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

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)

Additional Lectures

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

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

Parallelization of algorithms

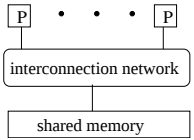
Necessary steps to parallelize an algorithm:

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

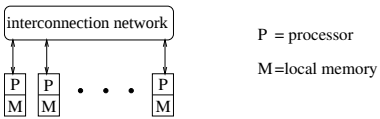
Distinction from the view of **memory** organization:

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

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;
  $\rightarrow$ 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.

Scalability

- 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

Processor Architecture and Technology Trends

- 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.
  $\rightsquigarrow$ 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
  $\rightarrow$ ILP (instruction level parallelism) processors
  $\rightarrow$ the different pipeline stages can work in parallel
  $\rightsquigarrow$ Reduction of the **average execution time** per instruction;

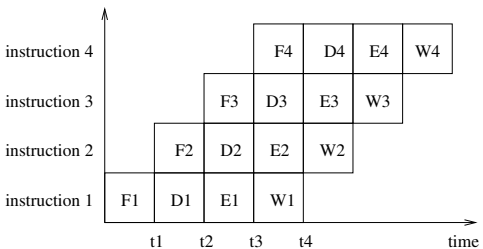
simple decomposition:

(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
  $\rightsquigarrow$ independent machine instructions can be executed in **parallel**
- parallelism at process or thread level $\rightarrow$ multicore processors
- Integration of **(multilevel) caches** onto the die of the processor
  $\rightsquigarrow$ Reduction of the **average memory access latency**;
  (reduces the Memory Wall Problem)
- VLIW (Very Long Instruction Word).

Limitations

(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** arrang independent instructions into long instruction words; $\rightsquigarrow$ the **parallel** processing of instructions is **explicitly** visible in the instruction words;
  $\rightsquigarrow$ 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;

Overview

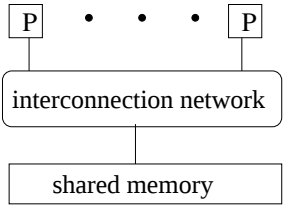
Flynn's Classification

- 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**

Overview

Shared Address Space: Multi-Processor Systems:

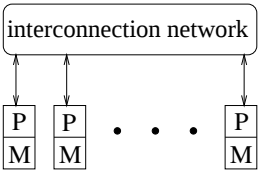
- Processors are connected to **global memory** via a network (**SMM - Shared Memory Machines**):



- **Data exchange** takes place through shared variables by means of coordinated reading and writing;
  $\rightarrow$ **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, irregardless 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**;
  $\rightarrow$ 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.
  $\rightsquigarrow$ memory access time does not keep pace with processor performance improvement
  $\rightsquigarrow$ 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
  $\rightarrow$ the **memory latency will be hidden** by executing instructions of other virtual processors
  $\rightarrow$ **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.

Overview

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. $\rightarrow$ The communication graph should be a connected graph.
- Most of the networks use **bidirectional** connections $\rightarrow$ 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 $(v_0, ..., v_k)$ 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:

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

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

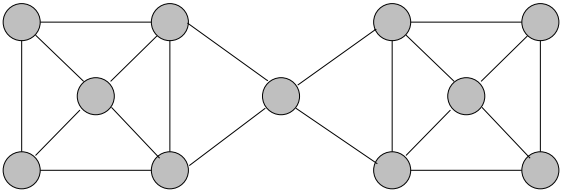
- **Edge connectivity: minimal number of edges** that must be removed to **break** the network:

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

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:**



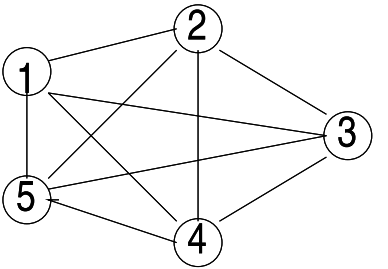
- Node connectivity $= 1$
- Edge connectivity $= 2$

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

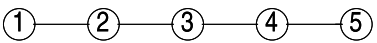
- **Example 1: Complete graph:** every node has a **direct** connection to every other node in the network.



  - **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

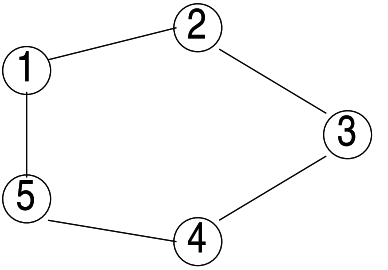
- **Example 2: Linear array:** linear ordering of processors, so that neighboring processors have a bidirectional connection: $V = \{v_1, ..., v_n\}, E = \{(v_i, v_{i+1}) | 1 \leq i < n\}$



  - **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

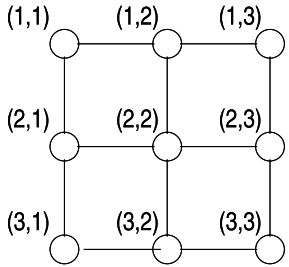
- **Example 3: Ring:** ring-like ordering of processors;
  compared to the linear array, an additional edge exists between the first and the last processor.



  - **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 $n = n_1 \cdot n_2 \cdot ... \cdot n_d$ nodes; n_j is the number of processors in dimension j:

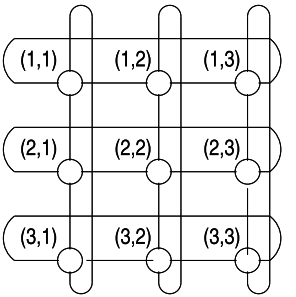


Nodes: $(x_1, ..., x_d)$ with $1 \leq x_j \leq n_j$ for $j = 1, ..., d$
An edge exists between node $(x_1, ..., x_d)$ and $(x'_1, ..., x'_d)$ if for a $\mu \in \{1, ..., d\}$ it is: $|x_\mu - x'_\mu| = 1$ and $x_j = x'_j$ for all $j \neq \mu$.
Special case: $n_j = r = \sqrt[d]{n}$ for all $j = 1, ..., d$.

- **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 $(x_1, ..., x_{j-1}, 1, x_{j+1}, ..., x_d)$ and $(x_1, ..., x_{j-1}, n_j, x_{j+1}, ..., x_d)$.

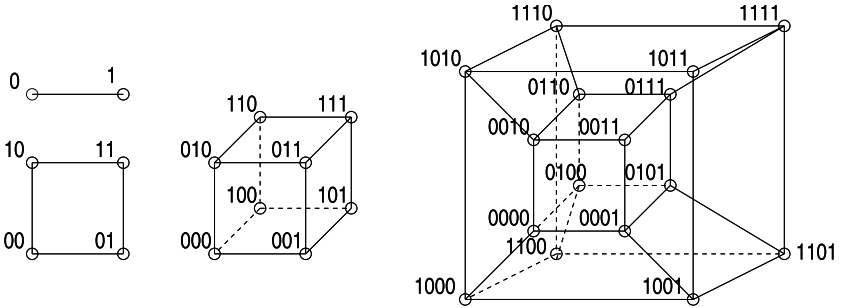


Special case: $n_j = r = \sqrt[d]{n}$ for all $j = 1, ..., d$.

- **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 $\alpha_0 ... \alpha_j ... \alpha_{k-1} \rightarrow$ node enumeration.

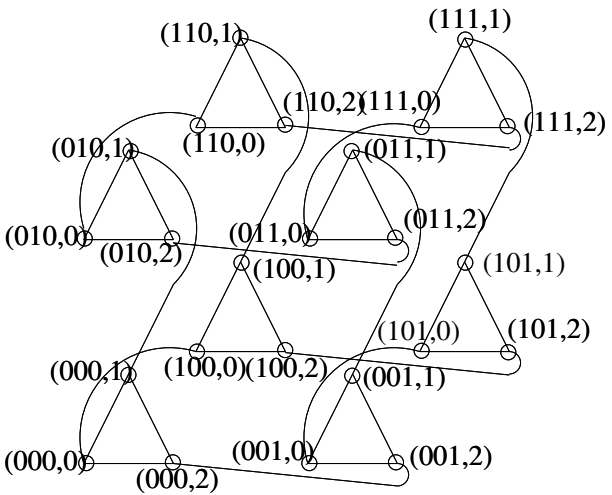
nodes: $V = \{0,1\}^k \rightarrow n = 2^k$ nodes.
Edges: an edge exists between node $\alpha_0 ... \alpha_j ... \alpha_{k-1}$ and node $\alpha_0 ... \bar{\alpha_j} ... \alpha_{k-1}$ for $0 \leq j \leq k-1$
$\rightarrow$ **neighboring** nodes differ in exactly one bit;
i.e., $\bar{\alpha_j} = 1$ for $\alpha_j = 0$ 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).

CCC Network

- **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: $V = \{0,1\}^k \times \{0,...,k-1\}$
  $\{0,1\}^k$ binary representation of the original hypercube node.
  $i \in \{0,...,k-1\}$ position within the ring.
- **Edge set:** The set of edges can be partitioned into (new) **ring edges** and **hypercube edges**:
  Ring edges:

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

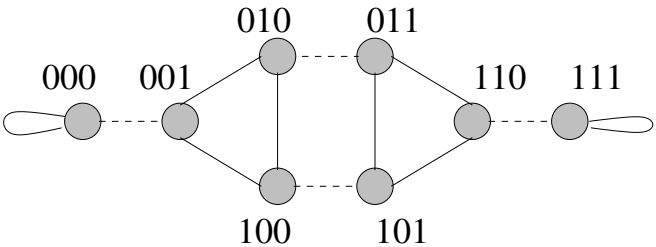
  Hypercube edges:

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

- **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.

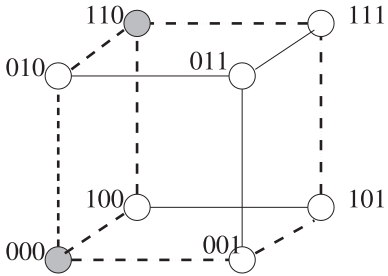
Connectivity of a k-dimensional Hypercube (2)

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, $i \in \{0, ..., l-1\}$:
    - Start with bit name A of v
    - Invert the bits $i, ..., l-1$ stepwise
    - Invert the bits $0, ..., i-1$ 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 $0, ..., l-1$ 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 $\sigma : V' \to V$, 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 $V' = \{1, ..., n\}$ 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)

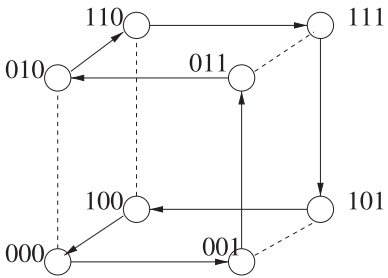
Recursive Definition of the reflected gray code:

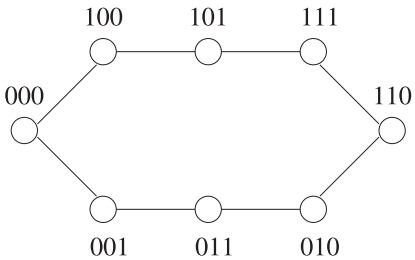
- The 1-bit Gray Code is $RGC_1 = (0, 1)$.
- The 2-bit Gray Code is obtained from RGC_1 by inserting a 0 and a 1 in front of RGC_1.
  The two resulting sequences (00, 01) and (10, 11) are concatenated after reversal of the second sequence.
  The resulting sequence is $RGC_2 = (00, 01, 11, 10)$.
- The k-bit Gray Code RGC_k for $k \geq 2$ is constructed from the $(k-1)$-bit Gray Code $RGC_{k-1} = (b_1, ..., b_m)$ 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 $(0b_1, ..., 0b_m)$ and $(1b_1, ..., 1b_m)$.
  RGC_k is constructed by a reversal of the second sequence and concatenation of the two sequences:

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

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

- **Example for $k = 3$:**





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

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

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

Embedding a 2D Mesh into a k-dimensional Hyercube (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 $RGC_{k_1} = (a_1, ..., a_{n_1})$ and Gray Code $RGC_{k_2} = (b_1, ..., b_{n_2})$ are used to construct a $n_1 \times n_2$ matrix M whose entries are k-bit strings.
  The matrix is of the form $M(i,j) = \{a_i, b_j\}_{i=1,...,n_1, j=1,...,n_2}$:

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

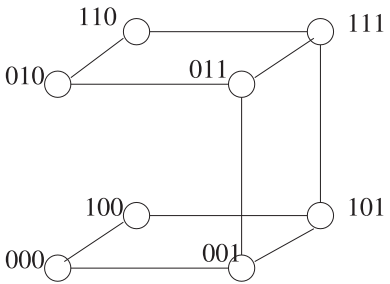
- **Properties**:
  - Adjacent nodes of matrix M differ in exactly one bit position.
  - All elements of M are different bit words of length k.

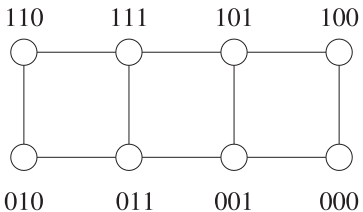
Embedding a 2D Mesh into a k-dimensional Hyercube (2)

- Embedding function:

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

- **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 $n = n_1 \cdot ... \cdot n_d$ nodes can be represented by its mesh coordinates $(x_1, ..., x_d)$, $1 \leq x_i \leq n_i$.
- The **mapping**

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

$$\text{with } \sigma((x_1, ..., x_d)) = s_1 s_2 ... s_d \text{ and } s_i = RGC_{k_i}(x_i)$$

  (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 $(x_1, ..., x_d)$ and $(y_1, ..., y_d)$ that are connected by an edge in the mesh, there exists exactly one dimension $i \in \{1, ..., d\}$ with $|x_i - y_i| = 1$. For all other dimensions $j \neq i$ it is $x_j = y_j$.
- For the corresponding hypercube nodes $\sigma((x_1, ..., x_d)) = s_1 s_2 ... s_d$ and $\sigma((y_1, ..., y_d)) = t_1 t_2 ... t_d$, 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.

  $\rightsquigarrow$ The nodes $s_1 s_2 ... s_d$ and $t_1 t_2 ... t_d$ are connected by an edge in the k-dimensional cube.

