Multithreaded Architectures
Lecture 1 of 4

Supercomputing '93 Tutorial

Friday, November 19, 1993

Portland, Oregon

Rishiyur S. Nikhil

Digital Equipment Corporation

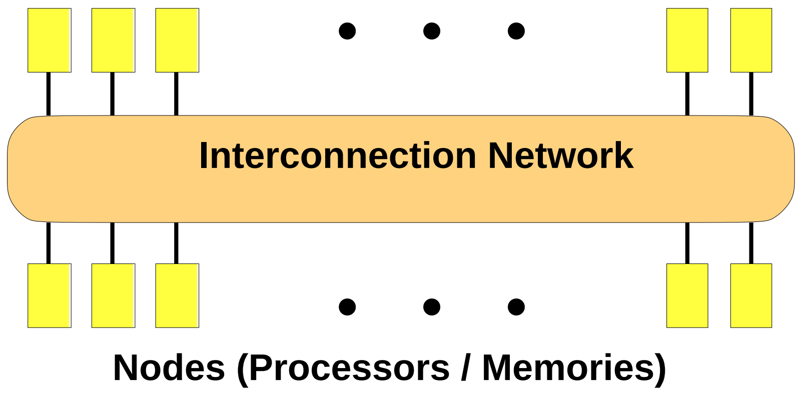
Cambridge Research Laboratory

Overview of Lecture 1

Basic Issues, ''Traditional'' Solutions

- **Asynchrony in Massively Parallel Architectures (MPAs)**
- **Common framework for studying asynchrony:**
  - **The "Remote Loads" problem**
  - **The "Synchronizing Loads" problem**
- **Basic ideas behind multithreading**
- **Directory−based cacheing (Partial solution for Remote Loads)**
  - **Stanford DASH**
  - **Kendall Square Research KSR−1**
  - **MIT Alewife**
- **Directory−based cacheing and Synchronizing Loads**

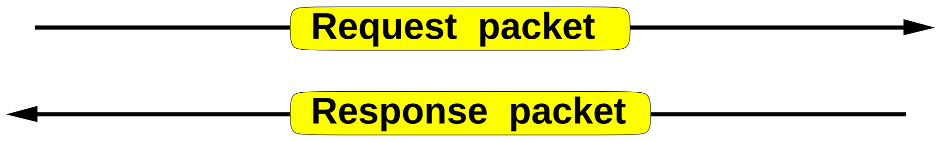
Massively Parallel Architectures



Interconnect: typically packet−switched

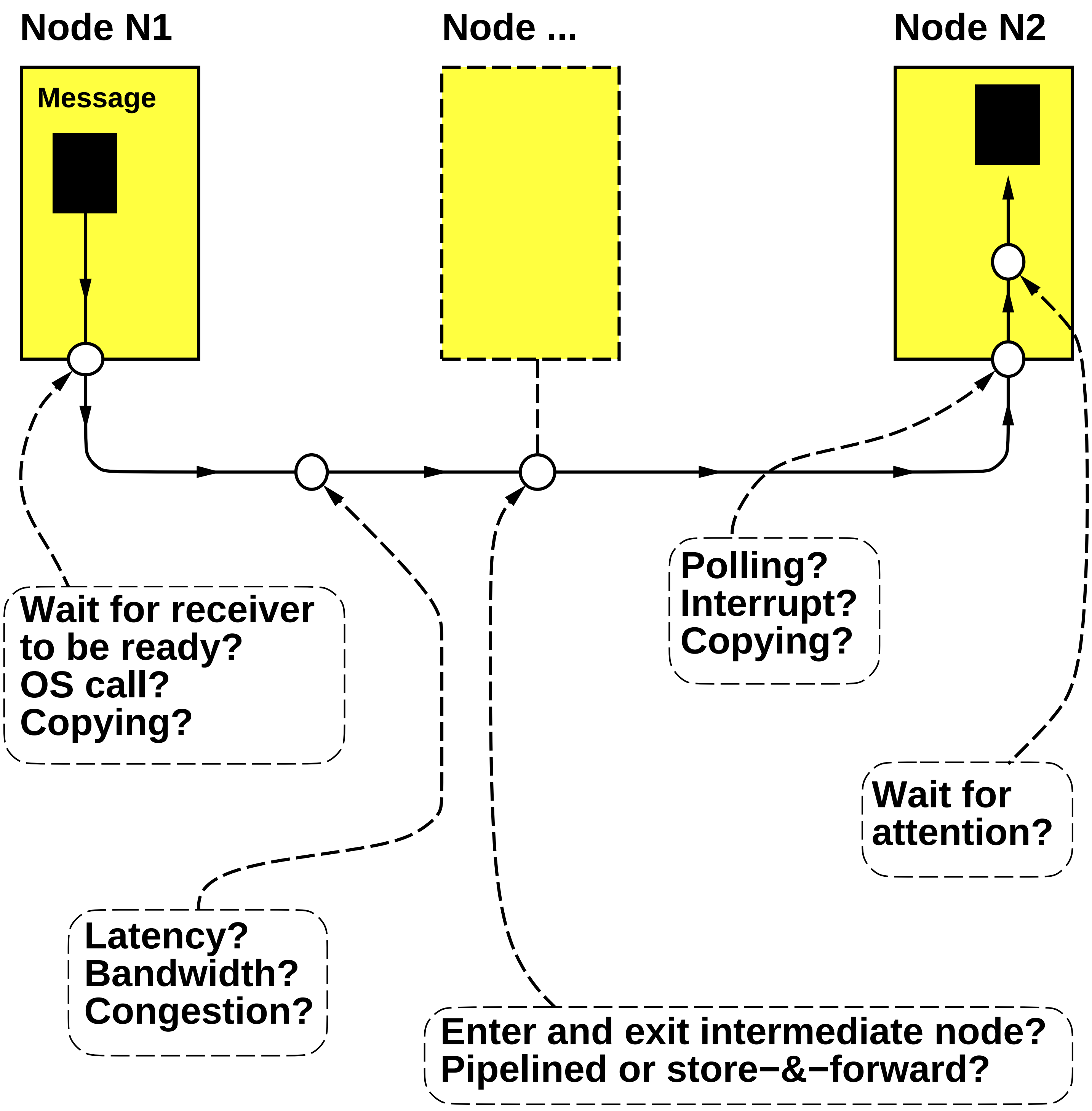
(grid, butterfly, hypercube, fat tree, ...)

Remote accesses: "split transactions"



Latency across network is typically much greater than node cycle time

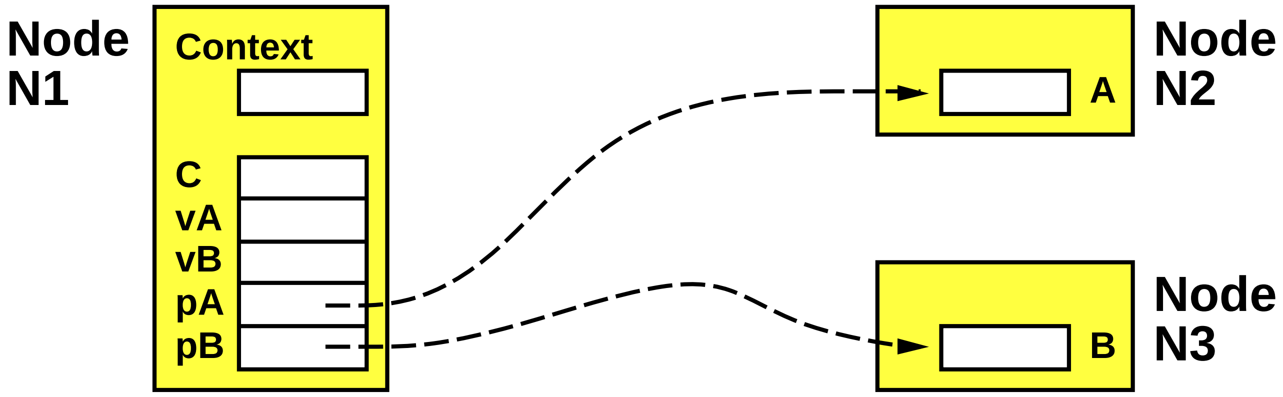
Communication Overheads


Node N1
Node ...
Node N2
Message
Wait for receiver to be ready? OS call? Copying?
Latency? Bandwidth? Congestion?
Enter and exit intermediate node? Pipelined or store−&−forward?
Polling? Interrupt? Copying?
Wait for attention?

nikhil−34

The "Remote Loads" Problem

(A common framework for evaluating communication overheads in massively parallel architectures)



