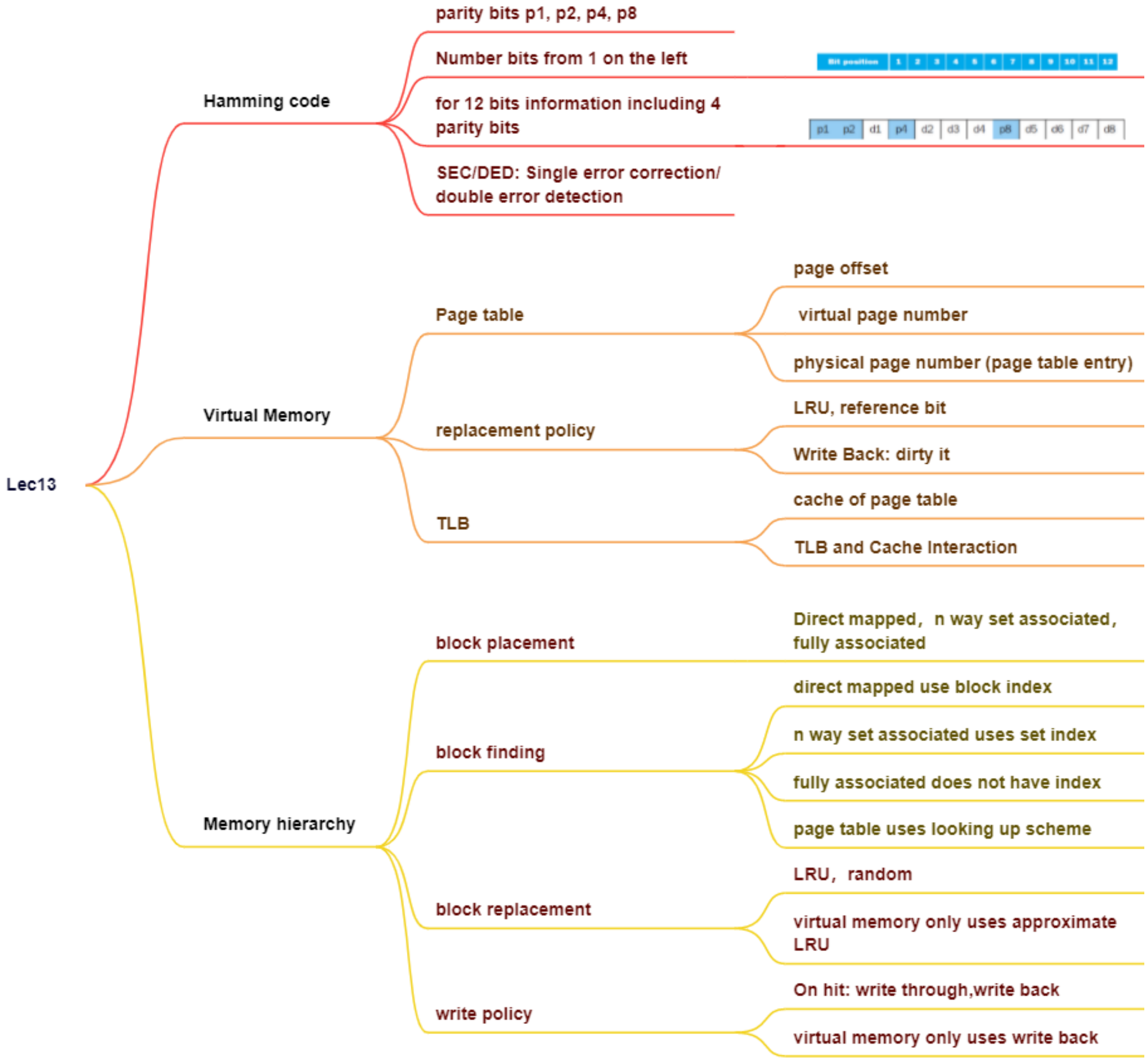
COMPUTER ORGANIZATION

Lecture 14
Parallel Processors from Client to Cloud

2024 Spring

Recap

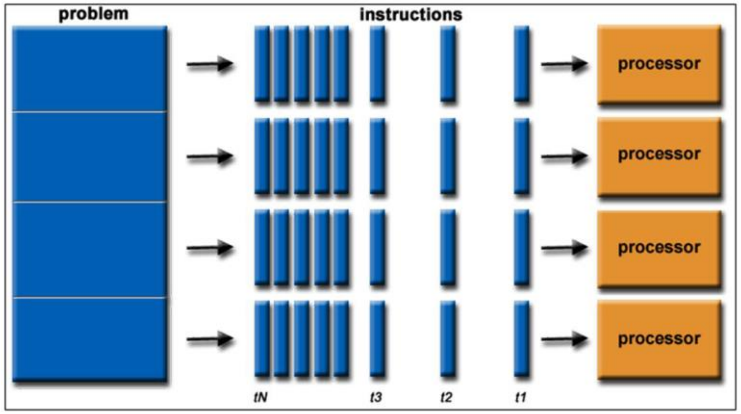


- Lec13
  - Hamming code
    - parity bits p1, p2, p4, p8
    - Number bits from 1 on the left
    - for 12 bits information including 4 parity bits
    - SEC/DED: Single error correction/ double error detection
  - Virtual Memory
    - Page table
      - page offset
      - virtual page number
      - physical page number (page table entry)
    - replacement policy
      - LRU, reference bit
      - Write Back: dirty it
    - TLB
      - cache of page table
      - TLB and Cache Interaction
  - Memory hierarchy
    - block placement
      - Direct mapped，n way set associated，fully associated
    - block finding
      - direct mapped use block index
      - n way set associated uses set index
      - fully associated does not have index
      - page table uses looking up scheme
    - block replacement
      - LRU，random
      - virtual memory only uses approximate LRU
    - write policy
      - On hit: write through,write back
      - virtual memory only uses write back

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8

Introduction

- Goal: replacing large inefficient processors with many smaller, efficient processors to get better performance per joule
    - Multiprocessors, cluster
    - Scalability, availability, power efficiency
- Task-level (process-level) parallelism
    - High throughput for independent jobs
