Computer Science 146
Computer Architecture

Fall 2019

Harvard University

Instructor: Prof. David Brooks

dbrooks@eecs.harvard.edu

Lecture 19: Multiprocessors

Lecture Outline

- Multiprocessors
  - Computation taxonomy
  - Interconnection overview
  - Shared Memory design criteria
    - Cache coherence
- Very broad overview
- H&P: Chapter 6, mostly 6.1-6.9, 8.5
- Next time: More Multiprocessors +Multithreading

Parallel Processing

- Multiple processors working cooperatively on problems: *not* the same as multiprogramming
- Goals/Motivation
  - Performance: limits of uniprocessors
    - ILP (branch prediction, RAW dependencies, memory)
  - Cost Efficiency: build big systems with commodity parts (uniprocessors)
  - Scalability: just add more processors to get more performance
  - Fault tolerance: One processor fails you still can continue processing

Parallel Processing

- Sounds great, but…
- As usual, software is the bottleneck
  - Difficult to parallelize applications
    - Compiler parallelization is hard (huge research efforts in 80s/90s)
    - By-hand parallelization is harder/expensive/error-prone
  - Difficult to make parallel applications run fast
    - Communication is expensive
    - Makes first task even harder
- Inherent problems
  - Insufficient parallelism in some apps
  - Long-latency remote communication

Recall Amdahl's Law

$$\text{Speedup}_{\text{Overall}} = \frac{1}{\left((1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}\right)}$$

- Example: Achieve speedup of 80x using 100 processors
  - $80 = 1/[\text{Frac}_{\text{parallel}}/100+1-\text{Frac}_{\text{parallel}}]$
  - $\text{Frac}_{\text{parallel}}=0.9975$ => only .25% of the work can be serial!
- Assumes linear speedup
  - Superlinear speedup is possible in some cases
  - Increased memory + cache with increased processor count

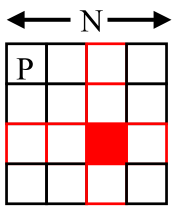
Parallel Application Domains

- Traditional Parallel Programs
- True parallelism in one job
  - Regular loop structures
  - Data usually tightly shared
  - Automatic parallelization
  - Called "data-level parallelism"
  - Can often exploit vectors as well
- Workloads
  - Scientific simulation codes (FFT, weather, fluid dynamics)
  - Dominant market segment in the 1980s

Parallel Program Example: Matrix Multiply

- Parameters
  - Size of matrix: (N*N)
  - P processors: $N/P^{1/2} * N/P^{1/2}$ blocks
  - Cij needs Aik, Bkj k=0 to n-1
- Growth Functions
  - Computation grows as $f(N^3)$
  - Computation per processor: $f(N^3/P)$
  - Data size: $f(N^2)$
  - Data size per processor: $f(N^2/P)$
  - Communication: $f(N^2/P^{1/2})$
  - Computation/Communication: $f(N/P^{1/2})$

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



Application Domains

- Parallel independent but similar tasks
  - Irregular control structures
  - Loosely shared data locked at different granularities
  - Programmer defines and fine tunes parallelism
  - Cannot exploit vectors
  - “Thread-level Parallelism” (throughput is important)
- Workloads
  - Transaction Processing, databases, web-servers
  - Dominant MP market segment today

Database Application Example

- Bank Database
- Parameters:
  - D = number of accounts
  - P = number of processors in server
  - N = number of ATMs (parallel transactions)
- Growth Functions
  - Computation: f(N)
  - Computation per processor: F(N/P)
  - Communication? *Lock* records while changing them
  - Communication: f(N)
  - Computation/communication: f(1)
  - No serial computation

Computation Taxonomy

- Proposed by Flynn (1966)
- Dimensions:
  - Instruction streams: single (SI) or multiple (MI)
  - Data streams: single (SD) or multiple (MD)
- Cross-product:
  - SISD: uniprocessors (chapters 3-4)
  - SIMD: multimedia/vector processors (MMX, HP-MAX)
  - MISD: no real examples
  - MIMD: multiprocessors + multicomputers (ch. 6)

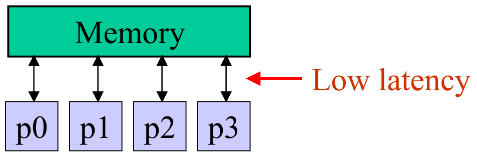
SIMD vs. MIMD

- Can you think of a commercial SIMD system?
- MP vs. SIMD
  - Programming model flexibility
    - Could simulate vectors with MIMD but not the other way
    - Dominant market segment cannot use vectors
  - Cost effectiveness
    - Commodity Parts: high volume (cheap) components
    - MPs can be made up of many uniprocessors
    - Allows easy scalability from small to large
  - Vectors making some noise lately (Berkeley, NEC)