C, A, B: individual memory words / objects / arrays

On Node N1, compute: C = A − B

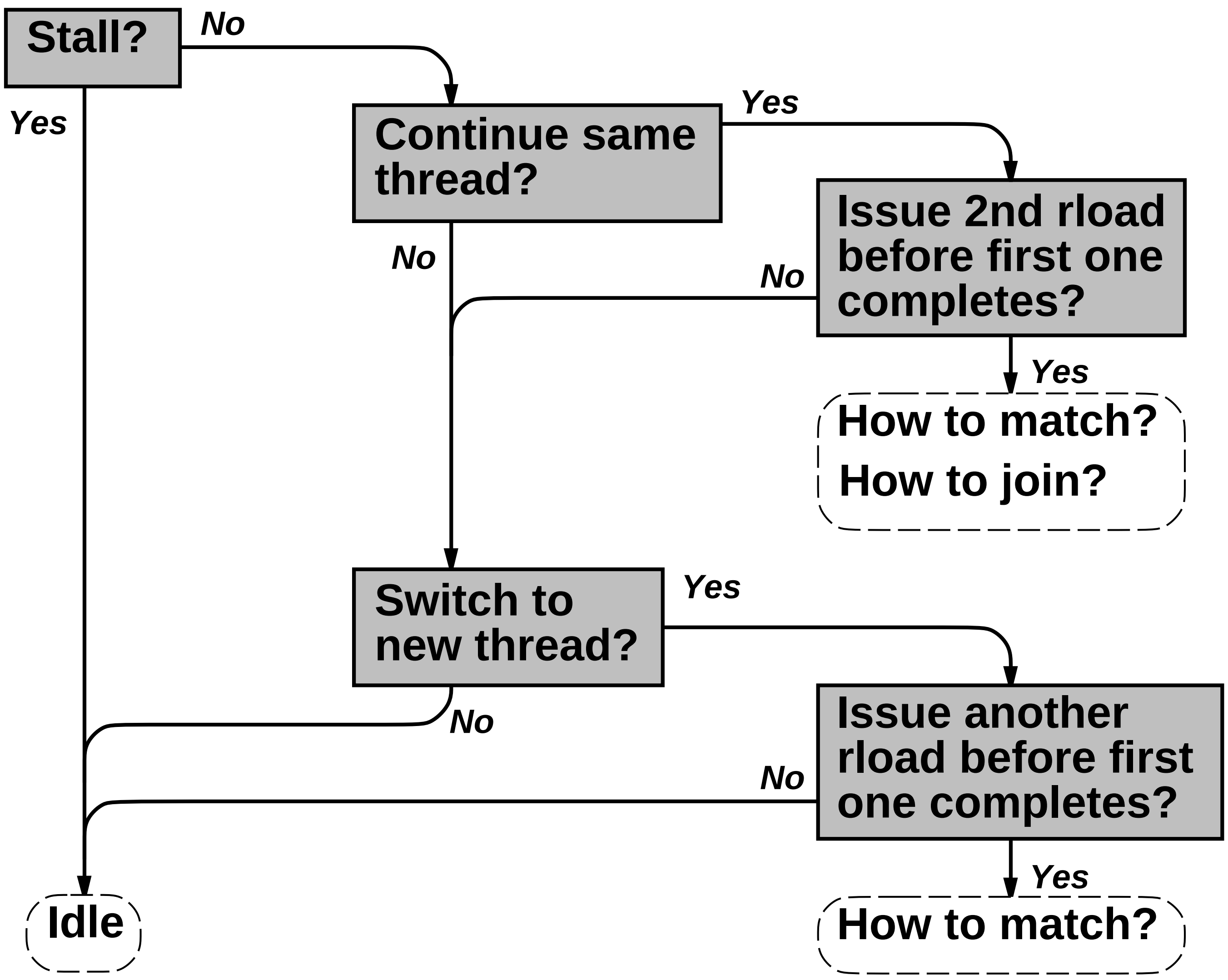
Schematic code:

```
vA  =  rload  pA   } ("remote loads")
vB  =  rload  pB   }
C   =  vA  -  vB
```

"Remote Loads" Issues

After issuing first rload:

What should Node N1 do until response arrives?



Match Problem: ***(unary synchronization)***

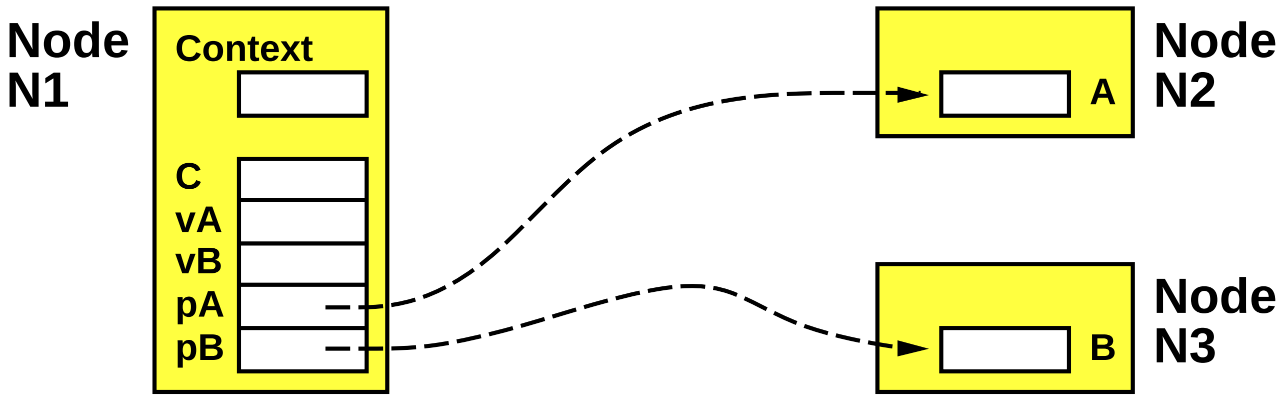
Matching a response with corresponding rload

Join Problem: ***(n−ary synchronization, n > 1)***

Knowing when several pending events are done

The "Synchronizing Loads" Problem

(A common framework for evaluating communication overheads in massively parallel architectures)



C, A, B: individual memory words / objects / arrays

On Node N1, compute: C = A − B

A and B computed concurrently

Thread on N1 must be notified when A, B are ready

Suppose, semaphores associated with A and B
Overheads and asynchrony:

- **Busy−waiting: message traffic, processor cycles?**
- **No busy−waiting: how/where to rendezvous?**

How Communication Overheads Restrict the Programming Model

High communication overheads:

- **Avoid frequent communication, so that total communication overhead does not cause slowdown**
- **Group communications together to amortize fixed overheads ("message vectorization")**

High latencies and synchronization overheads:

- **Carefully orchestrate computation and communication to avoid idle waiting ("Marching Band")**

These steps are feasible only for programs with regular control and data structures

("mostly synchronous" or "structured parallelism")

For *general−purpose programming,* which has more *unstructured parallelism,* we need to be able to tolerate *asynchrony* with more dynamic scheduling

Multithreading: A Solution for "Remote Loads" and possibly for "Synchronizing Loads"

When asynchrony cannot be avoided, tolerate it:

- **When a thread must wait, apply the processor quickly to another thread**

Need:

- **large pool of threads, so physical resources can be kept utilized** ***("Parallel Slackness")***
- **Solutions to Match and Join problems**
- **Solutions to Synchronizing Loads problem**

