

Parallel Programming

Prof. Dr. G. Rünger

Chemnitz University of Technology
Department of Computer Science
Chair for Practical Computer Science



CHEMNITZ UNIVERSITY OF
TECHNOLOGY

Summer Term 2015

Overview

Overview

Introduction

Architecture of Parallel Systems

Parallel Programming Models

Message Passing Programming

Introduction to MPI-2

Realization and Execution Times Analysis of global Communication Operations

Organization of the Lecture

- ▶ Website:
http://www.tu-chemnitz.de/informatik/PI/lehre/plan/SS15/teach_pp.php.en
- ▶ Lecture dates:
 - ▶ Mon 17:15–18:45
- ▶ Tutorial dates:
 - ▶ Tue 17:15–18:45 (English)
 - ▶ Thu 17:15–18:45 (English)
 - ▶ Fri 9:15–10:45 (German)

Overview

Introduction

- Motivation

- Basic concepts

- Evaluation of Parallel Programs

Architecture of Parallel Systems

- Processor Architecture and Technology Trends

- Flynn's Taxonomy of Parallel Architectures

- Memory Organization of Parallel Computers

- Interconnection Networks

- Routing and Switching

Parallel Programming Models

- Models for Parallel Systems

- Parallelization of Programs

- Levels of Parallelism

- Information Exchange

Message Passing Programming

- MPI – Message Passing Interface – Introduction

- MPI Point-to-Point Communication

- Deadlocks with Blocking Communication Operations

- Non-blocking Point-to-point Communication
- Global Communication Operations in MPI
- Example: Matrix-Vector Multiplication
- Deadlocks with Global Communication Operations
- Process Groups and Communicators
- Process Topologies

Introduction to MPI-2

- Process creation and management
- One Sided Communication

Realization and Execution Times Analysis of global Communication Operations

- Modeling of Execution Times
- Realization of Global Communication Operations
- Asymptotic Execution Times

Parallel Scientific Computing

- ▶ Summer term
- ▶ Solving systems of linear equations
- ▶ Solving systems of nonlinear equations
- ▶ Solving differential equations
- ▶ Irregular algorithms

Multicore Programming

- ▶ Winter term
- ▶ Architecture of multicore systems
- ▶ Programming with OpenMP, Pthreads
- ▶ Introduction to general purpose GPU programming

Recommended Literature

- ▶ Rauber Th., Rünger G.,
Parallel Programming: for Multicore and Cluster Systems.
Springer 2010, ISBN: 978-3-642-04817-3
- ▶ D.E. Culler, J.P. Singh, and A. Gupta.
Parallel Computer Architecture: A Hardware Software Approach.
Morgan Kaufmann, 1999.
- ▶ K. Hwang.
Advanced Computer Architecture: Parallelism, Scalability, Programmability.
McGraw-Hill, 1993.
- ▶ D. Sima, T. Fountain, and P. Kacsuk.
Advanced Computer Architectures.
Addison-Wesley, 1997.

Overview

Overview

Introduction

Architecture of Parallel Systems

Parallel Programming Models

Message Passing Programming

Introduction to MPI-2

Realization and Execution Times Analysis of global Communication Operations

Overview

Introduction

Motivation

Basic concepts

Evaluation of Parallel Programs

Motivation for the use of parallel systems

- ▶ Parallel programming and the design of efficient parallel programs is well-established in **high performance, scientific computing** for many years.
- ▶ Innovations in hardware architecture, like **hyper-threading or multicore processors**, make parallel computing resources available for inexpensive desktop computers.
- ▶ In a few years, many standard software products will be based on concepts of parallel programming.
- ▶ In the future, there will be an enormous need for software developers with parallel programming skills.

Use of parallel systems in High-Performance Computing

- ▶ The computational performance requirements of applications are constantly increasing;
Example: compute-intensive applications and simulations from the areas of **natural and applied sciences** and **scientific computing**, such as:
 - ▶ Weather forecast, climatic research and pollution propagation
 - ▶ Vehicle and airplane simulations
 - ▶ Computer-aided design and analysis of medicinal drugs
 - ▶ Application of computer graphics in the movie and ad industries
- ▶ Calculation of **larger problems** within the same computing time
- ▶ Increase of **simulation precision** in the same computing time
- ▶ Fulfillment of **larger storage requirements**
- ▶ Increase of **system's reliability**

Prerequisites to parallel programming

Architecture

Parallel computers or interconnected PCs/workstations provide **multiple processors or cores** for the calculation of the subtasks.

Application algorithm

The application algorithm permits the **decomposition** of the computations into **independent subtasks** that may be performed in parallel.

Software

A parallel programming language or a parallel programming environment is available to **manage the parallel execution**.

Overview

Introduction

Motivation

Basic concepts

Evaluation of Parallel Programs

Parallelization of algorithms

Necessary steps to parallelize an algorithm:

1. **Decomposition** of the computations into **tasks** that can be executed in parallel
2. distribution of the computations and the corresponding data on the processors;

The tasks can have a different **granularity** depending on the platform (instruction-level parallelism, data parallelism, function-level parallelism).

Goal: balanced distribution of computations (**load distribution**);

Method: static or dynamic **scheduling**

3. **Data exchange** between the subtasks (distributed address space), i.e., communication/coordination planning;

Goal: minimization of communication costs;

4. Definition of **synchronization points**;

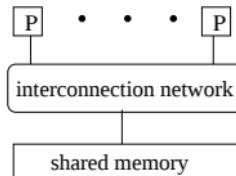
Goal: avoidance of inconsistent states (shared address space) and minimization of waiting times.

Parallel Computer Systems

Distinction from the view of **memory** organization:

1. Shared address space:

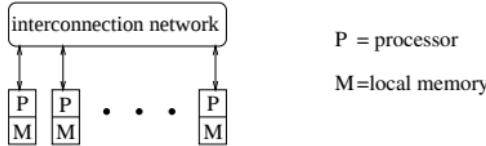
- ▶ Local and non-local memory accesses (implicit)



example: single multicore processors

2. Distributed address space:

- ▶ Message passing (explicit)



example: cluster systems

Parallel programming models

A parallel programming model describes the view of the programmer on a parallel computer system (= combination of software and hardware)

- ▶ Which **types/levels of parallelism** are used?
Instruction-level parallelism, data parallelism, function-level parallelism, mixed parallelism
- ▶ How is the parallelism **specified**?
 - ▶ explicit (e.g. message-passing)
 - ▶ implicit (e.g. functional languages)
- ▶ In which form is the parallelism being processed?
Fork-Join, SPMD or SIMD, Master-Slave, Pipelining
- ▶ How is **data exchange** implemented?
 - ▶ shared variables
 - ▶ explicit communication (point-to-point messages and collective communication operations)
- ▶ Does a **cost model** exist?

Software for parallel program development

- ▶ **Parallelizing compiler**
- ▶ **Parallel programming languages:** **Example:** HPF - High Performance Fortran, UPC, X10, Fortress, Chapel
- ▶ **Message-passing programming** (for a distributed address space):
Usage of communication libraries (with Fortran, C, C++, Java)
 - ▶ MPI - Message Passing Interface
 - ▶ PVM - Parallel Virtual Machine
- ▶ **Thread programming** (for a shared address space):
 - ▶ PThreads (library)
 - ▶ OpenMP (controlling the parallelism using pragmas)
 - ▶ Java (multithreaded programming language)

Distributed programming and parallel programming

Difference between **distributed** programming and **parallel** programming:

- ▶ **Distributed programming:** Programs on different autonomous systems (clients) **make inquiries** for the processing of **individual, usually separate functions** to remote systems (servers), which provide processing services. The server might distribute arriving inquiries on several processors.
The systems involved often use different operating systems.
- ▶ **Parallel programming:** Several computers/processors cooperate to process **one problem** divided into several subtasks.
The goal is to process the whole problem as fast as possible with the use of all available processors.
The computers involved often use the same operating system.

Overview

Introduction

Motivation

Basic concepts

Evaluation of Parallel Programs

Parallel runtime of a program

- ▶ The **parallel runtime** $T_p(n)$ of a parallel program P with problem size n running on p processors is defined as **the time between the beginning of the processing** of P and the **termination of the computation** of **all** processors involved.
- ▶ For processors with physically **distributed memory** $T_p(n)$ consists of:
 - ▶ Execution time of **local computations**;
 - ▶ Time for **data exchanges** by communication operations;
 - ▶ **Waiting times** of the processors, e.g., due to an unequal load distribution;
 - ▶ **Synchronization** times for all or a subset of the executing processors.
- ▶ For processors with a shared memory **global data access times** instead of data exchange times have to be considered.

Costs of a parallel program

- ▶ The **costs** $C_p(n)$ of a parallel program P with problem size n running on p processors is the **total accumulated** execution time of all processors involved in the computation:

$$C_p(n) = T_p(n) \cdot p$$

- ▶ The costs of a program is a **measure of the total number of calculations executed by all processors**.
- ▶ A parallel program is called *cost-optimal* if $C_p(n) = T^*(n)$, where $T^*(n)$ is the runtime of the fastest sequential program version;
A cost-optimal program executes as many computations as the fastest sequential program version.
Problem: the fastest sequential program version **may be unknown** or very difficult to determine.
- ▶ The costs are also called **work** or **processor-time product**.

Speedup of a parallel program

- ▶ The **speedup** $S_p(n)$ of a parallel program P with problem size n running on p processors is defined as:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- ▶ $T_p(n)$ - parallel runtime of parallel program P on p processors;
- ▶ $T^*(n)$ - runtime of the fastest sequential implementation of the same problem.
- ▶ Speedup is a **measure of the relative speed gain** in relation to the best sequential implementation.
- ▶ Typically, **linear speedup** can be attained at maximum: $S_p(n) \leq p$ (theoretical upper bound).
- ▶ In practice, **superlinear speedup** can occur due to cache effects.

Efficiency of a parallel program

- ▶ The **efficiency** $E_p(n)$ of a parallel program P with problem size n running on p processors is defined as:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)} = \frac{T^*(n)}{C_p(n)}$$

- ▶ $C_p(n)$ - costs of the parallel program;
- ▶ $T_p(n)$ - parallel runtime of the parallel program P ;
- ▶ $T^*(n)$ - runtime of the best sequential algorithm.
- ▶ Efficiency is a **measure of the proportion of runtime needed for calculations, which are also present in the sequential program.**
- ▶ The ideal speedup $S_p(n) = p$ corresponds to an efficiency of $E_p(n) = 1$.

Amdahl's Law

- ▶ If in a parallel implementation a constant fraction f with $0 \leq f \leq 1$ must be executed **sequentially**, then the runtime of the parallel implementation consists of:
 - ▶ Runtime $f \cdot T^*(n)$ of the **sequential** part and
 - ▶ Runtime of the parallel part, which is at least $(1 - f)/p \cdot T^*(n)$
- ▶ The achievable speedup is then:

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

- ▶ **Example:** $f = 0.2 \rightarrow S_p(n) \leq 5$, independent of the number of processors p ;
→ The sequential part has a huge influence on the achievable speedup.
For the effective employment of a **large number of processors** the **sequential section needs to be reduced** accordingly.

- ▶ The **scalability** of a parallel program on a given parallel computer is a **measure of the performance gain proportional to the number p of processors used**.
- ▶ Frequently observed characteristics:
 - ▶ Given a fixed problem size n and an increasing number of processors p , a **saturation of speedup** occurs;
 - ▶ Given a fixed number p of processors and an increasing problem size n , the **speedup grows** with n .
- ▶ Scalability denotes that the efficiency of a parallel program stays constant with the simultaneous rise of the number of processes p and of the problem size n .

Overview

Overview

Introduction

Architecture of Parallel Systems

Parallel Programming Models

Message Passing Programming

Introduction to MPI-2

Realization and Execution Times Analysis of global Communication Operations

Architecture of Parallel Systems

Processor Architecture and Technology Trends

Flynn's Taxonomy of Parallel Architectures

Memory Organization of Parallel Computers

Interconnection Networks

Routing and Switching

- ▶ 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 (empirical observation)

The number of transistors of a typical processor chip doubles every 18 to 24 months.

2007: 200-400 millions of transistors per processor chip;
2009: 400-800 millions of transistors per processor chip;
examples: Intel Core 2 Duo: 291 millions of transistors;
IBM Cell Prozessor ca. 250 millions of transistors;
Intel Core i7 quad: 731 billions of transistors;
2011: 8-core Xeon Nehalem-EX: 2.3 billions of transistors;
10-Core Xeon Westmere-EX: 2.6 billions of transistors;

Performance trends

- ▶ An important performance factor is the **clock frequency** of the processor, which determines the cycle time of the processor.
Between 1987 and 2003: average annual increase of about 40%
Since 2003: clock frequency remains nearly unchanged due to power consumption constraints
- ▶ 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.
- ▶ Use of benchmark programs to evaluate the performance of processors.
SPEC benchmarks (*System Performance and Evaluation Cooperative*):
www.spec.org
- ▶ Between 1986 and 2003: desktop processors show an average annual performance increase of about 50% (based on SPEC benchmarks)
Since 2003: average annual performance increase is about 22%
- ▶ The performance increase is caused by architectural improvements:
pipelining, multiple functional units, multiple cores

Performance Evolution of Processors

- ▶ Increase of **memory capacity** of DRAM memory chips by 40% to 60% per year since 1977.
- ▶ Decrease of **memory access latency** of DRAM memory chips in average by about 25% per year.
~~ Problem of memory wall

Reasons for the previous performance increase

- (1) Until 2003, the **clock rate** of micro processors increases in average about 30% per year
- (2) Architectural enhancements: Make use of the increasing number of transistors for **additional functionality** and **parallelism** on the instruction level (ILP - instruction level parallelism)

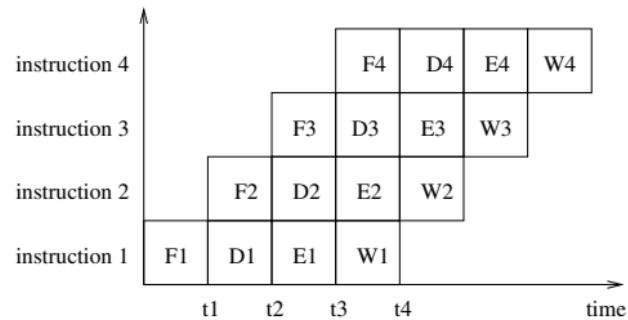
Architectural Improvements – pipelining

Additional functionality and parallelism on the instruction level

- ▶ Increase of the **internal word length** to 32 or 64 bits;
- ▶ Make use of **internal pipelining** by decomposing the instruction processing into pipeline stages
 - ILP (instruction level parallelism) processors
 - the different pipeline stages can work in parallel
 - ~~ Reduction of the **average execution time** per instruction;

simple decomposition:

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



Further Architectural Improvements

- ▶ Usage of multiple functional units, which get **dynamically instructions** from a dispatch unit
 - ~~ independent machine instructions can be executed in **parallel**
- ▶ parallelism at process or thread level → multicore processors
- ▶ Integration of (**multilevel**) **caches** onto the die of the processor
 - ~~ Reduction of the **average memory access latency**;
(reduces the Memory Wall Problem)
- ▶ VLIW (Very Long Instruction Word).

(2) ILP-Parallelism (Pipelining; several functional units):

- ▶ Can be increased from technological point of view;
- ▶ **Dependencies between instructions** prevent utilization of several functional units and deeper pipelines;
- ▶ The **control logic** requires a **high hardware complexity** and results in high energy requirements

Note:

A lot of (older) these architectural enhancements require no or only minor **support from the compiler or operating system**

- ▶ Dynamic instruction scheduling managed by hardware;
- ▶ Cache management with reload strategy handled by hardware;

VLIW-Processors (Very Long Instruction Word)

- ▶ **Basis:** The functional units are controlled by **static scheduling** by letting the **compiler** arrange independent instructions into long instruction words; ↪ the **parallel** processing of instructions is **explicitly** visible in the instruction words;
↪ EPIC (explicitly parallel instruction computers)
- ▶ **Examples:** Intel IA-64-Architecture (Itanium, Itanium 2);
Trimedia TM32-Architecture (processor for embedded systems);
Texas Instruments TMS320c6x (Digital Signal Processor);
- ▶ The **micro-processor hardware** is **simpler**, because the complexity for **dynamic scheduling** by the processor is omitted.
Using suitable analysis methods, the **compiler** can ensure that the functional units are efficiently utilized.
- ▶ **Requirement:** The compiler has to recognize independent instructions and has to arrange them into the instruction word;

Architecture of Parallel Systems

Processor Architecture and Technology Trends

Flynn's Taxonomy of Parallel Architectures

Memory Organization of Parallel Computers

Interconnection Networks

Routing and Switching

Flynn's Classification

- ▶ Flynn's classification characterizes parallel computers by the **organization of the global control** and the resulting **data and control flows**.
- ▶ The following four classes are distinguished:
 1. SISD - Single Instruction Single Data: in each processing step one instruction is applied to one value;
 2. MISD - Multiple Instructions Single Data: in each processing step different instructions are applied to one value;
 3. SIMD - Single Instruction Multiple Data: in each processing step the same instruction is applied to a number of values;
 4. MIMD - Multiple Instructions Multiple Data: in each step a number of instructions is applied to a number of values;
- ▶ A lot of computers in the past had a **SIMD** architecture.
GPUs are based on the SIMD concept; also SSX and AVX extensions
- ▶ Most of the parallel computers used today are **MIMD**

Architecture of Parallel Systems

Processor Architecture and Technology Trends

Flynn's Taxonomy of Parallel Architectures

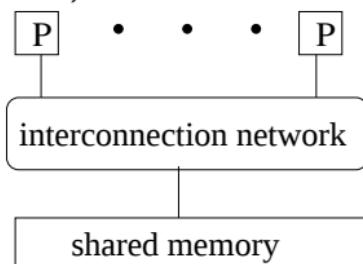
Memory Organization of Parallel Computers

Interconnection Networks

Routing and Switching

Shared Address Space: Multi-Processor Systems:

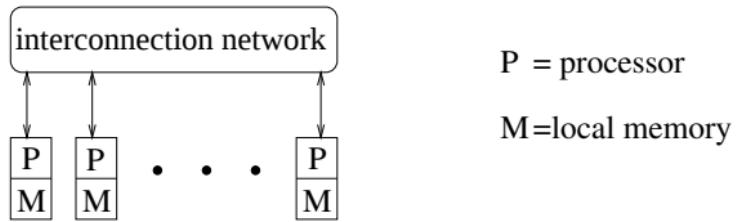
- ▶ Processors are connected to **global memory** via a network (**SMM - Shared Memory Machines**):



- ▶ **Data exchange** takes place through shared variables by means of coordinated reading and writing;
→ **Explicit synchronization operations**;
- ▶ Implicitly, **local and non-local memory accesses** can be distinguished.
- ▶ Option: **Symmetric Multiprocessors (SMP): Each access of shared memory** requires the same amount of time for each processor, regardless of memory address.
SMPs typically consist of a small number of **identical processors**, which are normally connected by a **central bus**.
- ▶ **Examples:** SGI Altix, server architectures.

Distributed Address Space: Multi-Computer System:

- ▶ Multi-computer systems consist of **nodes** (processors with local memory), which are interconnected via a network:



- ▶ **Data exchange** is accomplished through **message passing**;
→ Data exchange through **explicit communication operations**.
- ▶ **Examples:** IBM Blue Gene systems (BG/L, BG/P, BG/Q),
PC-Cluster: Myrinet-Network, Gigabit-Ethernet, Infiniband

Reducing memory access times

- ▶ Memory access time has a large influence on program performance.
- ▶ DRAM chips are used to build main memory of computers
- ▶ Increase of **memory capacity** of DRAM memory chips by 40% to 60% per year since 1977.
- ▶ Decrease of **memory access latency** of DRAM memory chips in average by about 25% per year.
 - ~~ memory access time does not keep pace with processor performance improvement
 - ~~ Problem of memory wall
- ▶ Reducing the average latency observed by a processor when accessing memory can increase the resulting program performance significantly.

Reducing memory access times – Multithreading

- ▶ **interleaved multithreading:** hide the latency of memory accesses by simulating a fixed number of virtual processors for each physical processor.
implicit switch to the next virtual processor after each instruction
 - the **memory latency will be hidden** by executing instructions of other virtual processors
 - **fine-grained multithreading**
- ▶ **coarse-grained multithreading:** switches between virtual processors only on costly stalls, such as level 2 cache misses

Reducing memory access times – Caches

- ▶ A **cache** is a small, but fast memory between the processor and main memory.
- ▶ For each memory access issued by the processor, it is first checked by hardware whether the memory address specified currently resides in the cache.
- ▶ 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.
- ▶ Multicore processors: 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.

Architecture of Parallel Systems

Processor Architecture and Technology Trends

Flynn's Taxonomy of Parallel Architectures

Memory Organization of Parallel Computers

Interconnection Networks

Routing and Switching

Communication Networks of Parallel Computers

- ▶ Used for **communication** in multi-computer systems.
- ▶ Used for **memory access** in multi-computer systems.

Important Design criteria for networks:

- ▶ Network **topology: geometric structure** of processor or memory unit connection.
 - ▶ **Static/direct** networks:
Nodes are interconnected in a point-to-point fashion.
 - ▶ **Dynamic/indirect** networks:
Processors and/or memory units are indirectly connected by several lines and intermediate switches (bus-based or switch-based networks).
- ▶ **Routing technique** - way of sending messages between processors or between processors and memory modules.
 - ▶ **Routing algorithm:** Path selection
 - ▶ **Switching strategy:** transmission mode

Static Networks

- ▶ Static networks can be represented as a **communication graph** $G = (V, E)$, where:
 - ▶ V is the set of **vertices**, each representing a **node** (processor);
 - ▶ E is the set of **edges**, each representing a direct **connection** between two nodes; for example, $(u, v) \in E$ means that there is a direct connection between $u \in V$ and $v \in V$.
- ▶ For each pair of nodes, there should exist a (direct or indirect) connection between them. → The communication graph should be a connected graph.
- ▶ Most of the networks use **bidirectional** connections → representation as an undirected graph.
- ▶ To enable communication between nodes u and v , which are not directly connected to each other, a path between u and v is used;
A path with length k is a sequence of nodes (v_0, \dots, v_k) with $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$.

Characteristics of static networks

- **Diameter:** maximal distance between any two nodes:

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

- **Degree** of a node: number of adjacent edges of the node;
Degree of a graph G : maximal **degree** of the nodes in the graph:

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

- **Bisection bandwidth:** minimal number of edges that must be removed to split the network into **two subnetworks of equal size** with no connection between the two subnetworks:

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

Node and edge connectivity

- ▶ **Node connectivity:** minimal number of nodes that must be removed to **disconnect** the network:

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

with $G_{V \setminus M} = (V \setminus M, E \cap ((V \setminus M) \times (V \setminus M)))$

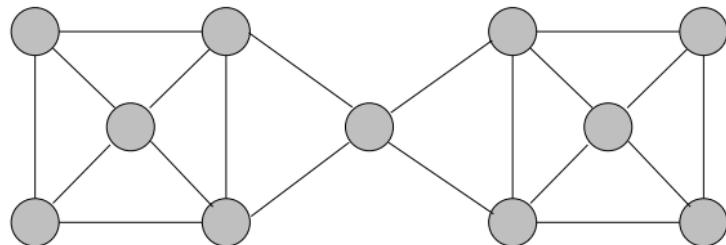
- ▶ **Edge connectivity:** minimal number of edges that must be removed to **break** the network:

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

with $G_{E \setminus F} = (V, E \setminus F)$

Node and edge connectivity – example

Node connectivity of a network can be **smaller** than its **edge connectivity**; Example:



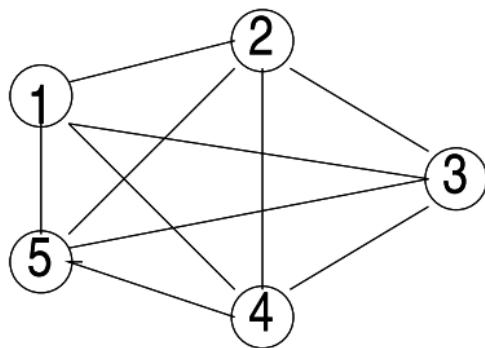
- ▶ Node connectivity = 1
- ▶ Edge connectivity = 2

Network requirements

- ▶ **Small diameter** ensures small distances of message transmission between transmitter and receiver.
- ▶ **Small degree** of every node reduces hardware requirements.
- ▶ **High bisection bandwidth** helps to achieve a high data throughput.
- ▶ **High connectivity** increases reliability.
- ▶ Simple expandability to a larger number of processors increases **scalability**.

Network Example: Complete graph

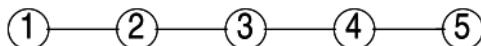
- ▶ **Example 1: Complete graph:** every node has a **direct** connection to every other node in the network.



- ▶ **Diameter:** $\delta(G) = 1$
- ▶ **Degree:** $g(G) = n - 1$
- ▶ **Connectivity:** $nc(G) = ec(G) = n - 1$
- ▶ **Bisection bandwidth:** $B(G) = n^2/4$ for even values of n

Network Example: Linear array

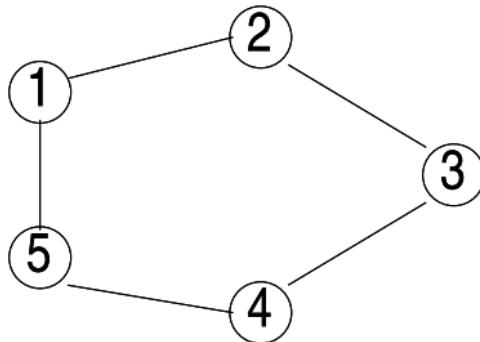
- ▶ **Example 2: Linear array:** linear ordering of processors, so that neighboring processors have a bidirectional connection:
 $V = \{v_1, \dots, v_n\}, E = \{(v_i, v_{i+1}) | 1 \leq i < n\}$



- ▶ **Diameter:** $\delta(G) = n - 1$
- ▶ **Degree:** $g(G) = 2$
- ▶ **Connectivity:** $nc(G) = ec(G) = 1$
- ▶ **Bisection bandwidth:** $B(G) = 1$
- ▶ **No fault-tolerance** with respect to message transmission.

Network Example: Ring

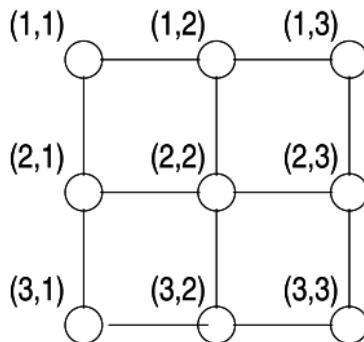
- ▶ **Example 3: Ring:** ring-like ordering of processors; compared to the linear array, an additional edge exists between the first and the last processor.



- ▶ **Diameter:** $\delta(G) = \lfloor n/2 \rfloor$
- ▶ **Degree:** $g(G) = 2$
- ▶ **Connectivity:** $nc(G) = ec(G) = 2$
- ▶ **Bisection bandwidth:** $B(G) = 2$

Network Example: Mesh

- ▶ **Example 4: d-dimensional mesh:** with $n = n_1 \cdot n_2 \cdot \dots \cdot n_d$ nodes; n_j is the number of processors in dimension j :



Nodes: (x_1, \dots, x_d) with $1 \leq x_j \leq n_j$ for $j = 1, \dots, d$

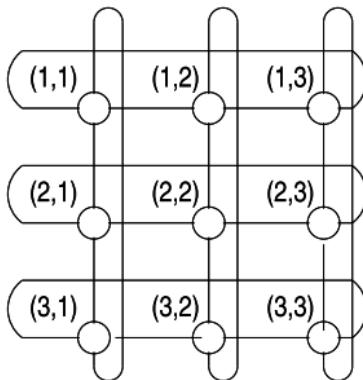
An edge exists between node (x_1, \dots, x_d) and (x'_1, \dots, x'_d) if for a $\mu \in \{1, \dots, d\}$ it is: $|x_\mu - x'_\mu| = 1$ and $x_j = x'_j$ for all $j \neq \mu$.

Special case: $n_j = r = \sqrt[d]{n}$ for all $j = 1, \dots, d$.

- ▶ **Diameter:** $\delta(G) = d \cdot (\sqrt[d]{n} - 1)$
- ▶ **Degree:** $g(G) = 2d$
- ▶ **Connectivity:** $nc(G) = ec(G) = d$
- ▶ **Usage examples:** Intel: 2D-mesh for Paragon (1992) and ASCI Red (1996); Cray Inc: 3D-mesh for Red Storm (2004), Intel Teraflop processor

Network Example: Torus

- ▶ **Example 5: d-dimensional torus:** extension of a d -dimensional mesh with additional edges in each dimension j : an additional edge exists between nodes $(x_1, \dots, x_{j-1}, 1, x_{j+1}, \dots, x_d)$ and $(x_1, \dots, x_{j-1}, n_j, x_{j+1}, \dots, x_d)$.

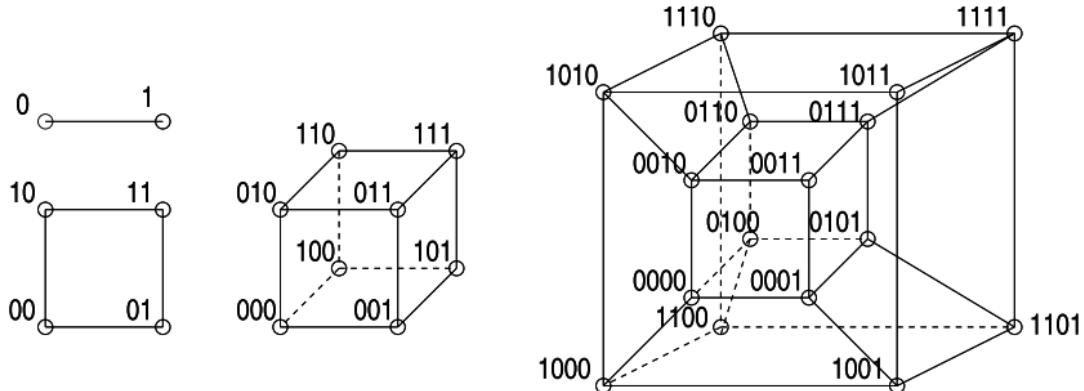


Special case: $n_j = r = \sqrt[d]{n}$ for all $j = 1, \dots, d$.

- ▶ **Diameter:** $\delta(G) = d \cdot \lfloor \sqrt[d]{n}/2 \rfloor$
- ▶ **Degree of all nodes:** $g(G) = 2d$
- ▶ **Connectivity:** $nc(G) = ec(G) = 2d$
- ▶ **Usage examples:** Cray Inc: T3D and T3E (1997, 3D-Torus), X1 (2003, 2D-Torus), Cray XT3, XT4, and XT5 (3D).

Network Example: hypercube

- ▶ **Example 6: k-dimensional cube or hypercube:** recursive construction.



Each node is assigned a k -bit word $\alpha_0 \dots \alpha_j \dots \alpha_{k-1} \rightarrow$ node enumeration.

nodes: $V = \{0, 1\}^k \rightarrow n = 2^k$ nodes.

Edges: an edge exists between node $\alpha_0 \dots \alpha_j \dots \alpha_{k-1}$ and node $\alpha_0 \dots \bar{\alpha}_j \dots \alpha_{k-1}$ for $0 \leq j \leq k - 1$

→ **neighboring** nodes differ in exactly one bit;
i.e., $\bar{\alpha}_j = 1$ for $\alpha_j = 0$ and $\bar{\alpha}_j = 0$ for $\alpha_j = 1$.

Properties Hypercube Network

- ▶ **Hamming distance:**

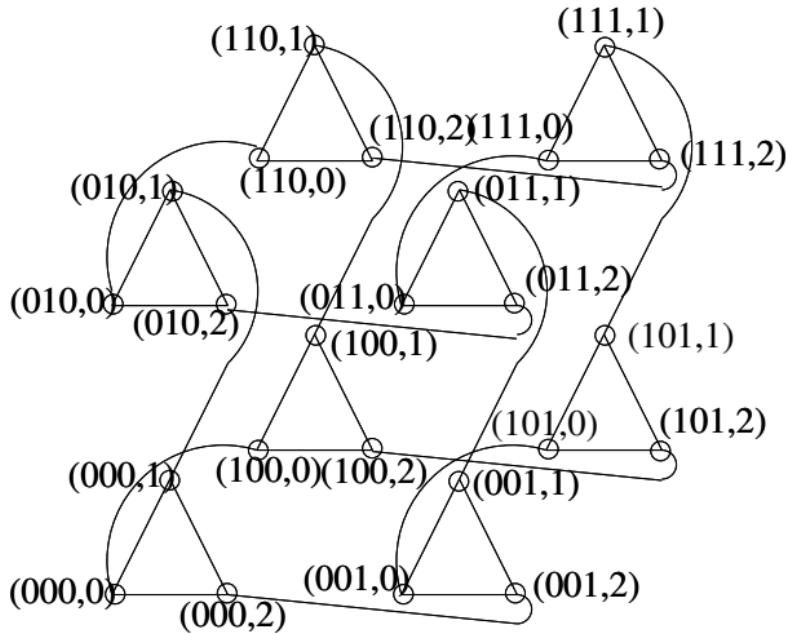
Hamming distance of two binary words of equal length is defined as the number of bit positions, in which the two words differ.

- ▶ **Two nodes of a k -dimensional cube are directly connected if their Hamming distance is 1.**
- ▶ **There exists a path of length d between two nodes $v, w \in V$ with Hamming distance d , where $1 \leq d \leq k$.** Determination by successive bit inversion of the bits in the k -bit word of v , which differ from the k -bit word of w .
- ▶ **Diameter** $\delta(G) = k = \log n$,
Bit representations of nodes can differ by a **maximum of k positions**.
Therefore, there is a path between any two nodes of length $\leq k$.
- ▶ **Degree** $g(G) = k = \log n$,
It is possible to have exactly k bit inversions in a bit word of length k .
The corresponding node has, therefore, k direct neighbors.
- ▶ **Node- and edge connectivity** $nc(G) = k = \log n$.
- ▶ **Usage examples:** SGI: ASCI Blue Mountain (1998), Origin 3000 (2002).

► **Example 7: CCC Network (Cube Connected Cycles):**

A k -dimensional CCC Network results from a k -dimensional hypercube by replacing each node with a ring of k nodes $\rightarrow k \cdot 2^k$ nodes.

Example: $k = 3$:



CCC Network – properties

- ▶ Each node in the ring has one connection to one neighbor of the original hypercube node.
- Set of nodes:** $V = \{0, 1\}^k \times \{0, \dots, k - 1\}$
- $\{0, 1\}^k$ binary representation of the original hypercube node.
 $i \in \{0, \dots, k - 1\}$ position within the ring.
- ▶ **Edge set:** The set of edges can be partitioned into (new) **ring edges** and **hypercube edges**:

Ring edges:

$$F = \{((\alpha, i), (\alpha, (i + 1) \bmod k)) \mid \alpha \in \{0, 1\}^k, 0 \leq i < k\}$$

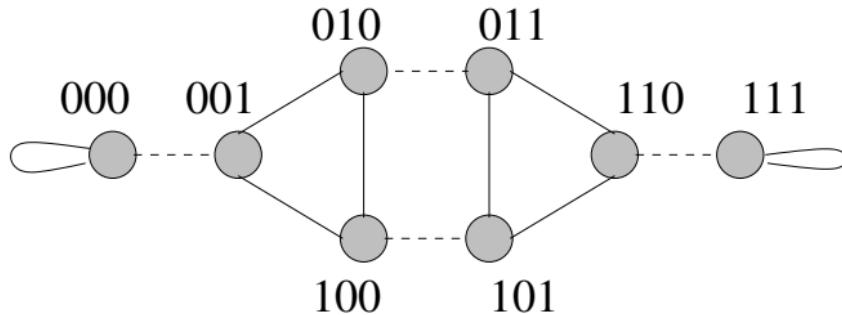
Hypercube edges:

$$E = \{((\alpha, i), (\beta, i)) \mid \alpha_i \neq \beta_i \text{ and } \alpha_j = \beta_j \text{ for } j \neq i\}$$

- ▶ **Degree** $g(G) = 3$.
- ▶ **Connectivity** $nc(G) = 3$
- ▶ **Diameter** $\delta(G) = 2k - 1 + \lfloor \frac{k}{2} \rfloor$

Shuffle-Exchange Network

- ▶ **Example 8:** k -dimensional **Shuffle-Exchange Network**:
 $n = 2^k$ nodes and $3 \cdot 2^{k-1}$ edges.



Nodes are identified by **k -bit words**.

A node α is connected to a node β , if:

1. α and β differ only in the **last (rightmost) bit** (**exchange edge**), or:
2. α results from β by a **cyclic left shift** or a **cyclic right shift** (**shuffle edge**).

Summary of the static properties of networks:

Network G with n nodes	Degree $g(G)$	Diameter $\delta(G)$	Edge connectivity	Bisection bandwidth
Complete Graph	$n - 1$	1	$n - 1$	$(\frac{n}{2})^2$
Linear Array	2	$n - 1$	1	1
Ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d -dimensional mesh $n = r^d$	$2d$	$d(\sqrt[d]{n} - 1)$	d	$n^{\frac{d-1}{d}}$
d -dimensional torus $n = r^d$	$2d$	$d\lfloor \frac{\sqrt[d]{n}}{2} \rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k -dimensional hypercube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k -dimensional CCC network ($n = k2^k$ for $k \geq 3$)	3	$2k - 1 + \lfloor \frac{k}{2} \rfloor$	3	$\frac{n}{2k}$
Complete binary tree ($n = 2^k - 1$)	3	$2 \log \frac{n+1}{2}$	1	1
k -ary d -Cube ($n = k^d$)	$2d$	$d\lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

Connectivity of a k -dimensional Hypercube (1)

The node and edge connectivity of a k -dimensional Hypercube is exactly k :

1. The connectivity is at most k , i.e., $nc(G) \leq k$ and $ec(G) \leq k$.

Explanation:

a node can be disconnected from the graph by removing all k neighbors or all k adjacent edges.

2. The connectivity is at least k , i.e., $nc(G) \geq k$ and $ec(G) \geq k$.

Construction of k independent paths between any pair of nodes v and w of the hypercube.

Remark: **Independent** paths between two nodes do not share any edges, i.e., the paths only share the two nodes v and w .

Consequence: The node and edge connectivity of a k -dimensional Hypercube is exactly k .

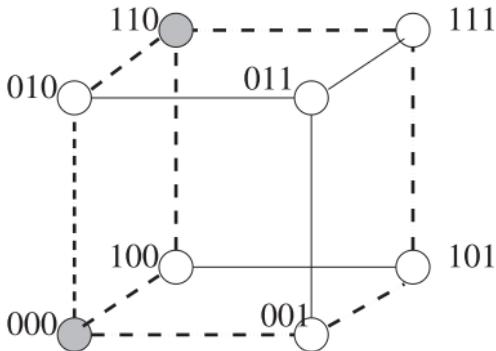
Connectivity of a k -dimensional Hypercube (2)

Construction of k independent paths

- ▶ Nodes v and w in a k -dimensional hypercube with bit names A and B .
- ▶ Bit names A and B differ in l bits, $1 \leq l \leq k$, and these are (possibly after renumbering) the first l bits on the left.
- ▶ Construction:
 - ▶ l paths between nodes v and w , each of length l :
Construction of the i -th path, $i \in \{0, \dots, l-1\}$:
 - ▶ Start with bit name A of v
 - ▶ Invert the bits $i, \dots, l-1$ stepwise
 - ▶ Invert the bits $0, \dots, i-1$ stepwise
 - ▶ $k-l$ paths between nodes v and w , each of length $l+2$:
Construction of the $(i+l)$ -th path, $0 \leq i < k-l$:
 - ▶ Start with bit name A of v
 - ▶ Invert bit $(l+i)$
 - ▶ Invert the bits $0, \dots, l-1$ stepwise
 - ▶ Invert bit $(l+i)$

Connectivity of a k -dimensional Hypercube (3)

Illustration of the independent paths in a 3-dimensional hypercube:



- ▶ In a 3-dimensional hypercube, there exist 3 independent paths from node 000 to node 110.
- ▶ The Hamming distance between nodes 000 and 110 is $l = 2$.
- ▶ There exist 2 paths between nodes 000 and 110 of length $l = 2$.
These paths are $(000, 100, 110)$ and $(000, 010, 110)$.
- ▶ There exists $k - l = 1$ path of length $l + 2 = 4$.
This path is $(000, 001, 101, 111, 110)$.

Embedding of Networks

- ▶ 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 edges between any two nodes in G' also exist between the associated nodes in G .
- ▶ An embedding of G' into G is described by a function $\sigma : V' \rightarrow V$, for which the following holds:
 - ▶ If $u \neq v$ for $u, v \in V'$ holds, then $\sigma(u) \neq \sigma(v)$.
 - ▶ If $(u, v) \in E'$ holds, then $(\sigma(u), \sigma(v)) \in E$.
- ▶ **Interpretation:** If network G' can be embedded into network G , then G is at least as **versatile and flexible** as network G' .
An algorithm for G' can be mapped to an algorithm for G by a renumbering according to σ .

Embedding a Ring into a k-dimensional Hypercube (1)

- ▶ **Procedure:** Mapping of the set of nodes $V' = \{1, \dots, n\}$ of the ring by a bijective function onto the set of nodes $V = \{0, 1\}^k$ of the hypercube. Edges $(i, j) \in E'$ of the ring are mapped to edges in E of the hypercube.
- ▶ **Method:** Construction of a sequence of hypercube nodes, such that there is an edge between neighboring nodes in the sequence.
- ▶ **Reflected Gray Code - RGC:** A k -bit Gray Code is a 2^k -tuple of k -bit numbers, where consecutive numbers in the tuple differ in exactly one bit position.

Embedding a Ring into a k-dimensional Hypercube (2)

Recursive Definition of the reflected gray code:

- ▶ The 1-bit Gray Code is $RGC_1 = (0, 1)$.
- ▶ The 2-bit Gray Code is obtained from RGC_1 by inserting a 0 and a 1 in front of RGC_1 .

The two resulting sequences $(00, 01)$ and $(10, 11)$ are concatenated after reversal of the second sequence.

The resulting sequence is $RGC_2 = (00, 01, 11, 10)$.

- ▶ The k -bit Gray Code RGC_k for $k \geq 2$ is constructed from the $(k - 1)$ -bit Gray Code $RGC_{k-1} = (b_1, \dots, b_m)$ with $m = 2^{k-1}$ where each entry b_i for $1 \leq i \leq m$ is a bit string of length $k - 1$.
- ▶ To construct RGC_k , RGC_{k-1} is duplicated; a 0 is put in front of each element of the original sequence, and a 1 is put in front of each element of the duplicate sequence.

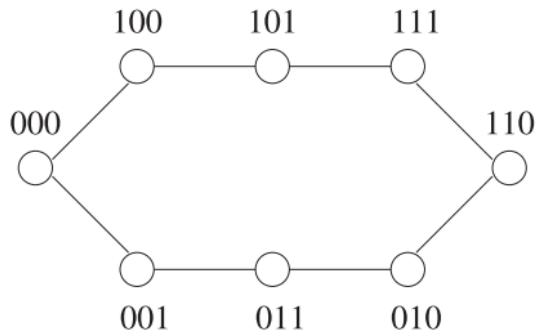
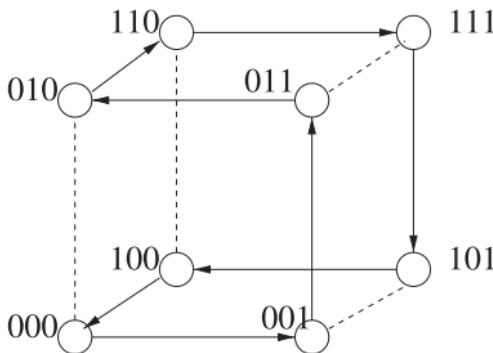
The resulting sequences are $(0b_1, \dots, 0b_m)$ and $(1b_1, \dots, 1b_m)$.

RGC_k is constructed by a reversal of the second sequence and concatenation of the two sequences:

$$RGC_k = (0b_1, \dots, 0b_m, 1b_m, \dots, 1b_1).$$

Embedding a Ring into a k-dimensional Hypercube (3)

- ▶ Example for $k = 3$:



- ▶ **Property:** RGC_k contains node numbering of a k -dimensional cube.
Neighboring elements of RGC_k differ from each other in exactly one bit.
- ▶ **Embedding** of a Ring into a k -dimensional cube:

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

$RGC_k(i)$ is the k -th element of the sequence RGC_k .

Embedding a 2D Mesh into a k -dimensional Hypercube (1)

- ▶ Let G' be a 2D mesh with $n = n_1 \cdot n_2$ nodes.
Let G be a k -dimensional cube with $n = 2^k$ nodes.
- ▶ **Generalization of the embedding of the ring:**
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 Code $RGC_{k_1} = (a_1, \dots, a_{n_1})$ and Gray Code $RGC_{k_2} = (b_1, \dots, b_{n_2})$ are used to construct a $n_1 \times n_2$ matrix M whose entries are k -bit strings.
The matrix is of the form $M(i, j) = \{a_i, b_j\}_{i=1, \dots, n_1, j=1, \dots, n_2}$:

$$M = \begin{bmatrix} a_1, b_1 & a_1, b_2 & \dots & a_1, b_{n_2} \\ a_2, b_1 & a_2, b_2 & \dots & a_2, b_{n_2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n_1}, b_1 & a_{n_1}, b_2 & \dots & a_{n_1}, b_{n_2} \end{bmatrix}$$

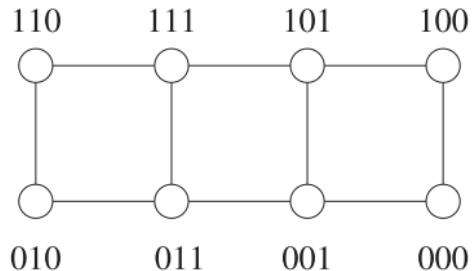
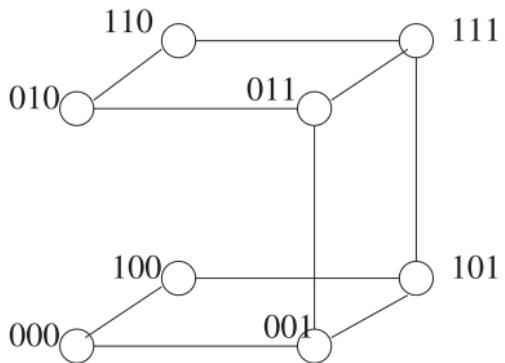
- ▶ **Properties:**
 - ▶ Adjacent nodes of matrix M differ in exactly one bit position.
 - ▶ All elements of M are different bit words of length k .

Embedding a 2D Mesh into a k -dimensional Hyercube (2)

- ▶ Embedding function:

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

- ▶ Example:



Embedding a d -dimensional Mesh into a k -dimensional Hypercube

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

$$\sigma : \{(x_1, \dots, x_d) \mid 1 \leq x_i \leq n_i, 1 \leq i \leq d\} \rightarrow \{0, 1\}^k$$

with $\sigma((x_1, \dots, x_d)) = s_1s_2\dots s_d$ and $s_i = RGC_{k_i}(x_i)$

(where s_i is the x_i -th bit string in Gray Code RGC_{k_i}) defines an embedding into a k -dimensional hypercube.

- ▶ For two mesh nodes (x_1, \dots, x_d) and (y_1, \dots, y_d) that are connected by an edge in the mesh, there exists exactly one dimension $i \in \{1, \dots, d\}$ with $|x_i - y_i| = 1$. For all other dimensions $j \neq i$ it is $x_j = y_j$.
- ▶ For the corresponding hypercube nodes $\sigma((x_1, \dots, x_d)) = s_1s_2\dots s_d$ and $\sigma((y_1, \dots, y_d)) = t_1t_2\dots t_d$, all components $s_j = RGC_{k_j}(x_j) = RGC_{k_j}(y_j)$ for $j \neq i$ are identical and $RGC_{k_i}(x_i)$ differs from $RGC_{k_i}(y_i)$ in exactly one bit position.
 - ~~~ The nodes $s_1s_2\dots s_d$ and $t_1t_2\dots t_d$ are connected by an edge in the k -dimensional cube.

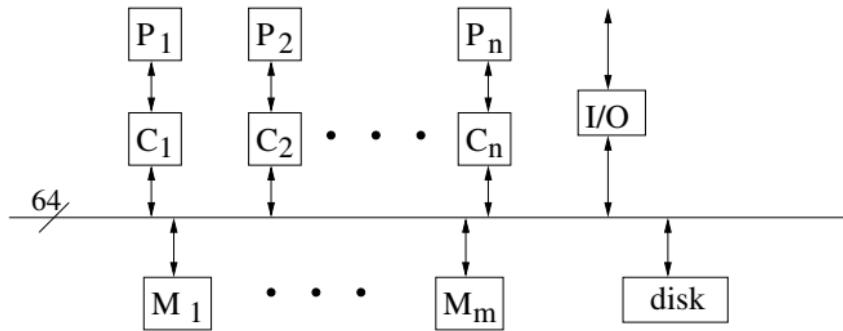
- ▶ Dynamic networks consist of a number of physical **wires** and **switches** in between.
- ▶ Processors and memory modules are connected indirectly via the network (and the switches).
- ▶ The switches are **dynamically configured** according to the requirements of the message transmission.
- ▶ Dynamic networks are mainly used for computers with **shared memory**.

Topological structures of dynamic networks:

- ▶ Bus networks
- ▶ Crossbar networks
- ▶ Multistage switch networks

Bus Networks

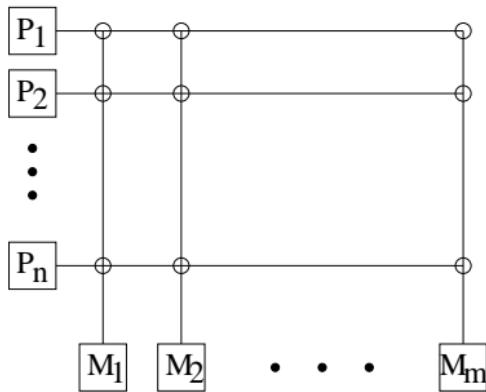
- ▶ A bus network consists of **a set of wires** which can be used to transport data from a sender to a receiver.
- ▶ **Example:** Bus with 64 bit wires to connect processors P_1, \dots, P_n with caches C_1, \dots, C_n to memory modules M_1, \dots, M_m :



- ▶ At each point in time, **only one data transport** can be performed (**time sharing**).
- ▶ **Bus arbiter** coordinates simultaneously incoming data transfer requests (**contention bus**).
- ▶ Bus networks are typically used for a **small number of processors** (32 to max. 64) only. **Used in SMP systems such as Sun Fire, IBM Regatta.**

Crossbar Networks (1)

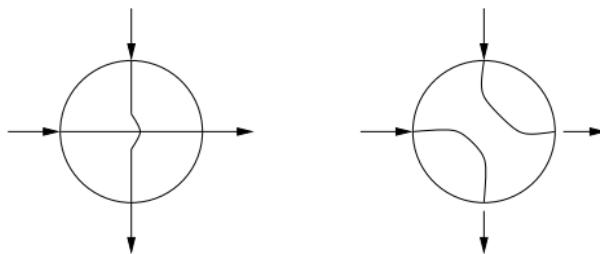
- ▶ An $n \times m$ crossbar network has n **inputs**, m **outputs** and $n \cdot m$ switches.
- ▶ **Example:** n processors P_1, \dots, P_n , m memory modules M_1, \dots, M_m :



- ▶ For each **data transfer** or **memory request** a connection is established within the network from the input to the output.

Crossbar Networks (2)

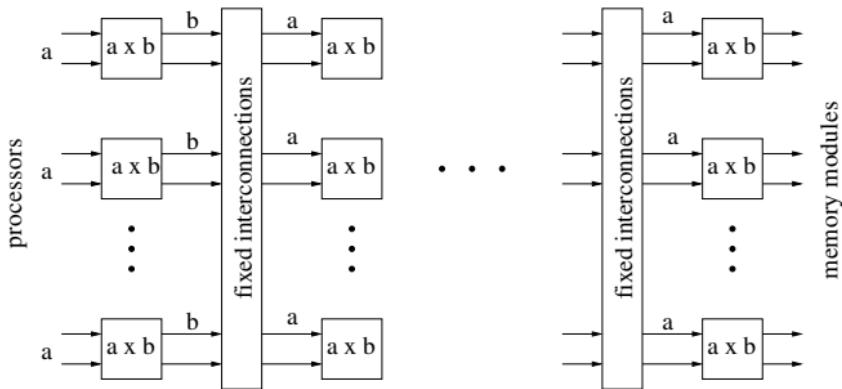
- ▶ **Possible switch positions** according to the requirements of data transfer or memory requests:



- ▶ At each point in time, each memory module can fulfill **only one memory request**;
→ in each column **only one switch** can be set to direction change.
- ▶ Processors can **simultaneously send multiple requests** to different memory modules.
- ▶ **Example** of a crossbar network:
Fujitsu VPP500 (1992) uses a 224×224 crossbar network.

Multistage Switching Networks (1)

- ▶ Multistage switching networks consist of **several stages of switches** with connecting wires between neighboring stages.
- ▶ Often $a \times b$ crossbars are used as switches.
Switches from neighboring stages are connected by **fixed connections**:



- ▶ An access of a processor P to a memory module M is performed by **selecting a path** from P to M and by **setting the switches** on that path appropriately.

Multi-stage Switching Networks (2)

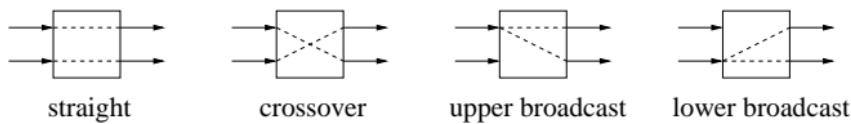
Graph representation:

- ▶ Each stage is represented as a **directed acyclic graph** of depth 1, in which the switches are the nodes.
- ▶ **Connections of adjacent stages** can be described by a permutation-function $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$, such that the output links of stage i are connected to the input links $(\pi(1), \dots, \pi(n))$ of stage $i + 1$.

Typical construction of a multistage interconnection network:

- ▶ $n = 2^{k+1}$ **inputs** and $n = 2^{k+1}$ **outputs**.
- ▶ $k + 1$ **stages** with 2^k nodes each ($(k + 1)2^k$ nodes overall).
- ▶ 2×2 crossbar switches as **switches**.

Switch configurations of a 2×2 crossbar switch:



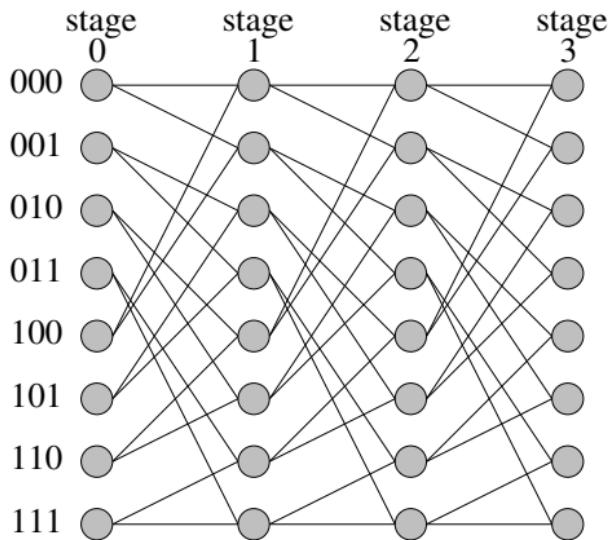
Examples: Omega-, baseline- or butterfly network.

Omega Network (1)

- ▶ An $n \times n$ omega network consists of $\log n$ stages with $n/2$ switches each (2×2 crossbar switches)
→ **($n/2$) $\log n$ switches in total.**
- ▶ Connection between the stages (**gluing function**) is the same for all stages.
- ▶ The switches are identified by pairs (α, i) where:
 - ▶ $\alpha \in \{0, 1\}^{\log n - 1}$ is a bit string of length $(\log n - 1)$ representing the position of a switch within a stage;
 - ▶ $i \in \{0, \dots, \log n - 1\}$ is the **stage number**.
- ▶ **Gluing function:** There exists an edge from switch (α, i) in stage i to **two switches** $(\beta_1, i + 1)$ and $(\beta_2, i + 1)$ in stage $i + 1$, which are defined as follows:
 1. β_1 results from α by a **cyclic left shift**, and
 2. β_2 results from β_1 by **inverting the last (rightmost) bit** (after the cyclic left shift of α).

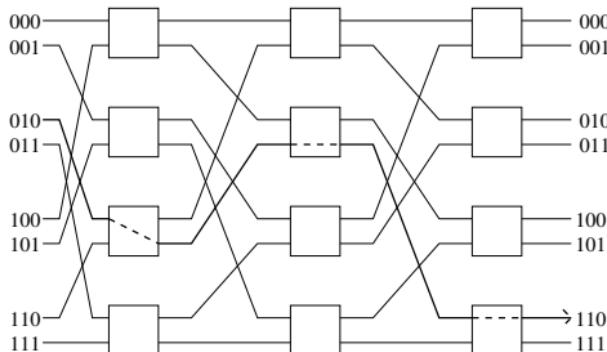
Omega Network (2)

- ▶ **Example:** 16×16 omega network with 4 stages and 8 switches per stage:



Routing in the Omega network (1)

- ▶ To forward a message from input channel with bit name α to output channel with bit name β the switch on stage k , $k = 0, \dots, \log n - 1$ considers the k -th bit β_k (from the left) of the destination name β and selects the output link using the following rule:
 - ▶ If the k -th bit $\beta_k = 0$ the message will be forwarded over the **upper link of the switch**.
 - ▶ If the k -th bit $\beta_k = 1$ the message will be forwarded over the **lower link of the switch**.
- ▶ **Example:** 8×8 omega network with path **010** to **110**:

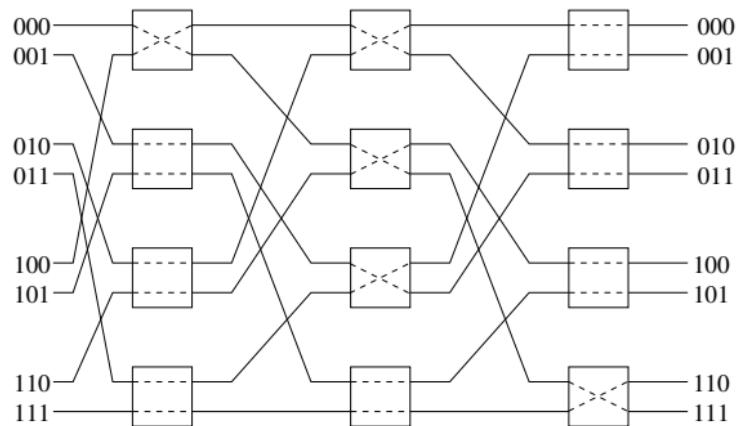


Routing in the Omega network (2)

- ▶ **The maximum** number of messages to be sent **simultaneously** over the network from different inputs to different outputs is n .
- ▶ **Example:** Message transmission with $n = 8$ in an 8×8 Omega network:

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

Corresponding **switch positions**:



Routing in the Omega network (3)

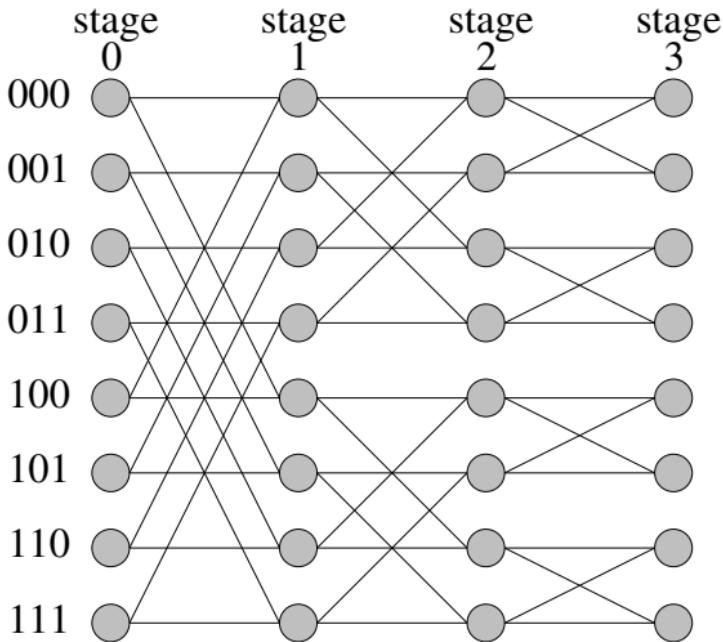
- ▶ Many simultaneous message transmissions that can be described by permutations $\pi^8 = \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ **cannot be executed concurrently**, since **network conflicts** occur.
Example: The two message transfers of $\alpha_1 = 010$ to $\beta_1 = 110$ and from $\alpha_2 = 000$ to $\beta_2 = 111$ in an 8×8 omega network lead to a conflict.
- ▶ These kinds of conflicts occur, since there is exactly one path for any pair (α, β) of input and output channels, i.e., there is no alternative to avoid the critical switch. **Networks with this characteristic are called blocking networks.**
- ▶ Conflicts in blocking networks can be resolved through multiple transmissions over the network.

Butterfly network (Banyan network) (1)

- ▶ A k -dimensional butterfly network consists of $n = 2^{k+1}$ **input links** and $n = 2^{k+1}$ **output links**.
- ▶ There are $k + 1$ stages with 2^k switches each.
→ $(k + 1)2^k$ switches overall.
- ▶ The switches are **identified** by pairs (α, i) with:
 - ▶ $i \in \{0, \dots, k\}$ denotes the stage of the switch.
 - ▶ the k -bit word $\alpha = \{0, 1\}^k$ denotes the position of the switch within the stage.
- ▶ **Gluing function:** The switches (α, i) in stage i and $(\beta, i + 1)$ in stage $i + 1$ are connected if either:
 1. α and β are identical (**straight edge**), or
 2. α and β differ from each other in exactly the $(i + 1)$ th bit from the left (**cross edge**).

Butterfly network (2)

- ▶ **Example:** 16×16 butterfly network with 4 stages:

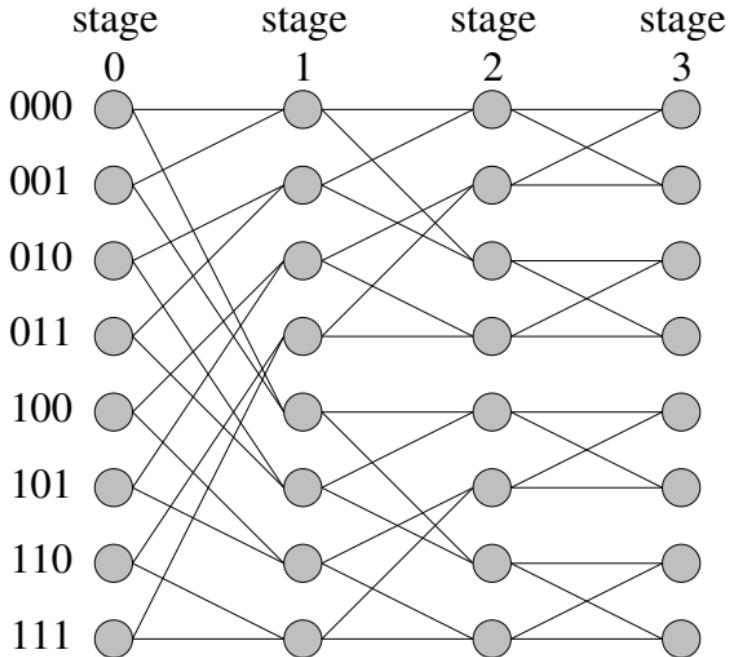


Baseline Network (1)

- ▶ The k -dimensional baseline network has the same number of nodes, edges and stages as the k -dimensional butterfly network.
- ▶ **Gluing function:** switch (α, i) , $0 \leq i \leq k$, is connected to switch $(\alpha', i + 1)$ if and only if:
 1. the k -bit word α' results from α by a **cyclic right shift** of the last $k - i$ bits of α , or
 2. the k -bit word α' results from α by first inverting the **last (rightmost) bit** of α and then performing a **cyclic right shift** of the last $k - i$ bits.

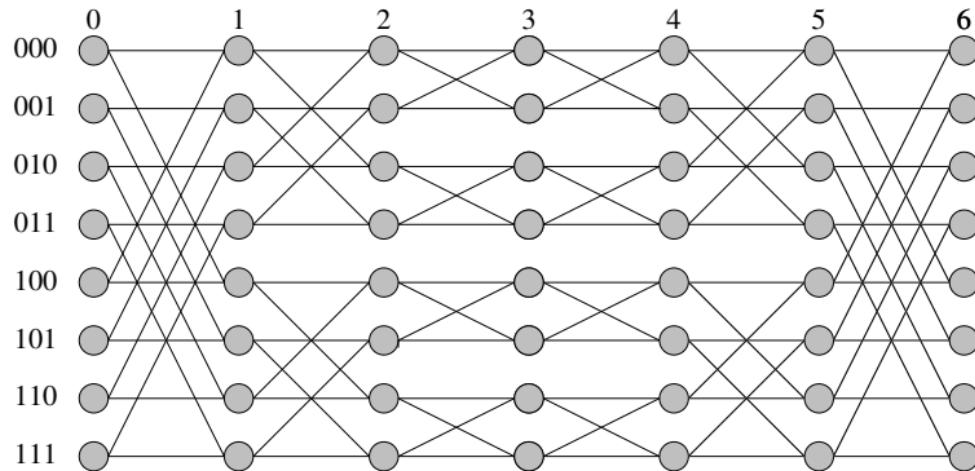
Baseline Network (2)

- ▶ **Example:** 16×16 baseline network with 4 stages:



Beneš Network

- ▶ The k -dimensional Beneš network consists of **two k -dimensional butterfly networks connected to each other**:
 - ▶ First $k + 1$ stages are a butterfly network.
 - ▶ Last $k + 1$ stages are a **reverted** butterfly network.
 - ▶ The last ($k + 1$ st) stage of the first butterfly network and the first stage of the reverted butterfly network are merged.
→ $2k + 1$ stages with $N = 2^k$ nodes/switches per stage.
- ▶ **Example:** Beneš network with 8 inputs:

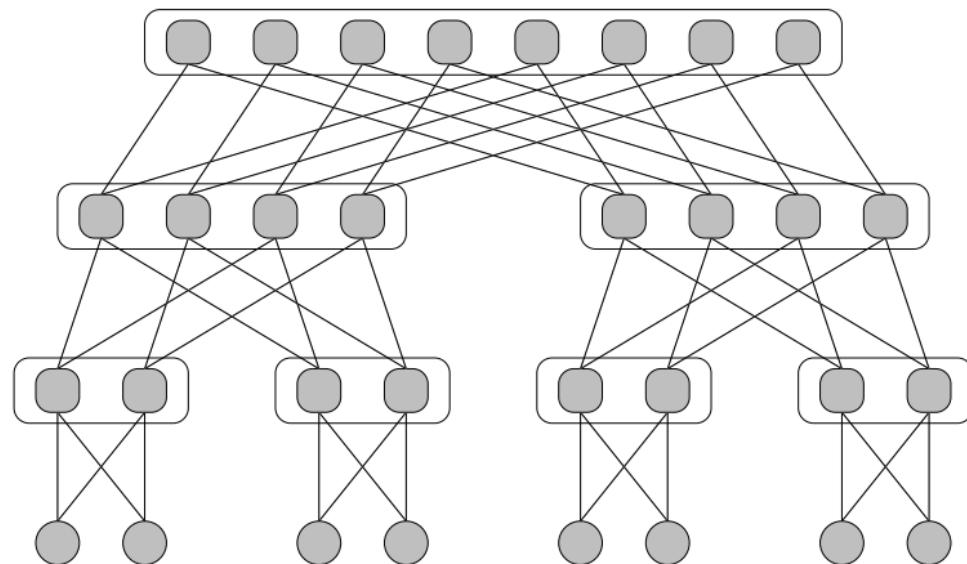


Fat Tree Network (Dynamic Tree Network) (1)

- ▶ The **basic structure** of a fat tree network is a complete binary tree:
 - ▶ n interconnected **processors** are located in the **leaves** of the tree (level 0).
 - ▶ **Inner nodes** are the **switches** whose properties depend on the tree level.
- ▶ **Properties of the switches** on level i , $i = 1, \dots, \log n$:
 - ▶ Each switch has 2^i **input links** and 2^i **output links**, $i = 1, \dots, \log n$.
 - ▶ Internal realization, e.g. from 2^{i-1} switches with two input and output links each.
 - Each level i has $n/2$ switches, which are **grouped** into $2^{\log n - i}$ nodes.

Fat Tree Network (2)

- ▶ **Example:** Fat tree for 16 processors with 4 levels (the leaf nodes are not shown):



Architecture of Parallel Systems

Processor Architecture and Technology Trends

Flynn's Taxonomy of Parallel Architectures

Memory Organization of Parallel Computers

Interconnection Networks

Routing and Switching

Routing Algorithms

Given:

- ▶ Two processor nodes A and B on a network **without direct** point-to-point communication.
- ▶ Message m should be sent from A to B .

Find:

Path in the network, e.g. in its graph representation, which connects A and B .

Routing algorithm

finds a **path** in the network over which a message from sender A to receiver B should be sent.

How is a routing algorithm specified?

Mostly, **topology-specific**:

at each **intermediate node** on the path from sender to receiver the subsequent node is being determined.

Requirements of a routing algorithm

Situation: Typically, there is **number of messages** m_i to be sent over the network between different processor nodes A_i and B_i .

Requirements of a **good** routing algorithm:

- ▶ **Uniform** utilization of the network.
- ▶ **Low** communication time on average.
- ▶ **Free of deadlocks.**

Criteria for choosing a routing algorithm:

- ▶ **Topology** determines the path connecting sender A with receiver B .
- ▶ **network contention** under high message load (competing demands on the connecting line → message delay).
- ▶ **Congestion avoidance** (too many messages accessing limited resources → messages get lost).

Classification of routing algorithms

- ▶ Consideration of network utilization
 - ▶ **Deterministic** routing algorithms: Establishment of a fixed path:
 - ▶ Source-based (Source = Sender): Sender chooses a path.
 - ▶ Distributed (intermediate nodes): Decision-making on intermediate nodes.
Example: dimensionally ordered routing.
 - ▶ **Adaptive** routing algorithms (dynamic): Consideration of dynamic utilization.
- ▶ Path length
 - ▶ **Minimal** routing algorithms: Choice of the shortest path → impacts communication time.
 - ▶ **Non-minimal** routing algorithms: Choice of a non-deterministic path based on observation of network utilization.

Dimensionally ordered routing

Routing path is chosen according to:

- ▶ Position of Sender and Receiver nodes,
- ▶ Order of dimensions.

Typical topologies:

- ▶ XY-routing for 2-dimensional mesh.
- ▶ E-Cube routing for k -dimensional hypercube.

XY -routing for 2-dimensional mesh

- ▶ Node names: X and Y coordinates of mesh topology (X coordinate - horizontal, Y coordinate - vertical).
- ▶ Node A is expressed as: (X_A, Y_A) .
- ▶ Node B is expressed as: (X_B, Y_B) .
- ▶ Message should be sent from Sender A to Receiver B .
- ▶ Routing algorithm:
 - ▶ First X direction, then Y direction.
 - ▶ Message will be sent in direction X until the X coordinate X_B is reached, i.e. till (X_B, Y_A) .
 - ▶ Message will then be sent in direction Y until the Y coordinate Y_B is reached, i.e. till (X_B, Y_B) .
- ▶ Path length: $|X_A - X_B| + |Y_A - Y_B|$ minimal routing algorithm.

E-Cube routing for k -dimensional hypercube

- ▶ Routing in k -dimensional cube with $n = 2^k$ nodes.
- ▶ Node names: binary word of length k .
- ▶ Names of k physical neighbors: inversion of exactly one of the k bits of the name.
- ▶ Node A is expressed as: $\alpha = \alpha_0, \dots, \alpha_{k-1}$.
- ▶ Node B is expressed as: $\beta = \beta_0, \dots, \beta_{k-1}$.
- ▶ Message should be sent from Sender A to Receiver B .
- ▶ Path used for sending the message:
$$A = A_0, A_1, \dots, A_i = B$$
- ▶ Dimensionally ordered routing for a k -dimensional cube uses the following to determine the routing path:
 - ▶ k -bit name of the sender.
 - ▶ k -bit name of the receiver.
 - ▶ k -bit name of intermediate nodes.

E-Cube routing for k -dimensional hypercube (2)

Routing algorithm:

Starting with A each **successor** node A_{i+1} of A_i is chosen according to the dimension, to which the message should be sent in the next step.

more precisely:

Let A_i be nodes on the path $A = A_0, A_1, \dots, A_i = B$, from which the successive one should be chosen.

Bit representation of A_i : $\gamma = \gamma_0, \dots, \gamma_{k-1}$.

- ▶ A_i calculates the k -bit word $\gamma \oplus \beta$, where the operator \oplus is the bitwise exclusive or (i.e. $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$)
 - ▶ If $\gamma \oplus \beta \neq 0$:
 A_i sends the message in the direction of dimension d , where d is the rightmost position of $\gamma \oplus \beta$ with value 1.

- ▶ ▶ The corresponding nodes A_{i+1} on the routing path are found by inverting of the d -th bit in γ , i.e. the node A_{i+1} has the k -bit name $\delta = \delta_0, \dots, \delta_{k-1}$,
where $\delta_j = \gamma_j$ for $j \neq d$ and $\delta_d = \gamma_d$
- ▶ If $\gamma \oplus \beta = 0$:
The destination node is reached.

Example:

- ▶ A with bit name $\alpha = 010$
 - ▶ B with bit name $\beta = 111$
1. $\alpha \oplus \beta = 010 \oplus 111 = 101$
Send in the dimension $d = 2$ to A_1 with name $\gamma = 011$
 2. $\gamma \oplus \beta = 011 \oplus 111 = 100$
Send in the dimension $d = 0$ to A_2 with name $\gamma = 111$

Note: Dimensions are enumerated from left to right.

Danger of Deadlocks in Routing Algorithms

...due to several messages in the network.

Proof of Deadlock freedom:

consider possible dependencies between communication channels based on:

- ▶ Topology
- ▶ Routing algorithm

Definition of dependencies:

Communication channel l_2 is dependent on communication channel l_1 if it is possible within the routing algorithm to choose a path which sends the message on l_1 and then **directly** to l_2 .

Channel dependency graph (for fixed communication channels and fixed routing algorithm):

Nodes \simeq Communication channels

Directed edge \simeq Dependencies between communication channels

If the channel dependency graph doesn't contain loops - the corresponding routing algorithm on the chosen topology is deadlock-free.

Deadlock Freedom for XY-Routing

Channel dependency graph for XY-routing:

- ▶ Nodes:

For each unidirectional connection of the 2D $n_x \times n_y$ mesh there exists a node,
i.e for each bidirectional edge of the mesh there exist two nodes.

- ▶ Edges:

There is a dependency of connection u on connection v if v :

- ▶ is in the same horizontal alignment or
- ▶ is in the same vertical alignment or
- ▶ has a shift of 90 degrees up or down
with respect to u .

Deadlock Freedom for XY-Routing (2)

To ensure deadlock freedom all the unidirectional connections of the mesh are numbered as follows:

- ▶ **Horizontal edges:**

Edges of nodes (i, y) to nodes $(i + 1, y)$
get a number $i + 1$, $i = 0, \dots, n_x - 2$ for each y -position.

The opposite edges of nodes $(i + 1, y)$ to nodes (i, y)
get a number $n_x - 1 - (i + 1) = n_x - i - 2$, $i = 0, \dots, n_x - 2$.

(The edges in ascending x -direction ascend with $1, \dots, n_x - 1$, the edges in descending direction of x are numbered $0, \dots, n_x - 2$)

- ▶ **Vertical edges:**

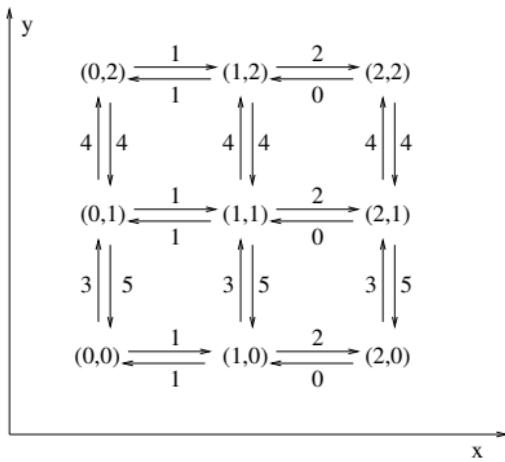
Edges between nodes (x, j) and $(x, j + 1)$
get a number $j + n_x$, $j = 0, \dots, n_y - 2$

The opposite edges get numbered: $n_x + n_y - (j + 1) = n_x + n_y - j - 1$

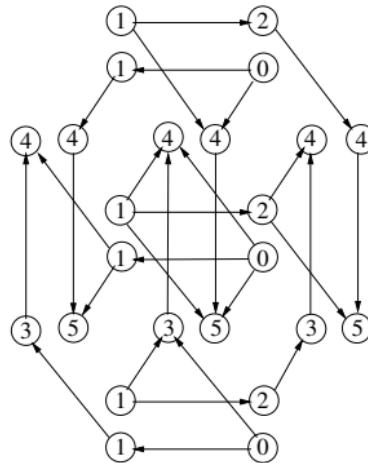
Deadlock Freedom for XY-Routing (3)

3×3 mesh and the according channel dependency graph for XY-routing case.

2D mesh with 3×3 nodes



channel dependence graph



Deadlock Freedom for XY-Routing (4)

Observation:

All edges in the channel dependency graph go from an edge with the **lower** number to an edge with the **higher** number.

A transmission delay on the routing path can only occur if after the message has been sent on the channel v with number i it is waiting for the next edge w with number $j > i$ to get free. And that edge is currently used by a different message (**delay condition**).

Deadlock situation:

There are a number of messages N_1, \dots, N_k and network edges n_1, \dots, n_k where:

- ▶ Each message N_i for $1 \leq i < k$
 - ▶ uses edge n_i for transfer and
 - ▶ waits for the edge n_{i+1} to become free.

- ▶ Message N_k uses the n_k edge and waits for the n_1 edge to become free from N_1 .

The following holds:

If $n()$ is the edge numbering used above, due to the **delay conditions**, it must be:

$$n(n_1) < n(n_2) < \dots < n(n_k) < n(n_1)$$

This is a contradiction and it follows that no deadlock can occur.

Further deterministic routing algorithms:

Source-based routing: the sender chooses the **whole** path for the message transfer.

- ▶ For each node on the path the next output channel is already defined.
- ▶ The sequence of output channels a_0, \dots, a_{n-1} is set as the message header.
- ▶ Routing information from the header is updated at the nodes on the path , i.e. the communication channel which was just used will be removed from the information.
- ▶ Possible disadvantage: Message is too long.

Table oriented routing:

- ▶ Each node of the networks has a **routing table**:
For each destination address the next output channel/node is given.
- ▶ Destination information of the message and the routing table produce the output channel to be used.
- ▶ Disadvantage: Tables may be too big.

Adaptive Routing Algorithms

Adaptive routing algorithms can

- ▶ be **minimal** or **non-minimal**;
- ▶ support **backtracking** or not.

Turn Model

- ▶ Deadlocks are avoided by a suitable selection of turns that are allowed for the routing
Turn models are available for *n*-**dimensional meshes** and general *k*-ary *n*-**cubes**.
- ▶ Goal: identify the minimal number of turns that must be prohibited to avoid the occurrence of cycles.
- ▶ Examples:
 - ▶ West-First routing for a 2-dimensional mesh
 - ▶ *P*-cube routing for an *n*-dimensional hypercube

Motivation

Deadlocks occur if messages change their direction of transmission in such a way that a cyclic wait results.

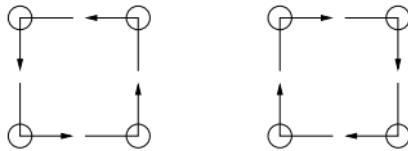
Deadlocks can be prevented by prohibiting certain direction changes.

Example

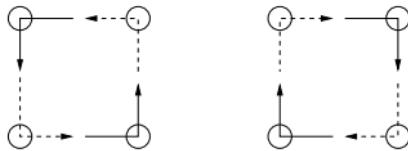
- ▶ XY-routing prohibits all turns from vertical to a horizontal direction.
- ▶ As a result, only 4 of the 8 possible turns in a 2-dimensional mesh are allowed.
- ▶ These four possible direction changes:
 - ▶ Allow **no loops**, making deadlocks impossible
 - ▶ Make adaptive routing impossible.

Turns for a Two-dimensional Mesh

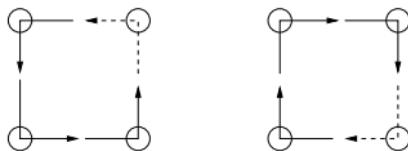
possible turns in a 2D mesh



turns allowed for XY-Routing



turns allowed for West-First-Routing



turns allowed:



turns not allowed:

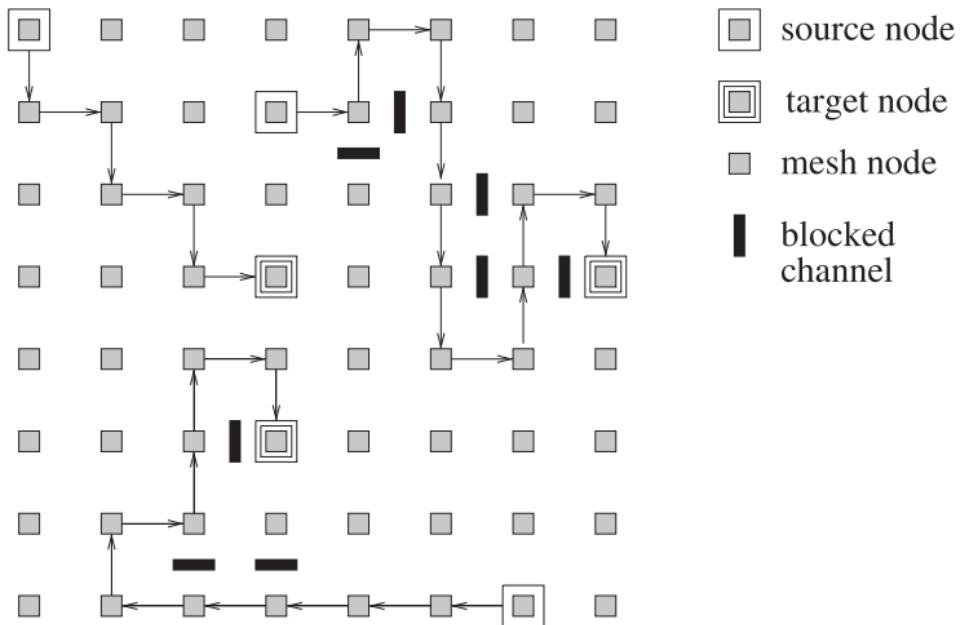


West-First Routing for Two-dimensional Mesh (1)

- ▶ Only **two** of the 8 possible direction changes are prohibited. These are direction changes to the left (west).
- ▶ Routing paths are set up as follows:
 - ▶ The message is first sent to the west (left) until at least the target x-coordinate is reached.
 - ▶ Then the message is routed adaptively in the directions south (down), east (right) or north (up).
- ▶ West-First routing is deadlock-free, since loops are eliminated.
- ▶ If only minimal routing paths are used, the algorithm is only adaptive if the receiver is located in the east (right).
- ▶ If non-minimal paths are also possible, the algorithm is always adaptive.

West-First Routing for Two-dimensional Mesh (2)

3 paths using West-First routing in an 8×8 mesh: one minimal, two non-minimal paths.



P-Cube Routing

P-Cube routing is a turn model for an n -dimensional hypercube.

- ▶ Consider sender A with n -bit name $\alpha_0 \dots \alpha_{n-1}$ and receiver B with n -bit name $\beta_0 \dots \beta_{n-1}$.
- ▶ The number of different bits is the Hamming distance from A to B and is the minimal length of any routing path.
- ▶ The set

$$E = \{i \mid \alpha_i \neq \beta_i, i = 0, \dots, n-1\}$$

of positions of the different bits is partitioned into the following two sets:

$$E_0 = \{i \in E \mid \alpha_i = 0 \text{ and } \beta_i = 1\} \text{ and } E_1 = \{i \in E \mid \alpha_i = 1 \text{ and } \beta_i = 0\}$$

- ▶ Sending a message from A to B according to these sets proceeds in two phases:
 - (i) First, the message is sent into the dimensions in E_0 ,
 - (ii) Then the message is sent into the dimensions in E_1 .

Virtual Channels (1)

- ▶ Virtual channels are often used for minimal adaptive routing algorithms.
- ▶ Physical connections are split into several **virtual channels**
(Additional hardware required, such as multiplexer)
The use of several physical channels would be too expensive.
- ▶ Each virtual channel has its own buffer.
- ▶ Assignment of the physical channels to virtual ones must be performed
in a **fair** way,
i.e., each virtual connection should be continuously reused.

Virtual Channels (2)

Example: Minimal adaptive routing for a 2-dimensional mesh.

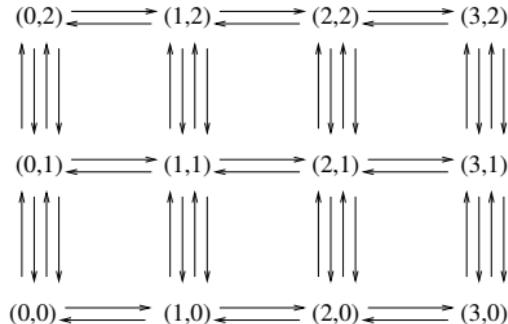
The 2-dimensional mesh is split up into two sub-networks, the $+X$ -subnet and $-X$ -subnet.

- ▶ Each subnet contains all nodes, but only a subset of the virtual channels.
- ▶ Subnet $+X$ contains all channels between nodes in the vertical direction, but in horizontal direction only the channels in positive direction.
- ▶ Subnet $-X$ also contains all channels between nodes in the vertical direction (which is possible when using virtual channels), but in horizontal direction only the channels in negative direction.
