Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University
DARK2 in a nutshell

1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. Multiprocessors (TLP, coherence, memory models, interconnects, scalability, ...)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, ...)
4. Future: (physical limitations, TLP+ILP in the CPU, ...)

DARK 2008
The era of the "Rocket Science Supercomputers" 1980-1995

- The one with the most blinking lights wins
- The one with the niftiest language wins
- The more different the better!
MP Taxonomy (more later...)

SIMD

Message-passing

MIMD

Shared Memory

Fine-grained

Coarse-grained

UMA  NUMA  COMA  For now!
Models of parallelism

- Processes (**fork** or **&** in UNIX)
  - A parallel execution, where each process has its own process state, e.g., memory mapping
- Threads (**thread_create** in POSIX)
  - Parallel threads of control inside a process
  - There are some thread-shared state, e.g., memory mappings.
- Sverker will tell you more...
Programming Model:

Shared Memory

Thr  Thr  Thr  Thr  Thr  Thr  Thr  Thread
Adding Caches: More Concurrency

Shared Memory

Threads connected to shared memory.
Caches: Automatic Replication of Data

Shared Memory

Thread
Read A
Read A
...
...
Read A

Thread
Read A
...
...
Read A

Thread
Read B
...
...
Read A
The Cache Coherent Memory System

Shared Memory

A: $\ldots$ Read A
   Read A
   ... 
   ... 

B: $\ldots$ Read A
   Read B
   ... 
   ... 
   Write A

INV

Thread

Thread

Thread
The Cache Coherent $2$
Summing up Coherence

There can be many copies of a datum, but only one value.

Too strong definition!

There is a single global order of value changes to each datum.
Implementation options for memory coherence

- Two coherence options
  - Snoop-based (“broadcast”)
  - Directory-based (“point to point”)
- Different memory models
- Varying scalability
Snoop-based Protocol Implementation

Shared Memory

"BUS"

BUS snoop

Cache

A-tag State Data

CPU access

CPU

BUS snoop

CPU access

CPU

Bus transaction

DARK 2008

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh
**Example: Bus Snoop MOSI**

**BUSrts: ReadtoShare** (reading the data with the intention to read it)

**BUSrtw, ReadToWrite** (reading the data with the intention to modify it)

**BUSwb:** Writing data back to memory

**BUSinv:** Invalidating other caches copies
Snoop-based Protocol Implementation

Shared Memory

BUS

CPU access

Cache

A-tag | State | Data
--- | --- | ---

BUS snoop

CPU

BUS snoop

CPU

Bus transaction
Example: CPU access MOSI

CPUwrite: Caused by a store miss
CPUread  Caused by a loadmiss
CPUrepl: Caused by a replacement

Diagram:

- CPUwrite/BUSrtw
- CPUrepl/BUSwb
- CPUread/BUSrts
- CPUwrite/BUSinv
- CPUrepl/BUSwb
- CPUwrite/BUSinv
- CPUread/-
"Upgrade" in snoop-based

A: 

B: 

Read A
Read A
...
...

Write A

Read B
...
Read A

Thread
Thread
Thread

BusINV

Have to INV

My INV

Have to INV

$ $ $
Directory-based coherence: per-cacheline info in the memory
"Upgrade" in dir-based
Cache-to-cache in dir-based

Who has a copy

Thread

Read A
Read A
...
...
Read A

Thread

Read A
...
Read A
Write A

Thread

Read B
...
Read A

ReadRequest

ReadDemand

Ack

Forward

network
Cache-to-cache in snoop-based

A: Read A
   Read A
   ...
   ...
   Read A

B: Read B
   ...
   Read A

BusRTS

My RTS → wait for data

Gotta answer

Thread

Read A

Thread

Write A

Thread

Read B

...
A New Kind of Cache Miss

- Capacity – too small cache
- Conflict – limited associativity
- Compulsory – accessing data the first time

Communication (or "Coherence") [Jouppi]
  - Caused by downgrade (modified $\rightarrow$ shared)
    "A store to data I had in state M, but now it’s in state S"
  - Caused my invalidation (shared $\rightarrow$ invalid)
    "A load to data I had in state S, but now it’s been invalidated"
Why snoop?

