Concurrent Systems (ComS 527)

Ali Jannesari

Department of Computer Science

Iowa State University, Spring 2023

PARALLEL ARCHITECTURES

Outline

- Classification
- Memory architecture
- Interconnection networks
- Examples
    - Condo or Pronto Cluster
    - Nova Cluster (HPC-class cluster)
- Cache coherence (self-study)
- Memory consistency (self-study)
- Synchronization

Taxonomies

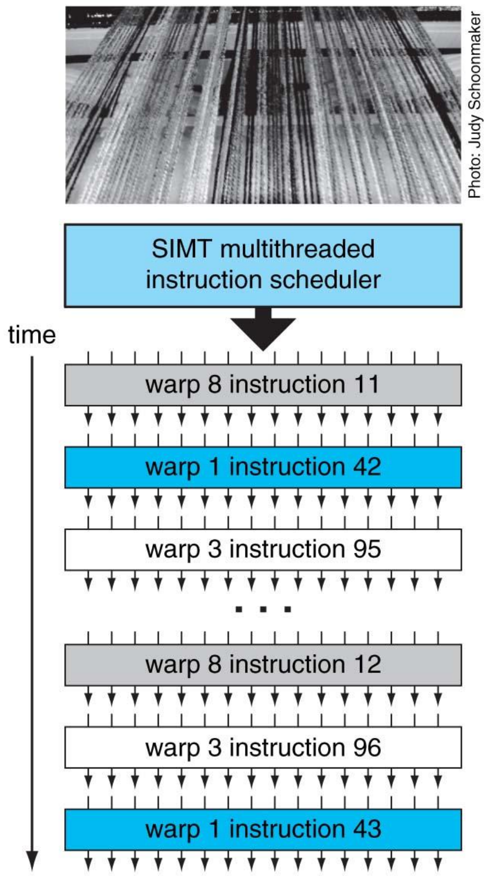
- Number of instruction streams vs. number of data streams
- Memory architecture
- Network architecture
- Degree of heterogeneity
- Degree of customization

Flynn’s classification [1966]

		#Instruction streams	
		Single	Multiple
#Data streams	Single	SISD • Classical uniprocessor	MISD • No commercial multiprocessor of this type ever built
	Multiple	SIMD • Same instruction is executed by multiple processors using different data streams • Data parallelism • Examples: SIMD extensions for multimedia, vector processors	MIMD • Each processor fetches its own instructions and operates on its own data • Thread-level parallelism

Single-instruction multiple threads (SIMT) - Used on GPUs -

- Creates, manages, schedules, and executes threads in groups of parallel threads called warps
- At each instruction issue time, SIMT instruction unit
  - Selects warp that is ready to execute its next instruction
  - Broadcasts instruction to all active threads of that warp
- Individual threads may be inactive to do independent branching

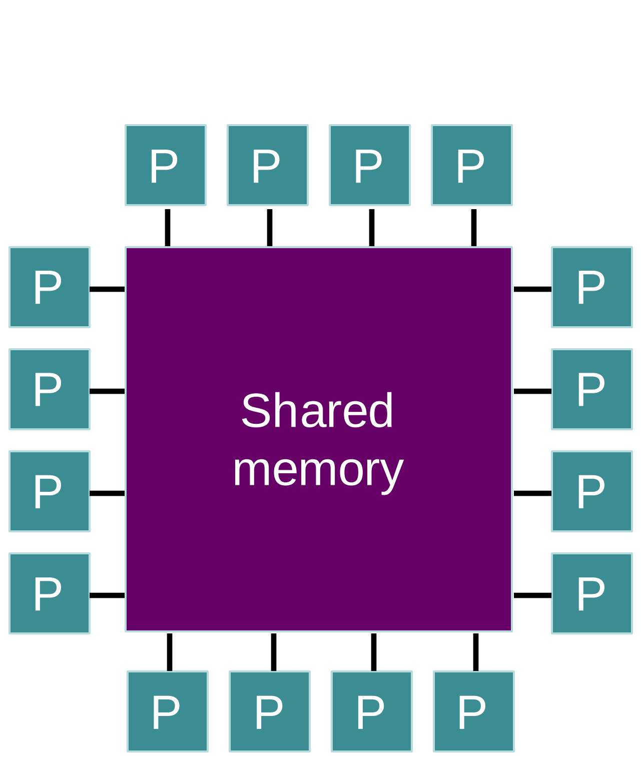


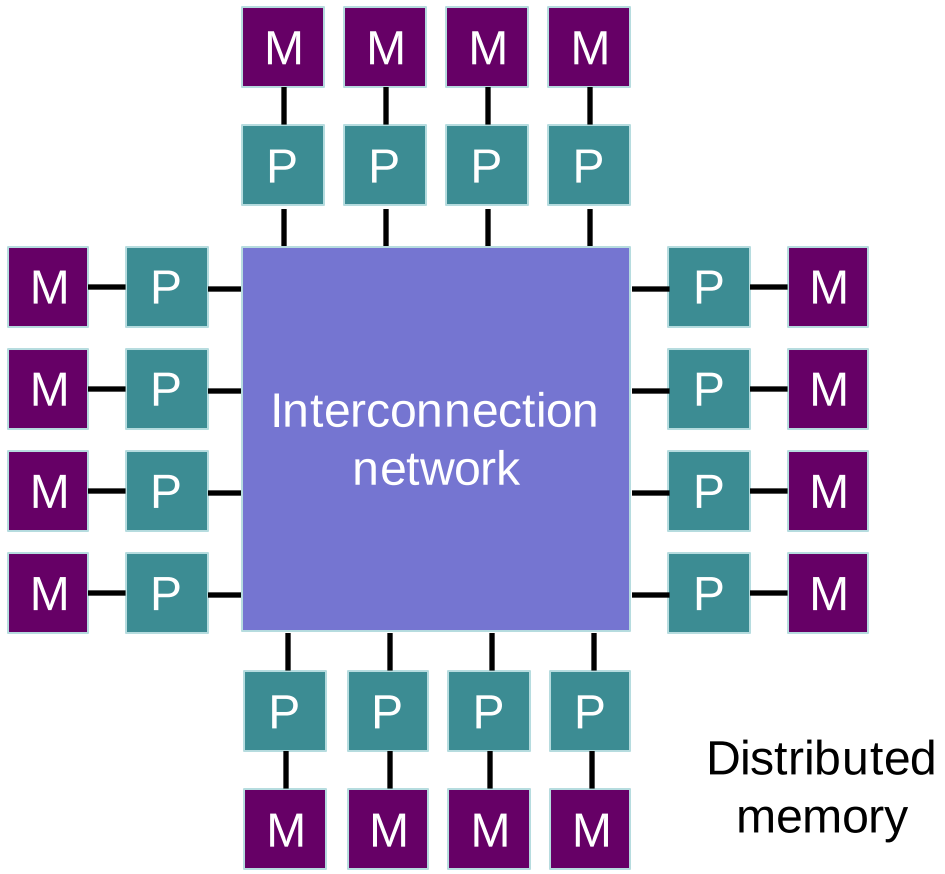
Source: Patterson, Hennessy: Computer Organization & Design, 4th edition, Morgan Kaufmann

MIMD

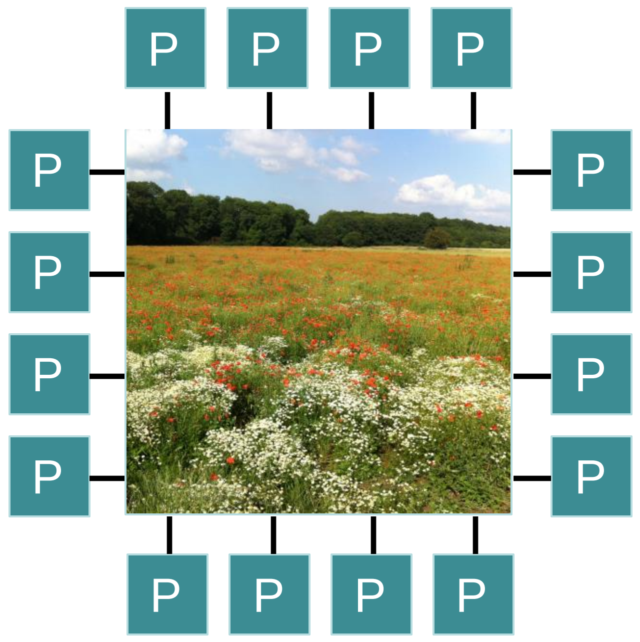
- Architecture of choice for general-purpose multiprocessors
- Offers high degree of flexibility
  - High performance for one application or multi-programmed multiprocessor
- Can take advantage of off-the-shelf processors
- Popular execution model - Single Program Multiple Data (SPMD)
  - The same program is executed in parallel with each instance having a potentially different control flow

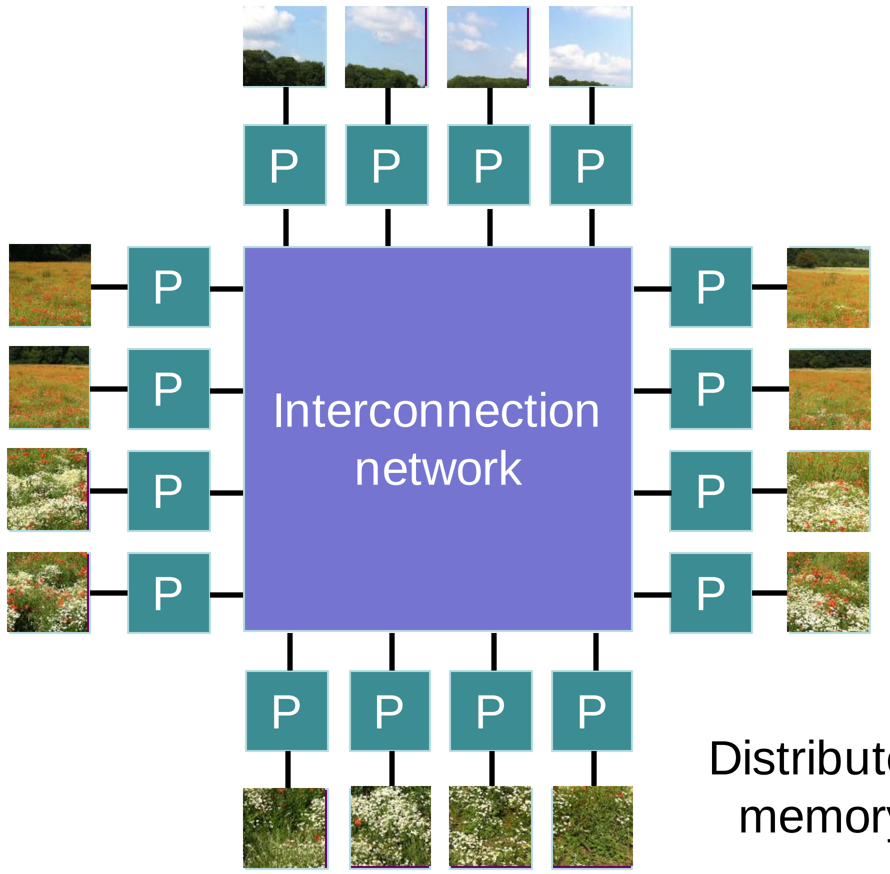
Memory architecture





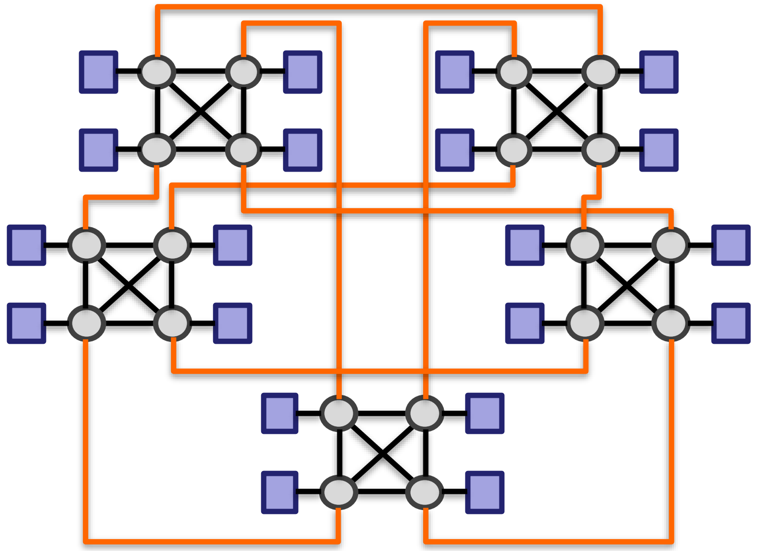
Memory architecture

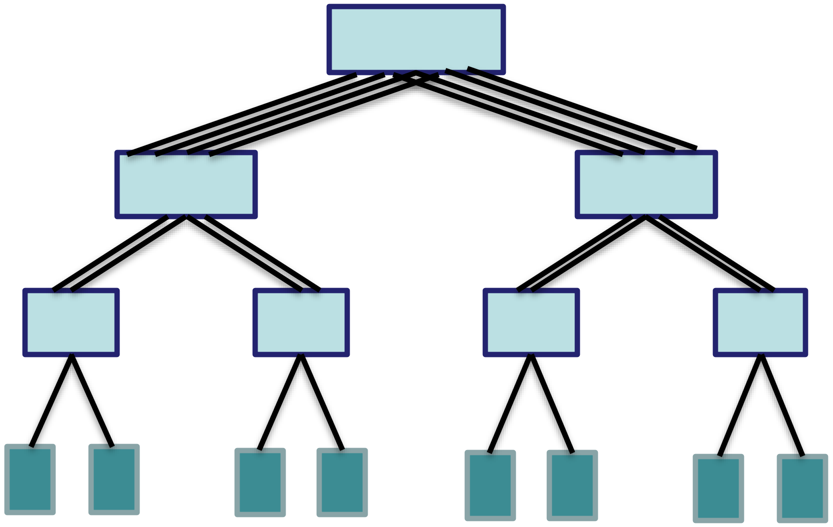




Distributed memory

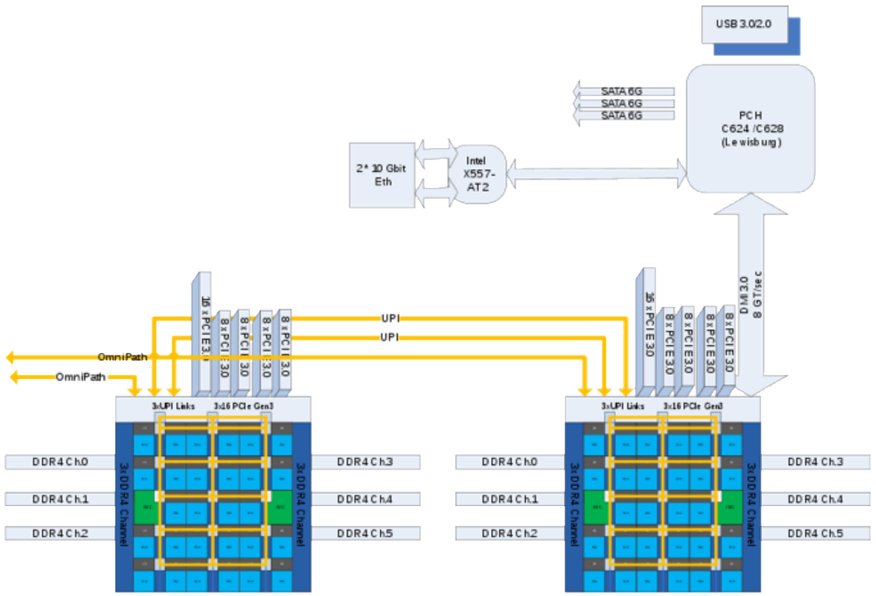
Popular network architectures for distributed memory systems


Dragonfly
(distributed switched network)


Fat tree
(centralized switched network)

Degree of heterogeneity

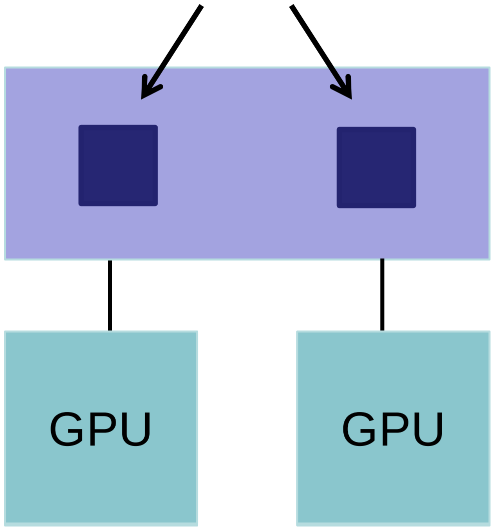
Homogeneous node architecture



Intel Skylake dual-socket system

Michael Wandinger - Eigenes Werk
CC BY-SA 3.0 de, https://commons.wikimedia.org/w/index.php?curid=63346901

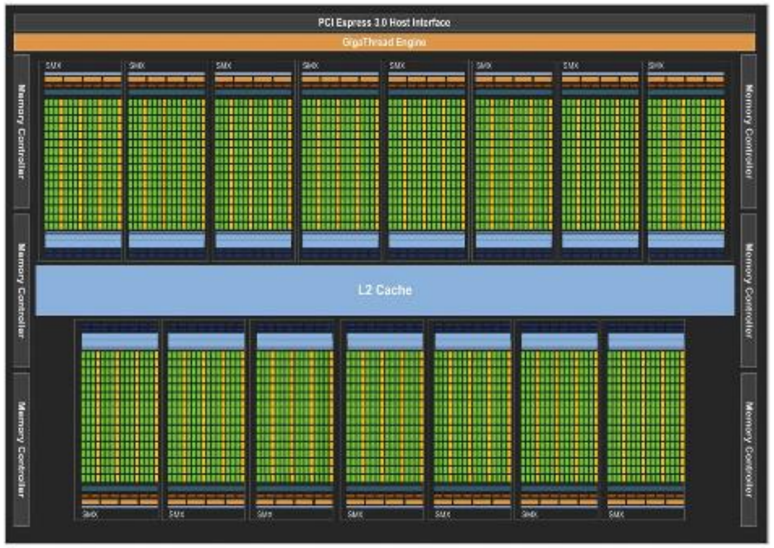
Classic server CPUs (e.g. Intel Xeon)



