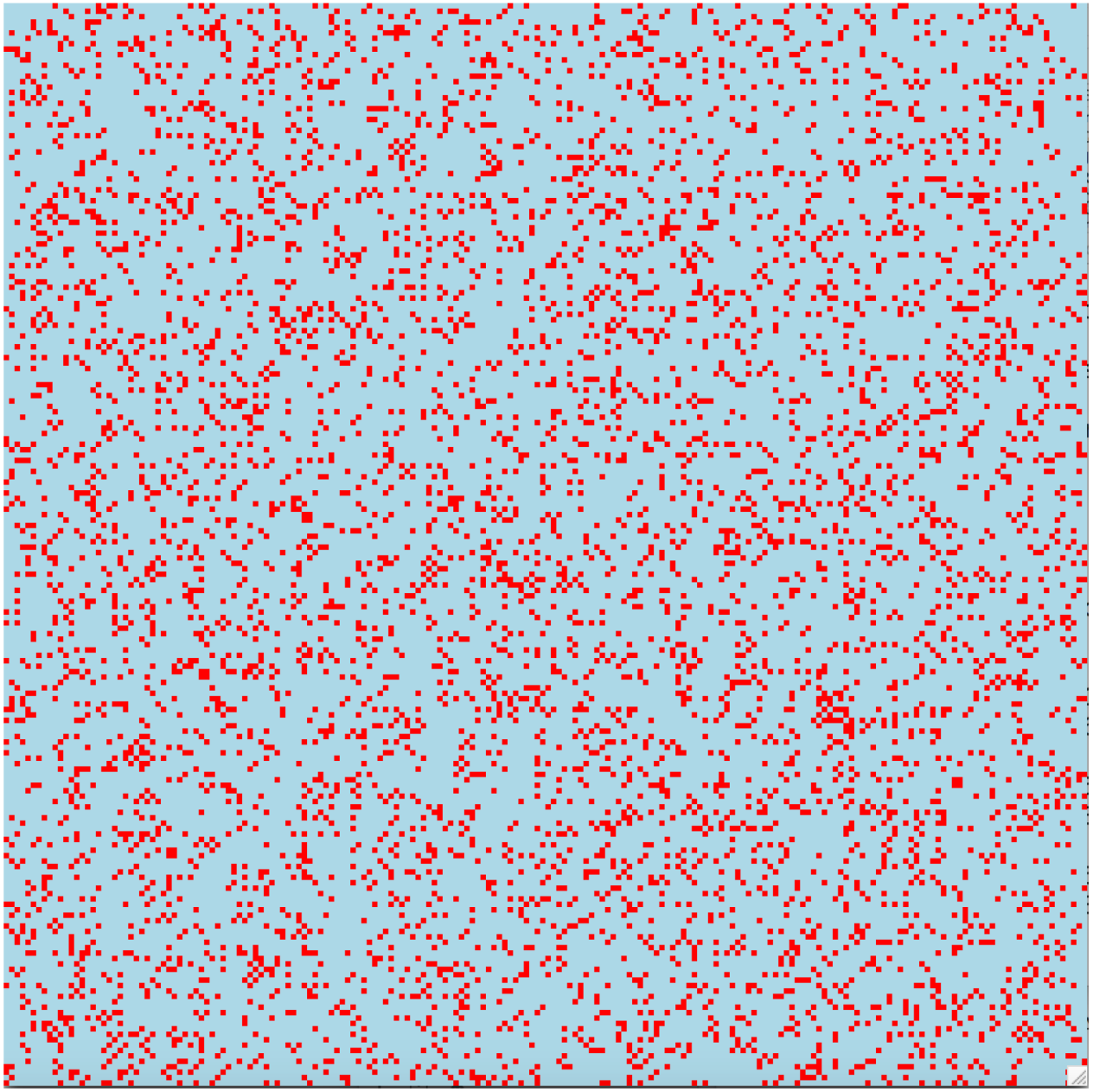
200x200 ocean

nfish	3500
nshark	10
Fish breeding age	4
Shark breeding age	100
Shark starvation age	4

