Why Parallelism?
One common definition

- A parallel computer is a collection of processing elements that cooperate to solve problems fast.

We care about performance *

We’re going to use multiple processors to get it.

Note: different motivation from “concurrent programming” using pthreads that will be done in Network programming lab course.
Parallel Processing, Concurrency & Distributed Computing

• Parallel processing
Performance (and capacity) is the main goal
More tightly coupled than distributed computation

• Concurrency
Concurrency control: serialize certain computations to ensure correctness, e.g. database transactions
Performance need not be the main goal

• Distributed computation
Geographically distributed
Multiple resources computing & communicating unreliably
“Cloud” or “Grid” computing, large amounts of storage
Looser, coarser grained communication and synchronization

• May or may not involve separate physical resources, e.g. multitasking “Virtual Parallelism”
Course theme 1:

Designing and writing parallel programs ... large scale!

Parallel thinking

1. Decomposing work into parallel pieces
2. Assigning work to processors
3. Orchestrating communication/synchronization

Abstractions for performing the above tasks

Writing code in popular parallel programming languages
Course theme 2:

Parallel computer hardware implementation: how parallel computers work

Mechanisms used to implement abstractions efficiently
•  - Performance characteristics of implementations
•  - Design trade-offs: performance vs. convenience vs. cost

Why do I need to know about HW?
•  Because the characteristics of the machine really matter
•  Because you care about performance (you are writing parallel programs)
Course theme 3:

Thinking about efficiency

FAST != EFFICIENT

Just because your program runs faster on a parallel computer, it doesn’t mean it is using the hardware efficiently

– Is 2x speedup on 10 processors is a good result?

Programmer’s perspective: make use of provided machine capabilities

HW designer’s perspective: choosing the right capabilities to put in system (performance/cost, cost = silicon area?, power?, etc.)
Course Logistics

Parallel Programming in C with MPI and OpenMP, Quinn (Quinn book)

Introduction to High Performance Computing for Computational Scientists and Engineers, by Georg Hager and Gerhard Wellein. (Hager book)

Parallel Algorithm Design – Foster’s Design Methodology – Example Problems. (Parallel Patterns from UIUC and UCB)
Message Passing programming Model – MPI – Point to Point & Collective Calls.
Algorithms for Illustrations – Sieve of Eratosthenes – Floyd’s Algorithm.
Performance analysis
  - Speed up and Efficiency
  - Amdahl’s Law
  - Gustafson’s Barsis Law
  - Karp Flatt Metric
  - Isoefficiency Metric.
Matrix Vector Multiplication
Monte Carlo Methods
Matrix Multiplication
Solving linear System
finite Difference Methods
sorting algorithm
combinatorial Search.
Shared Memory Programming – Open MP.
Piazza and github links

- Piazza site is up. (soft copies of Hager book and Pacheco book are available)
- Github site will be up soon.
- XSEDE accounts

- (2 or 3) Individual Programming Assignments (Academic integrity is must)
- (1 or 2) Group Programming Assignments
Why parallelism?

The answer 10 years ago
- To get performance that was faster than what clock frequency scaling would provide
- Because if you just waited until next year, your code would run faster on the next generation CPU

Parallelizing your code not always worth the time
- Do nothing: performance doubling ~ every 18 months
End of frequency scaling

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Power Wall

\[ P = CV^2F \]

- **P**: power
- **C**: capacitance
- **V**: voltage
- **F**: frequency

Higher frequencies typically require higher voltages
Power vs. core voltage

Pentium M

Credit: Shimin Chin
Programmable invisible parallelism

Bit level parallelism
- 16 bit  32 bit  64 bit

Instruction level parallelism (ILP)
- Two instructions that are independent can be executed simultaneously
- “Superscalar” execution
ILP example

\[ a = (x^2 + y^2 + z^2) \]
ILP scaling
Single core performance scaling

• The rate of single thread performance scaling has decreased (essentially to 0)

  1. Frequency scaling limited by power
  2. ILP scaling tapped out

• No more free lunch for software developers!
Why parallelism?

The answer 10 years ago

- To get performance that was faster than what clock frequency scaling would provide
- Because if you just waited until next year, your code would run faster on the next generation CPU

The answer today:

- Because it is the only way to achieve significantly higher application performance for the foreseeable future
Multi-cores

Intel Sandy Bridge
8 cores

IBM Power 7
8 cores

AMD MAGNY-COURS
12 cores

The 62-core Xeon Phi coprocessor
The PCI card housing a Xeon Phi coprocessor
NVIDIA Kepler (2012)

The Tesla K20 GPU coprocessor card
With 2496 CUDA cores, 1.17 Tflops DP
Mobile processing

Power limits heavily influencing designs

Apple A5: (in iPhone 4s and iPad 2)
Dual Core CPU + GPU + image processor
and more

NVIDIA Tegra:
Quad core CPU + GPU + image processor...
Supercomputing

• Today: clusters of CPUs + GPUs
• Pittsburgh Supercomputing Center: Backlight
• 512 eight core Intel Xeon processors
  - 4096 total cores
ORNL Titan (#1, Nov 2012)

Some more relevant info from Top500
Summary (what we learned)

Single thread performance scaling has ended

- To run faster, you will need to use multiple processing elements
- Which means you need to know how to write parallel code

Writing parallel programs can be challenging

- Problem partitioning, communication, synchronization
- Knowledge of machine characteristics is important
What you should get out of the course

In depth understanding of:

• When is parallel computing useful?
• Understanding of parallel computing hardware options.
• Overview of programming models (software) and tools, and experience using some of them
• Some important parallel applications and the algorithms
• Performance analysis and tuning
• Exposure to various open research questions
Programming for performance
Motivation

• Most applications run at < 10% of the “peak” performance of a system
  – Peak is the maximum the hardware can physically execute
• Much of this performance is lost on a single processor, i.e., the code running on one processor often runs at only 10-20% of the processor peak
• Most of the single processor performance loss is in the memory system
  – Moving data takes much longer than arithmetic and logic

• To understand this, we need to look under the hood of modern processors
  – For today, we will look at only a single “core” processor
  – These issues will exist on processors within any parallel computer
Matrix Multiplication

Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Performance [Gflop/s]

- Vendor compiler, best flags
- Exact same operations count (2n^3)
Possible lessons to learn from these courses

• “Computer architectures are fascinating, and I really want to understand why apparently simple programs can behave in such complex ways!”

• “I want to learn how to design algorithms that run really fast no matter how complicated the underlying computer architecture.”

• “I hope that most of the time I can use fast software that someone else has written and hidden all these details from me so I don’t have to worry about them!”

• All of the above, at different points in time