- A “bus”: a serialization point helps coherence and memory ordering
- Upgrade is faster [producer/ consumer and migratory sharing]
- Cache-to-cache is much faster [i.e., communication...]
- Synchronization, a combination of both
- ...but it is hard to scale the bandwidth 😞
Why directory-based

- P2P messages $\rightarrow$ **high** bandwidth
- Suits out-of-the-box coherence
- Note:
  - Dir-based can be used to build a uniform-memory architecture (UMA)
  - Bandwidth will be great!!
  - Memory latency will be OK
  - Cache-to-cache latency will not!
Update Instead of Invalidate?

- Write the new value to the other caches holding a shared copy (instead of invalidating...)
- Will avoid coherence misses
- Consumes a large amount of bandwidth
- Hard to implement strong coherence
- Few implementations: SPARCCenter2000, Xerox Dragon
Update in MOSI snoop-based

Thread

Read A
Read A
...
Read A

Thread

Read A
...
Write A

Thread

Read B
...
Read A

BusUpdate

Have to Update

My Update

Have to Update

$
Implementing Coherence (and Memory Models...)

Erik Hagersten
Uppsala University
Sweden
Snoop-based Protocol Implementation

Shared Memory

"BUS"

Cache

<table>
<thead>
<tr>
<th>A-tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
</table>

CPU access

CPU

Bus transaction

BUS snoop
Common Cache States

- **M – Modified**
  My dirty copy is the only cached copy

- **E – Exclusive**
  My clean copy is the only cached copy

- **O – Owner**
  I have a dirty copy, others may also have a copy

- **S – Shared**
  I have a clean copy, others may also have a copy

- **I – Invalid**
  I have no valid copy in my cache
Some Coherence Alternative

- MSI
  - Writeback to memory on a cache2cache.

- MOSI
  - Leave one dirty copy in a cache on a cache2cache

- MOESI
  - The first reader will go to E and can later write cheaply
The Cache Coherent Memory System

Shared Memory

Read A
Read A
...
...

Thread

INV

Read A
...
...
Write A

Thread

INV

Read B
...
...

Thread
Upgrade – the requesting CPU

CPUwrite: Caused by a store miss
CPUread: Caused by a loadmiss
CPUrepl: Caused by a replacement
**Upgrade – the other CPUs**

**BUSrts**: ReadToShare (reading the data with the intention to read it)

**BUSrtw, ReadToWrite** (reading the data with the intention to modify it)

**BUSwb**: Writing data back to memory

**BUSinv**: Invalidating other caches copies
Modern snoop-based architecture -- dual tags

Shared Memory

BUS

Snoop Tag (Obligatrion state)
(possibly time-sliced access to cache tags)

Access Tag (Permission state)
(possibly time-sliced access to cache tags)

Cache

CPU

BUS snoops

Bus transaction

Cache access

A-tag State

Data
"Upgrade" in snooped-based
The Cache Coherent Cache-to-cache

Shared Memory

Thread
- Read A
- Read A
- ...
- ...

Thread
- Read B
- ...
- Read A

Thread
- Read A
- ...
- Write A

© Erik Hagersten | user.it.uu.se/∼eh
Cache2cache – the requesting CPU

**CPUwrite**: Caused by a store miss

**CPUread**: Caused by a load miss

**CPUrepl**: Caused by a replacement
**BUSrts**: **ReadToShare** (reading the data with the intention to read it)

**BUSrtw, ReadToWrite** (reading the data with the intention to modify it)

**BUSwb**: Writing data back to memory

**BUSinv**: Invalidating other caches copies

---

**Cache-to-cache – the other CPU**

- BUSrts
- BUSrtw
- BUSinv
- BUSwb

Diagram showing interactions between different components within a cache-to-cache system.
Cache-to-cache in snoope-based

