Parallel Programming & Cluster Computing

Applications and Types of Parallelism

Henry Neeman, Director

OU Supercomputing Center for Education & Research

University of Oklahoma Information Technology

Oklahoma Supercomputing Symposium, Tue Oct 5 2010

Outline

- Monte Carlo: Client-Server
- N-Body: Task Parallelism
- Transport: Data Parallelism

Monte Carlo: Client-Server

[1]

Embarrassingly Parallel

An application is known as ***embarrassingly parallel*** if its parallel implementation:

1. can straightforwardly be broken up into roughly equal amounts of work per processor, **AND**
2. has minimal parallel overhead (for example, communication among processors).

We **love** embarrassingly parallel applications, because they get **near-perfect parallel speedup**, sometimes with modest programming effort.

Embarrassingly parallel applications are also known as ***loosely coupled***.

Monte Carlo Methods

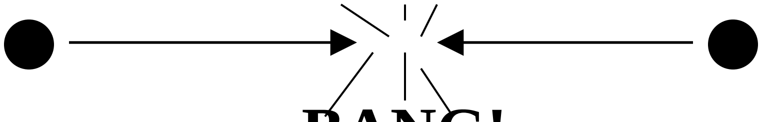
Monte Carlo is a European city where people gamble; that is, they play games of chance, which involve randomness.

Monte Carlo methods are ways of simulating (or otherwise calculating) physical phenomena based on randomness.

Monte Carlo simulations typically are embarrassingly parallel.

Monte Carlo Methods: Example

Suppose you have some physical phenomenon. For example, consider High Energy Physics, in which we bang tiny particles together at incredibly high speeds.



BANG!

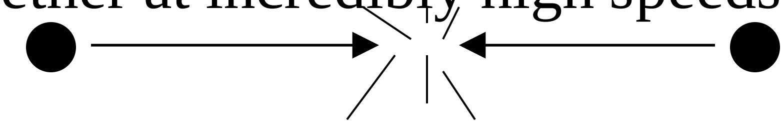
We want to know, say, the average properties of this phenomenon.

There are infinitely many ways that two particles can be banged together.

So, we can’t possibly simulate all of them.

Monte Carlo Methods: Example

Suppose you have some physical phenomenon. For example, consider High Energy Physics, in which we bang tiny particles together at incredibly high speeds.



BANG!

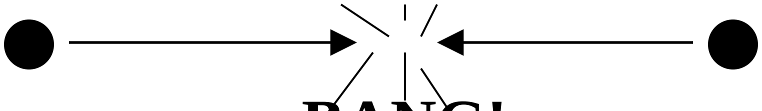
There are infinitely many ways that two particles can be banged together.

So, we can't possibly simulate all of them.

Instead, we can **randomly choose a finite subset** of these infinitely many ways and simulate only the subset.

Monte Carlo Methods: Example

Suppose you have some physical phenomenon. For example, consider High Energy Physics, in which we bang tiny particles together at incredibly high speeds.



BANG!

There are infinitely many ways that two particles can be banged together.

We randomly choose a finite subset of these infinitely many ways and simulate only the subset.

The average of this subset will be close to the actual average.

Monte Carlo Methods

In a Monte Carlo method, you randomly generate a large number of example cases (***realizations***) of a phenomenon, and then take the average of the properties of these realizations.

When the average of the realizations converges (that is, doesn’t change substantially if new realizations are generated), then the Monte Carlo simulation stops.

MC: Embarrassingly Parallel

Monte Carlo simulations are embarrassingly parallel, because each realization is completely independent of all of the other realizations.

That is, if you're going to run a million realizations, then:

1. you can straightforwardly break up into roughly 1M / N_p chunks of realizations, one chunk for each of the N_p processors, **AND**
2. the only parallel overhead (for example, communication) comes from tracking the average properties, which doesn't have to happen very often.

Serial Monte Carlo

Suppose you have an existing serial Monte Carlo simulation:

```
PROGRAM monte_carlo
  CALL read_input(…)
  DO realization = 1, number_of_realizations
    CALL generate_random_realization(…)
    CALL calculate_properties(…)
  END DO
  CALL calculate_average(…)
END PROGRAM monte_carlo
```

How would you parallelize this?

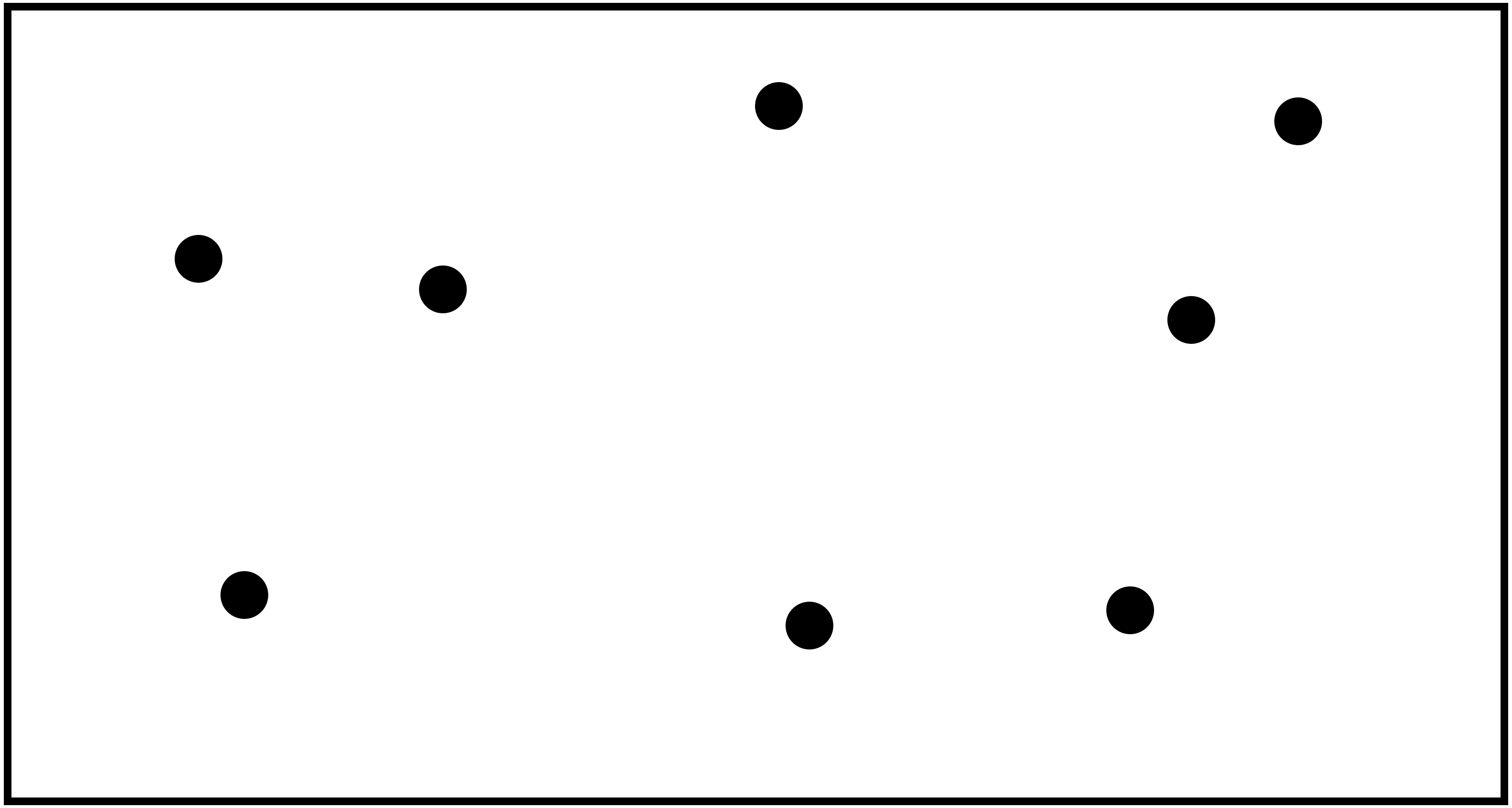
Parallel Monte Carlo

```
PROGRAM monte_carlo
  [MPI startup]
  IF (my_rank == server_rank) THEN
    CALL read_input(…)
  END IF
  CALL MPI_Bcast(…)
  DO realization = 1, number_of_realizations
    CALL generate_random_realization(…)
    CALL calculate_realization_properties(…)
    CALL calculate_local_running_average(...)
  END DO
  IF (my_rank == server_rank) THEN
      [collect properties]
  ELSE
      [send properties]
  END IF
  CALL calculate_global_average_from_local_averages(…)
  CALL output_overall_average(...)
  [MPI shutdown]
END PROGRAM monte_carlo
```

N-Body: Task Parallelism and Collective Communication

[2]

N Bodies



N-Body Problems

An ***N-body problem*** is a problem involving *N* “bodies” – that is, particles (that is, stars, atoms) – each of which applies a force to all of the others.

For example, if you have *N* stars, then each of the *N* stars exerts a force (gravity) on all of the other *N*–1 stars.

Likewise, if you have *N* atoms, then every atom exerts a force (nuclear) on all of the other *N*–1 atoms.

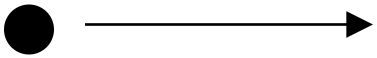
1-Body Problem

When *N* is 1, you have a simple 1-Body Problem: a single particle, with no forces acting on it.

Given the particle's position P and velocity V at some time t_0, you can trivially calculate the particle's position at time $t_0+\Delta t$:

$$P(t_0+\Delta t) = P(t_0) + V\Delta t$$

$$V(t_0+\Delta t) = V(t_0)$$

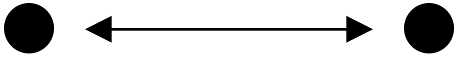


2-Body Problem

When *N* is 2, you have – surprise! – a ***2-Body Problem***: exactly 2 particles, each exerting a force that acts on the other.

The relationship between the 2 particles can be expressed as a differential equation that can be solved analytically, producing a closed-form solution.

So, given the particles' initial positions and velocities, you can trivially calculate their positions and velocities at any later time.

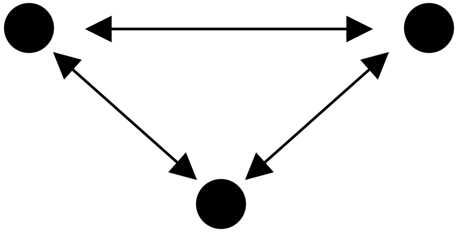


3-Body Problem

When *N* is 3, you have – surprise! – a ***3-Body Problem***: exactly 3 particles, each exerting a force that acts on the other.

The relationship between the 3 particles can be expressed as a differential equation that can be solved using an infinite series, producing a closed-form solution, due to Karl Fritiof Sundman in 1912.

However, in practice, the number of terms of the infinite series that you need to calculate to get a reasonable solution is so large that the infinite series is impractical, so you're stuck with the generalized formulation.



N-Body Problems ($N \geq 3$)

For *N* greater than 3 (and for *N* of 3 in practice), no one knows how to solve the equations to get a closed form solution.

