Course Notes

High Performance Computing

Marco Vanneschi © - Department of Computer Science, University of Pisa

Part 2

Parallel Architectures

This Part deals with a systematic treatment of parallel MIMD (Multiple Instruction Stream Multiple Data Stream) architectures: shared memory multiprocessors and distributed memory multicomputers. Concepts and techniques are exemplified on several examples of current multi/manycore, or Chip Multiprocessor (CMP), products.

In Section 1 the main characteristics of both kinds of architectures are introduced and discussed. Section 2 to 7 are dedicated to shared memory multiprocessors: the main issues are related to the exploitation of memory hierarchies and cache coherence, processor synchronization, interprocess communication run-time support, and cost models.

Section 3 contains a general treatment of interconnection networks which is valid for distributed memory multicomputers too.

The main issues about interprocessor and interprocess communication in multicomputers are studied in Section 8.

This Part has been written with contributions of

Gabriele Mencagli *(Post-Doc),* ***Daniele Buono*** *(PhD student),* ***Silvia Lametti*** *(PhD student),* ***Tiziano De Matteis*** *(PhD student), and* ***Fabio Luporini*** *(PhD student, London)*

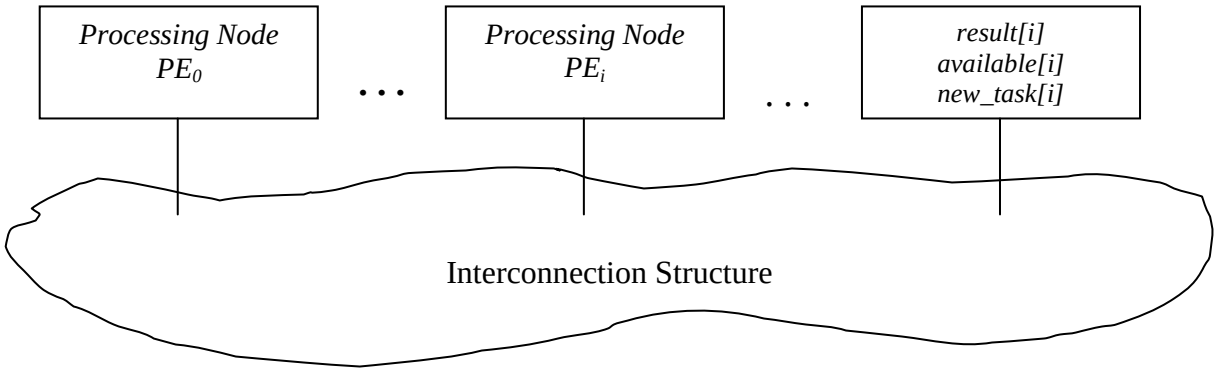
Contents

1. MIMD parallel architectures

MIMD (*Multiple Instruction Stream Multiple Data Stream*) architectures are the most widely adopted high-performance machines. They are inherently general-purpose, and range from medium-low parallelism servers and PC/workstation clusters, to massively parallel (MPP) enabling platforms. Application parallelism is exploited at the process (or thread) level, which acts as the intermediate virtual machine for user-oriented parallel applications, or is the primitive level in which the parallel programming methodology can be applied directly.

Though medium-high end servers and clusters have been used since several years, MIMD architectures are becoming more and more popular owing to the ***multicore/manycore*** evolution/revolution: by now on-chip MIMD architectures are a reality, and the "Moore law" is expected to be applied to the number of processors (cores), accompanied by a corresponding evolution of on-chip interconnection networks.

The overall, simplified view of a MIMD architecture is the following:



where the ***Processing Nodes***, *also called* ***Processing Elements*** *(PE),* can be complete computers (CPU, Main Memory, I/O subsystem), or CPUs possibly with a local memory and/or some limited I/O, or merely Memory modules.

The ***interconnection structure***, or *interconnection network*, is able to connect, directly or indirectly, any pair of PEs to exchange *firmware messages*. Firmware messages are used by the firmware interpreter and by the process run-time support, and must not be confused with messages exchanged between application processes at the process level. That is, firmware messages are an architectural feature for implementing shared memory accesses and/or communications between PEs, according to the architecture class.

For the moment being, we refer to *homogeneous* architectures, i.e. composed of *N* identical PEs. However, heterogeneous architectures are emerging as a powerful alternative, especially for very large platforms configurations and for exploiting the technology evolution at best.

Basically, we distinguish between two main MIMD classes:

1. ***Shared memory*** *architectures*, or ***multiprocessors***,
2. ***Distributed memory*** *architectures*, or ***multicomputers***.

In multiprocessors PEs are CPUs, possibly with local memory and I/O, sharing a physical memory space, while in multicomputers PEs are complete computers which are not able to share a physical memory.

Some kinds of interconnection networks are common to both MIMD classes, while some other kinds are specific, or better exploited, in one of them.

In multiprocessors the interconnection network is used for shared memory accesses and for direct interprocessor communication, while in multicomputers it is used for interprocessor communication only.

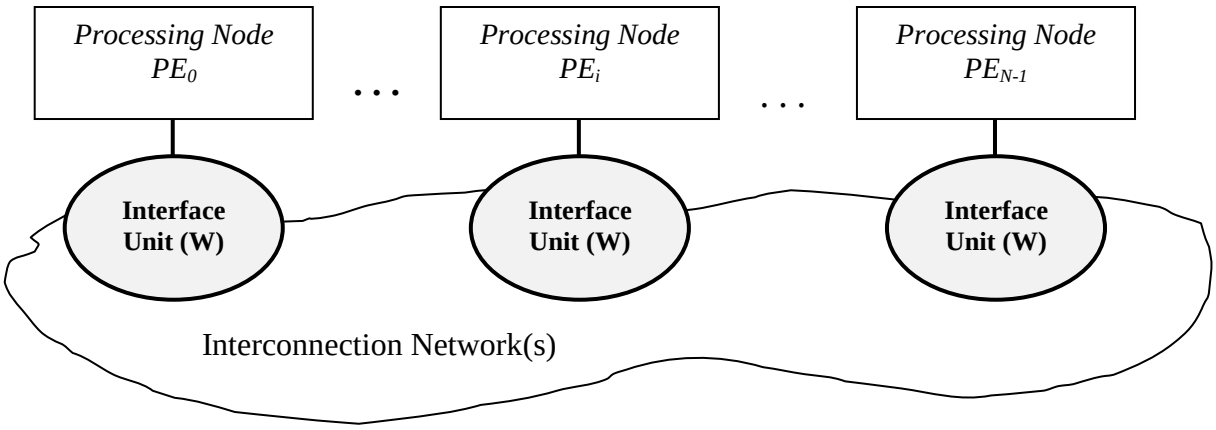
The technological evolution, and the multicore evolution in particular, has led to the realization of ***multiple interconnection structures*** for the same system, each network being dedicated to a specific purpose.

1.1 Processing Elements

In the following, we refer to MIMD architectures whose PEs are general-purpose, *commercial off-the-shelf* (*COTS*) CPUs or computers. This is true also for *multicore* architectures, which often integrate existing CPUs into the same chip. In other words, we wish to study a MIMD machine which is built on top of standard products, which are *basically* uniprocessor machines *with some notable modifications and extensions*.

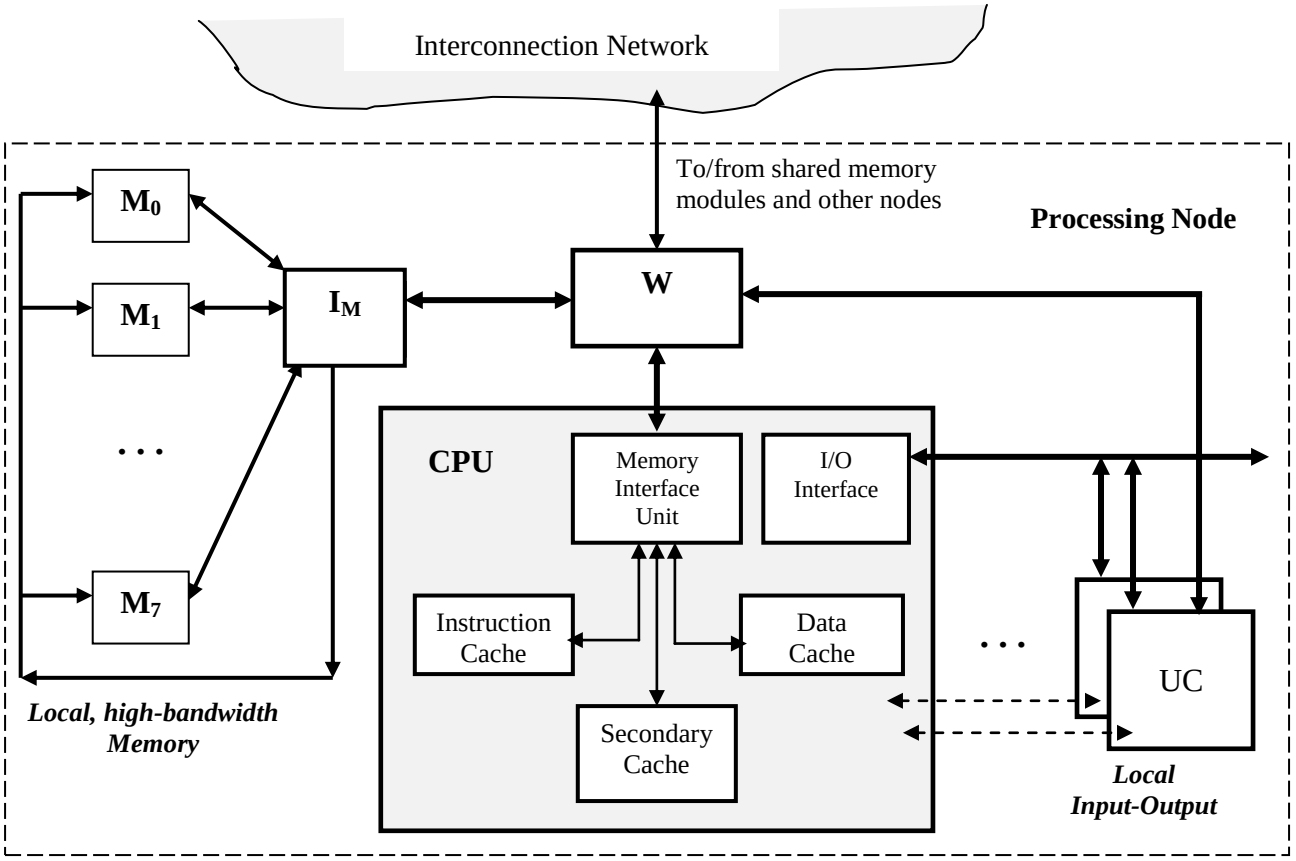
1.1.1 Node Interfaces

We must be able to interface COTS nodes to any kind of MIMD architecture and interconnection network. At least an ***Interface Unit*** must be present for each PE, as shown in the following figure:



The interface unit (W in the figure – W could be the initial of "Wrapping unit) plays several important architecture-dependent roles, according to the specific class of MIMD architecture.

For example, consider a possible scheme of a PE in a shared memory multiprocessor, as shown in the next figure.



As known (Part 0), a CPU is connected to the "rest of the world" through its *primitive external interfaces*, in the figure the *External Memory (Main Memory) Interface* and the *I/O Interface* (e.g., I/O Bus).

This scheme is valid for multicore CPUs too, where the Memory Interface(s) and the I/O Interface(s) are common to all, or subsets of, internal processors.

The interface unit W is directly connected to the Memory Interface, so it is able to intercept all the external memory requests and to transform them into proper firmware messages to/from the various sections of the architecture.

Typical firmware message are:

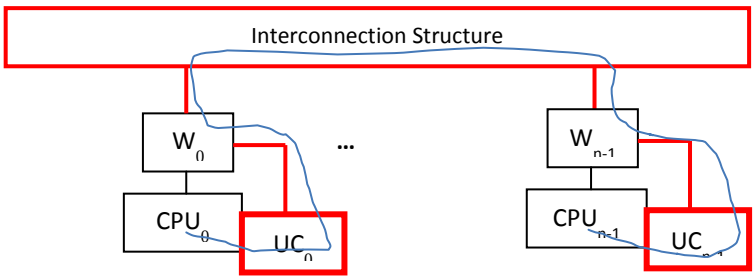
i) *external firmware messages*: through the interconnection network all the shared memory supports are visible, all well as some specific I/O units belonging to the other nodes (Memory Mapped I/O);

ii) *internal firmware messages*, exchanged with the local memory (where present) and the local I/O inside the same PE.

For example, in case *i)*, consider a read-request directed to the main memory (it may be the request of a single word or, more in general, of a cache block). According to the physical address, W is able to distinguish whether the request has to be forwarded: *a)* to the local memory or to the local I/O, or *b)* to the external shared memory. In case *a)* very few modifications are done to the firmware message received from the CPU (physical address, memory operation type, and other synchronization or annotation bits). In case *b)*, the request is transformed into a firmware message (consisting in one or few words) containing all the architecture-dependent information: the information received by the CPU is enriched by the message *header*, i.e. *network routing and flow control information*

(source node identifier, destination node identifier, message length, message type, and possibly other information), some of which are derived from the CPU request itself (e.g. the destination memory module identifier is a part of the physical address). In both cases *a)* and *b)*, W is able to serve other requests simultaneously, possibly from other nodes, including the reply from the main memory (single word or cache block) which is returned to the CPU. In some architectures, the Interface Unit could also be a complex subsystem, consisting of several units.

In the previous figure of a multiprocessor node, an additional I/O unit, called *Communication Unit* (UC), is provided for direct *interprocessor* communications support:



Though, in a multiprocessor, the majority of run-time support information are present in shared memory, there are some cases in which asynchronous I/O messages are preferred or even necessary. The main types of interprocessor communications are:

- for processor synchronization, i.e. for the implementation of some locking mechanism,
- for process low-level scheduling, notably for decentralized process wake-up or processor preemption,
- in some advanced systems, for primitive support to interprocess communication itself.

Such firmware messages might exploit *dedicated interconnection structures*, distinct from the shared memory interconnection. That is, the technological trend is towards ***multiple interconnection networks***, each one dedicated to a specific task. This is motivated by the requirement of high bandwidth and by the different traffic distributions for different tasks: for example, the shared memory access traffic is rather uniform with respect to PEs and shared memory modules, while other traffics tend to be distributed in bursts.

Let us assume that CPUs are *memory mapped I/O* machines (e.g., as in D-RISC and in the majority of commercial CPUs). An interprocessor communication from node PE_i to node PE_j begins with the transfer of the message from CPU_i to UC_i through one or more Store instructions. UC_i communicates the message to UC_j through the node interface unit and the interconnection network. UC_j transforms the received message into an interrupt to CPU_j, where the interrupt handler will be executed as usually.

In a *multicomputer*, the I/O interface is the point through which a PE cooperates with the others by communications, thus *W coincides with UC* only. The situation is conceptually similar to the previous figure, where the Communication Unit plays the role of, at least, the *"Network Card"* to which several communication support tasks or protocols are delegated. In a multicomputer, firmware messages through W/UC are larger than in a multiprocessor and more frequent, as they are the unique way for PEs cooperation, thus more intensive use of Memory-Mapped I/O and DMA is done.

1.1.2 CPU facilities for MIMD architectures

The COTS uniprocessor reutilization is not always free or easy. Some *additional MIMD facilities*, notably for the correct and/or efficient implementation of the firmware parallelism and of the process run-time support, must be present in the corresponding uniprocessor architecture *at the assembler and/or firmware level*. In other words, the processor is explicitly designed with the MIMD utilization in mind (this is the current trend), or at least a certain knowledge of the potential utilization in a MIMD architecture is *foreseen*. Notable examples of facilities for shared memory architectures are:

- especially for highly parallel machines, the *physical address space* must be extendable to very large capacities, for example 1 Tera (40-bit physical address) till 256 Tera (48-bit physical address). Though, at least in principle, this issue has no impact at the assembler machine level, it requires proper firmware support, notably in MMU and in the chip interface, and the proper definition of the address translation function;
- *cache management* multiprocessor-dependent options require special instructions or annotations, and or special firmware facilities which are not present in uniprocessor, notably for *cache coherence*;
- *locking mechanisms*, and other synchronization mechanisms, require special assembler instructions or annotations in assembler instructions, as well as proper information at the CPU interface;
- *memory consistency* mechanisms could be needed, depending on the way in which the effects of Load-Store sequences, executed by a node on shared memory, are rendered visible to the other nodes.

1.2 Process-level implementation of parallel programs

In Part 2 we will fully exploit our background of Part 0 and of Part 1: firmware structuring, assembler features, CPU architectures, caching, processes, addressing spaces, interprocess communication, and process run-time support.

In the following, at the process level we will not distinguish between *processes* and *threads*, unless "hardware" multithreading (Part 1, Section 22) is provided and we are interested in its exploitation. In some systems, the thread mechanism is the main, or the only, way to express processes, even with single-threaded processors, especially when the primitive cooperation model is the shared variables one. In such cases, the distinction is just a matter of terminology, and we will speak about *processes*.

1.2.1 Process run-time support: exclusive mapping *vs* multiprogrammed mapping

In Part 0, Section 6, we have studied solutions to the interprocess communication run-time support for uniprocessor machines. Though they represent the base, the implementation techniques must be reviewed and characterized for MIMD machines, both for correctness and for performance reasons. Notable issues are:

a) indivisibility of primitives,

b) cache hierarchy exploitation,

c) process low-level scheduling.

Points *a)* and *b)* characterizes shared memory multiprocessors. Both for multiprocessor and for multicomputers, point *c)* could be dealt with according to strategies which often are quite different from the uniprocessor machines. According to the parallel application development methodology of Part 1, we know that a *"**one process per processor**"* mapping is frequently adopted. In fact:

i. only in the "traditional" utilization of parallel machines the general concept of "multiprogrammed" PEs is exploited, i.e. the same PEs is shared by more than one process, relying on the existence of ready lists and context-switching functionalities. This is the typical scenario in which

   a. the parallel architecture is merely used for executing *independent sequential programs*, and parallelism issues are mainly due to the execution of *operating systems services*, or

   b. *parallel applications* are executed, but degradations of the completion time due to context-switching, and other processor management functionalities, are accepted. Such degradations are not easily predicted, so costs models become less reliable;

ii. instead, when a "truly" parallel application is executed, a fraction of *n* PEs (even all PEs) of the system are *exclusively* allocated to the application at loading time, where *n* is the actual degree of parallelism of the parallel program (the optimal one, if possible, or a lower value if parallel programs are parametrically scalable). In this scenario, the suspension of a process and the processor context-switching are unnecessary or useless. Better performances are achieved by using ***busy waiting*** only, analogously to what happens in I/O processing (Part 0, Section 3.6). For example, let us assume that A and B are the only processes allocated to nodes PE_i and PE_j respectively: if A waits for a message from B, then the waiting condition of A is implemented as a busy waiting state, without context-switching, i.e. A continues to "occupy" the processor by executing instructions (e.g. a loop testing the presence of the message, or a loop of NOPs) and/or waiting some external signaling. Proper firmware and assembler mechanisms might be provided for PEs cooperation in order to efficiently implement the busy waiting condition and in order to unblock a process when the waited event occurs.

In the following,

- the term ***exclusive mapping*** will be used for scenario *ii)*,
- while scenario *i)* will be called ***multiprogrammed mapping***.

A good implementation of a concurrent language, or process cooperation libraries, should provide the *process run-time support for both mapping strategies*: for example, an implementation of send-receive primitives with busy waiting and an implementation with context-switching. The compiler (or user) will select the most proper implementation according to the kind of utilization and application.

The exclusive mapping approach has some impact on the executable version of processes, thus on compilation and loading, because now the process virtual memory is different, e.g. some shared objects (Ready Lists, PCB, and so on) do not exist.

1.2.2 Simultaneous multithreading

For our purposes, the thread mechanism is meaningful and important when CPUs are multithreaded. As discussed in Part 1, Section 22, in multithreaded architectures we have an additional chance for parallelism exploitation, i.e. the thread-level parallelism inside the same CPU which, potentially, is equivalent to a small multiprocessor.

A system with N PEs, each of which is w-way multithreaded, is equivalent to a system with $N1 = e\ w\ N$ PEs, where e is the multithreading efficiency (i.e. ew is the multithreading scalability). For example, if $w = 2$, we have $ew = 1.4$ in typical benchmark suites (Part 1, Section 22.2): if these suites are representative of the application field, we could be able to exploit $N1 = 1.4 \times N$ nodes in parallel applications.

Several existing MIMD machines provide PEs with 2-way or 4-way simultaneous multithreading. The exploitation of this facility (for increasing the parallelism degree of parallel programs, or for concurrent implementation of run-time support functionalities, see Part 1, Section 22.4.3) is an advanced feature of current and future machines and a research area with several open issues.

1.3 Interconnection networks

As anticipated in Part 0, Section 2.4.5, and in Part 1, Section 18.4, ***buses*** and ***crossbars*** are interconnection solutions that can, and currently are, adopted for systems with a relatively low number of processing nodes. Examples are shared memory multicore chips with 2, 4, 8 PEs.

In general, highly parallel architectures utilize ***limited degree networks***, in which a processing node is directly connected to only a small subset of nodes, or it is indirectly connected to any other node through an intermediate path of switching nodes with few neighbors. Correspondingly, interconnection networks can be classified in:

- *direct* networks,
- *indirect* networks.

In a ***direct*** network, point to point dedicated links connect PEs in some fixed topology. Of course, messages can be routed to any PE which is not directly connected. Each *network node*, also called ***switch node*** (or simply *switch*), is connected to one and only one PE, possibly through the node interface unit (logically, W is not necessary if the network nodes play this role too). Notable examples of limited-degree direct networks for parallel architectures are:

- *Rings*,
- *Meshes* and *Tori* (toroidal meshes),
- *Cubes* (*k*-ary *n*-cubes),

shown in figure 1.

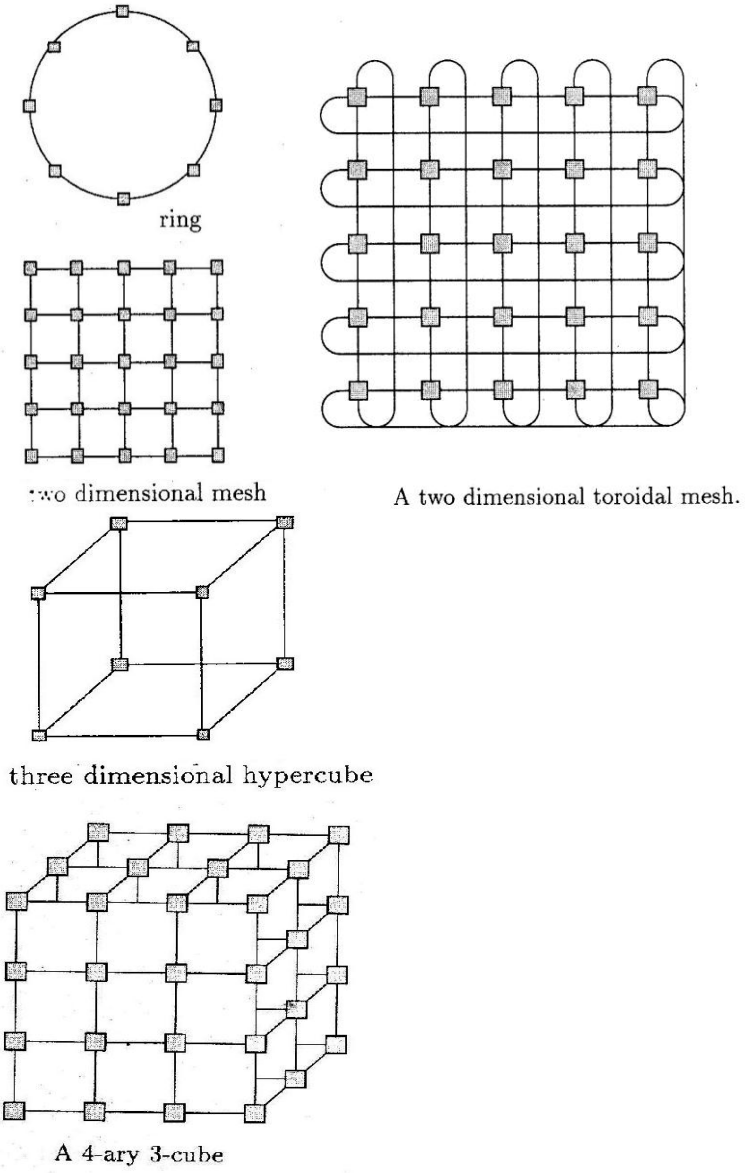
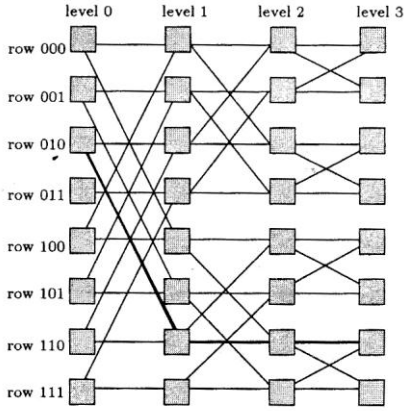


Figure 1 : examples of direct network topologies

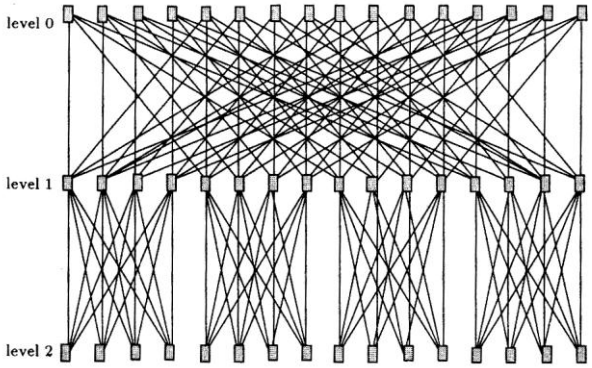
In an ***indirect*** network, PEs are not directly connected; they communicate through *intermediate switch nodes,* each of which has a limited number of neighbors. In general, more than one switch is used to establish a communication path between any pair of processing nodes. Notable examples of limited-degree indirect networks for parallel architectures are:

- *Multistage networks,*
- *Butterflies (k-ary n-fly),*
- *Trees* and *Fat Trees,*

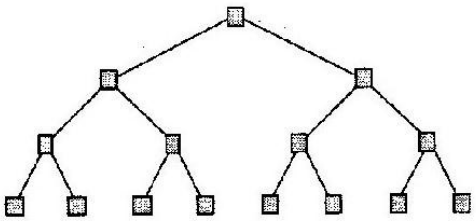
shown in the figure 2.



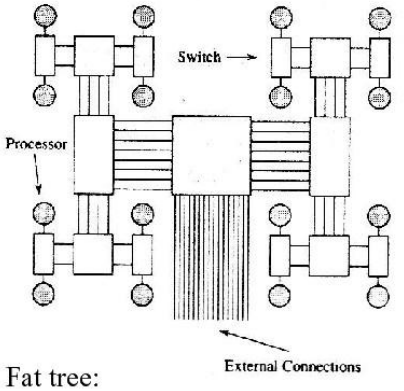
A 2-ary 3-fly



A 4-ary 2-fly



binary tree



Fat tree:
the channel capacity doubles at each level.

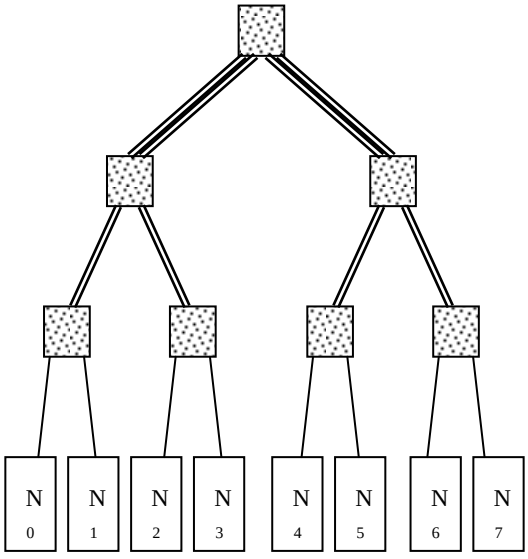


Figure 2: examples of indirect network topologies

The various topologies are characterized by the following orders of magnitude of communication latencies for firmware messages (N = number of processing nodes):

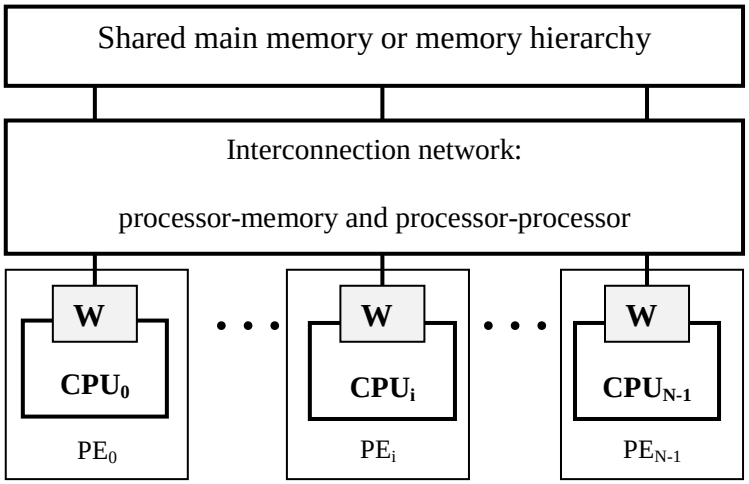
- Crossbar (not limited-degree network): *O(1)*
- Buses, Rings: *O(N)*
- Meshes, 2-dimension cubes: $O(\sqrt{N})$
- Hypercubes, multistage, butterflies, trees and fat trees: *O(log N).*

More precisely, these are the latencies evaluated in absence of contention (*base latency*).

In Section 3 we will describe and evaluate the various kinds of networks in detail.

1.4 Shared memory *vs* distributed memory architectures

In a *multiprocessor* architecture, N processors (better: CPUs) share the main memory (in general realized according to a high bandwidth organization). The following is a simplified, abstract view of a shared memory multiprocessor:



As an important alternative, in *multicore* chips *some levels of the memory hierarchy* (secondary cache, tertiary cache) can be shared among the all the CPUs (cores) or groups of CPUs.

The shared memory characteristic of multiprocessors means that *any processor is able to address any location of main memory*. That is, the result of the translation of a *logical address*, generated by any processor, can be any *physical address* of the main memory. Thus, processors are allowed

a) to physically share information (shared data structures or objects). More precisely: distinct processes can refer data which are *shared* in a primitive way, i.e. physically shared data in the same physical memory;

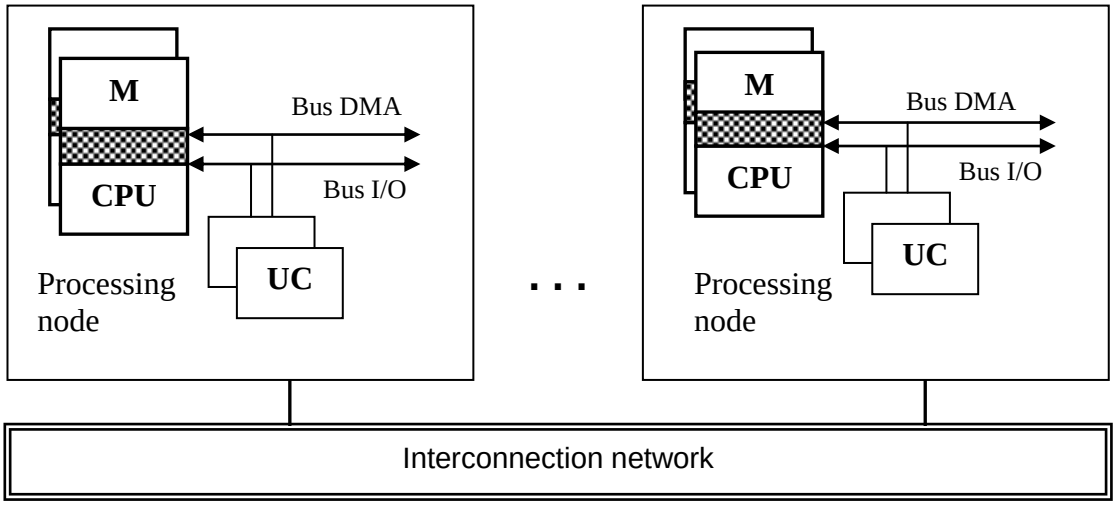
b) to have access to a common memory acting as a sort of repository for any information, including *private* information too, i.e. information (programs and data) that are private of processes and are not shared with other processes.

Notice that these characteristics are peculiar of the uniprocessor architecture too. In fact, from some respects, multiprocessors can be considered as a parallel extension of uniprocessors.

The shared memory characteristic is, at the same time, the advantage and the disadvantage of multiprocessors:

- on one hand, it is exploited (as in uniprocessors) to implement an *easy-to-design and potentially efficient run-time support* of interprocess cooperation. In fact, the run-time support is an extension of the basic solutions studied for uniprocessors in Part 0, Section 6, the extension being due to correctness (atomicity, consistency) and to efficiency reasons;
- on the other hand, shared memory is the primary source of *performance degradation*, since accesses to shared memory modules and/or shared data cause congestion. That is, formally the system can be modeled as a client-server queuing system, where the clients are processors and the servers are shared memory modules. The utilization factor of each memory module could be so high (though less than one) that, in absence of proper optimization techniques, the efficiency and scalability become relatively low despite a large number of processors is connected. Moreover, the *memory access "base "latency* (i.e. without considering congestion) is (much) higher than in a uniprocessor, because the interconnection network latency is a function of the number *N* of processing nodes.

In *multicomputers*, no memory sharing is physically possible among processes allocated on distinct processing nodes. That is, the result of the translation of a logical address, generated by any processor running on a node PE_i, *cannot* be a physical address of the main memory belonging to a distinct node PE_j. *Memory sharing is prevented at the firmware level*, though it could be emulated at a higher level. The only primitive architectural mechanism for node cooperation is the by-value communication, i.e., the cooperation via *input-output* mechanisms and interface units. Interprocess communication is implemented on top of such mechanisms:

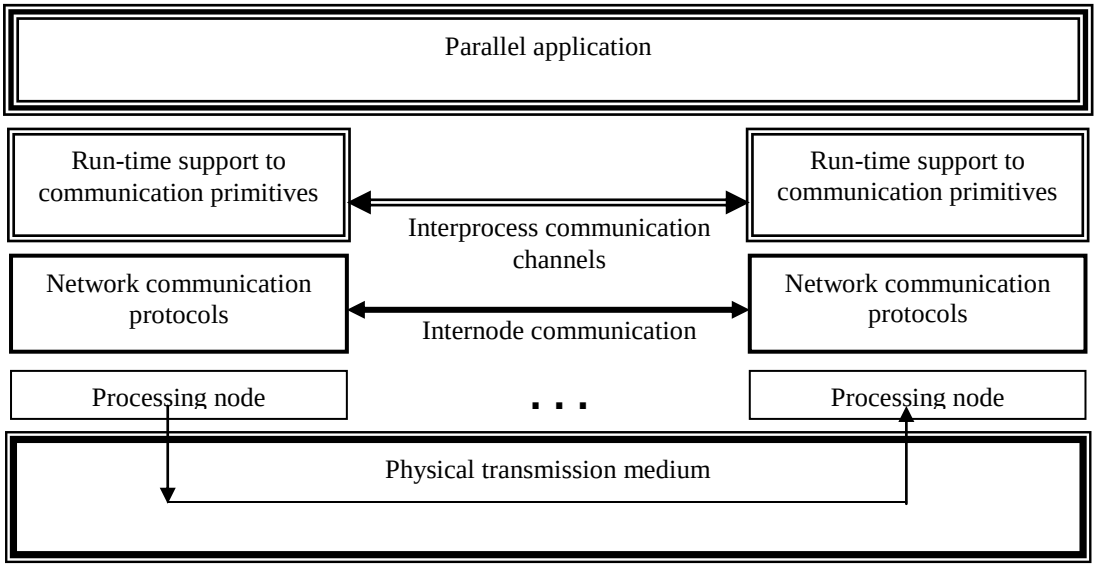


Many distributed enabling platforms belong to the multicomputer class: *massively parallel processors (MMP), network computers, clusters of PCs/workstations, multi-clusters, server farms, data centers, grids, clouds,* and so on.

Some multicomputer architectures are conceived to exploit an advanced firmware technology similar to multiprocessors (e.g. MPP) for relatively fine grain parallel applications. Others are no more than relatively inexpensive network infrastructures (e.g. network computers, simple clusters of PCs with fast Ethernet / 10Gbit Ethernet

interconnection) used for parallel computing purposes too, in general for coarse grain size computations. A very large variety of solutions exists, and is emerging, between these two extreme architectures.

The next figure applies a general, well-known concept to the multicomputer case: the process run-time support tries to exploit the characteristics of the physical underlying architecture at best. While for multiprocessors this means to exploit the physically shared memory, for multicomputers this means to exploit the protocols that are available for node communication:



In some cases, *standard communication protocols*. i.e. IP-like, are directly used, at the expense of a large overhead and performance unpredictability for parallel computations. In other cases, similarly to multiprocessors, the *primitive firmware protocol of the interconnection network* is used, with sensible improvements in bandwidth and latency of one or more orders of magnitude compared to IP-like protocols, as well as in performance predictability.

Also for multicomputers, the main peculiar characteristic (i.e. distributed memory) is an advantage and a disadvantage at the same time:

- on one hand, the absence of a shared memory leads to potentially scalable solutions,
- on the other hand, interprocess communication between processes allocated onto distinct nodes is delayed by the network latency, and possibly by heavy communication protocols, for long messages (longer than in multiprocessors, where firmware messages are used for remote memory accesses or short interprocessor communications only).

No simplistic conclusion about the performance comparison of the two MIMD classes is possible a priori, provided that also the multicomputer systems adopt the primitive firmware protocol of the interconnection network. Just two possible comparison issues are mentioned here:

- in both MIMD architectures *the interconnection network congestion problems* represent one of the main issues for performance degradation. In multiprocessors,

additional congestion is caused by memory sharing, however the shorter firmware messages have a positive effect on the exploitation of network bandwidth;

- multiprocessors can potentially benefit of by-reference communications (i.e., without physically copying the message value), while the message copy is unavoidable in multicomputers. However, the potential advantage is paid in multiprocessor because of the synchronization and cache coherence problems, which are absent in multicomputers.

2. Shared memory multiprocessors: overview

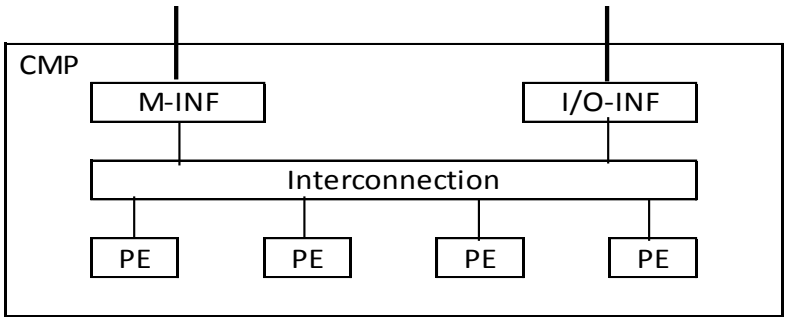
2.1 Multicore/manycore: Chip Multiprocessors

2.1.1 General architectural features

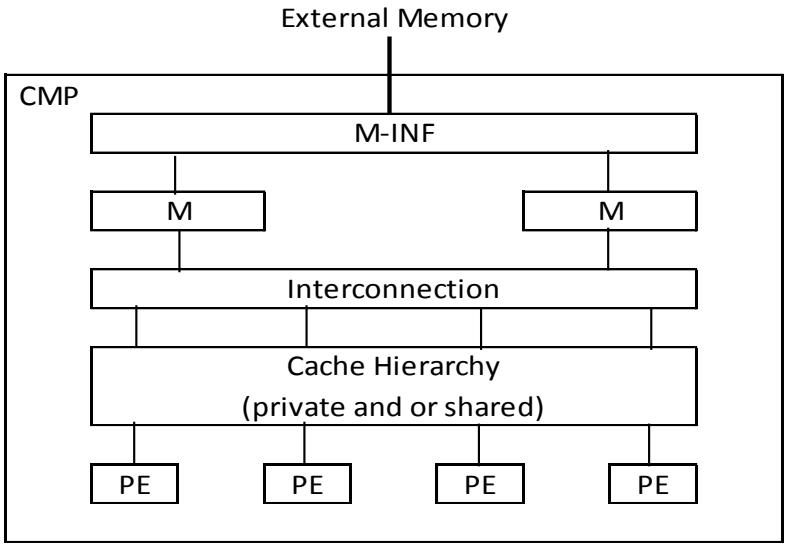
Almost all existing multicore/manycore chips have an internal shared memory multiprocessor architecture: the less vague term *Chip MultiProcessor* (CMP) is often adopted in alternative to "multicore" or "manycore".

In many cases the term CPU is used for the whole CMP, where the component internal processors (CPUs themselves) are called ***cores***. Some systems distinguish more than one multicore CPUs inside the chip. These distinctions are just a matter of commercial terminology: we will use the term *Processing Node* (PE) to denote a *core*, which is the basic unit of parallelism at the architecture level.

Often, *external CMP interfaces* (Memory Interface, I/O Interface) are common to all PEs, for example:

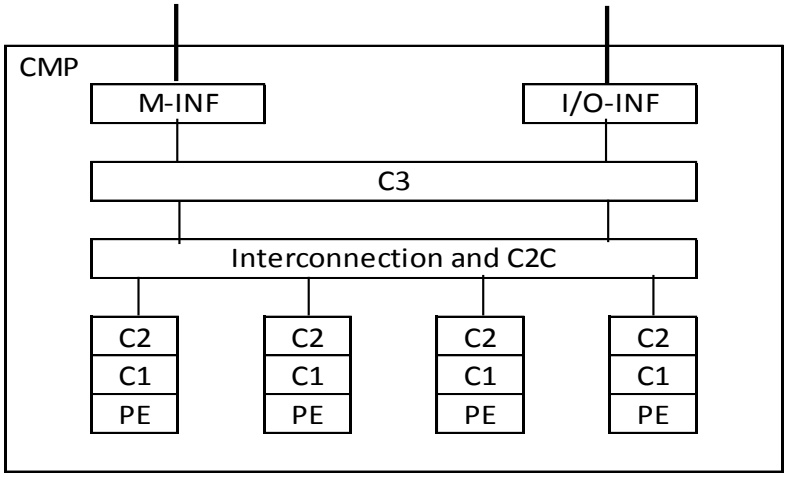


In this scheme the *main memory* is entirely external to the chip. However, some *local main memory* capacity may be present on chip. In any case, a substantial capacity of *cache memory* is provided inside the chip:



A notable feature of CMP architectures is the organization of *cache levels: primary (C1, or L1), secondary (C2, or L2),* and *tertiary cache (C3, or L3).* Each PE has a private C1. C2 may be private or shared (the trend is towards private C2). C3 is currently shared: where a

local main memory is not present on chip, in practice C3 acts as the real shared main memory:



For cache coherence implementation reasons, the interconnection can provide the possibility of *cache-to-cache* (C2C) direct copy between PE, i.e. a PE can copy a cache block from its own C1 or C2 cache into the C1 or C2 cache of another PE. Although this copy is not implemented according to a shared memory technique (i.e. the source PE does not address the destination PE cache, instead the destination PE-W is responsible of the writing operation), the global effect is equivalent to a limited form of sharing extended to all cache levels.

In some CMP products cache coherence and C2C operations are supported by a *dedicated interconnection network.*

Current *on-chip interconnection networks* are mainly

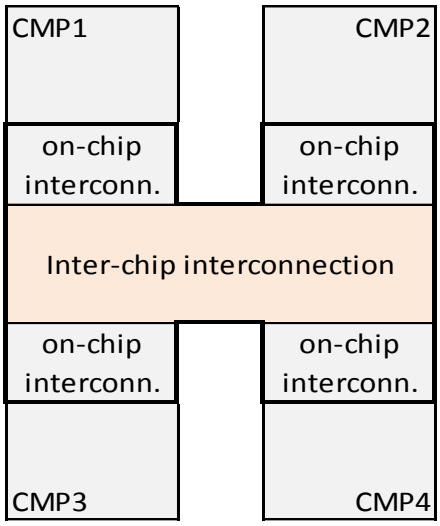
- *Crossbar*
- *Ring* and *multiple rings*
- *Mesh*

Where multiple interconnection networks are provided, they may be homogeneous or different, e.g. a crossbar for shared memory accesses and a ring for direct interprocessor communication and cache coherence supports.

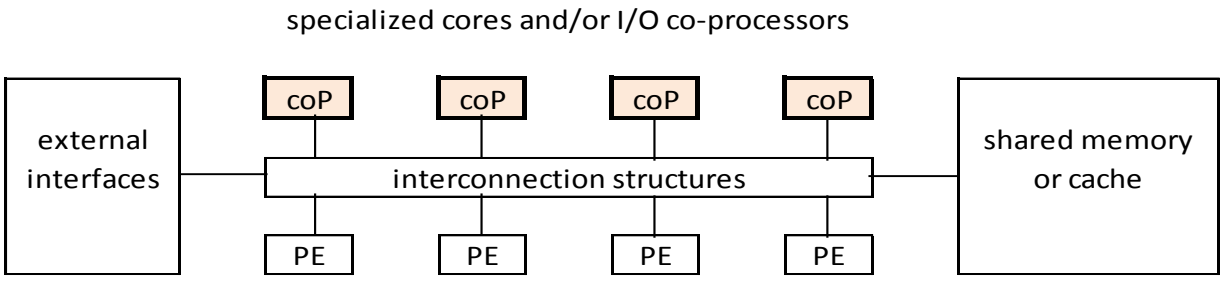
Moreover, as provided in some current products, *inter-chip interconnection structures* can be realized as extensions of the on-chip structure, thus providing a homogeneous architecture with a larger number of PEs, though on distinct chips, as shown in the next figure.

Current research on on-chip interconnection is studying the feasible integration of wired and of optical networks according to logarithmic limited degree networks (butterflies, fat trees).

Just to confirm the general knowledge of firmware communications (Part 1, Section 18.1), for an on-chip network the link transmission latency is really $T_{tr} \sim 0$, while in the best off-chip network for large non-CMP servers we have values of T_{tr} ranging from 10τ to 30τ.



An interesting trend is towards CMP including *specialized units and/or co-processors*, in addition to the general-purpose PEs:



These additional resources can be designed for run-time support facilities or for improving internal computation capabilities.

Of notable importance are the CMP architectures called ***Network Processors***, in which several networking functionalities are primitive at the assembler-firmware level, and/or are available as coprocessors, and/or are suitable for parallelization owing to suitable mechanisms and interconnection structures:

- Real-time processing of multiple data streams
- IP protocol packet switching, queuing and forwarding capabilities
- Checksum / CRC per packet
- Pattern matching per packet
- Tree searches
- Frame forwarding, filtering, alteration
- Traffic control and statistics
- QoS control
- Enhanced security
- Primitive network interfaces on-chip

2.1.2 Current CMP products

In this Section we overview some features of notable current CMP products:

1. ARM Cortex
2. AMD Opteron
3. AMD Bobcat
4. Intel Sandy Bridge
5. Intel Itanium
6. IBM Power 7
7. IBM Cell
8. IBM Wire Speed
9. Tilera Tile

In the various Sections of Part 2 these products will be used to exemplify the studied concepts and techniques.

We start with a characterization in terms of the *basic processor technology*: instruction set class and number of arithmetic Functional Units of the pipelined Execution Unit (Execution Unit bandwidth):

	Instr. Set	Instr./clock (time slot)
• ARM Cortex A9	RISC	3 Int + 1 FP
• AMD Bobcat	CISC	2 Int + 2 FP
• AMD Opteron	CISC	3 Int + 3 FP
• Intel Sandy Bridge	CISC	3 Int or 3 FP
• IBM Power7	RISC	4 Int + 5 FP
• Intel Itanium	VLIW	9 Int + 2 FP
• IBM Cell	RISC	1 Int or 1 FP
• IBM WireSpeed	RISC	2 Int
• Tilera Tile64	VLIW	2 Int

RISC processors are mainly used in the embedded, mobile, laptop and server market, CISC in addition covers the desktop market, and currently VLIW is mainly used for servers. According to Part 1, Section 21, a *VLIW* assembler machine corresponds to the VLIW-superscalar organization, in which long instructions are basically composed of RISC scalar instructions, and suitable optimizing compilers are provided to build long instructions.

The following table further summarizes the basic processor technology features in terms *in ILP (in-order vs out-of-order) and multithreading (SMT):*

AMD Opteron	CISC	Out of Order	-
Intel Sandy Bridge	CISC	Out of Order	SMT 2 way
IBM Power7	RISC	Out of Order	SMT 4 way
Intel Itanium	VLIW	In Order	SMT 2 way
ARM Cortex A9	RISC	Out of Order	-
AMD Bobcat	CISC	Out of Order	-
Tilera Tile64	VLIW	In Order	-
IBM Cell	RISC	In Order	-
IBM Wirespeed	RISC	In Order	SMT 4 way

The multicore organizations in terms of *chips per core* and *multi-chip extensions* provided directly are:

AMD Opteron	6 Processors	8 Chip
Intel Sandy Bridge	8 Processors (4+GPU)	4 Chip
IBM Power7	8 Processors	32 Chip
Intel Itanium	4 Processors	32 Chip
ARM Cortex A9	4 Processors	-
AMD Bobcat	2 Processors + GPU	-
Tilera Tile64	64 Processors	4 Chip
IBM Cell	8+1 Processors	2 Chip
IBM Wirespeed	16+4 Processors	4 Chip

Their characterization in terms of *cache hierarchy* is the following (P stands for Private, S for Shared, CC for automatic Cache Coherence; V (for Victim) and I for (Inclusive) will be discussed in Section 6):

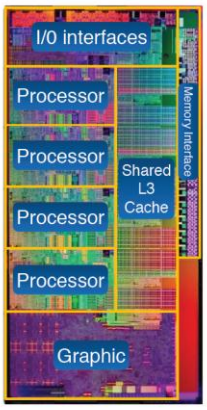
	P/Chip	L1	L2	L3	C.C.	C2C
Opteron	6	64K-P	512K-P	6M-S-V	✓	✓
Sandy Bridge	8	32K-P	256K-P	20M-S-I	✓	✓
Power7	8	32K-P	256K-P	4M-P-V	✓	✓
Itanium	4	16K-P	256K-P	6M-P-I	✓	✓
Cortex A9	4	32K-P	2M-S	-	✓	✓
Bobcat	2	32K-P	512K-P	-	✓	✓
Tile64	64	8K-P	64K-P	-	✓	(✓)
Cell	8+1	⇒	256K-P	-	-	-
Wirespeed	16+4	16K-P	2M-S(4)	-	✓	✓

The following table contains a summary of the *on-chip (NoC) interconnection structures* (the last column describes which memory unit hierarchies are connected):

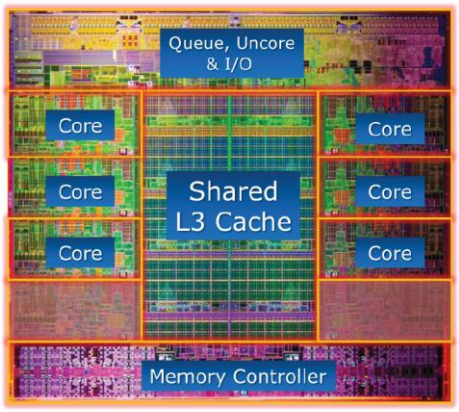
	P/Chip	Minf	NoC	Connecting
• Opteron	6	1	Crossbar	L2s - L3
• Sandy Bridge	8	1	Ring	L2s - L3
• Power7	8	2	8-Ring	L3s - Mem
• Itanium	4	2	Crossbar	L3s - Mem
• Cortex A9	4	1	Crossbar	L1s - L2
• Bobcat	2	1	Crossbar	L2s - Mem
• Tile64	64	4	Mesh	L2s - Mem
• Cell	8+1	1	4-Ring	L1s - Mem
• Wirespeed	4x4+4	4	Crossbar 4-Ring	L1s - L2 L2s - Mem

2.1.3 Two examples: current state and trends

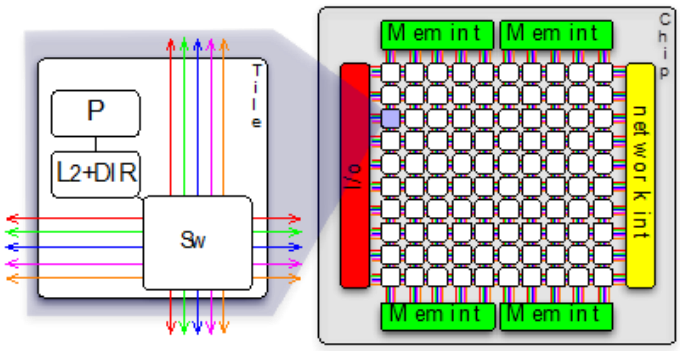
The following figures reports the chip-scheme of Sandy Bridge and of Tilera, respectively.



Sandy-Bridge (4 processors)



Sandy-Bridge-E (8 processors)



They are typical representatives of two different trends in CMP technology.

Sandy Bridge is a typical low-medium parallelism CMP (8 cores). More powerful configurations are achieved by connecting up to 4 chips and external GPU coprocessors. It is based on the classical Intel processor technology: CISC, out-of-order, medium-high bandwidth EU (17-stage pipeline) with some vectorization capabilities, 2-way SMT, PC-like cache capacities (32K for instructions + 32K for data, 8-block set associative, write-back private C1 with prefetching; 256K 8-block set associative, write-back private C2 with prefetching; 20M 4-module shared C3), graphic coprocessor, multi-ring interconnect between C2s and C3.

These features of Sandy Bridge show a typical "conservative" approach to multicore, basically consisting in an extension of old uniprocessor or dual-core chips. In particular, parallelism is low, thus core interconnect has limited capacity, and emphasis is placed on cache hierarchies to achieve a smooth transition from previous CPUs to multicore. Architectural mechanisms for synchronization and automatic cache coherence are rather powerful, though not necessarily suitable for highly parallel programs.

Tilera is a typical representative of the trend towards highly parallel CMP (64-core per chip, extendable up to 4 chips) with simple cores, a rich on-chip interconnect, and a variety of flexible mechanisms oriented to high parallelism exploitation.

Each core is VLIW, in-order, 5-stage EU with no floating point unit and no vectorization facilities, no SMT, 8K instruction direct C1 + 8K data 2-block set associative write-through C1, 64K 2-block set associative C2, no prefetching. There is no shared cache level, while there are 4 external memory interfaces. Four independent mesh networks are dedicated to specific tasks: for shared memory access, for I/O interactions, for cache coherence and C2C, and for interprocessor communication.

The forth network, called *User Dynamic Network* (UDN), supports low-latency exchange of small messages (up to 20 words) between cores (thus, between processes allocated according to the "one process per processor" mapping) through firmware queues associated to each core and values stored into processor registers.

A large degree of flexibility is offered for cache coherence: automatic management is provided, however it is possible to disable it, and even to disable the caching hierarchy itself. In the various Sections we will show the importance of new, non-conventional mechanisms for exploiting high parallelism.

2.1.4 Advanced architectural features

CMP evolution is characterized by the effective implementation of many advanced features that have been studied in the research world since several years, but that have been not yet realized in industrial products.

Some of them have been mentioned before:

- multiple interconnection networks,
- networks on-chip,
- flexible cache coherence strategies,
- specialized co-processors for application-specific and run-time-support-specific functionalities.

A further advanced feature is:

- ***smart memories*.**

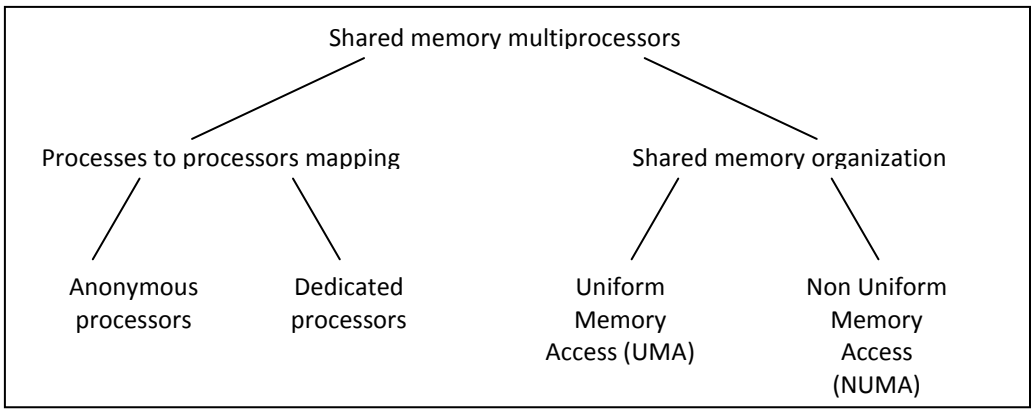
In principle, many functionalities can be delegated to the memory, in such a way that a sort of "processing in memory" is performed to help solving the processor-memory gap. Basic processor synchronization mechanisms are known examples since several years, and they could be improved substantially.

In Section 6.3 we'll see that the *Memory Interface Unit* I_M can implement efficient cache coherence Directory-based mechanisms by exploiting associative memories and other special hardware resources.

We mention here a feature able to reduce the mean access latency, an example of which is found in Tilera. In a multiprocessor architecture, we can think of *caching associated to the memory too*. We can buffer the most recently accessed blocks in a cache local to the memory module, so that some accesses, belonging to sequences characterized by locality and reuse, can be done directly with low latency.

2.2 Multiprocessor taxonomy

A useful multiprocessor taxonomy is shown in the following figure:



Processes to processors mapping refers to the strategy for allocating processes to processing nodes:

a) ***dedicated processors architecture***: process allocation is decided *statically*, once for all. That is, processes are partitioned into *N* disjoint sets, and each set is allocated to a distinct PE *at loading time*. Thus, a process can be executed only by the same PE. Explicit reallocation of processes to other nodes is possible at run-time too, notably for load balancing or fault tolerance reasons, however it is considered a relative rare event.

In this architecture both *exclusive mapping* and *multiprogrammed mapping* are possible. In the multiprogrammed case, each node has its own process Ready List, which is managed - as in a uniprocessor system - only by the local process set;

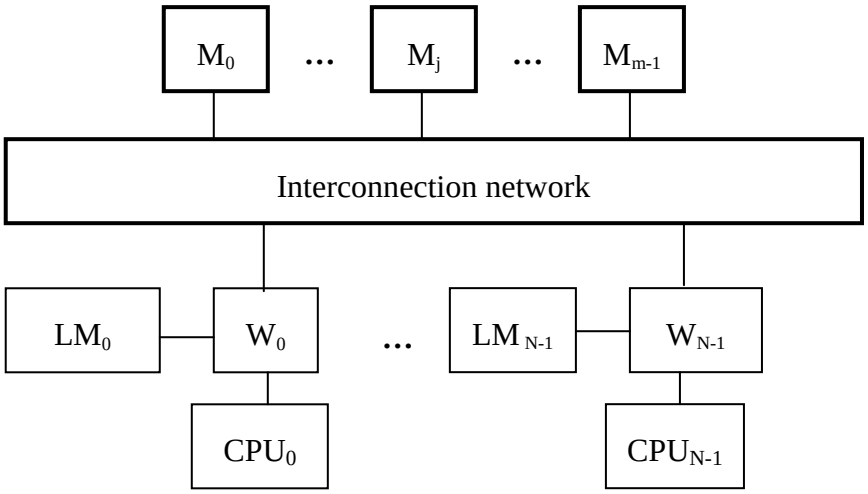
b) ***anonymous processors architecture****:* no static process allocation exists, thus any process can be executed by (i.e., it can run on) any PE. Process allocation is *dynamically* performed at run-time, i.e., it coincides with the *low-level scheduling* of processes. This architecture is meaningful for *multiprogrammed mapping* only. A unique system-wide Ready List exists, from which each PE, in the context-switch phase, "gets" the process to be executed.

Informally, the anonymous processors architecture is the natural generalization of multiprogrammed uniprocessor. Conceptually, it follows the *farm* paradigm, with the goal of achieving a good load balance of processors: PEs acts as workers, the low level scheduling is the abstract emitter functionality, and the Ready List acts as the task stream.

The dedicated processor architecture is a first step towards more distributed architectures with respect to uniprocessor machines. Conceptually, it follows the *functional partitioning with independent workers* paradigm: processors acts as workers, the task stream is thought of as decomposed into *N* distinct partitions, and the distribution functionality is statically established. As known, in principle this paradigm is prone to processors load unbalance, although this problem is dealt with at the programming level (programming methodology of Part 1) for exclusive mapping machines. If a multiprogrammed mapping approach is adopted, the load unbalance problems partially alleviated by the number of processes allocated to the same node and possibly by periodic reallocations.

In the taxonomy, *the shared memory organization* characteristic has an impact on the relative "distance" of the shared memory modules with respect to PEs, i.e., on the memory access latency:

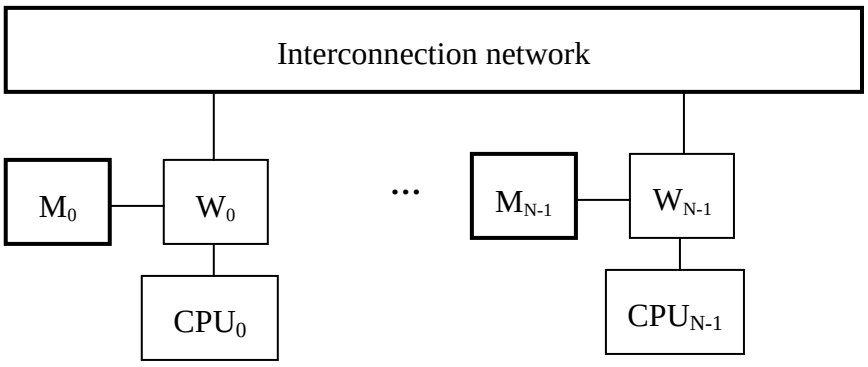
i. ***Uniform Memory Access (UMA)****, often called* ***SMP*** *(Symmetric MultiProcessor), organization*: the shared memory modules are "equidistant" from PEs. That is, the *base memory access latency* is equal for any PE-memory module pair. The following figure illustrates a typical UMA/SMP scheme, where the main memory is interleaved, and each PE has a private local memory:



As seen in Section 2.1, a variant, often adopted in CMPs, consists in *sharing the (secondary cache or) the tertiary cache* among all the PEs or groups of PEs on the same chip.

ii. ***Non Uniform Memory Access (NUMA)*** *organization:* the shared memory modules are not "equidistant" from the processing nodes. That is, the *base memory access latency* depends on the specific PE and on the specific memory module that cooperate. In a typical scheme, illustrated in the following figure, each node PE_i has a local memory M_i, and the shared memory is the *union* of all the local memories:

$$M = M_0 \cup M_1 \cup \ldots \cup M_{n-1}$$



Each CPU can address its own local memory *and* any other memory module.

The base latency for *local accesses* (CPU_i to M_i) is much lower than the *remote memory access* latency (CPU_i to M_j, with $i \neq j$), since the remote accesses utilize the interconnection network, while the local access exploit the (rich set of) dedicated links inside a PE (see Section 1.1.1).

A variant, called *COMA (Cache Only Memory Access)* consists in sharing the primary or secondary caches local to the nodes, or even in considering all the memory modules just as caches.

In the UMA organization all the shared memory accesses are remote ones, while in NUMA machines the goal is to maximize the local accesses and to minimize the remote ones.

Referring to the CMP examples of Section 2.1.3, Sandy Bridge is a classical SMP, while Tilera basically follows the NUMA approach. In Tilera, the four memory interfaces are shared by all cores, but each of them is closer to a partition of cores: i.e., shared memory is not equidistant. If the four core partitions of Tilera (16 PEs per partition) are considered equivalent CPUs, then the architecture is actually a NUMA.

In principle, all the possible combined architectures are feasible:

1. *Anonymous processors and UMA*
2. *Anonymous processors and NUMA*
3. *Dedicated processors and UMA*
4. *Dedicated processors and NUMA.*

The most "natural" combinations are 1 and 4:

- in an anonymous processors architecture with multiprogrammed mapping, shared memory acts as a repository of all, *private and shared*, information. Since a process can be executed by any processors, it is natural that any process "sees an equidistant memory" independently of the PE on which is currently allocated;
- in a dedicated processors architecture, with exclusive or multiprogrammed mapping, it is natural that the processes, allocated to a given PE, find the majority of information in the local memory of the PE itself, i.e. *all the private information and possibly some shared data*, while *only the remaining shared data* will be accessed remotely.

However, though combinations 1 and 4 are the most popular, also combinations 2 and 3 are meaningful and some notable examples exist. The reason lies in the *central role that*

memory hierarchy and caching play in multiprocessor architectures. Informally, if cache memories are allocated efficiently (i.e. if processes are characterized by high locality and reuse) the majority of accesses are performed locally in caches, thus smoothing the difference between local or remote main memory, which has impact on the block transfer latency only.

The importance of caching in multiprocessors architectures will be studied deeply in subsequent Sections.

2.3 Mapping parallel programs onto SMP and NUMA architectures

In this Section we reason about some relationships between architectural paradigms (SMP and NUMA multiprocessors) and structured parallel applications paradigms.

Let us consider a parallelization example, studied according to the methodology of Part 1.

Example specification

We wish to parallelize a sequential process P, which statically encapsulates two integer arrays, A[M] and B[M] with $M = 10^6$, and operates on a stream of integers *x*. For each *x*, P computes

c = x;

$\forall$ *i = 0 … M – 1: c = f (c, A[i], B[i])*

The obtained *c* value is sent onto the output stream.

Function *f* is a black-box of which the calculation time distribution is known: the uniform distribution in the interval (0 – 20 τ). The average interarrival time is equal to 10^5 τ.

Let the following three target architectures, based on the same processor technology, be available:

1. NUMA multiprocessor with 128 PEs, private C2 of 2 Mega words per PE, and local-shared memory of 1G words per node;
2. SMP multiprocessor with 128 PEs, private C2 of 2 Mega words per node, and equidistant shared main memory of 128G words;
3. SMP multiprocessor with 128 PEs, 16 secondary caches, each of which is 16 Mega words and is shared by a distinct group of 8 PEs, and equidistant shared main memory of 128G words.

In any configuration, each processing nodes has a primary data cache of 32K words, with block size σ = 8 words, and a communication processor. The interprocess communication parameters are assumed equal for the three architectures: $T_{setup} = 10^3\tau$, $T_{trasm} = 10^2\tau$.

Arrays A, B are statically allocated in P, thus they are statically encapsulated (replicated or partitioned) in the processes of any parallel version.

In any parallel version, the ideal parallelism degree is

$$n = \frac{T_{calc}}{T_A} = \frac{M\,T_f}{T_A} = 100$$

which is lower than the number of available processing nodes.

Two parallel versions can be identified:

a) A *farm* version with A and B fully replicated in $n = 100$ workers. Since the emitter and collector are not bottlenecks (their service time is equal to $T_{send}(1) << T_A$), the effective service time is equal to the ideal one, that is T_A. Each worker node requires a

memory capacity equal to the sequential version (about 2 Mega words, referring to data only), thus the whole memory capacity is equal to about 200 Mega words.

b) For studying a *data parallel* implementation, we observe that, since no computational property is known about function *f* (e.g., it is not known whether *f* is associative), the *M* steps, to be executed for each *x*, are linearly ordered. Thus, a notable data parallel paradigm is the *loop-unfolding pipeline*, in which *x* values enter the first stage and the intermediate stages communicate the partial *c* values. The virtual processor version consists of an array *VP[M]*, where the generic *VP[i]* encapsulates A[i] and B[i]. By mapping the virtual processor version onto the actual one with parallelism degree $n = 100$, each stage statically encapsulates a partition of A and a partition of B of size $M/n = 10^4$ integers. The ideal service time of each stage is equal to $10^5\tau$, thus the communication latency $T_{send}(1)$ is fully masked by the calculation time. The effective service time is equal to the ideal one, that is T_A. The whole memory capacity is equal to the sequential version (about 2 Mega words, referring to data only), while each node requires a memory capacity of about 20K words.

The advantages of the pipeline data-parallel solution in terms of required memory capacity are obvious. On the other hand, this solution is affected by *load unbalance problems*, due to the high variance of the calculation time.

The impact of the required memory capacity of both solutions will be studied for the three architectures.

For this purpose, let us characterize the sequential computation P from the *memory hierarchy utilization* viewpoint. For each input steam element *x*, the same values of A and B (1 Mega words each) are needed. Thus, A and B are characterized by *full reuse*, and of course locality too. We assume that the primary instruction cache is able to store all the code blocks (function *f*), thus in the following we focus on the data cache only. The *data cache working set* for the sequential abstract machine is given by all the A and B blocks (about 2 Mega words). This working set can be actually exploited only if the physical architecture is able to maintain the working set in primary cache permanently (option "not-deallocate", if available at the assembler machine level). The uniprocessor equivalent physical architecture has nodes with primary data cache of 32K words and secondary cache of 2 Mega words. Thus, *the uniprocessor equivalent physical architecture is not able to exploit the desired working set in primary data cache*. The consequence is that the reuse property is not exploited, thus

$$2M/\sigma = 256\text{K primary cache faults}$$

occur during the manipulation of each input stream element *x*. The effect on P service time is not evaluated in detail here (it depends on the organization of the block transfer and other architectural aspects), however we can say that the performance degradation is meaningful compared to the ideal situation of reuse exploitation.

1. NUMA multiprocessor with 128 PEs, private C2 of 2 Mega words per node, and local-shared memory of 1G words per node

1a) Farm solution

The situation is the same of the uniprocessor machine, replicated in every node. The primary data cache (32K words) has not sufficient capacity to store all the data working set (2 Mega words). The secondary private cache is strictly sufficient, provided that the exclusive mapping is adopted and, as usually, pre-fetching is applied at this level of memory hierarchy.

In conclusion, the farm solution will be penalized by the whole memory hierarchy inadequacy with respect to the working set requirements of the computation. The consequence is a non-negligible increase of service time with respect to the expected one. Alternatively, we can re-evaluate the sequential calculation time T_{calc}, taking into account the cache penalties; this leads to a greater value of the parallelism degree, which has to be compared to the actual number of PEs.

1b) Pipelined data parallel solution

This is a typical case in which the data parallel computation behaves much better than the sequential one ("hyper-scalability"). In fact, each node is able to maintain the respective working set partition in primary data cache permanently, passing from one stream element to the next, i.e. is able to exploit full reuse. Once loaded into the primary data cache for the first stream element, the A and B partitions (20K words) are no more deallocated. We are able to meet a very significant objective of multiprocessor architectures: the secondary caches and the shared memory itself are scarcely utilized, except for communications (which, on the other hand, are fully masked by calculations).

The load unbalance effect can mitigate the advantages of the data parallel solution. In this application, we can say that the better utilization of memory hierarchy overcomes the load unbalance effect. However, this issue would deserve an additional investigation (not discussed here for the sake of brevity).

2. SMP multiprocessor with 128 PEs, private C2 of 2 Mega words per node, and equidistant shared main memory of 128G words

Few differences exists with respect to NUMA, if exclusive mapping is adopted.

2a) Farm solution

The replicated arrays, A and B, are allocated in the shared main memory in 100 copies. They are loaded, at least in part, into the secondary caches once referred for the first time by the respective workers. The 256K faults generated by each primary data cache cannot always be served by the secondary cache alone, thus a certain fraction of block transfers from the main memory is needed. On the other hand, despite the anonymous processors characteristic, even with multiprogrammed mapping processor context-switching is a rare event in a well-designed farm program (statistically, emitter, workers and collectors are rarely blocked, owing also to the load balanced behavior). Thus, the situation is analogous to the execution on a NUMA architecture, with some additional degradation with multiprogrammed mapping.

2b) Pipelined data parallel solution

Analogously to the NUMA architecture, reuse of A and B partitions in primary data caches is fully exploited, once the A and B partitions have been acquired for the first time. With multiprogrammed mapping, the load unbalance can increase the probability of context-switching, with a consequent additional degradation with respect to the NUMA architecture, although the anonymous processor characteristic can marginally mitigate this problem.

3. SMP multiprocessor with 128 PEs, 16 secondary caches, each of which is 16 Mega words and is shared by a distinct group of 8 PEs, and equidistant shared main memory of 128G words

For this parallel program, we can say that, statistically, the memory hierarchy has the same performance of architecture 2, since the farm workers or the pipeline stages, belonging to the same group of 8 PEs, tend to share the respective secondary cache uniformly. Thus, similar considerations to architecture 2 apply to both the farm and the data parallel version.

This example shows some important ***relationships between parallel programming paradigms and parallel architectures***. Notably the following characteristics of parallel paradigms have a strong impact on the achievable performance:

- memory requirements,
- load balancing,
- predictability of sequential computations.

On the other hand, several architectural issues have to be taken into account, notably:

- interprocess communication costs,
- non-determinism in multiprogrammed process scheduling,
- memory hierarchy management, notably block transfer bandwidth and latency,
- options for primary cache management,
- strategies for secondary cache management.

2.4 Shared memory organization and caching

As said, multiprocessor performance is limited by the latency of the interconnection network and of shared memory modules. We distinguish between:

- *base latency*, i.e. without considering the effects of congestion,
- *under-load latency*, i.e. considering the effects of congestion, evaluated according to a client-server queuing model.

In order to reduce such latencies, modular memory and caching become the most critical issues. Both techniques aim to decrease the response time, since both the *server service time* (utilization factor) and the *server latency* are improved by applying these techniques.

A modular memory organization is adopted, often relying on *interleaving* to increase the bandwidth.

Caching plays a very important double role in multiprocessors:

a) as in uniprocessor machines, it contributes to minimize the *instruction service time*,

b) peculiarly of multiprocessor machines, it *contributes to reduce the memory congestion* (the memory utilization factor), as well as the *network congestion*.

About point *b)*, notice that even relatively slow caches could be accepted (any way, cache technologies are characterized by relatively low access times too).

The term ***all-cache*** architecture will be used for multiprocessors in which *any* information is transferred into the primary cache before being used. Many machines are all-cache (except Cell, Section 2.1.2), although some systems (e.g., Tilera) offer the *cache-disabling* option in order to avoid that some information are cached: this option is implemented by marking the entries of the process relocation table with a “cacheable/non-cacheable” bit.

2.4.1 High-bandwidth shared memory

The *interleaved* memory organization plays a double role in multiprocessors:

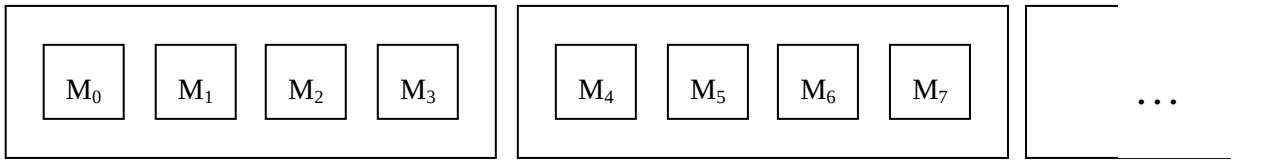
a) as in uniprocessor machines, it supports *high-bandwidth transfers of cache blocks*;

b) peculiarly of multiprocessor machines, it *contributes to reduce the memory congestion* (the memory utilization factor).

Jointly, points *a)* and *b)* motivate the following memory organization:

1) a modular memory is composed of several groups of modules, called ***macro-modules***,

2) each macro-module has an *interleaved* internal organization, for example:



Interleaved macro-module 0 *Interleaved macro-module 1* *Interleaved macro-module 2*

3) macro-modules may be organized each other:

- in an i*nterleaved* way, as in the SMP architecture (in the figure: the first location of module M_4 has physical address equal to 4), or
- in a *sequential* way as in a NUMA architecture (in the figure: the first location of module M_4 has a physical address which is equal to the last address of module M_3 plus one).

The global effect is high bandwidth for cache block transfers in any architecture (SMP, NUMA), and reduced contention on equidistant memory modules in SMP machines.

From a technological point of view, a macro-module with h modules, each with k-bit words, can be realized as just one module with *long word*, i.e. with $h*k$-bit words (similarly to superscalar/VLIW architectures). For example, a 256-bit word macro-module is equivalent to a macro-module composed of 8 interleaved modules with 32-bit words each.

Often the number of modules, or the number of words in a long word, of a macro-module coincides with the *cache block size* (cache "line").

2.4.2 Interleaved memory bandwidth and scalability upper bound

Performance evaluation of parallel programs is done by applying the cost model methodology of Part 1, where parameters T_{calc} and L_{com} are evaluated according to the characteristics of the specific multiprocessor architecture. In particular, the memory access latency and the interprocess communication latency will be studied in Sections 4 and 7 respectively.

Here we report an *asymptotic performance evaluation*, just to give an idea of the performance degradation problems in multiprocessor architectures.

This analysis is valid for a *SMP* system. It is based on the ***cost model for the offered bandwidth of an interleaved memory***, under the following assumptions:

i) *n* identical PEs;

ii) *m* interleaved memory modules. In the following, the term *memory module* will be used to denote single-word modules, or long-word modules, or interleaved macro-modules according to the specific memory organization;

iii) accesses (single words, or cache blocks in memory organizations with long-word modules or macro-modules) are done to private information only, or to shared information which do not require to be manipulated by indivisible sequences.

Under these assumptions the following statement holds: *the access of any PE to any memory module has a probability which is constant and equal to 1/m,* i.e. we have a fully uniform distribution of accesses from PEs to memory modules.

As a consequence, the *conflict probability p(k),* i.e. the probability that *k* PEs over *n* (*k = 1 ,..., n*) are trying to simultaneously access the same memory module, is distributed according to the binomial law:

$$p(k) = \binom{n}{k} \left(\frac{1}{m}\right)^{k} \left(1 - \frac{1}{m}\right)^{n-k}$$

For each module M_j *(j = 1 ,..., m),* let Z_j be a binary random variable defined as follows:

$$Z_j = \begin{cases} 0 & \text{if } M_j \text{ is idle} \\ 1 & \text{if } M_j \text{ is busy} \end{cases}$$

By definition of *p(k)*:

$$Z_j = \begin{cases} 0 & \text{with probability } p(0) \\ 1 & \text{with probability } p(k) \mid 1 \le k \le n \end{cases}$$

The mean value of this random variable is:

$$E(Z_j) = 1 - p(0) = 1 - (1 - 1/m)^n$$

The *offered bandwidth* of the interleaved memory, measured in words per access time or in cache blocks per access time, can be expressed as:

$$B_M = E\left(\sum_{j=1}^{m} Z_j\right) = m \cdot E(Z_j) = m \left\lceil 1 - \left(1 - \frac{1}{m}\right)^{n} \right\rceil$$

which is graphically shown in the next figure.

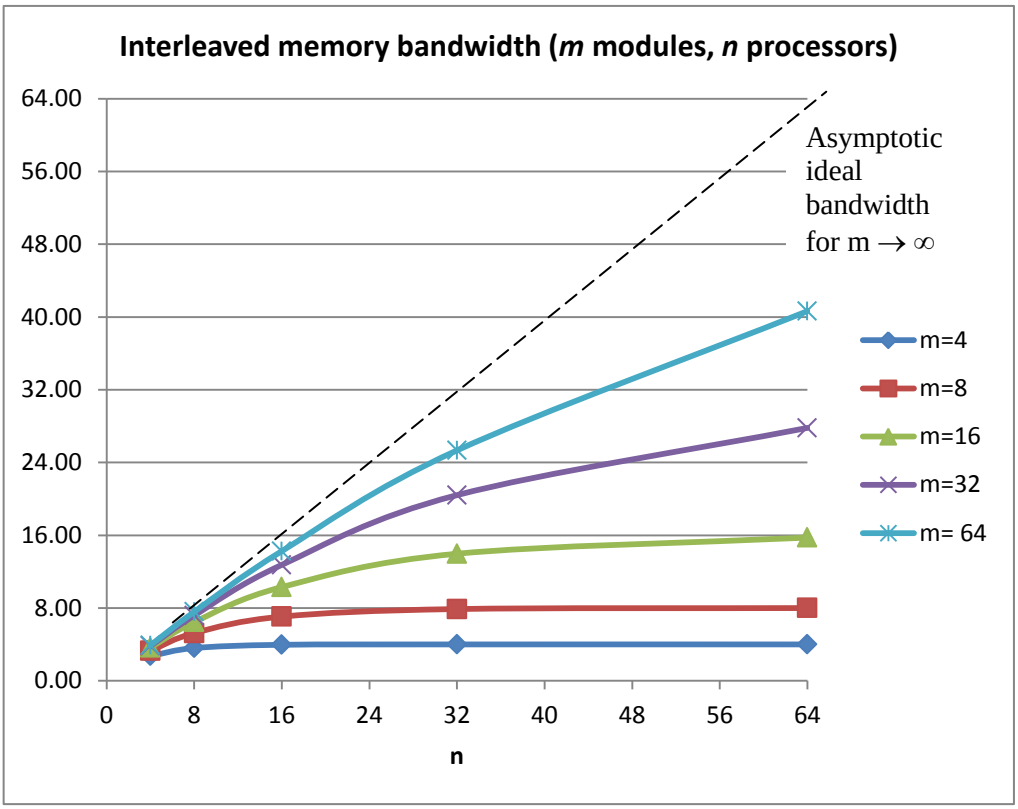
As expected, shared memory conflicts play an important, negative role in multiprocessor performance, because the memory bandwidth degradation is sensible for high values of *n*.

(Nevertheless, the absolute values of memory bandwidth are not necessarily so bad: it has to be remarked that, in long-word or macro-modules based organizations, the bandwidth values are measured in *cache blocks* per access time.)

This asymptotic evaluation can be interpreted as an upper bound of multiprocessor *scalability*:

$$s_n = m \left\lceil 1 - \left(1 - \frac{1}{m}\right)^{n} \right\rceil$$

In fact, owing to the uniform distribution of memory accesses, in a sufficiently long temporal window the average number of *active processors* (i.e. PEs not waiting for a memory reply) is statistically equal to B_M.



It is interesting to verify the asymptotic values of scalability:

$$\lim_{m\to\infty} s_n = n$$

$$\lim_{n\to\infty} s_n = m$$

That is, for finite n the ideal scalability is achieved with an infinite number of memory modules, while for finite m the scalability upper bound is given by m itself.

Though conceptually useful, this asymptotic evaluation cannot be used for a quantitative analysis of multiprocessor performance, because:

- B_M is just the *offered* bandwidth: no information is given about the *required* bandwidth, i.e. no parameter related to the application processing times is present;
- the estimated performance refers to the average number of instructions per time unit, but this does not supply any information about the real bandwidth of the parallel application. As an extreme case, it might characterize a system that executes run-time support functionalities only (i.e. with 100% overhead);
- the analysis holds for SMP architectures only, and for private information only.

2.4.3 Software lockout

The system congestion is further increased by the so-called *software-lockout* effect, that is the effect of *busy waiting* periods caused by *indivisible sequences on shared data structures*, e.g. run-time support data structures. As it will be studied in Sect. 6.1, these

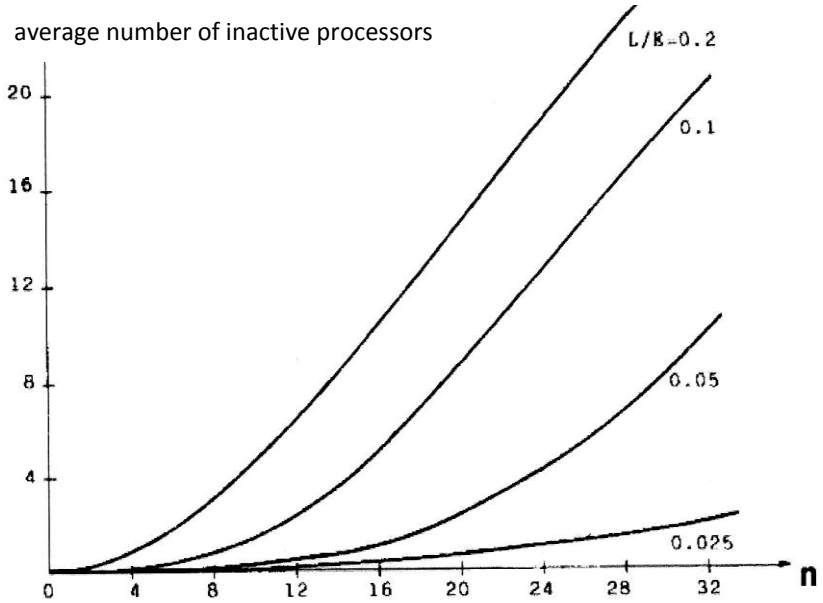
sequences are typically executed in *lock state*, i.e., mutually exclusive sections enclosed between a *lock* and an *unlock* operation. A processor, which is temporarily blocked to enter a *lock section*, is in a busy waiting state, thus it can be considered *inactive* from the point of view of the computation to be executed.

While the analysis of Section 2.4.2 refers to waiting periods in absence of indivisible sequences (i.e., sequences containing just one memory access), the software lockout analysis refers to indivisible sequences containing at least two memory accesses. A *worst-case* analysis of the software-lockout problem has been done under the assumption that the *busy waiting times are exponentially distributed*.

Let:

- L be the average time spent inside a lock section,
- E be the average time spent outside lock sections.

The average number of *inactive processors* is expressed by the following function of n and of L/E parameter:



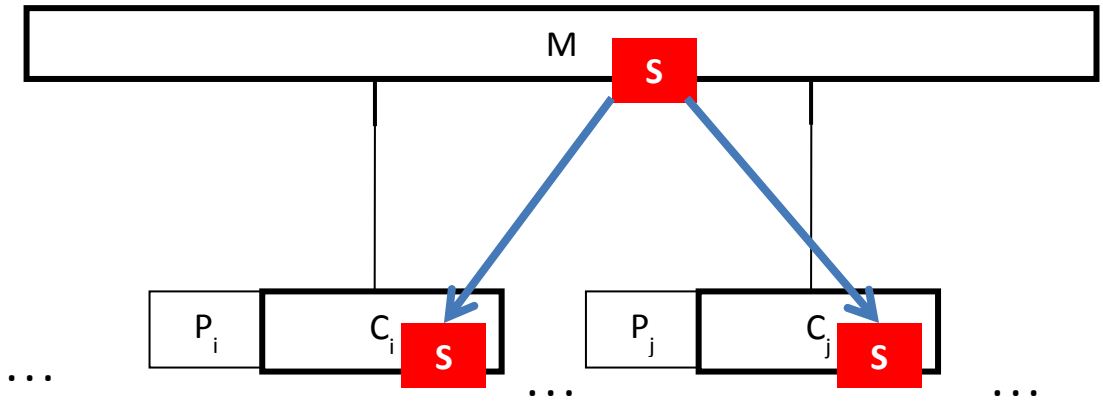
If L/E increases (i.e., L increases), the average number of inactive processors increases very rapidly. In the figure we can observe that L/E should be at most of the order of 10^{-2} for limiting the scalability degradation to acceptable values.

Moreover, for a given L/E, a threshold value n_c exists, such that for $n > n_c$ the average number of inactive processors grows linearly, i.e. in practice any additional PE over n_c is just an inactive one. For values of L/E having order of magnitude greater than 10^{-2} this threshold value is rather low. Consequently, in the run-time support design, the lock section lengths have to be *minimized*, notably by trying to decompose each lock section into a number of smaller, independent lock sections, possibly interleaved each other.

Although the above analysis is a worst case one, the software lockout effect might become a serious problem in parallel applications. Just to prove this assertion, it is worth knowing that software lockout was the main cause of the commercial failures of some industrial multiprocessor products which tried to adopt uniprocessor operating systems, like Unix. In such operating system versions, the L/E ratio is as large as 60% and over, therefore disappointing scalability values were achieved. In order to correct this conceptual and technological error, the operating system versions (Unix kernel) for multiprocessors were completely re-written adopting minimization techniques of lock sections.

2.5 Cache coherence

The potential important advantages of the all-cache feature have a counterpart in terms of increased complexity of multiprocessor design and utilization, because of the so-called *cache coherence* problem. It derives from the need to maintain *shared* data consistent in presence of caching. That is, assume that nodes PE_i and PE_j transfer the same block S from the main memory M into their respective primary caches C_i and C_j:



- if S is read-only, no consistency problem arises;
- if PE_i modifies (at least one word of) S in C_i, then the S copy in C_j becomes inconsistent (not coherent). This inconsistency cannot be solved by the existence of S copy in M, because:
    - for Write-Back caching, M is inconsistent with respect to C_i,
    - but even in a *Write-Through* system M is not "immediately" consistent with C_i (i.e., updating S in C_i and updating S in M are not atomic events, otherwise the Write-Through technique doesn't make any sense), thus it is not possible to rely on this writing technique only.

Solutions to this problem have been investigated intensively. We will distinguish between *automatic* techniques and *non-automatic* or *algorithm-dependent* techniques.

2.5.1 Automatic cache coherence

In many systems, cache coherence techniques are entirely implemented at the *firmware* level. Two main automatic techniques are used:

a) ***invalidation***: the *valid* copy of a block is one of those, usually the last one, that have been changed, invalidating all the other copies. Let C_i be a cache currently storing S. If another node PE_j wants to modify S, then PE_j invalidates the C_i copy, and, if S is not already in C_j, acquires the updated copy of S in Cj. Invalidation has the effect of *deallocating* the block from C_i. More than one cache can contain an updated copy of S, however, if a node PE_j modifies S, then only the copy in C_j becomes valid, and all the other copies are invalidated;

b) ***update***: each modification of S in Cj is communicated (multicasted) to all other caches containing S in an atomic manner. That is, more than one copy of the same block con be simultaneously allocated in as many caches, and all copies are maintained consistent.

In both cases, a proper protocol must exist to perform the required sequences of actions *atomically*, i.e. in an indivisible, time-independent way.

At first sight, the update technique appears simpler, while invalidation is potentially affected by a sort of inefficient "ping-pong" effect (two nodes, that try simultaneously to write into the same block, invalidate each other repeatedly). However, things are different in the real utilization of cache coherent systems: depending on the specific architecture, update might have a substantially higher overhead, also due to the fact that only a small fraction of nodes contain the same block. On the other hand, processor synchronization reduces the "ping-ping" effect, so invalidation is adopted in the majority of systems. Moreover, several optimizations have been proposed in order to reduce the number of invalidations.

Two main classes of architectural solutions have been developed for automatic cache coherence:

i) ***snoopy-based***, in which atomicity is achieved merely by means of a hardware centralization point, notably a single bus (*Snoopy Bus*). The bus is "snooped" by all PEs, so that each PE has always updated information about the relevant state of a block (e.g. modified or not): *snooping* and *broadcast* are primitive operations for the bus structure. This solution is clearly limited to machines with a low number of PEs, and forces the choice of the slowest kind of interconnection structure. Notice that a broadcast communication is needed both with invalidation and with update;

ii) ***directory-based***, in which atomicity is achieved by means of protocols on shared data without relying on hardware centralization points. Each block is associated a shared data structure, called the *block directory entry*, which maintains information about the set of caches that currently contain a block copy and its state. Broadcast/multicast communications are used only when strictly necessary, otherwise *point-to-point* interprocessor communications are employed. Though at the expense of a substantial overhead in terms of additional shared memory accesses, this class of solutions aims to solve the cache coherence problems also in highly parallel multiprocessors with powerful interconnection networks.

Automatic cache coherence protocols will be studied thoroughly in Section 6.

2.5.2 Non-automatic or algorithm-dependent cache coherence

Automatic cache coherence techniques, implemented at the firmware level, adopts a typical interpretation-based approach (Part 0, Section 1.1.1), thus the protocol overhead is paid for *any* access to cache memory. Optimizations, which are dependent on the computational problem to be solved, are not feasible or very rare.

On the contrary, the algorithm-dependent technique is characterized by *explicit* cache management strategies, which are specific of the computation to be executed, *without* relying on any automatic architectural support. Potentially, relying on the existence of explicit processor synchronizations and on proper data structures, it is possible to emulate cache coherence techniques in an efficient way for each computation, with the goal of reducing the number of memory accesses and cache transfers. No mechanism/structure exists that limit the architecture parallelism.

The intuitive pros and cons consist, respectively, in

- possible optimizations for the specific computation,
- increased complexity of programming.

2.5.3 Automatic *vs* non-automatic approaches: synchronization and reuse

In order to better understand relative strengths and weaknesses of automatic and non-automatic solutions, let us consider some typical situations in parallel or concurrent applications. We will distinguish shared-objects *vs* message-passing applications and, for each cooperation model, automatic *vs* non-automatic approaches.

a) Shared-objects applications

Consider an application expressed (compiled) according to a *global environment*, i.e. *shared-objects*, cooperation model. Consider two or more processes sharing an object A in mutual exclusion, e.g. each process contains a critical, or indivisible, section S of operations implying consistent reading and writing of A. Proper synchronization operations are provided to ensure mutual exclusion of S (see also Sect. 6.1): in turn, such operations exploit a shared object X (e.g. semaphore, monitor) manipulated in an indivisible manner:

```
Process Q ::
        while (...) do
        { enter_critical_section (X);
                A = F(A, ...);                    } S
          exit_critical_section (X);
        }
Process R :: similar
Process ...
```

Cache coherence has to be applied both to blocks of A and to X.

Furthermore, suppose that S is executed many times, for example is contained in a loop. For this reason, *reuse* is an important property of both X and A.

Consider the situation in which

i) the critical section S is executed by process Q, running on processor P, and
ii) no other process attempts to execute S before the second execution of S by Q on P.

a1) Shared objects applications and automatic approach

Once A and X have been transferred into the cache of P during the first execution of S, owing to assumption *ii)* an *automatic* approach is able to naturally exploit reuse: A and X are still in P cache during the second execution of S, thus automatically avoiding transfer overhead of objects A and X between the memory hierarchy levels for each iteration.

Instead, if assumption *ii)* doesn't hold (other processes execute S between the first and the second execution of Q), X and A have been automatically invalidated, thus they are not present in P cache during the second execution of S and must be acquired again through invalidation.

In other words, the automatic approach aims to perform the block transfers (through invalidation) in the strictly needed cases only, if no additional semantic information is available about the problem at hand.

a2) Shared objects applications and non-automatic approach

In a non-automatic approach, the simplest design style is to write a code that ensures cache coherency, but that does *not* ensure reuse: at the exit of primitives modifying X and at the exit of S, objects X and A are de-allocated from P cache and re-written in main shared memory.

However, more sophisticated design styles can be conceived, making exploitation of reuse possible in the automatic approach too. Some special operations can be defined which verify the consistency of X and A and, if objects have been modified by other processes, provide a sort of *invalidation by program*.

Again, the same power of the automatic approach is possible in the non-automatic case too, although at the expense of an increased complexity of programming for objects whose management is visible to the programmer. On the other hand, specific applications can have peculiar properties according to which optimizations can be recognized in a non-automatic solution with respect to the automatic case.

b) Message-passing applications

Consider now an application expressed (compiled) according to a *local environment*, i.e. *message-passing*, cooperation model. In this case, in a shared memory architecture, shared objects belong to the run-time supports of message-passing primitives only (communication channel descriptors, locking semaphores, target variables, and so on). In this case, the programming complexity disadvantage of the non-automatic approach doesn't exist, provided that the following strategy is adopted: *algorithm-dependent solutions are applied in the design of the run-time support to concurrency mechanisms*.

In a system with a clear structure by levels, this solution allows the application designer to neglect the cache coherence problems (as it must be, since the applications should be developed in an architecture-independent way), while *the only instances of the cache coherence problems are limited to the run-time support of the message-passing concurrent language in which the applications are expressed or compiled*. Thus, explicit cache coherence strategies are designed for the *very limited set of algorithms* used in the run-time support of message-passing primitives *only*. The final effect is that:

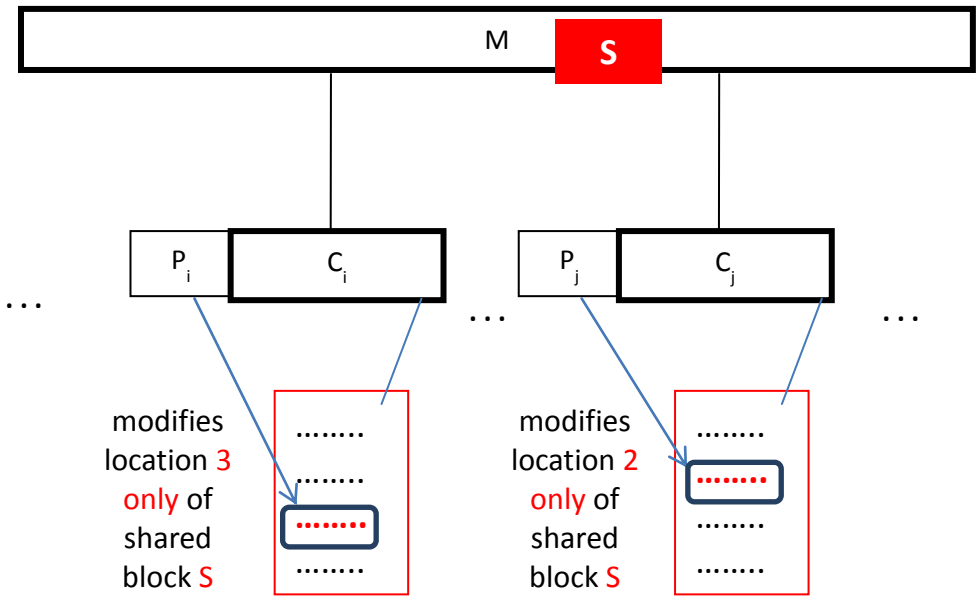
- the potential *reuse* between successive utilization of the same run-time support object is not exploited, e.g. two consecutive *send* on the same channel, not interleaved by the execution of primitives on the same channel;
- the cache coherence *overhead* is paid only when strictly needed, also taking into account that the probability of accessing a run-time object is relatively low.

Notice that *this is true also for the global environment model*, at least as far as the *synchronization primitives* are concerned (operation acting on X, in the previous example). Moreover, for such applications, some development frameworks/tools provide primitive operations to manipulate any shared object, e.g. also A in the previous example. Again, in such cases, the only instances of the cache coherence problems are limited to the *run-time support* of the concurrent language in which the applications are expressed or compiled.

The only case in which the non-automatic approach requires additional efforts to the application programmer is the following one: global environment applications in which some shared objects (e.g., A in the previous examples) are not manipulated by means of primitives of the concurrent part of the language, i.e. they are manipulated by usual mechanisms of sequential languages without any intervention of the concurrent compiler.

2.5.4 False sharing

Let us consider the situation in which two PEs share the same block *S*, but each of them modifies distinct locations of *S*, as shown in the next figure.



No real sharing of information, thus no real problem of cache coherence, exists in this case. However, it is not distinguished from a true sharing situation if *automatic* cache coherence techniques are applied, because the *cache block* is the elementary unit of consistency.

"Padding" techniques (data alignment to block size) might be applied in order to allocate words modified by distinct PEs in distinct blocks. This solution is effective when the situation is static and clearly recognizable according to the knowledge of program semantics.

The algorithm-dependent cache coherence approach deals with, and solves, this problem in a primitive manner (especially in the run-time support design), provided that instructions for manipulating single locations exist.

3. Interconnection networks

This Section deals with interconnection networks for any kind of MIMD architecture. We will see that, though few of them are more suitable for specific architectures, several networks can be used in both classes of parallel architectures (shared memory multiprocessors and distributed memory multicomputers), provided that proper mechanisms are realized in processing nodes and in networks nodes according to the specific architecture class.

The focus will be on *limited degree* networks which, with respect to traditional buses and crossbars, are suitable for scalable, highly parallel architectures. In an architecture based on limited degree networks, a PE is directly connected to a small subset of PEs and is indirectly connected to any other PE through an intermediate path of switching nodes. As a rule, these networks provide some degree of *symmetry*, for modularity, efficiency and cost-model predictability reasons.

3.1 Network topologies

For clarity and completeness of presentation, this Section contains a repetition of Section 1.3.

A first characteristic of interconnection networks consists in the following distinction:

- *direct* networks,
- *indirect* networks.

In a ***direct*** network, point to point dedicated links connect PEs in some fixed topology. Of course, messages can be routed to any PE which is not directly connected. Each *network node,* also called ***switch node*** (or simply *switch*), is connected to one and only one PE, possibly through the node interface unit (logically, W is not necessary if the network nodes play this role too). Notable examples of limited-degree direct networks for parallel architectures are:

- *Rings*,
- *Meshes* and *Tori* (toroidal meshes),
- *Cubes* (*k*-ary *n*-cubes),

shown in figure 1.

In an ***indirect*** network, PEs are not directly connected; they communicate through *intermediate switch nodes,* each of which has a limited number of neighbors. In general, more than one switch is used to establish a communication path between any pair of processing nodes. Notable examples of limited-degree indirect networks for parallel architectures are:

- *Multistage networks*,
- *Butterflies (k-ary n-fly)*,
- *Trees* and *Fat Trees*,

shown in the figure 2.

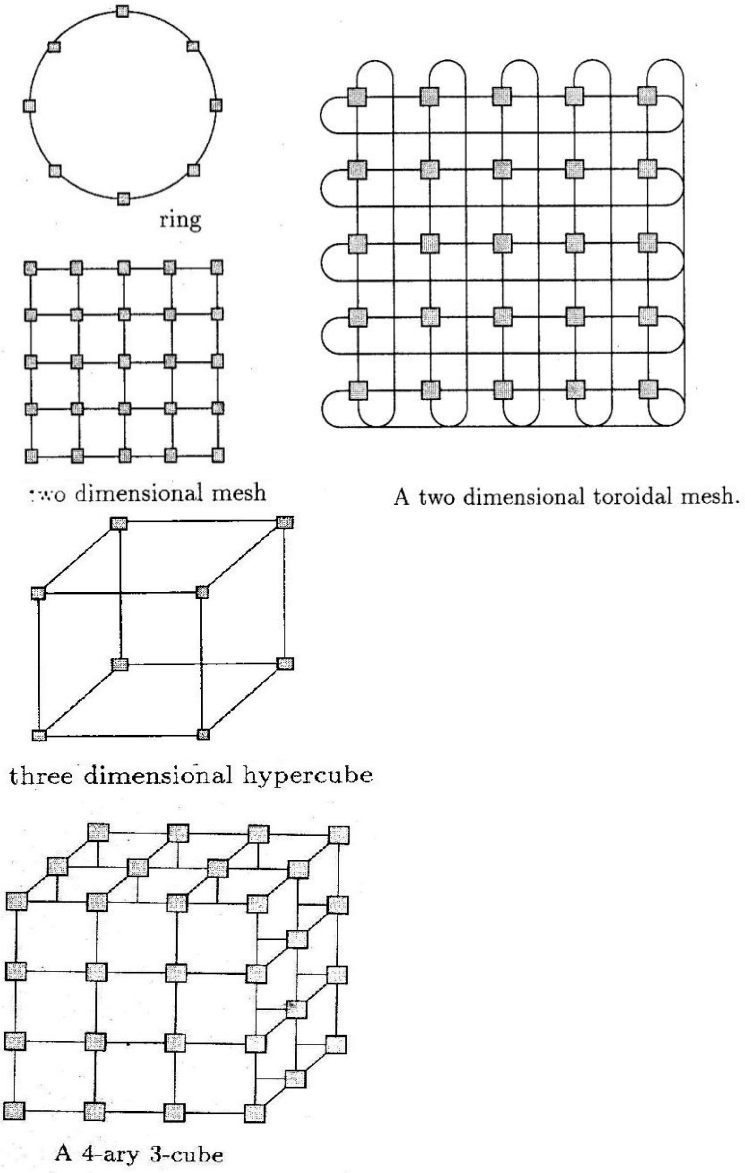
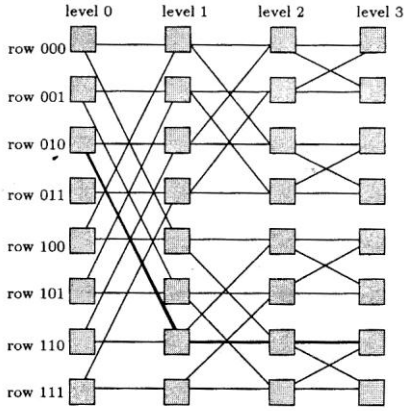


Figure 1 : examples of direct network topologies

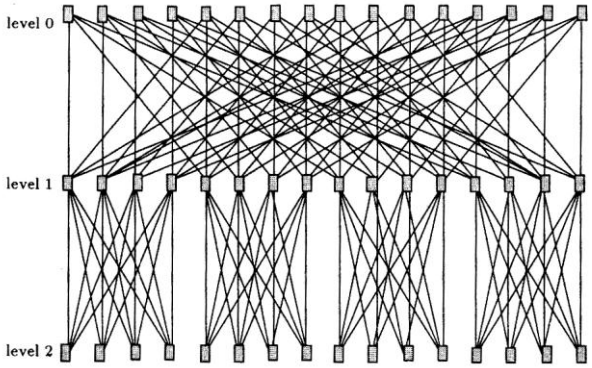
The various topologies are characterized by the following orders of magnitude of communication latencies for firmware messages (N = number of processing nodes):

- Crossbar (not limited-degree network): *O(1)*
- Buses, Rings: *O(N)*
- Meshes, 2-dimension cubes: $O(\sqrt{N})$
- Hypercubes, multistage, butterflies, trees and fat trees: *O(log N).*

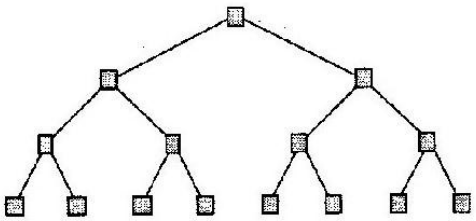
More precisely, these are the latencies evaluated in absence of contention (*base latency*).



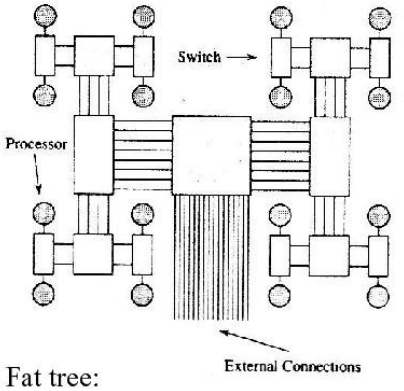
A 2-ary 3-fly



A 4-ary 2-fly



binary tree



Fat tree:
the channel capacity doubles at each level.

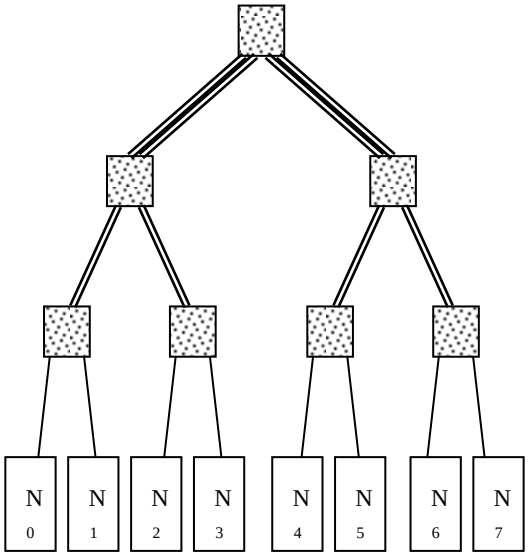


Figure 2: examples of indirect network topologies

3.2 Properties of interconnection networks

Node degree

This property is measured by the number of links incident in a network node. The definition can take into account *bidirectional* links, e.g. a pair of input and output unidirectional links are considered a single bidirectional link for the evaluation of node degree.

This metric reflects the cost of a node, which is mainly due to the *pin-count* measure, thus in a limited degree network, it should be kept as small as possible.

Distance measures

Communication between two nodes that are not directly connected must take place by *routing* mechanisms: through other network nodes in a direct network, or through intermediate switches in an indirect one. Although the actual path depends upon the routing algorithm, for the moment being we consider the *shortest path* as a topological characteristic.

The network *diameter* is defined as the maximum length of the shortest communication paths between all possible pairs of nodes.

Although the diameter is useful in comparing two topologies with identical node degree, it may be not always indicative of the actual performance. In practice, it is more important to measure the "distance" traveled by an "average" message. The *mean internode distance*, or *average distance*, is the average length of all the paths traveled by messages in the network. While the diameter depends only on the network topology, the distance also depends on the mapping of the cooperating processing modules and their communication patterns. Thus, for an arbitrary message distribution:

$$average\ distance = \sum_{i=1}^{d} i * p(i)$$

where *d* is the diameter, and *p(i)* is the probability of messages that travel a distance *i*.

We say that an application exploits *communication locality* if there is an upper bound *b* such that $p(j) = 0\ \forall\ j \geq b$. Communication locality is a desirable feature concerning the parallel application mapping. We will see that there are some network topologies which are locality sensitive, i.e. the communication latency depends heavily on the average distance, while for other notable networks the communication latency tends to increase slowly with the distance, at least for message traffics under a certain network utilization factor.

Unless specific assumptions about the parallel computation mapping are done, we assume the *uniform distribution* for *p(i),* i.e. each node send messages to any other node with equal probability. In this case:

$$average\ distance = \frac{1}{N} \sum_{i=1}^{d} i * n(i)$$

where *N* is the number of processing nodes, and *n(i)* is the number of nodes at a distance *i* from a given node.

Some interesting limited degree networks have an important feature: their average distance is *logarithmic* in the number of nodes

$$average\ distance = O(\log N)$$

Analogously to what happens in algorithm theory, a logarithmic distance is a very good result with respect to the best distance, i.e. with respect to a constant *O(1)* distance, which is typical of the too expensive non-limited degree crossbar networks.

Logarithmic networks are hypercubes (*k*-ary *n*-cubes with high *n*), multistage networks (*k*-ary *n*-flies), trees and fat trees.

Other limited degree networks, notably two dimensional meshes and *k*-ary *n*-cubes with low *n*, are characterized by a distance which, as order of magnitude, is higher compared to the logarithmic ones:

$$average\ distance = \sqrt[n]{N}$$

However, we will see that, according to the value of the multiplicative constant, some square-root (or cubic-root) networks have a better latency than some logarithmic networks of the some class (notably, low *n k*-ary *n*-cubes compared to high *n k*-ary *n*-cubes*).*

Of course, limited degree networks, such as buses and rings, are characterized by a distance which is not acceptable for highly parallel systems:

$$average\ distance = O(N)$$

Another performance metric is the *average traffic density* on a link, measured by the link offered bandwidth (the mean number of messages carried by the link in a time unit). The network offered bandwidth measures the so-called *network capacity*, which, as usually, depends on the bandwidth of the bottleneck links (i.e. the average traffic density of the bottleneck links). For example, in a binary tree the message density increases along a path from any leaf to the root under uniform traffic, for this reason the fat tree network is conceived in order to properly increase the link bandwidth as the distance increases.

Bisection width

This important property is used to evaluate the link and/or wires density and the network connectivity degree. It is also useful in estimating the area required for a VLSI implementation of the network.

Considering the network as a graph, the bisection width is defined as the minimum number of edges that must be removed to partition the original graph of *N* vertices into two subgraphs of *N/2* vertices each (if *N* is odd, two subgraphs of *(N+1)/2* and *(N-1)/2* respectively).

For example, the bisection width of a two dimensional mesh with *N* nodes is $\sqrt{N}$. For a binary *n*-dimension cube ($N = 2^n$ nodes) the bisection width is *N/2*. This shows the dense connectivity of the hypercubes compared to the meshes.

Network latency

The network latency is measured as the actual time latency needed to establish a communication through the network between the source and the destination node.

The ***base network latency*** is defined as the network latency measured *without contention.* Informally, this corresponds to the situation in which no link conflict between any pair of messages occurs. Formally, this means that *the utilization factor of each network link* (considered as a server in a client-server queuing system*) is equal to zero* ($\rho_{network} = 0$).

The base latency is not a function of message traffic. It is just a *function of architectural characteristics*:

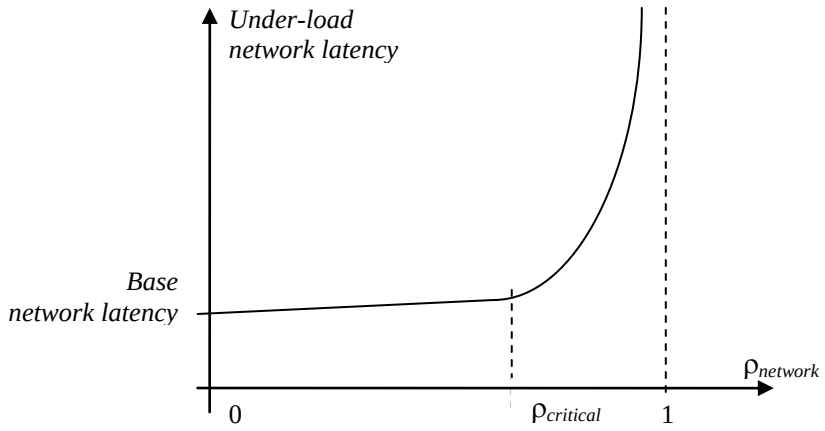
- switch delay (switching unit service time),
- wire delay (T_{tr}, wire transmission latency),
- network distance,
- message length,
- routing strategy,
- flow control strategy.

The base latency is an ideal latency evaluation, taking into account only the architectural features of a communication network.

The ***effective latency***, or ***under-load latency***, takes into account the contention, i.e. link conflict situations. Formally, we evaluate the network according to a queuing model, where each network link is a server in a client-server queuing system (Part 1, Section 15). The *network load,* or *network traffic,* is measured as the response time of the servers, thus as a function of

- network utilization factor $\rho_{network}$,
- service time t_S and interarrival time t_A distributions,
- network *base* latency itself, i.e the server latency L_S.

From a qualitative viewpoint, we have the known function shape, showing that the under-load latency is close to the base latency if the utilization factor is less than or equal to a critical threshold:

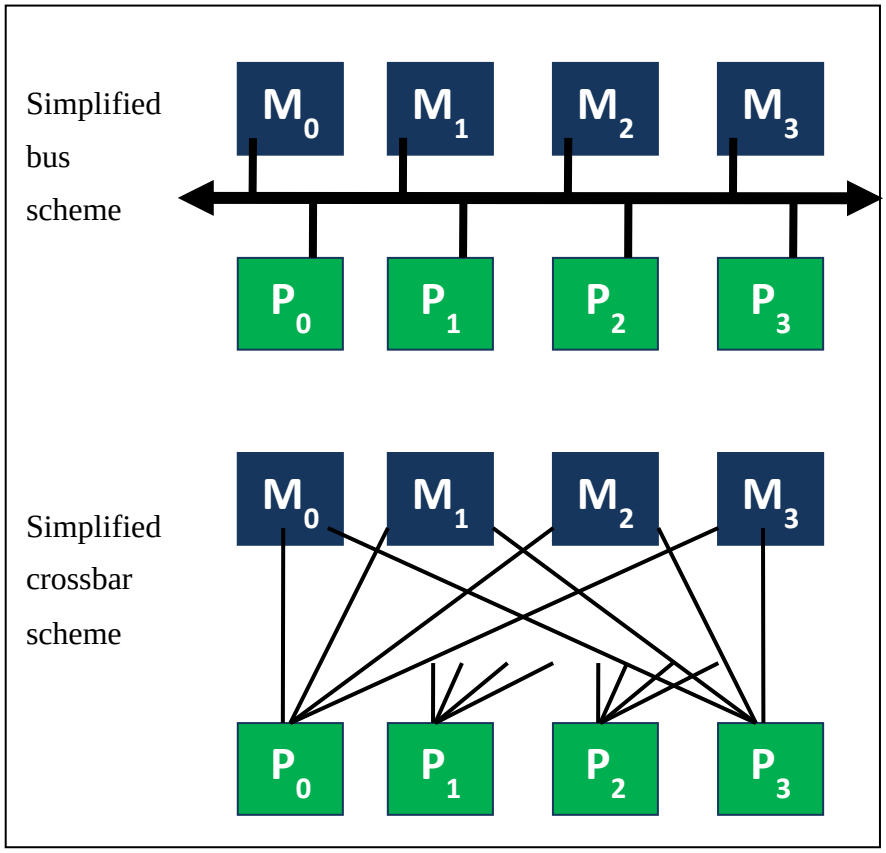


This aspect is extremely important for the effective exploitation of parallel architectures. We'll come back to discuss it at the end of the Section.

According to the parametric features of the structured parallel paradigms (Part 1), one of the main goals of parallel program mapping is achieve this optimal implementation (a first example has been discussed in Section 2.3).

3.3 Buses and crossbars

Buses and crossbars represent the two extreme solutions for interconnection networks. Their principle is illustrated in the following figure:

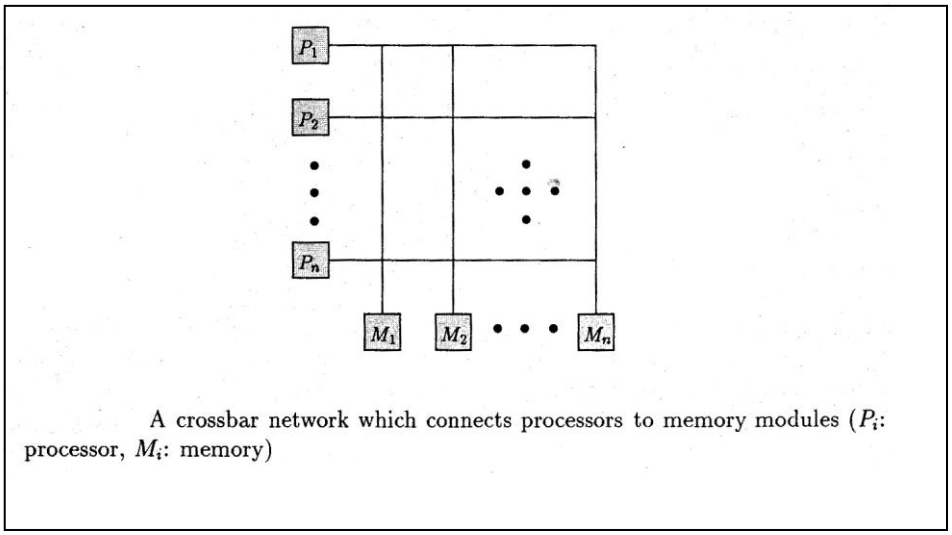


The highest-end structure is the fully interconnected crossbar, while the bus is the most elementary limited-degree network.

Buses are the most common interconnection networks for uniprocessor systems (I/O subsystem, including DMA) and for the first-generation low parallelism multiprocessors and CMPs. Though being limited degree structures (by definition), buses are not suitable for highly parallel systems, because of their linear distance and latency, their limited parallelism (at most one message can be transmitted at the time), thus limited scalability, as well as limited availability in case of failures.

The extension to *multiple buses* network (sometimes called *partial configuration* or *partial crossbar*), has been adopted in some systems to partially overcome the mentioned drawbacks.

A *crossbar*, shown in the following figure in the classic example of a shared memory SMP multiprocessor, is a fully interconnected network:



A crossbar network which connects processors to memory modules (P_i: processor, M_i: memory)

In the figure, a switching unit is placed at each of the n^2 crossing point of the horizontal and vertical lines.

A crossbar is not a limited degree network: it is of $O(N^2)$ complexity in terms of hardware resources (switches) and area, thus it can be used only in limited parallelism systems or subsystems, including CMPs with a small number of cores (see Section 2.1). In these cases, the distance and latency are constant, i.e. the best as possible.

It may be interesting to define networks built around a large number of limited degree switching units, each of which exploits an internal small crossbar structure.

Moreover, crossbar can be effectively emulated by *k*-ary *n*-fly networks and fat trees, with a soft latency degradation from *O(1)* to *O(logN)*, as studied in subsequent Sections.

3.4 Rings, multi-rings, meshes and tori

In the version with bidirectional links, a *ring* with *N* nodes is a simple direct network, with node degree equal to 2 (one bidirectional input link and one bidirectional output link per node), diameter *N/2* and average distance *N/4*.

Though not suitable for highly parallel systems and having limited availability in case of failures, rings have many interesting properties that can be exploited to implement sophisticated functionalities, such as multicast communications and cache coherence protocols. For this reason, and owing to its simplicity, it is adopted as a basis in some CMPs, notably in *multi-ring* version (Section 2.1).

Similarly, *bidimensional meshes* (node degree equal to 4), and their *toroidal* versions with wraparound connections, are adopted in several MPP machines and in some emerging CMPs (Section 2.1). As said, they have $O(\sqrt{N})$ distance and base latency.

At a first sight, it might seem that many simple parallel algorithms are mapped directly onto mesh-based architectures, for example image processing, matrix computations, numerical solutions of differential equations (convolutions), and computational geometry. However, often this correspondence is misleading: as we know according to Part 1 methodology, often the parallel paradigms for these applications need additional channels

that are not mapped directly onto mesh structures, e.g. scatter, gather, multicast, reduce, stencils.

More in general, *looking for a relationship between parallel computation structure and network structure is not meaningful*, except for some special-purpose machines. Instead, we are interested in studying limited degree network topologies with interesting properties for a large variety of parallel paradigms and their compositions. Similarly, attempts to reduce the mesh diameter by introducing additional links (e.g. a global bus, or rows and columns of busses) are not to be considered meaningful for future high performance systems, because they tend to specialize the architecture towards particular applications or algorithms.

Rings, meshes and their variants, as well as hypercubes, are particular cases of the general *k*-ary *n*-cube class, which is studied in a subsequent section.

3.5 Routing and flow control strategies

It is out of the scope of this course to give a complete treatment of routing strategies and algorithms, for which a rich specialized literature exists. In this Section, we focus on the main issues of interest about routing and flow control for highly parallel architectures.

A general consideration has to be stated first: though the basic concepts are common to any networking infrastructure, interconnection networks for parallel architectures are able to execute the routing and flow control strategies *directly at the firmware level*, i.e. according to *efficient* and *performance-predictable* primitive firmware protocols. This avoids the large overhead and performance unpredictability, which are typical of other networks, notably IP-like, whose goal is substantially different. However, some commercial networks are available in double version: with firmware-primitive protocol and with IP protocol; the latter version allows the application designer to reuse existing codes that employ IP protocols, unless a much higher and predictable performance is needed (in this case, the firmware-primitive version must be adopted).

3.5.1 Routing: path setup and path selection

The next figure shows a possible taxonomy of communication methods employed in interconnection networks.

Usually, in parallel architectures, the *path setup*, or path configuration, is determined *dynamically* on-demand. This can be done according to the classical methods of packet switching, or also message switching for short messages (e.g. for remote memory accesses), while circuit switching is rarely adopted.

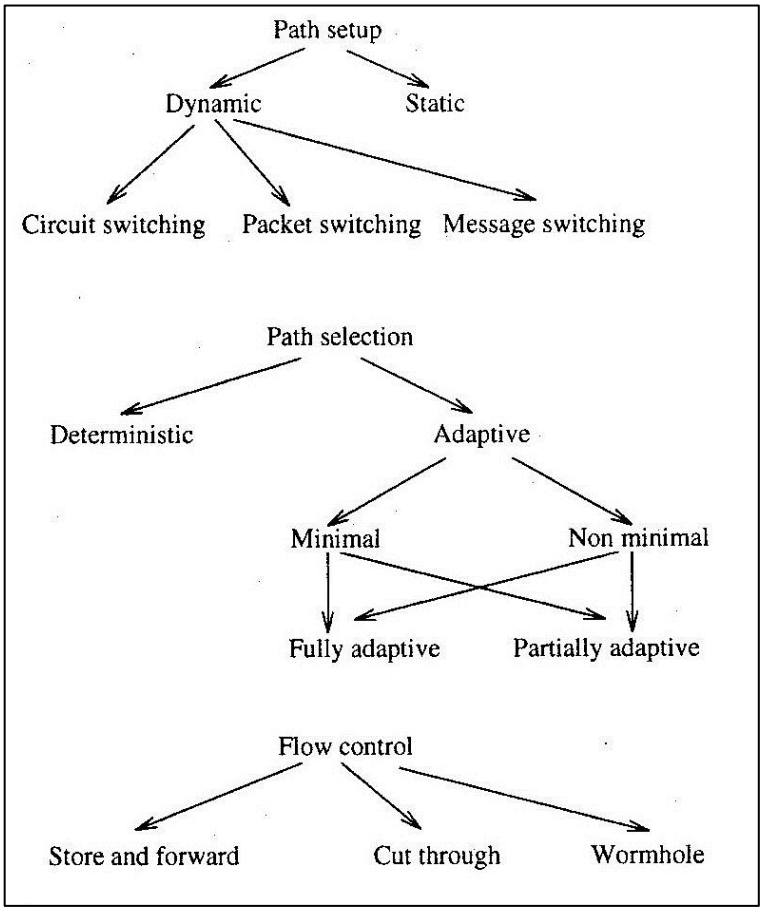
The routing *path selection* is defined in terms of a relation $\mathcal{R}$ and a function $\mathcal{F}$, defined on the set of physical links $\mathcal{L}$ and on the set of processing nodes $\mathcal{N}$:

$$\mathcal{R} \subset \mathcal{L} \times \mathcal{N} \times \mathcal{L}$$

$$\mathcal{F}: Perm(\mathcal{L}) \times \sigma \rightarrow \mathcal{L}$$

Given the current link $l \in \mathcal{L}$ and the destination node $n \in \mathcal{N}$, the routing relation $\mathcal{R}$ identifies a set of permissible links $Perm(\mathcal{L}) \subset \mathcal{L}$, that can be used in the next step on the route. $\mathcal{R}$ is a relation, because there can be several alternative paths to reach the destination. At each step of the route, the function $\mathcal{F}$ selects one link $l \in Perm(\mathcal{L})$, possibly

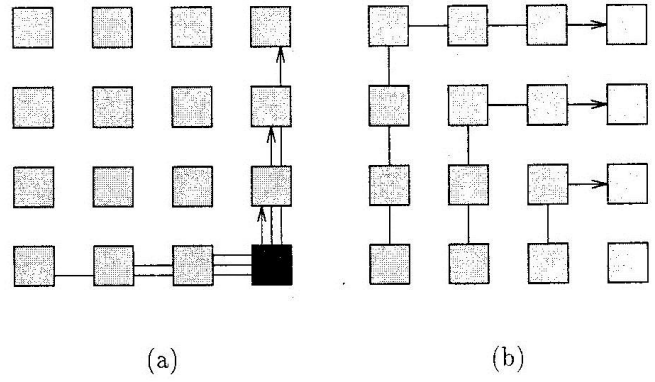
exploiting some additional information σ on the state of the network, for example current link/node load and availability or traffic.



The simplest routing approach is the *deterministic* one, in which the route is fully established by the source and destination addresses (identifiers). Though simple, this approach is unable to adapt the network to dynamic conditions, such as congestion or failures and, in general, communication patterns.

Adaptive routing, which has been a notable area in ICT research and technology, aims to provide network performance that is less sensitive to the communication pattern.

As shown in the simple example of the following figure,



(a) A set of routing paths created by a deterministic routing algorithm in a two dimensional mesh. (b) The same pattern routed adaptively to create disjoint paths.

in a deterministic routing strategy several message can cross the same "hot" node (in black), even if there are some alternative paths to handle the communication pattern. An adaptive routing strategy can route distinct messages along distinct paths.

Often, simple and efficient firmware mechanisms are sufficient to collect information on the state of the network in an adaptive routing strategy, notably the "ack" signals of the output interfaces in a switch node, possibly associated to firmware time-outs and history information. Moreover, the parallel paradigm constraints could be able to supply useful information about the communication pattern load.

A *minimal* adaptive routing limits the path selection to the *shortest* paths between any given pair (source, destination). With a *non-minimal* routing algorithm, the selected path may not always be a shortest path.

Adaptive routing is also able to better support *deadlock avoidance* strategies (not studied here).

3.5.2 Flow control and wormhole routing

In general, the term *flow control* denotes the set of techniques to manage the network resources, notably links, switch nodes, and buffering capacity.

In a ***store and forward*** strategy, at each intermediate node a packet (or a single-packet message) must be *entirely buffered* before being forwarded onto the proper output link, if and when available.

In any flow control strategy, the packet is the *unit of routing*, i.e. all the words belonging to the same packet must travel exactly the same path. In a *packet switching* path setup strategy, distinct packets of the same message can be routed independently onto distinct paths, thus exploiting potential parallelism at the packet grain size level.

In interconnection networks for parallel architectures, an additional source of parallelism can be efficiently exploited. In the so-called ***wormhole*** flow control, each single packet is further decomposed into smaller units, called *flits*, for example one words or few bytes, often coinciding with the link width. All the flits of the same packet follow the same route (as said above, the packet is the unit of routing), however flits are considered as stream elements in a *pipelined transmission*. That is, all the flits of the same packet are not buffered before being routed (as in store and forward), instead the *buffering unit* is the just the flit: as soon as a flit is received, it is forwarded onto the output link selected by the packet routing strategy, and the next flit is immediately used, or the flit waits in the network/switching node.

In this way, we are able to achieve the typical completion time benefits of a pipeline implementation compared to the sequential implementation (store and forward), *provided that the switching node bandwidth per flit is high enough*, as it holds in parallel architectures in which the routing and flow control strategies are implemented at the very primitive firmware level (one clock cycle). On the contrary, in a typical IP network, the routing and flow control service time could be too high to be able to exploit a wormhole strategy effectively.

As a rule, the *first* flit of a packet is the packet header containing the *routing information*, notably (source identifier, destination identifier) and packet length. Exploiting such information, the switching node executes the routing strategy and establishes the output link, so that all the other flits of the same packet will be forwarded onto *this* link, when

available. In case of a temporary block of the output link (e.g. the ack is delayed), the packet buffering is distributed backward in the preceding network units along the path.

In the ***virtual cut through*** variant, the pipeline transmission is equally applied, but, in case of a blocking situation, all the "worm" flits are received by, and buffered into, the blocked switching node, as in the store and forward flow control.

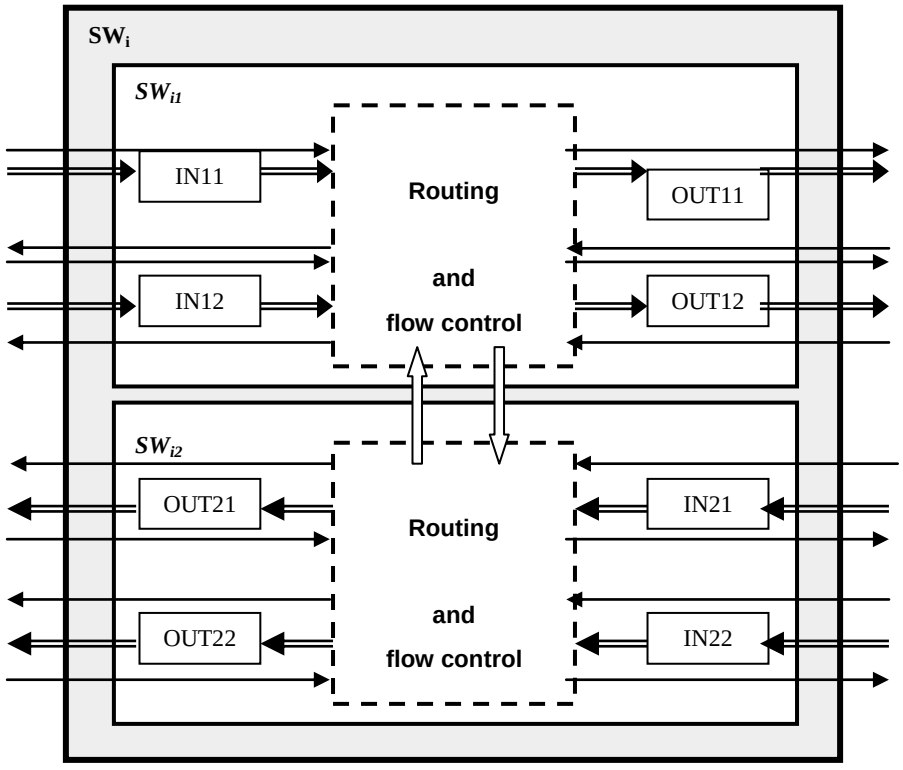
In addition to the latency reduction, wormhole is of special interest in the implementation of *networks on chip* (NoC), owing to the minimization of the buffering area. In fact, many current CMPs adopts wormhole flow control (Section 2.1), and this is a clear technological trend.

3.5.3 Switching nodes for wormhole networks

The structure of a wormhole network switching node, with node degree equal to four, is illustrated in the following figure:

The behavior is characterized as follows:

- all communications with the neighbor switching nodes are performed by asynchronous, level-transition interfaces with standard rdy-ack synchronization;



- each switching node SW_i is able to execute several unidirectional communications in parallel, for example by decomposing the node into two independent sub-units, SW_{i1} and SW_{i2}, one for each network direction;
- each sub-unit is able to execute the following actions *in a single clock cycle*:
  1. according to the header flit information of both the input messages (if both are present), determines their respective output interfaces: if they are distinct, both flits are routed in parallel, otherwise one is selected and the other header flit waits;

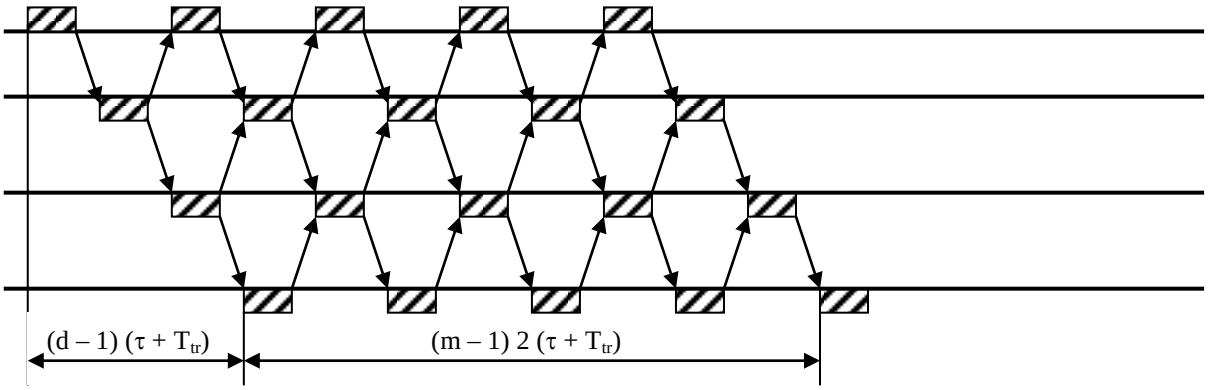
2. for each routed message, a loop controlled by the message length is executed, during which the subsequent flits are sent onto the already selected interface;
3. go to step 1, taking into account that, if one of the message transmission is still on-going, the other input interface is continuously tested too, in order to verify the possible presence of a new message. If so, this message can be routed immediately, provided that it uses the free interface, otherwise the header flit waits.

3.5.4 Communication latency of structures with pipelined flow control

In many cases, parallel architectures exploit forms of pipelined system-wide communication. For example, streams of data blocks read in parallel from a high bandwidth memory (interleaved macro-module, or long-word module), traveling through PE interface units, and, of course, through wormhole-based switches.

In other words, the pipelined transmission is applied to paths that include, but are not limited to, wormhole network sections: in this way, the bandwidths of the processing and of the switching units are balanced through the application of a fine grain pipeline paradigm.

Let any unit along a pipelined path have a service time equal to 1τ and any link a transmission latency equal to T_{tr}. The typical scheme of a pipeline, in which the first stage generates the stream and the final one collects the stream elements, is shown in the following figure:



Let

- ***d*** be the whole number of pipelined units
- ***m*** be the stream length

(in the example, $d = 4$, $m = 5$).

The latency is given by:

$$\Omega = (m - 1)\, 2\, (\tau + T_{tr}) + (d - 1)\, (\tau + T_{tr}) = (2m + d - 3)\, (\tau + T_{tr})$$

The final clock cycle is spent in processing actions, thus has not been counted in the latency expression.

This formula can be easily generalized in the case of different clock cycles or service times greater than a single clock cycle.

Let:

$$t_{hop} = \tau + T_{tr}$$

called ***hop latency***, i.e., the latency of a unit + link subsystem, which typically is equal to 1τ or 2τ in CMPs with NoCs, while could be some $10^1\tau$ in inter-chip networks (Section 2.1).

Thus:

$$\Omega = (2m + d - 3)\, t_{hop}$$

This latency can be substantially improved by using different interfacing mechanisms for two communicating units. Notably, in a CMP with NoC the acknowledgement could be implicit if a *time-dependent* physical solution is adopted. In this case the latency that in the previous figure is evaluated as

$$2(\tau + T_{tr})$$

reduces to

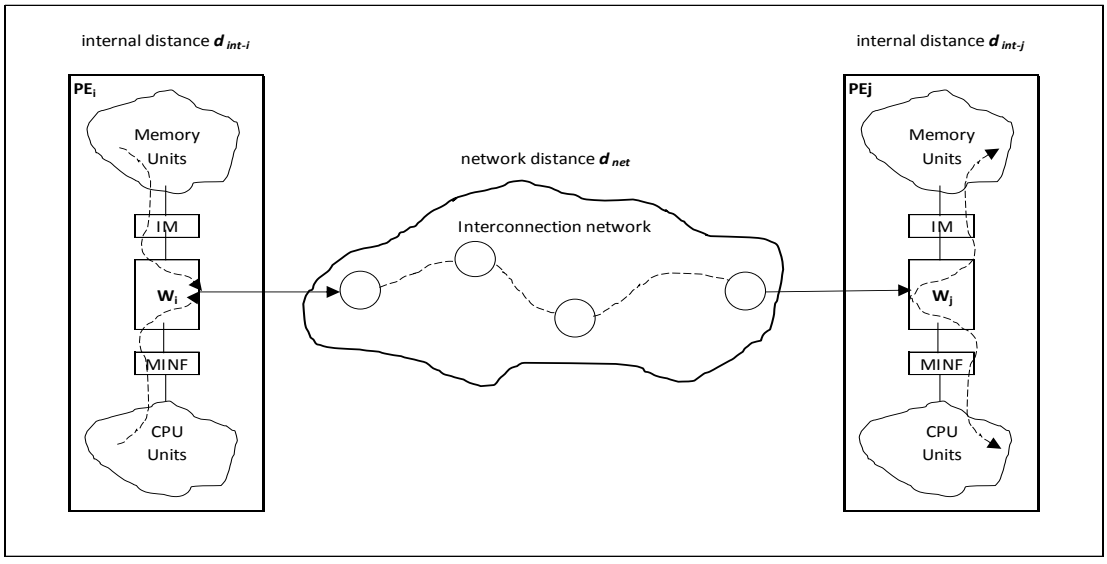
$$\tau + T_{tr}$$

Equivalently, the same result is achieved with a more general *time-independent* solution based on *double buffering* (Part 1, Section 18.2). The pipelined communication latency becomes

$$\Omega = (m + d - 2)\, t_{hop}$$

In the following, we will use this formula, which is valid also for interchip communication, provided that double buffering is exploited if time-dependent solutions are not reliable enough.

This formula can be written in a slightly different way, which is useful for our purposes. Consider a general scheme of the communication path between two PEs:



A firmware message crosses some pipelined nodes internal to source PE and destination PE, and some switches of the wormhole interconnection network. Let the internal distance d_{int} and the (average) network distance d_{net} measured in hops. Thus:

$$\Omega = (m + d_{int\text{-}source} + d_{net} + d_{int\text{-}dest} - 2)\, t_{hop}$$

In other Sections we'll evaluate the latency of some kinds of firmware messages in a multiprocessor architecture, notably request and reply messages for reading/writing accesses to shared memory

Some units inside source and destination PE (which could be a shared memory unit) take part to the pipelined communication: such units are counted in $d_{int\text{-}source}$ and $d_{int\text{-}dest}$.

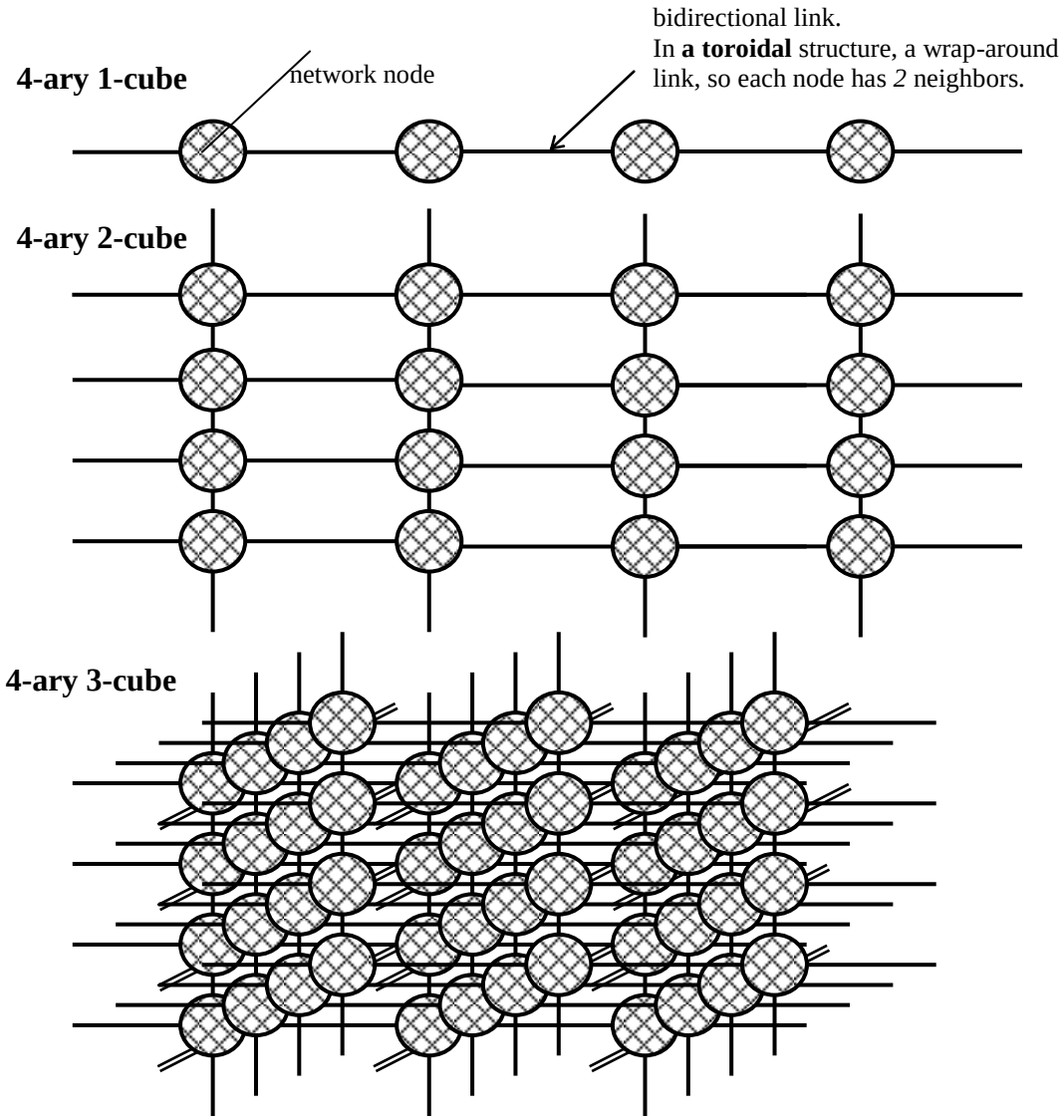
A firmware message, received by destination PE, may fire some actions which are not executed in pipeline. In evaluating such actions, their latency will be simply added to Ω.

3.6 *k*-ary *n*-cube networks

3.6.1 Characteristics and routing

This class generalizes rings, meshes, tori and hypercubes. *k*-ary *n*-cube networks are especially suitable for NUMA multiprocessors and for multicomputers, as well as SIMD coprocessors and other special-purpose machines.

The following figure shows how higher dimension cubes can be recursively built starting from lower dimension ones with the same *arity k*:



The *number of nodes* is given by:

$$N = k^n$$

where *n* is the cube *dimension*. The node degree is *2n*. For symmetry reasons, *toroidal* structures with wrap-around connections are often provided, so that no boundary nodes exist.

A *deterministic routing* strategy exploits one of the possible shortest paths between source and destination nodes. The message header includes the routing information: source and destination node identifiers expressed as *coordinates* in the *n*-dimension space:

message header = (source node identifier, destination node identifier, message length, message type)

The simple deterministic routing function is of *dimensional* kind. For example, for $n = 2$, let (Xs, Ys) be the source node coordinates and (Xd, Yd) the destination node coordinates. The *first* dimension is followed until the node with coordinates (Xd, Ys) is reached, then the *second* dimension is followed until the destination node (Xd, Yd) is reached. This algorithm can be easily generalized to any *n*.

Other *non-deterministic* routing algorithms can be the *adaptive* ones. The minimal adaptive algorithm is still a dimensional one, consisting in selecting *one of* the paths belonging to the shortest paths set.

3.6.2 Cost model: base latency under physical constraints

From the cost model viewpoint, it is useful to distinguish between ***low dimension*** cubes ($n = 2, 3$ dimension meshes) and ***high dimension*** cubes, typically the *binary hypercube* with $k = 2$. For example, a system with $N = 256$ PEs can be implemented as a 2-ary 8-cube (8-dimension binary hypercube), or as 16-ary 2-cube (2-dimension mesh with 16 nodes per side).

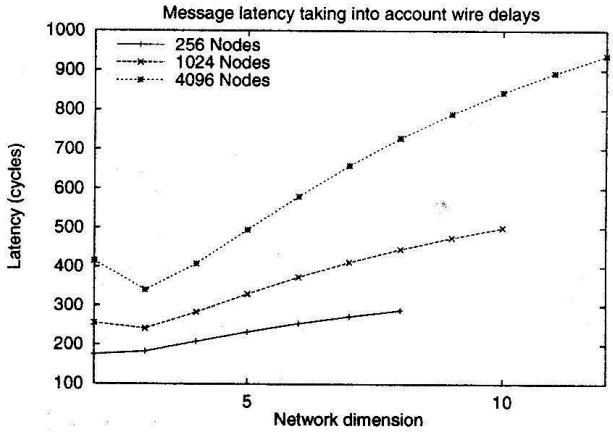
The node distance is of order $O(k * n)$. For a given number of nodes *N*,

- for relatively low dimension networks, *k* dominates, thus the average distance is $O(\sqrt[n]{N})$,
- for relatively high dimension networks, *n* dominates, thus the average distance is $O(log_k N)$.

Despite the different orders of magnitude, the comparison between the high dimension solution (e.g. 2-ary 8-cube) and the low dimension one (e.g. 8-ary 2-cube) is not so obvious, since the multiplicative constant value in the latency expression plays a very important role.

This evaluation has been studied by Dally and Agarwal for networks with *wormhole* flow control, under physical constraints. The next figure shows the ***base latency*** as a function of the network dimension *n*, considering the wire delay (T_{tr}, wire transmission latency) normalized with respect to the wire delay of a two dimensional cube, assuming a linear model for the wire delay. In this figure, constraints about the bisection width and the pin count are not considered. This is a quite optimistic evaluation for the high dimensional cubes, for which an unbounded number of network node connections have been assumed. This figure focuses on the *limits of signal propagation* alone. This physical limitation tends to favor low dimensional networks ($n = 2, 3$). For example, for a 4096 node system, the

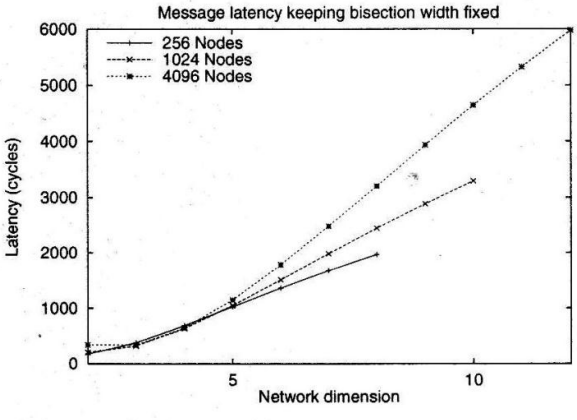
binary hypercube has a base latency three time higher than the corresponding three dimensional cubes.



Latency for machines with 256, 1024, and 4096 nodes, varying the network dimension. The switch delay s is set to 4.0 cycles, the message length L and the channel width W are set to 160 and 8 bits.

Embedding a logical topology into the physical word poses the problem of *wire density.* As seen, a measure of wire density is the *bisection width*. The bisection width for a torus connected *k*-ary *n*-cube is equal to $\frac{2WN}{k}$, where *W* is the link width in bits. For example, the bisection width is *2W* in a ring, *2Wk* in a mesh, and *WN* in a binary hypercube.

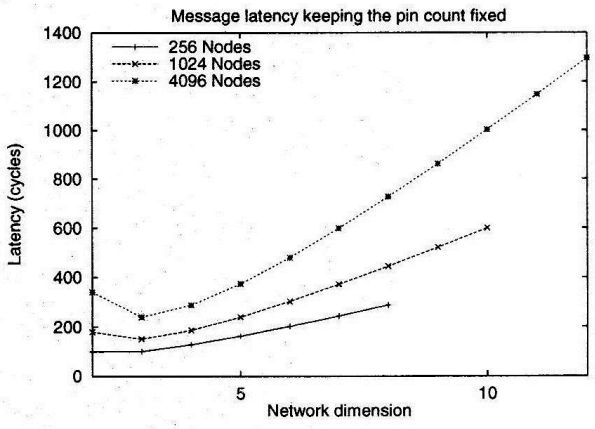
A meaningful evaluation is given by keeping the bisection width constant, *normalizing its value to N*, which is equal to the bisection width of a binary hypercube with serial links. The result is shown in the following figure:



Latency for machines with 256, 1024, and 4096 nodes, varying the network dimension and keeping the bisection width fixed to N. The switch delay s is set to 4.0 cycles, the message length L to 160 bits.

The fixed bisection width constraints is often considered too restrictive, because it indicates a cost factor more than a physical limitation.

A more realistic bound is provided by the *pin count* of each chip. In this case, we can keep constant the number of pins to 128 binary wires per node (not counting rdy-ack and ground lines). The impact on the base network latency is shown in the following figure:



Latency for machines with 256, 1024, and 4096 nodes, varying the network dimension and keeping the pin count fixed to 128. The switch delay s is set to 4.0 cycles, the message length L to 160 bits.

Once the pin count is fixed, the link parallelism (number of bits per link) of low dimensional cubes is clearly much greater than in high dimensional cubes, which, in practice, work with serial links. For this reason, the multiplicative constant in the latency expression is such that the square (or cubic) roof function is characterized by lower values than the logarithmic one.

In conclusion, the low dimensional cubes, which are able *to exploit much wider links with the same link and chip cost*, outperform high dimensional ones.

After the pioneering and fundamental experience of Hillis at MIT with the massively parallel SIMD Connection Machine, high dimensional cubes have not been adopted in parallel machines, even with large number of nodes.

These evaluation give reasons for the adoption of low dimensional cubes in the most recent CMPs, also owing to the high regularity of mesh structures.

3.7 *k*-ary *n*-fly networks

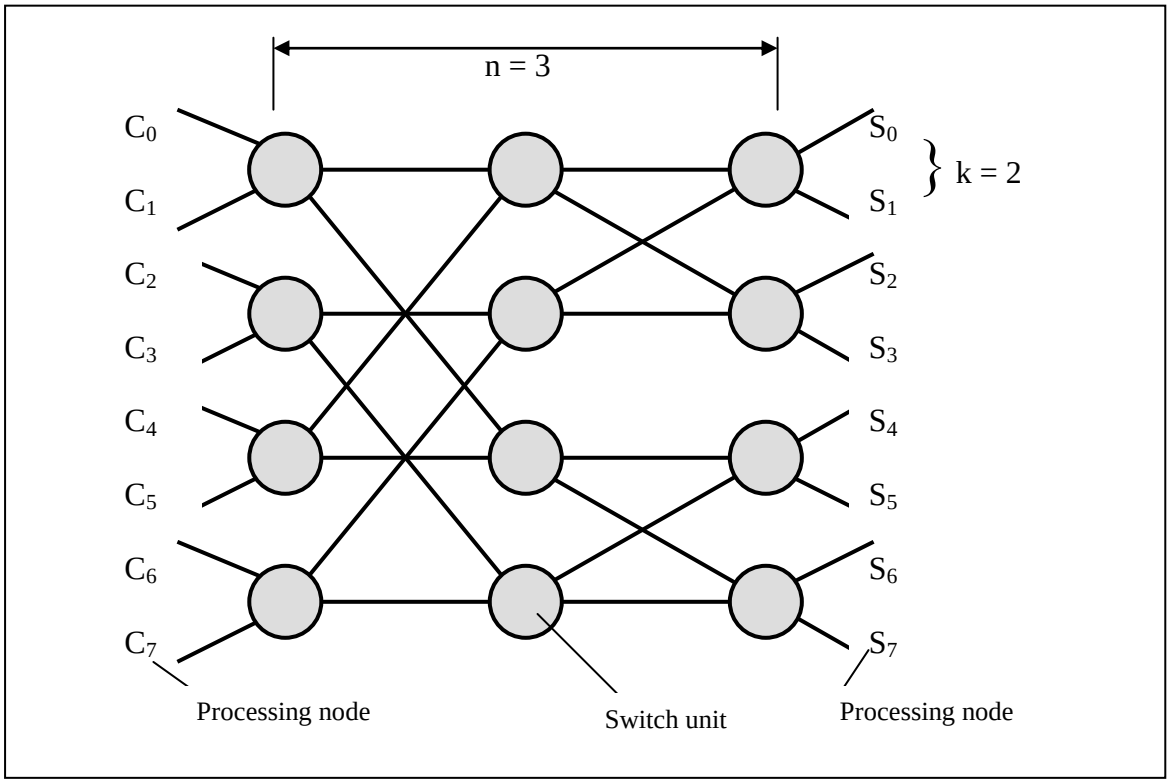
3.7.1 Basic characteristics

The next figure shows an indirect, multistage network modeled as a *butterfly of arity k and dimension n* (shortly, *k*-ary *n*-fly), in this example with $k = 2$ and $n = 3$.

It connects N processing nodes (C) with N processing nodes (S), in general of different kinds, where

$$N = k^n$$

For example, C nodes are PEs and S nodes are shared memory modules of a SMP multiprocessor, which is the most notable example of architecture using *k*-ary *n*-fly networks, along with shared memory SIMD machines.



The node degree is equal to *2k*.

The *distance*, coinciding with the diameter, is *constant* and equal to the dimension *n*. Thus, the *base latency* is proportional to *n*, i.e. a *k*-ary *n*-fly is a *logarithmic network*:

$$base\ latency = O(n) = O(\log N)$$

A *unique shortest path*, of length *n*, connects any C (or S) node to any S (or C) node.

Other multistage networks (Omega, Benes, etc) can be modeled as *k*-ary *n*-flies in terms of interconnection and communication properties.

3.7.2 Formalization of *k*-ary *n*-fly networks

In order to give a more formal definition of this structure and its properties, let us refer to a *binary butterfly* (2-ary *n*-fly). The treatment can be easily extended to any arity *k*.

A binary butterfly

- connects $N = 2^n$ processing nodes to $N = 2^n$ distinct processing nodes through *n levels* of *switch nodes*;
- each level contains *N/2* switch nodes;
- the two processing node sets are connected, two by two, to the first and to the last level of switch nodes.

The number of switch nodes and of links is, respectively:

$$n\ 2^{n-1}$$

$$(n-1)\ 2^n$$

Therefore, as order of magnitude, a butterfly is able to connect N to N processing nodes with

$$O(N \log N)$$

switch nodes and links, with an important saving compared to a $O(N^2)$ fully interconnected crossbar.

> It may be interesting to notice the analogy, in the area of signal processing, of the Fast Fourier Transform (FFT) with respect to the Discrete Fourier Transform (DFT). The FFT algorithm has exactly the binary butterfly topology, and reduces the complexity of DFT from $O(N^2)$ to $O(N \log N)$, where N is the amount of complex numbers to which the algorithm is applied. Programmed according to the data-parallel paradigm, FFT is a static-variable stencil computation which, with N virtual processors, is executed in $\log N$ steps, where, at i-th step, the stencil pattern is exactly determined by the topology of the i-th level.

The following ***connectivity rule*** holds:

- each switch node is defined by the coordinates (i, j), with
  - i is the row identifier: $0 \leq i \leq 2^n - 1$,
  - j is the column (or level) identifier: $0 \leq j \leq n - 1$;
- the generic switch node (i, j), with $0 \leq j \leq n - 2$, is connected to two switch nodes:
  - $(i, j + 1)$, through the so-called "straight link",
  - $(k, j + 1)$, through the so-called "oblique link", where k is such that

$$abs\,(k - i) = 2^{n-j-2}$$

In other words, the binary representation of k differs from the binary representation of i in only the j-th bit starting from the most significant.

The *recursive construction* of a binary butterfly is defined as follows:

- for $n = 1$, the butterfly consists of just one switch node with two input links and two output links;
- given the n dimensional butterfly, the $(n+1)$ dimensional butterfly is obtained by applying the following steps:
  - *i.* two n dimensional butterflies, topologically placed one over the other, represent the n final levels, each of 2^n switch nodes;
  - *ii.* the first level, of 2^n switch nodes, is added at the left;
  - *iii.* the first level nodes are connected to the second level nodes by applying the *connectivity rule*, so achieving the full *reachability* of the two processing nodes sets.

An n dimensional binary butterfly, with $n > 1$, is mapped *onto a (n-1) dimensional binary hypercube* (2-ary $(n-1)$-cube): this is achieved by merging all the butterfly nodes of the same row into one hypercube node, which is identified by the same binary representation of such row.

3.7.3 Deterministic routing algorithm for *k*-ary *n*-fly networks

The following routing algorithm derives from the above properties, and from the connectivity rule in particular. It applies to both network directions.

Let us consider a binary butterfly ($k = 2$), whose switch nodes are modeled as belonging to a matrix with N/k rows and n columns. Each processing node, C_i and S_j, is identified by a unique natural number represented by n bits. For example let C_3 = (011) the source node and S_6 = (110) the destination node. As usually, any message header contains the routing information consisting of such source and destination node identifiers.

Initially, the message is delivered from the source node to the directly connected switch node; for example, from C_3 to the switch node on the second row and first column. Now the algorithm follows n steps, corresponding to the network levels. During the i-th step, the switch node compares the i-th bit of the source and destination identifiers, starting from the most significant bit. If such i-th bits are equal, then the message is routed onto the straight link, otherwise onto the oblique link. In the example, the oblique link at the first step, and the straight link at the second step. The last step is executed by the switch node connected to the destination node, according to the value of the least significant bit of the destination identifier (in the example, to the first processing node).

For a S_j source and a C_i destination, the binary representation of identifiers is considered in reverse order, starting from the second least significant bit. For example, if S_6 is the source and C_3 the destination, the first link is straight and the second oblique. The least significant bit of the destination identifier is used for the final step delivery.

The algorithm is generalized for any arity k, by considering the k-base representation of processing node identifiers. At each step, the difference between the source identifier digit and the destination identifier one is used to uniquely identify the interface onto which the message is routed.

As for the other networks, *non-minimal adaptive* routing strategies, i.e. not exploiting the shortest paths, can be used.

Performance measures for k-ay n-fly networks will be given as evaluations of *generalized fat trees*: network structures that basically exploit k-ary n-flies to build high bandwidth trees.

3.8 Fat Trees

3.8.1 Channel capacity: from trees to fat trees

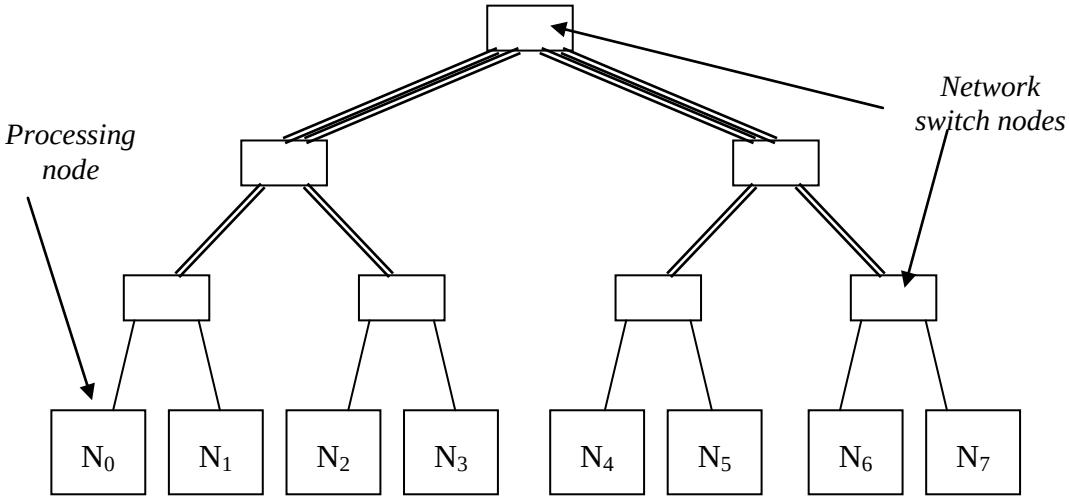
A *tree-structured* network can be used as an *indirect* networks for interconnecting nodes of the same kinds, notably PEs in a NUMA multiprocessor for both the processor-memory and the processor-processor structures, or in a multicomputer, or in a SMP multiprocessor for the processor-processor structure if distinct from the processor-memory one.

Communications occur between leaves, where the processing nodes are located/connected, while all the other nodes, including the root, are switch nodes.

The *deterministic routing* algorithm is relatively easy, since there is a unique minimal path between a pair of processing nodes *(i, j):* a message sent from i goes up the internal switches until it finds the nearest common ancestor and then down to j.

The *base latency* is *logarithmic*. However, for a "normal" tree, latency and bandwidth under traffic are penalized by the relatively high utilization factor of switch nodes and links (high conflict rate).

The solution for tree-structured indirect networks is the so-called *fat tree*, shown in the following figure:



With respect to a "normal" tree, in a fat tree *the link transmission bandwidth, or channel capacity, increases gradually from the leaves to the root*, in order to compensate the increasing congestion probability. In the figure the typical structure, in which the channel capacity doubles at each level, is shown. It is responsibility of the routing and flow control strategy *to select one of the output wires to evenly distribute the messages and to minimize congestion*. As consequence, the bandwidth is much greater than the "normal" tree bandwidth.

3.8.2 Average distance

The average distance, which impacts on the base latency evaluation, is the same for a normal tree and for a fat tree (instead, the under-load latency is very different).

Under the uniform distribution assumption, the average path length in a binary tree with 2^n leaves is given by:

$$d_{net} = \frac{\sum_{i=1}^{n} \frac{N}{2^i} 2(n-i+1)}{N-1} + 1 = \frac{\sum_{i=1}^{n} (n-i+1)\, 2^{(n-i+1)}}{2^n - 1} + 1$$

The following simplified formula holds with good approximation:

$$d_{net} = 1{,}9 * n$$

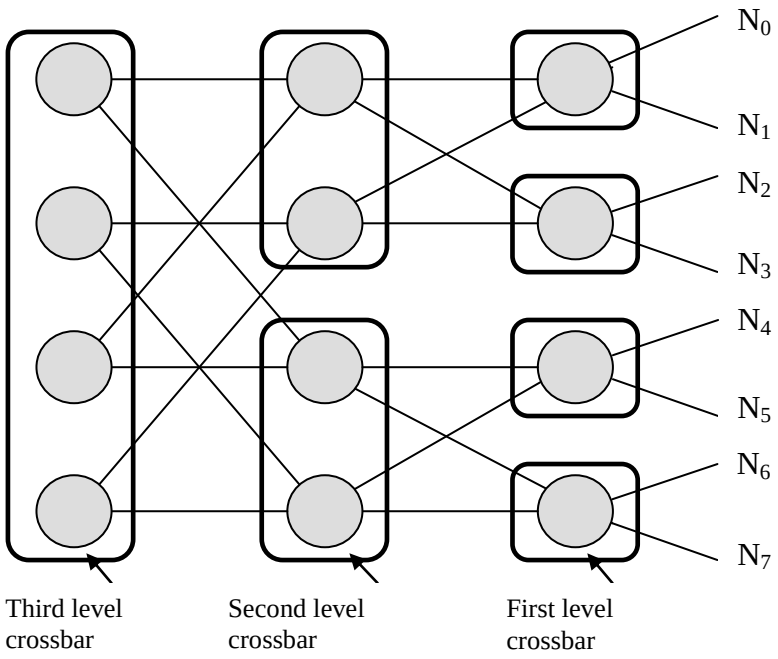
that is, a distance close to $2n$. For example, with $N = 256$ processing nodes ($n = 8$), we have $d_{net} = 15$. If communications are characterized by some locality, this evaluation is pessimistic especially when the program parallelism is much lower than N. In such cases, the majority of communications are concentrated inside the same half of the tree, thus $d_{net} \sim n$. However, $d_{net} \sim 2n$ is a realistic evaluation for highly parallel programs that use almost all the available processing nodes.

3.8.3 Generalized Fat Tree

The main problem, in the fat tree implementation, is the realization of switch nodes with increasing bandwidth and, at the same time, with a truly *limited node degree* or, equivalently, with an acceptable *pin count*. For example, in binary fat tree in which the channel capacity double at each level, a 1st level switch is a 2 x 2 crossbar, a 2nd level switch is a 4 x 4 crossbar, …, a *i*-th level switch is a 2^i x 2^i crossbar.

It should be clear that, from a certain level on, crossbars must be realized according to a modular and decentralized solution, *provided that* this does not penalize the latency and the bandwidth.

One effective and elegant solution to this requirement is the so-called *Generalized Fat Tree*. It is obtained from a *k-ary n-fly* structure, with $N = k^n$, by eliminating one of the two processing nodes sets, for example the C nodes. It is shown in the following figure:



At each fat tree level, the crossbar switch is realized by a proper number of butterfly switches operating *in parallel* without conflicts.

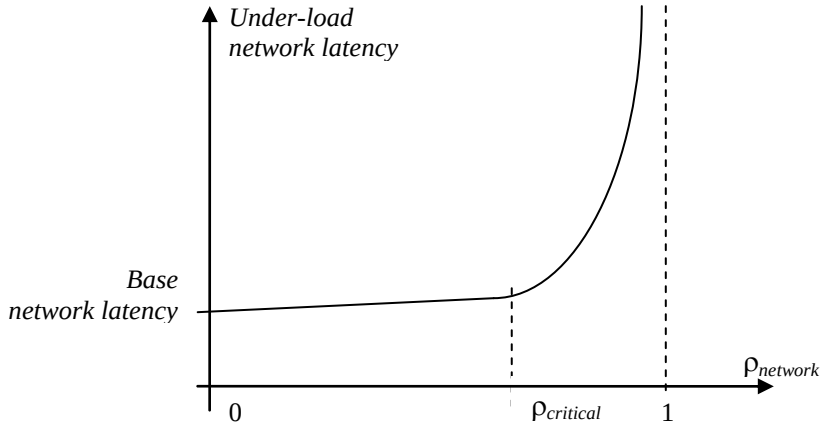
A *minimal adaptive routing* can be realized according to the butterfly properties, in order to minimize the conflict probability (the utilization factor), thus with good values of bandwidth and under-load latency. In practice, in a parallel architecture with Generalized Fat Tree network, as a first approximation *we can neglect the network conflicts with respect to the conflicts generated at the target shared memory module or destination node*.

In other words, *the critical utilization factor of a Generalized Fat Tree is relatively high* and the *cost model is quite stable* for different communication patterns.

Moreover, the Generalized Fat Tree is interesting for its flexibility too. That is, it can be used as a fat tree (for *N* processing node with distance *2n*) and as a butterfly (for *2N* processing node with distance *n*) at the same time, according to different message types, provided that *the switch nodes implement both routing strategies*. For example, in a SMP multiprocessor, the same network works for both processor-memory and for processor-processor communications.

3.9 Interconnection networks and CMPs

Let's come back to the under-load network latency as a function of the network utilization factor:



We have discussed how fat trees and generalized fat trees are potentially able to work in the "low latency region", owing to the strong reduction of switch congestion (and, possibly, to the adoption of smart routing algorithms). This is a typical case in which a bandwidth improvement leads to a latency improvement too (see Part 1 for the basic concepts about the bandwidth-latency relationships).

Moreover, once the congestion reduction goal is met, the network *base latency* becomes the other critical issue.

In CMPs, the network base latency is much lower than for parallel machines of the previous generation, and even than for multi-CMP architectures, owing to the very low transmission latency T_{tr}, which is close to zero, and to the wormhole flow control. However, not necessarily the current on-chip interconnection networks for highly parallel CMPs (meshes, multi-rings) have the same congestion reduction capability of fat trees and generalized fat trees. In this respect, the trend shown in Tilera and in some network processors is interesting: the bandwidth is partially improved, thus the congestion is partially reduced, through the adoption of *multiple networks*, one per specific task. Of course, the problem remains for the traffic of each network, in particular for the shared memory one and for the cache coherence one, depending on the parallel programs and their mapping (Section 2.3).

The reasons for which fat trees and generalized fat trees for CMP networks are still in the research phase is mainly related to their internal structure, and in particular to the different *length* of links. From this viewpoint, optical interconnect for NoCs is a very promising trend.

In this analysis, crossbars play an interlocutory role: low latency on-chip and low congestion are achieved at the expense of low parallelism. On the other hand, generalized fat trees show a possible trade-off, being a limited degree solution to crossbars for highly parallel structures.

A final consideration about the link transmission latency T_{tr}, which has a decisive impact in t_{hop}: currently, $T_{tr} \sim 0$ is typical of relatively low frequency CMPs (e.g. Tilera), otherwise in CMP exploiting more aggressive commercial VLSI technology T_{tr} is closer to 1τ o 2τ.

4. Shared memory cost models

In this Section we apply the concepts and techniques on interconnection networks to evaluate the shared memory access latency for multiprocessor architectures. Analytical cost models will be developed for the base latency and the under-load latency, in the latter case using queuing system theory. These evaluations are supported or integrated with experimental and simulation data.

4.1 Firmware messages for shared memory access

To refer to a concrete case, let us assume the following architectural characteristics:

- *NUMA* architecture;
- *all-cached* architecture;
- $N = 256$ processing nodes;
- *Generalized Fat Tree* based on a 2-ary 8-fly, with *wormhole* flow control and one-word flits;
- Physical addresses of 40 bits (1 Tera main memory), thus local addresses of 32 bit;
- Primary cache with block size $\sigma = 8$ words.

These assumption are done just for the sake of evidence and experience. However, the conclusions of this Section will be general and easily extendable to any architecture with any configuration.

Let us consider a typical organization of a processing node in a multiprocessor architecture (Section 1.1), shown in the next figure, including:

- *Node Interface Unit* W to/from the interconnection network, with service time equal to one clock cycle (τ);
- Communication Unit (UC) for serving inter-processor direct communications;
- *Interleaved local memory macro-module*, $M_0, \ldots, M_7$, with independent interface unit I_M. This unit, with service time equal to 1τ, can request the access to all local memory modules simultaneously for reading/writing a 8-word block. Block data from the modules are received in distinct I_M interfaces, and are sent to W one word at a time in pipeline.

All the intra-node links are explicitly marked with their width in bits for a 32-bit machine.

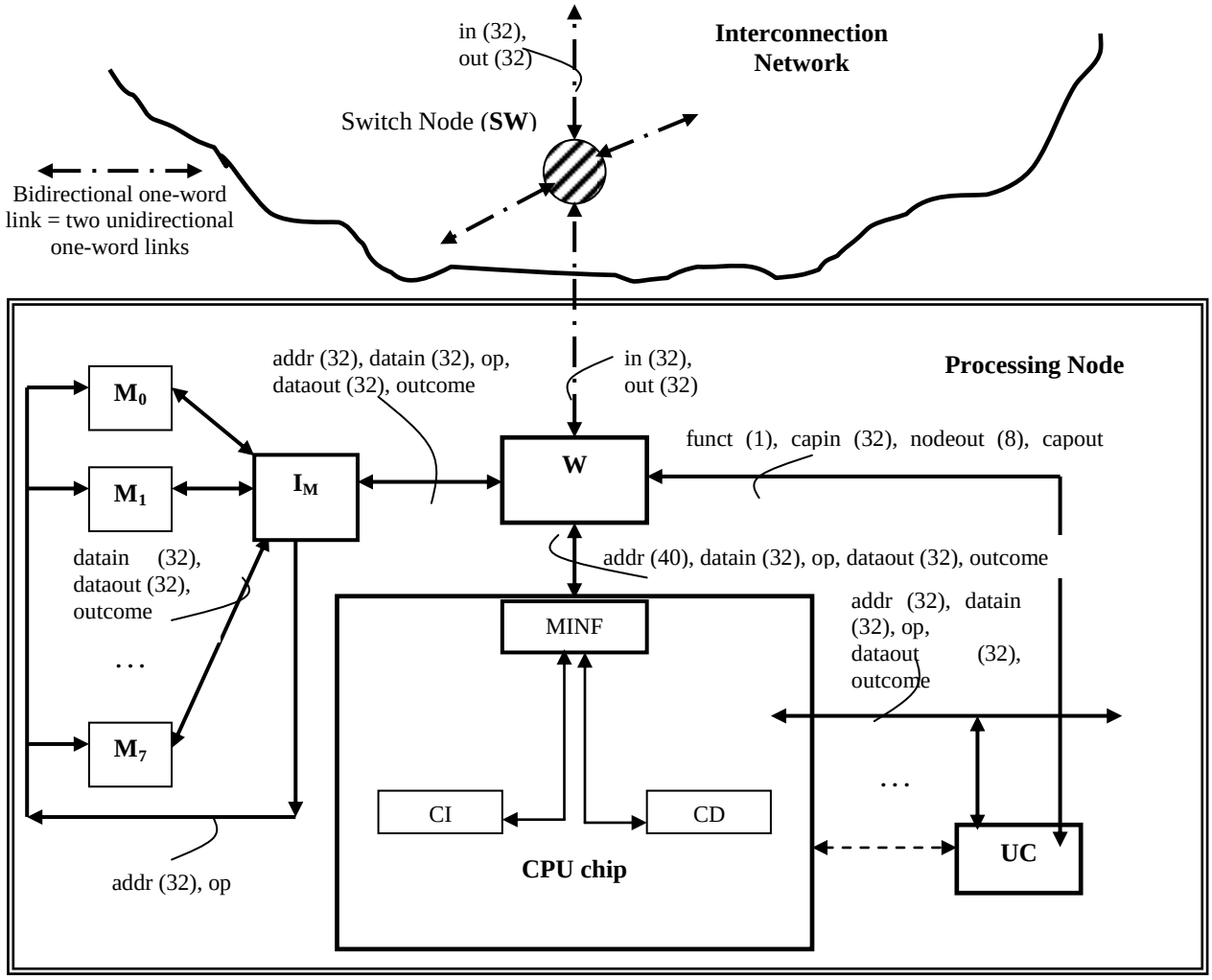
W unit receives from CPU (through the CPU memory interface unit, MINF) memory access requests for reading a cache block:

CPU_*R = (block physical base address, operation)*

or for writing a cache block:

CPU_W = (block physical base address, 8-word block, operation)

Through the most 8 significant bit of the block physical base address, W is able to distinguish requests to be forwarded to the local memory or to a remote memory.



Writing semantics

Writing operations may be implemented according to two different semantics:

- ***synchronous writing***: a reply, signaling the writing termination and possibly the outcome, is returned;
- ***asynchronous writing***: no reply is returned, this asynchronous writing operations are fully overlapped with the successive instructions.

Asynchronous writing is implemented in every system. While in some multiprocessor architectures this is the only writing semantics, in other machines also the synchronous version is provided. I

In principle, no there is no reason for synchronous writing, because possible "delayed" exceptions can be handled asynchronously, as it happens for the majority of exceptions and interrupts in ILP machines (Part 1, Section 19, 20). However, serious *memory consistency* problems are rendered more complex when asynchronous writing is adopted.

This issue will be studied in Section 5.4, where we'll see that additional instructions are required or some forms of synchronous writing has to be provided in addition to asynchronous writing, especially in the implementation of critical synchronization sequences.

Firmware messages

For local memory accesses, the CPU request is forwarded to I_M (the most significant 8 bit of the physical address are eliminated), then W receives the read reply on a single interface. Data blocks between W and I_M are transmitted in pipeline. W has a non-deterministic behavior, thus, between request and reply, W can receive other compatible messages, for example remote requests from the network for memory accesses or inter-processor communications.

For remote memory accesses, W sends a firmware message of type 0 (read block), or type 2 (write block), described in the following.

As a reply to message of type 0, W will receive from the network:

- a 8-word data block + an outcome value (message type 1), forwarded to CPU (MINF-Data Cache) word by word in pipeline

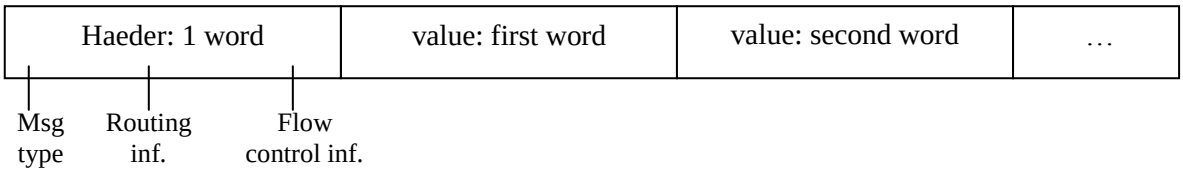
Analogous (message of type 3) for synchronous writing, if provided.

As said, between request and reply W can listen other messages and perform other services.

Messages of types 4 and 5 are for inter-processor communication, respectively from the network and from the local UC.

All the formatting and assembly/de-assembly operation in W are executed in 1τ, thus with zero overhead.

The firmware messages have the formats described as follows. The first word (first flit) is the *message header*:



For example:

- 4 bits: message type
- 8 bits: source node identifier
- 8 bits: destination node identifier; in this case, it is a memory macro-module, identified by the most 8 significant bits of physical address
- 8 bit: message length (number of words)
- 4 bits: other functions

The field sizes have been established having different configurations in mind, thus they are dimensioned to support the maximum configuration.

Let us describe the message formats corresponding to typical message types. Owing to the short message sizes, for efficiency reasons distinct information are adjusted to the word length ("padding"), with some limited waste of bits.

Message type 0: read block request (2 words). Value:

- physical address relative to the memory module: 32 bit.

Message type 1: block value and outcome received from memory (10 words). Value:

- outcome (8 bits);
- not used (24 bits);
- data (8 words).

Message type 2: write block request (10 words). Value:

- block physical base address (32 bit);
- data (8 words).

Message type 3 (if synchronous writing is provided): signaling of writing termination (1word). Value:

- outcome (8 bits);
- not used (24 bits);

Message type 4: ...

Message type 5: ...

The figure in the next page illustrates the communications involved in a *remote block read operation*. A path through the interconnection network crosses an average number of switch nodes, given by d_{net} = 15 in the worst case, or d_{net} = 8 in the best case for the assumed generalized fat tree.

4.2 Base memory access latency

The formula of Section 3.5.4

$$\Omega = (m + d_{int\text{-}source} + d_{net} + d_{int\text{-}dest} - 2)\, t_{hop}$$

will be used for the pipelined communication latency, exploiting the wormhole flow control of the interconnection network, as well as the pipelined behavior of other units.

For any memory operation in an all-cache architecture, in the source node the request firmware message originates in the Data Cache unit, and is propagated through the External Memory Interface unit MINF to the Node Interface unit W. Thus, $d_{int\text{-}source}$ = 3.

In the destination node, the firmware message is forwarded from W to the Internal Memory Interface unit I_M, then from I_M to the memory modules (both for reading and fro writing requests). Here the pipelined behavior, common to all memory request, terminates, since successive actions depend on the specific memory operation. Thus, $d_{int\text{-}dest}$ = 3. Notice that, a reading request is forwarded from I_M to all the memory modules in parallel.

In conclusion, for a read or write request, the distance is given by:

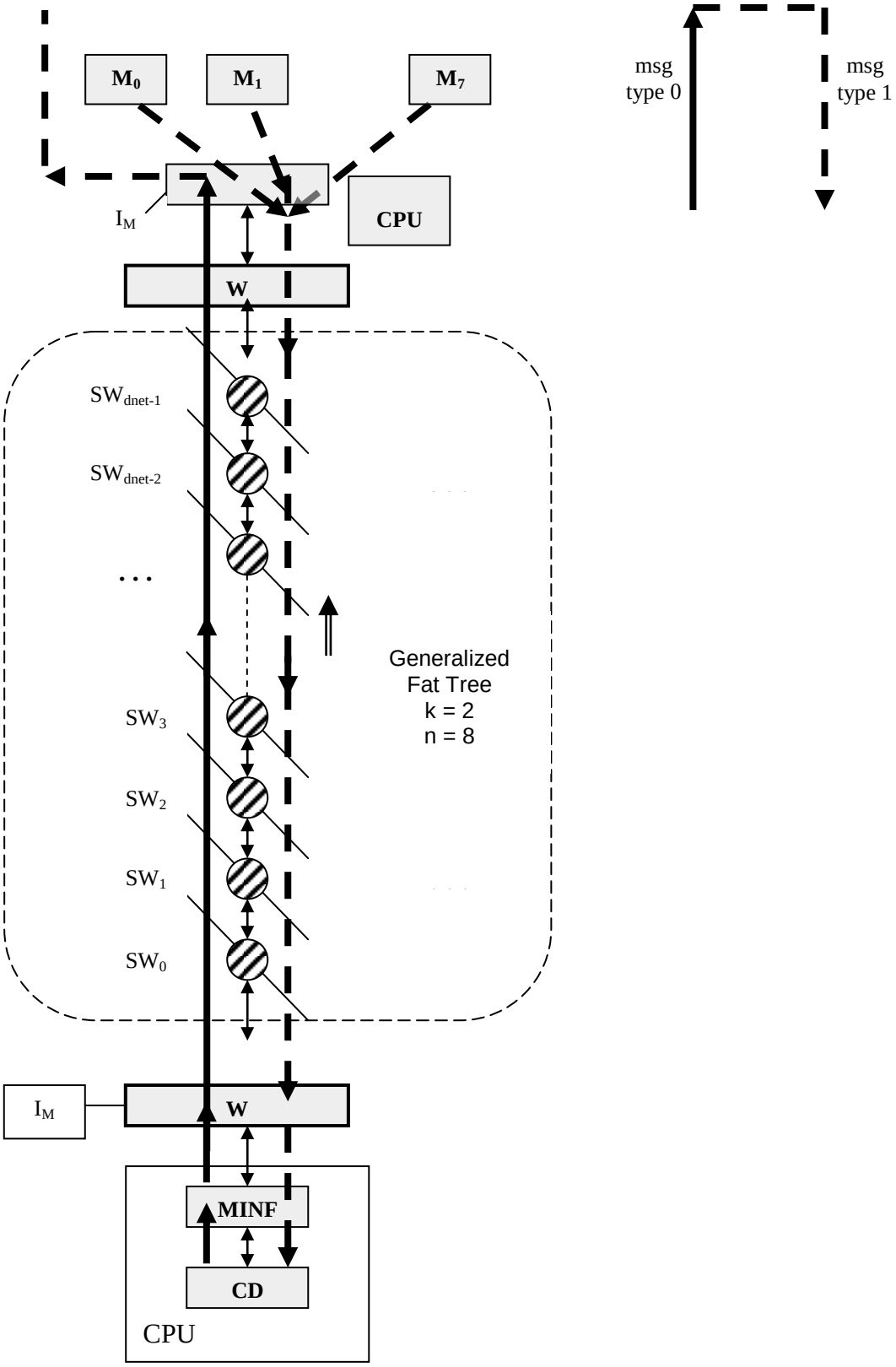
$$d_{int\text{-}source} + d_{net} + d_{int\text{-}dest} = d_{net} + 6$$

For a *read request* (message type 0):

$$m = 2: \qquad \Omega_{\text{read-req}} = (6 + d_{net})\, t_{hop}$$

For a *write request* (message type 2):

$$m = \sigma + 2: \qquad \Omega_{\text{write-req}} = (6 + \sigma + d_{net})\, t_{hop}$$



With a fat tree:

$$d_{net} = \delta n$$

where $n = lg_2 N$, and the δ parameter is such that $1 \leq \delta < 2$according to the locality degree of interprocess communications, or of any mechanism operating on shared memory.

The block reading operation includes the parallel read access on the destination PE modular memory, having a latency:

$$\tau_M$$

It is followed by the reply message which is pipelined from the destination memory to the source node Data cache, with distance d_{net} + 6 (as before) and a message length $m = \sigma + 2$. Thus this phase requires:

$$\Omega_{\text{read-reply}} = (6 + \sigma + d_{net})\, t_{hop}$$

The total base latency of a block reading operation is given by:

$$\Omega_{0\text{-read}} = \Omega_{\text{read-req}} + \Omega_{\text{read-reply}} + \tau_M = (12 + \sigma + 2\, d_{net})\, t_{hop} + \tau_M$$

In general, the ***block reading base latency*** can be expressed as:

$$\boldsymbol{\Omega_{0\text{-read}} = \Omega_{\text{read-req}} + \Omega_{\text{read-reply}} + \tau_M = (c_1 + \sigma + 2\, d_{net})\, t_{hop} + \tau_M}$$

where c_1 is an architecture-dependent constant.

For a *CMP* with $t_{hop} = 2\tau$ ($T_{tr} = 1\tau$) and memory clock cycle $\tau_M = 20\tau$, in the example $\Omega_{0\text{–read}}$ ranges from 92τ to 120τ, thus order of $10^2\tau$. It is worth noticing the impact of the link transmission latency T_{tr}: passing from $T_{tr} = 1\tau$ to $T_{tr} \sim 0$, the network latency is halved, and $\Omega_{0\text{-read}}$ reduces to 56τ or 70τ. As discussed in Section 3.9, currently $T_{tr} \sim 0$ is typical of CMPs with relatively low clock frequency.

For a multiprocessor on multiple chips, the most notable overhead factors are T_{tr} (order of magnitude of few 10τ) and τ_M (order of magnitude of several 10τ). For example, with $T_{tr} = 20\tau$ *and* $\tau_M = 50\tau$, $\Omega_{0\text{-read}}$ ranges from 756τ to 1050τ, thus order of $10^3\tau$.

An *asynchronous* write operation terminates in the destination PE with the pipelined block writing in the modular memory, thus the ***asynchronous block writing base latency*** is given by:

$$\Omega_{0\text{-as-write}} = \Omega_{\text{write-req}} + \tau_M = (c_2 + \sigma + d_{net})\, t_{hop} + \tau_M$$

In the example, from 76τ to 90τ for $t_{hop} = 2\tau$, and from 48τ to 55τ for $t_{hop} = 1\tau$.

Roughly, $\Omega_{0\text{-as-write}}$ is equal to about 70%- 80% Ω_{read}. This means that the reply penalty in tread latency is not so strong. However, $\Omega_{0\text{-as-write}}$ is not paid in the program performance, when the asynchronous behavior is semantically correct.

A *synchronous* write operation contains the reply phase. At most τ_M can be overlapped, so in the best case the ***synchronous block writing base latency***

$$\boldsymbol{\Omega_{0\text{-s-write}} = \Omega_{\text{write-req}} + \Omega_{\text{write-reply}} = (c2 + \sigma + 2\, d_{net})\, t_{hop}}$$

if the operation outcome is not returned, otherwise:

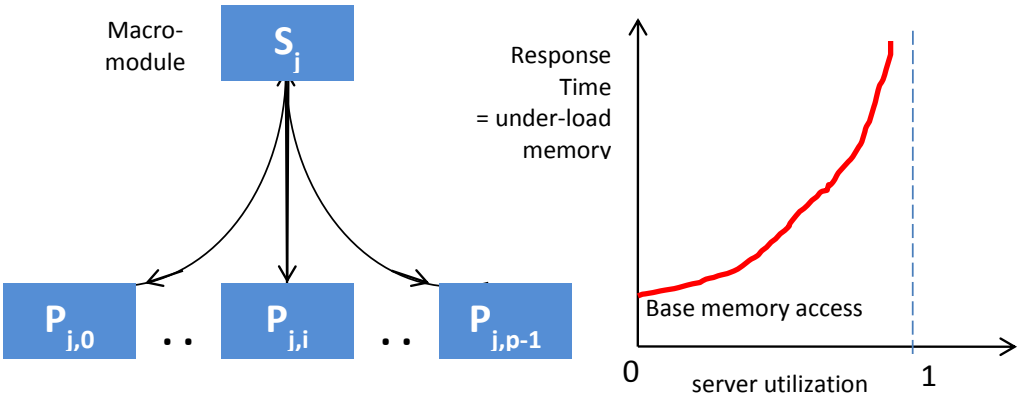
$$\boldsymbol{\Omega_{0\text{-s-write}} = \Omega 0_{\text{read}}}$$

As an acceptable approximation, we can assume that the synchronous writing has always the same cost of the reading operation.

4.3 Under-load memory access latency

Queuing model

The queuing model of a multiprocessor system has *m* distinct and independent servers (shared memory macro-modules), each of which is used by an average amount ***p*** of clients, i.e. *p* is the *average number of processing nodes sharing the same memory macro-module*:



Each server corresponds to a shared memory macro-module. Logically, the interconnection network path from each of the *p* nodes to the shared macro-module, including the used switch nodes, are included in the node itself.

In a SMP architecture, in which statistically the memory accesses are uniformly distributed over the *m* macro-modules, *p* can be estimated as the mean of the binomial distribution (Section 2.4.2).

In a NUMA architecture, the uniform distribution does not hold, and the *p* value depends on specific characteristics of the parallel program and its mapping onto the processing nodes. In Section 4.4 we will discuss the mapping problem in NUMA architectures.

The under-load memory access latency, denoted by **Ω**, *is given by the server response time* R_Q.

With asynchronous writing, writing operations have no impact on the client service time, however they have the same impact of reading operations on the server congestion. For this reason, the analysis is done for any read/write operation, thus assuming the same base latency:

$$\Omega_0 = \Omega_{0\text{-read}} = \Omega_{0\text{-write}}$$

In NUMA architectures it is meaningful to distinguish between remote memory access latency (Ω_{rem}) and local memory access latency (Ω_{loc}). Assuming that, in case of conflict between a remote request and a local request, this last is served with higher priority, we can write:

$$\Omega_{loc} = \Omega_0 \, (1 + \rho)$$

where ρ is the server utilization factor, which is a result of the queuing model resolution.

Assumptions and approximations

The cost model for a shared memory multiprocessor implies a complex evaluation because of the rather large number of variables, architectural variants and, perhaps most important, many different situations that are related to the parallel application characteristics and to the way in which the parallel computation exploits the multiprocessor architecture.

Our goal is to define a method for the under-load memory access evaluation, which is characterized by an acceptable complexity and, at the same time, is able to capture the essential elements and the qualitative behavior. This implies an approximate approach based on some assumptions. The most meaningful assumptions are:

1. all the *conflicts are concentrated on the memory macro-modules* only, i.e. conflicts on the network switch nodes and links have a negligible impact. Clearly, this assumption is a simplification, which is reasonable for fat-tree based networks and for the most recent trends in CMP technology. For different kinds of networks, in general also the conflicts along the interconnection network have to be evaluated: the cost model could be improved to include such conflicts too;
2. let $\boldsymbol{T_p}$ be the mean time between two consecutive accesses of the same PE to a memory macro-module (during this time, the processor executes instructions operating on registers, or local caches). We assume that T_p be the mean value of an *exponentially distributed* random variable. Actually this distribution depends on the parallel application characteristics, and could be different from the exponential one. However, for our purposes, we are interested in evaluating the interarrival time distribution, which can be approximated as an exponential one because of the independent behavior of the various PEs. In other words, for many different distributions of the time between two consecutive accesses, the combination of the requests of p processing nodes is approximately characterized by a random behavior. Thus, for the W_Q evaluation a good approximation is represented by the *M/D/1* queue. Again, a more accurate evaluation, taking into account the parallel application characteristics, could be provided to improve the analysis;

The *server service time* and *latency* are estimated as the memory macro-module clock cycle, τ_M, since the memory unit is the bottleneck of the pipelined behavior source PE – network stages – destination PE. Queuing Theory analytical results are valid for single servers (or, when multiple servers are considered, they are merely independent servers; as in our case of modular memories). Thus, for an analytical approach we are forced to exploit formulas for W_Q (the mean waiting time in queue) which have been derived for servers having equal service time and latency. For this reason, the latency of the path from a node to a macro-module

$$\boldsymbol{\Omega_{net} = \Omega_{req} + \Omega_{reply}}$$

has been included in the client ideal service time, i.e. added to T_p. The under-load memory access latency is evaluated as

$$\boldsymbol{\Omega = R_Q + \Omega_{net}}$$

where R_Q is the response time.

The approximation has been evaluated by comparing the results with simulations. Typically, the error is about 20% when the server utilization factor ρ is high (close to one), while is much lower for servers with medium-low ρ. That is, the method is sufficiently approximated in the cases of our main interest. The approximation on Ω is always by-excess, thus reliable for the utilization in the cost model of parallel programs.

Model resolution

Applying the methodology of Part1, Section 15, the response time R_Q is the solution of the system of equations:

$$\begin{cases} T_{cl} = T_G + R_Q \\ R_Q = W_Q(\rho) + L_s \\ \rho = \dfrac{T_s}{T_A} \\ T_A = \dfrac{T_{cl}}{p} \end{cases}$$

under the constraint

$$\rho < 1$$

where:

$$T_G = T_p + \Omega_{net}$$
$$L_s = \tau_M$$
$$T_s = \tau_M$$

From Section 15.1.2 of Part 1 we know that, for an infinite M/D/1 queue:

$$W_Q = T_s \frac{\rho}{2(1-\rho)}$$

The deterministic distribution of service time gives a much better approximation than the exponential service distribution, which is the most studied case in Queuing Theory. Using the infinite queue model has not a decisive impact on the accuracy, provided that the asynchrony degree is (at least) equal to the number of clients.

The solution of the queuing model is a second degree equation in ρ:

$$A\rho^2 + B\rho + C = 0$$

with coefficients:

$$A = 2\,(T_G - T_s) \qquad B = -2\,(T_G + p\,T_s) \qquad C = 2\,p\,T_s$$

This equations has always two real, positive roots, ρ_1 and ρ_2, with $\rho_1 < 1$ and $\rho_2 > 1$. Thus, ρ_1 is the solution of the queuing model.

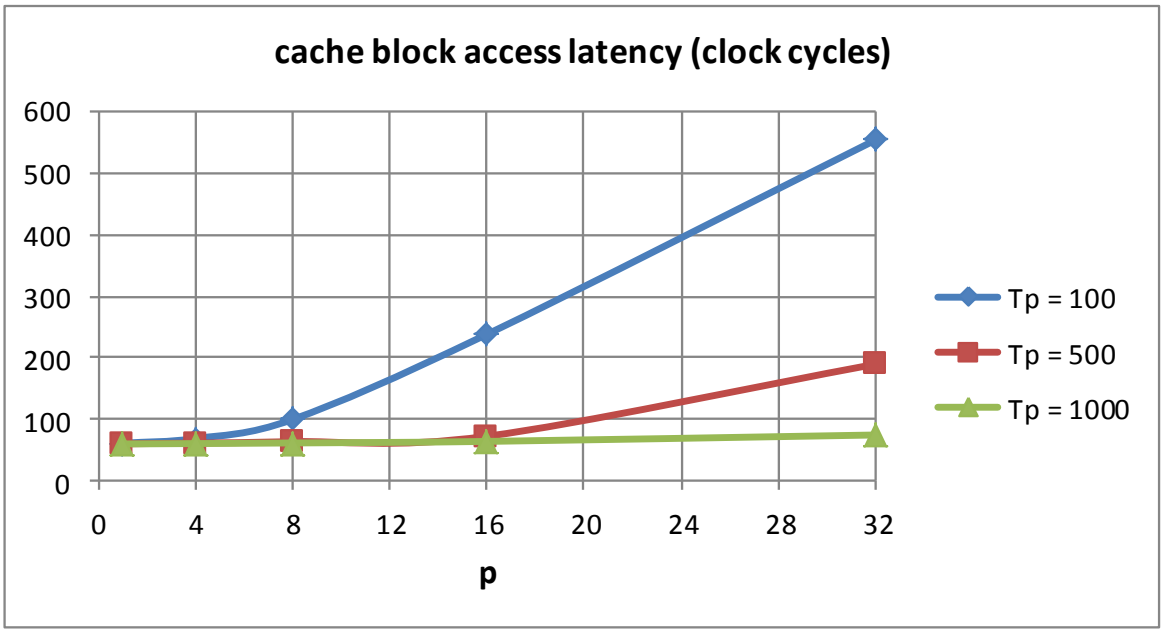
In the next figure, the memory access latency for a cache block

$$\mathbf{\Omega = R_Q + \Omega_{net}}$$

is shown as a function of the most relevant parameters p and T_p, for $\Omega_{net} = 40\tau$ and $\tau_M = 20\tau$, and typical values of the other parameters (Section 4.2).

The effect of p shows the importance of "*low-p* mappings" of parallel programs, as it will be discussed in the next Section.

As it is expected, the effect of T_p is heavy for *fine grain* computations (the utilization factor increases), while for *coarse grain* computations the impact on memory conflicts tends to become negligible, so the under-load latency Ω tends to the base one Ω_0 for large T_p values.



For $T_p = 1000\tau$ ρ varies between 0.02 and 0.6, for any p, with Ω very close to Ω_0, while for $T_p = 100\tau$ ρ varies between 0.12 and 0.98.

In the fine grain case ($T_p = 100\tau$), we observe the importance of p: for $p < 4$, ρ varies between 0.12 and 0.47, correspondingly Ω remains very close to Ω_0. That is, ρ = 0.47 can be considered the *critical utilization factor* for $T_p = 100\tau$.

For $T_p = 500\tau$ the critical ρ is equal to 0.56, with $p = 16$.

In conclusion, even for fine grain computations, a properly low value of p provides latencies close to Ω_0.

Moreover, it can be shown that;

a) The impact of the number of nodes N is not so relevant on Ω, while it is relevant on the base latency Ω_0. In terms of contention evaluation, the real parallelism degree is p instead of N.

b) The impact of the cache block size σ is significant. However, if the *macro-module parallelism* is high, larger blocks can be exploited: they have a positive impact on the overall application bandwidth, since the number of remote accesses is proportionally reduced. From this point of view, the importance of *wormhole* flow control has to be remarked.

c) The impact of δ is not too meaningful for low values of p. That is, basically, for "*low-p* mappings" (see next Section), the communication impact tends to be *distance insensitive*.

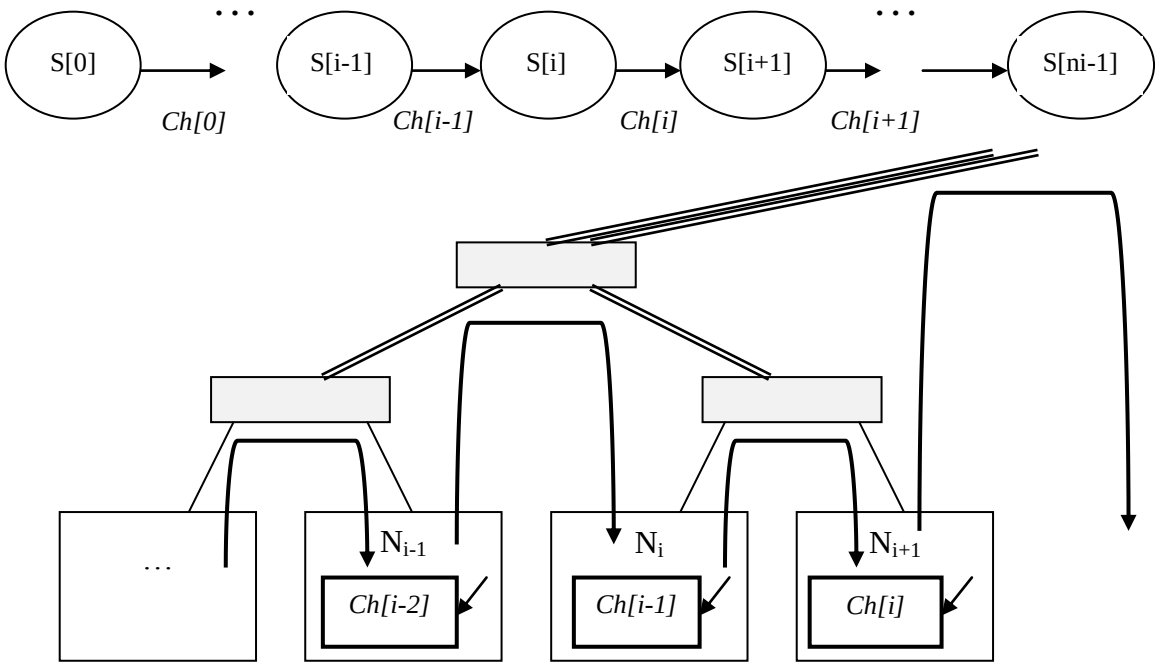
d) The impact of the memory clock cycle is significant for slow memory technologies. However, notice that, owing to the memory hierarchy, the impact is limited for memory clock cycles of the order of few tens τ, as in CMPs.

4.4 On parallel programs mapping and structured parallel paradigms

The main goal of parallel program mapping in a NUMA architecture is to keep the *p* value as low as possible: let's use the term ***low-p mapping*** to denote such a mapping strategy. It is worth remarking that the performance problems, though related to the distance effects, are *manly* influenced by the *contention* effects. An efficient exploitation of the architecture is allowed by data structures shared by a relatively low number *p* of nodes, *even if* the nodes are relatively distant. In fact, we know that the most interesting interconnection logarithmic structures, notably the fat tree, are relatively *insensitive to the distance*. In the base latency evaluation of Section 4.2, the difference between access latencies with $\delta = 1$ and $\delta = 2$ is few tens of clock cycles.

Consider the case study of Sect. 2.3, where the parallel program was structured as a data parallel pipeline with $n = 100$ identical stages, or as a farm with $n = 100$ identical workers. Let us assume that the parallel program is expressed in a message-passing formalism, as LC. In a NUMA run-time support, the shared data structures are *channel descriptors and target variables* only.

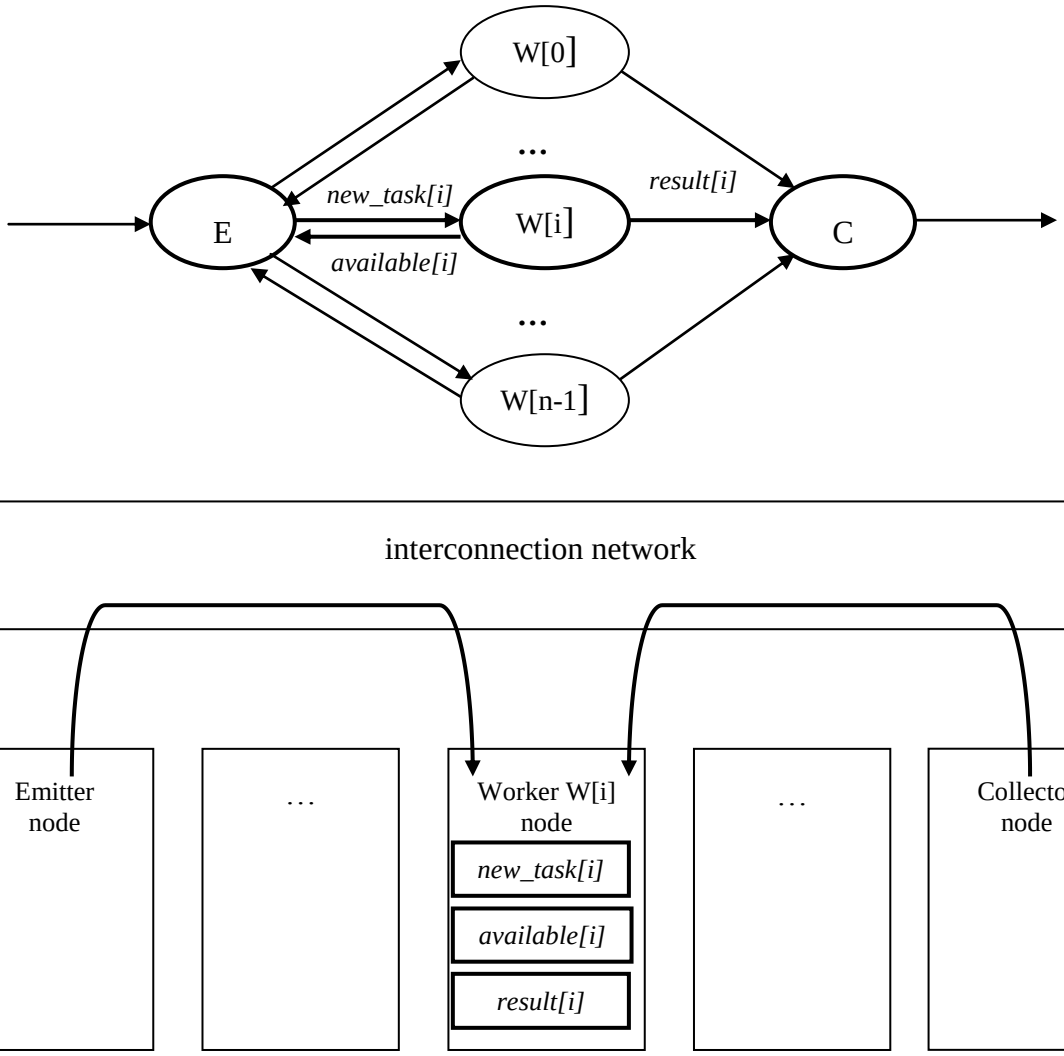
In the *data parallel pipelined* version, each channel descriptor is shared by two nodes (onto which the corresponding stages are mapped), thus $p = 2$, including the node having the channel descriptor in its local memory: this last can be the node onto which the *destination* stage is mapped:



Each memory macro-module is shared only by the local processor and by the processor onto which the preceding stage is mapped. The achieved *p* value represents a very interesting result, leading to negligible contention, thus the *under-load latency is very close to the base one*.

All the possible communication distances are present in the physical communication pattern. This means that the δ parameter, used as a relative evaluation of the average distance d_{net}, is close to 2, thus relatively high, despite the simplicity of the pipeline communication pattern. However, the effect of the low *p* value on under-load latency is prevailing with respect to the distance.

The best mapping for the *farm* version consists in using *symmetric channels* only, and, for any worker, in allocating the input channel descriptor (from emitter to worker) and the two output channel descriptors (from worker to emitter and from worker to collector) in the local memory of the node onto which the worker itself is mapped:



The emitter node and the collector node share all the local memories of worker nodes, while each worker node accesses its own local memory only. In this way we have $p = 3$, which again is a very good results for sharply reducing the contention effects, though the whole communication pattern exploits the full network, with δ close to 2.

For the farm paradigm some mappings exist that are very inefficient. Notably, the strategy "always allocate the channel descriptor in the local memory of the *destination* process node" is far from the optimal one. In a farm, if the communications from workers to emitter and to collector are expressed by *asymmetric* channels, the channel descriptors are forced to be allocated in the emitter node local memory and in the collector node local memory, then p is about equal to n, i.e. all the workers nodes share, and are in conflict on, the local memories of the emitter node and of the collector node.

Low-p mapping strategies, exploiting *symmetric channels*, can be applied to the implementation of collective communications in any *data parallel* paradigm (scatter, gather, multicast), as well as to collective operations (reduce), and to stencil communications.

Also limited degree process structures are suitable for *low-p* mapping, notably

- *process trees*
- *process rings* and *linear chains*.

As a conclusion, we have verified another very important property of *structured parallelism paradigms*: owing to the detailed knowledge of the process patterns, we are able to identify some efficient implementation strategies, notably in terms of communication forms and the related *low-p* mappings.

5. Processor synchronization

This Section deals with concepts and techniques on processor synchronization mechanisms for a correct and efficient utilization of shared memory.

Processor synchronization rely on the existence of proper mechanisms supported by the firmware architecture. We'll use the general term *locking* to denote processor synchronization mechanisms. Though a very large variety of locking mechanisms exist, we'll concentrate on basic techniques according to which all the main architectural and cost model issues can be studied.

As anticipated in the Abstract of Part 1, for the sake of explanation in this Section the synchronization techniques are evaluated without explicit references to the cache coherence issues, though basic principles of cache coherence are utilized. The detailed evaluation of synchronization mechanisms will be studied in Section 7, exploiting the cache coherence treatment of Section 6.

5.1 Processor synchronization issues

Locking mechanisms are used *to synchronize distinct processors* executing sequences of operations as *indivisible*, or *atomic*, actions. A sequence of operations, which can be executed by more than one processor, is indivisible, or atomic, if no concurrent execution is allowed for the sake of correctness. For example, let us consider two processors that can execute the following computations:

Processor_1:: p; c; q Processor_2:: r; c; s

Let us assume that the computation semantics is such that operation *c* must be executed at most by one processor at the time (by Processor_1, or by Processor_2, but not both), while no other constraint exists about the parallel execution of the other operations. This means that *c* (which, in turn is a *sequence* of operations) is an atomic (*sequence* of) operation(s). More in general, in

Processor_1:: p; c1; q Processor_2:: r; c2; s

c1 and *c2* are atomic if cannot be executed concurrently by Processor_1 and Processor_2.

Often (but not in general) atomic actions operate on *modifiable shared data*, that are maintained in a consistent state if no concurrent manipulations is allowed, e.g. a *read-modify-write* sequence.

Locking mechanisms extend what in a uniprocessor system is merely implemented through interrupt disabling (in a multiprocessor this is just a necessary, but not sufficient, condition for atomicity). For example, in the *send* run-time support (as in any other primitive at the process level), some sequences to be rendered indivisible are:

a) read-modify-write sequences on the channel descriptor. For example, assuming a *send* version in which the validity bit is not used:

< read the buffer insertion pointer;

modify insertion pointer and buffer current size;

read the wait boolean; if true:

modify wait >

b) wake-up operations on the ready list in a SMP architecture.

An indivisible sequence of actions is enclosed between two synchronization operations, called ***lock*** and ***unlock***, operating on a *locking semaphore* data type. For example, if *ch_semaphore* is a locking semaphore associated to the communication channel descriptor:

lock (ch_semaphore);

read the buffer insertion pointer;

modify insertion pointer and buffer current size;

read the wait boolean; if true:

modify wait

unlock (ch_semaphore);

This scheme implements a *mutual exclusion* of the enclosed sequence, i.e. at most one processor at the time is able to execute this sequence. If several processors try to execute the sequence simultaneously, then just one of them is able to complete the *lock* operation, while the others are blocked inside the *lock* operation itself. Once the sequence is completed, the *unlock* operation provides to unblock one of the possibly waiting processors. This semantics is feasible provided that *lock and unlock themselves are indivisible* operations.

Semantically, *lock-unlock* are similar to any semaphoric operation pair, like *P-V* or *wait-signal*. However, the difference is that:

i) the *lock-unlock* atomicity is implemented directly at the firmware level (while, in turn, the P-V atomicity is implemented by lock-unlock brackets),

ii) the *lock-unlock* mechanism synchronizes *processors* (not processes), thus it implies ***busy waiting*** (while P-V is characterized by the process transition into the waiting state, accompanied by a context switch, *unless* an exclusive mapping approach is adopted).

Point *ii)* implies an accurate trade-off between locking overhead minimization and software lockout minimization, as discussed in Sect. 2.4.3.

5.2 Indivisible sequences of memory accesses

Point *i)* about lock-unlock atomicity implies that

- the *lock-unlock* atomicity is obtained recognizing one or more *indivisible sequences* of *memory accesses* in the *lock* and *unlock* algorithms. For example, an indivisible sequence of memory accesses could be: *read (addr); read (addr + 1); write (addr*). In this example, *only the memory accesses* of an indivisible sequence of operations are pointed out: that is, in this context some calculation performed before the write operation are not of interest;
- indivisible sequences of memory accesses are implemented with the help of the *shared memory arbitration mechanism*. This can be done in at least two ways:

  1. block the access to currently used locking semaphore locations, or
  2. block the access to the entire memory module containing the locking semaphore.

  In practice, because of the relatively short duration of *lock* and *unlock* sequences on locking semaphores, the *second* solution should be preferred.

5.2.1 Processor mechanisms

This firmware mechanism, operating on the shared memory modules arbitration, employs an additional bit, called *indivisibility bit (indiv)*, belonging to the memory access request generated by the processor, e.g.

memory access request = (logical address, data, memory operation,
memory hierarchy management annotations, ***indiv****)*

The *indiv* bit is forwarded towards the memory module (along with physical address, data, and so on), and it is properly used by the one or more units in the interconnection path from the processor to the memory module (as usually, in an all-cache architecture, this situation occurs when a cache block is transferred from/into shared memory).

We can model an indivisible memory access sequence of *h* accesses in the following way: all the first *h-1* requests contain *indiv* = 1, and the *h-th* request contains *indiv* = 0.

Of course, in order that the *indiv* value is generated by the processor interpreter, the assembler machine must contain proper *instructions or annotations*:

- in many systems, assembler instructions for executing an indivisible *read-modify-write* sequence on a single location exist. For example, Test and Set (R, Y), Exchange (R, Y), Subtract from Memory (R,Y), Add to Memory (R, Y), where R denotes a processor register and Y the address of the memory location on which the sequence has to be performed;
- a more powerful and flexible technique consists in annotations inserted in any instruction that can refer shared memory locations (Load, Store).

In the didactic assembler machine D-RISC, both techniques are provided:

- explicit SET_INDIV, RESET_INDIV and Exchange instructions,
- *set_indiv* and *reset_indiv* annotations in Load and Store instructions.

5.2.2 Impact on memory behavior and interconnection structure

Once the assembler-level mechanisms are defined, the processor interpreter is straightforward (merely, put the *indiv* bit to 1 or to 0 in the memory output interface). The system strategy to implement indivisible sequences of memory accesses, according to the *indiv* value, characterizes the specific multiprocessor architecture.

In order to understand the problem, consider initially the two extreme situations, in which the interconnection structure is a *crossbar* or a single *bus* (Section 3.3):

a) in the *crossbar-based* architecture, a memory module (or its interface unit) has *N* input distinct links, one from each processor. The memory module firmware interpreter consists in *non-deterministically* testing all the *N* interfaces, in the same clock cycle, and in selecting and serving one of the ready requests. Let *J* be the interface identifier of the selected request. If this request contains *indiv* = 1, a *deterministic* phase is entered: only the *J-th* interface is listened and served until a request with *indiv* = 0 is received. During this period, *all the other possible requests are just waiting in their respective interfaces*. When *indiv* = 0, the non-deterministic behavior is resumed again. This mechanism is very efficient and simple to implement, e.g. an explicit waiting queue is not necessary (it is implemented by the unit interfaces themselves and by the non-deterministic selection strategy);

b) in the *bus-based* architecture, a memory module (or its interface unit) has just *one* input link from all the processors. The memory module firmware interpreter consists in testing such interface and in serving the ready request. Let *J* be the *processor* identifier associated to a received request. If this request contains *indiv* = 1, only the requests from the *J-th* processor are served until a request with *indiv* = 0 is received from processor *J*. All the other possible requests with processor identifier different from *J are received and explicitly buffered* inside the unit. When *indiv* = 0, the first queued request is served; if no queued requests exist, the input interface is tested again. It should be noted that this explicit buffering (a FIFO queue of *N* elements) is necessary, otherwise even the requests from processor *J* could not be received and served (i.e., deadlock situation).

Consider now all the other limited degree interconnection networks studied in Sect. 3. The scheme described in point *a)* cannot be adopted for any of them, i.e. *in any architecture based on a limited degree network* (including the bus), *an explicit buffering mechanism must be provided.*

For example, consider an SMP architecture with a *k*-ary *n*-fly network. The (unique) path from a processor P_i to a memory module M_j is not exclusive of P_i, instead it contains some sub-paths leading to M_j and shared by other processor subsets. From an indivisibility modeling viewpoint, the set of M_j sub-paths is equivalent to a bus: it is impossible for the memory module to listen only the processor which initiated the indivisible access sequence; all the requests must the received, some of them are buffered and one is served. In this kind of network, the queue could also be *decentralized* in the switch nodes belonging to the path, however this decentralized solution does not offer any performance advantage and complicates the switch design. Thus, a *centralized* buffer in each memory unit is provided.

Quite similar considerations apply to the other limited degree networks. The reader is invited to verify this concept for *k*-ary *n*-cubes and (fat) trees.

5.2.3 Memory congestion and fairness

Another important issue is related to the implementation of *busy waiting* in the execution of a *lock(x)* operation.

If a processor finds *x* = "red", a ***retry*** solution consisting of a

- loop of continuous read operations repeated until *x* = "green" is found

introduce notable performance penalties, because:

- the system congestion (shared memory module and interconnection network) is unnecessarily increased,
- the overhead of cache coherence mechanisms might be increased,
- the *unlock* execution itself is delayed by such attempts.

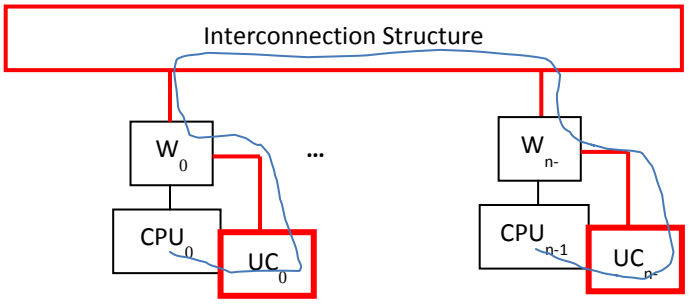
For the sake of clarity, initially the problem is discussed without cache coherence mechanisms. The impact of cache coherence will be discussed in Section 7.

Two main classes of solutions exist:

a) ***periodic retry locking***, (also called ***spin lock***): the read attempts are *spaced out by a constant time interval*, whose value is a function of the lock section duration (e.g. one half of such duration). Theoretically, this solution is affected by the fairness

problem, e.g. no processor is guaranteed to enter a lock section in a finite time (though the probability of finding a "red" semaphore is low).

b) ***explicit notify (fair locking***): a fair solution is also able to minimize the contention effects. The lock semaphore data structure contains a FIFO queue of processor names. If a processor finds *x* = "red", the processor name is inserted into the queue; the processor waits for an *unblocking interprocessor communication* from the processor executing the *unlock* operation:



If the processor executing *unlock* finds at least a processor name P in queue, it sends to P a *notify* firmware message (wake up) through the interprocessor communication structure, and the semaphore is left to the "red" value. Otherwise, the semaphore is assigned the "green" value.

Just one access to the lock semaphore is needed for every lock section, thus minimizing the system contention.

The efficiency, and the fairness, are paid with a slightly greater overhead because of the queue manipulation (queue pointers and current size variables); however no additional accesses to shared memory are done, provided that the semaphore data structure (value, queue) is contained in the same cache block.

Finally, notice that, when lock sections are very short, e.g. just two memory accesses, *lock-unlock* operations can be replaced by set_indiv and *reset_indiv* mechanisms themselves directly. Though being an *unfair* solution, it is *not* affected by the congestion problem.

5.3 Lock – unlock implementations

Let us study the locking mechanism implementation according to retry and explicit notify solutions. The pseudo-code notation

set indiv; S; *reset indiv*;

will be used for indivisible memory access sequences. They can be realized with explicit instructions or with instruction annotations. For example, in D-RISC:

```
LOAD   Rsemaphore, 0, Rtemp_semaphore, set_indiv
IF       < test semaphore condition >
< manipulate the local semaphore value >
STORE  Rsemaphore, 0, Rtemp_semaphore, reset_indiv
```

It should be remarked that locking *per se* is a low-latency mechanism, provided that primitive elementary supports exist at the assembler and firmware level. This is not the

situation when the available locking mechanisms are operating system calls to be executed in kernel space. However, implementing locking on top of an operating system is inefficient both for the excessive overhead (Part 0, Section 6), and for the much higher impact of software lockout (Section 2.4.3). In the following, we assume that processor synchronization is realized by locking mechanisms executed in *user space*, implemented by the primitive mechanisms described above, and that short lock sections are designed.

5.3.1 Period retry locking

A simple boolean lock semaphore is used, initialed at the "true" value.

lock (semlock)::

```
ok = false;
while not ok do
        { set indiv;
          if semlock then
                  { semlock = false; reset indiv;
                    ok = true }
          else { reset indiv;
                  busy waiting by program}
        }
```

unlock (semlock):: semlock = true

In the simplest case, busy waiting can be implemented by a loop of NOPs, or by special instructions having an equivalent semantics. We'll came back on this issue in Section 1.3.3 in relation with problems introduced by the multithreaded architectures.

5.3.2 Explicit notify

The lock semaphore is a *struct* (boolean value, FIFO queue of processor names).

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

lock (semlock)::

```
set indiv;
if semlock.val then
        { semlock.val = false; reset indiv }
else
        { put (my_name, semlock.queue) ; reset indiv;
          wait an I/O interrupt, generated by UC (Communication Unit) when an
          unblocking interprocessor message is received
        }
```

unlock (semlock)::

set indiv;
if empty *(*semlock.queue)
then
{ semlock.val = true; *reset indiv* }
else
{ waiting_processor_name = get (semlock.queue); *reset indiv;*
send to UC an unblocking interprocessor message, with destination = waiting_processor_name
}

Busy waiting implies waiting the interrupt fired by the interprocessor communication. This could be emulated by a NOP loop, which is exited on the interrupt occurrence (the interrupt handler merely causes the program continuation), with some efficiency problems, especially in multithreaded architectures (Section 1.3.3). Otherwise, special instructions should be provided: WAITINT in D-RISC is an example of the required semantics.

5.3.3 Locking and multithreading

The implementation of busy waiting can cause efficiency problems in multithreaded processors. We know that (Section 1.2.; Part 1, Section 22), according to the SMT processor architecture, the firmware resources utilized by concurrent threads may be private of every thread or shared between threads. For example (Part 1, Section 22.4.2), IU and EU_Master are private, while IM, DM and EU functional units are shared. As usually, sharing can introduce contention in resource utilization: for example one running thread can be temporarily delayed by the concurrent utilization of a functional unit and, most important, such delay can be amplified in presence of logical dependencies.

In locking mechanisms, while in principle the busy waiting condition should not imply the utilization of firmware resources, in fact its implementation might occupy processor resources for executing instructions. It has been evaluated that this issue causes meaningful performance degradation in some existing SMT machines, especially if they exploit complex out-of-ordering mechanisms (as remarked in Part 1, Sections 20 and 21, such mechanisms could have an indirect negative impact on other mechanism).

In x86 machines, the `pause` instruction is used in the implementation of periodic retry locking. It introduces a slight delay in the execution of the retry loop and de-pipelines its execution: a part of, but not all, firmware resources is not exploited, thus the problem is not solved completely.

The `monitor` and `mwait` instructions, introduced in recent versions of Intel architectures, represent an alternative solution. The `monitor` instruction supervises a certain memory location for the occurrence of a write activity. The `mwait` instruction places the processor in a special state until a write operation on the location supervised by the monitor occurs or a generic interrupt is received by the processor. In SMT (Hyperthreading in this case), the special state causes the thread to relinquish all firmware resources shared with the other thread running on the same core. This mechanism can be used in the periodic retry locking, and also in the notify locking owing to the interrupt handling of the `mwait` instruction. It has evaluated that the `monitor-mwait` solution has a lower latency than the `pause`-based solution, although currently it must be used in kernel space.

5.4 Memory ordering and memory barriers

By *memory ordering* we mean the order in which memory operations (Load and Store) are performed. Memory ordering might be changed with respect to the order specified in the program (program order) for some reasons:

1. the compiler might change memory ordering as a result of *static optimizations*;
2. the processor firmware interpreter might change memory ordering as a result of *dynamic optimizations*.

Some examples of both reasons have been seen for ILP machines (Part1, Section 19, 20, 21, in particular Section 20.6 about static and dynamic optimizations). For example, even in ILP in-order CPU, or better with FIFO controlled out-of-order, in a sequence of instructions

LOAD address i, a

STORE address j, b

with block(*i)* ≠ *block (j),* if the *block(i)* is not currently in cache and *block(j)* is already in cache, the second instruction can be executed while the first is waiting for the block transfer, because they are independent instructions.

Changing the memory ordering (of course, provided that data dependences are respected) is not a problem for programs running on a single CPU *and* without I/O interactions. However, the problem arises in multiprocessors and concerns *the order in which memory operations executed by a processor on shared variables become visible to the other processors*. In addition to the reasons 1 and 2, the following ones render the problem even more complex in a multiprocessor:

3. memory ordering might be changed *because of nondeterminism* in system behavior at the firmware level, notably in the interconnection structure (arbitration, alterative routing, blocking conditions, and so on), interface units, and memory structures;
4. while Load operations in shared memory have an implicit request-reply implementation, *Store* operations could have an asynchronous semantics (Section 4.1) and overlapped with the successive instructions.

As an example, consider the following fragment of process:

Lock (X):: …; *Store* false into X; …

Critical Section:: …. Load …; Load …; …; *Store* value into S

Unlock (X):: *Store* true into X

It is possible that the sequence *Store S, Store X* is visible to other process in reverse order. Also assuming that the order of *Store S* and *Store X* is not changed at compile-time nor at run-time, possible reasons might be: *i)* S and X are allocated in distinct memory modules M_S and M_X and the writing operation in M_S is momentarily blocked by the arbitration mechanism, while the writing in M_X is executed promptly; *ii)* the firmware writing messages are exchanged in the interconnection network and/or in the interface units. Thus, another processor operating on S in lock state might find X at true while S is not yet updated, thus it might use a non-consistent value of S in the critical section.

For a correct behavior, the memory ordering must be guaranteed. In the example, a sufficient condition is that Store X has to be executed when *all* the previous Store – in general, *all* the earlier memory operations – requested by the same processor have been completed.

Such condition can be achieved in two ways:

1) some particular Store operations are implemented according to an explicit *request-reply* implementation, provided that these particular cases (as in the example) are formally recognized at compile-time or in the code generation: in the previous example *Store X*,
2) an explicit special instruction is provided acting as a ***memory barrier*** that forces to wait the termination of all the earlier memory operations started by the processor. In our example, the Unlock implementation becomes:

   Unlock (X):: *Memory_Barrier*; *Store* true into X

   Similar instructions are also called *Memory Fence*.

In other, words, *a memory barrier is implemented in both cases*:

- in 1) by providing different firmware interpretations of Store instruction, i.e., an asynchronous Store and a synchronous Store,
- in 2) by an explicit special instruction. This way of reasoning has been applied in many other cases (e.g. for indivisible sequences of memory accesses, for cache optimizations, for ILP optimizations).

In the previous example, it can be seen that similar memory ordering problems might arise for the sequence of actions executed by the *lock* and in the critical section, thus the lock should contain a memory barrier implicitly (case 1) or explicitly (case 2).

Correspondingly, at least two main memory consistency strategies can be defined:

- ***Total Store Ordering*** (TSO), which ensures that Store memory operations are visible to the system in the order in which they occurred,
- ***Weak Store Ordering*** (WSO), which does not guarantee such ordering.

Because the problem is not so simple (e.g., consider mixed sequences of Loads and Stores), the definition of TSO must be given more formally:

> *Loads are ordered with respect to earlier Loads. Stores are ordered with respect to earlier Load and Stores.*
>
> Thus, Load can bypass earlier Stores but cannot bypass earlier Loads. Stores cannot bypass earlier Loads or Stores.

TSO is implemented in a primitive way in system with x86 and SPARC assembler machine. Systems with Power assembler machine and others (including Tilera) adopts WSO, thus require explicit Memory Barrier instructions (or proper algorithms).

If all the inconsistency memory problems are confined inside the run-time support of process primitives, including lock-unlock operations, then TSO is sufficient for correctness, provided that *all* the synchronization problems are solved using such primitive. In this case, no Memory Barrier is necessary (if you wish, they are implicitly present in the run-time supports of primitives). Otherwise, if the programmer is willing to express (complex) synchronization problems using different strategies and algorithms, without relying on the given primitives (e.g., the Dekker's algorithm for mutual exclusion), then Memory Barriers are required.

In WSO machines, Memory Barriers are required in almost all synchronization problems, including the design of process run-time support and shared objects applications.

In conclusion, from our point of view, *the memory ordering problem must be explicitly analyzed in the design of basic synchronization mechanisms, notably locking, adopting explicit Memory Barrier instructions for WSO machines.*

Alternatively, for *portability* reasons Memory Barrier instructions should anyway be inserted in the process code, and possibly ignored if the code is executed on TSO machines.

In any implementation, from the *cost model* view point, an important conclusion of this Section is that *an equivalent synchronous writing semantics has to be assumed in the locking latency evaluation.*

Memory Barriers vs Compiler Barriers

Finally, the Memory Barrier problem has not to be confused with the *Compiler Barrier* problem, which arises when modifications of memory ordering, done at compile-time on sequential codes, can affect the correctness, or even can seriously degrade the performance itself. In this case, a Compiler Barrier can be implemented by using proper source language constructs or annotations.

For example, in a program consisting of two consecutive distinct sections:

… Section_1; Section_2; …

we could be interested in avoiding that the ILP optimizing compiler "moves" instructions belonging to Section_2 into Section_1 or vice versa. One simple reason might be that Section_2 contains instructions that are used for profiling or monitoring the behavior of Section_1, thus Section_2 has to be executed after Section_1 even if there is no data dependence. A Compiler Barrier can be inserted between Section_1 and Section_2. (However, what is the efficacy of this barrier when the program is executed on an out-of-order CPU?)

Another, more restricted aspect of this issue is the utilization of the `volatile` keyword in C and C++: it prevents the compiler to reorder Loads and Stores referring `volatile` variables or to omit a Load or Store referring a `volatile` variable.

Compiler Barriers, and sometimes `volatile` variables, try to solve reordering problems in a *sequential* execution environment, however they do not solve the problem of memory ordering in *parallel* programs to be executed on multiprocessor systems.

6. Cache coherence implementation

In this Section we study the cache coherence implementation issues thoroughly, ranging from standard protocols to their architectural implications for snoopy-based and directory-based approaches, as well as the current state-of-the art and trends.

6.1 Cache Coherence Protocols

In all systems using automatic techniques, a proper protocol must exist in order to perform the required actions *atomically*.

A generic cache coherence protocol defines a set of *states* in which each cache block can be. Starting from this set, the protocol describes:

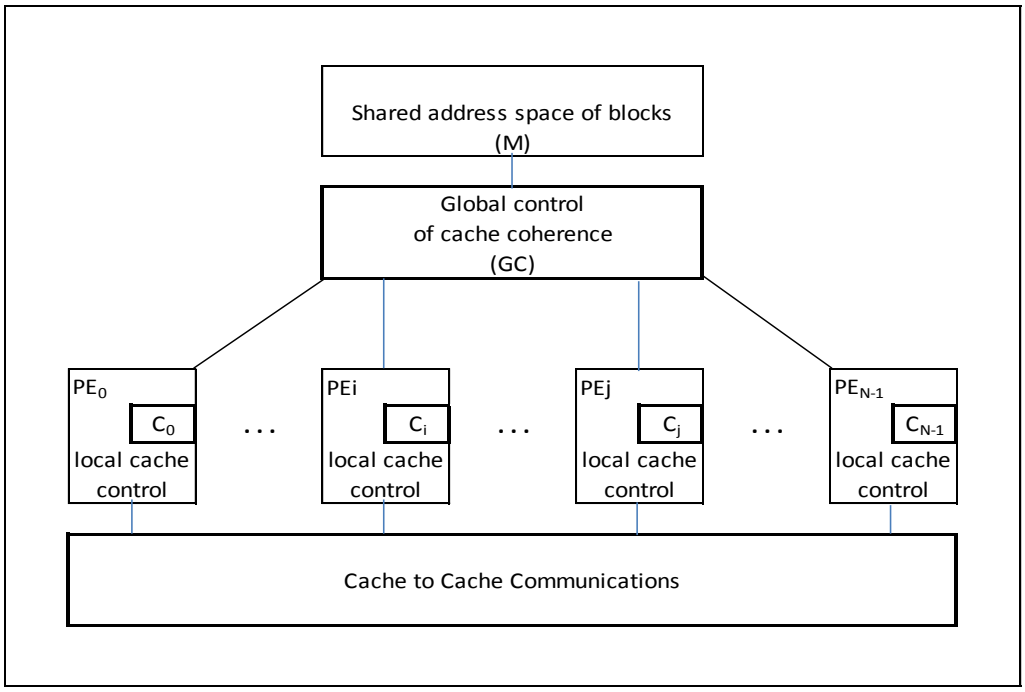
- how the state of a cache block changes when instructions (Load, Store) operating on that block are executed;
- which actions (i.e. invalidation or update communications, memory and/or cache accesses, inter-node communications) have to be performed in order to maintain all the data consistent.

Several protocols have been proposed, all based on invalidation or update mechanisms or on a combination of them. Each solution has tried to reduce the number of actions required to maintain the data consistent.

6.1.1 A general model for invalidation

We start with a general *abstract model of an invalidation-based architecture*: the goal is to explain the most important aspects of a cache coherence protocol, its implementation and its performance. In the next Sections this general model will be characterized to explain a standard protocol and its implications on the multiprocessor architectures.

The model is visualized in the following figure:



For the moment being, it is sufficient to suppose the minimal memory hierarchy, composed of shared main memory (M) and primary cache (C) for each PE.

The generic PE_i contains information about the *state* of all the blocks currently allocated in C_i. In addition to the usual *presence* and the *modification* bit, for each block the possible current existence of copies in other PEs must be locally known (*shared*/*exclusive* bit). Such information can be associated to the other information for cache management: address relocation function, replacement algorithm, reuse and prefetching characterization, and so on. Typically, they are contained in register tables in the cache unit or companion units (e.g., referring to the pipeline CPU of Part 1, Section 19, TAB-CD unit).

For each executed operation (Load, Store), the cache management actions depend on the local state of the referred block. However, the local knowledge is not sufficient for achieving cache coherence. In order to ensure atomicity, a *global knowledge* of the current system-wide situation of cache allocation and block states is needed. Not necessarily the global state contains the union of the information sets of all the local states: a more concise synthesis is sufficient to respect the consistency semantics. In particular, for each memory block: if it is currently allocated and, if so, in which caches, as well the main copy state.

Logically the global state is *centralized* in order to be always updated: in the abstract model this global knowledge is contained in, and managed by, a centralized module (GC). The *actual implementation* of GC will be *centralized*, or *decentralized* (statically partitioned), or *distributed* (dynamically partitioned/replicated), depending on the specific system architecture, as we'll see in the next Sections.

The model includes an *abstract Cache-to-Cache interconnection facility*, through which blocks can be exchanged between PEs according to the protocol actions. In the actual architecture, this facility can be implemented by means of one or more interconnection network *and* the shared main memory M. That is, the simplest way to transmit a block from a cache into a different cache is first to update the block copy in M, and then to read the updated copy from M. However, as we know (see Section 2.1 on CMPs), a direct cache-to-cache (C2C) interconnection can actually be provided too.

This simple model can now be used to understand the basic actions of a cache coherence protocol, distinguishing between reading (Load instruction) or writing (Store instruction) operations on blocks.

1. *Load executed by PE_i*
    a. block *b* is not present in C_i, and has to be transferred into Ci from M or from another C_j possessing a copy. PE_i informs GC of the event. In addition to the necessary global state updating, the intervention of GC solves possible critical races about the updating state of *b* in M or in other caches. Conceptually, GC performs the transfer from M or causes the transfer from Cj;
    b. block *b* is present in C_i. No protocol action is required, because either *b* is present in C_i only or, anyway, all the copies of *b* are consistent.
2. *Store executed by PE_i*
    a. if no copy of *b* is currently allocated, PE_i allocates the block in C_i, modifies it, updates the local state, and informs GC. The block transfer from M might be performed or not according to the write fault handling strategy;
    b. if *b* is currently allocated in C_i only, PE_i modifies it (and updates the local state),
    c. if other copies exist, they must be invalidated, GC informed and M updated.

In Section 2.5.1, two main classes of architectural solutions have been defined for automatic caching:

1. *Snoopy-based*
2. *Directory-based*

In Snoopy-based low parallelism architectures, the abstract GC centralized module is implemented just in a centralized manner. The snoopy bus arbitration logic and the (several) snooping messages strictly correspond to the existence of a centralized unit. The cache block transfers (Cache to Cache communications) are implemented through the same bus.

In Directory-based medium-high parallelism architectures, notably NUMA (or CC-NUMA: Cache-Coherent NUMA), the abstract GC centralized module is decentralized through the global state partitioning. In *memory-based schemes* each PE contains the global state directory entries corresponding to all the blocks in its local memory (*home PE*). In *cache-based schemes*, the information about cached copies is distributed among the copies themselves, and the home PE simply contains a pointer to one cached copy of the block; each cached copy then contains a pointer to the node that has the next cached copy of the block, in a distributed linked list organization.

Snoopy-based and Directory memory-based implementations will be described thoroughly in Sections 6.2 and 6.3.

6.1.2 MESI protocol

We now characterize thoroughly the general model by analysing *MESI*, a standard *invalidation-based* protocol for *write-back* caches, which is one of the most used (considering also the protocols that are MESI's extensions) in off-the-shelf architectures.

As with other cache coherence protocols, the letters of the protocol name identify the possible states in which a cache block can be:

- *Modified* (or *dirty*)
- *Exclusive*
- *Shared*
- *Invalid* (or *not present*)

The state *Invalid* is used in all invalidation-based protocols to represent one of these conditions:

i) the block has been invalidated,

ii) the block is not present, for example when it has been deallocated from the cache.

In an update-based protocol, there is no explicit invalid state, because a block is always kept up-to-date in the cache, so it is always correct to use the data present in the cache if the tag match succeeds. However, a similar state is necessary to represent the condition ii).

The same memory block can be in more than one cache, so the state *Shared* is used in cache C_i when a cache block

a) is present in C_i,

b) it has not been modified after its transfer from the main memory, which is also up-to-date,

c) zero or more other caches may also have an up-to-date (shared) copy of the block.

A cache block is in the *Exclusive* state in cache C_i when situations a) and b) occur and the block is present only in C_i.

A write-back cache requires the state *Modified* to re-write a cache block which has been modified after its transfer from the main memory (not already updated), and there are no other copies in other caches.

Let us suppose that PE_i performs a Load instruction relative to an Invalid, or not present in cache, block of C_i. If no other cache has a copy of the block or their copy is Invalid, PE_i can transfer the block into C_i setting the Exclusive state. Otherwise, if there is a cache C_j that has a copy of the block, PEi can transfer the block into C_i setting the Shared State. Depending on the global state, the copy of the block in main memory can be up-to-date or not. In fact, if the block is Modified in C_j, memory have to be updated. In both cases, the global state also changes in Shared.

The execution of a Load instruction doesn't require any actions if the data is present in cache C_i. In fact, if the block is Shared in C_i, the global state has to be also Shared and all the copies are consistent. If the block is Exclusive or Modified in C_i (the global state is Exclusive), so the copy remains consistent.

Now let us suppose that PE_i performs a Store instruction. In this case, the actions required to maintain the data consistent depend on the global state. In fact, if the global state is Invalid, P_i can simply allocate the block into C_i setting the Modified state. Otherwise, whatever the global state is, all the copies have to be invalidated. Moreover, if the block is Modified in C_j, memory have to be updated. In any case, the state of the block in C_i changes to (or remains) Modified.

A replacement of a block from a cache logically corresponds to change the state to Invalid. If the block being replaced was in Modified state, the replacement transition from Modified to Invalid involves updating the main memory. Instead, no action is taken if the block being replaced was in Shared or Exclusive state.

6.1.3 What happens inside a processing node

Independently from the architectural solution (snoopy-based, directory-based), some functionalities have to be added to PE in order to implement a generic protocol.

Hereafter, we consider a node with a *write-back* data cache for a pipelined PE, according to the scheme and terminology of Part 1, Section 19.1.

When IU requests a Load operation to DM, MMU_D translates the virtual memory address into the main memory addresses, then $TABC_D$ attempts to translate the main memory address into the cache address. $TABC_D$, according to the state of the cache block:

- sends the main memory address for a read request (`READ_REQ`) to the External Memory Interface (MINF), if the state is Invalid. When MINF returns the data obtained with the read reply (`READ_RESP`), the block is stored setting the Shared or Exclusive state depending on the global state of that memory block;
- otherwise, it generates the cache address and delivers the read request to the Data Cache unit CD without changing the state of the block.

When IU requests a Store operation to DM, and MMU_D has translated the virtual address, $TABC_D$, according to the state of the cache block:

- sends the main memory address for a write request (`WRITE_REQ`) to MINF, if the state is Invalid or Shared. This request is necessary to perform invalidations / updates and when the cache policy requires the block transfer from the main memory also for Store on Invalid cache block. A write request could also be sent if the state is Exclusive to maintain the global state updated. When $TABC_D$ receives the write response (`WRITE_RESP`), its sets the Modified state for the block;
- otherwise, it generates the cache address and delivers the write access to CD, changing the state of the block in Modified if the previous state was Exclusive;
- if a write-through flag is set, it send a write-through request (`WT_REQ`) to MINF, without changing the state of the block.

When a cache block is replaced, according to the state of the cache block, DM:

- sends the main memory address and the data for a write back request (`WB_REQ`) to MINF, if the block state is Modified;
- otherwise, no actions have to be performed.

Therefore, MINF forwards to the node interface unit W (Sections 1.1.1, 4.1) the following firmware messages:

`READ_REQ` = (block physical base address, operation)

`WRITE_REQ` = (block physical base address, operation)

`WT_REQ` = (physical address, 1-word, operation)

`WB_REQ` = (block physical base address, σ-word block, operation)

As we said before, a PE can receive requests from other nodes and/or main memory that are necessary to maintain all the copies of a block consistent, so the node must be able to do the necessary actions.

When another node sends a read request for a block, MINF could receive a request if the block is in CD and, according to the state of the cache block, $TABC_D$:

- changes the state in Shared, if the state was Exclusive;
- changes the state in Shared and sends the main memory address and the data for a write back request to the MINF, if the block state was Modified;
- otherwise, no actions have to be performed.

When another node sends a write request for a block, MINF receives a request if the block is in CD and, according to the state of the cache block, $TABC_D$:

- changes the state in Invalid, if the state was Shared or Exclusive;
- otherwise, the state was Modified, so it changes the state in Invalid and sends the main memory address and the data for a write back request to the MINF.

6.2 Implementation of a Snooping protocol

Each node is connected to the Snoopy Bus through the Node Interface unit W, that acts as a cache controller. In particular, W is able to "snoop on" the bus, i.e., it can observe every bus transaction, for example, every read or write on the shared bus. When a processor issues a request to its cache, W takes suitable action, which may include generating bus transactions to access memory.

Coherence is maintained by having all cache controllers "snoop" on the bus and monitor the transactions from other nodes, as illustrated in

Figure **1:**

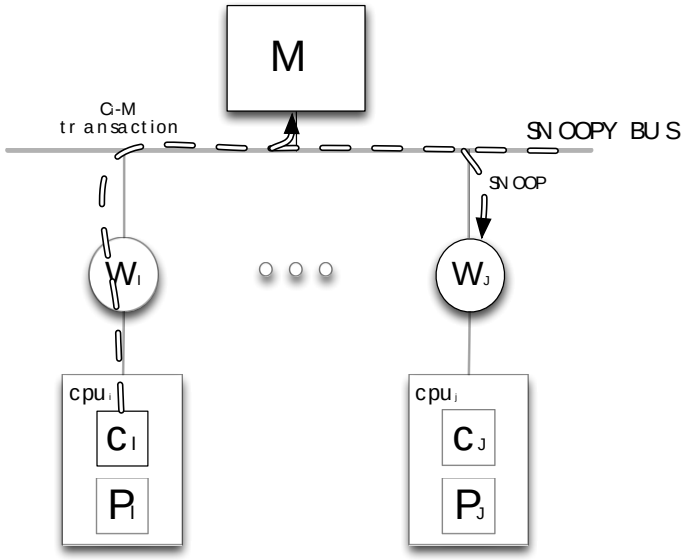


Figure 1 Implementation of a Snoopy-based Protocol

The key properties of a bus that support coherence are the following:

- all transactions that appear on the bus are visible to all cache controllers;
- they are visible to all controllers in the same order, the order in which they appear on the bus.

Let's see now how the unit W interacts with the bus in order to maintain consistency. In particular, suppose that the bus makes available the following transactions:

- `BusRd`, for the read request, which includes the address of the requested data;
- `BusRdEx`, sent from the unit W onto the bus when it receive a write request from the node to a block that is in Invalid or Shared state; it includes the address of the data and the allocation policy;
- `BusWr`, for the write-through request, which includes the address and the data that have to be written;
- `BusWB`, generated on write back request, which includes the address and the cache block that has to be written back in memory.

Therefore, we can describe the behavior of W considering that there are two sets of possible messages that W can receive:

- the memory requests issued by the node (`READ_REQ, WRITE_REQ, WT_REQ, WB_REQ`);
- the information snooped on the bus about bus transactions issued by other cache controllers (`BusRd, BusRdEx, BusWr, BusWB`);

When W snoops on the bus, it may determine whether or not the bus transaction is relevant for the node, that is if it involves a memory block of which it has a copy in its cache.

There are two alternative implementations:

1. W has a copy of a subset of the information maintained in $TABC_D$, in order to do essentially the same tag match that is performed for a request from the processor and determine the state of a cache block;
2. alternatively, each bus transaction is forwarded to DC.

Solution 1 requires that some actions performed inside the node have to be propagated to W in order to maintain the copy of $TABC_D$ updated. On the other side, solution 2 requires a more complex implementation of DC.

Hereafter, let us suppose to adopt solution 1 and analyse the action taken by W when it receives node requests and when it snoops relevant bus transaction.

Table 1 shows, for each node request, the bus transaction issued by W or how the copy of $TABC_D$ changes. Table 2 shows, for each bus transaction snooped by W, the information sent to the node (memory block and requests) and how both $TABC_D$ and its copy change. When W snoops a `BusRd` or a `BusRdEx` transaction and check that the node has a copy of the block requested in modified state, it sends to the DC a `WriteBack` request in order to update the main memory.

NODE REQUEST	BUS TRANSACTION	$TABC_D$
READ_REQ	BusRd	-
WRITE_REQ	BusRdEx	-
Write Propagation	–	E → M
WB_REQ Replacement	BusWB	
Block Replacement Propagation	–	E/S → I
WT_REQ	BusWr	

Table 1 Actions performed by W after node requests

BUS TRANSACTION SNOOPED	DATA AND REQUESTS PROPAGATED	$TABC_D$
READ_RESP	Memory block	I → E/S (*)
WRITE_RESP	Ack or Memory Block	I/S → M
BusRd	_(*)	E → S
	WriteBack	M → S
BusRdEx	–	E/S → I
	WriteBack	M → I
BusWB	–	
BusWr	Word modified	

Table 2 Actions performed by W after bus transactions snooped

(*) When a `BusRd` transaction is snooped, W checks the state of the block requested. If present a `SharedSignal` is sent. In this way, with the relative `READ_RESP` the node that had sent the `READ_REQ` knows if the block received has to be stored in Exclusive or Shared state.

6.3 Implementation of a Directory protocol

We know that, in highly parallel systems, proper limited-degree interconnection structures allow greater scalability than what can be achieved with linear latency networks. This choice is also reflected in the decision to integrate automatic cache coherence mechanisms that scale better than the solutions based on Snoopy bus. In fact, the protocols based on the snooping technique require that each node, including any unit W which acts as a controller of consistency, can communicate with every other node in order to implement the protocol.

To indicate a directory-based distributed architecture, typically with a NUMA organization, which provides a primitive and scalable support for cache coherence, we use the term ***CC-NUMA*** (Cache-Coherent, Non-Uniform Memory Access).

Scalable cache coherence is typically based on the concept of a *directory*. Since the state of a block in the caches can no longer be determined implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain this state explicitly in a place, just called directory. Imagine that each memory block corresponding to a cache block has associated with it a record of the caches that currently contain a copy of the block and the state of the block in those caches. This record is called the *directory entry* for that block, as shown in Figure 2.

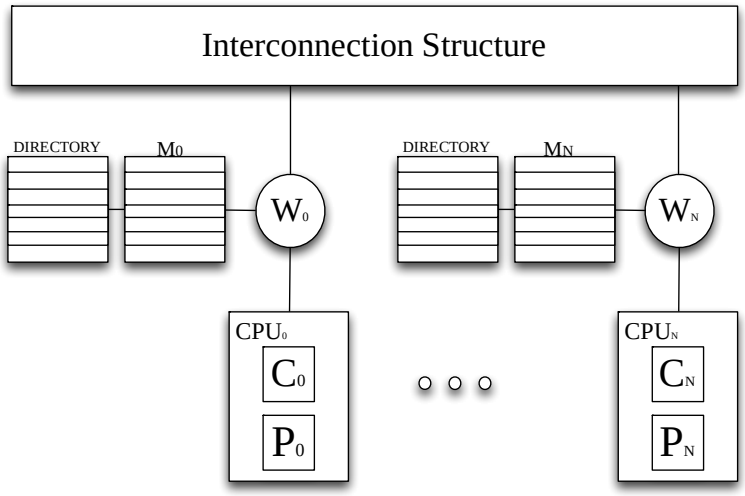


Figure 2 A NUMA organization with directories

Given a protocol, a directory-based system must provide mechanisms for maintain the data consistent; in particular it must perform the following steps when a node request occurs:

1) finding out enough information about the state of the cache block in other caches to determine what action to take;
2) locating those other copies, if needed (e.g., to invalidate them);
3) communicating with the other copies (e.g., obtaining data from them or invalidating or updating them).

In snoopy-based protocol, all these operations are performed by the broadcast and snooping mechanism. In directory-based protocol instead, information about the state of blocks in other caches is found by looking up the directory, while the location of the copies and any communication between the nodes are done through interprocessor communications between nodes, without resorting to broadcast communications. Since communication with cached copies is always done through interprocessor communications, the real differentiation among directory-based approaches is in the first two operations of

cache coherence protocols: 1) finding the source of the directory information, and 2) determining the locations of the relevant copies.

The following definitions are useful to distinguish the processing nodes that interact with one another; for a given cache block:

- ***home***: is the node in whose main local memory the block is allocated;
- ***local*** (or requestor): is the node that issues a request for the block;
- ***dirty***: is the node that has a copy of the block in its cache in Modified state; note that the home node and the dirty node for a block may be the same;
- ***owner***: is the node that currently holds the valid copy of a block and must supply the data when needed; in directory protocols, this is either the home node (when the block is not in dirty state in a cache) or the dirty node.

Depending on where the directory information is maintained, we can distinguish two of the main used directory schemes:

- ***memory-based schemes***, that store the directory information about all cached copies at the home node of the block;
- ***cache-based schemes***, where the information about cached copies is not all contained at the home but is distributed among the copies themselves; the home simply contains a pointer to one cached copy of the block; each cached copy then contains a pointer to the node that has the next cached copy of the block, in a distributed linked list organization.

We now consider the use of a *memory-based scheme*: as shown in Figure 3, the directory information is kept together with the main memory of each node:

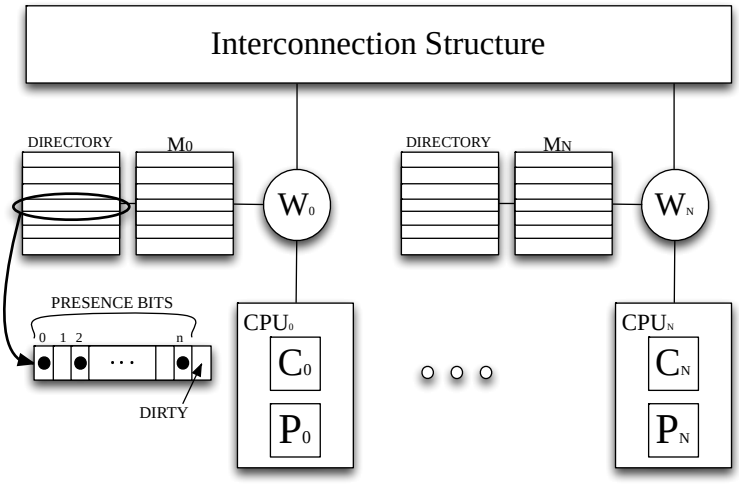


Figure 3 Directory memory-based scheme

A simple organization for the directory information for a block is as a bit vector of *N presence bits*, which indicate for each of the N nodes, whether that node has a cached copy of the block, together with one or more state bits. Let us assume for simplicity that there is only one state bit, called the dirty bit, which indicates if the block is Modified in one of the node caches. Of course, if the dirty bit is TRUE, then only one node (the dirty node) should be caching that block and only that node's presence bit should be TRUE. The directory information for a bock is simply main memory's (as said before, the "global" state) of the cache state of a block in different caches; the directory does not necessarily need to know

the exact state (e.g., MESI) in each cache but only enough information to determine what actions to take.

Let's see now how the unit W interacts with the node and the rest of the system in order to maintain the data consistent.

The unit W receives from the node (through the unit MINF) memory requests (`READ_REQ, WRITE_REQ, WT_REQ, WB_REQ`) and through the most significant bit of the physical address, W is able to distinguish requests to be forwarded to the local memory (the local node is also the home node) or to a remote memory.

For local memory accesses, the request is sent to the Memory Interface Unit I_M, while for remote memory accesses, a specific firmware message is sent through the interconnection structure.

Let us describe the typical firmware messages that can be exchanged between W units:

1. read block request
   `REMOTE_READ =` (header, physical address)
2. read block with invalidation request
   `REMOTE_READ_INV =` (header, physical address)
3. write block request
   `REMOTE_WRITE =` (header, physical address)
4. block value and outcome received from memory
   `RESP_READ =` (header, outcome, σ-word block)
5. block invalidation request
   `REMOTE_INV =` (header, physical address)
6. acknowledgment of block invalidation
   `ACK_INV =` (header, outcome)
7. write back request
   `REMOTE_WB =` (header, physical address, σ-word block)
8. write through request
   `REMOTE_WT =` (header, physical address, 1-word block)
9. write outcome received from memory
   `ACK_WRITE =` (header, outcome)
10. local state update request
    `REMOTE_UPDATE =` (header, physical address)

Now we can see what happens when I_M receives a request from W.

As shown in Figure 4, we can suppose that a directory can be realized with an *associative memory* used to access to the specific directory entry (*dir*) that correspond to the memory block with physical base address *ind*.

Depending on the operation requested (*op*) and the directory entry value, a new memory access request (*newop, sharers, state*) can be generated and forwarded to I_M, otherwise the data needed (*outcome, dataout*) are provided by the memory with the access performed in parallel.

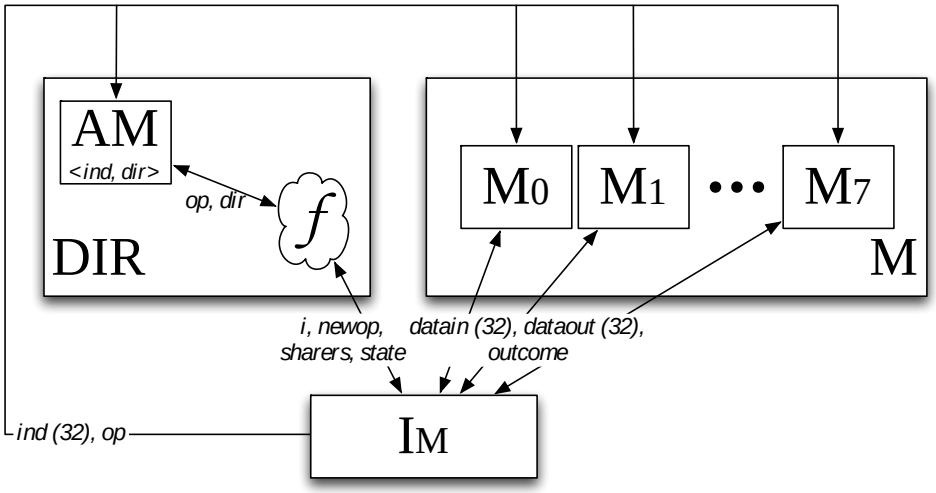


Figure 4 Directory implementation for a memory-based scheme

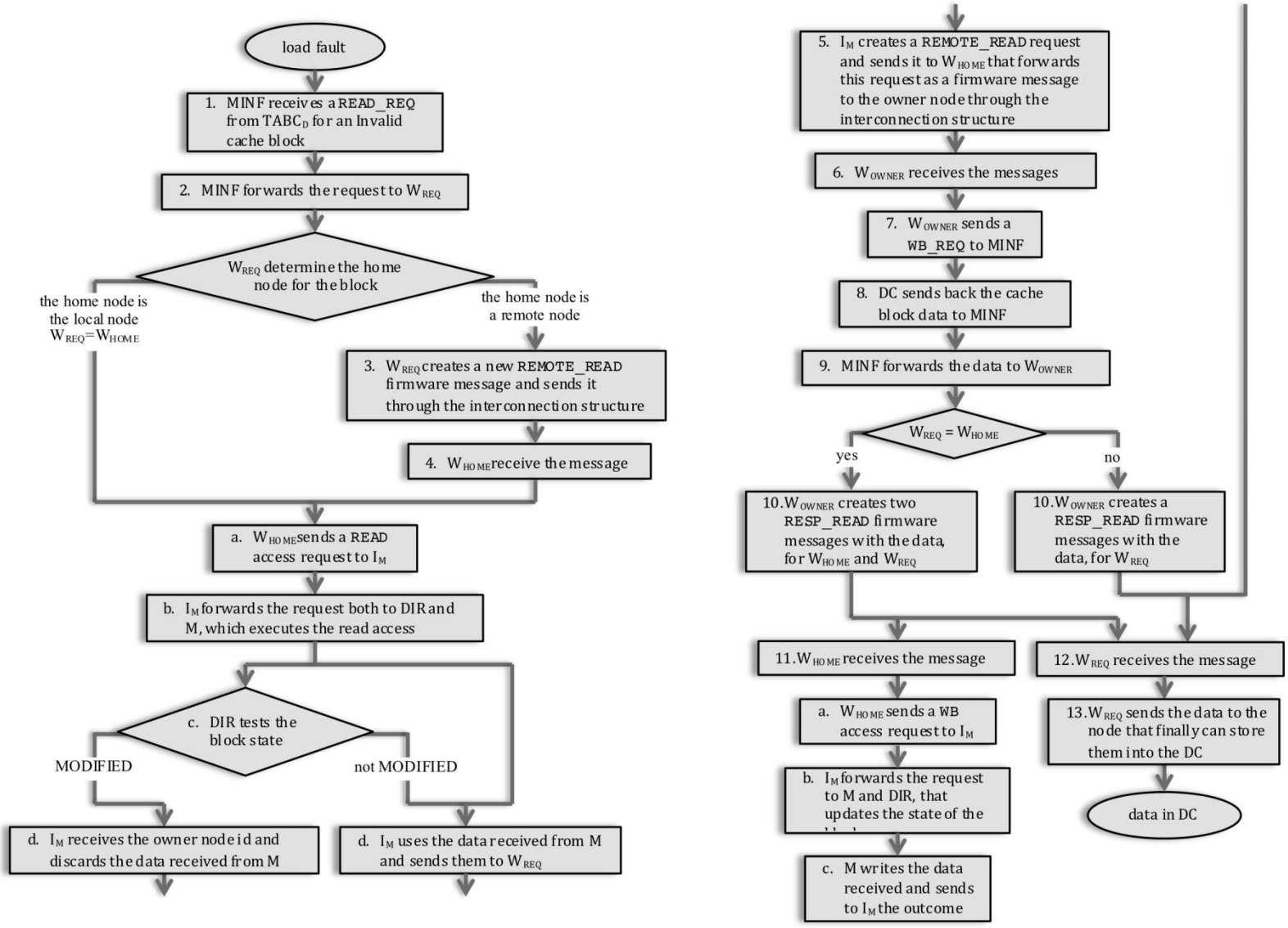
These operations can be managed by a function *f* that is part of the directory unit logic. The following pseudocode describe its behavior:

```
0   switch (op, DIR[ind].M, DIR[ind].num_sharers) {
1     case READ, 0, 0:
2           DIR[ind].sharers[i] = 1;
3           state = E;
4     case READ, 0, 1:
5           newop = REMOTE_UPDATE;
6           sharers = DIR[ind].sharers;
7           DIR[ind].sharers[i] = 1;
8           state = S;
9     case READ, 0, -:
10          DIR[ind].sharers[i] = 1;
11          state = S;
12    case READ, 1, -:
13          newop = REMOTE_READ;
14          sharers = DIR[ind].sharers;
15    case WRITE, 0, 0:
16          DIR[ind].sharers[i] = 1;
17          DIR[ind].M = 1;
18          state = M;
19    case WRITE, 0, -:
20          newop = REMOTE_INV;
21          sharers = DIR[ind].sharers;
22          DIR[ind].sharers = 0;
23          DIR[ind].sharers[i] = 1;
24    case WRITE, 1, -:
25          newop = REMOTE_READ_INV;
26          sharers = DIR[ind].sharers;
27          DIR[ind].sharers[i] = 1;
```

```
28      case WB, … : …
29      case WT, … : …
30      case UPDATE_DIR, … : …
31      …
32  }
```

The following flow chart represents the actions required and the firmware messages exchanged for a read request.

Figure 5 shows a complete example of a read request that involves three different nodes: the *requestor node*, the *home node* and the *owner node*.


load fault
1. MINF receives a READ_REQ from TABC_D for an Invalid cache block
2. MINF forwards the request to W_REQ
W_REQ determine the home node for the block
the home node is the local node W_REQ=W_HOME
the home node is a remote node
3. W_REQ creates a new REMOTE_READ firmware message and sends it through the interconnection structure
4. W_HOME receive the message
a. W_HOME sends a READ access request to I_M
b. I_M forwards the request both to DIR and M, which executes the read access
c. DIR tests the block state
MODIFIED
not MODIFIED
d. I_M receives the owner node id and discards the data received from M
d. I_M uses the data received from M and sends them to W_REQ
5. I_M creates a REMOTE_READ request and sends it to W_HOME that forwards this request as a firmware message to the owner node through the interconnection structure
6. W_OWNER receives the messages
7. W_OWNER sends a WB_REQ to MINF
8. DC sends back the cache block data to MINF
9. MINF forwards the data to W_OWNER
W_REQ = W_HOME
yes
no
10.W_OWNER creates two RESP_READ firmware messages with the data, for W_HOME and W_REQ
10.W_OWNER creates a RESP_READ firmware messages with the data, for W_REQ
11.W_HOME receives the message
12.W_REQ receives the message
a. W_HOME sends a WB access request to I_M
13.W_REQ sends the data to the node that finally can store them into the DC
b. I_M forwards the request to M and DIR, that updates the state of the
data in DC
c. M writes the data received and sends to I_M the outcome

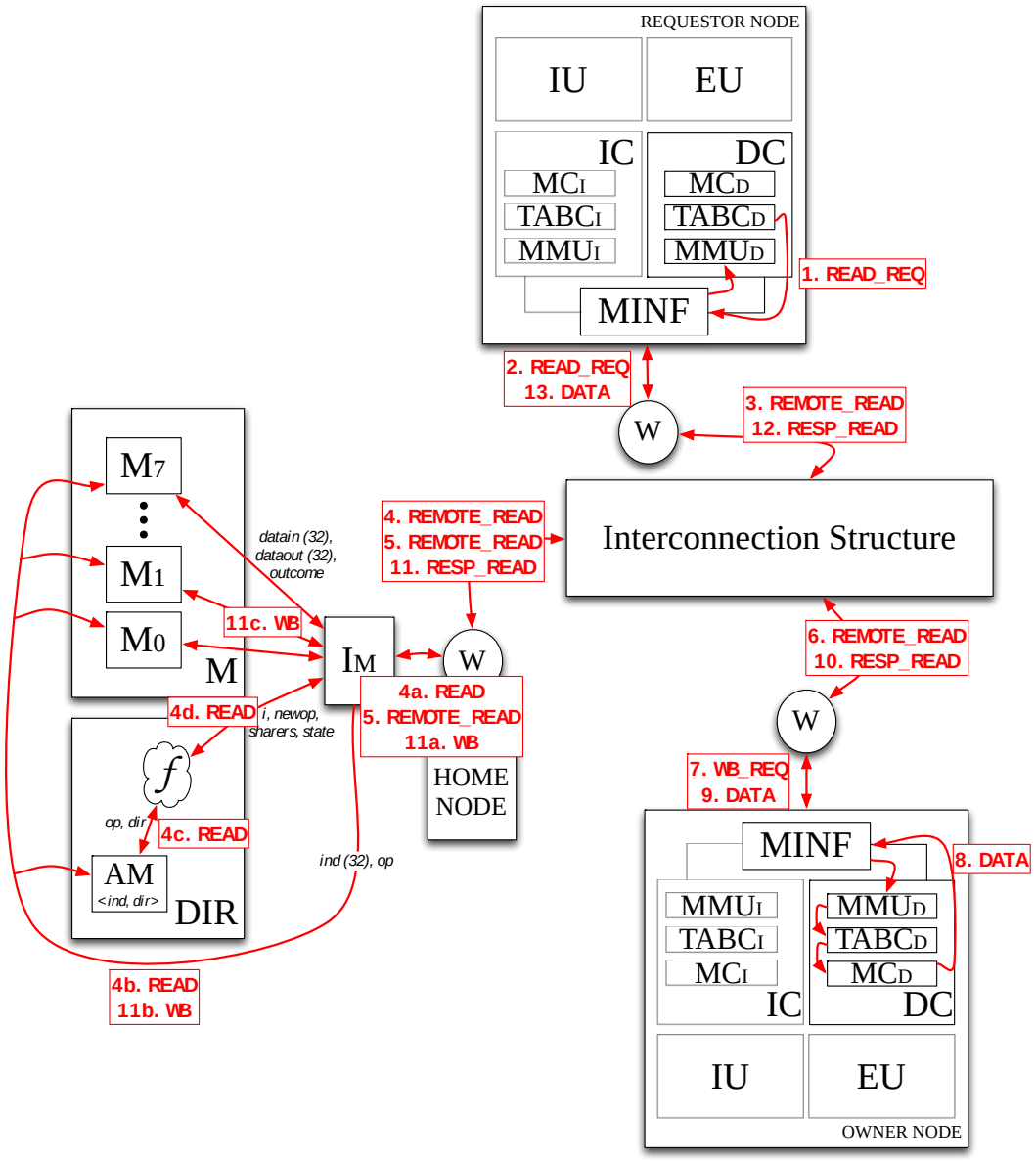


Figure 5 A complete example of the memory accesses and inter-node communications needed to perform a Load instruction relative to a cache block that is homed in a different node and in Modified state in a node different to the Home Node

This example can be useful to reason about some aspects. Obviously, some optimization can be adopted to minimize the number of communications performed.

But more important, what happens when I_M sends a remote request to the owner node and, while it is waiting for the response, it receives another request for the same block? All these actions have to be executed atomically. In this distributed scenario, the solution adopted can be the same used to implement indivisible sequences for lock-unlock atomicity, that is the indivisibility bit (*indiv*). I_M can use an *indiv* bit for each physical memory block.

