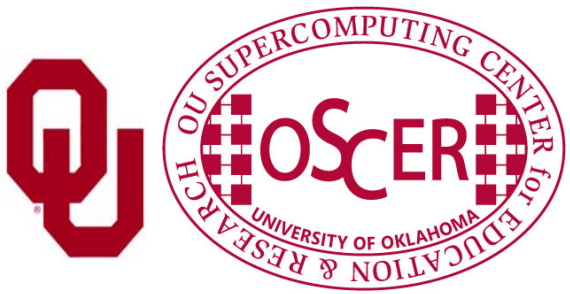
Parallel Programming & Cluster Computing

Transport Codes and Shifting

Henry Neeman, University of Oklahoma
Paul Gray, University of Northern Iowa

SC08 Education Program's Workshop on Parallel & Cluster computing
Oklahoma Supercomputing Symposium, Monday October 6 2008



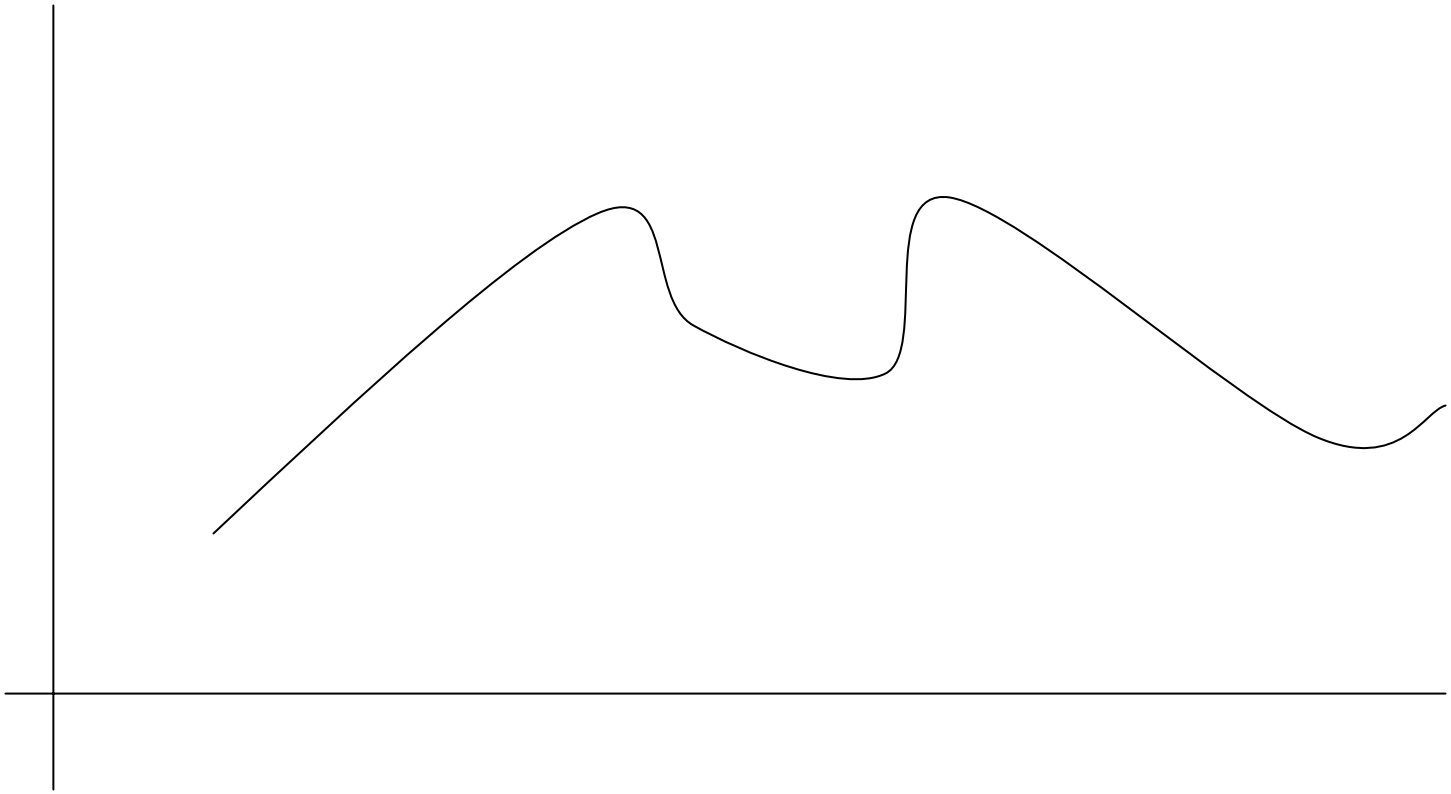
What is a Simulation?

All physical science ultimately is expressed as calculus (e.g., differential equations).

Except in the simplest (uninteresting) cases, equations based on calculus can't be directly solved on a computer.

Therefore, all physical science on computers has to be approximated.

