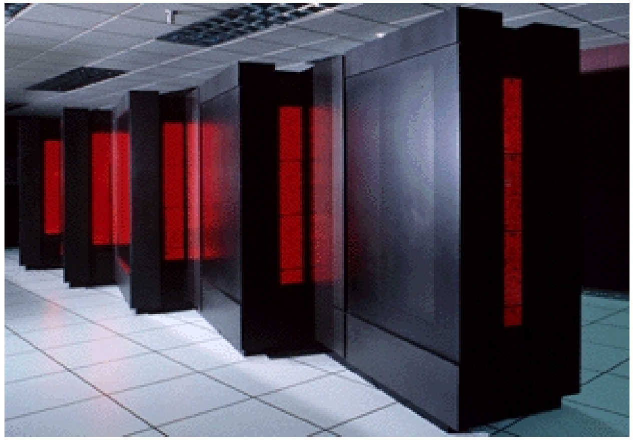
Background to Multicores

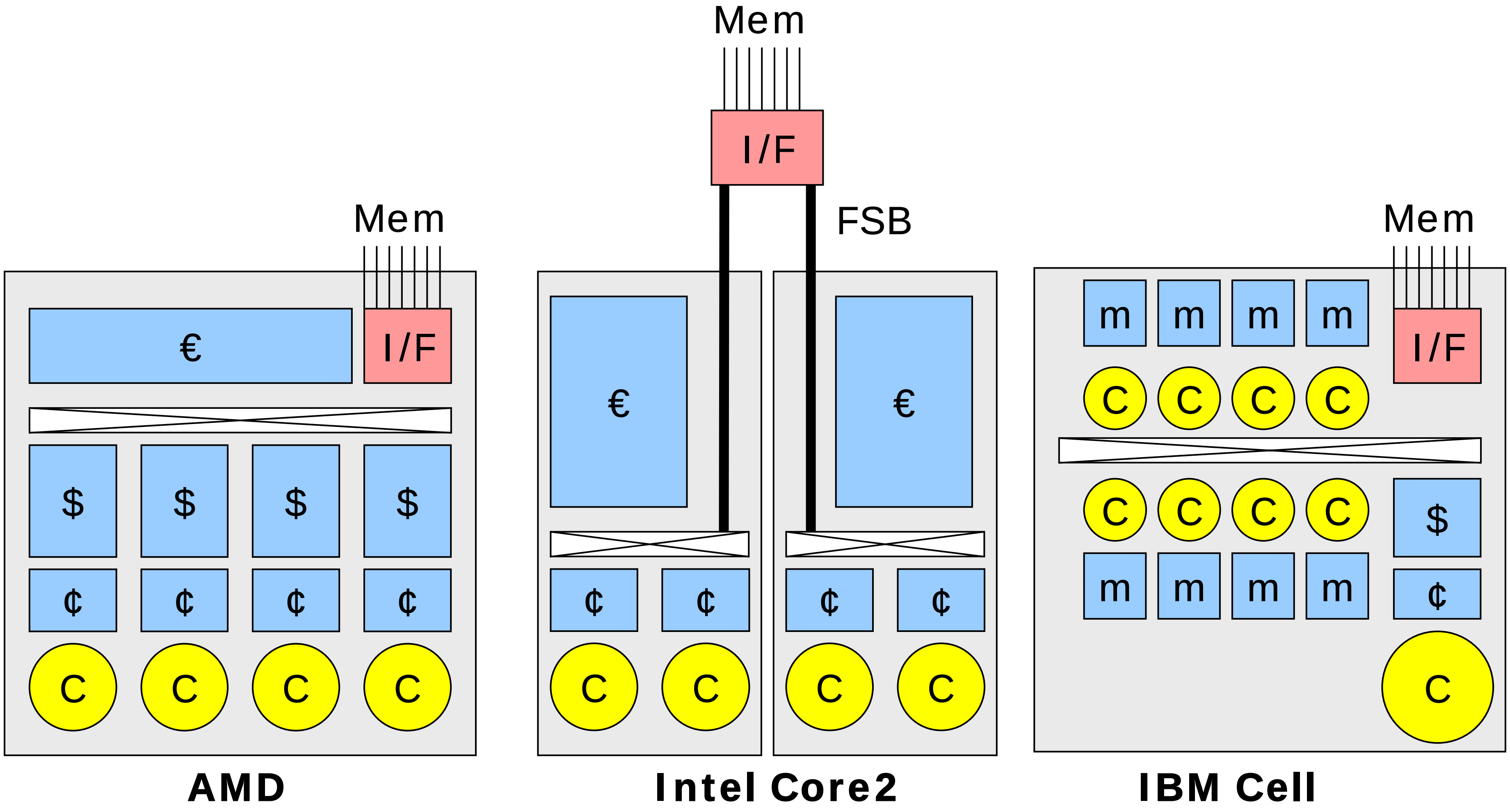
Erik Hagersten
Uppsala University

The era of the “supercomputer” multiprocessors

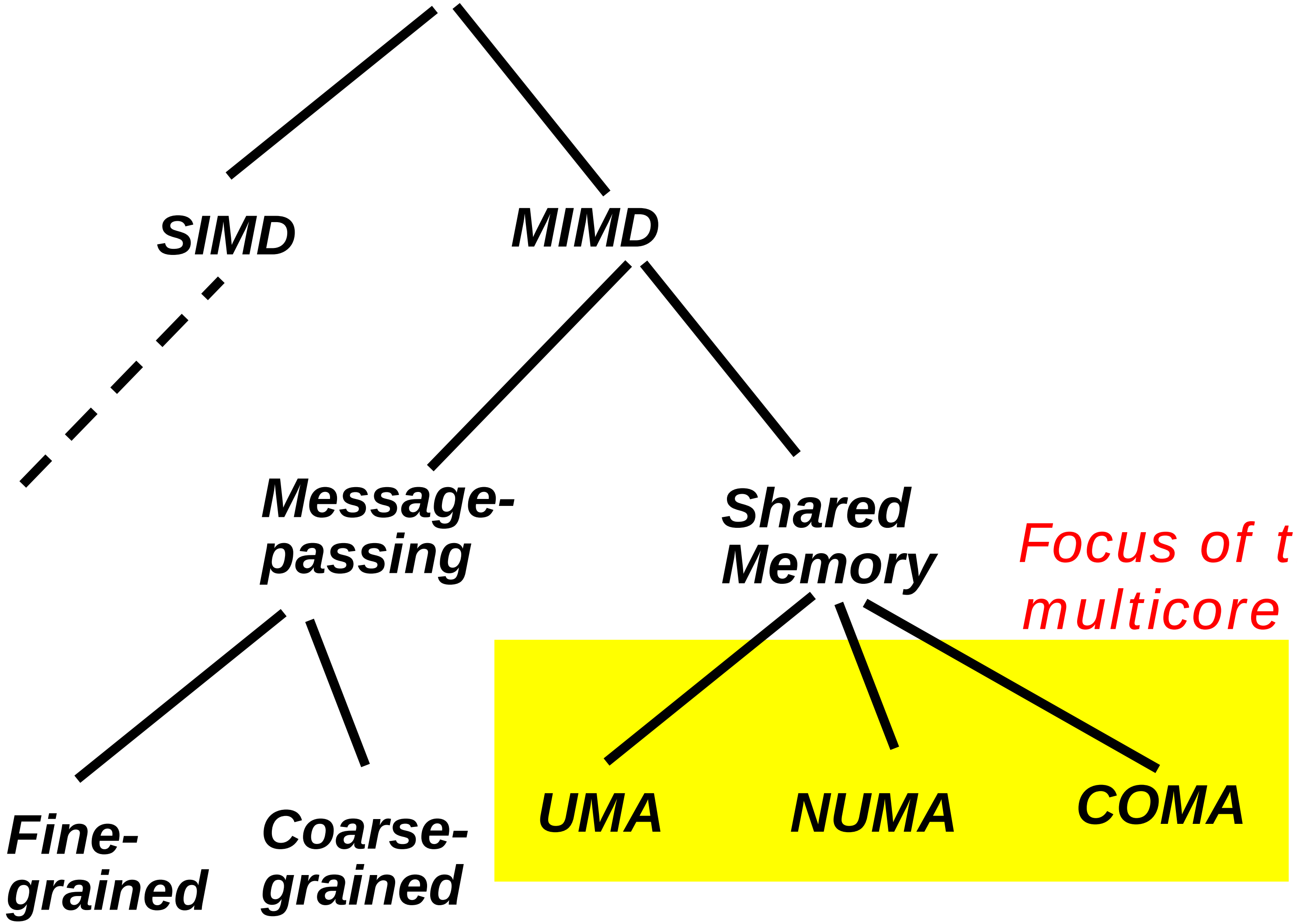
- The one with the most blinking lights wins
- The one with the strangest languages wins
- The niftier the better!



Multicore: Who has not got one?



Taxomy for Architectures [Flynn]



Focus of this multicore lecture

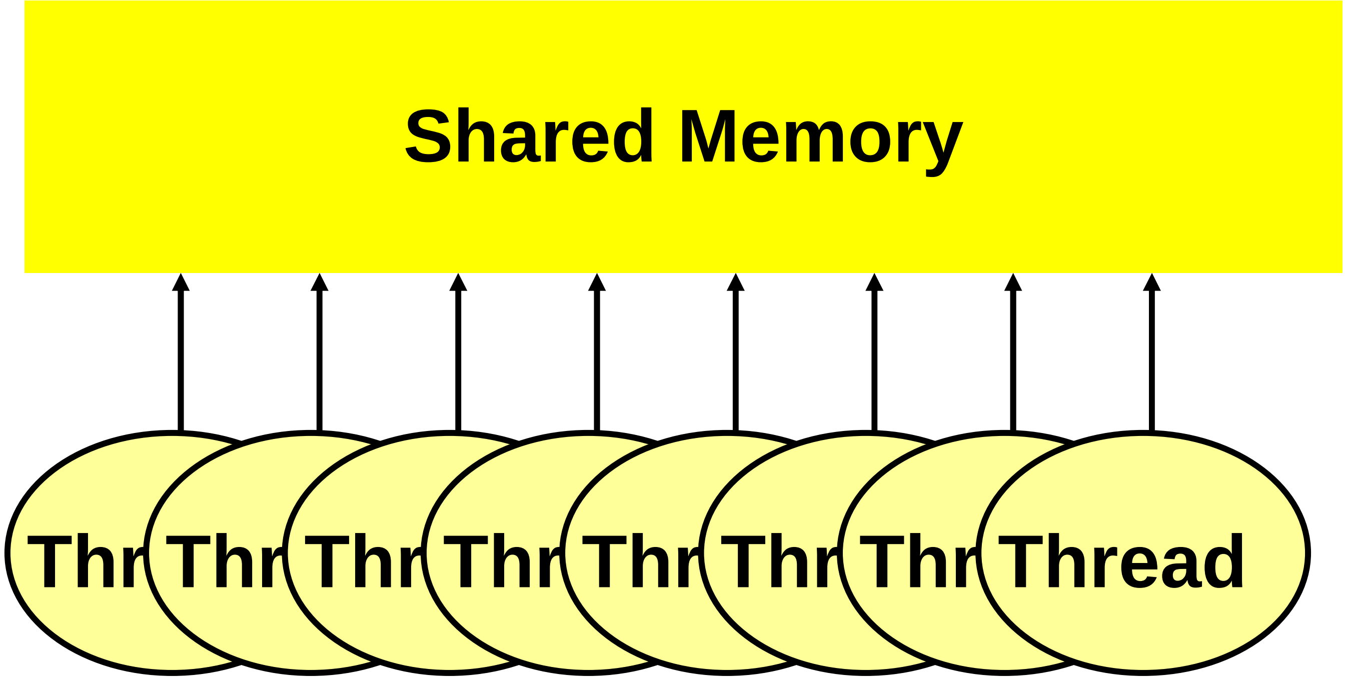
Outline

- Recap
  - Caches and Coherence
- More insight
  - Coherence
  - Memory models
- Example
  - Leveraging coherence properties for efficient synchronization

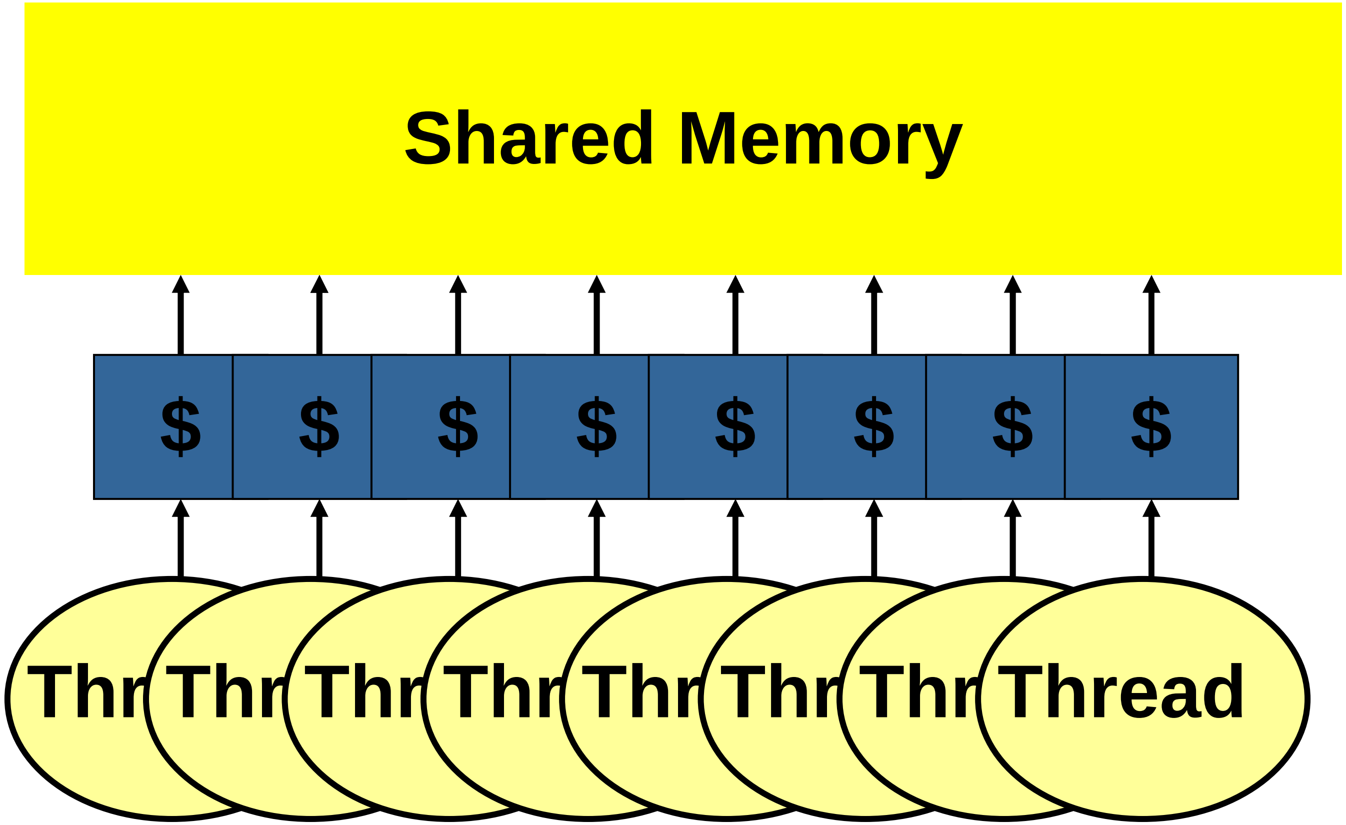
Coherent Shared Memory

Erik Hagersten
Uppsala University

The Shared Memory Programming Model (Pthreads/OpenMP, …)

