Los Alamos
NATIONAL LABORATORY
EST. 1943

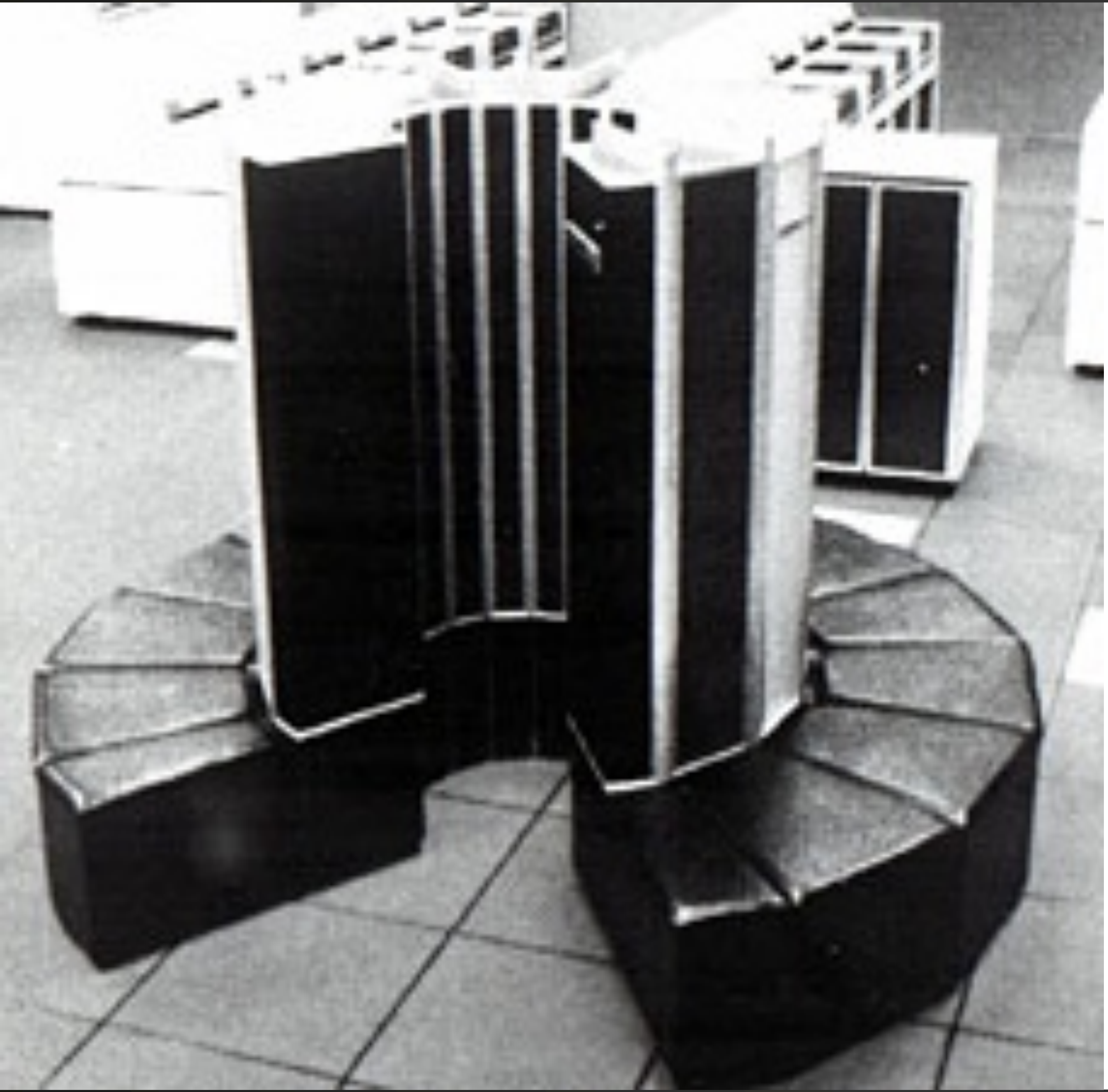
Los Alamos
NATIONAL LABORATORY
EST. 1943

