CS 179: GPU Programming

Lecture 20: Cross-system communication

The Wave Equation

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

→

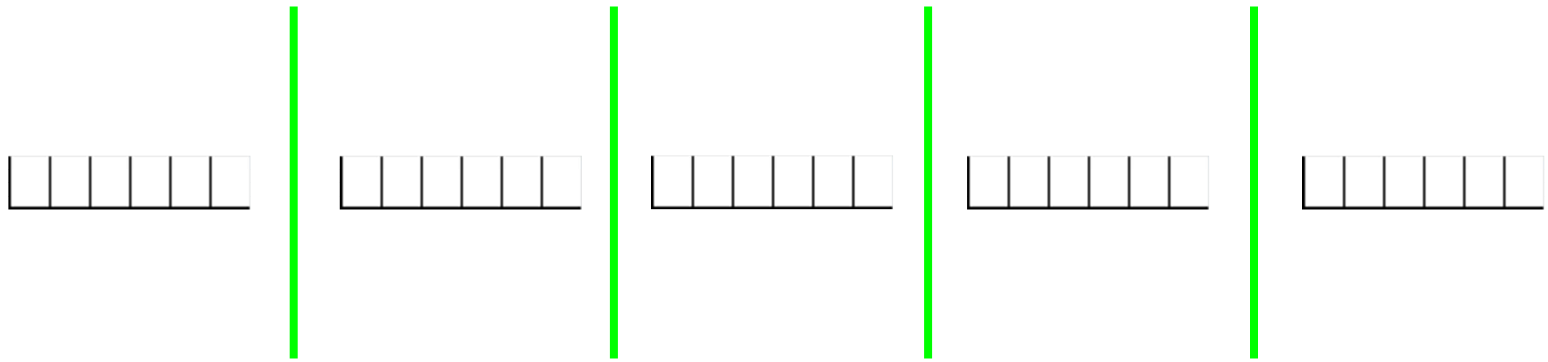
$$\frac{(y_{x,t+1} - y_{x,t}) - (y_{x,t} - y_{x,t-1})}{(\Delta t)^2} = c^2 \frac{(y_{x+1,t} - y_{x,t}) - (y_{x,t} - y_{x-1,t})}{(\Delta x)^2}$$

→

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

Multiple GPU Solution

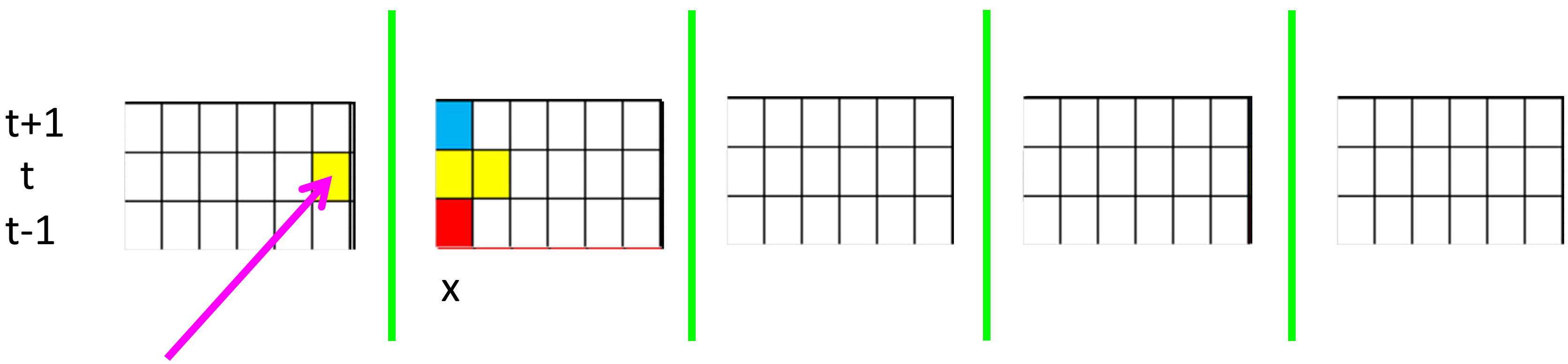
- Big idea: Divide our data array between *n* GPUs!



