

Lecture 5

Distributed shared memory

[Culler et al.'98], [PPP 4.7]

Overview, terminology

Cache coherence and memory consistency

Cache coherence protocols

False Sharing

Shared memory consistency models

Software DSM

[Gharachorloo/Adve'96]

Caches in CC-NUMA architectures

Cache = small, fast memory (SRAM) between processor and main memory
contains copies of main memory words

cache hit = accessed word already in cache, get it fast.

cache miss = not in cache, load from main memory (slower)

Cache line size: from 16 bytes (Dash) ...

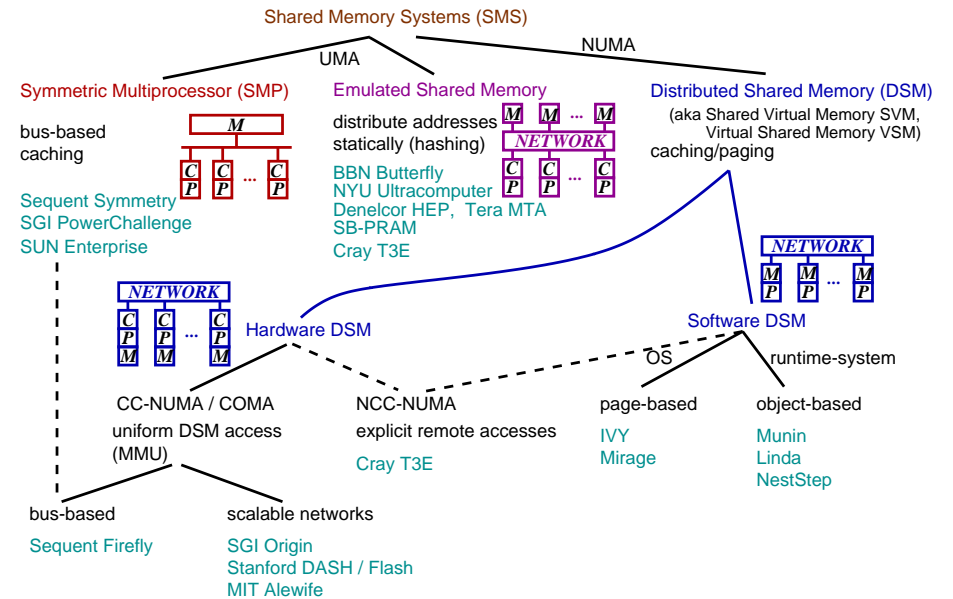
Memory page size: ... up to 8 KB (Mermaid)

Cache-based systems profit from

- + spatial access locality (access also other data in same cache line)
- + temporal access locality (access same location multiple times)
- + dynamic adaptivity of cache contents

→ suitable for applications with high (also dynamic) data locality

DSM Overview



Cache issues (1)

Mapping memory blocks → cache lines / page frames:

direct mapped: $\forall j \exists ! i : B_j \mapsto C_i$, namely where $i \equiv j \pmod m$.

fully-associative: any memory block may be placed in any cache line
set-associative

Replacement strategies (for fully- and set-associative caches)

- LRU least-recently used
- LFU least-frequently used

...

Cache issues (2): Memory update strategies

Write-through

- + consistency
- slow, write stall (→ write buffer)

Write-back

- + update only cache entry
- + write back to memory only when replacing cache line
- + write only if modified, marked by “dirty” bit for each C_i
- not consistent,
 - DMA access (I/O, other procs) may access stale values
 - must be protected by OS, write back on request

Cache coherence – formal definition

what does “most recent write access” to x mean?

Formally, 3 conditions must be fulfilled for **coherence**:

- (a) Each processor sees its *own* writes and reads in program order.

P_1 writes v to x at time t_1 , reads from x at $t_2 > t_1$,
 no other processor writes to x between t_1 and t_2
 → read yields v

- (b) The written value is eventually visible to all processors.

P_1 writes to l at t_1 , P_2 reads from l at $t_2 > t_1$,
 no other processor writes to l between t_1 and t_2 ,
 and $t_2 - t_1$ sufficiently large, then P_2 reads x .

- (c) All processors see one total order of all write accesses.
 (*total store ordering*)

Cache coherence and Memory consistency

Caching of (shared) variables leads to **consistency problems**.

A cache management system is called **coherent**

if a read access to a (shared) memory location x reproduces always the value corresponding to the most recent write access to x .

→ no access to **stale** values

A memory system is **consistent** (at a certain time)

if all copies of shared variables in the main memory and in the caches are identical.

Permanent cache-consistency implies cache-coherence.

Cache coherence protocols

Inconsistencies occur when modifying only the copy of a shared variable in a cache, not in the main memory and all other caches where it is held.

Write-update protocol

At a write access, all other copies in the system must be updated as well. Updating must be finished before the next access.

Write-invalidate protocol

Before modifying a copy in a cache, all other copies in the system must be declared as “invalid”.

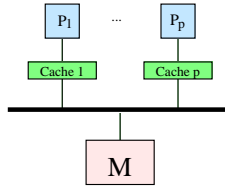
Most cache-based SMPs use a write-invalidate protocol.

Updating / invalidating straightforward in bus-based systems (**bus-snooping**) otherwise, a **directory mechanism** is necessary

Bus-Snooping

For bus-based SMP with caches and write-through strategy.

All relevant memory accesses go via the central bus.



Cache-controller of each processor listens to addresses on the bus:

write access to main memory is recognized and committed to the own cache.

- bus is performance bottleneck → poor scalability

→ Exercise

MSI-protocol: State transitions

State transitions:

triggered by bus operations and local processor reads/writes

Bus read (BusRd)

read access caused a cache miss

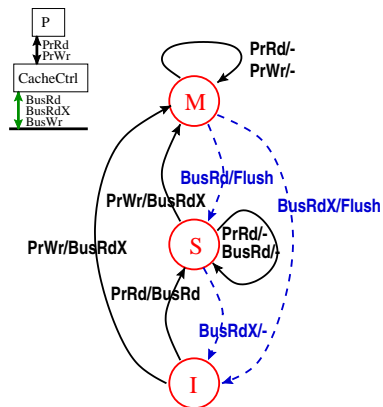
Bus read exclusive (BusRdX)

write attempt to non-modifiable copy → must invalidate other copies

Write back (BusWr), due to replacement

Processor reads (PrRd)

Processor writes (PrWr)



Processor operation / Cache controller operation

Observed operation / Cache controller operation

Flush = desired value put on the bus

Missing edges - no change of state

Write-back invalidation protocol (MSI protocol)

A block held in cache has one of 3 states:

M (modified)

only this cache entry is valid, all other copies + MM location are not.

S (shared)

cached on one or more processors, all copies are valid.

I (invalid)

this cache entry contains invalid values.

MESI-protocol

MSI protocol:

2 bus operations (BusRd, BusRdX) required

if a processor first reads (→ S), then writes (→ M) a memory location, even if no other processor works on this program.

→ generalization to MESI-protocol with new state

E (exclusive)

no other cache has a copy of this block, and this copy is not modified.

Modifications in MSI-protocol:

+ PrRd to a non-cached address (BusRd): → E (not S)

+ PrWr to E-address: local change to M, write (no bus operation)

+ read access from another processor to E-address (BusRd/Flush): → S

MESI supported by Intel Pentium, MIPS R4400, IBM PowerPC, ...

Directory-protocols for non-bus-based systems

No central medium:

- (a) → no cache coherence (e.g. Cray T3E)
- (b) → directory lookup

Directory keeps the copy set for each memory block

e.g. stored as bitvectors

1 presence bit per processor

status bits

e.g. dirty-bit for the status of the main memory copy

See [Culler'98, Ch. 8]

DSM problem: False sharing (cont.)

How to avoid false sharing?

