

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Gliederung

- Organisatorisches
- Zur Person
- Einführung



# **Organisatorisches**

Wintersemester 2020/21

# Kontaktinformation



- **Name:** Volker Rodehorst
- **Telefon:** ~~03643 / 58-3773~~
- **Email:** [volker.rodehorst@uni-weimar.de](mailto:volker.rodehorst@uni-weimar.de)
- **Büro:** Bauhausstraße 11 (B11)  
~~Raum 110.1~~
- **Sprechstunden:** Auf Anfrage  
(besser vorher E-Mail schreiben)

# Vorlesungen und Übungen

- **Vorlesungszeit:** 2. November 2020 – 5. Februar 2021  
(2 Wochen Weihnachtsferien)
- **Voraussetzungen:** Einführung in die Informatik,  
Grundlagen Programmiersprachen
- **Vorlesungen:** **Volker Rodehorst**
  - Beginn: 6. November 2020
  - Freitag: ~~9<sup>15</sup> – 10<sup>45</sup> im HS der HK7 Moodle~~
  - Aufwand: 2 SWS
- **Übungen / Projekt:** **Mariya Kaisheva**
  - Beginn: 13. November 2020
  - Freitag: ~~13<sup>30</sup> - 15<sup>00</sup> im LiNT-Pool der B11 Moodle~~
  - Aufwand: Übungen 1 SWS (~14-tägig) / Projekt 1 SWS
- **Prüfung:**
  - Zeitraum: Februar 2021
  - Form: schriftlich (VL+UE 4.5 ECTS, +PR 1.5 ECTS)



# Computer Vision Homepage

- <http://www.uni-weimar.de/medien/cv>

The screenshot shows the homepage of the Professur Computer Vision in Engineering. At the top, there are links for WEBMAIL, VORLESUNGSVERZEICHNIS, PINNWÄNDE, and language selection (DE | EN). On the right, there are links for SUCHEN and SCHNELLZUGRIFF. The main header is "Bauhaus-Universität Weimar". Below it, a sub-header reads "Professur Computer Vision in Engineering" and "Prof. Dr.-Ing. Volker Rodehorst - Medieninformatik & Bauingenieurwesen". A navigation bar includes links for AKTUELLES, PERSONEN, LEHRE (which is highlighted in blue), GEODÄSIE, FORSCHUNG, and PUBLIKATIONEN. The "LEHRE" section contains a list of topics: Übersicht, Photogrammetric Computer Vision, Bildanalyse Und Objekterkennung, Parallelle Und Verteilte Systeme (which is circled in red), Raumbezogene Informationssysteme (GIS), Geodäsie, Bauvermessung, Geo-Spatial Monitoring, Hot Topics In Computer Vision, 3D-Rekonstruktion Aus Bildern, and Bauhaus Summer School. A note below states: "führt zu immer größer und komplexer werdenden Datenmengen. Eine wichtige Herausforderung ist es, diese Daten für unterschiedliche Anwendungsbereiche exakt und effizient auszuwerten." To the right, there is a sidebar for "Engineering" with contact information for Bauhaus-Universität Weimar, Faculty of Media & Civil Engineering, address Bauhausstraße 11, 99423 Weimar, and a list of contact details including web, email, phone, and fax numbers.



404



# Lernplattform Moodle

<https://moodle.uni-weimar.de/course/view.php?id=26730>

Lernplattform Bauhaus-Universität Weimar    Deutsch (de) ▾    Meine Kurse ▾    Farben ▾    Support ▾    Volker ▾

Hinweise zum Kurse anlegen via Bison: Das Anlegen von Kursen via Bison wurde für das WiSe 2020 am 18.09.2020 freigeschaltet. Wenden Sie sich bitte an Ihre Fakultät, falls Ihr Kurs nicht im Bison-Connector erscheint. Sie können die Kurse in Moodle einen Tag nach Eintrag in Bison anlegen.

Bitte verkleinern Sie Videodateien, bevor Sie diese auf Moodle hochladen. [Ein Lernvideo dazu: Videodateien um 90% verkleinern \(bei guter Qualität\)](#)

Dashboard > Meine Kurse > Parallele und verteilte S...-46865

## Parallele und verteilte Systeme WiSe2020

Nachrichten  
Vorlesungssäulen  
Für Teilnehmer/innen verborgen  
Zusätzliche Materialien  
Für Teilnehmer/innen verborgen  
virtueller Übungsraum

Wahl der Gruppen   Übung 1   Übung 2   Übung 3   Übung 4   Übung 5   Übung 6

Alle Übungen sollen in kleinen Gruppen von maximal 3 Studierenden bearbeitet werden. Sie können sich bis zur Abgabe der ersten Übung am 19. November 2020 in eine Gruppe eintragen.

Bitte in eine Gruppe eintragen!



KURSADMINISTRATION

- Bearbeiten einschalten
- Einstellungen bearbeiten
- Rolle wechseln ...

KURSTEILNEHMER

- Teilnehmerliste
- Trainerliste
- Eingeschriebene Nutzer/innen
- Einschreibemethoden
- Gruppen
- Bewertungen
- Teilnehmerliste exportieren

SUCHE

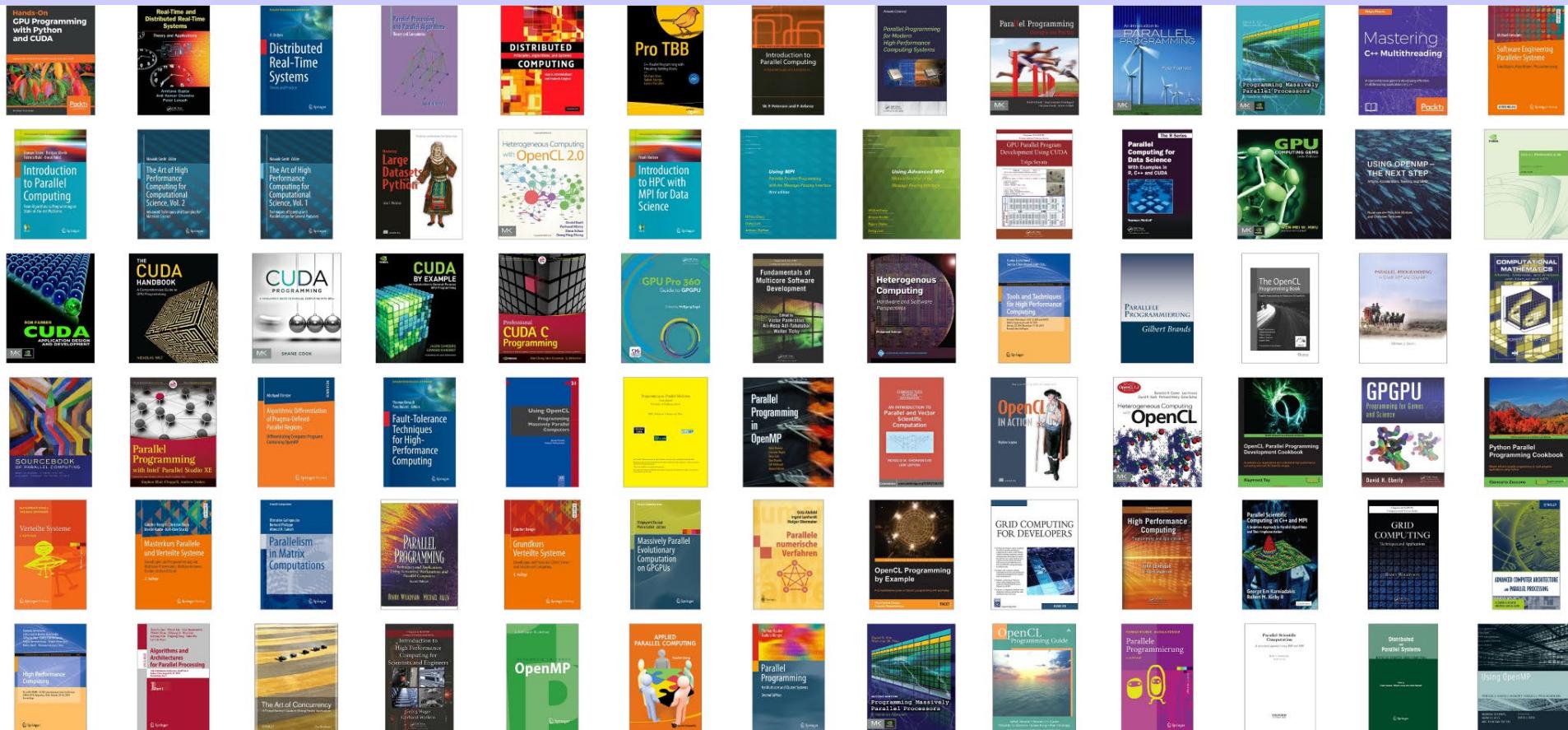
Suche nur in Parallele und verteilte

AKTUELLE TERMINE

Keine weiteren Termine

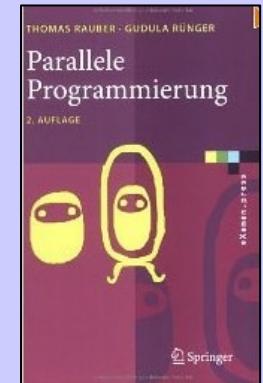
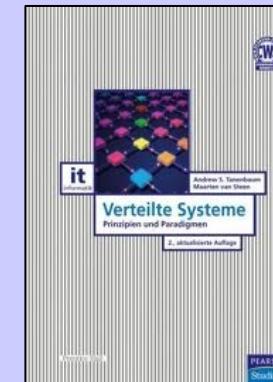
Zum Kalender ...

# Literatur (Auswahl)



# Literaturempfehlungen

- G. Bengel, C. Baun, M. Kunze und K.U. Stucky:  
**"Masterkurs Parallele und Verteilte Systeme: Grundlagen und Programmierung von Multicore-Prozessoren, Multiprozessoren, Cluster, Grid und Cloud"**, 2. Auflage, Springer Vieweg, 520 S., 2015
- S. Hoffmann, R. Lienhart:  
**"OpenMP - Eine Einführung in die parallele Programmierung mit C/C++"**  
Informatik im Fokus, Springer, 172 S., 2009
- A.S. Tanenbaum und M. van Steen:  
**"Verteilte Systeme: Prinzipien und Paradigmen"**  
2. Auflage, Pearson Studium - IT, 768 S., 2007
- T. Rauber und G. Rünger: **"Parallele Programmierung"**  
3. Auflage, Springer, 532 S., 2012





# Zur Person

Erfahrungen mit parallelem  
und verteiltem Rechnen

# 1. ParVis Stereobild-Verarbeitung

Linkes Bild



Rechtes Bild

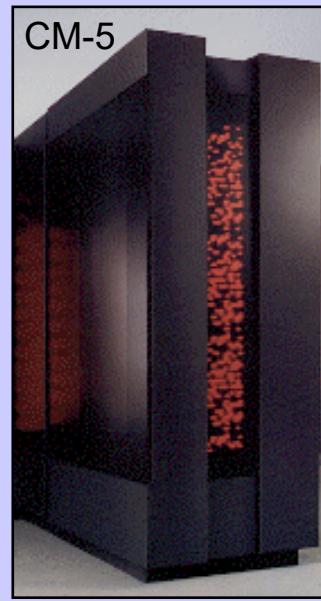


Dichte 3D Punktwolke

# TOP500 Supercomputer 1995



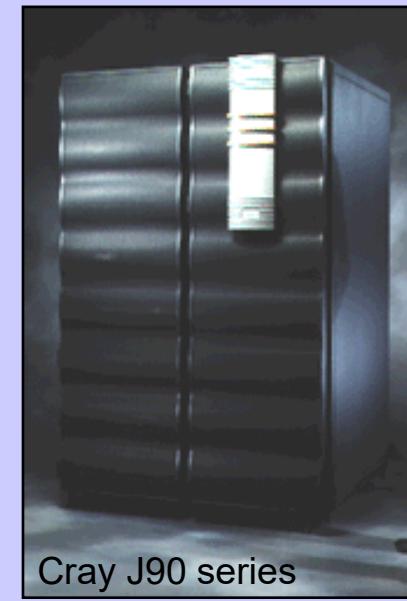
Cray T3D



CM-5



SGI

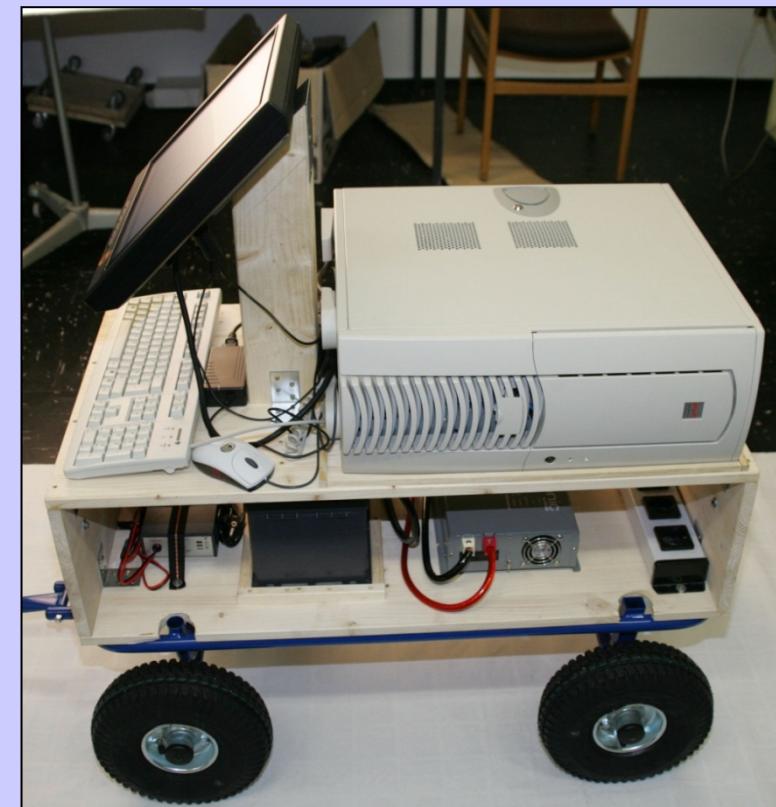


Cray J90 series

Nr.	Computer	Einsatzort	Jahr	Anwendung	PEs	$R_{MAX}$ [GFlops]
1	Fujitsu NWT	NAL, Japan	1993	Forschung	140	170400
39	Cray T3D	ZIB Berlin, D	1994	Lehre	192	19050
219	TMC CM-5	GMD Birlinghoven, D	1993	Forschung	64	3800
285	SGI Power Challenge	FU Berlin, D	1994	Lehre	12	3203
395	Cray J932	Uni Groningen, NL	1995	Lehre	16	2170

## 2. Trifokales Videosensor System

- Drei hochauflösende digitale Videokameras
- Auflösung 5MP unkomprimiert bei 16 Hz
- Effektive Datenrate  
 $2448 \times 2048 \approx 5 \text{ MB}$  pro Rohbild  
16 Bilder/Sek.  $\approx 80 \text{ MB}$   
3 Kameras  $\approx 240 \text{ MB/Sek.}$



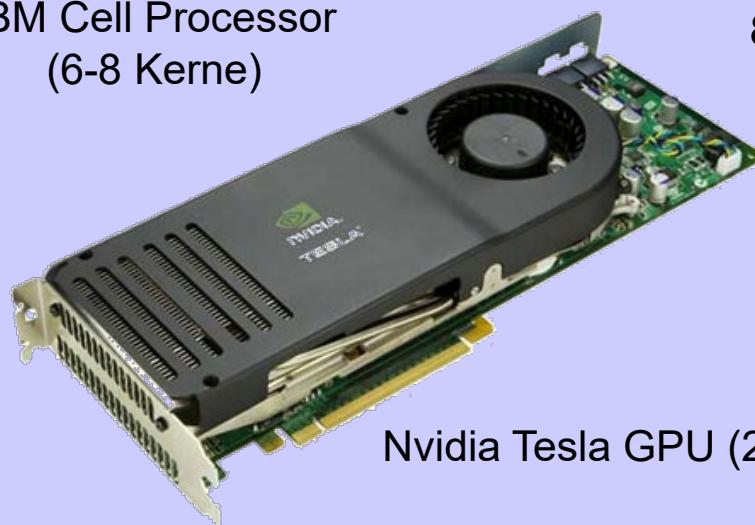
# Mehr Tempo durch Hardware 2004-2010



IBM Cell Processor  
(6-8 Kerne)



8 AMD Opteron Quad-Core CPUs  
(32 Kerne, 128 GB RAM)



Nvidia Tesla GPU (2496 Stream Prozessoren)

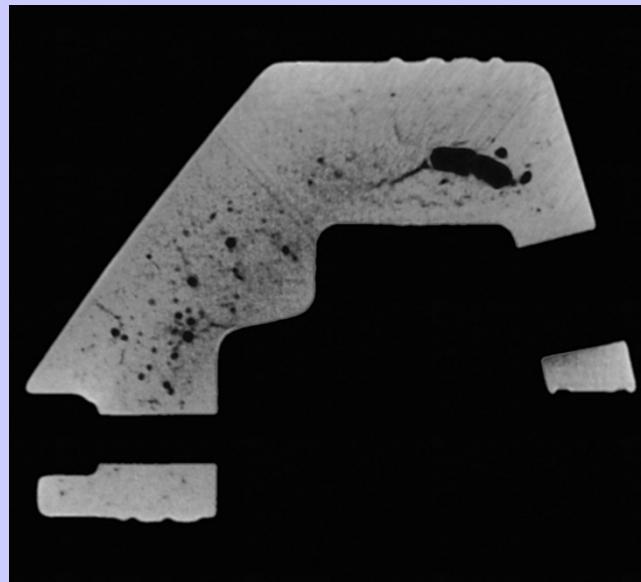
### 3. Industrie-Computertomograph 2012



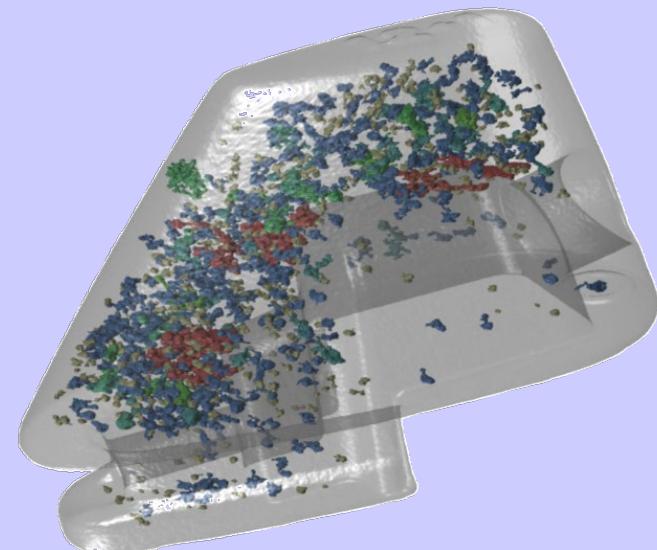
# Beispiel: Materialfehleranalyse



Aluminium Gussbauteil



2D-Ansicht der Rekonstruktion



Porositätsanalyse

# Einführung

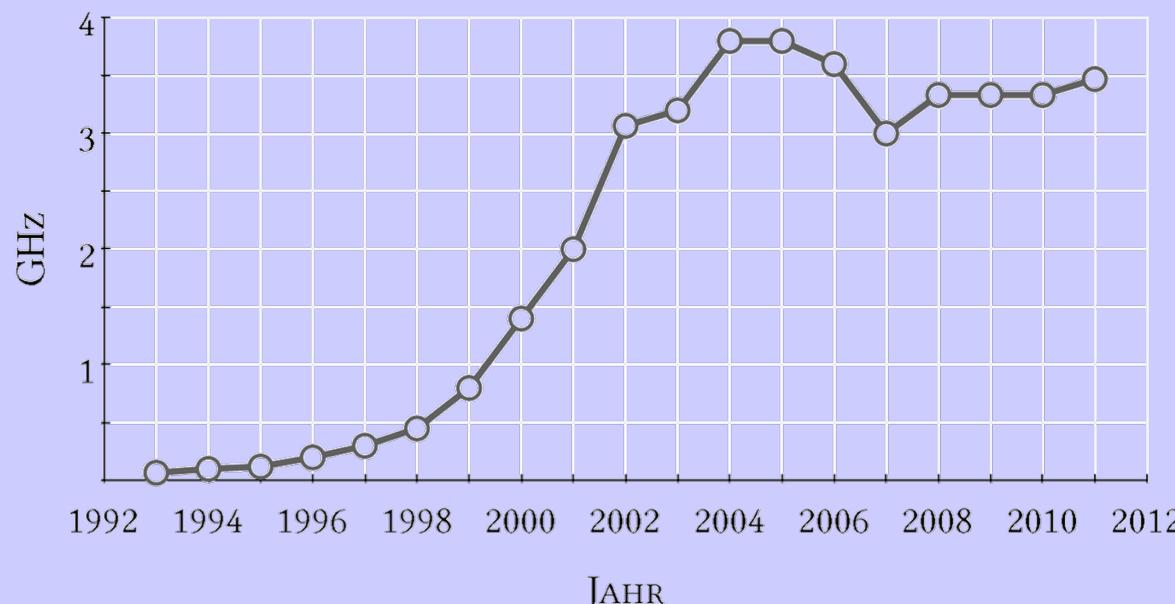
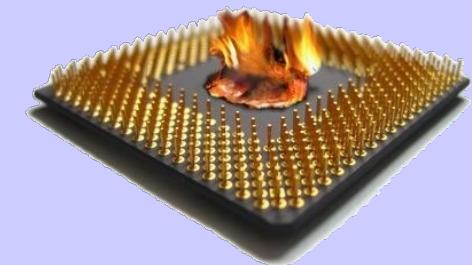
Parallele und verteilte Systeme



# Anwendung langsam! Was tun?

- **Härter arbeiten**

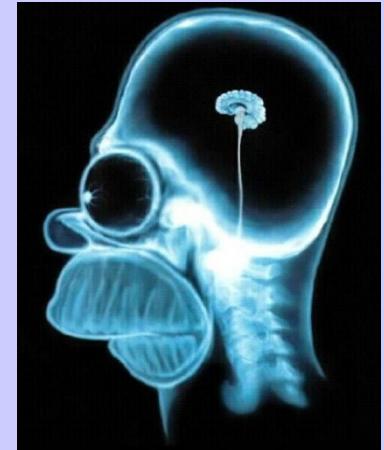
- Einen größeren und schneller getakteten Prozessor einsetzen, Arbeitsspeicher aufrüsten usw. ...
- Die Geschwindigkeitssteigerung bei Prozessoren stößt langsam an ihre Grenzen!
- Entwicklung der Taktraten von Prozessoren:



# Anwendung langsam! Was tun?

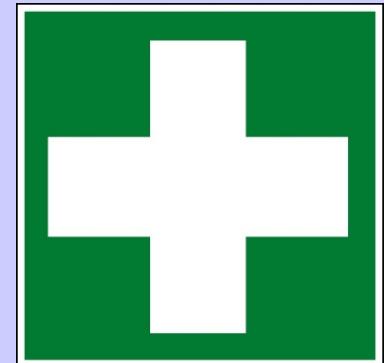
- **Cleverer arbeiten**

- Den Algorithmus optimieren (z.B. mit Profiler), Zugriffe auf kritische Ressourcen minimieren usw. ...
- Keine Grenzen, da kaum ein Algorithmus optimal ist. Allerdings ist der Aufwand für den Programmierer enorm!



- **Hilfe holen**

- Mehrere Prozessoren an dem gleichen Problem arbeiten lassen!



# Klassen von Parallelrechnern

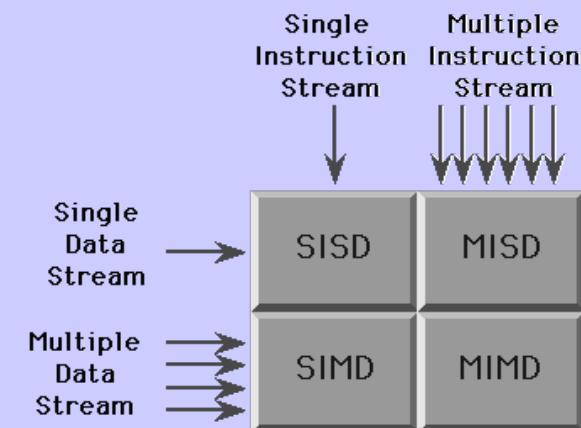
- Theoretische Aspekte
  - Flynn'sche Klassifikation
  - Mooresches & Amdahlsches Gesetz
- Mehrkern Berechnung
  - Mehrkern-Prozessor Varianten
  - Simultaneous Multi-Threading / Hyper-Threading
- Verteiltes Rechnen
  - Cluster Computing, Super-Computing
  - Grid Computing
- Spezialisierte Parallelrechner
  - Allzweck-Berechnung auf Grafikprozessoren (GP-GPU)

# Flynnsche Klassifikation / Taxonomie

## Unterteilung von Rechnerarchitekturen

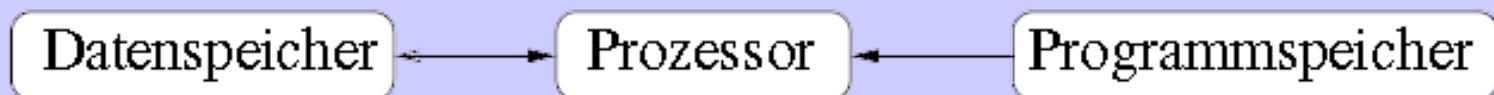
(MICHAEL J. FLYNN, 1966):

- **SISD** (Single Instruction, Single Data)  
z.B. Traditionelle Einprozessor-Rechner  
(Von-Neumann, ältere Personal Computer)
- **MISD** (Multiple Instruction, Single Data)  
Sollte es eigentlich nicht geben?  
Evtl. fehlertolerante Systeme, die redundante Berechnungen ausführen, Makropipelining ...
- **SIMD** (Single Instruction, Multiple Data)  
z.B. Supercomputer, Vektorrechner, Grafikkarten, ...
- **MIMD** (Multiple Instruction, Multiple Data)  
z.B. Supercomputer, verteilte Systeme, Mehrkern-Systeme, ...

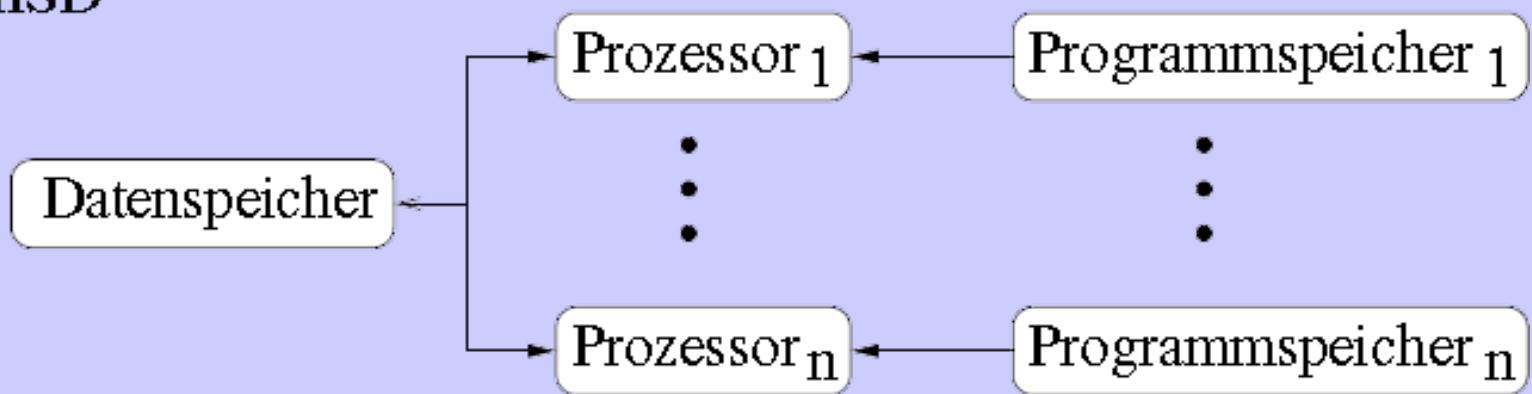


# Modell zur Flynnnschen Klassifikation 1

a) SISD

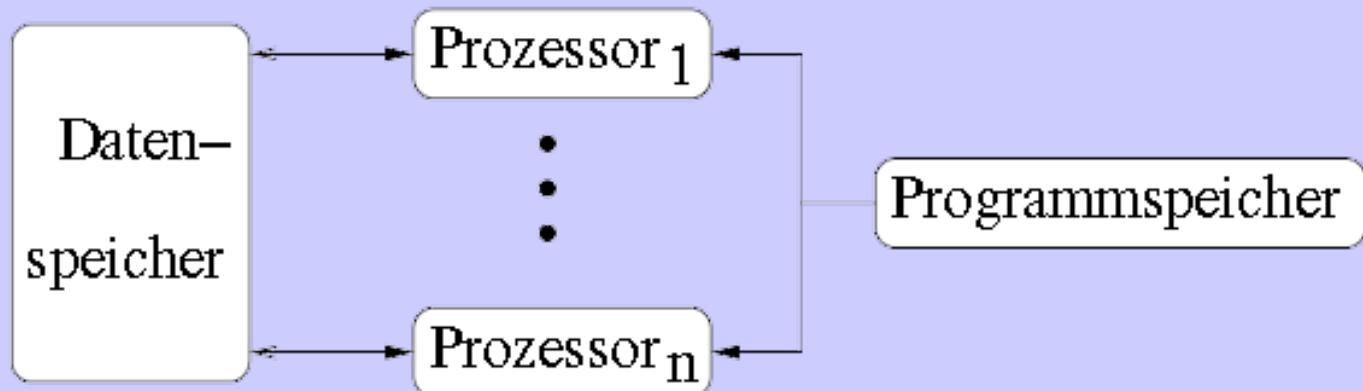


b) MISD

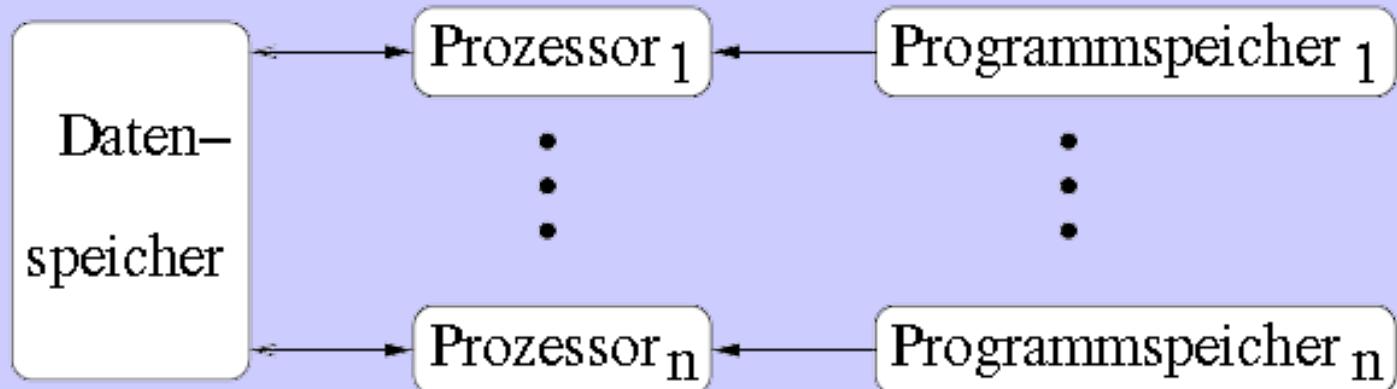


# Modell zur Flynnnschen Klassifikation 2

c) SIMD



d) MIMD

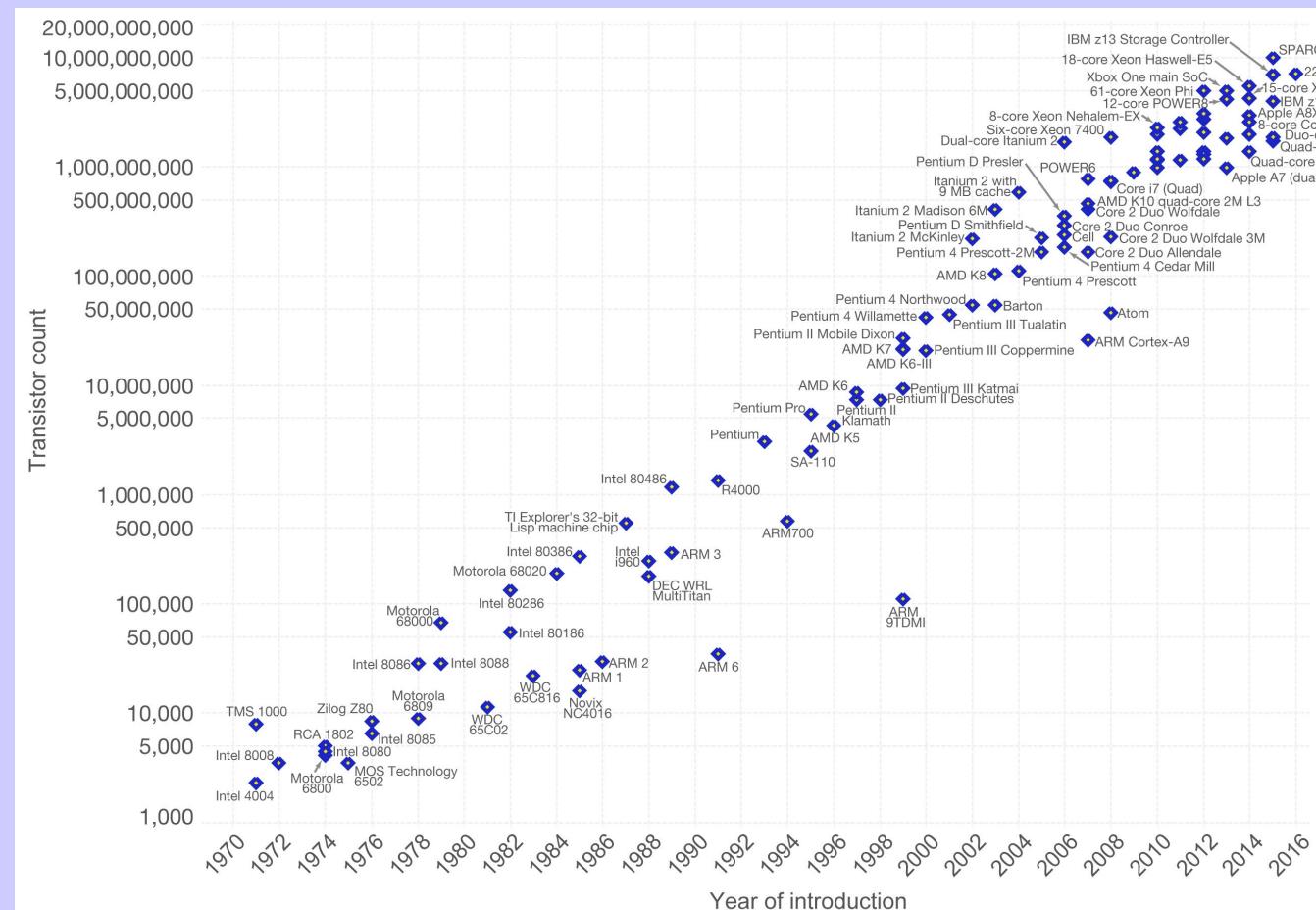
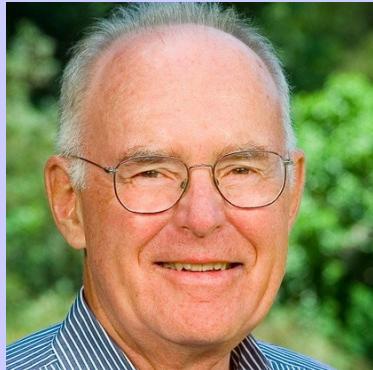


# Prozessorentwicklung

- Die Steigerung der Transistoren wurde früher primär zur Erhöhung der **Taktrate** genutzt
- Inzwischen werden diese für mehrere **Prozessorkerne** (Cores) in einem Prozessor verwendet (**MultiCore**)
- Aktuell sind Prozessoren mit bis zu acht Kernen üblich und im Server-Bereich auch bis zu 28 Kernen
- Im Tera-Scale Forschungsprojekt von Intel werden Prozessoren mit bis zu 100 Kernen (**ManyCore**) für das nächste Jahrzehnt erforscht

# Mooresche(s) Gesetz(mäßigkeit)

- Alle 18 Monate verdoppelt sich die Leistung (Anzahl der Transistoren) in einem Prozessor (**GORDON MOORE**, 1975):





# 1. Mehrkern-Berechnung

# 1.1 Mehrkern-Berechnung

- **Beispiel:** Dual-/Quad-/Hexa-/Octa-Core Prozessoren
- Mikroprozessor mit mehreren vollständig voneinander unabhängigen Prozessoren (**Cores, Kernen**) mit eigenen arithmetisch-logische Einheiten (ALU), Registersätzen und Fließkomma-Einheiten (FPU)
- Gemeinsamer Arbeitsspeicher (**Shared-Memory**)
- Eine Anwendung muss so modifiziert werden, dass sie komplett (oder Teile davon) gleichzeitig auf mehreren Kernen als **Threads** („Fäden“) ausgeführt werden
- **Software**
  - Open Multi-Processing (**OpenMP**)
  - POSIX Threads (Pthreads)
  - Boost C++ Bibliothek, C++11 ISO Standard



# 1.2 Mehrkern-Prozessor Varianten

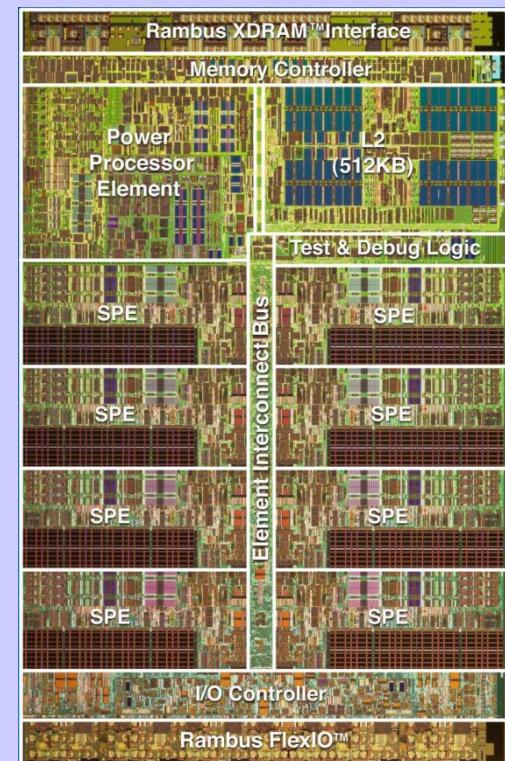
- **Symmetrische**

Alle Kerne sind gleich (Symmetrisches Multiprozessorsystem, **SMP**). Ein für diesen Prozessor übersetztes Programm kann auf jedem Kern ausgeführt werden

- **Asymmetrische**

Es gibt verschiedene Kerne, die unterschiedlich angesteuert. Ein Programm kann nur auf einem seiner Übersetzung entsprechenden Kern ausgeführt werden. Bei dieser Art von Mehrkernprozessoren arbeiten einige Kerne eher wie asynchrone **Coprozessoren**

**Beispiel:** IBM Cell-Prozessor



# 1.3 Simultaneous Multi-Threading

- Deutsch: "Gleichzeitiger Mehrfadenbetrieb" !?
- **Beispiele:**
  - Intels **Hyper-Threading (HT)** Technologie für Pentium 4, Xeon und Core i7 Prozessoren
  - IBMs **SMT** für Cell und Power5/6 Prozessoren
- Mikroprozessor kann mittels getrennter Pipelines und zusätzlicher Register **mehrere Threads gleichzeitig** ausführen. Sie teilen sich aber die gleichen Datenverarbeitungseinheiten (ALU/FPU)
- Kostengünstige aber leistungsärmere Alternative zu echten Mehrkern-Systemen



## 2. Verteiltes Rechnen

## 2.1 Cluster Computing

- **Beispiele:**
  - High-Performance-Computing (HPC) Cluster
  - Renderfarm
- Ein verteiltes System besteht aus mehreren **eigenständigen Rechnern**, die über ein Computernetzwerk (z.B. TCP/IP Ethernet) kommunizieren, um ein gemeinsames Ziel zu erreichen
- Jeder Rechner (oder Knoten) besitzt seinen eigenen lokalen Prozessor und Arbeitsspeicher
- Die Knoten tauschen Nachrichten aus (**Message Passing**), die z.B. Funktionsausrufe, Signale und Datenpakete enthalten können
- **Software:** z.B. Open Message Passing Interface (**Open MPI**)

# Beispiel: Computer Cluster



## 2.2 Massiv-paralleles Rechnen

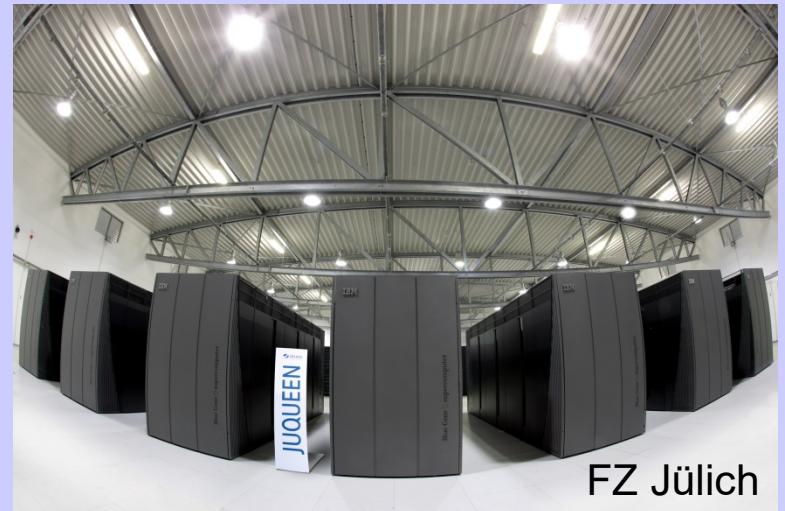
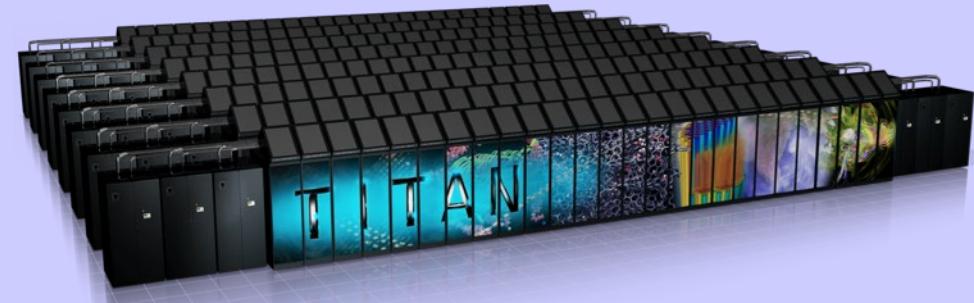
- **Beispiel:** Tianhe-2, IBM Blue Gene
- Verteilung der Aufgabe auf sehr viele ( $>> 100$ ) Hauptprozessoren in einem Rechner
- Jede Recheneinheit verfügt häufig über eigenen Arbeitsspeicher, eine Kopie des Betriebssystems und der Anwendung
- Jedes Subsystem kommuniziert über ein spezialisiertes Hochgeschwindigkeits-Netzwerk (z.B. 100GbE, InfiniBand, QsNet, ...)

# Top500 Supercomputer Juni 2020

Rank	System	Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6	HPC5 - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
7	Selene - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	272,800	27,580.0	34,568.6	1,344
8	Frontera - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
9	Marconi-100 - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476
10	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre [CSCS] Switzerland	387,872	21,230.0	27,154.3	2,384



# Beispiele: Supercomputer



## 2.3 Grid Computing

- **Beispiele:**
  - SETI@home (Suche nach außerirdischer Intelligenz),
  - Folding@home (Simulation der Faltung von Proteinen),
  - Bitcoin Network (Dezentrales Bezahlssystem der digitalen Währung)
  - World Community Grid (WCG)
- Verteilung der Aufgabe an unterschiedlichste Rechner, die über das Internet kommunizieren
- Aufgrund der geringen Bandbreite und hohen Verzögerung des Internets sind nur Anwendungen geeignet, die keine Abhängigkeiten besitzen und kaum kommunizieren müssen
- **Software:**
  - Berkeley Open Infrastructure for Network Computing (BOINC)



## 3. Spezialisierte Parallelrechner

# 3. Berechnung auf Grafikprozessoren

- **Beispiele:**

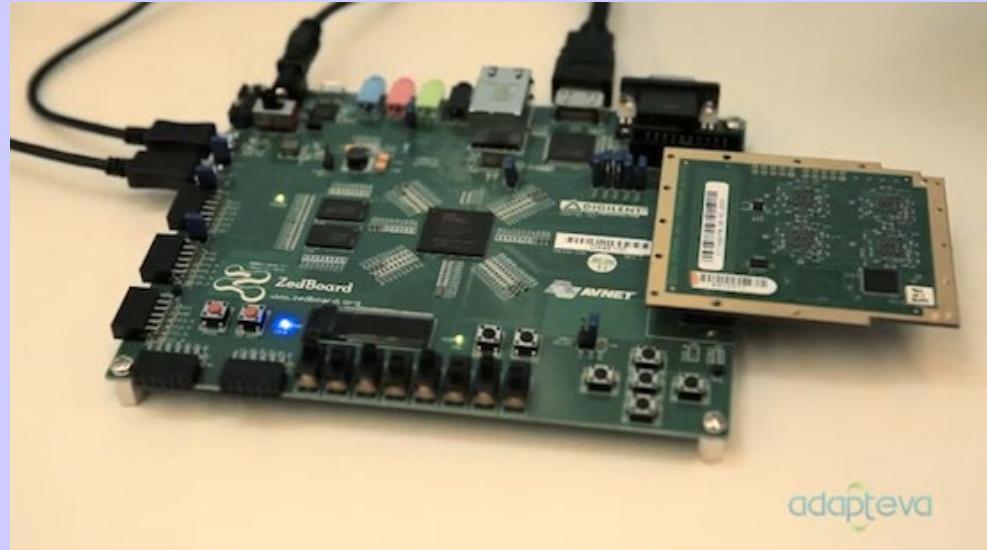
- Aktuelle Spiele-Grafikkarte
  - Nvidia Tesla, AMD FireStream, Intel Xeon Phi  
(bessere Fehlerkorrektur, schnellere FPU)



- Allgemeine wissenschaftlich-technischen Berechnungen auf einer **Graphics Processing Unit (GP-GPU)**
- Beim gleichzeitigen Ausführen von gleichförmigen Aufgaben ist die GPU durch massive Parallelisierung und schnellen Speicher sehr performant
- Moderne GPUs haben über 2.000 programmierbare **Prozessor-Einheiten (PE)**, die in **Shadern** zusammenfasst werden
- Eine GPU enthält Fragment-Shader (SIMD) und Vertex-Shader (MIMD), die gleichzeitig ausgeführt werden können
- **Software:**
  - Open Computing Language (**OpenCL**)
  - Nvidias Compute Unified Device Architecture (CUDA)

# Adapteva Parallel-Computer für Bastler

- Dual-core ARM A9 CPU
- Epiphany Multicore  
800 MHz  
**(64 cores, 102 GFlops)**
- 1 GB RAM
- MicroSD Card
- USB 2.0
- Ethernet 10/100/1000
- HDMI Anschluss
- Software: Ubuntu OS, C compiler, multicore debugger, Eclipse IDE, **OpenCL** SDK/compiler und run-time libraries.
- Größe: 8.6 x 5.3 cm
- Verbrauch: **5 Watt**
- Preis: ca. **76 EUR**



Übungsplattform für Parallelprogrammierer



# Parallelprogrammierung

# Amdahlsches Gesetz

- Abschätzung der maximalen Beschleunigung, **Speedup S** (GENE AMDAHL, 1967)
- Programme laufen nie vollständig parallel (z.B. wegen Initialisierung, Speicherallokation, auf Teilergebnisse warten, ...)
- **Anteil der Laufzeit:**

$$1 = (1 - P) + P$$

sequentieller      paralleler  
Anteil                Anteil

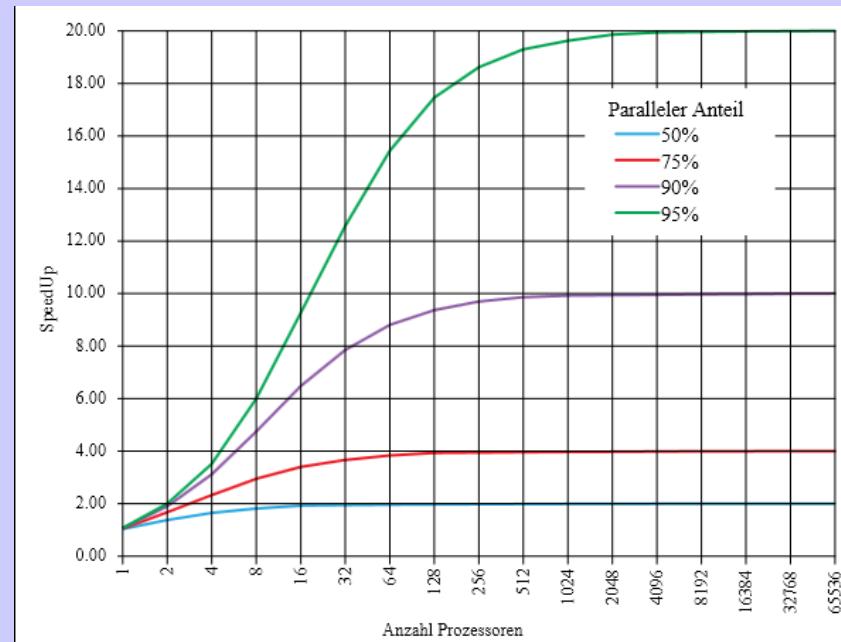


- **Beispiel:**

Anwendung läuft 20 h (1 h = 5 % sequentiell).  
 19 h = 95 % des Aufwandes werden verteilt.  
 Die Rechenzeit kann nicht < 1 h fallen,  
 d.h. der max. Speedup S ist  $1 / (1 - P) = 20$

- Bei  $N$  Prozessoren und **Kosten**  
 $o(N)$  für Kommunikation und  
 Synchronisation gilt:

$$S = \frac{1}{(1 - P) + o(N) + \frac{P}{N}} \leq \frac{1}{1 - P}$$



# Parallelprogrammierung

- Man benötigt **Mechanismen** zum
  - **Erzeugen** von Parallelität
  - **Unterscheiden** von Prozessen
  - **Kommunizieren** zwischen den Prozessen
- **Formulierung** paralleler Programme
  - **Kontrollparallel**  
(simultane Ausführung verschiedener Instruktionsströme,  
mehrere Kontrollflüsse, schwer skalierbar, passt gut zu MIMD)
  - **Datenparallel**  
(alle Datenelemente werden gleich behandelt, ein Kontrollfluss,  
gut skalierbar, passt gut zu SIMD)
  - Mischform (Kontroll- und Datenparallel)

# Begriffe

- **Prozess**

- Besteht aus eigenem Adressraum
  - Für automatischen Speicher (Stack), dynamischen Speicher (Heap), Programmcode, ...
  - Ist dadurch vor anderen Prozessen geschützt
- Kann Ressourcen reservieren
- Hat einen oder mehrere Threads, die den Code ausführen
- Beispiel:
  - Programm

- **Thread**

- Gehört zu einem Prozess
- Besteht aus eigenem Stack und CPU-Registerzustand
- Hat den Adressraum des zugehörigen Prozesses
  - Threads desselben Prozesses sind nicht voreinander geschützt
- Beispiel:
  - Auf mehrere Threads verteilte for-Schleife eines Programms

# Herausforderungen

- **Wettrennen (Race conditions)**
  - Situationen, in denen der Zugriff auf gemeinsame Ressourcen Auswirkungen auf das Ergebnis eines Programm hat
  - Beispiel:
    - Zwei Threads schreiben gleichzeitig in dieselbe Speicherstelle
- **Lösung durch Synchronisation**
  - **Mutex** (Mutual exclusion, Wechselseitiger Ausschluss)
    - Ein Mutex kann von mehreren Threads verlangt werden (lock), aber nur einer besitzt sie bis er sie wieder freigibt (unlock)
  - **Semaphore**
    - Kann von mehreren Threads verlangt werden, ist aber immer nur im Besitz von höchstens  $k$  Threads
    - Mutex ist eine binäre Semaphore mit  $k = 1$
  - **Barrier**
    - Eine Gruppe von Threads hält solange an einer Barriere an, bis alle angekommen sind

# Probleme

- Bei fehlender (bzw. ungeeigneter) Synchronisation entstehen:
  - **Verklemmung (dead locks)**
    - Threads warten gegenseitig auf die Ressourcen der anderen
  - **Aushungern (starvation)**
    - Ressource wechselt (unfairerweise) nur innerhalb einer Gruppe von Threads
    - Threads außerhalb der Gruppe erhalten die Ressource nie

# Beispiel: Speisende Philosophen

- **Philosophenproblem** (Dijkstra, 1965)
  - 5 Philosophen denken und essen abwechselnd
  - Zwischen den Tellern Spaghetti liegt eine Gabel
  - Zum Essen braucht jeder zwei Gabeln
  - Maximal 2 können gleichzeitig essen

- Beispiel: **Verklemmung**
  - Jeder nimmt die linke Gabel auf und wartet auf die rechte
  - Alle Philosophen verhungern

- Beispiel: **Aushungern**
  - Zwei Philosophen übergeben immer denselben anderen zwei Philosophen ihre Gabeln (unfaires Essverhalten)
  - Der fünfte Philosoph müsste verhungern



# Lösungsmöglichkeiten

- **Dirigenten**
  - Ein zusätzlicher Aufseher weiß, welche Gabeln gerade verwendet werden und kann Verklemmungen vermeiden
  - Wenn bereits vier Gabeln verwendet werden, muss der nächste Philosoph auf die Erlaubnis des Aufsehers warten
- **Ressourcen-Hierarchie**
  - Jeder Philosoph muss die Gabel mit der niedrigeren Nummer zuerst aufheben. Nach dem Essen muss er die Gabel mit der höheren Nummer zuerst auf den Tisch legen
  - Nur ein Philosoph hat Zugriff auf die Gabel mit der höchsten Nummer
- **Monitore**
  - Philosophen können nur dann essen, wenn kein Nachbar gerade isst
  - Monitore ermitteln den Zustand der Philosophen (Mutex)
  - Ein separater Zähler behält die Übersicht über die hungernden Philosophen
- **Chandy / Misra**
  - Keine zentrale Verwaltungsstelle.
  - Jede Gabel kann ‚sauber‘ oder ‚schmutzig‘ sein.
  - Kein Verhungern, da Philosophen, die gerade gegessen haben, benachteiligt werden

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- Mehrkern-Programmierung mit OpenMP

## Hinweis:

- **Seriell vs. parallel**  
(Statische Einheit, Komponenten, Infrastruktur, deterministisch, ...)
- **Sequentiell vs. nebenläufig**  
(Dynamisch, zeitabhängig, nicht deterministisch, ...)



# Einführung in OpenMP

Mehrkern-Programmierung

# Open Specifications for Multi-Processing

- OpenMP ist eine standardisierte Programmierschnittstelle (API) für **C/C++** und Fortran
- Schwerpunktmaßig zur **portablen** Programmierung von Parallelrechnern mit **gemeinsamen Speicher** (shared memory, non-uniform memory access NUMA)
- **Erweiterung** der Programmiersprache durch
  - Bibliotheksfunktionen
  - Compiler-Anweisungen
  - Umgebungsvariablen
- **Nützliche Links**
  - OpenMP Homepage: [openmp.org](http://openmp.org)
  - OpenMP Tutorial: [computing.llnl.gov/tutorials/openMP/](http://computing.llnl.gov/tutorials/openMP/)
  - Kurzübersicht (V5.0): [openmp.org/wp-content/uploads/OpenMPRef-5.0-0519-web.pdf](http://openmp.org/wp-content/uploads/OpenMPRef-5.0-0519-web.pdf)



# Vorteile von OpenMP

- OpenMP bietet einen hohen **Abstraktionsgrad**
- Der Quelltext muss nicht wesentlich verändert, sondern nur **ergänzt** werden
- Der Code bleibt stets auch **sequentiell lauffähig**
  - indem die Ergänzungen direkt vom Compiler ignoriert werden bzw.
  - die Thread-Anzahl auf 1 gesetzt wird
- Parallelisierungen sind **lokal** begrenzt, d.h. für eine erfolgreiche Parallelisierung genügt oft schon eine Erweiterung von geringem Umfang
- Mit OpenMP sind Leistungsoptimierungen „in letzter Minute“ möglich, da **kein Neuentwurf** der Applikation zur Parallelisierung notwendig ist

# Verfügbarkeit

- OpenMP ist ein **offener Standard** mit breiter Unterstützung durch Firmen (z.B. AMD, ARM, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, Nvidia, Red Hat, Sun/Oracle, TI, ...)
- Version 1.0 (1998) - Version **5.0** (November 2018)

Compiler	OpenMP Unterstützung	Compiler Schalter
g++	GCC 4.7: OpenMP 3.1 GCC 4.9: OpenMP 4.0 <b>GCC 6.1: OpenMP 4.5</b> GCC 9.0: OpenMP 5.0 (subset)	-fopenmp
Visual C++	VS 2005- <b>2019: OpenMP 2.0</b> (Professional, Enterprise, Ultimate, nicht in der Express Edition!)	/openmp (in der IDE)
Intel C++	V12: OpenMP 3.1 V15: OpenMP 4.0 <b>V17-19: OpenMP 4.5</b> V20: OpenMP 5.0 (subset)	Windows: -Qopenmp Linux: -openmp

- **Übersetzen von C / C++ -Programmen unter Linux**, z.B. mit

```
gcc -fopenmp -O3 -Wall -o Beispiel Beispiel.c  
g++ -fopenmp -O3 -Wall -o Beispiel Beispiel.cpp
```

# OpenMP Spezifikationen

- 1998: erstes API für C/C++ (V1.0)
- **2002: C/C++ (V2.0)**
- 2005: C/C++ und Fortran (V2.5)
  - Ein Standard für beide Sprachen
  - Klarstellungen besonders im Speichermodell
- 2008: V3.0
  - Erweiterungen wie Task-Parallelität, geschachtelte Schleifen, ...
- **2011: V3.1**
  - Erweiterungen wie Optimierung des Task-Modells,
  - Mechanismus um Threads an Prozesse zu binden
  - Erweiterung des atomic-Pragmas
  - Min/max-Reduktion für C/C++
  - ...
- 2013: V4.0
- 2015: V4.5
  - GPU-Support mit CUDA
- 2018: V5.0
  - Aktuelle Version, Unterstützung für eingebettete Systeme/Beschleuniger, bessere Portabilität

# OpenMP in C/C++

- **Bibliotheksfunktionen**

- `#include <omp.h>`

- **Compiler-Anweisungen**

- OpenMP-**Pragmas** haben die allgemeine Form:

```
#pragma omp Direktive [Klausel[,] Klausel] ... new-line
strukturierter Block
```

- **Klauseln** sind **optional** und beeinflussen das Verhalten der Direktive
  - Alle Anweisungen müssen mit dem **Zeilenumbruch** enden!
  - Direktiven über **mehrere Zeilen** werden folgendermaßen geschrieben:

```
#pragma omp Direktive hier_steh_t_der_Beginn \
und_hier_steh_t_der_Rest
```

# OpenMP in C/C++

## • Compiler-Anweisungen

- Direktiven und API-Funktionen werden **kleingeschrieben**
- Beziehen sich auf die **unmittelbar folgende** Anweisung bzw. auf den nachfolgenden Block

```
#pragma omp parallel {  
    // kompiliert nicht!  
}
```

```
#pragma omp parallel  
    printf("Ich bin parallel.\n");  
    printf("Ich leider nicht.\n");
```

```
#pragma omp parallel  
{  
    printf("Ich bin parallel.\n");  
    printf("Ich auch.\n");  
}
```

- Werden automatisch **ignoriert** (evtl. mit Warnungen), wenn der Compiler OpenMP nicht unterstützt
- Explizites **Ein-** bzw. **Ausschließen** von Codeabschnitten mit:

```
#ifdef __OPENMP  
    // Anweisungen  
#endif
```

# OpenMP 3.1 Quick Reference Card

**OpenMP API 3.1 C/C++**

# OpenMP 3.1 API C/C++ Syntax Quick Reference Card

OpenMP Application Program Interface (API) is a portable, scalable model that gives memory-parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

[n.n.n] refers to sections in the OpenMP API Specification available at [www.openmp.org](http://www.openmp.org).

## Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A structured-block is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

### Parallel [2.4.1]

The parallel construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[, clause]...]
    structured-block
clause:
  if[scalar-expression]
  num_threads[integer-expression]
  default[shared | none]
  private[list]
  firstprivate[list]
  shared[list]
  copyin[list]
  reduction(operator: list)
```

### Loop [2.5.1]

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for [clause[, clause]...]
  for-loop
clause:
  if[scalar-expression]
  firstprivate[list]
  lastprivate[list]
  reduction(operator: list)
  schedule[kind[, chunk_size]]
  collapse[n]
  ordered
  nowait
```

Most common form of the for loop:

```
for(war == 1b; war < relational-or-b; war += incr)
```

### Simple Parallel Loop Example

The following example demonstrates how to parallelize a simple loop using the parallel loop construct.

```
void simple(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i=1; i < n; i++) /* i: 1 to n-1 */
        b[i] = a[i] + a[i-1] / 2.0;
}
```

## Parallel Sections [2.6.2]

The parallel sections construct is a shortcut for specifying a parallel construct containing one sections construct and no other statements.

```
#pragma omp parallel sections [clause[, clause]...]
  [#pragma omp section]
  structured-block
  [#pragma omp section]
  structured-block
  ...
clause:
Any of the clauses accepted by the parallel or sections directives, except the nowait clause, with identical meanings and restrictions.
```

### Task [2.7.1]

The task construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

```
#pragma omp task [clause[, clause]...]
  structured-block
clause:
  if[scalar-expression]
  final[scalar-expression]
  untied
  default[shared | none]
  mergeable
  private[list]
  firstprivate[list]
  shared[list]
```

### Taskyield [2.7.2]

The taskyield construct specifies that the current task can be suspended in favor of execution of a different task.

```
#pragma omp taskyield
```

### Master [2.8.1]

The master construct specifies a structured block that is executed by the master thread of the team. There is no implicit entry either on entry to, or exit from, the master construct.

```
#pragma omp master
  structured-block
```

### Critical [2.8.2]

The critical construct restricts execution of the associated structured block to a single thread at a time.

```
#pragma omp critical [name]
  structured-block
```

### Barrier [2.8.3]

The barrier construct specifies an explicit barrier at the point at which the construct appears.

```
#pragma omp barrier
```

### Taskwait [2.8.4]

The taskwait construct specifies a wait on the completion of child tasks of the current task.

```
#pragma omp taskwait
```

### Atomic [2.8.5]

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

```
#pragma omp atomic [read | write | update | capture]
expression-stmt
```

### Pragma OpenMP Atomic Capture

```
#pragma omp atomic capture
structured-block
where expression-stmt may be one of the following forms
```

If clause is...	expression-stmt:
read	$x = x_r$
write	$x = x_w$
update or	$x = x_u$
is not present	$x = x_{np}$
capture	$y = x_{cap}$

and structured-block may be one of the following forms:

$y = x * b1$	$b1 = b1 * y$
$y = x + b1$	$b1 = b1 + y$
$y = x - b1$	$b1 = b1 - y$
$y = x / b1$	$b1 = b1 / y$

**Flush [2.8.6]**

The flush construct executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

```
#pragma omp flush [list]
```

### Ordered [2.8.7]

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```
#pragma omp ordered
  structured-block
```

### Threadprivate [2.9.2]

The threadprivate directive specifies that variables are replicated, with each thread having its own copy.

```
#pragma omp threadprivate [list]
list:
  A comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.
```

**OpenMP API 3.1 C/C++**

## Routine Library Routines

### Execution Environment Routines [3.2]

Execution environment routines affect and monitor threads, processors, and the parallel environment.

**void omp\_set\_num\_threads(  
    int num\_threads);**

Subsets the number of threads used for subsequent parallel regions that do not specify a `num_threads` clause.

**int omp\_get\_num\_threads();**

Returns the number of threads in the current team.

**int omp\_get\_max\_threads();**

Returns maximum number of threads that could be used to form a new team using a parallel construct without a `num_threads` clause.

**int omp\_get\_thread\_num();**

Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

**int omp\_get\_num\_procs();**

Returns the number of processors available to the program.

### Data Types For Runtime Library Routines

`omp_lock_t`: Represents a simple lock.  
`omp_nest_lock_t`: Represents a nestable lock.  
`omp_sched_t`: Represents a schedule.

## Clauses

The set of clauses that is valid on a particular directive is described with the directive. Most clauses are described as being applied to the entire directive. All clauses appearing in a clause must be visible.

### Data Sharing Attribute Clauses [2.9.3]

Data sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

**default(shared) none**

Controls the default data-sharing attributes of variables that are referenced in a parallel or task construct.

**shared([list])**

Declares one or more list items to be shared by tasks generated by a parallel or task construct.

**private([list])**

Declares one or more list items to be private to a task.

**private[private]([list])**

Declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item had when the construct is entered.

**lambda([list])**

Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

### reduction(operator|list)

Declares accumulation into the list items using the indicated associative operator. Reduction occurs into a private copy for each task, which is then combined with the original item.

Operators for reduction(initialization values)
<code>+ (0)</code>
<code>*</code>
<code>(1)</code>
<code>- (0)</code>
<code>&amp;&amp; (1)</code>
<code>&amp; (~0)</code>
<code>max (Least number in reduction list item type)</code>
<code>min (Largest number in reduction list item type)</code>

### Data Copying Clauses [2.9.4]

These clauses support the copying of data values from private or threadprivate variables on one thread or task or thread to the corresponding variables on other implicit tasks or threads in the team.

**copyin([list])**

Copies the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

**copyprivate([list])**

Broadcasts a value from the data environment of one implicit task to the data environments of other implicit tasks belonging to the parallel region.

### Lock Routines [3.3]

Lock routines support synchronization with OpenMP locks.

**void omp\_set\_lock(lock\_t \*lock);**

**void omp\_unset\_lock(lock\_t \*lock);**

These routines initialize an OpenMP lock.

**void omp\_destroy\_lock(lock\_t \*lock);**

These routines ensure that the OpenMP lock is uninitialized.

**void omp\_set\_nest\_lock(lock\_t \*lock);**

**void omp\_unset\_nest\_lock(lock\_t \*lock);**

These routines provide a means of setting an OpenMP lock.

**void omp\_nest\_lock(lock\_t \*lock);**

**void omp\_unset\_nest\_lock(lock\_t \*lock);**

These routines provide a means of unsetting an OpenMP lock.

**int omp\_test\_lock(lock\_t \*lock);**

**int omp\_test\_nest\_lock(lock\_t \*lock);**

These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

### Timing Routines [3.4]

Timing routines support a portable wall clock timer.

**double omp\_get\_wtime();**

Returns elapsed wall clock time in seconds.

**double omp\_get\_wtick();**

Returns the precision of the timer used `omp_get_wtime`.

## Environment Variables

Environment variables are described in section [4] of the API specification. Each environment variable has a name, case, and the values assigned to them are case insensitive and may have leading and trailing white space.

**OMP\_SCHEDULE type(chunk)**

Sets the `run-sched-var` IV for the runtime schedule type and chunk size. When OpenMP schedule type is static, dynamic, guided, or user-defined, positive integer that specifies chunk size.

**OMP\_NUM\_THREADS int**

Sets the `nthreads-var` IV for the number of threads to use for parallel regions.

**OMP\_DYNAMIC dynamic**

Sets the `dynamic-var` IV for the dynamic adjustment of threads to use for parallel regions. Valid values for dynamic are true or false.

**OMP\_PROC\_BIND bind**

Sets the value of the global `bind-var` IV. The value of this environment variable must be true or false.

**OMP\_WAIT\_POLICY policy**

Sets the `wait-var` IV that controls the behavior of waiting threads. Valid values for policy are ACTIVE (waiting threads consume processor cycles while waiting) and PASSIVE.

**OMP\_MAX\_ACTIVE\_LEVELS levels**

Sets the `max-active-levels-var` IV that controls the maximum number of nested active parallel regions.

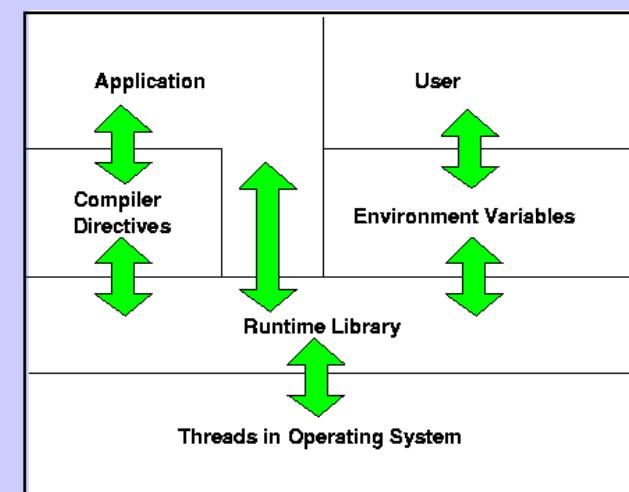
**OMP\_THREAD\_LIMIT limit**

Sets the `thread-limit-var` IV that controls the maximum number of threads participating in the OpenMP program.

Copyright © 2011 OpenMP Architecture Review Board. Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice is preserved on all copies. This document and the OpenMP logo are trademarks of the OpenMP Architecture Review Board. Products or publications based on one or more of the OpenMP specifications must acknowledge the copyright by displaying the following statement: "OpenMP trademark of the OpenMP Architecture Review Board. Portions of this product/publication may be derived from the OpenMP Language Application Program Interface Specification."

# OpenMP Funktionsübersicht

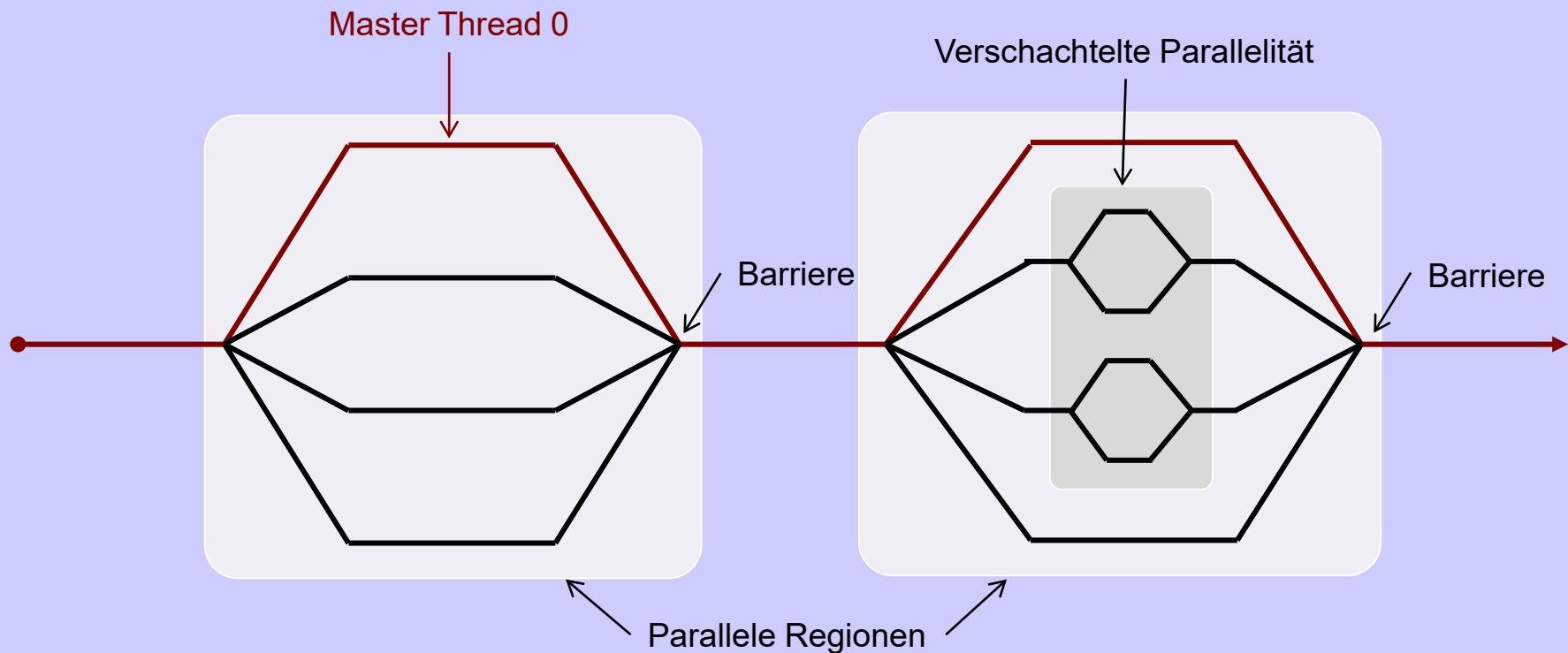
- Parallel Regionen
  - `omp parallel`
- Aufteilung von Schleifen
  - `omp for`
- Definition gemeinsamer/privater Daten
  - `omp shared`, `private`, ...
- Synchronisation
  - `omp atomic`, `barrier`, `critical`, ...
- Bibliotheksfunktionen
  - `omp_set_num_threads()`, `omp_get_thread_num()`,  
`omp_get_wtime()`, ...
- Umgebungsvariablen
  - `OMP_NUM_THREADS`, ...



# Programmiermodell

- Parallelität lässt sich schrittweise erzeugen
  - Ein zunächst sequentielles Programm wird nach und nach zu einem parallelen
  - Sequentielle Version weiterhin nutzbar
- Anweisung **parallel** startet ein Team von Threads
  - Aktueller Thread 0 ist Master
  - Alle Threads bearbeiten den folgenden Abschnitt
- Anweisung **for** teilt Iterationen auf Team-Mitglieder auf
  - Jeder Thread bearbeitet einen Teil
- Ende des parallelen Blocks
  - Implizite Synchronisation (Barriere)
  - Nur Master läuft weiter

# Struktur eines OpenMP-Programms



# Einfaches OpenMP-Beispiel 1

- Erzeugen von Parallelität:

```
#include <stdio.h> // fuer printf
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hallo Welt.\n");

    return 0;
}
```

- Ausgabe auf einem Dual-Core Rechner:

Hallo Welt.

Hallo Welt.

# Paralleler Abschnitt

```
#pragma omp parallel {parameter}  
{  
    // Anweisungsfolge  
}
```

- Die Anweisungsfolge wird von **allen** Threads, einschließlich dem Master-Thread parallel ausgeführt
- Aufteilung der Arbeit **manuell** durch den Programmierer oder mit weiteren Anweisungen (z.B. **for**, **sections**, ...)
- Wenn alle Threads fertig sind, arbeitet der Master sequentiell weiter (**implizite Synchronisation**)
- **Verschachtelung** möglich (aber je nach Compiler evtl. sequentielle Abarbeitung der inneren Abschnitte)

# Einfaches OpenMP-Beispiel 2

- Unterscheidung der Threads:

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Ich bin thread %d von %d.\n", omp_get_thread_num(),
           omp_get_num_threads());
    return 0;
}
```

- Ausgabe auf einem Dual-Core Rechner:

Ich bin thread 1 von 2.

Ich bin thread 0 von 2.

# Von Sequentiell zu Parallel

- **Sequentielles Programm:**

```
double A[10000];
for (int i = 0; i < 10000; i++) {
    A[i] = langwierige_berechnung(i);
}
```

- **Parallelisiertes Programm (automatische Aufteilung):**

```
double A[10000];
#pragma omp parallel
#pragma omp for
for (int i = 0; i < 10000; i++) {
    A[i] = langwierige_berechnung(i);
}
```

# Manuelle Aufteilung

- Paralleles Programm:

```
double A[10000];
int cnt = omp_get_num_threads();
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int i_start = 10000 * id/cnt;
    int i_end = 10000 * (id+1)/cnt - 1;
    langwierige_berechnung(A, i_start, i_end);
}
```

# for-Schleifen

- Aufteilung von Schleifeniterationen
  - OpenMP teilt Iterationen **automatisch** den einzelnen Threads zu

```
double A[10000];
#pragma omp parallel for
for (int i = 0; i < 10000; i++) {
    langwierige_berechnung(i);
}
```

- Seit OpenMP 3.0 sind auch Iterator-Schleifen parallelisierbar:

```
vector<int> v(10000);
typedef vector<int>::iterator iter;
#pragma omp parallel for
for (iter i = v.begin(); i < v.end(); i++) {
    langwierige_berechnung(*i);
}
```

# Bedingungen für **for**-Schleifen

## • Einschränkungen

- Nur **ganzzahlige** Schleifenvariablen **mit Vorzeichen**  
(ohne ab OpenMP 3.0)
- **Test** nur mit **<, <=, >, >=**
- Schleifenvariable verändern: Nur **einfache Ausdrücke**
  - Operatoren **+, -, ++, --, +=, -=**

```
++var, var++, --var, var--  
var += inkr, var -= inkr,  
var = var + inkr,  
var = inkr + var,  
var = var - inkr
```

- Obere und untere **Grenze unabhängig** von Schleifendurchlauf
  - Ausdrücke sind möglich

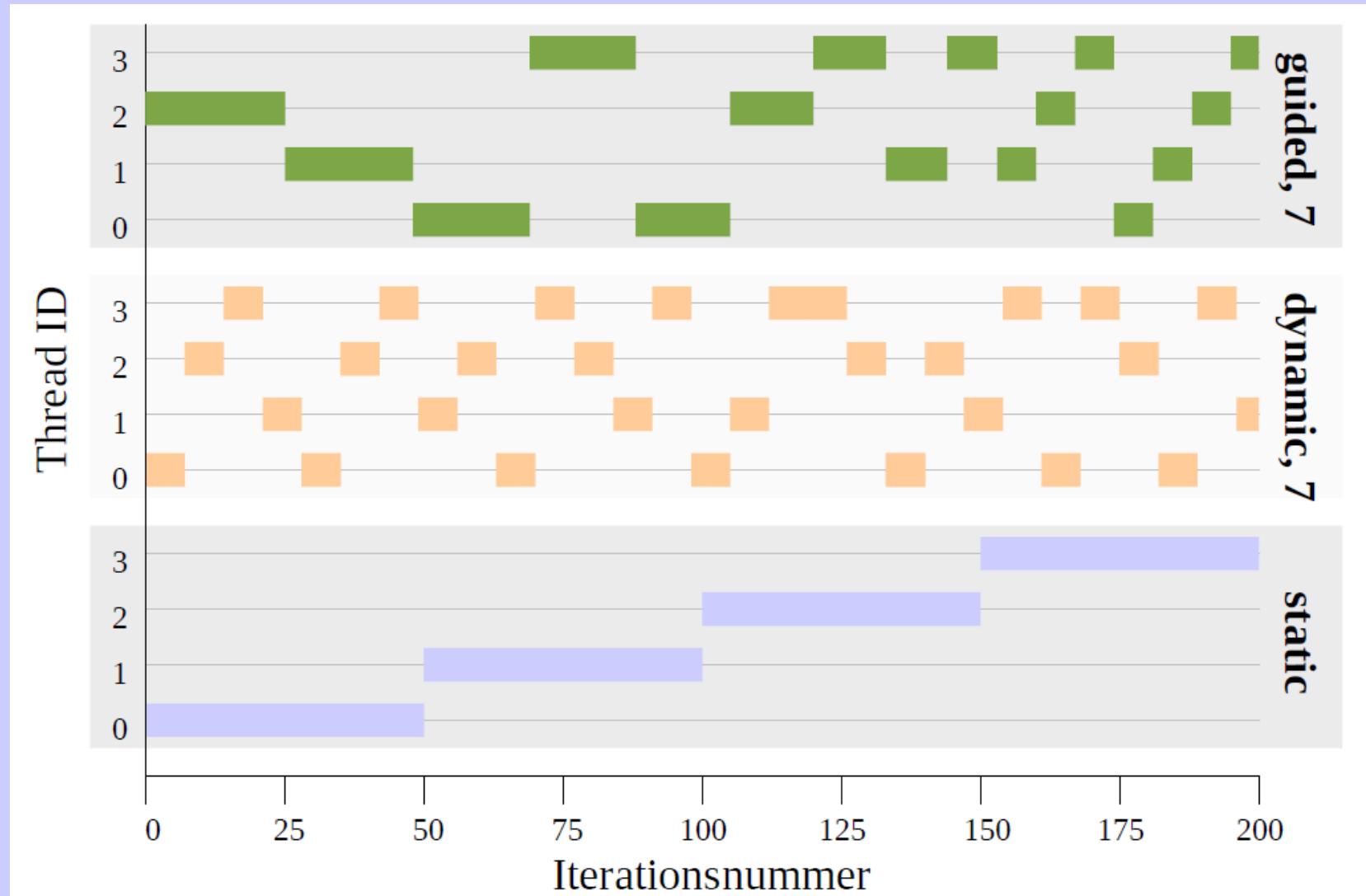
## • Automatische Synchronisation

- Standardmäßig implizite **Barriere** nach **for**
- Kann mit **nowait** unterdrückt werden

# Mit **for** Indexraum auf Threads verteilen

- **schedule(static[,k])**
  - Indexraum wird in Blöcke der Größe  $k$  zerlegt und den Threads **reihum** zugewiesen
  - $k=1$ : 012012...
  - $k=5$ : 000001111122222000001111122222...
- **schedule(dynamic[,k])**
  - Indexraum wird in Blöcke der Größe  $k$  zerlegt und den Threads **nach Bedarf** zugewiesen
- **schedule(guided[,k])**
  - Indexraum wird in Blöcke **proportional zur Restarbeit** auf Threads aufgeteilt und nach Bedarf zugewiesen;  $k$  = minimale Blockgröße
- **schedule(auto)**
  - Implementierungsabhängig (Standard, wenn keine Angabe)
- **schedule(runtime)**
  - Entscheidung zur Laufzeit (`omp_set_schedule`, `omp_get_schedule`, `OMP_SCHEDULE`)

# Schedulingstrategien



# Mehrdimensionale-Schleifen

- **Zusammenfassung von Schleifen** (ab OpenMP 3.0)
  - **for**-Anweisung wirkt nur auf die nächste Schleife
  - **collapse (n)** kombiniert den Indexraum der folgenden n Schleifen
- **Beispiel:**

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < dim1; i++)
    for (int j = 0; j < dim2; j++)
        for (int k = 0; k < dim3; k++) {
            // Nur hier darf beliebiger Code stehen!
            res[i] = log(fabs(i * j * k));
        }
```

# Steuerung der Thread-Anzahl

- Höchste Priorität: **Parameter** der Anweisung  
`#pragma omp parallel num_threads(4)`
- Mittlere Priorität: Aufruf der **Bibliotheksfunktion** vor parallelem Abschnitt:  
`omp_set_num_threads(2)`
- Niedrigste Priorität: **Umgebungsvariable**  
`OMP_NUM_THREADS = 8`
- Nachprüfen: Abfrage der Thread-Anzahl  
`omp_get_num_threads()`

# Zeitmessung (wall clock time)

- **Tatsächlich verbrauchte Zeit**  
(Summe von CPU-Zeit, I/O-Zeit und Kommunikationsaufwand)

```
double start, end;  
...  
start = omp_get_wtime();  
// zu messender Programmabschnitt  
...  
end = omp_get_wtime();  
printf ("Benoetigte Zeit: %f Sekunden\n", end - start);
```



# Sequentielle Version

- **Beispiel:** Skalare Multiplikation und Vektoraddition für zwei Vektoren  $x$  und  $y$  gleicher Größe und dem Skalar  $a$
- Die Berechnung einer Iteration ist unabhängig von den Ergebnissen der anderen Iterationen

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    for (unsigned int i = 0; i < num; i++) {
        y[i] += a * x[i];
    }
}
```

# Parallele Version – Fehler 1

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    #pragma omp parallel {
        for (unsigned int i = 0; i < num; i++) {
            y[i] += a * x[i];
        }
    }
}
```

- **Compiler-Fehler!**
  - Klammersetzung des parallelen Blocks

# Parallele Version - Fehler 2

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    #pragma omp parallel
    {
        for (unsigned int i = 0; i < num; i++) {
            y[i] += a * x[i];
        }
    }
}
```

- **Unbeabsichtigtes Ergebnis!**
  - N-fache Ausführung statt Aufteilung
  - Wettlaufsituation

# Parallele Version - Fehler 3

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    #pragma omp parallel
    {
        #pragma omp for
        {
            for (unsigned int i = 0; i < num; i++) {
                y[i] += a * x[i];
            }
        }
    }
}
```

- **Compiler-Fehler!**
  - Klammersetzung des **for**-Blocks
  - evtl. Zählvariable ohne Vorzeichen

# Parallele Version 4

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < num; i++) {
            y[i] += a * x[i];
        }
    }
}
```

- Ok, aber geht noch etwas eleganter und kürzer!

# Parallele Version 5

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    #pragma omp parallel for
    for (int i = 0; i < num; i++) {
        y[i] += a * x[i];
    }
}
```

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- **Mehrkern-Berechnung mit OpenMP**
  - Kommunikation zwischen Threads
    - Variablen-Zuordnung
  - Synchronisationstechniken
    - Kritische Abschnitte
    - Atomare Anweisungen
    - Reduktionsoperationen
    - Barrieren
    - Nicht-parallele Ausführung
    - Geordnete Ausführung

# Rückblick: Parallelle Version

```
// Berechnung y_i = a * x_i + y_i
void calc(float a, float x[], float y[], int num)
{
    #pragma omp parallel for
    for (int i = 0; i < num; i++) {
        y[i] += a * x[i];
    }
}
```

# Kommunikation zwischen Threads



# Kommunikation zwischen Threads

- Die Threads werden in einem **gemeinsamen Adressraum** (shared memory) ausgeführt
- Die Threads im Beispiel `calc()` können auf die meisten Variablen im parallelen Codeabschnitt zugreifen
- Um Daten zu senden, schreibt ein Thread in **gemeinsam genutzte Variablen**, von wo sie von anderen Threads gelesen werden können
- Manchmal ist dieses Verhalten aber unerwünscht!

# Ungeregelter Zugriff

## • Probleme

- Mehrere Threads versuchen **gleichzeitig** die gleiche Variable zu **ändern**
- Ein Thread **liest** eine Variable, während ein anderer die gleiche Variable gerade **ändert**

## • Ergebnis

- Der Wert der Variablen ist **undefiniert** oder hängt von der (unbekannten) **Reihenfolge** ab, in der die Threads auf die Variablen zugreifen
- Mehrere unabhängige Programmdurchläufe liefern **unterschiedliche Ergebnisse**

## • Fazit

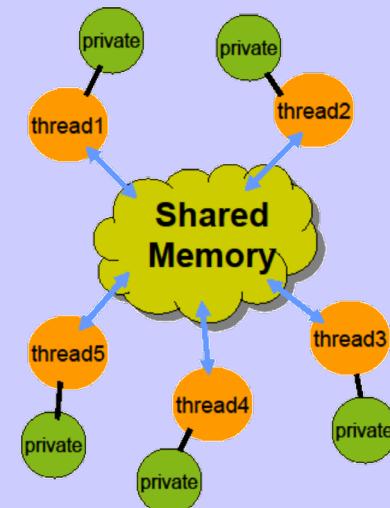
- **Unzulässig**, da das parallele Programm **in jedem Fall die gleichen Ergebnisse** liefern soll, wie das sequentielle Programm

# Kommunikation und Datenzugriff

- **shared** (Variablenliste)
  - Nur **eine Instanz** der Variable. Daten sind für **alle Threads sichtbar/änderbar**. Ein gemeinsamer Zugriff erfordert Synchronisation.
- **private** (Variablenliste)
  - Jeder Thread hat eine **lokale Kopie**, die außerhalb des parallelen Abschnitts **nicht bekannt** ist. Der Wert beim Ein- und Austritt in den parallelen Abschnitt ist **undefiniert!**
- **firstprivate** (Variablenliste)
  - Initialisierung privater Daten mit dem **letzten Wert** vor dem parallelen Abschnitt.
- **lastprivate** (Variablenliste)
  - Finalisierung privater Daten. Der Thread, der die **letzte Iteration** oder den **letzten parallelen Abschnitt** ausführt, übergibt den Wert an das Hauptprogramm.
- **threadprivate** (Variablenliste)
  - Globale Daten, die im parallelen Programmabschnitt jedoch als **privat** behandelt werden. Der **globale Wert** wird über den parallelen Abschnitt hinweg **bewahrt**.
- **reduction** (operator: list)
  - Private Daten, die am Ende auf **einen globalen Wert zusammengefasst** werden.

# Variablen-Zuordnung

- **default**(private / shared / none)
  - Das **Standardverhalten** der Zuordnung von Variablen kann mit dieser Anweisung angepasst werden
- **Standard-Zuordnung von Variablen**
  - Zählvariable einer parallelen Schleife: **private**
  - Innerhalb paralleler Abschnitte deklariert: **private**
  - Alle anderen: **shared**
- **Einschränkungen**
  - Es dürfen **mehrere private** und **shared** Anweisungen existieren
  - Eine Variable darf aber nur in **einer** dieser Anweisungen auftauchen
  - Eine Variable kann **nicht gleichzeitig** gemeinsam und privat genutzt werden



## falsch:

```
#pragma omp parallel private(x)
{
    #pragma omp for shared(x)
    ...
}
```

## zulässig:

```
#pragma omp parallel
{
    #pragma omp for private(x)
    ...
}
```

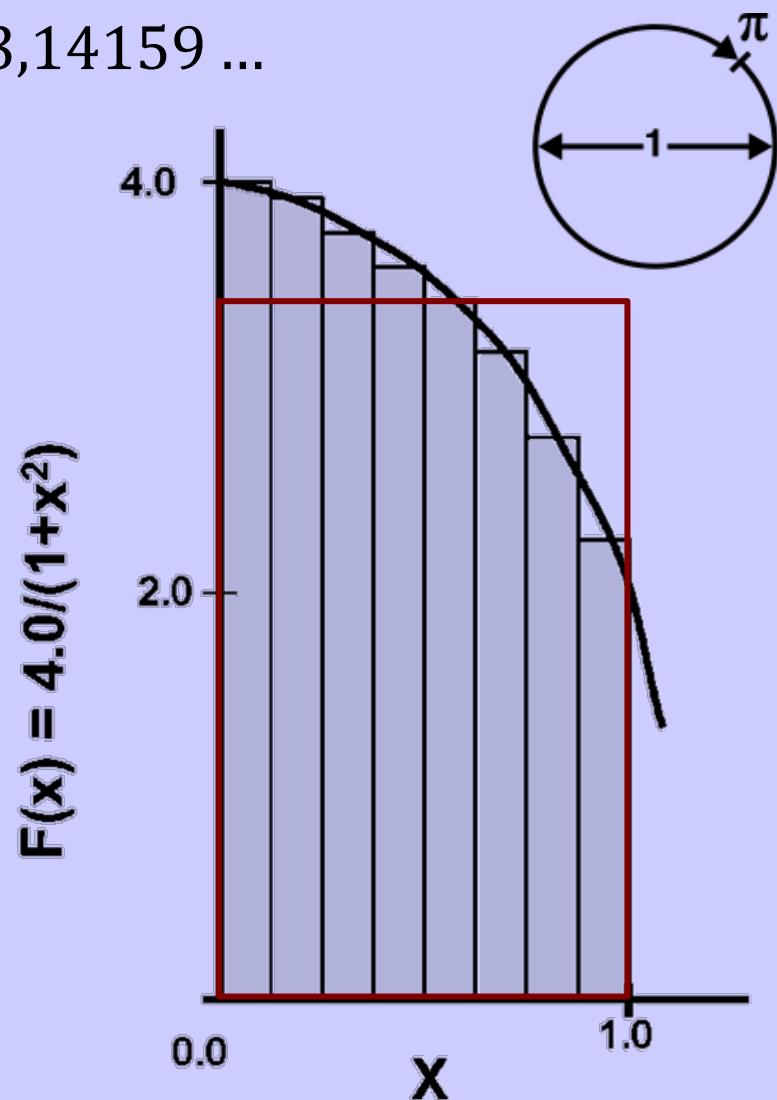
# Beispiel: Berechnung von $\pi$

- Iterative Berechnung der Kreiszahl  $\pi = 3,14159 \dots$
- **Algorithmus:**

$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx \approx \Delta x \cdot \sum_{i=1}^n \frac{4}{(1+x^2)}$$

mit  $\Delta x = \frac{1}{n}$  und  $x = \Delta x \cdot (i - 0.5)$

- **Beispiel:**
  - $n = 1: \pi = 3.200000$
  - $n = 10: \pi = 3.142426$
  - $n = 100: \pi = 3.141601$
  - $n = 1000: \pi = 3.141593$
  - 100 Millionen Iterationen für 13 Nachkommastellen



# Sequentielle Version

$$\pi \approx \Delta x \cdot \sum_{i=1}^n \frac{4}{(1+x^2)} \quad \text{mit } \Delta x = \frac{1}{n} \text{ und } x = \Delta x \cdot (i - 0.5)$$

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0;
int i;

for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}

return delta_x * sum;
```

# Parallele Version – Fehler 1

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0;
int i;

#pragma omp parallel for
for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}

return delta_x * sum;
```

- **Lokale Variablen** für  $x$  zum Zwischenspeichern erforderlich!

# Parallele Version – Fehler 2

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0;
int i;

#pragma omp parallel for private(x) shared(delta_x, sum)
for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}

return delta_x * sum;
```

- **Wettlaufsituation** bei gleichzeitiger Summenbildung!

# Kritischer Abschnitt

- Schreibzugriff wird durch einen **kritischen Abschnitt** geschützt

```
#pragma omp critical [ (name) ]
```



- Eine Synchronisation (**Mutex**) sorgt dafür, dass der nachfolgende Codeabschnitt nur von **einem Thread** aus dem Team **gleichzeitig** ausgeführt werden kann

# Parallele Version 3

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0;
int i;

#pragma omp parallel for private(x) shared(delta_x, sum)
for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    #pragma omp critical
    sum += 4.0 / (1.0 + x*x);
}

return delta_x * sum;
```

- Ok, aber zu **lange Berechnung** im kritischen Abschnitt!

# Parallele Version 4

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0, f_x;
int i;

#pragma omp parallel for private(x, f_x) shared(delta_x, sum)
for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    f_x = 4.0 / (1.0 + x*x);
    #pragma omp critical
        sum += f_x;
}

return delta_x * sum;
```

# Atomare Operationen

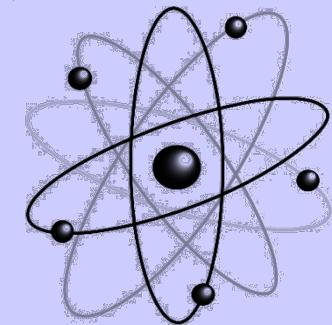
- Die Ausführung einzelner elementarer Schritte soll durch einen anderen Thread **nicht unterbrochen** werden
- Das Lesen und Verändern einer skalaren Variable soll nach außen als **unteilbare atomare Operation** erscheinen
- Anstelle eines kompletten Codeabschnitts muss nur **eine Speicherstelle** vor unerlaubtem Parallelzugriff geschützt werden (Laufzeitvorteile, da geringerer Verwaltungsaufwand)

**#pragma omp atomic**

Zuweisung

- **Erlaubte Zuweisungen** für atomare Operationen, z.B.:

=, +=, -=, \*=, &=, ^=, |=, x++, ++x, x--, --x



# Parallele Version 5

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0;
int i;

#pragma omp parallel for private(x) shared(delta_x, sum)
for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    #pragma omp atomic
    sum += 4.0 / (1.0 + x*x);
}

return delta_x * sum;
```

# Reduktions-Operator

- Mehrere Threads **akkumulieren einen Wert** und das Ergebnis hängt nicht von der Ausführungsreihenfolge ab (binäre Operationen sind kommutativ und assoziativ)

`reduction (op : Variable [, Variable] ...)`

- Für alle Variablen der Liste wird eine lokale Kopie angelegt und mit dem zum Operator passenden **neutralen Element initialisiert**



Operator	+	*	-	&		^	&&	
Neutrales Element	0	1	0	~0	0	0	1	0

- In jedem Thread werden Zwischenergebnisse **unsynchronisiert** akkumuliert
- Am Ende** des parallelen Abschnitts werden die Teilergebnisse in die ursprüngliche Variable des **Master-Threads** synchron aufakkumuliert

# Parallele Version 6

```
// Berechnung von pi
float delta_x = 1.0/num;
float x, sum = 0.0;
int i;

#pragma omp parallel for private(x) reduction(+:sum) shared(delta_x)
for (i = 1; i <= num; i++) {
    x = delta_x * (i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}

return delta_x * sum;
```

# Genauigkeitsanalyse

- Bei paralleler Ausführung können Zwischenergebnisse in anderer Reihenfolge berechnet werden
- **Rundungsfehler** bei Operationen auf Gleitkommazahlen akkumulieren sich anders, als bei sequentieller Berechnung
- Beispiel (mit **reduction**):  
Kreiszahl nach 100 Millionen Iterationen auf einem 32bit-System
  - 1 Thread:  $\pi = 3.1415926535904$
  - 4 Threads:  $\pi = 3.1415926535897$Parallele Variante ist hier sogar genauer!

# Laufzeitanalyse

- Vergleich der **Laufzeiten** auf einem **Dual-Core** Rechner bei den verschiedenen Synchronisationstechniken
- Berechnung von  $\pi$  mit 100.000.000 Iterationen
- **reduction** mit fast optimalem Speedup-Faktor 1,95!

Version	Sekunden
sequentiell	2,77
<b>critical</b>	30,98
<b>atomic</b>	28,52
<b>reduction</b>	1,42

# Bedingte Parallelisierung

- Die **Kosten** für das Starten, Verwalten und Beenden von Threads sind manchmal größer als der erhoffte Effekt der Parallelisierung
- Das Programm läuft möglicherweise **langsamer** als vorher
- Der Mehraufwand lohnt sich erst ab einer gewissen **Problemgröße**

```
#define elemente 10000

double A[elemente];

#pragma omp parallel for if (elemente > 100)
for (int i = 0; i < elemente; i++) {
    A[i] = kurze_berechnung(i);
}
```

# Synchronisationstechniken

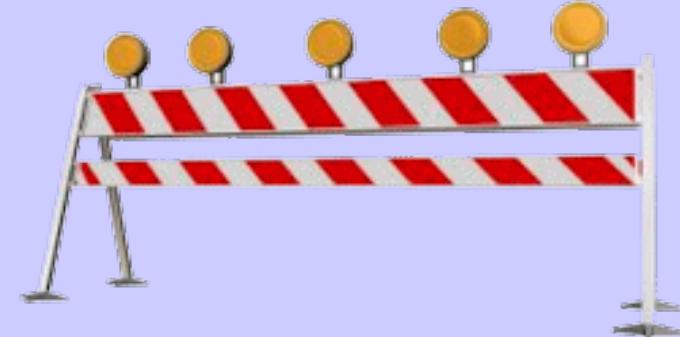


# Koordinieren von Ereignissen

- Bisher **Mutex-Sychronisationen** (critical, atomic, reduction), bei denen nur **ein Thread gleichzeitig** einen geschützten Bereich ausführen kann.
- OpenMP bietet aber weitere Möglichkeiten auf die **Ausführungsreihenfolge** der Threads Einfluss zu nehmen
  - Barrieren
  - Nicht-parallele Ausführung
  - Geordnete Ausführung

# Barrieren

- **Implizite Barrieren**
    - Am Ende von **parallel**-Abschnitten oder z.B. **for**-Schleifen
    - Alle Threads aus einem Team warten, bis alle den parallelen Abschnitt durchlaufen haben
  - **Implizite Barrieren umgehen**
    - Threads könnten bereits weitere Berechnungen im parallelen Abschnitt ausführen, statt untätig zu warten
    - Ist die Korrektheit sichergestellt, genügt das Anhängen der Anweisung **nowait**, um die **Barriere zu deaktivieren**
    - Die Barriere am Ende des **parallel**-Abschnitts kann **nicht** umgangen werden!
  - Barrieren können auch **explizit** festgelegt werden
    - Das Programm wird erst dann weiter ausgeführt, wenn der Letzte aus dem Team diesen Synchronisationspunkt erreicht hat
- #pragma omp barrier



# Beispiel: Implizite Barriere aufheben

```
void init(int a[], int b[], int c[], int size)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < size; i++)
            b[i] = a[i] * a[i];

        #pragma omp for nowait
        for (i = 0; i < size; i++)
            c[i] = a[i]/2;
    } // implizite Barriere
    ...
}
```

# Nicht-parallele Ausführung

- Bestimmte Anweisungen sollen **sequentiell** ausführt werden, ohne den parallelen Abschnitt verlassen zu müssen
- **Beispiel:** Initialisierung von Datenstrukturen oder I/O-Operationen
- Die Anweisung **single** sorgt dafür, dass der nachfolgende Codeblock von genau **einem Thread** des Teams durchlaufen wird (nicht notwendigerweise dem Master-Thread)

**#pragma omp single**

- Am Ende des Codeblocks existiert eine **implizite Barriere**, sofern nicht **nowait** angegeben wird
- Die Anweisung **master** sorgt dafür, dass der nachfolgende Codeblock ausschließlich vom Master-Thread durchlaufen wird

**#pragma omp master**

- Es existiert **keine implizite Barriere!**

# Beispiel: Single-Thread

```
int main(void)
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp single
        printf("read input\n"); // Nur ein Thread liest die Eingabe
        // implizite Barriere

        printf("compute results\n"); // Mehrere Threads berechnen das Ergebnis

        #pragma omp single
        printf("write output\n"); // Nur ein Thread schreibt die Ausgabe
        // implizite Barriere
    } ...
}
```

## Ausgabe:

```
read input
compute results
compute results
write output
```

# Die **copyprivate**-Anweisung

- Die **copyprivate**-Anweisung darf nur an eine **single**-Anweisung angehängt werden

**copyprivate** (Liste von Variablen)

- Nachdem der **single**-Block durch genau einen Thread ausgeführt wurde, werden die Werte der Variablen aus der Liste den **privaten Kopien** aller anderen Threads **zugewiesen**
- Es muss dann kein Umweg über gemeinsam genutzte Variablen genommen werden

# Beispiel: Master-Thread und Barriere

```
int main(void)
{
    int a[5], i;

    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++) // Eine Berechnung ausführen
            a[i] = i * i;           // implizite Barriere

        #pragma omp master
        for (i = 0; i < 5; i++) // Zwischenergebnisse anzeigen
            printf("a[%d] = %d\n", i, a[i]);
        #pragma omp barrier // Warten, da keine implizite Barriere

        #pragma omp for
        for (i = 0; i < 5; i++) // Berechnung fortsetzen
            a[i] += i;           // implizite Barriere
    } // implizite Barriere
    ...
}
```

# Geordnete Ausführung

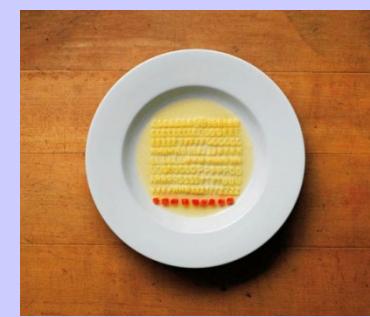
- Anweisungen in einer parallelen `for`-Schleife sollen **in bestimmter Reihenfolge** ausführt werden
- **Beispiel:**  
Schreiben von Werten auf den Bildschirm oder in eine Log-Datei
- Die Anweisung `ordered` muss **sowohl** bei der `for`-Schleife **als auch** beim entsprechenden Codeblock angegeben werden

```
#pragma omp parallel for ordered
```

...

```
#pragma omp ordered
```

- Der Code vor und nach dem geordneten Abschnitt wird weiter **parallel** ausgeführt
- Vor Eintritt in den geordneten Abschnitt wartet jeder Thread, bis die (sequentiell gesehen) **vorhergehende Iteration** durchgelaufen ist
- Mehrere geordnete Abschnitte in einer Schleife sind möglich.  
In jeder Iteration darf aber nur **ein georderter Abschnitt** durchlaufen werden!



# Beispiel: Geordnete Ausführung 1/2

```
int main(void)
{
    #pragma omp parallel for ordered      // als Klausel
    for (int i = 0; i < 10; i++) {
        #pragma omp ordered                // als Direktive
        printf("Iteration %d\n", i);
    }
    ...
}
```

# Beispiel: Geordnete Ausführung 2/2

```
#include <omp.h>
#include <stdio.h>
#include <iostream>

int main(void)
{
    #pragma omp parallel for ordered
    for (int i = 0; i < 10; i++) {
        if (i < 5) {
            #pragma omp ordered
            printf("Iteration %d\n", i);
        } else {
            #pragma omp ordered
            std::cout << "Iteration " << i << "b" << std::endl;
        }
    }
    ...
}
```

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- **Mehrkern-Berechnung mit OpenMP**
  - **Parallele Abschnitte**
    - Konsistente Speicherbelegung
    - Verschachtelte parallele Abschnitte
    - Parallelisierung mit Tasks ([OpenMP 3.0](#))
  - **Datenabhängigkeiten**
  - **Explizite Locks / Mutexe**

# Parallele Abschnitte



# Parallele Abschnitte (Statisch)

- Bisher nur Parallelisierung von Schleifen, d.h. iterativen Berechnungen
- OpenMP bietet aber auch Techniken zur Parallelisierung von **nicht-iterativen unabhängigen** Aufgaben
- Die Anweisung **sections** verteilt die Arbeit, indem voneinander unabhängige Codeblöcke jeweils genau **einmal einem Thread** aus dem Team **statisch** zugewiesen werden.

```
#pragma omp sections
{
    #pragma omp section // Angabe beim ersten Block optional
    {
    }

    #pragma omp section
    {
    }
}
```

- Am Ende des **sections**-Blocks existiert eine **implizite Barriere**

# Beispiel: Parallelle Abschnitte 1/2

```
void init(int a[], int b[], int c[], int size)
{
    int i;
    #pragma omp parallel sections // Zusammengefasste Kurzform
    {
        #pragma omp section
        for (i = 0; i < size; i++) // Komplette Schleifenberechnung
            b[i] = a[i] * a[i];    // auf einem Thread

        #pragma omp section
        for (i = 0; i < size; i++) // Vollständige Schleife
            c[i] = a[i] + 2;       // auf anderem Thread
    }
}
```

# Beispiel: Parallelle Abschnitte 2/2

```
void init(int a[], int b[], int c[], int size)
{
    int i;

#pragma omp parallel num_threads(2)
{
    #pragma omp sections nowait
    {
        for (i = 0; i < size; i++)
            b[i] = a[i] * a[i];

        #pragma omp section
        for (i = 0; i < size; i++)
            c[i] = a[i] + 2;
    }
    // ...

} // implizite Barriere
}
```

# Konsistente Speicherbelegung

- Die Threads teilen sich einen **gemeinsamen Speicherbereich**, in dem sie Variablen lesen und schreiben können
- Einer **OpenMP-Implementierung** ist es aber zur Effizienzsteigerung erlaubt, für jeden Thread einen **temporären Zustand** des gemeinsamen Speichers zu verwalten (vergleichbar mit einem Cache)
- Wenn ein Thread eine **gemeinsam genutzte Variable** ändert, kann es sein, dass ein anderer Thread noch den alten Wert liest
- Um die Variablenbelegungen **abzugleichen** gibt es die Anweisung

**#pragma omp flush [ (Liste von Variablen) ]**

- Entspricht einer **Barriere**, bis alle Lese- und Schreibzugriffe auf gemeinsame Variablen abgeschlossen sind
- Der Compiler synchronisiert nur diejenigen Variablen, auf die ein anderer Thread Zugriff haben könnte

# Beispiel: flush-Anweisung

```
int main(void)
{
    int data, flag = 0;

#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    {
        read(&data);
        #pragma omp flush(data)
        flag = 1;
        #pragma omp flush(flag)
        // ...
    }
    #pragma omp section
    {
        while (!flag) {
            #pragma omp flush(flag)
        }
        #pragma omp flush(data)
        process(&data);
    }
}
return 0;
}
```

# Verschachtelte parallele Abschnitte

```
#pragma omp parallel
{
    // ...
    #pragma omp parallel
    {
        // ...
    }
}
```

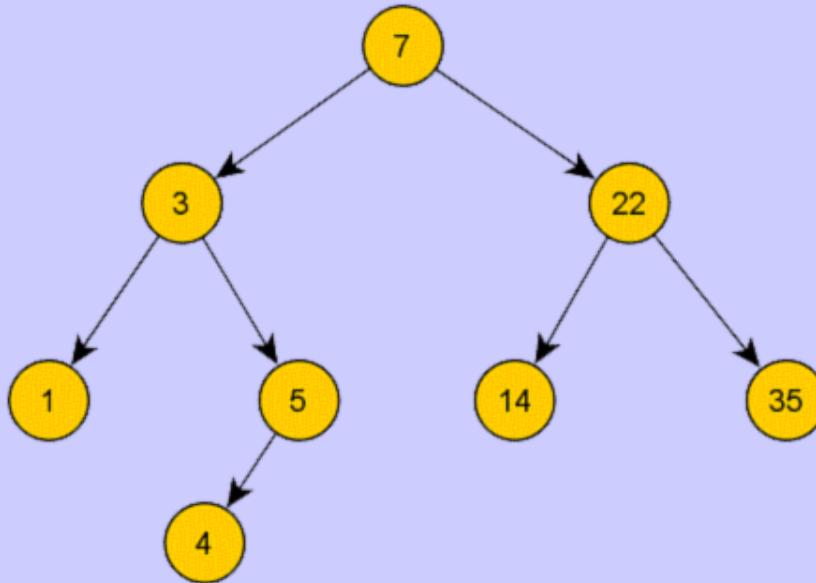
- Der OpenMP-Standard erlaubt **verschachtelte** Parallelität!
- Effiziente **Implementierung** ist **schwierig** für die Compiler-Hersteller und derzeit aktives Forschungsgebiet ...
- Abfragen bzw. Einstellen der Verschachtelung:
  - `int omp_in_parallel()` Aufruf in parallelem Abschnitt?
  - `int omp_get_nested()` Ist Verschachtelung aktiviert?
  - `void omp_set_nested(1)` Aktiviert verschachtelte Parallelität (1 = true), Standard (0 = false)

# Problem: Verschachtelungstiefe

```
void recursive(..., int depth)
{
    // ...
    if (depth <= 1) { // maximale Verschachtelungstiefe erreicht
        recursive(..., depth);           // sequentiell
        // ...
    } else {
        #pragma omp parallel
        {
            recursive(..., depth-1); // parallel
            // ...
        }
    }
}
```

- **Trick:** Manuelle Begrenzung der Verschachtelungstiefe

# Irreguläre Daten- und Kontrollstrukturen



Listen oder Bäume,  
While-Schleifen oder Rekursion

# Parallelisierung mit Tasks (OpenMP 3.0)

- Trifft ein Thread auf eine Task-Anweisung

```
#pragma omp task [...]  
// Codeblock
```

so kann er den Task **sofort selbst** ausführen oder seine Ausführung auf später **verschieben**. In diesem Fall wird der Task später durch einen **beliebigen Thread** aus dem Team ausgeführt

- Im Gegensatz zu allen anderen parallelen Methoden (Schleifeniterationen, Abschnitte) sind Tasks nicht an einen Thread gebunden. **Ein Task** kann zu verschiedenen Zeitpunkten von **verschiedenen Threads** ausgeführt werden.
- Jeder Task hat seinen eigenen **privaten Datenbereich**. Führt ein Thread einen Task aus, so geschieht das unter Verwendung des Datenbereichs des Tasks (und nicht des Threads).
- Die Anweisung

```
#pragma omp taskwait
```

hat Ähnlichkeit mit einer **Barriere**. Die Ausführung eines Tasks ist erst dann möglich, wenn **alle** seit Beginn des Tasks neu **generierten Tasks beendet** sind

# Beispiel: Fibonacci mit Tasks

```
int fib(int n)
{
    int x, y;

    if (n < 2) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x + y;
}
```

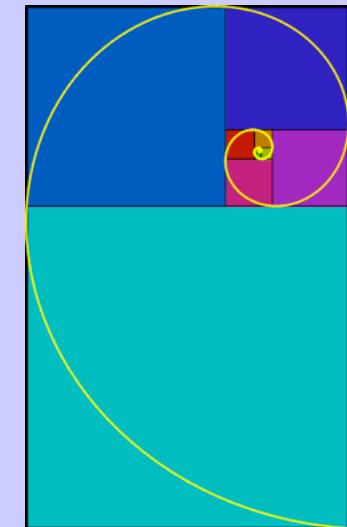
```
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        int x = fib(40);
    }
    return 0;
}
```

- **Fibonacci-Folge**

mit  $f_n = f_{n-1} + f_{n-2}$  (Summe der beiden Vorgänger)

für  $f_0=0$  und  $f_1=1$  (Initialisierung)

- **Beispiel:** 0, 1, 1, 2, 3, 5, 8, 13, 21, ...



# Datenabhängigkeiten



# Datenabhängigkeiten

- **Problem:** Manchmal hängt das Ergebnis eines Programmteils von einer bestimmten Ausführungsreihenfolge ab
- Die Iterationen von Schleifen werden automatisch auf verschiedene Threads verteilt
- Es dürfen also keine Datenabhängigkeiten zwischen verschiedenen Iterationen der Schleife existieren
- Variablen aus denen nur gelesen wird, sind uninteressant
- Wenn **schreibend** auf eine Variable zugegriffen wird und **mehrere** Schleifendurchläufe darauf zugreifen, dann liegt eine Datenabhängigkeit vor

# Datenabhängigkeiten finden

- Einfache Datenabhängigkeit

```
for (int i=1; i<num; i++) {  
    vec[i] = vec[i] + 3*vec[i-1];  
}
```

- Datenabhängigkeiten identifizieren?

```
for (int i=1; i<num; i+=2) {  
    vec[i] = vec[i] + 3*vec[i-1];  
}
```

```
for (int i=0; i<num/2; i++) {  
    vec[i] = vec[i] + 3*vec[i+num/2];  
}
```

```
for (int i=0; i<num/2+1; i++) {  
    vec[i] = vec[i] + 3*vec[i+num/2];  
}
```

// Annahme: vec[] ist groß genug

# Typen von Datenabhängigkeiten

- Seien die Anweisungen  $A_1$  und  $A_2$  gegeben, wobei  $A_1$  in der sequentiellen Ausführung vor  $A_2$  liegt, dann gilt:
  - **Echte Datenabhängigkeit** (*true dependence*) / **Flussabhängigkeit** wenn  $A_1$  in eine Speicherstelle schreibt, die von  $A_2$  als Eingabe gelesen wird, d.h. der Ausgabewert von  $A_1$  fließt in die Eingabe von  $A_2$
  - **Gegenabhängigkeit** (*antidependence*) wenn  $A_1$  von einer Speicherstelle liest, die anschließend von  $A_2$  überschrieben wird
  - **Ausgabeabhängigkeit** (*output dependence*) von  $A_2$  zu  $A_1$  besteht, wenn beide Anweisungen in die gleiche Speicherstelle schreiben und der Schreibzugriff von  $A_2$  nach dem von  $A_1$  erfolgt

# Beseitigung von Abhängigkeiten

- **Gegenabhängigkeiten** und **Ausgabeabhängigkeiten** entstehen durch Mehrfachnutzung von Speicherstellen
- Datenabhängigkeiten dieser Typen können immer durch konsistente Variablenumbenennungen entfernt werden
- Daher stammt der Oberbegriff **Namensabhängigkeiten** für diese beiden Typen
- **Echte Datenabhängigkeiten** können dagegen nicht immer entfernt werden

# Gegenabhängigkeit entfernen

- Gegenabhängigkeit

```
for (int i=0; i<num-1; i++) {  
    vec[i] = vec[i+1] + rand();  
}
```

- Parallelisierte Schleife ohne Gegenabhängigkeit

```
#pragma omp parallel  
{  
    #pragma omp for  
    for (int i=0; i<num-1; i++)  
        vec2[i] = vec[i+1];  
  
    #pragma omp for  
    for (int i=0; i<num-1; i++)  
        vec[i] = vec2[i] + rand();  
}
```

# Ausgabeabhängigkeit entfernen

- Ausgabeabhängigkeit

```
float d, e = -1.2, x = 0.0, y;  
for (int i=0; i<num; i++) {  
    x = (vb[i] + vc[i])/2.0;  
    va[i] += x;  
    d = 2*x;  
}  
y = x + d + e;
```

- Parallelisierte Schleife ohne Ausgabeabhängigkeit

```
#pragma omp parallel for lastprivate(x, d)  
for (int i=0; i<num; i++) {  
    x = (vb[i] + vc[i])/2.0;  
    va[i] += x;  
    d = 2*x;  
}  
y = x + d + e;
```

# Echte Abhängigkeit entfernen 1

- Echte Datenabhängigkeit

```
for (int i=1; i<num; i++) {  
    vb[i] += va[i-1];  
    va[i] += vc[i];  
}
```

- Keine Datenabhängigkeit durch Vor- und Nachberechnung

```
vb[1] += va[0];  
for (int i=1; i<num-1; i++) {  
    va[i] += vc[i];  
    vb[i+1] += va[i];  
}  
va[num-1] += vc[num-1];
```

# Echte Abhängigkeit entfernen 2

- Echte Datenabhängigkeit

```
float y = 0.0;
for (int i=1; i<num; i++) {
    va[i] += va[i-1];
    y += vc[i];
}
```

- Auslagerung der nicht behebbaren Datenabhängigkeit

```
for (int i=1; i<num; i++) {
    va[i] += va[i-1];
}
#pragma omp parallel for reduction(+:y)
for (int i=1; i<num; i++) {
    y += vc[i];
}
```

# Echte Abhängigkeit entfernen 3

- Echte Datenabhängigkeit

```
float x = 3.0;
for (int i=0; i<num; i++) {
    x += va[i];
    vb[i] = (vb[i] + vc[i]) * x;
}
```

- Neue Variablen mit Zwischenergebnissen anlegen

```
t[0] = x + va[0];
for (int i=1; i<num; i++) {
    t[i] = t[i-1] + va[i];
}
x = t[num-1];
#pragma omp parallel for
for (int i=0; i<num; i++) {
    vb[i] = (vb[i] + vc[i]) * t[i];
}
```

# Explizite Locks / Mutexe



# Locks / Mutexe

- OpenMP bietet auch explizite Synchronisation an. Es gibt einen Lock-Typ  
`omp_lock_t`  
Jeder Mutex ist entweder offen oder geschlossen.
- Folgende Funktionen werden bereitgestellt, um damit zu arbeiten:
  - `omp_init_lock(omp_lock_t*)`  
**Initialisiert** den Mutex. Der Mutex ist danach offen.
  - `omp_destroy_lock(omp_lock_t*)`  
**Zerstört** den Mutex.
  - `omp_set_lock(omp_lock_t*)`  
Versucht, den Mutex zu **schließen**. Wurde der Mutex bereits von einem anderen Thread geschlossen, so wird gewartet, bis dieser ihn wieder geöffnet hat.
  - `omp_unset_lock(omp_lock_t*)`  
**Öffnet** den Mutex wieder
  - `int omp_test_lock(omp_lock_t*)`  
**Versucht**, den Mutex zu schließen. Wurde der Mutex bereits von einem anderen Thread geschlossen, so wird 0 zurückgegeben, ansonsten 1.

# Beispiel: Locking – Fehler 1

```
omp_lock_t myLock;

#pragma omp parallel sections
{
    #pragma omp section
    {
        // ...
        omp_set_lock(&myLock);
        // ...
    }

    #pragma omp section
    {
        // ...
        omp_unset_lock(&myLock);
        // ...
    }
}
```

- Benutzung einer Lock-Variable ohne Initialisierung!

# Beispiel: Locking – Fehler 2

```
omp_lock_t myLock;
omp_init_lock(&myLock);

#pragma omp parallel sections
{
    #pragma omp section
    {
        // ...
        omp_set_lock(&myLock);
        // ...
    }
    #pragma omp section
    {
        // ...
        omp_unset_lock(&myLock);
        // ...
    }
}
```

- Lock-Freigabe von einem anderen Thread!

# Beispiel: Locking

```
omp_lock_t myLock;
omp_init_lock(&myLock);

#pragma omp parallel sections
{
    #pragma omp section
    {
        omp_set_lock(&myLock);
        // ...
        omp_unset_lock(&myLock);
    }
    #pragma omp section
    {
        omp_set_lock(&myLock);
        // ...
        omp_unset_lock(&myLock);
    }
}
```

# Rekursive Locks / Mutexe

- `omp_lock_t` ist ein nicht-rekursives Mutex, d.h. ein Thread darf einen Mutex **nicht mehrmals** schließen. Er würde in dem Fall auf sich selbst warten!
- Es gibt aber auch eine **rekursive Variante**

`omp_nest_lock_t`

- Die Funktionen, um damit umzugehen funktionieren analog
  - `omp_init_nest_lock(omp_nest_lock_t*)`
  - `omp_destroy_nest_lock(omp_nest_lock_t*)`
  - `omp_set_nest_lock(omp_nest_lock_t*)`
  - `omp_unset_nest_lock(omp_nest_lock_t*)`
  - `int omp_test_nest_lock(omp_nest_lock_t*)`

# Beispiel-Wrapper für C++

```
class Mutex
{
public:
    Mutex() {omp_init_lock(&lock);}
    ~Mutex() {omp_destroy_lock(&lock);}
    void lock() {omp_set_lock(&lock);}
    void unlock() {omp_unset_lock(&lock);}
    bool try_to_lock() {return omp_test_lock(&lock);}

private:
    Mutex(const Mutex&); // Copy constructor
    Mutex&operator=(const Mutex&); // Assignment operator
    omp_lock_t lock;
};

class ScopedLock
{
public:
    ScopedLock(Mutex&lock) :lock(lock) {lock.lock();}
    ~ScopedLock() {lock.unlock();}

private:
    ScopedLock(const ScopedLock&); // Copy constructor
    ScopedLock&operator=(const ScopedLock&); // Assignment operator
    Mutex&lock;
};
```

- **Problem:**  
Lock-Freigabe vergessen
- Wächterobjekt  
(*guard object*) in C++
- RAI-Prinzip (*resource acquisition is initialisation*)
- *Scoped locking*  
ist ausnahmefest
- **Wichtig:** die Lock-Variable  
vor dem parallelen  
Abschnitt anlegen und  
initialisieren, damit das  
Team sich über dieselbe  
Variable synchronisiert

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- **Mehrkern-Berechnung mit OpenMP**
  - Effiziente Parallelisierung
- **Programmierung verteilter Systeme mit MPI**
  - Interprozesskommunikation
  - Programmierung mit dem Message Passing Interface (MPI)
  - Senden und Empfangen



# **Effiziente Parallelisierung**

# Effiziente Parallelisierung

- OpenMP ist eine der **einfachsten** Möglichkeiten ein Programm zu parallelisieren
- In der **Praxis** ist das Erstellen eines vernünftig skalierbaren Programms aber **nicht trivial**
- **Korrektheit** ist eine notwendige Voraussetzung und darf nicht der Leistungssteigerung geopfert werden!
- Gerade beim **Einstieg** in OpenMP sind die Ergebnisse oft **ernüchternd**:
  - das parallelisierte Programm läuft um ein Vielfaches **langsamer** als vorher
  - es wurden nicht ausreichend **berechnungsintensive** Algorithmen ausgewählt
  - der zusätzliche **Verwaltungsaufwand** für mehrere Threads frisst die erreichbare Beschleunigung auf

# Lösungsansätze 1/5

- Ist die sequentielle Version ausreichend optimiert?
  - Teure Operationen vermeiden  
(z.B. Funktionsaufruf durch Look-Up-Table ersetzen)
  - Gemeinsam genutzten Speicherbereich klein halten  
(Cache-Misses verringern)
  - Nutzung von Parallelität auf anderen Ebenen versuchen
    - Nach Innen (z.B. durch erweiterte Maschinenbefehle eines Prozessors):
      - Multimedia:
        - MMX: Multi Media Extension (1997)
        - 3DNow!: (1998 – 2010)
        - SSE: Streaming SIMD Extensions (1999, SSE2 2001, SSE4 2008)
        - AVX2: Advanced Vector Extensions (2011, AVX2 2013)
      - Virtualisierung: VTx, AMD-V (2006), EPT (Extended Page Tables, 2012)
      - Verschlüsselung: AES (Advanced Encryption Standard, 2010)
    - Nach Außen:
      - Verteilte Systeme
      - Grafikkarten-Prozessoren (GPU), Koprozessoren (Intel Xeon Phi)

# Lösungsansätze 2/5

- Kann das Programm überhaupt skalieren?
- Welche Beschleunigung ist nach dem Amdahl'schen Gesetz zu erwarten?
  - Kritische Abschnitte erhöhen den sequentiellen Anteil
  - Die expliziten Lock-Funktionen erlauben eine exaktere Synchronisation auf gemeinsam genutzten Ressourcen als mit `critical`
  - Eigentlich gemeinsam genutzte Ressourcen jedem Thread als private Kopie mitgeben und am Ende die Ergebnisse sequentiell zusammenfassen, um den Synchronisationsaufwand zu minimieren

# Lösungsansätze 3/5

- **Welche Codebestandteile verbrauchen die meiste Rechenzeit?**
  - Mit einem **Performance-Profiler** lassen sich diejenigen Codeabschnitte identifizieren, die die meiste Rechenzeit verbrauchen
  - Konzentration auf häufig aufgerufene **problematische Abschnitte**
  - **Beispiele:** ompP, TAU, Scalasca, Intel vTune, GNU GPerfTools
- **Wurde die Gesamtanzahl paralleler Abschnitte gering gehalten?**
  - Das Starten und Beenden von Threads ist mit erheblichem **Verwaltungsaufwand** verbunden
  - Möglichst zwei **parallele Abschnitte zusammenfassen**, indem die dazwischen liegenden sequentiellen Anweisungen in einen **single** oder **master**-Block gefasst werden

# Lösungsansätze 4/5

- **Wurden verschachtelte Schleifen so weit außen wie möglich parallelisiert?**
  - Innen liegende parallele Abschnitte werden bei jeder Iteration der weiter außen liegenden Schleife betreten und tragen zu einer Multiplikation des Verwaltungsaufwandes bei
- **Wurden möglichst viele nowait-Anweisungen verwendet, um Wartezeiten an impliziten Barrieren zu minimieren?**
  - Dort, wo nicht aufgrund einer Wettlaufsituation notwendig, sollte **nicht unnötig** auf den letzten Thread des Teams **gewartet werden**
- **Sind alle kritischen Abschnitte benannt?**
  - Threads sollten nicht unnötig an **nicht** oder **gleich benannten** kritischen Abschnitten warten müssen, obwohl gar keine Notwendigkeit dazu besteht
  - Es sollte geprüft werden, ob ein **kritischer Abschnitt** durch eine **atomic**- oder **reduction**-Anweisung ersetzt werden kann

# Lösungsansätze 5/5

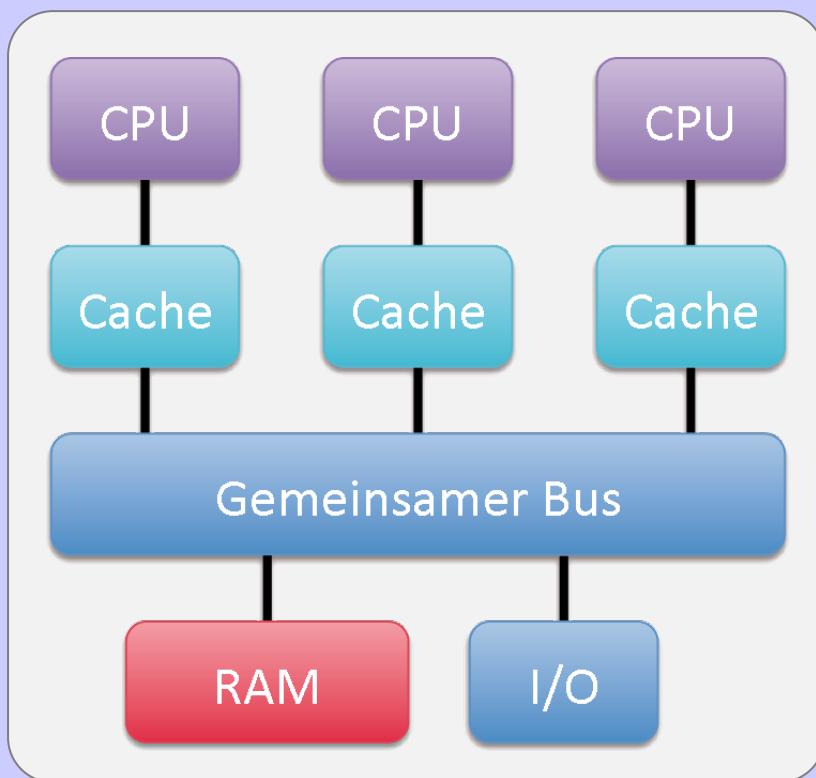
- **Arbeiten alle Prozessoren mit der gleichen Auslastung?**
  - Bei Ungleichgewicht der **Prozessorlast** muss die **Aufteilung** auf die Threads neu überdacht werden
  - Bei parallelen **Schleifen** genügt oft die Anpassung des **Ablaufplans** (z.B. dynamisch mit kleinerer Stückgröße)
  - Die parallele Ausführung von **kleinen Schleifen** mit wenig Rechenaufwand pro Iteration kann durch das Ergänzen einer **if**-Anweisung verhindert werden
- **Wurden die Möglichkeit zur Privatisierung von Daten optimal genutzt?**
  - Es ist bequem die OpenMP **Standard-Deklaration** von Variablen als **shared** zu belassen
  - **private** Variablen erhöhen jedoch die Lokalität im Sinne der **Cachenutzung**

# Verteilte Systeme

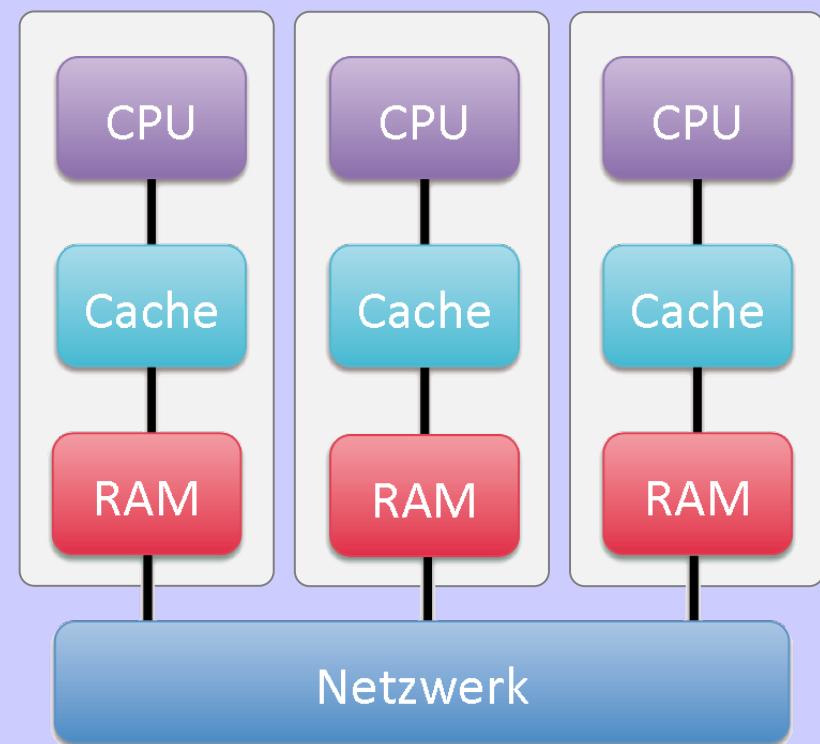


# Parallele Berechnung auf verteilten Systemen

Gemeinsamer Speicher



Verteilter Speicher

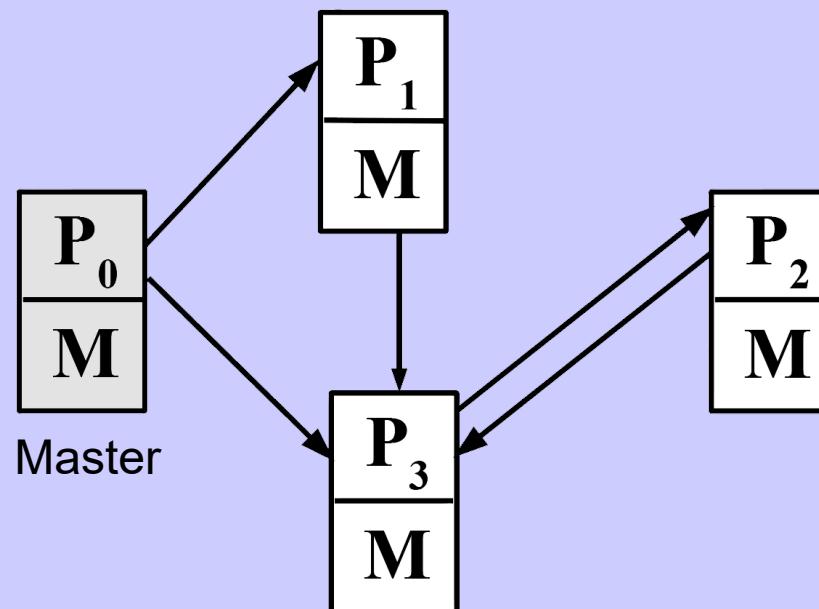


# Verteiltes System

- Es ist einfacher geworden, Systeme aus sehr **vielen Computern** mit einem **Hochgeschwindigkeitsnetzwerk** zusammenzubauen
- **Wichtige Aspekte**
  - **Hardware**: die Computer sind **autonom**
  - **Software**: der Benutzer glaubt, mit einem **einzigem System** zu agieren
- **Transparenz** (wünschenswerte Eigenschaft)
  - Dem Benutzer bleiben die **Unterschiede** einzelner Computer und wie sie verbunden sind, **verborgen**
  - Ein System soll **einfach** und für den Benutzer unsichtbar **erweiterbar** sein
- **Verteilte Anwendung**
  - **Dezentralisierte Prozesse** berechnen ein **gemeinsames Ergebnis**

# Modell Nachrichtenaustausch

- Mehrere **Prozesse**  $P_i$  laufen parallel
- Es existiert kein gemeinsamer **Speicher**  $M \Rightarrow$  Datenverteilung nötig
- **Kommunikation** zwischen Prozessen ausschließlich durch Senden und Empfangen von Nachrichten (**message passing**)
- Das **Senden und Empfangen** ist eine Aktivität zwischen zwei Prozessen, für die **beide explizit** Funktionen aufrufen müssen



# Interprozesskommunikation

- **Zweck:**

- **Datenverteilung** (Details sind vom Programmierer geplant)
- **Verbindungsaufbau** (Details zur Kommunikation sind Inhalt der Nachricht)
- **Synchronisation** (Spezialfall der Kommunikation)

- **Vorteil:**

- Der Nachrichtenaustausch funktioniert auch **über Rechnergrenzen** hinweg (z.B. über gemeinsamen Speicher, TCP/IP oder Infiniband)

- **Architekturen:**

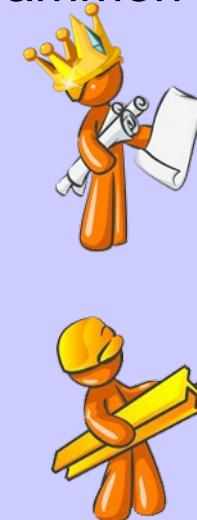
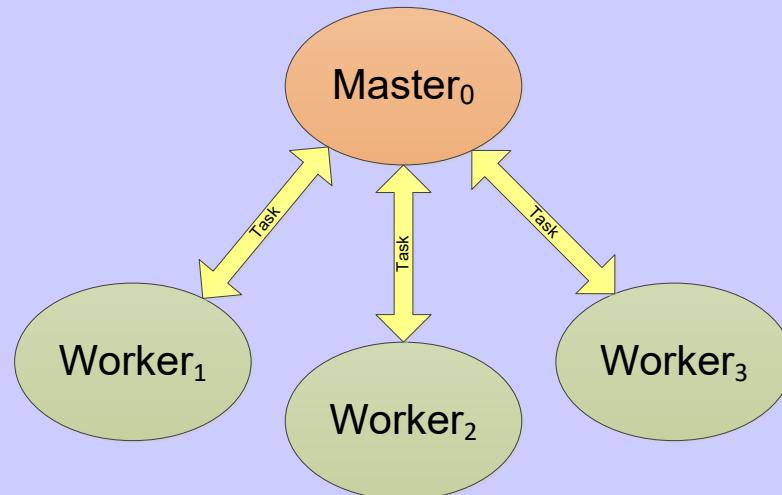
- Top500 Supercomputer, Computer **Cluster** oder **Multiprozessoren**
- Aber auch auf **Mehrkern-Systemen** (Portabilität)

- **Anzahl der Prozesse**

- Wird bei **Programmstart** festgelegt und bleibt während der Programmausführung **konstant**

# Master / ~~Slave~~ Worker

- Häufiges Muster zur **Strukturierung** von verteilten Programmen



- **Schema:**
  - Der Master sendet jedem Worker eine **Aufgabe (Task)**
  - Nach Beenden des Tasks sendet der Worker das **Ergebnis** an den Master zurück und erhält einen neuen Task
  - Vorgehen wird wiederholt, bis alle Tasks abgearbeitet sind
- Typischerweise unterscheiden sich Tasks nur in Parametern (**datenparallel**)
- Häufig rechnet der **Master** nicht mit

# Message Passing Interface (MPI)

## Versenden und Empfangen



# Message Passing Interface (MPI)

- **Standard für umfangreiche Bibliotheksfunktionen**

- Regelt die Kommunikation zwischen **Prozessen**
- Start von Prozessen, Abbildung von Prozessen auf Prozessoren
- Sprachbindung an **C / C++**, Fortran oder über Bibliothek (C#, Java, Python, ...)
- **Dokumentation:** [www mpi-forum org/docs/](http://www mpi-forum org/docs/)

- **MPI-Forum** (Firmen, Forschungseinrichtungen, Unis):

- 1994: MPI Version 1.0
- 1997: **MPI-2**: umfangreiche Erweiterung von MPI-1, ~130 Funktionen
  - z.T. „exotische“ Funktionalität
- 2009: MPI 2.2: Korrekturen, kleinere Erweiterungen
- 2012: Weiterentwicklung zu MPI 3.0, ~430 Funktionen
  - (einseitige Kommunikation, nichtblockierende kollektive Operationen)
- aktuell MPI Version 4.0.5

- **Implementierungen:**

- MPICH2 / MPICH - MPI over CHameleon ([www.mpich.org](http://www.mpich.org))
- **Open MPI** ([www.open-mpi.org](http://www.open-mpi.org))
- **Intel-MPI** 2019.9, **MS-MPI** 10.0 ...



# MPI-Grundfunktionen

- **MPI\_Init**
- **MPI\_Finalize**
- **MPI\_Comm\_size**
- **MPI\_Comm\_rank**
- **MPI\_Send**
- **MPI\_Recv**
- Alle Bibliotheksfunktionen, Datentypen, Konstanten beginnen mit **MPI\_** und sind in **mpi.h** deklariert
- Sie geben bei Erfolg **MPI\_SUCCESS** zurück



# Initialisieren und Finalisieren

- `int MPI_Init(int *argc, char *argv[])`
  - **Initialisiert** das MPI-Laufzeitsystem
  - **Davor** dürfen **keine** MPI-Funktionen aufgerufen werden
  - Übergebene Argumente entsprechen den **Kommandozeilenparametern** des Hauptprogramms `main`
- `int MPI_Finalize()`
  - **Meldet** Prozess beim MPI-Laufzeitsystem **ab**  
(Prozess wird nicht beendet)
  - **Danach** ist **kein** Aufruf von MPI-Funktionen erlaubt
  - Aufruf kann bis zum Abschluss aller MPI-Operationen **blockieren**

# Kommunikatoren und Identifikation

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `comm` ist der **Kommunikator** (MPI-interne Datenstruktur zur Verwaltung einer (Teil-)Menge von Prozessen)
- Für jeden Kommunikator sind die Prozesse durchnummeriert: 0, 1, ...
- **size** = Anzahl aller Prozesse im Kommunikator
- **rank** = eigene Nummer
- Bis auf Weiteres gilt: `comm = MPI_COMM_WORLD`  
(vordefinierte Menge aller Prozesse)

# Erstes MPI Beispiel

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

int main(int argc, char *argv[])
{
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello from process %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

- **Download** von [www.open-mpi.org/software/ompi/](http://www.open-mpi.org/software/ompi/) und installieren (Empfohlene **vorkompilierte Version** für Windows 1.6.1 32/64bit, Juli 2016)
- Den obigen Code **hello.cpp übersetzen** (z.B. mit **mpicc** oder **Visual Studio**)
- Das resultierende Programm **hello.exe** mit  $n$  Prozessen **ausführen**:

**mpirun -np  $n$  hello.exe**

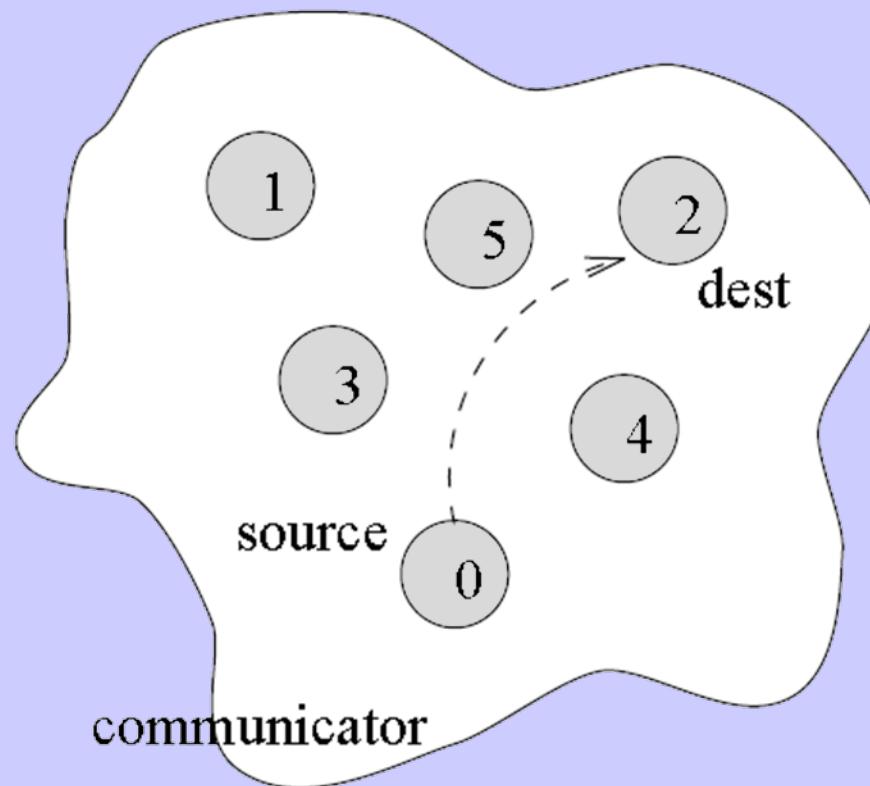
# Hinweis: Übergabe von Argumenten

```
int main(int argc, char *argv[ ] )  
{  
    // argc = 6? Zusätzliche Steuerparameter!  
    MPI_Init(&argc, &argv);  
    // argc = 3, argv[0] = "myprog.exe", argv[1] = "mympar1", argv[2] = "mympar2"  
    ...  
}
```

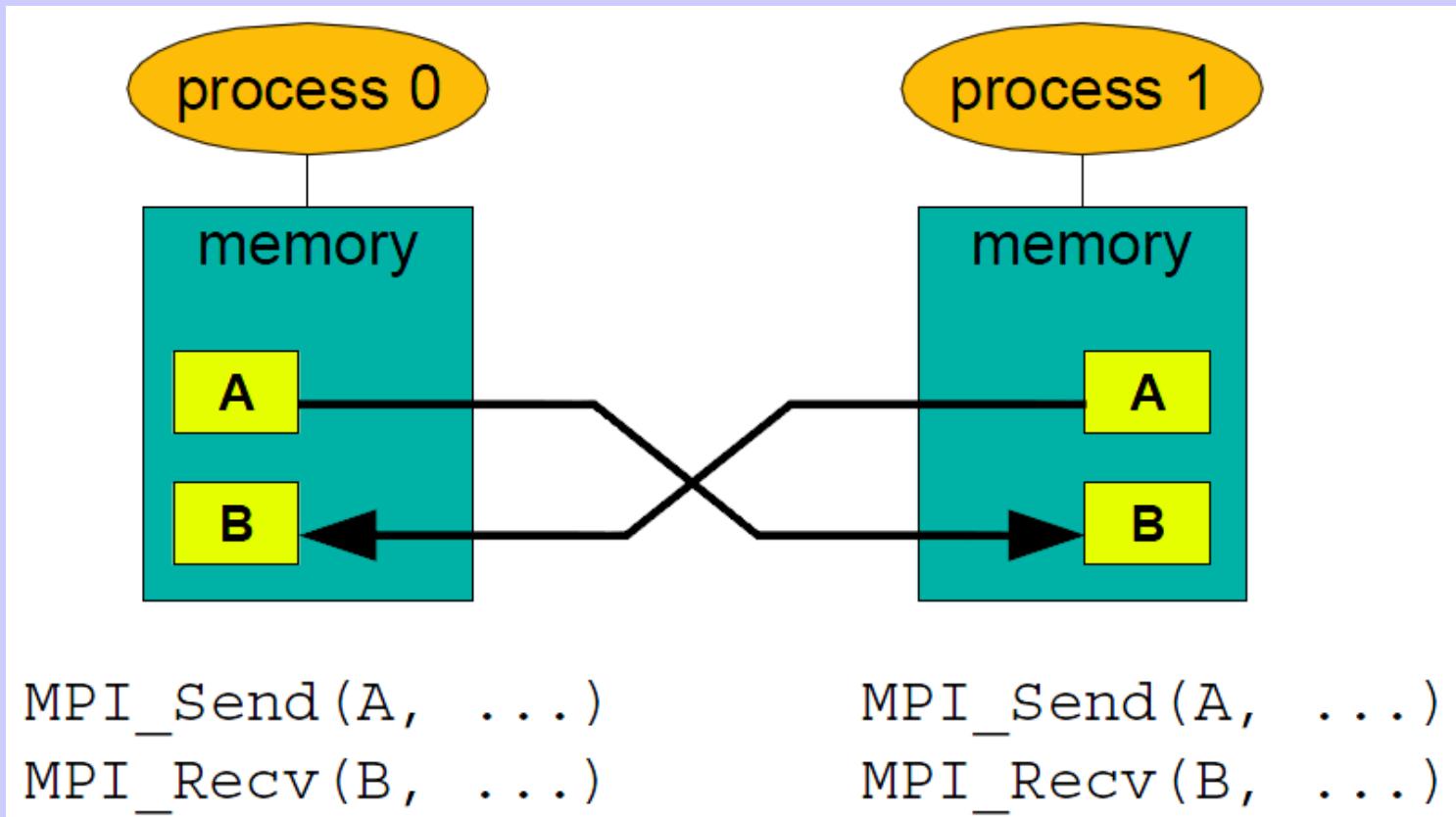
```
mpirun -np 4 myprog.exe mypar1 mypar2
```

# Punkt-zu-Punkt Kommunikation (P2P)

- Paarweise Einzeltransferoperation



# Explizite Kommunikation



# Senden

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - **buf** = Zeiger auf **Speicherbereich**, der gesendet werden soll (muss deklariert, reserviert und gefüllt sein!)
  - **count** = **Anzahl** der Daten, die gesendet werden sollen
  - **datatype** = **Typ** der zu sendenden Daten,  
(z.B. `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_BYTE`)
  - **dest** = Nummer des Zielprozesses (**rank**)
  - **tag** = Anwendungsspezifische **Kennzeichnung** der Nachricht

# Empfangen

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - **buf** = Zeiger auf den **Empfangspuffer** ausreichender Größe (muss deklariert und reserviert sein!)
  - **count** = obere Schranke für **Anzahl** der Daten
  - **datatype** = **Typ** der zu empfangenen Daten
  - **source** = Nummer (**rank**) des Senders (oder **MPI\_ANY\_SOURCE**)
  - **tag** = Erwartete **Kennzeichnung** der Nachricht (oder **MPI\_ANY\_TAG**)
  - **status** = Zeiger auf Struktur mit **Infos** zur empfangenen Nachricht

# Erfolgreicher Nachrichtenaustausch

- **Bedingungen:**
  - Gleicher Kommunikator
  - Sender passt oder **MPI\_ANY\_SOURCE**
  - Gleiches Tag oder **MPI\_ANY\_TAG**
  - Kompatible Datentypen
- **Senden und Empfangen** sind **blockierend** und **asynchron**:
  - **MPI\_Recv** kann ausgeführt werden, bevor das zugehörige **MPI\_Send** gestartet wurde
  - **MPI\_Recv** blockiert, bis die Nachricht vollständig empfangen wurde
  - **MPI\_Send** kann ausgeführt werden, bevor das zugehörige **MPI\_Recv** gestartet wurde
  - **MPI\_Send** blockiert, bis der Sendepuffer wiederverwendet werden kann (d.h. die Nachricht vollständig übermittelt oder zwischengepuffert wurde)

# Kommunikationsbeispiel für np=2

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

int main (int argc, char *argv[])
{
    int rank, p, tag = 4711;
    char msg[20];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) { // Master-Prozess
        strcpy(msg, "Hello!");
        MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    }
    if (rank == 1) { // Worker-Prozess
        MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
        printf("received \"%s\"\n", msg);
    }
    MPI_Finalize();
    return 0;
}
```

# Kommunikationsbeispiel für np=2

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

int main (int argc, char *argv[])
{
    int rank, p, tag = 4711;
    char msg[20];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) { // Master-Prozess
        strcpy(msg, "Hello!");
        MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    }
    if (rank == 1) { // Worker-Prozess
        MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
        printf("received \"%s\"\n", msg);
    }
    MPI_Finalize();
    return 0;
}
```

# Verklemmung für np>2

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

int main (int argc, char *argv[])
{
    int rank, p, tag = 4711;
    char msg[20];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) { // Master-Prozess
        strcpy(msg, "Hello!");
        MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    } else { // Worker-Prozess
        MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
        printf("received \"%s\"\n", msg);
    }
    MPI_Finalize();
    return 0;
}
```

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

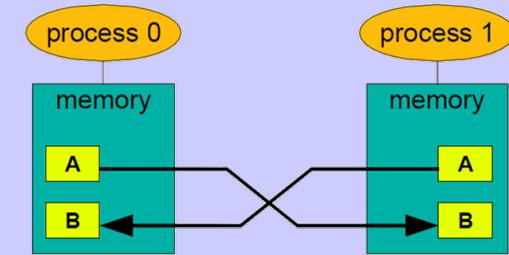
- **Programmierung verteilter Systeme mit MPI**
  - Senden und Empfangen
    - (nicht) blockierend
    - (a-)synchro
    - (nicht) lokale Kommunikation

# Problem blockierender Kommunikation

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD

// Prozess 0:
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status);

// Prozess 1:
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status);
```



## Problem:

- Das Senden ist keine lokale Operation. Sie kann blockieren, bis die passende Empfängeroperation gestartet wurde
- Beide Prozesse können beim Senden hängen bleiben, während sie auf den entsprechenden Empfänger warten (Verklemmung)

# 1. Lösungsvorschlag

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD

// Prozess 0:
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status);

// Prozess 1:
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status);
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
```

## Problem:

- Funktioniert nur in einfachen Fällen
- Verhindert Verklemmungen, aber serialisiert die Kommunikation
- Nutzt nicht die Möglichkeiten bi-direktionaler Hardware

# Beispiel: Sichere Empfangsreihenfolge

```
//...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {                                              // Master
    // Initialisieren
    MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD); // 1.
    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status); // 4.
    // Ausgabe
}
if (rank == 1) {                                              // Worker 1
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status); // 2.
    // Berechnen
    MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD); // 3.
}
```

# Sichere Empfangsreihenfolge für $np > 2$

```
//...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {                                              // Master
    // Initialisieren
    MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);      // 1.
    MPI_Send(sendbuf, count, MPI_INT, 2, tag, MPI_COMM_WORLD);
    // ...
    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status); // 4.
    MPI_Recv(recvbuf, count, MPI_INT, 2, tag, MPI_COMM_WORLD, &status);
    // Ausgabe
}
if (rank == 1) {                                              // Worker 1
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status); // 2a.
    // Berechnen
    MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);        // 3a.
}
if (rank == 2) {                                              // Worker 2
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status); // 2b.
    // Berechnen
    MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);        // 3b.
}
```

## 2. Lösungsvorschlag

- Kombiniertes Senden und Empfangen mit **zwei** unterschiedlichen oder **einem** gemeinsamen Puffer (z.B. für Weiterleiten in einem Ring)
- `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)`
- `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)`
- Das blockierende Senden und Empfangen erfolgt **gleichzeitig** in einem Aufruf
- Es soll Verklemmungen verhindern!?

# Gepuffertes Versenden von Daten

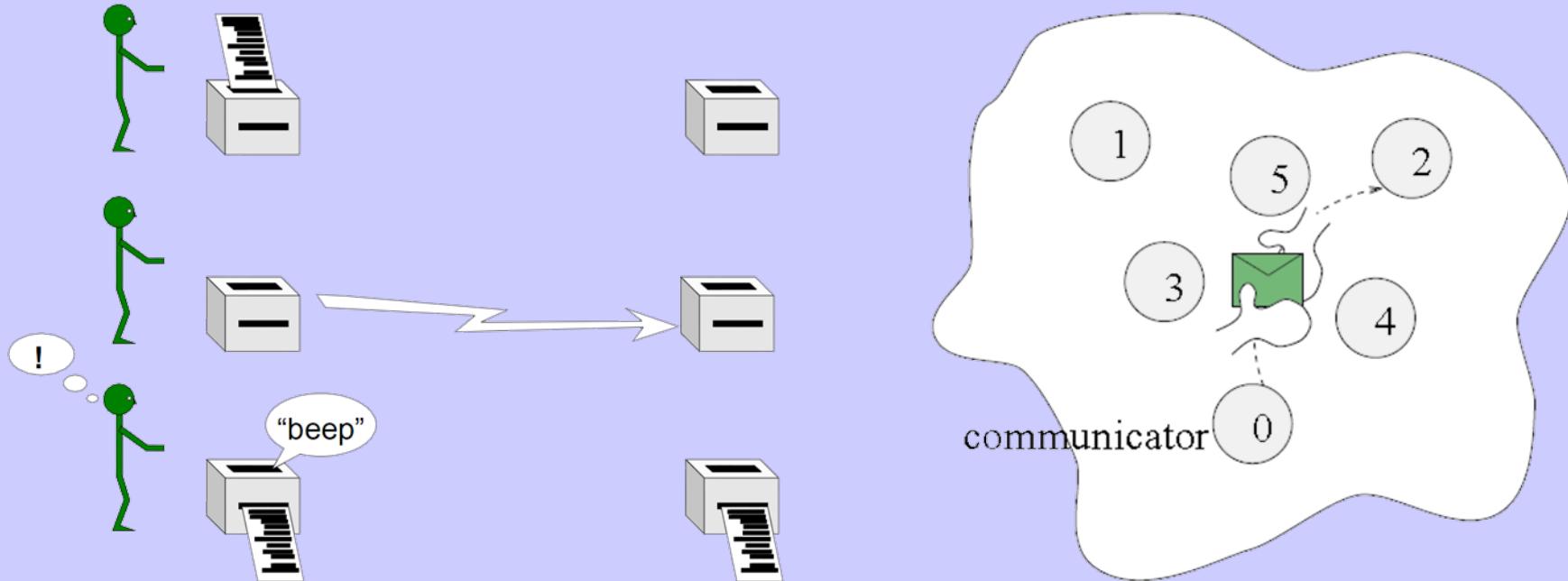
- int **MPI\_Bsend**(void \*sendbuf, int count, MPI\_Datatype type, int dest, int tag, MPI\_Comm comm)  
Startet ein Sende-Operation, bei der die Daten **zwischengepuffert** werden, um Blockierung zu vermeiden
  - int **MPI\_Buffer\_attach**(void \*buffer, int size)  
Macht MPI einen Speicherbereich als Puffer bekannt
  - int **MPI\_Buffer\_detach**(void \*buffer, int \*size)  
Gibt den Puffer wieder frei. Falls Nachricht enthalten, wird diese ausgeliefert.
  - Das gepufferte Senden muss **vor** der zugehörigen Empfangsoperation ausgeführt werden. Puffer bis zum Empfang nicht anderweitig verwenden.
  - Das Verwenden von Puffern verlangsamt Programme und verbraucht Speicher. Besser das **Kopieren** von Daten in Puffer vermeiden!

# Semantik von MPI

- **Blockierend**
  - ist eine MPI-Anweisung, falls die **Kontrolle** zum aufrufenden Prozess erst **zurückkehrt**, wenn alle Ressourcen, die für den Aufruf genutzt werden, wieder für andere Operationen zur Verfügung stehen. Die Steuerung kehrt erst zurück, **nachdem** die Nachricht gesendet und **empfangen wurde**
- **Nicht-blockierend**
  - ist eine MPI-Anweisung, falls die **Kontrolle** zum aufrufenden Prozess **zurückkehrt, bevor die** durch sie ausgelösten **Operationen** und Ressourcen **beendet sind** bzw. wieder benutzt werden dürfen
- **Synchrone Kommunikation**
  - Die Übertragung einer Nachricht findet nur statt, wenn **Sender und Empfänger gleichzeitig** an der Kommunikation teilnehmen
- **Asynchrone Kommunikation**
  - **Übertragung** findet statt, **ohne dass der Empfänger bereit ist**, die Nachricht zu empfangen. Der Sender übermittelt die Nachricht **einseitig**

# Synchrones Senden

- Information über die **vollständig erfolgreiche** Übermittlung (Handshake)



- `int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

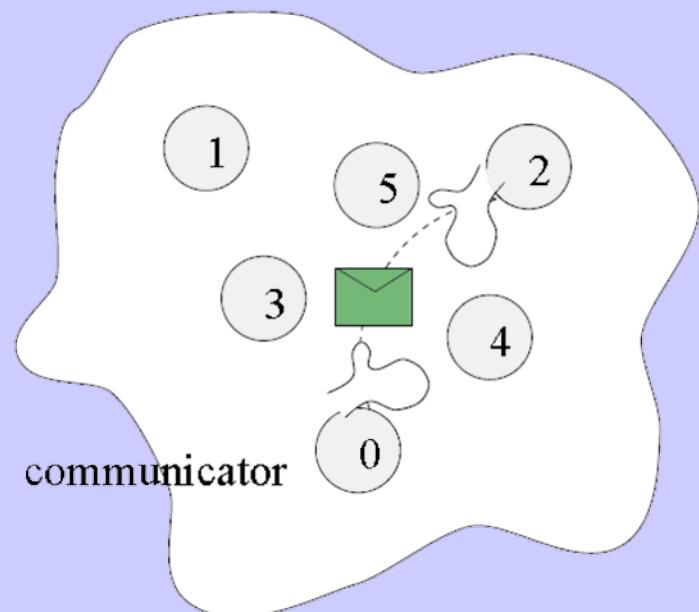
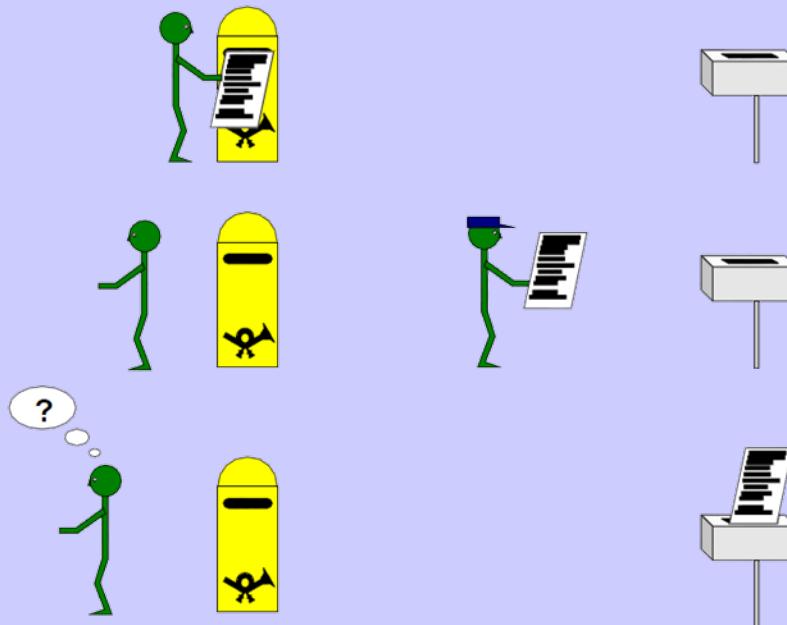
Sendet Daten **synchron** und **blockierend** an einen Prozess.

Gibt Kontrolle erst wieder zurück, wenn Daten wirklich empfangen wurden.

Falls nicht, tritt eine **Verklemmung** auf!

# Senden im Ready-mode

- Nur Information über das **erfolgreiche Absenden**.  
Der Prozess **hofft**, dass der andere die Nachricht erhält

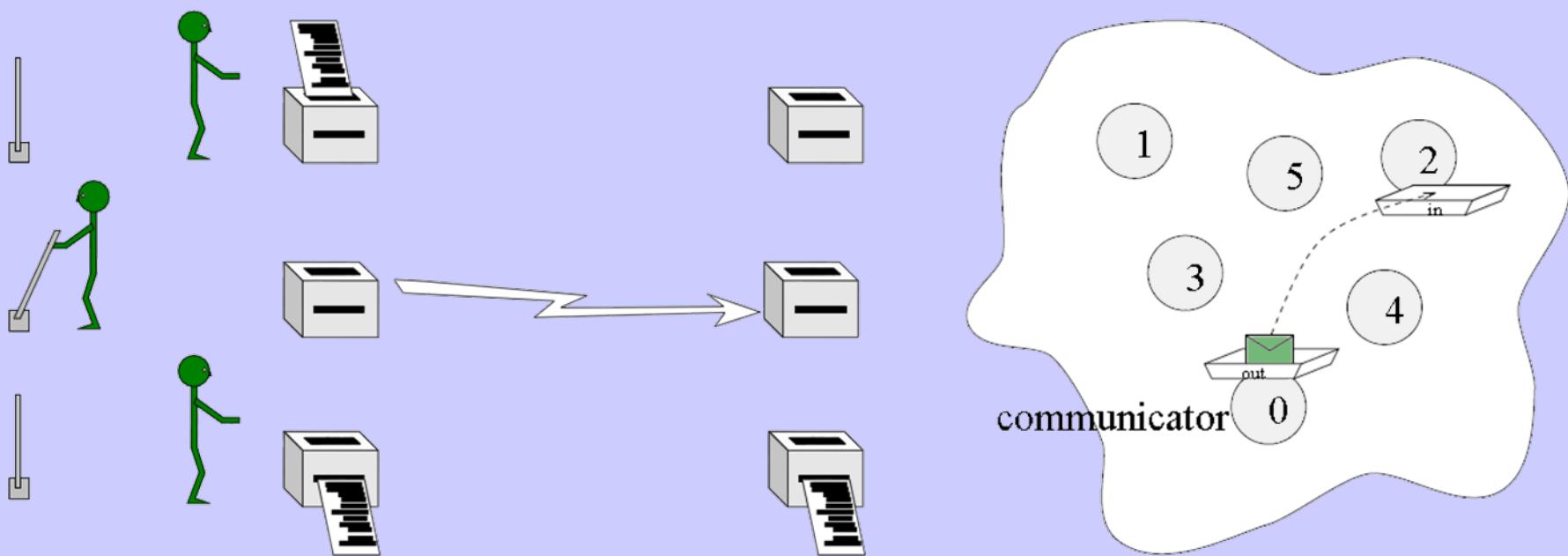


- ```
int MPI_Rsend(void *sendbuf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm)
```

  
Startet ein Sende-Operation im "**ready mode**", d.h. der Empfänger wartet schon auf Daten

# Nicht-blockierende Operationen

- Nach dem Senden können **weitere Arbeiten** erledigt werden.
- Später kann die erfolgreiche **Übermittlung geprüft** oder auf deren **Abschluss gewartet** werden



# Nicht-blockierende Eigenschaften

- Aufteilung der Kommunikation in drei Phasen:
  - **Leite** nicht-blockierende Kommunikation **ein**
  - Erledige andere Arbeiten (z.B. andere Kommunikation)
  - **Warte** auf den **Abschluss** der Kommunikation
- Ein **blockierendes** Senden kann auch mit einem **nicht-blockierenden** Empfangen kombiniert werden (und umgekehrt)
- Synchronisierung betrifft den **Abschluss** der Kommunikation und nicht deren **Einleitung**

# Nicht-blockierendes Senden

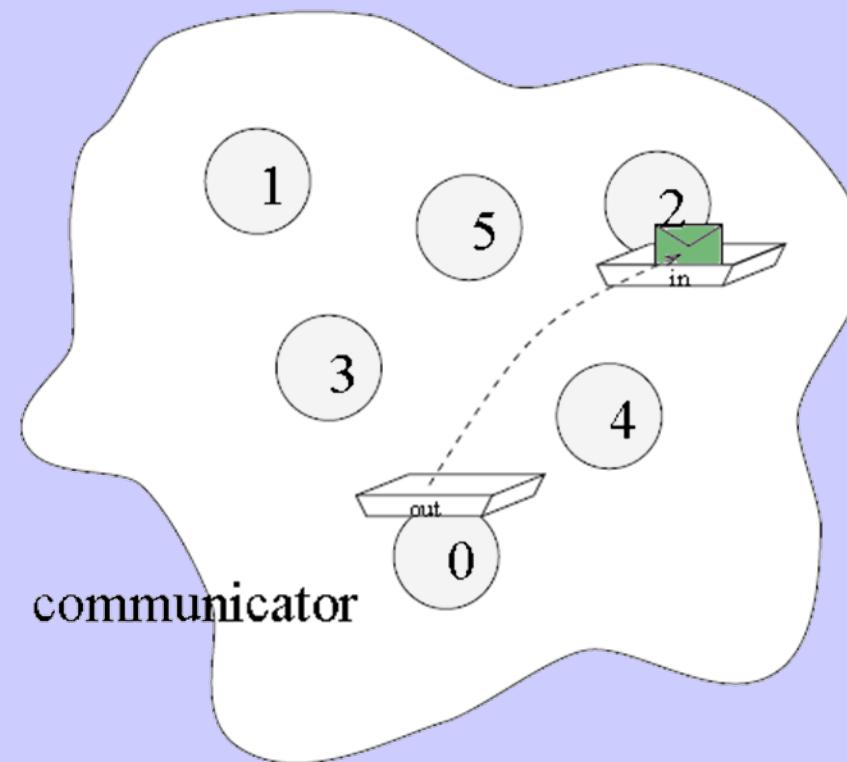
- `int MPI_ISEND(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Sendet Daten **nicht-blockierend** an einen Prozess, d.h. die Kontrolle wird sofort an das Programm zurückgegeben und der Auftrag irgendwann ausgeführt.  
Die spätere Identifikation läuft über das zurückgegebene **Handle** `req`
- `int MPI_IBSEND(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Startet ein Sende-Operation, bei der die Daten **zwischengepuffert** werden, um Blockierung zu vermeiden
- `int MPI_ISSEND(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Sendet Daten **synchron** und nicht-blockierend an einen Prozess.
- `int MPI_IRSEND(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Startet ein Sende-Operation im "**ready mode**", d.h. der Empfänger wartet schon auf Daten

**I** = immediate

# Nicht-blockierendes Empfangen

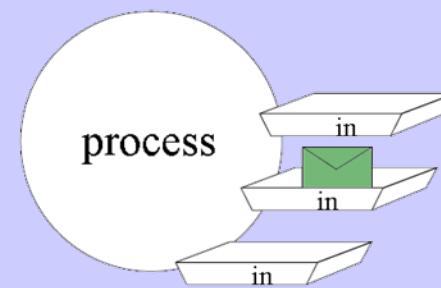
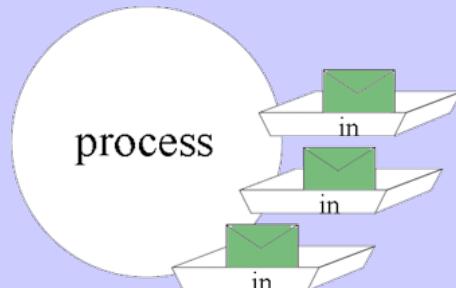
- `int MPI_Irecv(void *recvbuf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *req)`

Führt eine **nicht-blockierende** Empfangsoperation aus



# Erfolgreiche Übertragung testen

- `int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)`  
Testet ein Handle auf **Beendigung** der entsprechenden Operation (true, false)
- `int MPI_Testall(int count, MPI_Request req[], int *flag, MPI_Status status[])`  
Testet eine **Liste** von Handles, ob **alle** Kommunikationen beendet wurden
- `int MPI_Testany(int count, MPI_Request req[], int *index, int *flag, MPI_Status *status)`  
Testet eine Liste von Handles und gibt einen **Index** (oder `MPI_UNDEFINED`) sowie den **Beendigungsstatus** zurück
- `int MPI_Testsome(int count, MPI_Request req[], int *outcount, int indices[], MPI_Status status[])`  
Testet eine Liste von Handles und gibt die **Beendeten** als Liste zurück



# Erfolgreiche Übertragung abwarten

- `int MPI_Wait(MPI_Request *req, MPI_Status *status)`  
Wartet auf die Beendigung einer Operation, die durch das Handle repräsentiert wird
- `int MPI_Waitall(int count, MPI_Request req[], MPI_Status status[])`  
Wartet auf die Beendigung **aller** Operationen, die von den übergebenen Handles repräsentiert werden
- `int MPI_Waitany(int count, MPI_Request req[], int *index, MPI_Status *status)`  
Wartet auf die Beendigung **einer** der Operationen, die von allen übergebenen Handles repräsentiert werden
- `int MPI_Waitsome(int count, MPI_Request req[], int *outcount, int indices[], MPI_Status status[])`  
Wartet auf die Beendigung **mindestens einer** der Operationen, die von allen übergebenen Handles repräsentiert werden

# Nicht-blockierende Kommunikation 1

```
MPI_Request request;

MPI_Recv(buf, count, type, dest, tag, comm, status);
// =
MPI_Irecv(buf, count, type, dest, tag, comm, &request);
// +
MPI_Wait(&request, &status);

MPI_Send(buf, count, type, dest, tag, comm);
// =
MPI_Isend(buf, count, type, dest, tag, comm, &request);
// +
MPI_Wait(&request, &status);
```

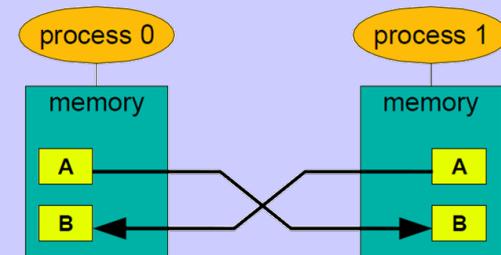
# Nicht-blockierende Kommunikation 2

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;

// Prozess 0:
MPI_Isend(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request);
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status);
MPI_Wait(&request, &status);

// Prozess 1:
MPI_Isend(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request);
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status);
MPI_Wait(&request, &status);
```

- Keine Verklemmung!
- **status** enthält beim Warten keine nützliche Info mehr!



# Status

- Die Variable vom Typ **MPI\_Status** ist eine Struktur
- Sie enthält Informationen über die gesendete Nachricht für den Empfänger
- Die Komponenten sind

```
typedef struct {  
    int MPI_SOURCE; // spezifiziert Sender der empfangenen Nachricht  
    int MPI_TAG;   // gibt die Markierung der empfangenen Nachricht an  
    int MPI_ERROR; // enthält den Fehlercode  
} MPI_Status
```

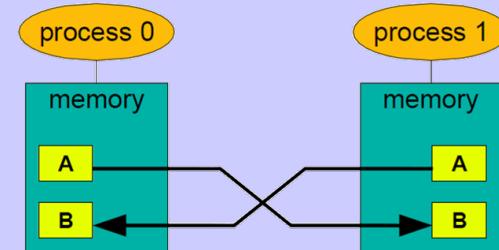
# Nicht-blockierende Kommunikation 3

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;

// Prozess 0:
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request);
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Wait(&request, &status);

// Prozess 1:
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request);
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
MPI_Wait(&request, &status);
```

- Besser `Irecv` anstelle von `Isend` verwenden ...
- Keine Verklemmung!



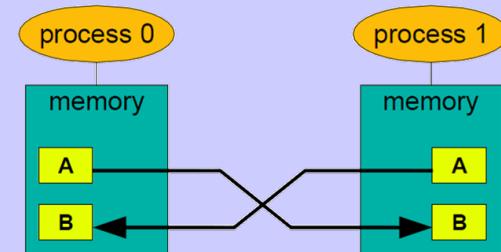
# Nicht-blockierende Kommunikation 4

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request[2];
MPI_Status status[2];

// Prozess 0:
MPI_Isend(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request[0]);
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request[1]);
// Erledige etwas Sinnvolles
MPI_Waitall(2, &request, &status);

// Prozess 1:
MPI_Isend(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request[0]);
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request[1]);
// Erledige etwas Sinnvolles
MPI_Waitall(2, &request, &status);
```

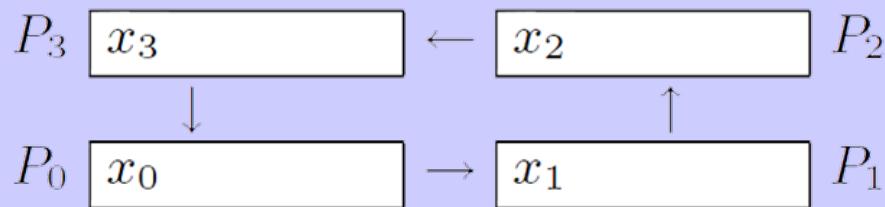
- Für **echte überlappende Kommunikation** ist Hardwareunterstützung notwendig!



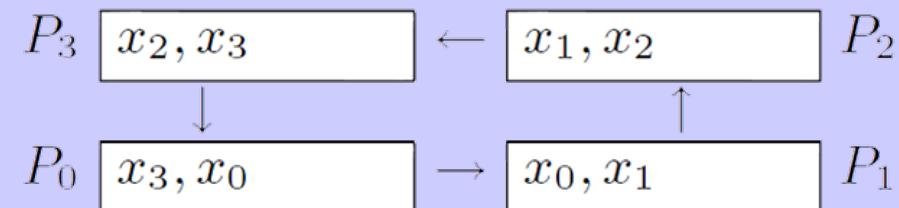
# Beispiel: Sammelring

- **Aufsammeln** von lokal berechneten Teildaten in einer logischer **Ringstruktur**

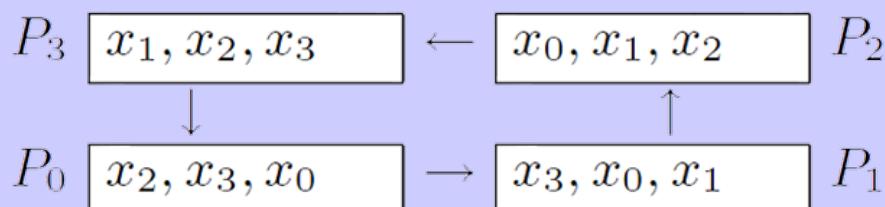
Schritt 1



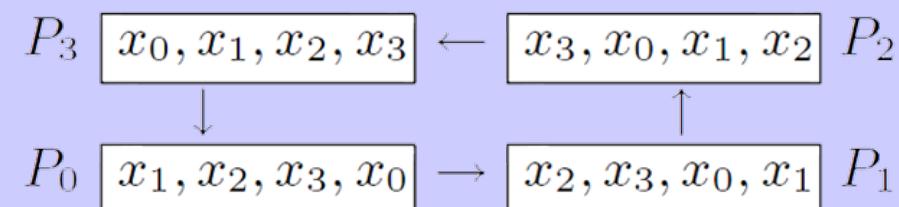
Schritt 2



Schritt 3

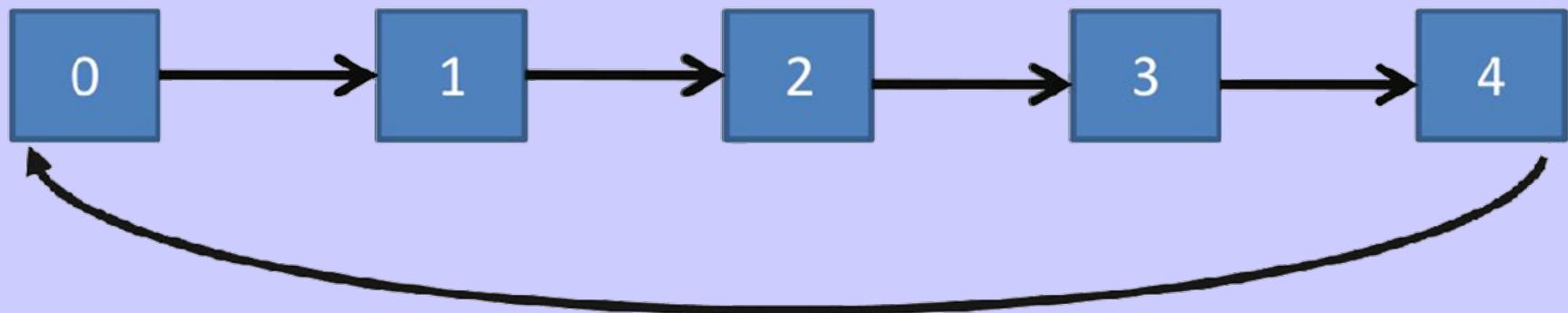


Schritt 4



# Ringkommunikation

- Prozess  $P_i$  empfängt von  $P_{i-1}$  und sendet an  $P_{i+1}$



```
succ = (rank+1) % p;      // Nachfolger  
pred = (rank-1+p) % p;    // Vorgänger
```

# Beispiel 1 (blockierend)

```
void Sammelring(float x[], int size, float y[])
{
    int i, p, rank, succ, pred;
    int send_offset, recv_offset;
    MPI_Status status;

    MPI_Comm_size(MPI_COMM_WORLD, &p);                                // Anzahl der Prozesse
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);                               // Eigene Prozess-ID

    for (i=0; i<size; i++)
        y[i+rank * size] = x[i];                                     // Lokale Daten x in großen Speicher y kopieren
    succ = (rank+1) % p;  // Prozess-ID des Nachfolgers
    pred = (rank-1+p) % p;  // Prozess-ID des Vorgängers

    for (i=0; i<p-1; i++) {
        send_offset = ((rank-i+p) % p) * size;
        recv_offset = ((rank-i-1+p) % p) * size;
        MPI_Send(y+send_offset, size, MPI_FLOAT, succ, 0, MPI_COMM_WORLD);
        MPI_Recv(y+recv_offset, size, MPI_FLOAT, pred, 0, MPI_COMM_WORLD, &status);
    }
}
```

# Beispiel 2 (nicht-blockierend)

```
void Sammelring(float x[], int size, float y[])
{
    int i, p, rank, succ, pred;
    int send_offset, recv_offset;
    MPI_Status status;
    MPI_Request send_request, recv_request;

    MPI_Comm_size (MPI_COMM_WORLD, &p);                                // Anzahl der Prozesse
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);                                // Eigene Prozess-ID

    for (i=0; i<size; i++)
        y[i+rank * size] = x[i];   // Lokale Daten x in globalen Speicher y kopieren
    succ = (rank+1) % p;   // Prozess-ID des Nachfolgers
    pred = (rank-1+p) % p;   // Prozess-ID des Vorgängers

    for (i=0; i<p-1; i++) {
        send_offset = ((rank-i+p) % p) * size;
        recv_offset = ((rank-i-1+p) % p) * size;
        MPI_Isend(y+send_offset, size, MPI_FLOAT, succ, 0, MPI_COMM_WORLD, &send_request);
        MPI_Irecv(y+recv_offset, size, MPI_FLOAT, pred, 0, MPI_COMM_WORLD, &recv_request);
        MPI_Wait(&send_request, &status);
        MPI_Wait(&recv_request, &status);
    }
}
```

# Beispiel 3 (nicht-blockierend)

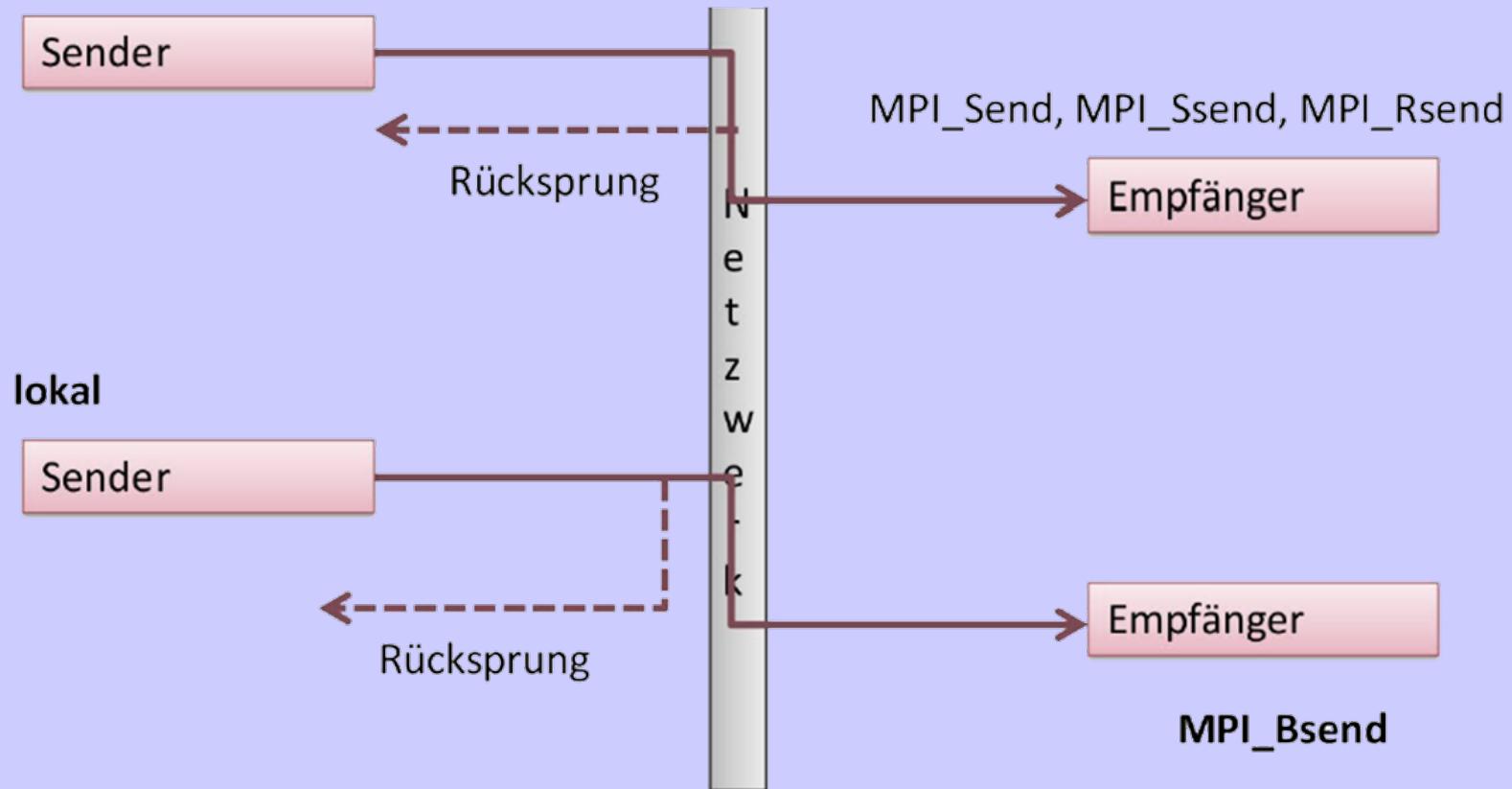
```
void Sammelring(float x[], int block, float y[])
{
    int i, p, rank, succ, pred;
    int send_offset, recv_offset;
    MPI_Status status;
    MPI_Request send_request, recv_request;

    MPI_Comm_size (MPI_COMM_WORLD, &p);                                // Anzahl der Prozesse
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);                                // Eigene Prozess-ID

    for (i=0; i<size; i++)
        y[i+rank * size] = x[i];           // Lokale Daten x in globalen Speicher y kopieren
    succ = (rank+1) % p;                      // Prozess-ID des Nachfolgers
    pred = (rank-1+p) % p;                    // Prozess-ID des Vorgängers

    send_offset = rank * size;
    recv_offset = ((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_request);
        MPI_Irecv(y+recv_offset, size, MPI_FLOAT, pred, 0, MPI_COMM_WORLD, &recv_request);
        send_offset = ((rank-i-1+p) % p) * size;      // Erledige etwas fast Sinnvolles :-
        recv_offset = ((rank-i-2+p) % p) * size;
        MPI_Wait(&send_request, &status);
        MPI_Wait(&recv_request, &status);
    }
}
```

# (Nicht) Lokale Kommunikation



- **Lokal:** Rücksprung unabhängig von Kontakt mit Zielfunktion
- **Nicht lokal:** Rücksprung erst, wenn Nachricht empfangen wurde.  
(Rücksprung hängt vom Zustand des Empfängers ab)

# Handles für blockierendes Senden 1

- Das **Vorbereiten** und **Versenden** von Daten kann man auch bei den blockierenden Funktionen trennen:
- `int MPI_Send_init(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Liefert ein Handle auf eine **blockierende** Sende-Operation.
- `int MPI_Bsend_init(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Liefert ein Handle auf eine **blockierende** Bsend-Operation zurück.
- `int MPI_Rsend_init(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Liefert ein Handle auf eine **blockierende** Rsend-Operation zurück.
- `int MPI_Ssend_init(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`  
Liefert ein Handle auf eine **synchrone blockierende** Sende-Operation.

# Handles für blockierendes Senden 2

- `int MPI_Recv_init(void *recvbuf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *req)`  
Liefert ein Handle auf eine blockierende Empfangsoperation.
- `int MPI_Start(MPI_Request *req)`  
Startet eine Operation, deren Handle übergeben wird.
- `int MPI_Startall(int count, MPI_Request *req)`  
Startet alle Operationen, deren Handle übergeben werden.
- `int MPI_Cancel(MPI_Request *req)`  
Bricht eine Operation ab.
- `int MPI_Test_cancelled(MPI_Status *status, int *flag)`  
Testet, ob eine Operation abgebrochen wurde.
- `int MPI_Request_free(MPI_Request *req)`  
Zerstört ein Handle und gibt die von ihm belegten Ressourcen wieder frei.

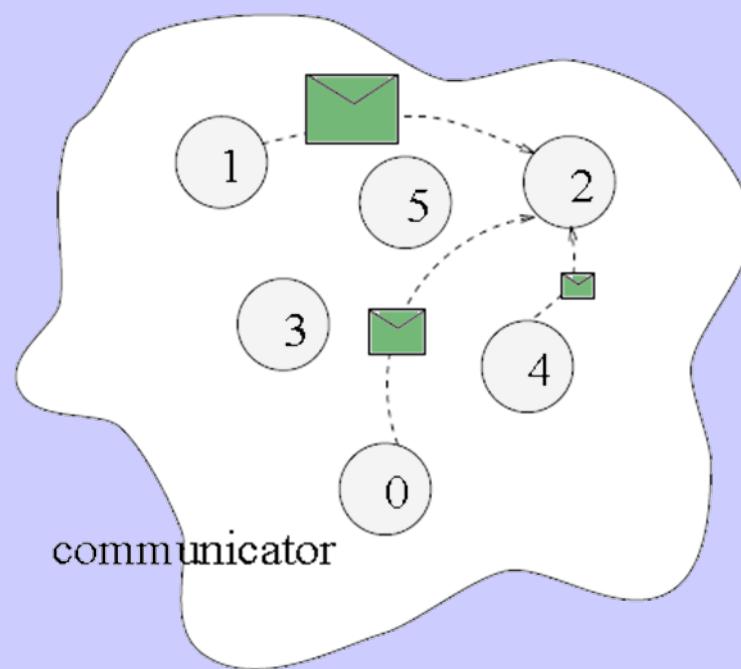
# Beispiel: Blockierende Handles

```
//...  
  
MPI_Request send_request, recv_request;    // zwei Handles  
MPI_Status status;  
int num = 4711;  
  
MPI_Send_init(&num, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &send_request);  
MPI_Recv_init(&num, 1, MPI_INT, from, 0, MPI_COMM_WORLD, &recv_request);  
  
while (num > 0)  
{  
    MPI_Start(&recv_request);  
    MPI_Wait(&recv_request, &status);  
    // ...  
    MPI_Start(&send_request);  
    MPI_Wait(&send_request, &status);  
}  
MPI_Request_free(&send_request);    // Handles löschen  
MPI_Request_free(&recv_request);
```

# Nachrichtengröße

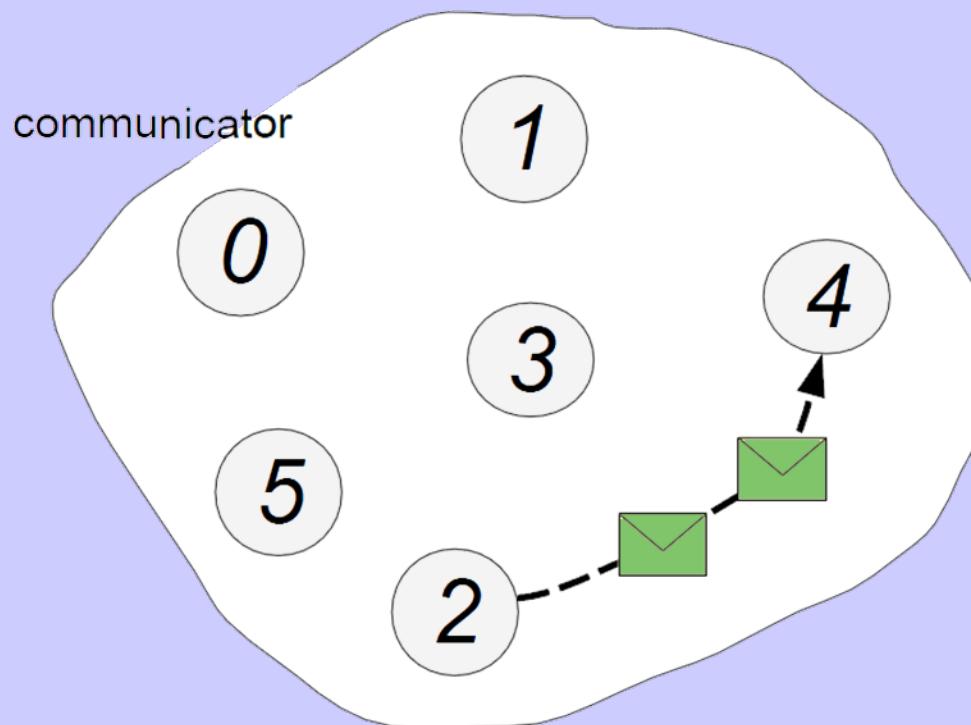
- Es können Nachrichten verschiedener Größe empfangen werden.
- **Tatsächliche Größe** der erhaltenen Nachricht

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



# Reihenfolge von Nachrichten

- Nachrichten bei (nicht-)synchronisiertem Senden können sich nicht überholen!



# Unsichere Empfangsreihenfolge

```
//...
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
    MPI_Send(sendbuf1, count, MPI_INT, 2, tag, comm);           // z.B. 4711
    MPI_Send(sendbuf2, count, MPI_INT, 1, tag, comm);           // z.B. 0815
}
if (rank == 1) {
    MPI_Recv(recvbuf1, count, MPI_INT, 0, tag, comm, &status);
    MPI_Send(recvbuf1, count, MPI_INT, 2, tag, comm);
}
if (rank == 2) {  // Was wird jeweils empfangen?
    MPI_Recv(recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
    MPI_Recv(recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
}
//...
```

# Empfangstests

- Schaue nach, ob es eine Nachricht gibt, die einem bestimmten Muster entspricht
- Die Nachricht selber wird nicht empfangen, es wird die Statusvariable gesetzt
- **Blockierender Empfangstest:**

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Wartet, bis eine Nachricht verfügbar ist, die mit einem `MPI_Recv` empfangen würde (**nicht-lokaler** Aufruf)

- **Nicht blockierender Empfangstest:**

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```

Testet, ob eine Nachricht verfügbar ist, die mit einem `MPI_Recv` empfangen würde (**lokaler** Aufruf)

# MPI Programm terminieren

- **Abbrechen von MPI**

```
int MPI_Abort(MPI_Comm comm, int exitcode);
```

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- **Verteilte Systeme mit MPI**
  - Kollektive Kommunikation
    - Broadcast
    - Scatter, Gather
    - Allgather, Alltoall
    - Reduktions-Operatoren
  - Abgeleitete Datentypen
  - Kommunikatoren und Gruppen



# **Verteilen und Sammeln**

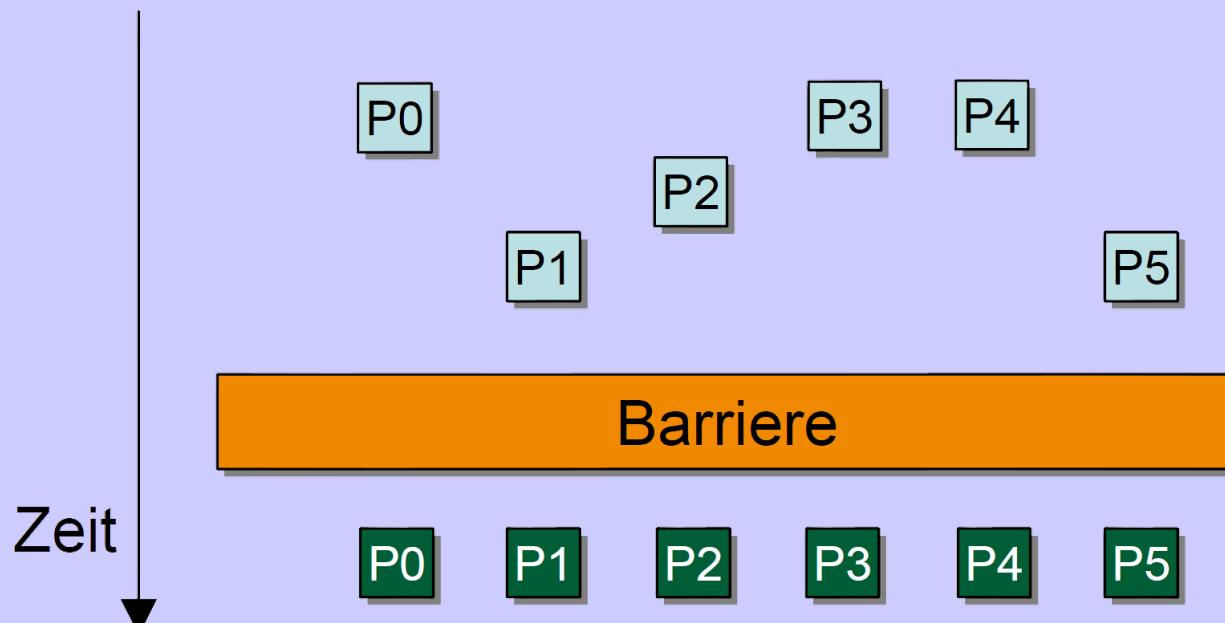
Kollektive Kommunikation

# Barrieren

- Synchronisierung von Prozessen

```
int MPI_Barrier(MPI_Comm comm);
```

- Kein Prozess verlässt die Barriere, bevor nicht alle anderen Prozesse die Barriere betreten haben.
- Das heißt, dass nicht alle Prozesse die Barriere notwendigerweise zum **gleichen Zeitpunkt** verlassen.



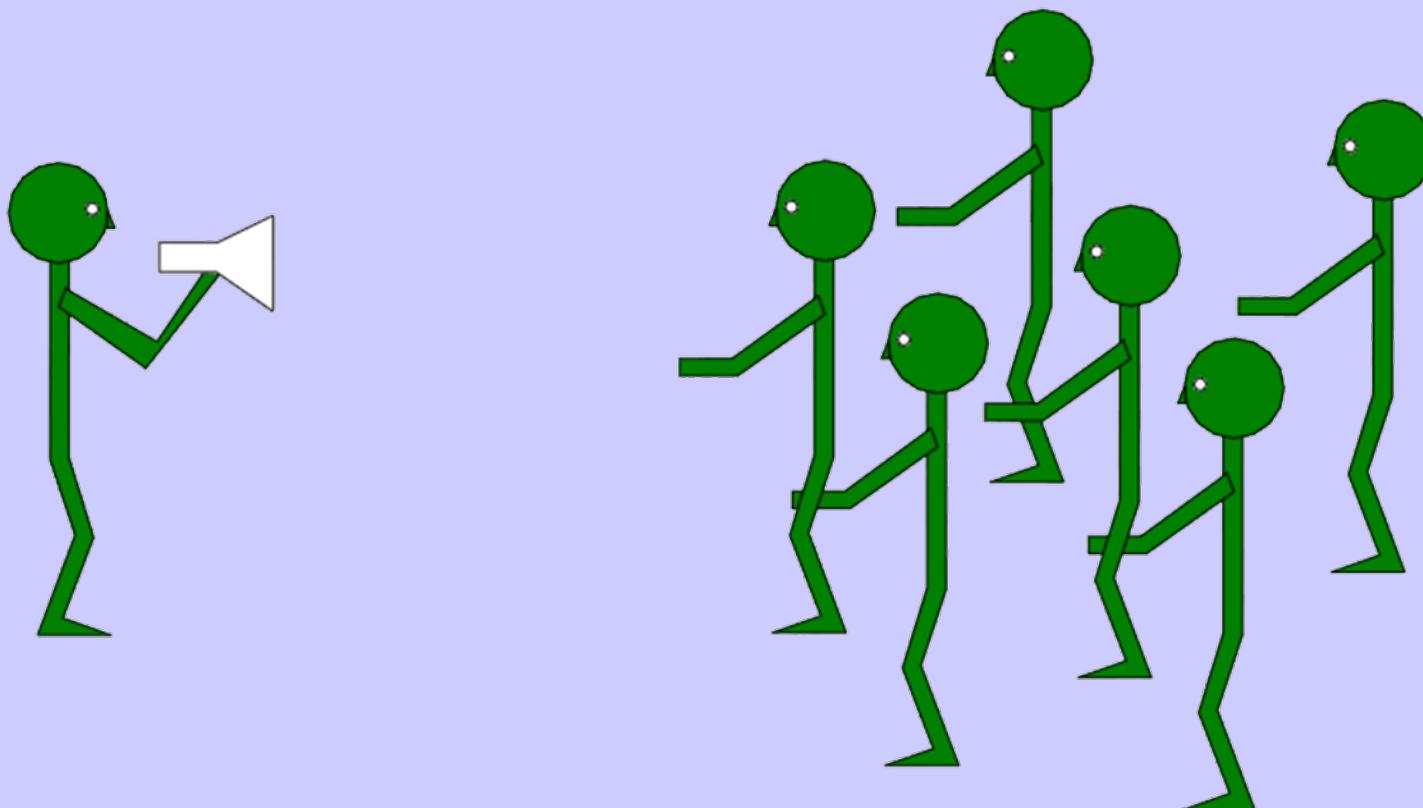
# Beispiel: Ablaufsynchronisation

- Alle Tasks warten gegenseitig auf das Erreichen der Barriere.

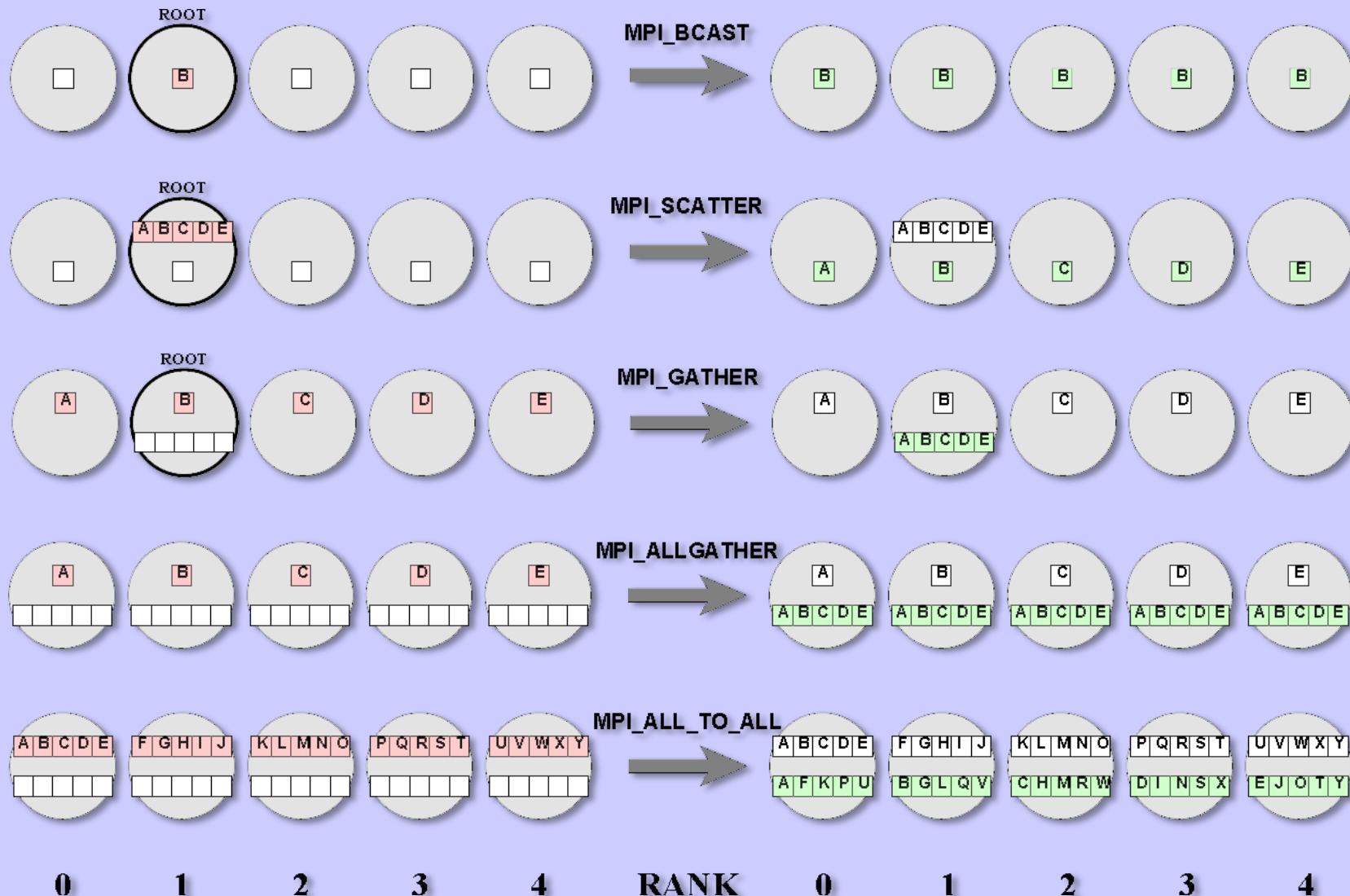
```
If (rank == 0) {  
    MPI_Isend(...);  
    MPI_Barrier(MPI_COMM_WORLD); // Task 0  
} else {  
    MPI_Barrier(MPI_COMM_WORLD); // Task 1...n  
    // ...  
}
```

- `MPI_Isend` ist nicht beendet. Auf Daten darf nicht zugegriffen werden.

# 1-zu-N Kommunikation



# Übersicht: Verteilen von Daten

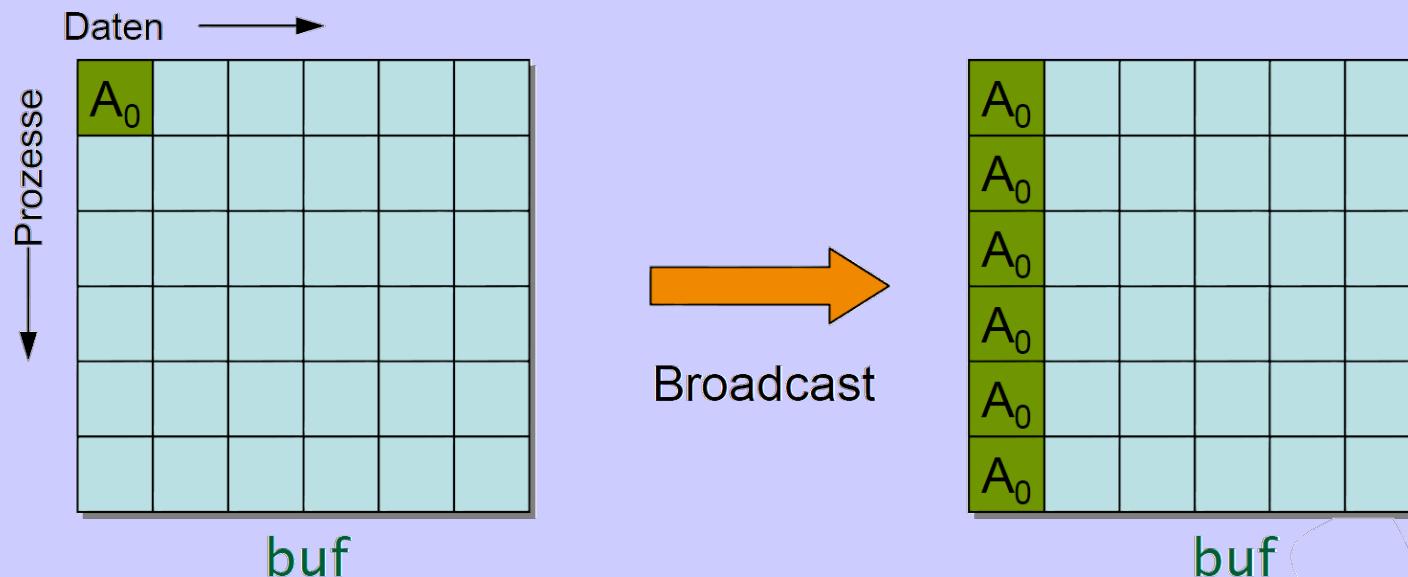


# 1. Broadcast

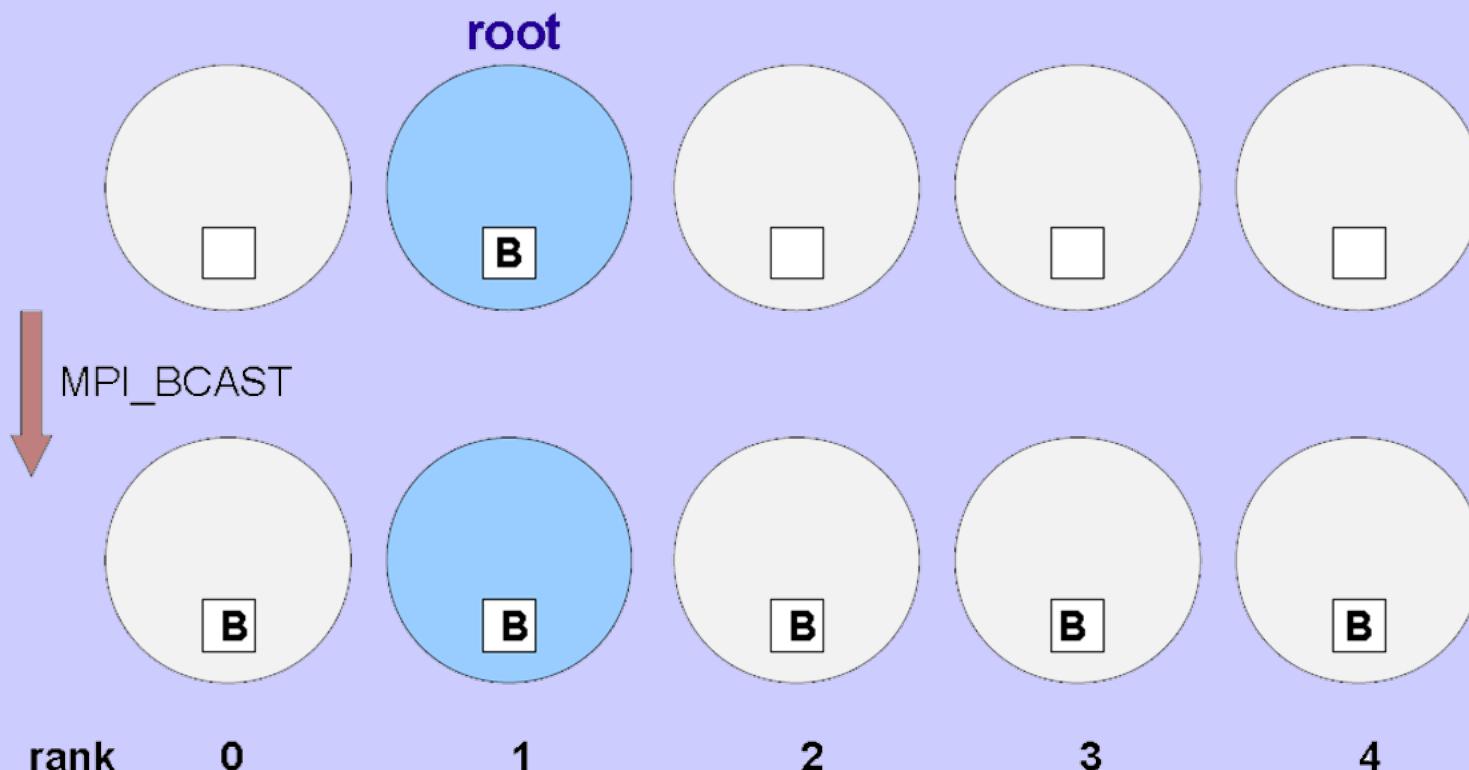
- Verteilen der Daten von einem Prozess zu vielen anderen Prozessen

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

- Alle Prozesse müssen diese Funktion aufrufen. Beim Besitzer (`root`) zeigt `buf` auf die Daten, die versendet werden sollen. Bei allen anderen Prozessen zeigt `buf` auf den Speicherbereich, wo die Daten eingetragen werden sollen.



# Beispiel: MPI\_Bcast

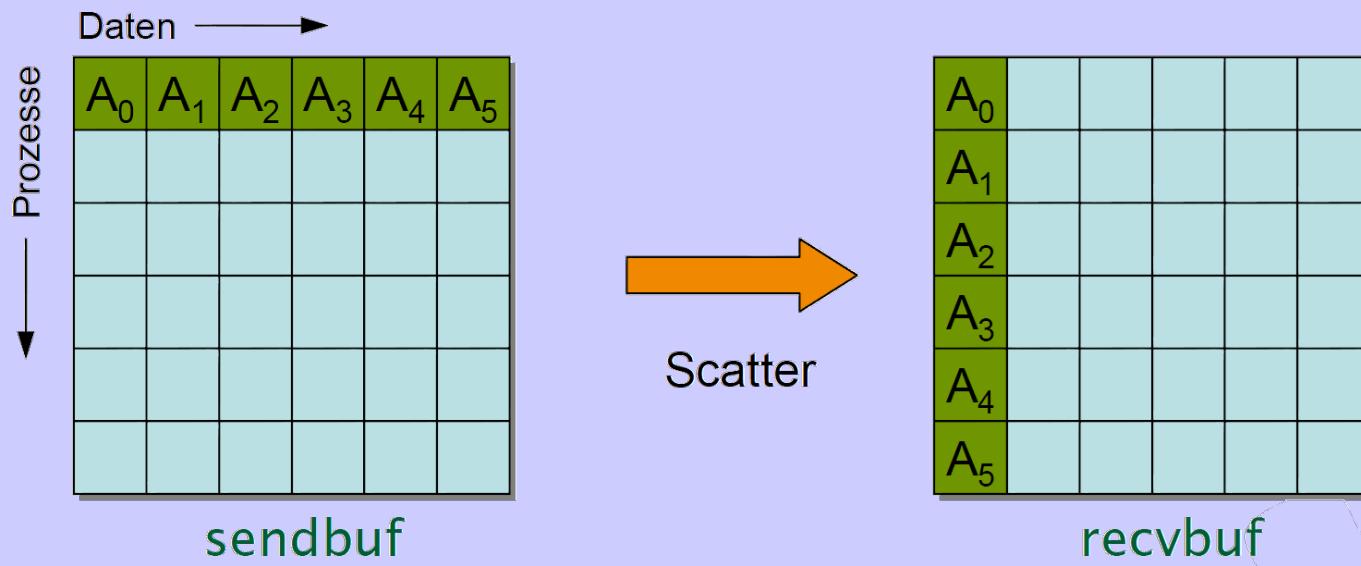


## 2. Scatter

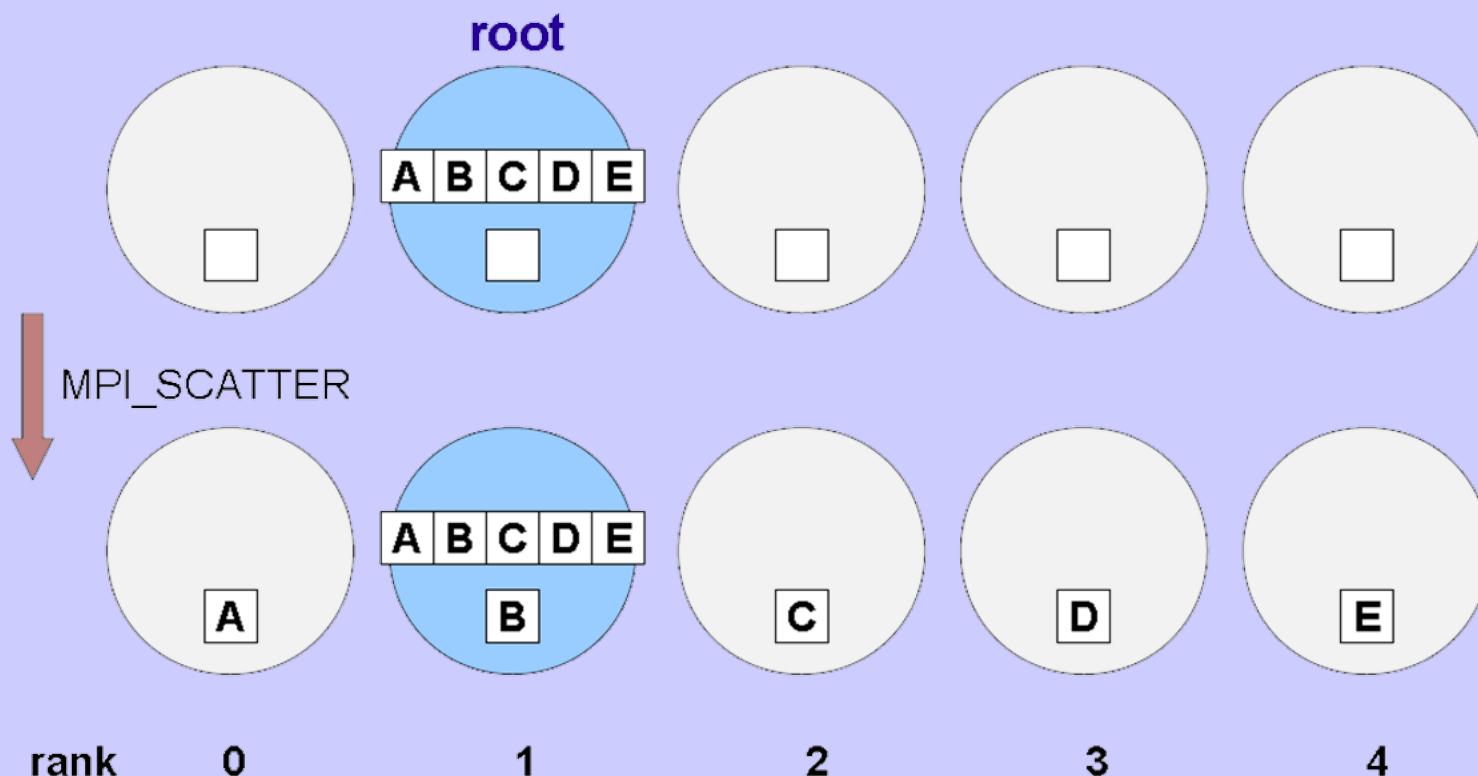
- **Aufteilen** von Elementen aus einem Datenpuffer an verschiedene Prozesse

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

- **Alle** Prozesse müssen diese Funktion aufrufen.



# Beispiel: MPI\_Scatter

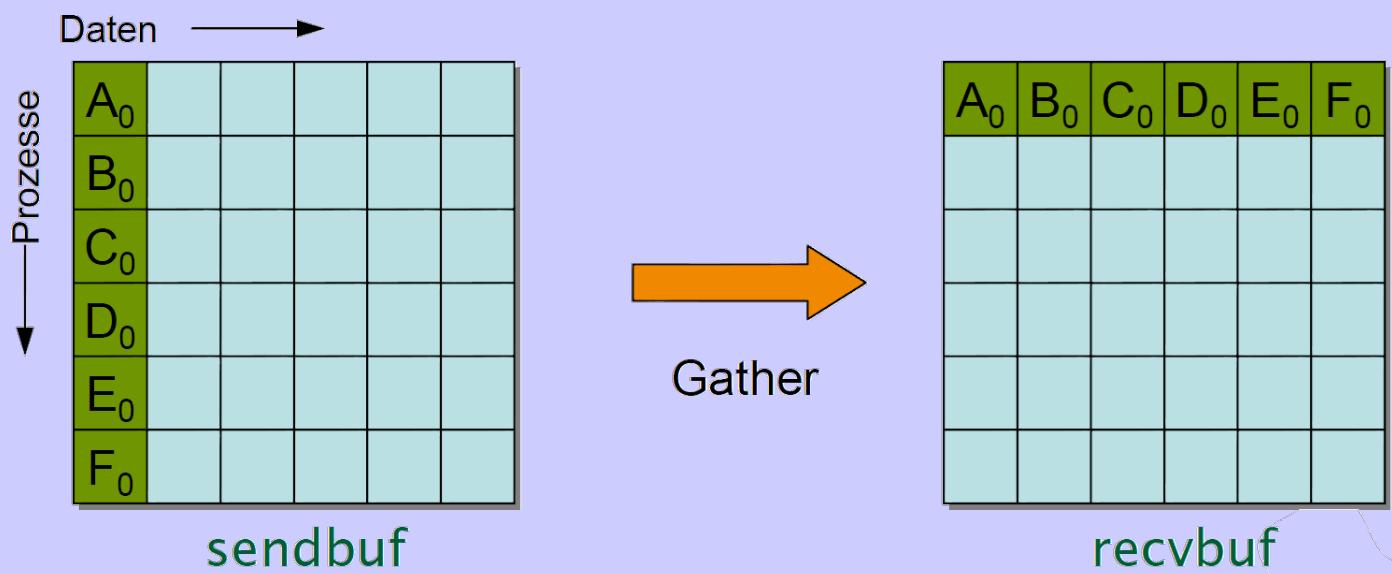


## 3. Gather

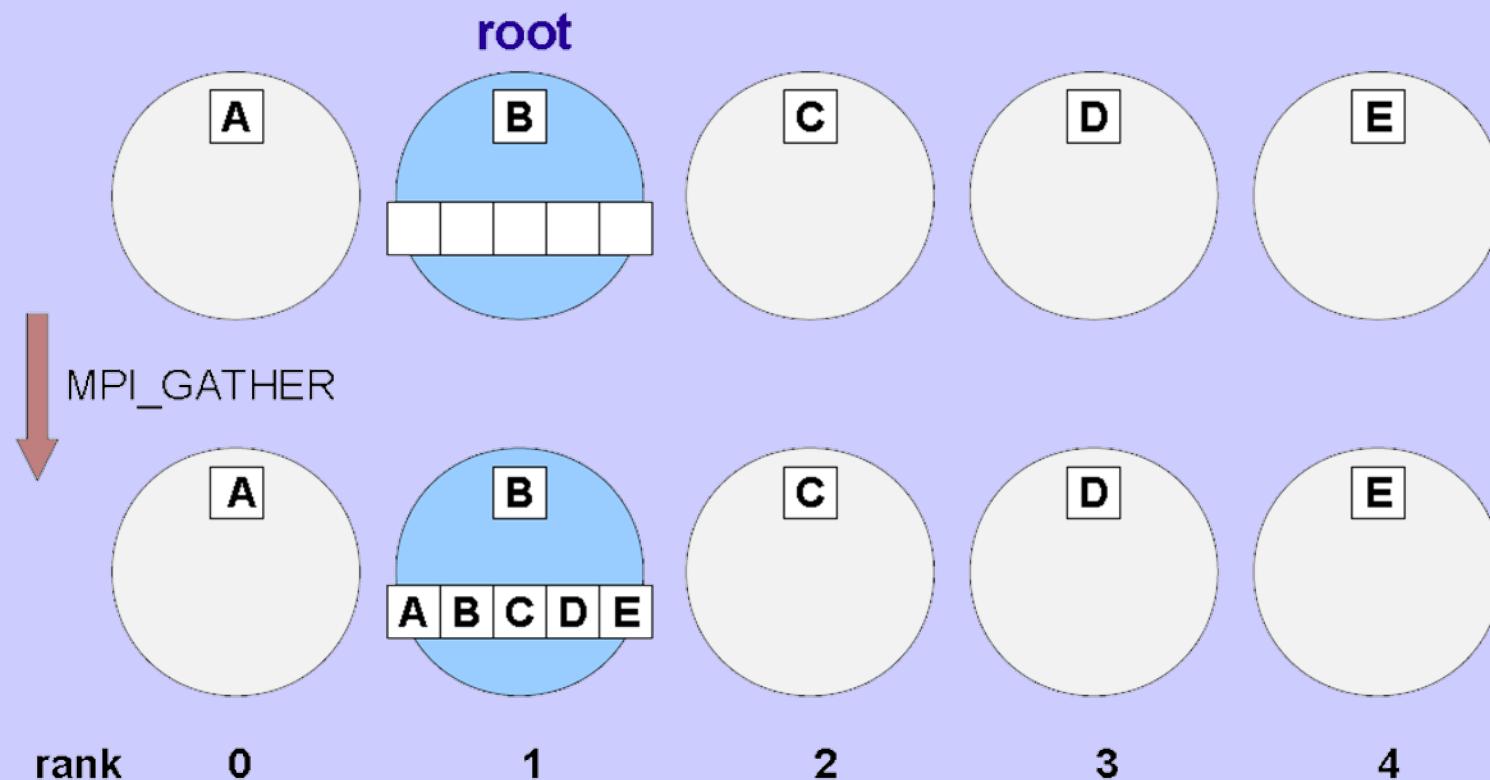
- **Aufsammeln** der Elemente eines Datenpuffers von verschiedenen Prozessen

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

- **Alle** Prozesse müssen diese Funktion aufrufen.



# Beispiel: MPI\_Gather

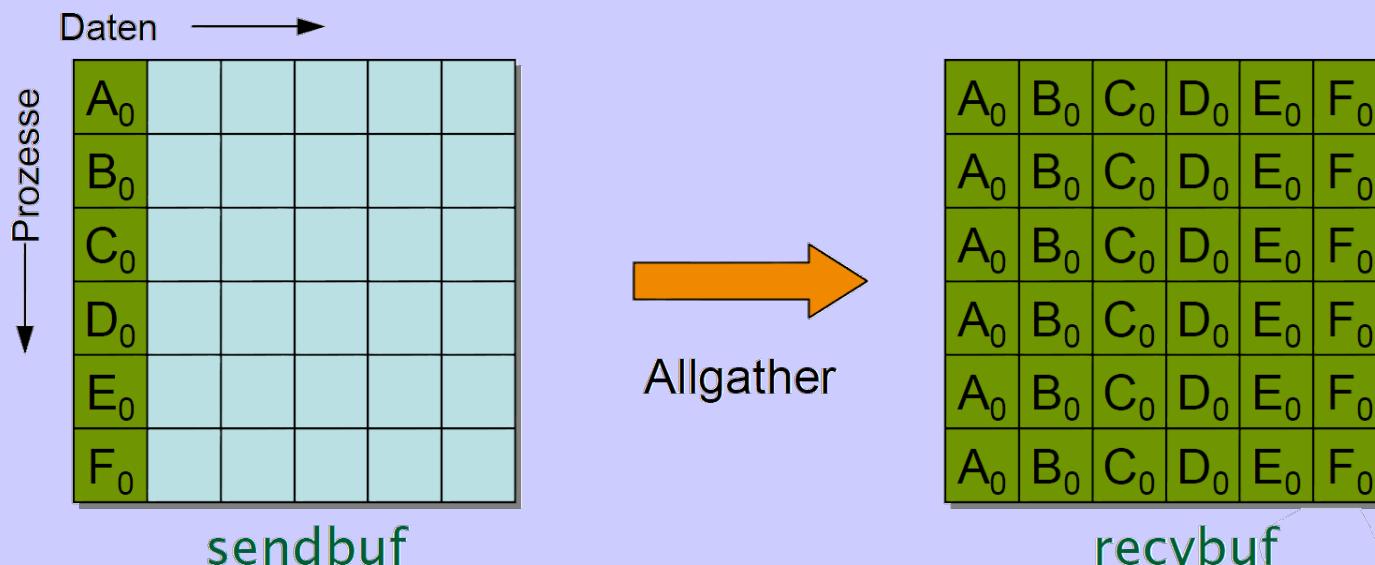


# 4. Allgather

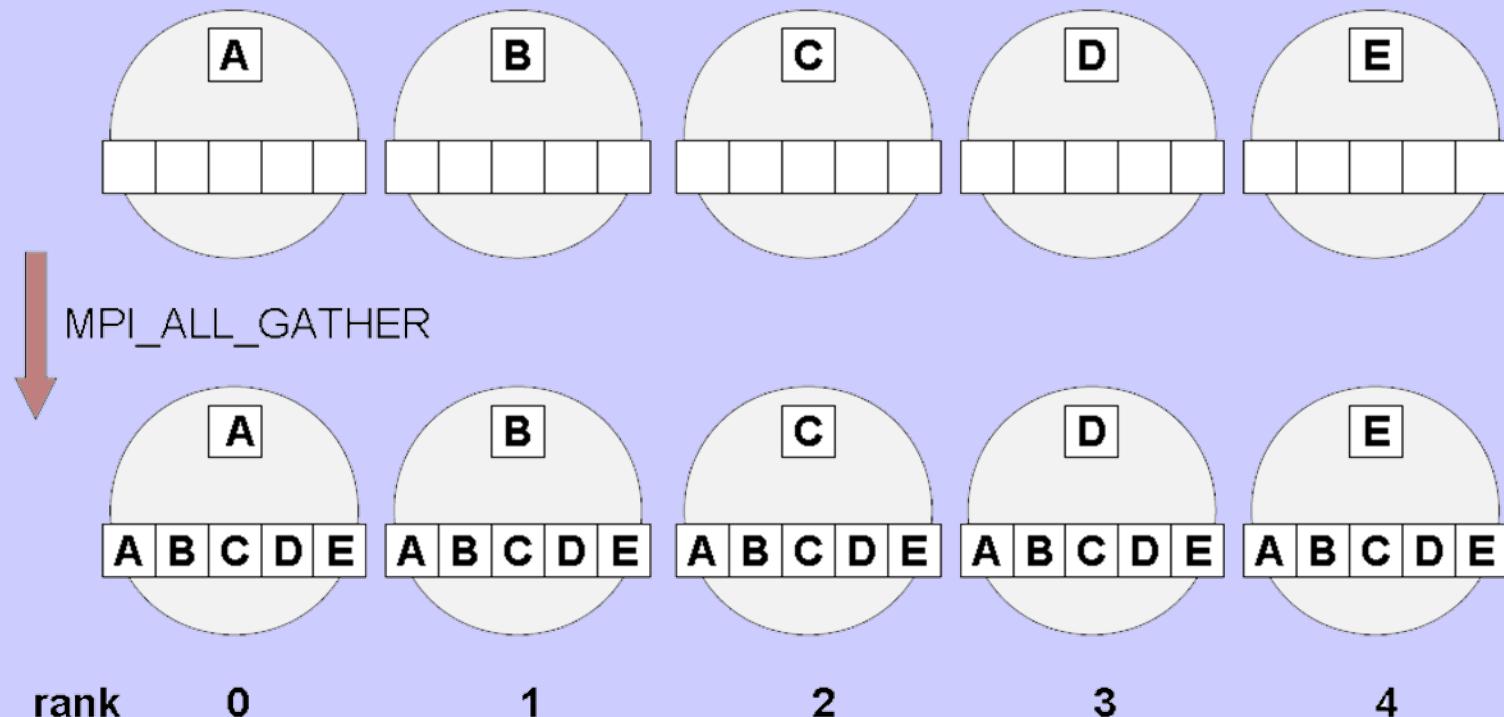
- Wie MPI\_Gather, aber die Elemente von **allen** Prozessen werden bei **jedem** Prozess gesammelt (Allgather = Gather + Bcast)

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

- Alle** Prozesse müssen diese Funktion aufrufen.



# Beispiel: MPI\_Allgather

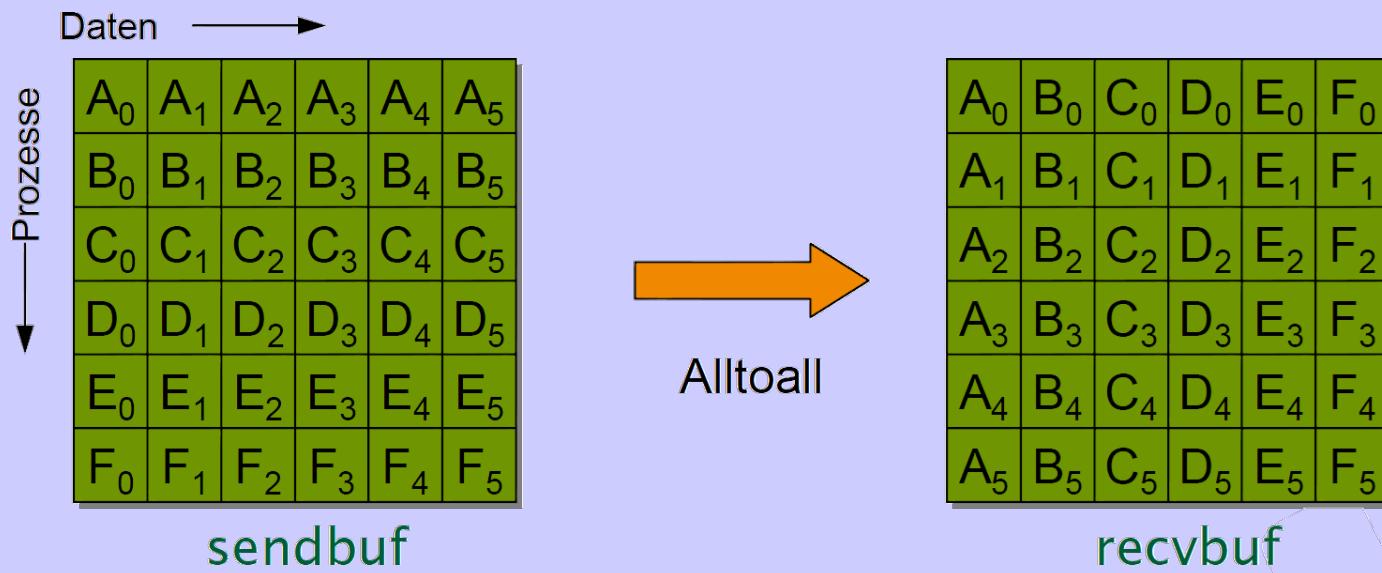


# 5. Alltoall

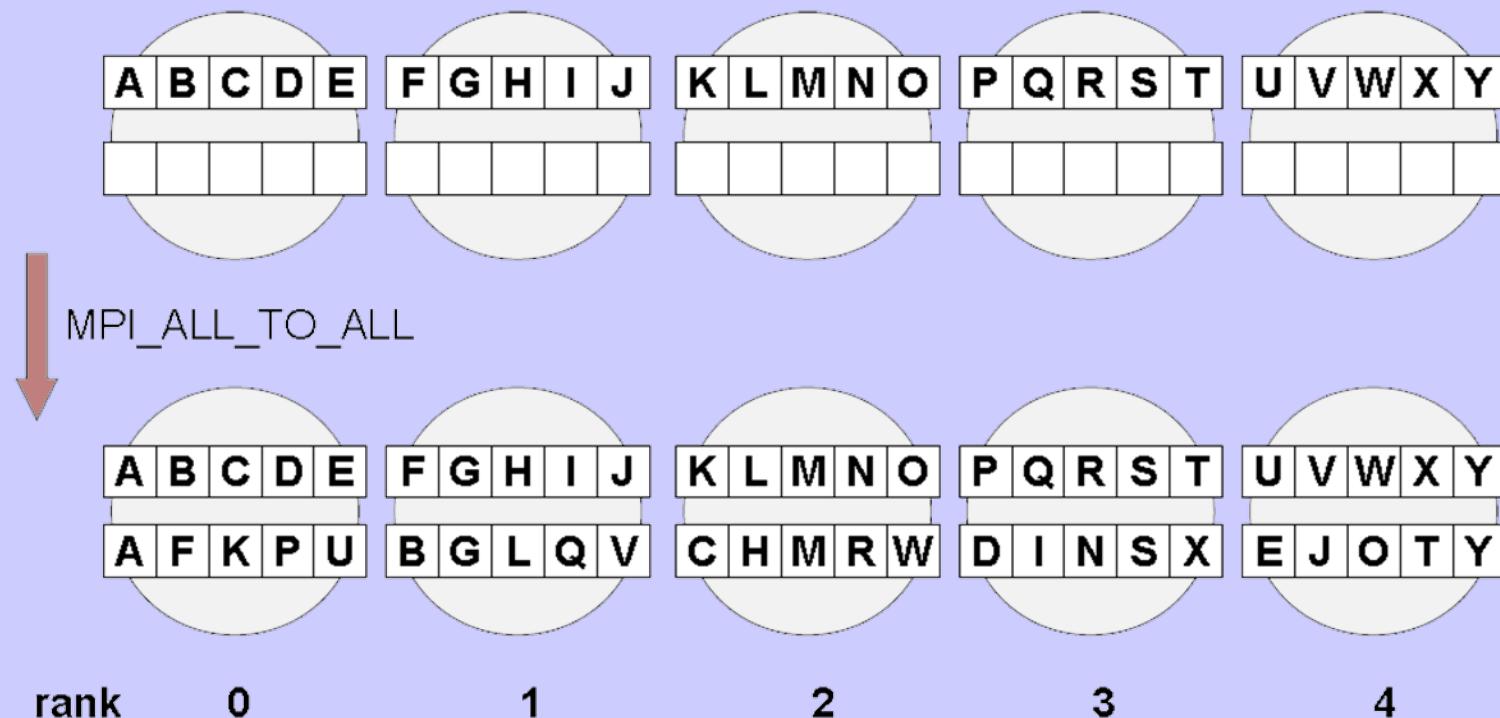
- Alle erhalten von allen anderen jeweils ein bestimmtes Element

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

- Alle Prozesse müssen diese Funktion aufrufen. Im Gegensatz zu MPI\_Allgather werden die Daten nicht zusammengefasst.



# Beispiel: MPI\_Alltoall



# Beispiel: Verteilen einer Matrix

```
#include <mpi.h>
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[])
{
    int num, rank, root = 1;
    float recvbuf[SIZE], sendbuf[SIZE][SIZE] = {
        {1.0, 2.0, 3.0, 4.0},           // Initialisierung eigentlich
        {5.0, 6.0, 7.0, 8.0},           // nur für root notwendig!
        {9.0, 10.0, 11.0, 12.0},
        {13.0, 14.0, 15.0, 16.0}
    };

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num);

    // Matrix verteilen, jeder Prozess bekommt eine Zeile
    MPI_Scatter(sendbuf, SIZE, MPI_FLOAT, recvbuf, SIZE,
                MPI_FLOAT, root, MPI_COMM_WORLD);

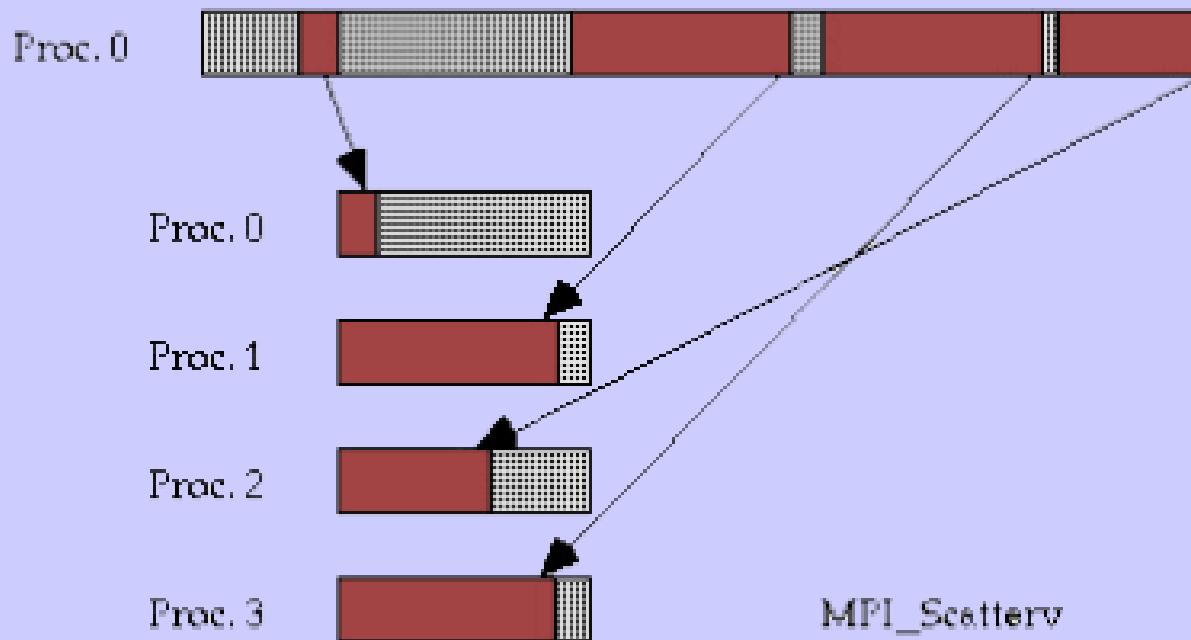
    printf("rank = %d, result: %f %f %f %f\n", rank,
           recvbuf[0], recvbuf[1], recvbuf[2], recvbuf[3]);
    MPI_Finalize();
    return 0;
}
```

## Ausgabe:

```
rank = 1, result: 5.0 6.0 7.0 8.0
rank = 2, result: 9.0 10.0 11.0 12.0
rank = 3, result: 13.0 14.0 15.0 16.0
rank = 0, result: 1.0 2.0 3.0 4.0
```

# Variable Austauschoperationen

- **MPI\_Scatter** und **MPI\_Gather** besitzen auch **variable** Versionen.
- **Variabel** sind
  - **Anzahl** der Daten, welche an die einzelnen Prozessoren verteilt wird
  - deren **Position** im Sendepuffer

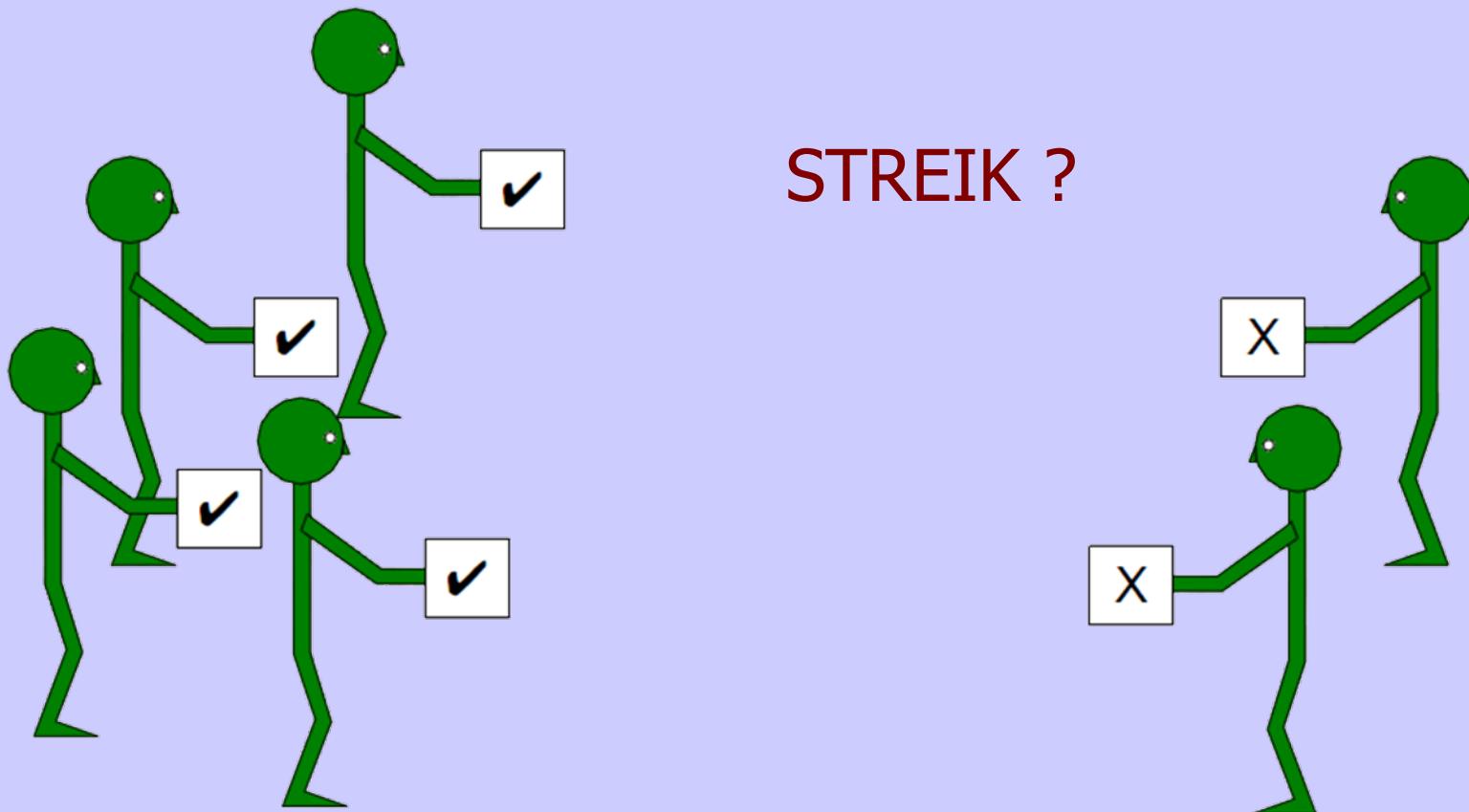


# Variable Austauschoperationen

- `int MPI_Scatterv(void *sbuf, int *scounts, int *displs, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)`
  - `scounts[i]` enthält die **Anzahl** der an Prozess i zu versendenden Datenelemente
  - `displs[i]` legt den **Beginn** des Datenblocks für Prozess i **relativ** zu `sbuf` fest
- `int MPI_Gatherv(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype, int root, MPI_Comm comm)`
- Es existieren auch die Funktionen
  - `MPI_Allgatherv`
  - `MPI_Alltoallv`

# Reduktions-Operatoren

- Kombinieren von Teilergebnissen zu einem einzelnen Endergebnis



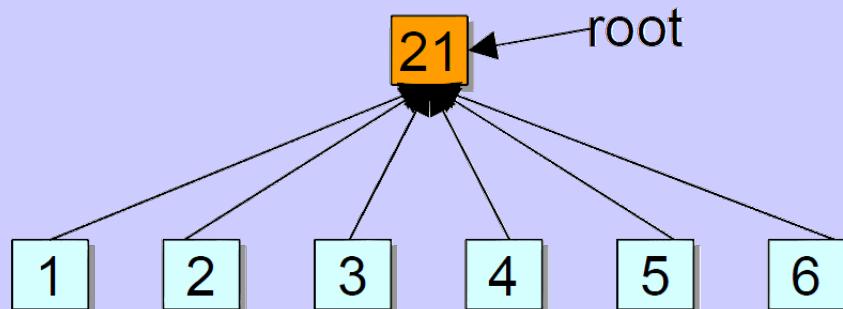
# 1. Reduktion

- Globale Operationen auf verteilten Daten.  
**Sammelt** Daten von allen Prozessen ein und **verrechnet** sie

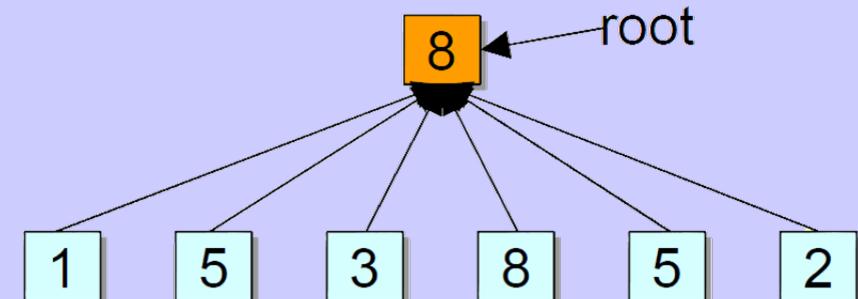
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,  
MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm);
```

- **Alle** Prozesse müssen diese Funktion aufrufen.  
Die Funktion **op** muss den Typ **type** verarbeiten können.

op = MPI\_SUM



op = MPI\_MAX



# Operatoren

- **Vordefinierte Operatoren**

- Maximum, Minimum, Summe, Produkt

`MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD`

- Boolesche Operatoren (logische / bitweise Verknüpfung)

`MPI_BAND, MPI_BAND`

`MPI_LOR, MPI_BOR`

`MPI_LXOR, MPI_BXOR`

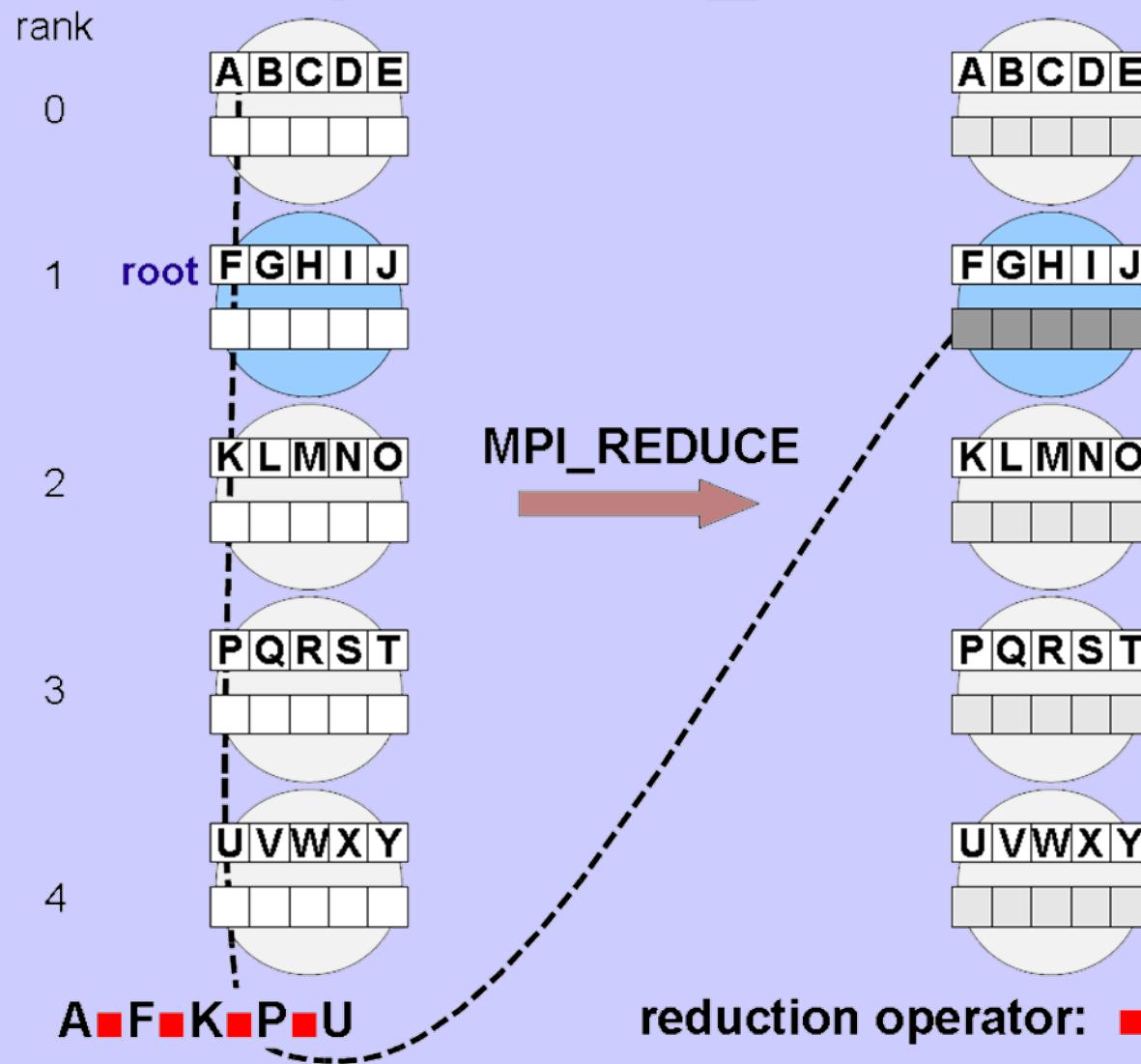
- Maximum oder Minimum inklusive der **Position**

`MPI_MAXLOC, MPI_MINLOC`

(als Datentyp wird ein **Paar** <Wert, Rang> verwendet)

- **Auch selbstdefinierte (assoziative) Operatoren möglich**

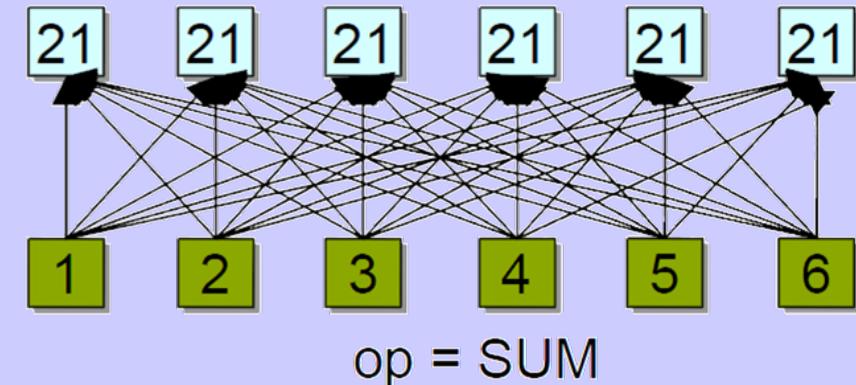
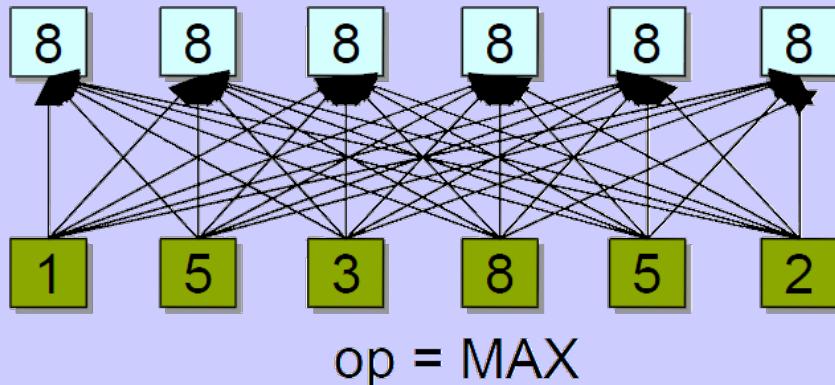
# Beispiel: MPI\_Reduce



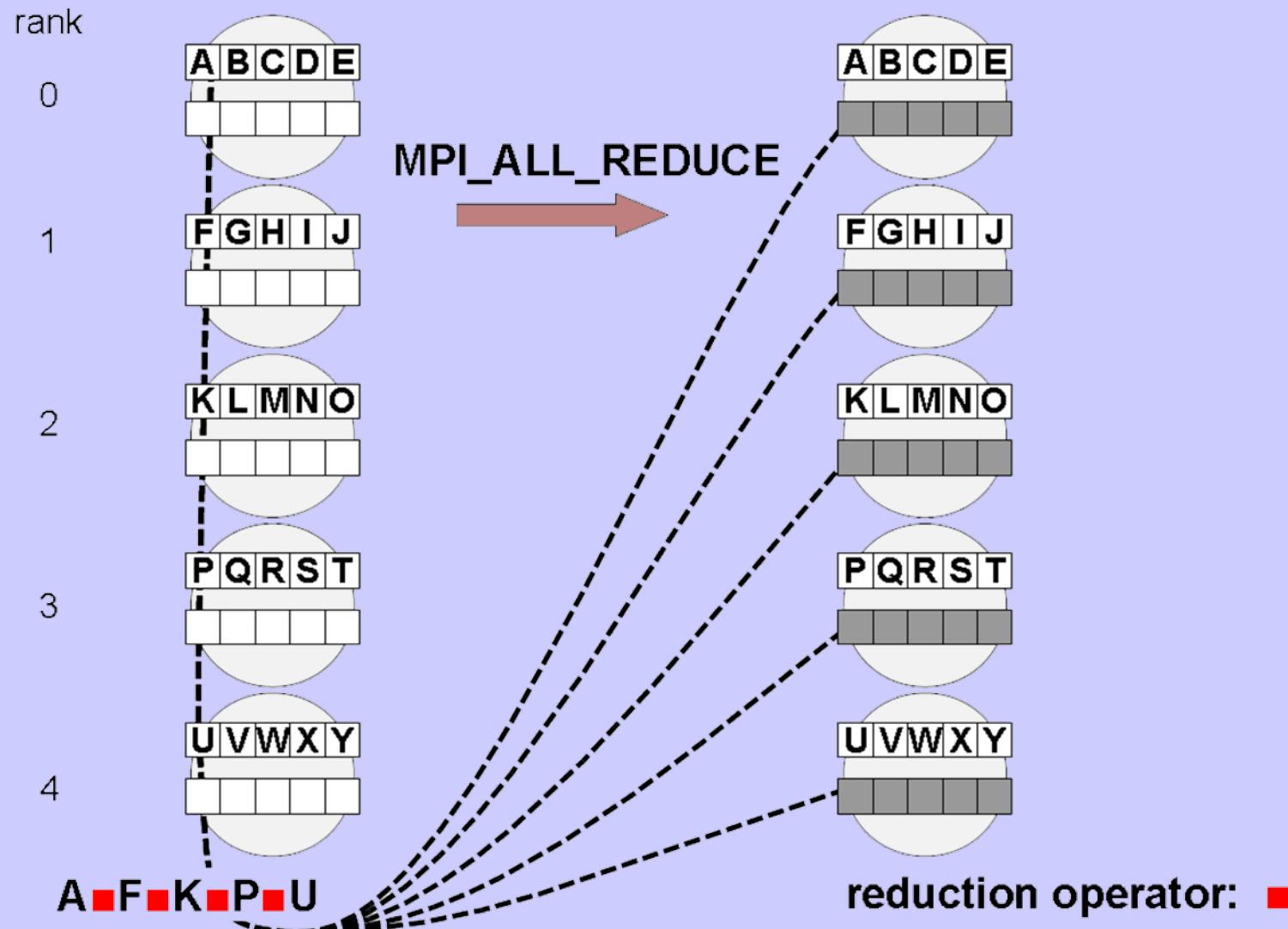
## 2. Reduzieren für Alle

- Sammelt wie **`MPI_Reduce`** Daten von allen Prozessen und **verrechnet** sie.  
Anschließend wird das Ergebnis an alle Prozesse **verteilt**.

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



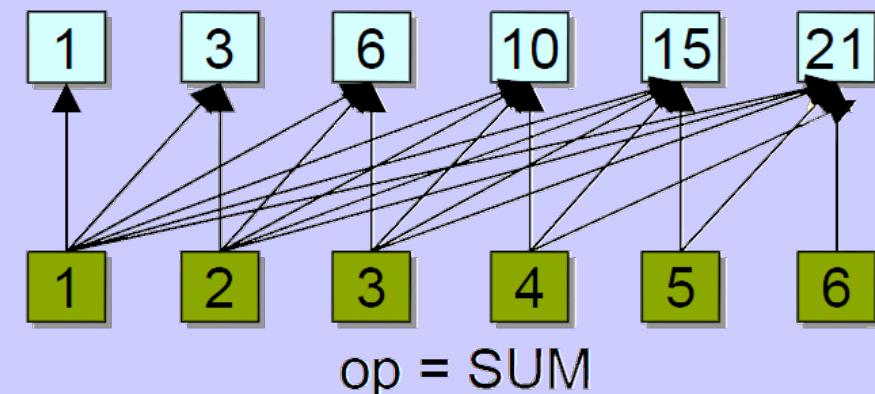
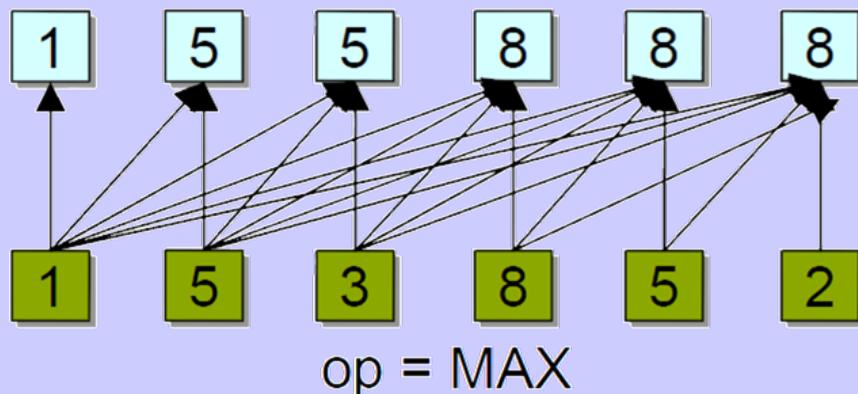
# Beispiel: MPI\_Allreduce



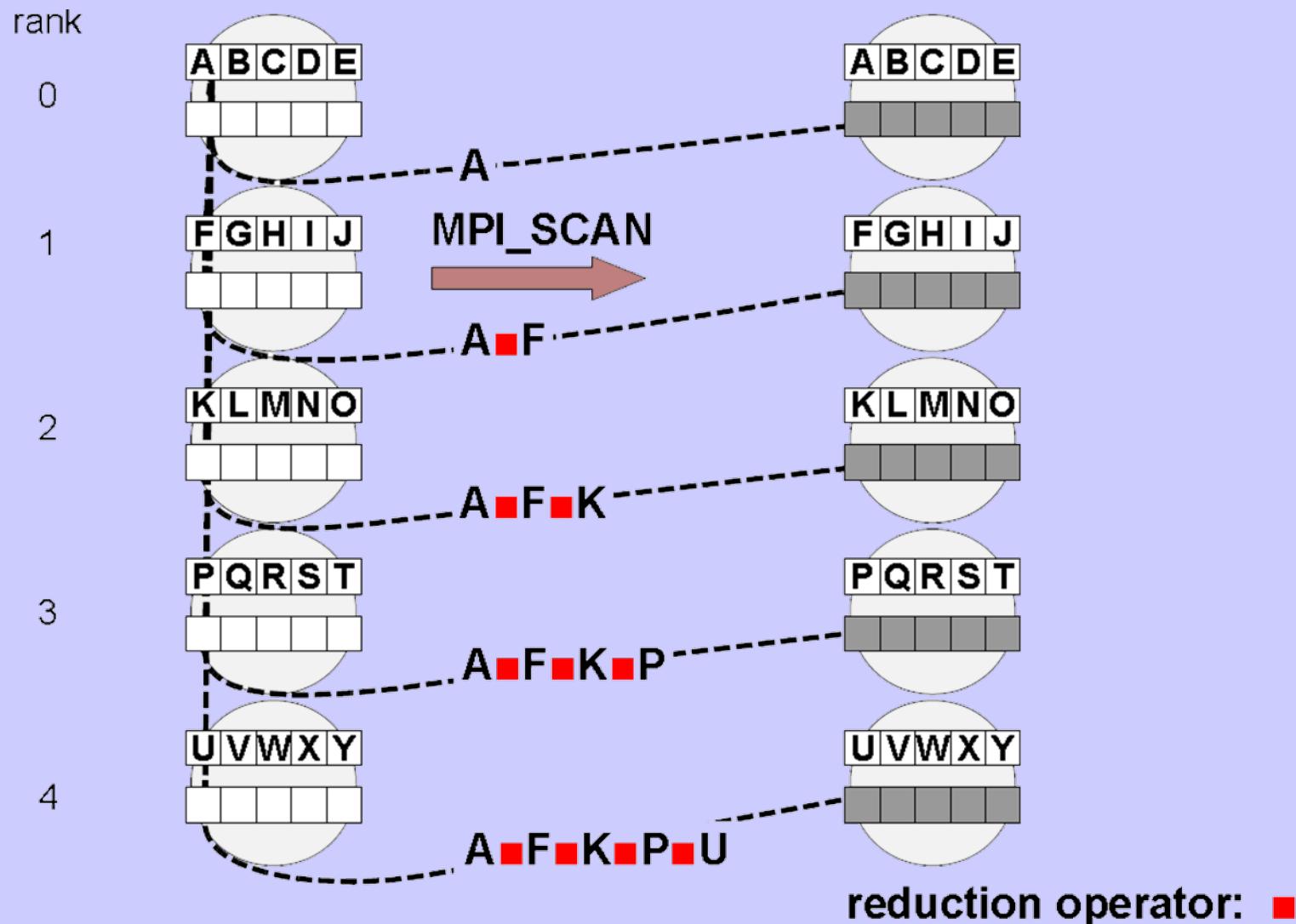
### 3. Reduzieren mit Zwischenergebnissen

- Sammelt Daten von einem Teil aller Prozessen und **verrechnet** sie. Prozess  $i$  erhält das **Teilergebnis** einer Reduktionsoperation auf den Prozessen  $1 \dots i$

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



# Beispiel: MPI\_Scan



# Beispiel: Maximumsuche mit Position

```
float ain[30], aout[30]; // Eingabe- und Ausgabefeld
struct {
    float val;                                // Datenpaar
    int rank;                               // Zahlenwert
} in[30], out[30];
int pid[30], rank, root = 1;

// ...
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Wer bin ich?
for (int i=0; i<30; i++) {           // Datenpaare erzeugen
    in[i].val = ain[i];                // Feld mit Zahlen
    in[i].rank = rank;                  // Prozess-ID
}
MPI_Reduce(in, out, 30, MPI_FLOAT_INT, MPI_MAXLOC,
              root, MPI_COMM_WORLD);

if (rank == root) {                  // Ergebnis liegt in root
    for (int i=0; i<30; i++) { // Datenpaare auslesen
        aout[i] = out[i].val;
        pid[i] = out[i].rank;
    }
}
// ...
```

## Aufgabe:

Jeder Prozess hat ein Feld `ain` mit 30 Fließkommazahlen. Für jedes Feldelement berechne den **Wert** und die **ID** von dem Prozess, der die **größte Zahl** enthält.

## Vordefinierte Datentyp-Paare:

**MPI\_FLOAT\_INT**  
**MPI\_DOUBLE\_INT**  
**MPI\_LONG\_INT**  
**MPI\_SHORT\_INT**  
**MPI\_LONG\_DOUBLE\_INT**

# Bedarf an eigenen Datentypen

- Optimale Nachricht für **gemischte Datentypen?**
  - Bislang einheitlich, zusammenhängend im Speicher
- Separates Versenden von verschiedenen Typen?
  - Verursacht Mehraufwand und bedeutet einen ineffizienten Nachrichtenaustausch
- Typ-Casting oder Konvertierung?
  - Kann gefährlich sein, besser vermeiden

# Abgeleitete Datentypen

- Ein allgemeiner Datentyp besteht aus zwei Komponenten:
  - einer Sequenz von **Basistypen**
  - einer Sequenz von **Abständen**  
(gerechnet von einer Basisadresse **buf** an).
- Dieses Komponentenpaar wird als **Typemap** bezeichnet:
  - Typemap =  $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$
- **Beispiel:**
  - Vordefinierter Typ **MPI\_FLOAT**: Typemap =  $\{(float, 0)\}$
- Ein neu erzeugter Typ muss **bekannt** gemacht werden
  - `int MPI_Type_commit(MPI_Datatype *datatype);`

# Zusammenhängende Daten und Vektoren

- int **MPI\_Type\_contiguous**(int count, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

- **Beispiel:** RGB Farbtyp mit 3 Ganzzahlen

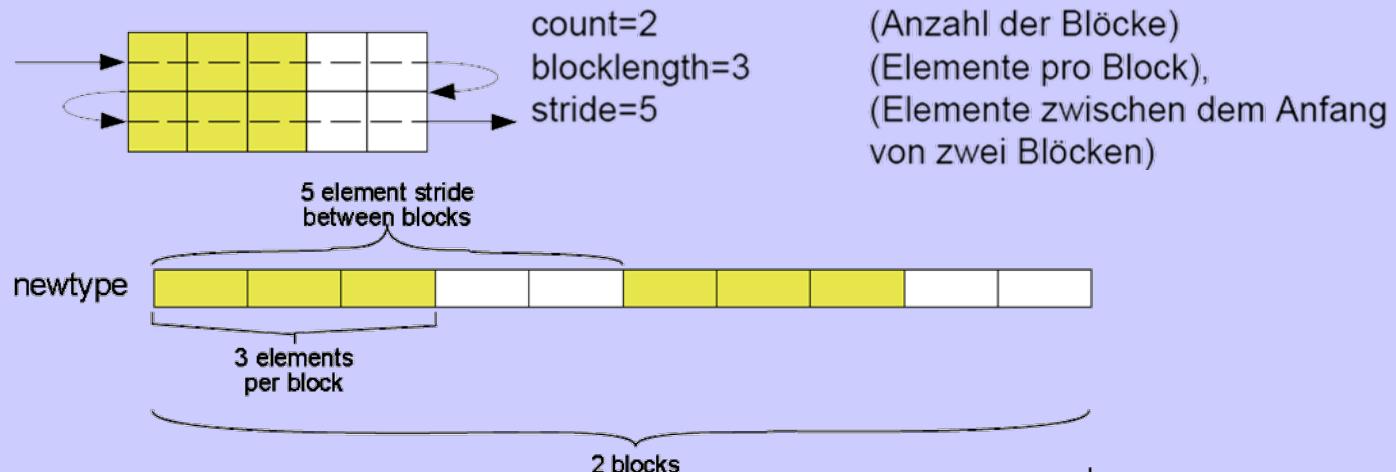
```
MPI_Type_contiguous(3, MPI_INT, &RGB_Color);
```

- int **MPI\_Type\_vector**(int count, int blocklength, int stride, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

- **MPI\_Type\_contiguous** = **MPI\_Type\_vector**(count, 1, 1, ...)

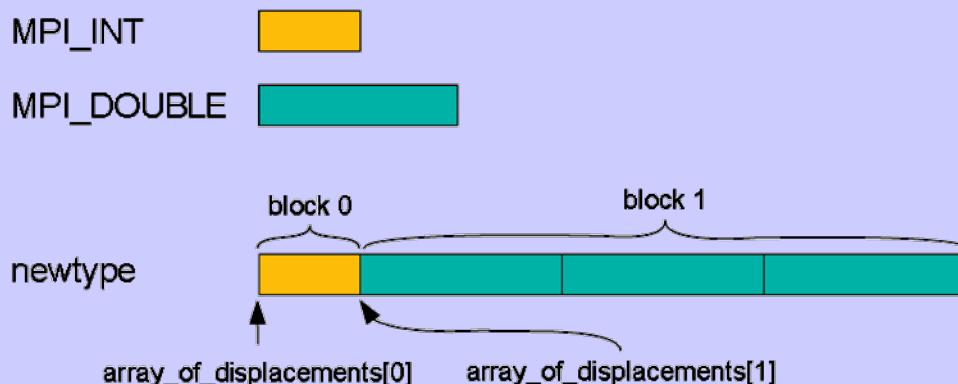
- **Beispiel:**

Eine  $3 \times 2$   
Untermatrix  
einer  $5 \times 2$   
Hauptmatrix



# Strukturen

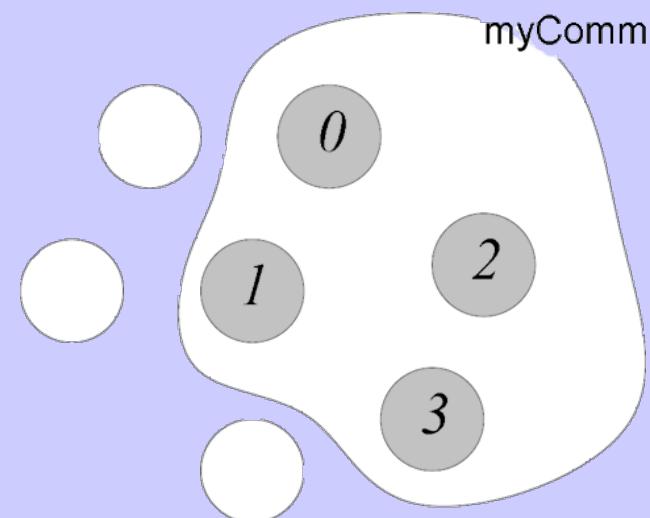
- `int MPI_Type_struct(int count,  
                      int *array_of_blocklengths,  
                      MPI_Aint *array_of_displacements,  
                      MPI_Datatype *array_of_types,  
                      MPI_Datatype *newtype)`
- **Beispiel:** 3D Raumkoordinate mit Nummer (ID, X, Y, Z)



```
count = 2  
  
array_of_blocklengths[0] = 1  
  
array_of_types[0] = MPI_INT  
  
array_of_blocklengths[1] = 3  
  
array_of_types[1] = MPI_DOUBLE
```

# Kommunikatoren und Gruppen

- **Unterscheidung** verschiedener Kontexte
- Konfliktfreie Organisation von **Gruppen**
- Kommunikatoren können nicht explizit erzeugt werden.  
Vielmehr kann ein Kommunikator nur aus einem **bestehenden** Kommunikator oder einer bestehenden Gruppe **abgeleitet** werden
- **Vordefinierte** Kommunikatoren
  - **MPI\_COMM\_WORLD**
  - **MPI\_COMM\_NULL**
  - **MPI\_COMM\_SELF**



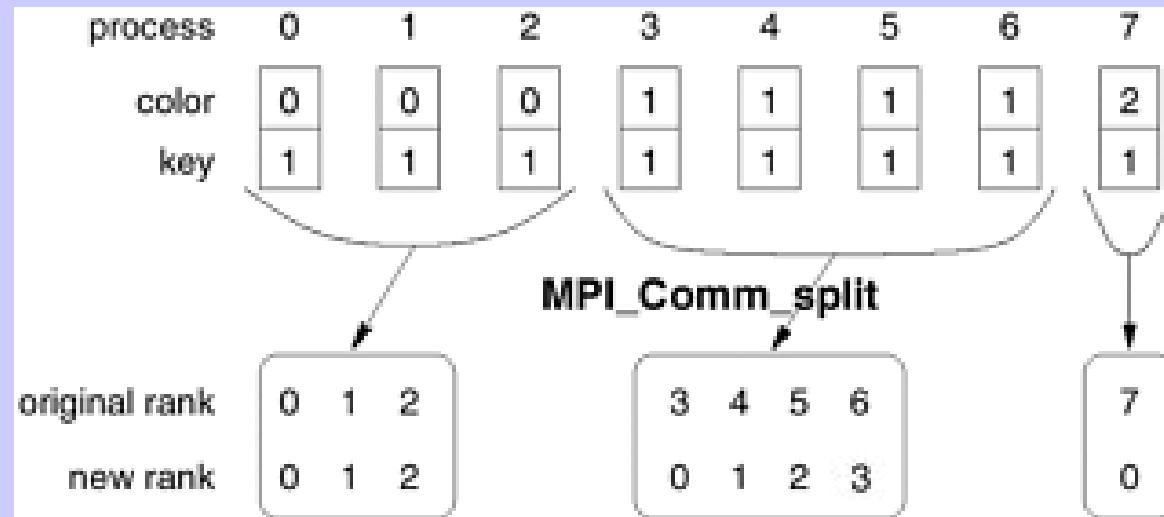
# Kommunikatoren duplizieren

- int **MPI\_Comm\_dup**(MPI\_Comm cOld, MPI\_Comm \*cNew) ;
- Erzeugt eine **Kopie** cNew vom Kommunikator cOld
- Erlaubt z.B. eindeutige Abgrenzung/Charakterisierung von Nachrichten
- **Beispiel**

```
MPI_COMM myworld;  
...  
MPI_Comm_dup(MPI_COMM_WORLD, &myworld)
```

# Kommunikatoren teilen

- `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *cNew);`
- Unterteilt Kommunikator `comm` in mehrere Kommunikatoren mit disjunkten Prozessgruppen
- Prozesse mit gleicher Farbe `color` bilden gemeinsamen neuen Kommunikator
- `key` steuert die Zuordnung der Ränge
- `MPI_Comm_split` muss von **allen** Prozessen in `comm` aufgerufen werden



# Beispiel: Kommunikator teilen

```
#include <mpi.h>

int main (int argc, char *argv[])
{
    int col, key, rank, size;
    MPI_Comm comm0, comm1, comm2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    col = rank % 3;                                // Color = 0,1,2
    key = size-rank-1;                             // Key = size-1,...,0
    if (col == 0) {
        MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, 0, &comm0);
    } else if (col == 1) {
        MPI_Comm_split(MPI_COMM_WORLD, col, key, &comm1);
    } else { // col == 2
        MPI_Comm_split(MPI_COMM_WORLD, col, key, &comm2);
    }
    MPI_Finalize();
}
```

# Ergebnis der Aufteilung $np = 9$

MPI\_COMM\_WORLD

| Rang  | P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 |
|-------|----|----|----|----|----|----|----|----|----|
| color | ⊥  | 1  | 2  | ⊥  | 1  | 2  | ⊥  | 1  | 2  |
| key   | 0  | 7  | 6  | 0  | 4  | 3  | 0  | 1  | 0  |



MPI\_COMM\_WORLD

comm1

|    |    |    |
|----|----|----|
| P1 | P4 | P7 |
| 2  | 1  | 0  |

comm2

|    |    |    |
|----|----|----|
| P2 | P5 | P8 |
| 2  | 1  | 0  |

|    |    |    |
|----|----|----|
| P0 | P3 | P6 |
| 0  | 1  | 2  |

# Kommunikatoren auflösen

- int **MPI\_Comm\_free**(MPI\_Comm \*comm) ;
- Löschen des Kommunikators **comm**
- Die von **comm** belegten Ressource werden von MPI freigegeben
- Kommunikator hat nach dem Aufruf den Wert des Null-Handles  
**MPI\_COMM\_NULL**
- Funktion muss von **allen** Prozessen aus **comm** aufgerufen werden

# Prozessgruppen

- Zu jedem **Kommunikator** lässt sich die **Gruppe** ermitteln
- Aus jeder **Gruppe** kann man einen **Kommunikator** konstruieren
- **Prozessgruppen** bestehen aus einer Menge **durchgehend** und **eindeutig** nummerierter Prozessidentifikatoren
- Außerdem legt sie fest, welche Prozesse in eine **kollektive Operation** einbezogen sind



# Funktionen für Gruppen

- int **MPI\_Comm\_group** (MPI\_Comm comm, MPI\_Group \*group)  
Zugriff auf die **Prozessgruppe** eines Kommunikators
- int **MPI\_Comm\_create** (MPI\_Comm cOld, MPI\_Group group,  
MPI\_Comm \*cNew)  
Erzeugen eines **Kommunikators** aus einer Gruppe
- int **MPI\_Group\_incl** (MPI\_Group gOld, int nranks, int \*ranks,  
MPI\_Group \*gNew)  
**Hinzufügen** von Prozessen in eine Gruppe
- int **MPI\_Group\_excl** (MPI\_Group gOld, int nranks, int \*ranks,  
MPI\_Group \*gNew)  
**Herauslösen** von Prozessen aus einer Gruppe
- int **MPI\_Group\_range\_incl** (MPI\_Group gOld, int nranges,  
int ranges[][][3], MPI\_Group \*gNew)  
Bilden einer Gruppe aus einfachen **Mustern** (Anfang, Ende, Abstand)
- int **MPI\_Group\_range\_excl** (MPI\_Group gOld, int nranges,  
int ranges[][][3], MPI\_Group \*gNew)  
Ausschließen von Prozessen mit einfachen Mustern

# Beispiel: Gruppe mit geraden PIDs

```
int main(int argc, char *argv[])
{
    MPI_Group group_world, even_group;;
    int i, p, Neven, members[8];

    //...
    MPI_Comm_size(MPI_COMM_WORLD, &p);           // für p <= 16
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);

    Neven = (p+1)/2;
    for (i = 0; i < Neven; i++) {
        members[i] = 2*i;
    }
    MPI_Group_incl(group_world, Neven, members, &even_group);
    //...
}
```

# Beispiel: Ausgewählte Bereiche

```
#define MAX 2

void main(int argc, char *argv[])
{
    MPI_Group group, newgroup;
    int ranges[MAX][3];

    // ...
    MPI_Comm_group(MPI_COMM_WORLD, &group);
    ranges[0][0] = 1;    // Erster
    ranges[0][1] = 5;    // Letzter
    ranges[0][2] = 2;    // Schrittweite
    ranges[1][0] = 6;    // Erster
    ranges[1][1] = 7;    // Letzter
    ranges[1][2] = 1;    // Schrittweite
    // Aus mindestens 8 Prozessen werden 1, 3, 5, 6 und 7 ausgewählt
    MPI_Group_range_incl(group, MAX, ranges, &newgroup);
    // ...
}
```

# Operationen auf Kommunikatorgruppen

- Darüber hinaus existieren weitere **Funktionen zur Gruppierung**:

- **Zusammenfassen** von Gruppen

```
int MPI_Group_union(MPI_Group g1, MPI_Group g2, MPI_Group *gRes)
```

- **Schnittmenge** von Gruppen

```
int MPI_Group_intersection(MPI_Group g1, MPI_Group g2,  
                           MPI_Group *gRes)
```

- **Differenz** von Gruppen

```
int MPI_Group_difference(MPI_Group g1, MPI_Group g2,  
                         MPI_Group *gRes)
```

- **Vergleich** von Gruppen (**MPI\_IDENT**, **MPI\_SIMILAR**, **MPI\_UNEQUAL**)

```
int MPI_Group_compare(MPI_Group g1, MPI_Group g2, int *result)
```

- **Auflösen** von Gruppen

```
int MPI_Group_free(MPI_Group *group)
```

- **Größe** einer Gruppe

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

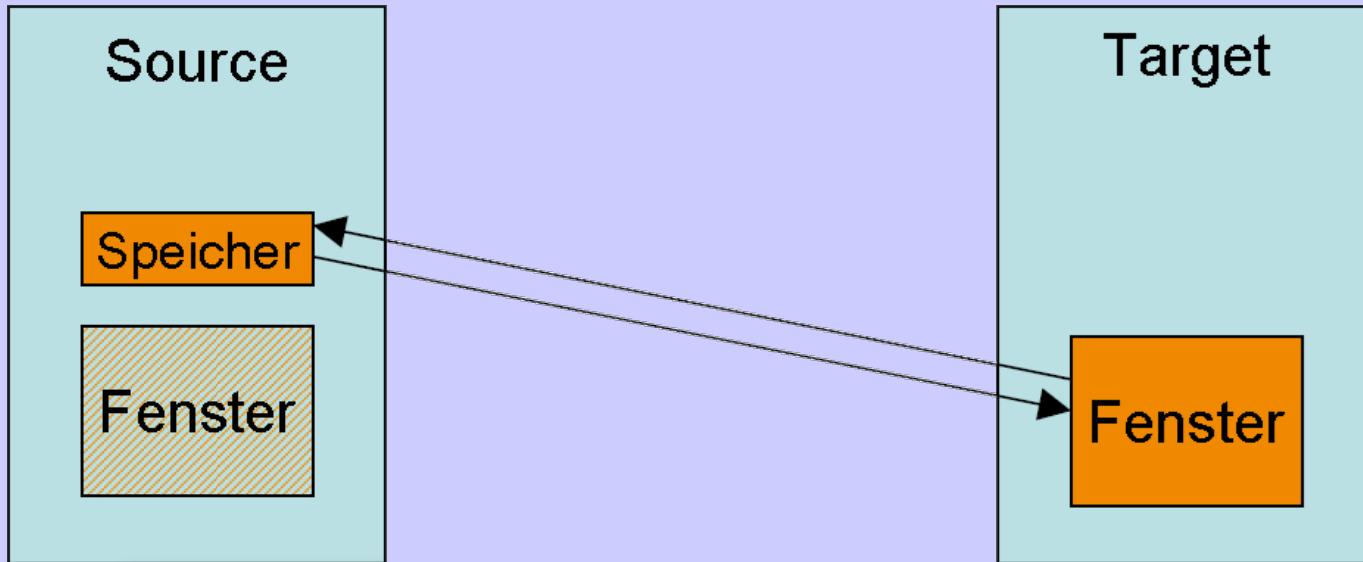
- **Rang** einer Gruppe

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

# Ausblick: Entwicklung MPI

- Neue Version des Standards, erweitert MPI in einigen Bereichen
- Loslösung vom starren **Prozessmodell**
  - **Starten** weiterer Prozesse **zur Laufzeit** möglich.
  - Mit den neu gestarteten Prozessen kann kommuniziert werden.
- Kommunikation mit schon **laufenden MPI-Anwendungen** möglich.
  - Kommunikation über sog. Ports.
  - Finden benannter Ports über eine Registry.
- **Einseitige Kommunikation:** Put, Get, Accumulate, sowie Methoden zur Synchronisation
- Parallele **Datei-Ein- und Ausgabeoperation**
- Anbindung für C++

# Einseitige Kommunikation



- Knoten „Source“ greift mittels `get()` und `put()` auf das Fenster des Knotens „Target“ zu.
- Für Quell- bzw. Zieldaten können im Prinzip beliebige Speicherbereiche verwendet werden.
- Knoten „Target“ stellt sein Fenster zur Verfügung.  
Evtl. geänderte Daten sind erst nach der Synchronisation sichtbar.

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- **Massiv parallele Programmierung**

- Einführung in OpenCL
  - Plattform-Modell
  - Ausführungs-Modell
  - Speicher-Modell
  - Programmierungs-Modell



# Einführung in OpenCL

Massive Parallelität mittels  
Grafikprozessoren (GPUs)

# Beispiel: Top500 Supercomputer

| Rank | Site                                                            | System                                                                                                                      | Cores      | Rmax<br>(TFlop/s) | Rpeak<br>(TFlop/s) | Power<br>(kW) |
|------|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|------------|-------------------|--------------------|---------------|
| 1    | National Supercomputing Center in Wuxi China                    | <b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRPC                                             | 10,649,600 | 93,014.6          | 125,435.9          | 15,371        |
| 2    | National Super Computer Center in Guangzhou China               | <b>Tianhe-2 (MilkyWay-2)</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT | 3,120,000  | 33,862.7          | 54,902.4           | 17,808        |
| 3    | DOE/SC/Oak Ridge National Laboratory United States              | <b>Titan</b> - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.                         | 560,640    | 17,590.0          | 27,112.5           | 8,209         |
| 4    | DOE/NNSA/LLNL United States                                     | <b>Sequoia</b> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM                                                             | 1,572,864  | 17,173.2          | 20,132.7           | 7,890         |
| 5    | RIKEN Advanced Institute for Computational Science (AICS) Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu                                                                | 705,024    | 10,510.0          | 11,280.4           | 12,660        |
| 6    | DOE/SC/Argonne National Laboratory United States                | <b>Mira</b> - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM                                                                 | 786,432    | 8,586.6           | 10,066.3           | 3,945         |
| 7    | DOE/NNSA/LANL/SNL United States                                 | <b>Trinity</b> - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.                                         | 301,056    | 8,100.9           | 11,078.9           |               |
| 8    | Swiss National Supercomputing Centre (CSCS) Switzerland         | <b>Piz Daint</b> - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect. NVIDIA K20x Cray Inc.                           | 115,984    | 6,271.0           | 7,788.9            | 2,325         |
| 9    | HLRS - Höchstleistungsrechenzentrum Stuttgart Germany           | <b>Hazel Hen</b> - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.                                       | 185,088    | 5,640.2           | 7,403.5            |               |
| 10   | King Abdullah University of Science and Technology Saudi Arabia | <b>Shaheen II</b> - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.                                      | 196,608    | 5,537.0           | 7,235.2            | 2,834         |

|                               | GPU               | CPU                     |
|-------------------------------|-------------------|-------------------------|
| Model                         | Nvidia Tesla K20X | Intel Xeon E5-2670 v3   |
| Architecture                  | Kepler            | Haswell                 |
| Launch                        | Nov-2012          | Sep-2014                |
| # of transistors              | 7.1billion        | 3.84billion             |
| # of cores                    | 2688 (simple)     | 12 (functional)         |
| Core clock                    | 732MHz            | 2.6GHz,<br>up to 3.5GHz |
| Peak Flops (single precision) | 3.95TFLOPS        | 998.4GFLOPS (with AVX2) |
| DRAM size                     | 6GB, GDDR5        | 768GB/socket,<br>DDR4   |
| Memory band                   | 250GB/s           | 68GB/s                  |
| Power consumption             | 235W              | 135W                    |
| Price                         | \$3,000           | \$2,094                 |

# (GP-)GPU-Cluster



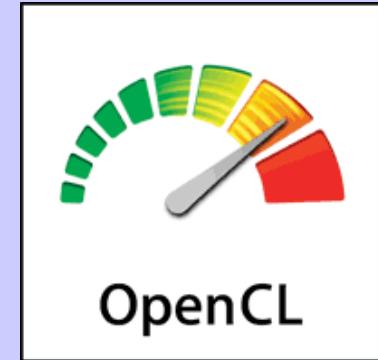
John the Ripper: Rekorde im  
Passwort-Knacken mit GPUs  
(25 AMD GPUs in 5 Servern)



Deep Learning mit NVIDIA (Tesla/Geforce)

# Open Computing Language (OpenCL)

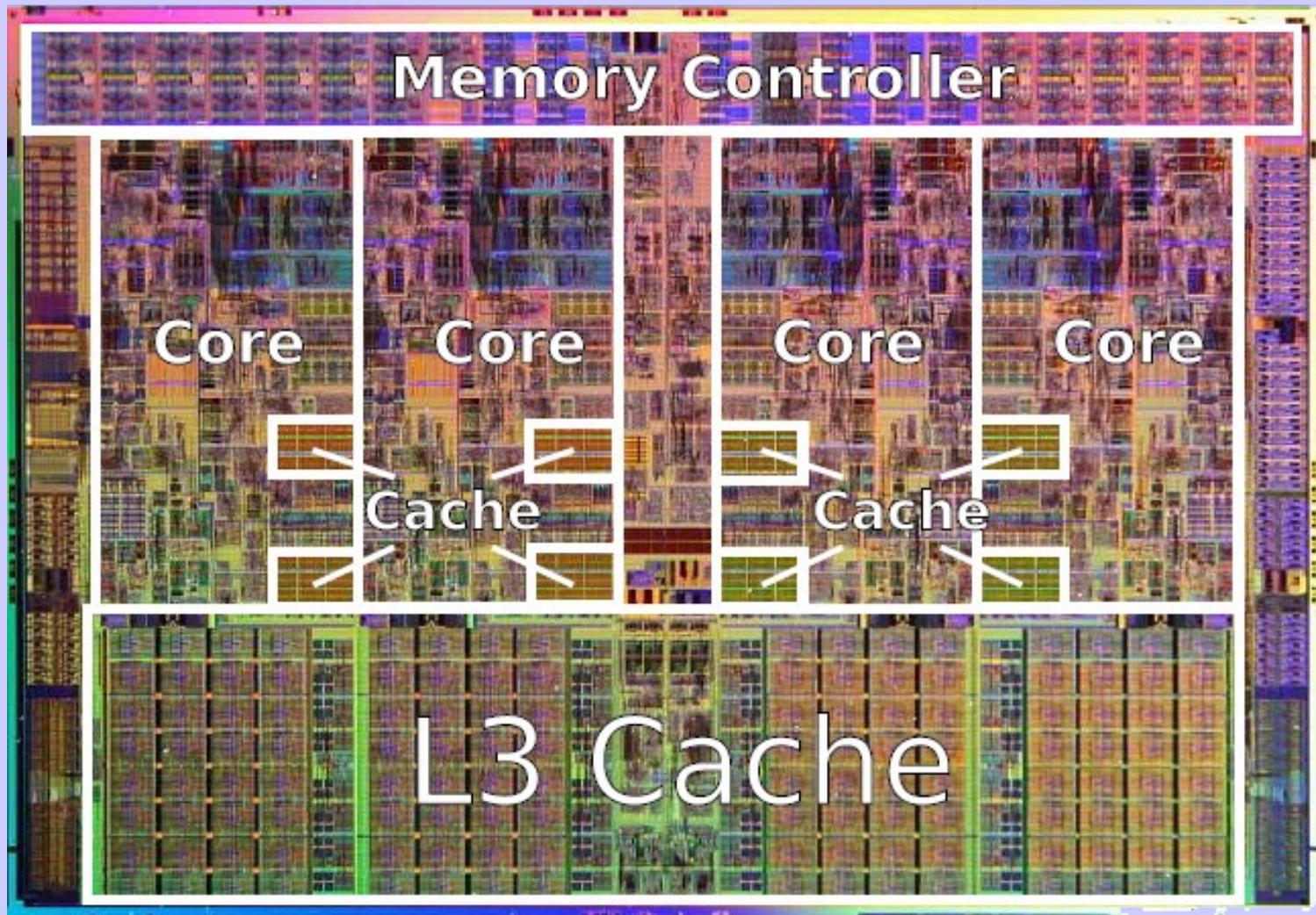
- OpenCL ist ein **offener** Standard für **plattformunabhängiges** Rechnen
- Programmierschnittstelle für Parallelrechner, die mit **Haupt-, Grafik- oder digitalen Signalprozessoren** ausgestattet sind
- Die zugehörige Programmiersprache ist **OpenCL C**
- Initiiert von Apple in Kooperation mit AMD, ARM, IBM, Intel und Nvidia
- Standardisierung betreut bei der **Khronos Group** ([www.khronos.org](http://www.khronos.org))
- Version 1.0 (2008) bis Version 2.2 (Mai 2017)
- Nützliche Links
  - OpenCL für **Nvidia**:  
<https://developer.nvidia.com/cuda-zone>  
**(CUDA Toolkit SDK 10.0)**
  - OpenCL für **AMD**:  
~~<http://developer.amd.com/tools-and-sdks/opencl-zone/>~~  
**(Accelerated Parallel Processing (APP) SDK 3.0)**



# Beispielanwendungen mit OpenCL

- GIMP, Adobe Photoshop, Agisoft PhotoScan,
- Autodesk Maya, Blender, Maxon Cinema 4D
- FFmpeg, Final Cut Pro, MAGIX Vegas,
- Siemens Nastran/Aquaqus FEM,
- Convolutional Neural Networks / DeepCL,
- Google Chrome, Mozilla Firefox
- Boost Library
- ...

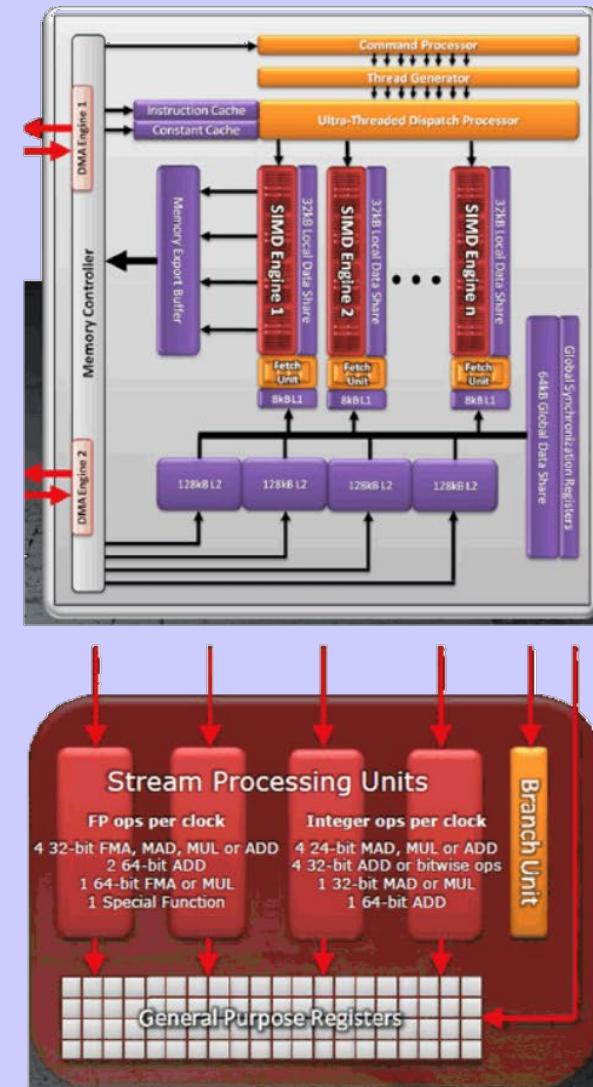
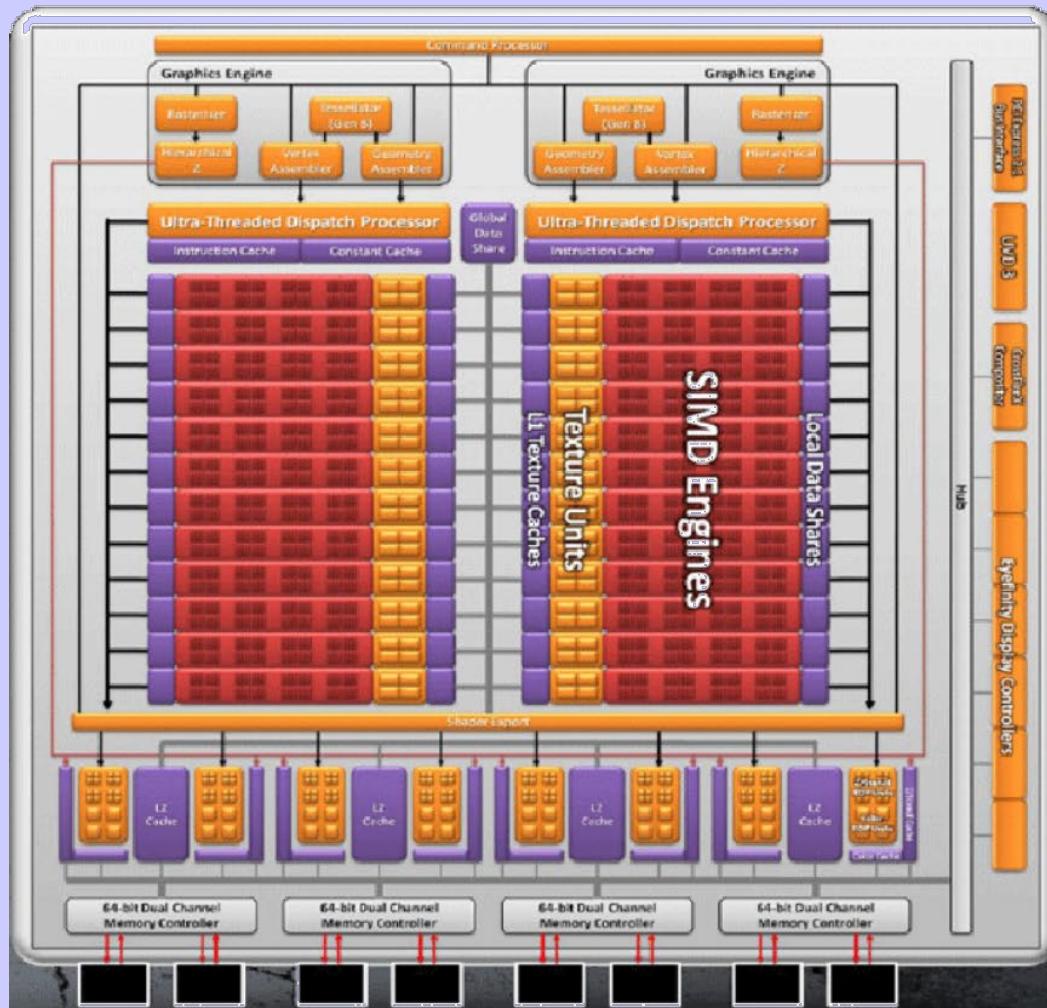
# Hauptprozessor Intel Core i7



# Nvidia GF100



# AMD Cayman



# Grundlagen von OpenCL

- **Plattformmodell:**

Eine abstrakte Beschreibung von heterogenen Systemen mit unterschiedlichen physikalischen Eigenschaften

- **Ausführungsmodell:**

Eine abstrakte Beschreibung, wie Befehle auf solch heterogenen Plattformen ausgeführt werden

- **Speichermodell:**

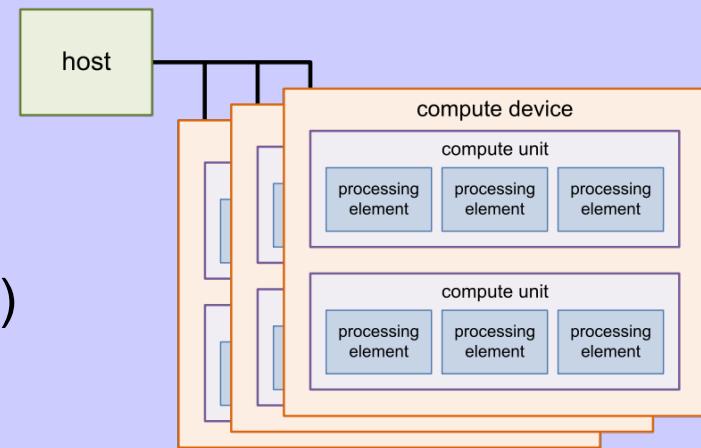
Die unterschiedlichen Speichertypen und wie sie während der Berechnung verwendet werden können

- **Programmiermodell:**

Die Abstraktion, die ein Programmierer verwenden kann, wenn er Algorithmen entwirft

# 1. Plattform-Modell

- **Host**
  - **Rechner**, der die *Compute Devices* verwaltet
  - Verteilt die Arbeit an die Devices
- **Compute Device**
  - Genutzte Rechenressource  
(z.B. **Grafikkarte**, Prozessor, Cell-Blade)
- **Compute Unit**
  - Zusammenschluss von „ausführenden Elementen“  
(z.B. **Shader**, Rechenkern, Coprozessoren)
- **Processing Element**
  - Eigentliches **Rechenelement**

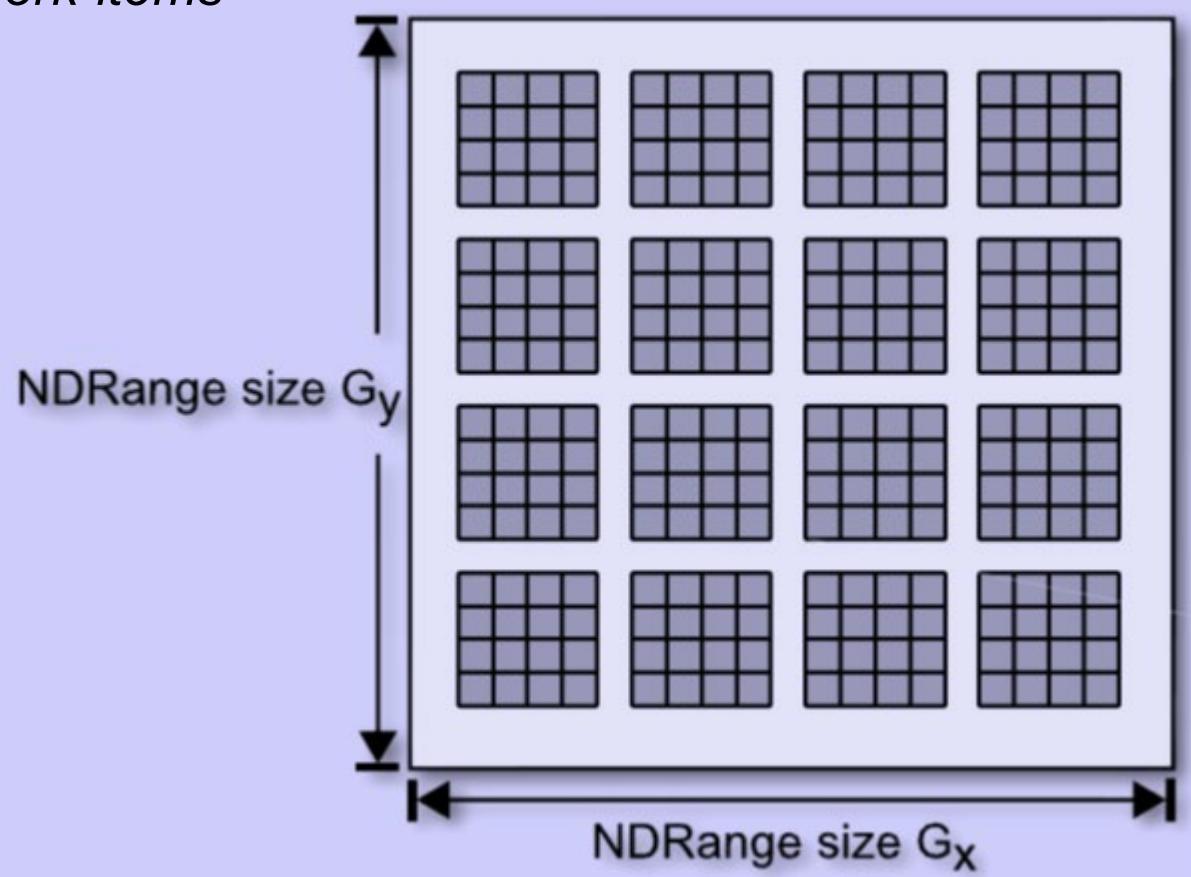


## 2. Ausführungs-Modell

- **Bestandteile eines OpenCL Programms**
  - Serieller **Host**-Code in C/C++  
(Initialisierung, Speicherverwaltung, etc.)
  - Paralleler **Kernel**-Code in OpenCL-C  
(wird auf dem Device ausgeführt)
- **Kernel Ausführung**
  - Hostprogramm ruft Kernel auf einem **Indexraum**  
mit 1-3 Dimensionen auf
  - Eine Instanz eines Kernels ist ein (logisches) **Work-item**
  - Einzelne *Work-items* sind in **Work-groups** zusammengefasst
- **Befehlswarteschlange**
  - Kernelausführung
  - Speichertransfer
  - Synchronisation

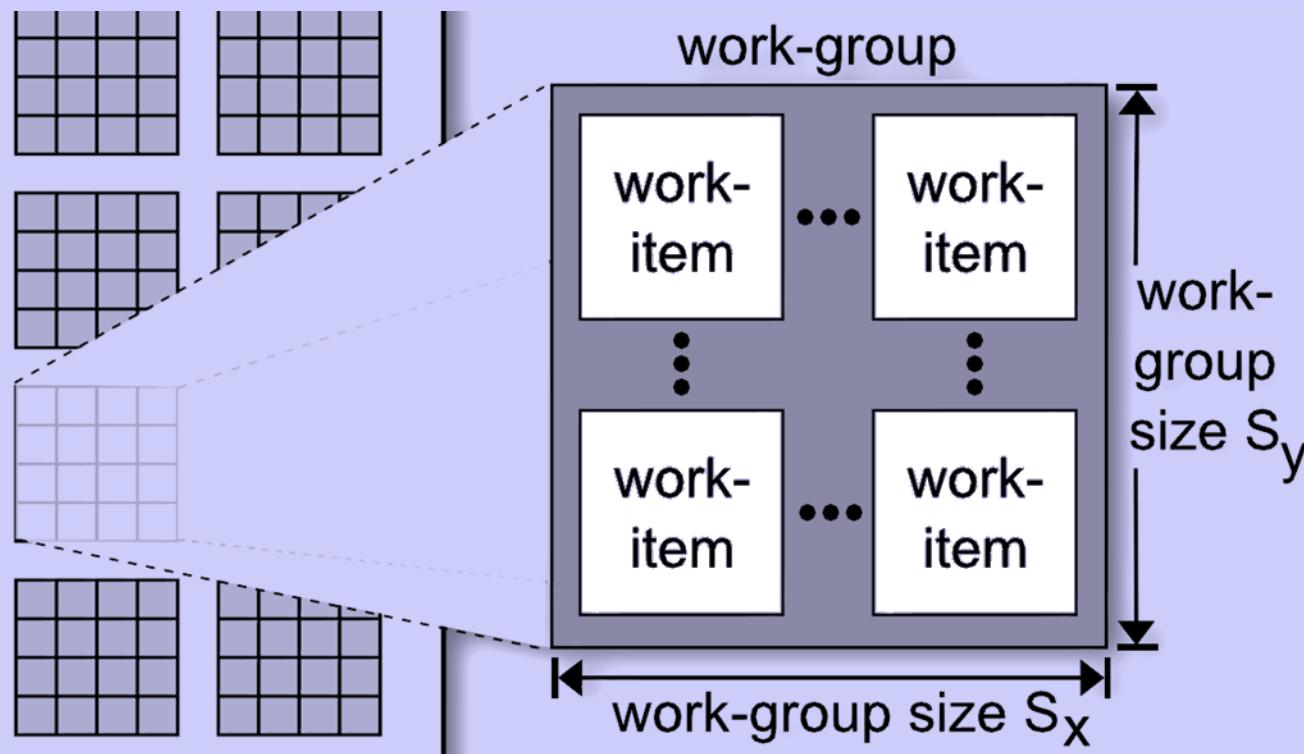
# N-dimensionaler Indexraum

- **NDRange**: Aufteilen der gesamten Aufgabe in einen  $n$ -dimensionalen Bereich ( $n = 1, \dots, 3$ )
- Globale ID eines *Work-items*

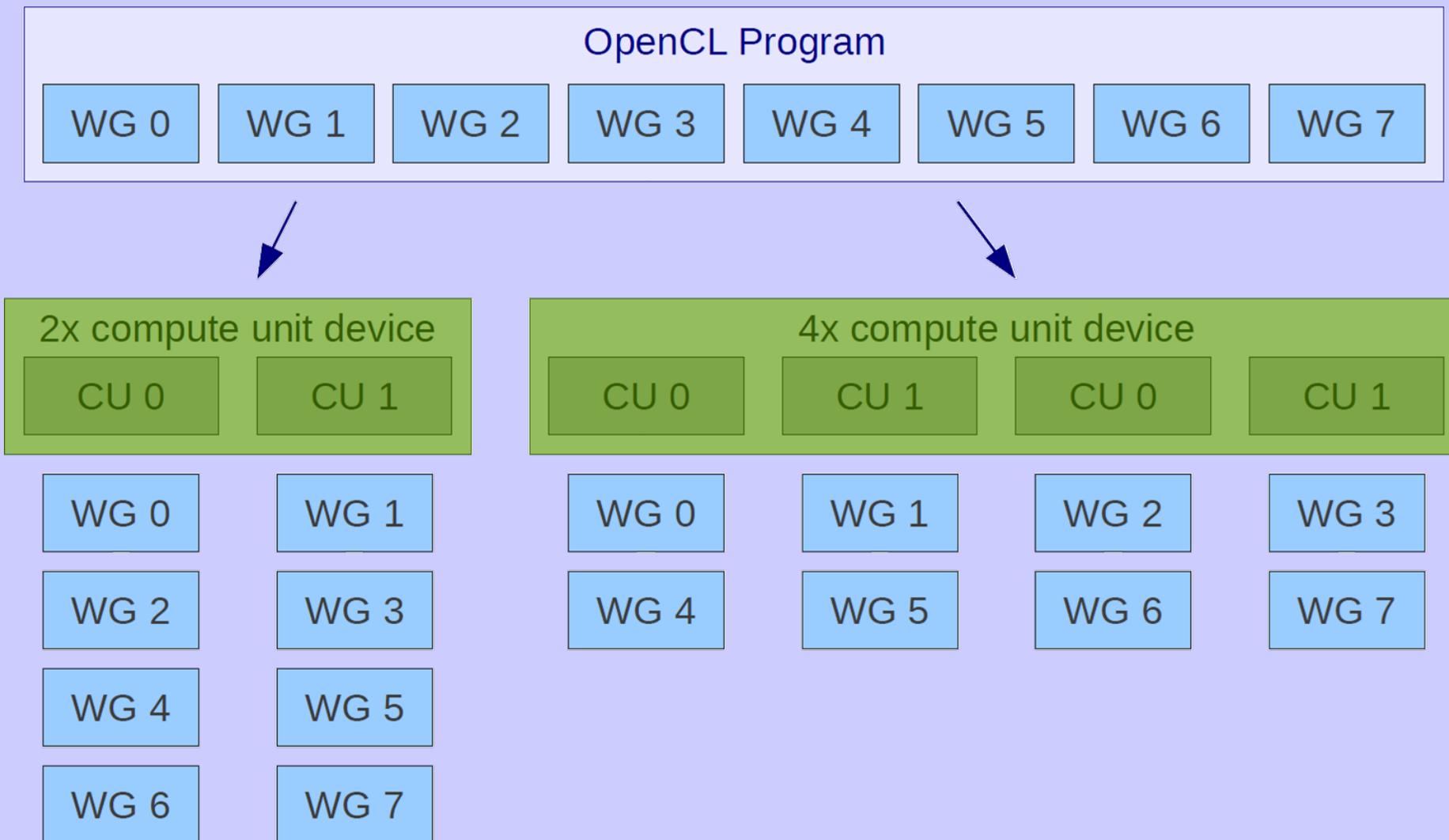


# Work-items und Work-groups

- **Work-item:** entspricht einer Kernel-Instanz
- **Work-group:** Zusammenfassung von *Work-items* mit gemeinsamem Speicher die gleichzeitig Rechnen

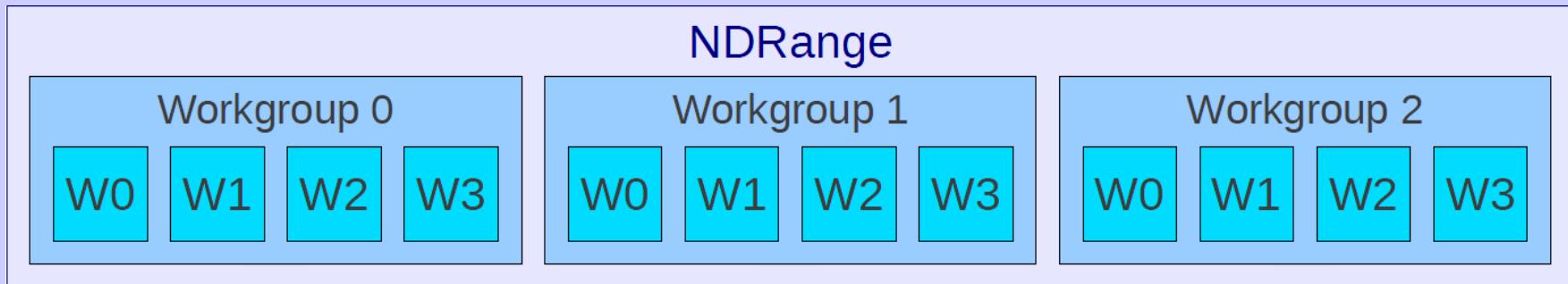


# Thread Topologie



# Thread Topologie

- OpenCL verwendet ein skalierbares Programmiermodell



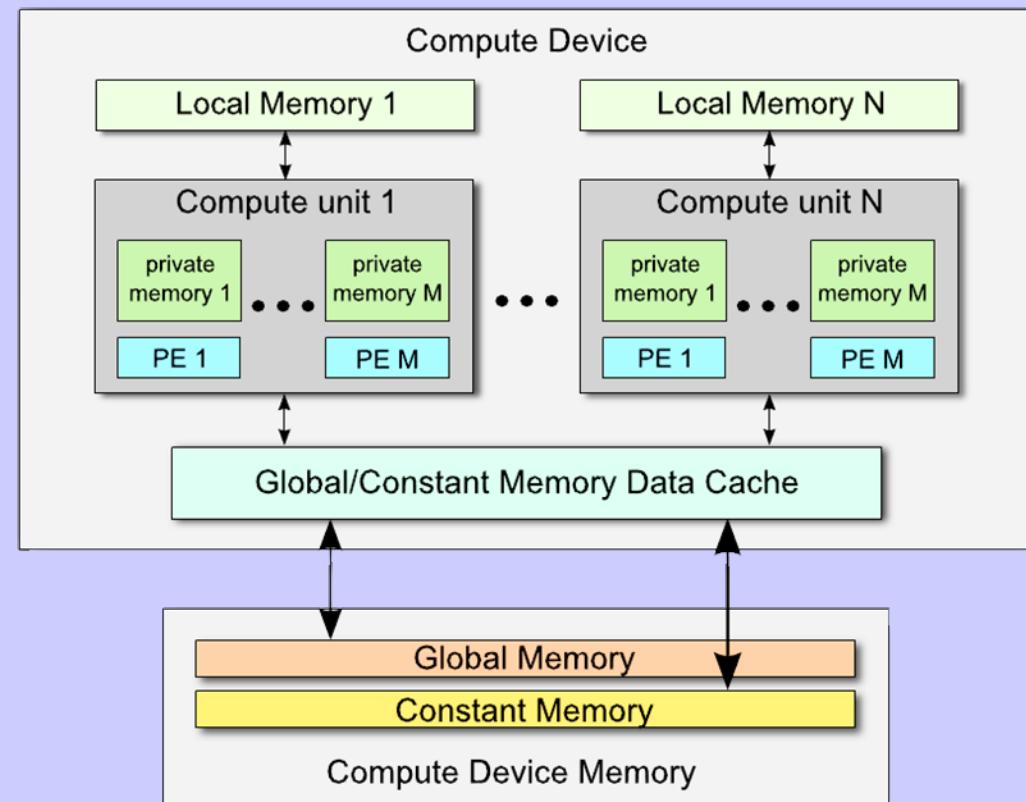
- Jedes *Work-item* kann die **Dimension** des *NDRange*, der *Work-Group* und seinen eigenen **Index** abfragen:

```
uint get_work_dim()  
size_t get_global_size(uint d)  
size_t get_global_id(uint d)  
size_t get_local_size(uint d)  
size_t get_local_id(uint d)
```

# 3. Speichermodell

## • Adressräume

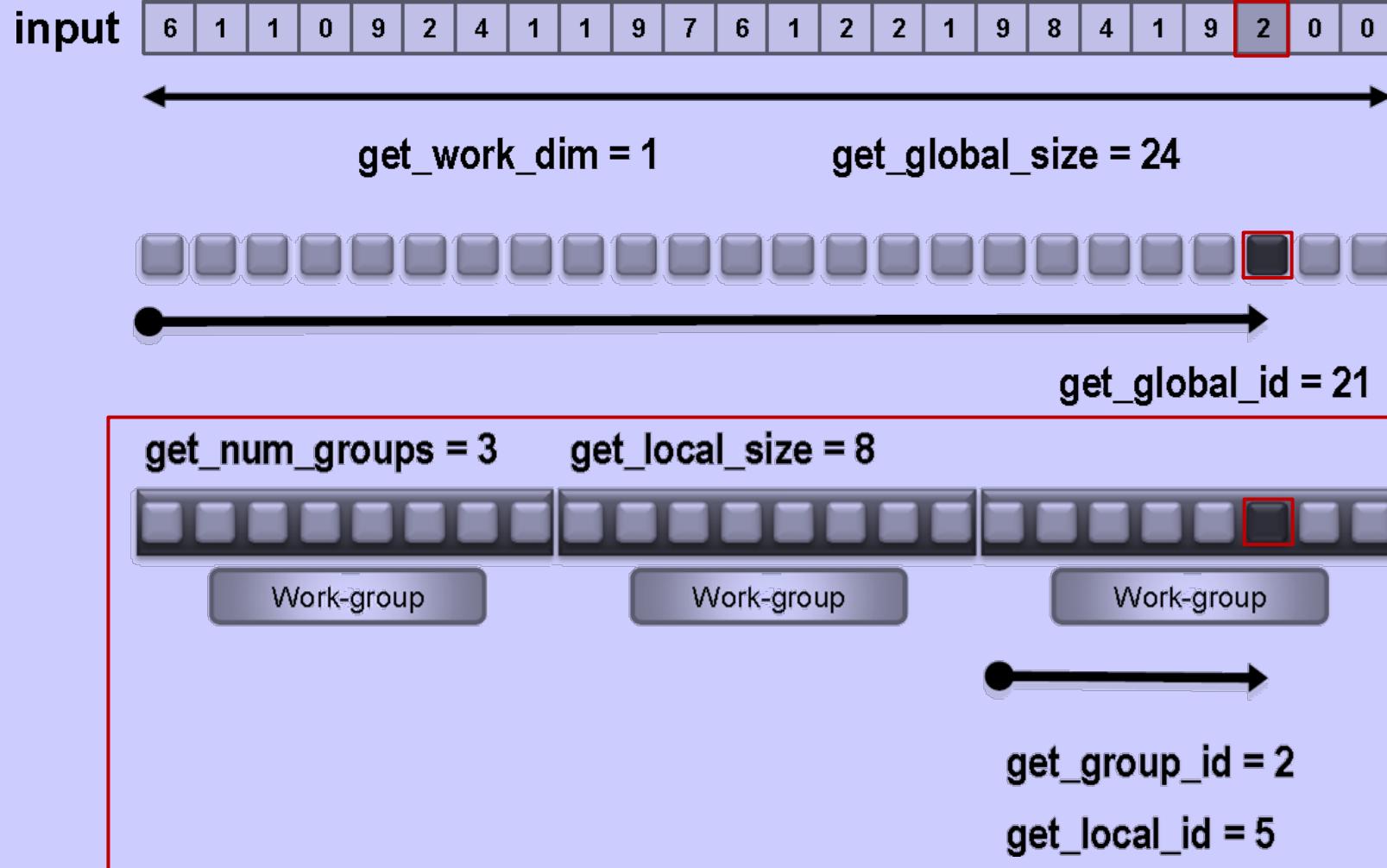
- **Global:** Arbeitsspeicher des *Compute Devices*. Von allen *Work-items* aus allen *Work-groups* nutzbar
- **Konstant:** Teil des *Global Memory* und konstant während der Laufzeit
- **Daten-Cache:** optional
- **Lokal:** zu einer *Work-group* gehörend
- **Privat:** zu einem *Work-item* gehörend



# Zugriffsrechte für Gerätespeicher

|               | Global              | Constant            | Local               | Private             |
|---------------|---------------------|---------------------|---------------------|---------------------|
| <b>Host</b>   | Dynamic allocation  | Dynamic allocation  | Dynamic allocation  | No allocation       |
|               | Read / Write access | Read / Write access | No access           | No access           |
| <b>Kernel</b> | No allocation       | Static allocation   | Static allocation   | Static allocation   |
|               | Read / Write access | Read-only access    | Read / Write access | Read / Write access |

# Lokaler Speicher und Arbeitsgruppen



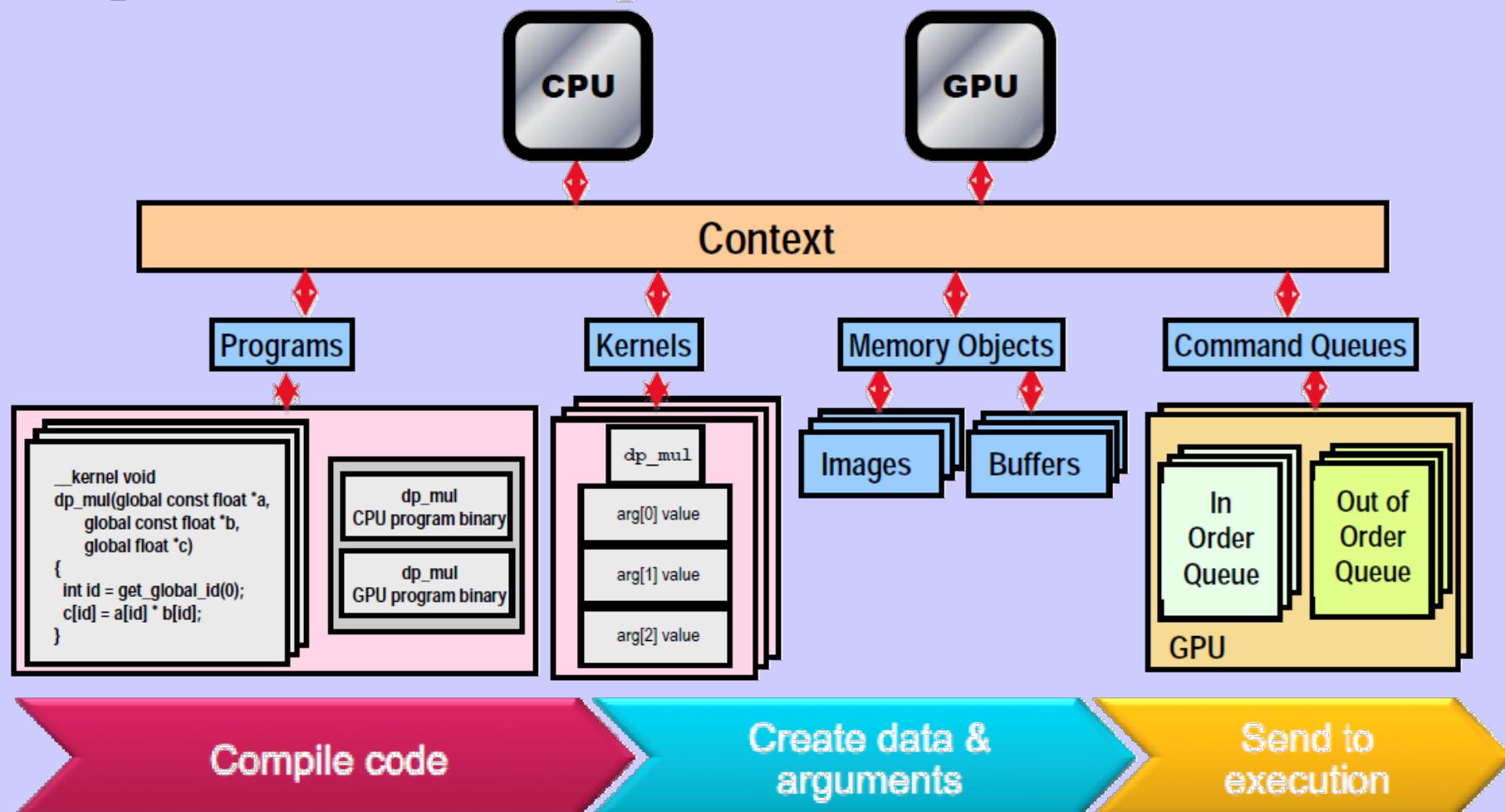
# 4. Programmierungs-Modell

- **Daten-paralleles Modell**

- Single Instruction Multiple Data (**SIMD**)
- Single Program Multiple Data (**SPMD**)
- **Implizites Modell:**  
Der *NDRange* wird automatisch verteilt
- **Explizites Modell:**  
Der Programmierer definiert die Größe der *Work-groups*

- **Task-paralleles Modell**

# Programmierung mit OpenCL



# Beispiel: "Hello World" auf GPUs?

- Grafikkarten erzeugen üblicherweise **keine Textausgabe** auf der Konsole!
- Einfaches **Beispiel-Programm**
  - Quadrieren von Elementen eines Feldes  
(auf die besonders harte Weise!)
- **Ziel** des Beispiels
  - Demonstrieren der **Initialisierung** von OpenCL
  - Bereitstellen eines einfachen OpenCL **Kerns**
  - Zeigen, dass obwohl **viele Schritte** benötigt werden,  
es nicht wirklich kompliziert ist (= Gebrauchsanweisung)

# Kern „hello“ mit $\text{output}_i = \text{input}_i^2$

```
__kernel void hello(__global float *input, __global float *output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

- **(Rechen)Kern / Kernel**
  - Code, der auf jedem einzelnen Processing Element ausgeführt wird
- **Hinweise:**
  - **float** mit **einfacher** Genauigkeit bevorzugt!
  - 2D-Matrizen sollten als **1D-Vektoren** angesprochen werden (z.B. bei dynamischen Matrizen z.B. mit **A[0]**)

# Spracheigenschaften OpenCL C/C++

- Kernel erlauben im wesentlichen C99, ohne
  - Funktionszeiger
  - Rekursion
  - Arrays variabler Länge
  - Strukturen
- Seit OpenCL 2.2 (2017) auch C++14
  - Klassen
  - Templates
  - Lambda-Ausdrücke
- Optionale Features
  - Gleitkommazahlen mit doppelter Genauigkeit

# Aufruf-Beispiel: Hello.cpp

```
#include "CL/cl.h"

int main(void)
{
    // ...
    clGetPlatformIDs(1, &platform_id, &num_of_platforms);
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_of_devices);
    clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    command_queue = clCreateCommandQueue(context, device_id, 0, &err);
    program = clCreateProgramWithSource(context, 1, (const char **)&KernelSource, NULL, &err);
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "hello", &err);
    input = clCreateBuffer(context, CL_MEM_READ_ONLY, MEM_SIZE, NULL, &err);
    output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, MEM_SIZE, NULL, &err);
    clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, 0, NULL, NULL);
    clFinish(command_queue);
    clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, MEM_SIZE, results, 0, NULL, NULL);
    // ...
}
```

# Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines OpenCL-Kontextes und Erstellen einer **Befehlswarteschlange**
3. Online-Kompilierung des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von OpenCL-Speicherobjekten für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-Ausführung und Sammeln der Ergebnisse



# 1. Initialisierung der Geräte

```
int main(void)
{
    cl_platform_id platform_id;
    cl_device_id device_id;
    cl_uint num_of_platforms = 0, num_of_devices = 0;
    cl_int err;
    char name[48];

    // Plattform ?
    if (clGetPlatformIDs(1, &platform_id, &num_of_platforms) != CL_SUCCESS) {
        printf("Unable to get platform id\n"); return 1;
    }
    // GPU Device ?
    if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1,
                       &device_id, &num_of_devices) != CL_SUCCESS) {
        printf("Unable to get device id\n"); return 1;
    }
    // Device Information ausgeben
    clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
    printf("Device: %s\n", name);
    // ...
}
```

# 1.1 Get Platform ID

- Abfragen der Anzahl verfügbarer Plattformen und ihrer ID
- **Beispiel:**

```
clGetPlatformIDs(1, &platform_id, &num_of_platforms);
```

- **Parameter:**

```
cl_int clGetPlatformIDs(  
    cl_uint      num_entries,  
    cl_platform_id *platforms,  
    cl_uint      *num_platforms)
```

- **Rückgabe:**

```
// Error Codes aus cl.h  
#define CL_SUCCESS 0  
#define CL_DEVICE_NOT_FOUND -1  
#define CL_DEVICE_NOT_AVAILABLE -2  
...
```

# 1.2 Get Device ID

- Abfragen der Anzahl verfügbarer Geräte und ihrer ID

- **Beispiel:**

```
clGetDeviceIDs (platform_id, CL_DEVICE_TYPE_GPU, 1,  
                 &device_id, &num_of_devices);
```

- **Parameter:**

```
cl_int clGetDeviceIDs (  
    cl_platform_id platform,  
    cl_device_type device_type,  
    cl_uint      num_entries,  
    cl_device_id *devices,  
    size_uint    *num_devices)
```

- **Optionen:**

```
CL_DEVICE_TYPE_ALL, CL_DEVICE_TYPE_CPU,  
CL_DEVICE_TYPE_ACCELERATOR, ...
```

# 1.3 Get Device Info

- Informationen über Geräte abfragen

- **Beispiel:**

```
clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name),  
                name, NULL);
```

- **Parameter:**

```
cl_int clGetDeviceInfo(  
    cl_device_id    device,  
    cl_device_info  param_name,  
    size_t          param_value_size,  
    void*           *param_value,  
    size_t          param_value_size_ret)
```

- **Optionen:**

```
CL_DEVICE_TYPE, CL_DEVICE_VENDOR_ID, CL_DEVICE_MAX_COMPUTE_UNITS,  
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, CL_DEVICE_MAX_WORK_ITEM_SIZES,  
CL_DEVICE_MAX_WORK_GROUP_SIZE, ...
```

# Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines **OpenCL-Kontextes** und Erstellen einer **Befehlswarteschlange**
3. Online-**Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von **OpenCL-Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse

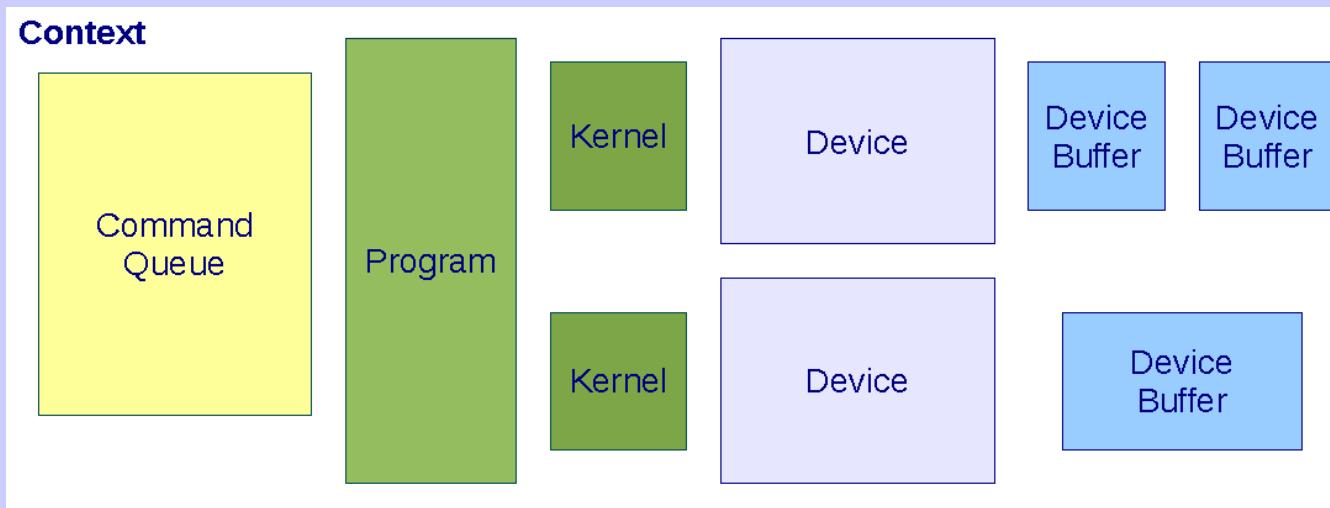


## 2. Kontext und Befehlswarteschlange

```
int main(void)
{
    cl_context context;
    cl_command_queue command_queue;

    // ... Kontext öffnen
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

    // Erzeugen einer Befehlswarteschlange (FIFO)
    command_queue = clCreateCommandQueue(context, device_id, 0, &err);
    // ...
}
```



## 2.1 Create Context

- Erzeuge einen OpenCL Kontext

- **Beispiel:**

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

- **Parameter:**

```
cl_context clCreateContext(  
    cl_context_properties *properties,  
    cl_uint                 num_devices,  
    cl_device_id            *devices,  
    void (CL_CALLBACK       *pfn_notify)  
        (const char *errinfo, const void  
         *private_info, size_t cb, void *user_data),  
    void                   *user_data,  
    cl_int                  *errcode_ret)
```

- **Optionen:**

```
properties[] = {CL_CONTEXT_PLATFORM,  
                (cl_context_properties)platform, 0};
```

## 2.2 Create Command Queue

- Eine Befehlswarteschlange erzeugen
- **Beispiel:**

```
command_queue = clCreateCommandQueue(context, device_id,  
0, &err);
```

- **Parameter:**

```
cl_command_queue clCreateCommandQueue(  
    cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret)
```

- **Optionen:**

```
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,  
CL_QUEUE_PROFILING_ENABLE
```

# Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines **OpenCL-Kontextes** und Erstellen einer **Befehlswarteschlange**
3. **Online-Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von **OpenCL-Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse



### 3. Kern kompilieren & Programm erstellen

```
const char *KernelSource =           // Quelltext des Kerns als Zeichenkette
"__kernel void hello(__global float *input, __global float *output)\n" \
"{\n    size_t i = get_global_id(0);\n    output[i] = input[i] * input[i];\n}\n\n";\n\nint main(void)\n{\n    cl_program program;\n    cl_kernel kernel;\n\n    // ... Erzeuge online ein Programm vom Quellcode des Kerns\n    program = clCreateProgramWithSource(context, 1, (const char **)\n   &KernelSource, NULL, &err);\n    if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {\n        printf("Error building program\n"); return 1;\n    }\n    kernel = clCreateKernel(program, "hello", &err); // Waehle Funktion im Kern\n    // ...
```

# 3.1 Create Program With Source

- Erstelle ein Programm

- Beispiel:

```
program = clCreateProgramWithSource(context, 1,  
                                (const char **) &KernelSource, NULL, &err);
```

- Parameter:

```
cl_program clCreateProgramWithSource (  
    cl_context      context,  
    cl_uint         count,  
    const char **strings,  
    const size_t *lengths, // (NULL if \0 terminated)  
    cl_int          *errcode_ret);
```

- Alternativen:

```
clCreateProgramWithBinary()
```

## 3.2 Build Program

- Kompiliere und Linke den Kernel-Quelltext

- **Beispiel:**

```
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

- **Parameter:**

```
cl_int clBuildProgram(  
    cl_program program,  
    cl_uint num_devices,  
    const cl_device_id *device_list,  
    const char *options,  
    void (CL_CALLBACK *pfn_notify)(  
        cl_program program, void *user_data),  
    void *user_data)
```

## 3.3 Create Kernel

- Definiere den Kernel Einsprungspunkt

- **Beispiel:**

```
kernel = clCreateKernel(program, "hello", &err);
```

- **Parameter:**

```
cl_kernel clCreateKernel(  
    cl_program program,  
    const char *kernel_name,  
    cl_int     *errcode_ret)
```

# Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines **OpenCL-Kontextes** und Erstellen einer **Befehlswarteschlange**
3. **Online-Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von **OpenCL-Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-Ausführung und Sammeln der Ergebnisse



# 4. Speicher für Ein- und Ausgabe zuweisen

```
#define DATA_SIZE 10                                // Anzahl der Daten
#define MEM_SIZE  DATA_SIZE*sizeof(float)    // Groesse der Daten im Speicher

int main(void)
{
    cl_mem input, output;
    float data[DATA_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // ... Erzeuge Puffer für Ein- und Ausgabe
    input = clCreateBuffer(context, CL_MEM_READ_ONLY, MEM_SIZE, NULL, &err);
    output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, MEM_SIZE, NULL, &err);

    // Kopiere zusammenhängende Daten aus 'data' in den Eingabe-Puffer 'input'
    clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);

    // Definiere die Reihenfolge der Argumente des Kerns: hello(input, output)
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
    // ...
```

## 4.1 Create Buffer

- Erzeugen eines OpenCL Puffers

- **Beispiel:**

```
input = clCreateBuffer(context, CL_MEM_READ_ONLY, MEM_SIZE,  
                      NULL, &err);
```

- **Parameter:**

```
cl_mem clCreateBuffer(  
    cl_context    context,  
    cl_mem_flags  flags,  
    size_t        size,  
    void          *host_ptr,  
    cl_int         *errcode_ret)
```

- **Optionen:**

```
CL_MEM_READ_WRITE, CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY,  
CL_MEM_USE_HOST_PTR, CL_MEM_COPY_HOST_PTR, ...
```

## 4.2 Enqueue Write Buffer

- Kopieren von Hauptspeicher in einen Puffer
- Beispiel:

```
clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0,  
                     MEM_SIZE, data, 0, NULL, NULL);
```

- Parameter:

```
cl_int clEnqueueWriteBuffer(  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t size,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

## 4.3 Set Kernel Arg

- Definiere Reihenfolge der Kernel Argumente

- **Beispiel**

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
```

- **Parameter**

```
cl_int clSetKernelArg(  
    cl_kernel    kernel,  
    cl_uint      arg_index,  
    size_t       arg_size,  
    const void  *arg_value)
```

# Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines **OpenCL-Kontextes** und Erstellen einer **Befehlswarteschlange**
3. **Online-Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von **OpenCL-Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse



# 5. Ausführung und Ergebnisse sammeln

```
#define DATA_SIZE 10                                // Anzahl der Daten
#define MEM_SIZE  DATA_SIZE*sizeof(float) // Groesse der Daten im Speicher

int main(void)
{
    size_t global[1] = {DATA_SIZE};
    float results[DATA_SIZE] = {0};

    // ... Einreihen des Kerns in die Befehlswarteschlange und Aufteilungsbereich angeben
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, NULL, 0, NULL, NULL);
    // Auf die Beendigung der Operation warten
    clFinish(command_queue);

    // Kopiere die Ergebnisse vom Ausgabe-Puffer 'output' in das Ergebnisfeld 'results'
    clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, MEM_SIZE, results, 0, NULL, NULL);
    // ...
}
```

# 5.1 Enqueue NDRange Kernel

- Bestimme Topologie (NDRange) und führe Kernel aus
- **Beispiel:**

```
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,  
                      global, NULL, 0, NULL, NULL);
```

- **Parameter:**

```
cl_int clEnqueueNDRangeKernel(  
    cl_command_queue command_queue,  
    cl_kernel kernel,  
    cl_uint work_dim,  
    const size_t *global_work_offset,  
    const size_t *global_work_size,  
    const size_t *local_work_size,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event event)
```

## 5.2 Enqueue Read Buffer

- Lesen vom Puffer in den Hauptspeicher
- Beispiel

```
clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0,  
                    MEM_SIZE, results, 0, NULL, NULL);
```

- Parameter

```
cl_int clEnqueueReadBuffer(  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_read,  
    size_t offset,  
    size_t size,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

# OpenCL Ressourcen wieder freigeben

```
int main(void)
{
    // ... Aufräumen der OpenCL Ressourcen
    clReleaseMemObject(input);
    clReleaseMemObject(output);

    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(command_queue);
    clReleaseContext(context);

    printf("Resulting Squared: ");
    for (int i=0; i<DATA_SIZE; i++) {
        printf("%.f ", results[i]);
    }
    printf("\n");

    return 0;
}
```

# Parallele und Verteilte Systeme

**Grundlagen zur Programmierung von Mehrkern-Systemen,  
verteilter Berechnung und massiver Parallelität**



Vorlesung WiSe 2020/21

(Unterlagen nur für den internen Gebrauch!)

# Übersicht

- **Massiv parallele Programmierung**
  - Praktische Hinweise
    - Geräteinformationen besser abfragen
    - Parameterübergabe
    - Zeitmessung
    - OpenCL Compilermeldungen
    - Watchdog Timer unter Windows
  - Optimierung der Matrix-Multiplikation
    - Mehrdimensionale Gitter (NDGRID)

# Aufruf-Beispiel: Hello.cpp

```
#include "CL/cl.h"

int main(void)
{
    // ...
    clGetPlatformIDs(1, &platform_id, &num_of_platforms);
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_of_devices);
    clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    command_queue = clCreateCommandQueue(context, device_id, 0, &err);
    program = clCreateProgramWithSource(context, 1, (const char **)&KernelSource, NULL, &err);
    clBuildProgram(program, 0, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "hello", &err);
    input = clCreateBuffer(context, CL_MEM_READ_ONLY, MEM_SIZE, NULL, &err);
    output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, MEM_SIZE, NULL, &err);
    clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, 0, NULL, NULL);
    clFinish(command_queue);
    clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, MEM_SIZE, results, 0, NULL, NULL);
    // ...
}
```



# Praktische Hinweise

# Problem: Initialisierung der Geräte

```
int main(void)
{
    cl_platform_id platform_id;
    cl_device_id device_id;
    cl_uint pnum = 0, dnum = 0;

    // Plattform ?
    if (clGetPlatformIDs(1, &platform_id, &pnum) != CL_SUCCESS) {
        printf("Unable to get platform id\n"); return 1;
    }
    // GPU Device ?
    if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &dnum) != CL_SUCCESS) {
        printf("Unable to get device id\n"); return 1;
    }
    // ...
}
```

# Initialisierung mehrerer Geräte

```
#include <string.h>   // für Zeichenkettenvergleich

int main(void)
{
    cl_uint          pnum = 0, pid = 0, dnum = 0, i;
    cl_platform_id *platforms = NULL;                      // Leeres Feld für Plattformen
    char            pname[1024];                            // Name der Plattform

    if (clGetPlatformIDs(0, NULL, &pnum) != CL_SUCCESS) {           // Anzahl verfügbarer Plattformen
        printf("No platform found.\n"); return 0;
    }
    platforms = (cl_platform_id *)malloc(pnum * sizeof(cl_platform_id)); // Speicher für das Feld
    if (!platforms) {
        printf("Can't allocate platform memory."); exit(1);
    }
    if (clGetPlatformIDs(pnum, platforms, NULL) != CL_SUCCESS) {
        printf("No platform found.\n"); return 0;
    }
    for (i=0; i<pnum; i++) {
        if (clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, sizeof(pname), pname, NULL) != CL_SUCCESS) {
            printf("Could not get platform information.\n"); return 0;
        }
        if (strstr(pname, "NVIDIA") != NULL) { // Vergleich ob Zeichenkette INTEL, IBM, AMD, ATI enthalten
            pid = i; break;                  // ID merken
        }
    }
    if (clGetDeviceIDs(platforms[pid], CL_DEVICE_TYPE_GPU, 0, NULL, &dnum) != CL_SUCCESS) { // Deviceanzahl
        printf("Could not get device.\n"); return 0;
    }
    // ...
}
```

# Geräteinformation abfragen

```
cl_uint i;
cl_ulong l, a[3] = {0, 0, 0};
char name[48];

clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
printf("Device           : %s\n", name);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(i), &i, NULL);
printf("Compute units   : %d\n", i);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(i), &i, NULL);
printf("Work item dim   : %d\n", i);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(a), &a, NULL);
printf("Work item sizes: %d / %d / %d\n", a[0], a[1], a[2]);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(l), &l, NULL);
printf("Work group size: %d\n", l);
clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(l), &l, NULL);
printf("Global mem size: %d MByte\n", l/1024/1024);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_MEM_ALLOC_SIZE, sizeof(l), &l, NULL);
printf("Max alloc size : %d MByte\n", l/1024/1024);
clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_SIZE, sizeof(l), &l, NULL);
printf("Local mem size : %d KByte\n", l/1024);
```

# Beispiel: Geräteinformation

- **Kapazitäten**

```
Device           : GeForce GTX 260
Compute units   : 24
Work item dim   : 3
Work item sizes: 512 / 512 / 64
Work group size: 512
Global mem size: 1792 MByte
Max alloc size : 448 MByte
Local mem size : 16 KByte
```

- **Abschätzung**

Matrixgröße mit Fließkommazahlen einfacher Genauigkeit, `sizeof(float)` = 4 Bytes:

$$10.000 \times 10.000 \times 4 = 400.000.000 = \mathbf{382 \text{ MByte}}$$

# Einlesen des Kernel-Quellcodes

```
#define MAX_SOURCE_SIZE (0x100000) // 1 MB

int main(void)
{
    FILE *fp;
    const char *FileName = "kernel.cl";
    char *KernelSource;

    fp = fopen(FileName, "r");
    if (!fp) {
        printf("Can't open kernel source: %s", FileName); exit(1);
    }
    KernelSource = (char *)malloc(MAX_SOURCE_SIZE);
    if (!KernelSource) {
        printf("Can't allocate kernel"); fclose(fp); exit(1);
    }
    fread(KernelSource, 1, MAX_SOURCE_SIZE, fp);
    fclose(fp);
    // ...
}
```

# Dimensionen als Parameter?

```
const char *KernelSource =
"#define DIM 1000                                     // Groesse der Matrix \n"
"__kernel void vecmult(__global float *A, __global float *B, __global float *C) { \n"
"    int i;   \n"
"    float A1[DIM];                                \n"
"    for (i = 0; i < DIM; i++)                      \n"
"        //..."                                     \n"
```

```
const char *KernelSource =
"__kernel void vecmult(__global float *A, __global float *B, __global float *C,      \n"
"                      const int dim) {           // fuer dim <= 1000! \n"
"    int i;   \n"
"    float A1[1000];                               \n"
"    for (i = 0; i < dim; i++)                      \n"
"        //..."                                     \n"
```

```
int main(void)
{
    cl_int dim = DIM;
    // ...
    clSetKernelArg(kernel, 3, sizeof(cl_int), &dim);
```

# Zeitmessung für den Kern

```
#include <omp.h>

double start, end;
start = omp_get_wtime();
//...
end = omp_get_wtime();
printf("time = %f sec\n", end - start);
```

```
cl_event event;
cl_ulong start, end;

command_queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &err);
// ...
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global, &local, 0, NULL, &event);
clFinish(command_queue);
// ...
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(start), &start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(end), &end, NULL);
printf("time = %.1f ms\n", ((end - start) / 1000000.0));
```

# OpenCL Compilermeldungen anzeigen

```
if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {  
    char *log;  
    size_t size;  
                                // 1. Länge des Logbuches?  
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &size);  
    log = (char *)malloc(size+1);  
    if (log) {  
        // 2. Hole das Logbuch ab  
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, size, log, NULL);  
        log[size] = '\0';  
        printf("%s", log);  
        free(log);  
    }  
    return 1;  
}
```

```
:9:39: error: expected ';' after expression  
        sum += A[i*n+k] * B[k*n+j]:  
                           ^  
                           ;  
:9:39: error: expected expression
```

# Speicherverwaltung vereinfachen

```
#define DATA_SIZE 10                                // Anzahl der Daten
#define MEM_SIZE  DATA_SIZE*sizeof(float)    // Groesse der Daten im Speicher

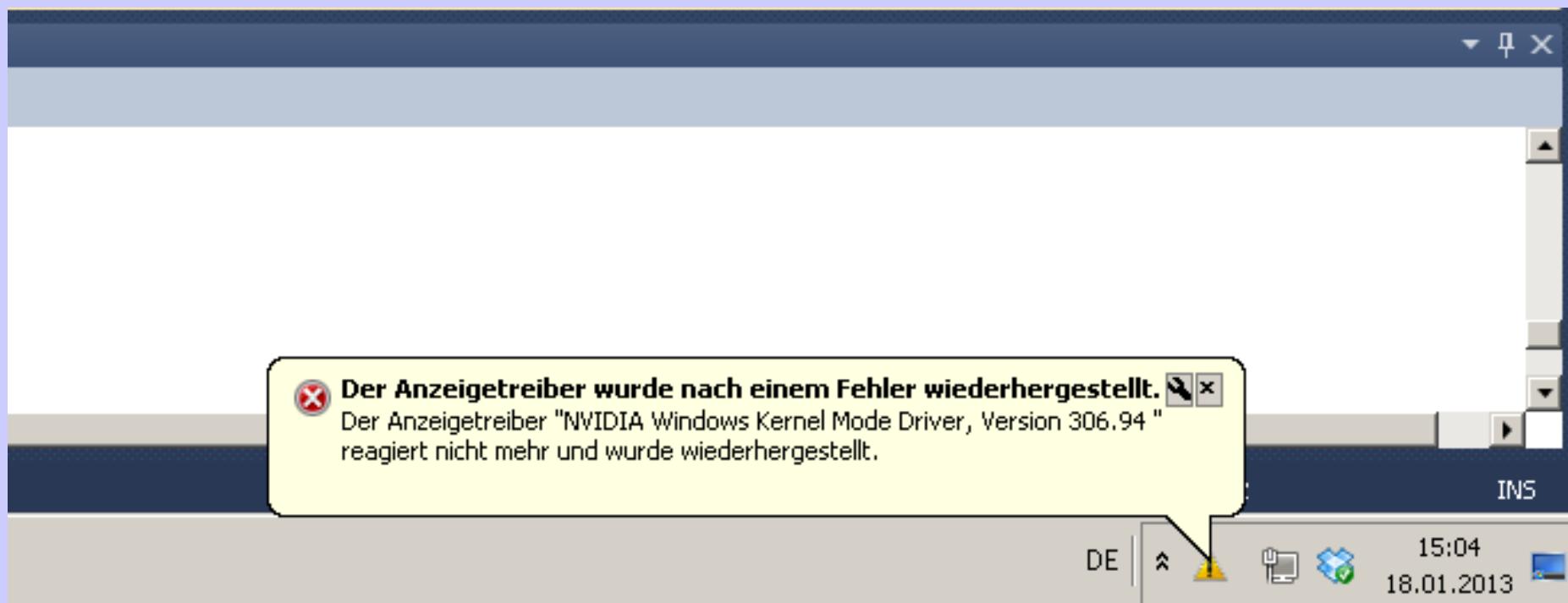
int main(void)
{
    cl_mem input;
    float data[DATA_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    // ... Erzeuge Puffer für Ein- und Ausgabe
    input = clCreateBuffer(context, CL_MEM_READ_ONLY, MEM_SIZE, NULL, &err);

    // Kopiere zusammenhängende Daten aus 'data' in den Eingabe-Puffer 'input'
    clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);

    // Reserviere globalen Speicher und kopiere Daten vom Host in einem Schritt
    input = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                          MEM_SIZE, data, &err);
```

# Absturz des NVIDIA Treibers



# NVIDIA Watchdog unter Windows

- **Problem:** Timeout Detection and Recovery (TDR) im Windows Display Driver Model (WDDM)
- **Idee:** Wenn der Grafiktreiber nicht mehr reagiert, wird er neu gestartet
- **Betroffen:** Windows + NVIDIA + angeschlossener Monitor
- **Lösung 1 (Nicht zu empfehlen, nur für Testzwecke!):**

in der Windows-Registrierungsdatenbank unter  
HKLM\System\CurrentControlSet\Control\GraphicsDrivers  
neue Einträge erstellen:

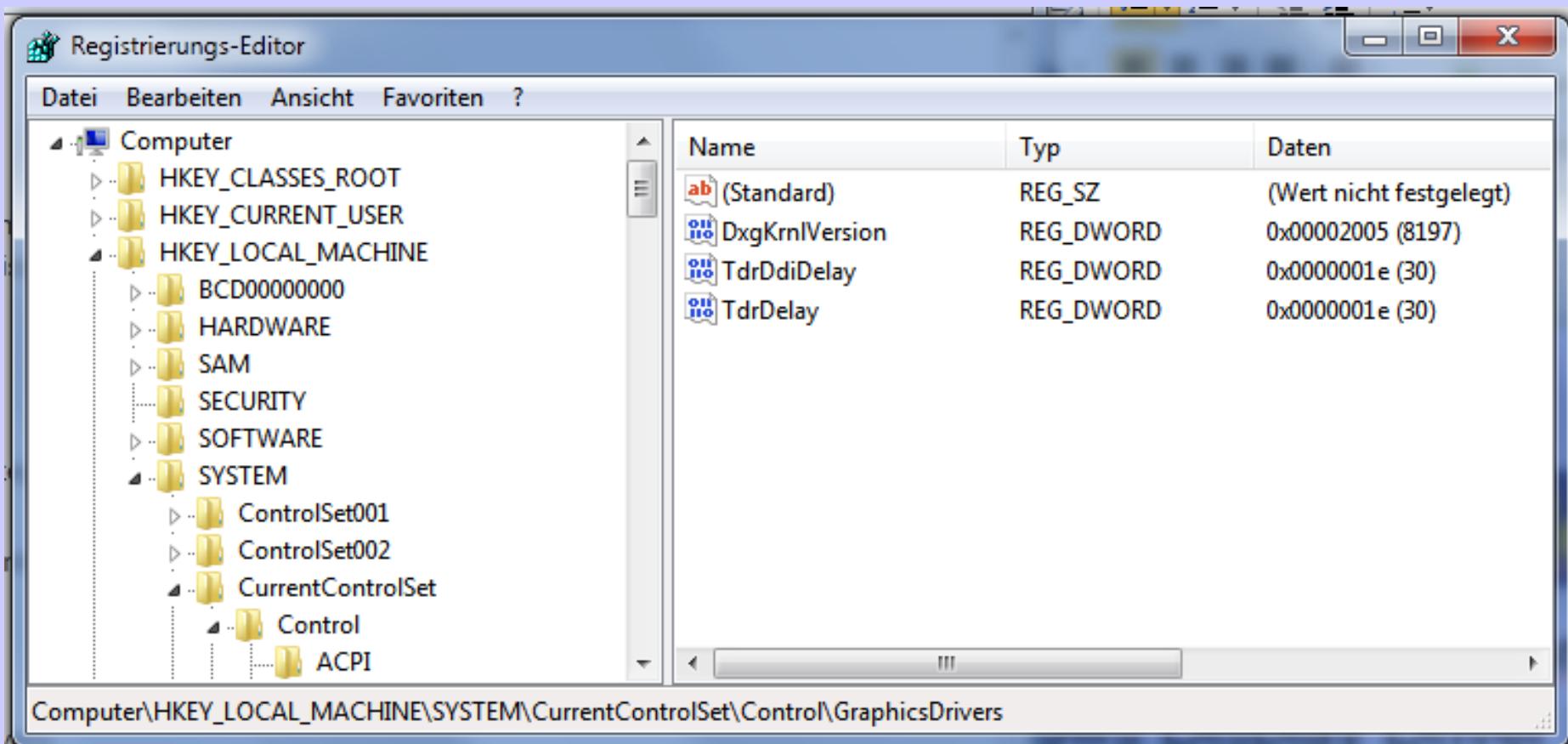
- **TdrDelay:** REG\_DWORD
- **TdrDdiDelay:** REG\_DWORD

Enthalten die Anzahl an Sekunden, die das Betriebssystem dem Grafikkartentreiber für eine Rückmeldung erlaubt.

Nach dieser Zeit wird der Treiber zurückgesetzt.

Die Standardwerte sind **2** bzw. **5**!

# Beispiel: Erhöhung auf 30 Sekunden



**Achtung:** In dieser Zeit hat man keine aktualisierte Bildschirmausgabe!

# Lösungsvorschlag

- **Lösung 2 (Empfehlung!):**

- Die langwierige Aufgabe für den Kernel in mehrere schnellere **Teilaufgaben** splitten (< 2 Sekunden)
- Alle Teilaufgaben in die Befehlswarteschlange einreihen
- Das Allokieren, Füllen und Leeren der globalen Puffer sowie das Warten auf Beendigung (`clFinish`) braucht nur **einmal** erfolgen!



# Matrix-Multiplikation mit OpenCL

# Matrix-Multiplikation (zeilenweise)

$$\begin{matrix} P_1 & P_2 & P_3 & P_1 & P_2 & P_3 & P_1 \\ \hline & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ \hline \end{matrix} = \begin{matrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ \hline \end{matrix} \cdot \begin{matrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ \hline \end{matrix}$$

**C**                    **A**                    **B**

# Version 1: C zeilenweise

```
const char *KernelSource =
"#define DIM 1000                                     // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) \n"
"{
"    int i, j, k;                                \n"
"    i = get_global_id(0);                         \n"
"    for (j = 0; j < DIM; j++)                      \n"
"        for (k = 0; k < DIM; k++)                  \n"
"            C[i*DIM+j] += A[i*DIM+k] * B[k*DIM+j]; \n"
"}\n";
```

# Änderungen am Hauptprogramm

```
#define DIM          1000                                // Matrix Dimensionen
#define DATA_SIZE  DIM*DIM*sizeof(float)                // Matrixgröße im Speicher

void main(int argc, char **argv)
{
    // ...
    float **A, **B, **C;                                // Matrizen
    cl_mem Ap, Bp, Cp;
    size_t global[1] = {DIM};                           // Oder: size_t global = DIM;

    A = alloc_mat(DIM, DIM); init_mat(A, DIM, DIM);      // Speicher für Matrizen holen
    B = alloc_mat(DIM, DIM); init_mat(B, DIM, DIM);      // und initialisieren
    C = alloc_mat(DIM, DIM);
    // ...

    kernel = clCreateKernel(program, "matmult", &err);           // Spezifizierte Kernel
    Ap = clCreateBuffer(context, CL_MEM_READ_ONLY, DATA_SIZE, NULL, &err); // Erzeuge Puffer
    Bp = clCreateBuffer(context, CL_MEM_READ_ONLY, DATA_SIZE, NULL, &err);
    Cp = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, &err);
    clEnqueueWriteBuffer(command_queue, Ap, CL_TRUE, 0, DATA_SIZE, A[0], 0, NULL, NULL);
    clEnqueueWriteBuffer(command_queue, Bp, CL_TRUE, 0, DATA_SIZE, B[0], 0, NULL, NULL);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &Ap);        // Setzte die Argumentliste für Kernel
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &Bp);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &Cp);
    // ...

    clEnqueueReadBuffer(command_queue, Cp, CL_TRUE, 0, DATA_SIZE, C[0], 0, NULL, NULL);
    // ...
}
```



# Optimierungspotential?

# Version 2: C zeilenweise, schreibend

```
const char *KernelSource =  
"#define DIM 1000                                     // Groesse der Matrix \n"  
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"  
"    int i, j, k;   \n"  
"    float sum;   // Private Variable für Zwischenergebnisse \n"  
"    i = get_global_id(0);                                \n"  
"    for (j = 0; j < DIM; j++) {                          \n"  
"        sum = 0.0;                                       \n"  
"        for (k = 0; k < DIM; k++)                         \n"  
"            sum += A[i*DIM+k] * B[k*DIM+j];                \n"  
"        C[i*DIM+j] = sum;                                \n"  
"    }   \n"  
"}
```

```
void main(int argc, char **argv)  
{  
// ...  
Cp = clCreateBuffer(context, CL_MEM_WRITE_ONLY, DATA_SIZE, NULL, &err);
```

# Matrix-Multiplikation (elementweise)

# Version 3: C elementweise

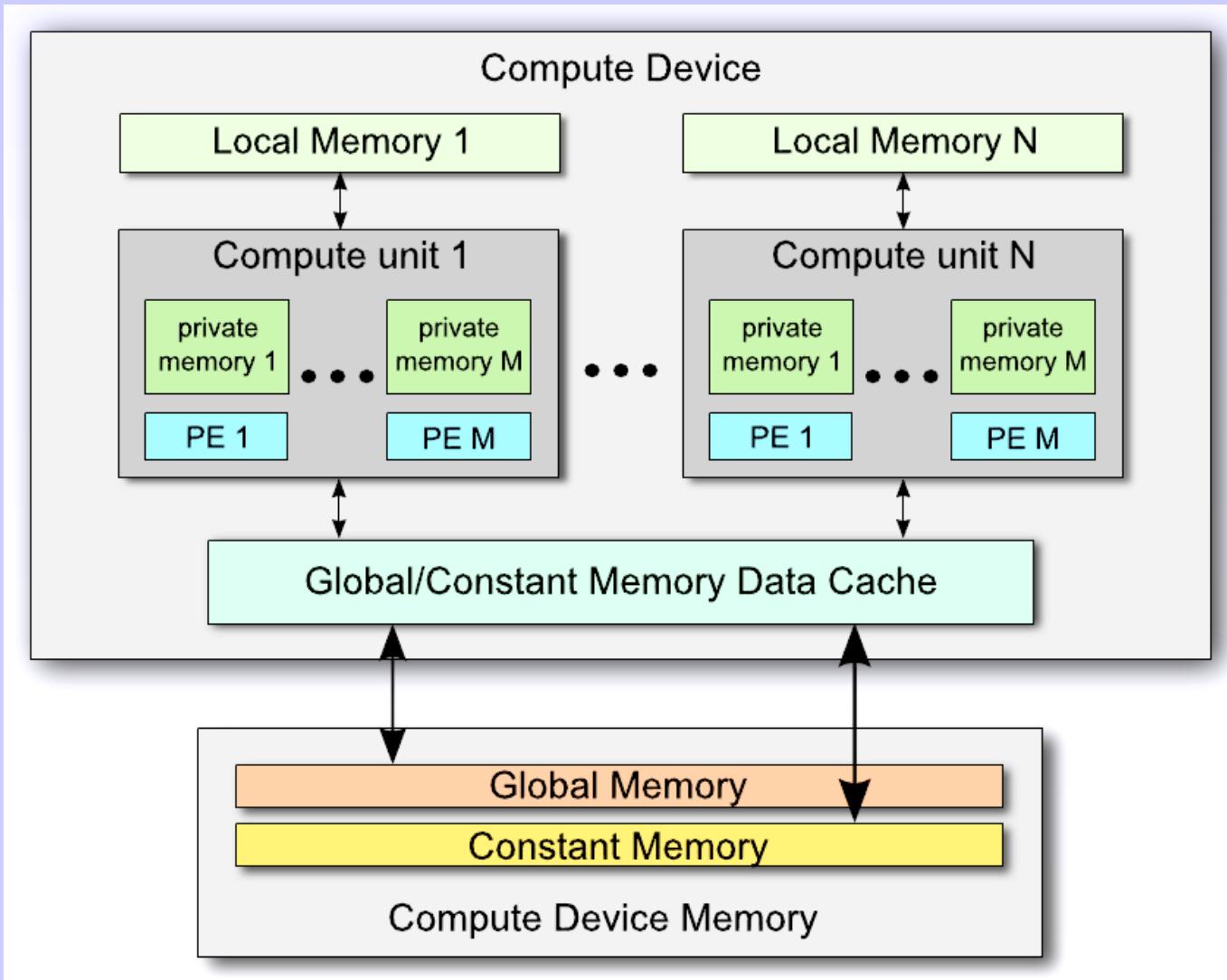
```
const char *KernelSource =
"#define DIM 1000                                     // Size of matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"
"    int i, j, k;                                \n"
"    float sum = 0.0;                            \n"
"    i = get_global_id(0);                      \n"
"    j = get_global_id(1);                  \n"
"    for (k = 0; k < DIM; k++)                   \n"
"        sum += A[i*DIM+k] * B[k*DIM+j];      \n"
"    C[i*DIM+j] = sum;                      \n"
"}  \n"
"\n";
```

```
void main(int argc, char **argv)
{
    size_t global[2] = {DIM, DIM};
    // ...
    clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
```

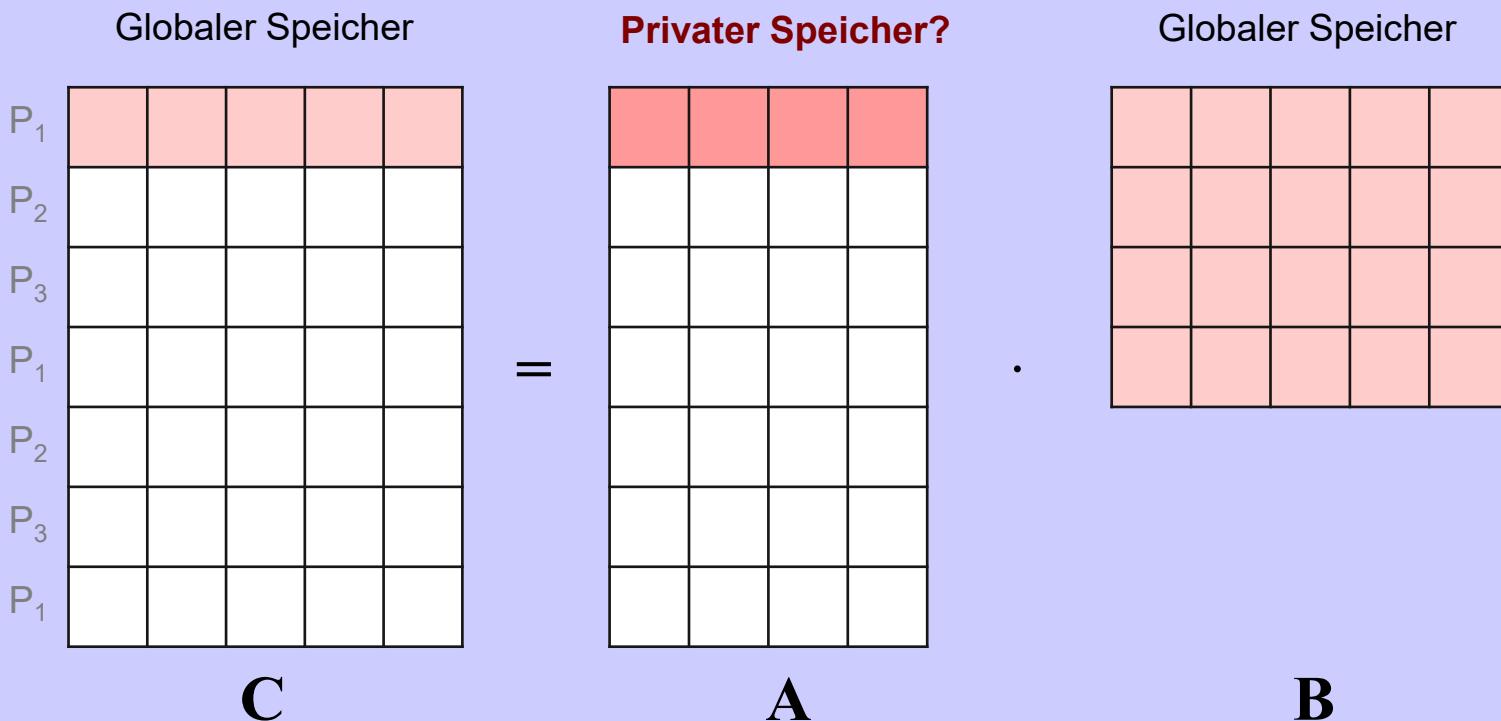
# Version 4: Schleifen tauschen?

```
const char *KernelSource =  
"#define DIM 1000                                     // Size of matrix \n"  
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"  
"    int i, j, k;  
"    float sum = 0.0;  
"    j = get_global_id(0);  
"    i = get_global_id(1);  
"    for (k = 0; k < DIM; k++)  
"        sum += A[i*DIM+k] * B[k*DIM+j];  
"    C[i*DIM+j] = sum;  
"}  
\n";
```

# Speicheroptimierung?



# Matrix-Multiplikation (zeilenweise)

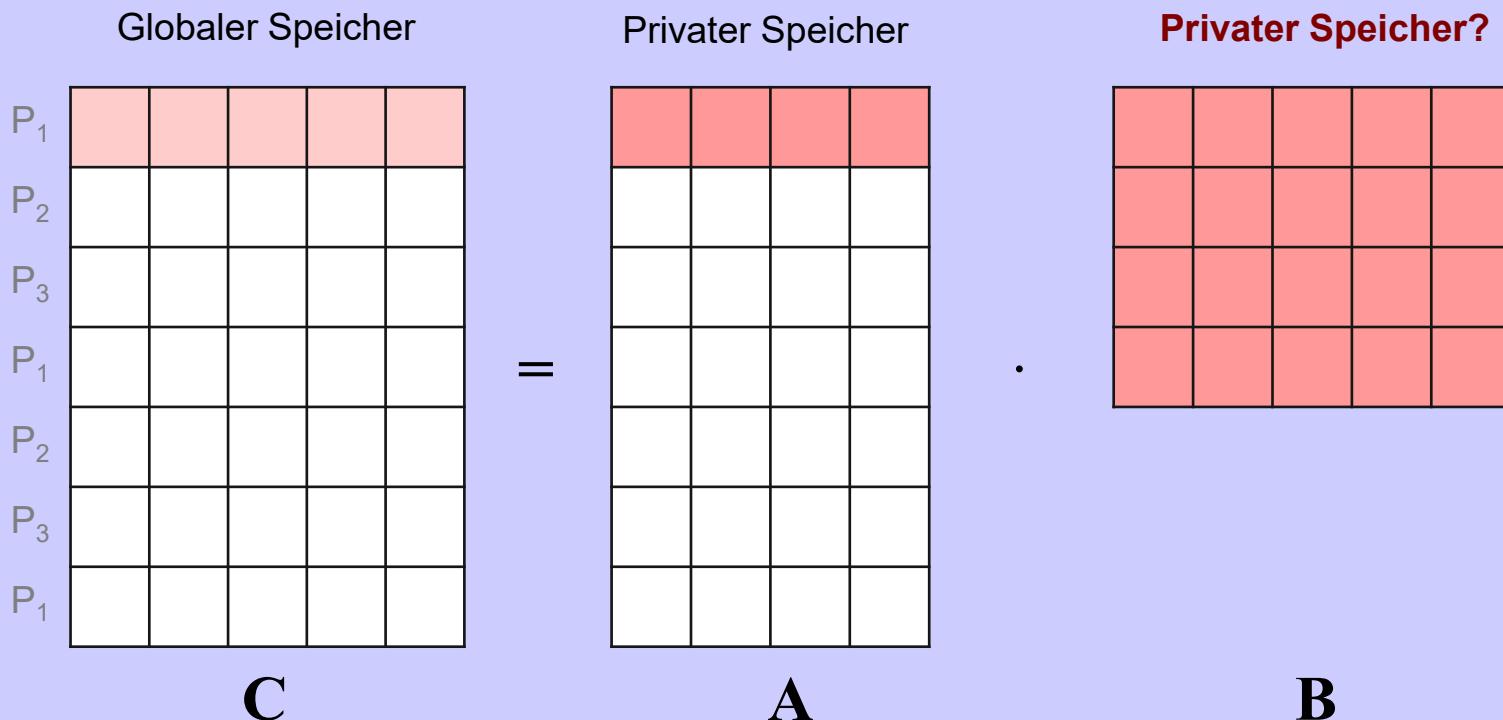


# Version 5: C zeilenweise, A privat

```
const char *KernelSource =  
"#define DIM 1000                                     // Groesse der Matrix \n"  
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"  
"    int k, j, i = get_global_id(0);                  \n"  
"    float Al[DIM], sum;                            // Privater Zeilenpuffer \n"  
"    for (k = 0; k < DIM; k++)                      \n"  
"        Al[k] = A[i*DIM+k];                         // Zeile i von A in den Puffer kopieren \n"  
"    for (j = 0; j < DIM; j++) {                      \n"  
"        sum = 0.0;                                  \n"  
"        for (k = 0; k < DIM; k++)                   \n"  
"            sum += Al[k] * B[k*DIM+j];              \n"  
"        C[i*DIM+j] = sum;                          \n"  
"    }  \n"  
" }  \n"  
"\n";
```

```
void main(int argc, char **argv)  
{  
    size_t global[1] = {DIM};  
    // ...  
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, NULL, 0, NULL, NULL);
```

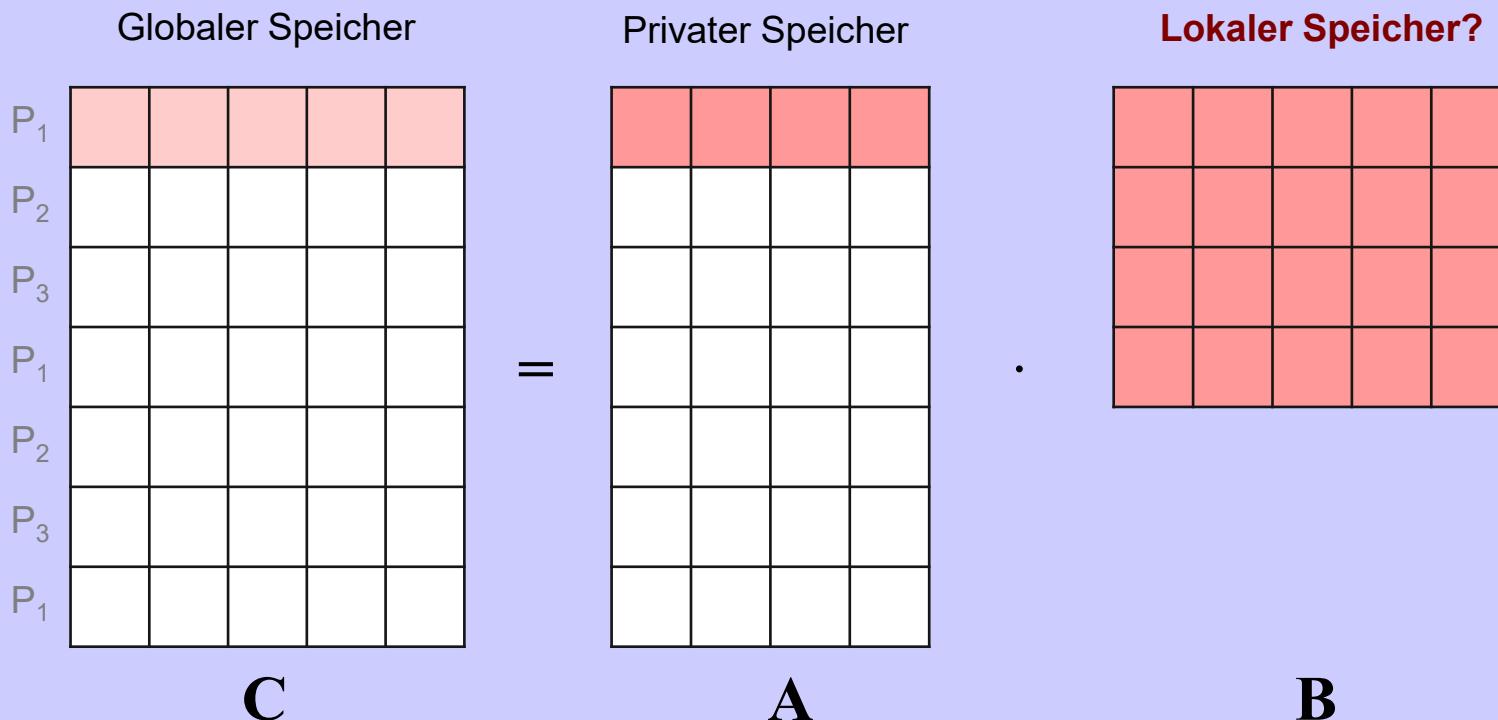
# Matrix-Multiplikation (zeilenweise)



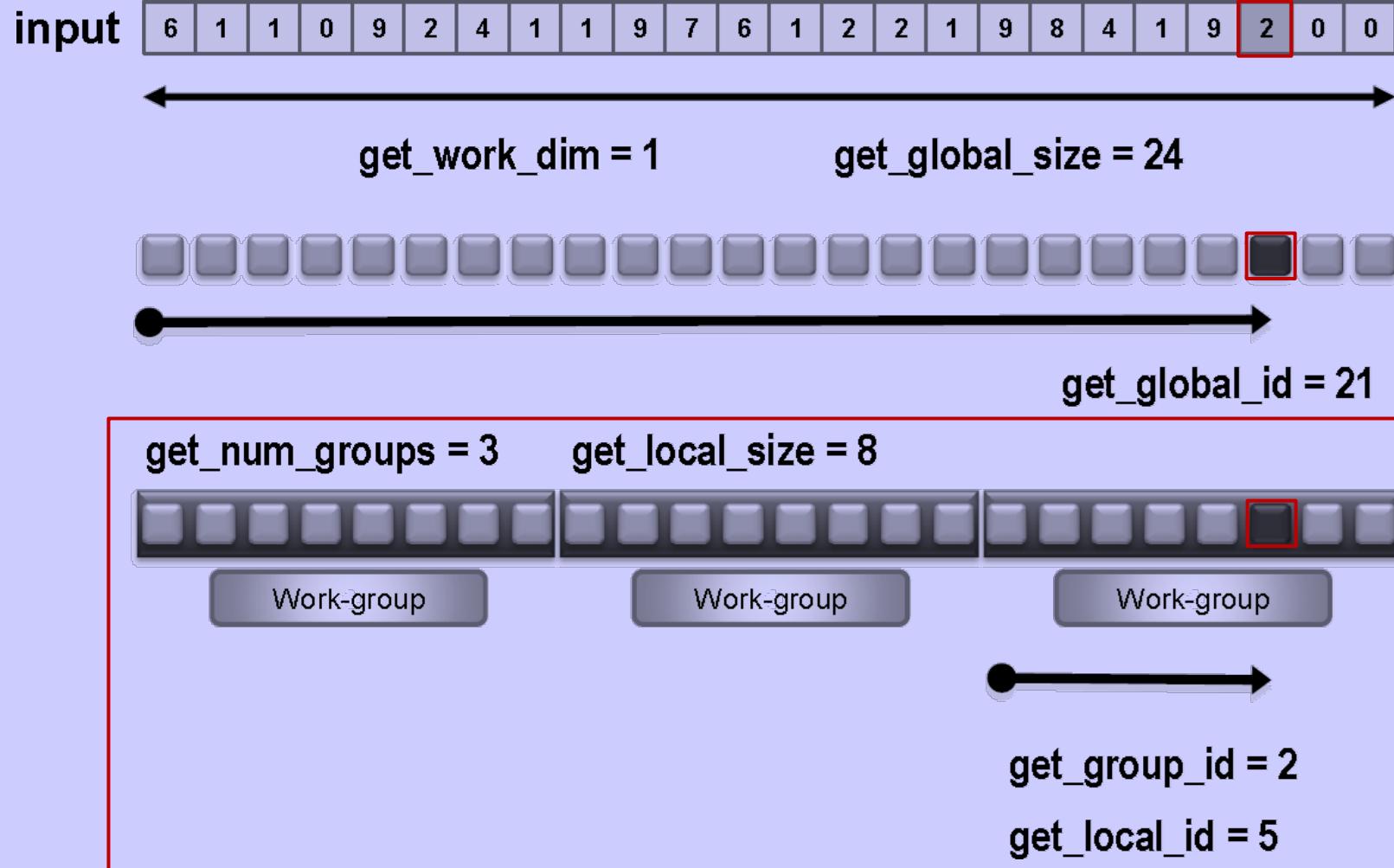
# Versuch 1: C zeilenweise, A + B privat

```
const char *KernelSource =  
"#define DIM 1000                                     // Groesse der Matrix \n"  
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"  
"    int k, j, i = get_global_id(0);  
"    float Al[DIM], Bl[DIM], sum;  
"    for (k = 0; k < DIM; k++)  
"        Al[k] = A[i*DIM+k];  
"    for (j = 0; j < DIM; j++) {  
"        for (k = 0; k < DIM; k++)  
"            Bl[k] = B[k*DIM+j];  
"        sum = 0.0;  
"        for (k = 0; k < DIM; k++)  
"            sum += Al[k] * Bl[k];  
"        C[i*DIM+j] = sum;  
"    }  
"\n";
```

# Matrix-Multiplikation (zeilenweise)



# Lokaler Speicher und Arbeitsgruppen



# Versuch 2: C zeilenweise, A priv. B lokal

```
const char *KernelSource =  
"#define DIM 1000                                     // Groesse der Matrix \n"  
"__kernel void matmult(__global float *A, __global float *B, __global float *C,    \n"  
"                      __local float *Bl) {           \n"  
"    int k, j, i = get_global_id(0);                \n"  
"    float Al[DIM], sum;                          \n"  
"    int il = get_local_id(0);                     // n work-items teilen sich das Kopieren \n"  
"    int nl = get_local_size(0);                   \n"  
"    for (k = 0; k < DIM; k++)                  \n"  
"        Al[k] = A[i*DIM+k];                    \n"  
"    for (j = 0; j < DIM; j++) {                 \n"  
"        for (k = il; k < DIM; k += nl)          // nur jede n-te Zeile von B kopieren \n"  
"            Bl[k] = B[k*DIM+j];                \n"  
"        sum = 0.0;                            \n"  
"        for (k = 0; k < DIM; k++)              \n"  
"            sum += Al[k] * Bl[k];            \n"  
"        C[i*DIM+j] = sum;                   \n"  
"    }   \n"  
"}   \n"\n";
```

# Versuch 2: C zeilenweise, A priv. B lokal

```
const char *KernelSource =  
"#define DIM 1000                                     // Groesse der Matrix \n"  
"__kernel void matmult(__global float *A, __global float *B, __global float *C,    \n"  
"                      __local float *Bl) {           \n"  
"      // ...   \n"  
" }   \n"  
"\n";
```

**Explizite Definition der Work-group Größe (ganzzahlig teilbar):**

```
void main(int argc, char **argv)  
{  
    size_t global[1] = {DIM}, local[1];  
    // ...  
    clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(size_t), local, NULL);  
    local[0] = ggt(global[0], local[0]); // Groesster gemeinsamer Teiler ggt(1000, 24) = 8  
    // ...  
    clSetKernelArg(kernel, 3, sizeof(float)*DIM, NULL);  
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, local, 0, NULL, NULL);
```

# Größter gemeinsamer Teiler

```
int ggt(int x, int y)
{
    int z;

    while (y) {
        z = x % y;
        x = y;
        y = z;
    }
    return x;
}
```

## Beispiel:

$$\begin{array}{rccccc} & x & \quad y & \quad z \\ 1000 & / & 24 & = & 41 & \text{Rest } 16 \\ & \swarrow & \uparrow & & & \\ 24 & / & 16 & = & 1 & \text{Rest } 8 \\ & \swarrow & \uparrow & & & \\ 16 & / & 8 & = & 2 & \text{Rest } 0 \\ & \swarrow & & & & \end{array}$$

# Version 6: C zeilenweise, A priv. B lokal

```
const char *KernelSource =
"#define DIM 1000                                     // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C,   \n"
"                      __local float *Bl) {                                \n"
"    int k, j, i = get_global_id(0);                           \n"
"    float Al[DIM], sum;   \n"
"    int il = get_local_id(0);                                    \n"
"    int nl = get_local_size(0);                                  \n"
"    for (k = 0; k < DIM; k++)                                 \n"
"        Al[k] = A[i*DIM+k];                                    \n"
"    for (j = 0; j < DIM; j++) {                                \n"
"        for (k = il; k < DIM; k+=nl)                            \n"
"            Bl[k] = B[k*DIM+j];                                \n"
"        barrier(CLK_LOCAL_MEM_FENCE);           // Warten, dass alle B kopiert haben \n"
"        sum = 0.0;   \n"
"        for (k = 0; k < DIM; k++)                          \n"
"            sum += Al[k] * Bl[k];                           \n"
"        C[i*DIM+j] = sum;                                \n"
"    }   \n"
"}  \n"
"\n";
```

# Frohe Weihnachten & gutes neues Jahr!

