Chapter 2
Parallel Computer Architecture

The possibility for a parallel execution of computations strongly depends on the architecture of the execution platform. This chapter gives an overview of the general structure of parallel computers which determines how computations of a program can be mapped to the available resources, such that a parallel execution is obtained. Section 2.1 gives a short overview of the use of parallelism within a single processor or processor core. Using the available resources within a single processor core at instruction level can lead to a significant performance increase. Sections 2.2 and 2.3 describe the control and data organization of parallel platforms. Based on this, Sect. 2.4 presents an overview of the architecture of multicore processors and describes the use of thread-based parallelism for simultaneous multithreading.

The following sections are devoted to specific components of parallel platforms. Section 2.5 describes important aspects of interconnection networks which are used to connect the resources of parallel platforms and which are used to exchange data and information between these resources. Interconnection networks also play an important role for multicore processors for the connection between the cores of a processor chip. Section 2.5 describes static and dynamic interconnection networks and discusses important characteristics like diameter, bisection bandwidth, and connectivity of different network types as well as the embedding of networks into other networks. Section 2.6 addresses routing techniques for selecting paths through networks and switching techniques for message forwarding over a given path. Section 2.7 considers memory hierarchies of sequential and parallel platforms and discusses cache coherence and memory consistency for shared memory platforms. As example for the architecture of a large parallel system, Sect. 2.8 gives a short description of the IBM Blue Gene/Q series.

2.1 Processor Architecture and Technology Trends

Processor chips are the key components of computers. Considering the trends that can be observed for processor chips during the last years, estimations for future developments can be deduced.

An important performance factor is the clock frequency of the processor, which determines the cycle time of the processor and therefore the time needed for the execution of an instruction. Between 1987 and 2003, an average annual increase of about 40% could be observed for desktop processors [94]. Since 2003, the clock frequency of desktop processors remains nearly unchanged and no significant increases can be expected in the near future [93, 122]. The reason for this development lies in the fact that an increase in clock frequency leads to an increase in power consumption, mainly due to leakage currents which are transformed into heat, which then requires a larger amount of cooling. Using current state-of-the-art cooling technology, processors with a clock rate significantly above 3.3GHz cannot be cooled permanently without a large additional effort. Another important influence on the processor development is technical improvements in processor manufacturing.

Internally, processor chips consist of transistors. The number of transistors contained in a processor chip can be used as a rough estimate of its complexity and performance. **Moore's law** is an empirical observation which states that the number of transistors of a typical processor chip doubles every 18 to 24 months. This observation has first been made by Gordon Moore in 1965 and is now valid for more than 40 years. The increasing number of transistors can be used for architectural improvements like additional functional units, more and larger caches, and more registers.

In 2012, a typical desktop processor chip contains 1–3 billions transistors. For example, an Intel Core i7 Sandy Bridge quad-core processor contains 995 million transistors, an Intel Core i7 Bridge-HE-4 quad-core processor contains 1.4 billion transistors, and an Intel Xeon Westmore-EX 10-core processor contains 2.6 billion transistors.

The increase of the number of transistors and the increase in clock speed has lead to a significant increase in the performance of computer systems. Processor performance can be measured by specific benchmark programs that have been selected from different application areas to get a representative performance measure of computer systems. Often, the SPEC benchmarks (*System Performance and Evaluation Cooperative*) are used to measure the integer and floating point performance of computer systems [102, 94, 183], see `http://www.spec.org`. Measurements with these benchmarks show that between 1986 and 2003, an average annual performance increase of about 50% could be reached for desktop processors [94]. It can be noted that the time period of this large performance increase corresponds to the time period in which the clock frequency could be significantly increased each year. Since 2003, the average annual performance increase of desktop processors is about 22%. This is still a significant increase which is reached, although the clock frequency nearly remained constant, showing that the annual increase in transistor count has been used for architectural improvements that lead to a reduction of the average time for executing an instruction.

In the following, we give a short overview of such architectural improvements. Four phases of microprocessor design trends can be observed [41] which are mainly driven by the internal use of parallelism:

1. **Parallelism at bit level**: Up to about 1986, the word size used by the processors for operations increased stepwise from 4 to 32 bits. This trend has slowed down and ended with the adoption of 64-bit operations beginning in the 1990s. This development has been driven by demands for improved floating point accuracy and a larger address space. The trend has stopped at a word size of 64 bits, since this gives sufficient accuracy for floating point numbers and covers a sufficiently large address space of 2^{64} bytes.
2. **Parallelism by pipelining**: The idea of pipelining at instruction level is an overlapping of the execution of multiple instructions. The execution of each instruction is partitioned into several steps which are performed by dedicated hardware units (pipeline stages) one after another. A typical partitioning could result in the following steps:

   (a) *fetch*: fetch the next instruction to be executed from memory;
   (b) *decode*: decode the instruction fetched in step (a);
   (c) *execute*: load the operands specified and execute the instruction;
   (d) *write back*: write the result into the target register.

   An instruction pipeline is like an assembly line in automobile industry. The advantage is that the different pipeline stages can operate in parallel, if there are no control or data dependencies between the instructions to be executed, see Fig. 2.1 for an illustration. To avoid waiting times, the execution of the different pipeline stages should take about the same amount of time. This time determines the cycle time of the processor. If there are no dependencies between the instructions, in each clock cycle the execution of one instruction is finished and the execution of another instruction is started. The number of instructions finished per time unit is defined as the *throughput* of the pipeline. Thus, in the absence of dependencies, the throughput is one instruction per clock cycle.
   In the absence of dependencies, all pipeline stages work in parallel. Thus, the number of pipeline stages determines the **degree of parallelism** attainable by a pipelined computation. The number of pipeline stages used in practice depends on the specific instruction and its potential to be partitioned into stages. Typical numbers of pipeline stages lie between 2 and 26 stages. Processors which use pipelining to execute instructions are called **ILP processors** (*instruction level parallelism*). Processors with a relatively large number of pipeline stages are sometimes called **superpipelined**. Although the available degree of parallelism

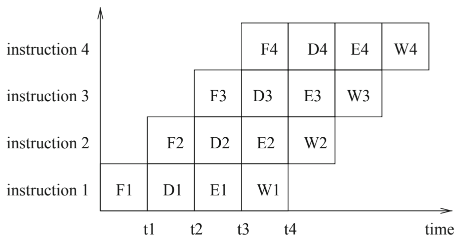


Fig. 2.1 Overlapping execution of four independent instructions by pipelining. The execution of each instruction is split into four stages: *fetch* (F), *decode* (D), *execute* (E), and *write back* (W).

increases with the number of pipeline stages, this number cannot be arbitrarily increased, since it is not possible to partition the execution of the instruction into a very large number of steps of equal size. Moreover, data dependencies often inhibit a completely parallel use of the stages.

3. **Parallelism by multiple functional units**: Many processors are *multiple-issue processors*. They use multiple, independent functional units like ALUs (*arithmetic logical unit*), FPUs (*floating point unit*), load/store units, or branch units. These units can work in parallel, i.e., different independent instructions can be executed in parallel by different functional units. Thus, the average execution rate of instructions can be increased. Multiple-issue processors can be distinguished into **superscalar** processors and **VLIW** (*very long instruction word*) processors, see [94, 41] for a more detailed treatment.
   The number of functional units that can efficiently be utilized is restricted because of data dependencies between neighboring instructions. For superscalar processors, these dependencies are determined at runtime dynamically by the hardware, and decoded instructions are dispatched to the instruction units using dynamic scheduling by the hardware. This may increase the complexity of the circuit significantly. Moreover, simulations have shown that superscalar processors with up to four functional units yield a substantial benefit over a single functional unit. But using even more functional units provides little additional gain [41, 110] because of dependencies between instructions and branching of control flow.
4. **Parallelism at process or thread level**: The three techniques described so far assume a *single sequential* control flow which is provided by the compiler and which determines the execution order if there are dependencies between instructions. For the programmer, this has the advantage that a sequential programming language can be used nevertheless leading to a parallel execution of instructions. However, the degree of parallelism obtained by pipelining and multiple functional units is limited. This limit has already been reached for some time for typical processors. But more and more transistors are available per processor chip according to Moore's law. This can be used to integrate larger caches on the chip. But the cache sizes cannot be arbitrarily increased either, as larger caches lead to a larger access time, see Section 2.7.
   An alternative approach to use the increasing number of transistors on a chip is to put multiple, independent processor cores onto a single processor chip. This approach has been used for typical desktop processors since 2005. The resulting processor chips are called **multicore processors**. Each of the cores of a multicore processor must obtain a separate flow of control, i.e., parallel programming techniques must be used. The cores of a processor chip access the same memory and may even share caches. Therefore, memory accesses of the cores must be coordinated. The coordination and synchronization techniques required are described in later chapters.

A more detailed description of parallelism by multiple functional units can be found in [41, 94, 155, 184]. Section 2.4 describes techniques like simultaneous multithreading and multicore processors requiring an explicit specification of parallelism.

2.2 Flynn's Taxonomy of Parallel Architectures

Parallel computers have been used for many years, and many different architectural alternatives have been proposed and used. In general, a parallel computer can be characterized as a collection of processing elements that can communicate and cooperate to solve large problems fast [14]. This definition is intensionally quite vague to capture a large variety of parallel platforms. Many important details are not addressed by the definition, including the number and complexity of the processing elements, the structure of the interconnection network between the processing elements, the coordination of the work between the processing elements as well as important characteristics of the problem to be solved.

For a more detailed investigation, it is useful to make a classification according to important characteristics of a parallel computer. A simple model for such a classification is given by **Flynn's taxonomy** [58]. This taxonomy characterizes parallel computers according to the global control and the resulting data and control flows. Four categories are distinguished:

1. **Single Instruction, Single Data (SISD)**: There is one processing element which has access to a single program and data storage. In each step, the processing element loads an instruction and the corresponding data and executes the instruction. The result is stored back in the data storage. Thus, SISD is the conventional sequential computer according to the *von Neumann model*.
2. **Multiple Instruction, Single Data (MISD)**: There are multiple processing elements each of which has a private program memory, but there is only one common access to a single global data memory. In each step, each processing element obtains the *same* data element from the data memory and loads an instruction from its private program memory. These possibly different instructions are then executed in parallel by the processing elements using the previously obtained (identical) data element as operand. This execution model is very restrictive and no commercial parallel computer of this type has ever been built.
3. **Single Instruction, Multiple Data (SIMD)**: There are multiple processing elements each of which has a private access to a (shared or distributed) data memory, see Section 2.3 for a discussion of shared and distributed address spaces. But there is only one program memory from which a special control processor fetches and dispatches instructions. In each step, each processing element obtains from the control processor and the *same* instruction and loads a separate data element through its private data access on which the instruction is performed. Thus, the instruction is synchronously applied in parallel by all processing elements to different data elements.

For applications with a significant degree of data parallelism, the SIMD approach can be very efficient. Examples are multimedia applications or computer graphics algorithms to generate realistic three-dimensional views of computer-generated environments.

4. **Multiple Instruction, Multiple Data (MIMD)**: There are multiple processing elements each of which has a separate instruction and data access to a (shared or distributed) program and data memory. In each step, each processing element loads a separate instruction and a separate data element, applies the instruction to the data element, and stores a possible result back into the data storage. The processing elements work asynchronously to each other. Multicore processors or cluster systems are examples for the MIMD model.

Compared to MIMD computers, SIMD computers have the advantage that they are easy to program, since there is only one program flow, and the synchronous execution does not require synchronization at program level. But the synchronous execution is also a restriction, since conditional statements of the form

```
if (b==0) c=a; else c = a/b;
```

must be executed in two steps. In the first step, all processing elements whose local value of `b` is zero execute the `then` part. In the second step, all other processing elements execute the `else` part.

Some processors support SIMD computations as additional possibility for processing large uniform data sets. An example is the x86 architecture which provides SIMD instructions in the form of SSE (streaming SIMD extensions) or AVX (advanced vector extensions) instructions, see Sect. 3.4 for a more detailed description. The computations of GPUs are also based on the SIMD concept, see Sect. 7.1.

MIMD computers are more flexible as SIMD computers, since each processing element can execute its own program flow. On the upper level, multicore processors as well as all parallel computers are based on the MIMD concept. Although Flynn's taxonomy only provides a coarse classification, it is useful to give an overview of the design space of parallel computers.

2.3 Memory Organization of Parallel Computers

Nearly, all general-purpose parallel computers are based on the MIMD model. A further classification of MIMD computers can be done according to their memory organization. Two aspects can be distinguished: the physical memory organization and the view of the programmer of the memory. For the physical organization, computers with a physically shared memory (also called *multiprocessors*) and computers with a physically distributed memory (also called *multicomputers*) can be distinguished, see Fig. 2.2 for an illustration. But there exist also many hybrid organizations, for example providing a virtually shared memory on top of a physically distributed memory.

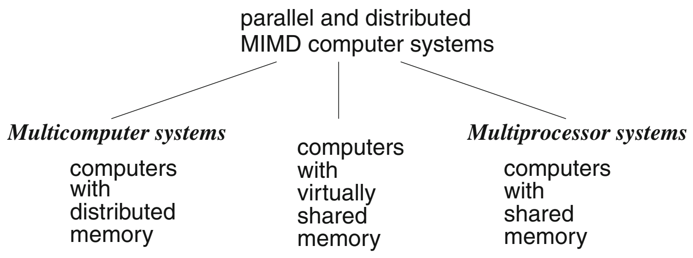

parallel and distributed MIMD computer systems
Multicomputer systems
computers with distributed memory
computers with virtually shared memory
Multiprocessor systems
computers with shared memory

Fig. 2.2 Forms of memory organization of MIMD computers.

From the programmer's point of view, it can be distinguished between computers with a distributed address space and computers with a shared address space. This view does not necessarily need to be conform with the physical memory. For example, a parallel computer with a physically distributed memory may appear to the programmer as a computer with a shared address space when a corresponding programming environment is used. In the following, we have a closer look at the physical organization of the memory.

2.3.1 Computers with Distributed Memory Organization

Computers with a physically distributed memory are also called **distributed memory machines** (DMM). They consist of a number of processing elements (called nodes) and an interconnection network which connects nodes and supports the transfer of data between nodes. A node is an independent unit, consisting of processor, local memory and, sometimes, periphery elements, see Fig. 2.3 a) for an illustration.

Program data are stored in the local memory of one or several nodes. All local memory is *private* and only the local processor can access the local memory directly. When a processor needs data from the local memory of other nodes to perform local computations, message passing has to be performed via the interconnection network. Therefore, distributed memory machines are strongly connected with the message-passing programming model which is based on communication between cooperating sequential processes and which will be considered in more detail in Chapts. 3 and 5. To perform message passing, two processes P_A and P_B on different nodes A and B issue corresponding send and receive operations. When P_B needs data from the local memory of node A, P_A performs a send operation containing the data for the destination process P_B. P_B performs a receive operation specifying a receive buffer to store the data from the source process P_A from which the data are expected.

The architecture of computers with a distributed memory has experienced many changes over the years, especially concerning the interconnection network and the coupling of network and nodes. The interconnection network of earlier multicomputers was often based on **point-to-point connections** between nodes. A node is

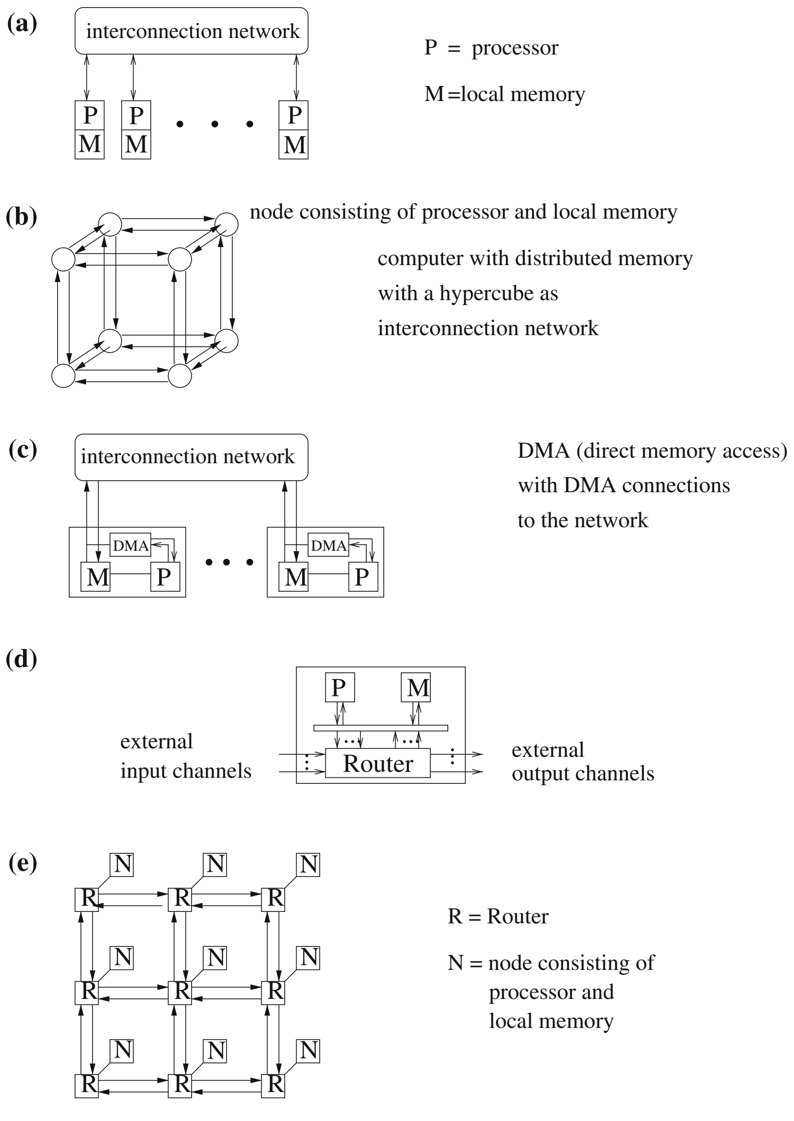

(a)
interconnection network
P
M
P = processor
M = local memory
(b)
node consisting of processor and local memory
computer with distributed memory with a hypercube as interconnection network
(c)
interconnection network
DMA
M
P
DMA (direct memory access) with DMA connections to the network
(d)
P
M
Router
external input channels
external output channels
(e)
N
R
R = Router
N = node consisting of processor and local memory

Fig. 2.3 Illustration of computers with distributed memory: (**a**) abstract structure, (**b**) computer with distributed memory and hypercube as interconnection structure, (**c**) DMA (direct memory access), (**d**) processor-memory node with router, and (**e**) interconnection network in form of a mesh to connect the routers of the different processor-memory nodes.

connected to a fixed set of other nodes by physical connections. The structure of the interconnection network can be represented as a graph structure. The nodes represent the processors, the edges represent the physical interconnections (also called *links*). Typically, the graph exhibits a regular structure. A typical network structure is the *hypercube* which is used in Fig. 2.3 b) to illustrate the node connections; a detailed description of interconnection structures is given in Sect. 2.5. In networks with point-to-point connection, the structure of the network determines the possible communications, since each node can only exchange data with its direct neighbor. To decouple send and receive operations, buffers can be used to store a message until the communication partner is ready. Point-to-point connections restrict parallel programming, since the network topology determines the possibilities for data exchange, and parallel algorithms have to be formulated, such that their communication fits to the given network structure [8, 130].

The execution of communication operations can be decoupled from the processor's operations by adding a **DMA controller** (DMA - direct memory access) to the nodes to control the data transfer between the local memory and the I/O controller. This enables data transfer from or to the local memory without participation of the processor (see Fig. 2.3 c) for an illustration) and allows asynchronous communication. A processor can issue a send operation to the DMA controller and can then continue local operations while the DMA controller executes the send operation. Messages are received at the destination node by its DMA controller which copies the enclosed data to a specific system location in local memory. When the processor then performs a receive operation, the data are copied from the system location to the specified receive buffer. Communication is still restricted to neighboring nodes in the network. Communication between nodes that do not have a direct connection must be controlled by software to send a message along a path of direct interconnections. Therefore, communication times between nodes that are not directly connected can be much larger than communication times between direct neighbors. Thus, it is still more efficient to use algorithms with communication according to the given network structure.

A further decoupling can be obtained by putting routers into the network, see Fig. 2.3 d). The routers form the actual network over which communication can be performed. The nodes are connected to the routers, see Fig. 2.3 e). Hardware-supported routing reduces communication times as messages for processors on remote nodes can be forwarded by the routers along a preselected path without interaction of the processors in the nodes along the path. With router support there is not a large difference in communication time between neighboring nodes and remote nodes, depending on the switching technique, see Sect. 2.6.3. Each physical I/O channel of a router can be used by one message only at a specific point in time. To decouple message forwarding, message buffers are used for each I/O channel to store messages and apply specific routing algorithms to avoid deadlocks, see also Sect. 2.6.1.

Technically, DMMs are quite easy to assemble since standard desktop computers can be used as nodes. The programming of DMMs requires a careful data layout, since each processor can directly access only its local data. Nonlocal data must be accessed via message passing, and the execution of the corresponding send and receive operations takes significantly longer than a local memory access. Depending on the interconnection network and the communication library used, the difference can be more than a factor of 100. Therefore, data layout may have a significant influence on the resulting parallel runtime of a program. The data layout should be selected, such that the number of message transfers and the size of the data blocks exchanged are minimized.

The structure of DMMs has many similarities with networks of workstations (NOWs) in which standard workstations are connected by a fast local area network (LAN). An important difference is that interconnection networks of DMMs are typically more specialized and provide larger bandwidths and lower latencies, thus leading to a faster message exchange.

Collections of complete computers with a dedicated interconnection network are often called **clusters**. Clusters are usually based on standard computers and even standard network topologies. The entire cluster is addressed and programmed as a single unit. The popularity of clusters as parallel machines comes from the availability of standard high-speed interconnections like FCS (Fibre Channel Standard), SCI (Scalable Coherent Interface), Switched Gigabit Ethernet, Myrinet, or InfiniBand, see [158, 94, 155]. A natural programming model of DMMs is the message-passing model that is supported by communication libraries like MPI or PVM, see Chapt. 5 for a detailed treatment of MPI. These libraries are often based on standard protocols like TCP/IP [125, 157].

The difference between cluster systems and **distributed systems** lies in the fact that the nodes in cluster systems use the same operating system and can usually not be addressed individually; instead a special job scheduler must be used. Several cluster systems can be connected to **grid systems** by using middleware software, such as the Globus Toolkit, see `http://www.globus.org` [65]. This allows a coordinated collaboration of several clusters. In grid systems, the execution of application programs is controlled by the middleware software.

Cluster systems are also used for the provision of services in the area of cloud computing. Using cloud computing, each user can allocate virtual resources which are provided via the cloud infrastructure as part of the cluster system. The user can dynamically allocate resources according to his or her computational requirements. Depending on the allocation, a virtual resource can be a single cluster node or a collection of cluster nodes. Examples for cloud infrastructures are the Amazon Elastic Compute Cloud (EC2) or the Windows Azure platform.

2.3.2 Computers with Shared Memory Organization

Computers with a physically shared memory are also called shared memory machines (SMMs); the shared memory is also called **global memory**. SMMs consist of a number of processors or cores, a shared physical memory (global memory) and an interconnection network to connect the processors with the memory. The shared memory can be implemented as a set of memory modules. Data can be exchanged between processors via the global memory by reading or writing shared variables. The cores of a multicore processor are an example for an SMM, see Sect. 2.4.3 for a more detailed description. Physically, the global memory usually consists of separate memory modules providing a common address space which can be accessed by all processors, see Fig. 2.4 for an illustration.

A natural programming model for SMMs is the use of **shared variables** which can be accessed by all processors. Communication and cooperation between the processors is organized by writing and reading shared variables that are stored in the global memory. Accessing shared variables concurrently by several processors should be avoided since **race conditions** with unpredictable effects can occur, see also Chaps. 3 and 6.

The existence of a global memory is a significant advantage, since communication via shared variables is easy and since no data replication is necessary as it is sometimes the case for DMMs. But technically, the realization of SMMs requires a larger effort, in particular because the interconnection network must provide fast access to the global memory for each processor. This can be ensured for a small number of processors, but scaling beyond a few dozen processors is difficult.

A special variant of SMMs are symmetric multiprocessors (SMPs). SMPs have a single shared memory which provides a uniform access time from any processor for all memory locations, i.e., all memory locations are equidistant to all processors [41, 94]. SMPs usually have a small number of processors that are connected via a central bus which also provides access to the shared memory. There are usually no private memories of processors or specific I/O processors, but each processor has a private cache hierarchy. As usual, access to a local cache is faster than access to the global memory. In the spirit of the definition from above, each multicore processor with several cores is an SMP system.

SMPs usually have only a small number of processors, since the central bus provides a constant bandwidth which is shared by all processors. When too many processors are connected, more and more access collisions may occur, thus increasing the effective memory access time. This can be alleviated by the use of caches and suitable cache coherence protocols, see Sect. 2.7.3. The maximum number of processors used in bus-based SMPs typically lies between 32 and 64.

Parallel programs for SMMs are often based on the execution of threads. A thread is a separate control flow which shares data with other threads via a global address space. It can be distinguished between **kernel threads** that are managed by the operating system, and **user threads** that are explicitly generated and controlled by the

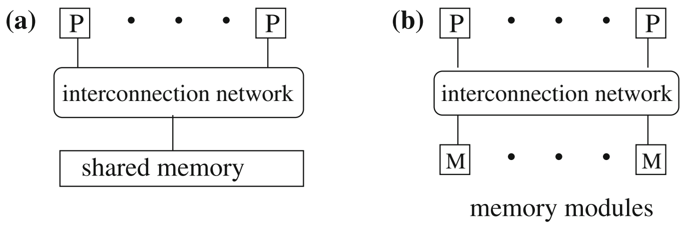


Fig. 2.4 Illustration of a computer with shared memory: (a) abstract view and (b) implementation of the shared memory with memory modules.

parallel program, see Section 3.8.2. The kernel threads are mapped by the operating system to processors for execution. User threads are managed by the specific programming environment used and are mapped to kernel threads for execution. The mapping algorithms as well as the exact number of processors can be hidden from the user by the operating system. The processors are completely controlled by the operating system. The operating system can also start multiple sequential programs from several users on different processors, when no parallel program is available. Small size SMP systems are often used as servers, because of their cost-effectiveness, see [41, 158] for a detailed description.

SMP systems can be used as nodes of a larger parallel computer by employing an interconnection network for data exchange between processors of different SMP nodes. For such systems, a shared address space can be defined by using a suitable cache coherence protocol, see Sect. 2.7.3. A coherence protocol provides the view of a shared address space, although the physical memory might be distributed. Such a protocol must ensure that any memory access returns the most recently written value for a specific memory address, no matter where this value is physically stored. The resulting systems are also called distributed shared memory (DSM) architectures. In contrast to single SMP systems, the access time in DSM systems depends on the location of a data value in the global memory, since an access to a data value in the local SMP memory is faster than an access to a data value in the memory of another SMP node via the coherence protocol. These systems are therefore also called NUMAs (nonuniform memory access), see Fig. 2.5. Since single SMP systems have a uniform memory latency for all processors, they are also called UMAs (uniform memory access).

2.3.3 Reducing memory access times

Memory access time has a large influence on program performance. This can also be observed for computer systems with a shared address space. The technological development with a steady reduction in the VLSI (Very-large-scale integration) feature size has led to significant improvements in processor performance. Since 1980, integer and floating-point performance on the SPEC benchmark suite has been increasing substantially per year, see Sect. 2.1. A significant contribution to these improvements comes from a reduction in processor cycle time. At the same time, the capacity of DRAM chips, which are used for building main memory, increased significantly: Between 1986 and 2003, the capacity of DRAM chips increased by about 60% per year; since 2003, the annual average increase lies between 25% and 40% [94].

For a performance evaluation of DRAM chips, the **latency** and the **bandwidth**, also denoted as **throughput**, are used. The latency of a DRAM chip is the total time that elapses until a memory access operation has been completely terminated. The latency is typically measured in microseconds or nanoseconds. The bandwidth denotes the number of data elements that can be read from a DRAM chip per time unit.

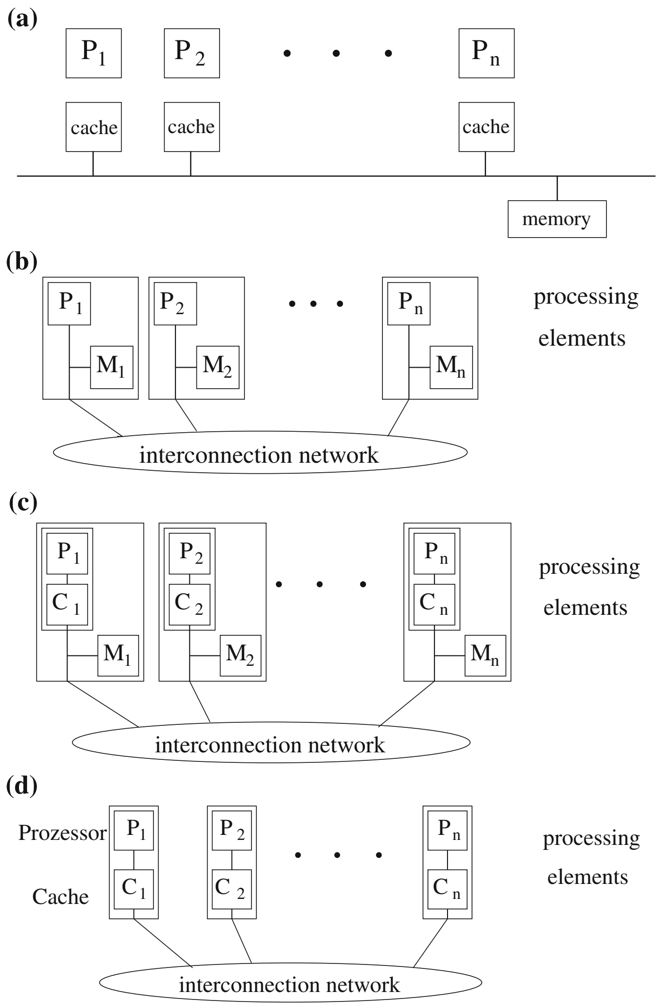


Fig. 2.5 Illustration of the architecture of computers with shared memory: **(a)** SMP – symmetric multiprocessors, **(b)** NUMA – nonuniform memory access, **(c)** CC-NUMA – cache coherent NUMA and **(d)** COMA – cache only memory access.

The bandwidth is typically measured in megabytes per second (MB/s) or gigabytes per second (GB/s). For the latency of DRAM chips, an average decrease of about 5% per year can be observed since 1980 [94]. In 2012, the latency of the newest DRAM technology (DDR3, *Double Data Rate*) lies between 24 and 30 ns. For the bandwidth of DRAM chips, an average increase of about 10% can be observed. For the DDR4 technology, which will be available in 2014, a bandwidth between 2100 MB/s and 3200 MB/s per DRAM chip is expected. Several DRAM chips (typically between 4

and 16) can be connected to DIMMs (*dual inline memory module*) to provide even larger bandwidths.
Considering DRAM latency, it can be observed that memory access time does not keep pace with processor performance improvement, and there is an increasing gap between processor cycle time and memory access time. A suitable organization of memory access becomes more and more important to get good performance results at program level. This is also true for parallel programs, in particular if a shared address space is used. Reducing the average latency observed by a processor when accessing memory can increase the resulting program performance significantly.

Two important approaches have been considered to reduce the average latency for memory access [14]: the simulation of **virtual processors** by each physical processor (multithreading) and the use of **local caches** to store data values that are accessed often. We give now a short overview of these approaches in the following.

2.3.3.1 Multithreading

The idea of **interleaved multithreading** is to hide the latency of memory accesses by simulating a fixed number of virtual processors for each physical processor. The physical processor contains a separate program counter (PC) as well as a separate set of registers for each virtual processor. After the execution of a machine instruction, an implicit switch to the next virtual processor is performed, i.e., the virtual processors are simulated by the physical processor in a round-robin fashion. The number of virtual processors per physical processor should be selected, such that the time between the execution of successive instructions of a virtual processor is sufficiently large to load required data from the global memory. Thus, the memory latency will be hidden by executing instructions of other virtual processors. This approach does not reduce the amount of data loaded from the global memory via the network. Instead, instruction execution is organized such that a virtual processor accesses requested data not before their arrival. Therefore, from the point of view of a virtual processor, the memory latency cannot be observed. This approach is also called **fine-grained multithreading**, since a switch is performed after each instruction. An alternative approach is **coarse-grained multithreading** which switches between virtual processors only on costly stalls, such as level 2 cache misses [94]. For the programming of fine-grained multithreading architectures, a PRAM-like programming model can be used, see Sect. 4.5.1. There are two drawbacks of fine-grained multithreading:

- The programming must be based on a large number of virtual processors. Therefore, the algorithm used must have a sufficiently large potential of parallelism to employ all virtual processors.
- The physical processors must be especially designed for the simulation of virtual processors. A software-based simulation using standard microprocessors is too slow.

There have been several examples for the use of fine-grained multithreading in the past, including Dencelor HEP (heterogeneous element processor) [180], NYU Ultracomputer [79], SB-PRAM [1], Tera MTA [41, 105], as well as the Sun T1 – T4 multiprocessors. For example, each T4 processor contains eight processor cores, each supporting eight threads which act as virtual processors. Section 2.4.1 will describe another variation of multithreading which is simultaneous multithreading.

2.3.3.2 Caches

A **cache** is a small, but fast memory between the processor and main memory. A cache can be used to store data that is often accessed by the processor, thus avoiding expensive main memory access. The data stored in a cache are always a subset of the data in main memory, and the management of the data elements in the cache is done by hardware, e.g., by employing a set-associative strategy, see [94] and Sect. 2.7.1 for a detailed treatment. For each memory access issued by the processor, it is first checked by hardware whether the memory address specified currently resides in the cache. If so, the data are loaded from the cache and no memory access is necessary. Therefore, memory accesses that go into the cache are significantly faster than memory accesses that require a load from the main memory. Since fast memory is expensive, several levels of caches are typically used, starting from a small, fast, and expensive level 1 (L1) cache over several stages (L2, L3) to the large, but slow main memory. For a typical processor architecture, access to the L1 cache only takes 2–4 cycles, whereas access to main memory can take up to several hundred cycles. The primary goal of cache organization is to reduce the average memory access time as far as possible and to achieve an access time as close as possible to that of the L1 cache. Whether this can be achieved depends on the memory access behavior of the program considered, see Sect. 2.7.

Caches are used for single-processor computers, but they also play an important role for SMPs and parallel computers with different memory organization. SMPs provide a shared address space. If shared data are used by multiple processors, it may be replicated in multiple caches to reduce access latencies. Each processor should have a coherent view to the memory system, i.e., any read access should return the most recently written value no matter which processor has issued the corresponding write operation. A coherent view would be destroyed if a processor p changes the value of a memory address in its local cache without writing this value back to main memory. If another processor q would later read this memory address, it would not get the most recently written value. But even if p writes the value back to main memory, this may not be sufficient if q has a copy of the same memory location in its local cache. In this case, it is also necessary to update the copy in the local cache of q. The problem of providing a coherent view to the memory system is often referred to as **cache coherence problem**. To ensure cache coherency, a **cache coherency protocol** must be used, see Sect. 2.7.3 and [41, 94, 89] for a more detailed description.

2.4 Thread-Level Parallelism

The architectural organization within a processor chip may require the use of explicitly parallel programs to efficiently use the resources provided. This is called **thread-level parallelism**, since the multiple control flows needed are often called threads. The corresponding architectural organization is also called **chip multiprocessing** (CMP). An example for CMP is the placement of multiple independent **execution cores** with all execution resources onto a single processor chip. The resulting processors are called **multicore processors**, see Sect. 2.4.3.

An alternative approach is the use of *multithreading* to execute multiple threads simultaneously on a single processor by switching between the different threads when needed by the hardware. As described in Sect. 2.3.3, this can be obtained by fine-grained or coarse-grained multithreadings. A variant of coarse-grained multithreading is **timeslice multithreading** in which the processor switches between the threads after a predefined timeslice interval has elapsed. This can lead to situations where the timeslices are not effectively used if a thread must wait for an event. If this happens in the middle of a timeslice, the processor may remain unused for the rest of the timeslice because of the waiting. Such unnecessary waiting times can be avoided by using **switch-on-event multithreading** [135] in which the processor can switch to the next thread if the current thread must wait for an event to occur as it can happen for cache misses.

A variant of this technique is **simultaneous multithreading** (SMT) which will be described in the following. This technique is called **hyperthreading** for some Intel processors. The technique is based on the observation that a single thread of control often does not provide enough instruction-level parallelism to use all functional units of modern superscalar processors.

2.4.1 Simultaneous Multithreading

The idea of simultaneous multithreading (SMT) is to use several threads and to schedule executable instructions from different threads in the same cycle if necessary, thus using the functional units of a processor more effectively. This leads to a simultaneous execution of several threads which gives the technique its name. In each cycle, instructions from several threads compete for the functional units of a processor. Hardware support for simultaneous multithreading is based on the replication of the chip area which is used to store the processor state. This includes the program counter (PC), user, and control registers as well as the interrupt controller with the corresponding registers. With this replication, the processor appears to the operating system and the user program as a set of **logical processors** to which processes or threads can be assigned for execution. These processes or threads can come from a single or several user programs. The number of replications of the processor state determines the number of logical processors.

Each logical processor stores its processor state in a separate processor resource. This avoids overhead for saving and restoring processor states when switching to another logical processor. All other resources of the processor chip like caches, bus system, and function and control units are shared by the logical processors. Therefore, the implementation of SMT only leads to a small increase in chip size. For two logical processors, the required increase in chip area for an Intel Xeon processor is less than 5% [135, 200]. The shared resources are assigned to the logical processors for simultaneous use, thus leading to a simultaneous execution of logical processors. When a logical processor must wait for an event, the resources can be assigned to another logical processor. This leads to a continuous use of the resources from the view of the physical processor. Waiting times for logical processors can occur for cache misses, wrong branch predictions, dependencies between instructions, and pipeline hazards.

Investigations have shown that the simultaneous use of processor resources by two logical processors can lead to performance improvements between 15% and 30%, depending on the application program [135]. Since the processor resources are shared by the logical processors, it cannot be expected that the use of more than two logical processors can lead to a significant additional performance improvement. Therefore, SMT will likely be restricted to a small number of logical processors. Examples of processors that support SMT are the Intel Core i3, i5, and i7 processors (two logical processors), the IBM Power7 processors (four logical processors) and the Sun/Oracle T4 processors (eight logical processors), see e.g., [94] for a more detailed description.

To use SMT to obtain performance improvements, it is necessary that the operating system is able to control logical processors. From the point of view of the application program, it is necessary that for each logical processor there is a separate thread available for execution. Therefore, the application program must apply parallel programming techniques to get performance improvements for SMT processors.

2.4.2 Energy Consumption of Processors

Until 2003, an average annual increase of the clock frequency of processors could be observed. This trend has stopped in 2003 at a clock frequency of about 3.3 GHz and no significant increase has occurred since then. A further increase of the clock frequency is difficult because of the increased heat production due to leakage currents. Such leakage currents also occur if the processor is not performing computations. Therefore, the resulting energy consumption is called *static energy consumption*; the energy consumption caused by computations is called *dynamic energy consumption*. In 2011, depending on the processor architecture, the static energy consumption typically contributed between 25% and 50% to the total energy consumption [94]. The heat produced by leakage currents must be carried away from the processor chip by using a sophisticated cooling technology.

An increase in clock frequency of a processor usually corresponds to a larger amount of leakage currents, leading to a larger energy consumption and an increased heat production. Models to capture this phenomenon describe the dynamic power consumption P_{dyn} of a processor by $P_{dyn} = \boldsymbol{\alpha} \cdot C_L \cdot V^2 \cdot f$ where $\boldsymbol{\alpha}$ is a switching probability, C_L is the load capacitance, V is the supply voltage and f is the clock frequency [113]. Since V depends linearly on f, a cubic dependence of the power consumption from the clock frequency results, i.e., the energy consumption increases significantly if the clock frequency is increased. This can be confirmed by looking at the history of processor development: The first 32-bit microprocessors had a power consumption of about 2 W, a recent 3.3 GHz Intel Core i7 (Nehalem) processor has a power consumption of about 130 W [94]. An increase in clock frequency which significantly exceeds 3.3 GHz would lead to an intolerable increase in energy consumption. The cubic dependency of the energy consumption on the clock frequency also explains why no significant increase in the clock frequency of desktop processors could be observed since 2003.

To reduce energy consumption, modern microprocessors use several techniques such as shutting down inactive parts of the processor chip as well as dynamic voltage and frequency scaling (DVFS). The idea of DVFS is to reduce the clock frequency of the processor chip to save energy during time periods with a small workload and to increase the clock frequency again if the workload increases again. Increasing the frequency reduces the cycle time of the processor, and in the same unit of time more instructions can be executed than for a smaller frequency. An example of a desktop microprocessor with DVFS capability is the Intel Core i7 processor (Sandy bridge) for which the clock frequency can be dynamically adjusted between 2.7 and 3.7 GHz, using frequency steps of 100 MHz. The dynamic adjustment can be performed by the operating system according to the current workload observed. Tools such as `cpufreq_set` can be used by the application programmer to adjust the clock frequency manually. For some processors, it is even possible to increase the frequency to a turbo mode, exceeding the maximum frequency available for a short period of time. This is also called *overclocking* and allows an especially fast execution of computations during time periods with a heavy workload.

2.4.3 Multicore Processors

According to Moore's law, the number of transistors of a processor chip doubles every 18–24 months. This enormous increase has enabled the hardware manufacturers for many years to provide a significant performance increase for application programs, see also Sect. 2.1. Thus, a typical computer is considered old fashioned and too slow after at most 5 years and customers buy new computers quite often. Hardware manufacturers are therefore trying to keep the obtained performance increase at least at the current level to avoid reduction in computer sales figures.

As discussed in Sect. 2.1, the most important factors for the performance increase per year have been an increase in clock speed and the internal use of parallel processing, like pipelined execution of instructions and the use of multiple functional units. But these traditional techniques have mainly reached their limits:

- Although it is possible to put additional functional units on the processor chip, this would not increase performance for most application programs, because dependencies between instructions of a single control thread inhibit their parallel execution. A single control flow does not provide enough instruction-level parallelism to keep a large number of functional units busy.
- There are two main reasons why the speed of processor clocks cannot be increased significantly [120]. First, the increase of the number of transistors on a chip is mainly achieved by increasing the transistor density. But this also increases the power density and heat production because of leakage current and power consumption, thus requiring an increased effort and more energy for cooling. Second, memory access times could not be reduced at the same rate as processor clock speed has been increased. This leads to an increased number of machine cycles for a memory access. For example, in 1990 main memory access has required between 6 and 8 cycles for a typical desktop computer system, whereas in 2012 memory access for an Intel Core i7 processor takes about 180 cycles. Therefore, memory access times could become a limiting factor for further performance increase, and cache memories are used to prevent this, see Sect. 2.7 for a further discussion. In future, it can be expected that the number of cycles needed for a memory access will not change significantly.

There are more problems that processor designers have to face: Using the increased number of transistors to increase the complexity of the processor architecture may also lead to an increase in processor-internal wire length to transfer control and data between the functional units of the processor. Here, the speed of signal transfers within the wires could become a limiting factor. For example, a 3 GHz processor has a cycle time of 0.33 ns. Assuming a signal transfer at the speed of light ($0.3 \cdot 10^9$m/s), a signal can cross a distance of $0.33 \cdot 10^{-9}$s $\cdot 0.3 \cdot 10^9$m/s $= 10$cm in one processor cycle. This is not significantly larger than the typical size of a processor chip, and wire lengths become an important issue.

Another problem is the following: The physical size of a processor chip limits the number of pins that can be used, thus limiting the bandwidth between CPU and main memory. This may lead to a processor-to-memory performance gap which is sometimes referred to as *memory wall*. This makes the use of high-bandwidth memory architectures with an efficient cache hierarchy necessary [17].

All these reasons inhibit a processor performance increase at the previous rate using the traditional techniques. Instead, new processor architectures have to be used, and the use of multiple cores on a single processor die is considered as the most promising approach. Instead of further increasing the complexity of the internal organization of a processor chip, this approach integrates multiple independent processing cores with a relatively simple architecture onto one processor chip. This has the additional advantage that the energy consumption of a processor chip can be reduced if necessary by switching off unused processor cores during idle times [92].

Multicore processors integrate multiple execution cores on a single processor chip. For the operating system, each execution core represents an independent logical processor with separate execution resources, such as functional units or execution pipelines. Each core has to be controlled separately, and the operating system can assign different application programs to the different cores to obtain a parallel execution. Background applications like virus checking, image compression, and encoding can run in parallel to application programs of the user. By using techniques of parallel programming, it is also possible to execute a computational-intensive application program (like computer games, computer vision, or scientific simulations) in parallel on a set of cores, thus reducing execution time compared to an execution on a single core or leading to more accurate results by performing more computations as in the sequential case. In the future, users of standard application programs as computer games will likely expect an efficient use of the execution cores of a processor chip. To achieve this, programmers have to use techniques from parallel programming.

The use of multiple cores on a single processor chip also enables standard programs, like text processing, office applications, or computer games, to provide additional features that are computed in the background on a separate core, so that the user does not notice any delay in the main application. But again, techniques of parallel programming have to be used for the implementation.

2.4.4 Architecture of Multicore Processors

There are many different design variants for multicore processors, differing in the number of cores, the structure and size of the caches, the access of cores to caches, and the use of heterogeneous components. From a high level view, three different types of architectures can be distinguished, and there are also hybrid organizations [121].

2.4.4.1 Hierarchical Design

For a hierarchical design, multiple cores share multiple caches. The caches are organized in a tree-like configuration, and the size of the caches increases from the leaves to the root, see Fig. 2.6 (left) for an illustration. The root represents the connection to external memory. Thus, each core can have a separate L1 cache and shares the L2 cache with other cores. All cores share the common external memory, resulting in a three-level hierarchy as illustrated in Fig. 2.6 (left). This can be extended to more levels. Additional sub-components can be used to connect the caches of one level with each other. A typical usage area for a hierarchical design are SMP configurations.

A hierarchical design is also often used for standard desktop or server processors.

An example for a hierarchical design is the Intel Core i7 quad-core processor, which contains four independent cores each of which can simulate two logical cores via hyperthreading. Each of the physical cores has a separate L1 cache (split into an instruction cache and a data cache) as well as a separate L2 cache; the L3 cache is shared by all cores of the processor chip. A more detailed description of the architecture of the Intel Core i7 will be given in Sect. 2.4.5. Other processors with a hierarchical design are the IBM Power 7 processors with a maximum number of eight cores per chip, where each core can simulate four threads via hyperthreading, and the AMD Opteron processors with up to eight cores per chip.

2.4.4.2 Pipelined Designs

For a pipelined design, data elements are processed by multiple execution cores in a pipelined way. Data elements enter the processor chip via an input port and are passed successively through different cores until the processed data elements leave the last core and the entire processor chip via an output port, see Fig. 2.6 (middle). Each core performs specific processing steps on each data element.

Pipelined designs are useful for application areas in which the same computation steps have to be applied to a long sequence of data elements. Network processors used in routers and graphics processors both perform this style of computations. Examples for network processors with a pipelined design are the Xelerator X10 and X11 processors [198, 121] for the successive processing of network packets in a pipelined way within the chip. The Xelerator X11 contains up to 800 separate cores which are arranged in a logically linear pipeline, see Fig. 2.7 for an illustration. The network packets to be processed enter the chip via multiple input ports on one side of the chip, are successively processed by the cores, and then exit the chip.

2.4.4.3 Network-based Design

For a network-based design, the cores of a processor chip and their local caches and memories are connected via an interconnection network with other cores of the chip,

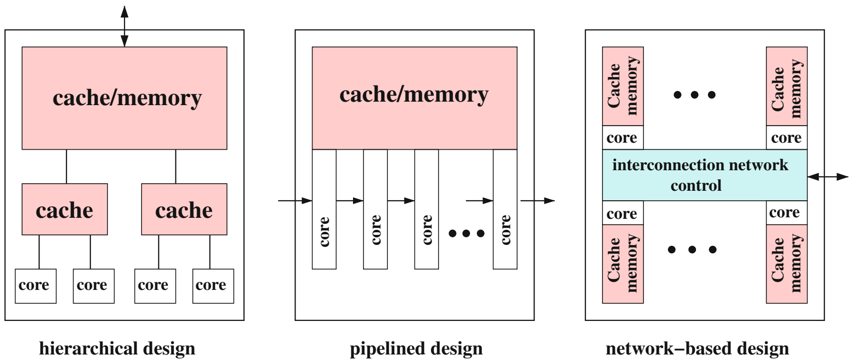


Fig. 2.6 Design choices for multicore chips according to [121].

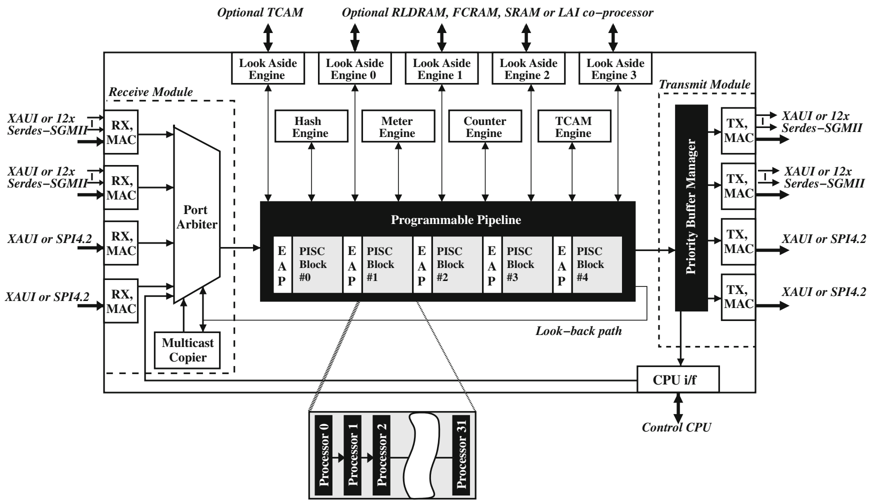


Fig. 2.7 Xelerator X11 Network Processor as an example for a pipelined design [198].

see Fig. 2.6 (right) for an illustration. Data transfer between the cores is performed via the interconnection network. This network may also provide support for the synchronization of the cores. Off-chip interfaces may be provided via specialized cores or DMA ports.

An example for a network-based design is the Sun Ultra SPARC T4 processor with eight cores, each of which can simulate eight threads by simultaneous multithreading (called CoolThreads by Sun), thus supporting the simultaneous execution of up to 64 threads. The cores are connected by a crossbar switch. A similar approach is used for the IBM BG/Q processors, see Sect. 2.8. Network-based designs have also been used for some research chips. Examples are the Intel Teraflops research chip which contained 80 cores connected by a 8×10 2D-mesh network [92, 17] and the Intel SCC (Single-chip Cloud Computer) containing 48 Pentium cores connected by a 4×6 2D-mesh network. Several Intel SCC chips have been produced and have been made publicly available for research projects exploring the scalability of multicore processors.

2.4.4.4 Future Trends and Developments

The potential of multicore processors has been realized by most processor manufacturers like Intel or AMD, and since about 2005, many manufacturers deliver processors with two or more cores. Since 2007, Intel and AMD provide quad-core processors (like the Quad-Core AMD Opteron and the Quad-Core Intel Xeon), and starting in 2010 the first oct-core processors were delivered.The Sun/Oracle SPARC T2 processor is available since 2012.

An important issue for the integration of a large number of cores in one processor chip is an efficient on-chip interconnection, which provides enough bandwidth for data transfers between the cores [92]. This interconnection should be *scalable* to support an increasing number of cores for future generations of processor designs and *robust* to tolerate failures of specific cores. If one or a few cores exhibit hardware failures, the rest of the cores should be able to continue operation. The interconnection should also support an efficient energy management which allows the scale-down of power consumption of individual cores by reducing the clock speed.

For an efficient use of processing cores, it is also important that the data to be processed can be transferred to the cores fast enough to avoid that the cores have to wait for the data to be available. Therefore, an efficient memory system and I/O system are important. The memory system may use private first-level (L1) caches which can only be accessed by their associated cores, as well as shared second-level (L2) caches which can contain data of different cores. In addition, a shared third-level (L3) cache is often used. Processor chip with dozens or hundreds of cores will likely require an additional level of caches in the memory hierarchy to fulfill bandwidth requirements [92]. The I/O system must be able to provide enough bandwidth to keep all cores busy for typical application programs. At the physical layer, the I/O system must be able to bring hundreds of gigabits per second onto the chip. Such powerful I/O systems are currently under development [92].

Table 2.1 gives a short overview of typical multicore processors in 2012. The size for the L1 cache given in the table is sizes of the L1 data cache. For the Intel processors, the IBM Power7 and Oracle SPARC T4 processors, L1 instruction caches of the same size as the data cache are additionally provided for each core. The AMD Bulldozer architecture additionally provides an L1 instruction cache of size 64 KB for each core. For a more detailed treatment of the architecture of multicore processors and further examples, we refer to [155, 94].

Table 2.1 Examples for multicore processors in 2012.

processor	number cores	number threads	clock GHz	L1 cache	L2 cache	L3 cache	year released
Intel Core i7 3770K ”Ivy Bridge”	4	8	3.5	4 x 32 KB	4 x 256 KB	8 MB	2012
Intel Xeon E5-2690 ”Sandy Bridge EP’	8	16	2.9	8 x 32 KB	8 x 256 MB	20 MB	2012
AMD Opteron 3280 ”Bulldozer”	8	8	2.4	8 x 16 KB	4 x 2 MB	8 MB	2012
AMD Opteron 6376 ”Piledriver”	16	16	2.3	16 x 16 KB	8 x 2 MB	2 x 8 MB	2012
IBM Power7	8	32	4.7	8 x 32 KB	8 x 256 KB	32 MB	2010
Oracle SPARC T4	8	64	3.0	8 x 16 KB	8 x 128 KB	4 MB	2011

2.4.5 Example: Architecture of the Intel Core i7

As example for the architecture of a typical desktop multicore processor, we give a short overview of the architecture of the Intel Core i7 processor. The Core i3, i5, and i7 processors have been introduced in 2008 as successor of the Core-2 processors. A more detailed description is given in [94, 112]. The Core i7 processor supports the Intel x86-64 architecture, which is a 64-bit extension of the x86-architecture that has been used by the Pentium processors. Since 2011, the Core i7 processors are based on the Sandy Bridge and Ivy Bridge microarchitectures. In addition to the normal processing cores and the memory hierarchy, the processor chip also contains a graphics processing unit, a memory controller, and a controller for the PCI express bus.

A Core i7 processor has two or four cores, each of which supports two simultaneous threads via hyperthreading. The internal organization of a single core is illustrated in Fig. 2.8. Each core can execute up to four x86 instructions in each machine cycle. An instruction fetch unit is responsible for the provision of instructions that are ready for execution. Branch prediction is used to predict the next instruction when conditional jump instructions are executed. The branch prediction uses a branch target buffer to store the target addresses of previous jumps. Based on the predicted target address, the instruction fetch unit loads 16 bytes into a predecode instruction buffer. To store instructions, a 32 KB L1 instruction cache (block size 64 bytes, eight-way associative) as well as a special micro-op cache, see below, is used.

A predecoder is used to decompose the 16 bytes in the predecode instruction buffer into x86 instructions, which are then stored into an instruction queue with 18 entries. The x86 instructions may differ significantly in their complexity and may also have different length. Before their execution, the x86 instructions are transformed into *micro-ops* with constant length. Four decode units are available for this transformation: Three of these decode units are dedicated to the transformation of x86 instructions that can directly be translated into a single micro-op. One decode unit is responsible for the transformation of complex x86 instructions for which a sequence of micro-ops must be generated. This decode unit can generate up to four micro-ops per machine cycle. The generated micro-ops are stored in a special micro-op cache in the order of the original x86 instructions.

The processor also contains a 1536-entry *micro-ops cache*, which contains a part of the instruction cache in the form of already decoded micro-ops. The micro-ops cache is eight-way associative and each block contains six micro-ops. If the address of the next instruction provided by the branch prediction can be found in the micro-op cache (cache hit), a maximum number of three cache blocks with neighboring instructions is loaded into a special buffer. In this case, the standard fetch-decode hardware with the four decoders described above is not needed, so that the decoders can produce more decoded micro-ops for the micro-ops cache.

The micro-ops provided by the decoders or the micro-ops cache are stored in a 28-entry decoder queue. Depending on the program to be executed, the decoder queue may store all instructions of a small loop. In this case, there is no need to load

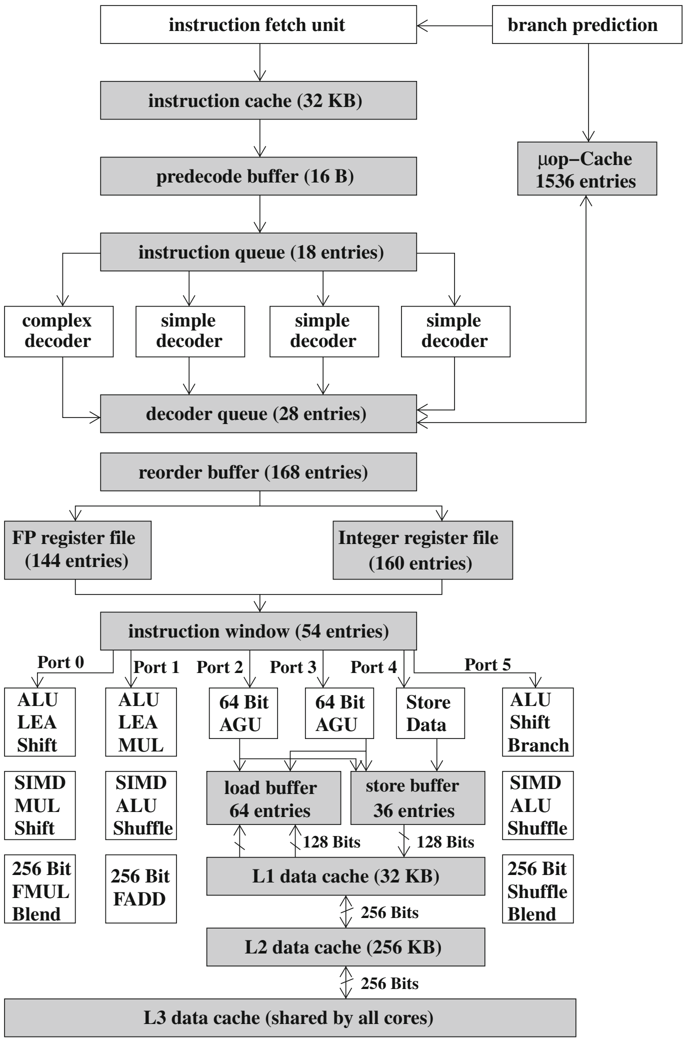


Fig. 2.8 Block diagram to illustrate the internal architecture of one core of an Intel Core i7 processor (Sandy Bridge).

new instructions during the execution of this loop and the decoder hardware can be deactivated, which reduces the energy consumption.

The register reordering used is based on a register file in which 160 64-bit integer values can be stored. Additionally there are 144 256-bit entries for floating-point values available to support the execution of SIMD AVX instructions, see Sect. 3.4. A centralized 54-entry instruction window is used for the mapping of micro-ops to function units. An instruction that is ready for execution can be assigned to a suitable function unit. In each machine cycle, the hardware is able to assign up to six instructions to function units, and four instructions can be terminated in one cycle.

The actual execution of instructions is performed by the function units, which are accessed via three execution ports (Port 0, Port 1, and Port 5). Each of these execution ports allows the access to function units for different types of instructions: integer instructions, SIMD integer instructions, and floating-point instructions (scalar or SIMD). If an instruction executed by a function unit provides a result, this result is stored in the destination register specified by the instruction. In addition to the three execution ports, three memory access ports (Port 2, Port 3, and Port 4) are available. At ports 2 and 3, general address generation units (AGUs) are available, which can be used for memory load and store operations. At port 4, a unit for storing values into the memory is available. A *load buffer* with 64 entries is used to store values that have been loaded from memory. Additionally, a 36-entry *store buffer* is available for values to be stored into memory.

The memory hierarchy of the Core i7 includes a three-level cache-hierarchy, consisting of a L1 cache, a L2 cache, and a L3 cache. Each physical core has a separate L1 and L2 cache, whereas the L3 cache is shared by all cores of the processor. The L2 data cache has size 32 KB and is eight-way associative. The block size is 64 bytes and a write-back replacement policy is used. The access latency is four cycles for integer values and 5–6 cycles for floating-point values or SIMD values. In each cycle, at most 128 bits can be loaded from the L1 cache or can be stored into the L1 cache.

The L2 cache has size 256 KB, is eight-way associative, and uses a write-back replacement policy. The access latency is 12 cycles. The L3 cache is share by all cores and the graphics unit of the processor. Access to the L3 cache is performed via a special ring network, consisting of four sub-rings for request, acknowledge, cache coherency, and data (32 bytes). The size of the L3 cache is scalable and depends on the specific processor version. In 2012, a typical size is 2 MB for each core. The L3 cache is 16-way associative and the access latency lies between 26 and 31 cycles. Table 2.2 summarizes important characteristics of the cache hierarchy. A detailed analysis of the performance and the memory system of the i7 architecture is provided in [94].

Table 2.2 Summary of important characteristics of the cache hierarchy of the Intel Core i7 processor (Sandy Bridge).

characteristic	L1	L2	L3
size	32 KB instructions 32 KB data	256 KB	2MB per core
associativity	4-way instructions 8-way data	8-way	16-way
access latency	4-6 cycles	12 cycles	26-31 cycles
replacement policy	write-back	write-back	write-back

2.5 Interconnection Networks

A physical connection between the different components of a parallel system is provided by an **interconnection network** . Similar to control flow and data flow, see Sect. 2.2, or memory organization, see Sect. 2.3, the interconnection network can also be used for a classification of parallel systems. Internally, the network consists of links and switches which are arranged and connected in some regular way. In multicomputer systems, the interconnection network is used to connect the processors or nodes with each other. Interactions between the processors for coordination, synchronization, or exchange of data are obtained by communication through message passing over the links of the interconnection network. In multiprocessor systems, the interconnection network is used to connect the processors with the memory modules. Thus, memory accesses of the processors are performed via the interconnection network.

In both cases, the main task of the interconnection network is to transfer a message from a specific processor to a specific destination. The message may contain data or a memory request. The destination may be another processor or a memory module. The requirement for the interconnection network is to perform the message transfer correctly as fast as possible, even if several messages have to be transferred at the same time. Message transfer and memory accesses represent a significant part of operations of parallel systems with a distributed or shared address space. Therefore, the interconnection network used represents a significant part of the design of a parallel system and may have a large influence on its performance. Important design criteria of networks are:

- the **topology** describing the interconnection structure used to connect different processors or processors and memory modules and
- the **routing technique** describing the exact message transmission used within the network between processors or processors and memory modules.

The topology of an interconnection network describes the geometric structure used for the arrangement of switches and links to connect processors or processors and memory modules. The geometric structure can be described as a graph in which switches, processors, or memory modules are represented as vertices and physical links are represented as edges. It can be distinguished between *static* and *dynamic* interconnection networks. **Static interconnection networks** connect nodes (processors or memory modules) *directly* with each other by fixed physical links. They are also called **direct networks** or **point-to-point networks**. The number of connections to or from a node may vary from only one in a star network to the total number of nodes in the network for a completely connected graph, see Sect. 2.5.2. Static networks are often used for systems with a distributed address space where a node comprises a processor and the corresponding memory module. **Dynamic interconnection networks** connect nodes *indirectly* via switches and links. They are also called **indirect networks**. Examples of indirect networks are *bus-based networks* or *switching networks* which consist of switches connected by links. Dynamic networks are used for both parallel systems with distributed and shared address space. Often, hybrid strategies are used [41].

The routing technique determines *how* and *along which path* messages are transferred in the network from a sender to a receiver. A path in the network is a series of nodes along which the message is transferred. Important aspects of the routing technique are the **routing algorithm** which determines the path to be used for the transmission and the **switching strategy** which determines whether and how messages are cut into pieces, how a routing path is assigned to a message, and how a message is forwarded along the processors or switches on the routing path.

The combination of routing algorithm, switching strategy, and network topology determines the performance of a network significantly. In Sects. 2.5.2 and 2.5.4, important direct and indirect networks are described in more detail. Specific routing algorithms and switching strategies are presented in Sects. 2.6.1 and 2.6.3. Efficient algorithms for the realization of common communication operations on different static networks are given in Chap. 4. A more detailed treatment of interconnection networks is given in [19, 41, 50, 81, 105, 130, 177].

2.5.1 Properties of Interconnection Networks

Static interconnection networks use fixed links between the nodes. They can be described by a connection graph $G = (V, E)$ where V is a set of nodes to be connected and E is a set of direct connection links between the nodes. If there is a direct physical connection in the network between the nodes $u \in V$ and $v \in V$, then it is $(u, v) \in E$. For most parallel systems, the interconnection network is *bidirectional*. This means that along a physical link messages can be transferred in both directions at the same time. Therefore, the connection graph is usually defined as an undirected graph. When a message must be transmitted from a node u to a node v and there is no direct connection between u and v in the network, a path from u to v must be selected which consists of several intermediate nodes along which the message is transferred. A sequence of nodes $(v_0, \ldots, v_k)$ is called *path* of length k between v_0 and v_k, if $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. For parallel systems, all interconnection networks fulfill the property that there is at least one path between any pair of nodes $u, v \in V$.

Static networks can be characterized by specific properties of the connection graph, including the following properties: number of nodes, diameter of the network, degree of the nodes, bisection bandwidth, node and edge connectivity of the network, and flexibility of embeddings into other networks as well as the embedding of other networks. In the following, a precise definition of these properties is given.

The **diameter** $\boldsymbol{\delta}(G)$ of a network G is defined as the maximum distance between any pair of nodes:

$$\delta(G) = \max_{u,v \in V} \min_{\substack{\varphi \text{ path} \\ \text{from } u \text{ to } v}} \{k \mid k \text{ is the length of the path } \varphi \text{ from } u \text{ to } v\}.$$

The diameter of a network determines the length of the paths to be used for message transmission between any pair of nodes. The **degree** $d(G)$ of a network G is the maximum degree of a node of the network. The degree of a node n is the number of direct neighbor nodes of n:

$$g(G) = max\{g(v) \mid g(v)\text{degree of } v \in V\}.$$

In the following, we assume that $|A|$ denotes the number of elements in a set A. The **bisection bandwidth** $B(G)$ of a network G is defined as the minimum number of edges that must be removed to partition the network into two parts of equal size without any connection between the two parts. For an uneven total number of nodes, the size of the parts may differ by 1. This leads to the following definition for $B(G)$:

$$B(G) = \min_{\substack{U_1, U_2 \text{ partition of } V \\ ||U_1| - |U_2|| \leq 1}} |\{(u, v) \in E \mid u \in U_1, v \in U_2\}|.$$

$B(G)+1$ messages can saturate a network G, if these messages must be transferred at the same time over the corresponding edges. Thus, bisection bandwidth is a measure for the capacity of a network when transmitting messages simultaneously.

The **node** and **edge connectivity** of a network measure the number of nodes or edges that must fail to disconnect the network. A high connectivity value indicates a high reliability of the network and is therefore desirable. Formally, the node connectivity of a network is defined as the minimum number of nodes that must be deleted to disconnect the network, i.e., to obtain two unconnected network parts (which do not necessarily need to have the same size as it is required for the bisection bandwidth). For an exact definition, let $G_{V\setminus M}$ be the rest graph which is obtained by deleting all nodes in $M \subset V$ as well as all edges adjacent to these nodes. Thus, it is $G_{V\setminus M} = (V \setminus M, E \cap ((V \setminus M) \times (V \setminus M)))$. The node connectivity $nc(G)$ of G is then defined as

$$nc(G) = \min_{M \subset V} \{|M| \mid \text{there exist } u, v \in V \setminus M \text{, such that there exists}$$
$$\text{no path in } G_{V\setminus M} \text{ from } u \text{ to } v\}.$$

Similarly, the edge connectivity of a network is defined as the minimum number of edges that must be deleted to disconnect the network. For an arbitrary subset $F \subset E$, let $G_{E\setminus F}$ be the rest graph which is obtained by deleting the edges in F, i.e., it is $G_{E\setminus F} = (V, E \setminus F)$. The edge connectivity $ec(G)$ of G is then defined as

$$ec(G) = \min_{F \subset E} \{|F| \mid \text{there exist } u, v \in V \text{, such that there exists}$$
$$\text{no path in } G_{E\setminus F} \text{ from } u \text{ to } v\}.$$

The node and edge connectivity of a network is a measure for the number of independent paths between any pair of nodes. A high connectivity of a network is important

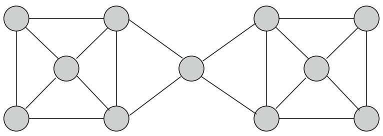


Fig. 2.9 Network with node connectivity 1, edge connectivity 2, and degree 4. The smallest degree of a node is 3.

for its availability and reliability, since many nodes or edges can fail before the network is disconnected. The minimum degree of a node in the network is an upper bound on the node or edge connectivity, since such a node can be completely separated from its neighboring nodes by deleting all incoming edges. Figure 2.9 shows that the node connectivity of a network can be smaller than its edge connectivity.

The flexibility of a network can be captured by the notion of **embedding**. Let $G = (V, E)$ and $G' = (V', E')$ be two networks. An embedding of G' into G assigns each node of G' to a node of G, such that different nodes of G' are mapped to different nodes of G and such that edges between two nodes in G' are also present between their associated nodes in G [19]. An embedding of G' into G can formally be described by a mapping function $\sigma : V' \to V$ such that the following holds:

- if $u \neq v$ for $u, v \in V'$, then $\sigma(u) \neq \sigma(v)$ and
- if $(u, v) \in E'$, then $(\sigma(u), \sigma(v)) \in E$.

If a network G' can be embedded into a network G, this means that G is at least as flexible as G', since any algorithm that is based on the network structure of G', e.g., by using edges between nodes for communication, can be re-formulated for G with the mapping function σ, thus using corresponding edges in G for communication.

The network of a parallel system should be designed to meet the requirements formulated for the architecture of the parallel system based on typical usage patterns. Generally, the following topological properties are desirable:

- a small diameter to ensure small distances for message transmission,
- a small node degree to reduce the hardware overhead for the nodes,
- a large bisection bandwidth to obtain large data throughputs,
- a large connectivity to ensure reliability of the network,
- embedding into a large number of networks to ensure flexibility, and
- easy extendability to a larger number of nodes.

Some of these properties are conflicting and there is no network that meets all demands in an optimal way. In the following, some popular direct networks are presented and analyzed. The topologies are illustrated in Fig. 2.10. The topological properties are summarized in Table 2.3.

2.5.2 *Direct Interconnection Networks*

Direct interconnection networks usually have a regular structure which is transferred to their graph representation $G = (V, E)$. In the following, we use $n = |V|$ for

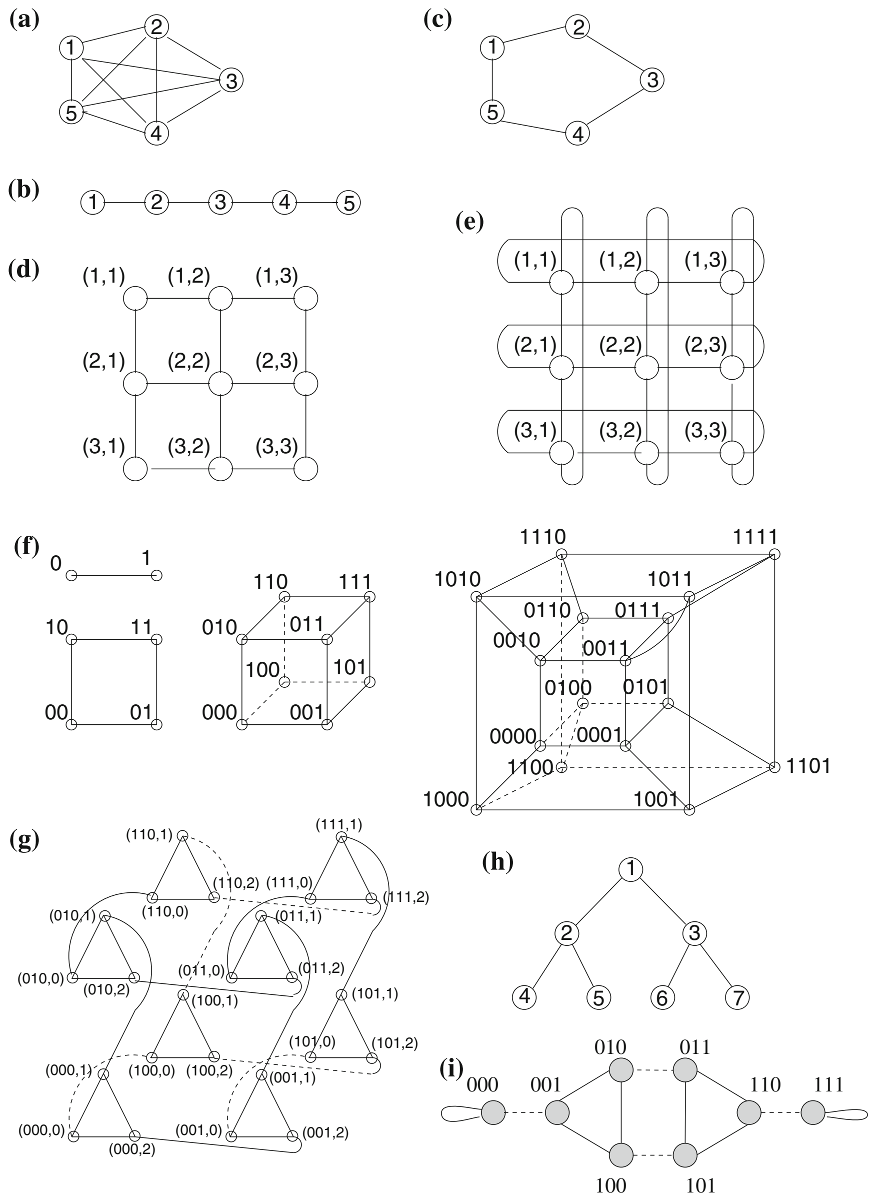


Fig. 2.10 Static interconnection networks: **(a)** complete graph, **(b)** linear array, **(c)** ring, **(d)** 2-dimensional mesh **(e)** 2-dimensional torus, **(f)** k-dimensional cube for k=1,2,3,4, **(g)** Cube-connected cycles network for k=3, **(h)** complete binary tree, **(i)** shuffle-exchange network with 8 nodes, where dashed edges represent exchange edges and straight edges represent shuffle edges.

Table 2.3 Summary of important characteristics of static interconnection networks for selected topologies.

network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity ec(G)	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\left\lfloor \frac{n}{2} \right\rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d\left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hyper-cube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dimensional CCC-network ($n = k2^k$ for $k \geq 3$)	3	$2k-1+\lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
complete binary tree ($n = 2^k - 1$)	3	$2\log\frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d\left\lfloor \frac{k}{2} \right\rfloor$	$2d$	$2k^{d-1}$

the number of nodes in the network and use this as a parameter of the network type considered. Thus, each network type captures an entire class of networks instead of a fixed network with a given number of nodes.

A **complete graph** is a network G in which each node is directly connected with every other node, see Fig. 2.10 (a). This results in diameter $\delta(G) = 1$ and degree $g(G) = n - 1$. The node and edge connectivity is $nc(G) = ec(G) = n - 1$, since a node can only be disconnected by deleting all $n - 1$ adjacent edges or neighboring nodes. For even values of n, the bisection bandwidth is $B(G) = n^2/4$: if two subsets of nodes of size $n/2$ each are built, there are $n/2$ edges from each of the nodes of one subset into the other subset, resulting in $n/2 \cdot n/2$ edges between the subsets. All other networks can be embedded into a complete graph, since there is a connection between any two nodes. Because of the large node degree, complete graph networks can only be built physically for a small number of nodes.

In a **linear array network**, nodes are arranged in a sequence and there is a bidirectional connection between any pair of neighboring nodes, see Fig. 2.10 (b), i.e., it is $V = \{v_1, \ldots, v_n\}$ and $E = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. Since $n - 1$ edges have to be traversed to reach v_n starting from v_1, the diameter is $\delta(G)0 = n - 1$. The connectivity is $nc(G) = ec(G) = 1$, since the elimination of one node or edge disconnects the network. The network degree is $g(G) = 2$ because of the inner nodes, and the bisection bandwidth is $B(G) = 1$. A linear array network can be embedded in nearly all standard networks except a tree network, see below. Since there is a link only between neighboring nodes, a linear array network does not provide fault tolerance for message transmission.

In a **ring network**, nodes are arranged in ring order. Compared to the linear array network, there is one additional bidirectional edge from the first node to the last node, see Fig. 2.10 (c). The resulting diameter is $\delta(G) = \lfloor n/2 \rfloor$, the degree is $g(G) = 2$, the connectivity is $nc(G) = ec(G) = 2$, and the bisection bandwidth is also $B(G) = 2$. In practice, ring networks can be used for small number of processors and as part of more complex networks.

A d**-dimensional mesh** (also called d**-dimensional array**) for $d \geq 1$ consists of $n = n_1 \cdot n_2 \cdot \ldots \cdot n_d$ nodes that are arranged as a d-dimensional mesh, see Fig. 2.10 (d). The parameter n_j denotes the extension of the mesh in dimension j for $j = 1, \ldots, d$. Each node in the mesh is represented by its position $(x_1, \ldots, x_d)$ in the mesh with $1 \leq x_j \leq n_j$ for $j = 1, \ldots, d$. There is an edge between node $(x_1, \ldots, x_d)$ and $(x'_1, \ldots x'_d)$, if there exists $\mu \in \{1, \ldots, d\}$ with

$$|x_\mu - x'_\mu| = 1 \text{ and } x_j = x'_j \text{ for all } j \neq \mu.$$

In the case that the mesh has the same extension in all dimensions (also called *symmetric mesh*), i.e., $n_j = r = \sqrt[d]{n}$ for all $j = 1, \ldots, d$, and therefore $n = r^d$, the network diameter is $\delta(G) = d \cdot (\sqrt[d]{n} - 1)$, resulting from the path length between nodes on opposite sides of the mesh. The node and edge connectivity is $nc(G) = ec(G) = d$, since the corner nodes of the mesh can be disconnected by deleting all d incoming edges or neighboring nodes. The network degree is $g(G) = 2d$, resulting from inner mesh nodes which have two neighbors in each dimension. A two-dimensional mesh has been used for the Teraflop processor from Intel, see Sect. 2.4.4.

A d**-dimensional torus** is a variation of a d-dimensional mesh. The differences are additional edges between the first and the last nodes in each dimension, i.e., for each dimension $j = 1, \ldots, d$ there is an edge between node $(x_1, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_d)$ and $(x_1, \ldots, x_{j-1}, n_j, x_{j+1}, \ldots, x_d)$ see Fig. 2.10 (e). For the symmetric case $n_j = \sqrt[d]{n}$ for all $j = 1, \ldots, d$, the diameter of the torus network is $\delta(G) = d \cdot \lfloor \sqrt[d]{n}/2 \rfloor$. The node degree is $2d$ for each node, i.e., $g(G) = 2d$. Therefore, node and edge connectivity are also $nc(G) = ec(G) = 2d$.

Torus networks have often been used for the implementation of large parallel systems. Examples are the IBM BlueGene systems, where the BG/L and BG/P systems used a 3D torus and the newer BG/Q systems use a 5D torus network as central interconnect. Torus networks have also been used for the Cray XT3, XT4, and XT5 systems (3D torus) as well as for the Tofu interconnect of the Fujitsu K computer (6D torus).

A k**-dimensional cube** or **hypercube** consists of $n = 2^k$ nodes which are connected by edges according to a recursive construction, see Fig. 2.10 (f). Each node is represented by a binary word of length k, corresponding to the numbers $0, \ldots, 2^k - 1$. A one-dimensional cube consists of two nodes with bit representations 0 and 1 which are connected by an edge. A k-dimensional cube is constructed from two given $(k-1)$-dimensional cubes, each using binary node representations $0, \ldots, 2^{k-1} - 1$. A k-dimensional cube results by adding edges between each pair of nodes with the same binary representation in the two $(k-1)$-dimensional cubes. The binary representations of the nodes in the resulting k-dimensional cube are obtained by adding a leading 0 to the previous representation of the first $(k-1)$-dimensional cube and adding a leading 1 to the previous representations of the second $(k-1)$-dimensional cube. Using the binary representations of the nodes $V = \{0, 1\}^k$, the recursive construction just mentioned implies that there is an edge between node $\boldsymbol{\alpha}_0 \ldots \boldsymbol{\alpha}_j \ldots \boldsymbol{\alpha}_{k-1}$ and node $\boldsymbol{\alpha}_0 \ldots \bar{\boldsymbol{\alpha}_j} \ldots \boldsymbol{\alpha}_{k-1}$ for $0 \leq j \leq k-1$ where $\bar{\boldsymbol{\alpha}_j} = 1$ for $\boldsymbol{\alpha}_j = 0$ and $\bar{\boldsymbol{\alpha}_j} = 0$ for $\boldsymbol{\alpha}_j = 1$. Thus, there is an edge between every pair of nodes whose binary representation differs in exactly one bit position. This fact can also be captured by the Hamming distance.

The **Hamming distance** of two binary words of the same length is defined as the number of bit positions in which their binary representations differ. Thus, two nodes of a k-dimensional cube are directly connected, if their Hamming distance is 1. Between two nodes $v, w \in V$ with Hamming distance d, $1 \leq d \leq k$, there exists a path of length d connecting v and w. This path can be determined by traversing the bit representation of v bitwise from left to right and inverting the bits in which v and w differ. Each bit inversion corresponds to a traversal of the corresponding edge to a neighboring node. Since the bit representation of any two nodes can differ in at most k positions, there is a path of length $\leq k$ between any pair of nodes. Thus, the diameter of a k-dimensional cube is $\boldsymbol{\delta}(G) = k$. The node degree is $g(G) = k$, since a binary representation of length k allows k bit inversions, i.e., each node has exactly k neighbors. The node and edge connectivity is $nc(G) = ec(G) = k$ as will be described in the following.

The connectivity of a hypercube is at most k, i.e., $nc(G) \leq k$, since each node can be completely disconnected from its neighbors by deleting all k neighbors or all k adjacent edges. To show that the connectivity is at least k, we show that there are exactly k independent paths between any pair of nodes v and w. Two paths are independent of each other if they do not share any edge, i.e., independent paths between v and w only share the two nodes v and w. The independent paths are constructed based on the binary representations of v and w, which are denoted by A and B, respectively, in the following. We assume that A and B differ in l positions, $1 \leq l \leq k$, and that these are the first l positions (which can be obtained by a renumbering). We can construct l paths of length l each between v and w by inverting the first l bits of A in different orders. For path i, $0 \leq i < l$, we stepwise invert bits $i, \ldots, l-1$ in this order first, and then invert bits $0, \ldots, i-1$ in this order. This results in l independent paths. Additional $k-l$ independent paths between v and w of length $l+2$ each can be constructed as follows: For i with $0 \leq i < k-l$, we first invert the bit $(l+i)$ of A and then the bits at positions $0, \ldots, l-1$ stepwise. Finally, we invert the bit $(l+i)$ again, obtaining bit representation B. This is shown in Fig. 2.11 for an example. All k paths constructed are independent from each other, showing that $nc(G) \geq k$ holds.

A k-dimensional cube allows the embedding of many other networks as will be shown in the next subsection.

A **cube-connected cycles** (CCC) network results from a k-dimensional cube by replacing each node with a cycle of k nodes. Each of the nodes in the cycle has

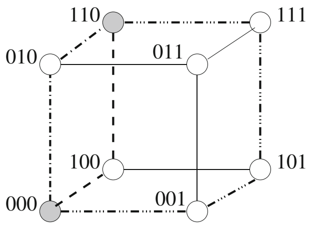


Fig. 2.11 In a three-dimensional cube network, we can construct three independent paths (from node 000 to node 110). The Hamming distance between node 000 and node 110 is $l = 2$. There are two independent paths between 000 and 110 of length $l = 2$: path (000, 100, and 110) and path (000, 010, and 110). Additionally, there are $k - l = 1$ paths of length $l + 2 = 4$: path (000, 001, 101, 111, and 110).

one off-cycle connection to one neighbor of the original node of the k-dimensional cube, thus covering all neighbors, see Fig. 2.10 (g). The nodes of a CCC network can be represented by $V = \{0, 1\}^k \times \{0, \ldots, k-1\}$ where $\{0, 1\}^k$ are the binary representations of the k-dimensional cube and $i \in \{0, \ldots, k-1\}$ represents the position in the cycle. It can be distinguished between cycle edges F and cube edges E:

$$F = \{((\boldsymbol{\alpha}, i), (\boldsymbol{\alpha}, (i+1) \bmod k)) \mid \boldsymbol{\alpha} \in \{0, 1\}^k, 0 \le i < k\},$$
$$E = \{((\boldsymbol{\alpha}, i), (\boldsymbol{\beta}, i)) \mid \boldsymbol{\alpha}_i \ne \boldsymbol{\beta}_i \text{ and } \boldsymbol{\alpha}_j = \boldsymbol{\beta}_j \text{ for } j \ne i\}.$$

Each of the $k \cdot 2^k$ nodes of the CCC network has degree $g(G) = 3$, thus eliminating a drawback of the k-dimensional cube. The connectivity is $nc(G) = ec(G) = 3$ since each node can be disconnected by deleting its three neighboring nodes or edges. An upper bound for the diameter is $\delta(G) = 2k - 1 + \lfloor k/2 \rfloor$. To construct a path of this length, we consider two nodes in two different cycles with maximum hypercube distance k. These are nodes $(\boldsymbol{\alpha}, i)$ and $(\boldsymbol{\beta}, j)$ for which $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ differ in all k bits. We construct a path from $(\boldsymbol{\alpha}, i)$ to $(\boldsymbol{\beta}, j)$ by sequentially traversing a cube edge and a cycle edge for each bit position. The path starts with $(\boldsymbol{\alpha}_0 \ldots \boldsymbol{\alpha}_i \ldots \boldsymbol{\alpha}_{k-1}, i)$ and reaches the next node by inverting $\boldsymbol{\alpha}_i$ to $\bar{\boldsymbol{\alpha}_i} = \boldsymbol{\beta}_i$. From $(\boldsymbol{\alpha}_0 \ldots \boldsymbol{\beta}_i \ldots \boldsymbol{\alpha}_{k-1}, i)$ the next node $(\boldsymbol{\alpha}_0 \ldots \boldsymbol{\beta}_i \ldots \boldsymbol{\alpha}_{k-1}, (i+1) \bmod k)$ is reached by using a cycle edge. In the next steps, the bits $\boldsymbol{\alpha}_{i+1}, \ldots, \boldsymbol{\alpha}_{k-1}$ and $\boldsymbol{\alpha}_0, \ldots, \boldsymbol{\alpha}_{i-1}$ are successively inverted in this way, using a cycle edge between the steps. This results in $2k-1$ edge traversals. Using at most $\lfloor k/2 \rfloor$ additional traversals of cycle edges starting from $(\boldsymbol{\beta}, i+k-1 \bmod k)$ leads to the target node $(\boldsymbol{\beta}, j)$.

A **complete binary tree** network has $n = 2^k - 1$ nodes which are arranged as a binary tree in which all leaf nodes have the same depth, see Fig. 2.10 (h). The degree of inner nodes is 3, leading to a total degree of $g(G) = 3$. The diameter of the network is $\delta(G) = 2 \cdot \log \frac{n+1}{2}$ and is determined by the path length between two leaf nodes in different subtrees of the root node; the path consists of a subpath from the first leaf to the root followed by a subpath from the root to the second leaf.

The connectivity of the network is $nc(G) = ec(G) = 1$, since the network can be disconnected by deleting the root or one of the edges to the root.

A k-dimensional **shuffle-exchange** network has $n = 2^k$ nodes and $3 \cdot 2^{k-1}$ edges [187]. The nodes can be represented by k-bit words. A node with bit representation $\boldsymbol{\alpha}$ is connected with a node with bit representation $\boldsymbol{\beta}$, if

- $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ differ in the last bit (*exchange edge*) or
- $\boldsymbol{\alpha}$ results from $\boldsymbol{\beta}$ by a cyclic left shift or a cyclic right shift (*shuffle edge*).

Figure 2.10 (i) shows a shuffle-exchange network with 8 nodes. The permutation $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ where $\boldsymbol{\beta}$ results from $\boldsymbol{\alpha}$ by a cyclic left shift is called **perfect shuffle**. The permutation $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ where $\boldsymbol{\beta}$ results from $\boldsymbol{\alpha}$ by a cyclic right shift is called **inverse perfect shuffle**, see [130] for a detailed treatment of shuffle-exchange networks.

A k**-ary** d**-cube** with $k \geq 2$ is a generalization of the d-dimensional cube with $n = k^d$ nodes where each dimension i with $i = 0, \ldots, d-1$ contains k nodes. Each node can be represented by a word with d numbers $(a_0, \ldots, a_{d-1})$ with $0 \leq a_i \leq k-1$, where a_i represents the position of the node in dimension i, $i = 0, \ldots, d-1$. Two nodes $A = (a_0, \ldots, a_{d-1})$ and $B = (b_0, \ldots, b_{d-1})$ are connected by an edge if there is a dimension $j \in \{0, \ldots, d-1\}$ for which $a_j = (b_j \pm 1) \bmod k$ and $a_i = b_i$ for all other dimensions $i = 0, \ldots, d-1, i \neq j$. For $k = 2$, each node has one neighbor in each dimension, resulting in degree $g(G) = d$. For $k > 2$, each node has two neighbors in each dimension, resulting in degree $g(G) = 2d$. The k-ary d-cube captures some of the previously considered topologies as special case: A k-ary 1-cube is a ring with k nodes, a k-ary 2-cube is a torus with k^2 nodes, a 3-ary 3-cube is a three-dimensional torus with $3 \times 3 \times 3$ nodes, and a 2-ary d-cube is a d-dimensional cube.

Table 2.3 summarizes important characteristics of the network topologies described.

2.5.3 *Embeddings*

In this section, we consider the embedding of several networks into a hypercube network, demonstrating that the hypercube topology is versatile and flexible.

2.5.3.1 Embedding a ring into a hypercube network

For an embedding of a ring network with $n = 2^k$ nodes represented by $V' = \{1, \ldots, n\}$ in a k-dimensional cube with nodes $V = \{0, 1\}^k$, a bijective function from V' to V is constructed, such that a ring edge $(i, j) \in E'$ is mapped to a hypercube edge. In the ring, there are edges between neighboring nodes in the sequence $1, \ldots, n$. To construct the embedding, we have to arrange the hypercube nodes in V in a sequence such that there is also an edge between neighboring nodes in the sequence. The sequence is constructed as reflected Gray code (RGC) sequence which is defined as follows:

A k-bit RGC is a sequence with 2^k binary strings of length k such that two neighboring strings differ in exactly one bit position. The RGC sequence is constructed recursively, as follows:

- the 1-bit RGC sequence is $\text{RGC}_1 = (0, 1)$,
- the 2-bit RGC sequence is obtained from RGC_1 by inserting a 0 and a 1 in front of RGC_1, resulting in the two sequences (00, 01) and (10, 11). Reversing the second sequence and concatenation yields RGC_2 = (00, 01, 11, 10).
- For $k \geq 2$, the k-bit Gray code RGC_k is constructed from the $(k-1)$-bit Gray code $\text{RGC}_{k-1} = (b_1, \ldots, b_m)$ with $m = 2^{k-1}$ where each entry b_i for $1 \leq i \leq m$ is a binary string of length $k-1$. To construct RGC_k, RGC_{k-1} is duplicated; a 0 is inserted in front of each b_i of the original sequence, and a 1 is inserted in front of each b_i of the duplicated sequence. This results in sequences $(0b_1, \ldots, 0b_m)$ and $(1b_1, \ldots, 1b_m)$. RGC_k results by reversing the second sequence and concatenating the two sequences; thus $RGC_k = (0b_1, \ldots, 0b_m, 1b_m, \ldots, 1b_1)$.

The Gray code sequences RGC_k constructed in this way have the property that they contain all binary representations of a k-dimensional hypercube, since the construction corresponds to the construction of a k-dimensional cube from two $(k-1)$-dimensional cubes as described in the previous section. Two neighboring k-bit words of RGC_k differ in exactly one bit position, as can be shown by induction. The statement is surely true for RGC_1. Assuming that the statement is true for RGC_{k-1}, it is true for the first 2^{k-1} elements of RGC_k as well as for the last 2^{k-1} elements, since these differ only by a leading 0 or 1 from RGC_{k-1}. The statement is also true for the two middle elements $0b_m$ and $1b_m$ at which the two sequences of length 2^{k-1} are concatenated. Similarly, the first element $0b_1$ and the last element $1b_1$ of RGC_k differ only in the first bit. Thus, neighboring elements of RGC_k are connected by a hypercube edge.

An embedding of a ring into a k-dimensional cube can be defined by the mapping

$$\sigma : \{1, \ldots, n\} \rightarrow \{0, 1\}^k \text{ with } \sigma(i) := RGC_k(i)$$

where $\text{RGC}_k(i)$ denotes the ith element of RGC_k. Figure 2.12 (a) shows an example for $k = 3$.

2.5.3.2 Embedding a two-dimensional mesh into a hypercube network

The embedding of a two-dimensional mesh with $n = n_1 \cdot n_2$ nodes into a k-dimensional cube with $n = 2^k$ nodes can be obtained by a generalization of the embedding of a ring network. For k_1 and k_2 with $n_1 = 2^{k_1}$ and $n_2 = 2^{k_2}$, i.e., $k_1 + k_2 = k$, the Gray codes $\text{RGC}_{k_1} = (a_1, \ldots, a_{n_1})$ and $\text{RGC}_{k_2} = (b_1, \ldots, b_{n_2})$ are used to construct an $n_1 \times n_2$ matrix M whose entries are k-bit strings. In particular, it is

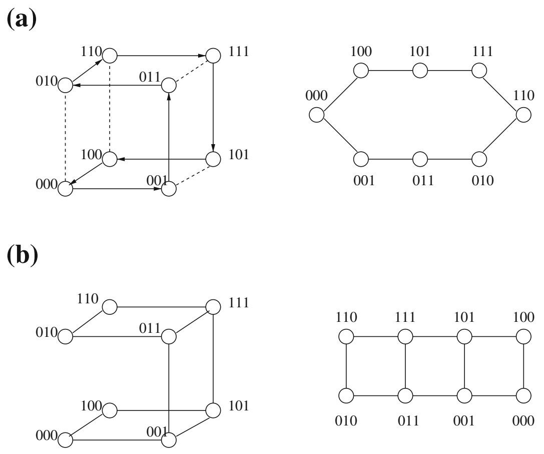


Fig. 2.12 Embeddings into a hypercube network: (a) embedding of a ring network with 8 nodes into a 3-dimensional hypercube and (b) embedding of a 2-dimensional 2 × 4 mesh into a three-dimensional hypercube.

$$M = \begin{bmatrix} a_1b_1 & a_1b_2 & \dots & a_1b_{n_2} \\ a_2b_1 & a_2b_2 & \dots & a_2b_{n_2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n_1}b_1 & a_{n_1}b_2 & \dots & a_{n_1}b_{n_2} \end{bmatrix}.$$

The matrix is constructed such that neighboring entries differ in exactly one bit position. This is true for neighboring elements in a row, since identical elements of RGC_{k_1} and neighboring elements of RGC_{k_2} are used. Similarly, this is true for neighboring elements in a column, since identical elements of RGC_{k_2} and neighboring elements of RGC_{k_1} are used. All elements of M are bit strings of length k and there are no identical bit strings according to the construction. Thus, the matrix M contains all bit representations of nodes in a k-dimensional cube and neighboring entries in M correspond to neighboring nodes in the k-dimensional cube, which are connected by an edge. Thus, the mapping

$$\sigma : \{1, \dots, n_1\} \times \{1, \dots, n_2\} \to \{0, 1\}^k \text{ with } \sigma((i, j)) = M(i, j)$$

is an embedding of the two-dimensional mesh into the k-dimensional cube. Figure 2.12 (b) shows an example.

2.5.3.3 Embedding of a d-dimensional mesh into a hypercube network

In a d-dimensional mesh with $n_i = 2^{k_i}$ nodes in dimension i, $1 \leq i \leq d$, there are $n = n_1 \cdot \dots \cdot n_d$ nodes in total. Each node can be represented by its mesh coordinates $(x_1, \dots, x_d)$ with $1 \leq x_i \leq n_i$. The mapping

$$\sigma : \{(x_1, \ldots, x_d) \mid 1 \le x_i \le n_i, 1 \le i \le d\} \longrightarrow \{0,1\}^k$$
$$\text{with } \sigma((x_1, \ldots, x_d)) = s_1 s_2 \ldots s_d \text{ and } s_i = RGC_{k_i}(x_i)$$

(where s_i is the x_ith bit string in the Gray code sequence RGC_{k_i}) defines an embedding into the k-dimensional cube. For two mesh nodes $(x_1, \ldots, x_d)$ and $(y_1, \ldots, y_d)$ that are connected by an edge in the d-dimensional mesh, there exists exactly one dimension $i \in \{1, \ldots, d\}$ with $|x_i - y_i| = 1$ and for all other dimensions $j \ne i$, it is $x_j = y_j$. Thus, for the corresponding hypercube nodes $\sigma((x_1, \ldots, x_d)) = s_1 s_2 \ldots s_d$ and $\sigma((y_1, \ldots, y_d)) = t_1 t_2 \ldots t_d$, all components $s_j = \mathrm{RGC}_{k_j}(x_j) = \mathrm{RGC}_{k_j}(y_j) = t_j$ for $j \ne i$ are identical. Moreover, $\mathrm{RGC}_{k_i}(x_i)$ and $\mathrm{RGC}_{k_i}(y_i)$ differ in exactly one bit position. Thus, the hypercube nodes $s_1 s_2 \ldots s_d$ and $t_1 t_2 \ldots t_d$ also differ in exactly one bit position and are therefore connected by an edge in the hypercube network.

2.5.4 Dynamic Interconnection Networks

Dynamic interconnection networks are also called indirect interconnection networks. In these networks, nodes or processors are not connected directly with each other. Instead, switches are used and provide an *indirect* connection between the nodes, giving these networks their name. From the processors' point of view, such a network forms an interconnection unit into which data can be sent and from which data can be received. Internally, a dynamic network consists of switches that are connected by physical links. For a message transmission from one node to another node, the switches can be configured *dynamically* such that a connection is established.

Dynamic interconnection networks can be characterized according to their topological structure. Popular forms are bus networks, multistage networks, and crossbar networks.

2.5.4.1 Bus networks

A bus essentially consists of a set of wires which can be used to transport data from a sender to a receiver, see Fig. 2.13 for an illustration. In some cases, several hundreds

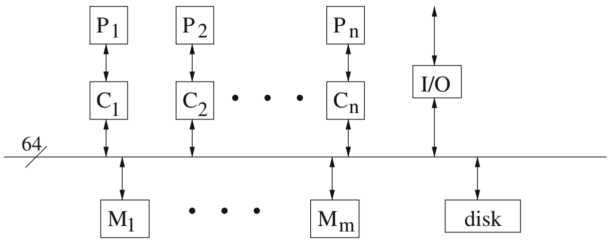


Fig. 2.13 Illustration of a bus network with 64 wires to connect processors $P_1, \ldots, P_n$ with caches $C_1, \ldots, C_n$ to memory modules $M_1, \ldots, M_m$.

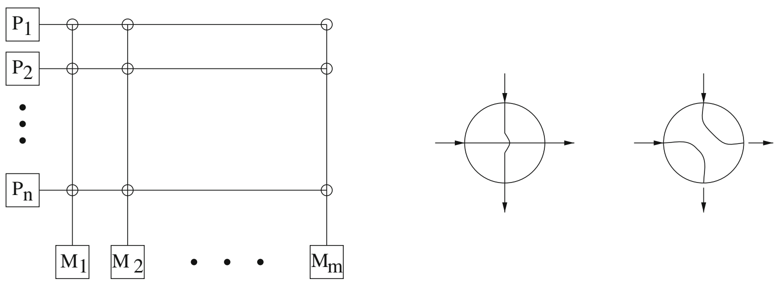


Fig. 2.14 Illustration of a $n \times m$ crossbar network for n processors and m memory modules (left). Each network switch can be in one of two states: straight or direction change (right).

of wires are used to ensure a fast transport of large data sets. At each point in time, only one data transport can be performed via the bus, i.e., the bus must be used in a time-sharing way. When several processors attempt to use the bus simultaneously, a **bus arbiter** is used for the coordination. Because the likelihood for simultaneous requests of processors increases with the number of processors, bus networks are typically used for a small number of processors only.

2.5.4.2 Crossbar networks

An $n \times m$ crossbar network has n inputs and m outputs. The actual network consists of $n \cdot m$ switches as illustrated in Fig. 2.14 (left). For a system with a shared address space, the input nodes may be processors and the outputs may be memory modules. For a system with a distributed address space, both the input nodes and the output nodes may be processors. For each request from a specific input to a specific output, a connection in the switching network is established. Depending on the specific input and output nodes, the switches on the connection path can have different states (straight or direction change) as illustrated in Fig. 2.14 (right). Typically, crossbar networks are used only for a small number of processors because of the large hardware overhead required.

2.5.4.3 Multistage switching networks

Multistage switching networks consist of several stages of switches with connecting wires between neighboring stages. The network is used to connect input devices to output devices. Input devices are typically the processors of a parallel system. Output devices can be processors (for distributed memory machines) or memory modules (for shared memory machines). The goal is to obtain a small distance for arbitrary pairs of input and output devices to ensure fast communication. The internal connections between the stages can be represented as a graph where switches are represented by nodes and wires between switches are represented by edges. Input

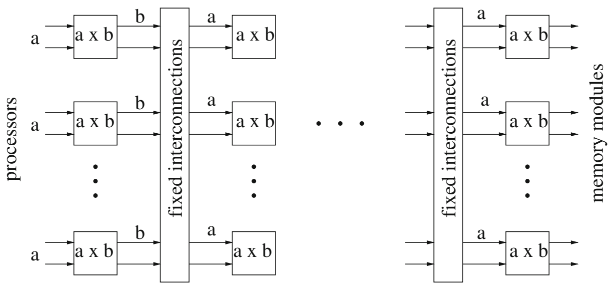


Fig. 2.15 Multistage interconnection networks with $a \times b$ crossbars as switches according to [105].

and output devices can be represented as specialized nodes with edges going into the actual switching network graph. The construction of the switching graph and the degree of the switches used are important characteristics of multistage switching networks.

Regular multistage interconnection networks are characterized by a *regular* construction method using the same degree of incoming and outgoing wires for all switches. For the switches, $a \times b$ crossbars are often used where a is the input degree and b is the output degree. The switches are arranged in stages such that neighboring stages are connected by fixed interconnections, see Fig. 2.15 for an illustration. The input wires of the switches of the first stage are connected with the input devices. The output wires of the switches of the last stage are connected with the output devices. Connections from input devices to output devices are performed by selecting a path from a specific input device to the selected output device and setting the switches on the path such that the connection is established.

The actual graph representing a regular multistage interconnection network results from *gluing* neighboring stages of switches together. The connection between neighboring stages can be described by a directed acyclic graph of depth 1. Using w nodes for each stage, the degree of each node is $g = n/w$ where n is the number of edges between neighboring stages. The connection between neighboring stages can be represented by a permutation $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ which specifies which output link of one stage is connected to which input link of the next stage. This means that the output links $\{1, \ldots, n\}$ of one stage are connected to the input links $(\pi(1), \ldots, \pi(n))$ of the next stage. Partitioning the permutation $(\pi(1), \ldots, \pi(n))$ into w parts results in the ordered set of input links of nodes of the next stage. For regular multistage interconnection networks, the same permutation is used for all stages, and the stage number can be used as parameter.

Popular regular multistage networks are the omega network, the baseline network, and the butterfly network. These networks use 2×2 crossbar switches which are arranged in $\log n$ stages. Each switch can be in one of four states as illustrated in Fig. 2.16. In the following, we give a short overview of the omega, baseline, butterfly, Beneš, and fat-tree networks, see [130] for a detailed description.

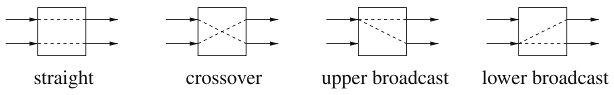


Fig. 2.16 Settings for switches in a omega, baseline, or butterfly network.

2.5.4.4 Omega network

An $n \times n$ omega network is based on 2×2 crossbar switches which are arranged in $\log n$ stages such that each stage contains $n/2$ switches where each switch has two input and two output links. Thus, there are $(n/2) \cdot \log n$ switches in total, with $\log n \equiv \log_2 n$. Each switch can be in one of four states, see Fig. 2.16. In the omega network, the permutation function describing the connection between neighboring stages is the same for all stages, independently from the number of the stage. The switches in the network are represented by pairs $(\boldsymbol{\alpha}, i)$ where $\boldsymbol{\alpha} \in \{0, 1\}^{\log n-1}$ is a bit string of length $\log n - 1$ representing the position of a switch within a stage and $i \in \{0, \dots, \log n - 1\}$ is the stage number. There is an edge from node $(\boldsymbol{\alpha}, i)$ in stage i to two nodes $(\boldsymbol{\beta}, i + 1)$ in stage $i + 1$ where $\boldsymbol{\beta}$ is defined as follows:

1. $\boldsymbol{\beta}$ results from $\boldsymbol{\alpha}$ by a cyclic left shift, or
2. $\boldsymbol{\beta}$ results from $\boldsymbol{\alpha}$ by a cyclic left shift followed by an inversion of the last (rightmost) bit.

An $n \times n$ omega network is also called $(\log n - 1)$-dimensional omega network. Figure 2.17 (a) shows a 16×16 (three-dimensional) omega network with four stages and eight switches per stage.

2.5.4.5 Butterfly network

Similar to the omega network, a k-dimensional butterfly network connects $n = 2^{k+1}$ inputs to $n = 2^{k+1}$ outputs using a network of 2×2 crossbar switches. Again, the switches are arranged in $k + 1$ stages with 2^k nodes/switches per stage. This results in a total number $(k + 1) \cdot 2^k$ of nodes. Again, the nodes are represented by pairs $(\boldsymbol{\alpha}, i)$ where i for $0 \leq i \leq k$ denotes the stage number and $\boldsymbol{\alpha} \in \{0, 1\}^k$ is the position of the node in the stage. The connection between neighboring stages i and $i + 1$ for $0 \leq i < k$ is defined as follows: Two nodes $(\boldsymbol{\alpha}, i)$ and $(\boldsymbol{\alpha}', i + 1)$ are connected if and only if

1. $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}'$ are identical (straight edge), or
2. $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}'$ differ in precisely the $(i + 1)$th bit from the left (cross edge).

Fig. 2.17 (b) shows a 16×16 butterfly network with four stages.

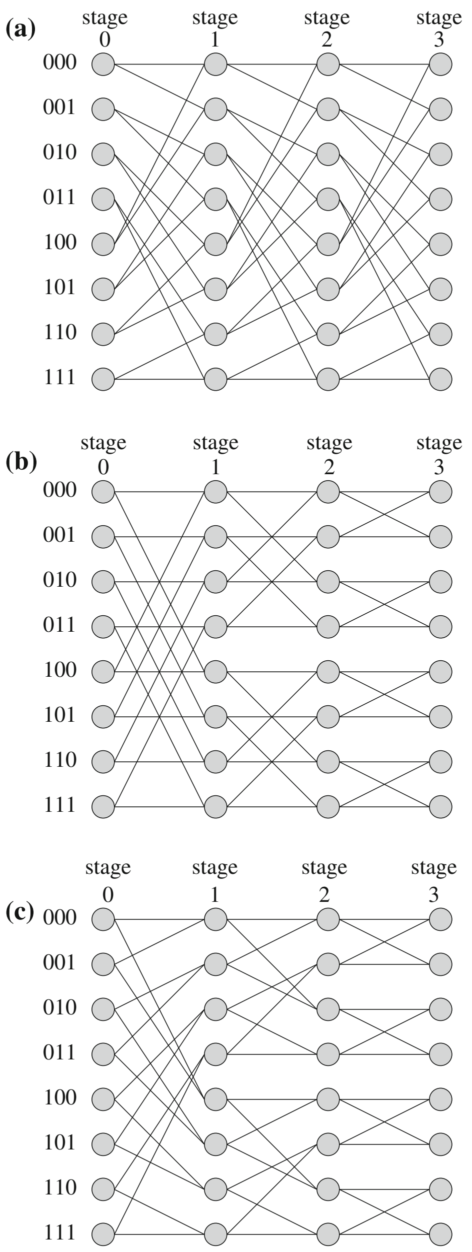


Fig. 2.17 Examples for dynamic interconnection networks: **(a)** 16 × 16 omega network, **(b)** 16 × 16 butterfly network, **(c)** 16 × 16 baseline network. All networks are three-dimensional.

2.5.4.6 Baseline network

The k-dimensional baseline network has the same number of nodes, edges and stages as the butterfly network. Neighboring stages are connected as follows: node $(\boldsymbol{\alpha}, i)$ is connected to node $(\boldsymbol{\alpha}', i + 1)$ for $0 \leq i < k$ if and only if

1. $\boldsymbol{\alpha}'$ results from $\boldsymbol{\alpha}$ by a cyclic right shift on the last $k - i$ bits of $\boldsymbol{\alpha}$, or
2. $\boldsymbol{\alpha}'$ results from $\boldsymbol{\alpha}$ by first inverting the last (rightmost) bit of $\boldsymbol{\alpha}$ and then performing a cyclic right shift on the last $k - i$ bits.

Figure 2.17 (c) shows a 16×16 baseline network with four stages.

2.5.4.7 Beneš network

The k-dimensional Beneš network is constructed from two k-dimensional butterfly networks such that the first $k + 1$ stages are a butterfly network and the last $k + 1$ stages are a reverted butterfly network. The last stage $(k + 1)$ of the first butterfly network and the first stage of the second (reverted) butterfly network are merged. In total, the k-dimensional Beneš network has $2k + 1$ stages with 2^k switches in each stage. Figure 2.18 (a) shows a three-dimensional Beneš network as an example.

2.5.4.8 Fat tree network

The basic structure of a *dynamic tree* or *fat tree* network is a complete binary tree. The difference to a normal tree is that the number of connections between the nodes increases toward the root to avoid bottlenecks. Inner tree nodes consist of switches whose structure depends on their position in the tree structure. The leaf level is level 0. For n processors, represented by the leaves of the tree, a switch on tree level i has 2^i input links and 2^i output links for $i = 1, \ldots, \log n$. This can be realized by assembling the switches on level i internally from 2^{i-1} switches with two input and two output links each. Thus, each level i consists of $n/2$ switches in total, grouped in $2^{\log n - i}$ nodes. This is shown in Fig. 2.18 (b) for a fat tree with four layers. Only the inner switching nodes are shown, not the leaf nodes representing the processors.

2.6 Routing and Switching

Direct and indirect interconnection networks provide the physical basis to send messages between processors. If two processors are not directly connected by a network link, a path in the network consisting of a sequence of nodes has to be used for message transmission. In the following, we give a short description on how to select a suitable path in the network (routing) and how messages are handled at intermediate nodes on the path (switching).

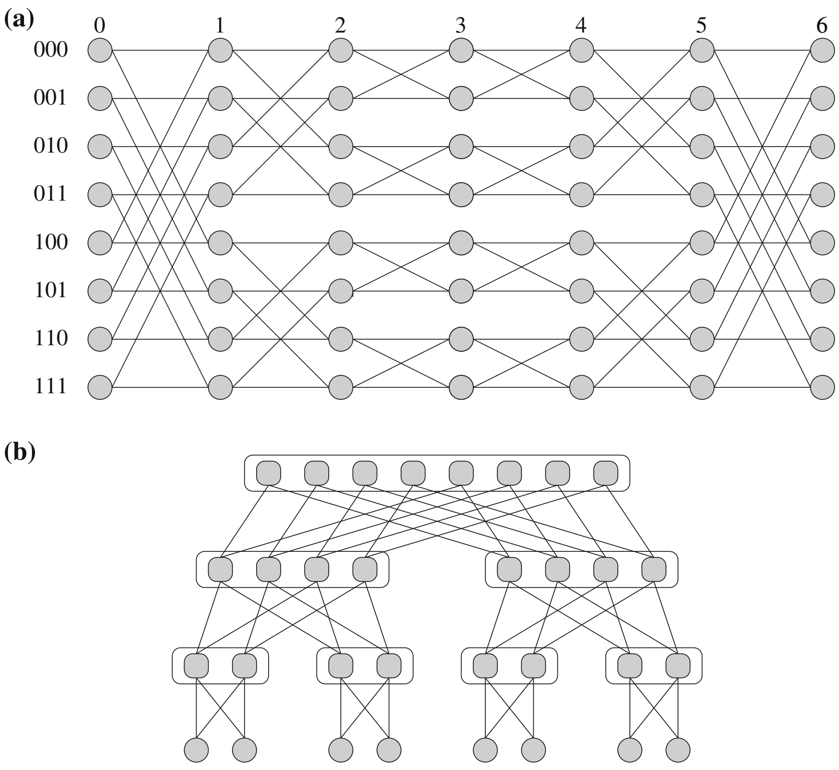


Fig. 2.18 Examples for dynamic interconnection networks: **(a)** three-dimensional Benes network and **(b)** fat tree network for 16 processors.

2.6.1 Routing Algorithms

A **routing algorithm** determines a path in a given network from a source node A to a destination node B. The path consists of a sequence of nodes, such that neighboring nodes in the sequence are connected by a physical network link. The path starts with node A and ends at node B. A large variety of routing algorithms have been proposed in the literature, and we can only give a short overview in the following. For a more detailed description and discussion, we refer to [41, 50].

Typically, multiple message transmissions are being executed concurrently according to the requirements of one or several parallel programs. A routing algorithm tries to reach an even load on the physical network links as well as to avoid the occurrence of deadlocks. A set of messages is in a **deadlock situation** if each of the messages is supposed to be transmitted over a link that is currently used by another message of the set. A routing algorithm tries to select a path in the network connecting nodes A and B such that minimum costs result, thus leading to a fast message transmission between A and B. The resulting communication costs depend not only on the length of the path used, but also on the load of the links on the path. The following issues are important for the path selection:

- **network topology**: the topology of the network determines which paths are available in the network to establish a connection between nodes A and B;
- **network contention**: contention occurs when two or more messages should be transmitted at the same time over the same network link, thus leading to a delay in message transmission;
- **network congestion**: congestion occurs when too many messages are assigned to a restricted resource (like a network link or buffer) such that arriving messages have to be discarded since they cannot be stored anywhere. Thus, in contrast to contention, congestion leads to an overflow situation with message loss [157].

A large variety of routing algorithms have been proposed in the literature. Several classification schemes can be used for a characterization. Using the path length, **minimal** and **nonminimal** routing algorithms can be distinguished. Minimal routing algorithms always select the shortest message transmission, which means that when using a link of the path selected, a message always gets closer to the target node. But this may lead to congestion situations. Nonminimal routing algorithms do not always use paths with minimum length if this is necessary to avoid congestion at intermediate nodes.

A further classification can be made by distinguishing **deterministic** routing algorithms and **adaptive** routing algorithms. A routing algorithm is deterministic if the path selected for message transmission only depends on the source and destination node regardless of other transmissions in the network. Therefore, deterministic routing can lead to unbalanced network load. Path selection can be done *source oriented* at the sending node or *distributed* during message transmission at intermediate nodes. An example for deterministic routing is **dimension-order routing** which can be applied for network topologies that can be partitioned into several orthogonal dimensions as it is the case for meshes, tori, and hypercube topologies. Using dimension-order routing, the routing path is determined based on the position of the source node and the target node by considering the dimensions in a fixed order and traversing a link in the dimension if necessary. This can lead to network contention because of the deterministic path selection.

Adaptive routing tries to avoid such contentions by dynamically selecting the routing path based on load information. Between any pair of nodes, multiple paths are available. The path to be used is dynamically selected such that network traffic is spread evenly over the available links, thus leading to an improvement of network utilization. Moreover, *fault tolerance* is provided, since an alternative path can be used in case of a link failure. Adaptive routing algorithms can be further categorized into minimal and nonminimal adaptive algorithms as described above. In the following, we give a short overview of important routing algorithms. For a more detailed treatment, we refer to [41, 105, 50, 130, 144].

2.6.1.1 Dimension-order routing

We give a short description of XY routing for two-dimensional meshes and E-cube routing for hypercubes as typical examples for dimension-order routing algorithms.

XY routing for two-dimensional meshes

For a two-dimensional mesh, the position of the nodes can be described by an X-coordinate and an Y-coordinate where X corresponds to the horizontal and Y corresponds to the vertical direction. To send a message from a source node A with position (X_A, Y_A) to target node B with position (X_B, Y_B), the message is sent from the source node into (positive or negative) X-direction until the X-coordinate X_B of B is reached. Then, the message is sent into Y-direction until Y_B is reached. The length of the resulting path is $\mid X_A - X_B \mid + \mid Y_A - Y_B \mid$. This routing algorithm is deterministic and minimal.

E-cube routing for hypercubes

In a k-dimensional hypercube, each of the $n = 2^k$ nodes has a direct interconnection link to each of its k neighbors. As introduced in Sect. 2.5.2, each of the nodes can be represented by a bit string of length k, such that the bit string of one of the k neighbors is obtained by inverting one of the bits in the bit string. E-cube uses the bit representation of a sending node A and a receiving node B to select a routing path between them. Let $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0 \ldots \boldsymbol{\alpha}_{k-1}$ be the bit representation of A and $\boldsymbol{\beta} = \boldsymbol{\beta}_0 \ldots \boldsymbol{\beta}_{k-1}$ be the bit representation of B. Starting with A, in each step a dimension is selected which determines the next node on the routing path. Let A_i with bit representation $\boldsymbol{\gamma} = \boldsymbol{\gamma}_0 \ldots \boldsymbol{\gamma}_{k-1}$ be a node on the routing path $A = A_0, A_1, \ldots, A_l = B$ from which the message should be forwarded in the next step. For the forwarding from A_i to A_{i+1}, the following two substeps are made:

- The bit string $\boldsymbol{\gamma} \oplus \boldsymbol{\beta}$ is computed where $\oplus$ denotes the bitwise exclusive or computation (i.e., $0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0$)
- The message is forwarded in dimension d where d is the rightmost bit position of $\boldsymbol{\gamma} \oplus \boldsymbol{\beta}$ with value 1. The next node A_{i+1} on the routing path is obtained by inverting the dth bit in $\boldsymbol{\gamma}$, i.e., the bit representation of A_{i+1} is $\boldsymbol{\delta} = \boldsymbol{\delta}_0 \ldots \boldsymbol{\delta}_{k-1}$ with $\boldsymbol{\delta}_j = \boldsymbol{\gamma}_j$ for $j \neq d$ and $\boldsymbol{\delta}_d = \bar{\boldsymbol{\gamma}}_d$. The target node B is reached when $\boldsymbol{\gamma} \oplus \boldsymbol{\beta} = 0$.

Example For $k = 3$, let A with bit representation $\boldsymbol{\alpha} = 010$ be the source node and B with bit representation $\boldsymbol{\beta} = 111$ be the target node. First, the message is sent from A into direction $d = 2$ to A_1 with bit representation 011 (since $\boldsymbol{\alpha} \oplus \boldsymbol{\beta} = 101$). Then, the message is sent in dimension $d = 0$ to $\boldsymbol{\beta}$ since $(011 \oplus 111 = 100)$. □

2.6.1.2 Deadlocks and routing algorithms

Usually, multiple messages are in transmission concurrently. A deadlock occurs if the transmission of a subset of the messages is blocked forever. This can happen in particular if network resources can be used only by one message at a time. If, for example, the links between two nodes can be used by only one message at a time and if a link can only be released when the following link on the path is free, then the mutual request for links can lead to a deadlock. Such deadlock situations can be avoided by using a suitable routing algorithm. Other deadlock situations that occur because of limited size of the input or output buffer of the interconnection links or because of an unsuited order of the send and receive operations are considered in Sect. 2.6.3 on switching strategies and Chap. 5 on message-passing programming.

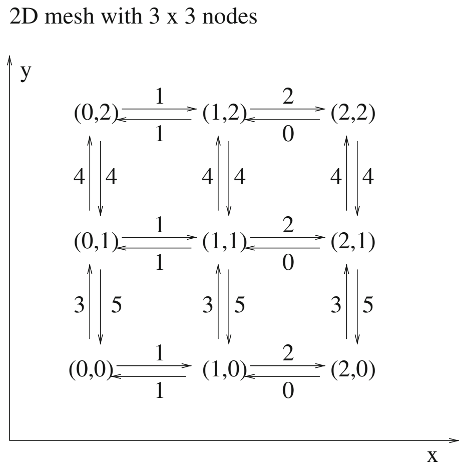
To prove the deadlock freedom of routing algorithms, possible dependencies between interconnection channels are considered. A dependence from an interconnection channel l_1 to an interconnection channel l_2 exists, if it is possible that the routing algorithm selects a path which contains channel l_2 directly after channel l_1. These dependencies between interconnection channels can be represented by a **channel dependence graph** which contains the interconnection channels as nodes; each dependence between two channels is represented by an edge. A routing algorithm is deadlock free for a given topology, if the channel dependence graph does not contain cycles. In this case, no communication pattern can ever lead to a deadlock.

For topologies that do not contain cycles, no channel dependence graph can contain cycles, and therefore each routing algorithm for such a topology must be deadlock free. For topologies with cycles, the channel dependence graph must be analyzed. In the following, we show that XY routing for 2D meshes with bidirectional links is deadlock free.

Deadlock freedom of XY routing

The channel dependence graph for XY routing contains a node for each unidirectional link of the two-dimensional $n_x \times n_Y$ mesh, i.e., there are two nodes for each bidirectional link of the mesh. There is a dependence from link u to link v, if v can be directly reached from u in horizontal or vertical direction or by a 90° turn down or up. To show the deadlock freedom, all uni-directional links of the mesh are numbered as follows:

- Each horizontal edge from node (i, y) to node $(i + 1, y)$ gets number $i + 1$ for $i = 0, \dots, n_x - 2$ for each valid value of y. The opposite edge from $(i + 1, y)$ to (i, y) gets number $n_x - 1 - (i + 1) = n_x - i - 2$ for $i = 0, \dots, n_x - 2$. Thus, the edges in increasing x-direction are numbered from 1 to $n_x - 1$, the edges in decreasing x-direction are numbered from 0 to $n_x - 2$.
- Each vertical edge from (x, j) to $(x, j + 1)$ gets number $j + n_x$ for $j = 0, \dots, n_y - 2$. The opposite edge from $(x, j + 1)$ to (x, j) gets number $n_x + n_y - (j + 1)$.



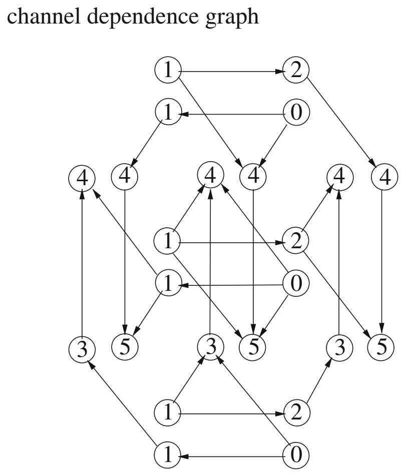


Fig. 2.19 3×3 mesh and corresponding channel dependence graph for XY routing.

Figure 2.19 shows a 3×3 mesh and the resulting channel dependence graph for XY routing. The nodes of the graph are annotated with the numbers assigned to the corresponding network links. It can be seen that all edges in the channel dependence graph go from a link with a smaller number to a link with a larger number. Thus, a delay during message transmission along a routing path can occur only if the message has to wait after the transmission along a link with number i for the release of a successive link w with number $j > i$ currently used by another message transmission (delay condition). A deadlock can only occur if a set of messages $\{N_1, \ldots, N_k\}$ and network links $\{n_1, \ldots, n_k\}$ exists such that for $1 \leq i < k$ each message N_i uses a link n_i for transmission and waits for the release of link n_{i+1} which is currently used for the transmission of message N_{i+1}. Additionally, N_k is currently transmitted using link n_k and waits for the release of n_1 used by N_1. If $n()$ denotes the numbering of the network links introduced above, the delay condition implies that for the deadlock situation just described, it must be

$$n(n_1) < n(n_2) < \ldots < n(n_k) < n(n_1).$$

This is a contradiction, and thus no deadlock can occur. Each routing path selected by XY routing consists of a sequence of links with increasing numbers. Each edge in the channel dependence graph points to a link with a larger number than the source link. Thus, there can be no cycles in the channel dependence graph. A similar approach can be used to show deadlock freedom for E-cube routing, see [44].

2.6.1.3 Source-based routing

Source-based routing is a deterministic routing algorithm for which the source node determines the entire path for message transmission. For each node n_i on the path, the output link number a_i is determined, and the sequence of output link numbers $a_0, \ldots, a_{n-1}$ to be used is added as header to the message. When the message passes a node, the first link number is stripped from the front of the header and the message is forwarded through the specified link to the next node.

2.6.1.4 Table-driven routing

For table-driven routing, each node contains a routing table which contains for each destination node the output link to be used for the transmission. When a message arrives at a node, a lookup in the routing table is used to determine how the message is forwarded to the next node.

2.6.1.5 Turn model routing

The turn model [74, 144] tries to avoid deadlocks by a suitable selection of turns that are allowed for the routing. Deadlocks occur if the paths for message transmission contain turns that may lead to cyclic waiting in some situations. Deadlocks can be avoided by prohibiting some of the turns. An example is the XY routing on a 2D mesh. From the eight possible turns, see Fig. 2.20 (above), only four are allowed for XY routing, prohibiting turns from vertical into horizontal direction, see Fig. 2.20 (middle) for an illustration. The remaining four turns are not allowed in order to prevent cycles in the networks. This not only avoids the occurrence of deadlocks, but also prevents the use of adaptive routing. For n-dimensional meshes and, in the general case, k-ary d-cubes, the turn model tries to identify a minimum number of turns that must be prohibited for routing paths to avoid the occurrence of cycles. Examples are the west-first routing for 2D meshes and the P-cube routing for n-dimensional hypercubes.

The **west-first routing** algorithm for a two-dimensional mesh prohibits only two of the eight possible turns: turns to the west (left) are prohibited, and only the turns shown in Fig. 2.20 (bottom) are allowed. Routing paths are selected such that messages that must travel to the west must do so before making any turns. Such messages are sent to the west first until the requested x-coordinate is reached. Then the message can be adaptively forwarded to the south (bottom), east (right), or north (top). Figure 2.21 shows some examples for possible routing paths [144]. West-first routing is deadlock free, since cycles are avoided. For the selection of minimal routing paths, the algorithm is adaptive only if the target node lies to the east (right). Using nonminimal routing paths, the algorithm is always adaptive.

Routing in the n-dimensional hypercube can be done with P**-cube routing**. To send a message from a sender A with bit representation $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0 \ldots \boldsymbol{\alpha}_{n-1}$ to a receiver

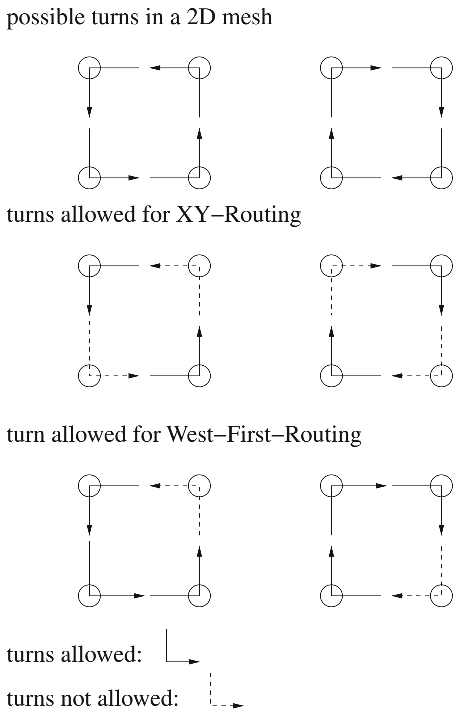

possible turns in a 2D mesh
turns allowed for XY−Routing
turn allowed for West−First−Routing
turns allowed:
turns not allowed:

Fig. 2.20 Illustration of turns for a 2D mesh with all possible turns (top), allowed turns for XY routing (middle), and allowed turns for west-first routing (bottom).

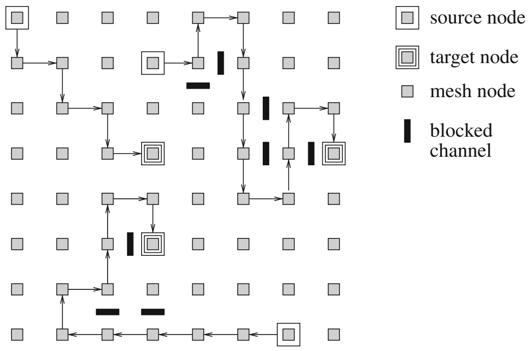


Fig. 2.21 Illustration of path selection for west-first routing in an 8 × 8 mesh. The links shown as blocked are used for other message transmissions and are not available for the current transmission. One of the paths shown is minimal, the other two are non-minimal, since some of the links are blocked.

B with bit representation $\boldsymbol{\beta} = \boldsymbol{\beta}_0 \dots \boldsymbol{\beta}_{n-1}$, the bit positions in which $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ differ are considered. The number of these bit positions is the Hamming distance between A and B which determines the minimum length of a routing path from A to B. The set $E = \{i \mid \boldsymbol{\alpha}_i \neq \boldsymbol{\beta}_i, i = 0, \dots, n-1\}$ of different bit positions is partitioned into two sets $E_0 = \{i \in E \mid \boldsymbol{\alpha}_i = 0 \text{ and } \boldsymbol{\beta}_i = 1\}$ and $E_1 = \{i \in E \mid \boldsymbol{\alpha}_i = 1 \text{ and } \boldsymbol{\beta}_i = 0\}$. Message transmission from A to B is split into two phases accordingly: First, the message is sent into the dimensions in E_0 and then into the dimensions in E_1.

2.6.1.6 Virtual channels

The concept of *virtual channels* is often used for minimal adaptive routing algorithms. To provide multiple (virtual) channels between neighboring network nodes, each physical link is split into multiple virtual channels. Each virtual channel has its own separate buffer. The provision of virtual channels does not increase the number of physical links in the network, but can be used for a systematic avoidance of deadlocks.

Based on virtual channels, a network can be split into several virtual networks, such that messages injected into a virtual network can only move into one direction for each dimension. This can be illustrated for a two-dimensional mesh which is split into two virtual networks, a $+X$ network and a $-X$ network, see Fig. 2.22 for an illustration. Each virtual network contains all nodes, but only a subset of the virtual channels. The $+X$ virtual network contains in the vertical direction all virtual channels between neighboring nodes, but in the horizontal direction only the virtual channels in positive direction. Similarly, the $-X$ virtual network contains in the horizontal direction only the virtual channels in negative direction, but all virtual channels in the vertical direction. The latter is possible by the definition of a suitable number of virtual channels in the vertical direction. Messages from a node A with x-coordinate x_A to a node B with x-coordinate x_B are sent in the $+X$ network, if $x_A < x_B$. Messages from A to B with $x_A > x_B$ are sent in the $-X$ network. For $x_A = x_B$, one of the two networks can be selected arbitrarily, possibly using load

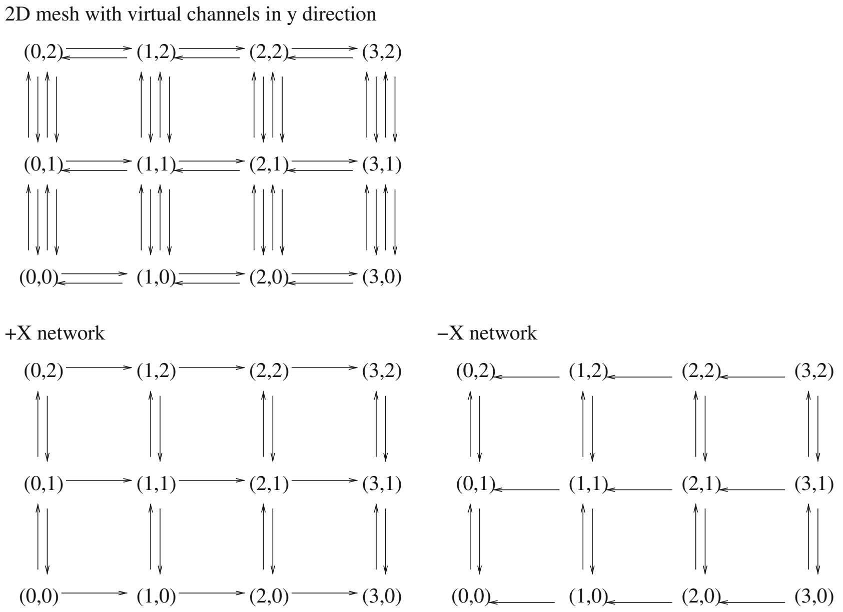


Fig. 2.22 Partitioning of a 2D mesh with virtual channels into a $+X$ network and a $-X$ network for applying a minimal adaptive routing algorithm.

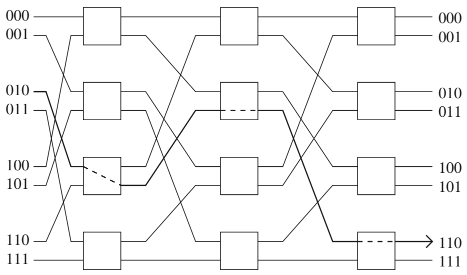


Fig. 2.23 8×8 omega network with path from 010 to 110.

information for the selection. The resulting adaptive routing algorithm is deadlock free [144]. For other topologies like hypercubes or tori, more virtual channels might be needed to provide deadlock freedom [144].

A nonminimal adaptive routing algorithm can send messages over longer paths if no minimal path is available. **Dimension reversal routing** can be applied to arbitrary meshes and k-ary d-cubes. The algorithm uses r pairs of virtual channels between any pair of nodes that are connected by a physical link. Correspondingly, the network is split into r virtual networks where network i for $i = 0, \ldots, r-1$ uses all virtual channels i between the nodes. Each message to be transmitted is assigned a class c with initialization $c = 0$ which can be increased to $c = 1, \ldots, r-1$ during message transmission. A message with class $c = i$ can be forwarded in network i in each dimension, but the dimensions must be traversed in increasing order. If a message must be transmitted in opposite order, its class is increased by 1 (reverse dimension order). The parameter r controls the number of dimension reversals that are allowed. If $c = r$ is reached, the message is forwarded according to dimension-ordered routing.

2.6.2 Routing in the Omega Network

The omega network introduced in Sect. 2.5.4 allows message forwarding using a distributed algorithm where each switch can forward the message without coordination with other switches. For the description of the algorithm, it is useful to represent each of the n input and output channels by a bit string of length $\log n$ [130]. To forward a message from an input channel with bit representation $\boldsymbol{\alpha}$ to an output channel with bit representation $\boldsymbol{\beta}$ the receiving switch on stage k of the network, $k = 0, \ldots, \log n - 1$, considers the kth bit $\boldsymbol{\beta}_k$ (from the left) of $\boldsymbol{\beta}$ and selects the output link for forwarding the message according to the following rule:

- for $\boldsymbol{\beta}_k = 0$, the message is forwarded over the upper link of the switch and
- for $\boldsymbol{\beta}_k = 1$, the message is forwarded over the lower link of the switch.

Figure 2.23 illustrates the path selected for message transmission from input channel $\boldsymbol{\alpha} = 010$ to the output channel $\boldsymbol{\beta} = 110$ according to the algorithm just described. In an $n \times n$ omega network, at most n messages from different input channels to

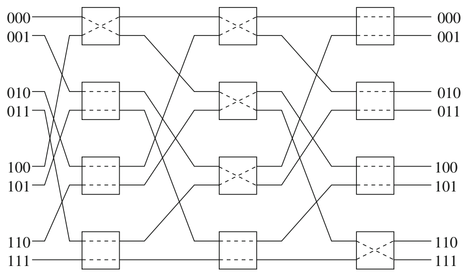


Fig. 2.24 8×8 Omega network with switch positions for the realization of π^8 from the text.

different output channels can be sent concurrently without collision. An example of a concurrent transmission of $n = 8$ messages in a 8×8 omega network can be described by the permutation

$$\pi^8 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 3 & 0 & 1 & 2 & 5 & 4 & 6 \end{pmatrix}$$

which specifies that the messages are sent from input channel i ($i = 0, \ldots, 7$) to output channel $\pi^8(i)$. The corresponding paths and switch positions for the eight paths are shown in Fig. 2.24.

Many simultaneous message transmissions that can be described by permutations $\pi^8 : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\}$ cannot be executed concurrently since **network conflicts** would occur. For example, the two message transmissions from $\boldsymbol{\alpha}_1 = 010$ to $\boldsymbol{\beta}_1 = 110$ and from $\boldsymbol{\alpha}_2 = 000$ to $\boldsymbol{\beta}_2 = 111$ in an 8×8 omega network would lead to a conflict. This kind of conflicts occur, since there is exactly one path for any pair $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ of input and output channels, i.e., there is no alternative to avoid a critical switch. Networks with this characteristic are also called **blocking networks**. Conflicts in blocking networks can be resolved by multiple transmissions through the network.

There is a notable number of permutations that cannot be implemented in one switching of the network. This can be seen as follows. For the connection from the n input channels to the n output channels, there are in total $n!$ possible permutations, since each output channel must be connected to exactly one input channel. There are in total $n/2 \cdot \log n$ switches in the omega network, each of which can be in one of two positions. This leads to $2^{n/2 \cdot \log n} = n^{n/2}$ different switchings of the entire network, corresponding to n concurrent paths through the network. In conclusion, only $n^{n/2}$ of the $n!$ possible permutations can be performed without conflicts.

Other examples for blocking networks are the butterfly or Banyan network, the baseline network and the delta network [130]. In contrast, the Beneš network is a nonblocking network since there are different paths from an input channel to an output channel. For each permutation $\pi : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\}$ there exists a switching of the Beneš network which realizes the connection from input i to output $\pi(i)$ for $i = 0, \ldots, n-1$ concurrently without collision, see [130] for

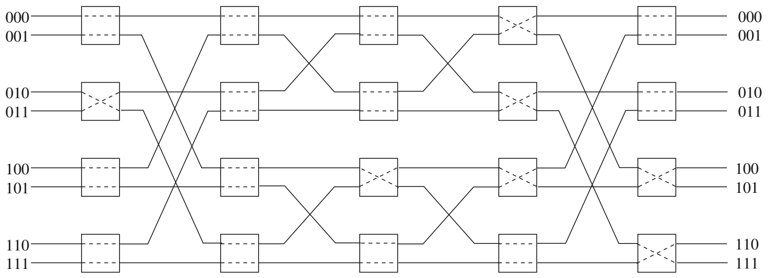


Fig. 2.25 8×8 Benes network with switch positions for the realization of π^8 from the text.

more details. As example, the switching for the permutation

$$\pi^8 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 3 & 4 & 7 & 0 & 1 & 2 & 6 \end{pmatrix}$$

is shown in Fig. 2.25.

2.6.3 Switching

The switching strategy determines how a message is transmitted along a path that has been selected by the routing algorithm. In particular, the switching strategy determines

- whether and how a message is split into pieces, which are called packets or *flits* (for *flow control units*),
- how the transmission path from the source node to the destination node is allocated, and
- how messages or pieces of messages are forwarded from the input channel to the output channel of a switch or a router. The routing algorithm only determines *which* output channel should be used.

The switching strategy may have a large influence on the message transmission time from a source to a destination. Before considering specific switching strategies, we first consider the time for message transmission between two nodes that are directly connected by a physical link.

2.6.3.1 Message transmission between neighboring processors

Message transmission between two directly connected processors is implemented as a series of steps. These steps are also called *protocol*. In the following, we sketch a simple example protocol [94]. To send a message, the sending processor performs the following steps:

1. The message is copied into a system buffer.
2. A checksum is computed and a *header* is added to the message, containing the checksum as well as additional information related to the message transmission.
3. A timer is started and the message is sent out over the network interface.

To receive a message, the receiving processor performs the following steps:

1. The message is copied from the network interface into a system buffer.
2. The checksum is computed over the data contained. This checksum is compared with the checksum stored in the header. If both checksums are identical, an acknowledgment message is sent to the sender. In case of a mismatch of the checksums, the message is discarded. The message will be re-sent again after the sender timer has elapsed.
3. If the checksum are identical, the message is copied from the system buffer into the user buffer, provided by the application program. The application program gets a notification and can continue execution.

After having sent out the message, the sending processor performs the following steps:

1. If an acknowledgment message arrives for the message sent out, the system buffer containing a copy of the message can be released.
2. If the timer has elapsed, the message will be re-sent again. The timer is started again, possibly with a longer time.

In this protocol, it has been assumed that the message is kept in the system buffer of the sender to be re-send if necessary. If message loss is tolerated, no re-send is necessary and the system buffer of the sender can be re-used as soon as the packet has been sent out. Message transmission protocols used in practice are typically much more complicated and may take additional aspects like network contention or possible overflows of the system buffer of the receiver into consideration. A detailed overview can be found in [125, 157].

The time for a message transmission consists of the actual transmission time over the physical link and the time needed for the software overhead of the protocol, both at the sender and the receiver side. Before considering the transmission time in more detail, we first review some performance measures that are often used in this context, see [94, 41] for more details.

- The **bandwidth** of a network link is defined as the maximum frequency at which data can be sent over the link. The bandwidth is measured in bits per second or bytes per second.
- The **byte transfer time** is the time which is required to transmit a single byte over a network link. If the bandwidth is measured in bytes per second, the byte transfer time is the reciprocal of the bandwidth.

- The **time of flight**, also referred to as *channel propagation delay*, is the time which the first bit of a message needs to arrive at the receiver. This time mainly depends on the physical distance between the sender and the receiver.
- The **transmission time** is the time needed to transmit the message over a network link. The transmission time is the message size in bytes divided by the bandwidth of the network link, measured in bytes per second. The transmission time does not take conflicts with other messages into consideration.
- The **transport latency** is the total time that is needed to transfer a message over a network link. This is the sum of the transmission time and the time of flight, capturing the entire time interval from putting the first bit of the message onto the network link at the sender and receiving the last bit at the receiver.
- The **sender overhead**, also referred to as *startup time*, is the time that the sender needs for the preparation of message transmission. This includes the time for computing the checksum, appending the header, and executing the routing algorithm.
- The **receiver overhead** is the time that the receiver needs to process an incoming message, including checksum comparison and generation of an acknowledgment if required by the specific protocol.
- The **throughput** of a network link is the effective bandwidth experienced by an application program.

Using these performance measures, the total latency $T(m)$ of a message of size m can be expressed as

$$T(m) = O_{send} + T_{delay} + m/B + O_{recv} \tag{2.1}$$

where O_{send} and O_{recv} are the sender and receiver overhead, respectively, T_{delay} is the time of flight, and B is the bandwidth of the network link. This expression does not take into consideration that a message may need to be transmitted multiple times because of checksum errors, network contention, or congestion.

The performance parameters introduced are illustrated in Fig. 2.26. Equation (2.1) can be reformulated by combining constant terms, yielding

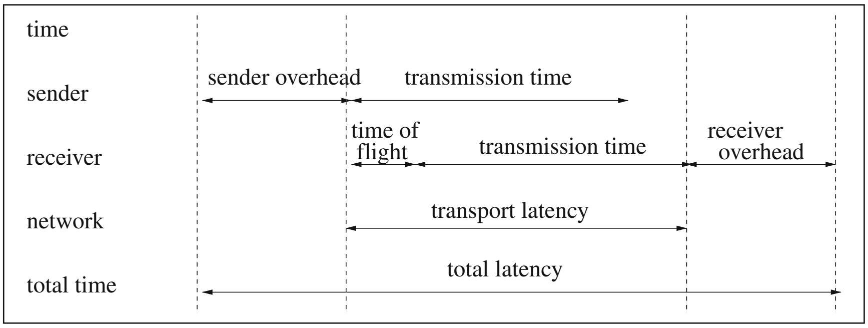


Fig. 2.26 Illustration of performance measures for the point-to-point transfer between neighboring nodes, see [94].

$$T(m) = T_{overhead} + m/B \tag{2.2}$$

with $T_{overhead} = T_{send} + T_{recv}$. Thus, the latency consists of an overhead which does not depend on the message size and a term which linearly increases with the message size. Using the byte transfer time $t_B = 1/B$, Eq. (2.2) can also be expressed as

$$T(m) = T_{overhead} + t_B \cdot m. \tag{2.3}$$

This equation is often used to describe the message transmission time over a network link. When transmitting a message between two nodes that are not directly connected in the network, the message must be transmitted along a path between the two nodes. For the transmission along the path, several switching techniques can be used, including circuit switching, packet switching with store-and-forward routing, virtual cut-through routing, and wormhole routing. We give a short overview in the following.

2.6.3.2 Circuit switching

The two basic switching strategies are circuit switching and packet switching, see [41, 94] for a detailed treatment. In **circuit switching**, the entire path from the source node to the destination node is established and reserved until the end of the transmission of this message. This means that the path is established exclusively for this message by setting the switches or routers on the path in a suitable way. Internally, the message can be split into pieces for the transmission. These pieces can be so-called *physical units* (*phits*) denoting the amount of data that can be transmitted over a network link in one cycle. The size of the phits is determined by the number of bits that can be transmitted over a physical channel in parallel. Typical phit sizes lie between 1 and 256 bits. The transmission path for a message can be established by using short *probe messages* along the path. After the path is established, all phits of the message are transmitted over this path. The path can be released again by a message trailer or by an acknowledgment message from the receiver of the sender.

Sending a control message along a path of length l takes time $l \cdot t_c$ where t_c is the time to transmit the control message over a single network link. If m_c is the size of the control message, it is $t_c = t_B \cdot m_c$. After the path has been established, the transmission of the actual message of size m takes time $m \cdot t_B$. Thus, the total time of message transmission along a path of length l with circuit switching is

$$T_{cs}(m, l) = T_{overhead} + t_c \cdot l + t_B \cdot m. \tag{2.4}$$

If m_c is small compared to m, this can be reduced to $T_{overhead} + t_B \cdot m$ which is linear in m, but independent of l. Message transfer with circuit switching is illustrated in Fig. 2.28 a).

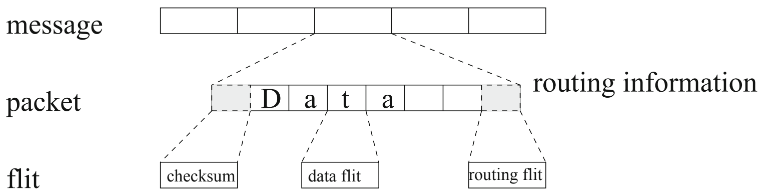


Fig. 2.27 Illustration of the partitioning of a message into packets and of packets into *flits* (*flow control units*).

2.6.3.3 Packet switching

For **packet switching** , the message to be transmitted is partitioned into a sequence of packets which are transferred independently from each other through the network from the sender to the receiver. Using an adaptive routing algorithm, the packets can be transmitted over different paths. Each packet consists of three parts: (i) a header, containing routing and control information, (ii) the data part, containing a part of the original message, and (iii) a trailer which may contain an error control code. Each packet is sent separately to the destination according to the routing information contained in the packet. Figure 2.27 illustrates the partitioning of a message into packets. The network links and buffers are used by one packet at a time.

Packet switching can be implemented in different ways. Packet switching with **store-and-forward routing** sends a packet along a path such that the entire packet is received by each switch on the path (*store*), before it is sent to the next switch on the path (*forward*). The connection between two switches A and B on the path is released for reuse by another packet as soon as the packet has been stored at B. This strategy is useful if the links connecting the switches on a path have different bandwidth as this is typically the case in *wide area networks* (WANs). In this case, store-and-forward routing allows the utilization of the full bandwidth for every link on the path. Another advantage is that a link on the path can be quickly released as soon as the packet has passed the links, thus reducing the danger of deadlocks. The drawback of this strategy is that the packet transmission time increases with the number of switches that must be traversed from source to destination. Moreover, the entire packet must be stored at each switch on the path, thus increasing the memory demands of the switches.

The time for sending a packet of size m over a single link takes time $t_h + t_B \cdot m$ where t_h is the constant time needed at each switch to store the packet in a receive buffer and to select the output channel to be used by inspecting the header information of the packet. Thus, for a path of length l, the entire time for packet transmission with store-and-forward routing is

$$T_{sf}(m,l) = t_S + l(t_h + t_B \cdot m). \tag{2.5}$$

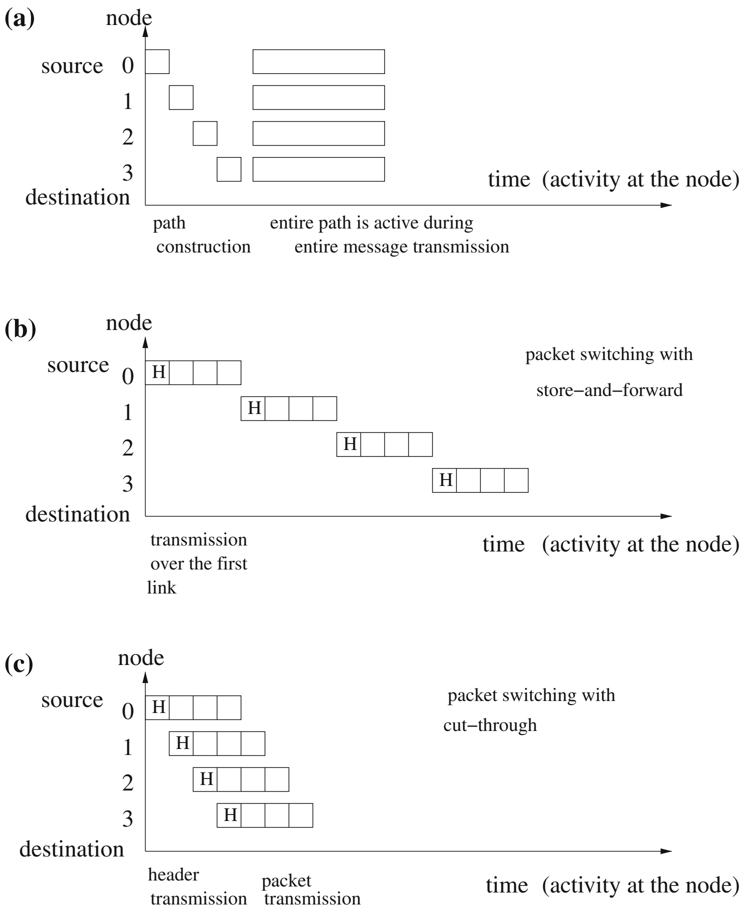

(a)
node
source 0
1
2
3
destination
time (activity at the node)
path construction
entire path is active during entire message transmission
(b)
node
source 0
1
2
3
destination
H
packet switching with store–and–forward
transmission over the first link
time (activity at the node)
(c)
node
source 0
1
2
3
destination
H
packet switching with cut–through
header transmission
packet transmission
time (activity at the node)

Fig. 2.28 Illustration of the latency of a point-to-point transmission along a lath of length $l = 4$ for **(a)** circuit-switching, **(b)** packet-switching with store-and-forward and **(c)** packet-switching with cut-through.

Since t_h is typically small compared to the other terms, this can be reduced to $T_{sf}(m, l) \approx t_S + l \cdot t_B \cdot m$. Thus, the time for packet transmission depends linearly on the packet size and the length l of the path. Packet transmission with store-and-forward routing is illustrated in Fig. 2.28 (b). The time for the transmission of an entire message, consisting of several packets, depends on the specific routing algorithm used. When using a deterministic routing algorithm, the message transmission time is the sum of the transmission time of all packets of the message, if no network delays occur. For adaptive routing algorithms, the transmission of the individual packets can be overlapped, thus potentially leading to a smaller message transmission time.

If all packets of a message are transmitted along the same path, **pipelining** can be used to reduce the transmission time of messages: using pipelining, the packets of a message are sent along a path such that the links on the path are used by successive packets in an overlapping way. Using pipelining for a message of size m and packet size m_p, the time of message transmission along a path of length l can be described by

$$t_S + (m - m_p)t_B + l(t_h + t_B \cdot m_p) \approx t_S + m \cdot t_B + (l - 1)t_B \cdot m_p \qquad (2.6)$$

where $l(t_h + t_B \cdot m_p)$ is the time that elapses before the first packet arrives at the destination node. After this time, a new packet arrives at the destination in each time step of size $m_p \cdot t_B$, assuming the same bandwidth for each link on the path.

2.6.3.4 Cut-through routing

The idea of the pipelining of message packets can be extended by applying pipelining to the individual packets. This approach is taken by **cut-through routing**. Using this approach, a message is again split into packets as required by the packet-switching approach. The different packets of a message can take different paths through the network to reach the destination. Each individual packet is sent through the network in a pipelined way. To do so, each switch on the path inspects the first few *phits* (*physical units*) of the packet header, containing the routing information, and then determines over which output channel the packet is forwarded. Thus, the transmission path of a packet is established by the packet header and the rest of the packet is transmitted along this path in a pipelined way. A link on this path can be released as soon as all *phits* of the packet, including a possible trailer, have been transmitted over this link.

The time for transmitting a header of size m_H along a single link is given by $t_H = t_B \cdot m_H$. The time for transmitting the header along a path of length l is then given by $t_H \cdot l$. After the header has arrived at the destination node, the additional time for the arrival of the rest of the packet of size m is given by $t_B(m - m_H)$. Thus, the time for transmitting a packet of size m along a path of length l using packet-switching with cut-through routing can be expressed as

$$T_{ct}(m, l) = t_S + l \cdot t_H + t_B \cdot (m - m_H) \ . \qquad (2.7)$$

If m_H is small compared to the packet size m, this can be reduced to $T_{ct}(m, l) \approx t_S + t_B \cdot m$. If all packets of a message use the same transmission path, and if packet transmission is also pipelined, this formula can also be used to describe the transmission time of the entire message. Message transmission time using packet-switching with cut-through routing is illustrated in Fig. 2.28 (c).

Until now, we have considered the transmission of a single message or packet through the network. If multiple transmissions are performed concurrently, network contention may occur because of conflicting requests to the same links. This increases the communication time observed for the transmission. The switching strategy must react appropriately if contention happens on one of the links of a transmission path.

Using store-and-forward routing, the packet can simply be buffered until the output channel is free again.

With cut-through routing, two popular options are available: *virtual cut-through routing* and *wormhole routing*. Using **virtual cut-through routing**, in case of a blocked output channel at a switch, all phits of the packet in transmission are collected in a buffer at the switch until the output channel is free again. If this happens at every switch on the path, cut-through routing degrades to store-and-forward routing. Using *partial cut-through routing*, the transmission of the buffered phits of a packet can continue as soon as the output channel is free again, i.e., not all phits of a packet need to be buffered.

The **wormhole routing** approach is based on the definition of *flow control units (flits)* which are usually at least as large as the packet header. The header *flit* establishes the path through the network. The rest of the flits of the packet follow in a pipelined way on the same path. In case of a blocked output channel at a switch, only a few flits are stored at this switch, the rest is kept on the preceding switches of the path. Therefore, a blocked packet may occupy buffer space along an entire path or at least a part of the path. Thus, this approach has some similarities to circuit switching at packet level. Storing the *flits* of a blocked message along the switches of a path may cause other packets to block, leading to network saturation. Moreover, deadlocks may occur because of cyclic waiting, see Fig. 2.29 [144, 177]. An advantage of the wormhole routing approach is that the buffers at the switches can be kept small, since they need to store only a small portion of a packet.

Since buffers at the switches can be implemented large enough with today's technology, virtual cut-through routing is the more commonly used switching technique

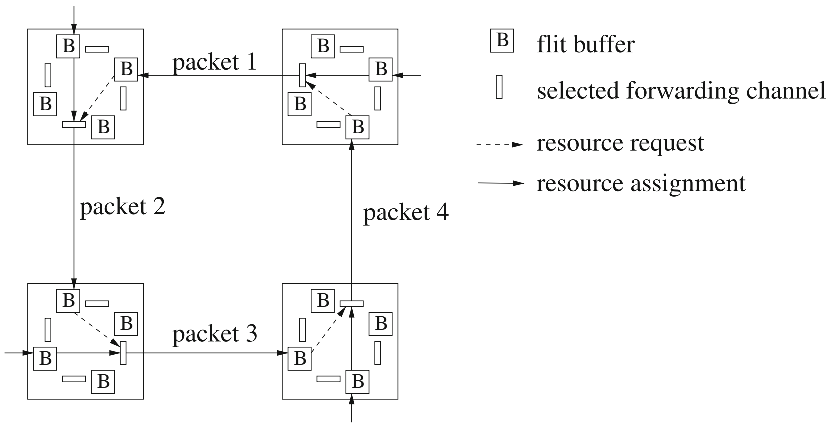


Fig. 2.29 Illustration of a deadlock situation with wormhole routing for the transmission of four packets over four switches. Each of the packets occupies a flit buffer and requests another flit buffer at the next switch; but this flit buffer is already occupied by another packet. A deadlock occurs, since none of the packets can be transmitted to the next switch.

[94]. The danger of deadlocks can be avoided by using suitable routing algorithms like dimension-ordered routing or by using virtual channels, see Sect. 2.6.1.

2.6.4 Flow control mechanisms

A general problem in network may arise form the fact that multiple messages can be in transmission at the same time and may attempt to use the same network links at the same time. If this happens, some of the message transmissions must be blocked while others are allowed to proceed. Techniques to coordinate concurrent message transmissions in networks are called *flow control mechanisms*. Such techniques are important in all kinds of networks, including local and wide area networks, and popular protocols like TCP contain sophisticated mechanisms for flow control to obtain a high effective network bandwidth, see [125, 157] for more details. Flow control is especially important for networks of parallel computers, since these must be able to transmit a large number of messages fast and reliably. A loss of messages cannot be tolerated, since this would lead to errors in the parallel program currently executed.

Flow control mechanisms typically try to avoid congestion in the network to guarantee fast message transmission. An important aspect is the flow control mechanisms at the link level which considers message or packet transmission over a single link of the network. The link connects two switches A and B. We assume that a packet should be transmitted from A to B. If the link between A and B is free, the packet can be transferred from the output port of A to the input port of B from which it is forwarded to the suitable output port of B. But if B is busy, there might be the situation that B does not have enough buffer space in the input port available to store the packet from A. In this case, the packet must be retained in the output buffer of A until there is enough space in the input buffer of B. But this may cause back pressure to switches preceding A, leading to the danger of network congestion. The idea of link-level flow control mechanisms is that the receiving switch provides a feedback to the sending switch, if not enough input buffer space is available to prevent the transmission of additional packets. This feedback rapidly propagates backwards in the network until the original sending node is reached. This sender can then reduce its transmission rate to avoid further packet delays.

Link-level flow control can help to reduce congestion, but the feedback propagation might be too slow and the network might already be congested when the original sender is reached. An *end-to-end flow control* with a direct feedback to the original sender may lead to a faster reaction. A windowing mechanism as it is used by the TCP protocol is one possibility for a implementation. Using this mechanism, the sender is provided with the available buffer space at the receiver and can adapt the number of packets sent such that no buffer overflow occurs. More information can be found in [125, 157, 94, 41].

2.7 Caches and Memory Hierarchy

A significant characteristic of the hardware development during the last decades has been the increasing gap between processor cycle time and main memory access time as described in Sect. 2.1. One of the reasons for this development has been the large increase of the clock frequency of the processors, whereas the memory access time could not be reduced significantly. We now have a closer look at this development: The main memory is constructed based on **DRAM** (dynamic random access memory). For a typical processor with a clock frequency of 3 GHz, corresponding to a cycle time of 0.33 ns, a memory access takes between 60 and 210 machine cycles, depending on the DRAM chip used. The use of caches can help to reduce this memory access time significantly. Caches are built with **SRAM** (static random access memory) chips, and SRAM chips are significantly faster than DRAM, but have a smaller capacity per unit area and are more costly.

In 2012, typical cache access times are about 1 ns for an L1 cache, between 3 and 10 ns for an L2 cache, and between 10 and 20 ns for an L3 cache. For an Intel Core i7 processor (Sandy Bridge), the access time to the L1 cache is 4–6 machine cycles, the access time to the L2 cache is 21 machine cycles, and the access time to the L3 cache is 26–31 machine cycles, a detailed description is given in Sect. 2.4.5. On the other hand, for a 3.3 GHz CPU and a DDR 1600 SDRAM based main memory, a total memory access time of 135 machine cycles results, taking the cache miss control into consideration. If the working space of an application is larger than the capacity of the main memory, a part of the working space must be kept on disk. However, a disk access takes considerably longer than an access to main memory, mainly caused by the mechanical movement of the read/write head: the typical disk access time lies between 10 and 20 ms, which corresponds to 30 to 100 millions machine cycles for a usual CPU frequency. Therefore, storing application data on disk can slow down the execution time considerably. A disk access loads a complete page block, which typically has the size 4 or 8 Kbytes to amortize the time for disk accesses in situation when the main memory is too small for the application data.
The simplest form of a memory hierarchy is the use of a single cache between the processor and main memory (one-level cache, L1 cache). The cache contains a subset of the data stored in the main memory, and a replacement strategy is used to bring new data from the main memory into the cache, replacing data elements that are no longer accessed. The goal is to keep those data elements in the cache which are currently used most. Today, two or three levels of cache are used for each processor, using a small and fast L1 cache and larger, but slower L2 and L3 caches.

For multiprocessor systems where each processor uses a separate local cache, there is the additional problem of keeping a consistent view to the shared address space for all processors. It must be ensured that a processor accessing a data element always accesses the most recently written data value, also in the case that another processor has written this value. This is also referred to as **cache coherence problem**, and will be considered in more detail in Sect. 2.7.3.

For multiprocessors with a shared address space, the top level of the memory hierarchy is the shared address space that can be accessed by each of the processors. The design of a memory hierarchy may have a large influence on the execution time of parallel programs, and memory accesses should be ordered such that a given memory hierarchy is used as efficiently as possible. Moreover, techniques to keep a memory hierarchy consistent may also have an important influence. In this section, we therefore give an overview of memory hierarchy designs and discuss issues of cache coherence and memory consistency. Since caches are the building blocks of memory hierarchies and have a significant influence on the memory consistency, we give a short overview of caches in the following subsection. For a more detailed treatment, we refer to [41, 94, 89, 155].

2.7.1 Characteristics of Caches

A cache is a small, but fast memory between the processor and the main memory. Caches are built with SRAM. Typical access times are 0.5 - 2.5 ns (ns = nanoseconds = 10^{-9} seconds) compared to 20-70 ns for DRAM [94]). In the following, we consider a one-level cache first. A cache contains a copy of a subset of the data in main memory. Data are moved in blocks, containing a small number of words, between the cache and main memory, see Fig. 2.30. These blocks of data are called **cache blocks** or **cache lines**. The size of the cache lines is fixed for a given architecture and cannot be changed during program execution.

Cache control is decoupled from the processor and is performed by a separate cache controller. During program execution, the processor specifies memory addresses to be read or to be written as given by the load and store operations of the machine program. The processor forwards the memory addresses to the memory system and waits until the corresponding values are returned or written. The processor specifies memory addresses independently of the organization of the memory system, i.e., the processor does not need to know the architecture of the memory system. After having received a memory access request from the processor, the cache controller checks whether the memory address specified belongs to a cache line which is currently stored in the cache. If this is the case, a **cache hit** occurs, and the requested word is delivered to the processor from the cache. If the corresponding cache line is not stored in the cache, a **cache miss** occurs, and the cache line is first copied from main memory into the cache before the requested word is delivered to

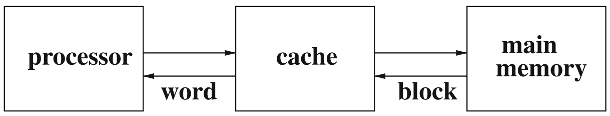


Fig. 2.30 Data transport between cache and main memory is done by the transfer of memory blocks comprising several words, whereas the processor accesses single words in the cache.

the processor. The corresponding delay time is also called **miss penalty**. Since the access time to main memory is significantly larger than the access time to the cache, a cache miss leads to a delay of operand delivery to the processor. Therefore, it is desirable to reduce the number of cache misses as much as possible.

The exact behavior of the cache controller is hidden from the processor. The processor observes that some memory accesses take longer than others, leading to a delay in operand delivery. During such a delay, the processor can perform other operations that are independent of the delayed operand. This is possible, since the processor is not directly occupied for the operand access from the memory system. Techniques like *operand prefetch* can be used to support an anticipated loading of operands, so that other independent operations can be executed, see [94].

The number of cache misses may have a significant influence on the resulting runtime of a program. If many memory accesses lead to cache misses, the processor may often have to wait for operands, and program execution may be quite slow. Since the cache management is implemented in hardware, the programmer cannot directly specify which data should reside in the cache at which point in program execution. But the order of memory accesses in a program can have a large influence on the resulting runtime, and a reordering of the memory accesses may lead to a significant reduction of program execution time. In this context, the **locality of memory accesses** is often used as a characterization of the memory accesses of a program. Spatial and temporal locality can be distinguished as follows:

- The memory accesses of a program have a high **spatial locality**, if the program often accesses memory locations with neighboring addresses at successive points in time during program execution. Thus, for programs with high spatial locality there is often the situation that after an access to a memory location, one or more memory locations of the same cache line are also accessed shortly afterwards. In such situations, after loading a cache block, several of the following memory locations can be loaded from this cache block, thus avoiding expensive cache misses. The use of cache blocks comprising several memory words is based on the assumption that most programs exhibit spatial locality, i.e., when loading a cache block not only one but several memory words of the cache block are accessed before the cache block is replaced again.
- The memory accesses of a program have a high **temporal locality**, if it often happens that the *same* memory location is accessed multiple times at successive points in time during program execution. Thus, for programs with a high temporal locality there is often the situation that after loading a cache block in the cache, the memory words of the cache block are accessed multiple times before the cache block is replaced again.

For programs with small spatial locality there is often the situation that after loading a cache block, only one of the memory words contained is accessed before the cache block is replaced again by another cache block. For programs with small temporal locality, there is often the situation that after loading a cache block because of a memory access, the corresponding memory location is accessed only once before the cache block is replaced again. Many program transformations to increase temporal or spatial locality of programs have been proposed, see [12, 197] for more details.

In the following, we give a short overview of important characteristics of caches. In particular, we consider cache size, mapping of memory blocks to cache blocks, replacement algorithms, and write-back policies. We also consider the use of multi-level caches.

2.7.1.1 Cache size

Using the same hardware technology, the access time of a cache increases (slightly) with the size of the cache because of an increased complexity of the addressing. But using a larger cache leads to a smaller number of replacements as a smaller cache, since more cache blocks can be kept in the cache. The size of the caches is limited by the available chip area. Off-chip caches are rarely used to avoid the additional time penalty of off-chips accesses. Typical sizes for L1 caches lie between 8K and 128K memory words where a memory word is four or eight bytes long, depending on the architecture. During the last years, the typical size of L1 caches has not been increased significantly.

If a cache miss occurs when accessing a memory location, an entire cache block is brought into the cache. For designing a memory hierarchy, the following points have to be taken into consideration when fixing the size of the cache blocks:

- Using larger blocks reduces the number of blocks that fit in the cache when using the same cache size. Therefore, cache blocks tend to be replaced earlier when using larger blocks compared to smaller blocks. This suggests to set the cache block size as small as possible.
- On the other hand, it is useful to use blocks with more than one memory word, since the transfer of a block with x memory words from main memory into the cache takes less time than x transfers of a single memory word. This suggests to use larger cache blocks.

As a compromise, a medium block size is used. Typical sizes for L1 cache blocks are four or eight memory words.

2.7.1.2 Mapping of memory blocks to cache blocks

Data are transferred between main memory and cache in blocks of a fixed length. Because the cache is significantly smaller than the main memory, not all memory blocks can be stored in the cache at the same time. Therefore, a mapping algorithm must be used to define at which position in the cache a memory block can be stored. The mapping algorithm used has a significant influence on the cache behavior and determines how a stored block is localized and retrieved from the cache. For the mapping, the notion of **associativity** plays an important role. Associativity determines at how many positions in the cache a memory block can be stored. The following methods are distinguished:

- for a **direct mapped** cache, each memory block can be stored at exactly one position in the cache;
- for a **fully associative** cache, each memory block can be stored at an arbitrary position in the cache;
- for a **set associative** cache, each memory block can be stored at a fixed number of positions.

In the following, we consider these three mapping methods in more detail for a memory system which consists of a main memory and a cache. We assume that the main memory comprises $n = 2^s$ blocks which we denote as B_j for $j = 0, \ldots n-1$. Furthermore, we assume that there are $m = 2^r$ cache positions available; we denote the corresponding cache blocks as $\bar{B}_i$ for $i = 0, \ldots, m-1$. The memory blocks and the cache blocks have the same size of $l = 2^w$ memory words. At different points of program execution, a cache block may contain different memory blocks. Therefore, for each cache block a **tag** must be stored, which identifies the memory block that is currently stored. The use of this tag information depends on the specific mapping algorithm and will be described in the following. As running example, we consider a memory system with a cache of size 64 Kbytes which uses cache blocks of 4 bytes. Thus, 16K = 2^{14} blocks of four bytes each fit into the cache. With the notation from above, it is $r = 14$ and $w = 2$. The main memory is 4 Gbytes = 2^{32} bytes large, i.e., it is $s = 30$ if we assume that a memory word is one byte. We now consider the three mapping methods in turn.

2.7.1.3 Direct mapped caches

The simplest form to map memory blocks to cache blocks is implemented by **direct mapped** caches. Each memory block B_j can be stored at only one specific cache location. The mapping of a memory block B_j to a cache block $\bar{B}_i$ is defined as follows:

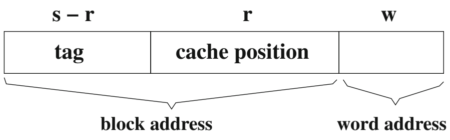
$$B_j \text{ is mapped to } \bar{B}_i \text{ if } i = j \bmod m.$$

Thus, there are $n/m = 2^{s-r}$ different memory blocks that can be stored in one specific cache block $\bar{B}_i$. Based on the mapping, memory blocks are assigned to cache positions as follows:

cache block	memory block
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m+1, 2m+1, \ldots, 2^s - m + 1$
$\vdots$	$\vdots$
$m-1$	$m-1, 2m-1, 3m-1, \ldots, 2^s - 1$

Since the cache size m is a power of 2, the modulo operation specified by the mapping function can be computed by using the low-order bits of the memory address specified by the processor. Since a cache block contains $l = 2^w$ memory words, the memory address can be partitioned into a word address and a block address. The block address specifies the position of the corresponding memory block in main memory. It consists of the s most significant (leftmost) bits of the memory address. The word address specifies the position of the memory location in the memory block, relatively to the first location of the memory block. It consists of the w least significant (rightmost) bits of the memory address.

For a direct-mapped cache, the r rightmost bits of the block address of a memory location define at which of the $m = 2^r$ cache positions, the corresponding memory block must be stored if the block is loaded into the cache. The remaining $s - r$ bits can be interpreted as tag which specifies which of the 2^{s-r} possible memory blocks is currently stored at a specific cache position. This tag must be stored with the cache block. Thus, each memory address is partitioned as follows:



For the running example, the tags consist of $s - r = 16$ bits for a direct mapped cache.

Memory access is illustrated in Fig. 2.31 (a) for an example memory system with block size 2 ($w = 1$), cache size 4 ($r = 2$), and main memory size 16 ($s = 4$). For each memory access specified by the processor, the cache position at which the requested memory block must be stored is identified by considering the r rightmost bits of the block address. Then the tag stored for this cache position is compared with the $s - r$ leftmost bits of the block address. If both tags are identical, the referenced memory block is currently stored in the cache and the memory access can be done via the cache. A *cache hit* occurs. If the two tags are different, the requested memory block must first be loaded into the cache at the given cache position before the memory location specified can be accessed.

Direct-mapped caches can be implemented in hardware without great effort, but they have the disadvantage that each memory block can be stored at only one cache position. Thus, it can happen that a program repeatedly specifies memory addresses in different memory blocks that are mapped to the same cache position. In this situation, the memory blocks will be continually loaded and replaced in the cache, leading to a large number of cache misses and therefore a large execution time. This phenomenon is also called *thrashing*.

2.7.1.4 Fully associative caches

In a fully associative cache, each memory block can be placed in *any* cache position, thus overcoming the disadvantage of direct-mapped caches. As for direct mapped

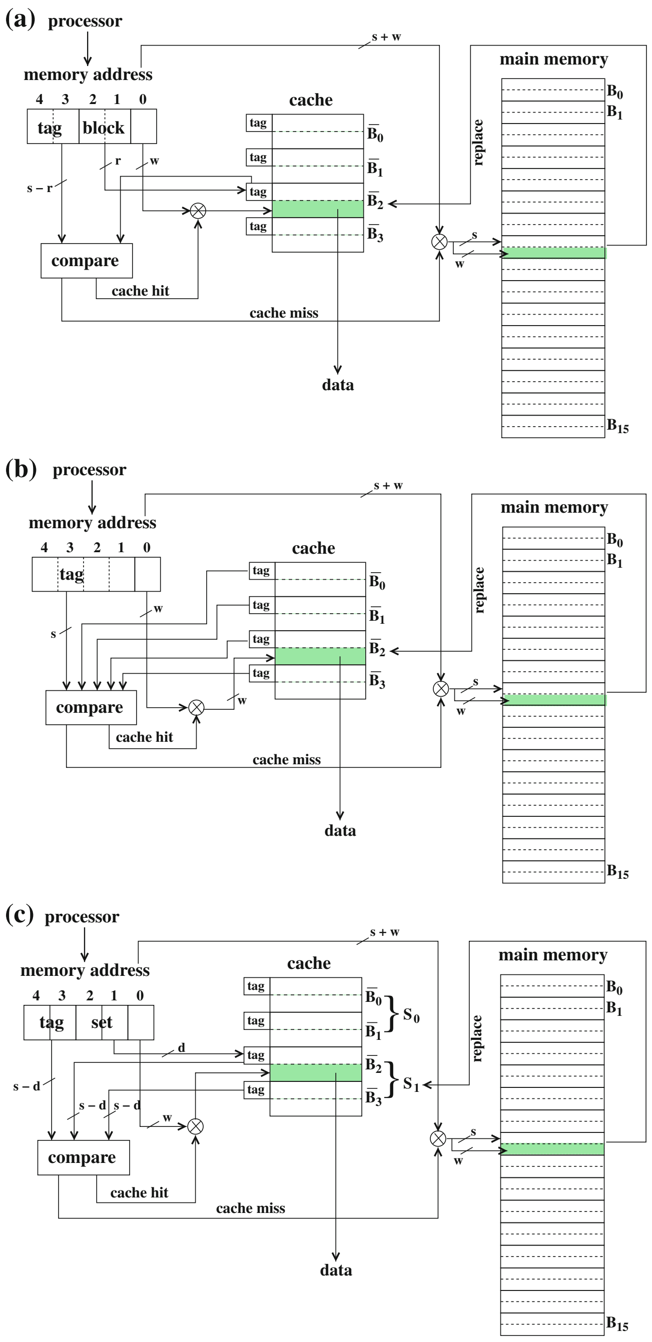
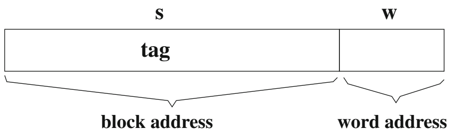


Fig. 2.31 Illustration of the mapping of memory blocks to cache blocks for a cache with $m = 4$ cache blocks ($r = 2$) and a main memory with $n = 16$ memory blocks ($s = 4$). Each block contains two memory words ($w = 1$). **(a)** direct mapped cache; **(b)** fully associative cache; **(c)** set associative cache with $k = 2$ blocks per set, using $v = 2$ sets ($d = 1$).

caches, a memory address can again be partitioned into a block address (s leftmost bits) and a word address (w rightmost bits). Since each cache block can contain any memory block, the entire block address must be used as tag and must be stored with the cache block to allow the identification of the memory block stored. Thus, each memory address is partitioned as follows:



To check whether a given memory block is stored in the cache, all the entries in the cache must be searched, since the memory block can be stored at any cache position. This is illustrated in Fig. 2.31 (b).

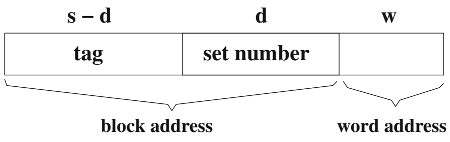
The advantage of fully associative caches lies in the increased flexibility when loading memory blocks into the cache. The main disadvantage is that for each memory access all cache positions must be considered to check whether the corresponding memory block is currently held in the cache. To make this search practical, it must be done in parallel using a separate comparator for each cache position, thus increasing the required hardware effort significantly. Another disadvantage is that the tags to be stored for each cache block are significantly larger as for direct-mapped caches. For the example cache introduced above, the tags must be 30 bits long for a fully associated cache, i.e., for each 32 bit memory block, a 30 bit tag must be stored. Because of the large search effort, a fully associative mapping is useful only for caches with a small number of positions.

2.7.1.5 Set associative caches

Set associative caches are a compromise between direct mapped and fully associative caches. In a set associative cache, the cache is partitioned into v sets $S_0, \dots, S_{v-1}$ where each set consists of $k = m/v$ blocks. A memory block B_j is not mapped to an individual cache block, but to a unique set in the cache. Within the set, the memory block can be placed in any cache block of that set, i.e., there are k different cache blocks in which a memory block can be stored. The set of a memory block B_j is defined as follows:

$$B_j \text{is mapped to set } S_i, \text{if } i = j \bmod v.$$

for $j = 0, \dots, n-1$. A memory access is illustrated in Fig. 2.31 (c). Again, a memory address consists of a block address (s bits) and a word address (w bits). The $d = \log v$ rightmost bits of the block address determine the set S_i to which the corresponding memory block is mapped. The leftmost $s - d$ bits of the block address are the tag that is used for the identification of the memory blocks stored in the individual cache blocks of a set. Thus, each memory address is partitioned as follows:



When a memory access occurs, the hardware first determines the set to which the memory block is assigned. Then, the tag of the memory block is compared with the tags of all cache blocks in the set. If there is a match, the memory access can be performed via the cache. Otherwise, the corresponding memory block must first be loaded into one of the cache blocks of the set.

For $v = m$ and $k = 1$, a set associative cache reduces to a direct-mapped cache. For $v = 1$ and $k = m$, a fully associative cache results. Typical cases are $v = m/4$ and $k = 4$, leading to a *4-way set associative cache*, and $v = m/8$ and $k = 8$, leading to an *8-way set associative cache*. For the example cache, using $k = 4$ leads to 4K sets; $d = 12$ bits of the block address determine the set to which a memory block is mapped. The tags used for the identification of memory blocks within a set are 18 bits long.

2.7.1.6 Block replacement methods

When a cache miss occurs, a new memory block must be loaded into the cache. To do this for a fully occupied cache, one of the memory blocks in the cache must be replaced. For a direct-mapped cache, there is only one position at which the new memory block can be stored, and the memory block occupying that position must be replaced. For a fully associative or set associative cache, there are several positions at which the new memory block can be stored. The block to be replaced is selected using a *replacement method*. A popular replacement method is **least recently used (LRU)** which replaces the block in a set that has not been used for the longest time.

For the implementation of the LRU method, the hardware must keep track for each block of a set when the block has been used last. The corresponding time entry must be updated at each usage time of the block. This implementation requires additional space to store the time entries for each block and additional control logic to update the time entries. For a 2-way set associative cache, the LRU method can be implemented more easily by keeping a USE bit for each of the two blocks in a set. When a cache block of a set is accessed, its USE bit is set to 1 and the USE bit of the other block in the set is set to 0. This is performed for each memory access. Thus, the block whose USE bit is 1 has been accessed last, and the other block should be replaced if a new block has to be loaded into the set. An alternative to LRU is *least frequently used* (LFU) which replaces the block of a set that has experienced the fewest references. But the LFU method also requires additional control logic since for each block a counter must be maintained which must be updated for each memory access. For a larger associativity, an exact implementation of LRU or LFU as described above is often considered as too costly [94], and approximations or other schemes are used. Often, the block to be replaced is selected *randomly*, since this can be implemented easily. Moreover, simulations have shown that random replacement leads to only slightly inferior performance as more sophisticated methods like LRU or LFU [94, 184].

2.7.2 Write Policy

A cache contains a subset of the memory blocks. When the processor issues a *write access* to a memory block that is currently stored in the cache, the referenced block is definitely updated in the cache, since the next read access must return the most recent value. There remains the question: When is the corresponding memory block in the main memory updated? The earliest possible update time for the main memory is immediately after the update in the cache; the latest possible update time for the main memory is when the cache block is replaced by another block. The exact replacement time and update method is captured by the write policy. The most popular policies are **write-through** and **write-back**.

2.7.2.1 Write-through policy

Using write-through, a modification of a block in the cache using a write access is immediately transferred to main memory, thus keeping the cache and the main memory consistent. An advantage of this approach is that other devices like I/O modules that have direct access to main memory always get the newest value of a memory block. This is also important for multicore systems, since after a write by one processor, all other processors always get the most recently written value when accessing the same block. A drawback of write-through is that every write in the cache causes also a write to main memory which typically takes at least 100 processor cycles to complete. This could slow down the processor if it had to wait for the completion. To avoid processor waiting, a *write buffer* can be used to store pending write operations into the main memory [155, 94]. After writing the data into the cache and into the write buffer, the processor can continue its execution without waiting for the completion of the write into the main memory. A write buffer entry can be freed after the write into main memory completes. When the processor performs a write and the write buffer is full, a write stall occurs, and the processor must wait until there is a free entry in the write buffer.

2.7.2.2 Write-back policy

Using write-back, a write operation to a memory block that is currently held in the cache is performed only in the cache; the corresponding memory entry is not updated immediately. Thus, the cache may contain newer values than the main memory. The modified memory block is written to the main memory when the cache block is replaced by another memory block. To check whether a write to main memory is necessary when a cache block is replaced, a separate bit (*dirty bit*) is held for each cache block which indicates whether the cache block has been modified or not. The dirty bit is initialized with 0 when a block is loaded into the cache. A write access to a cache block sets the dirty bit to 1, indicating that a write to main memory must be performed when the cache block is replaced.

Using write-back policy usually leads to fewer write operations to main memory than write-through, since cache blocks can be written multiple times before they are written back to main memory. The drawback of write-back is that the main memory may contain invalid entries, and hence I/O modules can access main memory only through the cache.

If a write to a memory location goes to a memory block that is currently not in the cache, most caches use the *write-allocate* method: the corresponding memory block is first brought into the cache, and then the modification is performed as described above. An alternative approach is *write no allocate*, which modifies in main memory without loading it into the cache. However, this approach is used less often.

2.7.2.3 Number of caches

So far, we have considered the behavior of a single cache which is placed between the processor and main memory and which stores data blocks of a program in execution. Such caches are also called **data caches**.

Besides the program data, a processor also accesses instructions of the program in execution before they are decoded and executed. Because of loops in the program, an instruction can be accessed multiple times. To avoid multiple loading operations from main memory, instructions are also held in cache. To store instructions and data, a single cache can be used (*unified cache*). But often, two separate caches are used on the first level, an **instruction cache** to store instructions and a separate data cache to store data. This approach is also called *split caches*. This enables a greater flexibility for the cache design, since the data and instruction caches can work independently of each other, and may have different size and associativity depending on the specific needs.

In practice, multiple levels of caches are typically used as illustrated in Fig. 2.32. Today, most desktop processors have a three-level cache hierarchy, consisting of a first-level (L1) cache, a second-level (L2) cache, and a third-level (L3) cache. All these caches are integrated onto the chip area.

For the L1 cache, split caches are typically used, for the remaining levels, unified caches are standard. In 2012, typical cache sizes lie between 8 Kbytes and 64 Kbytes for the L1 cache, between 128 Kbytes and 512 Kbytes for the L2 cache, and between 2 Mbytes and 16 Mbytes for the L3 cache. Typical sizes of the main memory lie between 8 Gbytes and 48 Gbytes. The caches are hierarchically organized, and for three levels, the L1 caches contain a subset of the L2 cache, which contains a subset of the L3 cache, which contains a subset of the main memory. Typical access times are one or a few processor cycles for the L1 cache, between 15 and 25 cycles for the L2 cache, between 100 and 300 cycles for main memory, and between 10 and 100 million cycles for the hard disk [94].

2.7.3 Cache coherency

Using a memory hierarchy with multiple levels of caches can help to bridge large access times to main memory. But the use of caches introduces the effect that memory blocks can be held in multiple copies in caches and main memory, and after an update in the L1 cache, other copies might become invalid, in particular if a write-back policy is used. This does not cause a problem as long as a single processor is the only accessing device. But if there are multiple accessing devices, as it is the case for multicore processors, inconsistent copies can occur and should be avoided, and each execution core should always access the most recent value of a memory location. The problem of keeping the different copies of a memory location consistent is also referred to as *cache coherency problem.*

In a multiprocessor system with different cores or processors, in which each processor has a separate local cache, the same memory block can be held as copy in the local cache of multiple processors. If one or more of the processors update a copy of a memory block in their local cache, the other copies become invalid and contain inconsistent values. The problem can be illustrated for a bus-based system with three processors [41] as shown in the following example.

Example We consider a bus-based SMP system with three processors P_1, P_2, P_3 where each processor P_i has a local cache C_i for $i = 1, 2, 3$. The processors are connected to a shared memory M via a central bus. The caches C_i use a write-through strategy. We consider a variable u with initial value 5 which is held in main memory before the following operations are performed at times t_1, t_2, t_3, t_4:

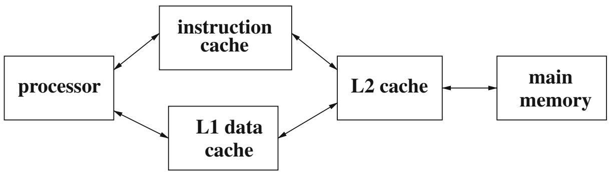


Fig. 2.32 Illustration of a two-level cache hierarchy.

t_1: Processor P_1 reads variable u. The memory block containing u is loaded into cache C_1 of P_1.
t_2: Processor P_3 reads variable u. The memory block containing u is also loaded into cache C_3 of P_3.
t_3: Processor P_3 writes the value 7 into u. This new value is also written into main memory because write-through is used.
t_4: Processor P_1 reads u by accessing the copy in its local cache.

At time t_4, processor P_1 reads the old value 5 instead of the new value 7, i.e., a cache coherency problem occurs. This is the case for both write-through and write-back caches: For write-through caches, at time t_3 the new value 7 is directly written into main memory by processor P_3, but the cache of P_1 will not be updated. For write-back caches, the new value of 7 is not even updated in main memory, i.e., if another processor P_2 reads the value of u after time t_3, it will obtain the old value, even when the variable u is not held in the local cache of P_2. □

For a correct execution of a parallel program on a shared address space, it must be ensured that for each possible order of read and write accesses performed by the participating processors according to their program statements, each processor obtains the right value, no matter whether the corresponding variable is held in cache or not.

The behavior of a memory system for read and write accesses performed by *different* processors to the *same* memory location is captured by the **coherency of the memory system**. Informally, a memory system is coherent if for each memory location any read access returns the most recently written value of that memory location. Since multiple processors may perform write operations to the same memory location at the same time, we first must define more precisely what the most recently written value is. For this definition, the order of the memory accesses in the parallel program executed is used as time measure, not the physical point in time at which the memory accesses are executed by the processors. This makes the definition independent from the specific execution environment and situation.

Using the program order of memory accesses, a memory system is coherent, if the following conditions are fulfilled [94].

1. If a processor P writes into a memory location x at time t_1 and reads from the same memory location x at time $t_2 > t_1$ and if between t_1 and t_2 no other processor performs a write into x, then P obtains at time t_2 the value written by itself at time t_1. Thus, for each processor the order of the memory accesses in its program is preserved despite a parallel execution.
2. If a processor P_1 writes into a memory location x at time t_1 and if another processor P_2 reads x at time $t_2 > t_1$, then P_2 obtains the value written by P_1, if between t_1 and t_2 no other processors write into x and if the period of time $t_2 - t_1$ is sufficiently large. Thus, a value written by one of the processors must become visible to the other processors after a certain amount of time.

3. If two processors write into the same memory location x, these write operations are *serialized*, so that all processors see the write operations in the *same order*. Thus, a global **write serialization** is performed.

To be coherent, a memory system must fulfill these three properties. In particular, for a memory system with caches which can store multiple copies of memory blocks, it must be ensured that each processor has a coherent view to the memory system through its local caches. To ensure this, hardware-based *cache coherence protocols* are used.

These protocols are based on the approach to maintain information about the current modification state for each memory block while it is stored in a local cache. Depending on the coupling of the different components of a parallel system, *snooping protocols* and *directory-based protocols* are used. Snooping protocols are based on the existence of a shared medium over which the memory accesses of the processor cores are transferred. The shared medium can be a shared bus or a shared cache. For directory-based protocols, it is not necessary that such a shared medium exists. In the following, we give a short overview of these protocols and refer to [41, 94, 136] for a more detailed treatment.

2.7.3.1 Snooping protocols

The technique of bus snooping has first been used for bus-based SMP systems for which the local caches of the processors use a write-through policy.

Snooping protocols can be used if all memory accesses are performed via a shared medium which can be observed by all processor cores. For SMP systems, the central bus has been used as shared medium. For current multicore processors, such as the Intel Core i7 processors, the shared medium is the connection between the private L1 and L2 caches of the cores and the shared L3 cache.

In the following, we assume the existence of a shared medium and consider a memory system with a write-through policy of the caches. Especially, we consider the use of a central bus as shared medium.

In this case, the snooping technique relies on the property that on such systems all memory accesses are performed via the central bus, i.e., the bus is used as broadcast medium. Thus, all memory accesses can be observed by the cache controller of each processors. When the cache controller observes a write into a memory location that is currently held in the local cache, it updates the value in the cache by copying the new value from the bus into the cache. Thus, the local caches always contain the most recently written values of memory locations. These protocols are also called *update-based protocols*, since the cache controllers directly perform an update. There are also *invalidation-based protocols* in which the cache block corresponding to a memory block is invalidated, so that the next read access must first perform an update from main memory. Using an update-based protocol in the example from page 85, processor P_1 can observe the write operation of P_3 at time t_3 and can update the value of u in its local cache C_1 accordingly. Thus, at time t_4, P_1 reads the correct value 7.

The technique of bus snooping has first been used for a write-through policy and relies on the existence of a broadcast medium, so that each cache controller can observe all write accesses to perform updates or invalidations. For newer architectures interconnection networks like crossbars or point-to-point networks are used. This makes updates or invalidations more complicated, since the interprocessor links are not shared, and the coherency protocol must use broadcasts to find potentially shared copies of memory blocks, see [94] for more details. Due to the coherence protocol, additional traffic occurs in the interconnection network, which may limit the effective memory access time of the processors. Snooping protocols are not restricted to write-through caches. The technique can also be applied to write-back caches as described in the following.

2.7.3.2 Write-back invalidation protocol

In the following, we describe a basic write-back invalidation protocol, see [41, 94] for more details. In the protocol, each cache block can be in one of three states [4]:

M (modified) means that the cache block contains the current value of the memory block and that all other copies of this memory block in other caches or in the main memory are invalid, i.e., the block has been updated in the cache;

S (shared) means that the cache block has not been updated in this cache and that this cache contains the current value, as does the main memory and zero or more other caches;

I (invalid) means that the cache block does not contain the most recent value of the memory block.

According to these three states, the protocol is also called **MSI protocol**. The same memory block can be in different states in different caches. Before a processor modifies a memory block in its local cache, all other copies of the memory block in other caches and the main memory are marked as invalid (I). This is performed by an operation on the broadcast medium. After that, the processor can perform one or more write operations to this memory block without performing other invalidations. The memory block is marked as modified (M) in the cache of the writing processor. The protocol provides three operations on the broadcast medium, which is a shared bus in the simplest case:

- **Bus Read** (`BusRd`): This operation is generated by a read operation (`PrRd`) of a processor to a memory block that is currently not stored in the cache of this processor. The cache controller requests a copy of the memory block by specifying the corresponding memory address. The requesting processor does not intend to modify the memory block. The most recent value of the memory block is provided from main memory or from another cache.

- **Bus Read Exclusive** (`BusRdEx`): This operation is generated by a write operation (`PrWr`) of a processor to a memory block that is currently not stored in the cache of this processor or that is currently not in the M state in this cache. The cache controller requests an exclusive copy of the memory block that it intends to modify; the request specifies the corresponding memory address. The memory system provides the most recent value of the memory block. All other copies of this memory block in other caches are marked invalid (I).
- **Write Back** (`BusWr`): The cache controller writes a cache block that is marked as modified (M) back to main memory. This operation is generated if the cache block is replaced by another memory block. After the operation, the main memory contains the latest value of the memory block.

The processor performs the usual read and write operations (`PrRd`, `PrWr`) to memory locations, see Fig. 2.33 (right). The cache controller provides the requested memory words to the processor by loading them from the local cache. In case of a cache miss, this includes the loading of the corresponding memory block using a bus operation. The exact behavior of the cache controller depends on the state of the cache block addressed and can be described by a state transition diagram that is shown in Fig. 2.33 (left).

A read and write operation to a cache block marked with M can be performed in the local cache without a bus operation. The same is true for a read operation to a cache block that is marked with S. To perform a write operation to a cache block marked with S, the cache controller first must execute a `BusRdEx` operation to become the exclusive owner of the cache block. The local state of the cache block is transformed from S to M. The cache controllers of other processors that have a local copy of the same cache block with state S observe the `BusRdEx` operation and perform a local state transition from S to I for this cache block.

When a processor tries to read a memory block that is not stored in its local cache or that is marked with I in its local cache, the corresponding cache controller performs a `BusRd` operation. This causes a valid copy to be stored in the local cache marked with S. If another processor observes a `BusRd` operation for a memory block, for which it has the only valid copy (state M), it puts the value of the memory block on the bus and marks its local copy with state S (shared).

When a processor tries to write into a memory block that is not stored in its local cache or that is marked with I, the cache controller performs a `BusRdEx` operation. This provides a valid copy of the memory block in the local cache, which is marked with M, i.e., the processor is the exclusive owner of this memory block. If another processor observes a `BusRdEx` operation for a memory block which is marked with M in its local cache, it puts the value of the memory block on the bus and performs a local state transition from M to I.

A drawback of the MSI protocol is that a processor which first reads a memory location and then writes into a memory location must perform two bus operations `BusRd` and `BusRdEx`, even if no other processor is involved. The `BusRd` provides the memory block in S state, the `BusRdEx` causes a state transition from S to M. This drawback can be eliminated by adding a new state E (exclusive):

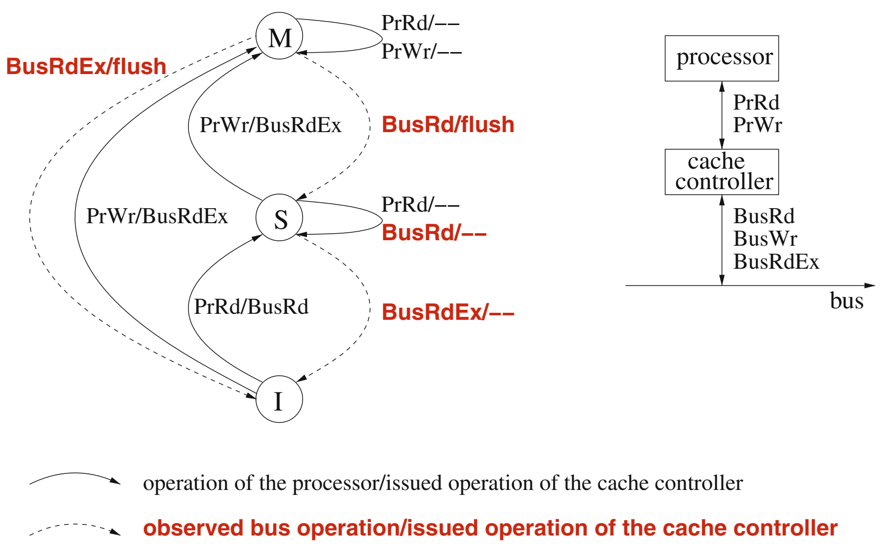


Fig. 2.33 Illustration of the MSI protocol: Each cache block can be in one of the states M (modified), S (shared), or I (invalid). State transitions are shown by arcs that are annotated with operations. A state transition can be caused by
(a) Operations of the processor (`PrRd`, `PrWr`) (solid arcs); the bus operations initiated by the cache controller are annotated behind the slash sign. If no bus operation is shown, the cache controller only accesses the local cache.
(b) Operations on the bus observed by the cache controller and issued by the cache controller of other processors (dashed arcs). Again, the corresponding operations of the local cache controller are shown behind the slash sign. The operation *flush* means that the cache controller puts the value of the requested memory block on the bus, thus making it available to other processors. If no arc is shown for a specific bus operation observed for a specific state, no state transition occurs and the cache controller does not need to perform an operation.

E (exclusive) means that the cache contains the only (exclusive) copy of the memory block and that this copy has not been modified. The main memory contains a valid copy of the block, but no other processor is caching this block.

If a processor requests a memory block by issuing a `PrRd` and if no other processor has a copy of this memory block in its local cache, then the block is marked with E (instead of S in the MSI protocol) in the local cache after being loaded from main memory with a `BusRd` operation. If at a later time, this processor performs a write into this memory block, a state transition from E to M is performed before the write. In this case, no additional bus operation is necessary. If between the local read and write operation, another processor performs a read to the same memory block, the local state is changed from E to S. The local write would then cause the same actions as in the MSI protocol. The resulting protocol is called **MESI protocol** according to the abbreviation of the four states. A more detailed discussion and a detailed description of several variants can be found in [41]. Variants of the MESI protocol are supported by many processors and the protocols play an important role for multicore processors to ensure the coherence of the local caches of the cores.

The MSI and MESI protocols are invalidation protocols. An alternative are **write-back update protocols** for write-back caches. In these protocols, after an update of a cache block with state M, all other caches which also contain a copy of the corresponding memory block are also updated. Therefore, the local caches always contain the most recent values of the cache blocks. In practice, these protocols are rarely used because they cause more traffic on the bus.

2.7.3.3 Directory-based cache coherence protocols

Snooping protocols rely on the existence of a shared broadcast medium like a bus or a switch through which all memory accesses are transferred. This is typically the case for multicore processors or small SMP systems. But for larger systems, such a shared medium often does not exist and other mechanisms have to be used.

A simple solution would be not to support cache coherence at hardware level. Using this approach, the local caches would only store memory blocks of the local main memory. There would be no hardware support to store memory blocks from the memory of other processors in the local cache. Instead, software support could be provided, but this requires more support from the programmer and is typically not as fast as a hardware solution.

An alternative to snooping protocols are **directory-based protocols**. These do not rely on a shared broadcast medium. Instead, a central directory is used to store the state of every memory block that may be held in cache. Instead of observing a shared broadcast medium, a cache controller can get the state of a memory block by a lookup in the directory. The directory can be held shared, but it could also be distributed among different processors to avoid bottlenecks when the directory is accessed by many processors. In the following, we give a short overview of directory-based protocols. For a more detailed description, we refer again to [41, 94].

As example, we consider a parallel machine with a distributed memory. We assume that for each local memory a directory is maintained that specifies for each memory block of the local memory which caches of other processors currently store a copy of this memory block. For a parallel machine with p processors, the directory can be implemented by maintaining a bit vector with *p presence bits* and a number of state bits for each memory block. Each presence bit indicates whether a specific processor has a valid copy of this memory block in its local cache (value 1) or not (value 0). An additional *dirty bit* is used to indicate whether the local memory contains a valid copy of the memory block (value 0) or not (value 1). Each directory is maintained by a *directory controller* which updates the directory entries according to the requests observed on the network.

Figure 2.34 illustrates the organization. In the local caches, the memory blocks are marked with M (modified), S (shared), or I (invalid), depending on their state, similar to the snooping protocols described above. The processors access the memory

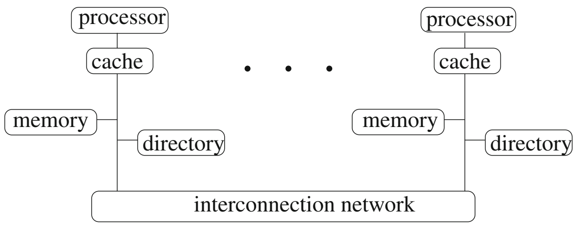


Fig. 2.34 Directory-based cache coherency.

system via their local cache controllers. We assume a global address space, i.e., each memory block has a memory address which is unique in the entire parallel system.

When a read miss or write miss occurs at a processor i, the associated cache controller contacts the local directory controller to obtain information about the accessed memory block. If this memory block belongs to the local memory and the local memory contains a valid copy (dirty bit 0), the memory block can be loaded into the cache with a local memory access. Otherwise, a nonlocal (remote) access must be performed. A request is sent via the network to the directory controller at the processor owning the memory block (home node). For a read miss, the receiving directory controller reacts as follows:

- If the dirty bit of the requested memory block is 0, the directory controller retrieves the memory block from local memory and sends it to the requesting node via the network. The presence bit of the receiving processor i is set to 1 to indicate that i has a valid copy of the memory block.
- If the dirty bit of the requested memory block is 1, there is exactly one processor j which has a valid copy of the memory block; the presence bit of this processor is 1. The directory controller sends a corresponding request to this processor j. The cache controller of j sets the local state of the memory block from M to S and sends the memory block both to the home node of the memory block and the processor i from which the original request came. The directory controller of the home node stores the current value in the local memory, sets the dirty bit of the memory block to 0, and sets the presence bit of processor i to 1. The presence bit of j remains 1.

For a write miss, the receiving directory controller does the following:

- If the dirty bit of the requested memory block is 0, the local memory of the home node contains a valid copy. The directory controller sends an invalidation request to all processors j for which the presence bit is 1. The cache controllers of these processors set the state of the memory block to I. The directory controller waits for an acknowledgment from these cache controllers, sets the presence bit for these processors to 0, and sends the memory block to the requesting processor i. The presence bit of i is set to 1, the dirty bit is also set to 1. After having received the memory block, the cache controller of i stores the block in its cache and sets its state to M.

- If the dirty bit of the requested memory block is 1, the memory block is requested from the processor j whose presence bit is 1. Upon arrival, the memory block is forwarded to processor i, the presence bit of i is set to 1, and the presence bit of j is set to 0. The dirty bit remains at 1. The cache controller of j sets the state of the memory block to I.

When a memory block with state M should be replaced by another memory block in the cache of processor i, it must be written back into its home memory, since this is the only valid copy of this memory block. To do so, the cache controller of i sends the memory block to the directory controller of the home node. This one writes the memory block back to the local memory and sets the dirty bit of the block and the presence bit of processor i to 0.

A cache block with state S can be replaced in a local cache without sending a notification to the responsible directory controller. Sending a notification avoids the responsible directory controller sending an unnecessary invalidation message to the replacing processor in case of a write miss as described above.

The directory protocol just described is kept quite simple. Directory protocols used in practice are typically more complex and contain additional optimizations to reduce the overhead as far as possible. Directory protocols are typically used for distributed memory machines as described. But they can also be used for shared memory machines. An example are the Sun T1 and T2 processors, see [94] for more details.

2.7.4 Memory consistency

Cache coherence ensures that each processor of a parallel system has the same consistent view of the memory through its local cache. Thus, at each point in time, each processor gets the same value for each variable if it performs a read access. But cache coherence does not specify in which order write accesses become visible to the other processors. This issue is addressed by memory consistency models. These models provide a formal specification of how the memory system will appear to the programmer. The consistency model sets some restrictions on the values that can be returned by a read operation in a shared address space. Intuitively, a read operation should always return the value that has been written last. In uniprocessors, the program order uniquely defines which value this is. In multiprocessors, different processors execute their programs concurrently and the memory accesses may take place in different order depending on the relative progress of the processors.

The following example illustrates the variety of different results of a parallel program if different execution orders of the program statements by the different processors are considered, see also [105].

Example We consider three processors P_1, P_2, P_3 which execute a parallel program with shared variables x_1, x_2, x_3. The three variables x_1, x_2, x_3 are assumed to be initialized with 0. The processors execute the following programs:

processor	P_1	P_2	P_3
program	(1) $x_1 = 1$;	(3) $x_2 = 1$;	(5) $x_3 = 1$;
	(2) print x_2, x_3;	(4) print x_1, x_3;	(6) print x_1, x_2;

Processor P_i sets the value of x_i, $i = 1, 2, 3$ to 1 and prints the values of the other variables x_j for $j \neq i$. In total, six values are printed which may be 0 or 1. Since there are no dependencies between the two statements executed by P_1, P_2, P_3, their order can be arbitrarily reversed. If we allow such a reordering and if the statements of the different processors can be mixed arbitrarily, there are in total $2^6 = 64$ possible output combinations consisting of 0 and 1. Different global orders may lead to the same output. If the processors are restricted to execute their statements in program order (e.g., P_1 must execute (1) before (2)), then output 000000 is *not* possible, since at least one of the variables x_1, x_2, x_3 must be set to 1 before a print operation occurs. A possible sequentialization of the statements is (1), (2), (3), (4), (5), (6). The corresponding output is 001011. □

To clearly describe the behavior of the memory system in multiprocessor environments, the concept of consistency models has been introduced. Using a consistency model, there is a clear definition of the allowable behavior of the memory system which can be used by the programmer for the design of parallel programs. The situation can be described as follows [185]: The input to the memory system is a set of memory accesses (read or write) which are partially ordered by the program order of the executing processors. The output of the memory system is a collection of values returned by the read accesses executed. A consistency model can be seen as a function that maps each input to a set of allowable outputs. The memory system using a specific consistency model guarantees that for any input, only outputs from the set of allowable outputs are produced. The programmer must write parallel programs, such that they work correctly for any output allowed by the consistency model. The use of a consistency model also has the advantage that it abstracts from the specific physical implementation of a memory system and provides a clear abstract interface for the programmer.

In the following, we give a short overview of popular consistency models. For a more detailed description, we refer to [3, 41, 94, 126, 185].

Memory consistency models can be classified according to the following two criteria:

- Are the memory access operations of each processor executed in program order?
- Do all processors observe the memory access operations performed in the same order?

Depending on the answer to these questions, different consistency models can be identified.

2.7.4.1 Sequential consistency

A popular model for memory consistency is the *sequential consistency model* (SC model) [126]. This model is an intuitive extension of the uniprocessor model and places strong restrictions on the execution order of the memory accesses. A memory system is sequentially consistent, if the memory accesses of each single processor are performed in the program order described by that processor's program and if the global result of all memory accesses of all processors appears to all processors in the *same* sequential order which results from an arbitrary interleaving of the memory accesses of the different processors. Memory accesses must be performed as *atomic* operations, i.e., the effect of each memory operation must become globally visible to all processors before the next memory operation of any processor is started.

The notion of program order leaves some room for interpretation. Program order could be the order of the statements performing memory accesses in the *source program*, but it could also be the order of the memory access operations in a machine program generated by an optimizing compiler which could perform statement reordering to obtain a better performance. In the following, we assume that the order in the source program is used.

Using sequential consistency, the memory operations are treated as atomic operations that are executed in the order given by the source program of each processor and that are centrally sequentialized. This leads to a *total order* of the memory operations of a parallel program which is the same for all processors of the system. In the example given above, not only output 001011, but also 111111 conforms to the SC model. The output 011001 is not possible for sequential consistency.

The requirement of a total order of the memory operations is a stronger restriction as it has been used for the coherence of a memory system in the last section (page 86). For a memory system to be coherent it is required that the write operations to the *same* memory location are sequentialized such that they appear to all processors in the same order. But there is no restriction on the order of write operations to different memory locations. On the other hand, sequential consistency requires that all write operations (to arbitrary memory locations) appear to all processors in the same order.

The following example illustrates that the atomicity of the write operations is important for the definition of sequential consistency and that the requirement of a sequentialization of the write operations alone is not sufficient.

Example Three processors P_1, P_2, P_3 execute the following statements:

processor	P_1	P_2	P_3
program	$(1) x_1 = 1;$	(2)while$(x_1 == 0);$	(4)while$(x_2 == 0);$
		$(3) x_2 = 1;$	(5)print$(x_1);$

The variables x_1 and x_2 are initialized with 0. Processor P_2 waits until x_1 has value 1 and then sets x_2 to 1. Processor P_3 waits until x_2 has value 1 and then prints the value of x_1. Assuming atomicity of write operations, the statements are executed in the order (1), (2), (3), (4), (5), and processor P_3 prints the value 1 for x_1, since write operation (1) of P_1 must become visible to P_3 before P_2 executes write operation (3). Using a sequentialization of the write operations of a variable without requiring atomicity and global sequentialization as it is required for sequential consistency would allow the execution of statement (3) before the effect of (1) becomes visible for P_3. Thus, (5) could print the value 0 for x_1.

To further illustrate this behavior, we consider a directory-based protocol and assume that the processors are connected via a network. In particular, we consider a directory-based invalidation protocol to keep the caches of the processors coherent. We assume that the variables x_1 and x_2 have been initialized with 0 and that they are both stored in the local caches of P_2 and P_3. The cache blocks are marked as shared (S).

The operations of each processor are executed in program order and a memory operation is started not before the preceding operations of the same processor have been completed. Since no assumptions on the transfer of the invalidation messages in the network are made, the following execution order is possible:

(1) P_1 executes the write operation (1) to x_1. Since x_1 is not stored in the cache of P_1, a write miss occurs. The directory entry of x_1 is accessed and invalidation messages are sent to P_2 and P_3.
(2) P_2 executes the read operation (2) to x_1. We assume that the invalidation message of P_1 has already reached P_2 and that the memory block of x_1 has been marked invalid (I) in the cache of P_2. Thus, a read miss occurs, and P_2 obtains the current value 1 of x_1 over the network from P_1. The copy of x_1 in the main memory is also updated.
After having received the current value of x_1, P_1 leaves the while loop and executes the write operation (3) to x_2. Because the corresponding cache block is marked as shared (S) in the cache of P_2, a write miss occurs. The directory entry of x_2 is accessed and invalidation messages are sent to P_1 and P_3.
(3) P_3 executes the read operation (4) to x_2. We assume that the invalidation message of P_2 has already reached P_3. Thus, P_3 obtains the current value 1 of x_2 over the network. After that, P_3 leaves the while loop and executes the print operation (5). Assuming that the invalidation message of P_1 for x_1 has not yet reached P_3, P_3 accesses the old value 0 for x_1 from its local cache, since the corresponding cache block is still marked with S. This behavior is possible if the invalidation messages may have different transfer times over the network.

In this example, sequential consistency is violated, since the processors observe different orders of the write operation: Processor P_2 observes the order $x_1 = 1, x_2 = 1$, whereas P_3 observes the order $x_2 = 1, x_1 = 1$ (since P_3 gets the *new* value of x_2, but the *old* value of x_1 for its read accesses). □

In a parallel system, sequential consistency can be guaranteed by the following *sufficient conditions* [41, 51, 176]:

(1) Every processor issues its memory operations in program order. In particular, the compiler is not allowed to change the order of memory operations, and no out-of-order executions of memory operations are allowed.

(2) After a processor has issued a write operation, it waits until the write operation has been completed before it issues the next operation. This includes that for a write miss all cache blocks which contain the memory location written must be marked invalid (I) before the next memory operation starts.
(3) After a processor has issued a read operation, it waits until this read operation and the write operation whose value is returned by the read operation has been entirely completed. This includes that the value returned to the issuing processor becomes visible to all other processors before the issuing processor submits the next memory operation.

These conditions do not contain specific requirements concerning the interconnection network, the memory organization or the cooperation of the processors in the parallel system. In the example from above, condition (3) ensures that after reading x_1, P_2 waits until the write operation (1) has been completed before it issues the next memory operation (3). Thus, P_3 always reads the new value of x_1 when it reaches statement (5). Therefore, sequential consistency is ensured.

For the programmer, sequential consistency provides an easy and intuitive model. But the model has a performance disadvantage, since all memory accesses must be atomic and since memory accesses must be performed one after another. Therefore, processors may have to wait for quite a long time before memory accesses that they have issued have been completed. To improve performance, consistency models with fewer restrictions have been proposed. We give a short overview in the following and refer to [41, 94] for a more detailed description. The goal of the less restricted models is to still provide a simple and intuitive model but to enable a more efficient implementation.

2.7.4.2 Relaxed consistency models

Sequential consistency requires that the read and write operations issued by a processor maintain the following orderings where $X \rightarrow Y$ means that the operation X must be completed before operation Y is executed:

- $R \rightarrow R$: The read accesses are performed in program order.
- $R \rightarrow W$: A read operation followed by a write operation is executed in program order. If both operations access the same memory location, an *anti-dependence* occurs. In this case, the given order must be preserved to ensure that the read operation accesses the correct value.
- $W \rightarrow W$: The write accesses are performed in program order. If both operations access the same memory location, an *output dependence* occurs. In this case, the given order must be preserved to ensure that the correct value is written last.
- $W \rightarrow R$: A write operation followed by a read operation is executed in program order. If both operations access the same memory location, a *flow dependence* (also called *true dependence*) occurs.

If there is a dependence between the read and write operation, the given order must be preserved to ensure the correctness of the program. If there is no such dependence, the given order must be kept to ensure sequential consistency. *Relaxed consistency models* abandon one or several of the orderings required for sequential consistency, if the data dependencies allow this.

Processor consistency models relax the $W \rightarrow R$ ordering to be able to partially hide the latency of write operations. Using this relaxation, a processor can execute a read operation even if a preceding write operation has not yet been completed, if there are no dependencies. Thus, a read operation can be performed even if the effect of a preceding write operation is not visible yet to all processors. Processor consistency models include *total store ordering* (**TSO model**) and *processor consistency* (**PC model**). In contrast to the TSO model, the PC model does not guarantee atomicity of the write operations. The differences between sequential consistency and the TSO or PC model are illustrated in the following example.

Example Two processors P_1 and P_2 execute the following statements:

processor	P_1	P_2
program	(1)$x_1 = 1$;	(3)$x_2 = 1$;
	(2)print(x_2);	(4)print(x_1);

Both variables x_1 and x_2 are initialized with 0. Using sequential consistency, statement (1) must be executed before statement (2), and statement (3) must be executed before statement (4). Thus, it is not possible that the value 0 is printed for both x_1 and x_2. But using TSO or PC, this output is possible, since, for example, the write operation (3) does not need to be completed before P_2 reads the value of x_1 in (4). Thus, both P_1 and P_2 may print the old value for x_1 and x_2, respectively. □

Partial store ordering (PSO) models relax both the $W \rightarrow W$ and the $W \rightarrow R$ ordering required for sequential consistency. Thus in PSO models, write operations can be completed in a different order as given in the program if there is no output dependence between the write operations. Successive write operations can be overlapped which may lead to a faster execution, in particular when write misses occur. The following example illustrates the differences between the different models.

Example We assume that the variables x_1 and *flag* are initialized with 0. Two processors P_1 and P_2 execute the following statements:

processor	P_1	P_2
program	(1)$x_1 = 1$;	(3)while(flag == 0);
	(2)flag = 1;	(4)print(x_1);

Using sequential consistency, PC or TSO, it is *not* possible that the value 0 is printed for x_1. But using the PSO model, the write operation (2) can be completed *before* $x_1 = 1$. Thus, it is possible that the value 0 is printed for x_1 in statement (4). This output does not conform with intuitive understanding of the program behavior in the example, making this model less attractive for the programmer. □

Weak ordering models additionally relax the $R \rightarrow R$ and $R \rightarrow W$ orderings. Thus, no completion order of the memory operations is guaranteed. To support programming, these models provide additional synchronization operations to ensure the following properties:

- All read and write operations which lie in the program *before* the synchronization operation are completed before the synchronization operation.
- The synchronization operation is completed before read or write operations are started which lie in the program *after* the synchronization operation.

The advent of multicore processors has lead to an increased availability of parallel systems and most processors provide hardware support for a memory consistency model. Often, relaxed consistency models are supported, as it is the case for the PowerPC architecture of IBM or the different Intel architectures. But different hardware manufacturers favor different models, and there is no standardization as yet.

2.8 Example: IBM Blue Gene supercomputer

The IBM Blue Gene supercomputers are popular examples for massively parallel systems. In this section, we give a short overview of the architecture of these systems. An important design goal for the Blue Gene systems was to provide a good power efficiency, i.e., to obtain a high performance for each unit of power, measured as the number of Floating-Point Operations per second (FLOPS) that can be computed per Watt. The Blue Gene (BG) supercomputers were introduced in 2004 as Blue Gene/L (BG/L) systems and were originally intended for life science computations, such as molecular dynamics or protein folding, but nowadays they are used for a large range of applications from science and technology. In 2007, the BG/P systems were introduced as successor of the BG/L systems, see [191] for a detailed description. In 2012, the BG/Q systems have been introduced.

The BG systems have a low power consumption especially because the processors used have a small clock frequency, leading to low leakage currents and a reduced heat dissipation, see Sect. 2.4.3 for a short discussion of the interaction between clock frequency and power consumption. The BG systems use processors that have been especially designed for these systems. The BG/L processors had a clock frequency of 700 MHz, the BG/P processors used a clock frequency of 850 MHz; the clock frequency of the current BG/Q processors is 1.6 GHz. The trend toward multicore processors can also be observed for the BG processors: the BG/L and BG/P processors were dual-core and quad-core processors, the BG/Q processors contain 16 cores. In the following, we give an overview of the BG/Q systems; a more detailed description can be found in [90, 29]. A comparison of important characteristics of the BG/L, BG/P, and BG/Q systems is given in Table 2.4, see also [191, 190].

Similar to the BG/L and BG/P processors, the BG/Q processors are based on a system-on-a-chip design. The processor chip contains not only the processor cores, but also control hardware for the memory subsystem and the networking subsystem.

Table 2.4 Important characteristics of the IBM BG/L, BG/P, and BG/Q systems concerning processor, network, and energy consumption, see [190]. The L2 cache is shared by all cores of a processor chip. The size of the main memory is the size per node. The latency between neighboring nodes is for 32-bytes packets.

characteristic	BG/L	BG/P	BG/Q
processor	32-Bit PPC 440	32-Bit PPC 450	64-Bit PPC A2
clock frequency	0,7 GHz	0,85 GHz	1,6 GHz
fabrication technology	130 nm	90 nm	45 nm
number of cores	2	4	16
node performance	5,6 GFLOPS	13,6 GFLOPS	204,8 GFLOPS
L1-Cache per core	32 KB	32 KB	16/16 KB
L2-Cache	4 MB	8 MB	32 MB
main memory	0,5/1 GB	2/4 GB	16 GB
network topology	3D-Torus	3D-Torus	5D-Torus
network bandwidth	2,1 GB/s	5,1 GB/s	32 GB/s
neighbor latency	200 ns	100 ns	80 ns
maximum latency	6,4 μs	5,5 μs	3 μs
performance rack	5,7 TF	13,3 TF	209 TF
power consumption rack	20 KW	32 KW	100 KW
energy efficiency	0,23 GF/W	0,37 GF/W	2,1 GF/W

Figure 2.35 shows a block diagram of the BG/Q processor organization. The processor chip contains 1.49 billion transistors. And 16 cores are integrated on the processor chip, each containing an augmented version of the PowerPC A2 processor core on the basis of the IBM PowerEN chip [109] and a SIMD-based quad-FPU (floating-point unit), which has been especially designed for the BG/Q computers. Each A2 processor core implements the 64-bit Power Instruction Set Architecture (Power ISA) and can execute four threads in parallel via hyperthreading. The quad-FPU extends the scalar floating-point operations of the Power ISA by specific SIMD floating-point instructions (called Quad Processing eXtensions, QPX). These instructions work on 32 256-bit registers, containing four 64-bit floating-point values, and in each cycle a floating-point operation can be applied to each of these four floating-point values, see Sect. 3.4 for a more detailed treatment of SIMD instructions. The QPX instructions provide the standard arithmetic operations and also a Fused Multiply-Add (FMA) operation; using these operations up to eight floating-point operations can be executed in each cycle. For a clock frequency of 1.6 GHz, this results in a maximum performance of 12.8 GFLOPS for each processor core. For the entire processor chip with its 16 cores, a maximum performance of 204.8 GFLOPS results.

In addition to the 16 processor cores, the processor chip contains a specialized service core for tasks of the operating system, such as interrupt and I/O handling. Moreover, there is an 18th core, which can be activated if a defect in one of the processing cores is detected during manufacturing testing, making it necessary to de-activate the defect core and replace it by the redundant core.

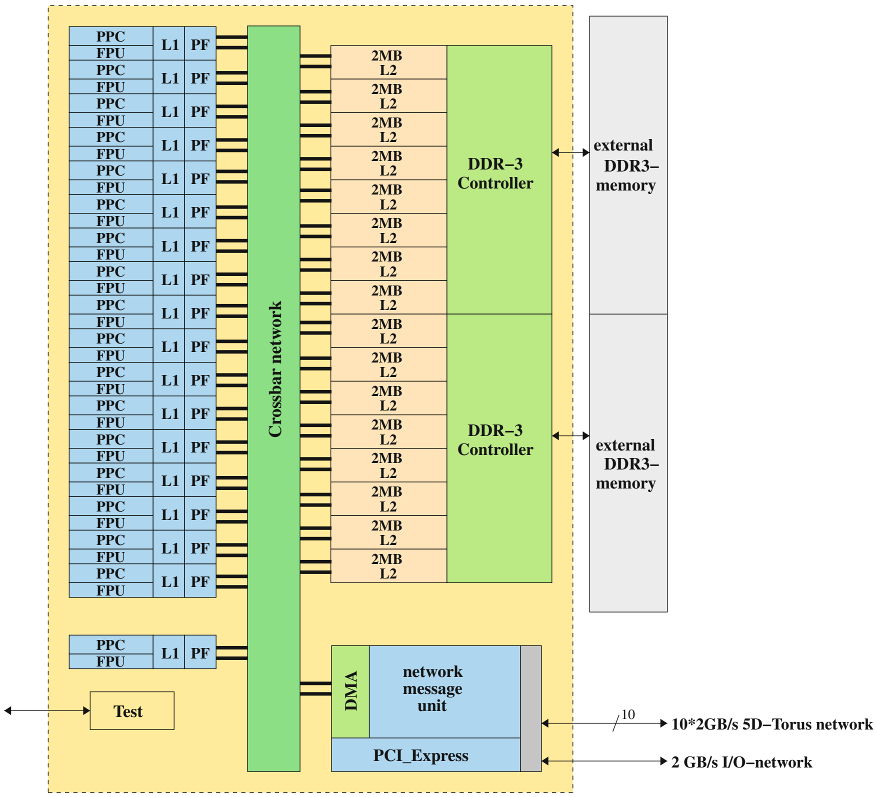


Fig. 2.35 Block diagram of a Blue Gene/Q processor chip.

The BG/Q processors contain a memory hierarchy. Each core has a private 16 KB L1 instruction cache (four-way associative) as well as L1 data cache (eight-way associative with cache lines of size 64 bytes) of the same size. Both are supported by an L1 prefetch unit (denotes as PF in Fig. 2.35 for anticipated loading of data to hide the latency of accesses to the L2 cache or the main memory, see the detailed description in [90]. Moreover, there is an L2 cache that is shared by all cores of the processor chip. To provide sufficient bandwidth, the L2 cache is partitioned into 16 sections of size 2 MB. Each section is 16-way associative and uses a write-back replacement policy. At the level of the L2 cache, the BG/Q processor also supports speculative execution of instructions, which can, e.g., be used for an efficient implementation of lock operations based on atomic operations or hardware-supported transactional memory. The processor chip also contains the control logic for the 5D torus interconnection network of the BG/Q system.

The internal interconnect of a BG/Q processor chip is supplied by a crossbar network, connecting the cores, the L2 cache, and the network controller with each other. The crossbar network and the L2 cache are clocked at 800 MHz, half the frequency of the cores. The maximum on-chip bisection bandwidth of the crossbar network is 563 GB/s. For the management of L2 cache misses, two memory controllers are integrated on the processor chip. These are the DDR-3 controllers in Fig. 2.35. Each of these controllers is responsible for eight L2 cache sections.

BG/Q systems contain compute nodes and I/O nodes where the I/O nodes establish the connection with the file system and the compute nodes perform the actual computations. The compute nodes are connected by a 5D torus network. Correspondingly, each compute node has 10 bidirectional links where each link can simultaneously send and receive two GB/s. The control logic for the network is integrated onto the processor chip (*message unit*) and provides a send and a receive unit with FIFO send and receive buffers for each of 10 torus links. The message unit is the interface between the network and the BG/Q memory system and is connected to the on-chip crossbar network. For a subset of the compute nodes (called *bridge nodes*), an additional I/O link connection is activated to provide a connection to the I/O nodes. This additional link also has a bidirectional bandwidth of 2 GB. A detailed description of the network of the BG/Q systems can be found in [29, 90].

The BG/Q interconnection network also shows the technological progress of the BG systems: For the BG/L and BG/P systems, a 3D torus network has been used, see Table 2.4. The transition to a 5D torus network for the BG/Q systems leads to a larger bisection bandwidth as well as a reduction of the number of hops required to transmit data between nodes that are not directly connected with each other, thus reducing the resulting transmission latency. The BG/L and BG/P systems included a barrier and a broadcast network in addition to the 3D torus network. For the BG/Q systems, this functionality is now integrated into the 5D torus network.

A BG/Q system is built up in several stages, see [104] for a detailed overview. A single BG/Q processor and 72 SDRAM DDR3 memory chips are integrated into a compute card, thus providing 16 GB main memory for each compute card. Almost, 32 compute cards are combined as a node board, which is internally organized as a $2 \times 2 \times 2 \times 2 \times 2$ torus. And 16 of these node boards are inserted into a midplane, resulting in a $4 \times 4 \times 4 \times 4 \times 2$ torus. A BG/Q rack contains one or two midplanes, resulting in a $4 \times 4 \times 4 \times 8 \times 2$ torus when using two midplanes. A maximum number of 5,112 racks can be used for a BG/Q system, corresponding to 524 288 BG/Q processors, each providing 16 cores. This leads to a peak performance of 100 PetaFLOPS.

2.9 Exercises for Chapter 2

Exercices 2.1. Consider a two-dimensional mesh network with n rows and m columns. What is the bisection bandwidth of this network?

Exercices 2.2. Consider a shuffle-exchange network with $n = 2^k$ nodes, $k > 1$. How many of the $3 \cdot 2^{k-1}$ edges are shuffle edges and how many are exchange edges? Draw a shuffle-exchange network for $k = 4$.

Exercices 2.3. In Sect. 2.5.2, page 43, we have shown that there exist k independent paths between any two nodes of a k-dimensional hypercube network. For $k = 5$, determine all paths between the following pairs of nodes: (i) nodes 01001 and 00011; (ii) nodes 00001 and 10000.

Exercices 2.4. Write a (sequential) program that determines all paths between any two nodes for hypercube networks of arbitrary dimension.

Exercices 2.5. The RGC sequences RGC_k can be used to compute embeddings of different networks into a hypercube network of dimension k. Determine RGC_3, RGC_4, and RGC_5. Determine an embedding of a Three-dimensional mesh with $4 \times 2 \times 4$ nodes into a five-dimensional hypercube network.

Exercices 2.6. Show how a complete binary tree with n leaves can be embedded into a butterfly network of dimension $\log n$. The leaves of the trees correspond to the butterfly nodes at level $\log n$.

Exercices 2.7. Construct an embedding of an three-dimensional torus network with $8 \times 8 \times 8$ nodes into a nine-dimensional hypercube network according to the construction in Sect. 2.5.3, page 48.

Exercices 2.8. A k-dimensional Beneš network consists of two connected k-dimensional butterfly networks, leading to $2k+1$ stages, see page 52. A Beneš network is *nonblocking*, i.e., any permutation between input and output nodes can be realized without blocking. Consider an 8×8 Benes network and determine the switch positions for the following two permutations:

$$\pi_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 4 & 3 & 5 & 7 & 6 \end{pmatrix} \quad \pi_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 7 & 4 & 6 & 0 & 5 & 3 & 1 \end{pmatrix}$$

Exercices 2.9. The cross-product $G_3 = (V_3, E_3) = G_1 \otimes G_2$ of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be defined as follows:
$V_3 = V_1 \times V_2$ and $E_3 = \{((u_1, u_2), (v_1, v_2)) \mid ((u_1 = v_1) \text{ and } (u_2, v_2) \in E_2)$. or $((u_2 = v_2) \text{ and } (u_1, v_1) \in E_1)\}$. The symbol $\bigotimes$ can be used as abbreviation with the following meaning

$$\bigotimes_{i=a}^{b} G_i = ((\cdots(G_a \otimes G_{a+1}) \otimes \cdots) \otimes G_b)$$

Draw the following graphs and determine their network characteristics (degree, node connectivity, edge connectivity, bisection bandwidth, and diameter)

(a) linear array of size 4 $\otimes$ linear array of size 2
(b) two-dimensional mesh with 2×4 nodes $\otimes$ linear array of size 3
(c) linear array of size 3 $\otimes$ complete graph with 4 nodes
(d) $\bigotimes_{i=2}^{4}$ linear array of size i

(e) $\bigotimes_{i=1}^{k}$ linear array of size 23. Draw the graph for $k = 4$, but determine the characteristics for general values of k.

Exercices 2.10. Consider a three-dimensional hypercube network and prove that E-cube routing is deadlock-free for this network, see Sect. 2.6.1, page 57.

Exercices 2.11. In the directory-based cache coherence protocol described in Sect. 2.7.3, page 92, in case of a read miss with dirty bit 1, the processor which has the requested cache block sends it to both the directory controller and the requesting processor. Instead, the owning processor could send the cache block to the directory controller and this one could forward the cache block to the requesting processor. Specify the details of this protocol.

Exercices 2.12. Consider the following sequence of memory accesses

$$2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, 11$$

Consider a cache of size 16 Bytes. For the following configurations of the cache determine for each of the memory accesses in the sequence whether it leads to a cache hit or a cache miss. Show the resulting cache state that results after each access with the memory locations currently held in cache. Determine the resulting miss rate.

a) direct-mapped cache with block-size 1
b) direct-mapped cache with block-size 4
c) two-way set-associative cache with block-size 1, LRU replacement strategy
d) two-way set-associative cache with block-size 4, LRU replacement strategy
e) fully associative cache with block-size 1, LRU replacement
f) fully associative cache with block-size 4, LRU replacement

Exercices 2.13. Consider the MSI protocol from Fig. 2.33, page 90, for a bus-based system with three processors P_1, P_2, P_3. Each processor has a direct-mapped cache. The following sequence of memory operations access two memory locations A and B which are mapped to the same cache line:

processor	action
P_1	write $A, 4$
P_3	write $B, 8$
P_2	read A
P_3	read A
P_3	write A, B
P_2	read A
P_1	read B
P_1	write $B, 10$

We assume that the variables are initialized with $A = 3$ and $B = 3$ and that the caches are initially empty. For each memory access determine

- the cache state of each processor after the memory operations,
- the content of the cache and the memory location for A and B,
- the processor actions (`PrWr`, `PrRd`) caused by the access, and
- the bus operations (`BusRd`, `BusRdEx`, `flush`) caused by the MSI protocol.

Exercices 2.14. Consider the following memory accesses of three processors P_1, P_2, P_3:

P_1	P_2	P_3
(1) $A = 1;$	(1) $B = A;$	(1) $D = C;$
(2) $C = 1;$		

The variables A, B, C, D are initialized with 0. Using the sequential consistency model, which values can the variables B and D have?

Exercices 2.15. Visit the Top500 webpage at `http://www.top500.org` and determine important characteristics of the five fastest parallel computers, including number of processors or core, interconnection network, processors used, and memory hierarchy.

Exercices 2.16. Consider the following two realizations of a matrix traversal and computation:

```
for (j=0; j<1500; j++)        for (i=0; i<1500; i++)
  for (i=0; i<1500; i++)        for (j=0; j<1500; j++)
    x[i][j] = 2 · x[i][j];        x[i][j] = 2 · x[i][j];
```

We assume a cache of size 8 Kbytes with a large enough associativity so that no conflict misses occur. The cache line size is 32 bytes. Each entry of the matrix `x` occupies 8 bytes. The implementations of the loops are given in C which uses a row-major storage order for matrices. Compute the number of cache lines that must be loaded for each of the two loop nests. Which of the two loop nests leads to a better spatial locality?