

332

# Advanced Computer Architecture

## Chapter 8

# Parallel architectures, shared memory, and cache coherency

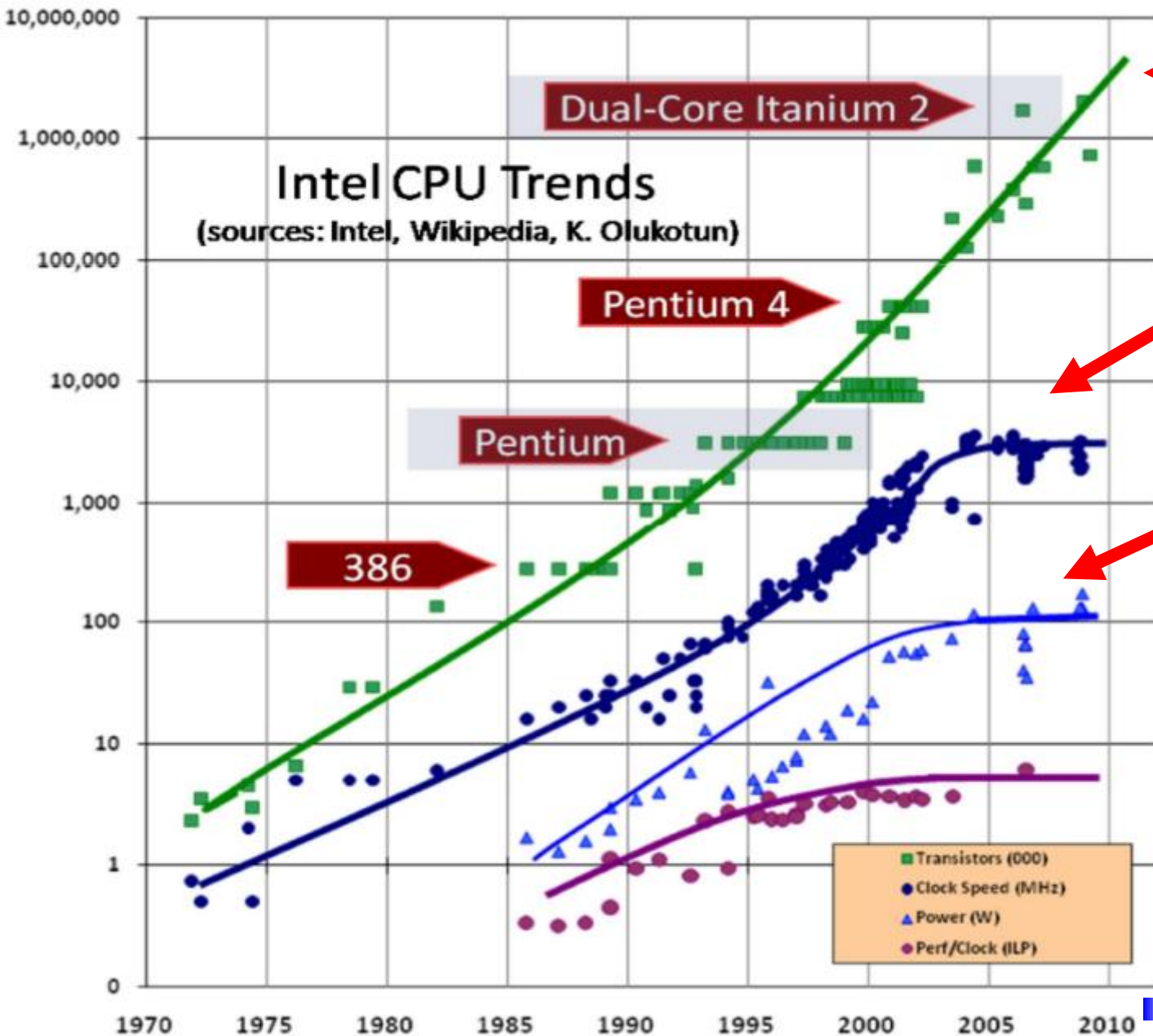
March 2018

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> eds), and on the lecture slides of David Patterson, John Kubiatawicz and Yujia Jin at Berkeley

- Why add another processor?
- What are *large* parallel machines used for?
- How do you program a parallel machine?
- How should they be connected – I/O, memory bus, L2 cache, registers?
- Cache coherency – the problem
- Coherency – what are the rules?
- Coherency using broadcast – update/invalidate
- Coherency using multicast – SMP vs ccNUMA
- Distributed directories, home nodes, ownership; the ccNUMA design space
- Beyond ccNUMA; COMA and Simple COMA
- Hybrids and clusters

# The free lunch is over



Moore's Law  
"escalator"  
continues

Clock speed  
escalator has  
stopped!

Power limit  
caps single-  
core  
performance

# Power is the critical constraint

- Dynamic power vs static leakage
  - **Dynamic**: Power is consumed when signals change
  - **Static**: Power is consumed when gates are powered-up
  - **“Dennard Scaling”**: dynamic power gets smaller if we make the transistors smaller
  - **“the end of Dennard Scaling”**: static leakage starts to dominate, especially at high voltage (that is needed for high clock rate)
- Power vs clock rate
  - Power increases sharply with clock rate because
    - High static leakage due to high voltage
    - High dynamic switching
- Clock vs parallelism: *much* more efficient to use
  - Lots of parallel units, low clock rate, at low voltage

Product	Normalized Performance	Normalized Power	EPI on 65 nm at 1.33 volts (nJ)
i486	1.0	1.0	10
Pentium	2.0	2.7	14
Pentium Pro	3.6	9	24
Pentium 4 (Willamette)	6.0	23	38
Pentium 4 (Cedarmill)	7.9	38	48
Pentium M (Dothan)	5.4	7	15
Core Duo (Yonah)	7.7	8	11

## ■ Energy matters...

- Can we remove the heat?
- TDP (thermal design power) – the amount of power the system can dissipate without exceeding its temperature limit

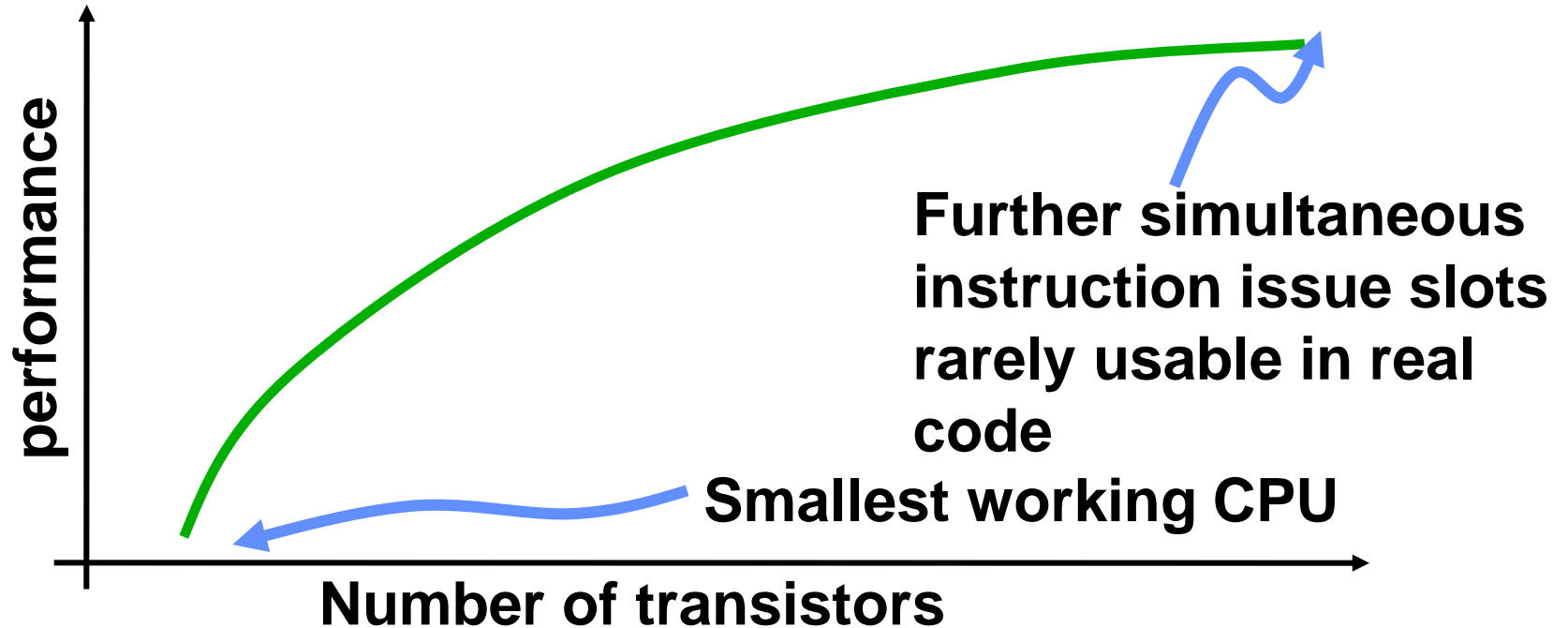
- Energy per instruction
- Assuming identical fabrication technology
- Factor 4 difference across Intel microarchitectures

## ■ Energy matters

- TCO (total cost of ownership)
- £0.12 per KWh
- A few hundred pounds a year for a typical server

- What can we do about power?
- Compute fast then turn it off!
- Compute just fast enough to meet deadline
- Clock gating, power gating
  - Turn units off when they're not being used
  - Functional units
  - Whole cores...
- Dynamic voltage, clock regulation
  - Reduce clock rate dynamically
  - Reduce supply voltage as well
  - Eg when battery is low
  - Eg when CPU is not the bottleneck (*why?*)
- Turbo mode
  - Boost clock rate when only one core is active

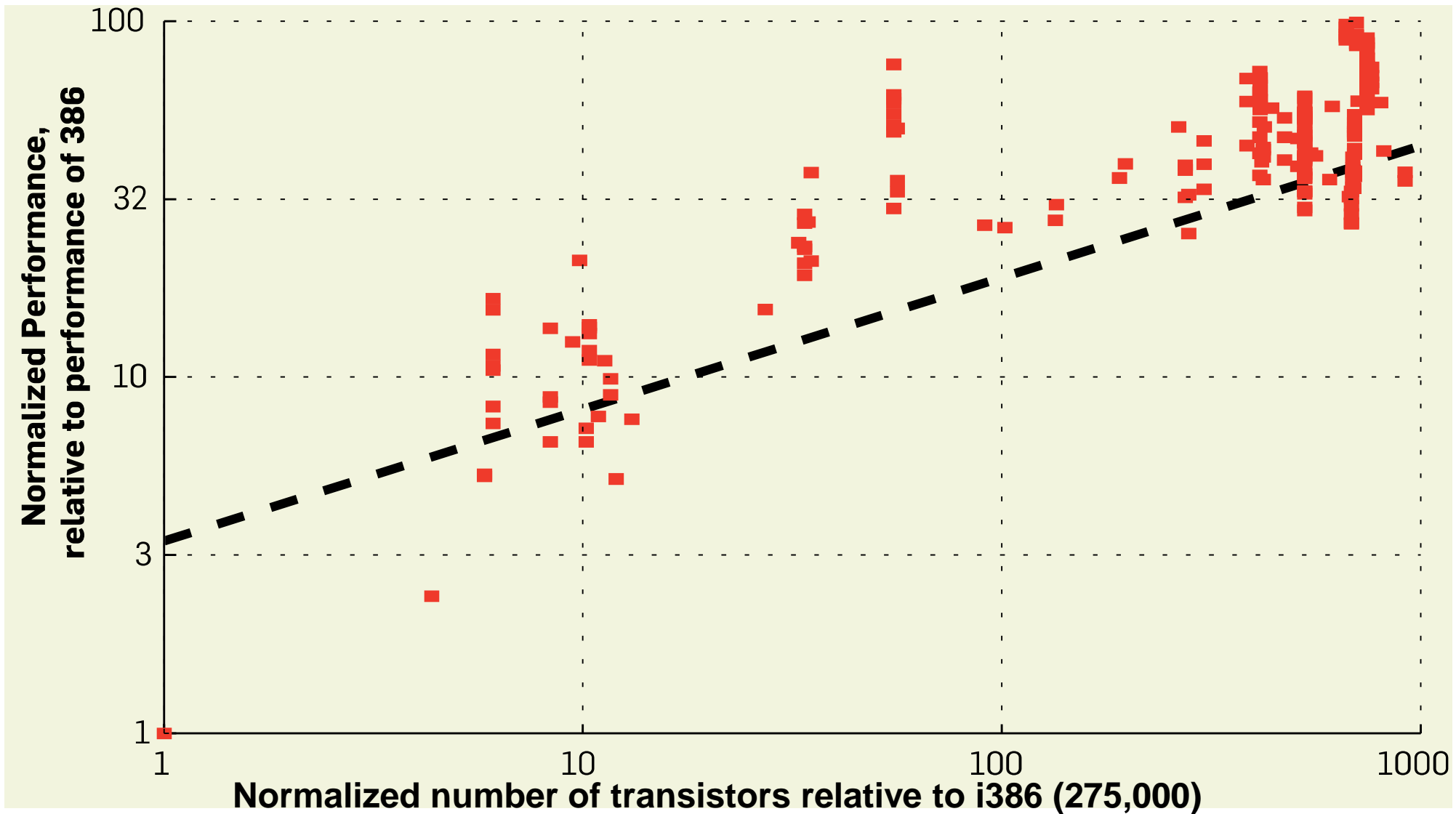
# Why add another processor?



Increasing the complexity of a single CPU leads to diminishing returns

- ➡ Due to lack of instruction-level parallelism
- ➡ Too many simultaneous accesses to one register file
- ➡ Forwarding wires between functional units too long - inter-cluster communication takes >1 cycle
- ➡ Pollack's rule, "performance scales as the square root of design complexity"

# Why add another processor?



See: Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. 2012. CPU DB: recording microprocessor history. *Commun. ACM* 55, 4 (April 2012), 55-63.

DOI=<http://dx.doi.org/10.1145/2133806.2133822>



# How to add another processor?

- ✚ Idea: instead of trying to exploit more instruction-level parallelism by building a bigger CPU, build two - or more
- ✚ This only makes sense if the application parallelism exists...
- ✚ Why might it be better?
  - ➡ No need for multiported register file
  - ➡ No need for long-range forwarding
  - ➡ CPUs can take independent control paths
    - ➡ Still need to synchronise and communicate
    - ➡ Program has to be structured appropriately...

# How to add another processor?

How should the CPUs be connected?

**Idea:** systems linked by network connected via I/O bus

➤ Eg PCIe - Infiniband, Myrinet, Quadrics

**Idea:** CPU/memory packages linked by network connecting main memory units

➤ Eg SGI's NUMALink, Cray's Seastar, Cray's Aries

**Idea:** CPUs share L2/L3 cache

➤ Eg IBM Power4,5,6,7, Intel Core2Duo, Core2Quad, Nehalem, Westmere, Haswell, ARM Cortex a15, etc

**Idea:** CPUs share L1 cache

?

**Idea:** CPUs share registers, functional units

➤ Cray/Tera MTA (multithreaded architecture), Symmetric multithreading (SMT), as in Hyperthreaded Pentium 4, Nehalem, Haswell, Alpha 21464, Power7, etc

# What are parallel computers used for?

## ▶ Executing loops in parallel

- ➡ Improve performance of single application
- ➡ Barrier synchronisation at end of loop
- ➡ Iteration  $i$  of loop 2 may read data produced by iteration  $i$  of loop 1 – but perhaps also from other iterations

## ▶ High-throughput servers

- ➡ Search, cloud-based services
- ➡ Database, transaction processing, web server, e-commerce
- ➡ Maybe also machine learning?
- ➡ Improve performance of single application
- ➡ Consists of many mostly-independent transactions
- ➡ Sharing remote access to data
- ➡ Synchronising to ensure consistency
- ➡ Transaction roll-back

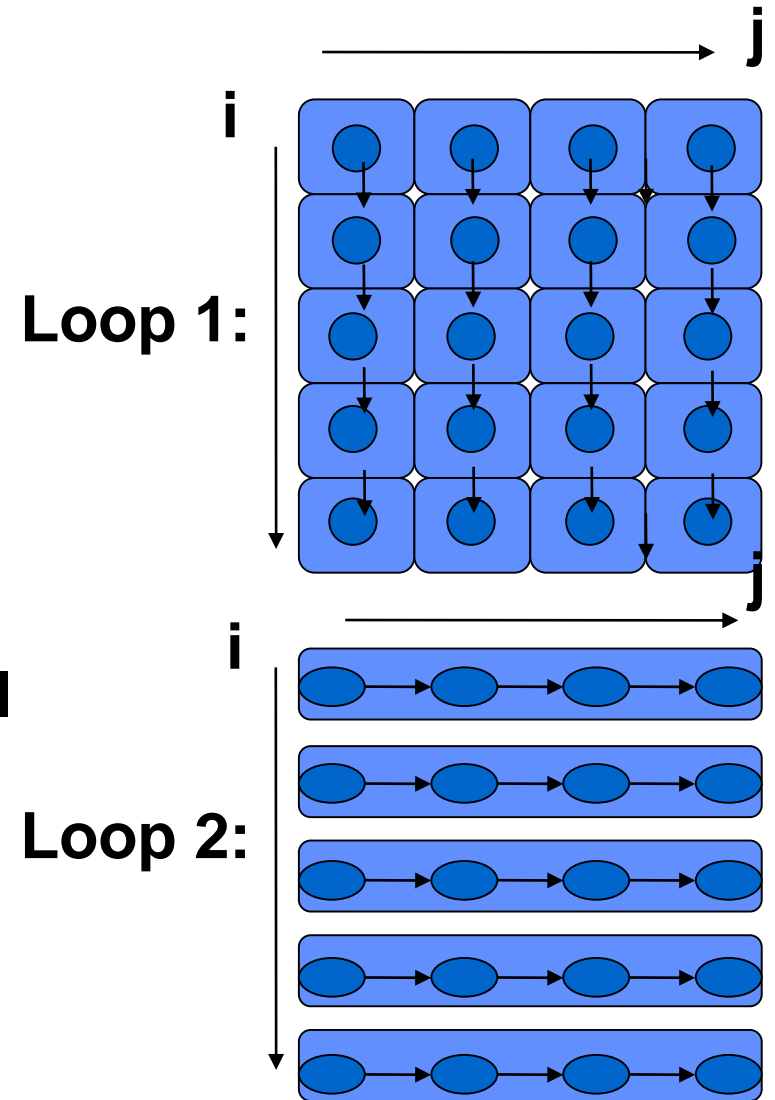
## ▶ Mixed, multiprocessing workloads

# How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
for(i=0; i<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; i<N; ++i)
  for(j=0; j<M; ++j)
    A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory... no problem!



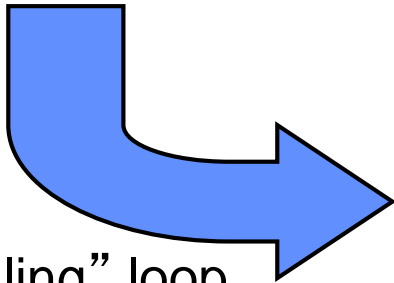
# Shared-memory parallel - OpenMP

- OpenMP is a standard design for language extensions for shared-memory parallel programming
- Language bindings exist for Fortran, C, C++ and to some extent (eg research prototypes) for Java and C#
- Implementation requires compiler support – as found in GCC, clang/llvm, Intel's compilers, Microsoft Visual Studio, Apple Xcode
- Example:

```
for(i=0; i<N; ++i)
    #pragma omp parallel for
    for(j=0; j<M; ++j)
        A[i,j] = (A[i-1,j] + A[i,j])*0.5;
#pragma omp parallel for
for(i=0; i<N; ++i)
    for(j=0; j<M; ++j)
        A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

# Implementing shared-memory parallel loop

```
for (i=0; i<N; i++) {  
    C[i] = A[i] + B[i];  
}
```



```
if (myThreadId() == 0)
```

```
    i = 0;
```

```
    barrier();
```

```
// on each thread
```

```
while (true) {
```

```
    local_i = FetchAndAdd(&i);
```

```
    if (local_i >= N) break;
```

```
    C[local_i] = 0.5*(A[local_i] + B[local_i]);
```

```
}
```

```
    barrier();
```

Barrier(): block  
until all threads  
reach this point

“self-scheduling” loop

FetchAndAdd() is atomic  
operation to get next un-  
executed loop iteration:

```
Int FetchAndAdd(int *i) {
```

```
    lock(i);
```

```
    r = *i;
```

```
    *i = *i+1;
```

```
    unlock(i);
```

```
    return(r);
```

```
}
```

## Optimisations:

- Work in chunks
- Avoid unnecessary barriers
- Exploit “cache affinity” from loop to loop

There are smarter ways to implement  
FetchAndAdd....

We could use locks:

```
Int FetchAndAdd(int *i) {  
    lock(i);  
    r = *i;  
    *i = *i+1;  
    unlock(i);  
    return(r);  
}
```

# Implementing Fetch-and-add

- Using locks is rather expensive (and we should discuss how they would be implemented)
- But on many processors there is support for atomic increment
- So use the GCC built-in:  
`__sync_fetch_and_add(p, inc)`
- Eg on x86 this is implemented using the “exchange and add” instruction in combination with the “lock” prefix:  
`LOCK XADDL r1 r2`
- The “lock” prefix ensures the exchange and increment are executed on a cache line which is held exclusively

Combining:

- In a large system, using FetchAndAdd() for parallel loops will lead to contention
- But FetchAndAdds can be combined in the *network*
- When two FetchAndAdd(p,1) messages meet, combine them into one FetchAndAdd(p,2) – and when it returns, pass the two values back.

# More OpenMP

```
#pragma omp parallel for \  
    default(shared) private(i) \  
    schedule(static,chunk) \  
    reduction(+:result)  
for (i=0; i < n; i++)  
    result = result + (a[i] * b[i]);
```

➤ **default(shared) private(i):**

All variables except i are shared by all threads.

➤ **schedule(static,chunk):**

Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the “team”

➤ **reduction(+:result):**

performs a reduction on the variables that appear in its argument list

- A private copy for each variable is created for each thread. At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.



# Distributed-memory parallel - MPI

- ❖ MPI (“Message-passing Interface”) is a standard API for parallel programming using message passing

- ❖ Usually implemented as library

- ❖ Six basic calls:

- ➡ MPI\_Init - Initialize MPI
- ➡ MPI\_Comm\_size - Find out how many processes there are
- ➡ MPI\_Comm\_rank - Find out which process I am
- ➡ MPI\_Send - Send a message
- ➡ MPI\_Recv - Receive a message
- ➡ MPI\_Finalize - Terminate MPI

- ❖ Key idea: collective operations

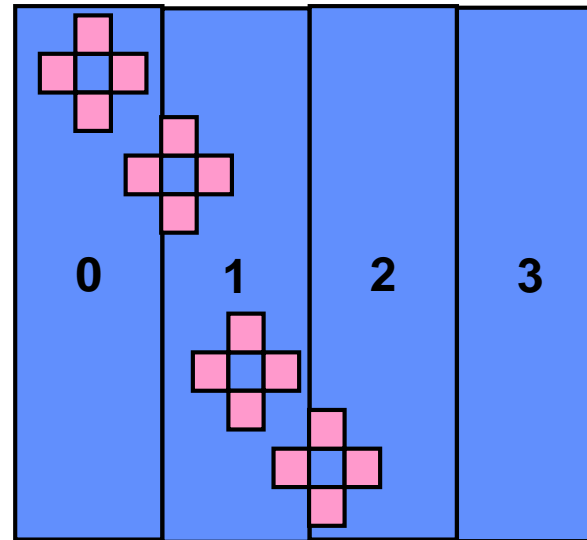
- ➡ MPI\_Bcast - broadcast data from the process with rank "root" to all other processes of the group.
- ➡ MPI\_Reduce – combine values on all processes into a single value using the operation defined by the parameter op.

# MPI Example: stencil

“stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
  DO i=1, n
    B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
  END DO
END DO
```

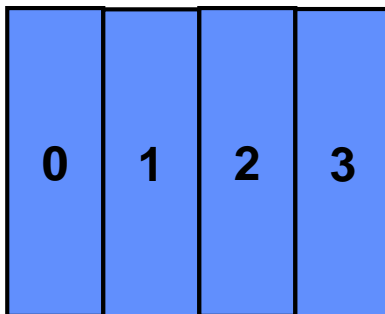
- ➡ To do this in parallel we could simply partition the outer loop
- ➡ At the strip boundaries, we need access to a column of neighbour data values
- ➡ In MPI we have to make this communication explicit



# MPI Example: initialisation

## SPMD

- “Single Program, Multiple Data”
- Each processor executes the program
- First has to work out what part it is to play
- “myrank” is index of this CPU
- “comm” is MPI “communicator” – abstract index space of  $p$  processors
- In this example, array is partitioned into slices



**! Compute number of processes and myrank**

**CALL MPI\_COMM\_SIZE(comm, p, ierr)**

**CALL MPI\_COMM\_RANK(comm, myrank, ierr)**

**! compute size of local block**

**m = n/p**

**IF (myrank.LT.(n-p\*m)) THEN**

**m = m+1**

**END IF**

**! Compute neighbors**

**IF (myrank.EQ.0) THEN**

**left = MPI\_PROC\_NULL**

**ELSE left = myrank - 1**

**END IF**

**IF (myrank.EQ.p-1) THEN**

**right = MPI\_PROC\_NULL**

**ELSE right = myrank+1**

**END IF**

**! Allocate local arrays**

**ALLOCATE (A(0:n+1,0:m+1), B(n,m))**

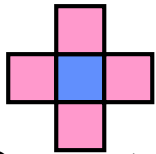


## Example: Jacobi2D

!Main Loop

DO WHILE(.NOT.converged)

- ➡ Sweep over A computing moving average of neighbouring four elements



- ➡ Compute new array B from A, then copy it back into B

- ➡ This version tries to overlap communication with computation

! compute boundary iterations so they're ready to be sent right away

DO i=1, n

$B(i,1)=0.25*(A(i-1,j)+A(i+1,j)+A(i,0)+A(i,2))$

$B(i,m)=0.25*(A(i-1,m)+A(i+1,m)+A(i,m-1)+A(i,m+1))$

END DO

! Communicate

CALL MPI\_ISEND(B(1,1),n, MPI\_REAL, left, tag, comm, req(1), ierr)

CALL MPI\_ISEND(B(1,m),n, MPI\_REAL, right, tag, comm, req(2), ierr)

CALL MPI\_IRECV(A(1,0),n, MPI\_REAL, left, tag, comm, req(3), ierr)

CALL MPI\_IRECV(A(1,m+1),n, MPI\_REAL, right, tag, comm, req(4), ierr)

! Compute interior

DO j=2, m-1

DO i=1, n

$B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))$

END DO

END DO

DO j=1, m

DO i=1, n

$A(i,j) = B(i,j)$

END DO

END DO

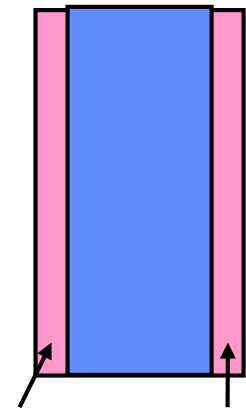
! Complete communication

DO i=1, 4

CALL MPI\_WAIT(req(i), status(1.i), ierr)

END DO

END DO




B(1:n,1)

B(1:n,m)

<http://www.netlib.org/utk/papers/mpi-book/node51.html>

# MPI vs OpenMP

 **Which is better – OpenMP or MPI?**

# MPI vs OpenMP

- Which is better – OpenMP or MPI?

- OpenMP is easy!

  - But it hides the communication

  - And unintended sharing can lead to tricky bugs

# MPI vs OpenMP

- Which is better – OpenMP or MPI?

- OpenMP is easy!

  - But it hides the communication

  - And unintended sharing can lead to tricky bugs

- MPI is hard work

  - You need to make data partitioning explicit

  - No hidden communication

  - Seems to require more copying of data

# MPI vs OpenMP

- Which is better – OpenMP or MPI?

- OpenMP is easy!

  - But it hides the communication

  - And unintended sharing can lead to tricky bugs

- MPI is hard work

  - You need to make data partitioning explicit

  - No hidden communication

  - Seems to require more copying of data

  - It's easier to see how to reduce communication and synchronisation (?)

- Lots of research on better parallel programming models...



332

# Advanced Computer Architecture

## Chapter 8.5

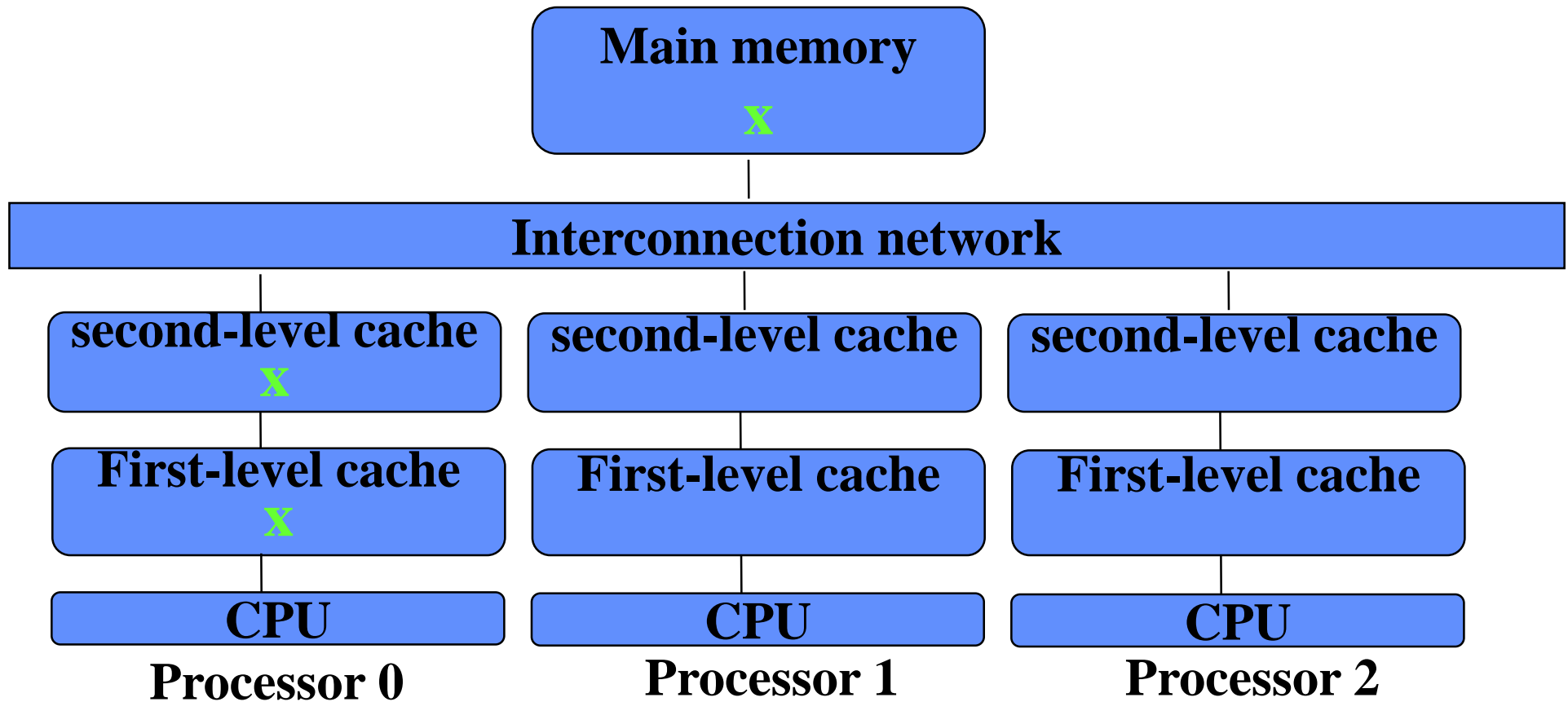
# Cache coherency

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> eds), and on the lecture slides of David Patterson, John Kubiawicz and Yujia Jin at Berkeley

- ▀ Snooping cache coherency protocols
  - ▀ Bus-based, small-scale
- ▀ Directory-based cache coherency protocols
  - ▀ Scalable shared memory
  - ▀ ccNUMA – coherent-cache non-uniform memory access
- ▀ Synchronisation and atomic operations
- ▀ Memory consistency models
- ▀ Case study – Sun's S3MP

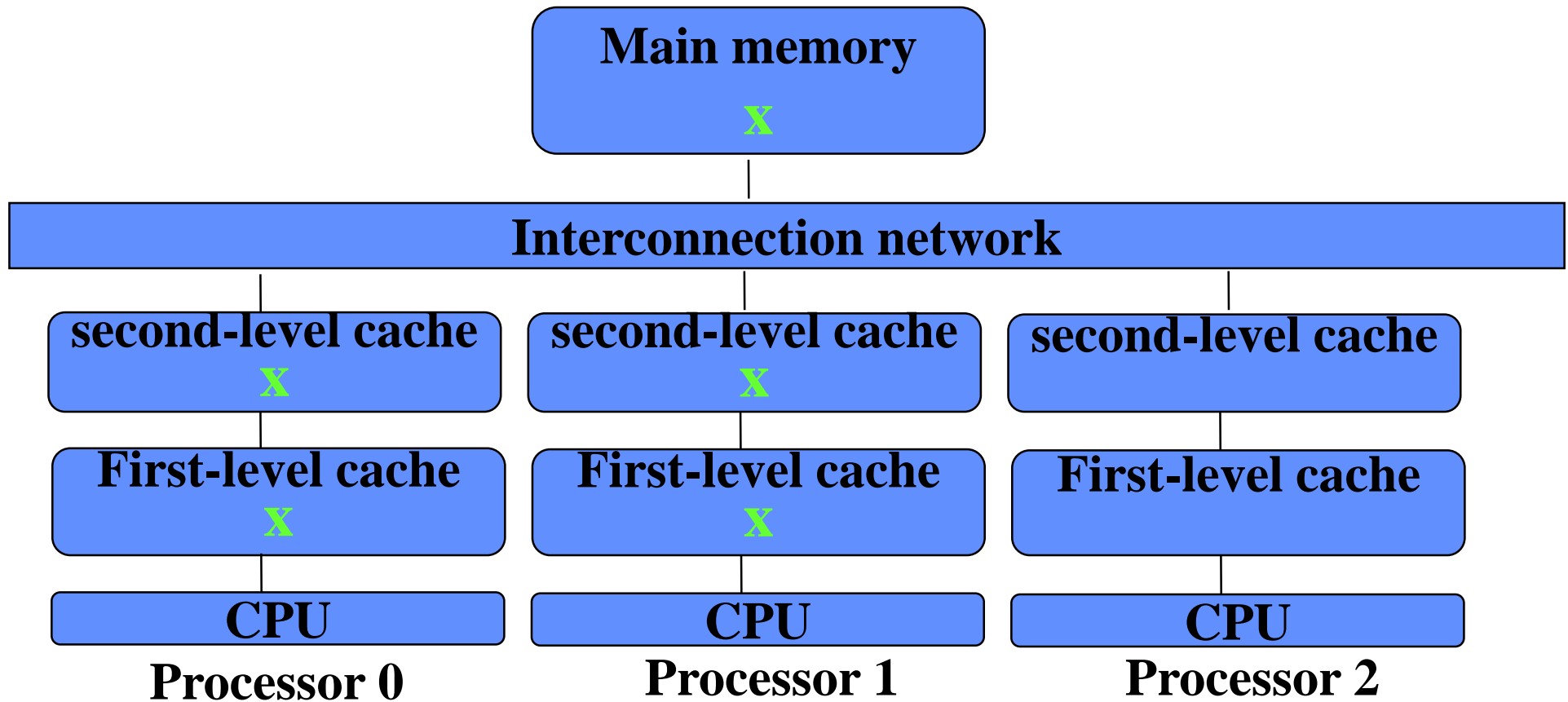
# Multiple caches... and trouble



Suppose processor 0 loads memory location **X**

**X** is fetched from main memory and allocated into processor 0's cache(s)

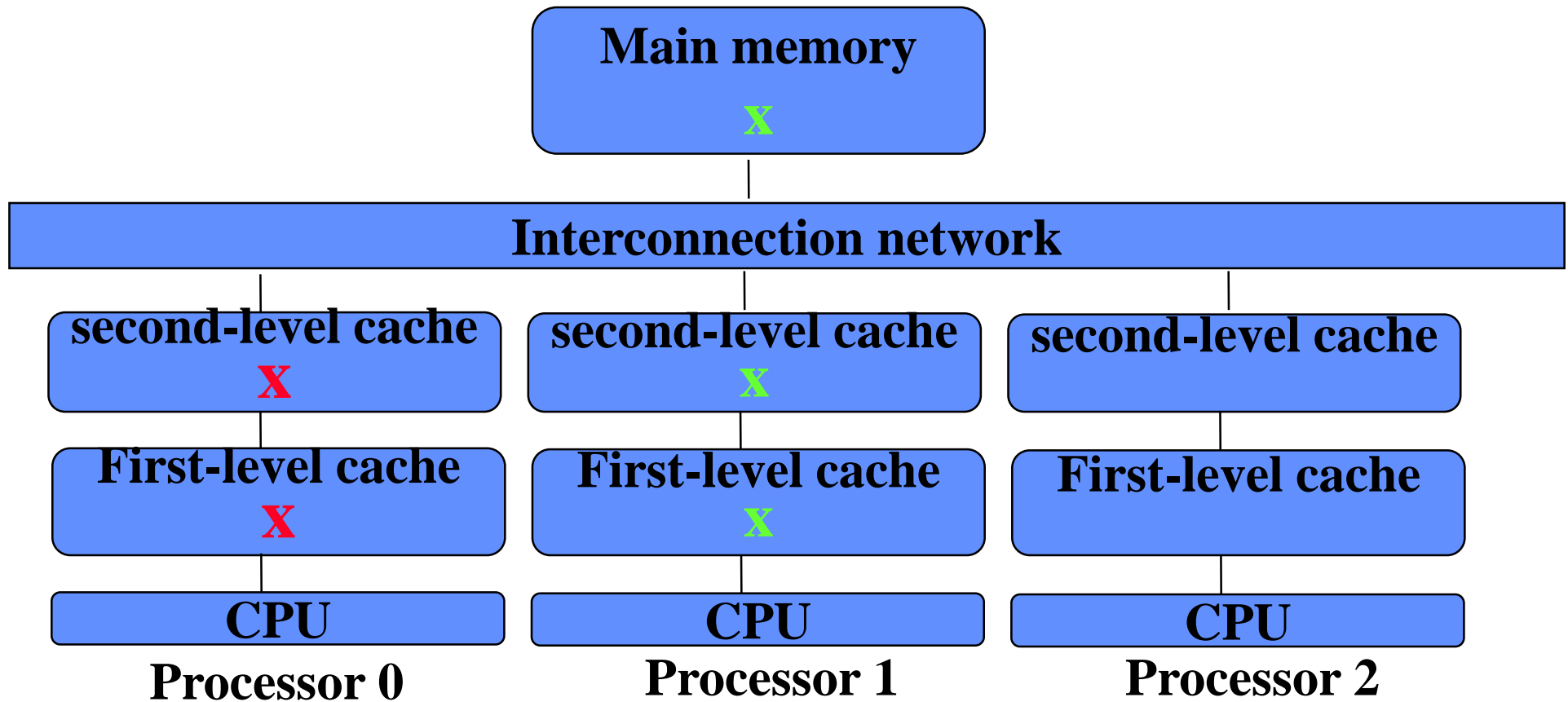
# Multiple caches... and trouble



Suppose processor 1 loads memory location **X**

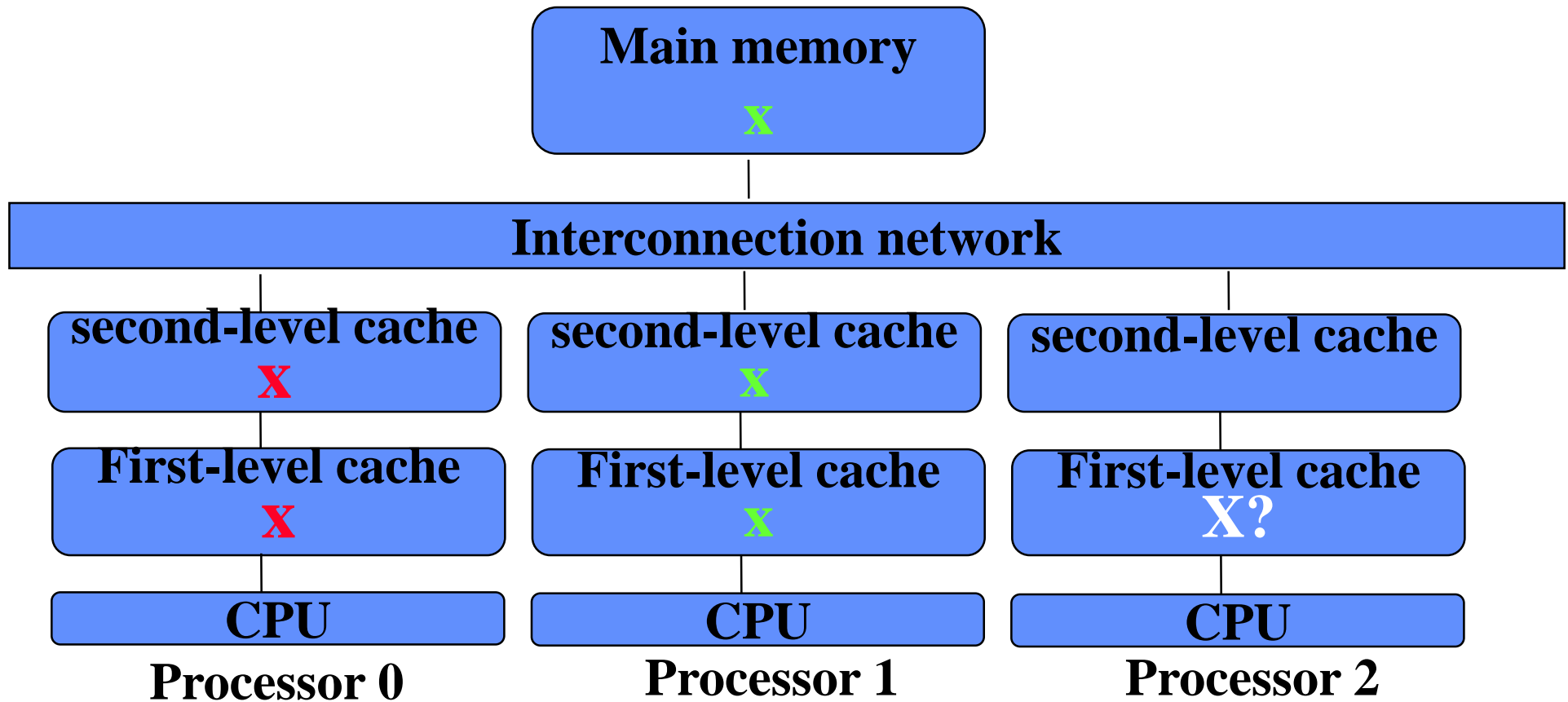
**X** is fetched from main memory and allocated into processor 1's cache(s) as well

# Multiple caches... and trouble



- Suppose processor 0 stores to memory location **X**
- Processor 0's cached copy of **X** is updated
- Processor 1 continues to use the old value of **X**


# Multiple caches... and trouble



Suppose processor 2 loads memory location **X**

How does it know whether to get **x** from main memory, processor 0 or processor 1?

# Implementing distributed, shared memory

-  Two issues:
1. How do you know where to find the latest version of the cache line?
  2. How do you know when you can use your cached copy – and when you have to look for a more up-to-date version?

We will find answers to this after first thinking about what a distributed shared memory implementation is supposed to do...

# Cache consistency (aka cache coherency)

## Goal (?):

- ➡ “Processors should not continue to use out-of-date data indefinitely”

## Goal (?):

- ➡ “Every load instruction should yield the result of the most recent store to that address”

## Goal (?): (definition: **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”

*(Leslie Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs” (IEEE Trans Computers Vol.C-28(9) Sept 1979)*



# Implementing Strong Consistency: update

✚ Idea #1: when a store to address  $x$  occurs, **update** all the remote cached copies

✚ To do this we need either:

- ➡ To broadcast every store to every remote cache
- ➡ Or to keep a list of which remote caches hold the cache line
- ➡ Or at least keep a note of whether there are *any* remote cached copies of this line (“SHARED” bit per line)

✚ But first...how well does this update idea work?

# Implementing Strong Consistency: update...

## Problems with update

1. What about if the cache line is several words long?
  - ➡ Each update to each word in the line leads to a broadcast
2. What about old data which other processors are no longer interested in?
  - ➡ We'll keep broadcasting updates indefinitely...
  - ➡ Do we really have to broadcast *every* store?
  - ➡ It would be nice to know that we have exclusive access to the cacheline so we don't have to broadcast updates...

# A more cunning plan... invalidation

- Suppose instead of **updating** remote cache lines, we **invalidate** them all when a store occurs?
- After the first write to a cache line we know there are no remote copies – so subsequent writes don't lead to communication
  - After invalidation we *know* we have the *only* copy
- Is invalidate always better than update?
  - Often
  - But not if the other processors really need the new data as soon as possible
- To exploit this, we need a couple of bits for each cache line to track its sharing state
- (analogous to write-back vs write-through caches)

# The “Berkeley” Protocol

Four cache line states:

- **INVALID**

Broadcast invalidations on bus  
unless cache line is  
exclusively “owned” (DIRTY)

- **VALID** : clean, potentially shared, unowned

- **SHARED-DIRTY** : modified, possibly shared, owned

- **DIRTY** : modified, only copy, owned

- **Read miss:**

- If another cache has the line in **SHARED-DIRTY** or **DIRTY**,

- it is supplied
    - changing state to **SHARED-DIRTY**

- Otherwise

- the line comes from memory. The state of the
    - line is set to **VALID**

- **Write hit:**

- No action if line is **DIRTY**

- If **VALID** or **SHARED-DIRTY**,

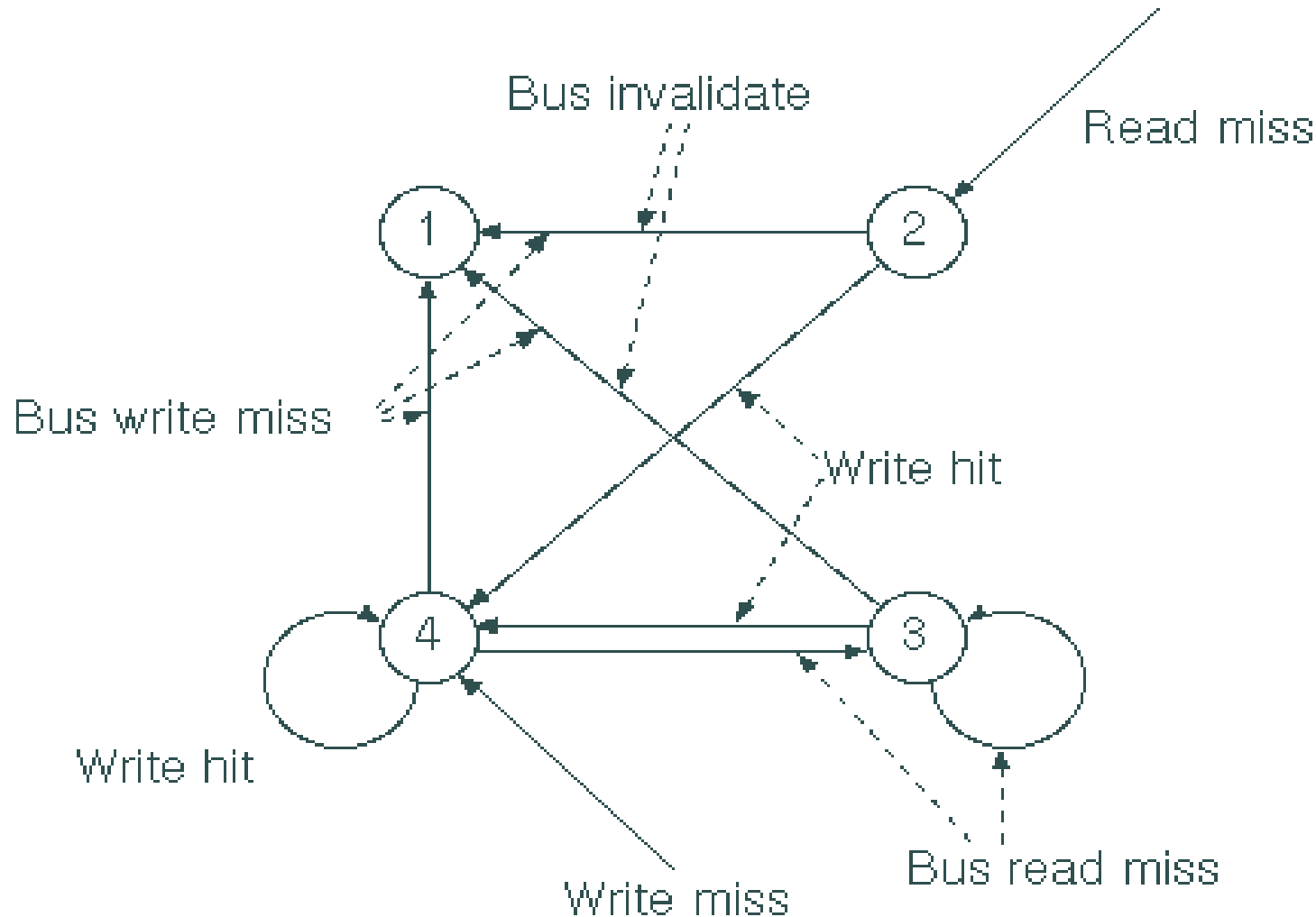
- an invalidation is sent, and
    - the local state set to **DIRTY**

- **Write miss:**

- Line comes from owner (as with read miss).

- All other copies set to **INVALID**, and line in requesting cache is set to **DIRTY**

Berkeley cache coherence protocol: state transition diagram



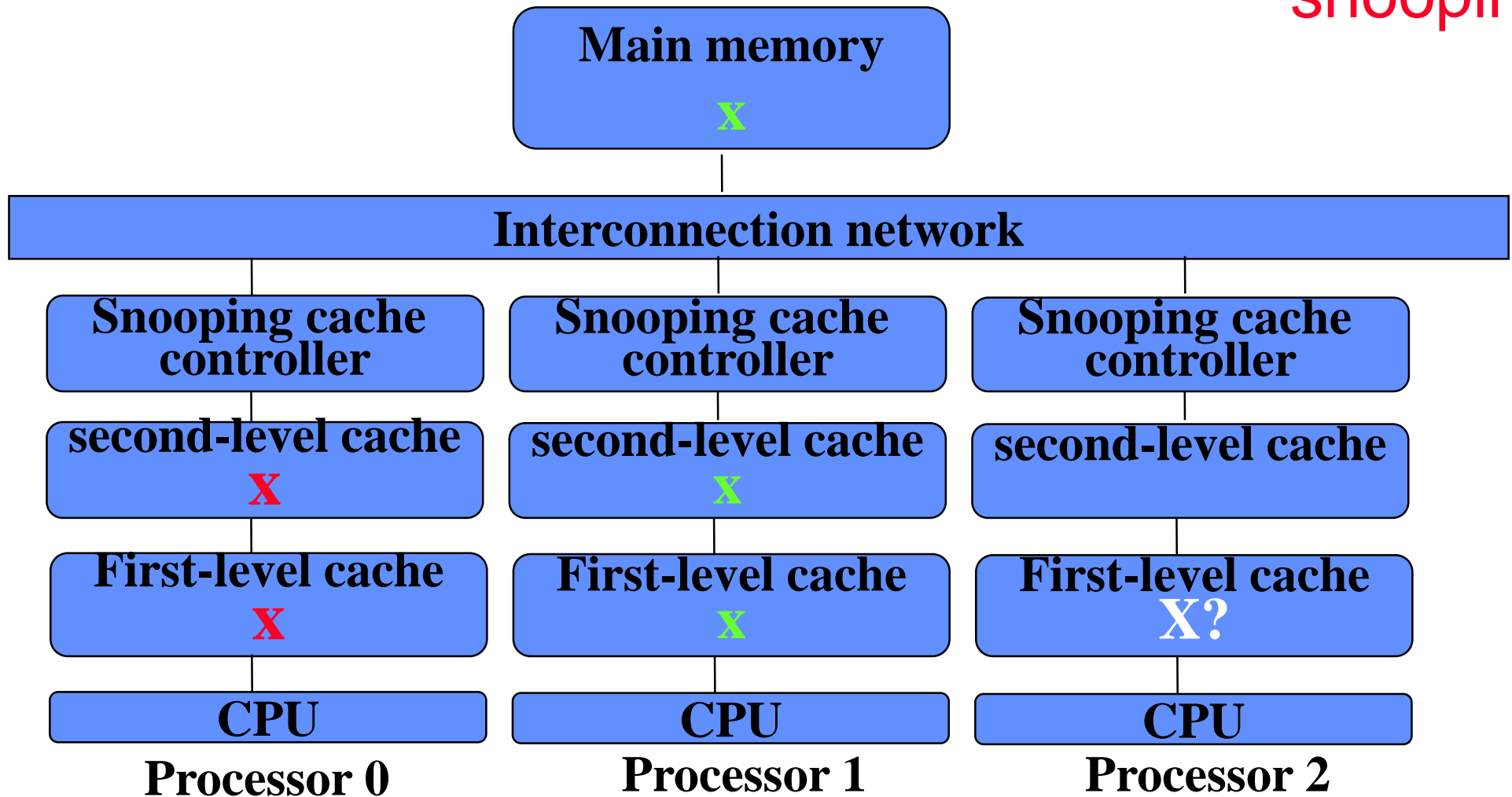
**The Berkeley protocol is representative of how typical bus-based SMPs work**

**Q: What has to happen on a “Bus read miss”?**

1. INVALID
2. VALID: clean, potentially shared, unowned
3. SHARED-DIRTY: modified, possibly shared, owned
4. DIRTY: modified, only copy, owned

# The job of the cache controller - snooping

- ✚ The protocol state transitions are implemented by the cache controller – which “snoops” all the bus traffic
- ✚ Transitions are triggered either by
  - ➡ the bus (Bus invalidate, Bus write miss, Bus read miss)
  - ➡ The CPU (Read hit, Read miss, Write hit, Write miss)
- ✚ For every bus transaction, it looks up the directory (cache line state) information for the specified address
  - ➡ If this processor holds the only valid data (DIRTY), it responds to a “Bus read miss” by providing the data to the requesting CPU
  - ➡ If the memory copy is out of date, one of the CPUs will have the cache line in the SHARED-DIRTY state (because it updated it last) – so must provide data to requesting CPU
  - ➡ State transition diagram doesn’t show what happens when a cache line is displaced...



➤ Snooping cache controller has to monitor *all* bus transactions

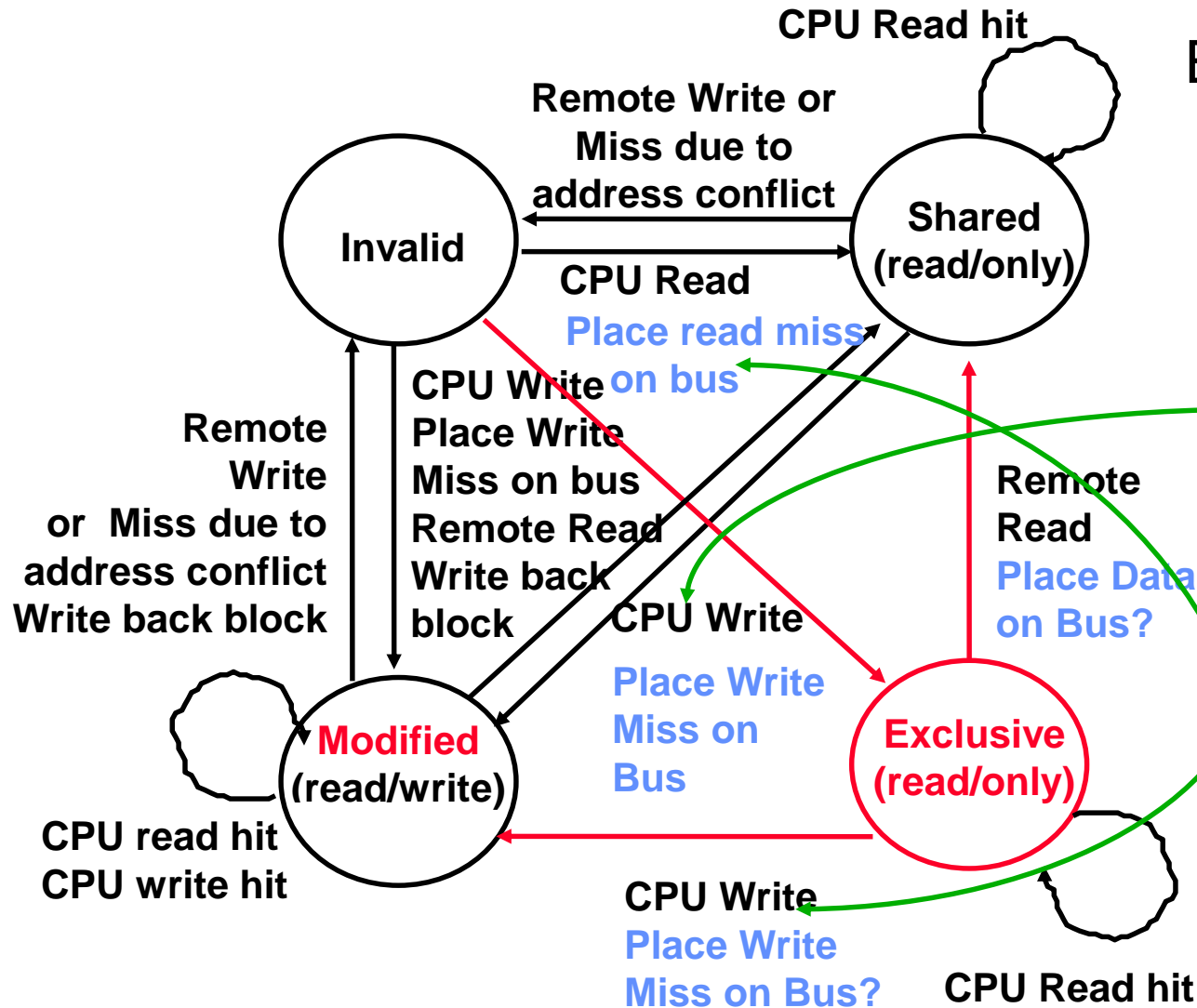
➤ And check them against the tags of its cache(s)

# Berkeley protocol - summary

- Invalidate is usually better than update
- Cache line state “DIRTY” bit records whether remote copies exist
  - ➡ If so, remote copies are invalidated by broadcasting message on bus – cache controllers snoop all traffic
- Where to get the up-to-date data from?
  - ➡ Broadcast read miss request on the bus
  - ➡ If this CPU’s copy is DIRTY, it responds
  - ➡ If no cache copies exist, main memory responds
  - ➡ If several copies exist, the CPU which holds it in “SHARED-DIRTY” state responds
  - ➡ If a SHARED-DIRTY cache line is displaced, ... need a plan
- How well does it work?
  - ➡ See extensive analysis in Hennessy and Patterson



# Snoop Cache Extensions



## Extensions:

- ➡ **Fourth State: Ownership**
  - **Shared-> Modified,**  
need invalidate only  
(upgrade request), don't  
read memory
- Berkeley Protocol**
- **Clean exclusive state (no  
miss for private data on  
write)**
- MESI Protocol**
- **Cache supplies data when  
shared state  
(no memory access)**
- Illinois Protocol**

# Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive	Owned Exclusive	Private Dirty	<u>M</u> odified (private, !=Memory)
Shared	Owned Shared	Private Clean	e <u>X</u> clusive (private, =Memory)
Invalid	Shared	Shared	<u>S</u> hared (shared, =Memory)
	Invalid	Invalid	<u>I</u> nvalid

Owner can update via bus invalidate operation  
 Owner must write back when replaced in cache

If read sourced from memory, then Private Clean  
 If read sourced from other cache, then Shared  
 Can write in cache if held private clean or dirty

# Implementing Snooping Caches

- All processors must be on the bus, with access to both addresses and data
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, there could be contention between bus and CPU access:
  - solution 1: **duplicate set of tags for L1 caches** just to allow checks in parallel with CPU
  - solution 2: **Use the L2 cache to “filter” invalidations**
    - *If everything in L1 is also in L2*
    - Then we only have to check L1 if the L2 tag matches
  - **Many systems enforce cache inclusivity**
    - Constrains cache design - block size, associativity

# Synchronization and atomic operations

➤ Why Synchronize? Need to know when it is safe for different processes to use shared data

➤ Issues for Synchronization:

- Uninterruptable instruction to fetch and update memory (atomic operation);
- User level synchronization operation using this primitive;
- For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

# Uninterruptable Instructions to Fetch from and Update Memory

- **Atomic exchange:** interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - ➡ Set register to 1 & swap
  - ➡ New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
  - ➡ Key is that exchange operation is indivisible
- **Test-and-set:** tests a value and sets it if the value passes the test
- **Fetch-and-increment:** it returns the value of a memory location and atomically increments it
  - ➡ 0 => synchronization variable is free

# Uninterruptable Instruction to Fetch and Update Memory

✚ Hard to have read & write in 1 instruction: use 2 instead

✚ **Load linked** (or load locked) + **store conditional**

- ➡ Load linked returns the initial value
- ➡ Store conditional returns 1 if it succeeds
  - ➡ Succeeds if there has been no other store to the same memory location since the preceding load) and 0 otherwise
  - ➡ 1e if no invalidation has been received

✚ Example doing atomic swap with LL & SC:

```
try:  mov    R3,R4           ; mov exchange value
      ll     R2,0(R1)        ; load linked
      sc     R3,0(R1)        ; store conditional
      beqz   R3,try          ; branch store fails (R3 = 0)
      mov    R4,R2          ; put load value in R4
```

✚ Example doing fetch & increment with LL & SC:

```
try:  ll     R2,0(R1)        ; load linked
      addi   R2,R2,#1        ; increment (OK if reg-reg)
      sc     R2,0(R1)        ; store conditional
      beqz   R2,try          ; branch store fails (R2 = 0)
```

**Implementation:**

**Check that no  
invalidation for the  
target line has  
been received**

**This idea  
generalises to  
...transactions...**

Alpha, ARM, MIPS, PowerPC

# User Level Synchronization— Operation Using this Primitive

Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
lockit:  li      R2,#1
         excl    R2,0(R1)      ;atomic exchange
         bnez    R2,lockit     ;already locked?
```

What about in a multicore processor with cache coherency?

- ➡ Want to spin on a cache copy to avoid keeping the memory busy
- ➡ Likely to get cache hits for such variables

Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):

```
try:
lockit: lw      R3,0(R1)      ;load var
         bnez    R3,lockit     ;not free=>spin
         excl    R2,0(R1)      ;atomic exchange
         bnez    R2,try        ;already locked?
```

# Memory Consistency Models

What is consistency? **When** must a processor see the new value? e.g. consider:

P1: A = 0;

P2: B = 0;

.....  
A = 1;

.....  
B = 1;

L1: if (B == 0) ...

L2: if (A == 0) ...

Impossible for both if statements L1 & L2 to be true?

What if write invalidate is delayed & processor continues?

Different processors have different *memory consistency models*

**Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above

SC: delay all memory accesses until all invalidates done



# Memory Consistency Models

- Weak consistency can be faster than sequential consistency
- Several processors provide fence instructions to enforce sequential consistency when an instruction stream passes such a point. Expensive!
- Not really an issue for most programs if they are explicitly *synchronized*

■ A program is synchronized if all access to shared data are ordered by synchronization operations

write (x)

...

release (s) {*unlock*}

...

acquire (s) {*lock*}

...

read(x)

- Only those programs willing to be nondeterministic are not synchronized: programs with “*data races*”
- There are several variants of weak consistency, characterized by attitude towards: RAR, WAR, RAW, WAW to different addresses

# Large-Scale Shared-Memory Multiprocessors: Directory-based cache coherency protocols

Bus inevitably becomes a bottleneck when many processors are used

- ➡ Use a more general interconnection network
- ➡ So snooping does not work

 DRAM memory is also distributed

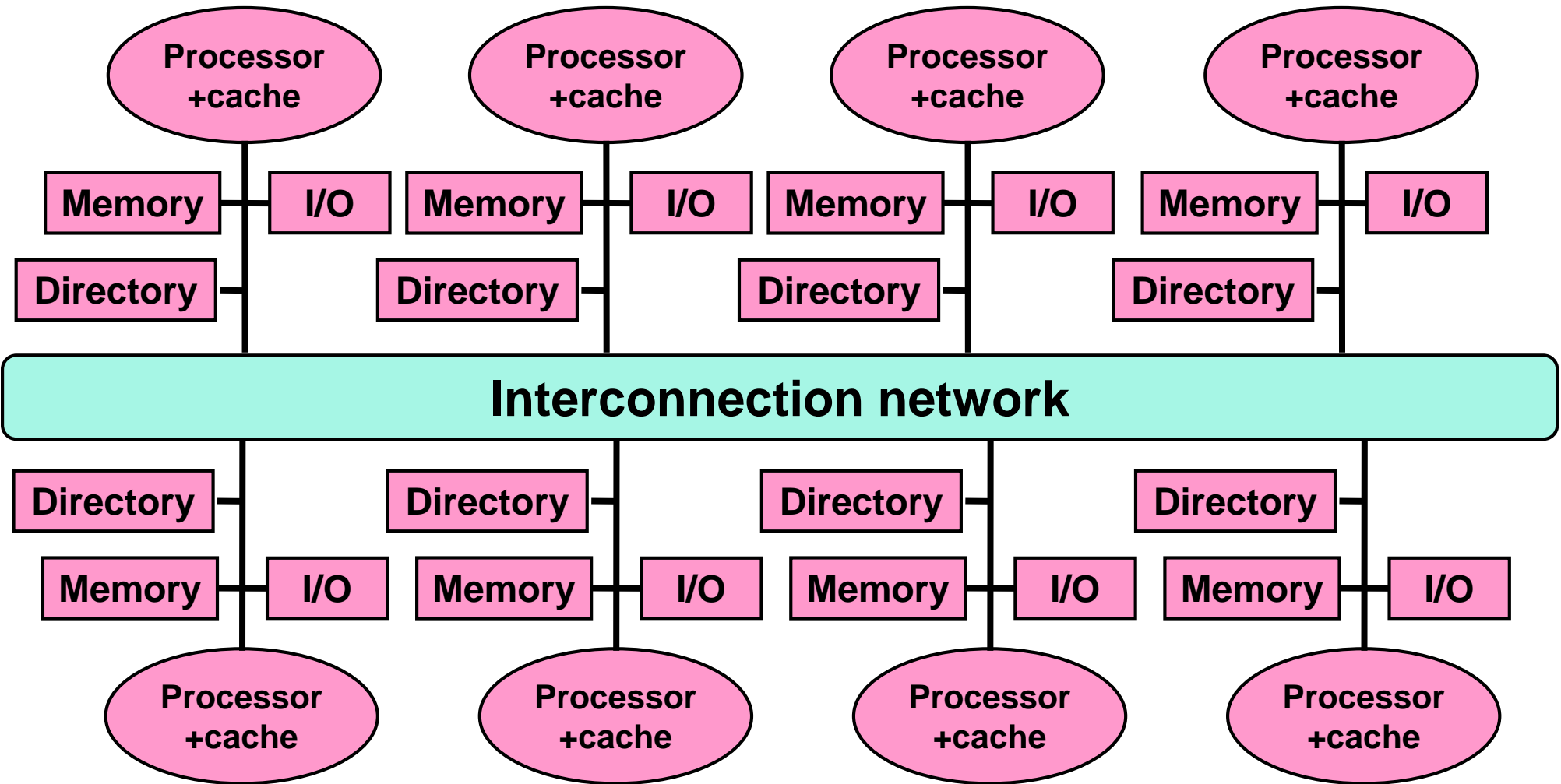
- ➡ Each node allocates space from local DRAM
- ➡ Copies of remote data are made in cache

 Major design issues:

- ➡ How to find and represent the “directory” of each line?
- ➡ How to find a copy of a line?

- ▶ Separate Memory per Processor
- ▶ Local or Remote access via memory controller
- ▶ Directory per cache that tracks state of every block in every cache
  - ▶ Which caches have a copies of block, dirty vs. clean, ...
- ▶ Info per memory block vs. per cache block?
  - ▶ PLUS: In memory => simpler protocol (centralized/one location)
  - ▶ MINUS: In memory => directory is  $f(\text{memory size})$  vs.  $f(\text{cache size})$
- ▶ How do we prevent the directory being a bottleneck?  
distribute directory entries with memory, each keeping track of which cores have copies of their blocks

# Distributed Directory MPs



## Similar to Snoopy Protocol: Three states

- ➡ Shared:  $\geq 1$  processors have data, memory up-to-date
- ➡ Uncached (no processor has it; not valid in any cache)
- ➡ Exclusive: 1 processor (**owner**) has data;  
memory out-of-date

➡ In addition to cache state, must track which processors have data when in the shared state (for example bit vector, 1 if processor has copy)

## ➡ Keep it simple(r):

- ➡ Writes to non-exclusive data  
=> write miss
- ➡ Processor blocks until access completes
- ➡ We will assume that messages are received and acted upon in order sent
  - ➡ This is not necessarily the case in general

# Directory Protocol

➤ No bus and don't want to broadcast:

- interconnect no longer single arbitration point
- all messages have explicit responses

➤ Terms: typically 3 processors involved

- **Local node** where a request originates
- **Home node** where the memory location of an address resides
- **Remote node** has a copy of a cache block, whether exclusive or shared

➤ Example messages on next slide:

P = processor number, A = address

# Directory Protocol Messages

Message type	Source	Destination	Msg Content
Read miss	Local cache	Home directory	P, A
➡ Processor <i>P</i> reads data at address <i>A</i> ; make <i>P</i> a read sharer and arrange to send data back			
Write miss	Local cache	Home directory	P, A
➡ Processor <i>P</i> writes data at address <i>A</i> ; make <i>P</i> the exclusive owner and arrange to send data back			
Invalidate	Home directory	Remote caches	A
➡ Invalidate a shared copy at address <i>A</i> .			
Fetch	Home directory	Remote cache	A
➡ Fetch the block at address <i>A</i> and send it to its home directory			
Fetch/Invalidate	Home directory	Remote cache	A
➡ Fetch the block at address <i>A</i> and send it to its home directory; invalidate the block in the cache			
Data value reply	Home directory	Local cache	Data
➡ Return a data value from the home memory (read miss response)			
Data write-back	Remote cache	Home directory	A, Data
➡ Write-back a data value for address <i>A</i> (invalidate response)			

# State Transition Diagram for an Individual Cache Block in a Directory Based System

- ▶ States identical to snoopy case; transactions very similar.
- ▶ Transitions caused by read misses, write misses, invalidates, data fetch requests
- ▶ Generates read miss & write miss msg to home directory.
- ▶ Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests.
- ▶ Note: on a write, a cache block is bigger, so need to read the full cache block



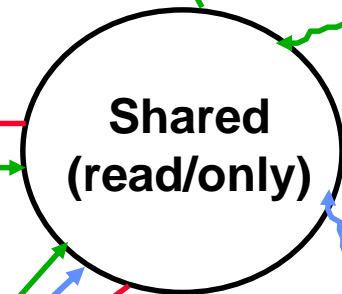
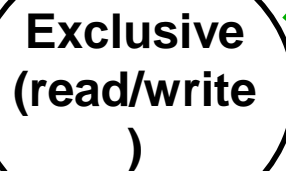
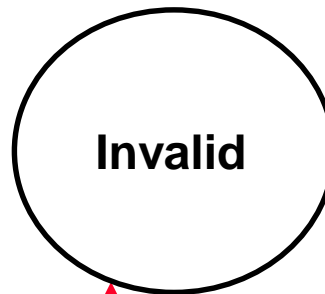
# CPU -Cache State Machine

State machine for CPU requests for each memory block

Invalid state if in memory

**Fetch/Invalidate**  
send Data Write Back message to home directory

**CPU read hit**  
**CPU write hit**



**Invalidate**

**CPU Read**

Send Read Miss message

**CPU Write:**

Send Write Miss msg to h.d.

**CPU Write:** Send Write Miss message to home directory

**Fetch:** send Data Write Back message to home directory

**CPU read miss:** send Data Write Back message and read miss to home directory

**CPU write miss:** send Data Write Back message and Write Miss to home directory

**CPU Read hit**

**CPU read miss:** Send Read Miss

**CPU Write:** Send Write Miss message to home directory

**Fetch:** send Data Write Back message to home directory

**CPU read miss:** send Data Write Back message and read miss to home directory

**CPU write miss:** send Data Write Back message and Write Miss to home directory

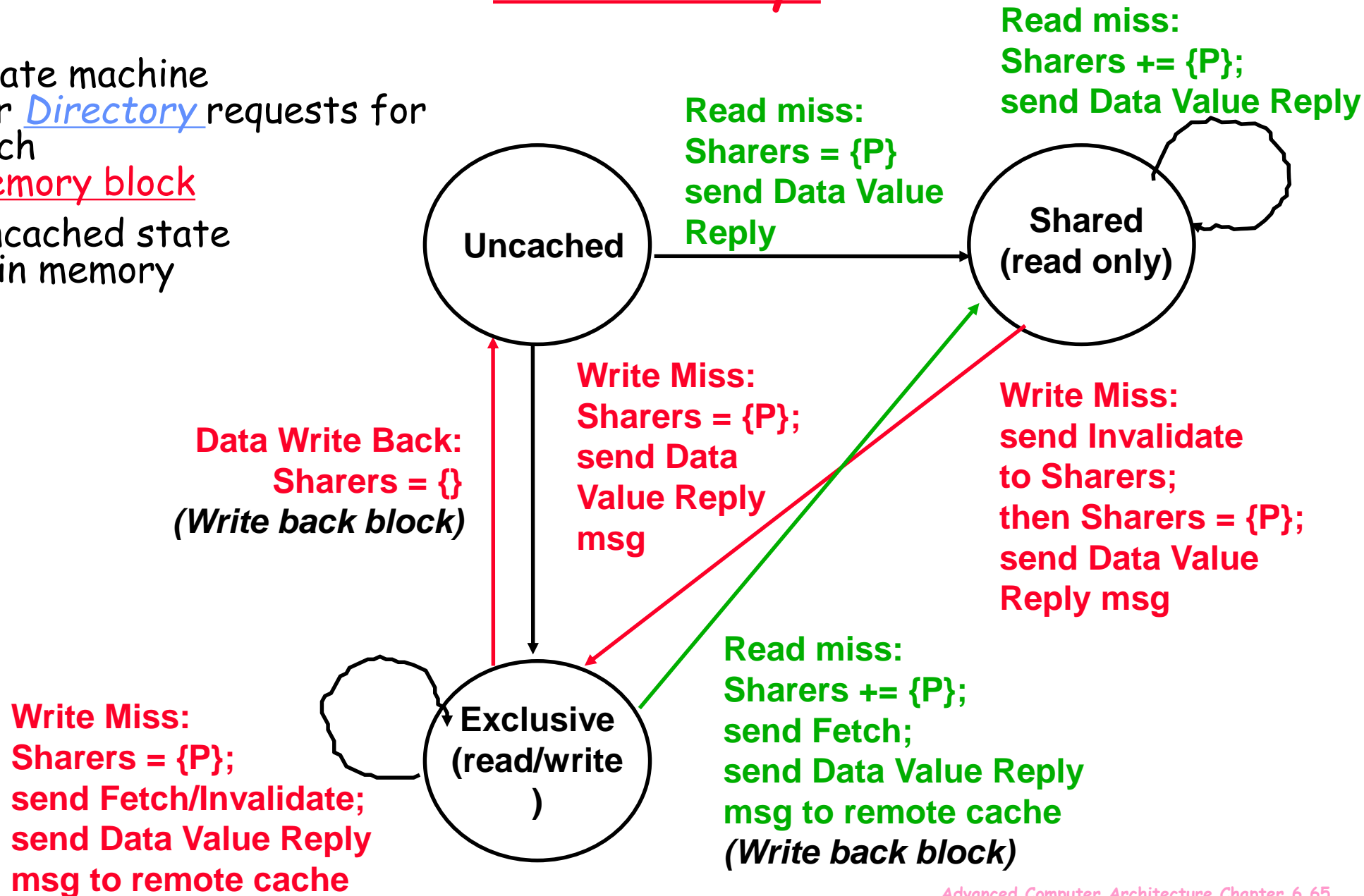
# State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send msgs to satisfy requests
- Tracks all copies of memory block.
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

# Directory State Machine

State machine for Directory requests for each memory block

Uncached state if in memory



# Example Directory Protocol

➤ Message sent to directory causes two actions:

- Update the directory
- More messages to satisfy request

➤ Block is in **Uncached** state: the copy in memory is the current value; only possible requests for that block are:

- **Read miss**: requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.
- **Write miss**: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

➤ Block is **Shared** => the memory value is up-to-date:

- **Read miss**: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
- **Write miss**: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

# Example Directory Protocol

Block is **Exclusive**: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:

- **Read miss**: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
- **Data write-back**: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
- **Write miss**: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

# Example

**Processor 1    Processor 2    Interconnect    Directory    Memory**

	<i>P1</i>			<i>P2</i>			<i>Bus</i>			<i>Directory</i>				<i>Memor</i>
<i>step</i>	<i>State</i>	<i>Addr</i>	<i>Value</i>	<i>State</i>	<i>Addr</i>	<i>Value</i>	<i>Action</i>	<i>Proc.</i>	<i>Addr</i>	<i>Value</i>	<i>Addr</i>	<i>State</i>	<i>{Procs}</i>	<i>Value</i>
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

**A1 and A2 map to the same cache block**

# Example

## Processor 1    Processor 2    Interconnect    Directory    Memory

	<i>P1</i>			<i>P2</i>			<i>Bus</i>			<i>Directory</i>				<i>Memor</i>
<i>step</i>	<i>State</i>	<i>Addr</i>	<i>Value</i>	<i>State</i>	<i>Addr</i>	<i>Value</i>	<i>Action</i>	<i>Proc.</i>	<i>Addr</i>	<i>Value</i>	<i>Addr</i>	<i>State</i>	<i>{Procs}</i>	<i>Value</i>
P1: Write 10 to A1							<i>WrMs</i>	P1	A1		<i>A1</i>	<i>Ex</i>	<i>{P1}</i>	
	<i>Excl.</i>	<i>A1</i>	<i>10</i>				<i>DaRp</i>	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

**A1 and A2 map to the same cache block**

# Example

Processor 1    Processor 2    Interconnect    Directory    Memory

	P1			P2			Bus			Directory				Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							<u>WrMs</u>	P1	A1		<u>A1</u>	<u>Ex</u>	<u>{P1}</u>	
	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>DaRp</u>	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block



# Example

## Processor 1    Processor 2    Interconnect    Directory    Memory

	P1			P2			Bus			Directory				Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

**A1 and A2 map to the same cache block**

# Example

## Processor 1    Processor 2    Interconnect    Directory    Memory

	P1			P2			Bus			Directory				Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							<u>WrMs</u>	P1	A1		<u>A1</u>	<u>Ex</u>	<u>{P1}</u>	
	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>DaRp</u>	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				<u>Shar.</u>	<u>A1</u>		<u>RdMs</u>	P2	A1					
	<u>Shar.</u>	A1	10				<u>Ftch</u>	P1	A1	10	<u>A1</u>			<u>10</u>
				Shar.	A1	<u>10</u>	<u>DaRp</u>	P2	A1	10	A1	<u>Shar.</u>	<u>P1,P2</u>	10
P2: Write 20 to A1				Excl.	A1	<u>20</u>	<u>WrMs</u>	P2	A1					10
	<u>Inv.</u>						<u>Inval.</u>	P1	A1		A1	<u>Excl.</u>	<u>{P2}</u>	10
P2: Write 40 to A2														

A1 and A2 map to the same cache block

# Example

## Processor 1    Processor 2    Interconnect    Directory    Memory

	P1			P2			Bus			Directory				Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							<u>WrMs</u>	P1	A1		<u>A1</u>	<u>Ex</u>	<u>{P1}</u>	
	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>DaRp</u>	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				<u>Shar.</u>	<u>A1</u>		<u>RdMs</u>	P2	A1					
	<u>Shar.</u>	A1	10				<u>Ftch</u>	P1	A1	10	<u>A1</u>			<u>10</u>
				Shar.	A1	<u>10</u>	<u>DaRp</u>	P2	A1	10	A1	<u>Shar.</u>	<u>P1,P2}</u>	10
P2: Write 20 to A1				Excl.	A1	<u>20</u>	<u>WrMs</u>	P2	A1					10
	<u>Inv.</u>						<u>Inval.</u>	P1	A1		A1	<u>Excl.</u>	<u>{P2}</u>	10
P2: Write 40 to A2							<u>WrMs</u>	P2	A2		<u>A2</u>	<u>Excl.</u>	<u>{P2}</u>	0
							<u>WrBk</u>	P2	A1	20	<u>A1</u>	<u>Unca.</u>	<u>{}</u>	<u>20</u>
				Excl.	<u>A2</u>	<u>40</u>	<u>DaRp</u>	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block

# Implementing a Directory

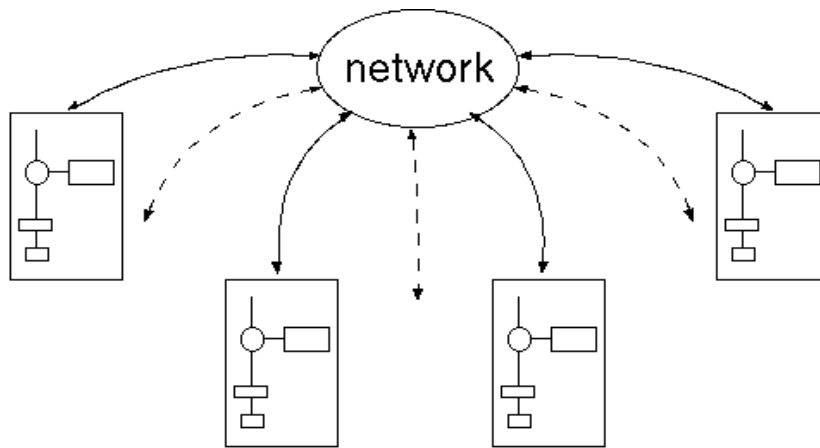
✚ We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see H&P 3<sup>rd</sup> ed Appendix E)

✚ Optimizations:

- ➡ read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor

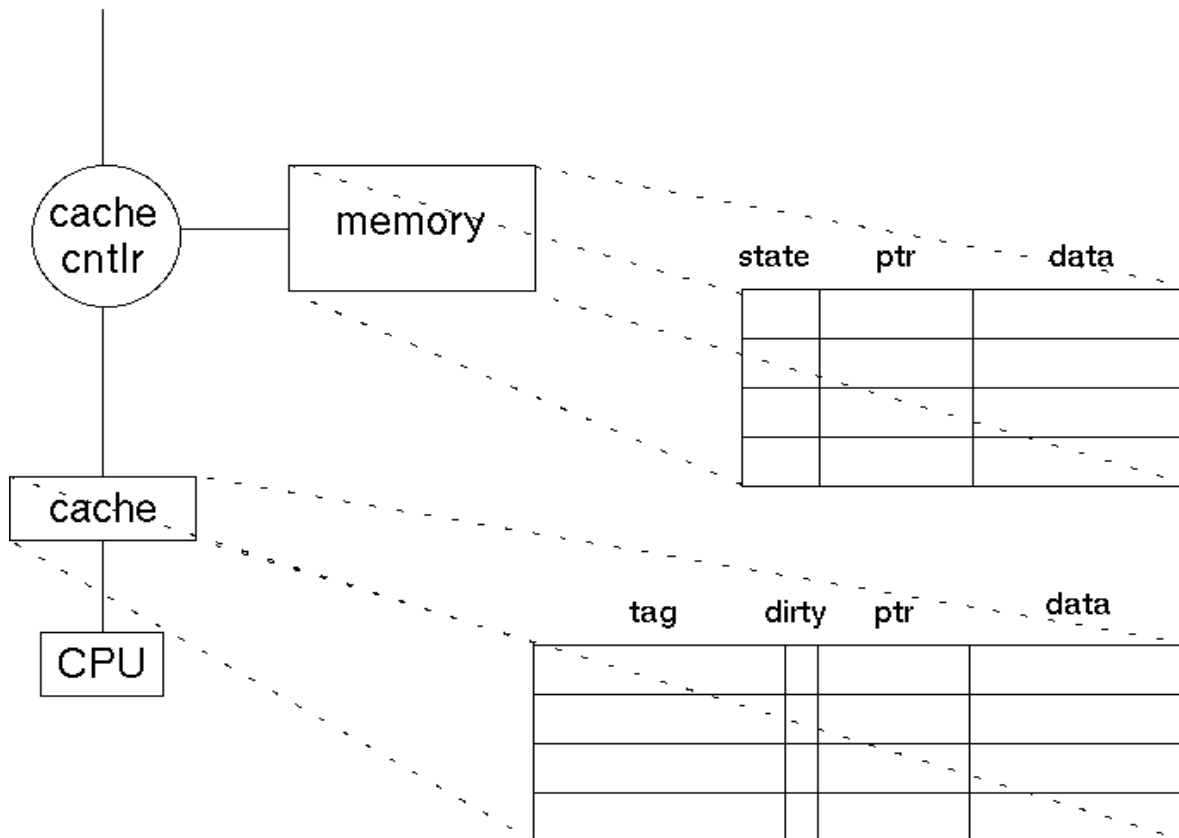
- ▶ Caches contain all information on state of cached memory blocks
- ▶ Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- ▶ Directory has extra data structure to keep track of state of all cache blocks
- ▶ Distributing directory => scalable shared address multiprocessor  
=> Cache coherent, Non uniform memory access

# Case study: Sun's S3MP



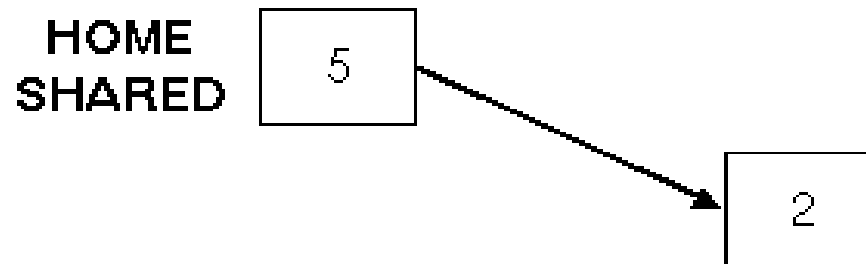
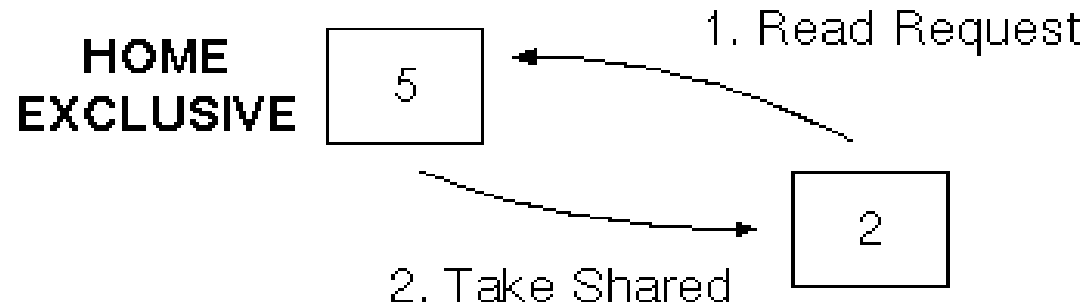
## Protocol Basics

- S3.MP uses distributed singly-linked sharing lists, with static homes
- Each line has a “home” node, which stores the root of the directory
- Requests are sent to the home node
- Home either has a copy of the line, or knows a node which does



# S3MP: Read Requests

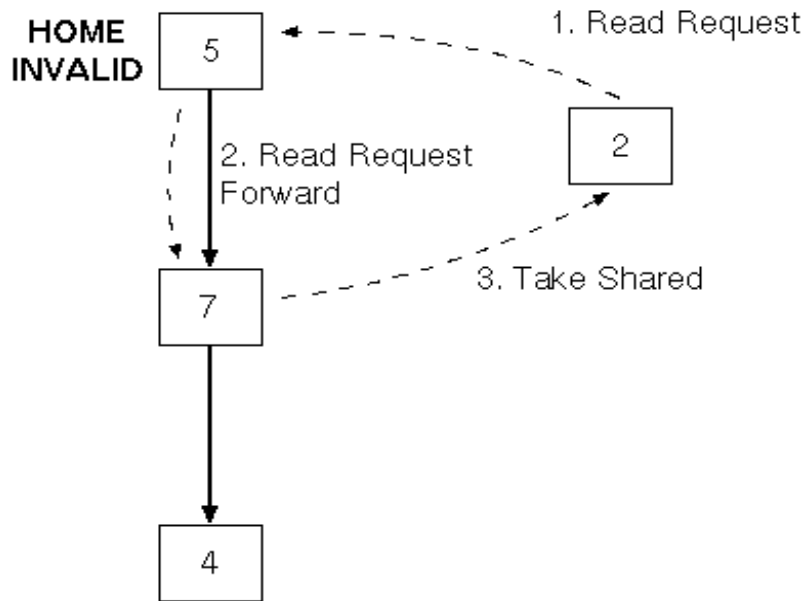
Simple case: initially only the home has the data:



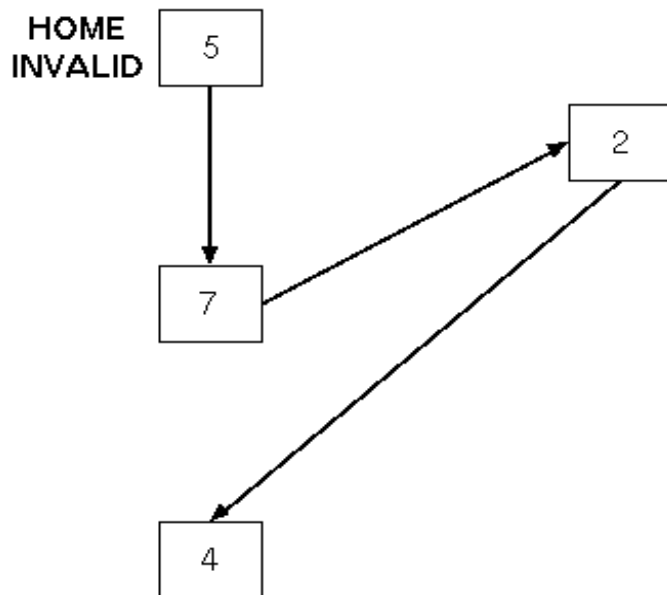
**Curved arrows show messages, bold straight arrows show pointers**

- **Home replies with the data, creating a sharing chain containing just the reader**

# S3MP: Read Requests - remote



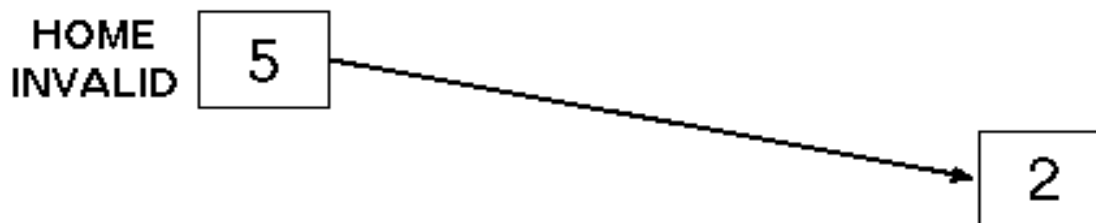
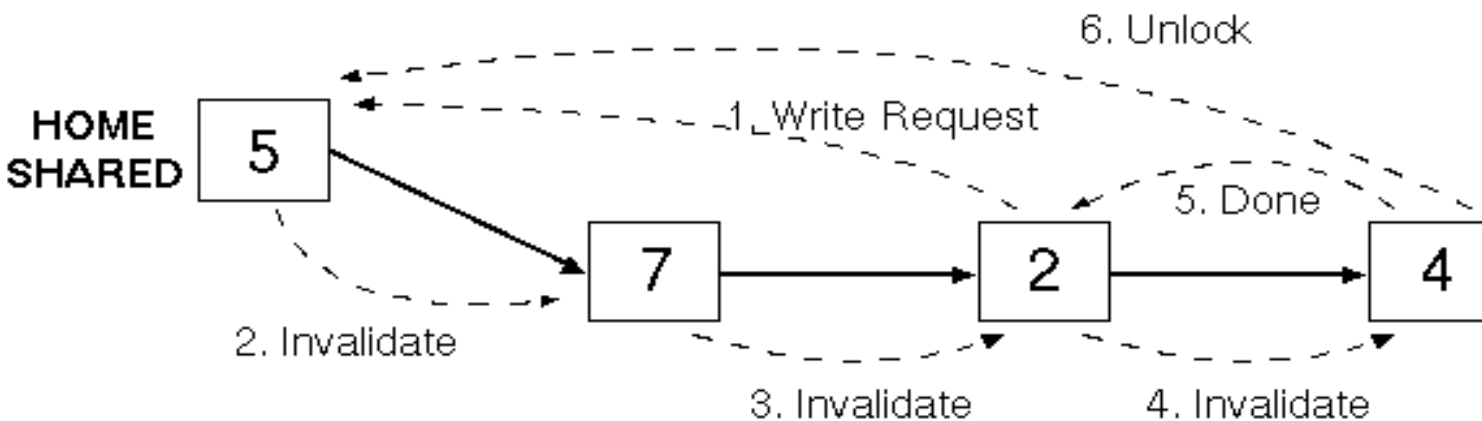
More interesting case: some other processor has the data



Home passes request to first processor in chain, adding requester into the sharing list

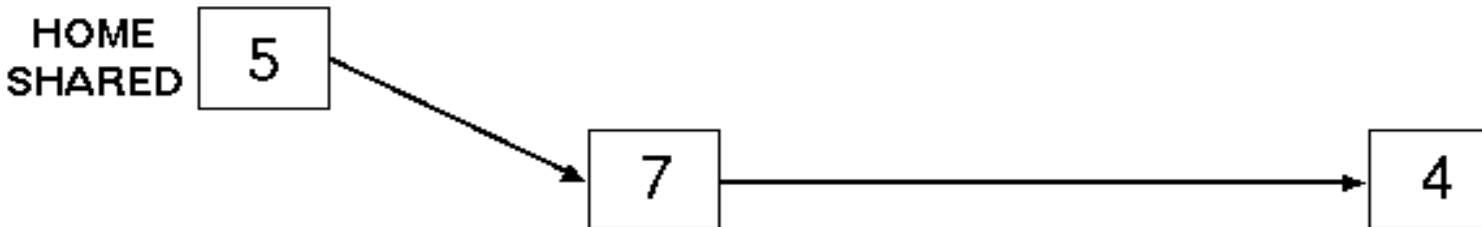
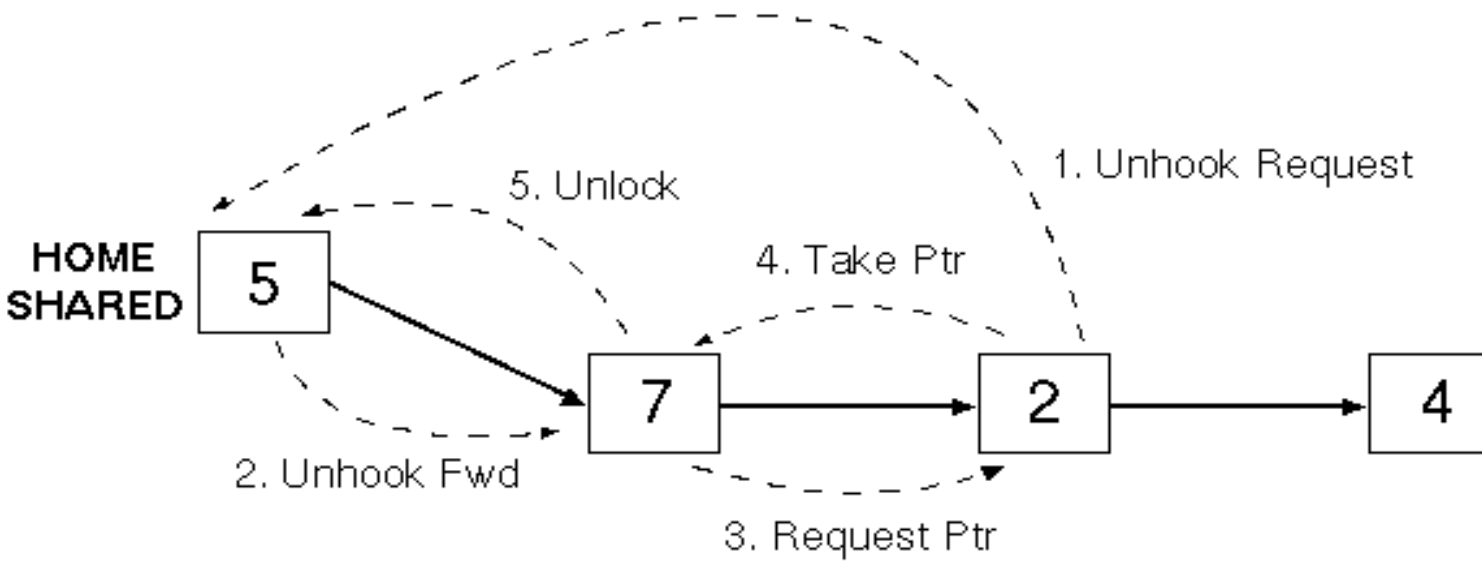


# S3MP - Writes



- ✎ If the line is exclusive (i.e. dirty bit is set) no message is required
- ✎ Else send a write-request to the home
  - ➡ Home sends an invalidation message down the chain
  - ➡ Each copy is invalidated (other than that of the requester)
  - ➡ Final node in chain acknowledges the requester and the home
- ✎ Chain is locked for the duration of the invalidation

# S3MP - Replacements



- When a read or write requires a line to be copied into the cache from another node, an existing line may need to be replaced
- Must remove it from the sharing list
- Must not lose last copy of the line

 How does a CPU find a valid copy of a specified address's data?

1. Translate virtual address to physical
2. Physical address includes bits which identify “home” node
3. Home node is where DRAM for this address resides
4. But current valid copy may not be there – may be in another CPU's cache
5. Home node holds pointer to sharing chain, so always knows where valid copy can be found

▶ S3MP's cache coherency protocol implements strong consistency

➡ Many recent designs implement a weaker consistency model...

▶ S3MP uses a singly-linked sharing chain

➡ Widely-shared data – long chains – long invalidations, nasty replacements

➡ “Widely shared data is rare”

▶ In real life:

➡ IEEE Scalable Coherent Interconnect (SCI): doubly-linked sharing list

➡ SGI Origin 2000: 64-bit vector sharing list

● Origin 2000 systems were delivered with 256 CPUs

➡ Sun E10000: hybrid multiple buses for invalidations, separate switched network for data transfers

● Many E10000s sold, often with 64 CPUs

## COMA: cache-only memory architecture

- Data migrates into local DRAM of CPUs where it is being used
- Handles very large working sets cleanly
- Replacement from DRAM is messy: have to make sure *someone* still has a copy
- Scope for interesting OS/architecture hybrid
- System slows down if total memory requirement approaches RAM available, since space for replication is reduced

## Examples: DDM, KSR-1/2

# Clustered architectures

- **Idea:** systems linked by network connected via I/O bus
  - Eg PC cluster with Myrinet or Quadrics interconnection network
  - Eg Quadrics+PCI-Express: 900MB/s (500MB/s for 2KB messages), 3us latency
- **Idea:** systems linked by network connected via memory interface
  - Eg Cray Seastar
- **Idea:** CPU/memory packages linked by network connecting main memory units
  - Eg SGI Origin, nVidia Tesla?
- **Idea:** CPUs share L2/L3 cache
  - Eg IBM Power4,5,6, Intel Core2, Nehalem, AMD Opteron, Phenom
- **Idea:** CPUs share L1 cache
- **Idea:** CPUs share registers, functional units
  - IBM Power5,6, PentiumIV(hyperthreading), Intel Atom, Alpha 21464/EV8, Cray/Tera MTA (multithreaded architecture), nVidia Tesla

- **Cunning Idea: do (almost) all the above at the same time**
- **Eg IBM SP/Power6: 2 CPUs/chip, 2-way SMT, “semi-shared” 4M/core L2, shared 32MB L3, multichip module packages/links 4 chips/node, with L3 and DRAM for each CPU on same board, with high-speed (ccNUMA) link to other nodes – assembled into a cluster of 8-way nodes linked by proprietary network**

## Which cache should the cache controller control?

- ✎ L1 cache is already very busy with CPU traffic
- ✎ L2 cache also very busy...
- ✎ L3 cache doesn't always have the current value for a cache line
  - Although L1 cache is often write-through, L2 is normally write-back
  - Some data may bypass L3 (and perhaps L2) cache (eg when stream-prefetched, or if L3 is managed as a victim cache)
- Eg in IBM Power4 and others, cache controller manages L2 cache – all external invalidations/requests
- L3 cache improves access to DRAM for accesses both from CPU and from network (essentially a *victim* cache)

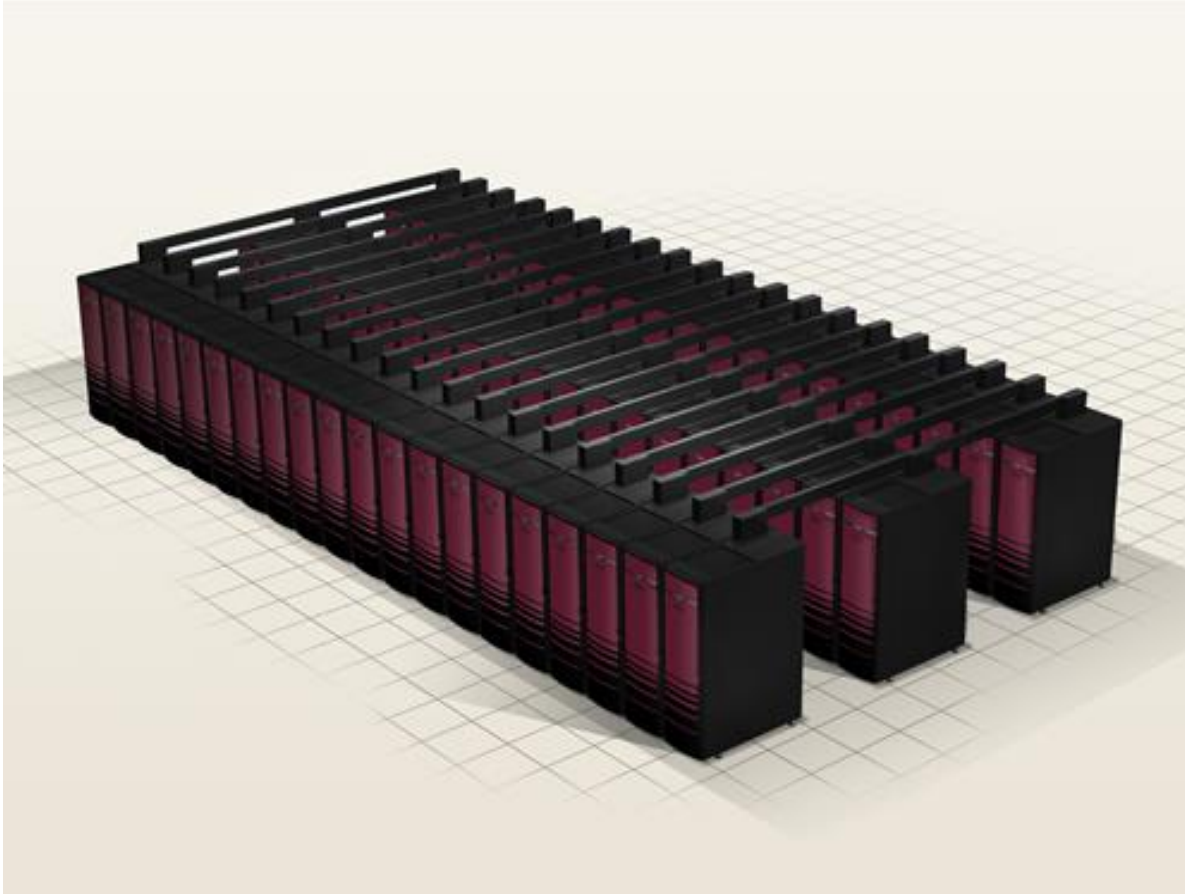
# Summary and Conclusions

- ❖ Caches can be used to form the basis of a parallel computer supporting a single, shared address space
- ❖ Bus-based multiprocessors do not scale well due to broadcasts and the need for each cache controller to snoop all the traffic
- ❖ Larger-scale shared-memory multiprocessors require a cache *directory* to track where copies are held
  - ❖ Hierarchical and hybrid schemes can work, with snooping within a cluster of cores, and a directory scheme at the cluster level
- ❖ ccNUMA: each node has a fragment of the system's DRAM, every physical address has a unique "home" node
- ❖ COMA: each node (sometimes called a NUMA domain) has a fragment of the system's DRAM, but data is migrated between NUMA domains adaptively
- ❖ NUCA: cache is distributed, so access latency is non-uniform (and management may include dynamic/adaptive placement strategies)

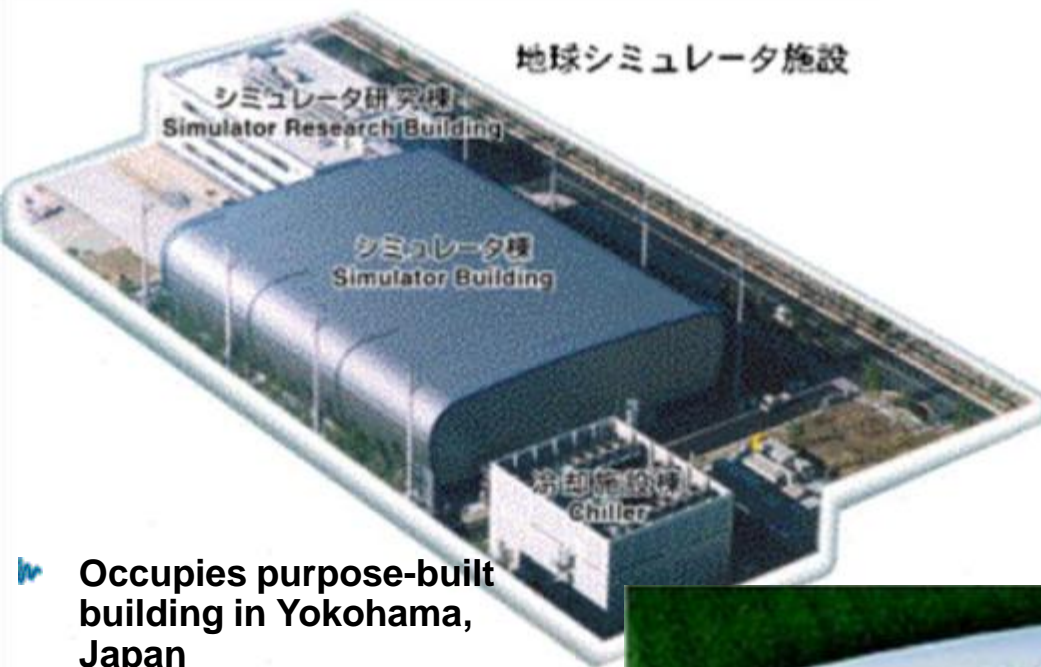


# **Additional slides for interest and context**

# HECToR – “High-End Computing Terascale Resource”



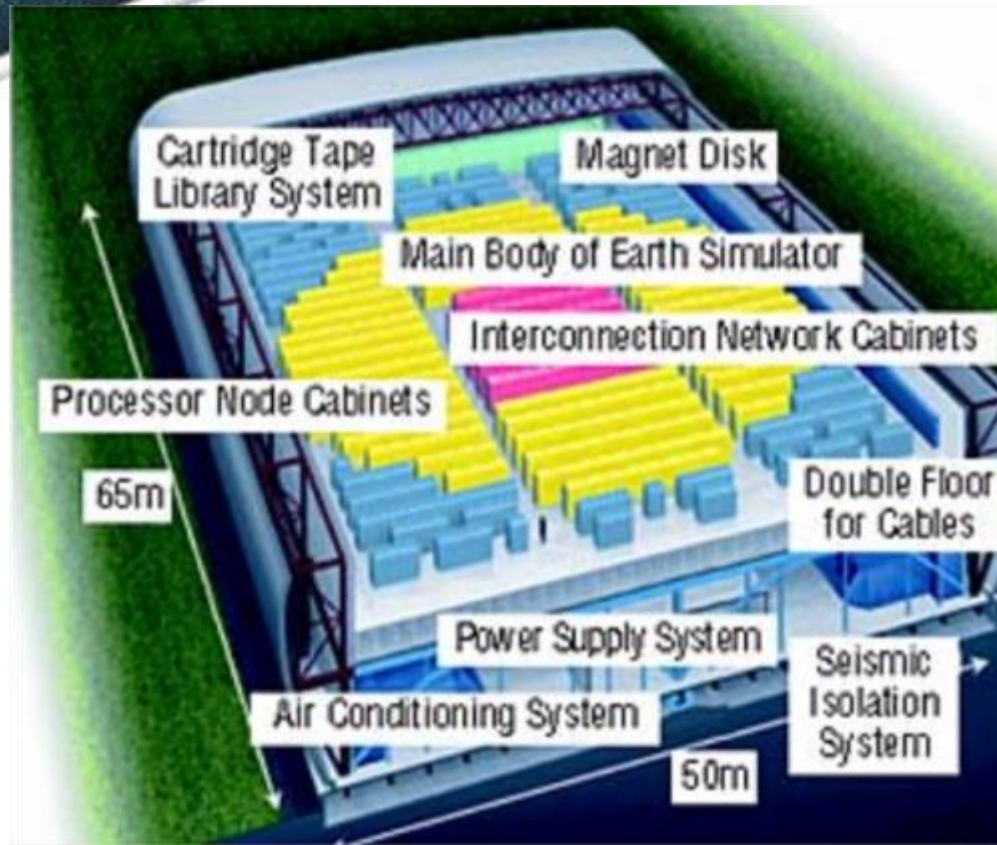
- Successor to HPCx, superseded by Archer
- Located at University of Edinburgh, operated on behalf of UK science community
- Six-year £133M contract, includes mid-life upgrades (60→250 TeraFLOPS)
- Cray XE6: 2816 nodes (704 blades) with two 16-core processors each, so 90,112 AMD “Interlagos” cores (2.3GHz)
- 16GB RAM per processor (32GB per node, 90TB in the machine)
- Cray “Gemini” interconnect (3D torus), directly connected to HyperTransport memory interface. Point-to-point bandwidth ~5GB/s, latency between two nodes 1-1.5us
- OS: Compute Node Linux kernel for compute nodes, full linux for I/O
- Large parallel file system, plus MAID (massive array of idle disks) backup, and tape library



# The “Earth Simulator”

- 5,120 (640 8-way nodes) 500 MHz NEC CPUs (later upgraded)
- 8 GFLOPS per CPU (41 TFLOPS total)
- 2 GB (4 512 MB FPLRAM modules) per CPU (10 TB total)
- shared memory inside the node
- 640 × 640 crossbar switch between the nodes
- 16 GB/s inter-node bandwidth
- 20 kVA power consumption per node

- Occupies purpose-built building in Yokohama, Japan
- Operational late 2001/early 2002
- Vector CPU using 0.15  $\mu\text{m}$  CMOS process, descendant of [NEC SX-5](#)
- Runs Super-UX OS



- CPU's housed in 320 cabinets, 2 8-CPU nodes per cabinet. The cabinets are organized in a ring around the interconnect, which is housed in another 65 cabinets
- Another layer of the circle is formed by disk array cabinets.
- The whole thing occupies a building 65 m long and 50 m wide



# IBM Bluegene/Q, Jülich Supercomputer Centre



- 28 racks (7 rows of 4 racks) - 28,672 nodes (458,752 cores)
- Rack: 2 midplanes of 16 nodeboards (16,384 cores)  
Nodeboard: 32 compute nodes  
Node: 16 cores
- Main memory: 448 TB
- Overall peak performance: 5.9 Petaflops
- Linpack: 5.0 Petaflops
- I/O Nodes: 248 (27x8 + 1x32) connected to 2 CISCO Switches
- IBM PowerPC® A2, 1.6 GHz, 16 cores + 16 GB SDRAM-DDR3 per node
- Networks  
5D Torus — 40 GBps; 2.5  $\mu$ sec latency (worst case)  
Collective network — part of the 5D Torus; collective logic operations supported  
Global Barrier/Interrupt — part of 5D Torus, PCIe x8 Gen2 based I/O  
1 GB Control Network — System Boot, Debug, Monitoring
- I/O Nodes (10 GbE)
- 16-way SMP processor; configurable in 8,16 or 32 I/O nodes per rack
- Node operating system: CNK, lightweight proprietary kernel

- 60 - 70 kW per rack (average)
- approx 1.7 MW in 2012 (compares to app. 4000 households)
- 90% water cooling (18-25° C, demineralized, closed circle); 10% air cooling  
Temperature: in: 18° C, out: 27° C
- 83 m<sup>2</sup> (+ 43 m<sup>2</sup> JUST Fileserver)
- Integrated 5D torus interconnection network
- Quad floating point unit (FPU) for 4-wide double precision FPU SIMD and 2-wide complex SIMD.
- “Perfect” prefetching for repeated memory reference patterns in arbitrarily long code segments.
- Multiversioning cache with transactional memory eliminates the need for locks.
- Speculative execution allows OpenMP threading with data dependencies.
- Atomic operations, pipelined at L2 with low latency even under high contention, provide a faster handoff for OpenMP work.
- A wake-up unit allows SMT threads to sleep while waiting for an event and avoids register-saving overhead.
- A 17th core manages OS related tasks thus reducing OS related noise.





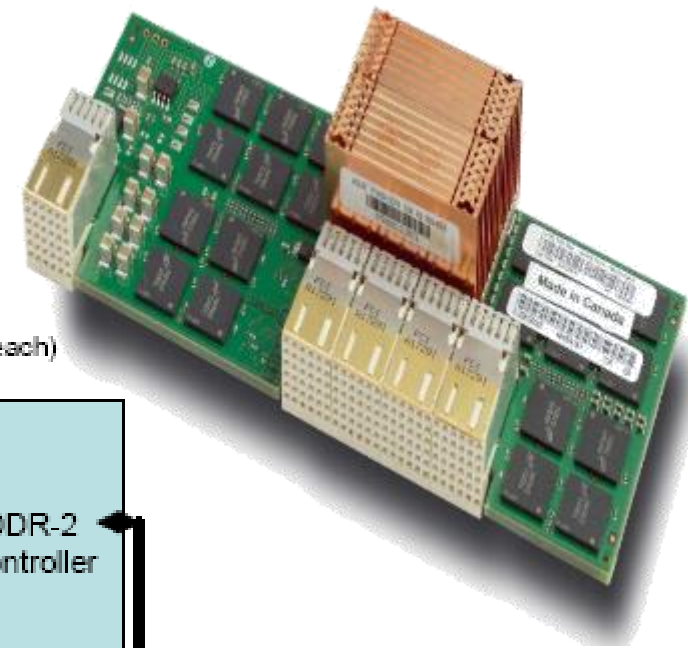
## Selected sample applications:

- "Gravoturbulent Planetesimal Formation: A key to understand planet formation"
- "Nuclear Lattice Simulations"
- "In Silico Exploration of Possible Routes to Prebiotic Peptide Synthesis by Ab Initio Metadynamics"
- "Charm physics on fine lattices with open boundaries"
- "Charm loop effects: decoupling and charmonium"
- "The pseudocritical line in the QCD phase diagram"
- "Hadronic corrections to the muon magnetic moment"
- "Hadron Scattering and Resonance Properties from Lattice QCD"
- "Data Assimilation for Improved Characterization of Fluxes Across Compartmental Interfaces"
- "Turbulent convection at very low Prandtl numbers"

1.7MW: burning 1 tonne of coal runs the machine for less than five hours



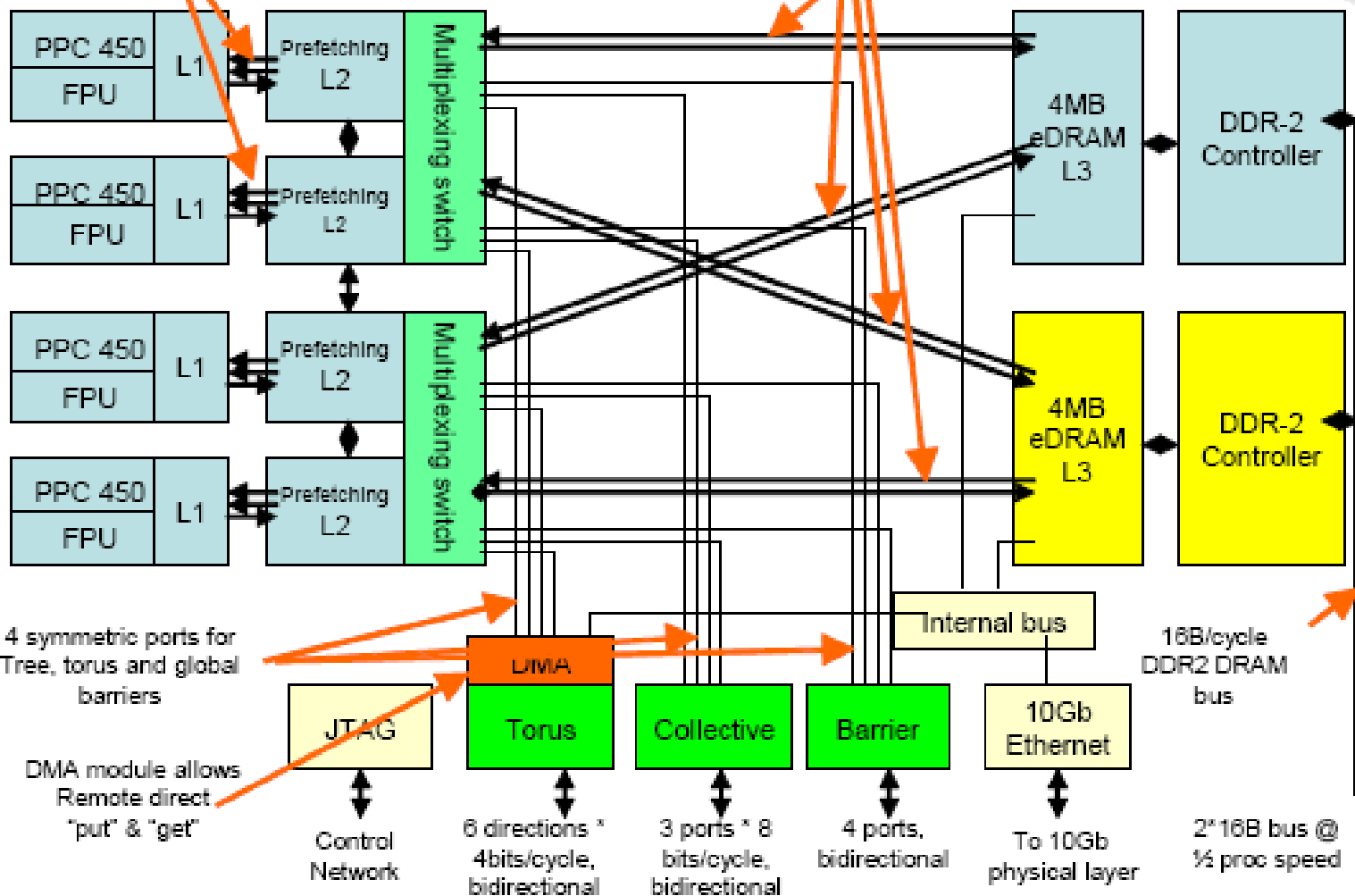
# Bluegene/P node ASIC



BlueGene/P node

Data read @ 8 B/cycle  
Data write @ 8 B/cycle  
Instruction @ 8 B/cycle

16B/cycle read (each), 16B/cycle write (each)





- 32 nodes ASICs + DRAM + interconnect
- High level of integration, thanks to IBM's system-on-chip technology (from embedded systems business)
- Extremely low power – 7.7W/core vs 51W/core for Cray XT4 (like HeCTOR)



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi [ <a href="#">/site/50623</a> ] China	<b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway [ <a href="#">/system/178764</a> ] NRCP	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou [ <a href="#">/site/50365</a> ] China	<b>Tianhe-2 (MilkyWay-2)</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P [ <a href="#">/system/177999</a> ] NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory [ <a href="#">/site/48553</a> ] United States	<b>Titan</b> - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x [ <a href="#">/system/177975</a> ] Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL [ <a href="#">/site/49763</a> ] United States	<b>Sequoia</b> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom [ <a href="#">/system/177556</a> ] IBM	1,572,864	17,173.2	20,132.7	7,890
5	DOE/SC/LBNL/NERSC [ <a href="#">/site/48429</a> ] United States	<b>Cori</b> - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect [ <a href="#">/system/178924</a> ] Cray Inc.	622,336	14,014.7	27,880.7	3,939
6	Joint Center for Advanced High Performance Computing [ <a href="#">/site/50673</a> ] Japan	<b>Oakforest-PACS</b> - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path [ <a href="#">/system/178932</a> ] Fujitsu	556,104	13,554.6	24,913.5	2,719
7	RIKEN Advanced Institute for Computational Science (AICS) [ <a href="#">/site/50313</a> ] Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect [ <a href="#">/system/177232</a> ] Fujitsu	705,024	10,510.0	11,280.4	12,660
8	Swiss National Supercomputing Centre (CSCS) [ <a href="#">/site/50422</a> ] Switzerland	<b>Piz Daint</b> - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 [ <a href="#">/system/177824</a> ] Cray Inc.	206,720	9,779.0	15,988.0	1,312
9	DOE/SC/Argonne National Laboratory [ <a href="#">/site/47347</a> ] United States	<b>Mira</b> - BlueGene/Q, Power BQC 16C 1.60GHz, Custom [ <a href="#">/system/177718</a> ] IBM	786,432	8,586.6	10,066.3	3,945
10	DOE/NNSA/LANL/SNL [ <a href="#">/site/50334</a> ] United States	<b>Trinity</b> - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect [ <a href="#">/system/178610</a> ] Cray Inc.	301,056	8,100.9	11,078.9	4,233
11	United Kingdom Meteorological Office [ <a href="#">/site/49064</a> ] United Kingdom	Cray XC40, Xeon E5-2695v4 18C 2.1GHz, Aries interconnect [ <a href="#">/system/178925</a> ] Cray Inc.	241,920	6,765.2	8,128.5	

## TOP500 List (Nov 2016)

- **Rmax and Rpeak values are in Gflops**
- **ranked by their performance on the [LINPACK Benchmark](#).**
- **“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”**

# What are parallel computers used for?

## QUINCY DATA CENTERS



COLOANDCLOUD.COM