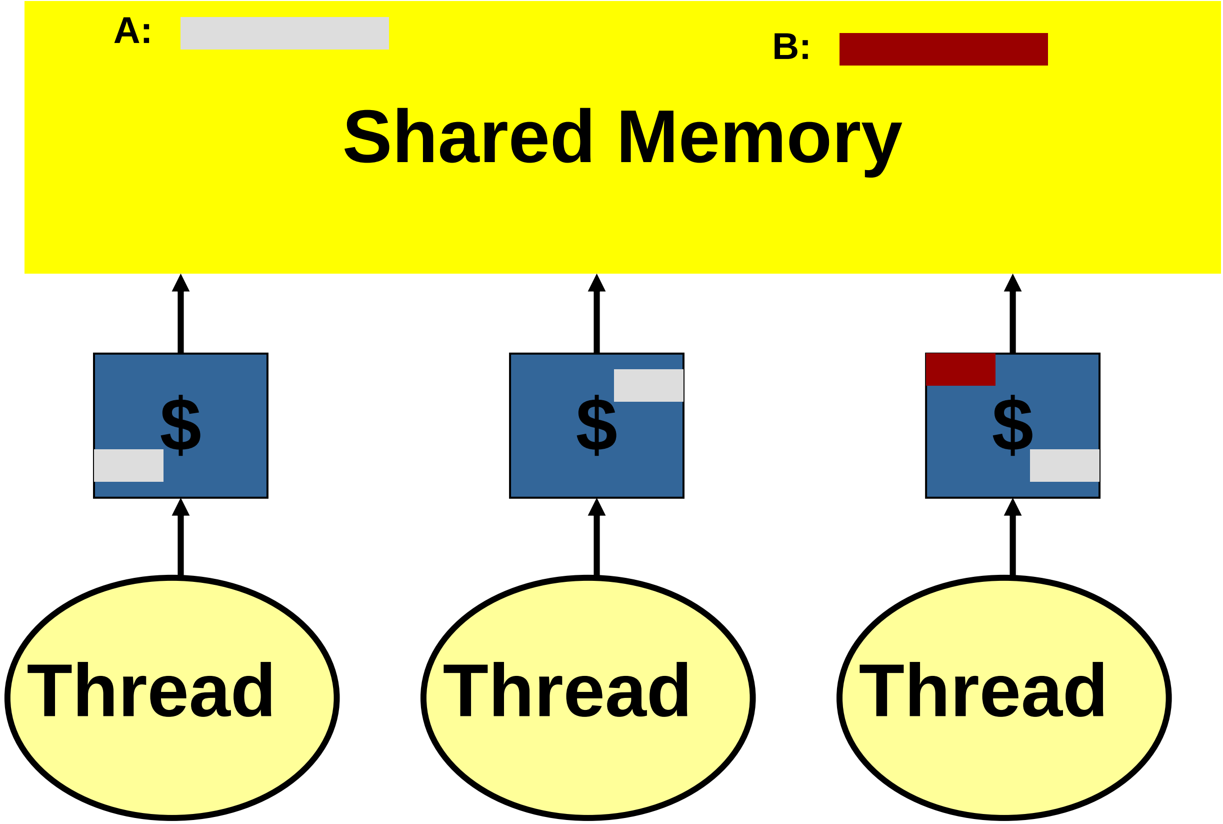


Adding Caches: More Concurrency



UPPSALA UNIVERSITET

Automatic Replication of Data



Read A
Read A
…
…
Read A

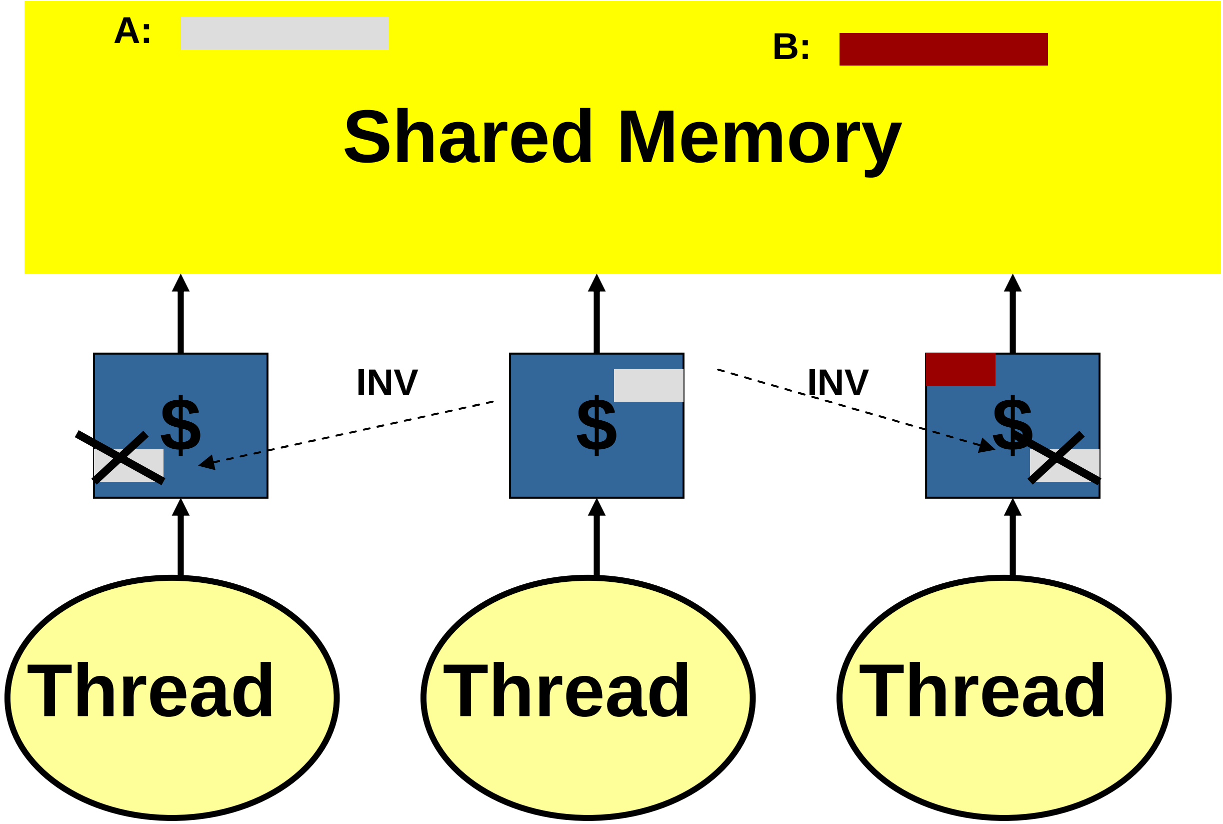
...
Read A
…

Read B
…
Read A

PDC Summer School 2010

UPPSALA UNIVERSITET

The Cache Coherent Memory System



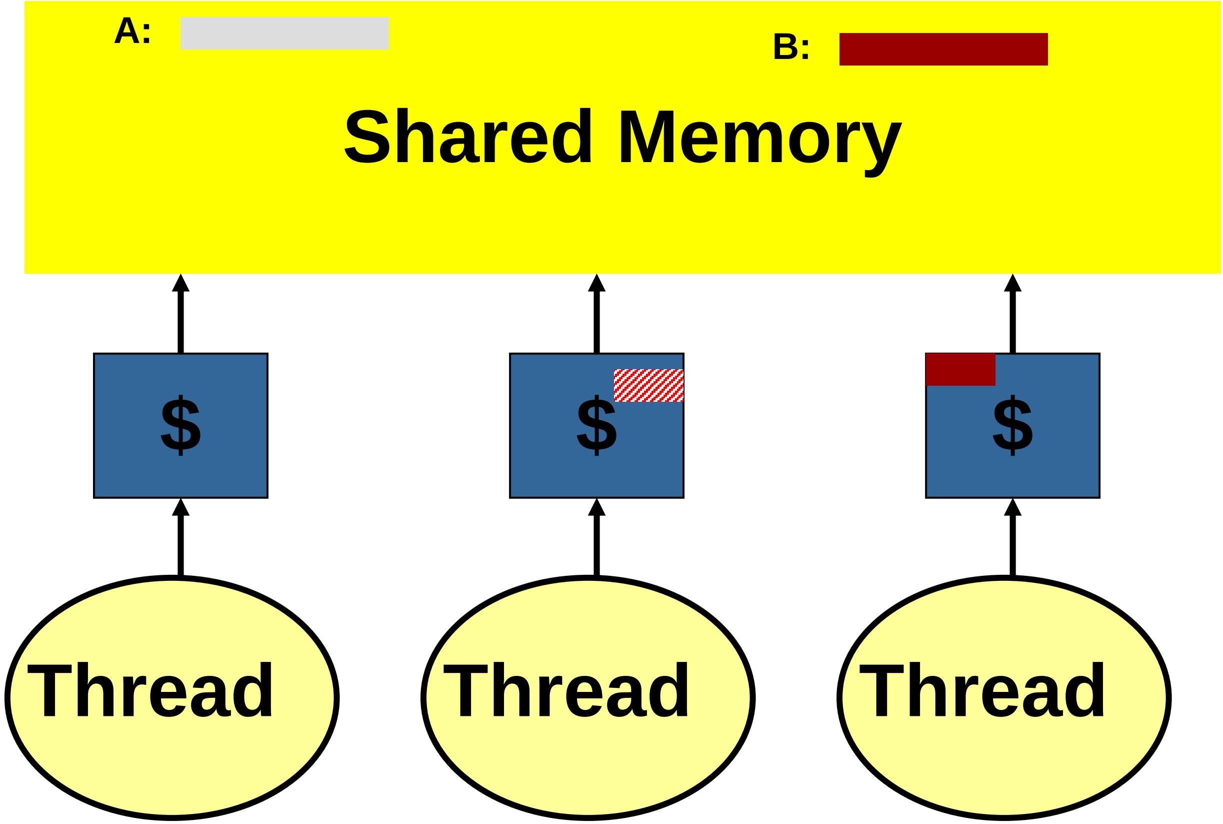
Read A
Read A
…
…

...
Read A
…
Write A

Read B
…
Read A

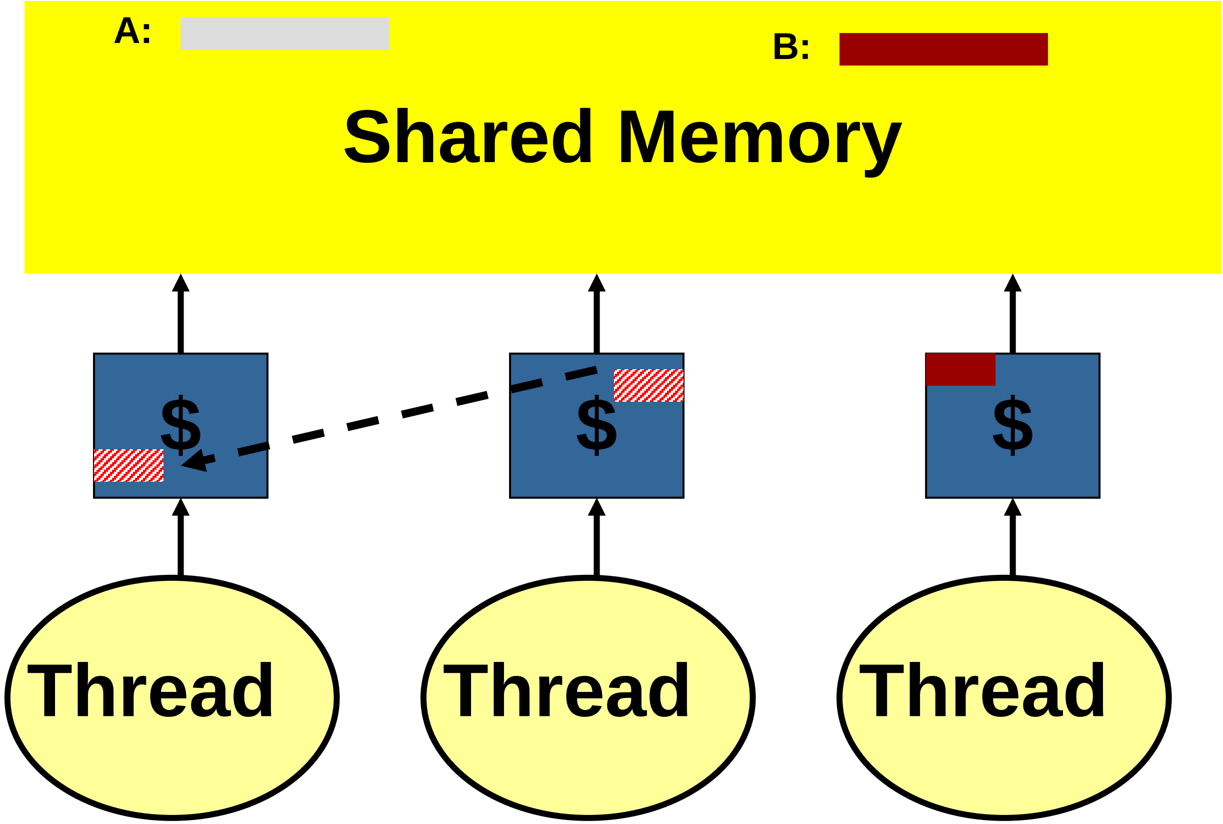
PDC Summer School 2010

The Cache Coherent Memory System



Read A	...	Read B
Read A	Read A	…
…	…	Read A
…	Write A	

The Cache Coherent Memory System

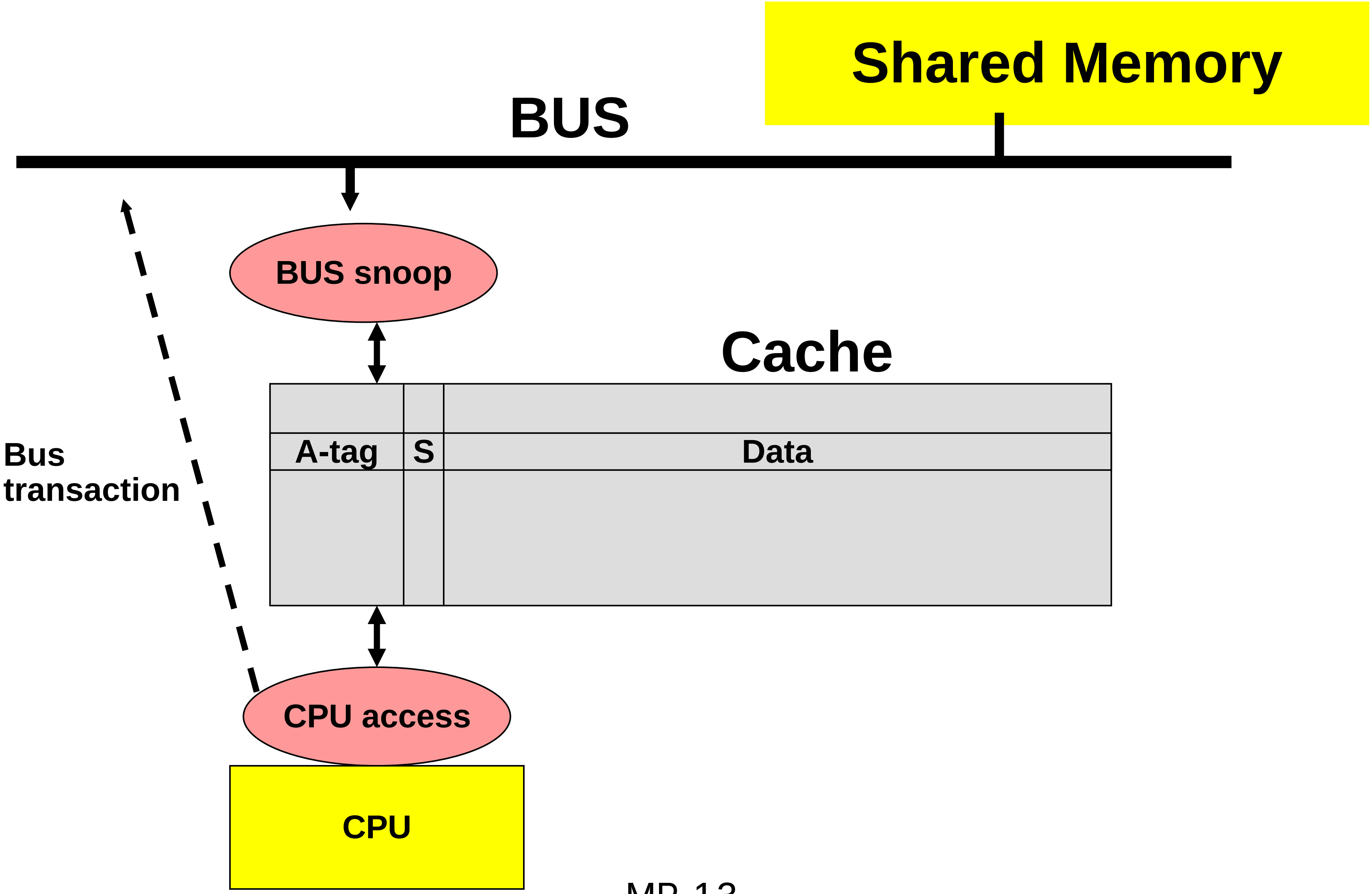


Read A
Read A
…
…
Read A

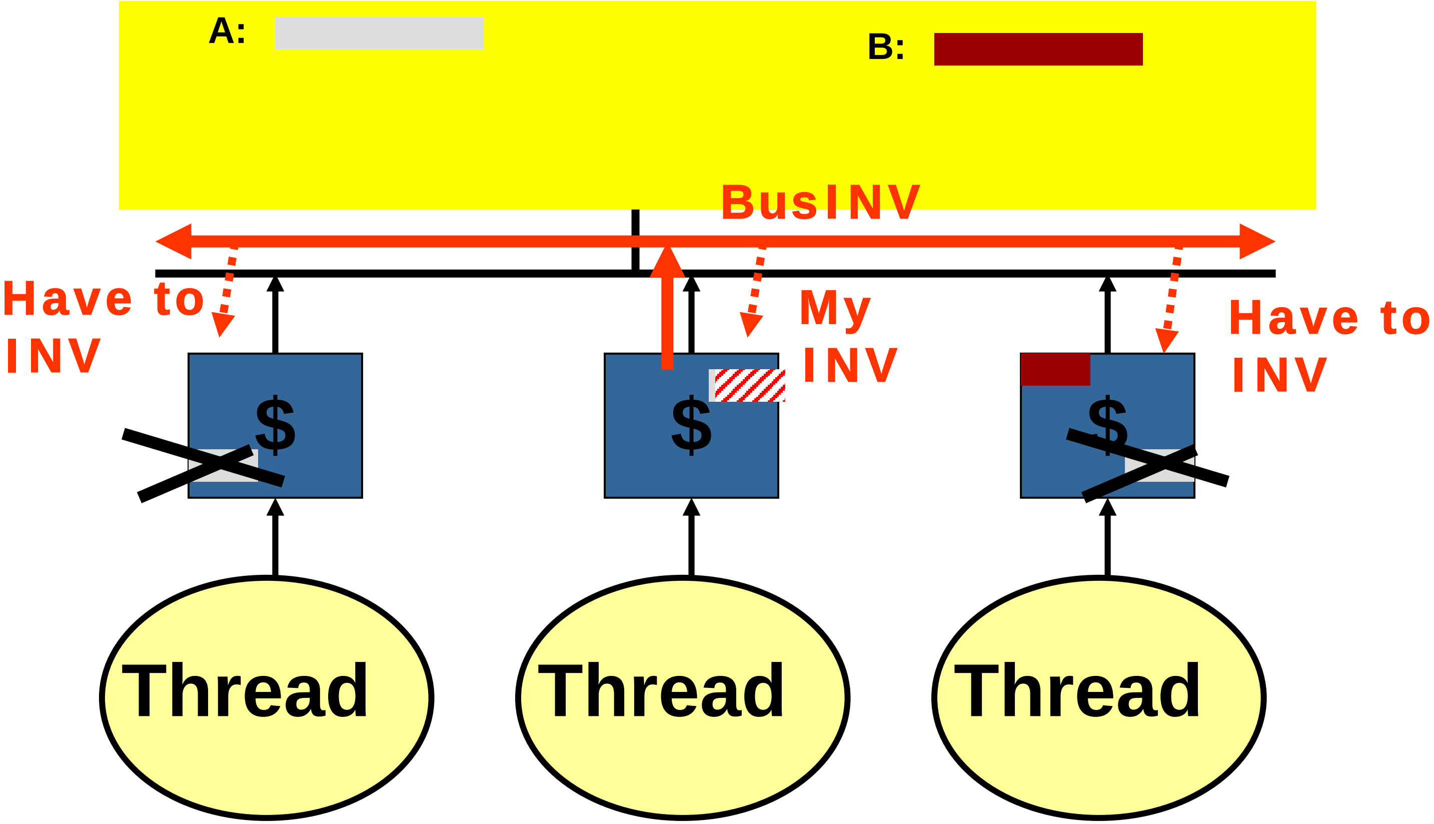
…
Read A
…
Write A

Read B
…
Read A

Snoop-based Protocol Implementation



”Upgrade” in snoop-based

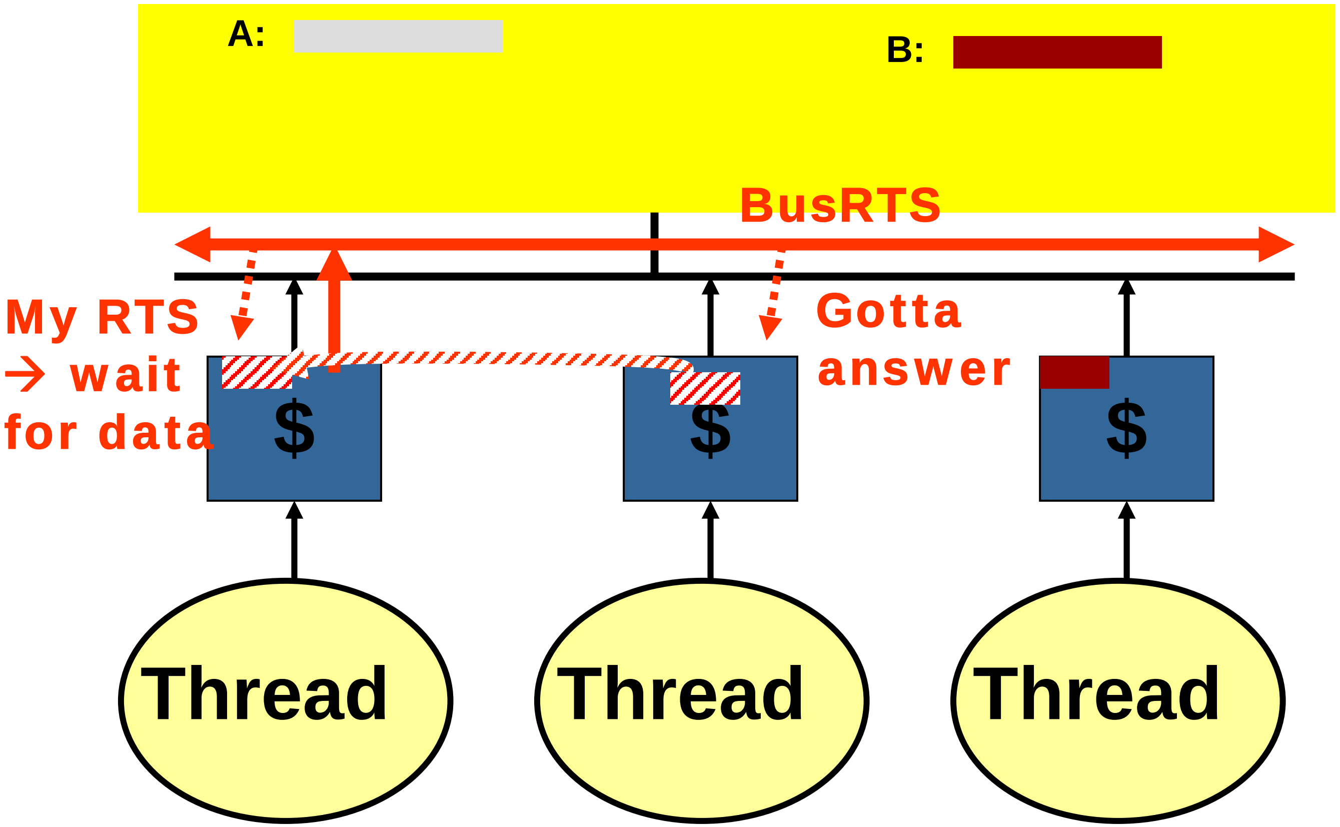


Read A
Read A
...
...

...
Read A
...
Write A

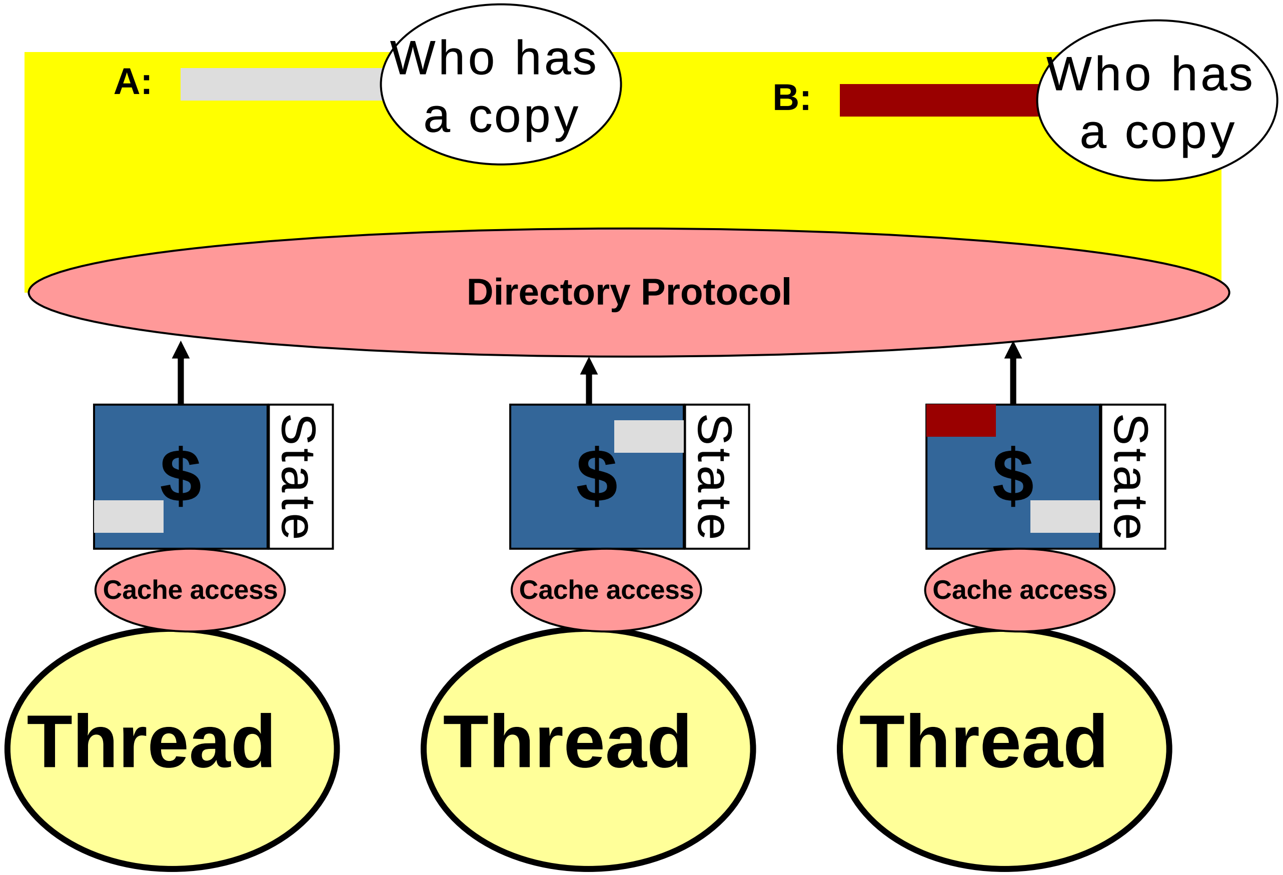
Read B
...
Read A

Cache-to-cache in snoop-based

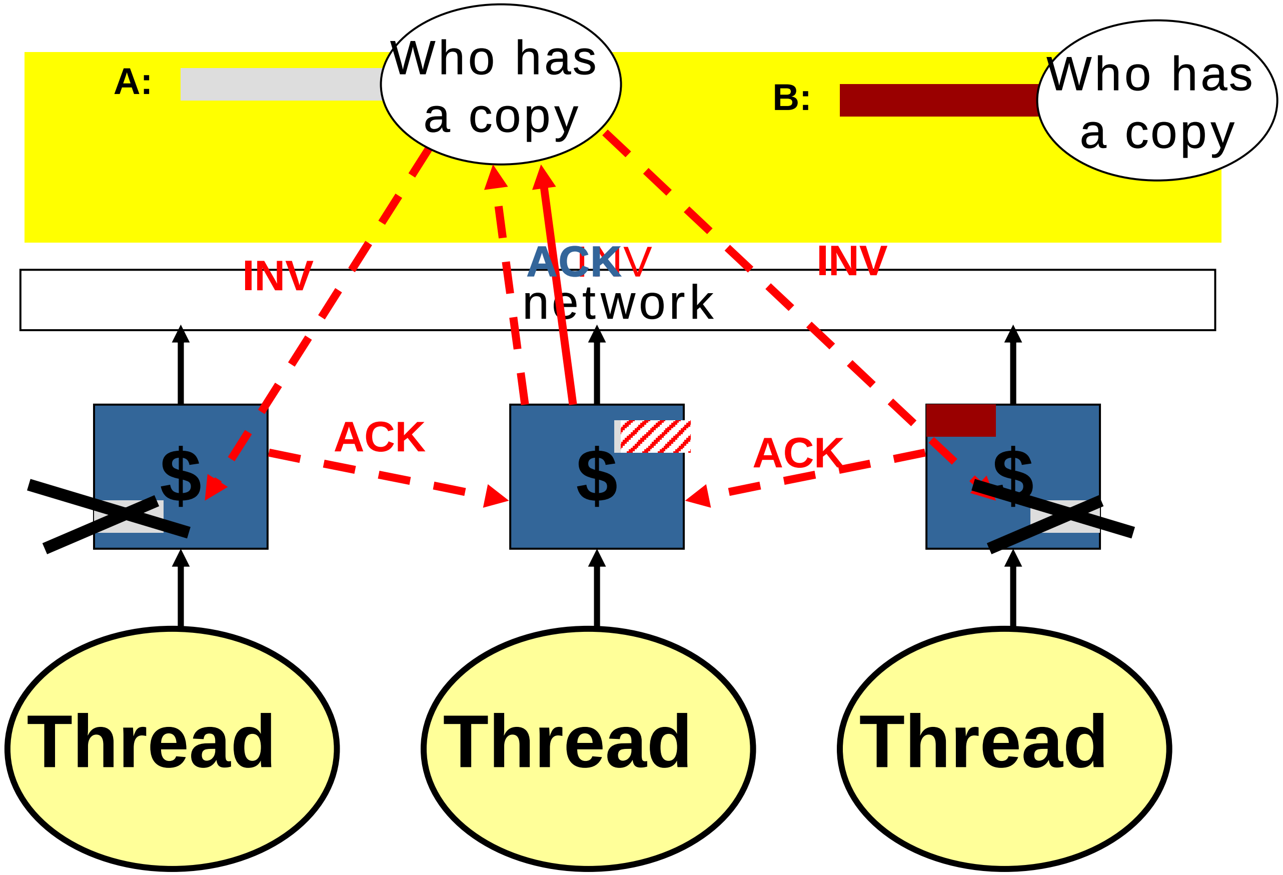


Read A	...	Read B
Read A	Read A	…
…	…	Read A
…	Write A	
Read A		

Directory-based coherence: per-cachline info in the memory



”Upgrade” in dir-based

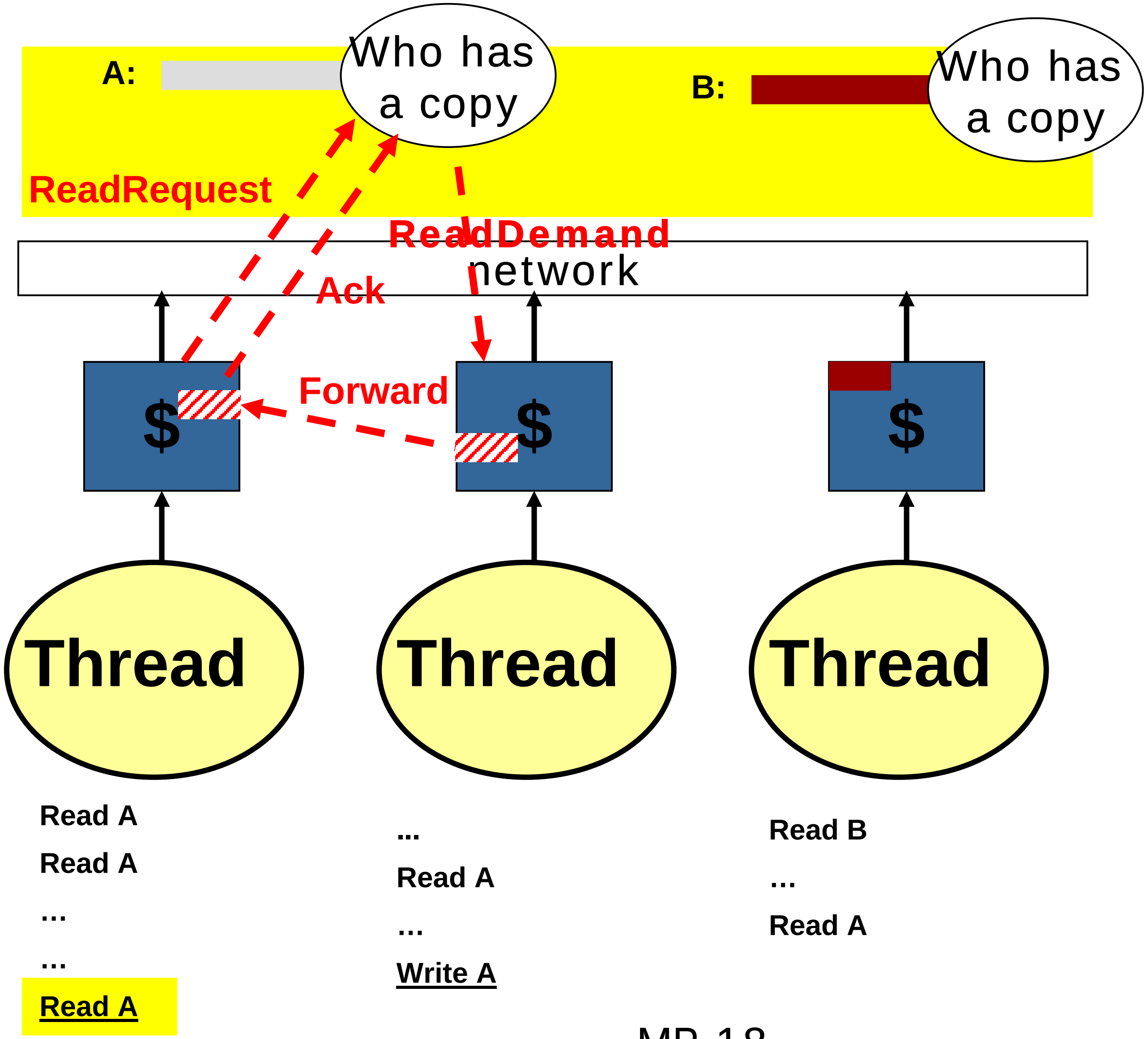


Read A
Read A
…
…

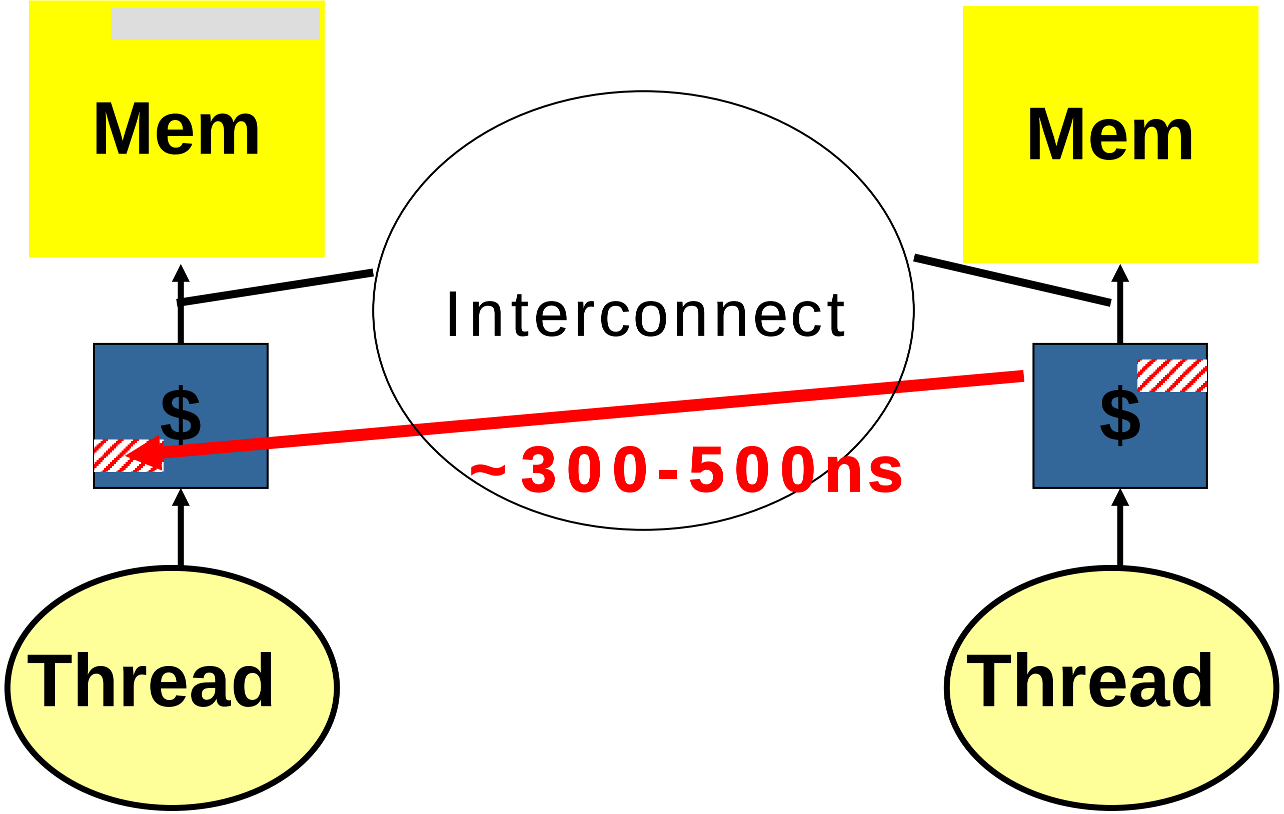
...
Read A
…
Write A

Read B
…
Read A

Cache-to-cache in dir-based



Non-uniform Architectures: NUMA & Multisocket➔ Communication cost is much worse! A case for directory-based coherence



Read A

Read A

...

