Lecture 13:

Directory-Based Cache Coherence

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2020

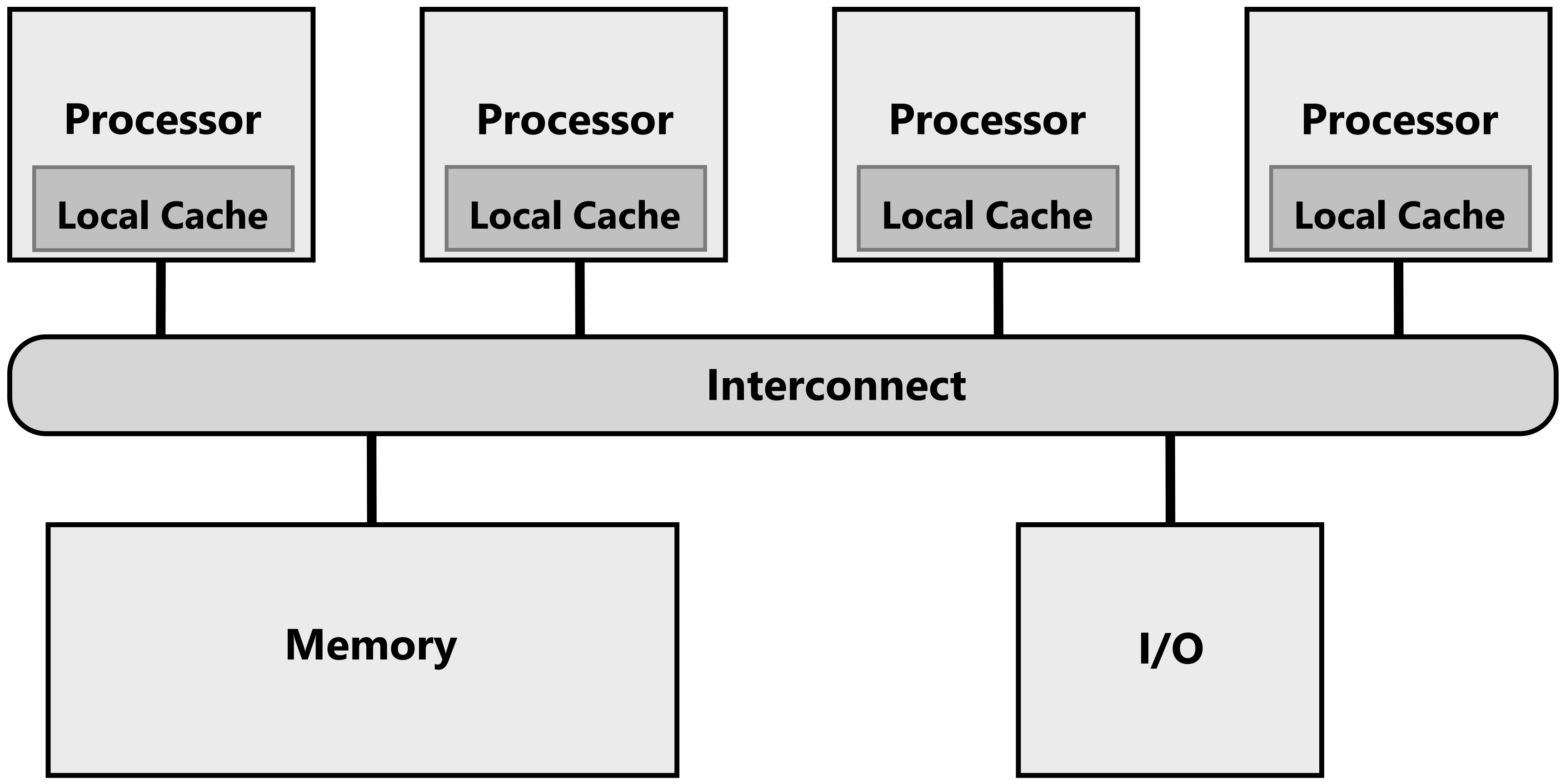
Today you will learn...

- What limits the scalability of snooping-based approaches to cache coherence
- How a directory-based scheme avoids these problems
- How the storage overhead of the directory structure be reduced (and at what cost)
- How the interconnection network (bus, point-to-point, ring) affect scalability and design choices

Implementing cache coherence

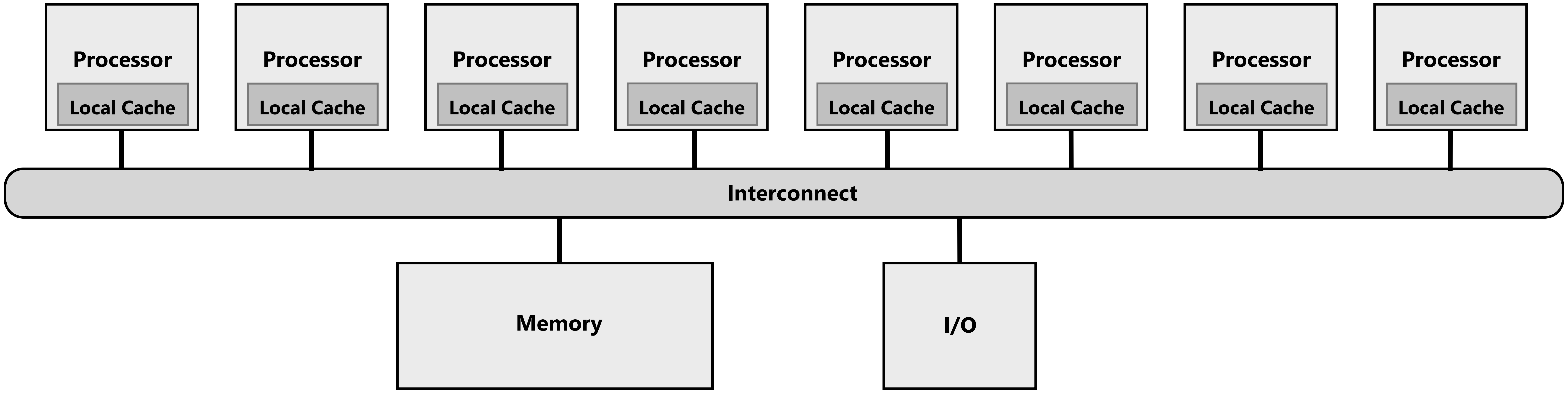
The snooping cache coherence protocols from the last lecture relied on <u>broadcasting</u> coherence information to all processors over the chip interconnect.

Every time a cache miss occurred, the triggering cache communicated with all other caches!



Scaling problems with snoopy coherence

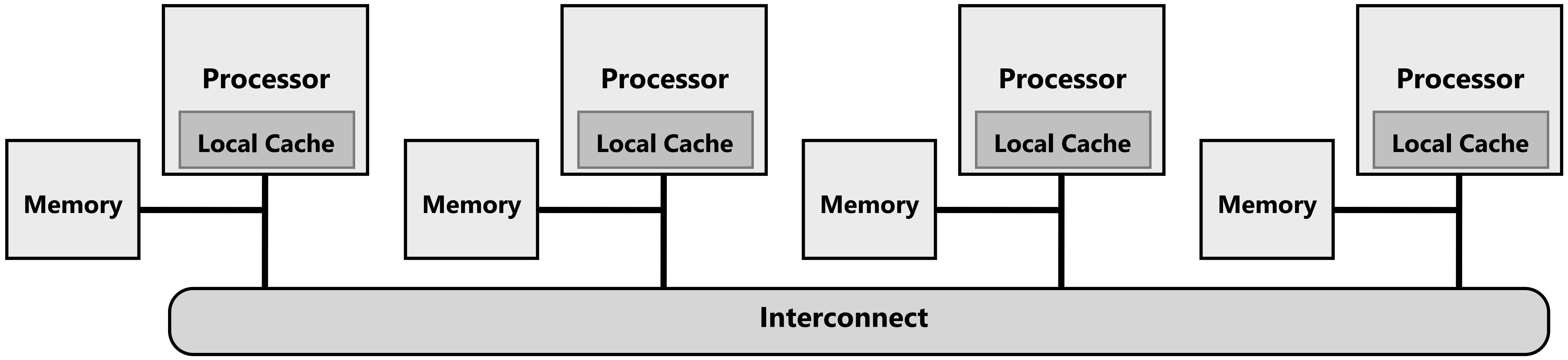
- How does performance scale with number of processors?



- Communication limited to one message at a time
- Even a scalable point-to-point interconnect requires P messages per memory request (occupying each cache)

Fundamentally, all-to-all broadcast will not scale.

Scaling cache coherence to large machines



Recall non-uniform memory access (NUMA) shared memory systems

Idea: locating regions of memory near the processors increases scalability: it yields higher aggregate bandwidth and reduced latency (especially when there is locality in the application)

But... efficiency of NUMA system does little good if the coherence protocol can't also be scaled!

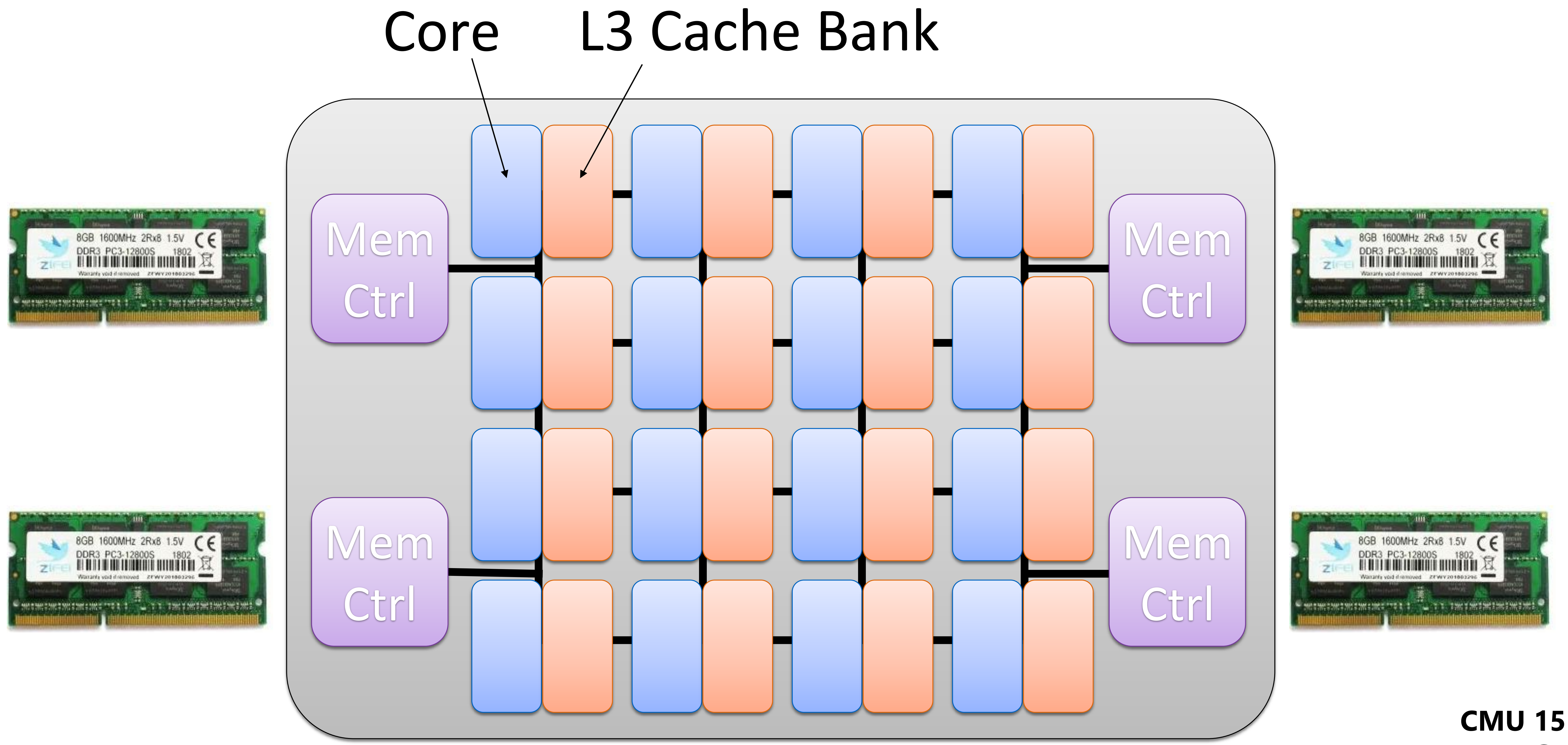
Consider this case: processor accesses nearby memory (good...), but to ensure coherence still must broadcast to all other processors it is doing so (bad...)

Some terminology:

- **cc-NUMA = "cache-coherent, non-uniform memory access"**
- **Distributed shared memory system (DSM): cache coherent, shared address space, but architecture implemented by physically distributed memories**

Scaling cache coherence in current multicores

- ccNUMA typically refers to supercomputing/clusters
- Same issues appear in multicores
  - NUMA: Memory controllers distributed around chip
  - NUCA (non-uniform cache access): L3 banks distributed around the chip too



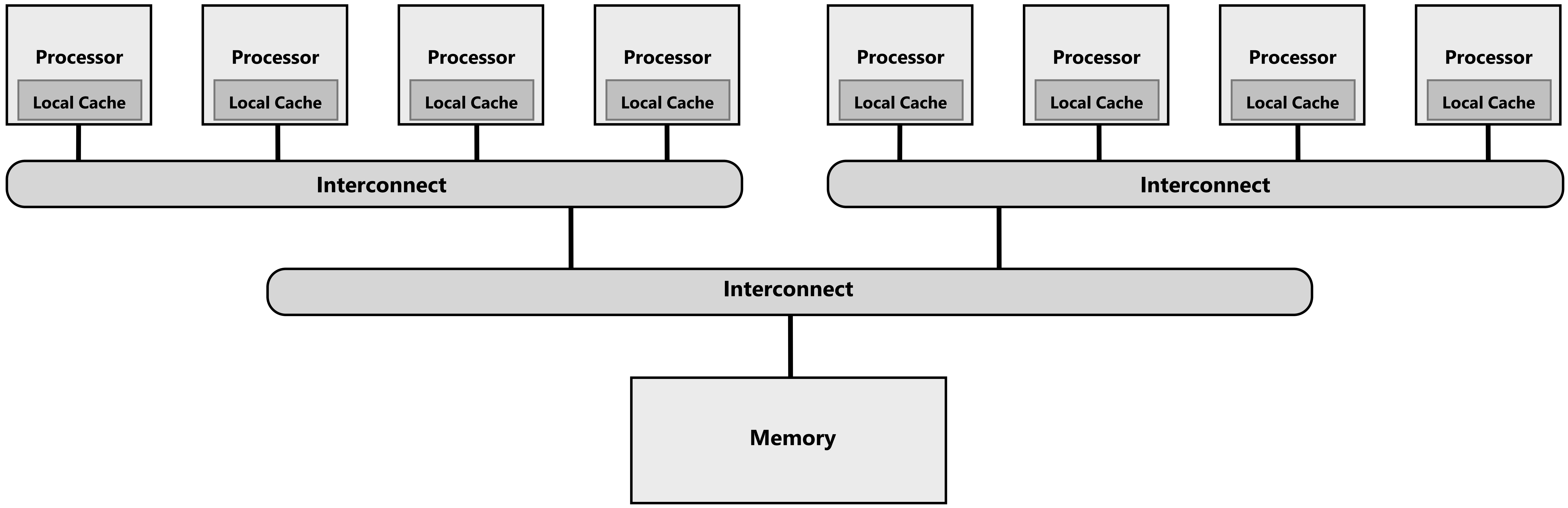
Why did we need a bus?

- Ordering
  - Bus serializes requests, ordering some before others
  - BUT: Coherence does not require ordering of requests to different addresses
    - (More on this when we discuss consistency)

- Communication
  - Simple, fast (but not scalable) broadcast medium
  - BUT: Coherence does not require broadcast
    - Only need to communicate with sharers
    - Most data is not shared by every cache

One possible solution: hierarchical snooping

Use snooping coherence at each level

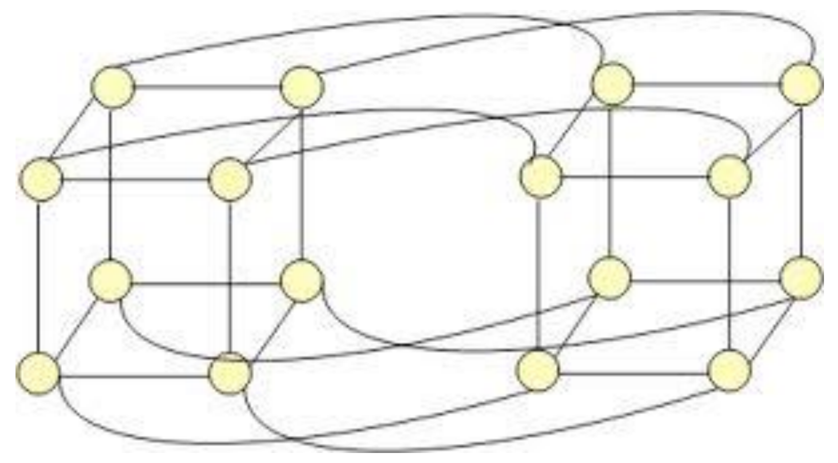


Advantages

- **Relatively simple to build (already must deal with similar issues due to multi-level caches)**

Disadvantages

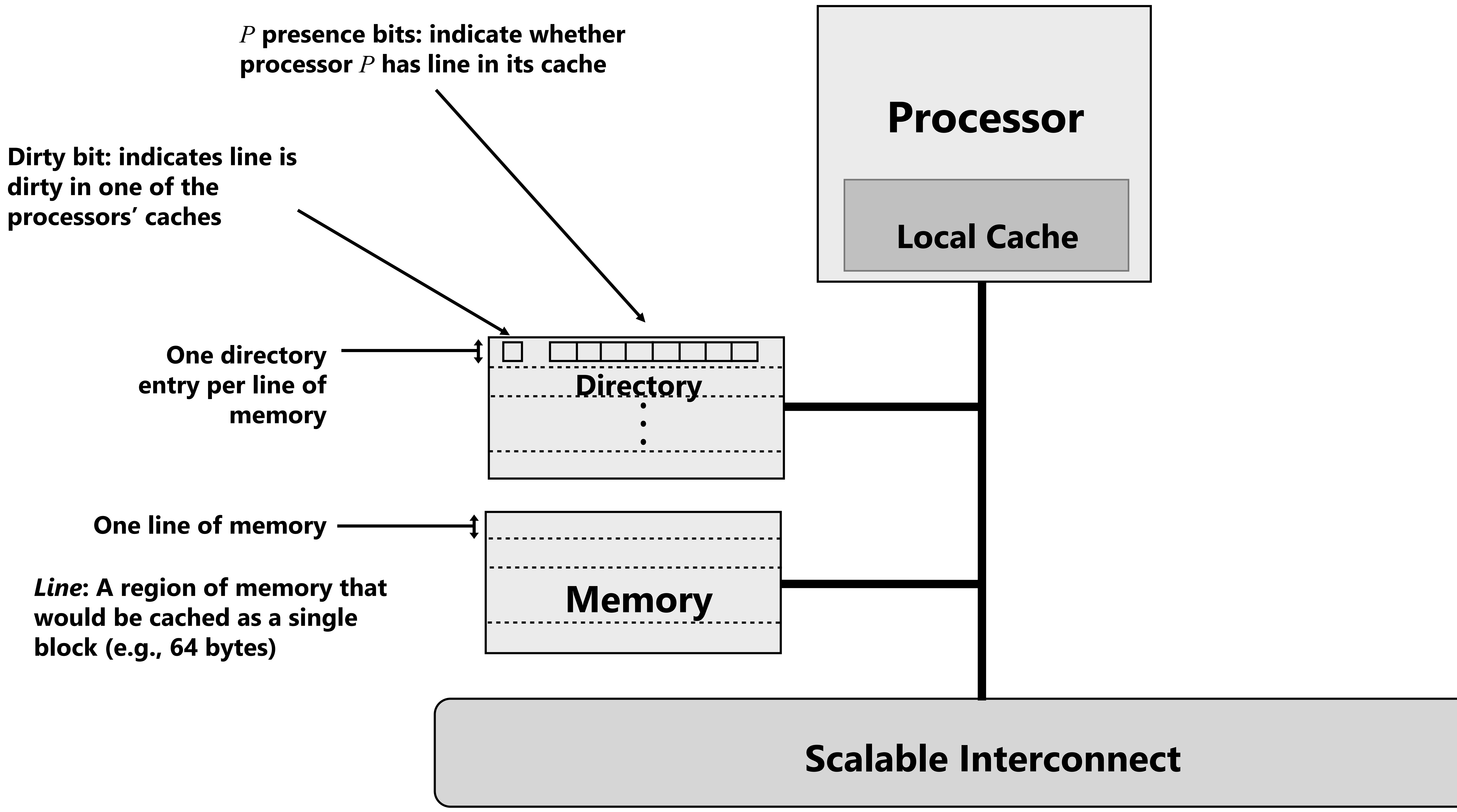
- **The root of the network can become a bottleneck**
- **Larger latencies than direct communication**
- **Does not apply to more general network topologies (meshes, cubes)**



Scalable cache coherence using directories

- Snooping schemes broadcast coherence messages to determine the state of a line in the other caches
- Alternative idea: avoid broadcast by storing information about the status of the line in one place: a "directory"
  - The directory entry for a cache line contains information about the state of the cache line in all caches.
  - Caches look up information from the directory as necessary
  - Cache coherence is maintained by point-to-point messages between the caches on a "need to know" basis (not by broadcast mechanisms)

A very simple directory



P presence bits: indicate whether processor P has line in its cache
Dirty bit: indicates line is dirty in one of the processors' caches
Processor
Local Cache
One directory entry per line of memory
Directory
One line of memory
Memory
Scalable Interconnect

Line: A region of memory that would be cached as a single block (e.g., 64 bytes)

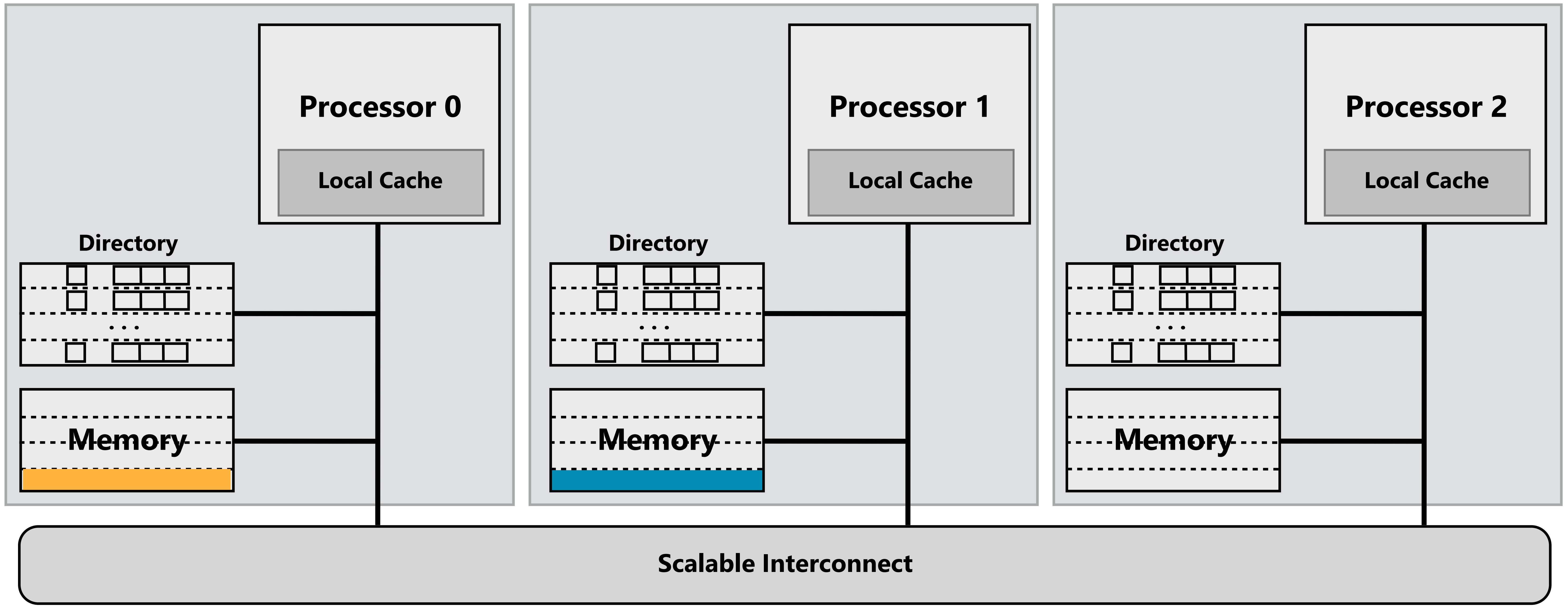
So we’ve traded a bus bottleneck for a memory bottleneck?

