# 01/25/22: (1.1 - 1.4) Intro to Parallel

* Office Hours: Zoom @ 2-3:30pm Wednesday
* Homeworks:
  + Can be submitted unlimited times (submit when each part is completed)
* Exams:
  + Completely open book
* Why Multiple Processors?
  + Moore’s Law isn’t really a law and doesn’t hold up to 50% overtime
  + The power constraints and heat generated are too much
  + One big core is not as good as multiple lower-frequency cores
  + The case for processors:
    - We can exploit types of parallelism
      * Parallelism among instructions
      * Parallelism among tasks/threads/processes
      * Processing different data at the same time
    - Reduces power consumption
    - Effective way to hide memory latency
    - Simpler cores = easier to design/test = higher yield
* Task-Parallelism:
  + Partition tasks to be carried out to the cores
  + Ex: Core A does Task 1, Core B does Task 2
* Data-Parallelism:
  + Partition data among all the cores
  + Ex: Core A does 50% of the data on Task 1 & 2, Core B does the two tasks on the rest of the data
* How parallelism works from a communication perspective:
  + **Communication** → How coordination actually takes place
  + **Load Balancing** → Making sure each core gets approximately same amount of work
  + **Synchronization** → Each core will wait until all cores are ready to go
* **Superscalar** → More execution units
  + Number of execution units is not enough to figure out how many threads can run

# 01/27/22: (2.1 - 2.2) Basic Parallel Hardware

### Von Neumann Background

* **Von Neumann Architecture:**
  + Executing single instruction at a time
  + Components:
    - **Main Memory** → Collection of locations
      * Each location can store instructions and data
      * Each location has address and contents
    - **CPU/Processor/Core**
      * Control Unit → Decides which instructions are executed
      * Data Path → Executes the instructions decided by control unit
    - **Bus** → Traditional Interconnect between Main Memory and Core
      * Data is *fetched* from memory
* **Registers** → Fast storage in CPU that stores data about executing program
  + **Program Counter** → Special register that stores address of next instruction set
* **Von Neumann Bottleneck:**
  + Separation between CPU & Memory
  + **The constraint in a VNM is the speed of the interconnect**

### Processes, Multitasking, Threads

* **Operating System (OS) →** Manages software and abstracts the hardware
* **Process →** An instance of a program
  + *Call Stack* → Keeps track of active functions
  + *Heap* → Memory allocated to process
* **Multitasking OS** → Possible by giving each process a time slice
* **Threads** → Run inside a process, lightweight
  + *Fork → Create new thread*
  + *Join → Collapse thread back into process*

### Caching

* **Cache** → Collection of memory locations that can be accessed quicker than the main memory
  + **Blocks/Lines** → Blocks of data and instructions
    - 8-16x more data than a single location
  + **Levels** → Division of CPU Cache into individual levels
  + **‘Hit/Miss’** → Info is or is not available in the cache
  + Inconsistency → Happens when data in cache and main memory are different
    - Solution: Write-Through caches can write to main memory when cache is updated
    - Solution 2: Write-Back caches can set update cache line to ‘dirty’ and write the ‘dirty’ line to memory
* **CPU Cache** → Collection that the CPU can access quicker than main memory
* **Locality** → Principle that access of one location is followed by access of nearby location
  + **Spatial Locality** → Nearby location
  + **Temporal Locality** → Near future access to location

# 02/01-03/22: (2.3) Advanced Parallel Hardware

* **Flynn’s Taxonomy** → Classifies parallel computers based on # of instruction streams and # of data streams it can manage at one time
  + **SISD** (Single instruction, single data stream) → Classic Von Neumann Machine
* **Shared Memory Systems** → Cores can share access to memory, cores coordinate work by modifying shared memory
* **Distributed Memory Systems** → Each core has its own memory
* **Vector →** Array of 4-256 elements (each 64-bit)
* **Scalar** → A single value

### SIMD Systems

* **Parallel Systems → Single Instruction Stream, Multiple Data Stream**
  + Applies one set of instructions on all the data streams
  + Components:
    - Single Control Unit, Multiple Datapaths
* This is essentially a system that performs **data-parallelism**
* **Vector Processors →** Main use of SIMD Systems
  + A processor that operates on *vectors*, instead of *scalars*
  + Pro: Very fast and easy to use
* **Graphics Processing Units (GPUs)**
  + **Graphics Processing Pipeline** → Converts internal representation of object into array of pixels
  + **Shader Functions** → Programmable stages of pipeline that converts internal object representation into something that can appear on screen
    - Optimize performance via SIMD parallelism b/c shader functions apply to each element in same way
  + Multithreading on Hardware → Avoids stalls in data movement
  + **NOT** pure SIMD systems, current GPUs can run more than one instruction stream on a single core

