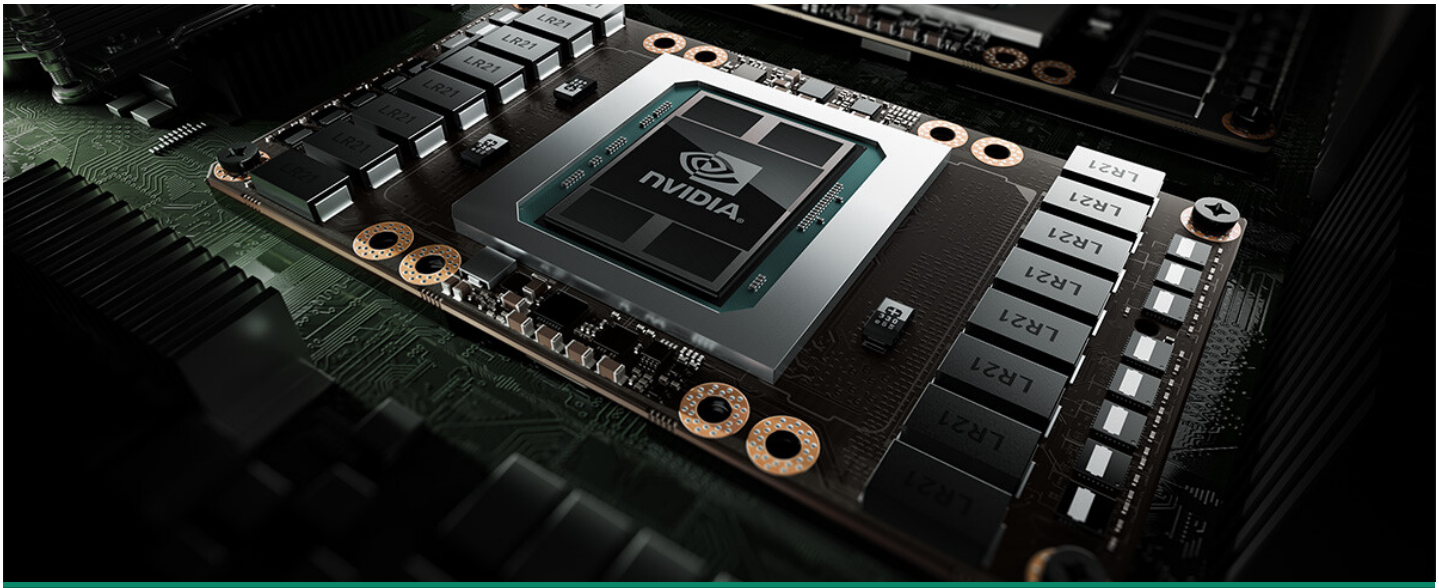
LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Administrivia and Introduction

Haidar M. Harmanani

Spring 2020



Administrivia

Course Introduction

- Lectures
  - TTh, 11:00-12:15 from January 21, 2019 until May 4, 2019
  - Prerequisites
    - Competency in a high-level programming language
    - CSC 310, Data Structures and Algorithms
    - CSC320, Computer Organization

Grading and Class Policies

- Grading
  - Midterm (1-2): 25%
  - Final: 40%
  - Programming Assignments [3-4]: 10%
  - Three milestones projects (OpenMP, OpenACC, and CUDA): 25%
- Exams Details
  - Exams are closed book, closed notes
- All assignments must be your own original work.
  - Cheating/copying/partnering will not be tolerated

Teaching Methodology

- We will use multiple choice questions to initiate discussions and reinforce learning
- We will also use a a flipped-classroom methodology for one part of the course

Programming Assignments

- All assignments and handouts will be communicated via `piazza`
  - Make sure you enable your account
- Use `piazza` for questions and inquiries
  - No questions will be answered via email
- All assignments must be submitted via `github`
  - `git` is a distributed version control system
  - `git` or its variations have become a universal standard for developing and sharing code
  - Make sure you get a private repo
    - Apply for a free account: https://education.github.com/discount_requests/new

Policy on Assignments and Independent Work

- With the exception of laboratories and assignments (projects and HW) that explicitly permit you to work in groups, all homework and projects are to be YOUR work and your work ALONE.
- It is NOT acceptable to copy solutions from other students.
- It is NOT acceptable to copy (or start your) solutions from the Web.
- PARTNER TEAMS MAY NOT WORK WITH OTHER PARTNER TEAMS
- You are encouraged to help teach other to debug. Beyond that, we don't want you sharing approaches or ideas or code or whiteboarding with other students, since sometimes the point of the assignment is the "algorithm" and if you share that, they won't learn what we want them to learn). We expect that what you hand in is yours.
- It is NOT acceptable to leave your code anywhere where an unscrupulous student could find and steal it (e.g., public GITHUBs, walking away while leaving yourself logged on, leaving printouts lying around,etc)
- The first offense is a zero on the assignment and an F in the course the second time
- Both Giver and Receiver are equally culpable and suffer equal penalties

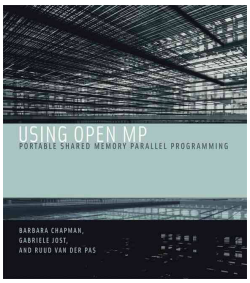
Contact Information

- Haidar M. Harmanani
  - Office: Block A, 810
  - Hours: TTh 9:00-10:30 or by appointment.
  - Email: haidar@lau.edu.lb

