3-5 Parallel Hardware

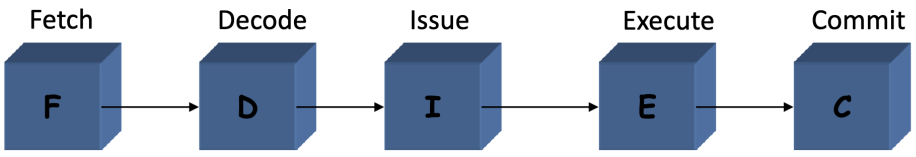
Lecture 3 Parallel Hardware: Basics

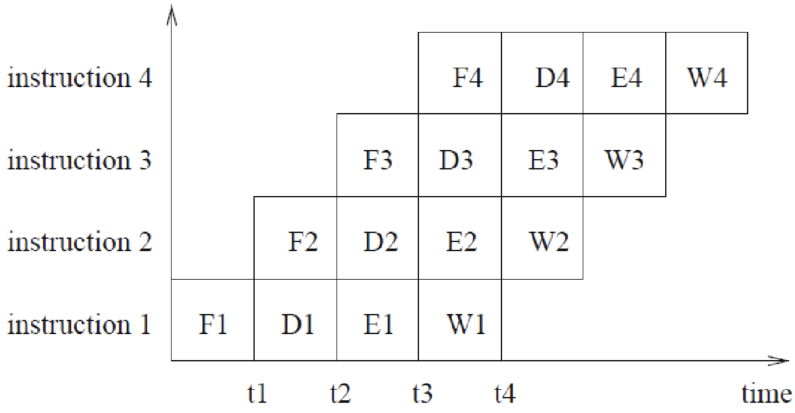
First Generation (1970s)

- Single Cycle Implementation

Second Generation (1980s)

- Pipelining
  - the hardware divided into stages
  - temporal parallelism





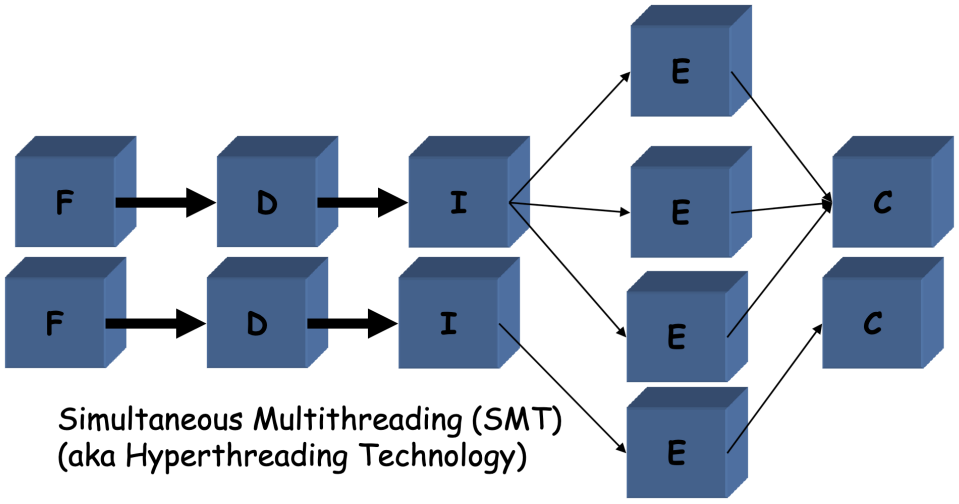
- Minimum CPI (Cycles Per Instruction) = 1

Third Generation (1990s)

- ILP (Instruction Level Parallelism)
- Spatial parallelism
- Executing several instructions at the same time is called superscalar capability.
- performance = several instructions per cycle (IPC)
- Speculative Execution (prediction of branch direction) is introduced to make the best use of superscalar capability → This can make some instructions execute out-of-order!!

Fourth Generation (2000s)

Simultaneous Multithreading (SMT) (aka Hyperthreading Technology)



Double (or triple or ...) some resources in the pipeline to host several programs at the same time. This allows better use of the execution resources.

Some definitions

An operating system “process”

- an instance of a computer program that is being executed.

Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running → time slice

Threading

- Threads are contained within processes.
- They allow programmers to divide their programs into (more or less) independent tasks.

Conclusions

The hardware evolution, driven by Moore’s law, was geared toward two things:

- Exploiting parallelism
- Dealing with memory (latency, capacity)

Lecture 4-5 Parallel Hardware: Advanced

Computer Technology ... Historically

- Memory: 64x size improvement in last decade.
- Processor: 100x performance in last decade, BUT we are hitting a wall.
- Disk: 250x size in last decade.