Smaller cache lines / pages

- false sharing less probable, but
- more administrative effort

Programmer or compiler gives hints for data placement

- more complicated

Time slices for exclusive use:

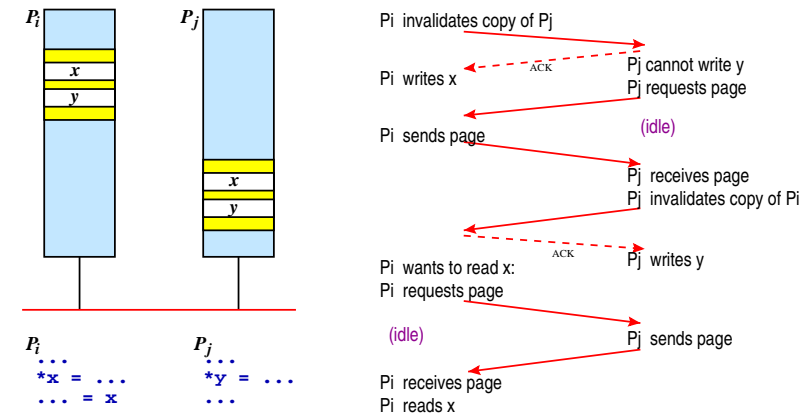
each page stays for $\geq d$ time units at one processor [Mirage](#) [How to partly](#)

avoid the performance penalty of false sharing?

Use a weaker memory consistency model

DSM problem: False sharing

False sharing in cache-/page-based DSM systems



Cache lines / pages treated as units → sequentialization, thrashing

Memory consistency models

Strict consistency

Sequential consistency

Causal consistency

Superstep consistency

“PRAM” consistency

Weak consistency

Release consistency / Barrier consistency

Lazy Release consistency

Entry consistency

Others (processor consistency, total/partial store ordering etc.)

[Culler et al.'98, Ch. 9.1], [Gharachorloo/Adve'96]

Consistency models: Superstep consistency

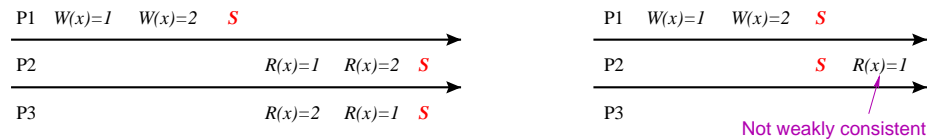
BSP Superstep consistency [K'00], [PPP 6.3]

- + BSP model:
 - Program execution is structured into barrier-separated supersteps.
- + Strict consistency for all shared variables immediately after barrier.
- + No writes are propagated during a superstep
- deterministic

Consistency models: Weak consistency

Weak consistency [Dubois/Scheurich/Briggs'86], see also [PPP 6.3.2.3]

- + Classification of shared variables (and their accesses):
 - synchronization variables** (locks, semaphores)
 - always consistent, atomic access
 - other shared variables**
 - kept consistent by the user, using synchronizations
- + Accesses to synchronization variables are sequentially consistent
- + All pending writes committed before accessing a synchr. variable
- + Synchronization before a read access to obtain most recent value

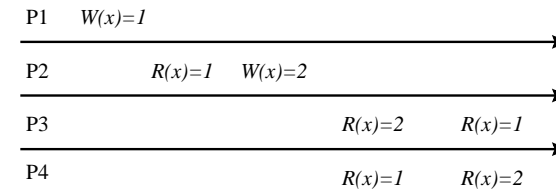


Consistency models: "PRAM" consistency (a.k.a. FIFO-consistency)

"PRAM" (Pipelined RAM) consistency [Lipton/Sandberg'88]

- + Write accesses by a processor P_i are seen by all others in issued order.
- + Write accesses by *different* P_i, P_j may be seen by others in different order.