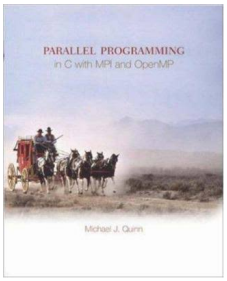
Topics

- Introduction to parallel programming
- Parallel programming performance
- Programming using `Pthreads`, `OpenMP`, `OpenACC`, and `CUDA`
- Introduction to Neural Networks and Deep Learning using `Python`

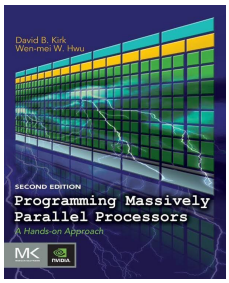
Course Materials



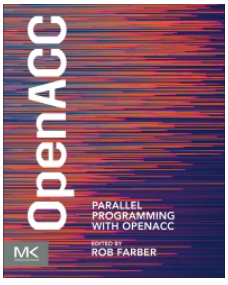
Optional



Optional



Required



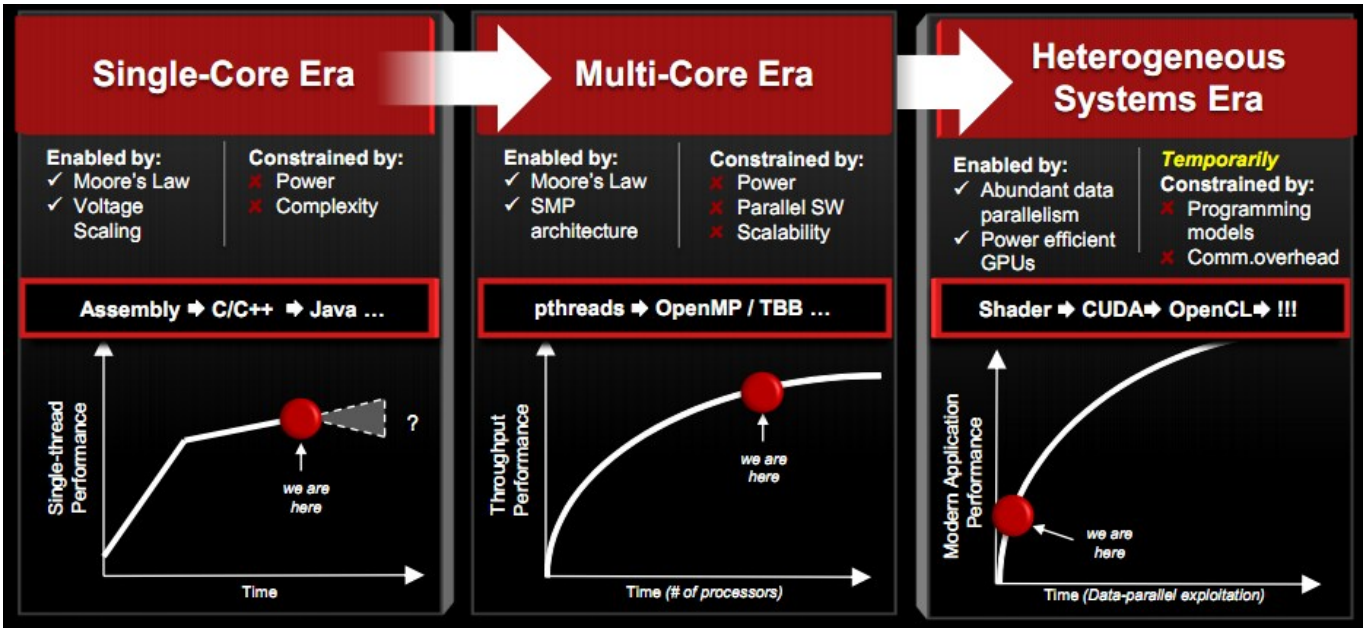
Optional

Computers/Programming

- You will be using:
  - The lab or your own machines for `Pthreads` and `OpenMP`
  - The Computer Science Lab for `OpenACC` and `CUDA` Programming, or your own machine if you have a CUDA-capable GPU
  - The *Cloud* for GPU Tutorial Labs