Taxonomy of Parallel (MIMD) Processors

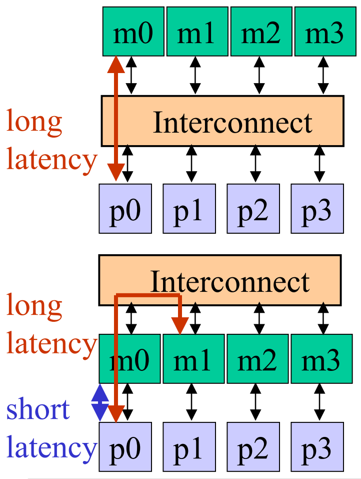
- Center on organization of main memory
  - Shared vs. Distributed
- Appearance of memory to *hardware*
  - Q1: Memory access latency uniform?
  - Shared (UMA): yes, doesn't matter where data goes
  - Distributed (NUMA): no, makes a big difference
- Appearance of memory to *software*
  - Q2: Can processors communicate directly via memory?
  - Shared (shared memory): yes, communicate via load/store
  - Distributed (message passing): no, communicate via messages
- Dimensions are orthogonal
  - e.g. DSM: (physically) distributed, (logically) shared memory

UMA vs. NUMA: Why it matters



- Ideal model:
  - Perfect (single-cycle) memory latency
  - Perfect (infinite) memory bandwidth
- Real systems:
  - Latencies are long and grow with system size
  - Bandwidth is limited
    - Add memory banks, interconnect to hook up (latency goes up)

UMA vs. NUMA: Why it matters

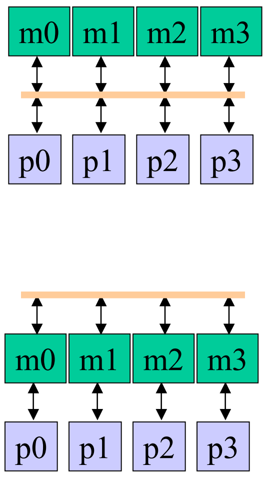


- UMA: uniform memory access
  - From p0 same latency to m0-m3
  - Data placement doesn't matter
  - Latency worse as system scales
  - Interconnect contention restricts bandwidth
  - Small multiprocessors only
- NUMA: non-uniform memory access
  - From p0 faster to m0 than m1-m3
  - Low latency to local memory helps performance
  - Data placement important (software!)
  - Less contention => more scalable
  - Large multiprocessor systems

Interconnection Networks

- Need something to connect those processors/memories
  - Direct: endpoints connect directly
  - Indirect: endpoints connect via switches/routers
- Interconnect issues
  - Latency: average latency more important than max
  - Bandwidth: per processor
  - Cost: #wires, #switches, #ports per switch
  - Scalability: how latency, bandwidth, cost grow with processors
- Interconnect *topology* primarily concerns architects

Interconnect 1: Bus



- Direct Interconnect Style

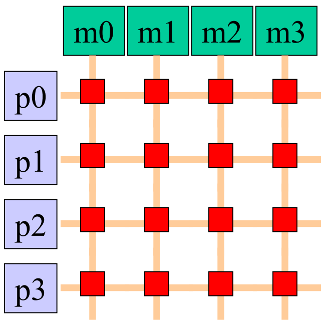
+ Cost: f(1) wires

+ Latency: f(1)

- Bandwidth: Not Scalable, f(1/P)
  - Only works in small systems (<=4)

+ Only performs *ordered broadcast*
  - No other protocols work

Interconnect 2: Crossbar Switch



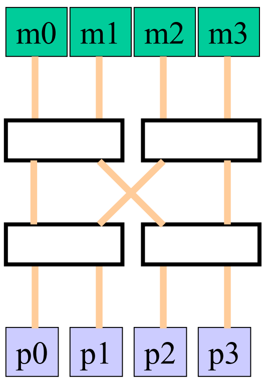
- Indirect Interconnect
  – Switch implemented as big MUXes

+ Latency: f(1)

+ Bandwidth: f(1)

- Cost:
  - f(2P) wires
  - F(P^2) switches
  - 4 wires per switch

Interconnect 3: Multistage Network



- Indirect Interconnect
  – Routing done by address decoding
  – *k*: switch arity (#inputs/#outputs per switch)
  – *d:* number of network stages = $\log_k$ P