- **Not quite; directories distributed across memory banks**
  - ***Different* ordering points for *different* addresses**

- **Can cache directory entries + avoid memory access**
  - **Caches are much faster + higher bandwidth than memory**

A distributed directory in ccNUMA

Example: directory partition is co-located with memory it describes

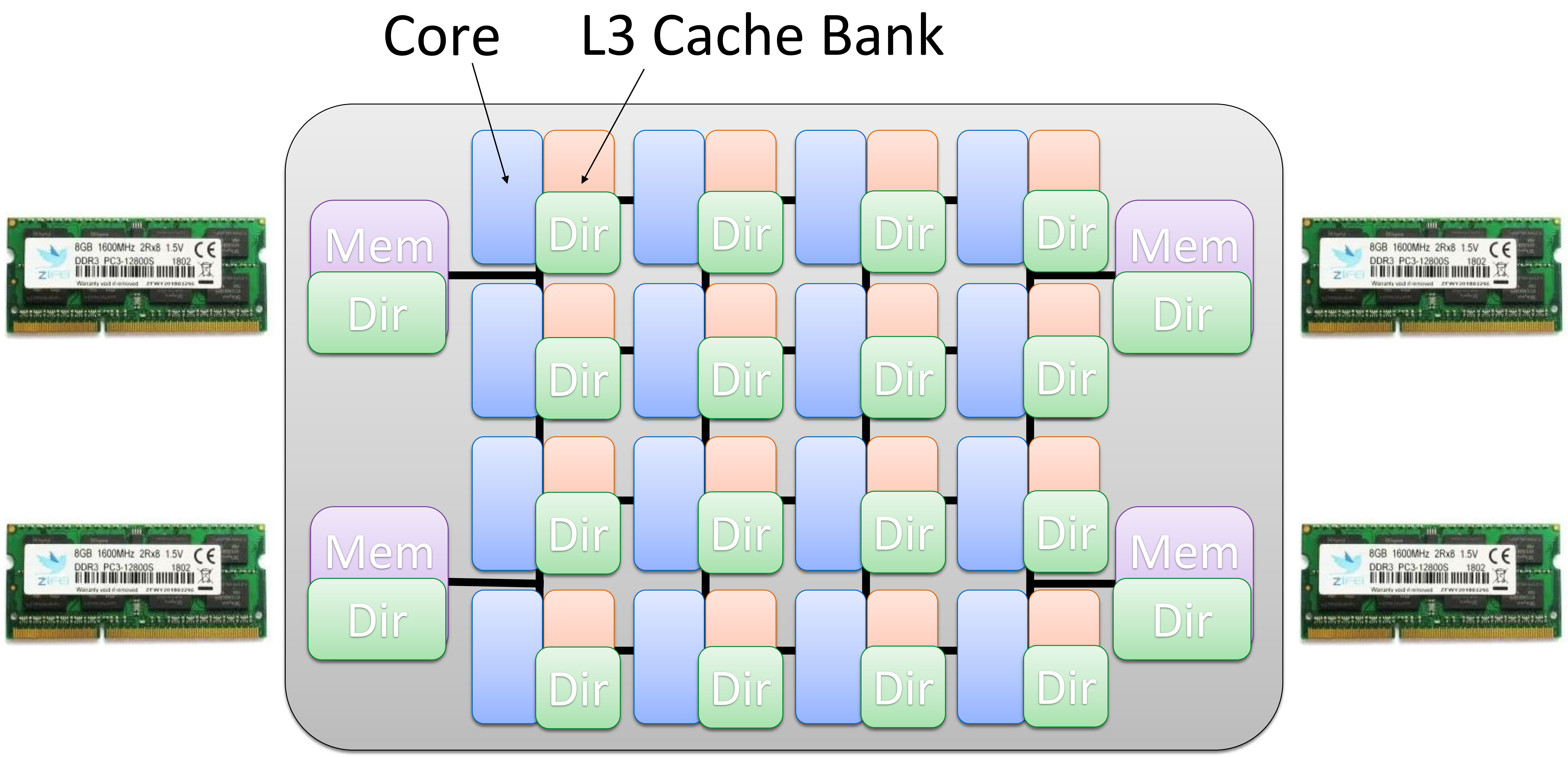


- **"Home node" of a line: node with memory holding the corresponding data for the line**

Example: node 0 is the home node of the yellow line, node 1 is the home node of the blue line

- **"Requesting node": node containing processor requesting line**

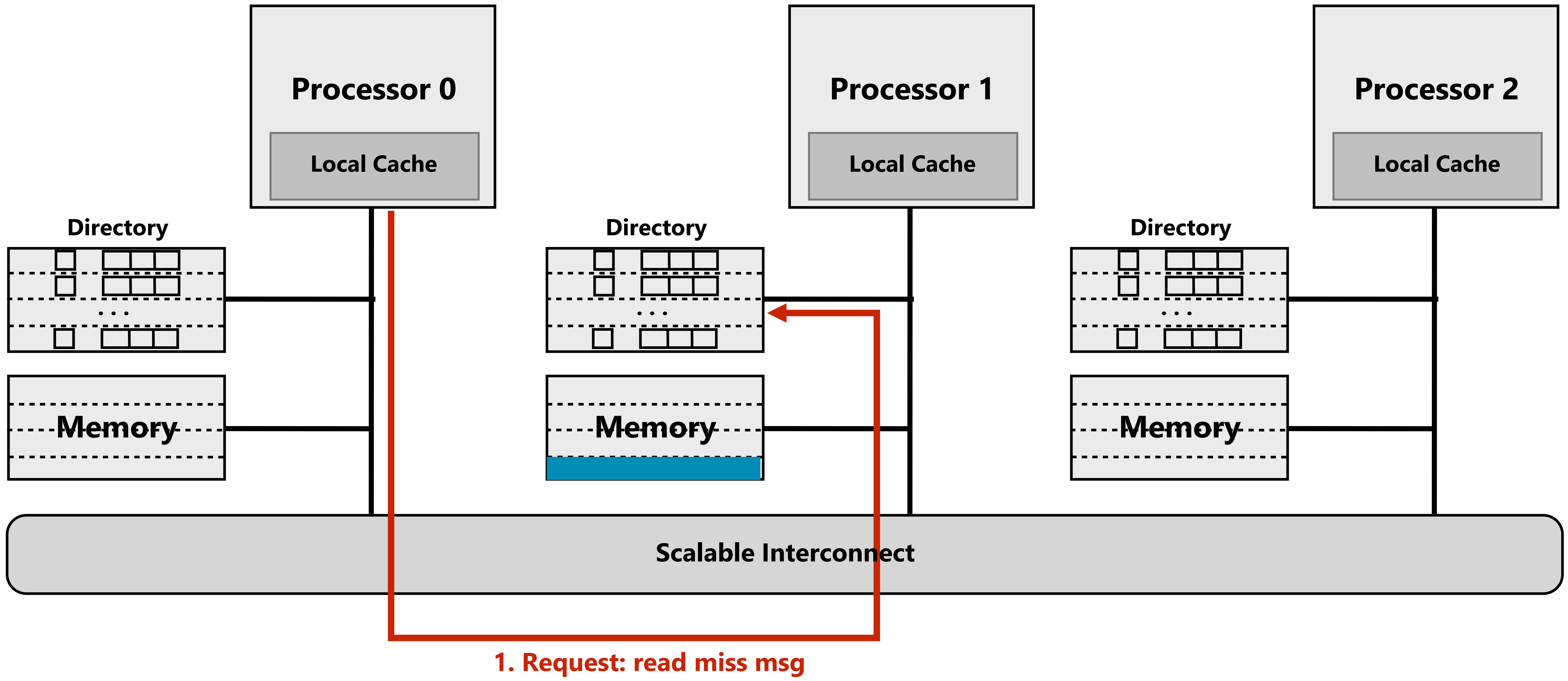
A distributed directory in a multicore



- **As we shall see, directories really live in each L3 bank**

Example 1: read miss to clean line

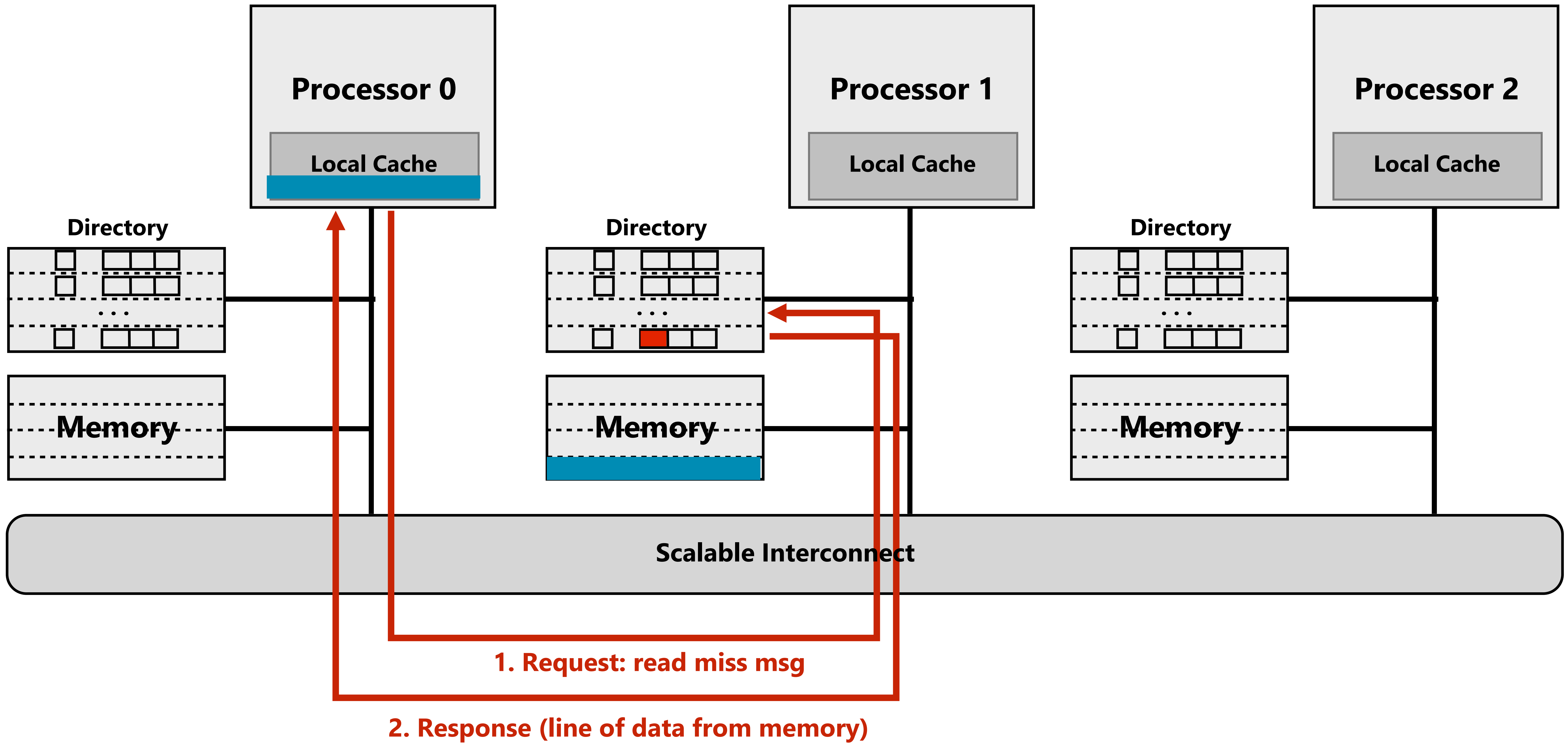
Read from main memory by processor 0 of the blue line: line is not dirty



- **Read miss message sent to home node of the requested line**
- **Home directory checks entry for line**

Example 1: read miss to clean line

Read from main memory by processor 0 of the blue line: line is not dirty

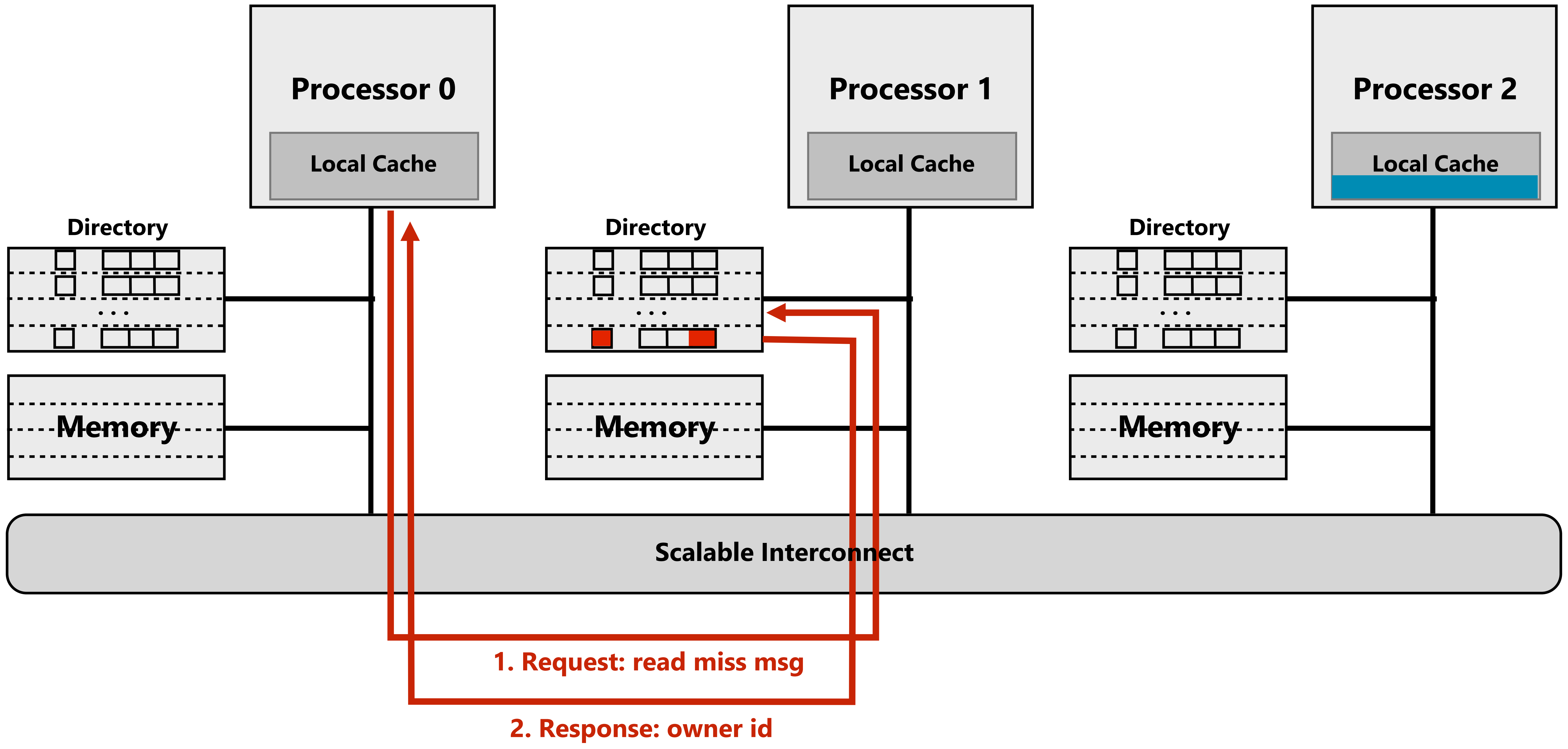


- **Read miss message sent to home node of the requested line**
- **Home directory checks entry for line**
  - **If dirty bit for cache line is OFF, respond with contents from memory, set presence[0] to true**

(to indicate line is cached by processor 0)

Example 2: read miss to dirty line

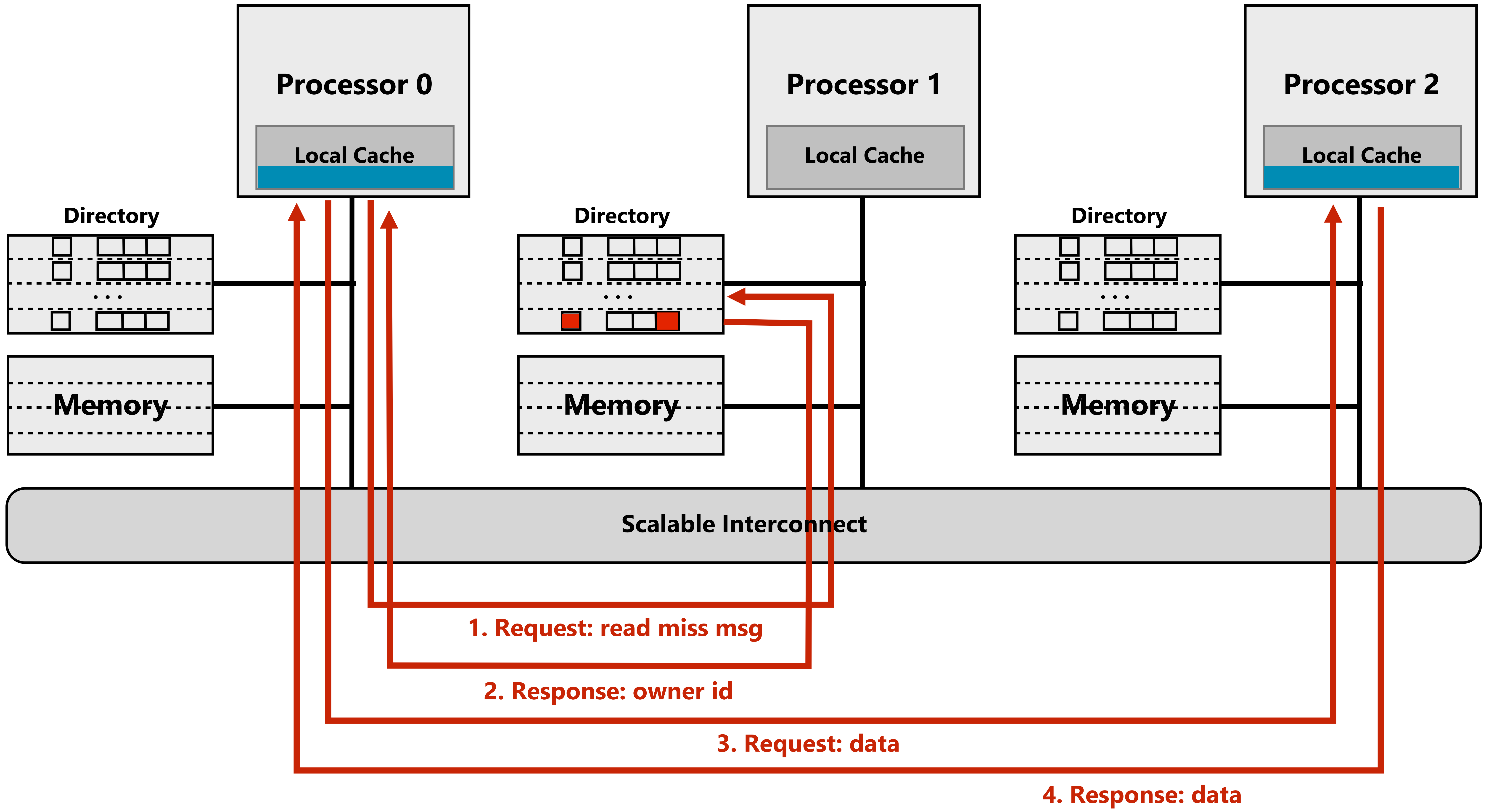
Read from main memory by processor 0 of the blue line: line is dirty (contents in P2's cache)



