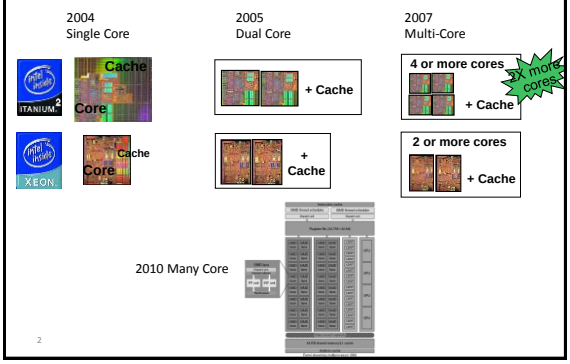
COMP4611: Design and Analysis of Computer Architectures

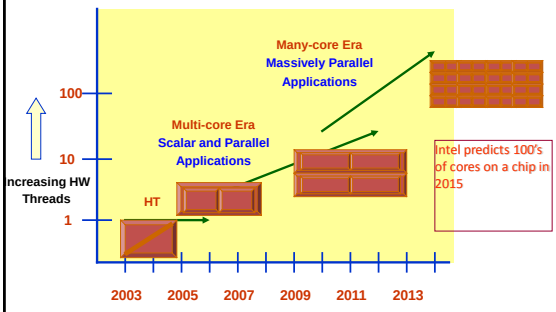
Multiprocessors

Lin Gu
CSE, HKUST

Multi-Core Technology



Multiprocessing within a chip: Many-Core



Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - Heat problems
  - Clock problems
  - Efficiency (Stall) problems
- Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, is extremely difficult
  - issue 3 or 4 data memory accesses per cycle,
  - rename and access more than 20 registers per cycle, and
  - fetch 12 to 24 instructions per cycle.
- Many new applications are multithreaded

A general trend in computer architecture is to shift towards more parallelism through more processors or processor cores

What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)

Thread-level parallelism (TLP)

- This is parallelism on a more coarse scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and sound in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

7

Thread-Level Parallelism

8

Introduction

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled shared-memory multiprocessors
- For *n* processors, need *n* threads
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

How to exploit TLP?

- Execute instructions from multiple threads on a single processor
  - Coarse-grain, fine-grain, SMT (Simultaneous Multi-Threading)
- Execute multiple threads on multiple processors
  - "Anything that can be threaded today will map efficiently to multi-core"

10

SMT – Simultaneous Multi-Threading

A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and thread-level parallelism (TLP)

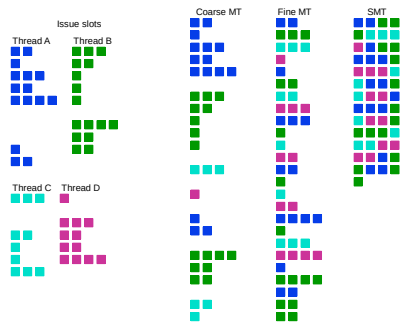
With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued in one cycle without regard to dependencies among them

Need separate rename tables (ROBs) for each thread

Need the capability to commit from multiple threads (i.e., from multiple ROBs) in one cycle

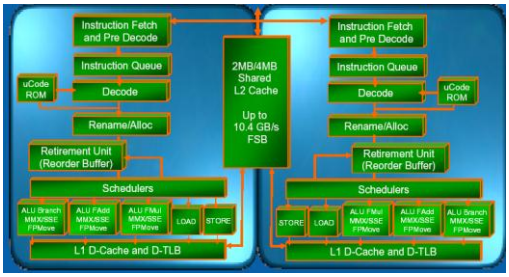
11

TLP a 4-issue superscalar processor



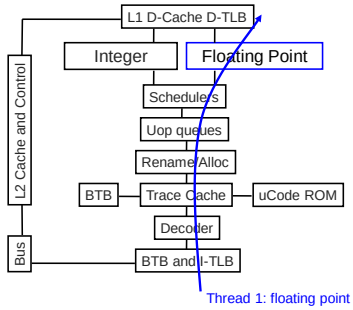
12

Core 2 Duo Microarchitecture



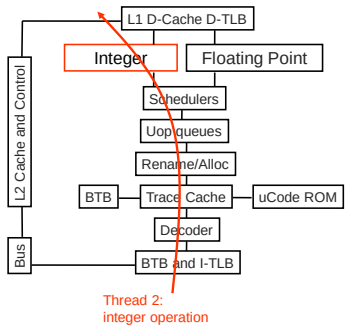
13

Without SMT, only a single thread can run at any given time



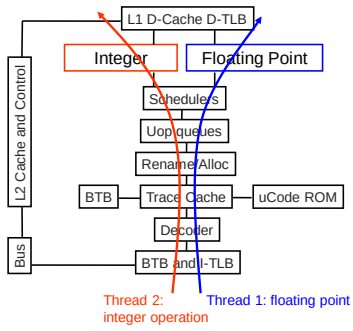
14

Without SMT, only a single thread can run at any given time



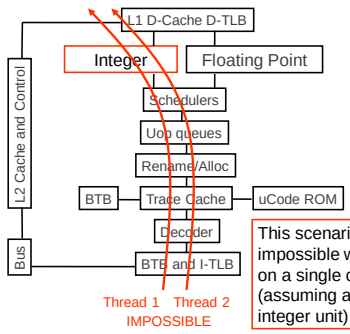
15

SMT processor: both threads can run concurrently



16

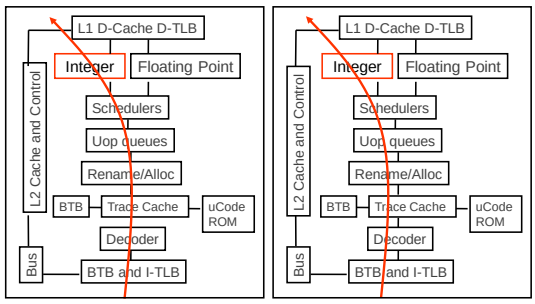
But: Can't simultaneously use the same functional unit