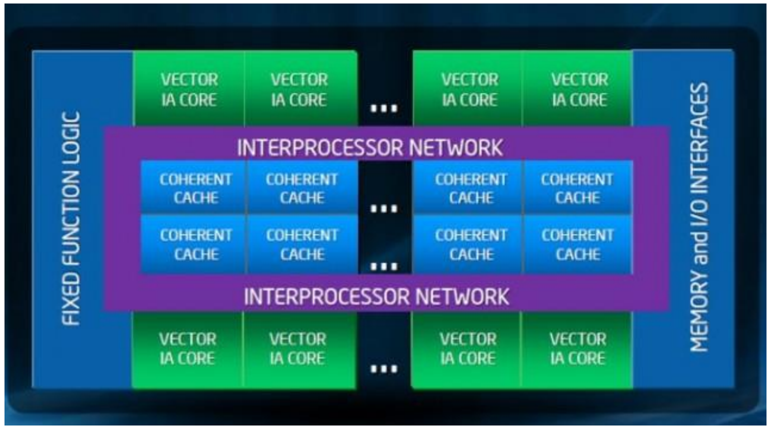
Accelerators

Heterogeneous node architecture

Accelerators



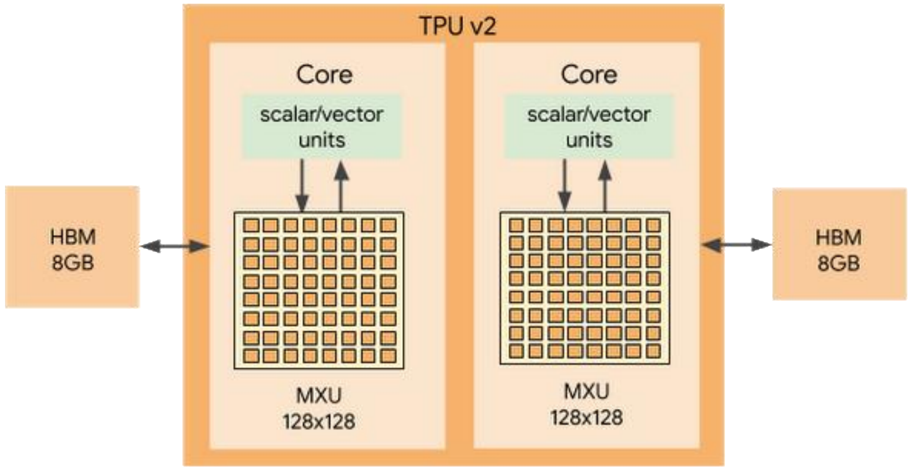
NVIDIA Kepler GPU

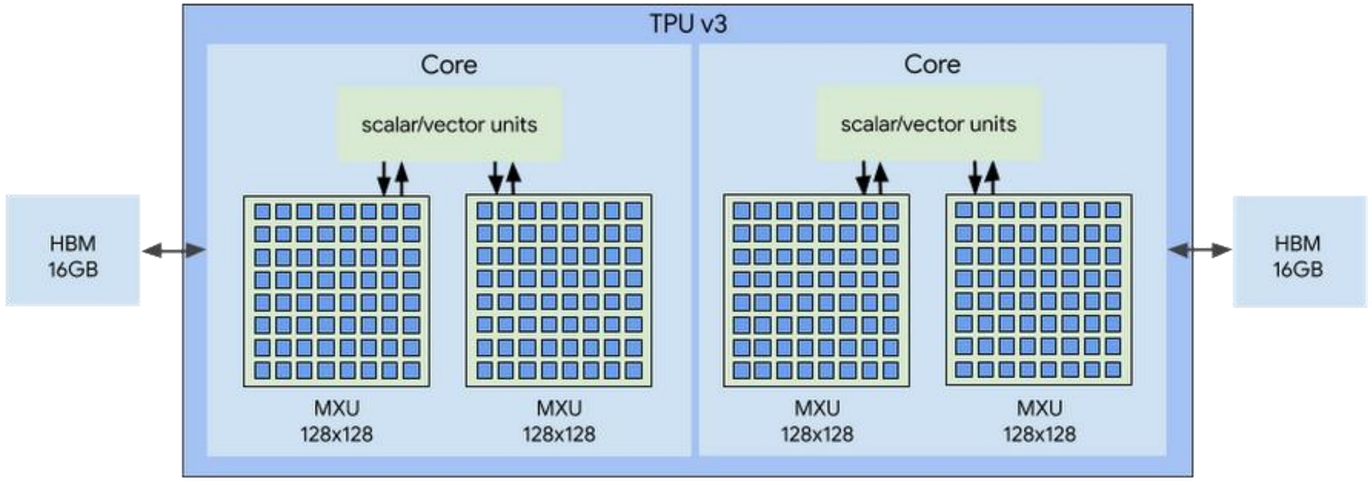


Intel Xeon Phi

Accelerators (2)

Google TPU: A Neural Network Accelerator Chip





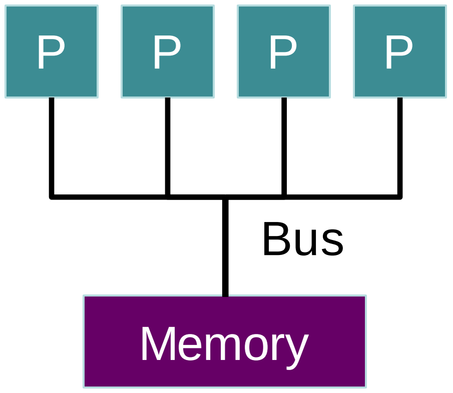
Degree of customization

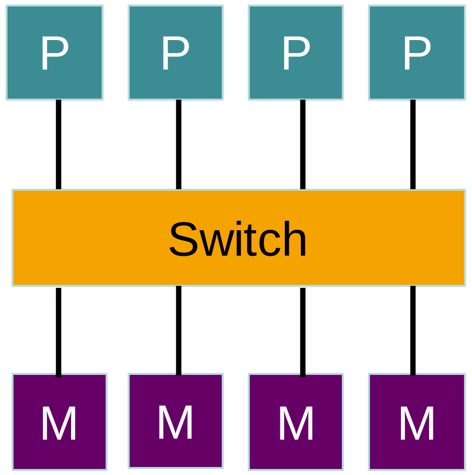
- Commodity clusters – standard nodes and standard network
  - Focus on applications with small communication requirements
  - Example: Beowulf cluster
- Custom clusters – custom nodes and custom network
  - Also called massively parallel processors
  - Focus on applications that exploit large amounts of parallelism on single problem
  - Example: IBM Blue Gene/Q, Summit (ORNL)
- Above classes are extremes of a broad spectrum

Shared memory

UMA (Uniform memory access)

- Each CPU has same access time to each memory address
- Simple design but limited scalability (multicore or less)

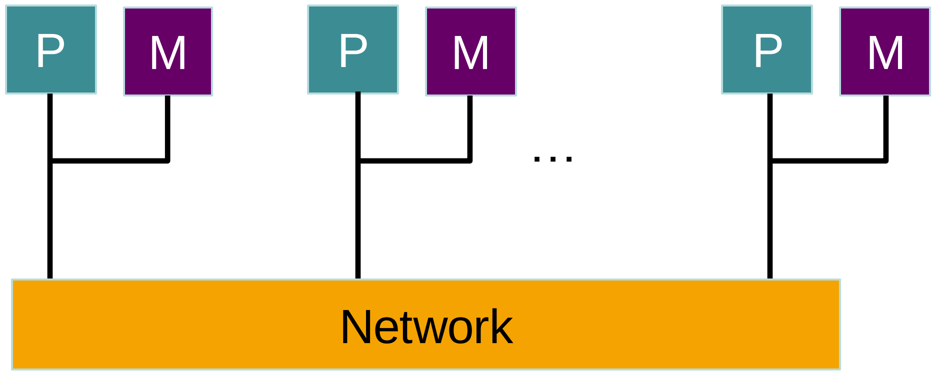




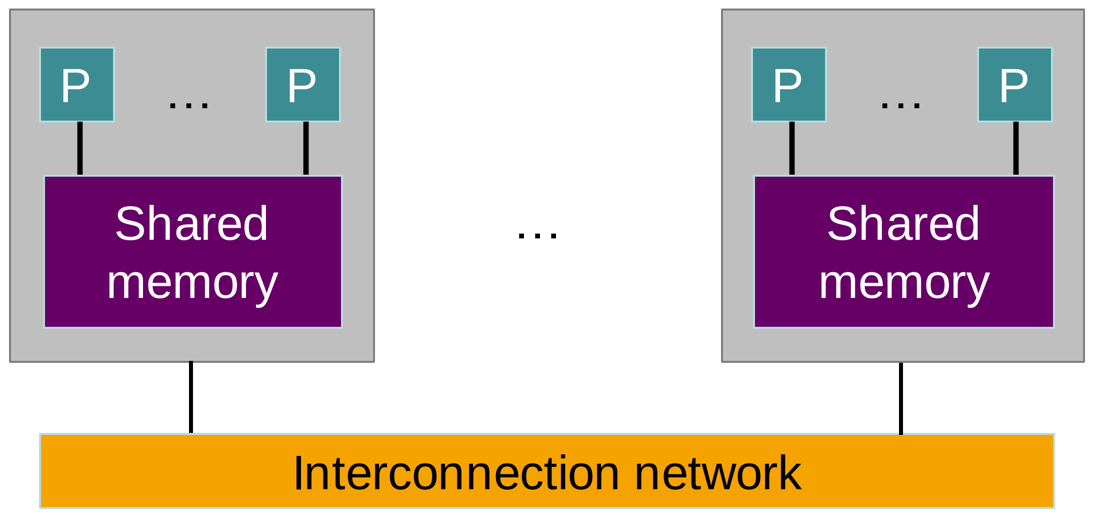
Shared memory (2)

NUMA (Non-uniform memory access)

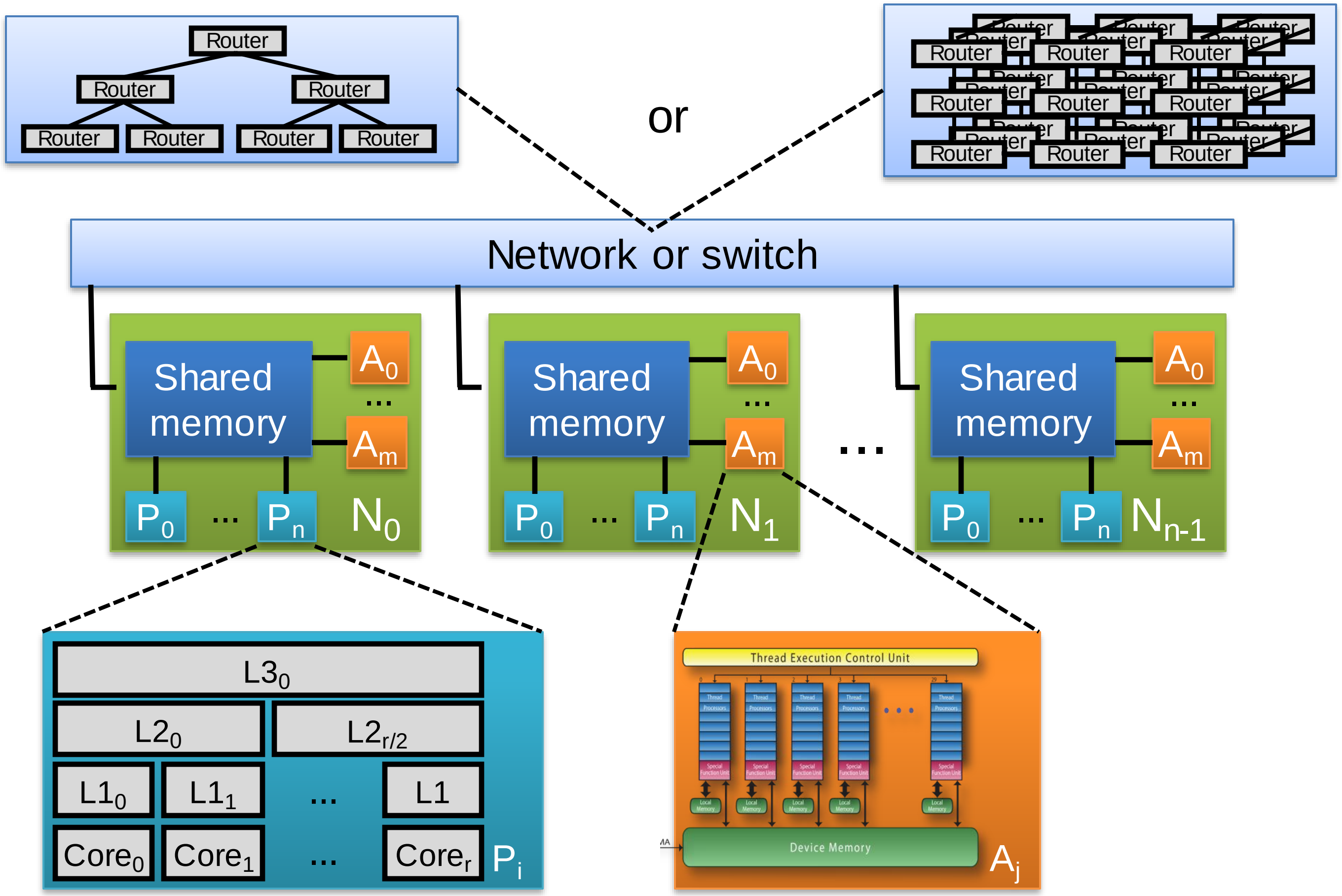
- Memory has affinity to a processor
- Access to local memory faster than to remote memory
- Harder to program but more scalable



Distributed memory (aka multicomputer)



Typical cluster architecture



Interconnection network

Physical link between components of a parallel system

- Between processors and memory
- Between nodes

Communication via exchange of messages

- Example: intermediate results, memory requests

Design elements

- Topology – determines geometric layout of links and switches
- Routing technique – determines paths of messages through network

Network performance

Bandwidth

- Maximum rate at which information can be transferred
- Aggregate bandwidth – total data bandwidth supplied by network
- Effective bandwidth or throughput – fraction of aggregate bandwidth delivered to an application

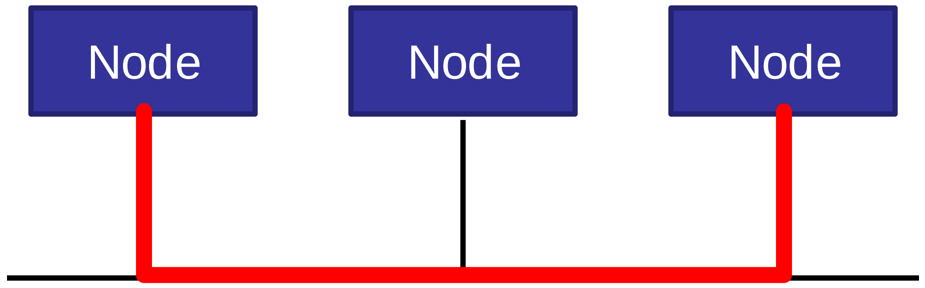
Latency

- Sending overhead + time of flight + $\dfrac{\text{Packet size}}{\text{Bandwidth}}$ + receiving overhead

Time for the first bit of the packet to arrive

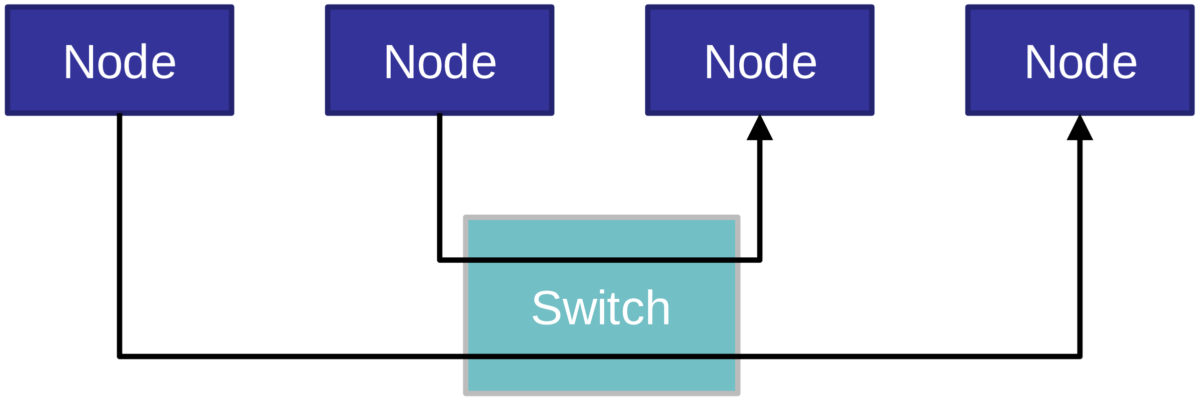
Shared-media networks

- Only one message at a time – processors broadcast their message over the medium
- Each processor “listens” to every message and receives the ones for which it is the destination
- Decentralized arbitration
  - Before sending a message, processors listen until medium is free
  - Message collision can degrade performance
- Low cost but not scalable
- Example – bus networks to connect processors to memory



Switched-media networks

- Support point-to-point messages between nodes
- Each node has its own communication path to the switch
- Advantages
    - Support concurrent transmission of multiple messages among different node pairs
    - Scale to very large numbers of nodes



Centralized switched networks

- Also called *indirect* or dynamic interconnection networks
- Connect processors / memory indirectly using several links and intermediate switches
- Examples: switching networks
- Used both for shared- and distributed-memory architectures

Crossbar switch

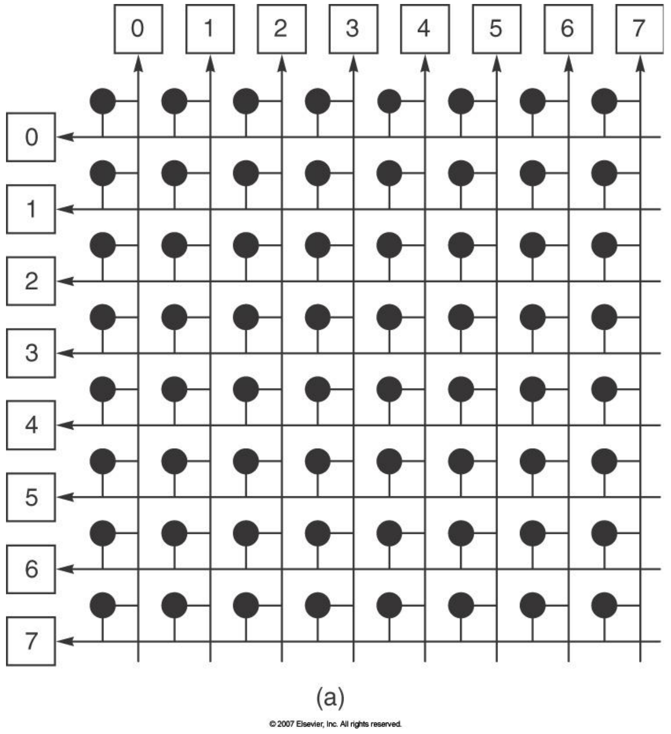
every node of the network
connected to every other node.