- Parallel processing program
    - Single program run on multiple processors
- Multicore microprocessors
    - Chips with multiple processors (cores)
    - Shared Memory Processors (SMP)



Hardware and Software

- Challenge: hardware and software design that enables parallel processing programs, which can be efficiently executed (in performance and energy) when number of cores scales.
- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., Core i7
- Software
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
- Sequential/concurrent software can run on serial/parallel hardware

Parallel Programming

- We use “parallel processing program” to mean either sequential or concurrent software running on parallel hardware
- It’s hard to create parallel software
- Parallel programming needs to achieve significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties of parallel programming:
  - Partitioning
  - Coordination
  - Communications overhead

Amdahl's Law

- Amdahl’s Law: Calculate how much a computation can be sped up by running part of it in parallel

$$\text{Speedup} = \frac{T_1 + T_2}{T'} = \frac{T_1 + T_2}{\frac{T_1}{N} + T_2} = \frac{1}{\frac{1}{N}\left(\frac{T_1}{T_1+T_2}\right) + \left(\frac{T_2}{T_1+T_2}\right)}$$

$$\boxed{\text{Speedup} = \frac{1}{\frac{1}{N}(F_{\text{parallelizable}}) + (1 - F_{\text{parallelizable}})}}$$

$T_1 = T_{\text{affected}}$
$T_2 = T_{\text{unaffected}}$
$T' = T_{\text{improved}}$
N = improve factor
F = parallel portion