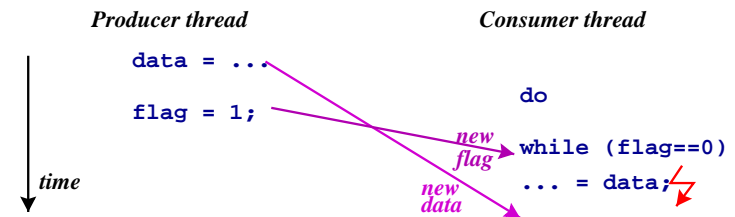
Weaker than causal consistency; writes by P_i can be pipelined (causality of write accesses by different processors is ignored)



Consistency models: Weak consistency in OpenMP

OpenMP implements weak consistency. Inconsistencies may occur due to

- + register allocation
- + compiler optimizations
- + caches with write buffers



Need explicit "memory fence" to control consistency: `flush` directive

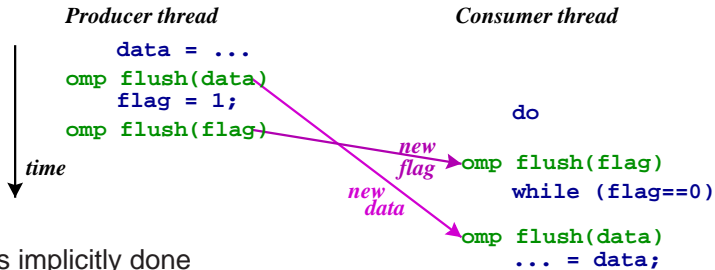
- write back register contents to memory
- forbid code moving compiler optimizations
- flush cache write buffers to memory
- re-read flushed values from memory

Consistency models: Weak consistency in OpenMP

```
!$omp flush ( shvarlist )
```

creates for the executing processor a consistent memory view for the shared variables in *shvarlist*.

If no parameter: create consistency of all accessible shared variables.



A flush is implicitly done at barrier, critical, end critical, end parallel, and at end do, end section, end single if no `nowait` parameter is given

Consistency models: Lazy Release consistency

Lazy Release consistency [Keleher, Cox, Zwaenepoel'92]

Release(*S*) does not commit writes to all copies in the system immediately.

Instead, possibly subsequent Acquire(*S*) by other processor must check (and if necessary, fetch and update) its copy before reading.

+ Saves network traffic:

copies not used in the future are not updated.

Consistency models: Release consistency

Release consistency [Gharachorloo et al.'90], Munin

+ Encapsulate critical section *S* by

Acquire(*S*) – acquiring access to synchronization variable

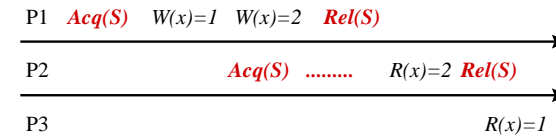
Release(*S*) – releasing access to synchronization variable

+ All pending Acquires of a processor *P_i*

must be finished before accessing a shared variable.

+ All accesses to shared variables must be finished before a Release.

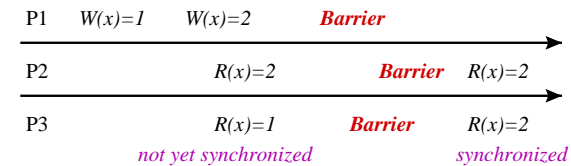
+ Acquire and Release must be “PRAM”-consistent.



Consistency models: Barrier consistency

Barrier-consistency

a special case of Release-consistency: Barrier = Acquire + Release



Consistency models: Entry consistency

Entry consistency [Bershad/Zekauskas/Sawdon'93]

- + Associate shared data (regions/objects) with synchronization variables (this binding may be changed during program execution)
- + Data is only consistent on an acquiring synchronization,
- + and only the data known to be guarded by the acquired object is guaranteed to be consistent.

Software DSM Management

Central Server – non-replicated, non-migrating

- central server may become performance bottleneck
- distribute shared data (by hashing addresses) over multiple servers

Migration – non-replicated, migrating

- susceptible to thrashing

Read-replication – replicated, non-migrating.

- preferably for read-only data

Full replication – replicated, migrating

- sequencer (→ fair lock) used to establish global write FIFO order

Software DSM

Page-based DSM

- emulate (coherent) caches of a CC-NUMA using MMU, OS, runtime system
- single linear address space partitioned into pages of fixed size
- pages may migrate dynamically over the network on demand.

