Multiprocessors and Multithreading

- why multiprocessors and/or multithreading?
  - and why is parallel processing difficult?
- types of MT/MP
- interconnection networks
- why caching shared memory is challenging
  - cache coherence, synchronization, and consistency

just the tip of the iceberg — take ECE 259/CPS 221 for more!

Readings

H+P

- chapter 3.5, 4
    - Will only loosely follow text

Recent Research Papers

- Power: A First Class Design Constraint
- SMT (Simultaneous Multithreading)
- Multiscalar
- Sun ROCK
- NVidia Tesla

Threads, Processes, Processors, etc.

some terminology to keep straight

- process
- thread
- processor (will use term “core” interchangeably)
- thread context
- multithreaded (MT) processor
- multiprocessor (MP) on 1 chip or on multiple chips

many issues are the same for MT and MP

- will discuss in terms of MPs, but will point out MT diffs

Why MT/MP?

Reason #1: Performance

- ILP is limited --> unicore performance is limited
- can’t even exploit all of the ILP we have
  - problems: branch prediction, cache misses, etc.
- it’s often easy to exploit thread level parallelism (TLP)
- can you think of programs with lots of TLP?

Reason #2: Power-efficiency

- Can use more cores, if cores at lower clock frequency
- Improved throughput (but perhaps worse latency)

More Reasons for MT/MP

Reason #3: Cost and cost effectiveness

- build big systems from *commodity parts* (ordinary cores/procs)
- enough transistors to make multithreaded cores
- enough transistors to make multicore chips

Reasons #4 and #5:

- smooth upgrade path (keep adding cores/processors)
- fault tolerance (one processor fails, still have P-1 working)

Most important (most cynical?) reason: We have no idea what else to do with so many transistors!

Why Parallel Processing Is Hard

in a word: *software*

- difficult to parallelize applications
  - compiler parallelization very hard (impossible holy grail?)
  - by-hand parallelization very hard (very error prone, not fun)
- difficult to make parallel applications run fast
  - communication very expensive (must be aware of it)
  - synchronization very complicated

IT'S THE SOFTWARE, STUPID!

Amdahl's Law Revisited

$\text{speedup} = 1/\,[\text{frac}_{\text{parallel}}/\text{speedup}_{\text{parallel}} + 1 - \text{frac}_{\text{parallel}}]$

- example
  - achieve speedup of 80 using 100 processors
  - $\Rightarrow 80 = 1\,/\,[\text{frac}_{\text{parallel}}/100 + 1 - \text{frac}_{\text{parallel}}]$
  - $\Rightarrow \text{frac}_{\text{parallel}} = 0.9975 \Rightarrow$ only 0.25% work can be serial!
- good application domains for parallel processing
  - problems where parallel parts scale faster than serial parts
  - e.g., $O(N^2)$ parallel vs. $O(N)$ serial
  - interesting programs require communication between parallel parts
  - problems where computation scales faster than communication

Application Domain 1: Parallel Programs

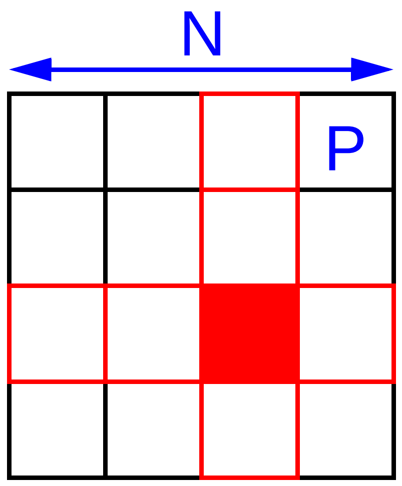
- true parallelism in one job
  - regular loop structures
  - data usually tightly shared
  - automatic parallelization
  - called "*data-level parallelism*"
  - can often exploit vectors as well

```
for (i=0;i<1000;i++){
    A[i] = B[i]*C[i];
}
```

- workloads
  - scientific simulation codes (e.g., FFT, weather, fluid dynamics, etc.)
  - was the dominant market segment of 10–20 years ago

Parallel Program Example: Matrix Multiply

- parameters
  - N = size of matrix (N*N)
  - P = number of processors
- growth functions
  - computation grows as $f(N^3)$
  - computation per processor grows as $f(N^3/P)$
  - data size grows as $f(N^2)$
  - data size per processor grows as $f(N^2/P)$
  - *communication* grows as $f(N^2/P^{1/2})$
  - computation/communication = $f(N/P^{1/2})$



Application Domain 2: Parallel Tasks

- parallel independent-but-similar tasks
  - irregular control structures
  - loosely shared data locked at different granularities
  - programmer defines & fine-tunes parallelism
  - cannot exploit vectors
  - called "*thread-level parallelism*" or "*throughput-oriented parallelism*"
- workload
  - transaction processing, OS, databases, web-servers
  - e.g., assign a thread to handle each request to server
  - dominant MP/MT market segment today

Many Ways to Design MT/MP Chips

How many cores per chip? Thread contexts per core? Should all cores on a chip be identical? Should they have the same clock frequency? Should they at least have the same ISA?