Memory Wall Is Here to Stay

- Memory access is still a big problem in parallel machines.

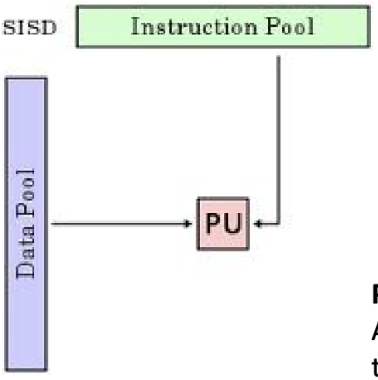
- Cache coherence (as we will see later) also has large negative effect on performance.

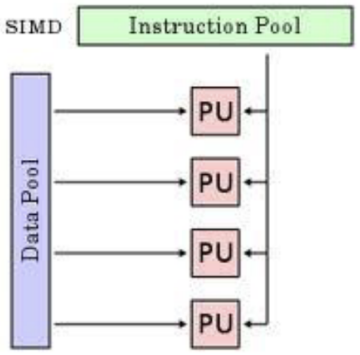
Processor-Memory Performance Gap (grows 50% / year)

Most of the single core performance loss is on the memory system!

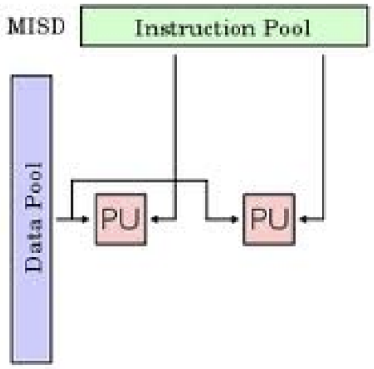
Flynn's Taxonomy

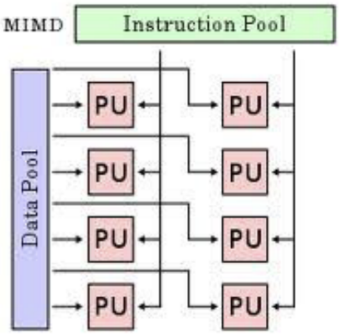
classic von Neumann SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream





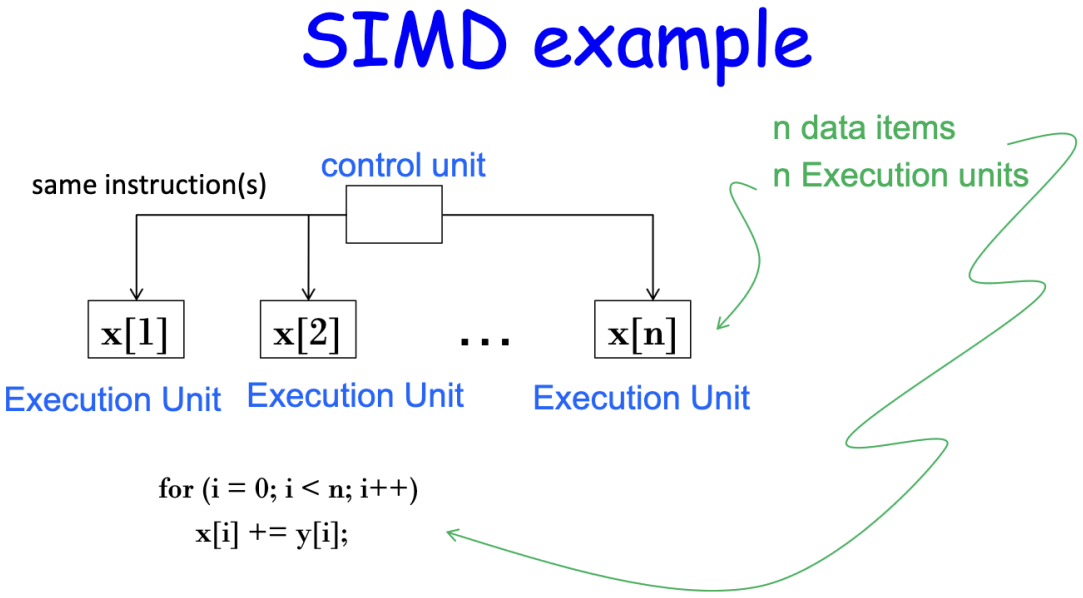
PU = Processing Unit
Any piece of hardware
that can do computation.





SIMD

- Applies the same instruction (or group of instructions) to multiple data items at once.
- data parallelism.
- Example:
  - GPUs
  - vector processors



Naming convention: Execution units are most often called **Arithmetic and Logic Units (ALUs)**

What if we don't have as many ALUs as data items? Divide the work and process iteratively.

SIMD drawbacks:

- All ALUs are required to execute the same instruction(s) or remain idle.
- In classic design, they must also operate synchronously (i.e. together at the same time).
- Efficient for large data parallel problems, but not other types of more complex problems.

SIMD Example:

Vector processors

- Processors execute instructions where operands are vectors instead of individual data elements or scalars.
- This needs:
  - Vector registers: Capable of storing a vector of operands and operating simultaneously on their contents.
  - Vectorized execution units: The same operation is applied to each element in the vector (or pairs of elements).
- Pros

    - Fast. Easy to use.
    - Vectorizing compilers are good at identifying code to exploit / code that cannot be vectorized.
    - Makes the best use of memory bandwidth.
- Cons
    - They don't handle irregular data structures.
    - A very finite limit to their ability to handle ever larger problems. (scalability)
        - The machine has a finite number of vectorized execution units.
        - The machine has a finite number of vectorized registers.

MIMD

- Supports multiple simultaneous instruction streams operating on multiple data streams.
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.
- Example: multicore processors, multiprocessor systems

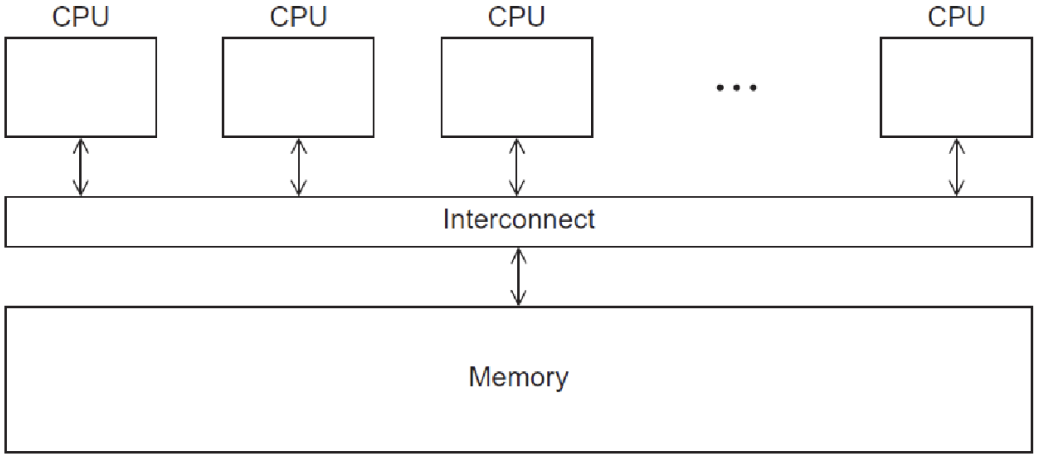
Let's classify MIMD based on how memory is designed.

Shared Memory System

A collection of autonomous processors/cores is connected to a memory system via an interconnection network.

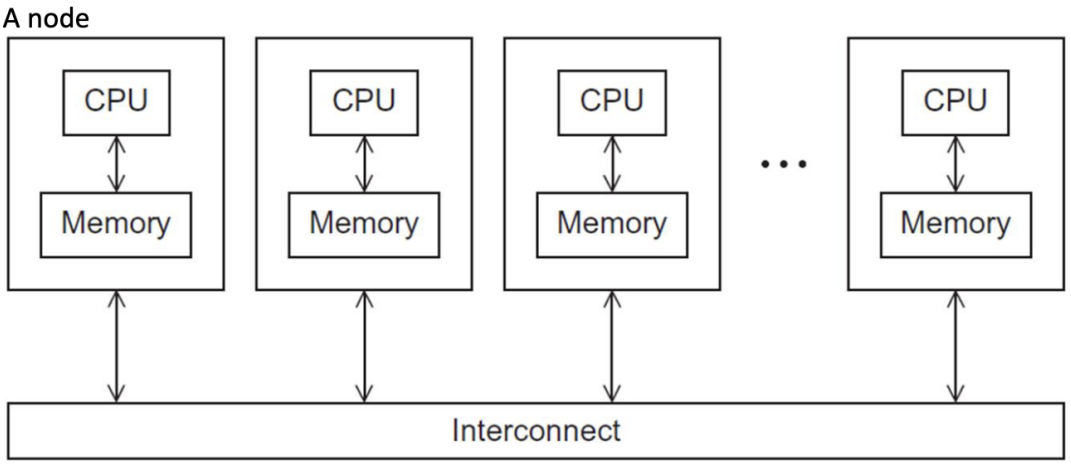
Each processor/core can access each memory location.

The processors/cores usually communicate implicitly by accessing data shared in memory.

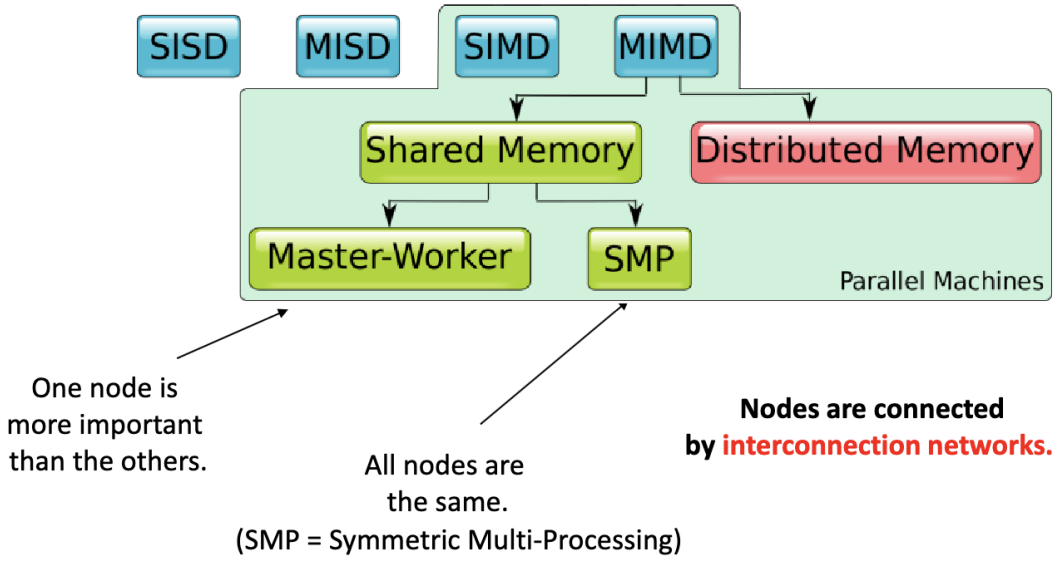


Distributed Memory System

A collection (cluster) of nodes, connected by an interconnection network



Summarization



Interconnection networks

Affects performance of both distributed and shared memory systems.

- Communication is very expensive.

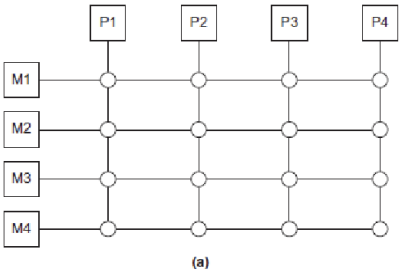
Shared memory interconnects

- Bus interconnect
  - A collection of parallel communication wires together with some hardware that controls access to the bus.
- Switched interconnect
  - Uses switches to control the routing of data among the connected devices.
  - Those switches are connected by wire forming network of some topology.
  - Example: Crossbar

A Crossbar

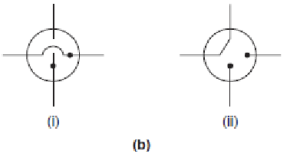
(a)

A crossbar switched network connecting 4 processors (P_i) and 4 memory modules (M_j)

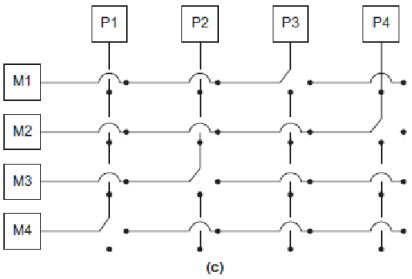


(b)