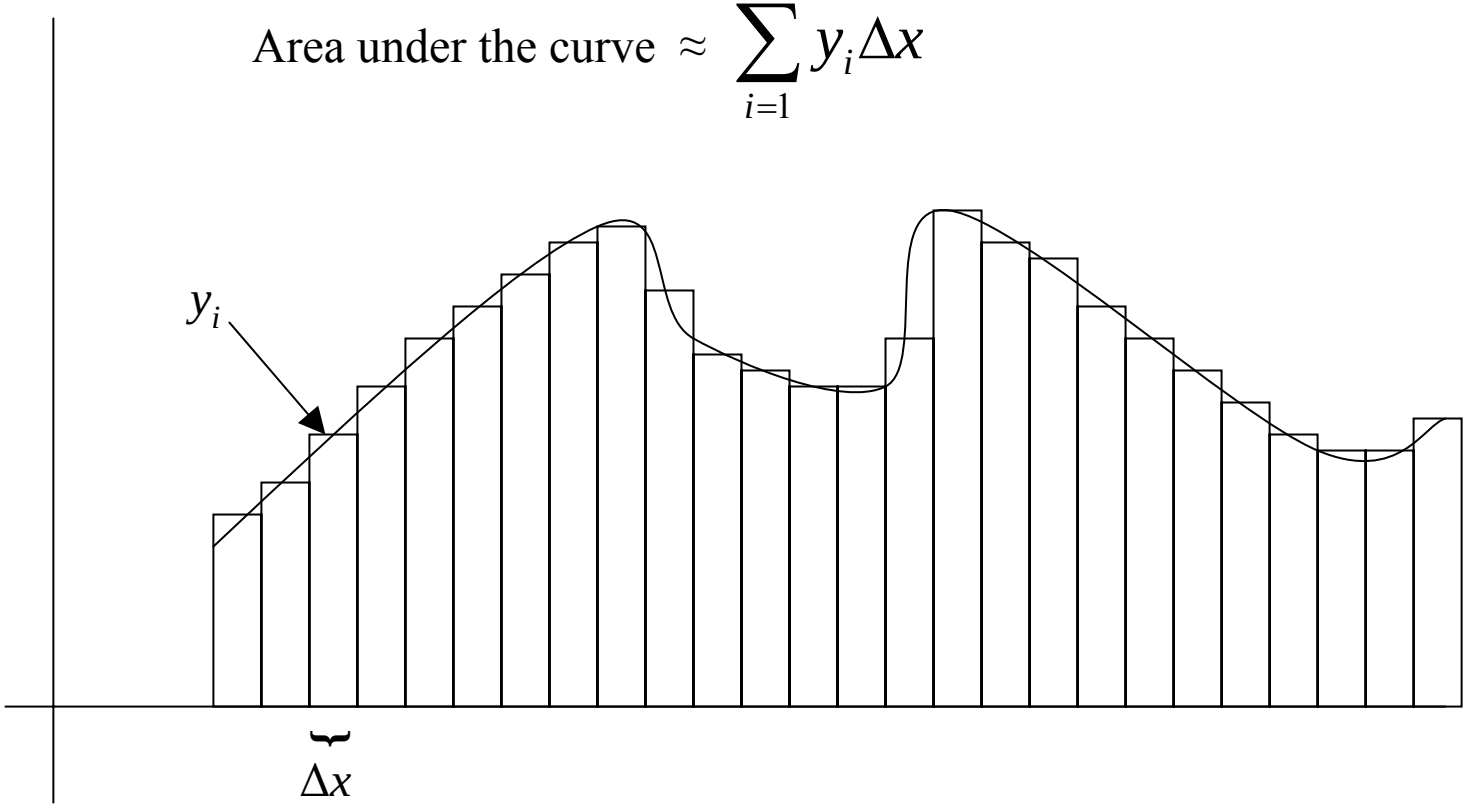
I Want the Area Under This Curve!



How can I get the area under this curve?

A Riemann Sum

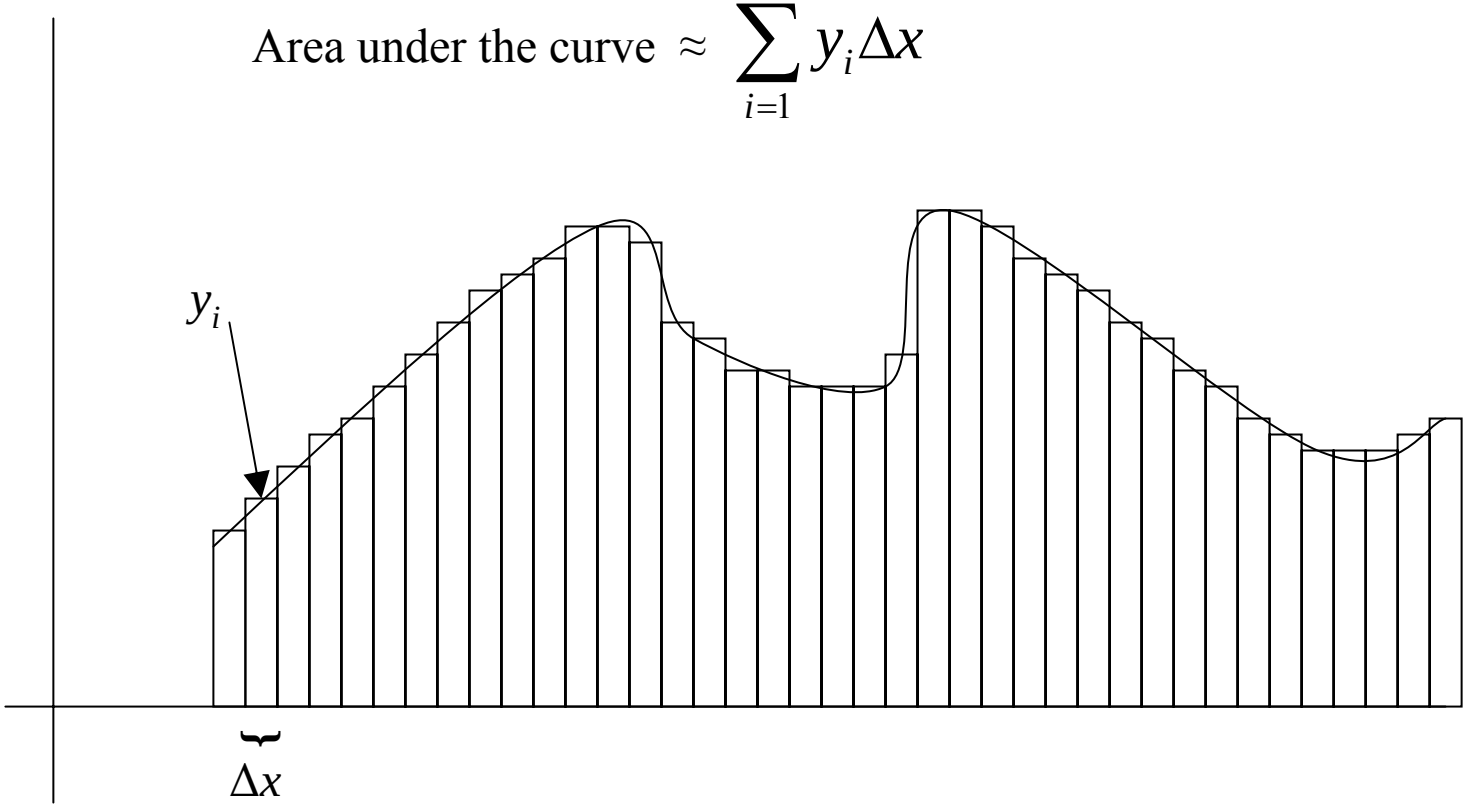
$$\text{Area under the curve} \approx \sum_{i=1}^{n} y_i \Delta x$$



C'est n'est un area under the curve: it's **approximate**!