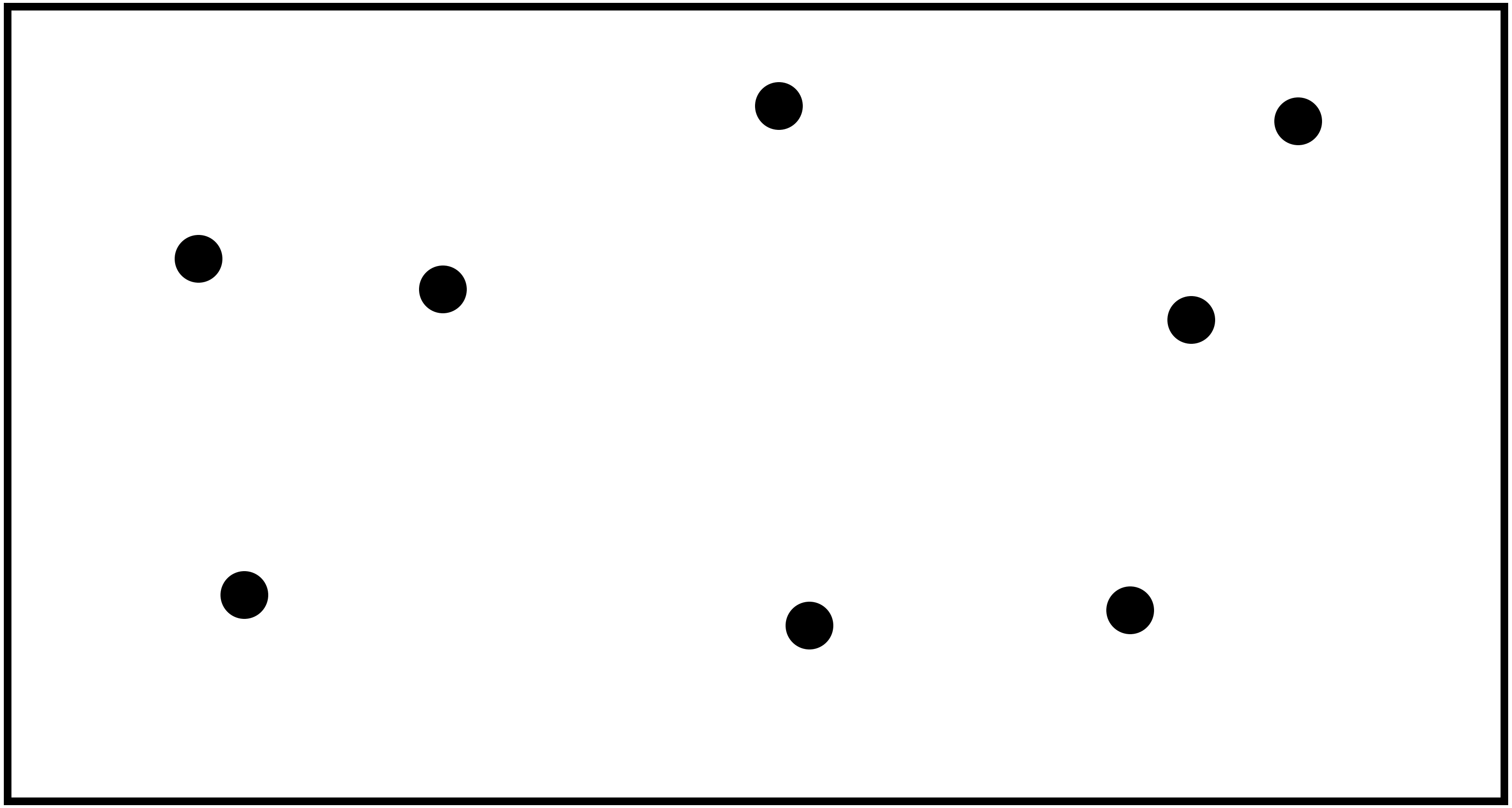
So, numerical simulation is pretty much the only way to study groups of 3 or more bodies.

Popular applications of N-body codes include:

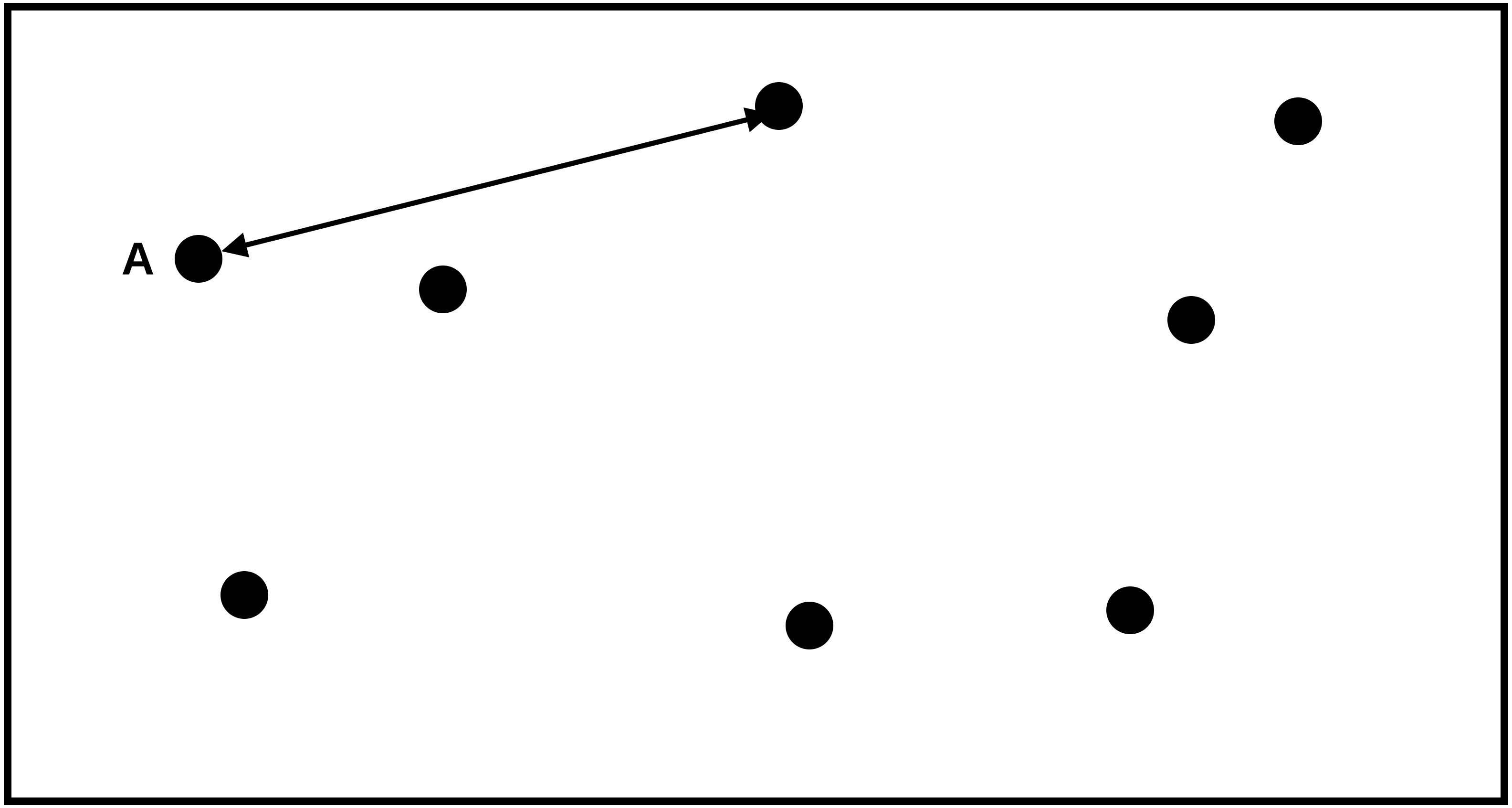
- astronomy (that is, galaxy formation, cosmology);
- chemistry (that is, protein folding, molecular dynamics).

Note that, for *N* bodies, there are on the order of N^2 forces, denoted **O**(N^2).

