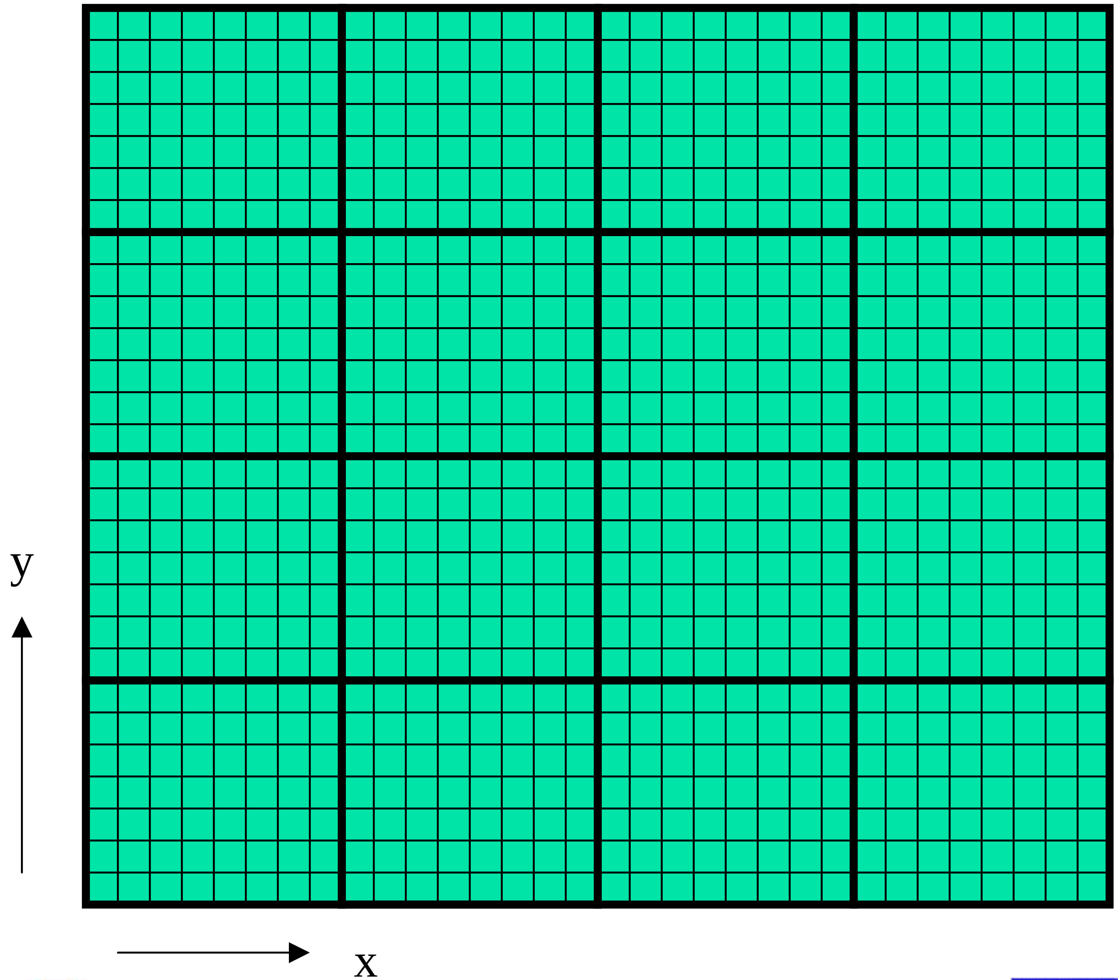
Parallel & Cluster Computing

Distributed Cartesian Meshes

National Computational Science Institute
August 8-14 2004

Paul Gray, University of Northern Iowa
David Joiner, Shodor Education Foundation
Tom Murphy, Contra Costa College
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College

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 array of the appropriate dimension.