This scenario is impossible with SMT on a single core (assuming a single integer unit)

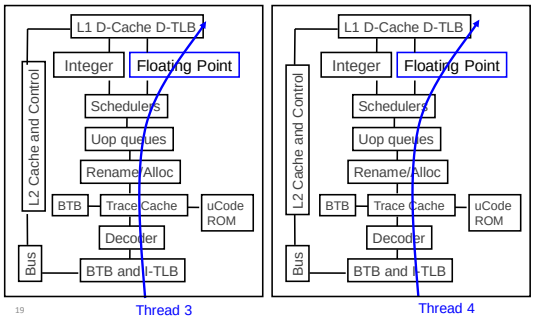
17

Multi-core: threads can run on separate cores



18

Multi-core: threads can run on separate cores



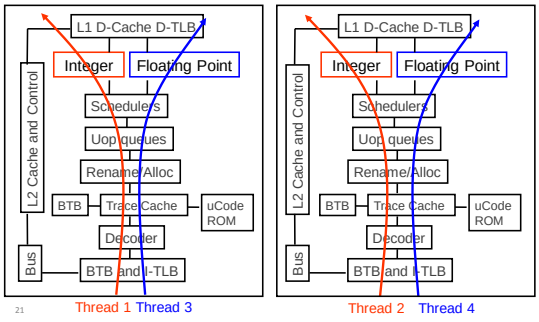
19

Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT
- The number of SMT threads: 2, 4, or sometimes 8 simultaneous threads
- Intel calls them "hyper-threads"

20

SMT Dual-core: all four threads can run concurrently



21

High-Performance Computing

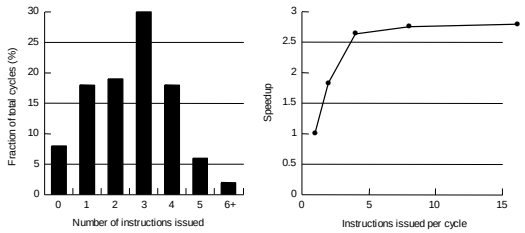
23

Processor Performance

- We have looked at various ways of increasing a single processor performance (Excluding VLSI techniques):
  - ✓ Pipelining
  - ✓ ILP
  - ✓ Superscalars
  - ✓ Out-of-order execution (Scoreboarding, Tomasulo)
  - ✓ VLIW
  - ✓ Cache (L1, L2, L3)
  - ✓ Interleaved memories
  - ✓ Compilers (Loop unrolling, branch prediction, etc.)
  - ✓ RAID
  - ✓ Etc …
- However, quite often even the best microprocessors are not fast enough for certain applications !!!

23

Example: How far will ILP go?



- Infinite resources and fetch bandwidth, perfect branch prediction and renaming

24

When Do We Need High Performance Computing?

Case1

–To do a time-consuming operation in **less time**

- I am an aircraft engineer
- I need to run a simulation to test the stability of the wings at high speed
- I'd rather have the result in 5 minutes than in 5 days so that I can complete the aircraft final design sooner.

25

When Do We Need High Performance Computing?

Case 2

– To do a **high number of operations** per seconds

- I am an engineer of Amazon.com
- My Web server gets 10,000 hits per seconds
- I'd like my Web server and my databases to handle 10,000 transactions per seconds so that customers do not experience bad delays

–Amazon does "process" several GBytes of data per seconds

26

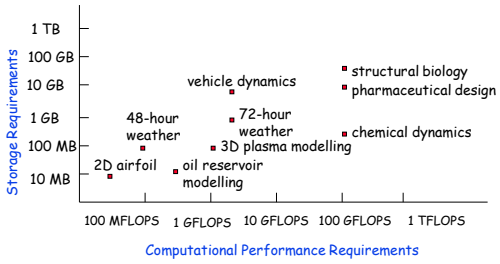
The need for High-Performance Computers

some examples

- Automotive design:
  - –Major automotive companies use large systems (500+ CPUs) for:
    - CAD-CAM, crash testing, structural integrity and aerodynamics.
  - –Savings: approx. $1 billion per company per year.
- Semiconductor industry:
  - –Semiconductor firms use large systems (500+ CPUs) for
    - device electronics simulation and logic validation
  - –Savings: approx. $1 billion per company per year.
- Airlines:
  - –System-wide logistics optimization systems on parallel systems.
  - –Savings: approx. $100 million per airline per year.

27

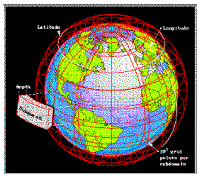
Grand Challenges



28

Weather Forecasting

- Suppose the whole global atmosphere divided into cells of size 1 km × 1 km × 1 km to a height of 10 km (10 cells high) - about 5×10^8 cells.
- Suppose each cell calculation requires 200 floating point operations. In one time step, 10^{11} floating point operations are necessary.
- To forecast the weather over 7 days using 1-minute intervals, a computer operating at 1Gflops (10^9 floating point operations/s) – **similar to the Pentium 4** - takes 10^6 seconds or over 10 days.
- To perform calculation in 5 minutes requires a computer operating at 3.4 Tflops (3.4×10^{12} floating point operations/sec).



29

Multiprocessing

- Multiprocessing (Parallel Processing): *Concurrent* execution of tasks (programs) using *multiple computing, memory and interconnection resources*.

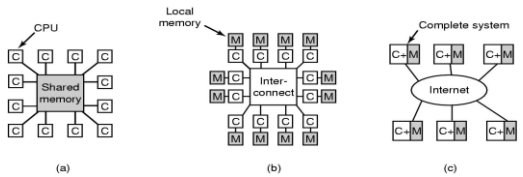
  Use multiple resources to solve problems faster

- Provides alternative to faster clock for performance
  - *Assuming a doubling of effective processor performance every 2 years, 1024-Processor system (assuming linear performance gain) can get you the performance that it would take 20 years for a single-processor system to deliver*
- Using multiple processors to solve a single problem
  - Divide problem into many small pieces
  - Distribute these small problems to be solved by multiple processors simultaneously

30

Multiprocessing

- For the last 30+ years multiprocessing has been seen as the best way to produce orders of magnitude performance gains.
  - Double the number of processors, get (theoretically) double performance (less than 2 times the cost).
- It turns out that the ability to develop and deliver software for multiprocessing systems induces impediment to wide adoption.



31

Amdahl's Law

- A parallel program has a sequential part (e.g., I/O) and a parallel part
  - $T_1 = \beta T_1 + (1-\beta)T_1$
  - $T_p = \beta T_1 + (1-\beta)T_1 / p$
- Therefore:

$$\text{Speedup}(p) = 1 / (\beta + (1-\beta)/p)$$
$$= p / (\beta p + 1 - \beta)$$
$$\leq 1 / \beta$$

- Example: if a code is 10% sequential (i.e., $\beta = .10$), the speedup will always be lower than 1 + 90/10 = 10, no matter how many processors are used!

32

Maximum Speedup

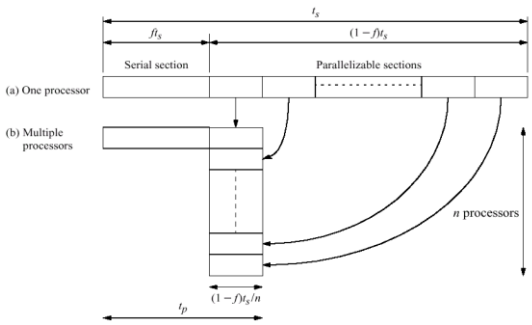


Figure 1.29 Parallelizing sequential problem — Amdahl's law.

Performance Potential Using Multiple Processors

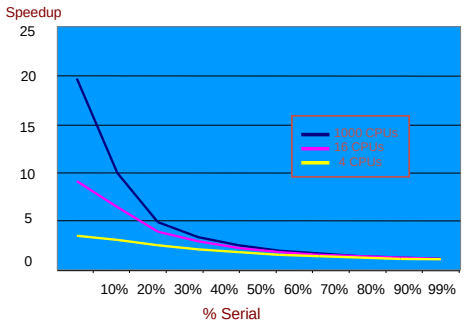
- Amdahl's Law is pessimistic (in this case)
  - Let s be the serial part
  - Let p be the part that can be parallelized n ways

```
- Serial:          SSPPPPPP
- 6 processors:    SSP
-                    P
-                    P
-                    P
-                    P
-                    P
```

  - Speedup = 8/3 = 2.67
  - $T(n) = \frac{1}{s+p/n}$
  - As $n \to \infty$, $T(n) \to \frac{1}{s}$
- Pessimistic

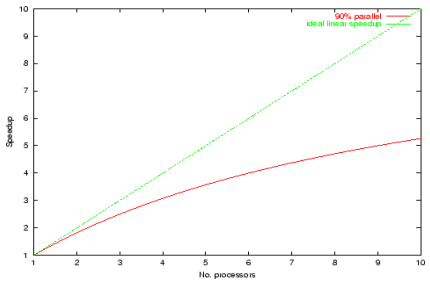
34

Amdahl's Law



35

Example



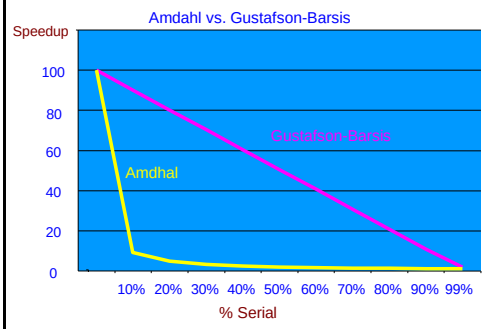
36

Performance Potential: Another view

- Gustafson view (more widely adopted for multiprocessors)
  - Parallel portion increases as the problem size increases
    - Serial time fixed (at s)
    - Parallel time proportional to problem size (true most of the time)
    - Gustafson's Law: Speedup(N) = N - β(N-1)
      - N: number of processors, β : weight of non-parallelizable part

```
Old Serial:        SSPPPPPP
6 processors:      SSPPPPPP
                     PPPPPP
                     PPPPPP
                     PPPPPP
                     PPPPPP
                     PPPPPP
Hypothetical Serial:
     SSPPPPPP PPPPPP PPPPPP PPPPPP PPPPPP PPPPPP
```

  - Speedup(6) = (8+5*6)/8 = 4.75
    - β =? in this calculation?
  - Speedup(N) = N(1- β) + β; Speedup'(∞) → ∞!!!!



TOP 5 Most computers in the world – must be multiprocessors

2008

No	Site Country	# of cores	Manufacturer, Model, Architecture	Rmax (TFlops)
1	DOE/NNSA/LANL, USA	129600	IBM, BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband	1105
2	Oak Ridge National Laboratory, USA	150152	Cray XT5 QC 2.3 GHz	1059
3	NASA/Ames Research Center/NAS, USA	51200	SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz	487
4	DOE/NNSA/LLNL, USA	212992	IBM, eServer Blue Gene Solution	478
5	Argonne National Laboratory, USA	163840	IBM, Blue Gene/P Solution	450

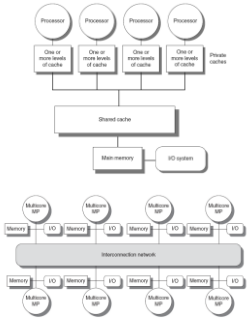
Nov. 2012

Site/Country/Year	# of cores	Manufacturer, Model, Architecture	Rmax (TFlops)
DOE/SC/Oak Ridge National Laboratory	560640	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	17590
DOE/NNSA/LLNL	1572864	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz	16324
RIKEN Advanced Institute for Computational Science (AICS)	705024	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect	10510
DOE/SC/Argonne National Laboratory	786432	Mira - BlueGene/Q, Power BQC 16C 1.60GHz	8162
Forschungszentrum Juelich (FZJ)	393216	JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect	4141

http://www.top500.org/

A Few Types

- Symmetric multiprocessors (SMP)
  - Small number of cores
  - Share single memory with uniform memory latency
- Distributed shared memory (DSM)
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (mult hop) interconnection networks

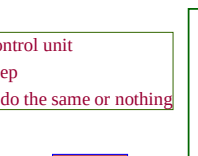


Flynn's Taxonomy of Computing

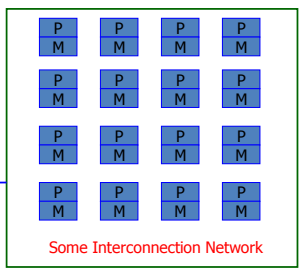
- *SISD* (Single Instruction, Single Data):
  - Typical uniprocessor systems that we've studied throughout this course.
  - Uniprocessor systems can time share and still be SISD.
- *SIMD* (Single Instruction, Multiple Data):
  - Multiple processors *simultaneously* executing the *same instruction* on different data.
  - Specialized applications (e.g., image processing).
- *MIMD* (Multiple Instruction, Multiple Data):
  - Multiple processors *autonomously* executing *different instructions on different data*.
  - Keep in mind that the processors are working together to solve a single problem.
  - This is a more general form of multiprocessing, and can be used in numerous applications

SIMD Systems

One control unit
Lockstep
All Ps do the same or nothing

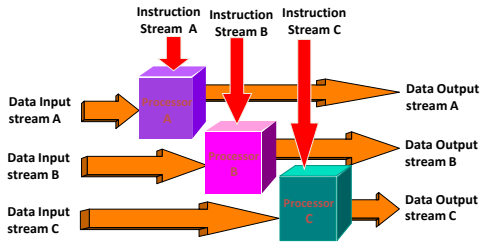


von Neumann Computer



Some Interconnection Network

MIMD Architecture

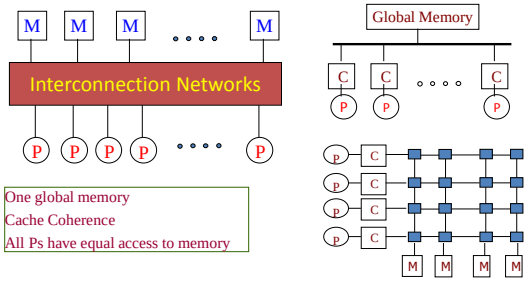


Unlike SIMD, MIMD computer works asynchronously.

- Shared memory (tightly coupled) MIMD
- Distributed memory (loosely coupled) MIMD

43

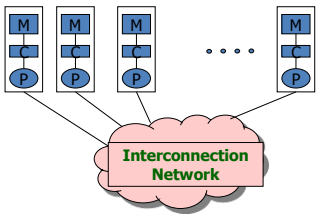
MIMD Shared Memory Systems



One global memory
Cache Coherence
All Ps have equal access to memory

44

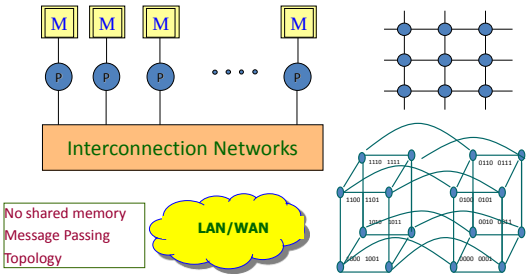
Cache Coherent NUMA



Each P has part of the shared memory
Non uniform memory access

45

MIMD Distributed Memory Systems

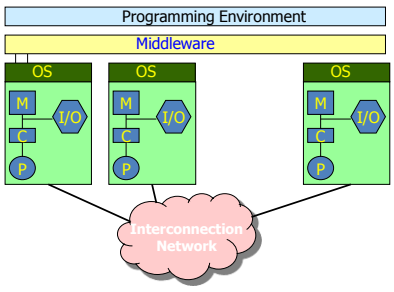


No shared memory
Message Passing
Topology

LAN/WAN

46

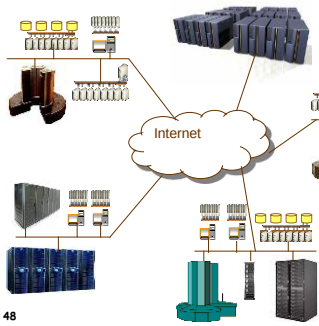
Cluster Architecture



Home cluster

47

Grids



Geographically distributed platforms.

Dependable, consistent, pervasive, and inexpensive access to high end computing.

48

SIMD

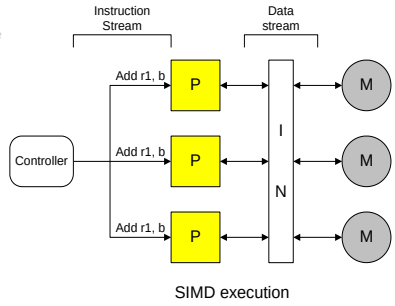
49

SIMD Parallel Computing

It can be a stand-alone multiprocessor

Or

Embedded in a single processor for specific applications (MMX)



SIMD execution

50

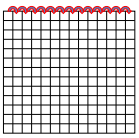
SIMD Applications

- Database, image processing, and signal processing.
- Image processing maps very naturally onto SIMD systems.
  - Each processor (Execution unit) performs operations on a single pixel or neighborhood of pixels.
  - The operations performed are fairly straightforward and simple.
  - Data could be streamed into the system and operated on in real-time or close to real-time.

51

SIMD Operations

- Image processing on SIMD systems.
  - Sequential pixel operations take a very long time to perform.
    - A 512x512 image would require 262,144 iterations through a sequential loop with each loop executing 10 instructions. That translates to 2,621,440 clock cycles (if each instruction is a single cycle) plus loop overhead.



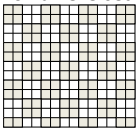
512x512 image

Each pixel is operated on sequentially one after another.

52

SIMD Operations

- Image processing on SIMD systems.
  - On a SIMD system with 64x64 processors (e.g., very simple ALUs) the same operations would take 640 cycles, where each processor operates on an 8x8 set of pixels plus loop overhead.



512x512 image

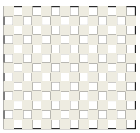
Each processor operates on an 8x8 set of pixels in parallel.

Speedup due to parallelism: 2,621,440/640 = 4096 = 64x64 (number of proc.) loop overhead ignored.

53

SIMD Operations

- Image processing on SIMD systems.
  - On a SIMD system with 512x512 processors (which is not unreasonable on SIMD machines) the same operation would take 10 cycles.



512x512 image

Each processor operates on a single pixel in parallel.

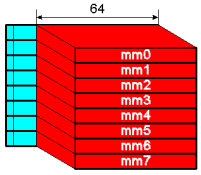
Speedup due to parallelism: 2,621,440/10 = 262,144 = 512x512 (number of proc.)!

Notice no loop overhead!

54

Pentium MMX MultiMedia eXtentions

- 57 new instructions
- Eight 64-bit wide MMX registers
- First available in 1997
- Supported on:
  - Intel Pentium-MMX, Pentium II, Pentium III, Pentium IV
  - AMD K6, K6-2, K6-3, K7 (and later)
  - Cyrix M2, MMX-enhanced MediaGX, Jalapeno (and later)
- Gives a large speedup in many multimedia applications



MMX Registers

55

MMX SIMD Operations

- Example: consider an image pixel data represented as bytes.
  - with MMX, eight of these pixels can be packed together in a 64-bit quantity and moved into an MMX register
  - MMX instruction performs the arithmetic or logical operation on all eight elements in parallel
- PADD(B/W/D): Addition

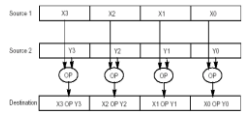
PADDB MM1, MM2

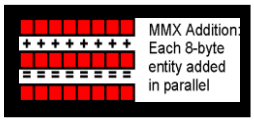
adds 64-bit contents of MM2 to MM1,

byte-by-byte any carries generated

are dropped, e.g., byte A0h + 70h = 10h

- PSUB(B/W/D): Subtraction





56

MMX: Image Dissolve Using Alpha Blending

- **Example:** MMX instructions speed up image composition
- A flower will dissolve a swan
- Alpha (a standard scheme) determines the intensity of the flower
- The full intensity, the flower's 8-bit alpha value is FFh, or 255
- The equation below calculates each pixel:

Result_pixel =Flower_pixel *(alpha/255) + Swan_pixel * [1-(alpha/255)]

For alpha 230, the resulting pixel is 90% flower and 10% swan

57

SIMD Multiprocessing

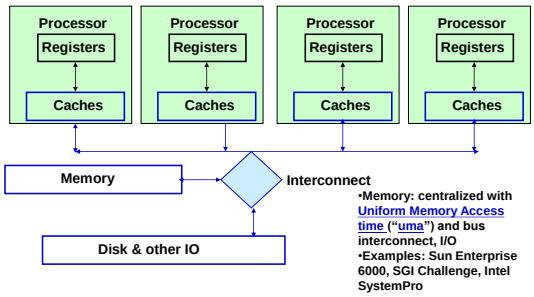
- Traditional vector computers are typical SIMD systems
- In the late 80s and early 90s, many SIMD machines were commercially available (e.g., Connection machine has 64K ALUs, and MasPar has 16K ALUs)
- GPU revives the SIMD computation, and is widely used in high-performance computers
- SPMD—Single Program, Multiple Data

58

Cache Coherence

59

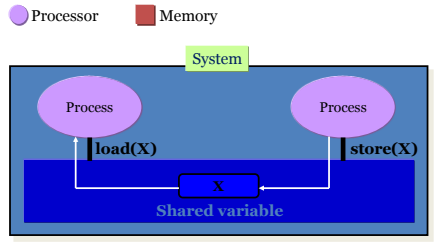
Shared Memory Multiprocessor



- Memory: centralized with Uniform Memory Access time ("uma") and bus interconnect, I/O
- Examples: Sun Enterprise 6000, SGI Challenge, Intel SystemPro

60

Shared Memory Programming Model

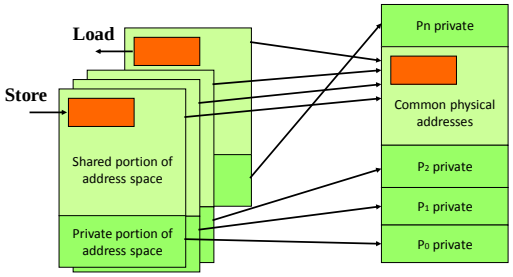


61

Shared Memory Model

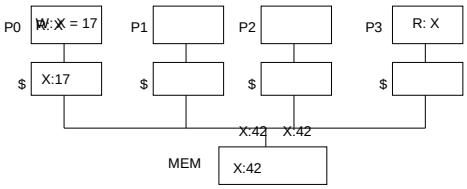
Virtual address spaces for a collection of processes communicating via shared addresses

Machine physical address space



62

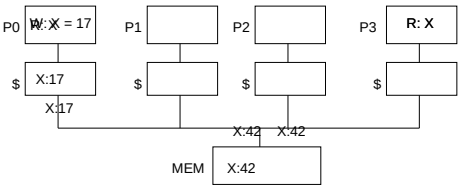
Cache Coherence Problem



- Processor 3 does not see the value written by processor 0

63

Write Through does not help



- Processor 3 sees 42 in cache (does not get the correct value (17) from memory.

64

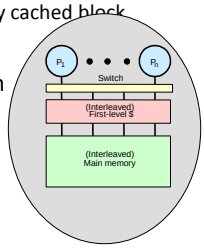
Why Don't Processors Share Cache

Advantages

- Cache placement identical to single cache
  - –only one copy of any cached block

Disadvantages

- Bandwidth limitation



Shared Cache

65

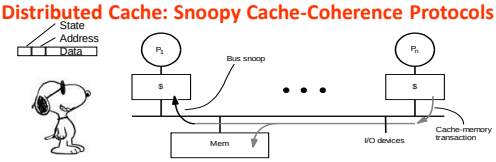
Cache Coherence

- Coherence
  - – All reads by any processor must return the most recently written value
  - – Writes to the same location by any two processors are seen in the same order by all processors
- Consistency
  - – When a written value will be returned by a read
  - – If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing Coherence

- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data
- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location
  - Snooping
    - Each core tracks sharing status of each block

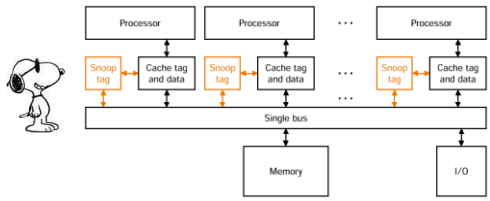
Distributed Cache: Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & caches know what they have
  - bus protocol: arbitration, command/addr, data
  - => Every device observes every transaction

68

Snooping Cache Coherency

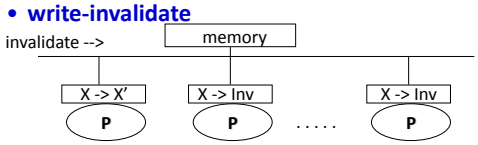


- Cache Controller "snoops" all transactions on the shared bus
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
    - take action to ensure coherence (invalidate, update, or supply value)

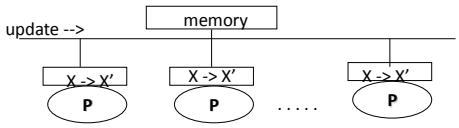
69

Hardware Cache Coherence

- **write-invalidate**



- **write-update** (also called distributed write)



An Example Snoopy Protocol

- Invalidation protocol, write-back cache
  - "invalidate" request on memory bus
- Each cache block is in one state (track these):
  - Shared : multiple caches potentially have copies of the block; the block can be read
  - OR Modified : this cache has the only copy of the block; the block is writeable and dirty
  - OR Invalid : the block contains no valid data

An Example Snoopy Protocol

Source	Request	State of block	Cache action	Function and explanation
CPU	Read hit	Shared, modified		
CPU	Read miss	Invalid		
CPU	Read miss	Shared		
CPU	Read miss	Modified		

Snoopy Coherence Protocols

- Write invalidate
  - On write, invalidate all other copies
  - Use bus itself to serialize
    - Write cannot complete until bus access is obtained

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

- Write update
  - On write, update all copies

An Example Snoopy Protocol

Source	Request	State of block	Cache action	Function and explanation
CPU	Write hit	Modified		
CPU	Write hit	Shared		
CPU	Write miss	Invalid		
CPU	Write miss	Shared		
CPU	Write miss	Modified		

An Example Snoopy Protocol

Source	Request	State of block	Cache action	Function and explanation
Bus	Read miss	Shared		
Bus	Read miss	Modified		
Bus	Invalidate	Shared		
Bus	Write miss	Shared		
Bus	Write miss	Modified		

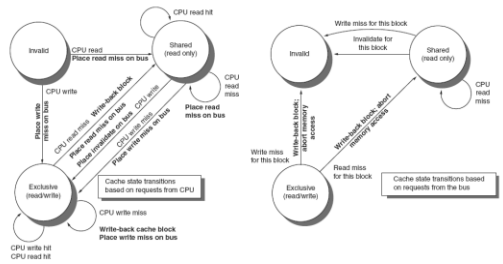
Snoopy Coherence Protocols

- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor
- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

Snoopy Coherence Protocols

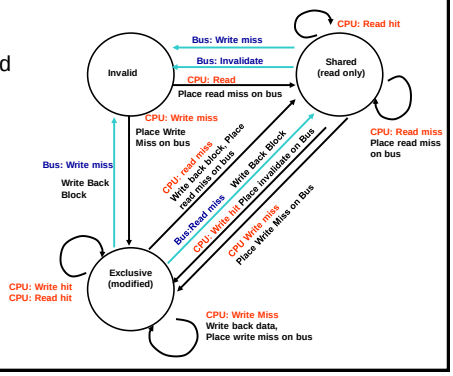
Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snoopy Coherence Protocols



Snoopy-Cache State Machine

- State machine for *CPU* requests for each cache block and for *bus* requests for each cache block



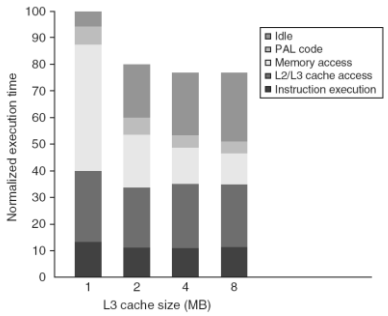
Snoopy Coherence Protocols

- Complications for the basic MSI protocol:
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
  - Add exclusive state to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned state

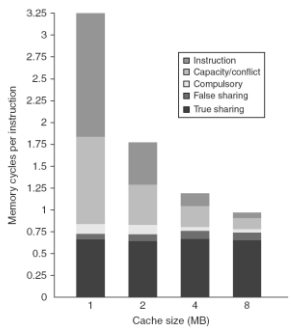
Performance

- Coherence influences cache miss rate
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block
    - False sharing misses
      - Read an unmodified word in an invalidated block

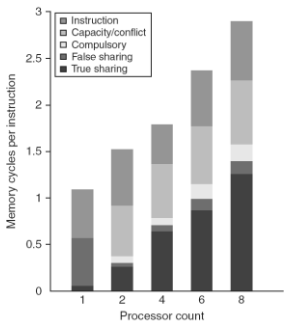
Performance Study: Commercial Workload



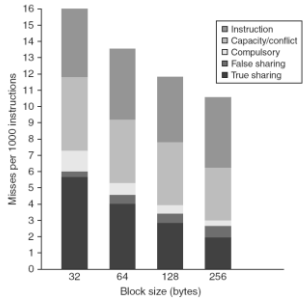
Performance Study: Commercial Workload



Performance Study: Commercial Workload



Performance Study: Commercial Workload

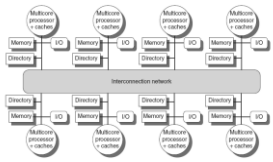


Revisit: Coherency Solutions

- Snooping Solution (Snoopy Bus):
  - Send all requests for data to all caches
  - Requires broadcast, works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)
- Directory-Based Schemes
  - Keep track of what is being shared in 1 centralized place (logically)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping

Directory Protocols

- Directory keeps track of every block
  - Which caches have each block
  - Dirty status of each block
- Implement in shared L3 cache
  - Keep bit vector of size = # cores for each block in L3
  - Not scalable beyond shared L3
- Implement in a distributed fashion:



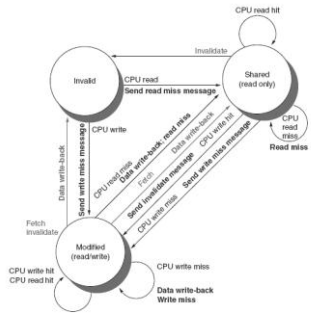
Directory Protocols

- For each block, maintain state:
  - Shared
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - Uncached
  - Modified
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID
- Directory maintains block states and sends invalidation messages

Messages

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Directory Protocols



Directory Protocols

- For uncached block:
  - Read miss
    - Requesting node is sent the requested data and is made the only sharing node, block is now shared
  - Write miss
    - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
  - Read miss
    - The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss
    - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory Protocols

- For exclusive block:
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Synchronization

- Basic building blocks:
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - Requires memory read and write in uninterruptable instruction
  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Implementing Locks

- Spin lock
  - If no coherence:

```
                DADDUI R2,R0,#1
lockit:         EXCH   R2,0(R1);atomic exchange
                BNEZ   R2,lockit;already locked?
```

  - If coherence:

```
lockit:         LD     R2,0(R1);load of lock
                BNEZ   R2,lockit;not available-spin
                DADDUI R2,R0,#1        ;load locked value
                EXCH   R2,0(R1);swap
                BNEZ   R2,lockit;branch if lock wasn't 0
```

Implementing Locks

Advantage of this scheme: reduces memory traffic

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Models of Memory Consistency

Processor 1:	Processor 2:
A=0	B=0
…	…
A=1	B=1
if (B==0) …	if (A==0) …

- Should be impossible for both if-statements to be evaluated as true
  - Delayed write invalidate?
- Sequential consistency:
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Implementing Locks

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
  - Reduces performance!
- Alternatives:
  - Program-enforced synchronization to force write on processor to occur before read on the other processor
    - Requires synchronization object for A and another for B
      - "Unlock" after write
      - "Lock" after read

Relaxed Consistency Models

- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - "Total store ordering"
  - Relax W → W
    - "Partial store order"
  - Relax R → W and R → R
    - "Weak ordering" and "release consistency"

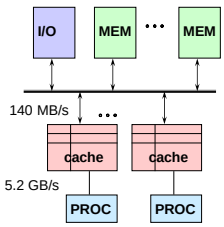
Relaxed Consistency Models

- Consistency model is multiprocessor specific
- Programmers will often implement explicit synchronization
- Speculation gives much of the performance advantage of relaxed models with sequential consistency
  - Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery

Interconnect

100

Limits of Bus-Based Shared Memory



Assume:

1 GHz processor

=> 4 GB/s inst BW per processor (32-bit)

=> 1.2 GB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate

=> 80 MB/s inst BW per processor

=> 60 MB/s data BW per processor

⇒140 MB/s combined BW

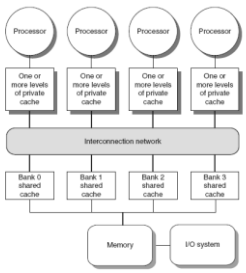
Assuming 1 GB/s bus bandwidth

∴ 8 processors will saturate the memory bus

101

Coherence Protocols: Extensions

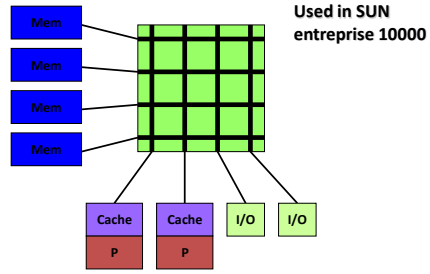
- Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors
  - Duplicating tags
  - Place directory in outermost cache
  - Use crossbars or point-to-point networks with banked memory



Coherence Protocols

- AMD Opteron:
  - Memory directly connected to each multicore chip in NUMA-like organization
  - Implement coherence protocol using point-to-point links
  - Use explicit acknowledgements to order operations

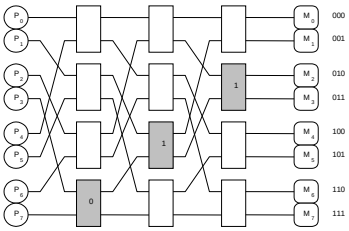
Scalable Shared Memory Architectures Crossbar Switch



104

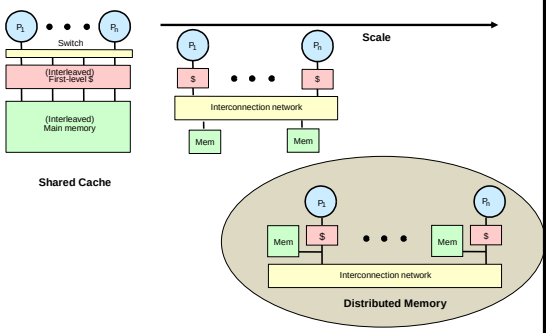
Scalable Shared Memory Architectures

- Used in IBM SP Multiprocessor



105

Approaches to Building Parallel Machines



106

Scales of Multiprocessors

- Small size multiprocessors (< 10 processors)
  - Use shared memory with shared bus
  - Not expensive
  - Commercially available, and highly used as small servers
- Medium size multiprocessors (< 64 processors)
  - Use shared memory with crossbar switch
  - Commercially available
  - Used as high-end servers and computing engines
- Large size multiprocessors (> 64 processors)
  - Use distributed memory with custom-made interconnection network
  - Very powerful computing machines
  - **Extremely expensive**