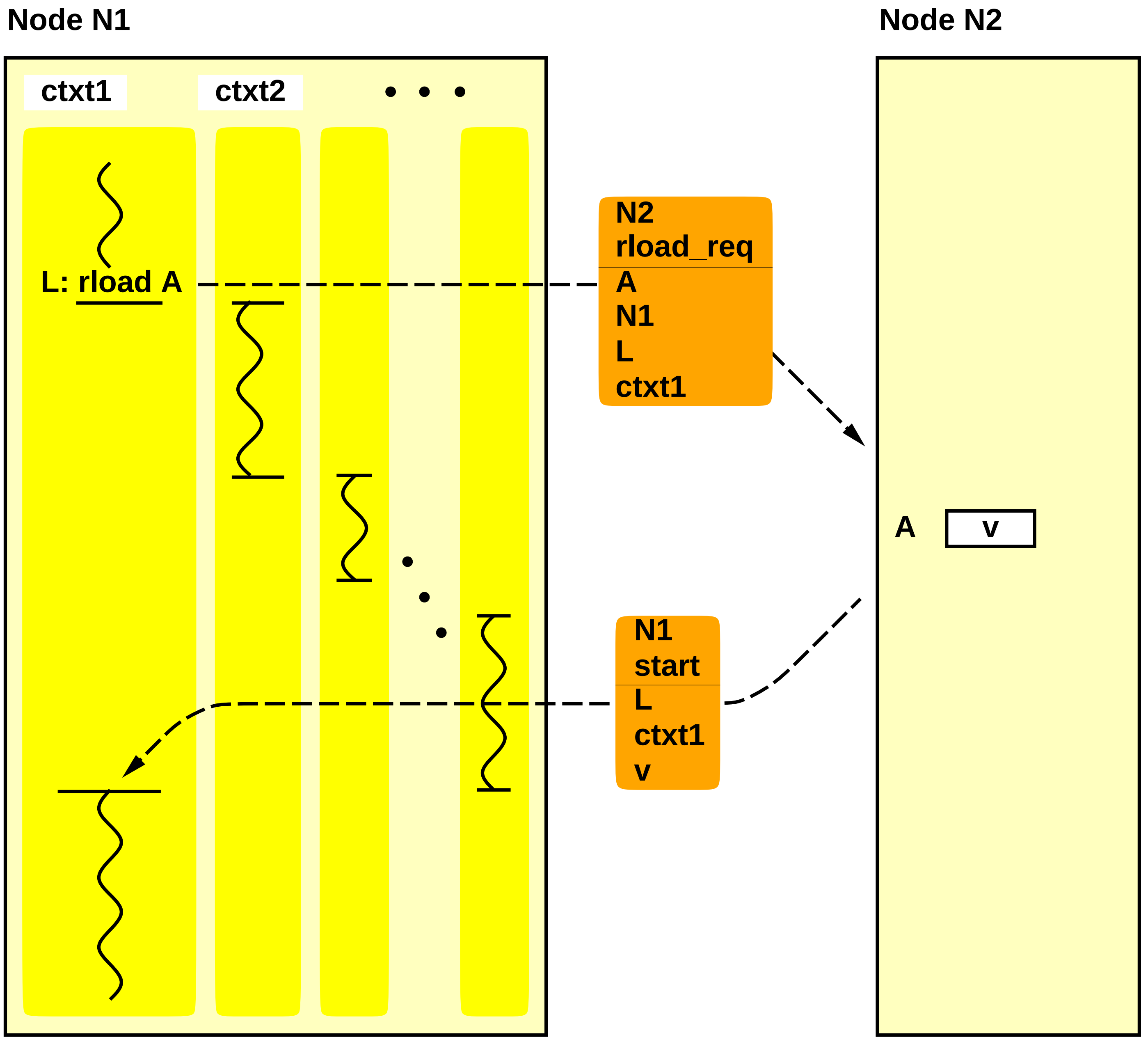
Issues at all levels:

Languages: **How to express so much parallelism?**

Compilers and runtime systems: **How to represent, implement such fine grain multithreading efficiently?**

Architectures: **How to implement such multiplexing efficently?**

Solution: Multithreading



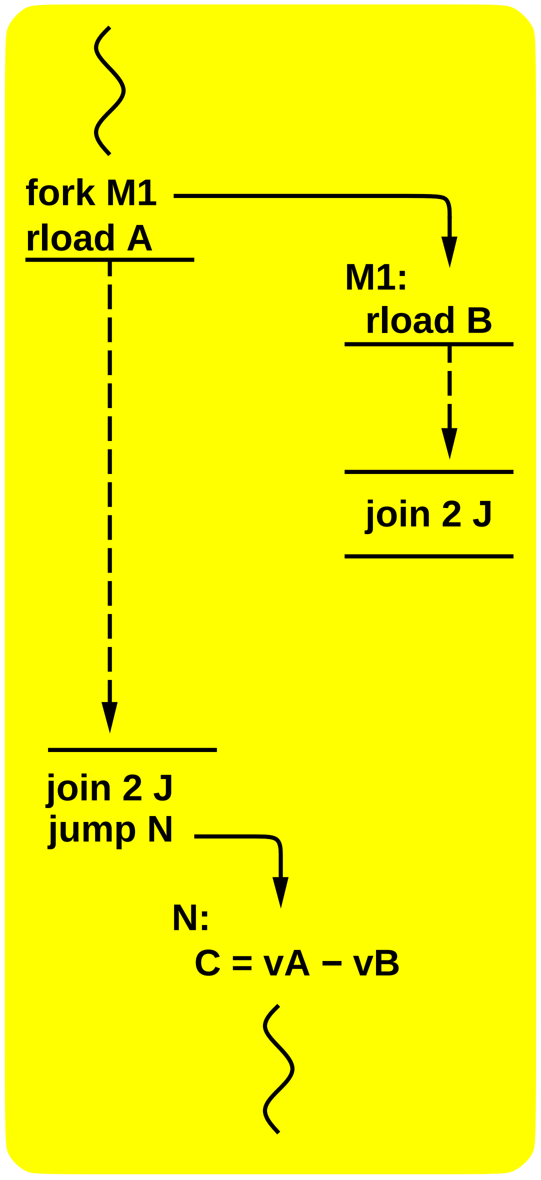
- Messages carry continuations; here: (N1, L, ctxt1)
- Cost of thread−switching should be very low
- More inter−node latency => need more threads

Multiple Remote Loads in Parallel
Fine Grain Multithreading

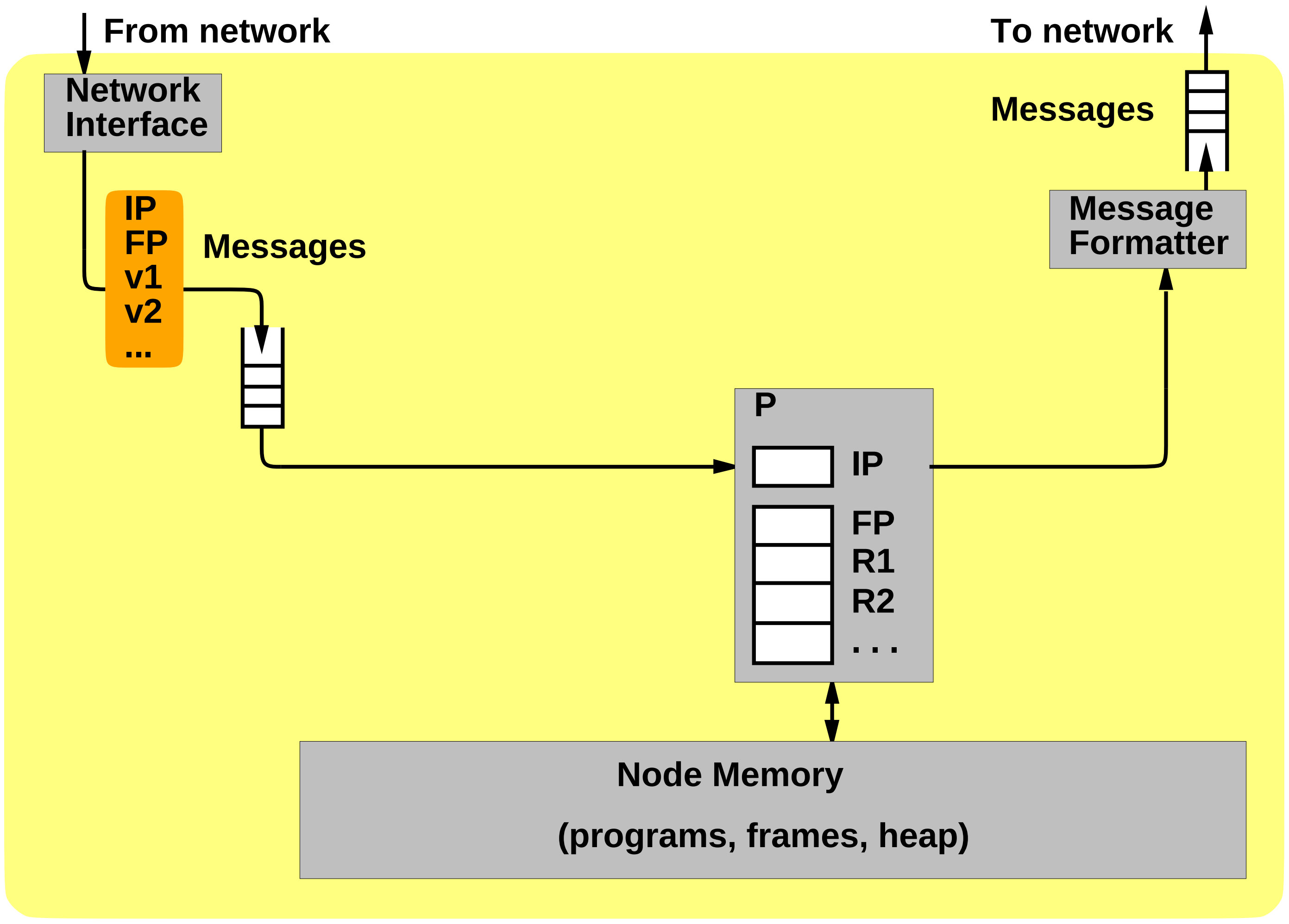
Schematic code:

```
      fork  M1

L1:   vA = rload pA
      join 2 J
      jump  N

M1:   vB = rload pB
      join 2 J
      jump  N

N:    C = vA − vB
```

Sample behavior:



Node Architecture



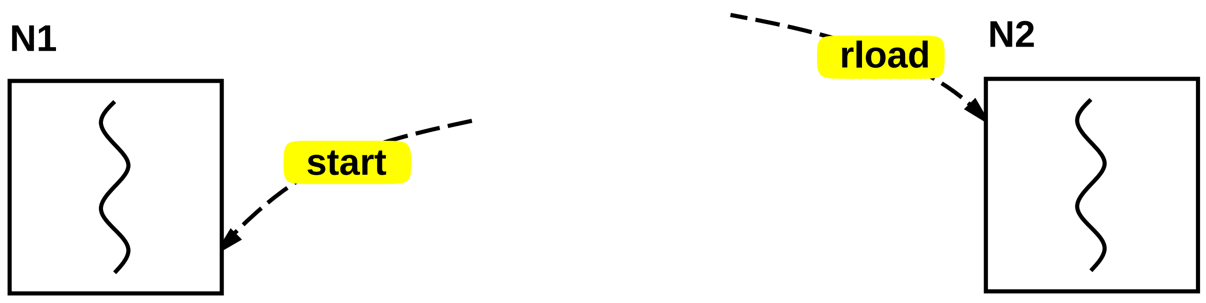
Commercial Machines:

Intel	iPSCs, Paragon, ...
Thinking Machines	CM 5
nCUBE	nCUBE 2
Meiko	CS−2

Research Machines:

MIT	J−Machine
MIT/Motorola	Monsoon, *T
ETL	EM 4, EM 5

Handling Messages



What to do when a message arrives?

(node is busy executing some other thread)

- **Interrupt (conventional):**
  - **Disruptive (fast processors have large state)**
  - **Can't keep up with fast networks**
- **Hardware support for lightweight interrupts:**
  - **Copy of processor state**
  - **Fast vectoring: IP (instruction ptr) on message**
- **Hardware support for enqueuing message (e.g., MIT J−Machine)**
- **Message/synchronization coprocessor**

Distributed Cacheing: A Solution for "Remote Loads"

(but not for "Synchronizing Loads")

Cacheing: classical solution to tolerating memory latency

Problem in MIMDs:

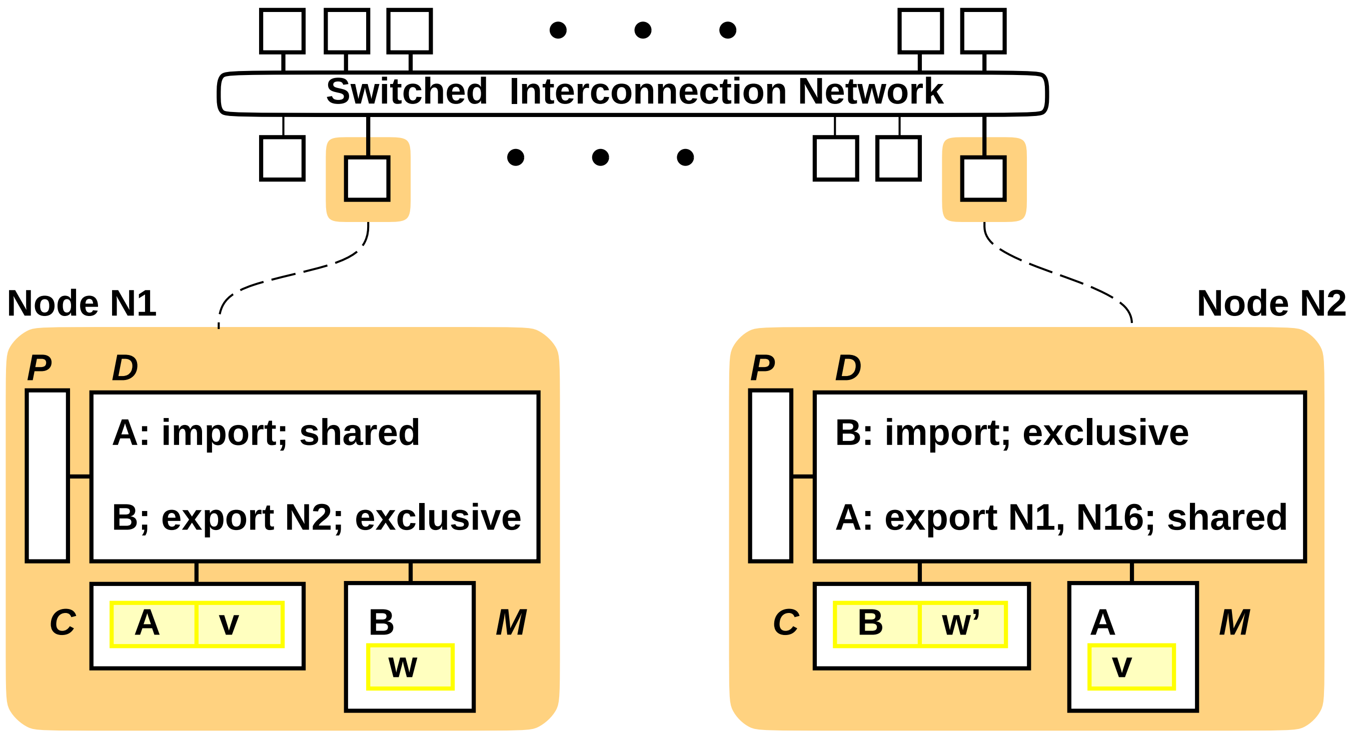
- **Multiple caches must be kept *coherent***
- **Snoopy caches not feasible in MPAs:**
  - **No *broadcast***
  - **cache traffic increases with number of nodes**

Directory−based cacheing

- **Stanford DASH**
- **Kendall Square Research KSR−1**
- **MIT Alewife (also uses multithreading)**

Directory−based Cacheing

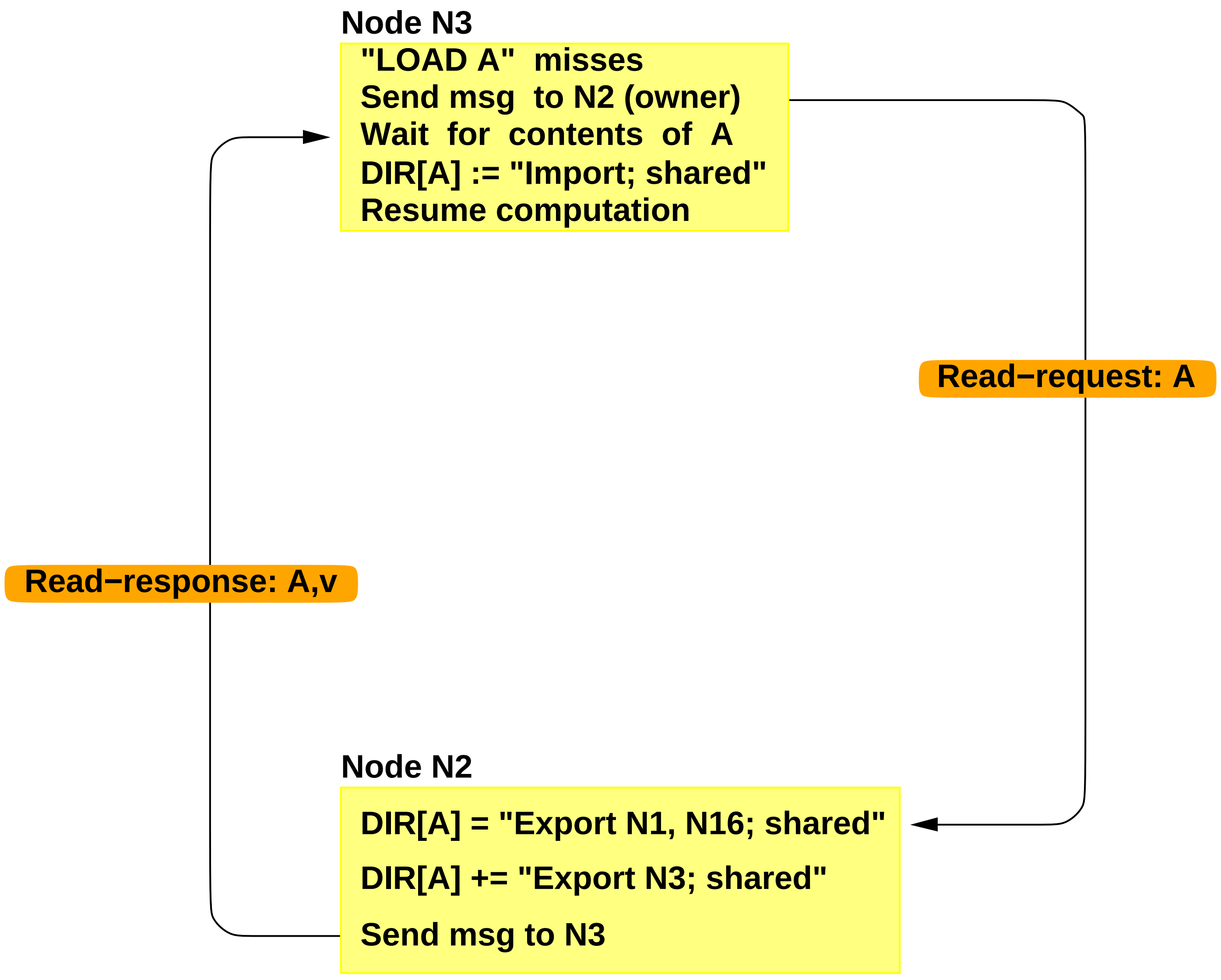
(“Anoopy” caches)