- If dirty bit is ON, then data must be sourced by another processor (with the most up-to-date copy of the line)
- Home node must tell requesting node where to find data
  - Responds with message providing identity of line owner ("get it from P2")

Example 2: read miss to dirty line

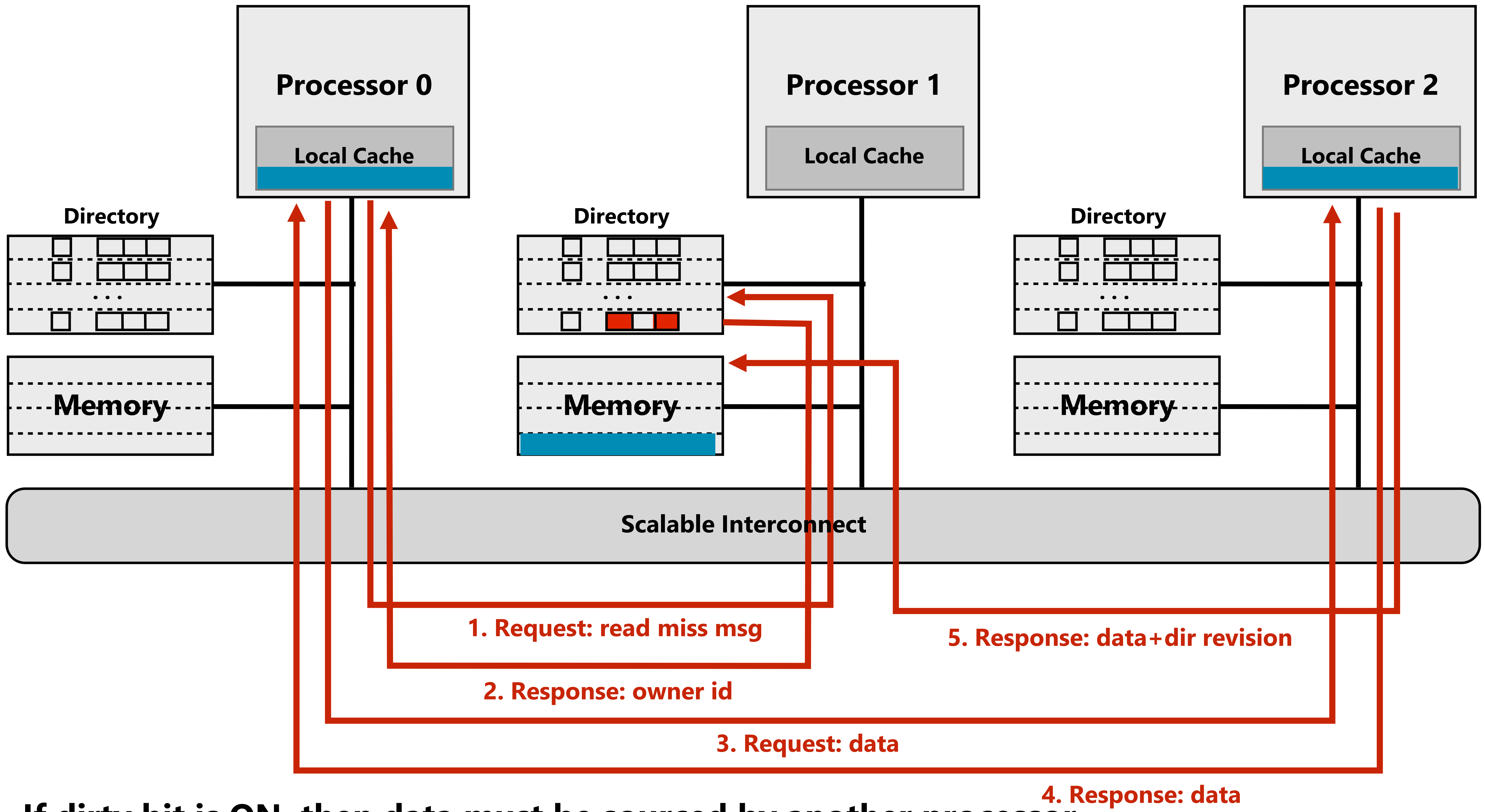
Read from main memory by processor 0 of the blue line: line is dirty (contents in P2's cache)



1. If dirty bit is ON, then data must be sourced by another processor
2. Home node responds with message providing identity of line owner
3. Requesting node requests data from owner
4. Owner changes state in cache to SHARED (read only), responds to requesting node

Example 2: read miss to dirty line

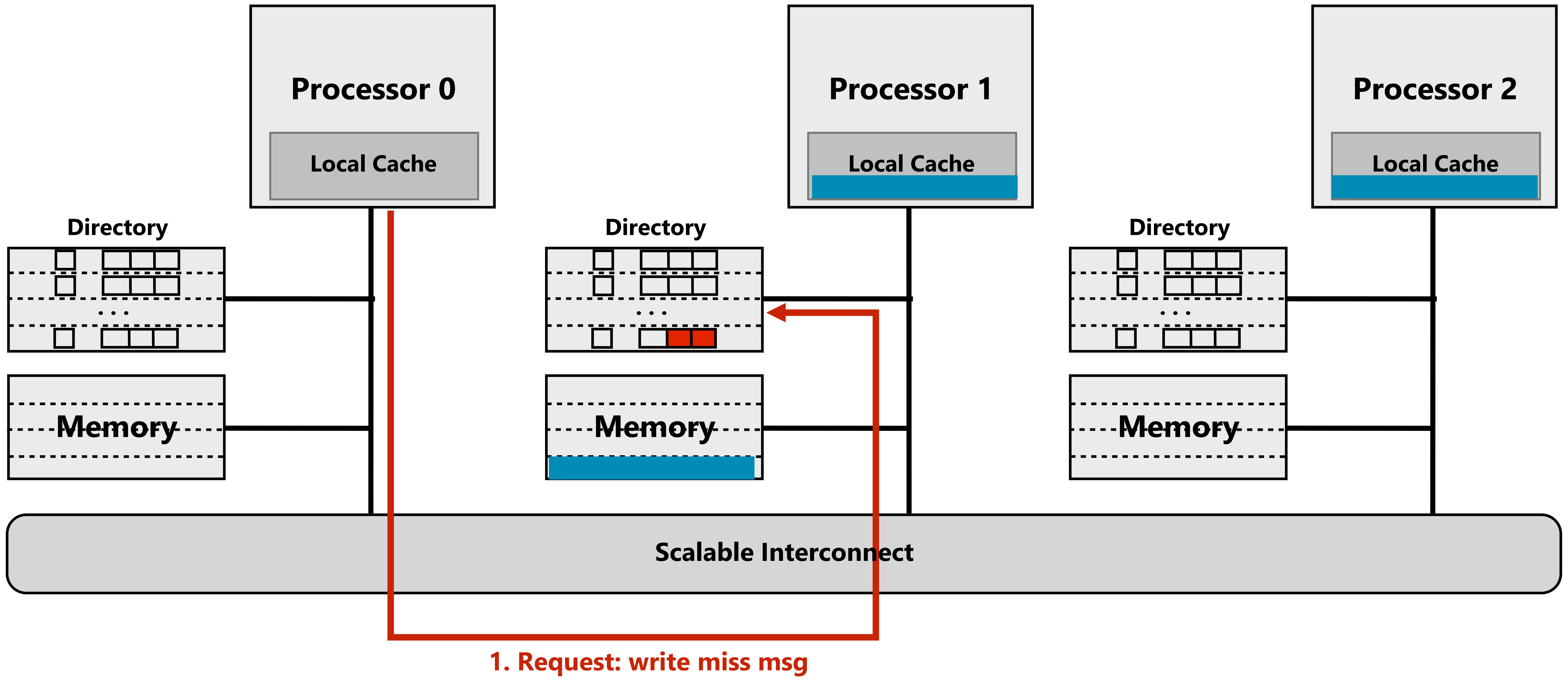
Read from main memory by processor 0 of the blue line: line is dirty (contents in P2's cache)



1. If dirty bit is ON, then data must be sourced by another processor
2. Home node responds with message providing identity of line owner
3. Requesting node requests data from owner
4. Owner responds to requesting node, changes state in cache to SHARED (read only)
5. Owner also responds to home node, home clears dirty, updates presence bits, updates memory

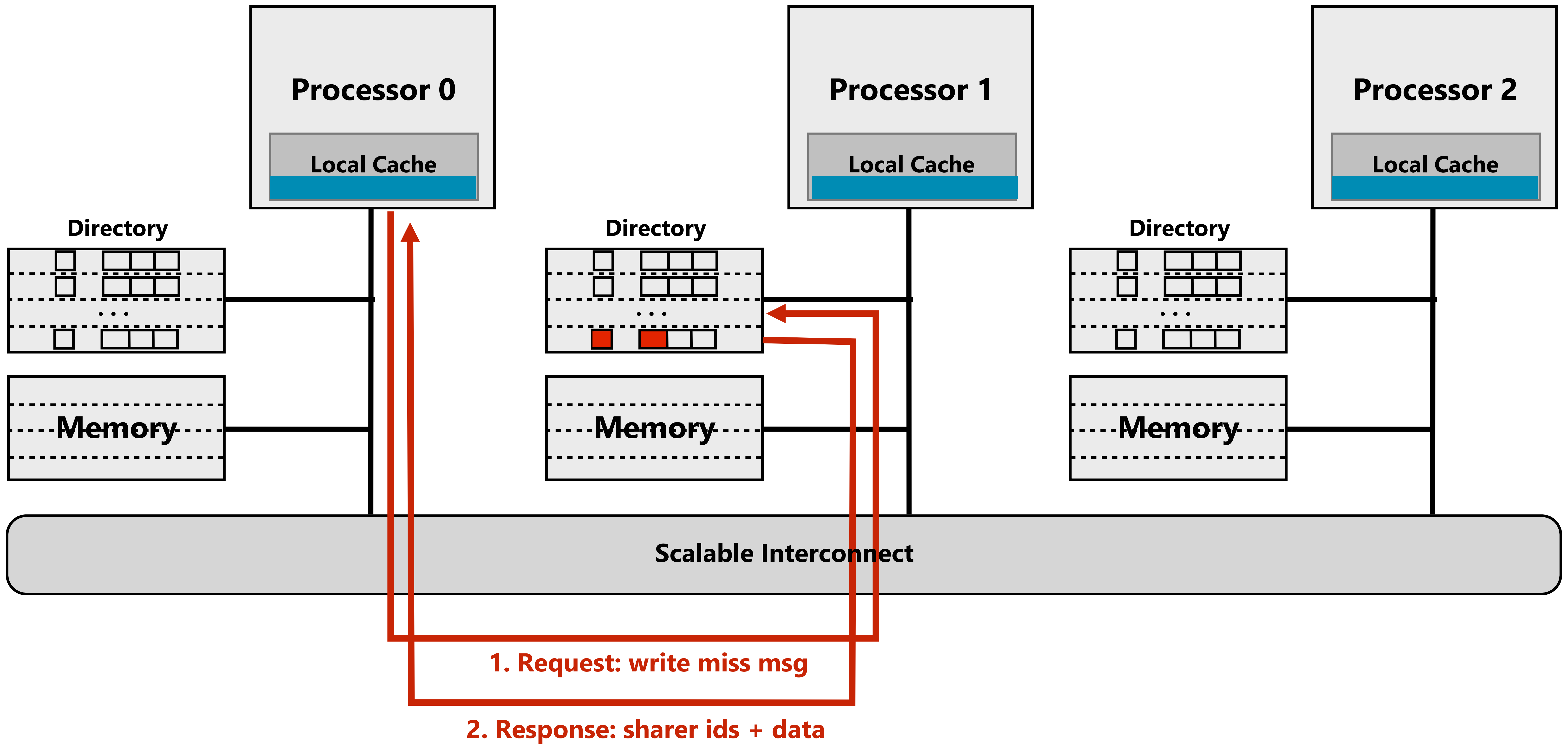
Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



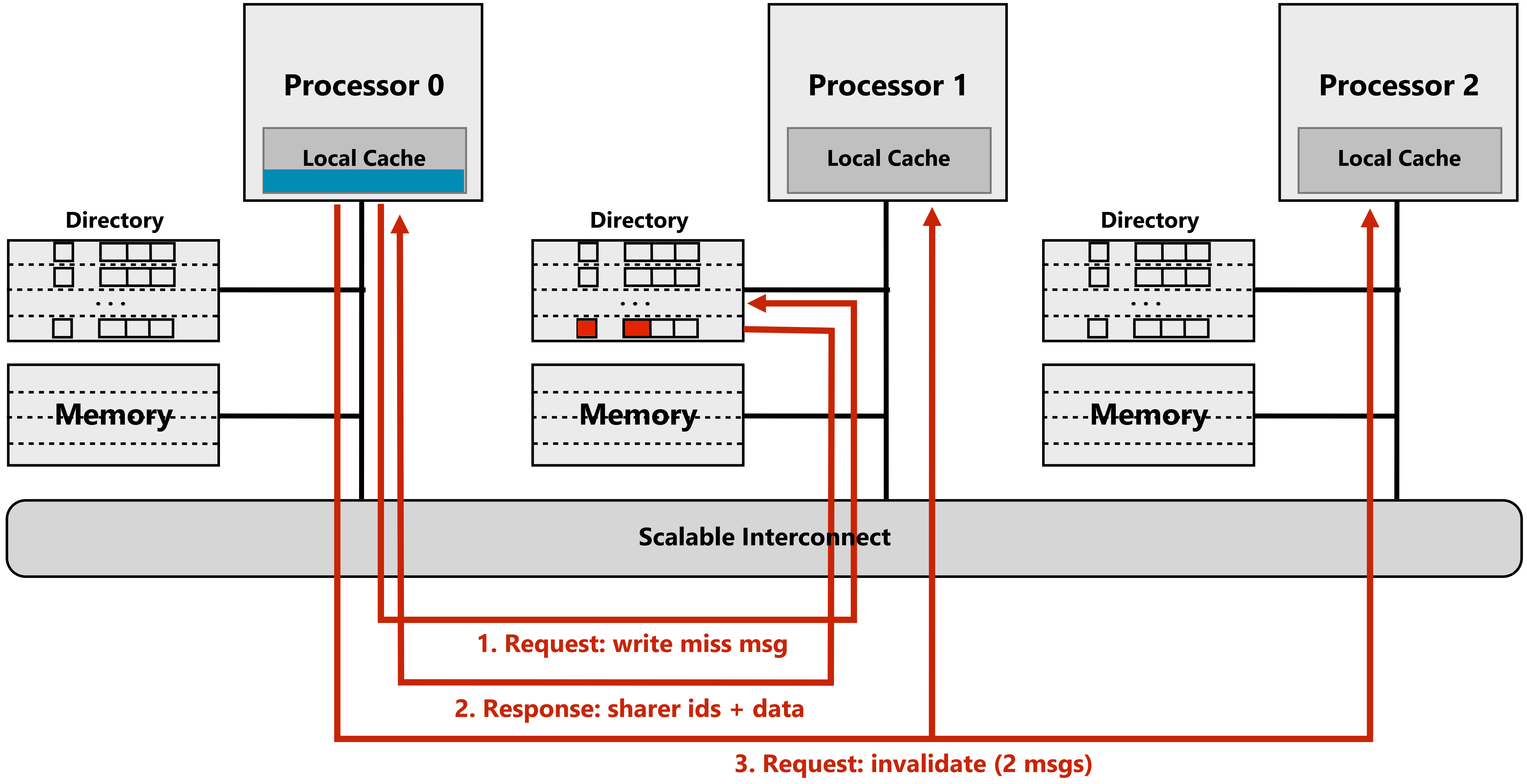
Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1′s and P2′s caches



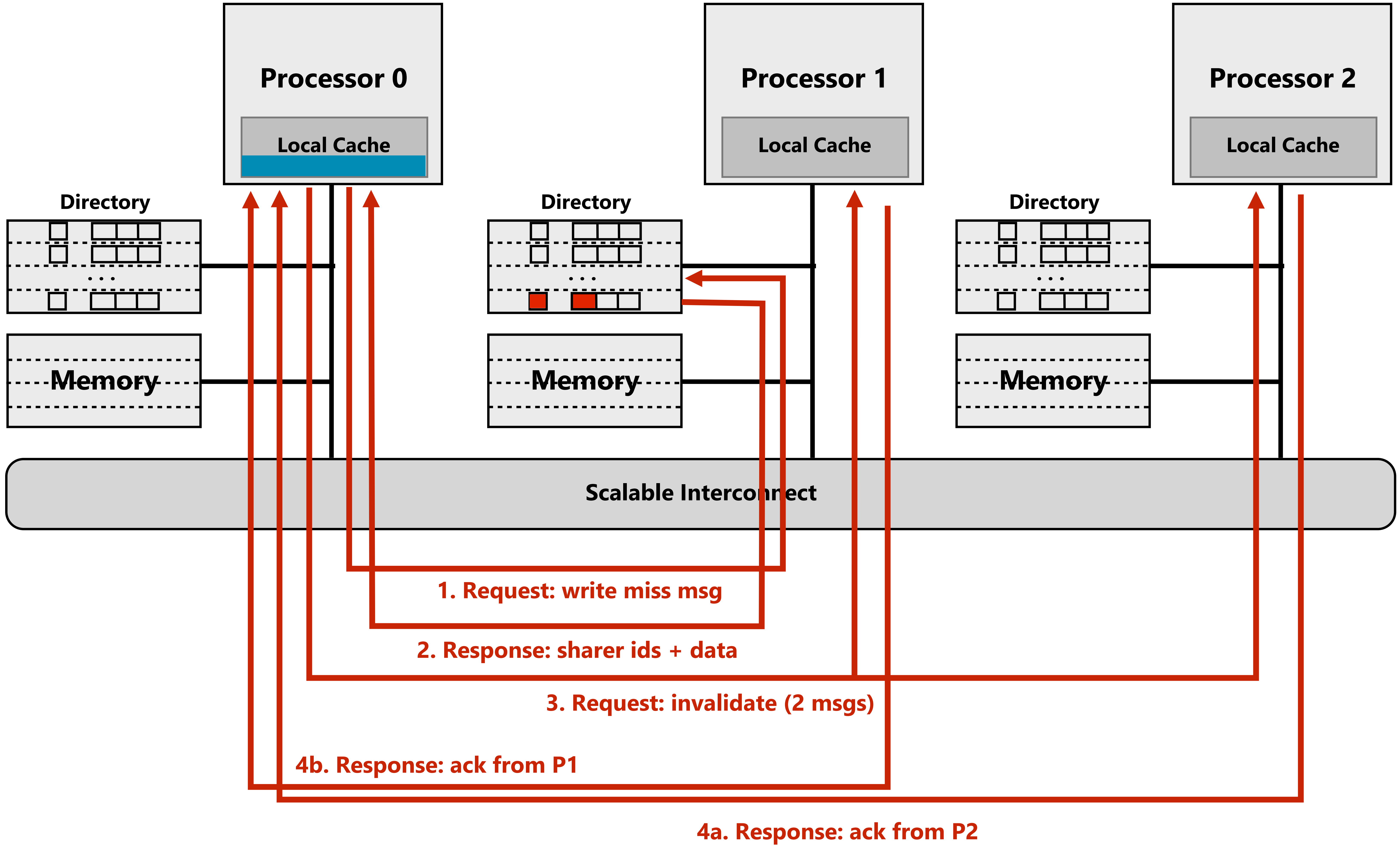
Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches


Processor 0
Local Cache
Processor 1
Local Cache
Processor 2
Local Cache
Directory
Directory
Directory
Memory
Memory
Memory
Scalable Interconnect
1. Request: write miss msg
2. Response: sharer ids + data
3. Request: invalidate (2 msgs)
4b. Response: ack from P1
4a. Response: ack from P2

After receiving both invalidation acks, P0 can perform write

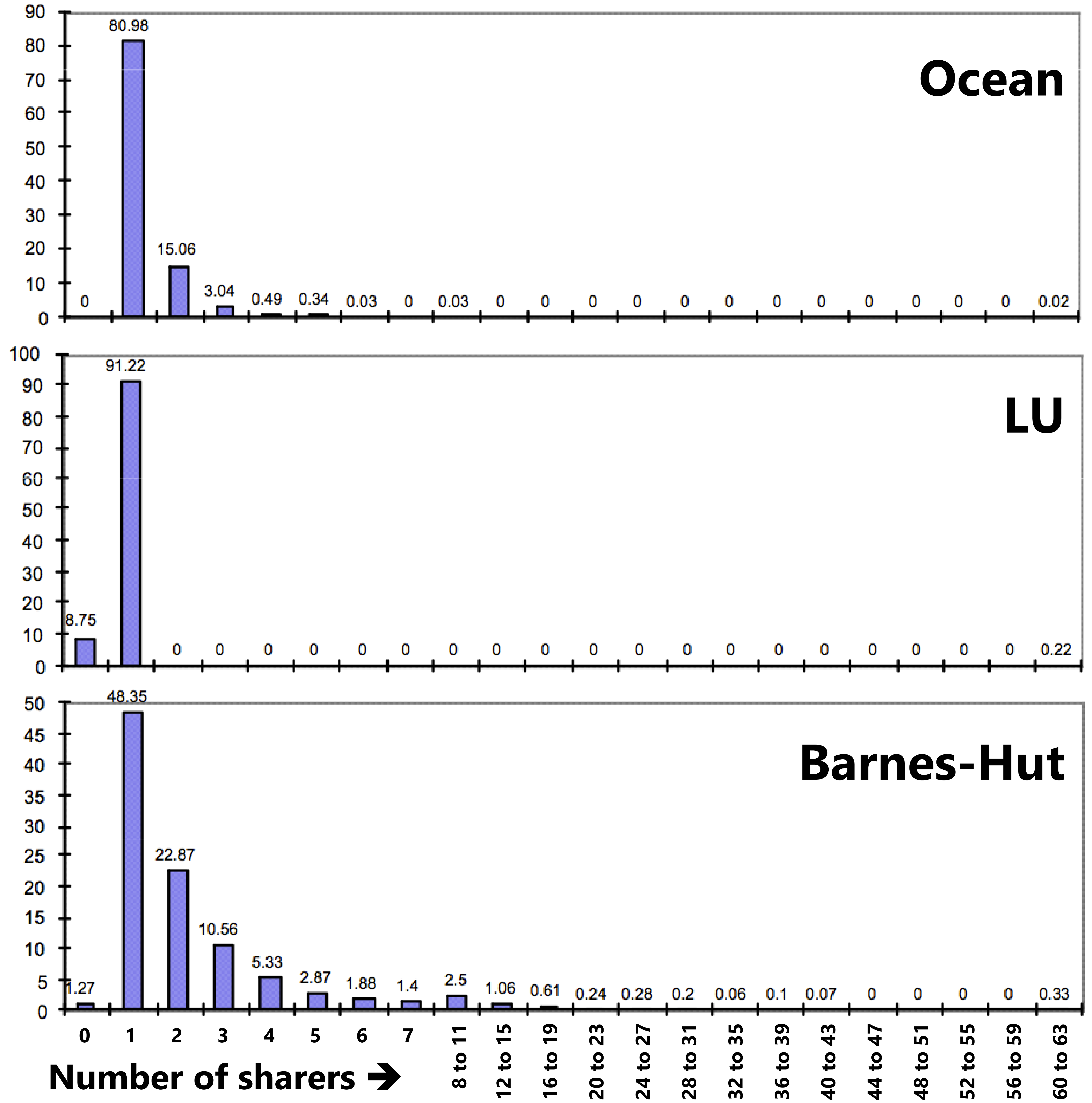
Advantage of directories

- On reads, directory tells requesting node exactly where to get the line from
  - Either from home node (if the line is clean)
  - Or from the owning node (if the line is dirty)
  - Either way, retrieving data involves only point-to-point communication

- On writes, the advantage of directories depends on the number of sharers
  - In the limit, if all caches are sharing data, all caches must be communicated with (just like broadcast in a snooping protocol)

Cache invalidation patterns

64 processor system



Graphs plot histogram of number of sharers of a line at the time of a write

In general only a few processors share the line (only a few processors must be told of writes)

Not shown here, but the expected number of sharers typically increases slowly with P (good!)