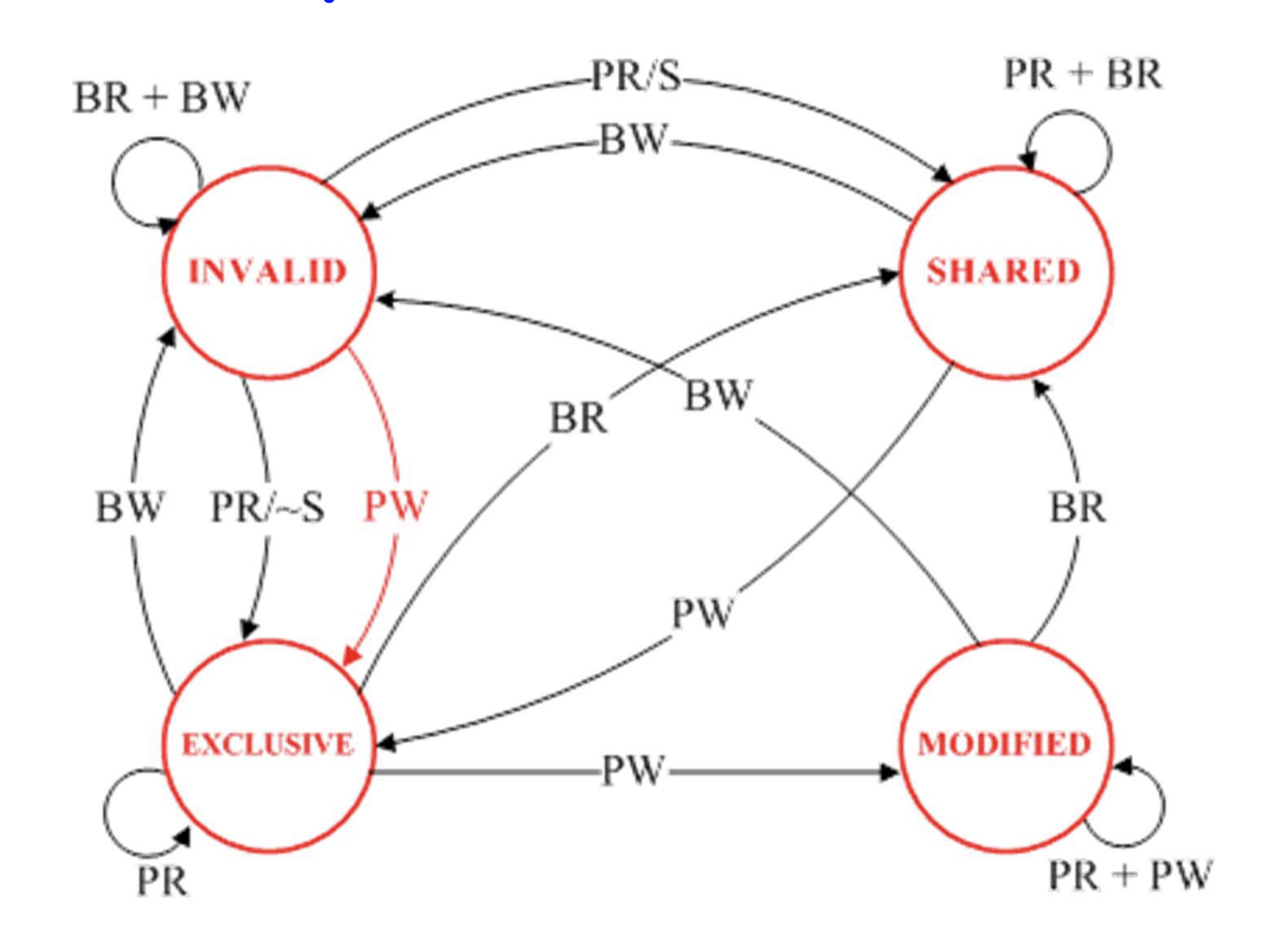
### MIMD (Most generic, can run anything)

* **Multiple Instruction, Multiple Data**
* Construction:
  + Collection of fully independent cores
* **Asynchronous** → Processors can operate at their own pace
  + No global clock, **no synchronization built-in**
* Types of Systems:
  + Shared-Memory System
    - Use multicore processors
      * Each processor can access each memory location
    - Private level 1 cache for each core
    - May be **Uniform/Non Uniform Memory Access**
      * Nonuniform may be due to location, like physical distance of memory from core
  + Distributed Memory System
    - Typically called **Clusters**

### Interconnection Networks

* The method of interconnection is the backbone of communication, communication is VERY expensive
* Why are there shared-memory and distributed-memory:
  + It has to be fast
  + In shared-memory, you can have 8, 16, 32 cores MAX
  + In distributed-memory, you can have 1000s of cores
* Shared-Memory Interconnects
  + Bus → Original method of interconnection
    - Collection of parallel communication wires
    - Scales poorly due to bus contention and noise impact
    - Low Cost and Flexibility
  + Switched Interconnect → Modern Interconnection method
    - Uses switches to control the routing of data among connected devices
    - Crossbar → Simple example or switched interconnect
* Distributed-Memory Interconnects
  + Must be scalable → Because it requires many machines
  + Direct Interconnect → Each switch is connected to processor-memory pair
    - All switches are connected to one another
    - Switches allow for the routing to a specific processor-memory pair
  + Indirect Interconnect
    - Switches are not directly connected to the processor
* Definitions related to interconnection networks
  + Latency → Time from start of data transmission to moment of data reception
  + Bandwidth → Rate at which the destination receives data
  + Message Tx Time → latency + (length\_of\_message/bandwidth)

### Cache Coherence

* Problem: What happens when two separate cores are trying to write to the same location in main memory?
* Cache coherence takes place above the shared memory cache but below L1 and L2
* Solutions
  + Snooping Cache Coherence
    - Cores must share a bus
    - Every core is snooping the bus, and if a core sends an update, all other cores will see that update data and do it to their cache as well
      * Interconnect does not HAVE to be a bus
      * Snooping work in write-through and write-back cache
  + Directory-Based Cache Coherence
    - Create a data structure called a ‘directory’
      * Stores a list of all the locations where a variable is saved
      * Coordinates between the caches in order to invalidate old data and allowing the most recent request for an update come through
* Coherence Protocols
  + MESI Protocol
    - 
* False Sharing
  + When both cores are accessing different locations but the system thinks they are accessing the same one
  + Problem: Too many memory accesses → Cache Coherence
* What will the cache do when it’s time to update data:
  + Write-Through Cache
    - Update to main memory will update all the caches
  + Write-Back Cache
    - Only that block will be updated
* Multicore example:
  + Cascade Lake (AKA: Intel Xeon)
    - L1D Cache: 32KB/core
      * 8-way set associative
      * 64 sets, 64B line size
    - L2 Cache
      * 1 MB/Core
      * 64B Line size
    - L3 Cache
      * 1.375MB/Core
      * 11-way set associative
  + Notes:
    - Bigger cache size = More time to access
    - More associative cache = More time to access
* Computer Example
  + FUGAKU → #1 best supercomputer as of November 2021
    - 7 million cores
* Trends:
  + More cores per chip
  + Non-Bus interconnect
  + NUMA/NUCA

# 02/08/22: (2.4.0-2.4.4) Parallel Software Basics

* The Basics:
  + Shared Memory Systems:
    - Single process and *fork* threads
    - Threads carry out tasks
  + Distributed Memory Systems:
    - Multiple processes
    - Processes carry out tasks
  + Accelerators (GPUs):
    - Start a process with one or more threads
    - The thread launches tasks on GPU
    - GPU will perform data-parallelism
* Single Program Multiple Data (**SPMD**)
  + Single executable program forked into different processes/threads
* Writing Parallel Programs:
  + Divide the work among processes/threads
    - Load balancing
    - Communication needs to be low
  + Arrange for synchronization if needed
  + Arrange for communication between processes/threads
* Shared Memory Systems (Threads):
  + Dynamic Threads: Master thread waits for work, forks new threads, and when finished, they terminate
    - Con: Thread creation/deletion is time consuming
    - Pro: Efficient use of system resources
  + Static Threads: Pool of threads created are allocated work
    - Con: Waste of system resources potentially
    - Pro: Better performance
  + Nondeterminism → Threads may finish before/after each other each time you run
    - * Race conditions
      * Critical section
  + Busy-Waiting
    - Rank → Thread ID (Unique to each thread)
    - Thread waits in a loop to move forward
* Distributed Memory Systems (Processes):
  + Message passing
    - Rank → Basically the thread number
    - Necessary if the threads are solving the same problem or are dependent on each other
* **Threads share Virtual Memory**, while **Processes share Physical Memory**