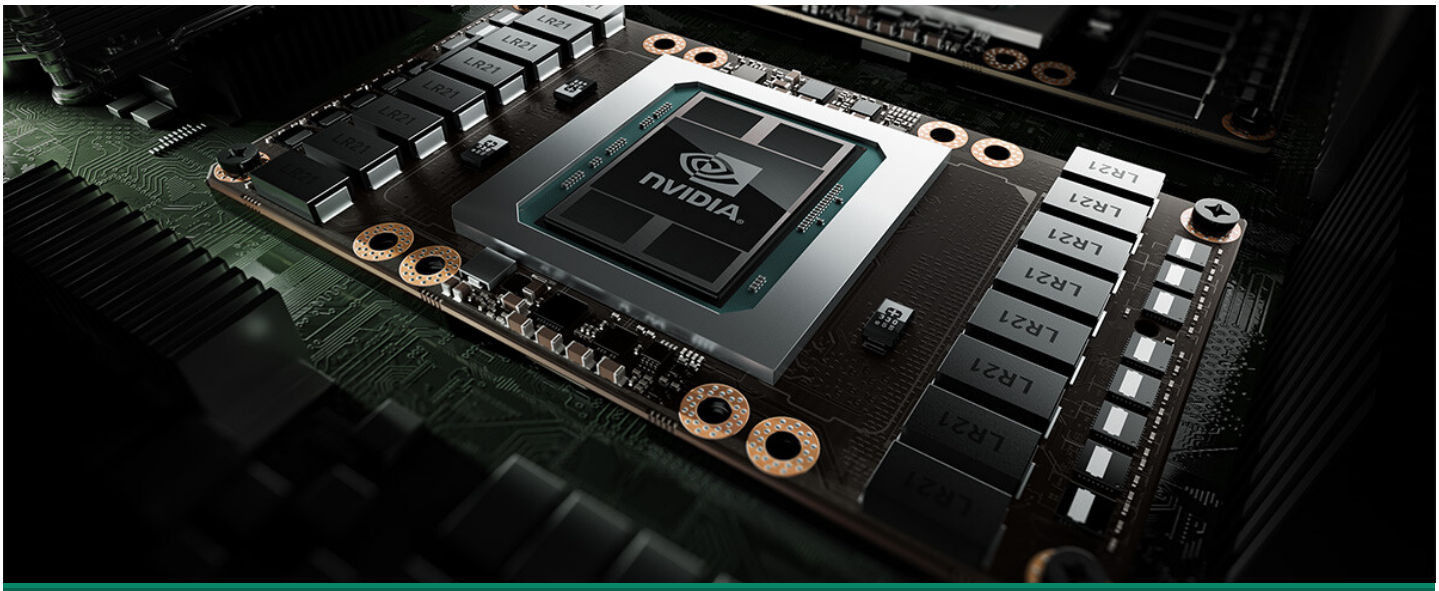
Supercomputing/Parallel Architecture Timeline

- Phase 1 (1950s): sequential instruction execution
- Phase 2 (1960s): sequential instruction issue
  - Pipeline execution, reservations stations
  - Instruction Level Parallelism (ILP)
- Phase 3 (1970s): vector processors
  - Pipelined arithmetic units
  - Registers, multi-bank (parallel) memory systems
- Phase 4 (1980s): SIMD and SMPs
- Phase 5 (1990s): MPPs and clusters
  - Communicating sequential processors
- Phase 6 (>2000): many cores, accelerators, scale, …



LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Administrative Questions?



Course Introduction

Definitions

- What is parallel?
  - Webster: *"An arrangement or state that permits several operations or tasks to be performed simultaneously rather than consecutively"*
- Parallel programming
  - Programming in a language that supports concurrency explicitly
  - An evolution of serial programming

Parallel Programming

- A parallel computer is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem
- Parallelism is all about performance! Really?

Concurrent Programming

- Concurrency is fundamental to computer science
  - Operating systems, databases, networking, …
- Multiple executing tasks are concurrent with respect to each if
  - They can execute asynchronously
  - Implies that there are no dependencies between the tasks

Concurrent Programming

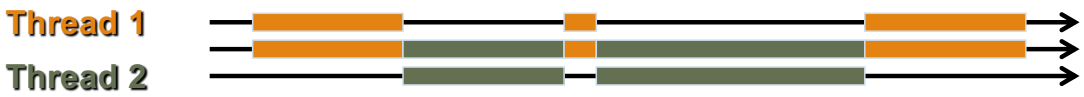
- Two threads can run *asynchronously* on the same core by interleaving instructions
- Dependencies
  - If a task requires results produced by other tasks in order to execute correctly, the task's execution is *dependent*
  - Some form of synchronization must be used to enforce (satisfy) dependencies

Concurrency and Parallelism

- Concurrent is not the same as parallel!
- Parallel execution
  - Concurrent tasks *actually* execute at the same time
  - Multiple (processing) resources have to be available
- Parallelism = concurrency + "parallel" hardware
  - Both are required
  - Find concurrent execution opportunities
  - Develop application to execute in parallel
  - Run application on parallel hardware

Concurrency vs. Parallelism

– Concurrency: two or more threads are in progress at the same time:



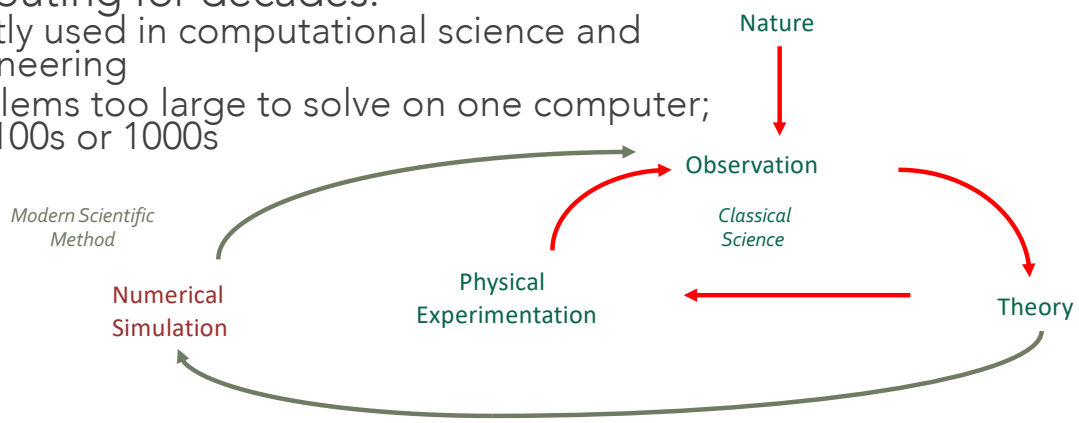
– Parallelism: two or more threads are executing at the same time

Why Parallel Computing Now?