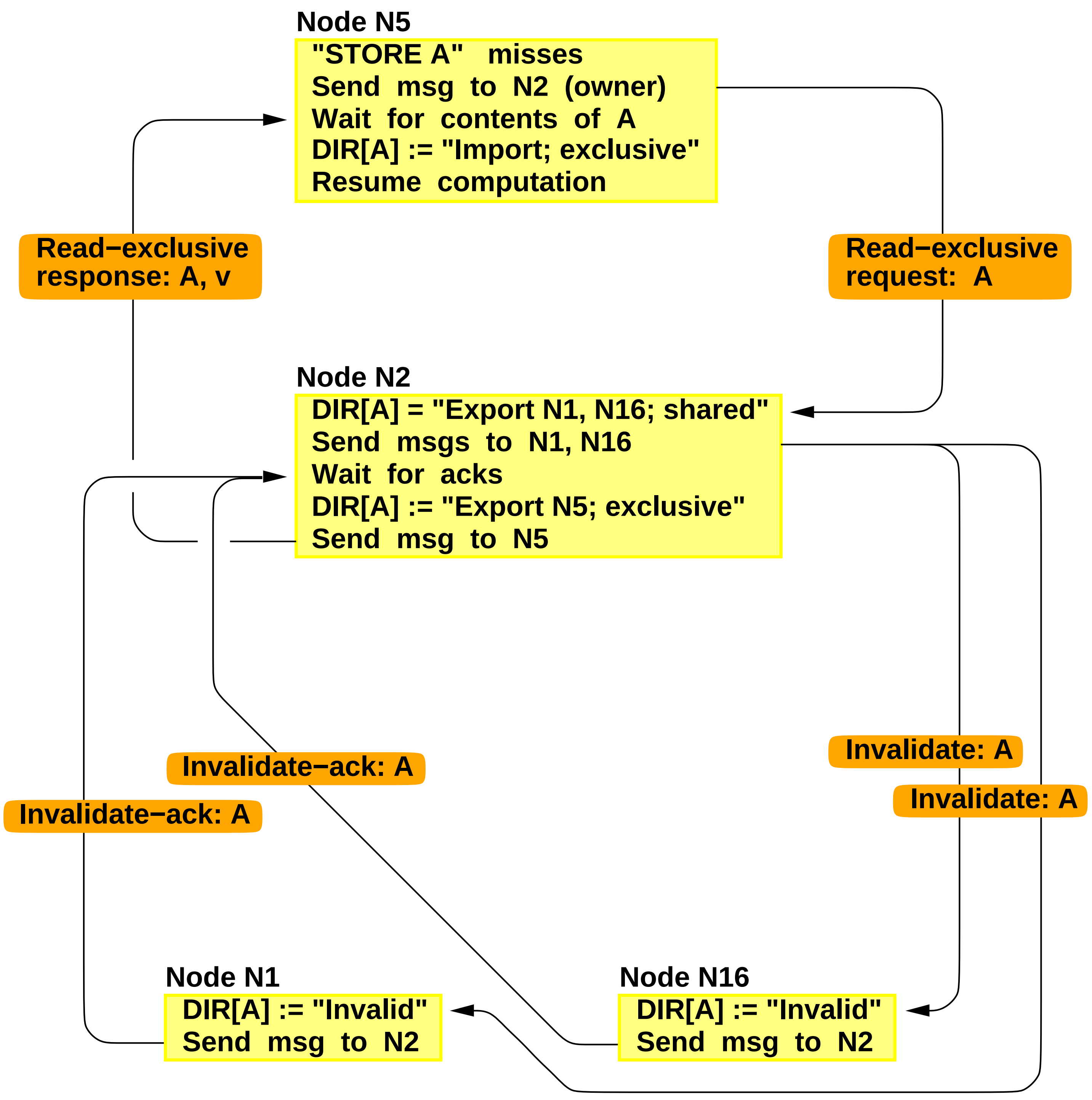
P=Processor; D=Directory; C=Cache; M = Memory

- Every memory location has an "owner" node *(N1 owns B, N2 owns A)*
- Directories contain "import−export" lists, and state:
  - Shared (for reads; many caches may hold copies)
  - Exclusive (for writes; one cache holds current value)
  - . . .
- Protocols for consistency (coherence)

Read Protocol Example



Write Protocol Example



nikhil−40

Directory−based Cacheing: Issues

Size of directories, coherence traffic:

- **For N−node machine, each with M memory locations, each (complete) directory would need O(NM) memory.**

  => total directory size: $O(N^2 M)$

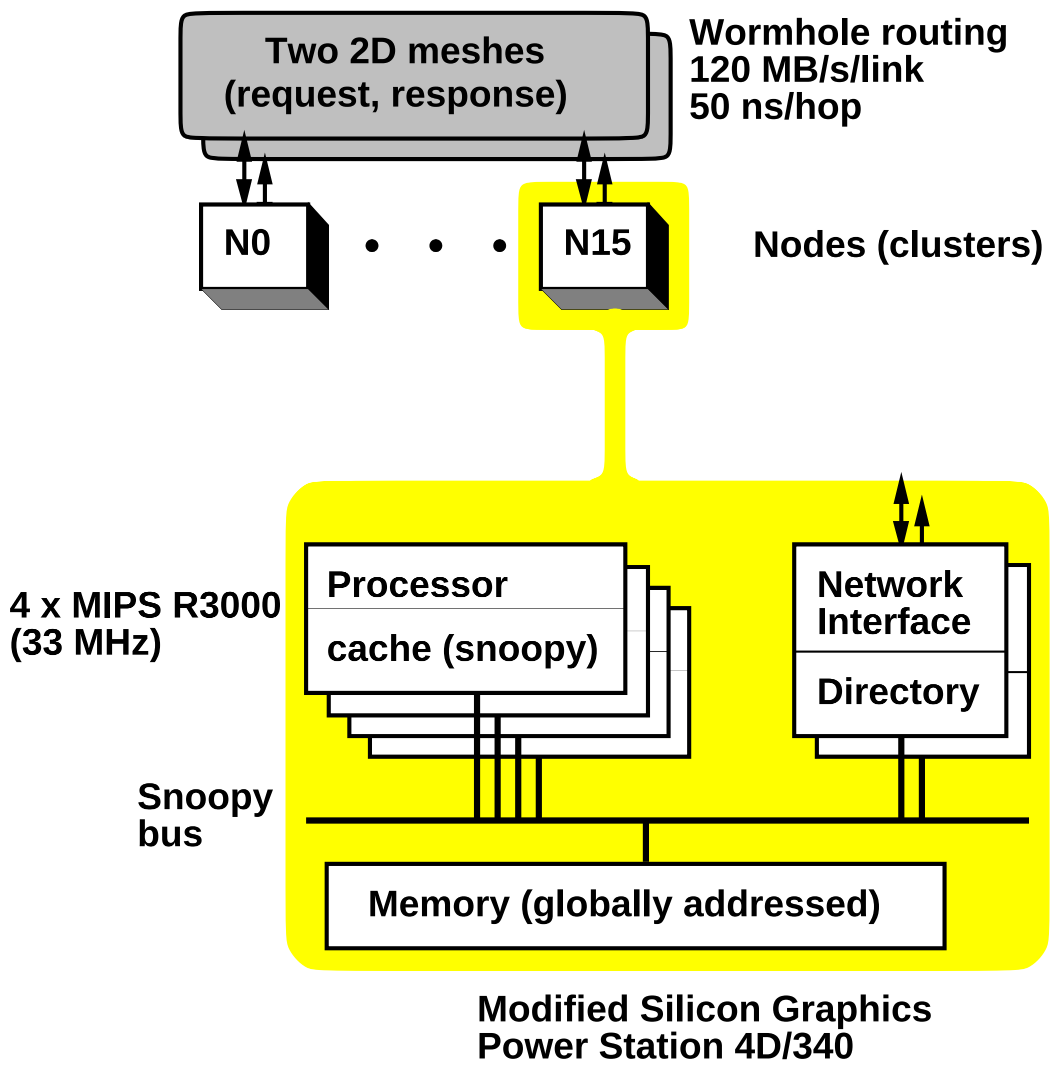
- ***Solutions:***
  - ***Sparse* directories: rely on locality to ensure directories do not grow large**
  - **Cache lines, not individual locations**
    - **But, software problem: *False Sharing***
      Nodes compete for same cache line even if interested in disjoint locations

What to do on a cache miss?

- **"Remote Loads" problem, again.**
  Latency may be worse because of directory system.
- ***Solution:* Various; see following pages**

"Synchronizing Loads" not addressed, per se

Stanford DASH



Timings for a read that misses in the processor cache:

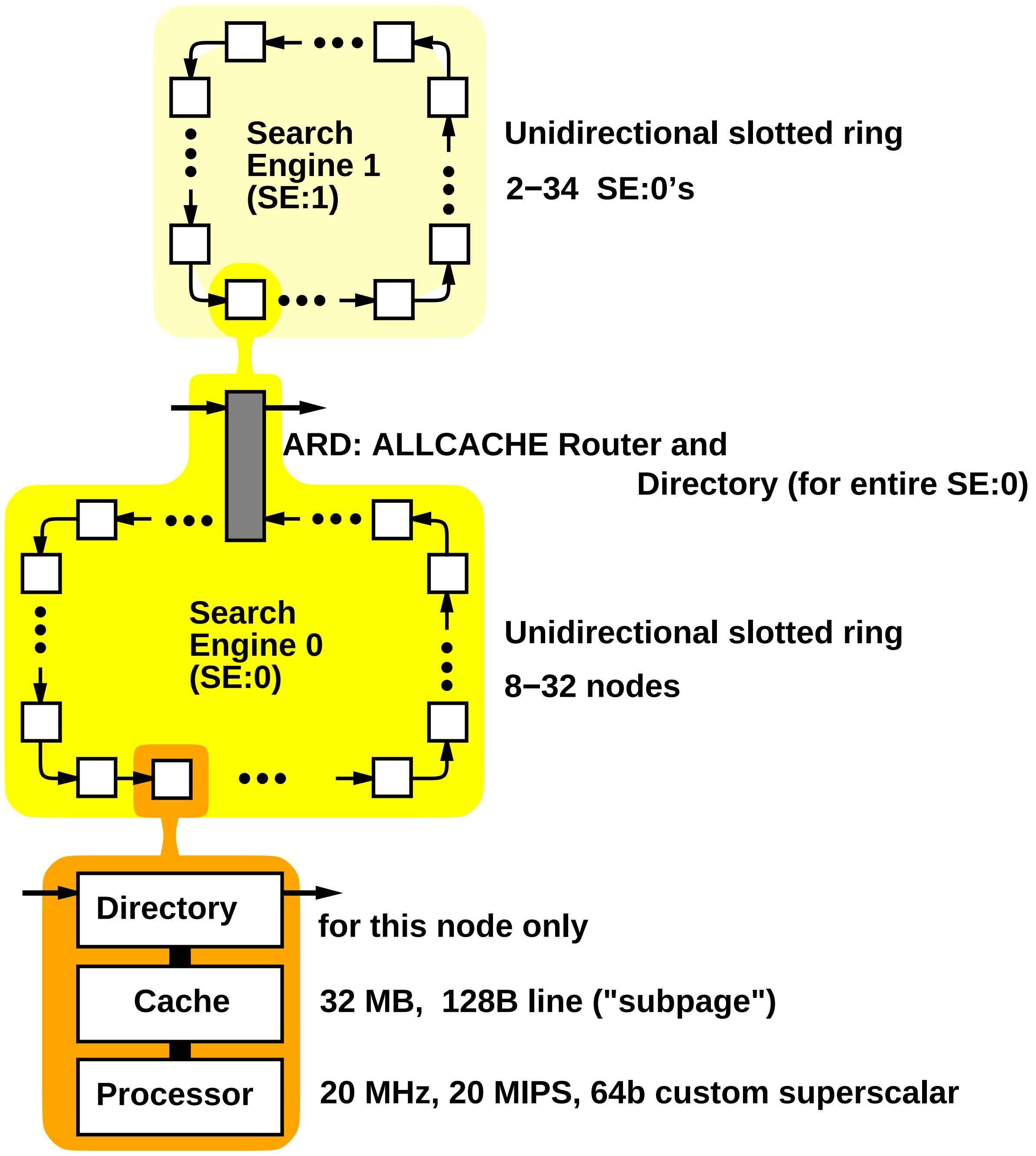
■ 29 ticks	Load from same node
■ > 100 ticks	Load from another node
■ > 130 ticks	Load from another node, dirty copy in 3rd node

(unloaded timings; congestion can add factor of 2x)

Stanford DASH
Further support for "Remote Loads"

- **A form of multithreading:**
  - **Directory for a node can handle multiple remote loads simultaneously**
  - **When one processor waits for a remote load, other processors in the node can continue execution**
- **Prefetch operations:**
  - **A processor can initiate a remote load/store before it really needs the location**
- **Release consistency:**
  - **Instead of guaranteeing consistency on every memory access, guarantee it only on special accesses, i.e., when a lock is released.**
  - **Allows out−of−order memory accesses, fewer write−invalidations, etc.**

Kendall Square Research KSR−1


Search Engine 1 (SE:1)
Unidirectional slotted ring
2−34 SE:0's
ARD: ALLCACHE Router and Directory (for entire SE:0)
Search Engine 0 (SE:0)
Unidirectional slotted ring
8−32 nodes
Directory
Cache
Processor
for this node only
32 MB, 128B line ("subpage")
20 MHz, 20 MIPS, 64b custom superscalar

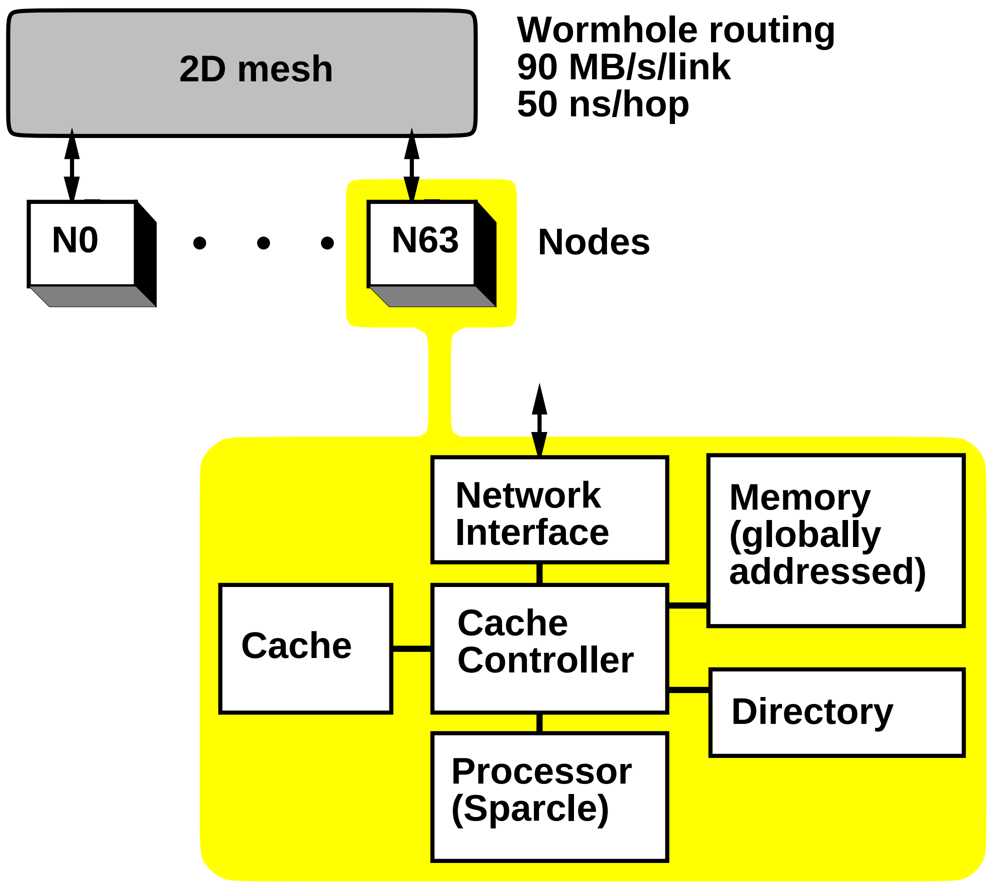
- Nodes: no conventional memory (ALLCACHE$^{\text{TM}}$)
- No fixed "home node" for a location.
  Ownership moves dynamically.
- Cache miss: rload message travels around SE:0, perhaps up into SE:1 and down to another SE:0 looking for owner, then back with response.

Kendall Square Research KSR−1

- **Timings:**
  - **2 ticks** **1st level cache**
  - **18 ticks** **2nd level cache (same node)**
  - **126 ticks** **Another cache, same SE:0**
  - **453 ticks** **Another cache, another SE:0**

- **Additional support for "Remote Loads":**
  - **Prefetch: a processor can initiate a remote load/store before it really needs the location**
  - **Upto 4 prefetches can be in progress for each node**

MIT Alewife



- Sparcle processor: modified 33 MHz Sparc
- Node memory: 4MB shared globally, 4 MB private
- Cache controller for globally shared memory
  - 16 B cache lines
  - "LimitLESS" directory scheme:
    - 5 hardware pointers to other caches
    - Overflow: trap to software, extend in memory

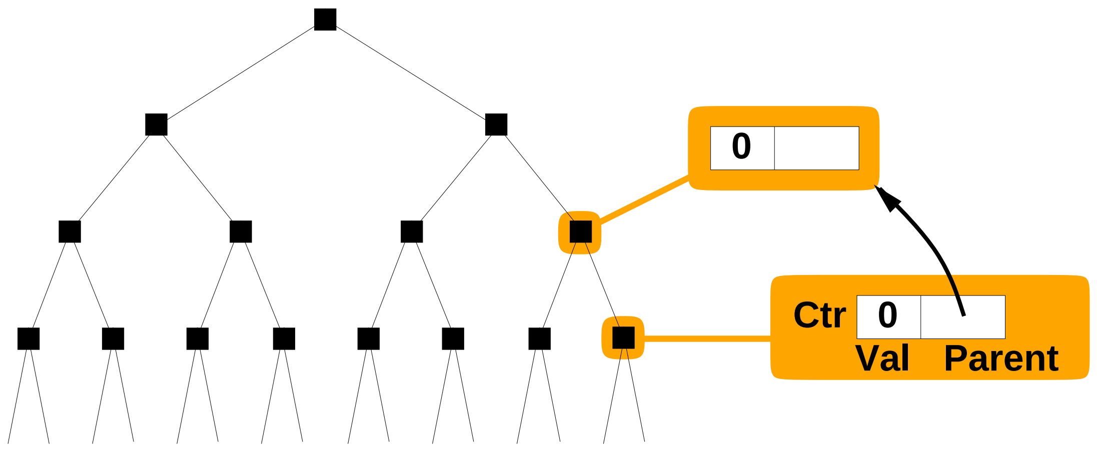
MIT Alewife: Remote Loads

Multithreading in the Sparcle processor

- **Standard Sparc: 8 register windows**
  Sparcle: Use them to hold 4 contexts
- **Each context:**
  - **Normal code's register set**
  - **Trap handler's register set**
- **On cache miss: trap, switch context**
  - **Can be done in 14 ticks**
  - **By the time old context is tried again, remote load should have completed**
  - **Timings (ticks):**
    - **2: cache hit (usual Sparc)**
    - **11: miss, local home**
    - **28 + 2 x hops: cache miss, remote home, clean**
    - **35 + 2 x hops: miss, remote home, dirty at home**
    - **58 + 2 x hops: miss, remote home, dirty elsewhere**

Distributed Cacheing: "Synchronizing Loads" are expensive

- **Example: Barrier Synchronization**



Leaves: Computations

Each leaf's action:

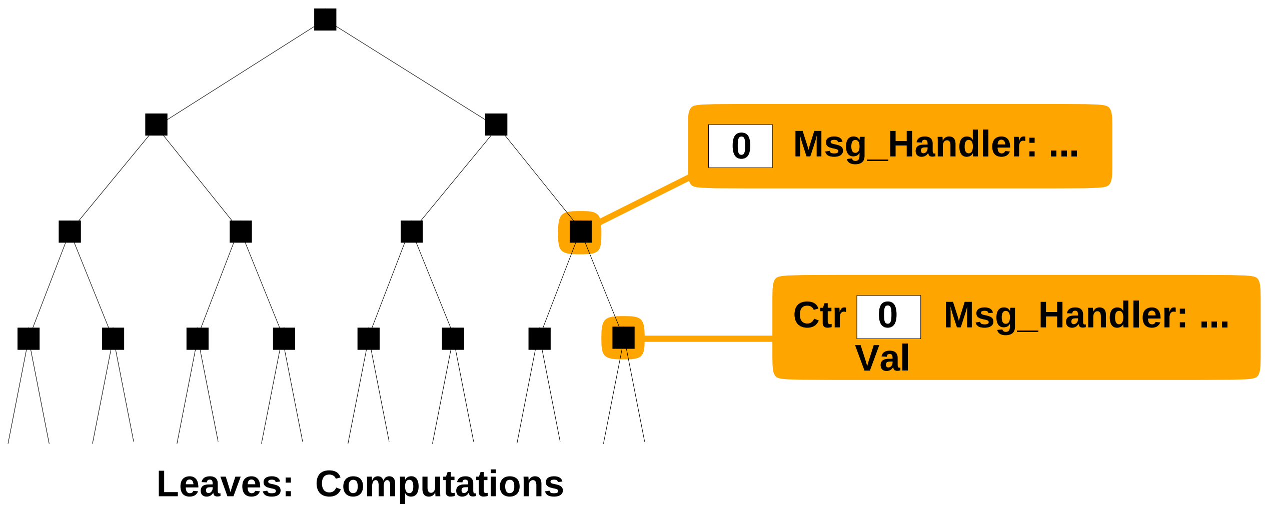
```
Compute (e.g., a loop iteration)
Ctr = address of parent ctr
Load Ctr, for exclusive write              }              . . .  (2 msgs, min)
Incr Ctr−>Val                              } Atomically!
Store Ctr, releasing exclusive write       }              . . .  (1 msg)
If Ctr−>Val == 2 and is root,   DONE!
Ctr = Ctr−>parent
```

If contention for a ctr, could be more messages

(failed load for exclusive write, retry, invalidation, ...)

Distributed Cacheing: "Synchronizing Loads" are expensive

- Example: Barrier Synchronization (contd.)



- Using Message−passing:
  - Leaf:

```
Compute
Send msg to Msg_handler in parent
```

  - Non−Leaf:

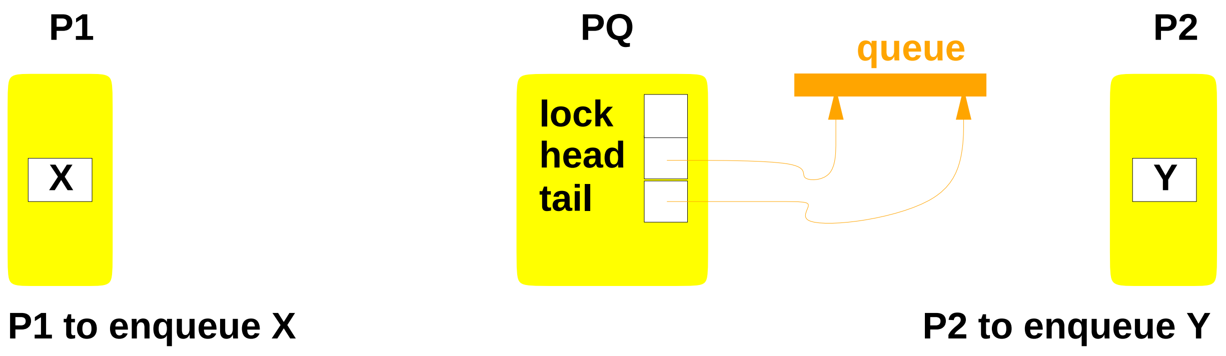
```
Msg_handler:   Incr Ctr−>Val
               If Ctr−>Val == 2
                    If is root,  DONE!
                    Send msg to Msg_handler in parent
```

- Much more efficient than Distrib. Cache method (Exactly 1 message per branch)

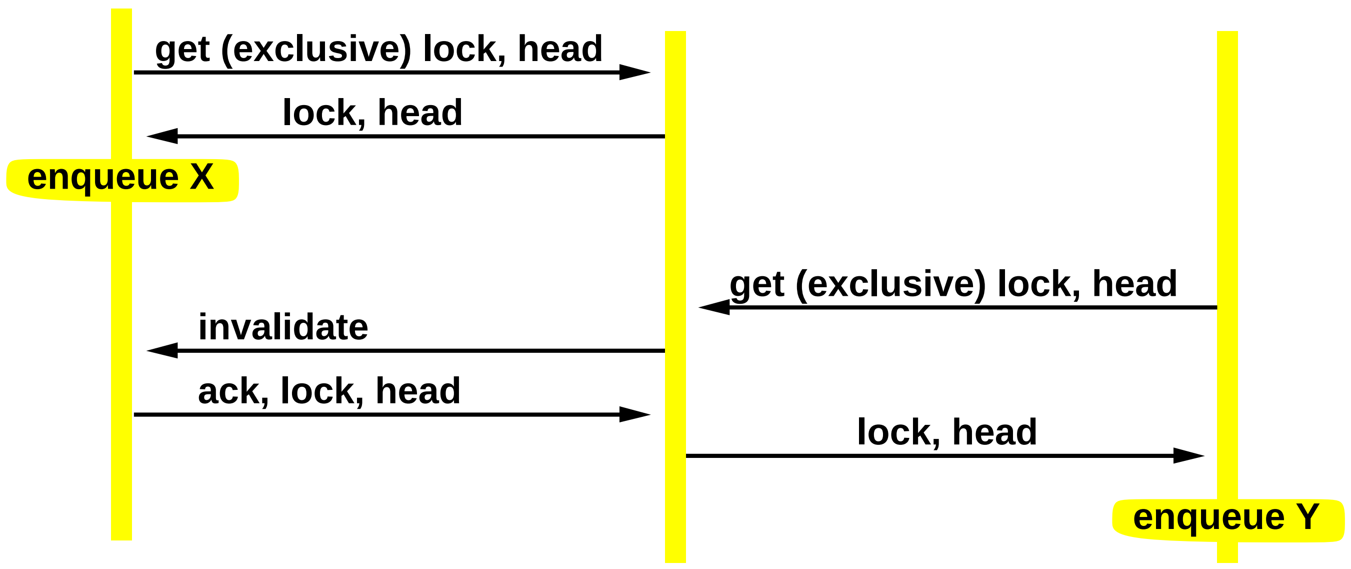
- Further, multithreading would allow Msg_handlers to run without disrupting leaf computations

Distributed Cacheing: "Synchronizing Loads" are expensive

Example: Shared queues

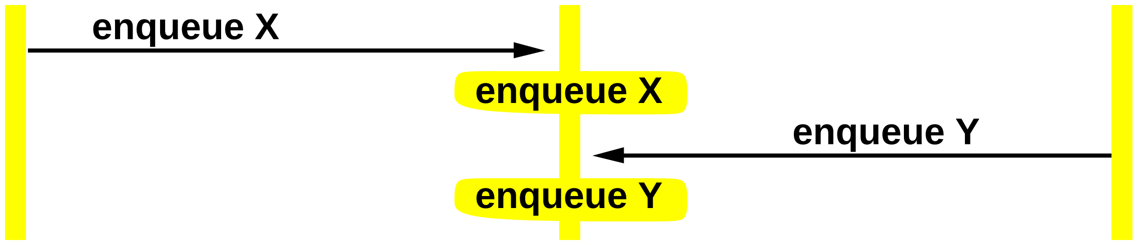


Distributed cacheing:



Contention could add more messages

Message−passing:



Further, multithreading would allows the enqueue handler on PQ to run without disrupting PQ's normal computation

Stanford DASH
Support for "Synchronizing Loads"

- **Update−write**
  - **Update a shared location, send new data immediately to all other nodes with cached copies (instead of invalidating them and later reloading)**
- **Deliver**
  - **Deliver dirty copy to specified caches, make shared**
- **Queue based−locks**
  - **Home node for a lock holds on to list of requestors**
  - **When lock is released, deliver to one requestor**
  - **But, requesting processors are stuck**
- **Fetch−and−op**
  - **Atomic incr/decr at home memory location**
  - **Like NYU Ultracomputer (but, no ‘‘combining’’ in network)**

Kendall Square Research KSR−1 Support for "Synchronizing Loads"

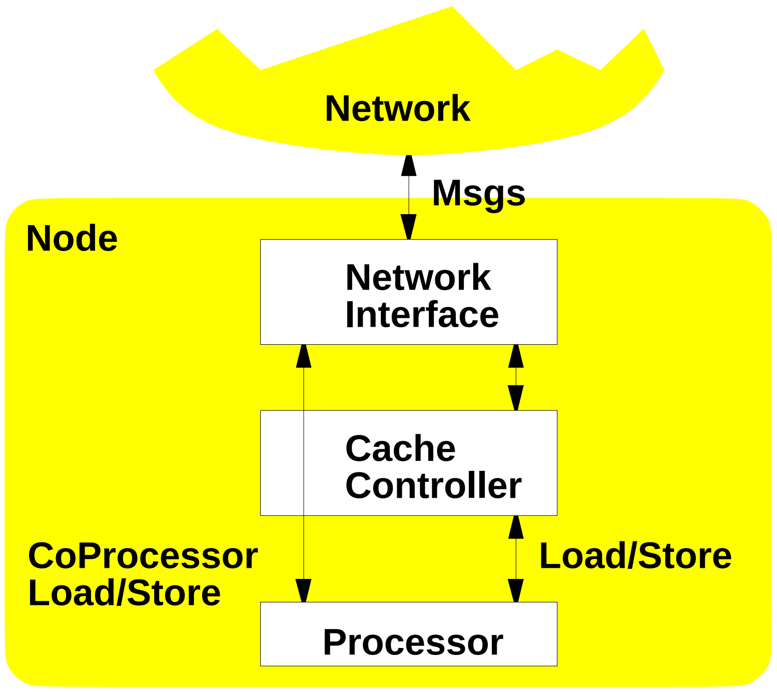
- **Processor can GET a cache line in ‘‘Atomic’’ state later RELEASE it.**
- **Meanwhile, other access attempts return failure flag**
- **Thus, can implement atomic read−modify−write**

MIT Alewife: Synchronizing Loads

FULL/EMPTY bit ("presence bit") on each location

- **Memory ops to read/modify these bits atomically**
- **Trap if bit is not in expected state; handler can:**
  - **Switch context (therefore, busy wait)**
  - **Unload context to try later, load new context**
- **On attempt to read remote empty location, it gets cached locally.**
  - **+ Busy−waiting is cheaper**
  - **+ Building wait queue is cheaper**
  - **− Extra coherence traffic when producer writes**

MIT Alewife: Synchronizing Loads Support for Message−Passing



- Processor can directly compose, send a message to another processor
- Arriving message interrupts processor. Can enter msg−handling thread in 5 cycles
- Msg−handler can directly read message contents
- Details:
  - Msg load/stores: cache access speed
  - User mode (no O.S. involvement)
  - Processor sees 16−word message buffer. DMA encodings allow longer msgs, gather/scatter

Directory−Based Cacheing Final Comments

- **DASH, KSR−1, Alewife leverage existing processor technology**
  - **DASH: off−the−shelf MIPS R3000s**
  - **KSR−1: custom, but ''conventional'' (!)**

    (decision influenced by unavailability of 64−bit off−the−shelf processors at start of project)
  - **Alewife: small modifications of existing Sparc**

- **Advantages:**
  - **Keep up with process technology**
  - **Compatibility with existing software**

- **Issues:**
  - **Less aggressive pursuit of parallelism**
  - **Depend heavily on compilers to obtain locality (feasible for general purpose programs?)**
  - **"Synchronizing Loads" still problematic**