Rechnerarchitektur SS 2014

Verbindungsnetze

Michael Engel

TU Dortmund, Fakultät für Informatik

Teilweise basierend auf Material von Gernot A. Fink und R. Yahyapour

3. Juli 2014

Leistungsbewertung von Kommunikationsmechanismen

Drei Bewertungsmaße wichtig für jeden Kommunikationsmechanismus:

1. *Kommunikations-Bandbreite*

   Im Idealfall nur abhängig von ...

   - Prozessoren, ...
   - Speicher und ...
   - Verbindungsbandbreite,

   aber *unabhängig* vom Kommunikationsmechanismus.

   I.d. Realität: Kommunikation beansprucht Ressourcen in einem Knoten / im Verbindungsnetz, die für andere Kommunikation nicht zur Verfügung stehen

   $\Rightarrow$ Blockierung / Verzögerung anderer Kommunikation

   $\Rightarrow$ Bandbreite eingeschränkter als im Idealfall (abhängig u.a. von Verbindungsstrukturen)

Leistungsbewertung von Kommunikationsmechanismen II

1. *Kommunikations-Bandbreite* ...
2. *Kommunikations-Latenz*
   Latenz = Overhead d. Senders + Übertragungszeit + Overhead des Empfängers
   - Overhead (Software/Hardware) durch Kommunikationsmechanismus bestimmt (entsteht u.a. durch "Adressierung" der übertragenen Daten)
   - Latenz beeinflußt sowohl Performanz als auch Einfachheit der Programmierung eines MP-Systems (je nach verwendetem Verfahren Aufwand unterschiedlich)
3. *Verstecken von Kommunikationslatenz*
   - Ziel: Überlappung von Kommunikationsereignissen mit Berechnungen / anderer Kommunikation durch Kommunikationsmechanismus (z.B. durch spekulative Ausführung, nicht blockierende Caches, reduzierte Anforderungen an Speicherkonsistenz)
   - Umfang der "versteckbaren" Latenz anwendungsabhängig!

Leistungsbewertung von Kommunikationsmechanismen III

1. *Kommunikations-Bandbreite* ...
2. *Kommunikations-Latenz* ...
3. *Verstecken von Kommunikationslatenz* ...

Leistungsgrößen beeinflußt durch Charakteristiken der erforderlichen Inter-Prozessor-Kommunikation

- Datengröße $\Rightarrow$ Bandbreite, Latenz (indirekt auch Auswirkung auf "versteckbare" Latenz)
- Regelmäßigkeit der Kommunikation $\Rightarrow$ Overhead bei Sender und Empfänger (z.B. wegen "Benennung" der transferierten Daten)

Kommunikationsmechanismen

Distributed Shared-Memory Kommunikation

- Kompatibilität mit Mechanismen, die für UMA-Architekturen (gemeinsamer phys. Speicher) benutzt werden (Standardisierungsbestrebungen: z.B. OpenMP)
- Leichtere Programmierung komplexerer / variierender Kommunikationsmuster
- Leichtere Anwendungsentwicklung durch Verwendung des bekannten shared-memory Modells
- Kleinerer Overhead und bessere Bandbreitenausnutzung für kleine Übertragungseinheiten (Grund: Speicherabbildung und Adressumsetzung in Hardware)
- Möglichkeit, hardware-kontrolliertes Caching häufig benutzter Daten zu verwenden

Kommunikationsmechanismen II

Message Passing (Kommunikation mit Nachrichten)

- Einfachere Hardware (keine hardware-mäßige Abbildung des verteilten Speichers, kein Caching mit Notwendigkeit, Kohärenz sicherzustellen)
- Kommunikation ist explizit (Senden einer Nachricht) und dadurch ggf. einfacher zu verstehen (Shared-Memory-Kommunikation: unklar, wann kommuniziert wird und wieviel Overhead entsteht)

$\Rightarrow$ Fokussiert Aufmerksamkeit des Entwicklers auf diesen aufwendigen Teil parallelen Rechnens?!

- Synchronisierung assoziiert mit Nachrichtenaustausch

$\Rightarrow$ weniger Fehler durch fehlende / falsche Synchronisation?!

- Einfacher, sender-initiierte Kommunikation zu realisieren

Verbindungsstrukturen

... für MP-Architekturen mit verteiltem Speicher

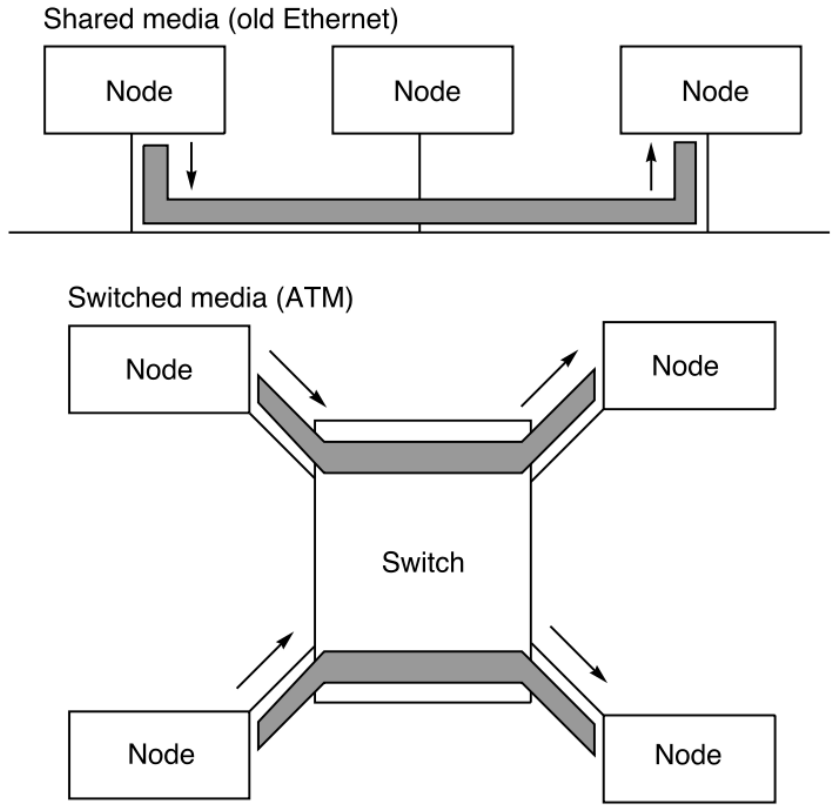
- Verbindungsstrukturen entsprechen prinzipiell Strukturen von Rechnernetzen $\Rightarrow$ Architekturen spezieller
- Folgende prinzipiellen Architektureigenschaften unterscheidbar:
  - Direkte vs. indirekte Verbindung
  - Shared vs. switched Medium
- Direkte Verbindung mit shared Medium
  - Entspricht im Prinzip Bussystem, das von mehreren Rechnern bzw. Prozessorknoten verwendet wird
  - Beachte: Arbitrierung erforderlich (d.h. Entscheidung, wer wann Medium benutzt)
- Direkte Verbindung, switched Medium ...

Verbindungsstrukturen II

- Direkte Verbindung mit shared Medium ...
- Direkte Verbindung, switched Medium
  - Punkt-zu-Punkt Kommunikation mit voller Bandbreite via “switch” (d.h. Verbindungsumschalter)
  - i.d.R. gleichzeitig zwischen verschiedenen Knotenpaaren möglich (Switch interpretiert Adressen der Kommumikationspakete)
  - Nachteile:
    - Overhead durch “switching” (setzen der Umschalter erfordert [Lauf-]Zeit!)
    - Hardwareaufwand im Switch (Realisierung der Schalter)
- Indirekte Verbindung
  - Keine direkte Netzwerkverbindung zwischen allen Knoten
  - i.d.R. Verbindung von Knoten mit best. Anzahl von Nachbarknoten via Switch $\Rightarrow$ verteilte Switches

Verbindungsstrukturen III

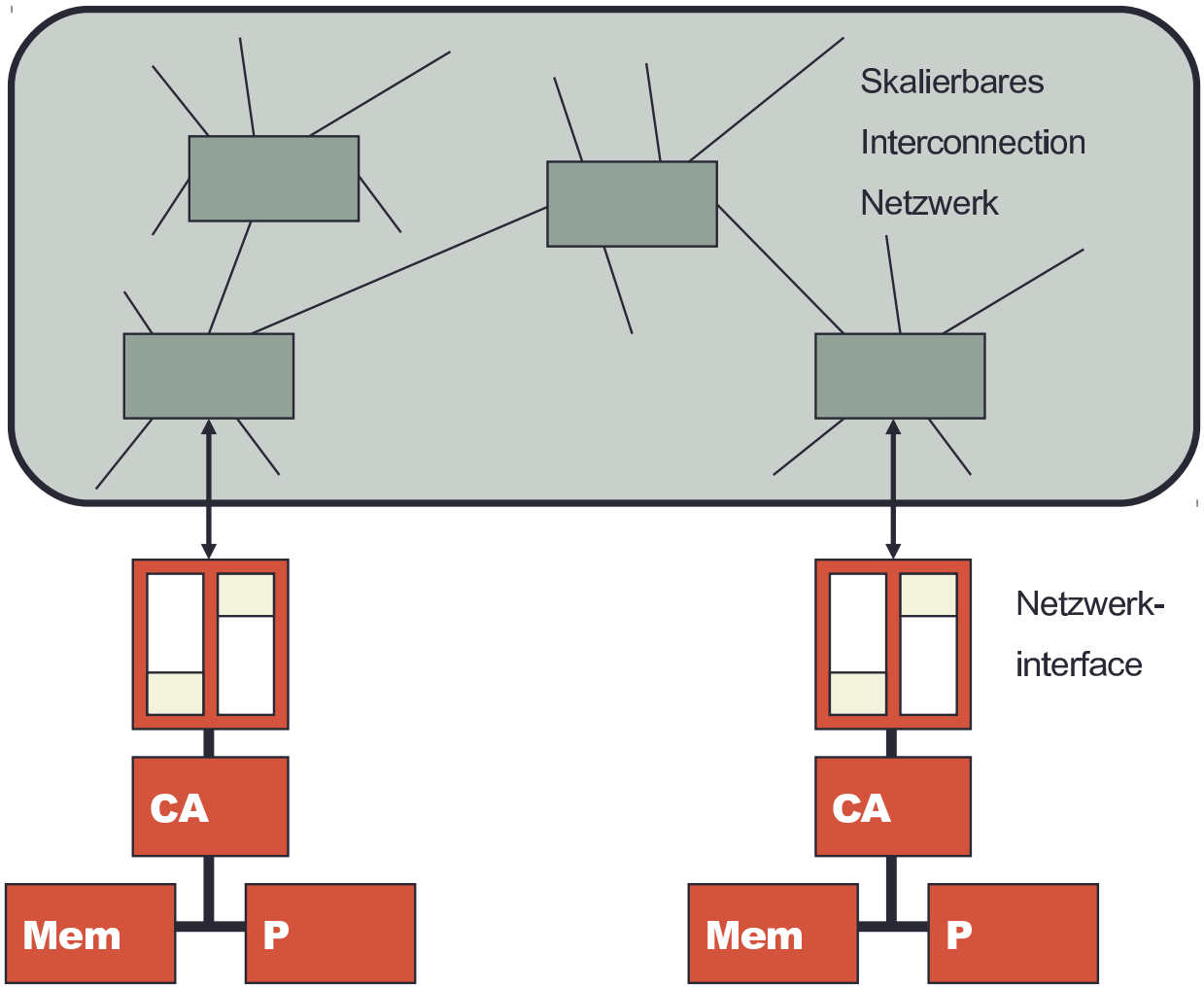
- Shared vesus switched Medium



- Shared Medium: z.B. Original Ethernet, SCSI
- Heute: Durch “strukturierte Verkabelung” Punkt-zu-Punkt Verbindungen auch im LAN-Bereich (Ethernet)

Verbindungsstrukturen IV

Allgemeine Verbindungsarchitektur für DSM-MP-System



Quelle: R. Yahyapour, ehem. TU Dortmund, nach Culler/Singh/Gupta, 1999

Verbindungsstukturen V

- Verbindungsnetz kann definiert werden als Graph bestehend aus
  - *Rechenknoten*,
  - *Kommunikationsknoten* (= Switches) und
  - *Kommunikationskanälen*.
- Netzwerk-Topologie beschreibt Verbindungsstruktur zwischen Knoten mit Kommunikationsfähigkeiten (d.h. i.d.R. *Kommunikationsknoten*)
- *Kommunikationsknoten* i.d.R. als Switches ausgelegt

Verbindungsstukturen V

- Verbindungsnetz kann definiert werden als Graph bestehend aus
  - *Rechenknoten*,
  - *Kommunikationsknoten* (= Switches) und
  - *Kommunikationskanälen*.
- Netzwerk-Topologie beschreibt Verbindungsstruktur zwischen Knoten mit Kommunikationsfähigkeiten (d.h. i.d.R. *Kommunikationsknoten*)
- *Kommunikationsknoten* i.d.R. als Switches ausgelegt

Behandeln zunächst Topologien, dann Struktur von Switches

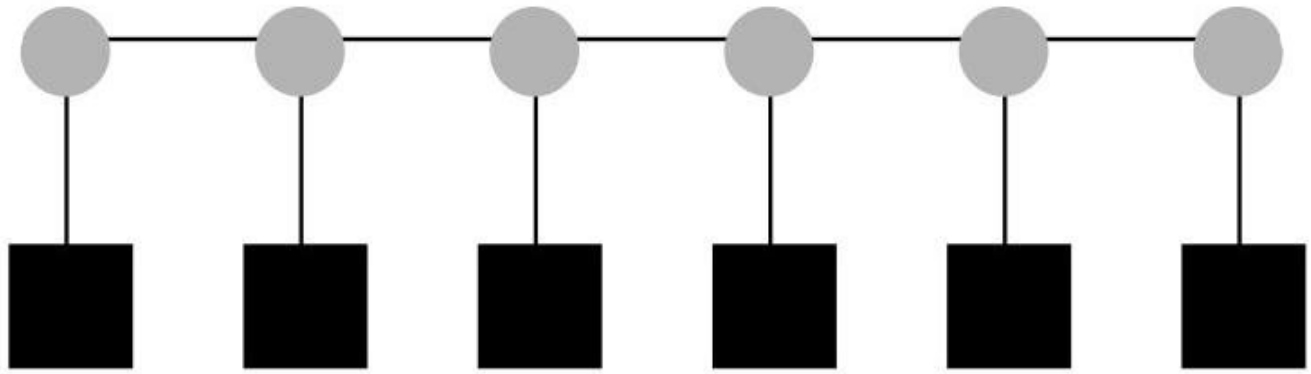
Bewertungskriterien für Netzwerke

- *Knoten-Grad*:
  Anzahl der mit einem Knoten oder Schalter verbundene Kanäle (betrachtet werden hier i.d.R. nur Kommunikationsknoten!)
- *Grad eines Netzwerkes*:
  Der maximale Knotengrad im Netzwerk
- *Durchmesser*:
  Länge des maximalen kürzesten Pfades in dem Netzwerk zwischen zwei beliebigen Knoten
- *Halbierungsbandbreite* (*bisection bandwidth*):
  - Die minimale Anzahl der gerichteten Kanten in einem Netzwerk zwischen zwei gleichen Hälften des Netzwerkes bei beliebiger Unterteilung, bzw.
  - Anzahl paralleler Verbindungen zwischen zwei Netzwerkhälften

Topologien

Topologien

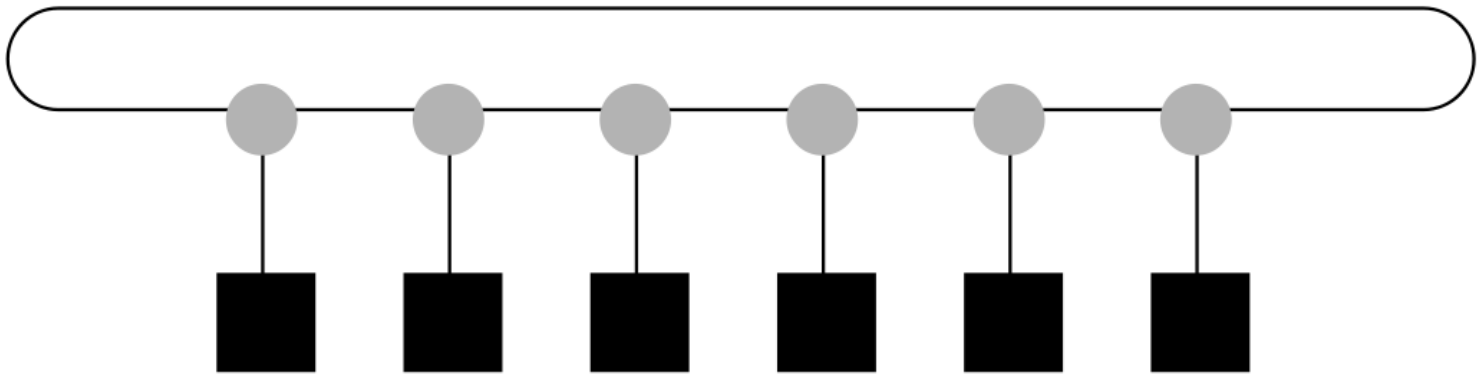
Lineare Felder / Arrays



Grad der Knoten:	2 für alle inneren Knoten
	1 für Randknoten
Durchmesser:	$N - 1$
Halbierungsbandbreite:	1 (i.S. Anzahl d. Links)
Skalierbarkeit:	keine Hardwarebeschränkung

Topologien II

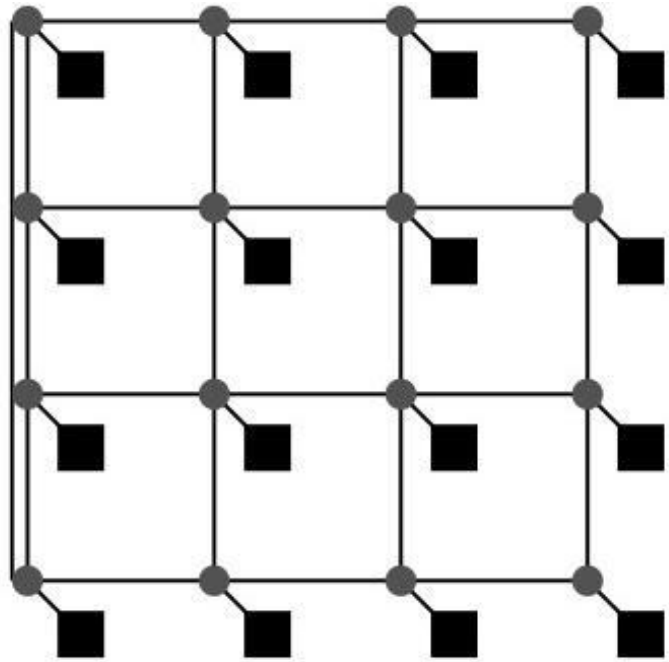
Ring-Topologien



Grad der Knoten:	2
Durchmesser:	$\lfloor N/2 \rfloor$ im bidirektionalen Fall
	$N-1$ im unidirektionalen Fall
Halbierungsbandbreite:	2 (i.S. Anzahl d. Links)
Skalierbarkeit:	keine Hardwarebeschränkung

Topologien III

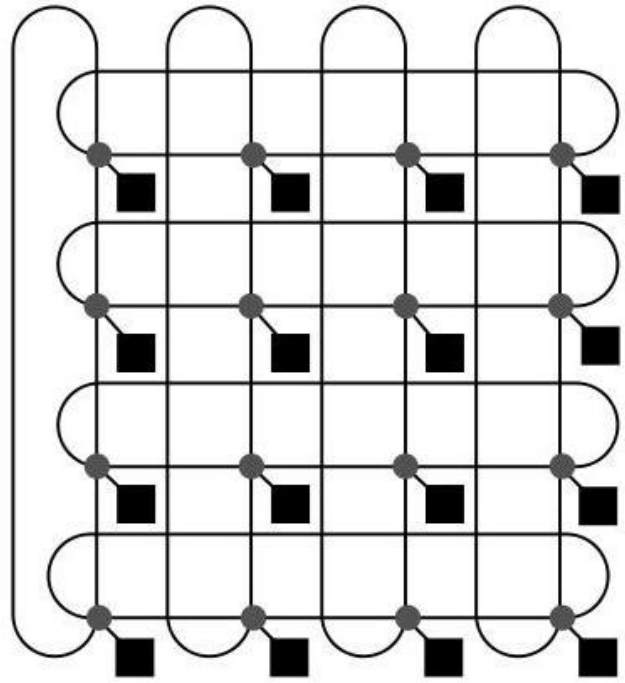
Gitter / Meshes
(in 2 und k Dimensionen)



	2-D ($N = 16$)	k-D	
Grad der Knoten:	2	k	für Eckknoten
	≤ 4	$\leq 2k$	für innere Knoten
Durchmesser:	6	$k\left(\sqrt[k]{N} - 1\right)$	
Halbierungsbandbreite:	4	$\left(\sqrt[k]{N}\right)^{k-1}$	

Topologien IV

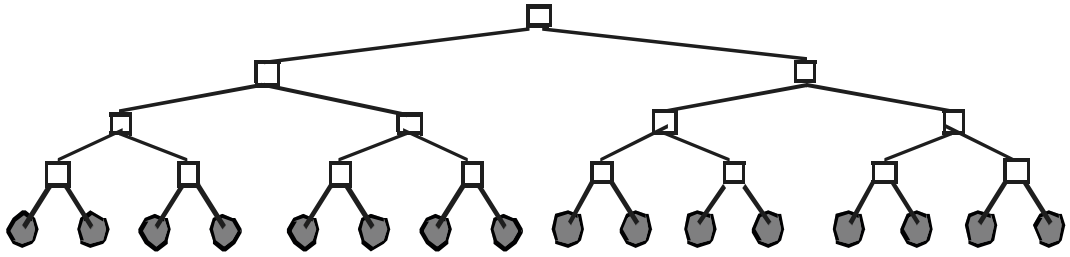
Torus

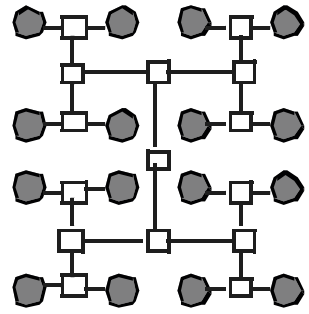


	2-D ($N = 16$)	k-D
Grad der Knoten:	4	$2k$
Durchmesser:	4	$k\lfloor \sqrt[k]{N/2} \rfloor$
Halbierungsbandbreite:	8	$2\left(\sqrt[k]{N}\right)^{k-1}$

Topologien V

Bäume (üblicherweise binär)



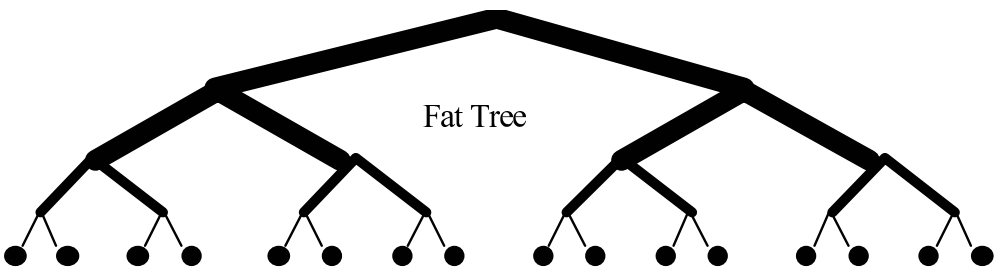


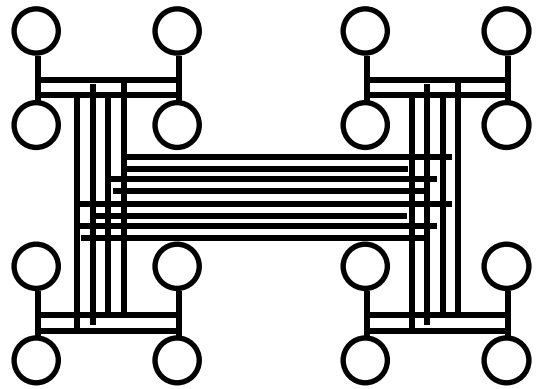
Grad der Knoten:	1 für Blatt-/Endknoten
	2 für Wurzelknoten
	3 für alle inneren Knoten
Durchmesser:	$2 \cdot \lfloor \log_2 N \rfloor$
Halbierungsbandbreite:	1

Hinweis: Bei baumartigen Verbindungsstrukturen sind Rechenknoten nur mit Blattknoten verbunden.

Topologien VI

Fat Trees



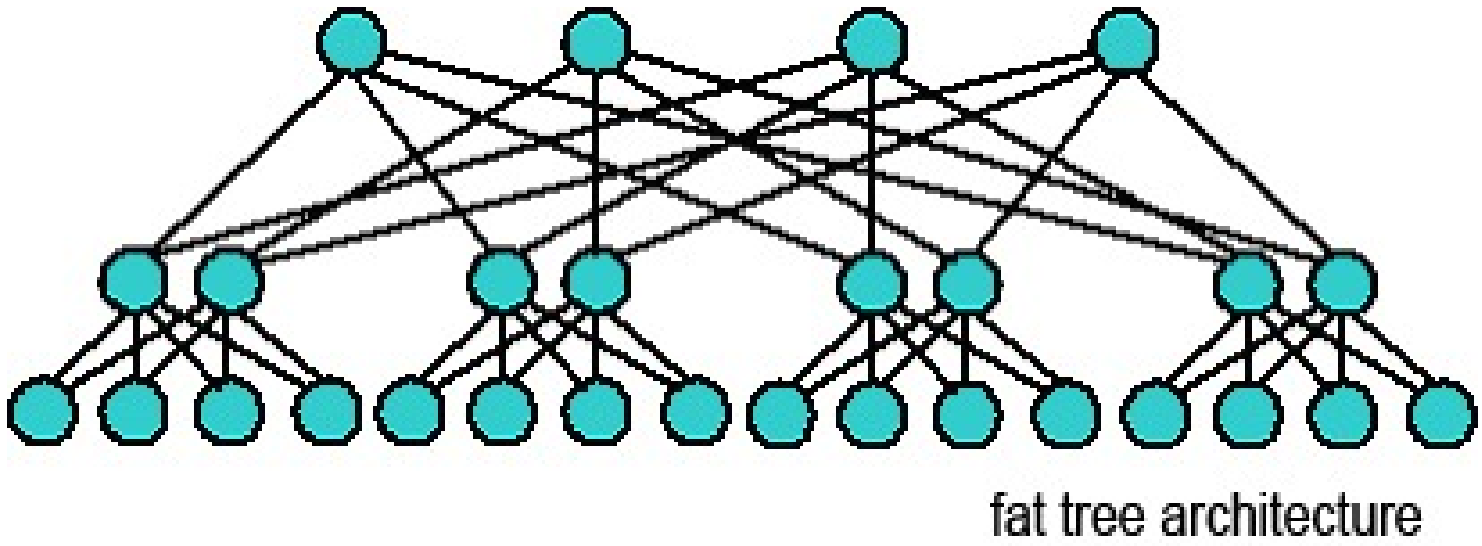


Anzahl der Verbindungen verdoppelt sich mit jeder Ebene im Baum

Grad der Knoten:	1 für Blattknoten (= Rechenknoten)
	$\leq 3 \cdot \lfloor \log_2 N \rfloor - 2$ in der 1. Ebene
Durchmesser:	$2 \cdot \lfloor \log_2 N \rfloor$
Halbierungsbandbreite:	$\log_2 N$

Topologien VII

Redundante Fat Trees

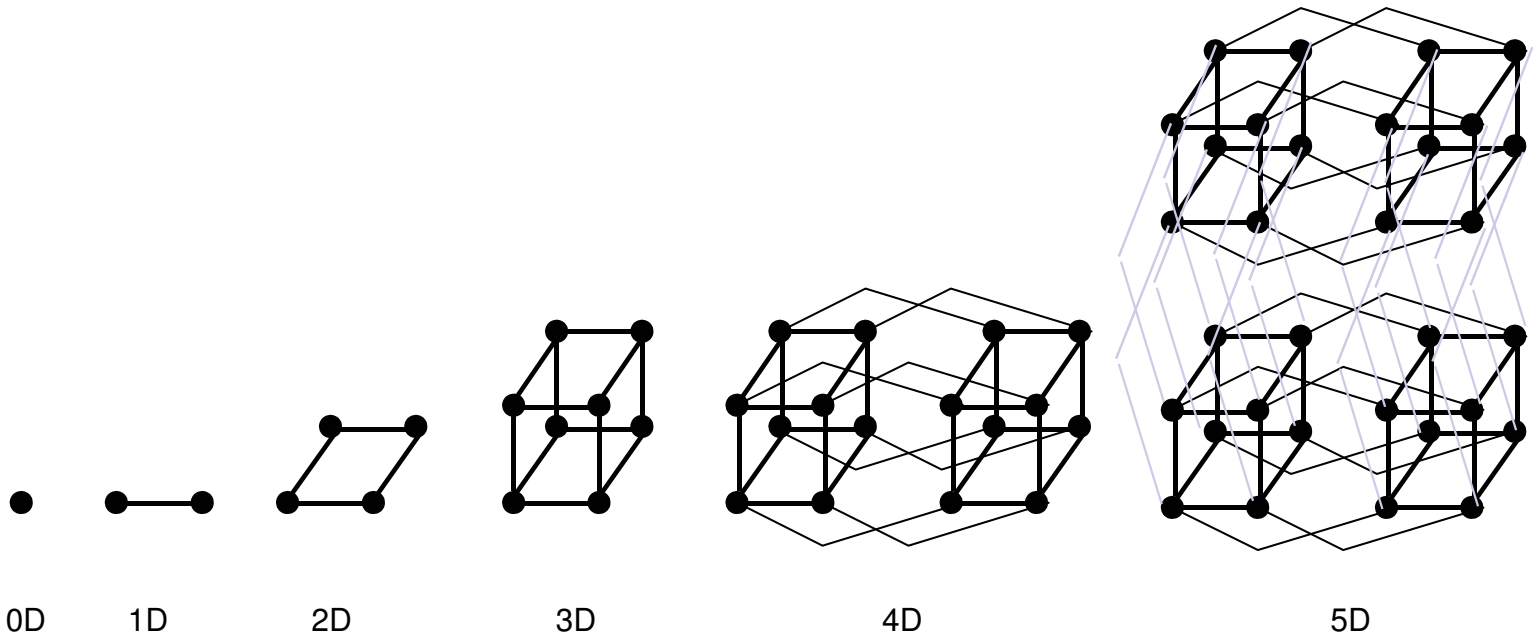


Quelle 24/7 Solutions

Weitere Verbesserung der Redundanz durch Replikation von Kommunikationsknoten im Baum und Verbindung zu *allen* Nachfolgerknoten (insbes. mehrer Wurzeln)

Topologien VIII

Hypercubes (hochdim. Würfel)



Quelle Foliensatz von Culler/Singh/Gupta

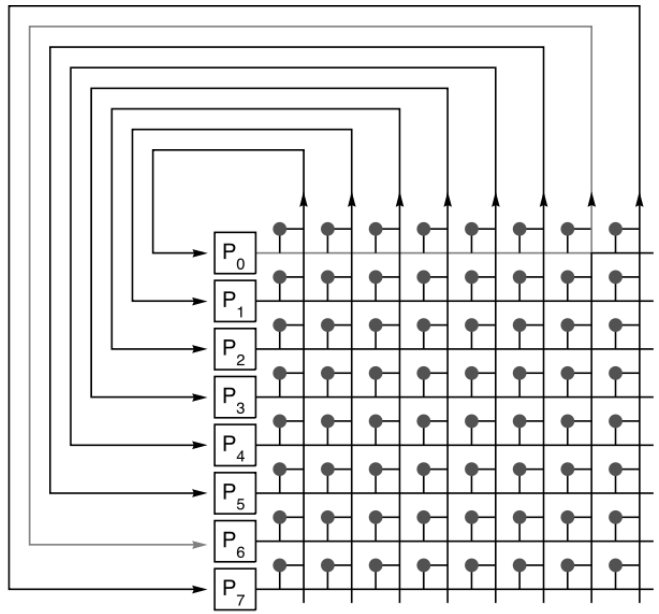
	3-D ($N = 8$)	k-D
Grad der Knoten:	3	$k = \log_2 N$
Durchmesser:	3	k
Halbierungsbandbreite:	4	2^{k-1}

Realisierung von Switches

Realisierung von Switches

Cross Bar

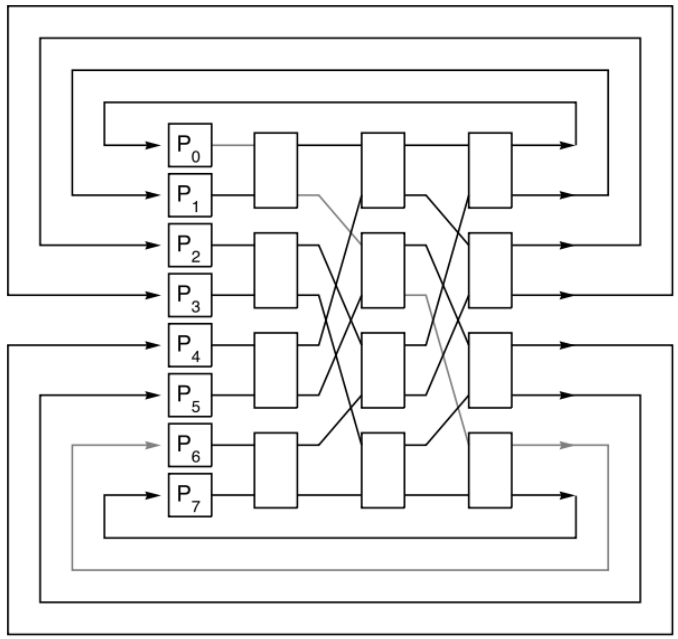
- Schaltung direkter Verbindungen aller möglichen Knotenpaare



- Vorteil: Kommunikation zwischen Knoten via Switch in einem Schritt möglich (nach Schaltung d. Verbindung)
- Nachteil: Extremer Hardware-Aufwand (N^2 Schalter)

Realisierung von Switches II

Omega-Netzwerk



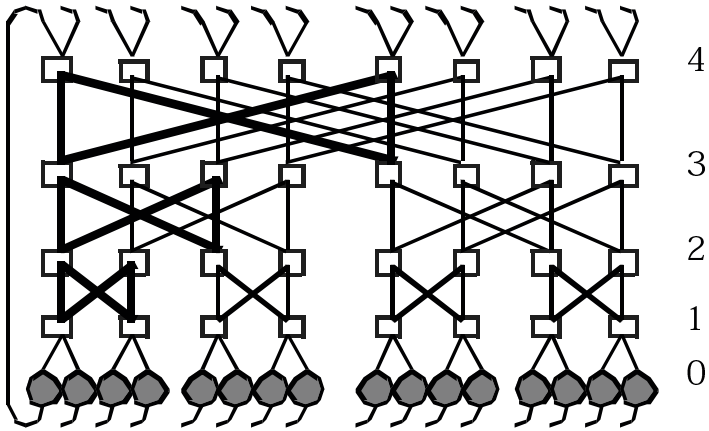
- Schaltung von Verbindungen aller möglichen Knotenpaare in mehreren Stufen

$\Rightarrow$ *multistage interconnection network (MIN)*

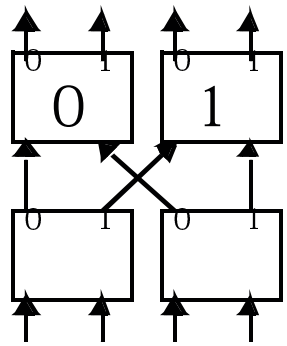
- Vorteil gegenüber Cross Bar: Geringerer Hardware-Aufwand (nur $N/2 \log N$ anstelle N^2 Switch-Elemente)
- Nachteile:
  - Weiterleitung von Daten erfordert mehrere Teilschritte
  - Gegenseitige Blockierung von Datenpfaden möglich

Realisierung von Switches III

Butterfly-Netzwerke



16 node butterfly



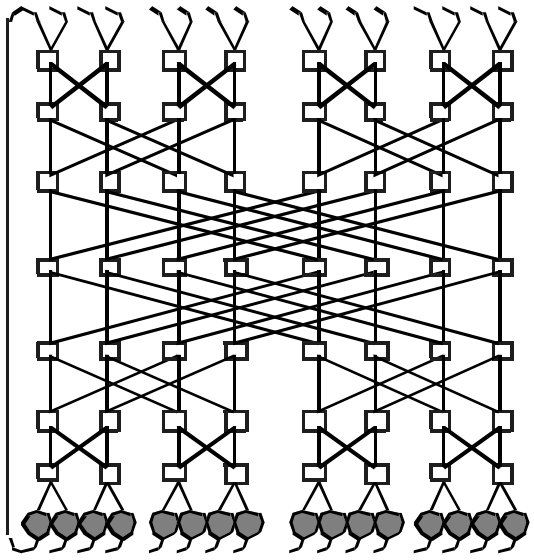
building block

- Prinzip vergleichbar mit Omega-Netzwerk
- Kommunikationswege folgen dem Prinzip des *bit reversal*
- Rekursiver Aufbau:
  Elementares Schaltelement entspricht 2×2 Butterfly
- Logarithmische Tiefe

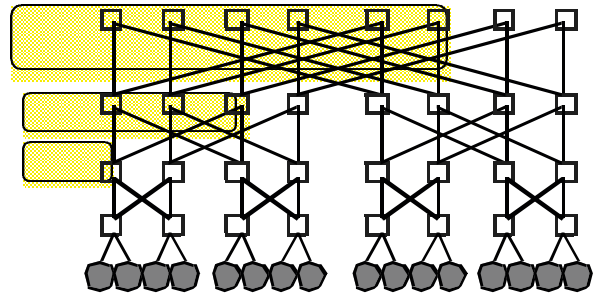
Realisierung von Switches IV

Benes Netzwerke

16-node Benes Network (Unidirectional)



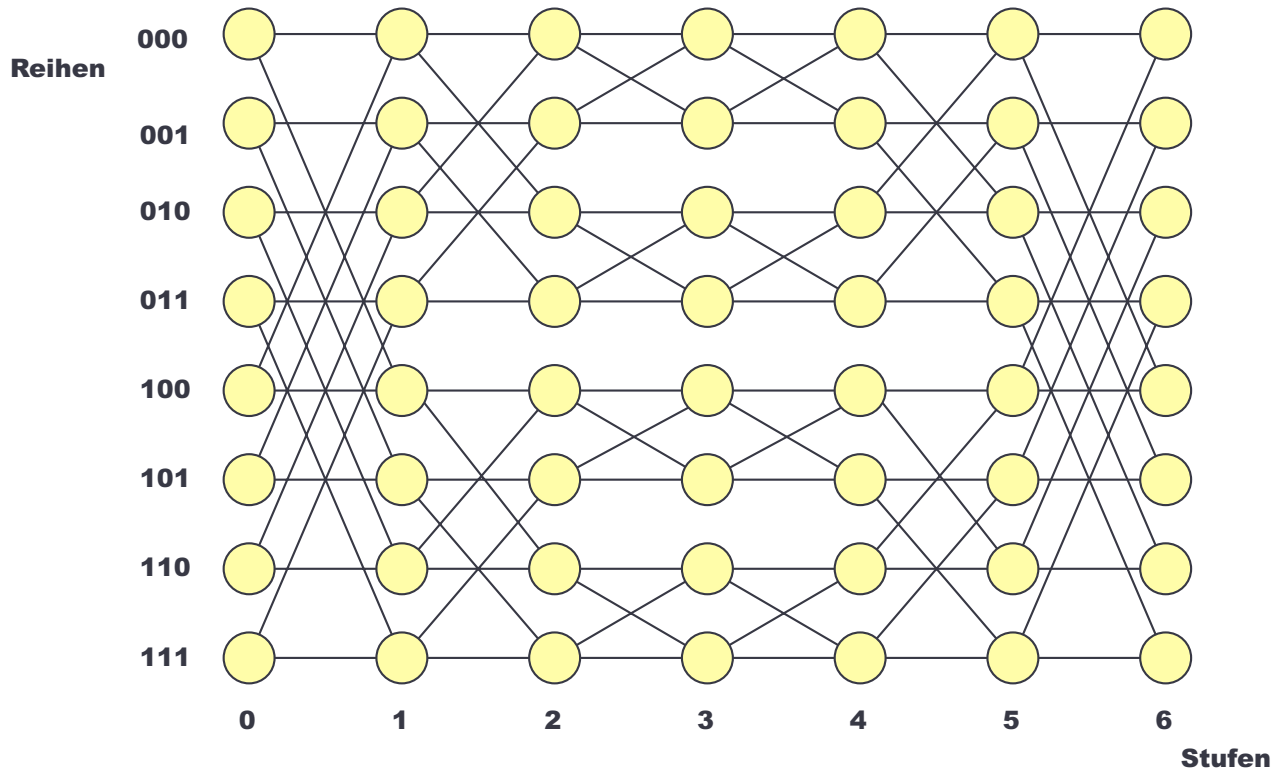
16-node 2-ary Fat-Tree (Bidirectional)



- Aufbau aus zwei rückwärtig verbundenen Butterfly-Netzwerken
- Alle Permutationen von paarweisen Knotenverbindungen schaltbar (bei offline Berechnung der Routen)
- Kann durch “Falten” in redundante Baumstruktur überführt werden (dabei Bildung von “fat nodes”)

Realisierung von Switches V

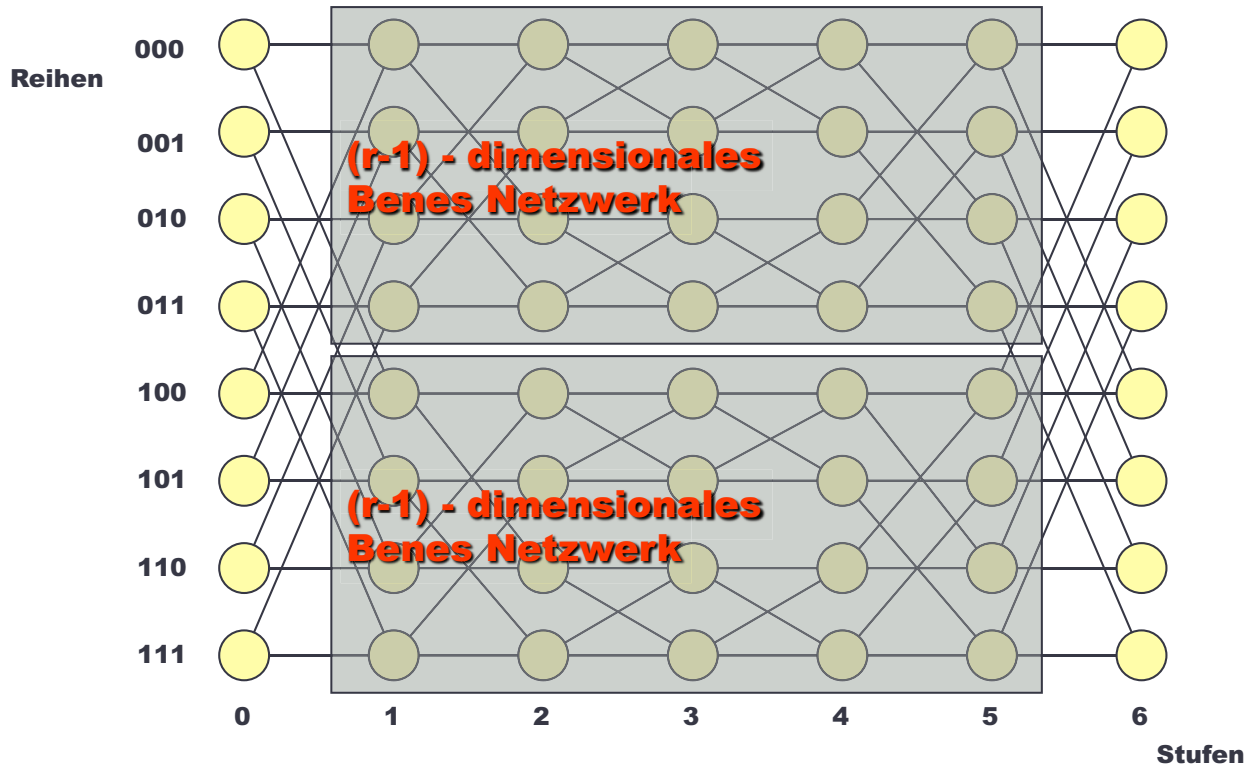
Benes Netzwerke



Quelle: R. Yahyapour, ehem. TU Dortmund

Realisierung von Switches V

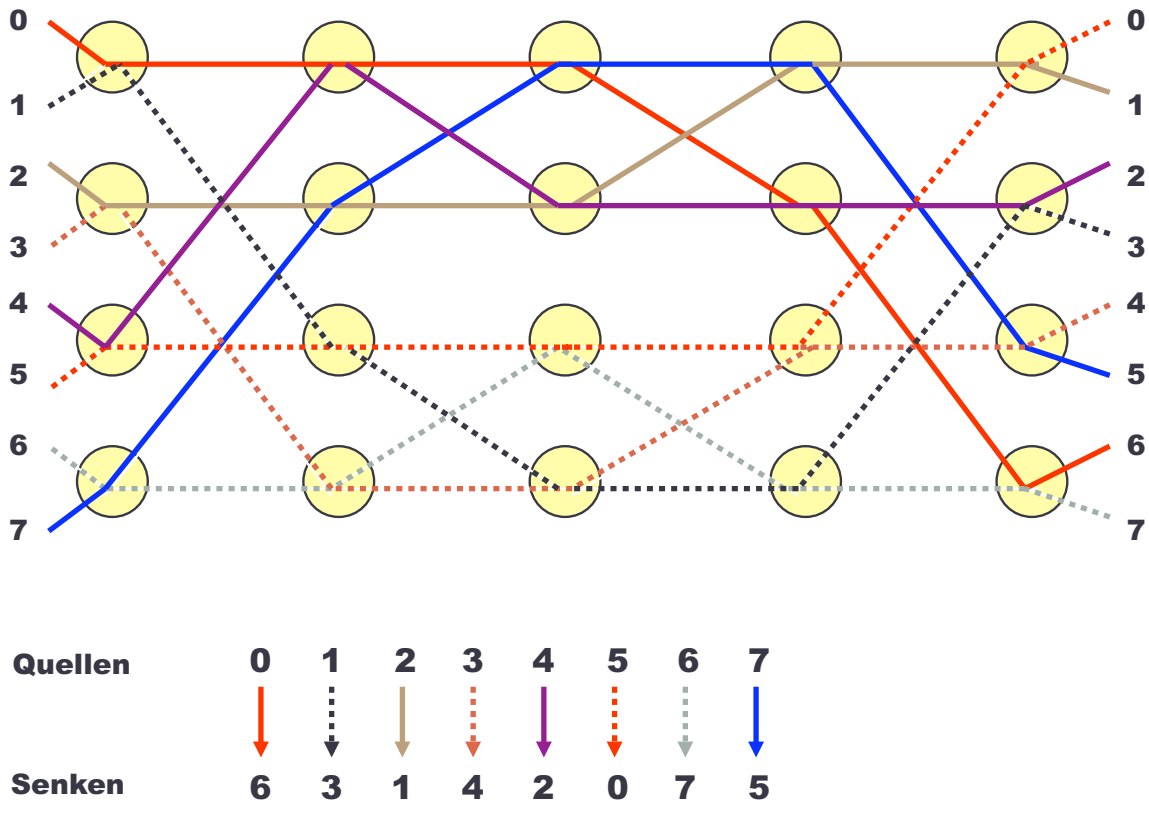
Benes Netzwerke



Quelle: R. Yahyapour, ehem. TU Dortmund

Realisierung von Switches V

Benes Netzwerke



Quelle: R. Yahyapour, ehem. TU Dortmund

Verbindungsstrukturen: Überblick

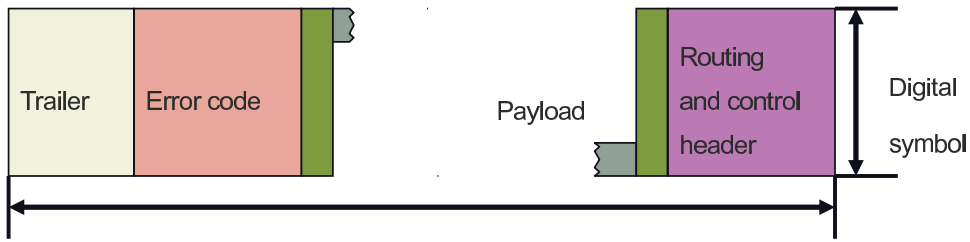
Company	System [network] name	Max. number of nodes [× # CPUs]	Basic network topology	Injection [reception] node BW in MB/sec	# of data bits per link per direction	Raw network link BW per direction in MB/sec	Raw network bisection BW (bidirectional) in GB/sec
Intel	ASCI Red Paragon	4816 [× 2]	2D mesh 64 × 64	400 [400]	16 bits	400	51.2
IBM	ASCI White SP Power3 [Colony]	512 [× 16]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	500 [500]	8 bits (+1 bit of control)	500	256
Intel	Thunder Itanium2 Tiger4 [QsNet^{II}]	1024 [× 4]	Fat tree with 8-port bidirectional switches	928 [928]	8 bits (+2 of control for 4b/5b encoding)	1333	1365
Cray	XT3 [SeaStar]	30,508 [× 1]	3D torus 40 × 32 × 24	3200 [3200]	12 bits	3800	5836.8
Cray	X1E	1024 [× 1]	4-way bristled 2D torus (~23 × 11) with express links	1600 [1600]	16 bits	1600	51.2
IBM	ASC Purple pSeries 575 [Federation]	>1280 [× 8]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	2000 [2000]	8 bits (+2 bits of control for novel 5b/6b encoding scheme)	2000	2560
IBM	Blue Gene/L eServer Sol. [Torus Net.]	65,536 [× 2]	3D torus 32 × 32 × 64	612.5 [1050]	1 bit (bit serial)	175	358.4

Routing

Kommunikation in Verbindungsnetzen

- Prinzip: Datenaustausch erfolgt paketorientiert.
  Typisches Paketformat:

Quelle: R. Yahyapour, ehem. TU Dortmund

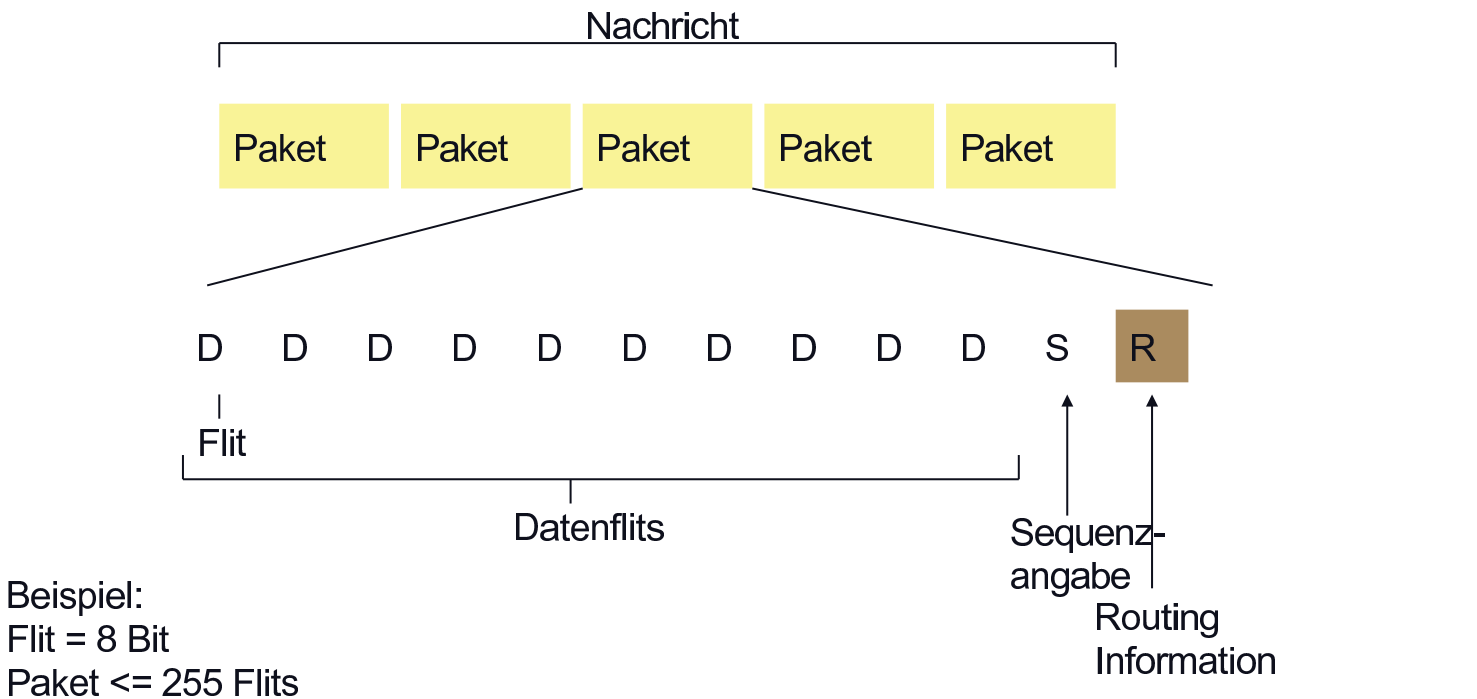


- Zwei grundlegende Mechanismen der Abstraktion:
  - Kapselung: Protokollinformation einer höheren Ebene wird uninterpretiert auf niedrigerer Ebene mit übertragen
  - Fragmentierung: Aufteilung der Nachrichtenstruktur (einer höheren Ebene) in Folge von Übertragunseinheiten

Beachte:
- Abstraktionshierarchie in Parallelrechnern i.d.R. flacher als im LAN/WAN-Bereich
- Greifen enger/effizienter ineinander

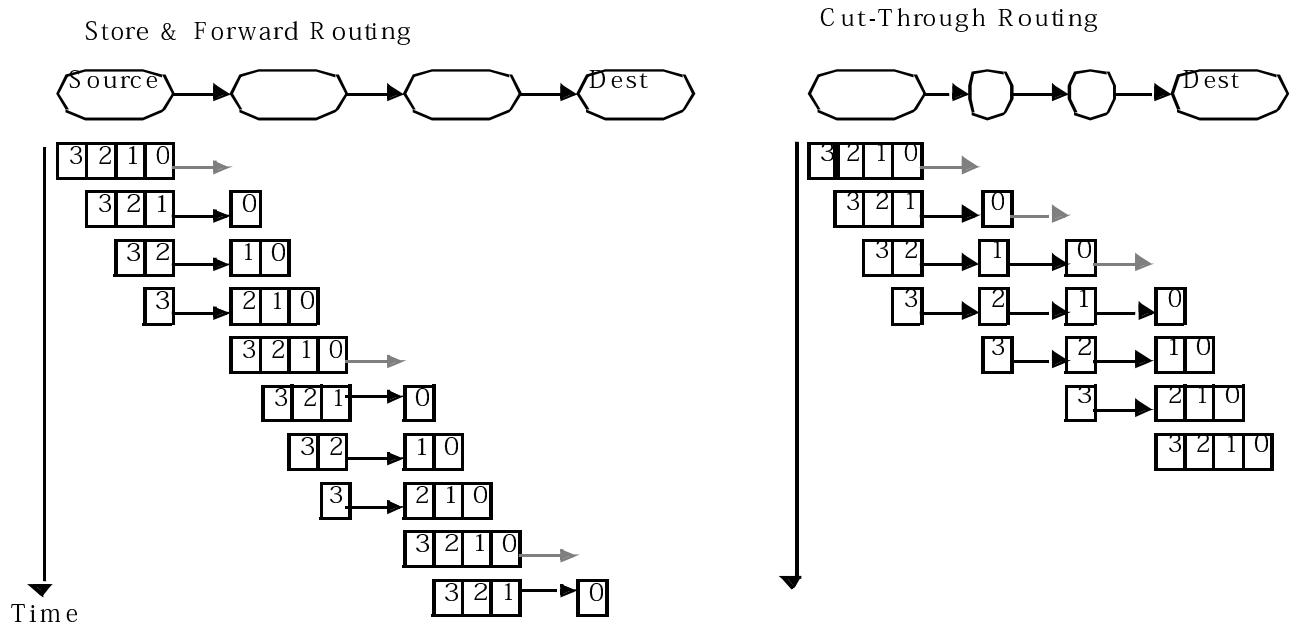
Kommunikation in Verbindungsnetzen II

- "Mikrostruktur" von Datenpaketen: Aufteilung in sog. Flits (*flow control bits*)



Quelle: R. Yahyapour, ehem. TU Dortmund

Routing-Mechanismen



- *Store-and-Forward*: Pakete werden erst weitergeleitet, wenn komplett empfangen
- *Cut-Through Routing*: Filts eines Pakets werden einzeln weiterleleitet; Entscheidung für Routing zum nächsten Knoten fällt bereits, wenn Paketheader bekannt
  $\Rightarrow$ Übertragungslatenz ähnlich Pipelining reduziert!

Routing-Mechanismen II

Grundlegendes Problem von Routing-Verfahren:
Auftreten von Kollisionen

- Mehr als ein Paket/Flit versucht dieselbe Verbindung zw. zwei Knoten zu überqueren
- Kollisionen können in jedem Vermittlungsknoten auftreten

  [im Gegensatz zu *circuit switching*: komplette Route wird zu Beginn der Kommunikation aufgebaut]
- Ein Paket/Flit kann übertragen werden
- Wie andere(s) Paket(e) behandeln?

$\Rightarrow$ Vier Möglichkeiten zur Behandlung von Kollisionen

Routing-Mechanismen III

Möglichkeiten zur Kollisionsbehandlung:

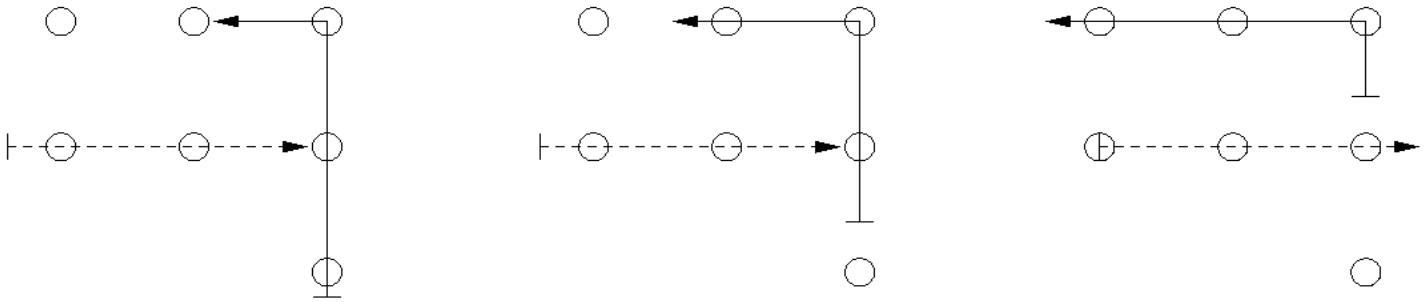
- *Verwerfen* des Pakets (im LAN/WAN-Bereich, insbes. Internet)
  - Netzwerkprotokoll muss Verlust detektieren und ...
  - Neuübertragung initiieren.
- *Puffern* der blockierten Übertragung
  - in Verbindung mit Cut-Through Routing: *Virtual Cut-Through*

    [nachfolgende Flits werden am Blockadepunkt in Puffer "aufgesammelt"]
  - Puffergröße ggf. problematisch!
- *Blockieren* der weiteren Übertragung
  - in Verbindung mit Cut-Through Routing: *Wormhole Routing*

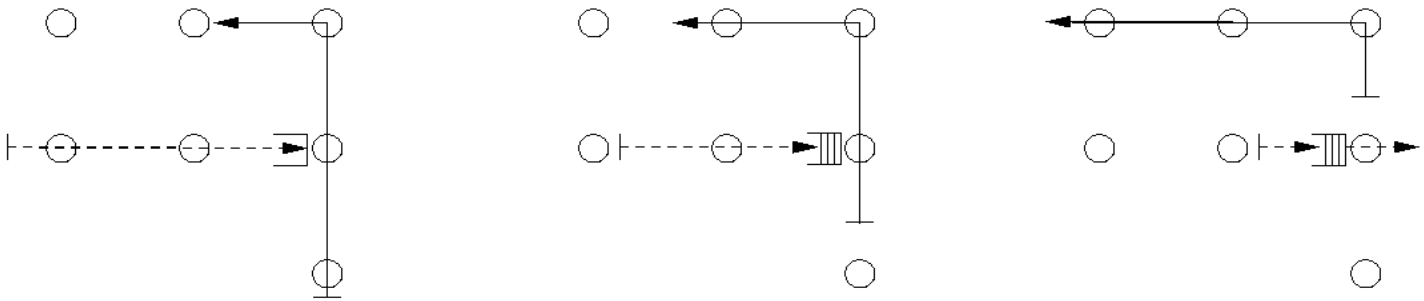
    [nur wenige (im Extremfall: eins) nachfolgende Flits werden am Blockadepunkt gepuffert]
  - Rückstau problematisch

    [blockiert bei Wormhole Routing ggf. n Knoten im Netzwerk, n = Anz. der Flits]
- *Umleitung* von Paketen
  - Erfordert entsprechendes Routing-Verfahren

Routing: Wormhole vs. Virtual Cut-Through



a) Wormhole



b) Virtual-Cut-Through

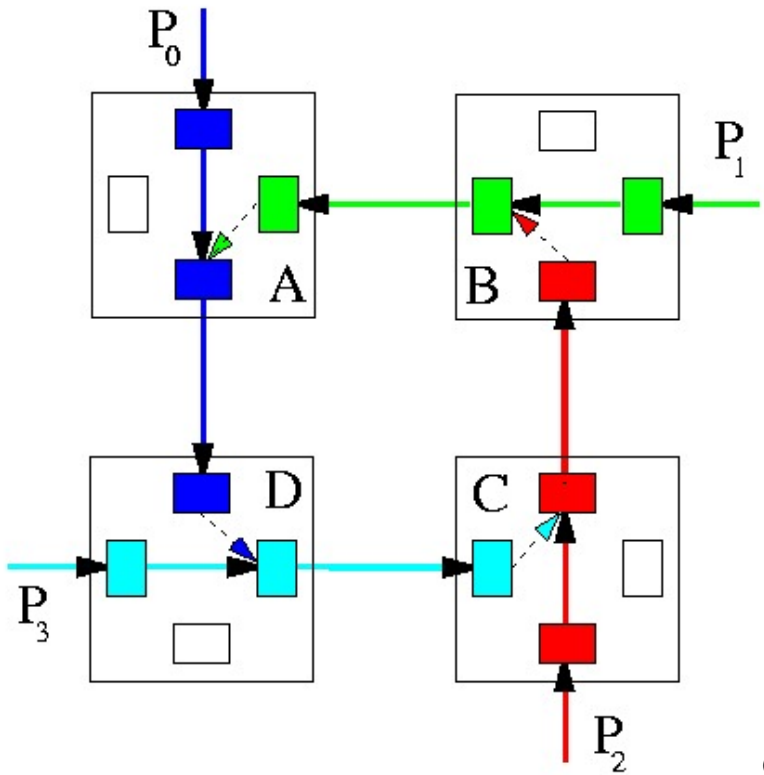
Quelle: Tichy/Pankratius/Jannesari, KIT

Beachte: Nachrichten-Trailer zur Freigabe der Kommunikationsknoten erforderlich

Routing-Mechanismen V

Weitergehendes Problem von Routing-Verfahren: Deadlocks

- Problem vergleichbar der Situation beim gegenseitigen Ausschluss: Für Übertragung von Nachrichten müssen physikalische Kanäle *exklusiv* belegt werden



Quelle: Univ. Wisconsin-Madison

Routing-Mechanismen V

Weitergehendes Problem von Routing-Verfahren: Deadlocks

- Problem vergleichbar der Situation beim gegenseitigen Ausschluss:
  Für Übertragung von Nachrichten müssen physikalische Kanäle *exklusiv* belegt werden
  $\Rightarrow$ Zyklus im Kanalbelegungsgraphen signalisiert Deadlock

Mögliche Lösungen:

- *Virtuelle Kanäle* (und strikte Ordnung bei der Kanalbelegung)
- Einschränkung der gültigen Routen:
  Routingverfahren selbst verhindert Entstehung von Zyklen
  $\Rightarrow$ Up*-Down*-Routing, Turn-Model Routing

Up*-Down* Routing

Verfahren zum *deadlock-freien* (Wormhole-)Routing

Strategie ähnlich zum Routing in einem baumartigen Netzwerk:

- Kanäle zwischen Verbindungsknoten sind bidirektional
- Konstruiere minimalen Spannbaum über die Verbindungsknoten
  $\Rightarrow$ Knoten können nach Position im Baum sortiert werden
- Kanten (d.h. richtungsgebundene Kanäle) werden als *up* bzw. *down* markiert (bzgl. Baum)

$\Rightarrow$ Jeder Zielknoten kann von jedem Quellknoten über Folge von *up* und *down*-Kanten erreicht werden

$\Rightarrow$ Routen werden eingeschränkt auf:

- Zunächst beliebige Folge von *up* Kanten
- Dann "Umkehr" und beliebige Folge von *down* Kanten

Turn-Model Routing

Verfahren zum *deadlock-freien* Routing in gitterartigen Netzwerkstrukturen

Beobachtung: 8 Richtungswechsel ($\hat{=}$ *turns*) in Routen mit bi-direktionalen Kanälen möglich (im 2D-Fall)

Grundidee: Richtungswechsel so einschränken, dass keine Zyklen entstehen können

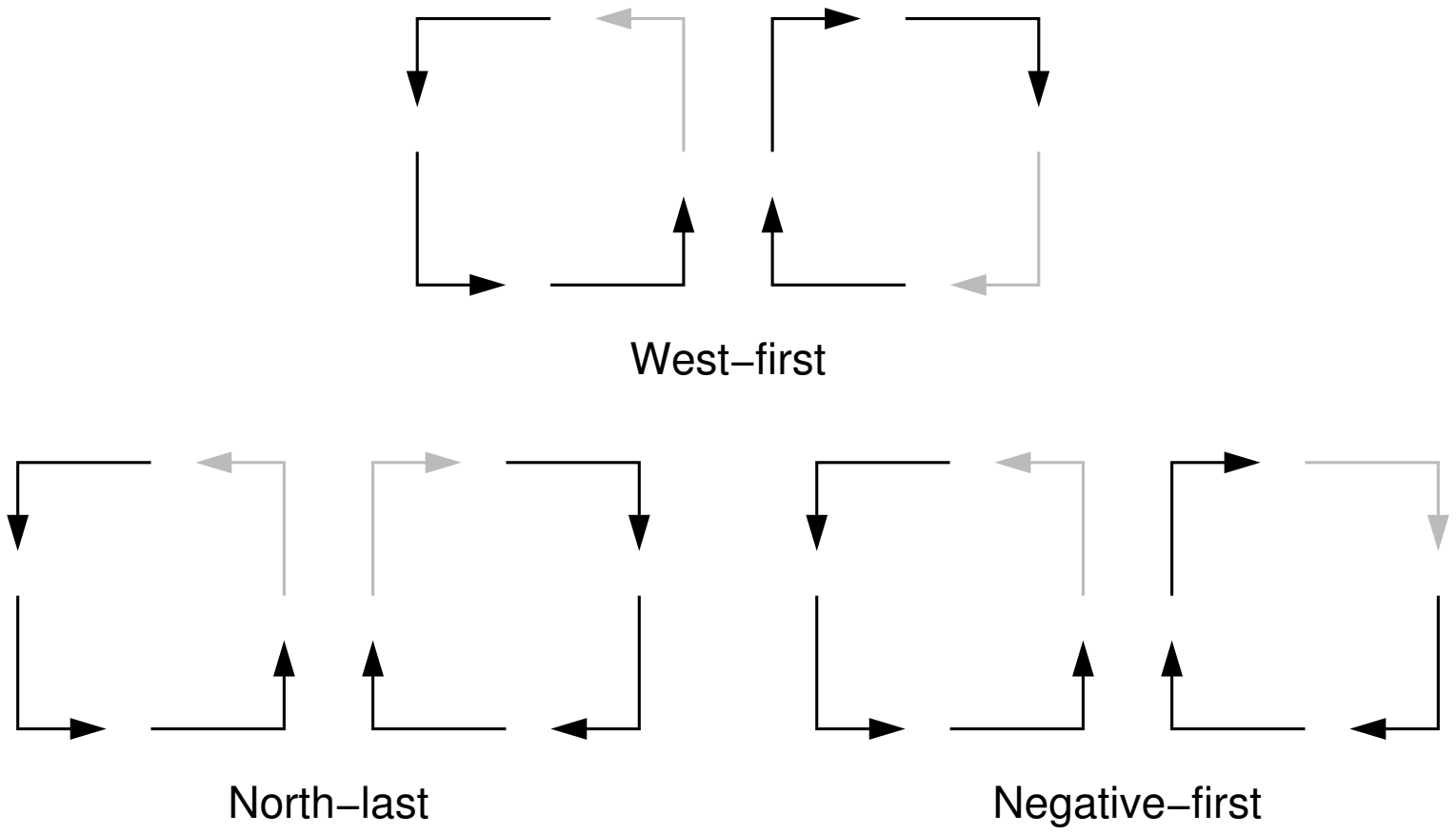
Einfaches Verfahren: *dimension order routing*

- Dimensionen im Gitter werden sortiert (z.B. x vor y vor ...)
- Dann Routing gemäß der Dimensionsordnung vornehmen (d.h. zuerst in $\pm\Delta x$ dann $\pm\Delta y$, ...; dabei Δx etc. $\hat{=}$ rel. Abstand zwischen Quelle und Ziel im Gitter)

Allgemeinere Lösungen: Verhindern von je zwei Richtungswechseln ausreichend für deadlock-freies Routing

Turn-Model Routing II

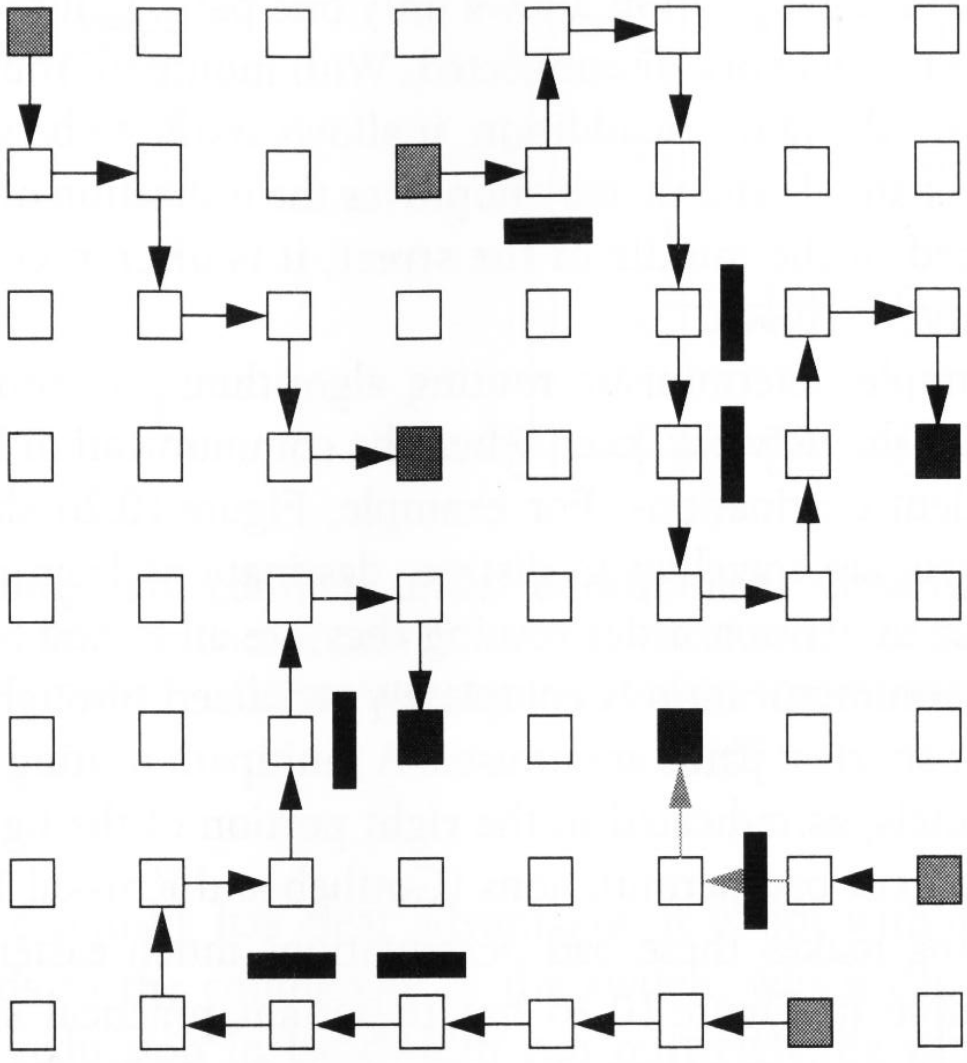
Beispiele für gültige Routingalgorithmen/-einschränkungen in 2D



nach Culler/Gupta/Singh

(jeweils inkl. entsprechend rotierter Schemata)

West-First Routing: Beispiel



MPI

Message Passing: MPI

- MPI $\hat{=}$ Message Passing Interface Standard
- Spezifiziert Bibliotheksschnittstelle für nachrichtenbasierte Kommunikation auf Parallelrechnern (für div. Programmiersprachen, z.B. `C`, `C++`, `Fortran-95`)
- Standardisierung erfolgt durch Gremium (Mitglieder: Hard- und Software-Hersteller, Forschungseinrichtungen u. Anwender)
- Weit verbreitetes API (verfügbar für Mehrzahl aktueller Parallelrechner)
- Ex. frei verfügbare Implementierungen des MPI-Standards:
  - **MPIch2** "... high-performance and widely portable implementation of the MPI Standard"
  - **Open MPI** "... open source MPI-2 implementation'"

MPI: Programmiermodell

MPI verwendet MIMD-Modell der Parallelverarbeitung, d.h.

- Ein MPI-Programm besteht aus mehreren autonomen Prozessen
- Jeder kann individuellen Code ausführen, aber ...
- alle verwenden dasselbe MPI-Programm!
  $\Rightarrow$ tatsächliches Parallelverarbeitungsmodell ist
  SPMD = Single Program, Multiple Data
- Prozesse kommunizieren über Aufrufe der MPI-Kommunikationsprimitive
- Prozesse verwenden normalerweise einen eigenen Adressraum (shared-memory Implementierungen sind aber möglich)

MPI: Nachrichtenaustausch

- Nachrichten $\hat{=}$ Datenpakete, die zwischen Prozessen eines parallelen Programms ausgetauscht werden
- Für Nachrichtenkommunikation sind folgende Informationen erforderlich:
  - Kennung des Senders
  - Kennung des Empfängers
  - Speicheradresse/-ort der Quelldaten
  - Datentyp(!) der Quelldaten
  - Anzahl der gesendeten/empfangenen Daten

MPI: Punkt-zu-Punkt-Kommunikation

Senden/Empfangen von Nachrichten $\hat{=}$ elementarer Kommunikationsmechanismus in MPI: `send` und `receive`

```
#include "mpi.h"
int main( int argc, char **argv ) {
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) /* code for process zero */ {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
        }
    else if (myrank == 1) /* code for process one */ {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
        }
    MPI_Finalize();
    }
```

Quelle: MPI-Forum: “MPI: A Message-Passing Interface Standard Version 2.2”, 9/2009

MPI: Datentypen

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

Daten werden in MPI *typisiert* verschickt/empfangen

- Sende/Emfpangs-Puffer $\hat{=}$ `count` Elemente vom Typ `datatype`
- Datentypen sind *maschinenunabhängig*, Programme portabel
- Ex. verschiedene Basisdatentypen, z.B.:

MPI Datentyp	C Datentyp
`MPI_CHAR`	`char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
...	...
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
...	...

MPI: Abgeleitete Datentypen

- Kommunikation nur mit Folgen elementarer Basistypen bedeutet starke Einschränkung
- Lösung: Definiere allgemeinen Datentyp bestehend aus:
  - Sequenz von elementaren Datentypen
  - Sequenz von Abständnen zwischen Elementen
- Paar von Sequenzen bezeichnet als *type map*

$$\text{Typemap} = \{(\text{type}_0, \text{disp}_0), ..., (\text{type}_{n-1}, \text{disp}_{n-1})\}$$

- Sequenz der elementaren Datentypen (ohne Offsets): *type signature*

$$\text{Typesig} = \{\text{type}_0, ..., \text{type}_{n-1}\}$$

Beachte: Da zusammengesetzte Datentypen im Speicher nicht zusammenhängend sein müssen, können so z.B. Sub-Matrizen o.Ä. definiert werden!

MPI: Nachrichtenformat

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

Meta-Informationen ($\hat{=}$ *message envelope*): Quelle (implizit), Ziel, *tag* und Kommunikator

- Kommunikatoren spezifizieren den Kommunikations*kontext* für MPI-Operationen
- Nachrichten werden immer in dem Kontext enpfangen, in dem sie gesendet wurden
- Nachrichten aus verschiedenen Kontexten interferieren nicht
- Ein Kommunikationskontext wird von einer Prozessgruppe gemeinsam verwendet (im Kommunikator spezifiziert)
- Die Prozessgruppe ist geordnet (gemäß Rang)
- Gültige Werte für `dest` sind damit $0, \ldots n-1$ (mit $n =$ Anzahl der Prozesse i.d. Gruppe)
- Standard-Kommunikator: `MPI_COMM_WORLD` (d.h. Komm. zw. allen Prozessen)

MPI: Kommunikationsmodi

Ex. 4 verschiedene Modi zur Ausführung einer (blockierenden) Sendeoperation:

- *Standard* (`send`): Aufruf terminiert, wenn Nachricht erfolgreich (an MPI-System!) übertragen (d.h. ob gleichzeitig empfangen implementierungsabhängig)
- *gepuffert* (`bsend`), d.h. asynchron: Aufruf terminiert, Nachricht wird von MPI gepuffert
- *synchron* (`ssend`): Aufruf terminiert erst, wenn korrespondierender Empfangsaufruf abgesetzt wurde
- *ready send* (`rsend`): Aufruf kann nur gestartet werden, wenn korrespondierender Empfangsaufruf aktiv

Nach erfolgreichem Terminieren des `xsend` kann Sendepuffer wiederverwendet werden!

MPI: Kommunikationsmodi II

Alle Sendeoperationen können auch *nicht-blockierend* ausgeführt werden:

- Aufspaltung der `send`/`recv`-Aufrufe in Paar aus Initiierungs- und Abschluss-Aufruf

  $$\texttt{send()} \longrightarrow \texttt{isend} + \texttt{wait}$$
  $$\texttt{recv()} \longrightarrow \texttt{irecv} + \texttt{wait}$$

  (Bezug über *request handle*)
- Auswahl des Kommunikationsmodus orthogonal (wird bei Initiierung festgelegt)

  $$\texttt{xsend()} \longrightarrow \texttt{ixsend} + \texttt{wait}$$
- Empfang (und Initiierung davon) unabhängig vom Kommunikationsmodus und davon, ob blockierend bzw. nicht-blockierend gesendet

MPI: Kommunikationsmodi III

Beispiel für nicht-blockierende Kommunikation (in FORTRAN):

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
      CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG,
                     comm, r1, ierr)
      CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
CALL MPI_WAIT(r2, status, ierr)
```

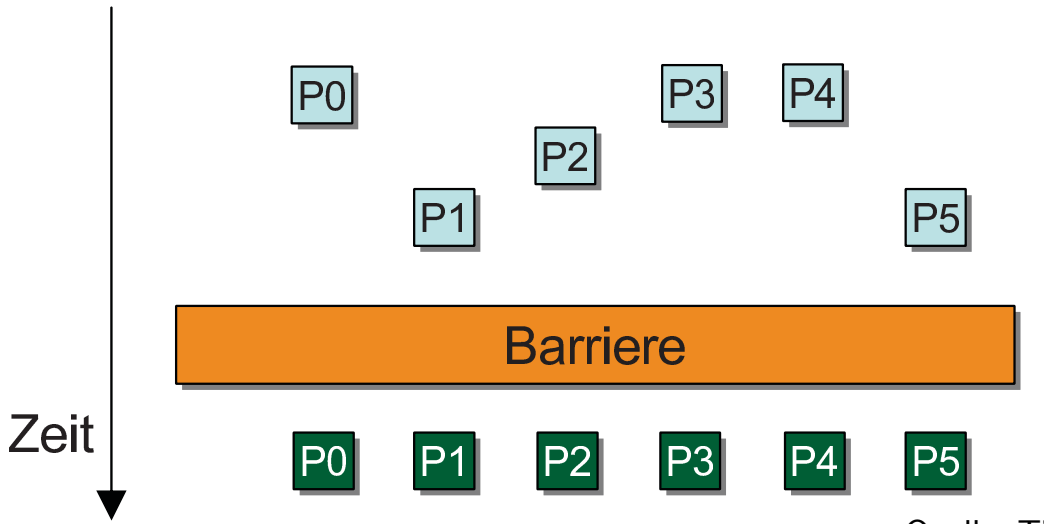
Quelle: MPI-Forum

Beachte: Nicht-blockierende Kommunikationsoperationen werden gemäß der Initiierungsreihenfolge geordnet!

MPI: Kollektive Kommunikation

Barriere : Synchronisation aller Prozesse in einer Prozessgruppe

```
int MPI_Barrier(MPI_Comm comm):
```



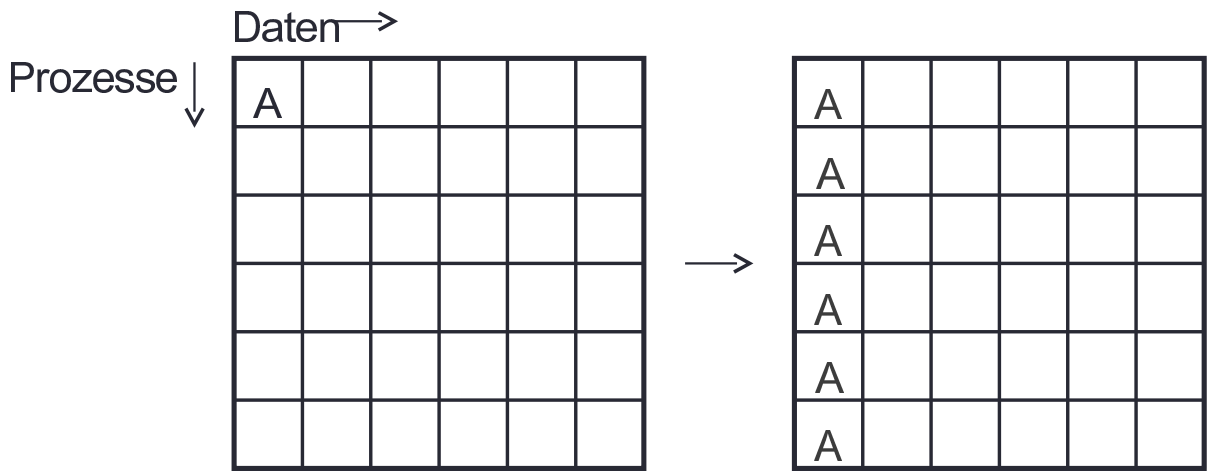
Quelle: Tichy/Moschny, KIT

Achtung: Es wird nicht sichergestellt, dass alle Prozesse die Barriere *gleichzeitig* verlassen!

MPI: Kollektive Kommunikation II

Broadcast : Verteilen von Daten von einem zu allen anderen Prozessen in einer Prozessgruppe

```
int MPI_Bcast(void* buffer, int count,
    MPI_Datatype datatype, int root, MPI_Comm comm);
```

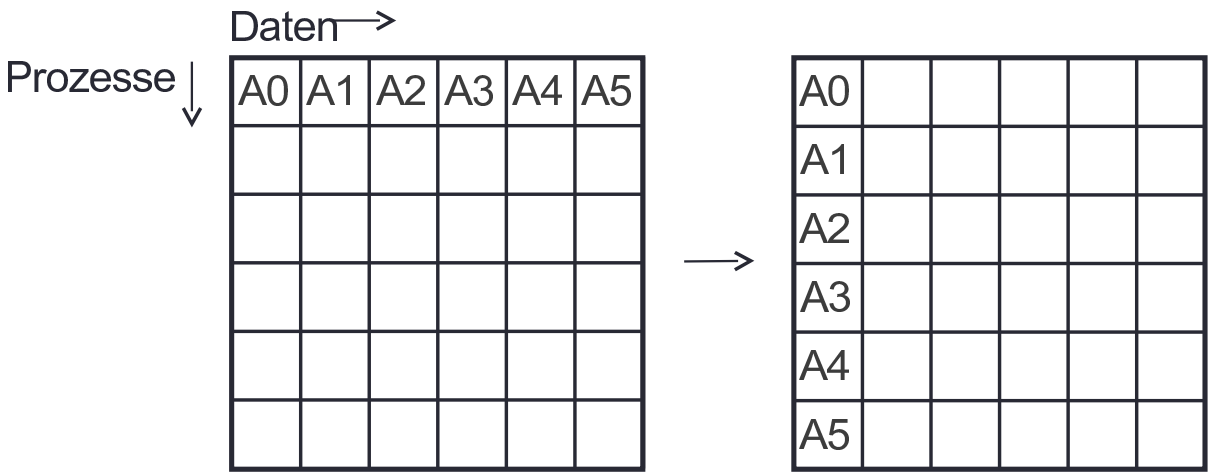


Quelle: R. Yahyapour, ehem. TU Dortmund

MPI: Kollektive Kommunikation III

Scatter : Verteilen von (verschiedenen) Daten auf andere Prozesse in einer Prozessgruppe

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

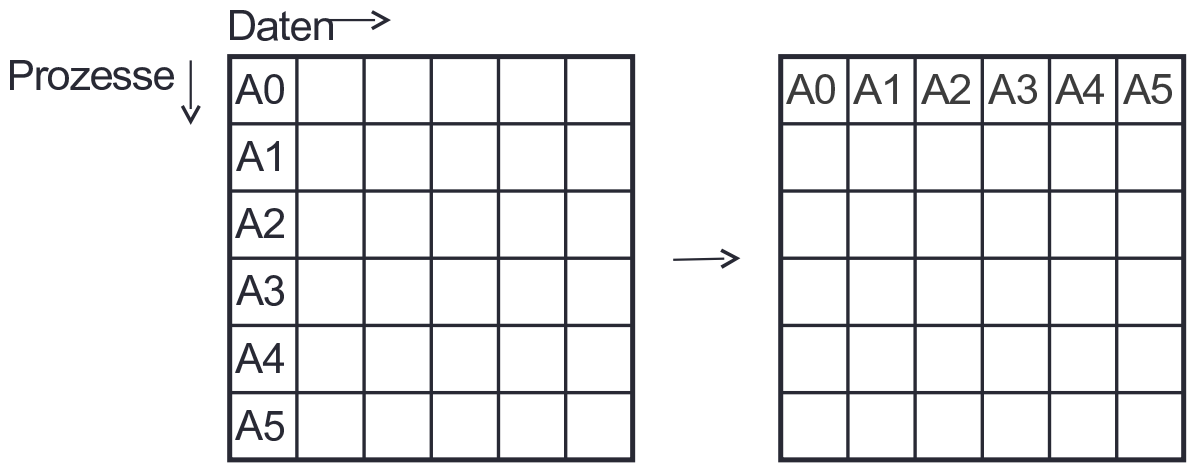


Quelle: R. Yahyapour, ehem. TU Dortmund

MPI: Kollektive Kommunikation IV

Gather : Empfang der Elemente eines größeren Datenbereichs von allen Prozessen in einer Prozessgruppe

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

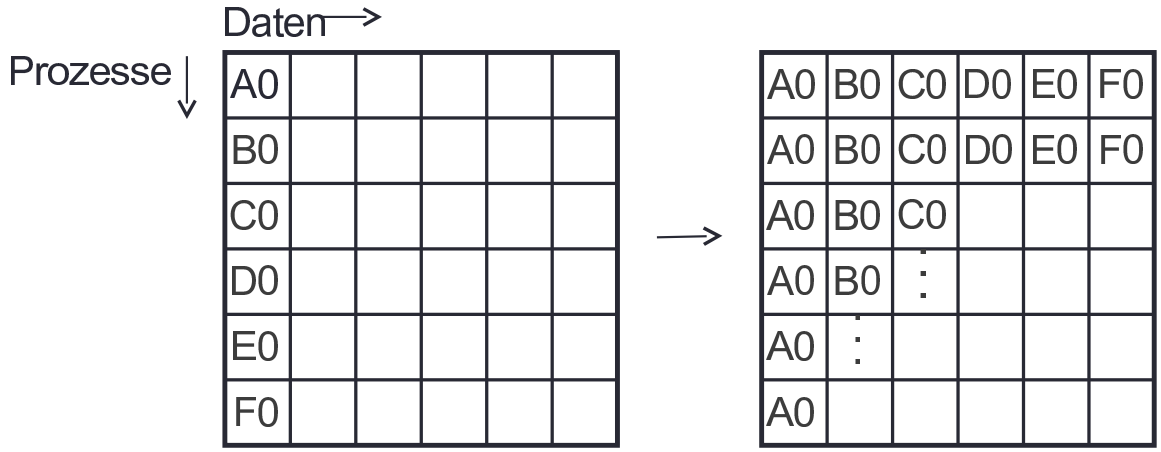


Quelle: R. Yahyapour, ehem. TU Dortmund

MPI: Kollektive Kommunikation V

Gather-to-all : Empfang der Elemente eines größeren Datenbereichs von allen Prozessen in einer Prozessgruppe *und durch jeden beteiligten Prozess*

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



Quelle: R. Yahyapour, ehem. TU Dortmund

MPI: Kollektive Kommunikation V

Gather-to-all : Empfang der Elemente eines größeren Datenbereichs von allen Prozessen in einer Prozessgruppe *und durch jeden beteiligten Prozess*

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

Beispiel:

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

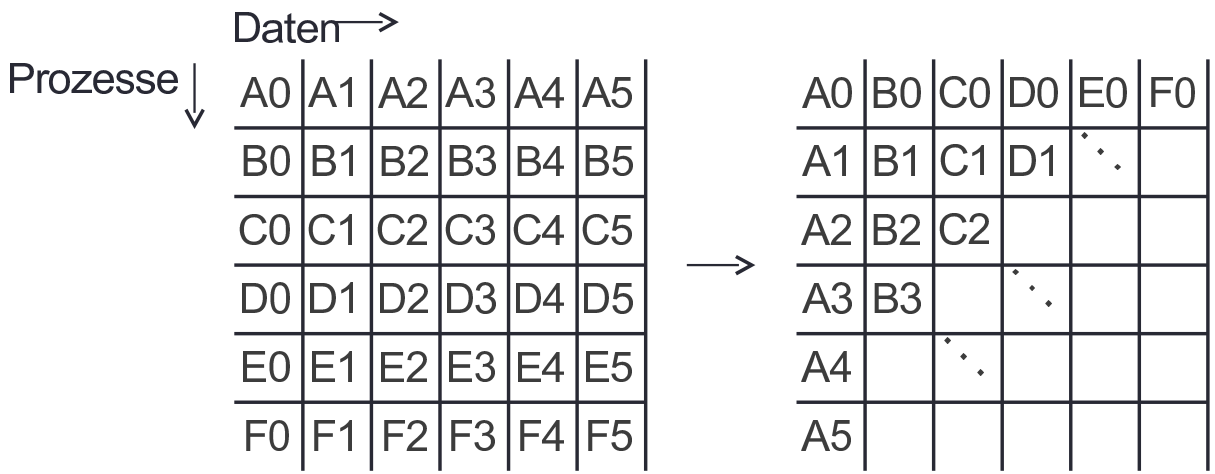
Quelle: MPI-Forum

MPI: Kollektive Kommunikation VI

All-to-all Scatter/Gather : Empfang je eines Datenelements von allen Prozessen und durch alle Prozesse in einer Prozessgruppe

[d.h. Kombination aus Scatter und Gather: Prozess j erhält von Prozess i das i-te Datenelement]

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



Quelle: R. Yahyapour, ehem. TU Dortmund

MPI: Kollektive Kommunikation VII

Reduktionsoperationen : Entspricht Gather-Kommunikation inklusive Anwendung einer Operation zur Kombination der Ergebnisse (z.B. Summe, Maximum, ...)

- Ex. best. vordefinierte Reduktionsoperationen (z.B. MIN, MAX, MINLOC [$\hat{=}$ Minimum + Position], SUM, PROD, AND, ...)
- Auch benutzerdefinierte Reduktionsoperationen möglich (werden von `root`-Prozess ausgeführt)
- Ex. Erweiterung zu `MPI_Allreduce()`, d.h. alle Prozesse erhalten Ergebnis
- Weitere Modifikation: Reduce-to-all mit zusätzlichem Zwischenergebnis: `MPI_Scan()`