- All major processor vendors are producing multicore chips
  – All machines are actually parallel machines
- Rise of artificial intelligence and machine learning!

Why Parallel Computing Now?

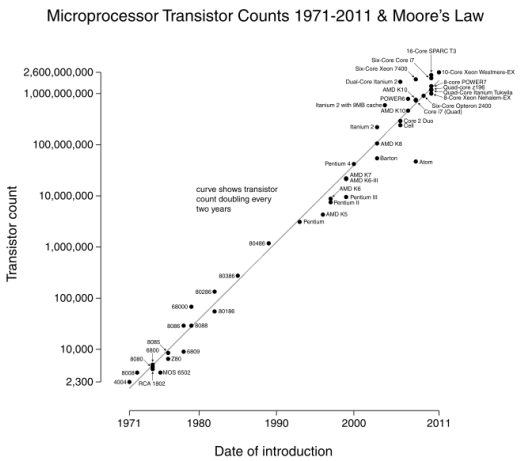
- Researchers have been using parallel computing for decades:
  - Mostly used in computational science and engineering
  - Problems too large to solve on one computer; use 100s or 1000s

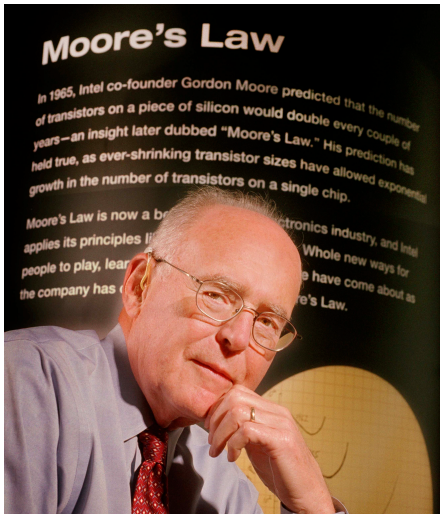


Why Parallel Computing Now?

- There are 3 ways to improve performance:
  - Work Harder
  - Work Smarter
  - Get Help
- Computer Analogy
  - Using faster hardware
  - Optimized algorithms and techniques used to solve computational tasks
  - Multiple computers to solve a particular task

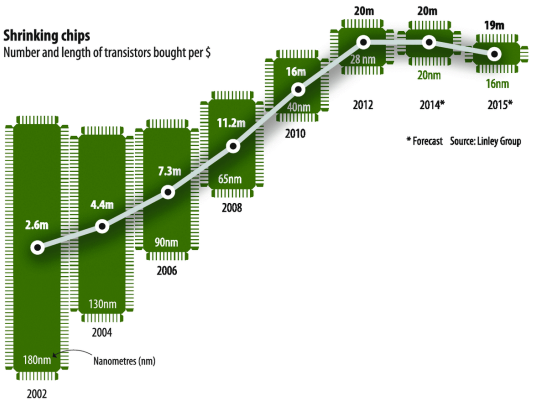
Technology Trends: Microprocessor Capacity

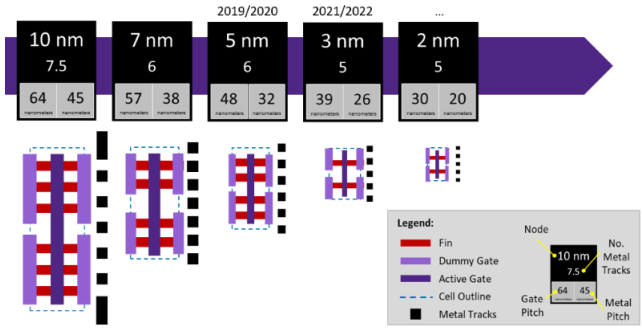




Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Technology Trends: Microprocessor Capacity





Microprocessors have become smaller, denser, and more powerful.