Configuration of internal switches in a crossbar



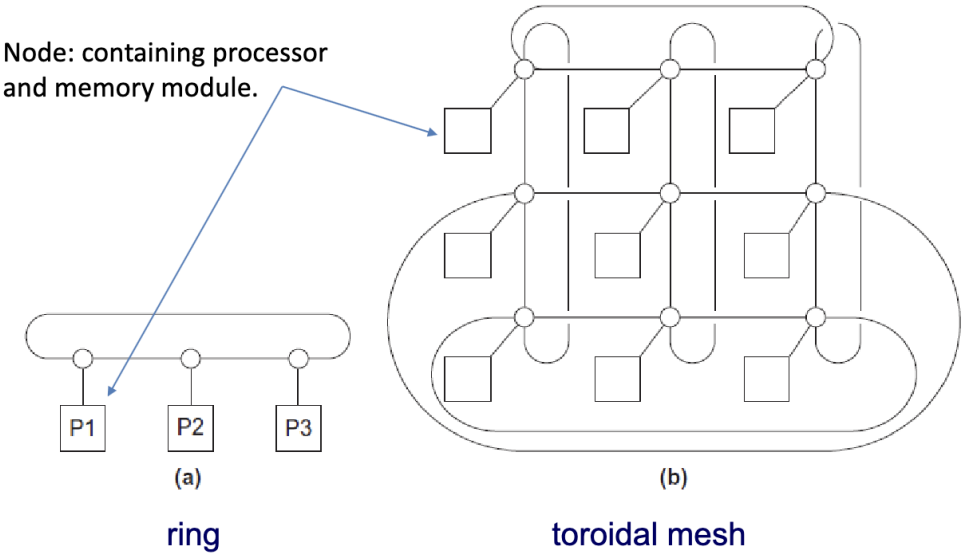
(c) Simultaneous memory accesses by the processors



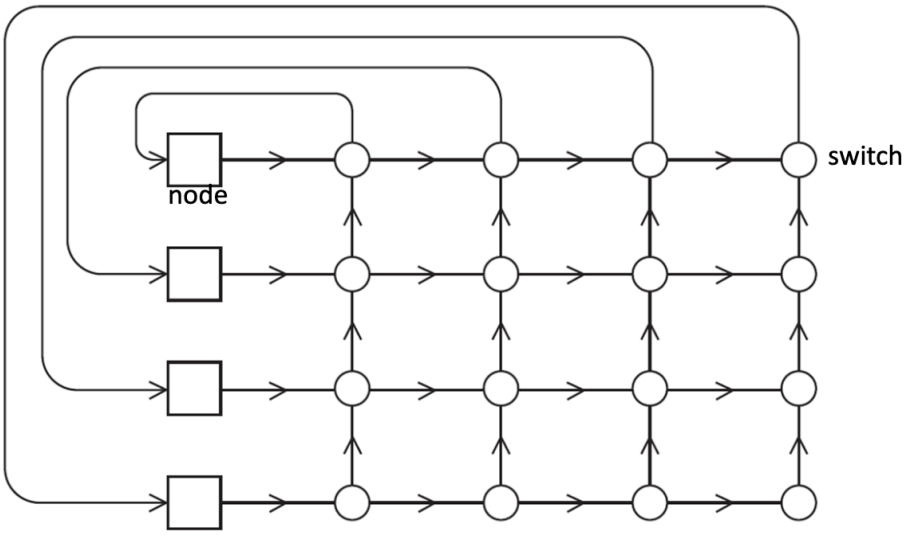
Distributed memory interconnects

- Direct interconnect: Each switch is directly connected to a processor-memory pair, and the switches are connected to each other.

Direct Interconnect: Examples



- Indirect Interconnect: Switches may not be directly connected to a processor.



Some Definitions Related to Interconnection Networks

- Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.
- Latency: The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.
- Bandwidth: The rate at which the destination receives data after it has started receiving the first byte.
- Message transmission time = l + n / b
  - l = latency (seconds); n = length of message (bytes); b = bandwidth (bytes per second)

Between the processor/core and the memory modules, there is one or more levels of caches.

This introduces a big challenge.

Cache coherence

- Programmers have no control over caches and when they get updated.
- But programmers can write cache friendly code.

Snooping Cache Coherence (no longer used)

- The cores must share a bus. Signals can be "seen" by all cores connected to the bus.
- When a cache updates its copy of x, other cores will "see" this and mark its copy of x invalid.

Directory Based Cache Coherence

- Uses a data structure called a directory that stores the status of each cache line.

- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Cache Coherence Protocols: Write invalidate + Write update

The trend now is:

- More cores per chip
- More heterogeneity
- Non-bus interconnect
- NUMA and NUCA (Non-Uniform Memory / Cache Access)

Communication and memory access are the two most expensive operations, NOT computations.