Shared Memory

BUS snoop

CPU: load

Gotta’ wait here for data

Cache access

A-tag

State

Data

I

M

CPB

MyRTS

S

O

A-tag

I

O

39
**Yet Another Cache-to-cache**

**BUSrts:** ReadtoShare (reading the data with the intention to read it)

**BUSrtw, ReadToWrite** (reading the data with the intention to modify it)

**BUSwb:** Writing data back to memory

**BUSinv:** Invalidating other caches copies
All the three RISC CPUs in a MOSI shared-memory sequentially consistent multiprocessor executes the following code almost at the same time:

```c
while(A != my_id){};  /* this is a primitive kind of lock */
B := B + A * 2;
A := A + 1;             /* this is a primitive kind of unlock */
while (A != 4) {};      /* this is a primitive kind of barrier*/
<after a long time>
<some other execution replaces A and B from the caches, if still present>
```

Initially, CPU1 has its local variable my_id=1, CPU has my_id=2 and CPU3 has my_id=3 and the globally shared variables A is equal to 1 and B is equal to 0. CPU2 and 3 are starting slightly ahead of CPU1 and will execute the first while statement before CPU1. Initially, both A and B only reside in memory.

The following four bus transaction types can be seen on the snooping bus connecting the CPUs:

- **RTS:** ReadToShare (reading the data with the intention to read it)
- **RTW:** ReadToWrite (reading the data with the intention to modify it)
- **WB:** Writing data back to memory
- **INV:** Invalidating other caches copies

Show every state change and/or value change of A and B in each CPU’s cache according to one possible interleaving of the memory accesses. After the parallel execution is done for all of the CPUs, the cache lines still in the caches will be replaced. These actions should also be shown. For each line, also state what bus transaction occurs on the bus (if any) as well as which device is providing the corresponding data (if any).
Example of a state transition sheet:

<table>
<thead>
<tr>
<th>CPU action</th>
<th>Bus Transaction (if any)</th>
<th>State/value after the CPU action</th>
<th>Data is provided by [CPU 1, 2, 3 or Mem] (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU1 A B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU2 A B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU3 A B</td>
<td></td>
</tr>
<tr>
<td>Initially</td>
<td>I I I I I I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1: LD A</td>
<td>RTS(A)</td>
<td>S/1</td>
<td>Mem</td>
</tr>
<tr>
<td>CPU2: LD B</td>
<td>RTS(B)</td>
<td>S/0</td>
<td>Mem</td>
</tr>
<tr>
<td>…some time elapses .</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1: replace A</td>
<td>- I</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>CPU2: replace B</td>
<td>- I</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
False sharing

Communication misses even though the threads do not share data “the cache line is too large”
Memory Ordering (aka Memory Consistency) -- tricky but important stuff

Erik Hagersten
Uppsala University
Sweden
Memory Ordering

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

Initially $A = B = 0$

```
...  
A := 1
...  
While (A==0) {}  
B := 1
...  
While (B==0) {}  
Read A
...  
...  
...  
Print A
```
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
- Is a “contract” between the HW and SW guys
- Without it, you can not say much about the result of a parallel execution
In which order were these threads executed?

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

Thread 1

LD A
ST B’
LD C
ST D’
LD E
...
...

Thread 2

ST A’
LD B’
ST C’
LD D
ST E’
...
...

(LD A happened before ST A’)

(A denotes a modified value to the data at addr A)
One possible observed order

Thread 1
- LD A
- ST B'
- LD C
- ST D'
- LD E
- ...
- ...

Thread 2
- LD A
- ST B'
- LD C
- ST D'
- LD E
- ...
- ...

Another possible observed order

Thread 1
- LD A
- ST B'
- LD C
- ST D'
- LD E
- ...
- ...

Thread 2
- LD A
- ST B'
- LD C
- ST D'
- LD E
- ...
- ...
“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”

\[ A := 1 \]
\[ \text{if } (B == 0) \text{ print(“A won“)} \]

\[ B := 1 \]
\[ \text{if } (A == 0) \text{ print(“B won“)} \]

Q: Is it possible that both \( A \) and \( B \) win?
Sequential Consistency (SC) Violation → Dekker: both wins

A := B := 0

A := 1

If (B == 0)
  print “Left wins”

B := 1

If (A == 0)
  print “Right wins”

Both Left and Right wins → SC violation

A := B := 0

ST A, 1
LD B → 0
ST B, 1
LD A → 0

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

Acess graph

c

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

a

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

d

Not SC

= PO: Program order: a < b
(the order specified by the program)
SC is OK if one thread wins

