CIS-4230 Parallel Programming

Peter C. Chapin
Vermont Technical College

Concurrent Programming

- Concurrent Programming
  - Threads define independent activities
  - Threads often blocked
  - Threads do not need to execute simultaneously
  - Execution on a uniprocessor makes sense
  - Examples
    - Thread for UI events; screen updates occur in concurrently with other processing.
    - Thread for background data processing. Program might (or might not) be doing other things at the same time.

Parallel Programming

- Parallel Programming
  - Threads work on the same task
  - Threads not blocked; program *CPU bound*
  - Threads must execute simultaneously
  - Execution on a uniprocessor is pointless
  - Examples
    - Large scientific and engineering computations: supernova simulations, flow of air through a jet engine.
    - Processing “big data”: queries over huge data sets.

Simple Parallel Example

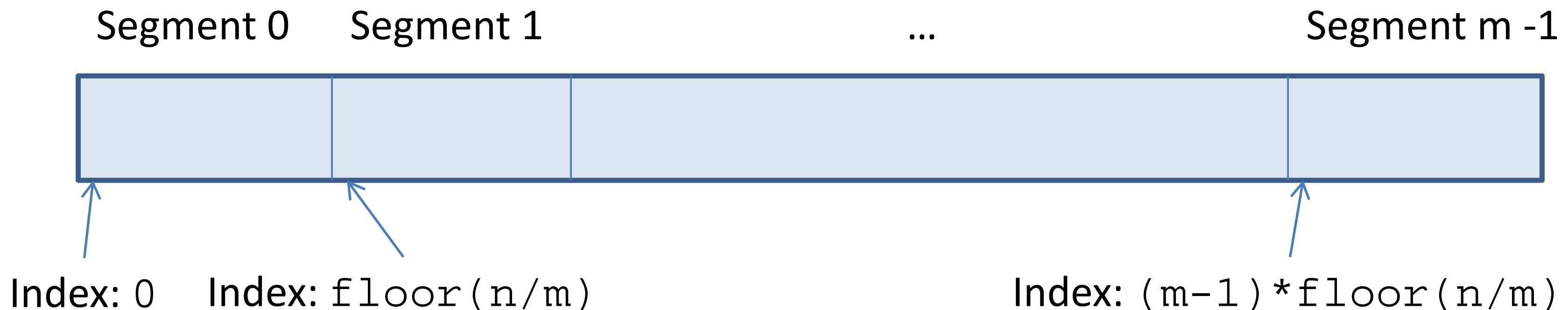
- Add Elements in Large Array
  - Serial version…

```
double sum_serial( double *array, int size )
{
  int i;
  double sum = 0.0;

  for( i = 0; i < size; ++i ) sum += array[i];
  return sum;
}
```

Simple Parallel Example

- Parallel Version; n Elements, m Threads
  - Partition array into m segments…



Each thread adds elements in one segment
Partial sums combined to compute final result

Simple Parallel Example

- Comments
  - One hopes it goes `m` times faster
    - BUT… ***complete waste of effort on uniprocessor***
  - Solution much more complicated
    - Create threads
    - Divide problem (map subproblems to threads)
    - Compute solution of subproblems in parallel (reduce each subproblem to a subsolution)
    - Combine subsolutions
  - Solution requires addition to be associative
    - Does it require addition to be commutative?
    - ***Additions no longer done in well defined order***.

Alternative Formulation

- Threads add interleaved data:
  - Thread #0 adds a[0], a[m], a[2m], …
  - Thread #1 adds a[1], a[m + 1], a[2m + 1], …
  - Thread #2 adds a[2], a[m + 2], a[2m + 2], …
- Does this require addition to be associative?
- Does this require addition to be commutative?

Goals

- Writing Parallel (not Concurrent) Programs
  - Make programs faster by using multiple *processing elements* (PEs) at the same time
  - Commonalities with concurrent programming:
    - Thread management and coordination
    - Problems associated with simultaneously updating shared data
  - Differences with concurrent programming:
    - Scaling to huge number of PEs.
    - Keeping PEs busy.

Why Do We Care?

- High Performance Computing (HPC)
  - Large scale scientific and engineering computation
    - Been using parallel systems (clusters, etc.) for years
- Multi-Core Processors
  - Desktop (and portable!) systems
    - Parallel processing is (relatively) new
    - Applications are different than with HPC. Unclear how to best parallelize them
  - *Increased performance now depends on utilizing multiple PEs. Faster processors slow in coming*.

Shared Memory Parallelism

- Shared Memory Parallelism
  - Everything I’ve talked about so far
  - All PEs read/write a common memory
    - Easy to understand; hard to program
    - Fast
    - Doesn’t scale well (100 PEs max?)
  - Symmetric Multi-Processors (SMP) and multi-core machines

Multi-Machine Parallelism

- Multi-Machine Parallelism (Clusters, Cloud)
  - Machines do not have a common memory
    - Inter-machine communication slow (e.g., network)
    - Programming model difficult; data synchronization easier
    - Scales well (10,000+ PEs feasible)
  - All modern super computers are designed like this

Fastest Machine on Earth

- As of November 2017: Sunway TaihuLight
  - National Supercomputing Center, Wuxi, China
  - Peak performance 125 PetaFLOPS (125,000 TeraFLOPS†)
  - Over 10,000,000 PEs.
  - Power consumption: 15.4 MW (yes, *mega*watts)
  - http://www.top500.org/

† FLOPS = "Floating Point Operations per Second"

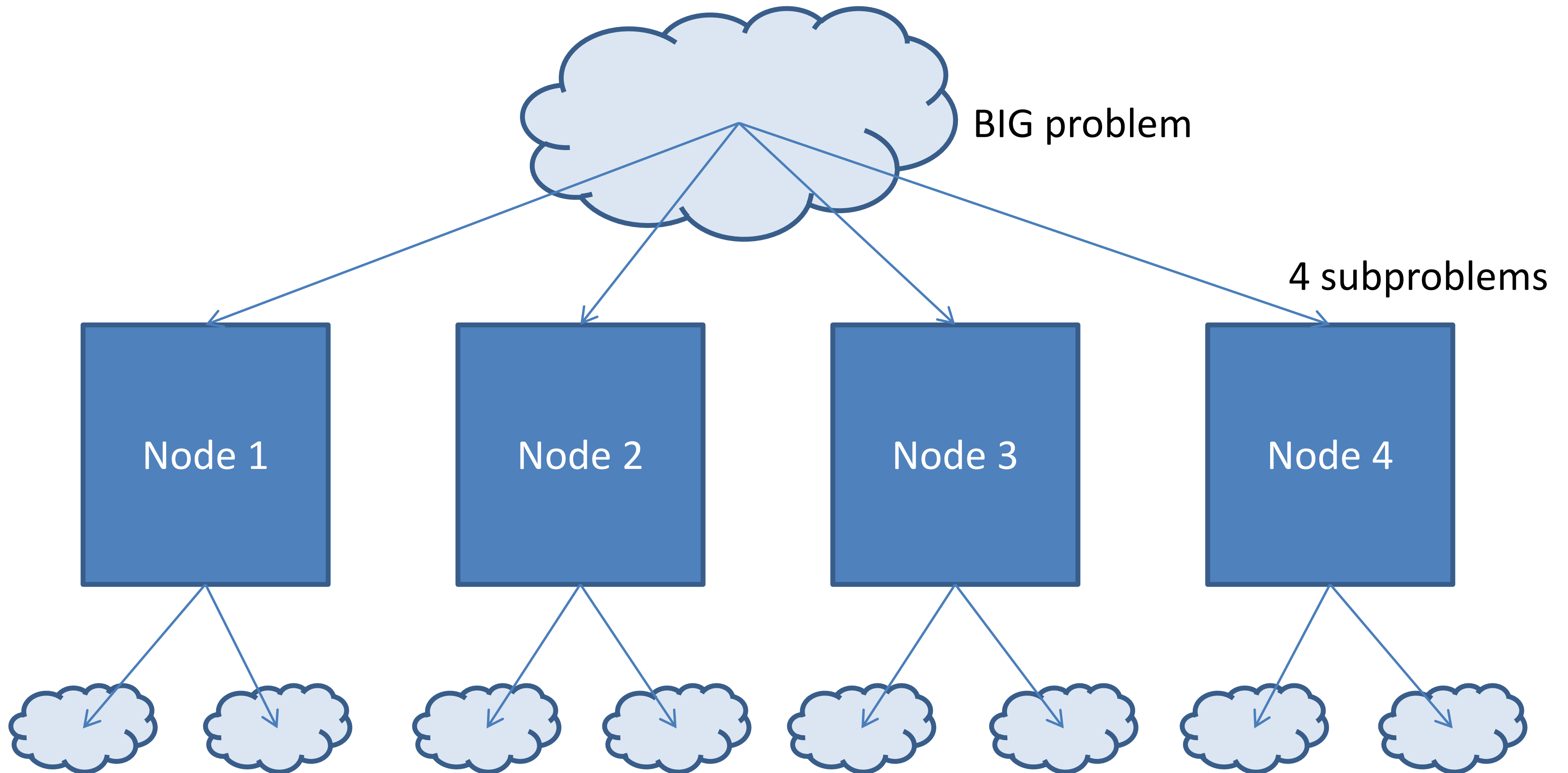
ExaFLOP Machines?

- ExaFLOP machine by 2020+?
  - Such a machine could do 10^{18} floating point operations per second
    - That's one billion operations per nanosecond!
  - Limiting factor: *power*
    - Estimated 2 GW. The power produced by Hoover Dam!