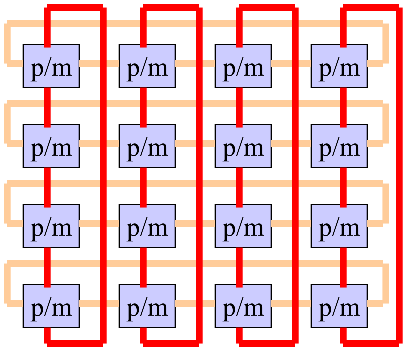
+ Cost
  - f(d*P/k) switches
  - f(P*d) wires
  - f(k) wires per switch

+ Latency: f(d)

+ Bandwidth: f(1)

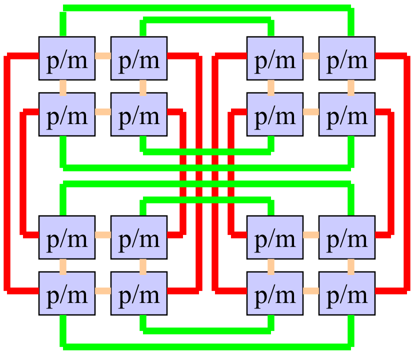
- Commonly used in large UMA systems

Interconnect 4: 2D Torus



- Direct Interconnect
- + Cost
  - f(2P) wires
  - 4 wires per switch
- + Latency: f($P^{1/2}$)
- + Bandwidth: f(1)
- Good scalability (widely used)
  - Variants: 1D (ring), 3D, mesh (no wraparound)

Interconnect 5: Hypercube



- Direct Interconnect
  - K: arity (#nodes per dimension)
  - D: dimension =$\log_k$ P
- + Latency: f(d)
- + Bandwidth: f(k*d)
- - Cost
  - F(k*d*P) wires
  - F(k*d) wires per switch
- Good scalability, expensive switches
  - new design needed for more nodes

Interconnect Routing

- *Store-and-Forward* Routing
  - Switch buffers entire message before passing it on
  - Latency = [(message length / bandwidth) + fixed switch overhead] * #hops
- *Wormhole* Routing
  - Pipeline message through interconnect
  - Switch passes message on before completely arrives
  - Latency = (message length / bandwidth) + (fixed switch overhead * #hops)
  - No buffering needed at switch
  - Latency (relative) independent of number of intermediate hops

Shared Memory vs. Message Passing

- MIMD (appearance of memory to software)
- *Message Passing* (multicomputers)
  - Each processor has its own address space
  - Processors send and receive messages to and from each other
  - Communication patterns explicit and precise
  - Explicit messaging forces programmer to optimize this
  - Used for scientific codes (explicit communication)
  - Message passing systems: PVM, MPI, OpenMP
  - Simple Hardware
  - Difficult programming Model

Shared Memory vs. Message Passing

- *Shared Memory* (multiprocessors)
  - One shared address space
  - Processors use conventional load/stores to access shared data
  - Communication can be complex/dynamic
  - Simpler programming model (compatible with uniprocessors)
  - Hardware controlled caching is useful to reduce latency + contention
  - Has drawbacks
    - Synchronization (discussed later)
    - More complex hardware needed

Parallel Systems (80s and 90s)

Machine	**Communication**	**Interconnect**	**#cpus**	**Remote latency (us)**
SPARCcenter	Shared memory	Bus	<=20	1
SGI Challenge	Shared memory	Bus	<=32	1
CRAY T3D	Shared memory	3D Torus	64-1024	1
Convex SPP	Shared memory	X-bar/ring	8-64	2
KSR-1	Shared memory	Bus/ring	32	2-6
TMC CM-5	Messages	Fat tree	64-1024	10
Intel Paragon	Messages	2D mesh	32-2048	10-30
IBM SP-2	Messages	Multistage	32-256	30-100

- Trend towards shared memory systems

Multiprocessor Trends

- Shared Memory
  - Easier, more dynamic programming model
  - Can do more to optimize the hardware
- Small-to-medium size UMA systems (2-8 processors)
  - Processor + memory + switch on single board (4x pentium)
  - Single-chip multiprocessors (POWER4)
  - Commodity parts soon – glueless MP systems
- Larger NUMAs built from smaller UMAs
  - Use commodity small UMAs with commodity interconnects (ethernet, myrinet)
  - NUMA clusters

Major issues for Shared Memory

- Cache coherence
  - "Common Sense"
    - P1 Read[X] => P1 Write[X] => P1 Read[X] will return X
    - P1 Write[X] => P2 Read[X] => will return value written by P1
    - P1 Write[X] => P2 Write[X] => Serialized (all processor see the writes in the same order)
- Synchronization
  - Atomic read/write operations
- Memory consistency Model
  - *When* will a written value be seen?
  - P1 Write[X] (10ps later) P2 Read[X] what happens?
- These are not issues for message passing systems
  - Why?

