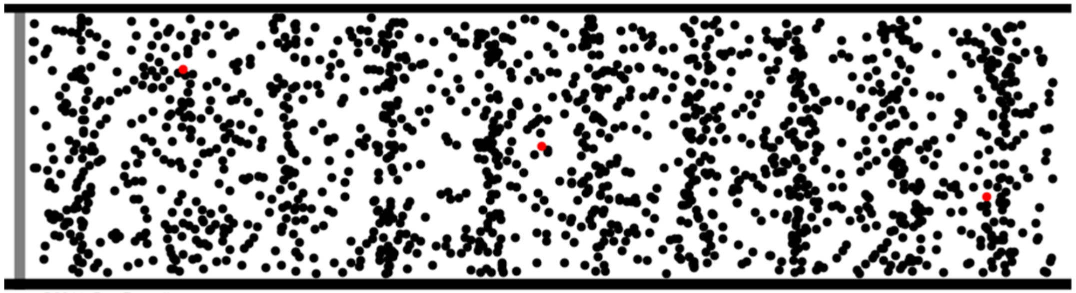
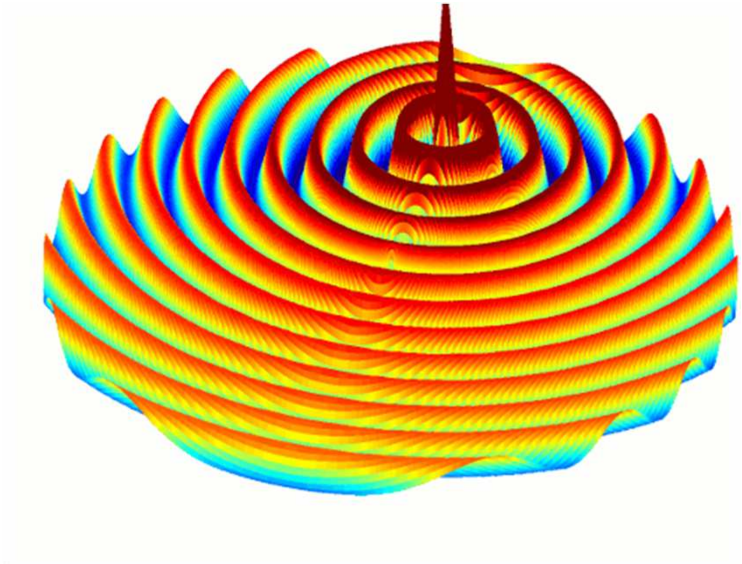
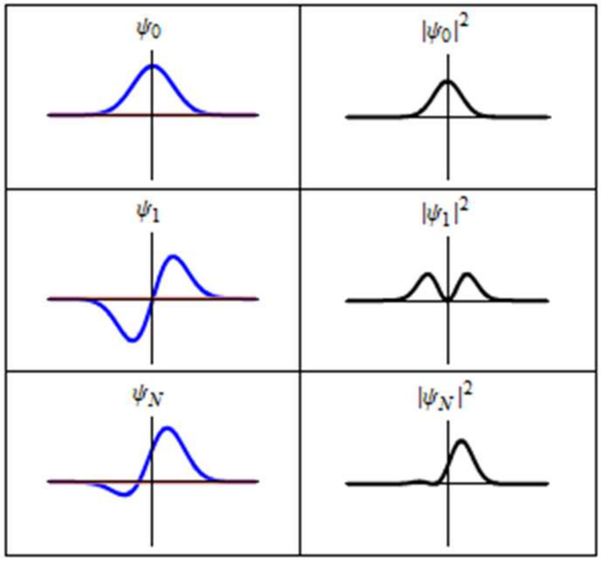
Waves!

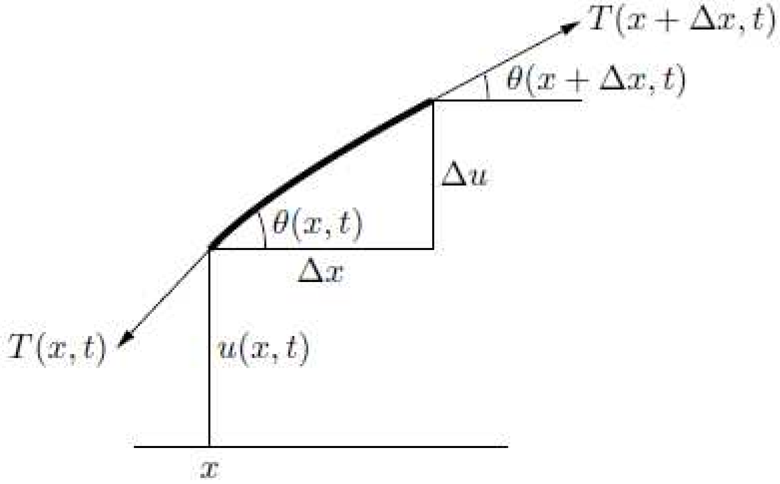


©2011. Dan Russell





Solving something like this…



The Wave Equation

- (1-D)

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}$$

- (n-D)

$$\frac{\partial^2 y}{\partial t^2} = c^2 \, \nabla^2 y$$

The Wave Equation

$$\frac{\partial}{\partial t}\frac{y_{x,t+1}-y_{x,t}}{\Delta t}=c^2\frac{\partial}{\partial x}\frac{y_{x+1,t}-y_{x,t}}{\Delta x}$$