Non-blocking

- Links are not shared among paths to unique destinations

Requires N^2 crosspoint switches

- Limited scalability

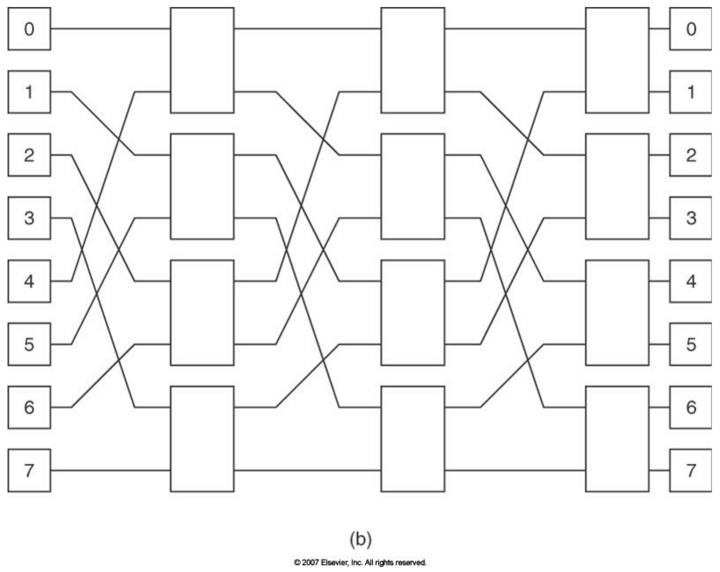


Source: Hennessy, Patterson: Computer Architecture, 4th edition, Morgan Kaufmann

Multistage interconnection network (MIN)

Example: Omega network

- Complexity O(N log N)
- Perfect shuffle permutation at each stage
- Blocking due to paths between different sources and destinations simultaneously sharing network links
- Omega with k x k switches
  - $\log_k N$ stages ; $N/k \log_k N$ switches

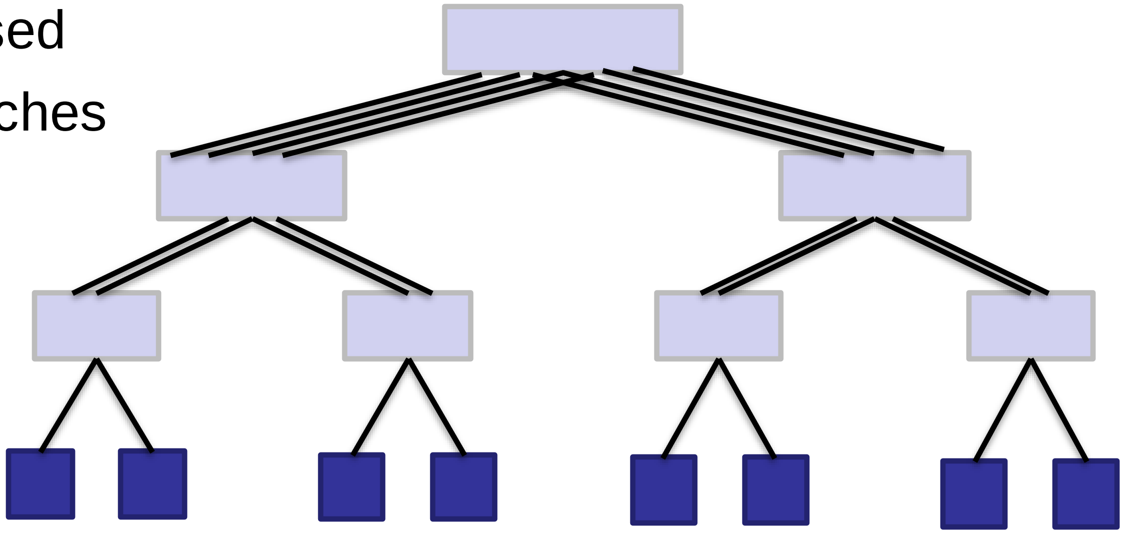


Source: Hennessy, Patterson: Computer Architecture, 4th edition, Morgan Kaufmann

MINs can be extended to rearrangeably non-blocking topologies

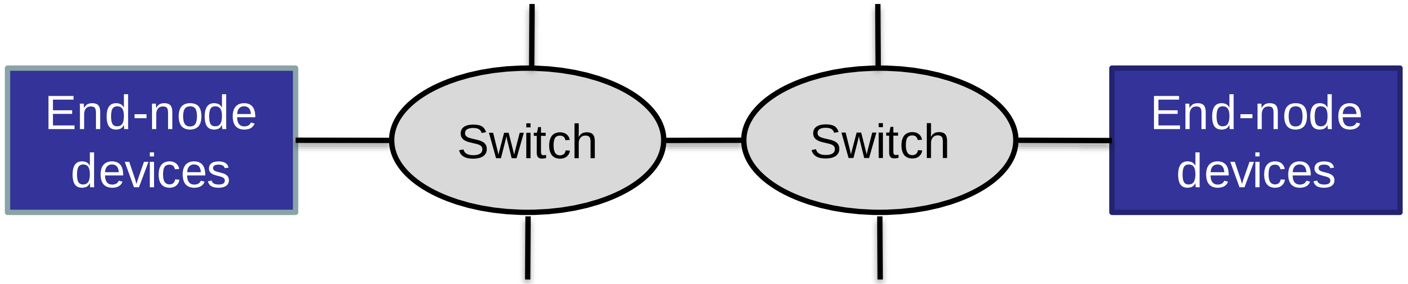
Fat tree

- Balanced tree where
    - Leaves = end node devices
    - Vertices = switches
- Total link bandwidth constant across all levels
- Switches often composed of multiple smaller switches
- Popular topology for cluster interconnects



Distributed switched networks

- Each network switch has one or more end node devices directly attached to it
- End node devices = processor(s) + memory
  - Directly connected to other nodes without going through external switches
  - Mostly used for distributed-memory architectures
- Also called *direct* or static interconnection networks
- Ratio of switches to nodes = 1:1



Evaluation criteria

Network degree

- Maximum node degree
- Node degree = number of adjacent nodes = (incoming + outgoing) edges

Diameter

- Largest distance between two nodes

Bisection width

- Minimum number of edges between nodes that must be removed to cut the network into two roughly equal halves
- Bisection bandwidth = bandwidth [bytes/s] between the two parts

Edge / node connectivity

- Minimum number of edges / nodes that need to be removed to render network disconnected

Embedding

- Mapping of one network onto another

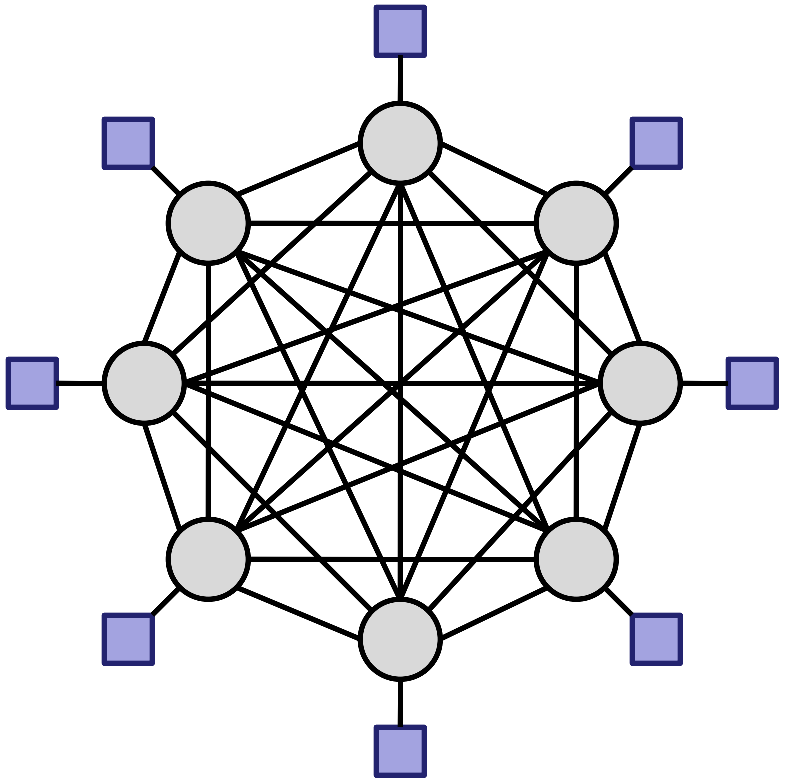
Requirements

- Low network degree to reduce hardware costs
- Low diameter to ensure low distance (i.e., latency) for message transfer
- High bisection bandwidth to ensure high throughput
- High connectivity to ensure robustness
- Option to embed many other networks to ensure flexibility

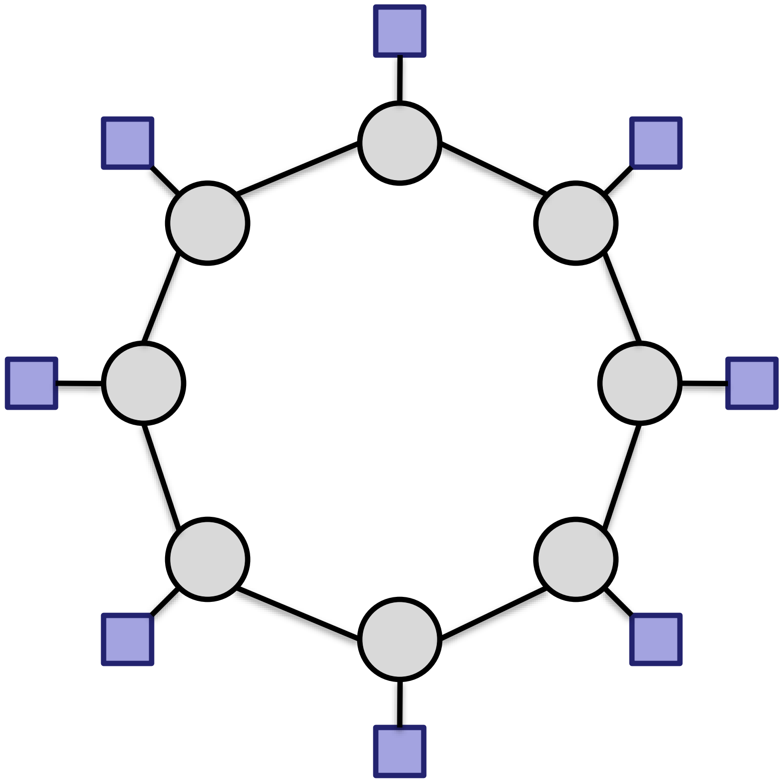
Often conflicting goals

Fully connected topology

- Each node is directly connected to every other node
- Expensive for large numbers of nodes
- Dedicated link between each pair of nodes
- Cheaper alternative: crossbar topology

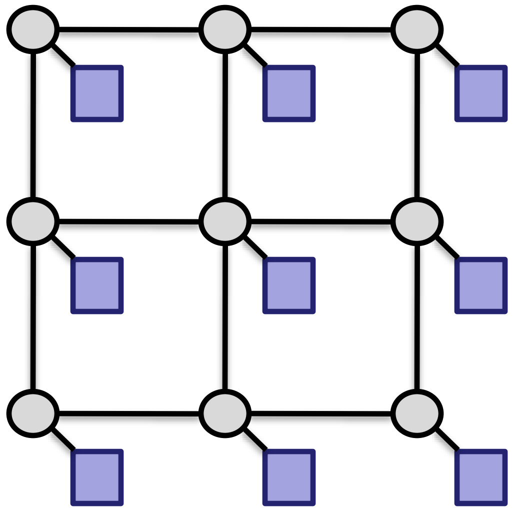


Ring topology



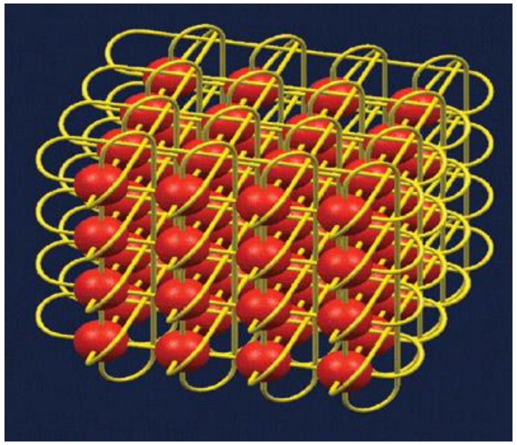
N-dimensional meshes

- Direct link to neighbors
- Each node has 1 or 2 neighbors per dimension
  - 2 in the center
  - Less for border or corner nodes
- Efficient nearest neighbor communication
- Suitable for large numbers of nodes

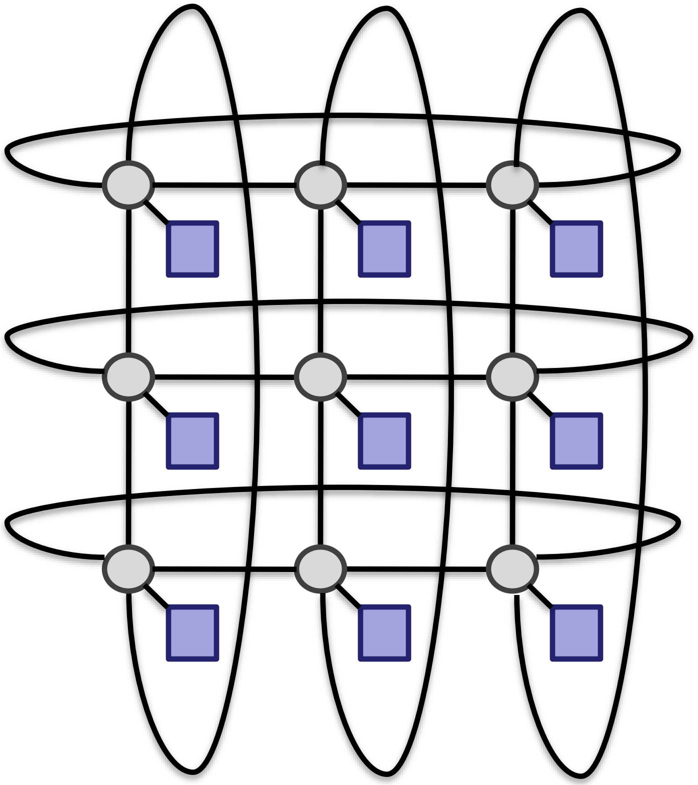

2D mesh

Torus

- Mesh with wrap-around connections
- Each node has exactly 2 neighbors per dimension
- Typically 3-6 dimensions



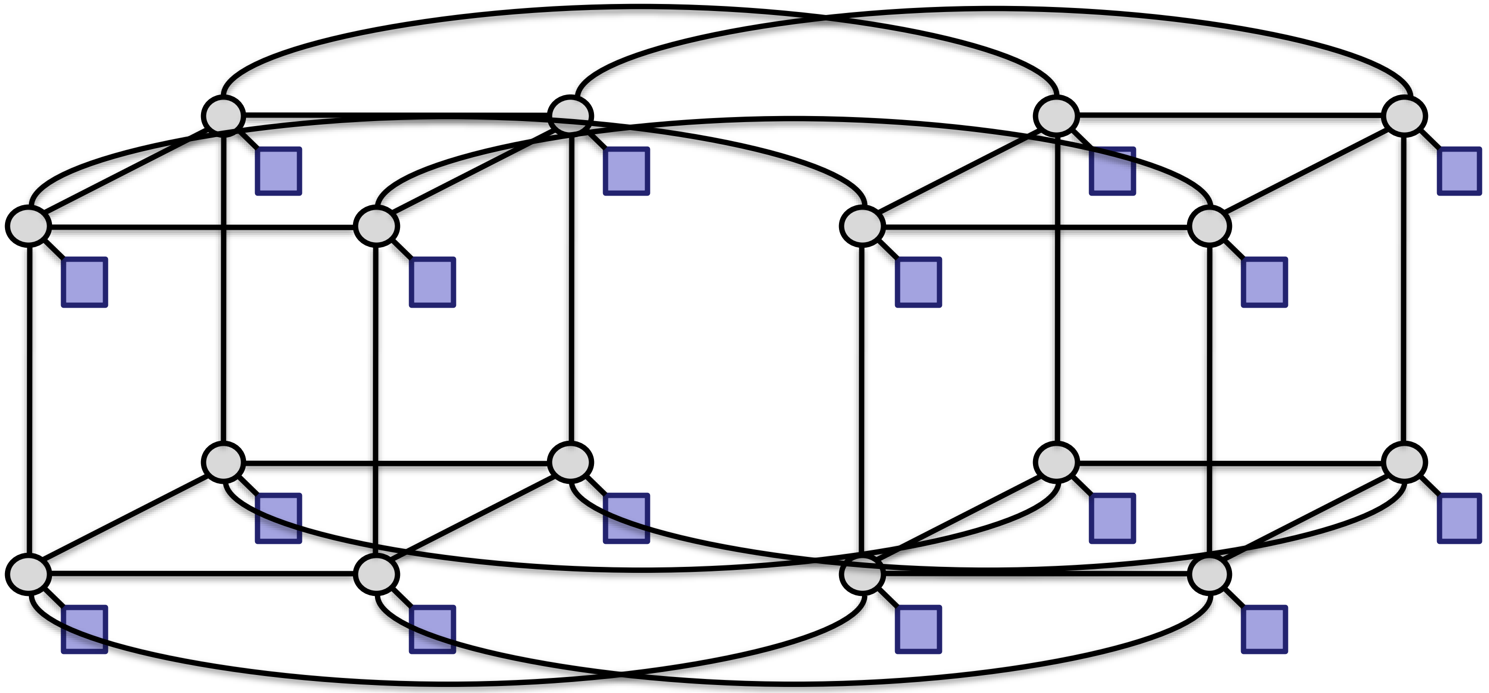
3D torus



2D torus

Hypercube

16 nodes
(16 = 2^4 so n = 4)

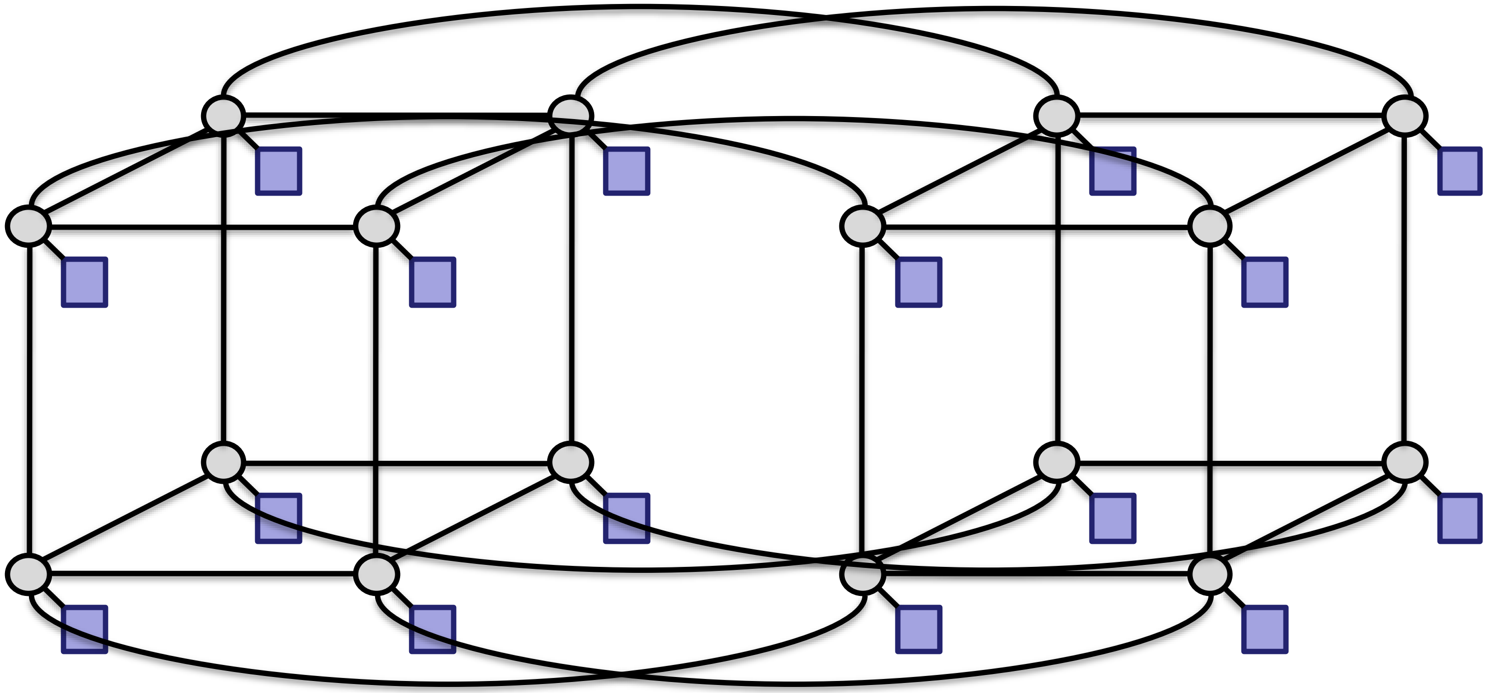


Each node has one connection along each dimension (n = #dimensions)

Usually better connectivity than tori at the expense of higher link and switch costs

Hypercube

16 nodes
(16 = 2^4 so n = 4)

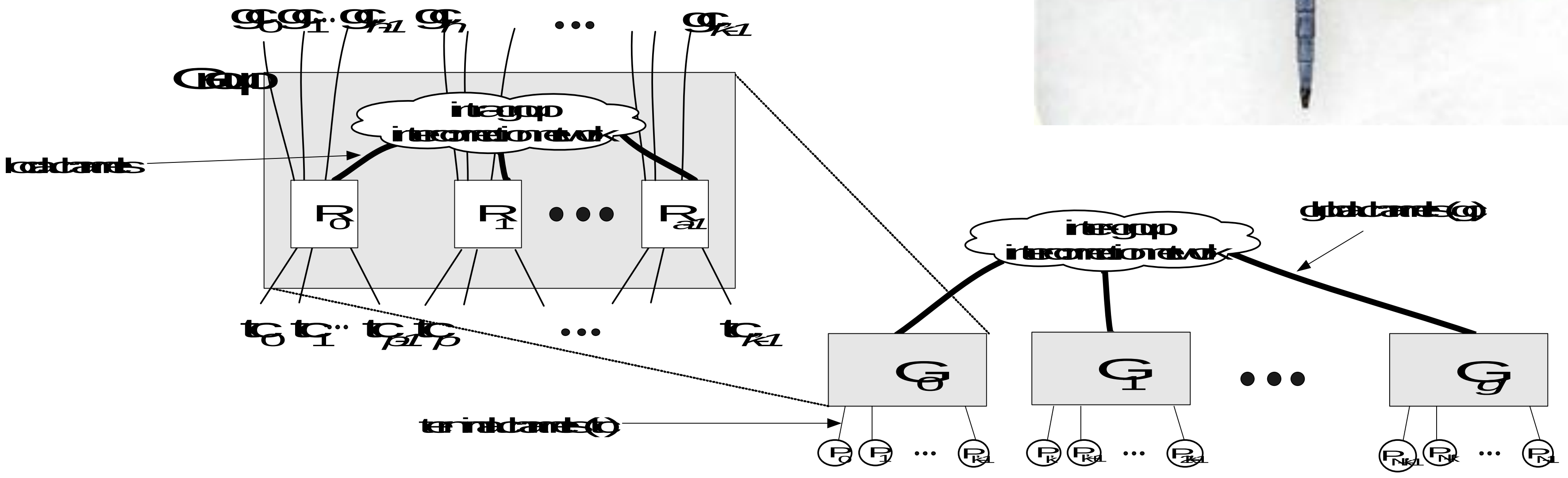


Each node has one connection along each dimension (n = #dimensions)

Usually better connectivity than tori at the expense of higher link and switch costs

Dragonfly

- Enabled by high-radix switches and long-distance optical signaling technology

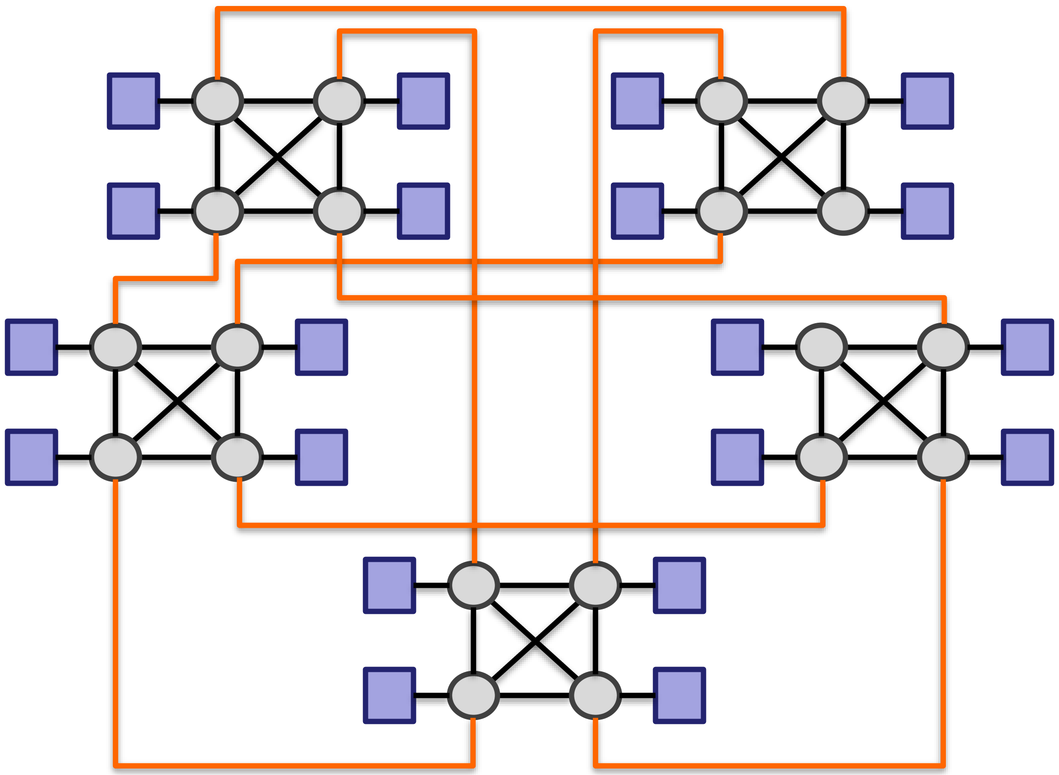


gc0 gc1 ... gch-1 gch ... gck-1
Group
intra-group interconnection network
local channels
R0
R1
Ra-1
tc0 tc1 ... tcp-1 tcp ... tck-1
global channels (gc)
inter-group interconnection network
G0
G1
Gg
terminal channels (tc)
P0 P1 ... Pk-1
Pk Pk+1 ... P2k-1
PN-k-1 PN-k ... PN-1

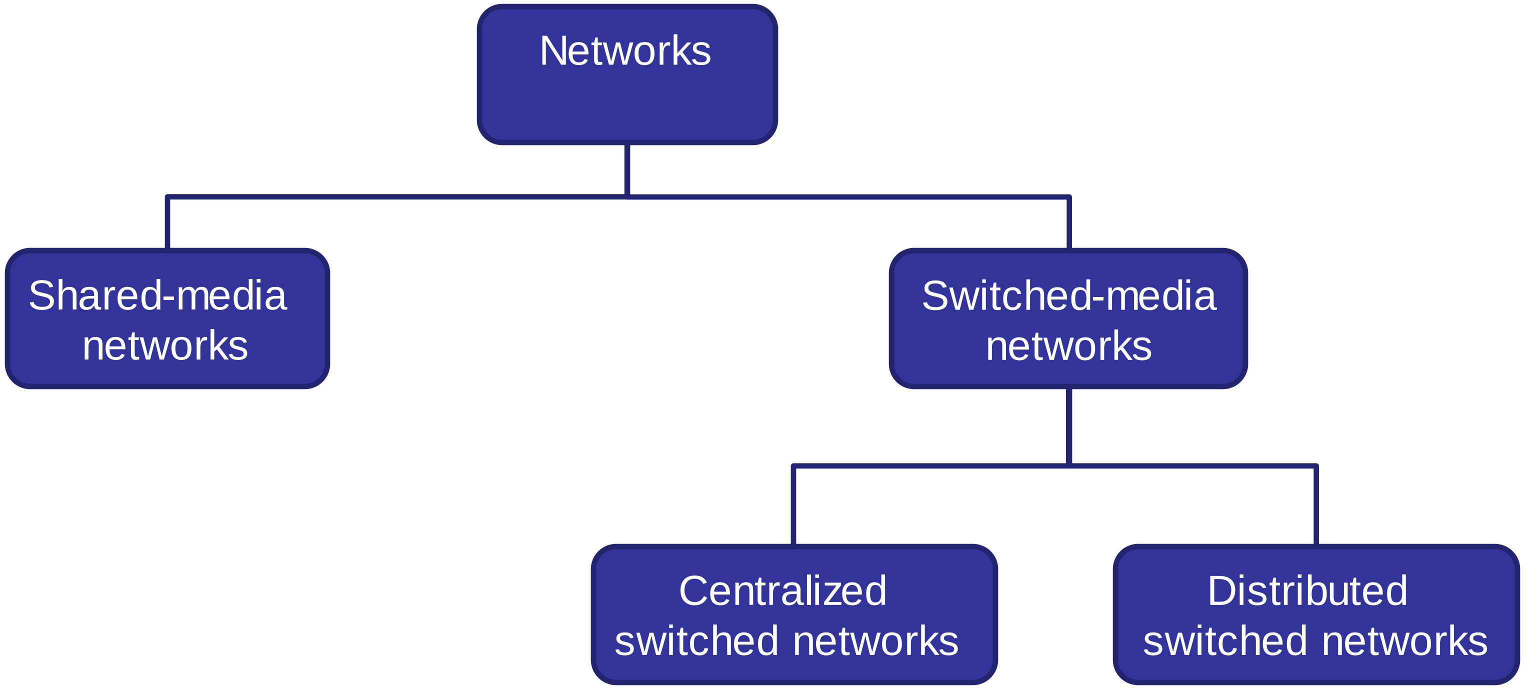
Source: J. Kim, W. Dally, S. Scott, and D. Abts, “Technology-driven, highly- scalable dragonfly topology”. *In Proc. of the 35th International Symposium on Computer Architecture (ISCA)*, June 2008, pp. 77–88.

Dragonfly (2)

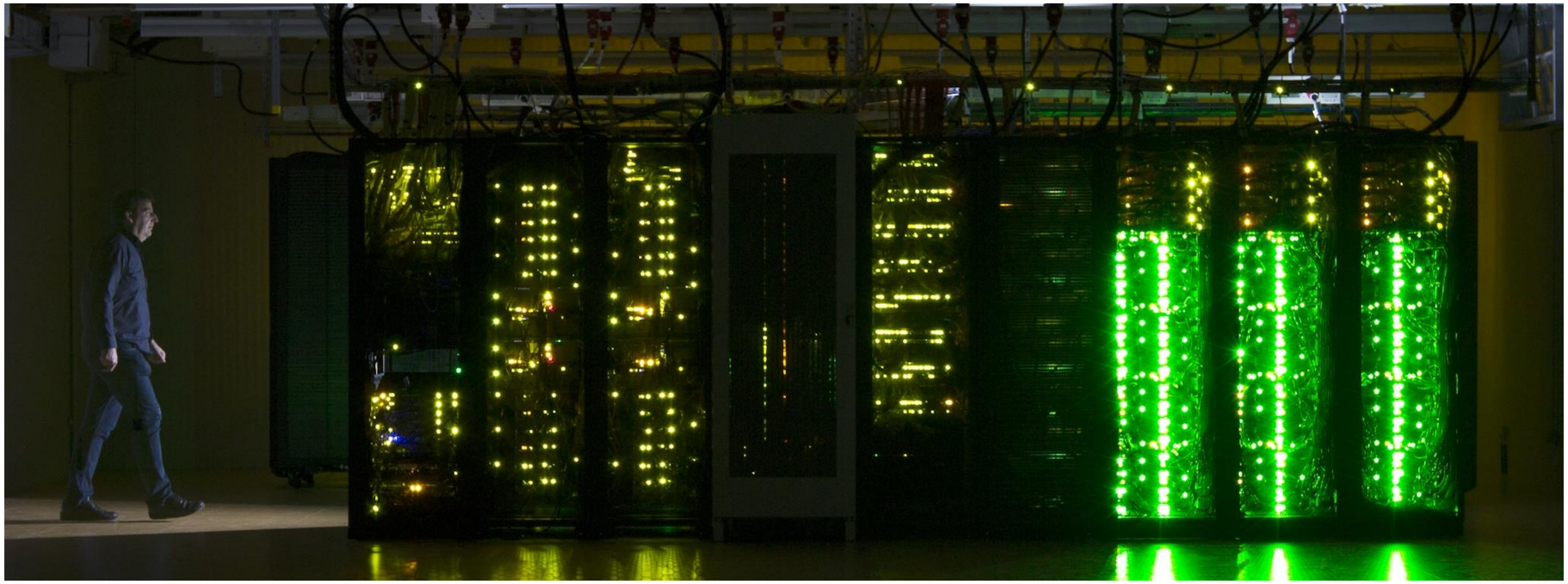
- Arbitrary networks possible for intra-group and inter-group networks
  - Example: fully connected networks for both



Network classes



Example of a cluster



Condo @ ISU

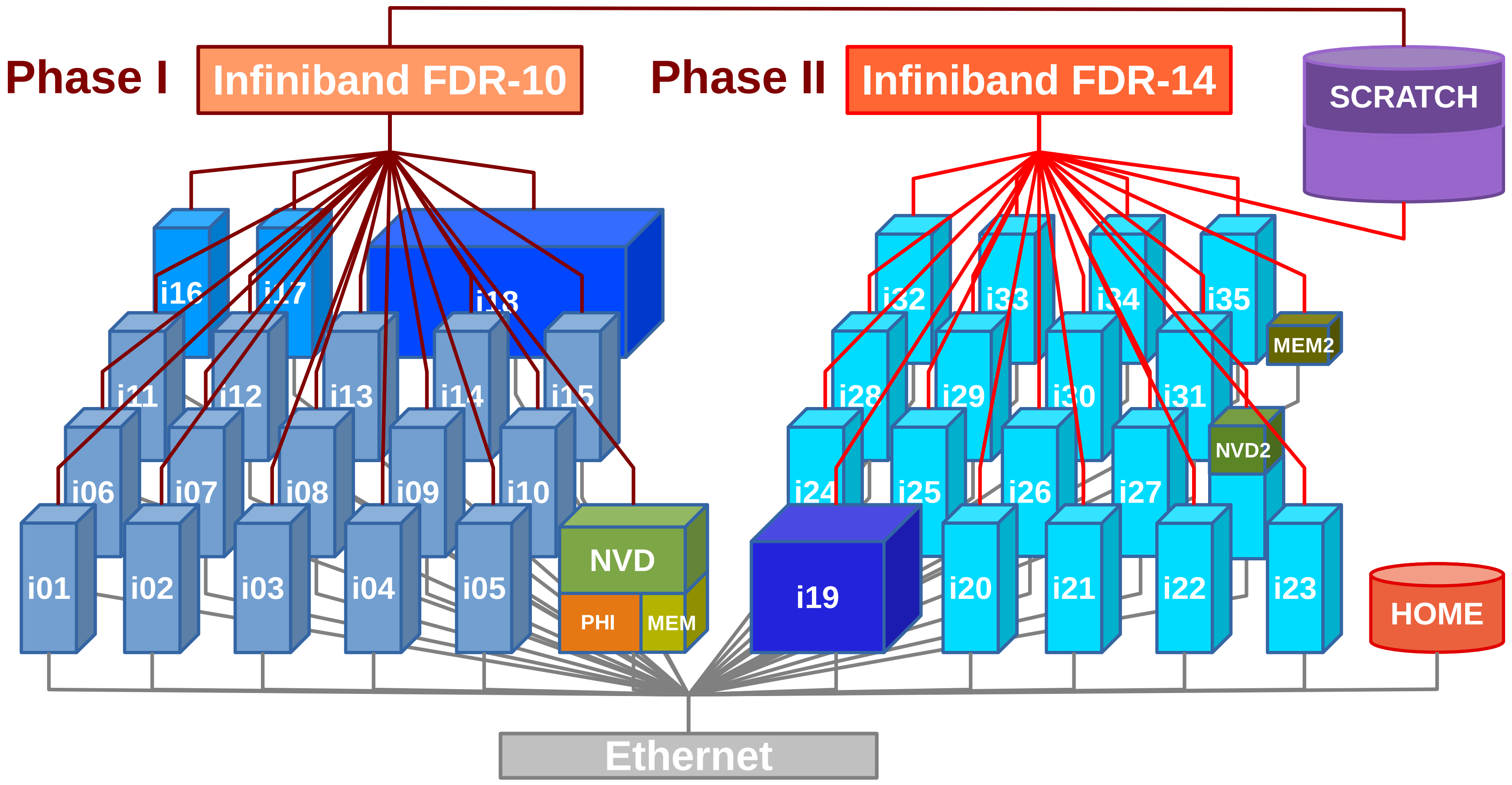
- **Condo @ ISU:** https://www.hpc.iastate.edu/guides/condo-2017
- **Cluster HPC-Class**: https://www.hpc.iastate.edu/guides/classroom-hpc-cluster
  - 48 SuperMicro servers each with 16 cores, 64 GB of memory, GigE and QDR (40Gbit) Infiniband interconnects
  - Four of these compute nodes contain an NVIDIA Tesla K20 GPU, and four other nodes contain 60-core Intel Xeon Phi Accelerator card

Number of Nodes	Processors per Node	Cores per Node	Memory per Node	Interconnect	Local $TMPDIR Disk	Accelerator Card
40	Two 2.0 GHz 8-Core Intel E5 2650	16	64 GB	40G IB	2.5 TB	N/A
4	Two 2.0 GHz 8-Core Intel E5 2650	16	64 GB	40G IB	2.5 TB	NVIDIA K20
4	Two 2.0 GHz 8-Core Intel E5 2650	16	64 GB	40G IB	2.5 TB	60 core Intel Phi 5110P

Nova @ ISU

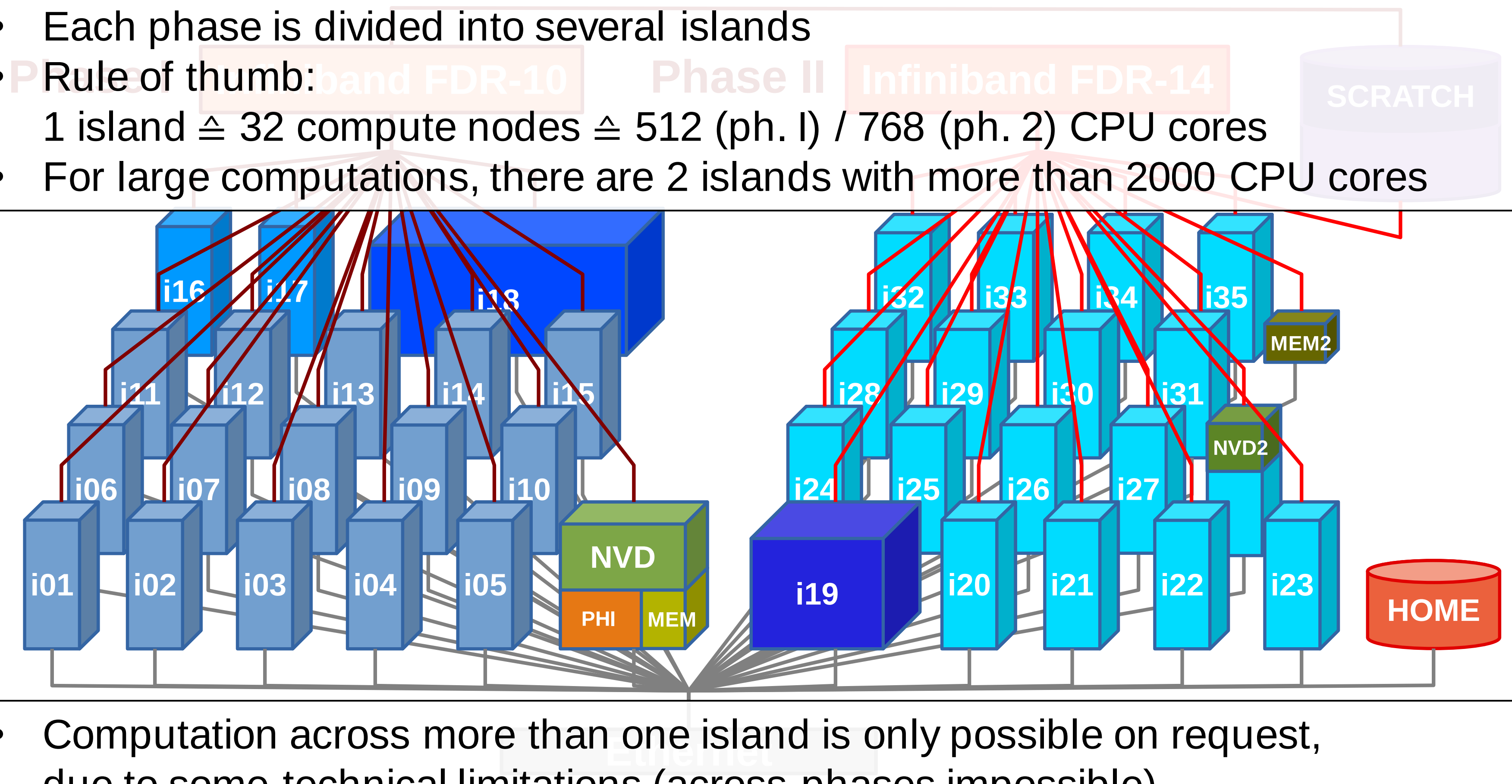
Number of Nodes	Processors per Node	Cores per Node	Memory per Node	Interconnect	Local $TMPDIR Disk	**Accelerator** Card	CPU-Hour Cost Factor
72	Two 18-Core Intel Skylake 6140	36	192 GB	100G IB	1.5 TB	N/A	1.0
40	Two 18-Core Intel Skylake 6140	36	384 GB	100G IB	1.5 TB	N/A	1.2
28	Two 24-Core Intel Skylake 8260	48	384 GB	100G IB	1.5 TB	N/A	1.2
2	Two 18-Core Intel Skylake 6140	36	192 GB	100G IB	1.5 TB	2x NVIDIA Tesla V100-32GB	2.7
1	Two 18-Core Intel Skylake 6140	36	192 GB	100G IB	1.5 TB	one NVIDIA Tesla V100-32GB	2.7
2	Two 18-Core Intel Skylake 6140	36	384 GB	100G IB	1.5 TB	2x NVIDIA Tesla V100-32GB	3.0
1	Four 16-Core Intel 6130	64	3 TB	100G IB	11 TB	N/A	6.2
2	Four 24-Core Intel 8260	96	3 TB	100G IB	1.5 TB	N/A	3.0
40	Two 32-Core AMD EPYC 7502	64	512 GB	100G IB	1.5 TB	N/A	
15	Two 32-Core AMD EPYC 7502	64	512 GB	100G IB	1.5 TB	four NVidia A100 80GB	
54	Two 32-Core Intel Icelake 8358	64	512GB	100G IB	1.6TB	N/A	
5	Two 24-Core AMD EPYC 7413	48	512GB	100G IB	960GB	eight NVidia A100 80GB	

One cluster – multiple islands



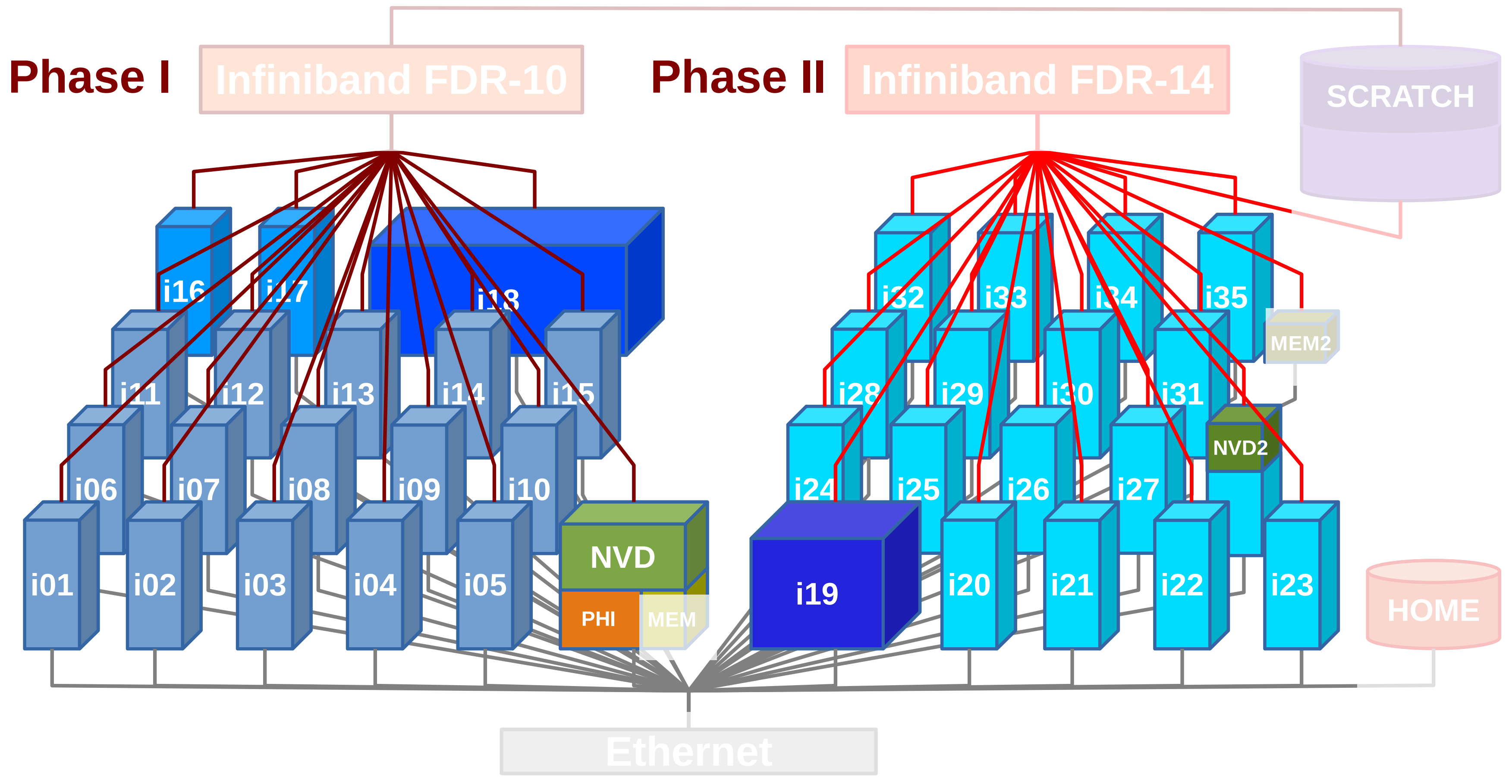
One cluster – multiple islands

- Cluster is divided into 2 phases
- Each phase is divided into several islands
- Rule of thumb:
  1 island ≙ 32 compute nodes ≙ 512 (ph. I) / 768 (ph. 2) CPU cores
- For large computations, there are 2 islands with more than 2000 CPU cores



- Computation across more than one island is only possible on request, due to some technical limitations (across phases impossible).

Compute nodes ("mpi", "nvd", "phi")



Compute nodes

Phase I (704+70 nodes):

Processors:

- 2 Intel Xeon E5-4650 (**Sandy Bridge**) processors ≙ 2 · 8 = **16 CPU cores**
- **2.7 GHz** (up to 3.3 GHz in turbo mode)

Main Memory:

- **32 GB RAM** (some have 64 GB)

Network:

- Gigabit Ethernet
- FDR-10 InfiniBand

Phase II (596+31 nodes):

Processors:

- 2 Intel Xeon E5-2680 v3 (**Haswell**) processors ≙ 2 · 12 = **24 CPU cores**
- **2.5 GHz** (up to 3.3 GHz in turbo mode)

Main Memory:

- **64 GB RAM**

Network:

- Gigabit Ethernet
- FDR-14 InfiniBand

Accelerator nodes

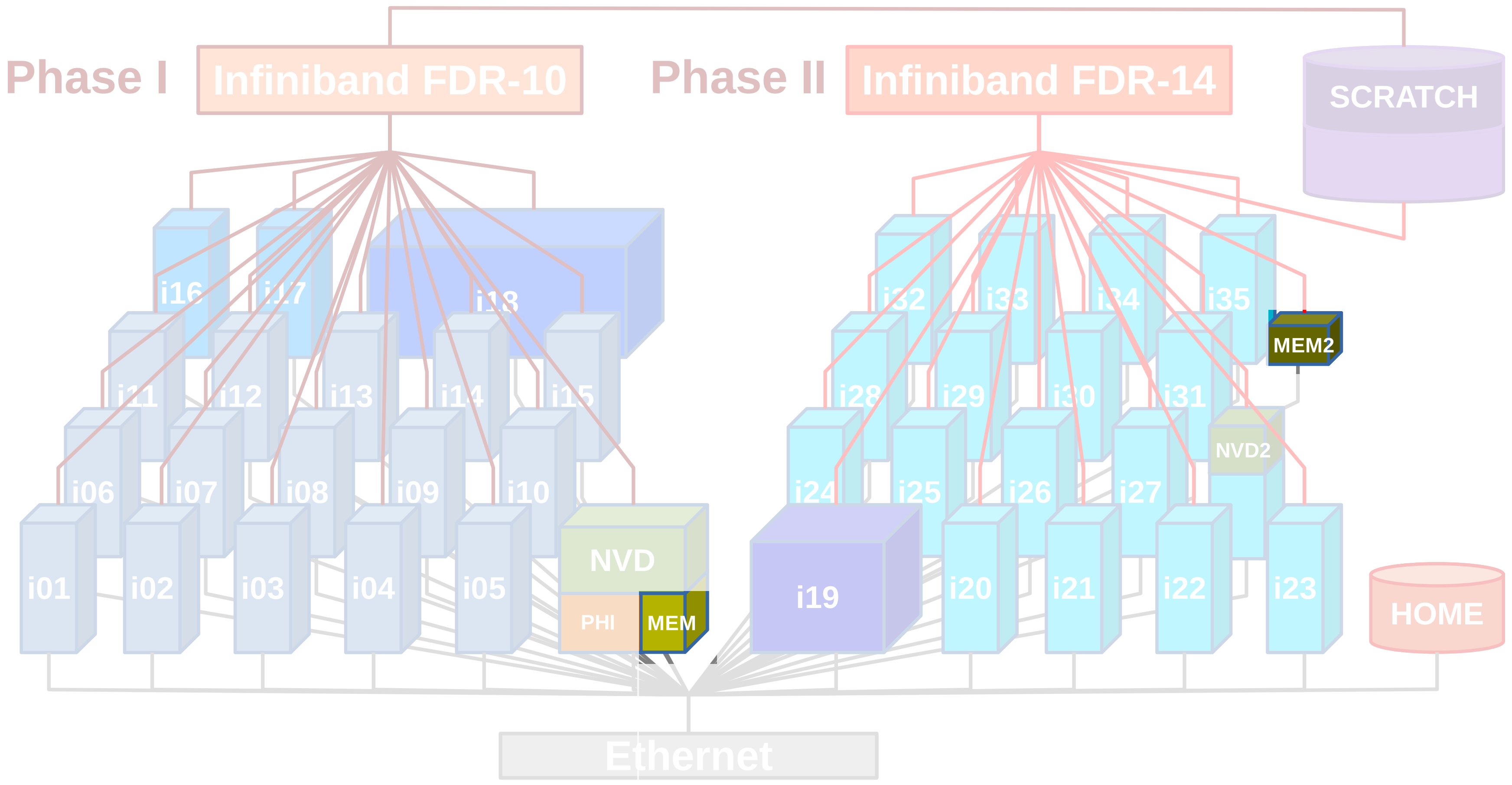
NVIDIA nodes

- 44 Sandy Bridge compute nodes have 2 **NVIDIA K20Xm** cards each
- 2 Haswell compute nodes have 2 **NVIDIA K40m** cards each
- 1 Haswell compute node has 2 **NVIDIA K80** cards

Xeon Phi Nodes

- 24 Sandy Bridge compute nodes have
  2 **Intel Xeon Phi** 5110P cards each
- 2 Sandy Bridge compute nodes have
  2 **Intel Xeon Phi** 7120P cards each

Big mem nodes (“mem”, “mem2”)



Mem nodes

Phase I (4 nodes):

Processors:

- 8 Intel Xeon E7-8837 (**Westmere**) processors $\triangleq$ 8 · 8 = **64 CPU cores**
- **2.66 GHz** (up to 2.8 GHz in turbo mode)

Main Memory:

- **1 TB** (1024 GB) **RAM**

Network:

- 10 Gigabit Ethernet
- 2 · FDR-10 InfiniBand

Phase II (4 nodes):

Processors:

- 4 Intel Xeon E7-4890 v2 (**Ivy Bridge**) processors $\triangleq$ 4 · 15 = **60 CPU cores**
- **2.8 GHz** (up to 3.4 GHz in turbo mode)

Main Memory:

- **1 TB** (1024 GB) **RAM**

Network:

- 10 Gigabit Ethernet
- 2 · FDR-14 InfiniBand

File systems

Mountpoint	/home	/work/scratch	/work/local
Size	> 300 TB	> 650 TB	> 100 GB per node
Access time	Normal (Ethernet)	Fast (InfiniBand)	Very fast (local HDD)
Accessibility	Global (cluster)	Global (cluster)	Local (node)
Data availability	permanent	≥ 1 month	Only during job runtime
Quota*	15 GB**	100 TB** 2 Mio. files**	none
Backup	Weekly + snapshots	none	none

* Use the command **`cquota`** to find out your current usage and quota.
** Can be increased on request.

Login nodes

4 nodes (hardware similar to Phase I):

Processors:

- 4 Intel Xeon E5-4650 (**Sandy Bridge**) processors ≙ 4 · 8 = **32 CPU cores**
- **2.7 GHz** (up to 3.3 GHz in turbo mode)

Main Memory:

- **128 GB RAM**

Network:

- 2 · 10 Gigabit Ethernet
- 2 · FDR-10 InfiniBand

8 nodes (hardware similar to Phase II):

Processors:

- 2 Intel Xeon E5-2680 v3 (**Haswell**) processors ≙ 2 · 12 = **24 CPU cores**
- **2.5 GHz** (up to 3.3 GHz in turbo mode)

Main Memory:

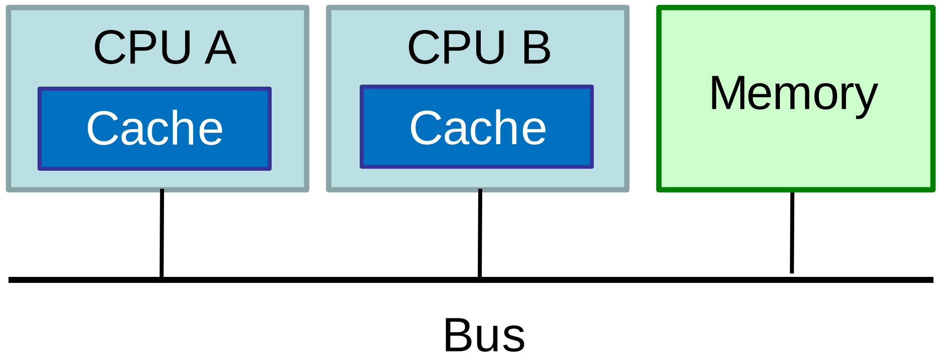
- **128 GB RAM**

Network:

- 2 · 10 Gigabit Ethernet
- FDR-14 InfiniBand

Parallelism and memory hierarchy

A processor’s view of the memory is through its cache



Cache coherence

- Problem – different processors may see different values
  - Without further precautions (!)

X:N = cache line address : contents version

Time	Event	Cache CPU A	Cache CPU B	Memory
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 in X	0	1	0

- Cache coherence – which value will be returned by a read?
  - Cache coherence protocols prevent different versions of the same cache line from appearing simultaneously in two or more caches

Coherence of a memory system

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. A read by a processor to location X that follows a write by another processor returns the written value if the read and the write are sufficiently separated in time and no other writes to X occur between the two accesses
3. Writes to the same location are serialized, that is, two writes to the same location by any two processors are seen in the same order by all processors.

Memory consistency

- Coherence refers to the behavior of the memory system when a single memory location is accessed by multiple threads
- Consistency refers to the ordering of accesses to different memory locations, observable from various threads in the system
  - When must a processor see a value that has been updated by another processor?
  - In what order does a processor observe the data writes of another processor?

Assumption for now: Sequential consistency

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

[Lamport, 1979]

- Advantage – simple programming paradigm
- Disadvantage – potential performance degradation

Cache coherence protocols

Snooping

- Every cache that has a copy of a block of physical memory also has a copy of the sharing status of the block
- No centralized state is kept
- All caches are accessible via some broadcast medium (i.e., bus)
- All caches snoop on the medium to see whether they have a copy of a block that is being requested

Directory-based

- The sharing status of a block of physical memory is kept in one location (i.e., the directory)
- Two variants
  - Centralized directory (UMA)
  - Distributed directory (NUMA)
- Distributed directory needed if scalability is a concern

Snooping protocols

Write invalidate

- Remaining copies are invalidated on a write
- Most common for both snooping and directory-based protocols
- Guarantees exclusive access

Event	Bus activity	Cache CPU A	Cache CPU B	Memory
				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 for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Write update or write broadcast

- All copies are updated on a write
- Consumes more bandwidth – less common

Implementation of a write invalidate

- Invalidate is performed by broadcasting address to be invalidated on the bus
- All processors snoop on the bus and watch the addresses: if address is in their cache, data are invalidated
- Writes to a shared data item are serialized

Finding data items on a miss

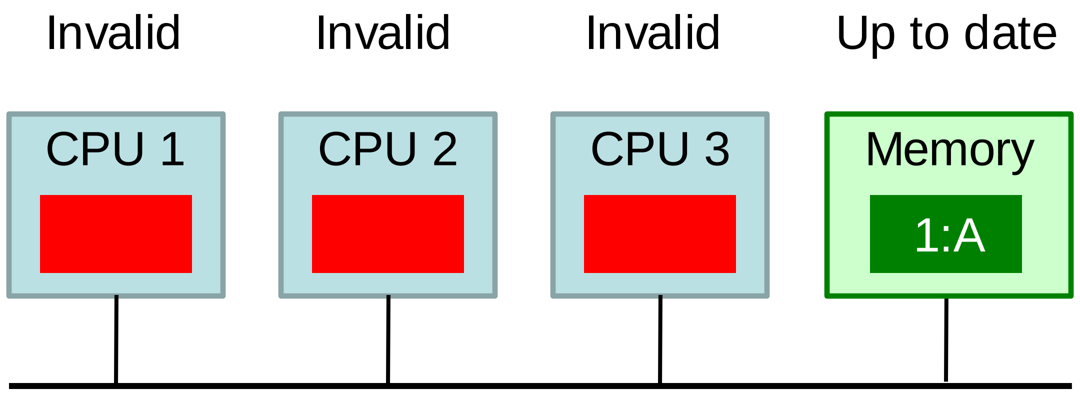
- Write-through caches
  - Data can be retrieved from main memory
- Write-back caches
  - Cache misses handled via snooping – if processor has dirty copy of a cache line, it provides the block in response to a request and causes memory access to be aborted
  - Complexity arises from transferring block to requesting processor – can take longer than getting it from main memory - especially if processors are on separate chips
  - Lower memory bandwidth demands & hence more scalable – often used at outermost cache levels

MESI protocol

- Popular write-back cache coherence protocol
  - Used e.g. in Intel Core i7
- Each cache line can be in one of the following four states

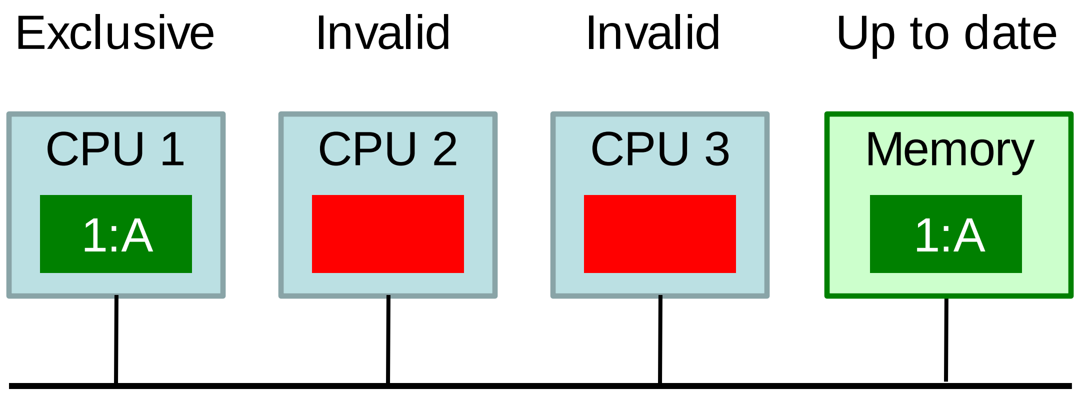
1. Modified = cache line is valid; memory is invalid; no other copies exist
2. Exclusive = no other cache holds the line; memory up to date;
3. Shared = multiple caches may hold the line; memory up to date
4. Invalid = no valid data

MESI protocol - example



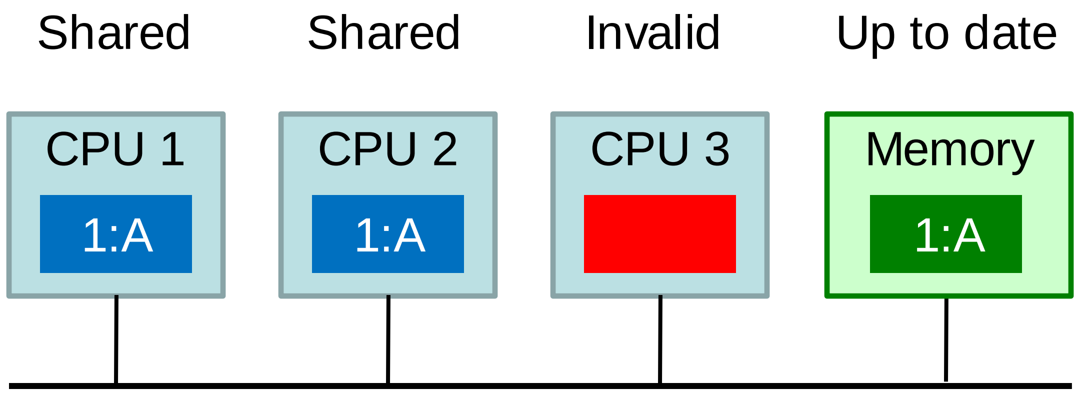
N:X = cache line address : contents version

MESI protocol - example



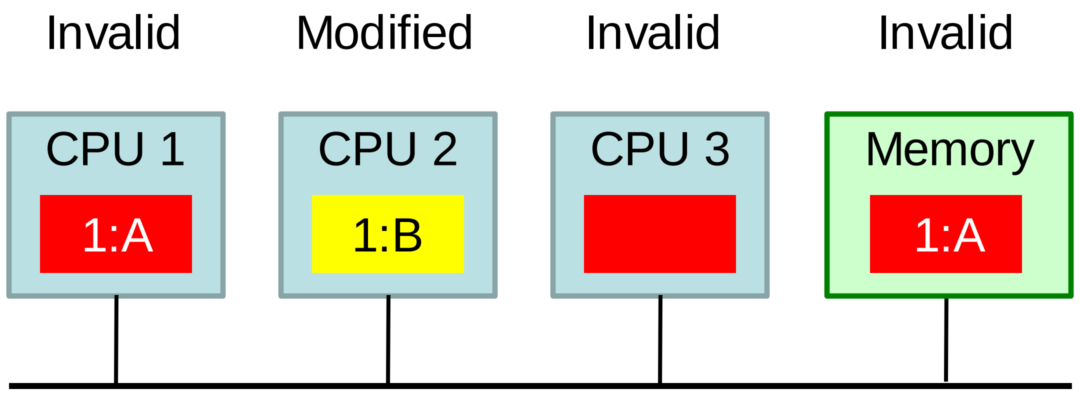
Initial state

MESI protocol - example



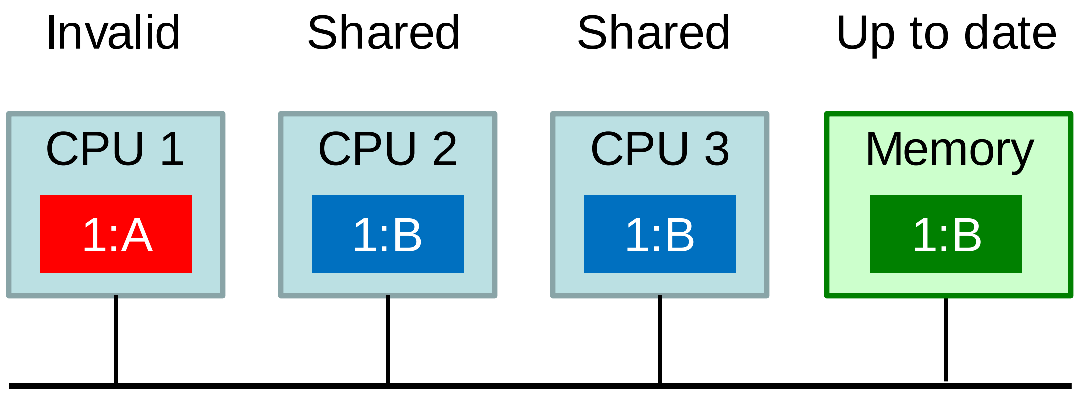
CPU2: read 1

MESI protocol - example



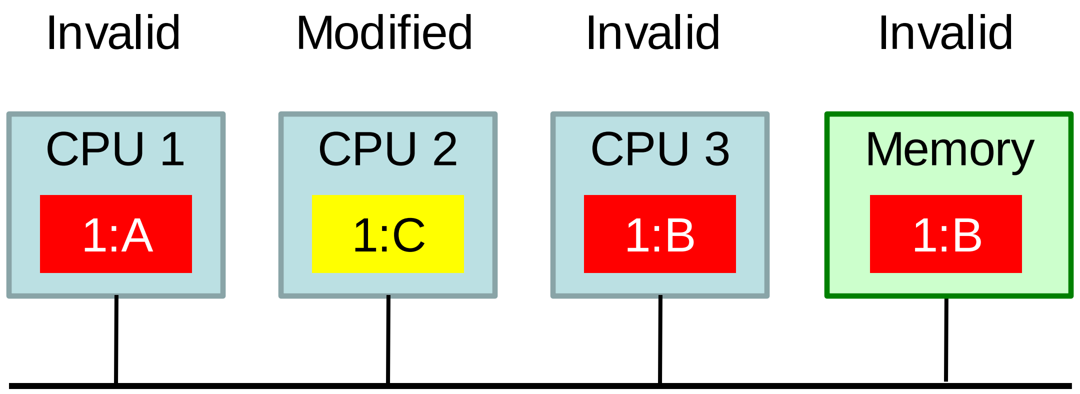
CPU2: write B to 1

MESI protocol - example



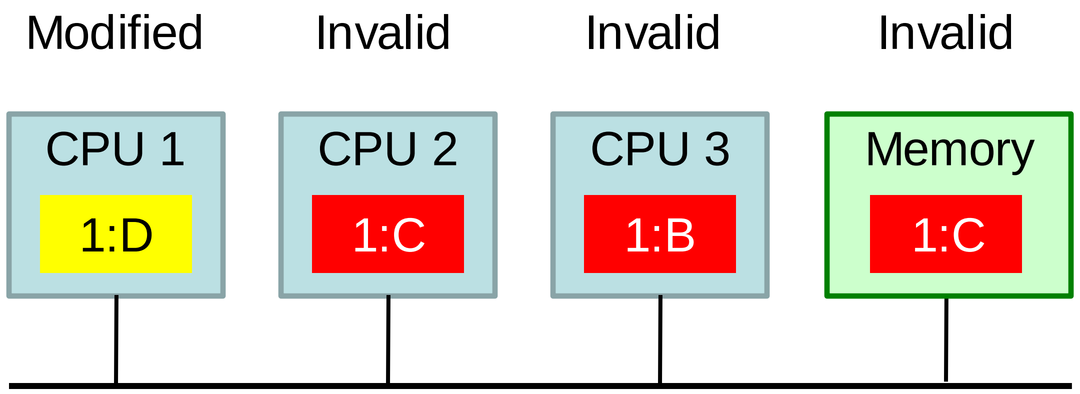
CPU3: read 1

MESI protocol - example



CPU2: write C to 1

MESI protocol - example



CPU1: write D to 1

Coherence misses

- Uniprocessor misses
  - Compulsory: Each memory block when first referenced
  - Capacity: due to the limited size of a cache
  - Conflict: data was in the cache previously, but got evicted
- True sharing miss – cache miss occurring because a block was invalidated by another processor writing the same word
  - Arises from communication of data through coherence mechanism
  - Independent of cache-line size
- False sharing miss – cache miss occurring because a block was invalidated by another processor writing a different word
  - Miss that would not occur if block size were one word

Coherence misses (2)

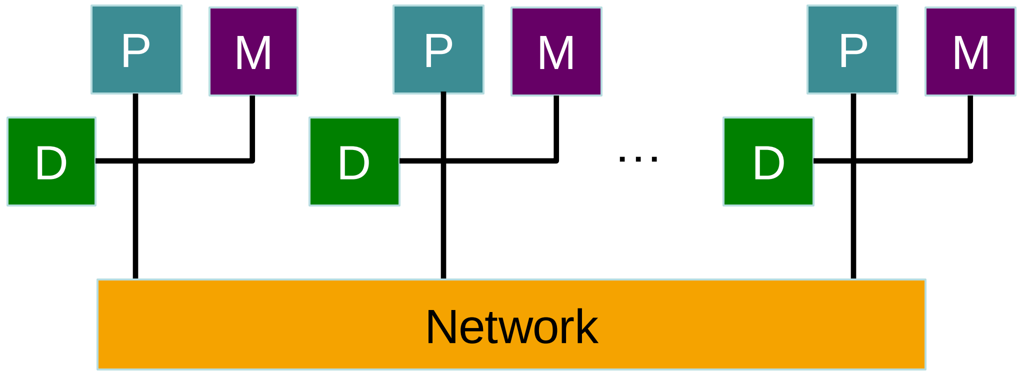
Assume X1 and X2 are in the same cache line, and initially in the caches of P1 and P2 in shared state

Time	P1	P2	Miss
1	Write X1		True miss – P2 has read X1 before; invalidate X1 & X2 in P2
2		Read X2	False miss – X2 irrelevant to P1
3	Write X1		Invalidate X1 & X2 on P2
4		Write X2	False miss – X2 irrelevant to P1; Invalidate X1 & X2 on P2
5	Read X2		True miss – P2 has written X2 before

- Coherence misses most important for tightly-coupled applications that share significant amounts of user data

Cache-coherent NUMA systems

- Coherence of caches established via directory
  - Distributed database storing location and status cache of lines
  - Requires fast hardware because it must be queried on every memory reference

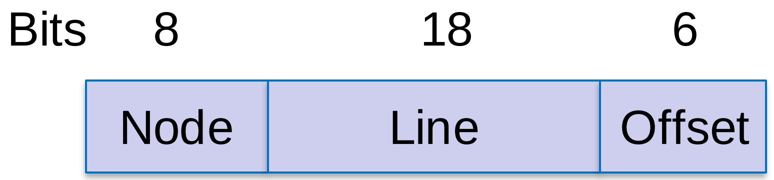


Example

- 256 nodes
- 1 CPU + 16 MB RAM per node
- Total memory 2^{32} bytes (~4 gigabytes): 2^4 (16) + 2^8 (256) + 2^20 (~1MB) = 2^32
- 2^{26} cache lines, 64 bytes each
- Memory statically allocated among nodes
  - 0-16M in node 0, 16-32M in node 1, etc.
- Directory of each node holds entries for the 2^{18} cache lines comprising its 2^{24} bytes of memory
- Assumption: a line can be held in at most one cache

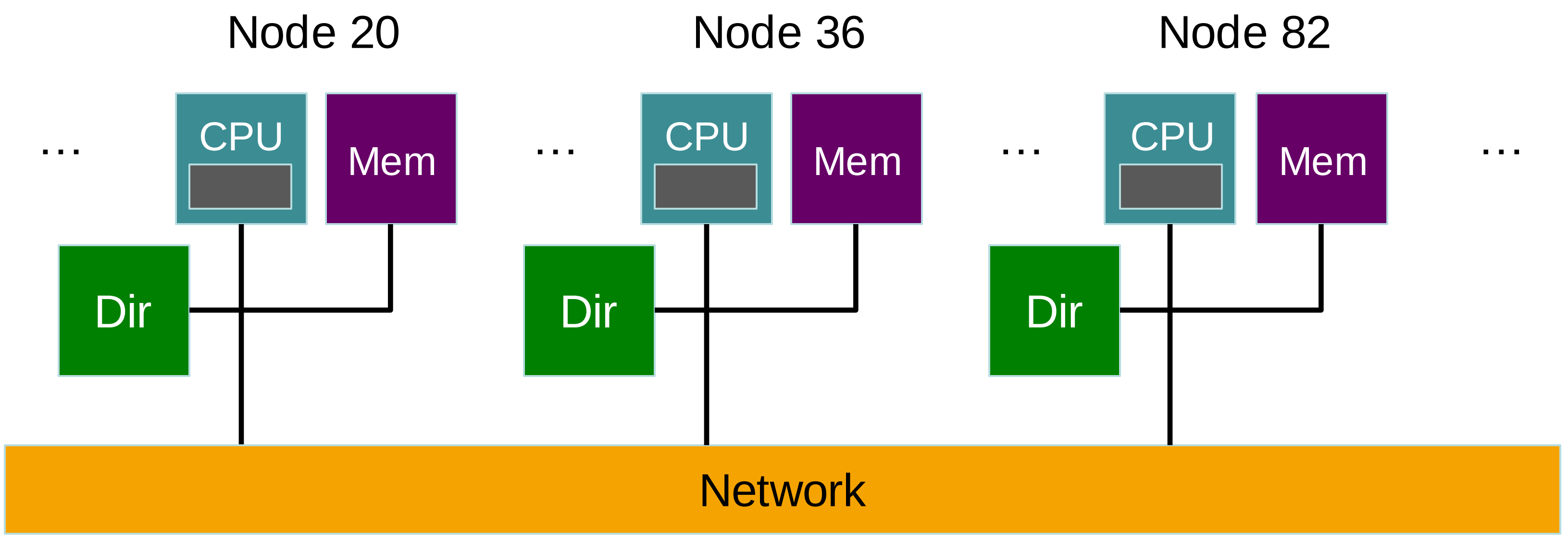
Example (2)

- Consider LOAD instruction from CPU 20 that references a cached line at address 0x24000108
- MMU splits address into three parts



- Address 0x24000108 = node 36 ; line 4 ; offset 8

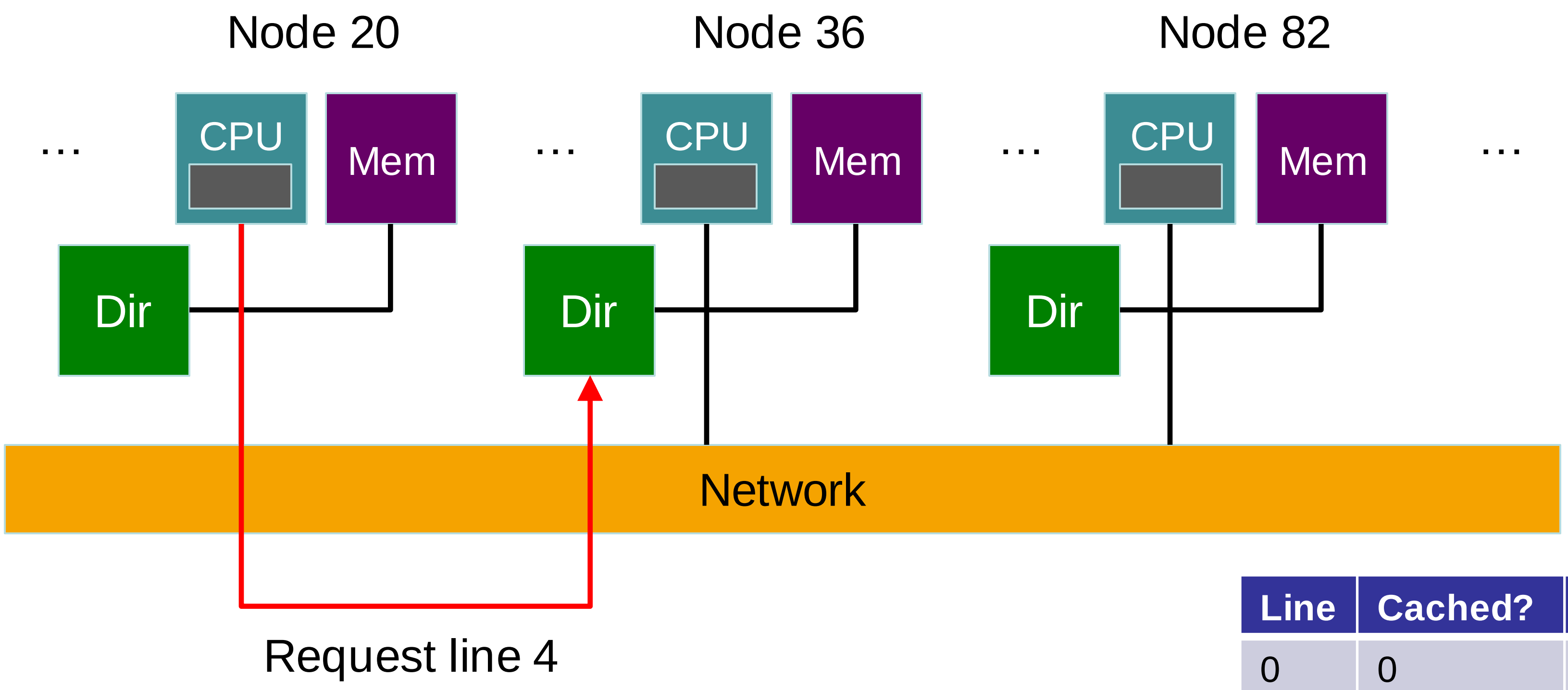
Example (3)



Line	Cached?	Where?
0	0	
1	0	
2	1	82
3	0	
4	0	

Directory of CPU 36

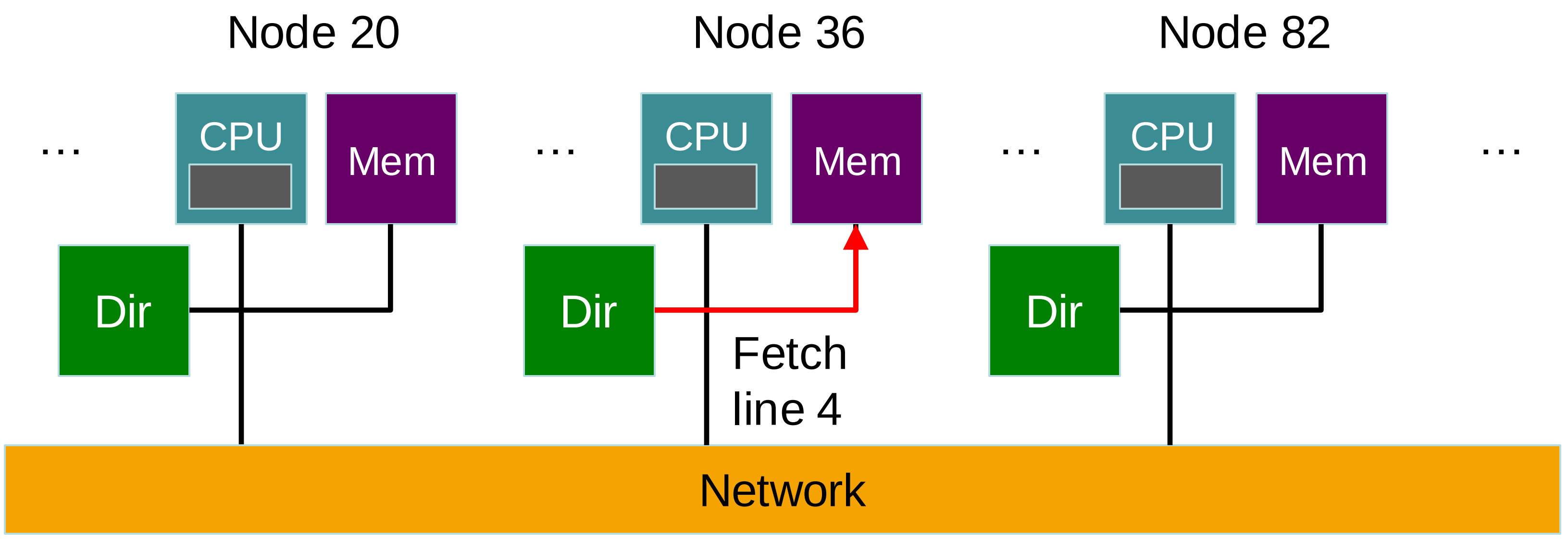
Example (3)



Directory of CPU 36

Line	Cached?	Where?
0	0	
1	0	
2	1	82
3	0	
4	0	

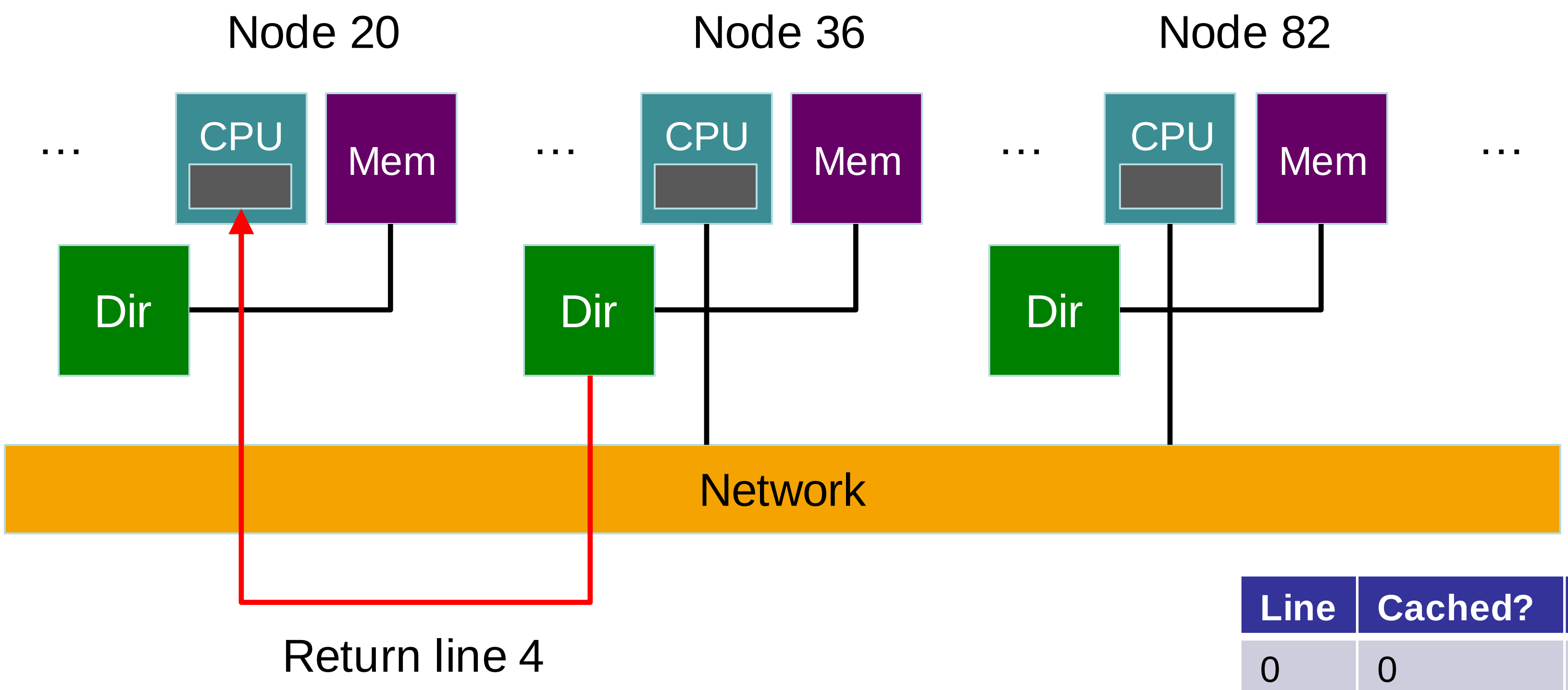
Example (3)



Directory of CPU 36

Line	Cached?	Where?
0	0	
1	0	
2	1	82
3	0	
4	0	

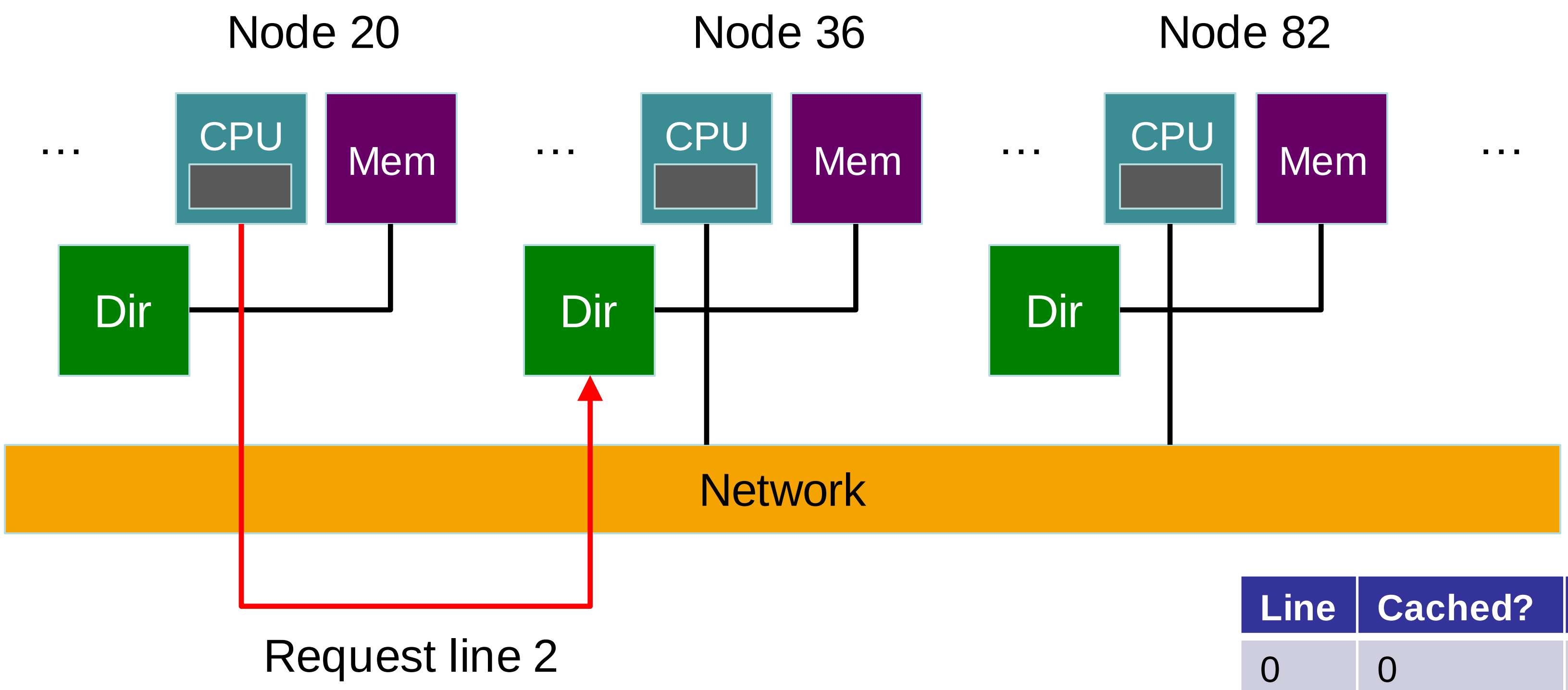
Example (3)



Directory of CPU 36

Line	Cached?	Where?
0	0	
1	0	
2	1	82
3	0	
4	1	20

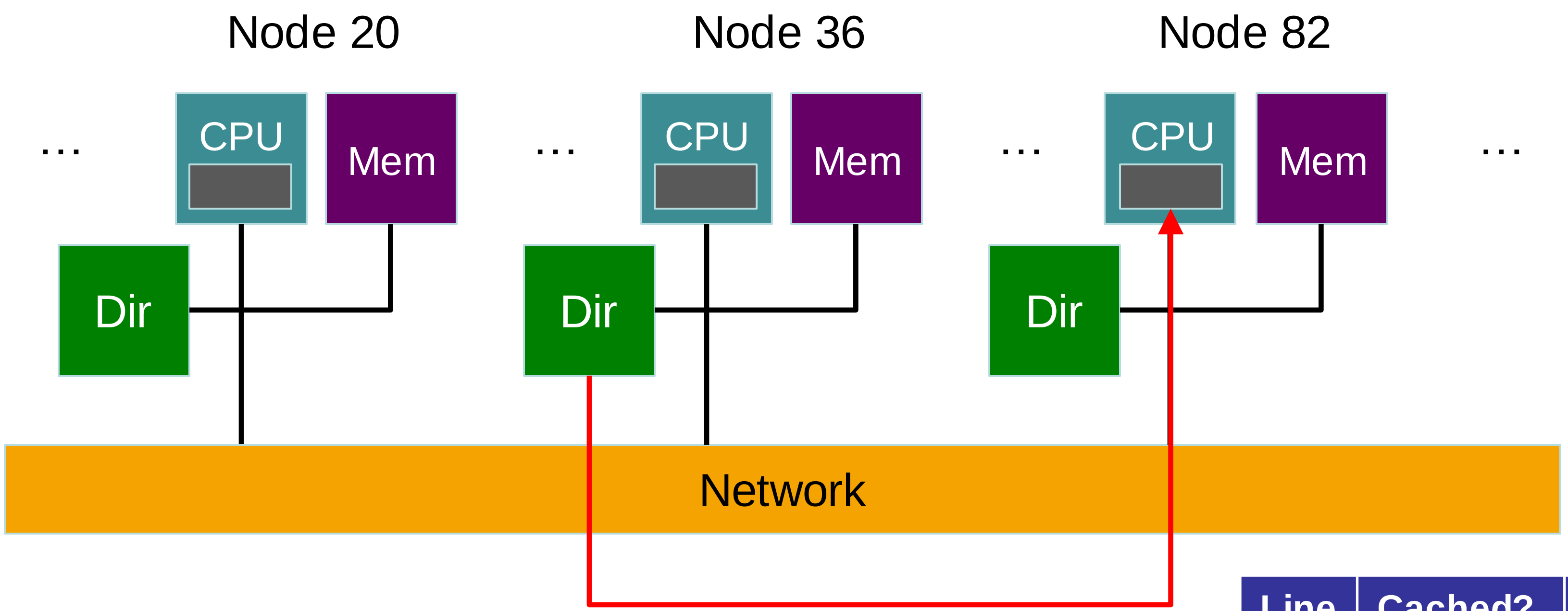
Example (3)



Directory of CPU 36

Line	Cached?	Where?
0	0	
1	0	
2	1	82
3	0	
4	1	20

Example (3)

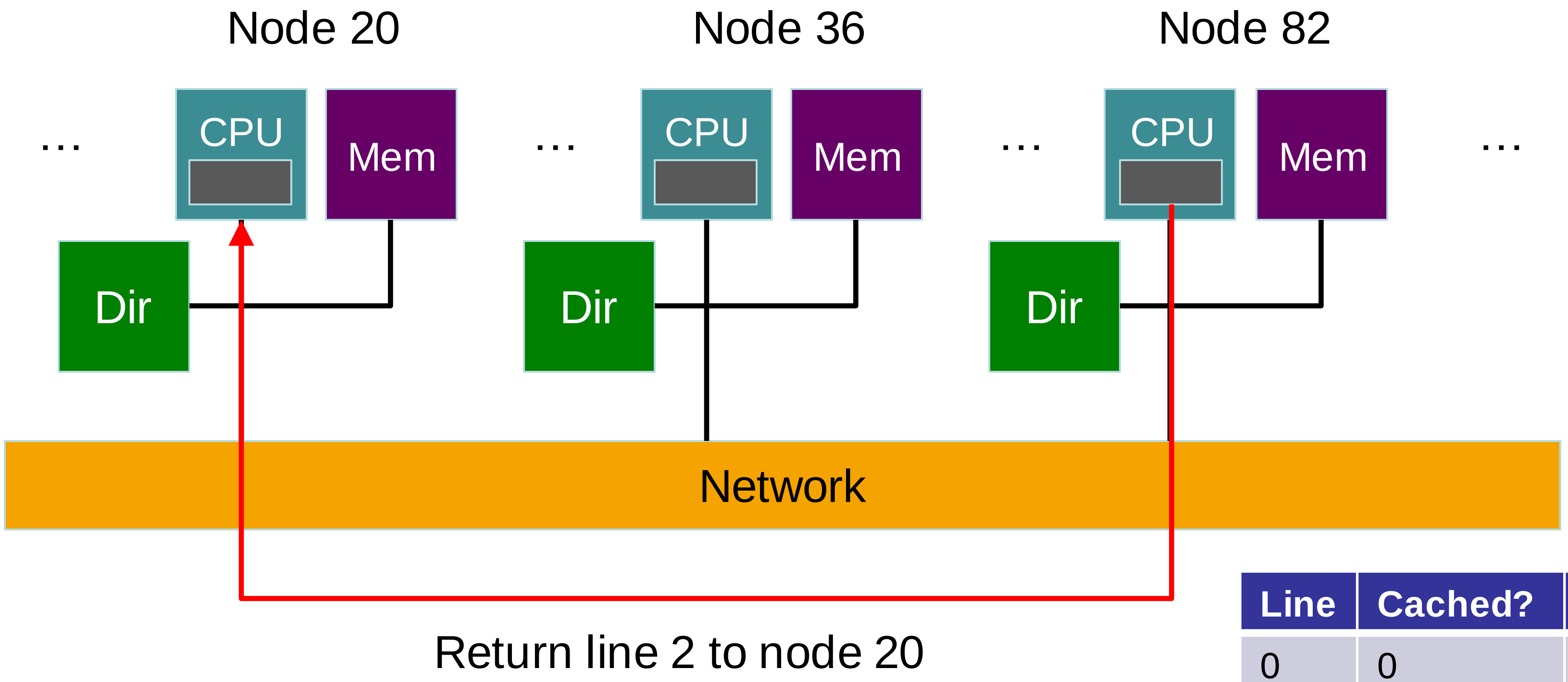


Please send line 2 to node 20

Directory of CPU 36

Line	Cached?	Where?
0	0	
1	0	
2	1	82
3	0	
4	1	20

Example (3)



Line	Cached?	Where?
0	0	
1	0	
2	1	20
3	0	
4	1	20

Directory of CPU 36

Number of copies

- Directory has 2^{18} 9-bit entries = 1.76% of total memory
- Limitation: a line can be cached at only one node
- Alternatives
  - k entries per line, allowing copies at up to k nodes
  - Bitmap with one bit per node (substantial increase in memory overhead)
  - 8-bit field as head of a linked list (requires extra storage for list pointers and time overheard for searching the list)

Status of a line

- Another optimization is to keep track of a line’s status (dirty or clean)
- The home node can satisfy a read request for a clean line from its local memory without having to forward it to another node’s cache
- A read request for a dirty line must still be forwarded to the node holding the copy
- No advantage if only one copy is allowed because any request requires invalidation of the previous copy
- Modification of a cached line requires home node to be informed and invalidation of all other copies
- Potentially significant coherence traffic

Memory semantics

- Shared memory = image of a single shared address space
  - Promises intuitive programming
- Implementation quite complex in reality
  - Many memory modules, each holding some portion of the physical memory
  - CPUs and memories often connected by complex interconnection network
  - Memory hierarchy with multiple levels (registers down to main memory)

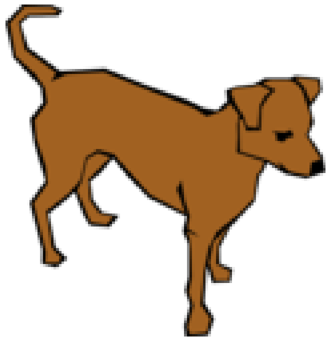
Order of updates

Can be influenced by two factors

- Order in which memory request “messages” arrive not necessarily the same as the one in which they were issued
  - A single thread may observe writes in an order different from the order another thread wrote them
  - Order may even differ among multiple readers
- Compiler may re-order instructions (even possible on uni-processor systems)

Piecewise update of reality

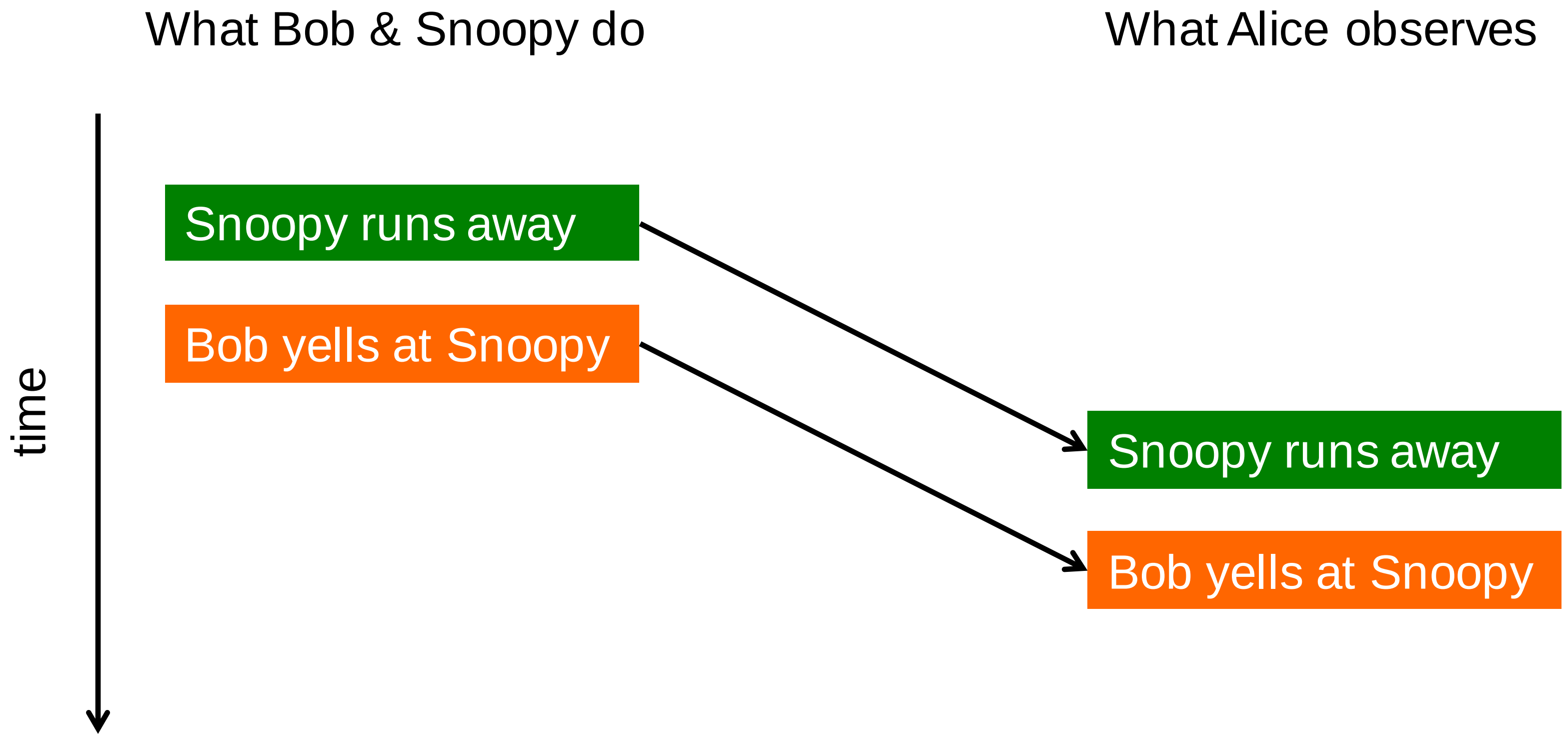
Bob



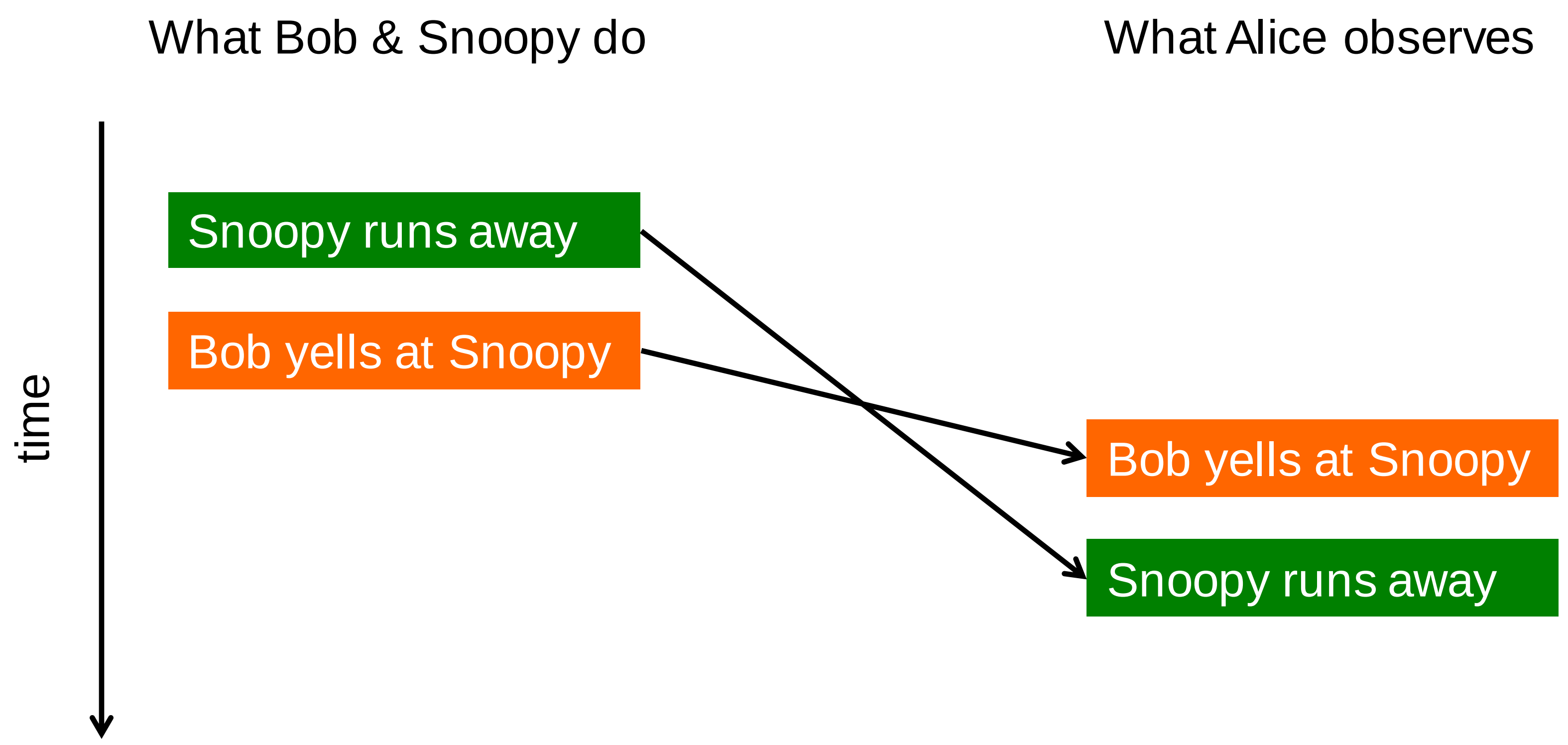
Snoopy
(Bob’s dog)

Alice
(observer)

Piecewise update of reality (2)



Perceived update order reversed



Instruction reordering

Thread 2

```
extern int x;
extern bool x_init;

void f2()
{
  int y;
  while (!x_init)
    usleep(10000);
  y = x;
  // …
}
```

Thread 1

```
int x;
bool x_init;

void init()
{
  x = initialize();
  x_init = true;
  // …
}
```

Instruction reordering (2)

- Compiler (or hardware instruction scheduler) may decide to execute x_init = true first
  - No thread-local dependence
  - Thread 2 may assign an uninitialized x to y

- Since thread 2 makes no assignments to x_init, an optimizer may decide to lift evaluation of x_init out of the loop
  - Thread 2 may sleep forever or not at all

Memory ordering

- Memory ordering specifies what a programmer can assume about the order in which a thread sees updates of memory locations
- Sequential consistency – simplest memory order
    - Every thread sees the effects of every operation in the same order
        - As if the operations of all the threads were executed in some sequential order (like by a single thread)
        - The operations of each individual thread appear in this sequence in the order specified by the program
    - Usually lowers concurrency
    - Still many possible sequentially consistent orders for a given set of threads

Relaxed consistency models

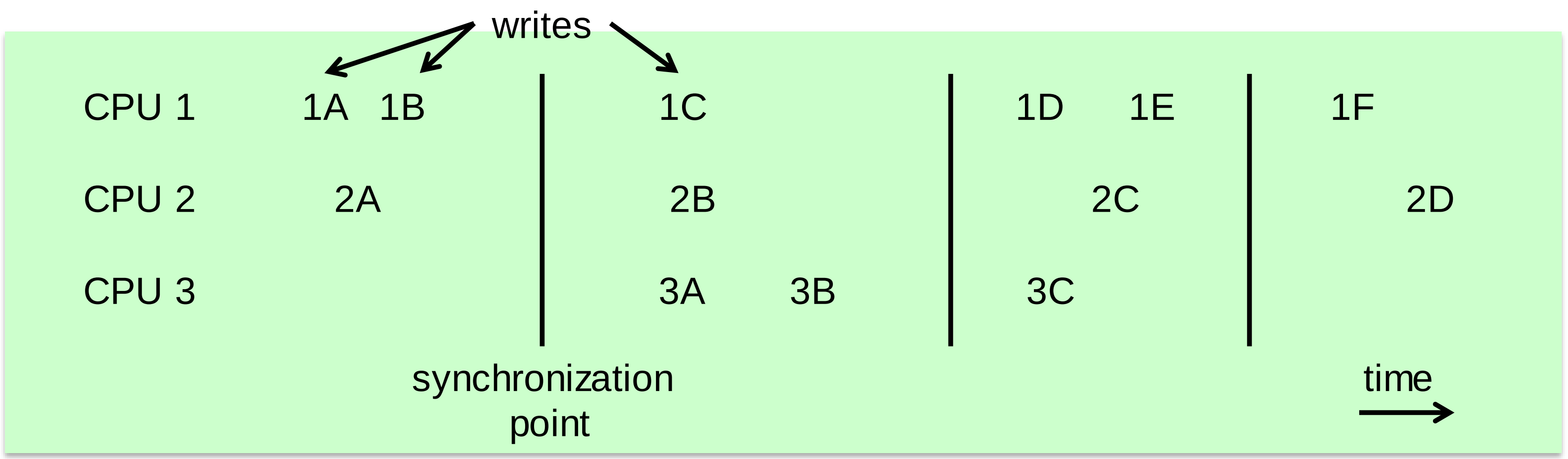
- Sequential consistency is convenient, but impractical because of potential performance degradation
- Idea of relaxed consistency models
  - Let reads and writes complete out of order
  - Use synchronization to enforce ordering where important
- Relaxed consistency models can be distinguished by the orderings they guarantee / relax

Processor consistency

1. Writes by any CPU are seen by all in the order they were issued
   - Assume CPU 1 issues writes to variable x with values 1A,1B, 1C
   - No other processor will ever see 1B followed by 1A
2. For every memory word, all CPUs see writes to it in the same order
   - Requires every memory word to have unambiguous value at the end

Weak consistency

- Does not guarantee that writes from a single CPU are seen in order
- Usually combined with sequentially consistent synchronization ops.
  - Finishes all pending writes and issues no new ones until the pending ones and the synchronization itself is done
- Time divided into epochs delimited by synchronization



Release consistency

- Finishing all pending writes after a synchronization is expensive
- Instead, release consistency adopts model similar to critical sections
    - When leaving a critical section, a thread does not force all the writes to complete immediately
    - It only ensures that they are completed before any thread enters the critical sections again
- Synchronization split into two parts
    - Acquire – get exclusive access to shared data
    - Release – indicate that exclusive access is finished

Synchronization

- A program is synchronized if all accesses to shared data are ordered by synchronization operations
- Updates of a single location not ordered by synchronization are called data races
- Synchronization allows programs to behave as under sequential consistency even if the architecture implements a more relaxed consistency model
- Building synchronization mechanisms is hard

Synchronization (2)

- Synchronization mechanisms implemented using hardware supplied synchronization instructions
- Uninterruptible instruction or instruction sequence capable of atomically reading and changing a value together with the ability to tell whether read and write were performed atomically
- Synchronization can become a bottleneck in
  - Large-scale multiprocessors
  - High-contention situations

Basic hardware primitives

- Atomic exchange – interchanges a value in a register for a value in memory
- Further primitives
    - Test and set (used in many older multiprocessors)
    - Fetch and increment
- Atomicity complicates implementation of coherence – does not allow any other operation to occur between the read and the write and yet must not deadlock
- Alternative - sequence of two instructions
    - Second instructions returns value from which can be deduced whether execution was atomic (e.g., load linked, store conditional)

Spin locks

- Atomic operation can be used to implement spin locks
    - Can be used if the lock to be held for a very short amount of time and if locking should be low latency
    - Caveat: spin locks tie up the processor
- Simple implementation keeps lock in memory

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

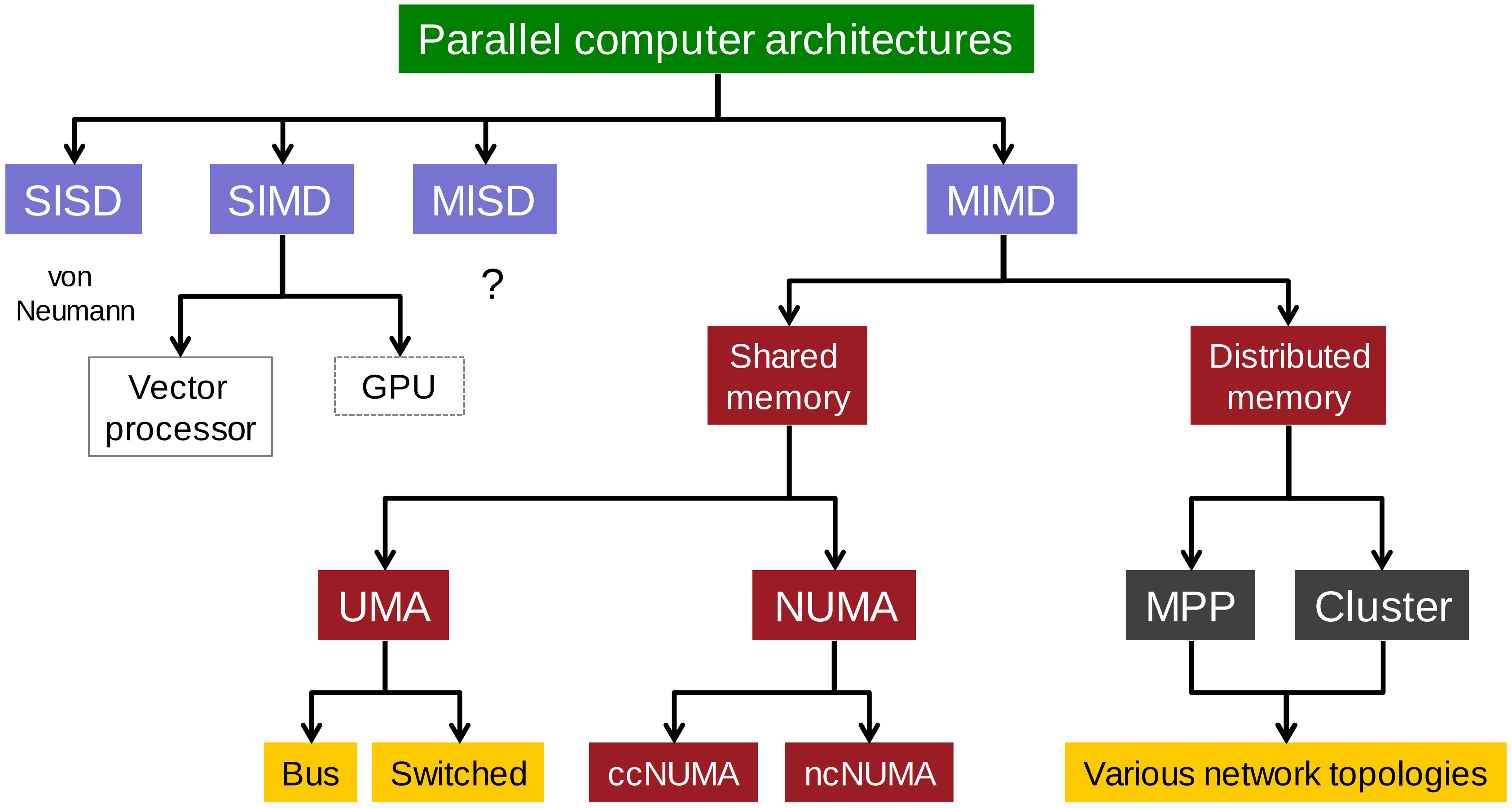
- Problem – each attempt to exchange requires a write operation
- If multiple processors attempt to write, many of the writes will generate write misses

Spinning on a cached copy

- Locks can be cached using cache coherence mechanism
  - Spinning on cached copy instead of main memory
  - Exploit locality: processor that used the lock last is likely to use it again in the near future
- Idea: execute exchange only after verifying that lock is available

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

Summary


Parallel computer architectures
SISD
SIMD
MISD
MIMD
von Neumann
?
Vector processor
GPU
Shared memory
Distributed memory
UMA
NUMA
MPP
Cluster
Bus
Switched
ccNUMA
ncNUMA
Various network topologies