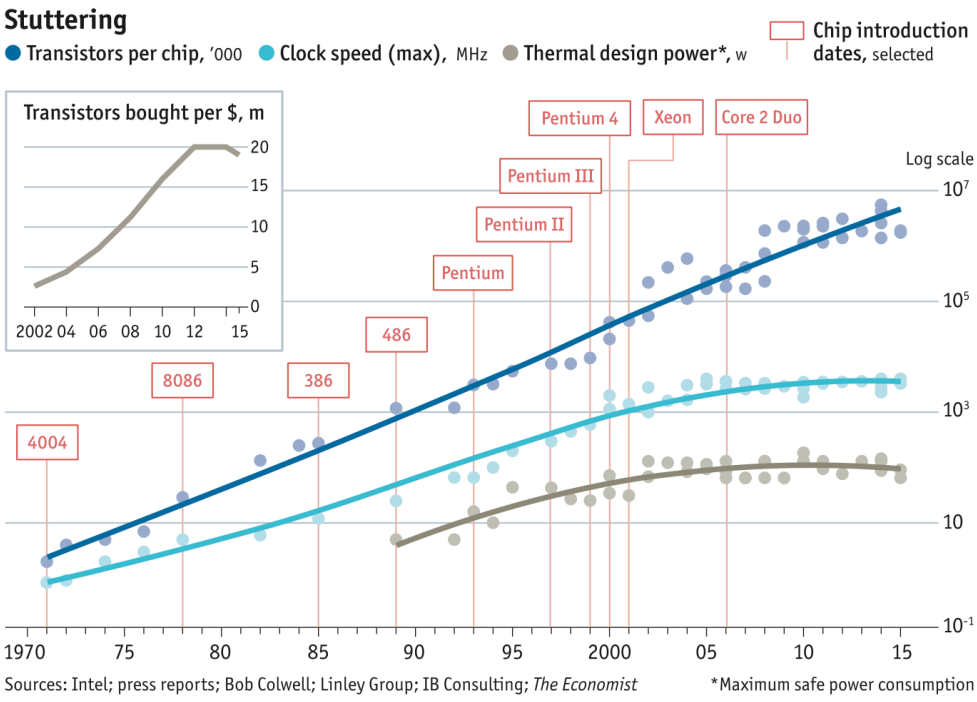
CPUs 1 Million Times Faster

- Faster clock speeds
- Greater system concurrency
  - Multiple functional units
  - Concurrent instruction execution
  - Speculative instruction execution

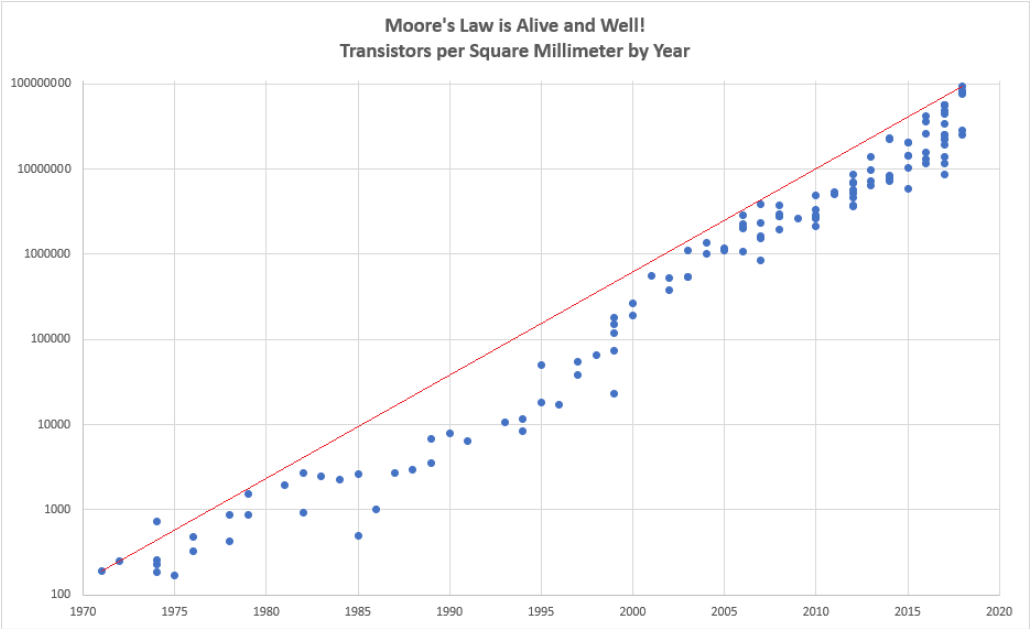
Systems 1 Billion Times Faster

- Processors are 1 million times faster
- Combine thousands of processors
- Parallel computer
  - Multiple processors
  - Supports parallel programming
- Parallel computing = Using a parallel computer to execute a program faster