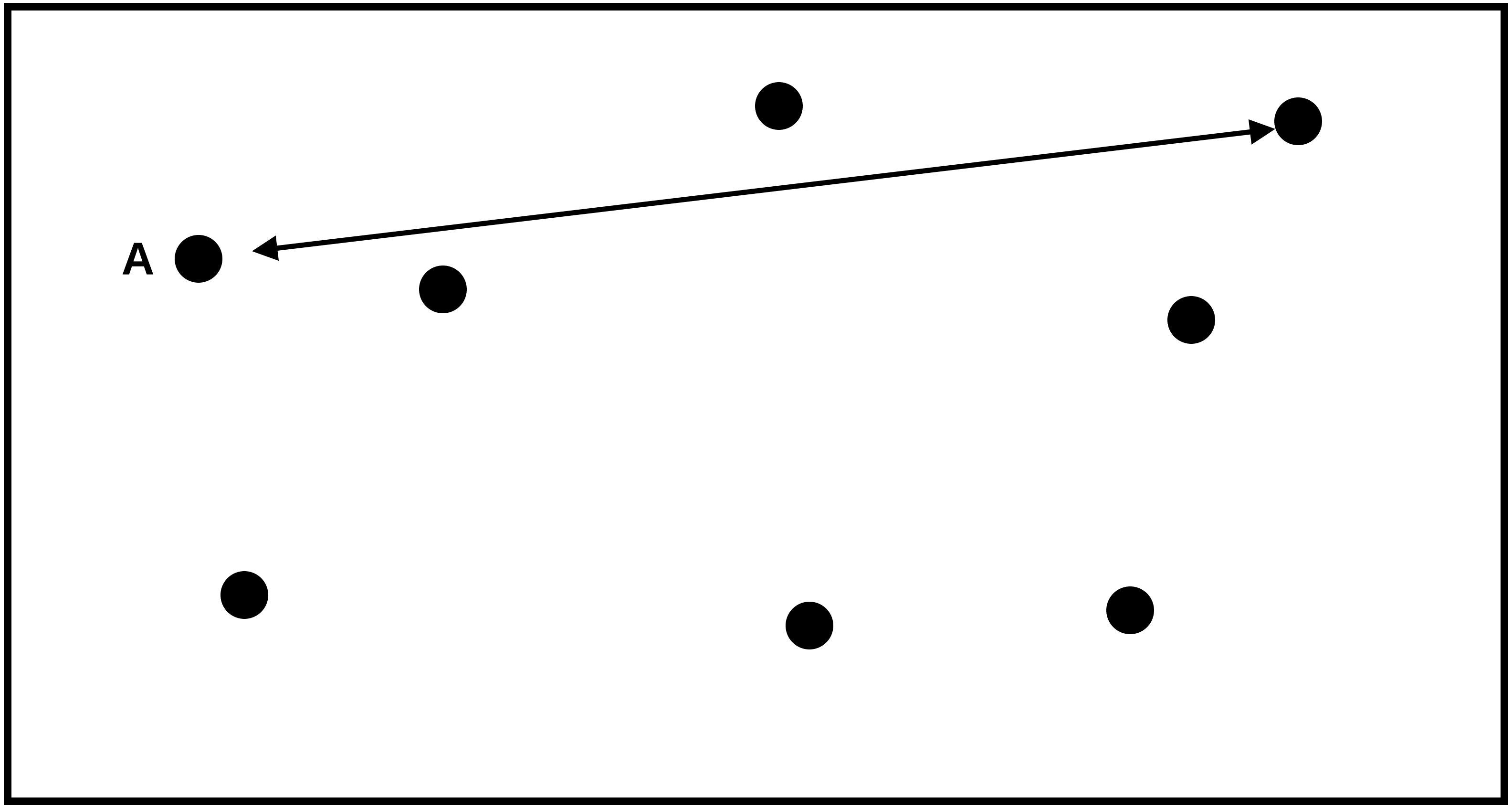
N Bodies



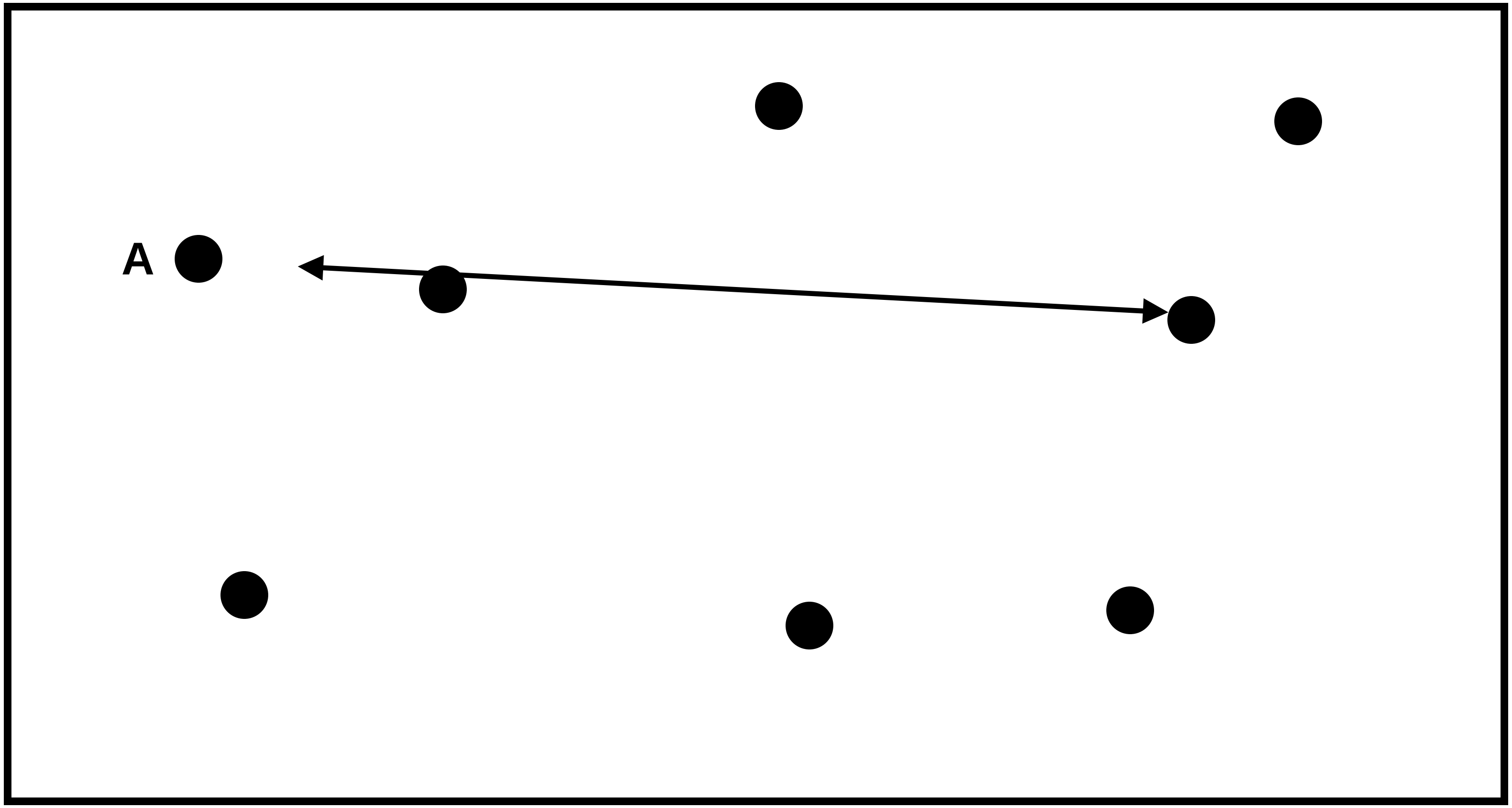
Force #1



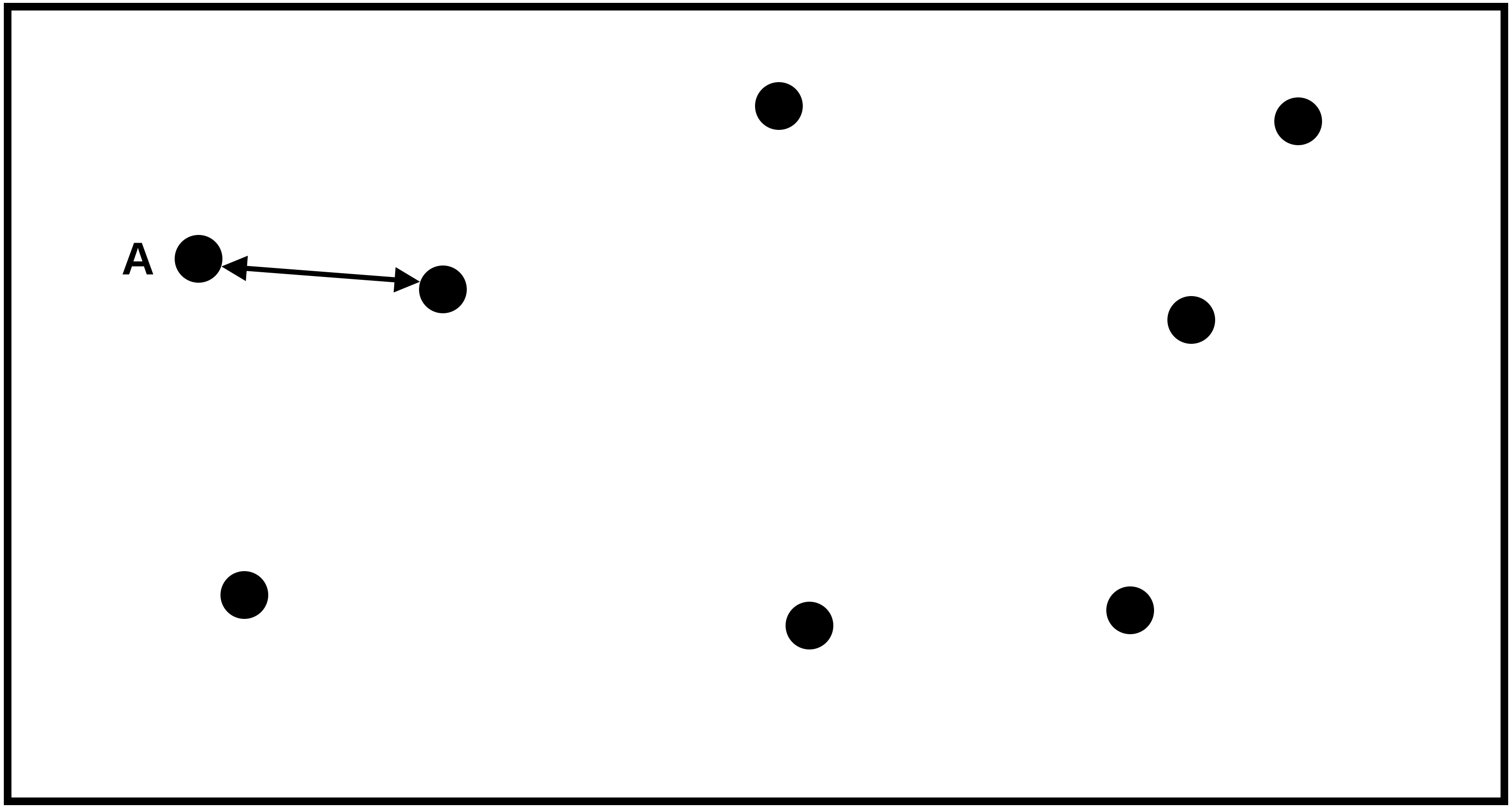
Force #2



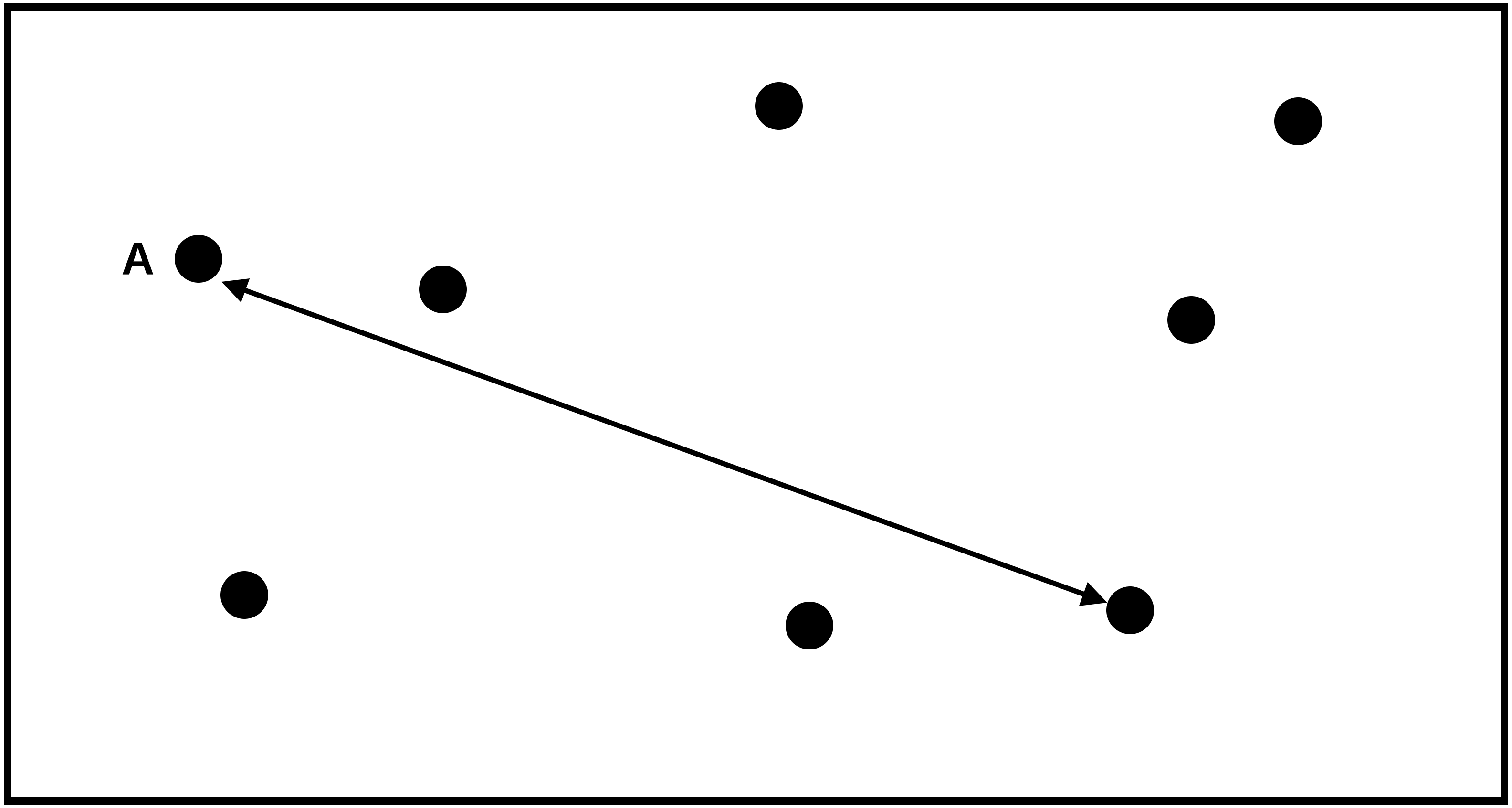
Force #3



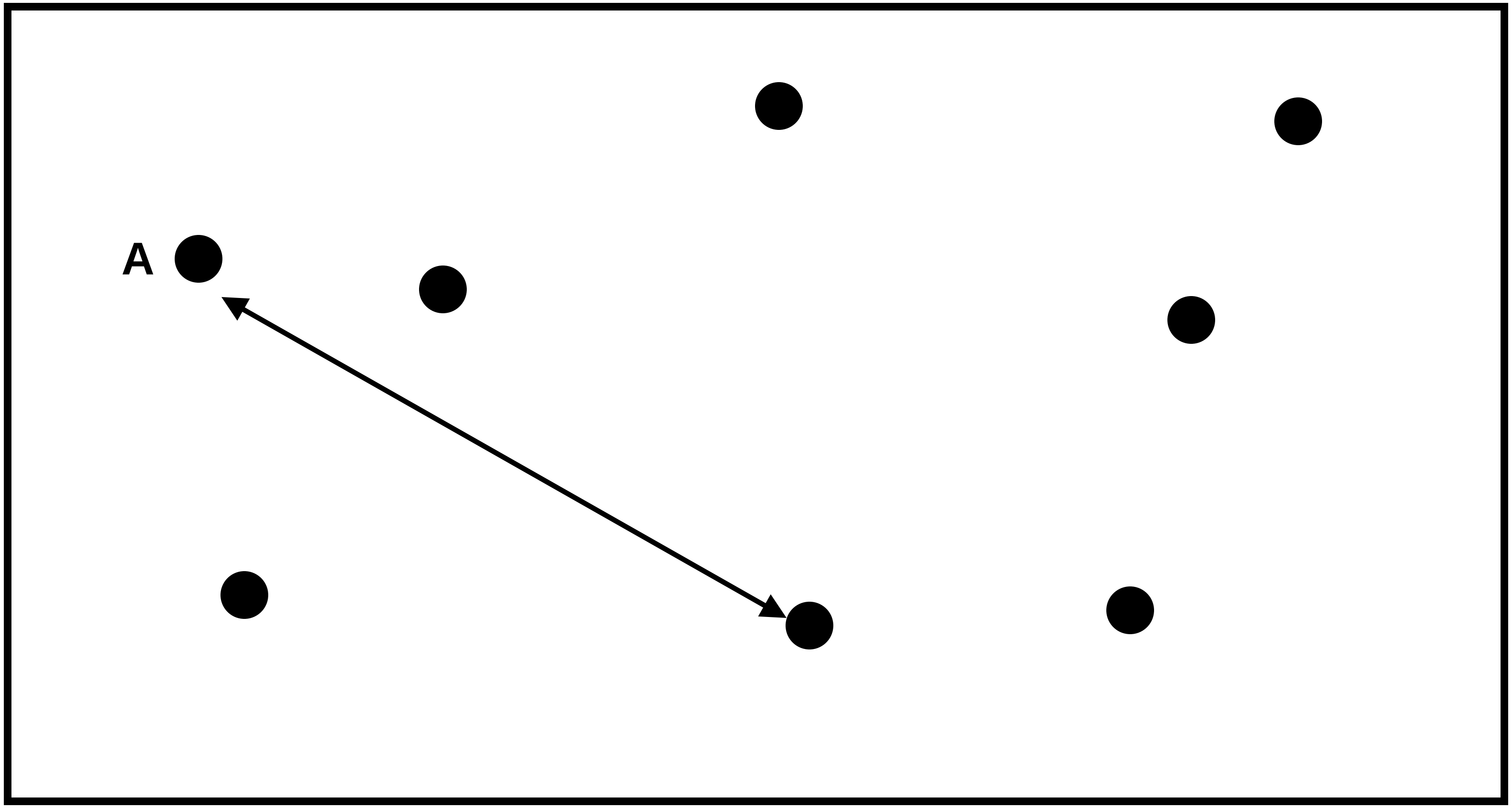
Force #4



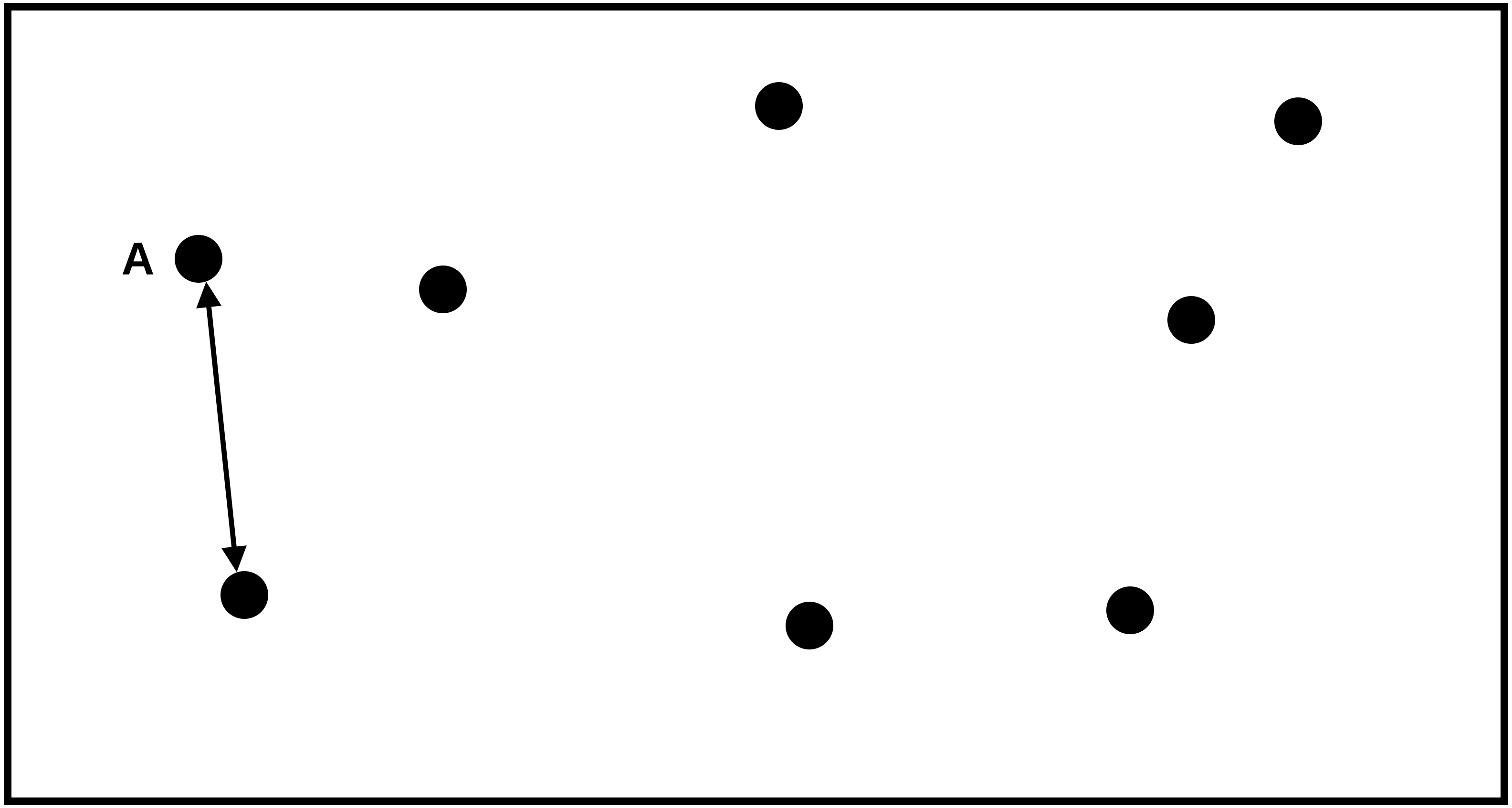
Force #5



Force #6



Force #N-1



N-Body Problems

Given N bodies, each body exerts a force on all of the other $N - 1$ bodies.

Therefore, there are $N \cdot (N - 1)$ forces in total.

You can also think of this as $(N \cdot (N - 1)) / 2$ forces, in the sense that the force from particle A to particle B is the same (except in the opposite direction) as the force from particle B to particle A.

Aside: Big-O Notation

Let’s say that you have some task to perform on a certain number of things, and that the task takes a certain amount of time to complete.

Let’s say that the amount of time can be expressed as a polynomial on the number of things to perform the task on.

For example, the amount of time it takes to read a book might be proportional to the number of words, plus the amount of time it takes to settle into your favorite easy chair.

$$C_1 \bullet N + C_2$$

Big-O: Dropping the Low Term

$$C_1 \bullet N + C_2$$

When *N* is very large, the time spent settling into your easy chair becomes such a small proportion of the total time that it’s virtually zero.

So from a practical perspective, for large *N*, the polynomial reduces to:

$$C_1 \bullet N$$

In fact, for any polynomial, if N is large, then all of the terms except the highest-order term are irrelevant.

Big-O: Dropping the Constant

$$C_1 \cdot N$$

Computers get faster and faster all the time. And there are many different flavors of computers, having many different speeds.

So, computer scientists don’t care about the constant, only about the order of the highest-order term of the polynomial.

They indicate this with Big-**O** notation:

$$\mathbf{O}(N)$$

This is often said as: “of order *N*.”

N-Body Problems

Given N bodies, each body exerts a force on all of the other $N - 1$ bodies.

Therefore, there are $N \cdot (N - 1)$ forces total.

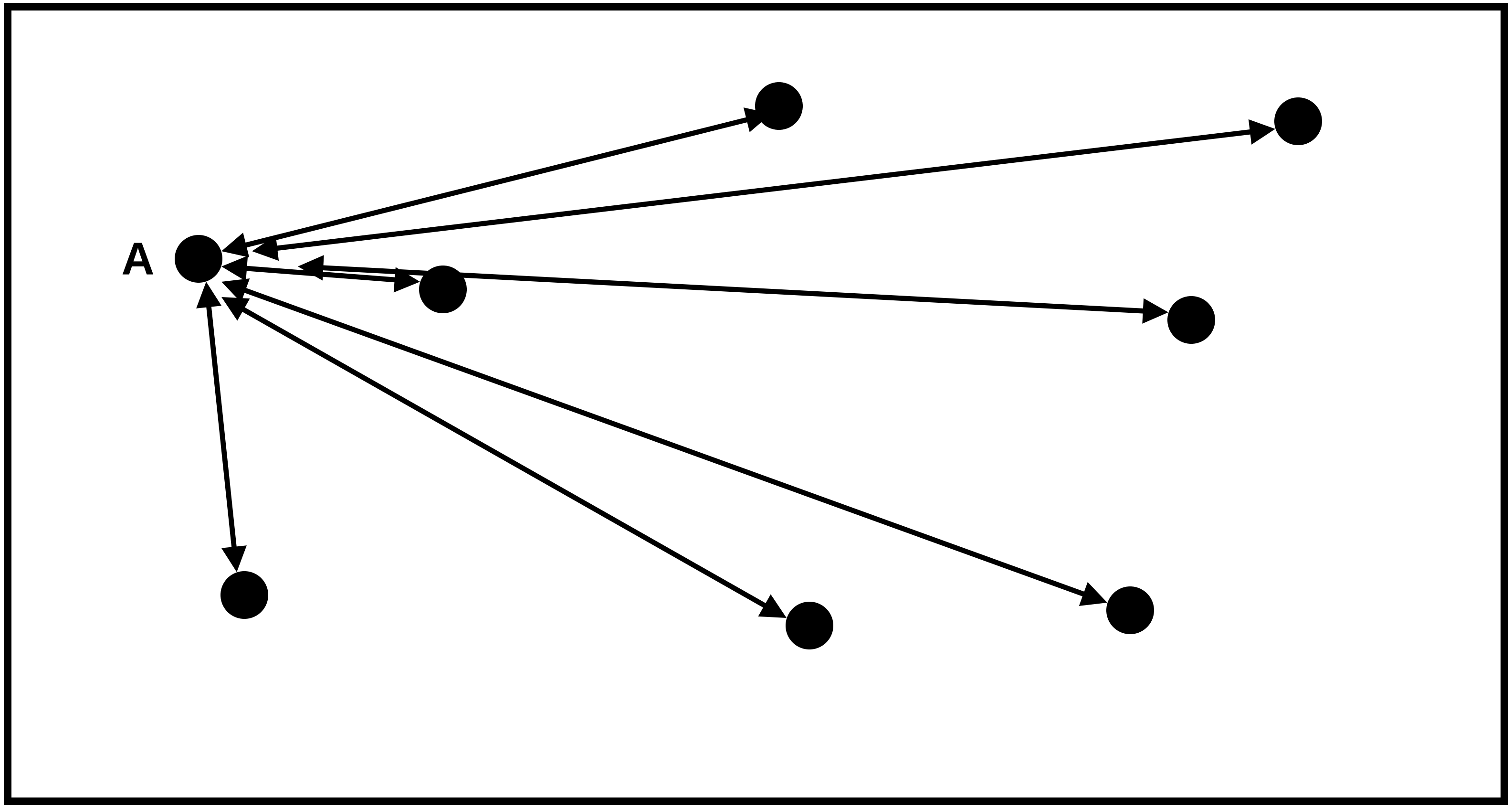
In Big-**O** notation, that's $\mathbf{O}(N^2)$ forces.

So, calculating the forces takes $\mathbf{O}(N^2)$ time to execute.