...

Read A

...

Read A

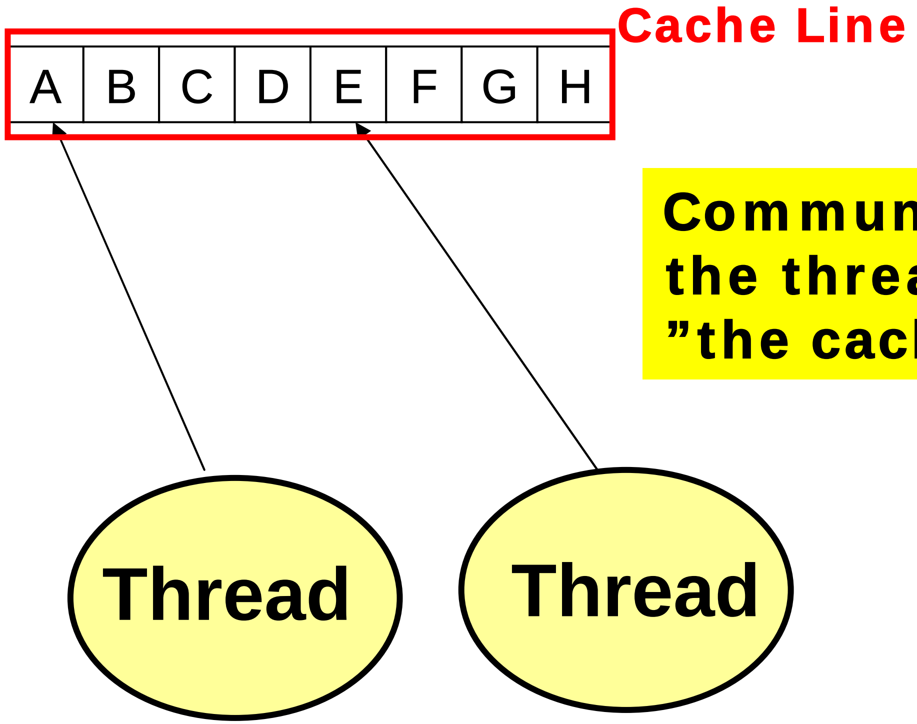
...

Write A

Why do you miss in a cache?

- Capacity misses – *the cache is too small*
- Conflict misses – *the cache organization is not perfect*
- Compulsory misses – *touching the data for the first time*
- Coherence misses – *caused by communication*

False sharing



Communication misses even though the threads do not share data ”the cache line is too large”

Read A
Write A
…
…
Read A

Read E
…
Write E

So Far...

Coherent shared memory

- "Snoopy-based coherence protocol"
  - All global accesses are broadcasted to all caches
  - Cache lines are automatically invalidated/fetched
- Directory-based coherence
- Ensures ordering and serialization for a single cache line

Sloppy definition: *There can be several copies of a datum, but only one value at "a given point in time"*

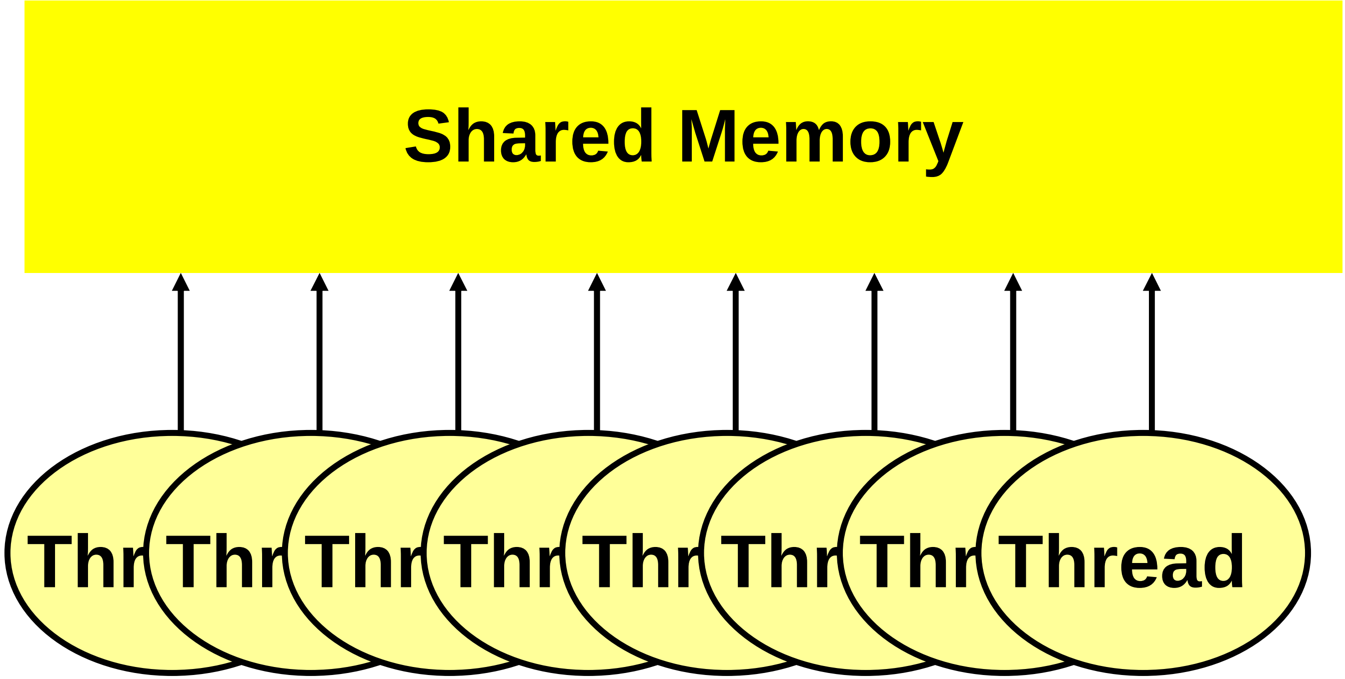
Better definition: *Only a single value-change order can be observed for each datum.*

Memory Ordering (aka Memory Consistency) -- tricky but important stuff

Erik Hagersten
Uppsala University
Sweden

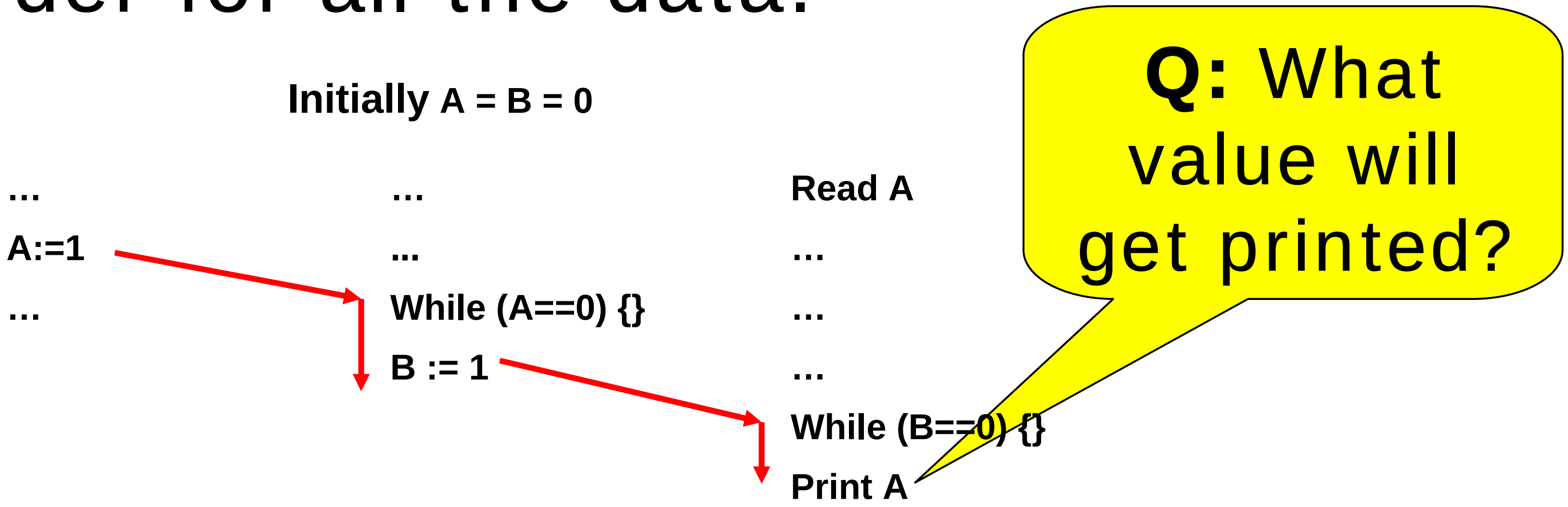
UPPSALA UNIVERSITET

The Shared Memory Programming Model (Pthreads/OpenMP, …)



Memory Ordering

- Coherence defines a per-datum valuechange order
- Memory model defines the valuechange order for all the data.

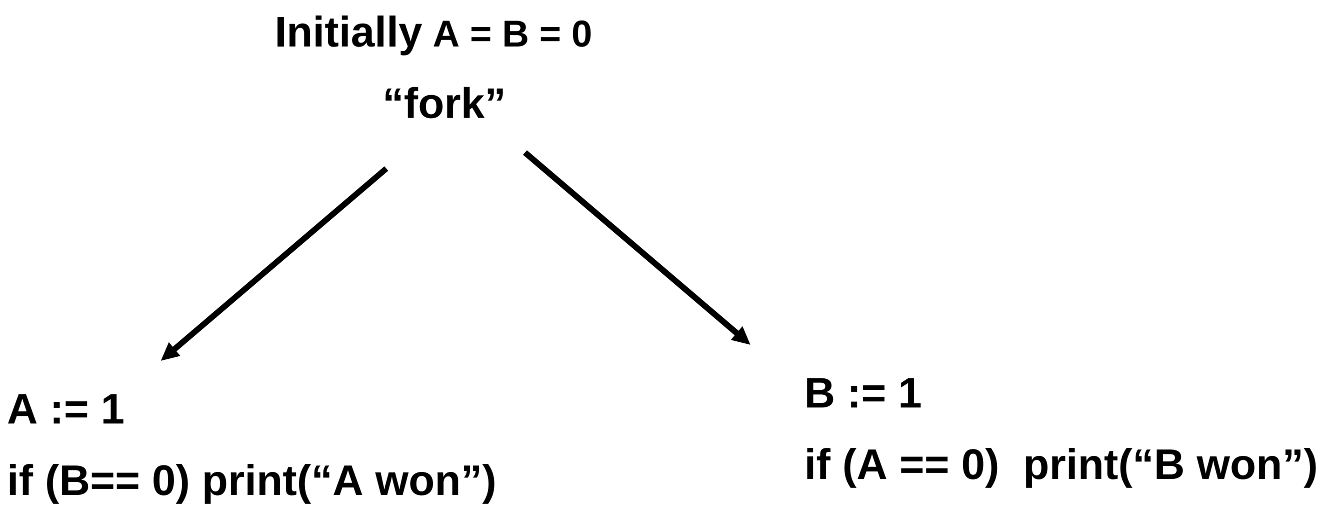


Initially A = B = 0

...	...	Read A
A:=1	...	...
...	While (A==0) {}	...
	B := 1	...
		While (B==0) {}
		Print A

Q: What value will get printed?

Dekker's Algorithm



Initially A = B = 0

"fork"

A := 1

if (B== 0) print("A won")

B := 1

if (A == 0) print("B won")

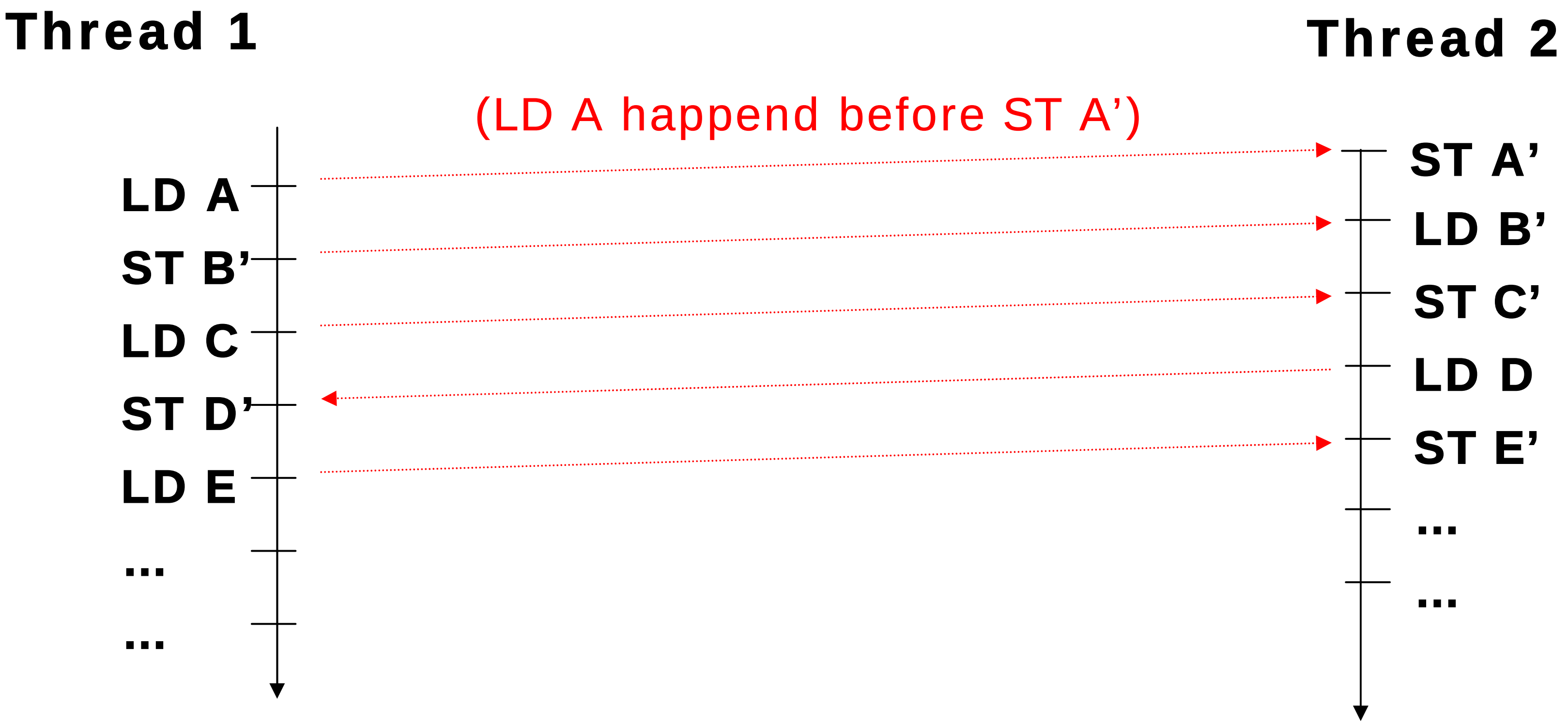
Q: Is it possible that both A and B win?

Memory Ordering

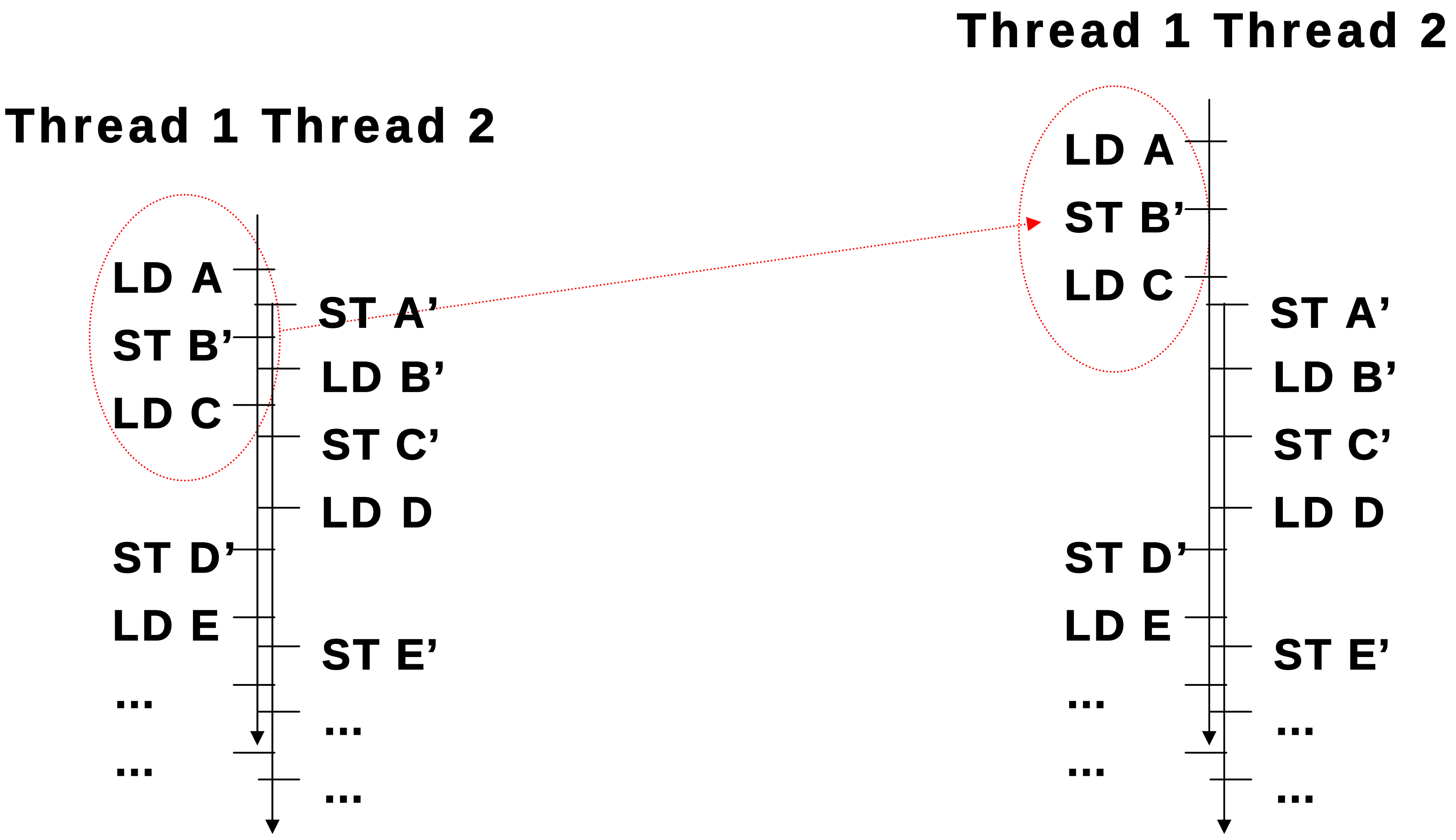
- Defines the guaranteed memory ordering: *If a thread has seen that A happens before B, what order can the other threads can observe?*
- Is a ”contract” between the HW and SW guys
- Without it, you can not say much about the result of a parallel execution

Human intuition: There is one global order!

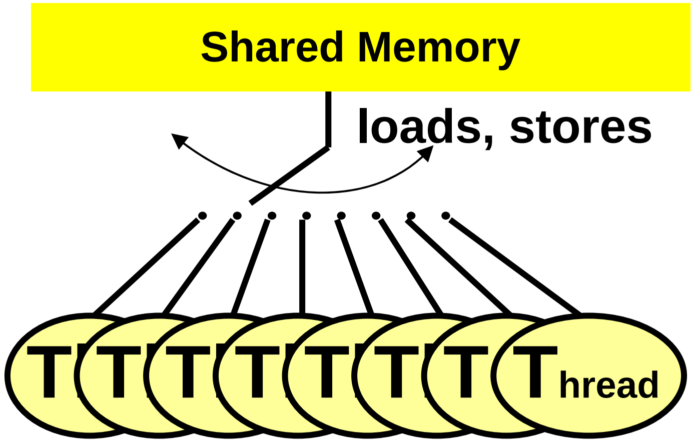
(A’ denotes a modified value to the data at addr A)



One possible observed order Another possible observed order

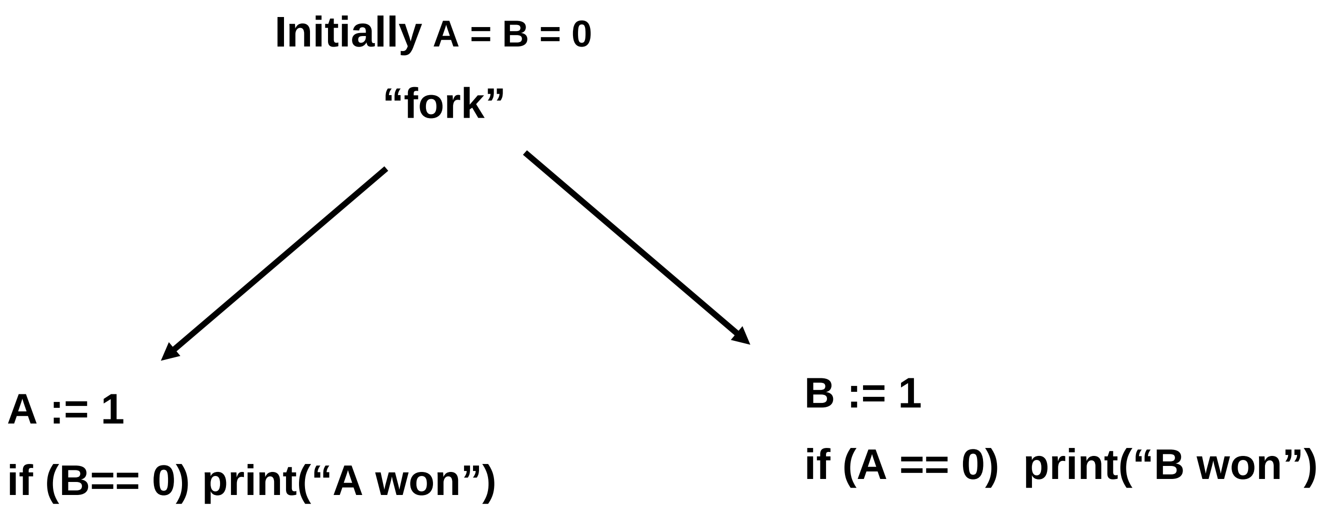


"The intuitive memory order" Sequential Consistency (Lamport)



- Global order achieved by *interleaving* all memory accesses from different threads
- "Programmer's intuition is maintained"
  - Store causality? Yes
  - Does Dekker work? Yes
- Unnecessarily restrictive ==> performance penalty

Dekker’s Algorithm



Initially A = B = 0

“fork”

A := 1

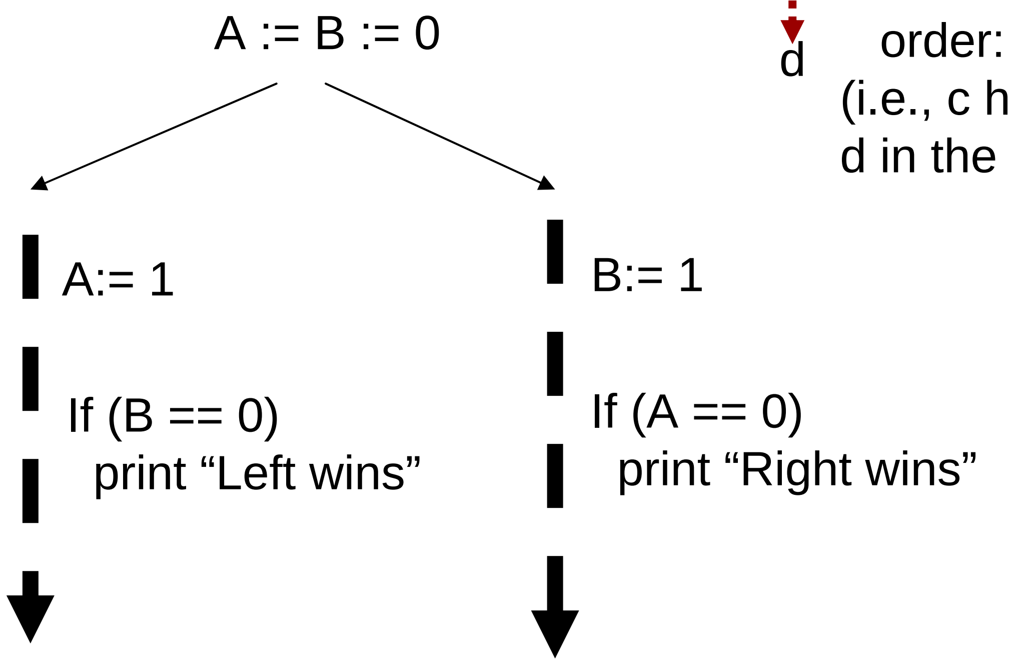
if (B== 0) print(“A won”)

B := 1

if (A == 0) print(“B won”)

Q: Is it possible that both A and B win?

Sequential Consistency (SC) Violation → Dekker: both wins

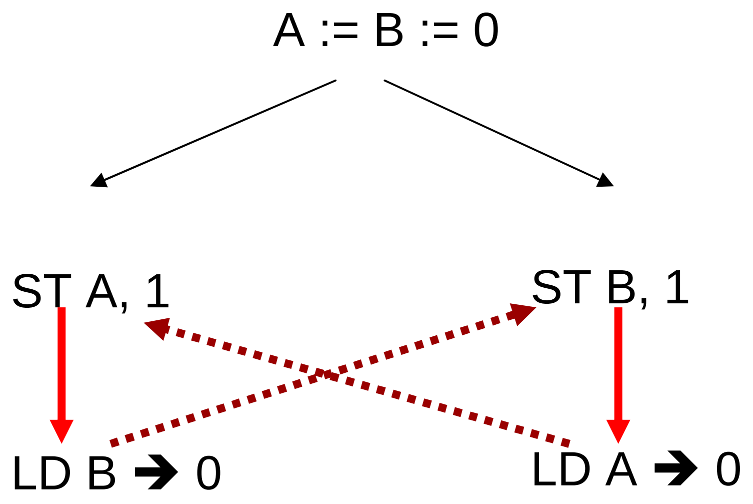


Both Left and Right wins → SC violation

Acess graph

c ⇣ d = VO: Value order: c < d (i.e., c happened before d in the global order)

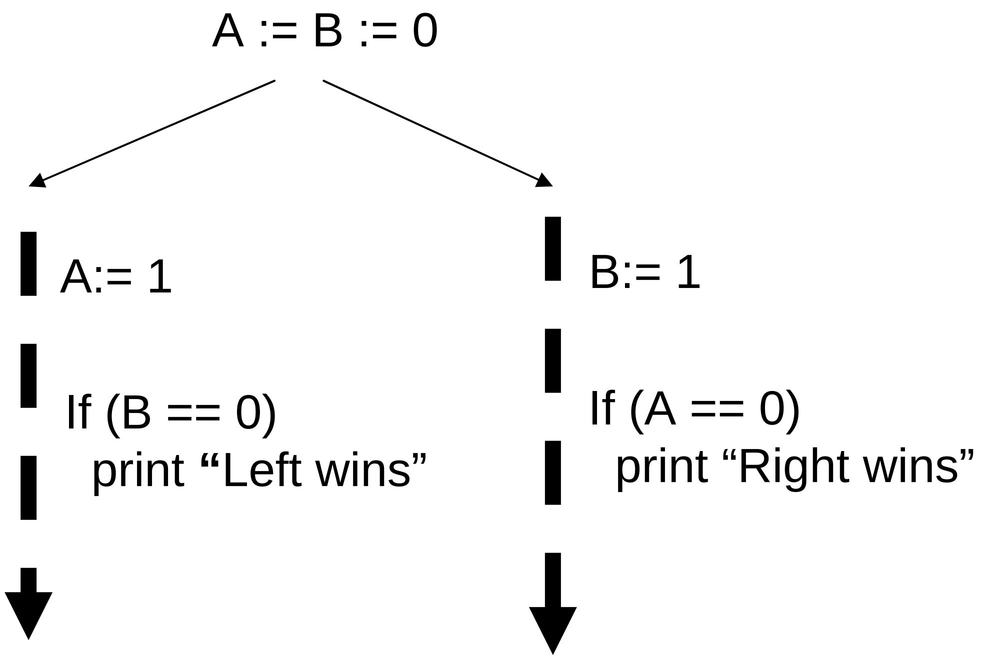
a ↓ b = PO: Program order: a < b (the order specified by the program)

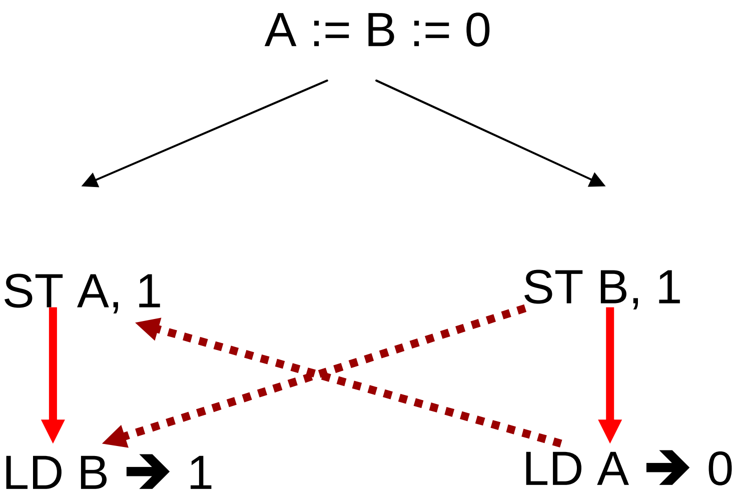


Cyclic access graph → Not SC (there is no global order)

SC is OK if one thread wins

Only Right wins → SC is OK



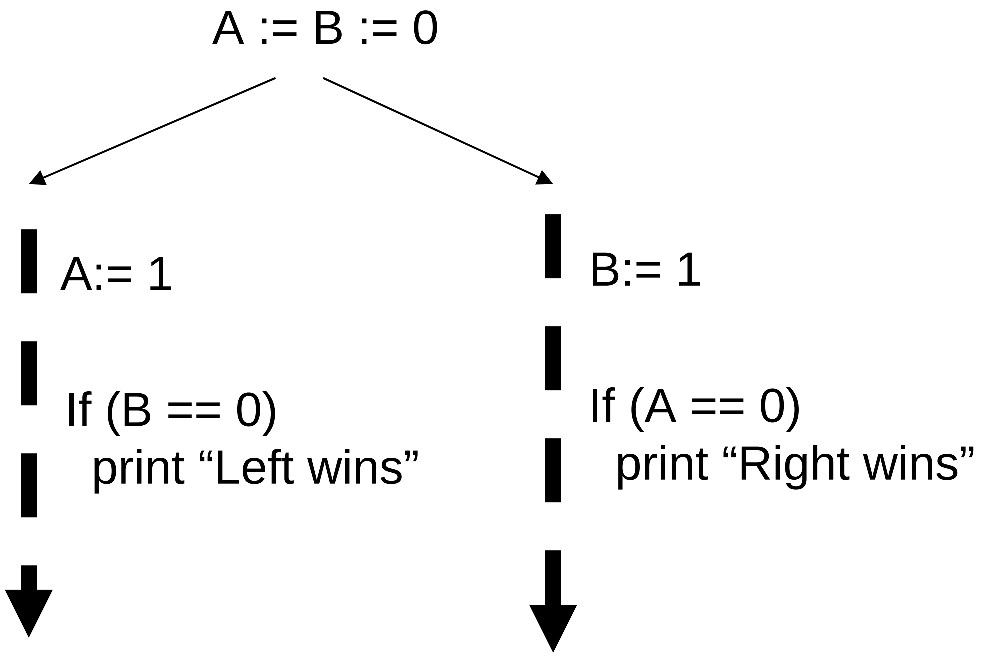


Not cyclic graph → SC

One global order:
STB < LDA < STA <LDB

SC is OK if no thread wins

No thread wins ➔ SC is OK



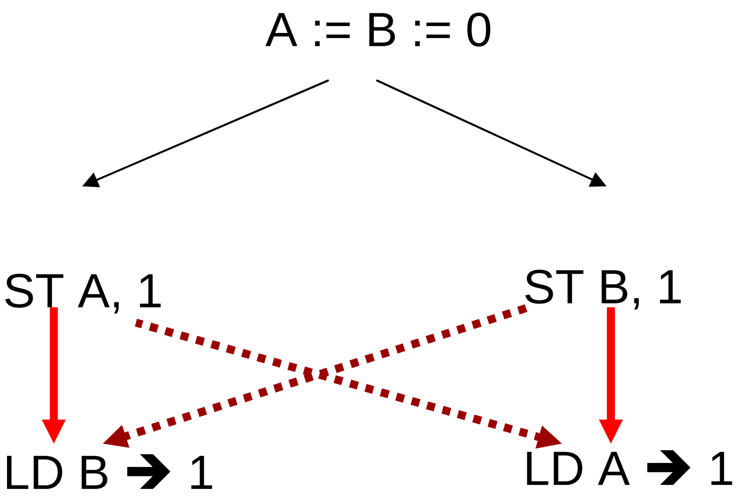
A := B := 0

A:= 1

If (B == 0)
print “Left wins”

B:= 1

If (A == 0)
print “Right wins”



A := B := 0

ST A, 1

ST B, 1

LD B ➔ 1

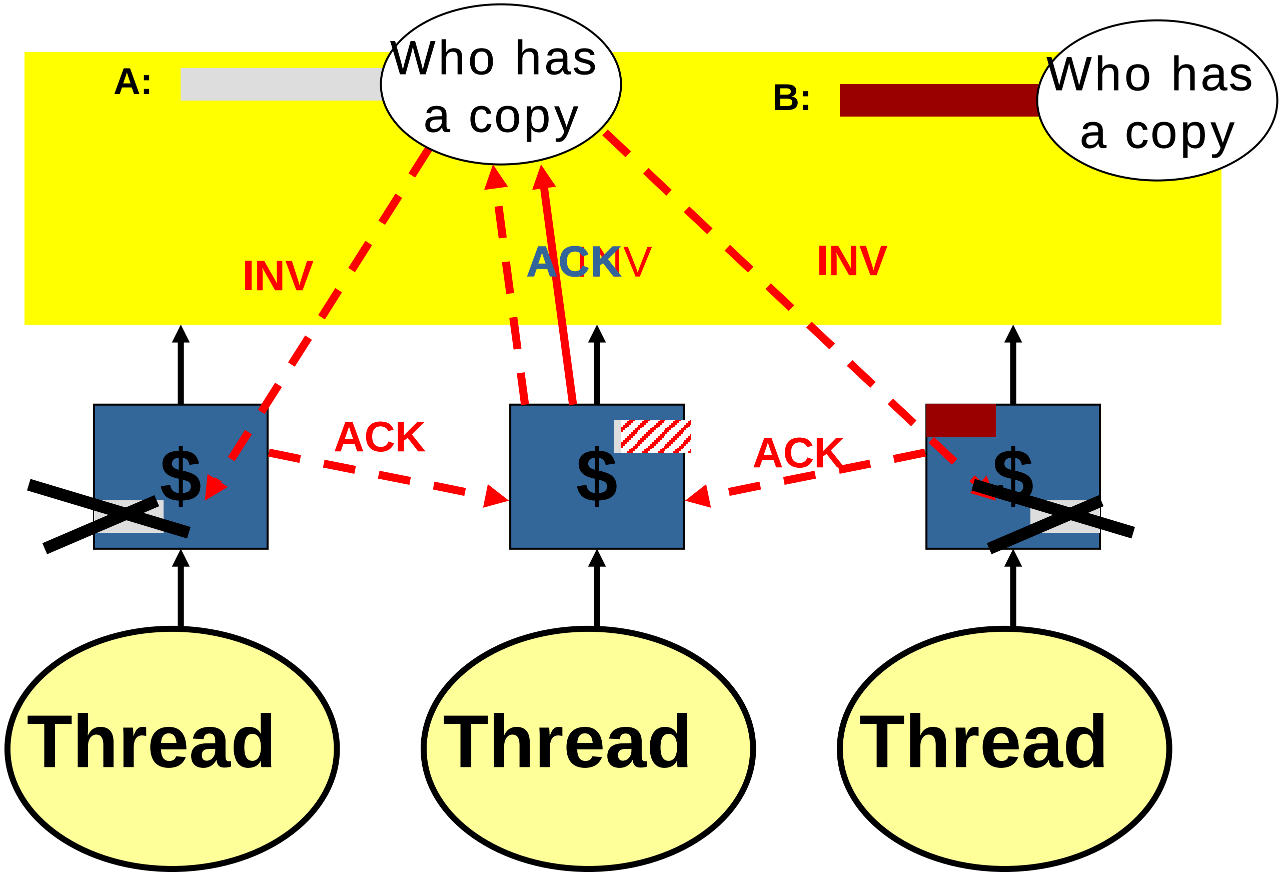
LD A ➔ 1

Not cyclic graph ➔ SC

Four Partial Orders, still SC

STB < LDA ; STA < LDA; STB < LDB ; STA < LDA

One implementation of SC in dir-based (….without speculation)



Read A
Read A
…
…

Read X
Read A
…
Write A
Read C

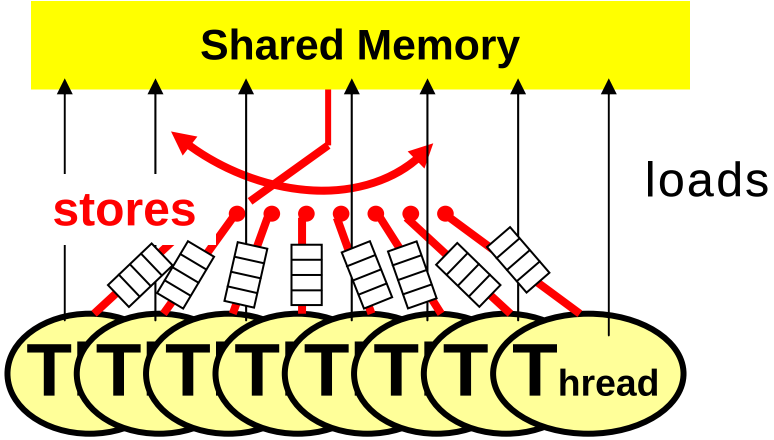
Read B
Read A

Read X must complete before starting Read A

Must receive all ACKs before continuing

UPPSALA UNIVERSITET

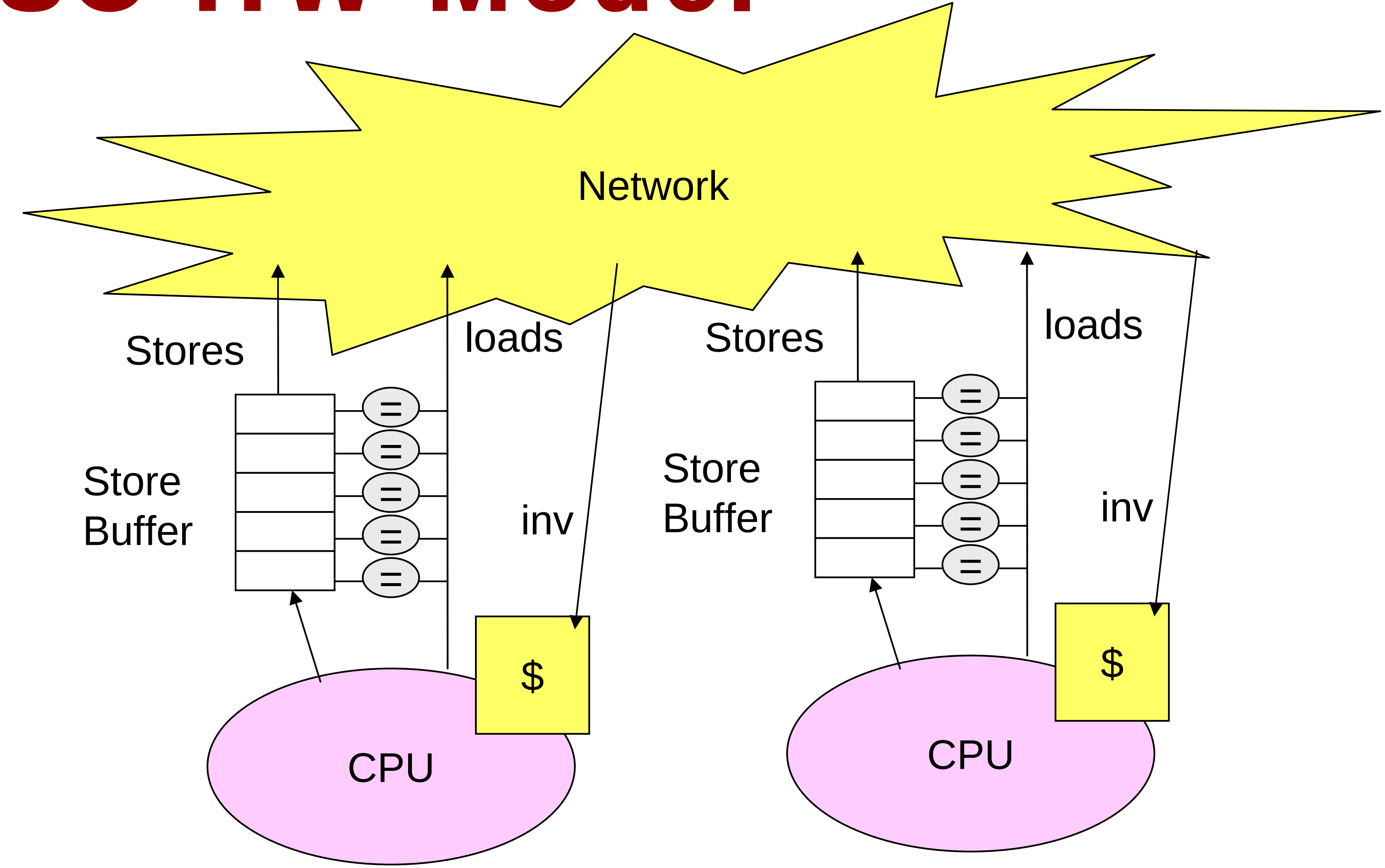
"Almost intuitive memory model" Total Store Ordering [TSO] (P. Sindhu)



- Global *interleaving* [order] for all stores from different threads (own stores excepted)
- "Programmer's intuition is maintained"
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive ==> performance penalty

UPPSALA UNIVERSITET

TSO HW Model



➔Stores are moved off the critical path

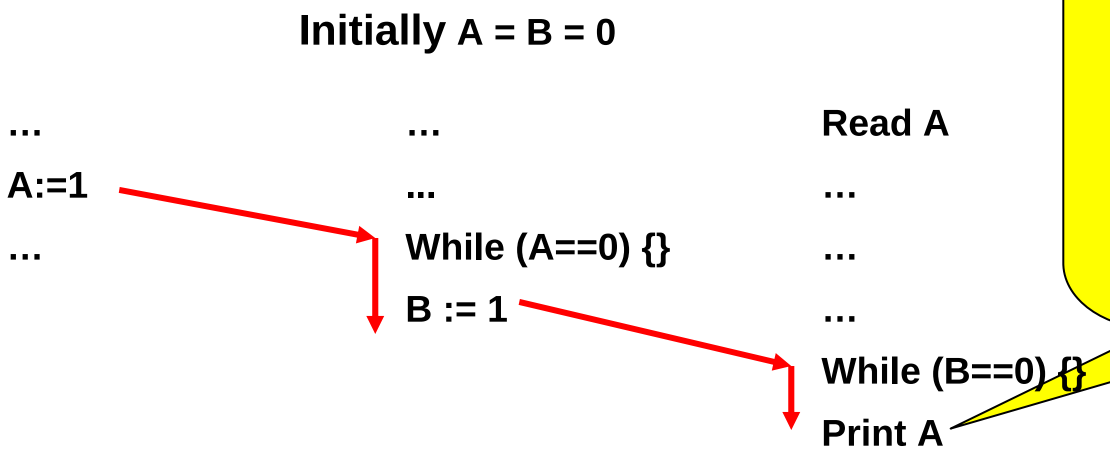
Coherence implementation can be the same as for SC

TSO

- Flag synchronization works

```
A := data                    while (flag != 1) {};
flag := 1                    X := A
```

- Provides causal correctness

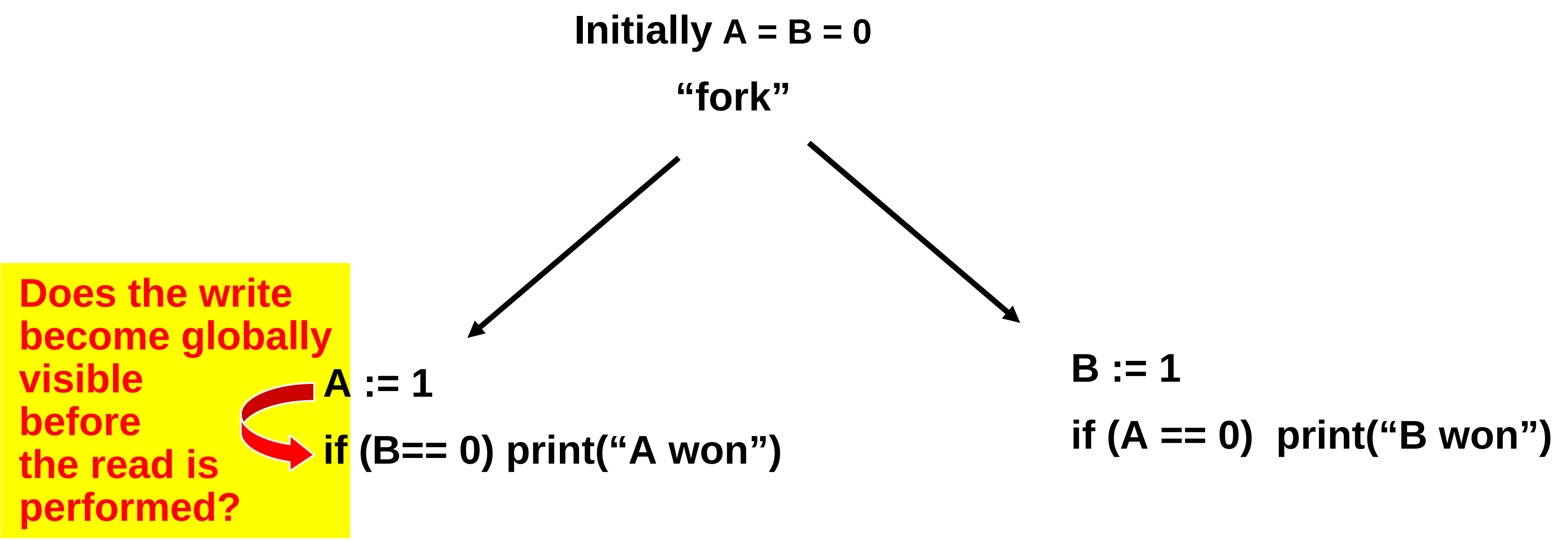


Initially A = B = 0

...	**...**	**Read A**
A:=1	**...**	**...**
...	**While (A==0) {}**	**...**
	B := 1	**...**
		While (B==0) {}
		Print A

<u>Q:</u> What value will get printed?
<u>Answer</u>: 1

Dekker's Algorithm, TSO



Initially A = B = 0

"fork"

Does the write become globally visible before the read is performed?

A := 1

if (B== 0) print("A won")

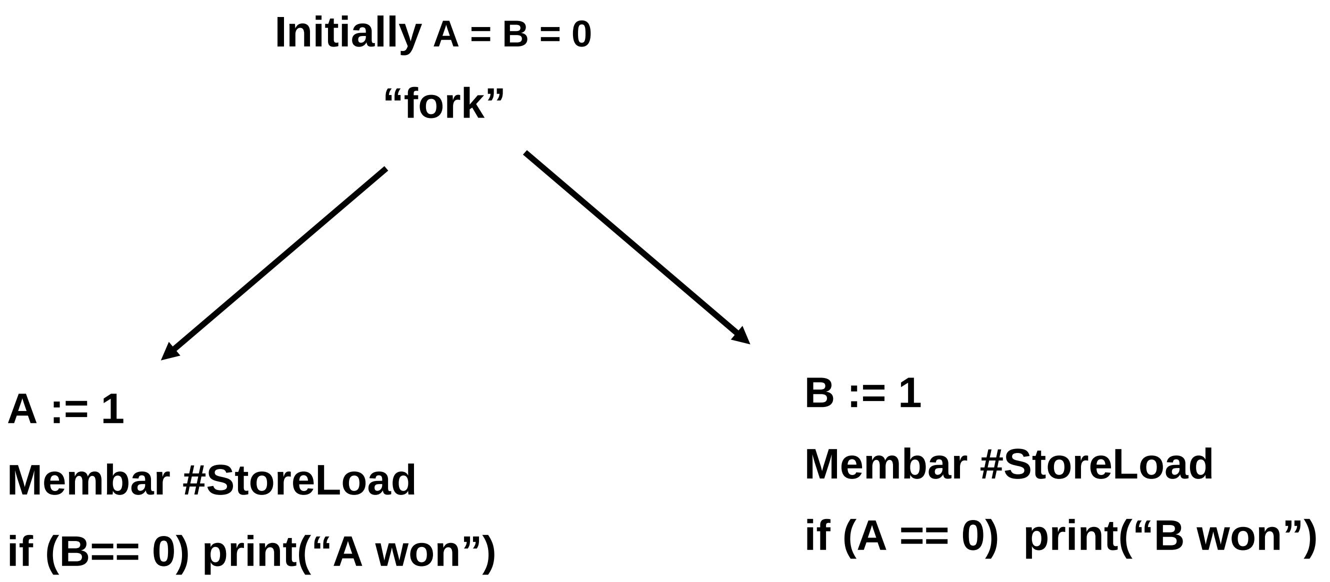
B := 1

if (A == 0) print("B won")

Q: Is it possible that both A and B wins?

Left: The read (i.e., test if B==0) can bypass the store (A:=1)
Right: The read (i.e., test if A==0) can bypass the store (B:=1)
➔both loads can be performed before any of the stores
➔yes, it is possible that both wins
➔➔ Dekker's algorithm breaks

Dekker’s Algorithm for TSO



Initially A = B = 0

“fork”

A := 1

Membar #StoreLoad

if (B== 0) print(“A won”)

B := 1

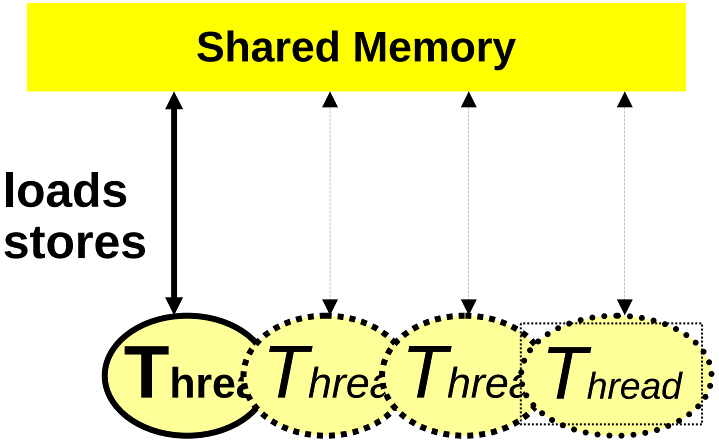
Membar #StoreLoad

if (A == 0) print(“B won”)

Q: Is it possible that both A and B win?

Membar: The read is stared after all previous stores have been ”globaly ordered”
➔behaves like SC
➔ Dekker’s algorithm works!

Weak/release Consistency (M. Dubois, K. Gharachorloo)



- **Most accesses are unordered**
- **"Programmer's intuition is not maintained"**
  - Store causality? No
  - Does Dekker work? No
- **Global order <u>only</u> established when the programmer explicitly inserts memory barrier instructions**

++ Better performance!!

--- Interesting bugs!!

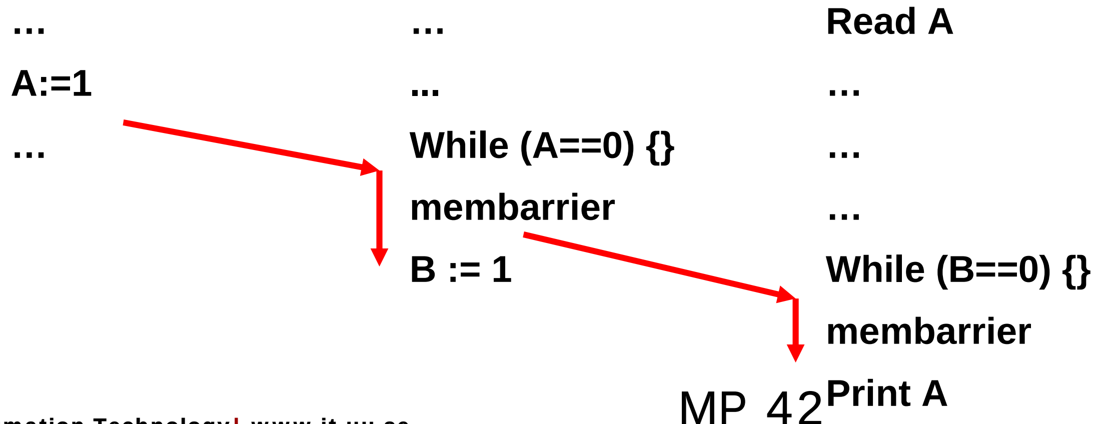
Weak/Release consistency

- New flag synchronization needed

```
A := data;              while (flag != 1) {};
membarrier;             membarrier;
flag := 1;              X := A;
```

- Dekker's: same as TSO
- Causal correctness provided for this code

Initially A = B = 0



...	**...**	**Read A**
A:=1	**...**	**...**
...	**While (A==0) {}**	**...**
	membarrier	**...**
	B := 1	**While (B==0) {}**
		membarrier
		Print A

Q: What value will get printed?
Answer: 1

Learning more about memory models

Shared Memory Consistency Models: A Tutorial
by Sarita Adve, Kouroush Gharachorloo
in IEEE Computer 1996 **(in the ”Papers” directory)**

RFM: Read the F*****n Manual of the system you are working on!
(Different microprocessors and systems supports different memory models.)