Delivering science and technology
to protect our nation
and promote world stability

Intro to High Performance Computing

Everything you need to know in 60 minutes

Presented by CSCNSI



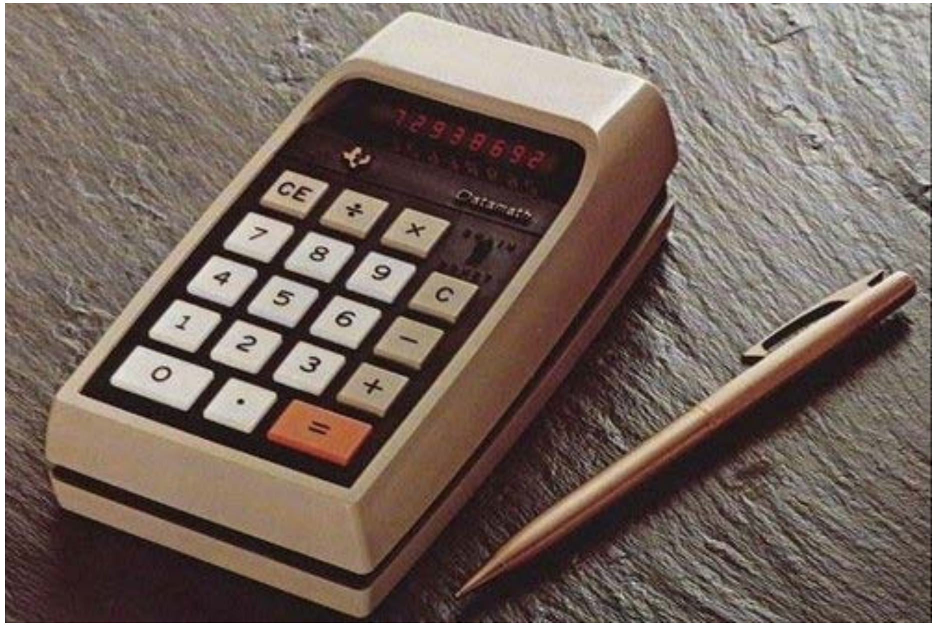
Agenda

- An toy problem
- An example real(ish) problem
- The Underlying Computer Science
- Cluster Computing
- HPC at Los Alamos

A Toy Problem: Summing Numbers

Serial Computing

- The problem: Add all of the numbers between 1 and 100
- The constraint: It takes you two seconds to add two numbers on your calculator
- How long does it take to add up 1 to 100?
  - Approach: Accumulate the sum in a single variable
    - n = 1+2; n = n+3; n = n+4; …, n = n+100
  - Time: (99 additions) * 2s = 198s = 3.3 minutes
  - Plus, your finger hurts



Serial Computing

- How could you make this faster?
  - Make the calculator's processor faster
    - But this isn't really the limiting factor
  - Make your finger work faster
    - This is the slow part, but there's a limit
  - Leverage the associative property of addition to recompose the problem as independent sets of addition, and then distribute those to more people

The women who operated the wartime Laboratory's desktop calculators and punched-card machines were themselves were called "computers." They often were the wives of Los Alamos scientists.

Parallel Computing

- The same problem: Add all of the numbers between 1 and 100
- The same constraint: It takes you two seconds to add two numbers on your calculator
- But you have three friends with calculators too
- How long does it take to add up 1 to 100?
  - Approach: Each person adds 25 numbers; one person then adds the four subtotals together
  - Parallel Time: (24 additions) *2s + (3 additions) *2s = 54s
  - Serial Time: 198 seconds
  - Speedup: 198s / 54s = 3.67x
- Note: 4x resources did not result in 4x speedup. Why not?

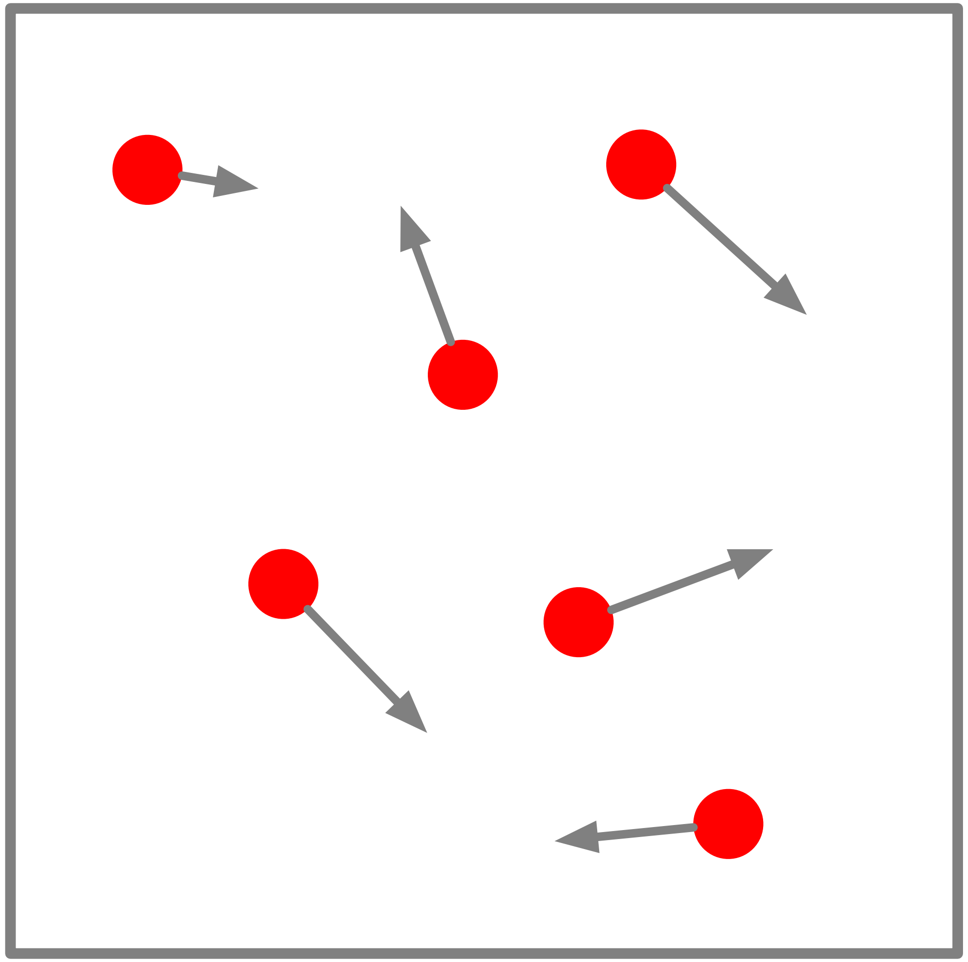
Limits of Parallel Computing

- This problem has two parts: one part that can be parallelized (the first stage additions), and one that can't (the final reduction)
- If you try to parallelize this too much, you start losing efficiency
  - 10 people
    - (9 additions)*2s + (9 additions)*2s = 36s
    - Speedup: 198s/36s = 5.5x
  - 25 people
    - (3 additions)*2s + (24 additions)*2s = 54s
    - Speedup: 198s/54s = 3.67x
  - 50 people
    - (1 addition)*2s + (49 additions)*2s = 100s
    - Speedup: 198s / 100s = 1.98x
- Eventually, the serial part of the computation dominates the parallel part

