Parallelverarbeitung

WS 2015/16

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Parallelverarbeitung

WS 2015/16

0 Organisatorisches

Zu meiner Person

- Studium der Informatik an der Techn. Univ. München
  - dort 1994 promoviert, 2001 habilitiert
- Seit 2004 Prof. für Betriebssysteme und verteilte Systeme
- **Forschung:** Beobachtung, Analyse und Steuerung paralleler und verteilter Systeme
- **Mentor** für die Bachelor-Studiengänge Informatik mit Nebenfach/Vertiefung Mathematik

- **e-mail:** roland.wismueller@uni-siegen.de
- **Tel.:** 0271/740-4050
- **Büro:** H-B 8404

- **Sprechstunde:** Mo., 14:15-15:15 Uhr

Zur Fachgruppe "Betriebssysteme / verteilte Systeme"

Andreas Hoffmann

andreas.hoffmann@uni-siegen.de
0271/740-4047
H-B 8405

- Elektronische Prüfungs- und Übungssysteme an Hochschulen
- IT-Sicherheit
- Webtechnologien
- Mobile Anwendungen

Adrian Kacso

adrian.kacso@uni-siegen.de
0271/740-3966
H-B 8406

- Kommunikationsprotokolle für drahtlose Sensornetze
- Kommunikation und Koordination in verteilten Systemen
- Betriebssysteme (RT, Embedded)

Julia Dauwe

julia.dauwe@uni-siegen.de
0271/740-2967
H-B 8405

- *Context Aware Systems*
- *Bring Your Own Device (BYOD)*
- Mobile Anwendungen und Datenschutz

Alexander Kordes

alexander.kordes@uni-siegen.de
0271/740-4011
H-B 8407

- *Automotive Electronics*
- Fahrzeugnetzwerke
- Robustheit, Fehleranalyse, Fehlerdetektion

Lehrangebot

Vorlesungen/Praktika

- Rechnernetze I, 5 LP (jedes SS)
- Rechnernetze Praktikum, 5 LP (jedes WS)
- Rechnernetze II, 5 LP (jedes SS)
- Betriebssysteme I, 5 LP (jedes WS)
- Parallelverarbeitung, 5 LP (jedes WS)
- Verteilte Systeme, 5 LP (jedes SS)
  - (wird auch als Betriebssysteme II anerkannt)
- Client/Server-Programmierung, 5 LP (jedes WS)

Lehrangebot ...

Projektgruppen

- z.B. Werkzeug zur Algorithmen-Visualisierung
- z.B. Infrastruktur zum Analysieren des Android Market

Abschlussarbeiten (Bachelor, Master, Diplom)

- Themengebiete: Mobile Plattformen (iOS, Android), Sensornetze, Parallelverarbeitung, Monitoring, ...
- z.B. Statische Analyse des Informationsflusses in Android Apps

Seminare

- Themengebiete: Webtechnologien, Sensornetze, Android, ...
- Ablauf: Blockseminare
  - 30 Min. Vortrag, 5000 Worte Ausarbeitung

Zur Vorlesung

- **Vorlesung mit Praktikum:** 2+2 SWS, 5 LP (8 LP möglich)
  - Tutor: Damian Ludwig
- **Termine:**
  - Mo. 12:30 - 14:00, H-F 001 (Vorl.) bzw. H-A 4111 (Übung)
  - Do. 16:00 - 17:30, H-C 3303 (Vorl.) bzw. H-A 4111 (Übung)
- **Information, Folien und Ankündigungen:**
  - im WWW: `http://www.bs.informatik.uni-siegen.de/lehre/ws1516/pv/`
    - Folienskript (PDF) ist verfügbar, wird jedoch noch ergänzt!
    - aktualisierte Folien werden i.d.R. spätestens am Tag vor der Vorlesung bereitgestellt
  - Codebeispiele finden Sie lokal auf den Laborrechnern unter `/home/wismueller/PV`

Zur Vorlesung ...

Lernziele

- Wissen um die Grundlagen, Techniken, Methoden und Werkzeuge der parallelen Programmierung
- Grundwissen über parallele Rechnerarchitekturen
- Praktische Erfahrungen mit paralleler Programmierung
- Kenntnisse / Anwendung der wichtigsten Programmiermodelle
- Wissen um die Möglichkeiten, Schwierigkeiten und Grenzen der Parallelverarbeitung
- Fähigkeit, erfolgversprechende Parallelisierungskonzepte erkennen und auswählen zu können
- Schwerpunkt: Hochleistungsrechnen

Zur Vorlesung ...

Methodik

- Vorlesung: Grundlagen
  - theoretisches Wissen zur Parallelverarbeitung
- Praktikum: praktische Anwendung
  - praktische Einführung in die Programmierumgebungen
  - “Hands-On“ Tutorials
  - **eigenständige Programmierarbeit**
  - praktische Fertigkeiten und Erfahrungen
  - auch: Aufwerfen von Fragen
  - unterschiedliche Parallelisierungen zweier repräsentativer Probleme
    - iteratives, numerisches Verfahren
    - kombinatorisches Suchproblem

Prüfung

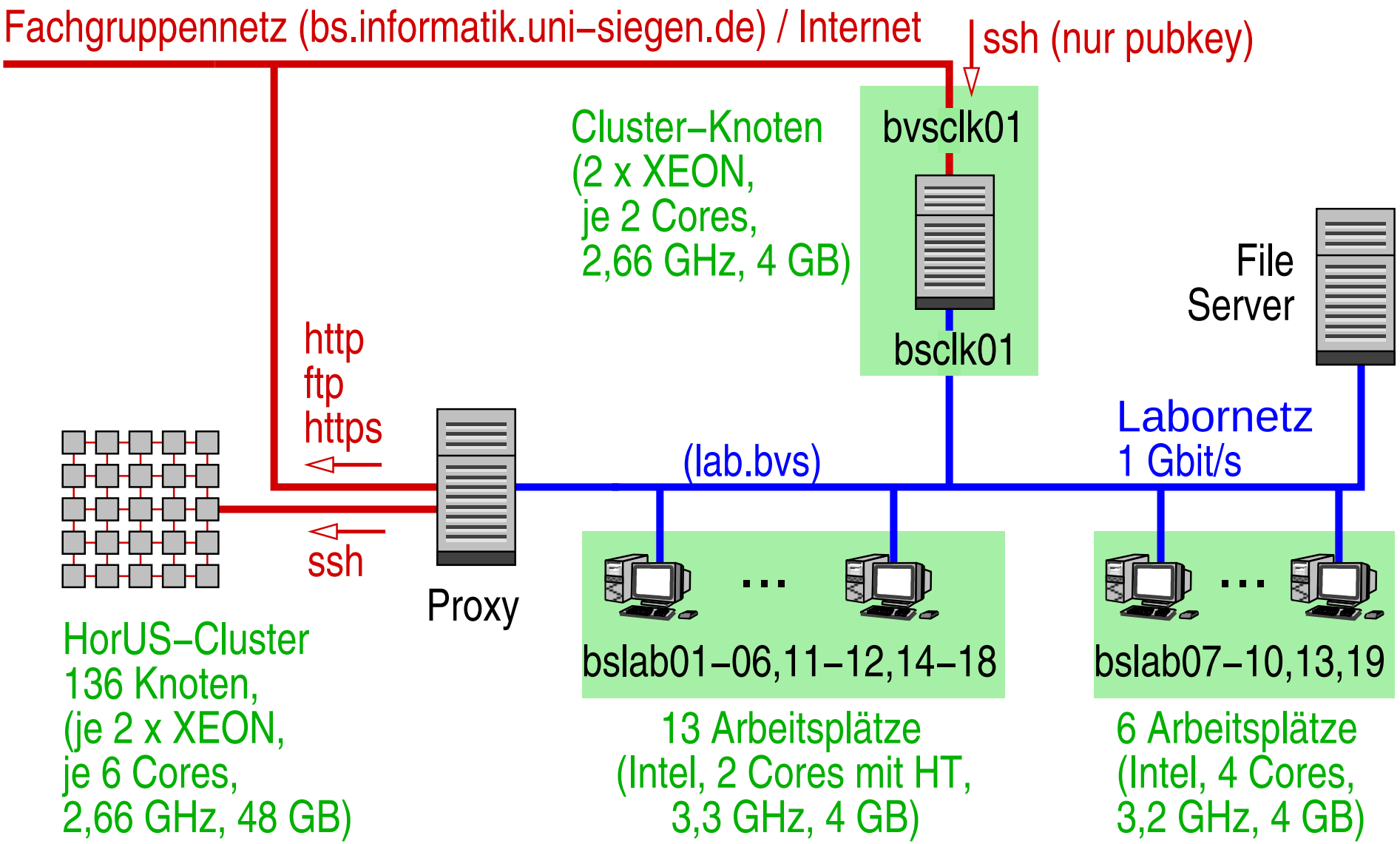
- Mündliche Prüfung (ca. 30-40 Min)
  - Stoff: Vorlesung und Praktikum!
  - Prüfung erstreckt sich auch auf die praktischen Arbeiten
- Zulassungsvoraussetzung: aktive Teilnahme am Praktikum
  - d.h. tauglicher Versuch für alle Haupt-Aufgaben
- Anmeldung:
  - Terminabsprache im Sekretariat bei Fr. Baule
    - per Email (`andrea.baule@eti.uni-siegen.de`)
    - oder persönlich (H-B 8403, nachmittags)
  - Anmeldung beim Prüfungsamt

Organisatorisches zum Praktikum

- Benutzerordnung und Kartenschlüsselantrag:
  - `http://www.bs.informatik.uni-siegen.de/lehre/ws1516/pv/`
  - Kartenschlüsselantrag bitte unterschreiben lassen und direkt bei Hr. Kiel (H-B 5413) abgeben
- **Praktikumsbeginn**: 12.11.
  - Einführung in die Rechner-Umgebung (Linux)
  - Ausgabe der Kennungen
    - Benutzerordnung im WWW beachten!
- Programmierung in C/C++

Rechnerumgebung im Labor H-A 4111

- Linux-PCs, privates IP-Netz, aber `ssh`-Zugang zum Cluster



Inhalt der Vorlesung

- Grundlagen
  - Motivation, Parallelität
  - Parallelrechner
  - Parallelisierung und Datenabhängigkeiten
  - Programmiermodelle
  - Entwurfsprozess
  - Organisationsformen paralleler Programme
  - Leistungsbetrachtungen
- Parallele Programmierung mit Speicherkopplung
  - Grundlagen
  - POSIX Threads
  - OpenMP

Inhalt der Vorlesung ...

- Parallele Programmierung mit Nachrichtenkopplung
  - Vorgehensweise
  - MPI
- Optimierungstechniken
  - Cache-Optimierungen
  - Kommunikations-Optimierung

Zeitplan der Vorlesung (vorläufig!)

Datum	**Montags-Termin**	Datum	**Donnerstags-Termin**
19.10.	V: Motivation, Parallelität	22.10.	V: Parallelrechner
26.10.	V: Programmiermodelle	29.10.	V: Entwurf, Organisation
02.11.	V: Leistung	05.11.	Ü: C-Tutorial
09.11.	V: POSIX Threads	12.11.	Ü: PThreads (Quicksort)
16.11.	V: OpenMP	19.11.	Ü: PThreads (Quicksort)
23.11.	V: OpenMP	26.11.	Ü: PThreads (Quicksort)
30.11.	Ü: OpenMP-Tutorial	03.12.	Ü: OpenMP (Jacobi)
07.12.	V: OpenMP	10.12.	Ü: OpenMP (Jacobi)
14.12.	Ü: Sokoban	17.12.	Ü: OpenMP (Jacobi)

Hellblau: freie Übungstermine

Dunkelblau: Tutorials bzw. Abgabetermine

Zeitplan der Vorlesung (vorläufig!) ...

Datum	**Montags-Termin**	Datum	**Donnerstags-Termin**
04.01.	V: MPI	07.01.	Ü: OpenMP (Jacobi)
11.01.	V: MPI	14.01.	Ü: OpenMP (Sokoban)
18.01.	V: MPI	21.01.	Ü: OpenMP (Sokoban)
25.01.	Ü: MPI-Tutorial	28.01.	Ü: OpenMP (Sokoban))
01.02.	V: Optimierung	04.02.	Ü: MPI (Jacobi)
08.02.	Ü: MPI (Jacobi)	11.02.	Ü: MPI (Jacobi)

Hellblau: freie Übungstermine

Dunkelblau: Tutorials bzw. Abgabetermine

Allgemeine Literatur

- Derzeit keine Empfehlung für ein allumfassendes Lehrbuch

- Barry Wilkinson, Michael Allen: *Parallel Programming*. internat. ed, 2. ed., Pearson Education international, 2005.
  - deckt Vorlesung i.W. ab, viele Beispiele
  - Kurzreferenzen zu MPI, PThreads, OpenMP
- A. Grama, A. Gupta, G. Karypis, V. Kumar: *Introduction to Parallel Computing*, 2nd Edition, Pearson, 2003.
  - viel zu Entwurf, Kommunikation, parallele Algorithmen
- Thomas Rauber, Gudula Rünger: *Parallele Programmierung*. 2. Auflage, Springer, 2007.
  - Architektur, Programmierung, Laufzeitanalyse, Algorithmen

Allgemeine Literatur ...

- Theo Ungerer: *Parallelrechner und parallele Programmierung*, Spektrum, Akad. Verl., 1997.
  - viel zu paralleler Hardware und Betriebssystemen
  - auch Grundlagen der Programmierung (MPI) und Compilertechniken
- Ian Foster: *Designing and Building Parallel Programs*, Addison-Wesley, 1995.
  - Entwurf paralleler Programme, Fallstudien, MPI
- Seyed Roosta: *Parallel Processing and Parallel Algorithms*, Springer, 2000.
  - i.W. Algorithmen (Entwurf, Beispiele)
  - auch: viele andere Ansätze zur parallelen Programmierung

Literatur zu speziellen Themen

- S. Hoffmann, R.Lienhart: *OpenMP*, Springer, 2008.
  - handliches Taschenbuch zu OpenMP
- W. Gropp, E. Lusk, A. Skjellum: *Using MPI*, MIT Press, 1994.
  - das Standardwerk zu MPI
- D.E. Culler, J.P. Singh: Parallel Computer Architecture - A Hardware / Software Approach. Morgan Kaufmann, 1999.
  - UMA/NUMA-Systeme, Cache-Kohärenz, Speicherkonsistenz
- Michael Wolfe: *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.
  - Vertiefung zu parallelisierenden Compilern

Parallelverarbeitung

WS 2015/16

1 Grundlagen

Inhalt

- Motivation
- Parallelität
- Parallelrechnerarchitekturen
- Parallele Programmiermodelle
- Leistung und Skalierbarkeit paralleler Programme
- Strategien zur Parallelisierung
- Organisationsformen paralleler Programme

Literatur

- Ungerer
- Grama, Gupta, Karypis, Kumar

Was ist Parallelität?

- Allgemein:
  - mehr als eine Aktion zu einer Zeit ausführen
- Speziell in Bezug auf Programmausführung:
  - zu einem Zeitpunkt wird
    - mehr als ein Befehl ausgeführt

    und / oder
    - mehr als ein Paar von Operanden verknüpft
- Ziel: schnellere Lösung der zu bearbeitenden Aufgabe
- Probleme: Aufteilung der Aufgabe, Overhead für Koordination

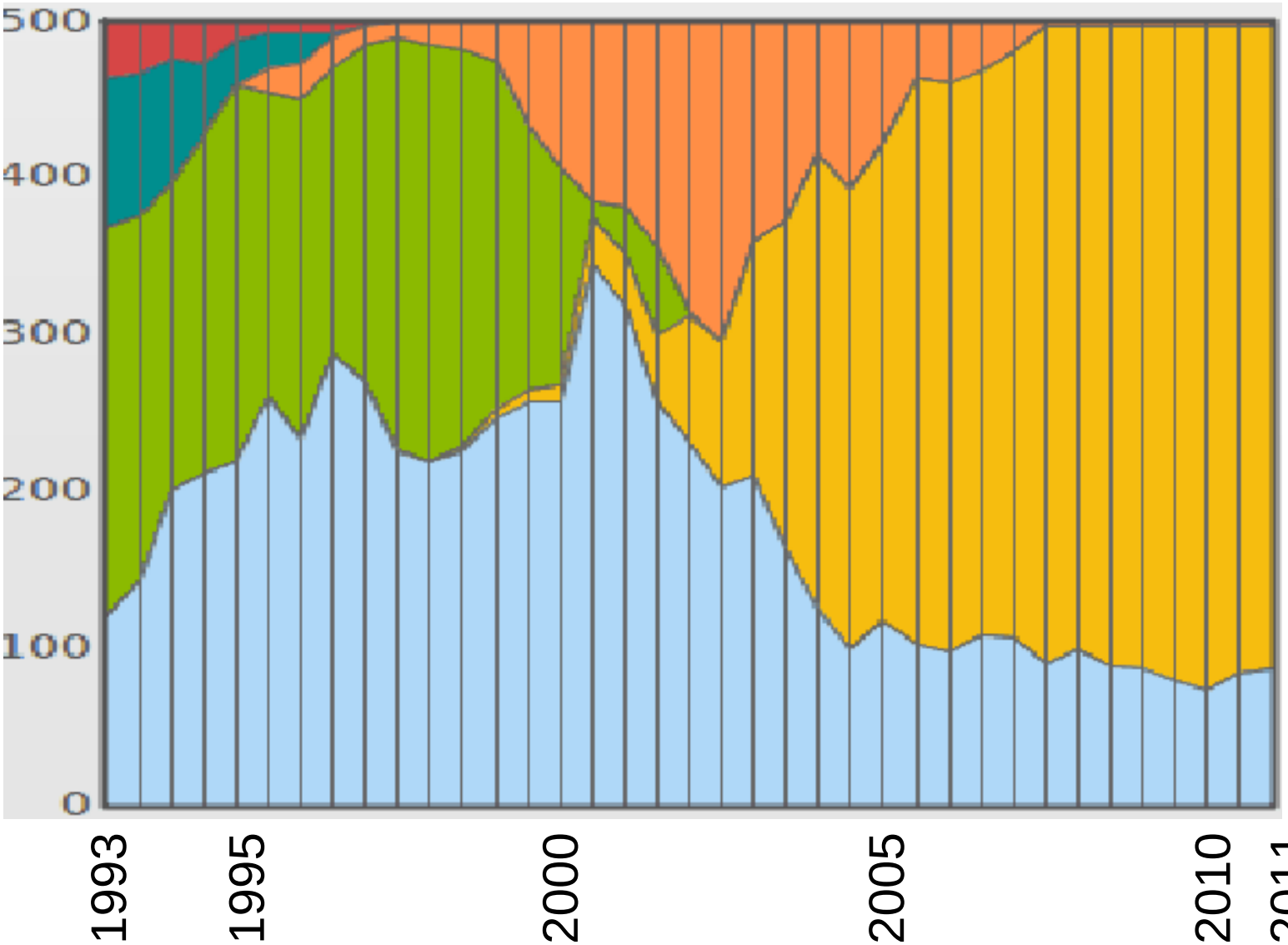
Warum Parallelverarbeitung?

- Anwendungen mit hohem Rechenbedarf, v.a. Simulationen
  - Klima, Erdbeben, Supraleitung, Molekülдesign, ...
- Beispiel: Proteinfaltung
  - 3D-Struktur, Funktion von Proteinen (Alzheimer, BSE, ...)
  - $1,5 \cdot 10^{11}$ Gleitkomma-Operationen (Flop) / Zeitschritt
  - Zeitschritt: $5 \cdot 10^{-15} s$
  - zu simulieren: $10^{-3} s$
  - $3 \cdot 10^{22}$ Flop / Simulation
  - $\Rightarrow$ 1 Jahr Rechenzeit auf einem PFlop/s Rechner!
- Zum Vergleich: derzeit schnellster Rechner der Welt: Tianhe-2 (China, Intel Xeon), 54,9 PFlop/s (mit 3120000 CPU-Kernen!)

Warum Parallelverarbeitung ...

- **Moore's Law**: Rechenleistung eines Prozessors verdoppelt sich alle 18 Monate
  - aber: Speichergeschwindigkeit bleibt zurück
  - ca. 2020: physikalische Grenze wird erreicht
- Daher:
  - Hochleistungsrechner basieren auf Parallelverarbeitung
  - selbst Standardprozessoren nutzen intern Parallelverarbeitung
    - Superskalarität, Pipelining, Multicore, ...
- Wirtschaftliche Vorteile von Parallelrechnern
  - billige Standardprozessoren statt Spezialentwicklungen

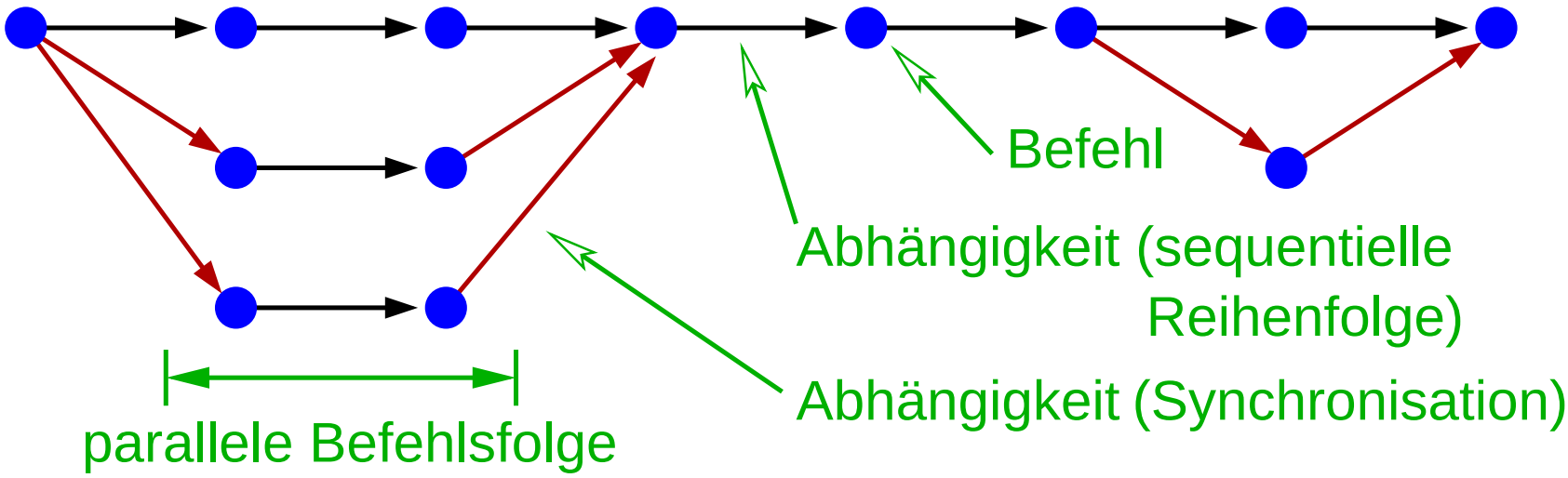
Architekturtrend bei Höchstleistungsrechnern



Quelle:
Top500 Liste
www.top500.org

Was ist ein paralelles Programm?

- Ein paralleles Programm kann als halbgeordnete Menge von Befehlen (Aktivitäten) dargestellt werden
  - die Ordnung ist durch die Abhängigkeiten der Befehle untereinander gegeben
- Unabhängige Befehle können parallel ausgeführt werden



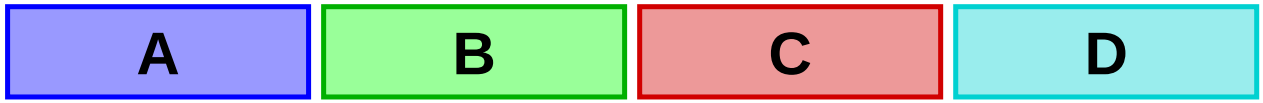
Nebenläufigkeit vs. Pipelining

- **Nebenläufigkeit**: Befehle werden gleichzeitig in mehreren Verarbeitungseinheiten ausgeführt
- **Pipelining**: Ausführung der Befehle ist in sequentielle Phasen zerlegt.
  Unterschiedliche Phasen **verschiedener** Befehls**instanzen** werden gleichzeitig ausgeführt.

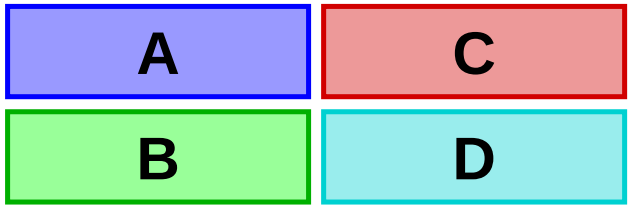
- Anmerkung: „Befehl“ meint hier allgemein eine Berechnungs-aktivität, abhängig von der betrachteten Ebene
  - z.B. Maschinenbefehl, Ausführung eines Unterprogramms

Nebenläufigkeit vs. Pipelining ...

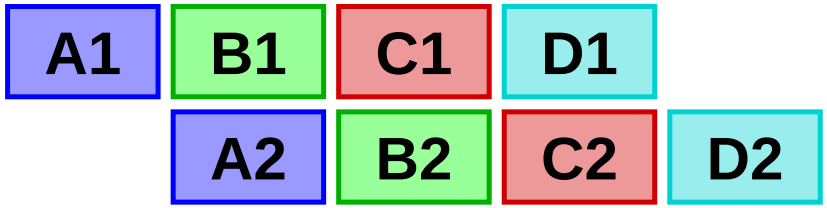
Sequentielle Ausführung



Nebenläufige Ausführung



Pipelining (2 Stufen)



Auf welchen Ebenen der Programmierung kann Parallelität genutzt werden?

- Es gibt keine einheitliche Klassifikation
- Z.B. Ebenen bei Waldschmidt (Parallelrechner: Architekturen - Systeme - Werkzeuge, Teubner, 1995):
  - Benutzerprogramme
  - kooperierende Prozesse
  - Datenstrukturen
  - Anweisungen und Schleifen
  - Maschinenbefehle

„*Sie sind heterogen, nach verschiedenen Merkmalen aufgeteilt und überlappen sich zum Teil*“

Sicht des Anwendungs-Entwicklers (Designphase):

- „natürlicher Parallelismus“
  - z.B. Berechnung der Kräfte für alle Sterne einer Galaxie
  - oft zu feinkörnig
- **Datenparallelität** (Gebietsaufteilung)
  - z.B. sequentielle Bearbeitung aller Sterne eines Raumgebiets
- **Taskparallelität** (Aufgabenaufteilung)
  - z.B. Vorverarbeitung, Berechnung, Nachbearbeitung, Visualisierung

Sicht des Programmierers:

- **Explizite Parallelität**
  - Datenaustausch (Kommunikation / Synchronisation) muß selbst programmiert werden
- **Implizite Parallelität**
  - durch Compiler
    - direktivengesteuert oder automatisch
    - Schleifenebene / Anweisungsebene
    - Compiler erzeugt Code für Kommunikation
  - innerhalb einer (nach außen hin sequentiellen) CPU
    - Superskalarität, Pipelining, ...

Sicht des Systems (Rechner/Betriebssystem):

- **Programmebene** (**Jobebene**)
  - unabhängige Programme
- **Prozessebene** (**Taskebene**)
  - kooperierende Prozesse
  - meist mit explizitem Nachrichtenaustausch
- **Blockebene**
  - leichtgewichtige Prozesse (Threads)
  - Kommunikation über gemeinsamen Speicher
  - oft durch Compiler erzeugt
    - Parallelisierung von Schleifen

Sicht des Systems (Rechner/Betriebssystem): ...

- **Anweisungsebene** (**Befehlsebene**)
  - elementare Anweisungen (in der Sprache nicht weiter zerlegbare Datenoperationen)
  - Scheduling automatisch durch Compiler und/oder zur Laufzeit durch Hardware
  - z.B. bei VLIW (EPIC), superskalaren Prozessoren
- **Suboperationsebene**
  - elementare Anweisungen werden durch den Compiler oder in der Maschine in Suboperationen aufgebrochen, die parallel ausgeführt werden
    - z.B. bei Vektor- oder Feldoperationen

Parallelverarbeitung

WS 2015/16

22.10.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Granularität

- Gegeben durch Verhältnis von Berechnung zu Kommunikation und Synchronisation
  - entspricht intuitiv der Länge der parallelen Befehlsfolgen in der Halbordnung
  - bestimmt Anforderung an Parallelrechner
    - v.a. Kommunikations-System
  - beinflußt die erreichbare Beschleunigung (*Speedup*)
- Grobkörnig: Programm- und Prozeßebene
- Mittelkörnig: Blockebene
- Feinkörnig: Anweisungsebene

1.3 Parallelisierung und Datenabhängigkeiten

- Wichtige Frage: wann können zwei Anweisungen S_1 und S_2 parallel ausgeführt werden?
  - Antwort: wenn es keine **Abhängigkeiten** zwischen ihnen gibt
- Annahme: Anweisung S_1 kann und soll laut sequentiellem Code **vor** Anweisung S_2 ausgeführt werden
  - z.B.: S_1: `x = b + 2 * a;`
    `y = a * (c - 5);`
    S_2: `z = abs(x - y);`
  - aber auch in verschiedenen Iterationen einer Schleife
- **Echte Abhängigkeit** (***true / flow dependence***) $S_1 \overset{\delta^t}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  a[i] = a[i-1] + b[i];
  ...
```

1.3 Parallelisierung und Datenabhängigkeiten

- Wichtige Frage: wann können zwei Anweisungen S_1 und S_2 parallel ausgeführt werden?
  - Antwort: wenn es keine **Abhängigkeiten** zwischen ihnen gibt
- Annahme: Anweisung S_1 kann und soll laut sequentiellem Code **vor** Anweisung S_2 ausgeführt werden
  - z.B.: S_1: `x = b + 2 * a;`
    `y = a * (c - 5);`
    S_2: `z = abs(x - y);`
  - aber auch in verschiedenen Iterationen einer Schleife
- **Echte Abhängigkeit** (***true / flow dependence***) $S_1 \overset{\delta^t}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  a[i] = a[i-1] + b[i];
  ...
```

S_1: `a[1] = a[0] + b[1];`

S_2: `a[2] = a[1] + b[2];`

1.3 Parallelisierung und Datenabhängigkeiten

- Wichtige Frage: wann können zwei Anweisungen S_1 und S_2 parallel ausgeführt werden?
  - Antwort: wenn es keine **Abhängigkeiten** zwischen ihnen gibt
- Annahme: Anweisung S_1 kann und soll laut sequentiellem Code **vor** Anweisung S_2 ausgeführt werden
  - z.B.: S_1: `x = b + 2 * a;`
    `y = a * (c - 5);`
    S_2: `z = abs(x - y);`
  - aber auch in verschiedenen Iterationen einer Schleife
- **Echte Abhängigkeit** (***true / flow dependence***) $S_1 \overset{\delta^t}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  a[i] = a[i-1] + b[i];
  ...
```

S_1: `a[1] = a[0] + b[1];`

δ^t

S_2: `a[2] = a[1] + b[2];`

1.3 Parallelisierung und Datenabhängigkeiten

- Wichtige Frage: wann können zwei Anweisungen S_1 und S_2 parallel ausgeführt werden?
  - Antwort: wenn es keine **Abhängigkeiten** zwischen ihnen gibt
- Annahme: Anweisung S_1 kann und soll laut sequentiellem Code **vor** Anweisung S_2 ausgeführt werden
  - z.B.:

```
S1:  x = b + 2 * a;
     y = a * (c - 5);
S2:  z = abs(x - y);
```

  - aber auch in verschiedenen Iterationen einer Schleife
- **Echte Abhängigkeit** (***true / flow dependence***) $S_1 \xrightarrow{\delta^t} S_2$

```
S1: a[1] = a[0] + b[1];
          δt
S2: a[2] = a[1] + b[2];
```

S1 (i=1) schreibt a[1], das von S2 (i=2) gelesen wird

- **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  a[i] = a[i+1];
  ...
```

➥ **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  a[i] = a[i+1];
  ...
```

S_1: `a[1] = a[2];`

S_2: `a[2] = a[3];`

➥ **Antiabhängigkeit** (***anti dependence***) $S_1 \xrightarrow{\delta^a} S_2$

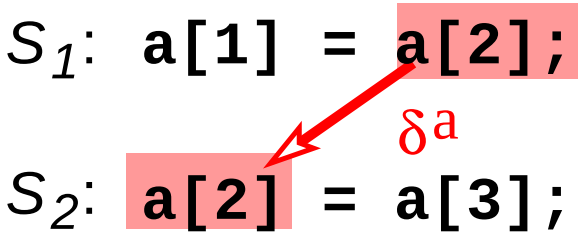
```
for (i=1; i<N; i++) {
  a[i] = a[i+1];
  ...
```

S_1: `a[1] = a[2];`

δ^a

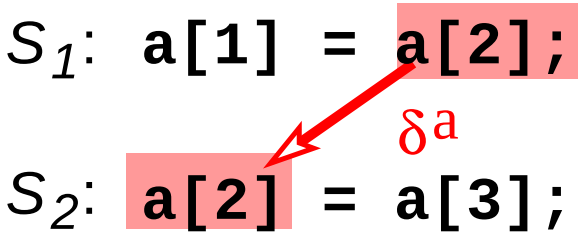
S_2: `a[2] = a[3];`

➥ **Antiabhängigkeit** (***anti dependence***) $S_1 \xrightarrow{\delta^a} S_2$



S1 (i=1) liest den Wert a[2], der von S2 (i=2) überschrieben wird

- **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$



S1 (i=1) liest den Wert a[2], der von S2 (i=2) überschrieben wird

- **Ausgabeabhängigkeit** (***output dependence***) $S_1 \overset{\delta^o}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  s = a[i];
  ...
```

- **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$

```
S1: a[1] = a[2];
S2: a[2] = a[3];
```

δ^a

S1 (i=1) liest den Wert a[2], der von S2 (i=2) überschrieben wird

- **Ausgabeabhängigkeit** (***output dependence***) $S_1 \overset{\delta^o}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  s = a[i];
  ...
```

```
S1: s = a[1];
S2: s = a[2];
```

- **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$

```
S1: a[1] = a[2];
          δa
S2: a[2] = a[3];
```

S1 (i=1) liest den Wert a[2], der von S2 (i=2) überschrieben wird

- **Ausgabeabhängigkeit** (***output dependence***) $S_1 \overset{\delta^o}{\rightarrow} S_2$

```
for (i=1; i<N; i++) {
  s = a[i];
  ...
```

```
S1: s = a[1];
       δo
S2: s = a[2];
```

- **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$

```
S1: a[1] = a[2];
        δa
S2: a[2] = a[3];
```

S1 (i=1) liest den Wert a[2], der von S2 (i=2) überschrieben wird

- **Ausgabeabhängigkeit** (***output dependence***) $S_1 \overset{\delta^o}{\rightarrow} S_2$

```
S1: s = a[1];
        δo
S2: s = a[2];
```

S1 (i=1) schreibt Wert in s, der von S2 (i=2) überschrieben wird

- **Antiabhängigkeit** (***anti dependence***) $S_1 \overset{\delta^a}{\rightarrow} S_2$

```
S1: a[1] = a[2];
        δa
S2: a[2] = a[3];
```

S1 (i=1) liest den Wert a[2], der von S2 (i=2) überschrieben wird

- **Ausgabeabhängigkeit** (***output dependence***) $S_1 \overset{\delta^o}{\rightarrow} S_2$

```
S1: s = a[1];
        δo
S2: s = a[2];
```

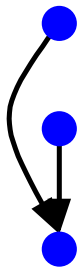
S1 (i=1) schreibt Wert in s, der von S2 (i=2) überschrieben wird

- **Anti-** und **Ausgabe**abhängigkeiten können immer durch (konsistente) Umbenennung von Variablen beseitigt werden

Datenabhängigkeiten und Synchronisation

- Zwei Anweisungen S_1 und S_2 mit einer Datenabhängigkeit $S_1 \rightarrow S_2$ können auf verschiedene Threads verteilt werden, **wenn** eine korrekte Synchronisation durchgeführt wird
  - S_2 muß **nach** S_1 ausgeführt werden
  - z.B. durch `signal`/`wait` oder Nachricht
- Im Beispiel von vorhin:

```
x = b + 2 * a;
y = a * (c-5);
z = abs(x-y);
```

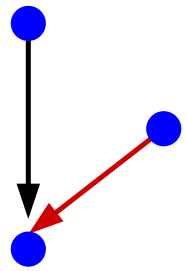


Datenabhängigkeiten und Synchronisation

- Zwei Anweisungen S_1 und S_2 mit einer Datenabhängigkeit $S_1 \rightarrow S_2$ können auf verschiedene Threads verteilt werden, **wenn** eine korrekte Synchronisation durchgeführt wird
  - S_2 muß **nach** S_1 ausgeführt werden
  - z.B. durch `signal`/`wait` oder Nachricht
- Im Beispiel von vorhin:

Thread 1

```
x = b + 2 * a;


wait(cond);
z = abs(x-y);
```



Thread 2

```
y = a * (c-5);
signal(cond);
```

Klassifikation von Rechnerarchitekturen nach Flynn

- Unterscheidungskriterien:
  - wieviele **Befehlsströme** bearbeitet ein Rechner zu einem gegebenen Zeitpunkt (einen, mehrere)?
  - wieviele **Datenströme** bearbeitet ein Rechner zu einem gegebenen Zeitpunkt (einen, mehrere)?
- Daraus ergeben sich vier mögliche Klassen:
  - SISD: ***S**ingle **I**nstruction stream, **S**ingle **D**ata stream*
    - Einprozessorrechner
  - MIMD: ***M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams*
    - alle Arten von Multiprozessorsystemen
  - SIMD: Vektorrechner bzw. -erweiterungen, GPUs
  - MISD: leer, da nicht wirklich sinnvoll

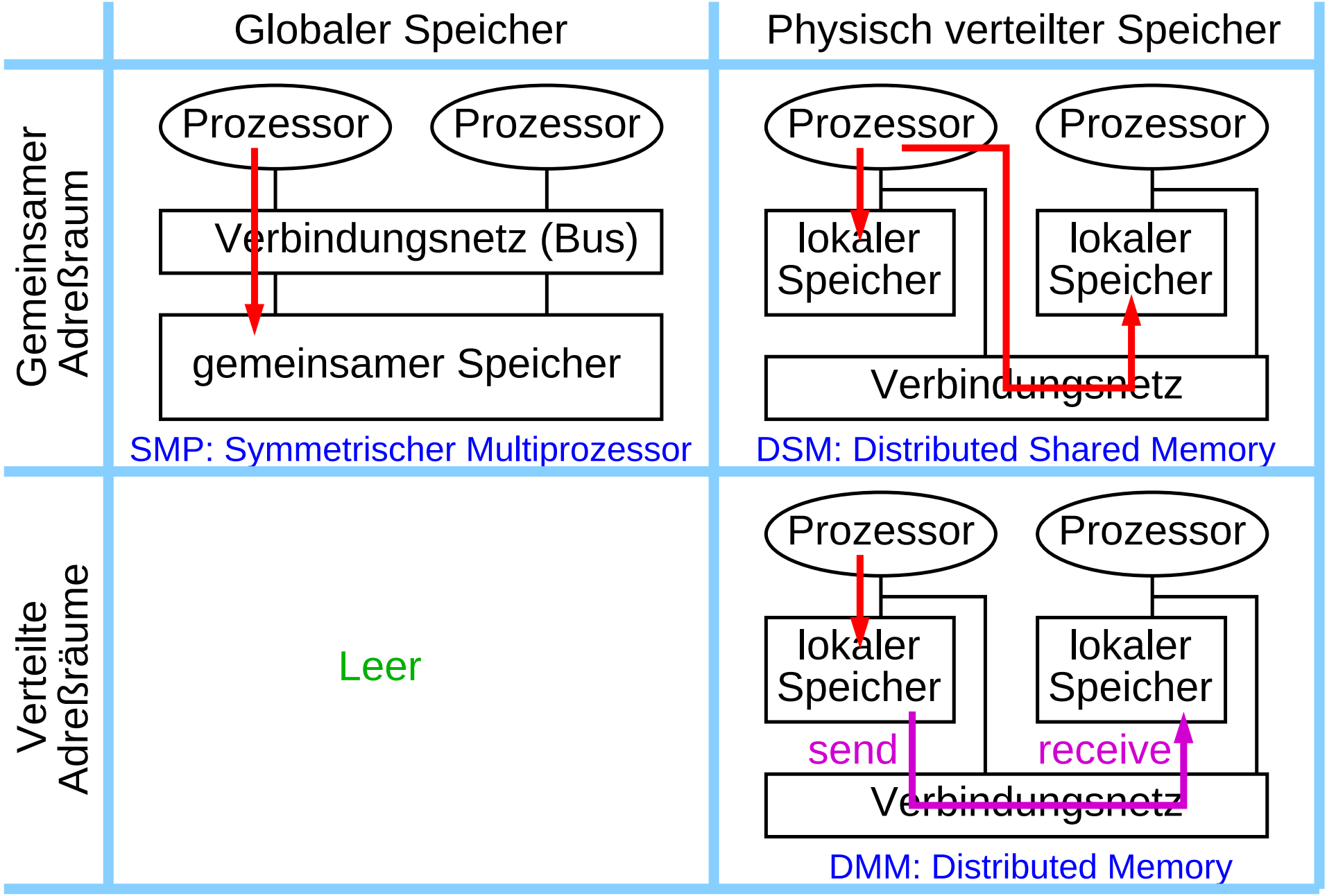
Klassen von MIMD-Rechnern

- Zwei Kriterien betrachtet:
  - Physisch globaler vs. verteilter Speicher
  - Gemeinsamer vs. verteilter Adressraum
- **NORMA**: ***No Remote Memory Access***
  - verteilter Speicher, verteilter Adressraum
  - d.h. kein Zugriff auf Speichermodule nicht-lokaler Knoten
  - Kommunikation nur über Nachrichten möglich
  - typische Vertreter der Klasse:
    - ***Distributed Memory*-Systeme (DMM)**
      - auch: MPP (*Massively Parallel Processor*) genannt
    - im Prinzip auch Rechnernetze (Cluster, Grid, Cloud, ...)

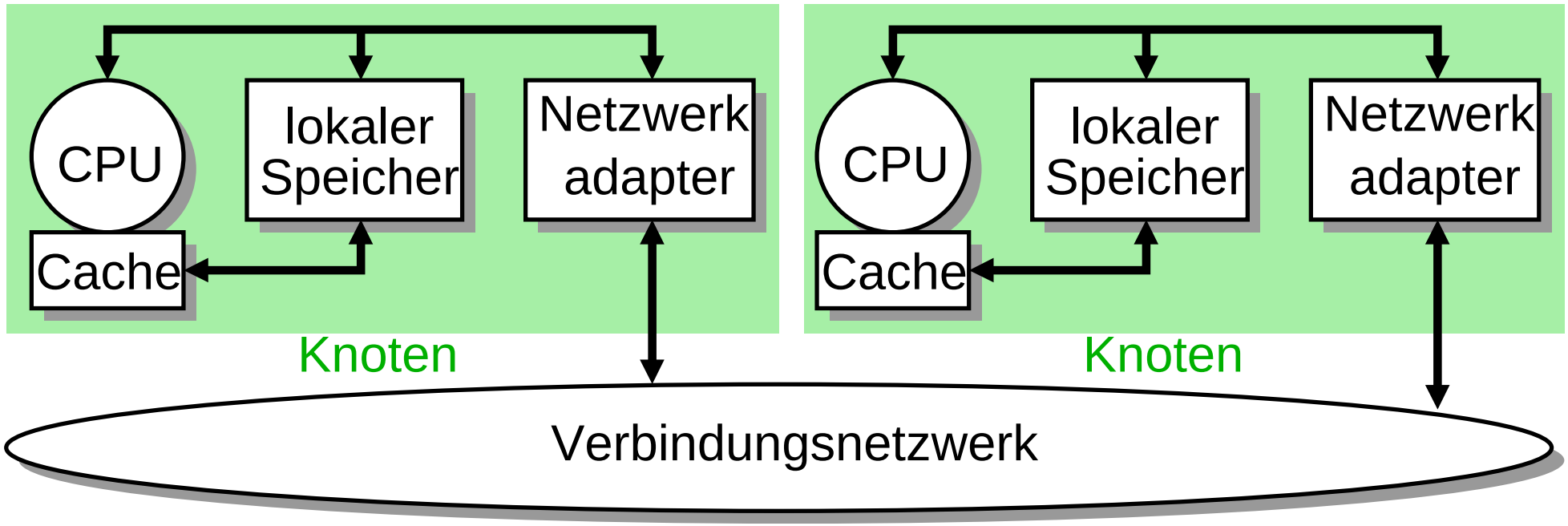
Klassen von MIMD-Rechnern ...

- **UMA**: ***Uniform Memory Access***
  - globaler Speicher, gemeinsamer Adressraum
  - alle Prozessoren greifen in gleicher Weise auf den Speicher zu
  - Zugriffszeit ist für alle Prozessoren gleich
  - typische Vertreter der Klasse:
    Symmetrische Multiprozessoren (SMP), *Multicore*-CPUs
- **NUMA**: ***Nonuniform Memory Access***
  - verteilter Speicher, gemeinsamer Adressraum
  - Zugriff auf lokalen Speicher schneller als auf entfernten
  - typische Vertreter der Klasse:
    ***Distributed Shared Memory*-Systeme (DSM)**

1.4 Parallelrechnerarchitekturen ...



Multiprozessorsysteme mit verteiltem Speicher



- **NORMA**: *No Remote Memory Access*
- Gut Skalierbar (bis mehrere 10000 Knoten)
- Kommunikation und Synchronisation über Nachrichtenaustausch

1.4.1 MIMD: Nachrichtengekoppelte Systeme ...

Entwicklung

- Früher: proprietäre Hardware für Knoten und Netzwerk
  - eigene Knotenarchitektur (Prozessor, Netzwerkadapter, ...)
  - oft statische Verbindungsnetze mit *Store and Forward*
  - oft eigene (Mini-)Betriebssysteme
- Heute:
  - Cluster aus Standardkomponenten (PC-Server)
    - ggf. mit Hochleistungs-Netzwerk (Infiniband, Myrinet, ...)
  - oft SMP- und/oder Vektor-Rechner als Knoten
    - für Höchstleistungsrechner
  - dynamische (geschaltete) Verbindungsnetze
  - Standard-Betriebssysteme (UNIX- oder Linux-Derivate)

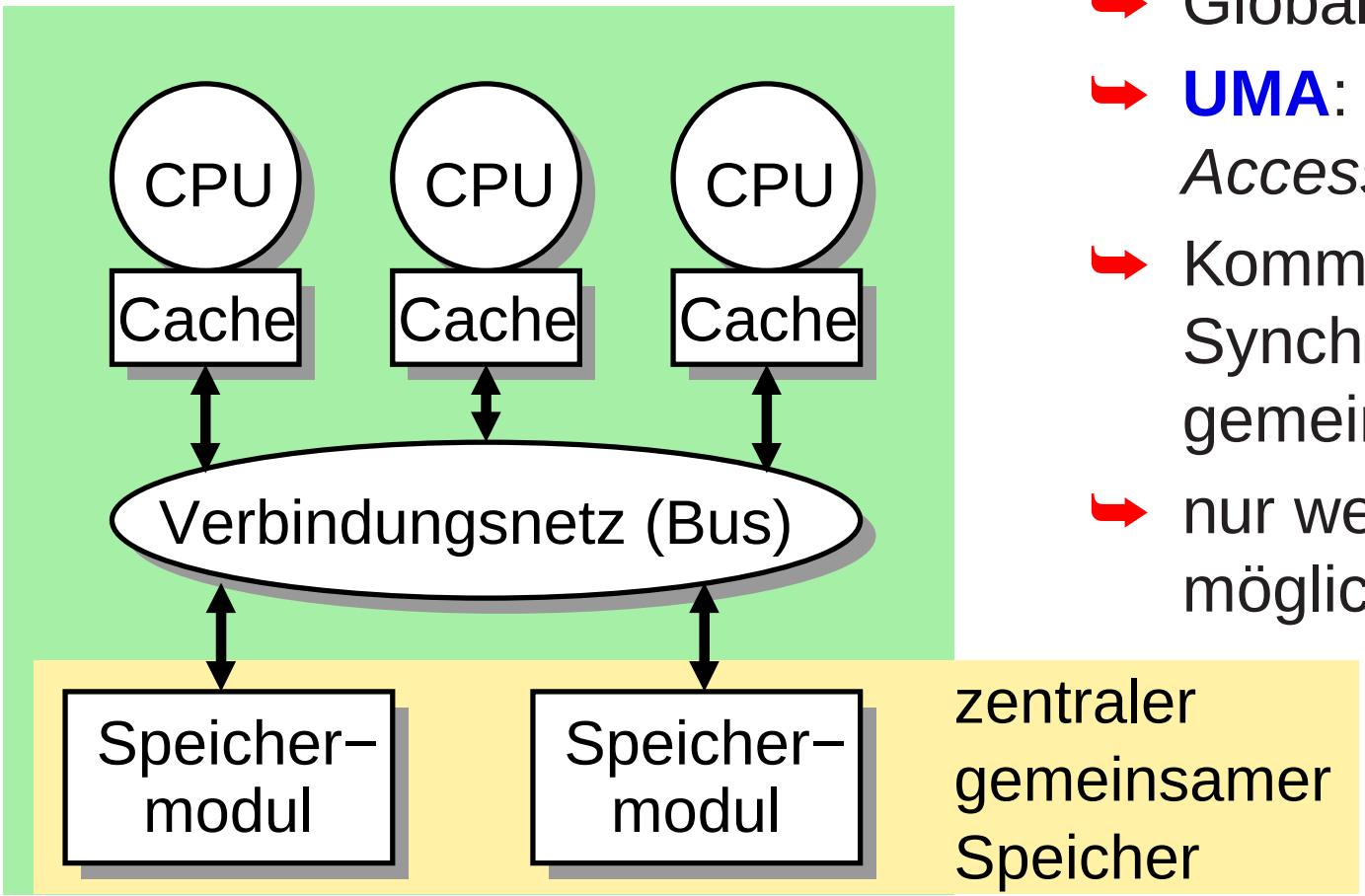
Eigenschaften

- Keine gemeinsamen Speicher- oder Adreßbereiche
- Kommunikation durch Austausch von Nachrichten
  - Anwendungsebene: Bibliotheken wie z.B. MPI
  - Systemebene: proprietäre Protokolle oder TCP/IP
  - Latenz durch Software meist wesentlich größer als Hardware-Latenz ($\sim 1 - 50 \mu s$ gegenüber $\sim 20 - 100 ns$)
- Im Prinzip unbegrenzte Skalierbarkeit
  - z.B. BlueGene/Q (Sequoia): 98304 Knoten, (1572864 *Cores*)

Eigenschaften ...

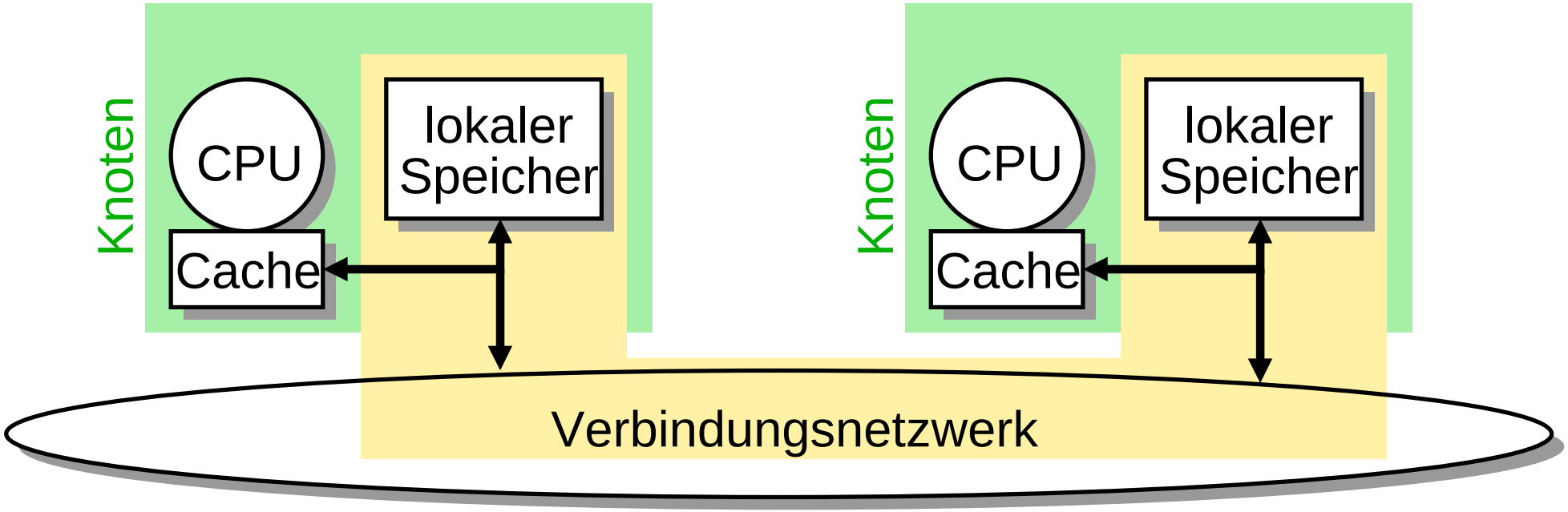
- Unabhängige Betriebssysteme auf jedem Knoten
- Oft gemeinsames Dateisystem
  - z.B. paralleles Dateisystem, über (eigenes) Verbindungsnetz an alle Knoten gekoppelt
  - oder einfach NFS (bei kleineren Clustern)
- In der Regel kein *Single System Image*
  - Benutzer/Administrator „sieht“ mehrere Rechner
- Oft kein direkter, interaktiver Zugang zu allen Knoten
  - Knoten werden durch *Batch Queueing Systeme* nur bei Bedarf an parallele Programme zugeteilt
    - meist exklusiv: *Space Sharing*, Partitionierung
  - meist kleine, feste Partition für Login / interaktives Arbeiten

Symmetrische Multiprozessoren (SMP)



- Globaler Adreßraum
- **UMA**: *Uniform Memory Access*
- Kommunikation und Synchronisation über gemeinsamen Speicher
- nur wenige Prozessoren möglich (ca. 8 - 32)

Multiprozessoren mit verteiltem gemeinsamem Speicher (DSM)



- Verteilter Speicher, von allen CPUs aus zugreifbar
- **NUMA**: *Non Uniform Memory Access*
- Kombiniert gemeinsamen Speicher mit Skalierbarkeit

1.4.2 MIMD: Speichergekoppelte Systeme ...

Eigenschaften

- Alle Prozessoren können auf alle Betriebsmittel in gleicher Weise zugreifen
  - aber: bei NUMA unterschiedliche Speicherzugriffszeit
    - Daten so aufteilen, dass Zugriffe möglichst lokal erfolgen
- Nur eine Betriebssystem-Instanz für den gesamten Rechner
  - verteilt Prozesse/Threads auf verfügbare Prozessoren
  - alle Prozessoren können gleichberechtigt Betriebssystemdienste ausführen
- *Single System Image*
  - für Benutzer/Administrator praktisch identisch zu Einprozessorsystem
- Besonders bei SMP (UMA) nur begrenzte Skalierbarkeit

Caches in speichergekoppelten Systemen

- **Cache**: schneller, prozessornaher Zwischenspeicher
  - speichert Kopien der zuletzt am häufigsten benutzten Daten aus dem Hauptspeicher
  - falls Daten im Cache: kein Hauptspeicherzugriff nötig
    - Zugriff ist 10-1000 mal schneller
- Caches sind in Multiprozessorsystemen essentiell
  - Speicher/Verbindungsnetz wird sonst schnell zum Engpaß
  - Ausnutzen einer Lokalitätseigenschaft
    - jeder Prozeß rechnet meist nur auf „seinen“ Daten
- Aber: Existenz mehrerer Kopien von Daten kann zu Inkonsistenzen führen: **Cache-Kohärenz-Problem** (☞ **BS-1**)

Sicherstellung der Cache-Kohärenz

- Bei einem Schreibzugriff müssen alle betroffenen Caches (= Caches mit Kopien) benachrichtig werden
  - Invalidierung oder Aktualisierung des betroffenen Eintrags
- Bei UMA-Systemen
  - Bus als Verbindungsnetz: jeder Hauptspeicherzugriff ist für alle sichtbar (*Broadcast*)
  - Caches „horchen“ am Bus mit (*Bus Snooping*)
  - (relativ) einfache Cache-Kohärenz-Protokolle
    - z.B. MESI-Protokoll
  - aber: schlecht skalierbar, da Bus zentrale Ressource ist

Sicherstellung der Cache-Kohärenz ...

- Bei NUMA-Systemen (ccNUMA: *cache coherent NUMA*)
  - Hauptspeicherzugriffe sind für andere Prozessoren i.d.R. nicht sichtbar
  - betroffene Caches müssen explizit benachrichtigt werden
    - benötigt Liste betroffener Caches (*Broadcast* an alle Prozessoren ist zu teuer)
    - Nachrichten-Laufzeit führt zu neuen Konsistenz-Problemen
  - Cache-Kohärenz-Protokolle (*Directory*-Protokolle) werden sehr komplex
  - aber: gute Skalierbarkeit

Speicherkonsistenz (*Memory Consistency*)

- Cache-Kohärenz beschreibt nur das Verhalten in Bezug auf jeweils **ein Speicherwort**
  - **welche Werte** kann eine Leseoperation zurückliefern?
- Verbleibende Frage:
  - **wann** sieht ein Prozessor den Wert, den ein anderer geschrieben hat?
  - genauer: in **welcher Reihenfolge** sieht ein Prozessor Schreiboperationen auf verschiedene Speicherworte?

Speicherkonsistenz: Einfaches Beispiel

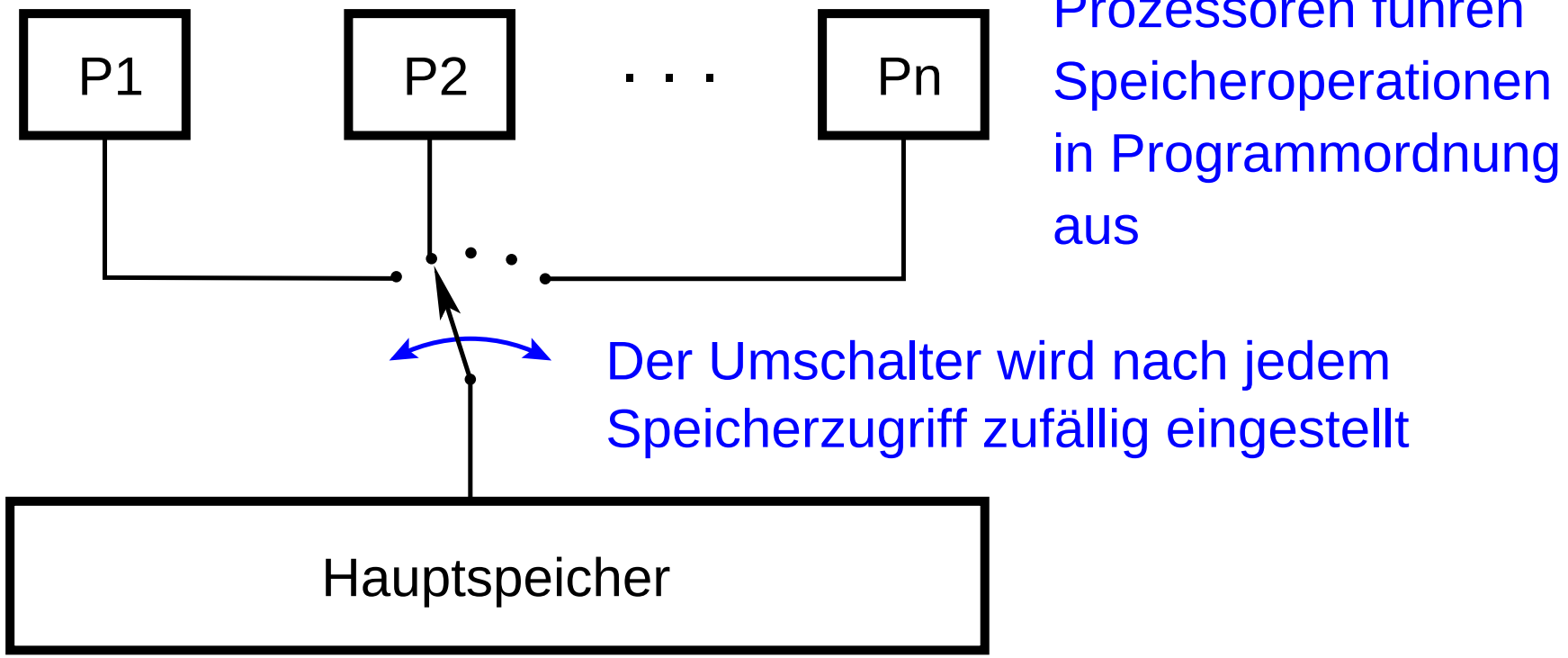
Thread T_1	Thread T_2
`A = 0;`	`B = 0;`
`...;`	`...;`
`A = 1;`	`B = 1;`
`print B;`	`print A;`

- Intuitive Erwartung: Ausgabe `"0 0"` ist unmöglich
- Aber: auf vielen SMPs/DSMs kann `"0 0"` ausgegeben werden
  - (CPUs mit dyn. Befelsscheduling oder Schreibpuffern)
- Trotz Cache-Kohärenz: intuitiv inkonsistente Sicht auf den Speicher:

T_1: `A=1, B=0` $\quad$ T_2: `A=0, B=1`

Definition: Sequentielle Konsistenz

Sequentiele Konsistenz liegt vor, wenn das Resultat jeder Ausführung eines parallelen Programms auch durch die folgende abstrakte Maschine produziert werden kann:



Verzahnungen (*Interleavings*) im Beispiel

Einige mögliche Abläufe mit der abstrakten Maschine:

A = 0	A = 0	A = 0
B = 0	B = 0	B = 0
A = 1	A = 1	B = 1
B = 1	print B	print A
print B	B = 1	A = 1
print A	print A	print B
B=1 A=1	B=0 A=1	B=1 A=0

Keine seq. Konsistenz:

A = 0
B = 0
B = 1
print A
A = 1
print B

B=0 A=0

Abgeschwächte Konsistenzmodelle

- Forderung nach sequentieller Konsistenz bedeutet starke Einschränkung für die Rechnerarchitektur
  - Befehlsscheduling und Schreibpuffer in CPUs nicht verwendbar
  - NUMA-Systeme wären nicht effizient realisierbar
- Daher: Parallelrechner mit gemeinsamem Speicher (UMA und NUMA) nutzen **abgeschwächte Konsistenzmodelle**!
  - erlaubt z.B. Vertauschung von Speicheroperationen
  - jeder Prozessor sieht seine **eigenen** Speicheroperationen jedoch **immer** in Programmordnung
- Anm.: auch optimierende Compiler führen zu abgeschwächter Konsistenz
  - Vertauschung von Befehlen, Register-Allokation, ...
  - betroffene Variable als `volatile` deklarieren!

Auswirkung abgeschwächter Konsistenz: Beispiele

- alle Variablen sind am Anfang 0

	Mögliche Ergebnisse bei sequentieller Konsistenz	"unerwartetes" Verhalten bei abgeschwächter Konsistenz:
`A=1;` `print B;` | `B=1;` `print A;`	0,1 1,0 1,1	durch Vertauschen von Lese− und Schreibzugriffen
`A=1;` `valid=1;` | `while (!valid);` `print A;`	1	durch Vertauschen der Schreibzugriffe auf A und valid

Abgeschwächte Konsistenzmodelle ...

- Speicherkonsistenz kann (und muß!) bei Bedarf über spezielle Befehle erzwungen werden
  - Speicherbarriere (*Fence*, *Memory Barrier*)
    - alle vorangehenden Speicheroperationen werden abgeschlossen, nachfolgende Speicheroperationen werden erst nach der Barriere gestartet
  - *Acquire* und *Release*
    - *Acquire*: nachfolgende Speicheroperationen werden erst nach Abschluß des *Acquire* gestartet
    - *Release*: alle vorangehende Speicheroperationen werden abgeschlossen
    - Verwendungsmuster wie bei *Mutex locks*

Korrekte Synchronisation in den Beispielen

- Hier mit Speicherbarrieren:

`A=1;`
`Fence;`
`print B;`
	`B=1;`
	`Fence;`
	`print A;`

`A=1;`
`Fence;`
`valid=1;`
	`while (!valid);`
	`Fence;`
	`print A;`

Fence stellt sicher, daß Schreiben beendet ist, bevor gelesen wird

Fence stellt sicher, daß 'A' gültig ist, bevor 'valid' gesetzt wird und daß A erst ausgelesen wird, wenn 'valid' gesetzt ist

Parallelverarbeitung

WS 2015/16

26.10.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

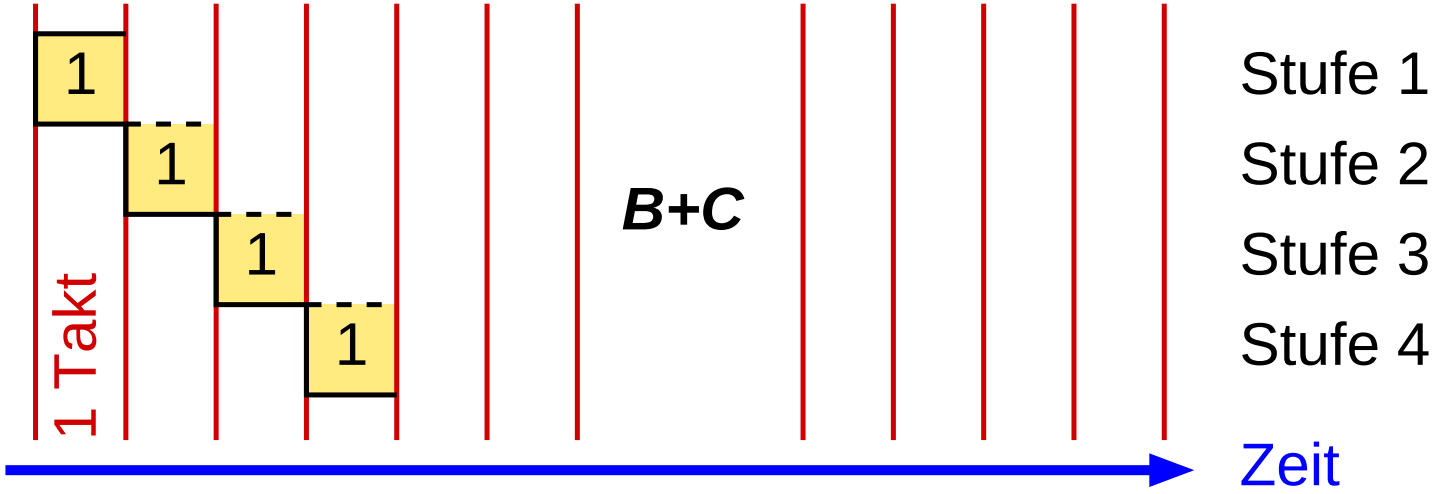
Stand: 1. Februar 2016

1.4.3 SIMD

- Nur ein Befehlsstrom, die Befehle haben jedoch **Vektoren** als Operanden ⇒ Datenparallelität
- **Vektor** = eindimensionales Feld (*Array*) von Zahlen
- Varianten:
  - Vektorrechner
    - pipelineartig aufgebaute Rechenwerke (Vektoreinheiten) zur Bearbeitung von Vektoren
  - SIMD-Erweiterungen in Prozessoren (SSE)
    - Intel: 128-Bit Register, mit z.B. 4 32-Bit Gleitkommazahlen
  - Graphikprozessoren (GPUs)
    - mehrere *Streaming*-Multiprozessoren
    - *Streaming*-Multiprozessor enthält mehrere Rechenwerke (*CUDA Cores*), die alle denselben Befehl bearbeiten

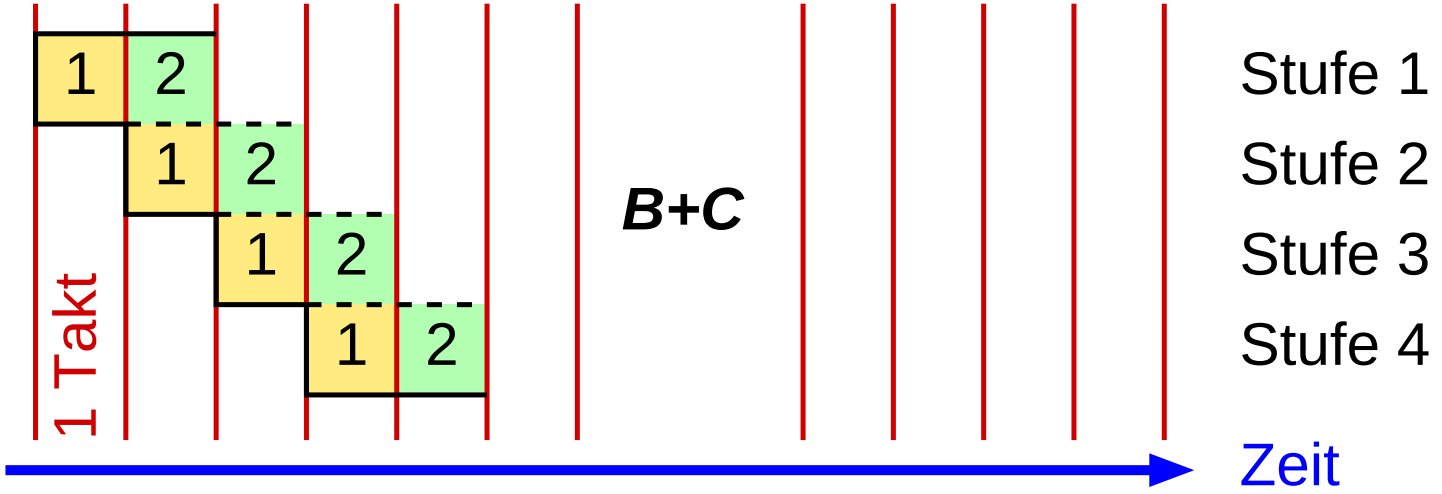
Beispiel: Addition zweier Vektoren

- $A(J) = B(J) + C(J)$, für alle $J = 1, 2, ..., N$
- Vektorrechner: Elemente der Vektoren werden in einer Pipeline **sequentiell**, aber **überlappt** addiert
  - falls eine einfache Addition vier Takte (d.h. 4 Pipelinestufen) benötigt, ergibt sich folgender Ablauf:



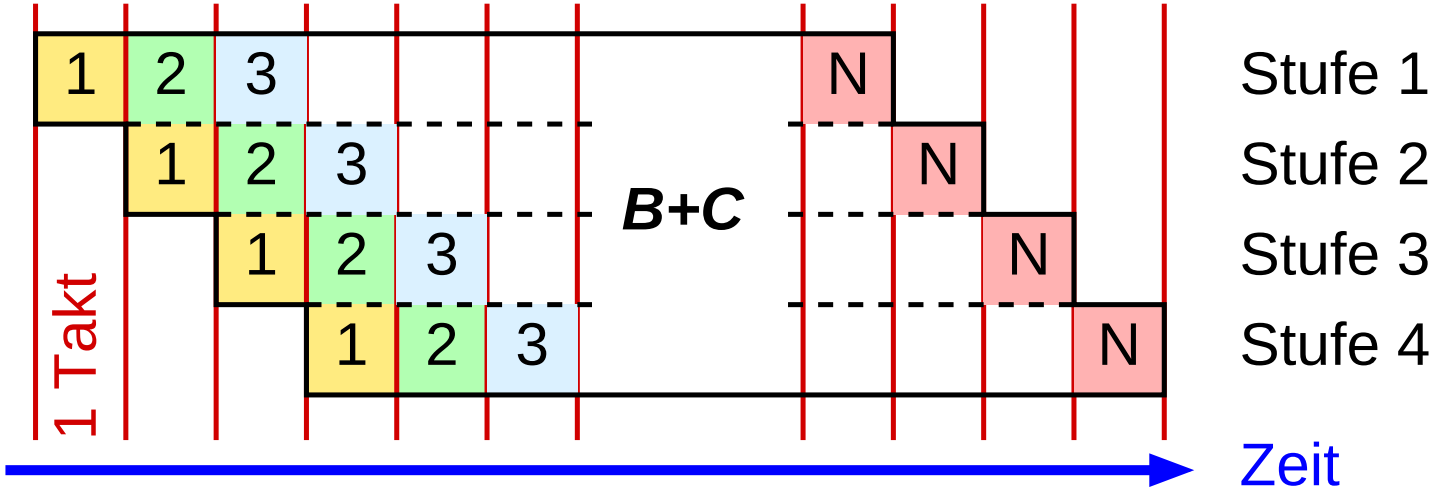
Beispiel: Addition zweier Vektoren

- $A(J) = B(J) + C(J)$, für alle $J = 1, 2, ..., N$
- Vektorrechner: Elemente der Vektoren werden in einer Pipeline **sequentiell**, aber **überlappt** addiert
  - falls eine einfache Addition vier Takte (d.h. 4 Pipelinestufen) benötigt, ergibt sich folgender Ablauf:



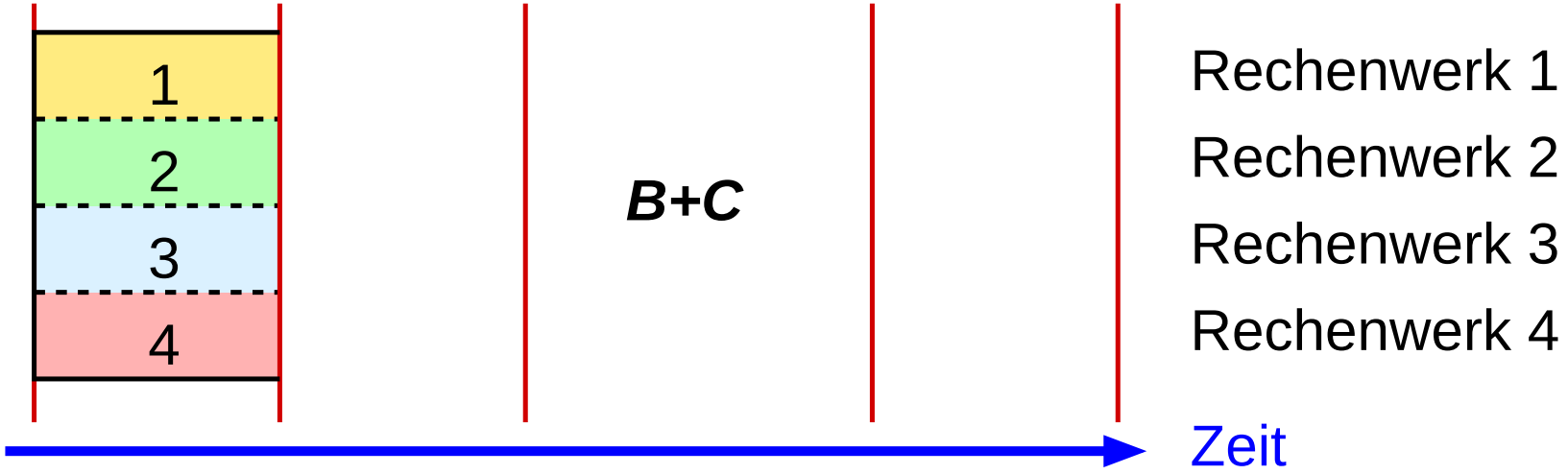
Beispiel: Addition zweier Vektoren

- $A(J) = B(J) + C(J)$, für alle $J = 1, 2, ..., N$
- Vektorrechner: Elemente der Vektoren werden in einer Pipeline **sequentiell**, aber **überlappt** addiert
  - falls eine einfache Addition vier Takte (d.h. 4 Pipelinestufen) benötigt, ergibt sich folgender Ablauf:



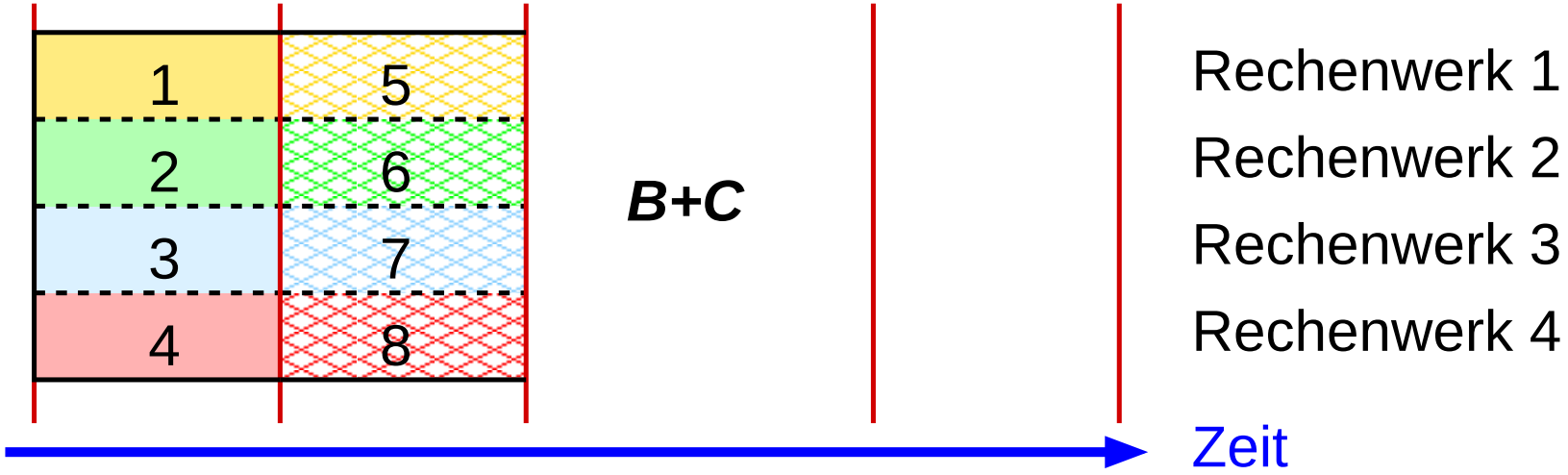
Beispiel: Addition zweier Vektoren

- $A(J) = B(J) + C(J)$, für alle $J = 1, 2, ..., N$
- SSE bzw. GPU: mehrere Elemente der Vektoren werden **gleichzeitig** (**parallel**) addiert
  - falls z.B. vier Additionen gleichzeitig möglich sind, ergibt sich folgender Ablauf:



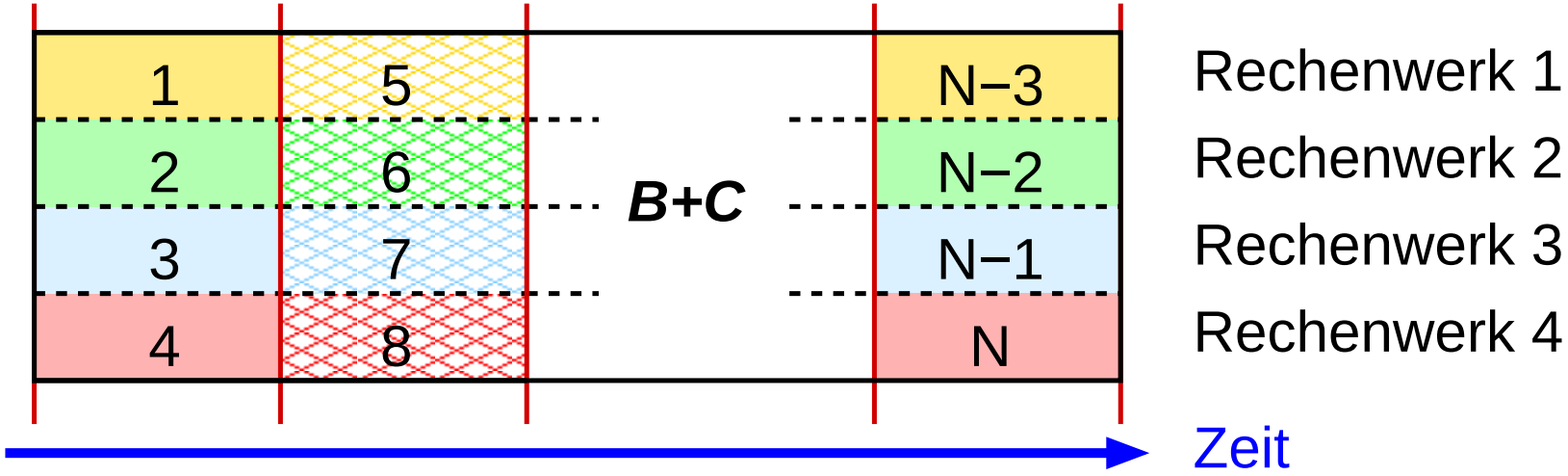
Beispiel: Addition zweier Vektoren

- $A(J) = B(J) + C(J)$, für alle $J = 1, 2, ..., N$
- SSE bzw. GPU: mehrere Elemente der Vektoren werden **gleichzeitig** (**parallel**) addiert
  - falls z.B. vier Additionen gleichzeitig möglich sind, ergibt sich folgender Ablauf:

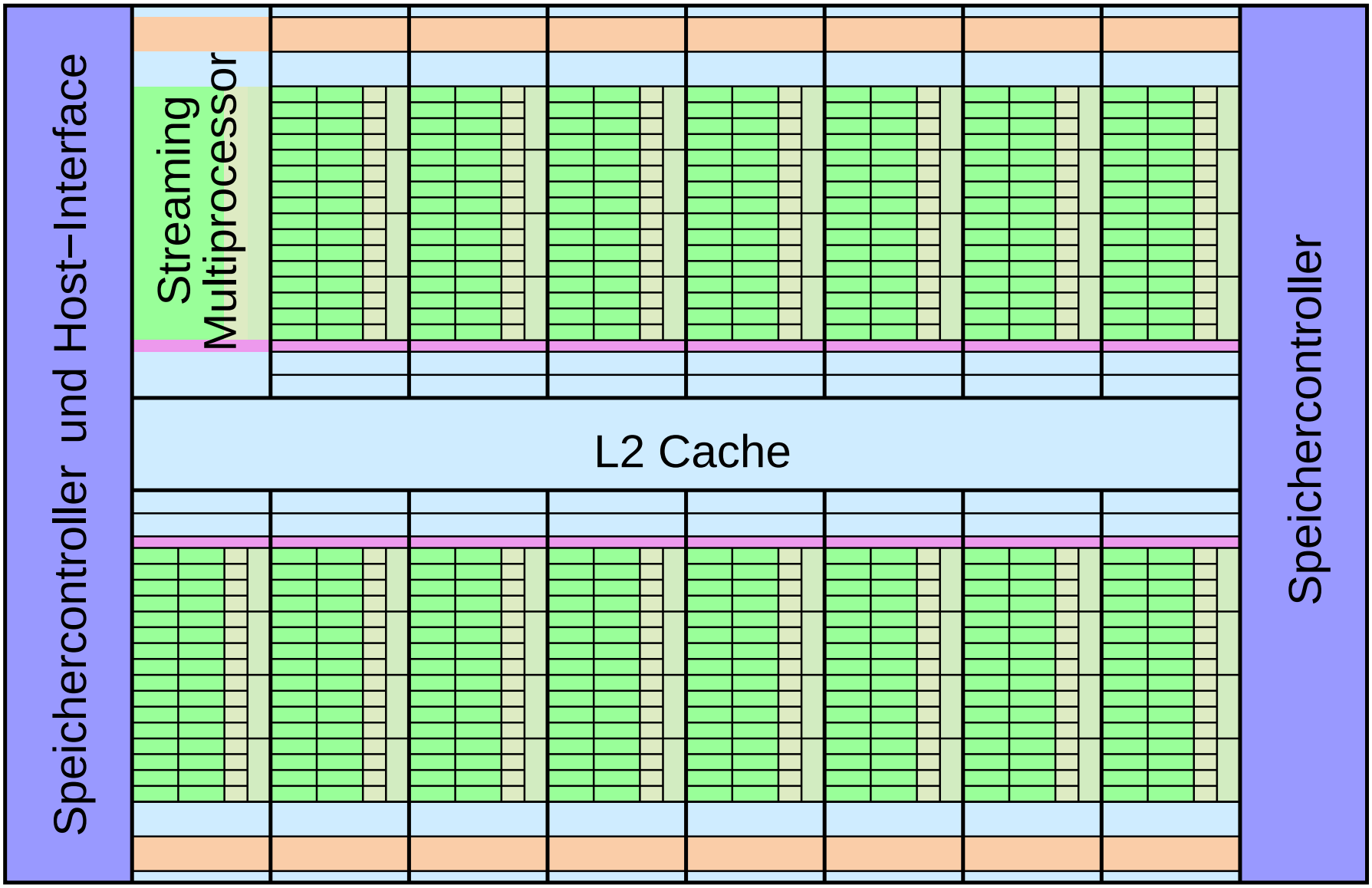


Beispiel: Addition zweier Vektoren

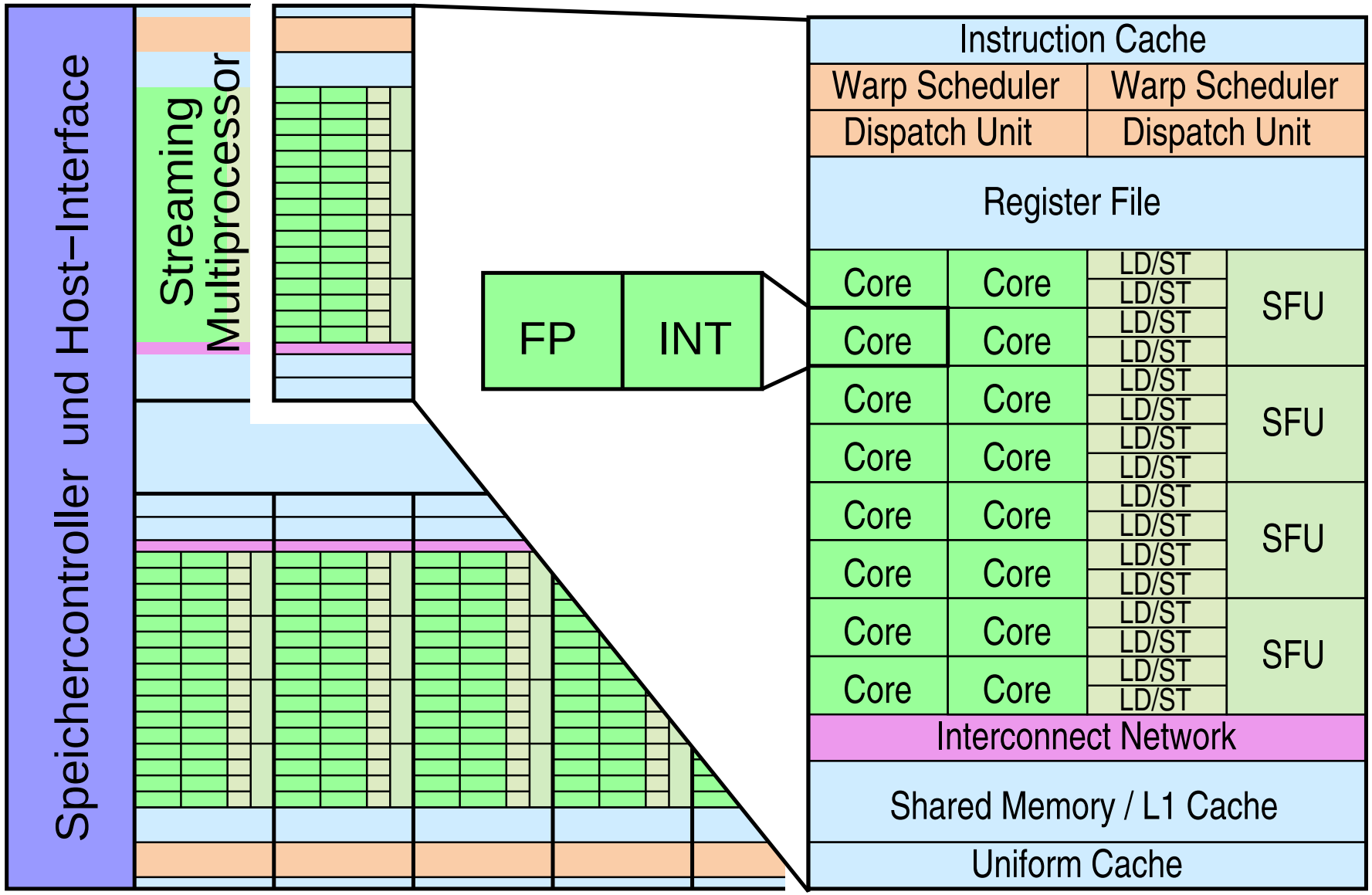
- $A(J) = B(J) + C(J)$, für alle $J = 1, 2, ..., N$
- SSE bzw. GPU: mehrere Elemente der Vektoren werden **gleichzeitig** (**parallel**) addiert
  - falls z.B. vier Additionen gleichzeitig möglich sind, ergibt sich folgender Ablauf:



Architektur einer GPU (NVIDIA Fermi)



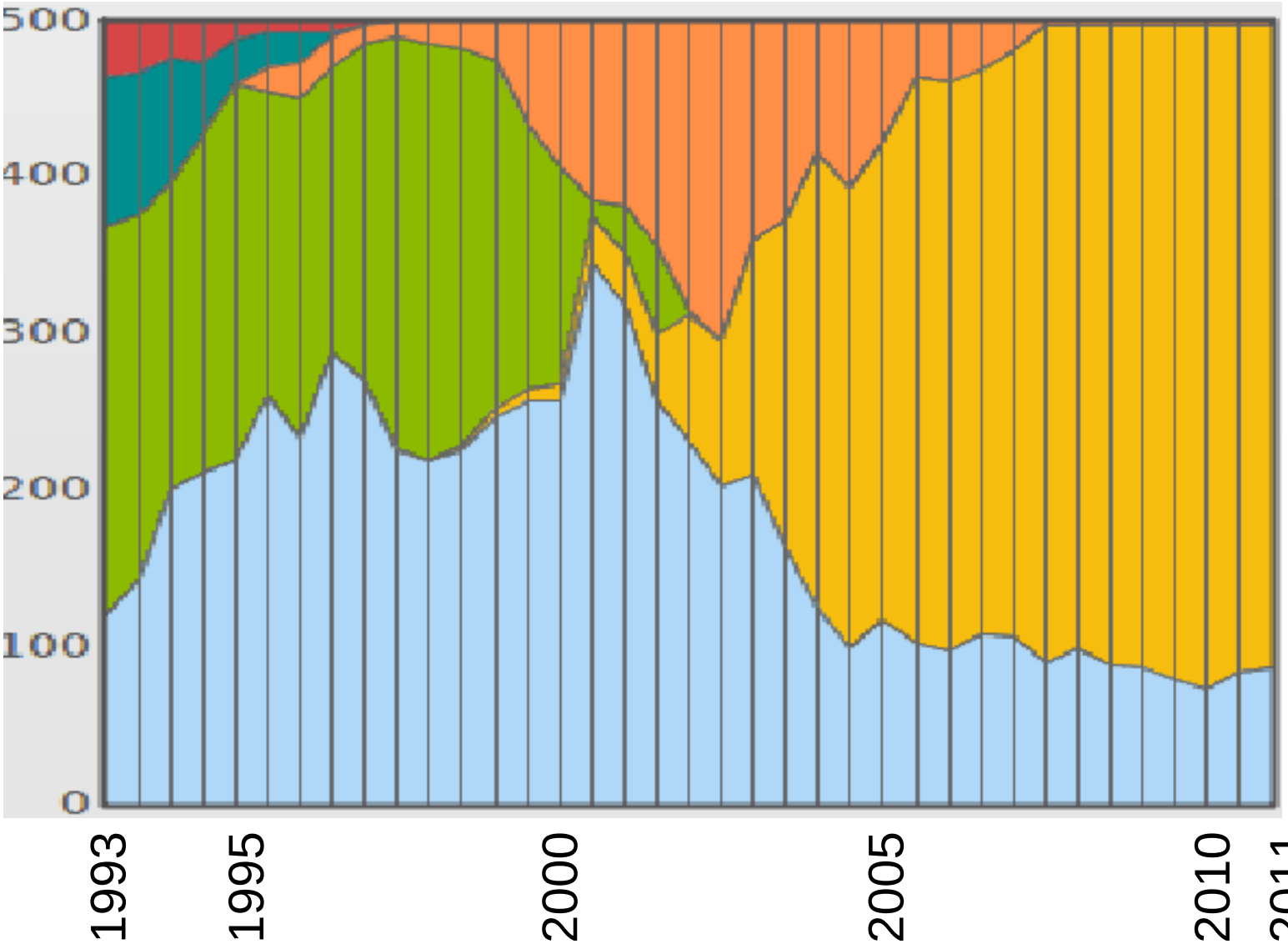
Architektur einer GPU (NVIDIA Fermi)



Programmierung von GPUs (NVIDIA Fermi)

- Aufteilung des Codes in Gruppen (*Warps*) von 32 Threads
- *Warps* werden auf Streaming-Multiprozessoren (SEs) verteilt
- Jeder der beiden *Warp-Scheduler* eines SE führt pro Takt einen Befehl mit 16 Threads aus
  - in SIMD-Manier, d.h., die Cores führen alle denselben Befehl (auf unterschiedlichen Daten) aus oder gar keinen
  - z.B. bei `if-then-else`:
    - erst führen einige Cores den `then`-Zweig aus,
    - dann die anderen Cores den `else`-Zweig
- Threads eines Warps sollten aufeinanderfolgende Speicherzellen adressieren
  - nur dann können Speicherzugriffe zusammengefasst werden

Trends



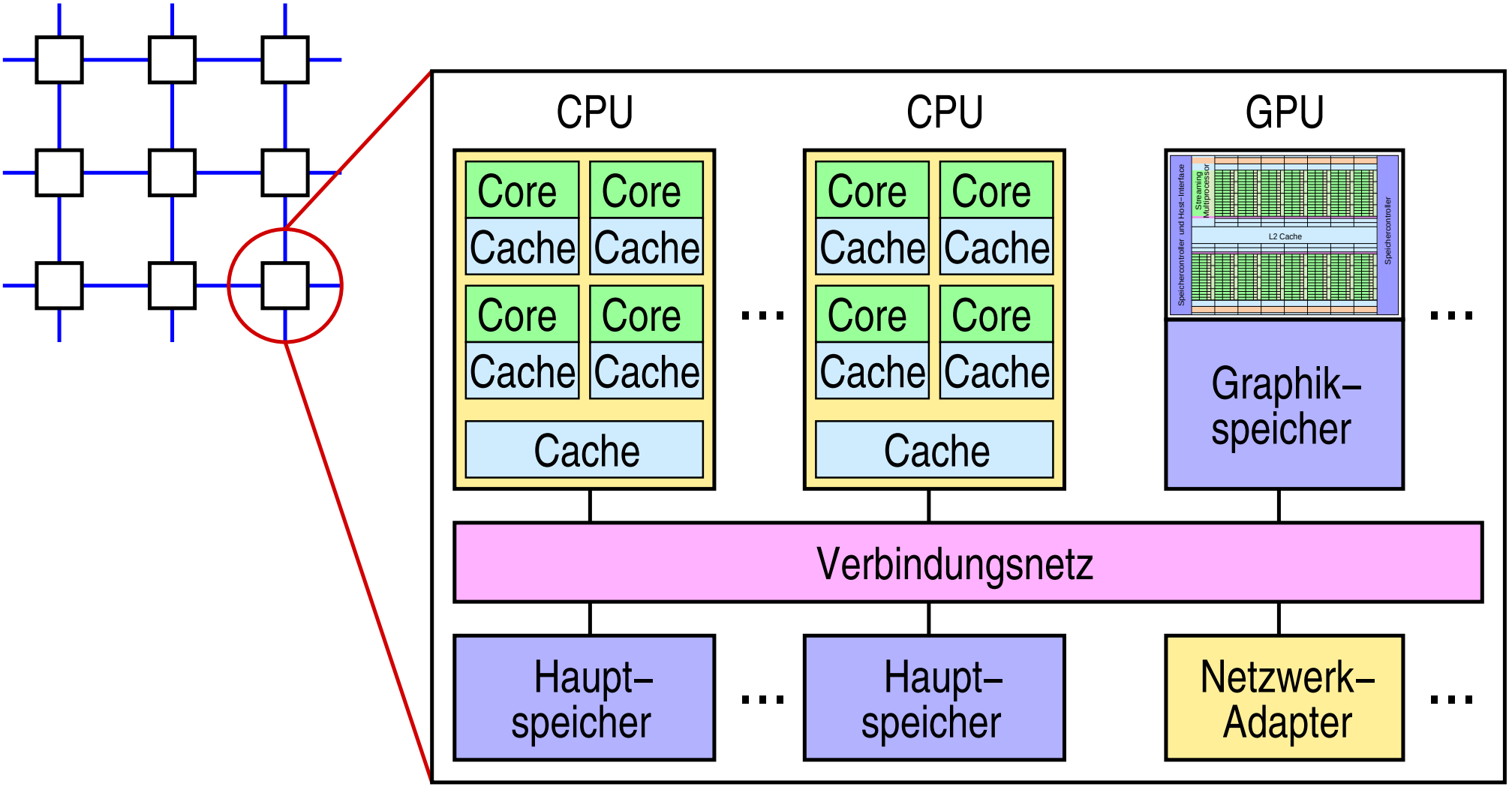
Quelle:
Top500 Liste
www.top500.org

1.4.4 Höchstleistungsrechner ...

Typische Architektur:

- Nachrichtengekoppelte Rechner mit SMP-Knoten und Beschleunigern (z.B. GPUs)
  - auf oberster Ebene: System mit verteiltem Speicher
  - Knoten sind NUMA-Systeme mit teilweise gemeinsamer Cache-Hierarchie
  - zusätzlich ein oder mehrere Beschleuniger pro Knoten
- Kompromiß aus Skalierbarkeit, Programmierbarkeit und Leistung
- Programmierung mit hybridem Programmiermodell
  - Nachrichtenaustausch zwischen Knoten (manuell, MPI)
  - gemeinsamer Speicher auf den Knoten (compilerunterstützt, z.B. OpenMP)
  - ggf. weiteres Programmiermodell für Beschleuniger

Typische Architektur: ...



Nachfolgend betrachtet:

- Gemeinsamer Speicher
- Nachrichtenaustausch
- Verteilte Objekte
- Datenparallele Sprachen

- Liste ist nicht vollständig (z.B. Datenflußmodell, PGAS)

1.5.1 Gemeinsamer Speicher

- Leichtgewichtige Prozesse (Threads) teilen sich einen gemeinsamen virtuellen Adreßraum
- „Einfacheres“ paralleles Programmiermodell
  - alle Threads haben Zugriff auf alle Daten
  - auch theoretisch gut untersucht (PRAM-Modell)
- Vornehmlich bei speichergekoppelten Rechnern
  - aber (mit starken Leistungseinbußen) auch auf nachrichtengekoppelten Rechnern realisierbar
    - *Virtual Shared Memory* (VSM)
- Beispiele:
  - PThreads, Java Threads
  - Intel Threading Building Blocks (TBB)
  - OpenMP (☞ **2.3**)

Beispiel für Datenaustausch

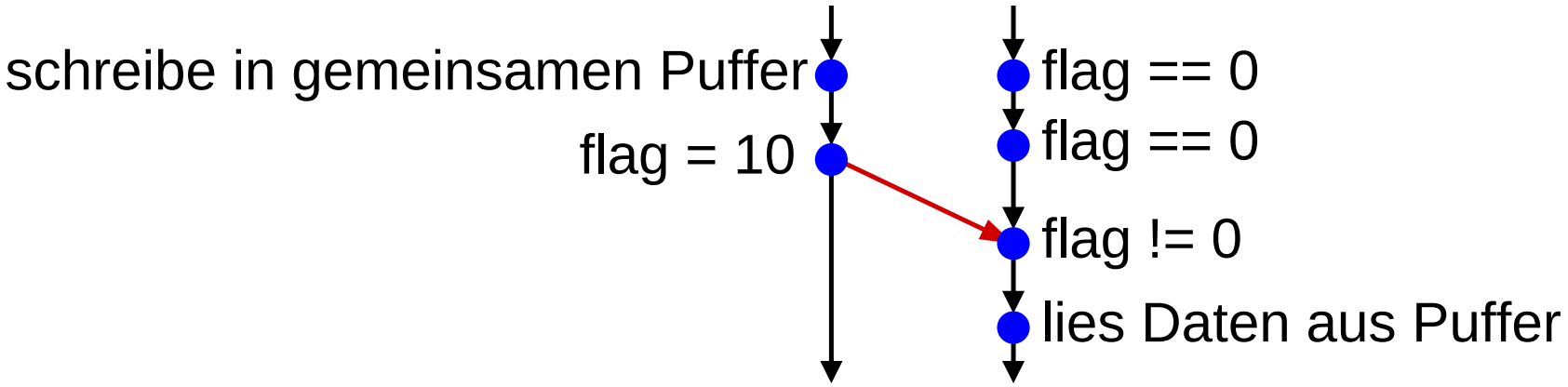
Erzeuger−Thread

```
for (i=0; i<size; i++)
   buffer[i] = produce();
flag = size;
```

Verbraucher−Thread

```
while(flag==0);
for (i=0; i<flag; i++)
   consume(buffer[i]);
```

Zeitlicher Ablauf:



1.5.2 Nachrichtenaustausch

- Prozesse mit getrennten Adreßräumen
- Bibliotheksfunktionen zum Versenden und Empfangen von Nachrichten
  - (informeller) Standard für parallele Programmierung: MPI (*Message Passing Interface*, ☞ **3.2**)
- Vornehmlich bei nachrichtengekoppelten Rechnern
  - aber auch bei speichergekoppelten Rechnern nutzbar
- Komplizierteres Programmiermodell
  - explizite Datenverteilung / expliziter Datentransfer
  - i.d.R. keine Compiler-/Sprachunterstützung
    - Parallelisierung erfolgt vollständig manuell

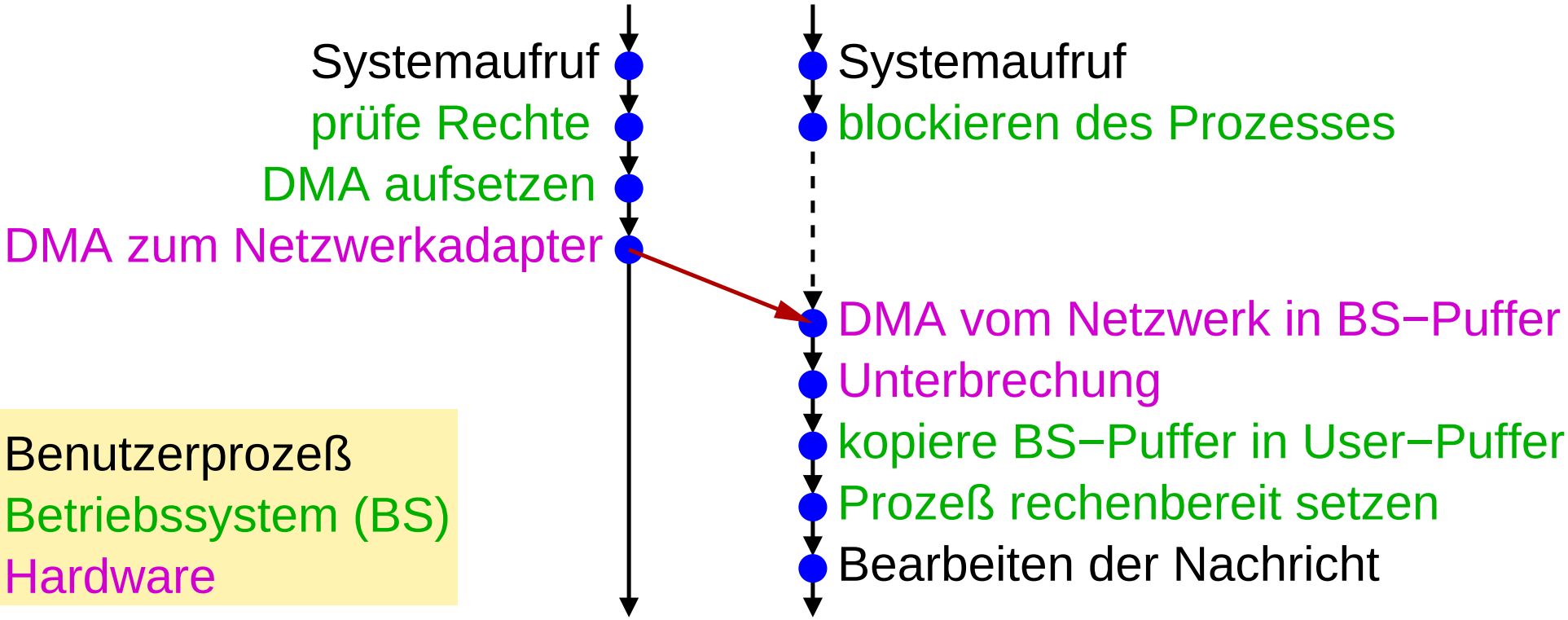
Beispiel für Datenaustausch

Erzeuger−Prozeß

```
send(receiver,
     &buffer, size);
```

Verbraucher−Prozeß

```
receive(&buffer,
        buffer_length);
```



1.5.3 Verteilte Objekte

- Basis: (rein) objektorientierte Programmierung
  - Zugriff auf Daten **nur** über Methodenaufrufe
- Dann: Objekte können auf verschiedene Adreßräume (Rechner) verteilt werden
  - bei Objekterzeugung: zusätzlich Angabe einer Rechners
  - Objektreferenz identifiziert dann auch diesen Rechner
  - Methodenaufrufe über RPC-Mechanismus
    - z.B. *Remote Method Invocation* (RMI) in Java
  - mehr dazu: Vorlesung „Verteilte Systeme“
- Verteilte Objekte an sich ermöglichen noch keine Parallelverarbeitung
  - zusätzliche Konzepte / Erweiterungen erforderlich
    - z.B. Threads, asynchroner RPC, *Futures*

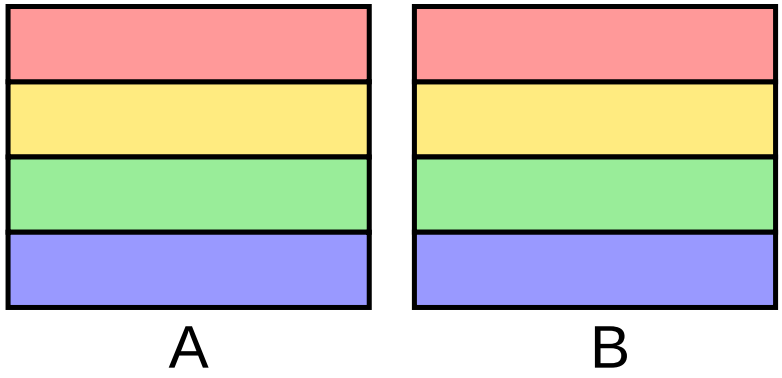
1.5.4 Datenparallele Sprachen

- Ziel: Unterstützung von Datenparallelität
- Sequentieller Code wird mit Compilerdirektiven ergänzt
  - Festlegung einer Aufteilung von Datenstrukturen (i.a.: Arrays) auf Prozessoren
- Compiler erzeugt automatisch Code für Synchronisation bzw. Kommunikation
  - Operationen werden auf dem Prozessor ausgeführt, der Ergebnisvariable besitzt (*Owner computes*-Regel)
- Beispiel: HPF (*High Performance Fortran*)
- Trotz einfacher Programmierung nicht wirklich erfolgreich
  - nur für eingeschränkte Anwendungsklasse geeignet
  - gute Leistung erfordert viel manuelle Optimierung

Beispiel zu HPF

```
REAL A(N,N), B(N,N)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(:,:) WITH A(:,:)

DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I,J) + B(J,I)
  END DO
END DO
```

Aufteilung bei 4 Prozessoren:

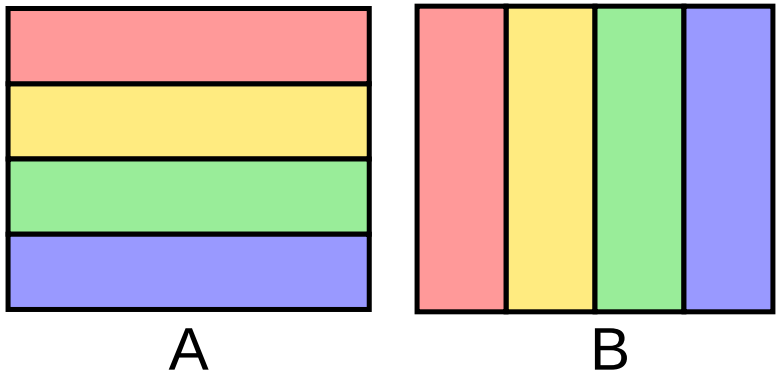


- Prozessor 0 führt Berechnungen für $\texttt{I} = 1 \,..\, N/4$ aus
- Problem im Beispiel: viel Kommunikation erforderlich
  - `B` müßte hier anders aufgeteilt werden

Beispiel zu HPF

```
REAL A(N,N), B(N,N)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(j,i) WITH A(i,j)

DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I,J) + B(J,I)
  END DO
END DO
```

Aufteilung bei 4 Prozessoren:



- Prozessor 0 führt Berechnungen für $\texttt{I} = 1 \,..\, N/4$ aus
- Keine Kommunikation mehr erforderlich
  - aber evtl. Umverteilung von `B`!

1.6 Einordnung der Vorlesung

- Explizite Parallelität
- Prozeß- und Blockebene
- Grob- und mittelkörnige Granularität

- MIMD-Rechner (mit SIMD-Erweiterungen)

- Programmiermodelle:
  - gemeinsamem Speicher
  - Nachrichtenaustausch

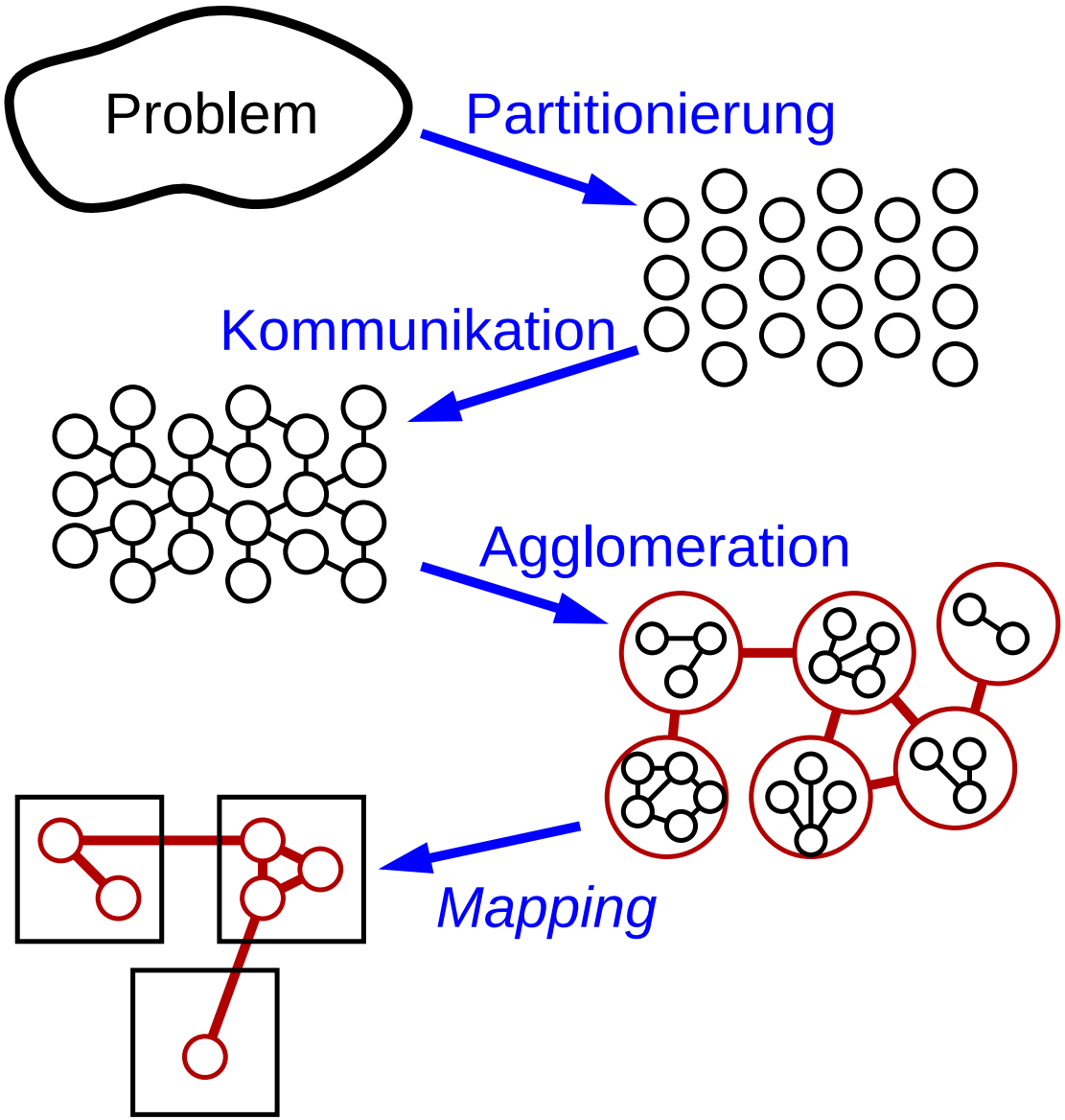
1.7 Ein Entwurfsprozeß für parallele Programme

Vier Entwurfsschritte:

1. Partitionierung
   - ➥ Aufteilung des Problems in viele Tasks
2. Kommunikation
   - ➥ Spezifikation des Informationsflusses zw. den Tasks
   - ➥ Festlegen der Kommunikationsstruktur
3. Agglomeration
   - ➥ Leistungsbewertung (Tasks, Kommunikationsstruktur)
   - ➥ ggf. Zusammenfassung von Tasks zu größeren Tasks
4. *Mapping*
   - ➥ Abbildung der Tasks auf Prozessoren

(Nach Foster: *Designing and Building Parallel Programs*, Kap. 2)

1.7 Ein Entwurfsprozeß für parallele Programme ...

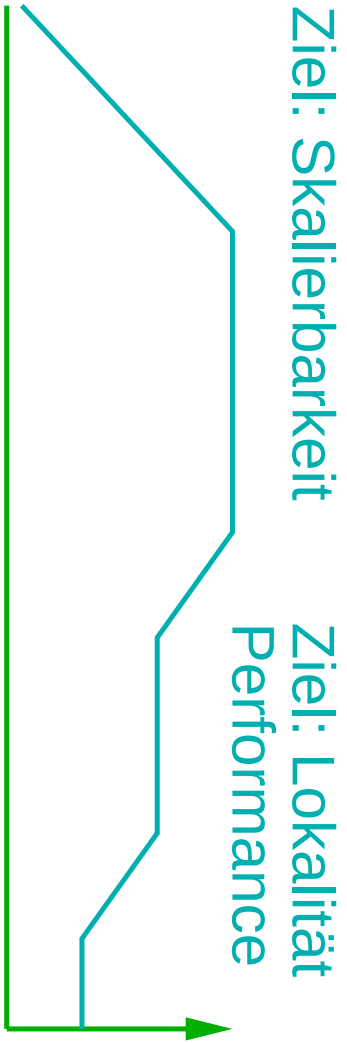


Zerlegung des Pro−blems in möglichst viele kleine Tasks

Datenaustausch zwischen Tasks

Zusammenfassung von Tasks

Zuweisung an Prozessoren



1.7.1 Partitionierung

- Ziel: Aufteilen des Problems in möglichst viele Tasks

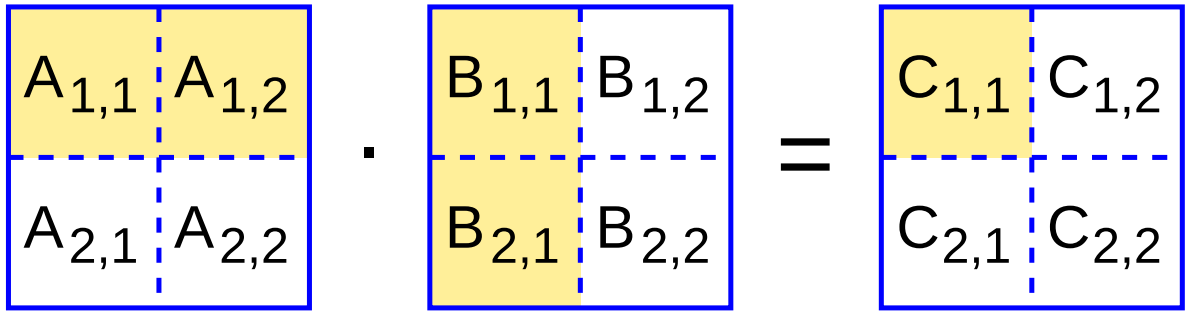
Datenpartitionierung (Datenparallelität)

- Tasks beschreiben **identische Aufgaben** für einen **Teil** der Daten
- i.a. hoher Parallelitätsgrad möglich
- aufgeteilt werden können:
  - Eingabedaten
  - Ausgabedaten
  - Zwischendaten
- ggf. auch rekursive Aufteilung (*divide and conquer*)
- Spezialfall: Suchraum-Aufteilung bei Suchproblemen

Beispiel: Matrix-Multiplikation

- Produkt $C = A \cdot B$ zweier quadratischer Matrizen
  - $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, für alle $i, j = 1 \ldots n$

- Die Formel gilt auch, wenn statt einzelner Elemente quadratische Untermatrizen A_{ik}, B_{kj}, C_{ij} betrachtet werden
  - Block-Matrix-Algorithmen:



$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

Beispiel: Matrix-Multiplikation ...

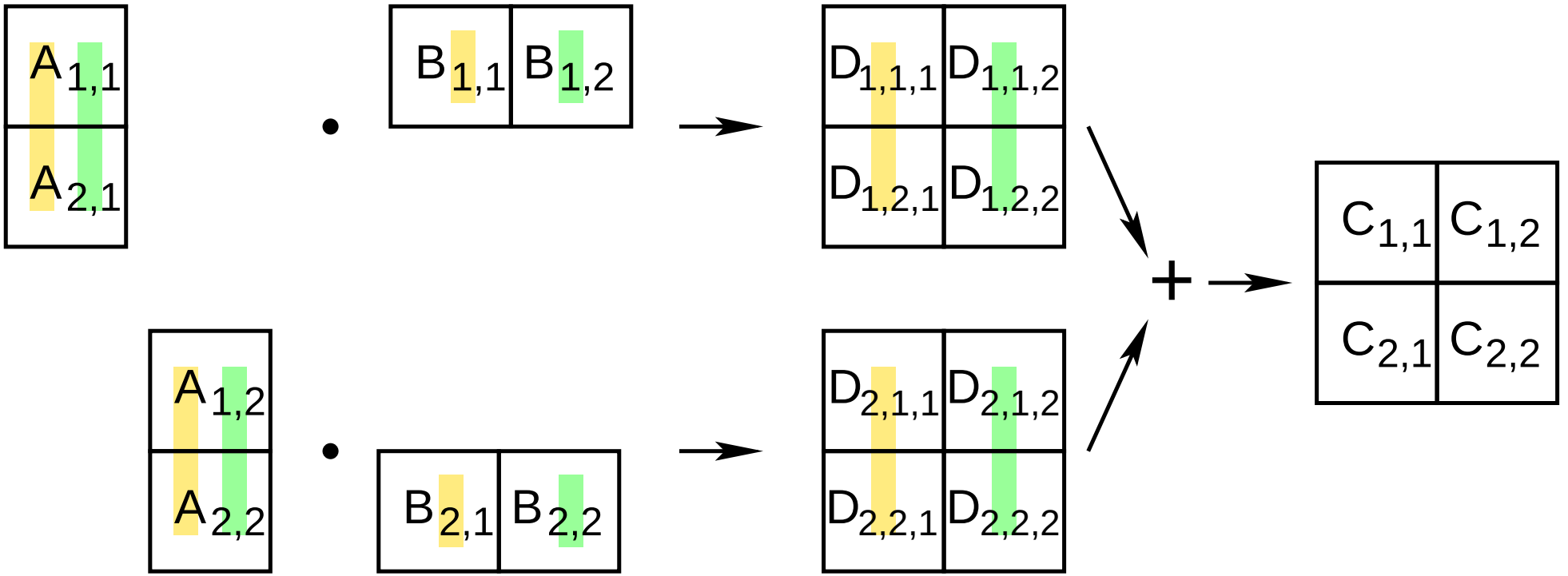
- Aufteilung der Ausgabedaten: jeder Task berechnet eine Untermatrix von C
- Z.B. Aufteilung von C in vier Untermatrizen

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

- Ergibt vier unabhängige Tasks:
  1. $C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$
  2. $C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$
  3. $C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$
  4. $C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$

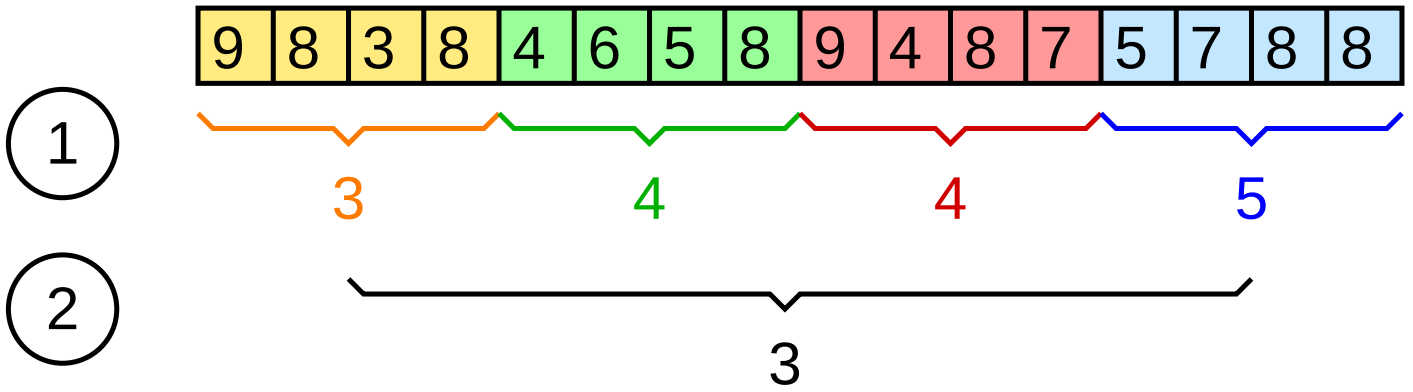
Beispiel: Matrix-Multiplikation $A \cdot B \rightarrow C$

- Aufteilung der Zwischendaten (höherer Parallelitätsgrad)
  - hier: 8 Multiplikationen von Untermatrizen

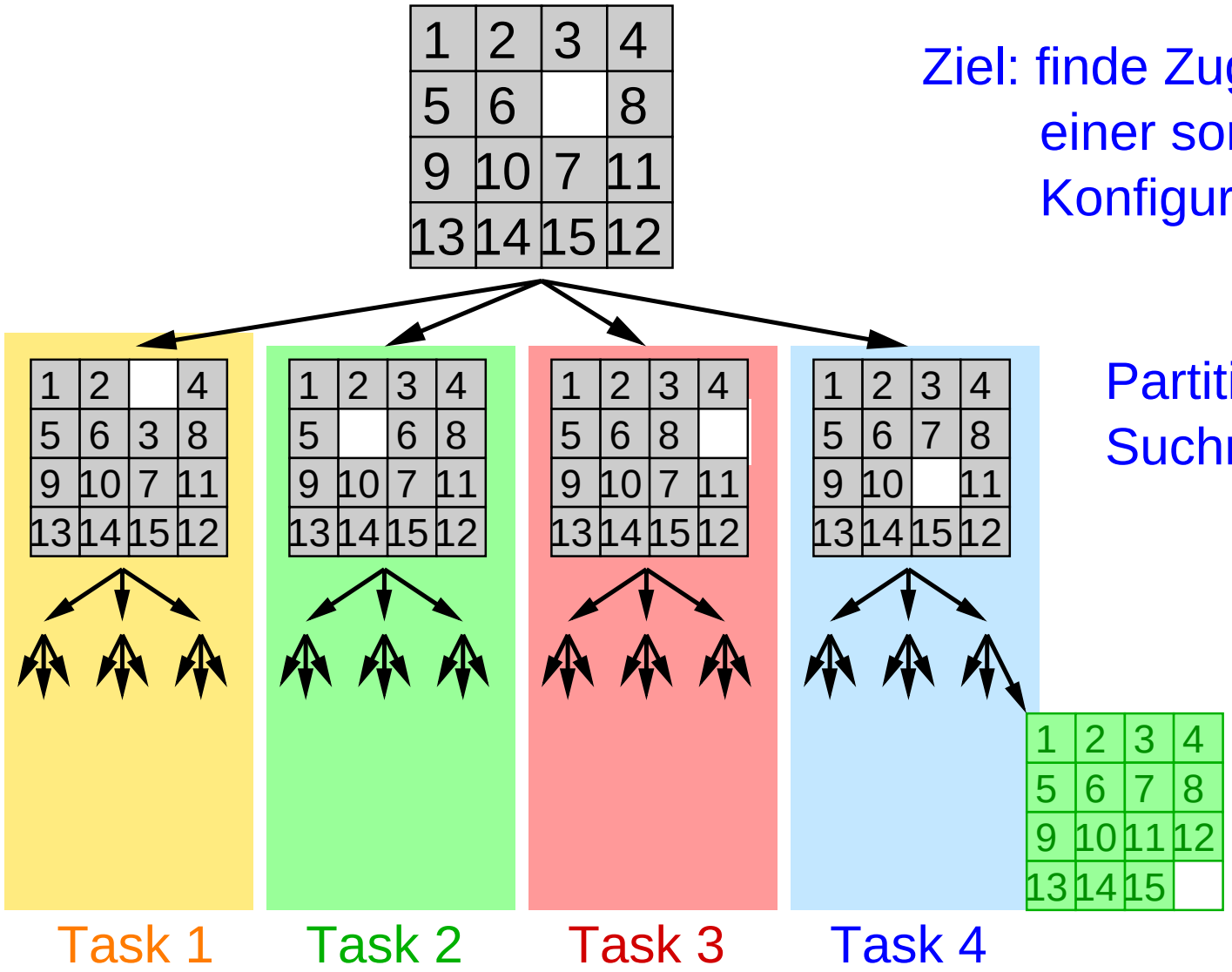


Beispiel: Minimum eines Arrays

- Aufteilung der Eingabedaten
  - Berechnung der einzelnen Minima
  - Danach: Berechung des globalen Minimums



Beispiel: Schiebepuzzle (Suchraum-Aufteilung)



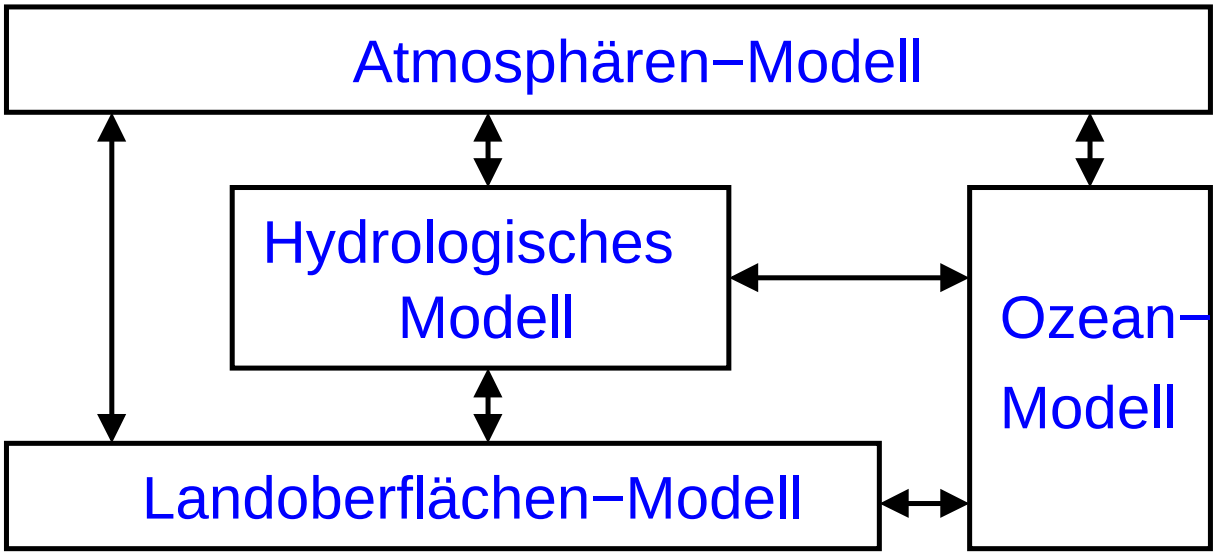
Ziel: finde Zugfolge, die zu einer sortierten Konfiguration führt.

Partitionierung des Suchraums

Lösung gefunden:
Ende!

Funktionspartitionierung (Taskparallelität)

- Tasks sind **unterschiedliche** Teilaufgaben (Bearbeitungsschritte) des Problems
- Z.B.: Klimamodell



- Tasks können nebenläufig oder als Pipeline arbeiten
- max. Gewinn: Zahl der Teilaufgaben (i.a. gering)
- oft als Ergänzung zu Datenpartitionierung

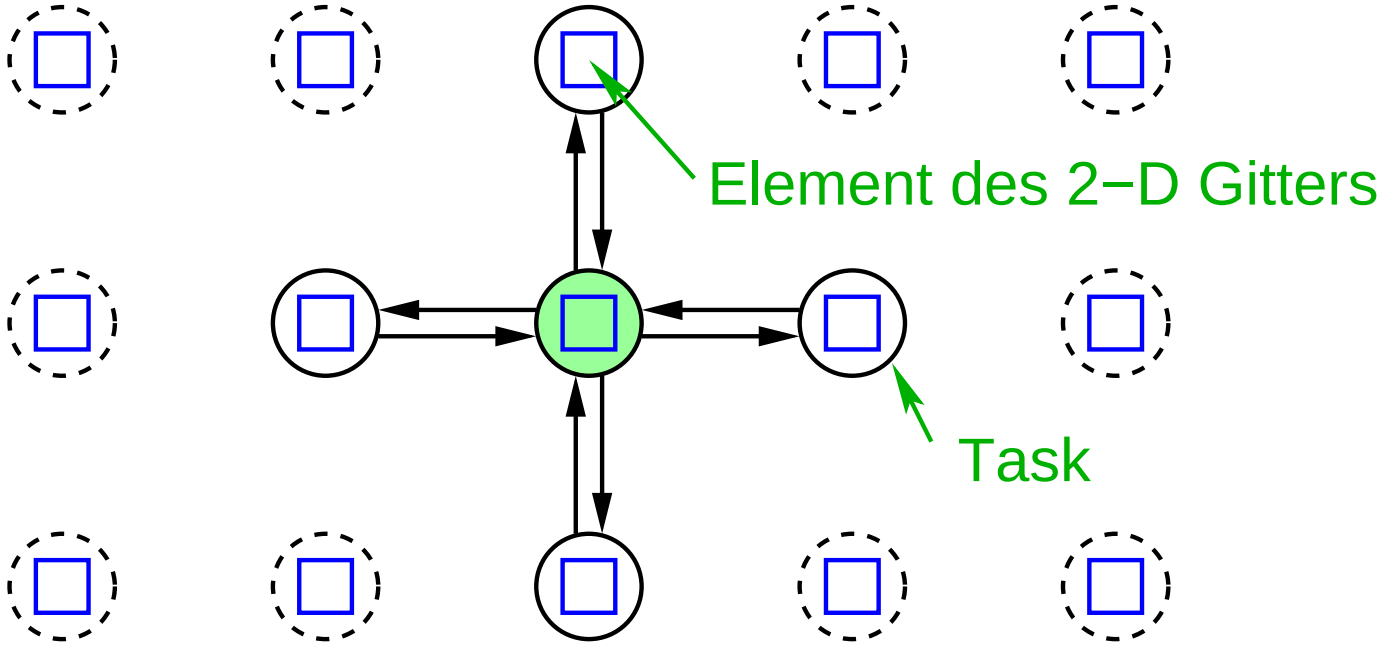
1.7.2 Kommunikation

- Zweistufiges Vorgehen
  - Definition der Kommunikationsstruktur
    - wer muß mit wem Daten austauschen?
    - bei Datenpartitionierung teilweise komplex
    - bei Funktionspartitionierung meist einfach
  - Definition der zu versendenden Nachrichten
    - welche Daten müssen wann ausgetauscht werden?
    - Berücksichtigung von Datenabhängigkeiten

Unterschiedliche Kommunikationsmuster:

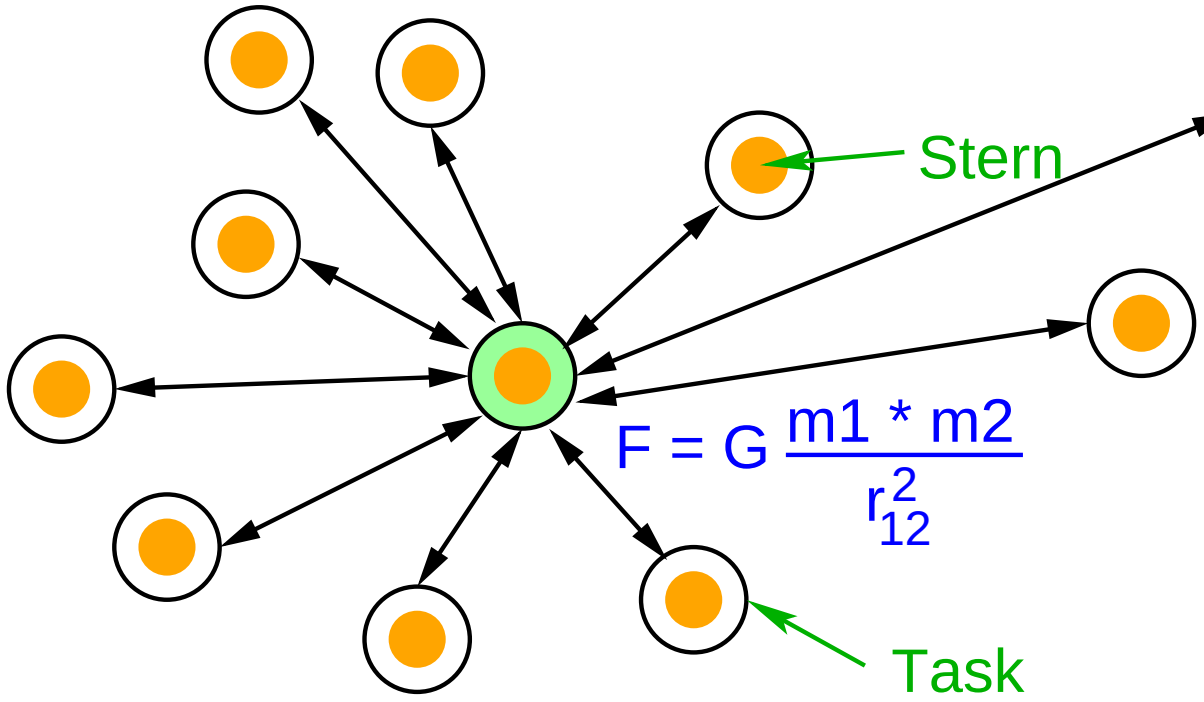
- Lokale vs. globale Kommunikation
  - lokal: Task kommuniziert nur mit kleiner Menge anderer Tasks (ihren „Nachbarn“)
  - global: Task kommuniziert mit vielen Tasks
- Strukturierte vs. unstrukturierte Kommunikation
  - strukturiert: regelmäßige Struktur, z.B. Gitter, Baum
- Statische vs. dynamische Kommunikation
  - dynamisch: Kommunikationsstruktur ändert sich zur Laufzeit, abhängig von berechneten Daten
- Synchrone vs. asynchrone Kommunikation
  - Task, die Daten besitzt, weiß nicht, wann andere Tasks auf die Daten zugreifen müssen

Beispiel für lokale Kommunikation: *Stencil*-Algorithmen



- Hier: 5-Punkt *Stencil* (auch andere möglich)
- Beispiele: Jacobi bzw. Gauss-Seidel-Verfahren, Filter in der Bildverarbeitung, ...

Beispiel für globale Kommunikation: N-Körper-Problem



Bewegung der Sterne in einem Sternhaufen:

1) Kräfte
2) Beschleunigungen
3) Geschwindigkeiten

- Kraft auf einen Stern in einem Sternhaufen abhängig von Masse und Ort aller anderen Sterne
  - ggf. Näherung: Einschränkung auf relativ nahe Sterne
    - dann dynamische Kommunikation

Parallelverarbeitung

WS 2015/16

29.10.2015

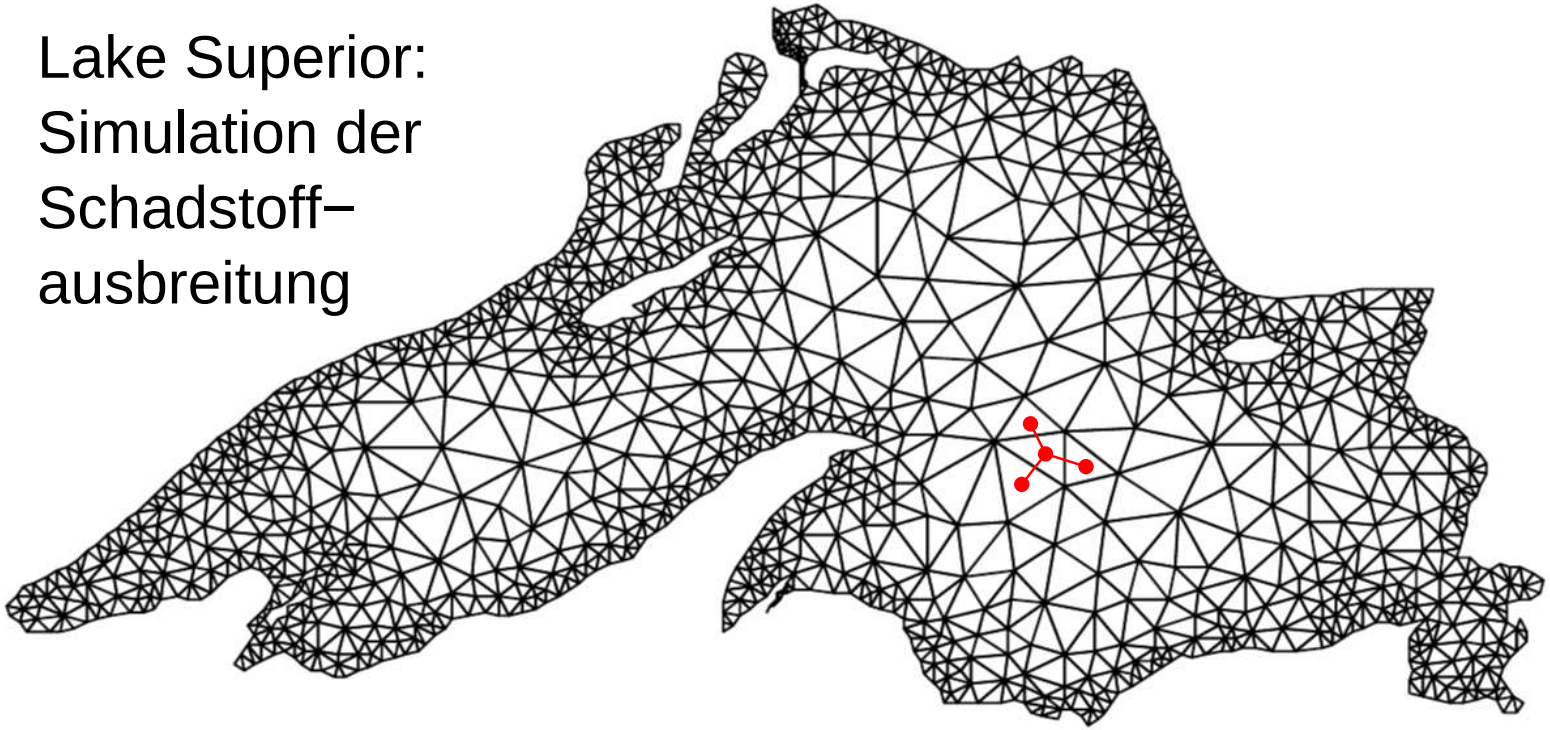
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Beispiele für strukturierte / unstrukturierte Kommunikation

- Strukturiert: *Stencil*-Algorithmen
- Unstrukturiert: „unstrukturierte Gitter“

Lake Superior: Simulation der Schadstoff–ausbreitung



- Gitterpunkte unterschiedlich dicht gesetzt
- Kanten: Nachbarschaftsverhältnisse (Kommunikation)

1.7.3 Agglomeration

- Bisher: abstrakte parallele Algorithmen
- Jetzt: konkretere Formulierung für reale Rechner
  - begrenzte Prozessorzahl
  - Kosten für Kommunikation, Prozeßerzeugung, Prozeßwechsel, ...
- Ziele:
  - Reduktion der Kommunikationskosten
    - Zusammenfassung von Tasks
    - Replikation von Daten bzw. Berechnungen
  - Beibehalten der Flexibilität
    - genügend feinkörnige Granularität für *Mapping*-Phase

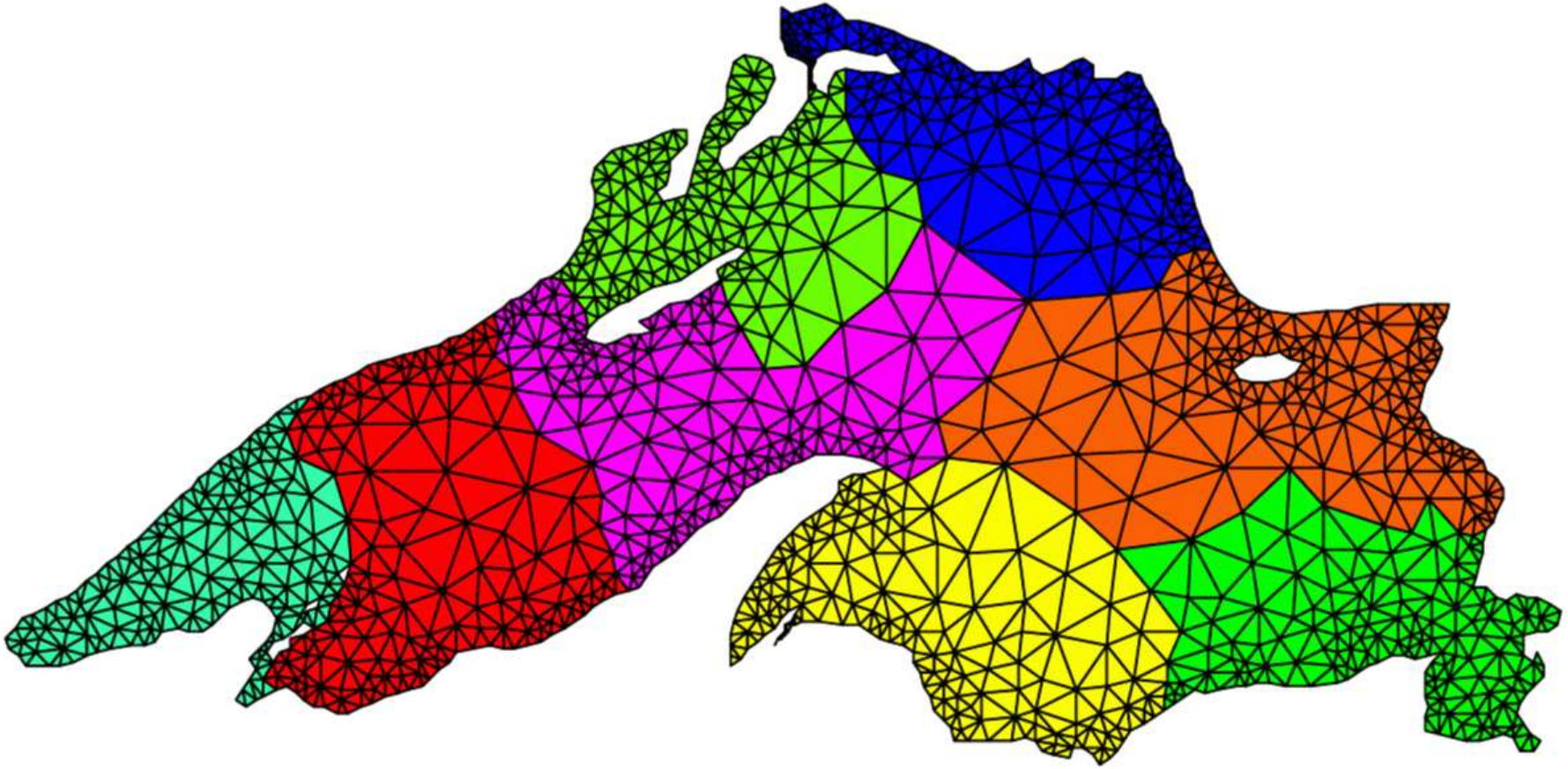
1.7.4 *Mapping*

- Aufgabe: Zuweisung von Tasks an verfügbare Prozessoren
- Ziel: Minimierung der Ausführungszeit
- Zwei (konfligierende) Strategien:
  - nebenläufig ausführbare Tasks auf unterschiedliche Prozessoren
    - hoher Parallelitätsgrad
  - kommunizierende Tasks auf denselben Prozessor
    - höhere Lokalität (weniger Kommunikation)
- Nebenbedingung: Lastausgleich
  - (etwa) gleicher Rechenaufwand für jeden Prozessor
- Das *Mapping*-Problem ist NP-vollständig

Mapping-Varianten

- Statisches Mapping
  - feste Zuweisung von Tasks zu Prozessoren bei Programmstart
  - bei Algorithmen auf Arrays bzw. kartesischen Gittern
    - oft manuell: blockweise bzw. zyklische Verteilung
  - bei unstrukturierten Gittern
    - Graph-Partitionierungsalgorithmen, z.B. *Greedy*, *Recursive Coordinate Bisection*, *Recursive Spectral Bisection*, ...
- Dynamisches Mapping (dynamischer Lastausgleich)
  - Zuweisung von Tasks an Prozessoren zur Laufzeit
  - Varianten:
    - Tasks bleiben bis zum Ende auf ihrem Prozessor
    - Verschiebung während der Ausführung möglich

Beispiel: statisches *Mapping* bei unstrukturiertem Gitter



- (Etwa) gleich viele Gitterpunkte pro Prozessor
- Kurze Grenzlinien: wenig Kommunikation

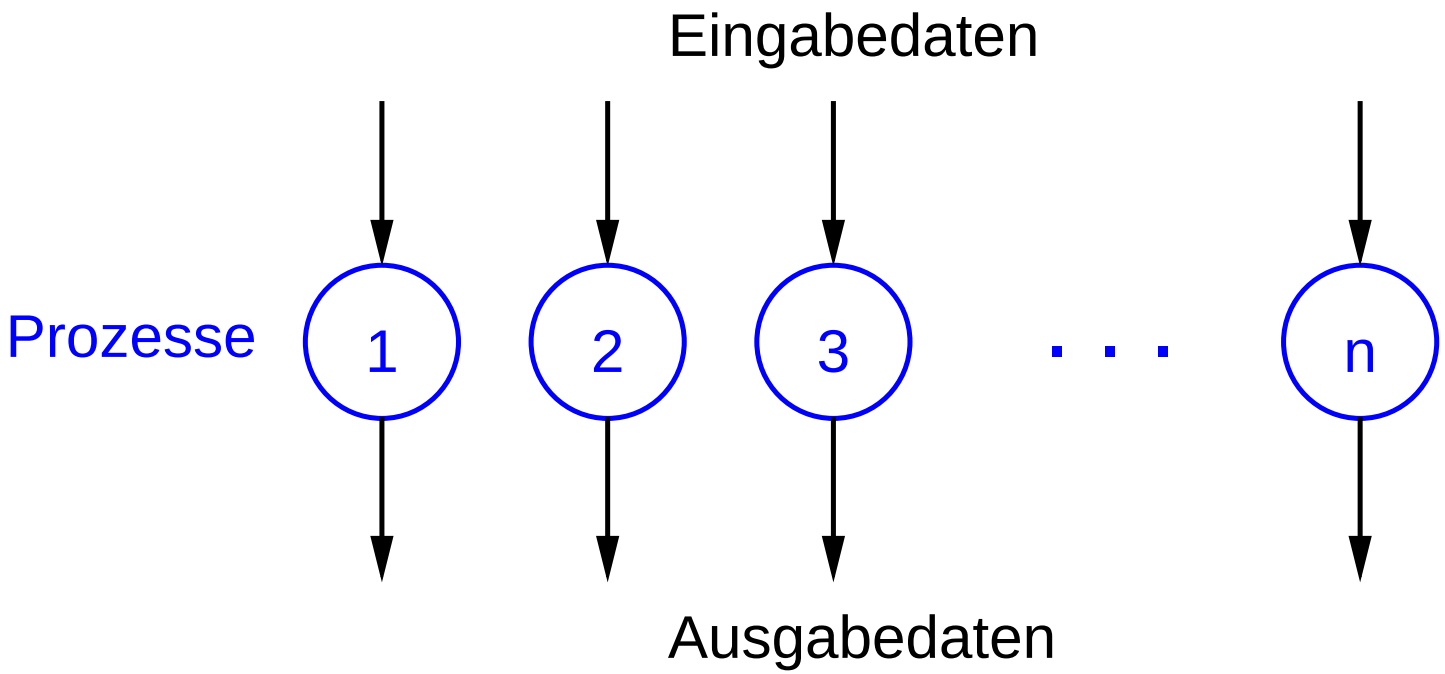
1.8 Organisationsformen paralleler Programme

- Modelle bzw. Muster (*Pattern*) für parallele Programme

1.8.1 *Embarrassingly Parallel*

- Zu lösende Aufgabe kann in eine Menge **vollständig unabhängiger** Teilaufgaben zerlegt werden
- Alle Teilaufgaben können parallel gelöst werden
- Keinerlei Datenaustausch (Kommunikation) zwischen den parallelen Threads / Prozessen erforderlich
- Ideale Situation!
  - bei Einsatz von n Prozessoren wird die Aufgabe (normalerweise) n-mal schneller gelöst
  - (zum Nachdenken: warum nur normalerweise?)

Veranschaulichung

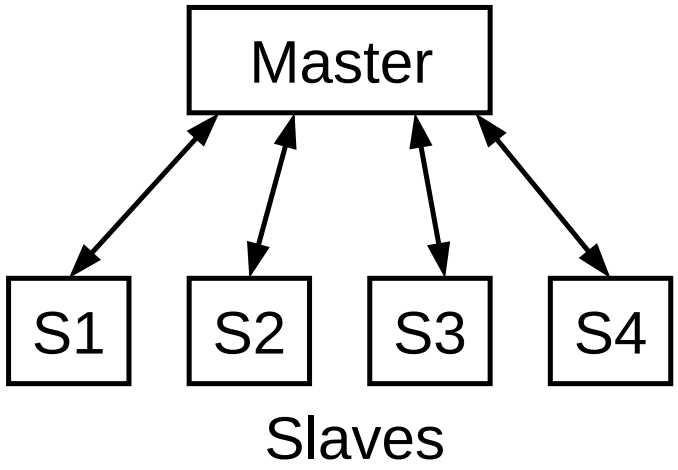


Beispiele für *embarrassingly parallel* Probleme

- Berechnung von Animationen
  - Trickfilme, Zoom in Mandelbrot-Menge, ...
  - jedes Bild kann unabhängig berechnet werden
- Parameterstudien
  - mehrere / viele Simulationen mit unterschiedlichen Eingangsparametern
  - z.B. Wettervorhersage mit Berücksichtigung der Meßfehler, Strömungssimulation zur Optimierung einer Tragfläche, ...

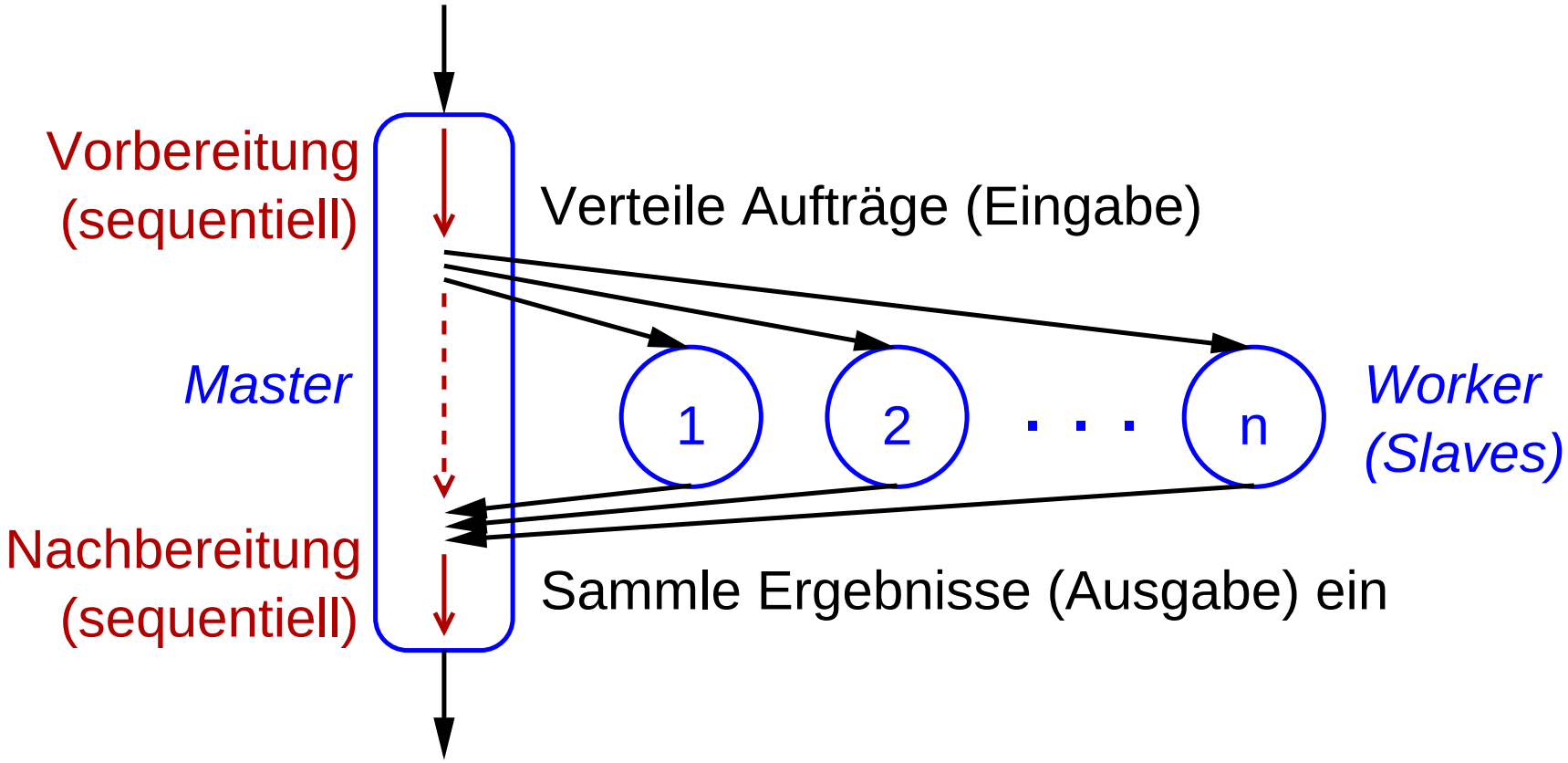
1.8.2 *Master-Slave*-Modell (*Manager-Worker*-Modell)

- *Master*-Prozeß erzeugt Tasks und teilt sie an *Slave*-Prozesse zu
  - auch mehrere *Master* möglich
  - auch Hierarchie möglich: *Slave* ist selbst wieder *Master* eigener *Slaves*
- *Master* kann weitere Tasks erzeugen, während *Slaves* arbeiten
- *Master* kann zum Flaschenhals werden
- *Master* sollte Ergebnisse asynchron (nichtblockierend) empfangen können



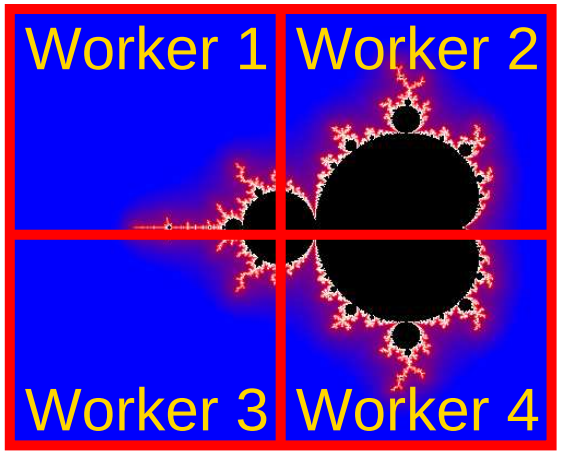
Typische Anwendung

- Oft ist ein Teil einer Aufgabe optimal parallelisierbar
- Im einfachsten Fall dann folgender Ablauf:

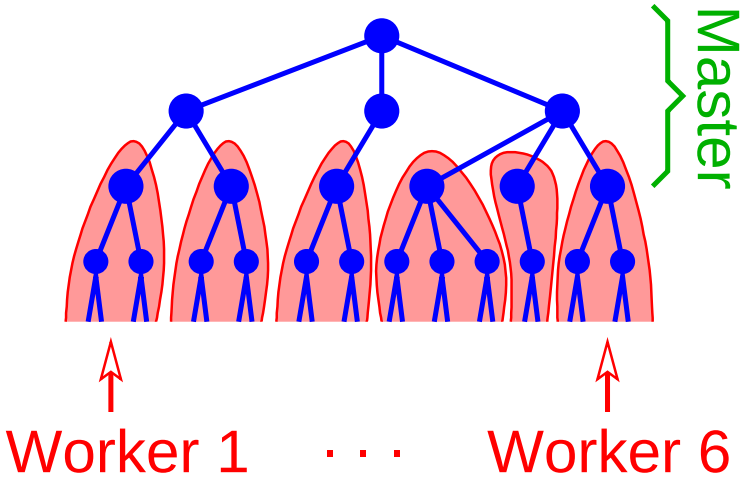


Beispiele

- Bilderzeugung und -verarbeitung
  - Master zerlegt Bild in Bereiche, jeder Bereich wird von einem Worker bearbeitet

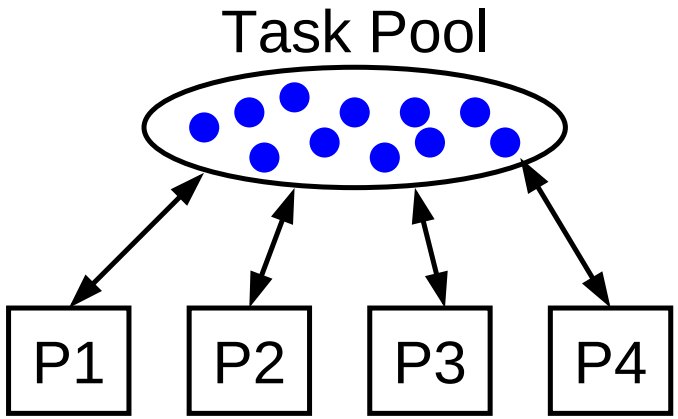


- Baumsuche
  - Master durchläuft Baum bis zu einer bestimmten Tiefe, Worker bearbeiten die Teilbäume



1.8.3 *Work Pool*-Modell (*Task-Pool*-Modell)

- Tasks werden explizit durch Datenstruktur beschrieben
- Zentraler oder verteilter Pool (Liste) von Tasks
  - Prozesse (Threads) holen sich Tasks aus dem Pool
    - i.d.R. wesentlich mehr Tasks als Prozesse
    - gute Lastverteilung möglich
  - Zugriffe müssen synchronisiert werden



- Prozesse können ggf. neue Tasks in den Pool legen
  - z.B. bei *Divide-and-Conquer*

1.8.4 *Divide and Conquer*

- **Rekursive** Aufteilung der Aufgabe in unabhängige Teilaufgaben
- Dynamische Erzeugung von Teilaufgaben
  - durch alle Threads (bzw. Prozesse)
- Problem: Begrenzung der Anzahl von Threads
  - Teilaufgaben nur ab einer gewissen Größe parallel ausführen
  - Verwaltung einer Auftrags-Warteschlange,
    Abarbeitung durch feste Anzahl von Threads

Beispiel: Paralleler Quicksort

Qsort($A_{1..n}$)

Falls $n = 1$: fertig.

Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1..k-1}$)
und Qsort($A_{k..n}$)
parallel aus.

Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

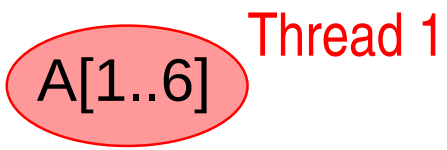
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

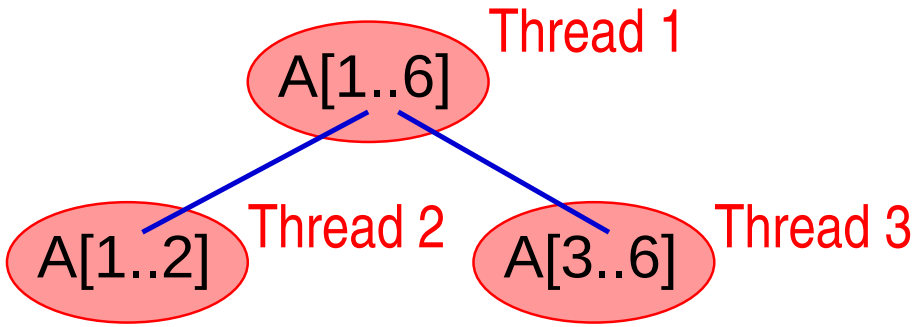
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

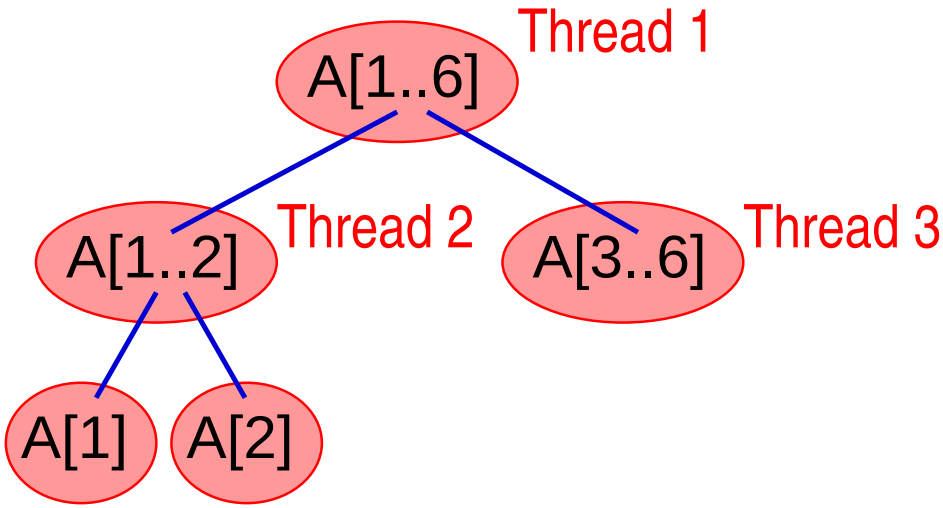
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

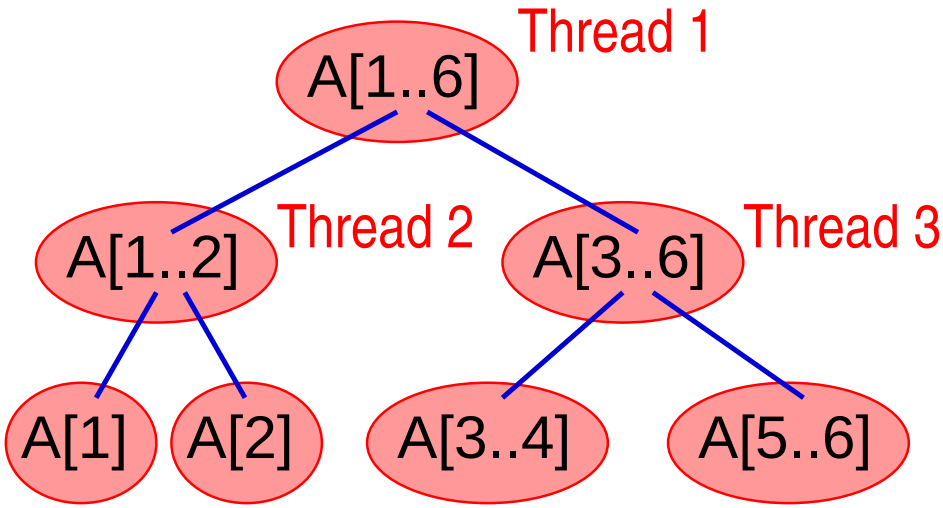
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

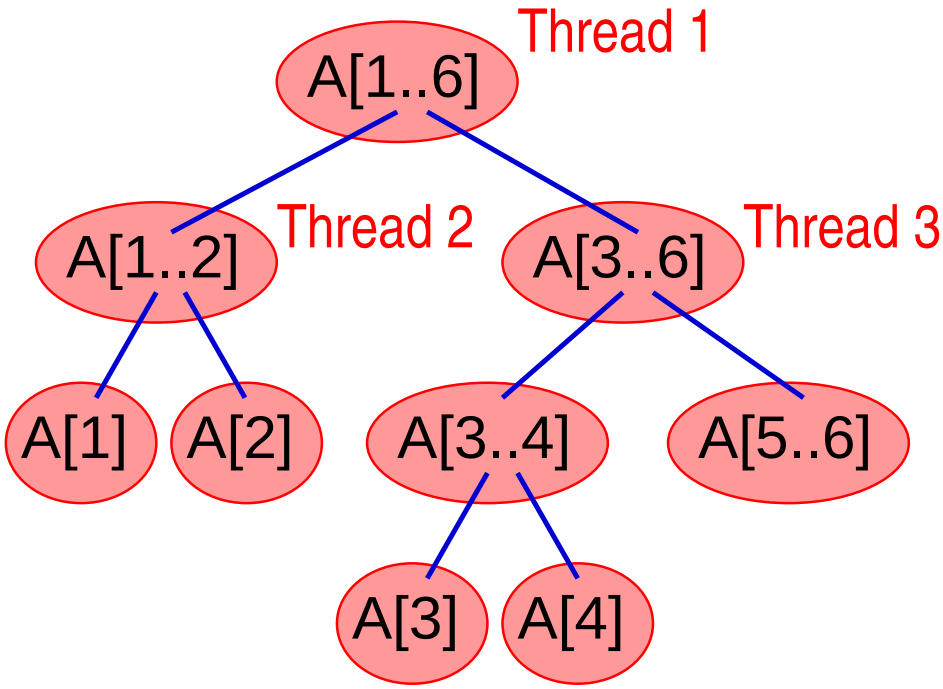
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

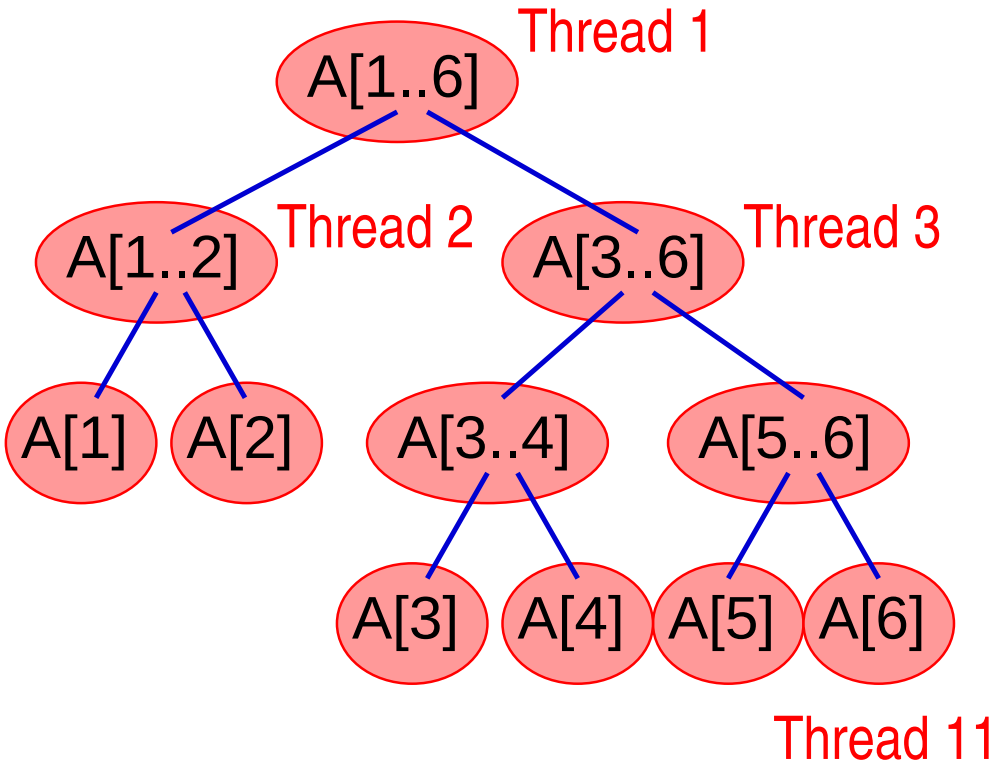
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

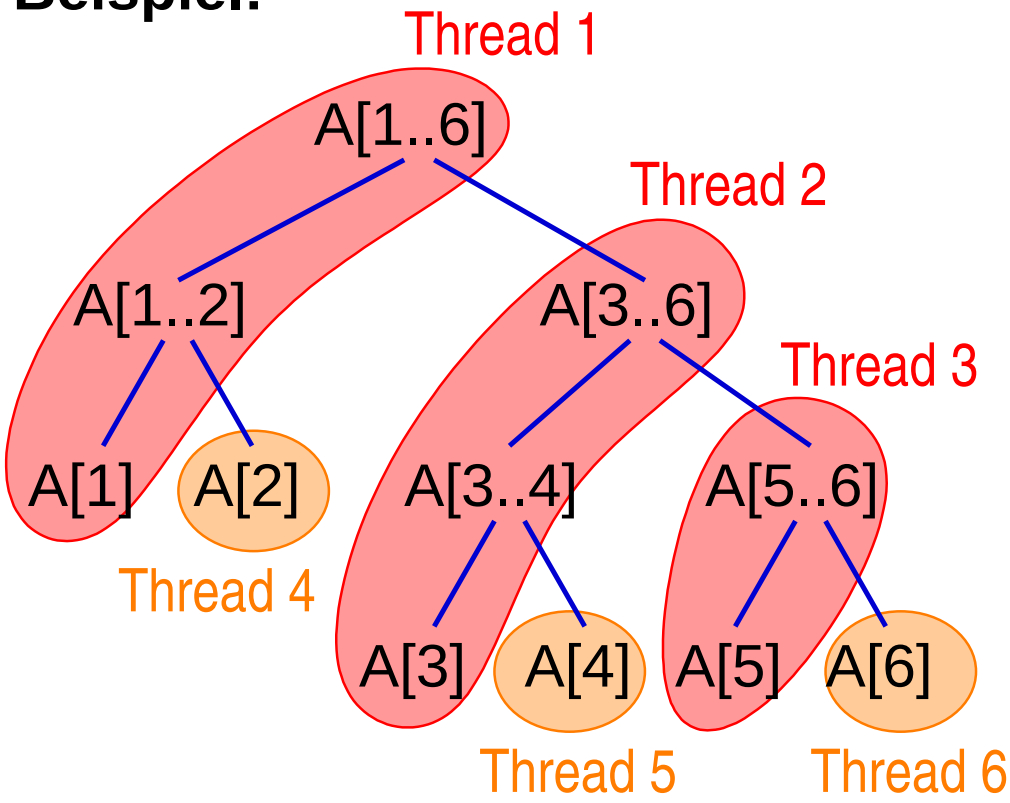
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:



* Annahme: Thread macht ersten Aufruf selbst, erzeugt für den zweiten einen neuen Thread

Beispiel: Paralleler Quicksort

Qsort($A_{1\,..\,n}$)

Falls $n = 1$: fertig.

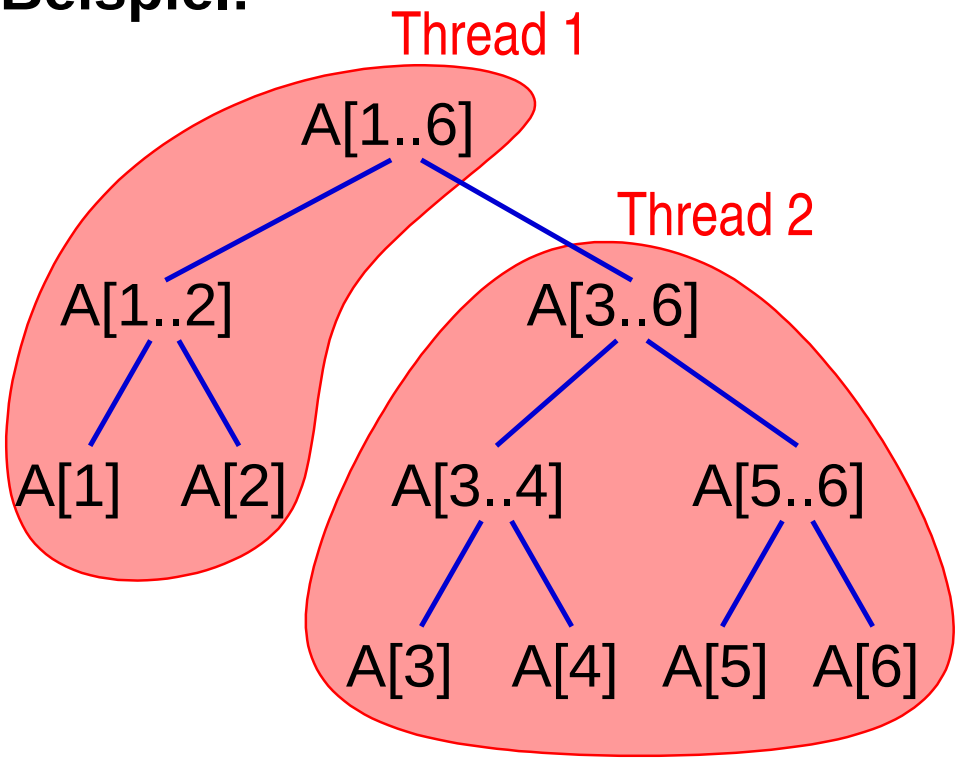
Sonst:

Bestimme Schnittzahl S.

Sortiere A so um,
daß $A_i \leq S$ für $i \in [1, k[$
und $A_i \geq S$ für $i \in [k, n]$.

Führe Qsort($A_{1\,..\,k-1}$)
und Qsort($A_{k\,..\,n}$)
parallel aus.

Beispiel:

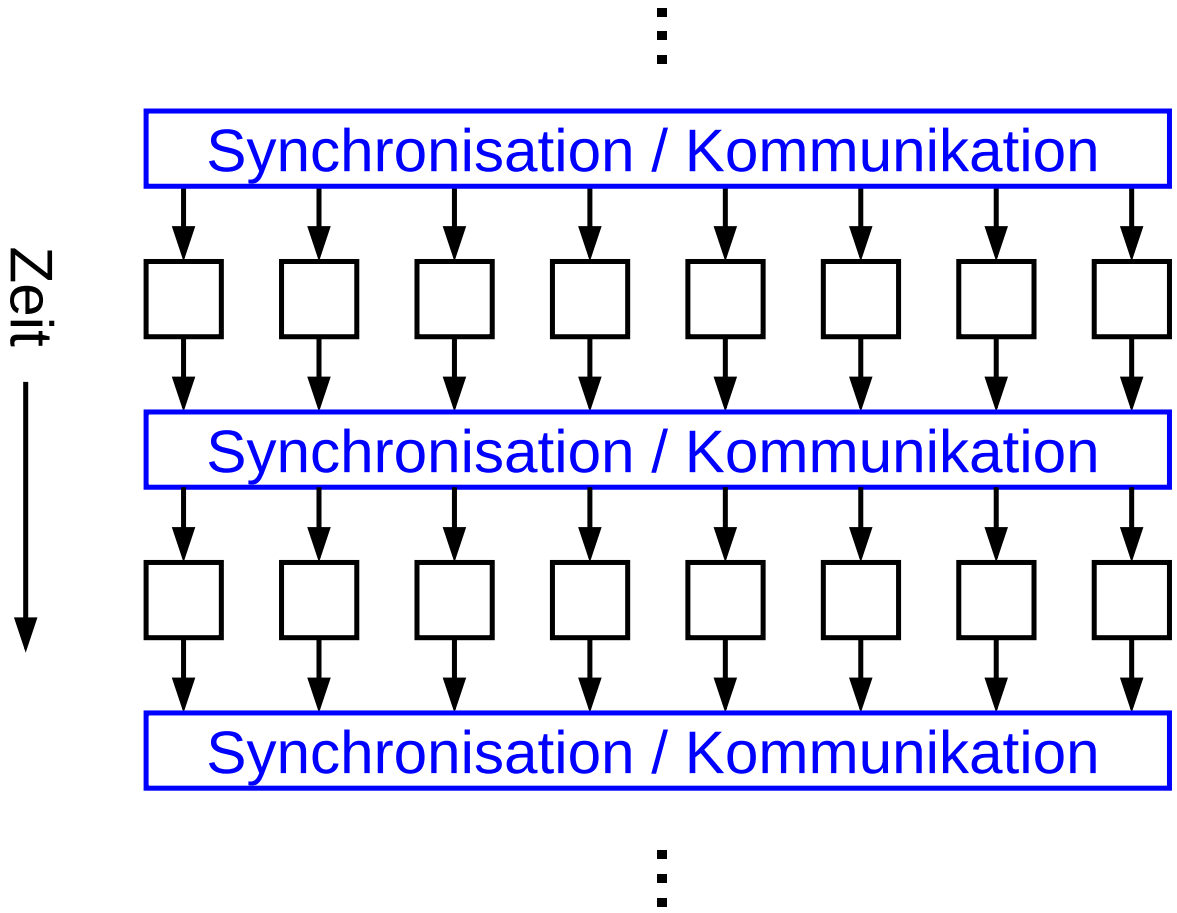


* zusätzliche Annahme: Neuer Thread wird nur erzeugt, wenn Arraylänge > 2

1.8.5 Datenparalleles Modell: SPMD

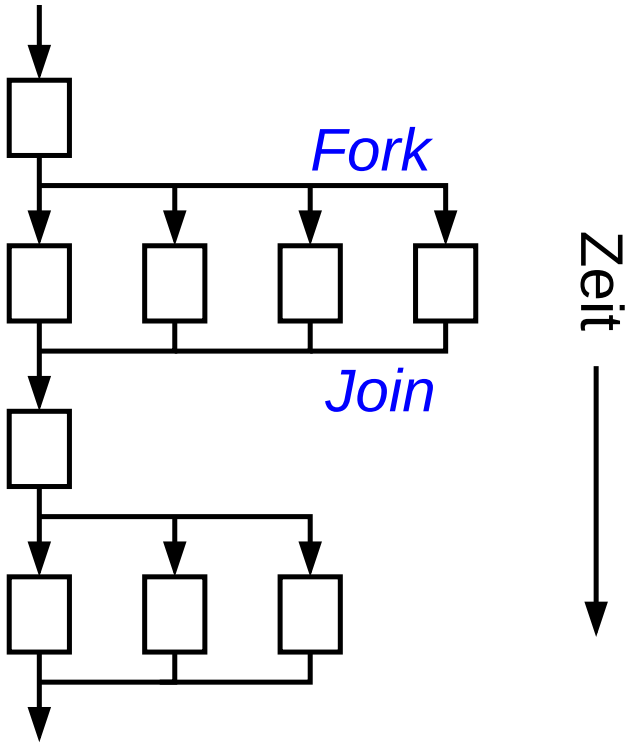
- Feste, konstante Anzahl von Prozessen (bzw. Threads)
- 1-zu-1 Zuordnung von Tasks zu Prozessen
- Alle Prozesse arbeiten denselben Programmcode ab
  - aber: Fallunterscheidungen möglich ...
- Bei nicht parallelisierbaren Programmteilen:
  - replizierte Ausführung in allen Prozessen
  - Ausführung in nur einem Prozeß, andere warten
- Meist lose synchroner Ablauf:
  - abwechselnde Phasen unabhängiger Berechung und Kommunikation / Synchronisation

Typischer Zeitablauf



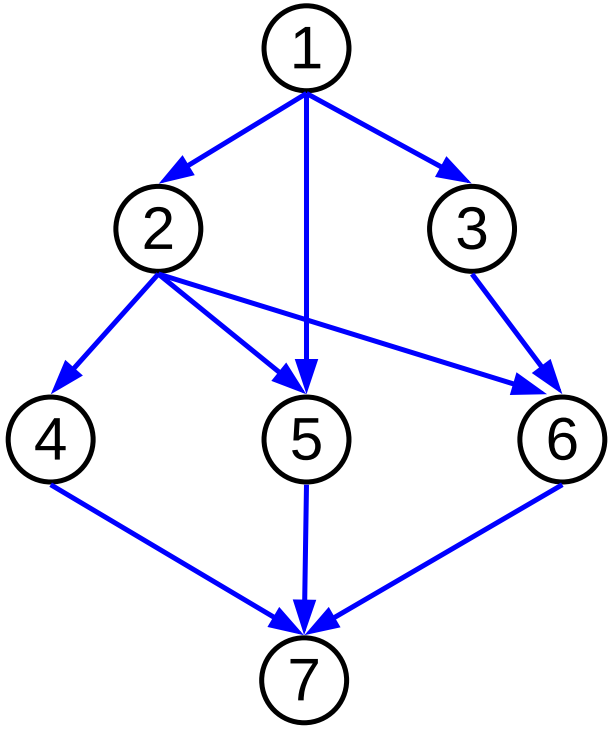
1.8.6 *Fork-Join*-Modell

- Programm besteht aus sequentiellen und parallelen Phasen
- Prozesse (bzw. Threads) für parallele Phase werden zur Laufzeit erzeugt (*Fork*)
  - einer pro Task
- Am Ende der parallelen Phase: Synchronisation und Beendigung der Threads (*Join*)



1.8.7 Task-Graph-Modell

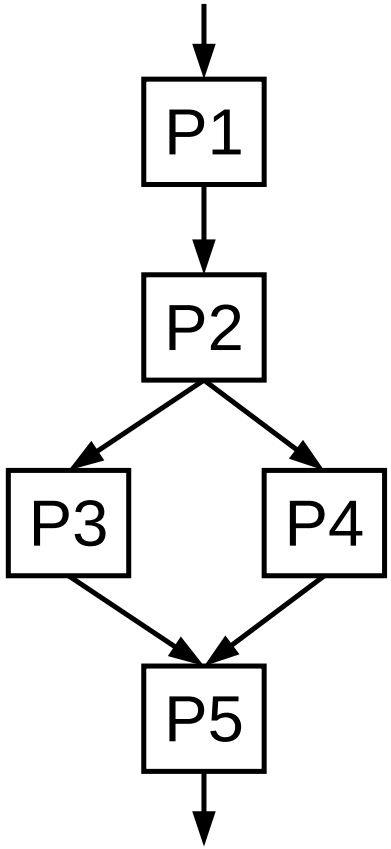
- Darstellung der Tasks und ihrer Abhängigkeiten (Datenfluß) als Graph
- Kante im Graph bedeutet Datenaustausch
  - z.B.: Task 1 produziert Daten, Task 2 startet Ausführung, wenn diese Daten vollständig vorliegen



- Zuordnung von Tasks zu Prozessen i.d.R. so, daß möglichst wenig Kommunikation notwendig wird
  - z.B. Tasks 1, 5, 7 in einem Prozeß

1.8.8 Pipeline-Modell

- Ein **Strom** von Daten wird durch eine Folge von Prozessen geleitet
- Bearbeitung eines Tasks startet, sobald ein Datenelement ankommt
- Pipeline muß nicht notwendigerweise linear sein
  - allgemeine (azyklische) Graphen wie bei Task-Graph-Modell möglich
- Zwischen den Prozessen: Erzeuger-Verbraucher-Synchronisation

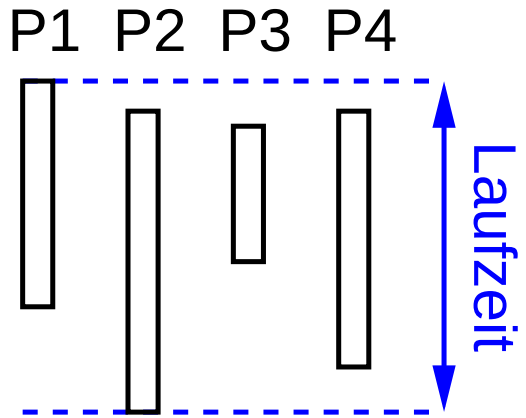


1.9 Leistungsbetrachtungen

- Welchen Leistungsgewinn ergibt die Parallelisierung?

- Mögliche Leistungsmetriken:
  - Laufzeit, Durchsatz, Speicherbedarf, Prozessorauslastung, Entwicklungskosten, Wartungskosten, ...

- Im folgenden betrachtet: Laufzeit
  - **Laufzeit** eines parallelen Programms: Zeit zwischen dem Start des Programms und dem Ende der Berechnung auf dem letzten Prozessor



Speedup (Beschleunigung)

- Laufzeit-Gewinn durch Parallelausführung
- **Absoluter *Speedup***

$$S(p) = \frac{T_s}{T(p)}$$

  - T_s = Laufzeit des sequentiellen Programms (bzw. des besten sequentiellen Algorithmus)
  - $T(p)$ = Laufzeit des parallelen Programms (Algorithmus) auf p Prozessoren

Speedup (Beschleunigung) ...

- **Relativer *Speedup*** (für „geschönte“ Ergebnisse ...)

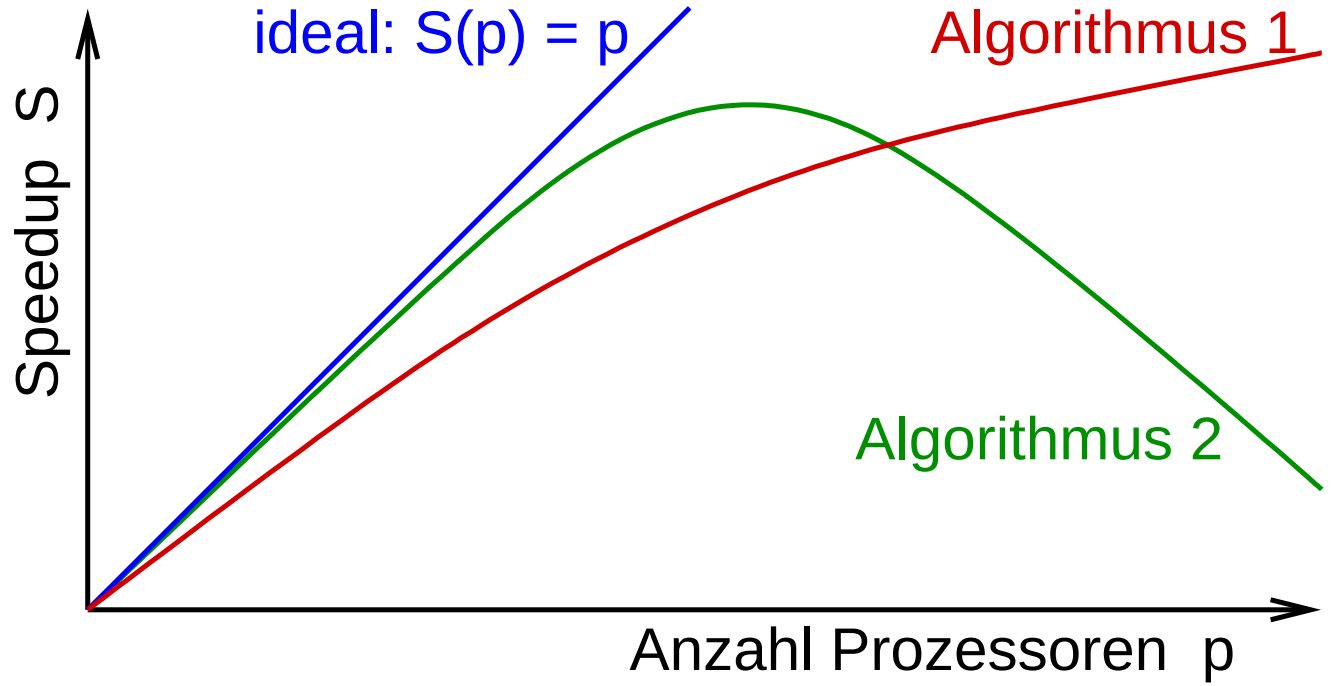
$$S(p) = \frac{T(1)}{T(p)}$$

  - $T(1) =$ Laufzeit des parallelen Programms (Algorithmus) auf einem Prozessor

- Optimal: $S(p) = p$
- Oft: bei **fester** Problemgröße sinkt $S(p)$ mit wachsendem p wieder
  - mehr Kommunikation, weniger Rechenarbeit pro Prozessor

Speedup (Beschleunigung) ...

- Typische Verläufe:



- Aussagen wie „Speedup von 7.5 auf 8 Prozessoren“ können nicht zu höheren Prozessorzahlen hin extrapoliert werden

Superlinearer *Speedup*

- Manchmal wird $S(p) > p$ beobachtet, obwohl dies eigentlich unmöglich sein sollte
- Ursachen:
  - implizite Änderung des Algorithmus
    - z.B. bei paralleler Baumsuche: es werden mehrere Pfade im Suchbaum gleichzeitig durchsucht
      - begrenzte Breitensuche statt Tiefensuche
  - Caching-Effekte
    - bei p Prozessoren: p mal so viel Cache-Speicher wie bei einem Prozessor
    - daher insgesamt auch höhere Trefferraten

Amdahls Gesetz

- Liefert obere Schranke für den *Speedup*
- Grundlage: in der Regel können nicht alle Teile eines Programms parallelisiert werden
  - wegen des Programmieraufwands
  - aufgrund von Datenabhängigkeiten
- Sei a der **Zeitanteil** dieser Programmteile in der **sequentiellen** Version des Programms. Dann:

$$S(p) = \frac{T_s}{T(p)} \leq \frac{1}{a + (1-a)/p} \leq \frac{1}{a}$$

- Bei 10% sequentiellem Anteil also $S(p) \leq 10$

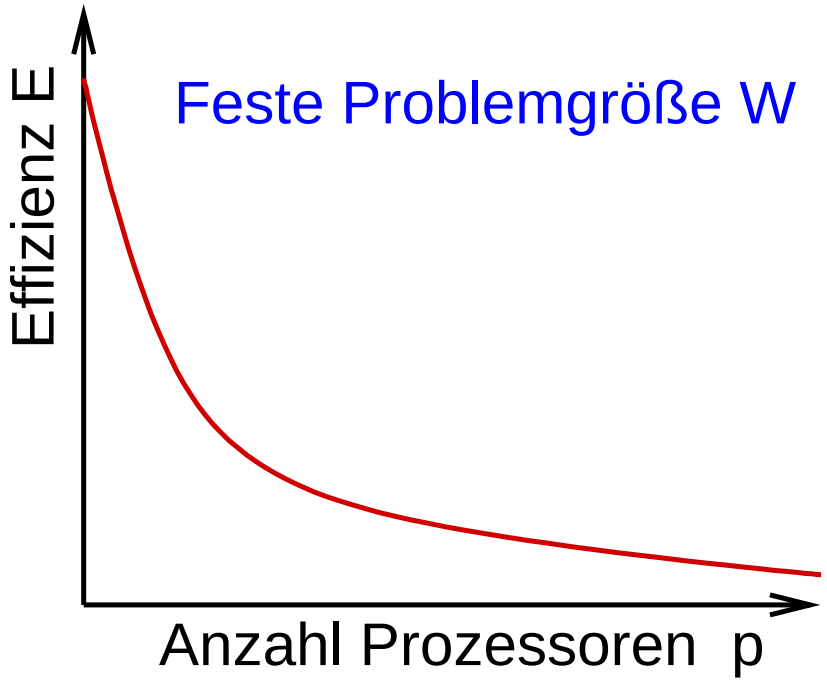
Effizienz

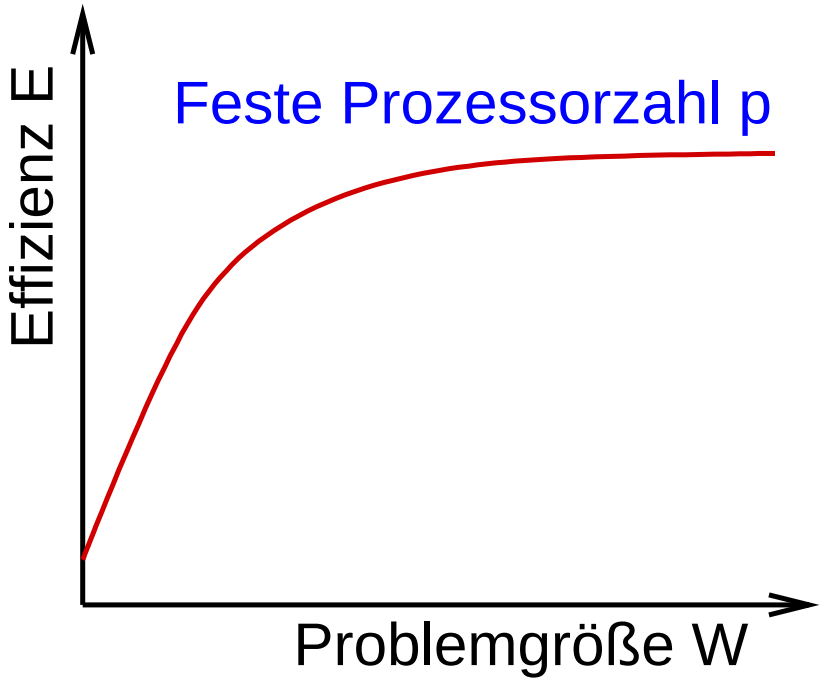
$$E(p) = \frac{S(p)}{p}$$

- Maß für die Ausnutzung des Parallelrechners
- $E(p) \leq 1$, optimal wäre $E(p) = 1$

Skalierbarkeit

- Typische Beobachtungen:





- Grund: bei steigendem p: weniger Arbeit pro Prozessor, aber gleich viel (oder mehr) Kommunikation

Skalierbarkeit ...

- Wie muß die Problemgröße W mit steigender Prozessorzahl p wachsen, damit die Effizienz konstant bleibt?
- Antwort gibt die **Isoeffizienz-Funktion**
- Parallele Laufzeit

$$T(p) = \frac{W + T_o(W,p)}{p}$$

  - $T_o(W,p) =$ Overhead der Parallelisierung
  - T u. W hier in Anzahl von Elementaroperationen gemessen

- Damit:

$$W = \frac{E}{1-E} \cdot T_o(W,p)$$

Skalierbarkeit ...

- Isoeffizienz-Funktion $I(p)$
  - Lösung der Gleichung $W = K \cdot T_o(W, p)$ nach W
  - K = Konstante, abhängig von gewünschter Effizienz
- Gute Skalierbarkeit: $I(p) = \mathcal{O}(p)$ oder $I(p) = \mathcal{O}(p \log p)$
- Schlechte Skalierbarkeit: $I(p) = \mathcal{O}(p^k)$
- Berechnung von $T_o(W, p)$ durch Analyse des parallelen Algorithmus
  - wieviel Zeit wird für Kommunikation / Synchronisation und evtl. zusätzliche Berechnungen benötigt?
  - Vertiefung und Beispiele in Kap. 1.9.5

Parallelverarbeitung

WS 2015/16

02.11.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

1.9.2 Gründe für Leistungsverluste

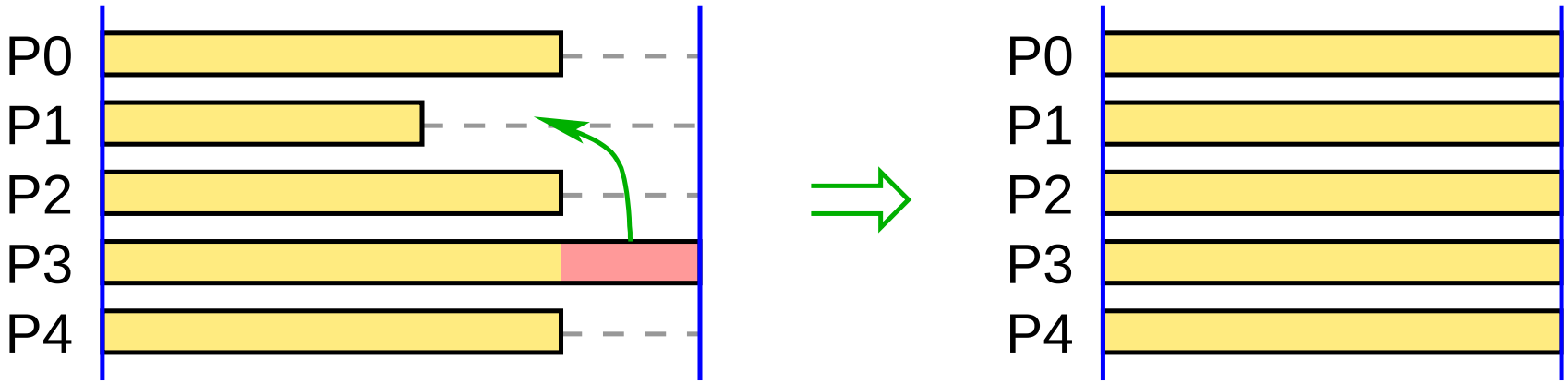
- **Zugriffsverluste** durch Datenaustausch zwischen Tasks
  - z.B. Nachrichtenaustausch, entfernter Speicherzugriff
- **Konfliktverluste** durch gemeinsame Benutzung von Ressourcen durch mehrere Tasks
  - z.B. Konflikte beim Netzzugriff, wechselseitiger Ausschluß beim Zugriff auf Daten
- **Auslastungsverluste** durch zu geringen Parallelitätsgrad
  - z.B. Warten auf Daten, Lastungleichheit
- **Bremsverluste** beim Beenden von Berechnungen
  - z.B. bei Suchproblemen: Benachrichtigung aller anderen Prozesse wenn eine Lösung gefunden wurde

1.9.2 Gründe für Leistungsverluste ...

- **Komplexitätsverluste** durch Zusatzaufwand bei der parallelen Abarbeitung
  - z.B. Partitionierung von unstrukturierten Gittern
- **Wegwerfverluste** durch Berechnungen, die redundant ausgeführt und nicht weiterverwendet werden
  - z.B. hinfällige Suche bei *Branch-and-Bound*
- **Algorithmische Verluste** durch Änderung des Algorithmus bei der Parallelisierung
  - z.B. schlechtere Konvergenz bei Iterationsverfahren

Einführung

- Für optimale Leistung: Prozessoren sollten zwischen zwei (globalen) Synchronisationen gleich lange rechnen
  - Synchronisation: auch Nachrichten, Programmstart/-ende



- Last hier: Rechenzeit zwischen zwei Synchronisationen
  - auch andere Lastmaße möglich, z.B. Kommunikationslast
- Lastausgleich ist eines der Ziele beim *Mapping*

Gründe für Lastungleichgewicht

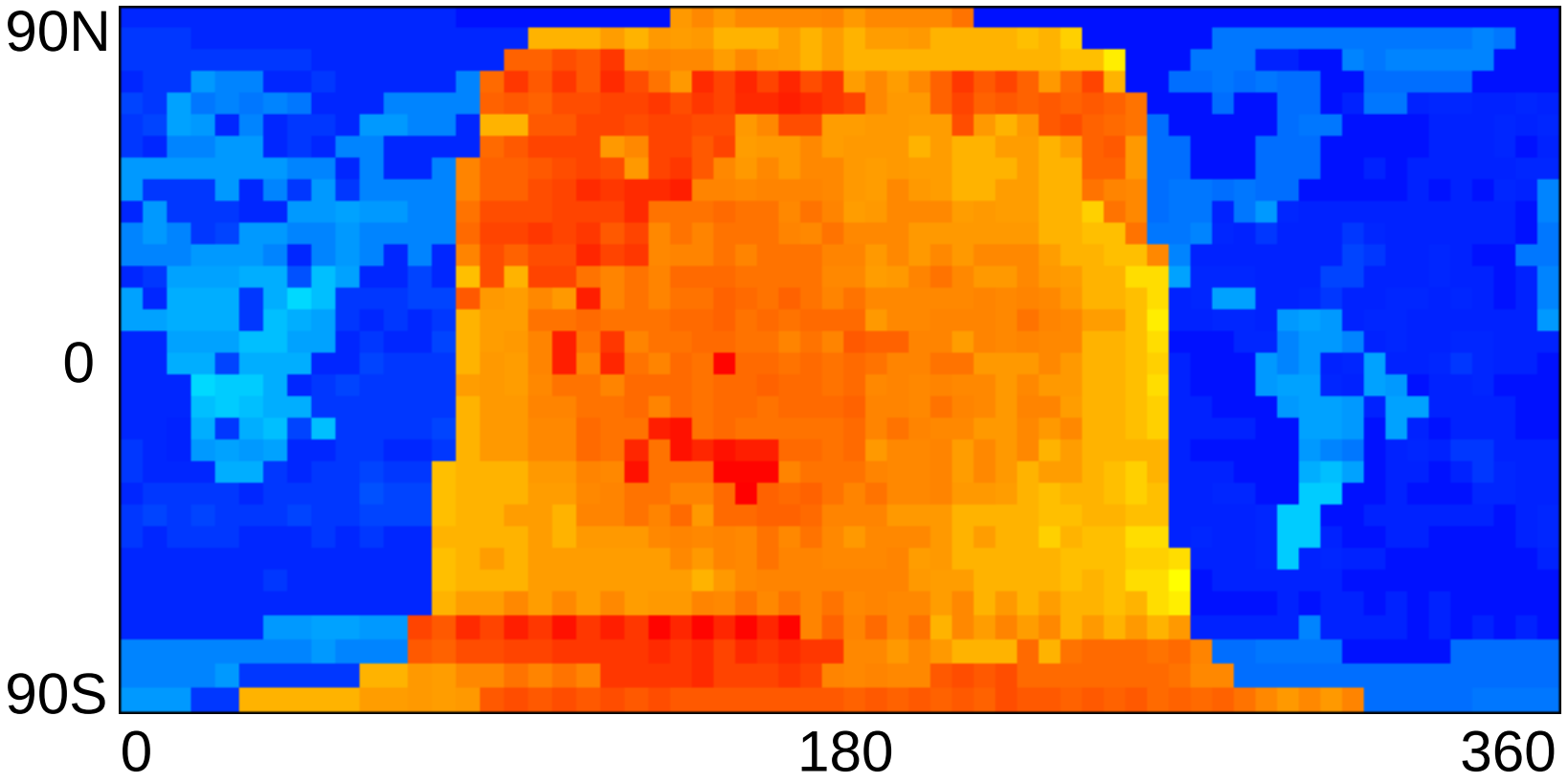
- Ungleiche Rechenlast der Tasks
  - z.B. Atmosphärenmodell: Bereiche über Land / Wasser
- Heterogene Ausführungsplattform
  - z.B. verschieden schnelle Prozessoren

statisch

- Rechenlast der Tasks ändert sich dynamisch
  - z.B. bei Atmosphärenmodell je nach simulierter Tageszeit (Sonneneinstrahlung)
- Hintergrundlast auf den Prozessoren
  - z.B. bei PC-Cluster

dynamisch

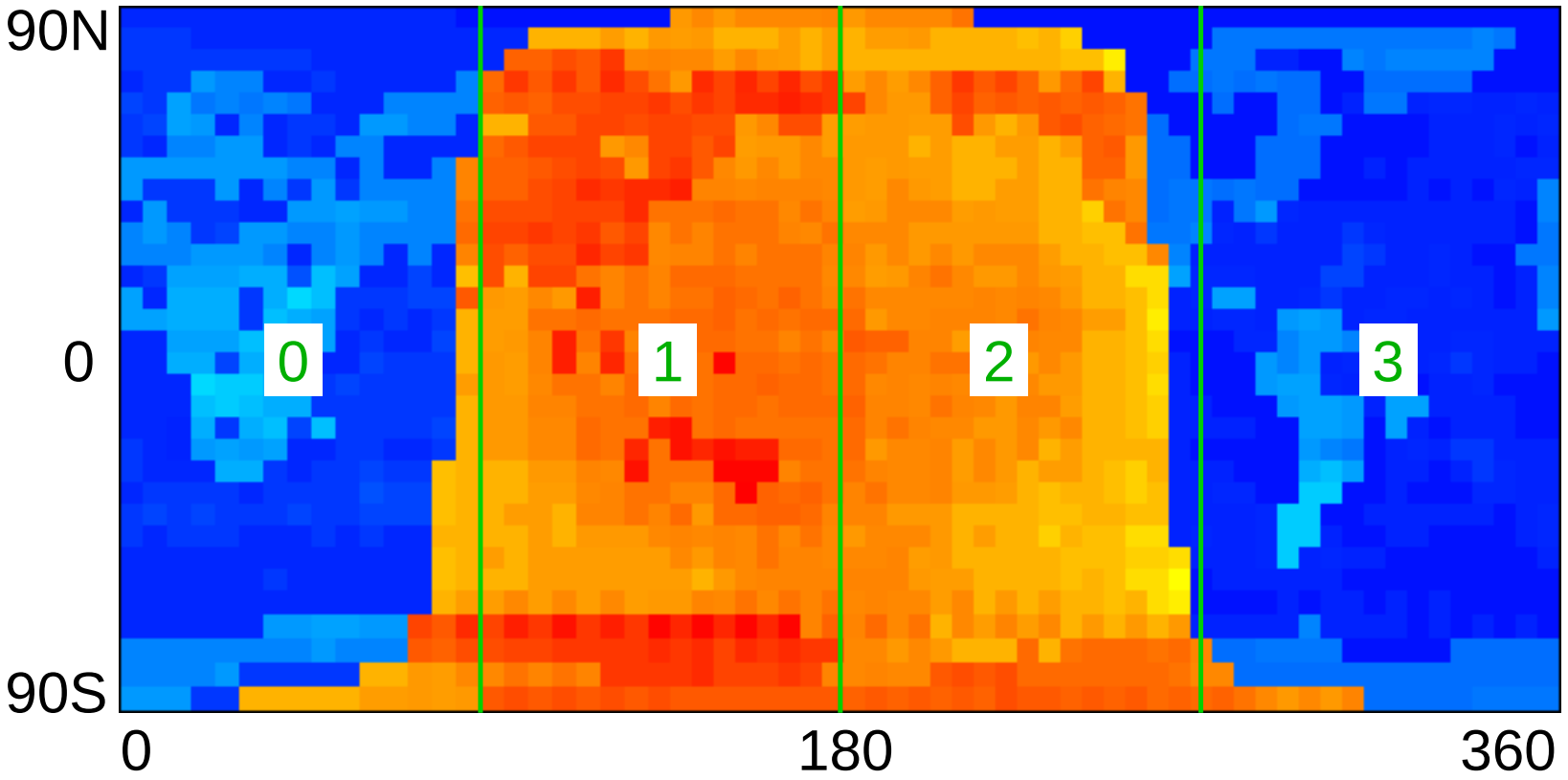
Beispiel: Atmosphärenmodell



- Nacht: Wenig Rechen−zeit pro Task
- Tag: Viel Rechen−zeit pro Task

- Kontinente: statisches Lastungleichgewicht
- Tag/Nacht-Grenze: dynamisches Lastungleichgewicht

Beispiel: Atmosphärenmodell



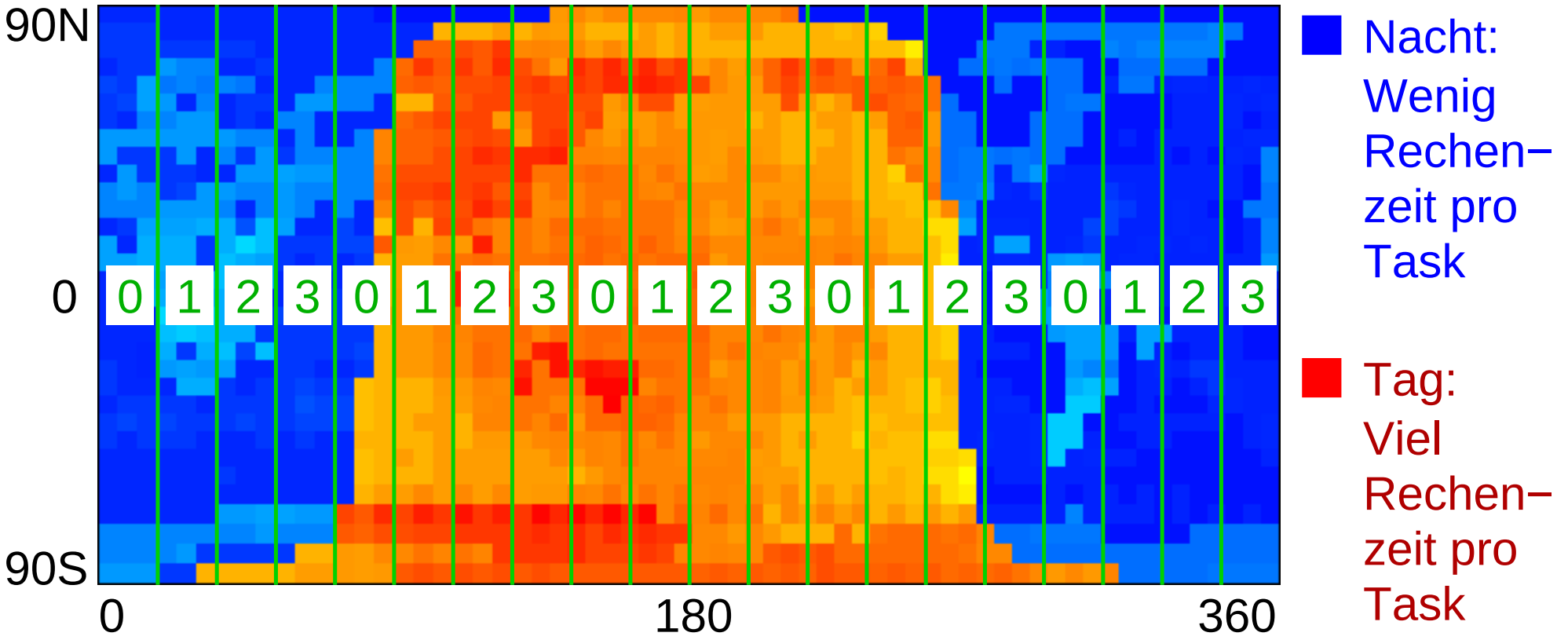
- Nacht: Wenig Rechen−zeit pro Task
- Tag: Viel Rechen−zeit pro Task

- Kontinente: statisches Lastungleichgewicht
- Tag/Nacht-Grenze: dynamisches Lastungleichgewicht

Statischer Lastausgleich

- Ziel: Tasks bei / vor Programmstart so auf Prozessoren verteilen, daß Rechenlast der Prozessoren identisch ist
- Zwei grundsätzlich verschiedene Vorgehensweisen:
  - Berücksichtige die unterschiedliche Rechenlast der Tasks bei der Aufteilung auf Prozessoren
    - Erweiterung von Graph-Partitionierungsalgorithmen
    - erfordert gute Abschätzung der Last einer Task
    - keine Lösung, wenn sich Last dynamisch ändert
  - feingranulares zyklisches oder zufälliges *Mapping*
    - erreicht (wahrscheinlich) gute Balanzierung der Last, auch bei dynamisch veränderlicher Last
    - Preis: i.a. höhere Kommunikationskosten

Beispiel: Atmosphärenmodell, zyklisches Mapping



- Jeder Prozessor bekommt Tasks mit hoher und niedriger Rechenlast

Dynamischer Lastausgleich

- Unabhängige (oft dyn. erzeugte) Tasks (z.B. Suchproblem)
  - Ziel: Prozessoren werden nicht untätig (*idle*), d.h. haben immer eine Task zum Bearbeiten
    - auch am Ende des Programms, d.h. alle Prozessoren werden gleichzeitig fertig
  - Tasks werden dynamisch an Prozessoren **zugeteilt**, bleiben dort bis zum Ende ihrer Bearbeitung
    - optimal: Tasks mit längster Bearbeitungszeit zuerst zuteilen
- Kommunizierende Tasks (SPMD, z.B. *Stencil*-Algorithmus)
  - Ziel: gleiche Rechenzeit zwischen Synchronisationen
  - Tasks werden ggf. während der Abarbeitung zwischen Prozessoren **verschoben**

Bestimmung der Leistungsmaße

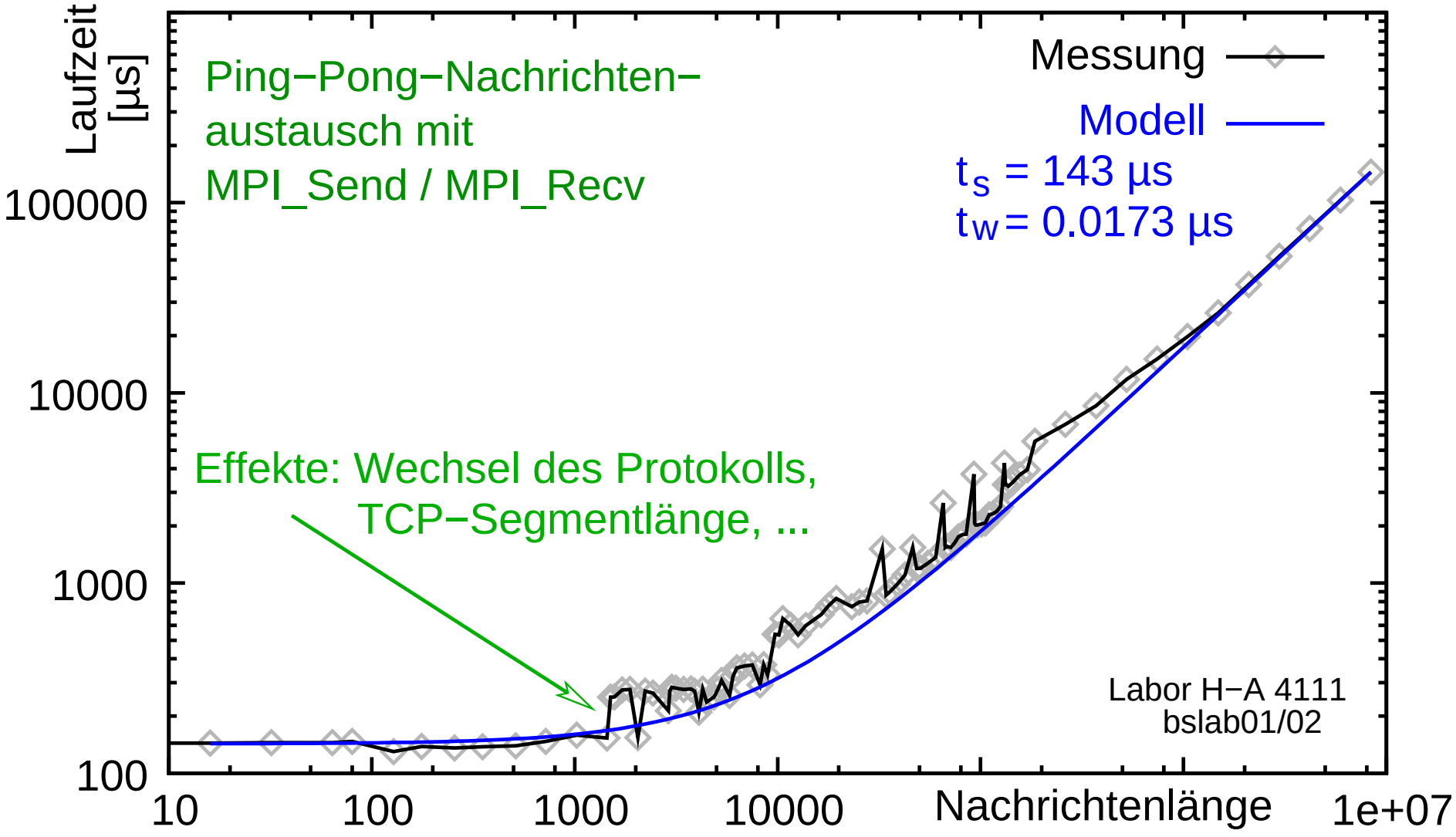
- Analytisches Modell des Algorithmus
  - Ansatz: Bestimmung von Rechen- und Kommunikationszeit
    - $T(p) = t_{comp} + t_{comm}$
    - Berechnungs/Kommunikations-Verhältnis t_{comp}/t_{comm} liefert grobe Leistungsabschätzung
  - benötigt Berechnungsmodell (Modell der Rechnerhardware)
- Messung am realen Programm
  - explizite Zeitmessung im Code
  - Werkzeuge zur Leistungsanalyse

Modelle für die Kommunikationszeit

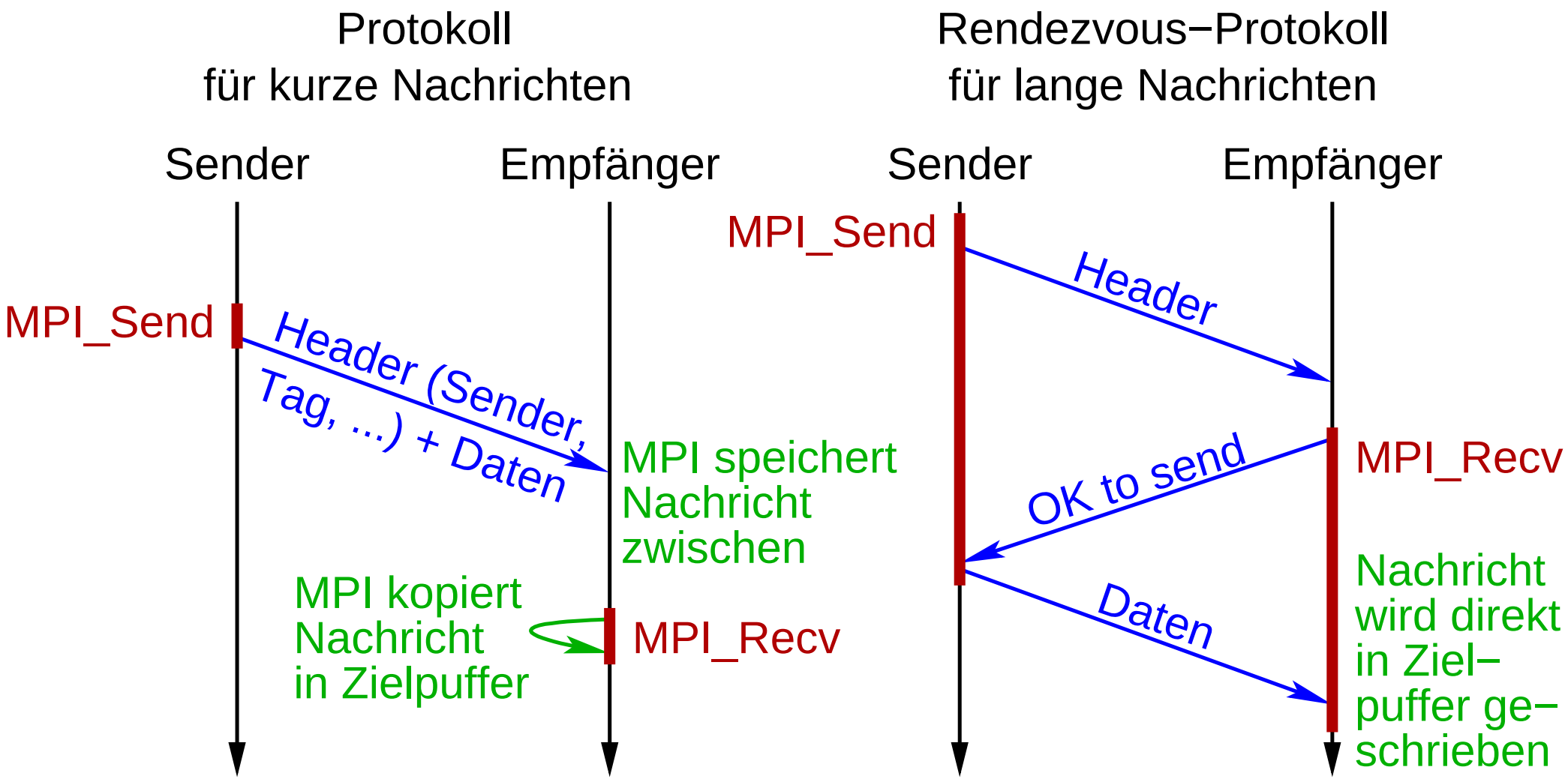
- Z.B. für MPI (nach Rauber: Parallele und verteilte Programmierung)
  - Punkt-zu-Punkt Senden: $t(m) = t_s + t_w \cdot m$
  - Broadcast: $t(p, m) = \tau \cdot \log p + t_w \cdot m \cdot \log p$
- Bestimmung der Parameter (t_s, t_w, τ) über **Mikrobenchmarks**
  - messen gezielt einzelne Aspekte des Systems aus
  - erlauben auch Ableitung von Implementierungs-Eigenschaften
  - Anpassung z.B. über Methode der kleinsten Quadrate
    - z.B. für Punkt-zu-Punkt Senden:

      PC-Cluster H-A 4111: $t_s = 71.5\ \mu s,\ t_w = 8{,}6\ ns$
      SMP-Cluster (remote): $t_s = 25.6\ \mu s,\ t_w = 8{,}5\ ns$
      SMP-Cluster (lokal): $t_s = 0{,}35\ \mu s,\ t_w = 0{,}5\ ns$

Beispiel: Ergebnisse des Mikrobenchmarks SKaMPI



Kommunikationsprotokolle in MPI

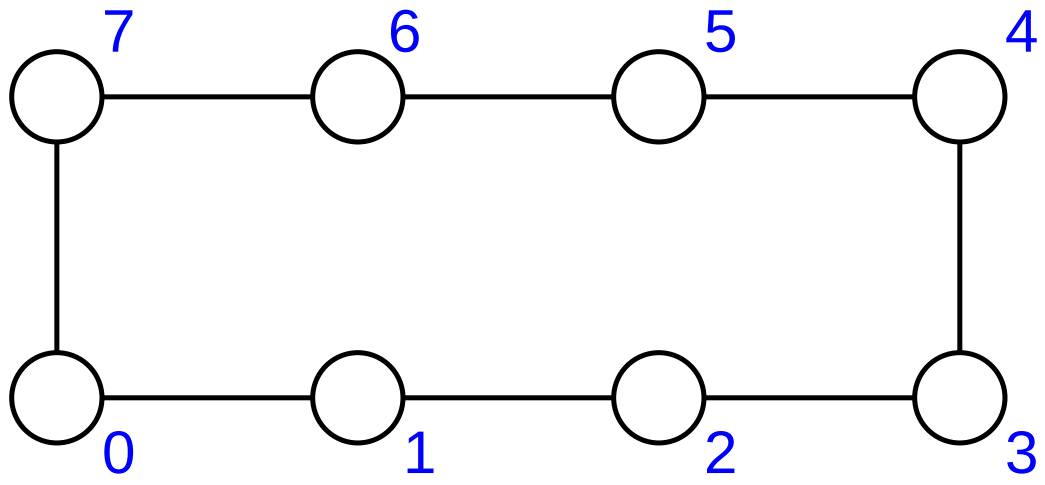


Beispiel: Matrix-Multiplikation

- Produkt $C = A \cdot B$ zweier quadratischer Matrizen
- Annahme: A, B, C blockweise auf p Prozessoren verteilt
  - Prozessor P_{ij} hat A_{ij} und B_{ij} und berechnet C_{ij}
- P_{ij} benötigt A_{ik} und B_{kj} für $k = 1 \ldots \sqrt{p}$
- Vorgehensweise:
  - *All-to-All-Broadcast* der A-Blöcke in jeder Zeile von Prozessoren
  - *All-to-All-Broadcast* der B-Blöcke in jeder Spalte von Prozessoren
  - Berechnung von $C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} \cdot B_{kj}$

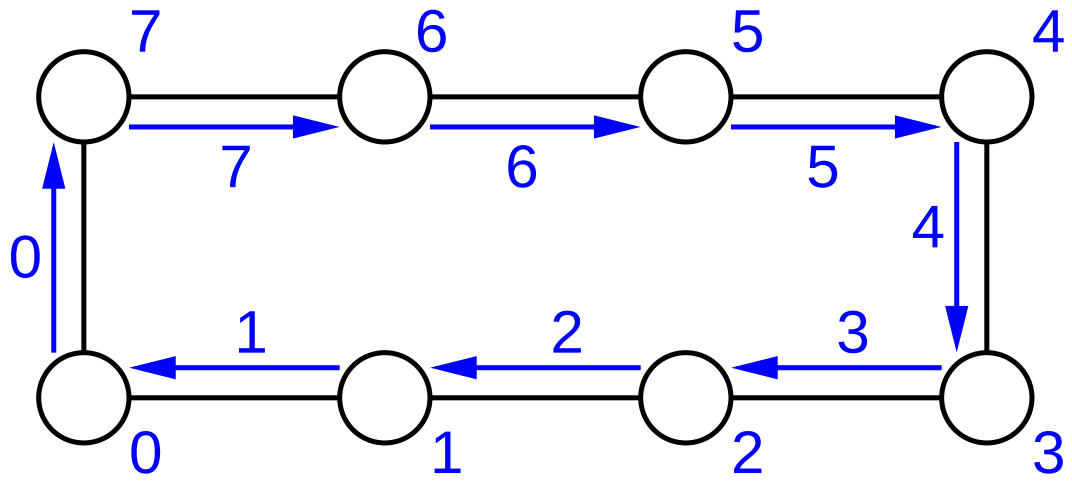
All-to-All-Broadcast

- Zeitaufwand abhängig von gewählter Kommunikationsstruktur
- Diese ggf. abhängig von Netzwerkstruktur des Parallelrechners
  - wer kann mit wem direkt kommunizieren?
- Beispiel: Ringtopologie



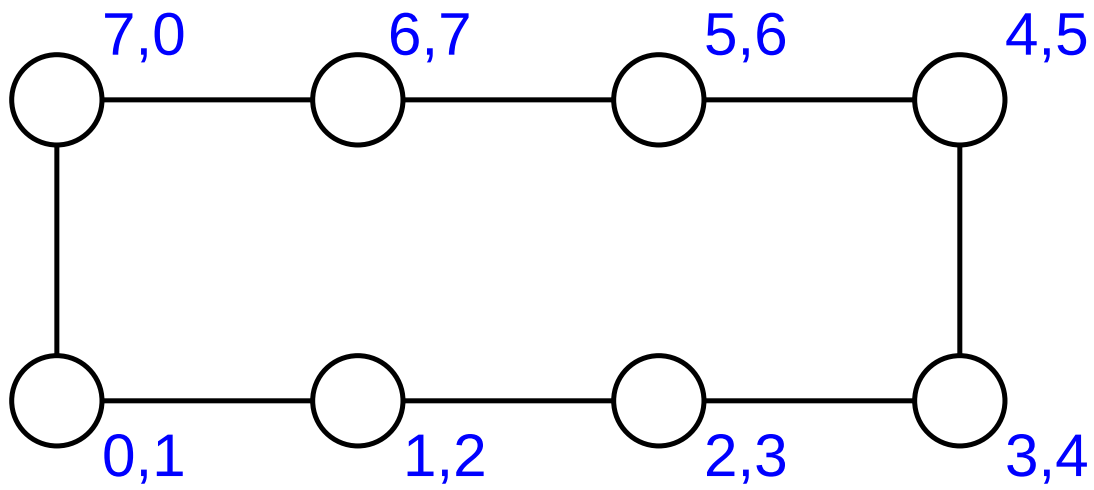
All-to-All-Broadcast

- Zeitaufwand abhängig von gewählter Kommunikationsstruktur
- Diese ggf. abhängig von Netzwerkstruktur des Parallelrechners
  - wer kann mit wem direkt kommunizieren?
- Beispiel: Ringtopologie



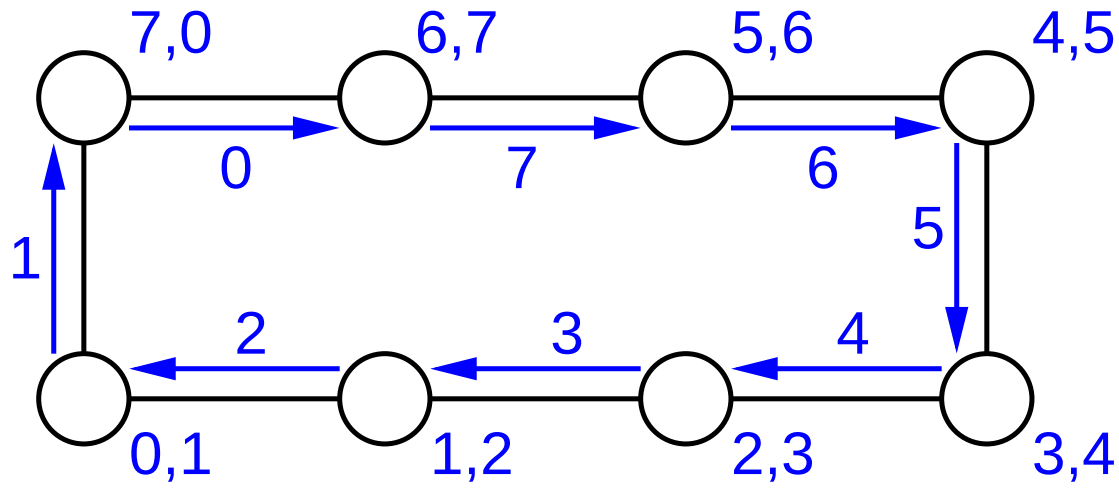
All-to-All-Broadcast

- Zeitaufwand abhängig von gewählter Kommunikationsstruktur
- Diese ggf. abhängig von Netzwerkstruktur des Parallelrechners
  - wer kann mit wem direkt kommunizieren?
- Beispiel: Ringtopologie



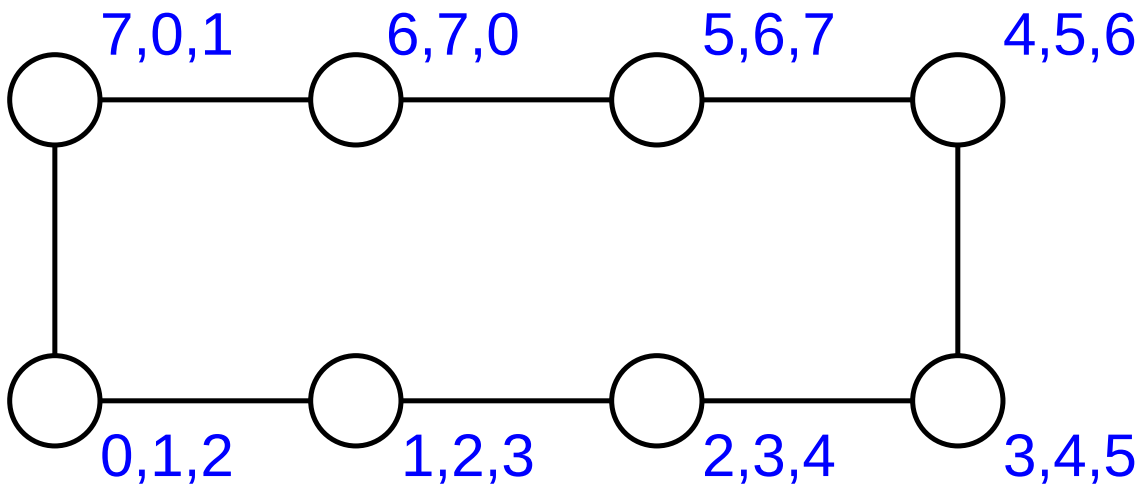
All-to-All-Broadcast

- Zeitaufwand abhängig von gewählter Kommunikationsstruktur
- Diese ggf. abhängig von Netzwerkstruktur des Parallelrechners
  - wer kann mit wem direkt kommunizieren?
- Beispiel: Ringtopologie



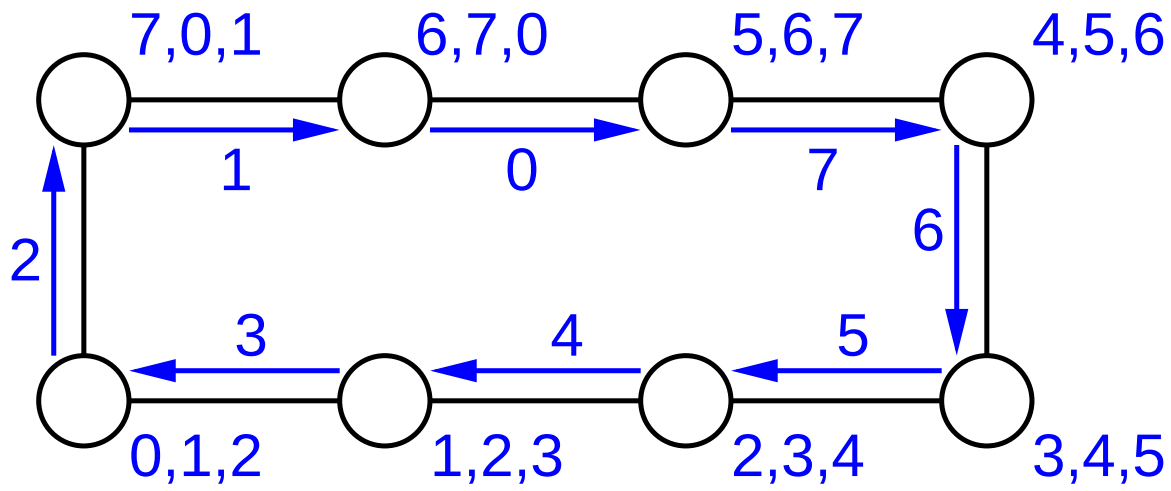
All-to-All-Broadcast

- Zeitaufwand abhängig von gewählter Kommunikationsstruktur
- Diese ggf. abhängig von Netzwerkstruktur des Parallelrechners
  - wer kann mit wem direkt kommunizieren?
- Beispiel: Ringtopologie



All-to-All-Broadcast

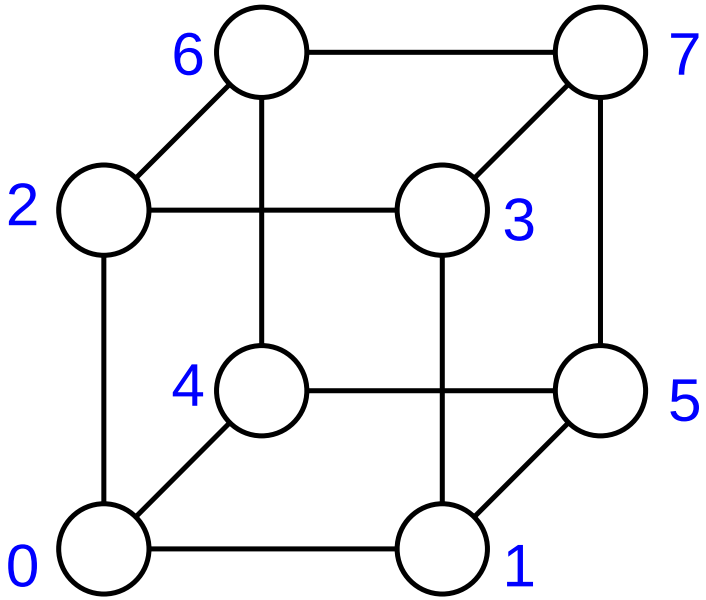
- Zeitaufwand abhängig von gewählter Kommunikationsstruktur
- Diese ggf. abhängig von Netzwerkstruktur des Parallelrechners
  - wer kann mit wem direkt kommunizieren?
- Beispiel: Ringtopologie



- Kosten: $t_s(p-1) + t_w m(p-1)$ (m: Datenlänge)

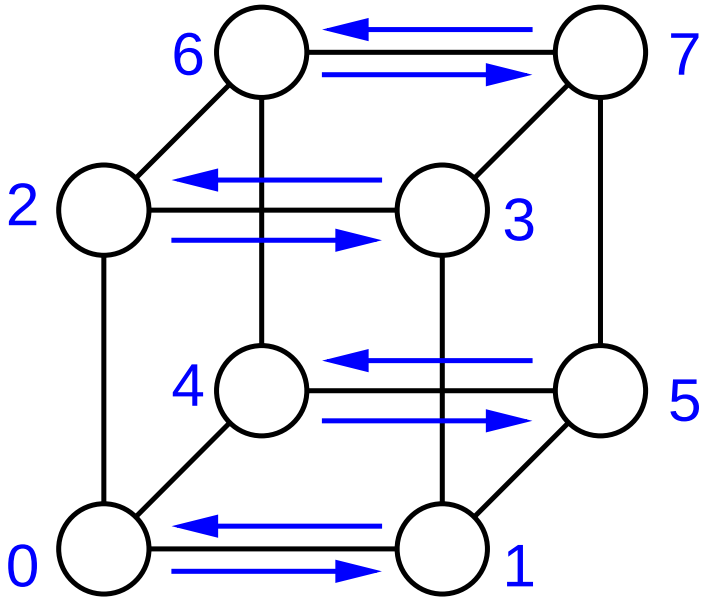
All-to-All-Broadcast ...

- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



All-to-All-Broadcast ...

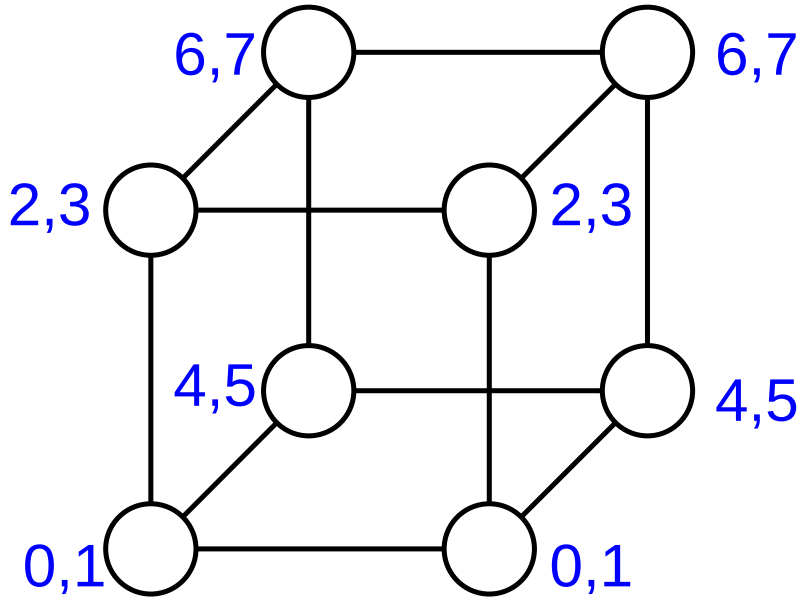
- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



1. Paarweiser Austausch in x−Richtung

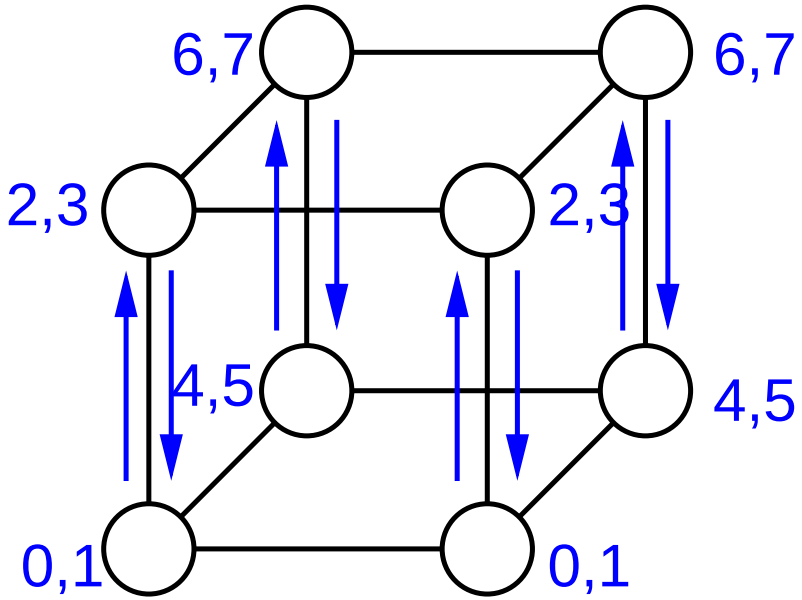
All-to-All-Broadcast ...

- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



All-to-All-Broadcast ...

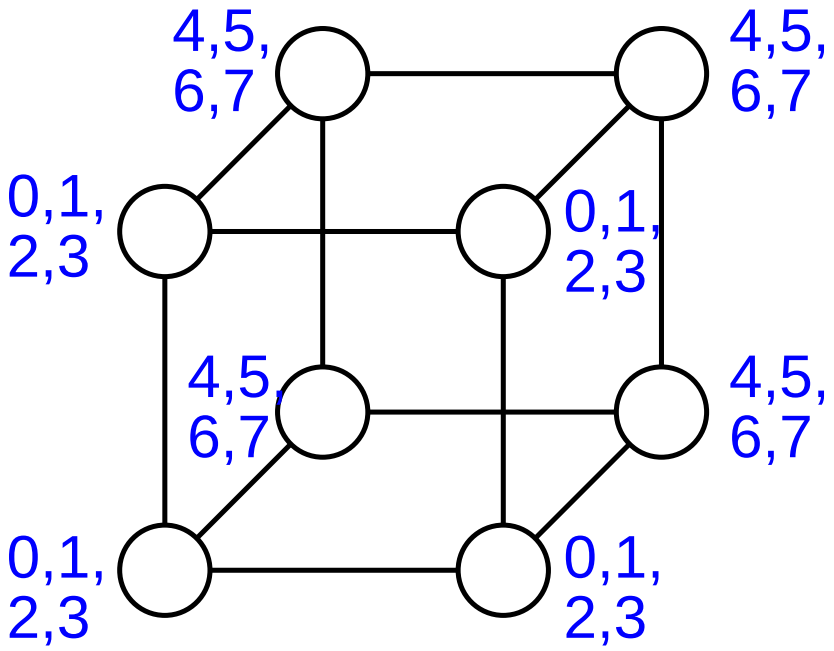
- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



2. Paarweiser Austausch in y−Richtung

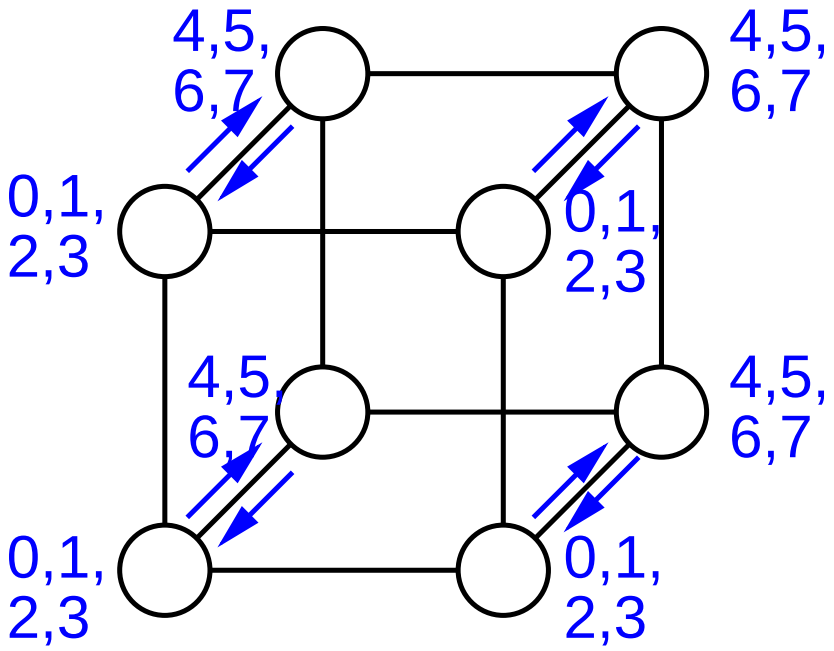
All-to-All-Broadcast ...

- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



All-to-All-Broadcast ...

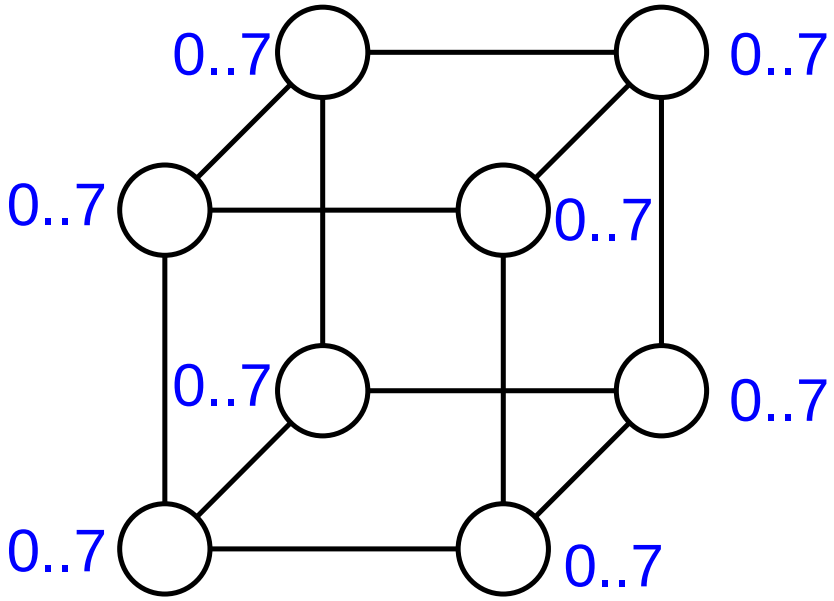
- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



3. Paarweiser Austausch
   in z−Richtung

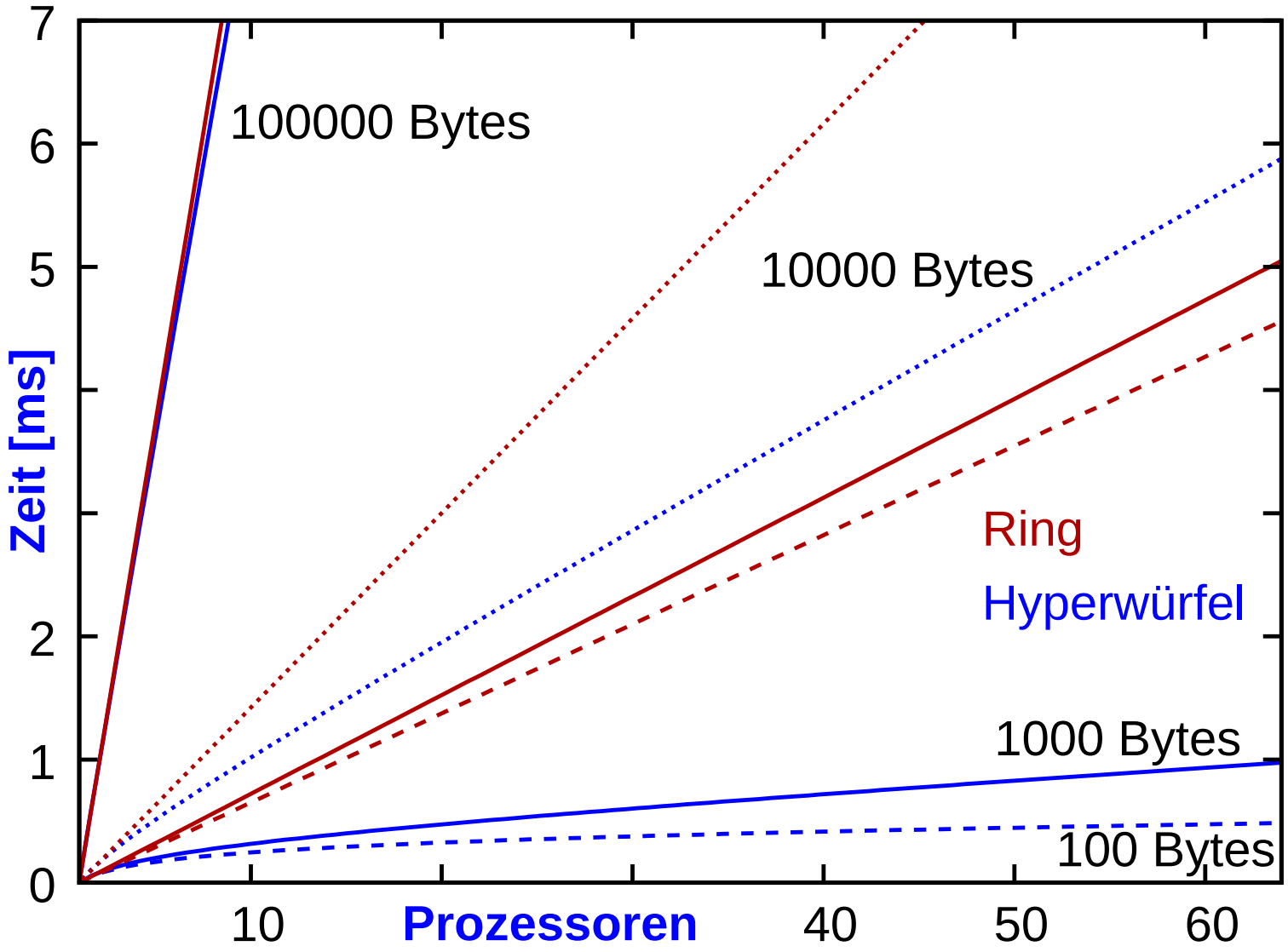
All-to-All-Broadcast ...

- Beispiel: Kommunikation entlang eines Hyperwürfels
  - benötigt nur $\log p$ Schritte bei p Prozessoren



- Kosten: $\sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m) = t_s \log p + t_w m(p-1)$

All-to-All-Broadcast ...



t_s = 71,5 µs

t_w = 8,65 ns

(Werte aus dem Labor H−A 4111)

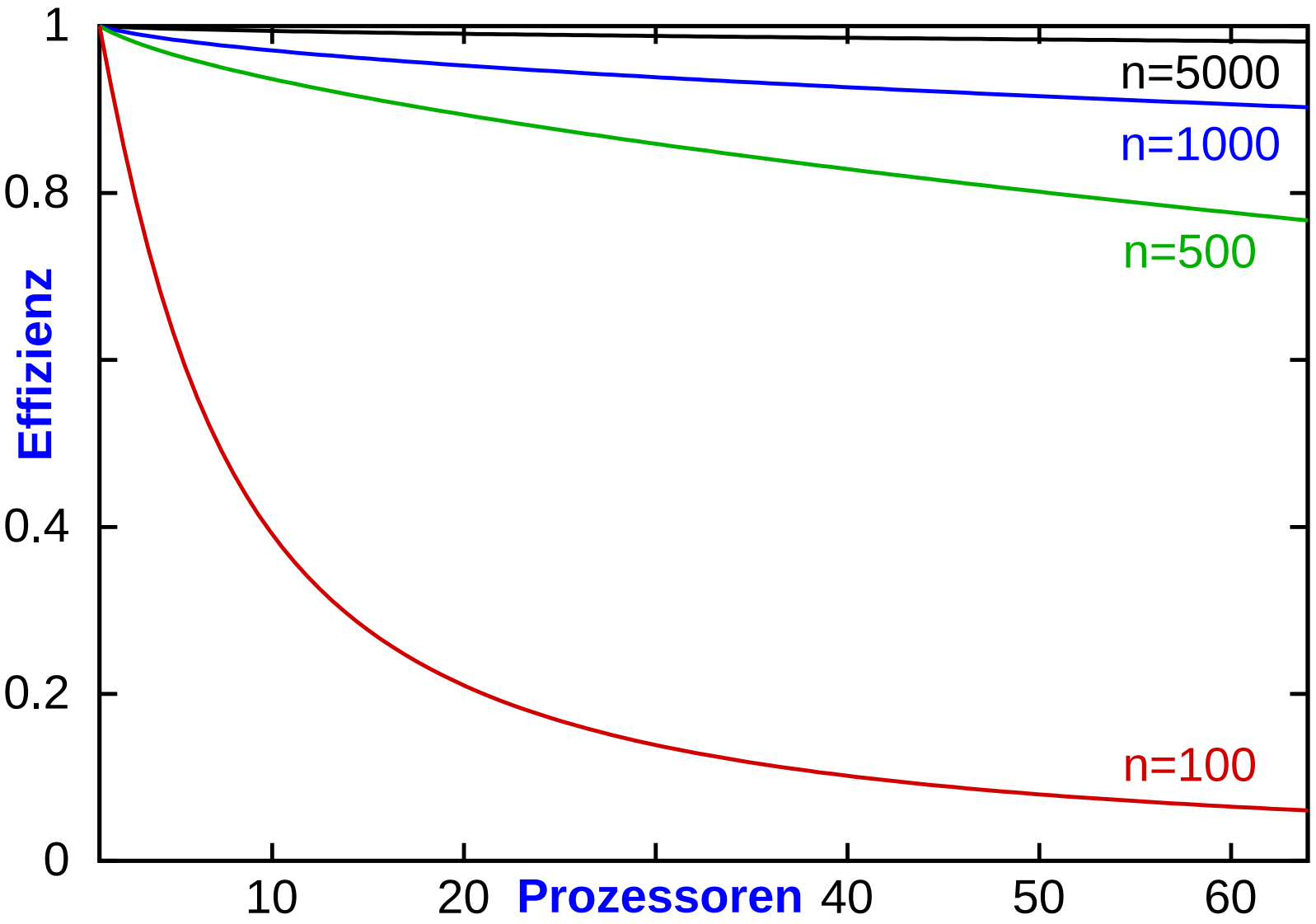
Gesamtanalyse der Matrix-Multiplikation

- Zwei *All-to-All-Broadcast*-Schritte zwischen $\sqrt{p}$ Prozessen
  - jeweils nebenläufig in $\sqrt{p}$ Zeilen / Spalten
- Kommunikationszeit: $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$

- $\sqrt{p}$ Multiplikationen von $(n/\sqrt{p}) \times (n/\sqrt{p})$ Untermatrizen
- Rechenzeit: $t_c\sqrt{p} \cdot (n/\sqrt{p})^3 = t_c n^3/p$

- Parallele Laufzeit: $T(p) \approx t_c n^3/p + t_s \log p + 2t_w(n^2/\sqrt{p})$
- Sequentielle Laufzeit: $T_s = t_c n^3$

Effizienz der Matrix-Multiplikation



Rechenzeit pro Matrixelement:

t_c = 1.3 ns

t_s = 71,5 µs

t_w = 8,65 ns

(Werte aus dem Labor H−A 4111)

Parallelverarbeitung

WS 2015/16

09.11.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

1.9.6 Leistungsanalyse-Werkzeuge

- Ziel: *Performance-Debugging*, Finden und Beseitigen von Leistungsengpässen
- Methode: Messung verschiedener Größen (Metriken), ggf. aufgeschlüsselt nach:
  - Ausführungseinheit (Rechnerknoten, Prozeß, Thread)
  - Quellcodeposition (Prozedur, Codezeile)
  - Zeit
- Werkzeuge im Detail sehr unterschiedlich
  - Meßmethode, notwendige Vorbereitung, Aufbereitung der Information, ...
- Einige Werkzeuge sind auch zur Visualisierung des Programmablaufs verwendbar

Metriken für die Leistungsanalyse

- CPU-Zeit (Bewertung des Rechenaufwands)
- *Wall Clock*-Zeit (inkl. Blockierungs-Zeiten)
- Kommunikationszeit und -volumen
- Metriken des Betriebssystems:
  - Seitenfehler, Prozeßwechsel, Systemaufrufe, Signale
- Hardware-Metriken (nur mit Unterstützung durch CPU-HW):
  - CPU-Zyklen, Gleitkommaoperationen, Speicherzugriffe
  - *Cache misses*, Cache-Invalidierungen, ...

Stichprobenbasierte Leistungsmessung (*Sampling*)

- Regelmäßige Unterbrechung des Programms
- Auslesen des aktuellen Befehlszählerstands (und evtl. des Aufrufkellers)
- Gesamter Meßwert wird dieser Stelle im Programm zugerechnet, z.B. bei Messung der CPU-Zeit:
  - periodische Unterbrechung alle $10ms$ CPU-Zeit
  - CPU-Zeit[aktueller BZ-Stand] += $10ms$
- Umsetzung auf Quellsprachebene erfolgt offline
- Ergebnis: Meßwert für jede Funktion / Zeile

Ereignisbasierte Leistungsmessung (*Profiling, Tracing*)

- Erfordert **Instrumentierung** des Programms, d.h. Einbau von Meßcode an interessanten Stellen
  - meist Beginn und Ende von Bibliotheksfunktionen, z.B. MPI_Recv, MPI_Barrier, ...
- Instrumentierung i.d.R. automatisch durch das Werkzeug
  - meist erneute Übersetzung oder erneutes Binden notwendig
- Auswertung der Ereignisse während der Erfassung (Profiling) oder erst nach Ende des Programmlaufs (Tracing)
- Ergebnis:
  - Meßwert für jede gemessene Funktion (Profiling, Tracing)
  - Verlauf des Meßwerts über der Zeit (Tracing)

Beispiel: Messung von *Cache Misses*

- Basis: Hardware-Zähler im Prozessor zählt Cache-Misses
- Stichprobenbasiert:
  - bei Erreichen eines bestimmten Zählerstandes (419) wird eine Unterbrechung ausgelöst
  - cache_misses[aktueller BZ] += 419
- Ereignisbasiert:
  - Einfügen von Code zum Auslesen des Zählers:

```
old_cm = read_hw_counter(25);
for (j=0;j<1000;j++)
    d += a[i][j];
cache_misses += read_hw_counter(25)-old_cm;
```

Vor- und Nachteile der Methoden

- *Sampling*
  - niedrige, vorhersagbare Rückwirkung; Quellcode-Bezug
  - begrenzte Genauigkeit, keine Zeitauflösung
- *Tracing*
  - Erfassung aller relevanter Daten mit hoher Zeitauflösung
  - relative große Rückwirkung, große Datenmengen
- *Profiling*
  - reduzierte Datenmenge, aber wenig flexibel

Parallelverarbeitung

WS 2015/16

2 Parallele Programmierung mit Speicherkopplung

Inhalt

- Grundlagen
- Posix Threads
- OpenMP Grundlagen
- Schleifenparallelisierung und Abhängigkeiten
- OpenMP Synchronisation
- OpenMP Vertiefung

Literatur

- Wilkinson/Allen, Kap. 8.4, 8.5, Anhang C
- Hoffmann/Lienhart

Ansätze zur Programmierung mit Threads

- Durch (System-)Bibliotheken
  - Beispiele: POSIX Threads (☞ **2.2**), Intel Threading Building Blocks (TBB), C++ Boost Threads
- Als Bestandteil einer Programmiersprache
  - Beispiel: Java Threads (☞ **BS_I**), C++11
- Durch Compilerdirektiven (Pragmas)
  - Beispiel: OpenMP (☞ **2.3**)

2.1.1 Synchronisation

- Sicherstellung von Bedingungen an die möglichen zeitlichen Abläufe von Threads
  - wechselseitiger Ausschluß
  - Reihenfolge von Aktionen in verschiedenen Threads
- Hilfsmittel:
  - gemeinsame Variable
  - Semaphore / Mutexe
  - Monitore / Bedingungsvariable
  - Barrieren

Synchronisation durch gemeinsame Variable

- Beispiel: Warten auf Ergebnis

Thread 1

```
// berechne und
// speichere Ergebnis
ready = true;
...
```

Thread 2

```
while (!ready); // warte
// lies / verarbeite Ergebnis
...
```

- Erweiterung: atomare *Read-Modify-Write*-Operationen der CPU
  - z.B. *test-and-set, fetch-and-add*
- Potentieller Nachteil: *Busy Waiting*
  - aber: bei Parallelverarbeitung oft genau ein Thread pro CPU
    ⇒ Geschwindigkeitsvorteil, da kein Systemaufruf

Semaphore

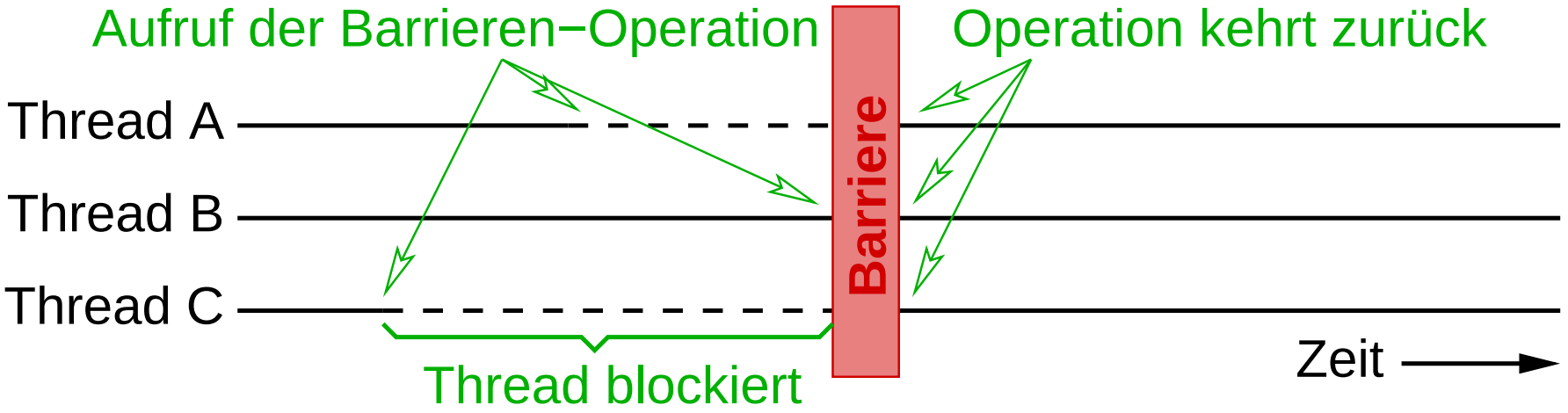
- Bestandteile: Zähler, Warteschlange blockierter Threads
- **atomare** Operationen:
  - `P()` (auch `acquire`, `wait` oder `down`)
    - verringert Zähler um 1
    - falls Zähler $<$ 0: Thread blockieren
  - `V()` (auch `release`, `signal` oder `up`)
    - erhöht Zähler um 1
    - falls Zähler $\leq$ 0: einen blockierten Thread wecken
- **Binäres Semaphor**: kann nur Werte 0 und 1 annehmen
  - i.d.R. für wechselseitigen Ausschluß

Monitore

- Modul mit Daten, Prozeduren und Initialisierungscode
  - Zugriff auf die Daten nur über Monitor-Prozeduren
  - (entspricht in etwa einer Klasse)
- Alle Prozeduren stehen unter wechselseitigem Ausschluß
- Weitergehende Synchronisation durch **Bedingungsvariable**
  - zwei Operationen:
    - `wait()`: Blockieren des aufrufenden Threads
    - `signal()`: Aufwecken blockierter Threads
      - Varianten: wecke nur einen / wecke alle Threads
  - kein „Gedächtnis“: `signal()` weckt nur einen Thread, der `wait()` bereits aufgerufen hat

Barriere

- Synchronisation von Gruppen von Prozessen bzw. Threads
- Semantik:
  - Thread, der Barriere erreicht, wird blockiert,
    bis auch alle anderen Threads die Barriere erreicht haben



- Strukturierung nebenläufiger Anwendungen in synchrone Phasen

2.1.2 Synchronisationsfehler

- Unzureichende Synchronisation: *Race Conditions*
  - Ergebnis der Berechnung ist je nach zeitlicher Verzahnung der Threads unterschiedlich (bzw. falsch)
  - Wichtig: keine FIFO-Semantik der Warteschlangen von Synchronisationskonstrukten voraussetzen!
- Verklemmungen (*Deadlocks*)
  - eine Gruppe von Threads wartet auf Bedingungen, die nur Threads in der Gruppe herstellen können
- Verhungerung (Unfairness)
  - ein Thread, der auf eine Bedingung wartet, kommt nie zum Zuge, obwohl die Bedingung regelmäßig erfüllt wird

Beispiel zu *Race Conditions*

- Aufgabe: zwei Threads so synchronisieren, daß sie abwechselnd etwas ausdrucken
- Falsche Lösung mit Semaphoren:

```
Semaphore s1 = 1;
Semaphore s2 = 0;
```

Thread 1

```
while (true) {
   P(s1);
   print("1");
   V(s2);
   V(s1);
}
```

Thread 2

```
while (true) {
   P(s2);
   P(s1);
   print("2");
   V(s1);
}
```

2.1.3 *Lock-free* bzw. *wait-free* Datenstrukturen

- Ziel: Datenstrukturen (typ. *Collections*) ohne wechselseitigem Ausschluss
  - performanter, keine Gefahr von Deadlocks
- *Lock-free*: unter **allen** Umständen macht mindestens ein Thread nach endlich vielen Schritten Fortschritt
  - *wait-free* verhindert zusätzlich auch Verhungerung
- Typische Vorgehensweise:
  - Verwendung atomarer *Read-Modify-Write*-Befehle anstelle von Sperren
  - im Konfliktfall, d.h. bei gleichzeitiger Änderung durch anderen Thread, wird die betroffene Operation wiederholt

Beispiel: (hinten) Anfügen an ein Array

```
int fetch_and_add(int* addr, int val) {
    int tmp = *addr;
    *addr += val;
    return tmp;
}
```

Atomar!

```
Data buffer[N];    // Puffer—Array
int wrPos = 0;     // Position für nächstes einzutragendes Element

void add_last(Data data) {
    int wrPosOld = fetch_and_add(&wrPos, 1);
    buffer[wrPosOld] = data;
}
```

Beispiel: (vorne) Anfügen an eine verkettete Liste

```
bool compare_and_swap(void* addr, void* exp, void* new) {
    if (*addr == *exp) {
        *addr = *new;
        return true;
    }
    return false;
}

Element* firstNode = NULL;          // Zeiger auf erstes Element

void add_first(Element* node) {
    Element* tmp;
    do {
      tmp = firstNode;
      node->next = tmp;
    } while (compare_and_swap(firstNode, tmp, node));
}
```

Atomar!

2.1.3 *Lock-free* bzw. *wait-free* Datenstrukturen ...

- Probleme
  - Wiederverwendung von Speicheradressen kann zu korrupten Datenstrukturen führen
    - Annahme bei verketteter Liste: wenn `firstNode` noch unverändert ist, wurde auf Liste nicht gleichzeitig zugegriffen
  - daher besondere Verfahren bei der Speicherfreigabe nötig

- Es gibt etliche Bibliotheken für C++ und auch für Java
  - C++: z.B. boost.lockfree, libcds, Concurrency Kit, liblfds
  - Java: z.B. Amino Concurrent Building Blocks, Highly Scalable Java

- Compiler stellen i.d.R. *Read-Modify-Write*-Operationen bereit
  - z.B. `gcc/g++`: eingebaute Funktionen `__sync_...()`

2.2 POSIX Threads (PThreads)

- IEEE 1003.1c: Standardschnittstelle zur Programmierung mit Threads
  - durch Systembibliothek realisiert
  - (weitestgehend) betriebssystemunabhängig
- Programmiermodell:
  - bei Programmstart: genau ein (Master-)Thread
  - Master-Thread erzeugt andere Threads
    und sollte i.d.R. auf deren Beendigung warten
  - Prozeß terminiert bei Beendigung des Master-Threads

Funktionen zur Threadverwaltung (unvollständig)

- `int pthread_create(pthread_t *thread,`
  `pthread_attr_t *attr,`
  `void *(*start_routine)(void *),`
  `void *arg)`
  - erzeugt einen neuen Thread
  - Eingabe-Parameter:
    - `attr`: Thread-Attribute
    - `start_routine`: Prozedur, die der Thread abarbeiten soll
    - `arg`: Parameter, der `start_routine` übergeben wird
  - Ergebnis:
    - `*thread`: Thread-*Handle* (= Referenz)

Funktionen zur Threadverwaltung ...

- `void pthread_exit(void *retval)`
  - eigene Terminierung (mit Ergebniswert retval)
- `int pthread_cancel(pthread_t thread)`
  - terminiert den spezifizierten Thread
  - Terminierung kann nur an bestimmten Punkten erfolgen
  - vor der Terminierung wird ein *Cleanup-Handler* gerufen, falls definiert
  - Terminierung kann maskiert werden
- `int pthread_join(pthread_t th, void **thread_return)`
  - wartet auf Terminierung eines Threads
  - liefert Ergebniswert in `*thread_return`

Beispiel: Hello World (☞ 02/helloThread.cpp)

```
#include <pthread.h>

void * SayHello(void *arg) {
  cout << "Hello World!\n";
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t t;
  if (pthread_create(&t, NULL, SayHello, NULL) != 0) {
    /* Fehler! */
  }
  pthread_join(t, NULL);
  return 0;
}
```

Parallelverarbeitung

WS 2015/16

16.11.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Beispiel: Summation eines Arrays mit mehreren Threads

(☞ 02/sum.cpp)

```
#include <pthread.h>

#define N 5
#define M 1000


/* Diese Funktion wird von jedem Thread ausgeführt */
void *sum_line(void *arg)
{
  int *line = (int *)arg;
  int i;
  long sum = 0;

  for (i=0; i<M; i++)
    sum += line[i];

  return (void *)sum;     /* Summe zurückliefern, beenden. */
}
```

```
/* Initialisieren des Arrays */
void init_array(int array[N][M])
{
  ...
}

/* Hauptprogramm */
int main(int argc, char **argv)
{
  int array[N][M];
  pthread_t threads[N];
  int i;
  void *result;
  long sum = 0;

  init_array(array);   /* Feld initialisieren */
```

```
  /* Thread f. jede Zeile erzeugen, Zeiger auf Zeile als Parameter */
  /* Achtung: Fehlerprüfungen und Threadattribute fehlen! */
  for (i=0; i<N; i++) {
    pthread_create(&threads[i], NULL, sum_line, array[i]);
  }

  /* Auf Terminierung warten, Ergebnisse summieren */
  for (i=0; i<N; i++) {
    pthread_join(threads[i], &result);
    sum += (long)result;
  }

  cout << "Summe: " << sum << "\n";
}
```

Übersetzen und Binden des Programms

```
➥ g++ -o sum sum.c -pthread
```

Anmerkungen zum Beispiel

- PThreads erlaubt nur die Übergabe *eines* Arguments
  - i.a. Zeiger auf Argument-Datenstruktur
    - darf nur dann lokale Variable sein, wenn `pthread_join` innerhalb der Prozedur erfolgt!
- Rückgabewert eines Threads ist `void *`
  - unschöne Typumwandlung
  - bei mehreren Rückgabewerten: Zeiger auf Datenstruktur (global oder dynamisch allokiert!)
- Keine Synchronisation erforderlich
- `pthread_join` kann nur auf einen bestimmten Thread warten
  - ineffizient bei unterschiedlicher Laufzeit der Threads

Synchronisation: Mutex-Variable

- Verhalten wie binäres Semaphor
  - Zustände: gesperrt, frei; Initialzustand: frei
- Deklaration und Initialisierung (globale/statische Variable):
  `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Operationen:
  - Sperren: `pthread_mutex_lock(&mutex)`
    - Verhalten bei rekursiver Belegung nicht festgelegt
  - Freigeben: `pthread_mutex_unlock(&mutex)`
    - bei Terminierung eines Threads werden Sperren *nicht* automatisch freigegeben!
  - Sperrversuch: `pthread_mutex_trylock(&mutex)`
    - blockiert nicht, liefert ggf. Fehlercode

Synchronisation: Bedingungsvariablen

- Deklaration und Initialisierung (globale/statische Variable):
  `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Operationen:
  - Warten: `pthread_cond_wait(&cond, &mutex)`
    - Thread wird blockiert, `mutex` wird temporär freigegeben
    - signalisierender Thread behält `mutex`, d.h. Bedingung ist nach Rückkehr der Funktion ggf. nicht mehr erfüllt!
    - typische Verwendung:

```
while (!bedingungErfuellt)
  pthread_cond_wait(&cond, &mutex);
```

  - Signalisieren:
    - eines Threads: `pthread_cond_signal(&cond)`
    - aller Threads: `pthread_cond_broadcast(&cond)`

Beispiel: Nachbildung eines Monitors mit PThreads

(☞ 02/monitor.cpp)

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
volatile int ready = 0;

volatile int ergebnis;

void StoreResult(int arg)
{
    pthread_mutex_lock(&mutex);
    ergebnis = arg;   /* speichere Ergebnis */
    ready = 1;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Beispiel: Nachbildung eines Monitors mit PThreads ...

```
int ReadResult()
{
    int tmp;
    pthread_mutex_lock(&mutex);
    while (ready != 1)
        pthread_cond_wait(&cond, &mutex);
    tmp = ergebnis;   /* lies Ergebnis */
    pthread_mutex_unlock(&mutex);
    return tmp;
}
```

- `while` ist wichtig, da wartender Thread `mutex` aufgibt
  - ein anderer Thread könnte Bedingung wieder zerstören, bis wartender Thread `mutex` wieder erhält
    (allerdings nicht in diesem konrekten Fall!)

Hintergrund

- Thread-Bibliotheken (für FORTRAN und C) sind für Anwendungsprogrammierer oft zu komplex (und z.T. systemabhängig)
  - gesucht: abstraktere, portable Konstrukte
- OpenMP ist ein Quasi-Standard
  - seit 1997 durch OpenMP-Forum (`www.openmp.org`)
- API zur speichergekoppelten, parallelen Programmierung mit FORTRAN / C / C++
  - **Quellcode-Direktiven**
  - Bibliotheks-Funktionen
  - Umgebungsvariablen
- Unterstützt neben paralleler Ausführung mit Threads auch SIMD-Erweiterungen und externe Beschleuniger (ab Version 4.0)

Direktivengesteuerte Parallelisierung

- Der Programmierer muß spezifizieren
  - welche Coderegionen parallel ausgeführt werden sollen
  - wo eine Synchronisation erforderlich ist
- Diese Spezifikation erfolgt durch **Direktiven** (**Pragmas**)
  - spezielle Steueranweisungen an den Compiler
  - unbekannte Direktiven werden vom Compiler ignoriert
- Damit: ein Programm mit OpenMP-Direktiven kann
  - mit einem OpenMP-Compiler in ein paralleles Programm
  - mit einem Standard-Compiler in ein sequentielles Programm

übersetzt werden

Direktivengesteuerte Parallelisierung ...

- Ziel der Parallelisierung mit OpenMP:
  - Ausführung des sequentiellen Programmcodes auf mehrere Threads aufteilen, ohne den Code zu ändern
  - identischer Quellcode für sequentielle und parallele Version
- Im Wesentlichen drei Klassen von Direktiven:
  - Direktive zur Thread-Erzeugung (`parallel`, parallele Region)
  - Innerhalb einer parallelen Region: Direktiven zur Aufteilung der Arbeit auf die einzelnen Threads
    - Datenparallelität: Verteilung von Schleifeniterationen (`for`)
    - Taskparallelität: parallele Coderegionen (`sections`) und explizite Tasks (`task`)
  - Direktiven zur Synchronisation

Direktivengesteuerte Parallelisierung: Diskussion

- Mittelweg zwischen
  - vollständig manueller Parallelisierung (wie z.B. bei MPI)
  - automatischer Parallelisierung durch den Compiler
- Compiler übernimmt die Organisation der parallelen Tasks
  - Thread-Erzeugung, Verteilung von Tasks, ...
- Programmierer übernimmt die erforderliche Abhängigkeits-analyse
  - welche Coderegionen können parallel ausgeführt werden?
  - erlaubt detaillierte Kontrolle über Parallelismus
  - aber: Programmierer ist verantwortlich für Korrektheit

2.3.1 Die `parallel`-Direktive

Ein Beispiel (☞ `02/firstprog.cpp`)

Programm

```
main() {
   cout << "Serial\n";

   {
      cout << "Parallel\n";
   }
   cout << "Serial\n";
}
```

2.3.1 Die `parallel`-Direktive

Ein Beispiel (☞ `02/firstprog.cpp`)

Programm

```
main() {
   cout << "Serial\n";
   #pragma omp parallel
   {
      cout << "Parallel\n";
   }
   cout << "Serial\n";
}
```

Übersetzung

```
g++ -fopenmp -o tst
    firstprog.cpp
```

2.3.1 Die `parallel`-Direktive

Ein Beispiel (☞ 02/firstprog.cpp)

Programm

```
main() {
   cout << "Serial\n";
   #pragma omp parallel
   {
      cout << "Parallel\n";
   }
   cout << "Serial\n";
}
```

Übersetzung

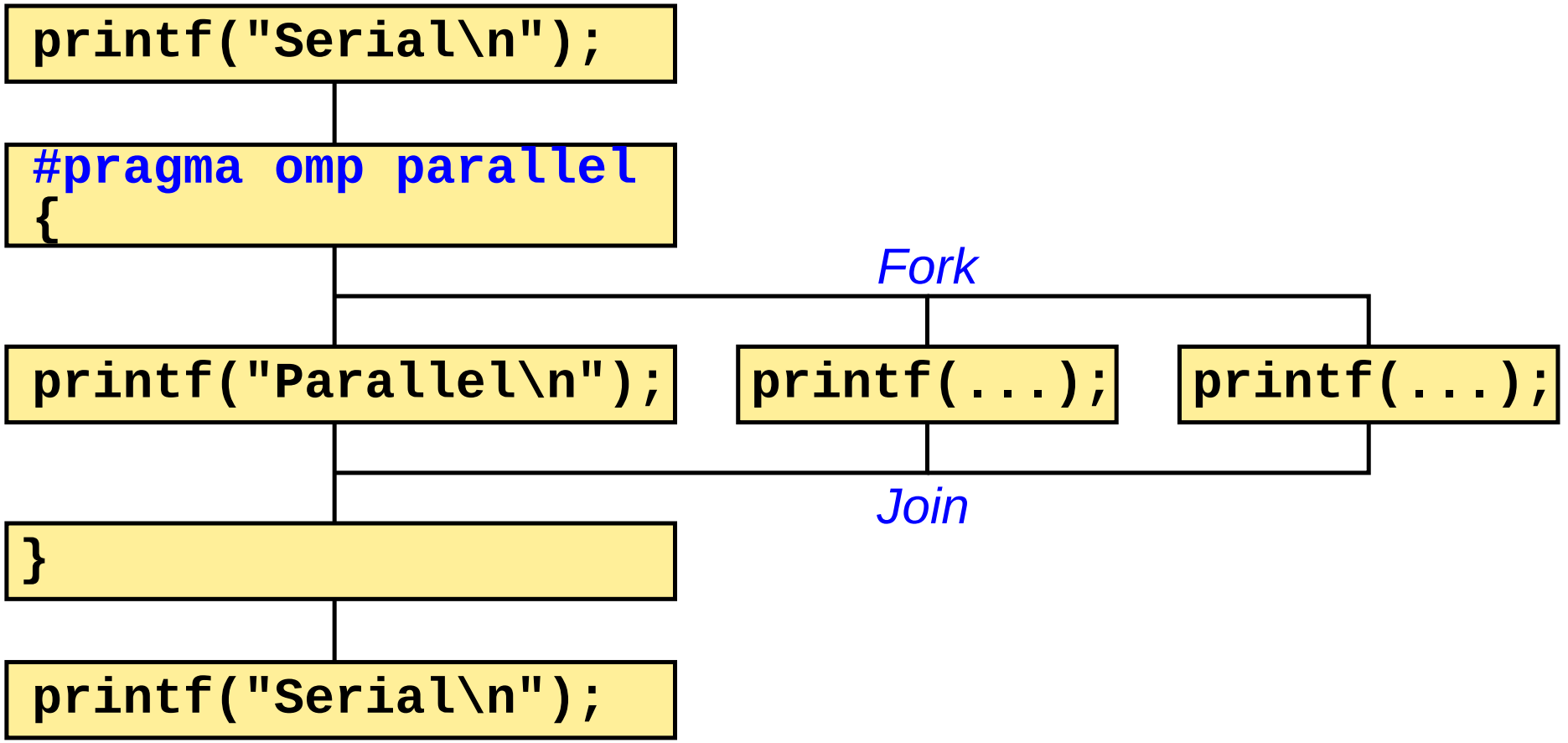
```
g++ -fopenmp -o tst
    firstprog.cpp
```

Aufruf

```
% export OMP_NUM_THREADS=2
% ./tst
Serial
Parallel
Parallel
Serial
```

```
% export OMP_NUM_THREADS=3
% ./tst
Serial
Parallel
Parallel
Parallel
Serial
```

Ausführungsmodell: *Fork-Join*



Ausführungsmodell: *Fork-Join* ...

- Programm startet mit genau einem *Master*-Thread
- Wenn eine parallele Region (`#pragma omp parallel`) erreicht wird, werden weitere Threads erzeugt (**Team**)
  - Umgebungsvariable `OMP_NUM_THREADS` legt Gesamtzahl fest
- Parallele Region wird von allen Threads im Team ausgeführt
  - zunächst redundant, weitere OpenMP Direktiven erlauben Arbeitsaufteilung
- Am Ende der parallelen Region:
  - alle Threads bis auf den *Master* werden beendet
  - *Master* wartet, bis alle anderen Threads beendet sind (Join)

2.3.1 Die `parallel`-Direktive ...

Zur Syntax der Direktiven (in C / C++)

- `#pragma omp <directive> [ <clause_list> ]`
  - `<clause_list>`: Liste von Optionen für die Direktive
- Direktive wirkt nur auf die unmittelbar folgende Anweisung bzw. den unmittelbar folgenden Block
  - **statischer Bereich** (***static extent***) der Direktive

```
#pragma omp parallel
cout << "Hallo\n"; // parallel
cout << "Huhu\n";   // ab hier wieder sequentiell
```

- **Dynamischer Bereich** (***dynamic extent***) einer Direktive
  - umfaßt auch im statischen Bereich aufgerufene Funktionen (die somit parallel ausgeführt werden)

Gemeinsame und private Variablen

- Für Variablen in einer parallelen Region gibt es zwei Alternativen
  - Variable ist allen Threads gemeinsam (*shared variable*)
    - alle Threads greifen auf dieselbe Variable zu
      - in der Regel Synchronisation notwendig!
  - jeder Thread hat eine private Instanz (*private variable*)
    - kann mit Wert des *Master*-Threads initialisiert werden
    - Wert wird am Ende der parallelen Region verworfen
- Für Variablen, die **innerhalb** des dynamischen Bereichs einer `parallel`-Direktive deklariert werden, gilt:
  - lokale Variable sind privat
  - `static`-Variable und Heap-Variable (`malloc()`) sind *shared*

Gemeinsame und private Variablen ...

- Für Variablen, die **vor dem Eintritt** in einen parallelen Bereich deklariert wurden, kann in der `parallel`-Direktive das Verhalten durch Optionen festgelegt werden:
  - `private ( <variable_list> )`
    - private Variable, ohne Initialisierung
  - `firstprivate ( <variable_list> )`
    - private Variable
    - Initialisierung mit dem Wert aus dem *Master*-Thread
  - `shared ( <variable_list> )`
    - gemeinsame Variable
    - `shared` ist die Voreinstellung für alle Variablen

Gemeinsame und private Variablen: Beispiel (☞ `02/private.cpp`)

Jeder Thread erhält eine (nicht initialisierte) Kopie von i

Jeder Thread erhält eine initialisierte Kopie von j

```
int i = 0, j = 1, k = 2;
#pragma omp parallel private(i) firstprivate(j)
{
   int h = random() % 100;
   cout << "P: i=" << i << ", j=" << j
        << ", k=" << k << ", h=" << h << "\n";
   i++; j++; k++;
}
cout << "S: i=" << i << ", j=" << j
     << ", k=" << k << "\n";
```

h ist privat

Zugriffe auf k müssten synchronisiert werden!

Ausgabe (mit 2 Threads):

```
P: i=1023456, j=1, k=2, h=86
P: i=-123059, j=1, k=3, h=83
S: i=0, j=1, k=4
```

2.3.2 Bibliotheks-Funktionen

- OpenMP definiert auch einige Bibliotheks-Funktionen, z.B.:
  - `int omp_get_num_threads()`: liefert Anzahl der Threads
  - `int omp_get_thread_num()`: liefert Thread-Nummer
    - von 0 (*Master*-Thread) bis `omp_get_num_threads()-1`
  - `int omp_get_num_procs()`: Anzahl der Prozessoren
  - `void omp_set_num_threads(int nthreads)`
    - setzt Zahl der Threads (maximal `OMP_NUM_THREADS`)
  - `double omp_get_wtime()`: Uhrzeit in Sekunden
    - für Laufzeit-Messungen
  - zusätzlich: Funktionen für Mutex-Locks
- Bei Verwendung der Bibliotheks-Funktionen ist der Code aber nicht mehr ohne OpenMP übersetzbar ...

Beispiel zu Bibliotheks-Funktionen (☞ 02/threads.cpp)

```
#include <omp.h>
int me;
omp_set_num_threads(2);   // nur 2 Threads benutzen
#pragma omp parallel private(me)
{
   me = omp_get_thread_num();   // Eigene Thread−Nr. (0 oder 1)
   cout << "Thread " << me << "\n";
   if (me == 0)      // Unterschiedlicher Thread−Code!
      cout << "Here is the master thread\n";
   else
      cout << "Here is the other thread\n";
}
```

- Zur Benutzung der Bibliotheks-Funktionen muß die Header- Datei `omp.h` eingebunden werden

Motivation

- Umsetzung von Datenparallelität
  - Threads führen identische Berechnungen auf einem Teil der Daten durch
- Zwei mögliche Ansätze:
  - Betrachte primär die Daten, teile diese auf
    - Aufteilung der Berechnungen ergibt sich daraus
    - z.B. bei HPF oder MPI
  - Betrachte primär die Berechnungen, teile diese auf
    - Berechnungen finden praktisch immer in Schleifen statt ($\Rightarrow$ Schleifenparallelisierung)
    - keine explizite Aufteilung der Daten
    - für Programmiermodelle mit gemeinsamem Speicher

Parallelverarbeitung

WS 2015/16

23.11.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

2.4.1 Die `for`-Direktive: Parallele Schleifen

```
#pragma omp for [<clause_list>]
for(...) ...
```

- Muß im dynamischen Bereich einer `parallel`-Direktive stehen
- Ausführung der Schleifeniterationen wird auf alle Threads verteilt
  - Schleifenvariable ist automatisch privat
- Nur für „einfache“ Schleifen erlaubt
  - kein `break`, ganzzahlige Schleifenvariable, ...
- Keine Synchronisation am Beginn der Schleife
- Barrieren-Synchronisation am Ende der Schleife
  - außer Option `nowait` wird angegeben

Beispiel: Vektor-Addition

```
double a[N], b[N], c[N];
int i;
#pragma omp parallel for
for (i=0; i<N; i++) {
   a[i] = b[i] + c[i];
}
```

Kurzform für

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

- Jeder Thread arbeitet einen Teil jedes Vektors ab
  - Datenaufteilung, datenparalleles Modell
- Frage: wie genau werden die Iterationen auf die Threads verteilt?
  - kann durch die `schedule`-Option festgelegt werden
  - Default: bei n Threads erhält Thread 1 das erste n-tel der Iterationen, Thread 2 das zweite, ...

Scheduling von Schleifen-Iterationen

- Option `schedule( <klasse> [ , <groesse> ] )`
- Scheduling-Klassen:
  - `static`: Blöcke gegebener Größe (optional) werden vor der Ausführung reihum auf die Threads verteilt
  - `dynamic`: Aufteilung der Iterationen in Blöcke gegebener Größe, Abarbeitung gemäß *Work-Pool*-Modell
    - besserer Lastausgleich, wenn Iterationen unterschiedlich viel Rechenzeit benötigen
  - `guided`: wie `dynamic`, aber Blockgröße nimmt exponentiell ab (kleinste Größe kann vorgegeben werden)
    - besserer Lastausgleich als bei gleich großen Blöcken
  - `auto`: Festlegung durch Compiler / Laufzeitsystem
  - `runtime`: Festlegung durch Umgebungsvariable

Beispiel zum Scheduling (☞ 02/loops.cpp)

```
int i, j;
double x;

#pragma omp parallel for private(i,j,x) schedule(runtime)
for (i=0; i<40000; i++) {
    x = 1.2;
    for (j=0; j<i; j++) {          // Dreiecksschleife
        x = sqrt(x) * sin(x*x);
    }
}
```

- Scheduling kann zur Laufzeit festgelegt werden, z.B.:
  - `export OMP_SCHEDULE="static,10"`
- Praktisch für Optimierungs-Versuche

Beispiel zum Scheduling: Resultate

- Laufzeiten mit je 4 Threads auf den Laborrechnern:

`OMP_SCHEDULE`	`"static"`	`"static,1"`	`"dynamic"`	`"guided"`
Zeit	3.1 s	1.9 s	1.8 s	1.8 s

- Lastungleichgewicht bei `"static"`
  - Thread 1: i=0..9999, Thread 4: i=30000..39999
- `"static,1"` und `"dynamic"` verwenden Blockgröße 1
  - Thread rechnet jede vierte Iteration der `i`-Schleife
  - wegen Caches ggf. sehr ineffizient (*False Sharing*, ☞ **4.1**)
    - Abhilfe ggf. größere Blockgröße (z.B.: `"dynamic,100"`)
- `"guided"` ist oft ein guter Kompromiß zwischen Lastausgleich und Lokalität (Cache-Nutzung)

Gemeinsame und private Variablen in Schleifen

- Die `for`-Direktive kann mit den Optionen `private`, `shared` und `firstprivate` versehen werden (siehe Folie 179 ff.)
- Zusätzlich gibt es eine Option `lastprivate`
  - private Variable
  - Master-Thread hat nach der Schleife den Wert aus der letzten Iteration
- Beispiel:

```
int i = 0;
#pragma omp parallel for lastprivate(i)
for (i=0; i<100; i++) {
    ...
}
printf("i=%d\n",i);          // druckt den Wert 100
```

2.4.2 Parallelisierbarkeit von Schleifen

Wann kann eine Schleife parallelisiert werden?

```
for(i=1;i<N;i++)
   a[i] = a[i]
        + b[i-1];
```

```
for(i=1;i<N;i++)
   a[i] = a[i-1]
        + b[i];
```

```
for(i=0;i<N;i++)
   a[i] = a[i+1]
        + b[i];
```

- Optimal: unabhängige Schleife (**forall**-Schleife)
  - Iterationen der Schleife können ohne Synchronisation nebenläufig ausgeführt werden
  - zwischen Anweisungen in **verschiedenen** Iterationen dürfen keine Datenabhängigkeiten bestehen
  - (äquivalent: Anweisungen in verschiedenen Iterationen müssen die **Bernstein-Bedingungen** erfüllen)

2.4.2 Parallelisierbarkeit von Schleifen

Wann kann eine Schleife parallelisiert werden?

```
for(i=1;i<N;i++)
   a[i] = a[i]
        + b[i-1];
```

keine Abhängigkeiten

```
for(i=1;i<N;i++)
   a[i] = a[i-1]
        + b[i];
```

echte Abhängigkeit

```
for(i=0;i<N;i++)
   a[i] = a[i+1]
        + b[i];
```

Antiabhängigkeit

- Optimal: unabhängige Schleife (**forall**-Schleife)
  - Iterationen der Schleife können ohne Synchronisation nebenläufig ausgeführt werden
  - zwischen Anweisungen in **verschiedenen** Iterationen dürfen keine Datenabhängigkeiten bestehen
  - (äquivalent: Anweisungen in verschiedenen Iterationen müssen die **Bernstein-Bedingungen** erfüllen)

Behandlung von Datenabhängigkeiten in Schleifen

- Bei Anti- und Ausgabeabhängigkeiten:
  - Auflösung ist immer möglich, z.B. durch Umbenennung von Variablen
  - im Beispiel von vorher:

```
#pragma omp parallel
{
   #pragma omp for
   for(i=1;i<=N;i++)
      a2[i] = a[i];
   #pragma omp for
   for(i=0;i<N;i++)
      a[i] = a2[i+1] + b[i];
}
```

  - Barriere am Ende der ersten Schleife ist notwendig!

Behandlung von Datenabhängigkeiten in Schleifen ...

- Bei echten Abhängigkeiten:
  - geeignete Synchronisation zwischen den Threads einführen
    - z.B. mit `ordered`-Direktive (☞ **2.6**):

```
#pragma omp parallel for ordered
for(i=1;i<N;i++){
   // lange Berechung von b[i]
   #pragma omp ordered
   a[i] = a[i-1] + b[i];
}
```

    - Nachteil: oft deutlich geringerer Parallelitätsgrad
  - ggf. ist Vektorisierung (SIMD) möglich (☞ **2.8.2**), z.B.:

```
#pragma omp simd safelen(4)
for(i=4;i<N;i++)
  a[i] = a[i-4] + b[i];
```

Matrix-Addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Matrix-Addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Keine Abhängigkeiten in j−Schleife:

- b wird nur gelesen
- Elemente von a immer nur in derselben j−Iteration gelesen, in der sie geschrieben werden

Matrix-Addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Keine Abhängigkeiten in j−Schleife:
- b wird nur gelesen
- Elemente von a immer nur in derselben j−Iteration gelesen, in der sie geschrieben werden

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
#pragma omp parallel for
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Innere Schleife kann parallel bearbeitet werden

Matrix-Addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Keine Abhängigkeiten in i−Schleife:

- b wird nur gelesen
- Elemente von a immer nur in derselben i−Iteration gelesen, in der sie geschrieben werden

Matrix-Addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Keine Abhängigkeiten in i−Schleife:
- b wird nur gelesen
- Elemente von a immer nur in derselben i−Iteration gelesen, in der sie geschrieben werden

```
double a[N][N];
double b[N][N];
int i,j;

#pragma omp parallel for
                    private(j)
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Äußere Schleife kann parallel bearbeitet werden

Vorteil: weniger Overhead!

Matrix-Multiplikation

```
double a[N][N], b[N][N], c[N][N];
int i,j,k;
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    c[i][j] = 0;
    for (k=0; k<N; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
  }
}
```

echte Abhängigkeiten über k−Schleife

Keine Abhängigkeiten über die i− und j−Schleifen

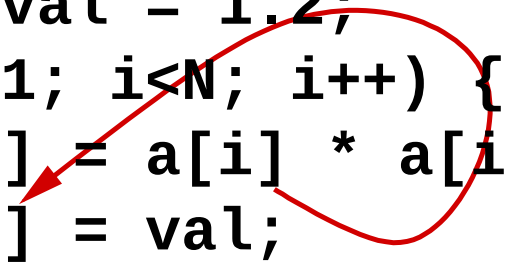
- Die `i`- und die `j`-Schleife können parallel ausgeführt werden
- I.A. Parallelisierung der äußeren Schleife, da geringerer Overhead

Auflösung von Abhängigkeiten

```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```

Auflösung von Abhängigkeiten

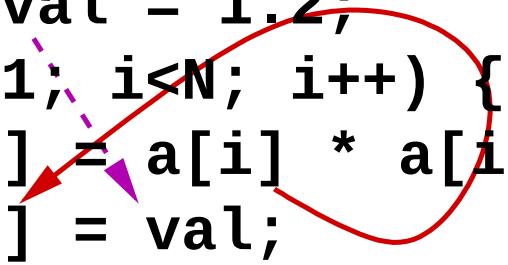
```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```



Antiabhängigkeit zw. Iterationen

Auflösung von Abhängigkeiten

```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```

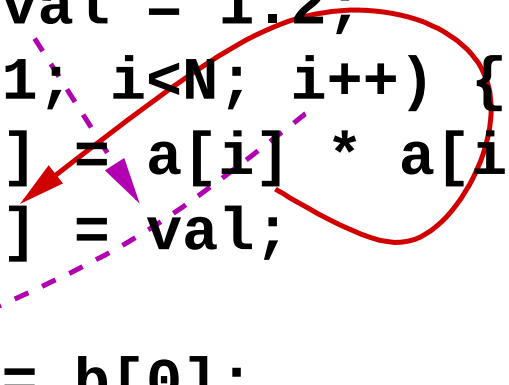


Antiabhängigkeit zw. Iterationen

Echte Abhängigkeiten zwischen Schleife und Umgebung

Auflösung von Abhängigkeiten

```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```

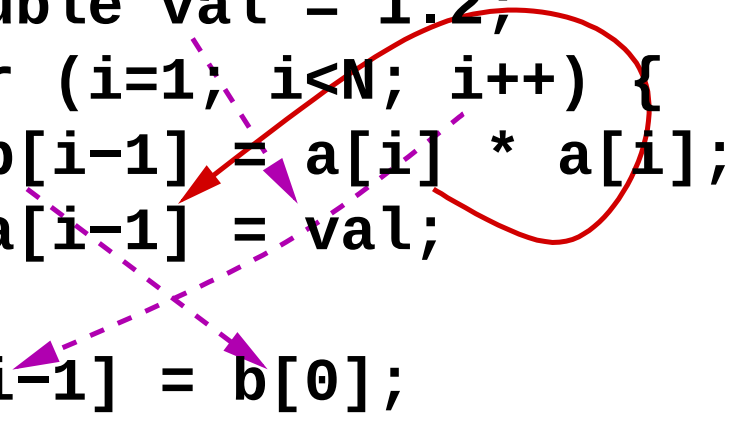


Antiabhängigkeit zw. Iterationen

Echte Abhängigkeiten zwischen Schleife und Umgebung

Auflösung von Abhängigkeiten

```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```

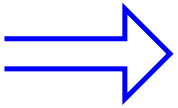


Antiabhängigkeit zw. Iterationen

Echte Abhängigkeiten zwischen Schleife und Umgebung

Auflösung von Abhängigkeiten

```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```



```
double a[N], b[N];
int i;
double val = 1.2;
#pragma omp parallel
{
  #pragma omp for
  for (i=1; i<N; i++)
    b[i-1] = a[i] * a[i];
  #pragma omp for
            lastprivate(i)
  for (i=1; i<N; i++)
    a[i-1] = val;
}
a[i-1] = b[0];
```

Antiabhängigkeit zw. Iterationen → Trennung der Schleife + Barriere

Echte Abhängigkeiten zwischen Schleife und Umgebung → lastprivate(i) + Barrieren

Richtungsvektoren

- Ist Abhängigkeit innerhalb einer Iteration oder über verschiedene Iterationen hinweg?

```
   for (i=0; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i] * 5;
   }
```

```
   for (i=1; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i-1] * 5;
   }
```

```
   for (i=1; i<N; i++) {
     for (j=1; j<N; j++) {
S1:    a[i][j] = b[i][j] + 2;
S2:    b[i][j] = a[i-1][j-1] - b[i][j];
     }
   }
```

Richtungsvektoren

- Ist Abhängigkeit innerhalb einer Iteration oder über verschiedene Iterationen hinweg?

```
   for (i=0; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i] * 5;
   }
```

S1 $\delta^{t}_{(=)}$ S2

Richtungsvektor:
S1 und S2 in derselben Iteration

```
   for (i=1; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i-1] * 5;
   }
```

```
   for (i=1; i<N; i++) {
     for (j=1; j<N; j++) {
S1:    a[i][j] = b[i][j] + 2;
S2:    b[i][j] = a[i-1][j-1] - b[i][j];
     }
   }
```

Richtungsvektoren

- Ist Abhängigkeit innerhalb einer Iteration oder über verschiedene Iterationen hinweg?

```
   for (i=0; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i] * 5;
   }
```

S1 $\delta^t_{(=)}$ S2

Richtungsvektor:

S1 und S2 in derselben Iteration

```
   for (i=1; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i-1] * 5;
   }
```

Schleifengetragene Abhängigkeit

S1 in früherer Iteration als S2

S1 $\delta^t_{(<)}$ S2

```
   for (i=1; i<N; i++) {
     for (j=1; j<N; j++) {
S1:    a[i][j] = b[i][j] + 2;
S2:    b[i][j] = a[i-1][j-1] - b[i][j];
     }
   }
```

Richtungsvektoren

- Ist Abhängigkeit innerhalb einer Iteration oder über verschiedene Iterationen hinweg?

```
   for (i=0; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i] * 5;
   }
```

S1 $\delta^{t}_{(=)}$ S2

Richtungsvektor:
S1 und S2 in derselben Iteration

```
   for (i=1; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i-1] * 5;
   }
```

Schleifengetragene Abhängigkeit

S1 in früherer Iteration als S2

S1 $\delta^{t}_{(<)}$ S2

```
   for (i=1; i<N; i++) {
     for (j=1; j<N; j++) {
S1:    a[i][j] = b[i][j] + 2;
S2:    b[i][j] = a[i-1][j-1] - b[i][j];
     }
   }
```

S1 $\delta^{a}_{(=,=)}$ S2

Richtungsvektoren

- Ist Abhängigkeit innerhalb einer Iteration oder über verschiedene Iterationen hinweg?

```
   for (i=0; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i] * 5;
   }
```

S1 $\delta^t_{(=)}$ S2

Richtungsvektor:
S1 und S2 in derselben Iteration

```
   for (i=1; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i-1] * 5;
   }
```

Schleifengetragene Abhängigkeit

S1 in früherer Iteration als S2

S1 $\delta^t_{(<)}$ S2

```
   for (i=1; i<N; i++) {
     for (j=1; j<N; j++) {
S1:    a[i][j] = b[i][j] + 2;
S2:    b[i][j] = a[i-1][j-1] - b[i][j];
     }
   }
```

S1 in früherer Iteration der I- und J-Schleife als S2

S1 $\delta^t_{(<,<)}$ S2

S1 $\delta^a_{(=,=)}$ S2

Formale Berechnung von Abhängigkeiten

- Basis: Suche nach ganzzahligen Lösungen von (Un-)gleichungssystemen
- Beispiel:

```
for (i=0; i<10; i++ {
  for (j=0; j<i; j++) {
    a[i*10+j] = ...;
    ... = a[i*20+j-1];
  }
}
```

Gleichungssystem:

$$0 \leq i_1 < 10$$
$$0 \leq i_2 < 10$$
$$0 \leq j_1 < i_1$$
$$0 \leq j_2 < i_2$$
$$10\,i_1 + j_1 = 20\,i_2 + j_2 - 1$$

- Abhängigkeitsanalyse ist immer konservative Näherung!
  - unbekannte Schleifengrenzen, nichtlineare Indexausdrücke, Zeiger (Aliasing), ...

Anwendung: Anwendbarkeit von Code-Transformationen

- Zulässigkeit einer Code-Transformation richtet sich nach den (möglicherweise) vorhandenen Datenabhängigkeiten
- Z.B.: Parallele Abarbeitung einer Schleife zulässig, wenn
  - diese Schleife keine Abhängigkeiten trägt
  - d.h., alle Richtungsvektoren haben die Form $(\ldots, =, \ldots)$ oder $(\ldots, \neq, \ldots, *, \ldots)$ [rot: betrachtete Schleife]
- Z.B.: *Loop Interchange* zulässig, wenn
  - Schleifen perfekt geschachtelt
  - Grenzen der inneren Scheife unabh. von äußerer Schleife
  - keine Abhängigkeiten mit Richtungsvektor $(\ldots, <, >, \ldots)$

Beispiel: Block-Algorithmus für Matrix-Multiplikation

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,N
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

Strip Mining

```
DO I = 1,N
        ↓
DO IT = 1,N,IS
DO I = IT, MIN(N,IT+IS-1)
```

```
DO IT = 1,N,IS
DO I = IT, MIN(N,IT+IS-1)
  DO JT = 1,N,JS
  DO J = JT, MIN(N,JT+JS-1)
    DO KT = 1,N,KS
    DO K = KT, MIN(N,KT+KS-1)
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

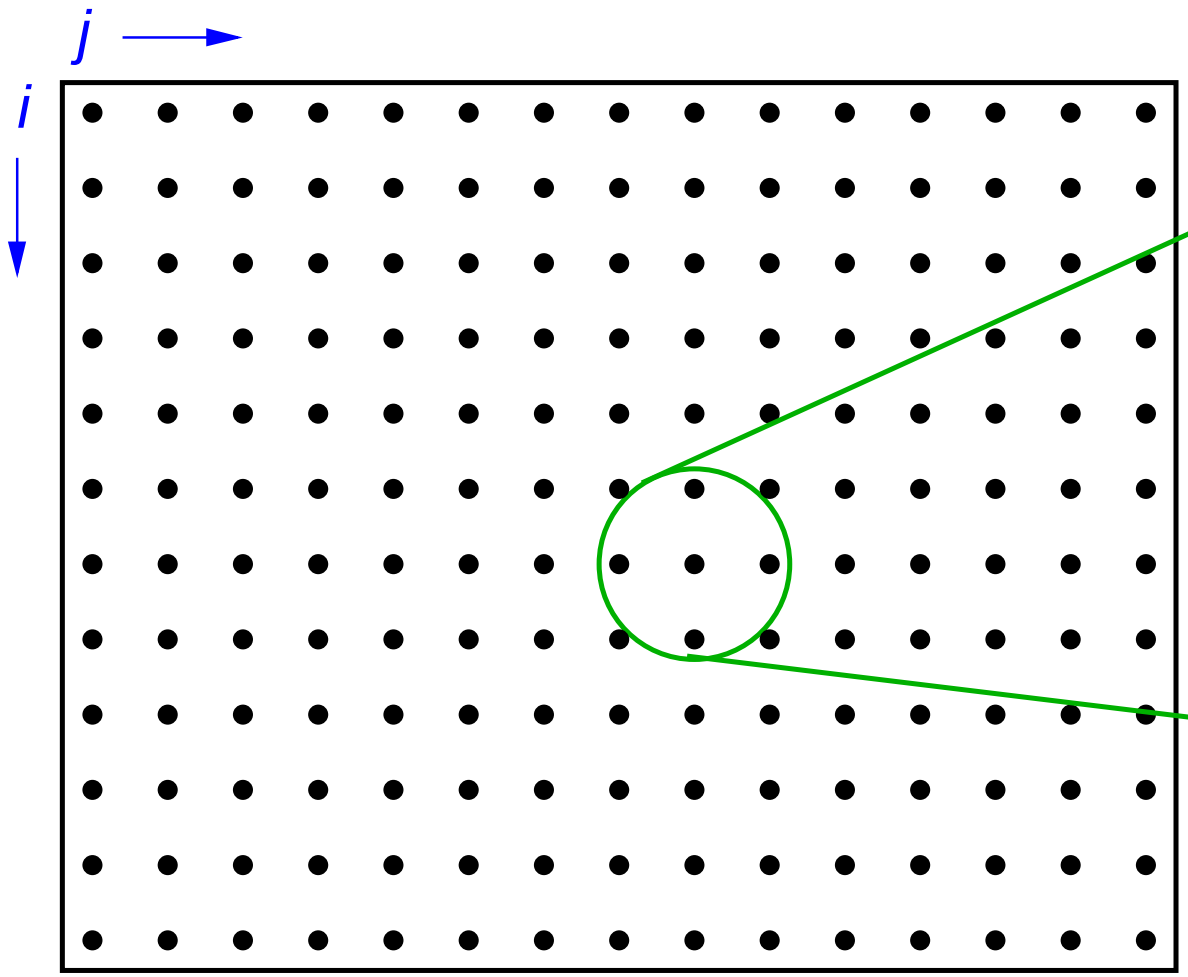
Loop Interchange

```
DO IT = 1,N,IS
DO JT = 1,N,JS
DO KT = 1,N,KS
  DO I = IT, MIN(N,IT+IS-1)
  DO J = JT, MIN(N,JT+JS-1)
  DO K = KT, MIN(N,KT+KS-1)
    A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

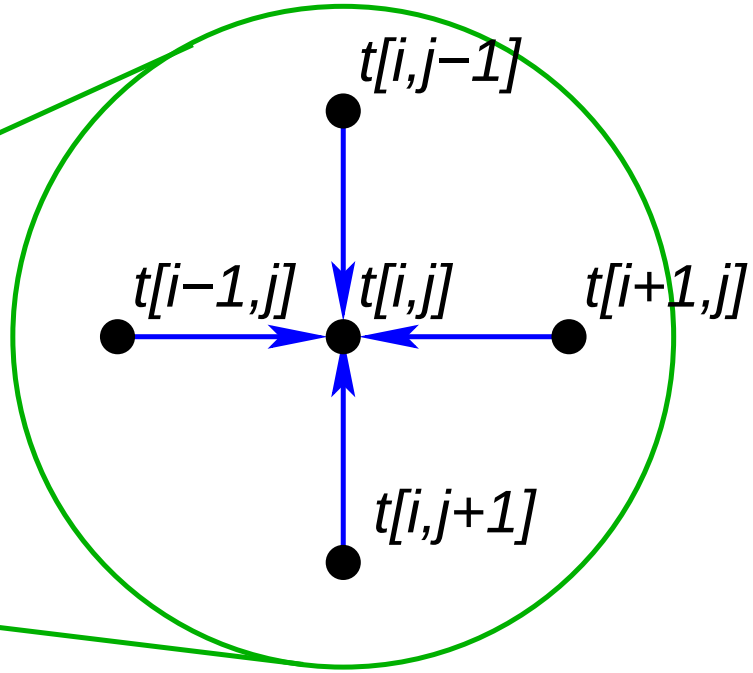
Numerische Lösung der Wärmeleitungsgleichungen

- Konkretes Problem: dünne Metallplatte
  - gegeben: Temperaturverlauf am Rand
  - gesucht: Temperatur im Inneren
- Ansatz:
  - Diskretisierung: Betrachte Temperatur nur bei äquidistanten Gitterpunkten
    - 2D-Array von Temperaturwerten
  - Iterative Lösung: Berechne immer bessere Näherungen
    - neue Näherung für Temperatur eines Gitterpunkts: Mittelwert der Temperaturen der Nachbarpunkte

Numerische Lösung der Wärmeleitungsgleichungen ...



Metallplatte



$$t[i,j] = 0.25 * (t[i-1,j] + t[i,j-1] + t[i,j+1] + t[i+1,j])$$

Varianten des Verfahrens

- **Jacobi-Iteration**
  - zur Berechnung der neuen Werte werden nur die Werte der letzten Iteration herangezogen
  - Berechnung auf zwei Matrizen
- **Gauss-Seidel-Relaxation**
  - zur Berechnung der neuen Werte werden auch schon Werte der aktuellen Iteration benutzt:
    - $t[i-1,j]$ und $t[i,j-1]$
  - Berechnung auf nur einer Matrix
  - meist schnellere Konvergenz

Varianten des Verfahrens ...

Jacobi

```
do {
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      b[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      a[i][j] = b[i][j];
    }
  }
} until (converged);
```

Gauss-Seidel

```
do {
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      a[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
} until (converged);
```

Parallelverarbeitung

WS 2015/16

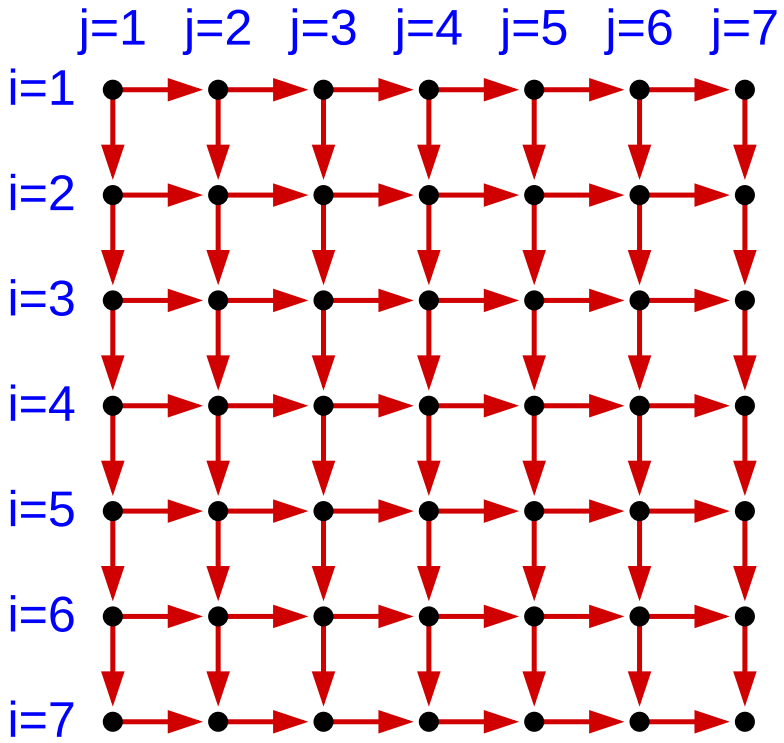
07.12.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Abhängigkeiten bei Jacobi und Gauss-Seidel

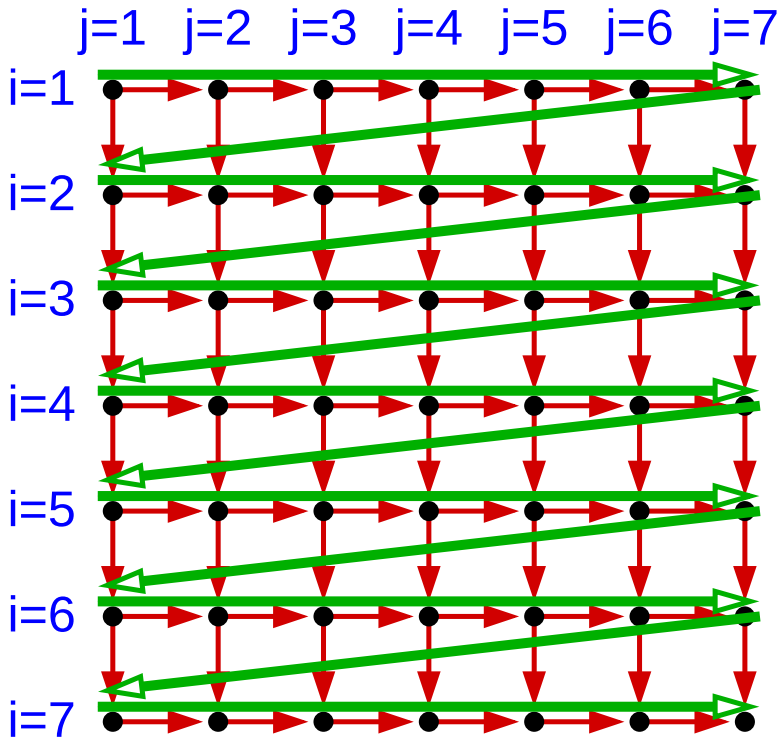
- Jacobi: nur zwischen den beiden i-Schleifen
- Gauss-Seidel: Iterationen der i, j-Schleife sind voneinander abhängig



Dargestellt sind die Schleifen–iterationen, nicht die Matrix–elemente!

Abhängigkeiten bei Jacobi und Gauss-Seidel

- Jacobi: nur zwischen den beiden i-Schleifen
- Gauss-Seidel: Iterationen der i, j-Schleife sind voneinander abhängig

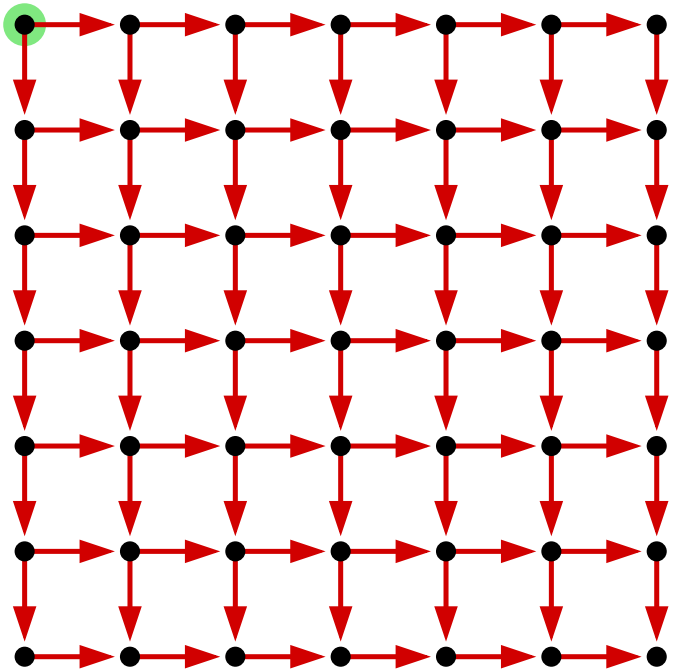


Sequentielle Ausführungs−reihenfolge

Dargestellt sind die Schleifen−iterationen, nicht die Matrix−elemente!

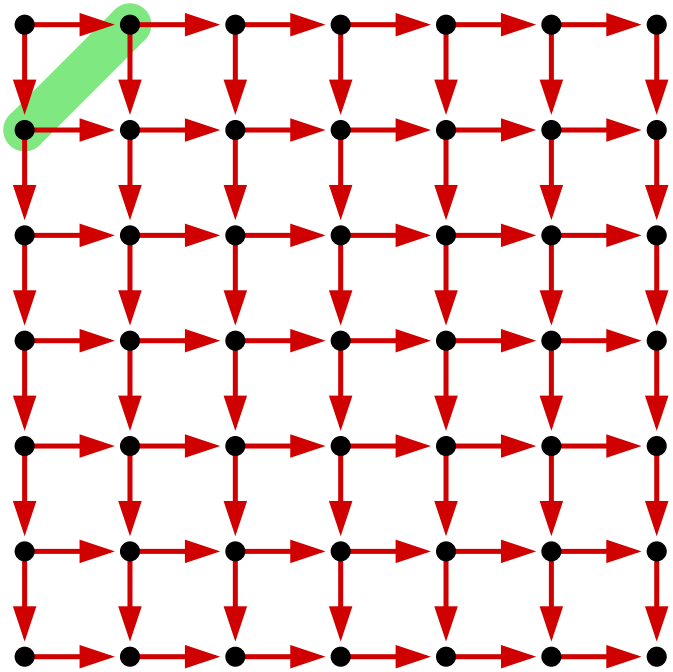
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



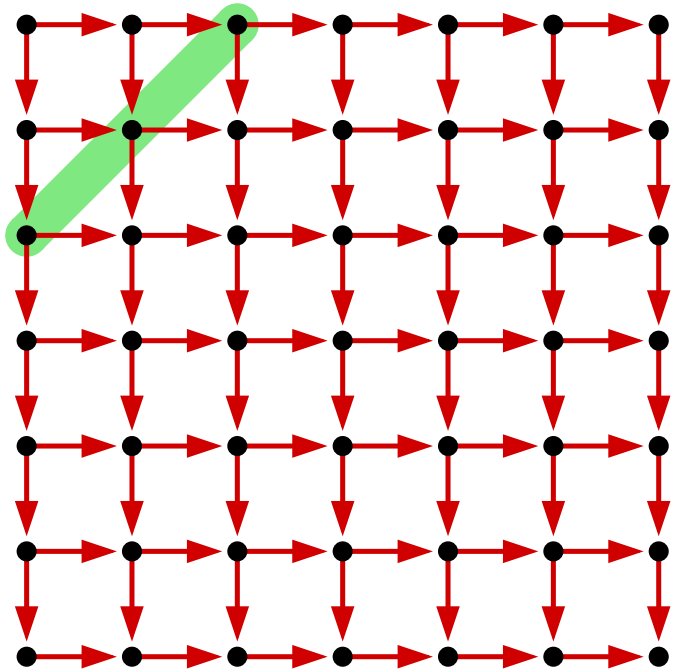
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



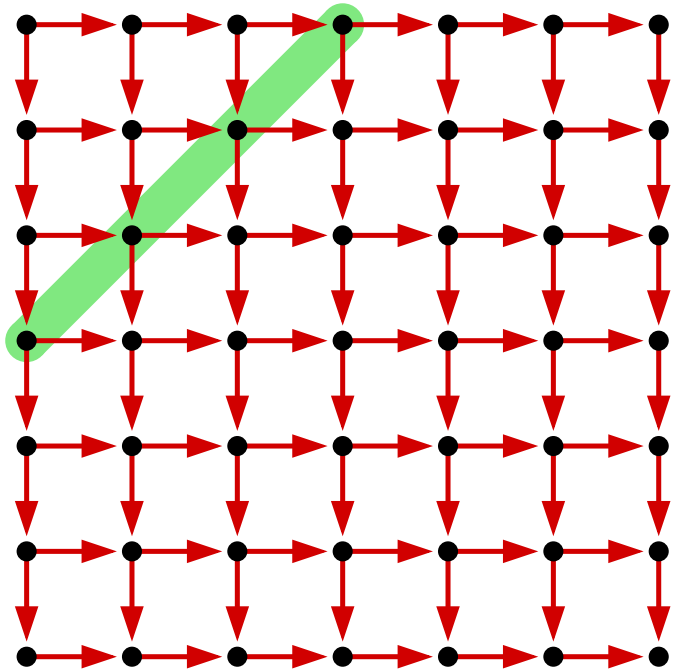
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



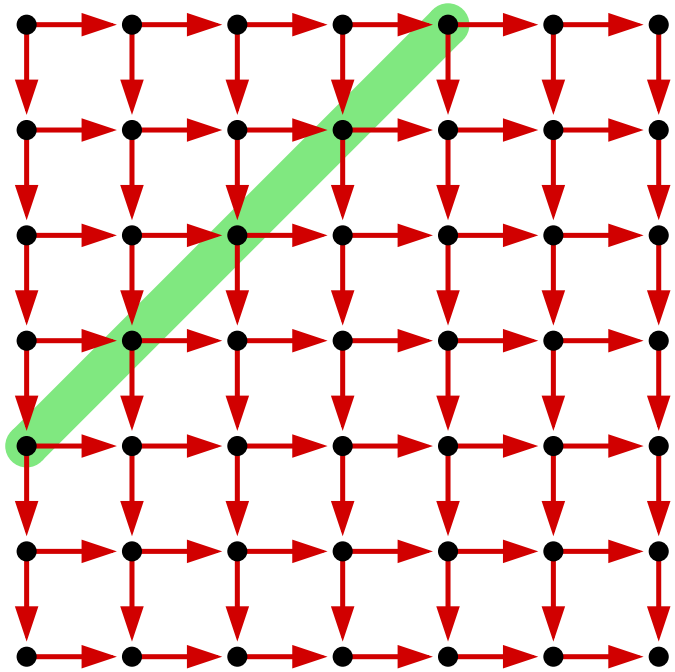
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



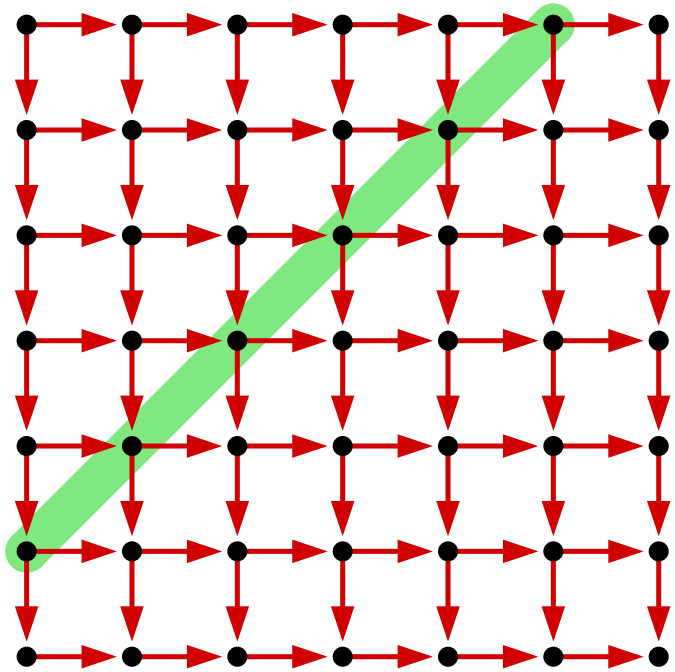
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



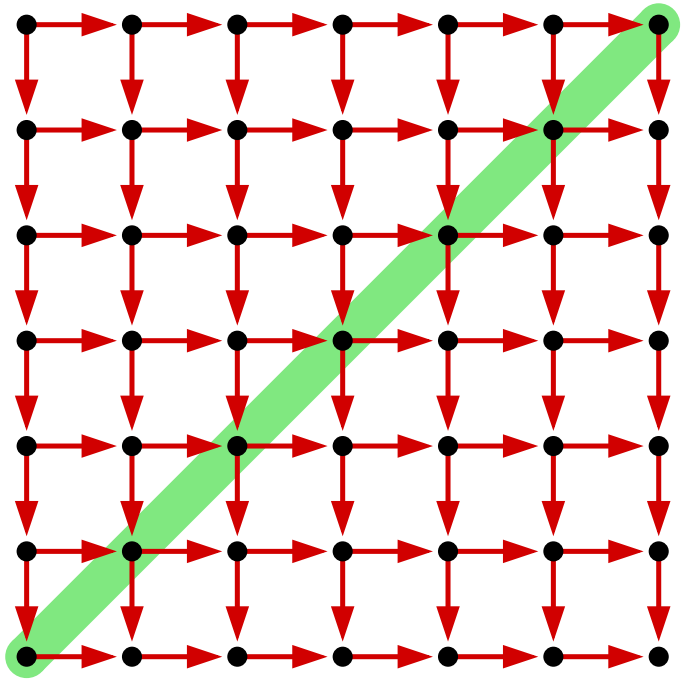
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



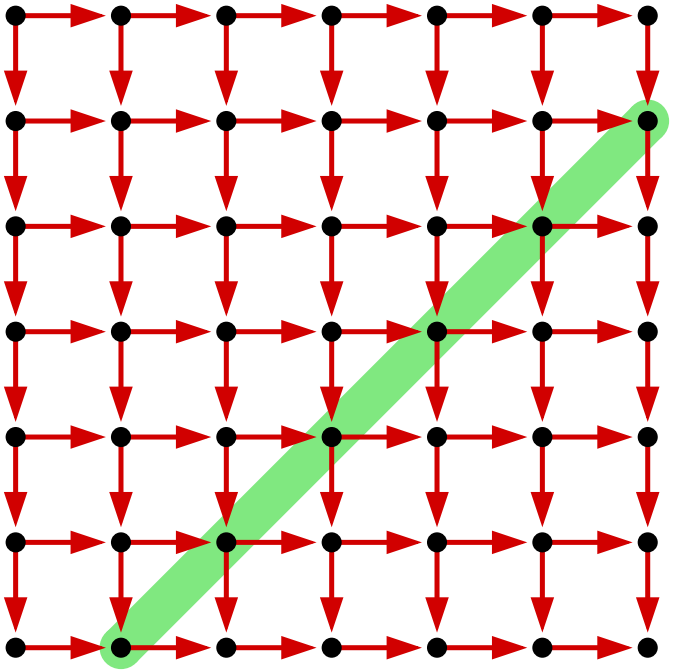
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



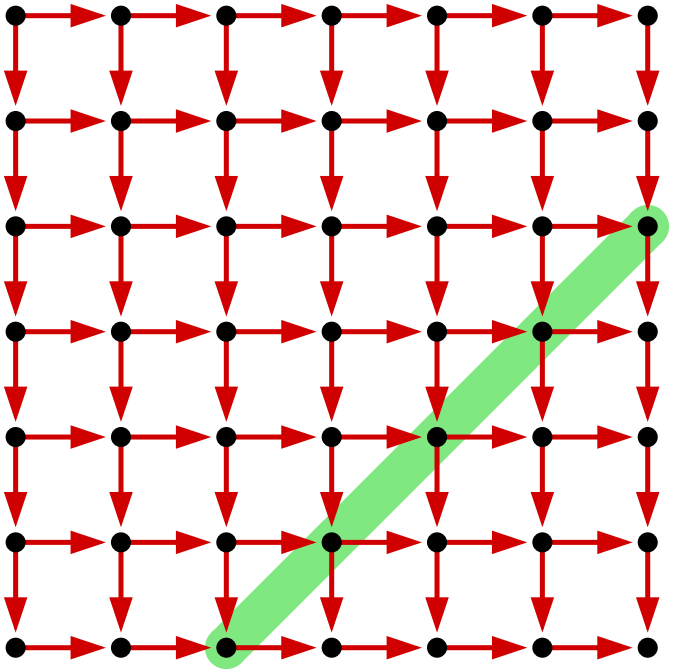
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



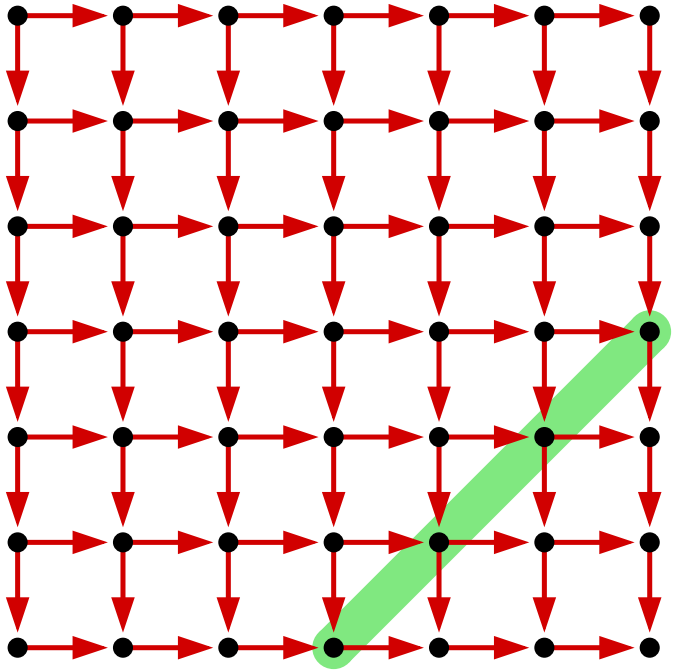
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



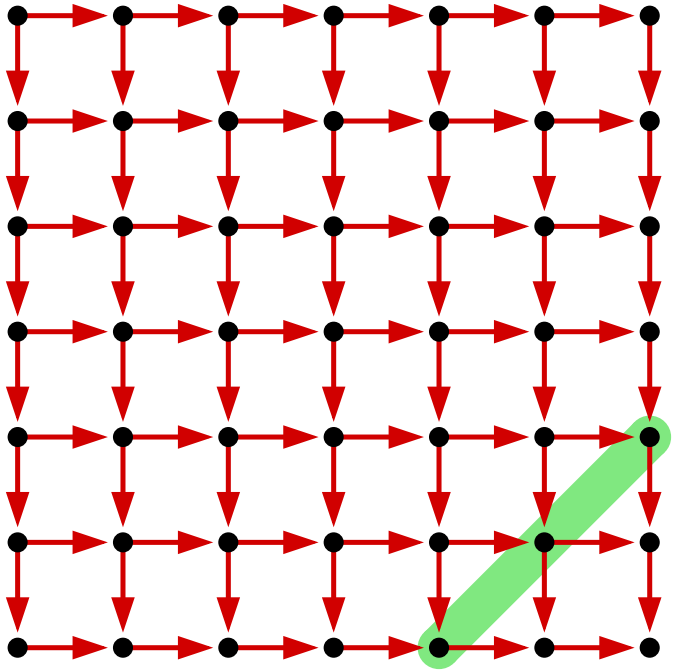
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



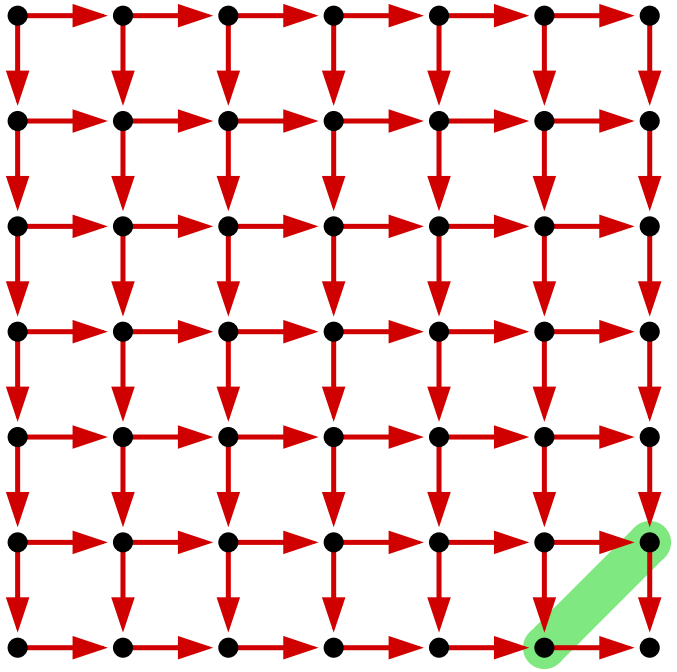
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



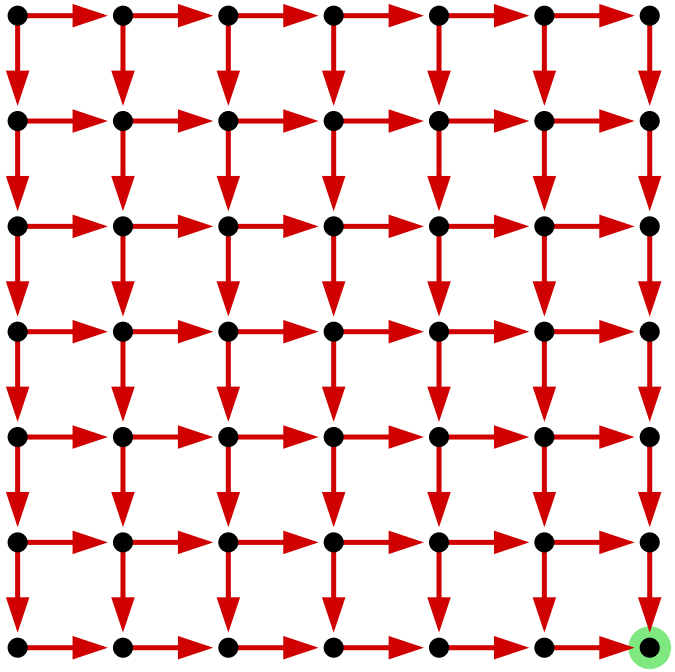
Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



Parallelisierung von Gauss-Seidel

- Umstrukturierung der i, j-Schleife so, daß Iterationsraum diagonal durchlaufen wird
  - innere Schleife dann ohne Abhängigkeiten
  - Problem: wechselnder Parallelitätsgrad



Umstrukturierung der Scheifen bei Gauss/Seidel

- Zeilenweiser Matrixdurchlauf:

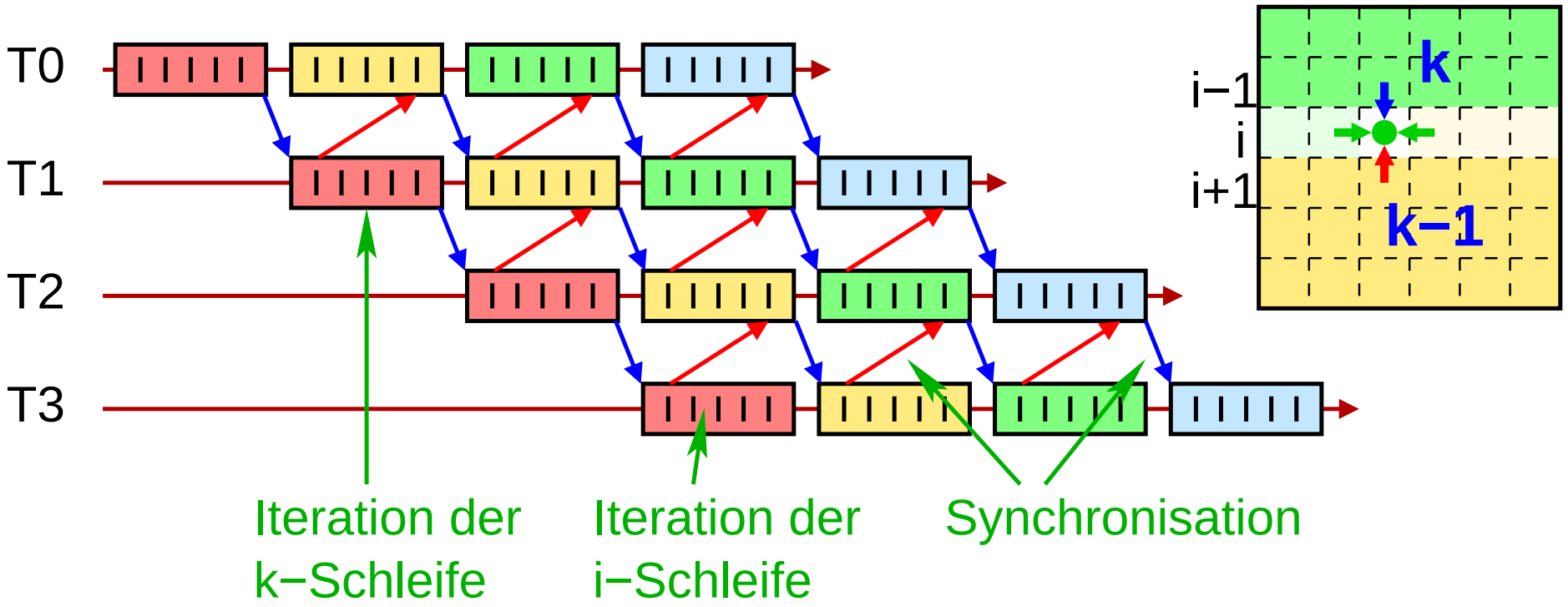
```
for (i=1; i<n-1; i++) {
    for (j=1; j<n-1; j++) {
        a[i][j] = ...;
```

- Diagonaler Matrixdurchlauf (☞ `02/diagonal.cpp`):

```
for (ij=1; ij<2*n-4; ij++) {
    int ja = (ij <= n-2) ? 1 : ij-(n-3);
    int je = (ij <= n-2) ? ij : n-2;
    for (j=ja; j<=je; j++) {
        i = ij-j+1;
        a[i][j] = ...;
```

Alternative Parallelisierung von Gauss/Seidel

- Voraussetzung: Zahl der Iterationen vorab bekannt
- Damit möglich: pipeline-artige Parallelisierung



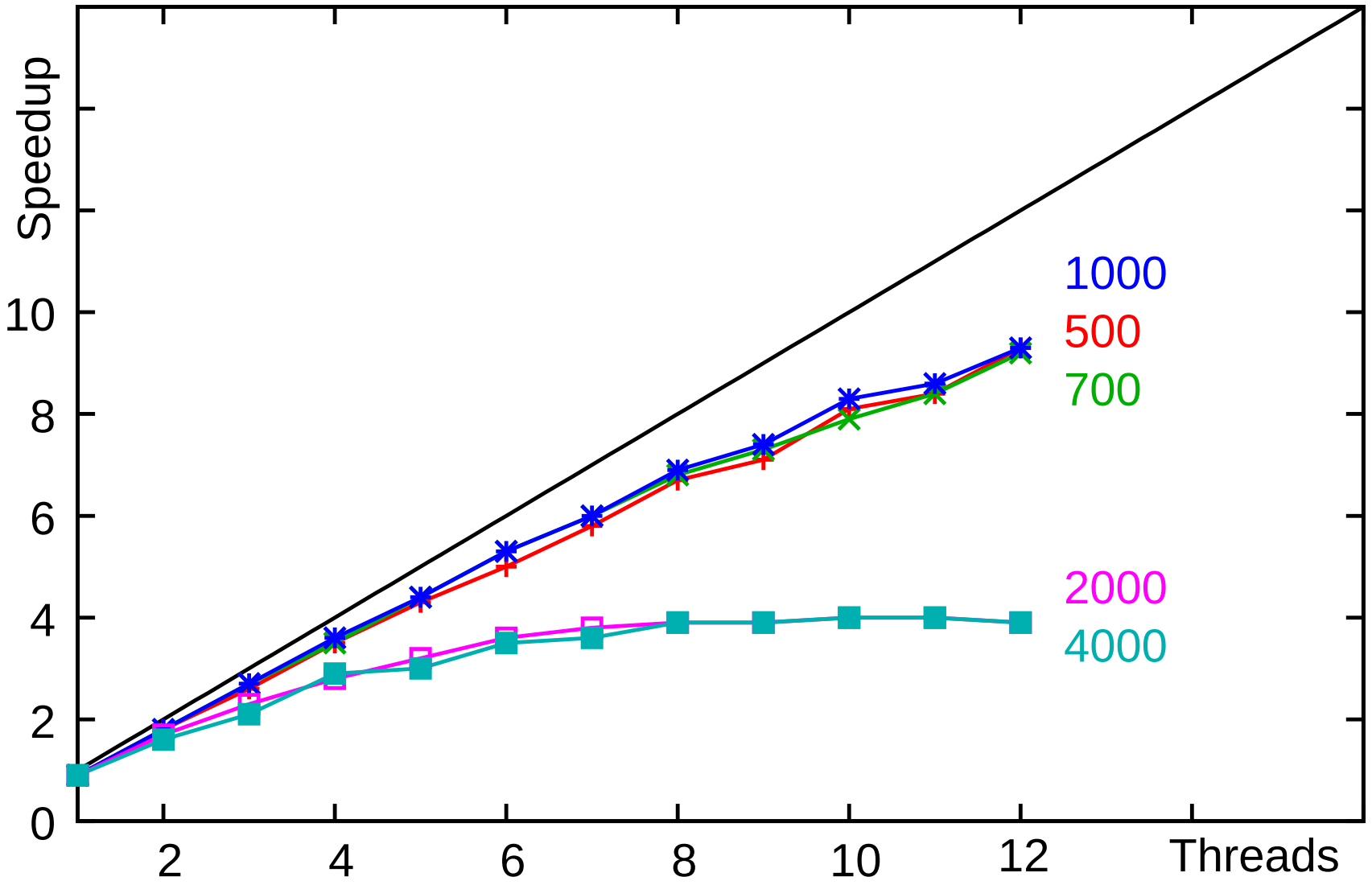
Ergebnisse

- Speedup mit `g++ -O` auf bslab10 im H-A 4111 (`eps`=0.001):

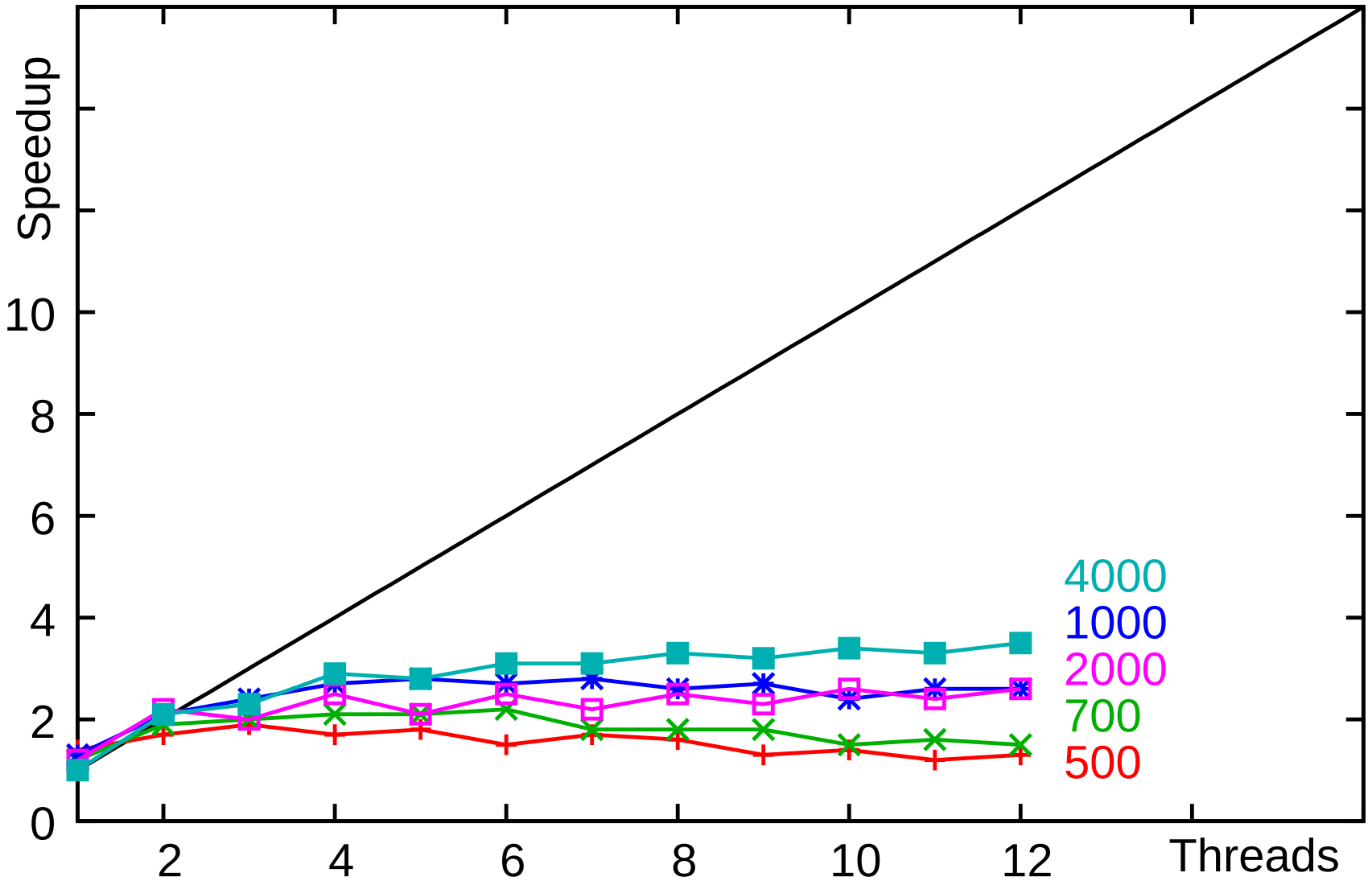
	Jacobi					Gauss/Seidel (diagonal)				
Thr.	500	700	1000	2000	4000	500	700	1000	2000	4000
1	0.9	0.9	0.9	0.9	0.9	1.8	2.0	1.6	1.6	1.3
2	1.8	1.5	1.4	1.4	1.4	3.5	3.7	2.1	2.6	2.6
3	2.6	2.0	1.6	1.6	1.6	4.0	4.4	2.5	2.7	3.1
4	3.3	2.3	1.7	1.6	1.6	4.1	4.8	3.0	3.0	3.5

- Leichter Leistungsverlust durch Übersetzung mit OpenMP
- Leistungsgewinn durch diagonalen Durchlauf bei Gauss/Seidel
- Hoher Speedup bei Gauss/Seidel mit Matrixgröße 700
  - Datengröße: $\sim$ 8MB, Cachegröße 4MB pro Dual-Core CPU

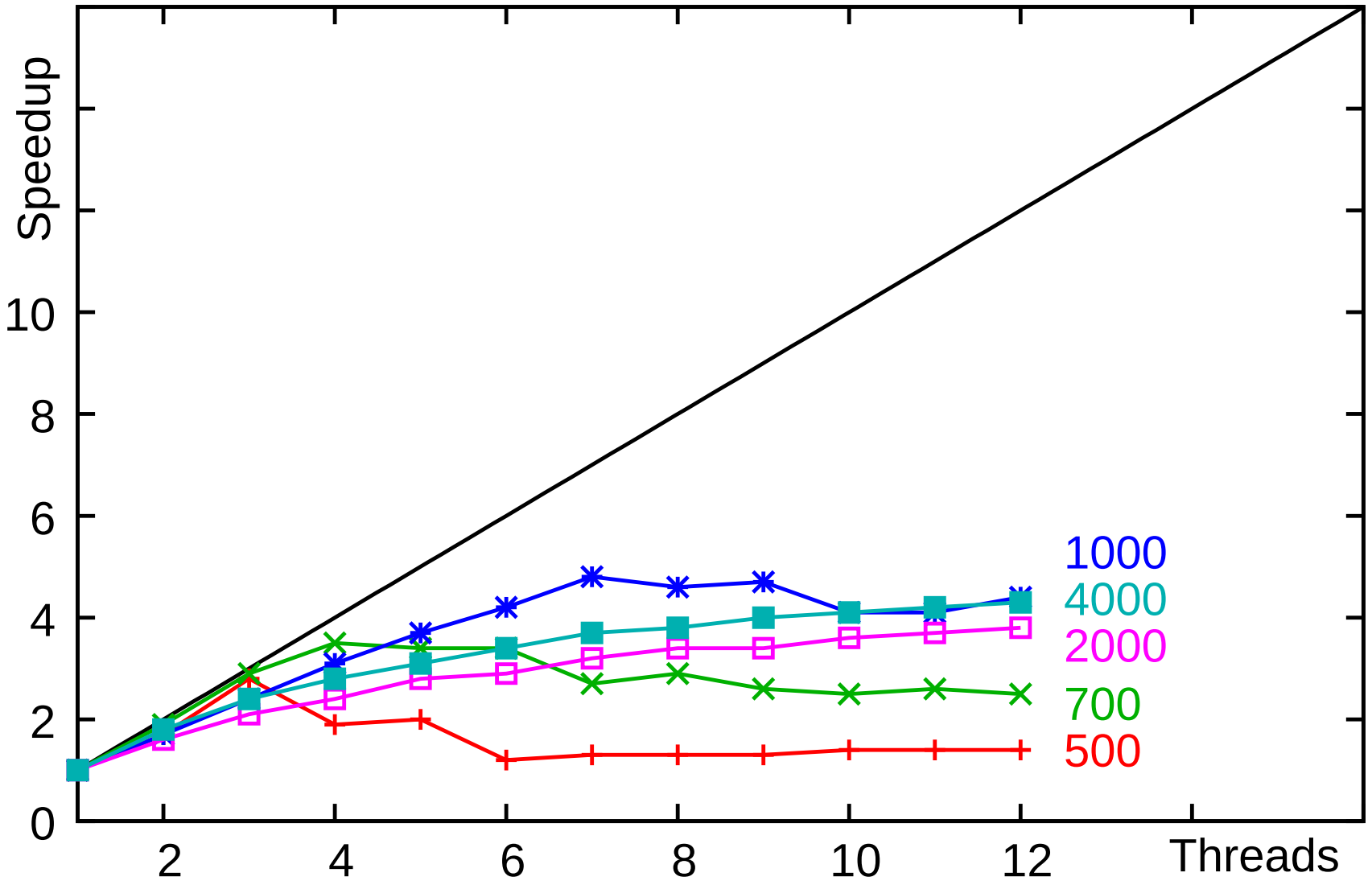
Speedup auf dem HorUS-Cluster: Jacobi



Speedup auf dem HorUS-Cluster: Gauss-Seidel (diagonal)



Speedup auf dem HorUS-Cluster: Gauss-Seidel (Pipeline)



2.6 OpenMP Synchronisation

- Mit OpenMP trägt der Programmierer die volle Verantwortung über die korrektre Synchronisation der Threads!
- Ein Beispiel als Motivation:

```
int i, j = 0;

#pragma omp parallel for private(i)
for (i=1; i<N; i++) {
    if (a[i] > a[j])
        j = i;
}
```

  - liefert dieses Programmstück mit der OpenMP-Direktive immer noch in `j` den Index des maximalen Elements von `a`?
  - die Speicherzugriffe der Threads können sich beliebig verzahnen $\Rightarrow$ nichtdeterministischer Fehler!

Synchronisation in OpenMP

- Höhere, einfach zu nutzende Konstrukte
- Realisierung durch Direktiven:
  - `barrier`: Barriere
  - `single` und `master`: Ausführung nur durch einen Thread
  - `critical`: Kritische Abschnitte
  - `atomic`: Atomare Operationen
  - `ordered`: Ausführung in Programmordnung
  - `taskwait` und `taskgroup`: Warten auf Tasks (☞ **2.7.2**)
  - `flush`: Speicher konsistent machen
    - Speicherbarriere (☞ **1.4.2**)
    - implizit bei anderen OpenMP Synchronisationen ausgeführt

Barriere

```
#pragma omp barrier
```

- Synchronisiert alle Threads
  - jeder Thread wartet, bis alle anderen die Barriere erreicht haben
- Implizite Barriere am Ende von `for`, `sections`, und `single` Direktiven
  - kann durch Angabe der Option `nowait` unterdrückt werden

Barriere: Beispiel (☞ 02/barrier.cpp)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 10000

float a[N][N];

main() {
  int i, j;

#pragma omp parallel
  {
    int thread = omp_get_thread_num();
    cout << "Thread " << thread << ": start loop 1\n";
#pragma omp for private(i,j)   // ggf. nowait
    for (i=0; i<N; i++) {
```

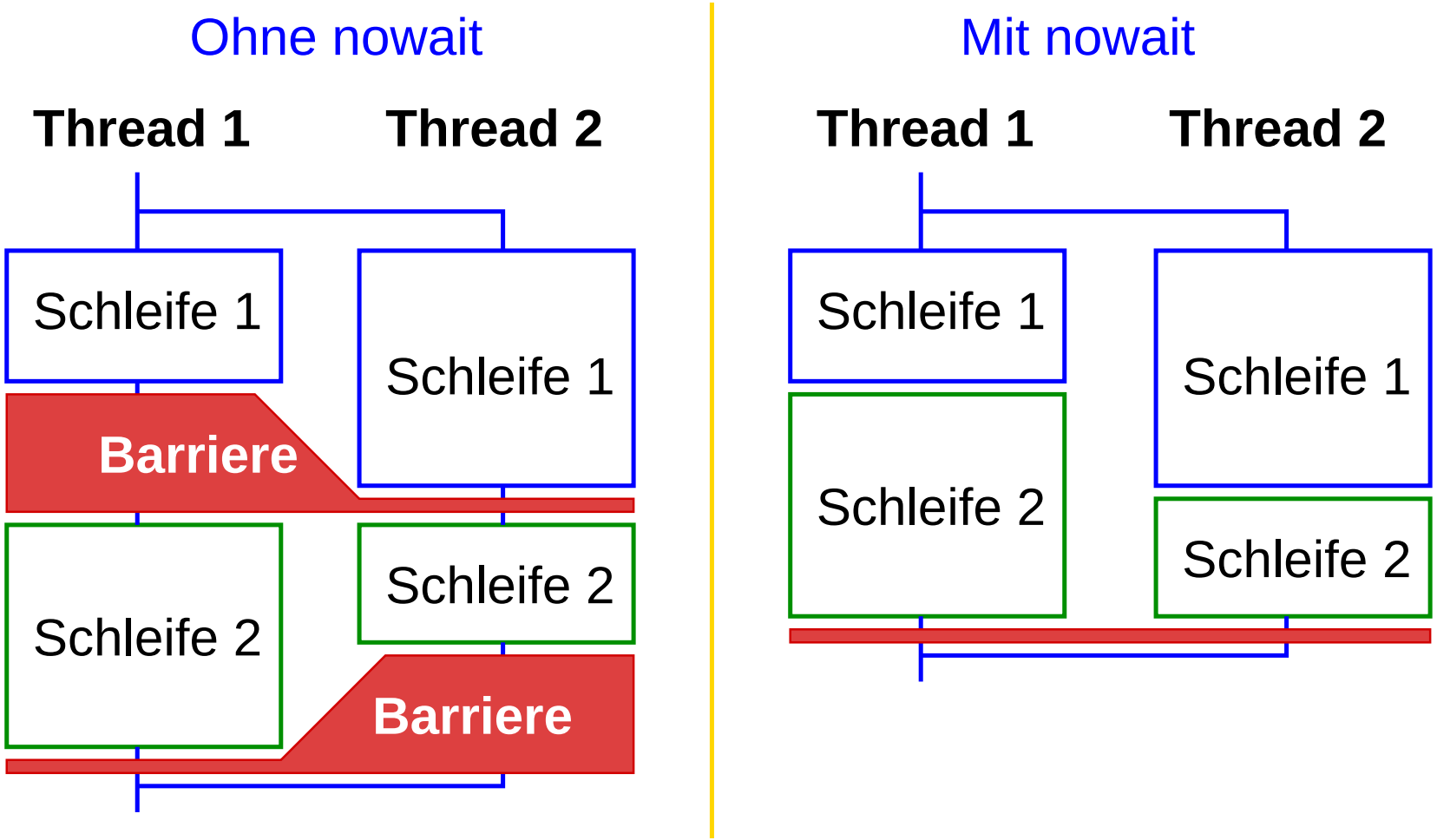
```
      for (j=0; j<i; j++) {
        a[i][j] = sqrt(i) * sin(j*j);
      }
    }

    cout << "Thread " << thread << ": start loop 2\n";
#pragma omp for private(i,j)
    for (i=0; i<N; i++) {
      for (j=i; j<N; j++) {
        a[i][j] = sqrt(i) * cos(j*j);
      }
    }
    cout << "Thread " << thread << ": end loop 2\n";
  }
}
```

Barriere: Beispiel ...

- Die erste Schleife bearbeitet das obere Dreieck der Matrix `a`, die zweite Schleife das untere Dreieck
  - Lastungleichgewicht zwischen den Threads
  - Barriere am Schleifenende führt zu Wartezeit
- Aber: die zweite Schleife hängt nicht von der ersten ab
  - d.h. Berechnung kann begonnen werden, bevor die erste Schleife vollständig abgearbeitet ist
  - Barriere bei der ersten Schleife kann entfernt werden
    - Option `nowait`
  - Laufzeit mit 2 Threads nur noch 4.8 s statt 7.2 s

Barriere: Beispiel ...

- Abläufe des Programms:



Ausführung durch einen einzigen Thread

```
#pragma omp single
    Anweisung / Block
```

```
#pragma omp master
    Anweisung / Block
```

- Block wird nur durch einen einzigen Thread ausgeführt
- Keine Synchronisation zu Beginn der Direktive
- Bei `single`:
  - Ausführung durch ersten ankommenden Thread
  - Barrieren-Synchronisation am Ende (außer: `nowait`)
- Bei `master`:
  - Ausführung des Blocks durch den *Master*-Thread
  - keine Synchronisation am Ende

Kritische Abschnitte

```
#pragma omp critical[(<name>)]
    Anweisung / Block
```

- Anweisung / Block wird unter wechselseitigem Ausschluß ausgeführt
- Zur Unterscheidung verschiedener kritischer Abschnitte kann ein Name vergeben weden

Atomare Operationen

```
#pragma omp atomic [read|write|update|capture][seq_cst]
    Anweisung / Block
```

- Anweisung bzw. Block (nur bei `capture`) wird atomar ausgeführt
  - i.d.R. durch Übersetzung in spezielle Maschinenbefehle
- Deutlich effizienter als kritischer Abschnitt
- Option definiert die Art der atomaren Operation:
  - `read` / `write`: atomares Lesen / Schreiben
  - `update` (Default): atomares Verändern einer Variablen
  - `capture`: atomares Verändern einer Variablen mit Speichern des alten bzw. neuen Wertes
- Option `seq_cst`: Herstellung der Speicherkonsistenz (`flush`)

Atomare Operationen: Beispiele

- Atomares Aufaddieren:

```
#pragma omp atomic update
x += a[i] * a[j];
```

  - die rechte Seite wird dabei **nicht** atomar ausgewertet!

- Atomares *fetch-and-add*:

```
#pragma omp atomic capture
{ old = counter; counter += size; }
```

- Statt + sind auch alle anderen binären Operatoren möglich

- Ein atomares *compare-and-swap* lässt sich mit OpenMP (derzeit noch) nicht realisieren

  - ggf. *Builtin*-Funktionen des Compilers verwenden

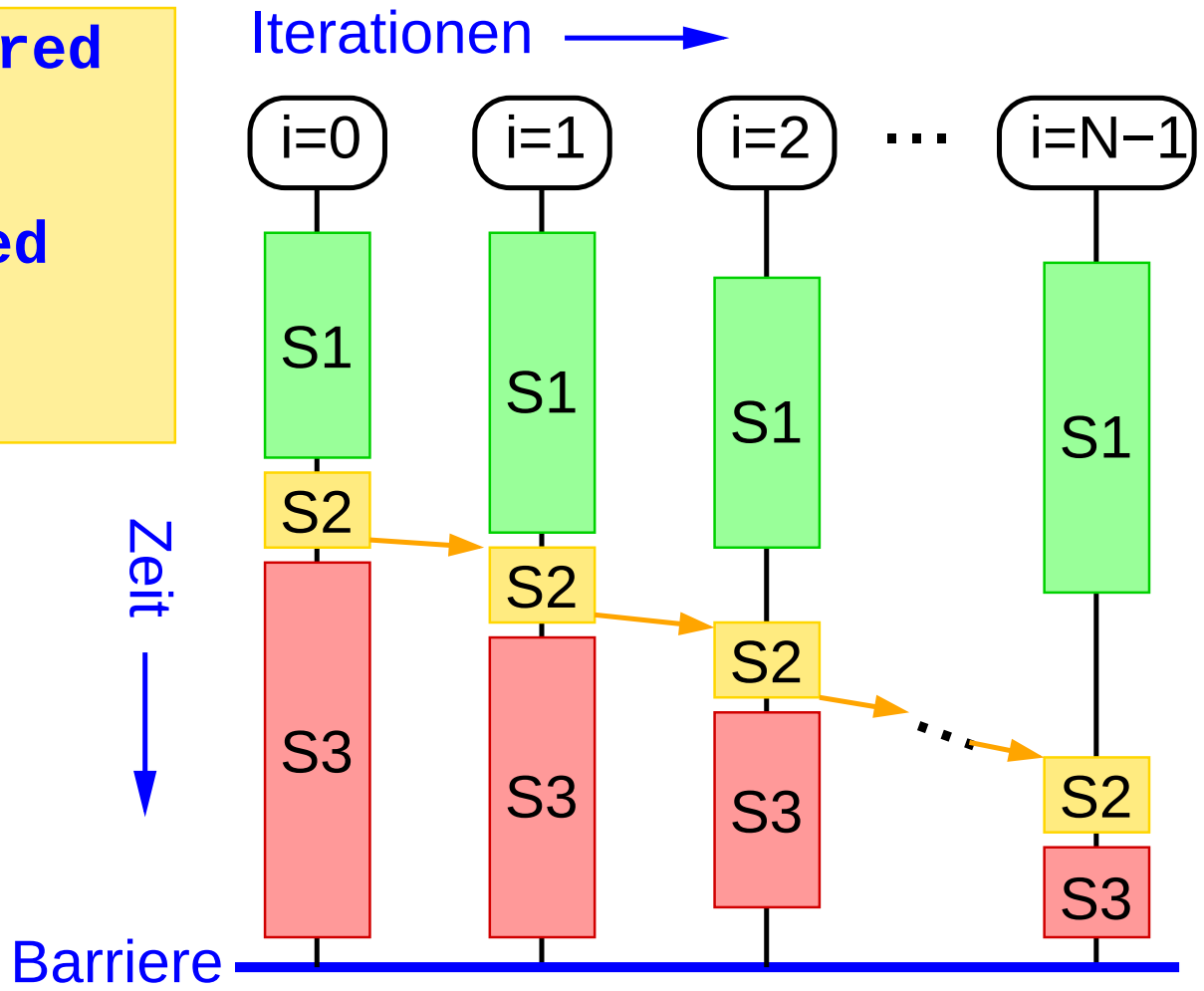
Ausführung in Programmordnung

```
#pragma omp for ordered
for(...) {
   ...
   #pragma omp ordered
   Anweisung / Block
   ...
}
```

- `ordered`-Direktive nur zulässig im dynamischen Bereich einer `for`-Direktive mit `ordered`-Option
  - sinnvoll: Option `schedule(static,1)`
- Anweisung / Block wird von den Threads in genau derselben Reihenfolge ausgeführt wie im sequentiellen Programm

Ablauf mit `ordered`

```
#pragma omp for ordered
for(i=0; i<N; i++) {
   S1;
   #pragma omp ordered
      S2;
   S3;
}
```



Reduktions-Operationen

- Oft werden in Schleifen Werte aggregiert, z.B.:

```
int a[N];
int i, sum = 0;
#pragma omp parallel for reduction(+: sum)
for (i=0; i<N; i++){
   sum += a[i];
}
printf("sum=%d\n",sum);
```

Am Ende der Schleife enthält sum die Summe aller Elemente

- `reduction` erspart kritischen Abschnitt
  - jeder Thread berechnet zunächst seine Teilsumme in einer privaten Variable
  - nach Ende der Schleife Berechnung der Gesamtsumme
- Statt `+` auch möglich: `-` `*` `&` `|` `^` `&&` `||` `min` `max`
  - zusätzlich auch selbstdefinierte Operationen möglich

Beispiel: Reduktion ohne `reduction`-Option

```
int a[N];
int i, sum = 0;
int lsum = 0;  // lokale Teilsumme

#pragma omp parallel firstprivate(lsum) private(i)
{
#   pragma omp for nowait
    for (i=0; i<N; i++) {
        lsum += a[i];
    }
#   pragma omp atomic
    sum += lsum;
}
printf("sum=%d\n",sum);
```

lsum wird mit 0 initialisiert

keine Barriere am Ende der Schleife

lokale Teilsummen auf globale Summe addieren

Beispiel: Reduktion ohne `reduction`-Option

```
int a[N];
int i, sum = 0;
#pragma omp parallel private(i)
{
   int lsum = 0; // lokale Teilsumme
#  pragma omp for nowait
   for (i=0; i<N; i++) {
      lsum += a[i];
   }
#  pragma omp atomic
   sum += lsum;
}
printf("sum=%d\n",sum);
```

keine Barriere am Ende der Schleife

lokale Teilsummen auf globale Summe addieren

2.7.1 Die `sections`-Direktive: parallele Code-Abschnitte

```
#pragma omp sections [<clause_list>]
{
  #pragma omp section
  Anweisung / Block
  #pragma omp section
  Anweisung / Block
  ...
}
```

- Jede Sektion wird genau einmal von einem (beliebigen) Thread ausgeführt
- Am Ende der `sections`-Direktive erfolgt eine Barrieren-Synchronisation
  - außer Option `nowait` wird angegeben

Beispiel: Unabhängige Code-Teile

```
double a[N], b[N];
int i;
#pragma omp parallel sections private(i)
{
    #pragma omp section
    for (i=0; i<N; i++)
        a[i] = 100;
    #pragma omp section
    for (i=0; i<N; i++)
        b[i] = 200;
}
```

Wichtig!!

- Die beiden Schleifen können nebenläufig zueinander ausgeführt werden
- Funktionsaufteilung

Beispiel: Einfluß von nowait (☞ 02/sections.cpp)

```
main() {
  int p;
#pragma omp parallel private(p)
  {
    int thread = omp_get_thread_num();
#pragma omp sections // ggf. nowait
    {
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 1 start\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 1 end\n";
        p = 1;
      }
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 2 start\n";
```

2.7.1 Die `sections`-Direktive ...

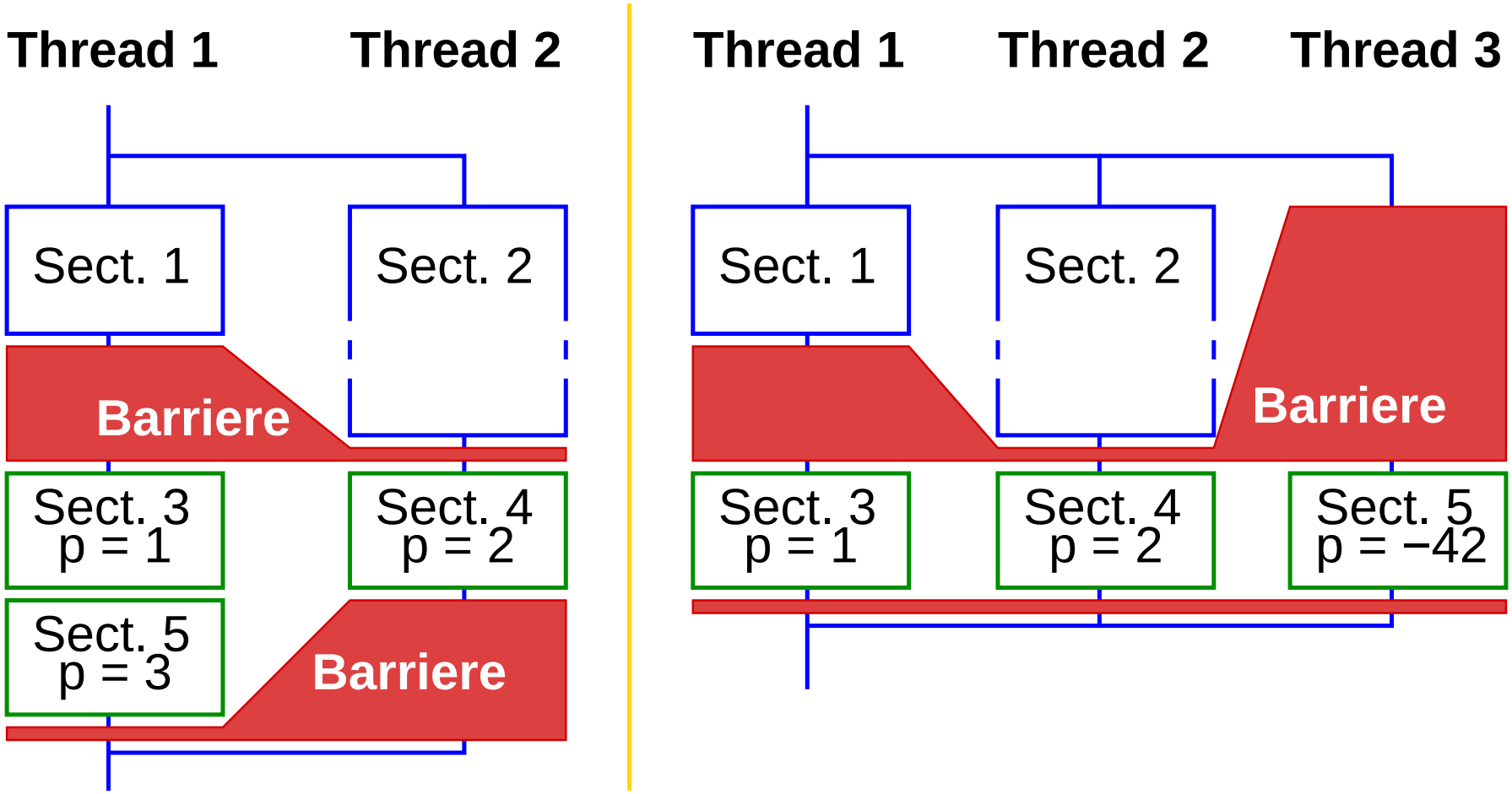
```
        usleep(1000000);
        cout << "Thread " << thread << ", Section 2 end\n";
        p = 2;
      }
    } // Ende omp sections
#pragma omp sections
    {
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 3 start, p = "
             << p << "\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 3 end\n";
        p = 3;
      }
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 4 start, p = "
```

2.7.1 Die `sections`-Direktive ...

```
                << p << "\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 4 end\n";
        p = 4;
      }
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 5 start, p = "
             << p << "\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 5 end\n";
        p = 5;
      }
    } // Ende omp sections
  } // Ende omp parallel
}
```

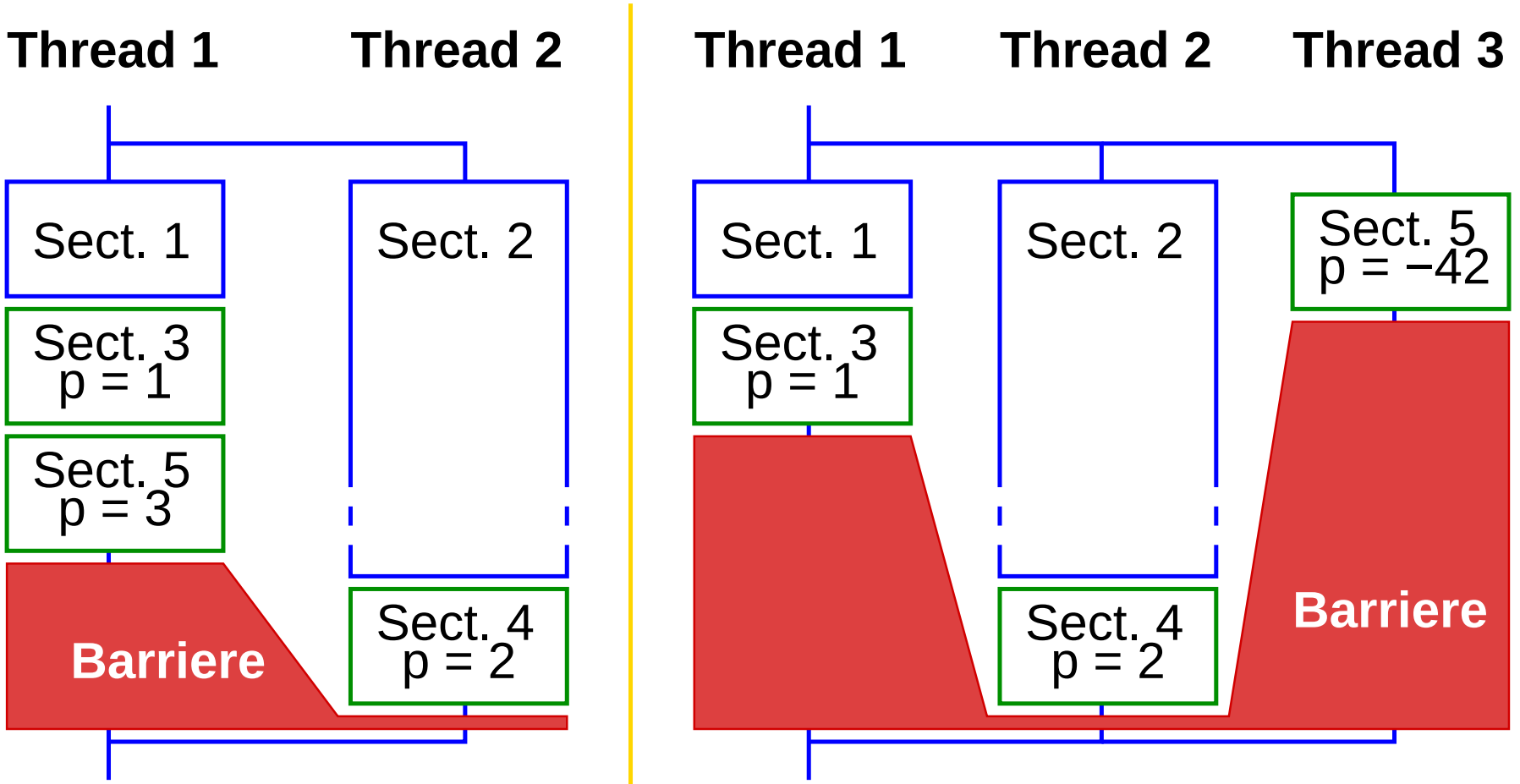
Beispiel: Einfluß von `nowait` ...

- Abläufe des Programms **ohne** `nowait` Option:



Beispiel: Einfluß von `nowait` ...

- Abläufe des Programms **mit** `nowait` Option:



2.7.2 Die `task`-Direktive: explizite Tasks

```
#pragma omp task[<clause_list>]
Anweisung/Block
```

- Erzeugt aus der Anweisung / dem Block einen expliziten Task
- Tasks werden durch die verfügbaren Threads abgearbeitet (*Work-Pool* Modell)
- Optionen `private`, `firstprivate`, `shared` regeln, welche Variablen zur Datenumgebung der Task gehören
- Option `if` erlaubt Festlegung, wann expliziter Task erzeugt werden soll

Beispiel: paralleler Quicksort (☞ 02/qsort.cpp)

```
void quicksort(int *a, int lo, int hi) {
  ...
  // Variablen sind per Default firstprivate
  #pragma omp task if (j-lo > 10000)
  quicksort(a, lo, j);
  quicksort(a, i, hi);
}

int main() {
  ...
  #pragma omp parallel
  #pragma omp single nowait    // Ausführung durch einen Thread
  quicksort(array, 0, n-1);
  // Vor Ende des parallelen Bereichs wird auf Beendigung aller Tasks gewartet
```

Task-Synchronisation

```
#pragma omp taskwait
```

```
#pragma omp taskgroup
{
    Block
}
```

- `taskwait`: wartet auf die Beendigung aller direkten Subtasks des aktuellen Tasks
- `taskgroup`: am Ende des Blocks wird auf alle Subtasks (direkte und indirekte) gewartet, die der aktuelle Task innerhalb des Blocks erzeugt hat
  - verfügbar ab OpenMP 4.0
  - Achtung: ältere Compiler ignorieren diese Direktive!

Beispiel: paralleler Quicksort (☞ `02/qsort.cpp`)

- Änderung beim Aufruf:

```
#pragma omp parallel
#pragma omp single nowait     // Ausführung durch einen Thread
{
   quicksort(array, 0, n-1);
   checkSorted(array, n);
}
```

- Problem:
  - `quicksort()` startet neue Tasks
  - Tasks sind noch nicht beendet, wenn `quicksort()` zurückkehrt

Beispiel: paralleler Quicksort ...

- Lösung 1:

```
void quicksort(int *a, int lo, int hi) {
  ...
  #pragma omp task if (j-lo > 10000)
  quicksort(a, lo, j);
  quicksort(a, i, hi);
  #pragma omp taskwait   ← warte auf erzeugen Task
}
```

  - Vorteil: Subtask wird beendet, bevor `quicksort()` zurückkehrt
    - notwendig, wenn nach dem rekursiven Aufruf noch Berechnungen folgen
  - Nachteil: relativ hoher Overhead

Beispiel: paralleler Quicksort ...

- Lösung 2:

```
#pragma omp parallel
#pragma omp single nowait     // Ausführung durch einen Thread
{
#pragma omp taskgroup
    {
        quicksort(array, 0, n-1);
    }                    ← warte auf alle im Block erzeugten Tasks
    checkSorted(array, n);
}
```

  - Vorteil: Warten nur an einer einzigen Stelle
  - Nachteil: Semantik von `quicksort()` muß gut dokumentiert werden

Parallelverarbeitung

WS 2015/16

14.12.2015

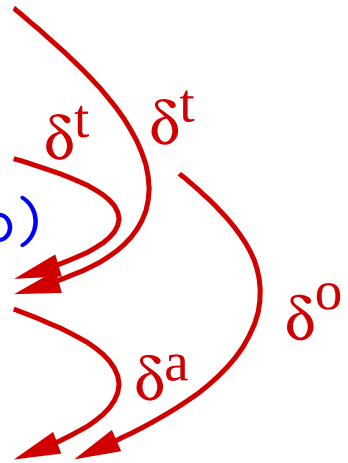
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Abhängigkeiten zwischen Tasks (☞ `02/tasks.cpp`)

- Option `depend` erlaubt Angabe von Abhängigkeiten zw. Tasks
  - angegeben werden die betroffenen Variablen (bzw. auch Array-Abschnitte) und die Richtung des Datenflusses
- Beispiel:

```
#pragma omp task shared(a) depend(out: a)
  a = computeA();
#pragma omp task shared(b) depend(out: b)
  b = computeB();
#pragma omp task shared(a,b,c) depend(in: a,b)
  c = computeCfromAandB(a, b);
#pragma omp task shared(b) depend(out: b)
  b = computeBagain();
```



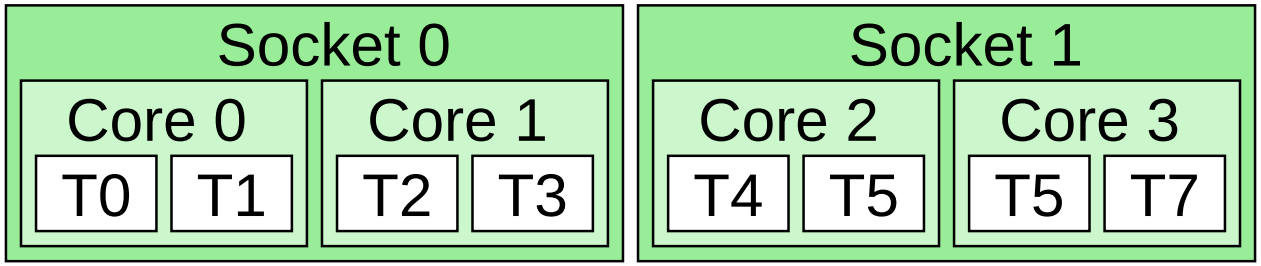
  - die Variablen `a`, `b`, `c` müssen hier `shared` sein, da sie das Ergebnis der Berechnung eines Taks enthalten

2.8.1 Thread-Affinität

- Ziel: Steuerung, wo Threads ausgeführt werden
  - d.h. durch welchen HW-Threads auf welchem Core auf welcher CPU
- Wichtig u.a. wegen Architektur heutiger Multicore-CPUs
  - HW-Threads teilen sich die Rechenwerke eines Cores
  - Cores teilen sich L2-Caches und Speicherzugang
- Konzept von OpenMP:
  - Einführung von Plätzen (***places***)
    - Platz: Menge von Hardware-Ausführungsumgebungen
    - z.B. Hardware-Thread, Core, Prozessor (Socket)
  - Optionen, um Verteilung von Threads auf Plätze zu steuern

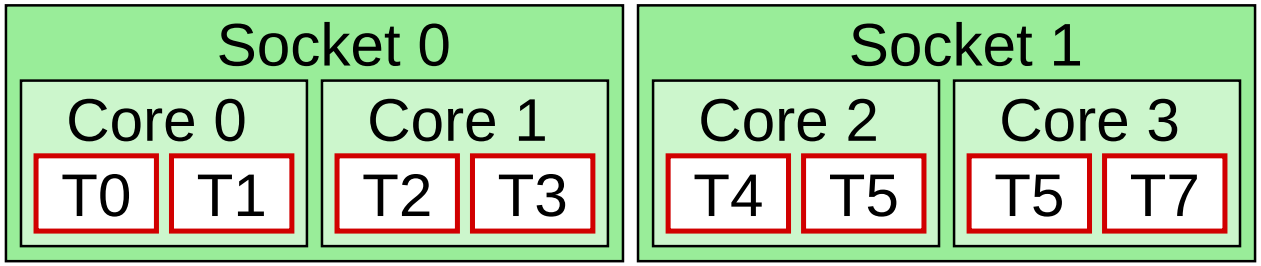
Umgebungsvariable `OMP_PLACES`

- Legt die Plätze eines Rechners fest
- Z.B. Knoten mit 2 Dualcore-CPUs mit je 2 HW-Threads:



Umgebungsvariable OMP_PLACES

- Legt die Plätze eines Rechners fest
- Z.B. Knoten mit 2 Dualcore-CPUs mit je 2 HW-Threads:

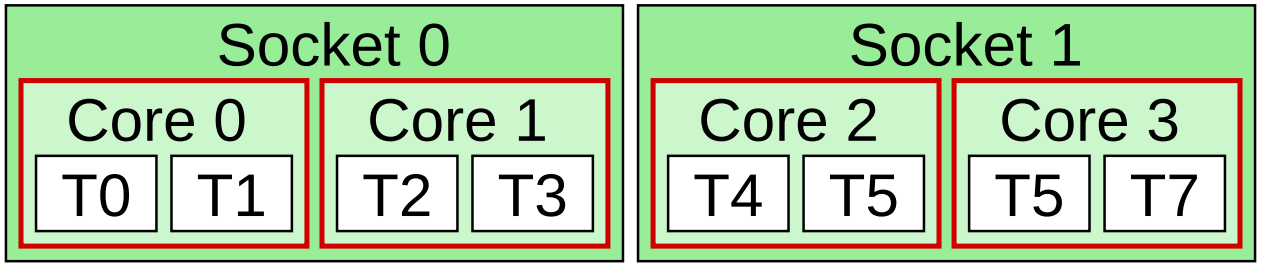


- Um jeden Hardware-Thread als Platz zu betrachten z.B.:

```
OMP_PLACES = "threads"
OMP_PLACES = "{0},{1},{2},{3},{4},{5},{6},{7}"
OMP_PLACES = "{0}:8:1"
```

Umgebungsvariable `OMP_PLACES`

- Legt die Plätze eines Rechners fest
- Z.B. Knoten mit 2 Dualcore-CPUs mit je 2 HW-Threads:

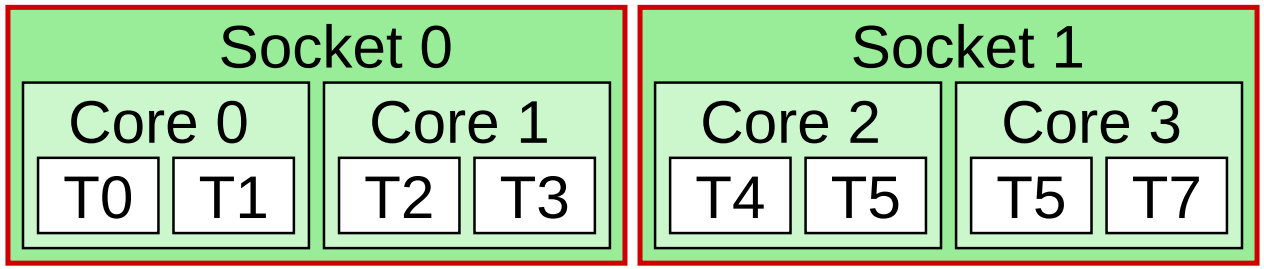


- Um jeden Core als Platz zu betrachten z.B.:

```
OMP_PLACES = "cores"
OMP_PLACES = "{0,1},{2,3},{4,5},{6,7}"
OMP_PLACES = "{0,1}:4:2"
```

Umgebungsvariable `OMP_PLACES`

- Legt die Plätze eines Rechners fest
- Z.B. Knoten mit 2 Dualcore-CPUs mit je 2 HW-Threads:



- Um jeden Socket als Platz zu betrachten z.B.:

```
OMP_PLACES = "sockets"
OMP_PLACES = "sockets(2)"
OMP_PLACES = "{0:4},{4:4}"
```

Zuordnung von Threads zu Plätzen

- Option `proc_bind( spread | close | master )` der `parallel`-Direktive
  - `spread`: Threads werden gleichmäßig auf die verfügbaren Plätze verteilt, Platz-Liste wird partitioniert
    - vermeidet Resourcen-Konflikte zwischen Threads
  - `close`: Threads werden möglichst nahe beim Master-Thread allokiert
    - um z.B. gemeinsamen Cache optimal zu nutzen
  - `master`: Threads werden demselben Platz zugewiesen wie Master-Thread
    - engstmögliche Lokalität zum Master-Thread
- In der Regel kombiniert mit verschachtelten parallelen Regionen

Parallelverarbeitung

WS 2015/16

04.01.2016

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

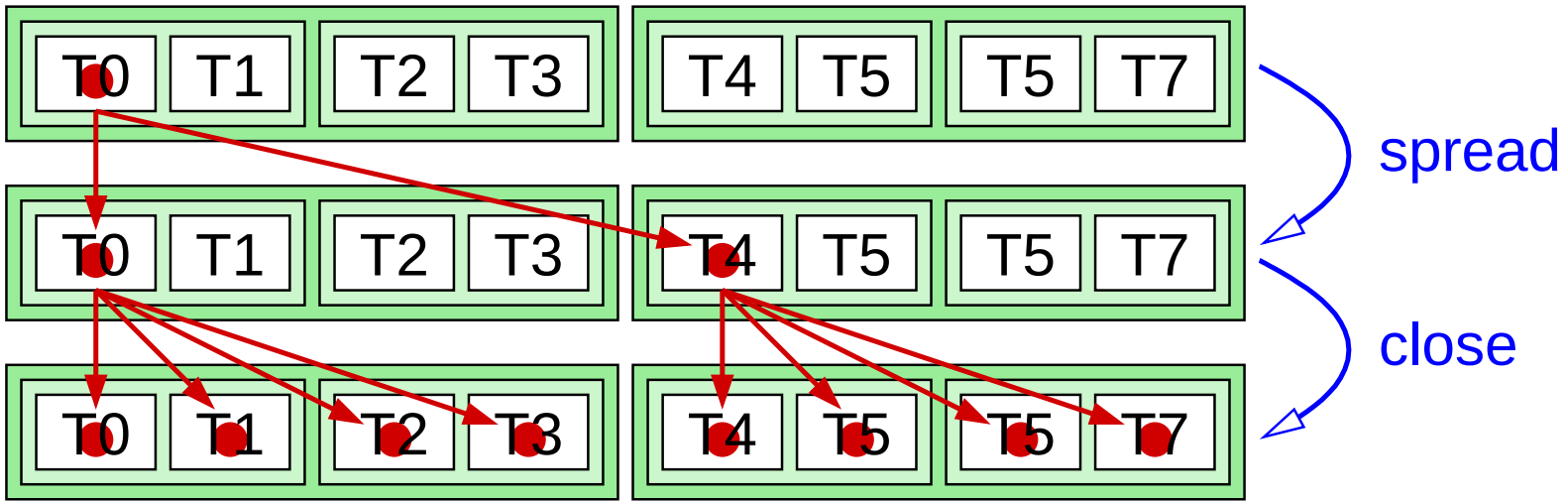
Stand: 1. Februar 2016

Beispiel: verschachtelte parallele Regionen

```
double f1(double x)
{
  #pragma omp parallel for proc_bind(close)
  for (i=0; i<N; i++) {
     ...
}
...
  #pragma omp parallel proc_bind(spread)
  #pragma omp single
  {
    #pragma omp task shared(a)
    a = f1(x);
    #pragma omp task shared(b)
    b = f1(y);
  }
  ...
```

Beispiel: verschachtelte parallele Regionen ...

- Erlaube verschachtelte Parallelität: `export OMP_NESTED=true`
- Definition der Tread-Anzahl pro Schachtelungstiefe:
  - `export OMP_NUM_THREADS=2,4`
- Definiere die Plätze: `export OMP_PLACES=cores`
- Erlaube das Binden von Threads an Plätze:
  - `export OMP_PROC_BIND=true`
- Ergebnis:



2.8.2 SIMD-Vektorisierung

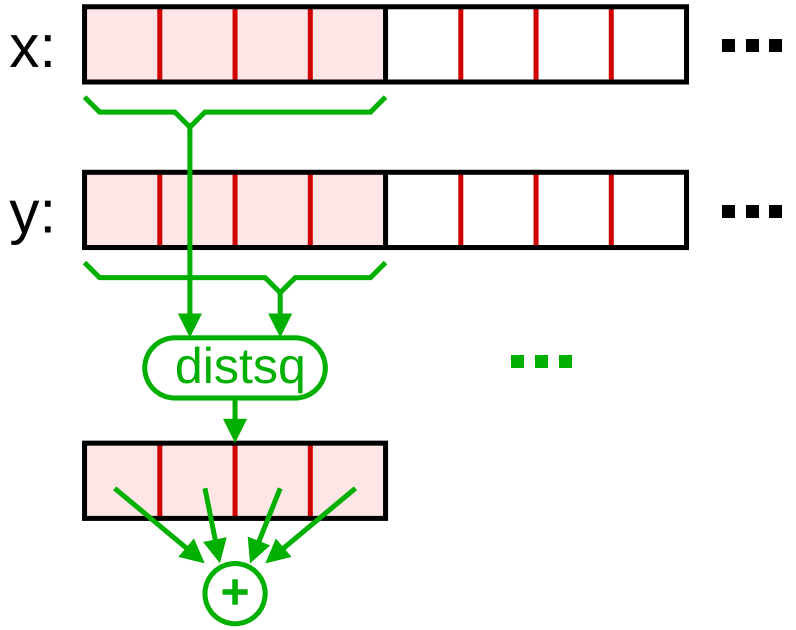
```
#pragma omp simd [<clause_list>]
for(...) ...
```

- Umformung einer Schleife zur Nutzung der SIMD Vektorregister
  - z.B. Intel SSE: 4 `float`-Operationen gleichzeitig
- Schleife wird mit einem einzigen Thread ausgeführt
  - Kombination mit `for` ist möglich
- Mögliche Optionen u.a.: `private`, `lastprivate`, `reduction`
- Option `safelen`: maximale Vektorlänge
  - d.h. Abstand (in Iterationen) von Datenabhängigkeiten, z.B.:

```
for (i=0; i<N; i++)
    a[i] = b[i] + a[i-4];   // safelen = 4
```

Beispiel

```
#pragma omp declare simd
float distsq(float x, float y) {
  return (x - y) * (x - y);
}
...
  #pragma omp simd reduction(+:s)
  for (i=0; i<N; i++) {
    s += distsq(x[i], y[i]);
  }
```



- Die Direktive `declare simd` erzeugt eine Version einer Funktion mit Vektorregistern als Argumenten bzw. Ergebnis
- Bei größerem `N` auch sinnvoll:

```
#pragma omp parallel for simd reduction(+:s)
```

2.8.3 Nutzung externer Beschleuniger

- Modell in OpenMP 4.0:
  - ein Host (Multiprozessor mit gemeinsamem Speicher) mit mehreren gleichen Beschleunigern (**Targets**)
  - Ausführung eines Code-Blocks kann per Direktive auf ein Target übertragen werden
  - Host und Target können sich den Speicher teilen, müssen aber nicht
    - Datentransport muss über Direktiven durchgeführt werden
- Zur Unterstützung des Ausführungsmodells von Graphikkarten
  - Einführung von Thread-Teams
  - Threads eines Teams werden durch einen Streaming-Multiprozessor ausgeführt (in SIMD-Manier!)

Die `target`-Direktive

```
#pragma omp target [data] [<clause_list>]
```

- Überträgt Ausführung und Daten auf ein Target
  - Daten werden zwischen CPU-Speicher und Target-Speicher kopiert
    - Abbildung und Transferrichtung durch `map`-Option festgelegt
  - nachfolgender Code-Block wird auf Target ausgeführt
    - ausser wenn mit `target data` nur eine Datenumgebung erzeugt wird
- Host wartet, bis Berechnung auf Target beendet ist
  - die `target`-Direktive ist aber auch innerhalb einer asynchronen Task möglich

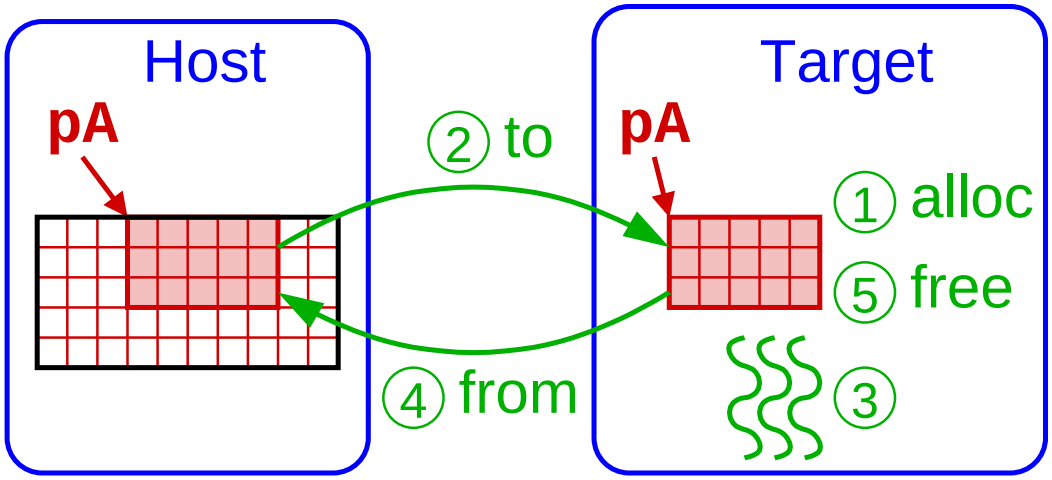
Die map-Option

- Bildet Variable v_H in der Host-Datenumgebung auf entsprechende Variable v_T in der Target-Datenumgebung ab
  - entweder werden Daten zwischen v_H und v_T kopiert oder v_H und v_T sind identisch
- Syntax: `map( alloc | to | from | tofrom : <list> )`
  - `<list>`: Liste der Original-Variablen
    - auch Array-Abschnitte erlaubt, ☞ **2.7.2**
  - `alloc`: allokiere nur Speicher für v_T
  - `to`: allokiere v_T, kopiere v_H nach v_T zu Beginn
  - `from`: allokiere v_T, kopiere v_T nach v_H am Ende
  - `tofrom`: Default-Wert, `to` und `from`

Target-Datenumgebung

- Ablauf beim `target`-Konstrukt:

```
#pragma omp target \
        map(tofrom: pA)
{①②
   ...③
}④⑤
```



- `target data` Umgebung zur Optimierung von Speichertransfers
  - mehrere Code-Blöcke können auf Target ausgeführt werden, ohne die Daten neu zu übertragen
  - Direktive `target update` erlaubt bei Bedarf Datentransfers innerhalb der `target data` Umgebung

Beispiel

```
#pragma omp target data map(alloc:tmp[:N]) \
        map(to:in[:N]) map(from:res)
{
  #pragma omp target
  #pragma omp parallel for
  for (i=0; i<N; i++)
    tmp[i] = compute_1(in[i]);

  modify_input_array(in);

  #pragma omp target update to(in[:N])
  #pragma omp target
  #pragma omp parallel for reduction(+:res)
  for (i=0; i<N; i++)
    res += compute_2(in[i], tmp[i])
}
```

Host

Target

Host

Target

Host

Thread Teams

- Erlauben zweistufige Parallelisierung, z.B. auf GPUs
- Erzeugung einer Menge von Thread Teams:

```
#pragma omp teams [<clause_list>]
Anweisung/Block
```

  - Anweisung/Block wird durch die Master-Threads der Teams ausgeführt
  - Teams können sich nicht synchronisieren
- Aufteilung einer Schleife auf die Master-Threads der Teams:

```
#pragma omp distribute [<clause_list>]
for(...) ...
```

- Parallelisierung innerhalb eines Teams z.B. mit `paralel for`

Beispiel: SAXPY

- Auf dem Host:

```
void saxpy(float *y, float *x, float a, int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
}
```

- Auf der GPU (naiv):

```
void saxpy(float *y, float *x, float a, int n) {
  #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
  {
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
      y[i] = a*x[i] + y[i];
  }
}
```

Beispiel: SAXPY ...

- Auf der GPU (optimiert): jedes Team bearbeitet einen Block

```
void saxpy(float *y, float *x, float a, int n) {
  int nBlk = numBlocks(n);   // Zahl der Blöcke
  int nThr = numThreads(n); // Zahl der Threads
  #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
  #pragma omp teams num_teams(nBlk) thread_limit(nThr)
  {
    #pragma omp distribute
    for (int i = 0; i < n; i += n/nBlk) {
      #pragma omp parallel for
      for (int j = i; j < i + n/nBlk; j++)
        y[j] = a*x[j] + y[j];
      }
    }
  }
}
```

Beispiel: SAXPY ...

- Auf der GPU (optimiert, kürzer):

```
void saxpy(float *y, float *x, float a, int n) {
  int nBlk = numBlocks(n);   // Zahl der Blöcke
  int nThr = numThreads(n); // Zahl der Threads
  #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
  #pragma omp teams distribute parallel for \
          num_teams(nBlk) thread_limit(nThr)
  for (int i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
}
```

- Iterationen werden zunächst in Blöcken auf die Streaming-Multiprozessoren verteilt, dort nochmal auf die einzelnen Threads

Parallelverarbeitung

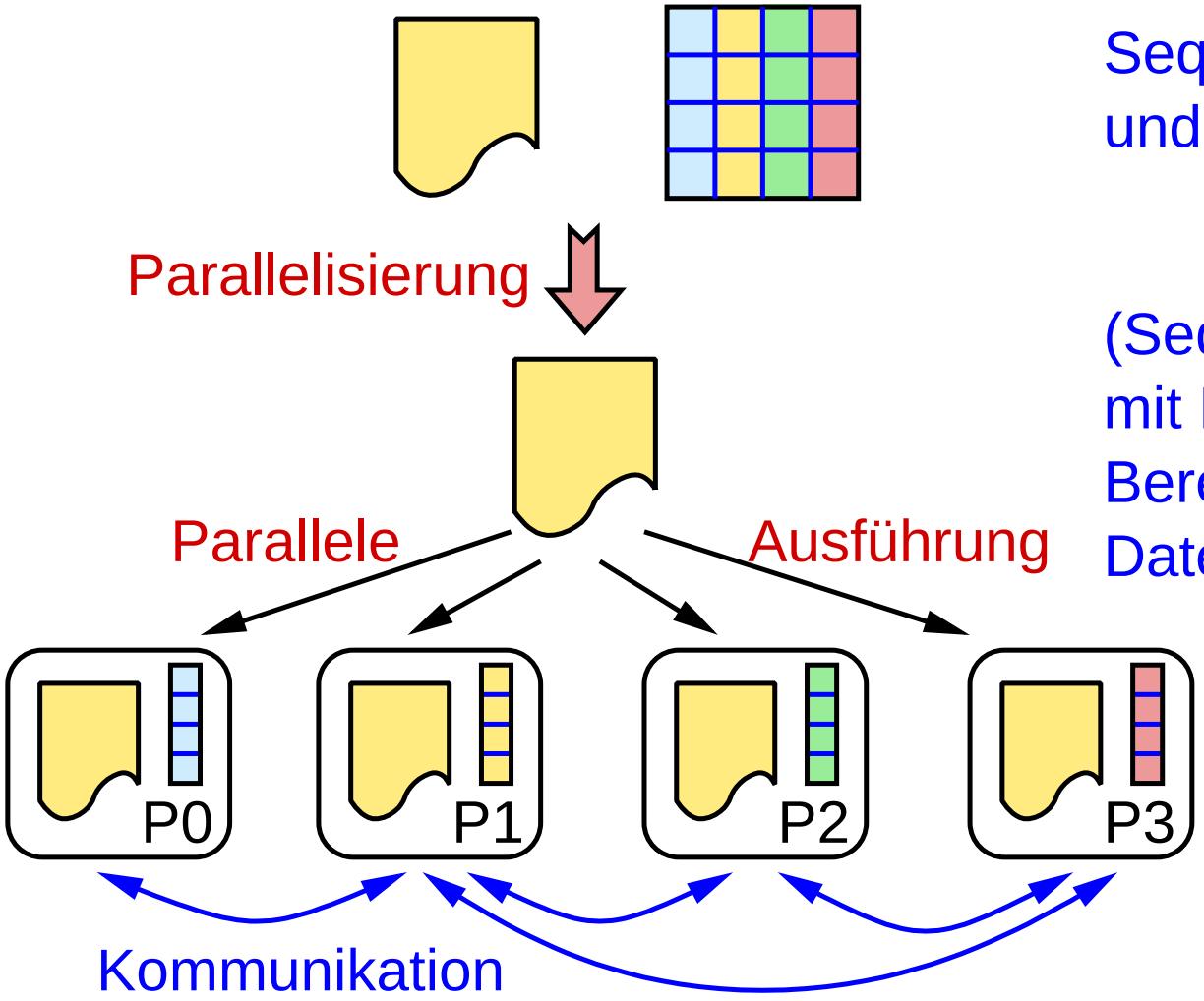
WS 2015/16

3 Parallele Programmierung mit Nachrichtenkopplung

Inhalt

- Typische Vorgehensweise
- MPI (*Message Passing Interface*)
  - Kernfunktionen
  - Einfache MPI-Programme
  - Punkt-zu-Punkt-Kommunikation
  - Komplexe Datentypen in Nachrichten
  - Kommunikatoren
  - Kollektive Operationen
  - Weitere Konzepte

Datenaufteilung mit SPMD-Modell



Sequentielles Programm und Datenaufteilung

(Sequentielles) Knotenprogramm mit Nachrichtenaustausch, führt Berechnung für einen Teil der Daten durch

Identische Kopien des Programms, parallel duch mehere Prozesse ausgeführt

Aktivitäten zur Erstellung des Knotenprogramms

- Anpassung der Array-Deklarationen
  - Knotenprogramm speichert nur einen Teil der Daten
  - (Annahme: Daten in Arrays gespeichert)
- Index-Transformation
  - globaler Index $\leftrightarrow$ (Prozeßnummer, lokaler Index)
- Arbeitsaufteilung
  - jeder Prozeß führt Berechnungen auf seinem Teil der Daten durch
- Kommunikation
  - falls ein Prozeß nicht-lokale Daten benötigt, muß ein entsprechender Nachrichtenaustausch programmiert werden

Zur Kommunikation

- Wenn ein Prozeß Daten benötigt: Eigentümer der Daten muß diese explizit senden
  - Ausnahme: einseitige Kommunikation (☞ **3.2.7**)
- Kommunikation sollte möglichst zusammengefaßt werden
  - eine große Nachricht ist besser als viele kleine
  - Datenabhängigkeiten dürfen dabei nicht verletzt werden

Sequentieller Ablauf

```
a[1] = ...;
a[2] = ...;
a[3] = a[1]+...;
a[4] = a[2]+...;
```

Paralleler Ablauf

Prozeß 1

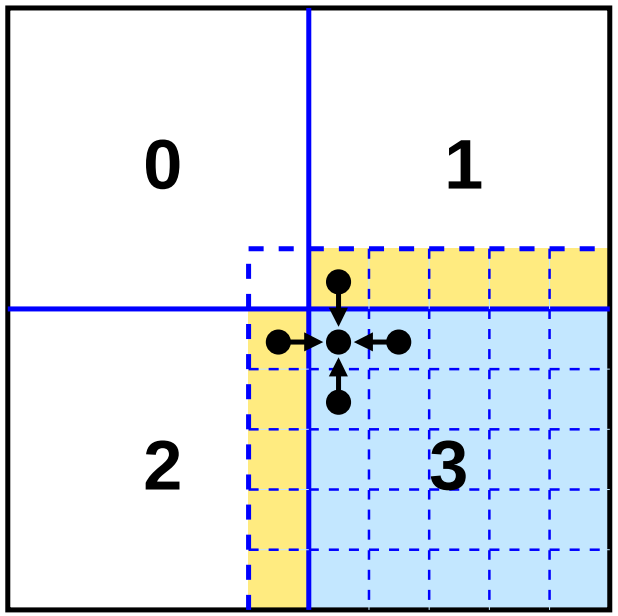
```
a[1] = ...;
a[2] = ...;
send(a[1],a[2]);
```

Prozeß 2

```
recv(a[1],a[2]);
a[3] = a[1]+...;
a[4] = a[2]+...;
```

Zur Kommunikation ...

- Oft: Knotenprogramm allokiert Pufferbereich (Überlappungs-bereich) für nicht-lokale Daten
- Beispiel: Jacobi-Iteration



Aufteilung der Matrix in 4 Teile

Jeder Prozeß allokiert zusätzlich eine Zeile/Spalte an den Rändern seiner Teilmatrix

Austausch der Daten am Ende jeder Iteration

Historie und Hintergrund

- Zu Beginn der Parallelrechner-Ära (Ende der 1980'er):
  - viele verschiedene Kommunikationsbibliotheken (NX, PARMACS, PVM, P4, ...)
  - parallele Programme schwer portierbar
- Festlegung eines Quasi-Standards durch MPI-Forum
  - 1994: MPI-1.0
  - 1997: MPI-1.2 und MPI-2.0 (erhebliche Erweiterungen)
  - 2009: MPI 2.2 (Klarstellungen, leichte Erweiterungen)
  - 2012/15: MPI-3.0 und MPI-3.1 (erhebliche Erweiterungen)
  - Dokumente unter `http://www.mpi-forum.org/docs`
- MPI definiert nur das API (d.h. die Programmier-Schnittstelle)
  - verschiedene Implementierungen, z.B. MPICH2, OpenMPI, ...

Programmiermodell

- Verteilter Speicher, Prozesse mit Nachrichtenaustausch
- SPMD: ein Programmcode für alle Prozesse
  - aber auch unterschiedliche Programmcodes möglich
- MPI-1: statisches Prozeßmodell
  - Alle Prozesse werden bei Programmstart erzeugt
    - Programmstart ist ab MPI-2 standardisiert
  - MPI-2 erlaubt auch Erzeugung neuer Prozesse zur Laufzeit
- MPI ist thread-sicher: Prozeß darf weitere Threads erzeugen
  - hybride Parallelisierung mit MPI und OpenMP ist möglich
- Programm ist beendet, wenn alle Prozesse beendet sind

3.2.1 Kernfunktionen

- MPI-1.2 hat 129 Funktionen (und MPI-2 noch mehr ...)
- Oft reichen aber bereits 6 Funktionen aus um praxisrelevante Programme zu schreiben:
  - `MPI_Init` – MPI Initialisierung
  - `MPI_Finalize` – MPI *Cleanup*
  - `MPI_Comm_size` – Anzahl der Prozesse
  - `MPI_Comm_rank` – Eigene Prozeßnummer
  - `MPI_Send` – Nachricht senden
  - `MPI_Recv` – Nachricht empfangen

MPI_Init

`int MPI_Init(int *argc, char ***argv)`

INOUT **`argc`**	Zeiger auf **`argc`** von **`main()`**
INOUT **`argv`**	Zeiger auf **`argv`** von **`main()`**
Ergebnis	**`MPI_SUCCESS`** oder Fehlercode

- Jeder MPI-Prozeß muß `MPI_Init` aufrufen, bevor andere MPI Funktionen verwendet werden können
- Typisch:

```
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  ...
```

- `MPI_Init` sorgt ggf. auch dafür, daß alle Prozesse die Kommandozeilen-Argumente erhalten

MPI_Finalize

```
int MPI_Finalize()
```

- Jeder MPI-Prozeß muß am Ende `MPI_Finalize` aufrufen
- Hauptfunktion: Ressourcenfreigabe
- Danach können keine anderen MPI-Funktionen mehr verwendet werden
  - auch kein weiteres `MPI_Init`
- `MPI_Finalize` beendet **nicht** den Prozeß!

MPI_Comm_size

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

IN	**comm**	Kommunikator
OUT	**size**	Anzahl der Prozesse in **comm**

- Typisch: `MPI_Comm_size(MPI_COMM_WORLD, &nprocs)`
  - liefert Zahl der MPI-Prozesse in `nprocs`

MPI_Comm_rank

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

IN	**comm**	Kommunikator
OUT	**rank**	Nummer des Prozesses in **comm**

- Prozessnummer („Rang“) zählt von 0 an aufwärts
  - einzige Unterscheidung der Prozesse bei SPMD

Kommunikatoren

- Ein Kommunikator besteht aus
  - einer Prozeßgruppe
    - Untermenge aller Prozesse der parallelen Anwendung
  - einem Kommunikationskontext
    - zum Auseinanderhalten verschiedener Kommunikations-beziehungen (☞ **3.2.5**)
- Es gibt einen vordefinierten Kommunikator `MPI_COMM_WORLD`
  - Prozeßgruppe umfaßt alle Prozesse der parallelen Anwendung
- Weitere Kommunikatoren können bei Bedarf erzeugt werden (☞ **3.2.5**)

MPI_Send

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

IN	**buf**	(Zeiger auf) die zu versendenden Daten (Sendepuffer)
IN	**count**	Anzahl der Datenelemente (vom Typ **dtype**)
IN	**dtype**	Datentyp der einzelnen Datenelemente
IN	**dest**	Rang des Zielprozesses im Kommunikator **comm**
IN	**tag**	Nachrichtenkennzeichnung *(Tag)*
IN	**comm**	Kommunikator

- Angabe des Datentyps: für Darstellungskonvertierung
- Zielprozeß immer relativ zu einem Kommunikator
- *Tag* zum Auseinanderhalten verschiedener Nachrichten(typen) im Programm

MPI_Send ...

- `MPI_Send` blockiert den Prozeß, bis alle Daten aus dem Sendepuffer gelesen wurden
  - Sendepuffer darf unmittelbar nach Rückkehr aus `MPI_Send` wiederverwendet (d.h. modifiziert) werden
- Implementierung entscheidet, ob solange blockiert wird, bis
  a) die Daten in einen Systempuffer kopiert wurden, oder
  b) die Daten vom Empfänger empfangen wurden.
  - diese Entscheidung kann sich u.U. auf die Korrektheit des Programms auswirken! (☞ Folie 286)

MPI_Recv

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

OUT	**buf**	(Zeiger auf) Empfangspuffer
IN	**count**	Puffergröße (Anzahl der Datenelemente vom Typ **dtype**)
IN	**dtype**	Datentyp der einzelnen Datenelemente
IN	**source**	Rang des Quellprozesses im Kommunikator **comm**
IN	**tag**	Nachrichtenkennzeichnung
IN	**comm**	Kommunikator
OUT	**status**	Status (u.a. tatsächliche Nachrichtenlänge)

- Prozeß wird blockiert, bis Nachricht empfangen und vollständig im Empfangspuffer abgelegt wurde

MPI_Recv ...

- Es wird nur eine Nachricht empfangen, bei der
  - Sender
  - Nachrichtenkennung
  - Kommunikator

  mit den Vorgaben übereinstimmen
- Für Quellprozeß (Sender) und Nachrichtenkennung (Tag) können *Wild-Cards* verwendet werden:
  - `MPI_ANY_SOURCE`: Sender ist egal
  - `MPI_ANY_TAG`: Nachrichtenkennung ist egal

MPI_Recv ...

- Nachricht darf nicht größer sein als Empfangspuffer
  - kann aber kleiner sein; unbenutzter Teil der Puffers bleibt dann unverändert
- Aus dem Rückgabewert `status` kann ermittelt werden:
  - Sender der Nachricht: `status.MPI_SOURCE`
  - Tag der Nachricht: `status.MPI_TAG`
  - Fehlercode: `status.MPI_ERROR`
  - Tatsächliche Länge der empfangenen Nachricht (Anzahl der Datenelemente): `MPI_Get_count(&status, dtype, &count)`

Einfache Datentypen (MPI_Datatype)

MPI	C	MPI	C
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short	MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long	MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float		
MPI_DOUBLE	double	MPI_LONG_DOUBLE	long double
MPI_BYTE	Byte mit 8 Bit	MPI_PACKED	Gepackte Daten*

*☞ 3.2.4

Parallelverarbeitung

WS 2015/16

11.01.2016

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Beispiel: typischer MPI-Programmrahmen (☞ 03/rahmen.cpp)

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main (int argc, char **argv)
{
    int i;
    int myrank, nprocs;
    int namelen;
    char name[MPI_MAX_PROCESSOR_NAME];

    /* MPI initialisieren und Argumente setzen */
    MPI_Init(&argc, &argv);

    /* Anzahl der Prozesse erfragen */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

3.2.2 Einfache MPI-Programme ...

```
    /* Eigenen Rang erfragen */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    /* Rechnernamen erfragen */
    MPI_Get_processor_name(name, &namelen);

    /* flush, damit Ausgabe sofort kommt */
    cout << "Process " << myrank << "/" << nprocs
         << "started on " << name << "\n" << flush;

    cout << "-- Arguments: ";
    for (i = 0; i<argc; i++)
        cout << argv[i] << " ";
    cout << "\n";

    /* MPI geordnet beenden */
    MPI_Finalize();

    return 0;
}
```

Start von MPI-Programmen: `mpiexec`

- `mpiexec -n 3 myProg arg1 arg2`
  - startet `myProg arg1 arg2` mit 3 Prozessen
  - Spezifikation der zu nutzenden Knoten ist implementierungs- und plattformabhängig
- Start des Beispiel-Programms unter MPICH:
  `mpiexec -n 3 -machinefile machines ./rahmen a1 a2`
- Ausgabe:

```
Process 0/3 started on bslab02.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 2/3 started on bslab03.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 1/3 started on bslab06.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
```

Beispiel: Ping Pong mit Nachrichten (☞ 03/pingpong.cpp)

```
int main (int argc, char **argv)
{
  int i, passes, size, myrank;
  char *buf;
  MPI_Status status;
  double start, end;

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

  passes = atoi(argv[1]); // Zahl der Durchläufe
  size = atoi(argv[2]);   // Nachrichtenlänge
  buf = malloc(size);
```

3.2.2 Einfache MPI-Programme ...

```
if (myrank == 0) {      /* Prozess 0 */

  start = MPI_Wtime(); // Aktuelle Zeit bestimmen

  for (i=0; i<passes; i++) {
    /* Nachricht an Prozess 1 senden, Tag = 42 */
    MPI_Send(buf, size, MPI_CHAR, 1, 42, MPI_COMM_WORLD);

    /* Auf Antwort warten, Tag ist egal */
    MPI_Recv(buf, size, MPI_CHAR, 1, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
  }

  end = MPI_Wtime(); // Aktuelle Zeit bestimmen

  cout << "Time for one message: "
       << ((end - start) * 1e6 / (2 * passes)) << "us\n";
  cout << "Bandwidth: "
       << (size*2*passes/(1024*1024*(end-start))) << "MB/s\
}
```

```
  else {   /* Prozess 1 */

    for (i=0; i<passes; i++) {
      /* Auf Nachricht von Prozeß 0 warten, Tag ist egal */
      MPI_Recv(buf, size, MPI_CHAR, 0, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);

      /* Nachricht an Prozess 0 zurücksenden, Tag = 24 */
      MPI_Send(buf, size, MPI_CHAR, 0, 24, MPI_COMM_WORLD);
    }
  }

  MPI_Finalize();
  return 0;
}
```

Beispiel: Ping Pong mit Nachrichten ...

- Ergebnisse (auf dem XEON-Cluster):
  - `mpiexec -n 2 ... ./pingpong 1000 1`
    ```
    Time for one message: 50.094485 us
    Bandwidth: 0.019038 MB/s
    ```
  - `mpiexec -n 2 ... ./pingpong 1000 100`
    ```
    Time for one message: 50.076485 us
    Bandwidth: 1.904435 MB/s
    ```
  - `mpiexec -n 2 ... ./pingpong 100 1000000`
    ```
    Time for one message: 9018.934965 us
    Bandwidth: 105.741345 MB/s
    ```
- (Nur) bei großen Nachrichten wird die Bandbreite des Verbindungsnetzes erreicht
  - XEON-Cluster: 1 GBit/s Ethernet ($\hat{=}$ 119.2 MB/s)

Weitere MPI-Funktionen in den Beispielen:

```
int MPI_Get_processor_name(char *name, int *len)
```

OUT	**name**	Zeiger auf Puffer für Rechnernamen
OUT	**len**	Länge des Rechnernamens
Ergebnis		**MPI_SUCCESS** oder Fehlercode

- Puffer für Rechnernamen sollte Länge `MPI_MAX_PROCESSOR_NAME` besitzen

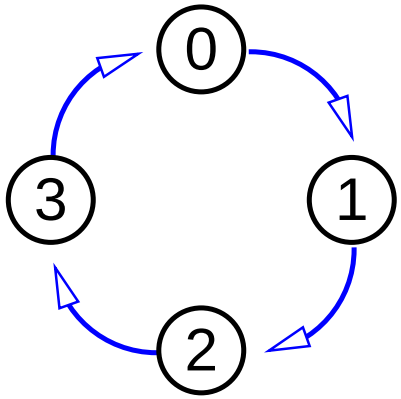
```
double MPI_Wtime()
```

Ergebnis	Aktuelle Uhrzeit in Sekunden

- für Zeitmessungen
- bei MPICH2: Zeit ist zwischen den Knoten synchronisiert

Beispiel: Senden im Kreis (☞ 03/ring.cpp)

```
int a[N];
...
MPI_Send(a, N, MPI_INT, (myrank+1) % nprocs,
         0, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_INT,
         (myrank+nprocs-1) % nprocs,
         0, MPI_COMM_WORLD, &status);
```



- Jeder Prozeß versucht zuerst zu senden, bevor er empfängt
- Funktioniert nur, **falls** MPI die Nachrichten puffert
- `MPI_Send` kann auch blockieren, bis Nachricht empfangen ist
  - Deadlock!

Beispiel: Senden im Kreis (korrekt)

- Einige Prozesse müssen erst empfangen und dann senden:

```
int a[N];
...
if (myrank % 2 == 0) {
   MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
   MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
}
else {
   MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
   MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
}
```

- Besser: nichtblockierende Operationen

Nichtblockierende Kommunikation

- `MPI_Isend` und `MPI_Irecv` kehren sofort zurück
  - bevor Nachricht wirklich gesendet / empfangen wurde
  - Ergebnis: Anfrage-Objekt (`MPI_Request`)
  - Sende- bzw. Empfangspuffer dürfen nicht verändert / verwendet werden, bis Kommunikation abgeschlossen ist
- `MPI_Test` prüft, ob Kommunikation abgeschlossen ist
- `MPI_Wait` blockiert, bis Kommunikation abgeschlossen ist
- Erlaubt Überlappung von Kommunikation und Berechung
- Kann mit blockierender Kommunikation “gemischt” werden
  - z.B. Senden mit `MPI_Send`, Empfangen mit `MPI_Irecv`

Beispiel: Senden im Kreis mit `MPI_Irecv` (☞ 03/ring2.cpp)

```
int sbuf[N];
int rbuf[N];
MPI_Status status;
MPI_Request request;
...
// Empfangs−Request aufsetzen
MPI_Irecv(rbuf, N, MPI_INT, (myrank+nprocs-1) % nprocs, 0,
          MPI_COMM_WORLD, &request);
// Senden
MPI_Send(sbuf, N, MPI_INT, (myrank+1) % nprocs, 0,
         MPI_COMM_WORLD);
// Auf Empfang der Nachricht warten
MPI_Wait(&request, &status);
```

3.2.4 Komplexe Datentypen in Nachrichten

- Bisher: Nur Arrays konnten als Nachrichten verschickt werden
- Was ist mit komplexeren Datentypen (z.B. Strukturen)?
  - z.B. `struct bsp { int a; double b[3]; char c; };`
- MPI bietet zwei Mechanismen
  - **Packen und Auspacken** der einzelnen Komponenten
    - `MPI_Pack` packt Komponenten nacheinander in Puffer, Versenden als `MPI_PACKED`, mit `MPI_Unpack` können die Komponenten wieder extrahiert werden
  - **Abgeleitete Datentypen**
    - `MPI_Send` erhält Zeiger auf die Datenstruktur, sowie eine Beschreibung des Datentyps
    - die Beschreibung des Datentyps muß durch Aufrufe von MPI-Funktionen erzeugt werden

Abgeleitete Datentypen

- MPI stellt Konstruktoren zur Verfügung, mit denen eigene (abgeleitete) Datentypen definiert werden können:
  - für zusammenhängende Daten: `MPI_Type_contiguous`
    - erlaubt Definition von Array-Typen
  - für nicht-zusammenhängende Daten: `MPI_Type_vector`
    - z.B. für Spalte einer Matrix oder Unter-Matrix
  - für Strukturen: `MPI_Type_create_struct`
- Nach der Erstellung muß der neue Datentyp bekanntgegeben werden: `MPI_Type_commit`
- Danach kann der Datentyp wie ein vordefinierter Datentyp (z.B. MPI_INT) verwendet werden

3.2.4 Komplexe Datentypen in Nachrichten ...

MPI_Type_vector: nicht-zusammenhängende Arrays

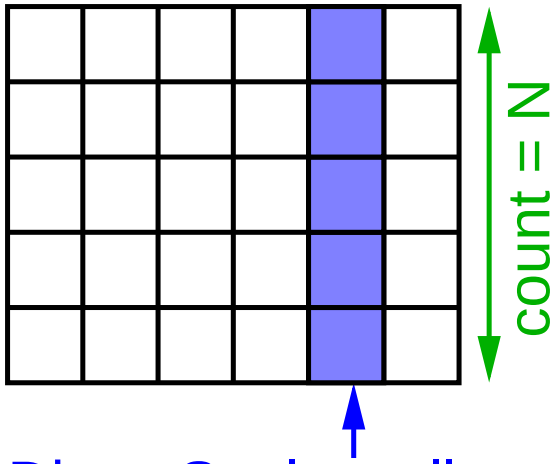
```
int MPI_Type_vector(int count, int blocklen, int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

IN	**count**	Zahl der Datenblöcke
IN	**blocklen**	Länge der einzelnen Datenblöcke
IN	**stride**	Abstand zwischen aufeinanderfolgenden Datenblöcken
IN	**oldtype**	Typ der Elemente in den Datenblöcken
OUT	**newtype**	neu erzeugter Typ

- Fasst eine Anzahl von Datenblöcken (als Arrays beschrieben) zu einem neuen Datentyp zusammen
- Das Ergebnis ist aber eher eine neue **Sicht** auf vorhandene Daten als ein neuer Daten**typ**

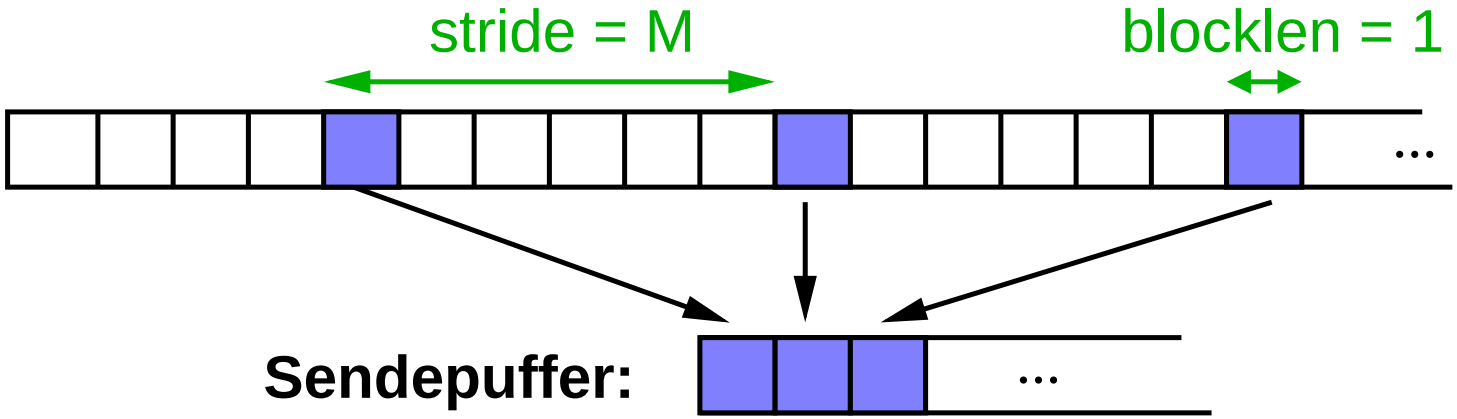
Beispiel: Übertragung einer Spalte einer Matrix

Matrix: a[N][M]



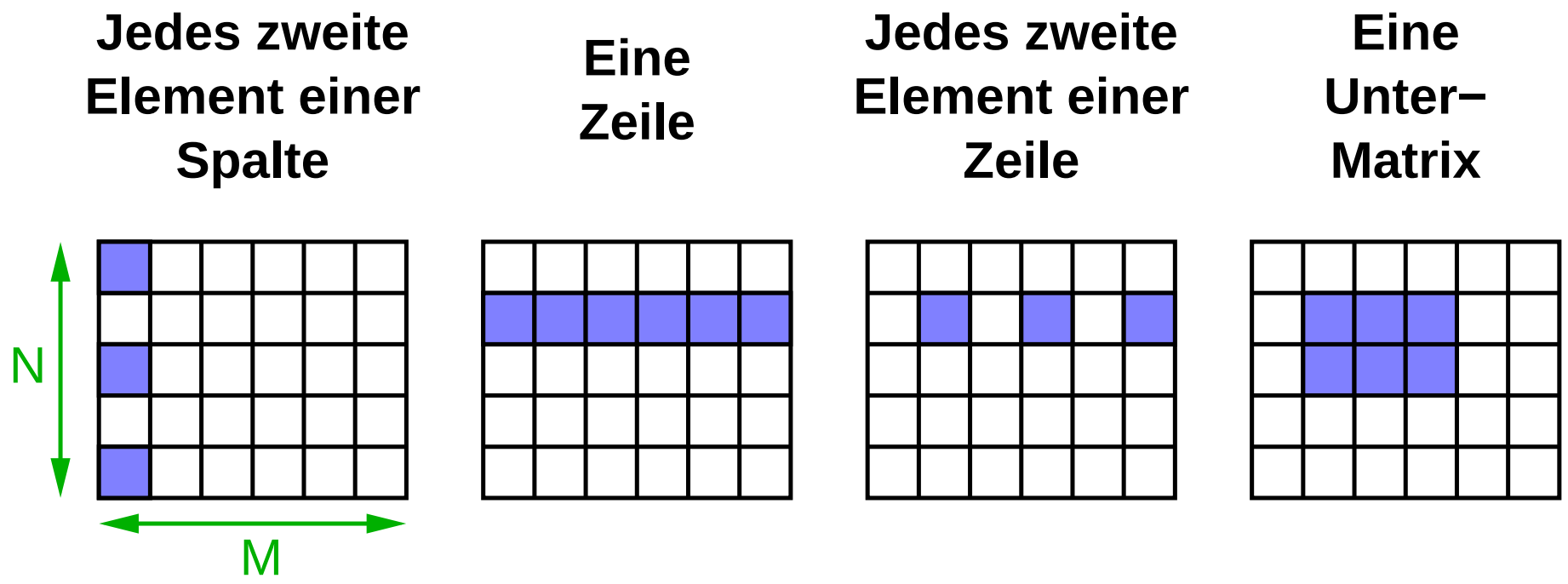
Diese Spalte soll gesendet werden

Speicher−Layout der Matrix:




```
MPI_type_vector(N, 1, M, MPI_INT, &spalte);
MPI_Type_commit(&spalte);
// Spalte übertragen
if (rank==0) MPI_Send(&a[0][4], 1, spalte, 1, 0, comm);
else MPI_Recv(&a[0][4], 1, spalte, 0, 0, comm, &status);
```

Weitere Möglichkeiten von `MPI_Type_vector`



count	N / 2	1	M	M / 2	2
blocklen	1	M	1	1	3
stride	2 * M	x	1	2	M

Anmerkung zu `MPI_Type_vector`

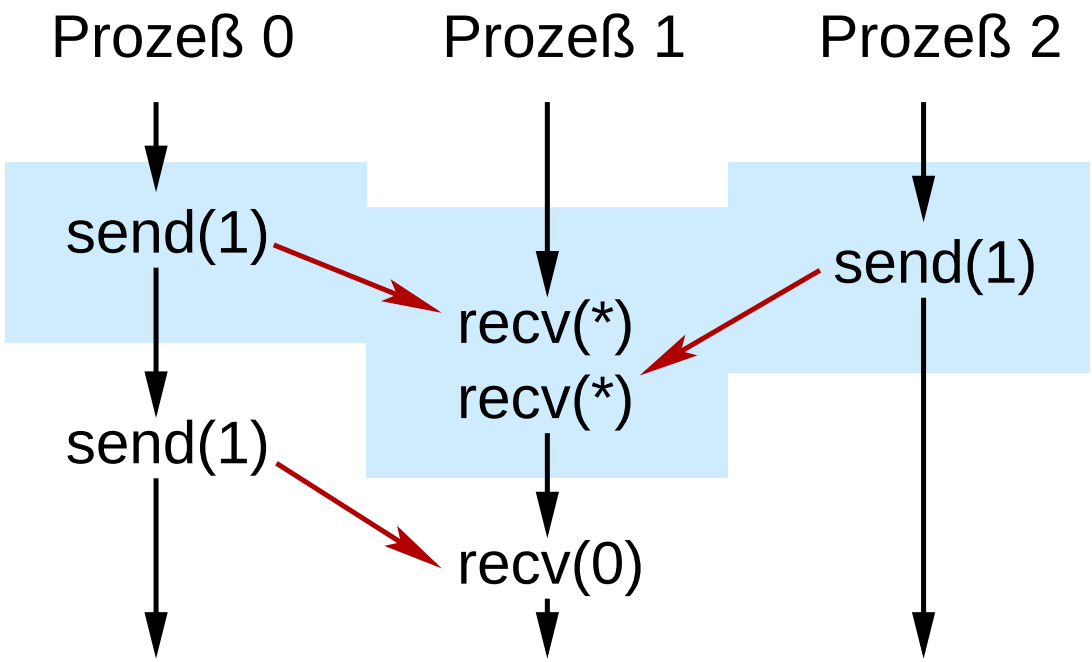
- Empfänger kann anderen Datentyp verwenden wie der Sender
- Vorgeschrieben ist nur, daß Zahl und Typ-Abfolge der gesendeten und empfangenen Elemente identsich sind
- Damit z.B. möglich:
  - Sender schickt eine Spalte einer Matrix
  - Empfänger speichert diese in einem Zeilen-Array

```
int a[N][M], b[N];
MPI_type_vector(N, 1, M, MPI_INT, &spalte);
MPI_Type_commit(&spalte);
if (rank==0) MPI_Send(&a[0][4], 1, spalte, 1, 0, comm);
else MPI_Recv(b, N, MPI_INT, 0, 0, comm, &status);
```

Auswahl der richtigen Vorgehensweise

- Homogene Daten (Elemente mit gleichem Typ):
  - zusammenhängend (*stride* 1): Standard-Datentyp und `count`-Parameter
  - nicht zusammenhängend:
    - *stride* ist konstant: `MPI_Type_vector`
    - *stride* ist irregulär: `MPI_Type_indexed`
- Heterogene Daten (Elemente unterschiedlichen Typs):
  - viel Daten, oft versendet: `MPI_Type_create_struct`
  - wenig Daten, selten versendet: `MPI_Pack` / `MPI_Unpack`
  - Strukturen variabler Länge: `MPI_Pack` / `MPI_Unpack`

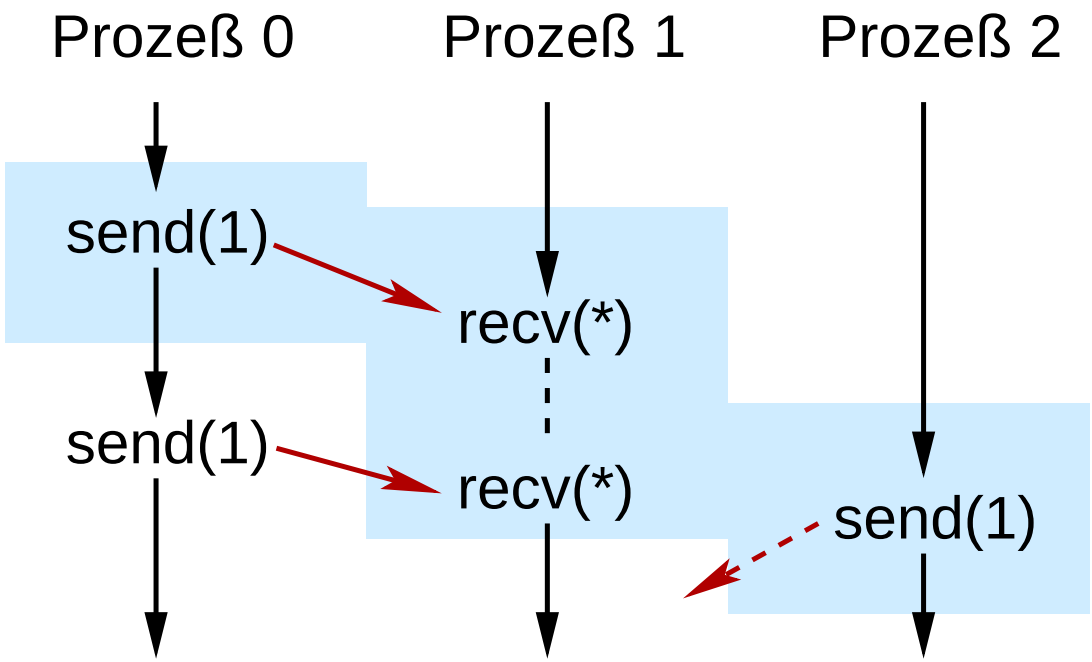
Motivation: Problem früherer Kommunikationsbibliotheken



: Code in einer parallelen (z.B. numerischen) Bibliothek

Prozeß 1 soll von 0 und 2 empfangen (in beliebiger Reihenfolge)

Motivation: Problem früherer Kommunikationsbibliotheken



: Code in einer parallelen (z.B. numerischen) Bibliothek

Falls Prozeß 2 aus irgend einem Grund "verspätet" ist: Kommunikation schlägt fehl!

- Nachrichten-Tags sind keine zuverlässige Lösung
  - Tags könnten zufällig gleich gewählt werden!
- Nötig: unterschiedliche Kommunikations-Kontexte

3.2.5 Kommunikatoren ...

- Kommunikator = Prozeßgruppe + Kontext
- Kommunikatoren unterstützen
  - das Arbeiten mit Prozeßgruppen
    - Task-Parallelismus
    - gekoppelte Simulationen
    - kollektive Kommunikation mit Teilmenge aller Prozesse
  - Kommunikations-Kontexte
    - für parallele Bibliotheken
- Ein Kommunikator repräsentiert eine Kommunikationsdomäne
  - Kommunikation nur innerhalb derselben Domäne möglich
    - kein *Wild-Card* für Kommunikator bei `MPI_Recv`
  - ein Prozeß kann mehreren Domänen angehören

Erzeugung neuer Kommunikatoren

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color
                   int key, MPI_Comm *newcomm)
```

- Kollektive Operationen
  - von allen Prozessen in `comm` „gleichzeitig“ ausgeführt
- `MPI_Comm_dup` erzeugt Kopie mit neuem Kontext
- `MPI_Comm_split` spaltet `comm` in mehrere Kommunikatoren auf
  - ein Kommunikator für jeden Wert von `color`
  - jeder Prozeß bekommt den Kommunikator als Ergebnis, dem er zugeteilt wurde
  - `key` legt Reihenfolge der neuen Ränge fest

Beispiel zu MPI_Comm_split

- *Multi-physics Code*: Luftverschmutzung
  - die Hälfte der Prozesse berechnet Luftströmung
  - die andere Hälfte berechnet chemische Reaktionen
- Erzeugung von zwei Kommunikatoren für die beiden Teile:

```
MPI_Comm_split(MPI_COMM_WORLD, myrank%2, myrank, &comm)
```

Prozeß	`myrank`	Farbe	Ergebnis in `comm`	Rang in C_0	Rang in C_1
P0	0	0	C_0	0	–
P1	1	1	C_1	–	0
P2	2	0	C_0	1	–
P3	3	1	C_1	–	1

3.2.6 Kollektive Operationen

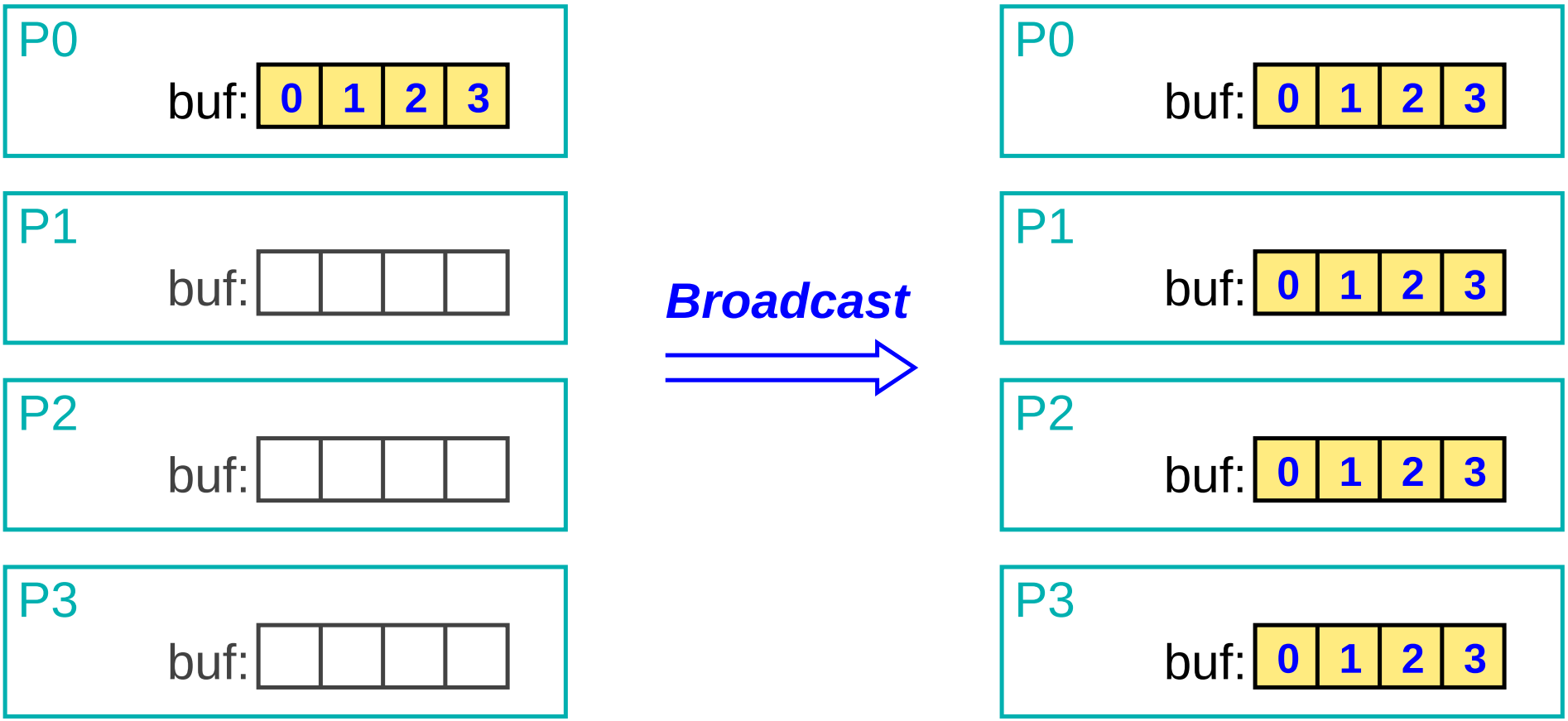
- Kollektive Operationen in MPI
  - müssen von allen Prozessen einer Prozeßgruppe (eines Kommunikators) „gleichzeitig“ ausgeführt werden
  - sind blockierend
  - führen aber nicht notwendigerweise zu einer globalen (Barrieren-)Synchronisation
- Kollektive Synchronisations- / Kommunikations-Funktionen:
  - Barrieren
  - Kommunikation: *Broadcast*, *Scatter*, *Gather*, ...
  - Reduktionen (Kommunikation mit Aggregation)

MPI_Barrier

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

- Barrieren-Synchronisation aller Prozesse aus `comm`
- Barriere bei Nachrichtenaustausch eigentlich nicht notwendig
  - Synchronisation erfolgt beim Nachrichtenaustausch
- Gründe für Barrieren:
  - Einfacheres Verständnis des Programms
  - Zeitmessungen, Debugging-Ausgaben
  - Ein-/Ausgabe?? (Bildschirm, Datei)
  - Einige sehr spezielle Fälle in MPI (s. Standard ...)

Kollektive Kommunikation: *Broadcast*



MPI_Bcast

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

IN **root** Rang des sendenden Prozesses

- Puffer wird von Prozeß `root` gesendet und von allen anderen empfangen
- Kollektive, blockierende Operation: kein Tag nötig
- `count`, `dtype`, `root`, `comm` muß in allen Prozessen gleich sein

Parallelverarbeitung

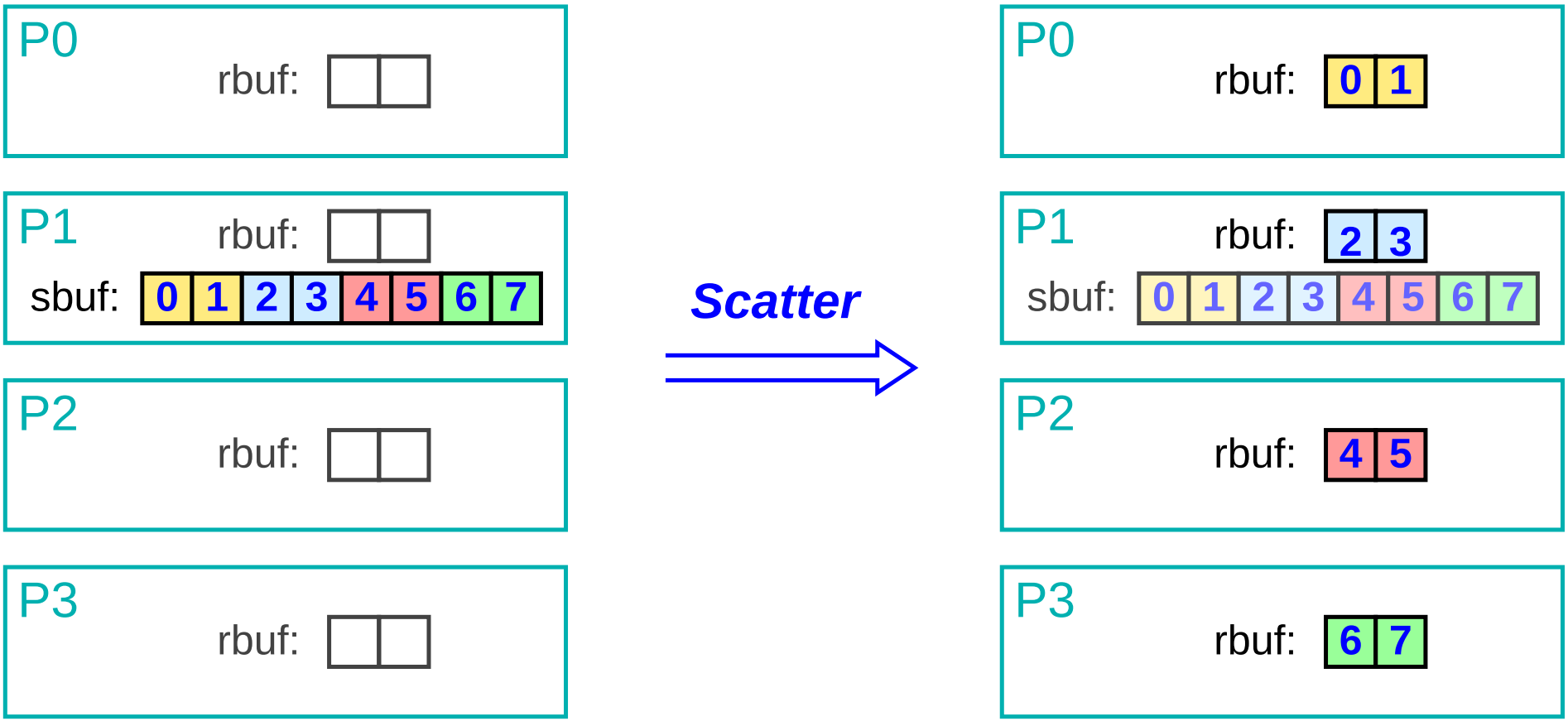
WS 2015/16

25.01.2016

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Kollektive Kommunikation: *Scatter*

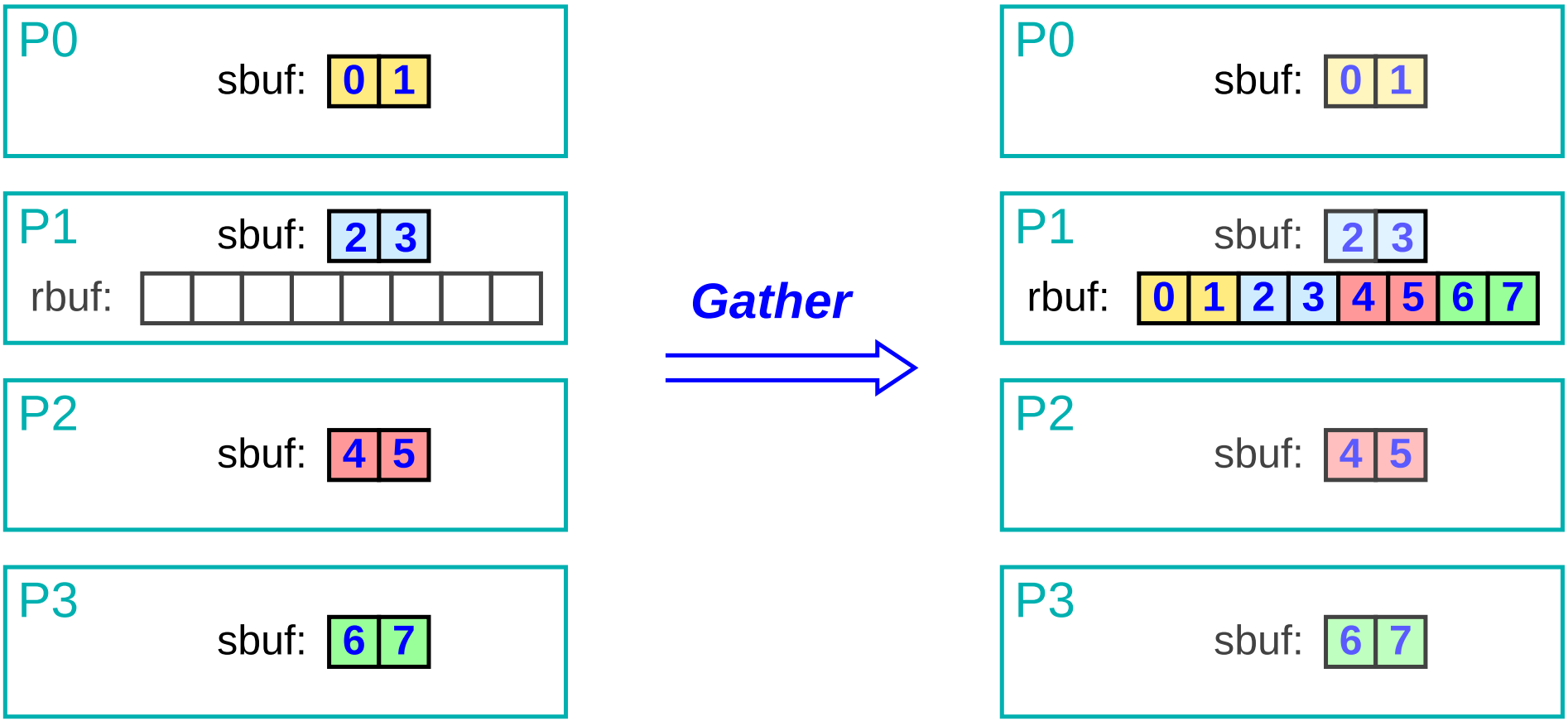


MPI_Scatter

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

- Prozeß `root` sendet einen Teil der Daten an jeden Prozeß
  - einschließlich sich selbst
- `sendcount`: Daten-Länge für jeden Prozeß (nicht Gesamtlänge!)
- Prozeß `i` erhält `sendcount` Elemente aus `sendbuf` ab Position `i * sendcount`
- Variante `MPI_Scatterv`: Länge und Position kann für jeden Empfänger individuell festgelegt werden

Kollektive Kommunikation: *Gather*



MPI_Gather

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

- Alle Prozesse senden `sendcount` Elemente an Prozeß `root`
  - auch `root` selbst
- Wichtig: jeder Prozeß muß gleiche Datenmenge senden
- Daten von Prozeß `i` werden bei `root` ab Position `i * recvcount` in `recvbuf` gespeichert
- `recvcount`: Daten-Länge von jedem Prozeß (nicht Gesamtlänge!)
- Variante `MPI_Gatherv`: analog zu `MPI_Scatterv`

Beispiel: Multiplikation Vektor mit Skalar (☞ 03/vecmult.cpp)

```
double a[N], factor, local_a[LOCAL_N];
... // Prozeß 0 liest a und factor aus Datei
MPI_Bcast(&factor, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(a, LOCAL_N, MPI_DOUBLE, local_a, LOCAL_N,
            MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0; i<LOCAL_N; i++)
   local_a[i] *= factor;
MPI_Gather(local_a, LOCAL_N, MPI_DOUBLE, a, LOCAL_N,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
... // Prozeß 0 schreibt a in Datei
```

- ➥ **Achtung:** LOCAL_N muß in allen Prozessen gleich sein!
  - ➥ sonst: MPI_Scatterv / MPI_Gatherv nutzen (☞ 03/vecmult3.cpp)

Reduktion: MPI_Reduce

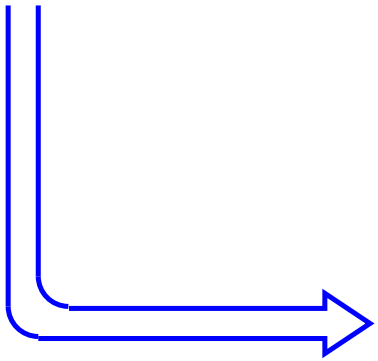
```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype dtype,
               MPI_Op op, int root,
               MPI_Comm comm)
```

- Jedes Element im Empfangspuffer ist Resultat einer Reduktions-Operation (z.B. Summe) der entsprechenden Elemente der Sendepuffer
- `op` definiert die Operation
  - vordefiniert: Minimum, Maximum, Summe, Produkt, AND, OR, XOR, ...
  - auch benutzerdefinierte Operationen möglich

Beispiel: Summation eines Arrays

Sequentiell

```
s = 0;
for (i=0;i<size;i++)
  s += a[i];
```



Parallel

```
local_s = 0;
for (i=0;i<local_size;i++)
   local_s += a[i];

MPI_Reduce(&local_s, &s,
           1, MPI_INT,
           MPI_SUM,
           0, MPI_COMM_WORLD);
```

Weitere kollektive Kommunikationsoperationen

- `MPI_Alltoall`: All-to-All-Broadcast (☞ **1.9.5**)
- `MPI_Allgather` und `MPI_Allgatherv`: Alle Prozesse haben am Ende die gesammelten Daten
  - entspricht *Gather* mit anschließendem *Broadcast*
- `MPI_Allreduce`: Alle Prozesse haben am Ende das Ergebnis der Reduktion
  - entspricht *Reduce* mit anschließendem *Broadcast*
- `MPI_Scan`: Präfix-Reduktion
  - z.B. mit Summe: Prozeß i erhält Summe der Daten von Prozeß 0 bis einschließlich i

3.2.7 Weitere Konzepte

- Topologien
  - Kommunikationsstruktur der Anwendung wird in einem Kommunikator hinterlegt
    - z.B. kartesisches Gitter
  - erlaubt Vereinfachung und Optimierung der Kommunikation
    - z.B. „sende an linken Nachbarn“
    - kommunizierende Prozesse auf benachbarten Knoten
- Dynamische Prozeßerzeugung (ab MPI-2)
  - neue Prozesse können zur Laufzeit erzeugt werden
  - Prozeßerzeugung ist kollektive Operation
  - erzeugte Prozeßgruppe erhält eigenes `MPI_COMM_WORLD`
    - Kommunikation zwischen den Prozessgruppen durch *Intercommunicator*

3.2.7 Weitere Konzepte ...

- Einseitige Kommunikation (ab MPI-2)
  - Zugriff auf Adreßraum anderer Prozesse
  - Operationen: Lesen, Schreiben, atomarer Update
  - abgeschwächte Konsistenz
    - explizite *Fence* bzw. *Lock*/*Unlock*-Operationen zur Synchronisation
  - Nutzung: Anwendungen mit irregulärer Kommunikation
    - ein Prozess kann die Kommunikation alleine abwickeln
- Parallele Ein-/Ausgabe (ab MPI-2)
  - Prozesse haben individuelle Sicht auf Datei
    - beschrieben durch MPI-Datentyp
  - Dateioperationen: individuell / kollektiv, privater / gemeinsamer Dateizeiger, blockierend / nichtblockierend

3.2.8 Zusammenfassung

- Basisfunktionen:
  - `Init`, `Finalize`, `Comm_size`, `Comm_rank`, `Send`, `Recv`
- Komplexe Datentypen in Nachrichten
  - `Pack` und `Unpack`
  - benutzerdefinierte Datentypen
    - auch für nichtzusammenhängende Daten (z.B. Spalte einer Matrix)
- Kommunikatoren: Prozeßgruppe + Kommunikationskontext
- Nichtblockierende Kommunikation: `Isend`, `Irecv`, `Test`, `Wait`
- Kollektive Operationen
  - `Barrier`, `Bcast`, `Scatter(v)`, `Gather(v)`, `Reduce`, ...

Parallelverarbeitung

WS 2015/16

4 Optimierungstechniken

4 Optimierungstechniken ...

- Im Folgenden: Beispiele für wichtige Techniken zur Optimierung paralleler Programme
- Gemeinsamer Speicher
  - Cache-Optimierungen: Verbesserung der Lokalität von Speicherzugriffen
    - *Loop Interchange*, *Tiling*
    - *Array Padding*
  - *False Sharing*
- Nachrichtenaustausch:
  - Zusammenfassung von Nachrichten
  - *Latency Hiding*

Beispiel: Summation einer Matrix in C++ (☞ 04/sum.cpp)

```
double a[N][N];
...
for (j=0;j<N;j++) {
  for (i=0;i<N;i++) {
    s += a[i][j];
  }
}
```
spaltenweiser Durchlauf

```
double a[N][N];
...
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
    s += a[i][j];
  }
}
```
zeilenweiser Duchlauf

N=8192:	Laufzeit: 4,15 s	Laufzeit: 0,14 s	(bsclk01, g++ −O3)
N=8193:	Laufzeit: 0,72 s	Laufzeit: 0,14 s	

- Ursache: Caches
  - höhere Trefferrate bei zeilenweiser Bearbeitung der Matrix
  - obwohl jedes Element nur einmal benutzt wird ...
- Anm.: C/C++ speichert Matrix zeilenweise, Fortran spaltenweise

Details zu Caches: Cachezeilen

- Datenspeicherung im Cache und Transport zwischen Hauptspeicher und Cache erfolgt in größeren Blöcken
  - Grund: nach Adressierung einer Speicherzelle können Folgezellen sehr schnell gelesen werden
  - Größe einer Cachezeile: 32-128 Byte
- Im Beispiel:
  - zeilenweise Bearbeitung: nach Laden der Cachezeile für `a[i][j]` stehen nachfolgend `a[i+1][j]`, `a[i+2][j]`, ... bereits im Cache zur Verfügung
  - spaltenweise Bearbeitung: Cachezeile für `a[i][j]` ist schon wieder verdrängt, bis `a[i+1][j]`, ... benutzt werden
- **Regel**: Speicher möglichst linear aufsteigend durchlaufen!

Details zu Caches: Set-assoziative Caches

- Ein Speicherblock (mit gegebener Adresse) kann nur an wenigen Stellen im Cache gespeichert werden
  - Grund: einfaches Wiederfinden der Daten in Hardware
  - Gruppe mit meist 2 oder 4 Einträgen
  - Eintrag in der Gruppe durch LRU-Strategie bestimmt
- Die unteren k Bits der Adresse bestimmen die Gruppe (k abhängig von Cache-Größe und Assoziativitätsgrad)
  - für alle Speicherzellen, deren untere k Adreßbits gleich sind, stehen nur 2 - 4 Cache-Einträge zur Verfügung!

Details zu Caches: Set-assoziative Caches ...

- Im Beispiel: bei $N = 4096$ und spaltenweiser Abarbeitung
  - Cache-Eintrag wird garantiert nach wenigen Iterationen der i-Schleife verdrängt (Adreßabstand ist Zweierpotenz)
  - Cache-Trefferrate ist praktisch Null
- **Regel**: Durchlaufen des Speichers mit Zweierpotenz als Schrittweite vermeiden!
  - (Zweierpotenzen als Matrixgrößen bei großen Matrizen vermeiden)

Wichtige Cache-Optimierungen

- *Loop Interchange*: Vertauschen von Schleifen
  - so, daß Speicher linear aufsteigend durchlaufen wird
  - bei C/C++: Matrizen zeilenweise durchlaufen
  - bei Fortran: Matrizen spaltenweise durchlaufen
- *Array Padding*
  - Matrizen ggf. größer allokieren, als notwendig, um Zweierpotenz als Zeilenlänge zu vermeiden
- *Tiling*: Blockaufteilung von Schleifeniterationen
  - Algorithmen so umstrukturieren, daß möglichst lange mit Teilmatrizen gearbeitet wird, die vollständig in die Caches passen

Beispiel: Matrizenmultiplikation (☞ 04/matmult.c)

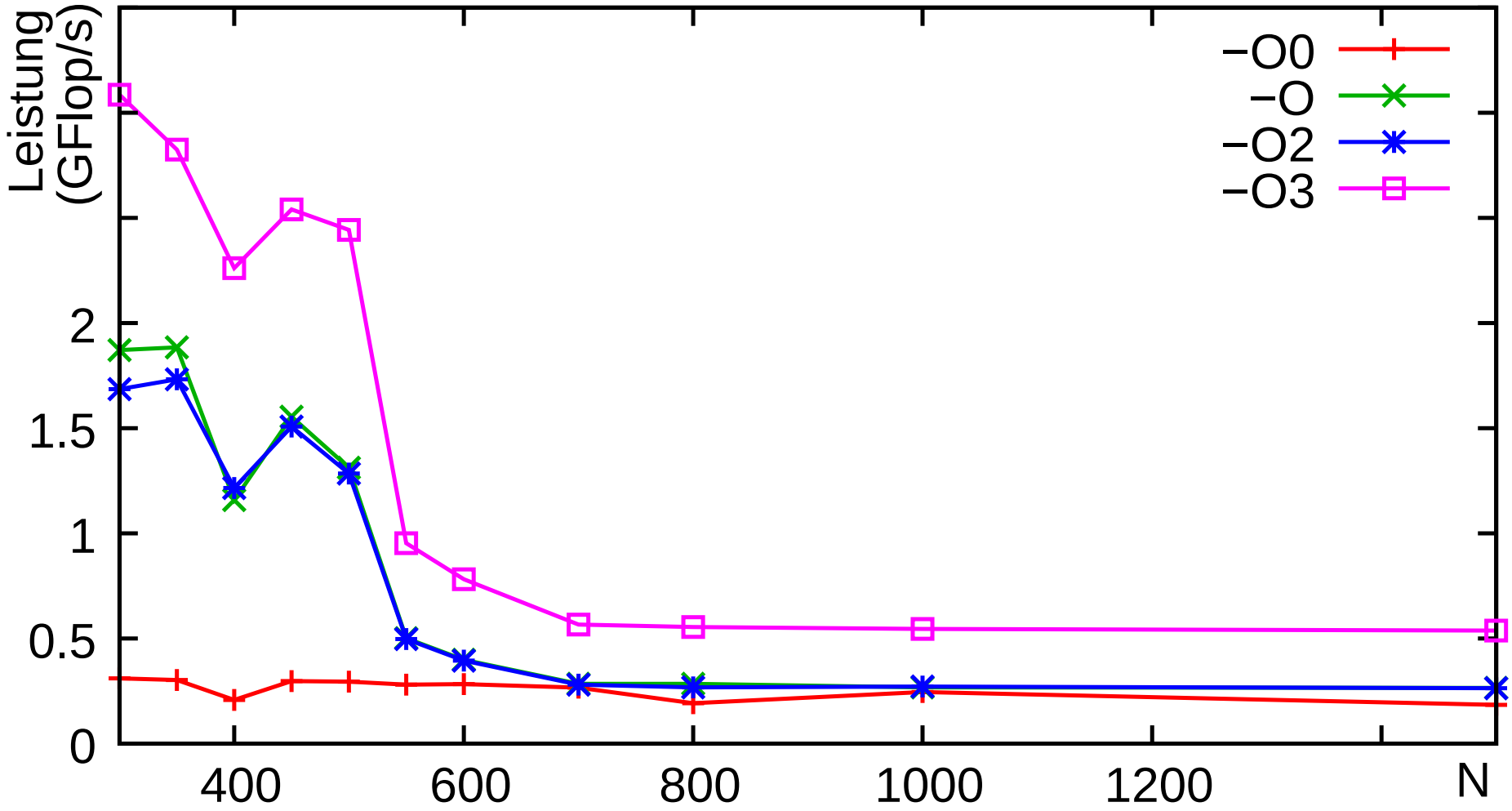
- Naiver Code:

```
double a[N][N], b[N][N], ...
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

- Leistung mit verschiedenen Optimierungsstufen:
  (N=500, g++ 4.6.3, Intel Core i7 2.8 GHz (bspc02))
  - -O0: 0.3 GFlop/s
  - -O: 1.3 GFlop/s
  - -O2: 1.3 GFlop/s
  - -O3: 2.4 GFlop/s (SIMD-Vektorisierung!)

Beispiel: Matrizenmultiplikation ...

➥ Skalierbarkeit der Leistung für verschiedene Matrixgrößen:



Beispiel: Matrizenmultiplikation ...

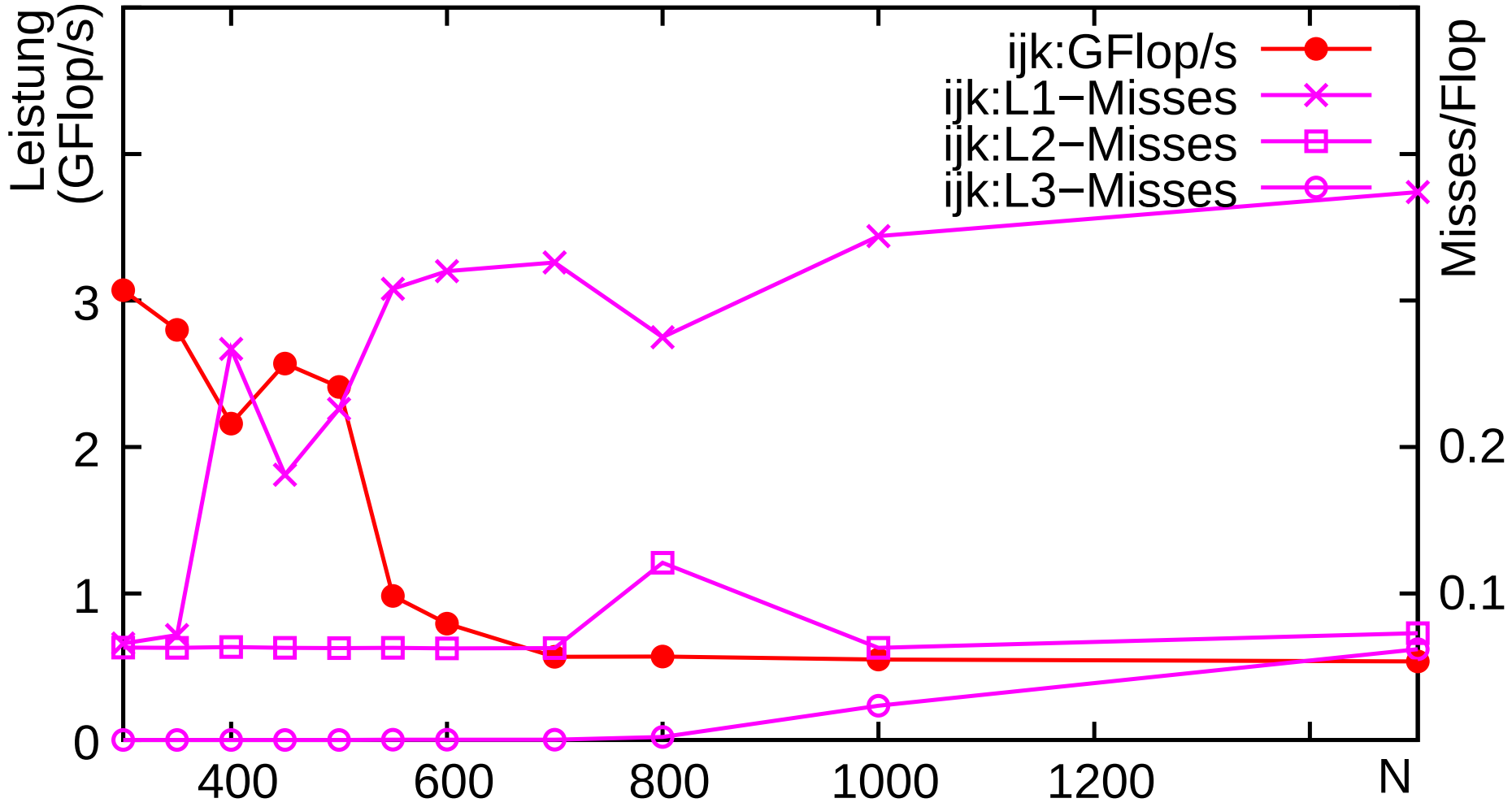
- Optimierte Reihenfolge der Schleifen:

```
double a[N][N], b[N][N], ...
for (i=0; i<N; i++)
  for (k=0; k<N; k++)
    for (j=0; j<N; j++)
      c[i][j] += a[i][k] * b[k][j];
```

- Matrix b wird jetzt zeilenweise durchlaufen
  - erheblich weniger L1-Cache-Misses
  - deutlich bessere Leistung:
    - N=500, -O3: 4.2 GFlop/s statt 2.4 GFlop/s
  - deutlich bessere Skalierbarkeit

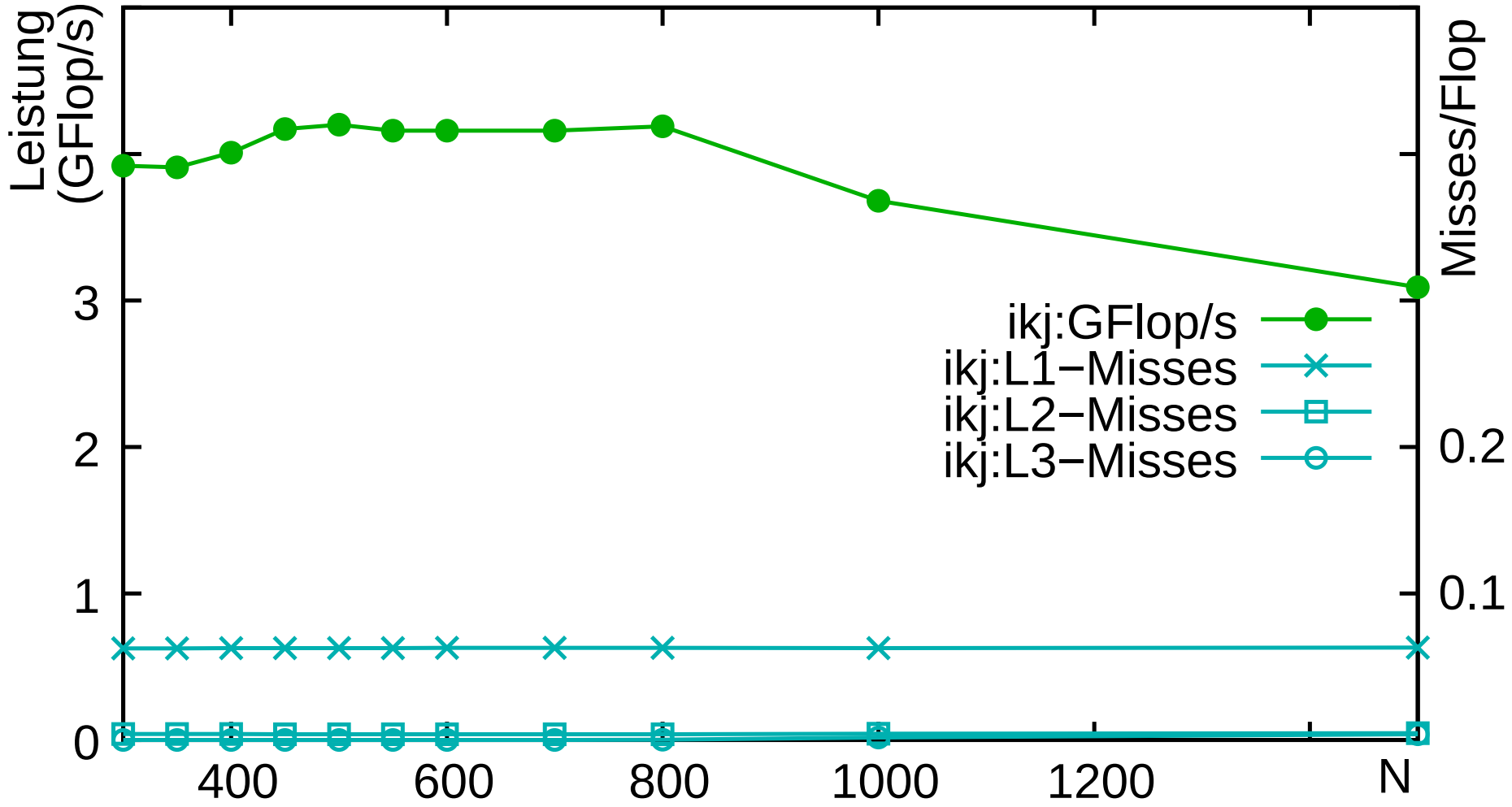
Beispiel: Matrizenmultiplikation ...

- Vergleich der beiden Schleifen-Reihenfolgen:



Beispiel: Matrizenmultiplikation ...

➥ Vergleich der beiden Schleifen-Reihenfolgen:



Parallelverarbeitung

WS 2015/16

01.02.2016

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

Beispiel: Matrizenmultiplikation ...

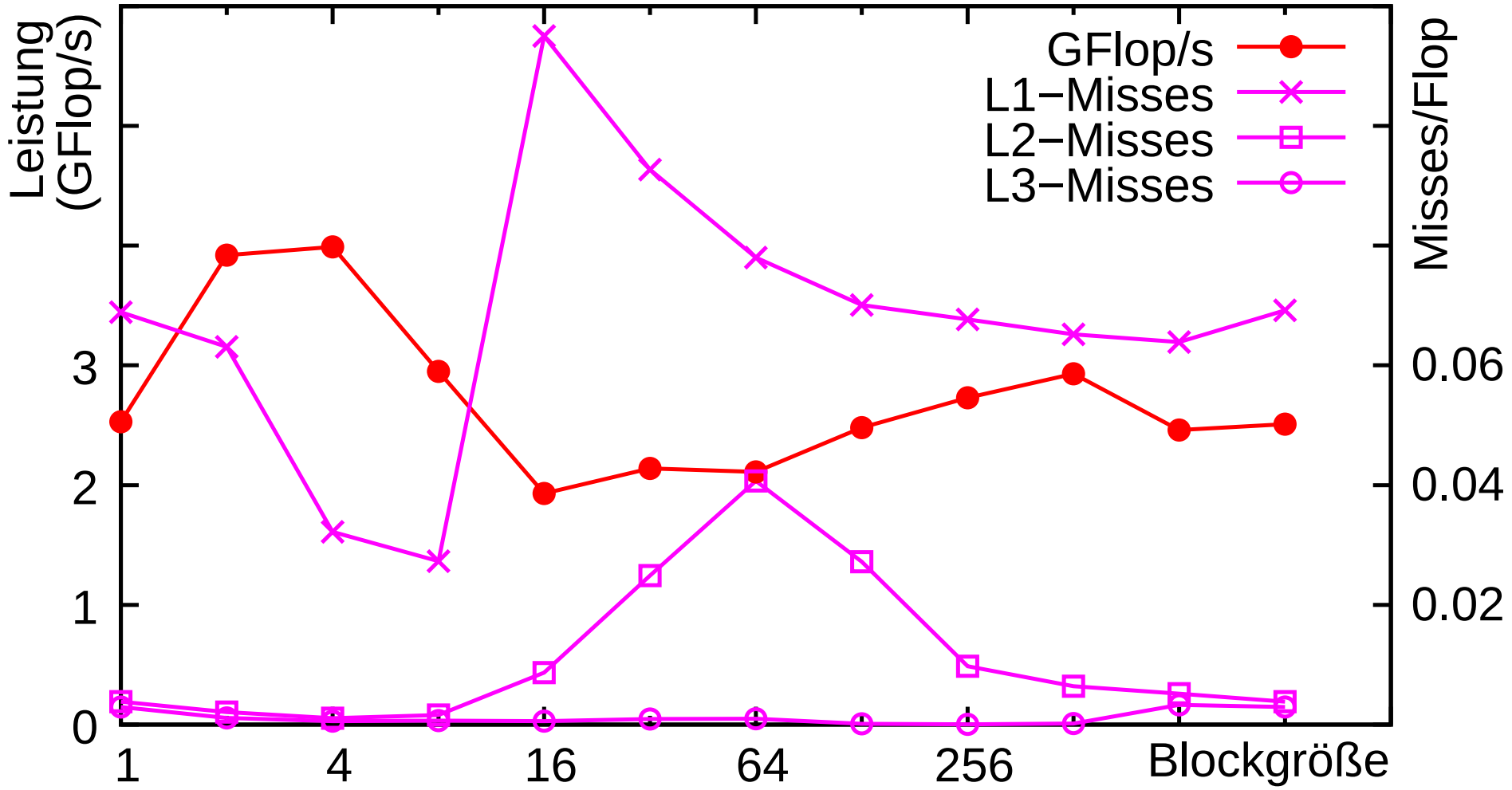
- Blockalgorithmus (*Tiling*) mit *Array Padding*:

```
double a[N][N+1], b[N][N+1], ...
for (ii=0; ii<N; ii+=4)
 for (kk=0; kk<N; kk+=4)
  for (jj=0; jj<N; jj+=4)
   for (i=0; i<4; i++)
    for (k=0; k<4; k++)
     for (j=0; j<4; j++)
      c[i+ii][j+jj] += a[i+ii][k+kk] * b[k+kk][j+jj];
```

- Matrix wird als Matrix von 4x4 Untermatrizen aufgefasst
  - Multiplikation der Untermatrizen passt in L1-Cache
- Erreicht Leistung von 4 GFlop/s auch bei N=2048

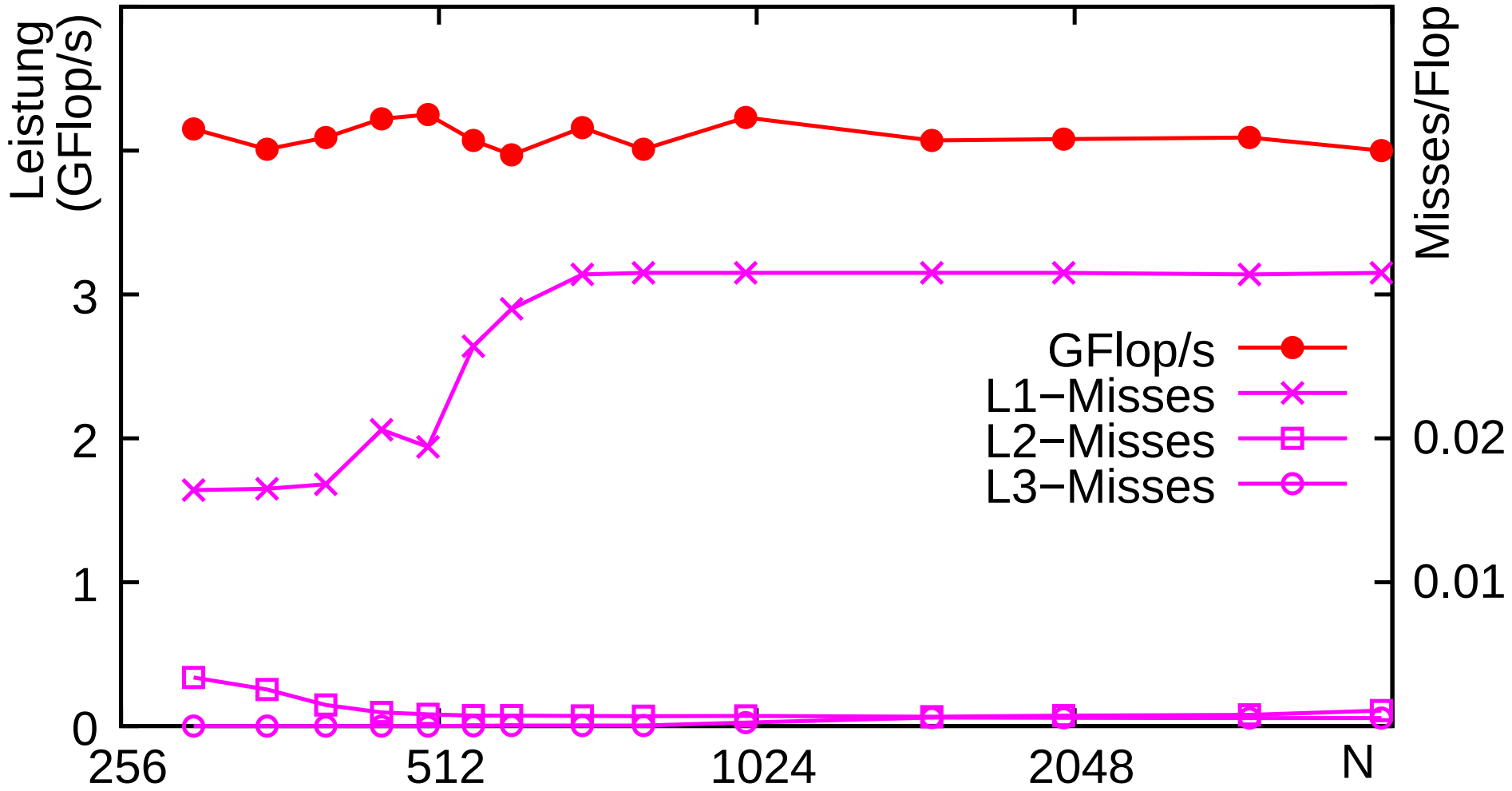
Beispiel: Matrizenmultiplikation ... ...

- Abhängigkeit der Leistung von der Blockgröße (N=2048):



Beispiel: Matrizenmultiplikation ... ...

- Skalierbarkeit der Leistung für verschiedene Matrixgrößen:



Cache-Optimierungen für Parallelrechner

- Cache-Optimierung auch und gerade bei Parallelrechnern (UMA und NUMA) wichtig
  - größerer Unterschied zwischen Cache- und Hauptspeicher-Zugriffszeit
  - Konflikte beim Zugriff auf den Hauptspeicher

- Zusätzliches Problem bei Parallelrechnern: *False Sharing*
  - mehrere logisch nicht zusammengehörige Variablen können (zufällig) in derselben Cachezeile liegen
  - Schreibzugriffe auf die Variablen führen zu häufigen Cache-Invalidierungen (wg. Cache-Kohärenz-Protokoll)
  - Leistung wird drastisch schlechter

Beispiel zu *False Sharing*: Parallele Summation eines Arrays

(☞ `04/false.cpp`)

- Globale Variable `double sum[P]` für die Partialsummen
- Variante 1: Thread `i` addiert auf `sum[i]` auf
  - Laufzeit(*) mit 4 Threads: 0.4 s, sequentiell: 0.24 s !
  - Performance-Verlust durch *False Sharing*: die `sum[i]` liegen in derselben Cache-Zeile
- Variante 2: Thread `i` addiert zunächst auf lokale Variable auf, speichert am Ende Ergebnis nach `sum[i]`
  - Laufzeit(*) mit 4 Threads: 0.09 s
- **Regel**: Variablen, die von verschiedenen Threads genutzt werden, im Speicher separieren (z.B. auch durch *Padding*)!

(*) 8000 x 8000 Matrix, Intel Xeon 2.66 GHz, ohne Compileroptimierungen

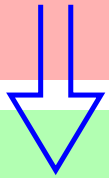
Zusammenfassen von Nachrichten

- Zeit zum Versenden kurzer Nachrichten wird durch (Software-)Latenz dominiert
  - d.h. eine lange Nachricht ist „billiger“ als viele kurze!
- Beispiel: PC-Cluster im H-A 4111 mit MPICH2
  - 32 Nachrichten á 32 Byte benötigen $\mathbf{32 \cdot 145 = 4640 \mu s}$
  - eine Nachricht mit 1024 Byte benötigt nur $\mathbf{159 \mu s}$
- Daher: zu versendende Daten nach Möglichkeit in wenige Nachrichten zusammenfassen
  - ggf. auch bei Kommunikation in Schleifen möglich (*Hoisting*)

Hoisting von Kommunikationsaufrufen

```
for (i=0; i<N; i++) {
  b = f(..., i);
  send(&b, 1, P2);
}
```

```
for (i=0; i<N; i++) {
  recv(&b, 1, P1);
  a[i] = a[i] + b;
}
```



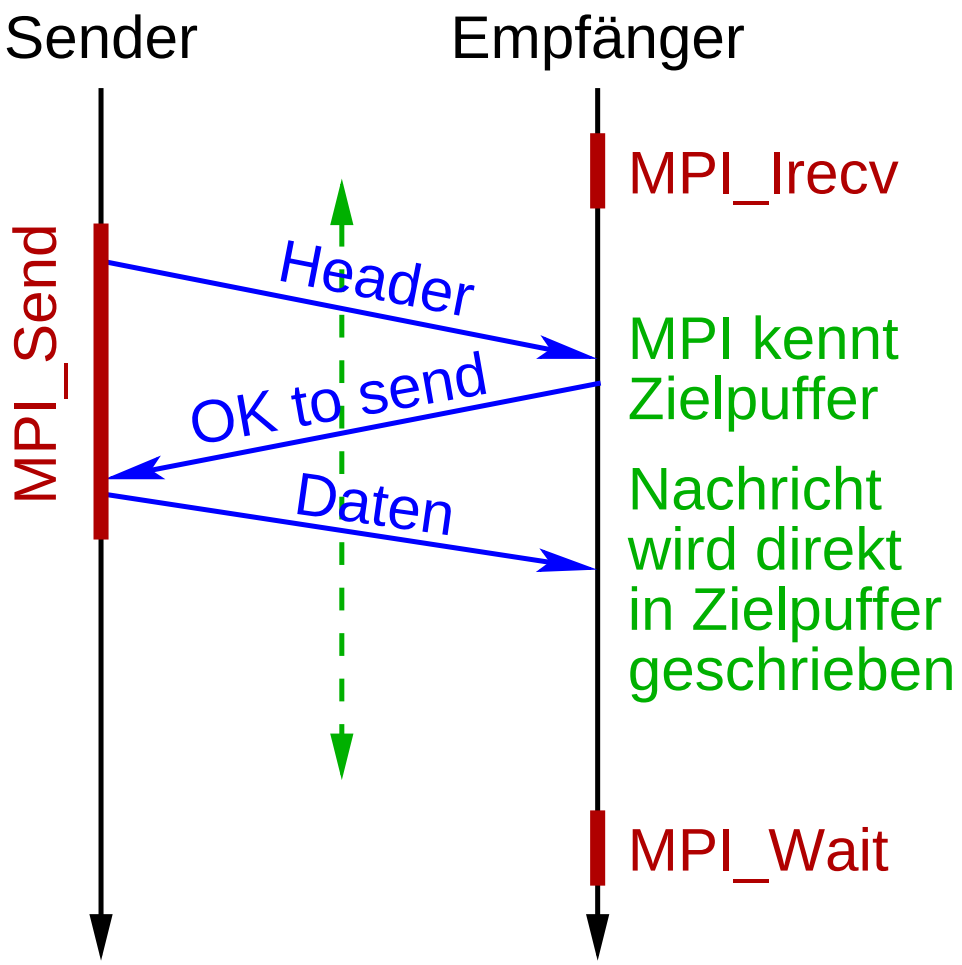
```
for (i=0; i<N; i++) {
  b[i] = f(..., i);
}
send(b, N, P2);
```

```
recv(b, N, P1);
for (i=0; i<N; i++) {
  a[i] = a[i] + b[i];
}
```

- Senden nach hinten aus der Schleife ziehen, Empfangen nach vorne
- Voraussetzung: Variablen werden in der Schleife nicht verändert (Sendeprozeß) bzw. benutzt (Empfängerprozeß)

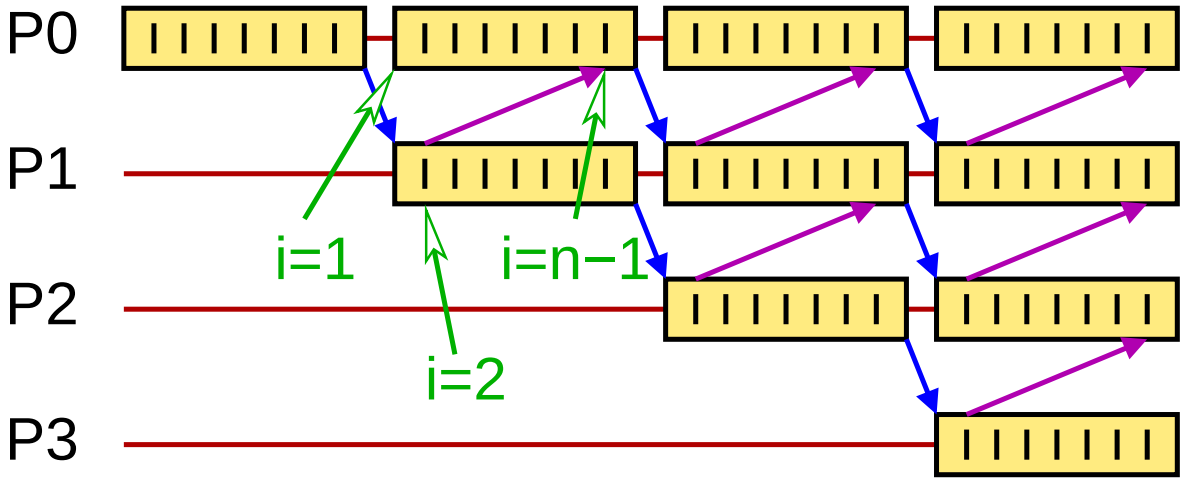
Latency Hiding

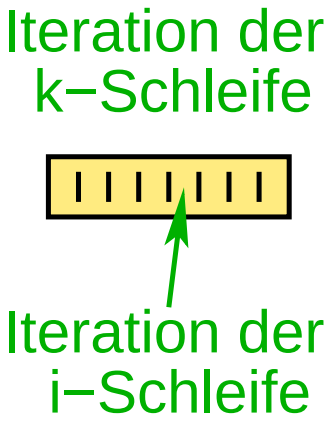
- Ziel: Kommunikations-latenz verstecken, d.h. mit Berechnungen überlappen
- Möglichst früh:
  - Empfang starten (MPI_Irecv)
- Dann:
  - Daten senden
- Möglichst spät:
  - Empfang abschließen (MPI_Wait)



Gauss-Seidel mit MPICH (Version 1) auf Intel Pentium 4

➥ Vorgesehener Zeitablauf des parallelen Programms:

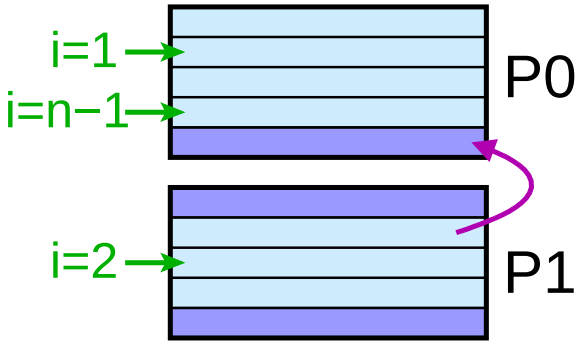




Ablauf der Kommunikation → :

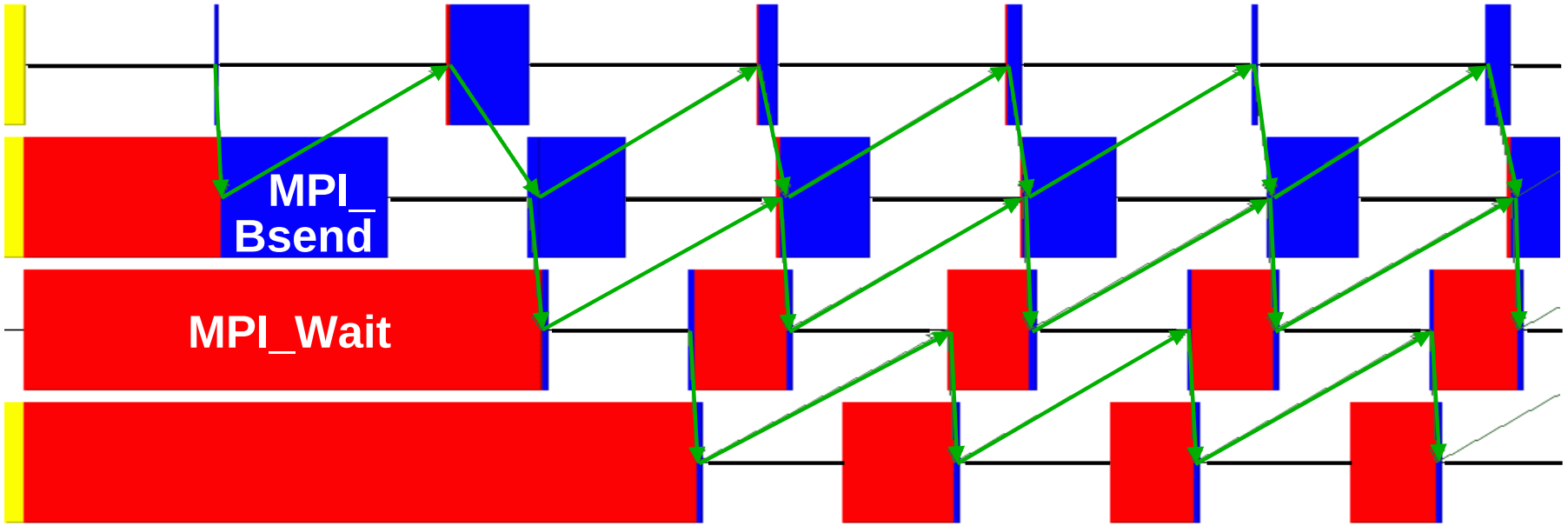
i=1: MPI_Irecv()
i=2: MPI_Bsend()
i=n−1: MPI_Wait()

} jeweils am Anfang der i−Schleife



Gauss-Seidel mit MPICH (Version 1) auf Intel Pentium 4 ...

- Tatsächliches Zeitverhalten (Jumpshot):



- Speedup nur 2.7 (4 Proz, 4000x4000 Matrix, Laufzeit: 12.3s)
- MPI_Bsend (gepuffertes Senden) blockiert! Warum?

Kommunikation in MPICH-p4

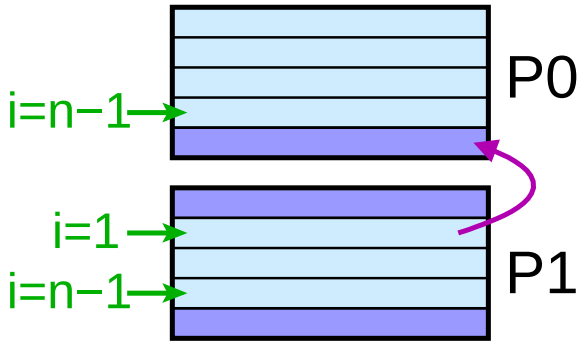
- Die MPI-Implementierung MPICH-p4 basiert auf TCP-IP
- MPICH-p4 holt die Nachrichten aus dem TCP-Puffer des Betriebssystems und kopiert sie in den Empfangspuffer des Prozesses
- Die MPI-Bibliothek kann das jedoch nur, wenn der Prozess regelmäßig (beliebige) MPI-Funktionen aufruft
  - bei Gauss-Seidel ist das während der Berechnungsphase aber der Fall
- Wenn der TCP-Empfangs-Puffer nicht geleert wird:
  - TCP-Puffer wird voll
  - TCP-Flußkontrolle blockiert den Sender-Prozeß

Gauss-Seidel: Verbesserungen

- Um Fortschritt der Kommunikation zu sichern:
  - Einstreuen von MPI_Test Aufrufen in die Berechnung
  - verbessert Laufzeit auf 11.8s, Speedup auf 2.85
  - Problem: Overhead durch die MPI_Test Aufrufe
- Anderer Ansatz: eng synchronisierte Kommunikation

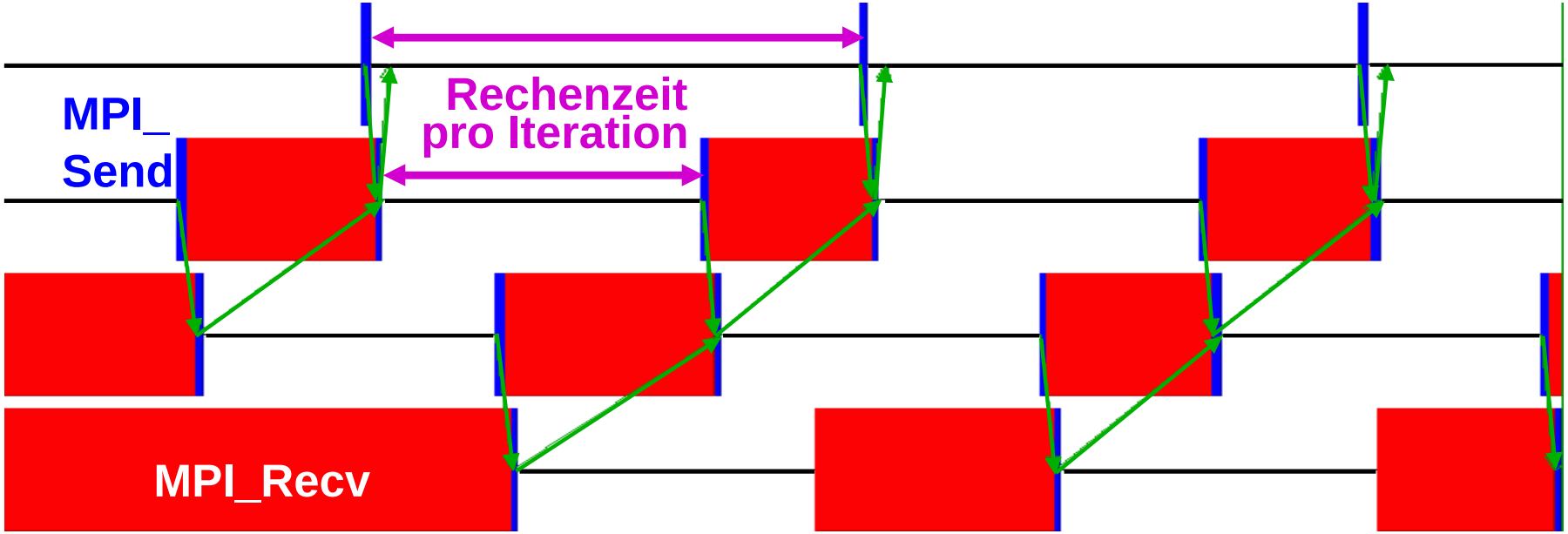
i=n−1: MPI_Send()
i=n−1: MPI_Recv()

} jeweils am Anfang der i−Schleife



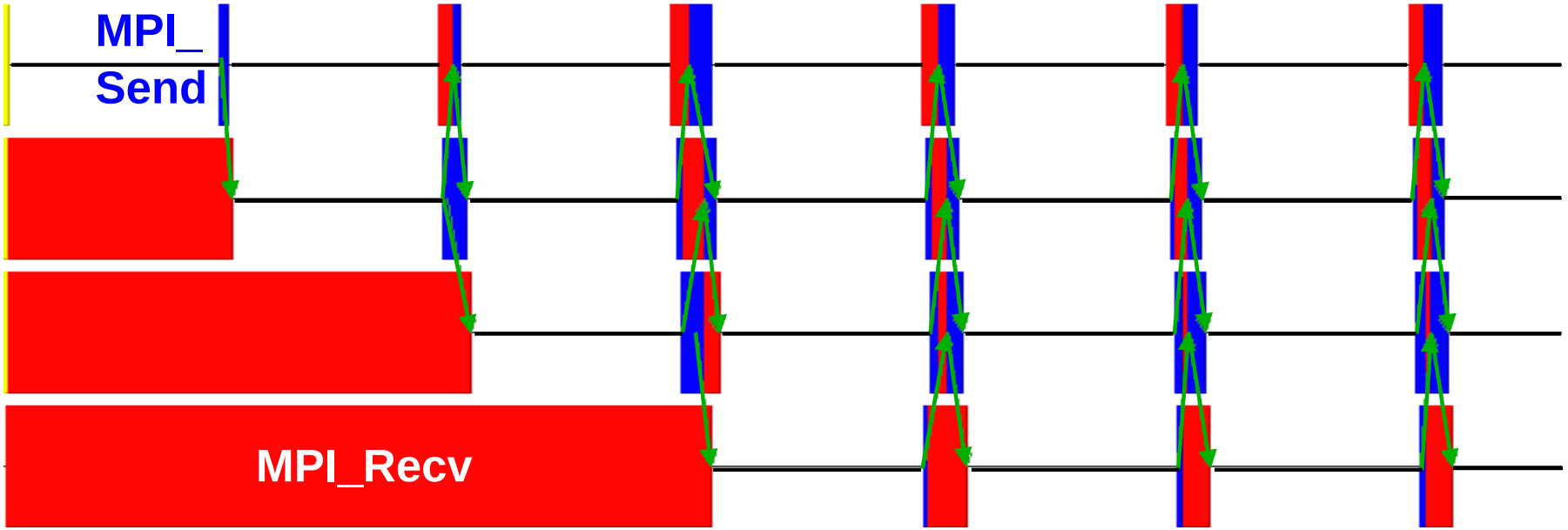
  - Laufzeit: 11.6s, Speedup 2.9
  - Nachteil: reagiert sehr sensibel auf Verzögerungen, z.B. Hintergrundlast auf den Knoten, Netzwerklast

Gauss-Seidel: Ergebnis



- Lastungleichheit trotz gleichmäßiger Aufteilung der Matrix!
  - Grund: Arithmetik des Pentium 4 extrem langsam bei denormalisierten Zahlen (Bereich $10^{-308} - 10^{-323}$)
  - z.B. Addition von 10^9 Zahlen: 220 s statt 9 s!

Gauss-Seidel: Erfolg!



- Initialisierung der Matrix mit 10^{-300} statt mit 0 beseitigt das Problem
- Laufzeit: 7.0 s, Speedup: 3.4
- Sequentielle Laufzeit nur noch 23.8 s statt 33.6 s

Gelernte Lektionen:

- *Latency Hiding* funktioniert nur dann vernünftig, wenn der Fortschritt der Kommunikation sichergestellt ist
  - z.B. bei MPI über Myrinet: Netzwerkadapter schreibt ankommende Daten direkt in Empfangspuffer des Prozesses
  - oder mit MPICH2 (eigener Thread)
- Eng synchronisierte Kommunikation kann besser sein, ist aber anfällig gegenüber Verzögerungen
- Lastungleichgewicht kann auch eintreten, wenn man es nicht erwartet
  - Ausführungszeiten auf modernen Prozessoren sind unberechenbar (im Wortsinne!)

4.4 Zusammenfassung

- Lokalität (Caches) berücksichtigen!
  - Matrizen in Reihenfolge der Speicherung durchlaufen
  - Zweierpotenzen als Schrittweite im Speicher vermeiden
  - Block-Algorithmen verwenden
- *False Sharing* vermeiden!
- Nachrichten möglichst zusammenfassen!
- *Latency Hiding* verwenden, wenn Kommunikationsbibliothek Nachrichtenempfang „im Hintergrund“ durchführen kann
- Falls Sende-Aufrufe blockieren: Senden und Empfangen möglichst synchron duchführen

Parallelverarbeitung

WS 2015/16

5 Anhang

Parallelverarbeitung

WS 2015/16

05.11.2015

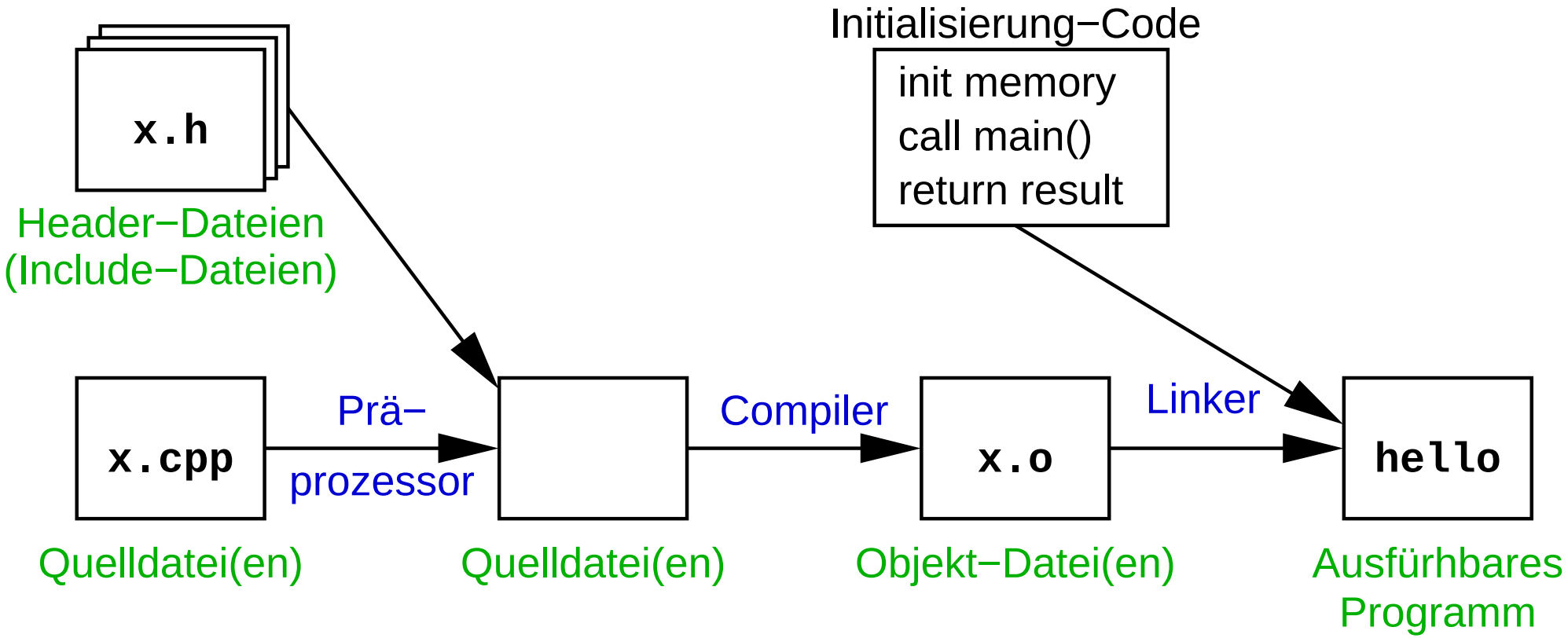
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

5.1.1 Grundsätzliches zu C++

- Gemeinsamkeiten zwischen C++ und Java:
  - imperative Programmiersprache
  - Syntax weitgehend identisch
- Unterschiede zwischen C++ und Java:
  - C++ ist nicht rein objektorientiert
  - C++-Programme werden direkt in Maschinencode übersetzt (keine virtuelle Maschine)
- Übliche Dateistruktur von C++-Programmen:
  - Header-Dateien (`*.h`) enthalten Deklarationen
    - Typen, Klassen, Konstante, ...
  - Quelldateien (`*.cpp`) enthalten Implementierungen
    - Methoden, Funktionen, globale Variable

Übersetzung von C++-Programmen



- Präprozessor: Einbinden von Dateien, Ersetzen von Makros
- Linker: fügt Objekt-Dateien und Bibliotheken zusammen

Übersetzung von C++-Programmen ...

- Aufruf des Compilers im Labor H-A 4111:
  - `g++ -Wall -o <Ausgabe-Datei> <Quelldateien>`
  - führt Präprozessor, Compiler und Linker aus
  - `-Wall`: alle Warnungen ausgeben
  - `-o <Ausgabe-Datei>`: Name der ausführbaren Datei
- Weitere Optionen:
  - `-g`: Quellcode-Debugging ermöglichen
  - `-O`: Code-Optimierungen erlauben
  - `-l<Bibliothek>`: genannte Bibliothek einbinden
  - `-c`: Linker nicht ausführen
    - später: `g++ -o <Ausgabe-Datei> <Objektdateien>`

Ein Beispiel: *Hello World*! (☞ 05/hello.cpp)

```
#include <iostream>
```
Präprozessor−Befehl: fügt Inhalt der Datei iostream ein (u.a. Deklaration von cout)

```
using namespace std;
```
Namensraum std verwenden

```
void sayHello()
```
Funktionsdefinition

```
{
  cout << "Hello World\n";
}
```
Ausgabe eines Textes

```
int main()
```
Hauptprogramm

```
{
  sayHello();
  return 0;
}
```
Rückkehr aus dem Programm: 0 = alles OK, 1,2,...,255: Fehler

- Übersetzung: `g++ -Wall -o hello hello.cpp`
- Start: `./hello`

Syntax

- Identisch zu Java sind u.a.:
  - Deklaration von Variablen und Parametern
  - Methodenaufrufe
  - Kontrollanweisungen (`if`, `while`, `for`, `case`, `return`, ...)
  - einfache Datentypen (`short`, `int`, `double`, `char`, `void`, ...)
    - Abweichungen: `bool` statt `boolean`; `char` ist 1 Byte groß
  - praktisch alle Operatoren (`+`, `*`, `%`, `<<`, `==`, `?:`, ...)
- Sehr ähnlich zu Java sind:
  - Arrays
  - Klassendeklarationen

Arrays

- Deklaration von Arrays
  - nur mit fester Größe, z.B.:

```
int ary1[10];                 // int—Array mit 10 Elementen
double ary2[100][200];        // 100 * 200 Array
int ary3[] = { 1, 2 };        // int—Array mit 2 Elementen
```

  - bei Parametern: Größe kann bei **erster** Dimension entfallen

```
int funct(int ary1[], double ary2[][200]) { ... }
```

- Arrays sind auch über Zeiger realisierbar (siehe später)
  - dann auch dynamisches Erzeugen möglich
- Zugriff auf Arrays
  - wie in Java, z.B.: `a[i][j] = b[i] * c[i+1][j];`
  - aber: **keine** Prüfung der Arraygrenzen!!!

Klassen und Objekte

- Deklaration von Klassen (typisch in `.h`-Datei):

```
class Example {
 private:              // Private Attribute/Methoden
   int attr1;                      // Attribut
   void pmeth(double d);           // Methode
 public:               // Öffentliche Attribute/Methoden
   Example();                      // Default-Konstruktur
   Example(int i);                 // Konstruktur
   Example(Example &from);         // Copy-Konstruktur
   ~Example();                     // Destruktor
   int meth();                     // Methode
   int attr2;                      // Attribut
   static int sattr;               // Klassenattribut
};
```

Klassen und Objekte ...

- Definition von Klassenattributen und Methoden (`*.cpp`-Datei):

```
int Example::sattr = 123;  // Klassenattribut

Example::Example(int i) {  // Konstruktor
  this->attr1 = i;
}
int Example::meth() {      // Medhode
  return attr1;
}
```

  - Angabe des Klassennamens bei Attributen und Methoden
    - Trennzeichen `::` statt .
  - `this` ist ein Zeiger (☞ **5.1.3**), daher `this->attr1`
  - Methodenrümpfe können alternativ auch in der Klassendefinition selbst angegeben werden

Klassen und Objekte ...

- Deklaration von Objekten:

```
{
   Example ex1;        // Initialisierung mit Default—Konstruktor
   Example ex2(10);    // Konstruktor mit Argument
   ...
} // Jetzt wird der Destruktor für ex1, ex2 aufgerufen
```

- Zugriff auf Attribute, Aufruf von Methoden

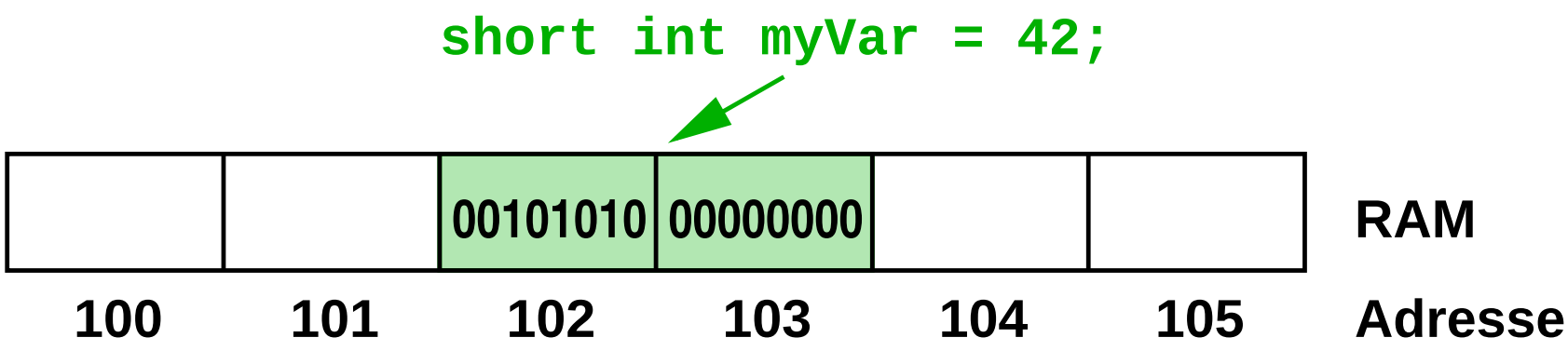
```
ex1.attr2 = ex2.meth();
j = Example::sattr;        // Klassenattribut
```

- Zuweisung / Kopieren von Objekten:

```
ex1 = ex2;              // Objekt wird kopiert!
Example ex3(ex2);       // Initialisierung durch Copy—Konstruktor
```

Variablen im Speicher

- Erinnerung: Variablen werden im Hauptspeicher abgelegt

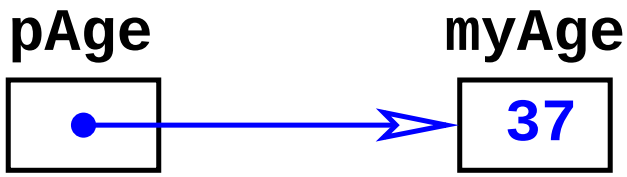


  - eine Variable gibt einem Speicherbereich einen Namen und einen Typ
    - hier: `myVar` belegt 2 Bytes (`short int`) ab Adresse 102
- Ein **Zeiger** ist eine Speicheradresse, verbunden mit einem Typ
  - Typ legt fest, wie der Speicherbereich interpretiert wird

Deklaration und Verwendung von Zeigern

- Beispiel:

```
int myAge = 25;      // eine int-Variable
int *pAge;           // ein Zeiger auf int-Werte
pAge = &myAge;       // pAge zeigt jetzt auf myAge
*pAge = 37;          // myAge hat jetzt den Wert 37
```



- Der **Adreßoperator** `&` bestimmt die Adresse einer Variablen
- Der Zugriff auf `*pAge` heißt **Dereferenzieren** von `pAge`
- Zeiger haben (fast) immer einen Typ
  - z.B. `int *`, `Example *`, `char **`, ...

Call by reference-Übergabe von Parametern

- Zeiger ermöglichen eine Parameterübergabe *by reference*
- Statt eines Werts wird ein **Zeiger** auf den Wert übergeben:

```
void byReference(Example *e, int *result) {
    *result = e->attr2;
}
int main() {
    Example obj(15);                // obj wird effizienter per
    int res;                        //    Referenz übergeben
    byReference(&obj, &res);        // res ist Ergebnisparameter
    ...
```

  - abkürzende Schreibweise: `e->attr2` bedeutet `(*e).attr2`

`void`-Zeiger und Typkonvertierung

- C++ erlaubt auch die Verwendung von generischen Zeigern
  - lediglich eine Speicheradresse ohne Typinformation
  - deklarierter Typ ist `void *` (Zeiger auf `void`)
- Dereferenzierung erst nach Typkonvertierung möglich
  - Achtung: keine Typsicherheit / Typprüfung!
- Anwendung häufig für generischer Parameter von Funktionen:

```
void bsp(int type, void *arg) {
   if (type == 1) {
      double d = *(double *)arg;   // arg erst in double *
      ...                          //   umwandeln
   } else {
      int i = *(int *)arg;         // int-Argument
```

Arrays und Zeiger

- C++ macht (außer bei der Deklaration) keinen Unterschied zwischen eindimensionalen Arrays und Zeigern
- Konsequenzen:
  - Array-Variable können wie (konstante) Zeiger verwendet werden
  - Zeiger-Variable können indiziert werden

```
int a[3] = { 1, 2, 3 };
int b = *a;                 // entspricht: b = a[0]
int c = *(a+1);             // entspricht: c = a[1]
int *p = a;                 // entspricht: int *p = &a[0]
int d = p[2];               // d = a[2]
```

Arrays und Zeiger ...

- Konsequenzen ...:
  - Parameterübergabe bei Arrays erfolgt immer *by reference*!

```
void swap(int a[], int i, int j) {
    int h = a[i];    // Vertausche a[i] und a[j]
    a[i] = a[j];
    a[j] = h;
}
int main() {
    int ary[] = { 1, 2, 3, 4 };
    swap(ary, 1, 3);
    // jetzt:  ary[1] = 4,  ary[3] = 2;
}
```

Dynamische Speicherverwaltung

- Allokieren von Objekten und Arrays wie in Java

```
Example *p = new Example(10);
int *a = new int[10];              // a wird nicht initialisiert!
int *b = new int[10]();            // b wird (mit 0) initialisiert
```

  - Allokieren mehrdimensionaler Arrays geht so jedoch nicht

- Wichtig: C++ hat keine *Garbage Collection*
  - daher explizite Speicherfreigabe notwendig:

```
delete p;     // Einzelobjekt
delete[] a;   // Array
```

  - Achtung: Speicher nicht mehrfach freigeben!

Funktionszeiger

- Zeiger können auch auf Funktionen zeigen:

```
void myFunct(int arg) { ... }
void test1() {
    void (*ptr)(int) = myFunct; // Funktionszeiger + Init.
    (*ptr)(10);                 // Funktionsaufruf über Zeiger
```

- Damit können Funktionen z.B. als Parameter an andere Funktionen übergeben werden:

```
void callIt(void (*f)(int)) {
    (*f)(123);        // Aufruf der übergebenen Funktion
}
void test2() {
  callIt(myFunct); // Funktion als (Referenz—)Parameter
```

5.1.4 Strings und Ausgabe

- C++ besitzt wie Java eine String-Klasse (`string`)
  - alternativ wird z.T. auch der Typ `char *` verwendet
- Zur Konsolenausgabe gibt es die Objekte `cout` und `cerr`
- Beides existiert im Namensraum (Paket) `std`
  - zur Verwendung ohne Namenspräfix:

```
using namespace std; // entspricht 'import std.*;' in Java
```

- Beispiel einer Ausgabe:

```
double x = 3.14;
cout << "Pi ist ungefähr " << x << "\n";
```

- Spezielle Formatierungsfunktionen für Zahlenausgabe, z.B.:

```
cout << setw(8) << fixed << setprecision(4) << x << "\n";
```

  - Ausgabe mit Feldlänge 8 und genau 4 Nachkommastellen

5.1.5 Weitere Besonderheiten von C++

- **Globale** Variablen
  - außerhalb einer Funktion bzw. Methode deklariert
  - leben während der gesamten Programmausführung
  - sind von allen Funktionen aus zugreifbar
- Globale Variablen und Funktionen können erst **nach** der Deklaration verwendet werden
  - für Funktionen daher: **Funktionsprototypen**

```
int funcB(int n);          // Funktionsprototyp
int funcA() {              // Funktionsdefinition
    return funcB(10);
}
int funcB(int n) {         // Funktionsdefinition
    return n * n;
}
```

5.1.5 Weitere Besonderheiten von C++ ...

- Schlüsselwort `static` vor der Deklaration globaler Variablen oder Funktionen

```
static int number;
static void output(char *str) { ... }
```

  - bewirkt, daß Variable/Funktion nur in der lokalen Quelldatei verwendet werden kann

- Schlüsselwort `const` vor der Deklaration von Variablen oder Parametern

```
const double PI = 3.14159265;
void print(const char *str) { ... }
```

  - bewirkt, daß die Variablen nicht verändert werden können
  - in etwa analog zu `final` in Java
  - (Hinweis: diese Beschreibung ist sehr stark vereinfacht!)

5.1.5 Weitere Besonderheiten von C++ ...

- Übergabe von Kommandozeilen-Argumenten:

```
int main(int argc, char **argv) {
    if (argc > 1)
        cout << "1. Argument: " << argv[1] << "\n";
}
```

Aufrufbeispiel:
```
bslab1% ./myprog -p arg2
1. Argument: -p
```

- `argc` ist Anzahl der Argumente (incl. Programmname)
- `argv` ist ein Array (der Länge `argc`) von Strings (`char *`)
- im Beispiel:
```
argv[0] = "./myprog"
argv[1] = "-p"
argv[2] = "arg2"
```
- wichtig: Prüfung des Index gegen `argc`

Übersicht

- Es gibt etliche (Standard-)Bibliotheken für C/C++, zu denen immer ein oder mehrere Header-Dateien gehören, z.B.:

Header-Datei	Bibliothek (g++-Option)	Beschreibung	stellt z.B. bereit
iostream		Ein-/Ausabe	cout, cerr
string		C++ Strings	string
stdlib.h		Standard-Fkt.	exit()
sys/time.h		Fkt. für Zeit	gettimeofday()
math.h	-lm	Math. Funktionen	sin(), cos(), fabs()
pthread.h	-lpthread	Threads	pthread_create()
mpi.h	-lmpich	MPI	MPI_Init()

Funktionen des Präprozessors:

- Einbinden von Header-Dateien

```
#include <stdio.h>   // sucht nur in Systemverzeichnissen
#include "myhdr.h"   // sucht auch im aktuellen Verzeichnis
```

- Ersetzen von Makros

```
#define BUFSIZE   100            // Konstante
#define VERYBAD   i + 1;         // extrem schlecher Stil !!
#define GOOD      (BUFSIZE+1)    // Klammern sind wichtig!
...
  int i = BUFSIZE;      // wird zu int i = 100;
  int a = 2*VERYBAD     // wird zu int a = 2*i + 1;
  int b = 2*GOOD;       // wird zu int a = 2*(100+1);
```

Funktionen des Präprozessors: ...

- Bedingte Übersetzung (z.B. für Debug-Ausgaben)

```
int main() {
#ifdef DEBUG
    cout << "Programm ist gestartet\n";
#endif
    ...
}
```

  - Ausgabeanweisung wird normalerweise nicht übersetzt
  - zur Aktivierung:
    - entweder `#define DEBUG` am Programmanfang
    - oder Übersetzen mit `g++ -DDEBUG ...`

Parallelverarbeitung

WS 2015/16

30.11.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

5.2.1 Übersetzung und Ausführung

- Übersetzung: mit gcc (g++)
  - typ. Aufruf: `g++ -fopenmp myProg.cpp -o myProg'`
  - OpenMP 3.0 ab gcc 4.4, OpenMP 4.0 ab gcc 4.9
- Ausführung: Start wie bei sequentiellem Programm
  - z.B.: `./myProg`
  - (maximale) Anzahl der Threads muß vorher über Umgebungsvariable `OMP_NUM_THREADS` festgelegt werden
    - z.B.: `export OMP_NUM_THREADS=4`
    - gilt dann für alle Programme im selben Kommandofenster
  - Auch möglich: temporäres (Um-)Setzen von `OMP_NUM_THREADS`
    - z.B.: `OMP_NUM_THREADS=2 ./myProg`

5.2.2 Debugging

- Es gibt nur wenige Debugger, die OpenMP voll unterstützen
  - z.B. Totalview
  - erfordert enge Zusammenarbeit zw. Compiler und Debugger
- Auf den PCs im Labor H-A 4111:
  - `g++`/`ddd` erlauben halbwegs vernünftiges Debugging
    - unterstützen mehrere Threads
  - `gdb`: textueller Debugger (Standard LINUX debugger)
  - `ddd`: graphisches Front-End für `gdb`
    - komfortabler, aber „schwergewichtiger“
- Auf dem HorUS Cluster: `totalview`
  - graphischer Debugger
  - unterstützt OpenMP und MPI

5.2.2 Debugging ...

- Voraussetzung: Übersetzung mit Debugging-Information
  - sequentiell: `g++ -g -o myProg myProg.cpp`
  - mit OpenMP: `g++ -g -fopenmp ...`
- Debugging ist auch eingeschränkt(!) in Verbindung mit Optimierung möglich
  - teilweise jedoch unerwartetes Verhalten des Debuggers
  - falls möglich: Optimierungen abschalten
    - `g++ -g -O0 ...`

Wichtige Funktionen eines Debuggers (Beispiele für `gdb`):

- Starten des Programms: `run arg1 arg2`
- Setzen von Haltepunkten auf Code-Zeilen: `break file.cpp:35`
- Setzen von Haltepunkten auf Funktionen: `break myFunc`
- Ausgabe des Prozeduraufruf-Kellers: `where`
- Navigation im Prozeduraufruf-Keller: `up` bzw. `down`
- Ausgabe von Variableninhalten: `print i`
- Ändern von Variablen: `set variable i=i*15`
- Weiterführen des Programms (nach Haltepunkt): `continue`
- Einzelschitt-Abarbeitung: `step` bzw. `next`

Wichtige Funktionen eines Debuggers (Beispiele für `gdb`): ...

- Anzeige aller Threads: `info threads`
- Auswahl eines Threads: `thread 2`
  - Kommandos wirken i.a. nur auf den ausgewählten Thread
- Quellcode-Listing: `list`
- Hilfe: `help`
- Beenden: `quit`

- Alle Kommandos können im `gdb` auch abgekürzt werden

Beispielsitzung mit gdb (sequentiell)

```
bsclk01> g++ -g -O0 -o ross ross.cpp   ← Option -g für Debugging
bsclk01> gdb ./ross
GNU gdb 6.6
Copyright 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public ...
(gdb) b main   ← Haltepunkt auf Prozedur main setzen
Breakpoint 1 at 0x400d00: file ross.cpp, line 289.
(gdb) run 5 5 0   ← Programm mit diesen Argumenten starten
Starting program: /home/wismueller/LEHRE/pv/ross 5 5 0
Breakpoint 1, main (argc=4, argv=0x7fff0a131488) at ross.cpp:289
289     if (argc != 4) {
(gdb) list   ← Listing um die aktuelle Zeile
284
285     /*
286     ** Get and check the command line arguments
```

5.2.2 Debugging ...

```
287     */
288
289     if (argc != 4) {
290         cerr << "Usage: ross <size_x> <size_y> ...
291         cerr << "       <size_x> <size_y>: size...
292         cerr << "       <all>: 0 = compute one ...
293         cerr << "              1 = compute all ...
(gdb) b 315   ← Haltepunkt auf Zeile 35 setzen
Breakpoint 2 at 0x400e59: file ross.cpp, line 315.
(gdb) c   ← Programm fortführen
Continuing.
Breakpoint 2, main (argc=4, argv=0x7fff0a131488) at ross.cpp:315
315         num_moves = Find_Route(size_x, size_y, moves);
(gdb) n   ← Nächste Programmzeile (hier: 315) ausführen
320         if (num_moves >= 0) {
(gdb) p num_moves   ← Variable num_moves ausgeben
$1 = 24
```

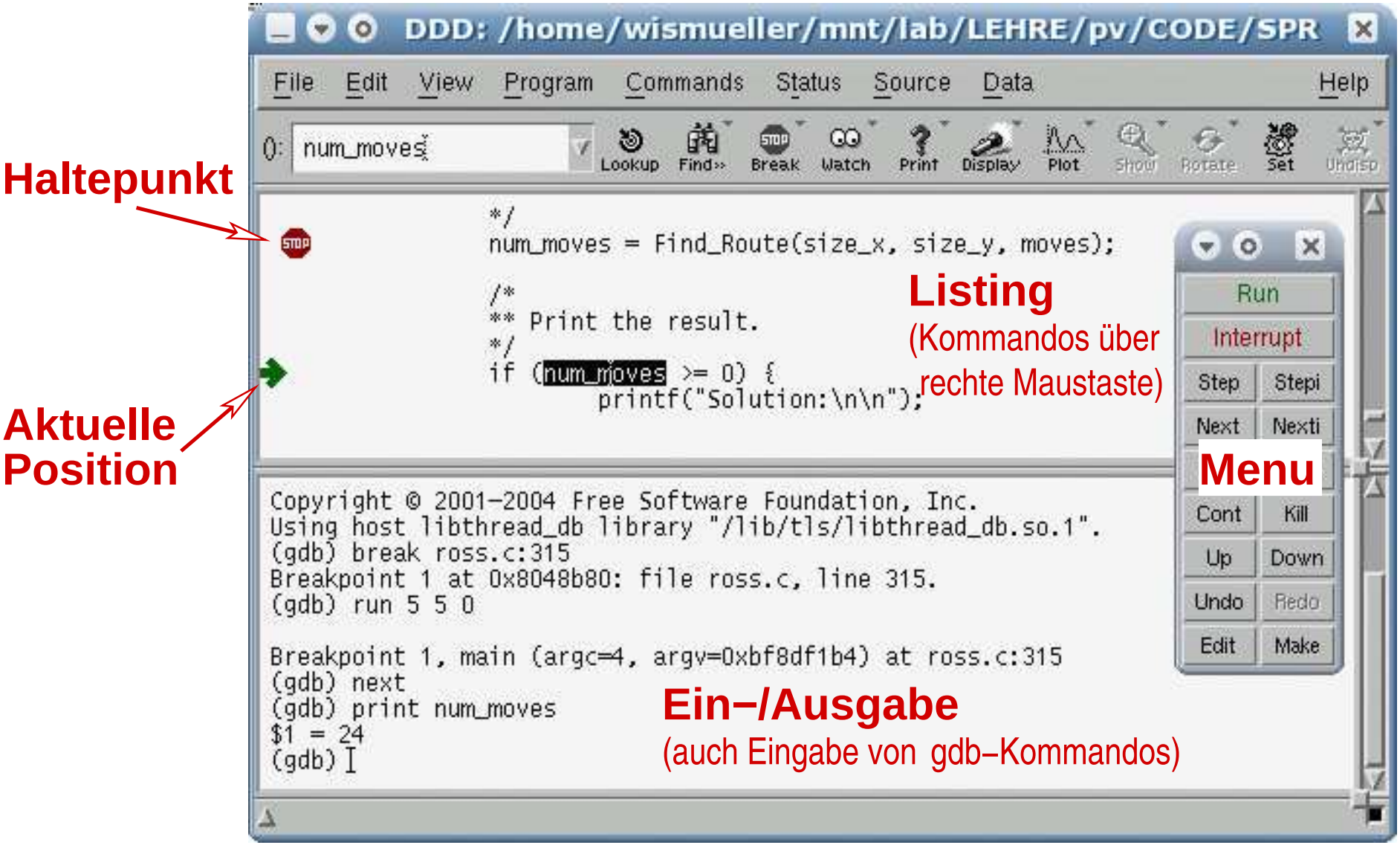
5.2.2 Debugging ...

```
(gdb) where   ← Wo steht das Programm gerade?
#0  main (argc=4, argv=0x7fff0a131488) at ross.cpp:320
(gdb) c   ← Programm fortführen
Continuing.
Solution:
  ...
Program exited normally.
(gdb) q   ← gdb beenden
bsclk01>
```

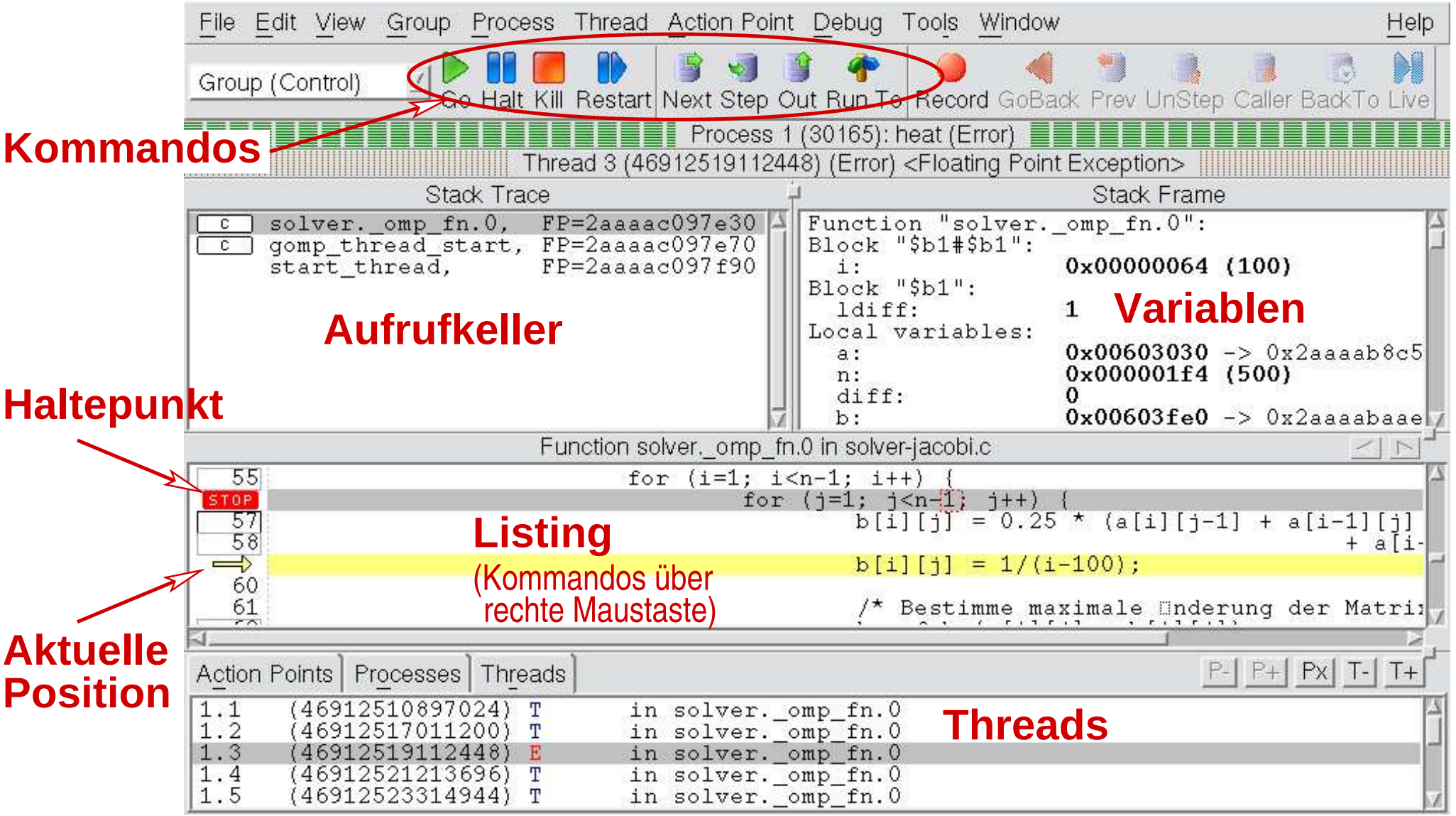
Beispielsitzung mit gdb (OpenMP)

```
bslab03> g++ -fopenmp -O0 -g -o heat heat.cpp solver-jacobi.cpp
bslab03> gdb ./heat
GNU gdb (GDB) SUSE (7.5.1-2.1.1)
...
(gdb) run 500
...
Program received signal SIGFPE, Arithmetic exception.
0x0000000000401711 in solver._omp_fn.0 () at solver-jacobi.cpp:58
58                                  b[i][j] = i/(i-100);
(gdb) info threads
  Id   Target Id         Frame
  4    Thread ... (LWP 6429) ... in ... at solver-jacobi.cpp:59
  3    Thread ... (LWP 6428) ... in ... at solver-jacobi.cpp:59
  2    Thread ... (LWP 6427) ... in ... at solver-jacobi.cpp:63
* 1    Thread ... (LWP 6423) ... in ... at solver-jacobi.cpp:58
(gdb) q
```

Beispielsitzung mit ddd



Beispielsitzung mit `totalview`



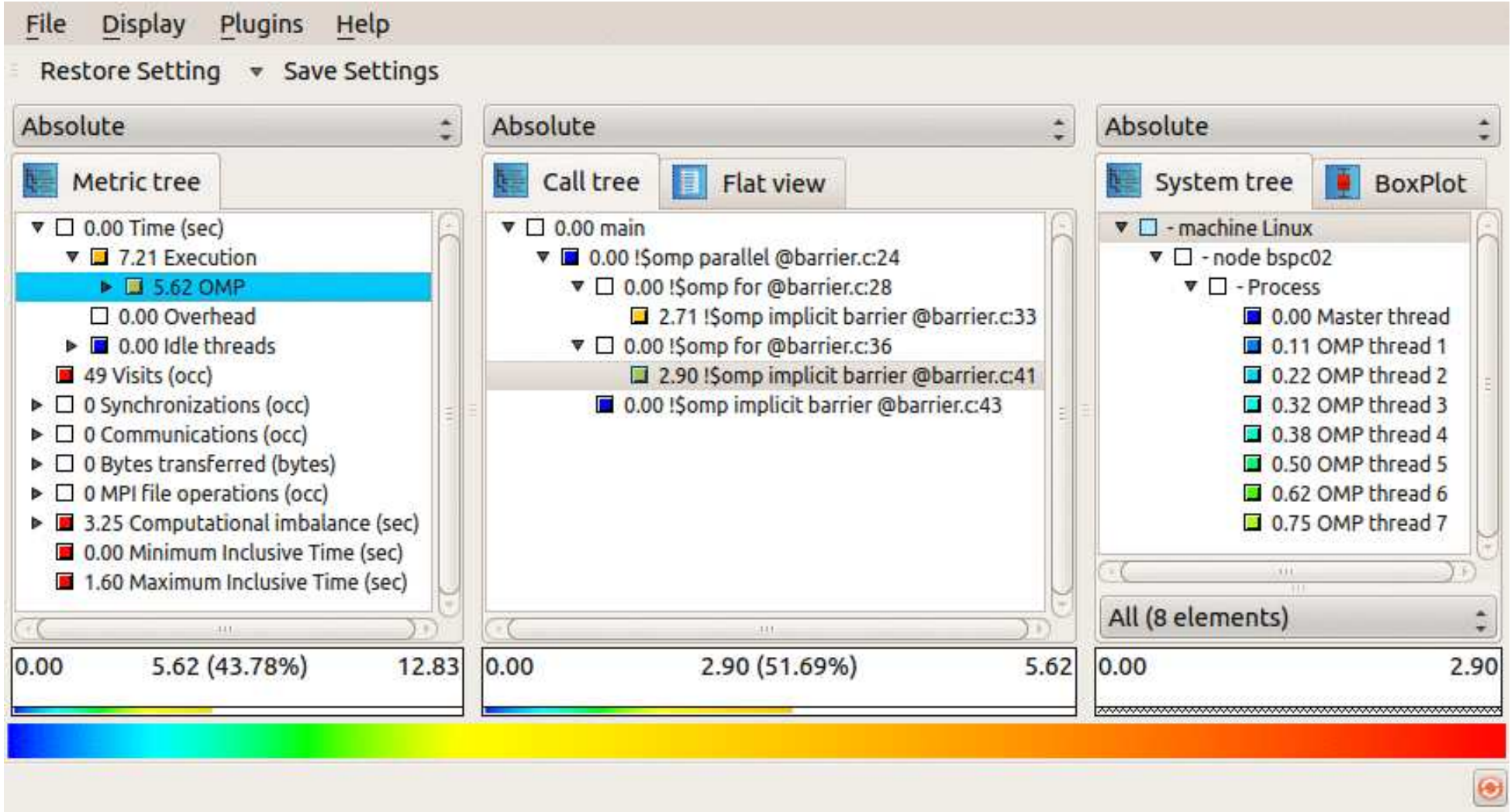
5.2.3 Leistungsanalyse

- Typisch: **Instrumentierung** des erzeugten Programmcodes bei/nach der Übersetzung
  - Einfügen von Code an wichtigen Stellen des Programms
    - zur Erfassung relevanter Ereignisse
    - z.B. Beginn/Ende von parallelen Regionen, Barrieren, ...
  - Während der Ausführung werden dann die Ereignisse
    - einzeln in eine **Spurdatei** (***Trace file***) protokolliert
    - oder bereits zu einem *Profile* zusammengefasst
  - Auswertung erfolgt nach Programmende
  - vgl. Abschnitt 1.9.6
- Im H-A 4111 und auf HorUS-Cluster: Scalasca

Leistungsanalyse mit Scalasca

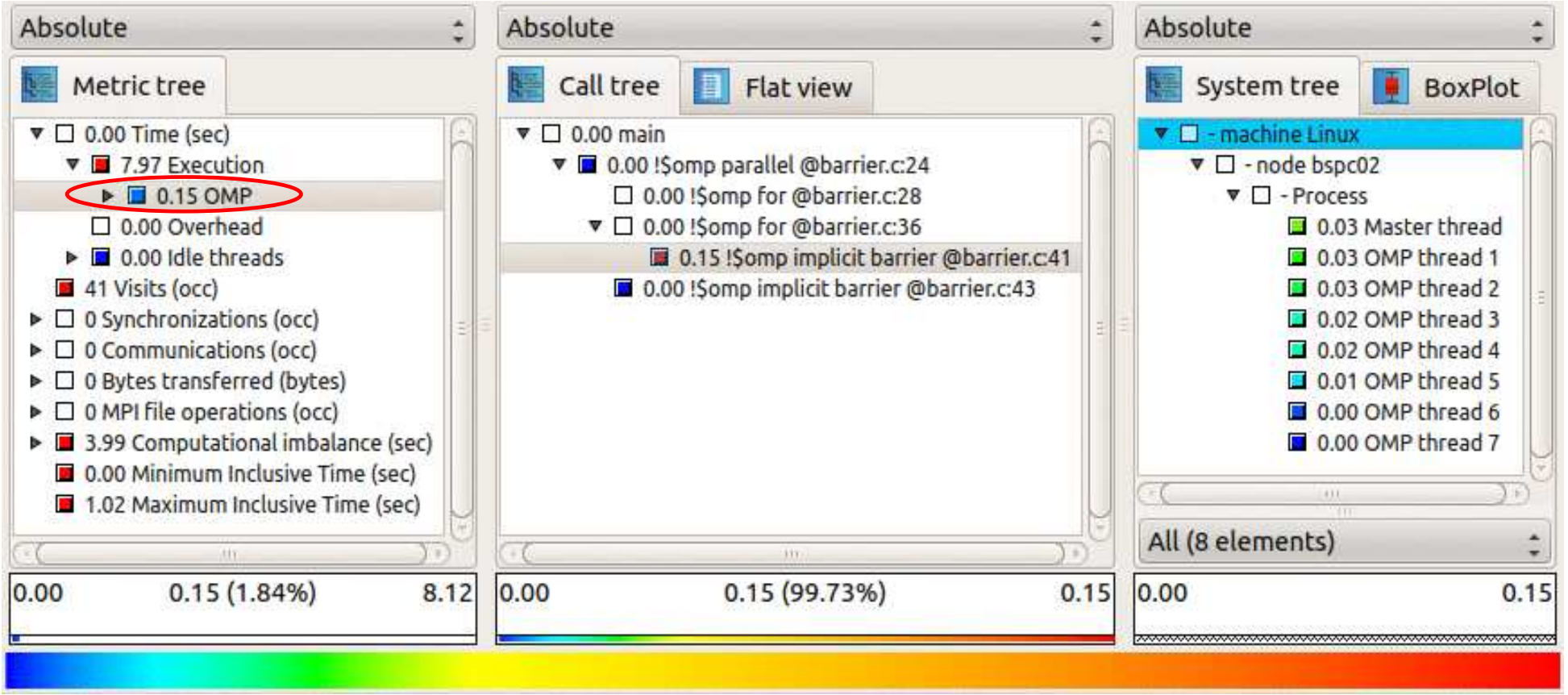
- Pfade setzen (im H-A 4111; HorUS: siehe 5.2.4)
  - `export PATH=/opt/dist/scorep-1.4.1/bin:\`
    `/opt/dist/scalasca-2.2.1/bin:$PATH`
- Übersetzen des Programms:
  - `scalasca -instrument g++ -fopenmp ... barrier.cpp`
- Ausführen des Programms:
  - `scalasca -analyze ./barrrier`
  - legt Daten in einem Verzeichnis `scorep_barrier_0x0_sum` ab
    - `0x0` zeigt Thread-Anzahl an (0 = Default)
    - Verzeichnis darf noch nicht existieren, ggf. löschen
- Interaktive Analyse der aufgezeichneten Daten:
  - `scalasca -examine scorep_barrier_0x0_sum`

Leistungsanalyse mit Scalasca: Beispiel von Folie 217



Leistungsanalyse mit Scalasca: Beispiel von Folie 217 ...

- Im Beispiel kann durch die Option `nowait` bei der ersten Schleife die Wartezeit in Barrieren drastisch reduziert werden:



Architektur des HorUS-Clusters

- 34 Dell PowerEdge C6100 Systeme mit je 4 Knoten
- 136-Compute-Knoten
  - CPU: 2 x Intel Xeon X5650, 2.66 GHz, 6 Cores pro CPU, 12 MB Cache
  - Hauptspeicher: 48 GB (4GB je Core, 1333 MHz DRR3)
  - Festplatte: 500 GB SATA (7,2k RPM, 3,5 Zoll)
- Insgesamt: 272 CPUs, 1632 Cores, 6,4 TB RAM, 40 TB Platte
- Paralleles Dateisystem: 33 TB, 2.5 GB/s
- Infiniband-Netzwerk (40 GBit/s)
- Peak-Performance: 17 TFlop/s

5.2.4 Nutzung des HorUS-Clusters ...

Zugang

- Über SSH: `ssh -X` *`g-Kennung`*`@slc2.zimt.uni-siegen.de`
- Im Labor H-A 4111:
  - Weiterleitung der SSH-Verbindung durch Labor-Gateway
  - `ssh -X -p 22222` *`g-Kennung`*`@bslabgate1.lab.bvs`
  - Am besten Nutzung der Datei `$HOME/.ssh/config`:
    - ```
      Host horus
           user g-Kennung
           hostname bslabgate1.lab.bvs
           ForwardX11 yes
           HostKeyAlias horus
           port 22222
      ```
    - Dann einfach: `ssh horus`
      ```

Aufsetzen der SSH im H-A 4111

- Erzeugen eines SSH-Schlüssels:
  - `ssh-keygen -b 2048 -t rsa` (oder `-b 4096`)
  - bei Frage `"Enter file ..."` einfach `Return` drücken
  - sichere Passphrase für privaten Schlüssel wählen!
- Anfügen des öffentlichen Schlüssels an die Liste autorisierter Schlüssel:
  - `cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys`
- Damit ist auch ein Einloggen auf andere Laborrechner ohne dauernde Passworteingabe möglich

Aufsetzen der Umgebung auf dem HorUS

- SSH-Schlüssel auf den HorUS kopieren
  - `ssh-copy-id -i ~/.ssh/id_rsa.pub horus`
  - Wichtig: achten Sie darauf, Ihr Paßwort korrekt einzutippen!
- Auf dem HorUS benötigte Module definieren:

```
module load openmpi/gcc/64/1.8.1
module load gcc/4.8.4
module load scalasca/2.2
module load scorep/gcc-openmpi_4.8.4-1.8.1/1.4
export PATH=$PATH:/cm/shared/ZIMT/apps/cube/4.3/gcc/bin
```

für Scalasca

  - am besten an `~/.bashrc` anfügen
  - für OpenMP 4.0: Modul `gcc/5.1.0` statt `gcc/4.8.4` laden
    - Scalasca kann nicht mehr verwendet werden!

Zur praktischen Nutzung des HorUS

- Mounten des HorUS-Dateisystems auf die Laborrechner
  - Verzeichnis für Mount-Punkt anlegen: `mkdir ~/mnt`
  - Mounten des HorUS-Dateisystems: `sshfs horus: ~/mnt`
  - Unmount: `fusermount -u ~/mnt`
- Start von Programmen auf dem HorUS
  - Nutzung des Batch-Queueing-Systems SLURM
    - siehe https://computing.llnl.gov/linux/slurm
  - Start eines OpenMP-Programms, z.B.:
    - ```
      export OMP_NUM_THREADS=8
      salloc --exclusive --partition debug \
             $HOME/GAUSS/heat 500
      ```
      ```

Parallelverarbeitung

WS 2015/16

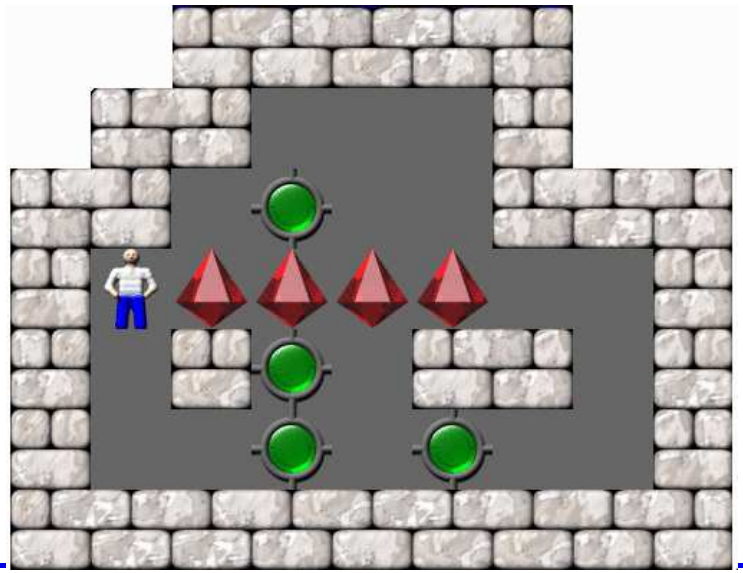
14.12.2015

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

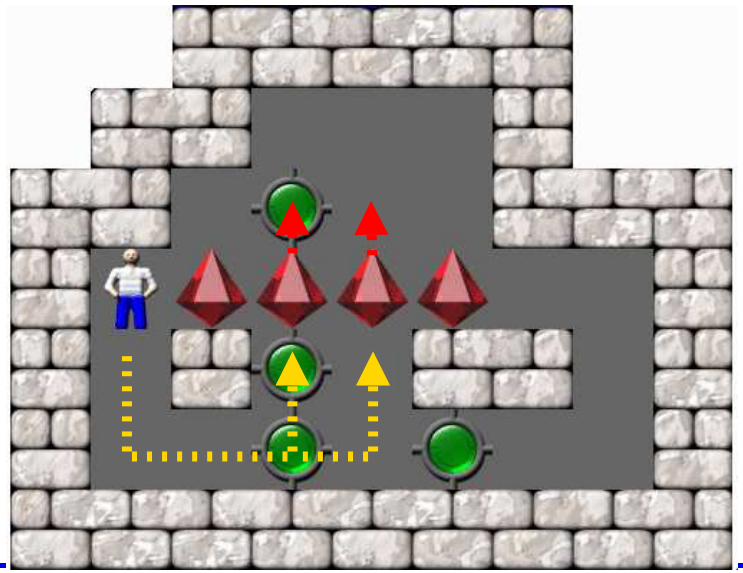
Hintergrund

- Sokoban: japanisch für „Lagerhausverwalter“
- Computerspiel von 1982, entwickelt von Hiroyuki Imabayashi
- Ziel: Spielfigur muss alle Objekte (Kisten) auf die Zielpositionen schieben
  - Kisten können nur geschoben, nicht gezogen werden
  - gleichzeitiges Verschieben mehrerer Kisten nicht möglich



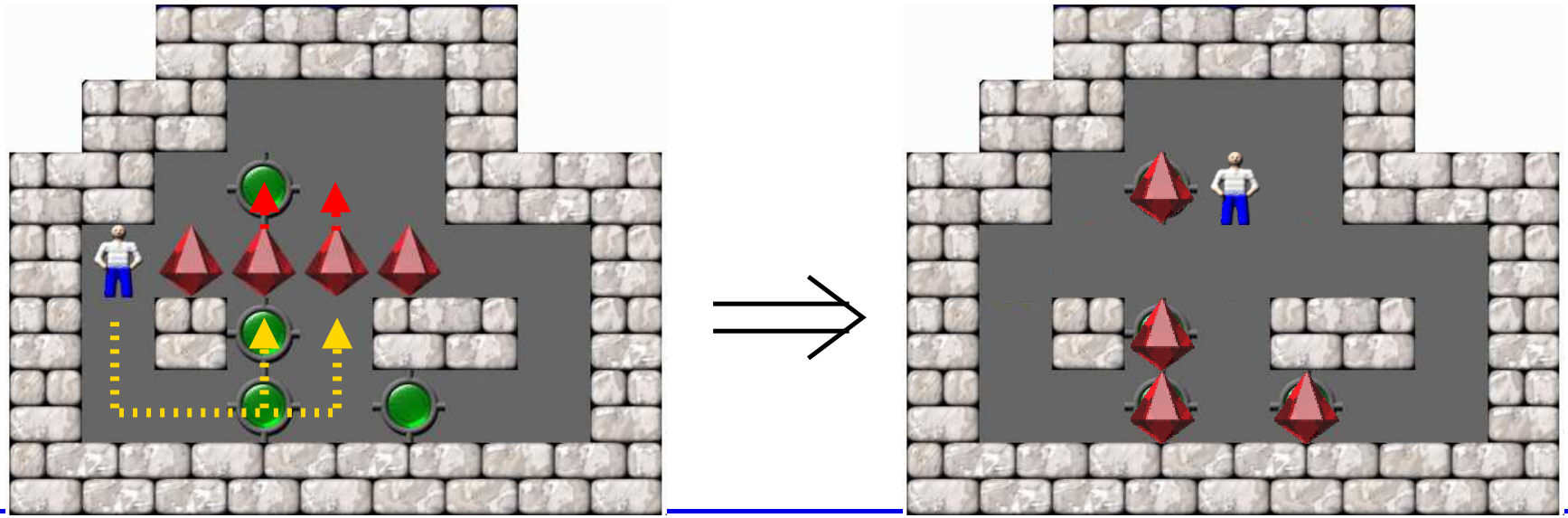
Hintergrund

- Sokoban: japanisch für „Lagerhausverwalter“
- Computerspiel von 1982, entwickelt von Hiroyuki Imabayashi
- Ziel: Spielfigur muss alle Objekte (Kisten) auf die Zielpositionen schieben
  - Kisten können nur geschoben, nicht gezogen werden
  - gleichzeitiges Verschieben mehrerer Kisten nicht möglich



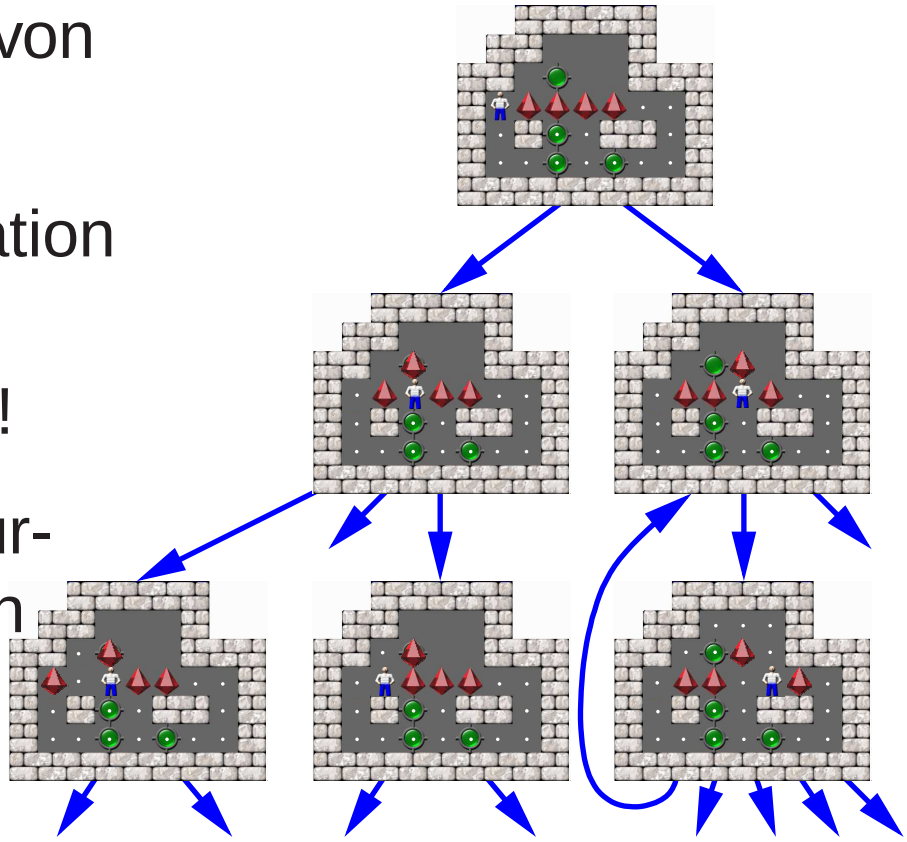
Hintergrund

- Sokoban: japanisch für „Lagerhausverwalter“
- Computerspiel von 1982, entwickelt von Hiroyuki Imabayashi
- Ziel: Spielfigur muss alle Objekte (Kisten) auf die Zielpositionen schieben
  - Kisten können nur geschoben, nicht gezogen werden
  - gleichzeitiges Verschieben mehrerer Kisten nicht möglich

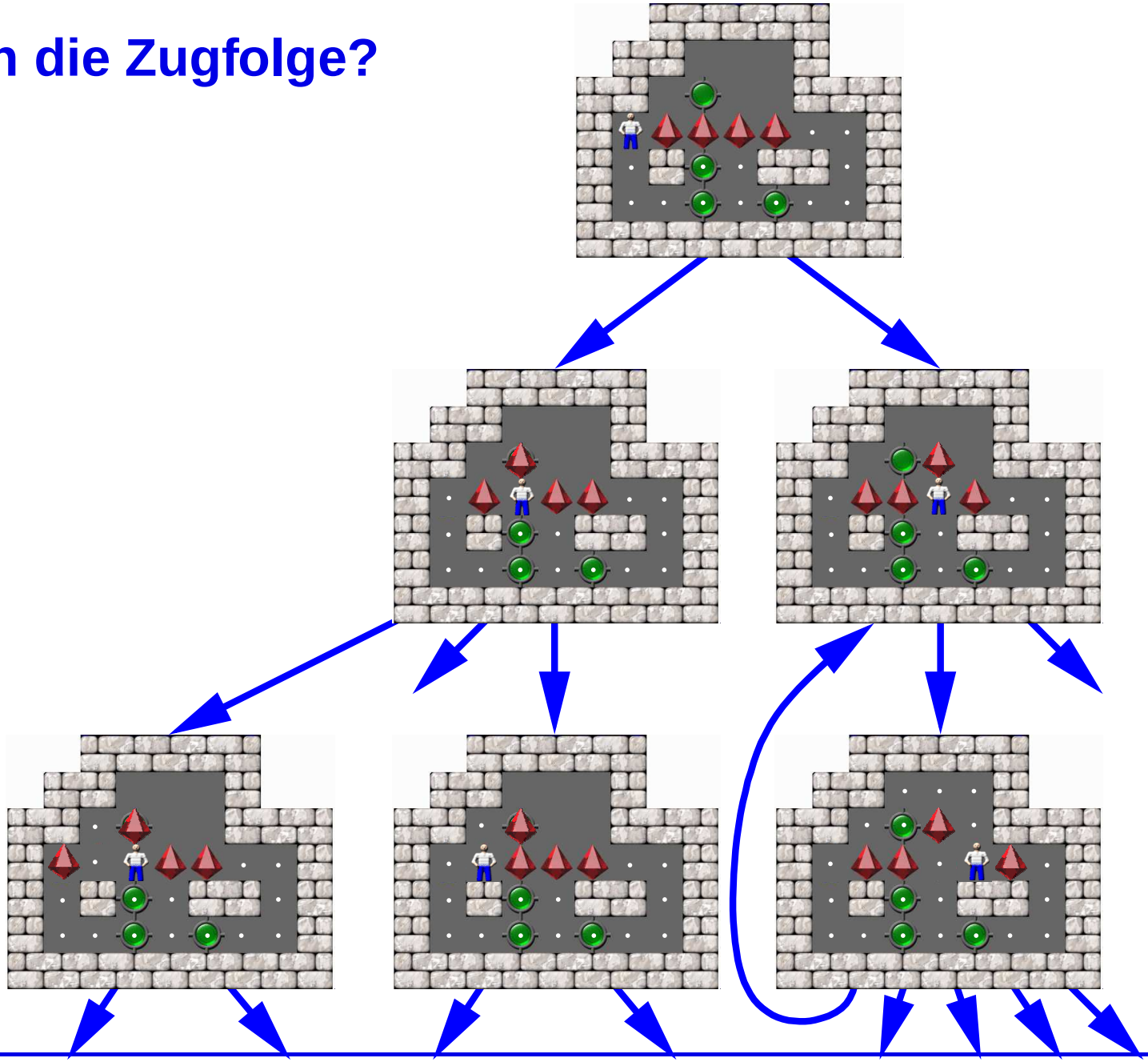


Wie findet man die Zugfolge?

- Konfiguration: Zustand des Spielfelds
  - Positionen der Kisten
  - Position der Spielfigur (Zusammenhangskomponente)
- Jede Konfiguration hat eine Menge von Nachfolge-Konfigurationen
- Konfigurationen mit Nachfolger-Relation bilden einen gerichteten Graphen
  - keinen Baum, da Zyklen möglich!
- Gesucht: kürzester Weg von der Wurzel desGraphen zur Zielkonfiguration
  - d.h. kleinste Zahl von Kistenverschiebungen

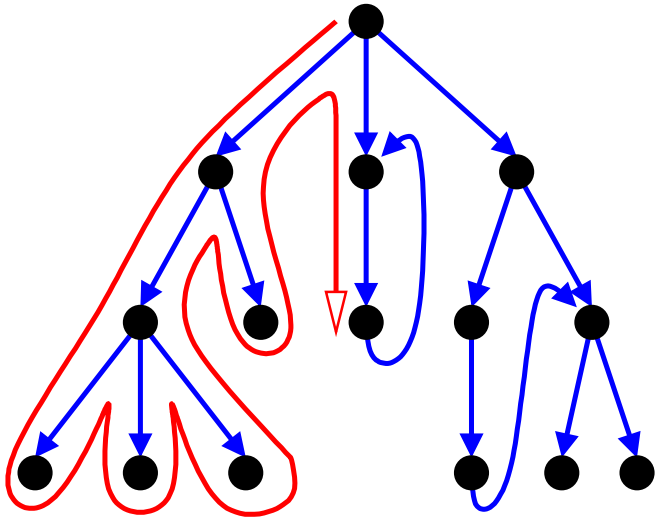


Wie findet man die Zugfolge?

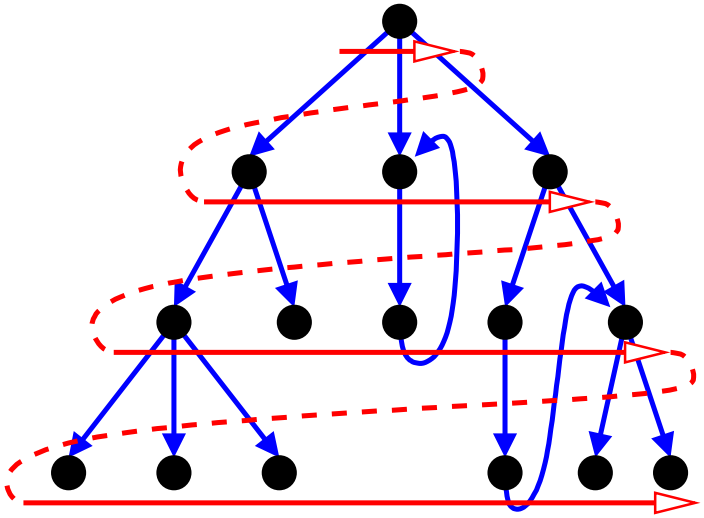


Wie findet man die Zugfolge? ...

- Zwei Alternativen:
  - Tiefensuche



  - Probleme:
    - Zyklen
    - Behandlung unterschiedlich langer Wege
  - Breitensuche



  - Probleme:
    - Rekonstruktion des Wegs zu einem Knoten
    - Speicherplatzbedarf

Backtracking-Algorithmus für Tiefensuche:

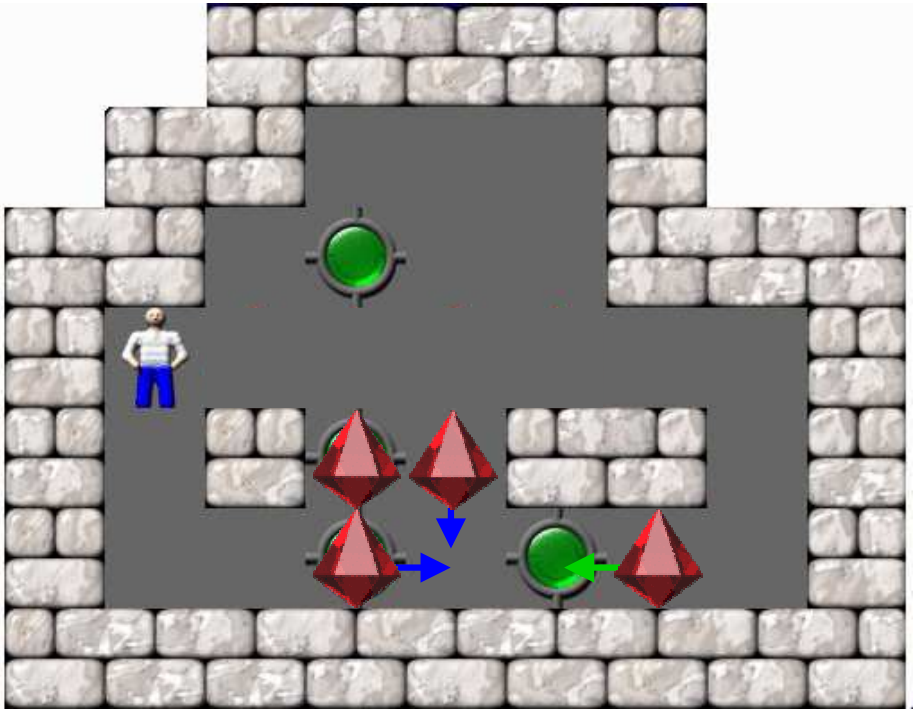
DepthFirstSearch(*conf*) // *conf* = aktuelle Konfiguration
    Füge *conf* an Lösungspfad an
    Falls *conf* Lösungskonfiguration:
        Lösungspfad gefunden
        return
    Falls Tiefe größer als die der bisher besten Lösung:
        Letztes Element vom Lösungspfed entfernen
        return // *Durchsuchung dieses Zweigs abbrechen*
    Für alle möglichen Nachfolgekonfigurationen *c* von *conf*:
        Falls *c* noch nicht in kleinerer/gleicher Tiefe gefunden wurde:
            Neue Tiefe für *c* merken
            DepthFirstSearch(*c*) // Rekursion
    Letztes Element vom Lösungspfad entfernen
    return // *backtrack*

Algorithmus für Breitensuche:

```
BreadthFirstSearch(conf)    // conf = Start-Konfiguration
    Füge conf an Warteschlange der Tiefe 0 an
    depth = 1;
    Solange Warteschlange der Tiefe depth-1 nicht leer:
        Für alle Konfigurationen conf in dieser Warteschlange:
            Für alle möglichen Nachfolgekonfigurationen c von conf:
                Falls Konfiguration c noch nicht besucht wurde:
                    Konfiguration c mit Vorgänger conf in Menge besuchter
                        Konfigurationen und in Warteschlagen der Tiefe depth
                        aufnehmen
                    Falls c Lösungskonfiguration:
                        Lösungspfad zu c bestimmen
                        return      // Lösung gefunden
        depth = depth+1
    return      // keine Lösung
```

Beispiel zum *Backtracking*-Algorithmus

Konfiguration mit möglichen Zügen

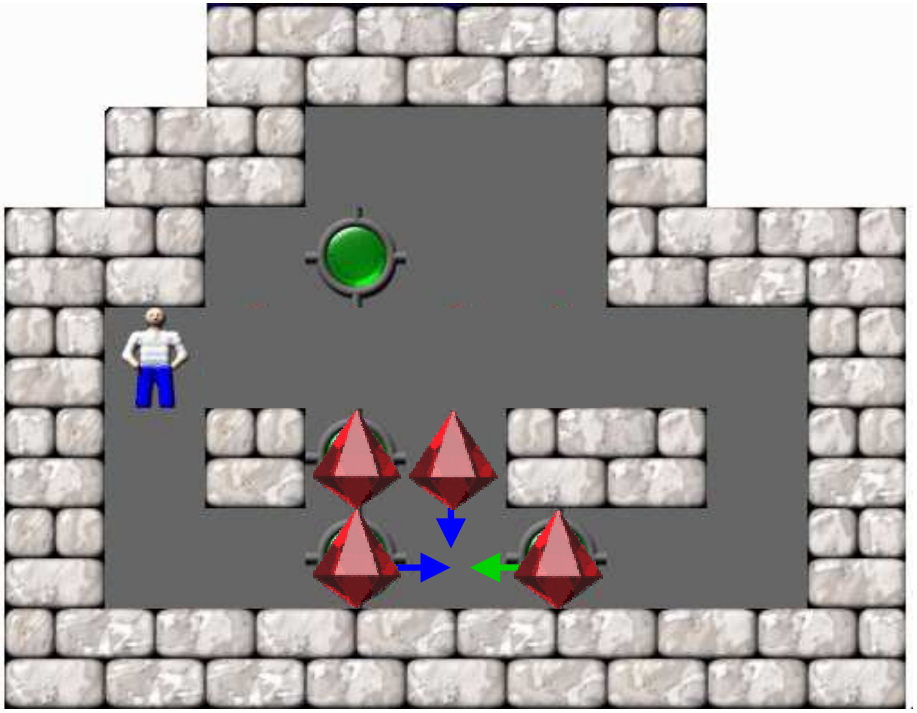


← Möglicher Zug

← Gewählter Zug

Beispiel zum *Backtracking*-Algorithmus

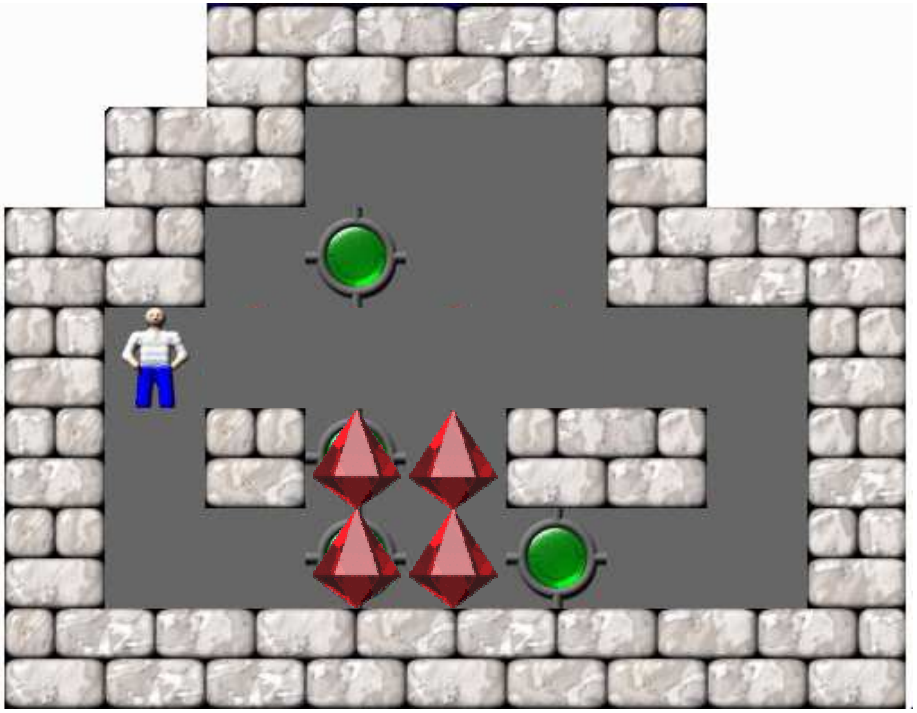
Zug ausgeführt
Neue Konfiguration mit möglichen Zügen



← Möglicher Zug
← Gewählter Zug

Beispiel zum *Backtracking*-Algorithmus

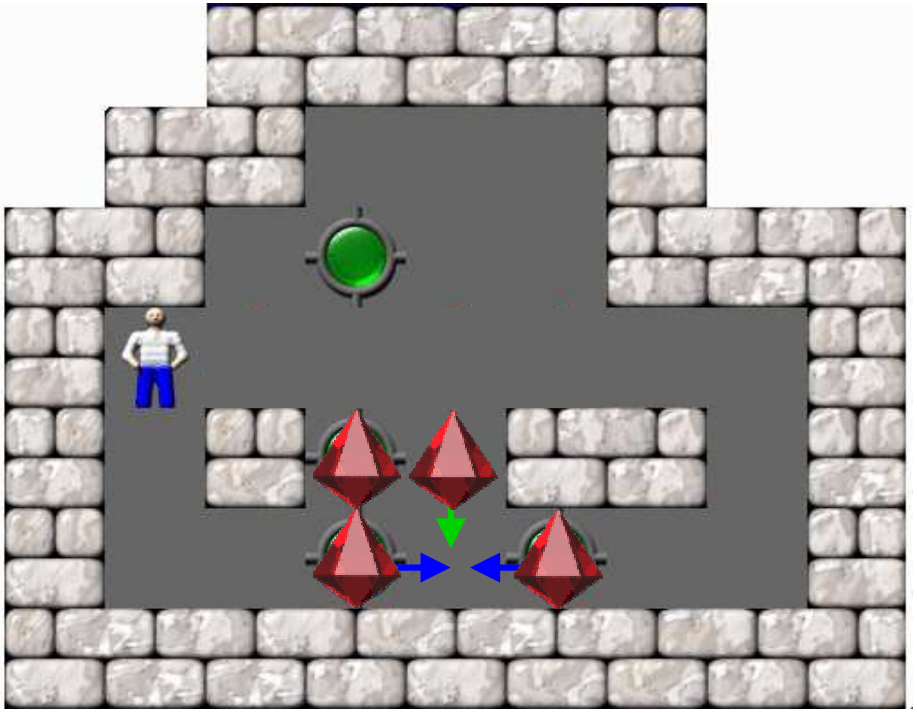
Zug ausgeführt
Kein weiterer Zug möglich



← Möglicher Zug
← Gewählter Zug

Beispiel zum *Backtracking*-Algorithmus

Backtrack
Zurück zu vorheriger Konfiguration, nächster Zug



← Möglicher Zug
← Gewählter Zug

Parallelverarbeitung

WS 2015/16

25.01.2016

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 1. Februar 2016

5.4.1 Übersetzung und Ausführung

Verwendete MPI-Implementierungen

- MPICH2 1.2.1 (im H-A 4111), OpenMPI 1.8.1 (HorUS)
- Portable Implementierungen des MPI-2 Standards
- MPICH2 enthält u.a. Visualisierungswerkzeug `jumpshot`

Übersetzung von MPI-Programmen: `mpic++`

- `mpic++ -o myProg myProg.cpp`
- kein eigener Compiler für MPI, lediglich ein Skript, das zusätzliche Compileroptionen setzt:
  - Include- und Linker-Pfade, MPI-Bibliotheken, ...
  - Option `-show` zeigt die ausgeführten Compiler-Aufrufe

Start von MPI-Programmen: `mpiexec`

- `mpiexec -n 3 myProg arg1 arg2`
  - startet `myProg arg1 arg2` mit 3 Prozessen
  - `myProg` muss im Suchpfad liegen oder mit Pfadnamen (absolut oder relativ) angegeben werden
- Auf welchen Knoten starten die Prozesse?
  - implementierungs- und plattformabhängig
  - in MPICH2 (mit Hydra Prozessmanager): Festlegung über eine Konfigurationsdatei möglich:

    `mpiexec -n 3 -machinefile machines myProg arg1 arg2`
    - Konfigurationsdatei enthält Liste von Rechnernamen, z.B.:

      `bslab01` ← einen Prozess auf `bslab03` starten
      `bslab05:2` ← zwei Prozesse auf `bslab05` starten

Verwendung von MPICH2 im H-A 4111

- Ggf. zunächst Pfad setzen (in `~/.bashrc`):
  - `export PATH=/opt/dist/mpich2-1.2.1/bin:$PATH`
- Hinweise zu `mpiexec`:
  - Die MPI-Programme starten auf den anderen Rechnern im selben Verzeichnis, in dem `mpiexec` aufgerufen wurde
  - `mpiexec` startet per `ssh` einen Hilfsprozess auf jedem Rechner, der in der Konfigurationsdatei genannt wird
    - auch wenn gar nicht so viele MPI-Prozesse gestartet werden sollen
  - Eintrag `localhost` in der Konfigurationsdatei vermeiden
    - führt zu Problemen mit `ssh`
  - Sporadisch werden Programme mit relativem Pfadnamen (`./myProg`) nicht gefunden, dann `$PWD/myProg` angeben

5.4.2 Debugging

- MPICH2 und OpenMPI unterstützen `gdb` und `totalview`
  - MPICH2 allerdings nur mit dem MPD Prozessmanager
- „Hack“ im H-A 4111: starte einen `gdb` / `ddd` für jeden Prozeß
  - Aufruf für `gdb`: `mpiexec -enable-x -n ... /opt/dist/mpidebug/mpigdb myProg args`
    - `gdb` startet jeweils in eigenem Textfenster
  - Aufruf für `ddd`: `mpiexec -enable-x -n ... /opt/dist/mpidebug/mpiddd myProg args`
  - Prozesse mit `continue` weiterlaufen lassen (nicht `run ...`)
- Voraussetzung: Übersetzung mit Debugging-Information
  - `mpic++ -g -o myProg myProg.cpp`

5.4.3 Leistungsanalyse mit Scalasca

- Prinzipiell genauso wie für OpenMP
- Übersetzen des Programms:
  - `scalasca -instrument mpic++ -o myprog myprog.cpp`
- Ausführen des Programms:
  - `scalasca -analyze mpiexec -n 4 ... ./myprog`
  - legt Verzeichnis `scorep_myprog_4_sum` an
    - `4` zeigt Zahl der Prozesse an
    - Verzeichnis darf noch nicht existieren, ggf. löschen
- Interaktive Analyse der aufgezeichneten Daten:
  - `scalasca -examine scorep_myprog_4_sum`

5.4.4 Leistungsanalyse und Visualisierung mit Jumpshot

- MPICH unterstützt die Erzeugung von Ereignisspuren
  - Ereignisse: MPI-Aufrufe und -Rückkehr, Senden, Empfangen, ...
- Einschalten der Funktion über Optionen von `mpic++`
- `mpic++ -mpe=mpitrace -o myProg myProg.cpp`
  - Ausgabe der Ereignisse auf den Bildschirm
- `mpic++ -mpe=mpilog -o myProg myProg.cpp`
  - Ausgabe der Ereignisse in Spurdatei `myProg.clog2`
    - eine gemeinsame Spurdatei für alle Prozesse
  - Analyse der Spurdatei mit `jumpshot`:
    - Formatwandlung: `clog2TOslog2 myProg.clog2`
    - Visualisierung: `jumpshot myProg.slog2`

Beispiel: Ping-Pong Programm

- Modifikation im Programm: Prozeß 0 führt zwischen Senden und Empfangen eine Berechnung durch
  - damit das Beispiel interessanter wird ...
- Kommandos:

```
mpic++ -mpe=mpilog -o pingpong pingpong.cpp
```

  - bindet Code zur Erzeugung von Spurdateien mit ein

```
mpiexec -n 2 -machinefile machines ./pingpong 5 100000
```

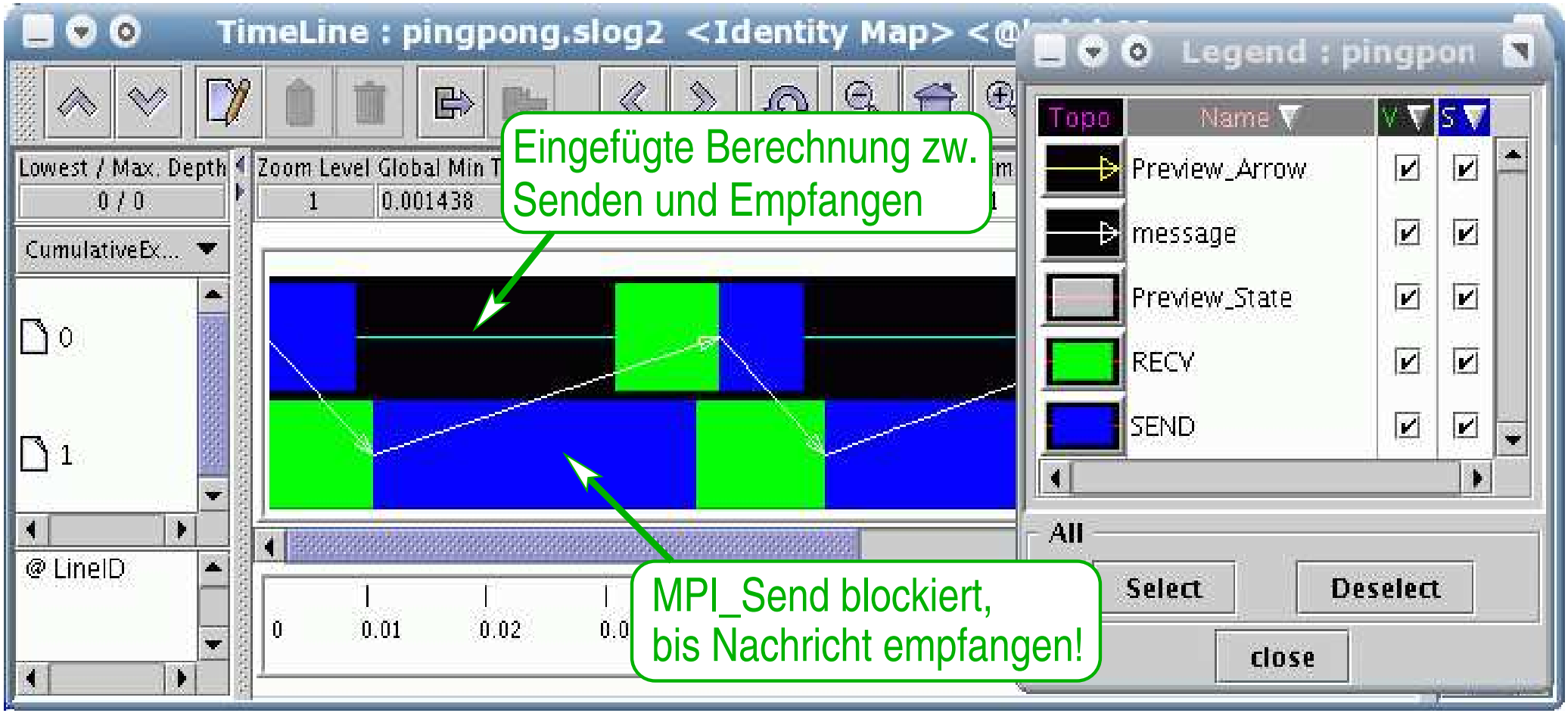
  - hinterläßt Spurdatei `pingpong.clog2`

```
clog2TOslog2 pingpong.clog2
```

  - wandelt `pingpong.clog2` in `pingpong.slog2` um

```
jumpshot pingpong.slog2
```

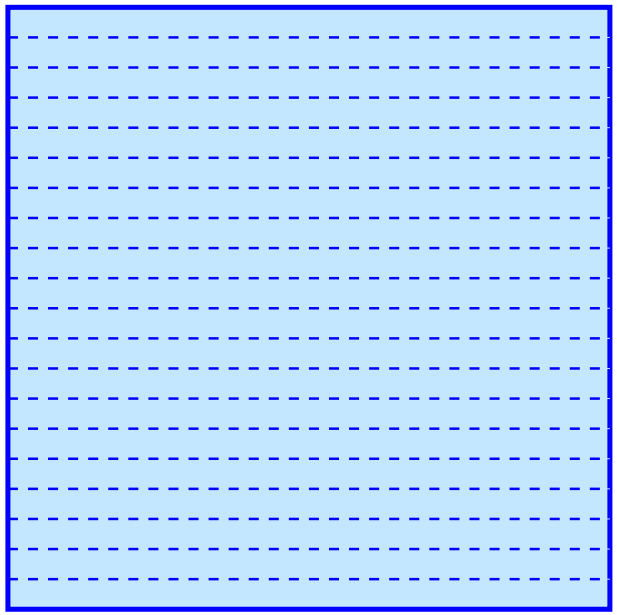
Beispiel: Ping-Pong Programm ...



5.4.5 Nutzung des HorUS Clusters

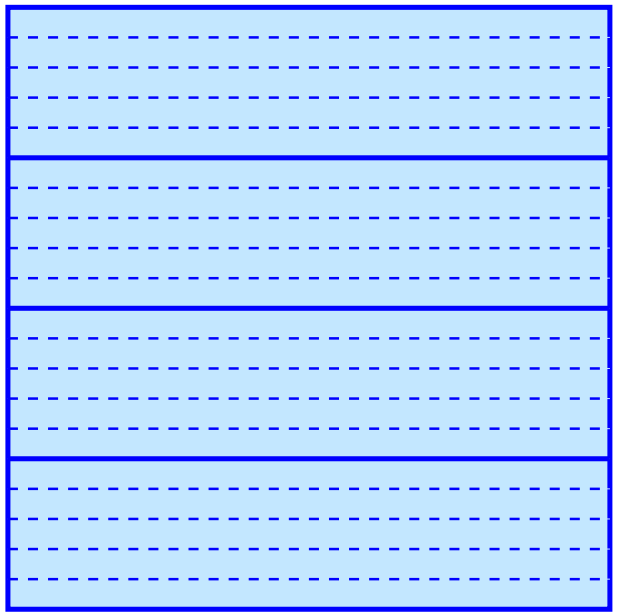
- Start von MPI-Programmen über SLURM, z.B. mit:
  - `salloc --exclusive --partition short --nodes=4 \`
    `--ntasks=16 mpiexec $HOME/GAUSS/heat 500`
  - wichtige Optionen von `salloc`:
    - `--nodes=4`: Zahl der zu allokierenden Knoten
    - `--ntasks=16`: Gesamtzahl der zu startenden Prozesse
- Zur Leistungsanalyse: Scalasca
  - `salloc ... scalasca -analyze mpiexec $HOME/GAUSS...`
- Zum Debugging: Totalview
  - `salloc ... mpiexec -debug $HOME/GAUSS/heat 500`

Grundsätzliche Vorgehensweise



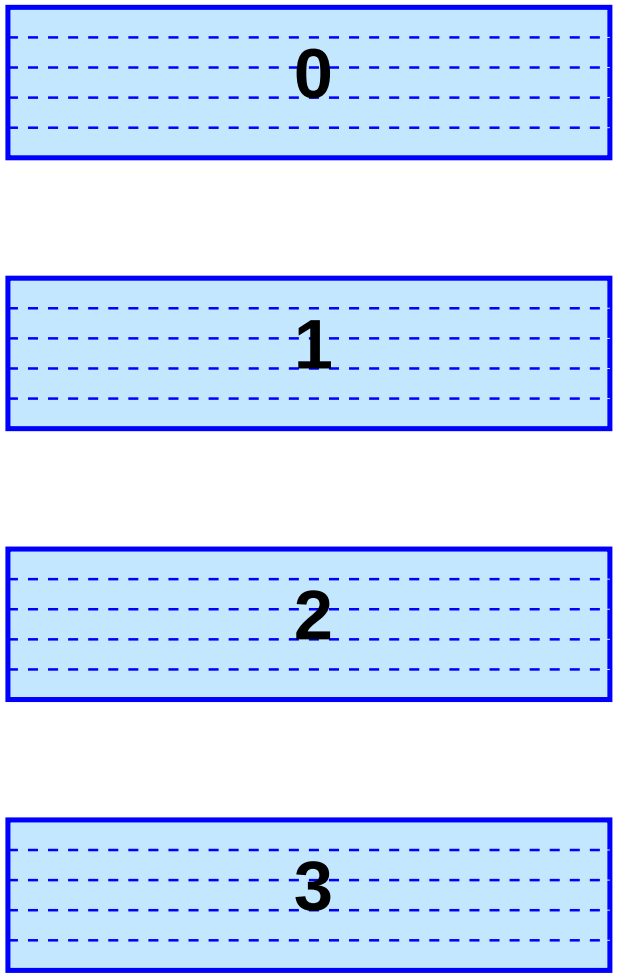
0. Matrix mit Temperaturwerten

Grundsätzliche Vorgehensweise



0. Matrix mit Temperaturwerten
1. Aufteilung der Matrix in Streifen

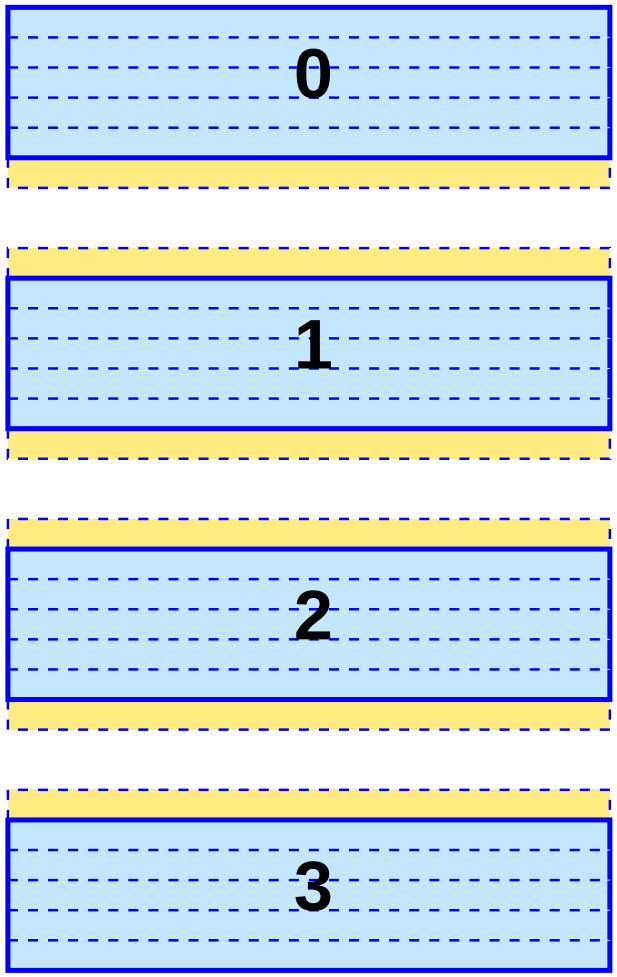
Grundsätzliche Vorgehensweise



0. Matrix mit Temperaturwerten
1. Aufteilung der Matrix in Streifen

   Jeder Prozeß speichert lokal nur einen Teil der Matrix

Grundsätzliche Vorgehensweise

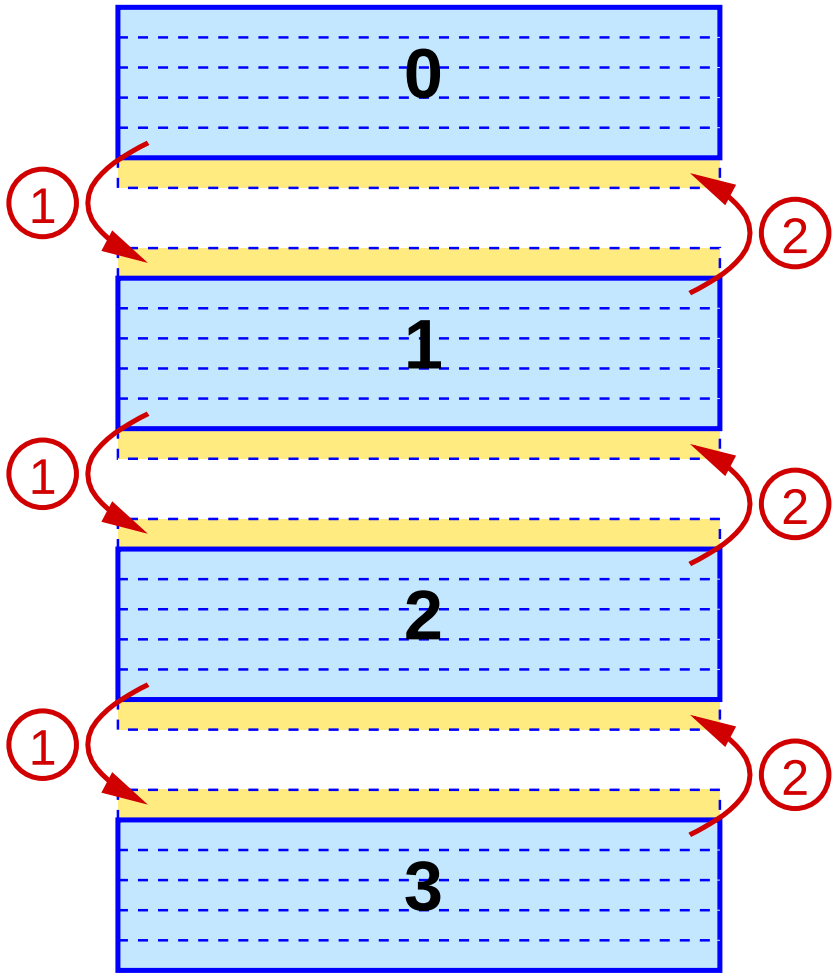


0. Matrix mit Temperaturwerten
1. Aufteilung der Matrix in Streifen

   Jeder Prozeß speichert lokal nur einen Teil der Matrix
2. Einführung von Überlappungsbereichen

   Jeder Prozeß speichert an den Schnitt–kanten jeweils eine Zeile zusätzlich

Grundsätzliche Vorgehensweise



0. Matrix mit Temperaturwerten
1. Aufteilung der Matrix in Streifen

   Jeder Prozeß speichert lokal nur einen Teil der Matrix
2. Einführung von Überlappungsbereichen

   Jeder Prozeß speichert an den Schnittkanten jeweils eine Zeile zusätzlich
3. Nach jeder Iteration werden die Überlappungsbereiche mit den Nachbarprozessen ausgetauscht

   Z.B. erst nach unten (1),
   dann nach oben (2)

Grundsätzliche Vorgehensweise ...

```
int nprocs, myrank;
double a[LINES][COLS];
MPI_Status status;

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* Schritt 1: Sende nach unten, Empfange von oben */
if (myrank != nprocs-1)
   MPI_Send(a[LINES-2], COLS, MPI_DOUBLE, myrank+1, 0,
            MPI_COMM_WORLD);
if (myrank != 0)
   MPI_Recv(a[0], COLS, MPI_DOUBLE, myrank-1, 0,
            MPI_COMM_WORLD, &status);
```

Aufteilung der Daten

- Geschlossene Formel zur gleichmäßigen Aufteilung eines Feldes der Länge n auf np Prozesse:
  - $\mathbf{start}(p) = n \div np \cdot p + \mathbf{max}(p - (np - n \bmod np), 0)$
  - $\mathbf{size}(p) = (n + p) \div np$
  - Prozess p erhält $\mathbf{size}(p)$ Elemente ab Index $\mathbf{start}(p)$
- Damit ergibt sich folgende Indextransformation:
  - $\mathbf{tolocal}(i) = (p, i - \mathbf{start}(p))$
    mit $p \in [0, np - 1]$ so, daß $0 \leq i - \mathbf{start}(p) < \mathbf{size}(p)$
  - $\mathbf{toglobal}(p, i) = i + \mathbf{start}(p)$
- Bei Jacobi und Gauss/Seidel sind zusätzlich noch die Überlappungsbereiche zu berücksichtigen!

Aufteilung der Berechnungen

- Aufteilung i.a. nach *Owner computes*-Regel
  - der Prozeß, der ein Datenelement schreibt, führt auch die entsprechende Berechnungen durch
- Zwei Möglichkeiten zur technischen Realisierung:
  - Index-Transformation und bedingte Ausführung
    - z.B. bei der Ausgabe der Kontrollwerte der Matrix

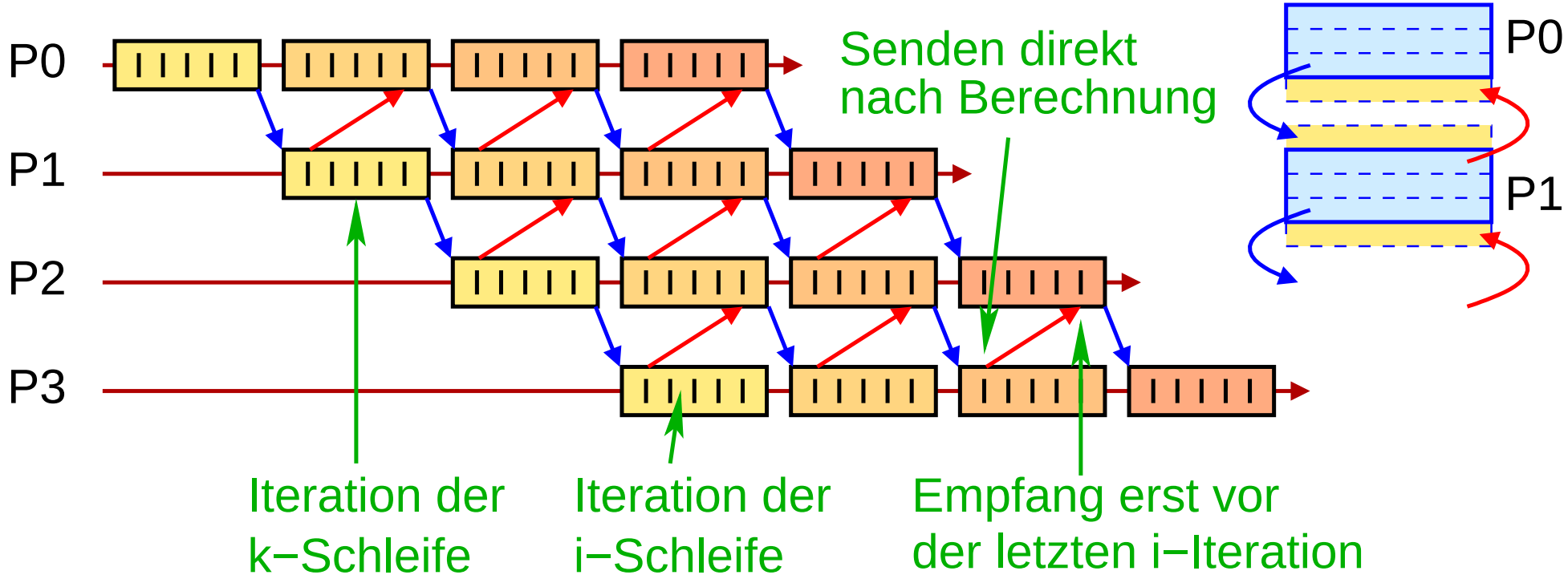
```
if ((x-start >= 0) && (x-start < size))
   cout << "a[" << x << "]=" << a[x-start] << "\n";
```

  - Anpassung der umgebenden Schleifen
    - z.B. bei der Iteration oder der Intialisierung der Matrix

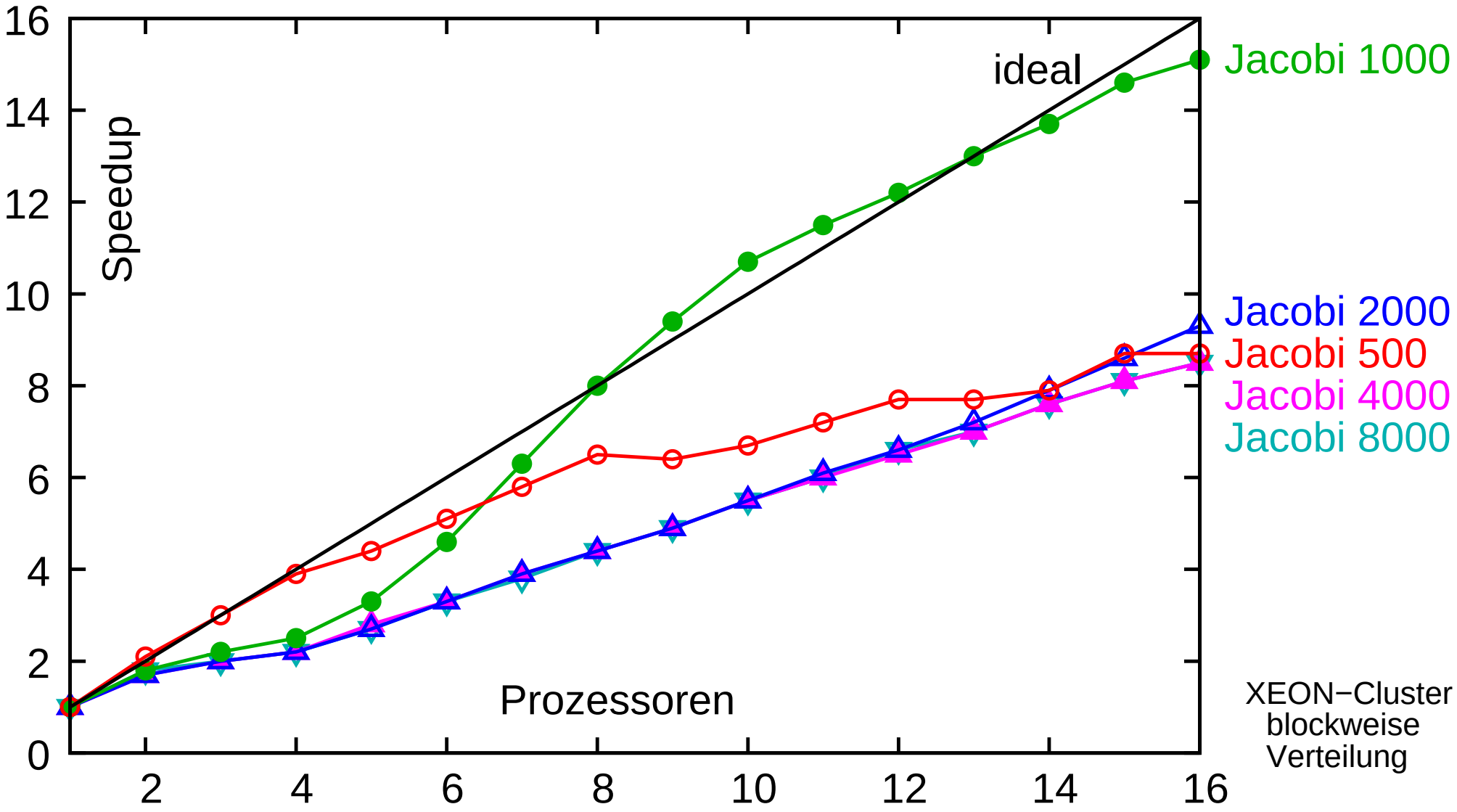
```
for (i=0; i<size; i++)
   a[i] = 0;
```

Zur Parallelisierung der Gauss/Seidel-Relaxation

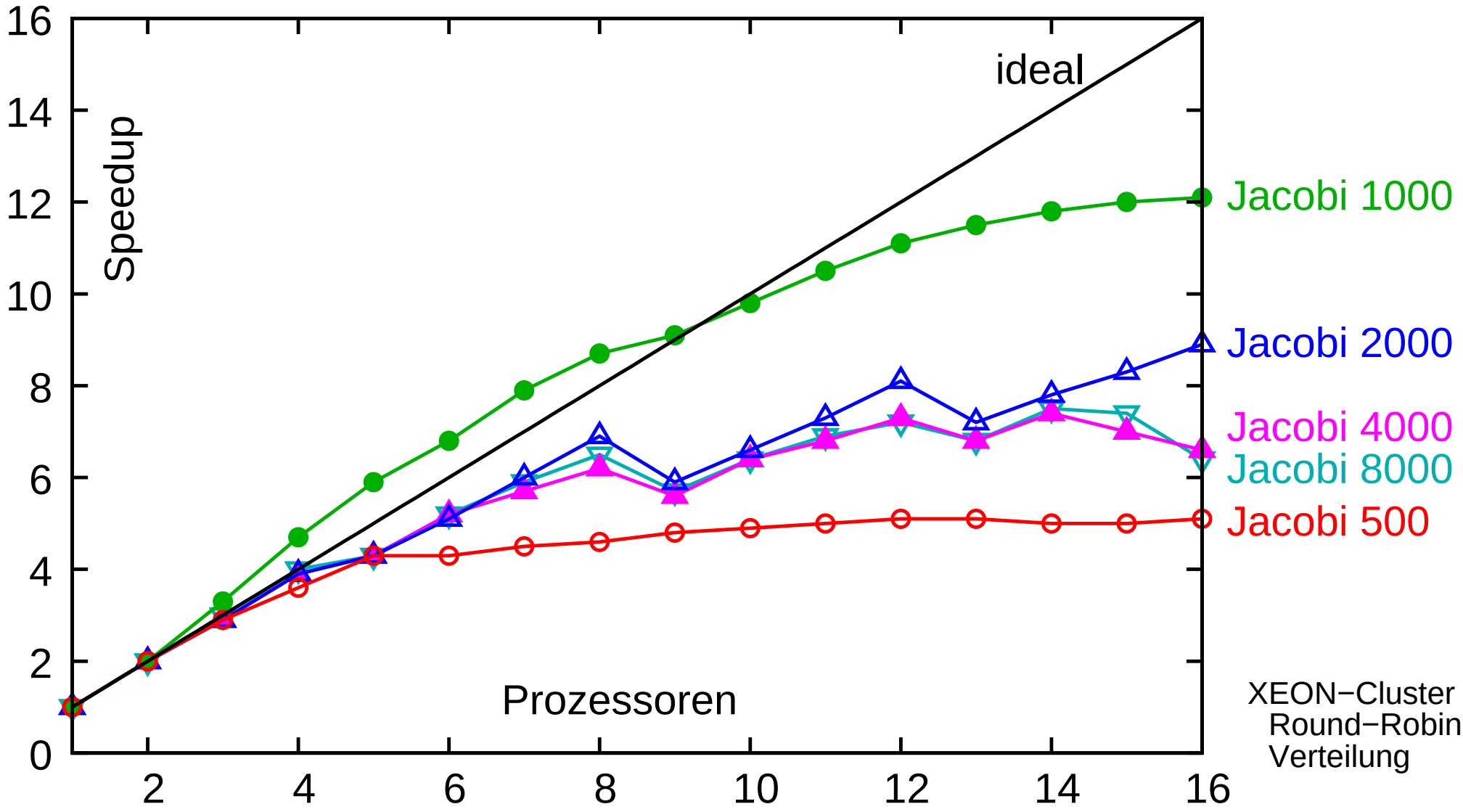
- Analog zur pipeline-artigen Parallelisierung mit OpenMP (☞ **2.5**)



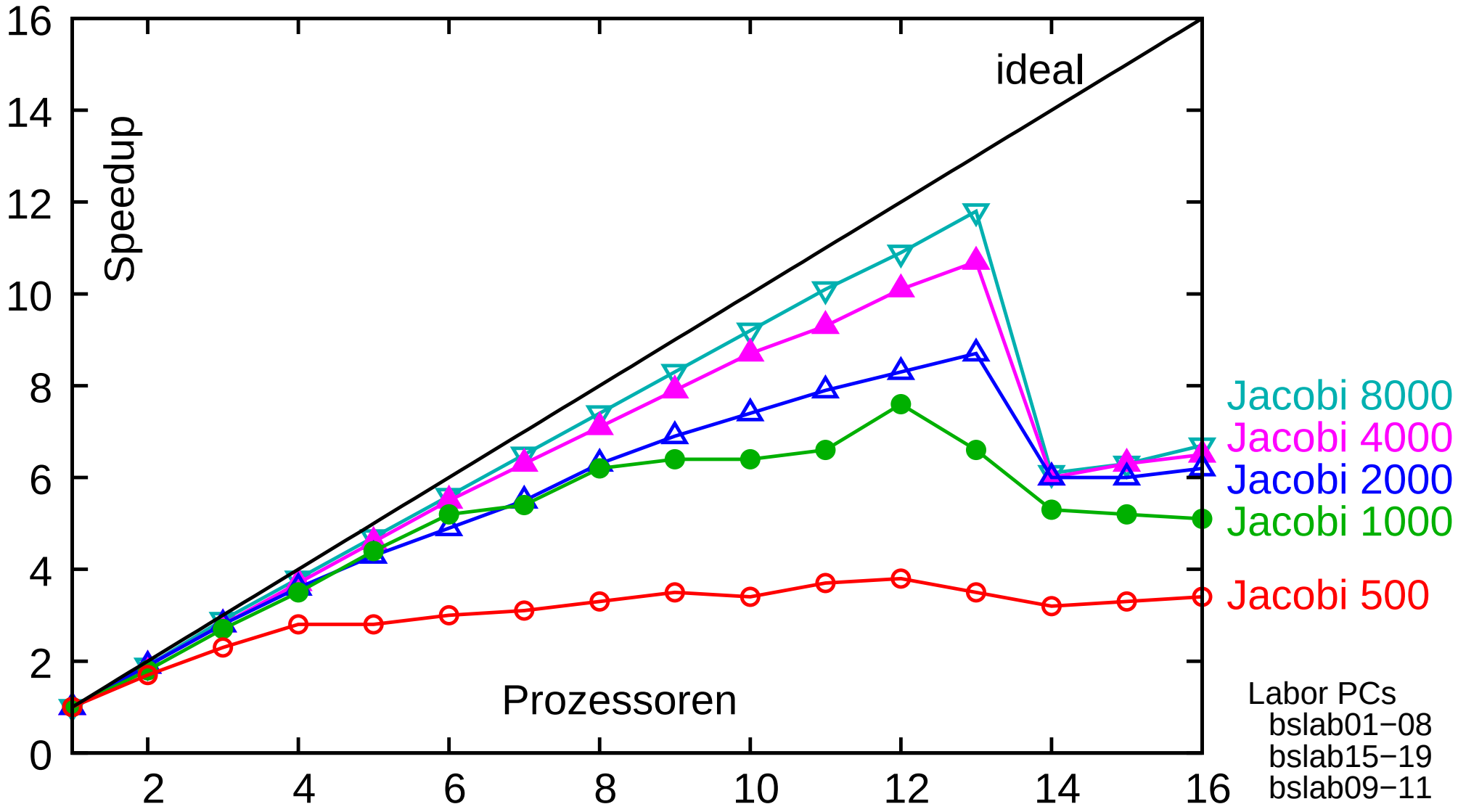
Erreichter Speedup für verschiedene Matrixgrößen



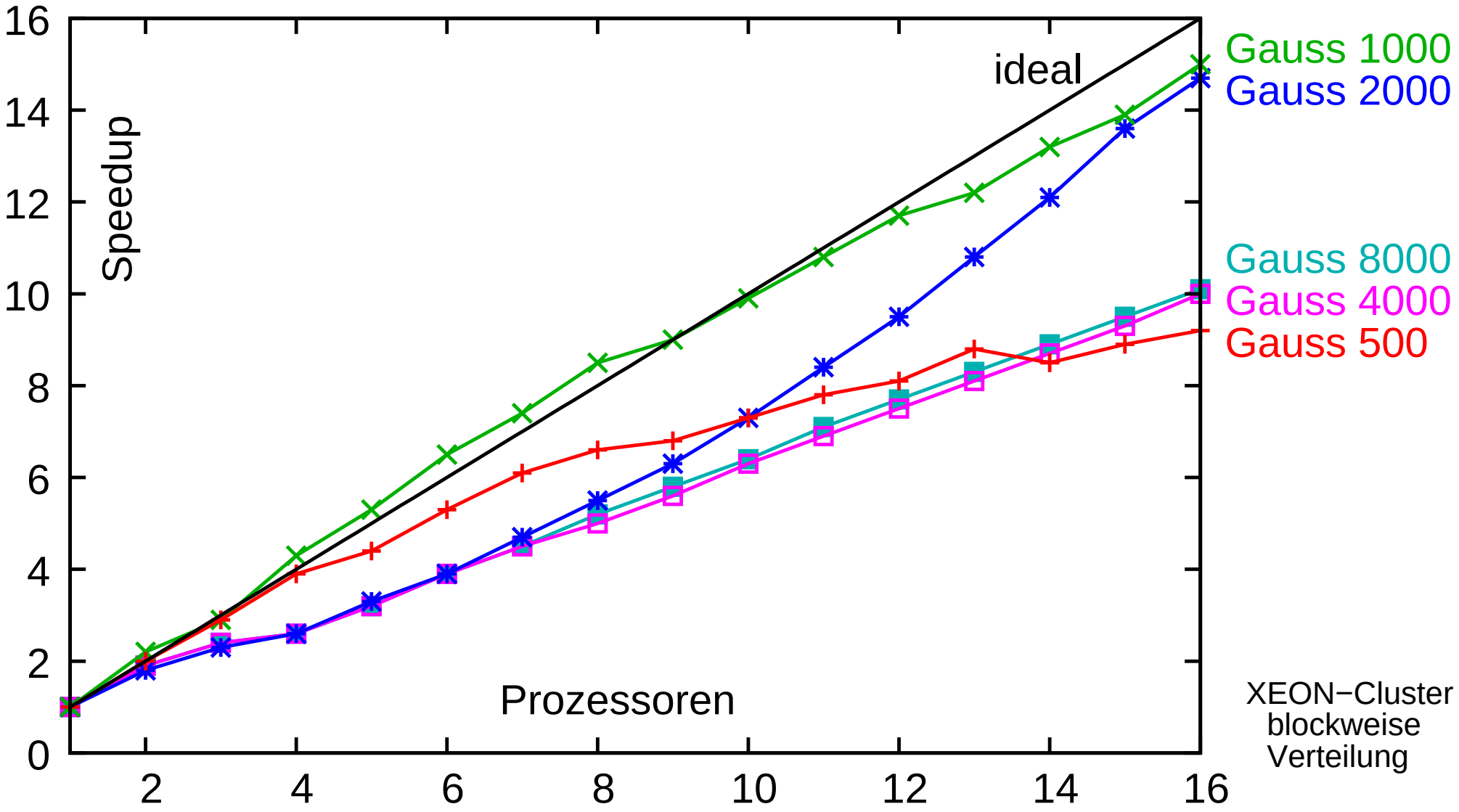
Erreichter Speedup für verschiedene Matrixgrößen



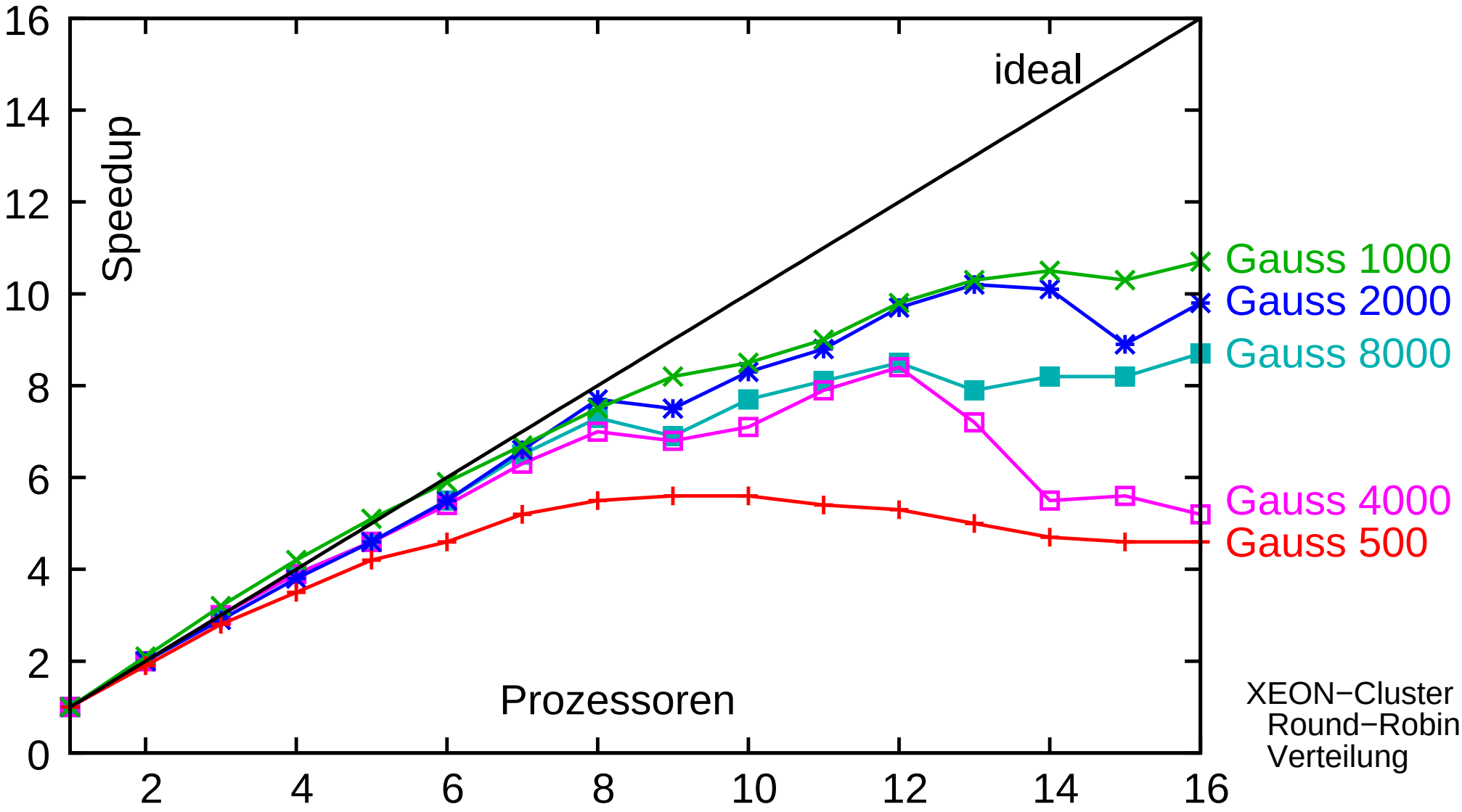
Erreichter Speedup für verschiedene Matrixgrößen



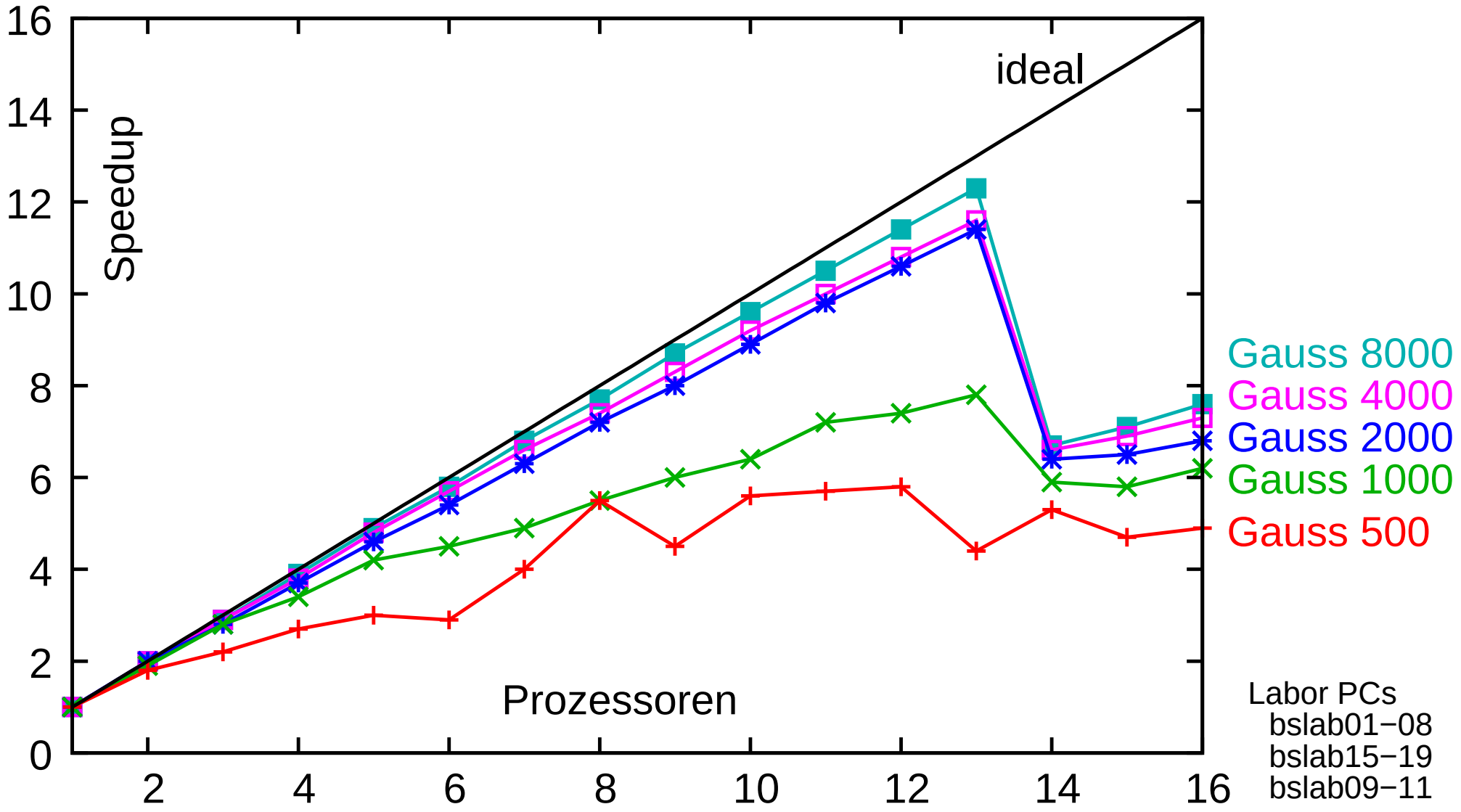
Erreichter Speedup für verschiedene Matrixgrößen



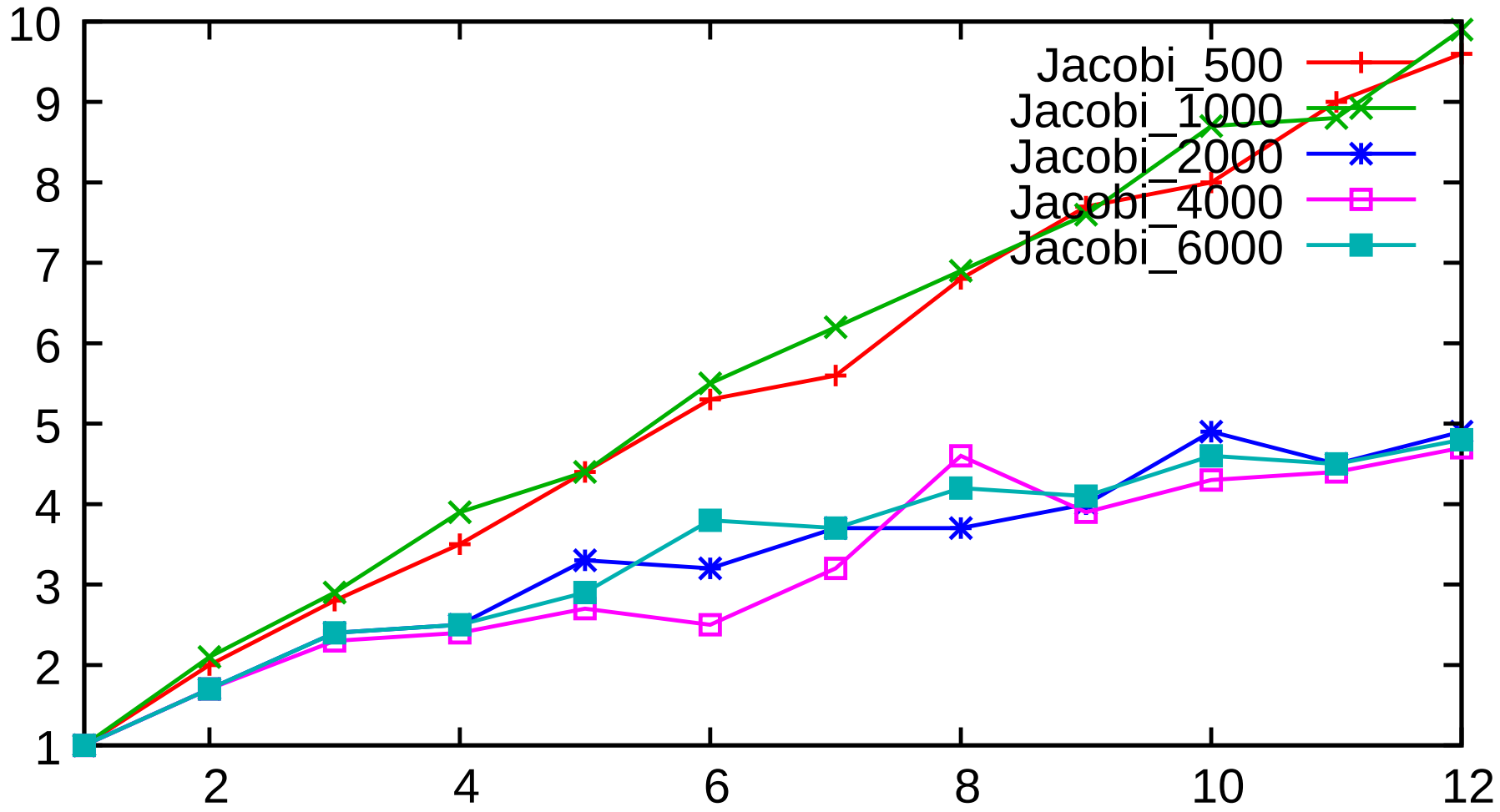
Erreichter Speedup für verschiedene Matrixgrößen



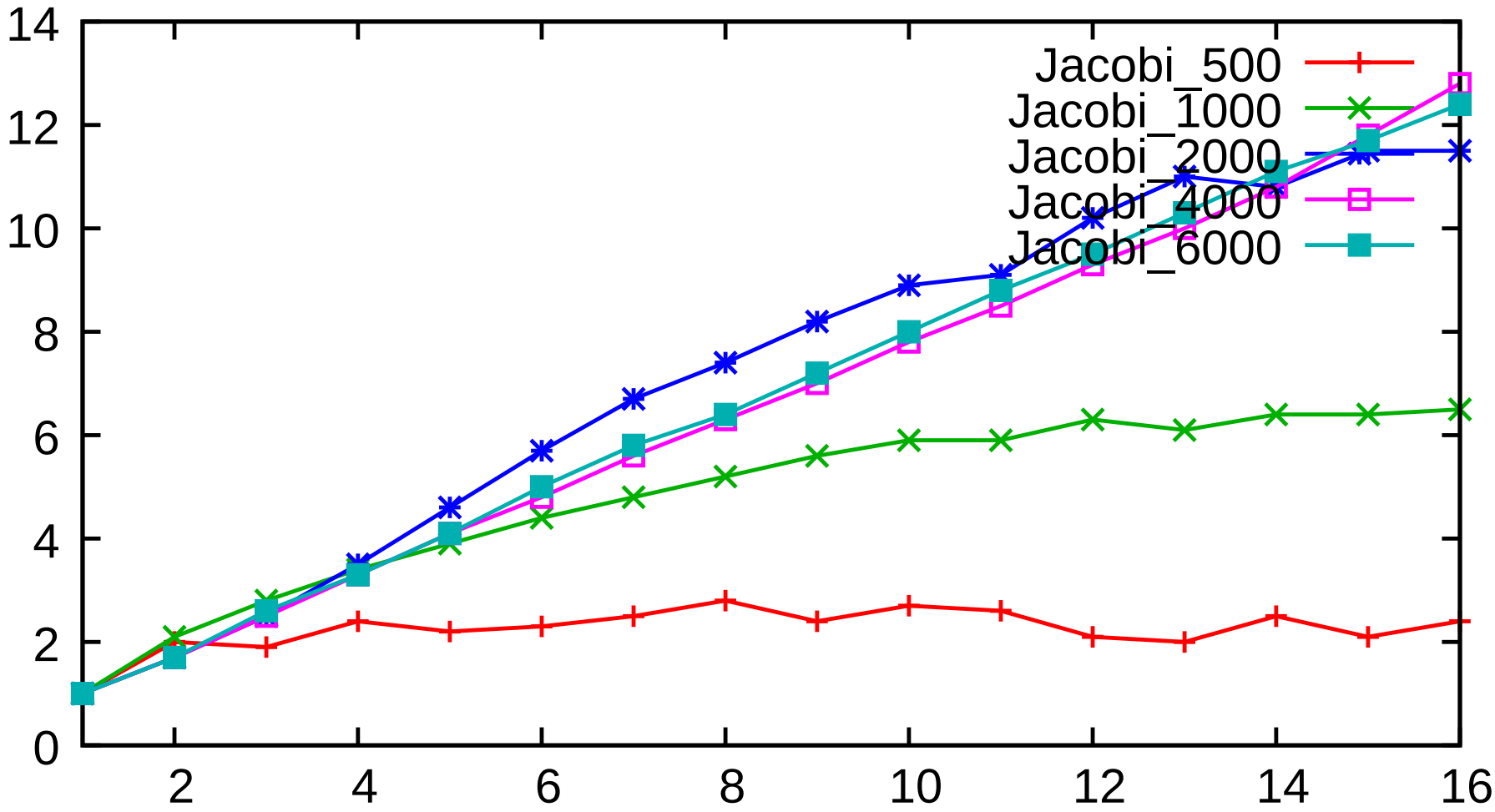
Erreichter Speedup für verschiedene Matrixgrößen



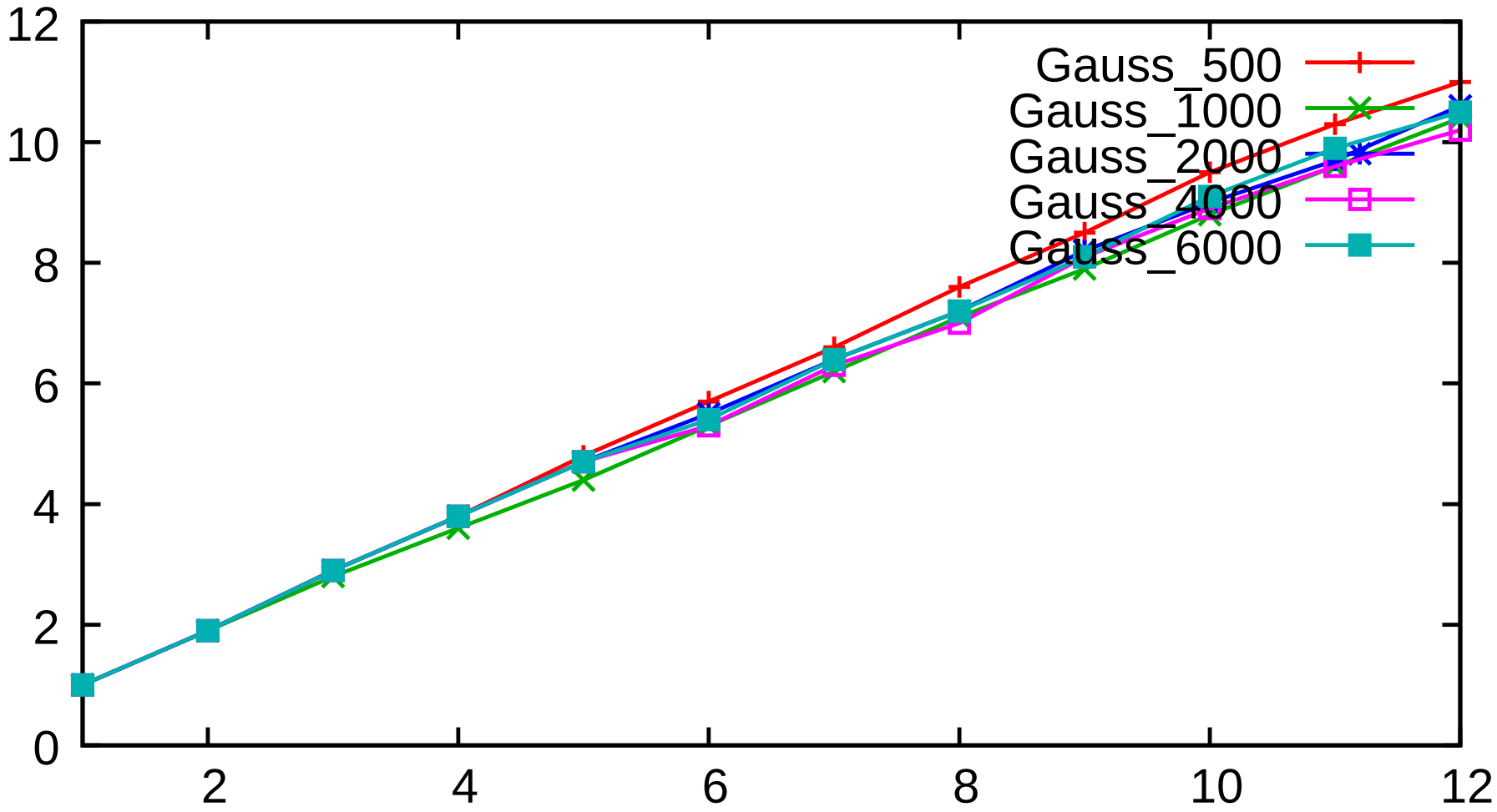
Speedup auf Horus-Cluster: Jacobi, 1 Knoten



Speedup auf Horus-Cluster: Jacobi, 2 Tasks/Knoten



Speedup auf Horus-Cluster: Gauss, 1 Knoten



Speedup auf Horus-Cluster: Gauss, 2 Tasks/Knoten