→

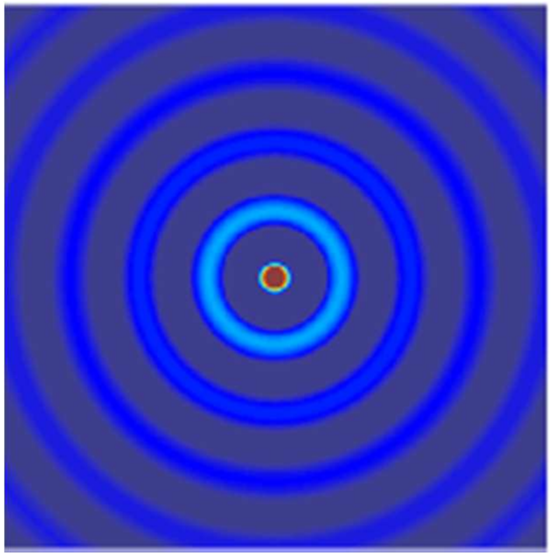
$$\frac{(y_{x,t+1}-y_{x,t})-(y_{x,t}-y_{x,t-1})}{(\Delta t)^2}=c^2\frac{(y_{x+1,t}-y_{x,t})-(y_{x,t}-y_{x-1,t})}{(\Delta x)^2}$$

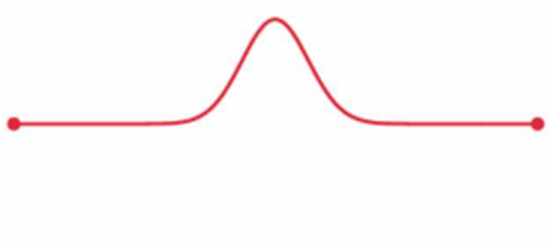
→

$$y_{x,t+1}=2y_{x,t}-y_{x,t-1}+\left(\frac{c\Delta t}{\Delta x}\right)^2(y_{x+1,t}-2y_{x,t}+y_{x-1,t})$$

Boundary Conditions

- Examples:
  - – Manual motion at an end
    - u(0, t) = f(t)
  - – Bounded ends:
    - u(0, t) = u(L, t) = 0 for all t





Discrete solution

- Deal with three states at a time (all positions at t -1, t, t+1)

- Let L = number of nodes (distinct discrete positions)
  - Create a 2D array of size 3*L
  - Denote pointers to where each region begins
  - Cyclically overwrite regions you don't need!

Discrete solution

- Sequential pseudocode:

```
fill data array with initial conditions

for all times t = 0… t_max
     point old_data pointer               where current_data used to be
     point current_data pointer           where new_data used to be
     point new_data pointer               where old_data used to be!
                                          (so that we can overwrite the old!)

     for all positions x = 1…up to number of nodes-2
              calculate f(x, t+1)
     end
     set any boundary conditions!

     (every so often, write results to file)
end
```

GPU Algorithm - Kernel

- (Assume kernel launched at some time t…)

- Calculate y(x, t+1)
  - Each thread handles only a few values of x!
    - Similar to polynomial problem

$$y_{x,t+1} = 2y_{x,t} - y_{x,t-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (y_{x+1,t} - 2y_{x,t} + y_{x-1,t})$$

GPU Algorithm – The Wrong Way

- Recall the old “GPU computing instructions”:
  - Setup inputs on the host (CPU-accessible memory)
  - Allocate memory for inputs on the GPU
  - Copy inputs from host to GPU
  - Allocate memory for outputs on the host
  - Allocate memory for outputs on the GPU
  - Start GPU kernel
  - Copy output from GPU to host

GPU Algorithm – The Wrong Way

- Sequential pseudocode:

```
fill data array with initial conditions

for all times t = 0… t_max
    adjust old_data pointer
    adjust current_data pointer
    adjust new_data pointer

    allocate memory on the GPU for old, current, new
    copy old, current data from CPU to GPU

    launch kernel
    copy new data from GPU to CPU
    free GPU memory

    set any boundary conditions!

    (every so often, write results to file)
end
```

GPU Algorithm – The Wrong Way

- Insidious memory transfer!
- Many memory allocations!

GPU Algorithm – The Right Way

- Sequential pseudocode:

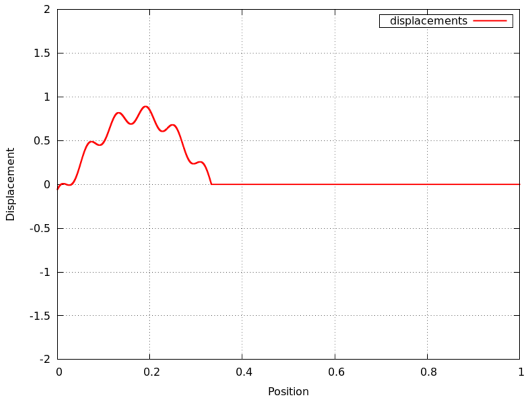
```
allocate memory on the GPU for old, current, new
Either:
    Create initial conditions on CPU, copy to GPU
    Or, calculate and/or set initial conditions on the GPU!

for all times t = 0… t_max
    adjust old_data address
    adjust current_data address
    adjust new_data address

    launch kernel with the above addresses

    Either:
        Set boundary conditions on CPU, copy to GPU
        Or, calculate and/or set boundary conditions on the GPU
End
free GPU memory
```

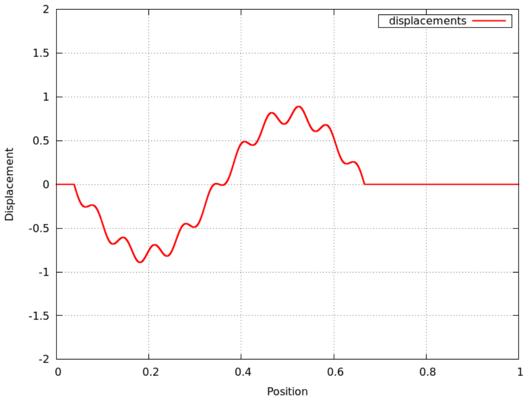
GPU Algorithm – The Right Way

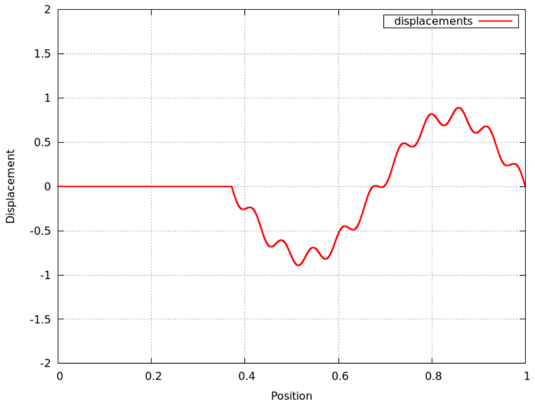
- Everything stays on the GPU all the time!
  - Almost…

Getting output

- What if we want to get a “snapshot” of the simulation?
  - That’s when we GPU-CPU copy!







GPU Algorithm – The Right Way

- Sequential pseudocode:

```
allocate memory on the GPU for old, current, new
Either:
    Create initial conditions on CPU, copy to GPU
    Or, calculate and/or set initial conditions on the GPU!

for all times t = 0… t_max
    adjust old_data address
    adjust current_data address
    adjust new_data address

    launch kernel with the above addresses

    Either:
        Set boundary conditions on CPU, copy to GPU
        Or, calculate and/or set boundary conditions on the GPU
    Every so often, copy from GPU to CPU and write to file
End
free GPU memory
```

Multi-GPU Utilization

- What if we want to use multiple GPUs…
  - Within the same machine?
  - Across a distributed system?

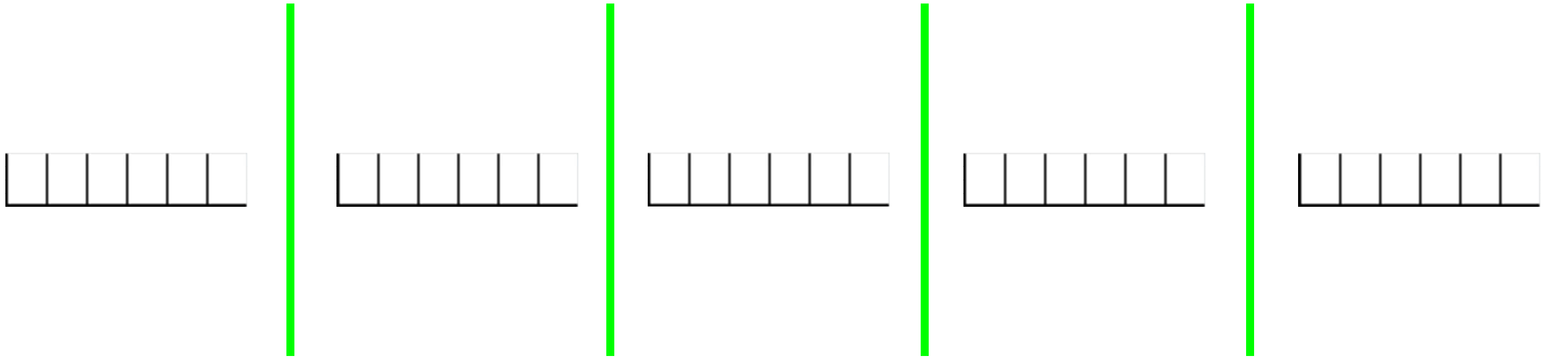
GPU cluster, CSIRO

- Recall our calculation:

$$y_{x,t+1} = 2y_{x,t} - y_{x,t-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (y_{x+1,t} - 2y_{x,t} + y_{x-1,t})$$

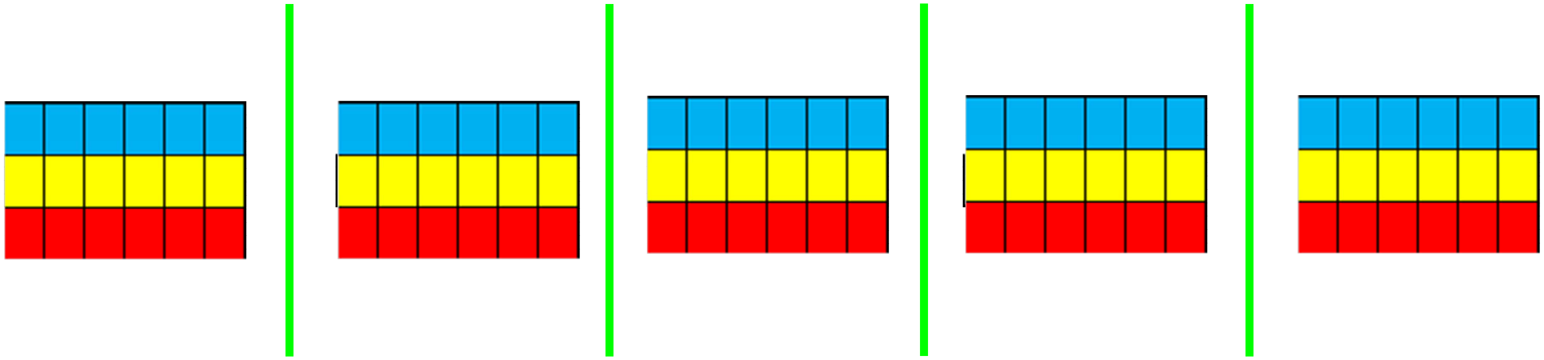
Multiple GPU Solution

- Big idea: Divide our data array between *n* GPUs!