Dynamic Networks

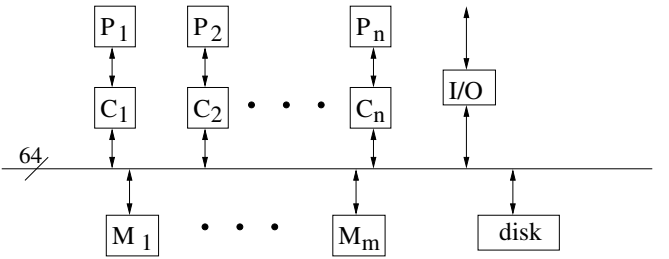
- Dynamic networks consist of a number of physical **wires** and **switches** in between.
- Processors and memory modules are connected indirectly via the network (and the switches).
- The switches are **dynamically configured** according to the requirements of the message transmission.
- Dynamic networks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

Topological structures of dynamic networks:

- Bus networks
- Crossbar networks
- Multistage switch networks

Bus Networks

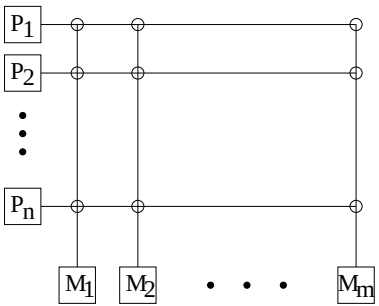
- A bus network consists of **a set of wires** which can be used to transport data from a sender to a receiver.
- **Example:** Bus with 64 bit wires to connect processors $P_1, ..., P_n$ with caches $C_1, ..., C_n$ to memory modules $M_1, ..., M_m$:



- 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)

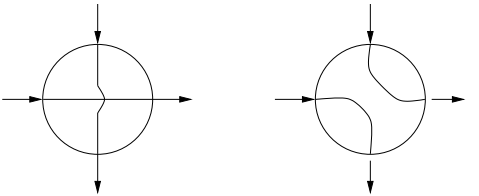
- An $n \times m$ crossbar network has n **inputs**, m **outputs** and $n \cdot m$ switches.
- **Example:** n processors $P_1, ..., P_n$, m memory modules $M_1, ..., M_m$:



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

Crossbar Networks (2)

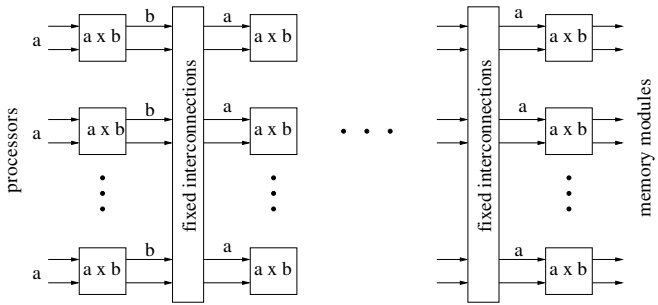
- **Possible switch positions** according to the requirements of data transfer or memory requests:



- At each point in time, each memory module can fulfill **only one memory request**;
  $\rightarrow$ 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)

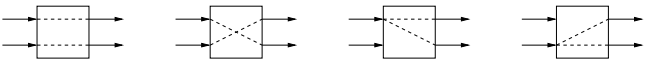
Graph representation:

- 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 $\pi : \{1, ..., n\} \rightarrow \{1, ..., n\}$, such that the output links of stage i are connected to the input links $(\pi(1), ..., \pi(n))$ 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:



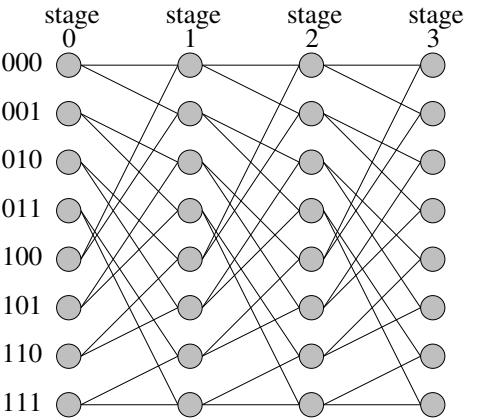
Examples: Omega-, baseline- or butterfly network.

Omega Network (1)

- An $n \times n$ omega network consists of $\log n$ stages with $n/2$ switches each (2×2 crossbar switches)
  $\rightarrow (n/2) \log n$ 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;
  - $i \in \{0, ..., \log n - 1\}$ 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).

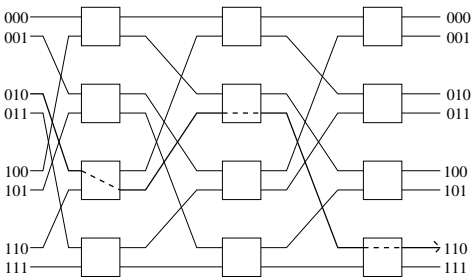
Omega Network (2)

- **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, $k = 0, ..., \log n - 1$ 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**:

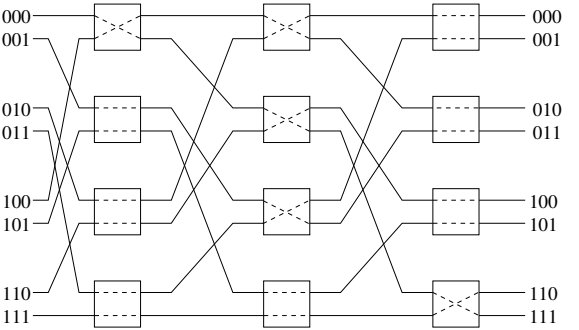


Routing in the Omega network (2)

- **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 $\pi^8 = \{0, ..., n-1\} \rightarrow \{0, ..., n-1\}$ **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**.

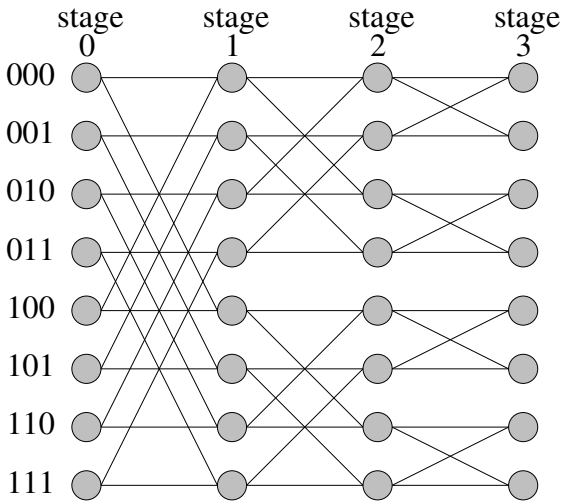
- Conflicts in blocking networks can be resolved through multiple transmissions over the network.

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.
  $\rightarrow (k + 1)2^k$ switches overall.
- The switches are **identified** by pairs (α, i) with:
  - $i \in \{0, ..., k\}$ 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:

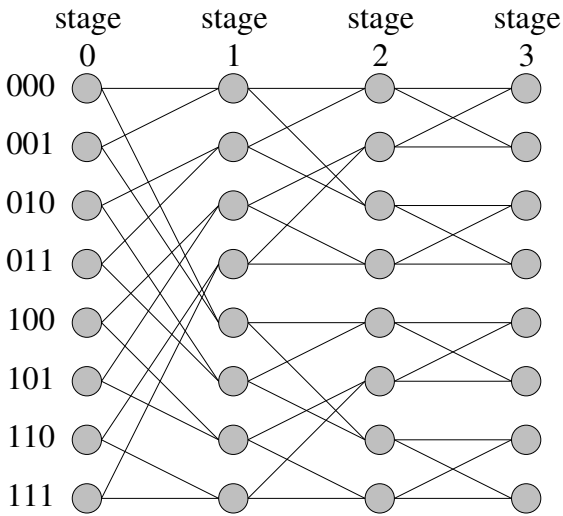


Baseline Network (1)

- 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.

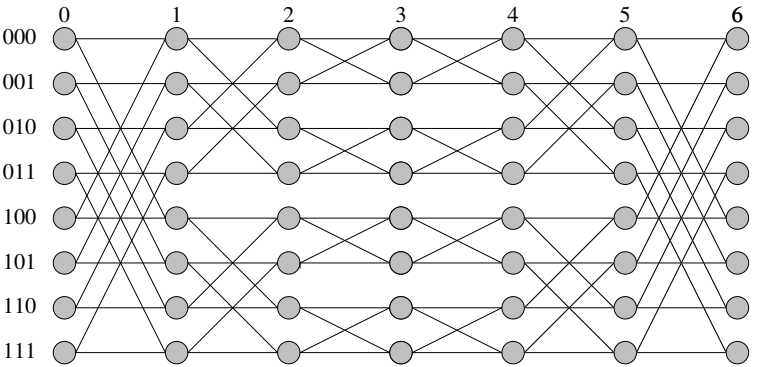
Baseline Network (2)

- **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.
    $\rightarrow 2k+1$ stages with $N = 2^k$ nodes/switches per stage.
- **Example:** Beneš network with 8 inputs:

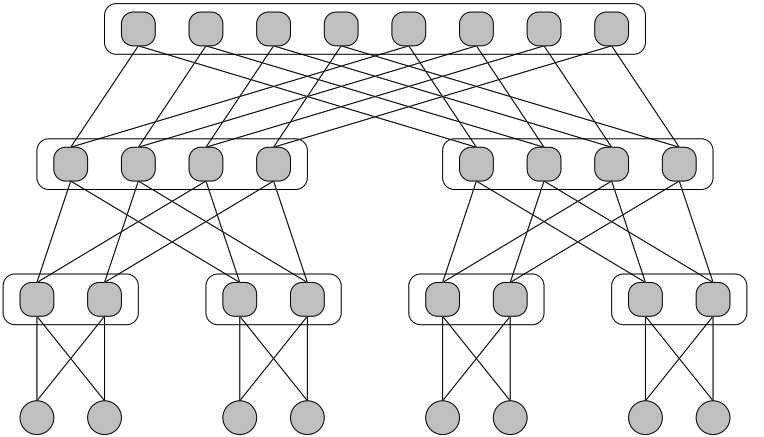


Fat Tree Network (Dynamic Tree Network) (1)

- 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, $i = 1, ..., \log n$:
  - Each switch has 2^i **input links** and 2^i **output links**, $i = 1, ..., \log n$.
  - Internal realization, e.g. from 2^{i-1} switches with two input and output links each.
    $\rightarrow$ 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):



Overview

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 $\rightarrow$ message delay).
- **Congestion** avoidance (too many messages accessing limited resources $\rightarrow$ 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 $\rightarrow$ impacts communication time.
  - **Non-minimal** routing algorithms: Choice of a non-deterministic path based on observation of network utilization.

Deterministic Routing Algorithms

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: $\alpha = \alpha_0, ..., \alpha_{k-1}$.
- Node B is expressed as: $\beta = \beta_0, ..., \beta_{k-1}$.
- Message should be sent from Sender A to Receiver B.
- Path used for sending the message:
  $A = A_0, A_1, ..., A_i = B$

- 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 $A = A_0, A_1, ..., A_i = B$, from which the successive one should be chosen.
Bit representation of A_i : $\gamma = \gamma_0, ..., \gamma_{k-1}$.

- 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 $\delta = \delta_0, ..., \delta_{k-1}$,
    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 I_2 is dependent on communication channel I_1 if it is possible within the routing algorithm to choose a path which sends the message on I_1 and then **directly** to I_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 $i+1$, $i = 0, ..., n_x - 2$ for each y-position.

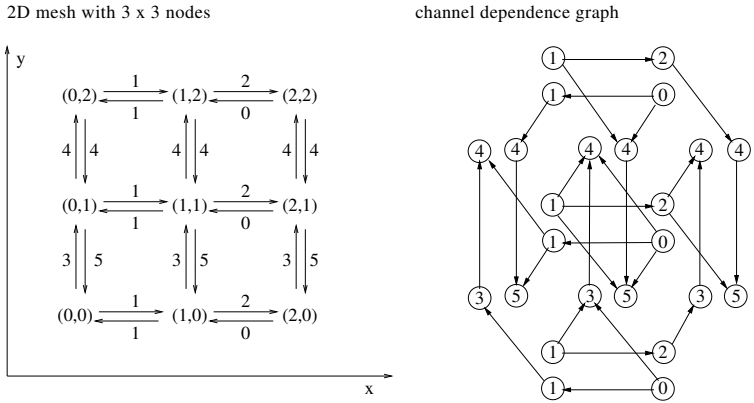
  The opposite edges of nodes $(i+1, y)$ to nodes (i, y)
  get a number $n_x - 1 - (i+1) = n_x - i - 2$, $i = 0, ..., n_x - 2$.

  (The edges in ascending x-direction ascend with $1, ..., n_x - 1$, the edges in descending direction of x are numbered $0, ..., n_x - 2$)
- **Vertical edges:**
  Edges between nodes (x, j) and $(x, j+1)$
  get a number $j + n_x$, $j = 0, ..., n_y - 2$

  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.



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 $N_1, ..., N_k$ and network edges $n_1, ..., n_k$ 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:

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

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 $a_0, ..., a_{n-1}$ 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 occurence of cycles.
- Examples:
  - West-First routing for a 2-dimensional mesh
  - P-cube routing for an n-dimensional hypercube

Turn Model

Motivation

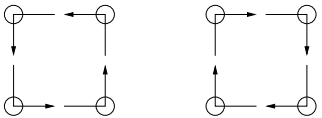
Deadlocks occur if messages change their direction of transmission in such a way that a cyclic wait results.
Deadlocks can be prevented by prohibiting certain direction changes.

Example

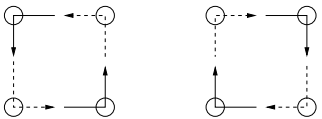
- *XY*-routing prohibits all turns from vertical to a horizontal direction.
- As a result, only 4 of the 8 possible turns in a 2-dimensional mesh are allowed.
- These four possible direction changes:
  - Allow **no loops**, making deadlocks impossible
  - Make adaptive routing impossible.

Turns for a Two-dimensional Mesh

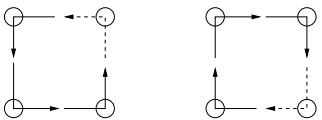
possible turns in a 2D mesh



turns allowed for XY-Routing



turns allowed for West-First-Routing



turns allowed:

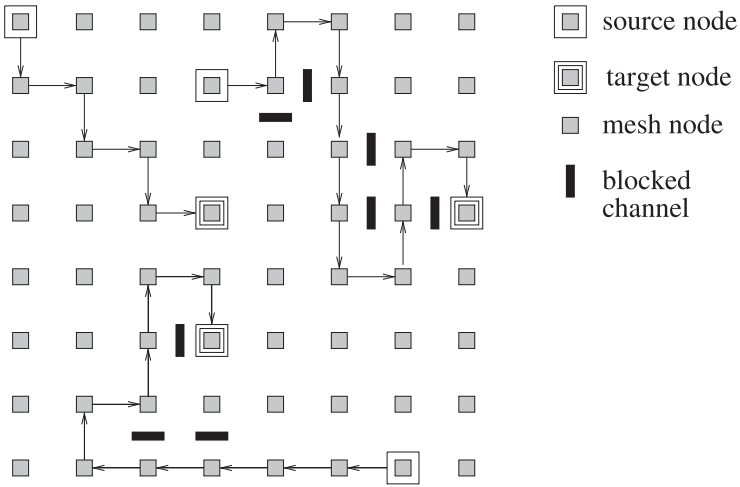
turns not allowed:

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

$$E = \{i \,|\, \alpha_i \neq \beta_i, i = 0, ..., n-1\}$$

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

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

- 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 preformed 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.

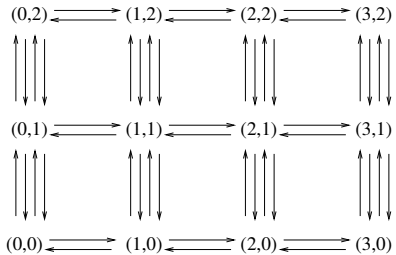
The 2-dimensional mesh is split up into two sub-networks, the $+X$-subnet and $-X$-subnet.

- 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.

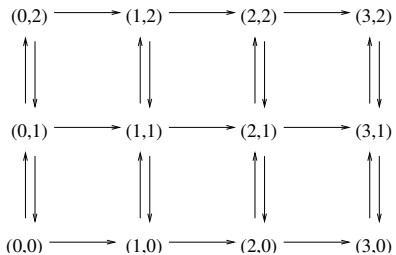
This minimal adaptive routing algorithm is deadlock-free.

Virtual Channels (3)

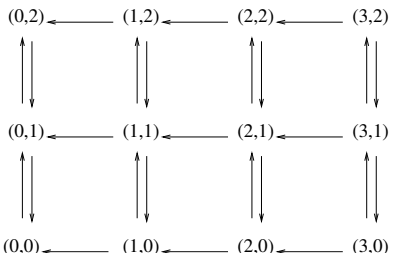
2D mesh with virtual channels in y direction



+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 ($k = 0, ..., \log n - 1$) uses the k-th bit β_k (from the left) of the target name β and chooses the according output channel using these rules:

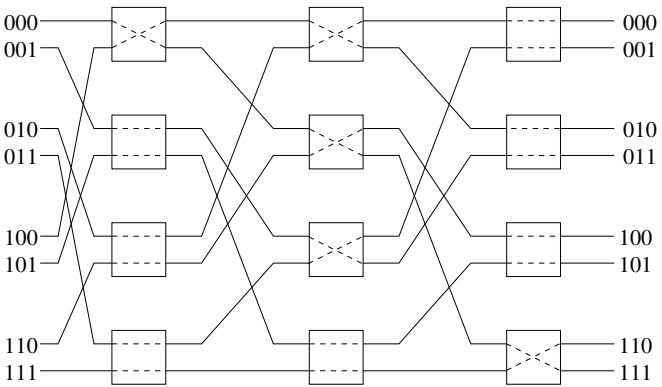
- If the k-th bit is $\beta_k = 0$ then the message will be forwarded over the upper link of the switch.
- If the k-th bit is $\beta_k = 1$ then the message will be forwarded over the lower line of the switch.

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)

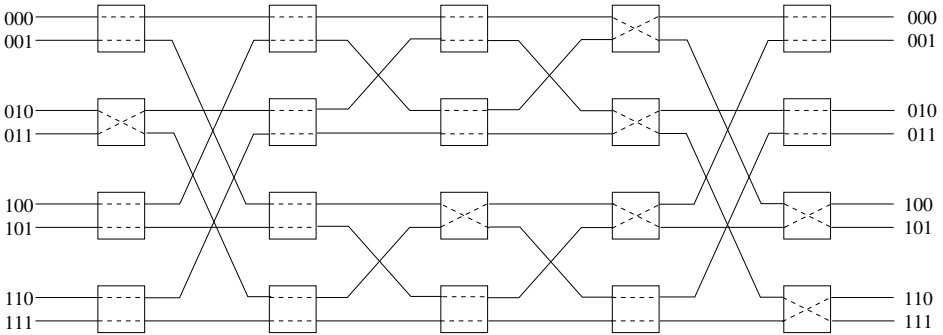
- Note: Many transmissions given through a permutation $\pi^8 = \{0, ..., n-1\} \rightarrow \{0, ..., n-1\}$ **cannot be performed in one step** (i.e., concurrently to each other), since there may be **conflicts** in the network.
  Example: The two message transfers from $\alpha_1 = 010$ to $\beta_1 = 110$ 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 $\rightarrow$ free the system buffer.
   - 3.2 Transmission time expired (timer) $\rightarrow$ 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:

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

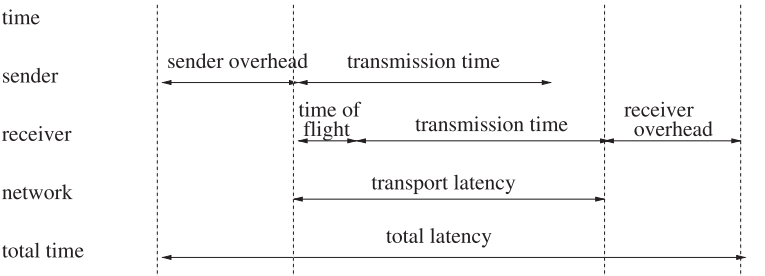
Sender Overhead (Startup Time) = time the sender needs to prepare the message.

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 **phits** (**physical units**).
- The entire path is freed after message transmission.

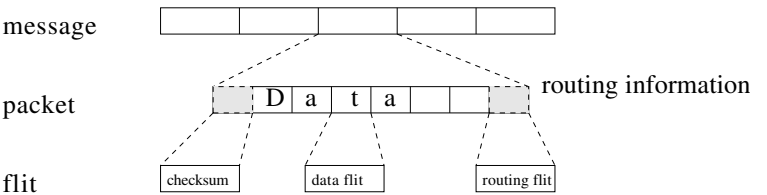
Costs:

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

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

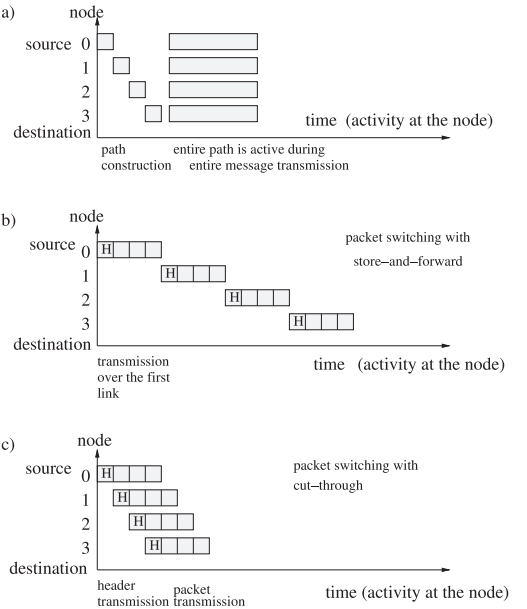
- Variation of Packet switching.
- Messages are partitioned into packets.
- Packets are sent separately using pipelining.

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).

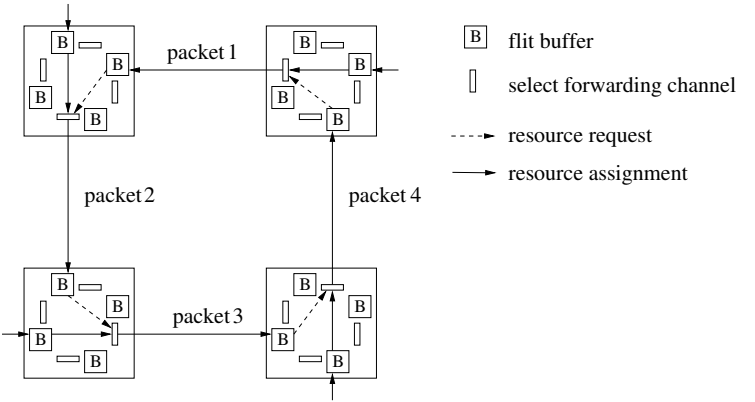
Transmission Costs for Different Switching Strategies



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

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**radom **a**ccess **m**machine)
- **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?

Overview

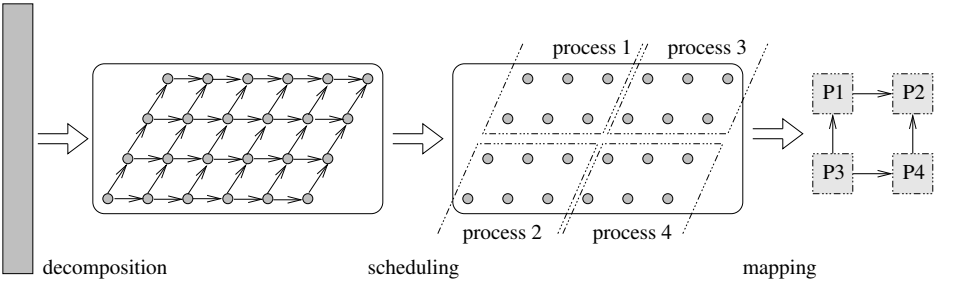
Parallelization of Programs

1. **Decomposition of the computations**
   - Decomposition of the algorithm into tasks.
   - Specification of task dependencies.
   - Tasks include a 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



Overview

Levels of Parallelism

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

Overview

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

$$\mathtt{a}(\mathtt{1}:\mathtt{n}) = \mathtt{a}(\mathtt{0}:\mathtt{n}-\mathtt{1}) + \mathtt{a}(\mathtt{2}:\mathtt{n}+\mathtt{1})$$

  of Fortran 90. If the `forall`-loops contains multiple assignment statements, they are executed on 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 where before the execution of the `dopar`-loop.

Example: Parallel Loops

```
for (i=1:4)
  a(i)=a(i)+1
  b(i)=a(i-1)+a(i+1)
endfor
```

```
forall (i=1:4)
  a(i)=a(i)+1
  b(i)=a(i-1)+a(i+1)
endforall
```

```
dopar (i=1:4)
  a(i)=a(i)+1
  b(i)=a(i-1)+a(i+1)
enddopar
```

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

Structuring parallel Programs

Creation of processes:

- Fork-Join-Construct:
  A already existinc process creates with the help of a fork-call a copy as a child process. Both copes 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.

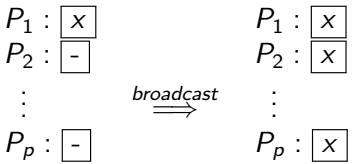
Overview

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**
  $\rightarrow$ **Sequentialization** of concurrent accesses
  $\rightarrow$ **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**

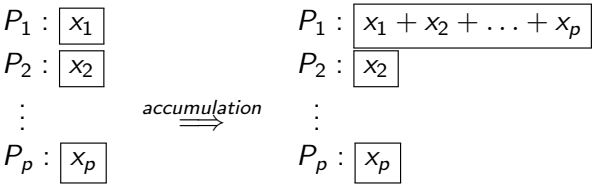
Overview over Communication Operations

- **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:



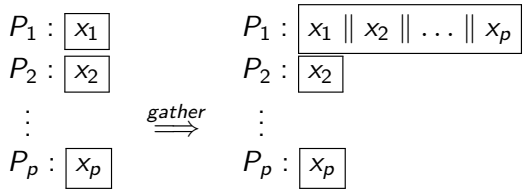
The **root processor** specifies the **send buffer** from which the broadcast message has to be send. All other processors specify the **receive buffer** in which the message is stored.

- **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**. $\rightarrow$ the result on the **root process** P_i is a single (composed) message. **Depiction:**



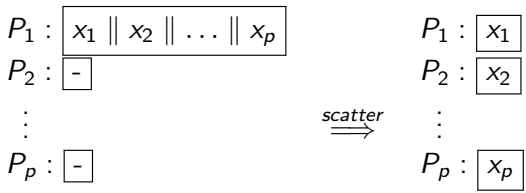
Each process specifies a **buffer** with the **data** to be combined and the **reduction operation** which to be used.

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

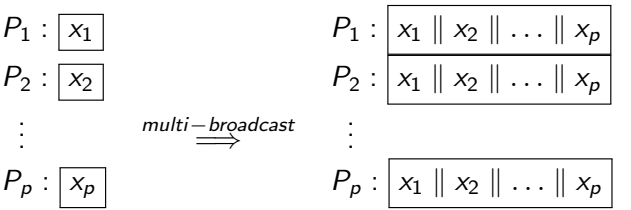


  Each process specifies a **buffer** storing the data to be sent. The **root process** specifies a **additional** buffer for the collected messages.

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

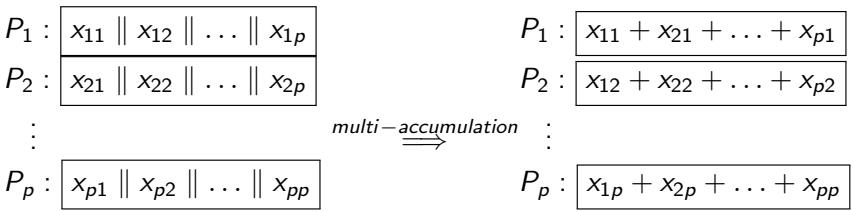


- **Multi-Broadcast Operation:** Each processor executes a single broadcast operation, i.e. **each processor** sends **each other processor** the **same message**. Contrary, each processor receives a message from each other processor, where the different receivers receive from one sender the same message.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



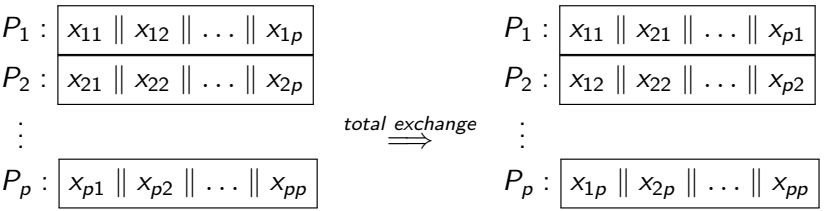
Note: There is no specific root process.

- **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:**



Note: There is no specific root process.

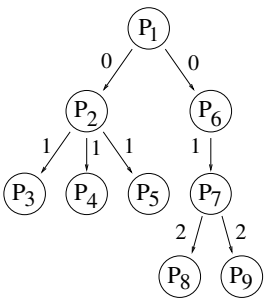
- **Total Exchange: Each processor** send to **each other processor** a possibly **different message**, without using a reduction operation, i.e. each processor executes a scatter operation. Contrary, each processor receives from each other processor a possibly different messages, i.e. each processor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

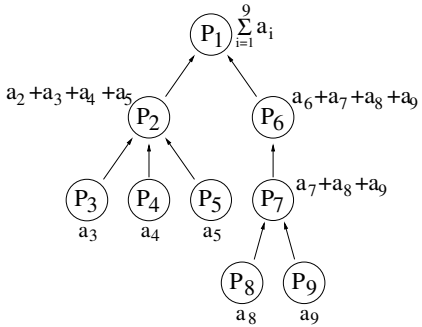


Note: There is no specific root process.

Duality of Communication Operations

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





$\rightarrow$ **broadcast** and **reduction** are dual operations.

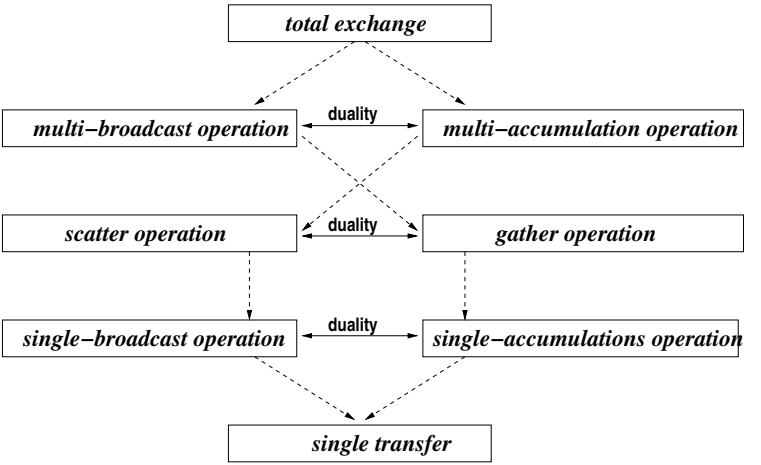
The **spanning tree** of a graph $G = (V, E)$ is the subgraph $G' = (V, E')$, which includes all vertices V und a subset of the edges $E' \subseteq E$ which represents a tree.

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)
$\rightarrow$ Representation as a Hierarchy possible:



Overview

Message Passing Programming

- A message passing program is executed by **multiple processes**.
  Each of these processes can execute a **different program** (MPMD).
  Simplification: Each process executes the **same program**.
  (**SPMD - single program multiple data**).
- Each process has a **separate address space**. Processes can exchange data by **sending messages**.
  Basic exchange operation: send/receive.
- Messages are sent using **explicit communication operations**.
- Communication libraries providing standardized interfaces are used.
  PVM (Parallel Virtual Machine), **MPI** (Message Passing Interface)
  $\rightsquigarrow$ portable programs

Overview

MPI – Message Passing Interface – Introduction

- MPI interfaces are available for C, FORTRAN 77 and C++ (MPI-2)
  In the following, we concentrate on the interface for C;
  further information: `www.mpi-forum.org`
- An MPI program consists of a **fixed number of processes**.
  The number of processes is determined at program start.
- Each processor executes a single process of the program.
- Program execution is based on the **SPMD model**;
  Processes are distinguished using the unique process number that is assigned at program start.
- MPI supports the following **communication operations**:
  - **point-to-point communication operations** to exchange data between two processes;
  - **global communication operations** to exchange data between multiple processes;
    global/collective communication operations

MPI – Classification of the Communication Operations

- Distinction on the basis of the **local view** of the processes:
  - **Blocking operation:** Control returns to the calling process not until all resources used by the sender/receiver to execute the operation are freed;
    $\rightarrow$ Resources can be reused immediately.
  - **Non-blocking operation:** Control may return to the calling process before the completion of the operation;
    $\rightarrow$ Resources used for the operation (e.g. buffers) may not be manipulated before the operation has signaled its completion to the calling process by setting appropriate flags.
- Global communication operations are always blocking in MPI.
- Distinction on the basis of the **global view** of the processes:
  - **Synchronous communication:** Transmission of the message only takes place, if sender and receiver participate at the same time at the communication.
  - **Asynchronous communication:** Sender can transmit data without ensuring that the receiver is ready to receive the data.

Starting and Initializing an MPI Program

- Execute an MPI program using the command:
  mpirun -np 4 <program name> <arguments>
- Initialization of the MPI runtime library:
  MPI_Init (&argv, &argc)
- Obtaining the local process number:
  MPI_Comm_rank (MPI_Comm, &num)
- Obtaining the total number of processes of the program:
  MPI_Comm_size (MPI_Comm, &count)
- Processes are combined into groups to coordinate the participation in the communication operations executed.
  The **processes of a group** can exchange data using **communicators**.
  Each **communication operation** requires a **communicator** as a parameter.
  The default communicator *MPI_COMM_WORLD* comprises all processes.

Overview

Sending Data using MPI Point-to-Point Communication

Basic form of a data exchange: operation of the **sender** in MPI:

```
int MPI_Send (void *smessage,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm)
```

- **smessage** Send buffer containing the elements to be sent successively;
- **count** Number of elements to be sent;
- **datatype** Data type common to all elements to be sent;
- **dest** Rank of the target process that should receive the data;
- **tag** Additional message tag (between 0 and 32767) to distinguish different messages of the same sender;
- **comm** Communicator of the underlying processor group;
- Data types available in MPI:
  - *MPI_{CHAR, SHORT, INT, LONG, BYTE, UNSIGNED, FLOAT, DOUBLE}*
  - *MPI_UNSIGNED_{CHAR, SHORT, LONG}*, *MPI_LONG_{DOUBLE}*, *MPI_PACKED*

Receiving Data with MPI Point-to-Point Communication (1)

The following point-to-point receive operation corresponds to `MPI_Send`:

```
int MPI_Recv (void *rmessage,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Status *status)
```

- **rmessage** Buffer of adequate size to receive the message;
- **count** Upper limit of the number of elements to accept;
- **datatype** Data type of the elements to be received;
- **source** Rank of the process from which to receive a message;
- **tag** Message tag of the message to be received;
- **comm** Communicator of the underlying processor group;
- **status** Data structure to be filled with information on the message received;

`MPI_Send()` and `MPI_Recv()` are **blocking** and **asynchronous** operations.

Receiving Data with MPI Point-to-Point Communication (2)

- Arbitrary messages can be received using:
  - **source**=*MPI_ANY_SOURCE* to receive a message from an arbitrary sender;
  - **tag**=*MPI_ANY_TAG* to receive a message with an arbitrary tag.
  - The sender/tag of the message received can be obtained by inspecting **status**.*MPI_*{*SOURCE, TAG*} after completion of the receive operation.
- The number of data elements transmitted to the receiver can be obtained from the data structure `status`:

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count_ptr)
```

  - **status** Pointer to the data structure returned by the corresponding call to `MPI_Recv()`.
  - **count_ptr** Address of a variable in which the number of elements received are returned.

Example: Message Passing from Process 0 to Process 1

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
   int my_rank, p, source, dest, tag=0;
   char msg [20];
   MPI_Status status;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
   if (my_rank == 0) {
      strcpy (msg, "Hello");
      MPI_Send (msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
   }
   if (my_rank == 1)
      MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Finalize ();
}
```

Implementation of Message Transfers in MPI

Internally a message transfer is implemented in **three steps**:

1. The data elements are copied from the **send buffer** `smessage` into a **system buffer** and the message to be sent is **assembled** by adding a **header** with information on the **sender** and the **receiver** of the message, the **tag**, and the **communicator**.
2. The message is sent via the **network** of the parallel platform **from the sender to the receiver**.
3. The **receiver** `dest` copies the data elements from the **system buffer** used to receive the message into the **receive buffer** `rmessage`.

Semantics of Blocking, Asynchronous Communication Operations

- An `MPI_Recv()` operation can also be **started** when the corresponding `MPI_Send()` operation has **not yet been started**.
  The `MPI_Recv()` operation **blocks** until the **receive buffer** specified contains the data elements sent.
- An `MPI_Send()` operation can also be **started** when the corresponding `MPI_Recv()` operation has **not yet been started**.
  The `MPI_Send()` operation **blocks** until the **send buffer** specified can be **reused**, i.e., new data elements can be copied to the buffer.

Implementation Variants of Blocking MPI Operations

a) Direct copy without a system buffer
The message is copied **directly** without using an intermediate **system buffer** into the **receive buffer** of another process.
→ The `MPI_Send()` operation is **blocked** until the **entire** message has been copied into the **receive buffer**.
In particular, control returns not until the corresponding `MPI_Recv()` operation has been **started**.

b) Intermediate buffering with a system buffer
The sender first copies the message to an internal **system buffer** .
The **sending process** can continue its operation as soon as the **copy operation** of the **sending side** is completed
(even before the start of the corresponding `MPI_Recv()` operation).
Advantage: the sender is **blocked** only for a short period of time;
Drawback: additional memory space for the system buffer;
additional execution time for copying data into the system buffer.

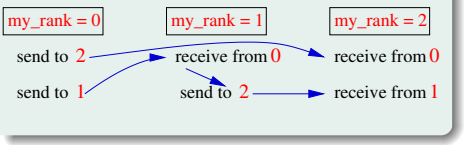
Order of MPI Communication Operations

- Each MPI implementation guarantees that messages of the same process do **not overtake** each other:
  If a sender sends two messages **one after another** to the **same receiver** and **both messages fit** to the first `MPI_Recv()` called by the receiver, the MPI runtime system ensures that the **first message sent will always be received first**.
- The involvement of a **third process** may disturb the order and may lead to a **violation** of the delivery order desired.

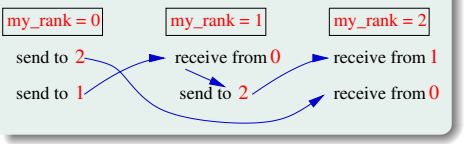
Example: Violation of the Delivery Order

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0 ) {
  MPI_Send(sendbuf1, count, MPI_INT, 2, tag, comm);
  MPI_Send(sendbuf2, count, MPI_INT, 1, tag, comm); }
else if (my_rank == 1) {
MPI_Recv(recvbuf1, count, MPI_INT, 0, tag, comm, &status);
MPI_Send(recvbuf1, count, MPI_INT, 2, tag, comm); }
else if (my_rank == 2) {
MPI_Recv(recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
MPI_Recv(recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
}
```

Expected delivery order:



Reversed delivery order:



Overview

Deadlocks with Blocking Point-to-Point Communication

- Careless usage of send and receive operation may lead to **deadlocks**.
- **Example:** Program fragment that leads to a **deadlock**

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm); }
else if (my_rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm); }
```

- Sequence of communication:

  Process 0 starts MPI_Recv() and process 1 starts MPI_Recv()
  → Process 0 wants to start MPI_Send() but has to wait for the termination of MPI_Recv()
  → this MPI_Recv() waits for the execution of MPI_Send() by process 1
  → this depends on the termination of MPI_Recv() by process 1
  → this MPI_Recv() waits for the execution of MPI_Send() by process 0

Deadlocks depending on the Implementation

- The occurence of a deadlock might also depend on the implementation of the MPI runtime system (usage of system buffers):
- **Example:** Implementation dependent **deadlock**

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status); }
else if (my_rank == 1) {
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status); }
```

- MPI implementation **with a system buffer** $\rightarrow$ **correct execution without a deadlock**. The messages sent are copied from the send buffer `sendbuf` into a system buffer. Control returns after the copy operation immediately to the sender.
- MPI implementation **without a system buffer** $\rightarrow$ **deadlock occurs**. Neither of the two processes can complete its `MPI_Send()` operation since the other process cannot execute the corresponding `MPI_Recv()`.

Secure Implementation with Blocking Communication

- An MPI program is called **secure** if its correctness does not depend on assumptions about the **existence of system buffers** or the **size of system buffers**.
  The MPI standard does **not specify** the provision of system buffers.
- In **absence of a system buffer** the operation must have **exclusive access to the program buffer** until the transfer is completed.
- **Example:** Program fragment without **deadlocks**

```
if (my_rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Secure Communication with $p > 2$ Processes (1)

- A secure implementation of point-to-point communication with more than two processes each sending and receiving a message is achieved by an **exact specification in which order** the send and receive operations are to be performed.
- **Example:** MPI program with p processes:
  Process i sends a message to process $(i+1) \bmod p$, $0 \leq i \leq p-1$
  Process i receives a message from process $(i-1) \bmod p$
  Thus, the messages are sent in a **logical ring**.
- A **secure implementation** cen be obtained by the following rule:
  Processes with an **even number** send first and then receive;
  Processes with an **odd number** receive first and then send.
- Exchange scheme for an **even number of processes**

Time	Process 0	Process 1	Process 2	Process 3
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 3	`MPI_Recv()` from 2
2	`MPI_Recv()` from 3	`MPI_Send()` to 2	`MPI_Recv()` from 1	`MPI_Send()` to 0

Secure Communication with $p > 2$ Processes (2)

- This scheme also leads to a secure implementation for an **odd number of processes**.
- Exchange scheme for an **odd number of processes**

Time	Process 0	Process 1	Process 2
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 0
2	`MPI_Recv()` from 2	`MPI_Send()` to 2	-wait-
3		-wait-	`MPI_Recv()` from 1

- Some communication operations like the `MPI_Send()` operation of process 2 can be **delayed** because the receiver calls the corresponding `MPI_Recv()` operation at a **later time**. But a **deadlock cannot occur**.

Data Exchange with MPI_Sendrecv() (1)

Situation: Each process **sends** and **receives** data

```
int MPI_Sendrecv (void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, int dest, int sendtag,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, int source, int recvtag,
                  MPI_Comm comm, MPI_Status *status)
```

- **sendbuf** Send buffer in which the data elements to be sent are stored;
- **sendcount** Number of data elements to be sent;
- **sendtype** Data type of the elements to be sent;
- **dest** Rank of the target process to which the data elements are sent;
- **sendtag** Tag for the message to be sent;
- **recvbuf** Receive buffer for the message to be received;
- **recvcount** Maximum number of data elements to be received;
- **recvtype** Data type of the data elements to be received;
- **source** Rank of the process from which the message is expected;
- **recvtag** Tag of the message to be received;
- **comm** Communicator used for the communication;
- **status** Data structure to store information on the message received.

Data Exchange with MPI_Sendrecv() (2)

- **Advantage** of MPI_Sendrecv(): The runtime system guarantees deadlock freedom.
- **Prerequisit**: sendbuf and recvbuf must be **disjoint, non-overlapping memory locations**.
- Messages of **different lengths** and **different data types** may be exchanged.
- If send and receive buffers are **identical**, the following MPI operation may be used:

```
int MPI_Sendrecv_replace (void *buffer, int count,
                          MPI_Datatype type, int dest,
                          int sendtag, int source, int recvtag,
                          MPI_Comm comm, MPI_Status *status)
```

- **buffer** Buffer that is used as both send and receive buffer.
- **Prerequisit:** The **number count** and the **data type type** of the data elements to be sent and to be received have to be **identical**.
  The runtime system is responsible for the temporary storage in system buffers when needed.

Overview

Non-blocking Communication with MPI

- **Blocking communication operations** do not fully utilize the communication hardware of parallel platforms that often operates asynchronously;
  $\rightarrow$ **Avoiding waiting times** with **non-blocking** operations.
- Properties of **non-blocking communication operations**
  - **Control** is **returned to the caller** immediately without blocking; the **communication operation** is only **initiated**.
  - The runtime system sends the message to the receiver **asynchronously**; $\rightarrow$ the program can perform other computations concurrently to the send operation;
  - **However:** The send buffer specified cannot be reused safely as long as the send operation is in progress.
  - The **sending process** is **notified** by the runtime system when the send operation is terminated using a `request` structure.

Non-blocking Send and Receive Operations

- Non-blocking **send operation**

```
int MPI_Isend (void *buffer, int count, MPI_Datatype type,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request)
```

  The parameters have the same meaning as for `MPI_Send()`.
  `MPI_Request` denotes an opaque object that **identifies a specific non-blocking communication operation**.

- Non-blocking **receive operation**

```
int MPI_Irecv (void *buffer, int count, MPI_Datatype type,
               int source, int tag, MPI_Comm comm,
               MPI_Request *request)
```

  Starting the receive operation informs the runtime system that the **receive buffer** specified is ready to **receive data**.

Status of Non-blocking Communication Operations

- Querying the **status of a non-blocking communication operation**

```
int MPI_Test (MPI_Request *request,
              int *flag,
              MPI_Status *status)
```

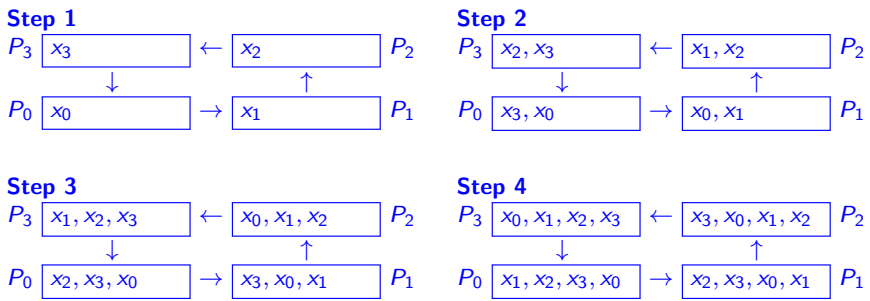
  - The call returns **flag = 1** if the send or receive communication operation specified by `request` has been completed; **flag = 0** denotes that the operation **is still in progress**.
  - If `MPI_Test()` is called for a **receive operation** that is completed the parameter `status` contains information on the message received as described by `MPI_Recv()`.

- Waiting for the **completion of a communication operation** The following MPI operation **blocks the calling process** until the send or receive operation specified by `request` is **completed**.

```
int MPI_Wait (MPI_Request *request,
              MPI_Status *status)
```

Example: Collection of Information from p Processes (1)

- Each of the p processes available stores the **same number of data elements in consecutive memory locations**. The data elements of each process should be made available to all other processes.
- The processes are logically arranged in a **ring**; Implementation using $p - 1$ steps:
- Illustration for $p = 4$ processes



Example: Collection of Information from p Processes (2)

Assumptions: Local data is provided in array **x**; Entire data is collected in array **y**; Blocking communication operations are used.

```
void Gather_ring (float x[], int blocksize, float y[]) {
   int i, p, my_rank, succ, pred, send_offset, recv_offset;
   MPI_Status status;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<blocksize; i++) y[i+my_rank * blocksize] = x[i];
   succ = (my_rank+1) % p;
   pred = (my_rank-1+p) % p;
   for (i=0; i<p-1; i++) {
      send_offset = ((my_rank-i+p) % p) * blocksize;
      recv_offset = ((my_rank-i-1+p) % p) * blocksize;
      MPI_Send (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
                MPI_COMM_WORLD);
      MPI_Recv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
                MPI_COMM_WORLD, &status);
}}
```

Deadlock freedom is ensured only if the MPI runtime system uses **system buffers** that are large enough.

Example: Collection of Information from p Processes (3)

Assumption: Non-blocking communication operations are used.

```
void Gather_ring (float x[], int size, float y[]) {
   int i, p, my_rank, succ, pred, send_offset, recv_offset;
   MPI_Status status;
   MPI_Request send_requ, recv_requ;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<size; i++) y[i+my_rank * size] = x[i];
   succ = (my_rank+1) % p; pred = (my_rank-1+p) % p;
   send_offset = my_rank * size;
   recv_offset = ((my_rank-1+p) % p) * size;
   for (i=0; i<p-1; i++) {
      MPI_Isend (y+send_offset, size, MPI_FLOAT, succ, 0,
                 MPI_COMM_WORLD, &send_requ);
      MPI_Irecv (y+recv_offset, size, MPI_FLOAT, pred, 0,
                 MPI_COMM_WORLD, &recv_requ);
      send_offset = ((my_rank-i-1+p) % p) * size;
      recv_offset = ((my_rank-i-2+p) % p) * size;
      MPI_Wait (&send_requ, &status);
      MPI_Wait (&recv_requ, &status);
}}
```

Synchronous Communication Mode

- The communication operations described until now use the **standard mode** of communication: The **runtime system** decides whether outgoing messages are **buffered** in a system buffer or transmitted directly **without buffering** to the receiver.
- Alternative: **Synchronous Mode:** A **send operation** will be **completed** not before the corresponding receive operation has started with the **reception of data**.
  $\rightarrow$ **Synchronization** between sender and receiver.
- **blocking send operation in synchronous mode:** `MPI_Ssend()`
  (Parameters identical to `MPI_Send()`)
- **non-blocking send operation in synchronous mode:** `MPI_Issend()`
  (Parameters identical to `MPI_Isend()`)
  There is **no synchronisation** between `MPI_Issend()` and the corresponding `MPI_Irecv()`; **Synchronization** between sender and receiver is performed when the **sender** calls `MPI_Wait()`.

Buffered Communication Mode

- **Buffered Mode:** Control will be **returned** by the send operation even if the corresponding **receive operation has not yet been started**.
  → The runtime system must **buffer** the outgoing message.
  - **blocking send operation in buffered mode:** `MPI_Bsend()`
    (Parameters identical to `MPI_Send()`)
  - **non-blocking send operation in buffered mode:** `MPI_Ibsend()`
    (Parameters identical to `MPI_Isend()`)
- The **buffer space** to be used by the runtime system must be **provided by the programmer**. **Provision of a buffer**:

  ```
  int MPI_Buffer_attach (void *buffer, int buffersize)
  ```

  buffersize is the size of the buffer `buffer` in **bytes**.
- **Detaching a buffer** previously provided:

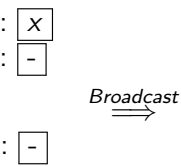
  ```
  int MPI_Buffer_detach (void *buffer, int *size)
  ```

- For **receive operations**, MPI provides the **standard mode** only.

Overview

Broadcast Operation

- All **global** communication operations are **blocking** in MPI.
- **Broadcast Operation:** The **root process** `root` sends the **same data block** to **all** other processes of the group.
  For a broadcast, **each process** has to call the following MPI function:

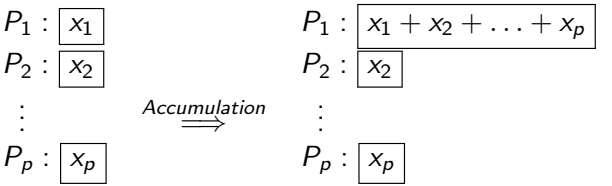



```
int MPI_Bcast (void *message,
               int count,
               MPI_Datatype type,
               int root,
               MPI_Comm comm)
```

- The **root process** provides the **data block** to be sent in the parameter `message`.
- All other processes provide in `message` their **receive buffer**.
- Control is returned to the calling process as soon as **its fraction** of the communication operation is completed.
  → `MPI_Bcast` does **not provide explicit synchronisation**.
- Each process has to specify the **same root process** `root`.

Accumulation Operation (1)

- There is a duality between a single broadcast and a **single accumulation operation**:
  - Each participating process provides a **block of data**.
  - Additionally, a **binary reduction operation** is provided.
  - The root process collects the data blocks provided by the participating processes and accumulates the individual data elements using the reduction operation specified.
- **Illustration** using the sum $(+)$ as the reduction operation:



Accumulation Operation (2)

- MPI provides the following predefined **reduction operations**:
  - arithmetical: $MPI\_\{MAX, MIN, SUM, PROD, MINLOC, MAXLOC\}$;
  - logical: $MPI\_\{LAND, BAND, LOR, BOR, LXOR, BXOR\}$;
  - $MPI\_\{MAXLOC, MINLOC\}$ additionally return the **index** attached by the **process** with the maximum or minimum value respectively;
  - Data types for a pair of values: $MPI\_\{FLOAT, \ldots, LONG\}\_INT$, MPI_2INT
- Syntax of the accumulation operation in MPI:

```
int MPI_Reduce (void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype type,
                MPI_Op op,
                int root,
                MPI_Comm comm)
```

Accumulation Operation (3)

- Additional reduction operations can be defined by the user with:

```
int MPI_Op_create (MPI_User_function *function,
                   int commute,
                   MPI_Op *op)
```

- The argument `function` specifies a **user-defined function** which must define the following four parameters:
  `void *in, void *out, int *len, MPI_Datatype *type.`
- The parameter `commute` specifies whether the function is **commutative** (`commute = 1`) or not (`commute = 0`).
- The call of `MPI_Op_create()` returns a reduction operation `op` which can then be used as parameter of `MPI_Reduce()`.

Accumulation Operation (4)

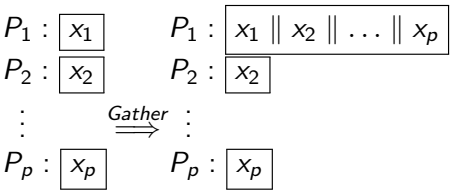
- **Example**: MPI program for the parallel computation of the **scalar product** (dot product)

```
int j, m, p, local_m, my_rank;
float local_dot, dot;
float local_x[100], local_y[100];

MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_size (MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf("%d", &m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j<local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce (local_dot, &dot, 1, MPI_FLOAT, MPI_SUM,
            0, MPI_COMM_WORLD);
MPI_Finalize();
```

Gather Operation (1)

- Impact of a **gather operation**: each of the participating n processes provides a block of data that is **collected at the root process**.
- Illustration and syntax of a gather operation




```
int MPI_Gather (void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm)
```

- **sendcount**: Number of data elements with data type **sendtype** to be sent;
- **sendbuf**: send buffer that is provided by each participating process;
- **recvbuf**: receive buffer provided by the root process **root** that is large enough to hold all data elements sent.

Gather Operation (2)

- Each data block provided has to comprise the **same number** of elements with the **same data type**.
  Data blocks will be stored equally spaced at the root process with offset **recvcount**.
  Each process has to specify the same **root process** `root`.
- **Example** for `MPI_Gather()`:
  - Each process provides **100 integer values**.
  - The **root process** collects the data blocks in its receive buffer.

```
MPI_Comm comm;
int sendbuf[100], my_rank, root = 0, gsize, *rbuf;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   rbuf = (int *) malloc (gsize*100*sizeof(int));
}
MPI_Gather (sendbuf, 100, MPI_INT, rbuf, 100,
            MPI_INT, root, comm);
```

Gatherv Operation (1)

- more general vector-based *MPI_Gatherv* operation:
  each process can provide a **different number** of elements.

```
int MPI_Gatherv (void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int *recvcounts,
                 int *displs,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
```

- **sendcount**: number of data elements to be sent;
- **recvcounts**: array, where `recvcounts[i]` denotes the number of elements provided by process `i`;
- **displs**: array that specifies the positions of the data blocks in **recvbuf**;
- **recvcounts** and **displs** are only significant at the **root process**.
- **Overlaps** in the receive buffer must not occur;
  $\rightarrow$ $displs_{root}[i+1] \geq displs_{root}[i] + sendcount_i$ with $recvcounts_{root}[i] = sendcount_i$

Gatherv Operation (2)

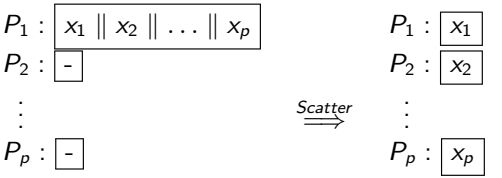
- **Example** for the use of `MPI_Gatherv()`:
  - Each process provides **100 integer values**.
  - The blocks received are stored in the receive buffer such that there is a **free gap of 10 elements** between two blocks.

```
int my_rank, root = 0, gsize, sbuf[100];
int *rbuf, *displs, *rcounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   rbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   rcounts = (int *) malloc(gsize*sizeof(int));
   for (i = 0; i < gsize; i++) {
      displs[i] = i*stride;
      rcounts[i] = 100; }
}
MPI_Gatherv (sbuf, 100, MPI_INT, rbuf, rcounts, displs,
             MPI_INT, root, comm);
```

- An **error** occurs for `stride<100`, since this would lead to **overlapping entries** in the receive buffer of the root process.

Scatter Operation (1)

- **Scatter:** The root process provides a data block (with the same size but possibly different elements) for each participating process.
- **Illustration**:



- Syntax of the MPI operation

```
int MPI_Scatter (void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

- **sendbuf** is the send buffer provided by the root process **root** which contains a data block with **sendcount** elements of data type **sendtype** for each process of communicator **comm**.

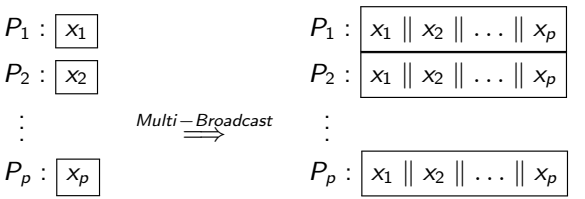
Scatter Operation (2)

- In the send buffer, the blocks are ordered in **rank order of the receiving processes**.
- **flexible number of data elements** per receiver: vector-based version **MPI_Scatterv**.
- **Example**: Process 0 sends **100 elements** to each process i such that there is a gap of **10 elements** between neighboring send blocks.

```
MPI_Comm comm;
int rbuf[100], my_rank, root=0, gsize;
int *sbuf, *displs, *scounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   sbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   scounts = (int *) malloc(gsize*sizeof(int));
   for (i=0; i<gsize; i++) {
      displs[i] = i*stride;
      scounts[i]=100; }
}
MPI_Scatterv (sbuf, scounts, displs, MPI_INT, rbuf,
              100, MPI_INT, root,comm);
```

Allgather Operation (1)

- **Multi-broadcast operation:** Each process sends **the same** block of data to each other process
  $\rightarrow$ Each process performs a **single-broadcast operation**.
- **Illustration**:



- Syntax of the MPI operation

```
int MPI_Allgather (void *sendbuf,
                   int sendcount,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int recvcount,
                   MPI_Datatype recvtype,
                   MPI_Comm comm)
```

- **sendbuf**: **send buffer** provided by each of the participating processes.

Allgather Operation (2)

- Each process provides a **receive buffer recvbuf** in which all received data blocks are collected in **rank order of the sending processes**.
- A multi-broadcast does **not have a distinguished root process**.
- **Example:** each process contributes a **send buffer** with 100 integer values which are made available by a multi-broadcast operation to all processes:

```
int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

- Syntax of the vector-based MPI operation **MPI_Allgatherv**:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm)
```

Allgather Operation (2)

- Each process provides a **receive buffer recvbuf** in which all received data blocks are collected in **rank order of the sending processes**.
- A multi-broadcast does **not have a distinguished root process**.
- **Example:** each process contributes a **send buffer** with 100 integer values which are made available by a multi-broadcast operation to all processes:

```
int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100,
               MPI_INT, comm);
```

- Syntax of the vector-based MPI operation **MPI_Allgatherv**:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm)
```

Multi-accumulation Operation (1)

- **Each** process provides a data block of the same size.
- The data blocks are accumulated with a **reduction operation** $\rightarrow$ multi-accumulation equals a **single-accumulation** with a subsequent **broadcast**.
- The available **reduction operations** are the same as for `MPI_Reduce()`.
- **Illustration**:

$$\begin{array}{l} P_0 : x_0 \\ P_1 : x_1 \\ \vdots \\ P_{p-1} : x_{p-1} \end{array} \quad \overset{Multi-Accumulation(+)}{\Longrightarrow} \quad \begin{array}{l} P_0 : x_0 + x_1 + \ldots + x_{p-1} \\ P_1 : x_0 + x_1 + \ldots + x_{p-1} \\ \vdots \\ P_{p-1} : x_0 + x_1 + \ldots + x_{p-1} \end{array}$$

Multi-accumulation Operation (2)

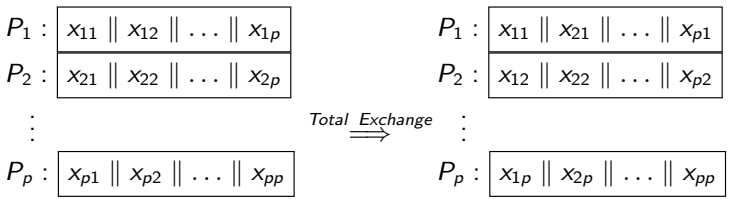
- Syntax of the MPI operation

```
int MPI_Allreduce (void *sendbuf,
                   void *recvbuf,
                   int count,
                   MPI_Datatype type,
                   MPI_Op op,
                   MPI_Comm comm).
```

- **sendbuf** is the **local buffer** in which each process provides its local data;
- **recvbuf** is the local buffer of each process in which the accumulated result is **collected**.

Total Exchange – MPI_Alltoall() (1)

- **Each** process provides a **different** block of data for **each** other process.
- Each process collects the data blocks provided for this particular process.
- **Illustration**:



Total Exchange – MPI_Alltoall() (2)

- Syntax of the MPI operation

```
int MPI_Alltoall (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)
```

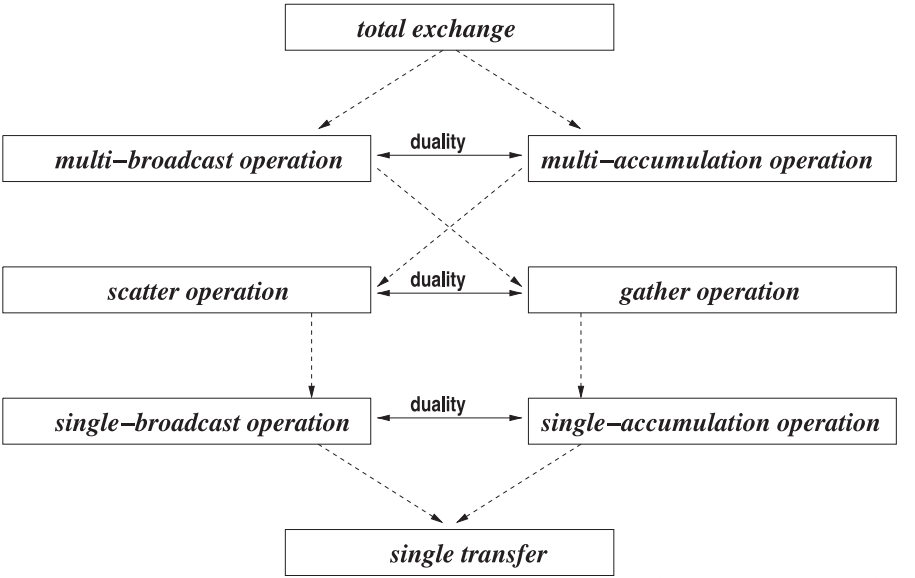
- **sendbuf** is the **send buffer** in which each process provides for each process a block of data with **sendcount** elements of type **sendtype**; The blocks are **arranged in rank order** of the target processes.
- Each process provides a receive buffer **recvbuf** in which the data blocks received from the other processes are stored;
  The blocks received are **stored in rank order** of the sending processes in communicator **comm**.

Total Exchange – MPI_Alltoall() (3)

- Syntax of the more general **vector-based version** for data blocks of different sizes

```
int MPI_Alltoallv (void *sendbuf,
                   int *scounts,
                   int *sdispls,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int *rcounts,
                   int *rdispls,
                   MPI_Datatype recvtype,
                   MPI_Comm comm)
```

Hierarchy of Communication Operations



Overview

Example: Matrix-Vector Multiplication – Overview

- Multiplication of a dense $n \times m$ matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ with a vector $\mathbf{b} \in \mathbb{R}^m$:

$$c_i = \sum_{j=1}^{m} a_{ij} b_j, \quad i = 1, ..., n,$$

- Computation of n **scalar products** of the rows $\mathbf{a}_1, ..., \mathbf{a}_n$ of $\mathbf{A}$ with the vector $\mathbf{b}$:

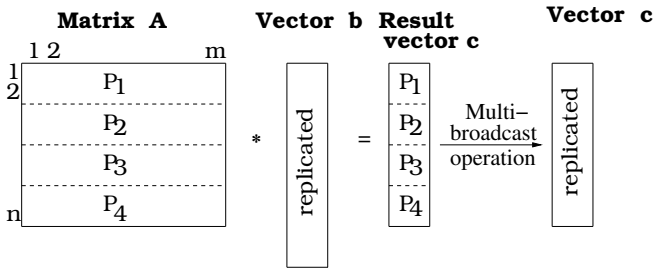
$$\mathbf{A} \cdot \mathbf{b} = \begin{pmatrix} (\mathbf{a}_1, \mathbf{b}) \\ \vdots \\ (\mathbf{a}_n, \mathbf{b}) \end{pmatrix},$$

- **Linear combination** of the columns $\tilde{\mathbf{a}}_1, ..., \tilde{\mathbf{a}}_m$ of $\mathbf{A}$, where $\mathbf{b} \in \mathbb{R}^n$ contains the coefficients of this linear combination

$$\mathbf{A} \cdot \mathbf{b} = \sum_{j=1}^{m} b_j \tilde{\mathbf{a}}_j \, .$$

Example: Matrix-Vector Multiplication – Distribution of the Scalar Products

- **Data distribution**: Distribution of **A** and **b** on the processors:
  - **row-oriented** blockwise distribution of **A**:
    processor P_k stores rows $\mathbf{a}_i$ for $i = n/p \cdot (k-1) + 1, \ldots, n/p \cdot k$ in its local memory
  - **replicated** storage of **b**



- The final result should be available on **all processors** $\rightarrow$ Re-distribution using a **multi-broadcast operation**;

Example: Matrix-Vector Multiplication – Program Sketch I

- **row-oriented** distribution of the matrix
  - each processor stores a **local array** `local_A` of dimension `local_n` $\times$ `m`;
  - initialization of the local array of processor P_k:

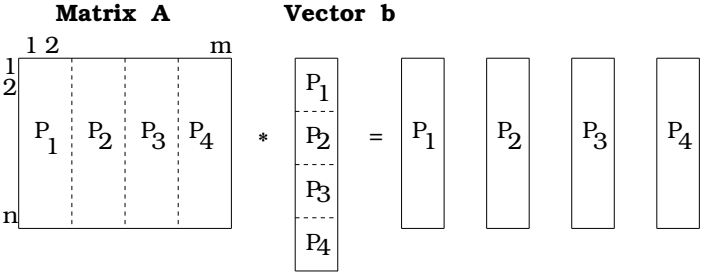
$$\texttt{local\_A[i][j]} = \texttt{A[i} + (\texttt{k} - 1) * \texttt{n/p][j]}$$

- Program Sketch I

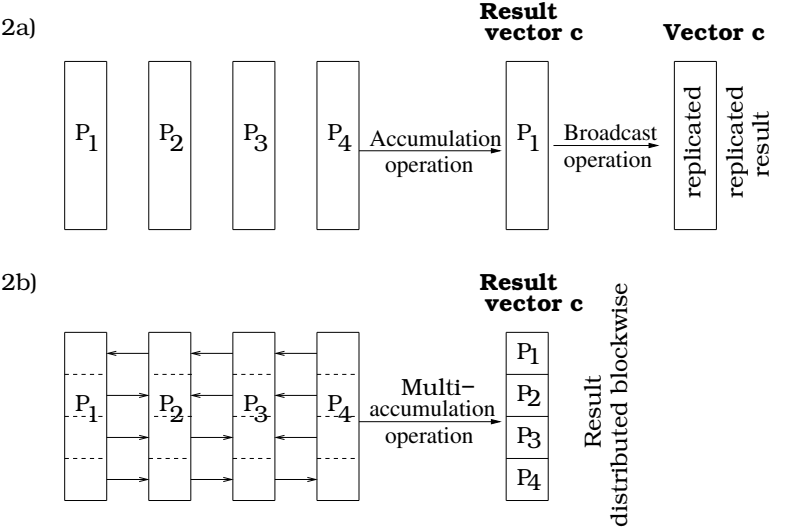
```
local_n = n/p;
for (i=0; i<local_n; i++)
   local_c[i] = 0;
for (i=0; i<local_n; i++)
   for (j=0; j<m; j++)
      local_c[i] = local_c[i] + local_A[i][j] * b[j];
MPI_Allgather( local_c, local_n, MPI_DOUBLE,
               global_c, local_n, MPI_DOUBLE, comm);
```

Example: Matrix-Vector Multiplication – Distribution of the Linear Combinations (1)

- **Data distribution**:
  - **column-oriented** blockwise distribution of **A**:
    processor P_k stores the columns $\tilde{\mathbf{a}}_i$ with
    $i = m/p \cdot (k-1) + 1, \ldots, m/p \cdot k$
  - blockwise distribution of **b**



Example: Matrix-Vector Multiplication – Distribution of the Linear Combinations (2)



Example: Matrix-Vector Multiplication – Program Sketch II

- **Data distribution**: **column-oriented** blockwise distribution of the matrix **A**:
  - each processor stores a **local array** `local_A` of dimension `n` $\times$ `local_m`;
  - P_k computes the **partial linear combination**

$$\mathbf{d}_k = \sum_{j=m/p\cdot(k-1)+1}^{m/p\cdot k} b_j \tilde{\mathbf{a}}_j.$$

- Program Sketch II

```
local_m=m/p;
for (i=0; i<n; i++) d[i] = 0;
for (j=0; j<local_m; j++)
   for (i=0; i<n; i++)
      d[i] = d[i] + local_b[j] * local_A[i][j];
MPI_Reduce (d, c, n, MPI_DOUBLE, MPI_SUM, 0,
            comm);
MPI_Bcast (c, n, MPI_DOUBLE, 0, comm);
```

Overview

Deadlocks with Global Communication Operations

- Global communication operations in MPI are always **blocking**; Thus, **without the use of a system buffer** these operations **synchronise** the participating processes.
- **Program fragment** with **two** participating processes:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Bcast (buf2, count, type, 1, comm); break;
case 1: MPI_Bcast (buf2, count, type, 1, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

- The runtime system detects an **error** if it matches the first calls to `MPI_Bcast()` of each process since **different root processes** are specified.
- A **deadlock** occurs if the MPI runtime system matches the `MPI_Bcast()` operations with **the same root process** and **no system buffers are used** or the system buffers **are too small**.
- The error or deadlock can be **avoided** by letting the participating processes call the matching `MPI_Bcast()` operation in the same order.

Deadlocks with Mixed Global and Point-to-point Communication

- **Deadlocks** can also occur when **mixing global communication operations** and **point-to-point communication**:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, 0, tag, comm,
                  &status);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

- If **no system buffers** are used, a **deadlock** occurs.
- The deadlock can be avoided by calling the **corresponding** communication operations in the **same order**.

Synchronization Behavior of Global Communication Operations (1)

- The **synchronization behavior** of collective communication operations depends on the use of system buffers by the MPI runtime system.
- **Example** that possibly leads to different execution orders

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
                  comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
                  comm, &status);
        break;
case 2: MPI_Send (buf2, count, type, 1, tag, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

Synchronization Behavior of Global Communication Operations (2)

- Execution order that may occur **with or without system buffers**:

process$_0$		**process$_1$**		**process$_2$**
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
`MPI_Bcast()`		`MPI_Bcast()`		`MPI_Bcast()`
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		

- Execution order that may only occur **with system buffers**:

process$_0$		**process$_1$**		**process$_2$**
`MPI_Bcast()`				
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		
		`MPI_Bcast()`		
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
				`MPI_Bcast()`

- **Conclusion:** a **non-deterministic program behavior** may result when system buffers are used.
- Secure **synchronization** of all processes of a communicator:

```
MPI_Barrier (MPI_Comm comm)
```

Overview

Process Groups and Communicators

- A **process group** is an ordered set of processes of a parallel program. Each process of a group gets a uniquely defined process number(**rank**).
- A process may be a member of **multiple groups** and may have different ranks in each of these groups.
- For the programmer, a group is an object of type `MPI_Group` which can only be accessed via a **handle**.
- **Global communication operations** can be **restricted** to previously defined groups.
- Each process group is associated with a **communication domain** that is *locally* represented by **communicators**.
  MPI_COMM_WORLD is the predefined communicator for the **global process group**.
- **Intra-communicators** support the execution of arbitrary collective communication operations on a **single** group of processes.
- **Inter-communicators** support the execution of point-to-point communication operations **between two process groups**.

Operations on Process Groups (1)

- The corresponding **process group** to a given communicator `comm` can be obtained by calling

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
```

  MPI_GROUP_EMPTY denotes the **empty process group**.

- New process groups can be constructed based on existing groups:
- **Union** of two existing groups `group1` and `group2`:

```
int MPI_Group_union (MPI_Group group1,
                     MPI_Group group2,
                     MPI_Group *new_group)
```

  The processes in `group1` **keep their ranks** from `group1` and the processes in `group2` which are not in `group1` get **subsequent ranks** in consecutive order.

Operations on Process Groups (2)

- The **intersection** of two groups is obtained by calling

```
int MPI_Group_intersection (MPI_Group group1,
                            MPI_Group group2,
                            MPI_Group *new_group)
```

  where the **process order** from `group1` is kept for `new_group`. The processes in `new_group` get **successive ranks** starting from 0.

- The **set difference** of two groups is obtained by calling

```
int MPI_Group_difference (MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group),
```

  where the process order from `group1` is kept as well.

Operations on Process Groups (3)

- **Construction of a subset** of an existing group:

```
int MPI_Group_incl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group)
```

  - `ranks` is an integer array with `p` entries.
  - The call creates a **new group** `new_group` with `p` processes which have **ranks** from `0` to `p-1`.
  - Process `i` is the process which has rank `ranks[i]` in group `group`.
  - The group `group` must contain **at least** `p` processes and the values `ranks[i]` must be **valid process ranks** in `group` which are **different from each other**.

- **Deletion of processes** from a group:

```
int MPI_Group_excl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group)
```

  The **new group** `new_group` is generated by **deleting** the processes with ranks `ranks[0]`,...,`ranks[p-1]` from `group`.

Operations to Obtain Information about Process Groups

- The **size of a process group** group can be obtained by calling

```
int MPI_Group_size (MPI_Group group, int *size)
```

The group size is returned in *size.
The **rank of the calling process** in a group group can be obtained by calling:

```
int MPI_Group_rank (MPI_Group group, int *rank)
```

- To check whether two **process groups** describe the **same process group** the function

```
int MPI_Group_compare (MPI_Group group1,
                       MPI_Group group2, int *res)
```

can be used. Possible return values *res:

  - *res = MPI_IDENT: the groups group1 and group2 contain the **same processes in the same order**;
  - *res = MPI_SIMILAR: both groups contain the **same processes** but group1 uses a different order than group2;
  - *res = MPI_UNEQUAL: the two groups contain **different processes**.

Operations on Communicators (1)

- Generation of a new **intra-communicator** to a given group of processes:

```
int MPI_Comm_create (MPI_Comm comm,
                     MPI_Group group,
                     MPI_Comm *new_comm)
```

  - `group` must specify a process group which is a **subset** of the process group associated with communicator `comm`.
  - **All** processes of `comm` must call `MPI_Comm_create()` with **the same group** as an argument.
  - Result of the call: each calling process which is a member of group `group` obtains a **pointer to the new communicator** `new_comm`.
  - Processes not belonging to `group` get `MPI_COMM_NULL` as return value in `new_comm`.

Operations on Communicators (2)

- A **Splitting** of a communicator can be obtained by calling:

```
int MPI_Comm_split (MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *new_comm)
```

  - The process group associated with communicator `comm` is partitioned into **a number of disjoint subgroups** that equals the number of **different values** specified in `color`.
  - Each **subgroup** contains all processes that specify **the same value** for `color`.
  - The **rank order** of the processes within a subgroup is defined by the argument `key`.
  - If two processes specify **the same value** for `key` the order of the original group is used.
  - If a process specifies `color = MPI_UNDEFINED`, it is **not a member of any of the subgroups** generated.
  - Each participating process gets a pointer `new_comm` to the communicator of that subgroup which the process belongs to.

Operations on Communicators (3)

- **Example**: We consider a group of **10 processes** each of which calls the operation `MPI_Comm_split()` with the following argument values:

process	a	b	c	d	e	f	g	h	i	j
rank	0	1	2	3	4	5	6	7	8	9
color	0	$\perp$	3	0	3	0	0	5	3	$\perp$
key	3	1	2	5	1	1	1	2	1	0

- This call generates **three subgroups**:
  {f, g, a, d}, {e, i, c} and {h}
- The groups contain the processes in the indicated order.

Overview

Process Topologies

Idea: The **arrangement** of the processes in a **grid structure** facilitates data exchanges in some applications.
Definition of a **virtual Cartesian grid structure of arbitrary dimension**:

```
int MPI_Cart_create (MPI_Comm comm,
                     int ndims,
                     int *dims,
                     int *periods,
                     int reorder,
                     MPI_Comm *new_comm).
```

- `comm` is the original communicator **without topology**;
- `ndims` specifies the **number** of dimensions of the grid to be created;
- `dims` is an integer array with `ndims` elements where `dims[i]` denotes the **total number of processes in dimension** i.
- The array `periods` of size `ndims` specifies for each dimension whether the grid is **periodic** (entry 1) or not (entry 0) in this dimension.
- For `reorder = false`, the processes in `new_comm` have the **same rank** as in `comm`.

Process Topologies – Example

Example: Let `comm` be a communicator with **12 processes**.
Using the initializations `dims[0] = 3`, `dims[1] = 4`, `period[0] = period[1] = 0`, `reorder = 0`, the call

```
MPI_Cart_create (comm, 2, dims, period, reorder, &new_comm)
```

generates a **virtual** 3×4 **grid** with the following group ranks and coordinates:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Process Topologies – Distribution of the Processes

The following function helps the programmer to select a balanced distribution of the processes for the different dimensions:

```
int MPI_Dims_create (int nnodes, int ndims,int *dims)
```

- `ndims` is the **number of dimensions** in the grid to be defined;
- `nnodes` is the **total number of processes** in the grid;
- `dims` is an integer array of size `ndims`.
  In the case `dims[i] = 0` is specified for the call, `dims[i]` contains the **number of processes** in dimension `i` after the call.
  The function tries to assign the **same number of processes** to each dimension.
- The number of processes in a dimension *i* can be **fixed** by setting `dims[i]` to the desired number of processes **before the call**.
  The MPI runtime system sets the entries of the **other, non-initialized** entries of `dims` accordingly.

Process Topologies – Translation of Cartesian Coordinates

- When defining a **virtual topology**, each process has a **group rank**, and also a position in the virtual grid which can be expressed by its **Cartesian coordinates**.
- **Translation** of **Cartesian coordinates** into **group ranks**:

```
int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

  The call translates the **Cartesian coordinates** of a process provided in the array `coords` into the **group rank** according to the virtual grid associated with `comm`.
- **Translation** of **group ranks** into **Cartesian coordinates**:

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int dims,
                     int *coords)
```

- `rank` contains the process number; `dims` denotes the **number of dimensions** in the virtual grid defined for communicator `comm`. The **Cartesian coordinates** of the process are returned in the array `coords`.

Process Topologies – Determining Neighboring Processes

- A typical communication pattern in many grid-based algorithms is that processes communicate with **their neighboring processes in a specific dimension**.
- **Determining the neighboring processes** in each dimension of the grid:

```
int MPI_Cart_shift (MPI_Comm comm, int dir, int dspl,
                    int *rk_src, int *rk_dst).
```

- `dir` specifies the **dimension** for which the neighboring process should be determined. The parameter `dspl` specifies the **displacement desired**.
  positive value: request neighbors in **upward direction**;
  negative value: request neighbors in **downward direction**.
- The result of the call is that `rk_dst` contains the **group rank of the neighboring process** in the specified dimension and distance; `rk_src` returns the **rank of the process** for which the calling process is the neighbor in the specified dimension and distance.

Process Topologies – Example (1)

We consider **12 processes** that are arranged in a 3×4 grid structure with **periodic connections**. Each process stores a **floating-point value** which is exchanged with the neighboring process in dimension 0:

```
int coords[2], dims[2], periods[2], source, dest;
int my_rank, reorder = 0;
float a, b;
MPI_Comm comm_2d;
MPI_Status status;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
dims[0] = 3; dims[1] = 4; periods[0] = periods[1] = 1;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods,
                 reorder, &comm_2d);
MPI_Cart_coords (comm_2d, my_rank, 2, coords);
MPI_Cart_shift  (comm_2d, 0, coords[1], &source, &dest);
a = my_rank;
MPI_Sendrecv (&a, 1, MPI_FLOAT, dest, 0, &b, 1, MPI_FLOAT,
              source, 0, comm_2d, &status);
```

- The operation `MPI_Cart_shift()` is used to determine the communication partners `dest` and `source` for the operation `MPI_Sendrecv()`.

Process Topologies – Example (2)

- The **distance between the neighbors** in dimension 0 **increases** with the coordinates in dimension 1 since `displs = coord` is used;
  $\rightarrow$ in **each column** of the grid a **different exchange** is executed.
- The following diagram illustrates the exchange. For each process, its **rank**, its **Cartesian coordinates**, and its communication partners in the form **source**|**dest** are given in this order.

0 (0,0) 0|0	1 (0,1) 9|5	2 (0,2) 6|10	3 (0,3) 3|3
4 (1,0) 4|4	5 (1,1) 1|9	6 (1,2) 10|2	7 (1,3) 7|7
8 (2,0) 8|8	9 (2,1) 5|1	10 (2,2) 2|6	11 (2,3) 11|11

- For example, for the process with `rank=5` it is `coords[1]=1` and therefore `source = 9` (lower neighbor in dimension 0) and `dest = 1` (upper neighbor in dimension 0).

Process Topologies – Subgrids (1)

A **virtual topology** can be partitioned into **subgrids**:

```
int MPI_Cart_sub (MPI_Comm comm, int *remain_dims,
                  MPI_Comm *new_comm)
```

- `comm` is the communicator for which the **virtual topology** has been defined;
- `new_comm` denotes the **new communicator** for which the new topology as a **subgrid of the original grid** is defined.
- The **subgrid selection** is controlled by the array `remain_dims` which contains an entry for each dimension of the original grid.
  Setting `remain_dims[i]=1` means that the *i*th dimension **is kept** in the subgrid;
  `remain_dims[i]=0` means that the *i*th dimension **is dropped** in the subgrid.
- If a dimension `i` does not exist in the subgrid, the size of dimension `i` defines the **number of subgrids** that have been generated for this dimension.

Process Topologies – Subgrids (2)

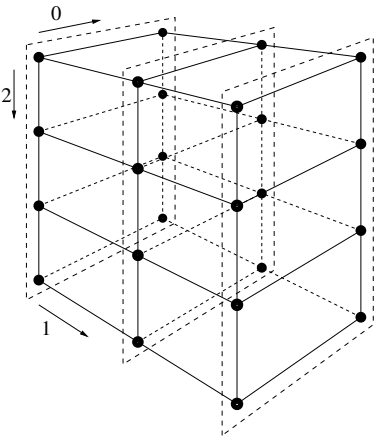
- A call to `MPI_Cart_sub()` generates a **new communicator** `new_comm` for each calling process, representing the corresponding subgroup of the subgrid to which the calling process belongs.
- The **dimensions of the different subgrids** result from the dimensions for which `remain_dims[i]` has been set to `1`.
- The **total number of different communicators** generated is defined by the product of the number of processes in all dimensions `i` for which `remain_dims[i]` has been set to `0`.

Process Topologies – Example for Subgrids

- **Example:** let `comm` be a communicator for which a $(2 \times 3 \times 4)$ virtual grid topology has been defined. Calling

  ```
  int MPI_Cart_sub (comm_3d, remain_dims, &new_comm)
  ```

  with `remain_dims = (1,0,1)` generates **three different communicators** each representing a (2×4) grid:



Process Topologies – Inquiring Information about a Virtual Topology

Information on the **virtual topology** that has been defined for a communicator can be inquired using the following two **MPI functions**:

- **Number of dimensions** of the virtual grid:

```
int MPI_Cartdim_get (MPI_Comm comm, int *ndims)
```

- **Cartesian coordinates** of the **calling process** within the virtual grid associated with communicator `comm` can be obtained by calling

```
int MPI_Cart_get (MPI_Comm comm, int maxdims, int *dims,
                  int *periods, int *coords)
```

  where `maxdims` is the number of **dimensions** of the virtual topology, and `dims`, `periods`, and `coords` are **arrays** of size `maxdims`.
  - The arrays `dims` and `periods` have the same meaning as for `MPI_Cart_create()`.
  - The array `coords` is used to **return the coordinates**.

Timings and Aborting Processes

- Measurement of the **execution times** of program parts:

  ```
  double MPI_Wtime (void)
  ```

- typical usage:

```
start = MPI_Wtime();
part_to_measure();
end = MPI_Wtime();
```

- `MPI_Wtime()` returns an **absolute time**, not the system time
- **Abortion** of the execution of all processes of a **communicator**:

  ```
  int MPI_Abort (MPI_Comm comm, int error_code)
  ```

Overview

Introduction to MPI-2

- MPI-2 is an **extension** of MPI.
- Each valid MPI program is also a valid MPI-2 program.
- Additional functionality resides in the areas of **dynamic** process management, **one-way** communication and **parallel in- / output**.

Overview

Process creation and management

- MPI-2 allows **creation** of new and **removal** of existing processes.
- Many MPI-2 functions use data structure of type `MPI_Info`, which enables interaction between the respective MPI-2 implementation and the underlying **operating system**.
- Use of `MPI_Info` leads to **limited portability**.
- Data structure of type `MPI_Info`, basically, contains **pairs** of the form `(key, value)` (which in C consist of character strings of type `char *` terminated by `'\0'`).

MPI_Info Data Structure

Access of the `MPI_Info` data structure is supported by multiple functions. By

```
int MPI_Info_create(MPI_Info *info)
```

a **new** structure of this type is created. The function

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

adds a **new** pair (`key, value`) to `info`, or **overwrites** an already existing pair by with the same content of `key`. Call of

```
int MPI_Info_get(MPI_Info info, char *key,
                 int valuelen, char *value, int flag)
```

searches in `info` for a pair with the provided `key` and writes in `value` the respective value with a max. length of `valuelen`. Value of `flag` is set to `false` if no matching pair was found, otherwise it is set to `true`. Pair (`key, value`) can be **removed** by the following function:

```
int MPI_Info_delete(MPI_Info info, char *key)
```

Creation of New Processes

New processes can be created in MPI-2 by the following function:

```
int MPI_Comm_spawn(char *command, char *argv[],
                   int maxprocs, MPI_Info info,
                   int root, MPI_Comm comm,
                   MPI_Comm *intercomm, int errcodes[])
```

- `command` denotes the name of the **program** to be started.
- `argv` contains **command line arguments** (where `argv[0]` not, as usual in C contains the program name). With `MPI_ARGV_NULL` an empty parameter list can be displayed.
- `maxprocs` indicates the **number** of processes to be started.
- `info` can be used to specify, among other things, the path, name and parameters of the program to be invoked.
  To preserve **portability** `MPI_INFO_NULL` should be passed.
- The above arguments are only evaluated by the **root process** `root`. It also splits new processes.

Creation of New Processes

- The function must be called by **all** processes of communicator `comm`.
- **Inter-communicator** `intercomm` serves for communication between the communicator `comm` and new processes.
- Newly split processes are located in a different group from the one the root process is in. All of them are contained in their **own** `MPI_COMM_WORLD` communicator.
- **Inter-communicator**, which is returned to the calling processes, can be determined by the **new** processors by means of the following function: `MPI_Comm_get_parent(MPI_Comm *parent)`
- `errcodes` is a field of min. length `maxprocs`. After the call each element contains either an **error code** or `MPI_SUCCESS` if the respective process was successfully started.

Creation of Multiple New Processes

Several **different** MPI programs with possibly different command line arguments can be split off as new processes by the following function:

```
int MPI_Comm_spawn _multiple(
                    int count, char *commands[] , char **argv[],
                    int maxprocs[], MPI_Info infos[],
                    int root, MPI_Comm comm,
                    MPI_Comm *intercomm, int errcodes[])
```

- `count` indicates the **number** of different calls.
- `commands`, `argv`, `maxprocs` and `infos` are fields of length `count`, the elements of which have the same meaning as the corresponding arguments of `MPI_Comm_Spawn()`.
- Other arguments directly correspond to those of `MPI_Comm_Spawn()`.
- `errcodes` contains $\sum_{i=0}^{\texttt{count}-1} \texttt{maxprocs}[i]$ error codes, where the order of those codes corresponds to the respective calls in `commands` field.

- By appropriate multiple calls of the function `MPI_Comm_spawn()` the same programs can be started similarly to a single call of the `MPI_Comm_spawn_multiple()` function. Though, in case of a single `MPI_Comm_spawn_multiple()` call only one new **shared** communicator `MPI_COMM_WORLD` is created for all new processes, while a single call of `MPI_Comm_spawn()` always creates another **different** communicator `MPI_COMM_WORLD`.

- The **maximum number of processes**, which can be active concurrently is provided at program launch by `MPI_Init()` in the `MPI_UNIVERSE_SIZE` attribute of the communicator `MPI_COMM_WORLD`. It, typically, corresponds to the number of available processors.

Overview

One Sided Communication

A source process can access the address space of a destination process without an active involvement of the destination process.
$\rightarrow$ RMA (remote memory access)

- flexible and dynamic distribution of program data on the memories of the participating processors
- flexible memory access of the participating processors
- coordination of the memory accesses through the programmer, i.e. at a time different processes should not modify memory addresses of an address space concurrently because then race conditions are possible

Overview

Window Objects I

Situation:

- Process A should access the local memory region from a process B Windows creation for the external access.
- Therefor the Process B has open its memory region for the external access.

```
int MPI_Win_create(void *base,
                   MPI_Aint size,
                   int displ_unit,
                   MPI_Info info,
                   MPI_Comm comm,
                   MPI_Win *win)
```

- each process from the communicator comm has to execute that operation
- base is the start address of the memory region

Window Objects II

- size is the size in bytes
  (MPI type `MPI_Aint` is used for representing the size of memory regions; larger than 2^{32})

Window Objects III

- `displ_unit` specifies the displacement between neighboring entries of the window
- info data structure used to give the runtime system additional hints (normally `info = MPI_INFO_NULL`)
- `comm` is the communicator of the processes taking part in the window creation with `MPI_Win_create()`
- returns: window object of type `MPI_Win`

Releasing a window for external access:
`int MPI_Win_free (MPI_Win *win)`
All operations of a participating processes have to be finished.

Overview

RMA-Operation I

Data transfer with three different non-blocking RMA operations:

- `MPI_Put()` transfers data from the memory of the calling process into the window of another process
- `MPI_Get()` transfers data from the window of the destination process into the memory of the calling process
- `MPI_Accumulate()` accumulates the data into the window of the destination process

Test of completeness of a non blocking RMA-operation by using:
Synchronization Operations
$\rightarrow$ Forwarding of a local buffer of a communication operation should only be done after using one of the following synchronization operations

Put-Operation I

Storing a data block into the memory of another process:

```
int MPI_Put(void *origin_addr,
            int origin_count,
            MPI_Datatype origin_type,
            int target_rank,
            MPI_Aint target_displ,
            int target_count,
            MPI_Datatype target_type,
            MPI_Win win)
```

- `origin_addr` starting address of the data block provided by the calling process,
- `origin_count` is the number of elements in this data block, `origin_type` is the data type of the elements.
- `target_rank` specifies the rank of the destination process. (This destination process has to have created a window object `win` by a previous call to `MPI_Win_create()`

Put-Operation II

- `target_displ` specifies the distance between the start of the window and the start of the destination buffer within the target process
- `target_count` specifies the number of elements to be received in the buffer at the target process
- `target_type` represent the data type at the target process.

The data block is stored into the memory of the target process starting at position:

```
target_addr := window_base + target_displ * displ_unit
```

mit

`window_base` = Starting address of the window of the target process and `displ_unit` = stride between consecutive elements of a window

Corresponds to the semantics of two sided communication: Source process with send operation

```
int MPI_Isend (origin_addr, origin_count, origin_type, target_rank,
tag, comm)
```

Target process with receive operation

```
int MPI_Recv (target_addr, target_count, target_type, source, tag,
comm, &status)
```

Put-Operation III

`comm` is communicator belonging to the group for window win.

Get-Operation I

Reading a data block from the memory of another process

```
int MPI_Get(void *origin_addr,
            int origin_count,
            MPI_Datatype origin_type,
            int target_rank,
            MPI_Aint target_displ,
            int target_count,
            MPI_Datatype target_type,
            MPI_Win win)
```

- `origin_addr` is the starting address of the receive buffer in the local memory of the calling process,
- `origin_count` specifies the number of elements from type origin_type, transferred to the receiving buffer.
- `target_rank` is the rank of the target process, i.e. the process to be read from
- `win` is the window object

Get-Operation II

- Analogous to the `MPI_Put()` operation the remaining parameters specify the position and the number of elements of the data elements read from the data block of the destination window
  Its starting address can be calculated by
  `target_addr := window_base + target_displ * displ_unit.`

Accumulate-Operation I

Accumulation of data in the memory of another process

```
int MPI_Accumulate(void *origin_addr,
                   int origin_count,
                   MPI_Datatype origin_type,
                   int target_rank,
                   MPI_Aint target_displ,
                   int target_count,
                   MPI_Datatype target_type,
                   MPI_Op op,
                   MPI_Win win)
```

- The parameters have the same meaning as the parameters of the `MPI_Put()`-operation.
- additional parameter op specifies reduction operation to be used, compare with reduction operations for `MPI_Reduce()` **no** user definable reduction operations

Restrictions of one sided communication I

- Each memory operation in a window is only allowed to be the target of one one sided communication operation at any time in program execution $\longrightarrow$
  i.e. a concurrent access of a memory location by different processes is not allowed
- Exception: several `MPI_Accumulate`-operations could be active for the same memory location at the same time
  $\longrightarrow$
  Results of the operation = any order of the executed operations (commutative reduction operations guarantee always the same result)
- A window of a process P is not allowed to be accessed by `MPI_Put()`- or `MPI_Accumulate()`-operations of a different process and by a modification of local write-operation from P at the same time (also not if different memory locations of the same window are accessed).

Overview

Global Synchronization I

Global synchronization of a process group of a window

- Suitable for regular applications with alternating
  - global computation phases and
  - global communication phases

Global Synchronization Operation:

```
int MPI_Win_fence (int assert, MPI_Win win)
```

- Has to be called by all processes of the process group of the window `win`.
- A calling process continues only then with the following instruction, if **all** from this processes issued and on the window win working **one sided communication operations** have finished
- The parameter `assert` can be used to specify the context of the call to `MPI_Win_fence()`, which may be used by the runtime system to do optimizations. (Normal case: no additional information, i.e. `assert = 0`)

Example I

- Iterative computation using a distributed data structure A.
- Per iteration step: Each participating process
  - updates its local part of the data structure (update())
  - copies parts of data in continuous buffer(update_buffer())
  - provides parts of its data structure to the neighbor processes (MPI_Put())
  - MPI_Win_fence() used before and after communication phases

```
while (!converged (A)) {
update(A);
update_buffer(A, from_buf);
MPI_Win_fence (0, win);
for (i=0; i<num_neighbors; i++)
    MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i],
             to_disp[i], size[i], MPI_INT, win);
MPI_Win_fence (0, win);
}
```

Loose Synchronization I

Synchronization for the accessing processes and the process whose window is accessed.

- Accessing process specifies the begin and end of an access epoch to the windows of other processes of the process group the accessing process is also part of by using `MPI_Win_start()` or `MPI_Win_complete()`
- Processes whose window is accessed specify the **begin** and the **end** of the **exposure epoch** by using `MPI_Win_post()` and `MPI_Win_wait()`
- Synchronization between `MPI_Win_start()` and `MPI_Win_post()`
  $\longrightarrow$
  RMA-operation of the accessing process (after `MPI_Win_start()`) only after the target process has finished its `MPI_Win_post()`-operation.
- Synchronization between `MPI_Win_complete()` and `MPI_Win_wait()`
  $\rightarrow$
  RMA-operation of the accessing process are finished before the target process has finish the `MPI_Win_wait()`-operation

Operations for Loose Synchronization I

-
```
int MPI_Win_start(MPI_Group group,
                  int assert,
                  MPI_Win win)
```
- `int MPI_Win_complete (MPI_Win win)`
-
```
int MPI_Win_post(MPI_Group group,
                 int assert,
                 MPI_Win win)
```
- `int MPI_Win_wait (MPI_Win win)`
- `int MPI_Win_test (MPI_Win win, int *flag)`
  `flag = 1` if all RMA-operations have finished on the window win
  `flag = 0` if not all RMA-operations have finished on the window win

Example: Loose Synchronization

```
while (!converged (A)) {
  update (A);
  update_buffer(A, from_buf);
  MPI_Win_start (target_group, 0, win);
  MPI_Win_post (source_group, 0, win);
  for (i=0; i<num_neighbors; i++)
       MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i],
                to_disp[i], size[i], MPI_INT, win);
  MPI_Win_complete (win);
  MPI_Win_wait (win);
}
```

- source_group are the neighbor processes accessing the local window of the actual process
- target_group are the neighbor processes whose window is accessed by the actual process
- Each process has different neighbors $\longrightarrow$ weaker synchronization

Lock Synchronization

Synchronization mechanism where only the accessing processes participate actively on the access (Shared Memory Model)

- two processes can communicate using RMA-accesses to the window of a third process,
  without active involvement of the third process, e.g. by actively giving access to the window
- **Methods**:
  Setting a lock before accessing
  and releasing a lock after access
  `MPI_Win_lock()` and `MPI_Win_unlock()`
- Two different lock-mechanism (parameter `lock_type`):
  - exclusive lock by using `lock_type = MPI_LOCK_EXCLUSIVE`
    $\longrightarrow$ suitable for modifications with `MPI_Put`
  - shared lock, by using `lock_type = MPI_LOCK_SHARED` $\longrightarrow$ good for accesses with `MPI_Get()` or `MPI_Accumulate()`

Lock-Operations

```
int MPI_Win_lock(int lock_type,
                 int rank,
                 int assert,
                 MPI_Win win)


int MPI_Win_unlock(int rank,
                   MPI_Win win)
```

Example: Lock-Synchronization

Each access to the window of another process is protected by an **exclusive lock**

```
while (!converged (A)) {
  update (A);
  update_buffer(A, from_buf);
  MPI_Win_start (target_group, 0, win);
  for (i=0; i<num_neighbors; i++) {
    MPI_Win_lock (MPI_LOCK_EXCLUSIVE, neighbor[i], 0, win);
    MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i],
    to_disp[i], size[i], MPI_INT, win);
    MPI_Win_unlock(neighbor[i], win);
  }
}
```