Microprocessor Transistors and Clock Rate

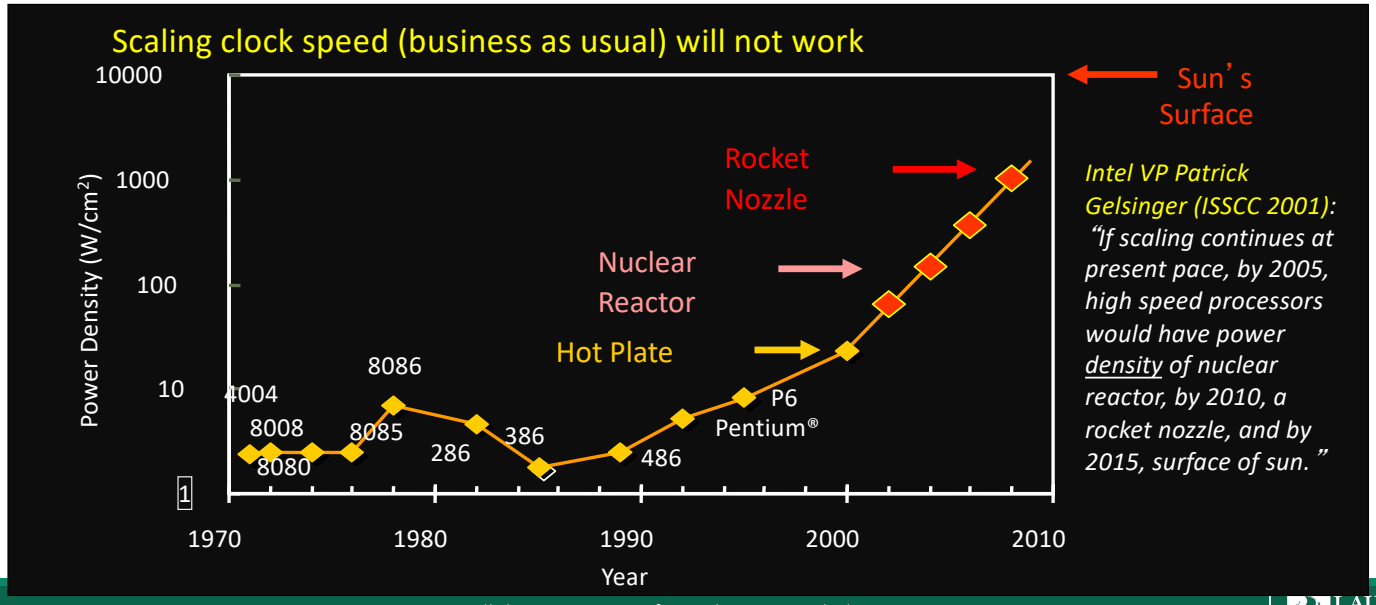


Microprocessor Transistors and Clock Rate



Why bother with parallel programming? Just wait a year or two…

Limit #1: Power density



Parallelism Saves Power

- Exploit explicit parallelism for reducing power

  Power = $(C * V^2 * F)/4$ Performance = $(Cores * F)*1$

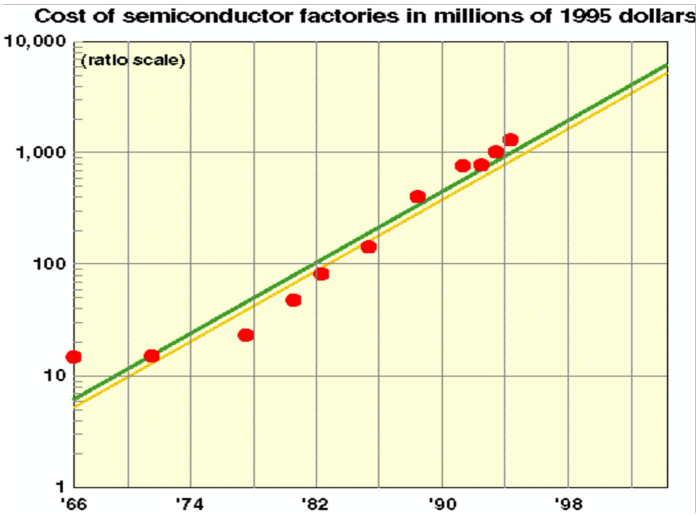
  Capacitance Voltage Frequency

- Using additional cores
  - Increase density (= more transistors = more capacitance)
  - Can increase cores (2x) and performance (2x)
  - Or increase cores (2x), but decrease frequency (1/2): same performance at ¼ the power
- Additional benefits
  - Small/simple cores → more predictable performance

Limit #2: Hidden Parallelism Tapped Out

- Superscalar designs provided many forms of parallelism not visible to programmer
  - Multiple instruction issue
  - Dynamic scheduling: hardware discovers parallelism between instructions
  - Speculative execution: look past predicted branches
  - Non-blocking caches: multiple outstanding memory ops
- Unfortunately, these sources have been used up

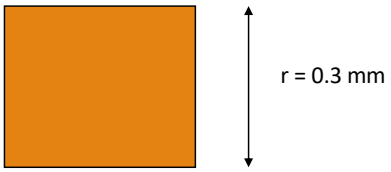
Limit #3: Manufacturing costs and yield problems limit use of density



- Moore's (Rock's) 2nd law: fabrication costs go up
- Yield (% usable chips) drops
- Parallelism can help
  - More smaller, simpler processors are easier to design and validate
  - Can use partially working chips:
  - E.g., Cell processor (PS3) is sold with 7 out of 8 "on" to improve yield

Limit #4: Speed of Light (Fundamental)

1 Tflop/s, 1 Tbyte sequential machine



- Consider the 1 Tflop/s sequential machine:
  - Data must travel some distance, r, to get from memory to CPU.
  - To get 1 data element per cycle, this means 1012 times per second at the speed of light, c = 3x108 m/s.
    - Thus r < c/1012 = 0.3 mm.
- Now put 1 Tbyte of storage in a 0.3 mm x 0.3 mm area:
  - Each bit occupies about 1 square Angstrom, or the size of a small atom.
- No choice but parallelism

So what is the problem?

Writing (fast) parallel programs is hard!

Parallel Programming Workflow

- Identify compute intensive parts of an application
- Adopt scalable algorithms
- Optimize data arrangements to maximize locality
- Performance Tuning
- Pay attention to code portability and maintainability

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
- Granularity
- Locality
- Load balance
- Coordination and synchronization

All of these things makes parallel programming even harder than sequential programming.

Finding Enough Parallelism

- Suppose only part of an application seems parallel
- Amdahl's law
  - let s be the fraction of work done sequentially, so (1-s) is fraction parallelizable
  - P = number of processors

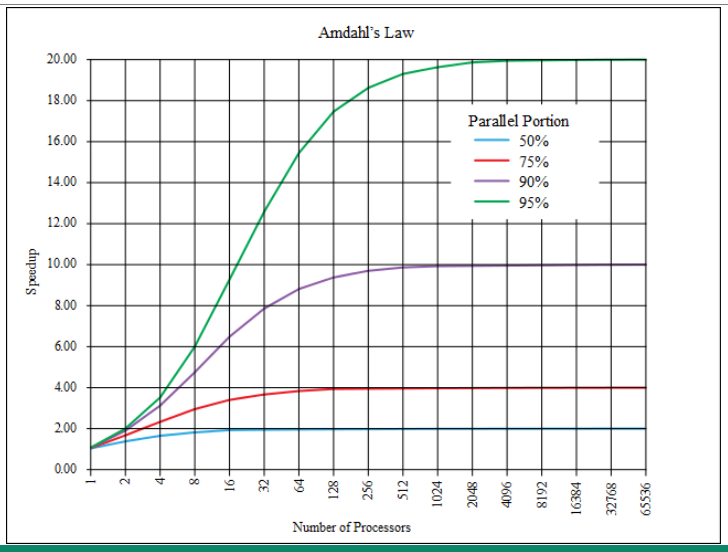
$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

$$\leq 1/(s + (1-s)/P)$$

$$\leq 1/s$$

- Even if the parallel part speeds up perfectly performance is limited by the sequential part

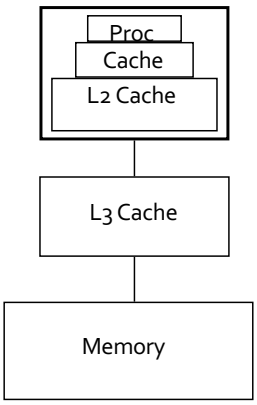
Finding Enough Parallelism (Amdahl's Law)

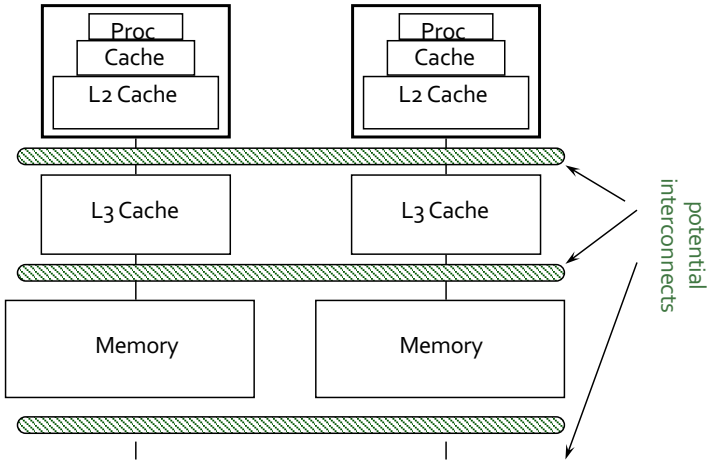


Granularity

- Given enough parallel work, this is the biggest barrier to getting desired speedup
- Parallelism overheads include:
  - Cost of starting a thread or process
  - Cost of communicating shared data
  - Cost of synchronizing
  - Extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (I.e. large granularity), but not so large that there is not enough parallel work

Conventional Storage Hierarchy





Large memories are slow, fast memories are small

Storage hierarchies are large and fast on average

Parallel processors, collectively, have large, fast cache
- the slow accesses to "remote" data we call "communication"

Algorithm should do most work on local data

Locality

Load Balance/Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to "interesting parts of a domain"
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load

Load Balance

Synchronization

- Need to manage the sequence of work and the tasks performing
- Often requires "serialization" of segments of the program
- Various types of synchronization maybe involved
  - Locks/Semaphores
  - Barrier
  - Synchronous Communication Operations

Performance Modeling

- Analyzing and tuning parallel program performance is more challenging than for serial programs.
- There is a need for parallel program performance analysis and tuning.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

So how do we do parallel computing?

Strategy 1: Extend Compilers

- Focus on making sequential programs parallel
- Parallelizing compiler
  - Detect parallelism in sequential program
  - Produce parallel executable program
- Advantages
  - Can leverage millions of lines of existing serial programs
  - Saves time and labor
  - Requires no retraining of programmers
  - Sequential programming easier than parallel programming
- Disadvantages
  - Parallelism may be irretrievably lost when programs written in sequential languages
  - Performance of parallelizing compilers on broad range of applications still up in air

Strategy 2: Extend Language

- Add functions to a sequential language
  - Create and terminate processes
  - Synchronize processes
  - Allow processes to communicate
- Advantages
  - Easiest, quickest, and least expensive
  - Allows existing compiler technology to be leveraged
  - New libraries can be ready soon after new parallel computers are available
- Disadvantages
  - Lack of compiler support to catch errors
  - Easy to write programs that are difficult to debug

Strategy 3: Add a Parallel Programming Layer

- Lower layer
  - Core of computation
  - Process manipulates its portion of data to produce its portion of result
- Upper layer
  - Creation and synchronization of processes
  - Partitioning of data among processes
- A few research prototypes have been built based on these principles

Strategy 4: Create a Parallel Language

- Develop a parallel language "from scratch"
  - occam is an example
- Add parallel constructs to an existing language
  - Fortran 90
  - High Performance Fortran
  - C*
- Advantages
  - Allows programmer to communicate parallelism to compiler
  - Improves probability that executable will achieve high performance
- Disadvantages
  - Requires development of new compilers
  - New languages may not become standards
  - Programmer resistance

Computational Thinking

Fundamentals of Parallel Computing

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time (strong scaling), and/or
  - To solve bigger problems (weak scaling), and/or
  - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Shared Memory vs. Message Passing

- We have focused on shared memory parallel programming
  - This is what CUDA (and OpenMP, OpenCL) is based on
  - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
  - However, you will find parallels for almost every technique you learned in this course
  - Need to be aware of space-time constraints

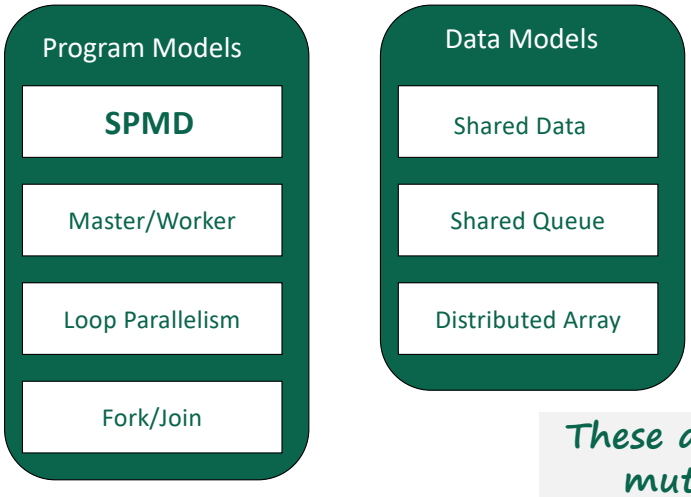
Data Sharing

- Data sharing can be a double-edged sword
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- Many:Many, One:Many, Many:One, One:One

Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
  - Watch out for serialization
- Important: be aware of which items of work are truly independent

Parallel Programming Coding Styles – Program and Data Models



These are not necessarily mutually exclusive.

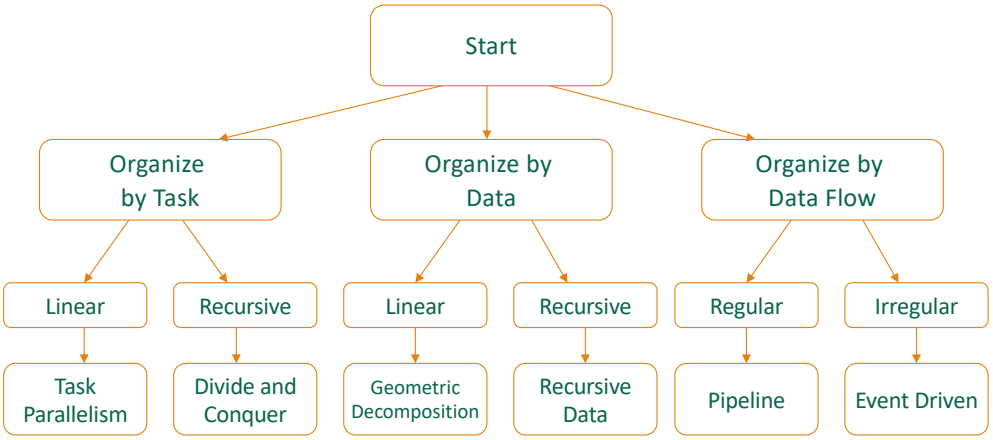
Program Models

- SPMD (Single Program, Multiple Data)
  - All PE's (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code
  - This is essentially the CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
  - Most general, generic way of creation of threads

Algorithm Structure



Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD (next to loop parallelism)
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - Use thread ID in loop index calculations to split loop iterations among threads
    - Potential for memory/data divergence
  - Use thread ID or conditions based on thread ID to branch to their specific actions
    - Potential for instruction/execution divergence

Both can have very different performance results and code complexity depending on the way they are done.

Making Science Better, not just Faster

or… in other words:
There will be no Nobel Prizes or Turing Awards awarded for "just recompile" or using more threads

Conclusion: Three Options

- Good: "Accelerate" Legacy Codes
  - Recompile/Run

  => good work for domain scientists (minimal CS required)
- Better: Rewrite / Create new codes
  - Opportunity for clever algorithmic thinking

  => good work for computer scientists (minimal domain knowledge required)
- Best: Rethink Numerical Methods & Algorithms
  - Potential for biggest performance advantage

  => Interdisciplinary: requires CS and domain insight

  => Exciting time to be a computational scientist

Think, Understand… then, Program

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
  - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part ☺)