A Better Riemann Sum

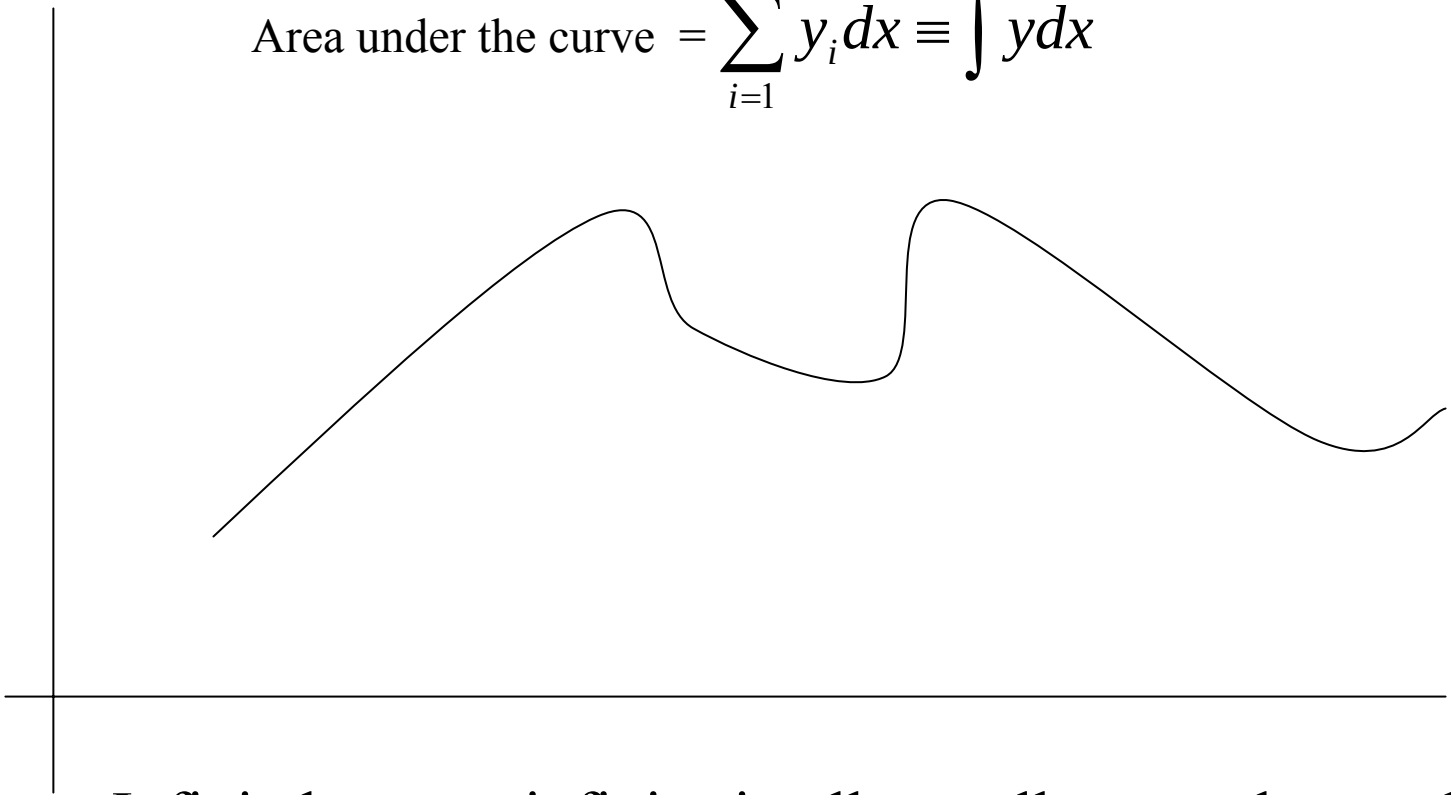
$$\text{Area under the curve} \approx \sum_{i=1}^{n} y_i \Delta x$$



More, smaller rectangles produce a **better approximation**.

The Best Riemann Sum

$$\text{Area under the curve} = \sum_{i=1}^{\infty} y_i dx \equiv \int y dx$$



Infinitely many infinitesimally small rectangles produce the area.

Differential Equations

A differential equation is an equation in which differentials (e.g., *dx*) appear as variables.