Wator 5

200x200 ocean

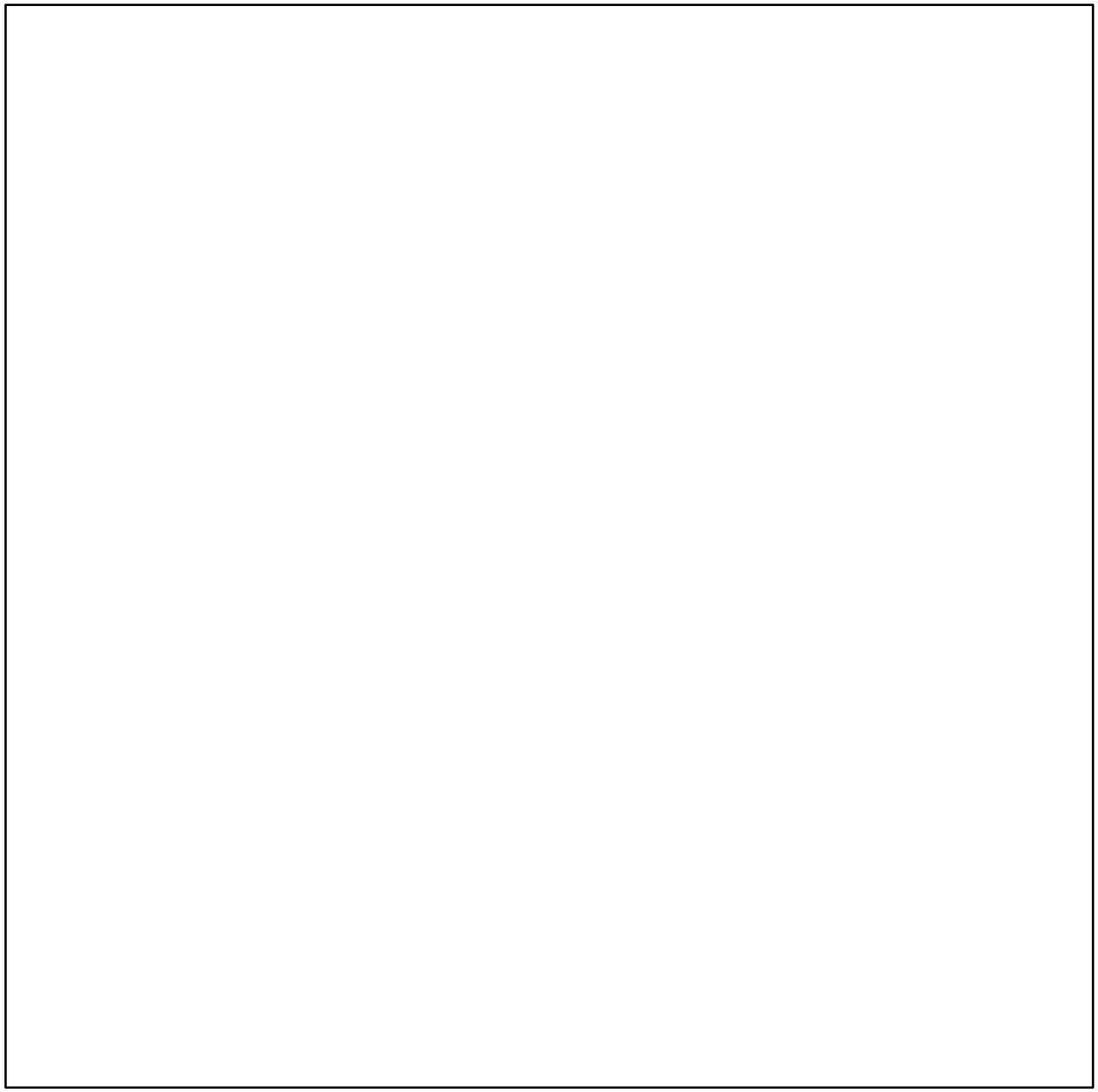
nfish	3500
nshark	10
Fish breeding age	4
Shark breeding age	4
Shark starvation age	5