Amdahl’s Law

- Sequential part can limit speedup
- Example: 100 processors, how to achieve 90× speedup?
  - 1 processor: $F_{sequential}+F_{paralizable}=1(T_{old})$
  - 100 processors: $T_{new} = F_{sequential} + F_{parallelizable}/100$

$$\text{Speedup} = \frac{1}{(1-F_{parallelizable})+F_{parallelizable}/100} = 90$$

  - ➔ $F_{parallelizable} = 0.999$
- sequential part should be no more than 0.1% of total task

Scaling Example

- Workload: sum of 10 scalars, sum of two 10 × 10 matrix
  - Assume the sum of 10 scalars cannot be paralleled
  - What is the speedup of 10 processors and 40 processors?
- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (5.5/10=55% of potential)
- 40 processors
  - Time = $10 \times t_{add} + 100/40 \times t_{add} = 12.5 \times t_{add}$
  - Speedup = 110/12.5 = 8.8 (8.8/40=22% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont.)

- What if matrix size is 20 × 20?
- Single processor: Time = (10 + 400) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 400/10 × t_{add} = 50 × t_{add}
  - Speedup = 410/50 = 8.2 (8.2/10=82% of potential)
- 40 processors
  - Time = 10 × t_{add} + 400/40 × t_{add} = 20 × t_{add}
  - Speedup = 410/20 = 20.5 (20.5/40=51% of potential)
- Assuming load balanced
- Make common case faster

Strong vs Weak Scaling

- Speedup:

task No. of cores	10 scaler 10*10 matrix	10 scaler 20*20 matrix
10	5.5 (55% of potential)	8.2 (82% of potential)
40	8.8 (22% of potential)	20.5 (51% of potential)

- Strong scaling: keep problem size fixed, time is reverse proportional to number of processors: $T(N,P)=T(1,P)/N$
- Weak scaling: constant time cost when problem size is proportional to number of processors: $T(N_1,P_1)=T(N_2,P_2)$
  - 10 processors (N_1), 10 × 10 matrix (P_1)
    - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - 40 processors (N_2), 20 × 20 matrix (P_2)
    - Time = $10 \times t_{add} + 400/40 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

Load Balancing

- In the above example
  - 40 processors are used to achieve 20.5 speedup
  - 40 processors are assumed to have balanced load (2.5% each)
- how about one processor with high load (5%)?
  - one processor takes 5%*400=20 adds, the others takes the rest 400-20=380 adds
  - Time = max(380t/39, 20t/1)+10t = 30t
  - speedup = 410t/30t = 14, smaller than 20.5
- how about one processor with higher load (12.5%)?

Parallel Processing

- The following techniques can enable parallel processing
  - SIMD, vector (section 6.3)
  - Multithreading (section 6.4)
  - SMPs and clusters (section 6.5)
  - GPUs (section 6.6)

SISD, MIMD, SPMD, SIMD and vector

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE of ISA x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Core i7

- SISD: single instruction stream, single data stream
  - Uniprocessor
- MIMD : multiple instruction, multiple data
  - Multi-core processor
  - SPMD: single program, multiple data
    - Typical way to write program on a multi-core processor
    - One program run on multiple processors
    - Different processors execute on different sections of code

SIMD

- Operate on vectors of data
  - Provide data level parallelism
  - E.g., MMX (MultiMedia eXtension) and SSE (Streaming SIMD Extension) instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to RISC-V
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
  - Vector instructions
    - fld.v, fsd.v: load/store vector
    - fadd.d.v: add vectors of double
    - fadd.d.vs: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

An example using vector instruction

Calculate:

$$Y = a \times X + Y$$

- X and Y are vectors of 64 double precision numbers
- the starting addresses of X and Y are in $s0 and $s1

```
       l.d      $f0,a($sp)       :load scalar a
       addiu    $t0,$s0,#512     :upper bound of what to load
loop:  l.d      $f2,0($s0)       :load x(i)
       mul.d    $f2,$f2,$f0      :a*x(i)
       l.d      $f4,0($s1)       :load y(i)
       add.d    $f4,$f4,$f2      :a*x(i) + y(i)
       s.d      $f4,0($s1)       :store into y(i)
       addiu    $s0,$s0,#8       :increment index to x
       addiu    $s1,$s1,#8       :increment index to y
       subu     $t1,$t0,$s0      :compute bound
       bne      $t1,$zero,loop   :check if done
```

Code using vector instructions:

```
       l.d      $f0,a($sp)       :load scalar a
       lv       $v1,0($s0)       :load vector x
       mulvs.d  $v2,$v1,$f0      :vector-scalar multiply
       lv       $v3,0($s1)       :load vector y
       addv.d   $v4,$v2,$v3      :add y to product
       sv       $v4,0($s1)       :store the result
```

Example: DAXPY (Y = a × X + Y)

Conventional RISC-V code:

```
      fld     f0,a(x3)     // load scalar a
      addi    x5,x19,512   // end of array X
loop: fld     f1,0(x19)    // load x[i]
      fmul.d  f1,f1,f0     // a * x[i]
      fld     f2,0(x20)    // load y[i]
      fadd.d  f2,f2,f1     // a * x[i] + y[i]
      fsd     f2,0(x20)    // store y[i]
      addi    x19,x19,8    // increment index to x
      addi    x20,x20,8    // increment index to y
      bltu    x19,x5,loop  // repeat if not done
```

Calculate:

$$Y = a \times X + Y$$

- X and Y are vectors of 64 double precision numbers
- the starting addresses of X and Y are in \$s0 and \$s1

Vector RISC-V code:

```
      fld        f0,a(x3)     // load scalar a
      fld.v      v0,0(x19)    // load vector x
      fmul.d.vs  v0,v0,f0     // vector-scalar multiply
      fld.v      v1,0(x20)    // load vector y
      fadd.d.v   v1,v1,v0     // vector-vector add
      fsd.v      v1,0(x20)    // store vector y
```

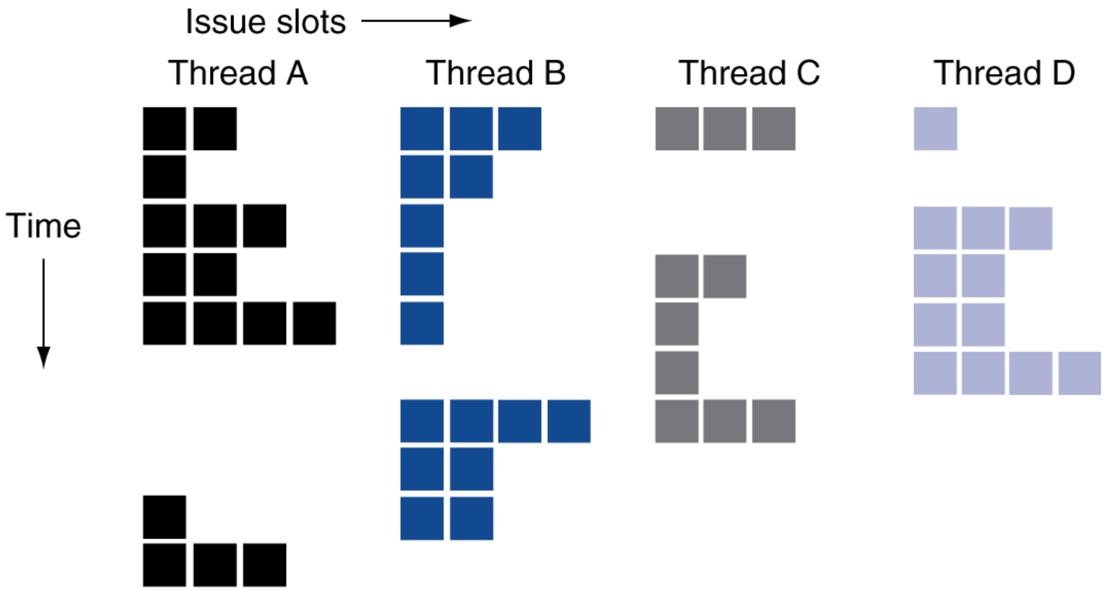
Multithreading

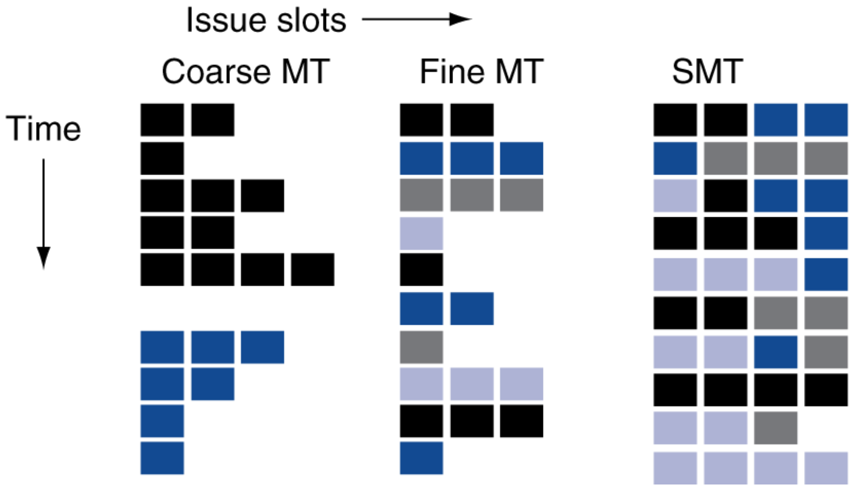
- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
  - Threads: a lightweight process, share single address space
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

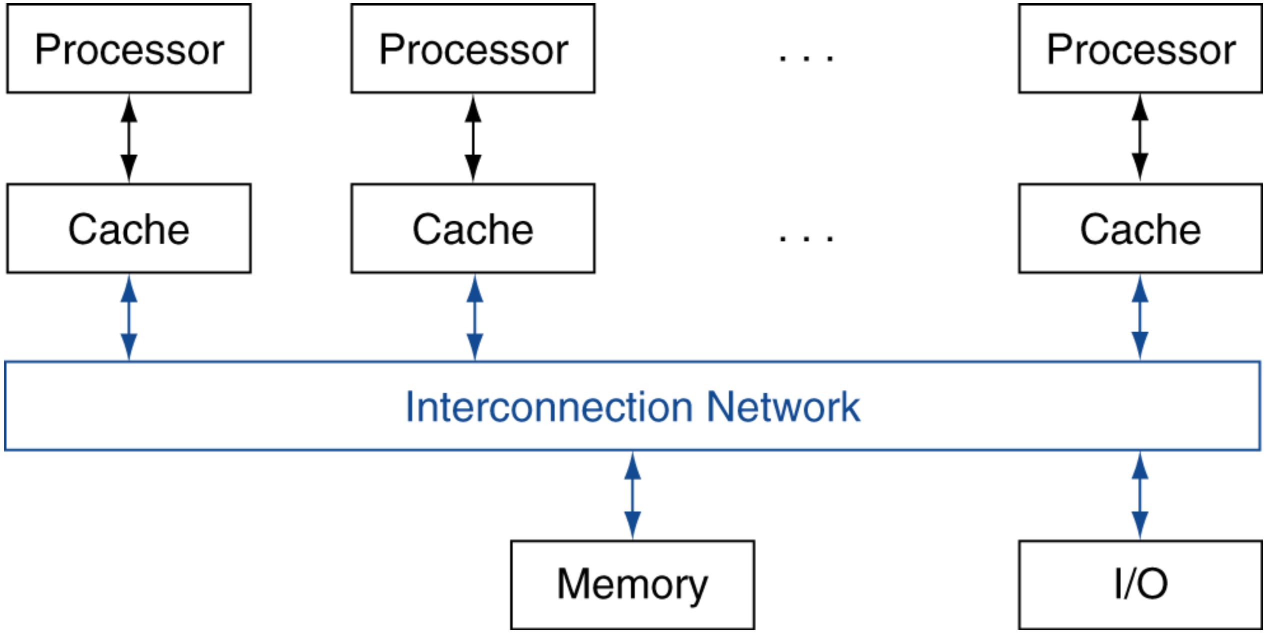
Multithreading Example





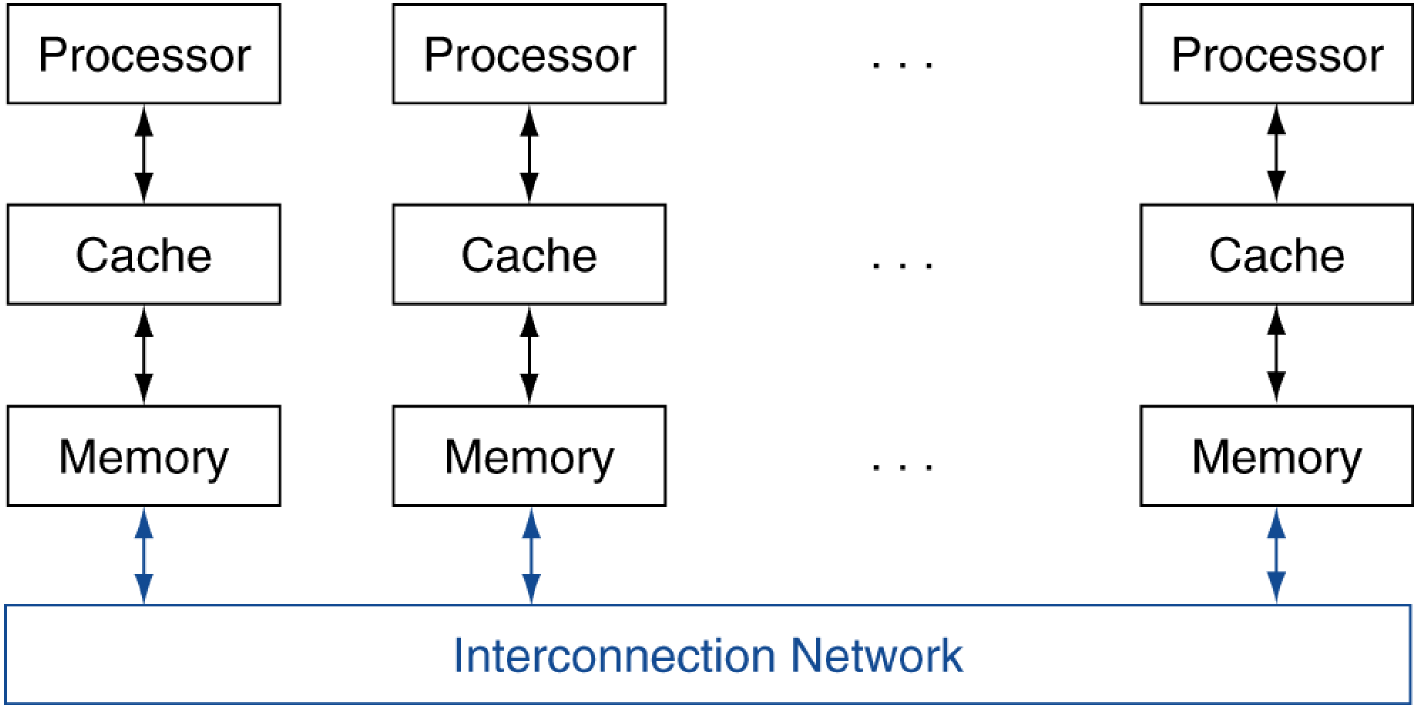
Shared Memory Multiprocessors

- Multithreading improves efficiency of one processor
- How to efficiently programming on multiprocessor?
  - Don’t want to rewrite old programs in single processor
  - Share memory among multiple cores
- SMP: shared memory multiprocessor



Message Parsing Multiprocessors

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

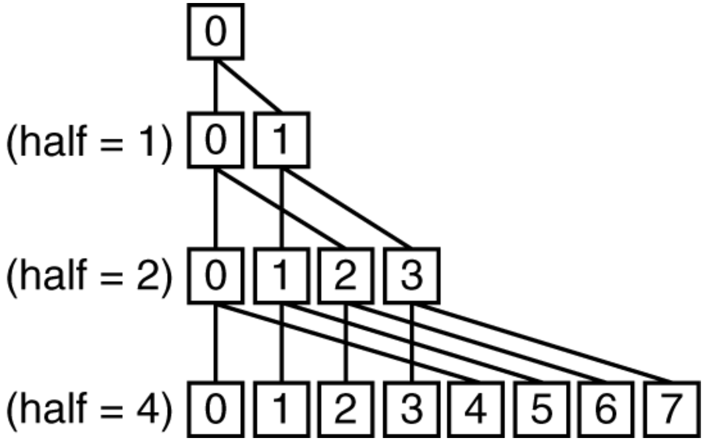
- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Example: Sum Reduction

- Sum 100,000 numbers on 100 processor
  - Partition 1000 numbers per processor
    - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
       i < 1000*(Pn+1); i = i + 1)
     sum[Pn] = sum[Pn] + A[i];
```

  - Now need to add these partial sums
    - Reduction: divide and conquer
    - Half the processors add pairs, then quarter, …



Cloud Computing and Data Center

- Cloud computing
  - Warehouse Scale Computers (WSC)
  - Software as a Service (SaaS)
  - Portion of software run on a PMD (personal mobile device) and a portion run in the Cloud
  - Amazon and Google
- Data centers
  - Millions of computers connected by off-the-shelf networking devices

Google Data Centers

Introduction of GPUs

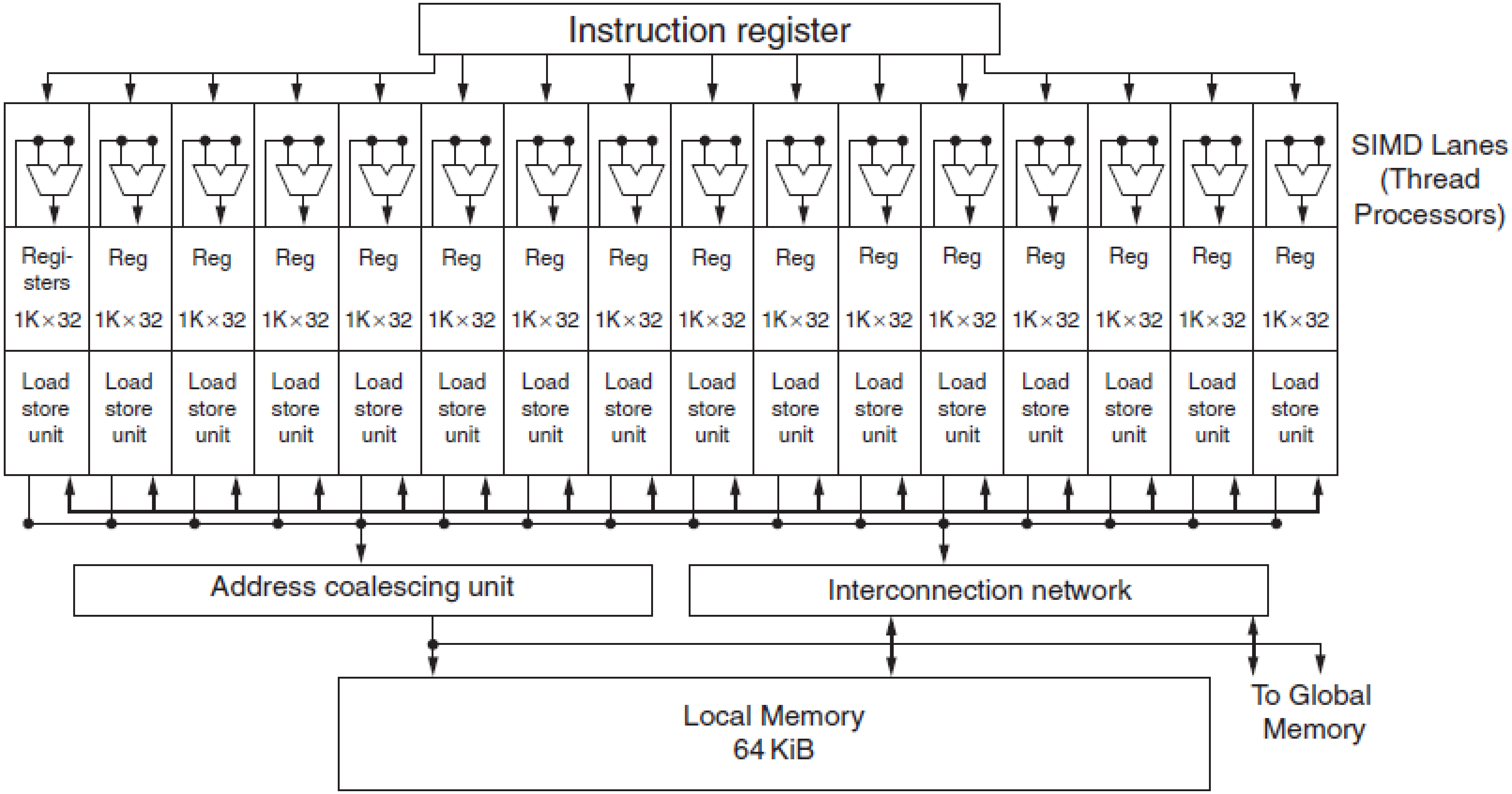
- Development of GPU
  - General-purpose CPU not suitable for graphic processing
  - Game industry drive the improvement of graphic processing
  - GPU (graphic processing unit) appear and developed faster than general-purpose CPU, high performance, low cost
  - Easy-use programming language helps GPU's popularity
- Difference between GPU and CPU
  - GPU supplement CPU, doesn't replace it. GPU doesn't need to perform all tasks of CPU
  - GPU problem size is MB to GB, not hundreds of GB.

GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded, 16-32 threads
- GPU memory oriented towards bandwidth rather then latency
  - Graphics memory is wide and high-bandwidth
  - GPU memory is smaller than CPU. 4-6GB, instead of 256GB or above
    - Less reliance on multi-level caches
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

Block Diagram of a SIMD processor in GPU

- A GPU consists of multiple multi-thread SIMD processors (Nvidia Fermi architecture)



Classifying GPUs

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	Nvidia Multiprocessor

xPUs

- CPU: good for control, sequential programming
- GPU: good for graphics, parallel programming
  - CPU/GPU mixed architecture
- TPU: Tenser processing unit
  - Proposed by Google, targeting at acceleration for tenserflow platform Suitable for machine learning model training and testing
- DPU: deep learning processing unit
  - Proposed by DeePhi Tech, FPGA-based processing unit
- NPU: neural network processing unit
  - IBM TrueNorth
- BPU: brain processing unit

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- From multicore to data centre
- Performance per dollar and performance per Joule drive both mobile and WSC

Summary for this course

- Digital logic → computer organization → operating system/embedded system
- Main content:
  - Processor
  - Memory
  - Parallel
- Hardware thinking is important: resource tradeoff