Shared Memory Multiprocessors

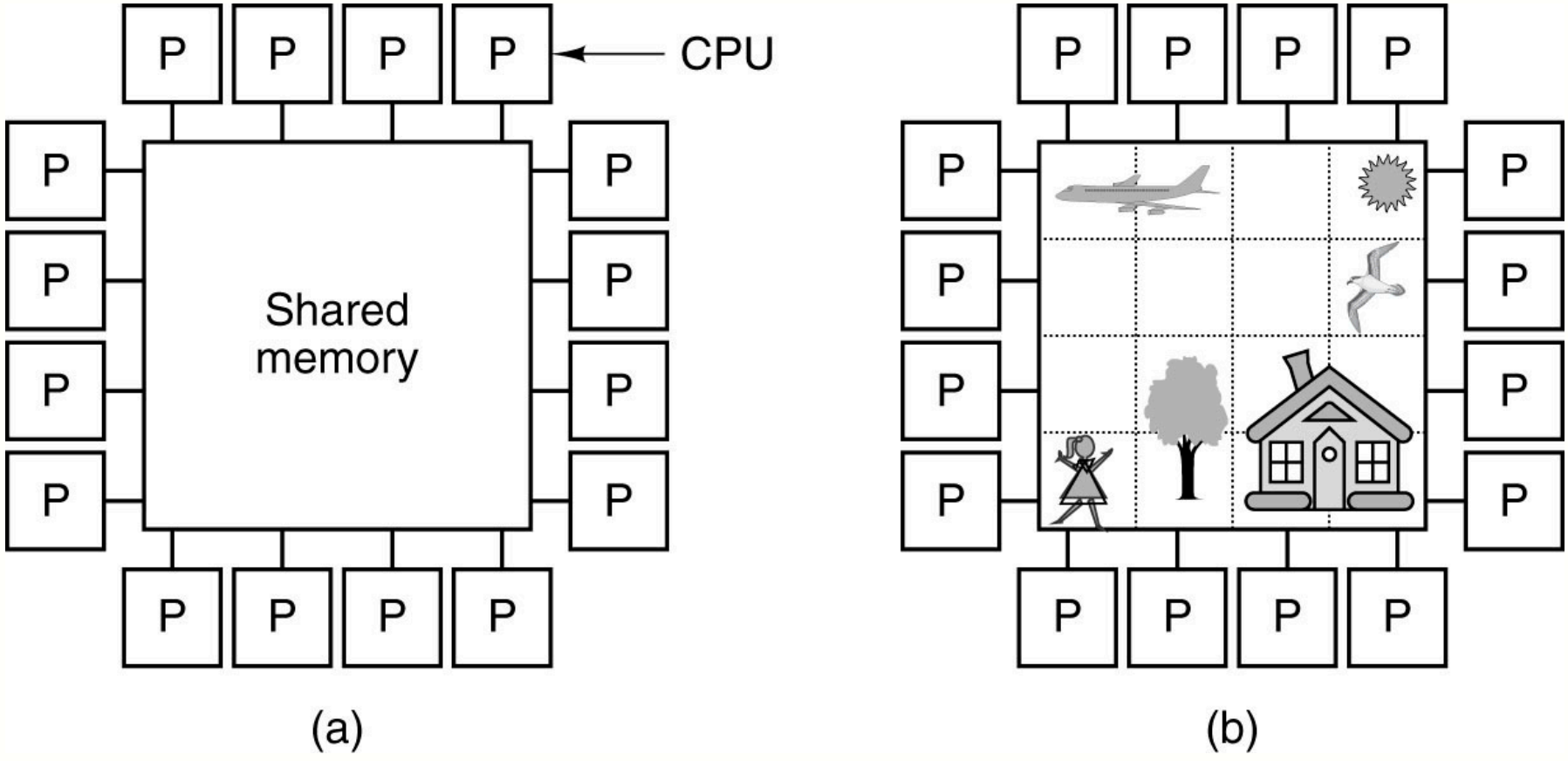
- Introduction
- UMA systems
- NUMA systems
- COMA systems
- Cache coherency
- Process synchronization
- Models of memory consistency

Shared memory multiprocessors

- A system with multiple CPUs "sharing" the same main memory is called **multiprocessor**.
- In a multiprocessor system all processes on the various CPUs share a ***unique logical address space***, which is mapped on a physical memory that can be distributed among the processors.
- Each process can read and write a data item simply using *load* and *store* operations, and *process communication* is through ***shared memory***.
- It is the hardware that makes all CPUs access and use the same main memory.

Shared memory multiprocessors

- This is an architectural model simple and easy to use for programming; it can be applied to a wide variety of problems that can be modeled as a set of tasks, to be executed in parallel (at least partially) (Tanenbaum, Fig. 8.17).



Shared memory multiprocessors

- Since all CPUs share the address space, only a single instance of the operating system is required.
- When a process terminates or goes into a wait state for whichever reason, the O.S. can look in the process table (more precisely, in the ready processes queue) for another process to be dispatched to the idle CPU.
- On the contrary, in systems with *no shared memory*, each CPU must have its own copy of the operating system, and processes can only communicate through *message passing*.
- The basic issue in shared memory multiprocessor systems is memory itself, since the larger the number of processors involved, the more difficult to work on memory efficiently.

Shared memory multiprocessors

- All modern OS (Windows, Solaris, Linux, MacOS) support **symmetric multiprocessing**, (**SMP**), with a scheduler running on every processor (a simplified description, of course).
- "ready to run" processes can be inserted into a single queue, that can be accessed by every scheduler, alternatively there can be a "ready to run" queue for each processor.
- When a scheduler is activated in a processor, it chooses one of the "ready to run" processes and dispatches it on its processor (with a single queue, things are somewhat more difficult, can you guess why?)

Shared memory multiprocessors

- A distinct feature in multiprocessor systems is **load balancing**.
- It is useless having many CPUs in a system, if processes are not distributed evenly among the cores.
- With a single “ready-to-run” queue, load balancing is usually automatic: if a processor is idle, its scheduler will pick a process from the shared queue and will start it on that processor.

Shared memory multiprocessors

- Modern OSs designed for SMP often have a separate queue for each processor (to avoid the problems associated with a single queue).
- There is an explicit mechanism for load balancing, by which a process on the wait list of an overloaded processor is moved to the queue of another, less loaded processor.
- As an example, SMP Linux activates its load balancing scheme every 200 ms, and whenever a processor queue empties.

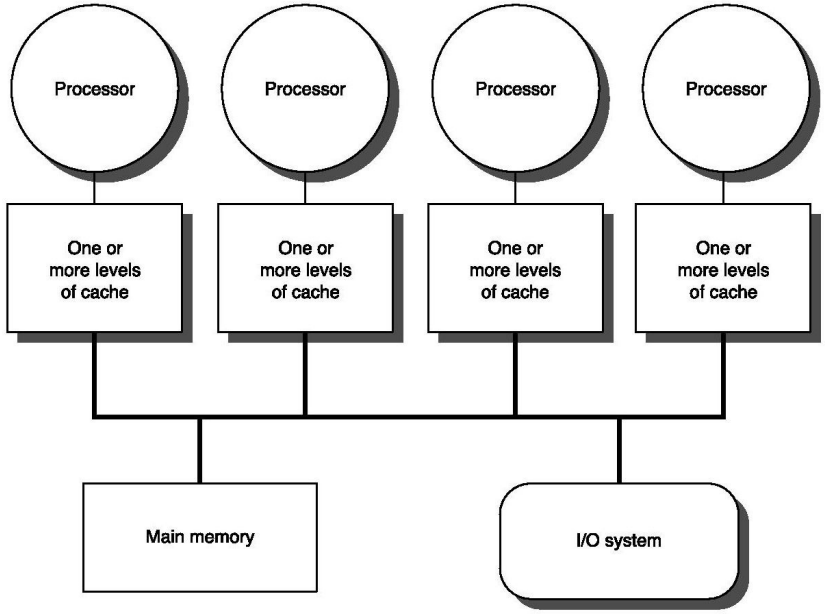
Shared memory multiprocessors

- Migrating a process to a different processor can be costly when each core has a private cache (can you guess why?).
- This is why some OSs, such as Linux, offer a system call to specify that a process is tied to the processor, independently of the processors load (affinity).
- There are three classes of multiprocessors, according to the way each CPU *sees* main memory:

Shared memory multiprocessors

1. **Uniform Memory Access (UMA)**: the name of this type of architecture hints to the fact that all processors share a unique centralized primary memory, so *each CPU has the same memory access time.*

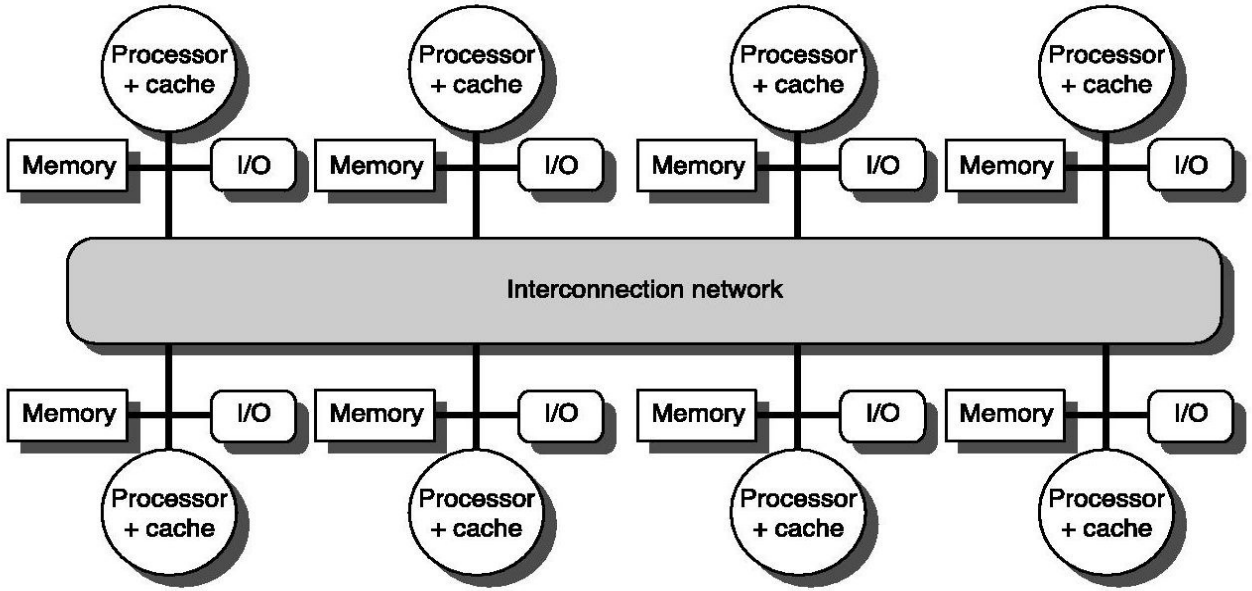
- Owing to this architecture, these systems are also called *Symmetric Shared-memory Multiprocessors (SMP)* (Hennessy-patterson, Fig. 6.1)



Shared memory multiprocessors

2. **Non Uniform Memory Access (NUMA)**: these systems have a shared logical address space, but physical memory is *distributed* among CPUs, so *that access time to data depends on data position, in local or in a remote memory* (thus the NUMA denomination)

- These systems are also called *Distributed Shared Memory (DSM)* architectures (Hennessy-Patterson, Fig. 6.2)



Shared memory multiprocessors

3. **Cache Only Memory Access (COMA)**: data have no specific "permanent" location (no specific memory address) where they stay and whence they can be read (copied into local caches) and/or modified (first in the cache and then updated at their "permanent" location).

- Data can migrate and/or can be replicated in the various memory banks of the central main memory.

UMA multiprocessors

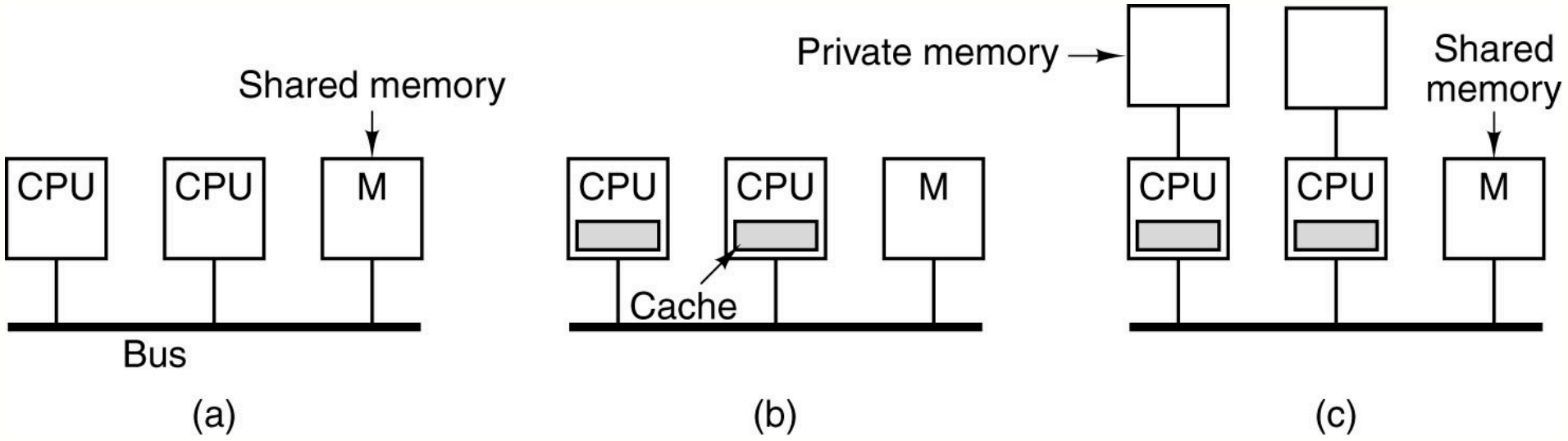
- The *simplest* multiprocessor system has a *single bus* to which connect at least two CPUs and a memory (shared among all processors).
- When a CPU wants to access a memory location, it checks if the bus is free, then it sends the request to the memory interface module and waits for the requested data to be available on the bus.
- **Multicore processors** are small UMA multiprocessor systems, where the first shared cache (L2 or L3) is actually the communication channel.
- Shared memory can quickly become a bottleneck for system performances, since all processors must synchronize on the single bus and memory access.

UMA multiprocessors

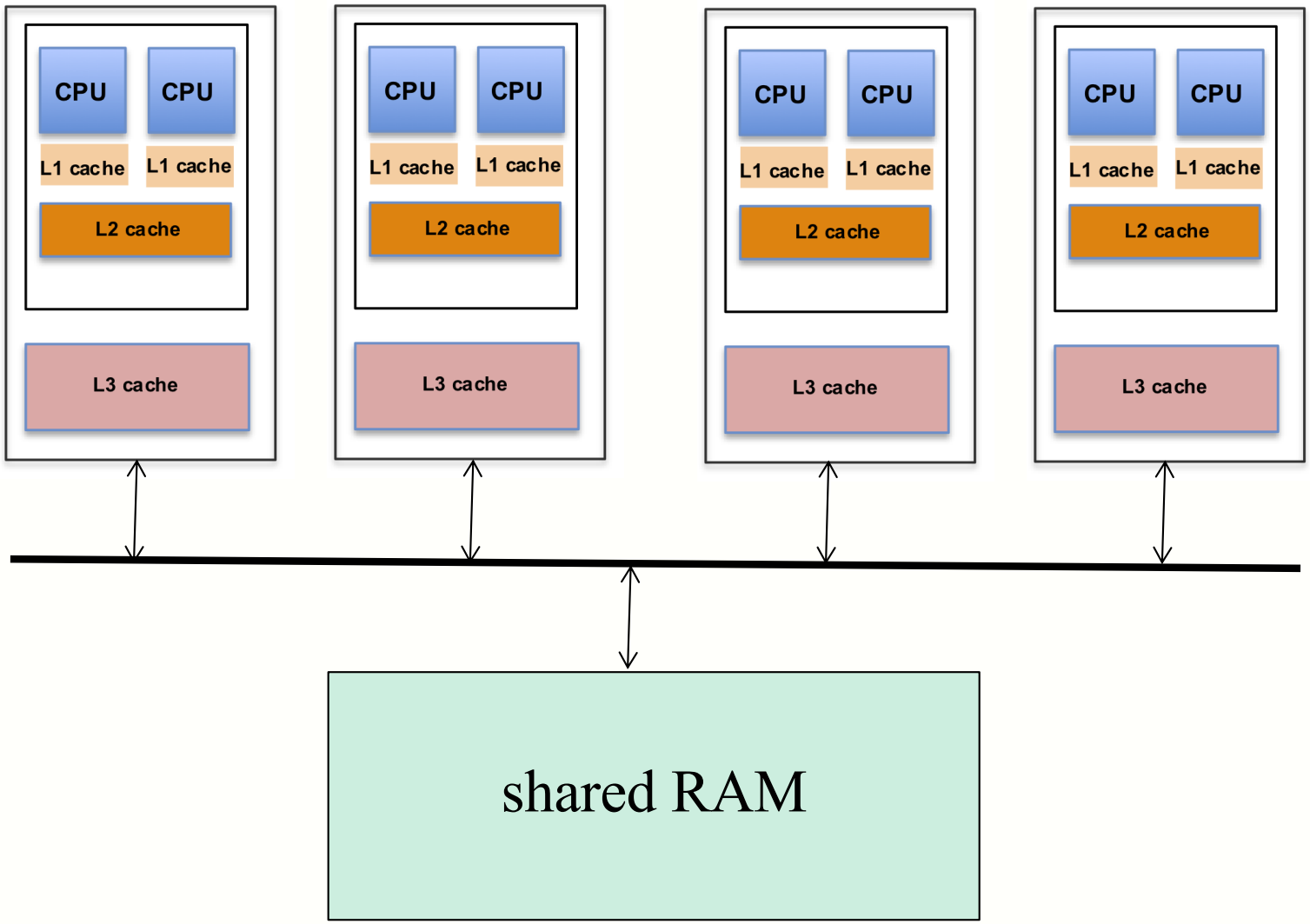
- *Larger* multiprocessor systems (>32 CPUs) cannot use a *single bus* to interconnet CPUs to memory modules, because bus contention becomes un-manegeable.
- CPU – memory is realized through an interconnection network (in jargon “fabric”).

UMA multiprocessors

- Caches local to each CPU alleviate the problem, furthermore each processor can be equipped with a private memory to store data of computations that need not be shared by other processors. Traffic to/from shared memory can reduce considerably (Tanenbaum, Fig. 8.24)



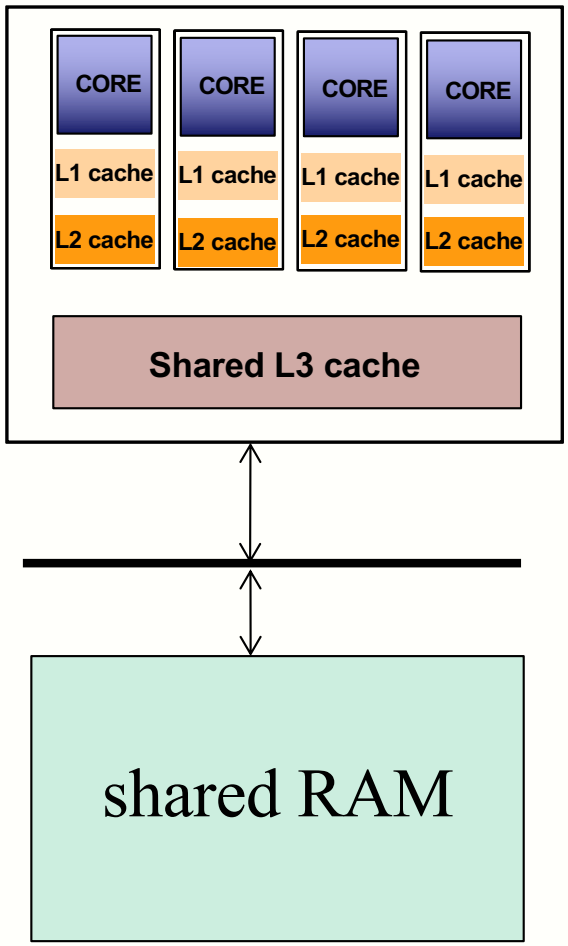
UMA multiprocessors

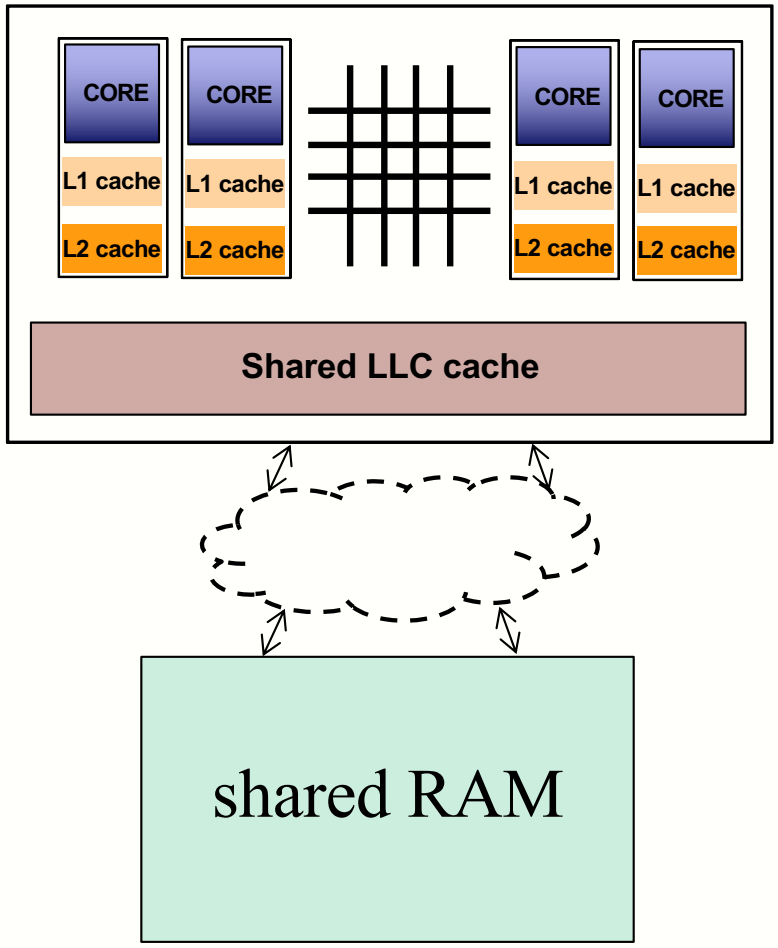


UMA multicores - manycores

multicores: 2 ÷ 36

manycores: ~ 70





Caches and memory in multiprocessors

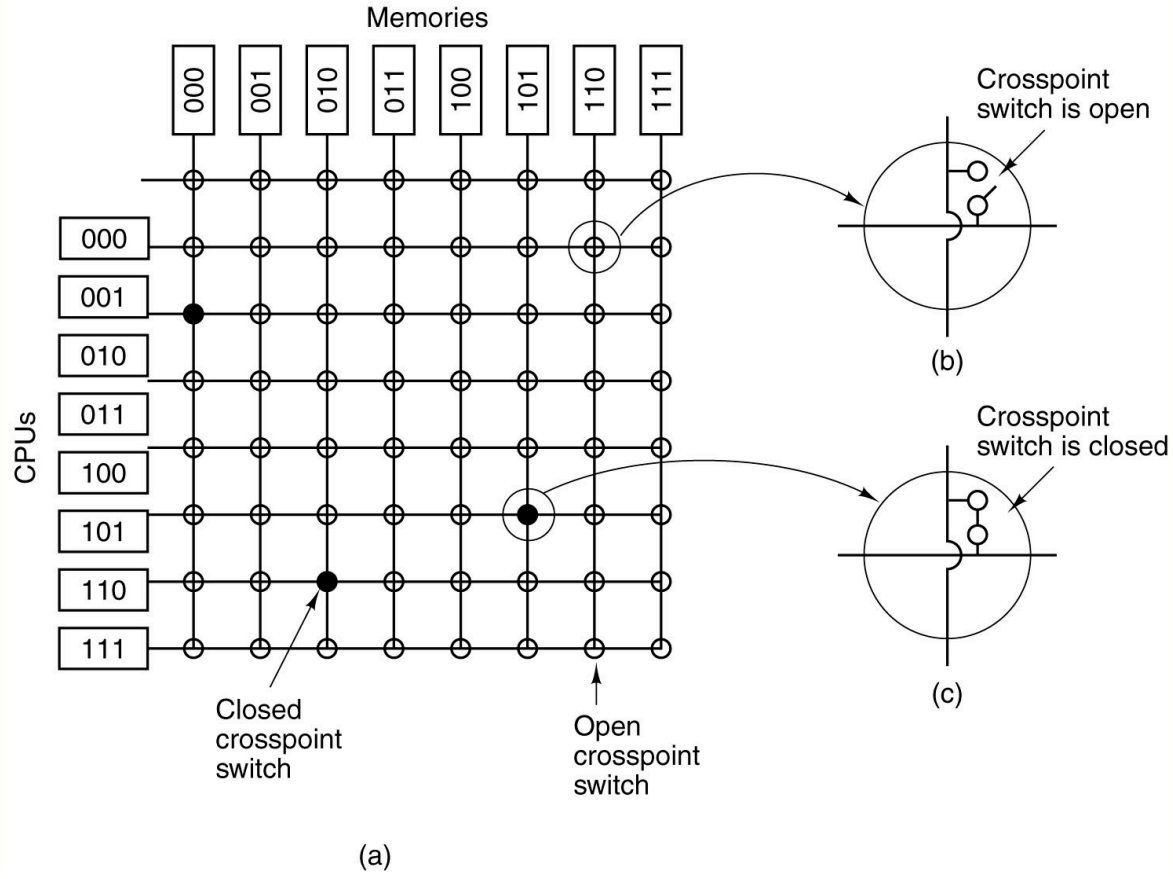
- Memory (and the memory hierarchy) in multiprocessors poses two different problems:
- Coherency: whenever the address space is *shared* – the *same* memory location can have multiple instances (cached data) at different processors
- Consistency: whenever different *access times* can be seen by processors – write operations from different processors require some model for guaranteeing a sound, consistent behaviour (the *when* issue – namely, the *ordering* of writes)

Crossbar switch UMA systems

- Even with a protocol like MESI, a single bus to interface all processors with memory limits the dimension of UMA multiprocessor systems, and usually 32 CPUs (cores) is considered a maximum.
- Beyond this limit, it is necessary to resort to another CPU-RAM interconnection system. The simplest scheme to interconnect *n* CPU with *k* memory is a *crossbar switch*, an architecture similar to that used for decades in telephone switching.

Crossbar switch UMA systems

- A switch is located at each crosspoint between a vertical and a horizontal line, allowing to connect the two, when required.
- In the figure, three switches are closed, thus connecting CPU-memory pairs (001-000), (101-101) and (110-010). (Tanenbaum, Fig. 8.27)

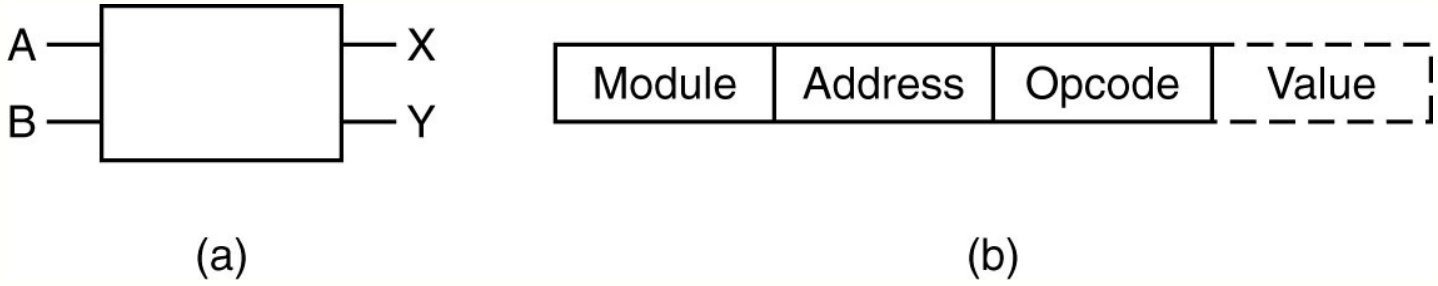


Crossbar switch UMA systems

- It is possible to configure the switches so that each CPU can connect to each memory bank (and this makes the system UMA)
- The number of switches for these scheme scales with the number of CPUs and memories; *n* CPU and *n* memories require $\boldsymbol{n^2}$ switches.
- This pattern fits well medium scale systems (various multiprocessor systems from Sun Corporation use this scheme); certainly, a 256-processor system cannot use it (256^2 switches would be required !!).

Multi-stage crossbar switch UMA

- To interconnect many CPUs, a solution is using a network set up with simple bi-directional switches with two inputs and two outputs: in these switches, each input can be redirected to each output (Tanenbaum, Fig. 8.28):



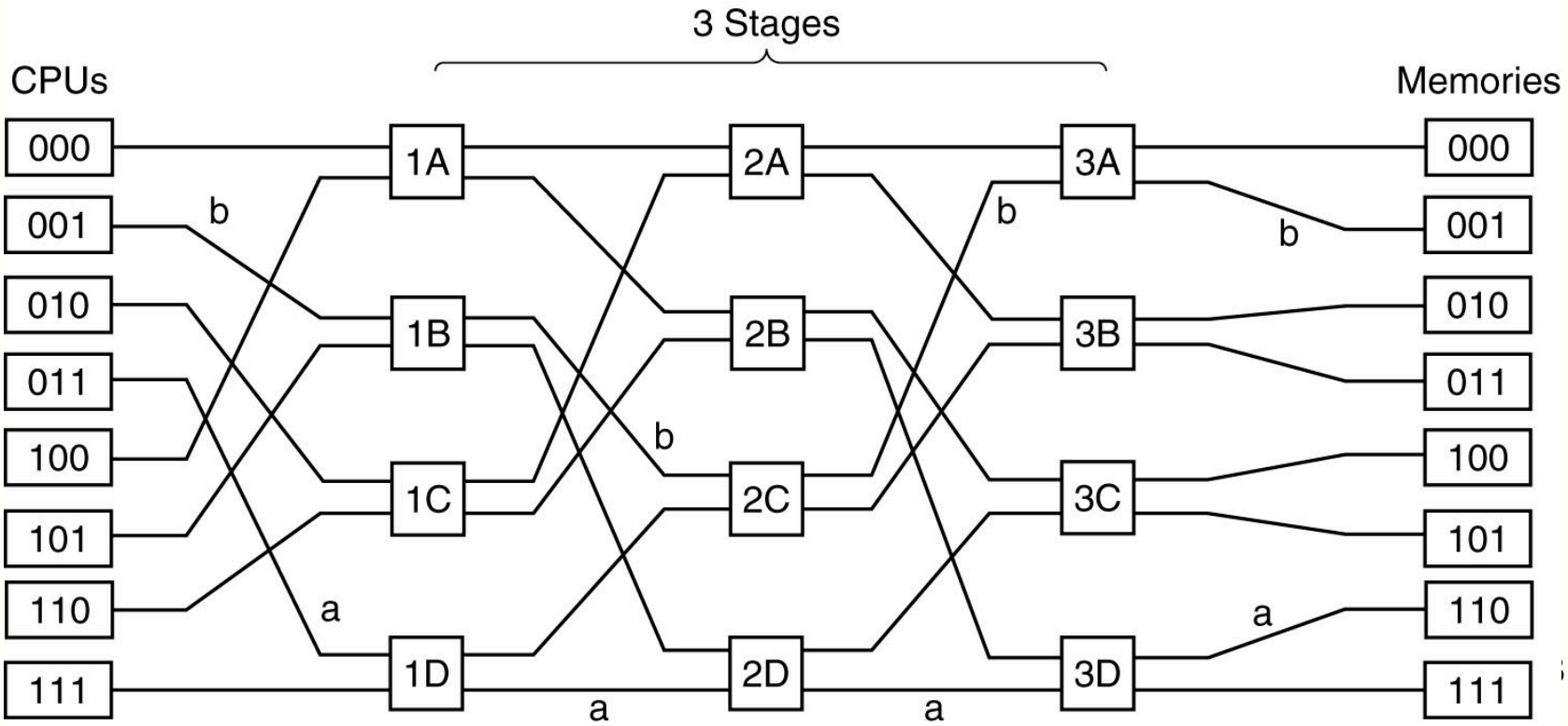
Multi-stage crossbar switch UMA

- Messages between CPU and memory consist of four parts:
- **Module**: *which memory block is requested – which CPU is requestor*
- **Address**: *address within memory block*;
- **Opcode**: *operation to carry out (READ or WRITE);*
- **Value** (optional): *value to be written (for a WRITE).*

- The switch can be programmed to analyse *Module* and to establish the output to forward the message to.

Multi-stage crossbar switch UMA

- 2 x 2 switches can be used in many ways to set up multi-stage interconnection networks. One simple case is **omega network** (Tanenbaum, Fig. 8.29):



Multi-stage crossbar switch UMA

- 8 CPUs are connected to 8 memories, using 12 switches laid out in three stages. Generalizing, n CPUs and n memories require $\log_2 n$ stages and $n/2$ switch per stage, giving a total of $(n/2)\log_2 n$ switches: much better than crossbar switches (n^2).
- Let us see how the network works. CPU 011 wants to read a data item in RAM block 110. the CPU sends a READ request to switch 1D with *Module* = 110 -- 011.
- The switch analyses the most significant (leftmost) bit and uses it for routing: 0 routes on the upper exit, 1 on the lower one.
- In this case, the request is routed to 2D.

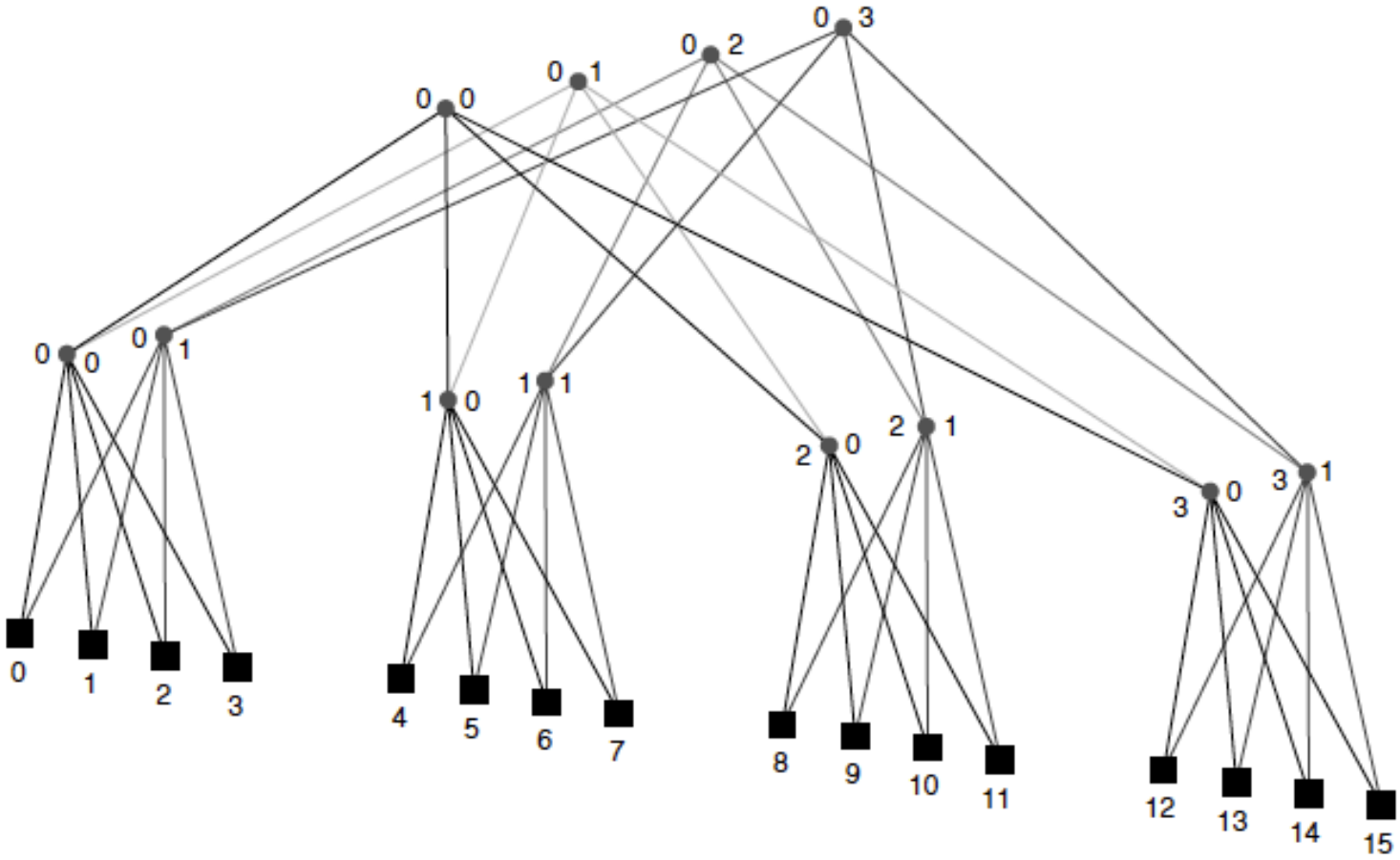
Multi-stage crossbar switch UMA

- Switch 2D does the same: analyses the second-most significant bit (the central one) and routes the request to 3D.
- Finally, the least significant bit is used for the last routing, to block 110 (path *a* in the drawing)
- At this point, the block is read and must be sent back to CPU 011: its“address” is used, but bits are analysed from right to left (least to most significant).
- Concurrently, CPU 001 wants to execute a WRITE in block 001. The process is similar (path *b* in the drawing). Since paths *a* and *b* do not use the same switches, the two requests proceed in parallel and do not interfere with each other.

Multi-stage crossbar switch UMA

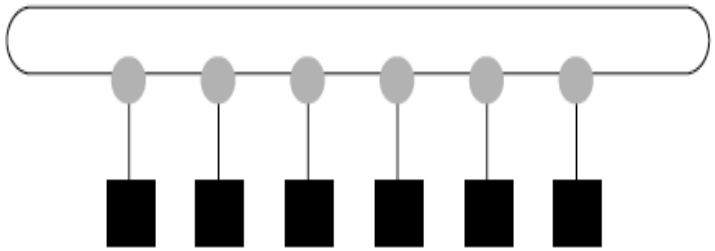
- Let us analyse what happpens if CPU 000 accesses block 000. This request clashes with that by CPU 001 at switch 3A: either request must wait.
- Contrary to crossbar switch networks, **omega networks are blocking**: not all sequences of requests can be served concurrently.
- Conflicts can arise in using a connection or a switch, in accessing a memory block or in answering a CPU request.
- A variety of techniques can be used to minimize the probability of conflicts meanwhile maximizing CPU-memory parallelism.

Other interconnection topologies

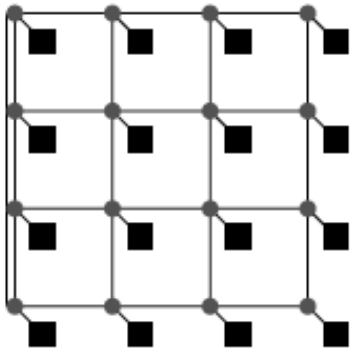


FAT tree (order 4)

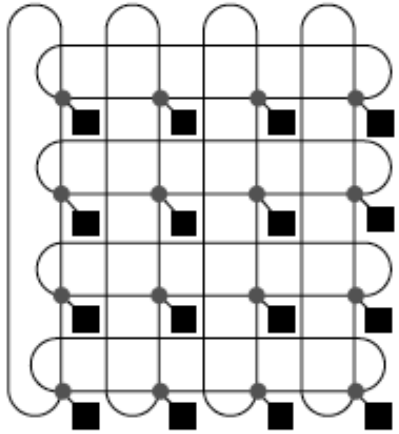
Other interconnection topologies



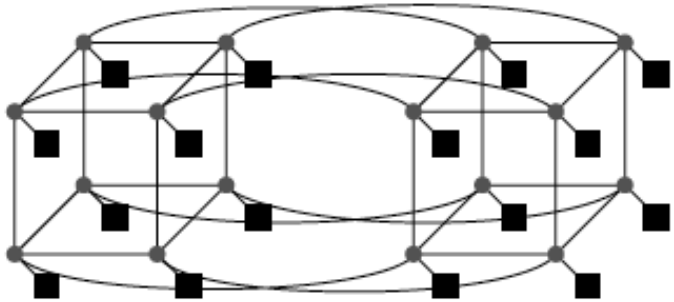
Ring



a. 2D grid or mesh of 16 nodes



b. 2D torus of 16 nodes



c. Hypercube tree of 16 nodes ($16 = 2^4$ so n = 4)

COMA multiprocessors

- In a monoprocessor architecture, as well as in shared memory architectures considered so far, each block, each line are located in a single, precise position of the logical address space, and have therefore an address (“home address”).
- When a processor accesses a data item, its logical address is translated into the physical address, and the content of the memory location containing the data is copied into the cache of the processor, where it can be read and/or modified.
- In the last case, the copy in RAM will be eventually overwritten with the updated copy present in the cache of the processor that modified it.

COMA multiprocessors

- This property (a most obvious one) turns the relationship between processors and memory into a critical one, both in UMA and in NUMA systems:
- In NUMA systems, distributed memory can generate a high number of messages to move data from one CPU to another, and to maintain cohrency in “home address” values.
- Moreover, remote memory references are much slower than local memory ones. In CC-NUMA systems, this effect is partially hidden by the caches (but if many CPUs require a lot of remote data, performances are affected all the same)
- In UMA systems, centralized memory causes a bottleneck, and limits the interconnection between CPU and memory, and its scalability.

COMA multiprocessors

- To overcome these problems, in **COMA** systems (**C**ache **O**nly **M**emory **A**ccess) the relationship between memory and CPU is managed according to a totally different principle.

- There is no longer a “ home address”, and the entire physical address space is considered a huge, single cache.

- Data can migrate (moving, not being copied) within the whole system, from a memory bank to another, according to the request of a specific CPU, that requires that data.

COMA multiprocessors

- This memory management technique raises considerably hit rate, and performances. There are however two basic issues to be addressed:

  1. When a logical address is translated into the corresponding physical one, and the addressed data is not in the cache or in local RAM, where is actually the data item?

  2. When a data item A is brought into the RAM of a CPU, it can be necessary to overwrite an item B (because of lack of free blocks) What happens if that is the last copy of B (something really difficult to know, by the way)?

- Many solutions have been put forward, using hardware and more protocols; these systems (and associated variations) are still being conceived and assessed.

Caches and coherency

- Local caches pose a fundamental issue: each processor sees memory though its own cache, thus two processors can see different values for the same memory location (Hennessy-Patterson, Fig. 6.7)

Time	Event	Cache A	Cache B	RAM location for X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Caches and coherency

- This issue is **cache coherency**, and if not solved, it prevents using caches in the processors, with heavy consequencies on performances.
- Many **cache coherency protocols** have been proposed, all of them designed to prevent different versions of the same cache block from being present in two or more caches (**false sharing**).
- All solutions are at the hardware level: the controller at each cache is capable of monitoring all memory requests on the bus coming from other CPUs, and, if necessary, the coherency protocols is activated.

Caches and coherency

Cache coherency is tackled with differents approaches in UMA and NUMA multiprocessors (and many-core processors).

- UMA multiprocessors have a processors-to-memory pathway that favors bus interconnection, so that cache coherency is based on **bus snooping**
- NUMA multiprocessors rely on complex interconnection networks for processor-to-memory communication, and the only viable solution is based on **cache directories**
- Mixed mode approaches are emerging for MANY-CORE multiprocessors, with chips hosting a fairly large numbers of cores with on die shared caches.

Snooping Caches

- A simple cache coherency protocol is **write through.** Let us review events that happen between a processor accessing data, and its cache:
- **read miss:** the CPU cache controller fetches from RAM the missing block and load its into the cache. Subsequent reads of the same data will be solved in the cache (**read hit**).
- **write miss:** the modified data are written directly in RAM: prior to this, the block containing the data is *not* loaded into local cache.
- **write hit:** the cache block is updated and the update is propagated to RAM.
- Write operations are propagated to RAM, whose content is always updated .

Snooping Caches

- Let us consider now the operations on the side of the snooper in another CPU (right column in table). *cache A* generates read/write ops., *cache B* is the snooping cache (Tanenbaum, Fig. 8.25).
- **read miss**: *cache B* sees *cache A* fetch a block from memory but does nothing (in case of **read hit** *cache B* sees nothing at all)
- **write miss/hit**: *cache B* checks if it holds a copy of the modified data: if not, it takes no action. However, if it does hold a copy, the block containing it is flagged as invalid in *cache B*.

Action	Local request	Remote request
Read miss	Fetch data from memory	
Read hit	Use data from local cache	
Write miss	Update data in memory	
Write hit	Update cache and memory	Invalidate cache entry

Snooping Caches

- Since all caches snoop on all memory actions from other caches, when a cache modifies a data item, the update is carried out in the cache itself (if necessary) and in memory; the "old" block is removed from all other caches (actually, it is simply flagged as invalid).
- According to this protocol, no cache can have inconsistent data.
- There are variations to this basic protocol. As an example, "old" blocks could be updated with the new value, rather than being flagged as invalid ("replicating writes").
- This version requires more work, but prevents future cache misses.

Snooping Caches

- The nicest feature of this cache coherency protocol is simplicity.
- The basic *disadvantage* of **write-through based** protocols is inefficiency, since each write operation is propagated to memory, and the communication bus is likely to become the bottleneck.
- To alleviate the problem, in these protocols not all write operations are immediately propagated to RAM: a bit is set in the cache block, to signal that the block is up-to-date, while memory is "old".
- Sooner or later, the modified block will be forwarded to RAM, possibly after more updates (not after each of them).

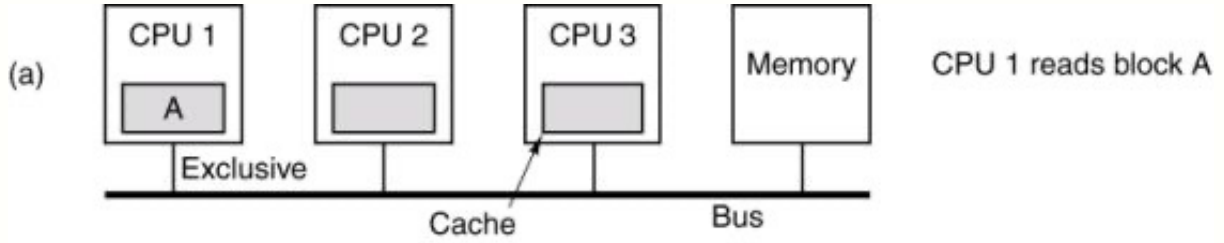
The MESI protocol

- One of the most common **write-back** cache coherency protocoll, used in modern processors, is **MESI**, where each cache entry can be in one of 4 possible states:

1.	**Invalid**	the cache entry does not contain valid data
2.	**Shared**	Multiple caches can hold the block, RAM is updated.
3.	**Exclusive**	No other caches holds the block, RAM is updated.
4.	**Modified**	The block is valid, RAM holds an old copy of the block, no other copies exist.

The MESI protocol

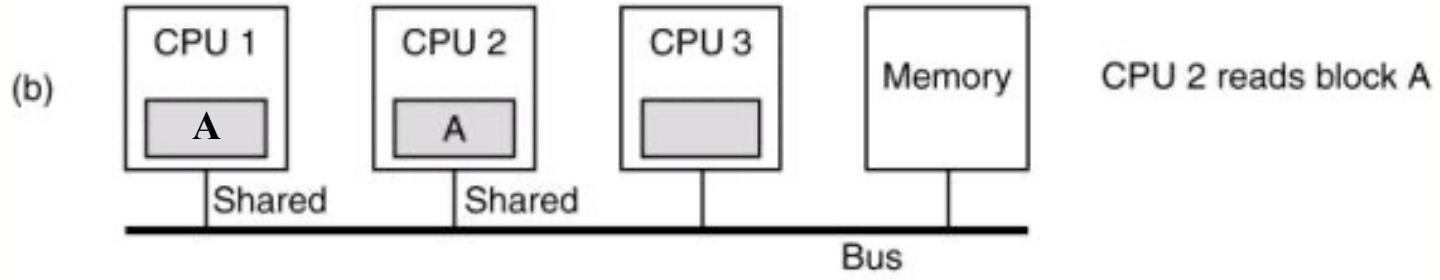
- At system startup, all cache entries are flagged I: Invalid.
- The first time a block is read into the cache of CPU 1, it is flagged E: Exclusive, because the cache is the exclusive owner of the block.



- Subsequent reads of the data item from the same CPU will hit in the cache and will not involve the bus. (Tanenbaum, Fig. 8.26a)

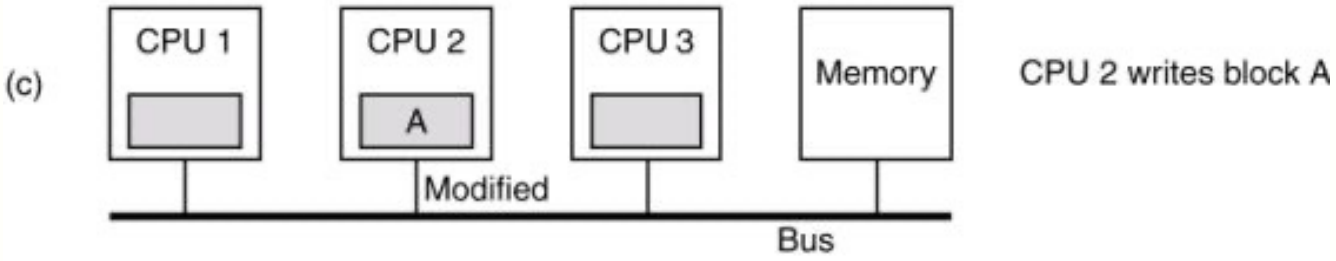
The MESI protocol

- If CPU 2 reads the same block, the snooper in CPU 1 detects the read and signals over the bus that CPU 1 holds a copy of the same buffer. Both entries in the caches are flagged S: Shared.
- Subsequent reads in the block from CPU 1 or CPU 2 will hit in the appropriate cache, with no access to the BUS (Tanenbaum, Fig. 8.26b)



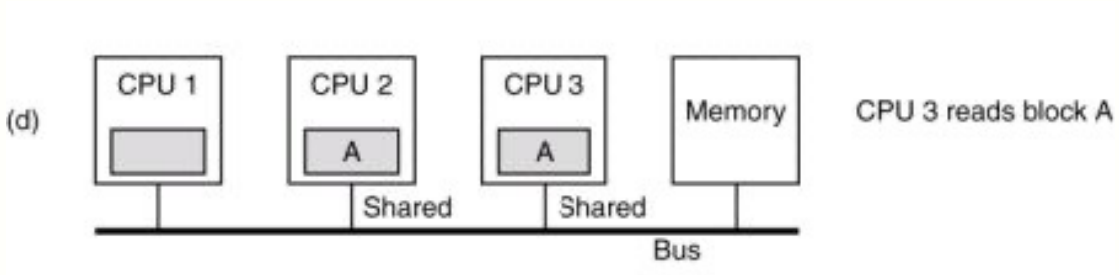
The MESI protocol

- If CPU 2 modifies a block flagged S, it sends over the bus an invalidate signal, so that other CPUs can invalidate their copy. The block is flagged M: Modified, and *it is not written to RAM* (if the block is flagged E, no signal is sent to other caches, since there are no other copies of block in other caches). (Tanenbaum, Fig. 8.26c).



The MESI protocol

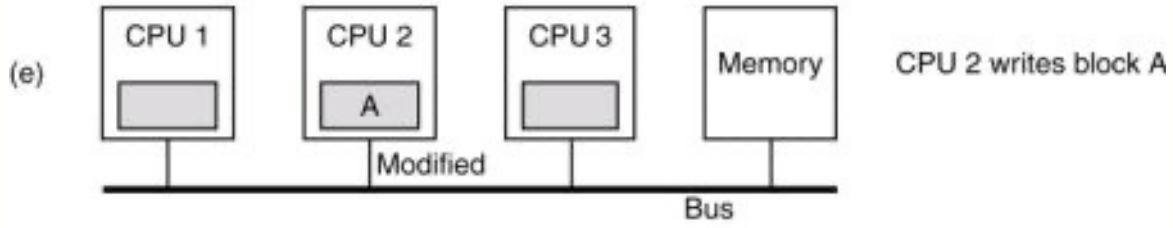
- What happens if CPU 3 tries to read the same block? The snooper in CPU 2 detects this, and holding the unique valid copy of block, it sends a wait signal to CPU 3, meanwhile updating the stale memory with the valid block.



- Once memory has been updated, CPU 3 can fetch the required block, and the two copies of the block are flagged S, shared, in both caches (Tanenbaum, Fig. 8.26.d).

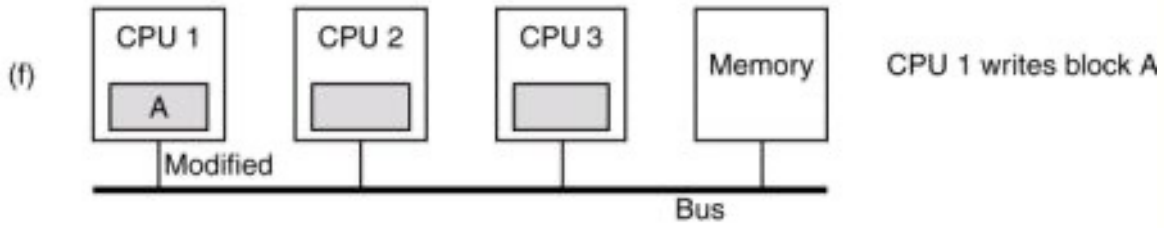
The MESI protocol

- If CPU 2 modifies the block again, in its cache, it will send again an invalidate signal over the bus, and all other copies (such as that in CPU 3) will be flagged I: Invalid. Block in CPU 2 is flagged again M: modified. (Tanenbaum, Fig. 8.26e).



The MESI protocol

- Finally, if CPU 1 tries to write into the block, CPU 2 detects the attempt, and sends a signal over the bus to make CPU 1 wait while the block is written to memory. At the end CPU 2 flags its copy as Invalid, since another CPU is about to modify it.
- At this point CPU 1 is writing a block that is stored in no cache.
- With a **write-allocate** policy, will be loaded in the cache of CPU 1 and flagged M (Tanenbaum, Fig. 8.26f).
- If no write-allocate policy is active, the write directly acts on RAM, and the block continues to be in no cache.



The MESIF protocol

- Developed by Intel for cache coherent non-uniform memory architectures. The protocol is based on five states, Modified (M), Exclusive (E), Shared (S), Invalid (I) and *Forward* (F).
- The F state is a specialized form of the S : it designates the (unique) responder for any requests for the given line, and it prevents overloading the bus due to multiple responder arbitration.
- This allows the requestor to receive a copy at cache-to-cache speeds, while allowing the use of as few multicast packets as the network topology will allow.
- Developed for supporting multi-core in a die, an adopted in recent (2015) many-core processors

NUMA multiprocessors

- As in UMA systems, in NUMA systems too all CPUs share the same address space, but each processor has a local memory, visible to all other processors.
- So, differently from UMA systems, in NUMA systems access to local memory blocks is quicker than access to remote memory blocks.
- Programs written for UMA systems run with no change in NUMA ones, possibly with different performances because of slower access times to remote memory blocks (all other conditions being equal …)

NUMA multiprocessors

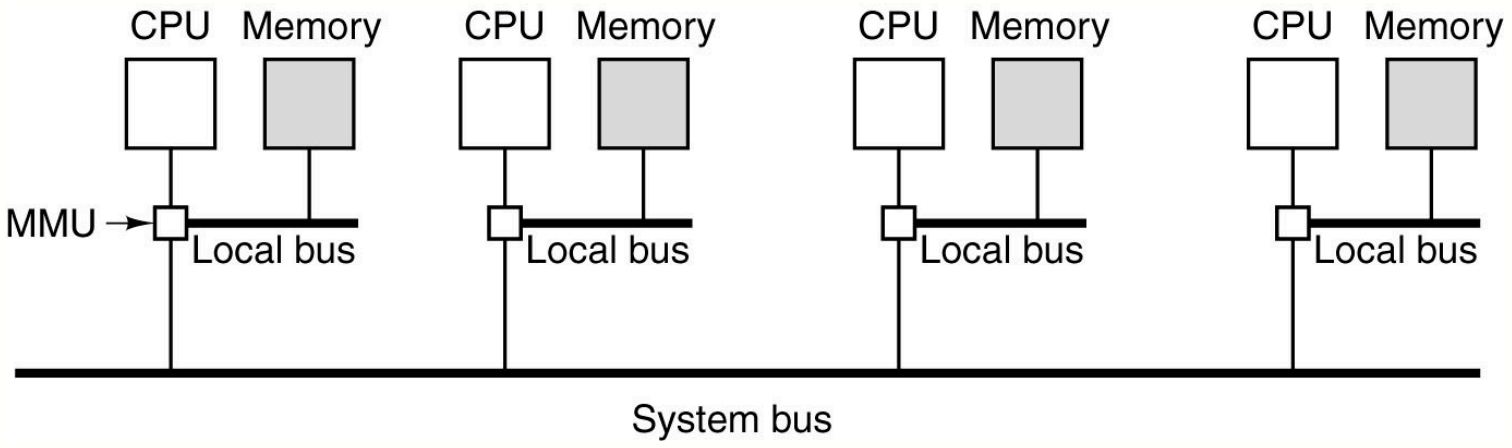
- Single bus UMA systems are limited in the number of processors, and costly hardware is necessary to connect more processors. Current technology prevents building UMA systems with more than 256 processors.
- To build larger processors, a compromise is mandatory: not all memory blocks can have the same access time with respect to each CPU.
- This is the origin of the name **NUMA** systems: **N**on **U**niform **M**emory **A**ccess.

NUMA multiprocessors

- Since all NUMA systems have a single logical address space shared by all CPUs, while physical memory is distributed among processors, there are two types of memories: *local* and *remote* memory.
- Yet, even remote memory is accessed by each CPU with LOAD and STORE instructions.
- There are two types of NUMA systems:
- **Non-Caching NUMA (NC-NUMA)**
- **Cache-Coherent NUMA (CC-NUMA)**

NC-NUMA multiprocessors

- In a NC-NUMA system, processors have no local cache.
- Each memory access is managed with a modified MMU, which controls if the request is for a local or for a remote block; in the latter case, the request is forwarded to the node containing the requested data.
- Obviously, programms using remote data (with respect to the CPU requesting them) will run much slower than what they would, if the data were stored in the local memory (Tanenbaum, Fig. 8.30).



NC-NUMA multiprocessors

- In NC-NUMA systems there is no cache coherency problem, because there is no caching at all: each memory item is in a single location.
- Remote memory access is however very inefficient. For this reason, NC-NUMA systems can resort to special software that relocates memory pages from one block to another, just to maximise performances.
- A page scanner demon activates every few seconds, examines statistics on memory usage, and moves pages from one block to another, to increase performances.

NC-NUMA multiprocessors

- Actually, in NC-NUMA systems, each processor can also have a private memory and a cache, and only private date (those allocated in the private local memory) can be in the cache.
- This solution increases the performances of each processor, and is adopted in Cray T3D/E.
- Yet, remote data access time remains very high, 400 processor clock cycles in Cray T3D/E, against 2 for retrieving data from local cache.

CC-NUMA multiprocessors

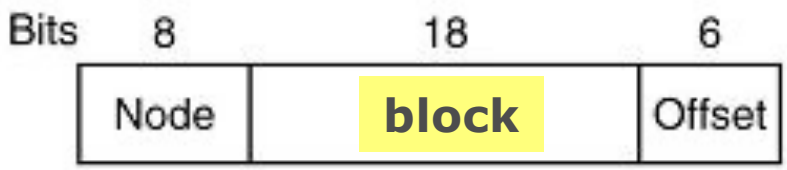
- Caching can alleviate the problem due to remote data access, but brings back the cache coherency issue.
- A method to enforce coherency is obviously bus snooping, but this techniques gets too expensive beyond a certain number of CPUS, and it is much too difficult to implemenet in systems that do not rely on bus-based interconnections. A different approach is required.
- The common approach in **CC-NUMA** systems with many CPUs to enforce cache coherency is the **directory-based protocol**.
- The basic idea is to associate each node in the system with a directory for its RAM blocks: a database stating in which cache is located a block, and what is its state.

CC-NUMA multiprocessors

- When a block of memory is addressed, the directory in the node where the block is located is queried, to know if the block is in any cache and, if so, if it has been changed respect to the copy in RAM.
- Since a directory is queried at each access by an instruction to the corresponding memory block, it must be implemented with very quick hardware, as an instance with an associative cache, or at least with static RAM.
- To illustrate a case of directory based protocol, let us consider a 256-node system, each node equipped with a CPU and 16 MB of local RAM.
- Total RAM memory is $2^{32} = 4$ GB, each node holds 2^{18} blocks each 64 bytes ($2^{18} \times 2^{6} = 2^{24} = 16$ MB).

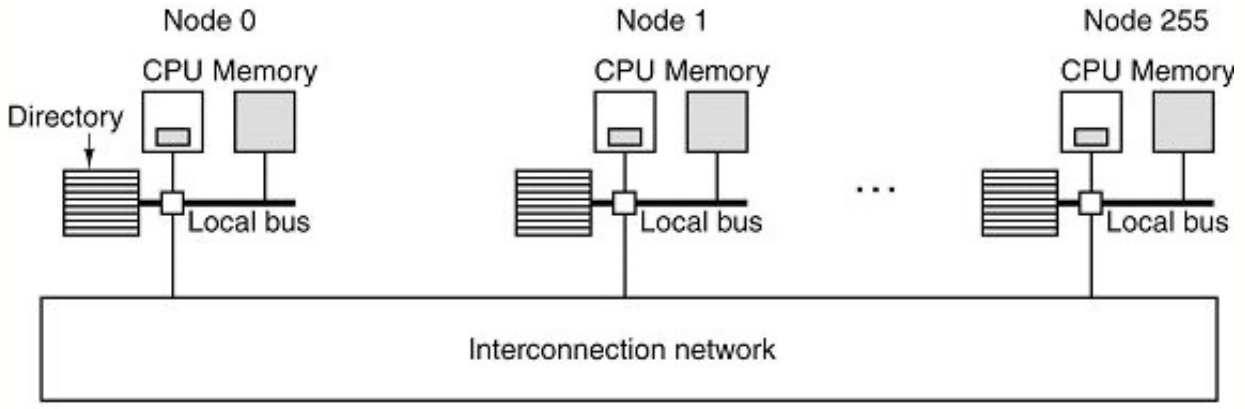
CC-NUMA multiprocessors

- The address space is shared, node 0 containing memory addresses from 0 a 16 MB, node 1 from 16 to 32 MB, and so on.
- The physical address has 32 bits:
- the 8 most significant bits specify the node number holding the RAM block containing the addressed data.
- the subsequent 18 bits identify the block within the 16 MB memory bank
- the remaining 6 least significant bits address the byte within the block (Tanenbaum, Fig. 8.31b):



CC-NUMA multiprocessors

- Let us assume a generic interconnection network to link nodes. Each node has a directory holding 2^{18} entries to keep track of the blocks of the associated local memory.
- Each entry in the directory registers if the block is stored in any cache, and if so, in which node.
- Let us assume that each 64-byte block is stored in a single cache in some processor at most (Tanenbaum, Fig. 8.31a).

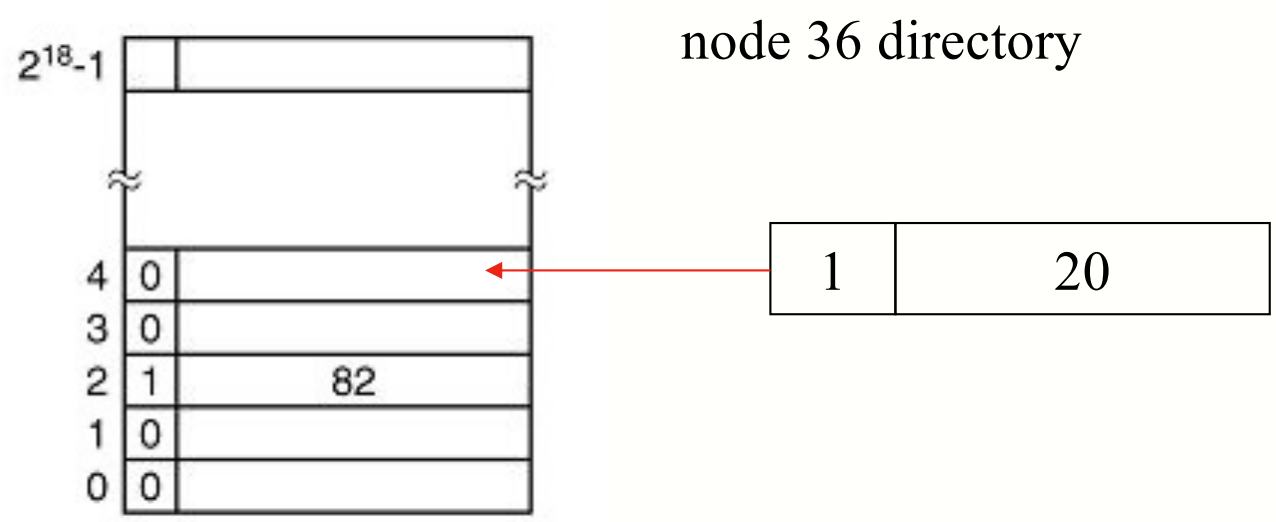


CC-NUMA multiprocessors

- As an example, let us considers what happens when CPU 20 executes a LOAD, thus specifying a RAM address
- CPU 20 forwards the addresss to its own MMU, which translates the LOAD into a physical address, e.g. 0x24000108.
- The MMU splits the address into three parts whose decimal representation are:
  - node 36
  - block 4
  - offset 8
- The MMU sees that the addressed data belongs in node 36, and sends a request through the network to that node, to know if block 4 is in a cache, and which one.

CC-NUMA multiprocessors

- Node 36 forwards the request to its own directory, which checks and discovers that the block is in no remote node cache
- the block is thus fetched from local RAM and sent to node 20, and the directory is updated to register that block 4 is in the cache at node 20 (Tanenbaum, Fig. 8.31c).



CC-NUMA multiprocessors

- Let us now consider the case of a request for block 2 in node 36. Node 36 directory discovers that the block is cached in node 82.
- Node 36 directory updates block 2 entry, to reflect that the block is at node 20, and sends node 82 a message requesting that block 2 is sent to node 20 and that the corresponding entry in node 82 be invalidated.
- When are blocks updated in RAM? Only when they are modified. The simplest solution is doing so when a CPU executes a STORE: the update is propagated to the RAM holding the block addressed by the STORE.

CC-NUMA multiprocessors

- It can be noted that this type of architecture, usually referred to as “shared memory multiprocessor” has a lot of messages flowing through the interconnection network.
- The resulting overhead can be easily tolerated. Each node has 16 MB of RAM, and 2^{18} 9-bit entries to keep track of the status of blocks (why 9?)
- The overhead is 9 x 2^{18} bits / 16 MB, roughly 1,76 %, that can be easily tolerated (even though the directory is a high speed, expensive memory).
- With 32-byte blocks the overhead increases to 4%, while it decreases with 128-byte blocks.

CC-NUMA multiprocessors

- In a real system, this **directory based** architecture is surely more complex:

1. In the example, a block can be in at most one cache, and system efficiency can be increased allowing blocks to be in more caches (nodes) at the same time.

2. by keeping track of the status of a block (modified, untouched) communication between CPU and memory can be minimized.

- For instance, if a cache block has not been modified, the original block in RAM is still valid, and a read from a remote CPU for that block can be answered by the RAM itself, without fetching the block from the cache that holds a copy (since the two copies are identical)

Process synchronization

- In a monoprocessor, processes synchronize using system calls or constructs of the programming language: semaphores, conditional critical regions, monitors.
- These synchronization methods leverage on specific harware synchronization primitives: usually an uninterruptible machine instruction capable of fetching and modifying a value, or of exchanging the contents of a register and of a memory word.
- In a multiprocessor system similar primitives are required: processes share a unique address space and synchronization must use this address space, rather than resorting to message exchange mechanisms.

Process synchronization

- Here is a classical solution to the critical section problem in a monoprocessor, using the atomic ***exchange*** operation. It can be used to build higher level synchronization primitives, such as semaphores and the like:

```
Shared var int lock = 0;                    // lock not set at the
beginning
    int v = 1;
    while ( v == 1) do  exch (v, lock);     //entry section
            critical section                // within this, lock = 1
    lock = 0;                               // exit section
    other code, not mutually exclusive      // lock is released
```

- this is the so called *spin lock*, because the process cycles on the lock variable, until the lock is released.

Process synchronization

- However, in a system where processes trying to synchronize through a shared variable actually run on different CPUs, atomicity of the synchronization instruction is not sufficient.
- On one processor, the atomic instruction will be executed without interrupts, but what about other processors?
- Would it be correct to disable all memory accesses since a synchronization primitive is launched, until the associated variable has been modified?
- It would work, but with a slow down in all memory operations not involved in synchronization (and so far we are ignoring any cache effect ...)

Process synchronization

- Many processors use a couple of instructions, executed in a sequel.
- The first instruction tries to bring to the CPU the shared variable used by all processors for synchronization.
- The second one tries to modify the shared variable, and returns a value that tells if **the couple has been executed in atomic fashion**, and in a multiprocessor this means:

1. no other process has modified the variable used for synchronization before the couple has completed execution, and:
2. no context switch occurred in the processor between the two instructions.

Process synchronization

- Let [0(R1)] be the content of memory word addressed with 0(R1), used as shared synchronization variable.

- **1) LL R2, 0 (R1) // linked load: loads[0(R1)] into R2**
  2) SC R3, 0 (R1) // store conditional: stores [R3] in [0(R1)]

- The execution of the two instructions with respect to [0 (R1)] is tied to what happens in-between :

1. if [0 (R1)] is modified (by another process) before SC executes, SC "fails", that is:
   [0 (R1)] is not modified by SC and 0 is written in R3.
   If SC does not fail:
   R3 is copied into [0 (R1)] and 1 is written in R3.

2. SC fails (with the same effects) if a context switch occurs in the CPU in-between the execution of the two instructions.

Process synchronization

- The two special instructions LL and SC use an “invisible register”, the *link register*, that is a register not part of the ISA specification
- The LL store the memory address of the memory reference in the link register.
- The link register is cleared if
  - the cache block it refers is invalidated
  - a context switch is executed
- The SC checks that the memory reference and the link register match; if so, the LL SC couple behaves as an atomic memory reference
- Care must be taken in inserting other instruction between the LL and SC; only register-register instructions are safe.

Process synchronization

- Let us examine how to obtain an "atomic" exchange between R4 and [0(R1)] in a shared memory multiprocessor system (something similar to a spin lock, by the way):

```
retry:  OR    R3, R4, R0     // copy exchange value R4 in R3
        LL    R2, 0 (R1)     // load linked: load [0(R1)] into R2
        SC    R3, 0 (R1)     // try to store exchange value in [0 (R1)]
        BEQZ  R3, retry      // spin if store failed
        MOV   R4, R2         // now put loaded value in R4
```

- When MOV is executed, R4 and [0 (R1)] have been exchanged "atomically", and we are guaranteed that [0 (R1)] has not been changed by other processes before the completion of the exchange. We define **EXCH** the operation realized though this code.

Process synchronization

- With an atomic operation such as EXCH, it is possible to implement *spin locks*: accesses to a critical section by a processor cycling on a *lock* variable, that controls the mutually exclusive access.
- The lock variable set to 0 or to 1 tells if the critical section is free or occupied by another process.
- Busy waiting to implement critical section is a viable solution only for very short critical sections.
- Very short critical sections can in turn be used to implement high level synchronization mechanisms and mutual exclusion, such as semaphores.
- Busy waiting is less of a problem in multiprocessors. Why?

Process synchronization

- If there were no cache (and no coherency), the lock variable could be left in memory: a process tries to get the lock with an atomic exchange, and checks if the lock is free.
- remember: [0(R1)] = 0 = lock free; [0(R1)] = 1 = lock set

```
        ADD   R2, R0, #1      // initialize the support value in R2
lockit: EXCH  R2, 0 (R1)      // atomic exchange
        BNEZ  R2, lockit      // Tries again if [0(R1)] = 1 = locked
        critical section
```

- To release to lock, (to leave the critical section) the processor writes 0 in [0(R1)].

Process synchronization

- If cache coherency is in place, the lock variable can be kept in the cache of each processor.
- This makes spin lock more efficient, because the operations from processors that try to get a lock work on the caches, more efficiently.
- The spin lock procedure has to be slightly modified: each processors executes a read on the cached copy of the lock, until it finds the lock free.
- At this point, it tries to acquire the lock (that is, entering the critical section) using the atomic exchange.

Process synchronization

```
lockit:  LD      R2, 0 (R1)     // load of lock
         BNEZ    R2, lockit     // lock not available, spin again
         ADD     R2, R0, #1     // prepare value for locking
         EXCH    R2, 0 (R1)     // atomic exchange
         BNEZ    R2, lockit     // spin if lock was not 0
```

- The following chart shows a case with three CPUs working according to MESI (Hennessy-Patterson, Fig. 6.37): once CPU 0 sets the lock to 0, the entries in the other two caches are invalidated, and the new value must be fetched from the cache in CPU 0.
- One of the two gets the value 0 first, and succeds in the exchange, the other processor finds the lock variable set to 1, and start spinning again.

	CPU 0	CPU 1	CPU 2	coherence state of lock	Bus activity
1	has lock	spins, testing if lock = 0	spins, testing if Lock = 0	Shared	None
2	set lock to 0	(Invalidate received)	(Invalidate received)	Modified (P0)	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	change from Modified to Shared	Bus starts serving P2 cache miss; write back from P0
4		(Wait while bus busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes exch	Shared	Cache miss for P1 satisfied
6		Executes exch	Completes exch: return 0 and sets Lock = 1	Modified (P2)	Write invalidate of lock variable from P2
7		Completes exch: return 1 and sets Lock = 1	Enter critical section	Modified (P1)	Write invalidate of lock variable from P1
8		Spins, testing if Lock = 0			None

Process synchronization

- This techniques works with little overhead (that is, spinning does not waist many clock cycles) only in systems with few CPUs, or at least few CPUs that try to get the lock.
- When the number of CPUs that try to get the lock can be “high” (say, > 10 CPUs), other more sophisticated mechanisms come into play, themselves based on the LL and SC synchronization instructions.

Memory consistency models

- Cache coherency makes the various CPUs of a multiprocessor system see main memory in a consistent way.
- However, cache coherency does not tell what this consistency means. That is : **in which order can/must a CPU see the changes on data performed by the other CPUs?**
- Indeed, CPUs communicate through shared variables, and a CPU uses READ to “see” WRITEs form another CPU.
- So, which properties must hold for READs and WRITEs executed by processors on the memory?

Memory consistency models

- Even in the simple case of NC-NUMA system, local and remote memory banks (with respect to CPUs) poses a memory consistency problem.
- Let us consider three CPUs that executes in order and in quick sequence the following three operations on memory word X:
  - CPU A: write #1, X
  - CPU B: write #2, X
  - CPU C: read X
- Which value does CPU C read? Missing a priori hypotheses, it can read 1, 2 or even the value preceding the first write, according to the distance of the three CPUS from the memory bank containing the addressed word.

Memory consistency models

- Generally speaking, at a given time there can be more CPUs trying to read or write the same memory location shared among all CPUs.
- In DSM systems, access requests to the same memory location can overrun each other, thus being executed in an order other than the order of issue.
- Furthermore, each CPU has one/two/three caches, that can hold different copies of the same RAM block, not guaranteed to be updated.
- The outcome can be a total mess, unless tight rules and operative modes are imposed on the memory system with respect to the set of processors that share it.

Memory consistency models

- These rules define what is said a **memory consistency model**; many proposals exist, and hardware support is available for them.
- Some consistency models are more strict, and more difficult to implement, others are more relaxed. In any case, every model must specify what happens in the situations just described.
- Only on the basis of a specific consistency model it is possible to design reliable application for such systems, *by relying explicitly on the behaviour of the memory subsystem.*
- In the following, a description of some among the most common models, without a thorough detailed discussion, keeping in mind that it is the hardware layer that guarantees efficiency.

Strict consistency

STRICT consistency

- It is the most obvious form of consistency:

  "*any read from memory location X returns always the **last** value written in that memory location*".

- Unfortunately, it is also the most difficult one to realize: it requires an interface between memory and CPUS that manages all memory accesses in tight *first come first served* modality.

- Memory thus becomes a bottleneck that slows down dramatically a system built to work as largely as possible in parallel.

- This consistency model is actually NOT implemented at all.

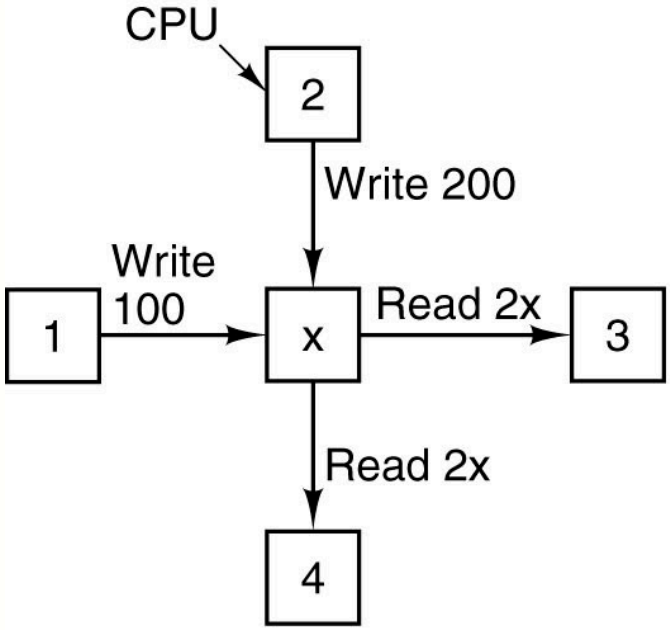
Sequential consistency

SEQUENTIAL consistency

- A more realistic consistency model assumes that, given a sequence of read and write accesses to a memory word, the hardware chooses "*an ordering (possibly in non deterministic way) and all CPUs see the same ordering*".
- As an example, let us consider CPU 1 that writes 100 in memory word X, and immediately after CPU 2 writes 200 to the same memory word.
- After the second write (possibly not yet completed), two more CPUs read twice (each of them) memory word X.

Sequential consistency

- Here are 3 possible orderings ((b), (c) and (d)) of the six ops. : all of them perfectly allowable, but only the first complies with sequential consistency, with CPU 3 e CPU 4 both reading (200, 200) (Tanenbaum, Fig. 8.22).



(a)

(b)	(c)	(d)
W100	W100	W200
W200	R3 = 100	R4 = 200
R3 = 200	W200	W100
R3 = 200	R4 = 200	R3 = 100
R4 = 200	R3 = 200	R4 = 100
R4 = 200	R4 = 200	R3 = 100

Sequential consistency

b) CPU 3 reads (200, 200) – CPU 4 reads (200, 200)

c) CPU 3 reads (100, 200) – CPU 4 reads (200, 200)

d) CPU 3 reads (100, 100) – CPU 4 reads (200, 100)

- These, and other, orderings are possible, because of the possibly different delays in the propagation of operations among CPUs and memory modules.
- But sequential consistency will prevent CPUs from seeing different orderings for events, so only case (b) will be allowed by a protocol enforcing sequential consistency.

Sequential consistency

- Actually, sequential consistency guarantees that, if there are multiple concurrent events,
  a specific ordering is chosen for these events, and this ordering is seen by all processors in the system (a thus by all processes in the system)
- It looks like a simple property, in reality it is very expensive to implement, and reduces potential performances, especially in parallel systems with many CPUs.
- Indeed, it requires imposing an ordering to the operations executed on the shared memory, withholding new operations until older ones have been completed.
- For this reason, more "relaxed" models have been proposed, that are easier to implement and less performance degrading. One of these is "*processor consistency*"

Processor consistency

- It is a consistency model less strict, easier to implement, having two basic properties:

1. *Writes from any CPU are seen by other CPUs in the order of issue*. If CPU 1 writes A, B and C in a memory word X, a CPU 2 that reads X more times in a sequence will read A first, then B and lastly C.

2. *For any memory location, any CPU reads all writes issued by any CPU to that location in the same order.*

Processor consistency

- Let us examine why this consistency model is weaker than the sequential one. Let us assume that CPU 1 writes into *var* values A, B and C, while CPU 2, concurrently to CPU 1, writes into *var* values X, Y and Z.
- According to *sequential consistency*, any other CPU reading more times *var* will read one of the possible combinations of the six writes, as an instance X, A, B, Y, C, Z, *and this same sequence will be seen by any CPU in the system.*
- According to *processor consistency*, different CPUs reading more times *var* **can see different sequences**. It is only guaranteed that no CPU will see a sequence in which B is before A, or Z before Y. **The order in which a CPU executes its writes is seen by all other CPUs in the same way**.

Processor consistency

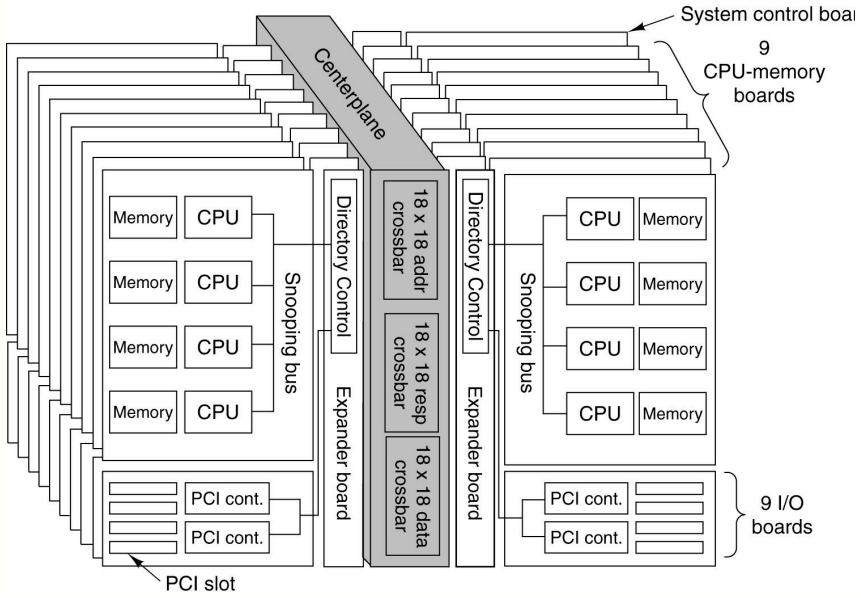
- So, CPU 3 can read: A, B, C, X, Y, Z, while CPU 4 could read X, Y, Z, A, B, C
- CPU 3 can read: A,X,B,Y,Z,C, CPU 4 reads: X,A,B,Y,Z,C
- This consistency model is adopted in many multiprocessor systems.

Sun Fire E25K

- A simple example of CC-NUMA architecture is the family Sun Fire by Sun Microsystems. Model E25K consists of 72 UltraSPARC IV CPUs.
- Each UltraSPARC IV CPU hosts a couple of UltraSPARC III processors sharing cache and memory
- Actually, the Sun Fire E25K system is a UMA / CC-NUMA system.
- The system has been in production until 2009, and it has been replaced by Sun Blade servers, that host more recent multi-core processors.

Sun Fire E25K

- The E25K consists of a maximum of 18 *boardsets*, each set up with:
  - a CPU-memory motherboard
  - an I/O board with four PCI (Peripheral Component Interconnect) slots
  - an expansion board, that connects the motherboard to the I/O board, and both to the centerplane.
- The centerplane hosts the various boards along with the switching circuitry for the communication among nodes



Sun Fire E25K

- Each CPU-memory board contains 4 UltraSPARC IV CPUs and 4 RAM modules, each 8 GB, with a total of 8 UltraSPARC III CPUs and 32 GB RAM.
- A system equipped with all 18 boardsets consists of 144 UltraSPARC III CPUs, 72 memory modules, with a total of 576 GB RAM and 72 PCI slots.
- Each UltraSPARC IV has 64KB + 64 KB L1 cache, 2 MB L2 cache (on-chip) and 32 MB L3 cache.
  - A funny annotation: 18 was chosen because a18-boardset system was the largest to pass a normal-sized door, without being disassembled.

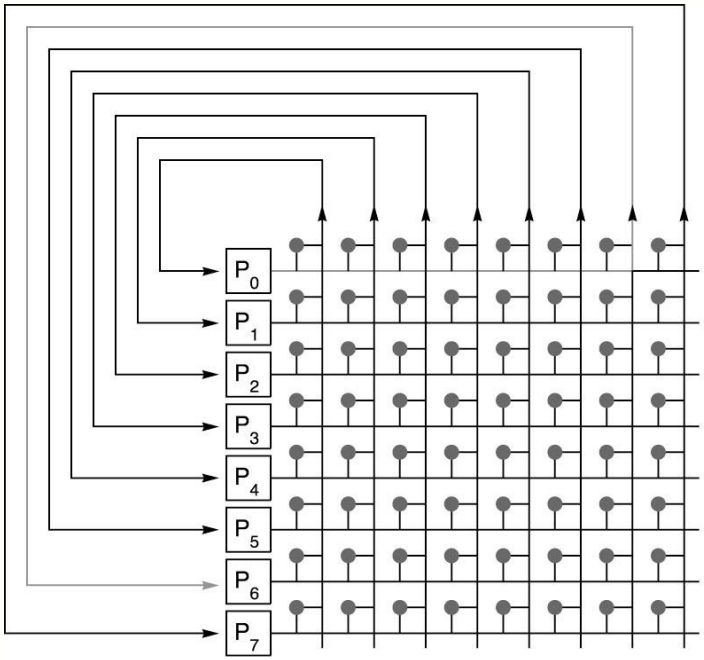
Sun Fire E25K

- The main feature of this architecture is the physically distributed, main memory. Which solution is there to connect 144 CPUs to 72 RAM module, meanwhile assuring cache coherency?
- As already discussed, a shared bus with snooping is no viable solution with many CPUS, it would soon become a communication bottleneck. All the same, a 144 x 72 crossbar switch would be much too difficult and costly to realize.
- In Sun Fire, instead, the centralplane is set up with 3 18x18 crossbar switches, that interconnect the 18 boardsets.
- So, from each (CPU) boardset it is possible to access each other (memory) boardset.

Sun Fire E25K

- Through one 18 x 18 crossbar switch transit the addresses for remote memory requests, through a second one the answers (acknowledges) to the requests, and through the third the actual data. Using three different lines allows to increase the number of parallel accesses to remote memory.

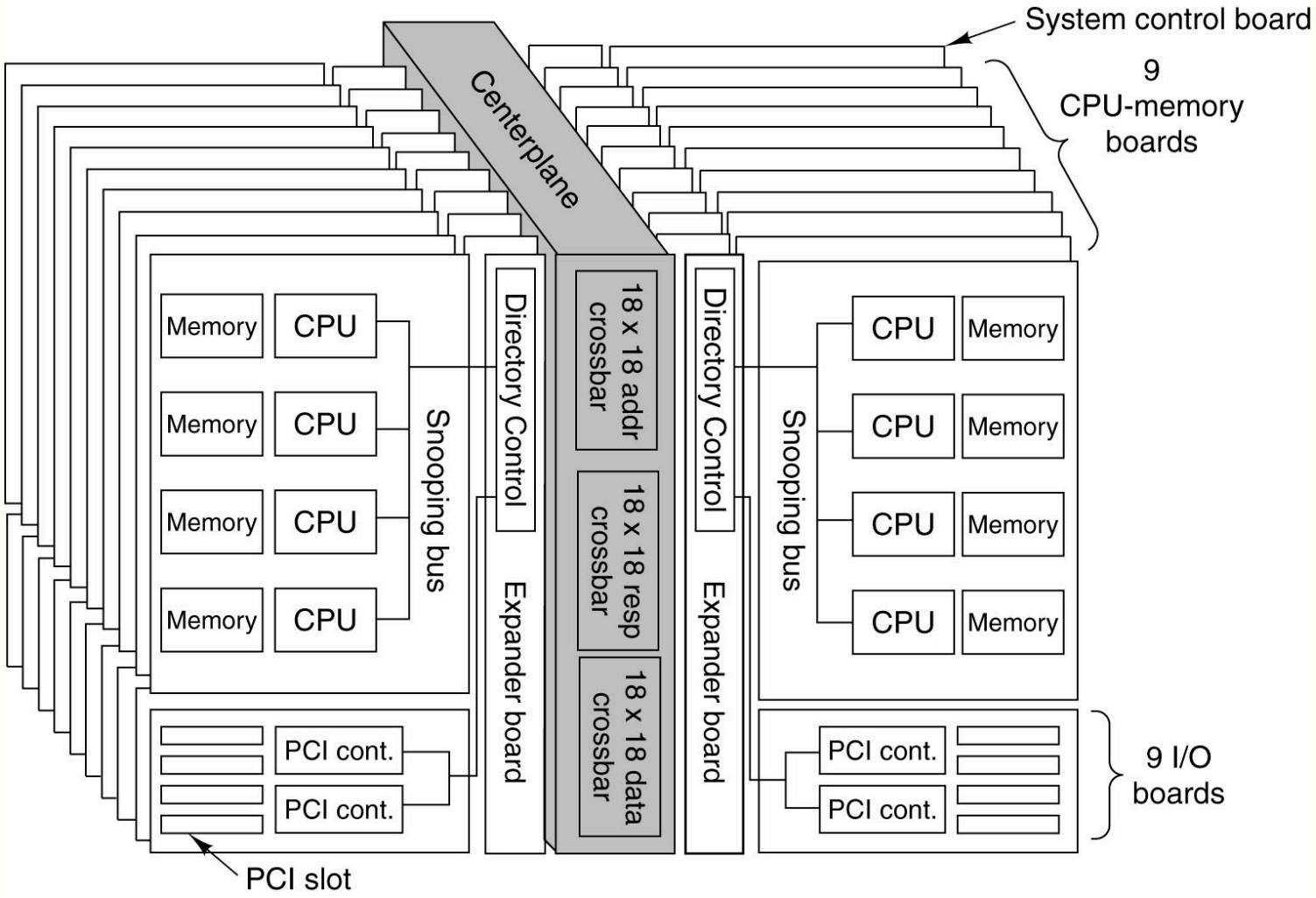
- In the figure, all 8 nodes are interconnected to one another through a 8 x 8 crossbar switch. With two more 8 x 8 crossbar switches, each node could sustain 3 concurrent communications with other nodes (Hennessy-Patterson, Fig. 8.13a)



a. Crossbar

Sun Fire E25K

- Within each boardset, instead, processors are (at least in principle)interconnected with a shared bus with snooping support (Tanenbaum, Fig. 8.32)



Sun Fire E25K

- Right because of this mixed-mode interconnection system, to enforce cache coherency, E25K applies a combined **snooping-directory based protocol**

- Within each boardset, a variation of the MESI snooping protocol (MOESI) controls the 8 local CPUs: when a CPU addresses an item belonging to the memory section hosted on the same boardset, the snooping protocol fetches the data item, while keeping the coherency for all caches in the boardset.

- If the address involves a memory block that does not belong to the boardset, a directory-based, single-directory per boardset scheme is used; it tracks all blocks in the boardset, and manages all requests coming from remote boardsets.

Sun Fire E25K

- With this memory management, Sun Fire E25K can reach high performances, with a minimum throughput of 40 Gb/sec among different boardets (in a fully configured systems with 18 boardsets).

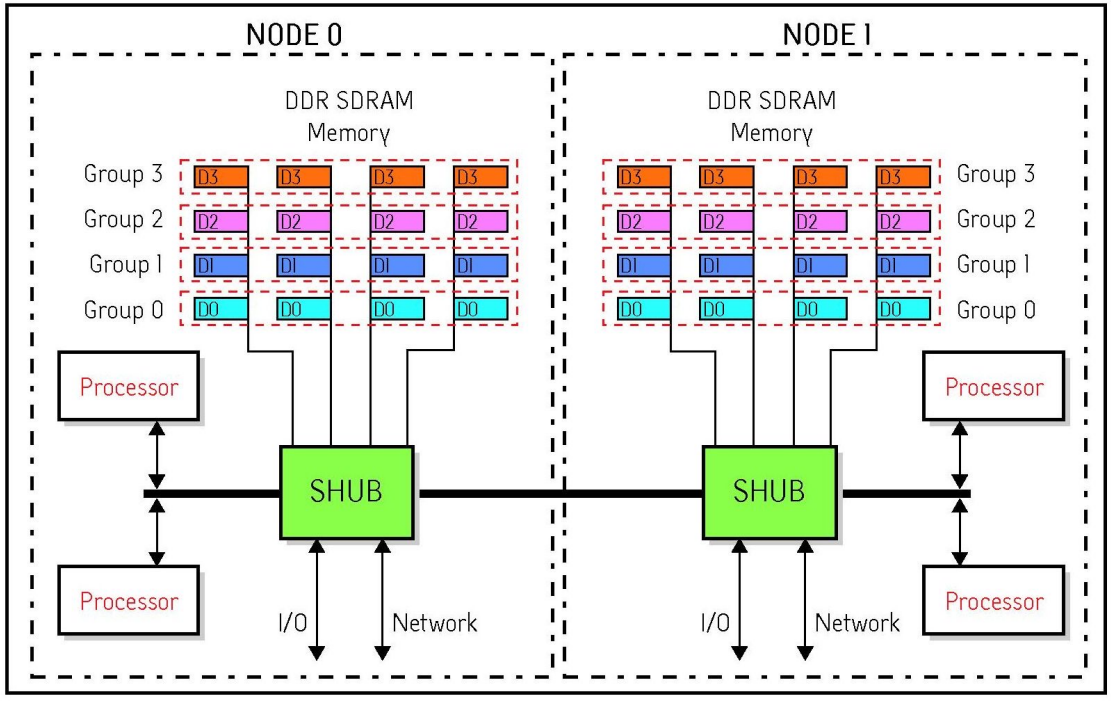
- However, if software is able to allocate pages among memory boards so that most memory accesses from CPUs turn out to be local, system performances benefit most.

SGI ALTIX

- SGI ALTIX systems by SGI (Silicon Graphics Inc.) are CC-NUMA architectures that scale very well, both in the number of processors and in addressable memory.
- ALTIX systems are designed mainly for scientific (and military) applications, while Sun servers (Fire and more recently Blade) are bettered taylored to business applications.
- They are a follow-up to SGI ORIGIN series, that hosted RISC MIPS-R1000 processors, replaced in ALTIX systems by ITANIUM 2 and Xeon.
- There are various systems, starting with ALTIX 330, configurable with a maximum of 16 processors and 128 GB RAM, up to ALTIX 4700, that can accommodate up to 2048 dual core Itanium 2 processors and 32 TB RAM. ALTIX UV (end of 2009), can be configured with 32 to 2048 Xeon cores and up to TB RAM

SGI ALTIX

- An ALTIX systems consists of a set of nodes, the main are named *C-brick*, computetional units hosting 4 Itanium 2 interconnected in couples to a maximum of 16+16 GB RAM.
- A Shared-hub (SHUB) connects each couple of processors to the interconnection network, and implemenets cache coherency.
- Nodes can be also memory banks only (*M-brick*), so that the amount of RAM is not limited by the number of systems in the CPU.



SGI ALTIX

- Itanium 2 was chosen because it can address up to 2^{50} bytes of physical memory, that is one million gigabyte RAM (a thousand terabyte)

- Itanium 2 can manage pages with dimensions ranging from 4KB to 4GB, and has 2 128-entry TLBs to handle separately the translation of logical addresses from instructions and data into physical addresses.

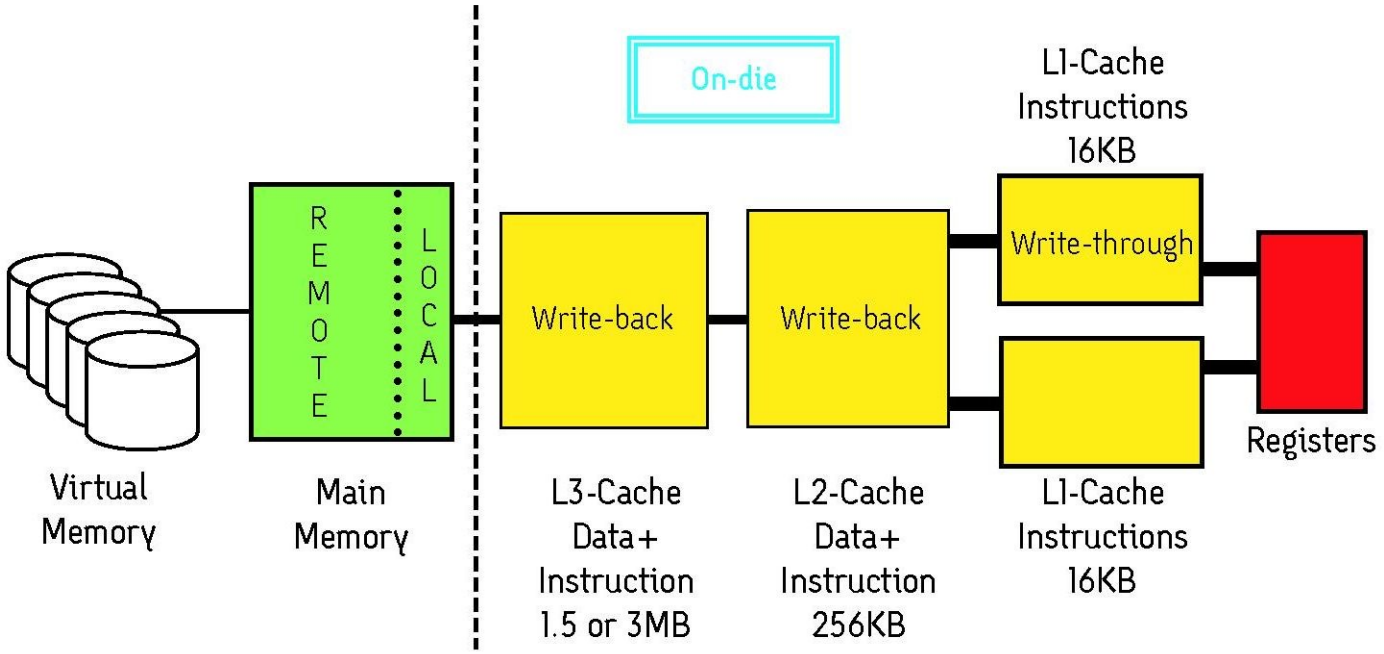
- It is possible (in the best case) to address up to 500 GB istructions and 500 GB data without incurring in “TLB page miss”, a very costly event in a system with physical distributed memoryun evento che è particolarmente pesante da gestire in un sistema a m.

SGI ALTIX

- The cache coherency protocol is realized with the SHUBs, that interface both the snooping logic of the 4 Itanium in a C-brick and directory base protocol used over the interconnection network.
- If the request by one CPU can be satisfied with the content of a cache in another CPU in the node, data are transmitted directly to the cache of the requesting processor without forwarding the request to memory.
- Otherwise, the request is handled by the directory based protocols, managed by the SHUBs, each hosting the directory of the corresponding Itanium couple.

SGI ALTIX

- Itanium Cache hierarchy: note the different snooping protocols used for the caches.



SGI ALTIX

- Nodes are intrconnected with a *fat tree* network (called NUMAlink 4) sustaining 1600 MB/sec through routers: the *R-brick*