Flow in Structured Meshes

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

$unew_{i,j} =$

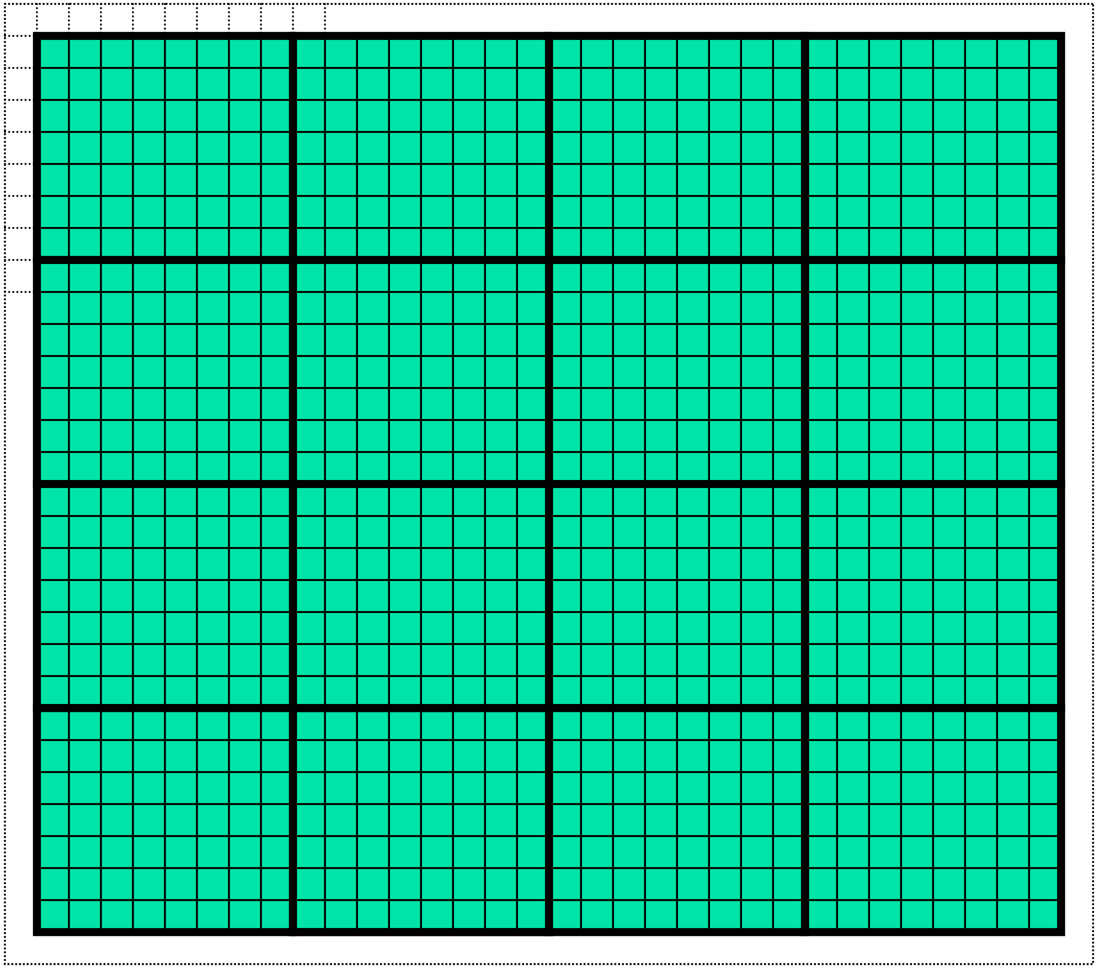
$F(\Delta t, uold_{i,j}, uold_{i-1,j}, uold_{i+1,j}, uold_{i,j-1}, uold_{i,j+1})$

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 Zones



Ghost 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.

Ghost 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 Zones

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

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

This approach generally works great on a serial code.

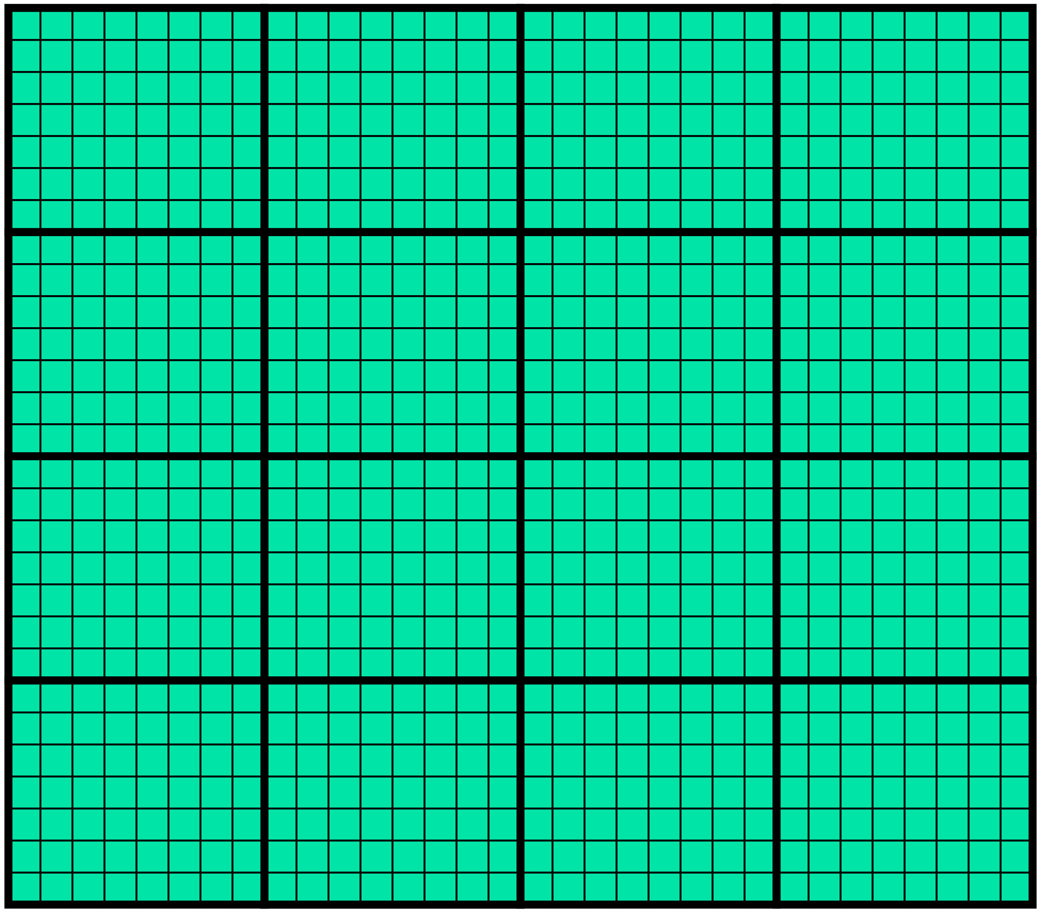
Ghost Zones in MPI

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 MPI_Sendrecv?

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

Well, these routines are synchronous (also called 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 `MPI_Sendrecv`?

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

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.

Why `MPI_Sendrecv`?

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

This will work, and is sometimes used, but it can be a pain in the rear end to manage.

Why `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 information to come in from the east while we send boundary information 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, ok_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**.

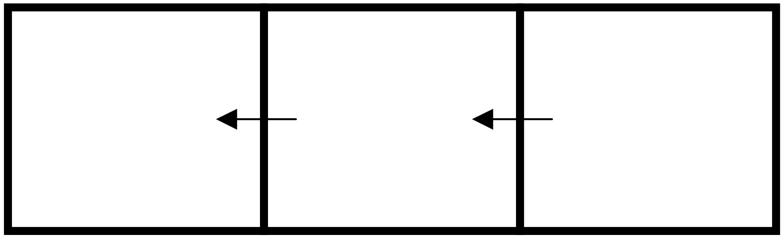
Why `MPI_Sendrecv`?

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.

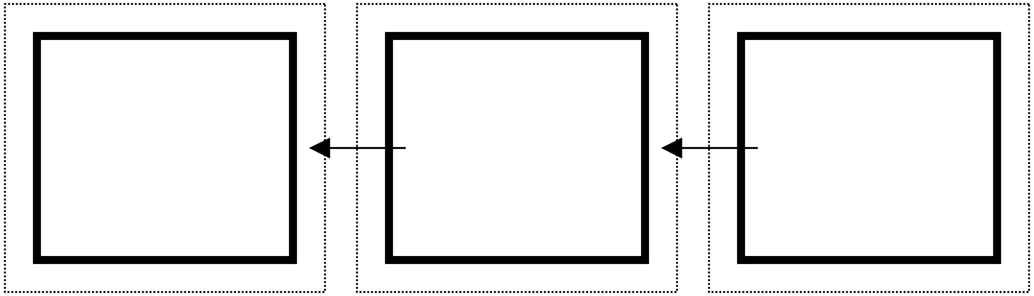
This is exactly what we need in Cartesian flow: we want the boundary information to come in from the east while we send boundary information out to the west – without us having to worry about deciding who should do what to who when.

MPI_Sendrecv

Concept
in Principle



Concept
in practice



MPI_Sendrecv