Parallel Programming

Peter Chapin
CIS-4230, Vermont State University

Concurrent Programming

- Concurrent Programming
  - Threads execute independent activities
  - Threads often blocked (or "suspended" or "sleeping")
  - Threads do not *need* to execute simultaneously
  - Execution on a uniprocessor makes sense
  - Examples
    - Threads for UI events (frequently blocked waiting for user)
    - Client/Server applications (e.g., server uses one thread per client)
    - Threads for background data processing (can execute while UI is blocked)

Parallel Programming

- Parallel Programming – All about speed
  - Threads work on the same job
  - Threads not blocked; program *CPU bound* (as opposed to *IO bound*)
  - Threads must execute simultaneously to be useful
  - Execution on a uniprocessor is pointless
  - Examples
    - Large scientific and engineering computations: supernova simulations, airflow through a jet engine.
    - Processing "big data": queries over giant data sets.

Kernel Threads

- The OS kernel manages kernel threads
  - Kernel creates them
  - Kernel coordinates them
- Kernel schedules threads onto the *processing elements*
  - Processing element: processor, core, etc.
- Processor management is hidden from the application…
  - … *only the kernel can schedule threads that will run simultaneously*

Context Switching

- What if there are more threads than processors?
  - The kernel switches between the threads so all get a chance to run
  - Switching can be pre-emptive. The kernel periodically suspends a thread and resumes another one (that was previously suspended)
    - The pre-emption rate might be, for example, 100 Hz
  - Switching can happen when a thread blocks
    - … on I/O wait (user interface, network interface, storage media, etc.)
    - … waiting for another thread (to terminate, to release a lock, etc.)
  - *Blocked threads do not consume any CPU time!*

Most Threads Are Blocked Most of the Time

CPU1 has 1% utilization

Lemuria has 16 PEs

```
top - 09:37:35 up 4 days, 20:23,  1 user,  load average: 0.28, 0.30, 0.23
Tasks: 349 total,   1 running, 348 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  1.0 us,  0.3 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.7 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :  0.0 us,  0.0 sy,  0.0 ni, 99.3 id,  0.7 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  24062.0 total,  13779.3 free,   4834.9 used,   5447.9 buff/cache
MiB Swap:  24560.0 total,  24551.4 free,      8.6 used.  18750.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   7319 hadoop    20   0 8448052 483980  19848 S   0.7   2.0  57:48.11 java
1069617 pchapin   20   0   13556   4196   3344 R   0.7   0.0   0:00.28 top
   1495 mongodb   20   0 1498880 110972  38880 S   0.3   0.5  24:26.57 mongod
   1534 bind      20   0 1753108  26924   7596 S   0.3   0.1   0:59.07 named
   1556 tomcat    20   0   10.3g 407616  24820 S   0.3   1.7   9:21.97 java
```

Top CPU users

Number of processes = 349

Uniprocesor?

- This behavior means even a single processing element is okay
  - For a <u>concurrent</u> system, one CPU can be context-switched to all runnable processes without any perceptible loss of performance

Parallel Programs Use Maximum CPU Time

Attempts to use all CPUs

High idle times are bad

```
top - 10:00:25 up 4 days, 20:46,  2 users,  load average: 2.75, 1.31, 0.63
Tasks: 346 total,   1 running, 345 sleeping,   0 stopped,   0 zombie
%Cpu0  :  7.9 us,  0.3 sy,  0.0 ni, 91.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  8.3 us,  0.3 sy,  0.0 ni, 91.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  8.6 us,  3.3 sy,  0.0 ni, 88.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  8.4 us,  0.0 sy,  0.0 ni, 91.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  7.9 us,  0.0 sy,  0.0 ni, 92.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  : 71.4 us,  0.7 sy,  0.0 ni, 27.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  : 68.2 us,  0.7 sy,  0.0 ni, 31.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  : 72.5 us,  0.3 sy,  0.0 ni, 27.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  : 67.8 us,  1.0 sy,  0.0 ni, 31.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  : 73.3 us,  0.7 sy,  0.0 ni, 26.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 : 67.8 us,  0.7 sy,  0.0 ni, 31.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 : 72.3 us,  0.3 sy,  0.0 ni, 27.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 : 67.7 us,  1.0 sy,  0.0 ni, 31.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 : 70.5 us,  0.7 sy,  0.0 ni, 28.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 : 68.2 us,  0.7 sy,  0.0 ni, 31.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 : 71.3 us,  0.3 sy,  0.0 ni, 28.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  24062.0 total,  12998.3 free,   5614.5 used,   5449.2 buff/cache
MiB Swap:  24560.0 total,  24551.4 free,      8.6 used.  17970.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1075043 pchapin   20   0 1046240 783144   1744 S  1116   3.2   2:08.50 Sum.exe
   2689 root      20   0 2369620 426800 284596 S   4.3   1.7  97:55.46 s1-agent
   7319 hadoop    20   0 8448052 483980  19848 S   1.6   2.0  57:59.54 java
   1495 mongodb   20   0 1498880 110908  38880 S   1.0   0.5  24:31.50 mongod
```

1100% CPU!

Equivalent to 11 processors

Note: This was captured right as Sum.exe was ending

Uniprocessor?

- A single processing element would be context-switched across the threads
  - Each CPU-bound thread would be effectively slower
  - *No benefit!*
- <u>Parallel programs need multiple processing elements</u>
  - They *should not try to use more threads than PEs*

User Mode Threads

- An application library that manages threads entirely contained in the application
  - Kernel is not aware of user mode threads
  - Cannot make use of multiple PEs
  - If one thread blocks, it can block the entire process unless the user mode thread library does fancy stuff with asynchronous I/O
- Only useful for concurrent programming
  - *Thread creation, synchronization, and context switching are faster*

Other Terms

- Fibers (not parallel)
  - A concept from the Windows API whereby a thread is broken into several concurrent executions.
  - It allows a single kernel thread to become multiple user threads
    - See CreateFiber in the Windows API
- Coroutines (not parallel)
  - Two functions that yield control to each other before returning
    - A feature of various programming languages: Python, Kotlin, JavaScript, C#, Swift, Rust, C++ (to name a few)

Simple Parallel Example

- Add elements in a large array (serial version)

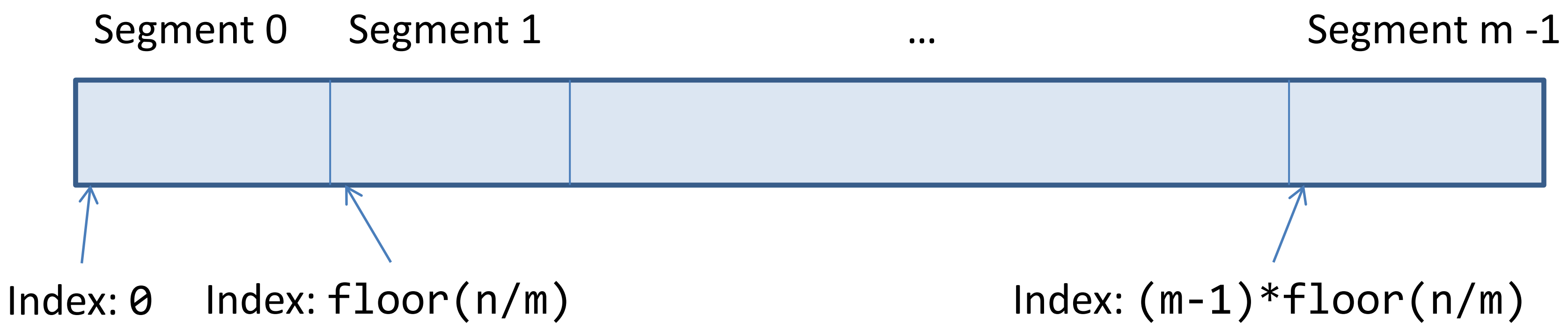
```
double sum_serial( double *array, size_t size )
{
    double sum = 0.0;

    for( size_t 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 the elements in one segment.
Partial sums are combined to compute the 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 the 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? Answer: No. (Why?)
    - ***Additions are no longer done in increasing-index 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?
  - Answer: Yes
- Does this require addition to be commutative?
  - Answer: Yes (Why?)

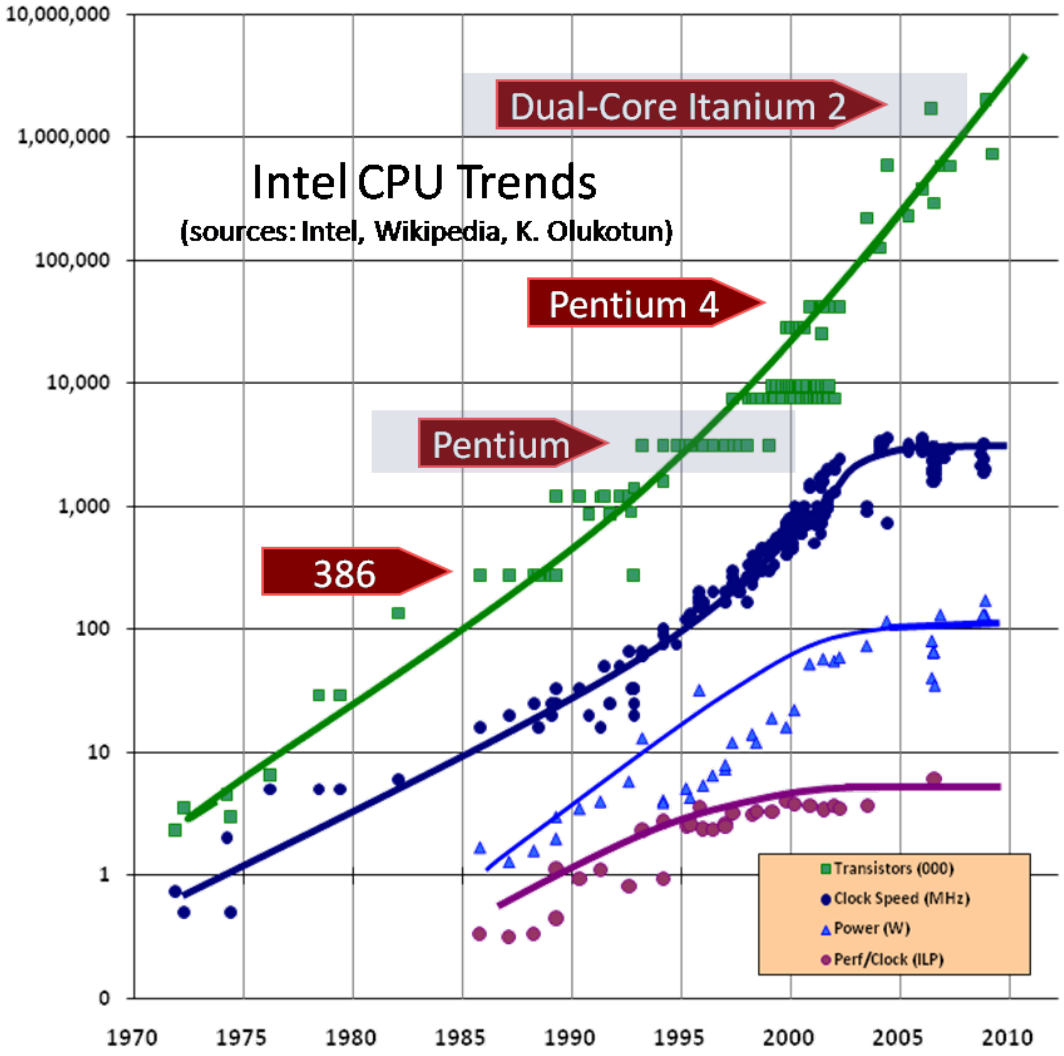
Goals

- 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 a 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*.

The Free Lunch is Over



[The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software (gotw.ca)]

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 2023: “Frontier”
  - Oak Ridge National Laboratory, USA
  - Peak performance 1,680 PetaFLOPS (1.68 x 10^{18} FLOPS†)
  - Almost 8,700,000 PEs.
  - Power consumption: 22.7 MW (yes, *mega*watts)
  - http://www.top500.org/

† FLOPS = “Floating Point Operations per Second”

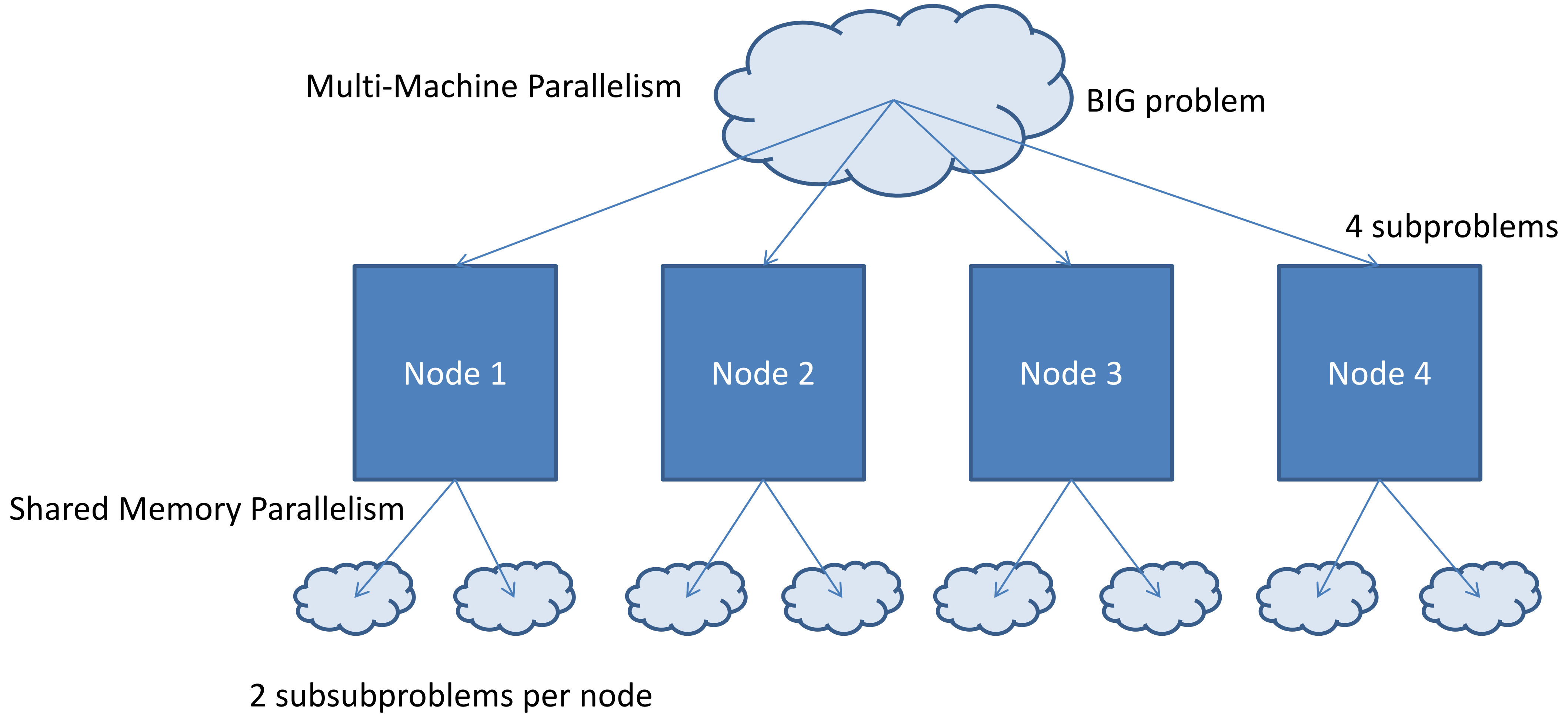
ExaFLOP Machines!

- ExaFLOP machines!
  - In 2010 it was estimated such a machine could be built by 2020.
    - 10^{18} floating point operations per second!
    - That's one billion floating point operations per nanosecond!
  - The limiting factor was: *power*
    - 2010 estimate: 2 GW. The power produced by Hoover Dam!
    - Today: Frontier uses 22.7 MW, or 100x less. *You can thank your phone!*

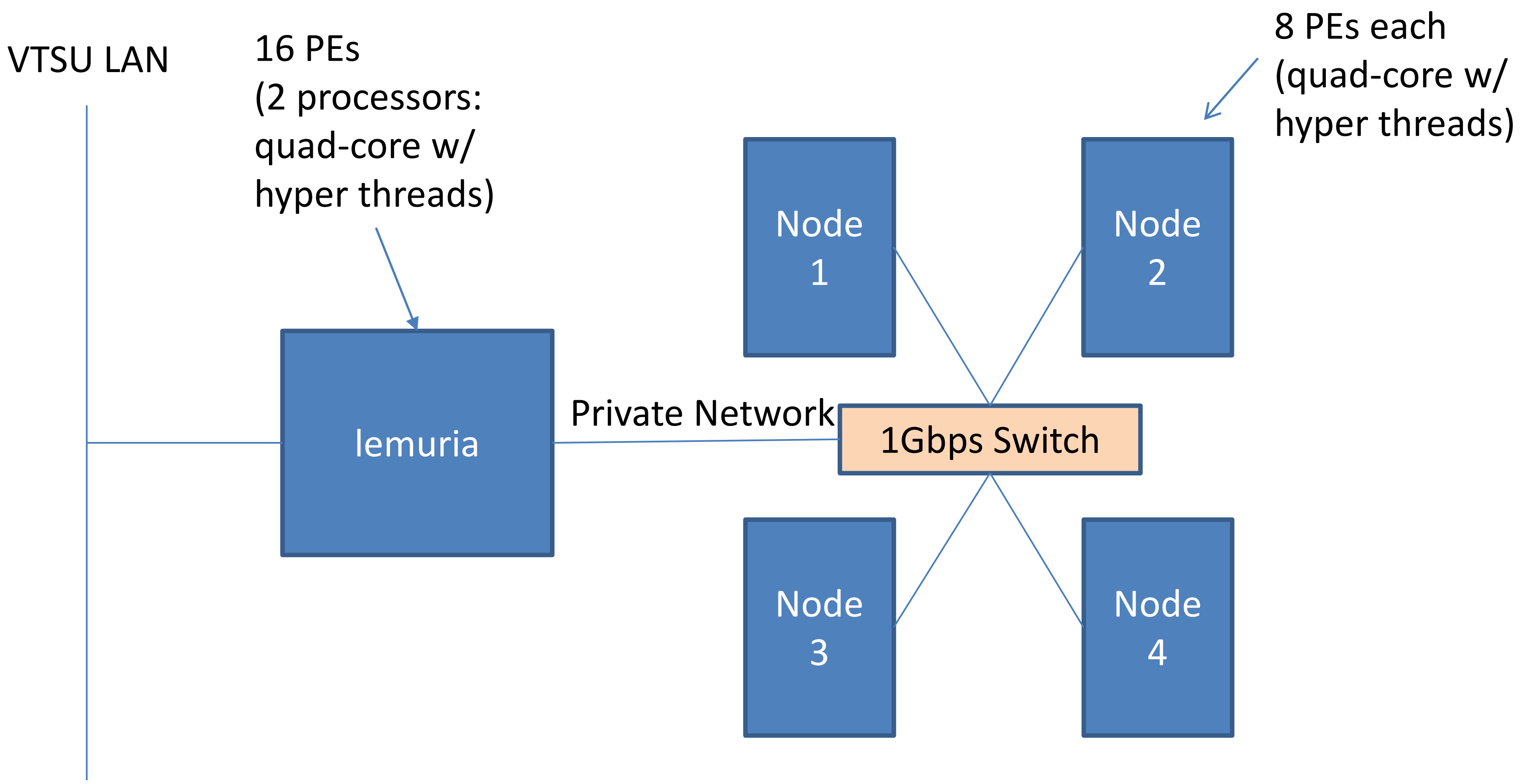
Communication vs Computation

- 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?



VTSU Cluster



General Purpose Graphics Processing Unit (GPGPU)

- Commodity Graphics Cards
  - Do lots of computation in parallel.
  - NVIDIA (and others) allows general-purpose programs to be executed on the graphics card.
    - CUDA (NVIDIA specific)
    - OpenCL (Vendor independent)
    - OpenACC (Vendor independent)
  - It is not suitable for all programs but is very fast when it works.
  - VTSU cluster nodes have NVIDIA CUDA graphics cards.

Course Organization

- Lectures on Zoom
- Class Materials on Web Site
  - http://lemuria.cis.vermontstate.edu/~pchapin/cis-4230/
  - First assignment already posted!
  - Homework submitted electronically on Canvas
- Programming with GCC on the Lemuria cluster
  - Programming in plain C. Use of C++ allowed
- Grade book on Canvas

Why C?

- C is very low level
  - Hard to use (correctly)
  - Thread management is complicated
- There are other languages/frameworks/libraries
  - Program at a higher level
  - Easier, more robust
- C is more educational!
  - See how things work. Gain a deeper understanding

A Story

- This happened:
  - Scala has a *parallel collections* library where methods run in parallel.
    - Very easy to use. Just change an import.
  - I saw on a Scala forum: “I wrote this small program using parallel collections, and it’s slower than the serial version. Why?”
  - The program tried to add 1000 integers using a parallel vector.
  - The answer: “The overhead of thread management far overshadows any benefit of parallelism with such a small collection.”
  - *The OP would have known that if they had taken this course!*

Why Not Julia?

- Julia is an interesting programming language
  - Focuses on scientific/engineering applications
  - Also: data science, ML, statistics
  - Competes with MATLAB, Python, R
- Easy syntax (like Python), fast (like C)
- But…
  - … Julia is a niche language right now
  - … unclear where it will go

Don’t forget to have fun!