Multiple GPU Solution

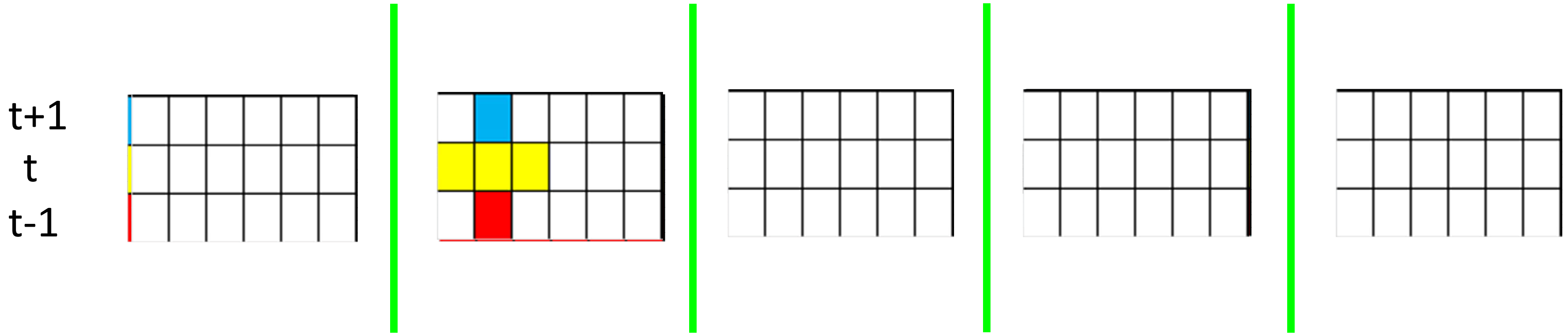
- In reality: We have three regions of data at a time (old, current, new)



Multiple GPU Solution

- Calculation for timestep t+1 uses the following data:

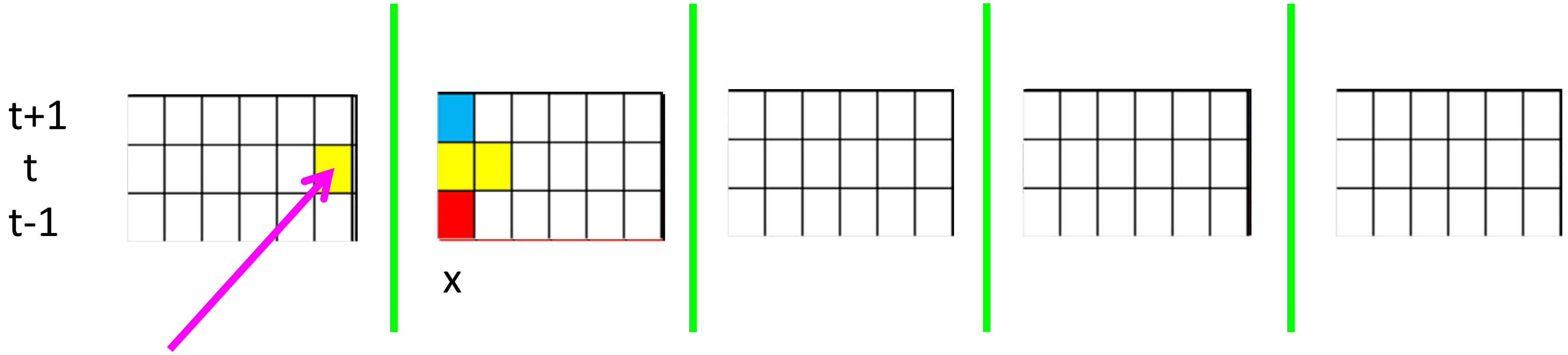
$$y_{x,t+1} = 2y_{x,t} - y_{x,t-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (y_{x+1,t} - 2y_{x,t} + y_{x-1,t})$$



Multiple GPU Solution

- Problem if we’re at the boundary of a process!

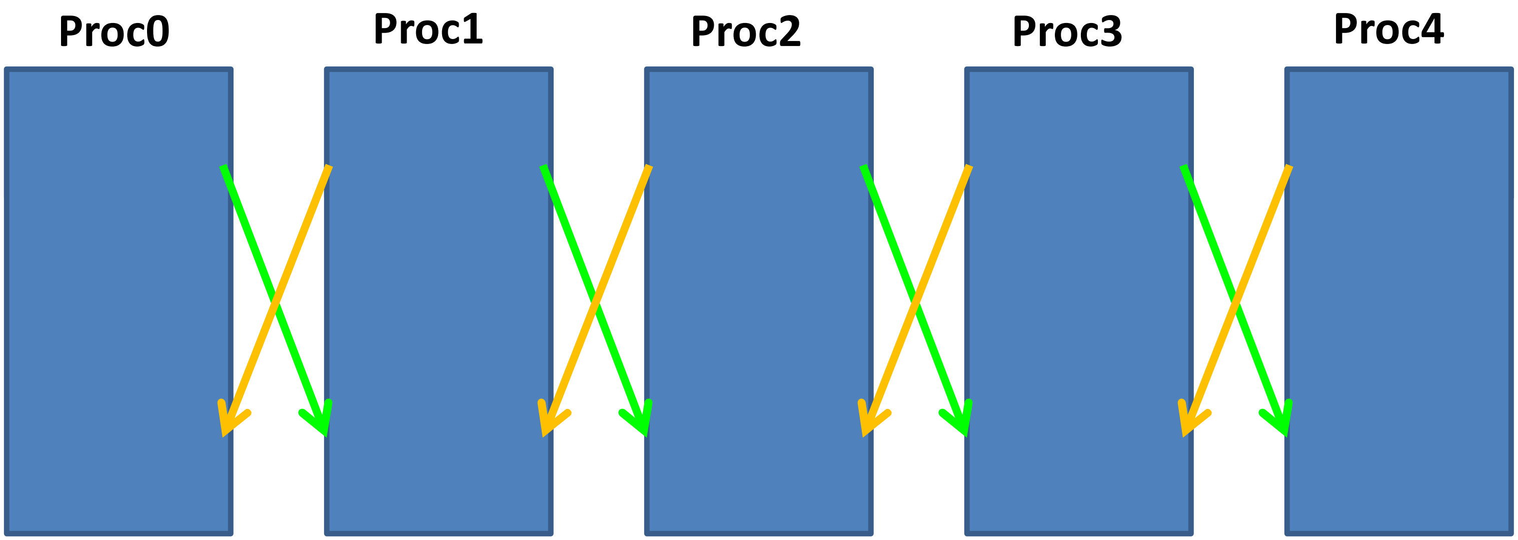
$$y_{x,t+1} = 2y_{x,t} - y_{x,t-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (y_{x+1,t} - 2y_{x,t} + y_{x-1,t})$$



Where do we get $y_{x-1,t}$? (It’s outside our process!)

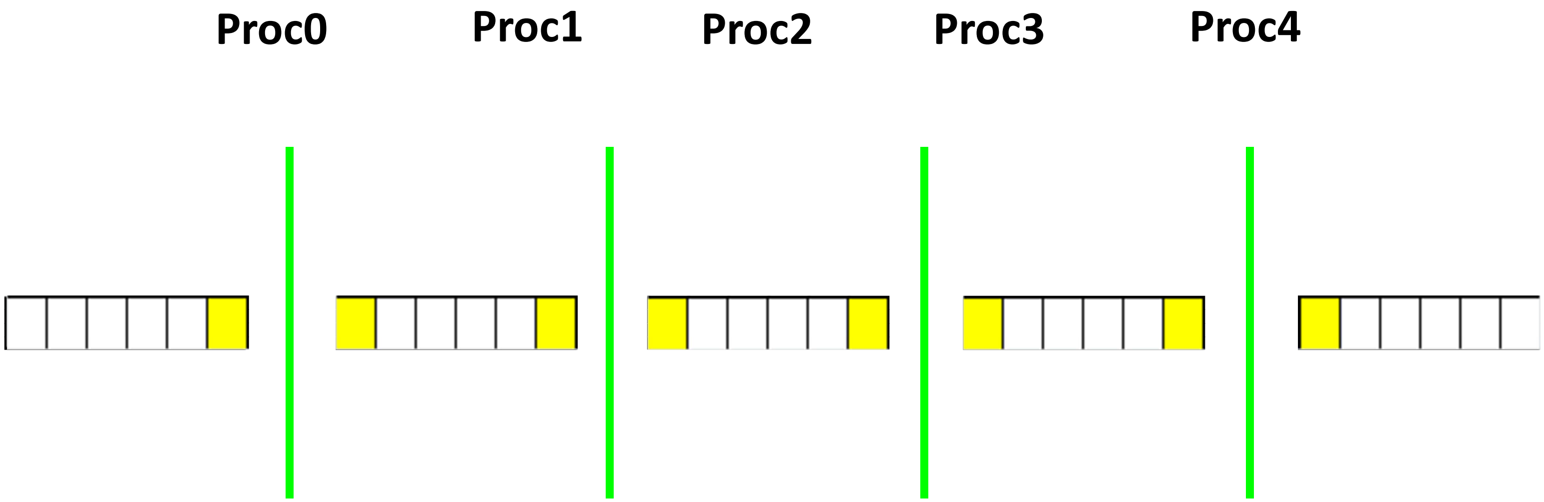
Multiple GPU Solution

- After every time-step, each process gives its leftmost and rightmost piece of “current” data to neighbor processes!



Multiple GPU Solution

- Pieces of data to communicate:



Multiple GPU Solution

- (More details next lecture)

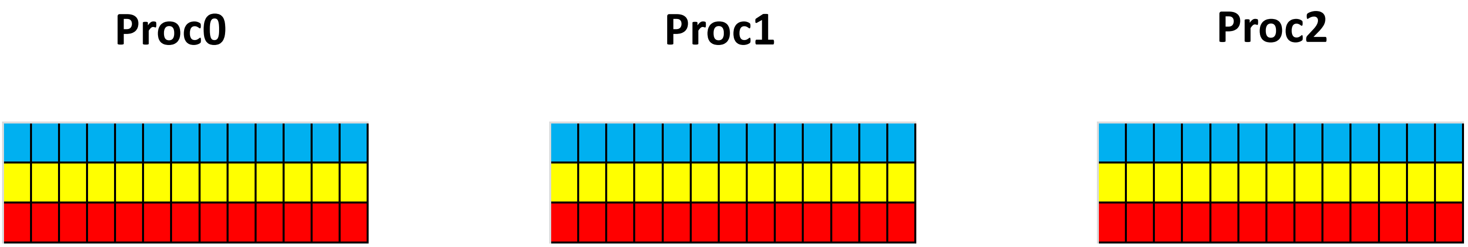
- General idea – suppose we're on GPU r in 0…(N-1):
  - If we're not GPU N-1:
    - Send data to process r+1
    - Receive data from process r+1
  - If we're not GPU 0:
    - Send data to process r-1
    - Receive data from process r-1
  - Wait on requests

Multiple GPU Solution

- Communication can be expensive!
  - Expensive to communicate every timestep to send 1 value!
  - Better solution: Send some *m* values every *m* timesteps!

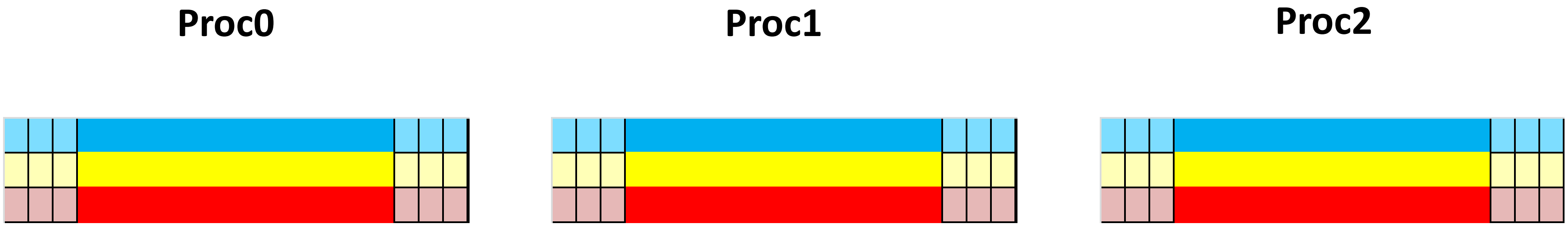
Possible Implementation

- Initial setup: (*Assume 3 processes*)



Possible Implementation

- Give each array "redundant regions"
- (*Assume communication interval = 3*)

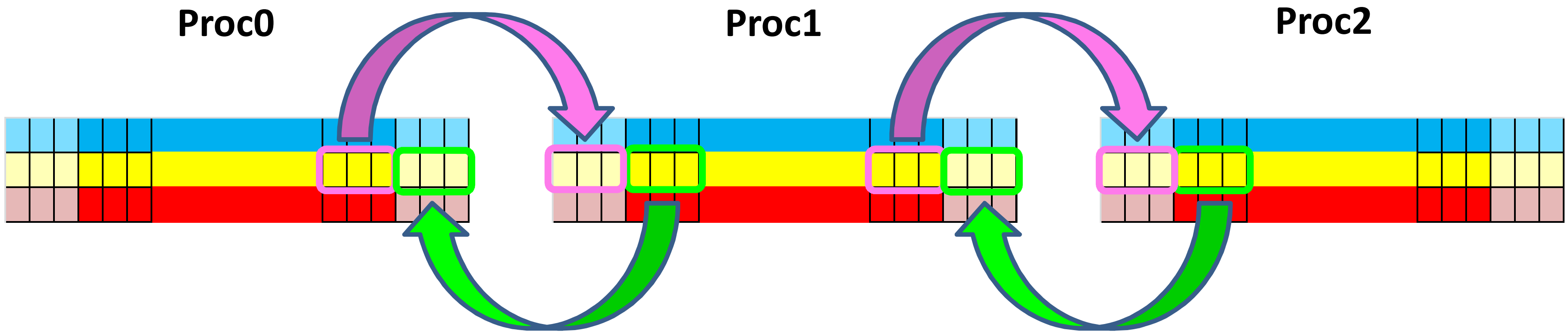


Possible Implementation

- Every (3) timesteps, send some of your data to neighbor processes!

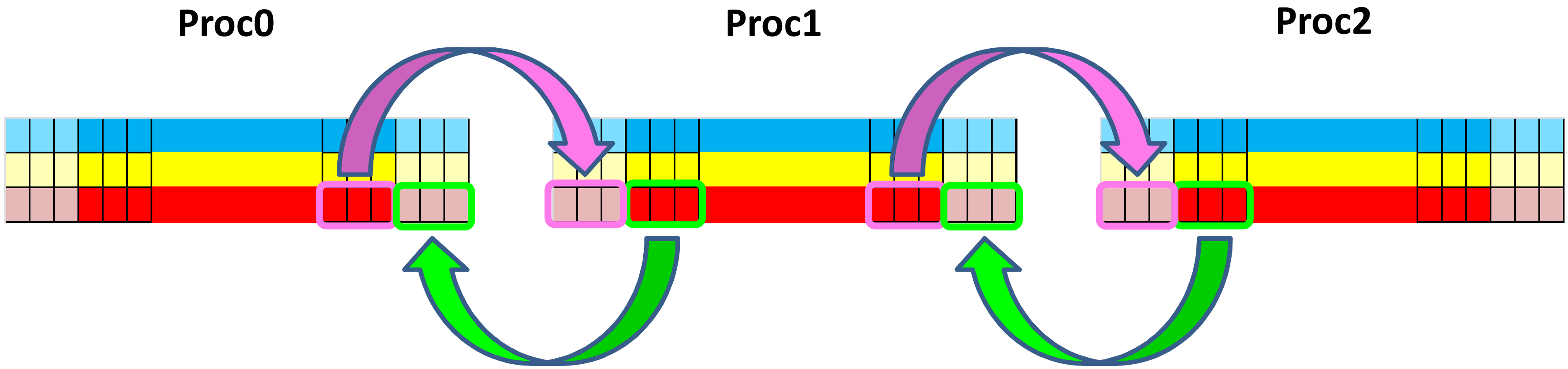
Possible Implementation

- Send “current” data (current at time of communication)



Possible Implementation

- Then send "old" data

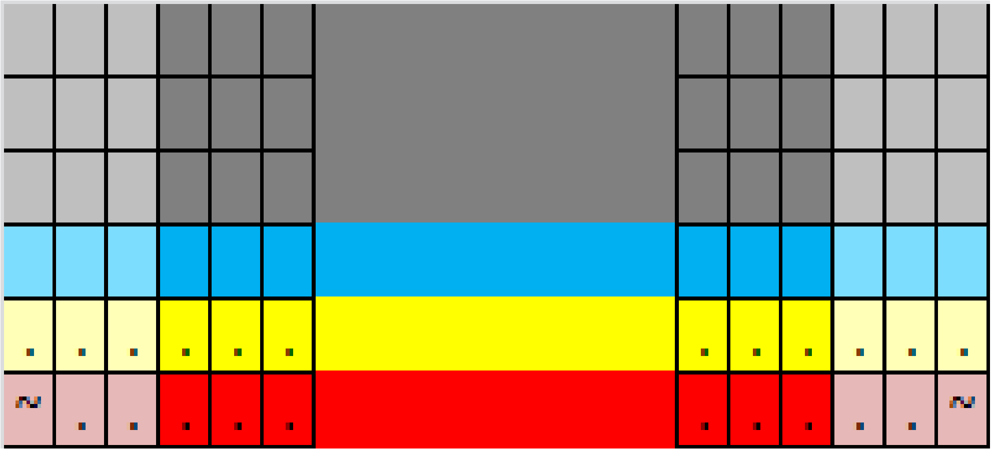


- Then…
  - Do our calculation as normal, if we’re not at the ends of our array
    - Our entire array, including redundancies!

$$y_{x,t+1} = 2y_{x,t} - y_{x,t-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (y_{x+1,t} - 2y_{x,t} + y_{x-1,t})$$

What about corruption?

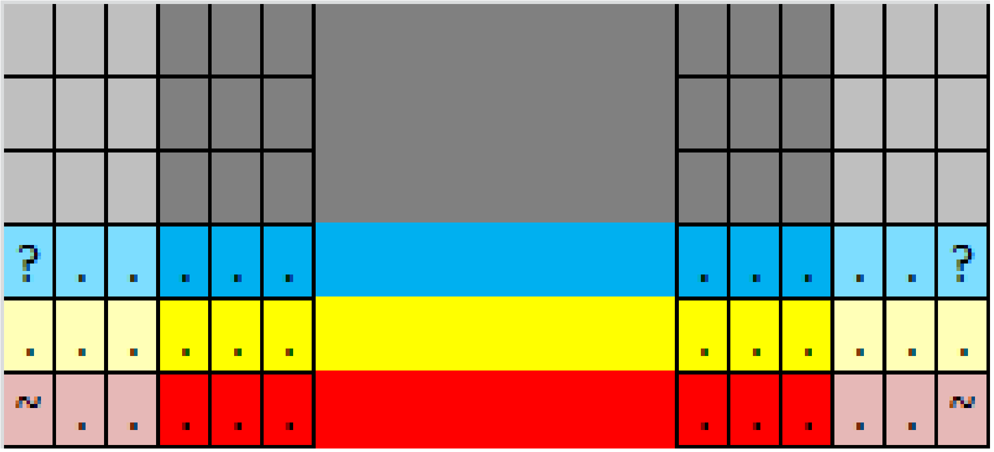
- Suppose we’ve just copied our data… (assume a non-boundary process)
    - . = valid
    - ? = garbage
    - ~ = doesn’t matter



    - (Recall that there exist only 3 spaces – gray areas are nonexistent in our current time

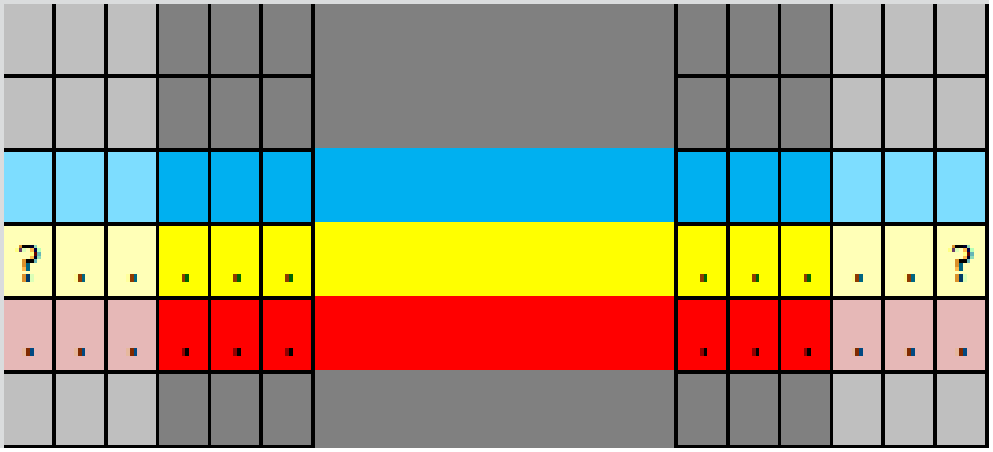
What about corruption?

- Calculate new data…
  - Value unknown!



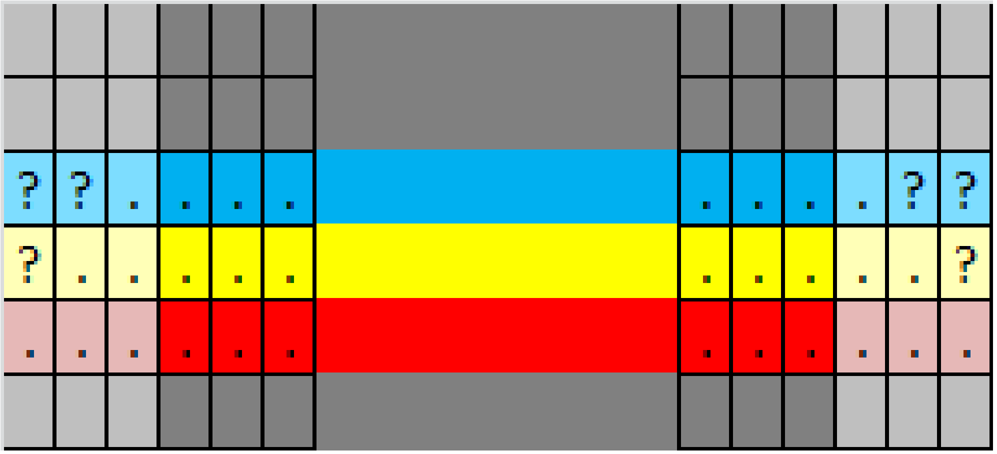
What about corruption?

- Time t+1:
  - Current -> old, new -> current (and space for old is overwritten by new…)



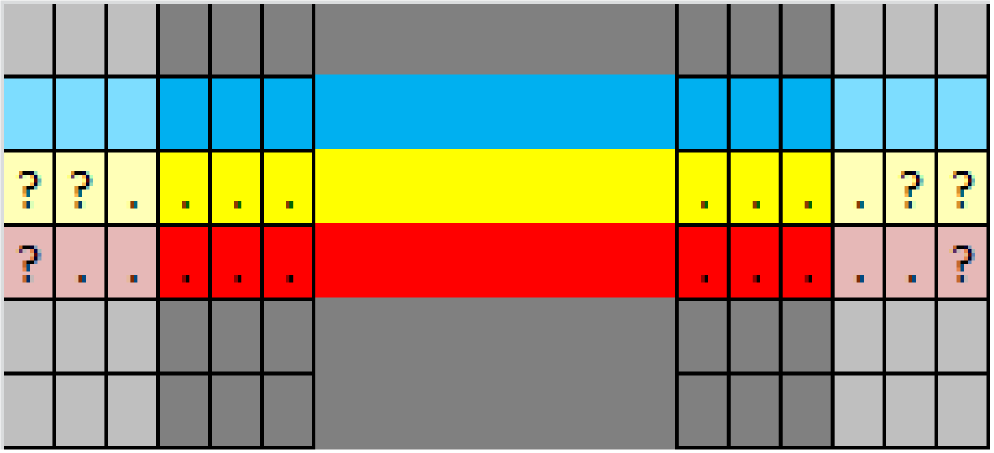
What about corruption?

- More garbage data!
  - “Garbage in, garbage out!”



What about corruption?

- Time t+2…



What about corruption?

- Even more garbage!

?	?	?	.	.	.		.	.	.	?	?	?
?	?	.	.	.	.		.	.	.	.	?	?
?	.	.	.	.	.		.	.	.	.	.	?

What about corruption?

- Time t+3…
  - Core data region - corruption imminent!?

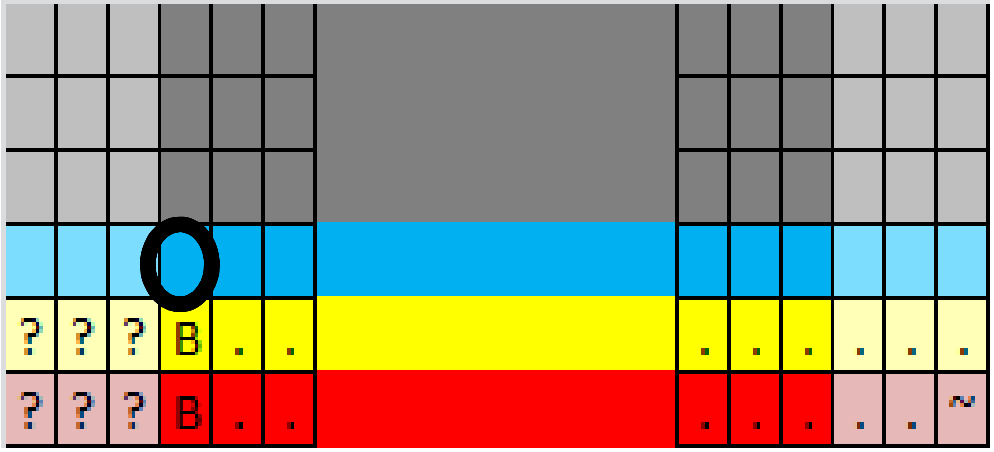
			!						!			
?	?	?	.	.	.		.	.	.	?	?	?
?	?	.	.	.	.		.	.	.	.	?	?

What about corruption?

- Saved!
  - Data exchange occurs after communication interval has passed!

Boundary corruption?

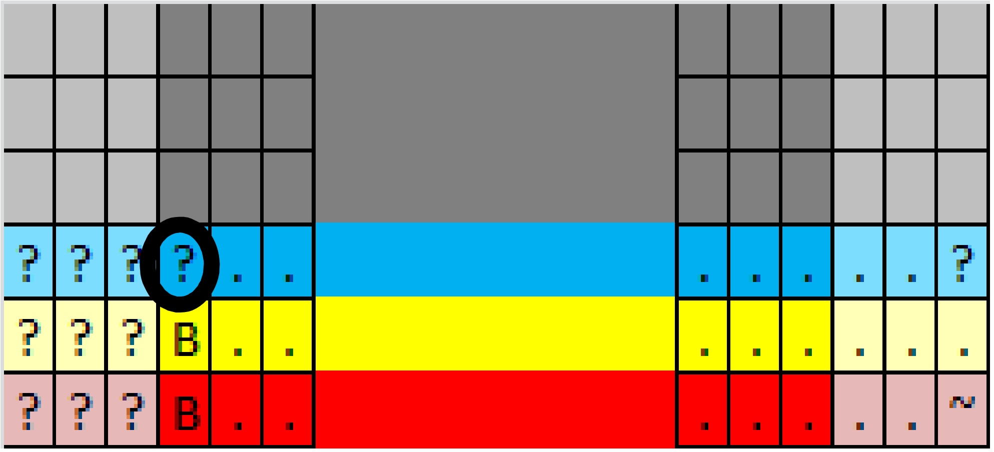
- Examine left-most process:
  - We never copy to it, so left redundant region is garbage!



(B = boundary condition set)

Boundary corruption?

- Calculation brings garbage into non-redundant region!



Boundary corruption?

- …but boundary condition is set at every interval!