Multiple GPU Solution

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

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



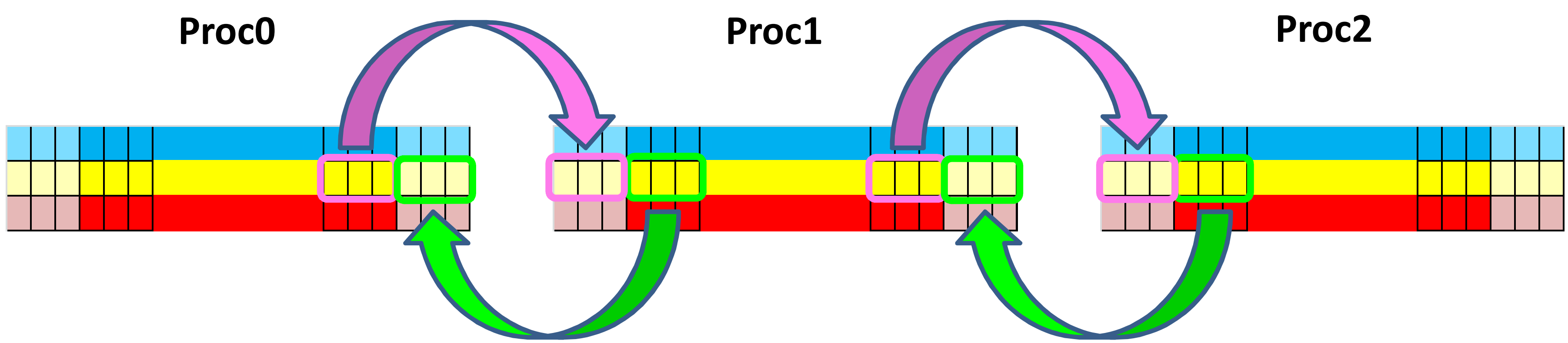
Where do we get $y_{x-1,t}$? (It's outside our process!)

Multiple GPU Solution

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

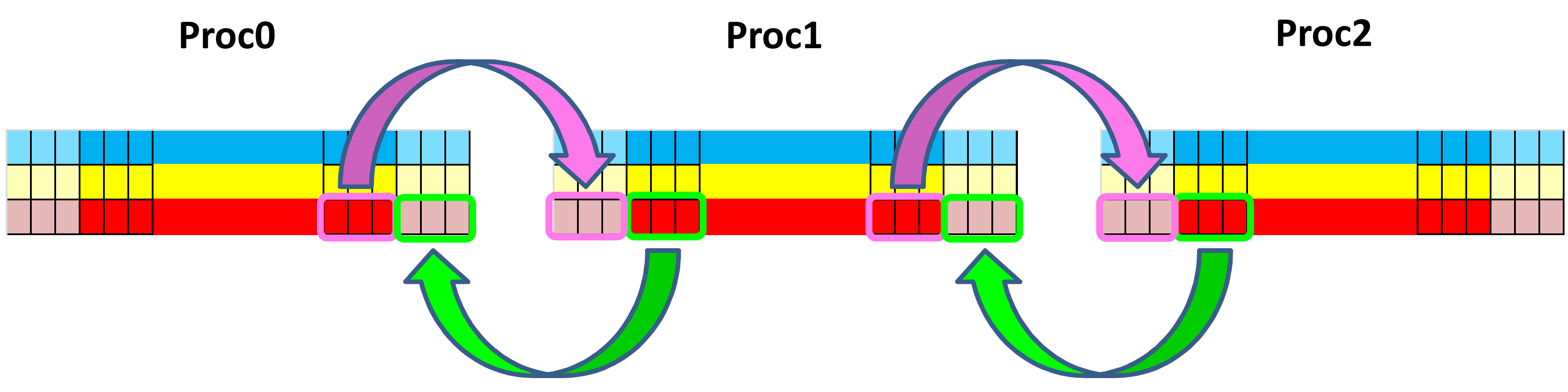
Possible Implementation

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



Possible Implementation

- Then send "old" data



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

- GPUs on same system:
  - Use CUDA-supplied functions (cudaMemcpyPeer, etc.)

- GPUs on different systems:
  - Need cross-system, *inter-process* communication…

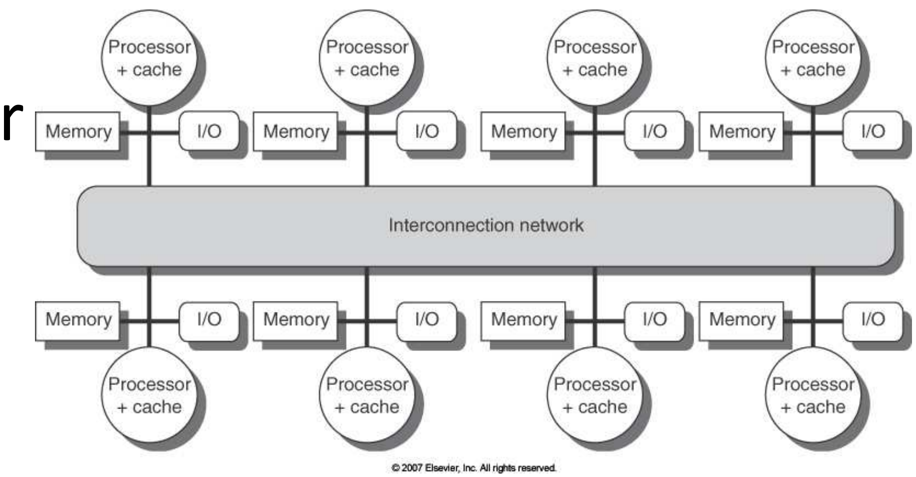
Supercomputers

- Often have:
  - Many different systems
  - Few GPUs/system

GPU cluster, CSIRO

Distributed System

- A collection of computers
  - Each computer has its own local memory!
  - Communication over



- Communication suddenly becomes harder! (and slower!)
- GPUs can't be trivially used between computers

Message Passing Interface (MPI)

- A *standard* for message-passing
  - Multiple implementations exist
  - Standard functions that allow easy communication of data between processes

- Non-networked systems:
  - Equivalent to memcpy on local system

MPI Functions

- There are seven basic functions:
  - MPI_Init *initialize MPI environment*
  - MPI_Finalize *terminate MPI environment*

  - MPI_Comm_size *how many processes we have running*
  - MPI_Comm_rank *the ID of our process*

  - MPI_Isend *send data (nonblocking)*
  - MPI_Irecv *receive data (nonblocking)*

  - MPI_Wait *wait for request to complete*

MPI Functions

- Some additional functions:

  - MPI_Barrier *wait for all processes to reach a certain point*

  - MPI_Bcast *send data to all other processes*
  - MPI_Reduce *receive data from all processes and reduce to a value*

  - MPI_Send *send data (blocking)*
  - MPI_Recv *receive data (blocking)*

Blocking vs. Non-blocking

- MPI_Isend and MPI_Irecv are *asynchronous (non-blocking)*
  - Calling these functions returns immediately
    - Operation may not be finished!
  - Should use MPI_Wait to make sure operations are completed
  - Special “request” objects for tracking status

- MPI_Send and MPI_Recv are *synchronous (blocking)*
  - Functions don’t return until operation is complete
  - Can cause deadlock!

  - (we won’t focus on these)

MPI Functions - Wait

- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

- Takes in…
  - A “request” object corresponding to a previous operation
    - Indicates what we’re waiting on
  - A “status” object
    - Basically, information about incoming data

MPI Functions - Reduce

- `int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

- Takes in…
  - A “send buffer” (data obtained from every process)
  - A “receive buffer” (where our final result will be placed)
  - Number of elements in send buffer
    - Can reduce element-wise array -> array
  - Type of data (MPI label, as before)

…

MPI Functions - Reduce

- `int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

- Takes in… (continued)
  - Reducing operation (special MPI labels, e.g. MPI_SUM, MPI_MIN)
  - ID of process that obtains result
  - MPI communication object (as before)

MPI Example

- Two processes
- Sends a number from process 0 to process 1
- Note: Both processes are running this code!

```
int main(int argc, char **argv) {
    int rank, numprocs;

    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    int tag=1234;
    int source=0;
    int destination=1;
    int count=1;

    int send_buffer;
    int recv_buffer;

    if(rank == source){
      send_buffer=5678;
      MPI_Isend(&send_buffer,count,MPI_INT,destination,tag,
              MPI_COMM_WORLD,&request);
    }

    if(rank == destination){
      MPI_Irecv(&recv_buffer,count,MPI_INT,source,tag,
              MPI_COMM_WORLD,&request);
    }

    MPI_Wait(&request,&status);

    if(rank == source){
      printf("processor %d  sent %d\n",rank,recv_buffer);
    }
    if(rank == destination){
      printf("processor %d  got %d\n",rank,recv_buffer);
    }
    MPI_Finalize();
    return 0;
}
```

Wave Equation – Simple Solution

- Can do this with MPI_Irecv, MPI_Isend, MPI_Wait:

- Suppose process has rank r:
  - If we're not the rightmost process:
    - Send data to process r+1
    - Receive data from process r+1
  - If we're not the leftmost process:
    - Send data to process r-1
    - Receive data from process r-1
  - Wait on requests

Wave Equation – Simple Solution

- Boundary conditions:
  - Use MPI_Comm_rank and MPI_Comm_size
    - Rank 0 process will set leftmost condition
    - Rank (size-1) process will set rightmost condition

Simple Solution – Problems

- Communication can be expensive!
  - Expensive to communicate every timestep to send 1 value!
  - Better solution: Send some *m* values every *m* timesteps!
  - Tradeoff between redundant computations and reduced network/communication overhead
    - Network (MPI) case worse than the multi-GPU case!