But, there are only N particles, each taking up the same amount of memory, so we say that N-body codes are of:

- $\mathbf{O}(N)$ spatial complexity (memory)
- $\mathbf{O}(N^2)$ time complexity

O(N^2) Forces



Note that this picture shows only the forces between A and everyone else.

How to Calculate?

Whatever your physics is, you have some function, $\boldsymbol{F}$(A,B), that expresses the force between two bodies A and B.

For example, for stars and galaxies,

$$\boldsymbol{F}(\mathrm{A},\mathrm{B}) = G \cdot m_{\mathrm{A}} \cdot m_{\mathrm{B}} / \mathrm{dist}(\mathrm{A},\mathrm{B})^2$$

where G is the gravitational constant and m is the mass of the body in question.

If you have all of the forces for every pair of particles, then you can calculate their sum, obtaining the force on every particle.

From that, you can calculate every particle’s new position and velocity.

How to Parallelize?

Okay, so let’s say you have a nice serial (single-CPU) code that does an N-body calculation.

How are you going to parallelize it?

You could:

- have a server feed particles to processes;
- have a server feed interactions to processes;
- have each process decide on its own subset of the particles, and then share around the forces;
- have each process decide its own subset of the interactions, and then share around the forces.

Do You Need a Master?

Let's say that you have *N* bodies, and therefore you have ½ *N* (*N* - 1) interactions (every particle interacts with all of the others, but you don't need to calculate both A → B and B → A).

Do you need a server?

Well, can each processor determine, on its own, either (a) which of the bodies to process, or (b) which of the interactions to process?

If the answer is yes, then you don't need a server.

Parallelize How?

Suppose you have N_p processors.

Should you parallelize:

- by assigning a subset of N / N_p of the bodies to each processor, OR
- by assigning a subset of ½ N (N - 1) / N_p of the interactions to each processor?

Data vs. Task Parallelism

- ***Data Parallelism*** means parallelizing by giving a subset of the data to each process, and then each process performs the same tasks on the different subsets of data.
- ***Task Parallelism*** means parallelizing by giving a subset of the tasks to each process, and then each process performs a different subset of tasks on the same data.

Data Parallelism for N-Body?

If you parallelize an N-body code **by data**, then each processor gets N / N_p pieces of data.

For example, if you have 8 bodies and 2 processors, then:

- P_0 gets the first 4 bodies;
- P_1 gets the second 4 bodies.

But, every piece of data (that is, every body) has to interact with every other piece of data, to calculate the forces.

So, every processor will have to send all of its data to all of the other processors, for every single interaction that it calculates.

That’s a lot of communication!

Task Parallelism for N-body?

If you parallelize an N-body code **by task**, then each processor gets all of the pieces of data that describe the particles (for example, positions, velocities, masses).

Then, each processor can calculate its subset of the interaction forces on its own, without talking to any of the other processors.

But, at the end of the force calculations, everyone has to share all of the forces that have been calculated, so that each particle ends up with the total force that acts on it (***global reduction)***.

MPI_Reduce

Here's the syntax for **MPI_Reduce**:

```
MPI_Reduce(sendbuffer, recvbuffer,
  count, datatype, operation,
  root, communicator);
```

For example, to do a sum over all of the particle forces:

```
MPI_Reduce(
  local_particle_force_sum,
  global_particle_force_sum,
  number_of_particles,
  MPI_DOUBLE, MPI_SUM,
  server_process, MPI_COMM_WORLD);
```

Sharing the Result

In the N-body case, we don't want just one processor to know the result of the sum, we want every processor to know.

So, we could do a reduce followed immediately by a broadcast.

But, MPI gives us a routine that packages all of that for us: **`MPI_Allreduce`**.

`MPI_Allreduce` is just like **`MPI_Reduce`** except that every process gets the result (so we drop the **`server_process`** argument).

MPI_Allreduce

Here’s the syntax for **MPI_Allreduce**:

```
MPI_Allreduce(sendbuffer,
  recvbuffer, count, datatype,
  operation, communicator);
```

For example, to do a sum over all of the particle forces:

```
MPI_Allreduce(
  local_particle_force_sum,
  global_particle_force_sum,
  number_of_particles,
  MPI_DOUBLE, MPI_SUM,
  MPI_COMM_WORLD);
```

Collective Communications

A ***collective communication*** is a communication that is shared among many processes, not just a sender and a receiver.

`MPI_Reduce` and **`MPI_Allreduce`** are collective communications.

Others include: broadcast, gather/scatter, all-to-all.

Collectives Are Expensive

Collective communications are very expensive relative to point-to-point communications, because so much more communication has to happen.

But, they can be much cheaper than doing zillions of point-to-point communications, if that’s the alternative.

Transport:
Data Parallelism

[2]

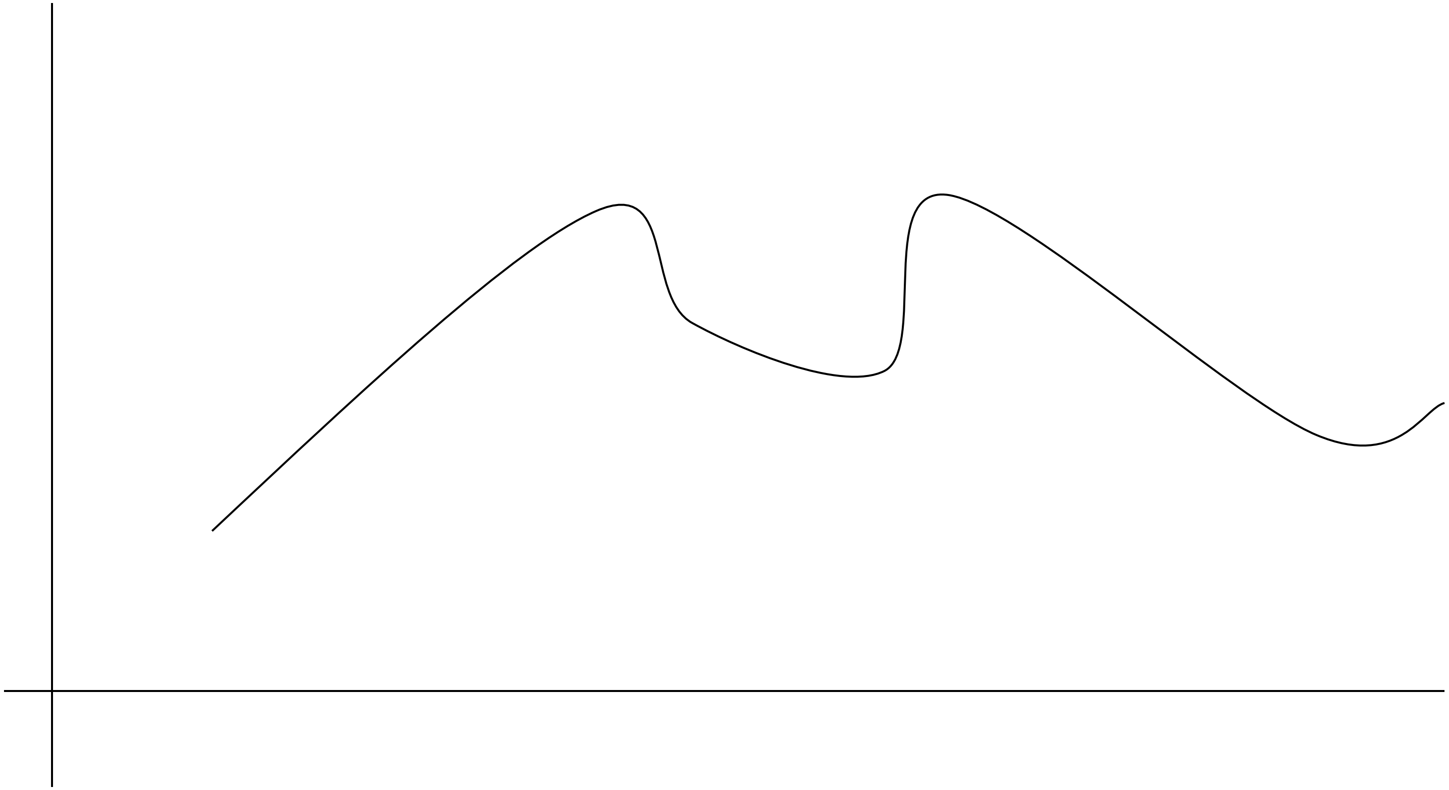
What is a Simulation?

All physical science ultimately is expressed as calculus (for example, 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!

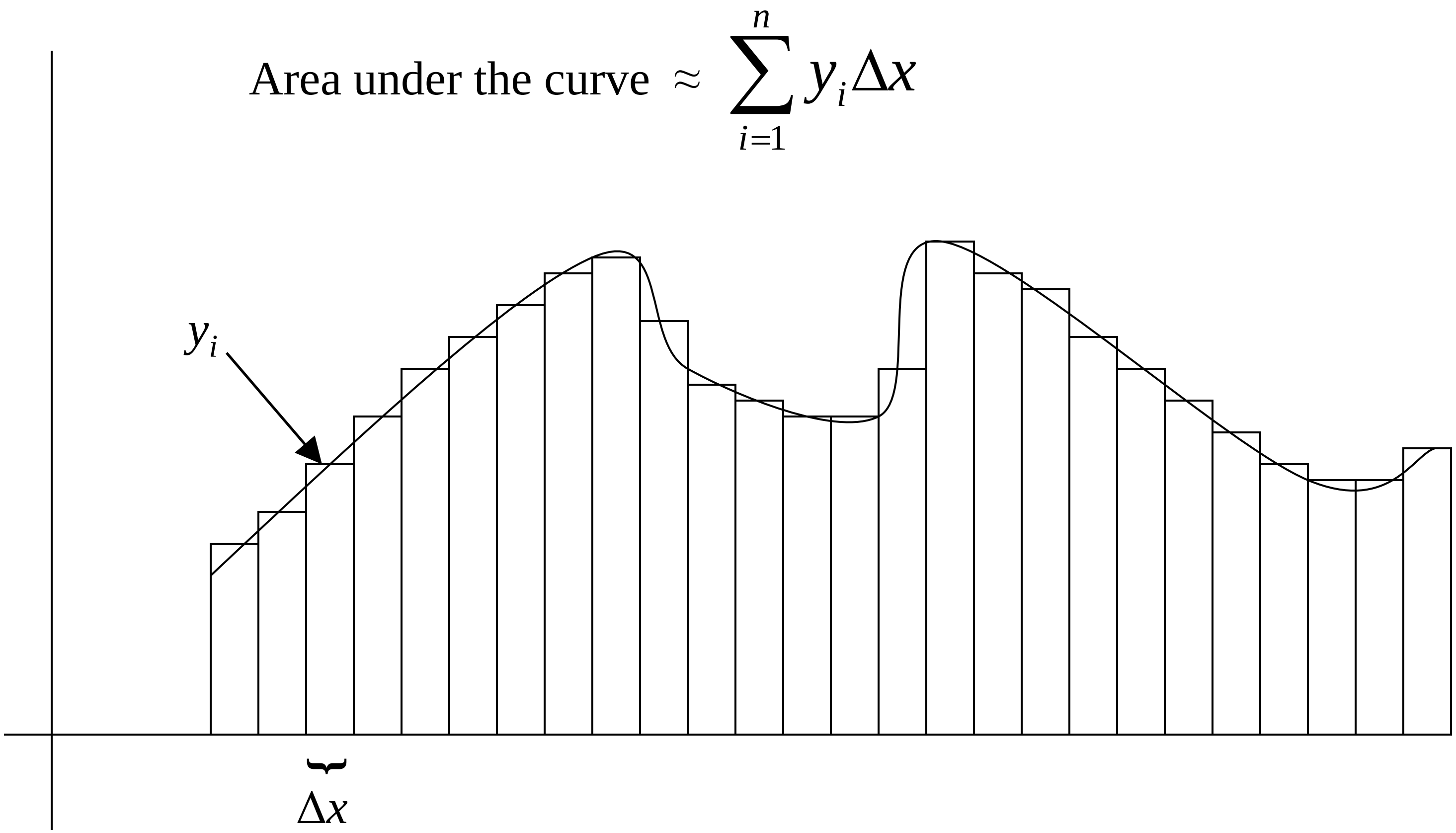


How can I get the area under this curve?