Only Right wins $\Rightarrow$ SC is OK

A := B := 0

A := 1

If (B == 0)
print “Left wins”

B := 1

If (A == 0)
print “Right wins”

Not cyclic graph $\Rightarrow$ SC

One global order:
ST B, 1 $\Rightarrow$ ST A, 1 $\Rightarrow$ LD B $\Rightarrow$ 1 $\Rightarrow$ LD A $\Rightarrow$ 0

STB < LDA < STA <LDB
SC is OK if no thread wins

No thread wins \(\implies\) SC is OK

A := B := 0

If (B == 0)
print “Left wins”

B := 1

If (A == 0)
print “Right wins”

A := B := 0

ST A, 1
LD B \(\rightarrow\) 1
ST B, 1
LD A \(\rightarrow\) 1

Not cyclic graph \(\implies\) SC

Four Partial Orders, still SC
STB < LDA ;   STA < LDA;   STB < LDB ;   STA < LDA
One implementation of SC in dir-based (....without speculation)

Thread
Read A
Read A
...
...

Who has a copy

Thread
Read A
Read X
...
Write A
Read C

Thread
Read X
Read A

A:Who has a copy

B:Who has a copy

Inv

Ack

Ack

Ack

Ack

Must receive all ACKs before continuing

Read X must complete before starting Read A
“Almost intuitive memory model”
Total Store Ordering (P. Sindhu)

- Global order achieved by *interleaving all* store accesses from different threads
- “Programmer’s intuition is maintained”
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive $\implies$ performance penalty
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 $A = B = 0$

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

**Q:** What value will get printed?  
**Answer:** 1
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 $B$ 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 only established when the programmer explicitly inserts memory barrier instructions
  + Better performance!!
  --- Interesting bugs!!
Weak/Release consistency

- New flag synchronization needed
  
  \[
  A := \text{data}; \quad \text{while} \ (\text{flag} \neq 1) \ \{} ; \\
  \text{membarrier}; \quad \text{membarrier}; \\
  \text{flag} := 1; \quad X := A;
  \]

- Dekker’s: same as TSO

- Causal correctness provided for this code

Initially \( A = B = 0 \)

- \( A := 1 \)
- \( B := 1 \)
- \( \text{membarrier} \)
- \( \text{membarrier} \)

Q: What value will get printed?

Answer: 1
Example 1: Causal Correctness Issues

A: Read A
A:=1
...

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

What is the value of A?
Example 1: Causal Correctness Issues

Shared Memory

Thread
Read A
A := 1
...
Write B
While (A == 0) {} 
B := 1
Read A
While (B == 0) {} 
Print A
Example 1: Causal Correctness Issues

Shared Memory

Thread

Read A
A:=1
...

Thread

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

Thread

Read A
...
While (B==0) {}
Print A
Example 1: Causal Correctness Issues

Shared Memory

Thread

Thread

Thread

INV

READ

A:

B:

Read A
A:=1
...

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

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

DARK 2008
Example 1: Causal Correctness Issues

Shared Memory

Thread

Read A
A := 1
...

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

Thread

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

INV
Example 1: Causal Correctness Issues

Shared Memory

Thread
Read A
A := 1
...
Write B
...
While (A == 0) {}
B := 1
Thread
Read A
...
While (B == 0) {}
Print A
Example 1: Causal Correctness Issues

Shared Memory

A: if store causality \( \Rightarrow \) ”1” will be printed

Thread

Read A
A:=1
...

