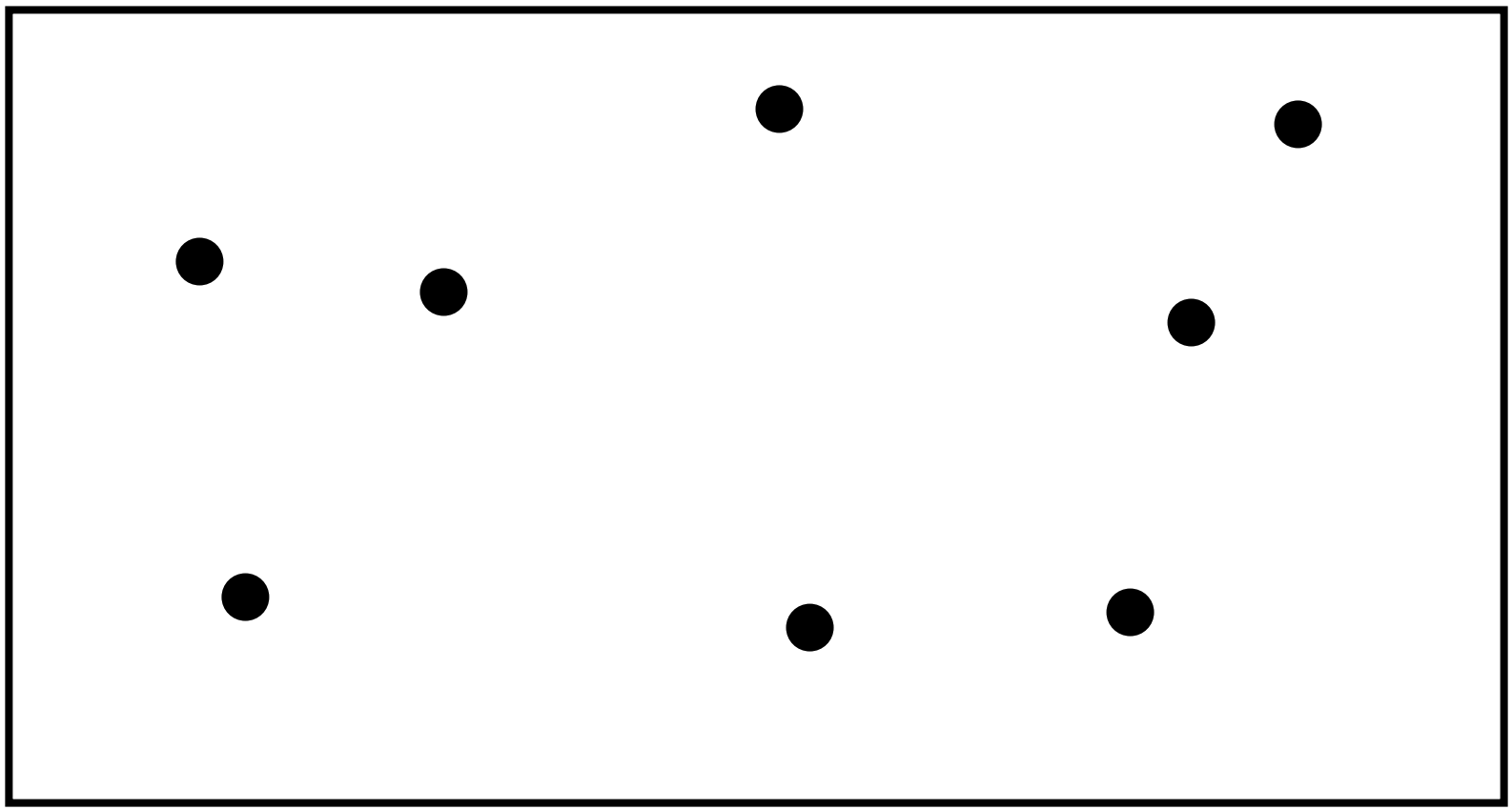
Parallel Programming & Cluster Computing

N-Body Simulation and Collective Communications

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

N Bodies



N-Body Problems

An ***N-body problem*** is a problem involving *N* "bodies" – that is, particles (e.g., 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.

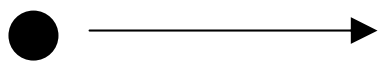
1-Body Problem

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 $t_0+\Delta t$:

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

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

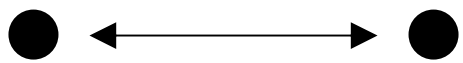


2-Body Problem

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.



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

For N of 3 or more, no one knows how to solve the equations to get a closed form solution.

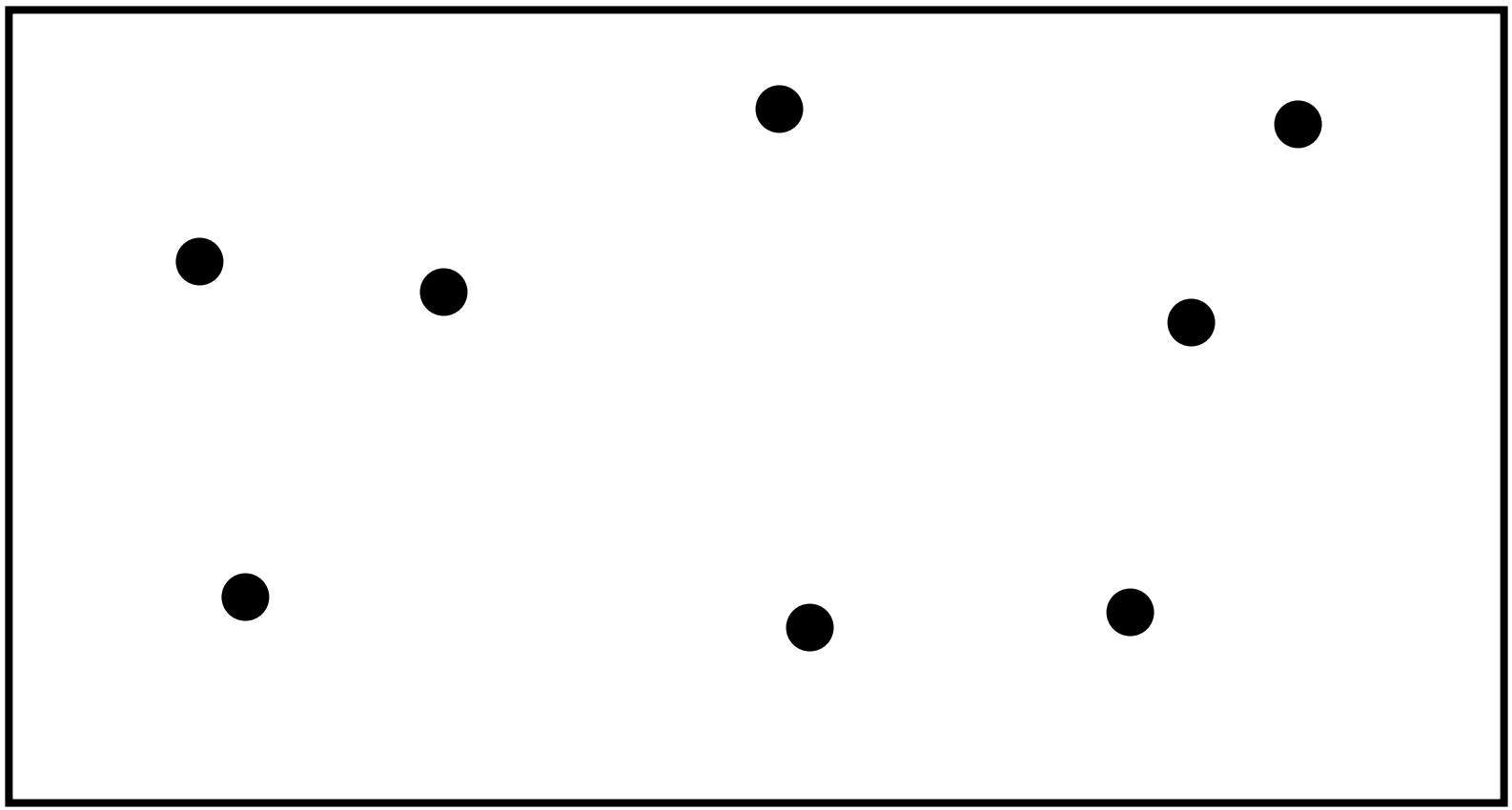
So, numerical simulation is pretty much the only way to study groups of 3 or more bodies.

Popular applications of N-body codes include:

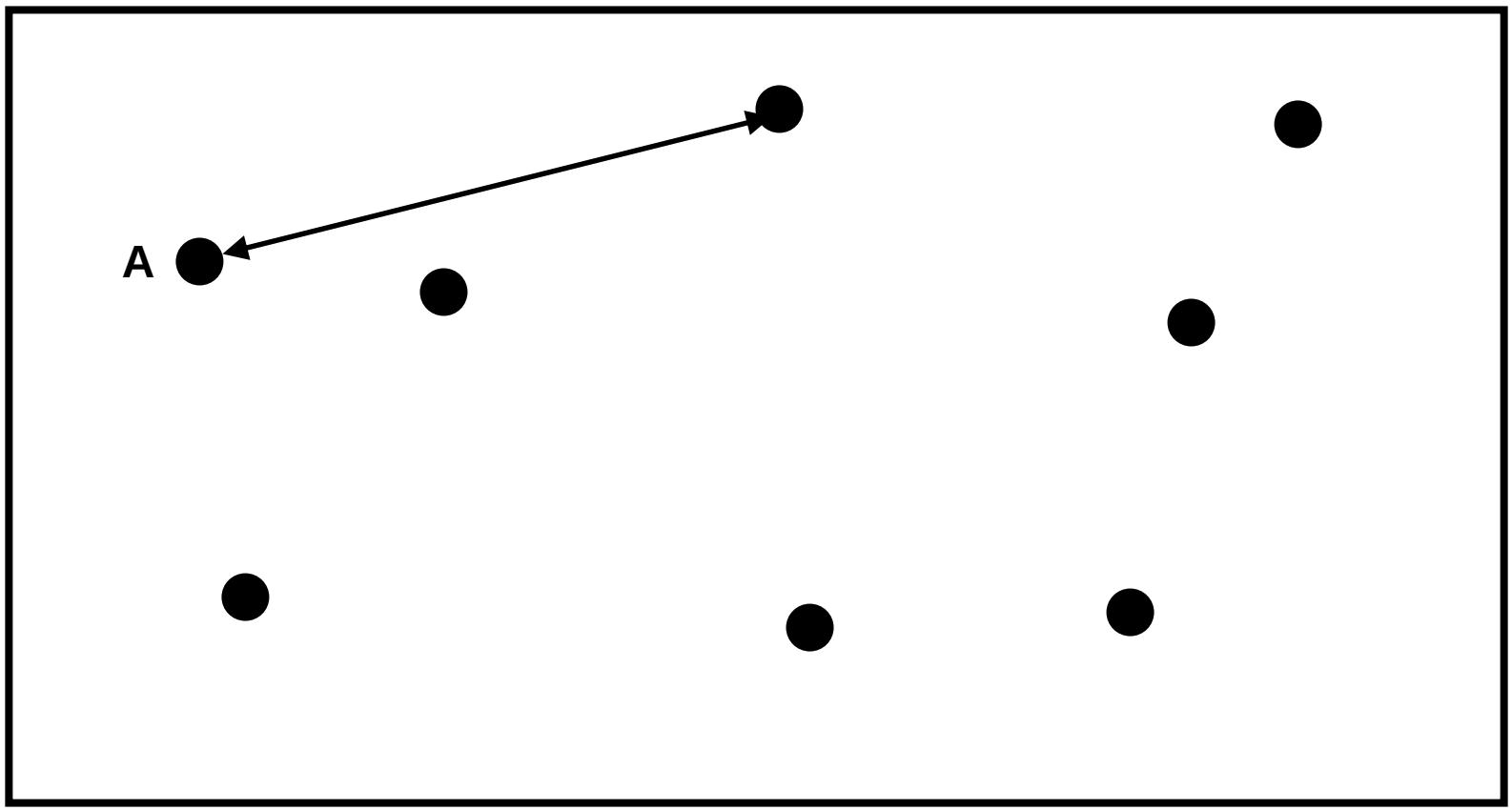
- astronomy (e.g., galaxy formation, cosmology);
- chemistry (e.g., protein folding, molecular dynamics).

Note that, for N bodies, there are on the order of N^2 forces, denoted $\mathbf{O}(N^2)$.

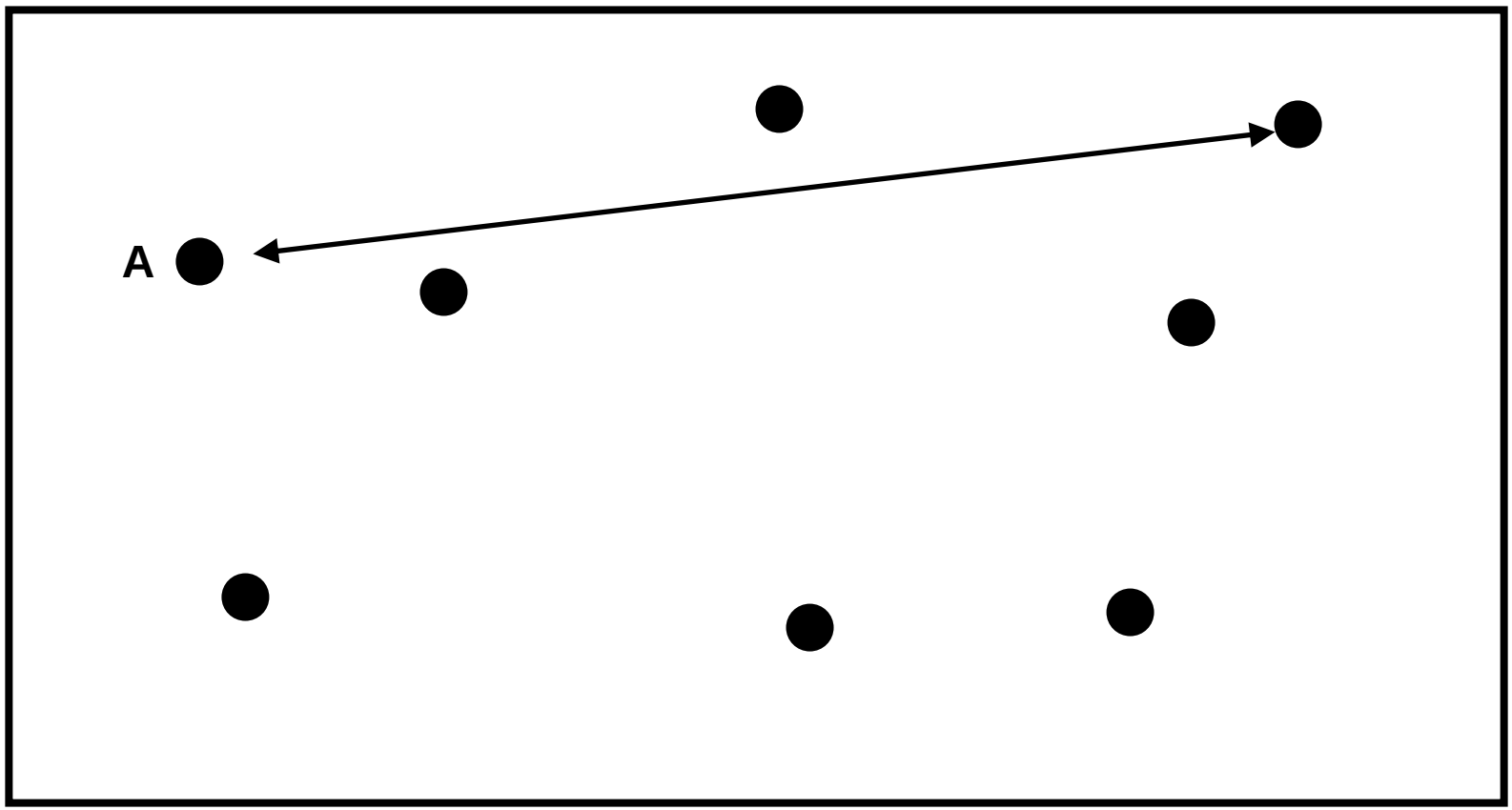
N Bodies



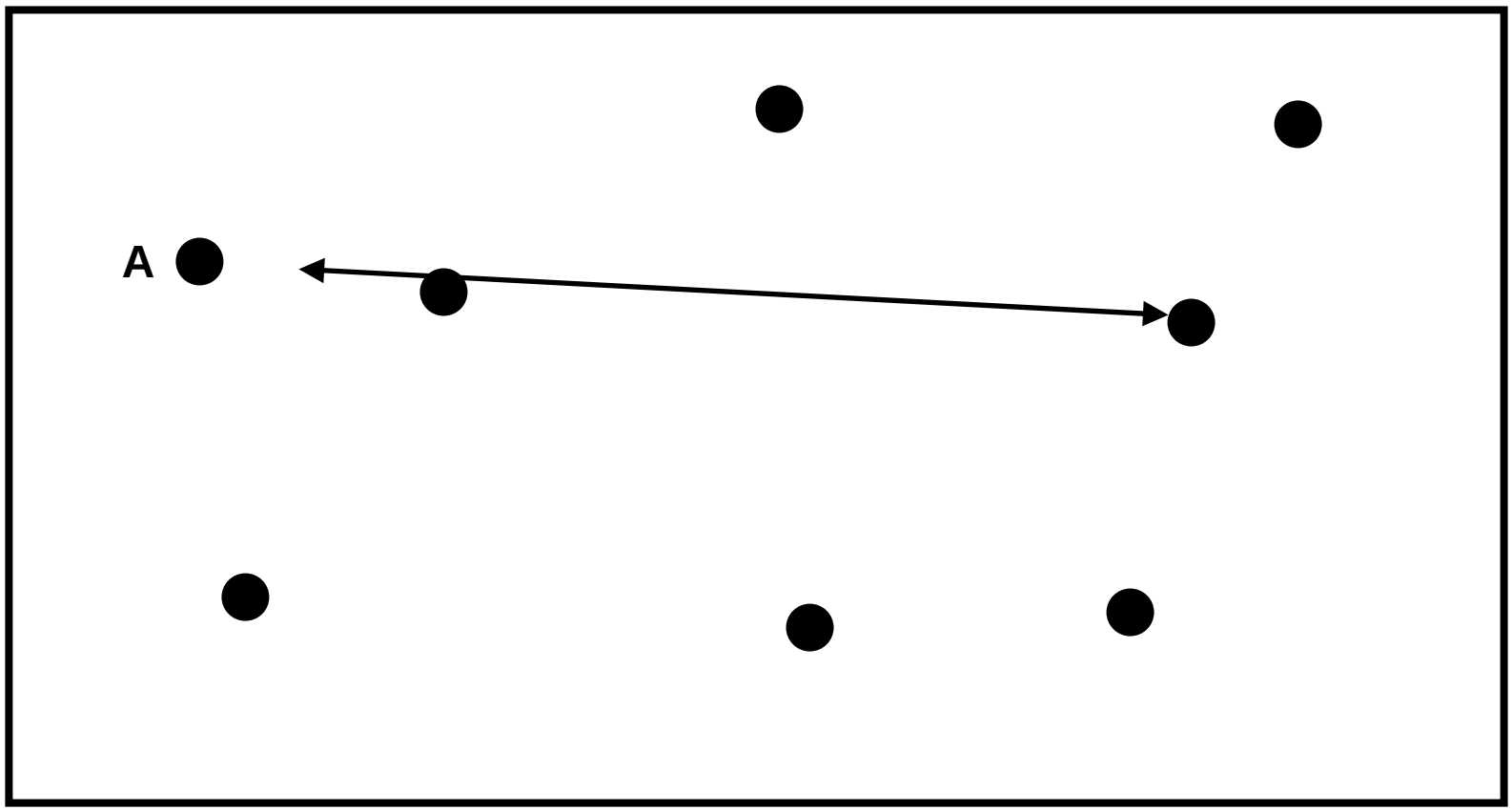
Force #1



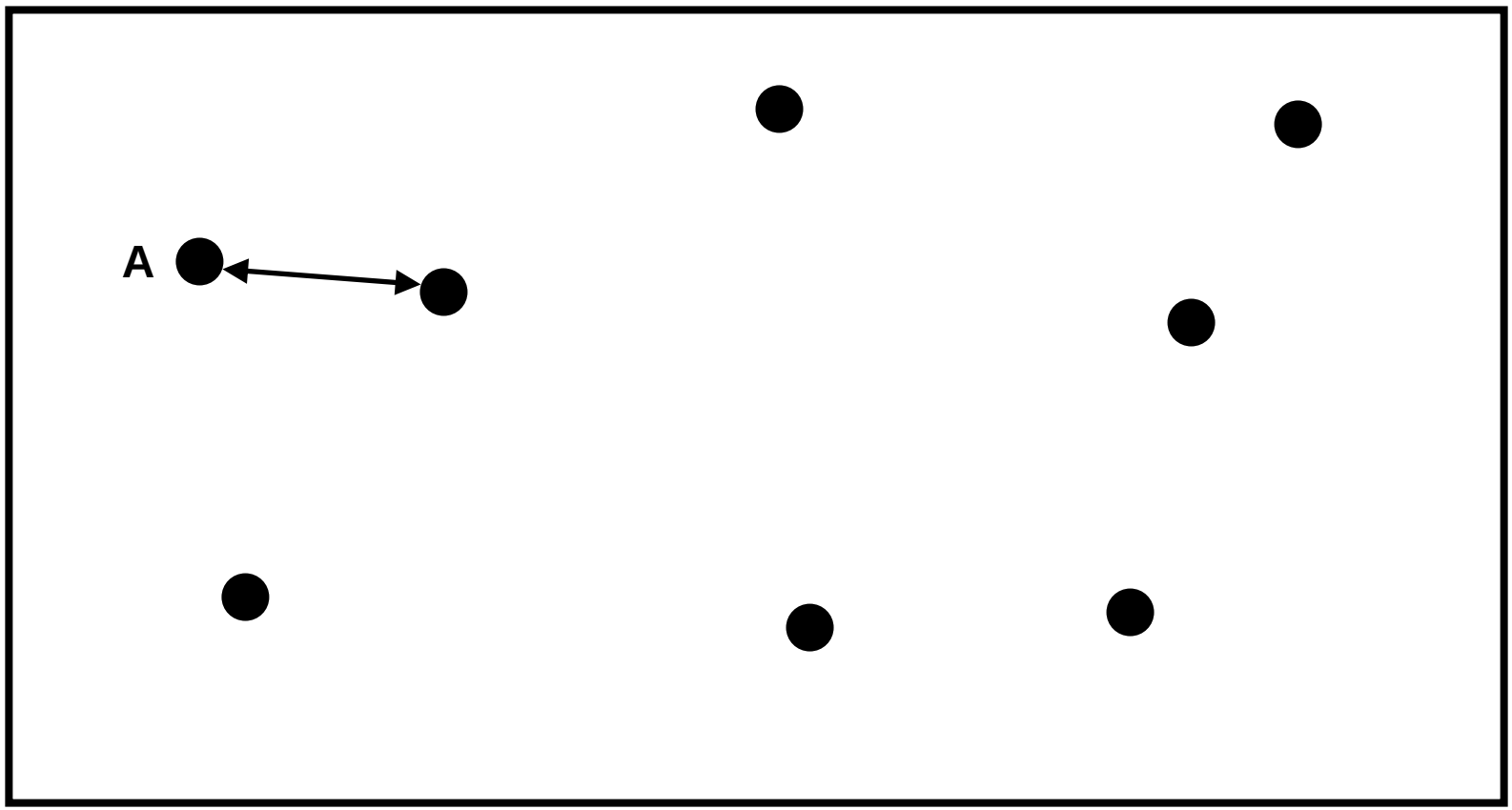
Force #2



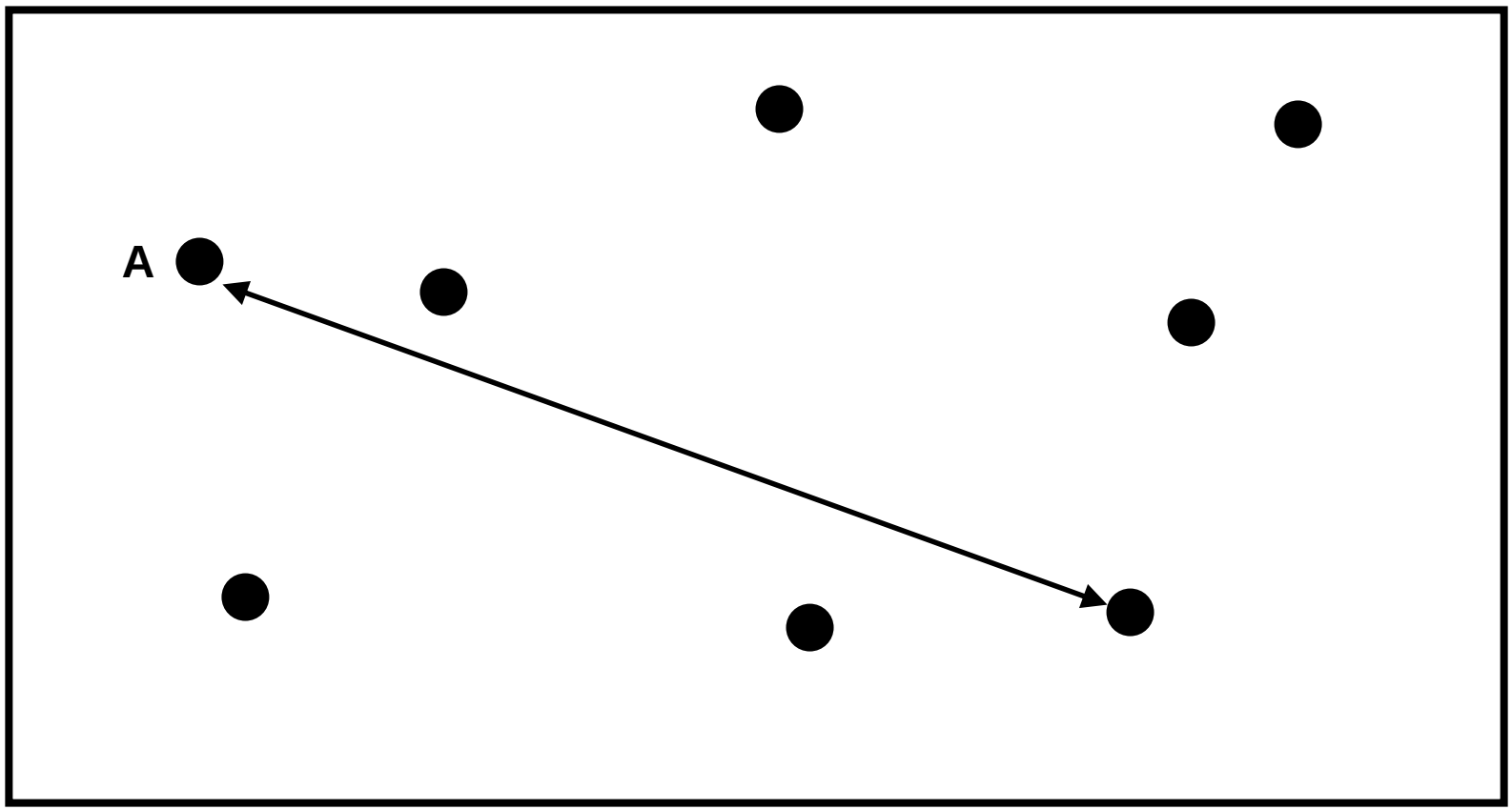
Force #3



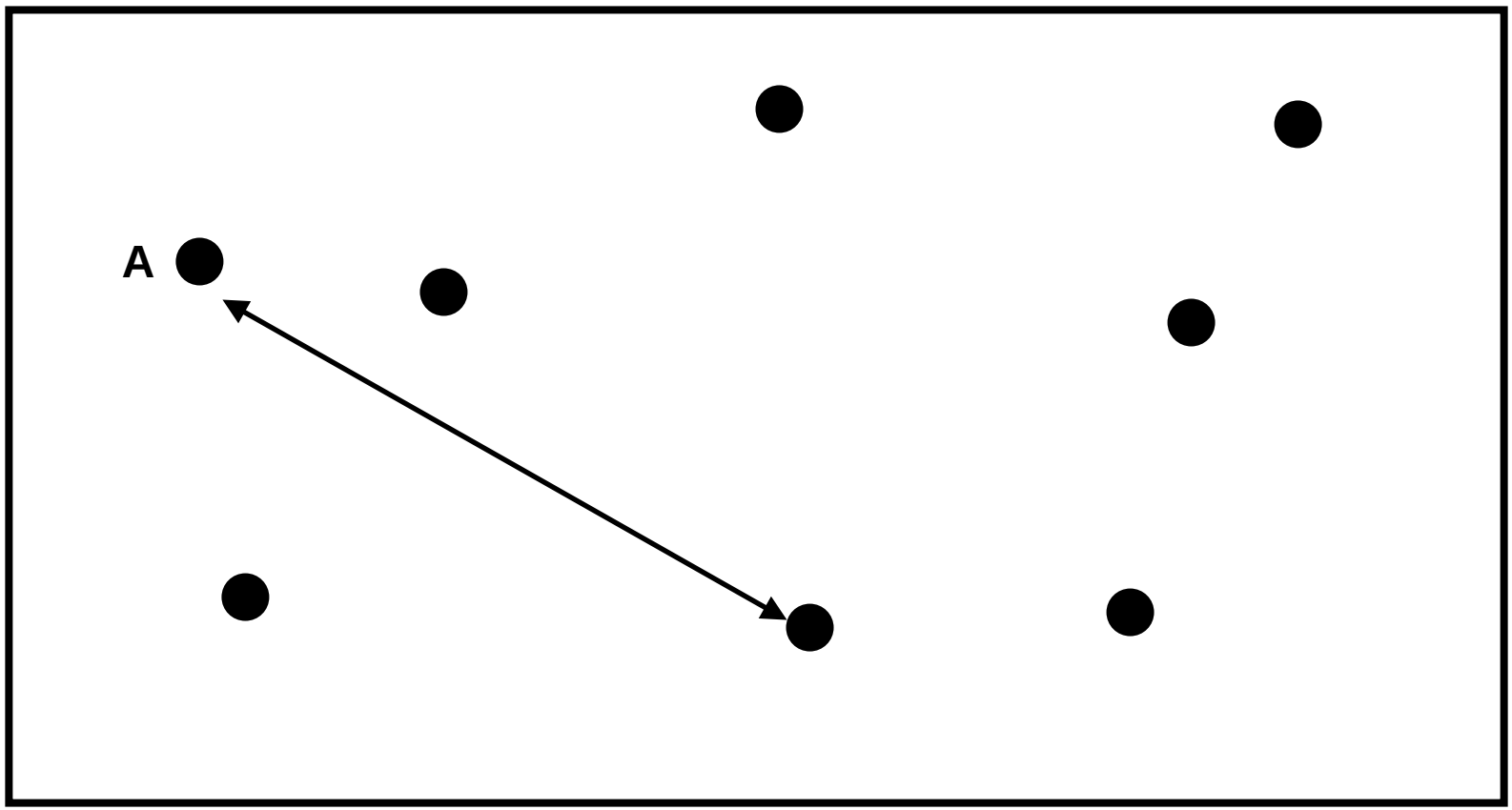
Force #4



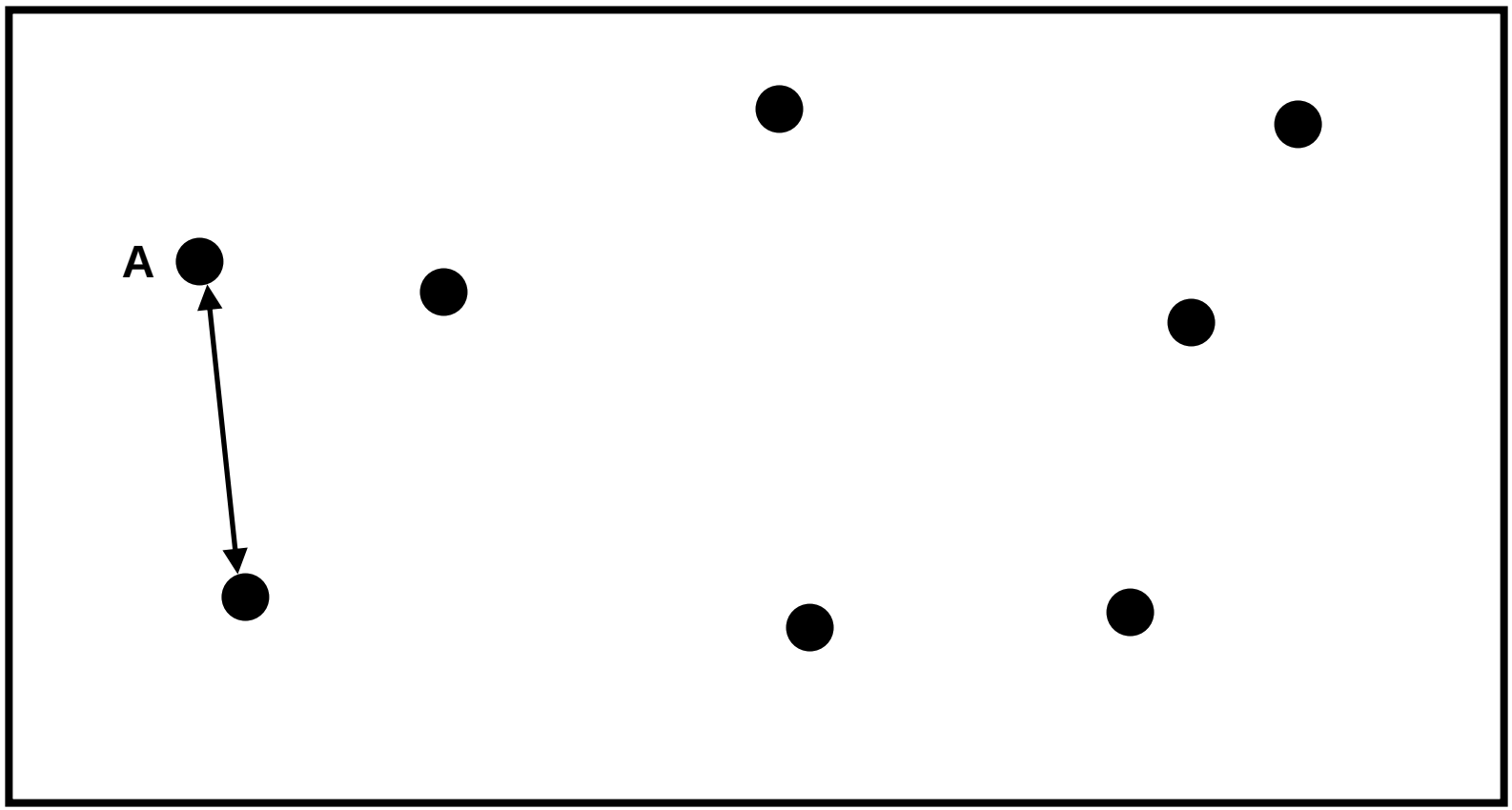
Force #5



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 $N \bullet (N - 1)$ forces in total.

You can also think of this as $(N \bullet (N - 1)) / 2$ 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.

$$C_1 \cdot N + C_2$$

Big-O: Dropping the Low Term

$$C_1 \cdot N + C_2$$

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:

$$C_1 \cdot N$$

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 $N \bullet (N - 1)$ forces total.

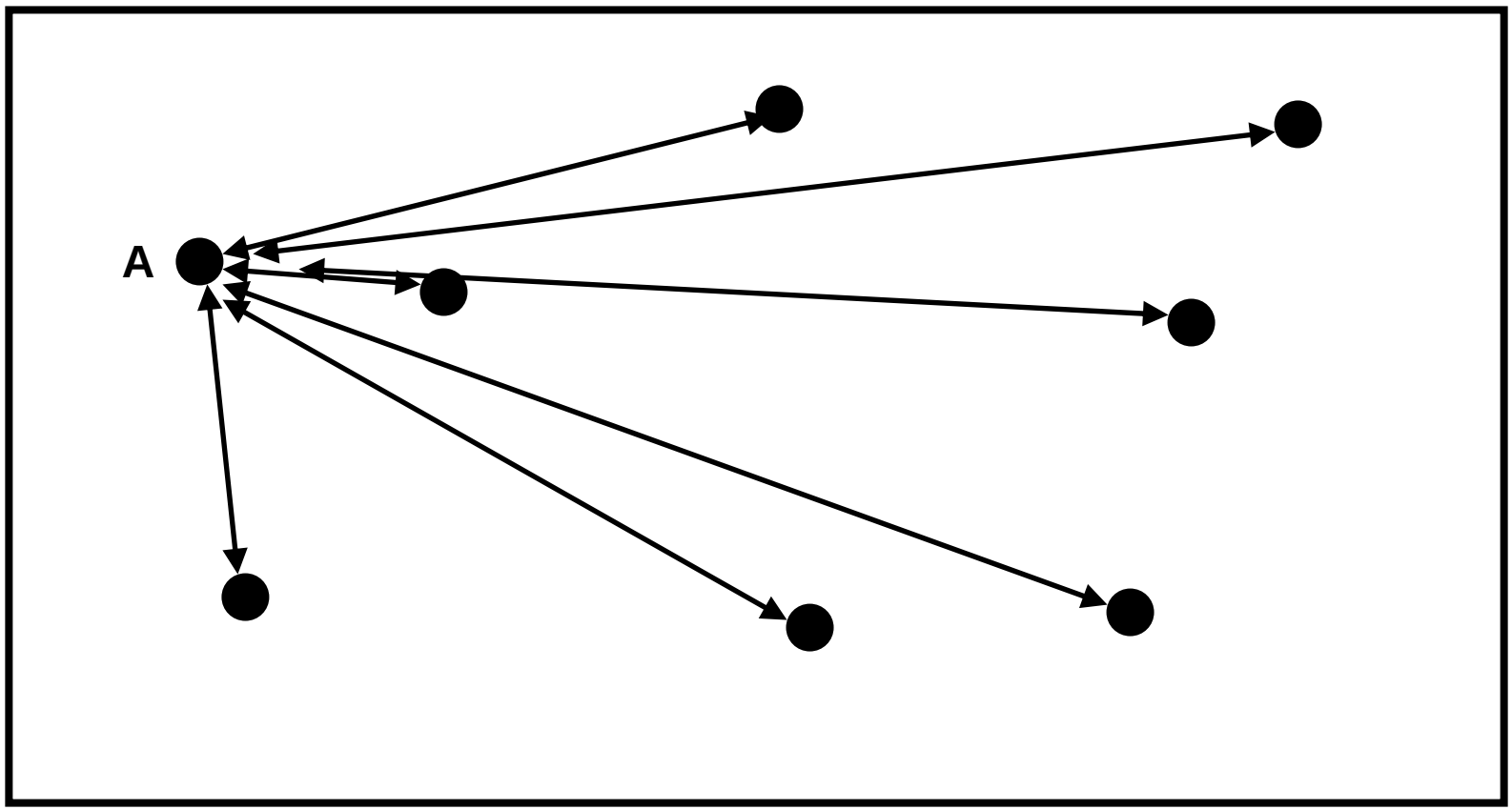
In Big-**O** notation, that's $\mathbf{O}(N^2)$ forces.

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, ***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 (i.e., 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 (e.g., positions, velocities).

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.

To Learn More

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

Thanks for your attention!

Questions?