A Real(ish) Problem: Particles in a Box

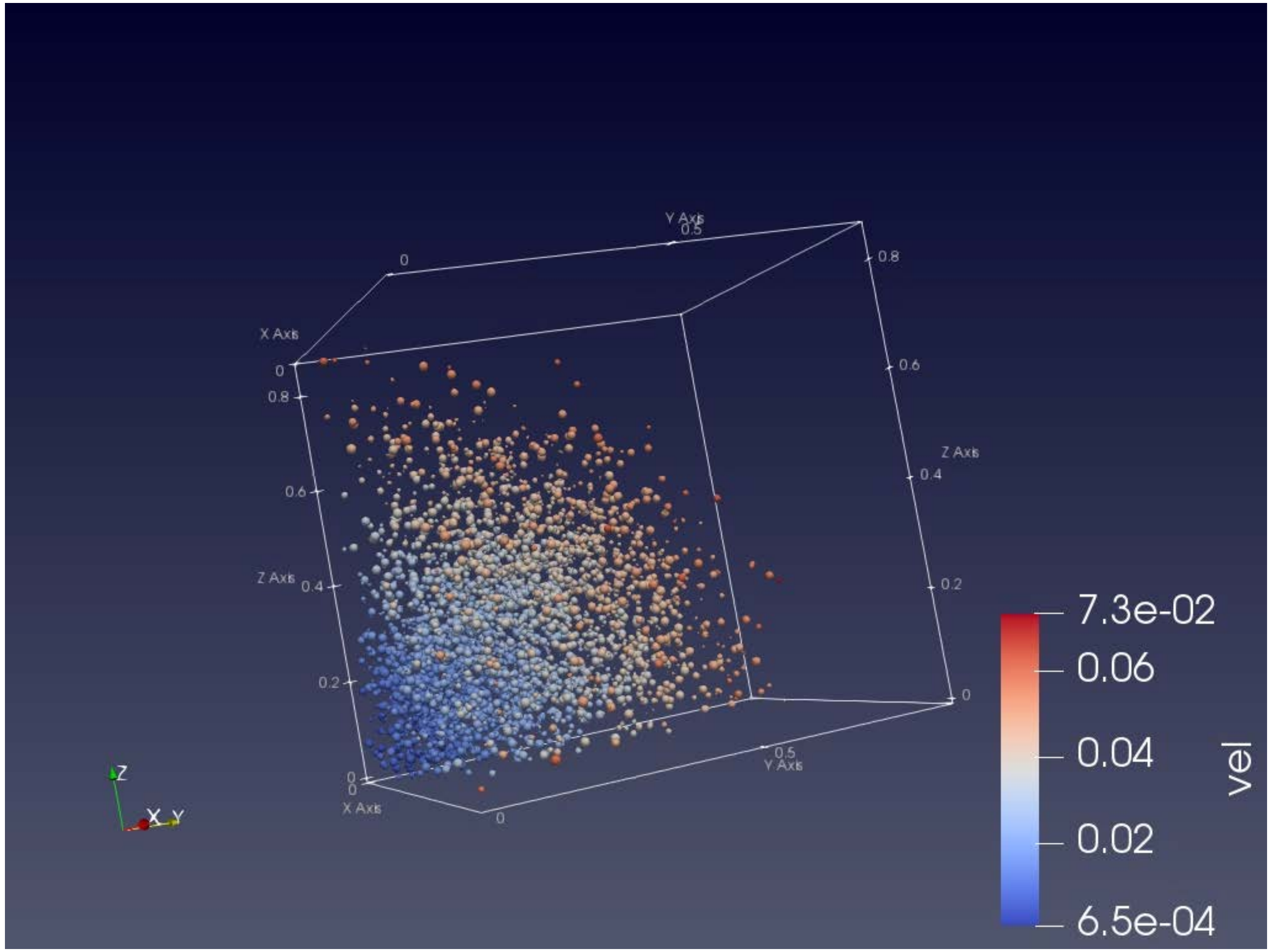
Real Parallel Computing

- The problem: Simulate the interactions between gas particles in a cube-shaped room
- The approach:
  - Model each particle as a set of coordinates in a three dimensional space and a three dimensional motion vector
  - Give each particle a set of initial conditions
  - Start a loop that performs one simulation timestep with each pass through the loop