6.4 Multilevel cache hierarchies

The simple design discussed until now made a simplifying assumption that is not valid on most modern systems: single-level caches. As we know (Section 1) any system uses on-chip secondary caches as well as a third level cache. Multilevel cache hierarchies would seem to complicate coherence since changes made by the processor to the first level cache may not be visible to the second level cache.

Let us consider a two-level hierarchy for concreteness; the extension to the multilevel case is straightforward.

First of all, we need to distinguish two cases:

- the second level cache is ***inclusive***, that is the cache hierarchy implements the following *inclusion property*:

  if a memory block is in L1 cache, then it must also be present in L2 cache; in other words, the contents of L1 cache must be a subset of the contents of L2 cache;

- the second level cache is not inclusive, or ***victim*** cache.

In the latter case, due to the possibility of have blocks in L1 that are not present in L2, it is necessary to handle coherence as in the case of two independent caches.

One obvious way to handle multilevel caches is to have independent cache controller for each level of the cache hierarchy. In snoopy-based solutions, each cache controller is directly connected to the snoopy bus, while in directory-based solutions, each cache controller is connected to MINF (for memory requests and other node requests) and to each other (to maintain all the levels coherent).

When using inclusive caches, designers ensure that they preserve the inclusion property, and in terms of cache coherence this property requires the following:

- all actions taken that are relevant to L1 cache are also relevant to L2 cache, so having only a cache coherence controller for L2 cache is sufficient.
- if the block is in modified state in L1 cache, then it must also be marked modified in L2 cache. In this way, if a request for a block that is in modified state in L1 cache or L2 cache, then it is enough to keep track of this information only for L2 cache.

Therefore, in order to maintain the inclusion property and the cache coherent, three aspects need to be considered:

1. processor references to L1 cache cause it to change state and perform replacements; these need to be handled in a manner that maintains inclusion;
2. requests from other nodes cause L2 cache to change state and flush blocks; these need to be forwarded to the first level;
3. the modified state must be propagated out to L2 cache.

At first glance, it might appear that inclusion would be satisfied automatically since all L1 cache faults go to the L2 cache. The problem is that the implementation of this approach can be complicated by the use of certain techniques typically implemented in cache hierarchies, such as block replacement policies based on the history of access (e.g., LRU replacement policy), the use of more cache at the same level (e.g., first-level cache is divided into instruction cache and data cache), or the use of different block sizes (σ1 and

σ2) between the two levels of hierarchy. In order not to give up the benefits obtained from the use of these techniques, inclusion is maintained explicitly by extending the mechanisms used for propagating coherence events through the cache hierarchy. Whenever a block in L2 cache is replaced, the address of that block is sent to L1 cache, asking it to invalidate or flush (if modified) the corresponding blocks (there can be multiple blocks if $\sigma 2 > \sigma 1$).

Considering requests from other nodes, some, but not all, of these requests relevant to L2 cache are also relevant to L1 cache and must be propagated to it. For example, if a block is invalidated in L2 cache, the invalidation must also be propagated to L1 cache if the data is present in it. Inform L1 cache of all actions that were relevant to L2 cache is the easiest solution but in many cases it is useless. A more attractive solution is for L2 cache to keep extra state (*inclusion bit*) with cache blocks, which records whether the block is also present in L1 cache. It can then suitably filter interventions to L1 cache at the cost of a little extra hardware and complexity.

Finally, on an L1 write, modifications need to be communicated to the L2 cache so it can supply the most recent data if necessary. One solution is to make the L1 cache write-through. The requirement can also be satisfied with write-back L1 caches since it is not necessary that the data in L2 cache be up-to-date but only that L2 cache knows when L1 cache has more recent data. Thus, the state information for L2 cache blocks is augmented so that blocks can be marked "*modified-but-stale*". The block in L2 caches behaves as a modified block for the cache coherence protocol, but data is fetched from L1 cache when it needs to be flushed to other nodes. One simple approach is to set both the modified and invalid bits in L2 cache.

6.5 State-of-the-art and trends

A typical solution used in CMP architectures is to provide a two-level cache coherence protocol hierarchy. In a composition of multicore chips, the caches within a CPU chip (multicore chip) are kept coherent by one coherence protocol and called the *inner protocol*. Coherence across CPU chips is maintained by another, and possibly different, protocol called the *outer protocol*. Each level can also adopt a different architectural solution to implement the protocol.

This hybrid approach is currently used in the composition of general-purpose architectures with low parallelism on chip, for example:

- *AMD Opteron* (Figure 6, see also Section 2.1.2) is a 6-core processor with private L1 cache (instructions and data separated) and a private L2 cache for each core and a shared L3 cache. L2 caches are connected to the L3 with a crossbar and a snoopy-based solution (with multicast communications) is used to keep coherent the data inside the chip. The L3 cache is directly connected to the MINF and maintains the directory of the blocks present in the entire node. The directory is used in order to implement a directory-based solution in a NUMA multiprocessor configuration, where each node is linked to each other in a partial crossbar.

- *Intel Sandy Bridge* (Figure 7, see also Section 2.1.2), like to what happened in its predecessor Intel Nehalem, uses a snoopy-based solution inside the node and it can adopt two alternative solutions to maintain caches coherent in a NUMA multiprocessor configuration. Sandy Bridge is a quad-core processor, where each core has a private (instructions and data separated) L1 cache, a private L2 cache which is not inclusive and an interleaved (4 modules) inclusive L3 cache. Each L1 caches (because of the non-inclusive L2), L2 caches, L3 cache modules and the MINF are connected through a cache controller to a ring interconnection structure. The L3 cache modules maintain for each cache block a set of inclusive bits. Finally, there are two options in the NUMA configuration that use a crossbar interconnection structure: a sort of a snoopy-based solution (with multicast communications between the node) or a directory-based solution.

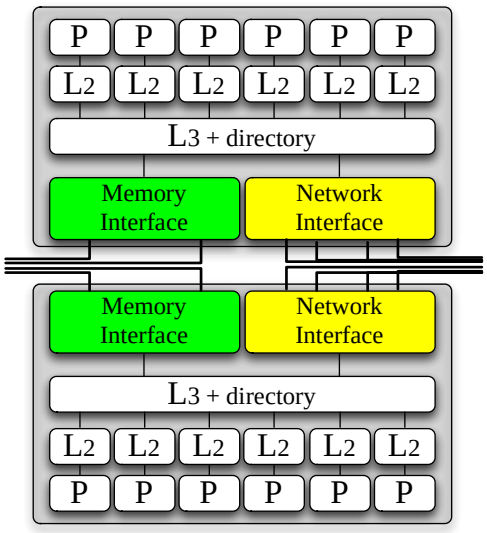


Figure 6 AMD Opteron 6100 with a snoopy-based inner protocol and a directory-based outer protocol

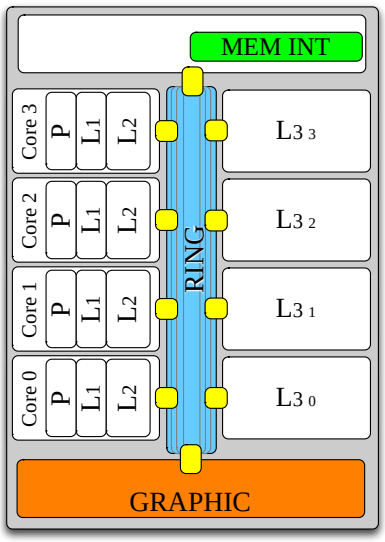


Figure 7 Intel Sandy Bridge with a snoopy-based inner protocol and a two possible outer protocols

But, what happens when the number of core increases?

An interesting example is provided by *Tilera TilePro64* architecture, where a more interesting interconnection structure is adopted.

Tilera TilePro64 (see also Section 2.1.2) consists of 64 identical cores connected to each other and to the four MINFs through a mesh interconnection structure. Each core has a private L1 cache (instruction and data separated) and a private inclusive L2 cache that also maintains directory information.

The directory-based solution adopted is optimized for the architecture to minimize the number of memory access.

As summarized in Figures 8-9, for each cache block a certain core acts as its *home core*, this means that on a L2 fault on core *i*:

- the request goes to the home core
- if the home incurs in a fault itself
    - it sends a read request for the block to the main memory
- otherwise,
    - for a read request, it sends back to the core *i* the data
    - for a write request, it also received from the node *i* the word to modify, and before to sends back an acknowledgment message
        - it eventually sends the necessary invalidation messages to the cores that share the corresponding block
        - waiting the invalidation acknowledgment messages

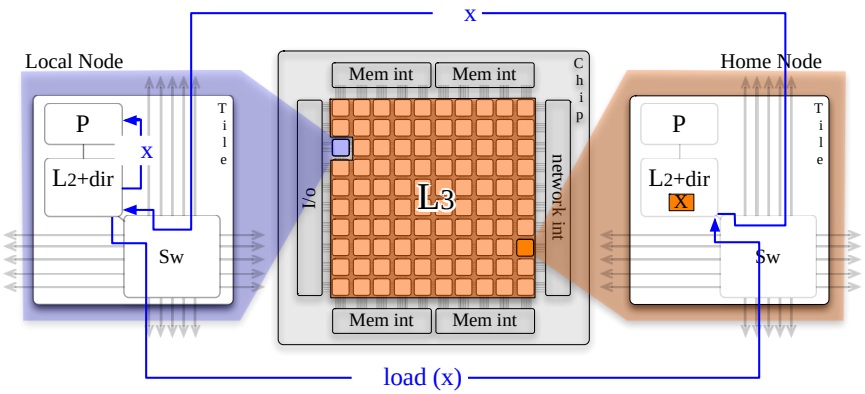


Figure 8 - Actions taken in Tilera TilePro64 on a L2 Load fault

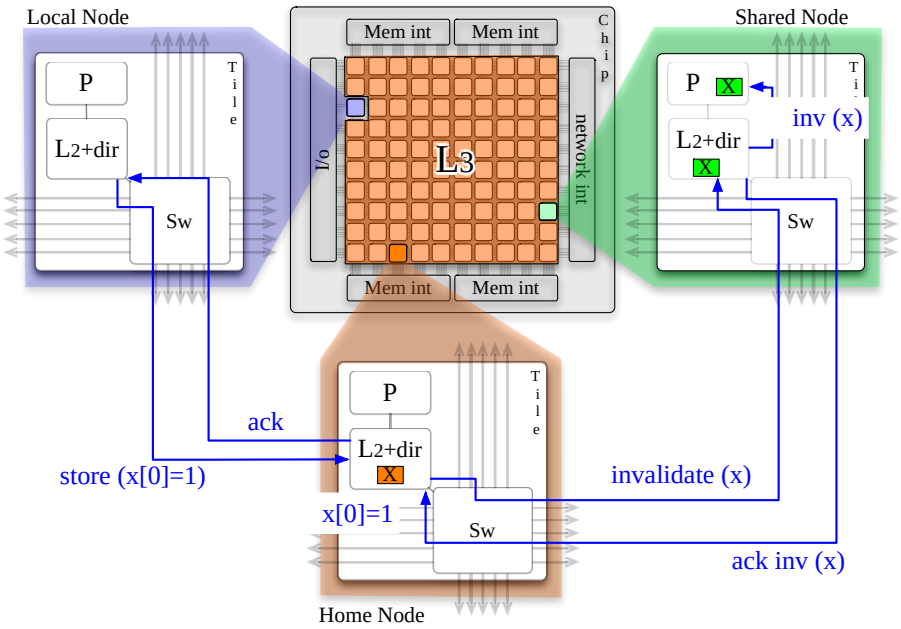


Figure 8 - Actions taken in Tilera TilePro64 after the execution of a Store

6.6 Cost model

The detailed descriptions reported through this Section 6 can be used to evaluate a reliable cost model for memory access latency in presence of automatic cache coherence techniques.

We'll refer to a ***CC-NUMA Directory and memory-based architecture***, for its interest in modern CMP architectures, and because it can be considered a "honest", i.e., clear and predictable, scheme. Latencies involved in Snoopy-based techniques are greater than, or equal to in the most favorable case, the corresponding latencies in Directory-based machines.

According the abstract model of Section 6.1.1, and to the detailed examples of the successive Sections, we can describe the system behavior through the cooperation of *at most* three PEs: *requestor* node, *home* node, *owner* node.

In our analysis we evaluate only the firmware message through the network(s) and the shared-local memory clock cycles. As a first approximation, we neglect the clock cycles spent in the various units for the protocol implementation (W, I_M, and so on), although this is the most favorable case.

In the descriptions, number in parentheses denote the sequence of messages, e.g. request (1) is the first message of a sequence and it is a read or write request.

Block reading

In case of *cache fault* for a Load operation, a request (1) – reply (2) interaction occurs between requestor and home.

If the block is *not Modified in another node*, this sequence is what is needed for the requestor node. Message (2) transfers the block from the local memory of home node into the cache of requestor node. In parallel to (2), a message (3) is sent from home to the possible other shared node to keep its local state updated. Thus, in this case the latency evaluation is equal to a complete memory reading latency:

$$\Omega = \Omega_{read}$$

If the block is in *Modified* state in another node, i.e. the owner, then a message may be equally sent from home (2) in order to synchronize the requestor (it could be avoided). In parallel, the request is forwarded (3) to the owner, then the block is transferred (4) from owner to requestor. The latency can be estimated as:

$$\Omega_{read} + \Omega_{read\text{-}req} = \Omega + \Omega_{read\text{-}req}$$

In this case, the automatic cache coherence management is paid with the overhead of an additional request message through the network from the home to the owner.

If *no cache fault* is generated by the Load operation, no information is exchanged with other PEs.

Block writing

In case of *cache fault*, if the block is not present in other caches, then a request (1) – reply (2) interaction is performed between requestor and home, at least for updating the global state. The block transfer may be provided or not (according to the writing fault strategy),

however with good approximation, the latency can be estimated as the latency of a *synchronous writing* operation

$$\Omega = \Omega_{\text{s-write}} = \Omega_{\text{read}}$$

It is worth noticing that the described behavior is typical *of Total Store Ordering* (TSO) machines (Section 5.4), while in WSO machines the interaction with the shared memory modules might be different from what is described here, and in any case it exploits some nondeterminism (thus, the need for Memory Barrier instructions in synchronization-critical situations).

In presence of fault or not, if block copies are present in other caches, they must be *invalidated*. This is done through messages (3) sent from the home node to such nodes, in parallel with message (2). As in the first case of block reading, the latency of messages (3) can be neglected in the most favorable case. With this first-approximation assumption, the latency is again:

$$\Omega = \Omega_{\text{s-write}} = \Omega_{\text{read}}$$

If no fault is generated and the block is Exclusive, the home node might be informed through a request message (1). If this message is provided, it is overlapped to the successive instructions, thus as a first approximation this latency can be neglected. However, in TSO machines we have again a request (1) – reply (2) behavior *when the Store is inserted in synchronization operations* (notably, lock-unlock). Equivalently, in WSO machines the reply is paid when the Memory Barrier is executed. In any case, *in synchronization-critical operations* the latency is evaluated as Ω.

In conclusion, automatic cache coherence has *no overhead only if no fault is generated in Load operations* and in some non-critical cases in Store operations. In all the other cases, the latency is greater than, or equal to, the latency in ***non-automatic cache coherence***, as summarized in the following table:

	Automatic CC	Non-automatic CC
read, no fault	0	0
read, fault, no Modified in other nodes	Ω	Ω
read, fault, Modified in other nodes	$\Omega + \Omega_{\text{req}}$	Ω
write, no fault, Exclusive	0 or Ω	0
write, fault, Exclusive	Ω	Ω
write, fault, Shared	Ω	Ω

Of course, we know that the automatic technique advantage is the better exploitation of block *reuse*, which could compensate the latency penalties.

7. Interprocess communication: run-time support and cost model

In this Section we apply our knowledge of several multiprocessor architecture issues (processor synchronization, cache coherence, memory latency cost model) to the detailed study of interprocess communication run-time support.

Without losing generality, we refer to the interprocess communication model of Part 0, Section 6 (LC message passing primitives). The starting point will be the uniprocessor version of zero-copy communication run-time support, which will be modified and extended for shared memory multiprocessor architectures.

7.1 Locked version of interprocess communication run-time support

The main modifications to the uniprocessor version of run-time support concern low-level scheduling (in multiprogrammed or exclusive mapping approaches) and processor synchronization of critical sections.

We start with processor synchronization. Let us first study a straightforward modification to the *zero-copy send run-time support*, with respect to Part 0 Section 6, by enclosing critical sections in *lock-unlock* brackets.

The *channel descriptor* is modified with the addition of 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 // *only for multiprogrammed mapping* //

The X semaphore is used for the mutual exclusion of all the critical sections ("*communication sections*"), except the ones used in multiprogrammed low-level scheduling actions. Communications sections are the same for SMP and for NUMA multiprocessors.

In the following, the phase ***WAITING STATE*** will be used each time a process is blocked. It corresponds to *a busy waiting* situation if the ***exclusive mapping*** approach is adopted, while in a ***multiprogrammed*** approach it corresponds to the *process de-scheduling*, i.e. transition into the wait state through context-switching.

The first version of *send* run-time support is shown in Figure 1.

This version can be optimized in order to reduce the size of lock sections, because of the software lockout problem. A first optimization is shown in Figure 2.

The code executed when the validity bit is zero has a very low probability, thus it has negligible impact on the *L* parameter. The most valuable optimization is the following: the communication critical section is of "zero length" if the partner is waiting.

Moreover, another important optimization could be introduced: *the message copy could be executed outside the critical section*, also when the partner is not in waiting state. This optimization is left as an exercise.

```
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);
                          WAITING STATE };
        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);
                          WAITING STATE }
                      else  unlock (X)
```

Figure 1

```
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);
                          WAITING STATE };
        if wait then
                        unlock (X);
        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;
        case wait, buffer_full of
                false, false: unlock (X);
                false, true:    { wait = true;
                                  copy reference to Sender_PCB into CH.PCB_ref ;
                                  unlock (X);
                                  WAITING STATE }
                true, –:        { wait = false;
                                  wake_up partner process (CH.PCB_ref) }
```

Figure 2

7.2 Low-level scheduling

As said, the low-level scheduling, and in particular the *process wake-up* phase, is specific of each architecture.

Moreover, the distinction between ***multiprogrammed mapping*** and ***exclusive mapping*** is crucial.

7.2.1 Multiprogrammed mapping: preemptive wake-up in anonymous processors architectures

In an anonymous processors architecture, the *non-preemptive* wake-up procedure consists merely in inserting the PCB of the waked-up process into the unique shared Ready List, operating *in lock state*. A lock semaphore is associated to the Ready List.

If a *priority-based preemptive scheduling* is provided, the following strategy is realized in order to exploit the anonymity feature:

- let B be a waiting process;
- let A be a process, running on processor P_i, which executes a primitive waking up process B;
- let C be a process, running on processor P_j, which has the minimum priority among all the N running processes;
- if *priority (B)* ≤ *priority (C)*, then A executes the non-preemptive wake-up procedure, putting PCB_B into the Ready List, and the wake-up procedure ends;
- otherwise, if *priority (B)* > *priority (C)*, then a preemption action occurs in processor P_j: C passes into the ready state and B passes directly into the running state. This is done according to one of the two following alternatives:
    - if $i = j$, thus A ≡ C, and the preemption action is executed by A itself on P_i, and the wake-up procedure ends;
    - otherwise, if $i \neq j$, then A sends an interprocessor message to P_j in order to cause the execution of the preemptive action on such processor. The preemption message contains the reference to PCB_B. As usually, the received interprocessor message is transformed by UC_j into an interrupt, which is handled by the running process C: the interrupt handler procedure consists just in the preemption action.

The information about the minimum priority running process must be available to A. For this purpose, a shared *Central Table* is provided. It contains N entries, each one corresponding to a distinct processor and containing the priority of the currently running process and other utility information. This Table can be implemented as an ordered list, and is updated at every context switch.

It is possible that a certain inconsistency occurs in the above procedure: during the interprocessor communication, processor P_j could have executed a context switch, or modified the C priority, so that the priority of the P_j running process is no longer less than the B priority value. Notice that such an inconsistency does not imply an incorrect behavior, but just a temporary lack of scheduling optimization. Processor P_j can

- execute the context switch, without worrying about the lack of optimization, or
- execute the whole preemptive wake-up procedure again, as if it were the waking-up process A. In theory, this can cause a "bouncing" procedure, which is not guaranteed to terminate in a finite time. In practice, after a very limited number of "bounces", the context switch is executed in any case. This solution requires that the interprocessor message includes also the B priority value.

The task of verifying the consistency of interprocessor request, and possibly of accessing the Central Table and starting the "bouncing" procedure, can be delegated to a "smart" UC_j, thus avoiding to interrupt CPU_j unnecessarily.

7.2.2 Multiprogrammed mapping: process wake-up in dedicated processors architectures

In a dedicated processor architecture, the low-level scheduling operations on a process allocated on P_j can be executed only by one of the P_j processes. If a process A, running on P_i, wishes to wake-up a process B, allocated on P_j:

- if $i = j$, then a local wake-up procedure is executed by A, exactly as in a uniprocessor system (with or without preemption);
- otherwise, if $i \neq j$, A sends an interprocessor message to P_j in order to cause the execution of the local wake-up procedure on such processor. The wake-up message contains the reference to PCB_B. As usually, the received interprocessor message is transformed by UC_j into an interrupt, which is handled by the running process C: the interrupt handler procedure consists just in the wake-up procedure.

7.2.3 Exclusive mapping and busy waiting

The busy waiting condition in send-receive code is quite analogous to what occurs in the locking mechanism (which, by definition, is based on busy waiting). Thus the mechanisms for process blocking and unblocking can be implemented:

1. by a periodic retry and a notify mechanism ***via shared memory***,
2. by waiting on a local condition and a notify mechanism ***via interprocessor communication***.

The former solution is adopted in systems which are not properly equipped with efficient interprocessor communication network.

The latter solution is the most interesting one in perspective. As seen in Section 2.1, recent CMP machines adopt a *distinct* on-chip interconnection networks for interprocessor communication, through which very low latency notify mechanisms can be implemented.

7.3 Locking cost model and cache coherence

Let us evaluate the retry and notify locking techniques (Section 5.3) at the light of the presence of cache coherence mechanisms (Section 6). We refer to a general situation of two processors executing a locking critical section CS:

PE_i ::	…	PE_j ::	…
	lock (X);		*lock (X);*
	< CS_i >;		*< CS_j >;*
	unlock (X);		*unlock (X);*
	…		…

In fact, this is the basic computational pattern in the interprocess communication run-time support code, though it is of more general application in many process cooperation mechanisms.

We evaluate the ***locking synchronization latency***, L_{sync}, as the latency of a lock-unlock bracket. The cache coherence cost model, studied in Section 6.6 for *Directory memory-based* automatic techniques *with invalidation*, is exploited.

Moreover, we are in a case in which (Section 5.4) writing operations have basically a *synchronous semantics*, both in a TSO and in a WSO machine. Thus, the latency of Load and Store instructions inside Lock and Unlock operations are paid entirely (Ω), unless reuse is exploited or other particular optimizations can be introduced.

Retry solutions

a) Non-automatic cache coherence

In the following we do not assume the existence of mechanism to emulate the invalidation when required, which is a typical potential optimization in the non-automatic approach.

For the non-automatic approach implementation, special instructions or annotations exist for explicit block de-allocation (*deallocate, not_deallocate*) and explicit block re-writing (*flush*) in shared, as well as for single-word access. See, for example, D-RISC Part 0, Section 3.3.1, *m)*).

The *lock* execution implies the block transfer to read X, and the block rewriting into shared memory at the end. However, an optimization is feasible: X is *not de-allocated*, so that the *unlock* execution finds X in cache. *Unlock* execution causes the modified block re-writing and de-allocation. This is an algorithm dependent optimization. An additional optimization might consist in executing single-word writing, instead of a whole block writing.

Therefore, the locking latency for the non-automatic with retry case is given by:

$$L_{sync} = 3\Omega$$

b) Automatic cache coherence

If the sequences lock-CS-unlock are executed in different times (i.e. the concurrent execution is not attempted), then automatic cache coherence fully exploits *reuse* on X passing from the *lock* to the *unlock* execution, i.e. X remains in cache. If the sequence is inserted in a loop (or invoked several times), in the best case X is found in cache even at the beginning of the next execution of the sequence. In other words, if the concurrent execution of a send-receive pair is a rare event, then automatic cache coherence minimizes the number of accesses to the shared memory.

Let us now suppose that PE_j executes *lock* while PE_i is in the critical section. PE_j reads X and, when modified, X is invalidated, thus deallocated from Ci (it has already been used for the *lock*) and transferred into C_j. Now P_j tests X repeatedly *in cache*, thus without accessing shared memory. Of course, these repeated tests always return the same answer (X is "red"). When P_i executes the *unlock* operation, it invalidates X, which is de-allocated from C_j and transferred into C_i. However, a potential "ping-pong" effect exists, because Pj is still testing X.

Therefore, also in presence of automatic cache coherence with invalidation the retry solution requires a *periodic* retry technique, in order to minimize (theoretically, without eliminating) the ping-pong effect.

Assuming that the lock semaphore X is stored in just one cache block, let

- α the probability that a processor P_i executing *lock(X)* finds block X in the Invalid state, which includes the case in which X is not present in C_i or in any other cache at the beginning of the sequence executed by P_i;
- β the probability that the processor executing *unlock(X)* finds block X in the Invalid state, which is the probability that the critical section is executed concurrently, i.e. that

at least another processor P_j tries to enter the critical section, controlled by the same locking semaphore X, while P_i is still in the critical section.

In conclusion, retry lock latency is equal to 2Ω with probability α and unlock latency is equal to Ω with probability β, thus:

$$L_{sync} = (2\alpha + \beta)\,\Omega$$

In case of full reuse during a sequence and between sequences, $\alpha = \beta = 0$. In the worst case, we have the same latency of non-automatic techniques.

Explicit notify solutions

Although this mechanism implies an additional (reduced) overhead, in the automatic invalidation techniques the ping-pong effect is avoided when concurrency exists between PE_i and PE_j sequences: during *lock*, Pj executes a single test on X in cache, then waits for the interprocessor communication notifying the unblocking event. The locking latency is again:

$$L_{sync} = (2\alpha + \beta)\,\Omega$$

In non-automatic techniques, the optimization consisting in not de-allocating X at the end of *lock* of PE_i is no more possible, because a lock possibly executed by PE_j, while PE_i is still in the critical section, modifies X (the waiting queue). Thus, now the lock latency is given by:

$$L_{sync} = 4\,R_Q$$

and no straightforward optimization is possible. Explicit block re-writing and de-allocation are performed both in *lock* and in *unlock.*

Notice that locking fairness is automatically guaranteed when the cooperating processors are just two (notably, in the locking section of *send-receive* on a symmetric channel): in these cases the periodic retry solution is adopted with non-automatic cache coherence.

7.4 Cost model of interprocess communication

The interprocess communication latency will be approximately expressed in terms of memory access latencies only (Ω), neglecting the processing time of the instructions not related to memory accesses.

Let us refer to the zero-copy communication on symmetric, asynchronous, deterministic channels (Section 7.1).

The *send* latency can be evaluated as the sum of the following latencies:

i) latency of locking synchronization, L_{sync},
ii) latency of additional transfers of channel descriptor (CH) cache blocks, L_{ch},
iii) latency of message copy into the target variable, L_{copy}:

$$L_{com}(msg\_l) = T_{send}\,(msg\_l) = L_{sync} + L_{ch} + L_{copy}$$

Being a one-to-one cooperation, we can adopt the *periodic retry locking* technique.

A feasible channel descriptor structure is the following

First CH block:

lock semaphore: 1 word

wait boolean: 1 word

message length: 1 word

buffer insertion index: 1 word

buffer extraction index: 1 word

buffer current size: 1 word

PCB reference: 1 word // not used in exclusive mapping //

Second CH block:

buffer: array of references to target variables (+ validity bits): σ words

We can assume that the asynchrony degree is numerically less than or equal to σ. Of course, for greater asynchrony degrees, additional cache blocks are needed.

As seen in Section 7.3,

L_{sync} = 3Ω in the non-automatic approach

$L_{sync} = (2\alpha + \beta)\Omega$ in the automatic approach

With good approximation, in general we can assume:

$$L_{sync} = 3\Omega$$

In some way, the worst-case analysis of the automatic approach partially compensates the best-case evaluation of the memory access latency itself, as discussed in Section 6.6 (notably, in neglecting the indirect effect of protocol message exchanged in parallel, as well as neglecting the overhead of the various PE units to implement the protocol).

With periodic retry locking, the first CH block has been transferred into the cache just for acquiring the lock semaphore X. Thus, only the read + write latency of the second block is paid during the channel descriptor manipulation:

$$T_{ch} = 2\Omega$$

in the most frequent case of asynchrony degree less than or equal to σ.

The message copy consists in a loop of message block reading operations (from the local memory in a NUMA machine) and writing operations into shared remote memory. In the most advanced implementations, C2C can be exploited intensively to reduce the latency. In any case, a *pipeline* implementation is able to hide the message block reading latency: it can be realized through the *block prefetching* option, if available, or by a proper realization of the node interface unit (W). This technique optimizes the latency without relying on TSO features or Memory Barriers. Thus, T_{copy} reduces to the synchronous writing latency only:

$$T_{copy}(L) = \left\lceil \frac{L}{\sigma} \right\rceil \Omega$$

In conclusion:

$$L_{com}(msg\_l) = T_{send}(msg\_l) = 5\Omega + \left\lceil \frac{L}{\sigma} \right\rceil \Omega$$

Thus (for *msg_l* multiple of σ):

$$T_{setup} = 5\Omega \qquad T_{transm} = \frac{\Omega}{\sigma}$$

Using the results of Section 4 about the under-load memory access latency, with Ω values of the order of 10^2 clock cycles, we have the typical values of the communication cost model parameters:

$$T_{setup} = a\,10^2\,\tau \qquad\qquad T_{transm} = b\,10^1\,\tau$$

with a and b constants depending on the PE architecture, the number of PEs, the interconnection network, and the under-load contention parameters, notably p and T_p.

For example (see Section 2.1):

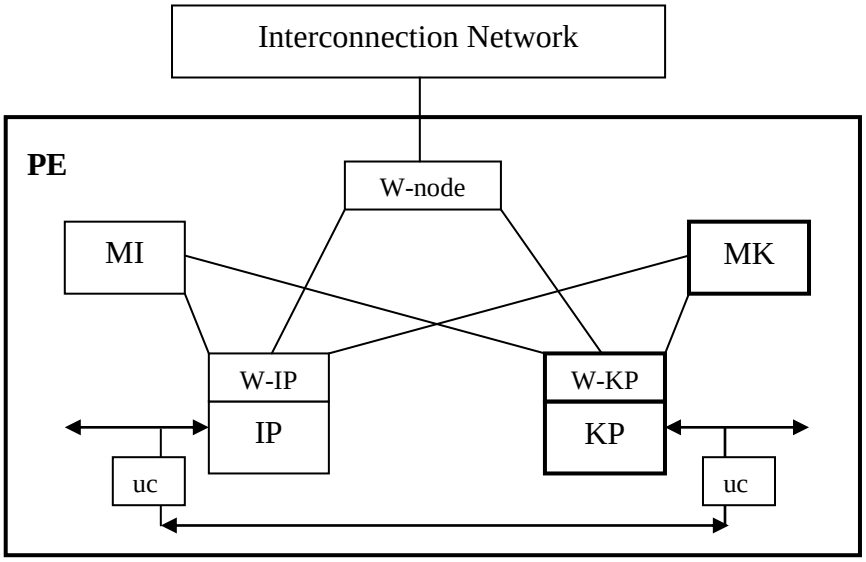
- for CMP chips T_{setup} is in the range (200 - 600) τ and T_{transm} in the range (20 - 60) τ;
- for multi-chip configurations, these values are further increased by a factor of 10, because of the greater values of interconnection network latency, t_{hop}, and external memory access time.

7.5 Communication processor

We know that, in some parallel program patterns, the interprocess communication latency might be *overlapped* to the calculation time. The needed architectural support consists in a *communication processor* (KP) associated to the *main processor* (IP) of the processing node. *KP is dedicated, or specialized, to the execution of the run-time support functionalities*, and in particular the *send* primitive. The principle is the following:

- when IP has to execute a *send* primitive on an *asynchronous* channel, it delegates this task to KP and continues the execution;
- the *receive* primitive is executed by IP entirely.

The architectural scheme of the processing node is based on a shared memory cooperation between IP and KP. As shown in the following figure, this can be achieved by realizing the node as a small multiprocessor with dedicated processors, as shown in the next figure, where MI and MK are local memories (or secondary/tertiary caches) of IP and KP respectively:



Equivalently, KP may be an *input-output coprocessor*, sharing memory with IP through DMA and Memory mapped I/O. In particular, this kind of implementation is adopted when the communication processor is provided "inside" the interconnection network box. That is, the externally available links of the interconnection network (i.e. the links to be connected to the processing nodes) are I/O links.

The parameter passing from IP to KP is done *by reference* (by *capability*) through shared memory, without additional copies. IP prepares a data structure *S* containing the channel identifier and the message reference. The reference to *S* is transmitted to KP via I/O (in the Figure; via the pair of communication units *uc*-*uc*). KP is a "daemon", that, once activated by the interrupt from IP, acquires the parameters into its addressing space, with very low overhead, and starts the *send* execution.

In this example, we have a clear proof of the capability role in the "shared pointer problem" (Part 0, Sect. 4.4). The KP process must share any possible message, target variable and PCB of communicating processes. A static allocation of such objects in the KP addressing space is extremely inefficient or practically impossible. The dynamic allocation allowed by the capability addressing solves the problem in an elegant and efficient manner.

Though it is possible to realize KP with the same architecture of IP, the trend is to design a much simpler CPU-KP, possibly specialized at the firmware level, as discussed in the multicore solutions of Sect. 2.1.2.

Let us know analyze the *send* implementation in more depth.

The *send* semantics is: 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 the *send* execution *entirely* to KP, then some complications are introduced in the *send* implementation because of the management of the waiting state of the sender process, and if we wish to achieve the objective of delegating *more than one* asynchronous communications (on the same channel or on different channels) overlapped to the same calculation section, e.g.

calculation ...; send ...; send ...; send ...; ...

A simpler and efficient solution, able to achieve this objective, consists in the following principle:

- *IP itself verifies the saturation of channel asynchrony*;
- IP delegates to KP the *send* continuation;
- if (*buffer_size* = *k*) IP sets the *wait* boolean variable in the channel descriptor, and suspends the sender process in busy waiting condition or by context-switching (if *k* denotes the asynchrony degree, *k + 1* is the number of buffer elements);

In the zero-copy communication, IP controls the *validity bit* too, if provided.

The channel descriptor is *locked* by IP and *unlocked* by KP.

This scheme eliminates the complexity of the general solution (full delegation to KP), at the expense of *an initial phase executed by IP itself, thus not overlapped to the internal calculation*. In the interprocess communication cost model, the latency of this phase must be included in the T_{calc} parameter. In practice, the overlapping is applied to the channel descriptor buffer manipulation, to the message copy, and to the low-level scheduling actions on the destination process.

7.6 Advanced solutions to interprocess communication

As introduced in Section 2.1, the existence of *distinct interprocessor communication networks* can be very useful for the implementation of interprocess communication (and for exclusive mapping low-level scheduling, as seen in Section 7.2.3).

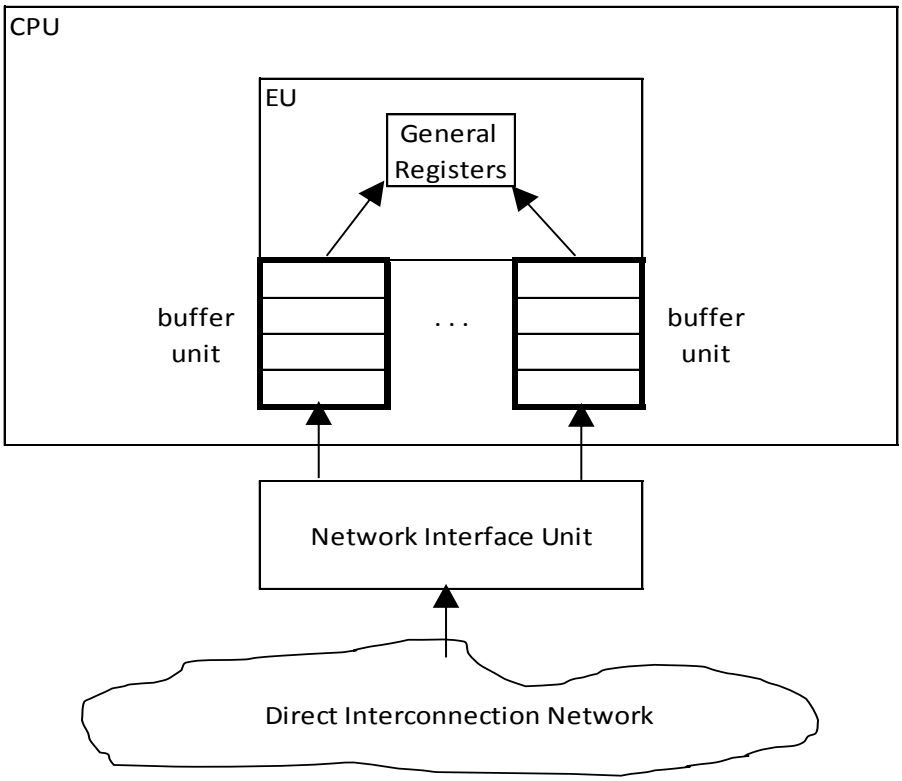
a) As a first, straightforward example, consider a multiprocessor architecture with two distinct interconnection networks for accessing the shared memory, one of which (*ComNet*) is dedicated to the interprocess communication run-time support only. In presence of Communication Processor, ComNet interconnects the Communication Processor subsystem.

In this case, no special solution is adopted and the double network is exploited just to increase the bandwidth and to reduce the memory contention between run-time support accesses (with a burst distribution of traffic) and "normal" accesses (with a more uniform distribution).

b) A notable improvement consists in optimizing the *message copy* latency by exploiting the *cache-to-cache* interconnection network, if it is visible and accessible to the programmer. In this case, the message is copied directly from the cache of the source PE (the node in which the source process is allocated) into the cache of the destination PE (the node in which the source process is allocated).

If the *exclusive mapping* approach is adopted, the copy can be done even at the *first level* cache, otherwise the *second level* cache has to be used.

c) As exemplified by Tilera UDN, or by some emerging network processors (Section 2.1), a distinct interconnection network can be used to implement a direct form of message passing at the firmware level. Each PE has a set of firmware queues connected directly to processor registers, notably to the general registers of the Execution Unit, as shown in the following figure:



The behavior is very similar to the firmware implementation of asynchronous communications by means of buffer units: for example, see Part 0, Section 2.4.3.

Of course, proper assembler instructions must be provided corresponding to primitive send and receive operations *between processors*.

Provided that the *exclusive mapping* approach is adopted, all the shared memory and cache hierarchy levels are skipped: with some constraints on message types, the interprocess run-time support can exploit this interprocessor communication facility to sharply reduce the message copy latency.

At the same time, this is also a mechanism for busy waiting unblocking.

These advanced solutions tend to reduce the various sources of overhead in interprocess communication run-time support, in particular locking and cache coherence. In the ideal situation, especially with the exclusive mapping approach, the interprocess communication latency should be spent mainly in the message copy phase, avoiding shared memory synchronizations and exploiting shared memory for fast cache-to-cache copies.

8. Distributed memory multicomputers

As introduced in Section 1, in this class of MIMD architectures no memory sharing is physically possible among processes allocated onto distinct processing nodes. That is, the result of the translation of a logical address, generated by any processor running on a node PE_i, *cannot* be a physical address of the main memory belonging to a distinct node PE_j. *Memory sharing is prevented at the firmware level*, though it could be emulated at a higher level. The only primitive architectural mechanism for node cooperation is the communication by value, that is the cooperation via *input-output* mechanisms and interface units. Interprocess communication is implemented on top of such mechanism.

8.1 Inter-node communication latency in multicomputers

The multiprocessor evaluation of interconnection networks could be extended to multicomputers too, at least from a qualitative viewpoint.

In particular, since multicomputers are *dedicated processors* architectures,

- the effect of *low-p* mappings (Section 4.4) is quite similar to what has been discussed about NUMA multiprocessors.

Analogous considerations can be done about the impact of T_p and δ parameters:

- coarse grain computations reduce the contention effects,
- for *low-p* and coarse grain computations, logarithmic networks, like fat tree, are basically distance insensitive.

As said, the same type of interconnection networks can be used in both a multiprocessor and a multicomputer: there are few basic differences in the type of traffic handled in the two architectures, as well as in the performance requirements placed on them. It is worth remembering the distinction between messages at different levels: firmware messages and inter-process messages (in a message passing cooperation model). The run-time support of inter-process communications makes use of some firmware messages crossing the interconnection network:

a) in a multiprocessor, they are mainly *remote memory access requests and replies*, and possibly explicit inter-processor communications for scheduling and processor synchronization purposes. Thus, the traffic handled in a shared memory multiprocessor consists of *relatively small blocks of data in a single packet*, i.e. typically a cache block, or few words for access requests and/or explicit inter-processor communications;

b) in a multicomputer, generally one of the firmware messages, organized in *more packets*, is used for implementing the transmission of *the true inter-process message*, enriched by an header and some other utility information of the run-time support. Moreover, there are further firmware messages for synchronization and scheduling purposes. Therefore, *the length of a firmware message generated in a multicomputer can vary over a wide range* and, on the average, is much larger than the length in a multiprocessor.

Also, the interconnection network in a multiprocessor must have low latency to keep the memory access time reasonable, while multicomputers can generally tolerate higher network latencies.

In this Section the interconnection network analysis is completed with experimental and simulation studies for low dimension cubes and generalized fat trees: they are due to *Fabrizio Petrini*.

It is worth remarking that some multicomputer architectures (MPP) exploit the firmware-primitive network protocols, in the same way of multiprocessors, while others (simple clusters) exploit IP-like protocols. In the latter case, the latency is largely dominated by the IP protocol itself (notably, an overhead of about 10^5 instructions).

8.1.1 Performance measures for low dimension *k*-ary *n*-cubes

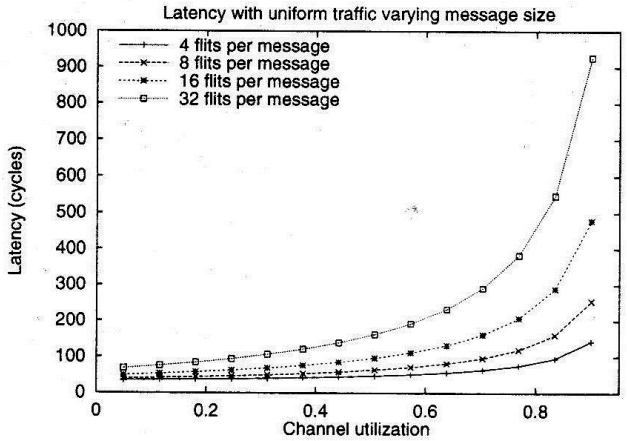
Assuming *uniform traffic*, the under-load latency has been estimated by Agarwal solving a proper queuing model:

$$L = \left[1 + \frac{\rho\, B\, (k_d - 1)}{(1 - \rho)^{k_d^2}}\, (1 + \frac{1}{n})\right] n\, k_d + B$$

where B is the packet size in flits (see wormhole flow control), and

$$k_d = \frac{k - 1}{2}$$

is the average distance with unidirectional channels and toroidal connections. This model is reported in the following figure, which respects the qualitative shape at the end of Sect. 3.6.2:

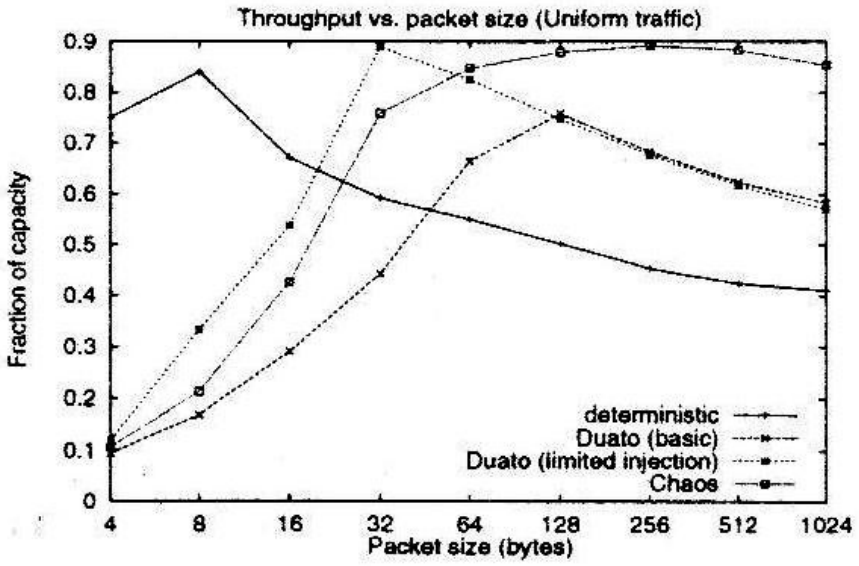


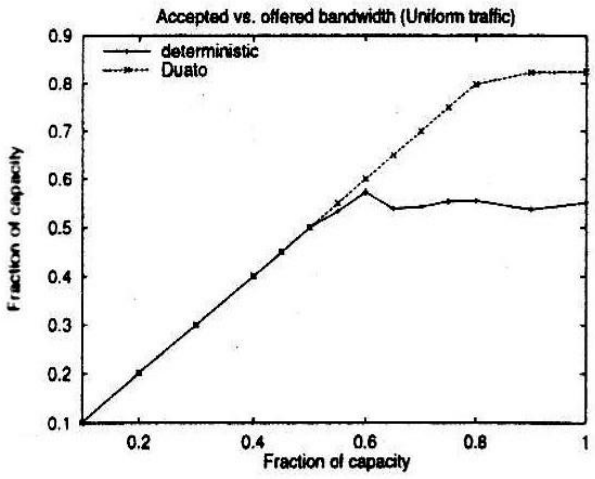
Message latency varying the channel utilization ρ, for several packet sizes in flits. The graph refers to a 32-ary 2-cube with dimension order routing, under uniform traffic.

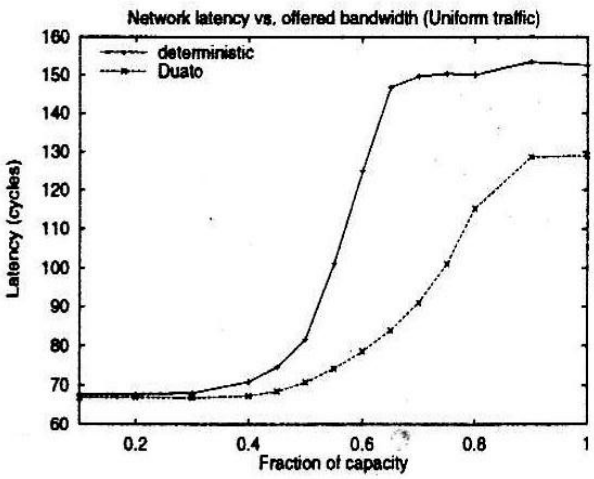
A meaningful result is that better latency is achieved with smaller packets: that is, *small packets* utilize the network closer to its theoretical bandwidth, without significant degradation in latency, and the average latency becomes *less sensitive to congestion*. This result can be applied to shared memory multiprocessors too.

More robust models must take into account higher network loads or non-uniform traffic, for example due to collective communications and other typical communication patterns of parallel paradigms.

The following figures have been derived by Fabrizio Petrini through simulation for a *16-ary 2-cube* under uniform traffic.







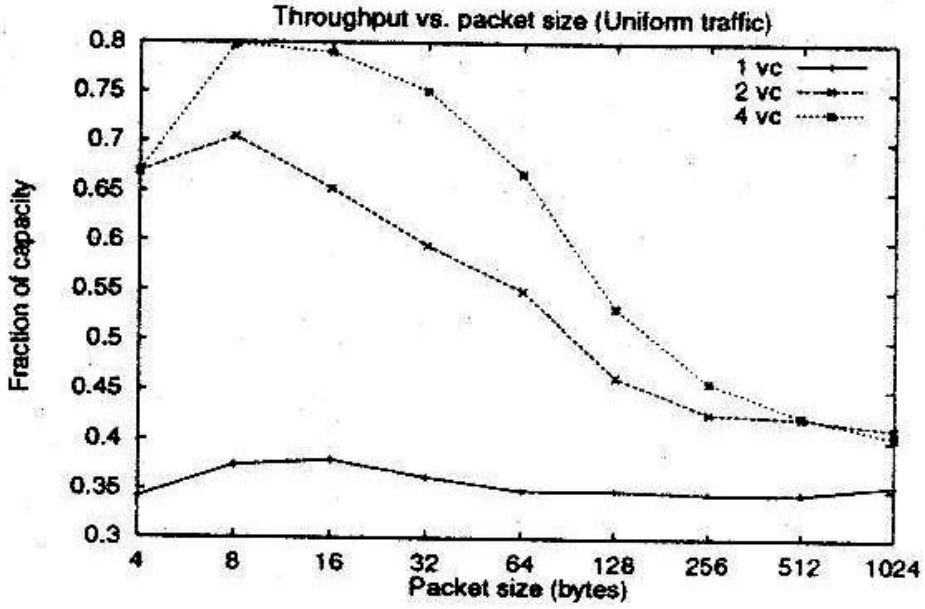
The first figure shows the bandwidth as a function of the packet size, the second the comparison of deterministic *vs* adaptive routing strategies (the adaptive routing algorithm is due to Duato):

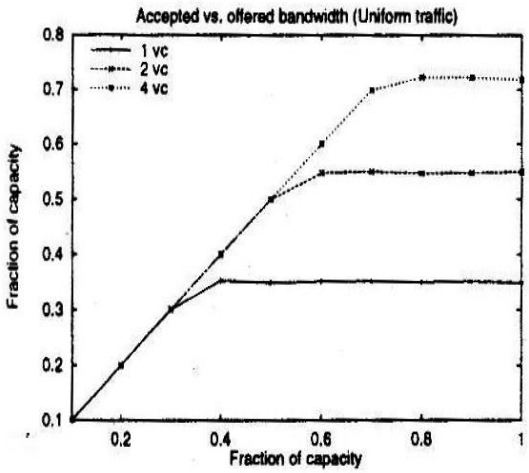
Other simulation studies have been done for different traffic models.

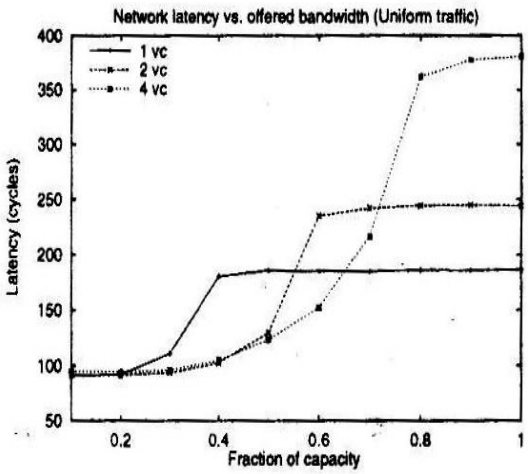
8.1.2 Performance measures for Generalized Fat Trees

The following figures show some simulation results on Generalized Fat Trees. The assumptions are:

1. uniform traffic,
2. minimal adaptive routing,
3. different number of logical communication channels inside the switch nodes,
4. Generalized Fat Tree based on a 4-ary 4-fly network.







The fourth assumption allows us to compare this network to the *k*-ary *n*-cube network evaluated in Sect. 3.6.2, with the same number of processing nodes ($N = 16^2 = 4^4 = 256$). It can be seen that, *under the uniform traffic assumption*, cube performances are better than, or comparable to, fat trees.

However, it can be seen that, for more *complex communication patterns*, for example, *collective communications*, the fate tree structures outperforms the cubes. This is due to the fact that fat trees are *much less sensitive to distance* than cubes. For this reason, many current commercial networks, including Myrinet and Infiniband, are basically structured as fat trees.

A meaningful benchmark for complex communication patterns is represented by the so-called *complement traffic*, in which

- all nodes communicate simultaneously,
- a node with binary identifier $a_0\ a_1\ \ldots\ a_{n-1}$ sends messages to the node whose binary identifier is $\overline{a_0}\ \overline{a_1}\ \ldots\ \overline{a_{n-1}}$.

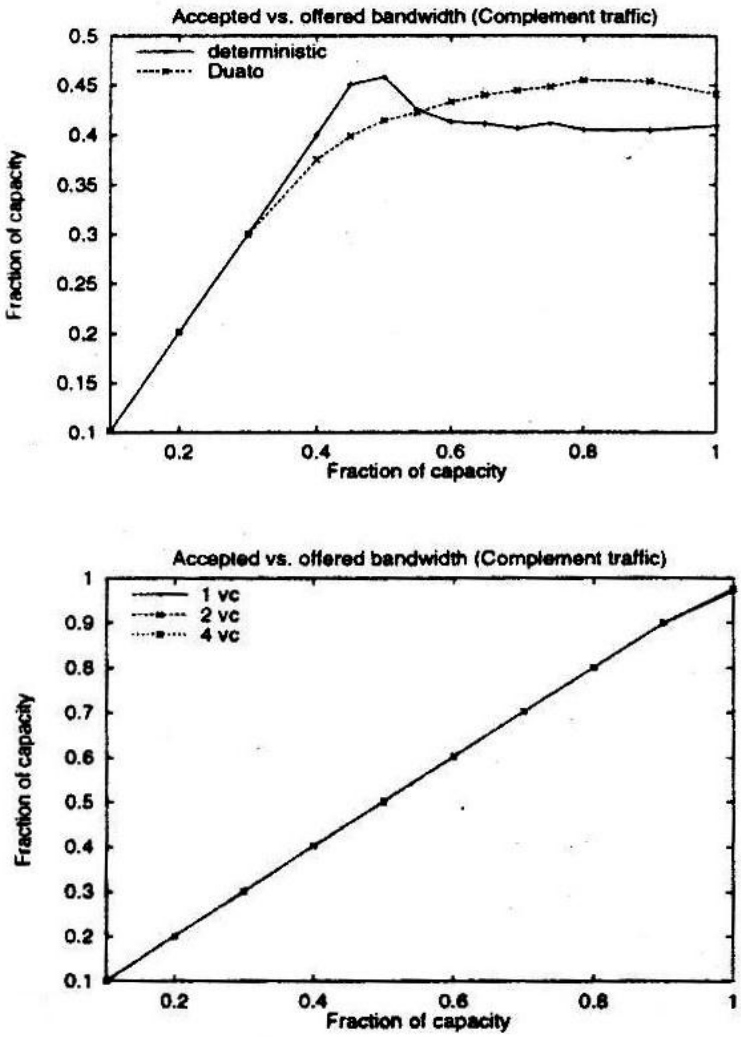
In other words, in a complement traffic pattern all the messages cross the network bisection and traverse a path whose length is the network diameter. Thus, it is a latency-sensitive pattern that stresses the network capability of dealing with a significant amount of possible conflicts.

The following figure shows the relative bandwidths of

a) 16-ary 2-cube,

b) Generalized Fat Tree based on a 4-ary 4-fly network,

under complement traffic, using comparable technologies and minimal adaptive routing:

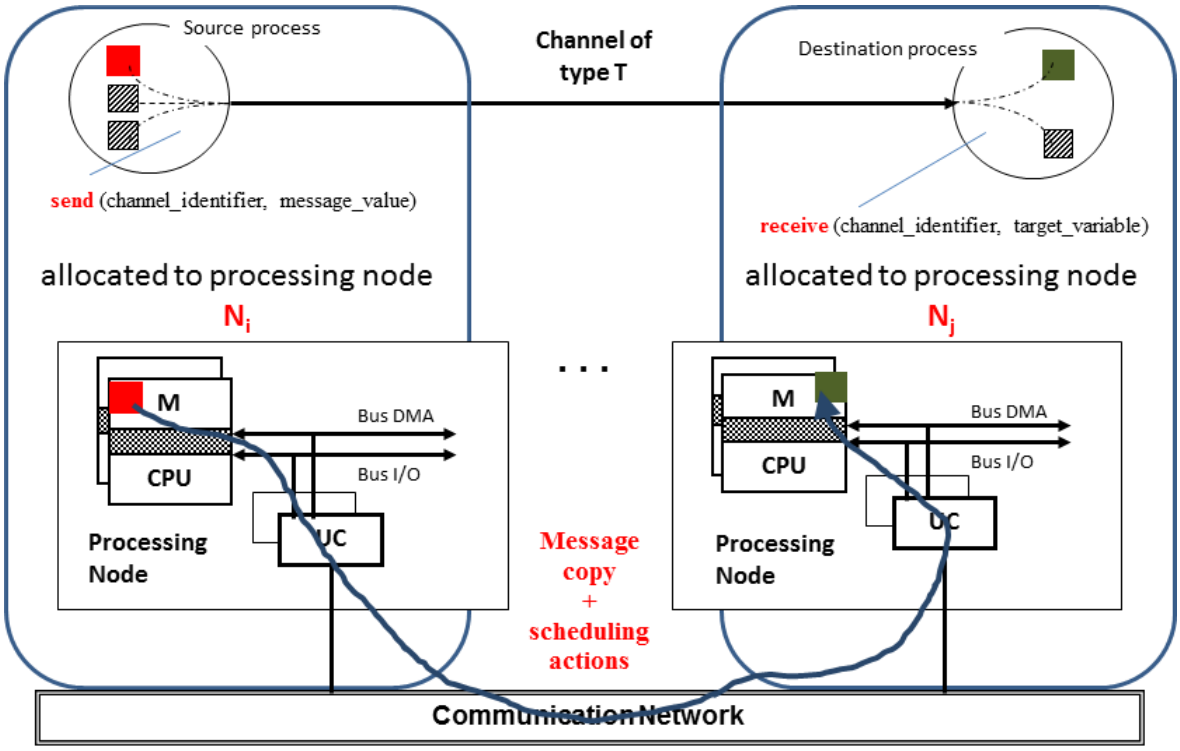


8.2 Run-time support

In Sect. 1.4 we saw some general characteristics of distributed memory multicomputers, ranging from low-medium-end servers/clusters to high-end massively parallel systems. Correspondingly, the interconnection networks range from simple Fast Ethernet and Gigabit Ethernet to more powerful Fat Tree and Generalized Fat Tree Myrinet (over 1 Gbit/sec) and Infiniband (from 1 to over 10 Gbit/sec) and their evolutions.

Basically, the run-time support of interprocess communication is different from the uni- and multi-processor versions, because of the distributed memory characteristic:

- let us consider two communicating processes A and B allocated onto distinct processing node PE_i and PE_j respectively;
- let the channel descriptor CH be allocated in the PE_j memory, i.e. the processing node where the destination process B is allocated;
- the *send* execution in A must verify whether the partner process is allocated in PE_i or in a different node. In the first case, a "normal" local *send* implementation, according to the PE_i architecture, is executed. Otherwise, A *delegates* to the (to one of the) process(es) currently running on PE_j the task of executing the *send* primitive locally. For this purpose, all the needed parameters are passed from PE_i to PE_j by value through the interconnection network, in particular the channel identifier and the message value;
- the *receive* primitive is always executed locally.



More in depth, a solution similar to the one described in Sect. 6.6 can be adopted.

A partial view of the channel descriptor (CH_s) is available in PE_i. It contains the following main information:

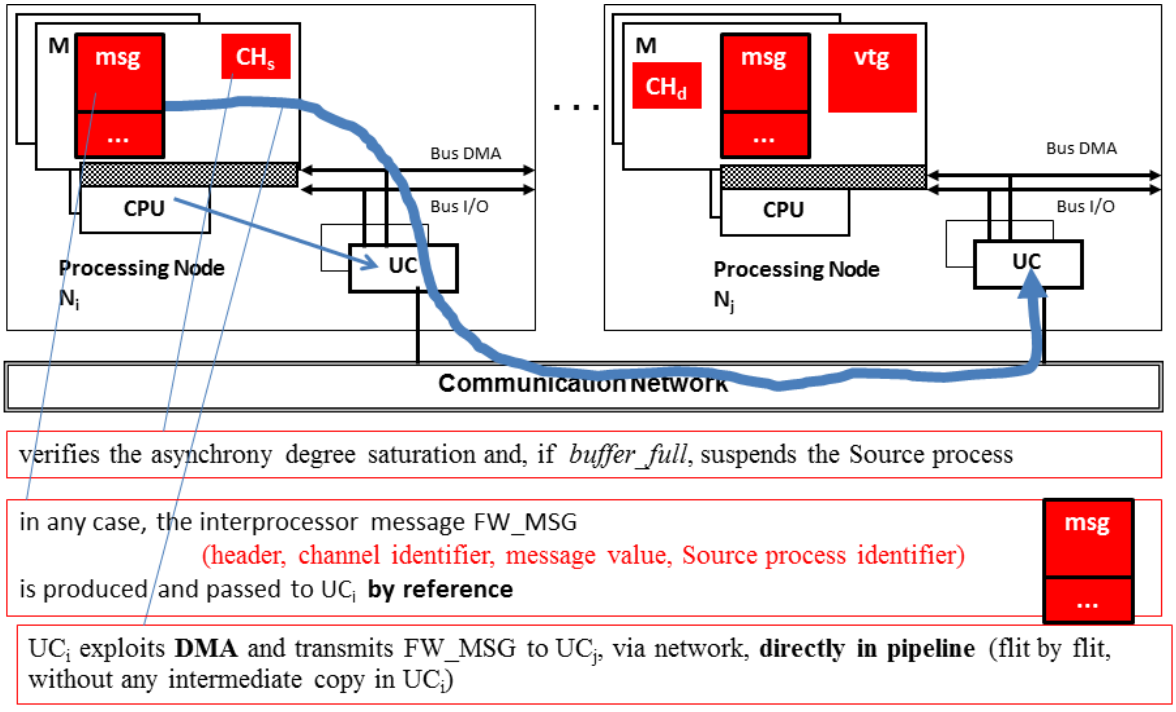
- processing node in which the destination process is allocated,
- message length,
- asynchrony degree,
- wait boolean variable,
- number of the currently buffered messages.

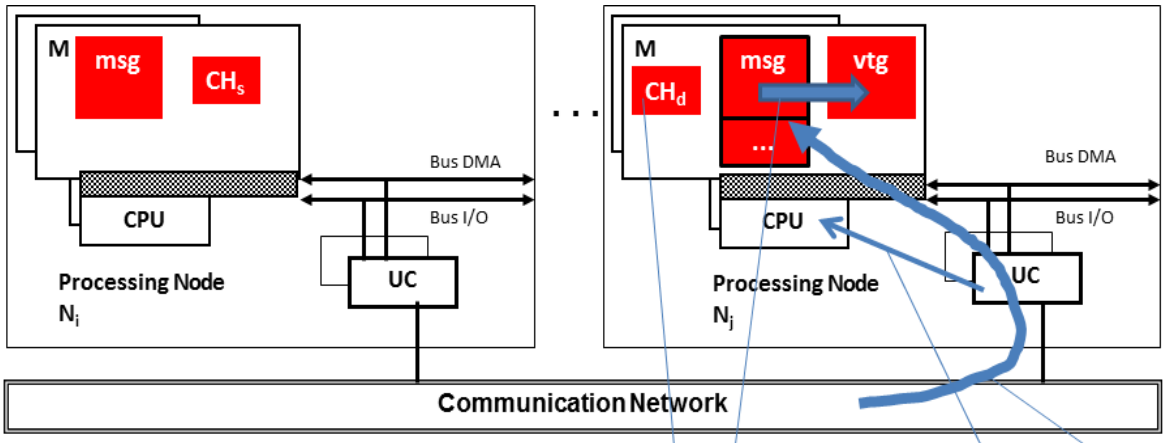
For a remote communication, an information packet (source node, destination node, message length, channel identifier, message value source process identifier) is sent to the Communication Unit (UC) in DMA, and from UC over the network to the destination node.

The sender process verifies the asynchrony degree saturation in CH_s and, if *buffer_full*, passes into the waiting state.

It is a task of the *receive* primitive to cause the updating of the *number of buffered messages* in CH_s, via an interprocessor message from PE_j to PE_i. In PE_i, the interrupt handler updates the number of currently buffered messages in CH_s and checks the *wait* boolean: if true, the source process is waked-up.

The following Figures summarize the distributed run-time support implementation.



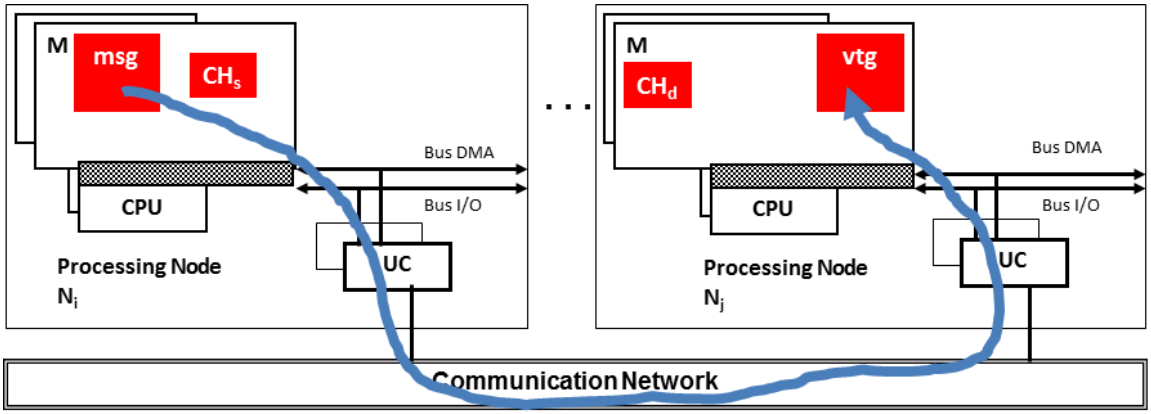


The **pipeline** trasmission is continued in N_j: UC_j copies FW_MSG, via DMA, in Nj memory **directly** (without any intermediate copy)

Running process (or KP) is interrupted by UCj; the interrupt message is the reference (capability) to FW_MSG

Running process (or KP) acquires FW_MSG, then Ch_{dest} and VTG, into its own addressing space. So, the *local send* can be executed (without checking buffer_full).

Optimization: UC is KP, thus the additional copy of FW_MSG is saved ! (on the fly execution)



In practice, even in a distributed memory architecture, a memory-to-memory copy can be implemented (plus additional operations for low level scheduling),

provided that the communication network protocol is the **primitive**, firmware one.

If **IP** protocol is adopted, then several additional copies and administrative operations are done. IP overhead is prevailing, also compared to the network latency.

About the interprocess communication *cost model*, also according to the study of Sect. 3.2, we can say that:

- If the communication network is used with the *primitive firmware routing and flow-control protocol*, we achieve similar results to the shared memory run-time:
  - for systems realized in a rack

$$T_{setup} \sim 10^3\ \tau,\ T_{transm} \sim 10^2\ \tau$$

  - otherwise, for long distance networks, the transmission latency dominates, e.g.

$$T_{setup} \sim 10^3\ \tau,\ T_{transm} \sim 10^4\ \tau \quad \text{till} \quad 10^6\ \tau$$

- If the communication network is used with the *IP protocol*, i.e., the application is IP-dependent, an additional overhead is paid due to the protocol actions (e.g., formatting, de-formatting) inside the nodes (plus transmission overhead on long distance networks):
  - on rack: $T_{setup} \sim 10^5\ \tau,\ T_{transm} \sim 10^4\ \tau$
  - on long distances: $T_{setup} \sim 10^7\ \tau,\ T_{transm} \sim 10^8\ \tau$

9. Solved exercises

General scheme of exercises integrating Part 1 and Part 2:

Consider the parallel programming exercises of Part 1, for example Sect. 17, 18.

Study their implementations on a SMP architecture and on a NUMA architecture. In particular, discuss

a) how the memory hierarchy is exploited,

b) how the shared data structures are mapped.

Assume typical reasonable values for the parameters characterizing the memory hierarchy, and suitable logarithmic interconnection networks for the SMP and for the NUMA architecture.

Moreover:

c) assuming that the given values of T_{setup} and T_{transm} have been estimated for a certain value of p, discuss under which conditions such estimate is accurate for the given computation and its mapping (all the other parameters of the cost model are assumed to be verified).

9.1 Exercise 1

Consider the data-parallel implementation of the following module operating on streams:

int A[M]; int x;

while (true) do

{ *receive (input_stream, A);*

$\forall$ *i = 0 .. M–1:*

A[i] = F (A[i]);

$x = G_{i=0}^{M-1} A[i]$; */G is an associative function/*

send (output_stream, x)

}

The data-parallel version, with parallelism degree n, is executed on a NUMA architecture with a binary Fat Tree interconnection network.

Discuss a process-level implementation and a mapping strategy able to exploit the given architecture at best, and determine the corresponding value of parameter p (mean number of processing nodes sharing the same memory module).

<u>Optional</u>*:* for an all-cache architecture and under the assumption that the A partition size is less than the processing node cache capacity, give an approximate evaluation of parameter T_p (mean time between two consecutive accesses of a processing node to the same shared memory module).

Symbol ♦ denotes that proper explanations must be inserted by the student according to what is contained in the Course Notes.

For each stream element, the module computation is functionally expressed by:

$$x = reduce\ (\ map\ (A, F), G\ )$$

which implies a pipeline execution of *map* and *reduce* on different stream elements.

Each input array is *scattered*: let us assume that this operation is implemented according to a linear strategy.

The *reduce* computation is implemented logarithmically by a tree structure mapped onto the same linear array of *n* workers. ♦

In a message-passing process-level implementation, we can use *symmetric* channels only.

One possible mapping strategy consists in allocating each *input* channel descriptor in the PE in which the corresponding worker is allocated. Thus, each worker PE stores one input scatter channel and from *1* to *log n* input reduce channels. An approximate average value of *p* is 3, however the high variance has a meaningful impact on the contention rate.

A more efficient mapping strategy is to allocate, in each PE, the input scatter channel and *the* reduce output channel for the logarithmic stencil implementation. Now, *p* is constant and equal to 2. This mapping strategy minimizes the value of *p*, which is the most critical parameter for the under-load memory access latency R_Q. ♦ For $p \sim 2$, R_Q is a just a bit greater that the base latency.

On the other hand, both the described mapping strategies exploit all the possible distances over the tree. However, we know that, for a Fat Tree architecture, the impact of the network distance (δ) is much lower than the impact of *p*. ♦

Evaluation of T_p:

- during the *receive* phase, about two accesses to the input_stream channel descriptor are done: this causes a cache fault;
- during the *map* phase, since A is characterized by locality only, we have a cache fault every σ iterations, where σ is the cache block size;
- under the assumption on the A partition size, during the *reduce* phase A is characterized by reuse; provided that during the *map* the A blocks are maintained in cache, we have no cache fault during the local *reduce* phase;
- during the logarithmic reduce phase, each worker performs a *send*, thus about two accesses to the stencil channel descriptor;
- during the *send* phase, we have about two cache faults in accessing the output_stream channel descriptor.

On the average, for zero-copy communications we can estimate:

$$T_p = \frac{M\ T_F + \frac{M}{n}\ T_G + \lg_2 n\ (T_G + T_{send}) + \ T_{send}}{6 + \frac{N}{\sigma}}$$

9.2 Exercise 2

Describe the behavior of the switching node of a wormhole Generalized Fat Tree with arity $k = 2$ and dimension $n = 8$, used in a SMP architecture. The description must highlight the detailed actions performed at the clock cycle level.

See Section 3.5.3. ♦

In particular, explain clearly that:

- the different routing algorithms for the butterfly behavior (processor to memory) and the tree behavior (processor to processor) have to be distinguished and their implementation described, *proving that this action requires just one clock cycle*,
- it must be proved that *no additional clock cycle* is spent when a non-conflicting packet is received while another transmission is already on going.

9.3 Exercise 3

Discuss the relationship between the utilization of locking mechanisms in parallel computations and the application of cache coherence techniques (automatic and non-automatic) for shared memory architectures.

See Section 7.3. ♦

In particular, *independently of the performance evaluation of a critical section*, it is important to stress the impact that locking *synchronizations* have in automatic techniques to avoid inefficient ping-pong effects. For non-automatic techniques, it must be explained how locking *synchronizations* are sufficient for implementing cache coherence, and, in addition, how it is possible to introduce optimizations according to the specific locking algorithm and assembler-level notations.

9.4 Exercise 4

1) A sequential program is defined by the following algorithm, with $M = 10^4$:

```
int A[M], B[M]; int x;
x = 0;
∀ i = 0 .. M–1:
        ∀ j = 0 .. M–1:
                x = x + A[i]*B[j]
```

Evaluate the completion time for a D-RISC scalar pipelined CPU with 4-stage multiplier functional unit, primary data cache of 32K words and blocks of 8 words, and secondary cache with access time of two clock cycles.

2) The program in *1)* defines a module Q operating on streams: *A* is received from the input stream, *x* is sent onto the output stream, and *B* is encapsulated statically with a given initial value.

The parallel architecture has $N = 32$ processing nodes, each one with communication processor and zero-copy interprocess communication. $T_{setup} = 10^3\ \tau$, $T_{transm} = 10^2\ \tau$, and the interarrival time = $10^7\ \tau$, where τ is the CPU clock cycle.

i. Define, explain and evaluate two different parallel versions of Q.
ii. Evaluate the relative efficiency of the whole computation and of each module contained in the two parallel versions.

3) Recognize which data structures of the two parallel versions must be implemented in a cache coherent manner in a NUMA architecture with automatic cache coherence and in a NUMA architecture with non-automatic cache coherence.

1) The non-optimized compiled code is (general registers Ri and Rx are initialized at zero, RA, RB and RX at the base addresses of A, B and X, RM at the M value):

```
LOOPi:    LOAD   RA, Ri, Ra
          CLEAR  Rj
LOOPj:    LOAD   RB, Rj, Rb
          MUL    Ra, Rb, Ra
          ADD    Rx, Ra, Rx
          INCR   Rj
          IF < Rj, RM, LOOPj
          INCR   Ri
          IF <   Ri, RM, LOOPi
          STORE RX, 0, Rx
```

The program performance depends on the innermost loop (executed M^2 times), thus we will focus on its optimization only. In the shown version, there is a logical dependence IU-EU induced by INCR Rj on IF < Rj (distance k = 1, $N_Q = 2$): the critical sequence contain a long latency instruction, which however is independent of INCR. Moreover, the IF instruction introduces an additional bubble. An optimized code is:

```
          …
          LOAD   RB, Rj, Rb, not_deallocate
LOOPj:    INCR   Rj
          MUL    Ra, Rb, Ra
          ADD    Rx, Ra, Rx
          IF < Rj, RM, LOOPj, delayed branch
          LOAD   RB, Rj, Rb, not_deallocate
          …
```

in which both the mentioned degradations are eliminated. Notice that instructions MUL and ADD do not affect the logical dependence delay: despite the long latency of MUL, they are executed in parallel to LOAD, INCR and IF, as it can be verified with the graphical simulation. Thus, the instruction service time is the ideal one:

$$T = t$$

The completion time of the innermost loop is:

$$T_{c\text{-}inner} = 5\,M\,T = 5\,M\,t = 10\,M\,\tau$$

With very good approximation, the ideal completion time (with perfect cache) is:

$$T_{c\text{-}id} \sim M\,T_{c\text{-}inner} = 10\,M^2\,\tau$$

The effective service time must take into account the delay introduced by cache faults:

$$T_c = T_{c-id} + N_{fault} * T_{fault}$$

The program data are characterized by locality of A and B and by reuse of all the elements of B. Since M is less than the cache capacity, B can be maintained in data cache once loaded for the first time, provided that instruction *LOAD ... Rb* contains the annotation "not_deallocate", as shown in the optimized code. Thus

$$N_{fault} * T_{fault} = 2\frac{M}{\sigma} * 2\sigma\tau = 4M\tau$$

which is negligible compared to $T_{c\text{-}id}$, owing to the reuse optimization. Notice that A and B are entirely contained in the secondary cache when they are referred. In conclusion,

$$T_c = 10M^2\tau = 10^9\tau$$

2) The completion time evaluated for the sequential program is the ideal service time of module Q operating on streams:

$$T_{calc} = 10^9\tau$$

According to the problem specifications, the *receive (..., A)* and the *send (..., x)* primitives do not affect the service time.

The optimal parallelism degree is given by:

$$n = \frac{T_{calc}}{T_A} = 100$$

which is greater than the number *N* of available nodes. Thus, it must be reduced to $N - n_p$, where n_p is the number of service modules.

Parallelization: farm version

Since Q is a pure function, it can be parallelized by the *farm* paradigm with statically replicated B. For the ideal version with *n* = 100, the Emitter module is not a bottleneck:

$$T_{E-id} = T_{send}(M) \sim 10^6\tau < T_A$$

thus the farm solution could be able to achieve the optimal service time T_A. Because of the limited number of nodes *N* and n_p = 2, the number of workers is reduced to:

$$n_w = 30$$

so the effective service time is equal to *3,3* T_A.

The relative efficiency of the whole farm computation and of the workers are

$$\epsilon_{farm} = 1$$

$$\epsilon_{worker} = 1$$

both for the ideal and for the effective parallelism degree.

For *n = 100*:

$$\epsilon_{emitter} = \rho_{emitter} = 10^{-1}$$

$$\epsilon_{collector} = \rho_{collector} = 10^{-4}$$

For *n = 30*:

$$\epsilon_{emitter} = \rho_{emitter} = \frac{10^{-1}}{3{,}3}$$

$$\epsilon_{collector} = \rho_{collector} = \frac{10^{-4}}{3{,}3}$$

Parallelization: data-parallel version

A *data-parallel solution* consist in a *map* followed by a *reduce*. With a linear array of *M* virtual processors VP[M], the generic VP[i] encapsulates A[i], the whole B (statically replicated), and a local copy of *x*: let us call it *X[i]*. At the end of the map phase, a global *reduce (X[M], +)* is applied.

In the ideal case, with *n = 100* workers, each one encapsulates *g = M/n = 100* elements of B statically, *g* elements of scattered *A*, and a local *X*. The scatter is not a bottleneck even if implemented by a single module and with *n = 100*:

$$T_{scatter} = n\, T_{send}\,(g) \sim 10^6\, \tau < T_A$$

The reduce is implemented by a tree structure mapped onto the linear array of workers, thus with logarithmic latency. The *(n-1)-th* worker is in charge of sending the final result onto the output stream

However, due to limited *N* and $n_p = 1$, the number of workers is reduced to

$$n_w = 31$$

so the effective service time is *3,2* T_A (a slightly better bandwidth, compared to the farm version, provided that the load is balanced).

The reduce latency

$$T_{reduce} = lg_2(n_W)\,(\,T_{send}\,(1) + T_+) \sim 10^4\, \tau \ll T_A$$

which is negligible.

The relative efficiencies are analogous to the farm evaluation:

$$\epsilon_{dp} = 1$$

$$\epsilon_{worker} = 1$$

$$\epsilon_{scatter} = \begin{cases} 10^{-1}\ for\ n = 100 \\ \dfrac{10^{-1}}{3{,}2}\ for\ n = 31 \end{cases}$$

3) In a message passing implementation, the shared data structures belong to the run-time support only: in a NUMA architecture, they are all the *channel descriptors* and the *target variables* (zero-copy implementation).

For the channel descriptors, see the Section 7.1, 7.4.

In a non-automatic cache coherence architecture, the coherence of the target variables does not require any overhead, provided that the Write-Through option exists in order to copy the message directly into the shared memory *skipping the cache*, i.e. without the need of allocating the target variable blocks in the sender cache and then copying them into the shared memory (for the sender process, the target variable is write-only).

In an automatic cache coherence architecture, such a distinction cannot be done because the firmware interpreter cannot distinguish between write-only and read-write blocks:

- the target variable blocks must be allocated in the sender cache, in general with an *invalidation* operation of the destination cache blocks, which requires an useless copy ,
- the message is copied into the sender cache blocks,
- *by invalidation* the sender cache blocks are copied into the target variable blocks in the destination cache, when the target variable is referred to by the destination process.

9.5 Exercise 5

a) Evaluate the base latency of a memory-to-memory copy in a NUMA multiprocessor, under the following assumptions:

- the source data structure is an integer array *A[M]* allocated in the local memory of the executing node PE_{source}, and the destination data structure is an integer array *B[M]* allocated in the local memory of a node PE_{dest}, where the distance between PE_{source} and PE_{dest} is the maximum for the given architecture,
- the algorithm is executed without locking,
- cache-coherence is algorithm-dependent,
- $N = 128$ processing nodes,
- each node has a pipelined scalar D-RISC CPU with clock cycle τ, primary data cache operating on demand with blocks of 8 words, and no secondary data cache,
- the local memory of each node is interleaved with 8 modules and clock cycle of 50τ,
- the interconnection structure is a binary Fat Tree with wormhole flow control, flits and links are one word wide, and the link transmission latency is equal to τ.

The base latency, to be expressed as a function of τ and M, must be evaluated as the completion time of a program optimized for, and executed on, the given CPU.

b) Discuss qualitatively the possible differences in the implementation of the memory-to-memory copy in case the architecture has automatic cache-coherence with invalidation.

The assembler code for the memory-to memory copy is:

```
LOOP:  LOAD   RA, Ri, Ra
       STORE  RB, Ri, Ra
       INCR   Ri
       IF <   Ri, RM, LOOP
```

which can be optimized for a scalar pipeline CPU (the base address of B is decremented by one):

```
       LOAD   RA, Ri, Ra
LOOP:  INCR   Ri
       STORE  RB, Ri, Ra
```

```
        IF <   Ri, RM, LOOP, delayed_branch
        LOAD   RA, Ri, Ra
```

The only cause of performance degradation is now the logical dependency induced by the LOAD onto the STORE instruction. Without taking into account the cache faults, we determine the instruction service time, that is (*add explanations:* e.g. based on the graphical simulation):

$$T_{id} = \frac{5}{4}\, t$$

and the completion time:

$$T_{c-id} = 4\, M\, T = \; 5\, M\, t = 10\, M\, \tau$$

Taking into account the cache hierarchy, we have *M/8* fault on A and *M/8* fault on B.

The blocks of A must be transferred from the main memory (local memory of PE_{source}) into the PE_{source} cache one block at the time.

In the given multiprocessor architecture, with *algorithm-dependen*t cache-coherence, we can design the computation in such a way that the blocks of B are *not* transferred into the cache of PE_{source}. Instead, they are merely allocated in the PE_{source} cache on each fault occurrence and, once their values are produced, explicitly written into the main memory (the local memory of PE_{dest}) one block at the time.

This behavior can be achieved by writing the obtained value of a cache block into the main memory every 8 iterations. We can compile the code with a partial loop unfolding, i.e. 8-loop unfolding, and adding the proper annotation in the STORE instruction of the 8-th iteration of the unfolding sequence:

```
/ first iteration of a sequence of 8 iterations /
        LOAD   RA, Ri, Ra
LOOP:   INCR   Ri
        STORE  RB, Ri, Ra
/ second iteration of a sequence of 8 iterations /
        LOAD   RA, Ri, Ra
LOOP:   INCR   Ri
        STORE  RB, Ri, Ra
…
/ 8-th iteration of a sequence of 8 iterations /
        LOAD   RA, Ri, Ra
        INCR   Ri
        STORE  RB, Ri, Ra, write_block
        IF <   Ri, RM, LOOP, delayed_branch
        LOAD   RA, Ri, Ra
```

(We achieve also some other improvements with this version, since a lower number of instructions is executed. However, it will not be considered for simplicity).

Thus, the required base latency is given by:

$$T_{mem-to-mem} = \; T_{c-id} + \frac{M}{8}\; T_{read-block} + \frac{M}{8}\; T_{write-block}$$

The block reading is from the local memory. Assuming a typical node with W node interface unit and I_M interface unit of the local memory modules (*add explanations*; see Part 2, Sect. 1.1), the base block reading latency is (Part 2. Sect. 2.5.4):

$$T_{read-block} = 3\ t_{hop} + \ \tau_M + \ (2m + d - 3)\ t_{hop}$$

where:

$$t_{hop} = 2\tau$$

$$\tau_M = 50\tau$$

$$m = 8\ (cache\ block)$$

$$d = 3\ (distance\ between\ the\ memory\ modules\ and\ the\ CPU)$$

Thus:

$$T_{read-block} = 88\ \tau$$

The remote block writing operation exploits the wormhole binary Fat Tree with dimension $n = lg_2N = 7$. The maximum distance is $d_{net} = 2n = 14$. We have (Part 2. Sect. 3.1.2; *add proper explanations*; the following formula is derived easily):

$$T_{write-block} = [\ 12 + 2\ (\sigma + \ d_{net})\ ]t_{hop} + \ \tau_M = 162\ \tau$$

In conclusion:

$$T_{mem-to-mem} \sim 41\ M\ \tau$$

It could be further optimized if the architecture makes it possible to overlap the block reading and block writing operations, thus reducing the base latency to the delays of writing operations only.

b) With an automatic cache-coherence architecture the assembler code does not contain the loop unfolding and the write-block annotation, because the automatic technique is independent of the specific algorithm and any block transfer is automatically executed as an invalidation.

The block writing latency is *doubled*, because each fault generated by the STORE instruction causes an invalidation of the B blocks, thus their transfer from the PE_{dest} local memory or cache into the PE_{source} cache. As seen in point *a)*, this transfer is semantically unnecessary, however it is imposed by the automatic cache-coherence behavior.

10. Other exercises

1) Assume that a wormhole Generalized Fat Tree network, with arity $k = 2$ and $n = 8$, is used in a SMP architecture with a double role: *a)* processor-to-memory interconnection, and *b)* processor-to-processor interconnection. The firmware messages contain in the first word: routing information (source identifier, destination identifier), message type (a or b), and message length in words. Links and flits size is one word.

Describe the detailed (e.g., at the clock cycle grain) behavior of a switching node.

2) Consider a *k-ary 2-cube* interconnection network with deterministic routing and wormhole flow control (one-word links, one-word flits).

Describe the structure and behavior of a Network Node. The description should clearly show whether it is feasible to achieve the maximum bidirectional bandwidth, provided that the proper traffic conditions hold.

3) Implement in detail the lock operation according to the fair algorithm. Evaluate its latency in case of *green* semaphore.

4) Find an alternative implementation of zero-copy *send* primitive, able to reduce the locking critical section duration.

5) Consider the preemptive wake-up procedure in a SMP architecture. In order to avoid the consistency problem arising in the processor to be preempted, a complex locking procedure could be implemented: which data structures should have been locked, and when unlocked ?

That is, implement the preemptive wake-up procedure in such a way that the situation detected by the waking process doesn't change until the wake-up has been completed.

6) Consider a NUMA multiprocessor architecture with 128 nodes. Each node includes a D-RISC pipelined CPU without secondary cache. Data cache blocks are 4-word wide. Local memory locations are 128-bit wide. The maximum capacity of main memory is 512 Giga words. The interconnection structure is of logarithmic kind, with one-word links and wormhole flow control. All the system processing units have the same clock cycle τ, except the local memory units having a clock cycle equal to 10τ. Any inter-chip link has a 4τ transmission latency.

- *a)* Explain the architecture of a generic node. Show the structure of firmware messages used for local memory accesses and for remote memory accesses, and explain how and where they are built.
- *b)* Determine the base latency of local memory accesses and of remote memory accesses.
- *c)* Explain the qualitative impact of parameter p (the average number of nodes accessing the same memory module) on the memory access latency. Estimate a reliable value of p for a system executing a parallel program structured as a 16-stage pipeline, where each stage is a 4-worker farm.

d) Evaluate the interprocess communication latency approximately, in terms of the *base* memory access latencies only, assuming zero-copy communication and 4-word messages. Referring to the example in point *c)*, estimate the approximation degree of this evaluation compared to the evaluation in terms of under-load memory access latencies.

7) Let's assume that the source code of run-time support for *send* and *receive* communication primitives is available in a uniprocessor version for a given assembler-firmware architecture S. We try to exploit this code for other kinds of architecture at best.

Explain which parts can be preserved, and which parts have to be modified/replaced *and in which way*, for *a)* SMP, *b)* NUMA, and *c)* cluster architectures, exploiting the given uniprocessor architecture S as building block. Explain possible specific features required to S for the *a), b), c)* implementations.

8)

a) Consider the under-load memory access latency in all-cache multiprocessor architectures. For all the parameters, that have influence on such latency, discuss their qualitative impact. Where possible, introduce some quantitative considerations, in particular under which conditions the impact of each parameters is more or less significant.

b) Consider an all-cache NUMA architecture with 32 nodes connected by a toroidal ring with wormhole flow control and 32-bit links. Show the architecture of a generic node based on D-RISC pipelined CPU. Evaluate the base memory access latency. Explain qualitatively, yet formally, why the under-load memory access latency decreases by replacing the single ring with *m* independent toroidal rings.

9) Consider the architectures with communication processor. 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.

10) Study algorithm-dependent solutions to cache coherence applied to the secondary caches in SMP architectures.

11) Describe the *send* and *receive* run-time support in detail for a multicomputer architecture.

12) Consider a multicomputer in which the nodes have an internal multiprocessor architecture.

Study the implications on the interprocess communication run-time support, distinguishing between SMP and NUMA architectures.