- ▶ Messages from node A with x -coordinate x_A to node B with x -coordinate x_B are sent over the $+X$ network if $x_A < x_B$.
- ▶ Messages from A to B are sent over $-X$ if $x_A > x_B$.
- ▶ For $x_A = x_B$ any of the two subnets can be used.
- ▶ The exact selection can be based on load information of the network.

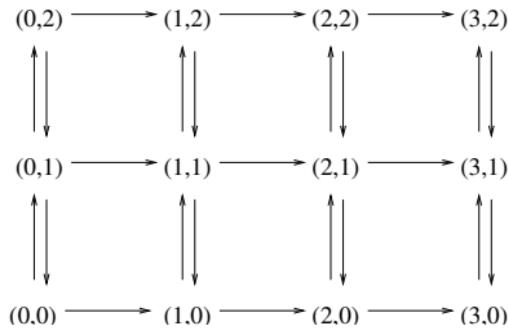
This minimal adaptive routing algorithm is deadlock-free.

Virtual Channels (3)

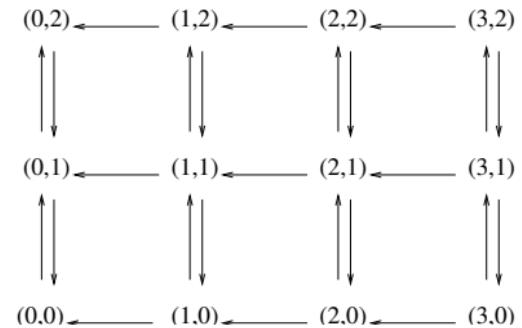
2D mesh with virtual channels in y direction



+X network



-X network



Routing in the Omega Network (1)

Distributed control scheme, in which every switch can forward the message *without* any coordination with the other switches.

To forward a message from the input channel α to the output channel β the switch on stage k ($k = 0, \dots, \log n - 1$) uses the k -th bit β_k (from the left) of the target name β and chooses the according output channel using these rules:

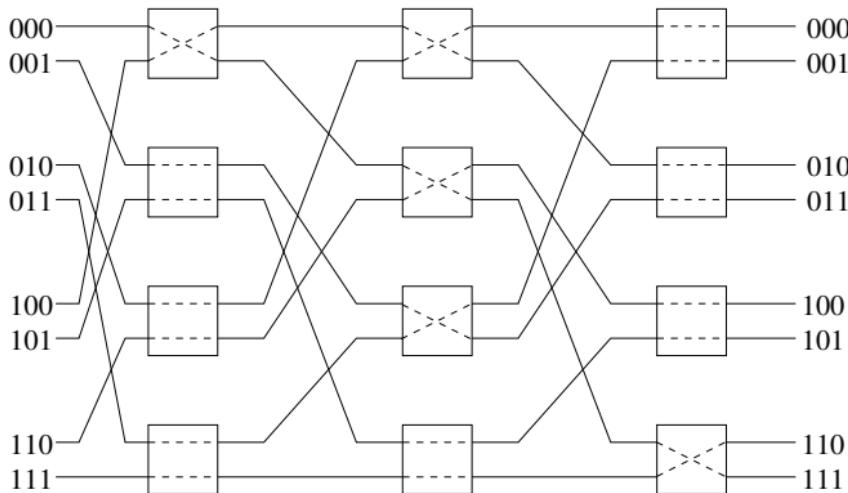
- ▶ If the k -th bit is $\beta_k = 0$ then the message will be forwarded over the upper link of the switch.
- ▶ If the k -th bit is $\beta_k = 1$ then the message will be forwarded over the lower line of the switch.

Routing in the Omega Network (2)

The maximum number of simultaneous transfers from different senders to different receivers over an Omega network is n .

Example: Concurrent message transfer with $n = 8$ in an 8×8 Omega network according to the permutation:

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

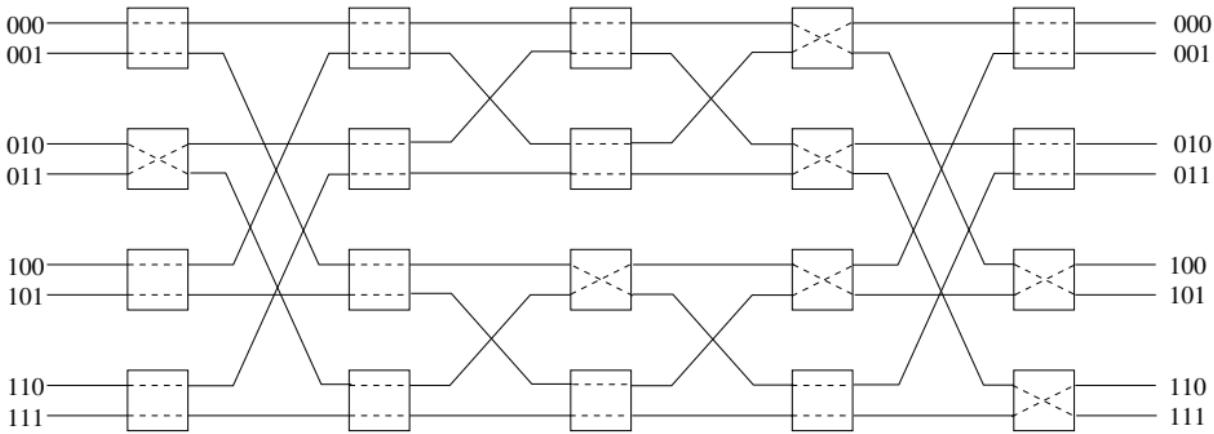


Routing in the Omega Network (3)

- ▶ Note: Many transmissions given through a permutation $\pi^8 = \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ **cannot be performed in one step** (i.e., concurrently to each other), since there may be **conflicts** in the network.
Example: The two message transfers from $\alpha_1 = 010$ to $\beta_1 = 110$ and from $\alpha_2 = 000$ to $\beta_2 = 111$ in an 8×8 omega network lead to a conflict.
- ▶ Conflicts of this type **cannot be resolved**, since for each pair (α, β) of input node and output node there exists **exactly one possible connection**, i.e., there is no alternative to avoid a critical switch.
- ▶ Networks with these properties are called **blocking networks**.

Routing in the Beneš Network

The Beneš network is a non-blocking network:



8 × 8 Beneš network with message transfer organized according to this permutation:

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

Switching Strategies

A **switching strategy** (**switching technique**) defines **how** the message is forwarded over the path chosen by the routing algorithm from the sender to the receiver, i.e.:

- ▶ **Whether and how** a message is split into pieces, which are called packets or flits (flow control units).
- ▶ **How** the transmission path from the sender to the receiver is allocated (fully or partially).
- ▶ **How** messages (or pieces of messages) are forwarded from the input channel to the output channel of a switch.

The routing algorithm only determines **which** of the channels should be used

The switching strategy has a large influence on the **message transmission time**.

Message Transmission Between Neighboring Processors

Software steps of a typical communication protocol:

1. Program steps for sending a message:
 - 1.1 **Copy** the message into a system buffer.
 - 1.2 Calculate the **checksum**, attach **header**, start **timer**.
 - 1.3 Send the message to the network interface, start hardware-based transmission.
2. Program steps for receiving a message:
 - 2.1 Copy the message from the network interface to system buffer.
 - 2.2 Calculate and check the **checksum**; Confirm receipt in case of the correct checksum value.
3. Other steps on the sender side:
 - 3.1 Receive acknowledgement of the receiver → free the system buffer.
 - 3.2 Transmission time expired (timer) → retransmit the message.

Performance Measures for Message Transfers (1)

Bandwidth = maximum frequency at which data can be sent over the link.
Unit: bytes/second.

Byte Transfer Time = time required to transfer one byte:

$$\text{Byte Transfer Time} = \frac{1}{\text{Bandwidth}}$$

Transmission Time = time required to transmit the message over a network link:

$$\text{Transmission Time} = \frac{\text{Message Size}}{\text{Bandwidth}}$$

Time of Flight (Channel Propagation Delay) = time required for the **first bit** of the message to arrive at the receiver.

Performance Measures for Message Transfers (2)

Transport Latency = total time needed to transfer a message over a network link:

$$\text{Transport Latency} = \text{Time of Flight} + \text{Transmission Time}$$

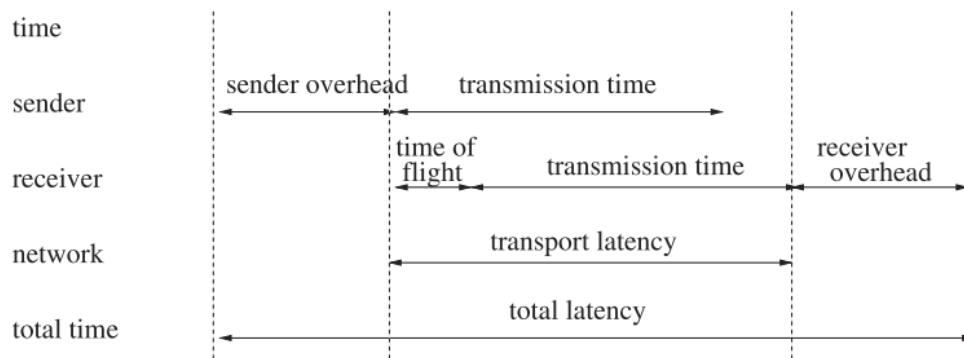
Sender Overhead (Startup Time) = time the sender needs to prepare the message.

Receiver Overhead = time the receiver needs for the software steps upon message receipt.

Throughput = effective network bandwidth that is achieved in a particular application.

Latency

$$\text{Latency} = \text{Sender Overhead} + \text{Time of Flight} + \frac{\text{Message Size}}{\text{Bandwidth}} + \text{Receiver Overhead}$$



Runtime Formula for Message Transmissions

Simplification:

$$\text{Latency} = \text{Overhead} + \frac{\text{Message Size}}{\text{Bandwidth}}$$

General notation:

$$T(m) = t_S + t_B \cdot m$$

where

m Message size in bytes

t_S Startup time

t_B Byte transfer time

Special Switching Strategies

If two nodes are **not** adjacent in a network, then a message between these nodes has to be sent over multiple connections using a path of nodes.

Basic types:

- ▶ Circuit switching
- ▶ Packet switching

Special types:

- ▶ Circuit switching.
- ▶ Packet switching with store-and-forward routing.
- ▶ Virtual cut-through routing.
- ▶ Wormhole routing.

Circuit Switching

- ▶ The **entire** path from the source node to the destination node is established.
- ▶ **Exclusive** assignment of the path to the message.
- ▶ The message is split into **p**hysic**s** (**physical units**).
- ▶ The entire path is freed after message transmission.

Costs:

$$T_{CS}(m, l) = t_S + t_c \cdot l + t_B \cdot m$$

where

m Message size in bytes

l Path length

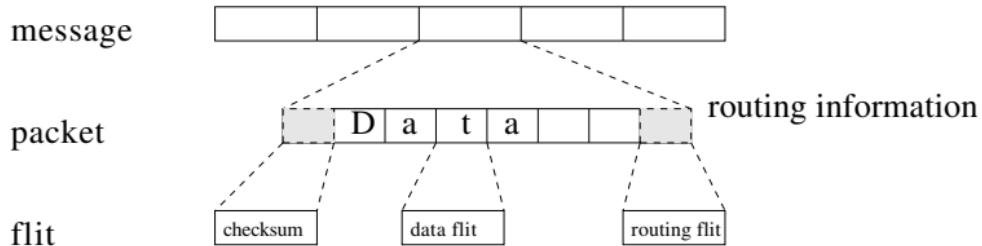
t_S Startup time

t_c Costs for sending the control message

t_B Byte transfer time

Packet Switching

- Messages are partitioned into **packets**.
- The packets are sent **separately** (over different paths).



Option: Store-and-forward routing

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

t_h = time spent at intermediate nodes

Option: Pipelining

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

m_p = packet size

Cut-through Switching

- ▶ Variation of Packet switching.
- ▶ Messages are partitioned into packets.
- ▶ Packets are sent separately using pipelining.

Costs:

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

t_H = costs of transferring the header over the network.

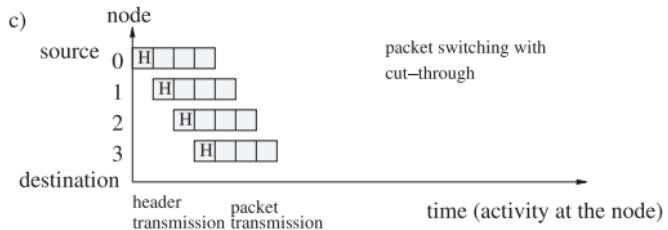
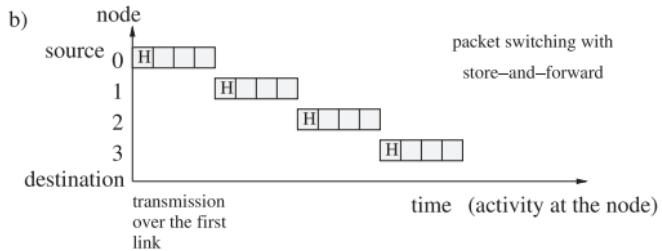
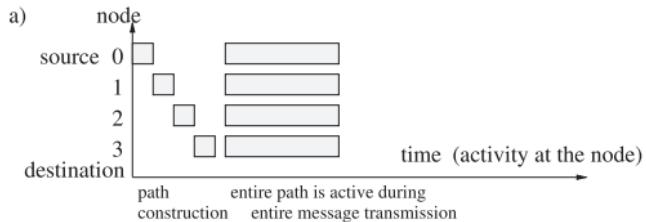
m_H - size of the header.

Option: Virtual cut-through routing

Option: Wormhole routing or hardware routing

Division of packets into flits (flow control units, between 1 and 8 bytes).

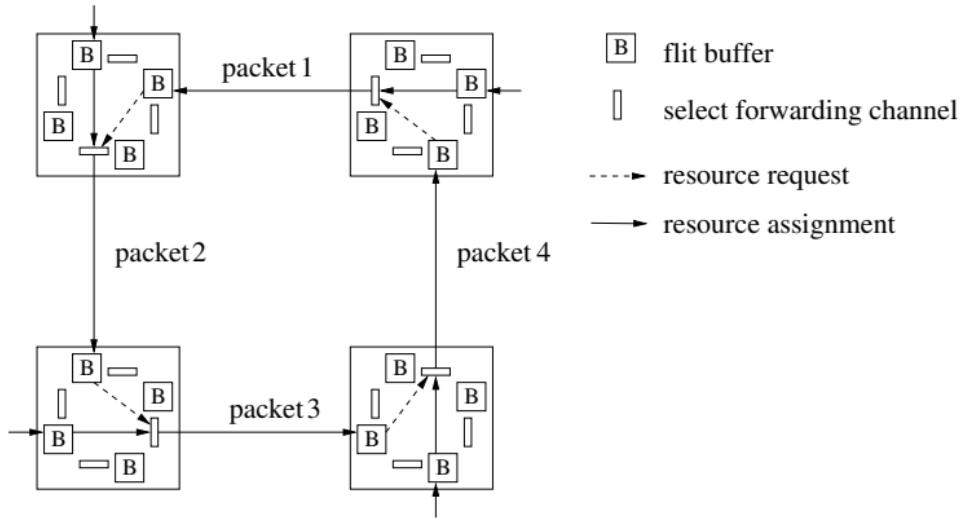
Transmission Costs for Different Switching Strategies



Wormhole Routing (Hardware Routing)

- ▶ Variation of cut-through routing.
- ▶ Division of the packets into small units **flits** (flow control units): typical size between 1 and 8 bytes.
- ▶ Header-flits define the path through the network.
- ▶ All other flits follow the **same** pipeline path.
- ▶ Intermediate buffers at the nodes can hold only a small number of flits.
- ▶ If the desired connection is not free:
 - ▶ Header blocks until the connection is free.
 - ▶ All following flits block and stay in their positions (different to cut-through routing).
- ▶ Advantage: low memory requirements for caching.
- ▶ Disadvantage: Deadlocks can occur resulting from a cyclic wait.

Wormhole Routing: Deadlock Situation



Overview

Overview

Introduction

Architecture of Parallel Systems

Parallel Programming Models

Message Passing Programming

Introduction to MPI-2

Realization and Execution Times Analysis of global Communication Operations

Parallel Programming Models

Models for Parallel Systems

Parallelization of Programs

Levels of Parallelism

Information Exchange

Models for Parallel Systems

Distinction according to level of abstraction

- ▶ **Parallel machine models:** lowest level of abstraction – hardware related description of the system
- ▶ **Parallel architectural models:** Abstraction of machine models – (topology, synchronous or asynchronous operation of the processor, SIMD or MIMD, memory organization)
- ▶ **Parallel computational models:** Extension of the architectural models, by which algorithms can be constructed and their costs can be considered, e.g. PRAM-Model (**p**arallel **r**andom **a**ccess **m**achine)
- ▶ **Parallel programming models:** Description of a parallel system by describing the programming language and -environment

Criteria for Parallel Programming Models

- ▶ What kind of parallelism from the computation can be used?
(instruction level parallelism, function level, parallel loops)
- ▶ Has the programmer to specify the parallelism and how is the parallelism specified? (explicit specification of parallelism or implicit specification of parallelism)
- ▶ In which way has the programmer to specify the parallelism? (e.g. independent tasks, managed by task pools or processes which are generated and which have to communicate to each other)
- ▶ How is the execution of the parallel units organized? (SIMD or SPMD, synchronous or asynchronous)
- ▶ How is the information exchange organized? (communication with messages or by using shared variables)
- ▶ What kind of synchronization can be used?

Parallel Programming Models

Models for Parallel Systems

Parallelization of Programs

Levels of Parallelism

Information Exchange

1. Decomposition of the computations

- ▶ Decomposition of the algorithm into tasks.
- ▶ Specification of task dependencies.
- ▶ Tasks include an unrestricted set of computations and access to shared variables (on shared memory systems) or they exchange messages by communication operations (on distributed memory systems)
- ▶ **Granularity** of a task: Number of computations performed by a task.

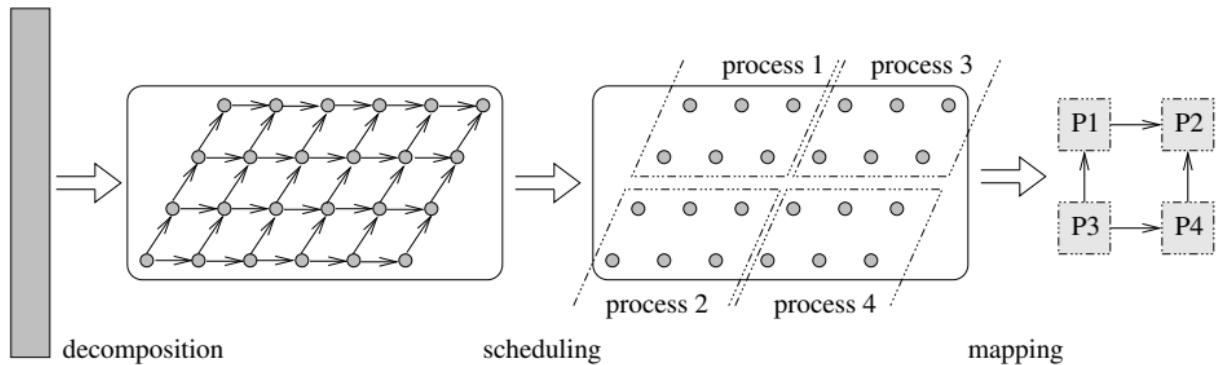
Program Parallelization (II)

2. Assignment of tasks to processes

- ▶ Processes execute tasks successively.
- ▶ The aim of the assignment of tasks to processes is to execute nearly the same number of computations by each process, such that a good **load balance** occurs.
- ▶ The assignment of tasks to processes is denoted as **scheduling**.

3. Mapping of processes to physical processors

Illustration: Parallelization of Programs



Parallel Programming Models

Models for Parallel Systems

Parallelization of Programs

Levels of Parallelism

Parallel Loops

Information Exchange

Levels of Parallelism

- ▶ Instruction parallelism
- ▶ Data parallelism
- ▶ Parallel loops
- ▶ Function parallelism

Parallel Programming Models

Models for Parallel Systems

Parallelization of Programs

Levels of Parallelism

Parallel Loops

Information Exchange

Parallel Loops

- ▶ **forall-Loops:**

each iteration uses the values of variables as they were before the execution of the forall-loops. The loop

```
forall (i = 1:n)
  a(i) = a(i-1) + a(i+1)
endforall
```

is equivalent to the vector-statement

$$a(1 : n) = a(0 : n - 1) + a(2 : n + 1)$$

of Fortran 90. If the forall-loops contains multiple assignment statements, they are executed one after another as vector-statements.

- ▶ Iterations of the dopar-loop are executed in parallel by different processes.

Each process executes all assigned iterations sequentially and uses the values of variables as they were before the execution of the dopar-loop.

Example: Parallel Loops

```
for (i=1:4)           forall (i=1:4)          dopar (i=1:4)
    a(i)=a(i)+1       a(i)=a(i)+1          a(i)=a(i)+1
    b(i)=a(i-1)+a(i+1) b(i)=a(i-1)+a(i+1) b(i)=a(i-1)+a(i+1)
endfor                  endforall            enddopar
```

initial values		after for-loop	after forall-loop	after dopar-loop
a(0)	1			
a(1)	2	b(1)	4	5
a(2)	3	b(2)	7	8
a(3)	4	b(3)	9	10
a(4)	5	b(4)	11	11
a(5)	6			10

Structuring parallel Programs

Creation of processes:

- ▶ Fork-Join-Construct:
A already existing process creates with the help of a fork-call a copy as a child process. Both copies execute the same program statements as long as both call a join statement.
- ▶ SPMD and SIMD:
All processes execute the same statements, which are applied to different data items. (SIMD - synchronously; SPMD -asynchronously)
- ▶ Master-Slave or Master-Worker: A process controls the whole work distribution of a program.
- ▶ Pipelining:
All processes are active at the same time and a stream of data is send from one process to the next process.

Parallel Programming Models

Models for Parallel Systems

Parallelization of Programs

Levels of Parallelism

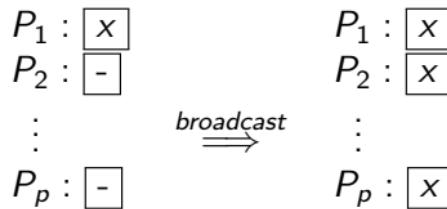
Information Exchange

Information Exchange

- ▶ The **information exchange** between processors of a parallel systems depends on the **organization of the memory subsystem**:
 - shared address space:** shared variables
 - distributed address space:** explicit communication operations
- ▶ **shared variables:** Concurrent accesses through several processor to the same address is protected by **synchronization operations**
 - **Sequentialization** of concurrent accesses
 - **Prevention of race conditions**
Simple synchronization by using (lock/unlock).
- ▶ **Communication operations:** Information exchange by **sending messages (message passing)**;
Differentiation between **point-to-point communication** and **global communication**

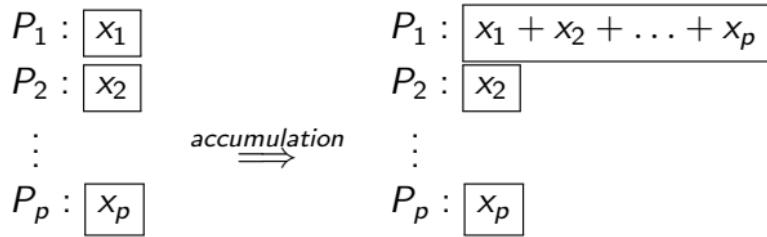
Overview over Communication Operations

- ▶ **Point-to-Point transfer:** A processor P_i (**sender**) sends a message to another processor P_j (**receiver**).
The **sender** executes a send operation (with the specification of a **send buffer**) and with the **identification number of the receiver**. The **receiver** executes a corresponding **receive operation** with the specification of an **receive buffer** and with the specification of the **identification number of the sender**.
- ▶ **Single Broadcast:** A specific processor P_i (**root**) sends the **same message** to **all** other processors. Depiction:



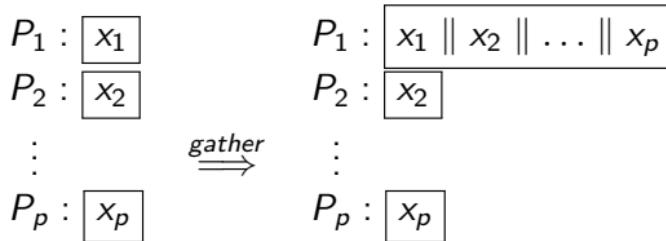
The **root processor** specifies the **send buffer** from which the broadcast message has to be send. All other processors specify the **receive buffer** in which the message is stored.

- ▶ **Single-Accumulation Operation:** Each process sends a message to a specific processor P_i (**root**) a message with data of the **same type**. The messages are combined elementwise with a specific **reduction operation**. → the result on the **root process** P_i is a single (composed) message. **Depiction:**



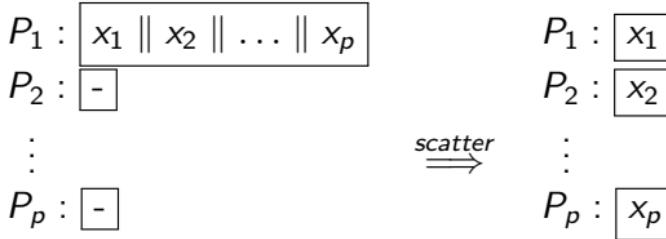
Each process specifies a **buffer** with the **data** to be combined and the **reduction operation** which to be used.

- ▶ **Gather:** Each process sends to a specific processor (**root**) a message. The **root processor** collects the messages without any reduction.
Depiction:

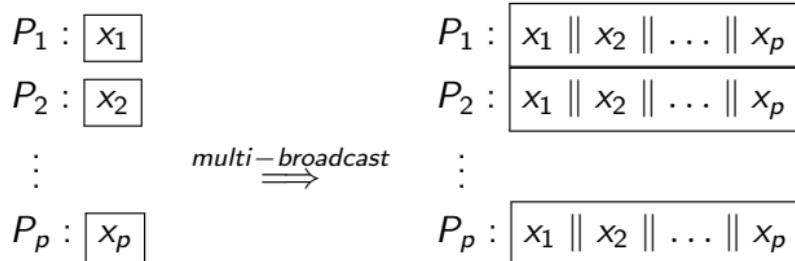


Each process specifies a **buffer** storing the data to be sent. The **root process** specifies a **additional** buffer for the collected messages.

- ▶ **Scatter:** A specific processor P_i (**root**) sends to the other processors a message, which might be different for each receiver. **Depiction:**



- ▶ **Multi-Broadcast Operation:** Each processor executes a single broadcast operation, i.e. **each processor** sends **each other processor** the **same message**. Contrary, each processor receives a message from each other processor, where the different receivers receive from one sender the same message. **Depiction:**



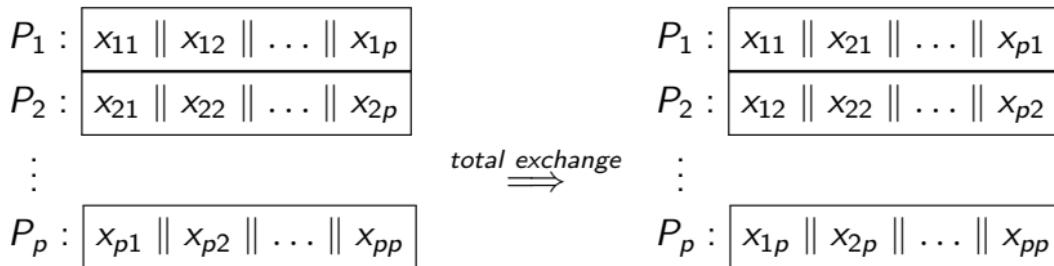
Note: There is no specific root process.

- ▶ **Multi-Accumulation Operation:** Each processor executes a single accumulation operation, i.e. **each process** provides for **each other process** a possible different message. The messages specific for each receiver are combined with a **reduction operation**, such that each receiver gets a combined message: **Depiction:**

$$\begin{array}{ll}
 P_1 : & \boxed{x_{11} \parallel x_{12} \parallel \dots \parallel x_{1p}} \\
 P_2 : & \boxed{x_{21} \parallel x_{22} \parallel \dots \parallel x_{2p}} \\
 & \vdots \\
 P_p : & \boxed{x_{p1} \parallel x_{p2} \parallel \dots \parallel x_{pp}}
 \end{array}
 \xrightarrow{\text{multi-accumulation}}
 \begin{array}{ll}
 P_1 : & \boxed{x_{11} + x_{21} + \dots + x_{p1}} \\
 P_2 : & \boxed{x_{12} + x_{22} + \dots + x_{p2}} \\
 & \vdots \\
 P_p : & \boxed{x_{1p} + x_{2p} + \dots + x_{pp}}
 \end{array}$$

Note: There is no specific root process.

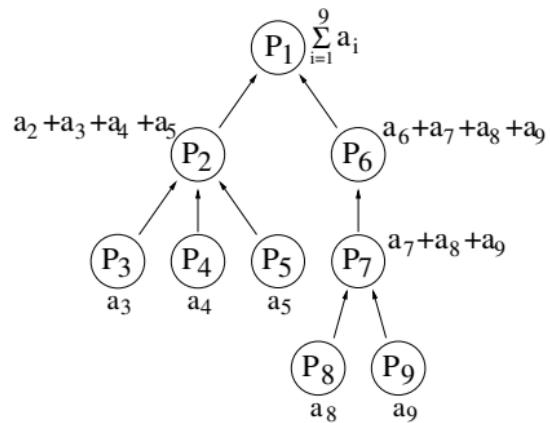
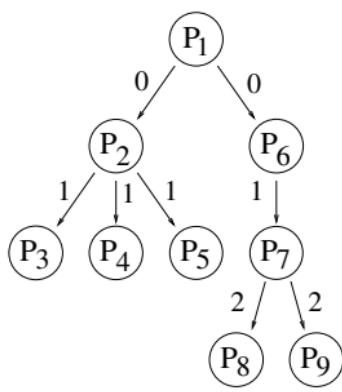
- ▶ **Total Exchange:** Each processor send to each other processor a possibly different message, without using a reduction operation, i.e. each processor executes a scatter operation. Contrary, each processor receives from each other processor a possibly different messages, i.e. each processor executes a gather operation. **Depiction:**



Note: There is no specific root process.

Duality of Communication Operations

E.g. the **broadcast operation** and the **reduction operation** can be realized with the same **spanning tree**:



→ **broadcast** and **reduction** are dual operations.

The **spanning tree** of a graph $G = (V, E)$ is the subgraph $G' = (V, E')$, which includes all vertices V and a subset of the edges $E' \subseteq E$ which represents a tree.

Additional dual operations are:

1. Scatter and Gather and 2. Multi-Broadcast and Multi-Accumulation

Hierarchy of Communication Operations

The communication operations result from a **stepwise specialization** from the most general operation (total exchange)
→ Representation as a Hierarchy possible:

