1 Course Overview

- Even though Moore's Law[1] is still valid, heat and power are of primary concerns.
  - These challenges can be overcome with smaller and more efficient processors or simply more processors
  - To make better use of the added computation power, parallelism is used.
- Parallel vs. Concurrent: In both cases, one of the difficulties is to actually determine which processes can overlap and which can't:
  - Concurrent: Focus on which activities may b executed at the same time (= overlapping execution)
  - Parallel: Overlapping execution on a real system with constraints imposed by the execution platform.
- Parallel/Concurrent vs. distributed: In addition to parallelism/concurrency, systems can actually be physically distributed (e.g. BOINC)
- Concerns in PP:
  - Expressing Parallelism
  - Managing state (data)
  - Controlling/coordinating parallel tasks and data

2 Parallel Architectures

- Turing machine:
  - Infinite tape
  - Head that reads/writes symbols on tape
  - State registers
  - Program is expressed as rules: (reg)(head) $\rightarrow$ (reg)(head)(movement)
- Today's computers:
  - Consist of CPU, memory and I/O
  - Stored Program: program instructions are stored in memory
  - Von Neumann Architecture: Program data and program instruction in the same memory
- Since accessing memory became slower than accessing CPU registers, CPUs now have caches which are closer (faster and smaller) to the CPU. Caches are:
  - Faster then memory
  - Smaller than memory

[1] "The number of transistors on integrated circuits doubles approximately every two years"

  - Organized in multi-level hierarchies (e.g. L1,L2,L3)

- To improve sequential processor performance, you can use the following parallelism techniques:
  - Vectorization
    For example, when adding vectors (load → operation(s) → store)
    * Normal: 1-at-a-time
    * Vectors: N-at-a-time (bigger registers)
  - Pipelining[2]

    maybe add diagram from slides?

    * Multiple stages (CPU Functional Units)
      · Instruction Fetch
      · Instruction Decode
      · Execution
      · Data access
      · Writeback
    * Each instruction takes 5 time units (cycles)
    * 1 instruction per cycle (not always possible though)
  - Instruction Level Parallelism (ILP)
    * Superscalar CPUs
      · Multiple instructions per cycle
      · multiple functional units
    * Out-of-Order (OoO) Execution
      · Potentially change execution order of instructions
      · As long as the programmer observes the sequential program order
    * Speculative execution
      · Predict results to continue execution

- Moore's Law
  - *"The number of transistors on integrated circuits doubles approximately every two years"* - Gordon E.Moore, 1965
  - Actually an observation
  - For a long time, CPU Architects improved sequential execution by exploiting Moore's Law and ILP
  - More transistors → more performance
  - Sequential programs were becoming exponentially faster with each new CPU
    * (most) programmers did not worry about performance
    * They waited for the next CPU model

[2]Think laundry: you can either wash, dry, fold and repeat, or while the n load is drying, the $n+1$ load can start washing

- Architects hit walls
  - Power dissipation wall: Making CPU faster → expensive to cool
  - Memory Wall: CPUs faster than memory access
  - ILP Wall: Limits in inherent program's ILP, complexity
  - **No longer affordable to increase sequential CPU performance**
- Multicore processors
  - Use transistors to add cores (instead of improving sequential performance)
  - Expose parallelism to software
  - Implication: Programmers need to write parallel programs to take advantage of new hardware
    - Past: Parallel programming was performed by a few
    - Now: Programmers need to worry about (parallel) performance
- Shared memory architectures

  Maybe add picture form slides, page 34 form 2 - parallel architectures; ADD PICTURES FOR EACH OF THE DIFFERENT TYPES OF SMA

  - SMT (Hyperthreading)
    - Single core
    - Multiple instruction streams (threads); Virtual (phony) cores
    - Between ILP ↔ multicore
      - ILP: Multiple units for one instruction stream
      - SMT: Multiple units for multiple instruction streams
    - Limited parallel performance
  - Multicores
    - Single chip, multiple cores
    - Dual-, Quad-, x8...
    - Each core has its own hardware units; computations un parallel perform well
    - Might share part of the cache hierarchy
  - SMP (Symmetric MultiProcessing)
    - Multiple CPUs on the same system
    - CPUs share memory: same cost to access memory
    - CPU caches coordinate: Cache coherence protocol
  - NUMA (Non-Uniform Memory Access)
    - Memory is distributed
    - Local/Remote (fast/slow)
    - Shared memory interface

- Flynn's Taxonomy

  Add diagram from page 49 of 2 - parallel architectures

- GPUs
  - Graphical Processing Units
    - Scene description → pixels
    - Highly data-parallel process
  - Massively parallel vector machines
  - Not a standard component up until recently
  - Started very specialized (rigid pipelines)
  - Driven by game industry success
- GP-GPUs
  - General programming using GPUs (CUDA, OpenCL)
  - Much research on "how to execute X on a GPU"
  - Generally GPUs are:
    - Well suited for data parallel programs
    - Not very well-suited for programs with random accesses
    - People are rethinking algorithms
  - GPUs are, currently, something of a standard in the HPC (High Performance Computing) domain

3 Basic Concepts

- Performance in sequential execution:
  - Computational complexity
    - Theoretical computer science
    - Asymptotic behavior: big O notation ($\mathcal{O}$), or big Θ
    - How many steps does an algortithm take
    - Complexity classes
  - Execution Time: The less time, the better
- Sequential programs are much easier to write, but if we care about performance we have to write parallel programs
- Parallel Performance
  - Sequential execution time: T_1
  - Execution time T_p on p CPUs:
    - $T_p = \frac{T_1}{p}$ (Perfection)
    - $T_p > \frac{T_1}{p}$ (Performance Loss, what normally happens)
    - $T_p < \frac{T_1}{p}$ (Sorcery!)

- (Parallel) Speedup
  - (Parallel) speedup S_p on p CPUs:

$$S_p = \frac{T_1}{T_p}$$

    * $S_p = p \rightarrow$ linear speedup (Perfection)
    * $S_p < p \rightarrow$ sublinear speedup (Performance loss)
    * $S_p > p \rightarrow$ superlinear speedup (Sorcery!)
  - Efficiency: $\frac{S_p}{p}$
- Scalability: How well a system reacts to increased load
  - In Parallel Programming
    * Speedup when we increase processors
    * What will happen if number of processors $\rightarrow \infty$
- Performance loss ($S_p < p$) happens because:
  - Programs may not contain enough parallelism, e.g.:
    * pipeline with 4 stages on a 32-CPU machine
    * Some parts of the program might be sequential
  - Overheads introduced by parallelization; typically associated with coordination
  - Architectural limitations, e.g. memory contention
- Amdahl's Law
  - b: sequential part (no speedup)
  - $1 - b$: parallel part (linear speedup)

$$T_p = T_1 \left(b + \frac{1-b}{p} \right) \qquad S_p = \frac{p}{1 + b\,(p-1)}$$

  - Remarks About Amdahl's Law:
    * It concerns maximum speedup (Optimistic approach). Architectural constrains will make factors worse
    * Takeaway: **All non-parallel parts of a program (no matter how small) can cause problems!**
    * Law of diminishing returns[3]
- Gustafson's Law
  - An Alternative (optimistic) view to Amdahl's Law
  - Observations:

[3]The law of diminishing returns (also law of diminishing marginal returns or law of increasing relative cost) states that in all productive processes, adding more of one factor of production, while holding all others constant, will at some point yield lower per-unit returns [Taken from Wikipedia]

        * Consider problem size
        * Run-time, not problem size, is constant
        * More processors allows to solve larger problems in the same time
        * Parallel part of a program scales with the problem size
    - Formula:
        * b: sequential part (no speedup)

$$T_1 = p(1-b)T_p + bT_p$$
$$S_p = p - b(p-1)$$

- Concurrency vs. Parallelism
    - Concurrency is:
        * A programming model
        * Programming via independently executing tasks
        * About structuring a program
        * A concurrent program does not have to be parallel
    - Parallelism:
        * About execution
        * Concurrent programming is suitable for parallelism

Add views of different architectures

- Concerns in Parallel programming
    - Expressing parallelism
        * Work partitioning (Split up work for a single program into parallel tasks). Can be done:
            · Manually (task parallelism): User explicitly expresses tasks
            · Automatically by the system (e.g. in data parallelism): User expresses an operation and the system takes care of how to split it up
        * Scheduling
            · Assign task to processors (usually done by the system)
            · goal: full utilization (no processor is ever idle)
    - Managing state (data)
        * Shared vs. distributed memory architectures (in the latter, data needs to be distributed)
        * Which parallel tasks access which data, and how (e.g. READ or WRITE access)
        * (Potentially) split up data. Ideal: each task exclusively accesses its own data
        * Depending on the application:
            · Tasks, then data
            · Data, then tasks

  - Controlling/Coordinating parallel tasks and data
    - Distributed data
      - No coordination (e.g. embarrassingly parallel)
      - Messages
    - Shared data: controlling concurrent access
      - Concurrent access may cause inconsistencies
      - Mutual exclusion to ensure data consistency

- Coarse vs. Fine Granularity
  - Fine granularity
    - More portable (can be executed in machines with more processors)
    - Better for scheduling
    - Parallel slackness (Expressed parallelism ¿¿ machine parallelism)
    - BUT: if scheduling overhead is comparable to a single task → overhead dominates
  - Guidelines:
    - As small as possible
    - but, significantly bigger than scheduling overhead; system designers strive to make overheads small
  - Coordinating tasks:
    - Enforcing ordering between tasks, e.g.:
      - Task X uses result of task A
      - Task X needs to wait for task A to finish
    - Example primitives:
      - *barrier*
      - *send()/receive()*
    - All tasks need to reach the barrier before they can proceed

4 Parallel programming models