Shared variable based DSM

- manages individual variables
- flexible; more overhead than direct page access
- eliminates false sharing, no data layout problem

Examples: [Munin \[Bennett/Carter/Zwaenepoel'90\]](#), [NestStep \[K.'99\]](#)

Object-based DSM

- manages individual shared objects → more modular; encapsulation
- access only via remote method invocation → synchr. integrated
- no linear address space

Example: [Orca \[Bal et al.'90\]](#), distributed [Linda \[Carriero/Gelernter'89\]](#)

Write-Invalidate Protocol

Implementation: multiple-reader-single-writer sharing

At any time, a data item may either be:

- accessed in **read-only mode** by one or more processors
- read and written (**exclusive mode**) by a single processor

Items in read-only mode can be copied indefinitely to other processes.

Write attempt to read-only data x :

- broadcast invalidation message to all other copies of x
- await acknowledgements before the write can take place
- Any processor attempting to access x are blocked if a writer exists.
- Eventually, control is transferred from the writer
- and other accesses may take place once the update has been sent.

→ all accesses to x processed on first-come-first-served basis.

Achieves sequential consistency.

Write-invalidate protocol (cont.)

- + parallelism (multiple readers)
- + updates propagated only when data are read
- + several updates can take place before communication is necessary
- Cost of invalidating read-only copies before a write can occur
 - + ok if read/write ratio is sufficiently high
 - + for small read/write ratio: single-reader-single-writer scheme
(at most one process gets read-only access at a time)

Finding the owner of a page / object

broadcast (ask all)

- request may contain access mode, need for a copy
- owner replies and potentially transfers ownership
- every processor must read the request (interrupt)
- high bandwidth consumption

page manager keeps track of who owns each page

- send request to page manager,
- page manager sends owner information back
- heavy load on page manager → use multiple page managers

keep track of probable owner of each valid page [Li/Hudak'89]

- send request to probable owner,
- probable owner forwards if ownership has changed.
- Periodically broadcast ownership info (after multiple ownership changes)

Write-update protocol

Write x :

- done locally + broadcast new value to all who have a copy of x
- these update their copies immediately.

Read x :

- read local copy of x , no need for communication.
- multiple readers
- several processes may write the same data item at the same time
(multiple-reader-multiple-writer sharing)

Sequential consistency if broadcasts are totally ordered and blocking

- all processors agree on the order of updates.
- the reads between writes are well defined

+ Reads are cheap

- totally ordered broadcast protocols quite expensive to implement

Finding all copies

All copies must be invalidated if a page is written.

broadcast page number to all

- every processor holding a copy invalidates it
- requires reliable broadcast

copy set

- owner or page manager keep for each page a set of copy holders
- invalidation request sent to all in the copy set

Page / Cache line replacement

Replacement necessary if no free page frame available

LRU etc. not generally applicable with migration/replication

For replacement, prefer

0. invalid pages
1. private (non-shared) pages
 - save to local main memory/disk, no need for communication
2. replicated pages owned by others (→ read-only)
 - no need for saving, another copy exists
 - inform the owner / page manager
3. replicated page owned by myself (abandon ownership)
 - inform new owner / page manager
4. non-replicated page: swap out to local disk as usual, or to remote disk (maybe assisted by a free page frame manager)

Writable copies

Multiple writers, multiple readers e.g. [Munin](#)

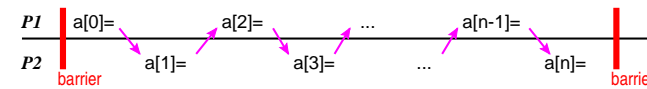
Programmer explicitly allows concurrent writing for some shared variables and is responsible for their correct use

```

P1          P2
a : [write-shared] array;
barrier(b);
for (i=0; i<n; i+=2)
    a[i] = a[i] + f[i];
barrier(b+1);

a : [write-shared] array;
barrier(b);
for (i=1; i<=n; i+=2)
    a[i] = a[i] + f[i];
barrier(b+1);
    
```

with sequential consistency:



with release consistency:

