Scuola Superiore Sant'Anna
di Studi Universitari e di Perfezionamento

IN SUPREMÆ DIGNITATIS
1343
UNIVERSITÀ DI PISA

Master Program (Laurea Magistrale) in Computer Science and Networking

High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

4. Shared Memory Parallel Architectures

4.3. Interprocess Communication and Cache Coherence

Run-time support of concurrency mechanisms

- Features of the multiprocessor run-time support:

  1. Locking: indivisibility (atomicity) of concurrency mechanims.
  2. Low-level scheduling: different versions of process wake-up procedure for anonymous *vs* dedicated processors architectures
  3. Communication processor: overlapping of interprocess communication to calculation. *send* implementation.
  4. Cache-coherence: automatic firmware support *vs* algorithm dependent approach. *send* implementation.
  5. Interprocess communication cost model, referring to the algorithm-dependent cache coherence implementation.

1. LOCKING

Locking

- *Indivisible (atomic) operation sequences* (often on shared data structures) are executed with disabled interrupts *and* implemented as critical sections such as:

  lock (X);

  < critical section >;

  unlock (X)

  where *X* is a **lock semaphore** associated to the critical section (if the critical section acts on a single shared data structure D, the X is associated to D).

- *Lock, unlock* operations are analogous to P, V (Wait, Signal) primitives, however
  - their atomicity is implemented by mechanisms ***at the hardware-firmware level***,
  - they synchronize *processors* (not processes), thus imply ***busy waiting*** in the lock operation.

Simplified locking operations

The simplest (though inefficient) implementation of locking operations is the following, where X is a boolean lock semaphore:

lock (X):: *indivisible_sequence* {wait until X; X = false}

unlock (X) :: X = true

The lock sequence (at the assembler or firmware level) is of the kind:

Lock: **read** X into register R;

if (R = 0) *then goto* Lock *else* **write** 1 into X

This read-write sequence must be implemented in an ***indivisible*** way, i.e. if processor P_i performs thefirst (read*)* operation of the sequence, no other P_j ($j \neq i$*)* can start the sequence (can read location X) until P_i completes the sequence.

Indivisible sequences of memory accesses

- The atomic implementation of locking mechanisms at the hardware-firmware level consists in the primitive implementation of indivisible sequences of memory read-write operations, e.g. (the sequences inside *lock* and *unlock* are examples)

  read (a)

  read (b)

  read (c)

  write (a, val)

- Assume that a, b, c are locations of the same memory module M. If the sequence is initiated by processor P_i, the module unit

  (or an arbiter of M, or an interface unit, or a switching unit connected to M)

  prevents any other P_j ($j \neq i$) from executing read-write operation on M until the sequence is completed.

- *Note:* it could be sufficient that any other P_j is prevented to access the data structure containing a. However, because we are interested in short sequences, the “block of memory module” solution is simpler and more efficient.

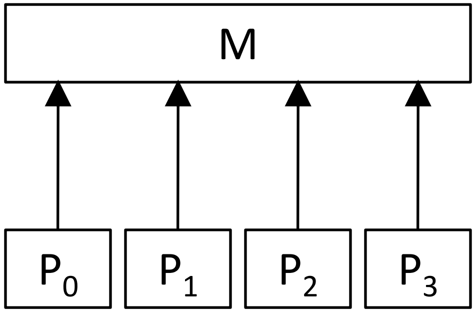
- *Note:* if a, b, c are not in the same module, it is sufficient to block the module of a, provided that there is no possibility to access b, c independently before accessing a.

Indivisible sequences of memory accesses

- In the firmware interpreter, an **indivisibility bit** (indiv) is associated to every memory access request, to denote (indiv = 1) the beginning of an indivisible sequence. It is mantained at indiv = 1 in all the successive requests of the sequence, except the last request in which indiv = 0.
- Often, the assembler level provides instructions like *Test and Set*, *Atomic exchange*, *Add to Memory*, etc.
- More flexible and efficient mechanism: directly control the indivisibility bit. For example, in D-RISC:
  - **special instructions**: SET_INDIV, RESET_INDIV
  - **annotations** (setindiv, resetindiv) in LOAD, STORE instructions.
- For very short lock sequences (1-2 memory accesses), *lock/unlock* can be replaced directly by *set_indiv /reset_indiv* . *Lock/unlock* are used for longer sequences.

Architectural implications

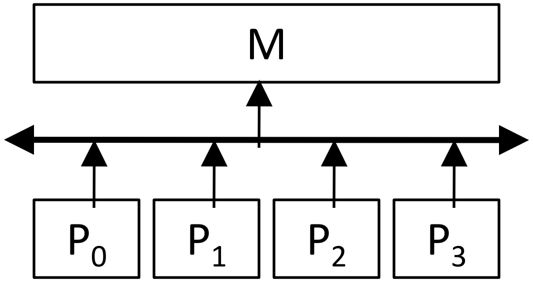
Two extreme situations, according to which all the architectures with the known interconnection structures can be studied.



1) M (or an interface/arbiter unit in front of M) has a *non-deterministic* behaviour when no indivisible sequence is in progress.

If a request from P_i with indiv = 1 is selected, then M enters a *deterministic* behaviour, during which M "listens" only the P_i requests, i.e. M doesn't test RDYs of the other interfaces.

When indiv = 0, M resumes the *non-deterministic* behaviour.



2) M (or an interface/arbiter unit in front of M) listens the only input link. A Request contains the name of the requesting processor.

If a request from P_i with indiv = 1 is selected, then M remember the name of P_i, and *continues to listen any request*. Requests from P_i are served, requests from the others processors are *buffered* into an internal FIFO queue.

When indiv = 0, M serves the queued requests and/or new input requests.

Exercize

According to the two extreme situations above, solve the problem of management of memory accesses indivisible sequences, for the following architectures:

a. SMP whose interconnection network is composed of *m* trees, one for each shared memory module (root in the memory module)

b. SMP with k-ary n-fly network

c. NUMA with Fat Tree network

d. NUMA with k-ary n-cube network

Locking with periodic retry

- In order to reduce the continuos accesses to shared memory of the simplified algorithm, the lock test is re-executed after a given time interval. The algorithm is “*unfair*” (potential infinite waiting).

Writing: ***set indiv*; S (x); *reset indiv*** means: sequence S, on shared data *x*, is executed with *indiv* = *1* for all the memory accesses to *x* except the last one.

lock (semlock)::

```
        ok = false;
        while not ok do { set indiv;
                          if semlock then {semlock = false; reset indiv; ok = true }
                                  else { reset indiv;
                                         wait for a given time-out }
                        }
```

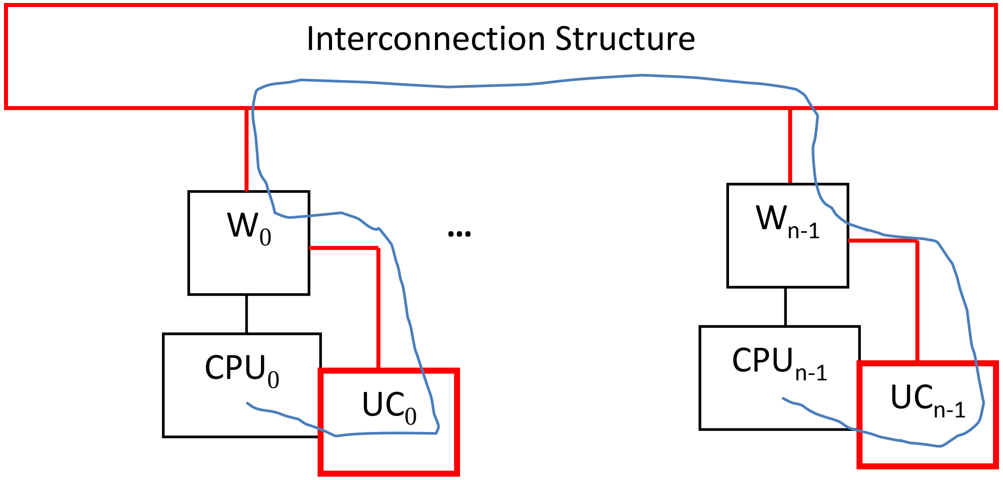
unlock (semlock)::

```
        semlock = true
```

Fair locking with queueing

- Lock semaphore is a data structure containing the value (boolean) and a FIFO queue of processor names.
- In case of *red* semaphore in lock operation, processor waits an explicit "*unblocking communication*" by any processor executing the unlock operation.
- This communication is a direct communication one, on the *processor-to-processor network* if distinct from the processor-to-memory network. If they coincide (e.g. Generalized Fat Tree), distinct message types are used for different usages.
- An I/O unit, called Communication Unit (UC) is provided for this purpose.
- In the waiting node, the unblocking communication is transformed into an *interrupt* from UC to the processor. In fact, the processor was *waiting for an interrupt* (see special instruction in D-RISC).

Direct *inter-processor* communications



Fair locking with queueing

Initialization:: semlock.val = true; semlock.queue = empty.

```
lock (semlock)::
        set indiv;
        if semlock.val then { semlock.val = false; reset indiv }
                else { put (processor_name, semlock.queue) ; reset indiv;
                     wait for an unblocking interrupt from UC}

unlock (semlock)::
        set indiv;
        if empty (semlock.queue) then { semlock.val = true; reset indiv }
                        else { processor_name = get (semlock.queue); reset indiv;
                             send to UC an ublocking message
                                           to be sent to processor_name }
```

Locking and caching

- Because of cache coherence reasons, the lock/unlock algorithms could also contain actions for the correcty management of memory hierarchy.
- *See the Cache Coherence section*.

Exercize

Implement in detail (in D-RISC) the lock operation according to the fair algorithm.

Evaluate its latency in case of *green* semaphore.

Technological remark

- In principle, locking operations are very efficient.
- They can be executed in few clock cycles (except remote accesses) if *proper assembler instructions* exist, and are executed in *user mode.*
- Again: inefficiencies are introduced when locking is a OS mechanism executed in kernel mode.
- Alternative: *lock-free algorithms* for synchronizations (e.g., ad-hoc solutions for the various concurrency problems), which, on the other hand, can be expensive from the computational viewpoint.

Run-time support of *send* primitive

- Let’s modify the uniprocessor version of *send* run-time support, including locking.
- Channel Descriptor is a linear sequence of words containing a *lock semaphore*:
  - **X: lock semaphore;**
  - WAIT: boolean;
  - Message_length: integer;
  - Buffer: FIFO queue of (k + 1) positions (reference to target variable, validity bit)
  - PCB_ref: reference to PCB
- The Channel descriptor address is the logical address of X.
- *Note:* the algorithm is the same for every multiprocessor architecture, except for the low-level scheduling procedure.

Run-time support of *send* primitive

```
send (ch_id, msg_address) ::
        CH address =TAB_CH (ch_id);
        lock (X);
        if (CH_Buffer [Insertion_Pointer].validity_bit = 0)  then
                { wait = true;
                  copy reference to Sender_PCB into CH.PCB_ref ;
                  unlock (X);
                  process transition into WAIT state: context switching };
        copy message value into the target variable referred by
                        CH_Buffer [Insertion_Pointer].reference_to_target_variable ;
        modify CH_Buffer. Insertion_Pointer and CH_Buffer_Current_Size;
        if wait then
                { wait = false;
                  wake_up partner process (CH.PCB_ref) };
        if buffer_full then
                { wait = true;
                  copy reference to Sender_PCB into CH.PCB_ref ;
                  unlock (X);
                  process transition into WAIT state: context switching }
             else  unlock (X)
```

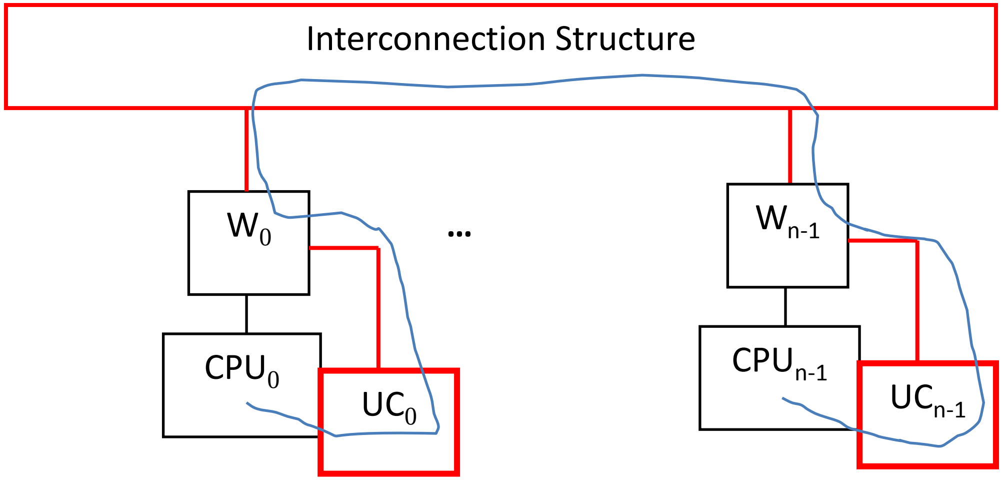
Exercize

Find an alternative implementation of *send* primitive, able to reduce the locking critical section duration.

2. LOW-LEVEL SCHEDULING

Process wake-up procedure

- This procedure is different for anonymous processors and dedicated processors architectures.
- This is another opportunity for *direct interprocessor communications:*



Preemptive wake-up in anonymous processors architectures

- An anonymous processors architecture
  - is a direct extension of a uniprocessor
  - aims to load balance processors.
- These features require a distributed strategy for *preemptive wake-up* scheduling (processes with priority).
- Assume that process B is WAITing and process A, running on P_i, wakes-up B.
- If $\exists$ P_j executing process C : priority (B) > priority (C) then B preempts C, i.e. B passes in Execution on P_j and C becomes Ready.
- Otherwise, A put PCB_B into the Ready List.

Preemptive wake-up in anonymous processors architectures

- A is in charge of consulting a *Cental Table*, where the correspondence (processor-process in execution-priority) is mantained.
- A selects the minimum priority entry (P_j, C, priority) of Central Table.
- If preemption can be applied, if C ≡ A, then A performs preemtion on P_i directly.
- Otherwise, A sends to P_j an interprocessor message (preemption, reference to B, priority of B).
- It is possible that, in the meantime, P_j has performed a context-switch , or C has changed priority. The UC associated to P_j can detect this situation and, usind DMA, can try to perform a preemtion onto a different processor, i.e., try to “bounce” the preemption message to another processor, without interrrupting P_j.
- This potentially infinite procedure can be stopped after a (very) limited number of “bounces”.
- Notice that, in order to avoid this problem (not so serious, nor leading to inconsistent state), a complex locking procedure should be implemented:
  - which data structures should have been locked, and when unlocked ?

Exercize

Preemptive wake-up in SMP. Answer the previous question:

… Notice that, in order to avoid this problem (…), a complex locking procedure should be implemented:

- which data structures should have been locked, and when unlocked ?

i.e., implement the preemptive wake-up procedure in such a way that the situation detected by A doesn't change until the wake-up has been completed (with preemption or not, on A's node or not).

Shared I/O

- In a UMA anonymous processors architecture, in principle I/O units are *anonymous* themselves, i.e. an I/O unit is not statically associated to a specific processor.
- An interrupts can be sent to, and served by, any processor.
- *The most interruptable processor* can be chosen, according to the same technique for preemptive wake-up.

Process wake-up in dedicated processors architectures

- In a dedicated processors architecture, every node has full responsibility of the low-level scheduling of its processes.
- Assume process A, running on P_i, wishes to wake-up process B, running on P_j.
- A replicated table contains the mapping correspondence (process, processor).
- If $i = j$, then A wakes-up B using the Ready List of P_i.
- Otherwise, A sends to P_j an interprocessor message (wake-up, reference to B). On P_j, the running process executes the wake-up procedure according to the local rules (with/without preemtion).

Exercize

Describe in detail the implementation of process wake-up procedure for SMP and NUMA machines, including the UC behaviour and the interrupt handling actions.

3. COMMUNICATION PROCESSOR

Principle

- Every processing node contains a second CPU (KP) *dedicated* to the implementation of the run-time support of interprocessor communication.
- The main CPU (IP) *delegates* to KP the execution of (the most expensive part of) the *send* primitive.
- If communication is *asynchronous*, *and* IP has work to do after a *send*, then communication is overlapped (at least in part).
- KP executes just one process permanently (a "demon"), i.e. the process consisting in the *send* interpreter.
- KP may be either a specialized co-processor, or identical to IP.
- *receive* primitive is executed by IP.

Examples

while … *do* { y = F(x, y); *send* (ch, y)}

Calculation is correctly overlapped to communication if the message value is not modified by calculation during the *send* execution.

while … *do* { y2 = F(x, y1); *send* (ch, y2); y1 = F(x, y2); *send* (ch, y1) }

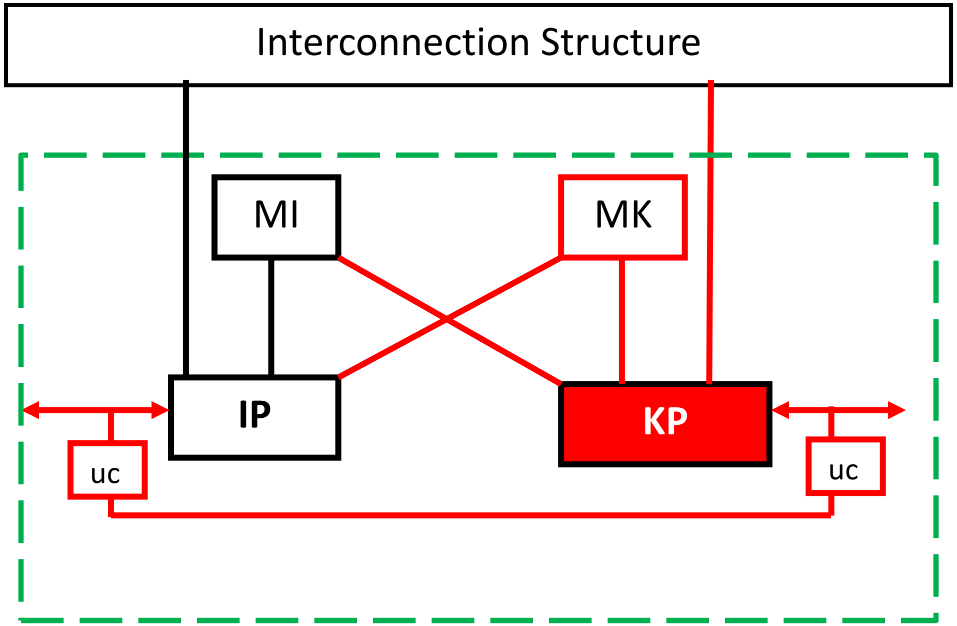
A partial loop-unfolding solves the problem easily (“double buffering”)

while … *do* { *receive* (ch1, x); y = F(x, y); *send* (ch2, y)}

receive is executed by IP, thus it is overlapped to *send* (and, with zero-copy communication, *receive* latency is relatively low);

apply loop-unfolding if necessary.

Processing node structure



IP, KP share local memories inside the node; *processing node is a small NUMA*.

KP has access to the whole shared memory of the system.

Delegation of *send* execution from IP to KP: IP transmits to KP a *reference* (e.g. capability) to the *send* parameters (ch_id, message value), via direct interprocessor communication (uc – uc).

The send execution by KP exploits KP resources, in particular KP cache.

Exercizes

1. Describe the detailed scheme of a processing node with communication processor, for an SMP and for a NUMA architecture, using the D-RISC Pipelined CPU as off-the-shelf building block.

2. Assuming KP identical to IP, a multiprocessor with N processing nodes has a total of 2N CPUs. Thus the problem arises: why not exploiting 2N processing nodes without communication processors?

   Discuss this problem, individuating pros and cons of the communication processor solution.

send implementation with KP

- *send* semantics:
  - copy the message and, if the asynchrony degree becomes saturated (buffer_full), suspend the Sender process.
- Of course, this condition must by verified in presence of KP too.
- If IP delegates to KP the *send* execution *entirely*, then some complications are introduced in the send implementation because of the management of the WAIT state (versions of this kind exist).
  - more than one communication should be delegated to KP:

```
calculation …;
send …;
send …;
send …;
```

send implementation with KP

- Simple and efficient solution, though *at the expense of a (small) reduction of overlapping*:
  - ***IP verifies the saturation of channel asynchrony degree*** (i.e. IP performs the buffer_full test);
  - IP delegates to KP the *send* continuation;
  - if (*buffer_size = k*) IP suspends the Sender process (*k* = asynchrony degree, *k + 1* = number of queue elements);
  - otherwise IP delegates to KP the *send* continuation, and Sender goes on.
- KP doesn’t perform the *buffer_full* test.
- Channel Descriptor is *locked* by IP and *unlocked* by KP.
- Zero-copy communication: IP controls the *validity bit* too.
- *Optimization:* mantain a consistent copy of *Buffer_Size* and *validity bit* in local memory of IP (avoid the remote channel access).

send implementation with KP

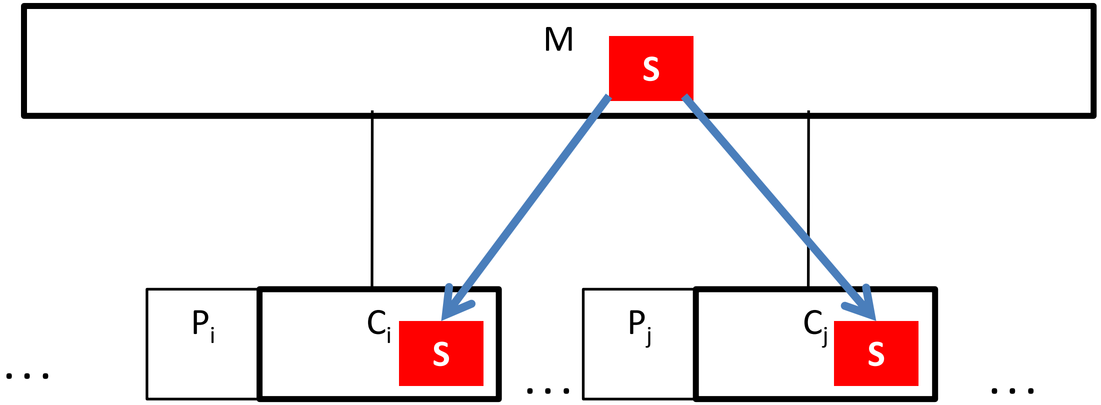
- Latency of the *buffer_full* and *validity bit* verification in IP has to be added to T_{calc} (not T_{setup}).

- In some advanced architectures, KPs are built into to the interconnection structure, and are connected to the nodes via interface units (W).

4. CACHE COHERENCE

Cache coherence problem

- If the a shared data (block) is loaded into more than one cache, all the *copies* must be identical (*consistent*) each other and identical to the value in shared Main Memory (or in further shared levels of the memory hierarchy).
- Abstract scheme, for our purpose : shared memory M and (processor P, primary data cache C) nodes

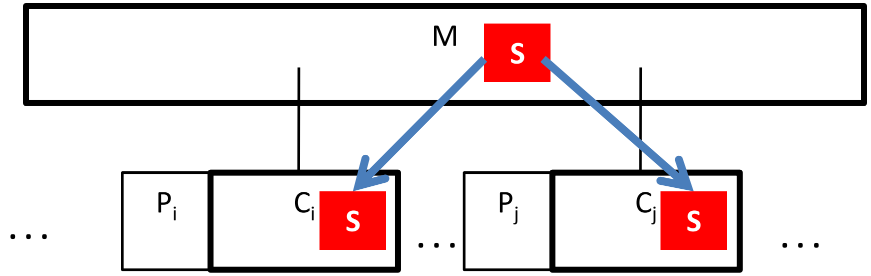


S (block) is loaded into C_i and C_j.

No consistency problem if S is used in a **read-only** mode by P_i and P_j.

If P_i **modifies** its S copy, then the copy in C_j is *inconsistent* (in general).

Cache coherence problem



P_i be the first processor that modifies S in its cache C_i.

Write-Back cache: S copy in M (main copy) is *inconsistent*

- Other processors find a inconsistent value of S in M, when they transfer S in their cache
- P_j can not rely on the copy of S when it need to read it again
- The only way to acquire a consistent value of S is *with the help of P_i*

Write-Through cache: S copy in M is *consistent*

- When ? Updating actions of S in C_i and in M are not simultaneous (if they were simultaneous: what about the Write Through performance ?)
- There is a time interval during which S in M is inconsistent.
- Other processors (and Pj itself) could find a consistent value of S in M, when they need to transfer S in their cache*, provided that* they are informed in time by $P_{i.}$
- Again: in some way, they need *the help of P_i*.

Automatic cache coherence

- Cache coherence solution at the *firmware* level: it is guaranteed that a processor always work on a consistent copy, in a seemless way.
- Two main techniques:
  1. Invalidation: only one cached copy is valid, i.e. the last modified, all the other copies are/become invalid
  2. Update: all the cached copies are mantained consistent (broadcasted value)
- In both cases, an accurate (often, heavy) synchronization mechanism has to be adopted, e.g.
  1. Invalidation: how can a P_j understand that its copy has been invalidated or not yet invalidated?
  2. Update: how can a P_j understand that its copy has been updated or not yet updated?

Synchronization in automatic cache coherence

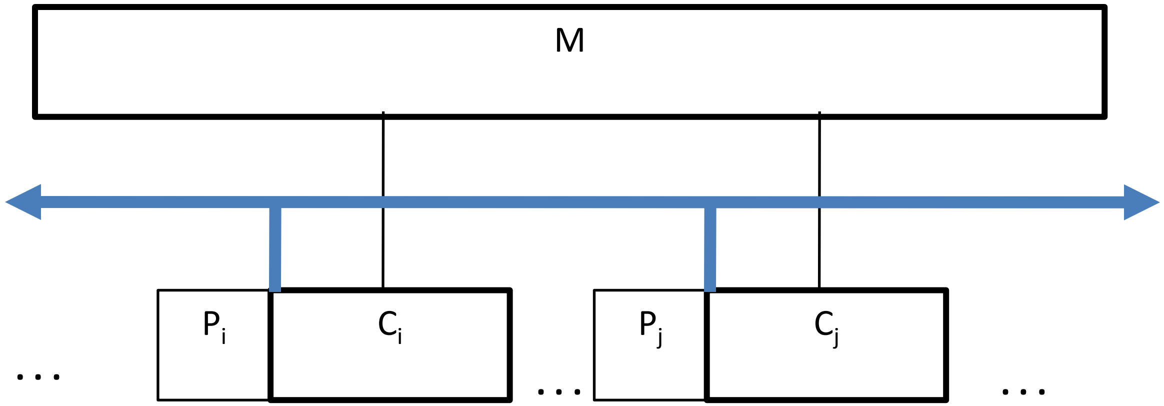
- Analogy: Pipelined CPU: General Registers copies in IU and EU
- *Update-like* solution: EU sends the updated value to IU
- Synchronization through a semaphoric solution in IU
  - Each reading operation of a register in IU can be executed if and only if the associated semaphore is green, otherwise IU must wait.
- Efficient solution in Pipelined CPU:
  - synchronization *per se* has zero cost, also because there is only one secondary copy,
  - but, what about the cost of the synchronization mechanism in a cache coherent multiprocessor?
- *Invalidation-like* solution in the IU-EU analogy: not meaningful, since IU doesn't modify registers.

Invalidation

- When a processor P_i modifies a block S in C_i, all the other copies of S (if any) are invalidated
  - An invalidated copy is **de-allocated**
  - Invalidation is communicated by P_i, with a broadcast message to all processors, or a multicast message to those processors having S in their cache
  - Write-Back or Write-Through: the valid, acquired copy can be read from M
  - **Potential "ping – pong" effect.**
- Synchronization:
  - for each cache access, every P_j waits for a possible invalidation communication (S)
  - Time-dependent solution ? (e.g., synchronized logical clocks)
  - Otherwise: complexity, and performance degradation, increases rapidly
- Centralization point at the firmware architecture level:
  - **Snoopy bus** in many commercial **SMP** multiprocessors with **low parallelism** (4 processors) and bus interconnection structure.

Snooping architectures (SMP)

- e.g. *MESI* protocol (see: [Drepper], section 3.3.4; other protocols and formal treatment in [Pong-Dubois])



Snoopy bus: to communicate invalidation events
Also: a centralization point for synchronization: bus arbiter
Before accessing its cache, a processor has to listen the bus

Analogous consideration for **Update** technique, in which the bus supports a synchronized multicast too.

Snooping: optimizations

E.g. *MESIF* protocol

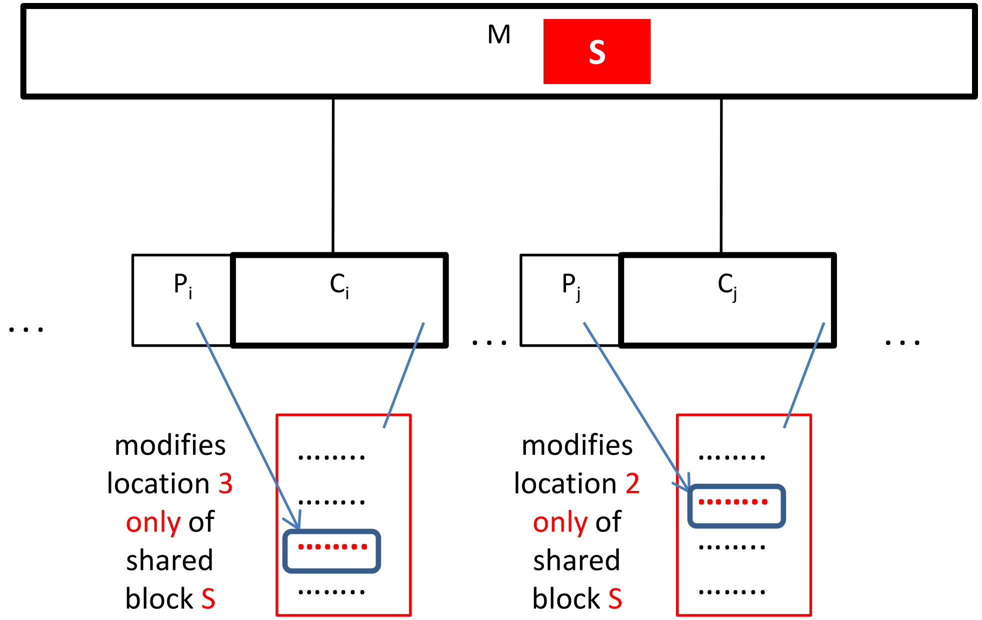
- For each block descriptor: bit_mask[N], where bit *i-th* corresponds to processor P_i
  - *in shared memory*
- if, for block *p*,
  - bit_mask[j] = 0: processor P_j can not have block *p* in cache C_j;
  - bit_mask[j] = 1: it is possible (but not sure) that processor P_j has block *p* in cache C_j;
- For each *p*, automatic coherence is applied to processors having bit_mask = 1.
- This technique limits the number of invalidation messages.

Directory-based architectures

- For higher parallelism systems, SMP and NUMA: avoiding centralization points, like buses
- High-parallelism interconnection structures can be used
- Synchronization through shared variables (*directories*) in local memories, containing all the information about cached and shared objects:
  - firmware support to mantain consistent these variables too: at a cost much lower than the cost of cache coherence itself for any information block (*specialized firmware for directories*).
- The directory-based solution can be implemented by program too: see the **algorithm-dependent caching**, i.e
  - exploit the *knowledge of the computation semantics t*o implement cache coherence without automatic support
  - *explicit utilization of locking synchronizations*.

"False sharing"

Distinct processors modify *different locations of the same cache block*:



No real sharing of information, thus no real problem of cache coherence.

However, this situation is considered as true sharing if **automatic** coherence is applied (*cache block* is the elementary unit of consistency),

thus the invalidation / update traffic is executed.

"Padding" techniques (data alignment to block size) in order to put words modified by distinct processors in distinct blocks.

- effective when recognizable (knowledge of program semantics)

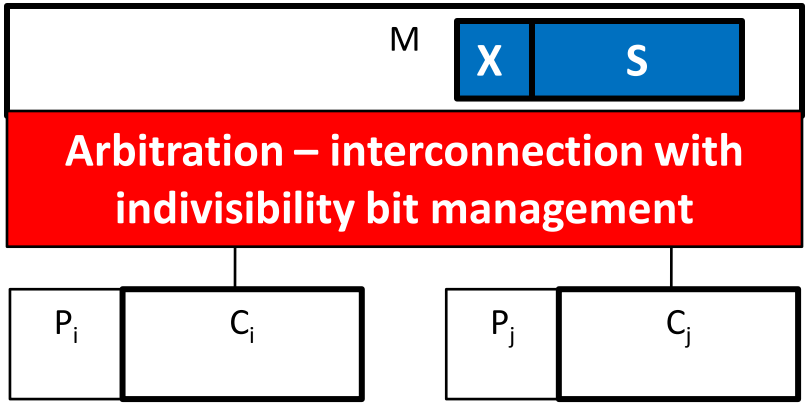
Automatic solutions: alternatives

- *Automatic solutions - Pros:* no visibility at the program level
- *Cons:* inefficiency and high overhead, especially in high parallelism multiprocessors, or low parallelism architectures (even with meaningful overhead too)

- The key point for alternative solutions is:
  - Processors adopt synchronization mechanisms at the hardware-firmware level: locking, indivisible bit, architectural supports in interconnection networks
  - Can these mechanisms help to solve the cache coherence problem without additional automatic mechanisms?
  - In which cases these alternative solutions are applicable, provided that the synchronization mechanisms (locking) are efficient (user mode) ?

Cache coherence in presence of locking

- Consider a system with automatic cache coherence
- Consider a process, running on P_i (cache C_i), and a process, running on P_j (cache C_j), sharing a structure S with lock semaphore X.
- S is accessed in *lock state* (mutual exclusion with busy waiting).



M
X
S
Arbitration – interconnection with indivisibility bit management
P_i
C_i
P_j
C_j

X belongs to the first block of S, or is allocated in an separate block.

Locking is supported, at the HW-FW level, by the **indivisibility bit mechanism**.
This mechanism represent a logically centralized, very efficient synchronization point.
This mechanism prevents that both processors are able to load the lock semaphore X in their respectives caches simultaneously.

Cache coherence and locking

- Assume the worst case for our purposes: P_i and P_j try to access X simultaneously.
- Both request for X reading are associated indiv = 1.
- Assume P_i is able to copy X in C_i. P_j’s request is pending (queued) in the proper unit for the indivisibility bit management.
- P_i executes lock (X), *modifying* X; at the end, executes *reset_indiv*. P_j is now able to access shared memory for copying X in $C_{j.}$
- The X copy in C_i is *invalidated*. This has no particular effect: P_i has already utilized X (has executed lock (X)).
- Pj executes lock (X) on its X copy: X is red, thus P_j enters busy waiting.

Cache coherence and locking (cont.)

- When P_i executes unlock (X), it loads X in C_i and *invalidates* X copy in C_j. This has no particukar effect on P_j, which was not using X (busy waiting), provided that lock (X) is implemented efficiently (better solution: fair lock with queue).
- The unlock (X) provides to unblock P_j, which loads X in C_j and invalides X copy in C_i. This has no particular effect on P_i, which doesn't need X any more.
- Conclusion: cache coherence *per se* is not computationally inefficient, e.g. invalidation doesn't introduce "ping-pong" effects, *owing to the locking synchronization and indivisibility bit mechanism.*
- Consequence: try to avoid the inefficiencies of automatic solutions (mechanism overhead, low parallelism), relying entirely to the locking synchronization mechanisms.

Algorithm-dependent cache coherence

- Cache coherence is implemented by program, relying on the locking synchronization mechanisms and knowing the computation semantics.
- This requires to individuate a proper algorithm.
- Explicit management of memory hierarchy is required. This is rendered efficient by suitable assembler supports, through LOAD/STORE instruction annotations.
- In D-RISC:
  - STORE …., **block_rewrite** : explicit re-writing of a block, **independently of Write-Back or Write-Through cache**
  - STORE …, **don't_modify** : even for Write-Through cache, a writing operation is executed *in cache only*.
  - STORE …, **single_word** : even for Write-Back cache, modify a *single* word in shared memory *directly*, instead of an entire block
  - LOAD/STORE …, **deallocate** : explicit cache deallocation of a block.
  - LOAD/STORE …, **don't_deallocate** : same annotation exploited for *reuse*

Algorithm-dependent cache coherence

- When it is a meaningful approach:
  - in the design of run-time support mechanisms, e.g. for interprocess communication
- In this way, in a structured approach to parallel programs, there is no other needs of cache coherence mechanisms,
  - i.e., the only situations in which we need to implement cache coherence on shared data structures in the the run-time support design.
- This leads to a good trade-off between programmability and performance.
- Many recent CPU and multicore products expose annotations and special instructions for explicit management of memory hierarchy, and the absence of automatic cache coherence or the possibility to disable the automatic cache coherence facility.

Send run-time support with algorithm-dependent cache coherence

- The *send* run-time algorithm is modified according to the principles of algorithm-dependent cache coherence (relationship between cache cohernce and locking, explicit annotations).
- Consider any *send* algorithm, e.g. zero-copy

Channel Descriptor ::

```
        X: lock semaphore
        WAIT
        Message_length
        Buffer
        PCB_ref
```

Algorithm overview:

```
        lock (X);
                Utilize and modify CH blocks
                Copy message
                …
        unlock (X);
        Procedure return or context-switching
```

Send run-time support with algorithm-dependent cache coherence

- lock (X) provides to load the first CH block in Sender cache, to modify X, to re-write explicitly X in shared memory (entire block, or only the X words), and to leave X block in cache (don’t deallocate).
  - in the meantime, it is possible that the node, executing the Receiver process, is blocked: initially because of the indivisibility bit, then because of red X.
- All the other CH blocks are loaded in Sender cache and possibly modified. When a block utilization ends, it is explicitly dellocated and, if modified, explicitly re-written in shared memory.
- The unlock (X) execution finds X still in cache, modifies X, explicitly re-writes X in shared memory, and explicitly deallocates X.
- Note: *no overhead* is paid for cache coherence.

Exercizes

1. Evaluate the difference between the latencies of two *send* implementations: with automatic cache coherence, with algorithm-dependent cache coherence.

2. Study algorithm-dependent solutions to cache coherence applied at the secondary caches in SMP architectures with D-RISC nodes.

5. INTERPROCESS COMMUNICATION COST MODEL

Approximate evaluation of T_{send}

- $T_{send} = T_{setup} + L * T_{transm}$
- Algorithm-dependent cache coherent version of *send* run-time support
- Zero-copy implementation
- SMP architecture, without communication processor
- Let's evaluate only the *memory access latencies for reading and writing operations on CH blocks and target variable blocks*
  - rough approximation of T_{send} ,
  - to be increased by a 20 – 30% to take into account the instruction execution on local (cached) objects and registers, and low-level scheduling.

Assumptions

- It is reasonable to assume that (most probable situation) the following objects have been *already loaded into the local cache*:
  - Channel Table
  - Relocation Table
  - Sender Process PCB
- while the following objects *are not supposed to be in local cache*:
  - Message value
  - Channel descriptor
  - Target variable
  - Ready List
- *Channel descriptor* is composed of 3 “padded” blocks containing
  - lock semaphore
  - WAIT, Message Length, Pointers and Current_Size of Buffer, PCB_ref
  - Buffer: Vector of target variable references
- *Assumption:* asynchrony degree $\leq$ block size.

send latency

- According to the algorithm-dependent cache-coherent *send* implementation of Section 4, the following block transfers from/to remote shared memory are performed:
  - Reading first block (**lock semaphore**)
  - Re-writing first block
  - Reading second block (in particular: Buffer Pointers and Current_Size)
  - Re-writing second block
  - $\lceil L/\sigma \rceil$ readings of message blocks (L = message length, σ = block size)
  - $\lceil L/\sigma \rceil$ writings into target variable.
- We can assume:

$$T_{block\text{-}read} = T_{block\text{-}write} = R_Q \text{ (under-load)}$$

(see **Section 4.2** on memory access latency evaluation).

Assuming L multiple of σ, we obtain:

send latency

$$T_{setup} = 4\ R_Q$$

$$T_{transm} = 2\ R_Q\ /\ \sigma$$

From Section 4.2, we can evaluate the following orders of magnitude:

$$T_{setup} \sim (10^3 – 10^4)\ \tau$$

$$T_{transm} \sim (\ 10^2 – 10^3)\ \tau$$

where the variability range depends on the various parameters: p, $T_{p,}$ τ_M, etc.

Exercizes

1. Evaluate T_{send} for a NUMA architecture, assuming the message value in local memory of the node.
2. Evaluate the impact of instruction executions and low-level scheduling on T_{send} for D-RISC Pipelined CPU nodes.
3. Evaluate T_{send} for a zero-copy implementation with communication processor.