A Riemann Sum

[3]

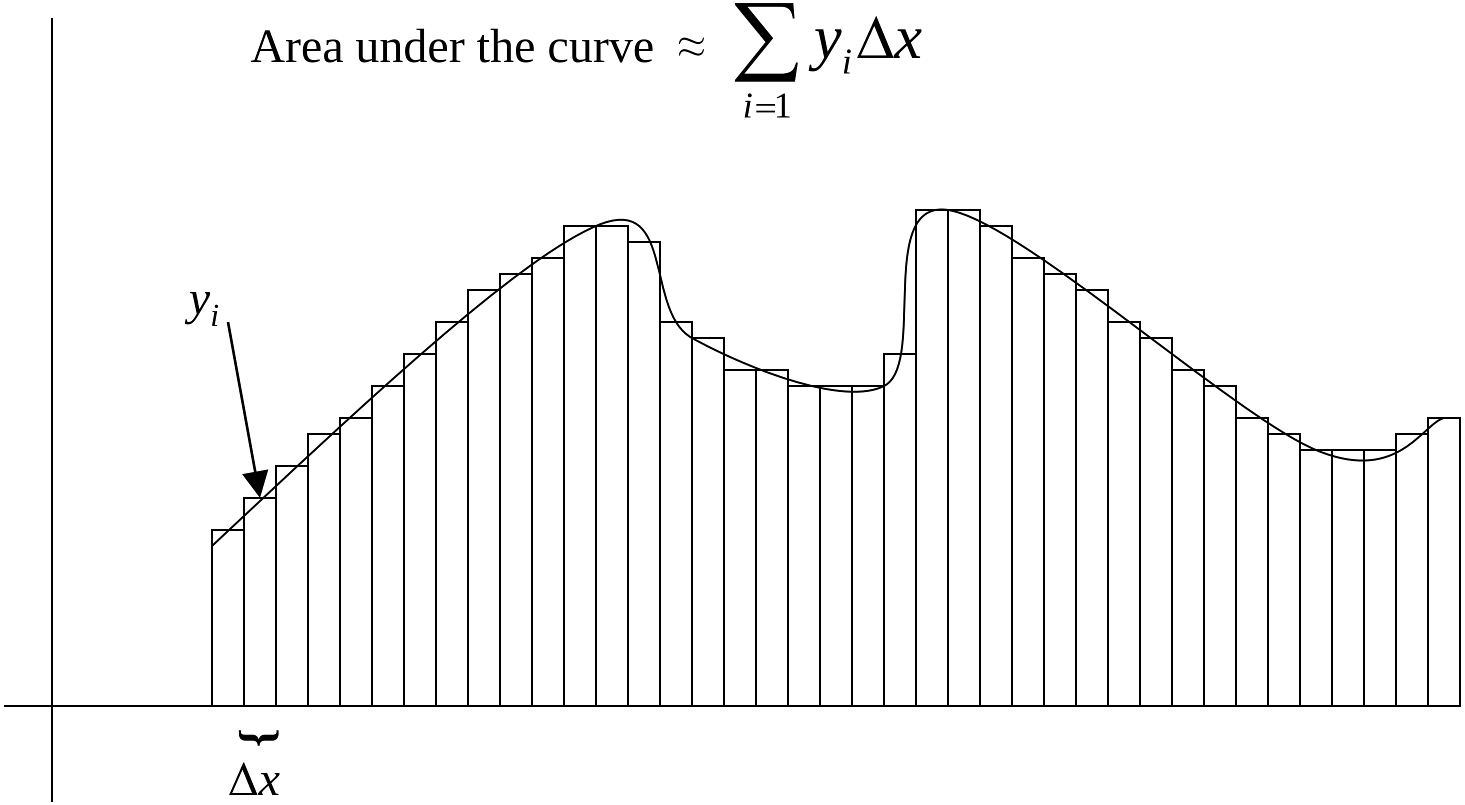
A Riemann Sum


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

Ceci n’est pas un area under the curve: it’s **approximate**!

A Better Riemann Sum

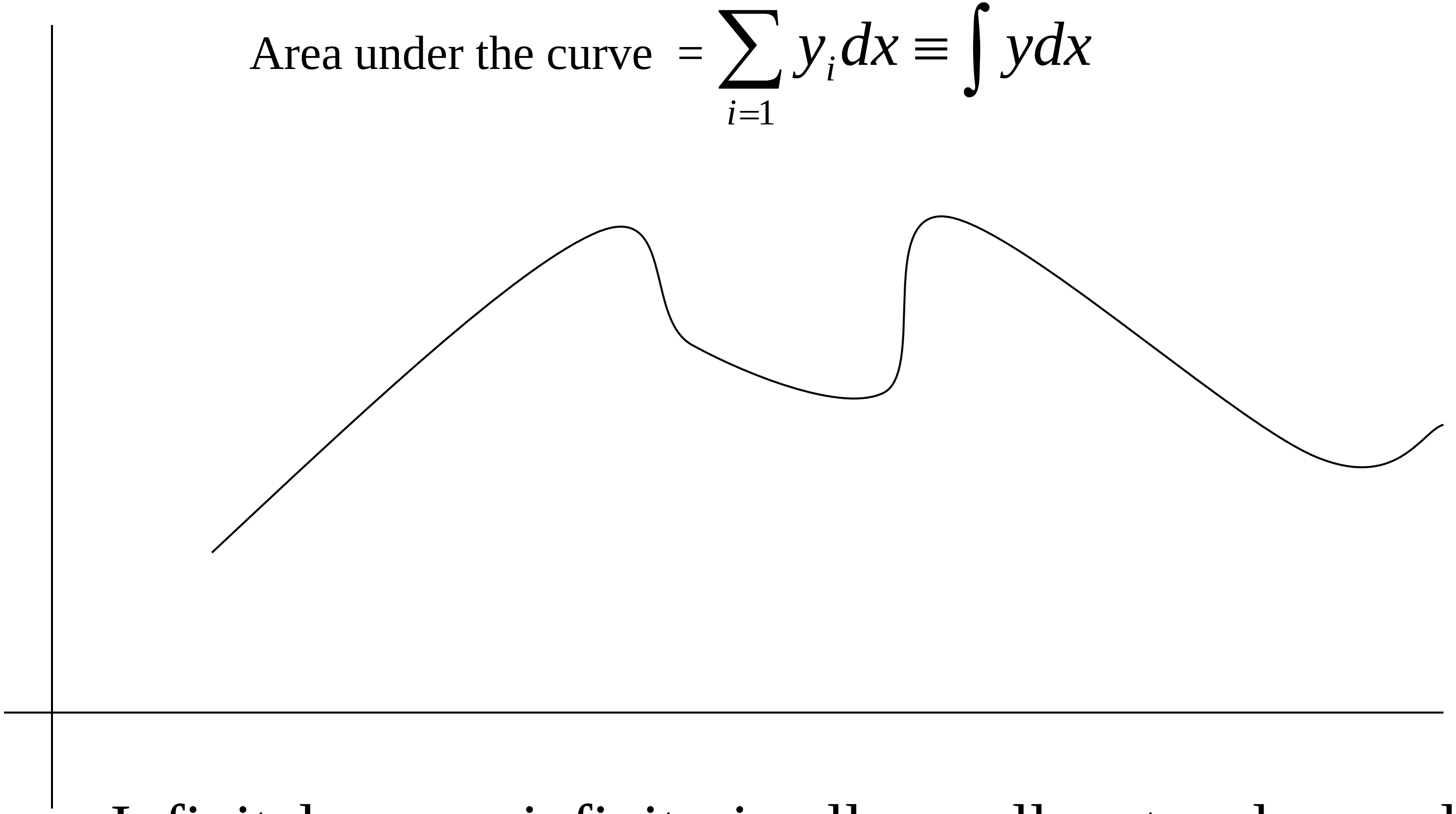
$$\text{Area under the curve} \approx \sum_{i=1}^{n} y_i \Delta x$$



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

The Best Riemann Sum

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



Infinitely many infinitesimally small rectangles produce the area.

The Best Riemann Sum

In the limit, infinitely many infinitesimally small rectangles produce the correct area.

Differential Equations

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

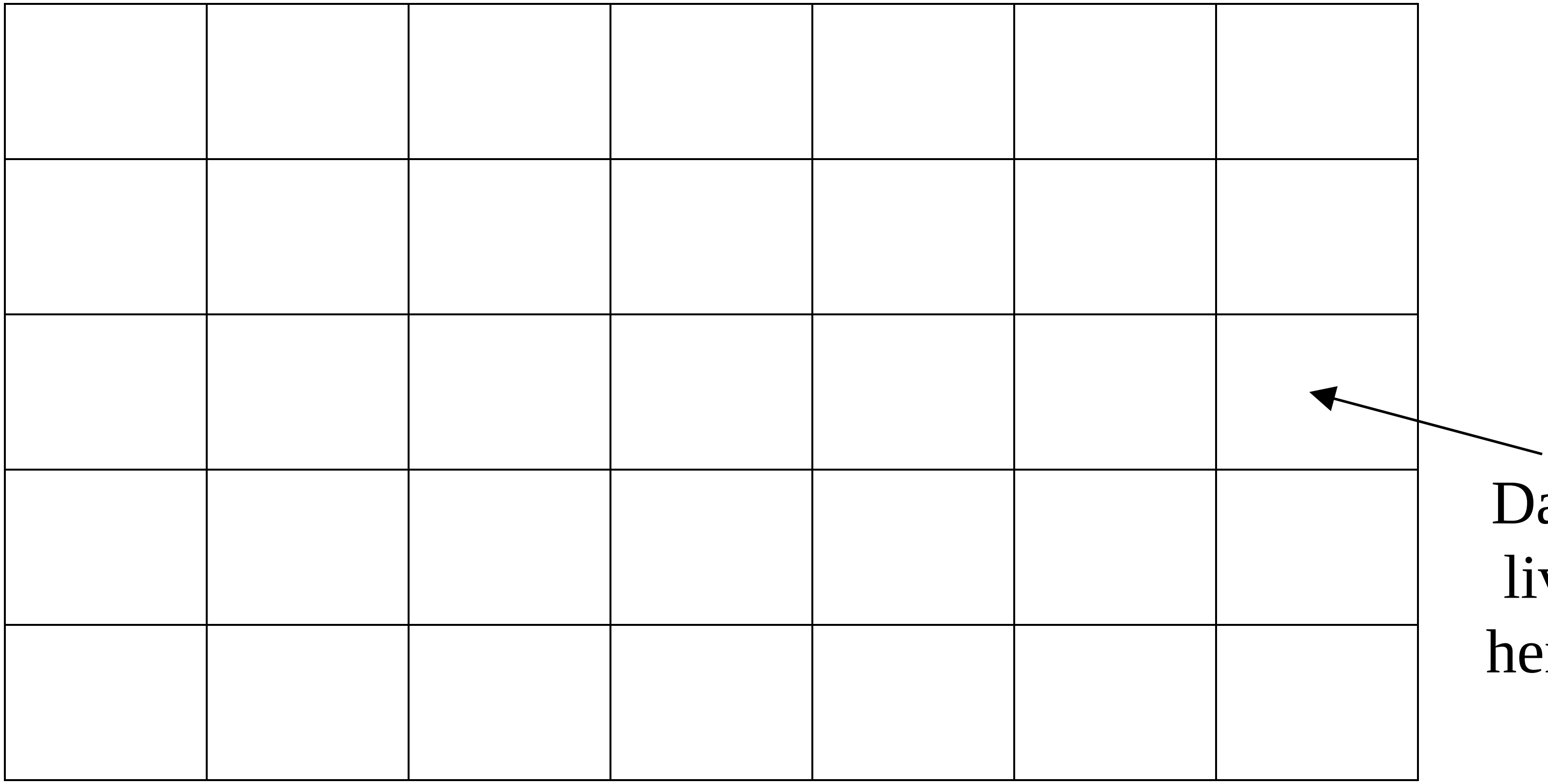
Most physics is best expressed as differential equations.

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.

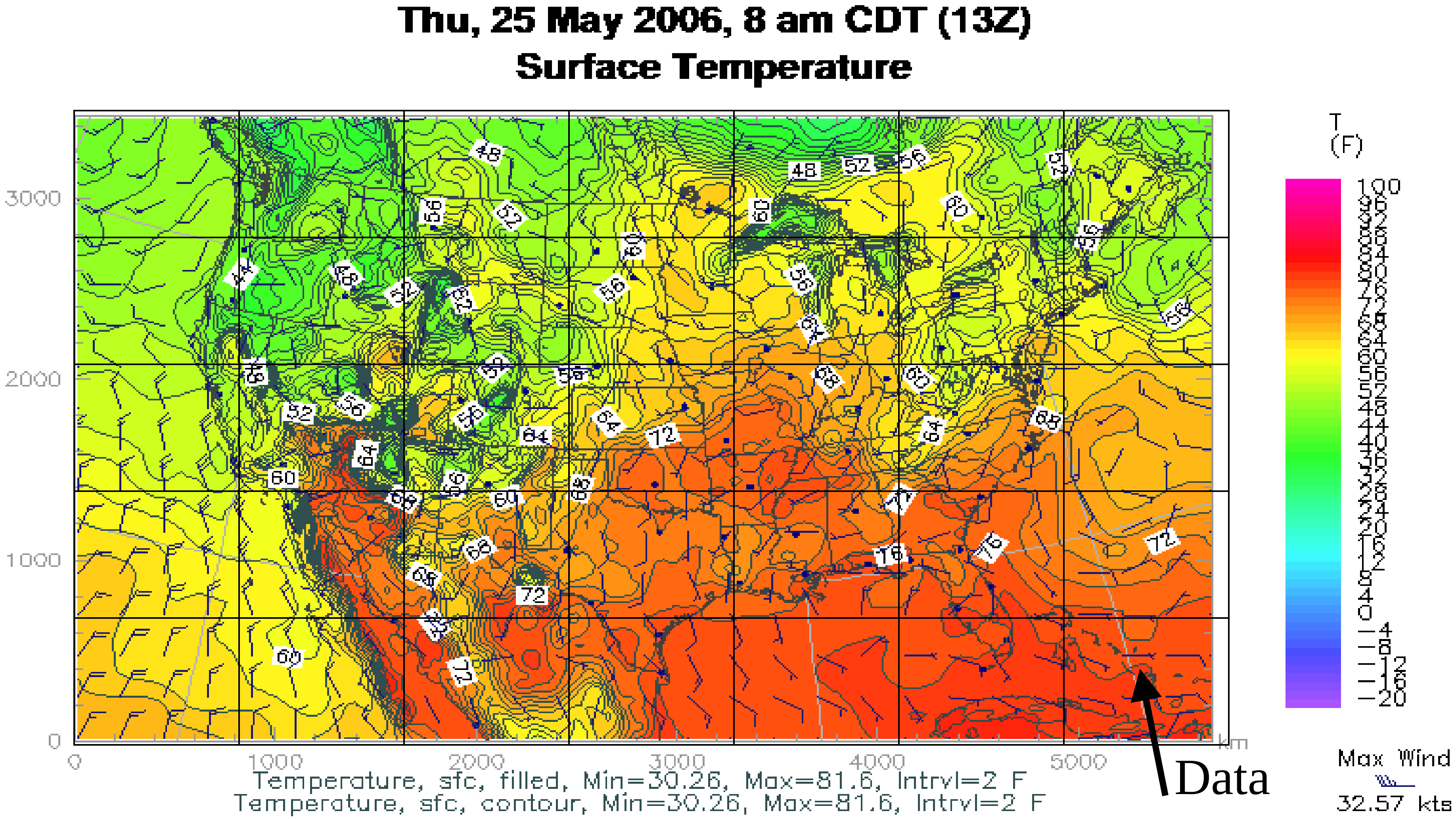
Solution: approximate!

A Discrete Mesh of Data



Data live here!

A Discrete Mesh of Data



Thu, 25 May 2006, 8 am CDT (13Z)

Surface Temperature

Temperature, sfc, filled, Min=30.26, Max=81.6, Intrvl=2 F

Temperature, sfc, contour, Min=30.26, Max=81.6, Intrvl=2 F

Data live here!

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 Δ*x* (has finite width).

Navier-Stokes Equation

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

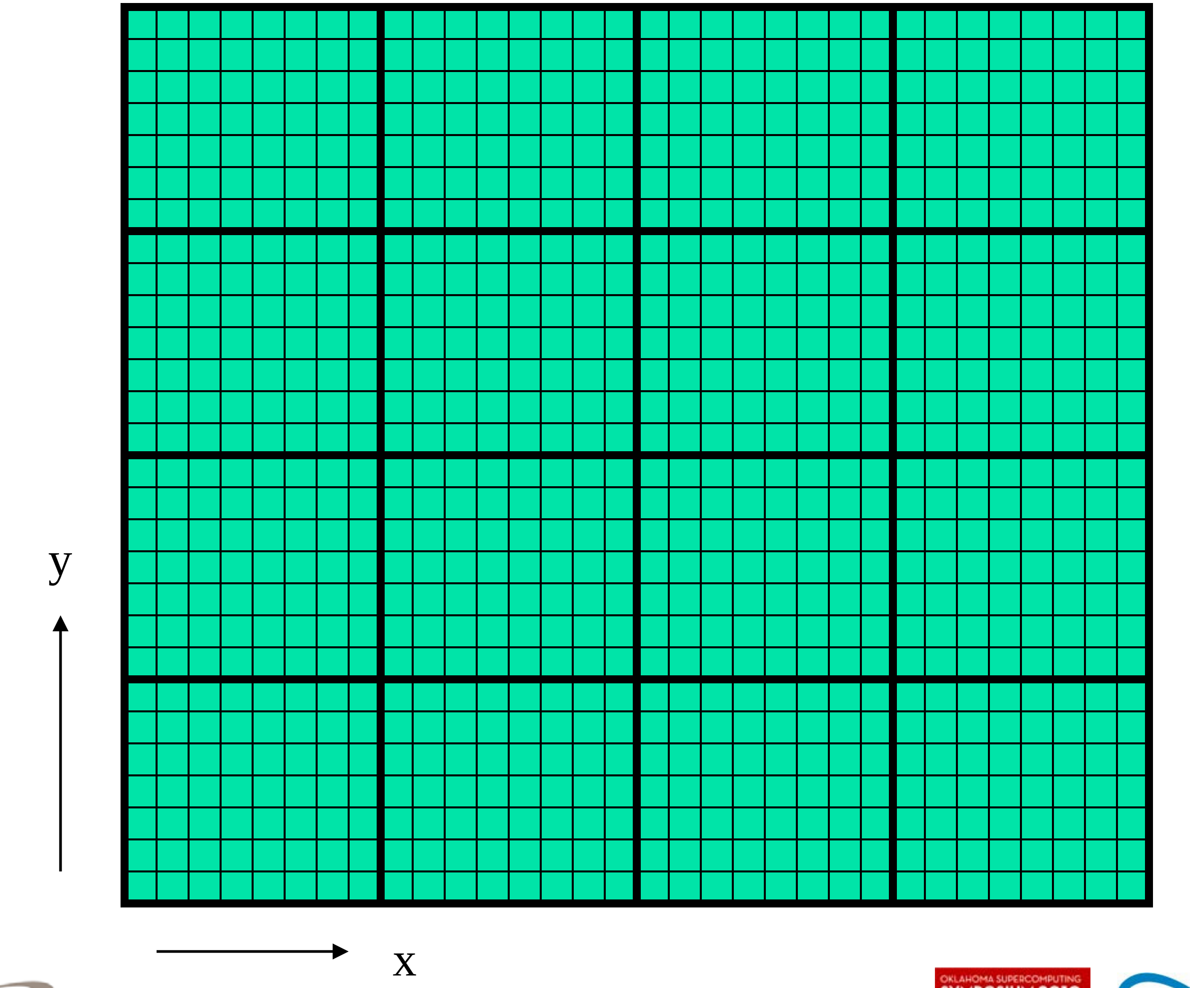
Differential Equation

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

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:

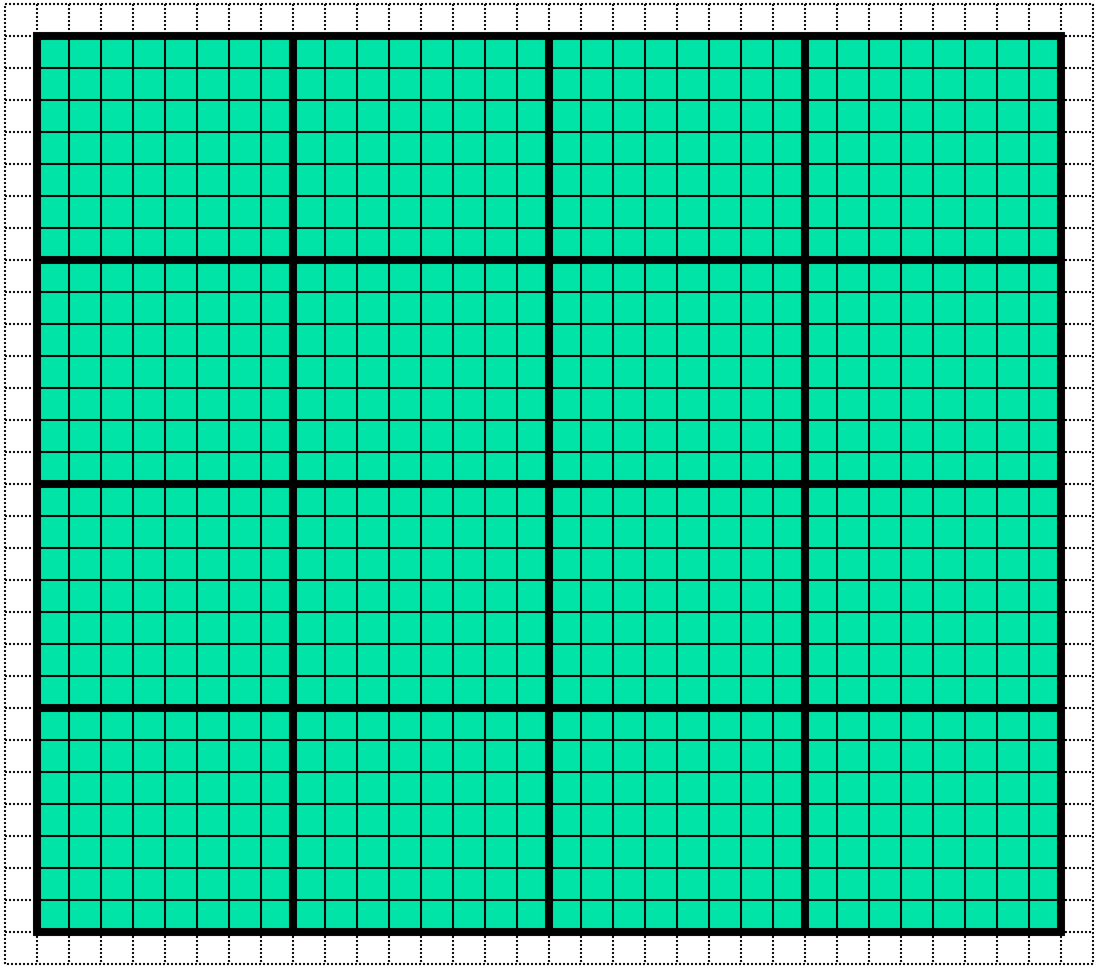
$$unew_{\mathrm{i,j}} = \\ \mathrm{F}(\Delta t, uold_{\mathrm{i,j}}, \\ uold_{\mathrm{i-1,j}}, uold_{\mathrm{i+1,j}}, uold_{\mathrm{i,j-1}}, uold_{\mathrm{i,j+1}})$$

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

In other words, you calculate the new value of $u_{\mathrm{i,j}}$, 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.