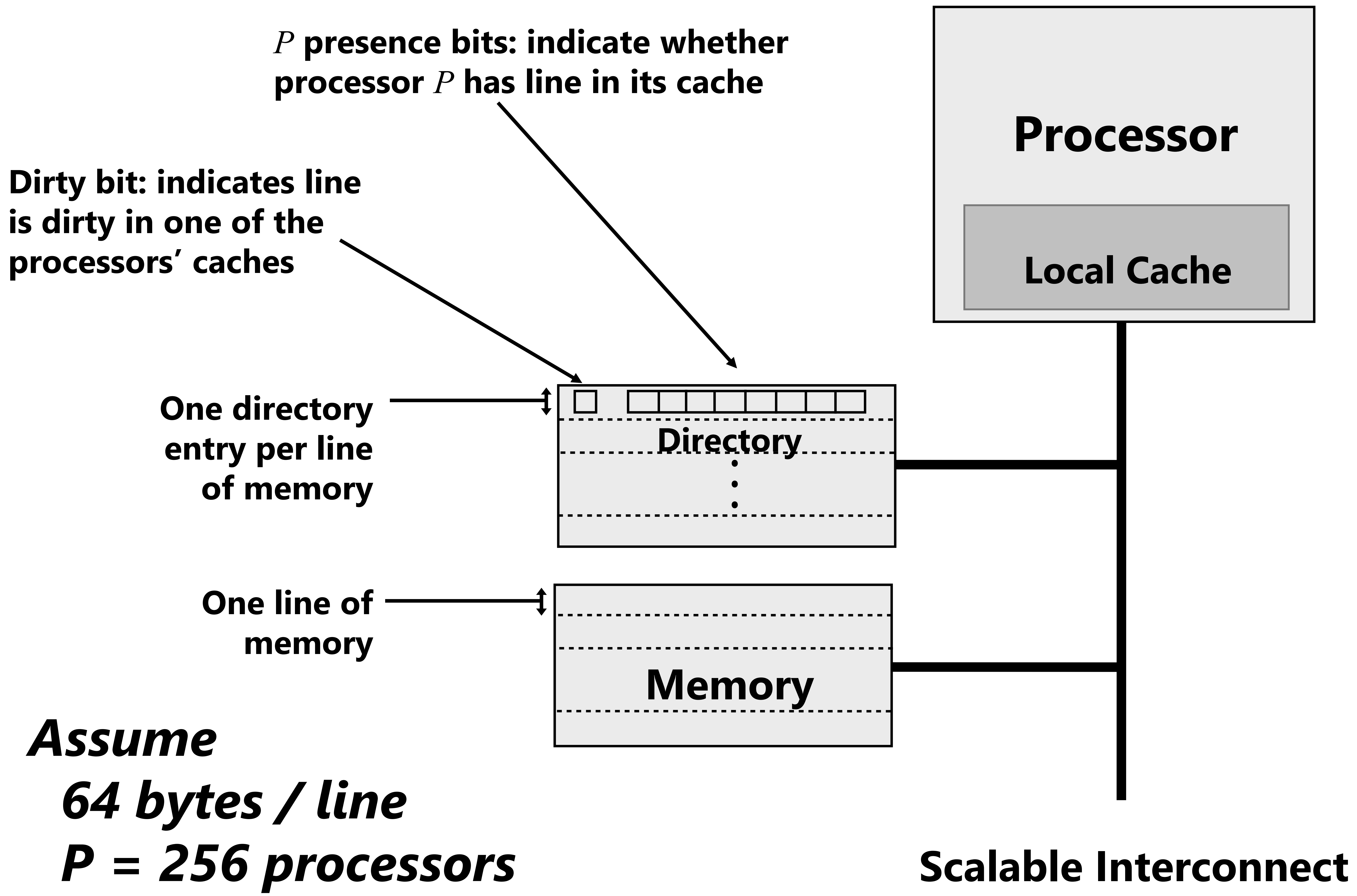
In general, only a few sharers during a write

- **Access patterns**
  - “Mostly-read” objects: lots of sharers but writes are infrequent, so minimal impact on performance (e.g., root node in Barnes-Hut)
  - Migratory objects (one processor reads/writes for while, then another, etc.): very few sharers, count does not scale with number of processors
  - Frequently read/written objects: frequent invalidations, but sharer count is low because count cannot build up in short time between invalidations (e.g, shared task queue)
  - Low-contention locks: infrequent invalidations, no performance problem
  - High-contention locks: can be a challenge, because many readers present when lock released

- **Implication 1: directories are useful for limiting coherence traffic**
  - Don’t need a broadcast mechanism to “tell everyone”

- **Implication 2: lets us limit directory storage overhead (how?)**

Very simple directory storage requirements


P presence bits: indicate whether processor P has line in its cache
Dirty bit: indicates line is dirty in one of the processors' caches
One directory entry per line of memory
Directory
One line of memory
Memory
Processor
Local Cache
Scalable Interconnect

Assume

64 bytes / line

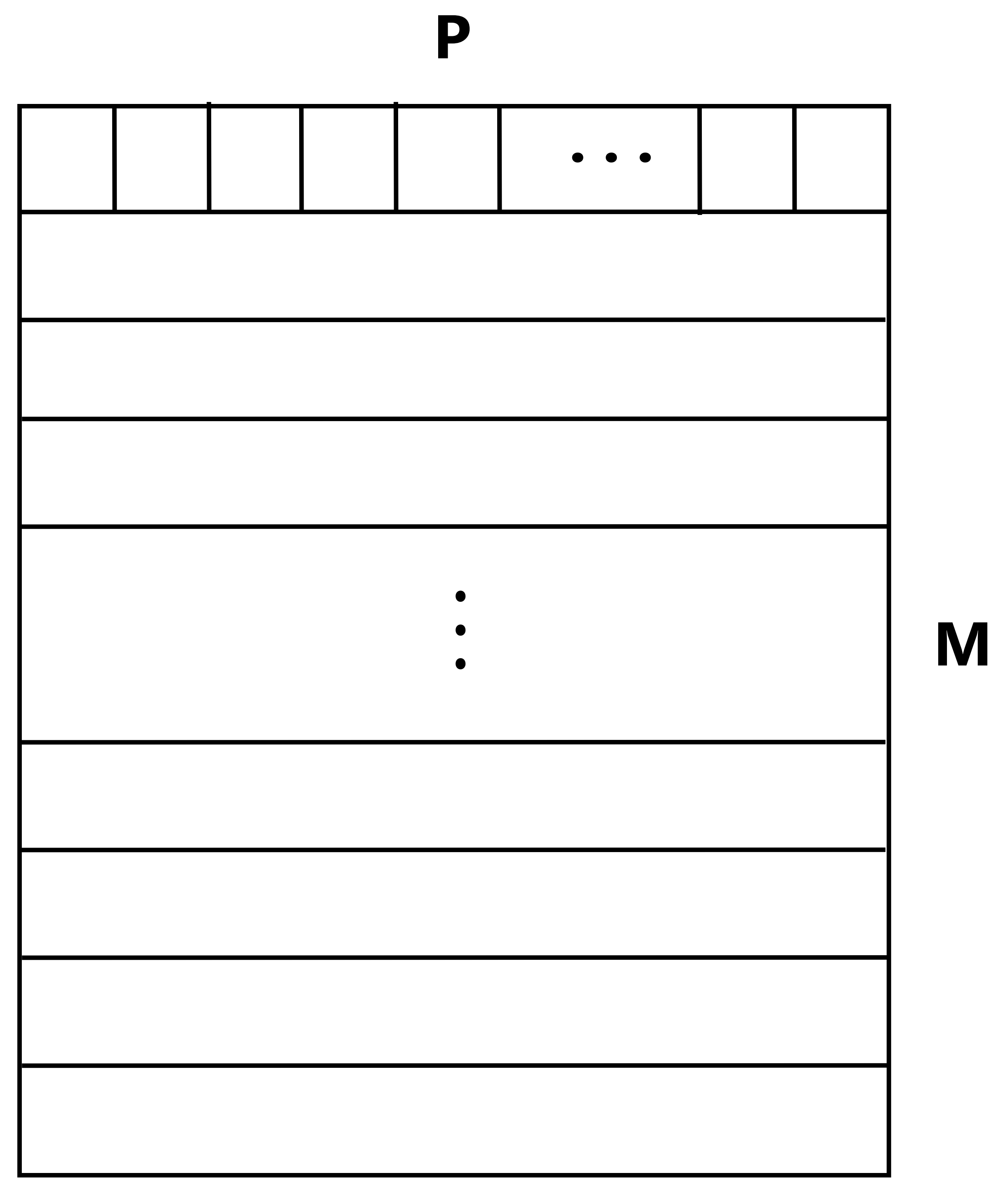
P = 256 processors

M = main memory size (bytes)

How big is directory?

Full-map directory representation

- **Recall: one presence bit per node**
- **Storage proportional to P x M**
  - P = number of nodes (e.g., processors)
  - M = number of lines in memory
  - # dir entries = M / 64B
  - # bytes / dir entry = P / 8
  - ➔ Dir size = (P * M) / (8 * 64)
- **Storage overhead rises with P**
  - Assume 64 byte cache line size (512 bits)
  - 64 nodes (P=64) →12% overhead
  - 256 nodes (P=256) → 50% overhead
  - 1024 nodes (P=1024) → 200% overhead



Reducing storage overhead of directory

- Optimizations on full-map directory scheme
  - Increase cache line size (reduce M term)
    - What are the downsides of this approach? (consider graphs from last lecture)
  - Group multiple processors into a single directory "node" (reduce P term)
    - Need only one directory bit per node, not one bit per processor
    - Hierarchical: could use snooping protocol to maintain coherence among processors in a node
    - But hierarchical coherence is very complicated + adds latency
- We will now discuss two alternative schemes
  - Limited pointer schemes (reduce P)
  - Sparse directories

Limited directory

Most data has few sharers
➔ Idea: Store only a few *pointers* in directory
(we only need a list of the nodes holding a valid copy of the line!)

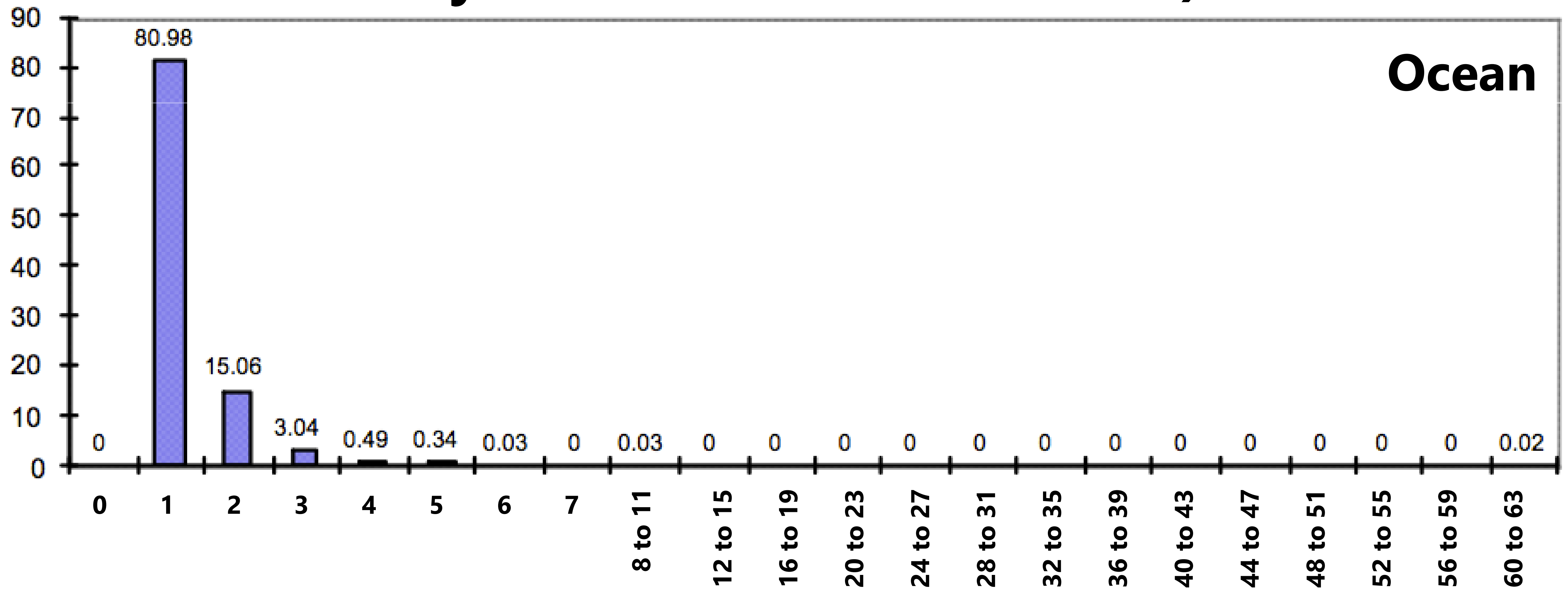
Example: 1024 processor system

Full bit vector scheme needs 1024 bits per line

Instead, we can track most accesses with a few pointers to sharers

Each pointer is $\log_2(1024) = 10\text{b}$, so anything <100 pointers is a win!

(E.g., >99% of writes with just a five sharers in Ocean)



Managing overflow in limited pointer schemes

Many possible approaches

- **Fallback to broadcast**

- **Replaces an existing sharer (invalidating its cache) with the new sharer**

- **Coarse bit-vector**
  - **Revert to bit vector representation**
  - **Each bit corresponds to a cluster of *K* processors**
  - **On write, invalidate all processors in the cluster**

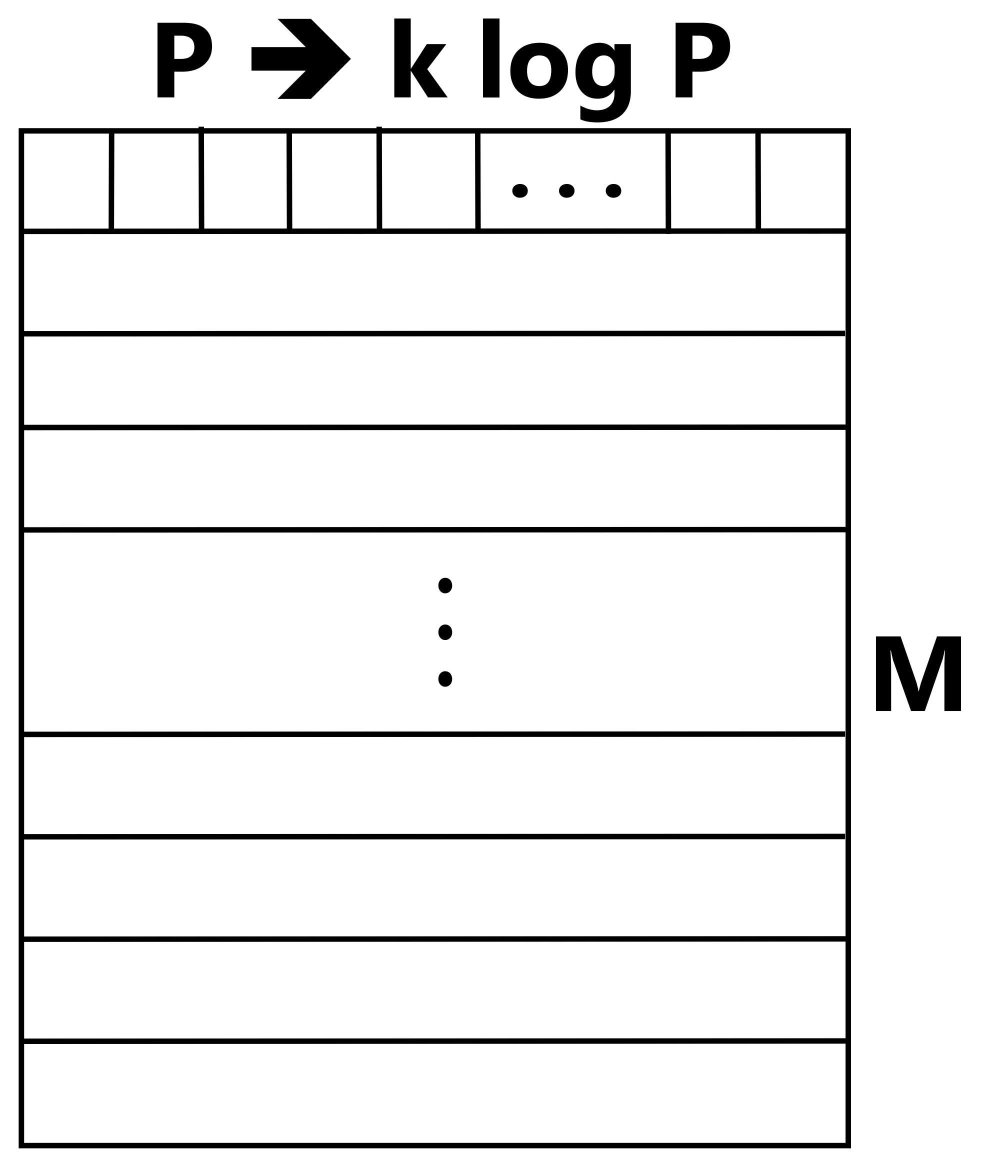
Optimize for the common case!