# 02/10/22: (2.4.0-2.4.4) Software - Advanced

* Concurrency → At least two tasks are making progress @ the same time frame
  + Not NECESSARILY same time
  + Include techniques like time-slicing
* Parallelism → At least two tasks execute **literally at the same time**
  + Requires hardware w/ multiple processing units
* Parallelism + Concurrency = High Performance
  + BUT, 2 cores doesn’t mean 2x improvement
    - Memory, Synchronization, Load Balancing, etc.
* Amdahl’s Law:
  + How much speedup can one get from a parallelized task?
  + Maximum speedup = 1/(F + (1-F)/ P)
    - 1/F → Theoretical Speedup
    - F → Sequential portions of your code
    - P → Number of cores
  + His point:
    - Don’t invest in a ton of processors b/c a lot of code is sequential
    - Just focus on getting faster cores
  + Good thing → Let’s you find the upper bound speedup really quickly with simple math
  + Why he was right back then:
    - Not easy to find F nowadays
    - Programs used to be hella sequential, not anymore
    - He didn’t take into account synchronization, communication, OS, load balancing
* Compiled Language → Translate literally the entire thing
* Interpreted Language → Translate line-by-line
* DAG Model for Multithreading
  + Take a multithreading program and draw it as a DAG
  + **Vertex** → a unit of execution (instruction/step/function)
    - **Basic Block** → Assembly instructions where executing the first line ensures that the last one will also be executed → Basically, a block of code that will run each step FOR SURE
  + **Edge** → Dependency, for example, vertex A must execute vertex B
    - **Work** → Total time spent on all instructions
    - T\_p → Fastest possible execution time on P processors/cores
    - Work law: ***T\_p >= T\_1 / P***
    - **Span** → Longest path of dependence in the DAG (T\_infinity, how long it would take to execute with infinite number of cores)
      * Timing measure, length in terms of ***time***
    - **Span Law:** T\_p >= T\_infinity
    - Parallelism = Number of cores after which we will see no performance increase = T\_1 / T\_infinity
      * Theoretically: Ceiling is optimal
      * In practice: Estimate performance on both and weigh costs of getting an extra core

# 02/15/22: (2.4.0-2.4.4) Software - Advanced

* Programming Model
  + Languages and libraries that create an abstraction of the machine
  + Control
    - How is parallelism created?
    - How are dependencies enforced?
  + Data
    - Shared or private?
    - How is shared data accessed or private data communicated?
  + Synchronization
    - What ops can coordinate parallelism
    - What are the atomic ops
* Note → Hardware is heterogeneous (some cores slow down, others speed up)
  + Instruction/Task/Data level parallelism → How to best use hardware for this end-goal
* Where do we lose performance?
  + Overhead
    - Synchronization
    - Communication
  + Artificial Dependencies
    - **Note: Try as much as possible to reduce global variables**
    - Hard to find
    - May introduce bugs
    - A lot of work to remove
  + Contention over hardware resources
* Coherence:
  + Extra bandwidth
  + Latency due to the protocol
  + False Sharing → Hardware doesn’t understand variables
* Load Balancing
  + Everybody has to wait for the slowest thread
  + **More synchronization = Greater sensitivity to load imbalance**
  + At least try to reduce it, you don’t have to eliminate it
* How to parallelize an algorithm:
  + Task-level (Embarrassingly parallel)
    - Break application into tasks (offline)
    - Low scalability (you know the tasks ahead of time)
    - Note: Just make sure you have some basic load-balancing
  + Divide & Conquer
    - Problem can be divided into subproblems indefinitely
  + Pipeline
    - A series of ordered but independent computations need to be applied
    - Useful for:
      * Streaming workloads
      * Loops that are hard to parallelize
    - How to?
      * Split each loop into independent stages (S1, S2, S3)
      * Assign each stage to a thread (Thread T1 does S1, etc.)
      * When a thread is done with some stage, start same stage for the next loop
    - Advantages:
      * Expose intra-loop parallelism
      * Locality increases for variables used across stages
  + Iterations (loops)
  + Client-Server (AKA Repository Model)
    - Use-case: You don’t know the program size when you start the program
    - Structure:
      * 1 thread (controller) will manage the repository of work
      * Work in repository is taken by threads and computed
        + Done via asynchronous function calls from repository
  + Geometric
  + Hybrid
* Conclusions:
  + Concurrency & Parallelism are NOT the same thing
  + Knowing hardware = Helps generate better task dependency graph = Helps you reason parallelism in your code

# 02/17/22: (2.6-2.6.4) Performance Analysis

* Defining Performance
  + IFF Performance = Speed:
    - Calculate speed performance on Machine X:
      * Performance = 1 / ExecutionTime
    - Compare speed performance on multiple Machines:
      * “Machine X is n times faster than Machine Y”:
      * n = Performance\_X / Performance\_Y
    - Calculate Speed up & Efficiency
      * Speedup = T\_serial / T\_parallel
      * Efficiency = Speedup/p
        + p = Number of cores
        + T\_serial = Serial run-time
        + T\_parallel = Parallel run-time
  + Scalability
    - Ability of a system/software to handle growing amount of work efficiently
      * Strongly scalable → Increasing number of processes/threads w/o increasing problem problem size
  + Sources of Overhead:
    - Creating threads/processes
    - Synchronization
    - Load imbalance
    - Communication
    - Extra computation
    - Memory access
* How to take time?
  + Types of timings:
    - Elapsed Time (wall-clock time)
      * Counts everything
      * Useful, but not good for comparisons
    - CPU time
      * Doesn’t count I/O or disk
      * Can be broken up into system and user time
    - **User CPU Time**
      * Time spent executing the lines of code that are “in” our program Taking timings
  + Metrics:
    - Response time (Execution time)
      * Time between start & end of task
    - Throughput
      * Total amount of work done in a given time
  + Execution time for sequential program
  + Clock Time:
    - Execution\_time = instruction\_count \* cycles\_per\_instruction \* cycle\_time
      * Variables:
        + cycle\_time = 1/frequency
      * Problems:
        + Each instruction doesn’t have same cycle count
        + Does not factor in memory
    - 10 = k \* 1/4GHZ
    - K = 40GHz
    - 6 = 48 \* 1/?GHZ
    - 8GHz
  + CPI
    - 2 implementations of the same instructions et architecture (ISA)
    - Machine A → 250ps clock time
      * CPI = 2.0
      * Execution time = k \* 2.0 \* 250
    - Machine B → 500ps clock time
      * CPI = 1.2
      * Execution time = k \* 1.2 \* 500
  + # instructions
* Pitfalls in timing in Parallel Machines
  + Multithreaded programs
    - Total # of instruction executed may be different across different runs
    - System-level code accounts for significant fraction of execution time
  + Program does not run in vacuum
    - OS adds variability
    - Multi-programming/Multi-threading is common
* How to check the performance of parallel machines?
  + Benchmarks
    - Performance best determined by running a real application
      * Use programs typical of expected workload
      * Companies have agreed on a set of real program and inputs
    - Current benchmarks:
      * Parallel: PARSEC, Rodinia, SPLASH-2
      * Sequential benchmarks: SPEC (System performance evaluation cooperative)
    - Purpose: Help designer explore architectural designs
      * Identify bottlenecks
      * Compare different systems
      * Conduct performance prediction
* Performance Evaluation = Very important
  + Capture some aspects but not great: MIPS, #instructions, #cycles, frequency
  + **Execution time is what really matters**

# 02/24/22: (3.1) MPI - I

* MPI - Message Passing Interface
  + Target: Distributed memory systems
  + Goal: Reduce messages but increase concurrency
  + Wanted quick adoption → MPI is a bunch of libraries on top of C, C++, Fortran
  + Processes → Assumption that each process is sequential
* Need to add <mpi.h> header file

# 03/01/22: (3.1) MPI - II

* MPI

FAQs:

* Can two processes have the same memory/operate in the same memory?
  + YES! They can’t write to the same memory but they can share a library that is abstracted in virtual memory from physical memory
* All processes in communicator MUST call the same collective function
  + Args passed by each process must

================== END OF MATERIAL FOR MIDTERM ==================

# 03/22/22: (3.1) MPI - III

* MPI\_Gather
  + Collects all of the components of the vector onto process *dest* → rank order
* Print a distributed vector
  + void Print\_vector()
* How to synchronize processes
  + MPI\_Barrier
    - Ensures that no process will return from calling until ALL processes have reached it