How do we connect the cores together?

Do cores share L2 caches? What about contention? What about cache coherence?

How do multiple threads share a single core's hardware? Round-robin sharing per cycle? All share at once?

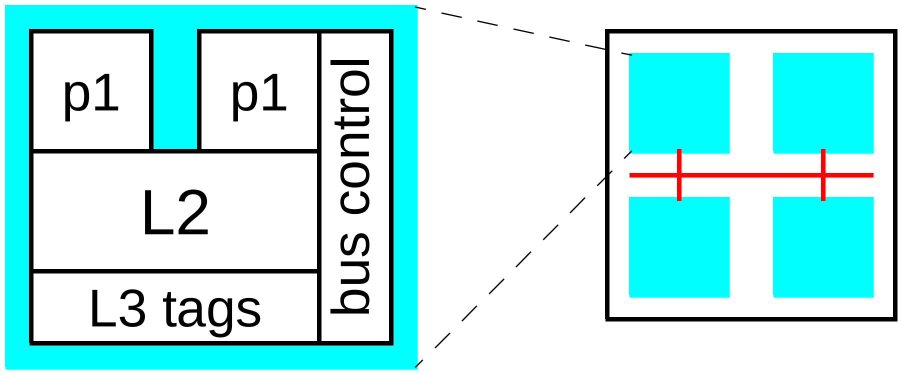
And many, many more issues to think about!

- Can you think of any other design issues?

Chip Multiprocessors (aka Multicore)

trend today: multiprocessors on a single chip (CMPs)

- can't spend all of the transistors on just one core
  - with limited ILP, single core would not exploit it
- an early example: IBM POWER4
  - 1 chip contains: 2 1GHz processors, L2, L3 tags, interconnect
  - can connect 4 chips on 1 MCM to create 8 processor system
  - targets threaded server workloads



Some Current CMPs

Intel and AMD: 2-core and 4-core chips

IBM: POWER6 has 2 cores, each of which is 2-way MT

IBM Xenon: 3 POWER cores

Cell processor: 9 cores (1 CPU and 8 SPUs)

Sun UltraSPARC T2: 8 cores, each of which is 8-way MT

Intel: 80-core prototype chip (Larrabee)

Azul Vega3: 54 cores

Tilera TILE64: 64 (no surprise!)

Cisco's CRS-1 router: 188 Tensilica cores

NVidia Tesla S1070: 960 "thread processors"

Multithreaded Processors

another trend: multithreaded processors

- processor utilization: IPC / processor width
  - decreases as processor width increases (~50% on 4 wide)
  - why? cache misses, branch mispredictions, RAW dependences
- idea: two (or more) processes (threads) share one pipeline
- replicate process (thread) state
  - PC, register file, bpred history, page table pointer, etc.
- one copy of stateless (or naturally tagged) structures
  - caches, functional units, buses, etc.
- hardware thread switch must be fast
  - multiple on-chip contexts $\Rightarrow$ no need to load from memory

Two Multithreading Paradigms

- coarse-grained
  - in-order processor with short pipeline
  - switch threads on long stalls (e.g., L2 cache misses)
  - instructions from one thread in stage per cycle
  - + threads don't interfere with each other much
  - – can't improve utilization on L1 misses, or branch mispredictions
  - e.g., IBM Northstar/Pulsar (2 threads)
- fine-grained: simultaneous multithreading (SMT)
  - out-of-order processor with deep pipeline
  - instructions from multiple threads in stage at same time
  - + improves utilization in all scenarios
  - – individual thread performances suffer due to interference
  - e.g., Pentium4 = 2 threads, Alpha 21464 (R.I.P.) = 4 threads

Taxonomy of Processors

Flynn Taxonomy [1966]

- not universal, but simple
- dimensions
  - instruction streams: single (SI) or multiple (MI)
  - data streams: single (SD) or multiple (MD)
- cross-product
  - SISD: uniprocessor (been there)
  - SIMD: vectors (brief sketch on next slide - more in ECE 259)
  - MISD: no practical examples (won’t do that)
  - *MIMD*: multiprocessors + multithreading (doing it now)

SIMD Vector Architectures

different type of ISA that has vector instructions

- addv $v1, $v2, $v3 # v1 = v2 + v3
- vector registers are like V N-bit scalar registers
  - for example, 128 32-bit values
- can operate on all entries of vector at once
- many Cray systems were vector architectures

useful for “vector code”

for i=1 to 128 { sum [i] = addend1 [i] + addend2 [i]; }

SIMD vs. MIMD

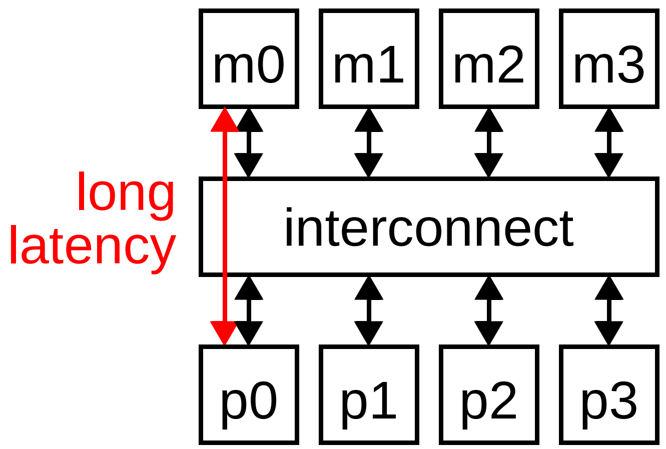
why are MPs (much) more common than vector processors?

- programming model flexibility
  - can simulate vectors with an MP, but not the other way around
  - dominant market segment cannot exploit vectors
- cost effectiveness
  - *commodity part*: high volume (translation: cheap) component
  - MPs made up of commodity parts (i.e., microprocessor cores)
  - can match size of MP to your budget
  - can't do this for a vector processor
- footnote: vectors have actually made a comeback
  - for graphics/multimedia applications (MMX, SSE, Tarantula)
  - NEC's EarthSimulator is an MP of vector processors

Taxonomy of Parallel (MIMD) Processors

- again, two dimensions
  - focuses on organization of main memory (shared vs. distributed)
- dimension I: appearance of memory to *hardware*
  - Q: is access to all memory uniform in latency?
  - *shared (UMA)*: yes ⇒ where you put data doesn't matter
  - *distributed (NUMA)*: no ⇒ where you put data <u>really</u> matters
- dimension II: appearance of memory to *software*
  - Q: can processors communicate via memory directly?
  - *shared (shared memory)*: yes ⇒ communicate via loads/stores
  - *distributed (message passing)*: no ⇒ communicate via messages
- dimensions are orthogonal
  - e.g., DSM: (physically) distributed (logically) shared memory

UMA vs. NUMA

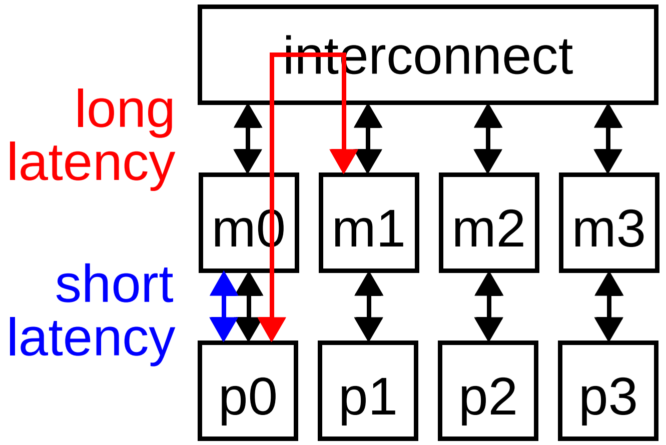


- *UMA: uniform memory access*
  - from p0, same latency to m0 as to m3

  + data placement unimportant (software is easier)

  – latency long, gets worse as system grows

  – interconnect contention restricts bandwidth
  - typically used in small multiprocessors only



- *NUMA: non-uniform memory accesss*
  - from p0 faster to m0 (local) than m3 (non-local)

  + low latency to local memory helps performance

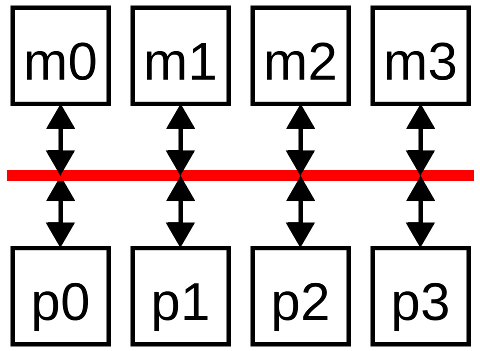
  – data placement important (software is harder)

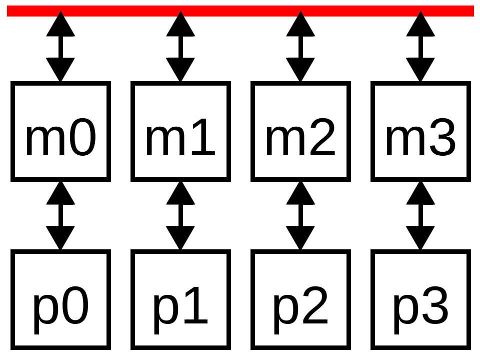
  + less contention (non-local only) ⇒ more scalable
  - typically used in larger multiprocessors

Interlude: What Is “Interconnect”?

- connects processors/memories to each other
  - *direct*: endpoints (i.e., procs, mems) connected directly (e.g., mesh)
  - *indirect*: endpoints connected via switches/routers (e.g., tree)
- interconnect issues
  - *latency*: average latency most important (locality optimizations?)
  - *bandwidth*: per processor (also, bisection bandwidth)
  - *cost*: # wires, # switches, # ports per switch
  - *scalability*: how latency, bandwidth, cost grow with # processors (P)
- we’ll mainly focus on *interconnect topology*
- can have separate interconnects for addresses and data

Interconnect 1: Bus





- direct interconnect

+ cost

- f(1) wires

+ latency: f(1)

- no neighbor/locality optimization

– bandwidth: *not scalable at all*, f(1/P)

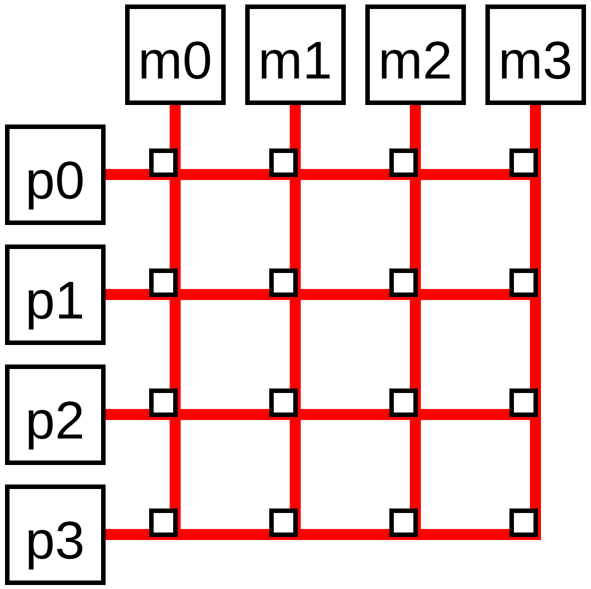
- only used in small systems (P <= 8)

+ capable of *ordered broadcast*

- incapable of anything else

- newer: logical buses w/point-to-point links
  - tree = logical bus, if all messages go to root
  - e.g., Sun UltraEnterprise E10000

Interconnect 2: Crossbar Switch



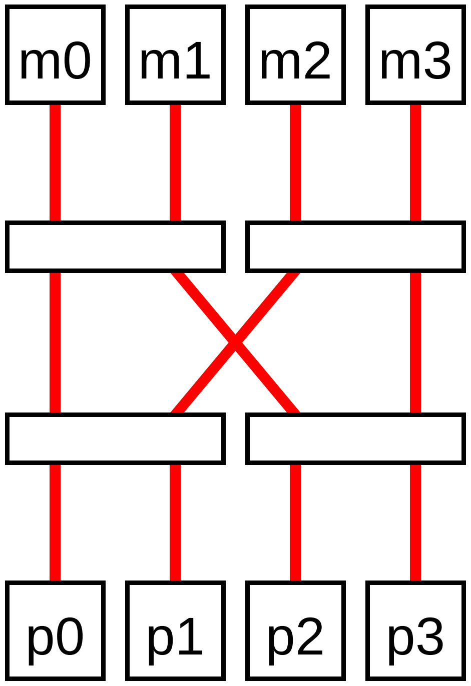
- indirect interconnect

+ latency: f(1)
  - no locality/neighbor optimizations

+ 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 bit decoding
  - *k*: switch arity (# inputs and outputs per switch)
  - *d*: number of network stages = $\log_k P$

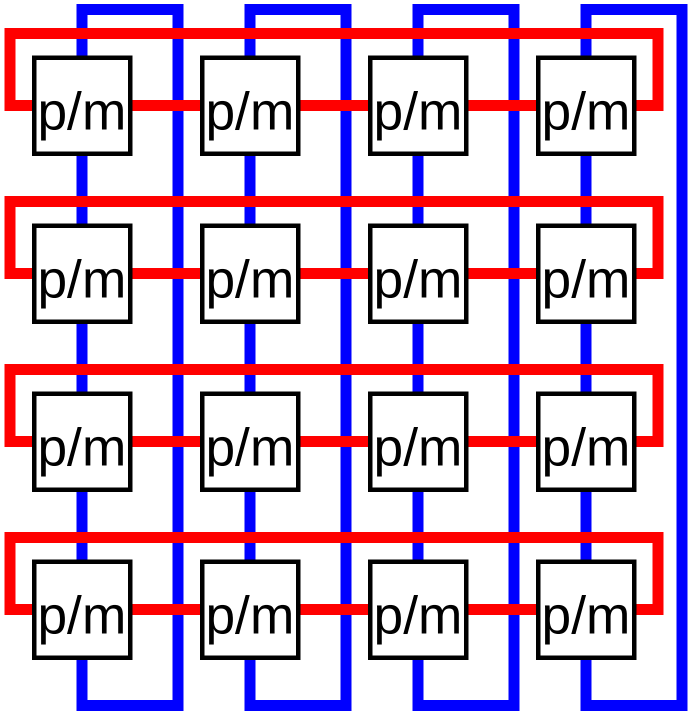
+ cost
  - f(d*P/k) switches
  - f(P*d) wires
  - f(k) wires per switch

+ latency: f(d)

+ bandwidth: f(1)

- commonly used in large UMA systems
  - a.k.a. butterfly, banyan, omega

Interconnect 4: 2D Torus



- direct interconnect
  - no dedicated switches

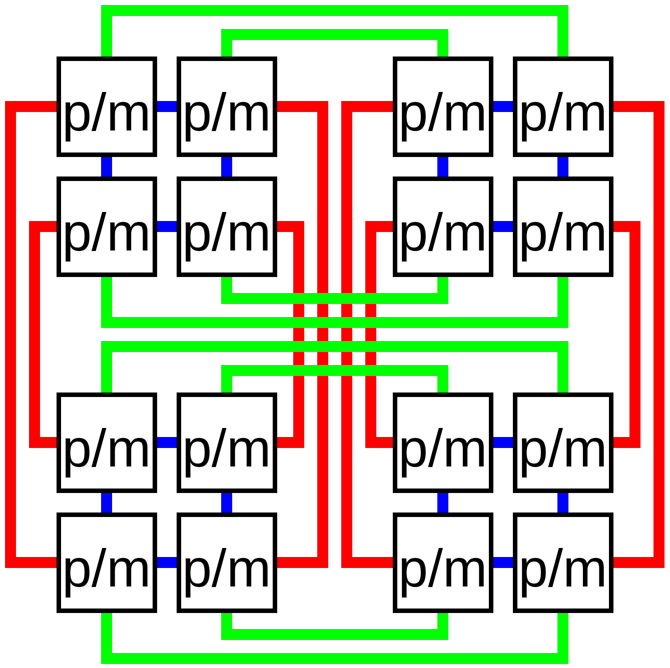
+ latency: f($P^{1/2}$)
  - locality/neighbor optimization

+ bandwidth: f(1), scales with P

+ cost
  - f(2P) wires
  - 4 wires per switch

- good scalability ⇒ widely used
  - variants: 3D, mesh (no “wraparound”)
- e.g., Alpha 21364-based MPs

Interconnect 5: Hypercube



- direct interconnect
  - *k*: arity (# nodes per dimension)
  - *d*: dimension = $\log_k P$
  - in figure: P = 16, k = 2, d = 4

+ latency: f(k/d)
  - locality/neighbor optimized

+ bandwidth: f((k–1)*d)

– cost
  - f((k–1)*d*P) wires
  - f((k–1)*d) wires per switch

- good scalability, expensive switches
  - – switch changes as nodes are added