Limited pointer schemes are a great example of understanding and optimizing for the common case:

1. **Workload-driven observation: in general the number of cache line sharers is low**
2. **Make the common case simple and fast: array of pointers for first N sharers**
3. **Uncommon case is still handled correctly, just with a slower, more complicated mechanism (the program still works!)**
4. **Extra expense of the complicated solution is tolerable, since it happens infrequently**

Limited pointer schemes: summary

- **Limited pointer schemes reduce directory storage overhead caused by large P**
  - Entry size w/ k pointers: k log P bits
  - Overhead = (k log P * M) / (64 * 8)
  - E.g., if k = 5:
    - P = 64: 5.8% overhead
    - P = 256: 7.8% overhead
    - P = 1024: 9.7% overhead



Directory

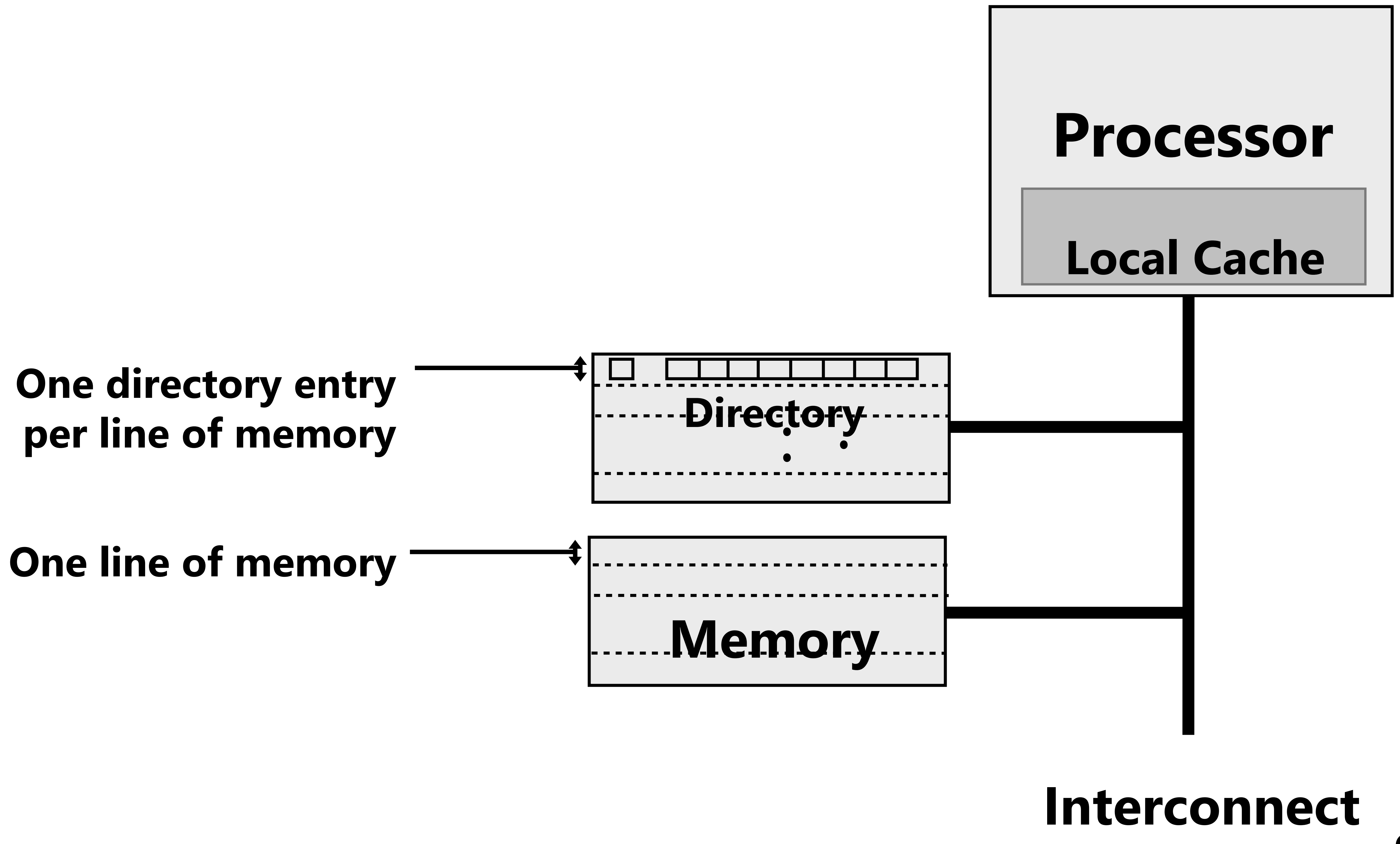
- **Most lines have few sharers, so this works well**

- **But do we really even need to maintain a list of sharers for each line of data in memory?**

Limiting size of directory: sparse directories

Key observations:
1) The coherence protocol only needs sharing information for lines in *some* cache.
2) Most memory is NOT resident in cache.
➔ Nearly <u>all</u> directory entries are empty

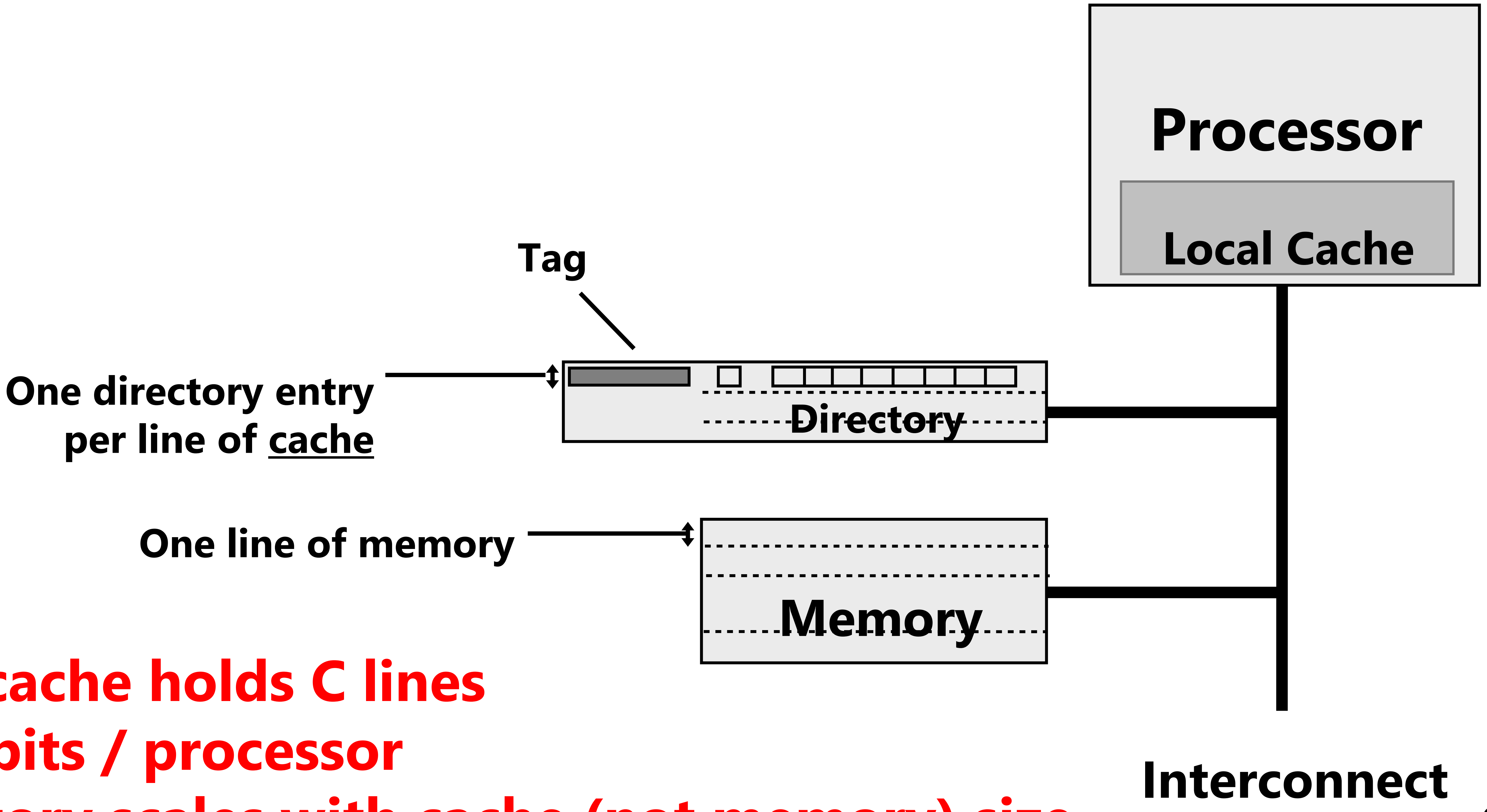
- **E.g., 1 MB cache, 1 GB memory → ≥ 99.9% of entries are empty**



Sparse directories: Reducing directory entries

Key observations:
1) The coherence protocol only needs sharing information for lines in *some* cache.
2) Most memory is NOT resident in cache.
➔ Almost <u>all</u> directory entries are empty

- **E.g., 1 MB cache, 1 GB memory → ≥ 99.9% of entries are empty**

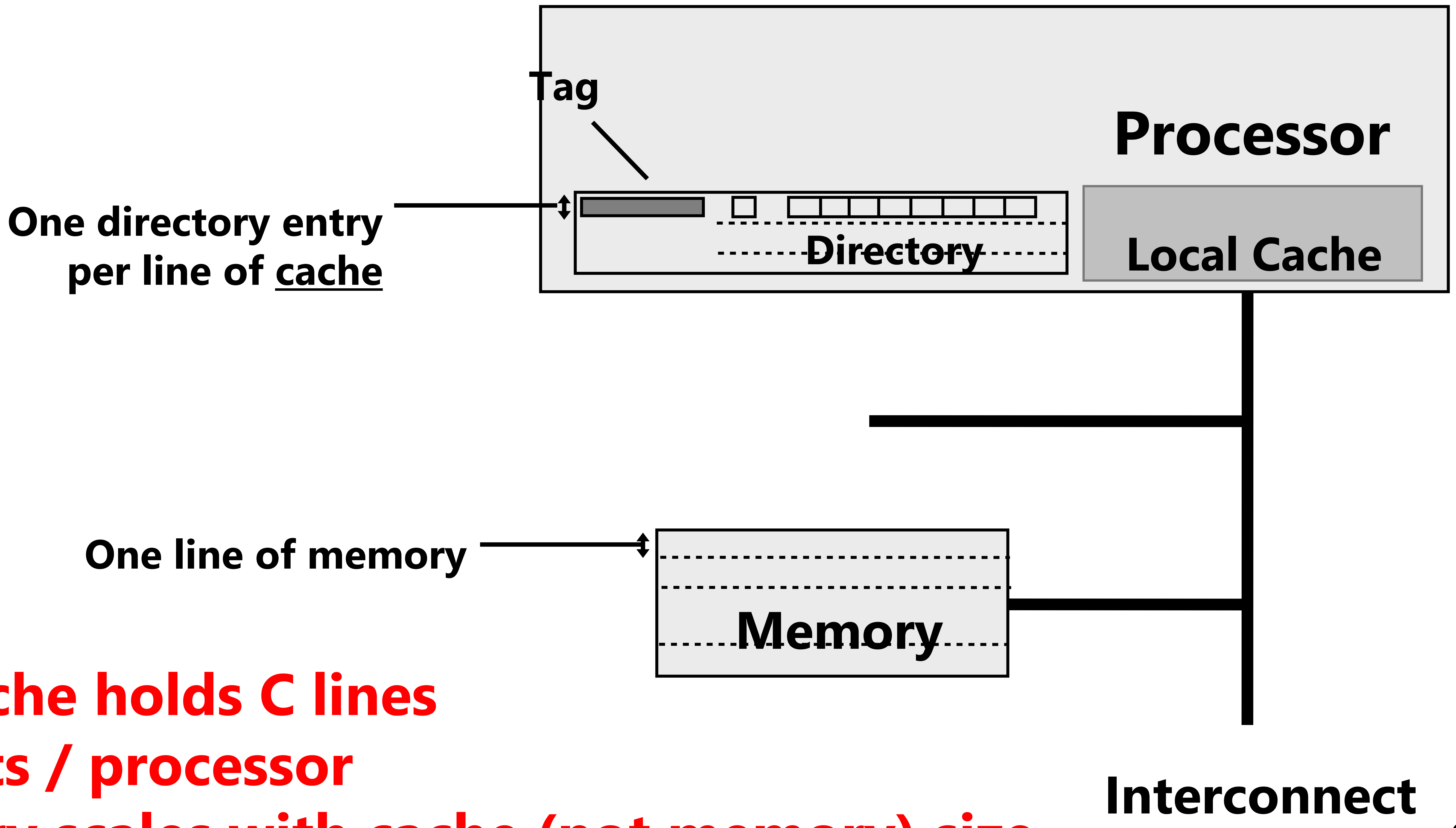


Scaling:
- **Each cache holds C lines**
- **P * C bits / processor**
- **Directory scales with <u>cache</u> (not memory) size**

Sparse directories: Reducing directory entries

Key observations:
1) The coherence protocol only needs sharing information for lines in *some* cache.
2) Most memory is NOT resident in cache.
➔ Almost <u>all</u> directory entries are empty

- **E.g., 1 MB cache, 1 GB memory → ≥ 99.9% of entries are empty**



Scaling:
- **Each cache holds C lines**
- **P * C bits / processor**
- **Directory scales with <u>cache</u> (not memory) size**

Sparse directories: Limiting entry size

Idea: Store sharers as a linked list

Directory at home node stores only 1 pointer to head of the list (not the full list)

Pointer to next node in list is stored with cache line (alongside the line's tag, dirty bits, etc)

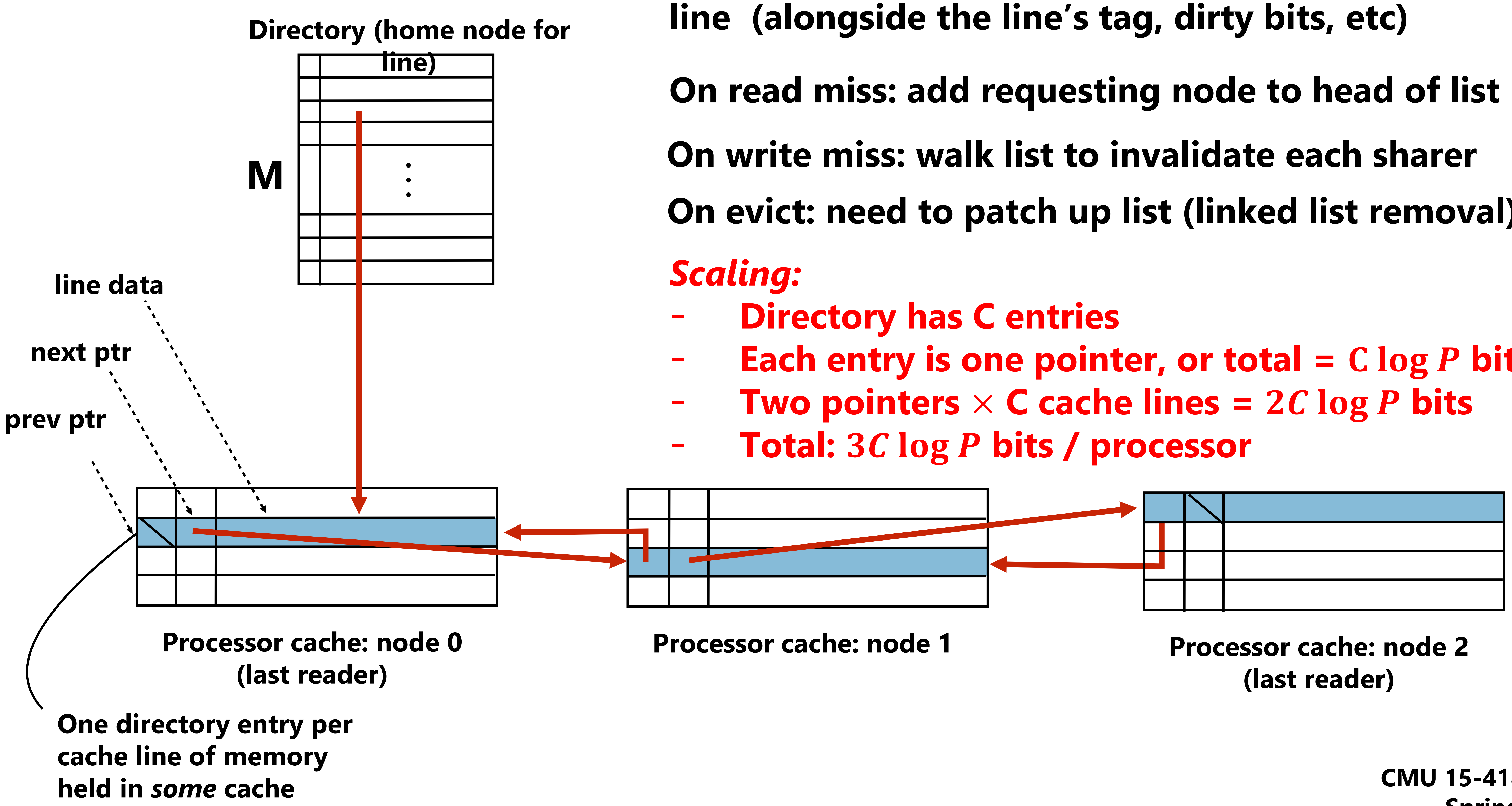
On read miss: add requesting node to head of list

On write miss: walk list to invalidate each sharer

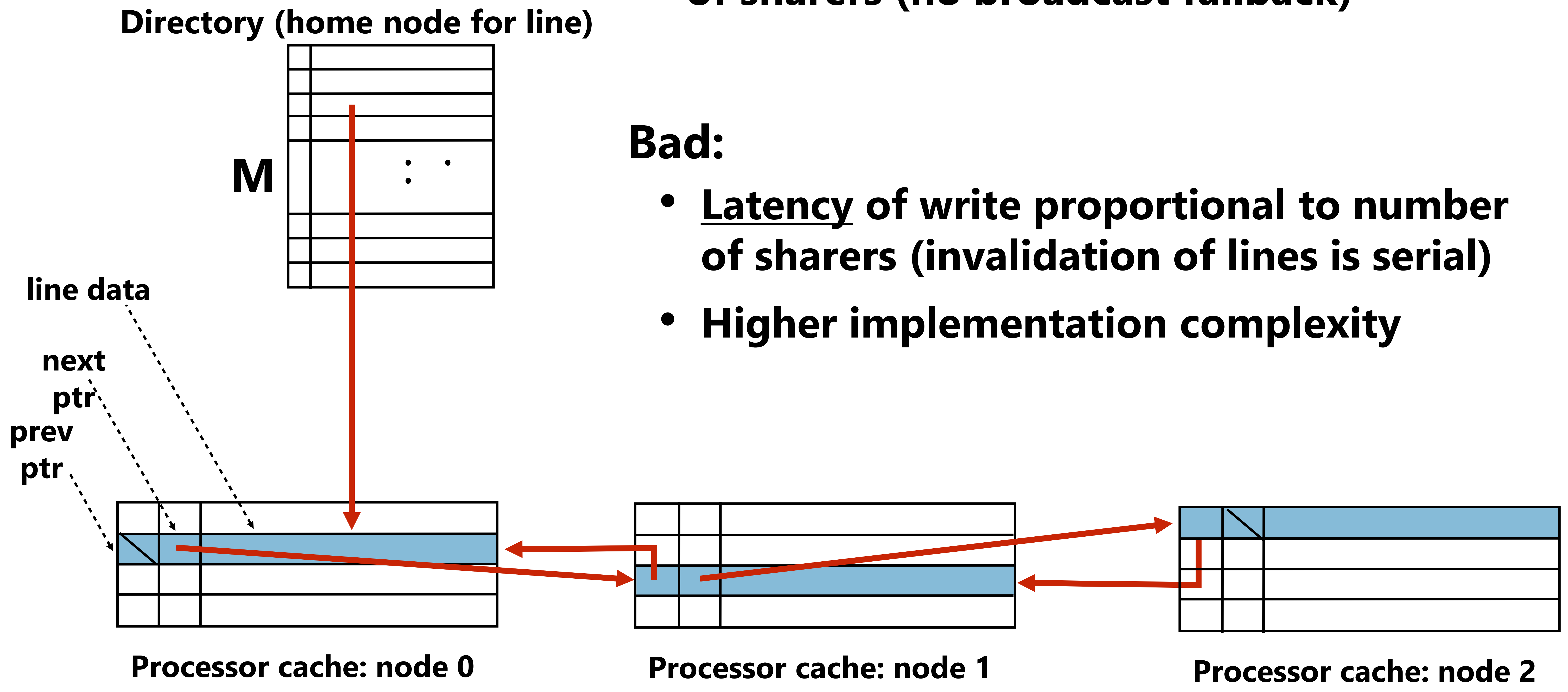
On evict: need to patch up list (linked list removal)

Scaling:

- Directory has C entries
- Each entry is one pointer, or total = $C \log P$ bits
- Two pointers × C cache lines = $2C \log P$ bits
- Total: $3C \log P$ bits / processor


Directory (home node for line)
M
line data
next ptr
prev ptr
Processor cache: node 0 (last reader)
Processor cache: node 1
Processor cache: node 2 (last reader)
One directory entry per cache line of memory held in *some* cache

Sparse directories: scaling properties



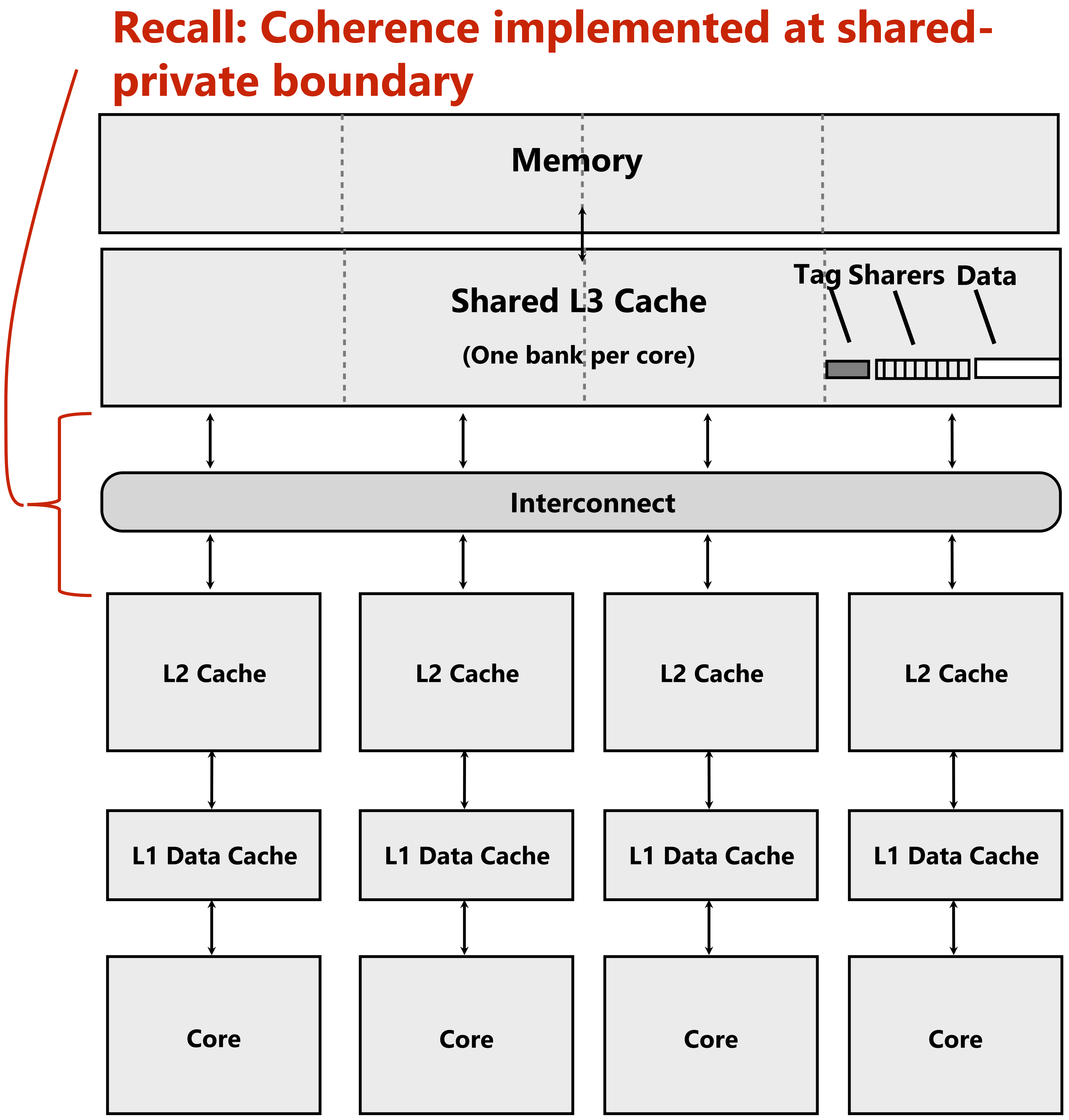
Good:

- Low memory storage overhead
- Traffic on write is proportional to number of sharers (no broadcast fallback)

Bad:

- Latency of write proportional to number of sharers (invalidation of lines is serial)
- Higher implementation complexity

In-cache directories



- **In-cache directory: Add directory entries to shared L3 cache**
  - $C_{L3} \times P$ **add'l bits**
  - **L3 must be inclusive**
- **Sparse directory: Add separately tagged structure to track L2 contents**
  - $P \times C_{L2} \times (T + P)$ **add'l bits**
  - **L3 need not be inclusive**
- **In-cache directories are common in multicores today**
  - **In-cache dir smaller if:**

$$\frac{C_{L3}}{C_{L2}} < T + P$$

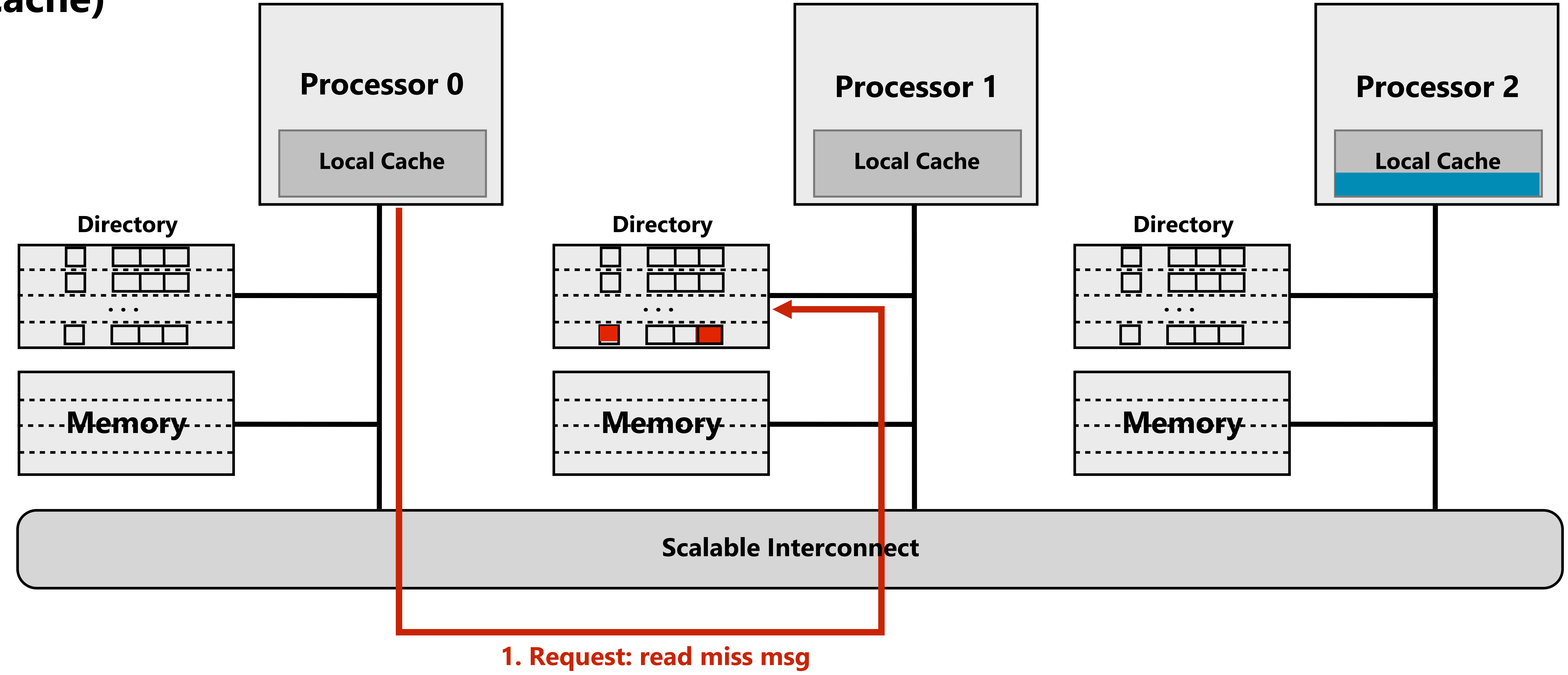
Optimizing directory-based coherence

- **Reducing storage overhead of directory data structure**
  - **Limited pointer schemes**
  - **Sparse directories**

- **Reducing number of messages sent to implement coherence protocol**

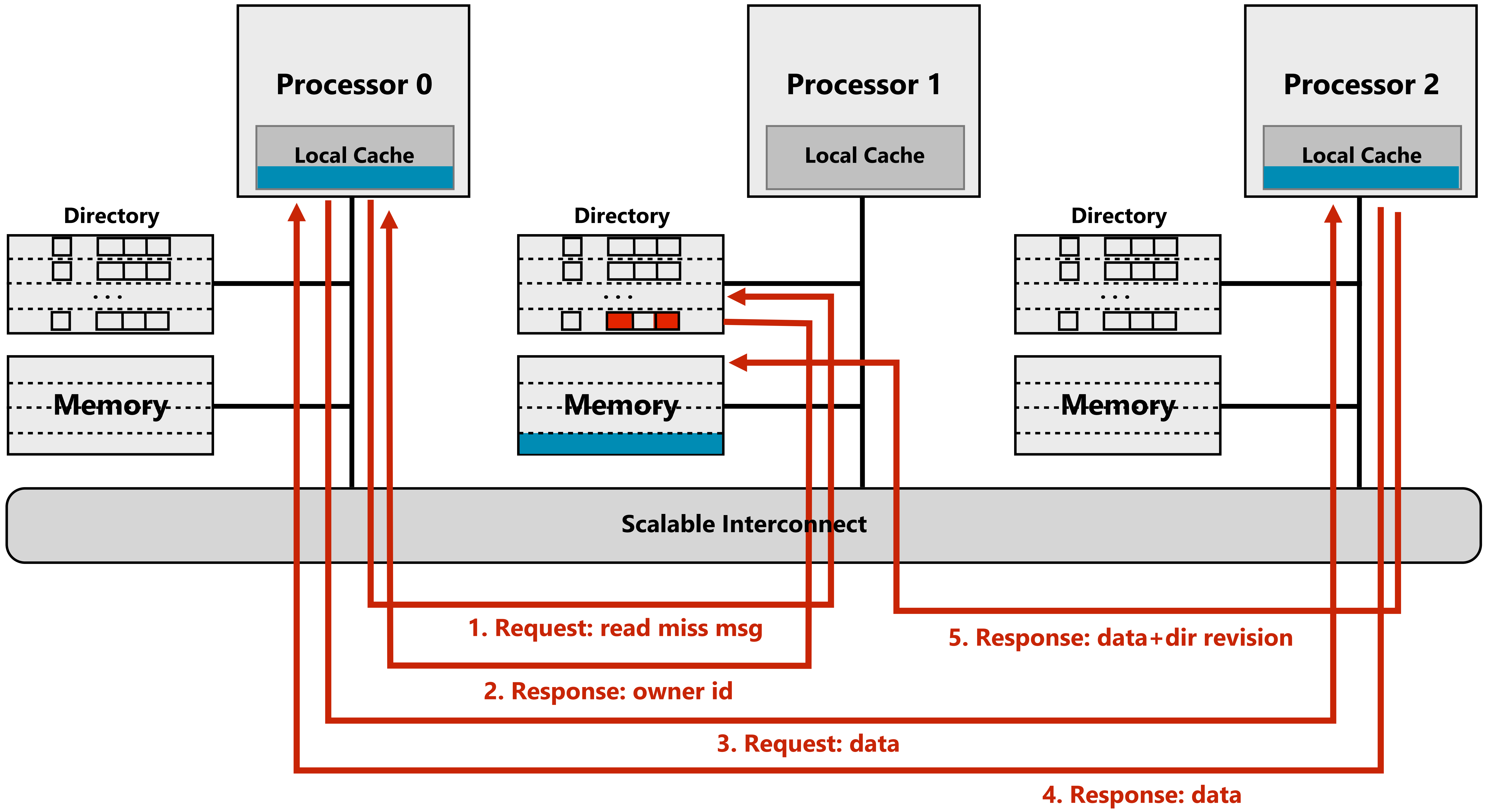
Recall: read miss to dirty line

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)



Recall: read miss to dirty line

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache) *(Note: figure below shows final state of system after operation is complete)*



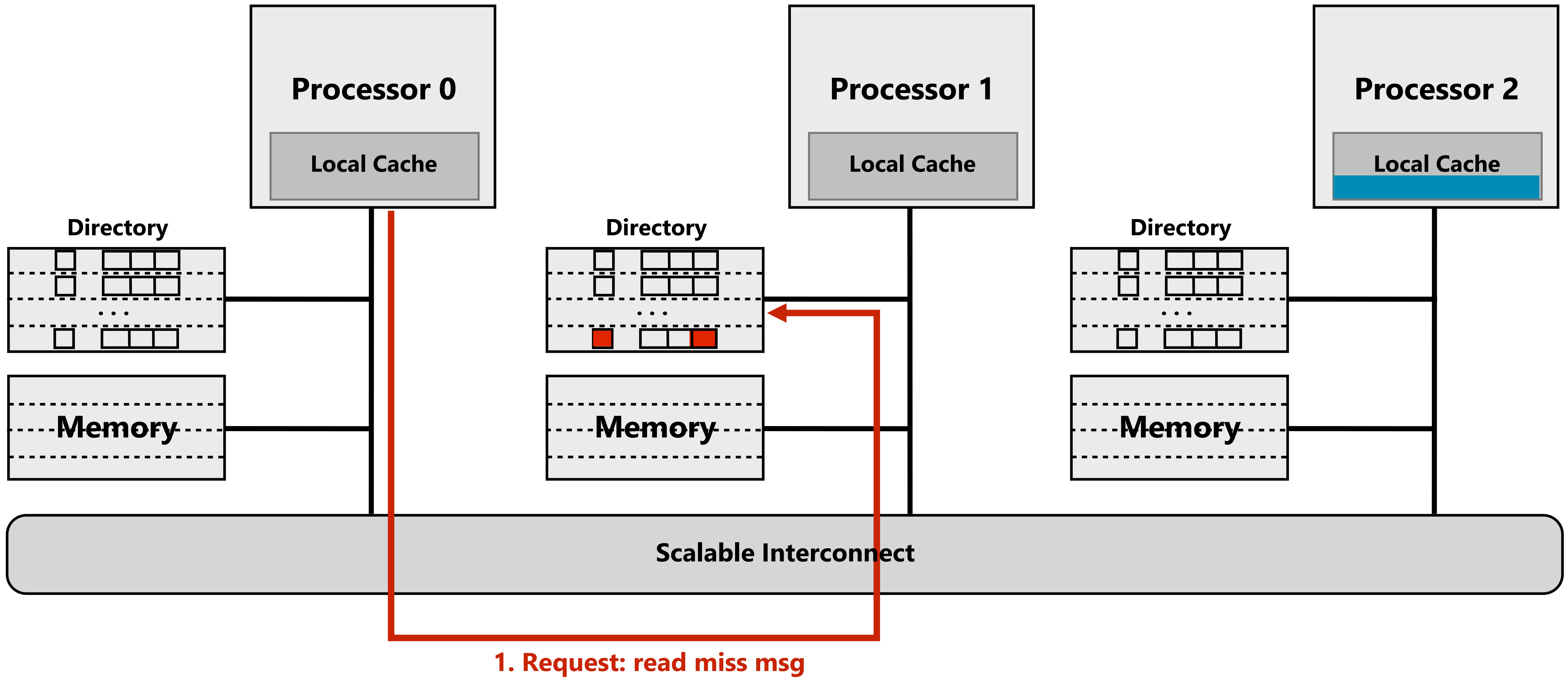
Five network transactions in total

Four of the transactions are on the "critical path" (transactions 4 and 5 can be done in parallel)

- **Critical path: sequence of dependent operations that must occur to complete operation**

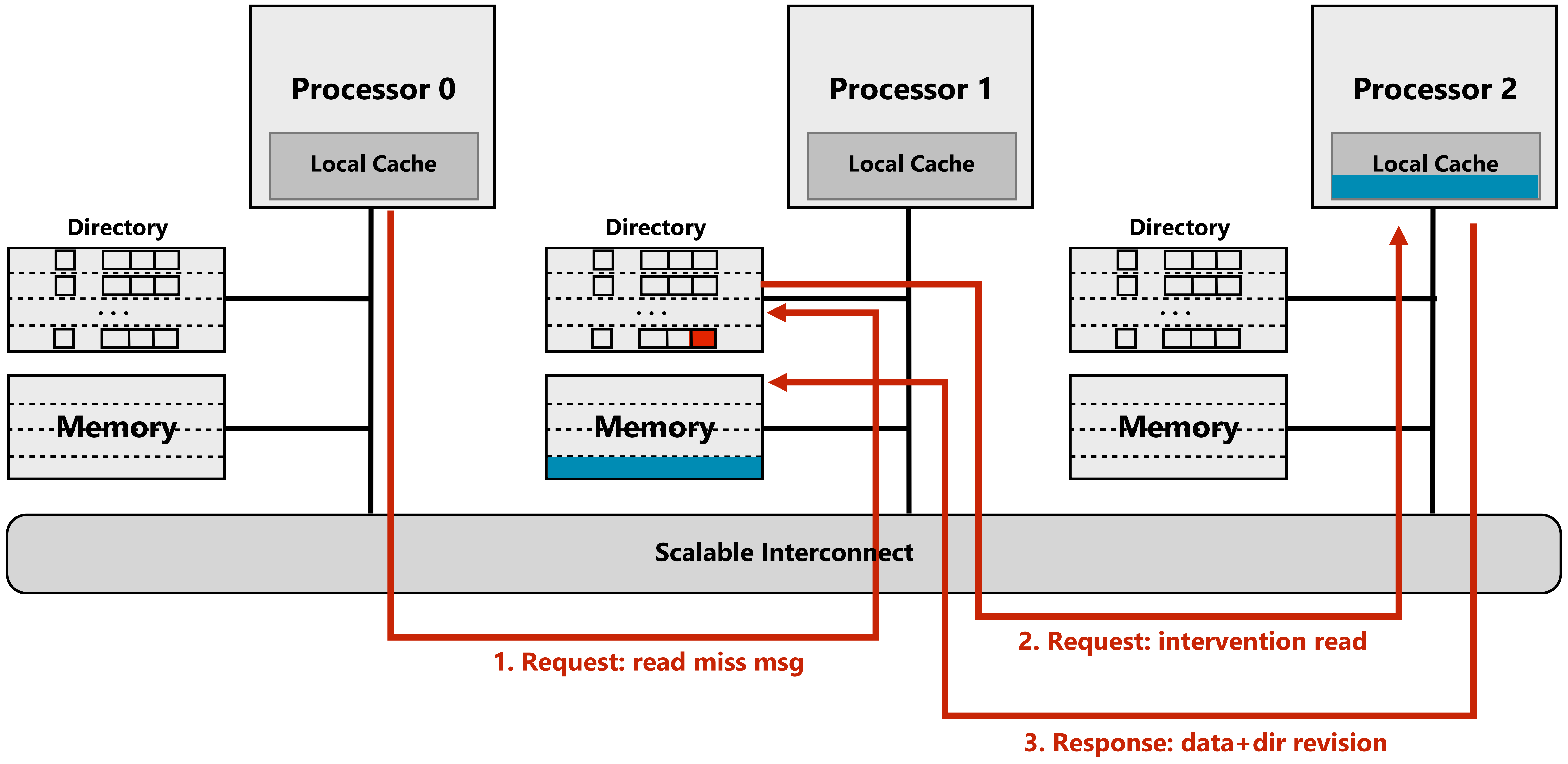
Intervention forwarding

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)



Intervention forwarding

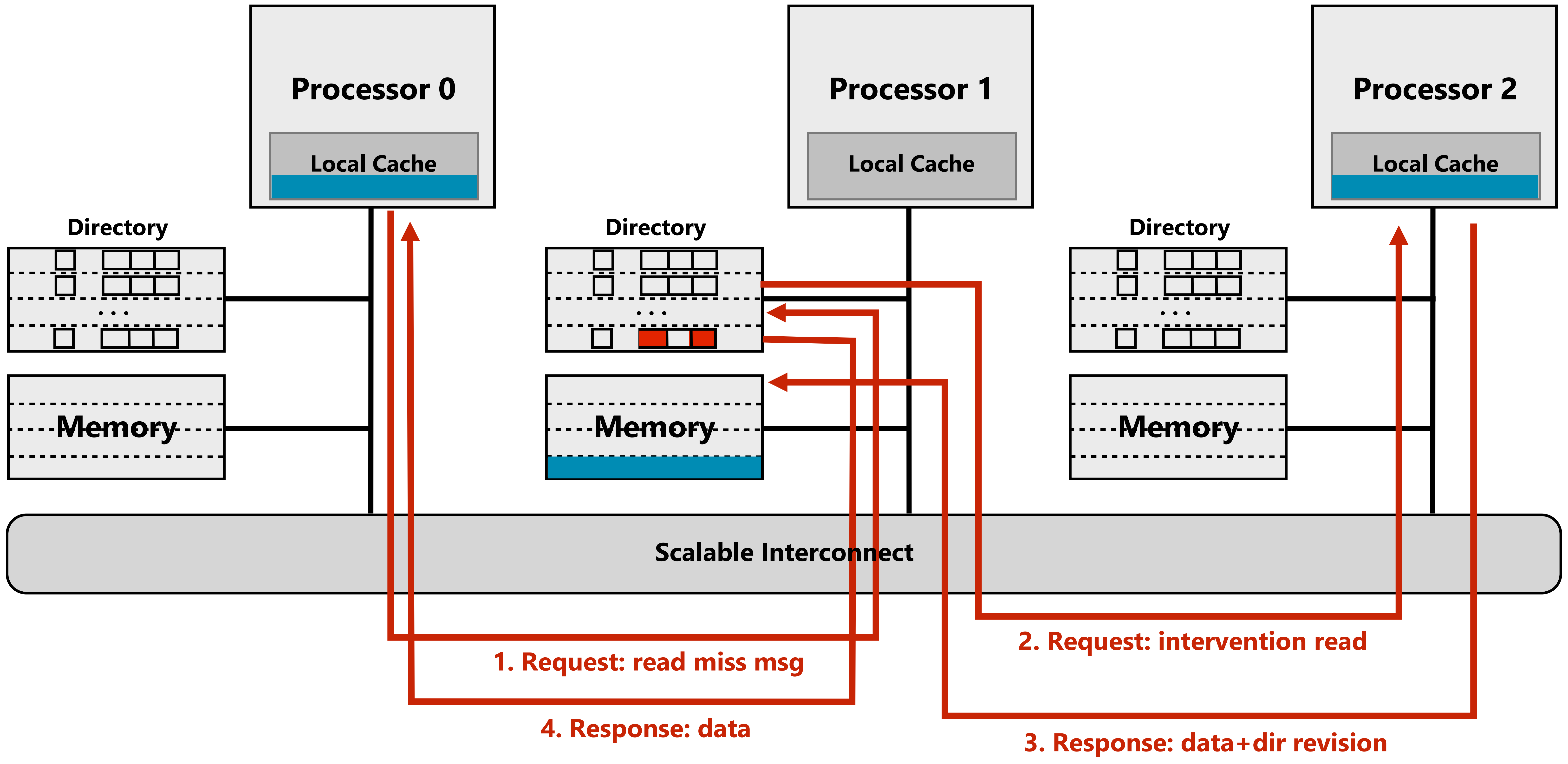
Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)



2. Home node requests data from owner node (processor 2)
3. Owning node responds

Intervention forwarding

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)

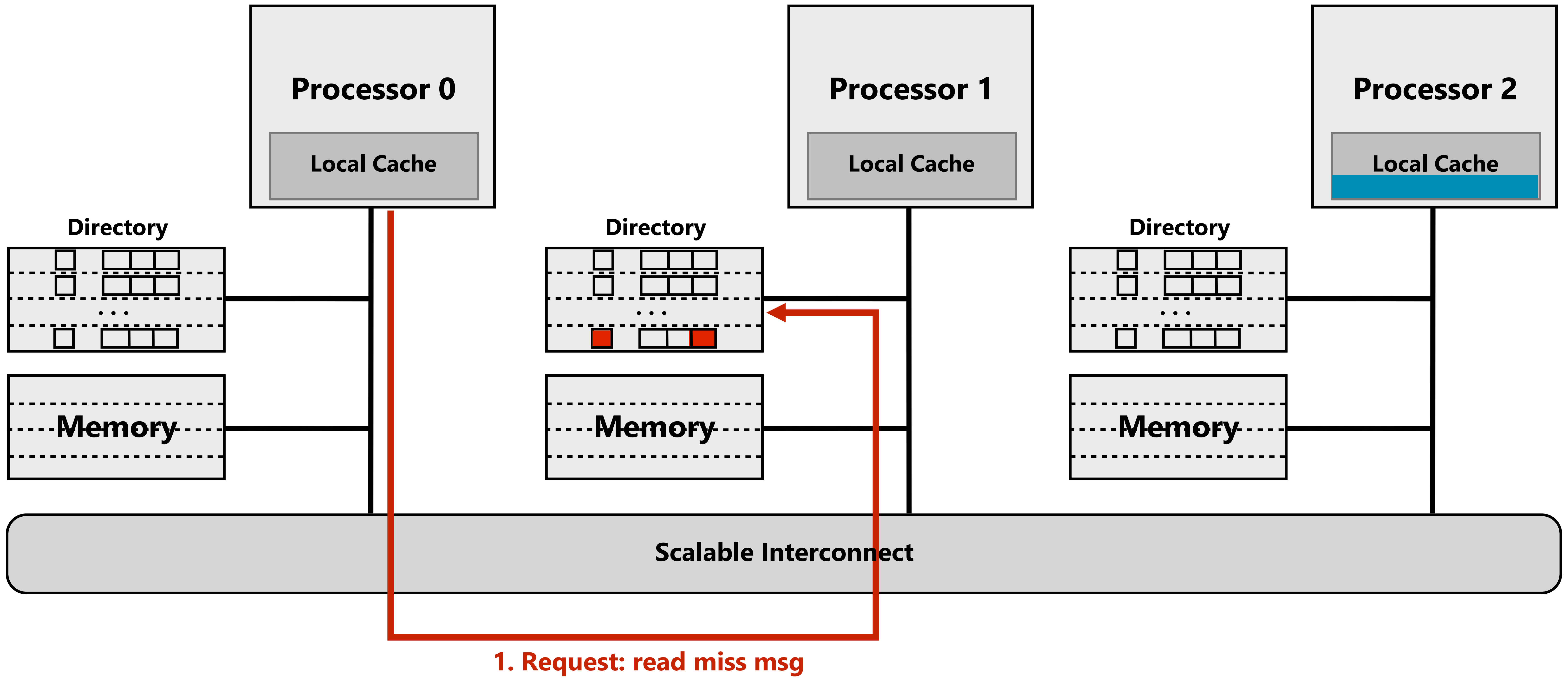


4. **Home node updates directory, and responds to requesting node with data**

Four network transactions in total (less traffic)
But all four of the transactions are on the "critical path."   **Can we do better?**

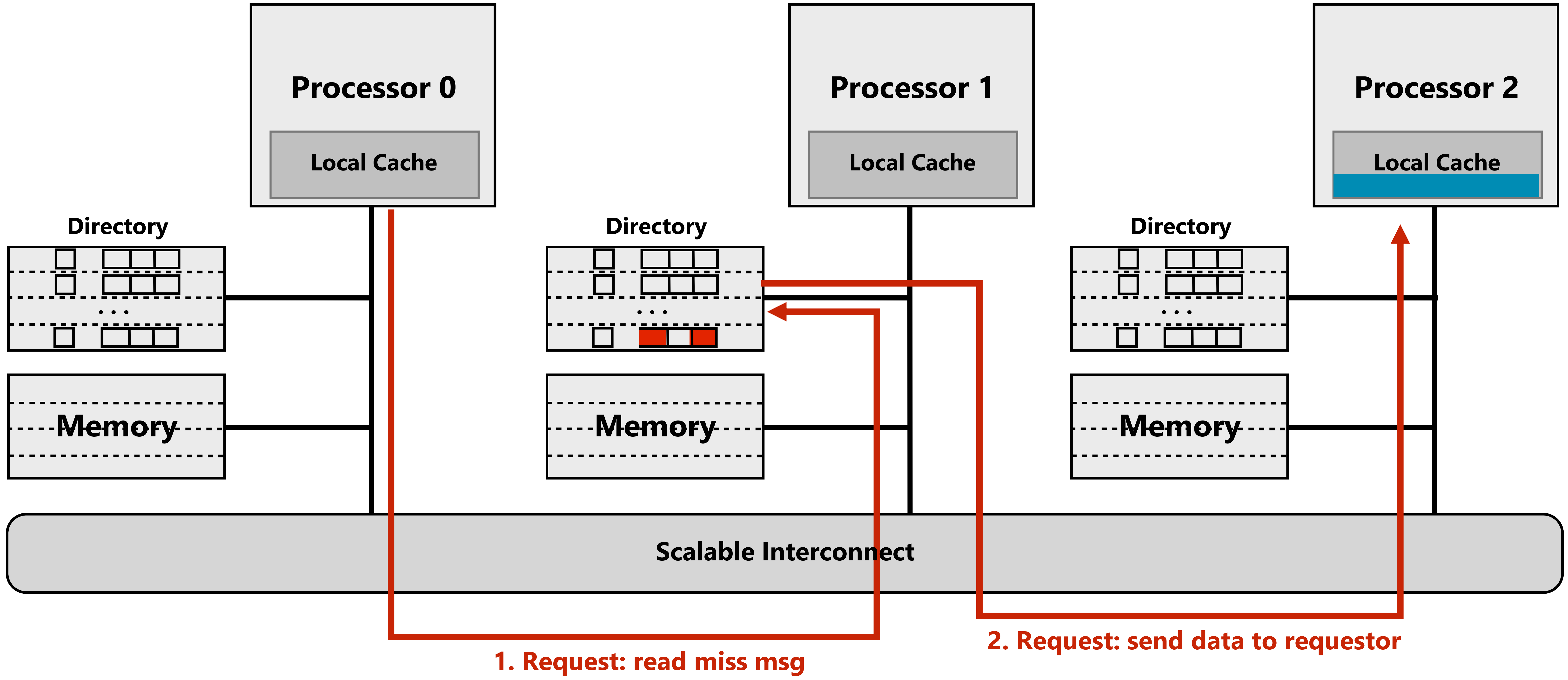
Request forwarding

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)



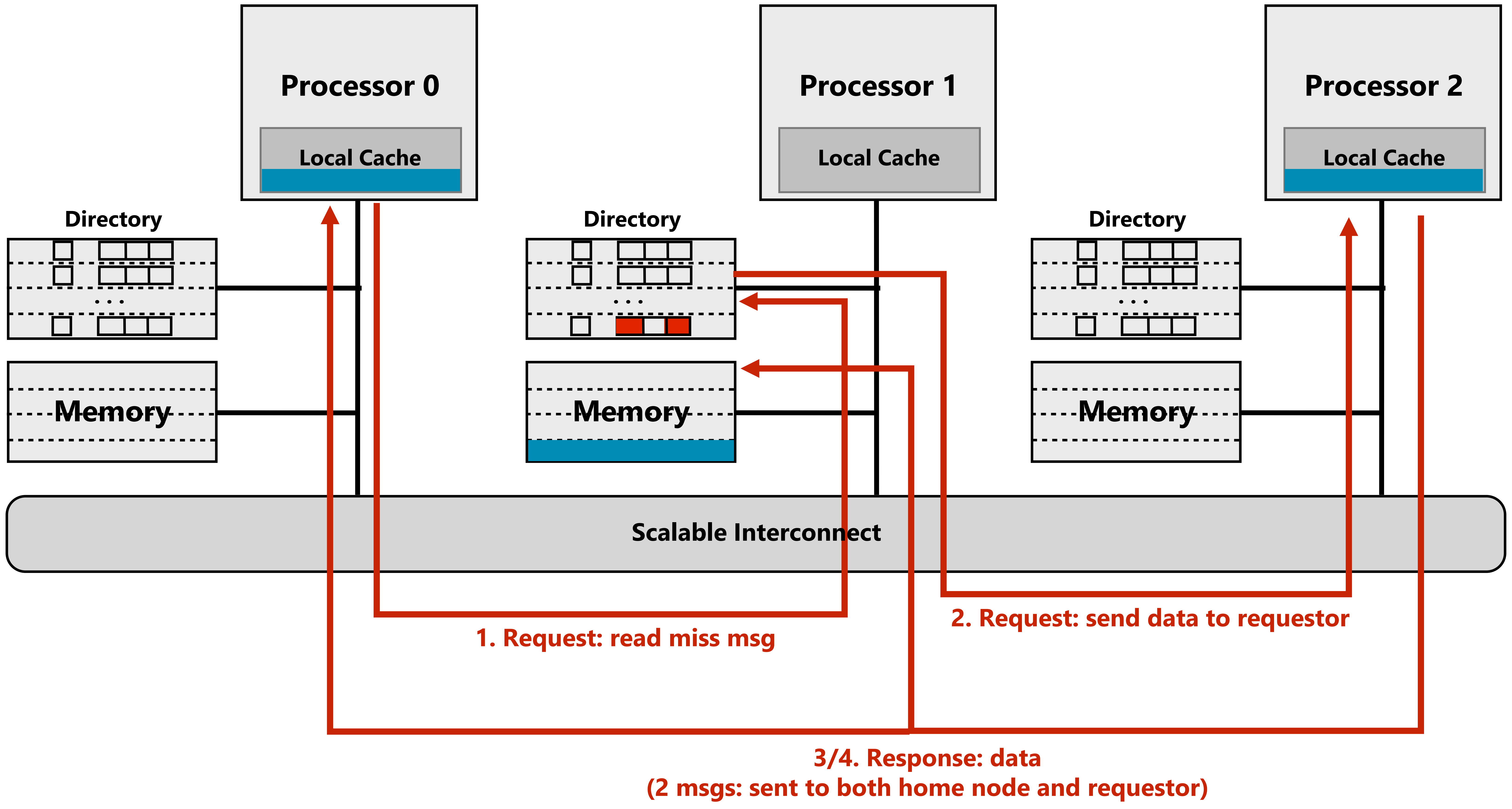
Request forwarding

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)



Request forwarding

Read from main memory by processor 0 of the blue line: line is dirty (contained in P2's cache)

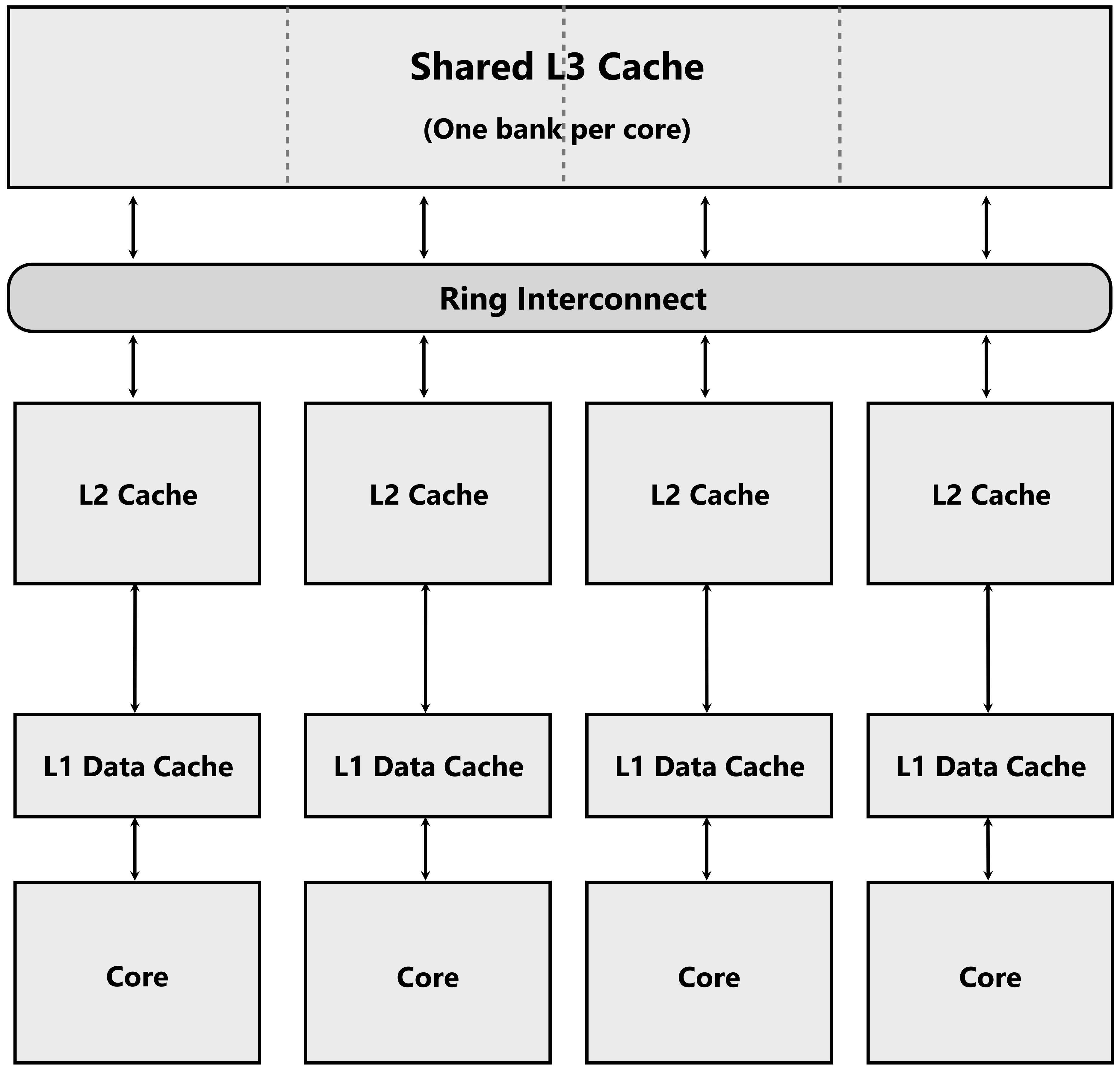


Four network transactions in total

Only three of the transactions are on the critical path (transactions 3 and 4 can be done in parallel)

Note: system is no longer pure request/response (since P0 sent request to P1, but receives response from P2)

Directory coherence in Intel Core i7 CPU

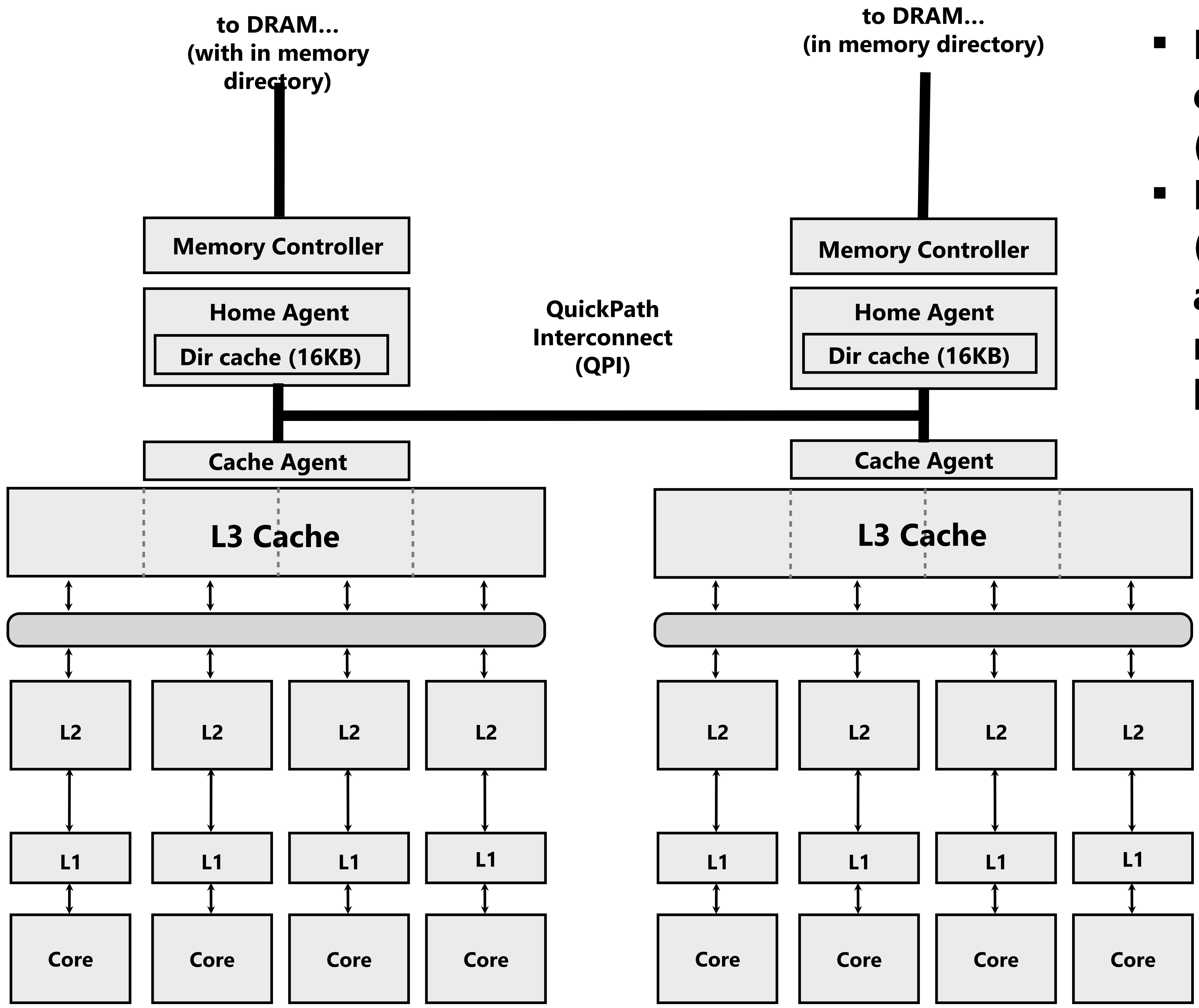


- **L3 hosts in-cache directory (and is inclusive)**
- **Directory maintains list of L2 caches containing line**
- **Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line**

(Core i7 interconnect is a ring, it is not a bus)

- **Directory dimensions:**
  - **P=4**
  - **M = number of L3 cache lines**

Coherence in multi-socket Intel systems



- L3 directory reduces on-chip coherence traffic (previous slide)
- In-memory directory (cached by home agent/memory controller) reduces coherence traffic between cores

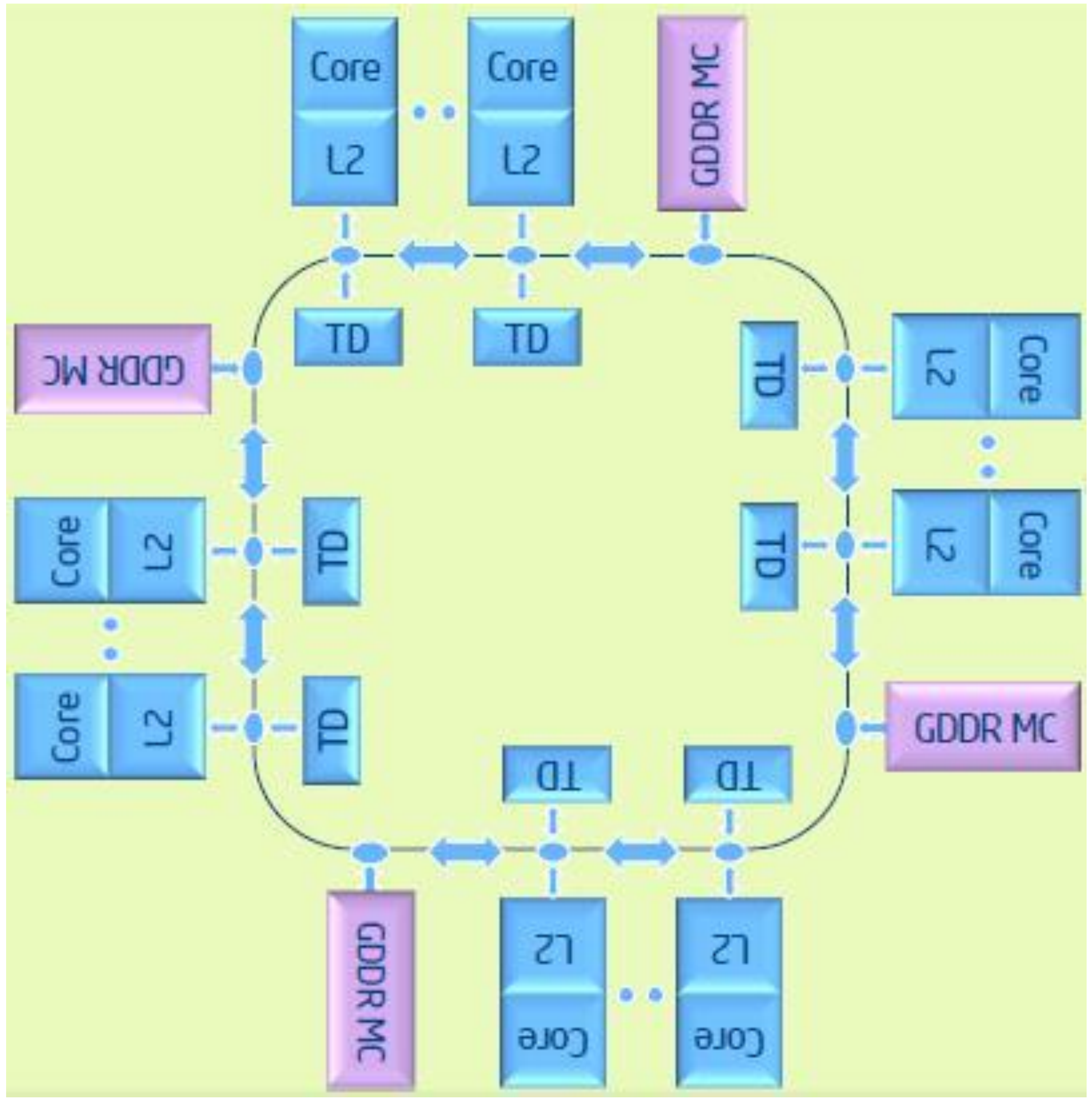
Xeon Phi

Prototype / internal names include:
Larrabee, Knight's Ferry, Knight's Corner, Knight's Landing, Knight's Hill

China's Tianhe-2 has 48,000 Knight's Corner chips

- Intel's NUMA on a chip
- Many (50+) x86 cores
  - Ours have 61
  - "Knight's Corner"
  - 4-way hyper threading
  - Each with 1–2 vector units
- Cache-coherent memory system
- Knight's Corner overall system:
  - Max. 8GB memory
  - Max. 2 TFLOPS
  - 0.004 bytes/flop
    - not balanced
  - 300 Watts

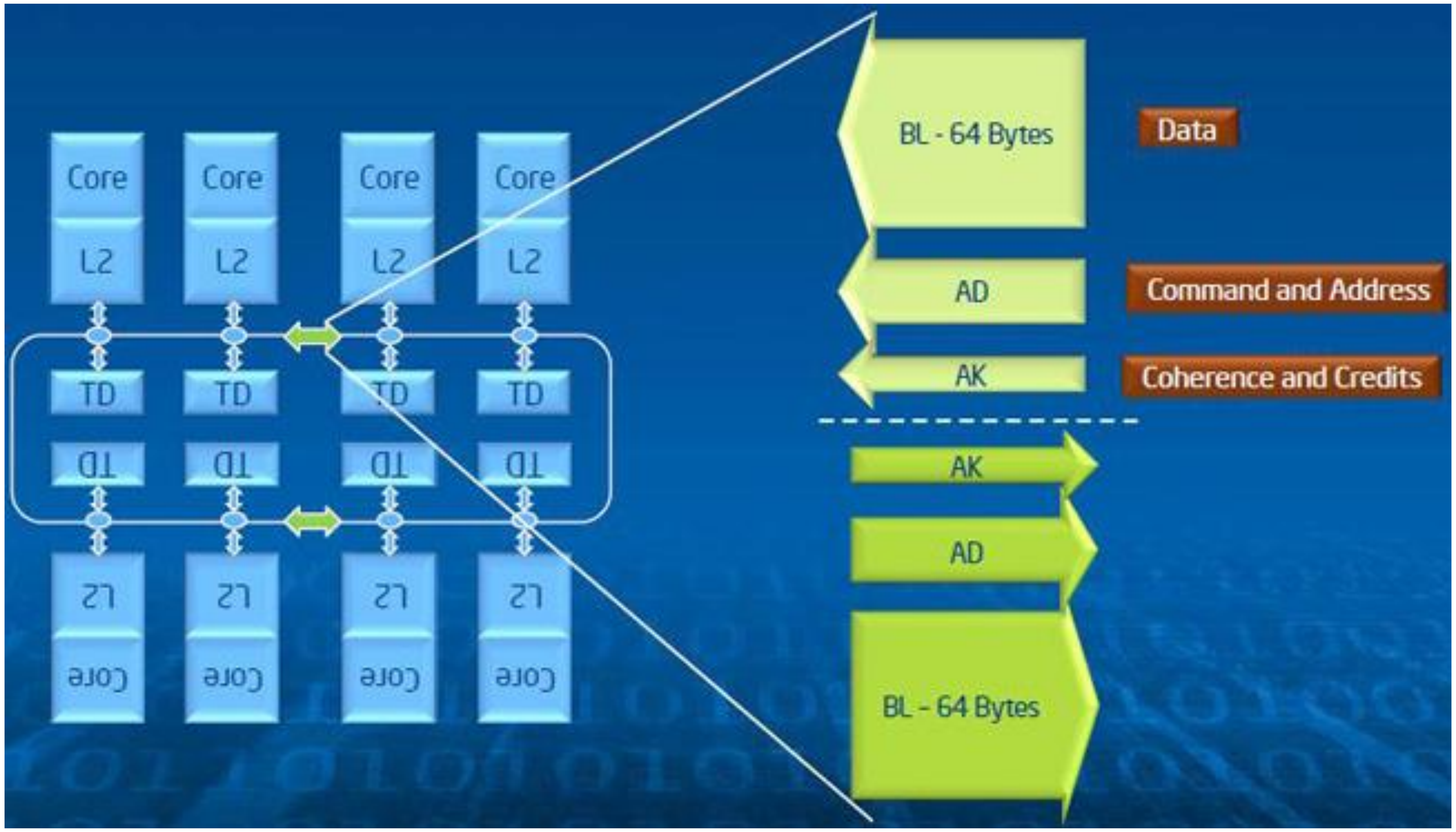
Knight's Corner Xeon Phi Cache Coherence



- **512KB L2 caches**
- **8 memory controllers**
  - **Total 8GB max.**

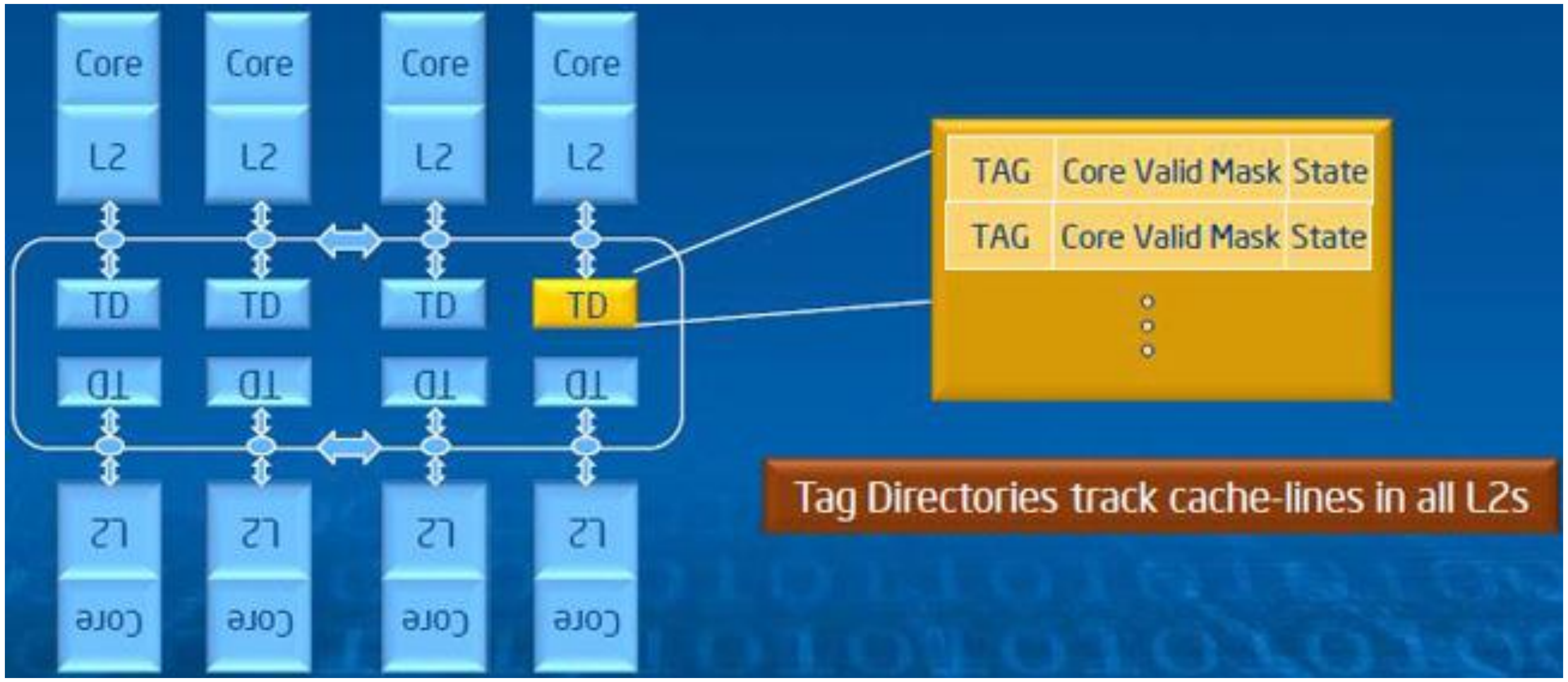
Prototype / internal names include: Larrabee, Knight's Ferry, Knight's Corner, Knight's Landing, Knight's Hill

KC Xeon Phi Ring Communication



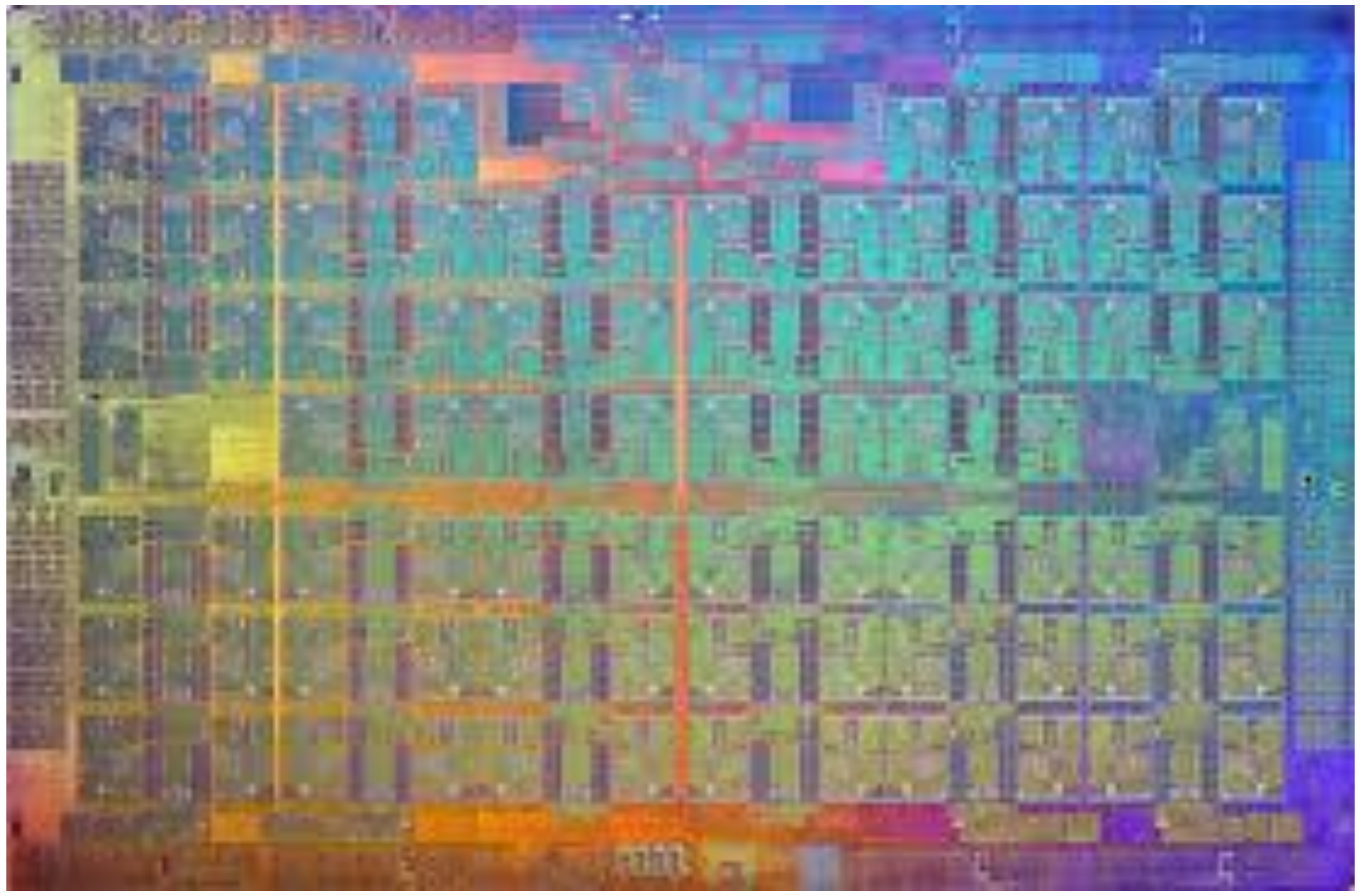
- Messages sent around bidirectional ring
  - Having everything on single chip enables very wide communication paths
  - Can get effect of broadcast by circulating message around entire ring
    - Advantage over point-to-point

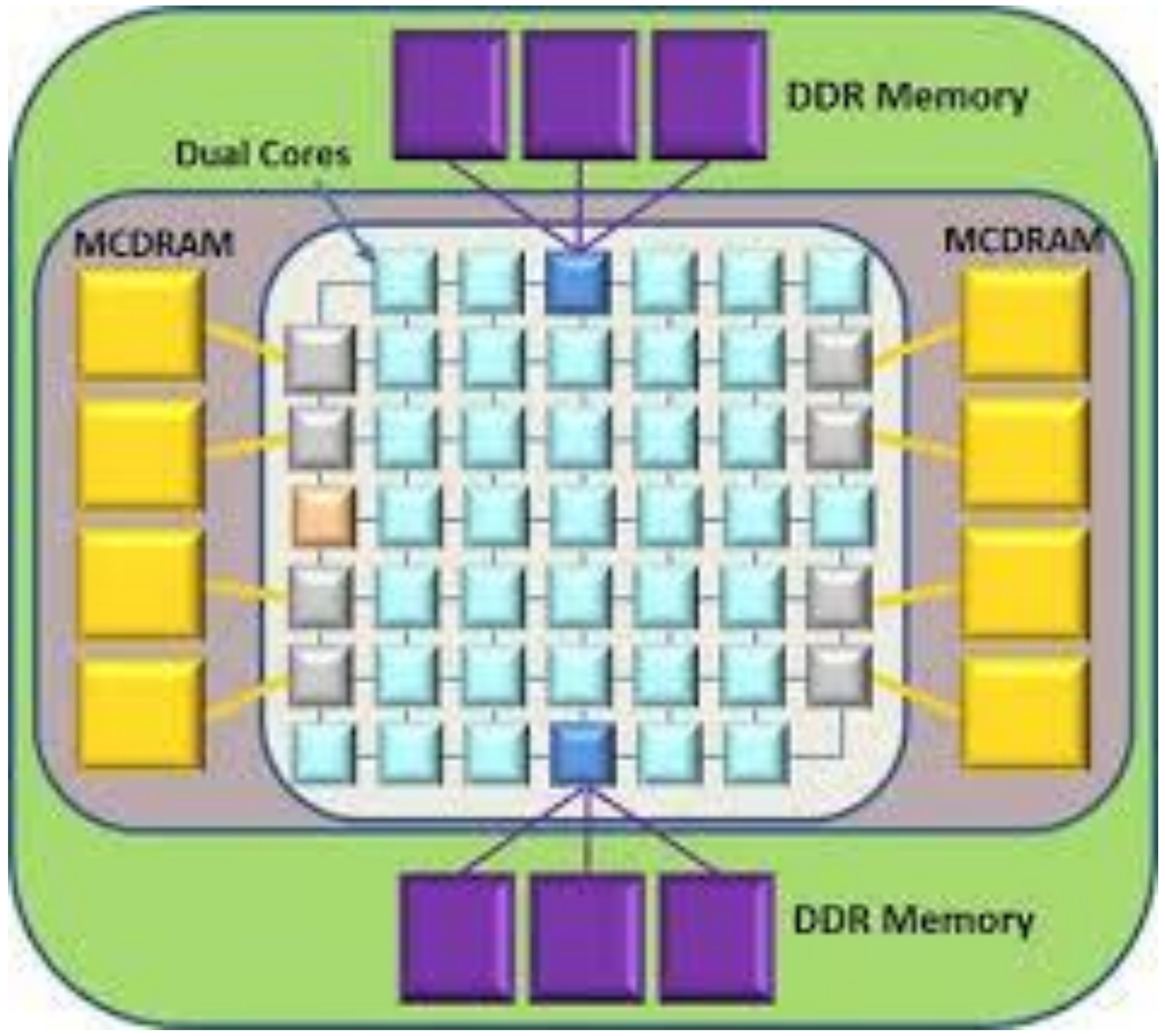
KC Xeon Phi Directory Structure



- Directory keeps track of which lines are resident in local L2
  - Same as with single-node system
- Worst-case memory read or write by P:
  1. Check local cache
  2. Circulate request around ring for line in some cache
  3. Send request around ring to memory controller

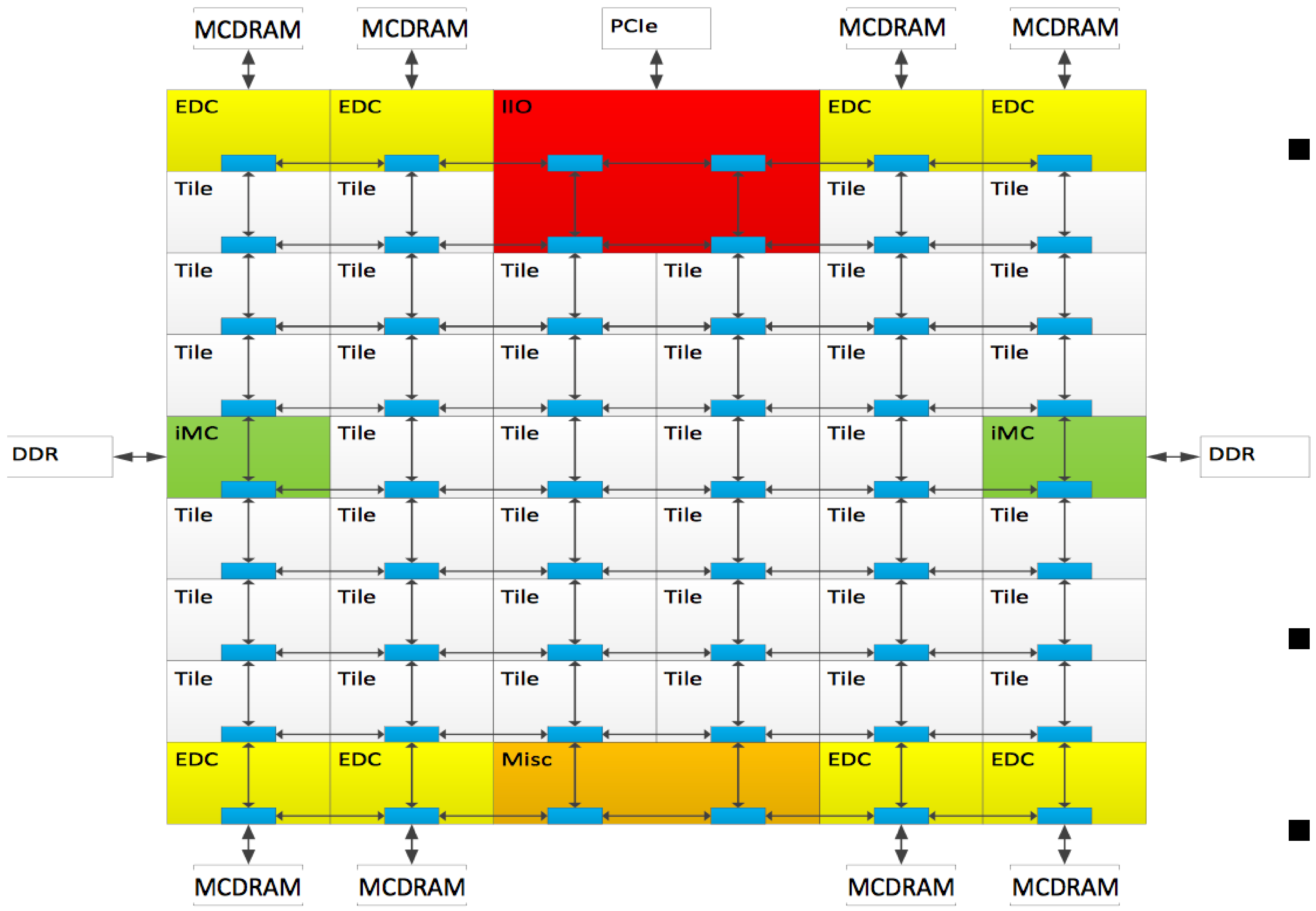
Next Generation Xeon Phi





- "Knight's Landing"
- 72 cores
  - Each with 4-way hyper threading
  - Each with 2 vector units
- Grouped into pairs to give 36 compute nodes
- Peak 6 SP TFLOPS
- 16 GB on package RAM
- Access to up to 384 GB off-package RAM

Knight's Landing Xeon Phi Cache Coherence



- **Nodes organized as 2-D mesh**
  - **Some for computation**
  - **Some for memory interfaces**
- **Use X/Y routing to send messages**
- **Must use more traditional directory-based scheme**

Summary: directory-based coherence

- Primary observation: broadcast doesn’t scale, but luckily we don’t need to broadcast to ensure coherence because often the number of caches containing a copy of a line is small
- Instead of snooping, just store the list of sharers in a “directory” and check the list as necessary
- One challenge: reducing overhead of directory storage
  - Use hierarchies of processors or larger line sizes
  - Limited pointer schemes: exploit fact the most processors not sharing line
  - Sparse directory schemes: exploit fact that most lines are not in cache
- Another challenge: reducing the number of messages sent (traffic) and critical path (latency) of message chains needed to implement coherence operations
- Ring-based schemes can be much simpler than point-to-point communication