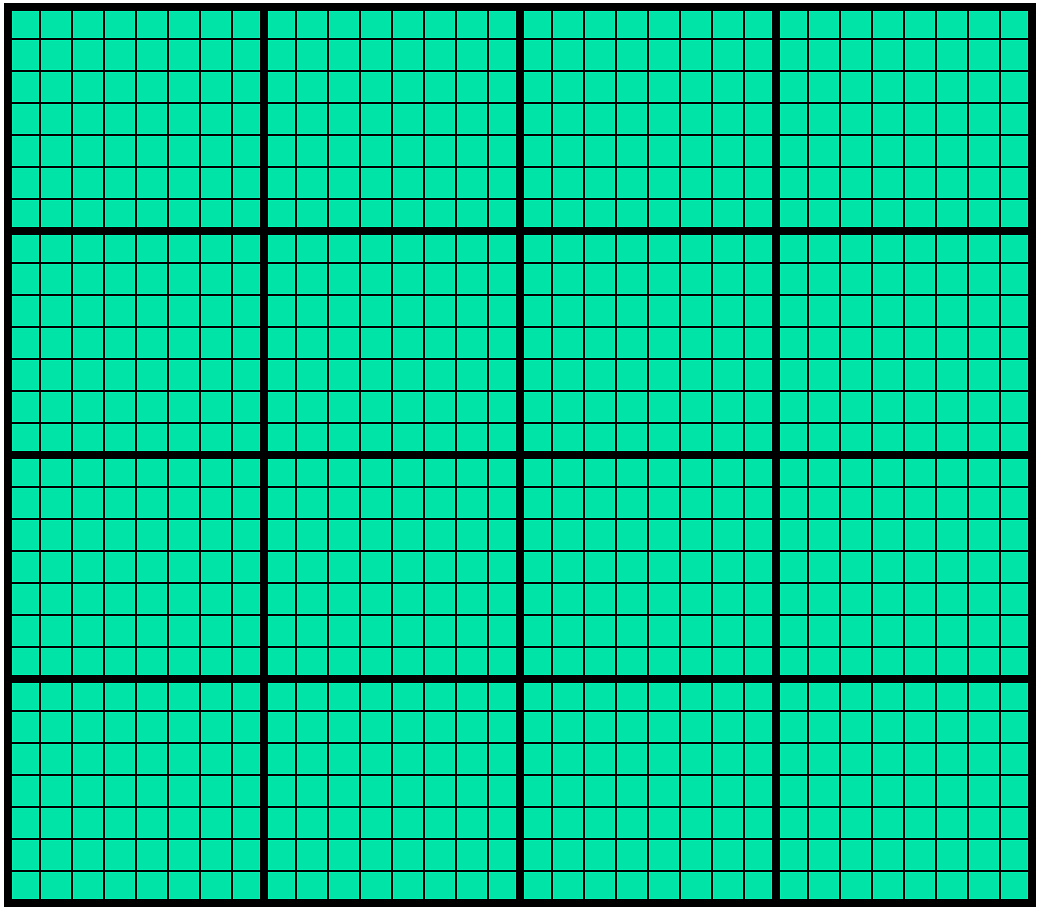
Ghost Boundary Zones in MPI

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);
```

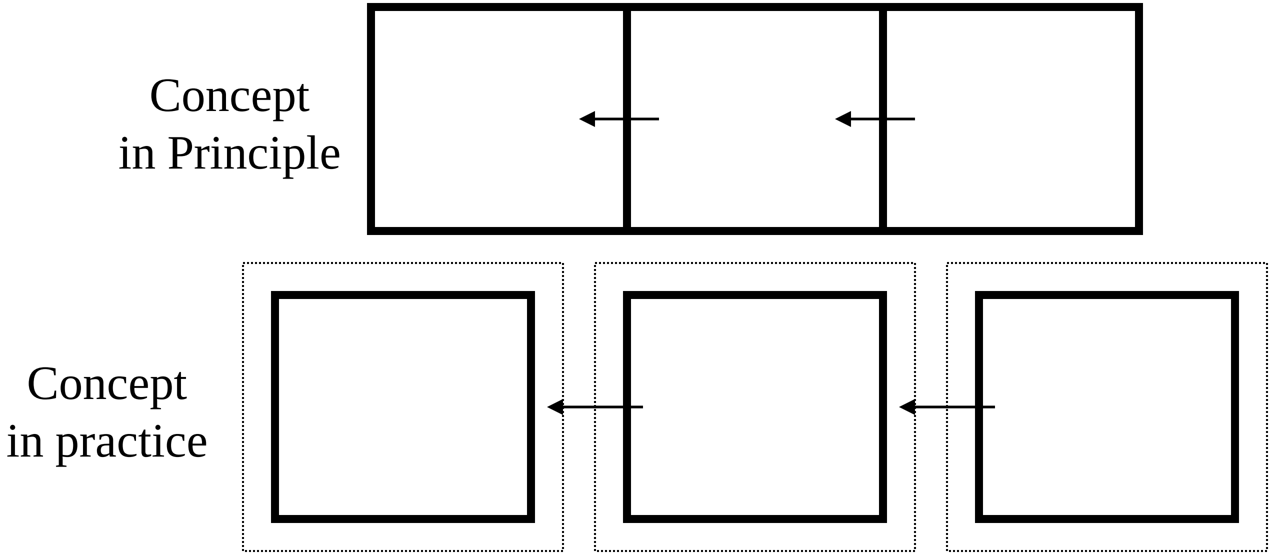
This call sends to `west_neighbor_process` the data in `westward_send_buffer`, and at the same time receives from `east_neighbor_process` a bunch of data that end up in `westward_recv_buffer`.

Why `MPI_Sendrecv`?

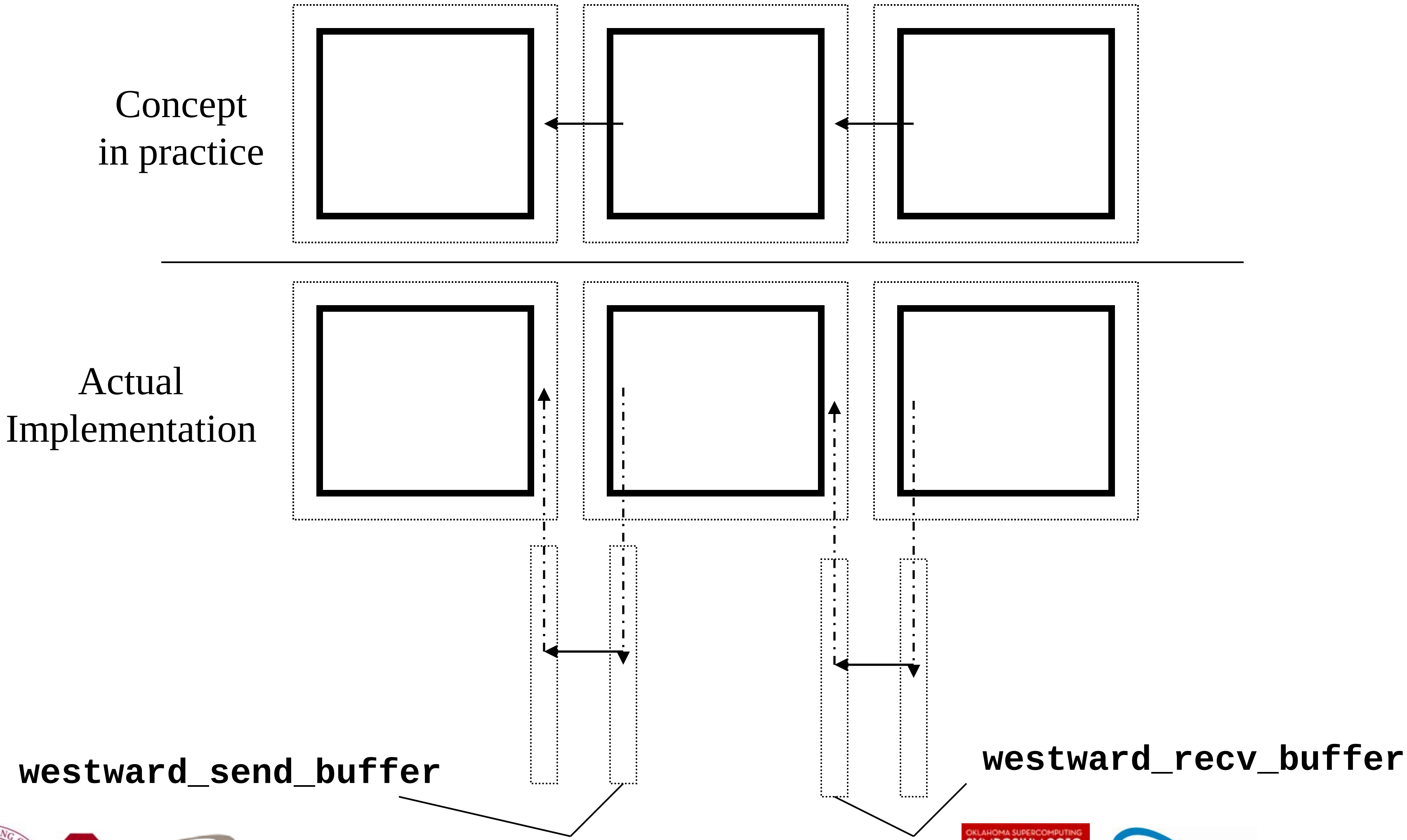
The advantage of `MPI_Sendrecv` is that it allows us the luxury of no longer having to worry about who should send when and who should receive when.

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



What About Edges and Corners?

If your numerical method involves faces, edges and/or corners, don’t despair.

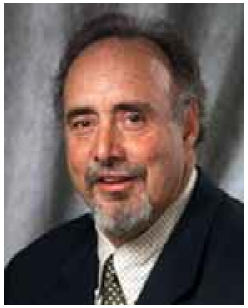
It turns out that, if you do the following, you’ll handle those correctly:

- When you send, send the entire ghost boundary’s worth, including the ghost boundary of the part you’re sending.
- Do in this order:
  - all east-west;
  - all north-south;
  - all up-down.
- At the end, everything will be in the correct place.

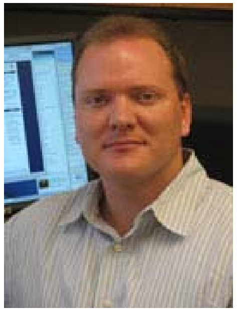
OK Supercomputing Symposium 2010

2003 Keynote:
Peter Freeman
NSF
Computer & Information Science & Engineering
Assistant Director

2004 Keynote:
Sangtae Kim
NSF Shared Cyberinfrastructure
Division Director


2005 Keynote:
Walt Brooks
NASA Advanced Supercomputing
Division Director

2006 Keynote:
Dan Atkins
Head of NSF's
Office of
Cyberinfrastructure


2007 Keynote:
Jay Boisseau
Director
Texas Advanced
Computing Center
U. Texas Austin

2008 Keynote:
José Munoz
Deputy Office
Director/ Senior
Scientific Advisor
NSF Office of
Cyberinfrastructure

2009 Keynote:
Douglass Post
Chief Scientist
US Dept of Defense
HPC Modernization
Program

2010 Keynote
Horst Simon, Director
National Energy Research Scientific Computing Center

FREE! Wed Oct 6 2010 @ OU

http://symposium2010.oscer.ou.edu/

Thanks for your attention!

Questions?

www.oscer.ou.edu

References

[1] http://en.wikipedia.org/wiki/Monte_carlo_simulation

[2] http://en.wikipedia.org/wiki/N-body_problem

[3] http://lostbiro.com/blog/wp-content/uploads/2007/10/Magritte-Pipe.jpg