Cache Coherence

- Benefits of coherent caches in parallel systems?
  - Migration and Replication of shared data
  - Migration: data moved locally lowers latency + main memory bandwidth
  - Replication: data being simultaneously read can be replicated to reduce latency + contention
- Problem: sharing of writeable data

Processor 0	**Processor 1**	**Correct value of A in:**
		Memory
Read A		Memory, p0 cache
	Read a	Memory, p0 cache, p1 cache
Write A		P0 cache, memory (if write-through)
	Read A	P1 gets stale value on hit

Solutions to Cache Coherence

- No caches
  - Not likely :)
- Make shared data non-cacheable
  - Simple software solution
  - low performance if lots of shared data
- Software flush at strategic times
  - Relatively simple, but could be costly (frequent syncs)
- Hardware cache coherence
  - Make memory and caches coherent/consistent with each other

HW Coherence Protocols

- Absolute coherence
  - All copies of each block have same data at all times
  - A little bit overkill…
- Need *appearance* of absolute coherence
  - Temporary incoherence is ok
    - Similar to write-back cache
    - As long as all loads get their correct value
- Coherence protocol: FSM that runs at every cache
  - Invalidate protocols: invalidate copies in other caches
  - Update protocols: update copies in other caches
  - Snooping vs. Directory based protocols (HW implementation)
  - Memory is always updated

Write Invalidate

- Much more common for most systems
- Mechanics
  - Broadcast address of cache line to invalidate
  - All processor snoop until they see it, then invalidate if in local cache
  - Same policy can be used to service cache misses in write-back caches

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidation miss for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Write Update (Broadcast)

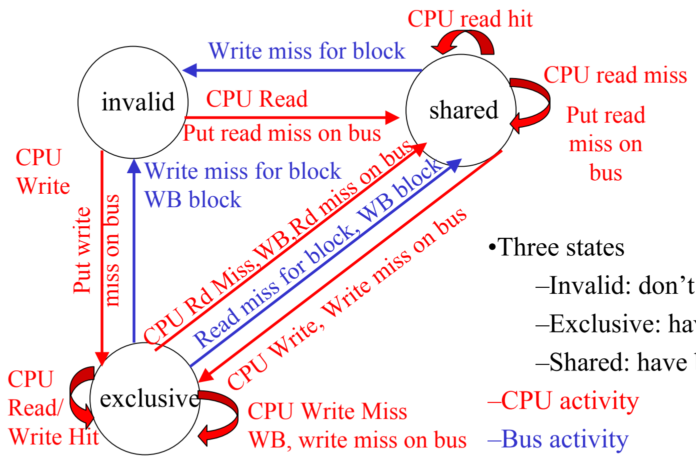
Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Write Broadcast for X	1	1	1
CPU B reads X		1	1	1

- Bandwidth requirements are excessive
  - Can reduce by checking if word is shared (also can reduce write-invalidate traffic)
- Comparison with Write invalidate
  - Multiple writes, no interveaning reads require multiple broadcasts
  - Multiple broadcasts needed for multiple word cache line writes (only 1 invalidate)
  - Advantage: other processors can get the data faster

Bus-based protocols (Snooping)

- Snooping
  - All caches see and react to *all* bus events
  - Protocol relies on global visibility of events (ordered broadcast)
- Events:
  - Processor (events from own processor)
    - Read (R), Write (W), Writeback (WB)
  - Bus Events (events from other processors)
    - Bus Read (BR), Bus Write (BW)

Three-State Invalidate Protocol



•Three states

–Invalid: don't have block

–Exclusive: have block and wrote it

–Shared: have block but only read it

–CPU activity

–Bus activity

Three State Invalidate Problems

- Real implementations are much more complicated
  - Assumed all operations are *atomic*
    - Multiphase operation can be done with no intervening ops
    - E.g. write miss detected, acquire bus, receive response
    - Even read misses are non-atomic with split transaction busses
    - Deadlock is a possibility, see Appendix I
  - Other Simplifications
    - Write hits/misses treated same
    - Don't distinguish between true sharing and clean in one cache
    - Other caches could supply data on misses to shared block

Next Lecture

- More multiprocessors
  - Example of FSM
  - Directory based systems
  - Synchronization
  - Consistency
- Multithreading
  - In Extra Readings – Intel’s paper “Hyper-threading Technology Architecture and Microarchitecture”