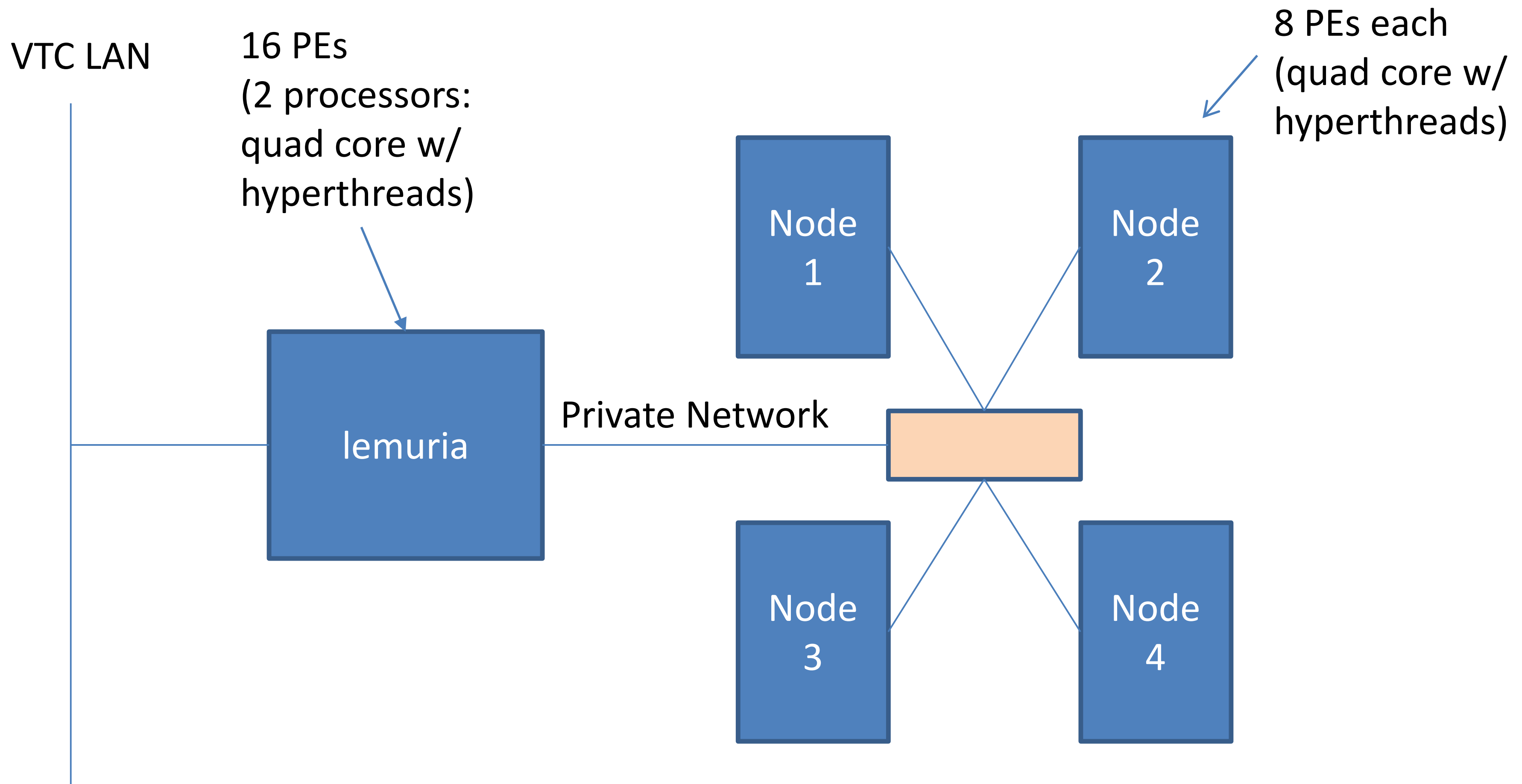
Communication vs Computation

- BIG Problem → Many subproblems
  - Subproblems largely independent
    - Lots of computation in each subproblem
    - Minimal communication between subproblems
    - *Good for implementation on cluster*
  - Subproblems tightly coupled
    - Lots of communication between subproblems
    - *Good for shared memory*
    - Hard to apply a huge number of PEs.

Best of Both Worlds?



VTC Cluster



GPGPU

- Commodity Graphics Cards
  - Do lots of computation in parallel
  - NVIDIA (and others) allow general purpose programs to execute on the graphics card
    - CUDA
    - OpenCL
    - OpenACC
  - Not suitable for all programs but very fast when it works. Excellent performance/price ratio.
  - VTC cluster nodes have CUDA cards

Course Organization

- Lectures on Adobe Connect
- Class Materials on Web Site
  - http://www.pchapin.org/VTC/cis-4230/
  - First assignment already posted!
  - Home work submitted electronically on Moodle
- Programming with gcc on lemuria & cluster
  - Programming in plain C. Use of C++ allowed
- Grade book on Moodle

Don’t forget to have fun!