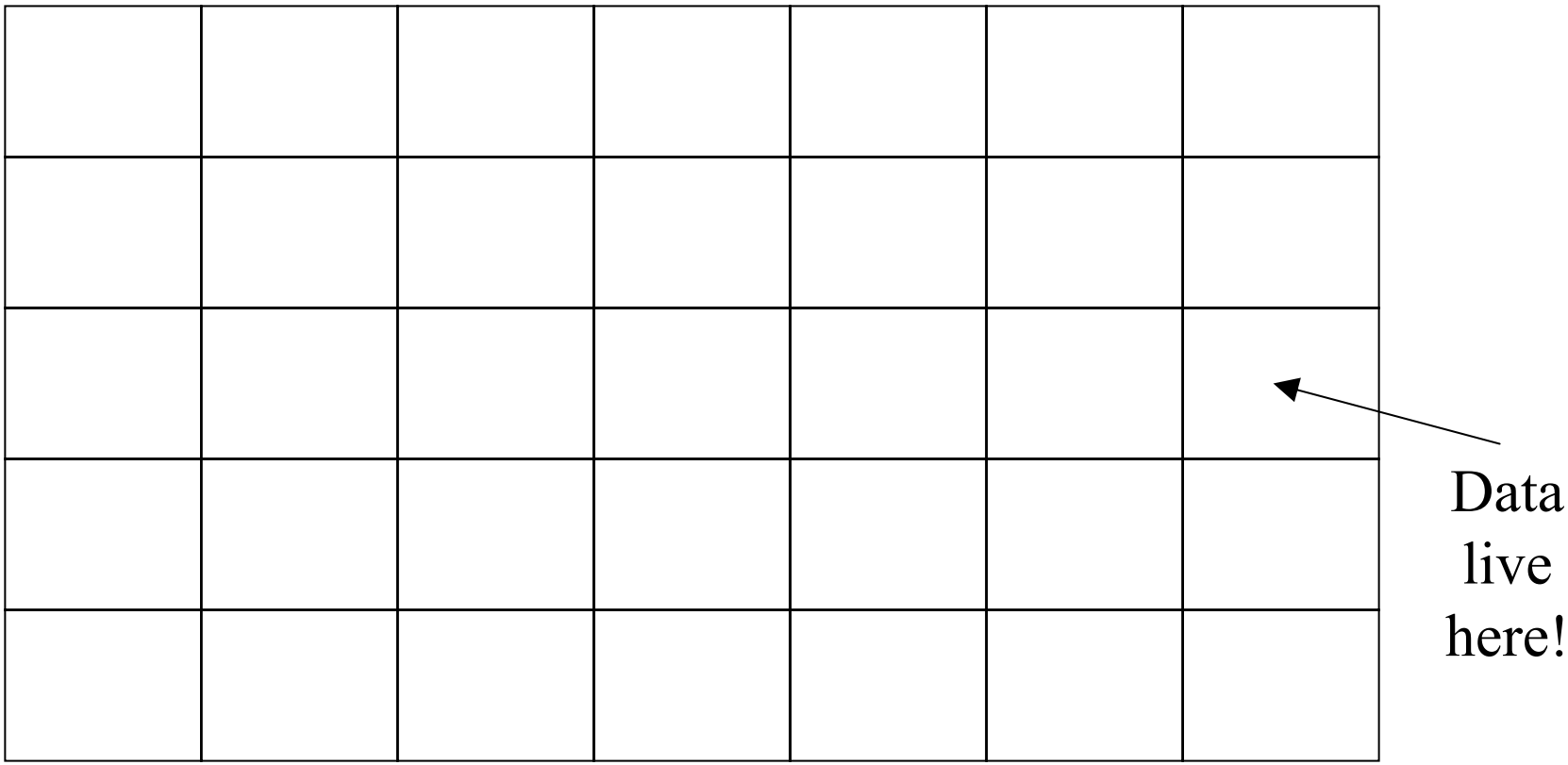
Most physics is best expressed as differential equations.

Very simple differential equations can be solved in "closed form," meaning that a bit of algebraic manipulation gets the exact answer.

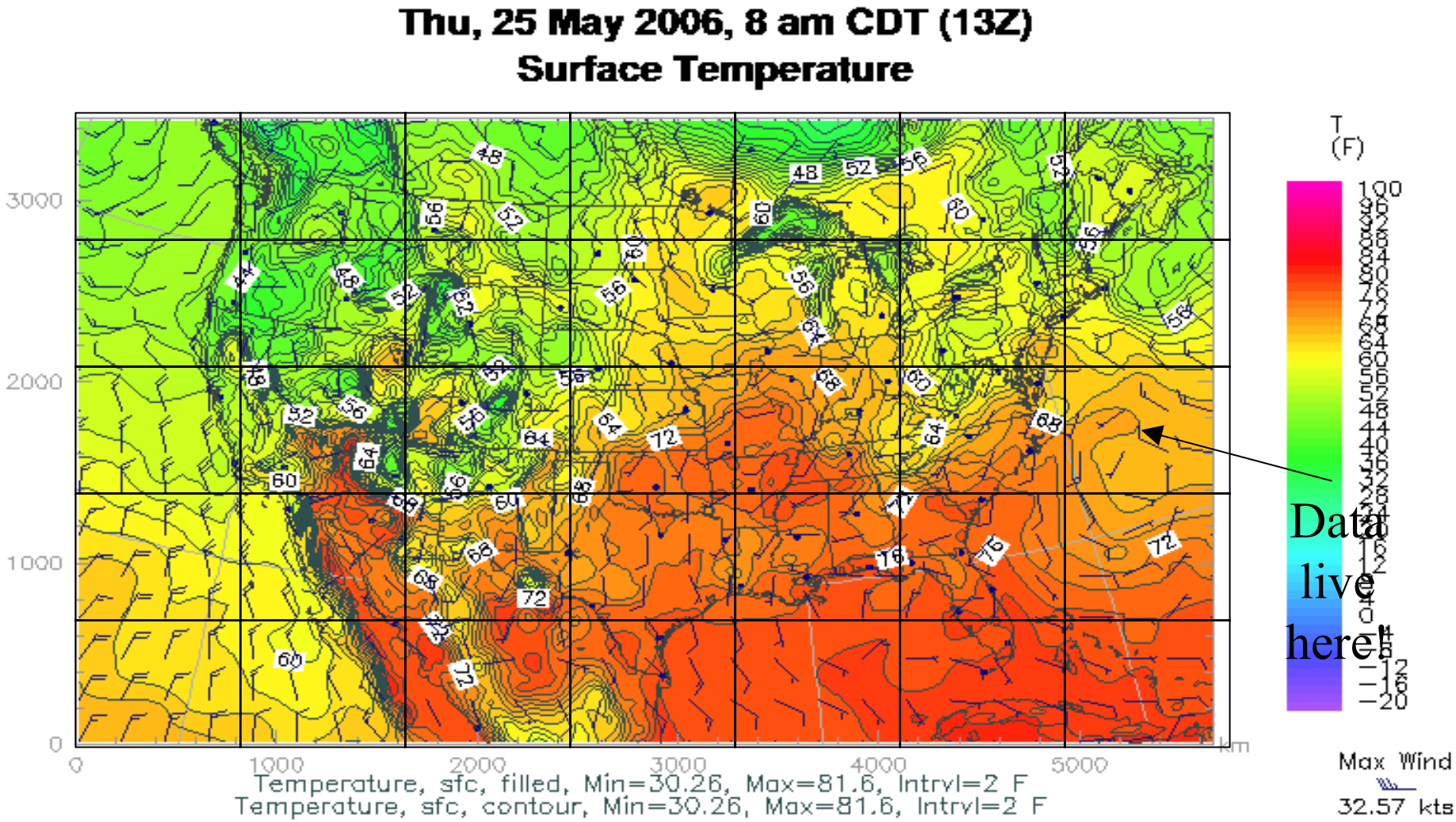
Interesting differential equations, like the ones governing interesting physics, can't be solved in close form.

Solution: approximate!

A Discrete Mesh of Data



A Discrete Mesh of Data



Finite Difference

A typical (though not the only) way of approximating the solution of a differential equation is through finite differencing: convert each dx (infinitely thin) into a Δx (has finite width).

Navier-Stokes Equation

$$\frac{F_i}{V} = \frac{\partial}{\partial x_j}\left[\eta\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) + \lambda\delta_{ij}\nabla\cdot\mathrm{u}\right]$$

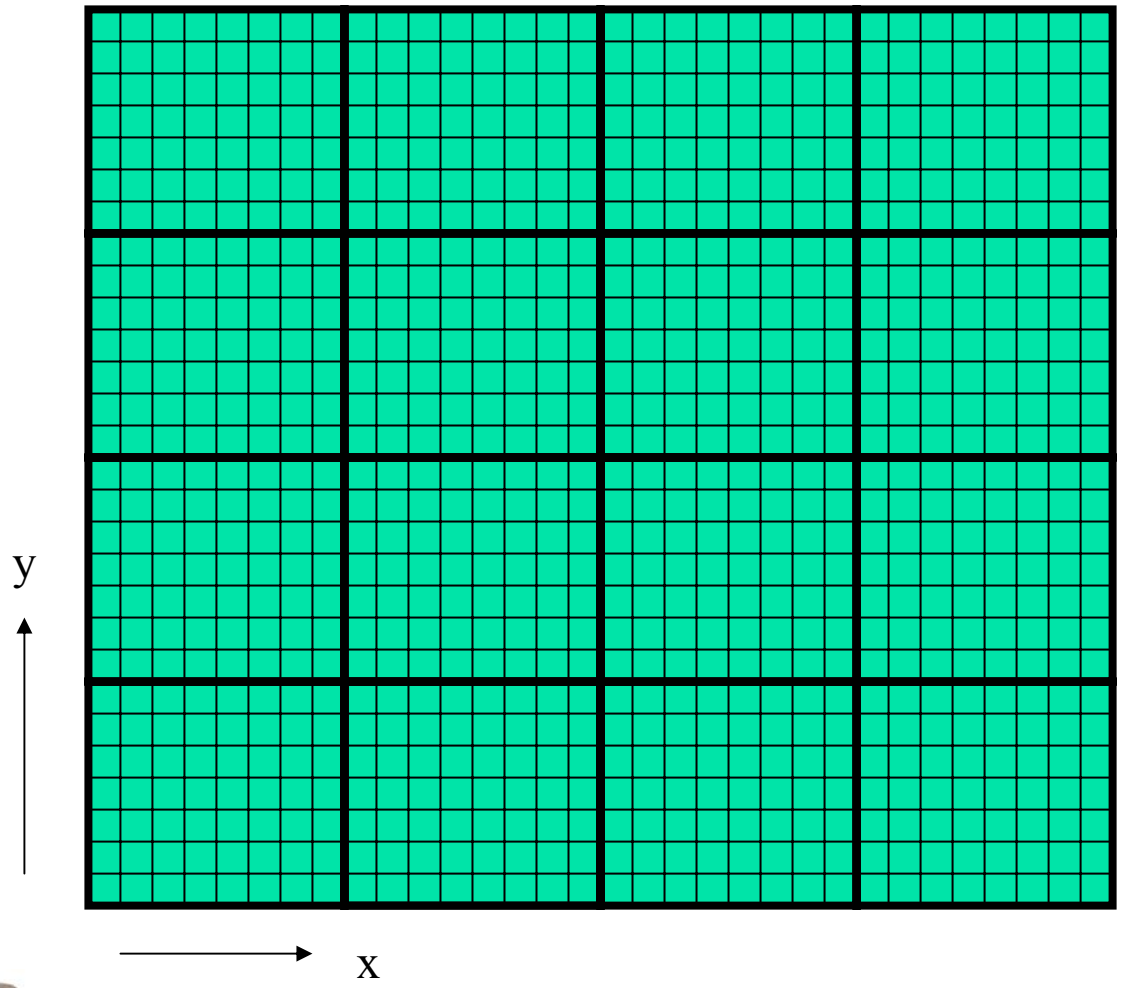
Differential Equation

$$\frac{F_i}{V} = \frac{\Delta}{\Delta x_j}\left[\eta\left(\frac{\Delta u_i}{\Delta x_j} + \frac{\Delta u_j}{\Delta x_i}\right) + \lambda\delta_{ij}\nabla\cdot\mathrm{u}\right]$$

Finite Difference Equation

The Navier-Stokes equations governs the movement of fluids (water, air, etc).

Cartesian Coordinates



Structured Mesh

A ***structured mesh*** is like the mesh on the previous slide. It's nice and regular and rectangular, and can be stored in a standard Fortran or C or C++ array of the appropriate dimension and shape.

Flow in Structured Meshes

When calculating flow in a structured mesh, you typically use a finite difference equation, like so:

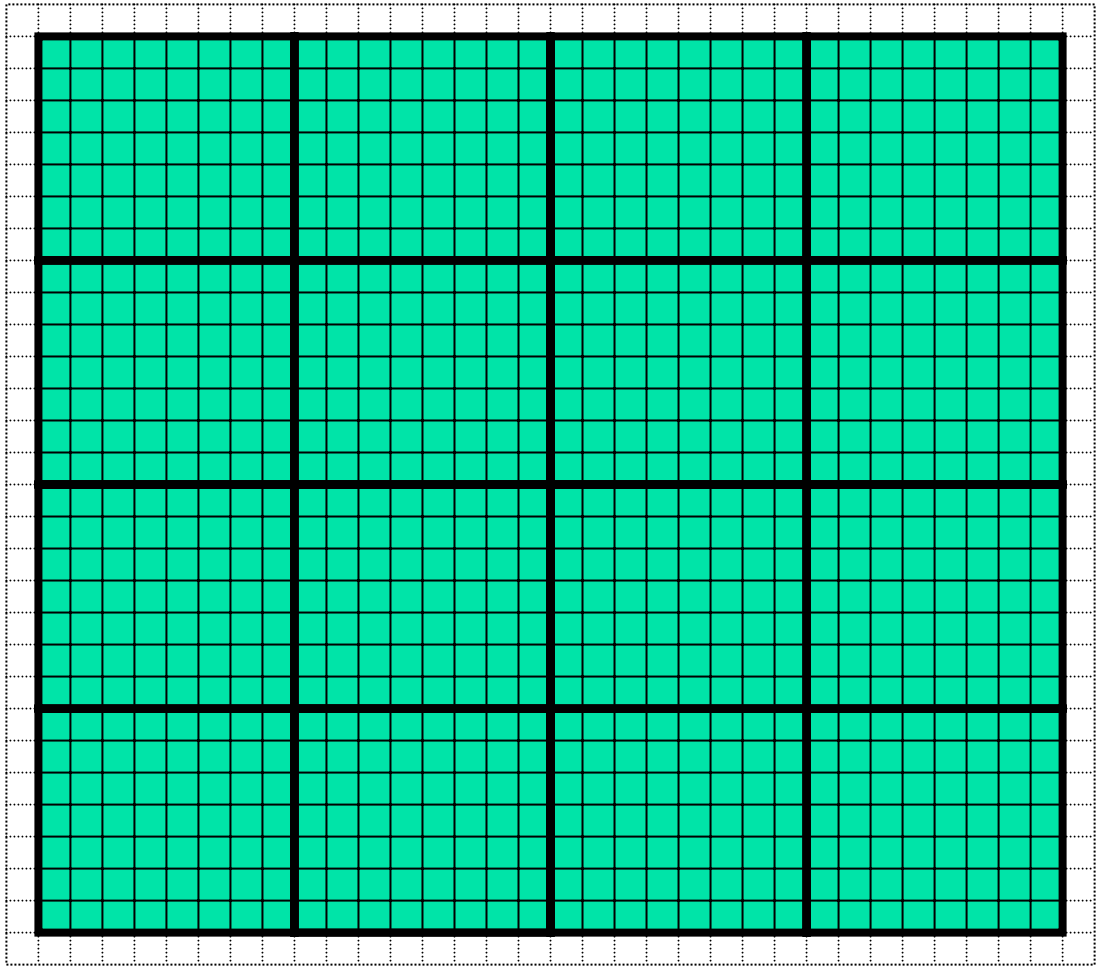
$$unew_{i,j} = F(\Delta t, uold_{i,j}, uold_{i-1,j}, uold_{i+1,j}, uold_{i,j-1}, uold_{i,j+1})$$

for some function F, where $uold_{i,j}$ is at time t and $unew_{i,j}$ is at time $t + \Delta t$.

In other words, you calculate the new value of $u_{i,j}$, based on its old value as well as the old values of its immediate neighbors.

Actually, it may use neighbors a few farther away.

Ghost Boundary Zones



Ghost Boundary Zones

We want to calculate values in the part of the mesh that we care about, but to do that, we need values on the boundaries.

For example, to calculate $unew_{1,1}$, you need $uold_{0,1}$ and $uold_{1,0}$.

Ghost boundary zones are mesh zones that aren't really part of the problem domain that we care about, but that hold boundary data for calculating the parts that we do care about.

Using Ghost Boundary Zones

A good basic algorithm for flow that uses ghost boundary zones is:

```
DO timestep = 1, number_of_timesteps
  CALL fill_ghost_boundary(…)
  CALL advance_to_new_from_old(…)
END DO
```

This approach generally works great on a serial code.

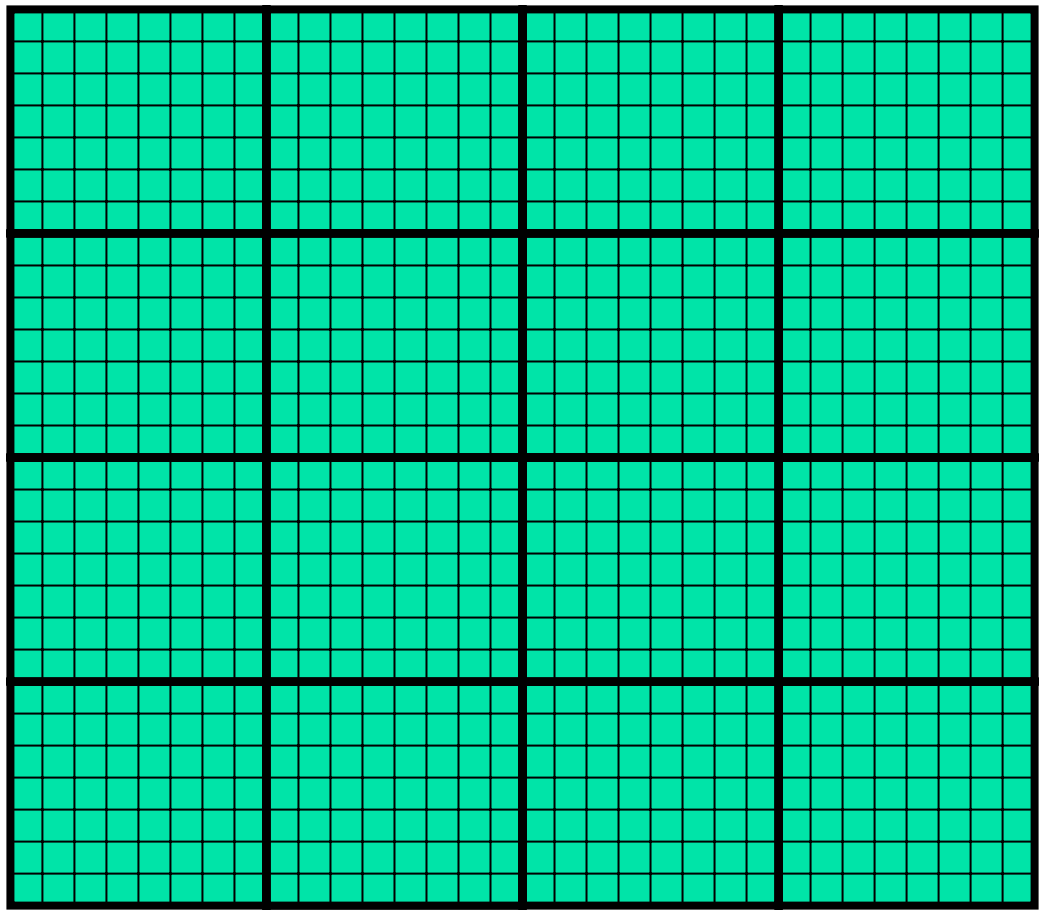
Ghost Boundary Zones in MPI

What if you want to parallelize a Cartesian flow code in MPI?

You'll need to:

- decompose the mesh into ***submeshes***;
- figure out how each submesh talks to its neighbors.

Data Decomposition



Data Decomposition

We want to split the data into chunks of equal size, and give each chunk to a processor to work on.

Then, each processor can work independently of all of the others, except when it's exchanging boundary data with its neighbors.

MPI_Cart_*

MPI supports exactly this kind of calculation, with a set of functions **MPI_Cart_***:

- **MPI_Cart_create**
- **MPI_Cart_coords**
- **MPI_Cart_shift**

These routines create and describe a new communicator, one that replaces **MPI_COMM_WORLD** in your code.

MPI_Sendrecv

MPI_Sendrecv is just like an **MPI_Send** followed by an **MPI_Recv**, except that it's much better than that.

With **MPI_Send** and **MPI_Recv**, these are your choices:

Everyone calls **MPI_Recv**, and then everyone calls **MPI_Send**.

Everyone calls **MPI_Send**, and then everyone calls **MPI_Recv**.

Some call **MPI_Send** while others call **MPI_Recv**, and then they swap roles.

Why not Recv then Send?

Suppose that everyone calls **MPI_Recv**, and then everyone calls **MPI_Send**.

```
MPI_Recv(incoming_data, ...);
MPI_Send(outgoing_data, ...);
```

Well, these routines are ***blocking***, meaning that the communication has to complete before the process can continue on farther into the program.

That means that, when everyone calls **MPI_Recv**, they're waiting for someone else to call **MPI_Send**.

We call this ***deadlock***.

Officially, the MPI standard forbids this approach.

Why not Send then Recv?

Suppose that everyone calls **`MPI_Send`**, and then everyone calls **`MPI_Recv`**:

```
MPI_Send(outgoing_data, ...);
MPI_Recv(incoming_data, ...);
```

Well, this will only work if there's enough ***buffer space*** available to hold everyone's messages until after everyone is done sending.

Sometimes, there isn't enough buffer space.

Officially, the MPI standard allows MPI implementers to support this, but it's not part of the official MPI standard; that is, a particular MPI implementation doesn't have to allow it.

Alternate Send and Recv?

Suppose that some processors call **MPI_Send** while others call **MPI_Recv**, and then they swap roles:

```
if ((my_rank % 2) == 0) {
   MPI_Send(outgoing_data, ...);
   MPI_Recv(incoming_data, ...);
}
else {
   MPI_Recv(incoming_data, ...);
   MPI_Send(outgoing_data, ...);
}
```

This will work, and is sometimes used, but it can be painful to manage – especially if you have an odd number of processors.

MPI_Sendrecv

MPI_Sendrecv allows each processor to simultaneously send to one processor and receive from another.

For example, P_1 could send to P_0 while simultaneously receiving from P_2 .

This is exactly what we need in Cartesian flow: we want the boundary data to come in from the east while we send boundary data out to the west, and then vice versa.

These are called ***shifts***.

MPI_Sendrecv

```
MPI_Sendrecv(
  westward_send_buffer,
  westward_send_size, MPI_REAL,
  west_neighbor_process, westward_tag,
  westward_recv_buffer,
  westward_recv_size, MPI_REAL,
  east_neighbor_process, westward_tag,
  cartesian_communicator, mpi_status);
```

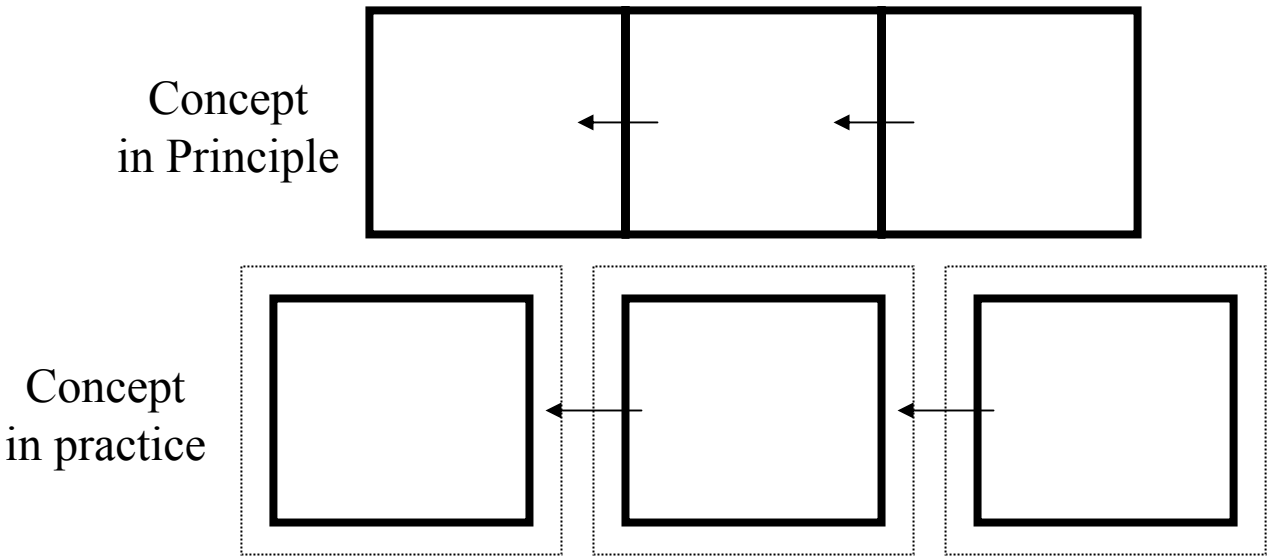
This call sends to **west_neighbor_process** the data in **westward_send_buffer**, and at the same time receives from **east_neighbor_process** a bunch of data that end up in **westward_recv_buffer**.

Why `MPI_Sendrecv`?

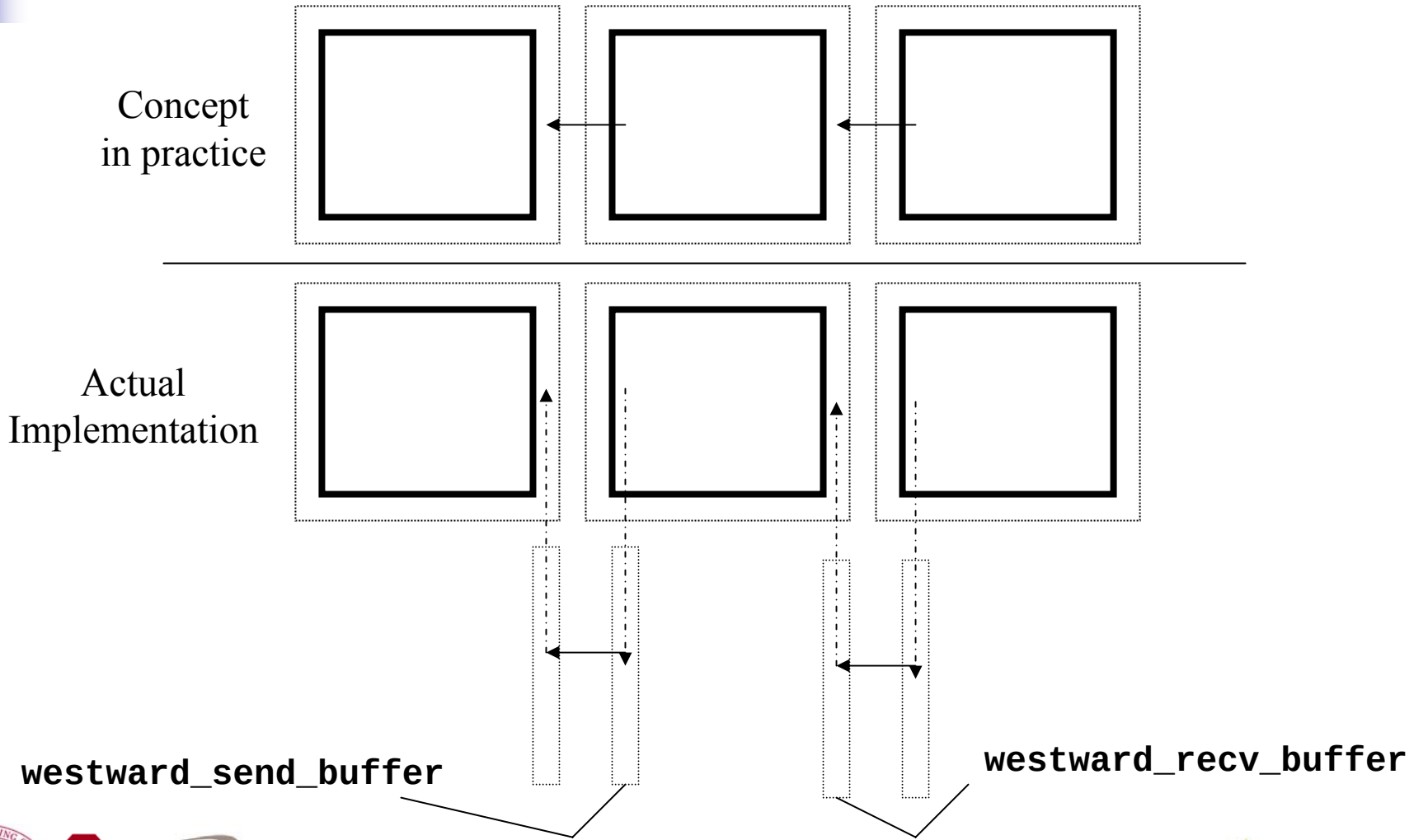
The advantage of **`MPI_Sendrecv`** is that it allows us the luxury of no longer having to worry about who should send when and who should receive when.

This is exactly what we need in Cartesian flow: we want the boundary information to come in from the east while we send boundary information out to the west – without us having to worry about deciding who should do what to who when.

MPI_Sendrecv



MPI_Sendrecv



To Learn More

http://www.oscer.ou.edu/

Thanks for your attention!

Questions?