Issue to think about:
What code reordering may compilers really do?
Have to use ”volatile” declarations in C.

X86's new memory model

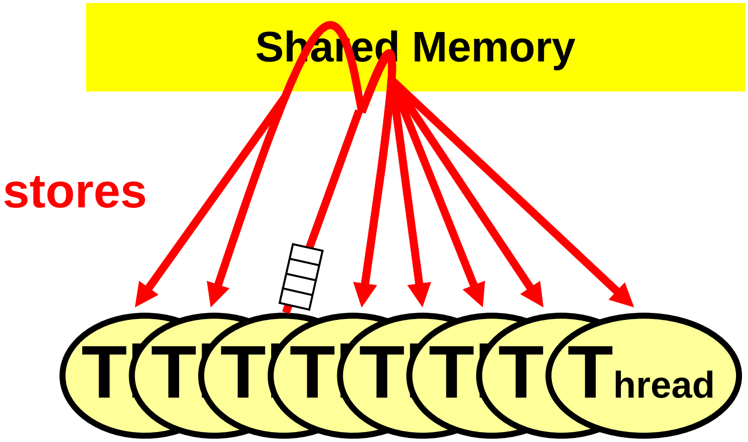
- Processor consistency with causual correctness for non-atomic memory ops
- TSO for atomic memory ops

- Video presentation:

  http://www.youtube.com/watch?v=WUfvvFD5tAA&hl=sv

- See section 8.2 in this manual:
  http://developer.intel.com/Assets/PDF/manual/253668.pdf

Processor Consistency [PC] (J. Goodman)



- PC: The stores from a processor appears to others in program order
- Causal correctness (often added to PC): if a processor observes a store before performing a new store, the observed store must be observed before the new store by all processors

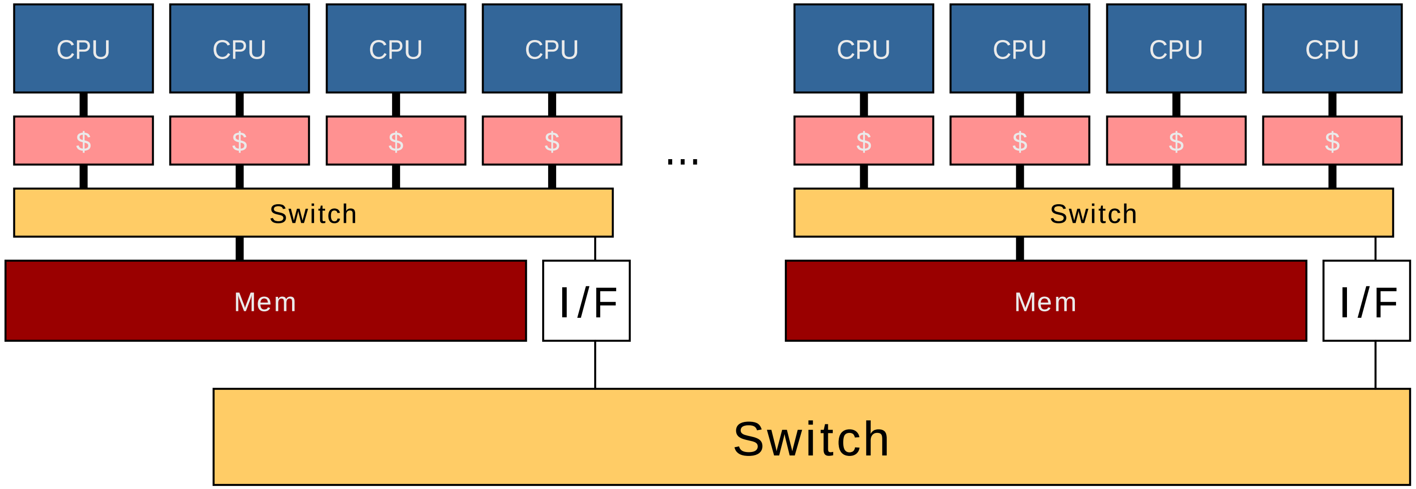
➔ Flag synchronization works.

➔ No causal correctness issues

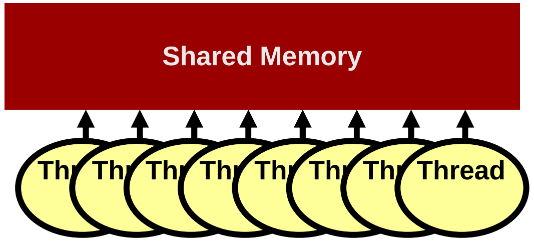
Scalable Systems

Erik Hagersten
Uppsala University

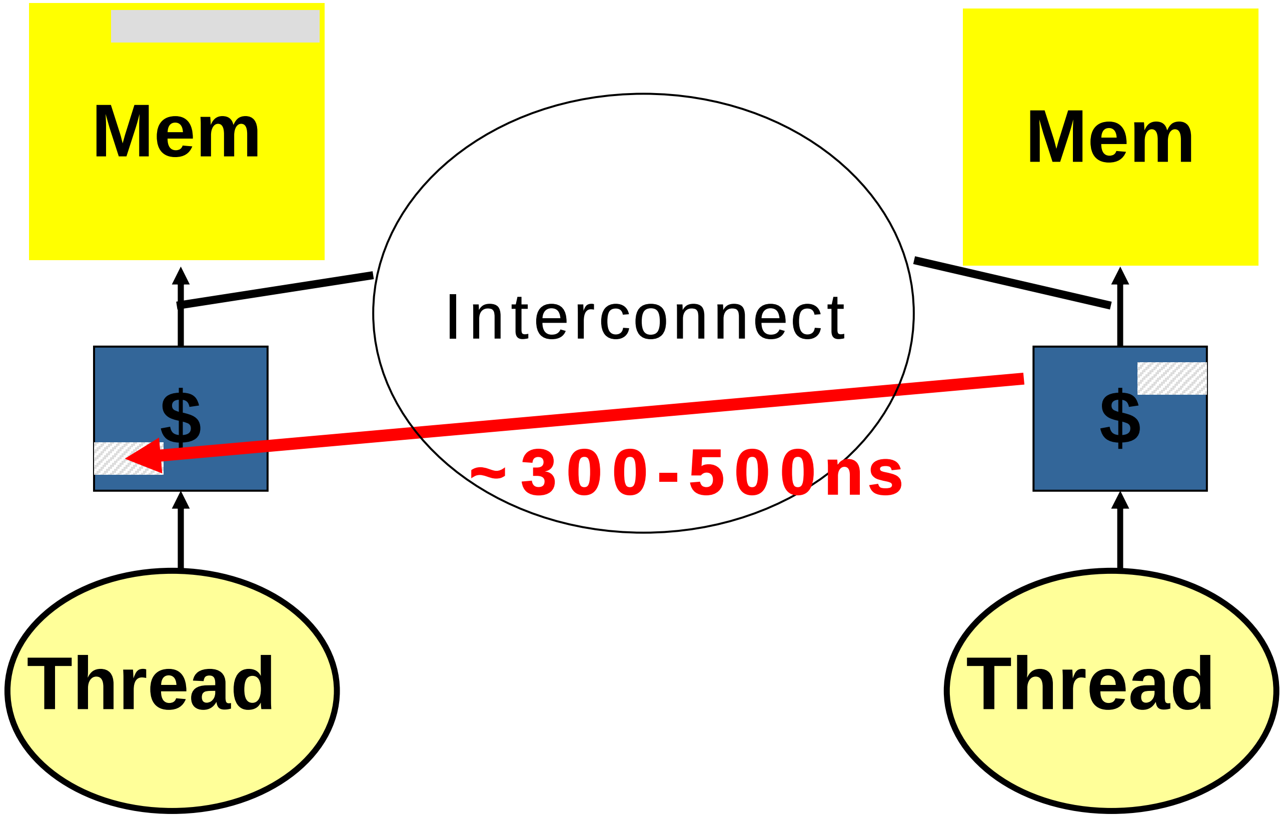
NUMA: Non-uniform memory architecture



Same SW view:



Non-uniform Architectures: NUMA
→ Communication cost is much worse!



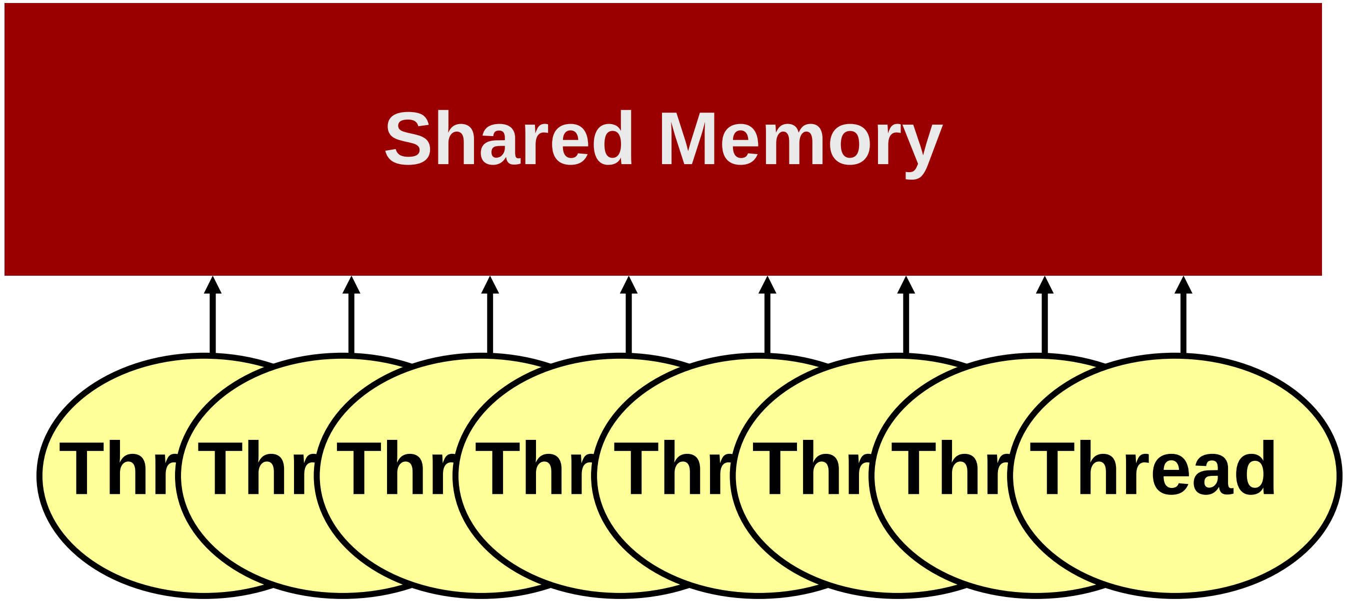
Read A	…
Read A	Read A
…	…
…	Write A
Read A	

Example

Efficient us of coherent cache Synchronization

Erik Hagersten
Uppsala University

The programmer's view



Synchronization

- Locking primitives are needed to ensure that only one process can be in the critical section:

```
LOCK(lock_variable) /* wait for your turn */
  if (sum > threshold)
    sum := my_sum + sum
UNLOCK(lock_variable) /* release the lock*/
```

Critical Section

How to implement Lock / Unlock efficiently with coherent cache?
1. Read-replication is cheap;
2. Coherence traffic is expensive

A Bad Example: ”POUNDING”

```
proc lock(lock_variable) {
    while (TAS[lock_variable]==1)  {}; /* bang on the lock until free */
}

proc unlock(lock_variable) {
     lock_variable = 0;
}
```

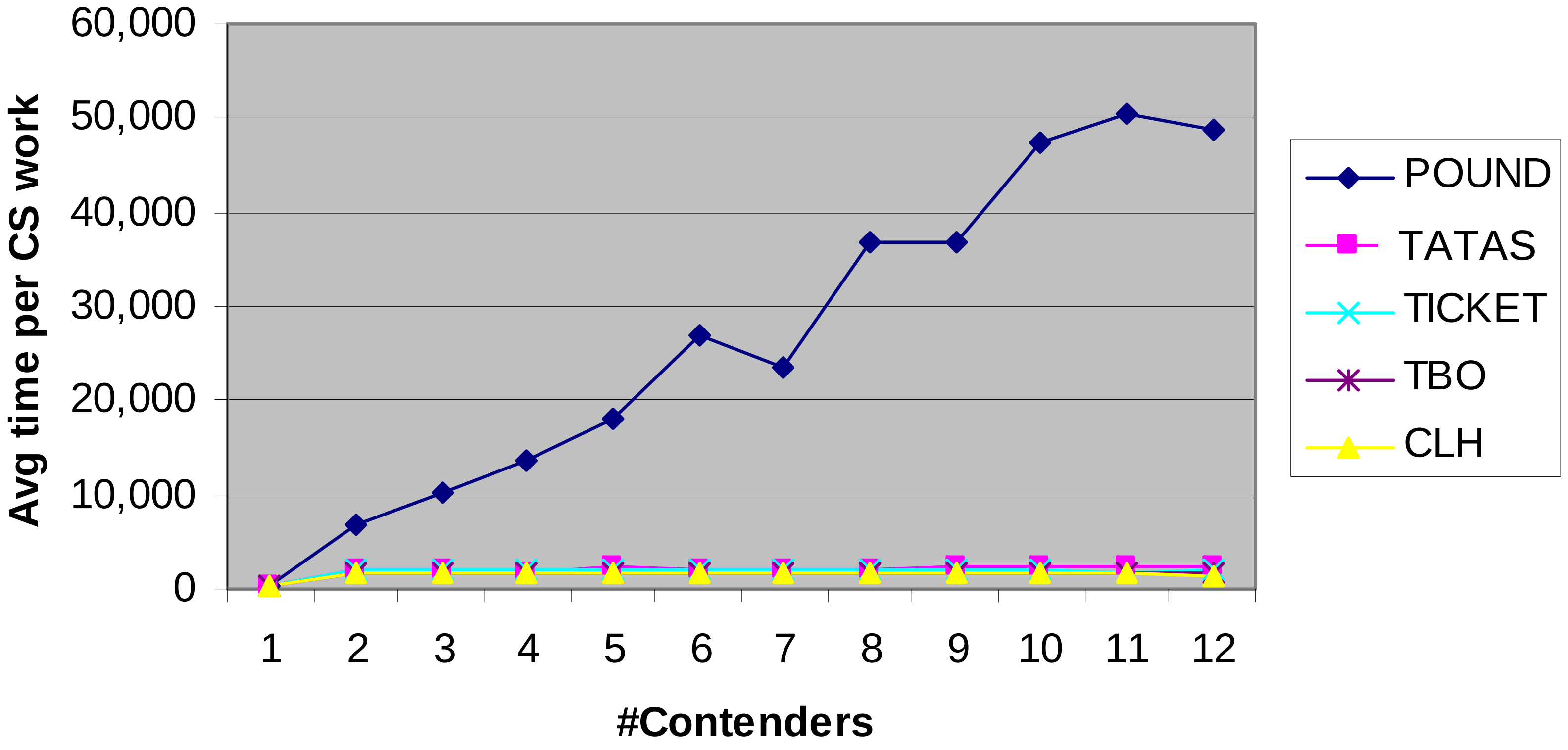
- *Assume: The atomic function TAS (test and set) returns the current memory value and atomically writes “1” to the memory location.*

Optimistic Test&Set Lock ”spinlock” (called “TATAS”, test and test-and-set)

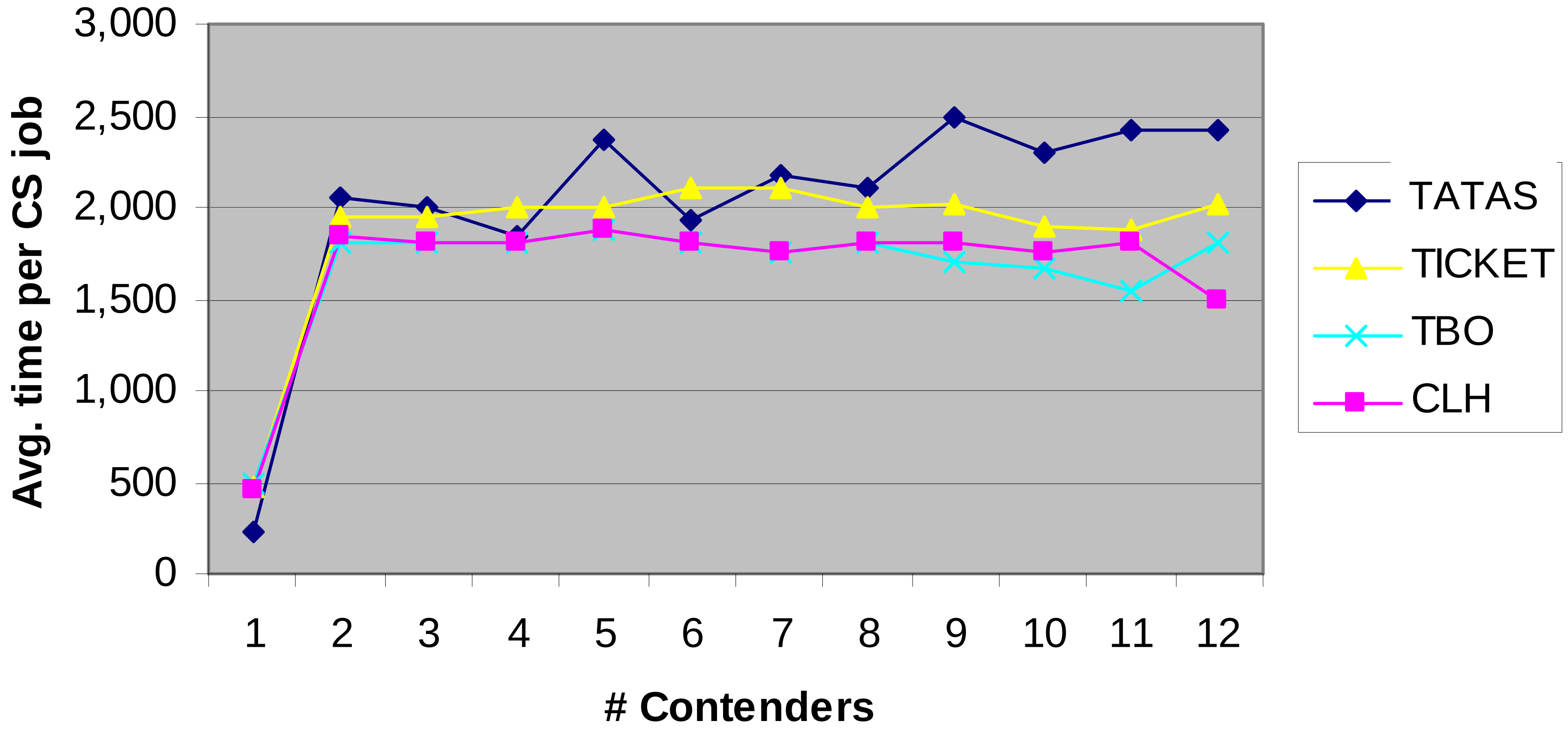
```
proc lock(lock_variable) {
     while true {
       if (TAS[lock_variable] ==0)  break;      /* test once, done if TAS==0 */
       while(lock_variable != 0) {}             /* spin locally in your cache until
                                                /* ”0” is observed*/
   }
}

proc unlock(lock_variable) {
      lock_variable := 0
}
```