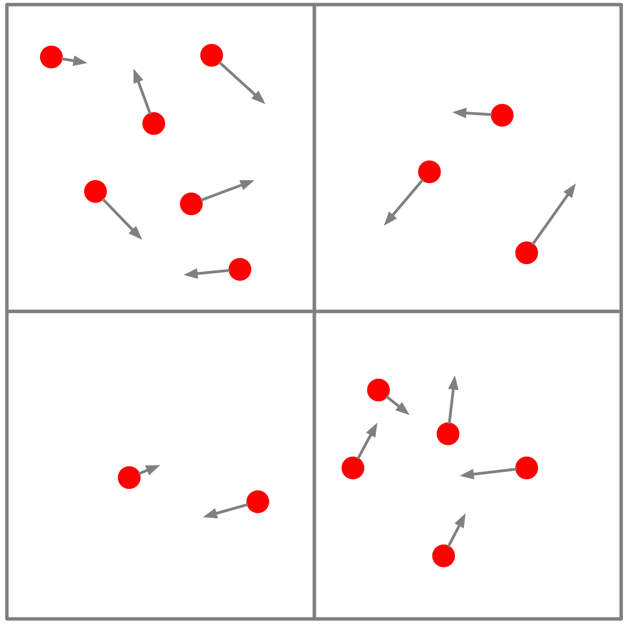


Serial Particles in a Box

- Running on one single processor system
- Store the particles in an array
- For each timestep of the simulation:
  - For each particle in the array:
    - Given the position and motion of this particle, calculate which other particles it will collide with in this timestep
    - Update motion vectors accordingly
- Some Advantages:
  - Simple to code
- Some Disadvantages:
  - Slower than parallel approaches
  - Size of simulation limited by system's memory size



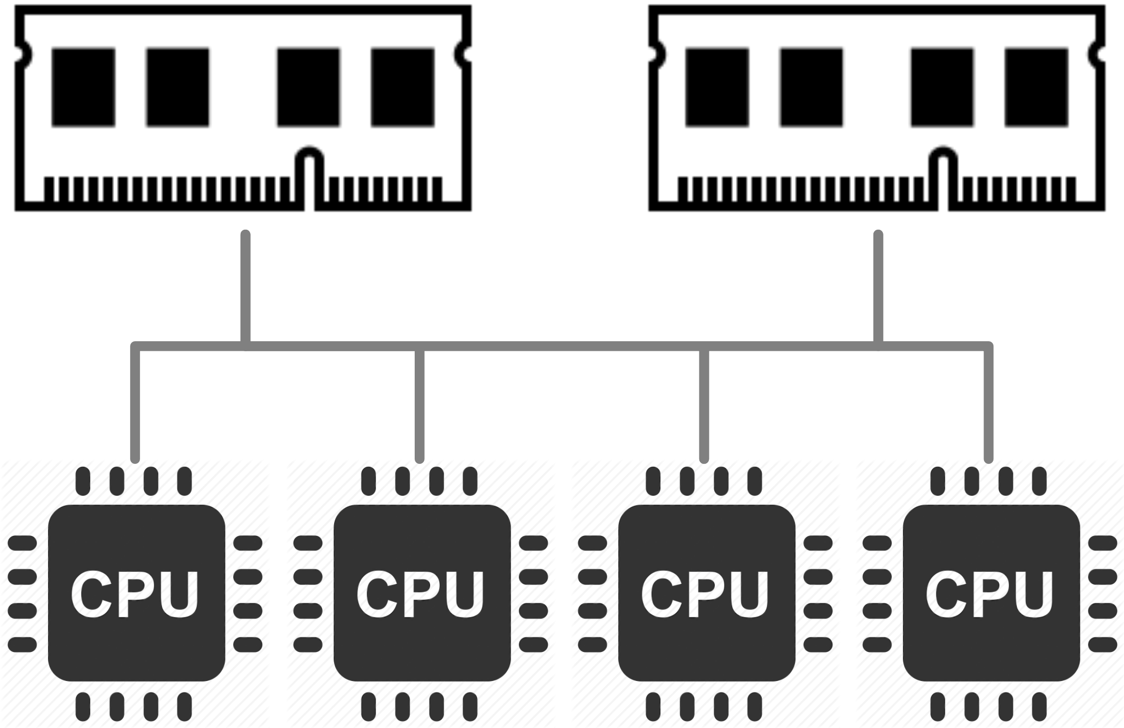
Parallel, Shared Memory Particles in a Box



- Running on one multi-core system
- Divide room into *n* equally-sized cells
- Store particles in one shared array
- Assign each cell to one of the processor cores:
  - Each core is responsible for running calculations on particles in its cell
- For each timestep in the calculation, each core runs:
  - For each particle currently located in the core's cell:
    - Given the position and motion of this particle, calculate which other particles it will collide with in this timestep
    - Update motion vectors accordingly
- Since this is a shared-memory system, all cores can see and update all particles

Parallel, Shared Memory Particles in a Box

- Some advantages
  - Can run many times faster than the serial version
- Some disadvantages
  - More complex than the serial version
  - Scaling larger requires adding higher-core processors to the system
  - Size of simulation still limited by system's memory size



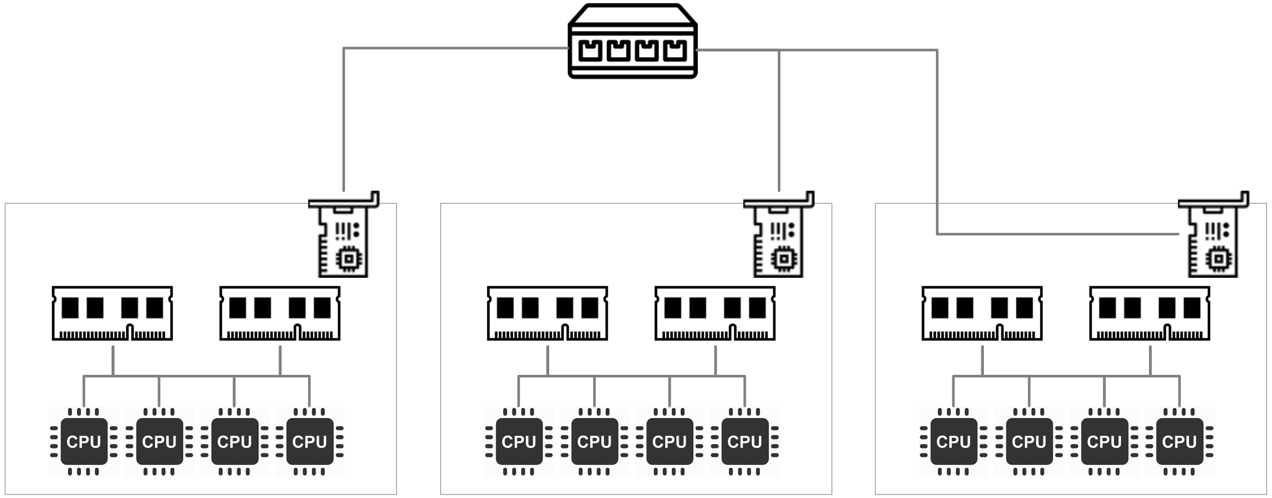
Parallel, Distributed Memory Particles in a Box

- Running on many single-core systems
- Divide room into *n* equally-sized cells
- Assign each cell to one of the systems:
  - Each system is responsible for running calculations on particles in its cell
- Store a cell's particles in an array on the system that owns it
- For each timestep in the calculation, each system runs:
  - For each particle currently located in the cell:
    - Given the position and motion of this particle, calculate which other particles it will collide with in this timestep
    - Update motion vectors accordingly
- Since this is a distributed memory system, information about a particle that leaves a cell must be explicitly sent to the system that owns the particle's new cell

Parallel, Distributed Memory Particles in a Box

- Some advantages
  - System memory only limits the size of a cell, not the size of the whole simulation
  - More processors can be added by adding new compute nodes
- Some disadvantages
  - Much more complex than the serial version
  - Communication can quickly become the bottleneck

- *One final version*: Run on many multi-core systems
  - Local cells can share memory, remote cells need to pass information
  - This is how modern parallel codes actually work