Thread

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

Thread

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

What is the value of A?
It depends...
Dekker’s Algorithm

Initially $A = B = 0$

“fork”

$A := 1$

if $(B == 0)$ print(“A won”)

$B := 1$

if $(A == 0)$ print(“B won”)

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

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

A: Only known if you know the memory model
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.
Synchronization

Erik Hagersten
Uppsala University
Sweden
while (sum < N) {
    sum := sum + 1
}

PSEUDO ASM CODE

LOOP:
LD R1, N
LD R2, sum
SUB R1, R1, R2
BGZ R3, CONT:
ADD R2, R2, #1
ST R2, sum
BR LOOP:
CONT:

printf (sum)

Need to introduce synchronization

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

\[
\text{LOCK}(\text{lock\_variable}) \quad /* \text{wait for your turn} */ \\
\text{if (sum > threshold)} \{ \\
\quad \text{sum := my\_sum + sum} \\
\} \\
\text{UNLOCK}(\text{lock\_variable}) \quad /* \text{release the lock} */ \\
\]

Critical Section

if (sum > threshold) {
    \text{LOCK}(\text{lock\_variable}) \quad /* \text{wait for your turn} */ \\
    \text{sum := my\_sum + sum} \\
    \text{UNLOCK}(\text{lock\_variable}) \quad /* \text{release the lock} */ \\
}
Components of a Synchronization Event

- Acquire method
  - Acquire right to the synch (enter critical section, go past event)
- Waiting algorithm
  - Wait for synch to become available when it isn’t
- Release method
  - Enable other processors to acquire right to the synch
Atomic Instruction to Acquire

Atomic example: test&set “TAS” (SPARC: LDSTB)
- The value at Mem(lock_addr) loaded into the specified register
- Constant “1” atomically stored into Mem(lock_addr) (SPARC: “FF”)
- Software can determine if won (i.e., set changed the value from 0 to 1)
- Other constants could be used instead of 1 and 0

Looks like a store instruction to the caches/memory system

Implementation:
1. Get an exclusive copy of the cache line
2. Make the atomic modification to the cached copy

Other read-modify-write primitives can be used too
- Swap (SWAP): atomically swap the value of REG with Mem(lock_addr)
- Compare&swap (CAS): SWAP if Mem(lock_addr)==REG2
Waiting Algorithms

Blocking
- Waiting processes/threads are de-scheduled
- High overhead
- Allows processor to do other things

Busy-waiting
- Waiting processes repeatedly test a lock_variable until it changes value
- Releasing process sets the lock_variable
- Lower overhead, but consumes processor resources
- Can cause network traffic

Hybrid methods: busy-wait a while, then block
Release Algorithm

- Typically just a store “0”
- More complicated locks may require a conditional store or a “wake-up”. 
A Bad Example: "POUNDING"

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

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

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

Generates too much traffic!!
-- spinning threads produce traffic!
Optimistic Test&Set Lock “spinlock”

```plaintext
proc lock(lock_variable) {
    while true {
        if (TAS[lock_variable] == 0) break; /* bang on the lock once, done if TAS==0 */
        while(lock_variable != 0) {} /* spin locally in your cache until "0" observed*/
    }
}

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

Much less coherence traffic!!
-- still lots of traffic at lock handover!
It could still get messy!

\[ \begin{array}{c}
\text{L} = 1 \\
\text{L} := 0 \\
\text{N reads} \\
\text{L} = 0
\end{array} \]

\[ \text{Interconnect} \]

\[ \begin{array}{ccccccccc}
\circ & \circ & \text{CS} & \circ & \circ & \circ & \circ & \ldots & \circ \\
\times & \times & \text{L} := 0 & \times & \times & \times & \times & \ldots & \times \\
\text{L} = 0 & \text{L} = 0 & \text{L} = 0 & \text{L} = 0 & \text{L} = 0 & \text{L} = 0 & \ldots & \text{L} = 0
\end{array} \]
...messy (part 2)

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

L:=0

Interconnect

T&S T&S T&S T&S T&S ... T&S

L== 1

Interconnect

CS T&T T&T L:=0 L:=0 ... L:=0

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
Fix1: Queue-based locks
Could Get Even Worse on a NUMA

- Poor communication latency
- Serialization of accesses to the same cache line
- WF: added hardware optimization:
  - TAS can bypass loads in the coherence protocol
  - \( \Rightarrow \text{N-2 loads queue up in the protocol} \)
  - \( \Rightarrow \text{the winner’s atomic TAS will bypass the loads} \)
  - \( \Rightarrow \text{the loads will return “busy”} \)
Ticket-based queue locks: “ticket”