UPPSALA UNIVERSITET

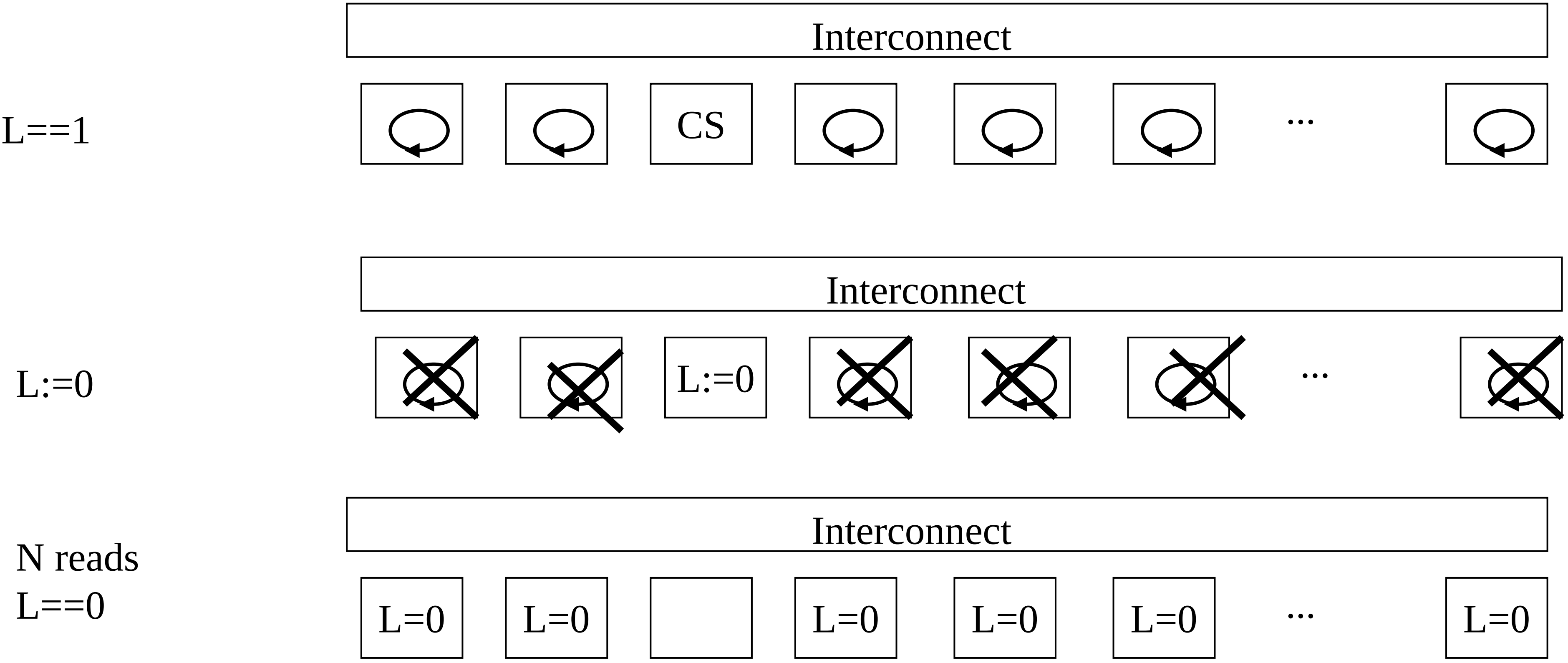
Uniform memory architecture: E6800 12 CPUs



E6800 locks (exluding POUND)

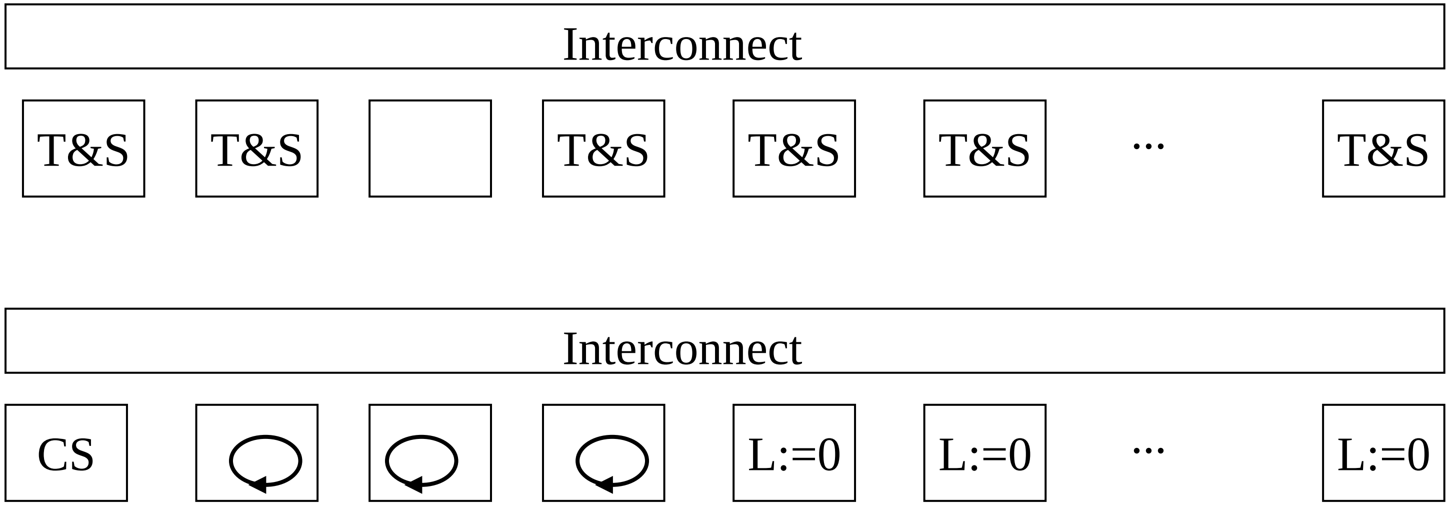


TATAS could still get messy!



...messy (part 2)

N-1 Test&Set
(i.e., N writes)



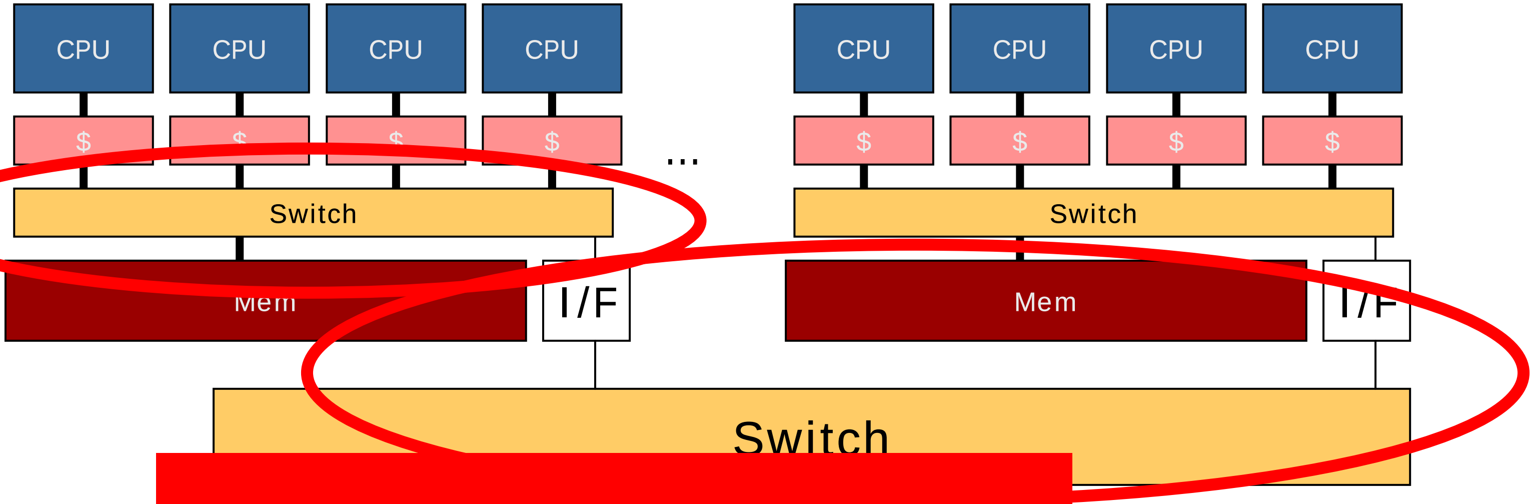
L== 1

potentially: ~N*N/2 reads :-(

Problem1: Contention on the interconnect slows down the CS proc
Problem2: The lock hand-over time is N*read_throughput
Fix1: some back-off strategy, bad news for hand-over latency
Fix2: Queue-based locks
Fix3: Generate handover locality

NUCA: Non-uniform Comm Arch.

NUMA:



Snoop

Directory-latency = 6x snoop
i.e., roughly CMP NUCA-ness

Queue-based Locks

- Inserts the contenders in a FIFO order
- Gauge the completion of your predecessor [only] while waiting

➔ Fairness

➔ Reduced Traffic

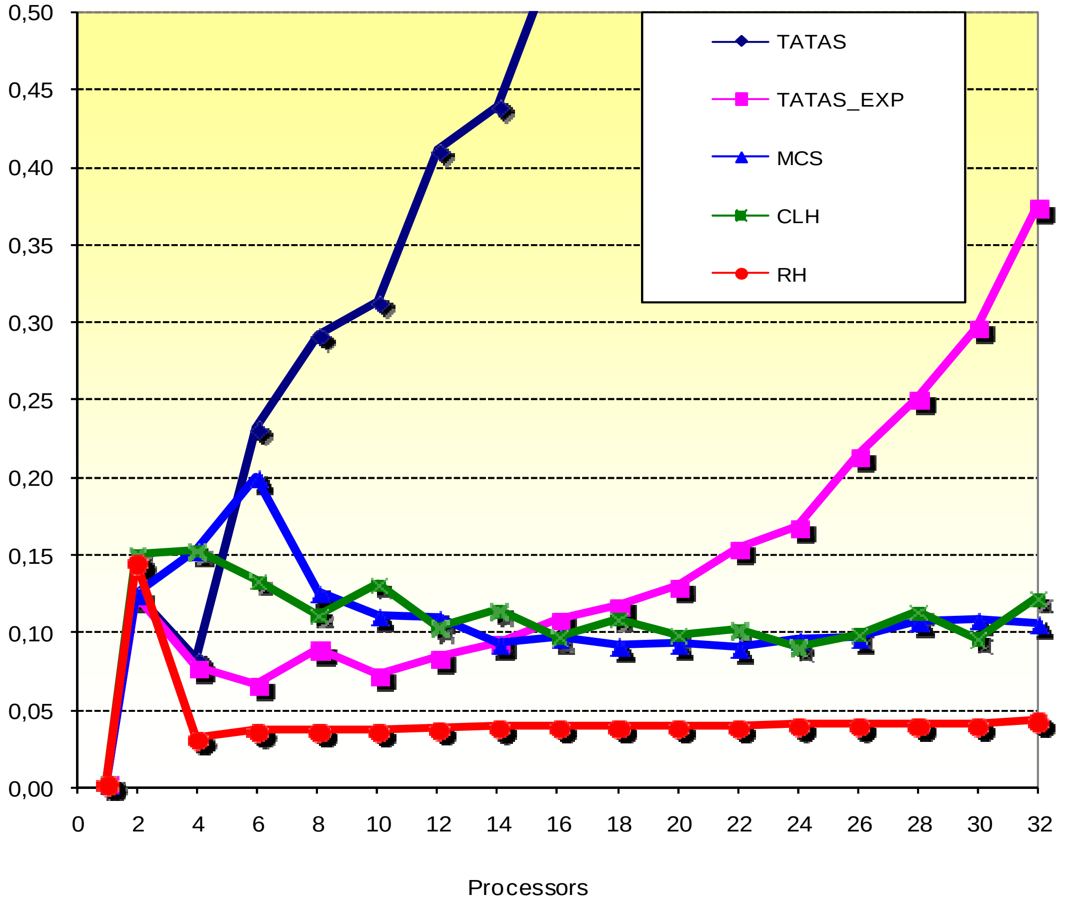
Examples: MCS locks and CLH locks

UPPSALA UNIVERSITET

UART research: RH locks

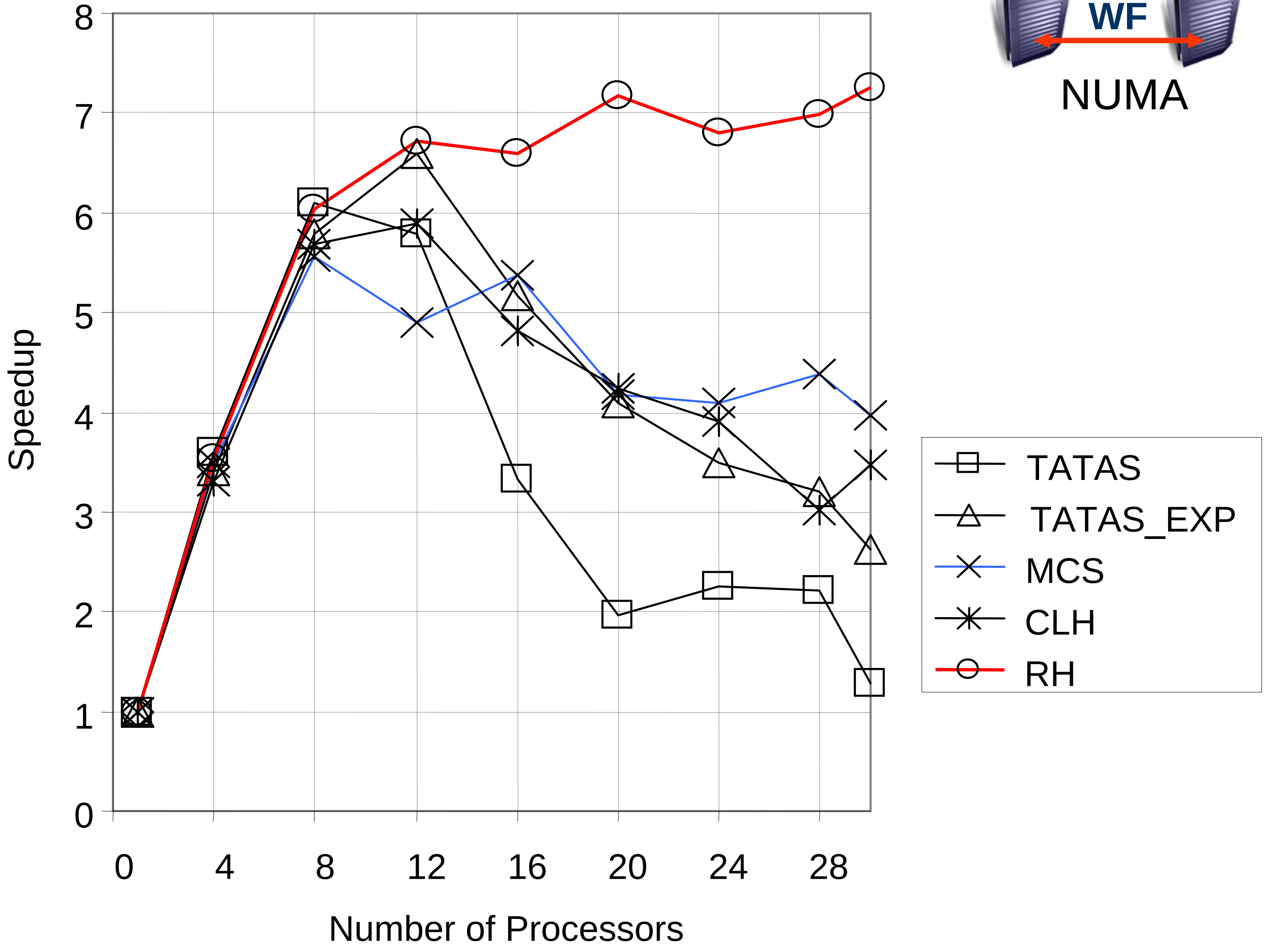
Thanks: Zoran Radovic

Lock Hand-over Time



Application performance

Ex: Raytrace Speedup


Speedup
Number of Processors
TATAS
TATAS_EXP
MCS
CLH
RH