The Underlying Computer Science

Some Computer Science

- Processors can handle instructions and data in different ways:
  - SISD: Single Instruction, Single Data
  - MISD: Multiple Instruction, Single Data
  - SIMD: Single Instruction, Multiple Data
  - MIMD: Multiple Instruction, Multiple Data

Some Computer Science: The Processors

- Single-core CPU: SISD
  - One instruction run on one set of registers during each clock cycle
- Multi-core CPU: MIMD
  - Independent instructions run on independent sets of registers during each clock cycle
- Single-core CPU with Vector Additions: SISD + SIMD
  - Can do both: one instruction run on one set of registers, or one instruction run against a vector (or “list” or “array”) of registers
  - Examples: SSE, AVX
- GPU: SIMD
  - Load up lots of registers, run instructions across all of them *very* quickly

Some Computer Science: Sharing Data

- Shared memory:
  - On a single system, all processors can see all of the memory
  - This makes it easy to share memory between processes: it’s all accessible
  - But! It might not all be accessible at the same speed
  - Non-uniform Memory Access (NUMA) designs place parts of memory “closer” to some processors than others – this results in longer access times for cross-NUMA domain reads
  - Scales to as many processors and as much memory as you can fit in a single system
- Parallelization techniques that make use of shared memory:
  - Threads: The most basic way to share memory between multiple process-like objects
  - OpenMP: Specification for adding parallelism to C, C++, and Fortran

Some Computer Science: Sharing Data

- Message passing:
  - Distributed systems cannot see each others' memory
  - This makes it harder to share memory between processors, but not impossible
  - Software libraries exist for explicitly sending information between systems
    - This is referred to as Message Passing
  - This information is normally sent over a high speed, low latency network
  - But! Again, it may not all be accessible at the same speed
    - Network topologies make some nodes "closer" to and "farther" from other nodes
  - Scales as far as your network can grow
- Parallelization techniques that make use of message passing:
  - RPC: A standard concept with many different implementations for moving data between systems
  - MPI: The most common high performance computing API for message passing

Cluster Computing

Designing a compute cluster

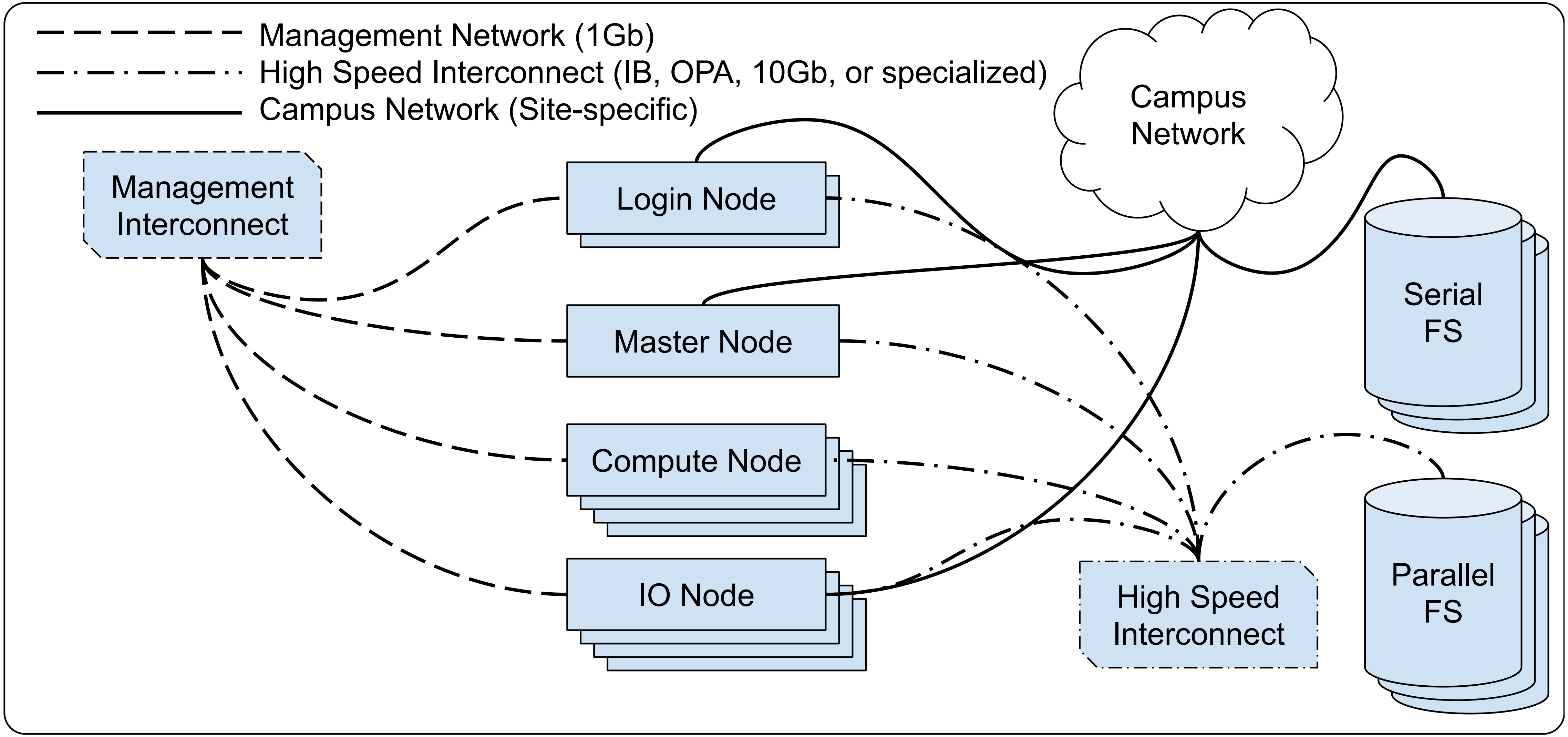
- Using what we now know about parallelization and memory, how do we build a system that can actually simulate the gas particles in a room?

- Today, HPC systems are most commonly implemented as clusters of independent computers
  - So they use a combination of shared memory & message passing.
- In this model, applications have to balance the available processing cores, memory space, network bandwidth, and other factors to get the best performance
- When designing a cluster, we try to find the best combination of hardware, software, and physical layout to optimize one or more of these factors

Typical cluster hardware

- HPC Clusters consist of…
  - Compute nodes
    - Tens, hundreds, thousands, or tens of thousands of individual computers
  - Infrastructure nodes
    - Tens or hundreds of computers that provide login gateways, manage compute nodes, interface with external filesystems, and perform other duties
  - Networks
    - Low-speed (1Gbps, frequently) management network
    - High speed (hundreds of Gbps, frequently) computation network
  - Filesystems
    - Frequently separated from clusters, both physically and logically
    - Standard serial network filesystems (NFS, normally) for home directories, specialized parallel filesystems (Lustre, GPFS) for computational use

A Typical HPC Cluster Layout



HPC at Los Alamos

So, what is HPC?

- There is no single definition of what high performance computing is
- But there are several indicators:
  - Often scientific in nature
  - Often highly parallel
  - Often tightly coupled
  - Often involves a dedicated high-speed network
  - Often involves a dedicated parallel filesystem
- … but sometimes doesn’t involve many of those at all

HPC at LANL

- LANL HPC is generally…
  - **Scientific**: physics, chemistry, biology, astrophysics, mechanical engineering, climatology, computational entomology, and more
  - **Parallel**: sometimes small parallelization (O(10^1) cores on one node), sometimes enormous parallelization (O(10^4) cores on thousands of nodes)
  - **Tightly coupled**: frequent communication between many or all nodes
  - **Run on clusters**: systems with dedicated high-speed networks and parallel filesystems using message passing

HPC in the Cluster Institute

- Over the next two weeks, you will build a cluster that is very similar to larger clusters at the lab
- These clusters will exhibit many of the traits we’ve talked about in this lecture
- Specs:
    - 10 individual compute nodes
    - Each compute node having multiple cores
    - All compute nodes tied together by an Infiniband high speed network
    - Capable of running parallel scientific applications
- We’ll start building this system later today!

Questions?