```c
proc lock(lstruct) {
    int my_num;
    my_num := INC(lstruct.ticket) /* get your unique number*/
    while(my_num != lstruct.nowserving) {} /* wait here for your turn */
}

proc unlock(lstruct) {
    lstruct.nowserving++ /* next in line please */
}
```

Less traffic at lock handover!
Ticket-based back-off “TBO”

```c
proc lock(lstruct) {
    int my_num;
    my_num := INC(lstruct.ticket) /* get your number*/
    while(my_num != lstruct.nowserving) {
        idle_wait(lstruct.nowserving - my_num) /* my turn ?*/
        idle_wait(lstruct.nowserving - my_num) /* do other shopping */
    }
}

proc unlock(lock_struct) {
    lock_struct.nowserving++ /* next in line please */
}
```

Even less traffic at lock handover!
Queue-based lock: CLH-lock
-- a variation of the MCS lock

"Initially, each process owns one global flag, pointed to by private *I and *P
The global lock is pointed to by global *L
1) Initialize the *I flag to busy (= "1")
2) Atomically, make *L point our "flag" and make *P point where *L pointed
3) Wait until *P points to a "0"

**lock**

```
proc lock(int **L, **I, **P)
{   **I =1;  /*initialized as “busy”*/
    atomic_swap {*P =*L; *L=*P}
    /* P now stores a pointer to the flag L pointed to*/
    /* L now stores a pointer to our flag */
    while (**P != 0){} } /* keep spinning until prev owner releases lock*/
```

**unlock**

```
proc unlock(int **I, **P)
{   **I =0;  /* release the lock */
    *I =*P; } /* next time *I to reuse the previous guys flag*/
```
CLH lock

```c
proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap {*P = *L; *L = *P}
    /* *L now points to our I* */
    while (**P != 0){}  }
/* spin unit prev is done */
```
proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap {*P = *L; *L = *P}
    /* *L now point to our I */ */
    while (**P != 0) {} }
/* spin unit prev is done */
```c
proc lock(int **L, **I, **P)
{
    **I = 1 /* init to “busy”*/
    atomic_swap {*P = *L; *L = *P}
    /* *L now point to our I */
    while (**P != 0){} }
/* spin unit prev is done */
```
proc lock(int **L, **I, **P)
{
    **I = 1 /* init to “busy”*/
    atomic_swap {*P = *L; *L = *P}
    /* *L now point to our I */ */
    while (**P != 0) {};
    /* spin unit prev is done */
}
proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap {*P =*L; *L=*P}
    /* *L now point to our I*/
    while (**P != 0){} }
/* spin unit prev is done */
proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic swap {*P = *L; *L = *P;}
    while (**P != 0){} /* spin unit prev is done */
}

proc lock(int **L, **I, **P)
{
    **I = 1 /* init to “busy”*/
    atomic_swap {*P =*L; *L=*P;}
    /* *L now point to our I*/
    while (**P != 0){} 
}
/* spin unit prev is done */

In CS

while *P

while *P

while *P
proc unlock(int **I, **P)
{
    **I = 0;
    /* release the lock */
    *I = *P; }
/* reuse the previous guy's *P*/
proc unlock(int **I, **P)
{
    **I = 0;
    /* release the lock */
    *I = *P; }
    /* reuse the previous guy’s *P*/
proc unlock(int **I, **P)
{
    **I = 0;
    *I = *P;
}
In CS

Minimizes traffic at lock handover! May be too fair for NUMAs ....

```c
proc unlock(int **I, **P)
{
    **I = 0;
    *I = *P;
}
```
E6800 locks 12 CPUs

![Graph showing the average time per CS work for different contention levels. The x-axis represents the number of contenders (1 to 12), and the y-axis represents the time in a logarithmic scale (from 1 to 60,000). The graph compares the performance of POUND, SPIN, TICKET, TBO, and CLH methods. The POUND method shows a significant increase in time as the number of contenders increases, while the other methods remain relatively flat.]
E6800 locks (excluding POUND)

Avg. time per CS job

# Contenders

0 500 1,000 1,500 2,000 2,500 3,000

1 2 3 4 5 6 7 8 9 10 11 12

SPIN
TICKET
TBO
CLH
NUMA:

NUCA: Non-uniform Comm Arch.

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

Snoop

WF
Trad. chart over lock performance on a hierarchical NUMA (round robin scheduling)

Benchmark:

for i = 1 to 10000 {
    lock(AL)
    A := A + 1;
    unlock(AL)
}
Introducing RH locks

Benchmark:

```plaintext
for i = 1 to 10000 {
  lock(AL)
  A := A + 1;
  unlock(AL)
}```
RH locks: encourages unfairness

Time per lock handover

Node migration (%)

- spin
- spin_exp
- MCS-queue
- CLH-queue
- RH-locks

Processors

Time/Processors [seconds]

0,00 0,05 0,10 0,15 0,20 0,25 0,30 0,35 0,40 0,45 0,50

Processors

Node-handoffs [%]

0 10 20 30 40 50 60 70 80 90 100

© Erik Hagersten | user.it.uu.se/~eh
Ex: Splash Raytrace Application Speedup

HBO@HPCA 2003
RH@SC 2002

Number of Processors vs. Speedup

- SPIN
- SPIN_EXP
- MCS
- CLH
- RH
Performance under contention

- TestAndSet
- ExpBackoff
- Queuebased
- RH

Time between entering into CS vs Contention
Barriers: Make the first threads wait for the last thread to reach a point in the program

1. Software algorithms implemented using locks, flags, counters

2. Hardware barriers
   - Wired-AND line separate from address/data bus
   - Set input high when arrive, wait for output to be high to leave
   - (In practice, multiple wires to allow reuse)
   - Difficult to support arbitrary subset of processors
     - even harder with multiple processes per processor
   - Difficult to dynamically change number and identity of participants
     - e.g. latter due to process migration
A Centralized Barrier

BARRIER (bar_name, p) {
    int loops;
    loops = 0;

    local_sense = !(local_sense); /* toggle private sense variable each time the barrier is used */

    LOCK(bar_name.lock);
    bar_name.counter++; /* globally increment the barrier count */
    if (bar_name.counter == p) { /* everybody here yet? */
        bar_name.flag = local_sense; /* release waiters*/
        UNLOCK(bar_name.lock)
    } else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) { /* wait for the last guy */
            if (loops++ > UNREASONABLE) report_warning(pid)}
    }
}
Centralized Barrier Performance

- Latency
  - Want short critical path in barrier
  - Centralized has critical path length at least proportional to $p$

- Traffic
  - Barriers likely to be highly contended, so want traffic to scale well
  - About $3p$ bus transactions in centralized

- Storage Cost
  - Very low: centralized counter and flag

- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node

→ Hierarchical barriers
New kind of synchronization: Transactional Memory

- OLD: lock(ID); unlock(ID) around critical sections
- NEW: start_transaction; end_transaction around "critical sections" (note: no ID!!)
  - Underlying mechanism to guarantee atomic behavior
  - This is not the same as guaranteeing that only one thread is in the critical action!!
  - Supported in HW (soon) or in SW (normally very inefficient)
- Suggested by Maurice Herlihy in 1993
- HW support announced by Sun and Intel (??)
Support for TM

- **Start_transaction:**
  - Save original state to allow for rollback (i.e., save register values)

- **In critical section**
  - Do not make any global state change
  - Detect “atomic violations” (others writing data you’ve read in CS or reading data you have written)
  - At atomic violation: roll-back to original state
  - Forward progress must be guaranteed

- **End_transaction**
  - Atomically commit all changes performed in the critical section.
Advantage of TM

- Do not have to "name" CS
- Less risk for deadlocks
- Performance:
  - Several threads can be in "the same" CS as long as they do not mess with each other
  - CS can often be large with a small performance penalty
Introduction to Multiprocessors

Erik Hagersten
Uppsala University
Flynn’s Taxonomy

\{Single,Multiple\}Instruction + \{Single,Multiple\}Data

- SISD – Our good old “simple” CPUs
- SIMD – Vectors, “MMX”, DSPs, CM-2,…
- MIMD – TLP, cluster, shared-mem MP,…
- MISD – Can’t think of any…
MP Taxonomy

- SIMD
  - Message-passing
    - Fine-grained
    - Coarse-grained
  - Shared Memory
    - UMA
    - NUMA
    - COMA

DARK 2008
**SIMD** = “Dataparallelism”
SIMD: Thinking Machine

- Connection Machine: CM1, CM2, CM200
  (at KTH ~1990: CM200 “Bellman”)
- One-bit ALU and a small local memory
- FP accelerator available
- Programmed in “ASM”, *C and *Lisp
- Hard to program (in my opinion...)

© Erik Hagersten | user.it.uu.se/~eh
Other Flavors of SIMD

- MMX/AltiVec/VIS instructions/SSE...
  - Divide register content into smaller items (e.g., bytes)
  - Special instructions operate on all items in parallel, e.g., BYTE-COMPARE...
- Some DSPs (Digital Signal Processors)
- Some Image Processors
Vector architectures
CRAY, NEC, Fujitsu, ...

- Vectory Processors
  - LD/ST operate on vectors of data
  - ALU Ops operate on vectors of data

- Example:
  - 8 “vector register” contains 64 vector entries each
  - A single LD/ST instr loads/stores entire vectors
  - A single ALU instr V1 ← V2 op V3
  - 64 bit mask vectors make execution conditional
  - Overlaps Mem and ALU ops
  - One form of “SIMD” -- Single Instruction Multiple Data
MIMD: Message-passing

- SIMD
- MIMD
  - Message-passing
    - Fine-grained
    - Coarse-grained
  - Shared Memory
    - UMA
    - NUMA
    - COMA
Message-passing Arch
MIMD

Explicit Messages
Message-Passing HW

- Programmed in MPI or PVM (or HPFortran...)
  Thinking Machines: CM5
  Intel: Paragon
  IBM: SP2
  Meiko (Bristol, UK!!): CS2
  Today: Clusters with high-speed interconnect
  (Important today, but not covered in this course)

- Clusters can be used as message-passing HW, put is most often used as capacity computing (i.e., throughput computing)
Dataflow

- Often programmed in functional languages (e.g., ID)
- Compile program to Dataflow graph
- Operands + graph = executable
- Operation ready when the source operands are available
Dataflow Example:

\[ X := A + B \]
\[ Y := C + D \]
If \((X > Y)\)
  output \(X\)
else
  output \(Y\)
Static Dataflow (Dennis)

\[ X := A + B \]
\[ Y := C + D \]
If \((X > Y)\)
output \(X\)
else
output \(Y\)

Each operand executed exactly once per program
Location assigned for each input data
Fine-grained Message-passing
Dataflow ==> Multithreading

Implicit Messages (Tokens)
Dynamic Dataflow (Arvind)

- Allows for recursion and loops
- Each invocation is assigned a “color”
- Pairs of operands are matched dynamically
  - Based on \{Color, Operation\}
  - In the Waiting-Matching Section (I.e., a cache)

- One problem: too much parallelism in the wrong place
Carlstedts Elektornik
Gunnar Carlstedt, Staffan Truve’ et al

- Processor “8601”
  - Gothenburg 1990-1997
  - Functional language “H”
  - Execution performed by a reduction a CAM memory
  - ALU rarely used
  - Many parallel processors on a wafer (Wafer-scale integration)

⇒ CRT (Carlstedt Research Technology)
Today’s Topic

SIMD

MIMD

Message-passing

Shared Memory

Fine-grained

Coarse-grained

UMA

NUMA

COMA
The server market 1995

<table>
<thead>
<tr>
<th>Server Size</th>
<th>High-Perf. Computing</th>
<th>Commercial Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; $10k</td>
<td>1%</td>
<td>19%</td>
</tr>
<tr>
<td>&lt; $50k</td>
<td>5%</td>
<td>21%</td>
</tr>
<tr>
<td>&lt; $250k</td>
<td>5%</td>
<td>21%</td>
</tr>
<tr>
<td>&lt; $1M</td>
<td>2%</td>
<td>9%</td>
</tr>
<tr>
<td>&gt; $1M</td>
<td>3%</td>
<td>8%</td>
</tr>
</tbody>
</table>

The target of the rocket science supercomputers

UNIX shared-mem servers