Wator 6

200x200 ocean

nfish	3500
nshark	10
Fish breeding age	4
Shark breeding age	4
Shark starvation age	6



Wator 7

200x200 ocean

nfish	3500
nshark	10
Fish breeding age	5
Shark breeding age	10
Shark starvation age	4

Summary of Results

- SS = steady state

Input	nfish	nshark	Fish breeding age	Shark breeding age	Shark starvation age	Outcome
wator1	3500	10	4	5	4	SS
wator2	3500	10	4	30	4	SS
wator3	3500	10	4	50	4	SS
wator4	3500	10	4	100	4	SS
wator5	3500	10	4	4	5	Empty
wator6	3500	10	4	4	6	No fish
wator7	3500	10	5	10	4	All fish

Outline of Sequential Code

Initialise

- Initialise ocean array by placing fish and sharks at random grid locations.
- Initialise fish/shark list.

Update

- In each time step, we process the fish and sharks in the order in which they appear in the list.
- We may also update the display, or perform some other output, in each time step.

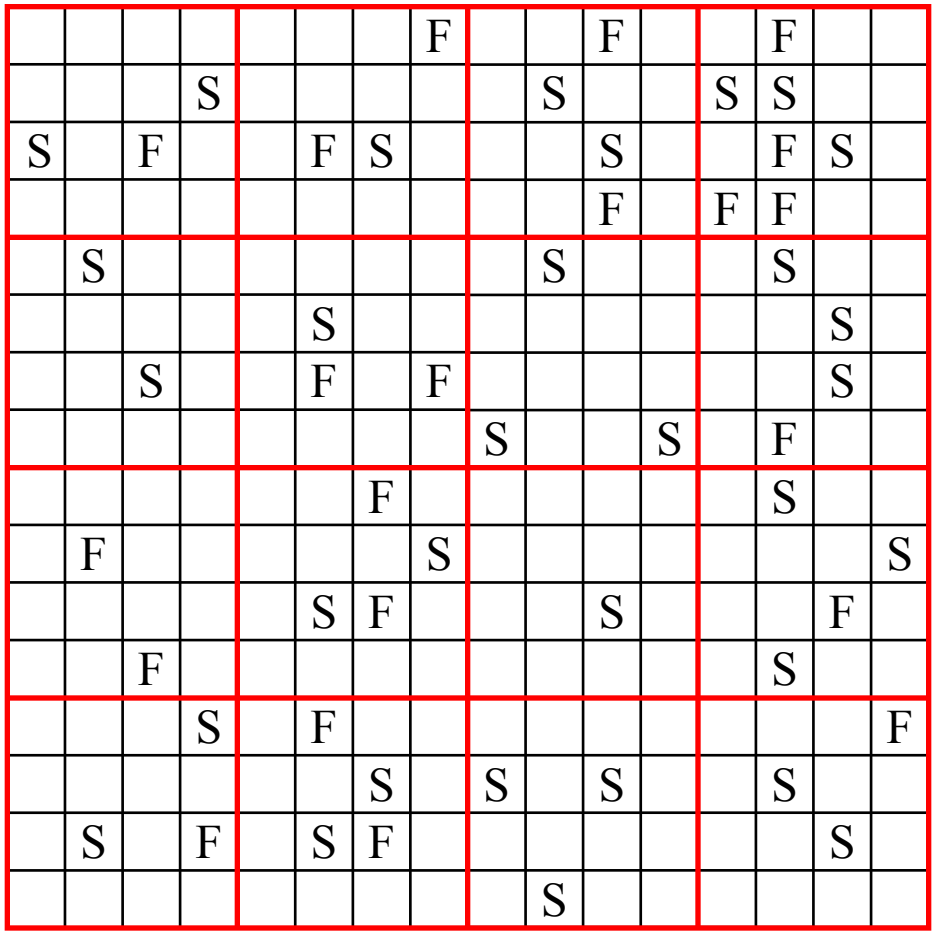
Finalise

- Output final state, statistics, etc.

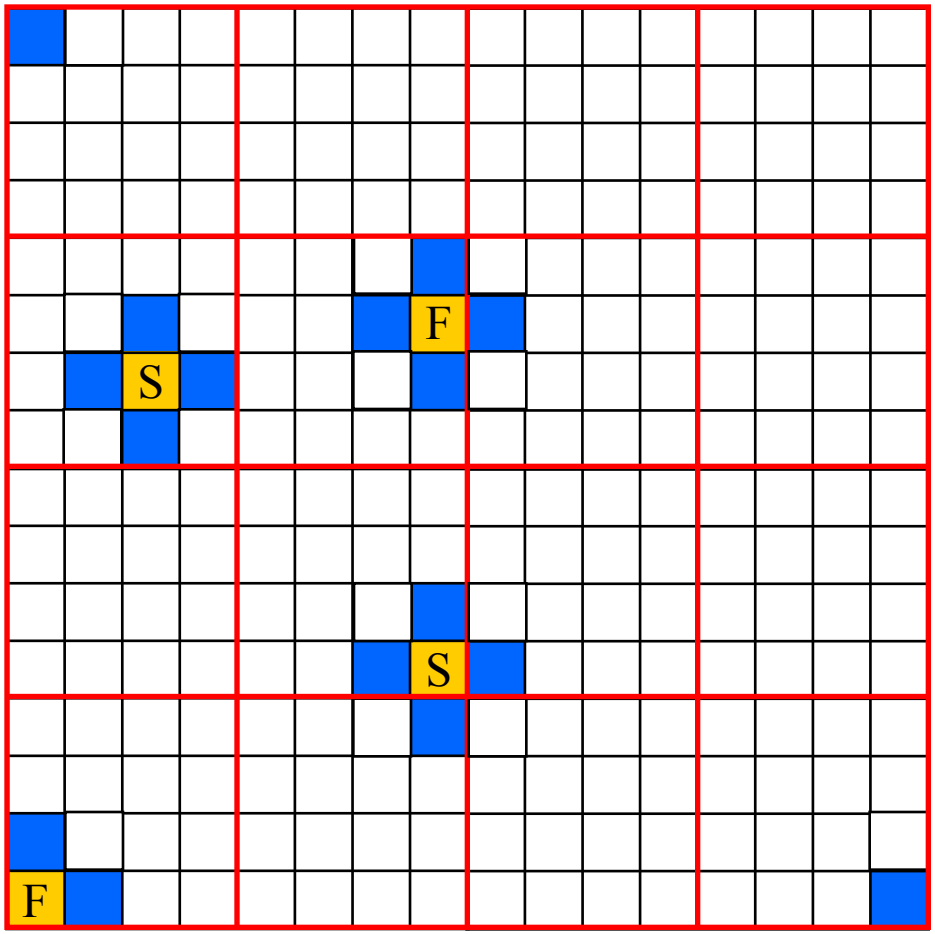
Data Distribution

- Initially we shall use a simple 2D block data distribution, just the same as in the Laplace equation problem and the molecular dynamics simulation.
- Each process looks after a block of the ocean and all the fish and sharks in it.
- We need to know the process number (rank), location in the process mesh, and the process numbers of the neighbouring processes.

Data Distribution 2



Data Dependencies



- Fish and sharks lying along the boundary of a process need to know about the grid cells lying along the boundary of other processes in order to follow the WaTor rules
- Each process needs data from 4 neighbouring processes.

Array Declarations

- When we receive fishes and sharks from ocean cells lying along the boundary of adjacent processes we store them in the fish/shark list and update the local ocean cell to indicate that is is occupied.

localcellx = 4, localcelly = 4

2D array of cells

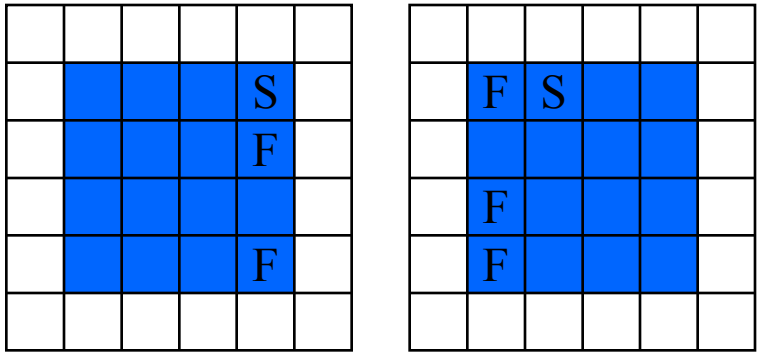
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

Potential for Data Inconsistency

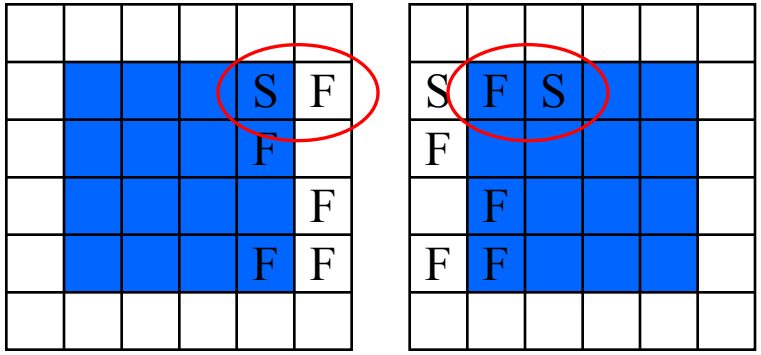
- In the course of an update a fish or shark may move into a “border” grid cell. Thus, unlike in the Laplace and molecular dynamics problems, the border cells in a process may change in an update.
- These changes must be communicated back to the process that originally sent that border strip, to be re-integrated into its data structures.
- However, the process owning those grid cells may also have updated them, and these updates may be in conflict.

Example of Data Inconsistency

Time T



Time T+1



- Here we show two adjacent processes at successive time steps.
- Two sharks try to eat the same fish!

Inconsistency and Non-Determinism

- On a shared memory parallel computer this type of data conflict would indicate a non-deterministic algorithm.
- On a distributed memory machine using message passing the communication operations determine the update order for a memory location, so the algorithm is still deterministic – but we would still like to avoid data conflicts.

Rollback

One way to resolve conflicts is called *rollback*, and works as follows:

1. Return the fish or shark that crossed the process boundary back to its original process, and place it back in its original position.
2. If that position has been occupied by another fish or shark, then that fish or shark must be rolled back.
3. This rollback process continues until until every fish and shark has a place to go.

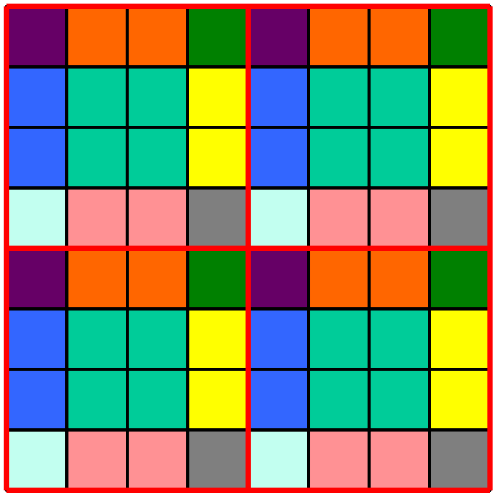
Rollback 2

- Rollback requires us to remember the previous position of each fish and shark.
- Rollback can result in complicated communication requirements if a sequence of rollbacks traverses multiple processes.
- Rollback has been used in certain event-driven simulations, such as battlefields simulations.

Isolating Boundary Updates

- We could perform updates by looping over the ocean cells instead of by processing the fish/shark list.
- Then we can update all the interior (i.e., non-boundary) cells.
- Next we do a left shift of the lefthand boundary fish/sharks, and update the righthand boundary (except the corners).
- This is repeated to update the lefthand, upper, and lower boundaries. Then do the corners.

Update Order



- There are 9 distinct regions that must be separately updated – 4 corners, 4 edges, and one central region.

Sub-Partitioning

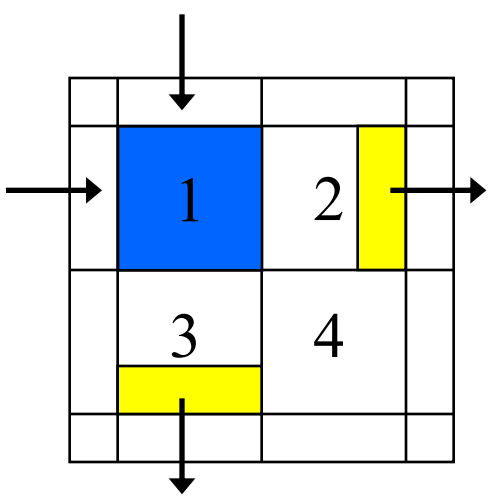
- A third way to avoid data inconsistencies is to partition the part of the ocean assigned to each process into 4 parts.
- A process has a separate fish/shark list for each of these 4 sub-partitions.
- Each of the 4 sub-partitions is updated in turn.
- This avoids adjacent ocean cells being updated concurrently.

Sub-Partitioning Algorithm

1. Divide the ocean array of each process into 4 smaller sub-grids, labelled 1, 2, 3, and 4.
2. Exchange the parts of sub-grids 2 and 3 that have data along their boundaries needed to update sub-grid 1 in adjacent processes.
3. Update sub-grid 1 in each process.
4. Return boundary information to original owner and update data structures.
5. Repeat steps 2, 3, and 4 for each of the other sub-grids in turn.

Update Cycle for Sub-Partition 1

Step 1

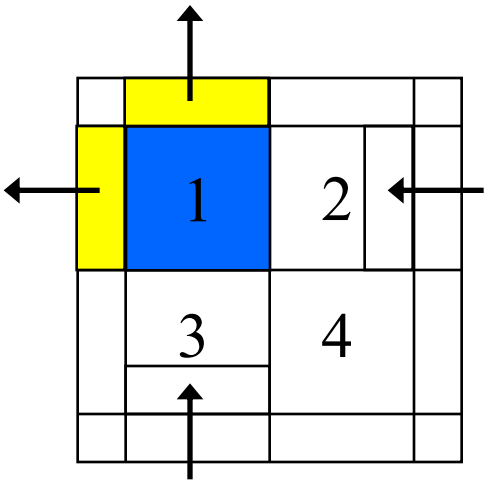


Right shift, down shift

Step 2

Update fish and sharks in list for sub-partition 1

Step 3



Left shift, up shift

Communication Phases

- To update sub-partition 1: shift right and down before update, then shift left and up after update.
- To update sub-partition 2: shift left and down before update, then shift right and up after update.
- To update sub-partition 3: shift right and up before update, then shift left and down after update.
- To update sub-partition 4: shift left and up before update, then shift right and down after update.

Outline of Parallel Code

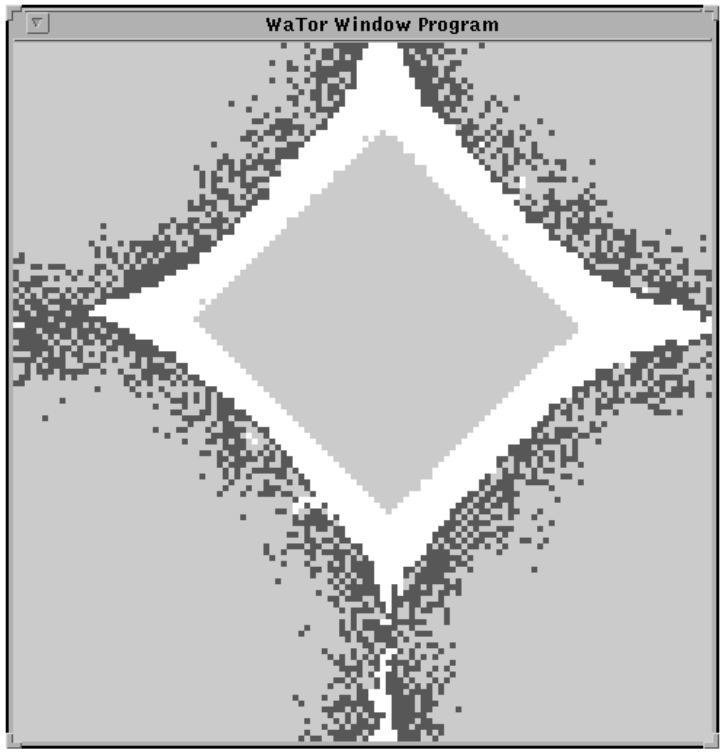
The update loop of the parallel version of WaTor using sub-grids is as follows:

```
for (each time step){
    for (each sub-partition, i=1,2,3,4){
        shift boundary data across 2 edges of sub-partition i
        store data received in border of ocean and in
            fish/shark list
        update fish and sharks in sub-partition i
        shift boundary data back across the 2 edges,
            overwriting original data with updated data
    }
}
```

Load Imbalance in WaTor

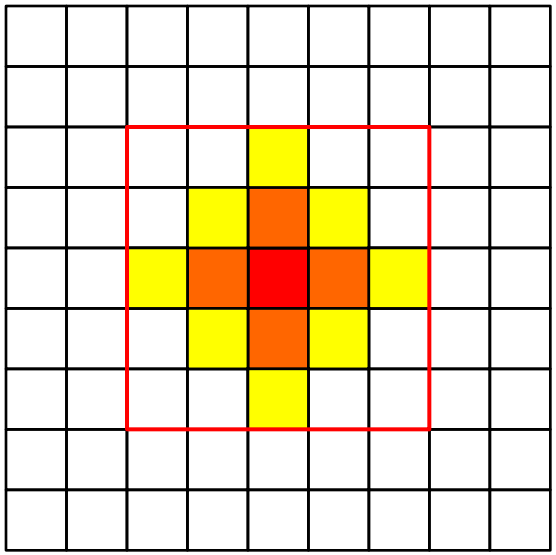
- Load balance is an important consideration in WaTor and many other applications.
- In WaTor the workload if generally not evenly distributed over the ocean, so distributing the data in contiguous blocks means that some processes have less work than others at certain times.
- Load balance in WaTor changes with time as the fish and sharks move.

Example of WaTor Output



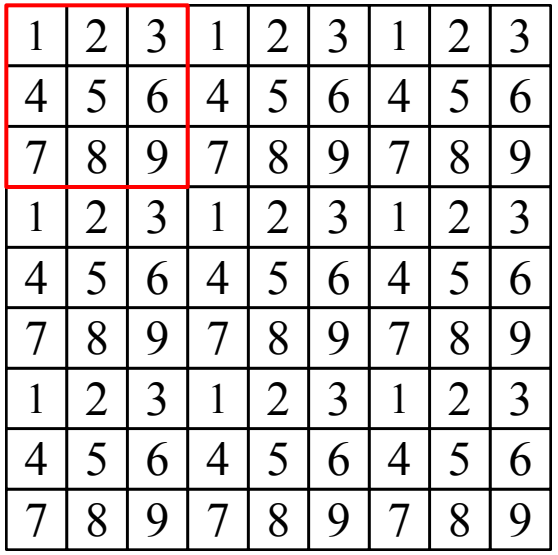
- White = fish
- Light grey = empty
- Dark grey = sharks

Concurrent Updates in WaTor



Consider which locations cannot be update at the same time as the red location.

Ordering Updates in WaTor



Locations with the same number can be updated concurrently without giving rise to a race condition.

This would require 9 kernel calls in CUDA.

Can also use sub-partitioning in CUDA