

# **Hochleistungsrechnen**

## Vorlesung im Wintersemester 2012/13

Skriptversion 07.11.2012

Prof. Dr. Thomas Ludwig  
Universität Hamburg – Informatik – Wissenschaftliches Rechnen

# Einleitung

---

Definition in Wikipedia:

- ▶ **Hochleistungsrechnen** (englisch: *high-performance computing – HPC*) ist ein Bereich des computergestützten Rechnens. Er umfasst alle Rechenarbeiten, deren Bearbeitung einer hohen Rechenleistung oder Speicherkapazität bedarf.
- ▶ **Hochleistungsrechner** sind Rechnersysteme, die geeignet sind, Aufgaben des Hochleistungsrechnens zu bearbeiten.

Ein wichtiges Gebiet der Informatik mit sehr vielen Facetten und Bezügen zur Mathematik

# Anwendungsbereiche

---

Anwendungsbereiche sind in allen Wissenschaften mit einem hohen Bedarf an Rechen- und Speicherleistung

Klassisch:

- ▶ Physik (Astronomie, Teilchenphysik, ...)
- ▶ Erdsystemforschung (Klima, Ozeanographie, ...)
- ▶ Bioinformatik (Stammbaumberechnungen, Pharmazie, ...)

Neu dazugekommen:

- ▶ Finanzwirtschaft
- ▶ Sozialwissenschaften (Simulation von Gesellschaften)

# Ausprägungen

---

- ▶ Klein
  - ▶ Mehrere Prozessoren in einem Rechner oder Prozessorkerne in einem Prozessor
  - ▶ Einige hundert Euro
- ▶ Groß
  - ▶ Hunderttausende von Prozessorkernen in einem Großrechner
  - ▶ Plattspeicher im Bereich einzelner Petabyte
  - ▶ Bandarchive im Bereich dutzender Petabyte
  - ▶ 10...200 Millionen Euro



# Themenüberblick

---

- ▶ Teil I: Hardware- und Software-Konzepte
- ▶ Teil II: Programmierung
- ▶ Teil III: Programmierwerkzeuge
- ▶ Teil IV: Aktuelle Fragestellungen

# Teil I: Hardware- und Software-Konzepte

---

- ▶ Hardware-Architekturen (17-46)
- ▶ Die TOP500-Liste (47-107)
- ▶ Vernetzungkonzepte (111-144)
- ▶ Hochleistungs-Eingabe/Ausgabe (145-178)
- ▶ Betriebssystemaspekte (179-207)

# Teil II: Programmierung

---

- ▶ Parallel Programmierung (208-248)
- ▶ Programmiermodell Nachrichtenaustausch (249-285)
- ▶ Parallel Eingabe/Ausgabe (286-311)
- ▶ Programmierung mit Threads (312-353)
- ▶ Programmierung mit OpenMP (354-393)
- ▶ Optimierung sequentieller Programme (394-438)
- ▶ Leistungsmodellierung (439-468)
- ▶ Hybride Programmierung
- ▶ Grafikkartenprogrammierung

# Teil III: Programmierwerkzeuge

---

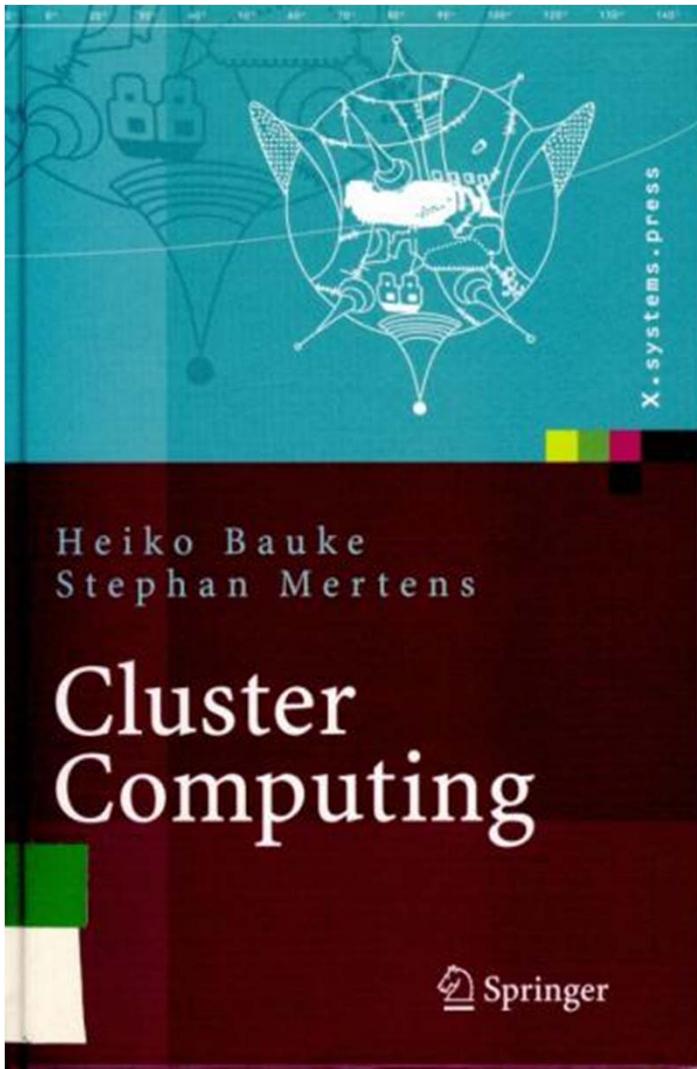
- ▶ Werkzeugarchitekturen (469-506)
- ▶ Fehlersuche (507-542)
- ▶ Leistungsanalyse (543-572)
- ▶ Leistungsoptimierung
- ▶ Lastausgleich (573-595)
- ▶ Fehlertoleranz (596-633)

# Teil IV: Aktuelle Fragestellungen

---

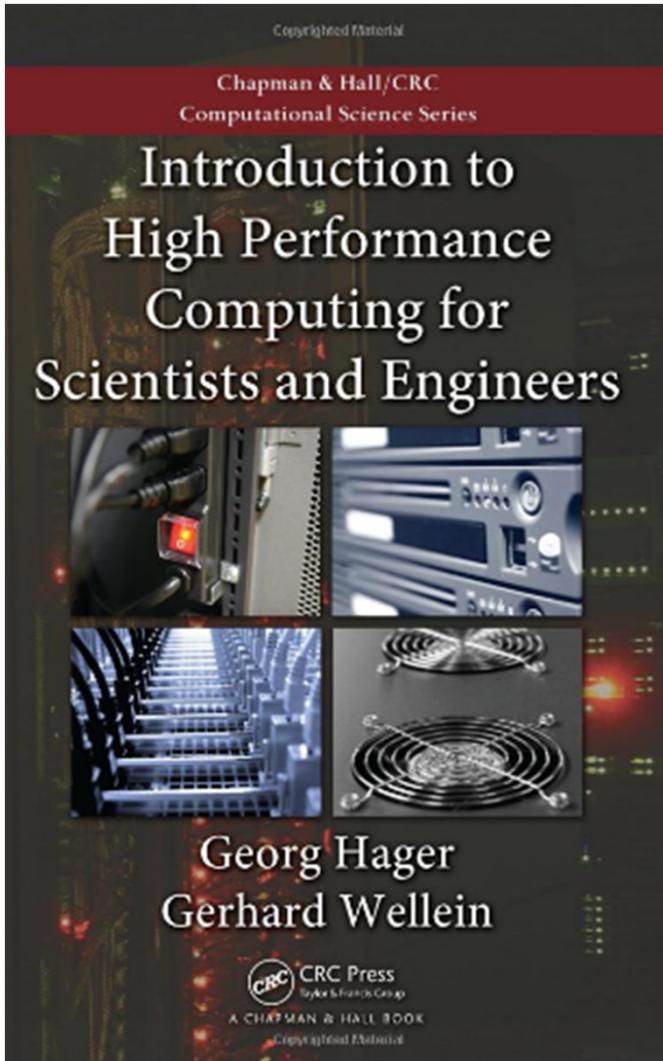
- ▶ Grid- und Cloud-Computing (629-662)
- ▶ Energieeffizienz im Hochleistungsrechnen
- ▶ Die Geschichte des parallelen Rechnens
- ▶ Die Zukunft des parallelen Rechnens

# Literatur



Heiko Bauke &  
Stephan Mertens  
Cluster Computing  
Springer, 2006  
460 Seiten  
Amazon: 10 Euro  
ISBN-10: 3-540-42299-4

# Literatur...



Georg Hager &  
Gerhard Wellein  
Introduction to High  
Performance Computing  
for Scientists and  
Engineers  
Chapman & Hall, 2010  
330 Seiten  
Amazon: 52 Euro  
ISBN-10: 143981192X

# Weiterführende Informationen

---

- ▶ Materialienseite
  - ▶ [http://wr.informatik.uni-hamburg.de/teaching/  
wintersemester\\_2012\\_2013/hochleistungsrechnen](http://wr.informatik.uni-hamburg.de/teaching/wintersemester_2012_2013/hochleistungsrechnen)
  - ▶ Foliensätze, Übungsblätter usw.
- ▶ Mailingliste
  - ▶ <http://wr.informatik.uni-hamburg.de/listinfo/hr-1213>
  - ▶ Wichtige Neuigkeiten und Diskussionen zwischen den  
Teilnehmern

# Leistungsnachweis

---

- ▶ Für alle Studierenden aus den verschiedenen Studiengängen: Klausur
  - ▶ I. Termin Dienstag, 29.01.13, 14-16 Uhr
  - ▶ 2. Termin Dienstag, 26.03.13, 14-16 Uhr
- ▶ Welche Studiengänge sind vertreten?
- ▶ Übungen
  - ▶ Organisiert durch Michael Kuhn, Hermann Lenhart

# Arbeitsbereich Wissenschaftliches Rechnen

---

- ▶ Im Fachbereich für Informatik der Universität Hamburg seit Herbst 2009
- ▶ Leiter: Prof. Dr. Thomas Ludwig
  - ▶ Gleichzeitig Geschäftsführer des Deutschen Klimarechenzentrums
- ▶ Räumliche Unterbringung
  - ▶ Bundesstraße 45a
  - ▶ Keine Räume im Informatikum

# Lehrveranstaltungen im WS 2012/13

---

Wird ggf. WS 2013/14 wiederholt!

- ▶ Praktikum „Aufbau eines heterogenen Clusters“
- ▶ Projekt mit Seminar „Paralleles Programmieren“
- ▶ Projekt „Parallelrechnerevaluation“
- ▶ Proseminar „Architekturen von Hochleistungsrechnern“
- ▶ Seminar „Effiziente Programmierung in C“
- ▶ Seminar „Modellierung und Simulation“
- ▶ Vorlesung „Hochleistungsrechnen“
- ▶ Workshop „Programmierung mit C“
- ▶ Workshop „Versionsverwaltung mit Git“

Beachten Sie auch das Angebot für SS 2013!

# Mitarbeit in der Arbeitsgruppe

---

- ▶ Forschungsthemen
  - ▶ Speicherung großer Datenmengen
  - ▶ Simulation von Hochleistungsrechensystemen
  - ▶ Energieeffizienz von Hochleistungsrechnern
  - ▶ Anwendung in den Geowissenschaften

Wir betreuen:

- ▶ Bachelor-, Master-, Diplomarbeiten, Promotionen

Wir stellen potentiell ein:

- ▶ Studentische Hilfskräfte in der Arbeitsgruppe / am DKRZ
- ▶ Mitarbeiter in der Arbeitsgruppe / am DKRZ

# Hardware-Architekturen

---

- ▶ Parallelismus
- ▶ Klassifikation nach Flynn
- ▶ Erweiterung: die Sicht auf den Speicher
- ▶ Mehrprozessorsysteme mit verteiltem Speicher
- ▶ Mehrprozessorsysteme mit gemeinsamem Speicher
- ▶ Diskussion der beiden Ansätze
- ▶ Skalierbarkeit
- ▶ Verbindungsnetze und Topologien
- ▶ Betriebssystemaspekte

# Hardware-Architekturen

## Die zehn wichtigsten Fragen

- ▶ Auf welchen Ebenen finden wir Parallelismus ?
- ▶ Wie unterteilt Flynn die Rechnerarchitekturen ?
- ▶ Wie funktionieren Systeme mit verteiltem Speicher ?
- ▶ Wie funktionieren Systeme mit gemeinsamem Speicher?
- ▶ Welche Vor- und Nachteile haben die Ansätze ?
- ▶ Wie sind reale Systeme aufgebaut ?
- ▶ Welche Aufgabe hat das Verbindungsnetz und wie ist es strukturiert ?
- ▶ Welche Konzepte finden wir beim Hintergrundspeicher ?
- ▶ Welche Betriebssysteme finden wir bei HLR ?
- ▶ Welche weiteren Architekturen finden wir im Umfeld ?

# Parallelismus – Die Sache mit den Ochsen

---

*If you were plowing a field, what would you rather use?  
Two strong oxen or 1024 chickens?*

Seymour Cray (1925-1996)

*To pull a bigger wagon, it is easier to add more oxen than  
to grow a giant ox.*

W. Gropp, E. Lusk, A. Skjellum

- ▶ Wir erzielen höhere Leistung durch die parallele Nutzung leistungsschwächerer Einzelkomponenten
- ▶ Hochleistungsrechner sind immer Parallelrechner

# Ebenen des Parallelismus im Rechner

---

- ▶ Parallele Rechnerarchitekturen
  - ▶ Besitzen Verarbeitungseinheiten die koordiniert gleichzeitig an einer Aufgabe arbeiten
- ▶ Verarbeitungseinheiten
  - ▶ Spezialisierte Einheiten wie z.B. Pipelines
  - ▶ Gleichartige Rechenwerke
  - ▶ Prozessorkerne
  - ▶ Prozessoren
  - ▶ Vollständige Rechner
  - ▶ Hochleistungsrechnersysteme

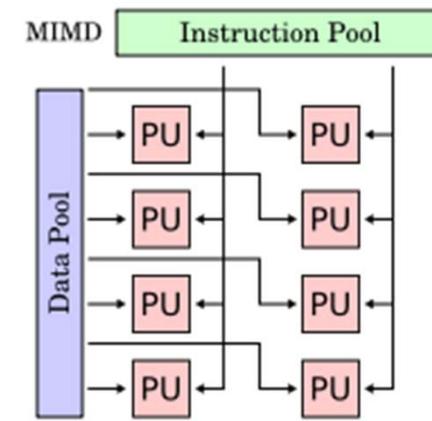
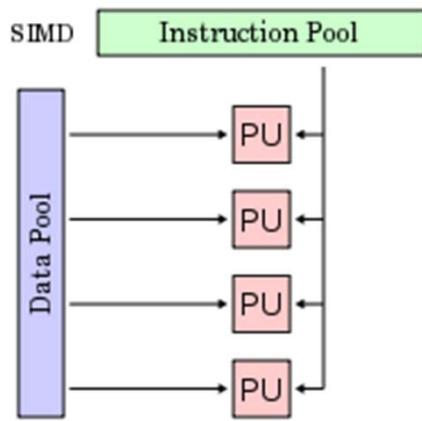
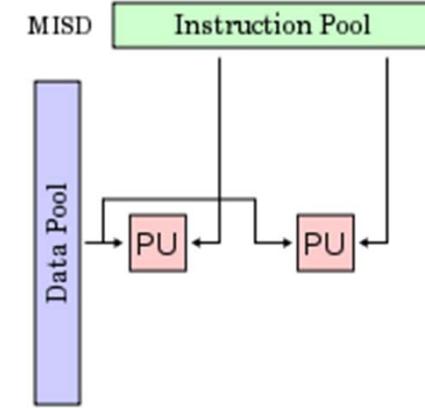
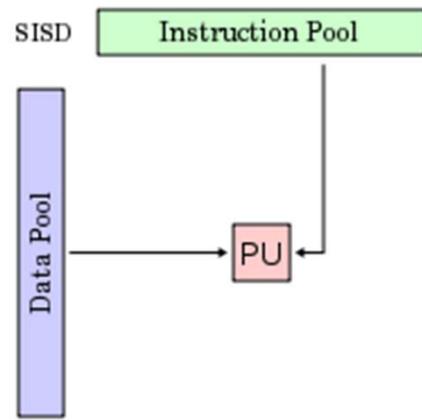
# Flynn'sche Klassifikation

---

## Klassifikation nach Flynn (1972)

- ▶ Rechner arbeiten mit Befehlsströmen und Datenströmen
  - ▶ Aus ihrer Kombination ergeben sich 4 Varianten
- 
- ▶ SISD single instruction, single data stream
  - ▶ SIMD single instruction, multiple data stream
  - ▶ MISD multiple instruction, single data stream
  - ▶ MIMD multiple instruction, multiple data stream

# Flynnnsche Klassifikation...



# Flynn'sche Klassifikation...

Was ist was bei Flynn?

- ▶ SISD: klassische von-Neumann-Architektur Monoprozessor-Rechner
- ▶ SIMD: Vektorrechner und Feldrechner
- ▶ MISD: diese Klasse ist leer
- ▶ MIMD: alles, was uns interessiert: die Mehrprozessorsysteme

Wir müssen die Klasse der MIMD-Rechner weiter aufgliedern

# Unterteilung von Flynns MIMD-Klasse

---

## Unterteilung der Flynn'schen MIMD-Systeme

Die Rechner bestehen aus mehreren Prozessoren, die über ein Verbindungsnetz kommunizieren

Über die Verbindungen erfolgt der Informationsaustausch zwischen Prozessen auf verschiedenen Prozessoren sowie Synchronisation und Kooperation

# Unterteilung von Flynns MIMD-Klasse...

---

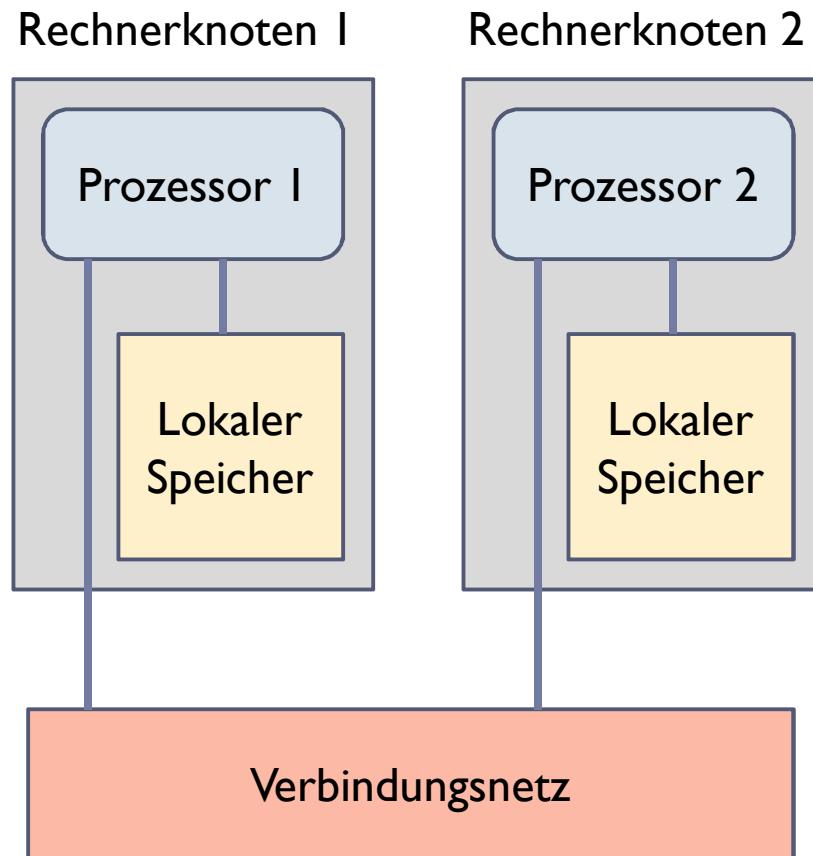
## Neue Unterscheidungskriterien

- ▶ Wie sehen die Prozessoren den Adreßraum des Speichers?
- ▶ Wie sind die Speicherkomponenten mit dem Prozessor gekoppelt?

## Neue Klassen

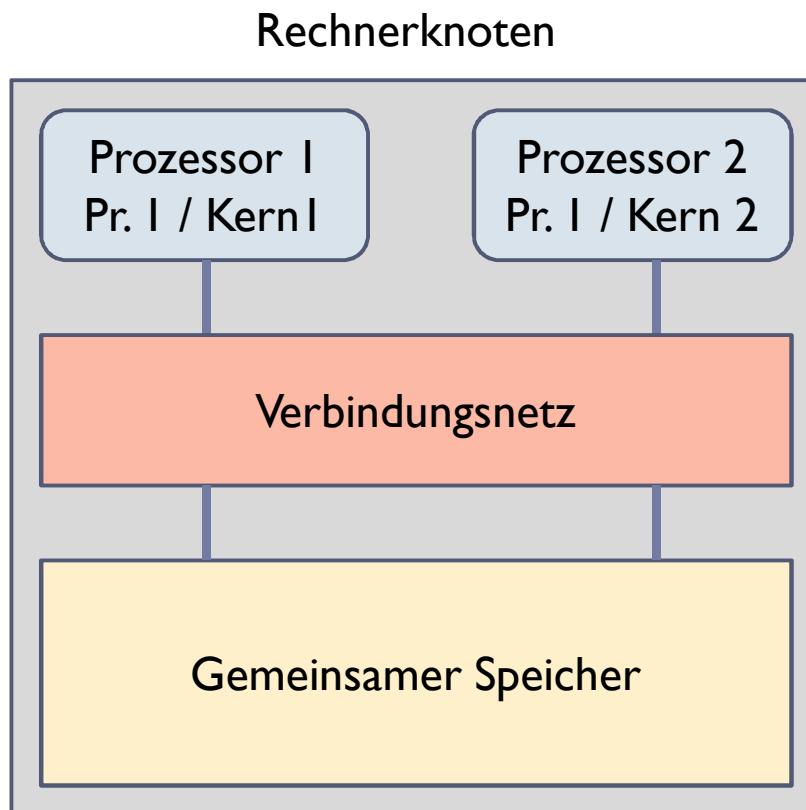
- ▶ Rechner mit verteiltem Speicher
- ▶ Rechner mit gemeinsamem Speicher
- ▶ Mischformen

# Mehrprozessorsysteme mit verteiltem Speicher



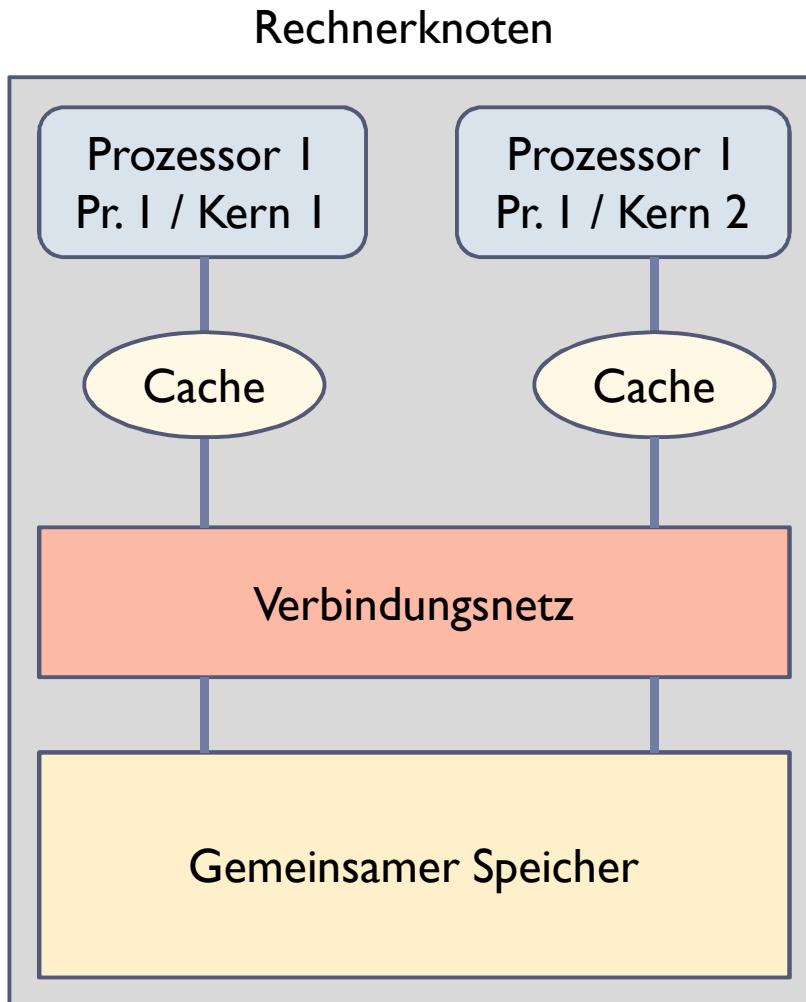
- ▶ Prozesse sehen nur den Adressraum im lokalen Speicher
- ▶ Leistungssteigerung: Dasselbe Programm läuft parallel auf allen Prozessoren; seine Daten sind auf die lokalen Speicher der Rechnerknoten aufgeteilt

# Mehrprozessorsysteme mit gemeinsamem Speicher



- ▶ Jeder Prozeß sieht den gesamten Adressraum des gemeinsamen Speichers
- ▶ Leistungssteigerung: Dasselbe Programm läuft parallel auf allen Prozessoren; seine Daten sind auf im gemeinsamen Speicher für alle zugreifbar

# Mehrprozessorsysteme mit gemeinsamem Speicher und Cache (!)



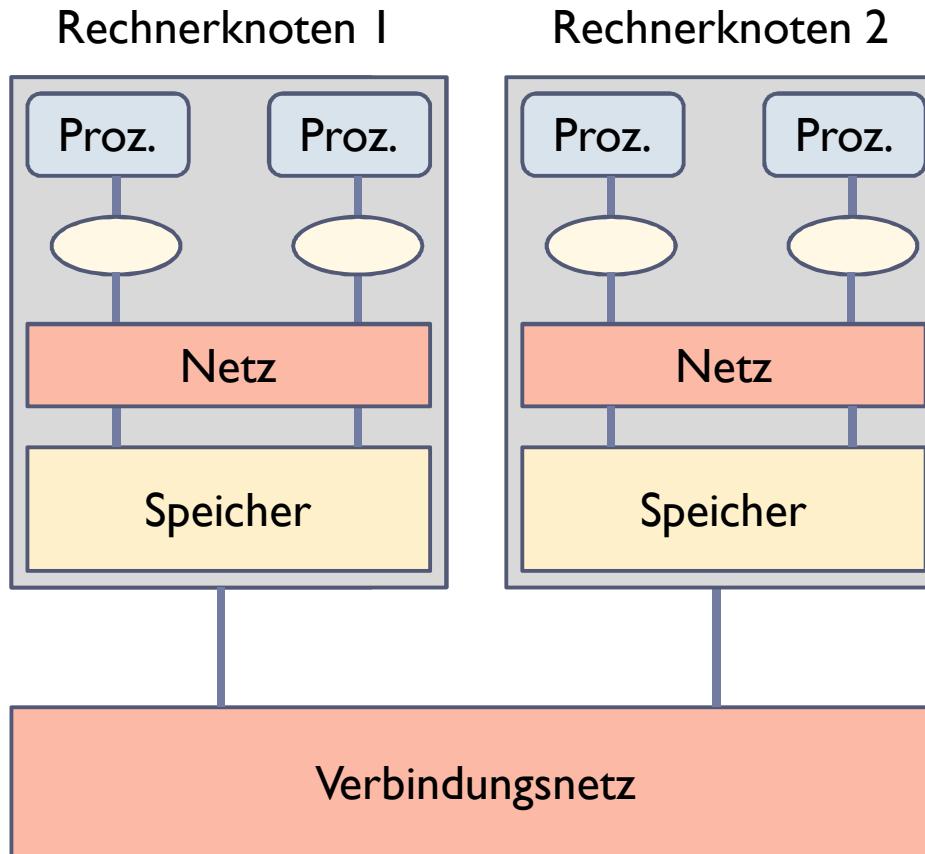
- ▶ In der Realität immer auch mehrstufige Cache-Speicher
- ▶ Sehr komplex mit Konsistenz und Kohärenz
- ▶ Neue Fragen der Prozessorzuteilung treten auf (Scheduling)

# Vor- und Nachteile der Ansätze

---

- ▶ Mehrprozessorsysteme mit verteiltem Speicher
  - ▶ Hohe Ausbaubarkeit (100.000+ Prozessoren)
  - ▶ Komplexe Programmierung (Nachrichtenaustausch)
- ▶ Mehrprozessorsysteme mit gemeinsamem Speicher
  - ▶ Geringe Ausbaubarkeit (einige dutzend Prozessorkerne oder Prozessoren)
  - ▶ „Einfachere“ Programmierung (Verwendung gemeinsamer Speicherbereiche)

# Reale Systeme: Alles in Einem



- ▶ Existierende HLR sind heute meist eine Kombination aus Rechnerknoten mit gemeinsamem Speicher, von den man viele verwendet und über ein Verbindungsnetz verbindet

# Weitere Bezeichnungen

---

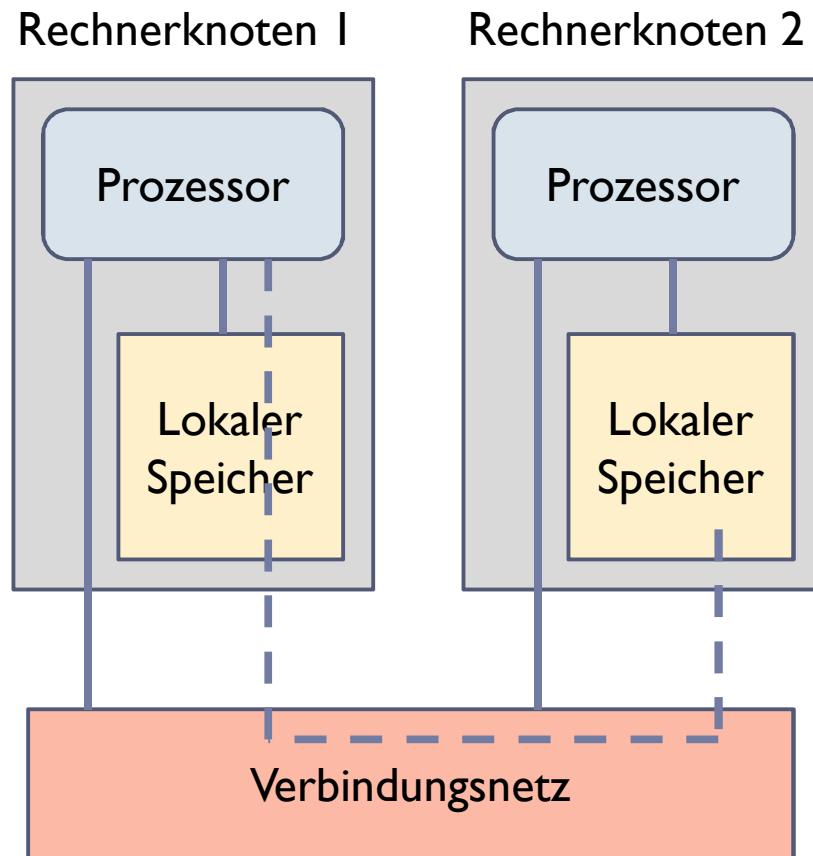
## Verteilter Speicher

- ▶ Multicomputersystem
- ▶ Schwache Kopplung
- ▶ Lose Kopplung
- ▶ Massiv paralleles System
- ▶ MPP – massive parallel processing

## Gemeinsamer Speicher

- ▶ Multiprozessorsystem
- ▶ Enge Kopplung
- ▶ SMP – symmetric multiprocessing

# Mischform: Verteilter gemeinsamer Speicher



- ▶ Logisch sieht jeder Prozeß den gesamten Adreßraum aller aggregierten lokalen Speicher
- ▶ Physikalisch ist der Adreßraum verteilt
- ▶ Bereitstellung durch Hardware und/oder Software
- ▶ Bezeichnung auch: DSM (distributed shared memory)

# Modellierung bzgl. Zugriffszeiten

---

- ▶ UMA: uniform memory access model
  - ▶ Gemeinsamer Speicher
- ▶ (cc)NUMA: (cache coherent) non uniform memory access model
  - ▶ Verteilter gemeinsamer Speicher (mit Cache-Kohärenz)
- ▶ NORMA: no remote memory access model
  - ▶ Verteilter Speicher
- ▶ (N)UCA: (non) uniform communication architecture model
  - ▶ Nicht uniform: z.B. Cluster von SMP-Maschinen

# Der Begriff Skalierbarkeit

---

„Skalierbarkeit“ nirgends eindeutig definiert, aber der wohl am häufigsten benutzte Begriff beim Hochleistungsrechnen

Gemeint ist: Ausbaubarkeit unter Beibehaltung gewisser positiver Charakteristika

- ▶ Z.B. Ein Programm skaliert gut, wenn es bei großer Prozeßzahl noch hohe Leistung bringt
- ▶ Ein Netz skaliert gut, wenn beim Ausbau die Leistung mit dem investierten Geld korreliert

# Verbindungsnetze

---

- ▶ Im einfachsten Fall
  - ▶ Gemeinsamer Speicher: Bussystem
  - ▶ Verteilter Speicher: Sterntopologie mit Switch
- ▶ Im komplexen Fall
  - ▶ Alle Varianten, jedoch keine Vollvermaschung

## Probleme

- ▶ Latenzzzeiten, Übertragungszeiten
- ▶ Netzbelastung, Kollisionen

# Beispiele von Verbindungsnetzen

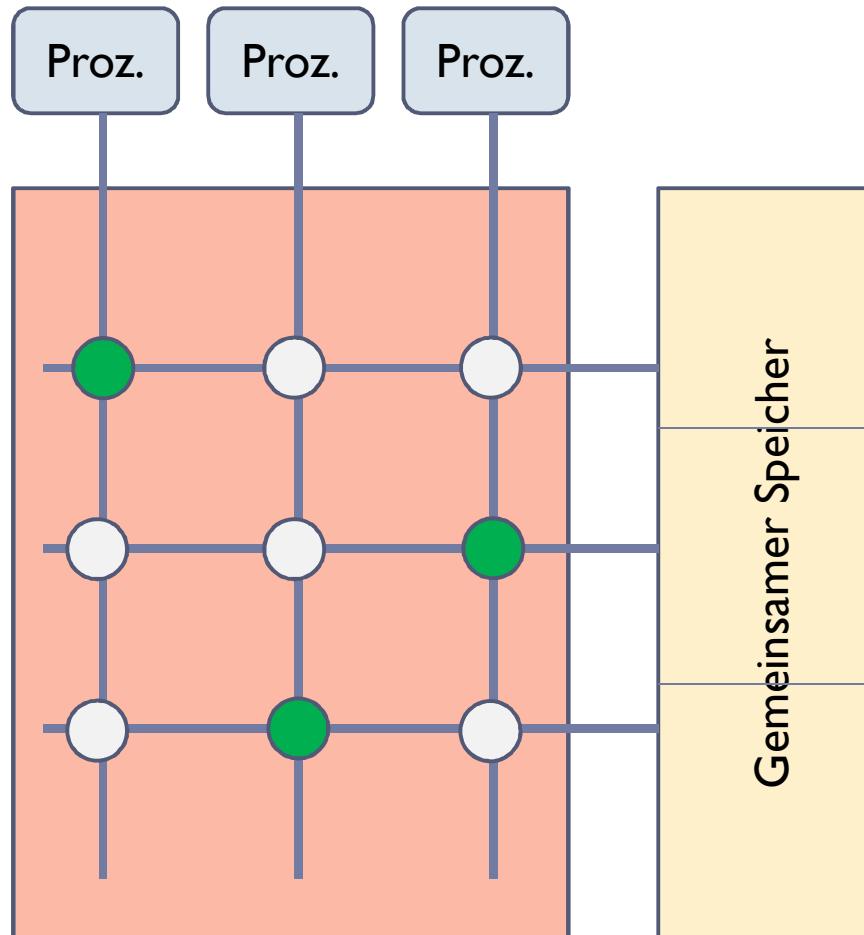
---

Es gibt hier eine Vielzahl von Konzepten !

Wir greifen drei davon zur Illustration heraus:

- ▶ Kreuzschienenverteiler
- ▶ Zweidimensionaler Torus
- ▶ Hypercube

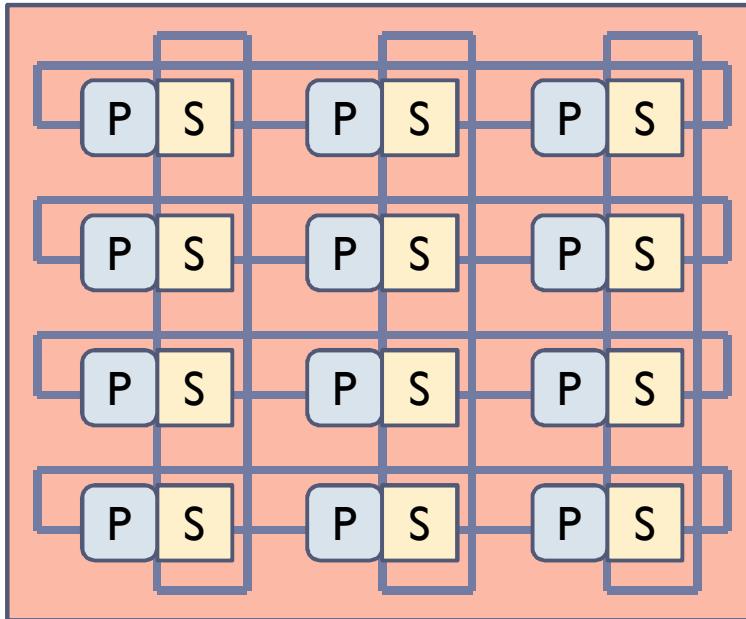
# Verbindungsnetz bei gemeinsamem Speicher



3x3-Kreuzschienenverteiler

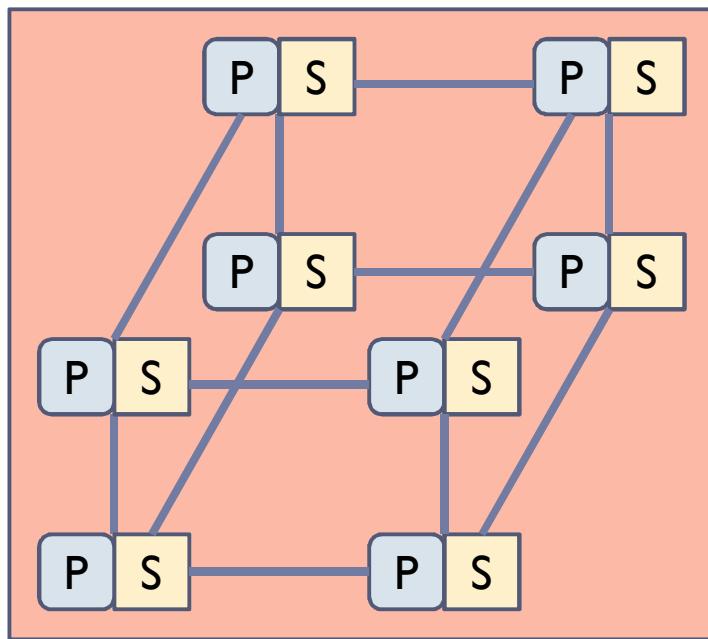
- ▶ Kreuzschienenverteiler  
 $n \times m$
- ▶ Im günstigsten Fall wie ein m-Bus-System
- ▶ Hoher technischer Aufwand
- ▶ Reduktion der Konflikte auf dem Bus

# Verbindungsnetz bei verteiltem Speicher (1)



- ▶ Zweidimensionaler Torus/Array
- ▶ Konstante Nachbarschaft, deshalb beliebig erweiterbar
- ▶ Entfernungsabhängige Übertragungszeiten
- ▶ Knotenzahl verdoppelt, maximaler Pfad wächst stark an

# Verbindungsnetz bei verteiltem Speicher (2)



- ▶ Hypercube (n-dimen.  
Binärer Würfel)
- ▶ #Nachbarn = Dimension
- ▶ Kurze maximale  
Entfernung
- ▶ Hoher Grad der  
Vernetzung
- ▶ Knotenzahl doppelt,  
maximaler Pfad wächst  
um eins

# Hintergrundspeicher

---

- ▶ Lokale Platte an jedem Rechnerknoten
  - ▶ Heute meist nur zur Zwischenspeicherung
- ▶ Dateiserver ins Netz eingebunden
  - ▶ Persistente Datenhaltung
  - ▶ Flaschenhals bei Datenzugriff
- ▶ Storage Area Network (SAN)
  - ▶ Speicherkomponenten mit eigenem Netz an die Komponenten des Clusters angehängt
- ▶ Bandarchive

Ein-/Ausgabe war bisher vernachlässigte Fragestellung  
– jetzt intensiver untersucht

# Betriebssysteme

---

- ▶ Betriebssysteme für Hochleistungsrechner
  - ▶ Auf den Rechnerknoten:  
Fast immer Unix-Derivate (meist Linux), selten Windows
  - ▶ Über alle Knoten hinweg  
Zusatzsoftware, die den Verbund nutzbar macht  
(nicht in das Betriebssystem integriert)
- ▶ Betriebssystemforschung
  - ▶ Single System Image (SSI)
    - ▶ Das System erscheint dem Anwender wie ein System mit einem Knoten
    - ▶ Lokalisierung von Diensten verborgen

# Spezialkonzept: Rechnercluster

---

- ▶ Cluster of workstations (COW)
- ▶ Network of workstations (NOW)
- ▶ Beowulf cluster (Sterling et al.)
  - ▶ Nur Standardkomponenten (commodity of the shelf components, COTS)  
Pentium, Ethernet, Linux

## Der Arme-Leute-Parallelrechner

# Spezialkonzept: Grid

---

- ▶ Jederzeit verfügbare (hohe) Rechenleistung
  - ▶ Vergleichbar zu Elektrizität heute
- ▶ Netz von Hochleistungsrechnern

Der Superrechner des reichen Mannes ☹

Konzept kam nie so richtig zum Fliegen trotz vieler Millionen von Forschungsmitteln weltweit

# Spezialkonzept: Cloud

---

- ▶ Jederzeit verfügbare (hohe) Leistung zum Rechnen, Speichern, Programmenutzen ...
- ▶ Netz von IT-Komponenten

Der Rechner für die Zukunft ?

2020: Konzept kam nie so richtig zum Fliegen trotz vieler Millionen von Forschungsmitteln weltweit ?

# Abgrenzungen

|                                  | Verteilter Speicher   | Gemeinsamer Speicher | Cluster             | Verteiltes System            |
|----------------------------------|-----------------------|----------------------|---------------------|------------------------------|
| #Prozessoren                     | $O(100) - O(100.000)$ | $O(10)$              | $O(10) - O(10.000)$ | $O(10) - O(1000)$            |
| Kommunikation zwischen Prozessen | Nachrichten           | Gemeinsame Variable  | Nachrichten         | RPC, Nachrichten, Middleware |
| Single System Image              | Selten                | Immer                | Selten              | Nie                          |
| Betriebssystem                   | Sparversion           | SMP-BS               | Unix homogen        | Verschiedene heterogen       |
| Besitzer                         | Einer                 | Einer                | Einer oder mehrere  | Mehrere                      |

# Hardware-Architekturen

## Zusammenfassung

- ▶ Erste wichtige Begriffsbildung durch Flynn
- ▶ Wir unterscheiden Architekturen mit verteiltem und mit gemeinsamem Speicher
- ▶ Die Skalierbarkeit ist bei verteiltem Speicher sehr hoch, dafür erschwert sich die Programmierbarkeit
- ▶ Reale Hochleistungsrechner sind meist viele vernetzte Rechnerknoten mit jeweils gemeinsamem Speicher und Mehrkernprozessoren
- ▶ Verbindungsnetze gibt es mit vielen Topologien
- ▶ Speichersysteme nutzen ebenfalls Parallelität
- ▶ Als Betriebssystem kommt meist Linux zum Einsatz

# Die TOP500-Liste

---

- ▶ Vergleich von Rechnersystemen
- ▶ Die TOP500-Liste
- ▶ Details vom November 2009
- ▶ Historische Entwicklung wichtiger Aspekte
- ▶ Leistungsentwicklung
- ▶ Ausgewählte Systeme
- ▶ Ein Blick zurück

# Die TOP500-Liste

## Die zehn wichtigsten Fragen

- ▶ In welcher Einheit wird die Leistung angegeben?
- ▶ Wie wird die Leistung evaluiert?
- ▶ Welche Zielsetzung verfolgt das TOP500-Projekt?
- ▶ In welchen Größenordnungen liegen die stärksten Rechner?
- ▶ Welche Hersteller dominieren wie den Markt?
- ▶ Welches sind aktuelle Anwendergebiete?
- ▶ Welche Prozessorenarchitekturen und –familien dominieren?
- ▶ Welche Verbindungstechnologien dominieren in welchem Bereich?
- ▶ Wie verhält sich die Leistungssteigerung zu Moore´s Law?
- ▶ Welche Leistung brachten Systeme 1993?

# Vergleich von Rechnersystemen

---

## Komplexe Fragestellung

- ▶ Erster Ansatz: FLOPS (floating point operations per second)
- ▶ Theoretisches Maximum ergibt sich aus der Anzahl der Zyklen pro Gleitkommaoperation

## Bewertung durch sogenannte Benchmark-Programme

- ▶ Synthetische Benchmarks (meist Assembler)
- ▶ CPU-Benchmark (meist numerische Programme)

# Zum Vergleich: Prozessoren

Beispiele der GFLOP-Werte an einigen CPUs<sup>[1]</sup>

| Linpack 1kx1k (DP)         | Peak GigaFLOPS | Actual GigaFLOPS                 | Effizienz (in %) |
|----------------------------|----------------|----------------------------------|------------------|
| Cell, 1 SPU, 3,2 GHz       | 1,83           | 1,45                             | 79,23            |
| Cell, 8 SPUs, 3,2 GHz      | 14,63          | 9,46                             | 64,66            |
| Pentium 4, 3,2 GHz         | 6,4            | 3,1                              | 48,44            |
| Pentium 4 + SSE3, 3,6 GHz  | 14,4           | 7,2                              | 50,00            |
| Core i7, 3,2 GHz, 4 Kerne  | 51,2           | 33,0 (HT enabled) <sup>[2]</sup> | 64,45            |
| Core i7, 3,33 GHz, 6 Kerne | 80             |                                  |                  |
| Itanium, 1,6 GHz           | 6,4            | 5,95                             | 92,97            |

# Zum Vergleich: Anwendungen

---

- ▶ Rechnersystem Blizzard am DKRZ 2010
  - ▶ 110 TFLOPS LINPACK-Leistung
  - ▶ Bei ca. 8000 Prozessoren macht das ca. 13 GFLOPS/Prozessor
- ▶ Klimaberechnung IPCC AR5
  - ▶ Geschätzter Bedarf: 30 Millionen Prozessorstunden am DKRZ
  - ▶ Ca. 50 Tflop pro Prozessorstunde

# Vergleich von Rechnersystemen...

## Der parallele LINPACK-Benchmark

- ▶ Entwickelt von Jack Dongarra (Knoxville, TN)
- ▶ Ist gleichzeitig eine vollwertige Bibliothek für lineare Algebra
- ▶ Benchmark: dicht besetztes Gleichungssystem
- ▶  $R_{\max}$  ist maximale Leistung bei Problemgröße  $N_{\max}$
- ▶ ( $N_{1/2}$  ist Problemgröße bei Leistung  $R_{1/2}$ )
- ▶  $R_{\text{peak}}$  ist die theoretische Maximalleistung

# Die TOP500-Liste

---

Website [www.top500.org](http://www.top500.org)

- ▶ Hans Meuer (Universität Mannheim)
- ▶ Jack Dongarra (Univ.Tennessee, Knoxville)
- ▶ Erich Strohmeier (NERSC/LBNL)
- ▶ Horst Simon (NERSC/LBNL)

Zwei Aktualisierungen pro Jahr

- ▶ Juni: International Supercomputing Conference Deutschland
- ▶ November: Supercomputing Conference USA

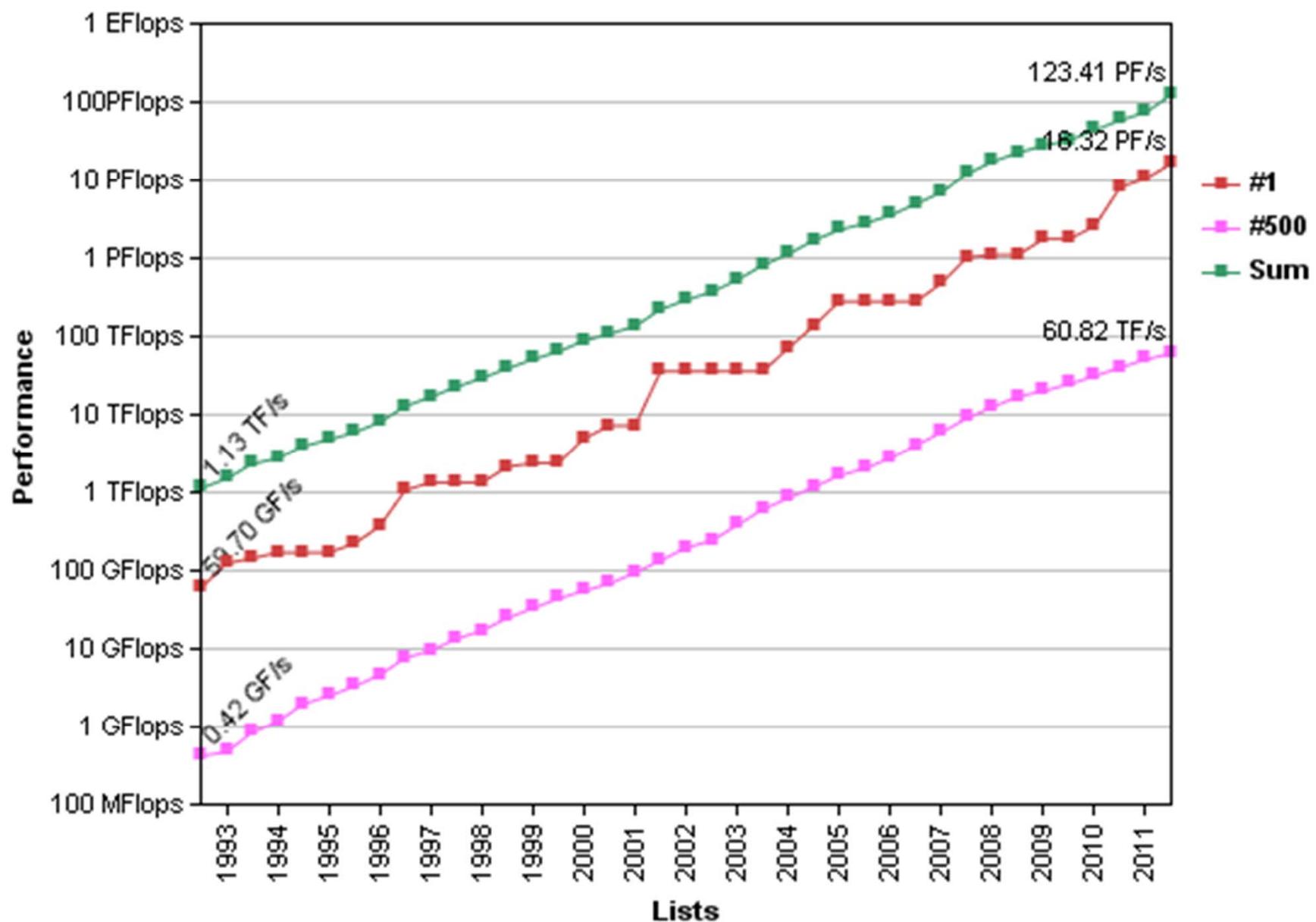
Basiert auf dem LINPACK-Benchmark

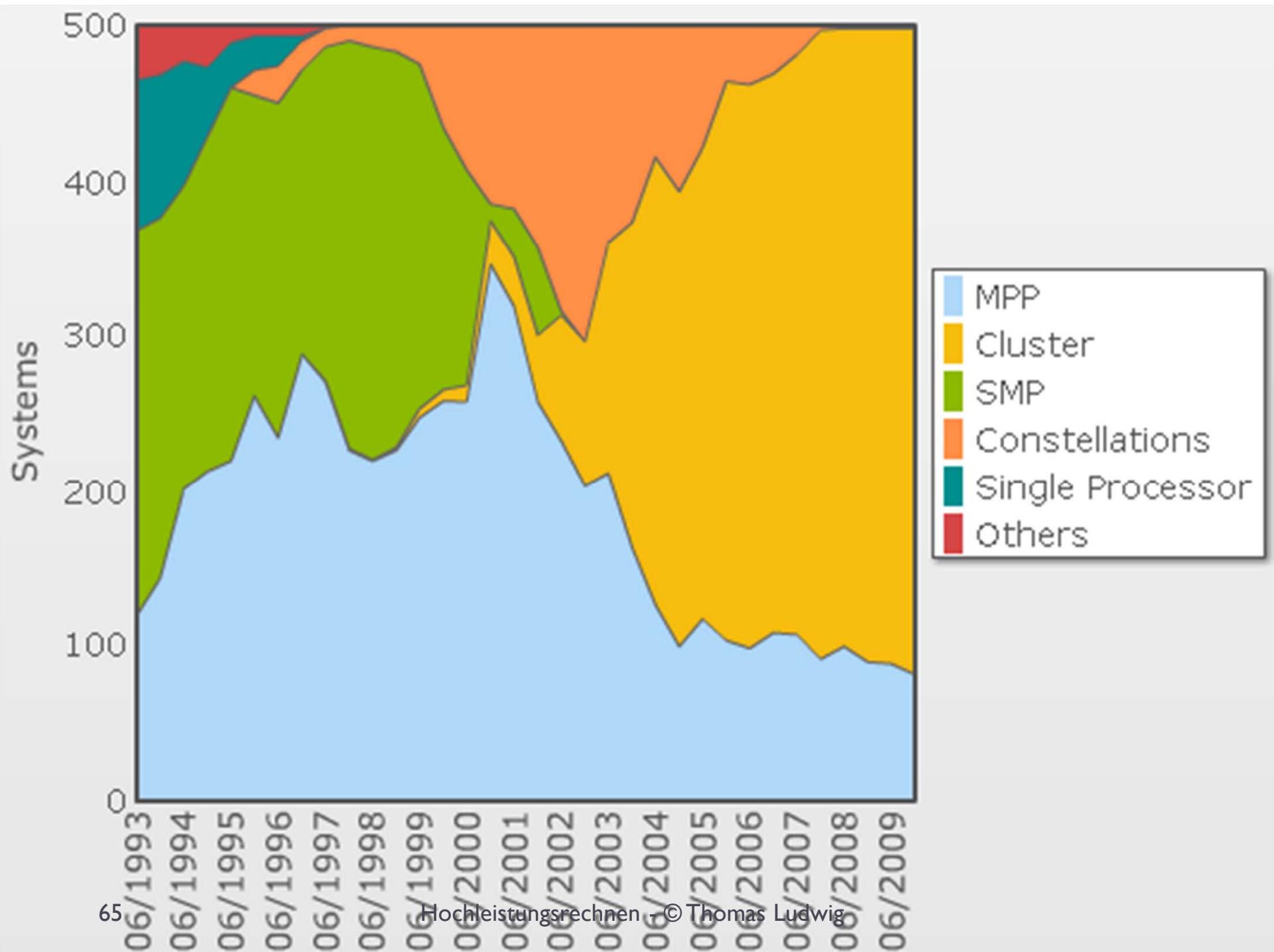
| Rank | Site   | Computer/Year Vendor  | Cores   | R <sub>max</sub> | R <sub>peak</sub> | Power   |
|------|--|---|---------|------------------|-------------------|---------|
| 1    | DOE/NNSA/LLNL<br>United States                                     | Sequoia - BlueGene/Q, Power BQC 16C<br>1.60 GHz, Custom / 2011<br>IBM                                   | 1572864 | 16324.75         | 20132.66          | 7890.0  |
| 2    | RIKEN Advanced Institute for Computational Science (AICS)<br>Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011<br>Fujitsu                                  | 705024  | 10510.00         | 11280.38          | 12659.9 |
| 3    | DOE/SC/Argonne National Laboratory<br>United States                | Mira - BlueGene/Q, Power BQC 16C<br>1.60GHz, Custom / 2012<br>IBM                                       | 786432  | 8162.38          | 10066.33          | 3945.0  |
| 4    | Leibniz Rechenzentrum<br>Germany                                   | SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012<br>IBM                     | 147456  | 2897.00          | 3185.05           | 3422.7  |
| 5    | National Supercomputing Center in Tianjin<br>China                 | Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010<br>NUDT                             | 186368  | 2566.00          | 4701.00           | 4040.0  |
| 6    | DOE/SC/Oak Ridge National Laboratory<br>United States              | Jaguar - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009<br>Cray Inc. | 298592  | 1941.00          | 2627.61           | 5142.0  |
| 7    | CINECA<br>Italy  | Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012<br>IBM   | 163840  | 1725.49          | 2097.15           | 821.9   |
| 8    | Forschungszentrum Juelich (FZJ)<br>Germany                         | JuQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012<br>Hochleistungsrechner © Thomas Ludwig IBM  | 131072  | 1380.39          | 1697.12           | 657.5   |

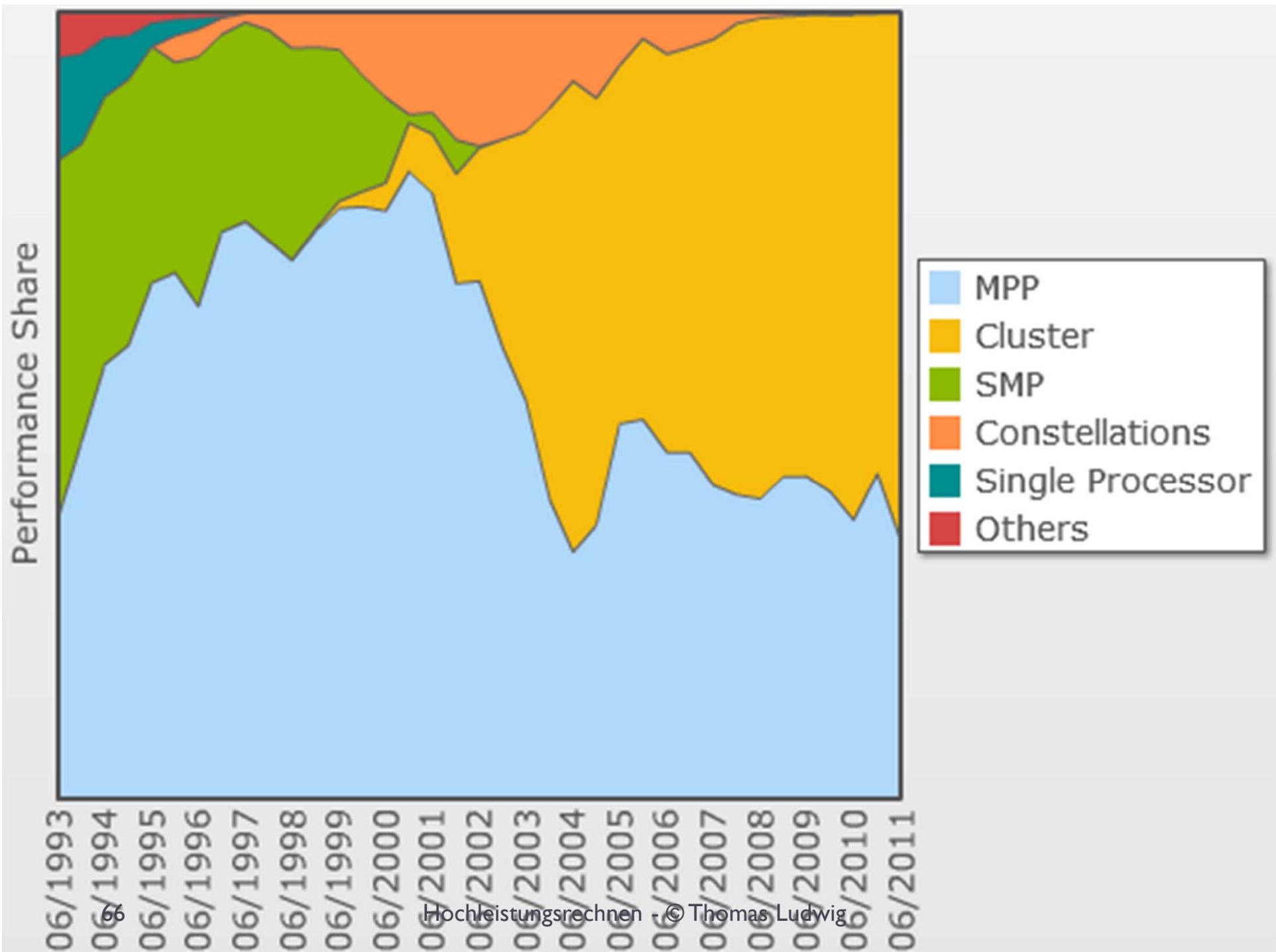
|    |  |  |        |         |         |        |
|----|--|--|--------|---------|---------|--------|
| 9  | CEA/TGCC-GENCI<br>France   | <b>Curie thin nodes</b> - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012<br>Bull  | 77184  | 1359.00 | 1667.17 | 2251.0 |
| 10 | National Supercomputing Centre in Shenzhen (NSCS)<br>China   | <b>Nebulae</b> - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010<br>Dawning                         | 120640 | 1271.00 | 2984.30 | 2580.0 |
| 11 | NASA/Ames Research Center/NAS<br>United States   | <b>Pleiades</b> - SGI Altix ICE X/8200EX /8400EX, Xeon 54xx 3.0/5570 /5670/E5-2670 2.93/2.6/3.06/3.0 Ghz, Infiniband QDR/FDR / 2011<br>SGI | 125980 | 1243.00 | 1731.84 | 3987.0 |
| 12 | International Fusion Energy Research Centre (IFERC), EU(F4E) - Japan Broader Approach collaboration<br>Japan | <b>Helios</b> - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2011<br>Bull  | 70560  | 1237.00 | 1524.10 | 2200.0 |
| 13 | Science and Technology Facilities Council - Daresbury Laboratory<br>United Kingdom                           | <b>Blue Joule</b> - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012<br>IBM  | 114688 | 1207.84 | 1468.01 | 575.3  |
| 14 | GSIC Center, Tokyo Institute of Technology<br>Japan  | <b>TSUBAME 2.0</b> - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010<br>NEC/HP                                       | 73278  | 1192.00 | 2287.63 | 1398.6 |
| 15 | DOE/NNSA/LANL/SNL<br>United States   | <b>Cielo</b> - Cray XE6, Opteron 6136 8C 2.40GHz, Custom / 2011<br>Cray Inc.   | 142272 | 1110.00 | 1365.81 | 3980.0 |
| 16 | DOE/SC/LBNL/NERSC<br>United States   | <b>Hopper</b> - Cray XE6, Opteron 6172 12C 2.10GHz, Custom / 2010<br>Hochleistungsrechner © Thomas Ludwig Cray Inc.                        | 153408 | 1054.00 | 1288.63 | 2910.0 |

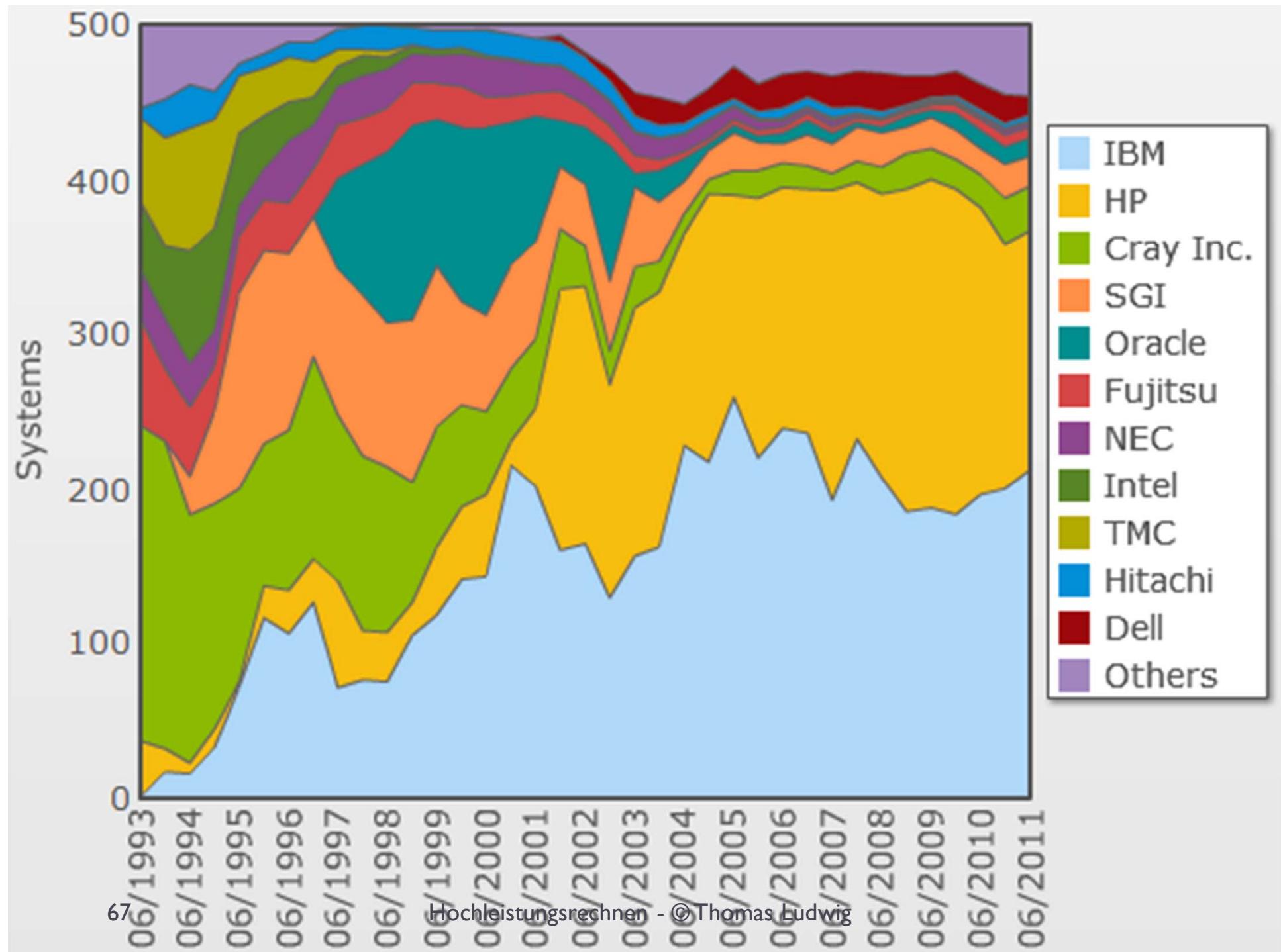
|     |   |   |        |        |        |        |
|-----|---|---|--------|--------|--------|--------|
| 4   | <a href="#">Leibniz Rechenzentrum<br/>Germany</a>                   | SuperMUC - iDataPlex DX360M4, Xeon E5-2680<br>8C 2.70GHz, Infiniband FDR<br>IBM   | 147456 | 2897.0 | 3185.1 | 3422.7 |
| 8   | <a href="#">Forschungszentrum Juelich<br/>(FZJ)<br/>Germany</a>     | JuQUEEN - BlueGene/Q, Power BQC 16C<br>1.60GHz, Custom<br>IBM   | 131072 | 1380.4 | 1677.7 | 657.5  |
| 24  | <a href="#">HWW/Universitaet Stuttgart<br/>Germany</a>              | HERMIT - Cray XE6, Opteron 6276 16C 2.30<br>GHz, Cray Gemini interconnect<br>Cray Inc.  | 113472 | 831.4  | 1043.9 |        |
| 25  | <a href="#">Forschungszentrum Juelich<br/>(FZJ)<br/>Germany</a>     | JUGENE - Blue Gene/P Solution<br>IBM  | 294912 | 825.5  | 1002.7 | 2268   |
| 59  | <a href="#">Universitaet Frankfurt<br/>Germany</a>                  | LOEWE-CSC - Supermicro Cluster, QC Opteron<br>2.1 GHz, ATI Radeon GPU, Infiniband<br>Clustervision/Supermicro                                 | 16368  | 299.3  | 508.5  | 416.8  |
| 63  | <a href="#">Forschungszentrum Juelich<br/>(FZJ)<br/>Germany</a>     | JUROPA - Sun Constellation, NovaScale<br>R422-E2, Intel Xeon X5570, 2.93 GHz, Sun<br>M9/Mellanox QDR Infiniband/Partec Parastation<br>Bull SA | 26304  | 274.8  | 308.3  | 1549   |
| 78  | <a href="#">Universitaet Aachen/RWTH<br/>Germany</a>                | RWTH Compute Cluster (RCC) - Bullx B500<br>Cluster, Xeon X56xx 3.06Ghz, QDR Infiniband<br>Bull SA   | 25448  | 219.8  | 270.5  |        |
| 81  | <a href="#">Universitaet Mainz<br/>Germany</a>                      | MOGON - Saxonid 6100, Opteron 6272 16C<br>2.100GHz, Infiniband QDR<br>Megware   | 33792  | 205.0  | 283.9  | 432    |
| 87  | <a href="#">Max-Planck-Gesellschaft<br/>MPI/IPP<br/>Germany</a>     | iDataPlex DX360M4, Xeon E5-2670 8C<br>2.600GHz, Infiniband FDR<br>IBM   | 9904   | 187.4  | 206.0  | 205.7  |
| 152 | <a href="#">DKRZ - Deutsches<br/>Klimarechenzentrum<br/>Germany</a> | Power 575, p6 4.7 GHz, Infiniband<br>IBM  | 8064   | 115.9  | 151.6  | 1288   |

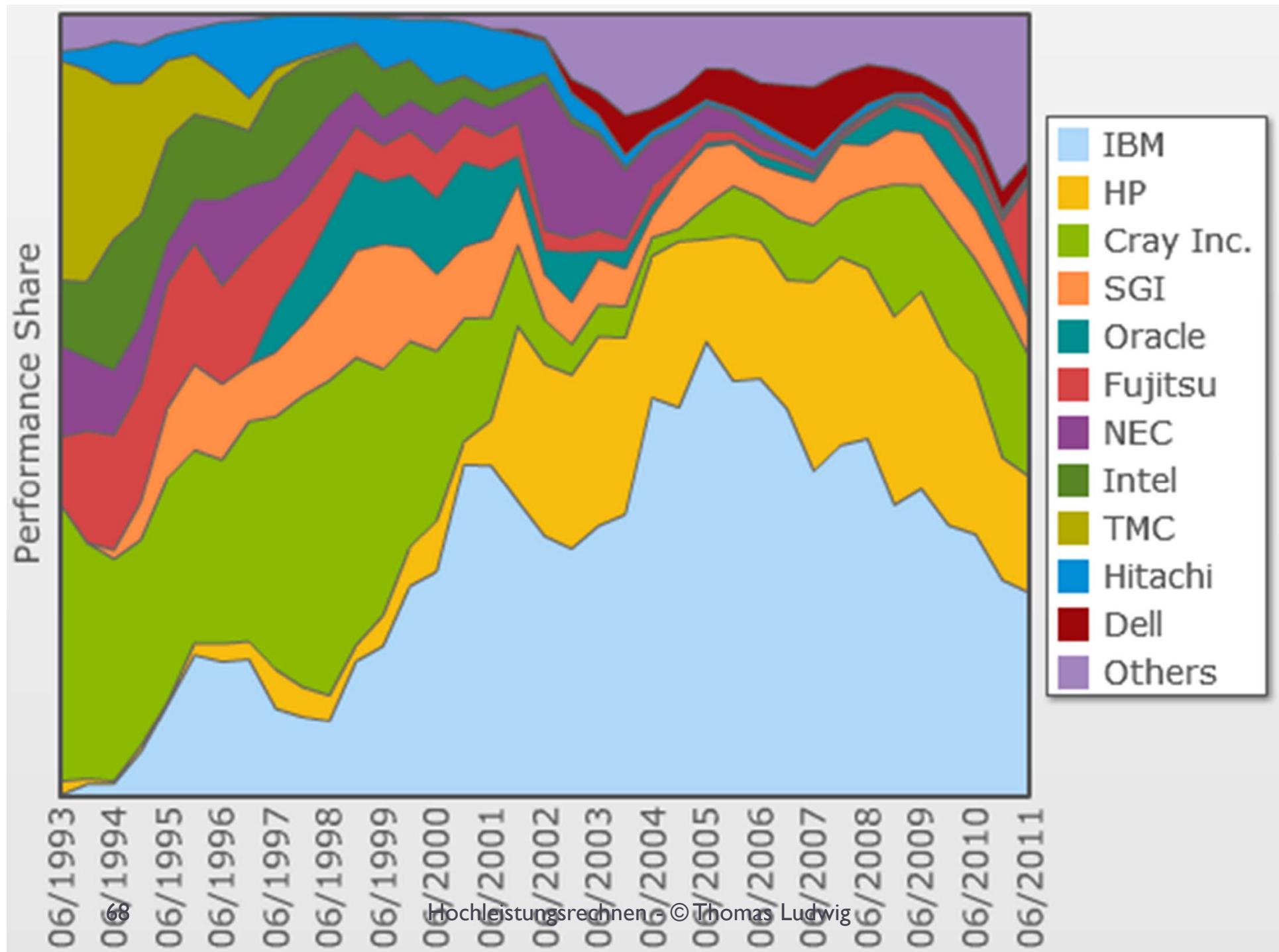
## Performance Development

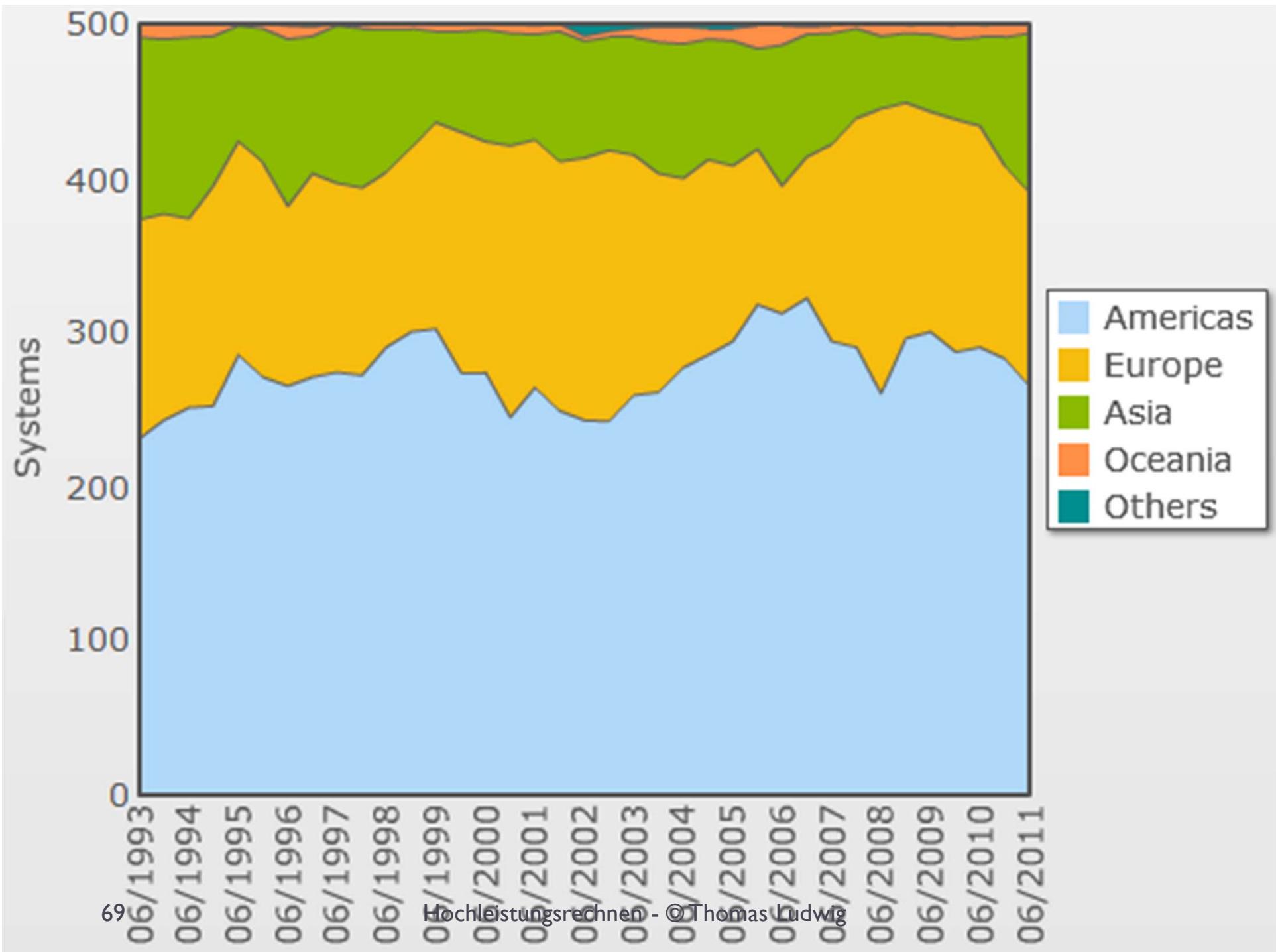


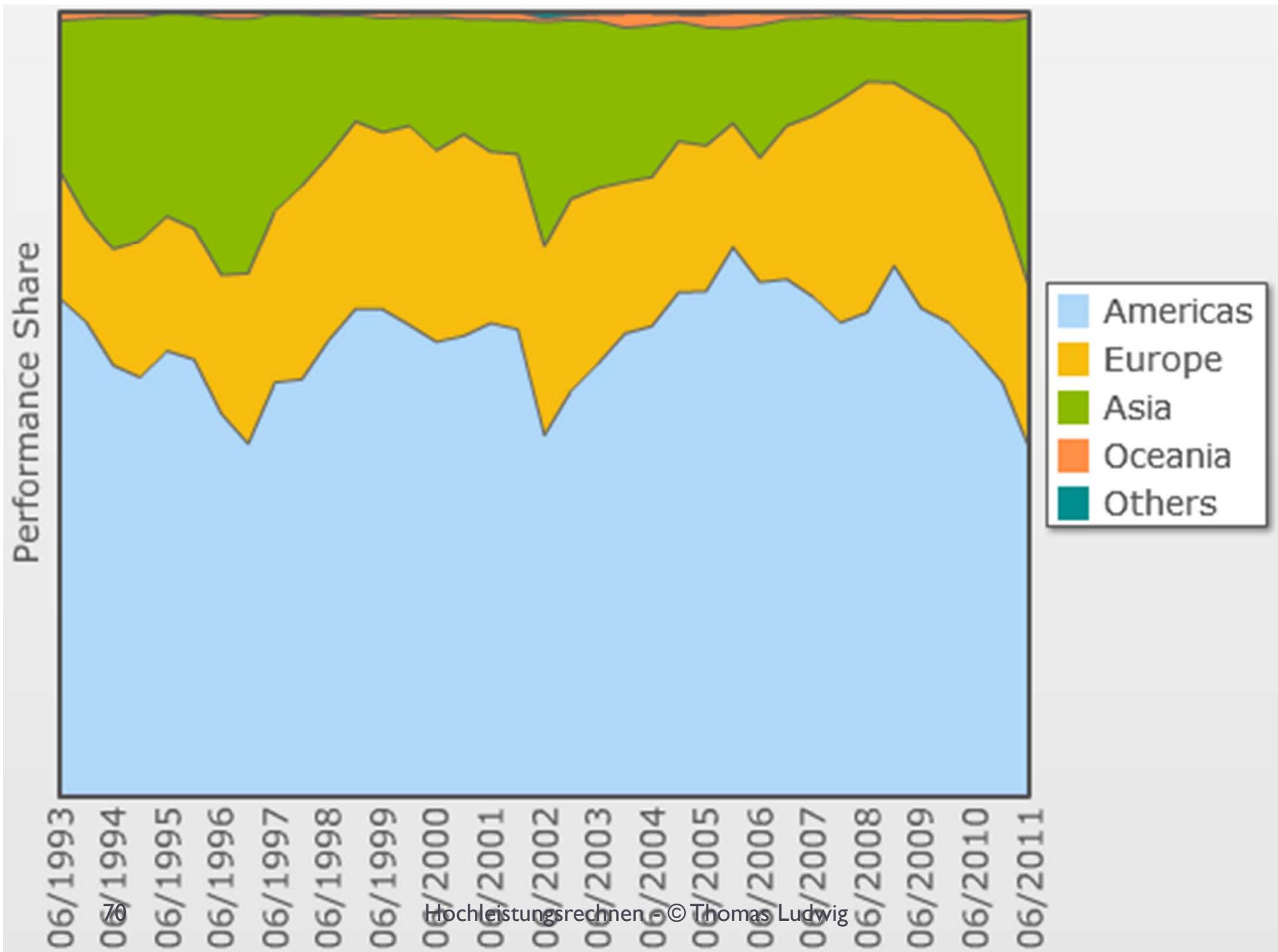


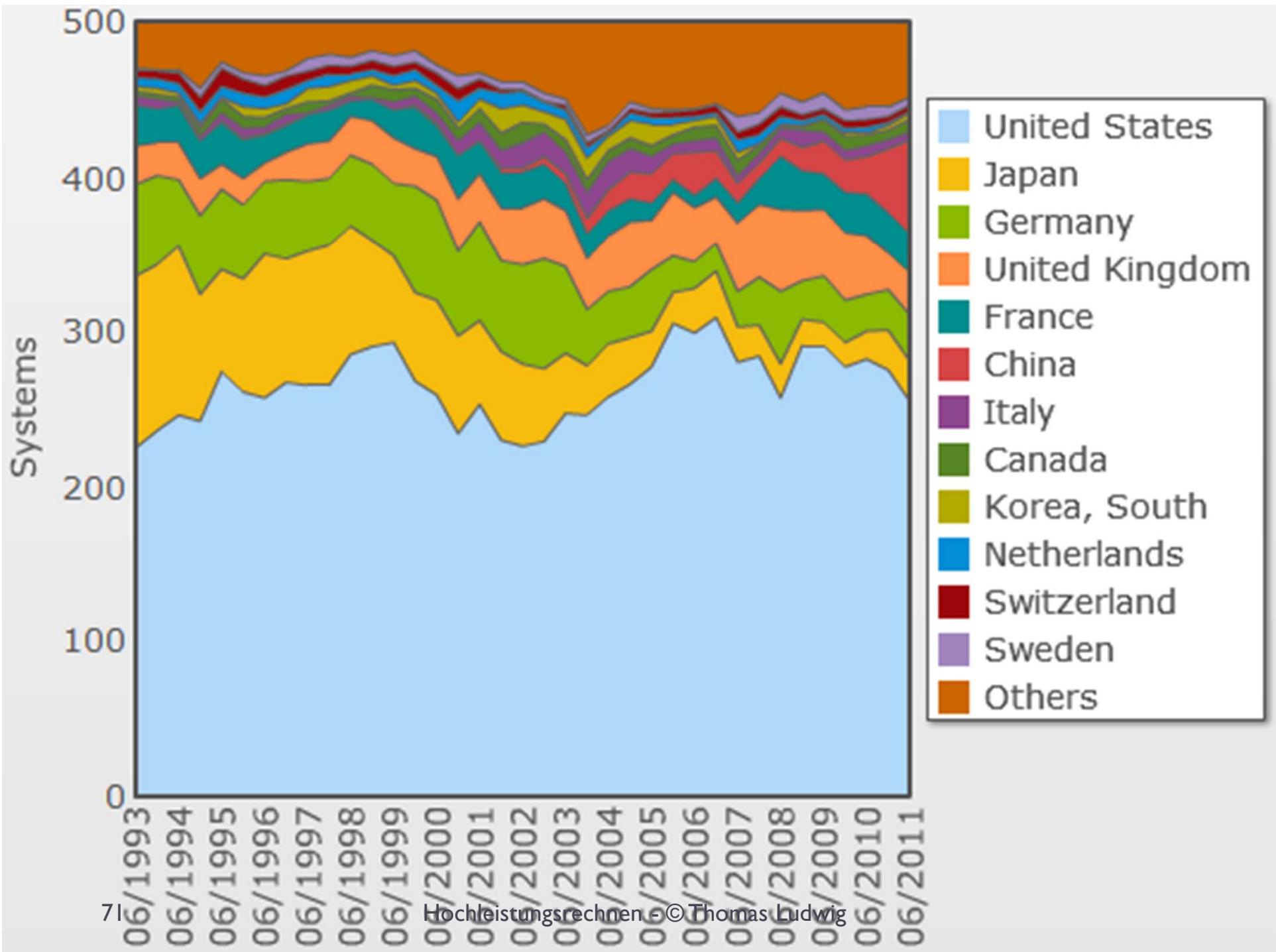


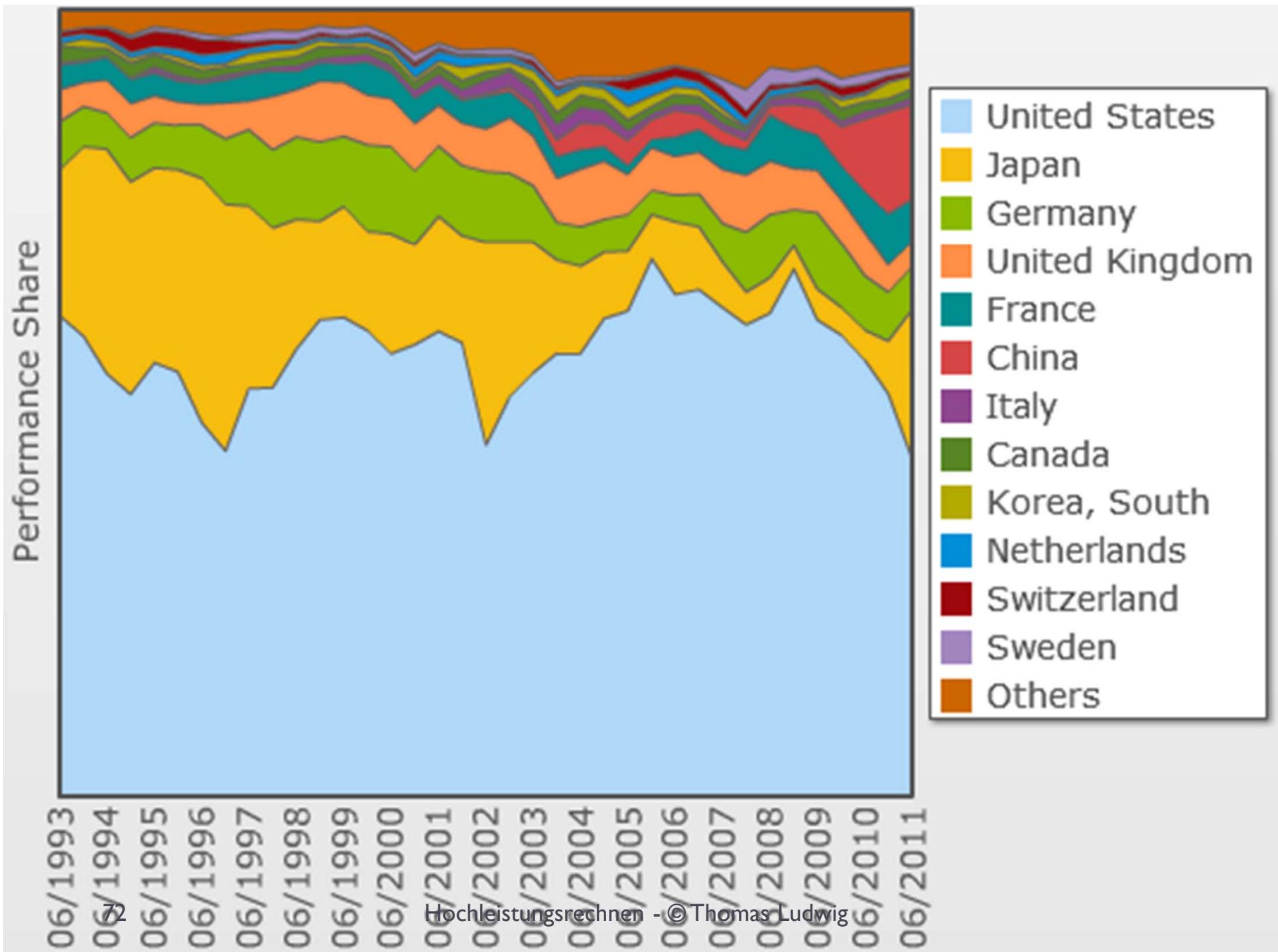


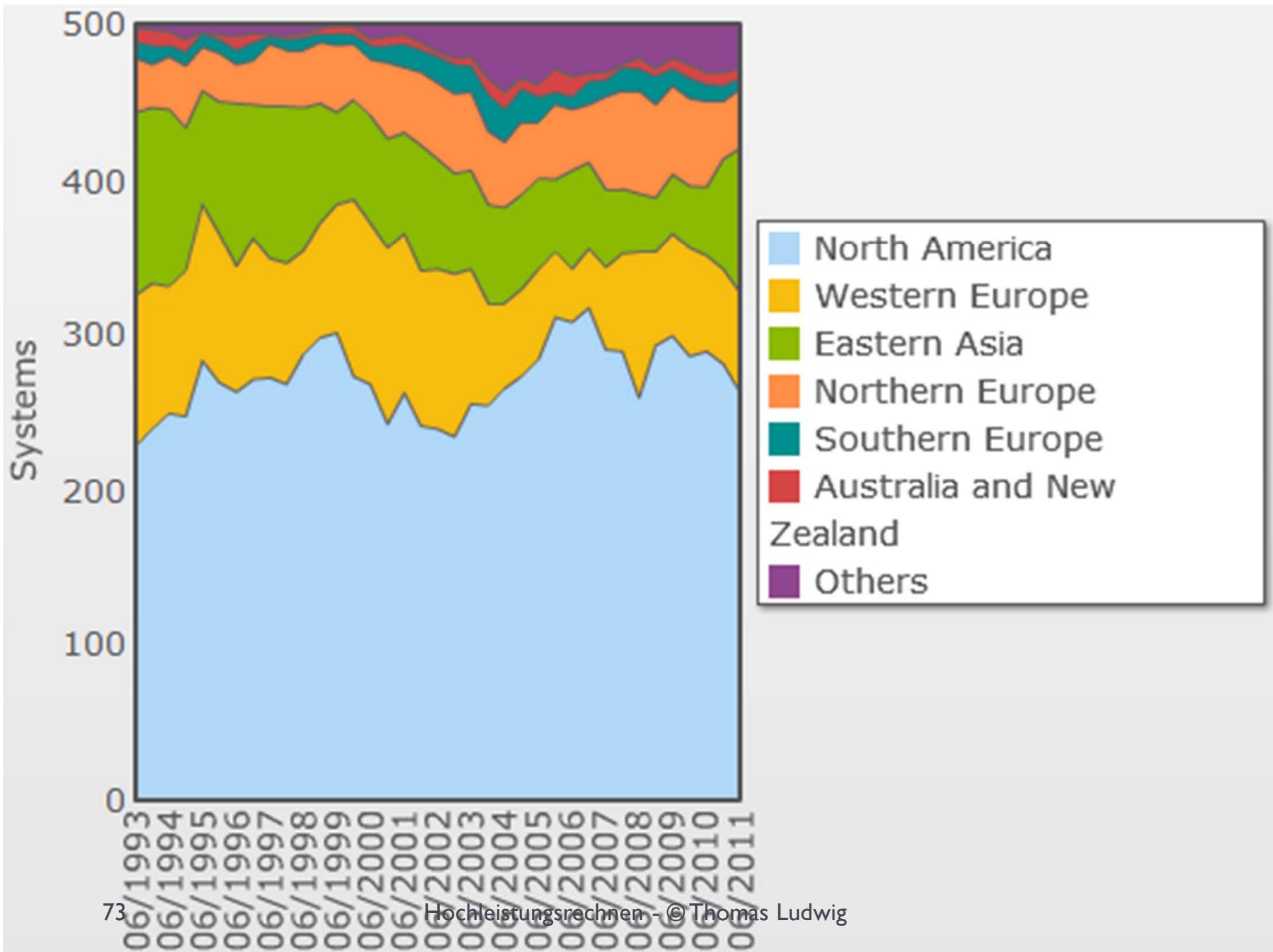


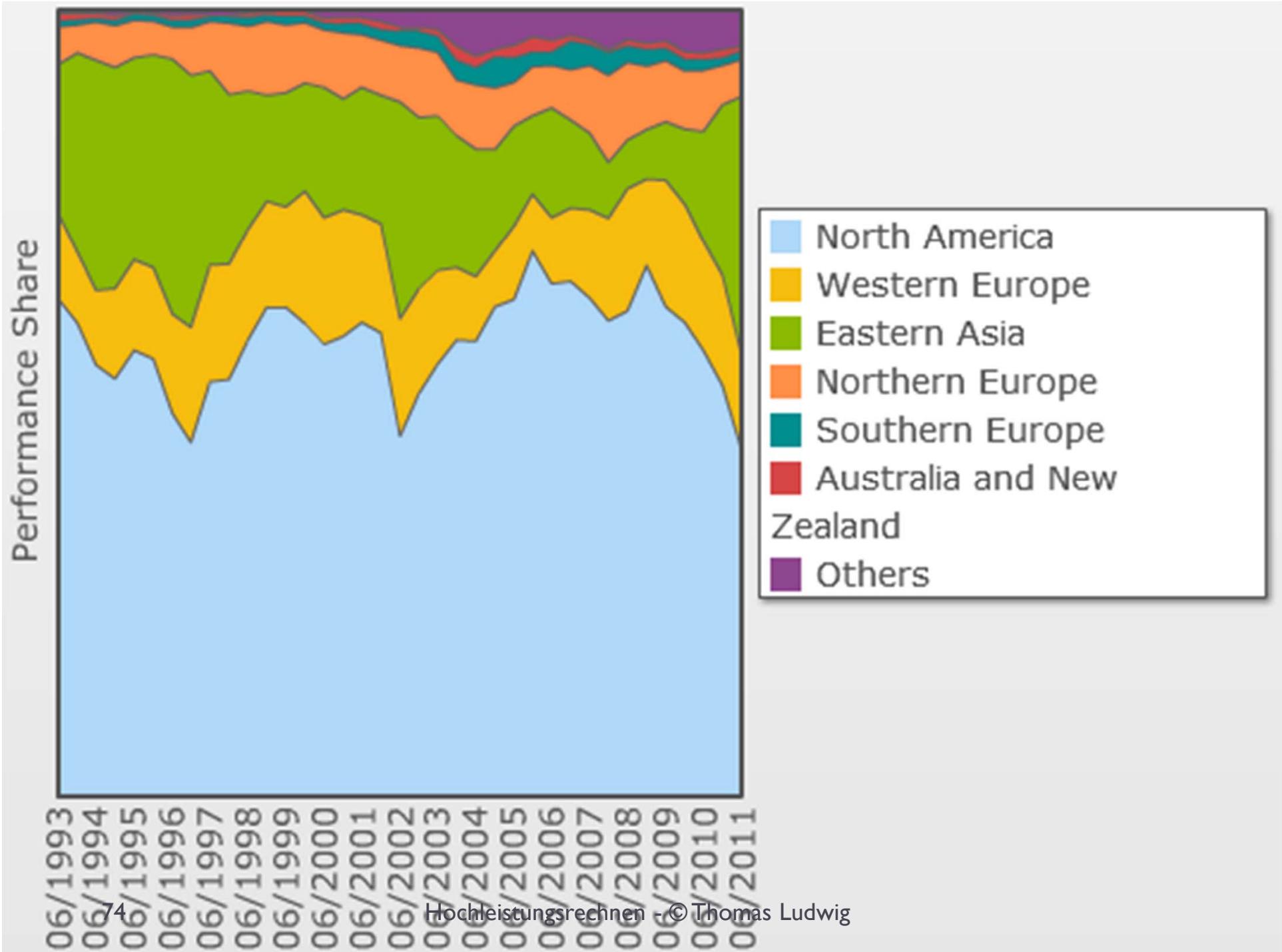


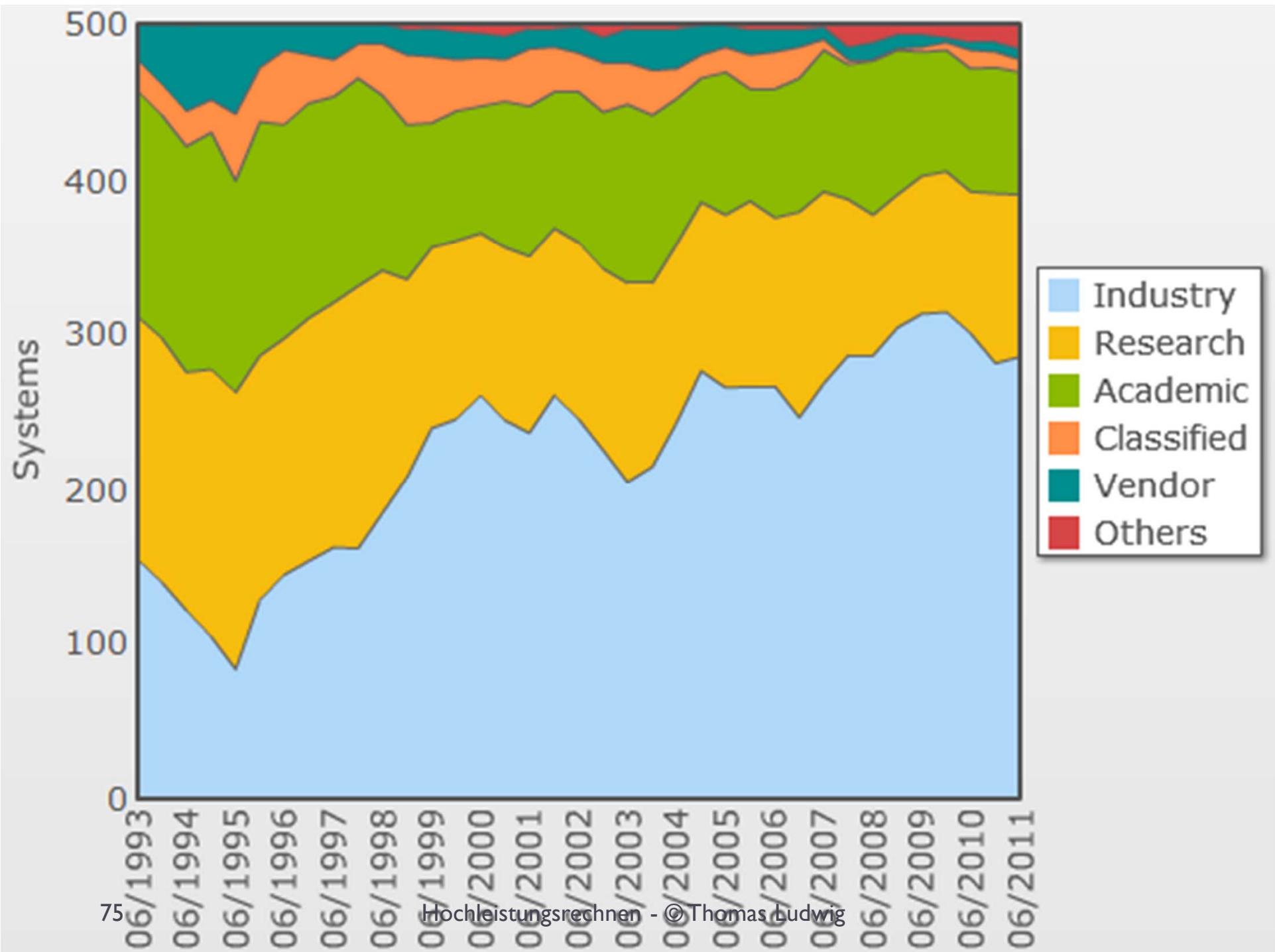


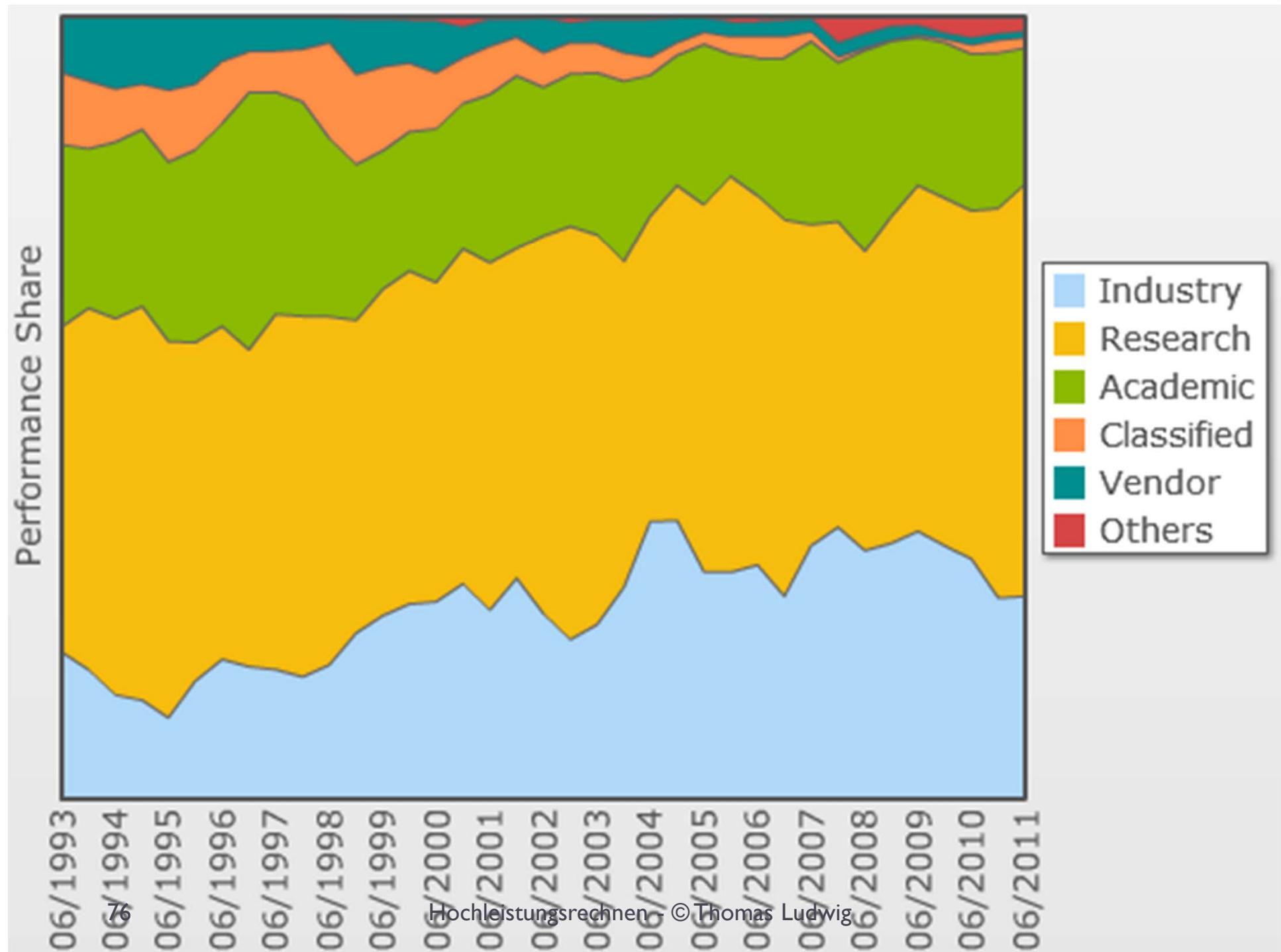


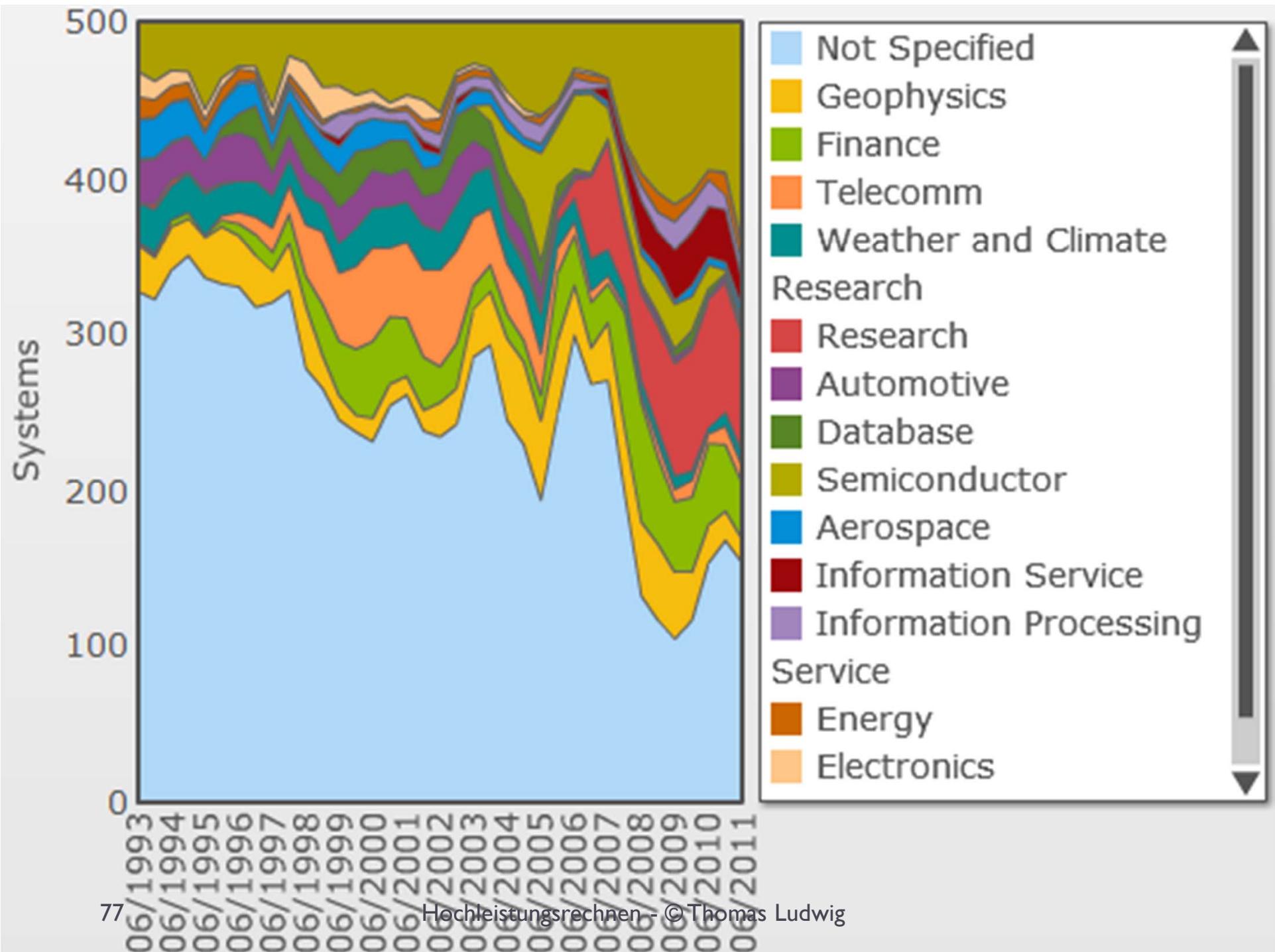


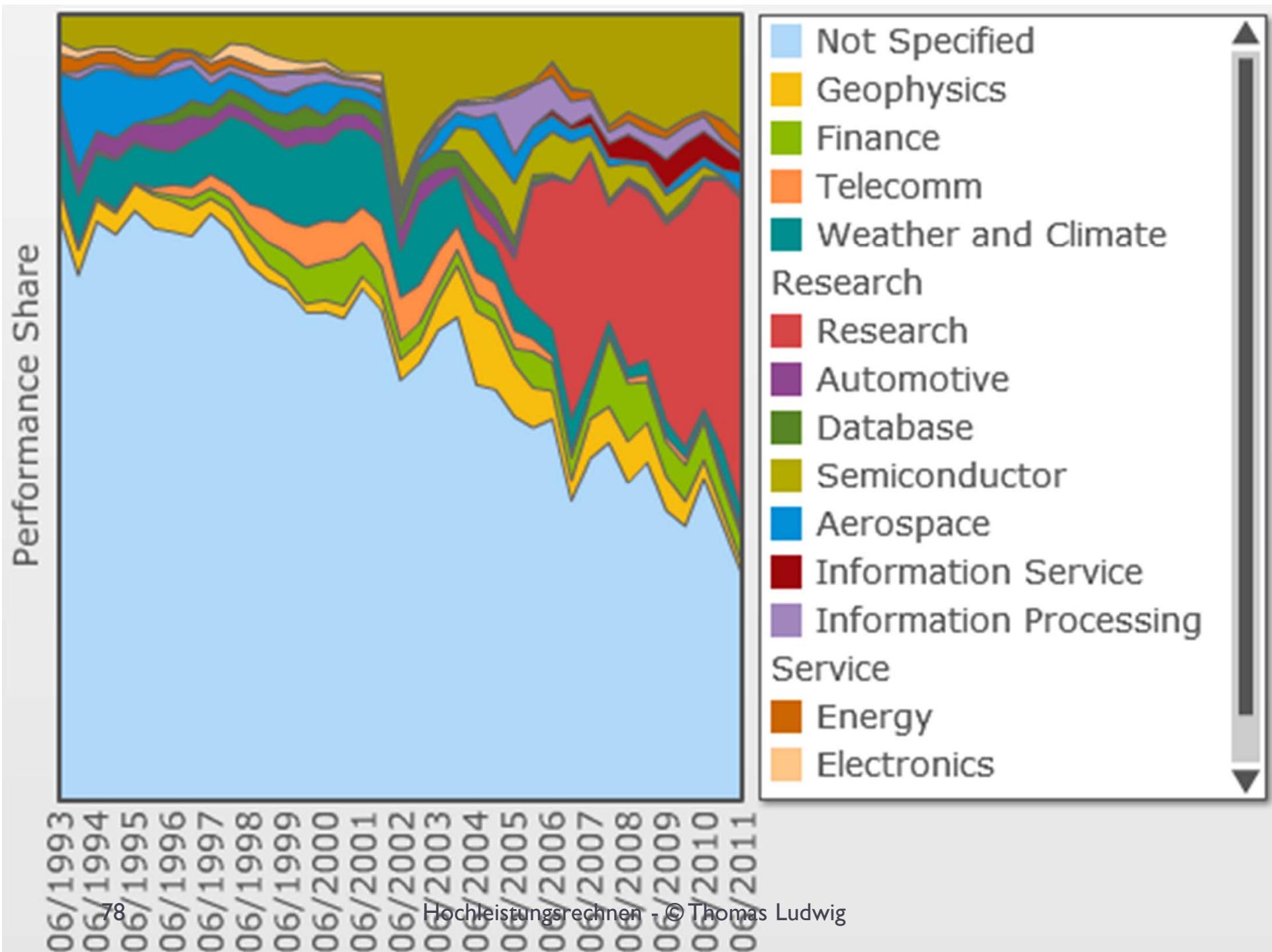


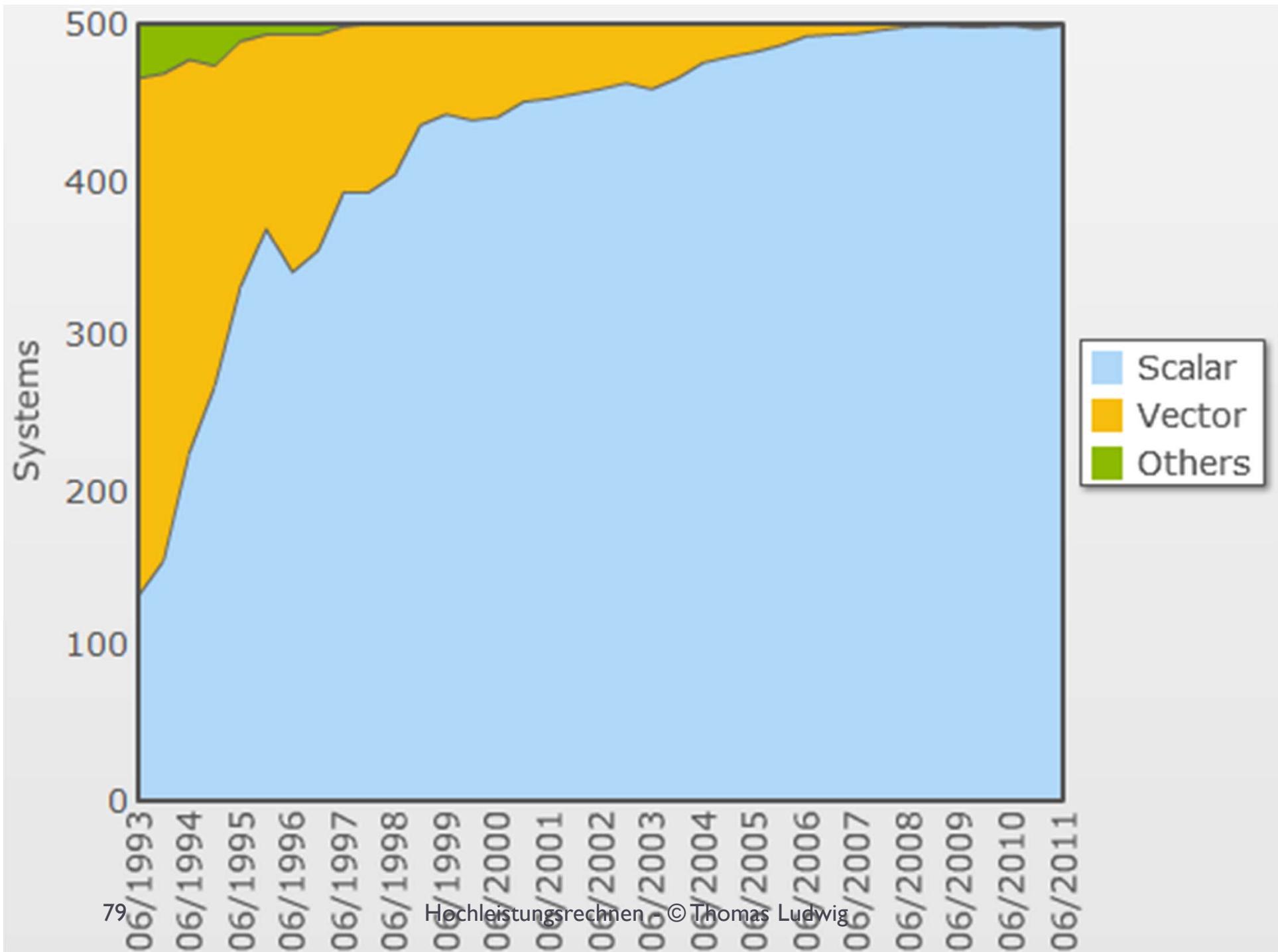


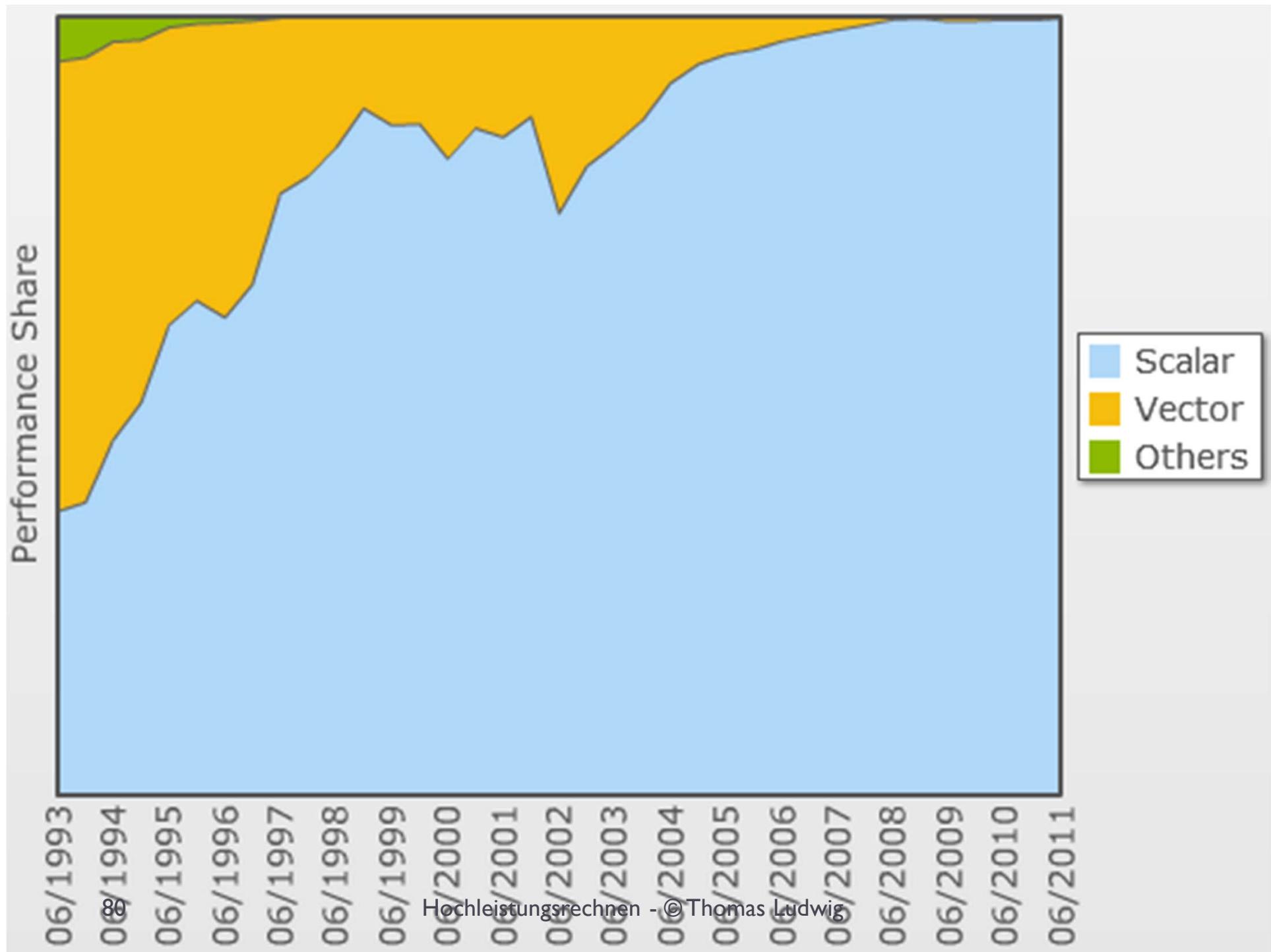


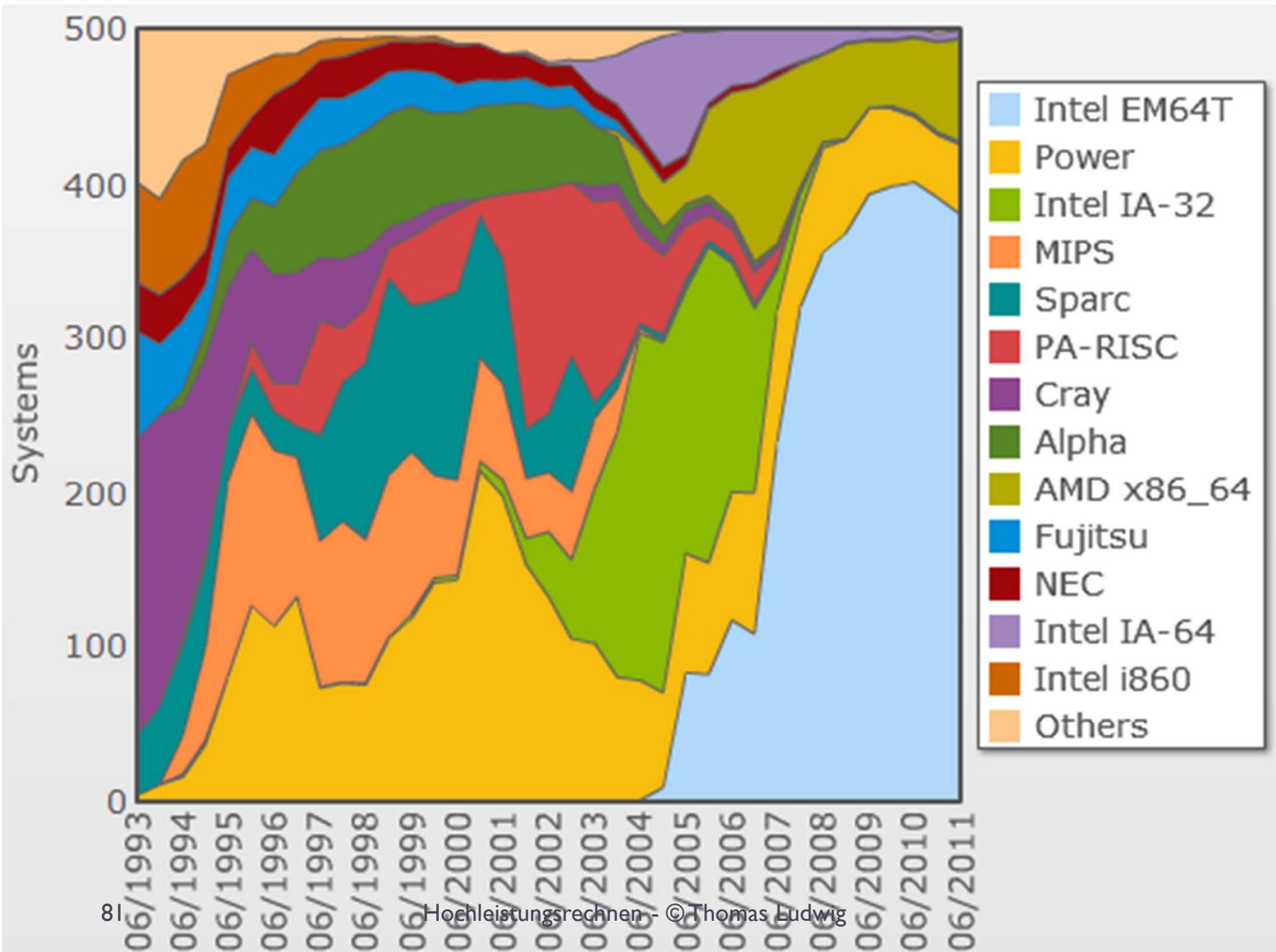


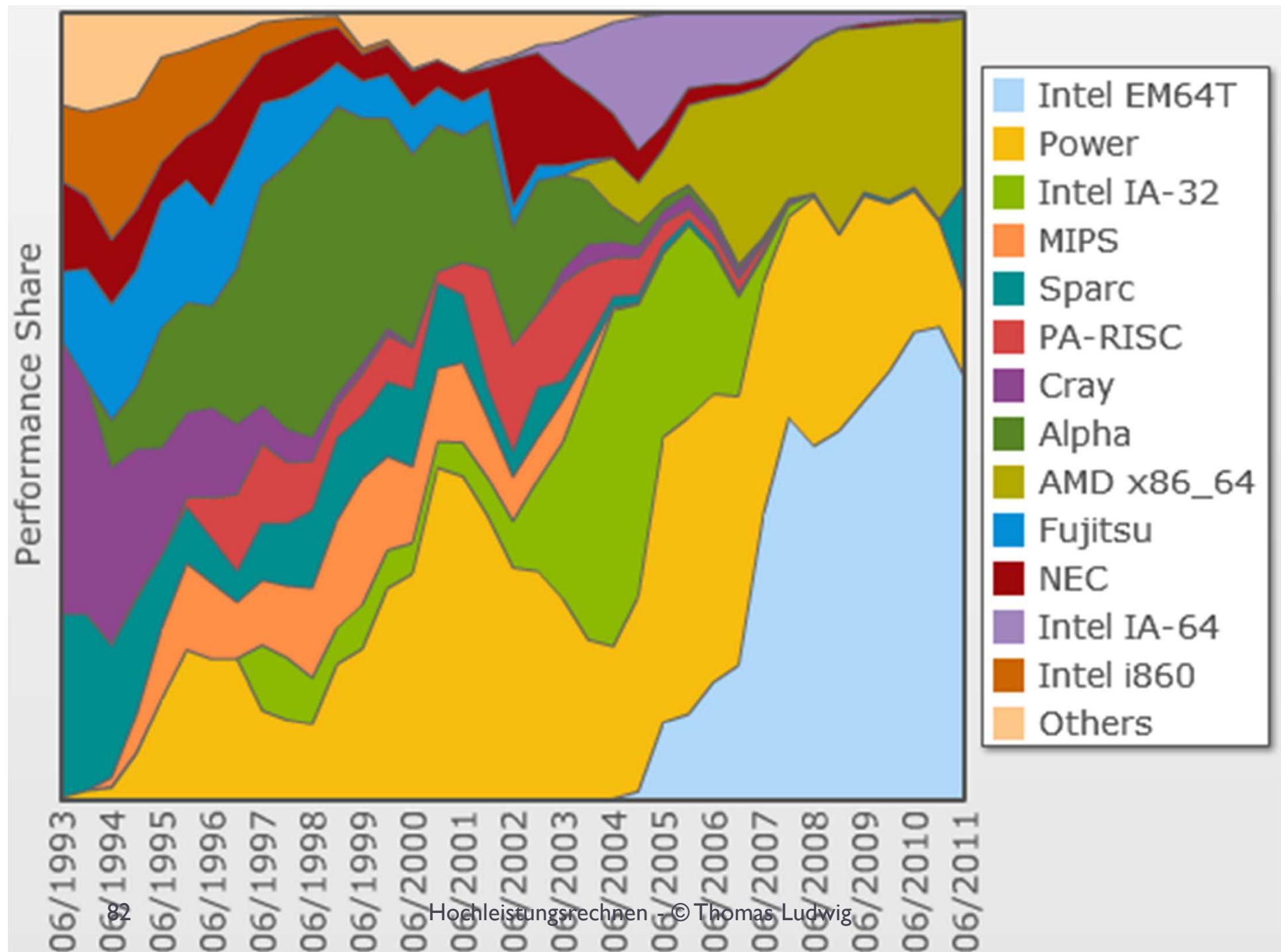


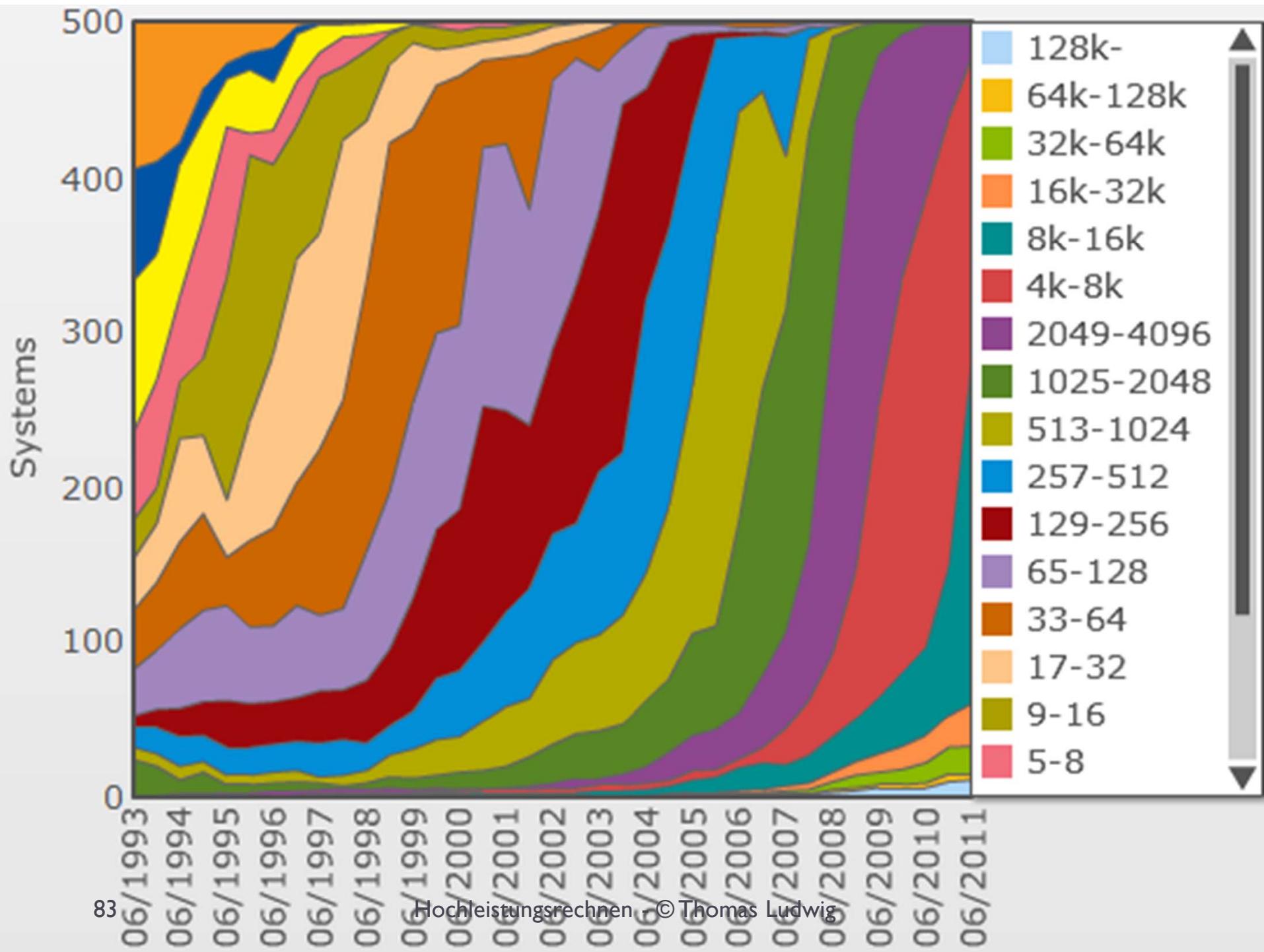


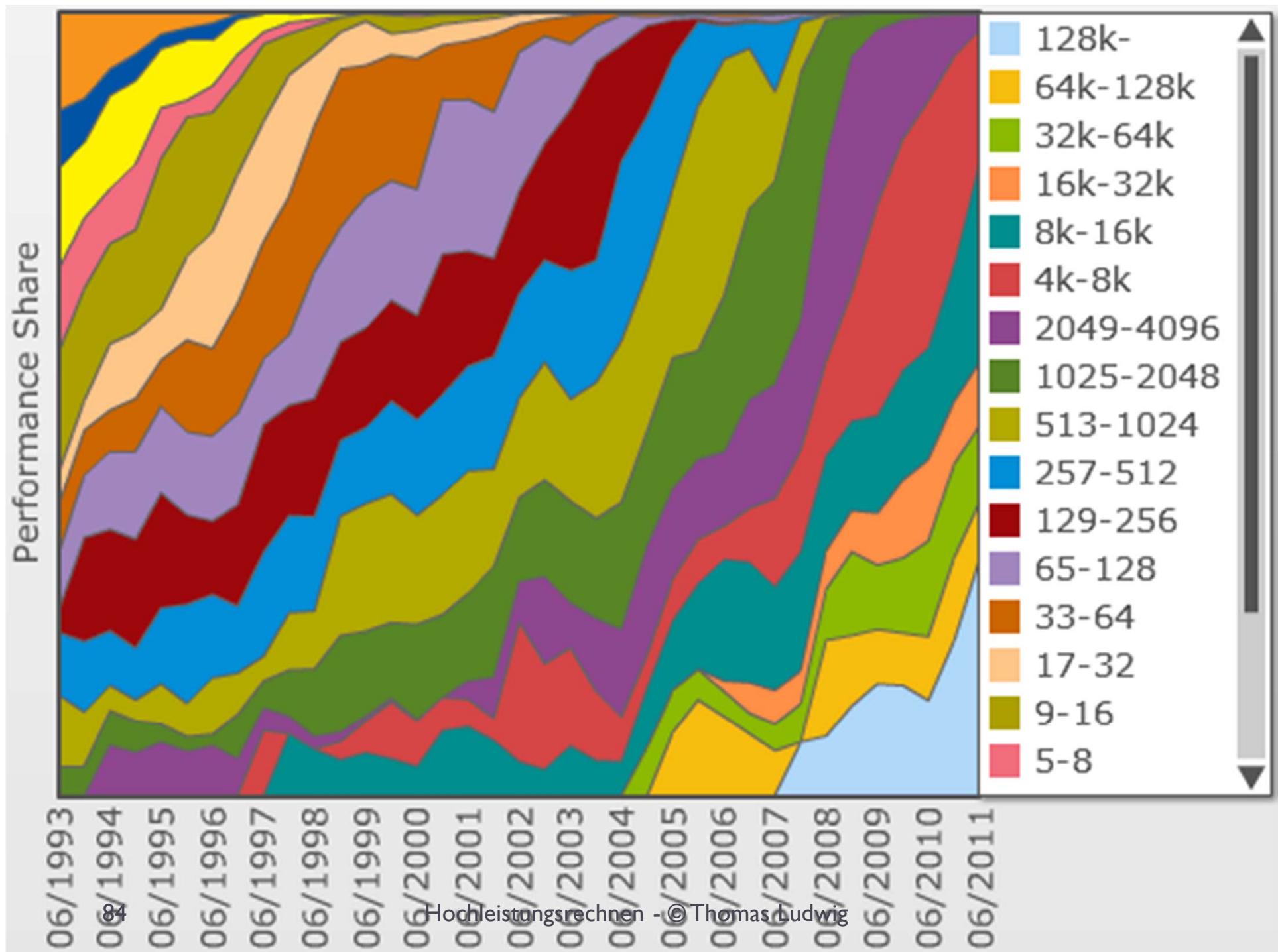


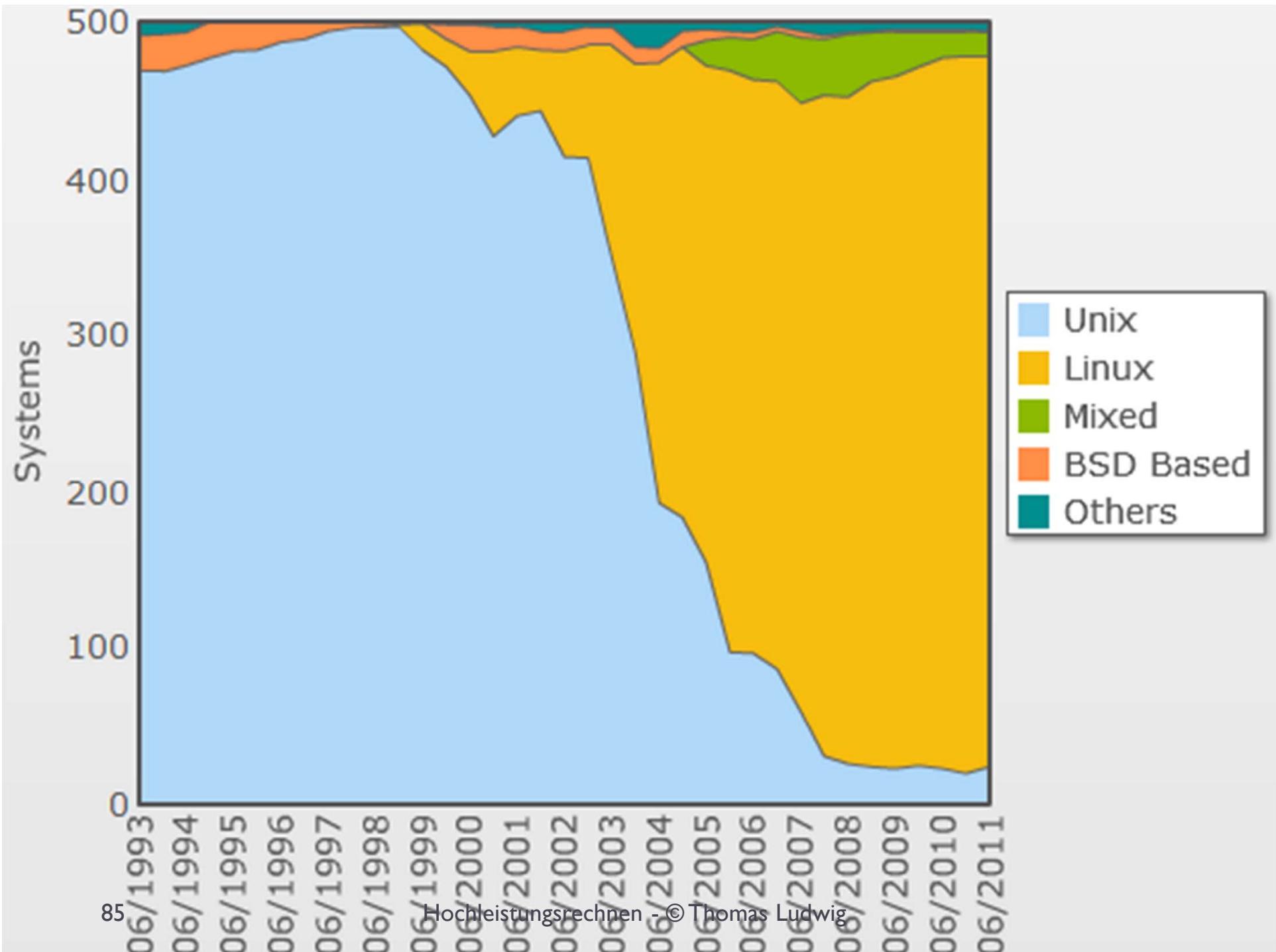


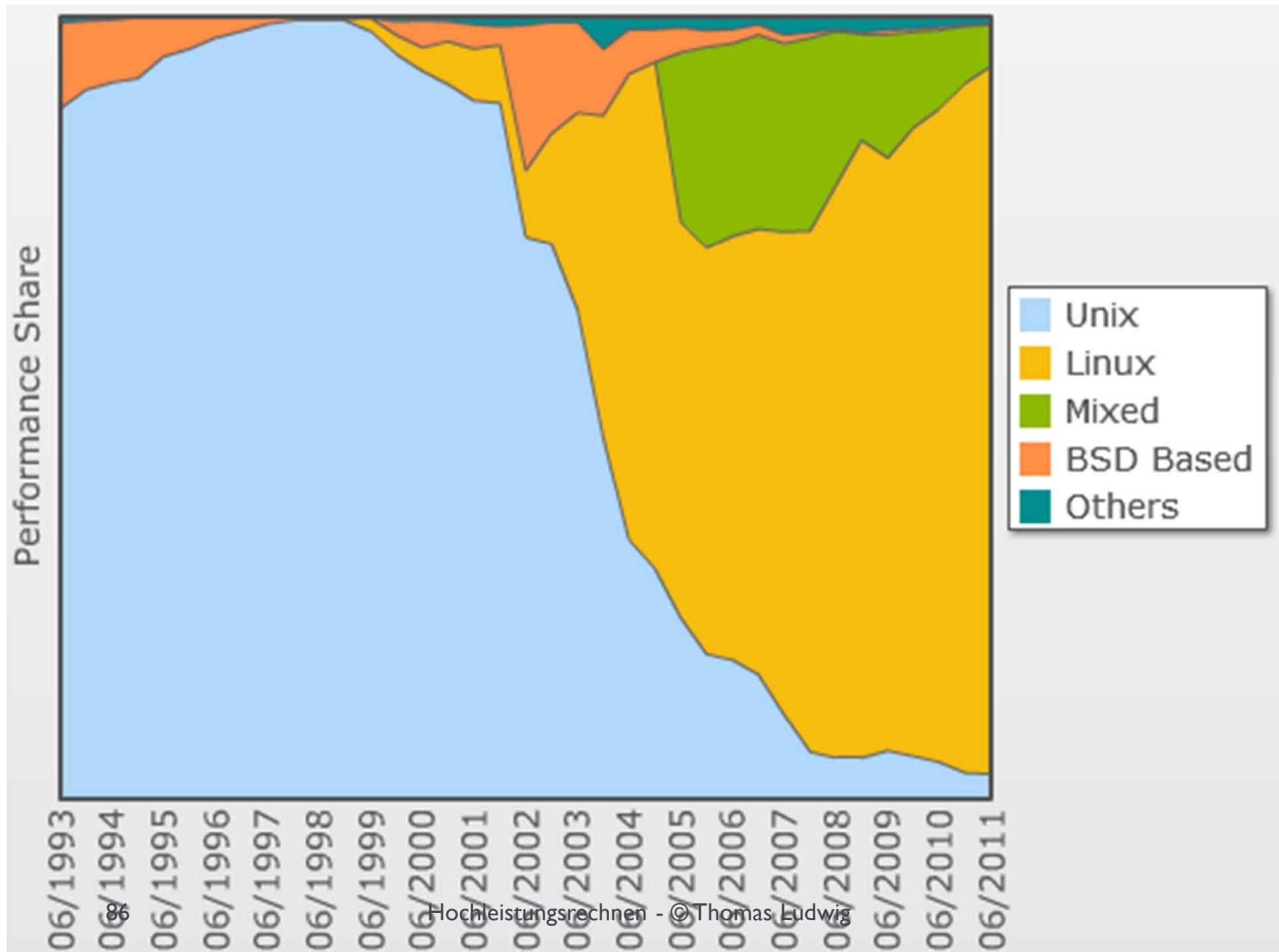












# Verbindungstechnologien

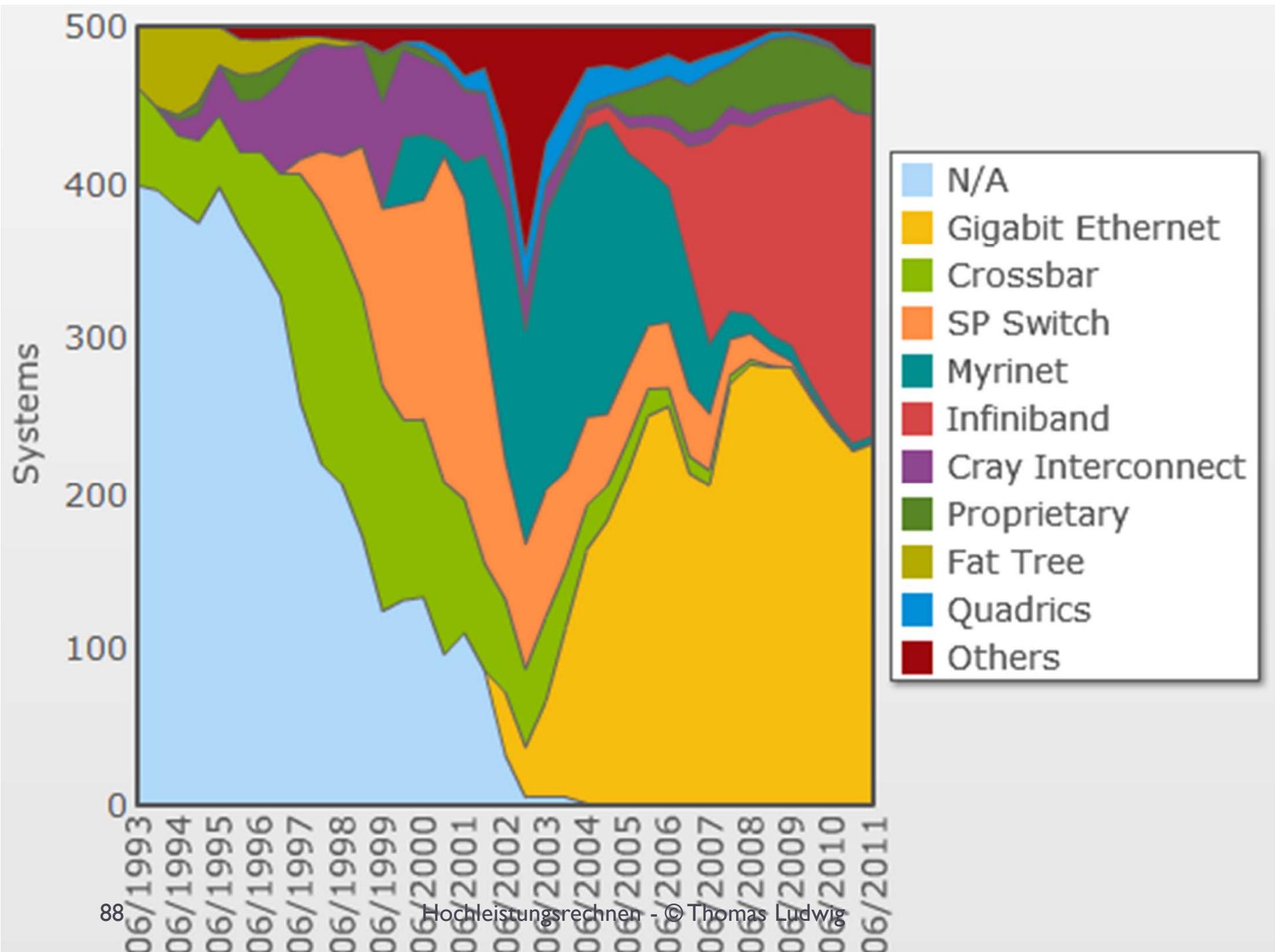
---

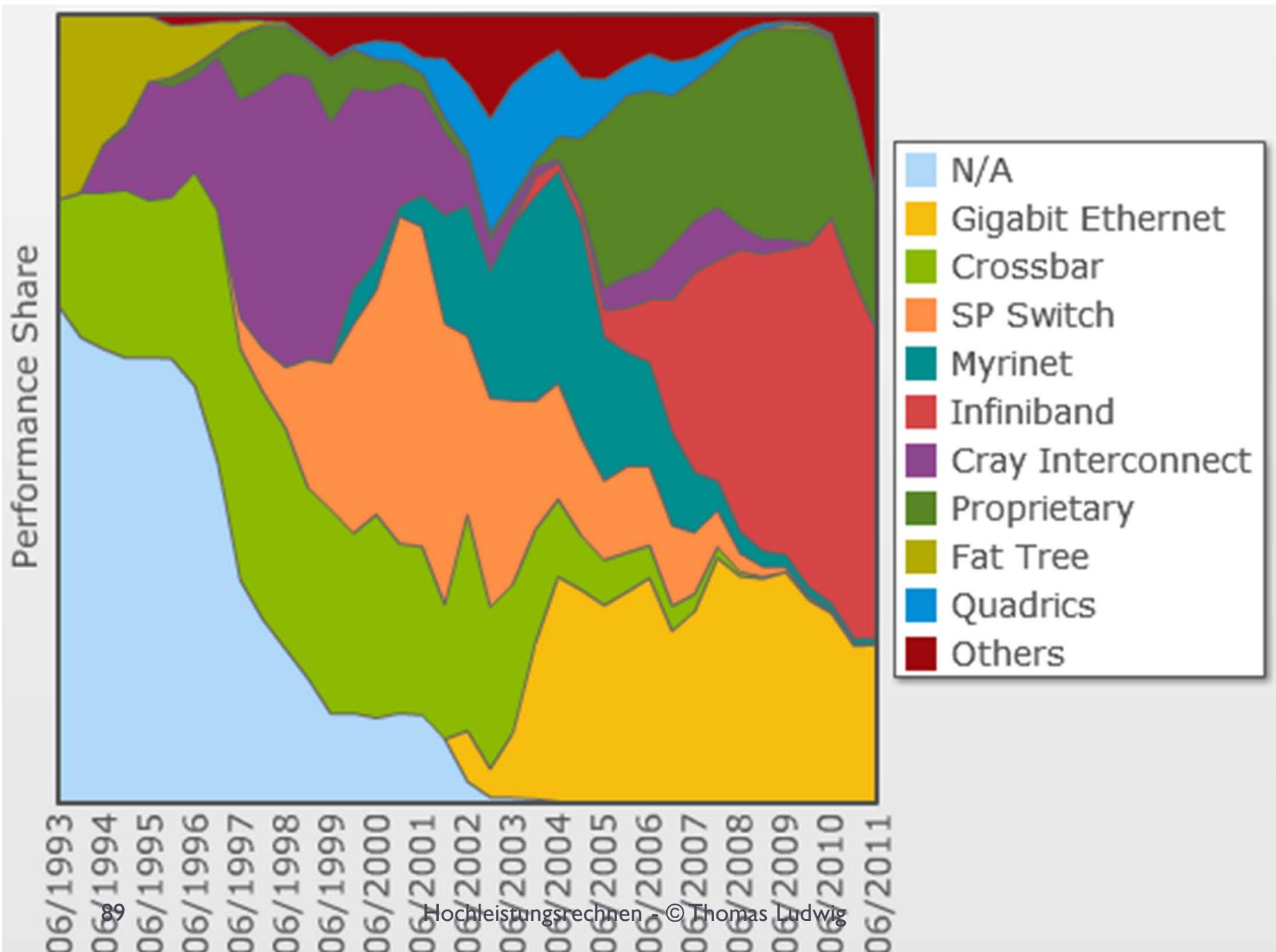
## In dedizierten Parallelrechnern

- ▶ Spezielle Hochleistungsnetze mit verschiedenen Topologien

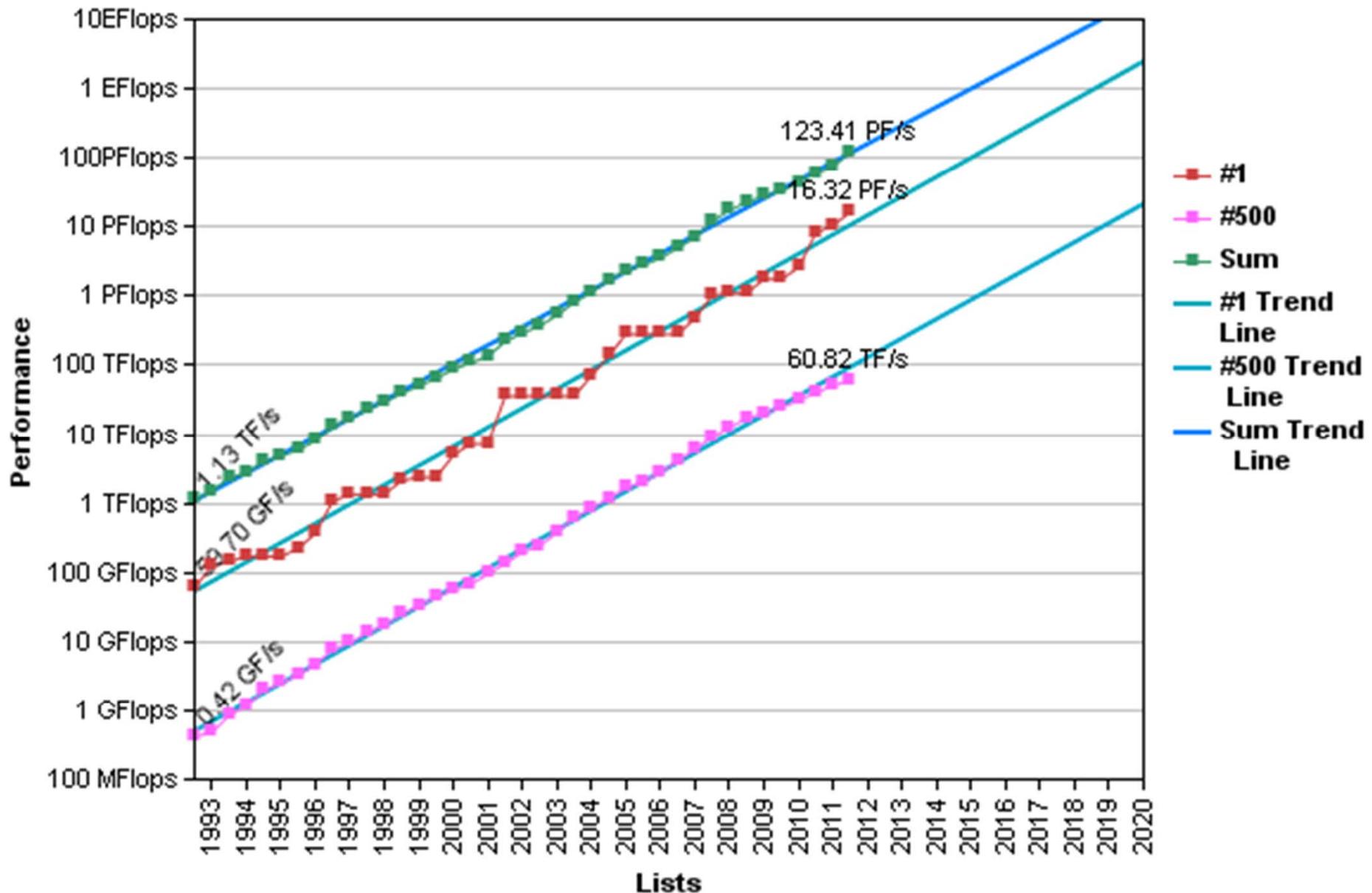
## In Cluster-Architekturen

- ▶ Preiswert: Gigabit-Ethernet
- ▶ Teuer: Myrinet
- ▶ Sehr teuer: InfiniBand, Quadrics





## Projected Performance Development



# Leistungsentwicklung bis Juni 2012

---

Moore´s Law

„Verdopplung der Transistorzahl alle 18 Monate“  
(entspricht evtl. Leistungsverdopplung)

Jun93–Jun12: 19 Jahre = 12x19 Monate  
etwa Faktor  $2^{12}=4096$

Leistung Summe: 1 TFlop/s – 123.000 TFlop/s (x 123k)

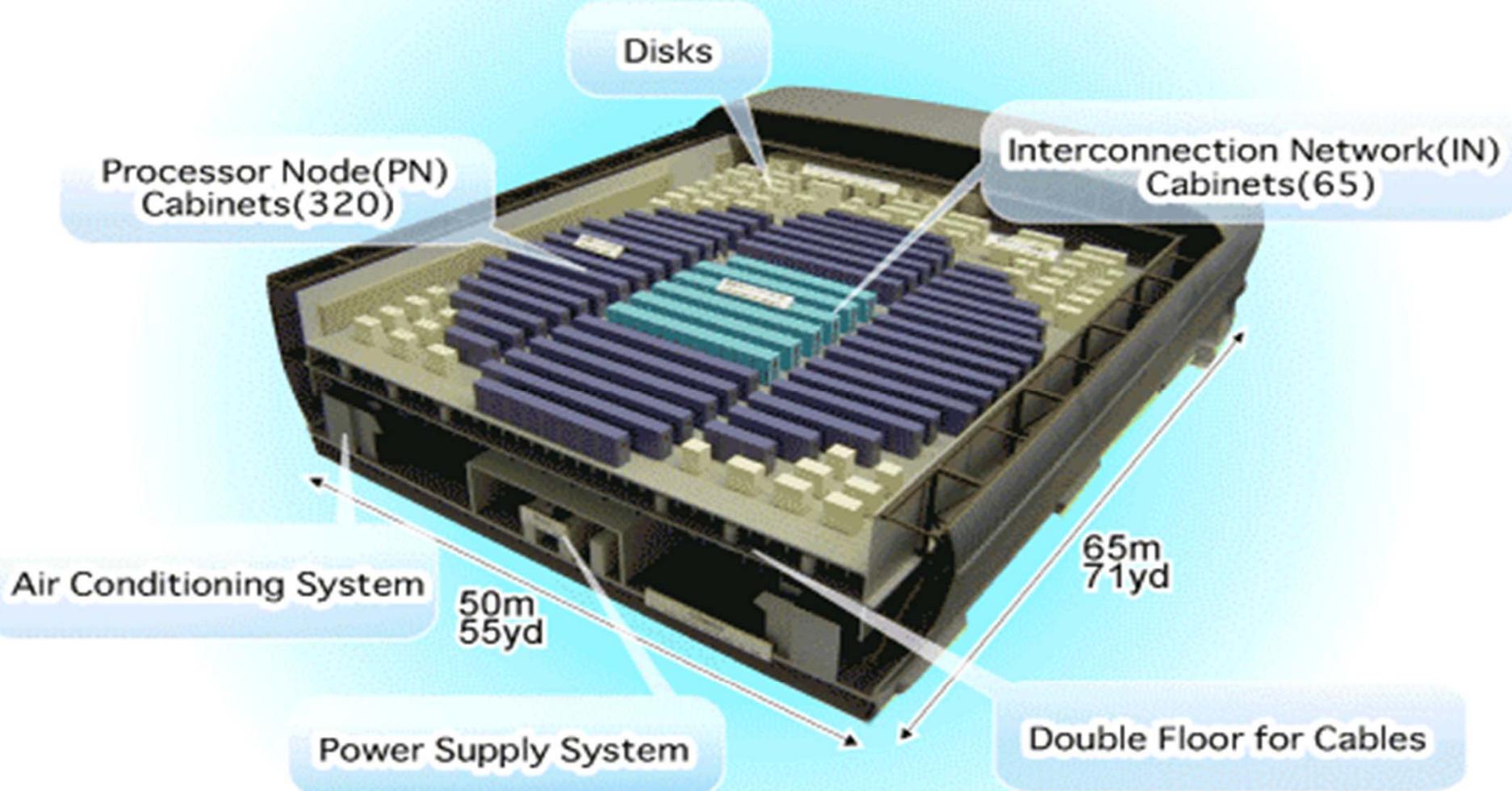
Leistung #1: 60 GFlop/s – 18.000 TFlop/s (x 300k)

Leistung #500: 0,4 GFlop/s – 60.000 GFlop/s (x 150k)

# NECs Earth Simulator (6/2002-11/2008)

- ▶ 640 Knoten
- ▶ Zu je 8 Vektorprozess.
- ▶ 5120 Prozessoren
- ▶ 0,15 mikron Kupfer
  
- ▶ 200 MioUSD Rechner
- ▶ 200 MioUSD Gebäude und Kraftwerk
- ▶ Zum Zwecke der Klimaforschung etc.
  
- ▶ 36 TFLOPS
- ▶ 10 TByte Hauptspeicher
- ▶ 700 TByte Festplatten
- ▶ 1,6 PByte Bandspeicher
- ▶ 83.000 Kupferkabel
- ▶ 2.800 km/220 t Kabel
- ▶ 3250qm
- ▶ Erdbebensicher

Computenic-Shock



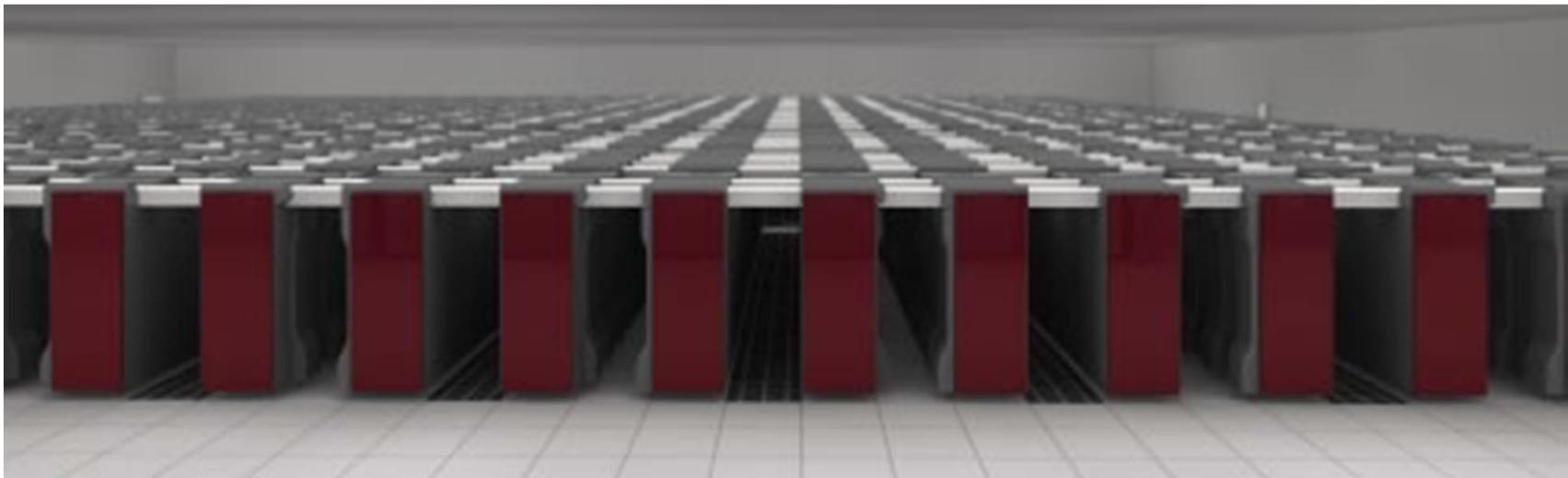
# IBMs BlueGene-Programm

---

- ▶ Ursprünglich Hauptanwendung: Proteinfaltung
- ▶ Spezialprozessoren: leistungsschwach aber stromsparend – wenig Hauptspeicher pro Core
- ▶ Betriebssystem: mehrere Varianten von Linux
- ▶ Verbindungsnetz: proprietärer 3dim-Torus mit Zusatznetzen für globale Kommunikation, E/A und Verwaltung
- ▶ Systeme:
  - ▶ Blue Gene/L: 180TFLOPS, Ende 2004
    - ▶ Bis Ende 2005 verdoppelt auf 370TFLOPS
  - ▶ Blue Gene/P: 1PFLOPS 2006/2007
  - ▶ Blue Gene/Q: 3PFLOPS, 2007/2008

# Fujitsu K Computer

- ▶ RIKEN Advanced Institute for Computational Science (AICS), Japan



- ▶ 68.544 SPARC64 VIIIfx CPUs (2,0 GHz) mit je 8 Prozessorkernen in 672 Schränken
- ▶ November 2012: 864 Schränke und 10 PFLOPS



# Historische Sicht

---

- ▶ Die TOP500 im Juni 1993
- ▶ Deutschland in der TOP500 im Juni 1993

| Rank | Manufacturer<br>Computer/Procs      | R <sub>max</sub><br>R <sub>peak</sub> | GFLOPS   | Installation Site<br>Country/Year | Inst. type<br>Installation Area | Nmax<br>Nhalf            | Computer Family<br>Computer Type |
|------|-------------------------------------|---------------------------------------|--|-----------------------------------|---------------------------------|--------------------------|----------------------------------|
| 1    | TMC<br>CM-5/1024/ 1024              | 59.70<br>131.00                       | Los Alamos National Laboratory<br>USA/               | Research<br>Energy                | 52224<br>24064                  | TMC CM5<br>CM5           |                                  |
| 2    | TMC<br>CM-5/1024/ 1024              | 59.70<br>131.00                       | National Security Agency<br>USA/                     | Classified                        | 52224<br>24064                  | TMC CM5<br>CM5           |                                  |
| 3    | TMC<br>CM-5/544/ 544                | 30.40<br>70.00                        | Minnesota Supercomputer Center<br>USA/               | Industry                          | 36864<br>16384                  | TMC CM5<br>CM5           |                                  |
| 4    | TMC<br>CM-5/512/ 512                | 30.40<br>66.00                        | NCSA<br>USA/   | Academic                          | 36864<br>16384                  | TMC CM5<br>CM5           |                                  |
| 5    | NEC<br>SX-3/44R/ 4                  | 23.20<br>26.00                        | NEC Fuchu Plant<br>Japan/1990                        | Vendor                            | 6400<br>830                     | NEC Vector<br>SX3        |                                  |
| 6    | NEC<br>SX-3/44/ 4                   | 20.00<br>22.00                        | Atmospheric Environment Service (AES)<br>Canada/1991 | Research<br>Weather               | 6144<br>832                     | NEC Vector<br>SX3        |                                  |
| 7    | TMC<br>CM-5/256/ 256                | 15.10<br>33.00                        | Naval Research Laboratory (NRL)<br>USA/1992          | Research                          | 26112<br>12032                  | TMC CM5<br>CM5           |                                  |
| 8    | Intel<br>Delta/ 512                 | 13.90<br>20.48                        | Caltech<br>USA/                                      | Academic                          | 25000<br>7500                   | intel Paragon<br>Paragon |                                  |
| 9    | Cray/SGI<br>Y-MP C916/16256/ 16     | 13.70<br>15.24                        | Cray Research<br>USA/                                | Vendor                            | 10000<br>650                    | Cray Vector<br>C90       |                                  |
| 10   | Cray/SGI<br>Y-MP C916/16256/ 16     | 13.70<br>15.24                        | DOE/Bettis Atomic Power Laboratory<br>USA/1993       | Research                          | 10000<br>650                    | Cray Vector<br>C90       |                                  |
| 11   | Cray/SGI<br>Y-MP C916/16256/ 16     | 13.70<br>15.24                        | DOE/Knolls Atomic Power Laboratory<br>USA/1993       | Research                          | 10000<br>650                    | Cray Vector<br>C90       |                                  |
| 12   | Cray/SGI<br>Y-MP C916/16128/ 16     | 13.70<br>15.24                        | ECMWF<br>UK/1993                                     | Research<br>Weather               | 10000<br>650                    | Cray Vector<br>C90       |                                  |
| 13   | Cray/SGI<br>Y-MP C916/161024/<br>16 | 13.70<br>15.24                        | Government<br>USA/1992                               | Classified                        | 10000<br>650                    | Cray Vector<br>C90       |                                  |
| 14   | Cray/SGI<br>Y-MP C916/161024/<br>16 | 13.70<br>15.24                        | Government<br>USA/1992                               | Classified                        | 10000<br>650                    | Cray Vector<br>C90       |                                  |

| Rank | Manufacturer<br>Computer/Procs | R <sub>max</sub><br>R <sub>peak</sub> | GFLOPS                          | Installation Site<br>Country/Year | Inst. type<br>Installation Area | Nmax<br>Nhalf | Computer Family<br>Computer Type |
|------|--------------------------------|---------------------------------------|---------------------------------|-----------------------------------|---------------------------------|---------------|----------------------------------|
| 56   | Fujitsu<br>S600/20/ 1          | 4.01<br>5.00                          | Universitaet Aachen             | Germany/1991                      | Academic                        |               | Fujitsu VP<br>VP2000             |
| 57   | Fujitsu<br>S600/20/ 1          | 4.01<br>5.00                          | Universitaet Karlsruhe          | Germany/1990                      | Academic                        |               | Fujitsu VP<br>VP2000             |
| 60   | TMC<br>CM-5/64/ 64             | 3.80<br>8.19                          | GMD                             | Germany/1993                      | Research                        | 13056<br>6016 | TMC CM5<br>CM5                   |
| 65   | Fujitsu<br>S400/40/ 2          | 3.62<br>5.00                          | Universitaet Darmstadt          | Germany/1991                      | Academic                        |               | Fujitsu VP<br>VP2000             |
| 66   | Fujitsu<br>S400/40/ 2          | 3.62<br>5.00                          | Universitaet Hannover / RRZN    | Germany/1991                      | Academic                        |               | Fujitsu VP<br>VP2000             |
| 76   | TMC<br>CM-2/32k/ 1024          | 2.60<br>7.00                          | AMK                             | Germany/1990                      | Classified                      |               | TMC CM2<br>CM2                   |
| 98   | Cray/SGI<br>Y-MP8/832/ 8       | 2.14<br>2.67                          | Forschungszentrum Juelich (FZJ) | Germany/1989                      | Research                        |               | Cray Vector<br>YMP               |
| 102  | Cray/SGI<br>Y-MP8/864/ 8       | 2.14<br>2.67                          | Leibniz Rechenzentrum           | Germany/1992                      | Academic                        |               | Cray Vector<br>YMP               |
| 142  | TMC<br>CM-5/32/ 32             | 1.90<br>4.10                          | Universitaet Wuppertal          | Germany/1992                      | Academic                        | 9216<br>4096  | TMC CM5<br>CM5                   |
| 149  | Intel<br>XP/S5/ 66             | 1.90<br>3.30                          | Forschungszentrum Juelich (FZJ) | Germany/1992                      | Research                        |               | intel Paragon<br>Paragon         |
| 155  | Intel<br>XP/S5-32/ 66          | 1.90<br>3.30                          | Universitaet Stuttgart          | Germany/1992                      | Academic                        |               | intel Paragon<br>Paragon         |
| 190  | Cray/SGI<br>CRAY-2s/4-128/ 4   | 1.41<br>1.95                          | DKRZ                            | Germany/1988                      | Research<br>Weather             |               | Cray2/3<br>Cray 2                |
| 206  | Cray/SGI<br>CRAY-2/4-256/ 4    | 1.41<br>1.95                          | Universitaet Stuttgart          | Germany/1986                      | Academic                        |               | Cray2/3<br>Cray 2                |
| 218  | TMC<br>CM-2/16k/ 512           | 1.30<br>3.50                          | GMD                             | Germany/1990                      | Research                        |               | TMC CM2<br>CM2                   |
| 223  | NEC<br>SX-3/11/ 1              | 1.30<br>1.37                          | Universitaet Koeln              | Germany/1990                      | Academic                        | 2816<br>192   | NEC Vector<br>SX3                |

# Die TOP500-Liste

## Zusammenfassung

- ▶ Die Rechnerleistung wird mit einem numerischen Benchmark-Programm (LINPACK) evaluiert
- ▶ Die TOP500-Liste verzeichnet halbjährig die schnellsten Rechner weltweit
- ▶ Die schnellsten Rechner haben die Petaflops-Grenze durchbrochen
- ▶ IBM ist der bedeutendste Leistungslieferant
- ▶ Die USA dominieren den Markt
- ▶ Wichtigste Architektur: Cluster
- ▶ Wichtigste Vernetzung: Gigabit-Ethernet, Infiniband
- ▶ Fast nur noch CMOS off-the-shelf-Prozessoren
- ▶ Aktuelles Problem: Energiebedarf

# Vernetzungkonzepte

---

- ▶ Vernetzung von Rechnern und Eingabe/Ausgabe
- ▶ Allgemeine Betrachtungen
- ▶ Designaspekte effizienter Kommunikation
- ▶ Leistungsentwicklung
- ▶ Leistungsmaße
- ▶ Netztopologien
- ▶ Einbindung in TCP/IP
- ▶ InfiniBand-Vernetzung

# Vernetzungskonzepte

## Die zehn wichtigsten Fragen

- ▶ Welche Aufgaben hat die Vernetzung?
- ▶ Welche Komponenten weist eine Vernetzung auf?
- ▶ Welche Fragen finden wir bei der Pufferverwaltung?
- ▶ Was bedeutet Überlagerung von Berechnung und Kommunikation?
- ▶ Was versteht man unter Hardware-Realisierung von Software?
- ▶ Welche Bandbreiten finden wir bei aktuellen Netztechnologien?
- ▶ Welche Charakteristiken sollte die Netzhardware aufweisen?
- ▶ Was versteht man unter Bisektionsbandbreite?
- ▶ Wie umgeht man die Engstelle TCP/IP
- ▶ Welche Protokolle finden wir bei InfiniBand?

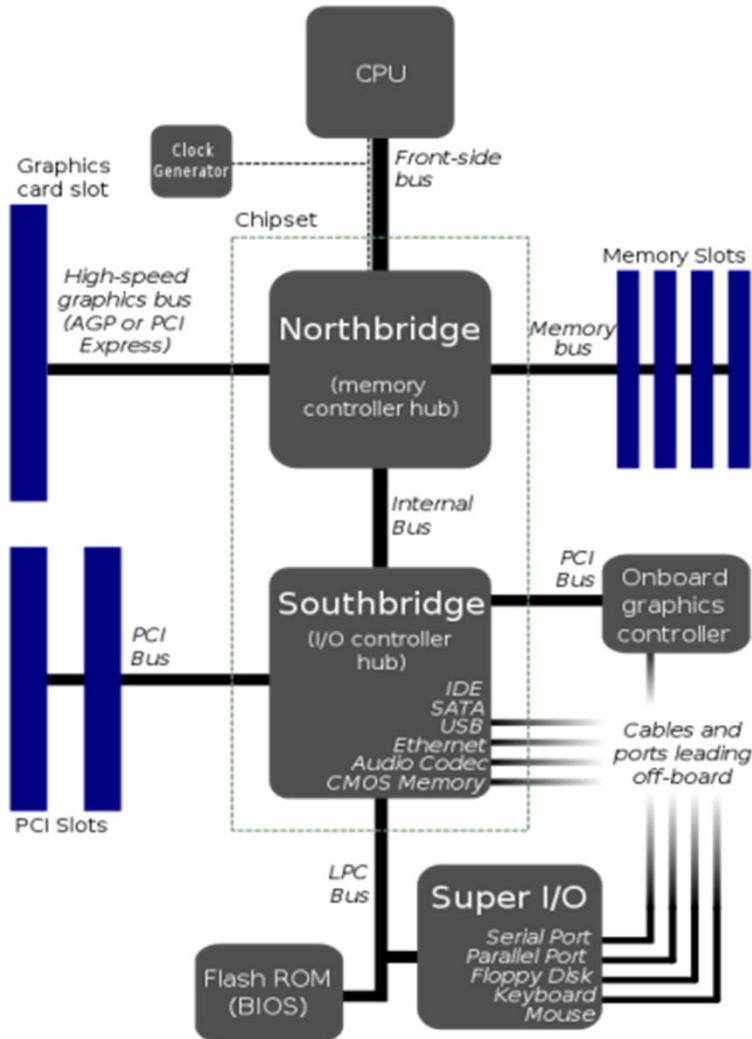
# Aufgabenstellung

---

## Wozu Vernetzung?

- ▶ Die Vernetzung verbindet Rechnerknoten miteinander, E/A-Knoten miteinander und Rechner mit E/A-Knoten
  
- ▶ Ermöglicht die Interprozeßkommunikation
  - ▶ Prozesse auf verschiedenen Knoten
- ▶ Ermöglicht Prozeß-Eingabe/Ausgabe
  - ▶ E/A-Hardware meist nicht knotenlokal angeschlossen

# Rechnerinterne Vernetzung



- ▶ **Front-Side-Bus**
  - ▶ Z.B. 64 Bit mit 100 MHz und 4 Übertragungen/Takt ergibt 3.200 MByte/s
- ▶ **Northbridge** jetzt oft im Prozessor
- ▶ Bussystem an der **Southbridge** schafft Verbindung zu Peripherie
- ▶ **Prozeß-zu-Prozeß**
  - ▶ Gemeinsamer Speicher
  - ▶ Socketkommunikation
- ▶ **Prozeß-zu-Datei**
  - ▶ Datei-E/A

# Vernetzung von Rechnern und E/A

---

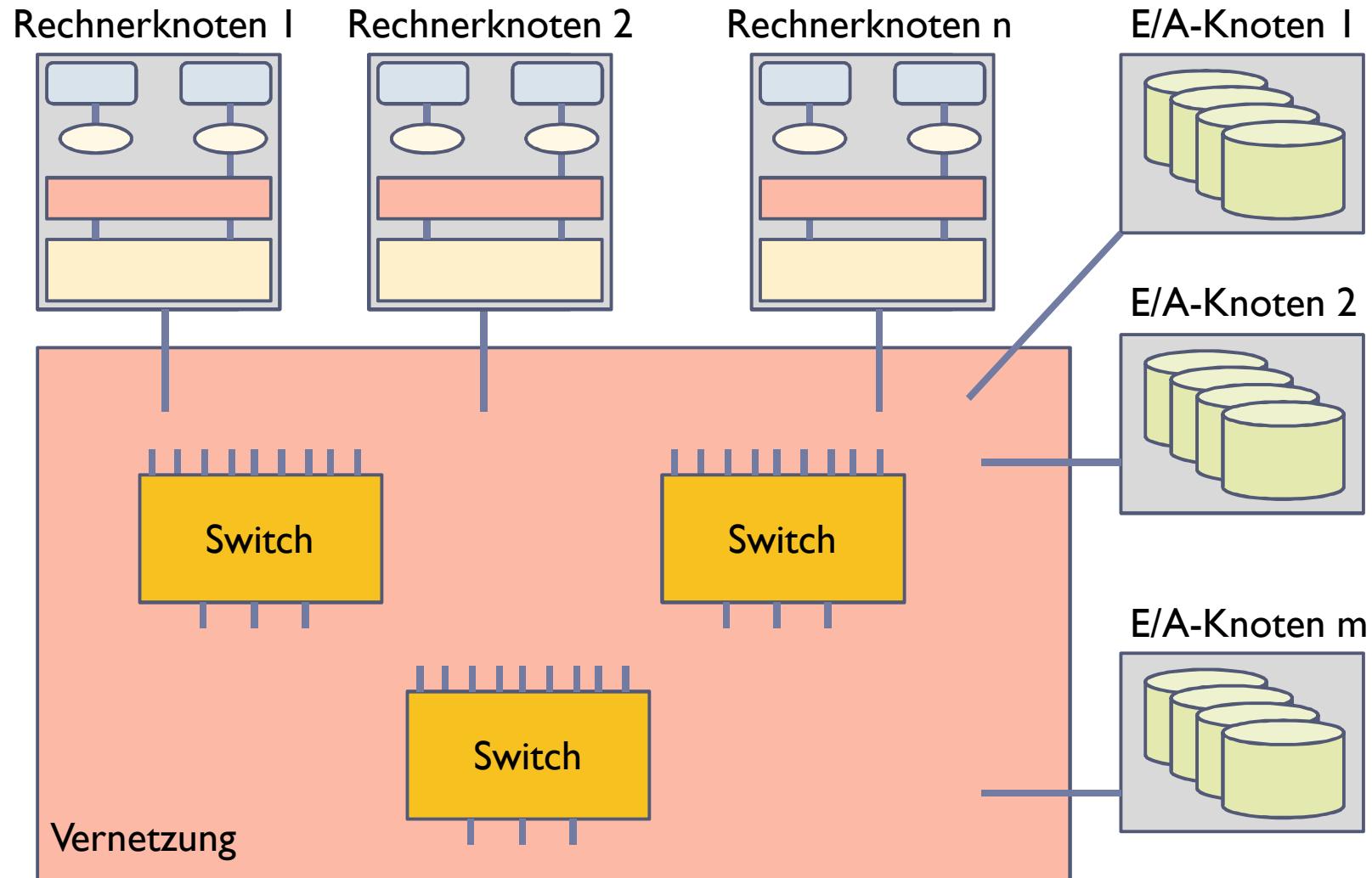
Bestandteile der Vernetzung

- ▶ NIC (network interface card) – mindestens eine pro Rechner
- ▶ Verbindungsleitungen: Kupfer, Glasfaser
- ▶ Switches – zur wechselseitigen Verbindung von Komponenten (Rechnern, E/A-Knoten)

Gegebenenfalls getrennte Vernetzung für

- ▶ Prozeßkommunikation
- ▶ Eingabe/Ausgabe zu Dateisystemen und Bandarchiv
- ▶ Wartung und Kontrolle der Rechner

# Vernetzung von Rechnern und E/A



# Allgemeine Betrachtungen zur Software

---

Was interessiert uns an der Vernetzung?

- ▶ Programmierschnittstelle hoher Abstraktion
  - ▶ Portabilität der Programme wichtig
- ▶ Geringe Zusatzlast
- ▶ Hardware voll ausnutzbar
- ▶ Anpaßbar an unterschiedliche Realisierungen der Vernetzungshardware
- ▶ Software zum Netzmanagement

# Allgemeine Betrachtungen zur Hardware

---

Was interessiert uns an der Vernetzung?

- ▶ Datenraten
- ▶ Latenzzeiten
- ▶ Bestandteile der Vernetzung
  - ▶ Kabel: Kupfer, Glasfaser
  - ▶ Netzkomponenten: Karten, Switches
- ▶ Ausfallsicherheit
- ▶ Adaptive Wegewahl
  - ▶ Bei Überlast / bei Fehlern
- ▶ Anbindung an TCP/IP

# Designaspekte effizienter Kommunikation

---

Überblick über wichtige Designaspekte

- ▶ Pufferverwaltung
- ▶ Überlappung von Berechnung und Kommunikation
- ▶ Realisierung in Hardware
- ▶ Datentransport

# Pufferverwaltung (1/2)

---

Warum ist Pufferverwaltung relevant?

- ▶ Verwaltung von Speicherplatz ist sehr teuer
- ▶ Umkopiovorgänge sind sehr teuer

Aufgabenstellung

- ▶ Nachricht steht sendebereit im Adreßraum des sendenden Prozesses
- ▶ Nachricht durch alle Softwareschichten und das Netz in den Speicher des Empfängers befördern
- ▶ Nachricht im Adreßraum des Empfängers zur Verfügung stellen

# Pufferverwaltung (2/2)

---

- ▶ Zero-Copy-Mechanismen
  - ▶ Am besten aus einem Adreßraum sofort in den Netzadapter übertragen – schwierig!
- ▶ Speicherregistrierung
  - ▶ Moderne Vernetzungen gestatten Remote Direct Memory Access (RDMA)
  - ▶ Setzt die Registrierung von Speicherbereichen voraus – zeitaufwendig
- ▶ Unerwartete Nachrichten
  - ▶ Der Empfänger hat keine Kenntnis, daß eine Nachricht eintreffen wird
  - ▶ Entsprechend sind keine Puffer allokiert und der Ablauf verlangsamt sich

# Überlappung von Berechnung und Kommunikation

---

Welche Phasen sehen wir bei der Kommunikation?

- ▶ Daten aus der Anwendung zum Sendenetzadapter
- ▶ Daten vom Sendenetzadapter zum Empfangsnetzadapter übertragen
- ▶ Daten vom Empfangsnetzadapter zur Anwendung

Was soll überlappend stattfinden?

- ▶ Am besten alle drei Phasen!

Was kann aktuelle Hardware

- ▶ Verschieden gute Varianten der optimalen Lösung

Noch problematisch:

- ▶ Kann die Software das ausnutzen?

# Realisierung in Hardware

---

Moderne Netztechnologien gestatten die Abarbeitung eines Teils der Software-Schichten in der Adapterhardware (genannt offloading)

Diese Hardware nennt man für TCP/IP:

- ▶ TCP/IP offload Engine, kurz ToE

Erhöht gegebenenfalls die Überlappung von Berechnung und Kommunikation

# Datentransport

---

- ▶ Zuverlässigkeit: weniger kritisch als in WANs
- ▶ Paketgröße und MTU
  - ▶ IP-Paket: max. 64 KB, Ethernet Standard: 1.500 Byte
  - ▶ Ethernet verwendet sog. Jumbo-Frames
- ▶ DMA-basiert
  - ▶ Entlastet Prozessor
- ▶ Unterbrechungen und Polling (Abfrage)
  - ▶ Unterbrechungen sind schwergewichtig
  - ▶ Polling benötigt Zeit vom Prozessor
  - ▶ Wir finden beide Realisierungen

# Leistungsentwicklung

---

In den Jahren von 1990-2010 sehen wir verschiedene Generationen von internen Bussen und externen Netztechnologien

Die Geschwindigkeitssteigerungen sind viel geringer als bei den Rechnern!

# Bussysteme

|                                |  |   |
|--------------------------------|--|---|
| PCI                            | 1990   | 33MHz/32bit: 1.05Gbps (shared bidirectional)  |
| PCI-X                          | 1998 (v1.0)<br>2003 (v2.0)                           | 133MHz/64bit: 8.5Gbps (shared bidirectional)<br>266-533MHz/64bit: 17Gbps (shared bidirectional)   |
| HyperTransport (HT)<br>by AMD  | 2001 (v1.0), 2004 (v2.0)<br>2006 (v3.0), 2008 (v3.1) | 102.4Gbps (v1.0), 179.2Gbps (v2.0)<br>332.8Gbps (v3.0), 409.6Gbps (v3.1)  |
| PCI-Express (PCIe)<br>by Intel | 2003 (Gen1), 2007 (Gen2)<br>2009 (Gen3 standard)     | Gen1: 4X (8Gbps), 8X (16Gbps), 16X (32Gbps)<br>Gen2: 4X (16Gbps), 8X (32Gbps), 16X (64Gbps)<br>Gen3: 4X (~32Gbps), 8X (~64Gbps), 16X (~128Gbps) |
| Intel QuickPath                | 2009   | 153.6-204.8Gbps per link  |

# Vernetzungstechnologien

|                               |                       |
|-------------------------------|-----------------------|
| Ethernet (1979 - )            | 10 Mbit/sec           |
| Fast Ethernet (1993 - )       | 100 Mbit/sec          |
| Gigabit Ethernet (1995 - )    | 1000 Mbit /sec        |
| ATM (1995 - )                 | 155/622/1024 Mbit/sec |
| Myrinet (1993 - )             | 1 Gbit/sec            |
| Fibre Channel (1994 - )       | 1 Gbit/sec            |
| InfiniBand (2001 - )          | 2 Gbit/sec (1X SDR)   |
| 10-Gigabit Ethernet (2001 - ) | 10 Gbit/sec           |
| InfiniBand (2003 - )          | 8 Gbit/sec (4X SDR)   |
| InfiniBand (2005 - )          | 16 Gbit/sec (4X DDR)  |
|                               | 24 Gbit/sec (12X SDR) |
| InfiniBand (2007 - )          | 32 Gbit/sec (4X QDR)  |
| InfiniBand (2011-)            | 64 Gbit/sec (4X EDR)  |

# Leistungsmaße

---

## Wünschenswerte Charakteristika

- ▶ Niedrige Latenz (Aufsetzzeit der Nachrichten)
- ▶ Hohe Bandbreite
- ▶ Geringe Belastung der CPU
- ▶ Hohe Bisektionsbandbreite des Netzes

## Vorgriff auf Programmierkonzepte

- ▶ Möglichst wenig Kommunikation verwendet
- ▶ Wenn überhaupt, dann lieber wenige große als viele kleine Pakete

# Netztopologien

---

## Wünschenswerte Eigenschaften

- ▶ Identische Datenraten zwischen beliebigen Knoten
- ▶ Vermeidung von Engpässen, Überlastungen, Ausfällen
- ▶ Erweiterbarkeit
- ▶ Skalierbarkeit
- ▶ Gutes Preis/Leistungsverhältnis

# Bisektion des Netzes

---

## Bisektionsbreite

Anzahl der Verbindungen, die man trennen muß,  
damit das Netz in zwei isolierte Teile mit gleicher  
Knotenzahl zerfällt

- ▶ Je mehr, desto besser

## Bisektionsbandbreite

Aggregierte Bandbreite der durchtrennten Leitungen  
(=Fluß an der engsten Stelle)

# Bisektion des Netzes

---

## Volle Bisektionsbandbreite

Ein Netz mit  $n$  Knoten hat volle Bisektionsbandbreite, wenn die Summe aller Bandbreiten von Verbindungen zwischen zwei beliebigen Hälften des Netzes  $n/2$ -mal die Bandbreite einer einzelnen Verbindung ist

## Warum interessiert uns das?

- › Wir benötigen ausreichend Switches und Wege, um alle Komponenten leistungsfähig zu vernetzen

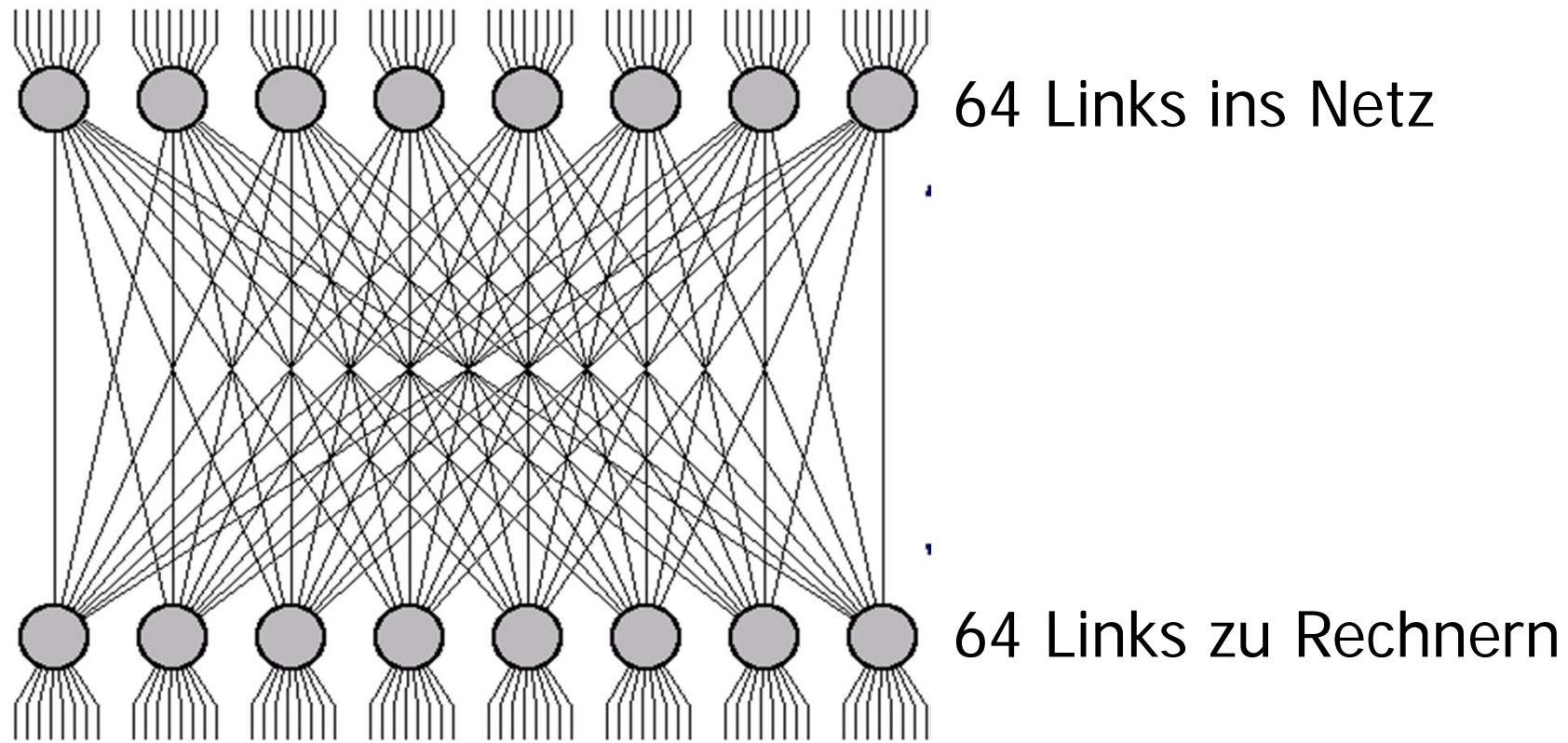
# Bisektion am Beispiel Myrinet

- ▶ Basisbaustein: Kreuzschiene mit 16 Anschlüssen
- ▶ Topologie benannt nach Charles Clos (1952)

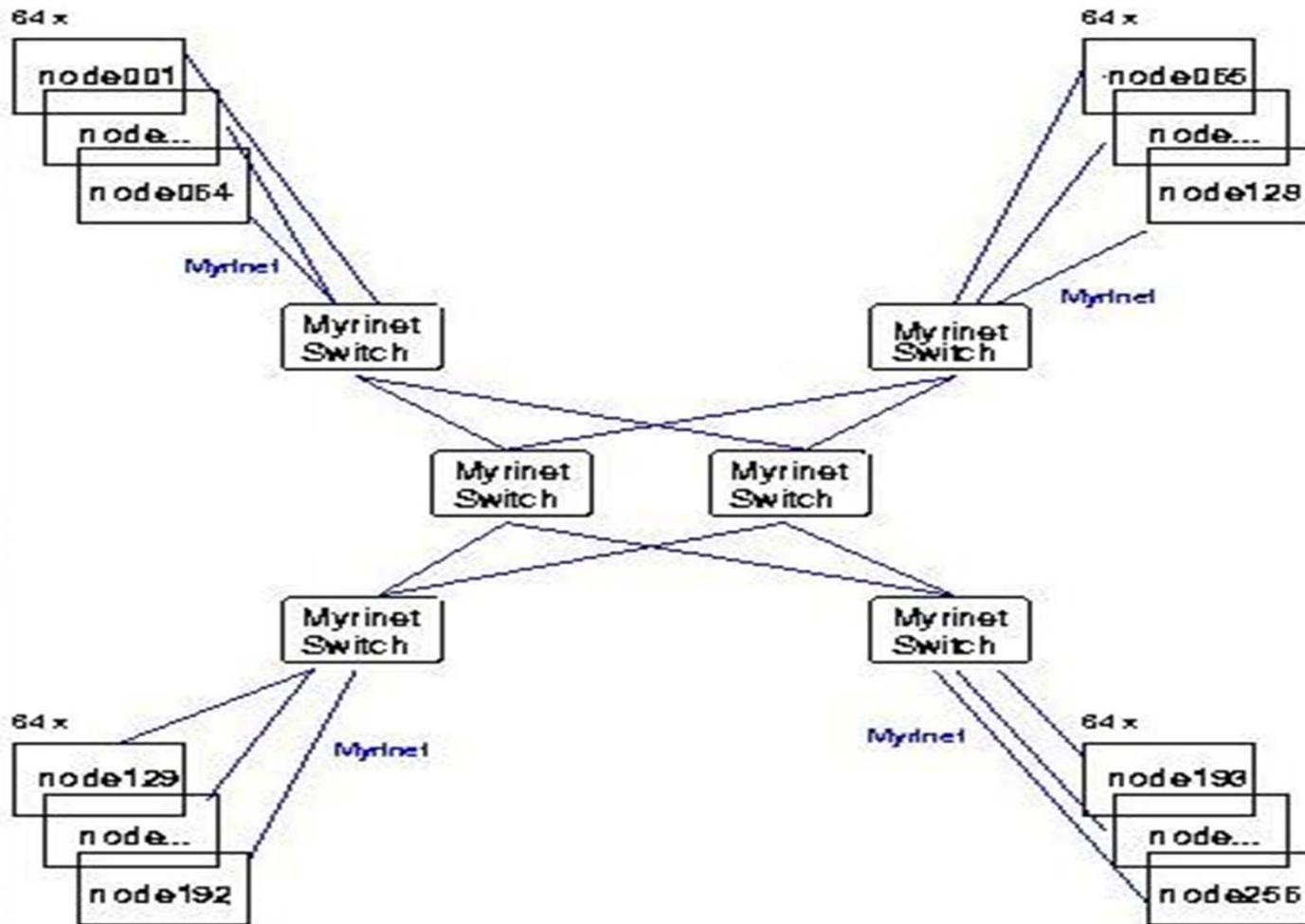


# Myrinet-Switch

Neue Basiskomponente aus 16 des bisherigen Typs

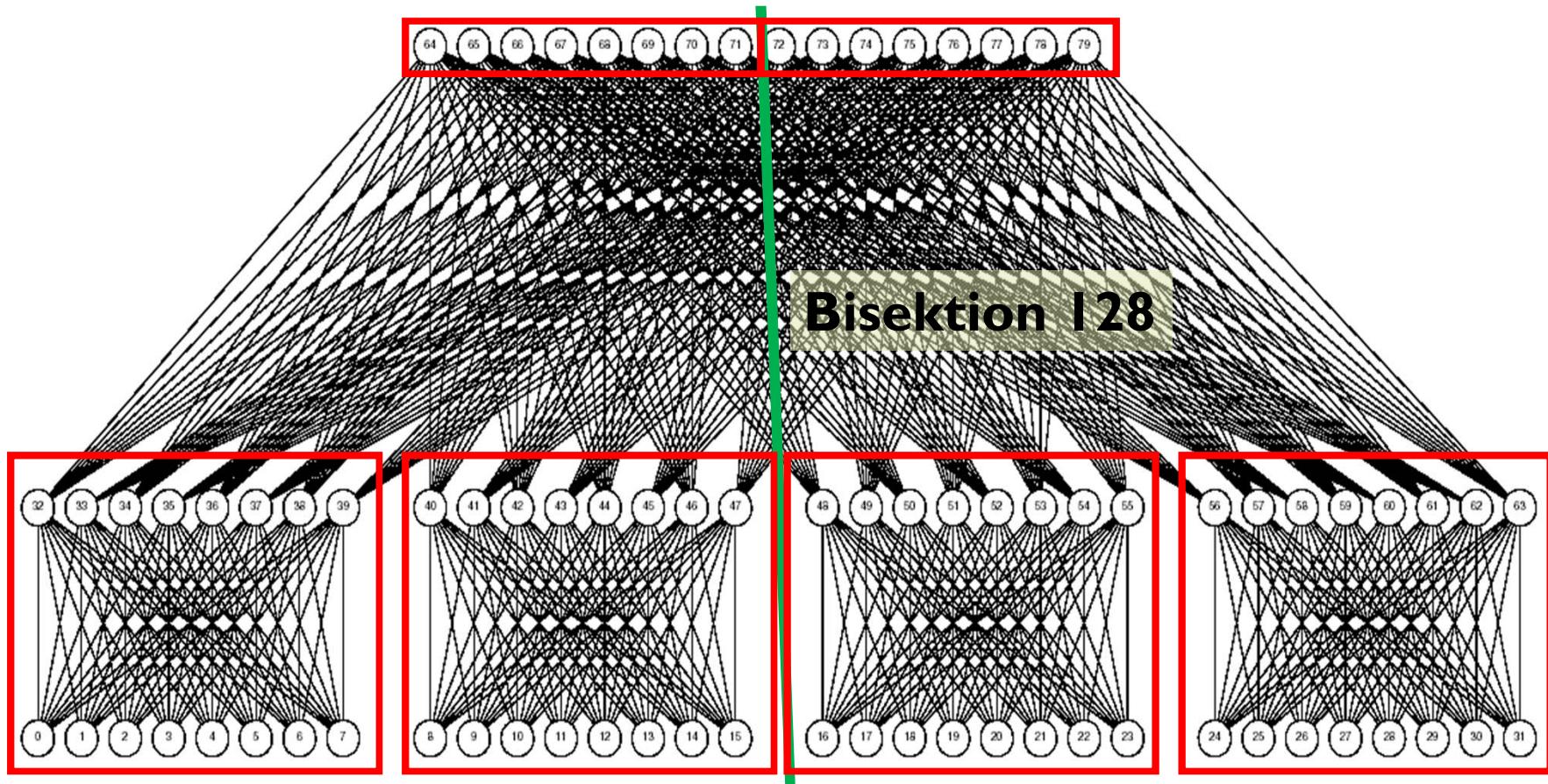


# Heidelberger Helics-Cluster mit Myrinet



# Myrinet im Detail...

4 Clos64-Bausteine und 2 Switche für 256 Rechner



# ... und in der Praxis



# Einbindung in TCP/IP

---

- ▶ Aus Sicht des Programms
  - ▶ Socket-Programmierung
  - ▶ TCP/IP-Stack im Betriebssystem
- ▶ Aus Sicht der Hardware
  - ▶ Varianten von Ethernet sehr populär
- ▶ Im Rechner-Cluster praktisch immer auch TCP/IP involviert
- ▶ Problem: TCP/IP ist schwerfällig

# Einbindung in TCP/IP

---

- ▶ Probleme mit TCP/IP
  - ▶ Viele Protokollsichten
  - ▶ Nicht geringe Prozessorbelastung
  - ▶ Integration in das Betriebssystem
  - ▶ Kopiervorgänge
  - ▶ Anzahl der Unterbrechungen

# Einbindung in TCP/IP

---

- ▶ Lösungsansätze
  - ▶ TCP-Bypass
    - ▶ Definition eines eigenen Paketformats
    - ▶ Evtl. Problem: nur cluster-lokal verwendbar
  - ▶ TCP Offload Engine
    - ▶ Z.B. eine GigE Verbindung kann einen Pentium IV mit 2,4 GHz auslasten
    - ▶ Deshalb extra Prozessor für TCP/IP-Protokollstapel
    - ▶ Normalerweise auf der NIC untergebracht
  - ▶ Kernel Bypass
    - ▶ Die NIC kommuniziert direkt mit dem Programm
    - ▶ Dies findet man bei allen nicht Ethernet-Vernetzungen!

# Einbindung in TCP/IP

---

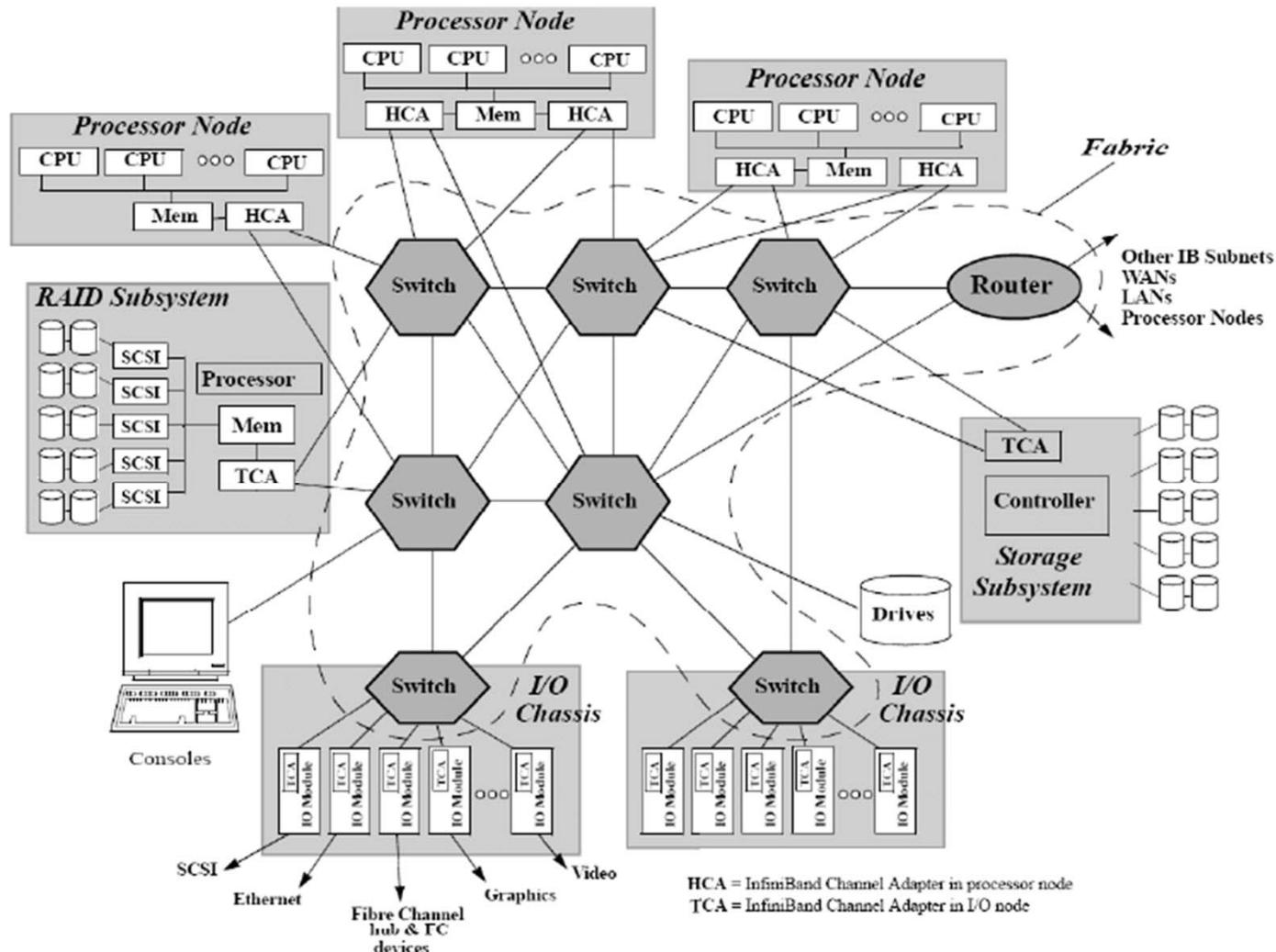
- ▶ Lösungsansätze...
  - ▶ Remote Direct Memory Access (RDMA)
    - ▶ Die NIC kann direkt in den Speicher eines entfernten Rechners schreiben
  - ▶ Zero-Copy Networking
    - ▶ Vermeide Kopien zwischen Betriebssystem und Anwendungprogramm
    - ▶ Verwende stattdessen DMA und MMU
  - ▶ Interrupt Mitigation
    - ▶ Fasse mehrere Unterbrechungen zu einer zusammen

# InfiniBand-Vernetzung

---

- ▶ InfiniBand ist ein Industriestandard einer Gruppe von Herstellern genannt Open Fabrics Alliance
- ▶ Definiert einen Standard für ein Systemnetz
  - ▶ Früher auch als Spezifikation eines rechnerinternen Busses bekannt – dieser Ansatz wurde nicht weiter verfolgt
- ▶ Unterscheidung zwischen Rechnern und E/A-Geräten
  - ▶ Für Rechner verwendet: Host Channel Adapter (HCA)
  - ▶ Für E/A-Geräte verwendet: Target Channel Adapter (TCA)

# InfiniBand-Netz: ein Überblick



# InfiniBand-Software

---

- ▶ IP over InfiniBand (IPoIB)
  - ▶ Ermöglicht IP-basierten Protokollen eine Kommunikation über InfiniBand
  - ▶ Durchsatz von 300 MByte/s ist höher als der von Gigabit-Ethernet (GE)
  - ▶ Latenz von 20 µs vergleichbar zu GE
- ▶ Sockets direct protocol (SDP)
  - ▶ Durchsätze von 900 MByte/s
  - ▶ Latenzen von 12 µs

# Vernetzungskonzepte

## Zusammenfassung

- ▶ Die Vernetzung bringt Rechner miteinander und mit den E/A-Geräten in Verbindung
- ▶ Vernetzungshardware sind Netzadapter, Kabel, Switches
- ▶ Wir möchten hohe Datenraten und geringe Latenzen bei gleichzeitiger guter Skalierbarkeit
- ▶ Effiziente Kommunikation umfaßt viele Einzelaspekte
- ▶ In den letzten 10 Jahren haben sich Rechnergeschwindigkeiten 10x schneller gesteigert als Netzgeschwindigkeiten
- ▶ Ein wichtiges Maß der Topologie ist die Bisektionsbandbreite
- ▶ TCP/IP ist schwerfällig und wird häufig ersetzt
- ▶ InfiniBand ist die aktuelle Hochleistungsvernetzung beim Hochleistungsrechnen

# Hochleistungs- Eingabe/Ausgabe

---

- ▶ Motivation
- ▶ Abstraktionsebenen
- ▶ Traditionelle und moderne E/A
- ▶ E/A-Klassen bei numerischen Anwendungen
- ▶ Benutzungsschnittstellen
- ▶ Parallel Virtual File System (PVFS/PVFS2)
- ▶ Forschungsthemen (allgemein und hier)

# Hochleistungs-Eingabe/Ausgabe

## Die zehn wichtigsten Fragen

- ▶ Warum ist E/A auf einmal ein Thema?
- ▶ Wie hantiert man mit Daten in Dateien?
- ▶ Welche Abstraktionsschichten unterscheidet man?
- ▶ Welche Varianten der E/A finden wir bei numerischen Anwendungen?
- ▶ Welche Benutzungsschnittstellen gibt es?
- ▶ Wie funktioniert E/A im Cluster?
- ▶ Wie funktioniert E/A im Hochleistungsrechner?
- ▶ Was ist ein paralleles Dateisystem?
- ▶ Wie funktioniert das Parallel Virtual File System?
- ▶ Welche offenen Fragestellungen gibt es?

# Warum ist das ein Thema?

---

- ▶ Gespeicherte Datenmengen steigen stark an
  - ▶ Berkeley-Bericht „How Much Information? 2003“
    - ▶ 2002: 5 Exabyte **mehr** abgespeichert
    - ▶ 92% davon auf magnetische Medien, meistens Festplatten
  - ▶ IDC-Report sieht für 2009 800 Exabyte gespeichert
    - ▶ Vermutet für 2010 einen Zuwachs von 400 Exabyte
- ▶ Hauptspeichergrößen steigen stark an
  - ▶ Heute einzelne Gigabyte pro Rechner
    - ▶ Z.B. DKRZ/Blizzard: 20 TByte Hauptspeicher, 6 Pbyte Platten
    - ▶ Füllt 5 aktuelle große Festplatten
      - ▶ vgl. PC: 4 GByte Hauptspeicher vs. 4 TByte Platte
    - ▶ 1 TByte lesen bei 50 MByte/s dauert 20.000 s

# Speicherung großer Datenmengen

---

- ▶ Datenaufkommen
  - ▶ Besonders hoch in den Naturwissenschaften  
Klimaforschung, Physik, Biologie, Astronomie, ...
- ▶ Zugriff
  - ▶ Alle Anwender wollen immer alle Daten aufheben und sie jederzeit zugreifbar haben
- ▶ Verfügbarkeit
  - ▶ Die Datenspeicherung soll mit Fehlern in der Hardware problemlos umgehen können
- ▶ Sicherheit
  - ▶ Daten müssen vor Einblick, Veränderung und Löschen geschützt werden

# Komplexere E/A-Systeme

---

- ▶ RAID – Redundant Array of Inexpensive Disks
- ▶ MAID – Massive Array of Idle Disks
- ▶ JBOD – Just a Bunch of Disks
  
- ▶ SAN – Storage Area Network (blockorientiert)
- ▶ NAS – Network Attached Storage (dateiorientiert)
  
- ▶ Dateisysteme mit verteiltem Zugriff
  - ▶ NFS – Network File System
  - ▶ AFS – Andrew File System
- ▶ Dedizierte Spezial-Hardware in Hochleistungsrechnern

# Abstraktionsebenen

---

Begriff der „parallelen Eingabe/Ausgabe“

▶ Programmsicht:

Der Zugriff auf die Bytes *einer* Datei erfolgt aus mehreren parallelen Prozesse heraus

▶ Systemsicht:

Die Bytes einer Datei liegen über mehrere Platten verteilt

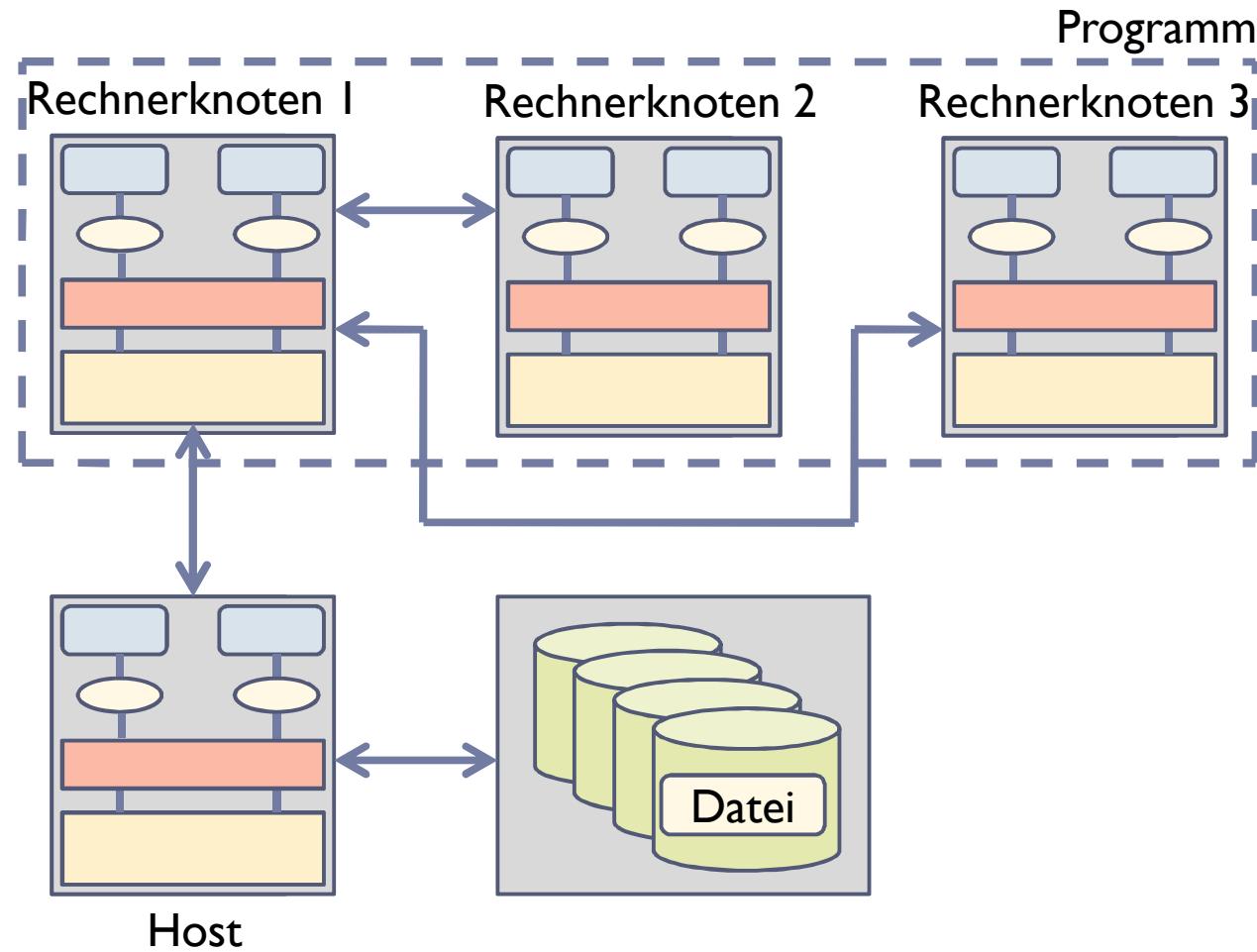
Wie üblich: Ebenen voneinander unabhängig

# Varianten der E/A

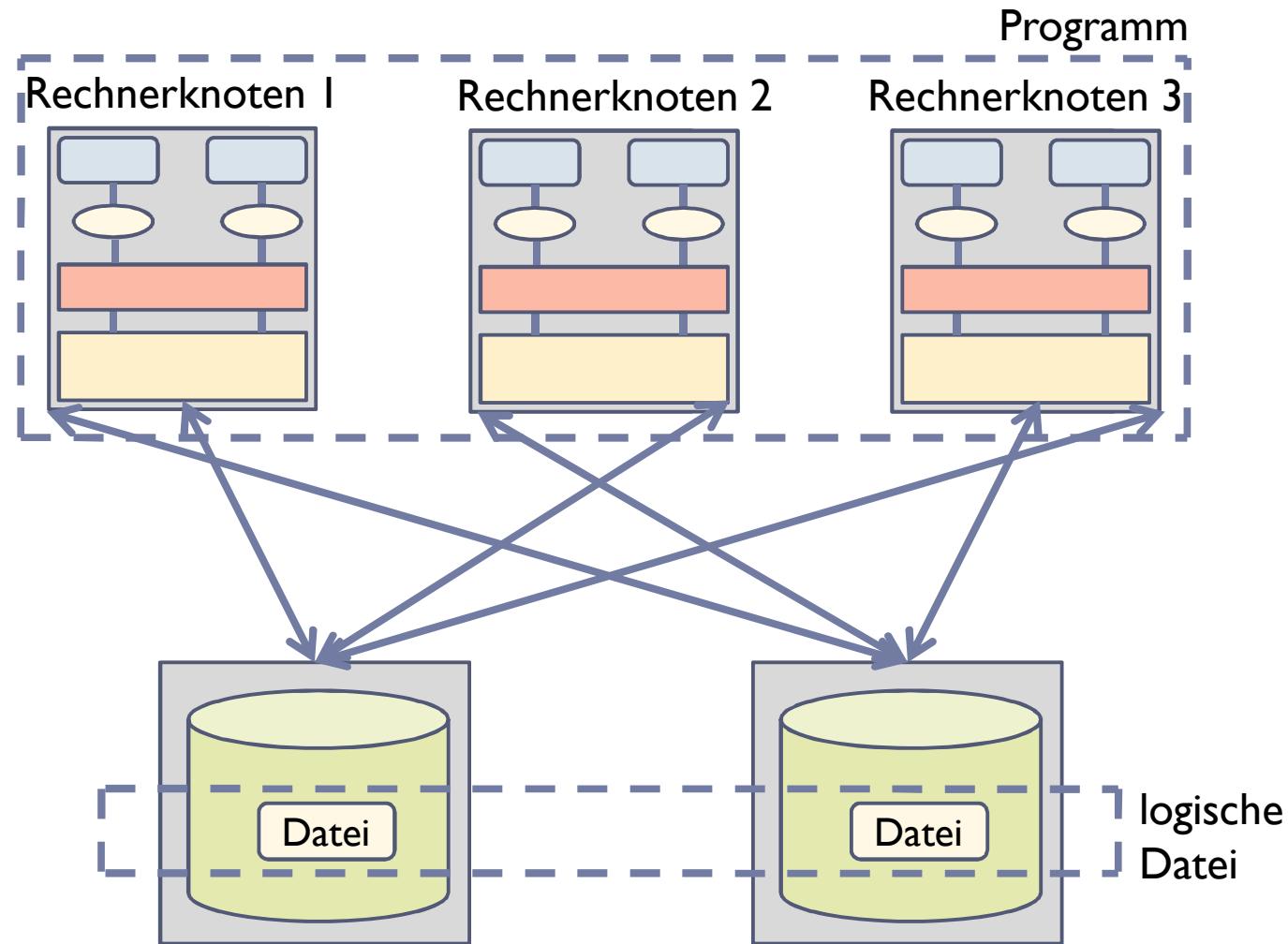
---

- ▶ Traditionell
  - ▶ E/A nur über Hostrechner
  - ▶ E/A nur durch festgelegten Prozeß
- Bewertung
  - ▶ Aus Effizienzgründen nicht mehr möglich
- ▶ Modern
  - ▶ E/A über viele Knoten
  - ▶ E/A durch alle Prozesse
- Bewertung
  - ▶ Verwendung moderner Techniken

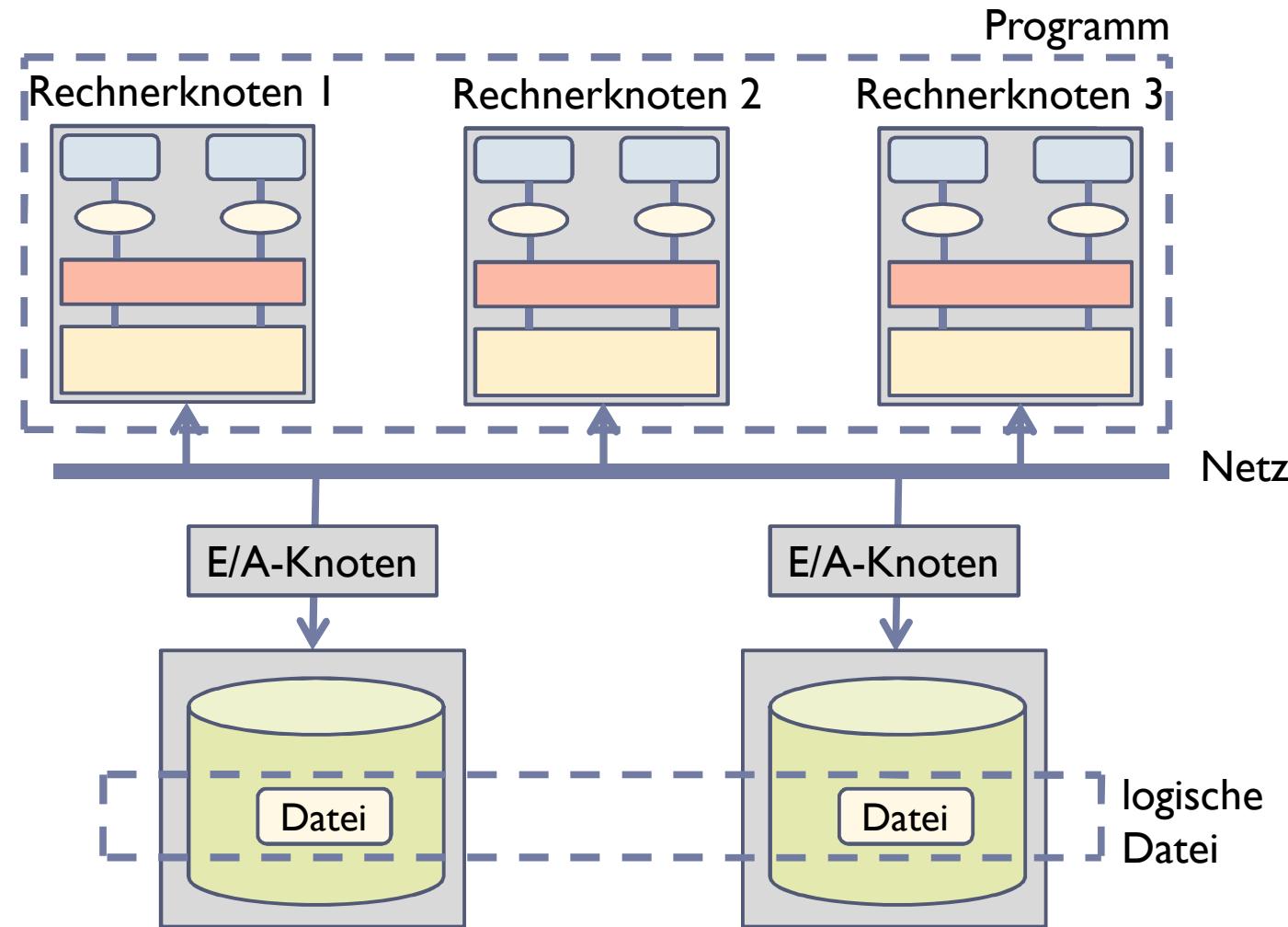
# Traditionelle E/A



# Moderne E/A



# Moderne E/A...



# Geschichte der parallelen E/A

---

- ▶ Parallele E/A-Systeme und parallele E/A-Bibliotheken entstehen Anfang der 90er Jahre
- ▶ Viel Forschung bereits Mitte der 90er
- ▶ Nur wenige existierende Systeme im Produktionsbetrieb
- ▶ Aber: Sehr viele proprietäre Hochleistungs-E/A-Systeme bei Hochleistungsrechnern; jedoch nicht so oft echt parallele

# Kategorien massiver E/A

---

## Anwendungssicht

- ▶ Nichtnumerische Anwendungen
  - ▶ Unregelmäßig strukturierte Daten
    - z.B. Datenbank-Anwendungen
    - ▶ Datenströme
      - z.B. Multimedia-Anwendungen
- Beides hier nicht weiter betrachtet
- ▶ Numerische Anwendungen
  - ▶ Regelmäßig strukturierte Daten
    - z.B. Vektoren, Arrays mit großen Dimensionen
    - ▶ Auch unregelmäßig strukturierte Daten
      - Listen, dünnbesetzte Matrizen



# E/A-Klassen bei numerischen Anwendungen

---

- ▶ Lesen der Programmeingabe und Schreiben der Programmausgabe
- ▶ Sicherungspunkte
- ▶ Temporäre Daten
- ▶ „Out-of-core Execution“

# Programmeingabe/-ausgabe

---

- ▶ Zeitpunkte
  - ▶ Programmstart, Programmende, Zwischenergebnisse
- ▶ Datenmengen
  - ▶ Maximal Größe des gesamten Hauptspeichers
- ▶ Wichtiges Szenarium: Pipelining
  - ▶ Daten von einem anderen Gerät  
z.B. von physikalischem Experiment
  - ▶ Daten zu einem anderen Gerät  
z.B. Ergebnisvisualisierung, Datenarchivierung

# Sicherungspunkte

---

- ▶ Verwendet zur Programmfortsetzung
  - ▶ Nach Absturz oder Unterbrechung
- ▶ Sichern aller wichtigen Daten
  - ▶ Kompletter Speicherabzug
  - ▶ Vom Benutzer ausgewählte Daten
- ▶ Anzahl benötigter Sicherungspunkte
  - ▶ Mindestens zwei
- ▶ Redundanz der Daten notwendig zur Ausfallsicherung

# Temporäre Dateien

---

- ▶ Abspeicherung während des Programmlaufs
- ▶ Evtl. nicht-parallele E/A auf lokaler Platte ausreichend
- ▶ Zur Kommunikation zwischen Prozessen aber parallele E/A erforderlich
  - ▶ Verwendung einer temporären Datei über alle Platten hinweg

# Out-of-Core-Execution

---

- ▶ Out-of-core-Execution:  
Bearbeitung einer größeren Datenmenge, als in den  
Hauptspeicher paßt
  - ▶ Eigenprogrammiertes Aus- und Einlagern der überschüssigen  
Datenmengen
  - ▶ Effizienter als Swapping durch Betriebssystems
- ▶ Spezialfall für Spezialanwendungen
  - ▶ Bei numerischen Anwendungen wird typischerweise der  
Hauptspeicher exakt gefüllt

# Zugriffsmuster bei E/A

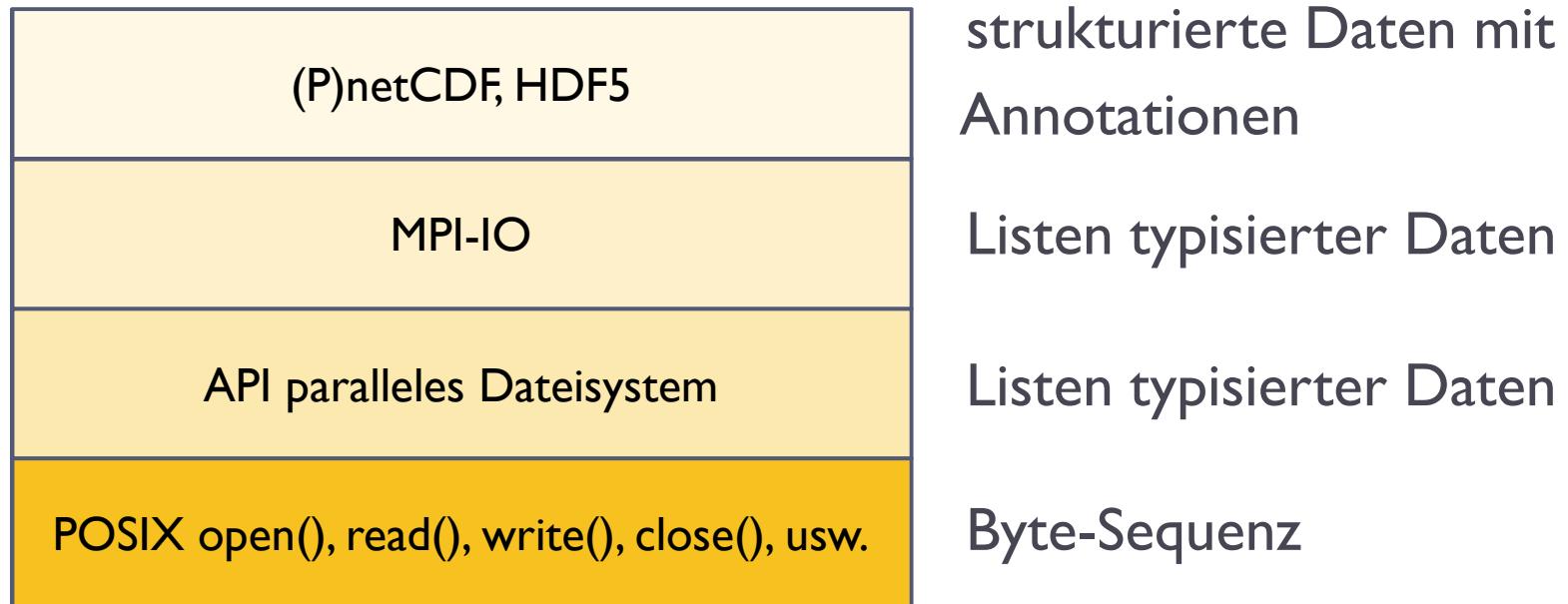
---

- ▶ Wichtig für Fall 1: E/A bei Programmen
- ▶ Fragestellung
  - ▶ Wann wird E/A durchgeführt?
  - ▶ Welche Mengen werden transferiert?
  - ▶ Welche Byte einer Datei werden angesprochen?
- ▶ Ausführliche Studien aus den 90ern
- ▶ Wir müssen hier lernen, was die Klimaforscher tun
- ▶ Wichtig um die Abbildung der logischen Datei auf die physikalische zu optimieren

# Benutzungsschnittstellen

- ▶ Hierarchie von Schnittstellen

Im Moment nur paralleles Dateisystem betrachtet



# E/A im Rechnercluster

---

- ▶ Gelegentlich: An jedem Knoten auch eine Platte
  - ▶ Entweder nur für temporäre Dateien
  - ▶ Oder als richtiges paralleles Dateisystem über alle Platten hinweg
- ▶ Alternativen
  - ▶ Jeder Rechenknoten ist auch E/A-Knoten
  - ▶ Dedizierte E/A-Knoten
    - Balanzierung der Rechen- und E/A-Leistung
- ▶ Betriebsproblematik
  - ▶ Wer darf wann welche Platten benutzen?  
(Bisher nur Konzepte für Knotenzuteilung)

# E/A im Hochleistungsrechner

---

- ▶ Knoten haben nie Platten
- ▶ Ausgewählte Knoten dienen als E/A-Knoten
  - ▶ Dicker Netzanschluß
  - ▶ Fetter Hauptspeicherausbau
  - ▶ Keine Programme auf diesen Knoten
  - ▶ Keine eigenen Festplatten
- ▶ Wie geht es?
  - ▶ E/A-Knoten leitet die gesamte E/A zum E/A-System bestehend aus eigenen Rechnern und sehr vielen Platten

# Paralleles Dateisystem

---

- ▶ Merkmale
  - ▶ Mehrere Prozesse können gleichzeitig auf dieselben Dateien zugreifen.
  - ▶ Die Daten einer Datei liegen physikalisch verteilt
- ▶ Schicht
  - ▶ Zwischen dem Anwenderprogramm und dem physikalischen E/A-System der E/A-Knoten
  - ▶ Somit typische Middleware-Software

# Systeme

---

- ▶ GPFS (Cluster-Dateisystem)
  - ▶ Produkt der IBM; ausgreiftes System
- ▶ PVFS/PVFS2 (Paralleles Dateisystem)
  - ▶ Verbreitetes System bei Selbstbau-Clustern
- ▶ Lustre (Cluster-Dateisystem)
  - ▶ Neuerer Ansatz, in dem alles besser gemacht wird
  - ▶ Sehr hohe Komplexität!
  - ▶ Open-Source und frei erhältlich

# Beispiel: Parallel Virtual File System

---

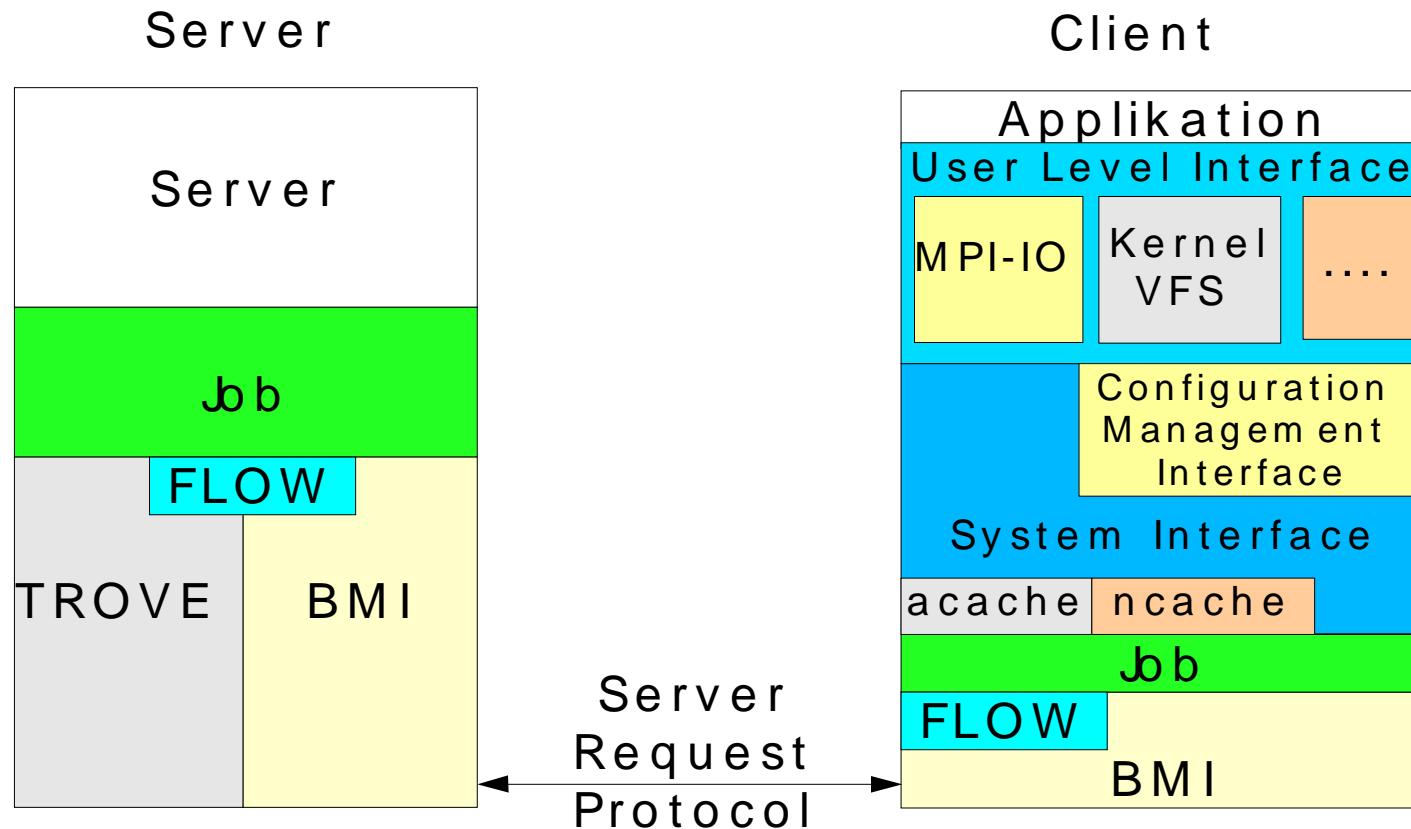
- ▶ Ziel: E/A massiver Datenmengen in Clustern
- ▶ Entwicklergruppen
  - ▶ Parallel Architecture Research Laboratory  
Clemson University
  - ▶ Mathematics and Computer Science Division  
Argonne National Laboratories
- ▶ Historie
  - ▶ Beginn ~1996
  - ▶ PVFS2 (Herbst 2003) umbenannt in PVFS 2.0

# PVFS2-Eigenschaften

---

- ▶ Komplette Neuentwicklung (ggü. PVFS)
- ▶ Setzt auf realem Dateisystem auf (ext[23] o.ä.)
- ▶ Modular und hierarchisch aufgebaut
  - ▶ Module zum Auswechseln vorgesehen
- ▶ Enge MPI-IO-Integration
- ▶ Effiziente Durchführung strukturierter nicht-kontinuierlicher Zugriffe
- ▶ Zustandloser Objektzugriff
- ▶ Erweiterbare Datenverteilungsfunktion (striping usw.)
- ▶ Semantik des Dateisystems variabel
- ▶ Explizite Unterstützung paralleler Abläufe
- ▶ Redundantes Speichern von Daten und Metadaten

# PVFS2-Schichtenmodell



# Arbeitsbereich Wissenschaftliches Rechnen und parallele E/A-Systeme

---

Der Arbeitsbereich Wissenschaftliches Rechnen (ehemals Arbeitsgruppe „Parallele und verteilte Systeme“ an der Universität Heidelberg) verwendet PVFS und GPFS als Basis für eigene Forschungs- und Entwicklungsarbeiten auf dem Gebiet der parallelen E/A

Themen für Abschlußarbeiten und zur Mitarbeit sind vorhanden!



# Forschungsthemen

---

- ▶ Wie soll parallele E/A genutzt werden?
- ▶ Wie steigern wir Leistung und Skalierbarkeit?
- ▶ Wie erhöht man die Verfügbarkeit?
- ▶ Wie ermittelt man die Leistung des E/A-Systems?

Jetzt neu:

- ▶ Kombination mit Bandarchiv und Hierarchical Storage Management (HSM)

# Forschungsthema Nutzung

---

- ▶ Die Frage der Schnittstelle zum Programm ist noch offen
  - ▶ Welche Zugriffsvarianten (Semantiken)?
  - ▶ Schnittstellen auf welchem Abstraktionsniveau?
- ▶ Details im Vortrag zu MPI-IO

# Forschungsthema Leistung

---

- ▶ Zugriffsmuster erkennen
- ▶ Abbildung logische Daten auf physikalische Daten optimieren oder dynamisch gestalten
- ▶ Nichtzusammenhängende Zugriffe optimieren
- ▶ Kollektive Operationen unterstützen
- ▶ Metadatenzugriff verbessern

Allgemein: Skalierbarkeit steigern

# Forschungsthema Verfügbarkeit

---

- ▶ Wie behandeln wir Plattenausfälle?
- ▶ Kurzfristige Verfügbarkeit
  - ▶ Abhängig von Datensemantik und Überlappung Rechen-E/A-Knoten
  - ▶ Nur Sicherungspunktdaten sichern
- ▶ Langfristige Verfügbarkeit
  - ▶ Datenverlust keinesfalls akzeptabel
  - ▶ Fehlertoleranz z.B. durch Spiegelung

# Forschungsthema Leistungsbestimmung

---

- ▶ Keine standardisierten Benchmarks
- ▶ Was wollen wir messen?
  - ▶ E/A-Bandbreiten beim Datenzugriff
    - ▶ Verschiedene Zugriffsmuster
    - ▶ Verschiedene Datenmengen
    - ▶ Unabhängige/identische Orte in Dateien
  - ▶ Zugriffsraten beim Metadatenzugriff
- ▶ Zur Zeit keine vernünftigen quantitativen Vergleiche zwischen Systemen möglich

# Eigene Forschungen

---

- ▶ Leistungsvisualisierung
- ▶ Leistungsbewertung
- ▶ Modellierung und Simulation
- ▶ Metadatenverwaltung
- ▶ Anwendung im Produktionsbetrieb
- ▶ Zugriffsmuster
- ▶ Datenkomprimierung

**Mitarbeit ist willkommen!**

# Hochleistungs-Eingabe/Ausgabe

## Zusammenfassung

- ▶ Traditionelle Varianten der E/A jetzt ungenügend
- ▶ Parallelisierung der E/A notwendig und möglich
- ▶ Parallele E/A etwa ab Mitte der 90er entwickelt
- ▶ Uns interessieren hier nur numerische Anwendungen
- ▶ E/A gliedert sich in verschiedene Klassen auf
- ▶ Benutzerschnittstellen auf verschiedenen Ebenen
- ▶ Im Parallelen Dateisystem liegen die Dateien über Platten verteilt und werden von parallelen Prozessen aus gelesen und geschrieben
- ▶ PVFS ist prominenter Vertreter dieser E/A-Systeme
- ▶ Im Bereich E/A viele offene Forschungsfragen

# Betriebssystemaspekte

---

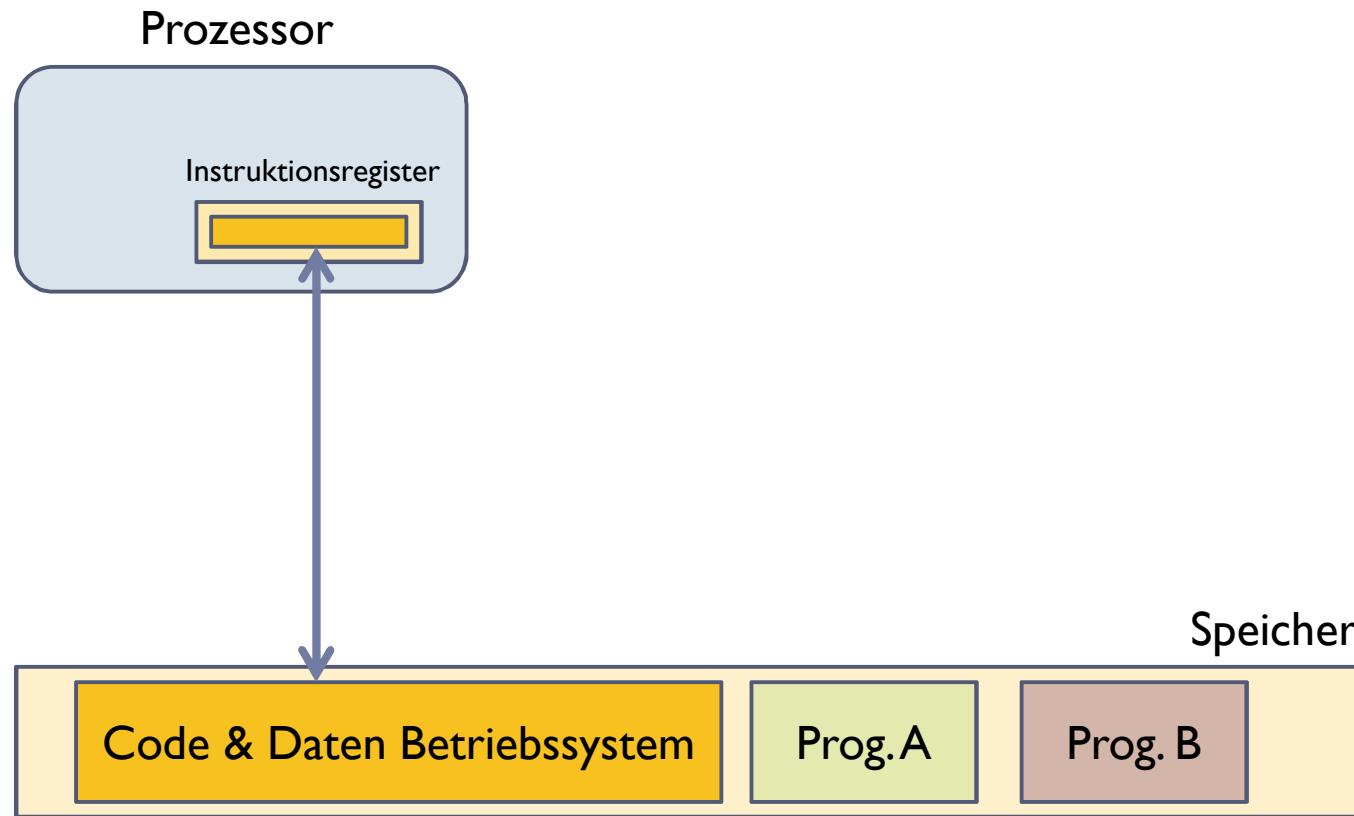
- ▶ Einprozessor-Einkern-Systeme
- ▶ Mehrprozessor/Mehrkern-Systeme
- ▶ SMP-Hardware
- ▶ SMP-Betriebssystem
- ▶ Synchronisationsmechanismen
- ▶ Eweiterte Betriebssystemfunktionalität

# Betriebssystemaspekte

## Die zehn wichtigsten Fragen

- ▶ Wie funktioniert Synchronisation im traditionellen UNIX-Kern?
- ▶ Was versteht man unter Wiedereintrittsfähigkeit?
- ▶ Wie werden Maschinenbefehle in Mehrprozessor/Mehrkern-Systemen abgearbeitet?
- ▶ Was benötige ich für ein Mehrprozessor-System?
- ▶ Was kennzeichnet ein SMP-Betriebssystem?
- ▶ Welche Varianten von Sperren finden wir im SMP-Betriebssystem?
- ▶ Wie wird ein Betriebssystem SMP-fähig?
- ▶ Wie funktioniert Synchronisation mittels Semaphor?
- ▶ Wie funktioniert Synchronisation mittels Spinlock?
- ▶ Welche weiteren Funktionen muß ein SMP-Betriebssystem aufweisen?

# Einkern-Einprozessor-System



# Moduswechsel

---

- ▶ Der Prozessor arbeitet
  - ▶ **entweder** im Betriebssystemmodus
    - ▶ und bearbeitet Code des Betriebssystems
  - ▶ **oder** im Benutzermodus
    - ▶ und bearbeitet Code des Benutzerprozesses

## Wechsel des Modus

- ▶ Systemaufruf im Programm (**synchron**)
- ▶ Unterbrechung (**asynchron**)

# Synchronisation im traditionellen UNIX-Kern

---

Feststellung: der UNIX-Kern ist wiedereintrittsfähig (reentrant)

- ▶ Mehrere Prozesse arbeiten im Kern zur selben Zeit und evtl. im selben Code-Bereich
- ▶ Natürlich nicht echt gleichzeitig sondern verschränkt !
  - ▶ Es gibt ja nur einen Prozessor

Frage: Könnte es zu inkonsistenten Daten kommen?  
Wenn ja, wie vermeidet man es?

- ▶ Die Situation könnte auftreten, wenn zwei Prozesse dieselben Daten, z.B. Puffer, benutzen
- ▶ Natürlich darf Dateninkonsistenz nicht auftreten

# Synchronisation im traditionellen UNIX-Kern ...

---

## Vermeidung von Inkonsistenzen

- ▶ Das Betriebssystem ist zunächst einmal nicht unterbrechbar (non-preemptive)
- ▶ D.h. eine BS-Aktivität wird zuendegeführt, auch wenn dadurch die Zeitscheibe des zugehörigen Prozesses überschritten wird
- ▶ Nach dem Ende sind alle Datenstrukturen konsistent und ein anderer Prozeß kann unbedenklich damit arbeiten

Aber ...

# Synchronisation im traditionellen UNIX-Kern ...

---

## Problem: Unterbrechungen

- ▶ Während der Aktivität im BS-Kern könnte eine Unterbrechung erfolgen
- ▶ Die Unterbrechungsbehandlungsroutine könnte zufällig dieselben Datenstrukturen bearbeiten

## Lösung:

- ▶ Vorübergehendes Verbieten von Unterbrechungen durch Erhöhung des *ipl* (interrupt priority level)
- ▶ Kritischer Bereich mit Erhöhung/Verringerung eingerahmt

# Synchronisation im traditionellen UNIX-Kern ...

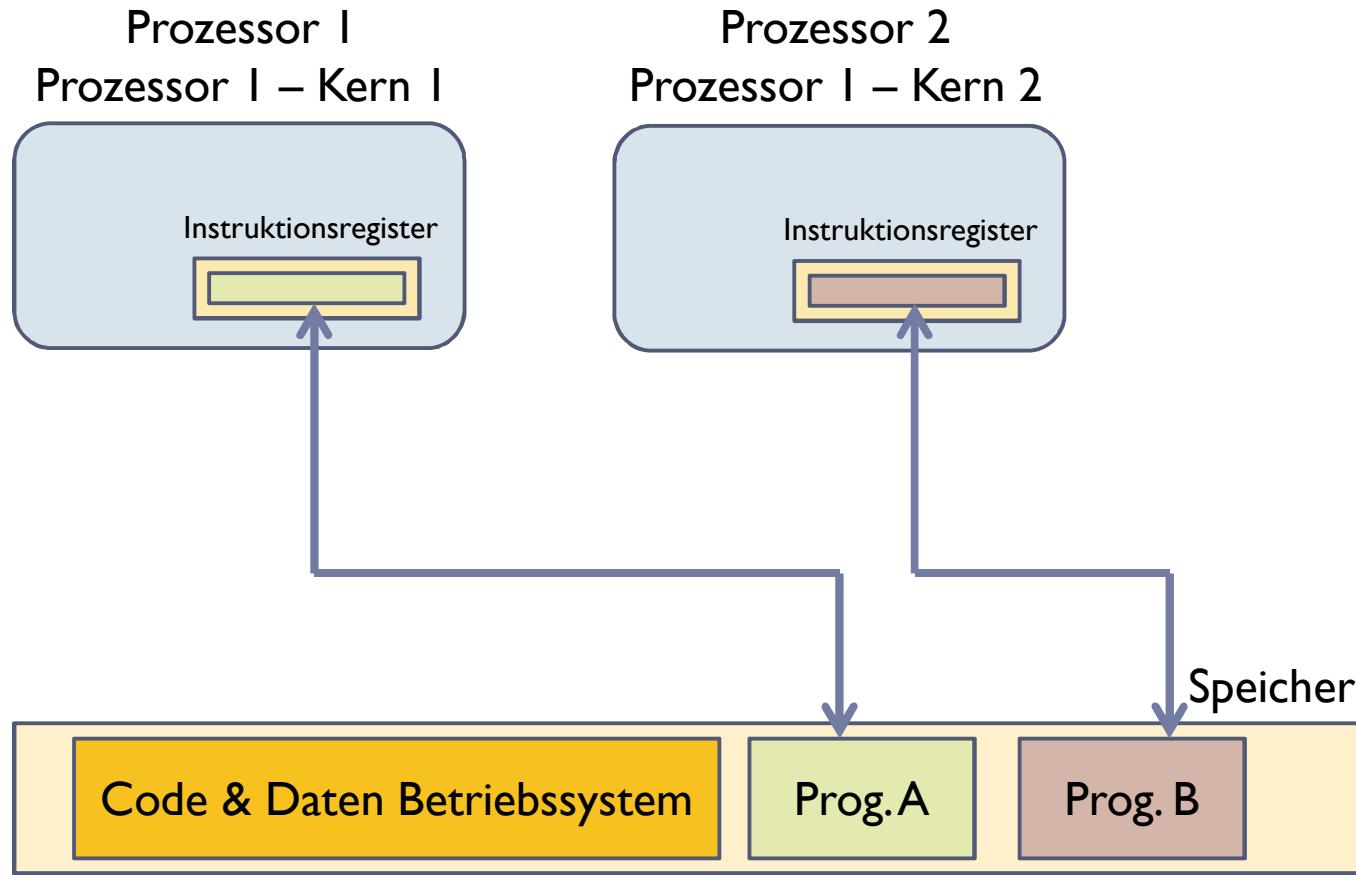
---

## Beim Einprozessorsystem

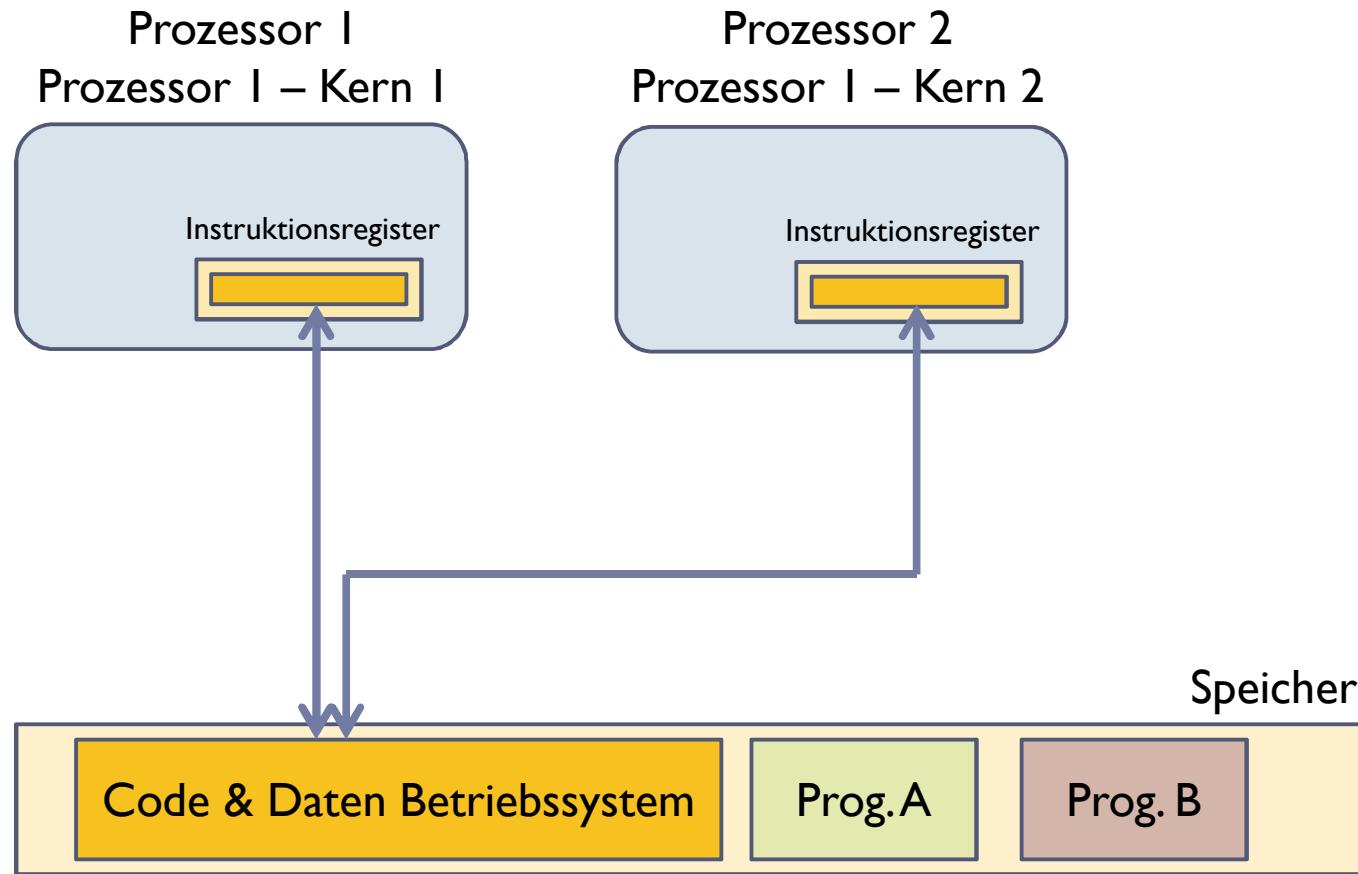
- ▶ Synchronisation problemlos möglich
- ▶ Nur kleinere Probleme
- ▶ Ununterbrechbarkeit des Kerns ist starker Schutz

(Bei Echtzeitbetriebssystemen ist die Ununterbrechbarkeit des Kern nicht mehr gegeben – damit neue Probleme)

# Mehrprozessor/Mehrkern-Systeme



# Mehrprozessor/Mehrkern-Systeme ...



# Mehrprozessor/Mehrkern-Systeme ...

---

## Unkritisch

- ▶ Beide Prozessoren bearbeiten Code verschiedener Programme/Prozesse

## Beginn aller Probleme

- ▶ Ein Prozessor wechselt in den Systemmodus
  - ▶ Wegen Systemaufruf oder Unterbrechung
- ▶ Frage: Was macht anderer Prozessor?
  - ▶ Systemmodus verbieten? Ineffizient!
  - ▶ Systemmodus erlauben? Gefährlich!

# Mehrprozessor/Mehrkern-Systeme ...

---

Was benötige ich alles?

- ▶ Spezial-Hardware zur Unterbrechungssteuerung an jedem einzelnen Prozessor
- ▶ Cache-Kohärenz-Mechanismen
- ▶ **Nebenläufig ausführbares Betriebssystem**
  - ▶ **Threads im Betriebssystem**

# SMP-Hardware

---

## SMP Symmetric Multiprocessing

- ▶ Variante des Betriebssystems, die mehrere Prozessoren/Kerne unterstützen kann

## Intels Multiprocessor Specification MPS 1.4

- ▶ Ist die heutige gültige Referenz
- ▶ Beschreibt Intel-SMP-Systeme
- ▶ Festlegung der BIOS-Eigenschaften
- ▶ Beschreibung des APIC  
Advanced Programmable Interrupt Controller
- ▶ Cache-Kohärenz, MESI-Protokoll
- ▶ In der Praxis damals typisch: 2- und 4-Wege-Systeme

# SMP-Betriebssystem

---

Zunächst eine Begriffsbestimmung

- ▶ In einem SMP-System sind alle Prozessoren gleichberechtigt
- ▶ Sie greifen gemeinsam auf denselben Code und dieselben Daten des Betriebssystemkerns zu und stehen im Wettbewerb um Systemressourcen
- ▶ Jeder Benutzerprozeß kann auf jedem Prozessor zur Ausführung gebracht werden

# SMP-Betriebssystem ...

---

Was bedeutet das?

- ▶ Tatsächlich ist das SMP-Betriebssystem ein paralleles Programm auf einer Maschine mit gemeinsamem Speicher
- ▶ Der Code wird nebenläufig ausgeführt
  - ▶ (vgl. verteiltes Betriebssystem)
- ▶ Die Daten können beliebig manipuliert werden
- ▶ Inkonsistenzen vermeidet man durch Sperren
  - ▶ Sperren von Code-Abschnitten
  - ▶ Sperren von Datenbereichen

# SMP-Betriebssystem ...

---

Alles dreht sich um die Sperren

- ▶ Eine große Sperre (sog. giant lock):  
Nur ein Prozessor kann in das Betriebssystem  
Daten bzgl. der Konsistenz geschützt
  - ▶ Problem gelöst; Effizienz vernichtet
- ▶ Viele kurze Sperren:  
Daten bzgl. der Konsistenz geschützt
  - ▶ Gute Nebenläufigkeit bei geringen Kosten
- ▶ Ganz viele sehr kurze Sperren:  
Daten bzgl. der Konsistenz geschützt
  - ▶ Bessere Nebenläufigkeit bei höheren Kosten

# SMP-Betriebssystem ...

---

Wie mache ich mein Einprozessor-Einkern-Betriebssystem SMP-fähig?

- ▶ Analysiere alle Datenstrukturen und Abläufe im BS-Code
  - ▶ Zunächst Subsysteme wie Speicherverwaltung, Scheduling, Ein-/Ausgabe etc.
- ▶ Schütze kritische Bereich vor gleichzeitigem Zugriff
- ▶ Verfeinere den Schutz (mehr Nebenläufigkeit)
  - ▶ Inkrementelle Parallelisierung

# Synchronisationsmechanismen

---

- ▶ Es müssen jetzt verschiedenste Datenstrukturen geschützt werden, die bei einem einzelnen Prozessor unkritisch waren
- ▶ Einfache Flags reichen nicht aus, da diese gleichzeitig manipuliert werden könnten
- ▶ Unterbrechungssperren müssen global gesetzt werden können
- ▶ Traditioneller Sleep/wakeup-Mechanismus auf Mehrprozessor-Systemen unbrauchbar
- ▶ Wiederaufwecken von Threads kritisch
  - ▶ Thundering-herd-problem

# Synchronisationsmechanismen...

---

Essentiell: Hardware-Unterstützung

- ▶ **Atomares Testen-und-Setzen**
  - ▶ Testet Bit, setzt Wert auf '1', gibt alten Wert zurück
  - ▶ Nach Abschluß ist das Bit '1'
  - ▶ Rückgabewert '0': man hat jetzt Zugriff, Ressource war frei
  - ▶ Rückgabewert '1': Ressource von anderen belegt
- ▶ Anweisung kann nicht einmal durch eine Unterbrechung unterbrochen werden
- ▶ In vielen Systemen sogar atomare Nutzung des Speicherbusses

# Synchronisation mittels Semaphor

---

Standardverfahren:

- ▶ P() dekrementiert Semaphor und blockiert, wenn Wert kleiner 0 wird
- ▶ V() inkrementiert Semaphor und weckt Thread auf, wenn Wert kleiner oder gleich 0 ist

BS-Kern garantiert Atomarität der Aktion

- ▶ Einprozessor-Einkern-System:  
durch Ununterbrechbarkeit der BS-Kerns
- ▶ Mehrprozessor/Mehrkern-System:  
durch tieferliegende unteilbare Aktion

# Synchronisation mittels Semaphor...

---

## Problem

- ▶ Blockieren und Aufwecken erfordert Kontextwechsel im Betriebssystem: langsam
- ▶ Nicht akzeptabel für kurze Blockierungen
- ▶ Weniger aufwendiger Mechanismus benötigt

# Synchronisation mittels Spinlock

---

## Einfachster Sperrenmechanismus: Spinlock

- ▶ engl: *spin lock*, *simple lock*, *simple mutex*
- ▶ Skalare Variable
  - ▶ '0' bedeutet verfügbar
  - ▶ '1' bedeutet belegt
- ▶ Manipulation mittels aktivem Warten (**busy-wait**) und atomarem Testen-und-Setzen

# Synchronisation mittels Spinlock...

```
void spin_lock (spinlock_t *s) {  
    while (test_and_set (s) != 0) /* belegt */  
        ; /* warte auf Freigabe */  
}  
  
void spin_unlock (spinlock_t *s) {  
    *s = 0;  
}
```

Möglicher Nachteil: Busblockierung

# Synchronisation mittels Spinlock...

```
void spin_lock (spinlock_t *s) {  
    while (test_and_set (s) != 0) /*belegt*/  
        while (*s != 0);  
            /* warte auf Freigabe */  
            /* hier nur lesender Zugriff ! */  
    }  
  
void spin_unlock (spinlock_t *s) {  
    *s = 0;  
}
```

# Synchronisation mittels Spinlock...

## Verwendung von Spinlocks

- ▶ Kurzzeitige Sperre kritischer Bereiche im Betriebssystem-Code
- ▶ Niemals für blockierenden Code einsetzen

## Analyse

- ▶ Aktives Warten (normalerweise unerwünscht)
- ▶ Sehr billig, wenn Ressource nicht belegt ist
- ▶ Insgesamt billig bei geringem Belegt-Grad

# SMP-Betriebssystem... Linux

---

- ▶ Bei Linux dauert(e) dieser Vorgang Jahre!
  - ▶ Erstmals SMP-fähig in Kernel 2.2
- ▶ Im Kernel 2.4
  - ▶ Alle Subsysteme durch feingranulare Sperren abgesichert
  - ▶ „Kernel subsystems are fully threaded“
- ▶ Skalierbarkeit bzgl. Anzahl der Prozessoren ist problematisch
  - ▶ Bisherige Grenze: 4 (?)

# Erweiterte Betriebssystemfunktionalität

---

Was brauchen wir sonst noch?

- ▶ Feingranulare Ausführungsobjekte
  - ▶ Kernel-Threads
    - Werden im BS-Code erzeugt und teilen sich mit ihm den Adreßraum
- ▶ Speicherverwaltung für Mehrprozessor-Systeme
  - ▶ Verwaltung mehrerer Speicherbänke
- ▶ Scheduling für Mehrprozessor-Systeme

# Erweiterte Betriebssystemfunktionalität...

---

## Mehrprozessor-Scheduling

- ▶ Prozessor-Affinität

Ein Prozeß oder ein Thread sollte auf dem Prozessorkern fortgesetzt werden, auf dem er zuletzt lief

Grund: Gültiger Inhalt im Cache des Prozessorkerns

- ▶ Gang-Scheduling

Alle Threads eines Prozesses sollten zusammen zugeteilt werden

Grund: Verzögerungen an gemeinsam genutzten Sperren vermeiden

# Betriebssystemaspekte

## Zusammenfassung

- ▶ Synchronisation in Einprozessor-Einkern-Systemen fast ohne Probleme
- ▶ Bei SMP-Systemen läuft das Betriebssystem auf allen Prozessoren
- ▶ Hauptproblem: innere Synchronisation und Schutz der Datenstrukturen
- ▶ Betriebssystem-Code wird durch den Einbau von Sperren parallelisiert
- ▶ Feinere Sperren bedeuten mehr Nebenläufigkeit
- ▶ Hardware-Unterstützung für die Sperren notwendig
- ▶ Semaphor ist zu kostspielig
- ▶ Spinlock ist das wichtigste Synchronisationskonzept
- ▶ Scheduling von Threads ist sehr komplex

# Parallele Programmierung

---

- ▶ Was ist Parallelisierung?
- ▶ Paradigmen der parallelen Programmierung
- ▶ Werkzeuge zur Parallelisierung
- ▶ Algorithmische Aspekte
- ▶ Beispiele

# Parallele Programmierung

## Die zehn wichtigsten Fragen

- ▶ Was ist Parallelisierung eigentlich?
- ▶ Wie ist die allgemeine Vorgehensweise?
- ▶ Welche Paradigmen der Parallelisierung gibt es?
- ▶ Für welche Algorithmen sind diese Paradigmen jeweils gut geeignet?
- ▶ Mit welchen Werkzeugen wird parallelisiert?
- ▶ Welche Probleme gibt es bei der Parallelisierung durch den Compiler?
- ▶ Welche Klassen von Algorithmen können unterschieden werden?
- ▶ Wie parallelisiert man einfache numerische Verfahren?
- ▶ Wie parallelisiert man Anwendungen im Gebiet der Strömungsmechanik?
- ▶ Wie parallelisiert man Suchbaumverfahren?

# Was ist Parallelisierung?

---

## Aufgabe

- ▶ Finde impliziten Parallelismus im Algorithmus und mache ihn explizit
- ▶ Mittel: Verteile Programme und Daten auf die Ressourcen des Systems
- ▶ Wer? Was?
  - ▶ Programmierer und/oder Compiler und/oder Laufzeitsystem

# Was ist Parallelisierung...

---

- ▶ Verteilung erzeugt neue Last (*overhead*)
  - ▶ Minimale Zusatzlast wenn nicht verteilt
- ▶ Verteilung nutzt Ressourcen optimal
  - ▶ Optimale Leistung wenn vollständig verteilt

Zielvorgabe

Nutze alle Ressourcen und minimiere die Zusatzlast

# Anforderungen

---

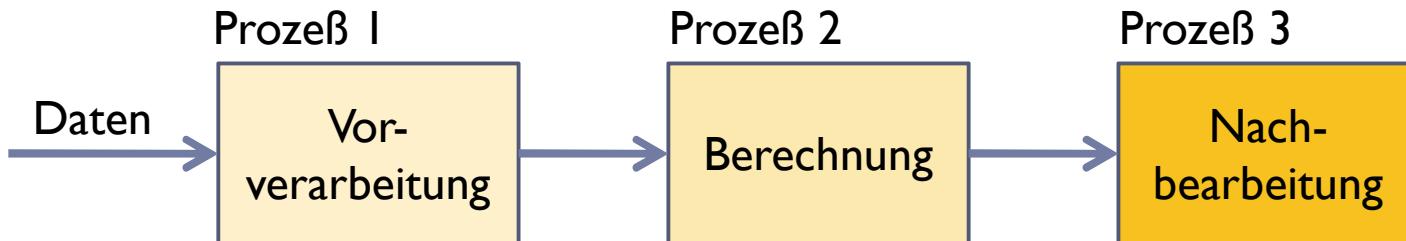
- ▶ Zusätzlich zur sequentiellen Software
  - ▶ Aufteilung des Programms in kleine Teile (*partitioning*)
  - ▶ Hinzufügen von Koordination und Kommunikation
  - ▶ Abbildung der Teile auf die Komponenten des Computers (*mapping*)
- ▶ Probleme
  - ▶ Fehlersuche (neue Fehlerarten)
  - ▶ Leistungsanalyse (Programmbeeinflussung)
  - ▶ Lastausgleich (für optimale Leistung)

# Paradigmen der Parallelisierung

## Code-Aufteilung (auch: Macro-Pipelining)

Verteile den Programmcode über die Knoten

- ▶ Unterschiedlicher Code auf jedem Knoten
- ▶ Daten variieren gemäß dem Fluß der Berechnung
- ▶ Koordinator: erster/letzter Prozeß



# Paradigmen der Parallelisierung...

---

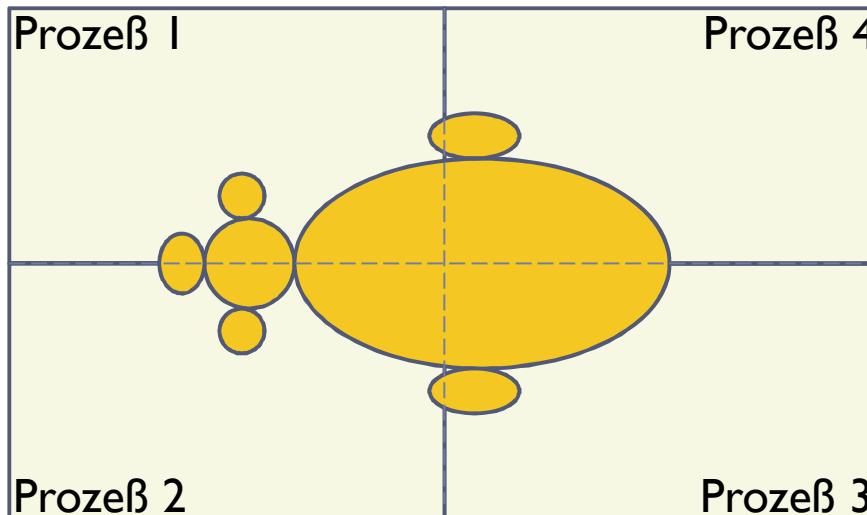
- ▶ **Vorteile** der Code-Aufteilung
  - ▶ Manchmal sehr einfach zu entwerfen
  - ▶ Passende Algorithmen existieren (z.B. FFT)
- ▶ **Nachteil** der Code-Aufteilung
  - ▶ Mehrere Quellcode-Dateien nötig
  - ▶ Schwierig an Zielmaschine anpaßbar
  - ▶ Komplexe Kommunikationsschemata
  - ▶ Schwierige Fehlersuche
  - ▶ Schwieriger Lastausgleich

# Paradigmen der Parallelisierung...

## Datenaufteilung

Verteile die Datenstrukturen auf die Knoten

- ▶ Identischer Code auf jedem Knoten
- ▶ Daten variieren von Knoten zu Knoten
- ▶ Durch ausgewählten Prozeß koordiniert



# Paradigmen der Parallelisierung...

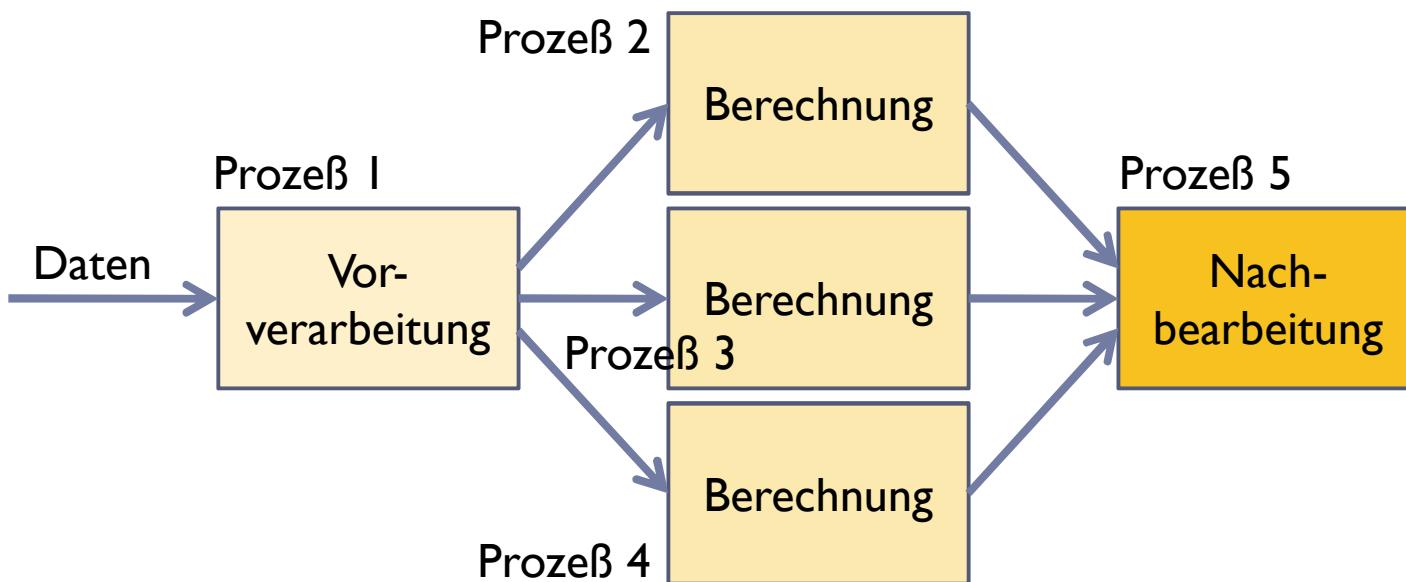
---

- ▶ **Vorteile** der Datenaufteilung
  - ▶ Leicht programmierbar: nur ein Quellcode
  - ▶ Einfach an Zielmaschinen anpaßbar
  - ▶ Sehr oft reguläre Datenaustauschschemata
  - ▶ Einfachere Fehlersuche
- ▶ **Nachteile** der Datenaufteilung
  - ▶ Manchmal dem Algorithmus nicht angemessen
- ▶ Datenaufteilung ist **Quasistandard**
  - ▶ Genannt **SPMD** (single program, multiple data)

# Paradigmen der Parallelisierung...

## Gemischte Code- und Datenaufteilung

- ▶ Dies ist unsere Zielvorstellung
  - ▶ Vereint Vorteile beider Ansätze
  - ▶ Benötigt Mehrprozeßbetrieb auf den Knoten



# Werkzeuge zur Parallelisierung

---

- ▶ Automatisch parallelisierende Compiler
- ▶ Manuell parallelisierende Compiler
- ▶ Parallelisierung durch Laufzeitumgebung
- ▶ Parallele Spracherweiterungen
- ▶ Parallele Erweiterungen zu existierenden sequentiellen Sprachen
- ▶ Parallele Programmierbibliotheken für existierende sequentielle Sprachen

# Automatisch parallelisierende Compiler

---

## Parallelisierung auf Schleifenebene (Fortran)

- ▶ Compiler analysiert Datenabhängigkeiten
- ▶ Erkennt parallel ausführbare Schleifenindizes und verteilt sie
- ▶ Compiler-Pragmas notwendig (Steueranweisungen)
- ▶ Normalerweise schlechte Leistungsausbeute



# Manuell parallelisierende Compiler

---

- ▶ Wichtiger neuer Ansatz: OpenMP
- ▶ Parallelisierung für Systeme mit gemeinsamem Speicher
- ▶ Übersetzung durch spezielle Kommentare kontrolliert
- ▶ Zusätzliche Verwendung von Bibliotheken

# Parallelisierung durch Laufzeitumgebung

---

- ▶ Anwender über gibt dem System eine (höhere) Anzahl von Einzelaufträgen
- ▶ Laufzeitsystem schiebt mittels dynamischem Lastausgleich die Aufträge auf unbelastete Rechnerknoten
- ▶ Nur geeignet für grobgranulare Parallelisierung auf der Ebene der Aufträge

# Parallele Sprachen und Spracherweiterungen

---

Ansatz: High Performance Fortran (HPF)

- ▶ Entworfen 1990+ durch ein Konsortium
- ▶ Schwindende Bedeutung für das Hochleistungsrechnen

Problem: Parallelisierungs-Qualität der Compiler

# Parallele Programmierbibliotheken

---

Sind der Standard im Hochleistungsrechnen

- ▶ Konzept
  - ▶ Starten autonomer Prozesse (*spawning*)
  - ▶ Kooperation mittels Nachrichtenaustausch
- ▶ Beispiele
  - ▶ Parallel Virtual Machine (PVM) [veraltet]
  - ▶ Message Passing Interface (MPI)

# Software/Hardware-Wechselspiel

---

Prinzipiell sind alle Programmierkonzepte auf allen Architekturen einsetzbar

In der Realität nutzen wir aus Effizienzgründen

- ▶ Bibliotheken zum Nachrichtenaustausch für Architekturen mit verteiltem Speicher
- ▶ Threads und automatische/manuelle Parallelisierung für Architekturen mit gemeinsamem Speicher

# Algorithmische Aspekte

---

Zweigeteilte Welt des Programmierers

- ▶ Numerische Algorithmen
  - ▶ Grand Challenge-Algorithmen: Wettervorhersage, Klimabestimmung, Proteindesign usw.
- ▶ Nichtnumerische Algorithmen
  - ▶ Suchverfahren: Theorembeweiser, Spieleprogramme usw.
  - ▶ Datenbank-Anwendungen

# Numerische Algorithmen

---

- ▶ Strömungsmechanik (*Computational fluid dynamics, CFD*), numerische Berechnungen, Optimierungen, Simulationen usw.
  - ▶ Iterative Algorithmen
  - ▶ Beenden aufgrund einer globalen Bedingung
  - ▶ Reguläre Datenstrukturen (Vektoren, Felder)
  - ▶ Reguläre Kommunikationsstrukturen
  - ▶ Statische Prozeßstruktur

# Nichtnumerische Algorithmen

---

- ▶ Datenbankanwendungen, künstliche Intelligenz
  - ▶ Suchbaumverfahren
  - ▶ Irreguläre Kommunikationsstrukturen
  - ▶ Irreguläre Datenstrukturen (dynamisch, *garbage collection*)
  - ▶ Dynamische Prozeß/Thread-Struktur

# Eine erste Zusammenfassung

---

- ▶ Paradigmen der Parallelisierung
  - ▶ Datenaufteilung, Code-Aufteilung
- ▶ Werkzeuge zur parallelen Programmierung
  - ▶ Compiler und Bibliotheken
  - ▶ Das wichtigste: natürliche Intelligenz
- ▶ Zweigeteilte Welt
  - ▶ Numerische / nichtnumerische Algorithmen

# Beispiele von Parallelisierungen

---

Drei Beispiele

- ▶ Numerische Anwendung
  - ▶ Diskussion bzgl. der Aufteilung und der Speicherarchitektur
- ▶ Strömungsmechanik (CFD)
  - ▶ Diskussion bzgl. der zu verteilenden Objekte
- ▶ Suchbaumverfahren
  - ▶ Diskussion bzgl. der Speicherarchitektur

Alle Beispiele manuell parallelisiert

# Beispiel 1: Numerik (1/11)

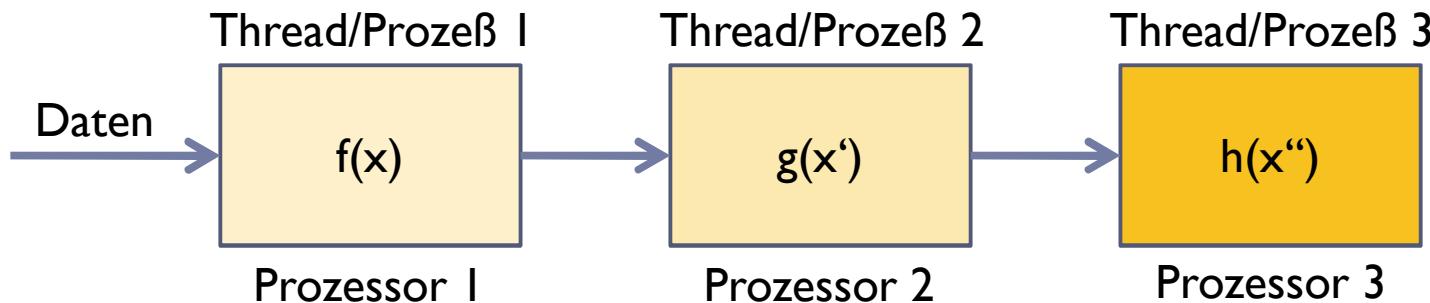
---

- ▶ Drei Funktionen  $f()$ ,  $g()$  und  $h()$
- ▶ Wende Funktionen auf eine Menge von Werten an und berechne Ergebnis  $h(g(f(x)))$
  
- ▶ Wir betrachten vier Fälle:
  - ▶ Code-Aufteilung / Datenaufteilung
  - ▶ Gemeinsamer Speicher / verteilter Speicher

# Beispiel 1: Numerik (2/11)

## Code-Aufteilung

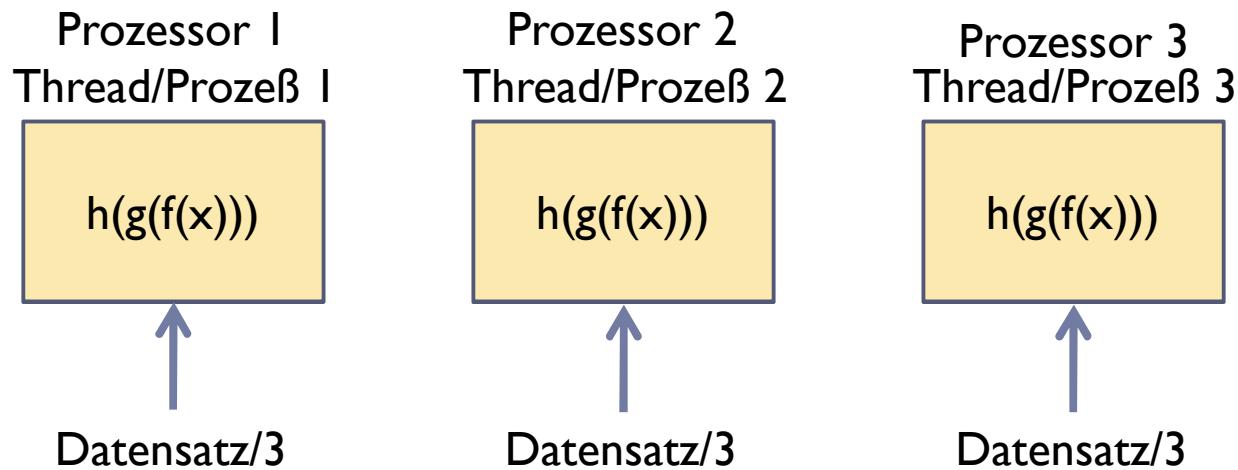
- ▶ Verteile drei Funktionen auf drei Knoten
- ▶ Arbeitet im Makro-Pipelining-Modus
- ▶ Wertemenge ist Eingabedatensatz
- ▶ Nur drei Prozessoren sinnvoll nutzbar



# Beispiel 1: Numerik (3/11)

## Daten-Aufteilung

- ▶ Repliziere die Funktionen auf den Knoten
- ▶ Verteile die Wertemenge auf die Knoten



# Beispiel 1: Numerik (4/11)

---

Code-Aufteilung mit gemeinsamem Speicher

- ▶ Basis-Implementierung
  - ▶ Werte in Vektor gespeichert
  - ▶ Drei Threads, jeder auf eigenem Prozessor
  - ▶ Jeder Thread berechnet eine Funktion f, g, h
  - ▶ Vektoreinträge werden durch Berechnungsergebnisse ersetzt
  - ▶ Variable kennzeichnet den aktiven Thread
- ▶ Nachteil
  - ▶ Dies ist **kein** paralleles Programm (offensichtlich)!

# Beispiel 1: Numerik (5/11)

---

- ▶ Verbesserung der Basis-Implementierung
  - ▶ Zähler für Thread [i], der seine Position anzeigt
  - ▶ Thread [i] darf bis zum Zählerstand von Thread [i-1] -1 vorrücken
- ▶ Vorteil
  - ▶ Gute parallele Implementierung
- ▶ Nachteil
  - ▶ Schlechtes Koordinations/Berechnungs-Verhältnis: zu häufige Inspektion der Zählervariablen

# Beispiel 1: Numerik (6/11)

---

- ▶ Zweite Verbesserung
  - ▶ Erhöhe Granularität der Berechnung
  - ▶ Erhöhe Zähler jeweils um 1000, nicht um 1
- ▶ Vorteil
  - ▶ Gute parallele Implementierung
  - ▶ Besseres Koordinations/Berechnungs-Verhältnis
- ▶ Nachteil
  - ▶ Längere Phasen zum Füllen und Entleeren der Pipeline

# Beispiel 1: Numerik (7/11)

---

Code-Aufteilung mit verteiltem Speicher

- ▶ Basis-Implementierung
  - ▶ Werte in Vektor abgespeichert
  - ▶ Drei Prozesse, jeder läuft auf eigenem Prozessor
  - ▶ Jeder Prozeß berechnet eine der Funktionen f, g, h
  - ▶ Prozeß [i] berechnet alle Zwischenergebnisse und sendet Vektor an Prozeß [i+1]
- ▶ Nachteil
  - ▶ Dies ist wieder **kein** paralleles Programm!

# Beispiel 1: Numerik (8/11)

---

- ▶ Verbesserung der Basis-Implementierung
  - ▶ Prozeß [i] sendet berechnete Werte sofort zu Prozeß [i+1]
- ▶ Vorteil
  - ▶ Gute parallele Implementierung
- ▶ Nachteil
  - ▶ Schlechtes Kommunikations/Berechnungs-Verhältnis: häufiges Senden von Werten

# Beispiel 1: Numerik (9/11)

---

- ▶ Zweite Verbesserung
  - ▶ Erhöhe die Granularität
  - ▶ Sende Werte in Blöcken zu 1000
- ▶ Vorteil
  - ▶ Gute parallele Implementierung
  - ▶ Besseres Kommunikations/Berechnungs-Verhältnis
- ▶ Nachteil
  - ▶ Längere Phase zum Füllen und Leeren der Pipeline

# Beispiel 1: Numerik (10/11)

---

Datenaufteilung mit gemeinsamem Speicher

- ▶ Basis-Implementierung
  - ▶ Werte auf drei Blöcke aufgeteilt
  - ▶ Drei Threads berechnen je  $h(g(f(x)))$  pro Block
  - ▶ Eine Variable pro Block signalisiert das Ende der Berechnung
  - ▶ Ausgewählter Thread organisiert Ausgabe der Ergebnisse
- ▶ Vorteile
  - ▶ Gute parallele Implementierung
  - ▶ Keine Zugriffskonflikte bei den einzelnen Werten
- ▶ Nachteil (gering)
  - ▶ Potentieller Zugriffskonflikt auf Binärkode der Threads

# Beispiel 1: Numerik (11/11)

---

## Datenaufteilung mit verteiltem Speicher

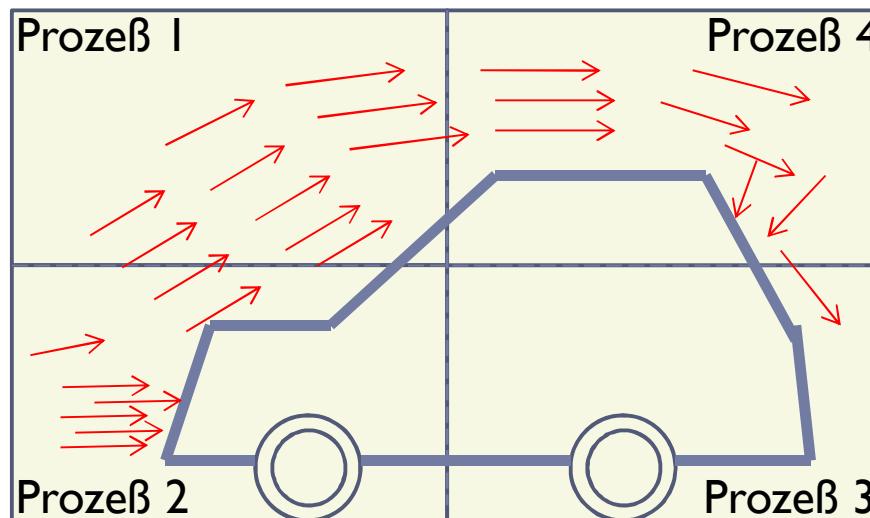
- ▶ Basis-Implementierung
  - ▶ Werte sind auf die Knoten verteilt
  - ▶ Drei Prozesse auf drei Knoten berechnen  $h(g(f(x)))$
  - ▶ Ergebnisse werden einzeln an Prozeß 0 gesendet
- ▶ Vorteile
  - ▶ Gute parallele Implementierung
  - ▶ Gutes Kommunikations/Berechnungs-Verhältnis
- ▶ Nachteil
  - ▶ Programmierung der Datenverteilung

# Beispiel 2: Strömungsmechanik (1/3)

(Das Beispiel illustriert die Probleme in der Praxis)

## Simulation eines Windkanals

- ▶ Iterative Berechnung mit Zeitschritt  $t$ 
  - ▶ Mikroskopischer Ansatz: Berechne Teilchen  
Molekulardynamik vs. Monte Carlo
  - ▶ Makroskopischer Ansatz: Berechne Verteilung von Druck, Temperatur usw.



# Beispiel 2: Strömungsmechanik (2/3)

---

## Mikroskopischer Ansatz:

### Erste Variante: Verteile Teilchen

- ▶ Jeder Prozeß (Prozessor) verwaltet und berechnet einen Teil der Teilchen
- ▶ Nachteil
  - ▶ Physikalisch benachbarte Teilchen nur schwer bestimmbar
- ▶ Vorteil
  - ▶ Gleichverteilung der Anzahl der Teilchen ergibt meist guten Lastausgleich

## Beispiel 2: Strömungsmechanik (3/3)

---

Mikroskopischer Ansatz:

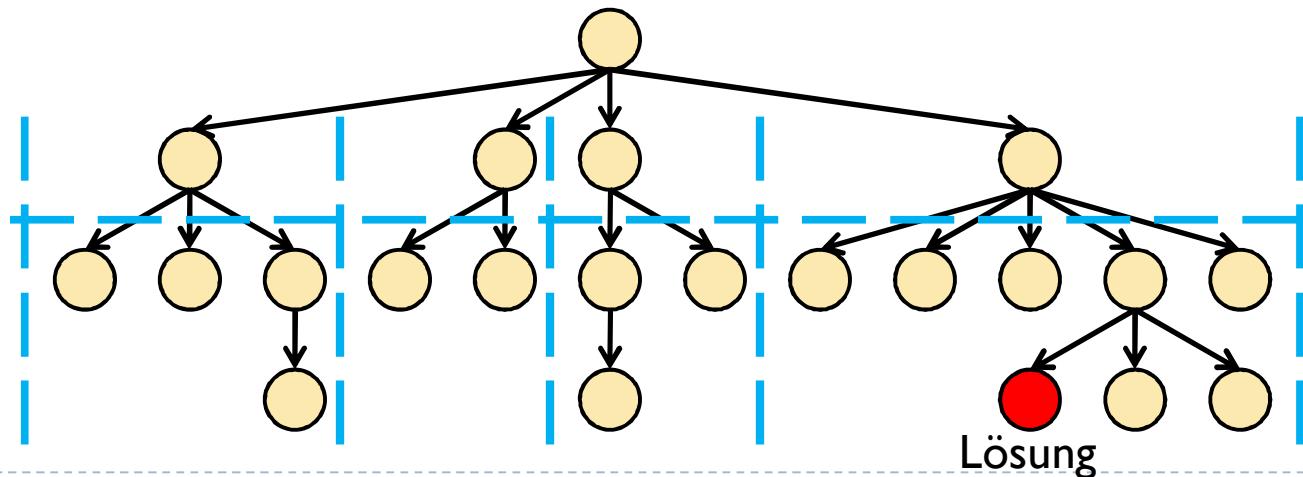
Zweite Variante: Verteile Volumenanteile

- ▶ Jeder Prozessor berechnet einen Teil des Volumens
- ▶ Nachteil
  - ▶ Wechselnde Anzahl von Teilchen führt meist zu schlechter Lastbalance
- ▶ Vorteil
  - ▶ Benachbarte Teilchen leicht bestimmbar

# Beispiel 3: Suchbaumverfahren (1/3)

(Das Beispiel illustriert eine wichtige Problemklasse)

- ▶ An jeder Position mehrere Fortsetzungen möglich
- ▶ Probleme
  - ▶ Ebene der Lösung(en) unbekannt
  - ▶ Lastausgleich zwischen den Prozessoren
  - ▶ Feststellung des Programmendes



## Beispiel 3: Suchbaumverfahren (2/3)

---

Baumsuche mit **gemeinsamem** Speicher

- ▶ Thread berechnet Baum bis zur Ebene  $j$  und gibt Beschreibung in eine Warteschlange
- ▶ Verfügbare Threads entnehmen Beschreibung aus der Warteschlange und berechnen den verbleibenden Baum
- ▶ Gute parallele Implementierung
  - ▶ Lastausgleich ist kein Problem
  - ▶ Feststellung des Berechnungsendes: setze Ende-Bit; andere Prozesse prüfen regelmäßig

## Beispiel 3: Suchbaumverfahren (3/3)

---

### Baumsuche mit **verteilt**em Speicher

- ▶ Prozeß i berechnet bis zum Level j und gibt Beschreibung in eine lokale Warteschlange
- ▶ Untätige Prozesse kontaktieren Prozeß i, bekommen Elemente der Warteschlange als Nachricht und berechnen den restlichen Baum

### Gute parallele Implementierung

- ▶ Lastausgleich kein Problem, aber mehr Aufwand
- ▶ Feststellen des Berechnungsendes: sende Enden-Nachricht an alle anderen Prozesse; diese prüfen regelmäßig

# Zusammenfassung zu den Beispielen

---

- ▶ Es gibt verschiedenste Varianten, Code zu parallelisieren
- ▶ **Die konkrete Variante beeinflußt die maximal erzielbare Leistung des Verfahrens**
- ▶ ***Normalerweise kann die Effizienz der parallelen Programmvariante nicht am sequentiellen Programm bestimmt werden***
- ▶ **Datenaufteilung ist in den meisten Fällen einfacher zu programmieren**
- ▶ Suchbaumverfahren sind oft trivial zu parallelisieren

# Parallele Programmierung

## Zusammenfassung

- ▶ Parallelisierung von Programmen ist eine komplexe Tätigkeit
- ▶ Man nutzt Code-Aufteilung und Datenaufteilung
- ▶ Datenaufteilung meist einfach und effizient
- ▶ Werkzeuge: Programmierbibliotheken, Erfahrung
- ▶ Deutliche Unterschiede zwischen numerischen und nichtnumerischen Algorithmen
- ▶ Effizienz der Parallelisierung selten vorhersagbar

# Programmiermodell Nachrichtenaustausch

---

- ▶ Problemstellung
- ▶ Das Message Passing Interface (MPI)
- ▶ Ziele und Spezifikationsumfang
- ▶ Punkt-zu-Punkt-Kommunikation
- ▶ Abgeleitete Datentypen
- ▶ Kollektive Kommunikationen
- ▶ Gruppen, Kontexte, Prozeßtopologien
- ▶ Bewertungen

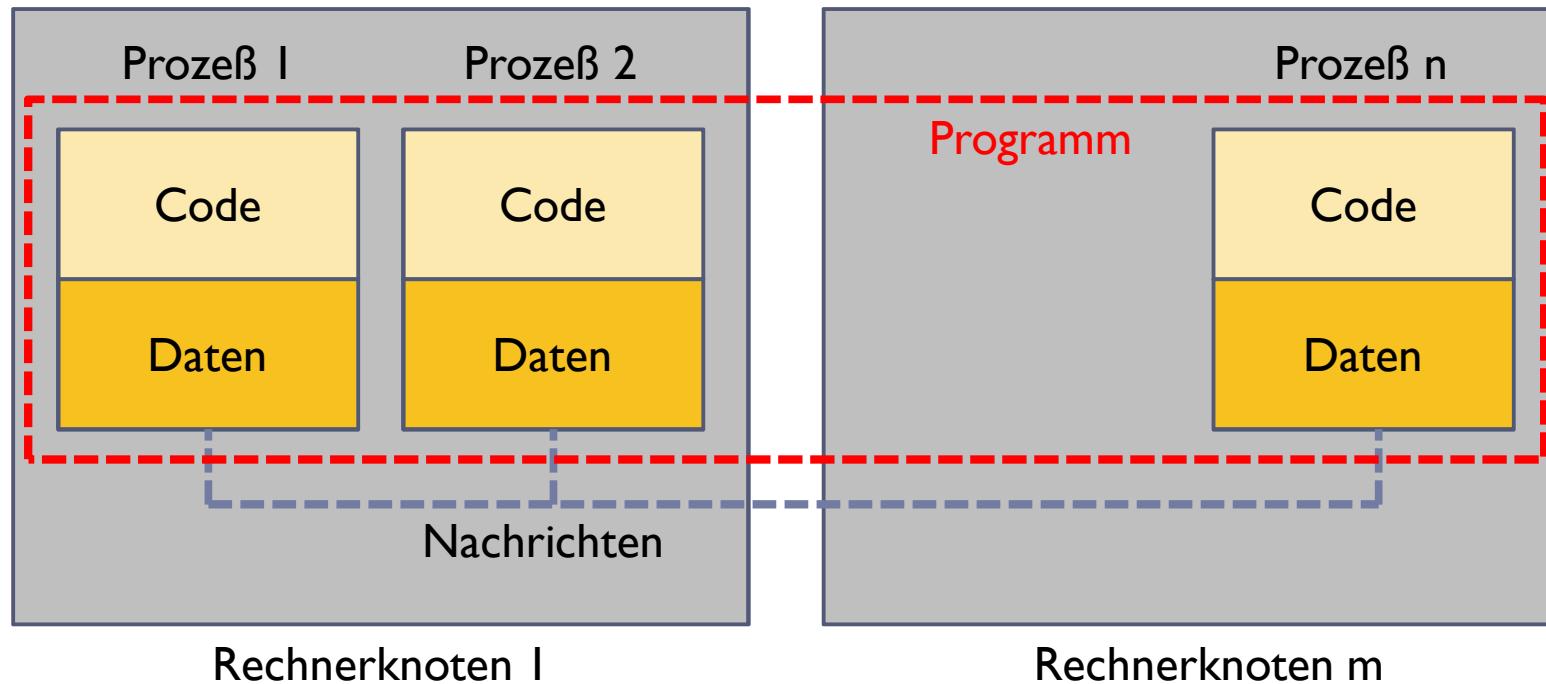
# Programmiermodell Nachrichtenaustausch

## Die zehn wichtigsten Fragen

- ▶ Welche Kommunikationsschemata gibt es?
- ▶ Was sind die Ziele der MPI-Definition?
- ▶ Was enthält die MPI-Definition?
- ▶ Welche Variationen von Blockierungen gibt es bei den Funktionsaufrufen?
- ▶ Wie ist die Punkt-zu-Punkt-Kommunikation definiert?
- ▶ Wie funktioniert nichtblockierende Kommunikation?
- ▶ Was sind abgeleitete Datentypen?
- ▶ Was sind kollektive Kommunikationen?
- ▶ Was versteht man unter Prozeßtopologien?
- ▶ Wie funktioniert das Profiling-Interface?

# Problemstellungen

Die Codeteile des Programms in den Prozessen können identisch oder verschieden sein



# Problemstellungen...

---

- ▶ Kompilieren für unterschiedliche Architekturen
- ▶ Laden des Codes auf unterschiedliche Knoten
- ▶ Start der Prozesse auf den Knoten
- ▶ Wechselseitiges Bekanntmachen der Prozesse
- ▶ Informationsaustausch zwischen Prozessen
- ▶ Optimierung der Kommunikationseffizienz
- ▶ Kommunikationsrelationen der Prozesse zueinander
- ▶ Überwachungsmöglichkeit der Abläufe

# Laden und Starten des Codes

---

- ▶ Prinzipieller Aufruf

```
spawn(<binary_name>,<node_list>,....);
```

- ▶ Ist ähnlich wie bei einer Thread-Erzeugung

- ▶ Wenn nur ein Programmcode existiert

```
if (myid() == 0)
then /* I'm the first */
    spawn(...); /* if others do not exist */
    send(init_data);
else /* I was spawned */
    receive(init_data);
fi
```

Nicht notwendigerweise nur ein Prozeß pro Prozessor

# Informationsaustausch

---

- ▶ Senden von Nachrichten

```
send(<to_proc_id>,<data>);
```

```
broadcast(<data>);
```

- ▶ Empfangen von Nachrichten

```
receive(<from_proc_id>,<data>);
```

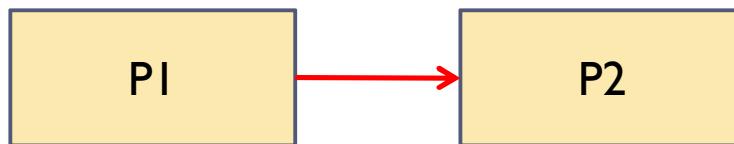
```
testreceive(<from_proc_id>);
```

Charakteristisch: eigenhändiges Einfügen der Kommunikationsanweisungen in den Code

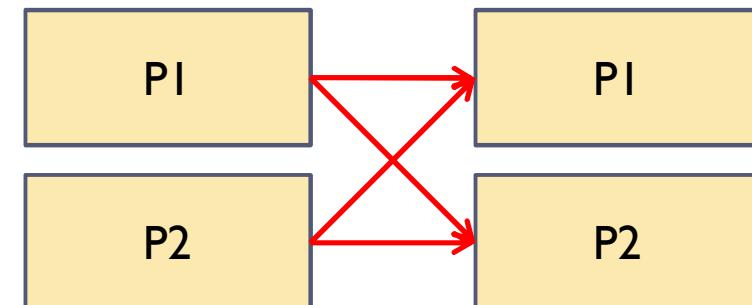
Hoher Aufwand aber auch hohe Leistungsausbeute

# Kommunikationsschemata

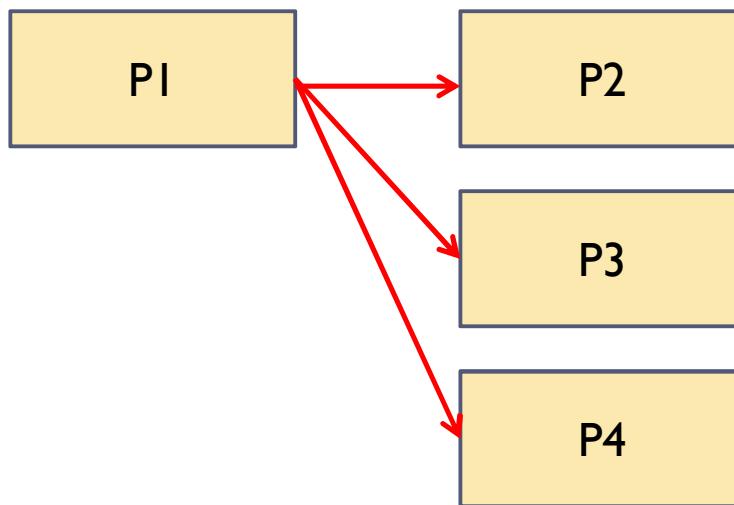
I:I-Kommunikation



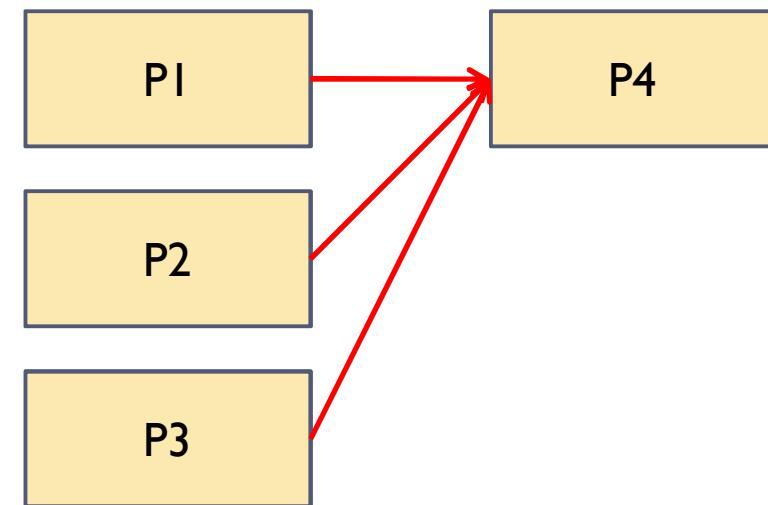
n:n-Kommunikation



I:n-Kommunikation



m:I-Kommunikation



# Optimierung der Kommunikationseffizienz



Möglichst Senden und Empfangen nebenläufig abwickeln

Kombination aus Hardware und Software erforderlich



# Existierende Ansätze Anfang der 90er

---

- ▶ **P4, Parmacs, Chameleon, NX, ...**  
Historie der Bibliotheken zum Nachrichtenaustausch
- ▶ **Parallel Virtual Machine (PVM)**  
Implementierung einer Bibliothek für nahezu alle Architekturen  
Lange Zeit de facto-Standard bei Clustern
- ▶ **Message Passing Interface (MPI)**  
Spezifikation einer Schnittstelle zum Nachrichtenaustausch  
De facto-Standard auf allen Hochleistungsrechnern und auf Cluster-Architekturen

# Message Passing Interface (MPI)

---

- ▶ Vorangetrieben vom MPI-Forum  
(Firmen, Universitäten, ...)
- ▶ Beginn 1992
- ▶ MPI Standard 1995 (nur Kommunikation)
- ▶ MPI-2 Standard 1997 (der nötige Rest)
- ▶ Vorteile eines Standards: Portabilität, Einfachheit  
Vorher etwa ein Dutzend konkurrierende Ansätze
  - ▶ Alle funktional fast identisch aber syntaktisch unterschiedlich
- ▶ Probleme: Standardkonformität der Implementierungen

# Ziele von MPI

---

- ▶ Entwurf einer Programmierschnittstelle (API)
- ▶ Unterstützung effizienter Kommunikationsmethoden
- ▶ Unterstützung heterogener Umgebungen
- ▶ Sprachanbindungen für Fortran77 und C/C++  
(jetzt auch für Java und Skript-Sprachen)
- ▶ Konstrukte nahe an bereits Existierendem
- ▶ Semantik der Schnittstelle soll sprachunabhängig sein
- ▶ Soll eine thread-sichere Implementierung gestatten
  - ▶ Wiedereintrittsfähige Routinen der Bibliotheksimplementierung!



# Was MPI enthält

---

- ▶ Punkt-zu-Punkt-Kommunikation
- ▶ Kollektive Operationen
- ▶ Prozeßgruppen
- ▶ Kommunikationskontakte
- ▶ Prozeßtopologien
- ▶ Abfragefunktionen zur Programmumgebung
- ▶ Profiling-Schnittstelle

# Was MPI (zunächst) nicht enthält

---

- ▶ Explizite Operationen für gemeinsamen Speicher
- ▶ Zusätzliche Unterstützung durch das Betriebssystem für z.B. unterbrechungsgesteuerte Kommunikation
- ▶ Explizite Unterstützung zur Prozeßverwaltung
- ▶ Parallele Ein-/Ausgabe

MPI zunächst nur Nachrichtenaustausch

MPI-2 geht die obigen Punkte an

# MPI-Spezifikationsmethode

---

- ▶ Aufrufe sprachunabhängig definiert
- ▶ Argumente mit IN, OUT oder INOUT annotiert

Z.B. **MPI\_WAIT(request,status)**

    INOUT **request**

    OUT   **status**

C:     **int MPI\_Wait(MPI\_Request \*request,**  
                 **MPI\_Status \*status)**

F77: **MPI\_WAIT(REQUEST,STATUS,IERROR)**  
                 **INTEGER REQUEST,**  
                 **STATUS(MPI\_STATUS\_SIZE),**  
                 **IERROR**

# MPI Definitionen

---

MPI sehr sorgsam mit Problemen der Sprache  
Wichtige Begriffe werden eindeutig definiert

- ▶ *Nonblocking*: Der Aufruf kehrt zurück, bevor die Operation abgeschlossen ist und bevor die Ressourcen wiederverwendet werden dürfen
- ▶ *Locally blocking*: Bei Rückkehr dürfen die lokalen Ressourcen wiederverwendet werden
  - ▶ Hängt nur vom lokalen Prozeß ab
- ▶ *Globally blocking*: Bei Rückkehr ist die Kommunikationsoperation abgeschlossen
  - ▶ Hängt von anderen Prozessen ab
- ▶ *Collective*: Alle Prozesse einer Gruppe müssen den Aufruf ausführen

# Punkt-zu-Punkt-Kommunikation

---

## Senden

**MPI\_SEND(buf, count, datatype, dest, tag, comm)**

|                    |                               |
|--------------------|-------------------------------|
| <b>IN buf</b>      | Adresse des Sendepuffers      |
| <b>IN count</b>    | Anzahl der Elemente im Puffer |
| <b>IN datatype</b> | Datentyp des Elements         |
| <b>IN dest</b>     | Rangangabe des Ziels          |
| <b>IN tag</b>      | Nachrichtenkennung            |
| <b>IN comm</b>     | Kommunikator(Gruppe, Kontext) |

Datentypen: int, long int, float, char, ...

Nachrichten bestehen aus Inhalt und Umschlag

# Punkt-zu-Punkt-Kommunikation...

## Empfangen

```
MPI_RECV(buf, count, datatype, source, tag,  
          comm, status)
```

|                    |                                     |
|--------------------|-------------------------------------|
| <b>OUT buf</b>     | <b>Adresse des Empfangspuffers</b>  |
| <b>IN count</b>    | <b>Anz. der Elemente im Puffer</b>  |
| <b>IN datatype</b> | <b>Datentyp des Elements</b>        |
| <b>IN source</b>   | <b>Rangangabe der Quelle</b>        |
| <b>IN tag</b>      | <b>Nachrichtenkennung</b>           |
| <b>IN comm</b>     | <b>Kommunika. (Gruppe, Kontext)</b> |
| <b>OUT status</b>  | <b>Ergebnis des Empfangens</b>      |

# Punkt-zu-Punkt-Kommunikation...

---

Empfangen...

- ▶ Gesteuert durch den Umschlag  
**`MPI_ANY_SOURCE`, `MPI_ANY_TAG`** (Wildcard)
  
- ▶ Abfrage mittels  
**`MPI_GET_SOURCE()`, `MPI_GET_TAG()`**

# Punkt-zu-Punkt-Kommunikation...

---

- ▶ Semantik der Kommunikation
  - ▶ Nachrichtenreihenfolge bleibt erhalten
- ▶ Datenumwandlung
  - ▶ In heterogenen Netzen automatische Umwandlung
- ▶ Modi
  - ▶ Normal: lokal blockierend
  - ▶ Ready Communication: Senden darf erst aufgerufen werden, wenn Empfangen schon bereit ist (effizientere Realisierung der Datenübertragung möglich)
  - ▶ Synchronous Communication: global blockierend; schließt ab, wenn der Empfang begonnen hat

# Punkt-zu-Punkt-Kommunikation...

---

- ▶ Nichtblockierende Kommunikation
  - ▶ Verbesserte Effizienz durch Überlappung von Berechnung und Kommunikation
- ▶ Wichtige Unterscheidung
  - ▶ Blockierend / nichtblockierend  
(wann kehrt der Aufruf zurück)
  - ▶ Synchron / asynchron  
(wann ist der Auftrag ausgeführt)
  - ▶ Prinzip: der Aufruf wird mit einer Referenz versehen  
Durch Abfragen bzgl. der Referenz kann der Status der Ausführung ermittelt werden

# Punkt-zu-Punkt-Kommunikation...

---

## Nichtblockierend

|                                      |                   |
|--------------------------------------|-------------------|
| <b>MPI_ISEND(...)</b>                | immediate send    |
| <b>MPI_IRecv(...)</b>                | immediate receive |
| <b>MPI_TEST(request,flag,status)</b> | nichtblockierend  |
| <b>MPI_WAIT(request)</b>             | blockierend       |
| <b>MPI_CANCEL(request)</b>           |                   |

# MPI „Hello World“

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

int main (int argc, char *argv[])
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("Hello World from process %d of %d\n",
           rank, size );
    MPI_Finalize();
    return 0;
}
```

# Abgeleitete Datentypen

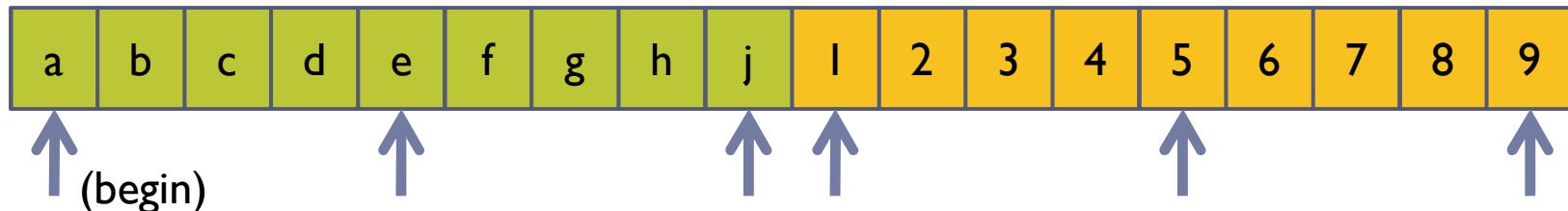
---

- ▶ Verwendungszweck
  - ▶ Nachrichten mit gemischten Datentypen
  - ▶ Nachrichten mit nichtzusammenhängenden Bereichen
- ▶ Ein-/Auspicken der Nachrichten erfordert Rechenaufwand
- ▶ Effizienz hängt von der Hardware ab  
(z.B. Direct Memory Access, DMA)

# Abgeleitete Datentypen...

Beispiel: Zwei Matrizen mit komplexen Zahlen

Aufgabe: Versende die beiden Diagonalen



```
MPI_TYPE_VECTOR(3/*blocks */, 1/*element/block*/,
    4/*blockstride*/, MPI_COMPLEX, diag)
MPI_TYPE_CREATE_HVECTOR(2/*blocks*/, 1/*elm/blck*/,
    9*sizeof(MPI_COMPLEX),diag,doublediag)
MPI_TYPE_COMMIT(doublediag)
MPI_SEND(begin,1,doublediag,me,other,comm)
```

# Kollektive Kommunikationen

---

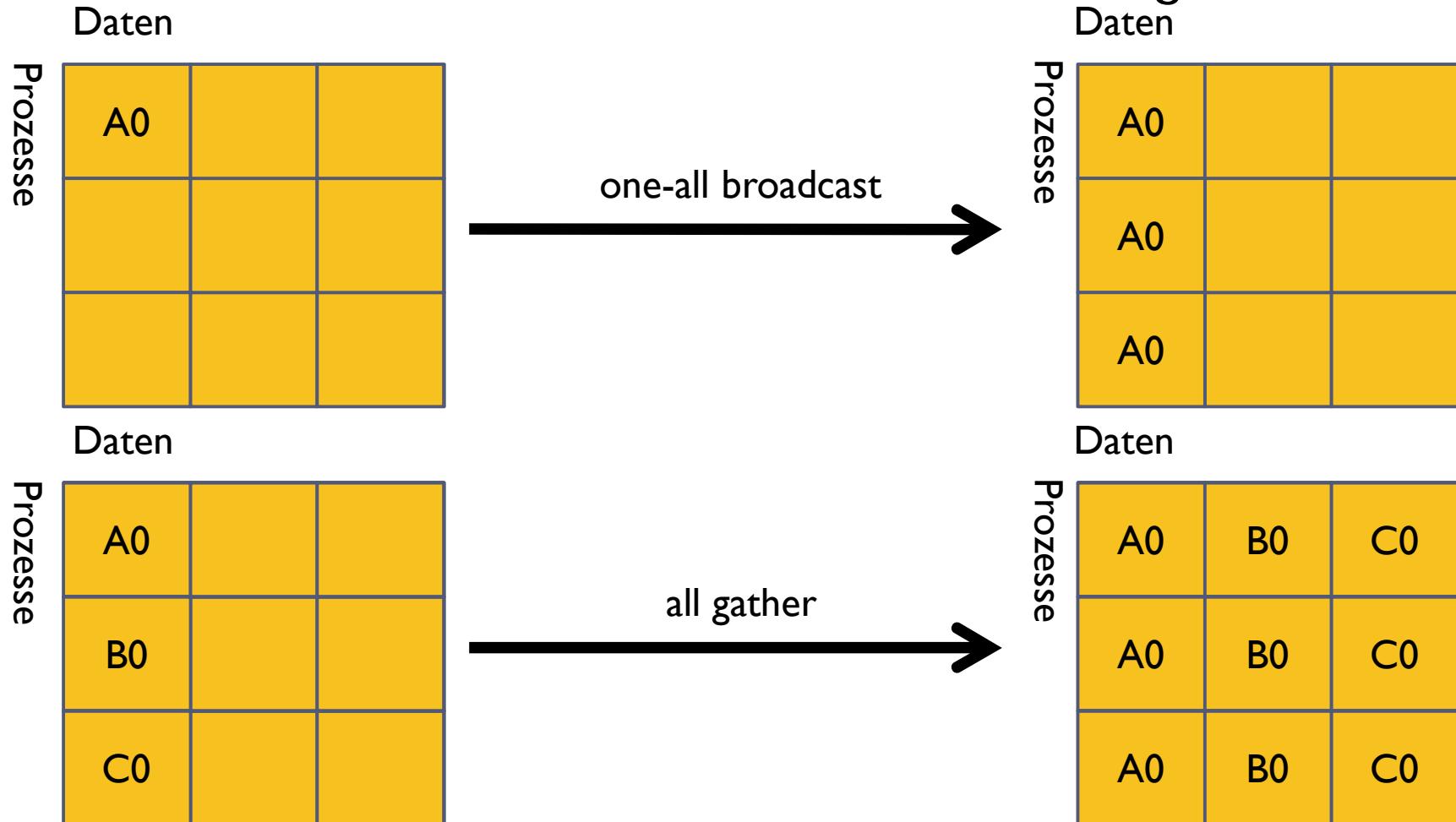
Kollektive Kommunikationen werden immer von allen Mitgliedern einer Gruppe durchgeführt

- ▶ Broadcast von einem an alle
- ▶ Barrierensynchronisation
- ▶ Daten einsammeln / verteilen
- ▶ Globale Berechnung von Funktionen

Möglicherweise durch spezielle Hardware unterstützt  
Spielraum für Implementierungsoptimierungen

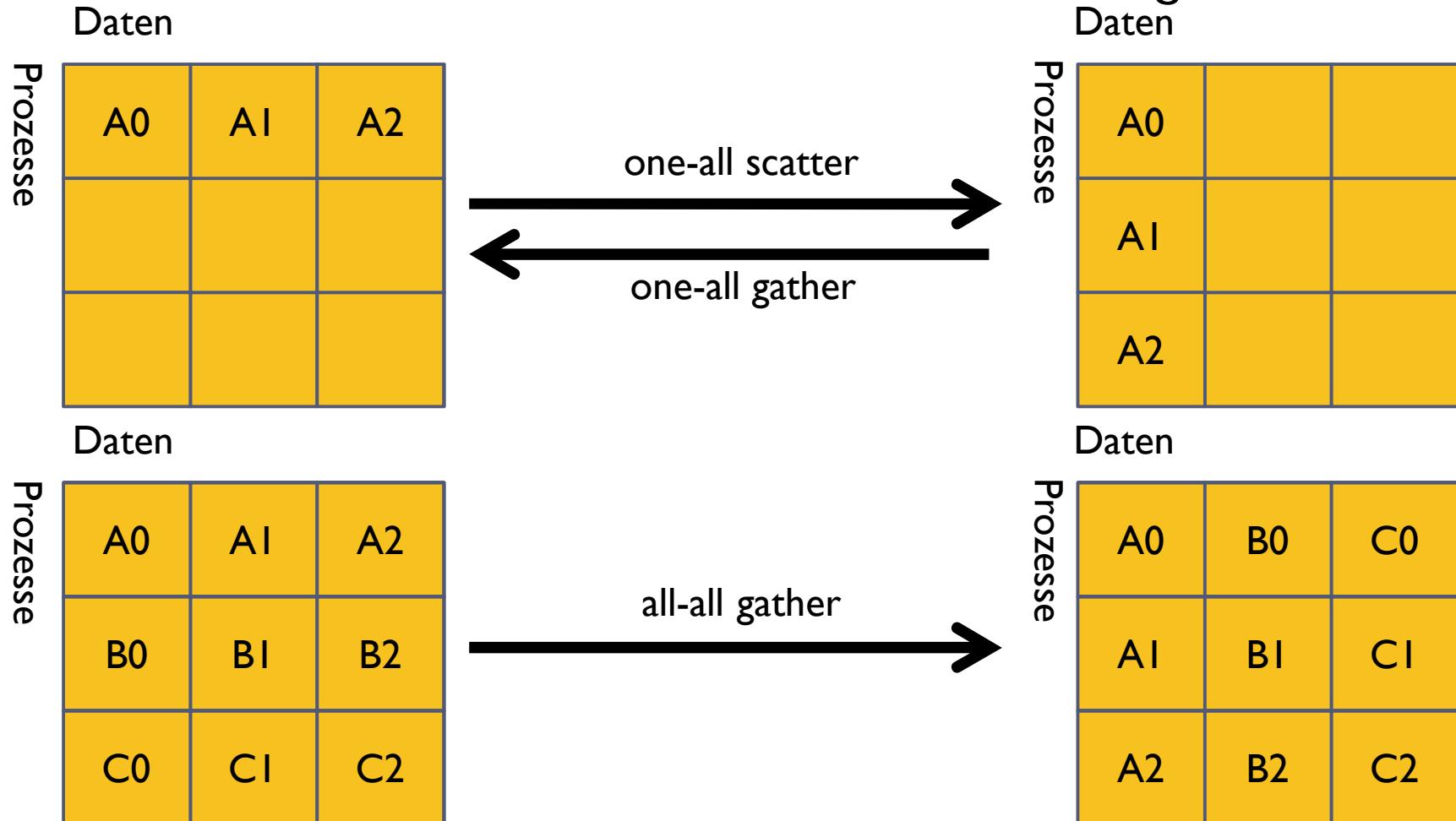
# Kollektive Kommunikationen...

## Kollektive Funktionen zur Datenverschiebung



# Kollektive Kommunikationen...

## Kollektive Funktionen zur Datenverschiebung



# Kollektive Berechnungen

---

- ▶ Häufig müssen alle Prozesse dieselbe Funktion auf Daten anwenden, z.B. die Summenoperation
- ▶ Funktion **`MPI_REDUCE( . . . , op, . . . )`**
  - ▶ Jeder Prozeß trägt seinen Datenanteil bei
  - ▶ Am Ende hat jeder Prozeß das Endergebnis  
**`max, min, sum, product, AND, OR, XOR`**
- ▶ Auswertereihenfolge beliebig
  - ▶ Evtl. Nichtdeterministisches Ergebnis
- ▶ In Parallelrechnern teilweise durch Hardware unterstützt
- ▶ Eigene Funktionen möglich (kritisch)

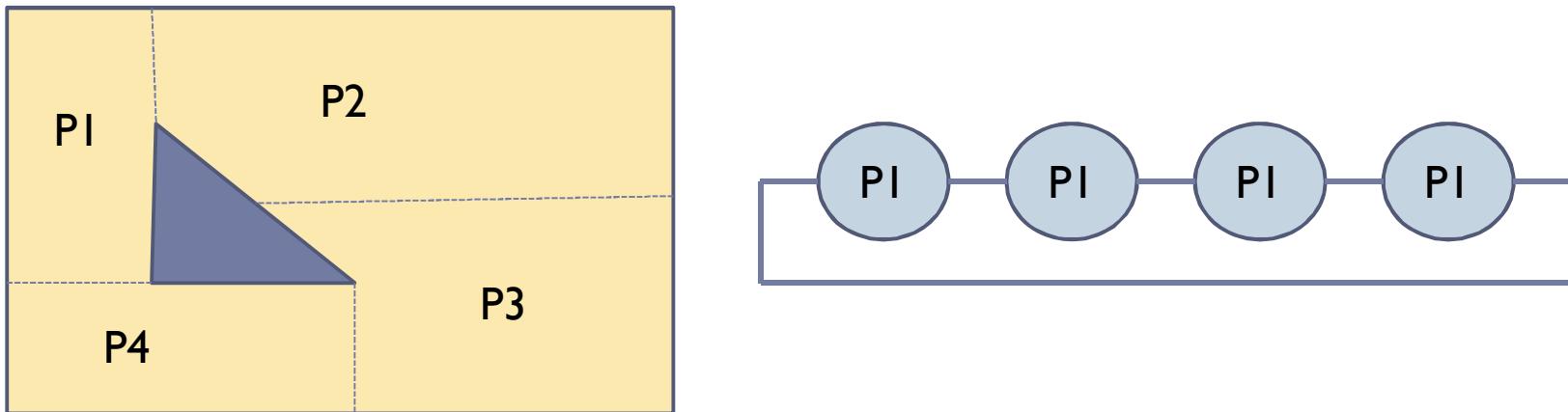
# Gruppen, Kontexte, Kommunikatoren

---

- ▶ Neues Konzept, das es vorher nirgends gab
- ▶ Problem:
  - ▶ Drittanbieter entwickeln Bibliotheken mit Nachrichtenaustausch
  - ▶ Kennungen dieser Nachrichten und Rangangaben dürfen nicht mit dem Anwenderprogramm in Konflikt geraten
- ▶ Lösung
  - ▶ Gruppen fassen zusammengehörige Prozesse zusammen
  - ▶ Kontexte unterscheiden logische Teile des Programms
  - ▶ Kommunikator: faßt Gruppe und Kontext zusammen
  - ▶ Default-Kommunikator: **MPI\_COMM\_WORLD**

# Prozeß-Topologien

Problem: Rangangaben sagen nichts über Beziehungen aus



- ▶ Benutzersicht: Nur bestimmte Kommunikationsmuster treten auf  
Zugriff auf Nachbarn über symbolische Namen
- ▶ MPI unterstützt die Topologieverwaltung

# Profiling-Interface

---

- ▶ Möglichkeit zum Anschluß von Werkzeugen in MPI integriert
- ▶ Konzept
  - ▶ Unterstütze die Aktivierung von Überwachungen beim Aufruf von MPI-Funktionen
- ▶ Realisierung
  - ▶ Jede Funktion `MPI_xyz` muß auch über den Namen `PMPI_xyz` aufrufbar sein (*profiling*)

# Profiling-Interface...

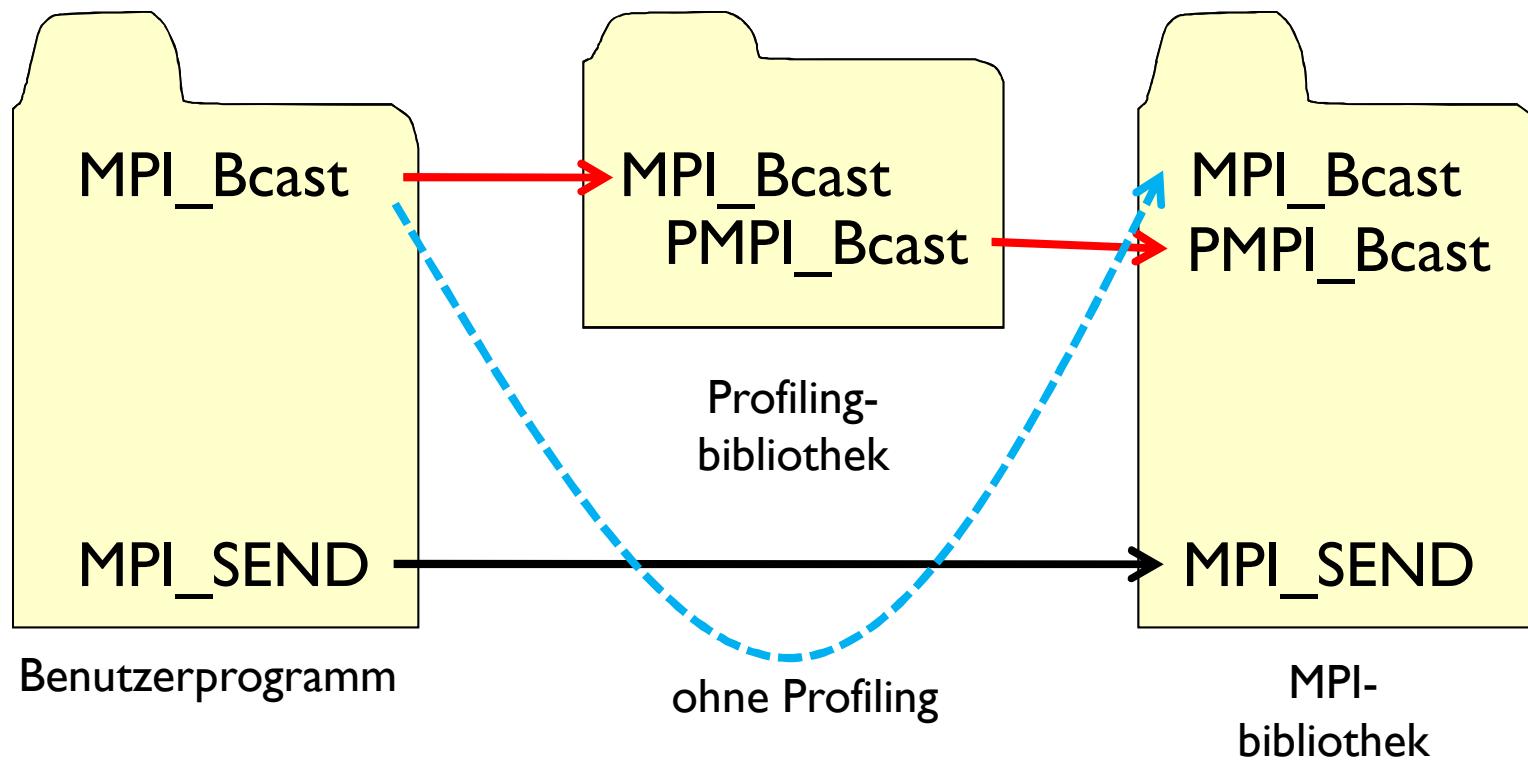
Beispiel: überwache Broadcast-Funktion

```
int MPI_Bcast(...)  
{ int result;  
    write_log_entry(...);  
    start_timer();  
    result=PMPI_Bcast(...);  
    stop_timer(); write_log_entry(...);  
    return result;  
}
```

Dies definiert eine Profiling-Version der Funktion

# Profiling-Interface...

Der Trick: Reihenfolge beim Linken  
zuerst: Profiling-Bibliothek, dann: libmpi



# Bewertung MPI

---

- ▶ Spezifikation ausschließlich für Nachrichtenaustausch
- ▶ Sehr viele Funktionen
- ▶ Prozeßverwaltung fehlt
- ▶ Kein dynamisches Prozeßkonzept
  - Keine Programme mit dynamisch variierender Prozeßanzahl

# Ausblick auf MPI-2

---

- ▶ MPI-2 ist eine Erweiterung zu MPI, nicht eine neue Version
- ▶ Umfaßt Klarstellungen zu MPI und Erweiterungen
- ▶ Wichtige Erweiterung: Prozeßverwaltung  
(Vorher machte jeder Hersteller was er wollte)
- ▶ Wichtige Erweiterung: Ein-/Ausgabe  
(Idee: äquivalent zu Senden und Empfangen von Nachrichten)
- ▶ Nachteil: sehr viele neue Funktionen

# Vergleich der Ansätze

|                             | Pthreads | OpenMP     | MPI        |
|-----------------------------|----------|------------|------------|
| Skalierbarkeit              | Begrenzt | Begrenzt   | Ja         |
| Fortran / C und C++         | Ja? / Ja | Ja / Ja    | Ja / Ja    |
| Hohe Abstraktion            | Nein     | Ja         | Nein       |
| Leistungsorientierung       | Nein     | Ja         | Ja         |
| Portierbarkeit              | Ja       | Ja         | Ja         |
| Herstellerunterstützung     | Unix/SMP | Verbreitet | Verbreitet |
| Inkrement. Parallelisierung | Nein     | Ja         | Nein       |

# Programmiermodell Nachrichtenaustausch

## Zusammenfassung

---

- ▶ Relevante Probleme beim Nachrichtenaustausch:  
Kommunikationsschemata, Effizienz,  
Prozeßverwaltung
- ▶ MPI ist eine Spezifikation eines API zum  
Nachrichtenaustausch
- ▶ Punkt-zu-Punkt-Kommunikation mit vielen Varianten  
möglich:  
synchron/asynchron, blockierend/nichtblockierend
- ▶ Abgeleitete Datentypen vereinfachen die  
Kommunikation
- ▶ Gruppen und Kontexte dienen zur wechselseitigen  
Abgrenzung von Programmteilen
- ▶ MPI-2 erweitert MPI um wesentliche Aspekte

# Parallele Eingabe/Ausgabe

---

- ▶ Einleitung, Konzepte, Definitionen
- ▶ Einfache E/A
- ▶ Nichtzusammenhängende Zugriffe
- ▶ Kollektive Aufrufe
- ▶ Nichtblockierende E/A
- ▶ Gemeinsame Dateizeiger
- ▶ Dateiformate
- ▶ Leistungsaspekte
- ▶ Die Implementierung

# Parallele Eingabe/Ausgabe

## Die zehn wichtigsten Fragen

- ▶ Was ist MPI-2 I/O und wozu braucht man es?
- ▶ Welche Konzepte gibt es dabei?
- ▶ Wie ist eine Datei strukturiert?
- ▶ Wie geht die einfachste E/A?
- ▶ Wie funktionieren nichtzusammenhängende Zugriffe?
- ▶ Wie funktionieren kollektive Aufrufe?
- ▶ Wie funktioniert nichtblockierende E/A?
- ▶ Wozu verwendet man gemeinsame Dateizeiger?
- ▶ Wie optimiert man die Leistung?
- ▶ Welche Implementierung gibt es?

# Was ist MPI-2 I/O

---

- ▶ Erweiterung des MPI-Standards um parallele Eingabe/Ausgabe
- ▶ Wird definiert im MPI-2-Standard
- ▶ Semantik ist analog zum Nachrichtenaustausch von Prozessen
  - ▶ Z.B. collective, nonblocking werden auf E/A übertragen
  - ▶ E/A äquivalent zum Senden und Empfangen von Nachrichten



# Wozu parallele E/A in MPI?

---

- ▶ Leistungsgewinn
  - ▶ Z.B. durch kollektive Aufrufe
  - ▶ Z.B. durch asynchrone E/A
  - ▶ Z.B. durch spezielle Dateisichten
- ▶ Einfacherer Zugriff durch Problemanpassung
  - ▶ Z.B. abgeleitete Datentypen bei irregulären Daten
  - ▶ Dadurch auch Portabilität in heterogenen Umgebungen

# Konzepte der MPI-I/O

---

- ▶ File View (Dateisicht)
  - ▶ Prozeßbezogene Sicht auf die Daten einer Datei
- ▶ File Pointer (Dateizeiger)
  - ▶ Individueller/gemeinsamer Dateizeiger
- ▶ Noncontiguous Access (Nichtzusammenhängender Zugriff)
- ▶ Collective Call (Kollektiver Aufruf)
- ▶ Hints (Hinweise)
  - ▶ Informationen für die Implementierungsschicht

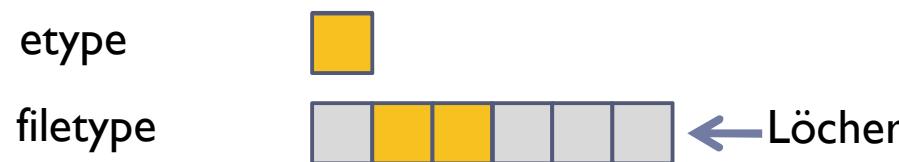
# Einige Definitionen

---

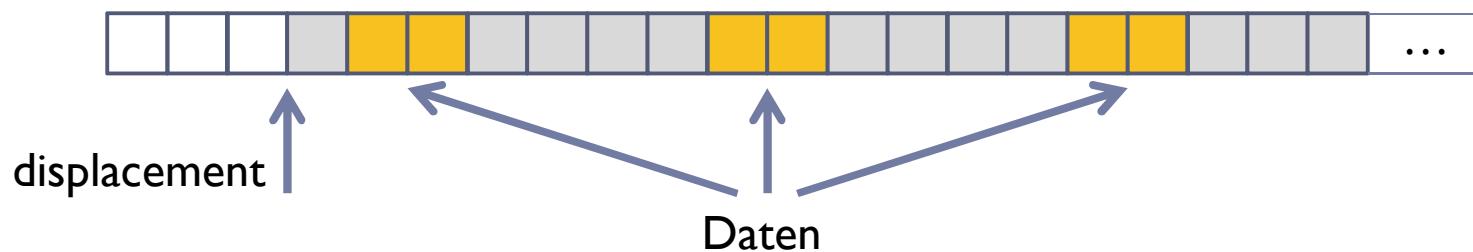
- ▶ file (Datei)
  - ▶ Eine geordnete Sammlung typisierter Daten
  - ▶ Zugriff erfolgt wahlfrei oder sequentiell
  - ▶ Kollektives Öffnen durch eine Gruppe von Prozessen
- ▶ displacement (Versatz)
  - ▶ Eine absolute Byte-Position relativ zum Dateianfang, an der eine Dateisicht beginnt
- ▶ etype (elementary datatype)
  - ▶ Die Einheit, mit der auf die Datei zugegriffen und in ihr positioniert wird

# Einige Definitionen...

- ▶ filetype (Dateityp)
  - ▶ Schablone, nach der eine Datei aufgebaut wird
  - ▶ Besteht aus etype's und gleichgroßen Löchern

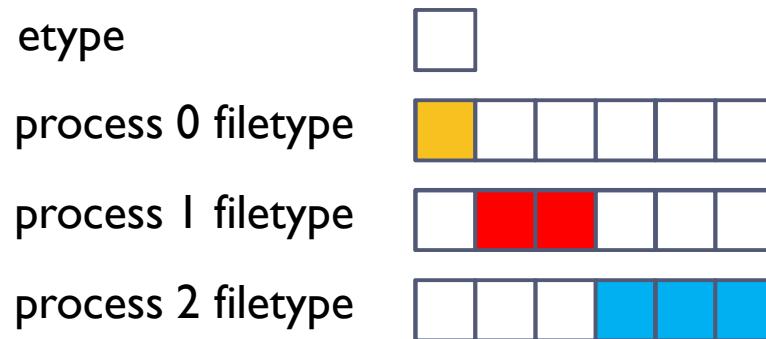


Aufbau einer Datei

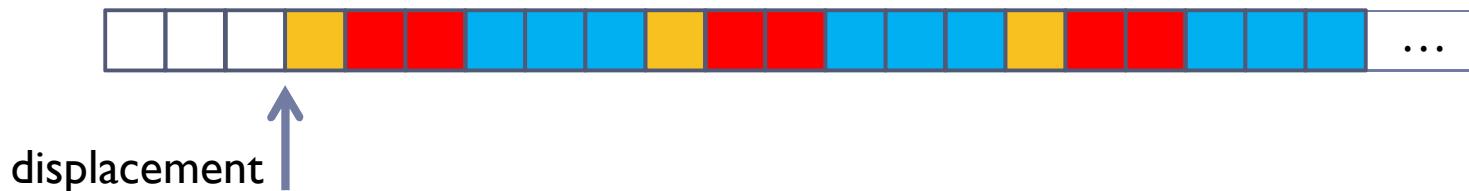


# Einige Definitionen...

- ▶ view (Prozeßdateisicht)
  - ▶ Definiert durch displacement, etype und filetype



Aufbau einer Datei



# Einige Definitionen...

---

- ▶ offset (Versatz)
  - ▶ Position in der Datei relativ im aktuellen view ausgedrückt durch die Anzahl der etype's
- ▶ file size (Dateigröße)
  - ▶ Anzahl Bytes ab dem Anfang der Datei
- ▶ file pointer (Dateizeiger)
  - ▶ Intern von MPI verwalteter Versatz
  - ▶ individual file pointer: jeder Prozeß hat einen
  - ▶ shared file pointer: alle Prozesse teilen sich einen
- ▶ file handle (Datei-Handle ☺)
  - ▶ Wie üblich

# Einfache E/A: mehrere Prozesse lesen/schreiben Datei

---

- ▶ Prozesse öffnen (kollektiv!) eine Datei, ...  
**MPI\_FILE\_OPEN**
- ▶ ... jeder Prozeß positioniert mit seinem eigenen Dateizeiger ...  
**MPI\_FILE\_SEEK**
- ▶ ... und liest aus der Datei/schreibt in die Datei ...  
**MPI\_FILE\_READ**  
**MPI\_FILE\_WRITE**
- ▶ ... und schließt die Datei  
**MPI\_FILE\_CLOSE**

# Einfache E/A: Prototypen (C)

---

```
int MPI_File_open (MPI_Comm comm,
                   char *filename, int amode, MPI_Info info,
                   MPI_File *fh)

int MPI_File_seek (MPI_File fh, MPI_Offset,
                   int whence)

int MPI_File_read (MPI_File fh, void *buf,
                   int count, MPI_Datatype datatype,
                   MPI_Status *status)

int MPI_File_write (MPI_File fh, void *buf,
                    int count, MPI_Datatype datatype,
                    MPI_Status *status)

int MPI_File_close (MPI_File *fh)
```

# Datenzugriff: Positionierung

---

- ▶ Drei Varianten der Positionierung
  - ▶ Explicit offsets
  - ▶ Individual file pointers
  - ▶ Shared file pointers
- ▶ Können innerhalb eines Programms gemischt verwendet werden
- ▶ Syntax
  - ▶ Explicit offsets: **MPI...\_AT**
  - ▶ Shared: **MPI...\_SHARED, MPI...\_ORDERED**

# Nichtzusammenhängende Zugriffe und kollektive Aufrufe

---

- ▶ Bisher vorgestellte E/A auch durch üblich Unix-E/A bewerkstelligbar: eine Datei, zusammenhängende Daten
- ▶ Aber: parallele Programme greifen oft mit mehreren Prozessen unabhängig und auf nichtzusammenhängende Positionen einer Datei zu
- ▶ MPI-2 I/O bietet Funktionen, die mit **einem** Aufruf nichtzusammenhängende Daten lesen können und es mehreren Prozessen gestatten, gleichzeitig auf die Datei zuzugreifen

# Nichtzusammenhängende Zugriffe: Dateisicht

---

- ▶ Durch Dateisichten sieht jeder Prozeß nur „seine“ Daten
- ▶ Dateisicht definiert durch
  - ▶ displacement, etype, filetype  
etype und filetype sind Standard-Datentypen oder aus ihnen abgeleitete Datentypen!
- ▶ Dateisicht definiert durch  
**MPI\_FILE\_SET\_VIEW**
- ▶ Löcher müssen auch definiert werden  
**MPI\_TYPE\_CREATE\_RESIZED**

# Nichtzusammenhängende Zugriffe: Beispiel

```
/* 2 MPI_INT zusammenhängend als derived data type */
MPI_Type_contiguous(2,MPI_INT,&contig);

/* 4 Löcher anhängen; ergibt Größe 6 */
lower_boundary=0;
extent=6*sizeof(int);
MPI_Type_create_resized(contig,lower_boundary,extent,
&filetype);

/* und machen den neuen Typ bekannt */
MPI_Type_commit(&filetyp);

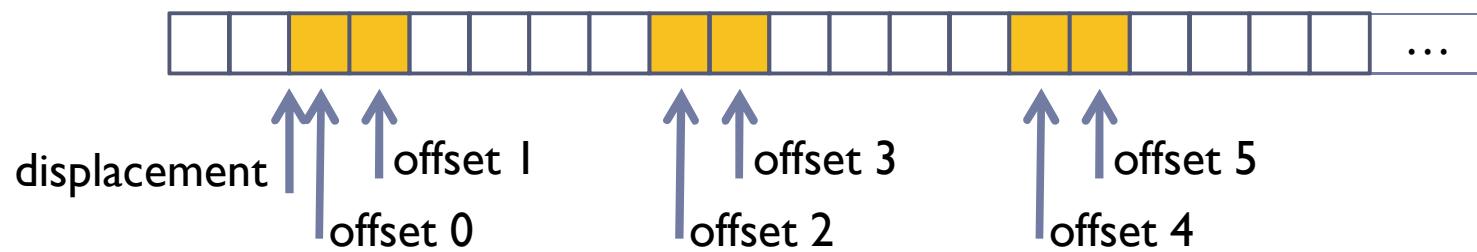
/* und jetzt die Dateisicht */
MPI_File_set_view(filehandle,displacement,etype,filetype,
"native",MPI_INFO_NULL);
```

# Nichtzusammenhängende Zugriffe: Beispiel

etype = MPI\_INT 

filetype = 2\*MPI\_INT resized zur Größe 6 

Aufbau einer Datei



# Kollektive Aufrufe

---

- ▶ Zur weiteren Optimierung können alle Prozesse gleichzeitig in der Datei zugreifen
- ▶ Definition einer Sicht wie zuvor, zusätzlich aber spezielle Funktionen
  - ▶ Erlaubt es der MPI-Implementierung, Zugriffe mehrerer Prozesse zu optimieren
- ▶ Selbst wenn jeder Prozeß nur kleine, unzusammenhängende Stücke liest, kann die MPI-Implementierung (womöglich) einen großen, zusammenhängenden Zugriff daraus erstellen
- ▶ **MPI\_FILE\_READ\_ALL, MPI\_FILE\_WRITE\_ALL**

# Nichtblockierende E/A

---

- ▶ Wird verwendet, um E/A mit Kommunikation und/oder Berechnung zu überlappen
- ▶ Alle nicht-kollektiven(!) Lese- und Schreibfunktionen haben nichtblockierende Entsprechungen
  - ▶ Zur Überprüfung der Beendigung kommt die Standard-MPI-Test-Funktion zum Einsatz
- ▶ Namenskonvention: **MPI\_FILE\_I...**  
Also z.B. **MPI\_FILE\_IREAD**

# Gemeinsamer Dateizeiger

---

- ▶ Bisher nur individuelle Zeiger und Versatz
- ▶ Ebenso möglich: gemeinsamer Zeiger
  - ▶ Von allen Prozessen gemeinsam genutzt
  - ▶ Jeder Zugriff irgendeines Prozesses verändert die Position
  - ▶ Nächster zugreifender Prozeß sieht neue Position
- ▶ Funktionen

**`MPI_FILE_SEEK_SHARED`**

**`MPI_FILE_READ_SHARED`**

**`MPI_FILE_WRITE_SHARED`**

# Gemeinsamer Dateizeiger...

---

- ▶ Bei kollektiven Aufrufen kann gemäß dem Rang der Prozesse serialisiert werden
  - MPI\_FILE\_READ\_ORDERED**
  - MPI\_FILE\_READ\_ORDERED\_BEGIN**
- ▶ Typischer Anwendungsfall
  - ▶ Gemeinsame Protokolldateien

# Hinweise (hints)

---

- ▶ Hinweise geben dem Nutzer die Möglichkeit, Informationen an die MPI-Implementierung durchzurichten
- ▶ Beispiele für Hinweise sind hier
  - ▶ Anzahl der Festplatten, über die eine Datei verteilt werden soll (striping)
  - ▶ Breite der Streifen (stripsize)
- ▶ Hinweise sind immer optional, der Benutzer muß sie nicht angeben
  - ▶ Gleichzeitig darf eine Implementierung Hinweise beliebig ignorieren

# Dateiformate

---

- ▶ Dateien werden als Folge von Bytes gesehen  
Die konkrete Abspeicherung ist Sache des Implementierung
- ▶ MPI definiert drei Daten-Repräsentationen, die unterschiedliche Portabilität erlauben
  - ▶ „native“: keine Wandlung (=Speicherabbild)  
schnell und nichtportabel
  - ▶ „internal“: portabel zwischen den Plattformen, die diese MPI-Implementierung unterstützt
  - ▶ „external32“: 32-bit big endian; portabel zu jeder MPI-Implementierung auf jeder Architektur; langsam

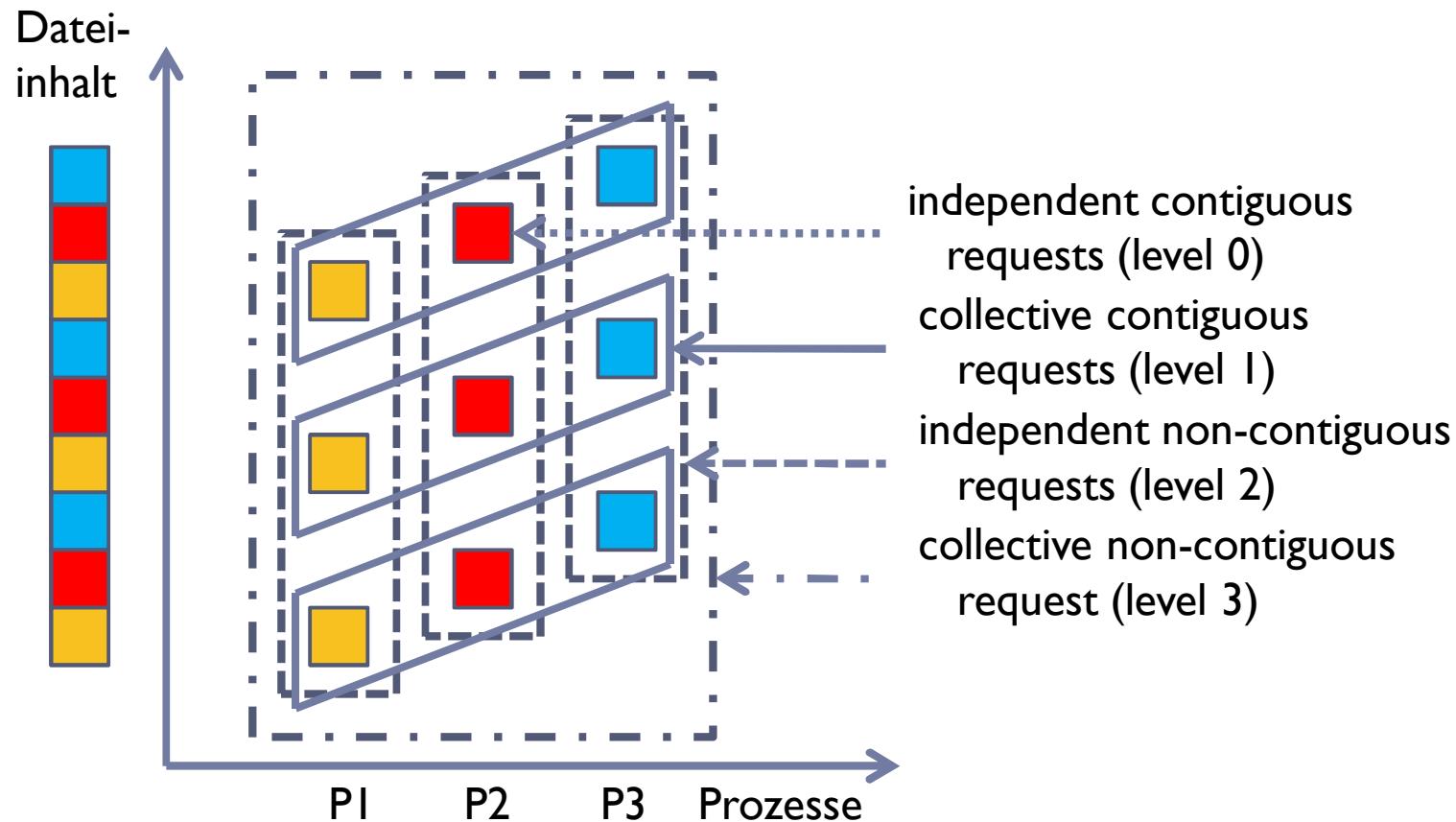


# Leistungsaspekte

---

- ▶ Die Wahl der geeigneten E/A-Methode bestimmt die erzielbare E/A-Bandbreite
  - ▶ Zusammenhängend / nichtzusammenhängend
  - ▶ Kollektiv / nichtkollektiv
- ▶ Beispiel
  - ▶ Datei einer 3x3-Matrix komplexer Datentypen

# Leistungsaspekte...



# Implementierung ROMIO

---

- ▶ ROMIO ist eine/(die) Implementierung von MPI-2 I/O
  - ▶ Gehört zu MPICH, kann aber separat verwendet werden, insbesondere in anderen MPI-Implementierungen
- ▶ ROMIO unterstützt eine Reihe von Hardware-Architekturen und Dateisystemen
- ▶ ROMIO unterstützt alle(?) Merkmale von MPI-2 I/O

# Parallele Eingabe/Ausgabe

## Zusammenfassung

- ▶ Parallele E/A analog zur Kommunikation definiert
- ▶ Verwendet auch abgeleitete Datentypen
- ▶ Dateien sind eine Sequenz elementarer Datentypen
- ▶ Jeder Prozeß hat seine eigene Dateisicht
- ▶ Wir positionieren explizit, mit individuellen Dateizeigern oder einem gemeinsamen
- ▶ Nichtzusammenhängende Zugriffe erhöhen die Effizienz
- ▶ Kollektive Aufrufe erhöhen die Effizienz
- ▶ ROMIO ist die Standard-Implementierung

# Programmierung mit Threads

---

- ▶ Prozesse und Threads
- ▶ Einsatzbereiche Threads
- ▶ Thread-Programmierung
- ▶ Die Pthreads-Schnittstelle
- ▶ (Arten von Threads)
- ▶ (Thread-Scheduling)
- ▶ Linux und Threads

# Programmierung mit Threads

## Die zehn wichtigsten Fragen

- ▶ Worin unterscheiden sich Threads und Prozesse?
- ▶ Warum sind Threads ein Thema beim Hochleistungsrechnen?
- ▶ Wann programmieren wir mit Threads?
- ▶ Was versteht man unter thread-sicheren Funktionsaufrufen?
- ▶ Welche Grundoperationen bietet die Pthreads-Schnittstelle?
- ▶ Wie werden Threads erzeugt?
- ▶ Wie funktioniert ein Mutex?
- ▶ (Was unterscheidet Threads auf Benutzerebene von denen auf Systemebene?)
- ▶ (Wie funktioniert das Thread-Scheduling?)
- ▶ Wie sind Threads im Linux-Betriebssystemkern genutzt?

# Prozesse und Threads

---

## Traditionelle Prozesse

- ▶ Ein Prozeß ist der Ablauf eines Programms
- ▶ Aus Betriebssystemsicht
  - ▶ Prozeß ist Einheit der Ressourcenbelegungen  
(Speicher, Dateien, E/A-Ports)
  - ▶ Prozeß ist Einheit der Prozessorzuteilung
- ▶ Grundidee von Threads
  - ▶ Aufspaltung dieser Eigenschaften:  
Prozeß ist Einheit der Ressourcenbelegung  
Thread ist Einheit der Prozessorzuteilung

# Prozesse und Threads...

---

## Eigenschaften von Threads

- ▶ Threads eines Prozesses haben gemeinsame Ressourcen (Speicher, Dateien, ...)
  - ▶ Einfache, effiziente Kooperation möglich
  - ▶ Aber: kein wechselseitiger Schutz
- ▶ Parameter zur Erzeugung eines Threads
  - ▶ Zeiger auf Programmcode (typisch auf Funktion)
  - ▶ Weitere Parameter: Kellergröße, Scheduling usw.
- ▶ Einheit der Prozessorzuteilung
- ▶ Geringer Verwaltungsaufwand
- ▶ Schneller Wechsel (innerhalb desselben Prozesses)

# Prozesse und Threads...

## Bedeutung von Threads

- ▶ Verringerung der Antwortzeit von Servern
  - ▶ Unterbrechbarkeit langer Anfragen
- ▶ Durchsatzsteigerung
  - ▶ Überlappung blockierender Systemaufrufe
- ▶ Behandlung asynchroner Ereignisse
- ▶ Realzeitanwendungen
  - ▶ Hochpriore Threads für zeitkritische Aufgaben
- ▶ Basis für Parallelverarbeitung auf SMPs/Mehrkernprozessoren
- ▶ Strukturierung von Programmen

# Threads und Hochleistungsrechnen?

---

## Fakten

- ▶ Alle modernen Betriebssysteme basieren auf Threads
- ▶ Moderne Bibliotheken arbeiten mit Threads oder müssen zumindest thread-sicheren Code implementieren
- ▶ Der Compiler für OpenMP-Programme erzeugt Threads

## Benötigtes Wissen beim Programmierer/Anwender

- ▶ Thread-sichere Programmierung
  - ▶ Codeteile müssen von Threads korrekt ausführbar sein
- ▶ Leistungsaspekte bei Abarbeitung von Threads
  - ▶ Scheduling, Priorisierung, ...

# Thread-sichere Funktionen

```
int fib (int n)
{
    int fibn = 0;
    int n1 = 0;
    int n2 = 1;
    int i;

    for (i = 0; i < n; i++)
    {
        fibn = n1 + n2;
        n2 = n1;
        n1 = fibn;
    }
    return fibn;
}
```

Wichtig:  
lokale Variable!

# Thread-Programmierung

---

## Nochmal im Überblick

- ▶ Prozesse
  - ▶ Jeder Prozeß hat eigenen Adreßraum
  - ▶ Kommunikation nur mit Betriebssystem-Unterstützung
    - Signale, pipes, sockets, streams
    - Gemeinsame Speicherbereiche
- ▶ Threads
  - ▶ Gemeinsamer Programmcode für alle Threads
  - ▶ Gemeinsamer Speicher, E/A, usw.
  - ▶ Koordination (Synchronisation) wesentlich!

# Thread-Programmierung...

Varianten der Programmierung von SMPs/Mehrkernprozessoren

- ▶ Unabhängige Prozesse
  - ▶ Z.B. bei WWW- und Datei-Servern (`fork()`)
  - ▶ Keine spezielle Programmierung nötig
  - ▶ Wechselseitiger Schutz der Server-Prozesse
  - ▶ Hohe Antwortzeiten
- ▶ Kommunizierende Prozesse
  - ▶ Wenn Kooperation **und** Schutz erforderlich sind  
Z.B. X Window Client und Server
  - ▶ Kommunikation aufwendig und unkomfortabel

# Thread-Programmierung...

- ▶ Threads
  - ▶ Bei allen Arten von Servern
  - ▶ Für parallele Programme
    - ▶ Hohe Effizienz
    - ▶ Einfache Kooperation
    - ▶ Kein wechselseitiger Schutz durch Betriebssystem
    - ▶ Korrekte Synchronisation schwieriger

# Thread-Programmierung...

Im folgenden

- ▶ Einführung in POSIX-Threads (Pthreads)
  - ▶ Standard IEEE P1003.1c
  - ▶ In vielen Systemen realisiert
  - ▶ Konzepte in anderen Thread-Realisierungen meist ähnlich
  - ▶ POSIX-konforme Realisierungen unter Linux

# Die Pthread-Schnittstelle

---

Eingeteilt in drei (informelle) Klassen

- ▶ Thread-Verwaltung
- ▶ Mutex-Verwaltung
- ▶ Bedingungsvariablen-Verwaltung

# Die Pthread-Schnittstelle...

## Thread-Erzeugung

- ▶ Programmiermodell
  - ▶ Bei Start eines Prozesses existiert genau ein Thread
  - ▶ Dieser erzeugt ggf. weitere Threads und wartet auf deren Ende
  - ▶ Terminierung des Prozesses bei Terminierung des Master-Thread

- ▶ Funktion zur Thread-Erzeugung

```
int pthread_create(pthread_t *new_thrd_ID,  
                  const pthread_attr_t *attr,  
                  void *(*start_func)(void *),  
                  void *arg)
```

# Die Pthread-Schnittstelle...

```
void *PrintHello(void *threadid)
{ long tid; tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n",
         tid);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{ pthread_t threads[NUM_THREADS];
  int rc; long t;
  for(t=0;t<NUM_THREADS;t++){
    printf("In main: creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL,
                       PrintHello, (void *)t);
  }
}
```

# Die Pthread-Schnittstelle...

---

Mögliche Ausgabe des Programms bei NUM\_THREADS=5

In main: creating thread 0

In main: creating thread 1

Hello World! It's me, thread #0!

In main: creating thread 2

Hello World! It's me, thread #1!

Hello World! It's me, thread #2!

In main: creating thread 3

In main: creating thread 4

Hello World! It's me, thread #3!

Hello World! It's me, thread #4!

# Die Pthread-Schnittstelle...

- ▶ Thread-Attribute
  - ▶ Keller und Kellergröße
  - ▶ Scheduling-Schema und Priorität
  - ▶ **detachstate:** Verhalten bei Beendigung des Threads

Bei **detached thread** erfolgt die Freigabe der Ressourcen sofort bei Beendigung, ansonsten erst nach Abschlußsynch�nisation

# Die Pthread-Schnittstelle...

## Thread-Attribute

- ▶ **`pthread_attr_init`**  
Attributstruktur initialisieren
- ▶ **`pthread_attr_destroy`**  
Attributstruktur löschen
- ▶ **`pthread_attr_get / pthread_attr_set`**  
Lesen und setzen von Attributwerten

# Die Pthread-Schnittstelle...

## Thread-Verwaltung

- ▶ **pthread\_self**  
Liefert eigene Thread-ID
- ▶ **pthread\_exit**  
Eigene Terminierung
- ▶ **pthread\_cancel**  
Terminiert anderen Thread
  - ▶ Kann maskiert werden
  - ▶ Terminierung nur an bestimmten Punkten
  - ▶ Vor Terminierung **cleanup handler** aufrufen

# Die Pthread-Schnittstelle...

---

## Thread-Verwaltung...

- ▶ **pthread\_join**  
Wartet auf Terminierung des spezifizierten Threads  
(Abschlußsynchronisation)
- ▶ **pthread\_sigmask**  
Setzt Signalmaske (jeder Thread hat eigene Maske)
- ▶ **pthread\_kill**  
Sendet Signal an anderen Thread innerhalb des Prozesses
  - ▶ Von außen nur Signal an *irgendeinen* Thread möglich

# Die Pthread-Schnittstelle...

---

## Thread-Synchronisation durch eine Barriere

- ▶ **pthread\_barrier\_init**  
Erzeuge und initiere eine Barriere und spezifizierte die Anzahl der Threads, die über sie synchronisiert werden sollen
- ▶ **pthread\_barrier\_wait**  
Jeder zu synchronisierende Thread ruft die Funktion auf und wird schlafend gelegt bis die vorher spezifizierte Anzahl von Threads in die Barriere eingelaufen sind. Danach werden alle fortgesetzt. Einer(!) der fortgesetzten Threads erhält den reservierten Rückgabewert PTHREAD\_BARRIER\_SERIAL\_THREAD. Man nutzt diesen, um Aktionen zu steuern, die nur einer durchführen soll, wie z.B. das Melden des Verlassens der Barriere. Alle anderen erhalten den Rückgabewert 0.
- ▶ **pthread\_barrier\_destroy**  
Lösche die Barriere mit ihren Ressourcen

# Die Pthread-Schnittstelle...

---

## Mutex-Operation

- ▶ (wechselseitiger Ausschluß, mutual exclusion)
- ▶ Z.B. Schutz von globalen Variablen bei mehreren Schreibern
  
- ▶ **pthread\_mutex\_init**  
Initialisiert Sperrvariable (mutex)
- ▶ **pthread\_mutex\_destroy**
- ▶ **pthread\_mutex\_lock**  
Blockiert Thread, bis Sperre frei ist (a) und belegt dann die Sperre (b)
- ▶ **pthread\_mutex\_trylock**  
Belegt Sperre, falls möglich / kein Blockieren
- ▶ **pthread\_mutex\_unlock**

# Die Pthread-Schnittstelle...

---

## Anmerkungen zu Mutex-Operationen

- ▶ Operationenpaar (a), (b) muß unteilbar sein
- ▶ Bei Terminierung eines Threads werden Sperren nicht automatisch freigegeben
- ▶ Prioritätswechselprotokolle in einigen Implementierungen

# Die Pthread-Schnittstelle...

---

## Bedingungsvariablen

- ▶ Zur Signalisierung von Bedingungen zwischen Threads
- ▶ Erlauben Realisierung von Monitoren
  - (strukturierte Form des wechselseitigen Ausschluß nach Hoare)
  - ▶ Extern sichtbare Funktionen eines Moduls stehen unter wechselseitigem Ausschluß
  - ▶ Aufrufer braucht sich damit nicht um Synchronisation zu kümmern

# Die Pthread-Schnittstelle...

## Operationen auf Bedingungsvariablen

- ▶ **pthread\_cond\_init**  
Initialisiert Bedingungsvariable
- ▶ **pthread\_cond\_destroy**
- ▶ **pthread\_cond\_wait**  
Gibt eine Sperre frei (a), blockiert dann bis Bedingung signalisiert wird (b) und belegt Sperre wieder
- ▶ **pthread\_cond\_signal**  
Signalisiert Bedingung; setzt einen wartenden Thread fort

# Die Pthread-Schnittstelle...

---

## Operationen auf Bedingungsvariablen...

- ▶ **pthread\_cond\_timewait**  
Wie **wait** aber mit begrenzter Wartezeit
- ▶ **pthread\_cond\_broadcast**  
Wie **signal**, aber mit Fortsetzung aller wartenden Threads

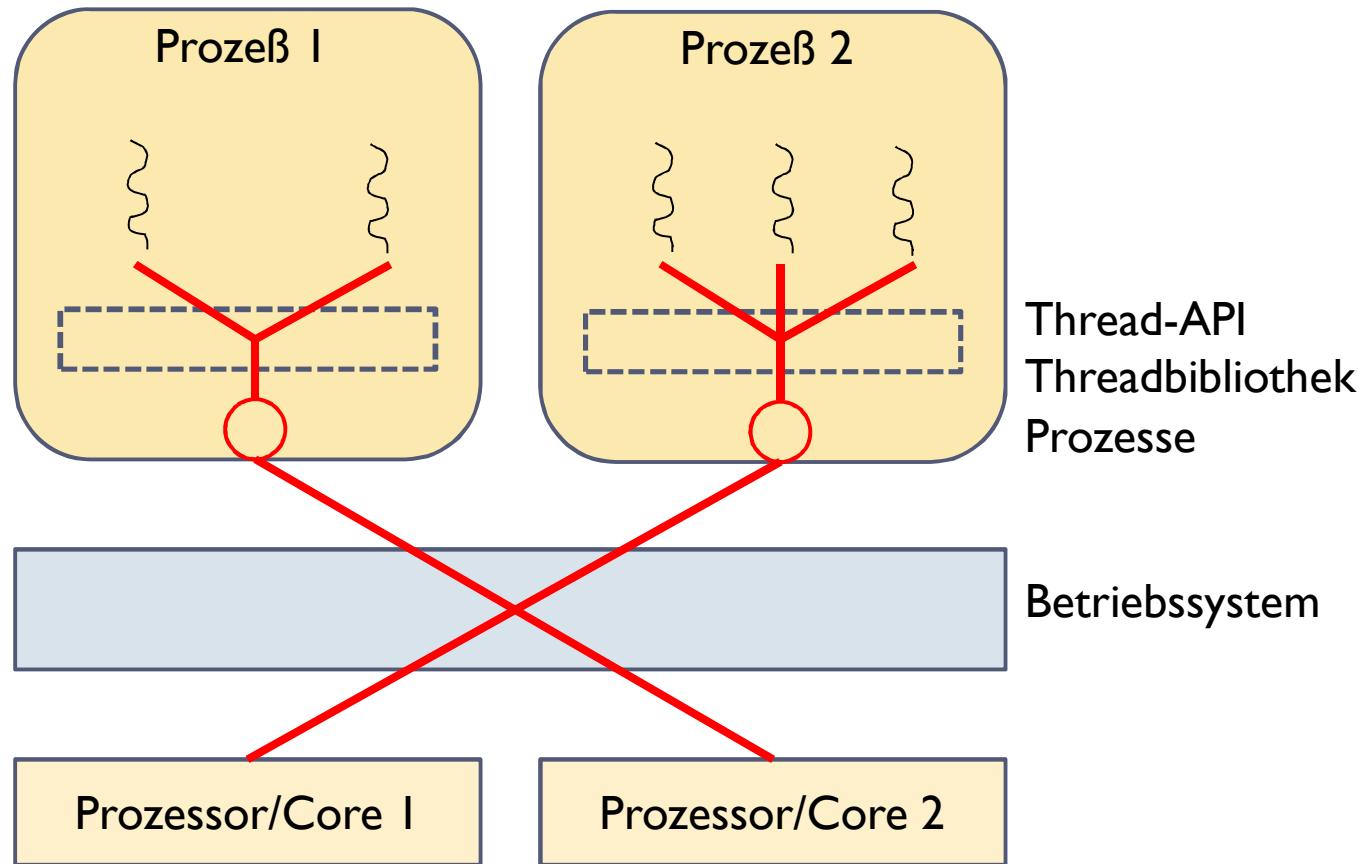
# Die Pthread-Schnittstelle...

## Amerkungen zu Bedingungsvariablen

- ▶ Operationspaar (a), (b) in `pthread_cond_wait` (Freigabe des Mutex, Warten auf Bedingung) muß unteilbar implementiert sein
- ▶ Bedingungsvariable „merken“ sich die Signalisierung nicht
  - ▶ Wenn bei Signalisierung kein wartender Thread existiert, bleibt sie ohne Wirkung
  - ▶ Auch dann, wenn später ein Thread auf die Bedingung wartet
- ▶ Signalisierung muß bei belegtem Mutex erfolgen

# (Arten von Threads)

Threads auf Benutzerebene (user threads)



# Arten von Threads...

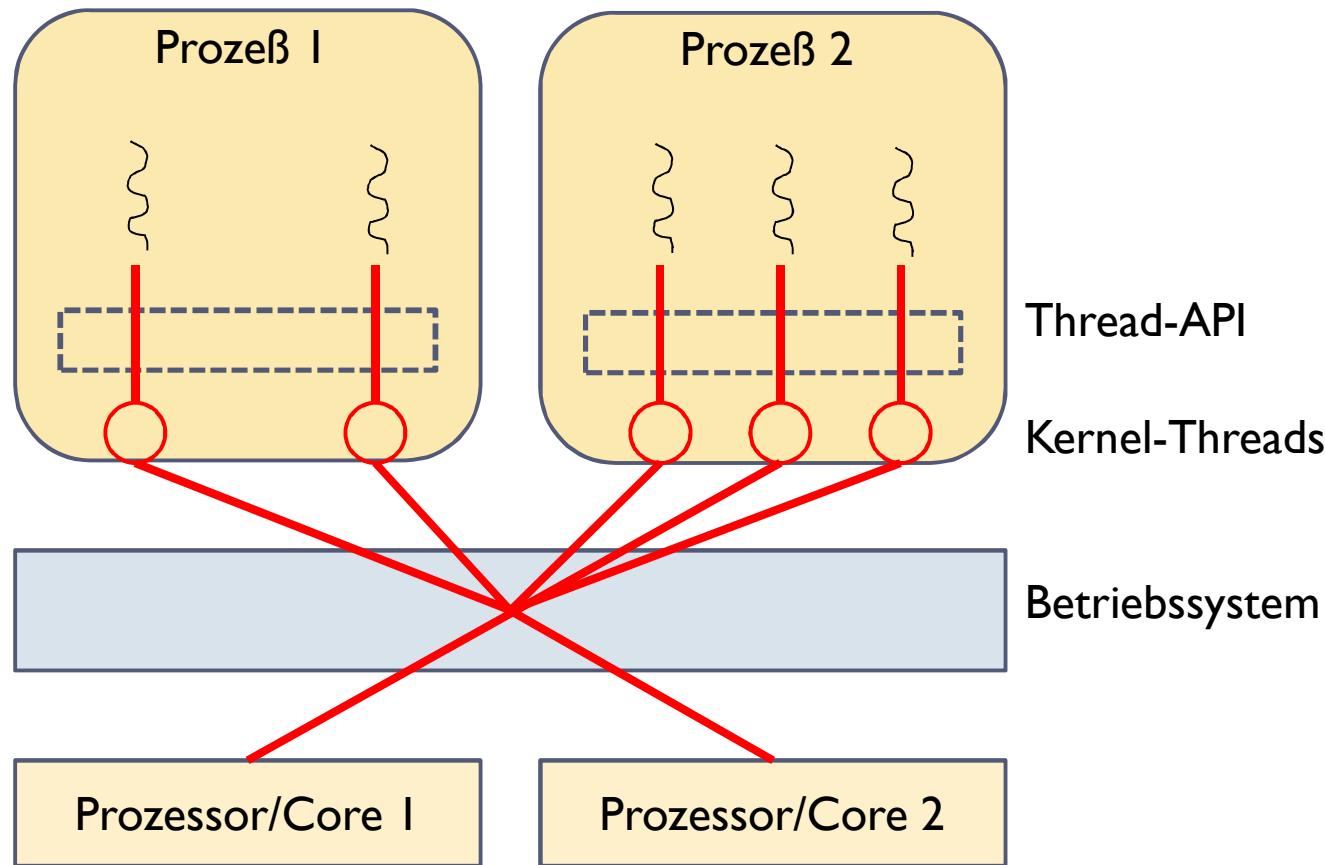
---

## Threads auf Benutzerebene...

- ▶ Schnelle Thread-Erzeugung und Kontextwechsel
  - ▶ Keine Betriebssystemaufrufe notwendig
- ▶ Relativ gute Portierbarkeit
  - ▶ Aber: Problem mit nicht-wiedereintrittsfähigen Laufzeitbibliotheken
- ▶ Geringe Belastung des BS-Kerns
- ▶ Keine echte Parallelität innerhalb eines Prozesses
- ▶ Blockierung des Prozesses und damit aller seiner Threads bei blockierendem Systemaufruf
- ▶ Keine Koordination zwischen Betriebssystem- und Thread-Scheduler

# Arten von Threads...

## Threads auf Systemebene (kernel threads)



# Arten von Threads...

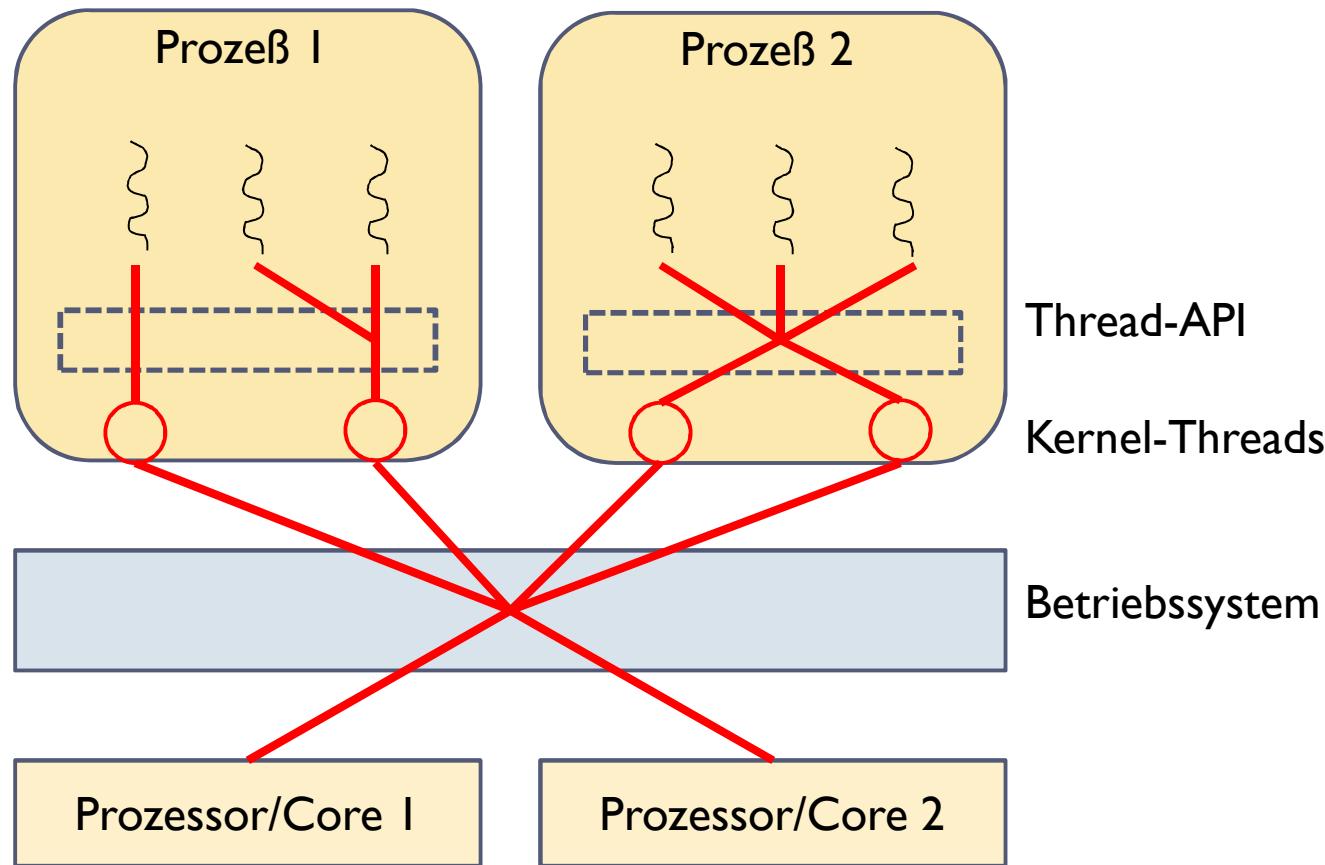
---

## Threads auf Systemebene...

- ▶ Langsamer als User Threads wegen Systemaufrufen
  - ▶ Z.B. Zeit zur Erzeugung auf SparcStation2  
Userthread: 50µs / Kernelthread: 350µs / Prozeß 1700µs
- ▶ Unterschiedliche Schnittstellen und Semantiken bei unterschiedlichen Schnittstellen
  - ▶ Aber POSIX-Standard IEEE P1003.1c: Pthreads
- ▶ Parallelität auch innerhalb eines Prozesses
- ▶ Keine Probleme mit Blockierungen und Wiedereintrittsfähigkeit (Reentrancy)

# Arten von Threads...

## Hybride Thread-Realisierung



# Arten von Threads...

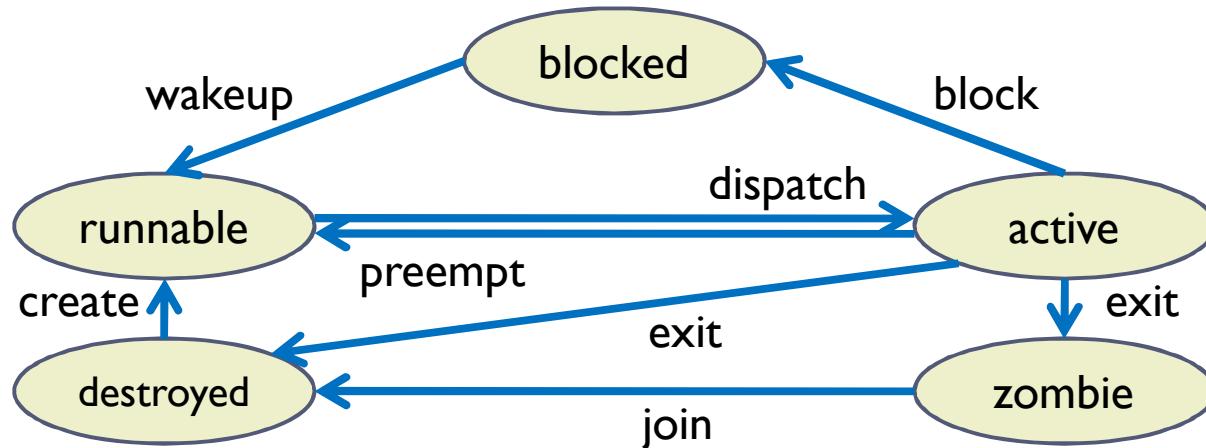
---

## Hybride Thread-Realisierung

- ▶ Anzahl der Kernel-Threads von der Thread-Bibliothek bestimmt
  - Abhängig von Anzahl User-Threads und Prozessoren
- ▶ Anzahl Kernel-Threads und Zuordnung von User-Thread zu Kernel-Thread steuerbar
  - ▶ Z.B. eins-zu-eins-Zuordnung wenn zeitkritisch
- ▶ Anzahl von Kernel-Threads bestimmt
  - ▶ Maximalen Parallelitätsgrad
  - ▶ Maximale Zahl gleichzeitig blockierender Systemaufrufe

# (Thread-Scheduling)

- ▶ Scheduling: Zuteilung rechenbereiter Threads an Prozessoren
- ▶ Präemptives Scheduling: rechnender Thread kann unterbrochen werden
- ▶ Threadzustände  
(POSIX-Implementierungsmodell)



# Thread-Scheduling...

## Schedulingverfahren

- ▶ Üblicherweise: Scheduling mit Prioritäten
- ▶ Ziel: Threads mit höchster Priorität rechnen
- ▶ Strategie bei gleichen Prioritäten
  - ▶ FIFO (First-In-First-Out)  
Unterbrechnung nur durch höherpriore Threads
  - ▶ RR (Round Robin)  
Unterbrechung und Weiterschalten nach Ablauf einer Zeitscheibe

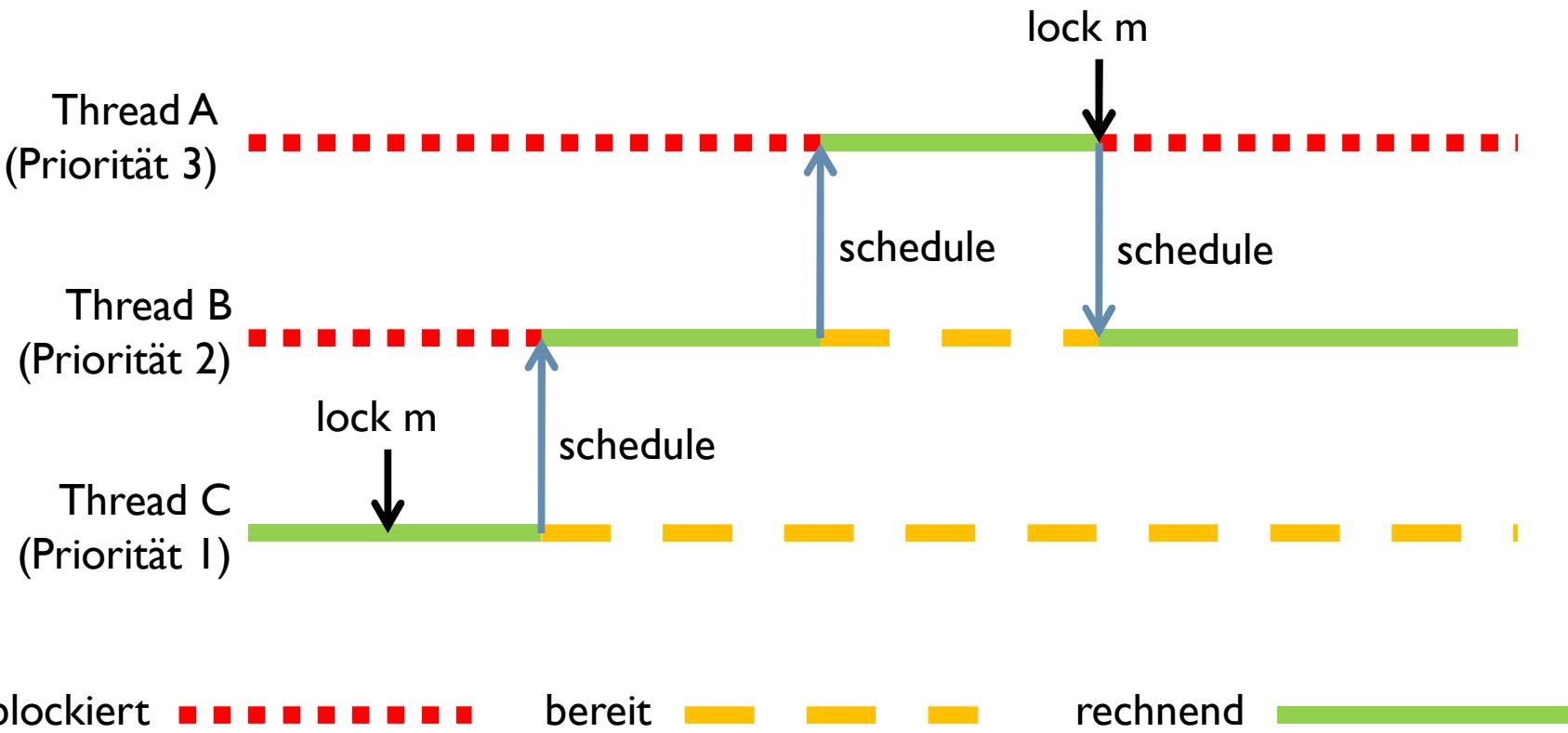
# Thread-Scheduling...

## Probleme

- ▶ Fairness bei Mehrbenutzersystemen mit FIFO  
Bösartiger Benutzer kann System blockieren
- ▶ Abhilfe: Zeitscheiben und dynamische Änderung der Prioritäten durch das Betriebssystem
  - ▶ Durch Rechnen: Priorität ↓, Zeitscheibe ↑
  - ▶ Durch Warten: Priorität ↑, Zeitscheibe ↓
- ▶ Niederpriore Threads können höherpriore blockieren
  - ▶ Prioritätsinversion

# Thread-Scheduling...

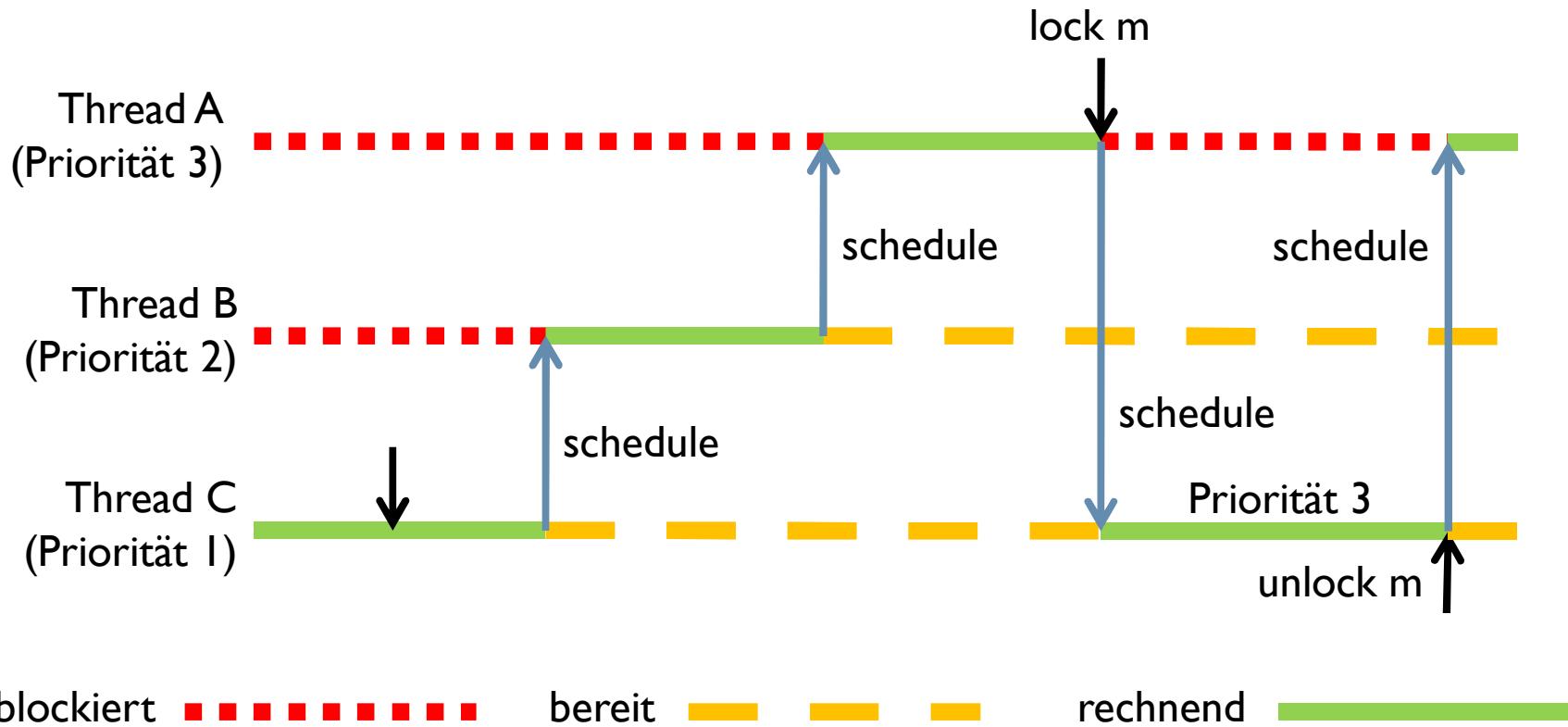
## Prioritätsinversion



# Thread-Scheduling...

Lösung: Prioritätswechselprotokolle

Thread, der Sperre hält, erbt Priorität des wartenden Threads



# Thread-Scheduling...

---

## Komplikationen

- ▶ Manche Threads müssen auf einem bestimmten Prozessor bleiben (z.B. Interrupt-Threads)
- ▶ Interaktion mit Caches
  - ▶ Effizienzverlust, wenn Thread wandert  
Konzept: Prozessoraffinität
- ▶ Bei starker Synchronisation zwischen Threads
  - ▶ Effiziente Abarbeitung nur, wenn alle Threads eines Prozesses gleichzeitig laufen  
Konzept: Gangscheduling

# Linux und Threads

---

## Linux Thread-Bibliotheken

- ▶ Next Generation Posix Threading (NGPT)
  - ▶ nicht mehr weitergeführt (!)
- ▶ LinuxThreads
  - ▶ nicht mehr weitergeführt (!)
- ▶ The Native Posix Thread Library (NPTL)
  - ▶ dies ist jetzt im Kernel 2.6 die Standard-Implementierung für die POSIX-Threads
  - ▶ ein POSIX-Thread wird auf einen Linux-Threads abgebildet

# Linux und Threads...

## Linux-Threads

- ▶ Spezielle Variante der allgemeinen Prozesse
- ▶ Schnell erzeugbar, effiziente Nutzung
- ▶ Teilen sich alle Ressourcen mit Eltern-Prozeß
- ▶ Erzeugt mit `clone()`-Aufruf (wie Kindprozesse auch)
  - ▶ Andere Parameter gestatten gemeinsame Ressourcennutzung

## Kernel-Threads in Linux

- ▶ Spezielles Thread-Objekt im Betriebssystemkern
- ▶ Kein eigener Adreßraum
- ▶ Arbeiten nur im Betriebssystemmodus
- ▶ Sind allerdings dem Scheduler unterworfen
- ▶ Zur Strukturierung von Aktivitäten im Kernel

# Vergleich der Ansätze

|                             | Pthreads  | OpenMP     | MPI        |
|-----------------------------|-----------|------------|------------|
| Skalierbarkeit              | Begrenzt  | Begrenzt   | Ja         |
| Fortran / C und C++         | Nein / Ja | Ja / Ja    | Ja / Ja    |
| Hohe Abstraktion            | Nein      | Ja         | Nein       |
| Leistungsorientierung       | Nein      | Ja         | Ja         |
| Portierbarkeit              | Ja        | Ja         | Ja         |
| Herstellerunterstützung     | Unix/SMP  | Verbreitet | Verbreitet |
| Inkrement. Parallelisierung | Nein      | Ja         | Nein       |

# Programmierung mit Threads

## Zusammenfassung

- ▶ Threads trennen Einheiten der Ressourcenbelegung von Einheiten der Prozessorzuteilung
- ▶ Threads werden für echt parallele Programme aber auch als Strukturierungsmittel eingesetzt
- ▶ Die Pthreads-Schnittstelle definiert einen Standard zur Nutzung von Threads
- ▶ (Threads können auf der Ebene des Benutzers und des Systems realisiert werden)
- ▶ (In der Praxis finden wir hybride Ansätze (1:1))
- ▶ (Das Scheduling von Threads ist komplexer als das von Prozessen)

# Programmierung mit OpenMP

---

- ▶ Konzepte und **Hello World**
- ▶ Überblick
- ▶ Parallelisierung einer Schleife
- ▶ Eine komplexere Schleife
- ▶ Parallele Bereiche
- ▶ Lastausgleich
- ▶ Parallele und sequentielle Abschnitte
- ▶ Vergleich mit anderen Ansätzen

# Programmierung mit OpenMP

## Die zehn wichtigsten Fragen

- ▶ Was charakterisiert den Ansatz von OpenMP?
- ▶ Welche Konzeptklassen beinhaltet OpenMP?
- ▶ Welche Konstrukte zur Parallelarbeit gibt es?
- ▶ Wie werden Variablen verwaltet?
- ▶ Welche Synchronisationskonzepte gibt es?
- ▶ Wie werden Schleifen parallelisiert?
- ▶ Was ist das Hauptproblem der parallelen Schleifen?
- ▶ Wofür verwendet man sequentielle Abschnitte?
- ▶ Wie programmiert man allgemeine Parallelarbeit?
- ▶ Welche Konzepte zum Lastausgleich gibt es?

# Bisherige Programmiermodelle

---

## Ansätze mit Bibliotheken

- ▶ MPI für verteilten Speicher
- ▶ Pthreads für gemeinsamen Speicher

Automatische Parallelisierung durch Compiler immer noch nicht möglich

- ▶ Trotz langjähriger Forschung weder für Nachrichtenaustausch noch für gemeinsame Speicherbereiche

Aber: Compilergestützte Parallelisierung möglich

- ▶ Im Falle von OpenMP für gemeinsamen Speicher!

# Neuer Ansatz: OpenMP (Open Multi-Processing)

---

- ▶ Keine neue Programmiersprache
- ▶ Arbeitet mit Fortran und C/C++ zusammen
  
- ▶ Compiler-Direktiven steuern Übersetzung
- ▶ Zusätzliche (kleine) Bibliothek
- ▶ Direktiven+Bibliothek sind das API von OpenMP
  
- ▶ OpenMP-Compiler übersetzt in Programme mit Threads (nicht weiter spezifiziert)
  - ▶ Verwendung ausschließlich für gemeinsamen Speicher!

# OpenMP´s Hello World

```
programm hello
    print *, "Hello world from thread:"
 !$omp parallel
    print *, omp_get_thread_num()
 !$omp end parallel
    print *, "Back to the sequential world."
end
```

- ▶ Umgebungsvariable: **OMP\_NUM\_THREADS**
- ▶ Bibliotheksauftrag: **omp\_get\_thread\_num()**
- ▶ Compiler-Direktive: **!\$omp parallel**
  
- ▶ Thread-Nummern: 0...**OMP\_NUM\_THREADS-1**

# Warum ein Compiler-Ansatz?

---

Zunächst reiner Compiler-Ansatz geplant

Vorteil gegenüber Bibliotheken

- ▶ Nicht-OpenMP-Compiler ignorieren parallele Konstrukte automatisch
- ▶ Compiler können zusätzlich optimieren
- ▶ Inkrementelle Parallelisierung möglich

Reiner Compiler-Ansatz zu schwierig

- ▶ Erweiterung durch einige einfache Bibliotheksaufrufe

# Die Geschichte von OpenMP

---

- ▶ OpenMP sehr neu: seit 1997
- ▶ Erste Ansätze von SGI vorangetrieben
- ▶ Hat aber lange Vorgeschichte
  - ▶ Ehemaliger ANSI X3H5-Standard zur Programmierung von gemeinsamem Speicher  
Früher auf parallelen Maschinen verbreitet
- ▶ OpenMP jetzt von allen Herstellern akzeptiert
- ▶ Von unabhängiger Organisation gefördert

# Überblick über OpenMP

---

Portabilität: Neuübersetzung reicht aus

Kategorien der Spracherweiterungen

- ▶ Kontrollstrukturen, um Parallelismus auszudrücken
- ▶ Datenumgebungskonstrukte zur Kommunikation zwischen Threads
- ▶ Synchronisationskonstrukte zur Ablaufsteuerung von Threads

# Überblick über OpenMP...

---

## Compiler-Direktiven

- ▶ in Fortran

```
!$OMP <directive> <clauses>
```

- ▶ In C/C++

```
#pragma omp <directive> <clauses>
```

Zusätzlich bedingte Übersetzung der OpenMP-Bibliotheksaufrufe

# Überblick über OpenMP...

---

## Parallele Kontrollstrukturen

- ▶ Ausführungsmodell genannt fork/join-Modell
- ▶ Parallele Kontrollstrukturen starten neue Threads und übergeben ihnen die Kontrolle

## Zwei Varianten

- ▶ **parallel**-Direktive: umschließt Block und erzeugt Menge von Threads, die den Block nebenläufig abarbeiten
- ▶ **do**-Direktive: verteilt Instanzen von Schleifendurchläufen auf Threads

# Überblick über OpenMP...

---

Kommunikation und Datenumgebung

- ▶ Regelt, wer wann wo welche Daten sehen kann

Programm beginnt immer mit einem Thread  
(master-Thread)

Bei **parallel** werden neue Threads gestartet –  
jeweils mit eigenem Keller

Variablen können von folgenden Typen sein

- ▶ **shared** – allen Threads gemeinsam zugängliche Variable
- ▶ **private** – thread-lokale Variable
- ▶ **reduction** – Mischform zur Ergebniszusammenführung

# Überblick über OpenMP...

---

## Synchronisation

- ▶ Regelt den Ablauf der Threads

## Zwei Hauptformen

- ▶ Wechselseitiger Ausschluß mittels **critical**-Direktive
- ▶ Ereignis-Synchronisation mittels **barrier**-Direktive

## Weitere Konstrukte zur Bequemlichkeit oder Leistungsoptimierung

# Parallelisierung einer Schleife

```
subroutine saxpy(z,a,x,y,n)
integer i,n
real z(n),a,x(n),y

do i=1,n
    z(i)=a*x(i)+y
enddo

return
end
```

Keine Datenabhängigkeiten in der Schleife

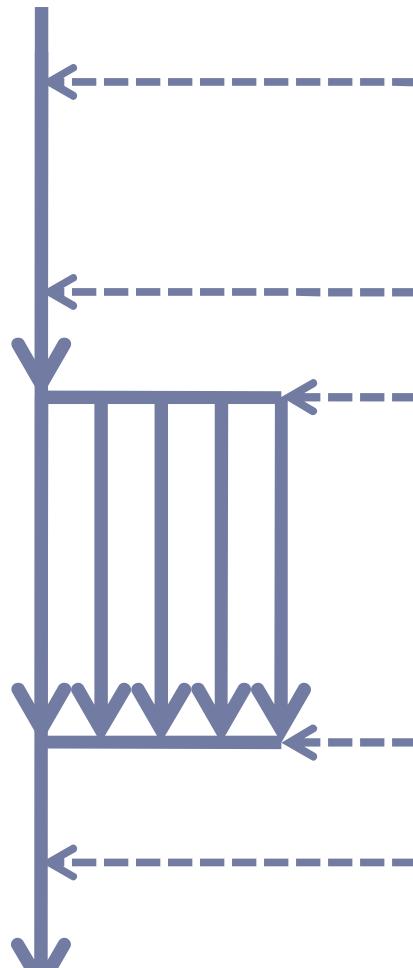
# Parallelisierung einer Schleife...

```
subroutine saxpy(z,a,x,y,n)
integer i,n
real z(n),a,x(n),y
 !$omp parallel do
 do i=1,n
   z(i)=a*x(i)+y
 enddo
 !$omp end parallel do
 return
end
```

Hier Parallelisierung alleine auf Schleifenindexebene

- ▶ Andere Ebenen auch möglich

# Parallelisierung einer Schleife...



## Ausführungsmodell

master-Thread (mT) bearbeitet seriellen Anteil des Codes

mT betritt saxpy-Routine

mT trifft auf **parallel-do**-Direktive und kreiert Threads

mT und Kind-Threads teilen die Iterationen auf und bearbeiten sie nebenläufig

Implizite Barriere: warte auf alle Threads

mT macht nach der **do**-Schleife alleine weiter  
Kindthreads verschwinden

# Parallelisierung einer Schleife...

## Kommunikation und Datengültigkeit

- ▶ Außerhalb des **parallel-do**-Blocks
  - ▶ Die Variablen z, a, x, y, n, i sind nur einmal vorhanden
- ▶ Innerhalb des **parallel-do**-Blocks
  - ▶ Die Variablen z, a, x, y, n sind nur einmal vorhanden  
Vorsicht mit der Semantik beim Zugriff!
  - ▶ Die Schleifenvariable wird als thread-lokale Variable angelegt  
Aktualisierungen in einem Thread sind in anderen Threads nicht sichtbar

# Parallelisierung einer Schleife...

## Synchronisation

- ▶ Anforderungen
  - ▶ Variable z muß aktualisiert worden sein, wenn mit Anweisungen nach der Schleife fortgesetzt wird
- ▶ Realisierung
  - ▶ **parallel do**-Direktive hat implizite Barriere am Schleifenende

# Eine kompliziertere Schleife

```
real*8 x,y
integer i,j,m,n,maxiter
integer depth(*,*)
integer mandel_val
...
maxiter=200

do i=1,m
    do j=1,n
        x=i/real(m)
        y=j/real(n)
        depth(j,i)=mandel_val(x,y,maxiter)
    enddo
enddo
```

# Eine kompliziertere Schleife...

## Zur Funktion `mandel_val`

- ▶ Darf nur von Eingabeparametern abhängen
- ▶ D.h. muß durch parallele Threads nutzbar sein (*thread-safe*)

## Zu den Variablen

- ▶ Variable i per default **private**  
(weil diese Schleife parallelisiert wird)
- ▶ Variablen j,x,y explizit auf **private** gesetzt  
(default wäre **shared**)

# Eine kompliziertere Schleife...

```
real*8 x,y
integer i,j,m,n,maxiter
integer depth(*,*)
integer mandel_val
...
maxiter=200
!$omp parallel do private(j,x,y)
do i=1,m
    do j=1,n
        x=i/real(m)
        y=j/real(n)
        depth(j,i)=mandel_val(x,y,maxiter)
    enddo
enddo
!$omp end parallel do
```

# Eine Verkomplizierung

```
maxiter=200
do i=1,m
    do j=1,n
        x=i/real(m)
        y=j/real(n)
        depth(j,i)=mandel_val(x,y,maxiter)
        total_iters=total_iters+depth(j,i)
    enddo
enddo
```

Mitzählen der gesamten Iterationen

Variable **total\_iters** per default **shared**

# Eine Verkomplizierung...

Zugriff auf **total\_iters** in kritischem Bereich

```
!$omp critical
```

```
    total_iters=total_iters+depth(j,i)
```

```
!$omp end critical
```

Verfahren wird beim Zugriff auf **total\_iters** serialisiert

- ▶ Zugriffszeit sollte prozentual kleiner Anteil sein!

# Reduktion

```
maxiter=200
total_iters=0
!$omp parallel do private(j,x,y)
!$omp+ reduction(+:total_iters)
    do i=1,m
        do j=1,n
            x=i/real(m)
            y=j/real(n)
            depth(j,i)=mandel_val(x,y,maxiter)
            total_iters=total_iters+depth(j,i)
        enddo
    enddo
!$omp end parallel do
```

# Schleifenparallelisierung

---

Hauptproblem der Praxis:

Datenabhängigkeiten zwischen Schleifenindizes

Bspiel:

```
do i=2,n  
    a(i)=a(i)+a(i-1)  
enddo
```

Lösung des Problems

- ▶ Komplizierte Methoden zum Finden der Abhängigkeiten
  - ▶ Großes Forschungsgebiet seit vielen Jahren
- ▶ Verschiedene Methoden zu ihrer Beseitigung
  - ▶ Zusätzliche Variable
  - ▶ Zugriffskoordination mit kritischem Bereich

# Parallele Bereiche

---

Bisher nur parallele Schleifen (feingranular)

Jetzt auch grobgranularer Parallelismus

Konstrukt: **parallel** / **end parallel**

- ▶ eingeschlossener Code wird mit mehreren Threads nebenläufig bearbeitet

# Parallele Bereiche...

```
maxiter=200
  do i=1,m
    do j=1,n
      x=i/real(m)
      y=j/real(n)
      depth(j,i)=mandel_val(x,y,maxiter)
    enddo
  enddo
  do i=1,m
    do j=1,n
      dith_depth(j,i)=0.5*depth(j,i) +
$          0.25*(depth(j-1,i)+depth(j+1,i))
    enddo
  enddo
```

# Parallele Bereiche...

```
maxiter=200
 !$omp parallel
 !$omp+ private(i,j,x,y)
 !$omp+ private(my_width,my_thread,i_start,i_end)
     my_width=m/2
     my_thread=omp_get_thread_num()
     i_start=1+my_thread*my_width
     i_end=i_start+my_width-1
     do i=i_start,i_end
         do j=1,n
             x=i/real(m)
             y=j/real(n)
             depth(j,i)=mandel_val(x,y,maxiter)
         enddo
     enddo
```

# Parallele Bereiche...

```
do i=i_start,i_end
    do j=1,n
        dith_depth(j,i)=0.5*depth(j,i) +
$                          0.25*(depth(j-1,i)+depth(j+1,i))
    enddo
enddo
!$omp end parallel
```

Diese Parallelisierung ist für genau 2 Threads  
programmiert  
Keine Compiler-Parallelisierung auf Schleifenebene

# Lastausgleich

---

Standard: jeder Thread erledigt gleich viele Iterationen einer Schleife

Aber: falls Schleifenrumpf in der Bearbeitungszeit variiert, führt das zu Lastungleichheit

Mechanismus: **schedule**-Klausel

- ▶ **static**: Zuteilung der Indizes zu Schleifenbeginn
- ▶ **dynamic**: Indizes werden zur Laufzeit zugeteilt

# Lastausgleich...

## **schedule(type[,chunk])**

- ▶ **static**: etwa Gleichverteilung
- ▶ **static chunk**: Rundumverteilung von Blöcken der Größe **chunk**
- ▶ **dynamic**: dynamische Rundumverteilung von Blöcken der Größe **chunk** (default=1)
- ▶ **guided**: Die Blockgröße sinkt exponentiell bis auf **chunk** ab; dynamische Rundumverteilung
- ▶ **runtime**: Verfahren wird durch die Umgebungsvariable **OMP\_SCHEDULE** bestimmt

# Parallele Abschnitte

---

Bei nichtiterativen Arbeitslasten:

Zuteilung von Code zu Threads

```
!$omp section [clause [,] [clause...]]
[ !$omp section]
    code for the first section
[ !$omp section]
    code for the second section
    ...
]

!$omp end sections [nowait]
```

# Parallele Abschnitte...

---

- ▶ Die Anzahl der gewählten Threads bearbeitet unabhängig die Abschnitte
- ▶ Jeder Abschnitt genau einmal durchlaufen
- ▶ Nicht bestimmbar, welcher Thread welchen Abschnitt bearbeitet
- ▶ Nicht bestimmbar, wann welcher Abschnitt an die Reihe kommt
- ▶ Deshalb: Ausgabe eines Abschnittes sollte nicht Eingabe für einen anderen sein

# Sequentielle Abschnitte

---

Parallele Abarbeitung manchmal zeitweilig nicht erwünscht

```
!$omp single [clause [,] [clause...]]
  Anweisungsblock der nur von einem
  Thread bearbeitet wird
!$omp end single [nowait]
```

Keine Barriere zu Beginn des sequentiellen Abschnitts

Mittels **nowait** warten Threads nicht auf das Ende des sequentiellen Abschnitts

# Sequentielle Abschnitte...

Beispiel: Ausgabe

```
!$omp parallel shared (out,len)
    ...
 !$omp single
     call write_array(out,len)
 !$omp end single nowait
    ...
 !$omp end parallel
```

# Dynamische Threads

---

In Mehrbenutzerumgebungen Threadanzahl nicht optimal einstellbar

- ▶ Zu viele Threads: Verluste durch Umschalten
- ▶ Zu wenige Threads: Nicht genutzte Ressourcen

OpenMP bietet dynamische Anpassung der Threadanzahl durch Laufzeitumgebung  
Umgebungsvariable **OMP\_DYNAMIC**

# Ereignissynchronisierung

---

Barrieren: alle Threads warten an der Barriere, bis alle parallelen Threads eingetroffen sind – dann erfolgt die Fortsetzung der Arbeit

**`!$omp barrier`**

Ordnung: erzwingt ein Durchlaufen von Anweisungen in der ursprünglichen Reihenfolge der Indexwerte (d.h. wie in einem sequentiellen Programm)

**`!$omp ordered`**

**`block`**

**`!$omp end ordered`**

# Bibliotheken

---

## Zusammenbinden von OpenMP-Programmen mit Bibliotheken von Dritten

- ▶ Es muß sichergestellt sein, daß die Bibliotheksaufrufe *thread-sicher* sind
- ▶ Andernfalls Bibliothek nicht in parallelen Bereichen verwenden
- ▶ Oder z.B. als kritischen Bereich kennzeichnen
  
- ▶ Heutzutage sind allerdings die meisten Bibliotheken schon *thread-sicher*

# Compiler

Früher spezielle Präprozessoren und Compiler

Heute: in alle gängigen Compiler integriert

Beispiel gcc (ab Version 4.2):

- ▶ Option -fopenmp

# Vergleich der Ansätze

|                             | Pthreads | OpenMP     | MPI        |
|-----------------------------|----------|------------|------------|
| Skalierbarkeit              | Begrenzt | Begrenzt   | Ja         |
| Fortran / C und C++         | Ja? / Ja | Ja / Ja    | Ja / Ja    |
| Hohe Abstraktion            | Nein     | Ja         | Nein       |
| Leistungsorientierung       | Nein     | Ja         | Ja         |
| Portierbarkeit              | Ja       | Ja         | Ja         |
| Herstellerunterstützung     | Unix/SMP | Verbreitet | Verbreitet |
| Inkrement. Parallelisierung | Nein     | Ja         | Nein       |

# Programmierung mit OpenMP

## Zusammenfassung

- ▶ OpenMP wird ausschließlich für Architekturen mit gemeinsamem Speicher verwendet
- ▶ OpenMP ist ein Ansatz, der auf Compiler-Direktiven aufbaut
- ▶ OpenMP-Programme mit regulärem Compiler problemlos übersetzbbar
- ▶ Konstrukte zur Parallelisierung von Schleifen (feingranular) und anderen Code-Bereichen (grobgranular)
- ▶ Konstrukte zur Kontrolle der Variableninstanzen in den Threads
- ▶ Konstrukte zur Instanziierung von Threads und zu deren Beendigung
- ▶ Konstrukte zur Synchronisation der Threads untereinander
- ▶ OpenMP bildet mit MPI den Standard der parallelen Programmierung für alle modernen Maschinen

# Optimierung sequentieller Programme

---

- ▶ Motivation und Ansätze
- ▶ Ebenen und Potentiale der Optimierung
- ▶ Anwendungsklassen
- ▶ Programmiersprachen
- ▶ Theoretische Leistungsabschätzung
- ▶ Praktische Leistungsabschätzung
- ▶ Optimierung der Mathematik
- ▶ Optimierung der Programmierung
- ▶ Optimierung mit dem Compiler
- ▶ Fazit

# Optimierung sequentieller Programme

## Die zehn wichtigsten Fragen

- ▶ In welchen Fällen versuche ich, die Leistung zu steigern?
- ▶ Auf welchen Ebenen setzt eine Optimierung an?
- ▶ Wie sind hier die Optimierungspotentiale?
- ▶ Welche Kenntnisse muß ich haben?
- ▶ Wie schätze ich die theoretische Leistungsfähigkeit ab?
- ▶ Wie schätze ich die praktische Leistungsfähigkeit ab?
- ▶ Wie optimiere ich die Mathematik?
- ▶ Wie optimiere ich auf der Programmebene?
- ▶ Wie nutze ich die Compileroptimierungen?
- ▶ Was muß ich alles lernen?



Programmierung ist eine Krücke, die wir nur benutzen,  
weil wir noch nicht weit genug sind, mathematische  
Darstellungen direkt in Ergebnisse zu transformieren

Mathematik  
Numerik  
Programm  
Ergebnisdaten



# **Teil 1: Allgemeine Betrachtungen**

# Motivation

---

## „Gefühlte Programmgeschwindigkeit“

- ▶ Wichtig für den eigenen Arbeitsablauf
- ▶ Stark situationsabhängig
- ▶ Psychologische Effekte beachten

## Beispiele

- ▶ Reaktionszeit beim Anklicken eines Knopfes
  - ▶ Nach max. 2 Sekunden sollte eine Rückmeldung kommen
- ▶ Ausführungszeit der angeklickten Operation
  - ▶ Beliebig, aber durch Benutzererwartungen beschränkt
  - ▶ Vorhersagbarkeit wäre gut
  - ▶ Falls nicht das, dann wenigstens Fortschrittsanzeige

# Motivation...

---

## Programme mit GUI (Geschäftsprogramme)

- ▶ Schnelle Reaktion der GUI
- ▶ Beliebige Reaktion des Programms

## Programme auf Kommandozeile (technisch/wissensch.)

- ▶ Meist wünscht man sich kürzere Programmlaufzeiten

## Programme in Maschinensteuerungen

- ▶ Vorgegebene maximale Laufzeit (Echtzeitprogramme)  
Z.B. Auslösung eines Airbags

# Ansätze

---

## Wir warten auf schnellere Hardware

- ▶ Ging von 1941 – ca. 2005
  - Permanente Leistungssteigerung der Prozessoren unter Beibehaltung des Nutzungskonzepts
    - ▶ Compiler erzeugt Programm für den Prozessor (8bit – 64bit)
- ▶ Seit ca. 2005 Mehrkernprozessoren auch im PC
  - D.h. weitere Leistungssteigerungen nur noch durch manuelles Parallelisieren des Codes
    - ▶ Leider kann der Compiler das nicht

## Wir parallelisieren den Code

- ▶ Konzept seit den 1970ern genutzt
- ▶ Seit den 1990ern Rechnercluster, dann auch mit Linux
- ▶ Wird später besprochen

# Ebenen der Optimierung

---

## Mathematisch/algorithmische Ebene

- ▶ Welche alternativen mathematischen Verfahren sind in der Ausführung schneller?

## Programmiersprachliche Ebene

- ▶ Geeignete Verfahren
  - ▶ Welcher Algorithmus läuft am besten?
  - ▶ Welche Datenstrukturen sind am geeignetsten?
- ▶ Optimale Anpassung an die Architektur
  - ▶ Wieviel Hauptspeicher hat mein Rechner?
- ▶ Optimale Anpassung an den Compiler
  - ▶ Wie legt der Compiler die Daten im Speicher ab?

# Ebenen der Optimierung...

---

## Compilerebene

- ▶ Welche Optimierungen führt der Compiler durch?
- ▶ Wie kann ich Optimierungen gezielt auswählen?

## Hardware-Ebene

- ▶ Kann ich meinen Rechner an das Problem anpassen?
  - ▶ Z.B. Einbau von mehr Hauptspeicher
  - ▶ Z.B. Einbau von speziellen Beschleunigerkarten (GPGPU, FPGA)

# Benötigte Fachkenntnisse

---

- ▶ Wissen über die Anwendung
- ▶ Wissen zu mathematischen Verfahren
- ▶ Wissen zu Programmiertechniken
- ▶ Wissen über Compilerkonzepte
- ▶ Wissen über Rechnersysteme
- + **Wissen über das Zusammenwirken aller fünf Ebenen**

Wunschdenken des Naturwissenschaftlers

Mich kümmert nur meine Naturwissenschaft!

- ▶ Geht klar, aber dann wird das Werkzeug Computer nicht optimal eingebunden werden können

Disziplinabhängige Unterschiede: Physiker vs. alle anderen

# Optimierungspotentiale

---

## Mathematik

- ▶ Sehr hoch
  - ▶ Komplexitätsverringerung der Verfahren bringt Größenordnungen

## Programmiertechnik

- ▶ Sehr hoch
  - ▶ Komplexitätsverringerung der Algorithmen bringt Größenordnungen
  - ▶ Effiziente Datenstrukturen bringt vielleicht noch eine Größenordnung
  - ▶ Optimale Anpassung an eingesetzte Hardware bringt vielleicht auch noch eine Größenordnung

## Compiler

- ▶ Mittel
  - ▶ Optimierungen des Maschinencodes werden durchgeführt

# Optimierungspotentiale...

---

## Hardware-Umbauten im normalen Rechner

- ▶ Mittel bis hoch, aber schwierig in der Umsetzung

## Hardware-Umbau: Erwerb eines Hochleistungsrechners

- ▶ Sehr hoch: Faktor 10 bis 100.000
- ▶ Schwierig in der Realisierung

## Wahl einer optimalen Programmiersprache

- ▶ Gering
- ▶ Komfort vs. Geschwindigkeit
- ▶ Mathematische und programmiertechnische Optimierungen mit jeder Sprache möglich
- ▶ Pfusch ebenso!

# Anwendungsklassen

---

## Geschäftssoftware (als Gegenbeispiel)

- ▶ Oft nicht zeitkritisch in der Ausführung, weil eher kurz
- ▶ Ggf. Einmaloptimierung und dann langer Produktionsbetrieb

## Wissenschaftliche Software

- ▶ Typischerweise oft zeitkritisch, weil komplexe Berechnungen
- ▶ Probleme
  - ▶ Ständiger Wandel des Codes, der z.B. ein mathematisches Modell realisiert
  - ▶ Wenig Produktionsbetrieb mit unverändertem Code
  - ▶ Wissenschaftler hat keine Zeit zur Codeoptimierung
  - ▶ Wissenschaftler hat keine Kenntnis über Möglichkeiten

# (Traurige) Tatsachen

---

## Kein systematisches Leistungs-Engineering

- ▶ Kenntnisse über Optimierungen auf allen Ebenen sind nur bruchstückhaft bei den Anwendern vorhanden
- ▶ Keine Lehre zu diesem Thema
- ▶ Nahezu keine schriftlichen Unterlagen
- ▶ Niemand weiß, wie schnell das Programm sein müßte

## Ausnutzung der nominellen Prozessorleistung gering

- ▶ In vielen Fällen werden nur 5-10% der **Rechenleistung** genutzt
- ▶ Gründe beispielhaft:
  - ▶ Sprünge im Code, nicht genügend Mathematik im Code

# (Traurige) Tatsachen...

Wissenschaftliche Software meist schlecht optimiert

- ▶ Ergebnisse kommen zu langsam
- ▶ Ressourcen werden nicht optimal genutzt
  - ▶ Klimacodes für IPCC AR5 brauchen am DKRZ 30 MCPUh und das kostet 1 MEuro für Strom

Energiekosten sind jetzt ein wichtiger Faktor

- ▶ Nicht mehr alleine interessant: time-to-solution
- Sondern auch: kWh-to-solution

# Kosten/Nutzen-Analyse

---

## Kosten

- ▶ Einführung neuer Mathematik
- ▶ Verbesserung der Programmstruktur
- ▶ Installation optimierter Bibliotheken

## Nutzen

- ▶ Verkürzte Programmlaufzeit

## Sinnvolles Vorgehen

- ▶ Aufgewandte Zeit und gesparte Zeit in Relation setzen
- ▶ Unterm Strich sollte eine Ersparnis herauskommen
  - ▶ Aufwand für Optimierung an das Einsparpotential anpassen

# Die Wahl der Programmiersprache

---

C

- ▶ Gute Anpassung an Hardware möglich
- ▶ Programmierung vergleichsweise maschinennah
- ▶ Effizienter Maschinencode

C++

- ▶ Vorteile/Nachteile von C
- ▶ Zusätzliche objektorientierte Programmierung

Fortran

- ▶ Gute Anpassung an Mathematik
- ▶ Programmierung nicht so maschinennah wie C
- ▶ Trotzdem effizienter Maschinecode

# Die Wahl der Programmiersprache...

---

## Java

- ▶ Gute Programmierkonzepte und hoher Programmierkomfort
- ▶ Keine optimale Anpassung an die Hardware
- ▶ Keine optimale Anpassung an die Mathematik
- ▶ Einigermaßen effizienter Maschinencode

## Skriptsprachen / Matlab etc.

- ▶ Schnelle Programmerstellung möglich
- ▶ Komfort geht auf Kosten der Laufzeitoptimierung

## Zusammenfassung

- ▶ Leistungsausbeute bei Sprachen hängt eher vom Vermögen des Programmierers ab
- ▶ Objektorientierung kostet Leistung und bringt Komfort

# Ideale Vorgehensweise

---

- ▶ Sauberer Entwurf der Mathematik und der Implementierung
  - ▶ Nicht nur wegen Laufzeit sondern auch wegen
    - ▶ Fehlerfreiheit, Wartbarkeit, Erweiterbarkeit
- ▶ Messen der Programmlaufzeiten
- ▶ Bewerten
  - ▶ Kann ich mit dieser Laufzeit meine wissenschaftliche Arbeit durchführen?
- ▶ Aufdecken von Leistungsengpässen
  - ▶ Welche Werkzeuge gibt es?
- ▶ Beseitigung von Leistungsengpässen
  - ▶ Die wichtigsten zuerst



## Teil 2: Handwerkszeug

# Beurteilung der Leistungsfähigkeit

---

## Theoretisch

- ▶ Komplexitätsmaße für Zeit- und Speicherbedarf ermitteln  
Sehr schwierig – am besten ggf. Literatur heranziehen

## Praktisch

- ▶ Laufzeiten und Speichernutzungen messen  
Das kann jeder

# Theoretische Leistungsabschätzung

---

Lernt der Informatiker in der Komplexitätstheorie

In aller Kürze:

- ▶ Zeitbedarf und Speicherbedarf haben eine funktionale Abhängigkeit von der Anzahl und Größe der Eingabedaten
- ▶ Bezeichnet durch  $O(X)$ , wobei X eine Funktion von n ist

Beispiele (für Laufzeitkomplexität)

- ▶  $O(n)$ : Das Programm ist linear von n abhängig
  - ▶ Beispiel: Durchlaufen aller Eingabewerte und Maximum finden
- ▶  $O(n^2)$ : Das Programm ist quadratisch von n abhängig
  - ▶ Beispiel: Schlechte Sortierverfahren, die alle Werte mit allen vergleichen

# Theoretische Leistungsabschätzung...

## Auszug aus Sortierverfahren bei Wikipedia

| Sortierverfahren                               | Best-Case  | Average-Case   | Worst-Case   |
|--|--|--|--|
| AVL Tree Sort<br>(höhen-balanciert)            | $\mathcal{O}(n)$   | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n \cdot \log(n))$   |
| Binary Tree Sort                               | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n^2)$   |
| Bogosort                                       | $\mathcal{O}(n)$   | $\mathcal{O}(n \cdot n!)$  | $\infty$   |
| Bubblesort<br>(Vergleiche)<br>(Kopieraktionen) | $\mathcal{O} \left( \begin{matrix} n \\ (n-1) \\ (0) \end{matrix} \right)$ | $\mathcal{O} \left( \begin{matrix} n^2 \\ \frac{n^2}{4} \\ \frac{n^2}{4} \end{matrix} \right)$ | $\mathcal{O} \left( \begin{matrix} n^2 \\ \frac{n^2}{2} \\ \frac{n^2}{2} \end{matrix} \right)$ |
| Combsort                                       | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n^2)$   |
| Gnomesort                                      | $\mathcal{O}(n)$   |  | $\mathcal{O}(n^2)$   |
| Heapsort                                       | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n \cdot \log(n))$   | $\mathcal{O}(n \cdot \log(n))$   |

Wichtig: welches ist die minimale Komplexität?

# Theoretische Leistungsabschätzung...

## Weitere Beispiele

- ▶  $O(1)$ : Die Laufzeit ist fest und hängt nicht von  $n$  ab
  - ▶ Beispiel: Ein Programm, das immer gleich abstürzt ☺
- ▶  $O(\log n)$ : Die Laufzeit hängt logarithmisch von  $n$  ab
  - ▶ Beispiel: Suche in einem Binärbaum
- ▶  $O(n^k)$ : Die Laufzeit hängt polynomial von  $n$  ab
  - ▶ Beispiele: kaum welche mit Praxisrelevanz für  $k > 2$
- ▶  $O(2^n)$ : Die Laufzeit hängt exponentiell von  $n$  ab
  - ▶ Beispiele: viele theoretische, die praktisch nicht berechnet werden können

## Beachte

- ▶ Bei sehr kleinen  $n$  ist manchmal auch eine höhere Komplexität noch akzeptabel (z.B.  $O(n^2)$  beim Sortieren statt  $O(n \log n)$ )
- ▶ Die Bestimmung der Komplexität ist sehr komplex ☺

# Praktische Leistungsabschätzung

---

Es gibt verschiedene Meßwerkzeuge

- ▶ Wenige für sequentielle Programme
- ▶ Einige komplexe für parallele Programme

Unter Linux

- ▶ Kommandos **time** (der Shell) und **/usr/bin/time**
  - ▶ Letzteres zeigt auch den Speicherverbrauch und andere Daten
- ▶ Einfach auf der Kommandozeile dem Programmaufruf voranstellen
  - ▶ Ermittelt die Gesamtlaufzeit des Programms
- ▶ Kommando **gprof**
  - ▶ Programm mit Compileroption für Profiling übersetzen (gcc: -pg)
  - ▶ Laufenlassen des Programms erzeugt Datei gmon.out
  - ▶ Kann mit gprof angesehen werden

# Praktische Leistungsabschätzung...

Beispiel (Hager/Wellein):

| %     | cummulative | self    |          | self    | total   |           |
|-------|-------------|---------|----------|---------|---------|-----------|
| Time  | seconds     | seconds | calls    | ms/call | ms/call | name      |
| 70.45 | 5.14        | 5.14    | 26074562 | 0.00    | 0.00    | intersect |
| 26.01 | 7.03        | 1.90    | 4000000  | 0.00    | 0.00    | shade     |
| 3.72  | 7.30        | 0.27    | 100      | 2.71    | 73.03   | calc_tile |

## Erläuterung

- ▶ self seconds ist die Laufzeit in der Funktion
- ▶ cummulative seconds ist die aufsummierte Laufzeit, wenn nach self seconds sortiert wird

# Praktische Leistungsabschätzung...

## Vorgehensweise

- ▶ Wir optimieren die Funktionen mit dem höchsten Zeitanteil
  - ▶ Hier zunächst intersect
- ▶ Eine Abschätzung des Optimierungspotentials der einzelnen Funktionen gibt Aufschluß über das Gesamtpotential

## Aspekte von gprof

- ▶ Funktionsbasiert – d.h., wer sein Programm nicht in Funktionen unterteilt, kann nichts messen ☺
- ▶ Inlining von Funktionen durch den Compiler muß korrekt behandelt werden, sonst sind die Meßwerte falsch
  - ▶ Inlining: der Compiler ersetzt im Maschinencode einen Funktionsaufruf durch die Funktion selber

# Optimierung der Mathematik

---

- ▶ Am besten zusammen mit den Mathematikern
  - ▶ Kooperationen mit Numerikern/Optimierern
- ▶ Bessere mathematische Verfahren brauchen Zeit für die Entwicklung und Evaluation
- ▶ Kann nicht vom Naturwissenschaftler geleistet werden
  - ▶ Muß aber in Zusammenarbeit mit ihm erfolgen, da meist die Kenntnis der Anwendung von Nöten ist

# Optimierung der Mathematik...

---

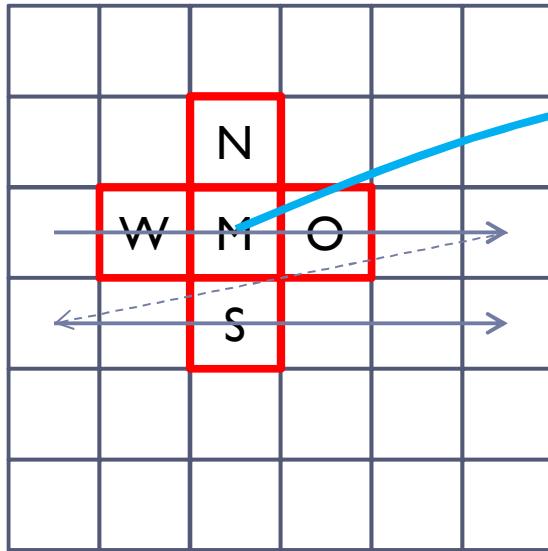
Beispiel: partielle Differentialgleichungen mittels Jacobi- oder Gauß/Seidel-Verfahren

- ▶ Löst ein lineares Gleichungssystem
- ▶ Z.B. für folgende Anwendung: wir erwärmen eine Platte an den Ecken auf eine bestimmte Temperatur – wie ist dann die Verteilung der Temperatur über die Platte hinweg?

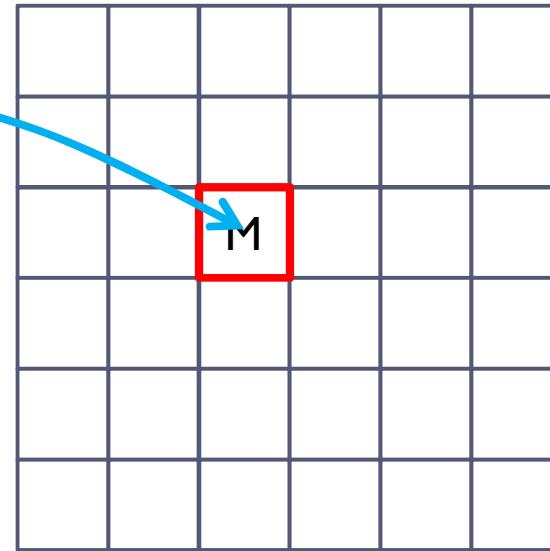
Bewertung:

- ▶ Gauß-Seidel-Verfahren konvergiert schneller
- ▶ Seit neuestem aber: Jacobi lässt sich für hohe Anzahl von Prozessoren besser parallelisieren

# Optimierung der Mathematik...

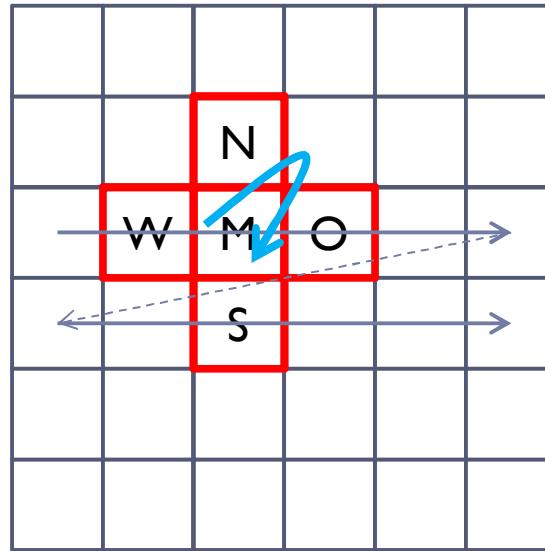


Jacobi



- ▶ Es werden zwei Matrizen verwendet: eine mit den aktuellen Werten und eine für die Werte der nächsten Iteration
- ▶ Der neue Wert M wird aus den alten Nachbarwerten von M (N, W, S und O) ermittelt
- ▶ Wenn alle neuen Werte bestimmt sind, werden die beiden Matrizen getauscht und die nächste Iteration beginnt
- ▶ Das Verfahren endet, wenn für alle neuen M die Änderung kleiner einer unteren Schranke ist

# Optimierung der Mathematik...



## Gauß-Seidel

- ▶ Nur eine Matrix verwandt, also weniger Speicher *und* auch schneller
- ▶ Der neue Wert M wird aus den Nachbarwerten von M (N, W, S und O) ermittelt
- ▶ Hier jetzt: N und W wurden bereits aktualisiert, S und O noch nicht. Das mathematische Verfahren läuft somit anders ab
- ▶ Das Verfahren endet, wenn für alle neuen M die Änderung kleiner einer unteren Schranke ist

# Optimierung der Programmierung

---

Am besten zusammen mit den Informatikern

Drei Ebenen

- ▶ Pure Programmierung ohne Berücksichtigung von Compiler und Hardware
- ▶ Programmoptimierung im Zusammenspiel mit dem Compiler
- ▶ Programmoptimierung im Zusammenspiel mit der Hardware

Allgemeine Probleme

- ▶ Bei einem Wechsel der Zielarchitektur müssen die Optimierungen erneut evaluiert und dann angepaßt oder ausgetauscht werden
- ▶ Dasselbe gilt bei einem Wechsel auf parallele Architekturen

# Optimierung der Programmierung...

---

Unabhängig von Compiler und Hardware

- ▶ Effiziente Algorithmen
  - ▶ Effizientes Sortieren, effizientes Suchen
- ▶ Effiziente Datenstrukturen
  - ▶ Listen, Bäume, Hashtabellen, dünn besetzte Matrizen

Findet man in Büchern und Vorlesungen

- ▶ Informatikergrundvorlesung „Algorithmen & Datenstrukturen“
  - ▶ Das Minimum dessen, was der Naturwissenschaftler wissen sollte!
- ▶ Amazon: „Algorithmen und Datenstrukturen“

# Optimierung der Programmierung...

---

## Beispiel: dünnbesetzte Matrizen

- ▶ NxN Einträge, aber nur 0,1% sind ungleich von null

## Speicherung

- ▶ Ablage in einer verketteten Liste (einfach oder doppelt) mit Angabe der x,y-Koordinate
- ▶ Zusätzlich noch ein Feld mit Zeigern auf z.B. jedes 1000ste Element

## Zugriff

- ▶ Einstieg an einem der Zeiger, Ablaufen der Liste bis zur gewünschten Koordinate

## Matrizenmultiplikation

- ▶ Wird jetzt ganz neu implementiert

# Optimierung der Programmierung...

## Abhängig vom Compiler

Beispiel:

- ▶ Abbildung von logischen Datenstrukturen in den Hauptspeicher
- ▶ Hier: zweidimensionale Felder
- ▶ Z.B. wird zeilenweise in den Hauptspeicher abgebildet
- ▶ Programm lese z.B. die Werte zeilenweise oder spaltenweise
  - ▶ Was passiert mit der Zugriffszeit?
- ▶ Man würde meinen: gar nichts – wäre da nicht der Cache
  - ▶ Der holt sich nicht nur den fehlenden Wert sondern noch mehrere andere

# Optimierung der Programmierung...

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

logische  
Datenstruktur

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|

Speicherabbild

|   |   |   |
|---|---|---|
| a | b | c |
| 1 | 2 | 3 |
| x | y | z |

Cache

Cacheinhalt  
nach Zugriff auf  
Element '1'

- ▶ Nach Zugriff auf '1' ist eine Cacheline geladen, gerade eben die Elemente '1', '2' und '3'
- ▶ Greift das Programm auf '2' und '3' zu, so geht dies schnell (Cache-Hits)
- ▶ Greift das Programm nach '1' auf '4' zu, so muß eine zweite Cacheline '4', '5' und '6' geladen werden
  - ▶ Das kostet Zeit! (Cache-Miss)
  - ▶ Dasselbe Problem wiederholt sich, wenn dann auf '7' zugegriffen wird

# Optimierung der Programmierung...

Abhängig von der Hardware

Beispiel 1:

- ▶ Wir haben 2 GB Hauptspeicher
- ▶ Das Programm hat aber viele GB virtuellen Adreßraum
- ▶ Daten, die nicht im Hauptspeicher gehalten werden können, werden auf die Platte ausgelagert (Swapping)
- ▶ Das kostet Zeit!
- ▶ Also: Datenstrukturen des Programms an freien Platz anpassen

Beispiel 2:

- ▶ Im Rechner ist eine zusätzliche Grafikkarte verbaut
- ▶ Wir könnten diese zur Beschleunigung von Berechnungen verwenden

Beispiel 3:

- ▶ Das Programm wurde für einen 32bit-Prozessor entwickelt
- ▶ Es soll jetzt auf einem 64-bit Prozessor laufen
- ▶ Im einfachsten Fall Neuübersetzung für den neuen Zielprozessor (der kommt nie vor ☺)

# Optimierung mit dem Compiler

---

Alle Compiler haben aufwendig Codeoptimierungen eingebaut

- ▶ Manche sind unabhängig vom Zielprozessor
- ▶ Manche sind genau auf Befehlssätze, Register usw. zugeschnitten

Der Programmierer kann per Optionen verschiedene Optimierungsstufen auswählen

- ▶ Weiß dann mehr oder weniger, was passiert. Eher weniger

# Optimierung mit dem Compiler...

## Optimierungsoptionen für den GNU-C-Compiler **gcc**

-O0

führe keine Optimierungen durch

-O1

der Compiler versucht Codegröße und Programmlaufzeit zu verringern,  
ohne die Übersetzungszeit wesentlich zu erhöhen

-O2

mehr Optimierungen aber kein Inlining, kein Loop Unrolling  
Code wird schneller, Übersetzungszeit steigt

-O3

Inlining wird auch aktiviert

-Os

Wie -O2, aber ohne Optimierungen, die den Code vergrößern

# Optimierung mit dem Compiler...

---

## Zwei Beispiele für Optimierungsverfahren

- ▶ Inlining von Funktionen
  - ▶ Der Sourcecode einer Funktion wird übersetzt und überall da direkt eingebaut, wo die Funktion aufgerufen wird
  - ▶ Zeitaufwendige Sprünge entfallen
  - ▶ Maschinencode wird länger
- ▶ Loop Unrolling
  - ▶ Wenn eine Schleife z.B. 10x durchlaufen wird, dann wird der Code 10x hintereinander abgelegt
  - ▶ Zeitaufwendige Sprünge entfallen
  - ▶ Maschinencode wird länger

# Optimierung mit dem Compiler...

## Wann wähle ich welche Stufe?

- ▶ Phase der Fehlersuche: immer mit  $-O0$ 
  - ▶ Ansonsten sind die Umbauten im Code für die Fehlersuche hinderlich, weil Code umgestellt, zum Teil eliminiert wird usw.
  - ▶ Eine eindeutige Zuordnung zu den Zeilen des Quellcodes ist dann nicht mehr möglich
- ▶ Phase des Profiling: ohne Funktionen-Inlining
  - ▶ Nicht alle Level sind mit der Funktionsweise des gewählten Profiler kompatibel
  - ▶ Im Einzelfall das Handbuch lesen
- ▶ Phase des Produktionsbetriebs z.B. mit  $-O3$ 
  - ▶ Manchmal treten aber Fehler auf, die aus einem komplexen Zusammenspiel zwischen maschinennaher Programmierung und Compileroptimierung entstehen
  - ▶ Dann den Optimierungslevel heruntersetzen



## **Teil 3: Zusammenfassung**

# Fazit

---

Es kann nur in Zusammenarbeit der Disziplinen eine Verbesserung erzielt werden

Anwendungswissenschaftler – Informatiker – Mathematiker

Viel Wissen für eine optimale Optimierung nötig

- ▶ Der Einzelne weiß dazu meist zu wenig
- ▶ Ist aber keine Entschuldigung, jegliches Wissen abzuweisen

Aufwandsabschätzung

- ▶ Was nützt es mir bei meiner Abschlußarbeit
- ▶ Was kostet mich das
- ▶ Was kostet es in der Folge Dritte

# To-Do-Liste

---

## Was muß ich wissen?

- ▶ Grundlagen der Komplexität bzgl. Zeit- und Speicherbedarf
- ▶ Wichtige Datenstrukturen und wichtige Algorithmen
- ▶ Kenntnis über das Ausmessen von Programmen
- ▶ Kenntnis über wichtige Aspekte der Compileroptimierung

## Was muß ich tun?

- ▶ Ein Buch zu „Algorithmen und Datenstrukturen“ besorgen und nach nützlichen Konzepten durchsehen
- ▶ **time** und **gprof** ausprobieren und einüben
- ▶ Programme regelmäßig bzgl. Ihrer Leistung analysieren und wichtige Einsichten aufschreiben
- ▶ Diskussion mit anderen Entwicklern über dieses Thema

# Optimierung sequentieller Programme

## Zusammenfassung

- ▶ Es gibt keine systematischen Ansätze zur Optimierung von sequentiellem Code
- ▶ Die Optimierungen finden auf der Ebene der Mathematik, der Programmierung und des Compilers statt
- ▶ Die Optimierungspotentiale sind da durchwegs sehr hoch
- ▶ Die konkrete Wahl der Programmiersprache ist weniger wichtig (solange es eine Übersetzer-Sprache ist)
- ▶ Mit der Komplexitätstheorie schätzt man Laufzeiten und Speicherbedürfnisse von Algorithmen ab
- ▶ Zur konkreten Programmanalyse gibt es bei Linux verschiedene Werkzeuge
- ▶ Optimierungen der Mathematik können die Laufzeit deutlich verbessern
- ▶ Optimierungen bei der Programmierung können die Laufzeit deutlich verbessern
- ▶ Optimierungen mit dem Compiler können die Laufzeit deutlich verbessern

# Leistungsmodellierung

---

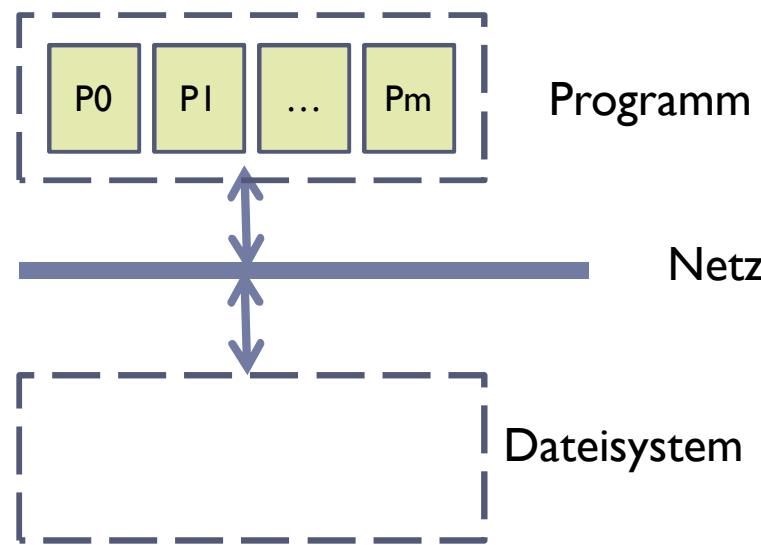
- ▶ Leistungscharakteristika
- ▶ Prozessor-, Knoten-, Cluster-Leistungsfähigkeit
- ▶ E/A-Leistungsfähigkeit
- ▶ Praktische Anwendung
  - ▶ Prozessplatzierung
  - ▶ Programmanalyse/Optimierung
  - ▶ Vorgehensweise
  - ▶ Typische Ursachen

# Leistungsmodellierung

## Die 6 wichtigsten Fragen

- ▶ Welche Leistung können wir von unserem Programm auf unserem Supercomputer erwarten?
- ▶ Welche Leistung stellt das System bereit?
- ▶ Wie identifizieren wir den Engpass?
- ▶ Welche Ursachen könnte die geringe Leistung haben?
- ▶ Wie gut passt gemessene Leistung zur erwarteten?
- ▶ Wie platzieren wir die Prozesse bestmöglich auf die Knoten?

# Logische (Prozess)-Sicht



- ▶ Batch-Scheduling:
  - ▶ Nutzer fordert n Knoten und p Prozessoren an
- ▶ Programm-Sicht:
  - ▶ Verteilung auf physikalische Geräte unbekannt
- ▶ Ausnutzung der Ressourcen ist wichtig
  - ▶ Bestmögliche Verteilung?
  - ▶ Aufdeckung eines Engpasses?
  - ⇒ Leistungsmodell nötig!

# Leistungscharakteristika

---

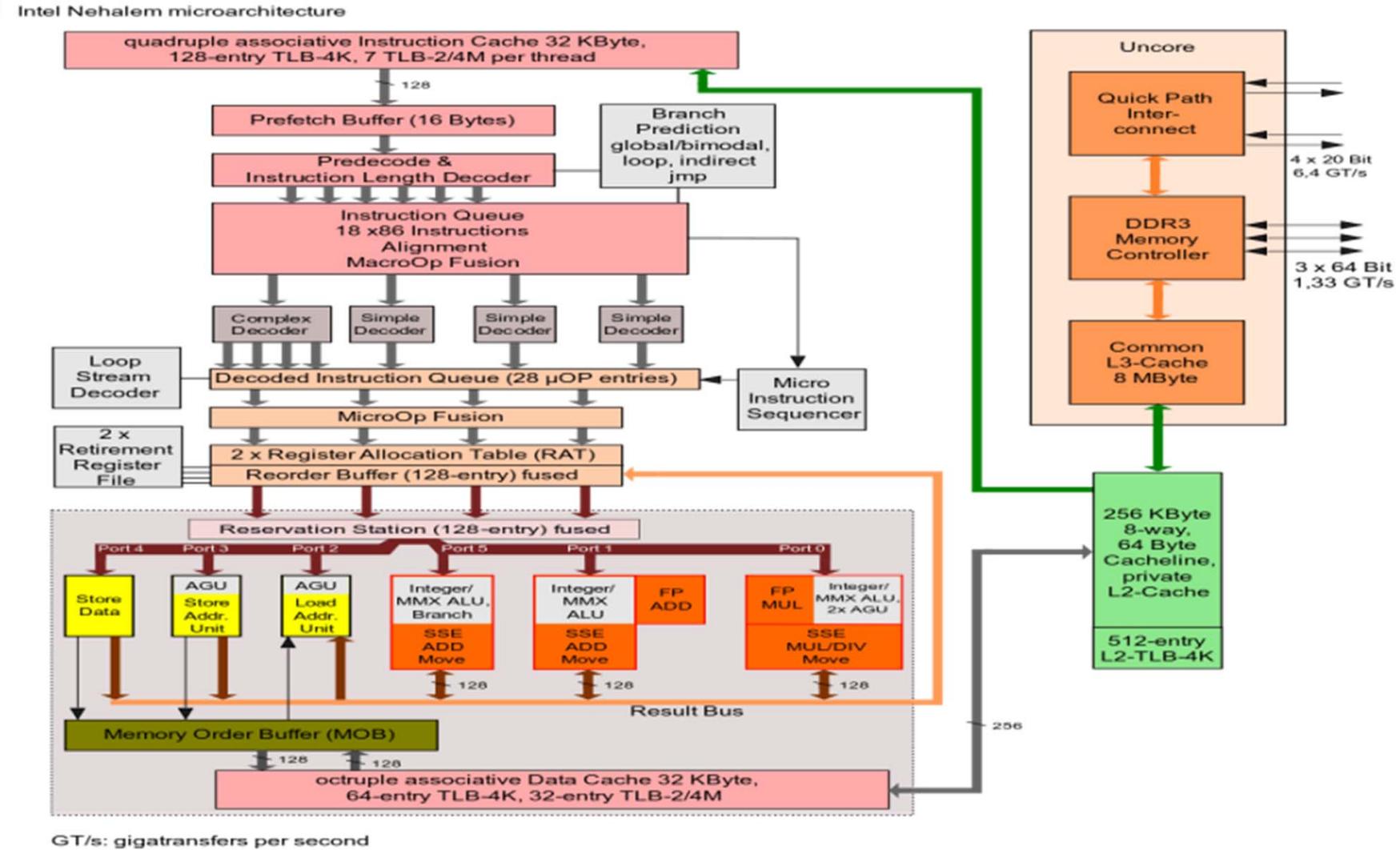
- ▶ Einzelne Komponenten sehr komplex
- ▶ Ohne Kenntnis des Systems suboptimale Leistung
  - ▶ Aber bis zu welchem Detailgrad brauchen wir sie?
- ▶ Daher hier einfaches Modell für Kerncharakteristika
  - ▶ Modell bildet Realität nicht exakt ab
  - ▶ Aber hinreichend gut um Resultate zu bewerten
  - ▶ Ausblick auf komplexere Zusammenhänge
- ▶ Viele Probleme können so identifiziert werden
  - ▶ Systematische Identifikation des Engpasses

# Woher kommen die Modell-Referenzwerte?

---

- ▶ Herstellerangaben
  - ▶ Oft optimistisch
- ▶ Benchmarks zum Ermitteln der erzielbaren Leistung
  - ▶ Wie messen wir ein Charakteristikum des Systems?
  - ▶ Siehe Kapitel Leistungsmessung
- ▶ Vergleich mit bestehenden Systemen

# Beispiel: Prozessorarchitektur – Intel Nehalem

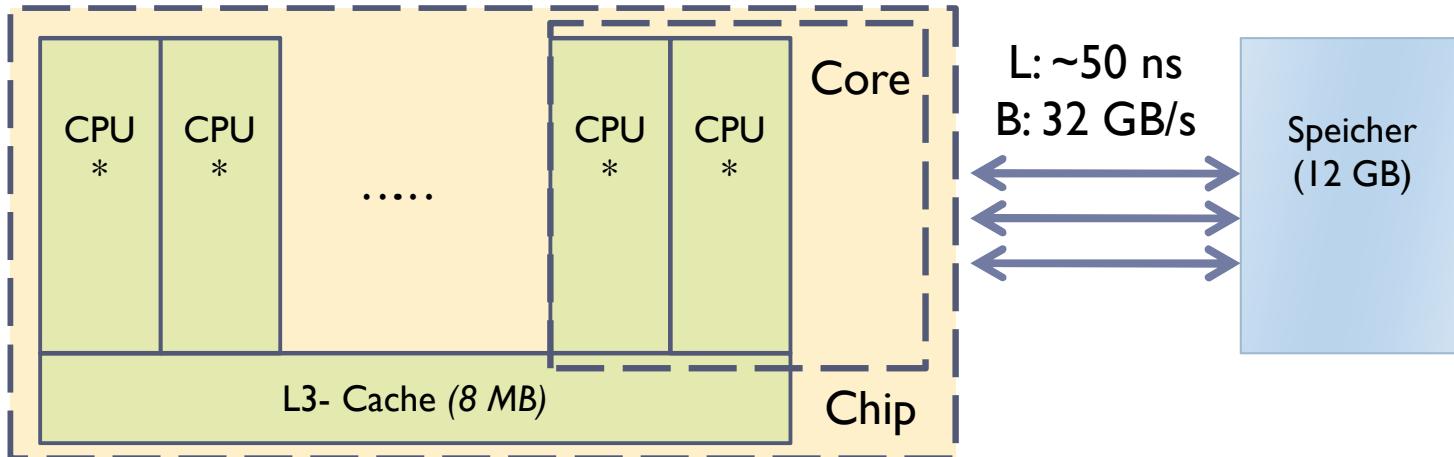


# Relevante Leistungscharakteristika

---

- ▶ Prozessorleistungsfähigkeit
  - ▶ Instruktionen/Sekunde, Simultaneous Multithreading (SMT)?
  - ▶ Größe der L1-, L2-, L3-Caches
- ▶ Speicheranbindung
  - ▶ Topologie: Einzelter Bus, Bus/Chip (z. B. Nehalem), Interconnect
  - ▶ Latenz und Bandbreite
- ▶ E/A-Leistungsfähigkeit
  - ▶ Bandbreite (pro Knoten und Server)
  - ▶ IOPS – Anzahl der E/A-Operationen pro Sekunde
- ▶ Netzwerkleistungsfähigkeit
  - ▶ Latenz und Bandbreite
  - ▶ Topologie

# Physikalische Sicht – Prozessor (mit SMT)



*CPU kann pro Takt z. B. 2 FLOP ausführen*

*Pro Core:*

*LI: 32K Instruktion/32K Daten*

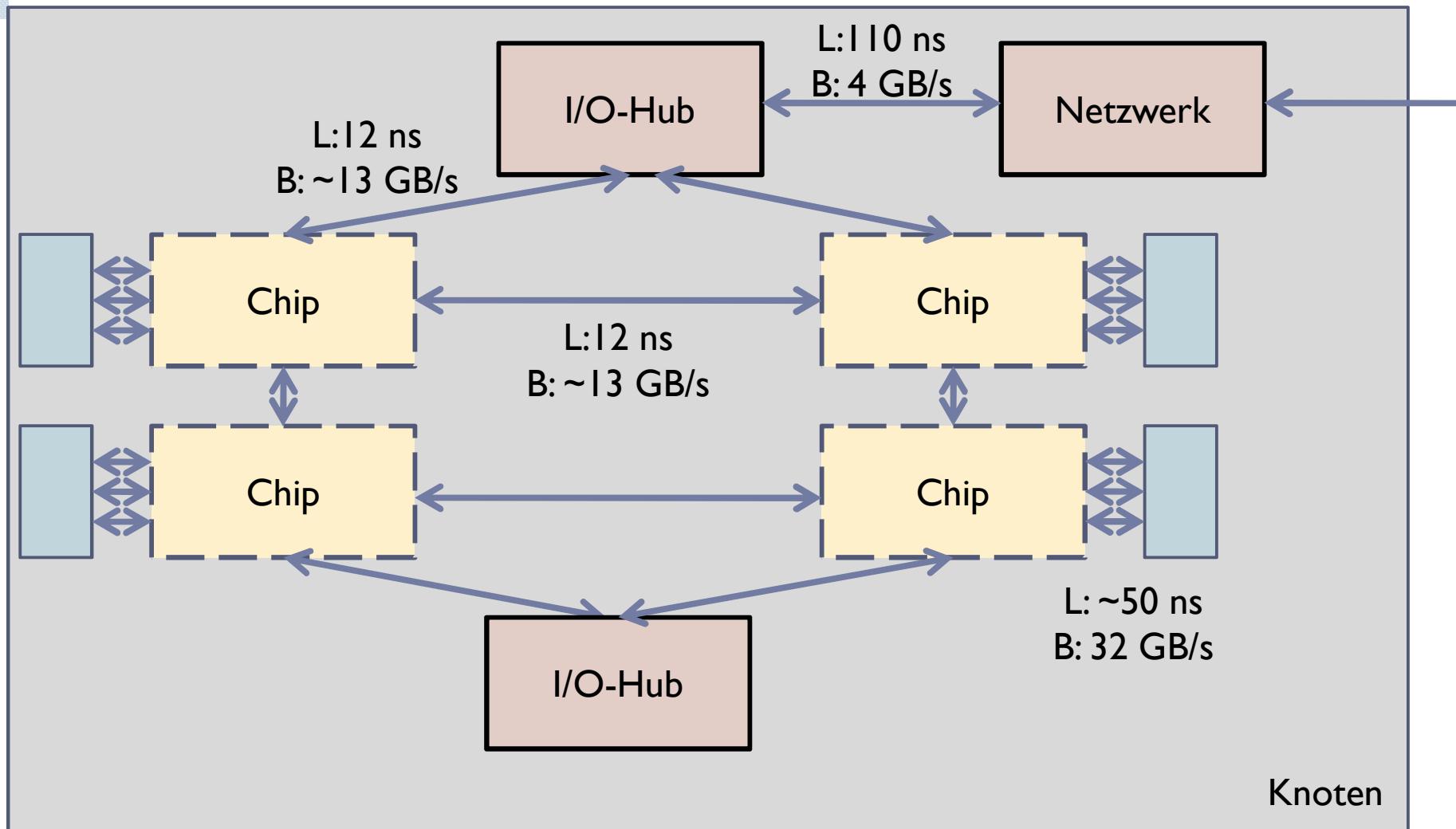
*L2: 256K*

*Latenz:*

*4 Zyklen*

*10 Zyklen*

# Physikalische Sicht – Mehrprozessor-Knoten

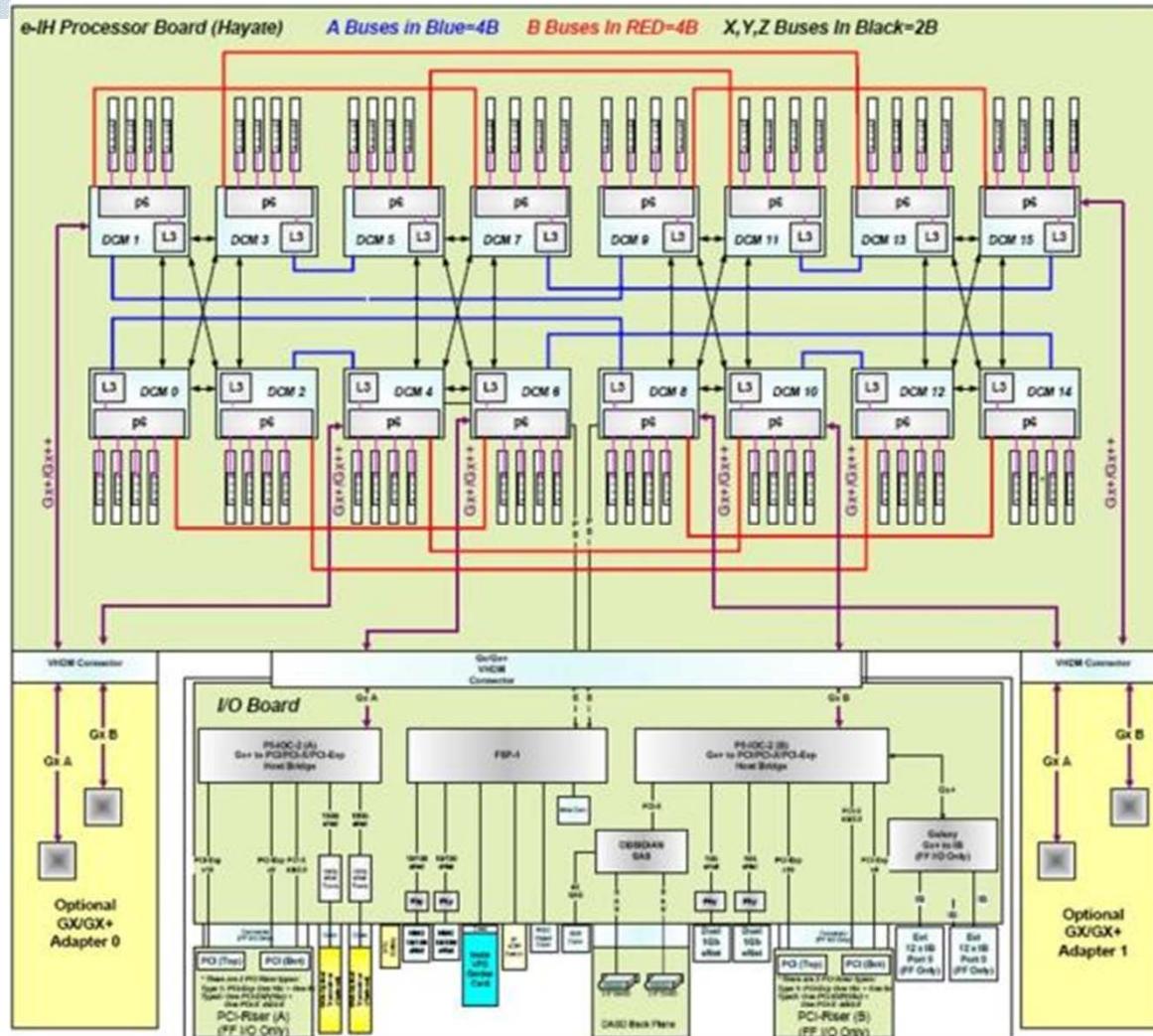


# Beobachtung

---

- ▶ NUMA-Charakteristik
  - ▶ Speicherorganisation beachten
  - ▶ Prozessmigration zwischen CPUs bzw. CPU-Pinning
- ▶ Netzwerkeffizienz – oben beiden Chips benutzen
- ▶ Caches verschiedener Größe und Latenzen
- ▶ Netzwerk kann u. U. nur von mehreren Prozessoren saturiert werden

# Praxisbeispiel: IBM-Power6-Server des DKRZ



16 Node-Cards a 2 Cores  
Power6 mit 4.7 GHz

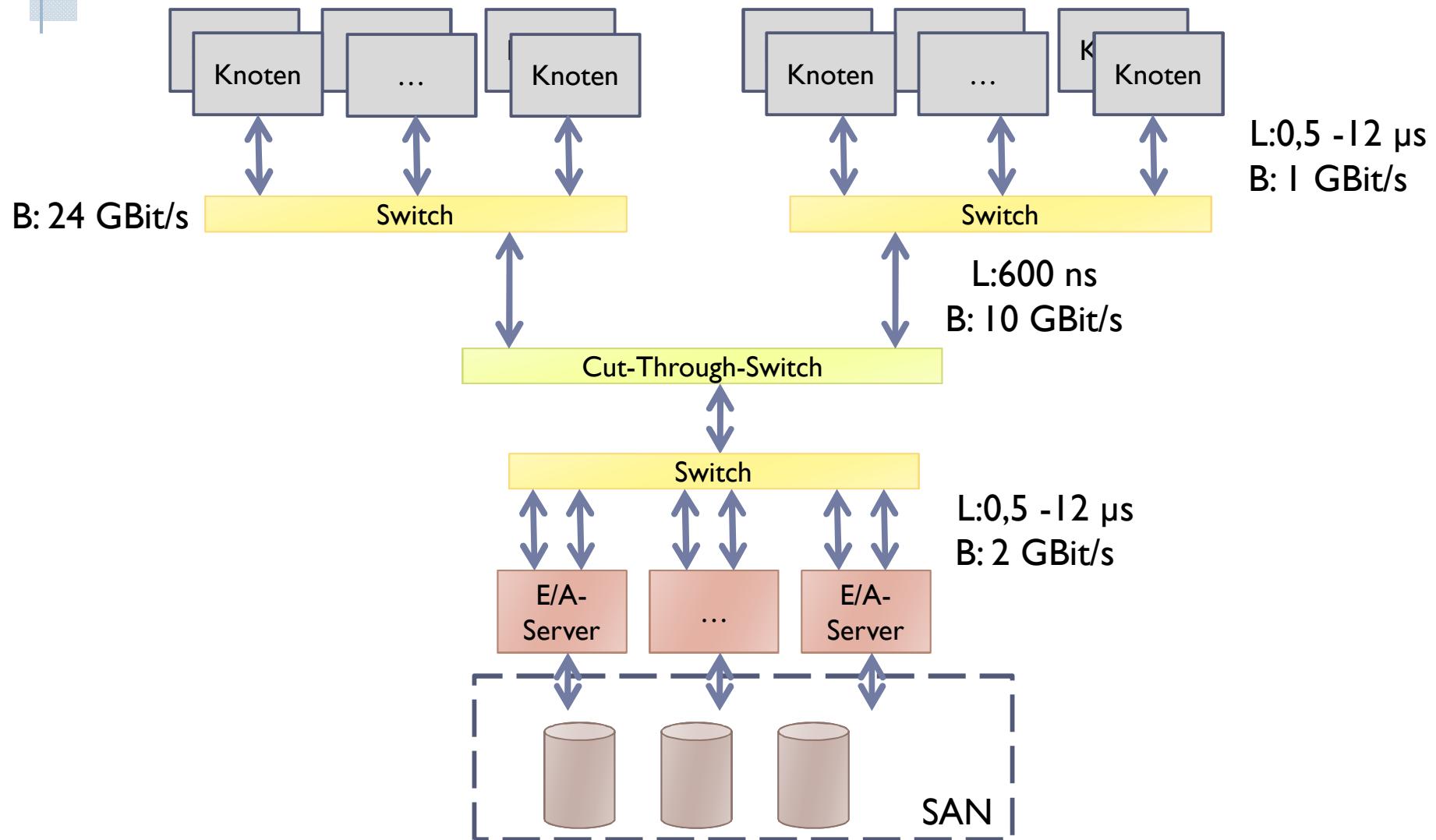
256 Knoten  
Infiniband-Vernetzung

# Betriebssystemeinflüsse

---

- ▶ Hintergrundaktivität erzeugt Rauschen (Jitter)
  - ▶ Verarbeitung von Unterbrechungen
  - ▶ Zugeteilte Zeitscheibe z. B. 10ms
  - ▶ Moduswechsel (Betriebssystemaufrufe)
    - Können zu Prozesswechsel führen
- ▶ Prozessumschaltung kostet Zeit
  - ▶ Bspw. 4.2 µs – bei Leeren des L2 Caches z. B. 200 µs
- ▶ Kommunikationslatenz zwischen zwei Knoten ist protokollabhängig!
  - ▶ TCP/IP typischerweise 100 µs
  - ▶ SCS-MPI-Bibliothek z. B. ~ 4 µs
- ▶ Seiten-Ein-/Auslagerungsalgorithmen (Swapping)

# Physikalische Sicht – Cluster (mit Ethernet)



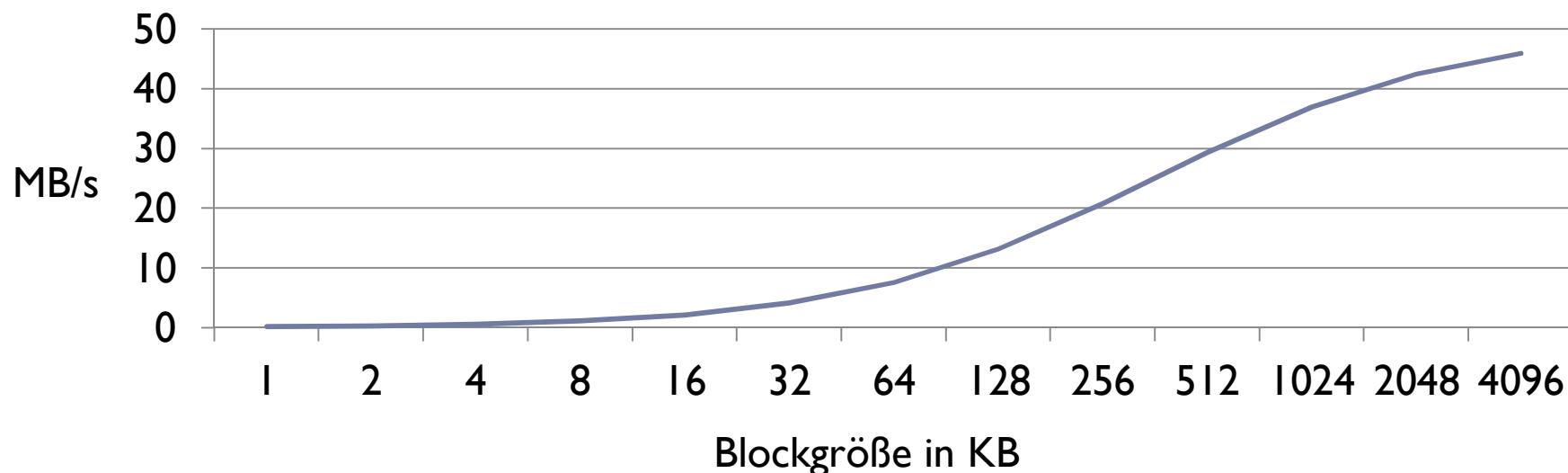
# E/A-Leistung

---

- ▶ Zugriffsmuster der Anwendung entscheidend
  - ▶ Zeitliches und örtliches Zugriffsmuster
- ▶ E/A meist um Größenordnung langsamer als Netzwerk
- ▶ Caching von Daten auf vielen Ebenen
  - ▶ Betriebssystem der Knoten (durch Arbeitsspeicher begrenzt)
  - ▶ Auf Servern des (parallelen) Dateisystems
  - ▶ Plattencache
- ▶ Optimierung in genutzten Bibliotheken:
  - ▶ HDF5 / NetCDF
  - ▶ MPI-I/O (Kollektive Operationen)
- ▶ RAID-Charakteristika

# Charakteristika einer Festplatte

- ▶ Mechanische Bauteile
  - ▶ Zugriffszeit abhängig von Position der Köpfe und Zugriffsort
  - ▶ Mittlere Zugriffszeit: 7 ms
  - ▶ Durchsatz: 50 MB/s
- ▶ Beispielelleistung für zufällige Zugriffe:



# Praktische Anwendung

---

- ▶ Platzierung der Prozesse auf CPUs
  - ▶ Leistungsentscheidend, wird oft falsch gemacht
- ▶ Optimierung/Analyse eines Programmes:
  - ▶ Wie nah ist das Programm an der Maximalleistung
  - ▶ Lohnt sich der Aufwand gegenüber des zu erwartenden Leistungsgewinns?
  - ▶ Wo ist der Engpass?
    - ▶ Typische Engpässe!

# Prozessplatzierung

---

- ▶ Platzierung der Prozesse auf Knoten und Prozessoren
  - ▶ Kenntnis des Programmverhaltens nötig!
  - ▶ Prozesskopplung beachten
    - ▶ Viel Kommunikation => Prozesse „nah“ zueinander platzieren
    - ▶ NUMA-Datenzugriff vermeiden (bei Shared Memory)
  - ▶ SMT evaluieren => typischerweise verwenden
  - ▶ Netzwerk: Kommunikation über Switchgrenzen vermeiden
  - ▶ Hinreichend Speicher pro Prozess verfügbar machen
- ▶ E/A-Anbindung ans Netzwerk aber nicht vergessen!
  - ▶ Typischerweise alle gleich angebunden

# Platzierungsbeispiel:

Fakten:

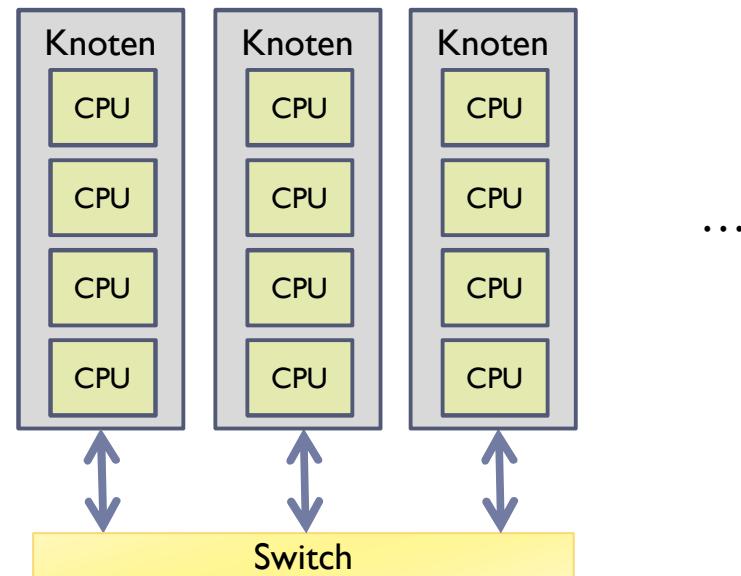
- Datenaufteilung
- 9 Prozesse

Eingabedaten:

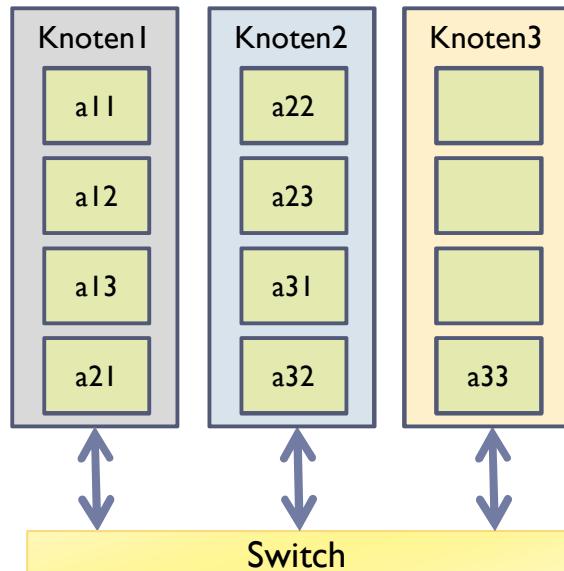
$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \hline a_{21} & a_{22} & a_{23} \\ \hline a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Austausch erfolgt über die  
Linien

Vorhandene Hardware:



# Platzierung 1:

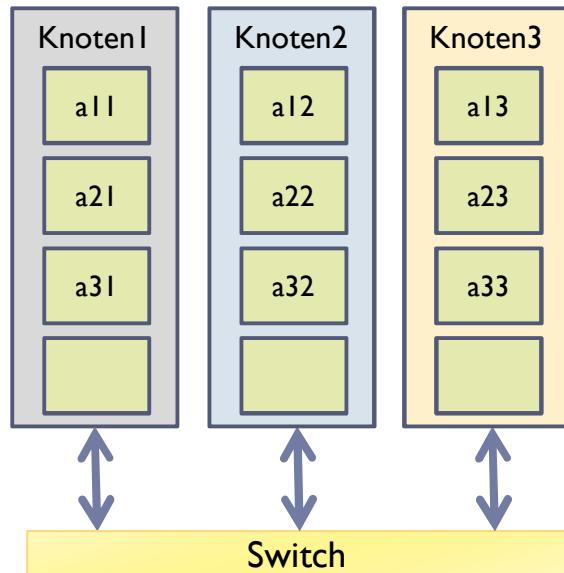


$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & \boxed{a_{22}} & \boxed{a_{23}} \\ a_{31} & a_{32} & \boxed{a_{33}} \end{pmatrix}$$

## Beobachtungen:

- Knoten 3 ist nicht voll ausgelastet
- Kommunikationslast:
  - Innerhalb der Knoten:
    - Knoten 1 – 3 Grenzen
    - Knoten 2 – 3 Grenzen
    - Knoten 3 – 0 Grenzen
  - Zwischen Knoten:
    - Knoten 1  $\Leftrightarrow$  Knoten 2 – 4 Grenzen
    - Knoten 2  $\Leftrightarrow$  Knoten 3 – 2 Grenzen
  - Knoten 2 an 6 Kommunikationen beteiligt
- Aufteilung der Berechnung
  - Im Falle von Multicore?
  - Im Fall von SMT? Suboptimal!

# Platzierung 2:



## Beobachtungen:

- Knoten gleich ausgelastet, balancierte Konfiguration
- Kommunikationslast:
  - Auf jedem Knoten je 2 Grenzen
  - Zwischen Knoten:
    - Knoten1,3  $\Leftrightarrow$  Knoten2 – je 3 Grenzen
    - Knoten2 insgesamt 6 Kommunikationen!
- Im Falle von Multicore?
- Im Fall von SMT?

$$A = \left( \begin{array}{c|cc} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{array} \right)$$

# Programmanalyse/Optimierung

---

Optimierungszyklus:

1. Leistung erfassen
2. Vergleichen zur Modellvorstellung
3. Bewerten ob das Programm effizient läuft:
  1. Engpässe identifizieren
  2. Abschätzung des Leistungsgewinns durch Behebung
  3. Aufwandabschätzung für Tuning durchführen

⇒ fertig, falls das Programm „hinreichend“ gut ist
4. Tuning (evtl. Algorithmus), goto 1

# Leistungsgewinn durch Optimierung

---

- ▶ I: Zeit die ein Programm ineffizient verbringt
- ▶ T: Bisherige Laufzeit

$$\text{OptimierungsEffekt} = \frac{T}{T - I}$$

- ▶ Verbringt ein Programm 90% der Zeit ineffizient so kann es 10 mal schneller rechnen!
- ▶ Nicht mit marginalen Optimierungen aufhalten

# Engpässe identifizieren

---

- ▶ Leistungsverlust durch Kommunikation und E/A bestimmen:
  - ▶ Wieviel Zeit verbringt das Programm ausschließlich mit diesen Tätigkeiten?
  - ▶ Wieviel Zeit rechnet das Programm?
- ▶ In der Praxis:
  - ▶ „Statistiken“ für CPU, E/A und Netzwerk erfassen
  - ▶ Mit Modellwerten vergleichen (oder Ausreißer feststellen)
  - ▶ Vergleich der Prozess-/Knotenleistung untereinander
    - ▶ Lastungleichheiten entdecken
  - ▶ Stellen im Code identifizieren
  - ▶ Evtl. temporale Zusammenhänge aufdecken

# Klassifikation des Engpasses

---

- ▶ Wie groß ist der Einfluss der Rechenzeit, Speicher, E/A ?
  - ▶ Bezogen auf konkrete Hardware!
  - ▶ Hilfsmittel um Analyse fortsetzen zu können
  - ▶ Wir betrachten Abschnitte der Aktivität über die Zeit
- 
- ▶ Klassen:
    - ▶ Rechenintensiv (CPU-bound)
    - ▶ Speicherintensiv (memory-bound)
    - ▶ Kommunikationsintensiv (network-bound)
    - ▶ E/A-intensiv (I/O-bound)

# Rechen-/Speicherintensive Programme

---

- ▶ Metriken der CPU verwenden:
  - ▶ Instruktionen per Cycle
    - Anzahl der Instruktionen pro Takt
  - ▶ FLOP(s)
    - Anzahl der Fließkommaoperationen
  - ▶ Cache-Miss-Ratio
    - Wurde der L1/L2/L3-Cache gut genutzt?
  - ▶ Cache-Bandbreite
    - Datenmenge die zwischen Cache & CPU transferiert wurde
  - ▶ Speicher-Bandbreite
    - Datenmenge die aus dem Speicher geladen/gespeichert wurde
  - ▶ ...
- ▶ Möglichkeiten:
  - ▶ Compileroptimierungen, Datenstrukturen, Cache-Alignment, ...

# Kommunikationsintensive Programme

---

- ▶ Kommunikationspartner als Matrix darstellen
  - ▶ Wie oft bzw. wieviele Daten wurden mit den einzelnen Prozessen ausgetauscht
- ▶ Netzwerkbandbreite/Paketanzahl auf NIC erfassen
- ▶ Warten Prozesse auf Kommunikationspartner?
  - ▶ Late Sender / Early Receiver
  - ▶ Kollektive Operationen
  - ▶ Oftmals durch Lastungleichheit erzeugt
- ▶ Möglichkeiten:
  - ▶ Passendere MPI-Funktion wählen
    - ▶ Asynchrone Kommunikation?
  - ▶ Mapping der Prozesse überprüfen
  - ▶ Datenpartitionierung verändern
  - ▶ Algorithmus?

# E/A-Intensiv

---

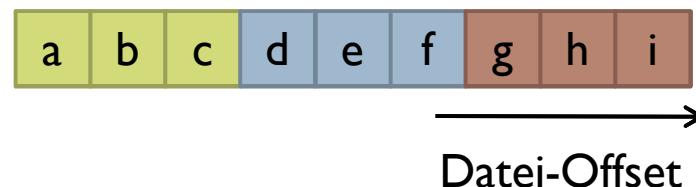
- ▶ Analyse sehr komplex!
  - ▶ Annahme: Paralleles Dateisystem
  - ▶ Client- und Server-E/A-Aktivität erfassen
- 
- ▶ Räumliches (und zeitliches) Zugriffsmuster betrachten
    - ▶ Wieviele Knoten und Server sind an der E/A beteiligt?
  - ▶ Möglichkeiten
    - ▶ Zugriffsmuster optimieren => grobgranulare Zugriffe!
      - ▶ Caching auf Anwendungsebene
      - ▶ Datenlayout verändern (Charakteristika einer Platte beachten!)
    - ▶ Kollektive E/A vs. individuelle E/A
    - ▶ Anpassen der Parameter für das Dateisystem
    - ▶ Asynchrone E/A, Write-Behind?

# Dateilayout

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Speicherreihenfolge:  
Zuerst (a, d, g)  
(b, e, h) dann (c, f, i)

- ▶ Variante 1:



- ▶ Variante 2:



# Zugriffsmuster und E/A-Durchsatz

- ▶ An der E/A beteiligte Komponenten:
  - ▶ I/O-Durchsatz  $\leq$  AnzahlClients  $\times$  Netzwerkbandbreite
  - ▶ I/O-Durchsatz  $\leq$  AnzahlServer  $\times$  Netzwerkbandbreite
  - ▶ Lastungleichheit der Server im Zugriffsmuster vermeiden!
- ▶ Beispiel:
  - ▶ 3 Prozesse lesen eine Datei aus, jeweils in gleichen Blöcken:



- ▶ Angenommene physikalische Abbildung auf zwei Servern/Platten:
- ▶ Lastungleichheit auf Servern meist schwer zu identifizieren!



# Leistungsmodellierung

## Zusammenfassung

- ▶ Verständnis der Hardwarearchitektur leistungsentscheidend
- ▶ Leistungsvergleiche mit Modellwerten erlaubt eine Bewertung der gemessenen Leistung
- ▶ Die größten Engpässe zuerst identifizieren und beheben

# Werkzeugarchitekturen

---

- ▶ Werkzeuge allgemein
- ▶ Grobstruktur von Werkzeugen
- ▶ Interaktive und automatische Werkzeuge
- ▶ Effizienter Entwurf von Werkzeugen
- ▶ Schnittstellenbasierter Entwurf eines universellen Monitorsystems

# Werkzeugarchitekturen

## Die zehn wichtigsten Fragen

- ▶ Was unterscheidet Offline- und Online-Werkzeuge?
- ▶ Was unterscheidet interaktive und automatische Werkzeuge?
- ▶ Was versteht man unter interoperablen Werkzeugen?
- ▶ Welche Werkzeugtypen gibt es zur Unterstützung der Programmierung?
- ▶ Erläutern Sie die Struktur von Werkzeugen
- ▶ Was versteht man unter Instrumentierung?
- ▶ Beschreiben Sie die Struktur von Offline-Werkzeugen
- ▶ Beschreiben Sie die Struktur von Online-Werkzeugen
- ▶ Welche allgemeinen Probleme hat der Werkzeugentwurf?
- ▶ Erläutern Sie den Ansatz eines schnittstellenbasierten Werkzeugentwurfs

# Was sind Werkzeuge?

---

Hier: Werkzeuge in den späteren Phasen des Lebenszyklus eines parallelen Programms

- ▶ Fehlersuche
- ▶ Optimierung
- ▶ Wartung
- ▶ Produktionsbetrieb

Wir betrachten nur Werkzeuge ab dem Zeitpunkt, zu dem ein halbwegs lauffähiges Programm vorliegt

# Was sind Werkzeuge?

---

## Begriffe

- ▶ **Offline-Werkzeuge**

Zeigen Informationen zum Programm nach dessen Ende oder zumindest deutlich verzögert an

- ▶ **Online-Werkzeuge**

Zeigen Informationen zum Programm gleichzeitig zum Ablauf an

# Was sind Werkzeuge?

---

## Begriffe...

- ▶ **Interaktive Werkzeuge**  
Gestatten eine direkte Interaktion mit dem Programm
- ▶ **Automatische Werkzeuge**  
Bearbeiten ohne Benutzereinwirkung das Programm zur Laufzeit
- ▶ **Interaktiv und automatisch nur bei Online-Werkzeugen von Bedeutung**

# Was sind Werkzeuge?

---

## Begriffe...

- ▶ Integrierte Werkzeugumgebungen
  - Mehrere Werkzeuge unter einer einheitlichen GUI vereint und meist gemeinsam nutzbar
- ▶ Interoperable Werkzeuge
  - Unabhängig voneinander entworfene und lauffähige Werkzeuge, die nun zusammenarbeiten
  - Sehr schwer zu realisieren; Forschungsziel

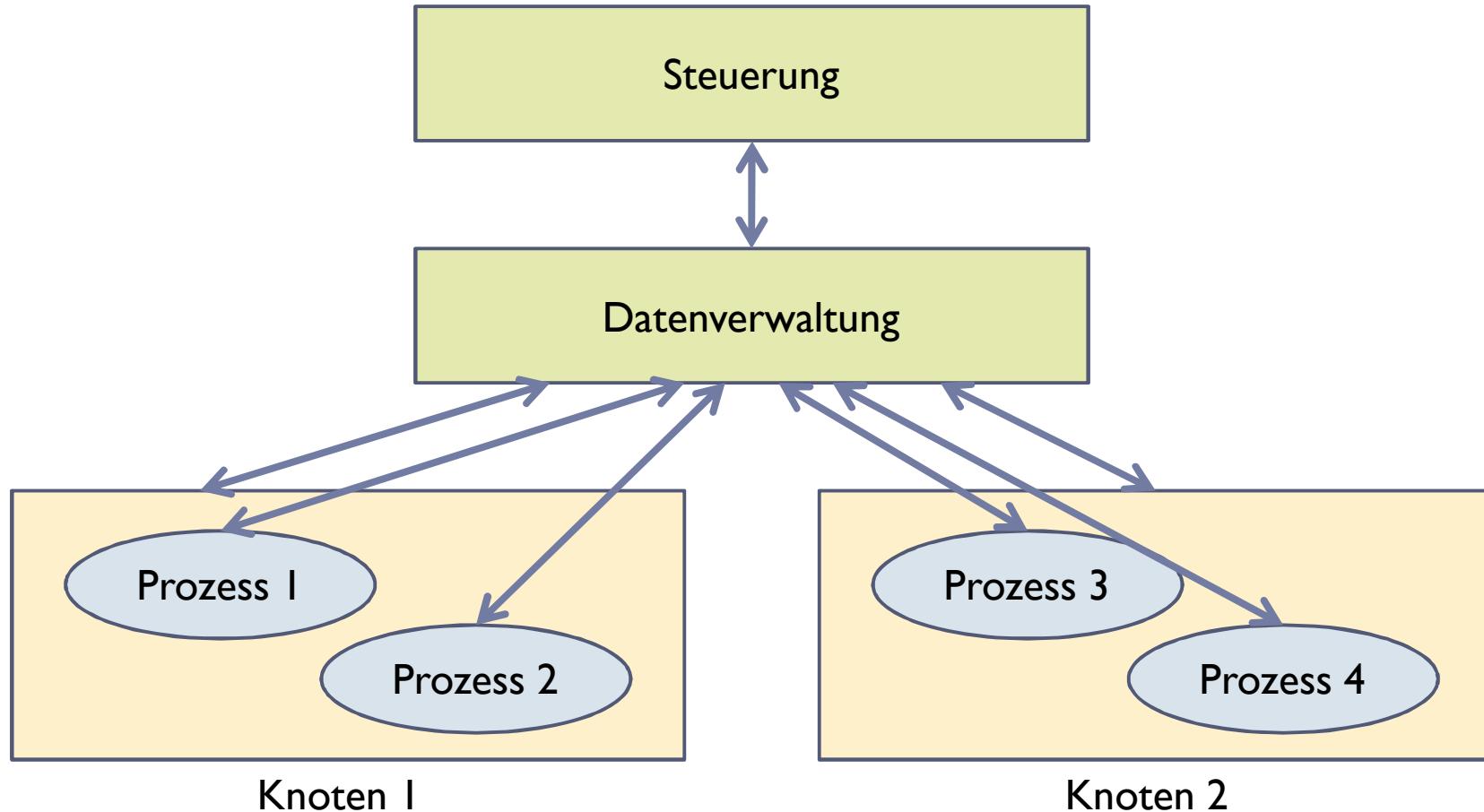
# Was sind Werkzeuge?

---

Hier nicht betrachtete Werkzeuge

- ▶ Werkzeuge zur Code-Erstellung
- ▶ Werkzeuge zur Parallelisierung
  - Interaktives Parallelisieren von Daten und Codebereichen
- ▶ Werkzeuge zur Gebietszerlegung der Daten
  - Gittergeneratoren bei numerischen Anwendungen

# Struktur von Werkzeugen



# Struktur von Werkzeugen...

---

Abstraktes Strukturbild klar

Aber: Realität sehr komplex

Gründe:

- ▶ Programm ist verteilt, also muß auch das Werkzeug verteilte Komponenten haben  
Werkzeug ist MPI-Programm?
- ▶ Die einzelnen Komponenten laufen auch auf Knoten des Rechnersystems  
Zusatzlast, Ausfallsicherheit

# Struktur von Werkzeugen...

## Begriffe

- ▶ Monitorsystem
  - Summe der Komponenten im System zur Beobachtung und Manipulation von HW und SW der Zielmaschine
- ▶ Monitor (knotenlokal)
  - Komponenten auf einem Rechnerknoten, die HW und SW des Knotens überwachen können

# Struktur von Werkzeugen...

---

## Begriffe...

- ▶ **Instrumentierung**

Einbringen von zusätzlichem Code, der die Erfassung von Daten steuert und die Beeinflussung des Programms gestattet

- ▶ **Sourcecode-Instrumentierung**

Primitivste Variante: Einbau von `printf(...)`

- ▶ **Binärcode-Instrumentierung**

Dynamisch eingebaute Sprünge in den Code des Monitorsystems

# Struktur von Werkzeugen...

## Offline-Werkzeuge

- ▶ Erfassung von Kenndaten zur Laufzeit des Programms
  - ▶ Visualisierung nach Programmende
  - ▶ Keine Beeinflussungsmöglichkeit
- 
- ▶ Überwachung wird aktiviert und Monitorsystem speichert Spurdaten ab (*trace*)
  - ▶ Visualisierung der Spur nach Programmende

# Struktur von Werkzeugen...

---

## Online-Werkzeuge

- ▶ Erfassung von Kenndaten zur Laufzeit
  - ▶ Sofortige Visualisierung
  - ▶ Sofortige Beeinflussung möglich
- 
- ▶ Direkte Interaktion zwischen Steuerung und Datenverwaltung
  - ▶ Optional Generierung von Spurdaten

# Struktur von Werkzeugen...

---

## Interaktive (Online-)Werkzeuge

- ▶ Benutzungsschnittstelle  
(graphisch oder Kommandozeile)
- ▶ Sofortige Darstellung  
(Skalierung der Darstellung schwierig)
- ▶ Steueranweisungen des Benutzers an Programm weiterleiten  
(Problem der zeitlichen Nähe)
- ▶ Nicht bei Stapelverarbeitung einsetzbar

# Struktur von Werkzeugen...

---

## Automatische (Online-)Werkzeuge

- ▶ Keine Benutzerinteraktion vorgesehen
- ▶ Steuerung bewertet Situation aufgrund von Heuristiken
- ▶ Steuerung manipuliert laufendes Programm und startet/stoppt einzelne Überwachungen

# Struktur von Werkzeugen...

---

## Allgemeine Probleme

- ▶ Skalierbarkeit der Datenerfassung  
Wie kann ich von den vielen Prozessoren die Daten effizient einsammeln?
- ▶ Konsistenz der Daten  
Sind sie alle zum selben Zeitpunkt entstanden?
- ▶ Skalierbarkeit der Darstellung  
Wie kann ich von den vielen Prozessen und Threads die Ergebnisdaten übersichtlich darstellen?
- ▶ Beeinflussungsfreiheit  
Das Programm soll nicht im Ablauf gestört werden
- ▶ Dynamik im Ablauf  
Variierende Knotenmenge / Prozeßmenge



# Werkzeuge

---

Die typischen interaktiven Werkzeuge:

- ▶ Fehlersuche (*debugging*)
- ▶ Leistungsanalyse (*performance analysis*)
- ▶ Programmvisualisierung
- ▶ Ergebnisvisualisierung
- ▶ Ablaufsteuerung (*computational steering*)
- ▶ *Problem Solving Environments*



# Werkzeuge...

---

## Die typischen automatischen Werkzeuge

- ▶ Ressourcenverwaltung
- ▶ Lastausgleich
- ▶ Sicherungspunktverwaltung
- ▶ Fehlertoleranzmechanismen

# Werkzeuge zur Fehlersuche

---

## Zweck

- ▶ Erkennen von Fehlerzuständen
- ▶ Auffinden von Fehlerursachen

## Probleme

- ▶ Anzahl der überwachten Prozesse
- ▶ Nichtdeterminismus im Ablauf
- ▶ Beeinflussungsfreie Beobachtung

# Werkzeuge zur Leistungsanalyse

---

## Zweck

- ▶ Visualisierung wichtiger Leistungsdaten
- ▶ Erkennen von Leistungsengpässen

## Probleme

- ▶ Erfassung der Daten produziert selber Last
- ▶ Zusatzlast minimieren oder herausrechnen
- ▶ Abweichende Abstraktionsebenen der Programmierung und der Meßdatenerfassung

# Werkzeuge zur Programmvisualisierung

---

## Zweck

- ▶ Darstellung des Ablaufs beim Nachrichtenaustausch
  - Wer kommuniziert wann mit wem
  - Leichte Erkennung von Verklemmungen

## Probleme

- ▶ Skalierung der graphischen Darstellung
- ▶ Zeitnähe

# Werkzeuge zur Ergebnisvisualisierung

---

## Zweck

- ▶ Visualisierung wichtiger Datenstrukturen  
Vor allem bei numerischen Anwendungen

## Probleme

- ▶ Falls online: Datenkonsistenz  
Z.B. alle Daten aus derselben Iterationsstufe des Programms
- ▶ Falls online: Datenmenge

# Werkzeuge zur Ablaufsteuerung

---

## Zweck

- ▶ Manipulation algorithmischer Kenngrößen zur Laufzeit  
Z.B. Algorithmus konvergiert schlecht; dann Korrektur einzelner Parameter
- ▶ Wichtig bei langlaufenden Programmen

## Probleme

- ▶ Wie bei Online-Ergebnisvisualisierung und Fehlersuche zusammen

# Problem Solving Environments

---

## Zweck

- ▶ Gruppe von Werkzeugen für einen bestimmten Einsatzzweck  
Z.B. für Anwendungen der Strömungsmechanik
- ▶ Umfaßt Gittergenerator, Ergebnisvisualisierer,  
Ablaufsteuerungskomponente etc.

## Probleme

- ▶ Vielfältig, deshalb noch kaum Vertreter dieser Gruppe

# Werkzeuge zur Ressourcenverwaltung

---

## Zweck

- ▶ Zuteilung von parallelen Programmen zu Mengen von Knoten aufgrund definierter Strategien
- ▶ Verwaltung der Anfragen um Rechenzeit

## Probleme

- ▶ Erfassen der Lastverhältnisse im System
- ▶ Berechnen einer optimalen Zuteilung (NP-hard)

# Werkzeuge zum Lastausgleich

---

## Zweck

- ▶ Korrektur von Ungleichbelastungen beliebiger Ressourcen  
(meist CPU) zur Laufzeit
- ▶ Erzielt optimale Ressourcennutzung  
= meist minimale Programmlaufzeit

## Probleme

- ▶ Welche lasterzeugende Komponente soll verlagert werden
- ▶ Wann? Wie?

# Werkzeuge zur Sicherungspunktverwaltung

---

## Zweck

- ▶ Bei langlaufenden Programmen automatische Abspeicherung von Sicherungspunkten
- ▶ Nach z.B. Rechnerausfall Fortsetzung des Programms vom Sicherungspunkt aus

## Probleme

- ▶ Konsistente Sicherungspunkte hinsichtlich z.B. nicht abgeschlossener Kommunikationen
- ▶ Wiederanlauf mit veränderter Knotenzahl

# Werkzeuge zur Fehlertoleranz

---

## Zweck

- ▶ Automatisches Tolerieren von Knotenausfällen
- ▶ Programm läuft auf anderer/kleinerer Konfiguration automatisch weiter
- ▶ Meist mit Sicherungspunkten realisiert

## Probleme

- ▶ Hoher Implementierungsaufwand

# Erste Zusammenfassung

---

- ▶ Entwurf und Implementierung von Werkzeugen sehr komplex
- ▶ Offline-Werkzeuge viel einfacher zu bauen  
Aber: genügen selten unseren Anforderungen
- ▶ Online-Werkzeuge wären nicht so viel schwieriger zu entwickeln  
Aber: Monitorsysteme *sehr* komplex
- ▶ **Monitorsystem ist selber verteiltes Programm**

# Konzepte zum effizienten Werkzeugentwurf

---

Wir betrachten jetzt Monitoring-Systeme für Online-Werkzeuge

Ziele:

- ▶ Getrennte Entwicklung von Monitoring-System und Werkzeug (Steuerkomponente)
- ▶ Dadurch mehr und bessere Werkzeuge in kürzerer Zeit

Warum geht das überhaupt?

- ▶ Die meisten Werkzeuge benötigen identische Funktionen des Monitorsystems

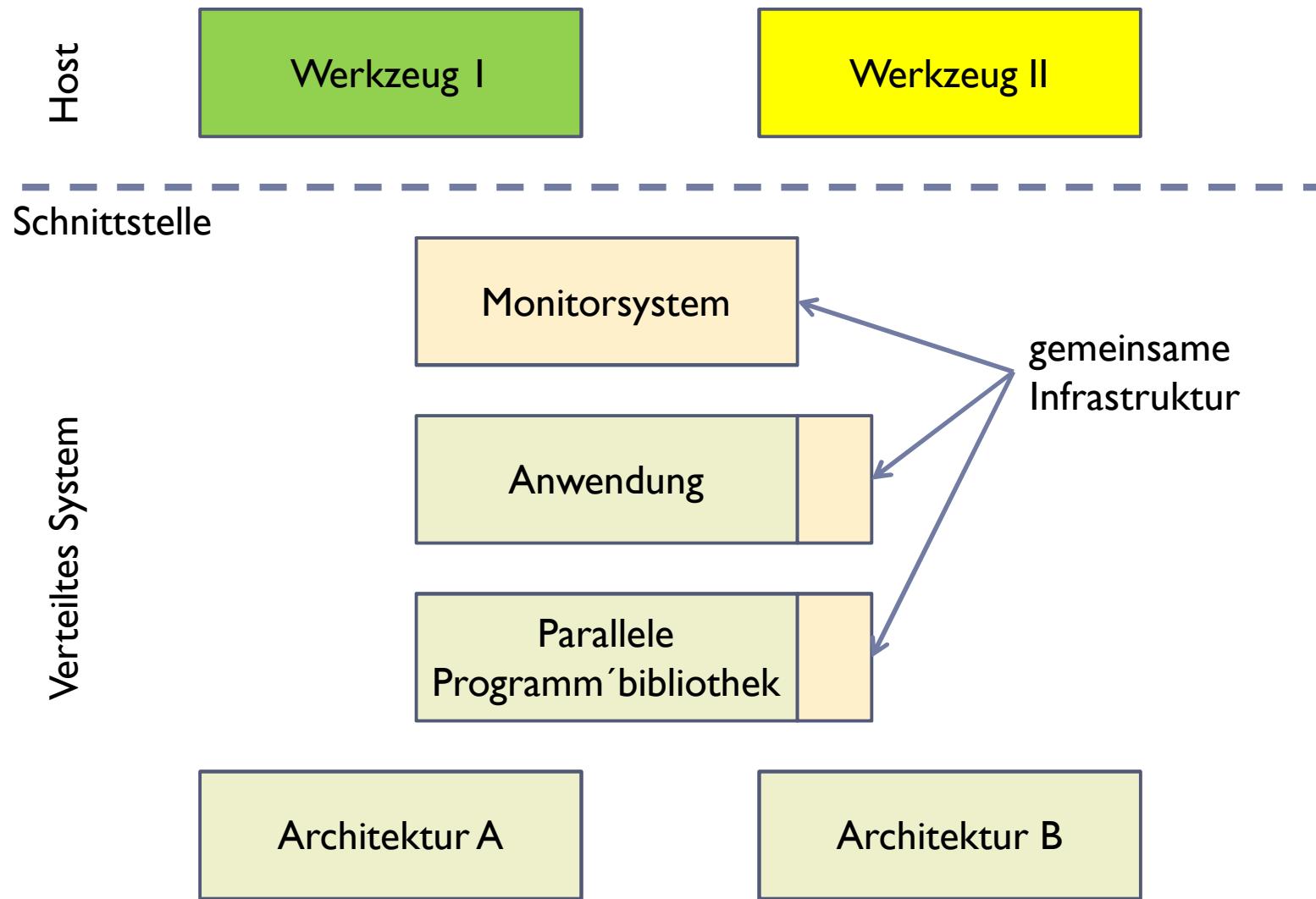
# Effizienter Werkzeugentwurf

---

## Vorgehensweise

- ▶ Abtrennen der Steuerkomponente des Werkzeugs von der Werkzeug-Infrastruktur
- ▶ Einführung einer genormten Schnittstelle zwischen beiden Teilen
- ▶ Identifikation von Infrastrukturteilen, die vielen Werkzeugen gemeinsam sind
- ▶ Zusammenführung gemeinsamer Komponenten in einer einheitlichen Implementierung

# Effizienter Werkzeugentwurf...



# Struktur der Schnittstelle

---

## Aus Sicht des Werkzeugs

- ▶ Monitorsystem ist Server, der Dienste an Objekten anbietet

## Diensteklassen

- ▶ Informationsdienste (I)
- ▶ Manipulationsdienste (M)
- ▶ Benachrichtigungsdienste (B)

## Objektklassen

System, Knoten, Prozesse, Threads, Nachrichten,  
Nachrichtenpuffer, Monitorobjekte

# Arbeitsprinzip der Schnittstelle

---

## Ereignis/Aktions-Modell

- ▶ Ereignis: interessierender Zustandsübergang im überwachten Programm
- ▶ Aktion: erwünschte Beobachtung oder Manipulation des Programms

## Dienstanforderungen

- ▶ Verhalten des Monitorsystems ist durch Ereignis/Aktions-Relationen bestimmt
- ▶ Programmierung der Relationen über definierte Schnittstelle
- ▶ Aktionen jeweils ausgelöst, wenn Ereignis erkannt wird
- ▶ Leere Ereignisdefinition => unbedingte Aktion

# Dienste der Schnittstelle (Beispiele)

---

## Prozesse

- I:** Statische / dynamische Informationen
- M:** Erzeugung, Überwachung einrichten / aufgeben, Modifikation des Speichers, Änderung Priorität, ...
- B:** Terminierung, Empfang eines Signals, ...

## Nachrichtenpuffer und Nachrichten

- I:** Inhalt des Puffers, Sender einer Nachricht, ...
- M:** Nachricht aus Puffer löschen, Nachricht markieren, ...
- B:** Eintrag einer Nachricht in einen Puffer, Empfang einer markierten Nachricht, ...

# Programmierung der Werkzeuge

---

## Vorgehensweise

- ▶ Einzelwerkzeuge setzen Diensteanforderungen an Monitorsystem ab und programmieren es somit für ihre Zwecke
- ▶ Bekommen Daten und Rückmeldungen zurück

## Wichtiger Aspekt

- ▶ Verschiedene Werkzeuge benutzen gleiche Ereignisdefinitionen und ähnliche Aktionen

# Beispiel: Leistungsanalyse

---

```
thread_has_started_lib_call([p_2],"MPI_SEND"):  
    pt_integrator_start(pt_i_1)  
    pt_counter_add(pt_c_1,$par5)
```

Wenn Prozeß p\_2 einen Sendeauftrag startet: aktiviere einen integrierenden Zähler und addiere die Nachrichtenlänge (\$par5) auf einen Zähler

```
thread_has_ended_lib_call([p_2],"MPI_SEND"):  
    pt_integrator_stop(pt_i_1)
```

Wenn der Prozeß den Aufruf beendet: halte den Zähler an

# Werkzeugarchitekturen

## Zusammenfassung

- ▶ Uns interessieren Werkzeuge für die späteren Lebenszyklusphasen der parallelen Programme
- ▶ Wir unterscheiden Offline- und Online-Werkzeuge
- ▶ Wir unterscheiden interaktive und automatische Werkzeuge
- ▶ Typisch: Fehlersuche online, Leistungsanalyse offline
- ▶ Online-Werkzeuge haben eine komplexe Struktur wegen der verwendeten Monitorsysteme
- ▶ Eine geeignete Schnittstellendefinition trennt das Monitorsystem von den Werkzeugen und vereinfacht so die Implementierung der Werkzeuge

# Fehlersuche

---

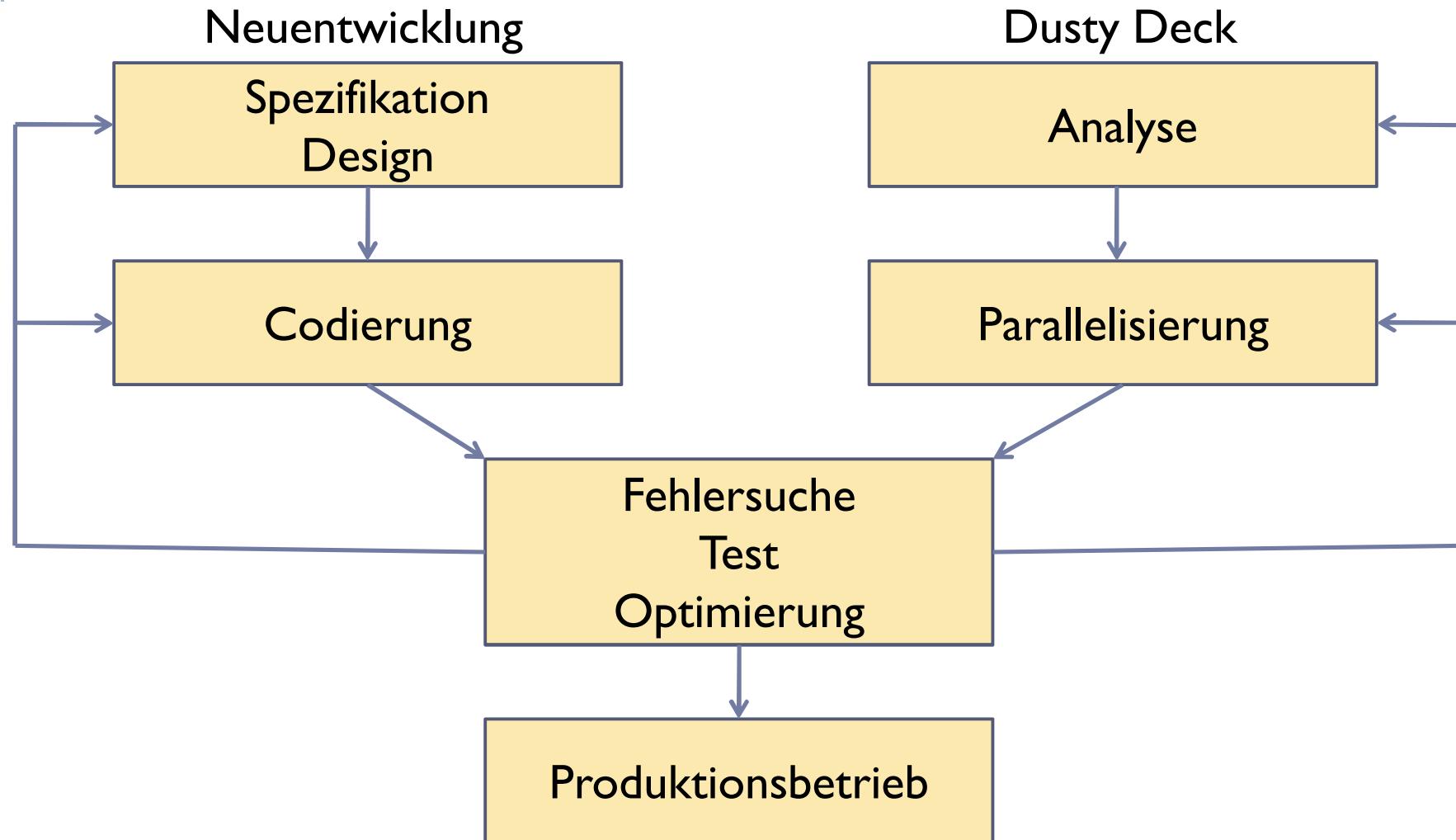
- ▶ Entwicklungszyklus paralleler Programme
- ▶ Fehlersuche
- ▶ Häufige Fehlerquellen
- ▶ Problemstellungen
- ▶ Werkzeugunterstützung
- ▶ Laufzeit-Debugger
- ▶ Spurbasierte Werkzeuge
- ▶ Haltepunktbasierte Werkzeuge
- ▶ Konzepte paralleler Debugger
- ▶ Ablaufkontrolle und Sicherungspunkte

# Fehlersuche

## Die zehn wichtigsten Fragen

- ▶ Welche Schritte umfaßt die Fehlersuche?
- ▶ Was sind die häufigsten Fehlerquellen in Programmen?
- ▶ Was versteht man unter einer Verklemmung?
- ▶ Was versteht man unter einem Überholvorgang?
- ▶ Welche Problemstellungen gibt es bei parallelen Programmen?
- ▶ Welche Kategorien der Werkzeugunterstützung unterscheiden wir?
- ▶ Wie stellen spurbasierte Werkzeuge typischerweise ihre Informationen dar?
- ▶ Welche Funktionen bietet ein haltepunktbasierter Debugger?
- ▶ Was ist deterministische Ablaufkontrolle und wie funktioniert sie?
- ▶ Was ist der Sinn von Sicherungspunkten?

# Entwicklungszyklus paralleler Programme



# Fehlersuche (Debugging)

---

Aufspüren von Fehlerzuständen und die Beseitigung  
ihrer Ursachen

## 4 Schritte

- ▶ Test, Regressionstest
- ▶ Erkennen der Fehlerwirkung
- ▶ Schließen auf die Fehlerursache
- ▶ Beseitigen der Fehlerursache

# Debugging?

9/9

0800 arctan started

1000 .. stopped - arctan ✓

13° 00' (032) MP-MC

(033) PRO 2

conv&

{ 1.2700 9.037847025

9.037846995 conv&

~~1.982647000~~ ~~2.130476415~~ 4.615925059(-2)

2.130476415

2.130676415

Rely 6-2 in 033 failed special sped test  
in relay

" 10.00 test .

Rely 2145

Rely 3376

Rely's changed

1100 Started Cosine Tape (Sine check)

1525 Started Multi Adder Test.

1545



Relay #70 Panel F  
(moth) in relay.

First actual case of bug being found.  
~~1630~~ arctangent started.

1700 closed down .

# Beispiel

```
foo (a, b, x, &result);
/* Ursache: `&' vergessen
   Compiler-Warning: pointer from
   integer without a cast */

void foo (int a, int b, int *x,
          int *result)
{
    *x = a+b;
    /* segmentation violation */
}
```

# Häufigste Fehlerquellen

---

## Sequentielle Programmierung

- ▶ Schnittstellenprobleme (Typen, Zeiger auf Parameter, ...)
- ▶ Zeiger und dynamische Speicherverwaltung
- ▶ Logische und arithmetische Fehler

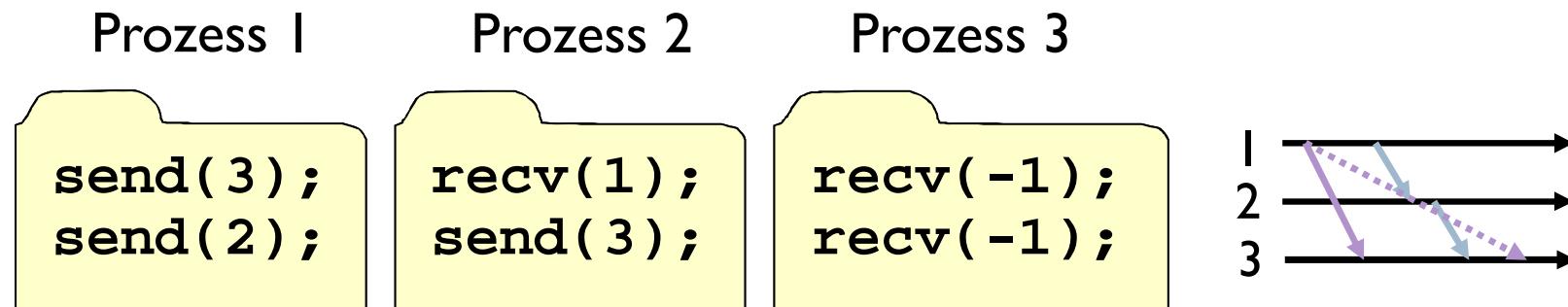
## Parallele Programme

- ▶ Kommunikationsfehler (Protokolle)
- ▶ Überholvorgänge (*races*)
- ▶ Verklemmungen (*deadlocks*)

# Häufigste Fehlerquellen: Überholtvorgänge

Definition: Ein Überholtvorgang entsteht durch unsynchronisierte, modifizierende Zugriffe auf gemeinsame Objekte (Adressebereiche, Nachrichtenpuffer)

Beispiel:



Konsequenz: Nichtdeterminismus,  
Nichtreproduzierbarkeit (schwer feststellbar)

# Häufigste Fehlerquellen: Verklemmungen

Definition: Bei einer Verklemmung warten Prozesse blockierend auf Ereignisse anderer Prozesse, die auch blockiert sind

Beispiel:

Prozess 1

Prozess 2

```
recv(...,2,...);  
send(...,2,...);
```

```
recv(...,1,...);  
send(...,1,...);
```

Konsequenz: Programm bleibt hängen  
(leicht feststellbar)

# Problemstellungen

---

Zusätzlich zu den normalen Problemen der Fehlersuche

- ▶ Erkennen einer Fehlerwirkung
- ▶ Suchen der Fehlerursache
  - ▶ Nichtreproduzierbarkeit der Fehlerwirkung
  - ▶ Nichtdeterminismus der Programmausführung
  - ▶ Ursache: Zeitabhängigkeit bei der Fehlerursache
- ▶ Unübersichtlichkeit: viele Prozesse
- ▶ Physische Verteiltheit
- ▶ Dynamik: Knoten- und Prozeßmengen variieren potentiell

# Erkennen einer Fehlerwirkung

---

## Normalerweise

- ▶ Zustand des Programms entspricht nicht der Spezifikation  
(am Ende / mittendrin)
- ▶ Vergleich mit Testdaten

## Bei parallelen Programmen

- ▶ Ergebnisse nicht nachrechenbar
- ▶ Ablauf abhängig von der Prozessorzahl
- ▶ Ablauf abhängig von zeitlichen Verhältnissen

=> Falsche Berechnungen schwer erkennbar



# Werkzeugunterstützung

---

- ▶ Statische Analyse
- ▶ printf() und WRITE
- ▶ Laufzeit-Debugger
  - ▶ Spurbasierte Werkzeuge
  - ▶ Haltepunktbasierter Werkzeuge
- ▶ Ablaufkontrolle und Sicherungspunkte

# Statische Analyse

---

Analyse des Programmtextes vor/zur Übersetzungszeit

- ▶ Sequentielle Aspekte
  - ▶ Strikte Typ- und Parameterprüfung
  - ▶ Erweiterte semantische Tests
  - ▶ Einsatz spezieller Werkzeuge: lint, insight
  - ▶ Gute ANSI-C-Compiler, Option –Wall (alle Warnungen)
- ▶ Parallele Aspekte
  - ▶ Erkennen möglicher Überholvorgänge
  - ▶ Prüfung auf Verklemmungsfreiheit
  - = Forschungsthemen (bisher ungelöst)

# printf() und WRITE

**Die Werkzeuge zur Fehlersuche schlechthin!**

Bei parallelen Programmen aber:

- ▶ Zuordnung zu einzelnen Prozessen schwierig  
Zeichen- / zeilenweises Mischen möglich
- ▶ Bei Netzen: Umleitung in ein gemeinsames Fenster?!
- ▶ Sortierung oft unmöglich, da keine globale Zeit
- ▶ Korrekte kausale Ordnung der Ausgaben nicht gewährleistet

# Laufzeit-Werkzeuge

---

Automatische Fehlerprüfung zur Laufzeit

Sequentielle Aspekte

- ▶ Dynamische Speicherverwaltung

Parallele Aspekte

- ▶ Parameterprüfung bei Programmierbibliothek
- ▶ Race-Erkennung (Forschungsthema)

Vorbereitung der Anwendung (Alternativen)

- ▶ Präprozessor und Neuübersetzung
- ▶ Binden mit speziell instrumentierter Bibliothek
- ▶ Instrumentierung der Binärdatei

# Spurbasierte Werkzeuge

---

## Merkmale

- ▶ Aufzeichnung relevanter Ereignisse des Programmlaufs
- ▶ Betrachtung der Spur durch „Browser“  
offline und auch online möglich
- ▶ Im Prinzip: automatisiertes `printf()`

## Aufgezeichnete Ereignisse

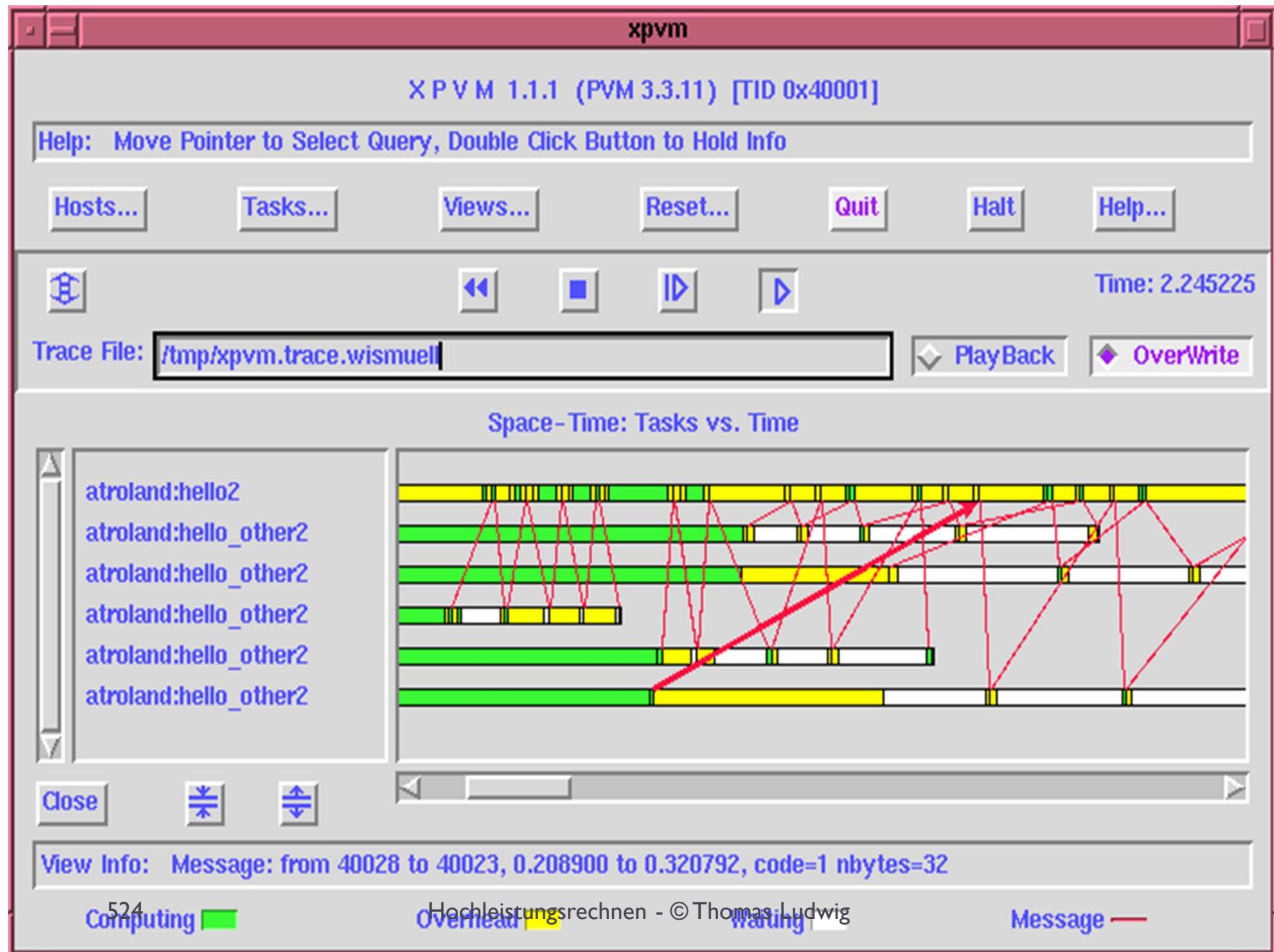
- ▶ Aufruf und Rückkehr der Funktionen der Programmierbibliothek
- Lokale Zeit, Dauer, Parameter
- ▶ Zum Teil auch benutzerdefinierte Ereignisse möglich

# Spurbasierte Werkzeuge...

---

## Darstellungsarten

- ▶ Raum-Zeit-Diagramme, Gannt-Diagramme
  - Darstellung einzelner Prozeßzustände
  - Knoten- und/oder prozeßorientierte Darstellung
  - Gut: globaler Überblick
  - Schlecht: Globale Ordnung meist trügerisch
- ▶ Folge von Schnappschüssen
  - Darstellung des globalen Zustands zu bestimmten Zeiten



# Spurbasierte Werkzeuge...

---

## Steuerung

- ▶ VCR-ähnliche Elemente: Start, Stop, Vor, Zurück, Einzelschritt
- ▶ Meist Auswahl relevanter Knoten und Prozesse

## Vorbereitung

- ▶ Präprozessor und Neuübersetzung
- ▶ Binden mit instrumentierter Bibliothek
- ▶ Laufzeitoption der Programmierbibliothek

## Bewertung

- ▶ Für globalen Überblick und zur Überwachung der Kommunikation

# Haltepunktbasierte Debugger

---

## Vorgehen

- ▶ Anhalten des Programms an interessanten Stellen
- ▶ Inspizieren des Programmzustandes
- ▶ Fortsetzen (oder Neustart) des Programms

## Bei erkanntem Fehler

- ▶ Hypothese zur Fehlerursache
- ▶ Neustart des Programms und Überprüfen der Hypothese

# Haltepunktbasierte Debugger...

---

## Typischer Funktionsumfang

- ▶ Anhalten des Programms  
Bedingt und/oder unbedingt
- ▶ Inspizieren des Programmzustandes  
Prozeduraufkeller, Parameter, Variablen
- ▶ Modifikation des Programmzustandes  
Setzen von Variablen, Veränderung des Codes(!)
- ▶ Ausführungskontrolle
  - ▶ Start und Stop
  - ▶ Einzelschritt (Anweisungen, Prozeduren)

# Konzepte paralleler Debugger

---

## Eigenschaften paralleler Programme

- ▶ Mehrere Aktivitätsträger  
Prozesse, evtl. Threads; evtl. mehrere Binärformate
- ▶ Dynamik  
Zur Laufzeit Änderungen der Knoten, Prozesse, ...
- ▶ Interaktion  
Kommunikation und Synchronisation zwischen Prozessen
- ▶ Verteiltheit  
Verteilte Information; kein globaler Systemzustand

Berücksichtigung dieser Eigenschaften sehr unterschiedlich

Kein Standard auf dem Gebiet in Sicht

# Umgang mit mehreren Prozessen

---

Zwei Methoden:  
Fenstertechnik und Prozeßmengen

- ▶ Pro Prozeß ein Fenster
  - Unabhängiger sequentieller Debugger pro Fenster
  - Leicht zu entwickeln; schwierige Benutzung bei vielen Prozessen
  - => (v.a.) für funktionsparallele Programme
- ▶ Ein einziges Fenster für alle Prozesse
  - Auswahl eines Prozesses zur Fehlersuche
  - Kommandos für Prozeßmengen
  - => (v.a) für datenparallele Programme
- ▶ Mehrere Fenster für beliebige Teilmengen von Prozessen
  - => für beliebige Programme (DETOP)

# Skalierbarkeit und Dynamik

---

Problem: Umgang mit höheren Prozeßanzahlen

- ▶ Kommandos für Gruppen von Prozessen
- ▶ Zusammenfassen identischer Ergebnisse verschiedener Prozesse
- ▶ Einsatz geeigneter graphischer Darstellungen

Problem: Debugging dynamisch generierter Prozesse

- ▶ Stoppen aller neu erzeugten Prozesse; manuelle Auswahl
- ▶ Automatische Auswahl über Mustervergleich mit Zusatzinformation

# Interaktion

---

## Überwachung von Kommunikation und Synchronisation

- ▶ Möglich durch Haltepunkte auf Bibliotheksfunktionen
- ▶ Meist keine weitergehende Unterstützung, wie z.B.
  - ▶ Ausgabe wartender Prozesse
  - ▶ Status von Nachrichtenwarteschlangen
  - ▶ Haltepunkte auf Nachrichten
- ▶ Ausweg
  - Gleichzeitige Benutzung spurbasierter Werkzeuge (i.a. nur lesender Zugriff) und von Spezialwerkzeugen (message queue manager, mqm)

# Verteiltheit

---

Daten der Anwendung sind verteilt

- ▶ Spezialwerkzeuge liefern globale Sicht

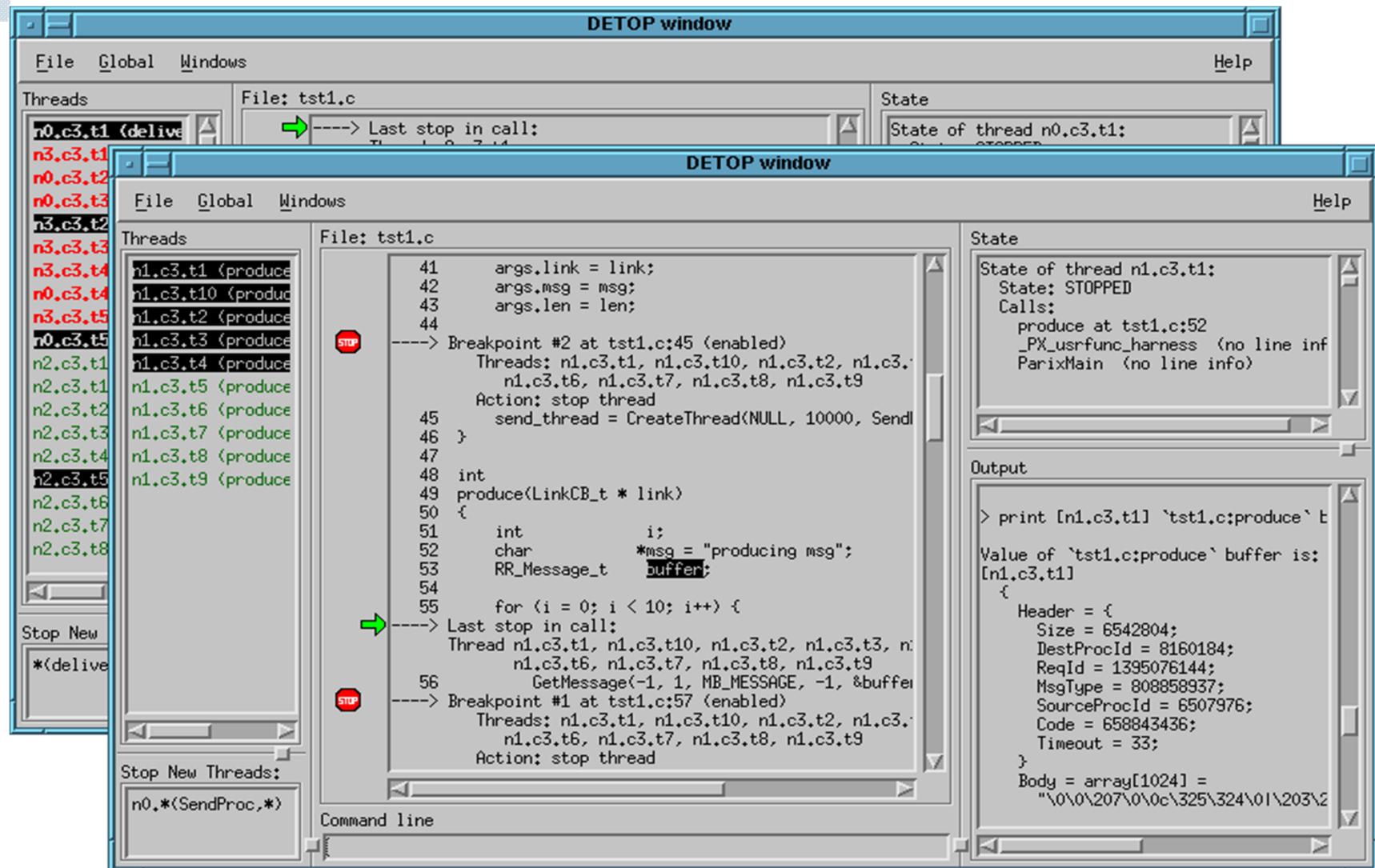
Gemeinsame Daten verteilter Prozesse

- ▶ Beispiele: MPI-Gruppen, gemeinsame Speichersegmente
- ▶ Problem: Einfrieren des Zustandes bei Erreichen des Haltepunktes
- ▶ Meist nur Anhalten eines Prozesses unterstützt
- ▶ Wenn globales Anhalten unterstützt, dann nie sofort(!)  
=> Zustandsveränderungen sind möglich

Globale Ereigniserkennung (Forschungsthema)

- ▶ Verknüpfung von Ereignissen in verschiedenen Prozessen
- ▶ Z.B. Ereignisse a und b sind kausal abhängig/unabhängig

# Beispiel DETOP



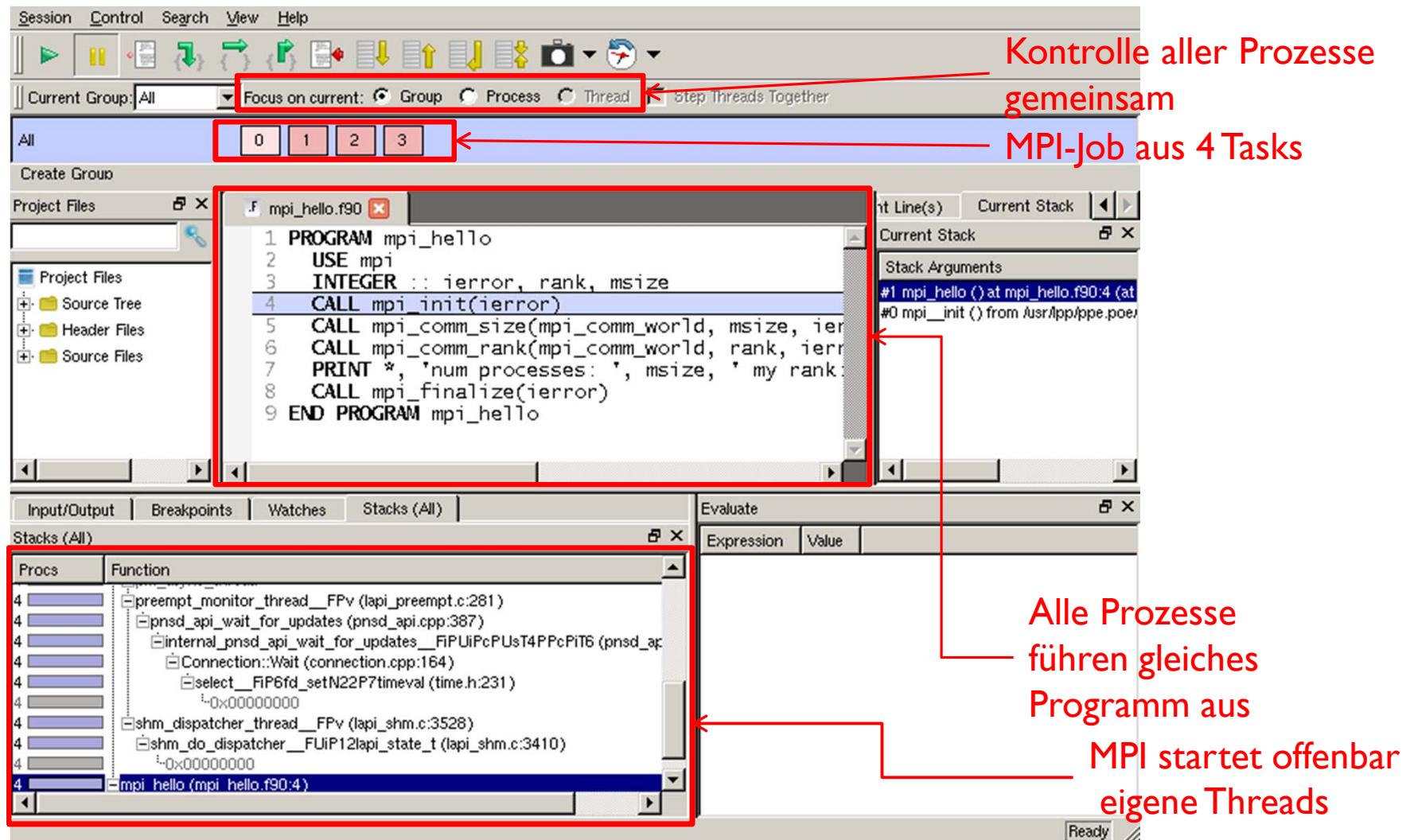
# Beispiel Allinea DDT

---

## The Distributed Debugging Tool (DDT)

*"DDT, the Distributed Debugging Tool is a comprehensive graphical debugger for scalar, multi-threaded and large-scale parallel applications that are written in C, C++ and Fortran."* Allinea

# Übersicht

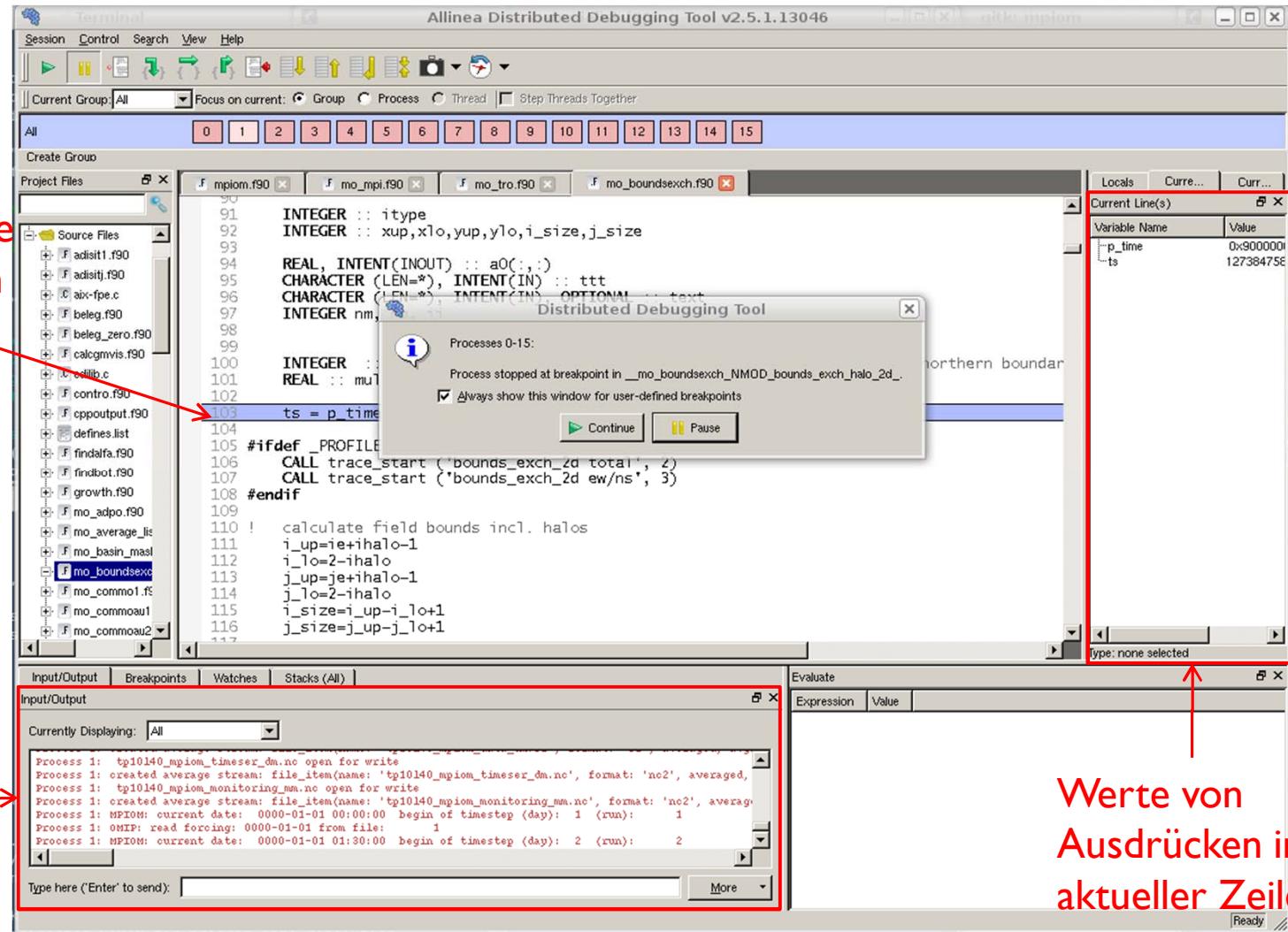


# Breakpoint im Ozeanmodell MPIOM

Erste ausführbare Zeile in Funktion

Standardausgabe und -fehler aller Prozesse

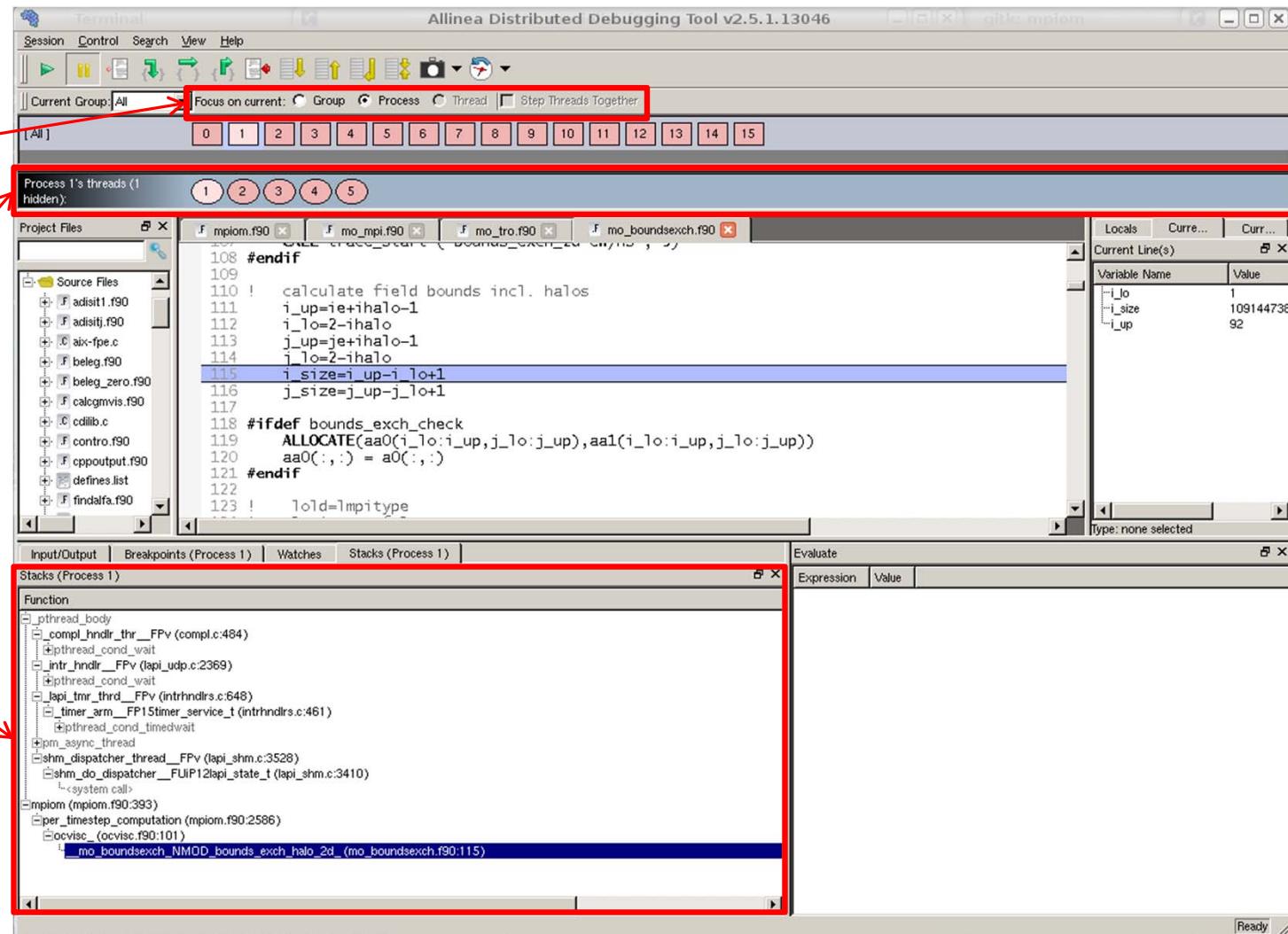
Werte von Ausdrücken in aktueller Zeile



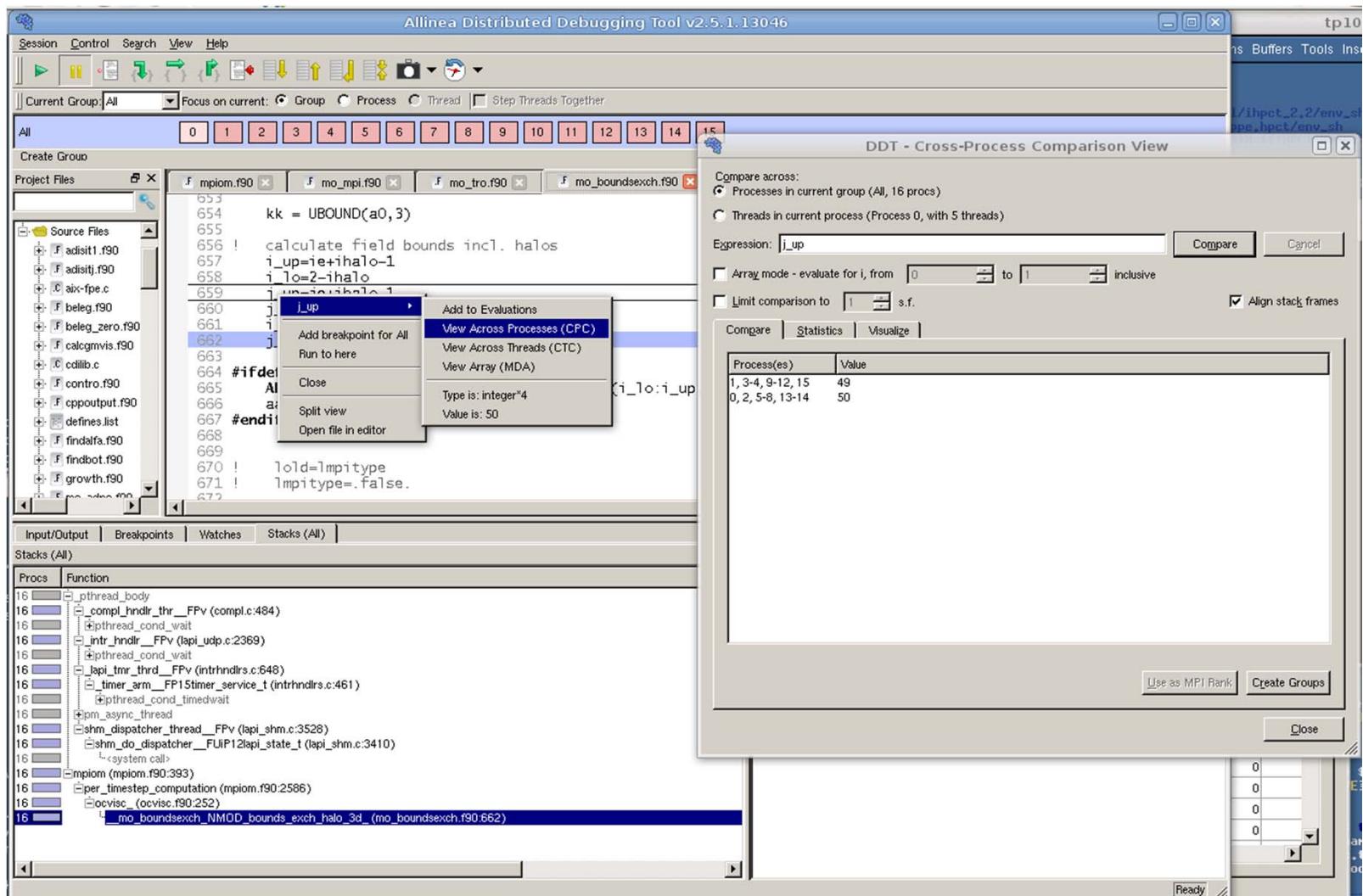
# Ausführung in einzelnen Prozess

Fokus auf  
einzelnen  
Prozess  
gewechselt

Thread- und  
Stackansicht  
wechseln mit



# Variablenansicht



# Deterministische Ablaufkontrolle

---

## Problem des Nichtdeterminismus

- ▶ Erzwingen einer deterministischen Abarbeitungsreihenfolge der Kommunikation  
Programm dadurch evtl. verlangsamt  
Varianten der Reihenfolgen systematisch testbar  
(Bei sequentiellen Programmen kein Problem!)

# Deterministische Ablaufkontrolle...

---

Funktionsweise eines Werkzeugs hierzu:

- ▶ Erster Programmlauf
  - Aufzeichnen der Reihenfolge des Eintreffens von Nachrichten bei Empfängern
- ▶ Weitere Programmläufe (*deterministic replay*)
  - Verwendung der Informationen aus dem ersten Programmlauf
  - Die Reihenfolge, wie Nachrichten beim Empfänger ankommen, wird gesteuert: zu früh eintreffende werden zurückgestellt

# Sicherungspunkte

---

## Problem der Zykluszeit

- ▶ Bei Fehler muß das Programm vom Anfang wiederholt werden, um nach der Fehlerursache zu suchen

## Vorgehensweise

- ▶ Zyklisches Erstellen von Sicherungspunkten
- ▶ Im Fehlerfall: Auswahl eines geeigneten Sicherungspunktes und Wiederanlauf des Programms von diesem Zeitpunkt aus.
- ▶ Evtl gekoppelt mit Ablaufkontrolle

# Fehlersuche

## Zusammenfassung

- ▶ Unterscheide Fehlerursache und Fehlerwirkung
- ▶ Typische Fehler paralleler Programme
  - ▶ Überholtvorgänge, Verklemmungen
- ▶ Probleme
  - ▶ Nichtreproduzierbarkeit, Nichtdeterminismus
  - ▶ Unübersichtlichkeit, physische Verteiltheit, Dynamik
- ▶ Spurbasierte Werkzeuge für einen globalen Überblick und Prüfung der Kommunikation
- ▶ Haltepunktbasierte Debugger für Detailuntersuchungen
- ▶ Ablaufkontrolle beseitigt Nichtdeterminismus
- ▶ Sicherungspunkte verkürzen den Testzyklus

# Leistungsanalyse

---

- ▶ Problemstellung
- ▶ Ziele der Leistungsanalyse
- ▶ Grundüberlegungen und Historie
- ▶ Leistungsmaße
- ▶ Amdahls Gesetz
- ▶ Einfluß der Problemgröße
- ▶ Superlinearer Speedup

# Leistungsanalyse

## Die zehn wichtigsten Fragen

- ▶ Was wollen wir mit der Leistungsanalyse bewerten?
- ▶ Welche Ursachen von Leistungsverlusten gibt es?
- ▶ Welche pessimistische Abschätzung machte Minsky?
- ▶ Welche zwei Maße charakterisieren typischerweise die Qualität einer Parallelisierung?
- ▶ Was beschreibt Amdahls Gesetz?
- ▶ Welchen Effekt hat ein sequentieller Anteil von n% auf die Leistungsausbeute?
- ▶ Wie verändern sich Aufwendungen von Berechnung und Kommunikation bei veränderten Datenstrukturen oder veränderter Hardware?
- ▶ Was ist superlinearer Speedup?
- ▶ Welche Formen der Parallelisierung widersetzen sich einer eindeutigen Speedup-Ermittlung?
- ▶ Wie sehen typische Speedup-Kurven aus für reale Anwendungen aus?

# Problemstellung

---

- ▶ Beurteilung der Leistungsfähigkeit des Hochleistungsrechners
  - ▶ Hardware, Betriebssystem, Compiler
- ▶ Beurteilung der Qualität der Parallelisierung  
Programmierbibliotheken, Programme
- ▶ Zur Kaufentscheidung  
Wieviel kostet es, wenn mein Programm in Zeit  $t$  berechnet sein muß

# Ziele der Leistungsanalyse

---

- ▶ Optimierung der Ressourcenauslastung  
Prozessoren, Speicher, Netzwerk, Platten
- ▶ Leistungsoptimierung bei exklusiven Ressourcen  
Minimale Laufzeit des Programms
- ▶ Verweilzeitoptimierung bei nicht-exklusiver  
Ressourcennutzung  
Ebenso wichtig: Fairness zwischen den  
Programmen
- ▶ Visualisierung des dynamischen Ablaufs

# Grundüberlegungen

---

- ▶ Leistungsanalyse ist sehr komplexes Themengebiet
  - ▶ Literatur z.B. bei Hwang
- ▶ Hier nur einfacher Ansatz verfolgt:
  - ▶ Prozessoranzahl (space-sharing-Betrieb)
  - ▶ Programmlaufzeit (wall clock time)

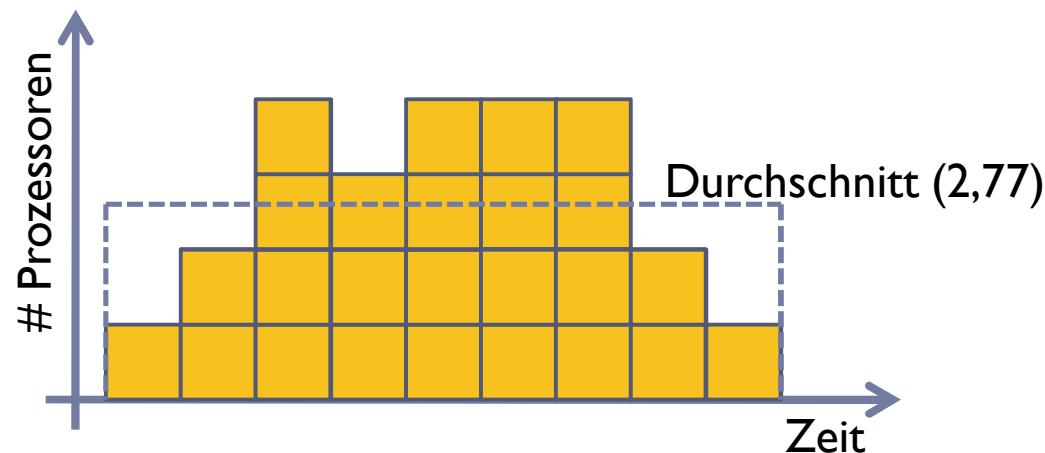
Annahme: Programme nutzen Ressourcen exklusiv
- ▶ Leistungsmodellierung/Leistungsvorhersage
  - ▶ Auch kompliziert und von Bedeutung

# Grundüberlegungen...

n Prozessoren, m ist maximaler Parallelismus ( $m \leq n$ ),  
l ist Leistung

Gesamte geleistete Arbeit  $W = l \cdot \sum_{i=1}^n i \cdot t_i$

Durchschnittlicher Parallelismus  $A = \frac{\sum_{i=1}^m i \cdot t_i}{\sum_{i=1}^m t_i}$



# Historische Überlegungen

---

Anfänglich gab es sehr pessimistische Überlegungen zum Leistungspotential von Parallelrechnern

Es wurde generell das Potential zur Leistungssteigerung kritisch gesehen bzw. verneint

Aus der Menge der Aussagen betrachten wir zwei:

- ▶ Minsky, 1971
- ▶ Amdahl, 1967

# Historische Überlegung: Minsky

---

- ▶ Ausgangspunkt  
Minsky betrachtet SIMD-Systeme (Vektorrechner)
- ▶ Überlegung: Viele Programme nutzen im ersten Arbeitsschritt alle Prozessoren, dann nur noch die Hälfte, ein Viertel usw.  
Beispiel: Addiere  $2^p$  Zahlen mit  $p$  Prozessoren
- ▶ Voraussage: Durchschnittlicher Parallelismus gleich  $\text{Id}(p)$
- ▶ Bewertung: Es gibt praktisch keine Programme, die so aufgebaut sind

# Leistungsmaße: Speedup und Effizienz

---

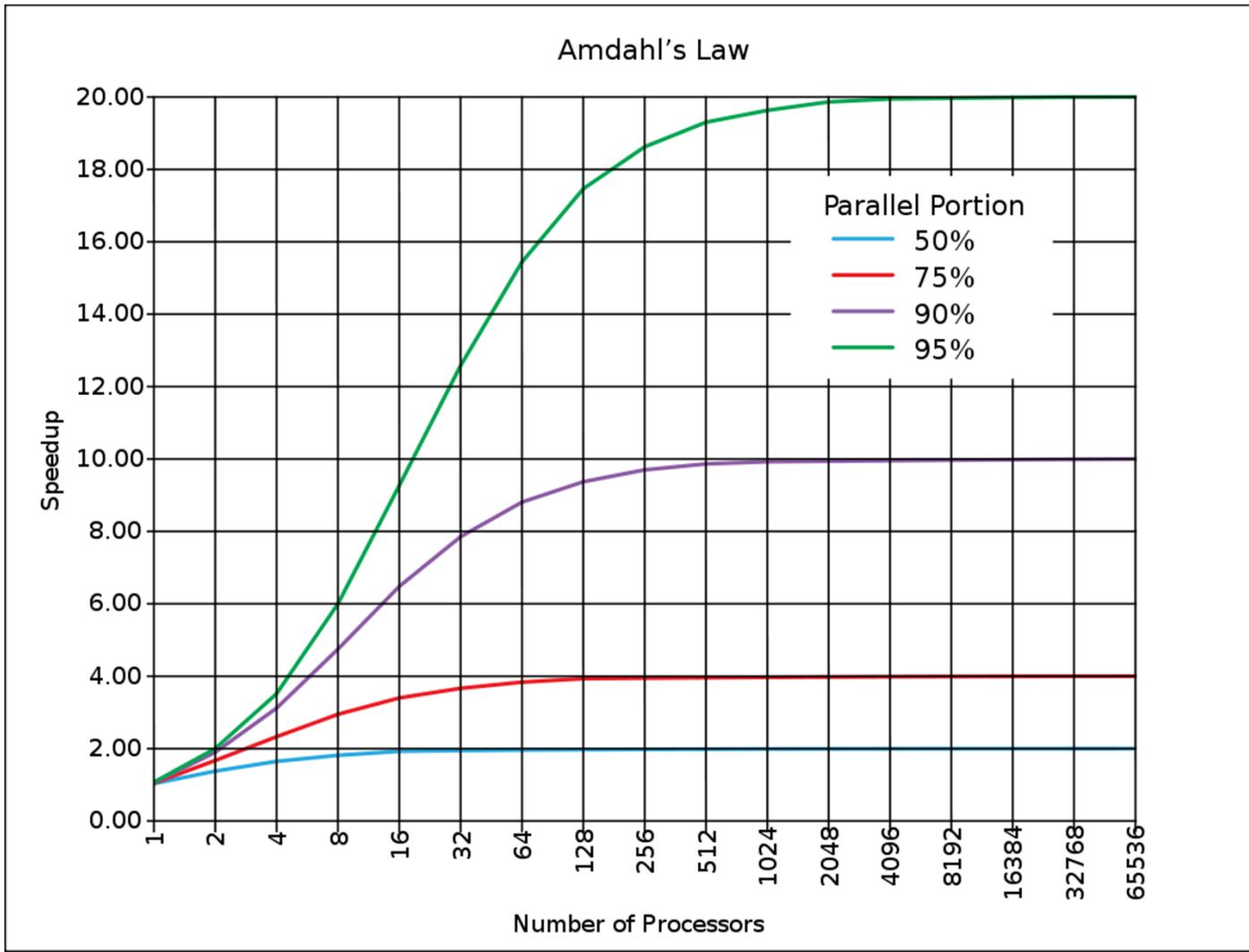
- ▶ Speedup:  $S = T(1)/T(p)$ 
  - $T(1)$  ist die Rechenzeit auf einem Prozessor
  - $T(p)$  ist die Rechenzeit auf  $p$  Prozessoren
- ▶ Forderung
  - Wähle jeweils den schnellstmöglichen Algorithmus  
(kritisch!!)
- ▶ Effizient:  $E = S/p$ 
  - Normalisierung nach der Prozessorzahl.  $E$  gibt an, wieviel Prozent des maximalen Speedup  $S=p$  erreicht werden
- ▶ Vermutung:  $S < p$  und damit  $E < 1$

# Amdahls Gesetz

---

- ▶ Ausgangspunkt: Jedes Programm enthält einen Bruchteil  $f$  an Operationen, die nur sequentiell ausgeführt werden können
- ▶ Es gilt daher für den Speedup
$$S \leq \frac{1}{f + (1-f)/p} \Rightarrow S_{\max} \leq \frac{1}{f}$$
- ▶ Beispiel:  $f=0.01 \Rightarrow S_{\max}=100$
- ▶ Praktische Erfahrung  
Sequentieller Anteil meist sehr gering
- ▶ Bewertung: Amdahls Gesetz gilt! Man muß versuchen, den sequentiellen Anteil klein zu halten

# Amdahls Gesetz...



# Ursachen von Leistungsverlusten

---

- ▶ Zugriffsverluste
  - ▶ Bei der Überwindung von Distanzen beim Datenaustausch zwischen Systemkomponenten
    - ▶ z.B. Nachrichtenaustausch oder entfernter Speicherzugriff bei NUMA
- ▶ Konfliktverluste
  - ▶ Bei der gemeinsamen Nutzung von Ressourcen durch mehrere Programmeinheiten
    - ▶ z.B. beim Bus- und Netzzugriff
- ▶ Auslastungsverluste
  - ▶ Bei zu geringem Parallelitätsgrad des Programms
    - ▶ z.B. permanente oder zeitweilige Lastungleichheit

# Ursachen von Leistungsverlusten...

---

- ▶ Bremsverluste
  - ▶ Beim Beenden gewisser Berechnungen, wenn bereits eine Lösung gefunden wurde
    - ▶ z.B. Suchbaumverfahren
- ▶ Komplexitätsverluste
  - ▶ Durch Zusatzaufwand im parallelen Programm gegenüber dem sequentiellen
    - ▶ z.B. Aufteilung der Daten
- ▶ Wegwerfverluste
  - ▶ Ergebnis mehrfach berechnet, aber nur eines von ihnen weiterbenutzt
    - ▶ z.B. Doppelberechnungen bei globalen Operationen

# Ursache von trügerischen Leistungsgewinnen

---

- ▶ Geänderte Ressourcennutzung
  - ▶ Bei fester Problemgröße und steigender Anzahl Prozessoren: relevante Daten- und Code-Anteile passen auf einmal wieder in den Cache  
=> deutlich schnellere Abarbeitung

# Einfluß der Problemgröße

---

Wir wollen nun betrachten, wie sich in verschiedenen Situationen die Aufwände für Kommunikation und Berechnung zueinander verhalten

## Schwache Skalierung (weak scaling)

- ▶ Bei schwacher Skalierung behalten wir die Problemgröße **pro Prozessor** bei und erhöhen die Anzahl der Prozessoren. Die Problemgröße nimmt dabei also zu!

## Starke Skalierung (strong scaling)

- ▶ Bei starker Skalierung behalten wir die gesamte Problemgröße bei und erhöhen die Anzahl der Prozessoren. Jeder hat dann zunehmend weniger zu tun – dies wird bei der Speedupbestimmung zugrunde gelegt.

# Einfluß der Problemgröße...

---

## Beispiel: Berechnung einer Matrix

- ▶ Größe:  $1000 \times 1000$  Elemente
- ▶ Betrachtung eines Iterationsschrittes
- ▶ Berechnung dauert  $1000 \times 1000 \times 1$  Zeiteinheit

## Parallelisierung mit $p=5$

- ▶ Aufteilung in Blöcke von Zeilen
- ▶ Berechnung:  $200 \times 1000 \times 1$  Zeiteinheit
- ▶ Kommunikation: benachbarte Blöcke tauschen eine Zeile aus  
(alle gleichzeitig)  
Aufwand:  $2 \times 1000 \times 1$  Zeiteinheit
- ▶ Kommunikations/Berechnungs-Verhältnis:  $1/100$

# Einfluß der Problemgröße...

Wir verwenden mehr Prozessoren

Parallelisierung mit  $p=10$

- ▶ Aufteilung in Blöcke von Zeilen
- ▶ Berechnung:  $100 \times 1000 \times 1$  Zeiteinheit
- ▶ Kommunikation: benachbarte Blöcke tauschen eine Zeile aus  
(alle gleichzeitig)  
Aufwand:  $2 \times 1000 \times 1$  Zeiteinheit  
Kommunikations/Berechnungs-Verhältnis:  $1/50$

Parallelisierung mit 100 Prozessoren

- ▶ Kommunikations/Berechnungsverhältnis:  $1/5$

# Einfluß der Problemgröße...

Wir vergrößern das Problem

Parallelisierung mit  $p=10$  und doppelt so großer Matrix

- ▶ Berechnung:  $141 \times 1414 \times 1$  Zeiteinheit
- ▶ Kommunikation:  $2 \times 1414 \times 1$  Zeiteinheit
- ▶ Kommunikations/Berechnungs-Verhältnis:  $1/70$   
( $p=5$  und Matrix  $1000 \times 1000$ :  $1/100$ )

Trotz doppelter Prozessoranzahl und doppelter  
Problemgröße verschlechtert sich das K/B-Verhältnis

# Einfluß der Problemgröße...

Wir kaufen neue Prozessoren

Parallelisierung mit  $p=10$  und doppelt so großer Matrix

- ▶ Berechnung:  $141 \times 1414 \times 0.3$  Zeiteinheiten
- ▶ Kommunikation:  $2 \times 1414 \times 1$  Zeiteinheit
- ▶ Kommunikations/Berechnungs-Verhältnis:  $1/21$

Zu dumm: die Effizienz der Parallelisierung wurde dadurch verschlechtert!

# Superlinearer Speedup

---

## Aufgabenstellung

- ▶  $n$  Sortieralgorithmen mit Laufzeiten  $t_i$
- ▶ Finde den besten Sortieralgorithmus

## Sequentielles Programm

- ▶ Lasse den ersten Algorithmus laufen; liefert  $t_{\min} = t_1$
- ▶ Starte nächsten Algorithmus; wenn er  $t_{\min}$  überschreitet, dann sofort stoppen; andernfalls  $t_{\min} = t_j$
- ▶ Wiederhole mit allen Algorithmen
- ▶ Zeitbedarf  $t_{\text{seq}} \geq t_{\min} \times n$

# Superlinearer Speedup...

## Paralleles Programm

- ▶ Nimm  $n$  Prozessoren
- ▶ Starte  $n$  Algorithmen gleichzeitig auf den Prozessoren
- ▶ Stoppe die Berechnung, wenn der erste fertig ist
- ▶ Zeitbedarf:  $t_{\text{par}} = t_{\text{min}}$

Erzielter Speedup:  $S = t_{\text{seq}} / t_{\text{par}} \Rightarrow S \geq n !?$

Perpetuum mobile des parallelen Rechnens?

# Superlinearer Speedup...

## Analyse des Beispiels:

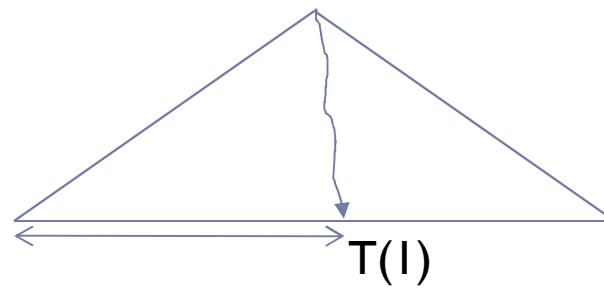
- ▶ Sequentielles Programm ist nicht „das schnellstmögliche“ sequentielle Programm
- ▶ Speedup-Definition fordert „schnellstmögliches“ Programm
- ▶ „Schnellstmöglich“ hier: quasiparallele Abarbeitung des Algorithmenvergleich auf einem Prozessor
- ▶ Damit verschwindet dann der superlineare Speedup

# Superlinearer Speedup...

Suchbaum-Verfahren sequentiell/parallel

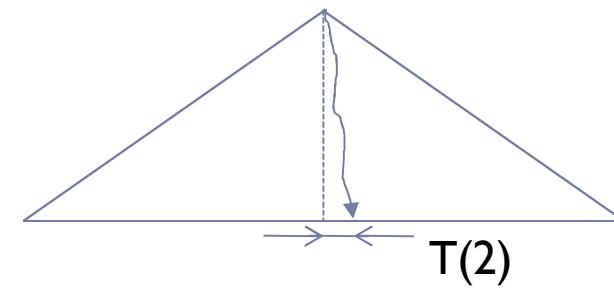
- ▶ Wir suchen **eine** Lösung

Fall I

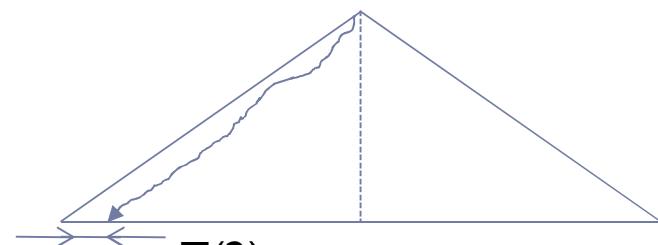
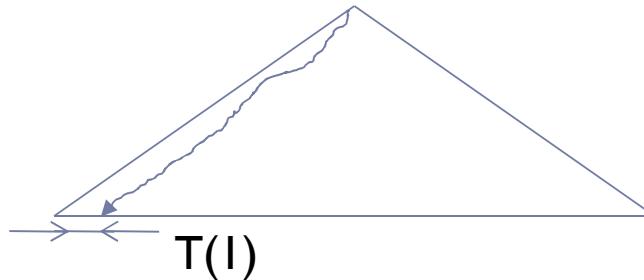


sequentiell

parallel



Fall 2



# Superlinearer Speedup...

---

Analyse des Beispiels:

- ▶ Für breite Bäume geht im Fall a) der Speedup gegen unendlich
- ▶ Für breite Bäume geht im Fall b) der Speedup gegen eins

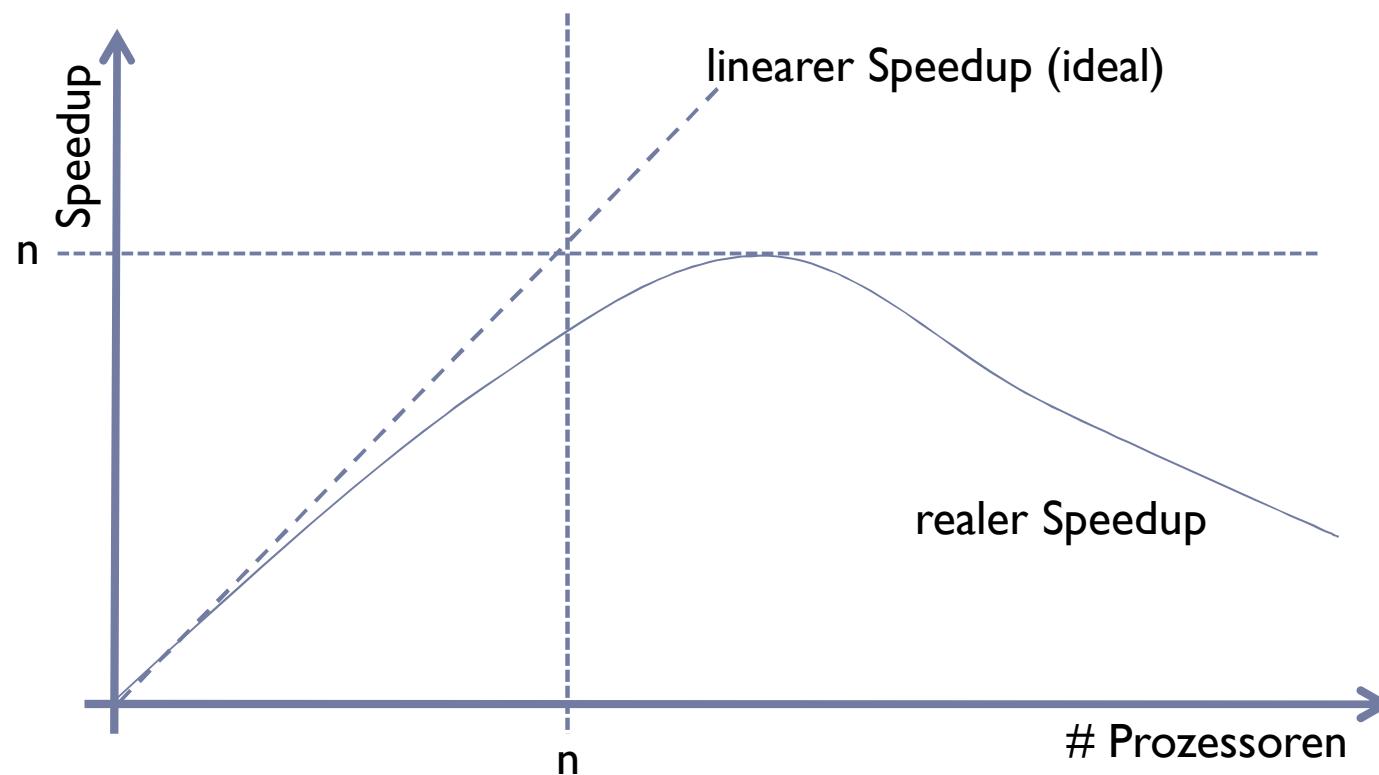
Angabe des Speedup sinnlos!  
Parallelisierung ebenso?

Bei Suche nach allen Lösungen problemlos!

- ▶ Hier verschwindet das Problem

# Speedup-Bestimmung

Verwendet in wissenschaftlichen Artikeln, in denen die Verfahren parallelisiert wurden



# Speedup-Bestimmung...

Speedup-Kurven angegeben für festen Datensatz

## Kurvenverlauf

- ▶ Bei guten numerischen Verfahren
  - S nahe bei p für alle p, die wir einsetzen
  - Effizienz zwischen 80% und 100%
- ▶ Bei schlechten Verfahren
  - Effizienz von 50% oder schlechter

Was tun, wenn die Kurven nicht optimal aussehen?

# Speedup-Bestimmung...

---

David Bailey

*Twelve Ways to Fool the Masses When Giving  
Performance Results on Parallel Computers*

Supercomputer Review, August 1991

Liste sehr beliebter (Selbst-)Täuschungsmethoden zur  
Erzielung besserer Ergebnisse

## Speedup-Bestimmung...

---

- (2) Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application
- (4) Scale up the problem size with the number of processors, but omit any mention of this fact
- (5) Quote performance results projected to a full system
- (7) When direct run time comparisons are required, compare with an old code on an obsolete system

# Speedup-Bestimmung...

- (12) If all else fails, show pretty pictures and animated videos, and don´t talk about performance

# Leistungsanalyse

## Zusammenfassung

- ▶ Leistungsanalyse dient der Rechner- und Programmbewertung
- ▶ Es gibt unterschiedliche Ursachen für Leistungsverluste
- ▶ Speedup und Effizienz charakterisieren die Qualität des parallelen Programms
- ▶ Amdahl: der Speedup ist nach oben begrenzt
- ▶ Problemgröße beeinflußt den erreichbaren Speedup
- ▶ Superlinearer Speedup ist nicht systematisch nutzbar
- ▶ Es gibt viele Möglichkeiten für irrtümlich oder absichtlich falsche Speedup- bzw. Leistungs-Angaben



# Lastausgleich

---

- ▶ Einführung und Problemstellung
- ▶ Die grundsätzliche Lösung
- ▶ Interessante Fragestellungen
- ▶ Die Lastbewertung
- ▶ Die Lastverschiebung

# Lastausgleich

## Die acht wichtigsten Fragen

- ▶ Was ist Lastungleichheit
- ▶ Was charakterisiert das Problem?
- ▶ Wie sieht die grundsätzliche Lösung aus?
- ▶ Welche interessanten Fragestellungen gibt es?
- ▶ Wie kann man Lastausgleich integrieren?
- ▶ Wie werden Lasten bewertet?
- ▶ Wie werden Lasten verschoben?
- ▶ Wie arbeiten Systeme mit Lastausgleich?

# Was ist Lastausgleich?

---

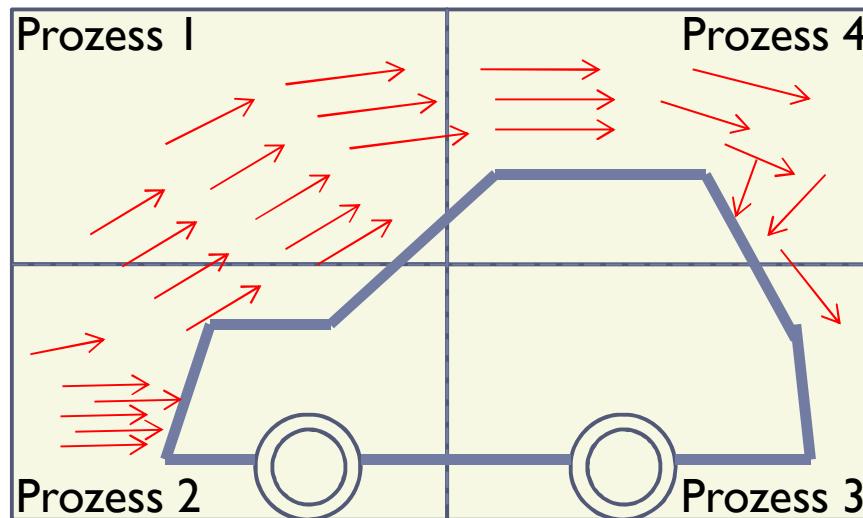
Besser sollte man zuerst fragen:

Was ist Lastungleichheit?

- ▶ Kritische Situation: die parallele Anwendung nutzt nicht alle Rechen-Ressourcen zu jeder Zeit  
Folge: Speedup-Kurve bekommt Knick
- ▶ Grund: Rechenlast ist unausgeglichen  
Möglicherweise zu Programmstart ausgeglichen, dynamischer Effekt zur Laufzeit
- ▶ Folgen: längere Programmlaufzeit, geringere Effizienz der Parallelisierung

# Beispiel für Lastungleichheit

- ▶ Gleichverteilung der Volumen zu Beginn:
  - ▶ alles ausgeglichen
- ▶ Danach Verlagerung der Teilchen:
  - ▶ Lastungleichheit zwischen den Volumenteilen



# Warum ist das interessant?

---

## Praktische Gründe

- ▶ Reduziere Programmlaufzeit
- ▶ Erhöhe Effizienz der Rechnernutzung

## Forschungsgegenstände

- ▶ NP-harte Optimierungsprobleme
- Globales Scheduling (was wird wann wo berechnet?)
- ▶ Anspruchsvolle Fragestellungen aus dem Bereich der Betriebssystemtechnik

# Das Problem

---

Lastungleichheit variiert dynamisch

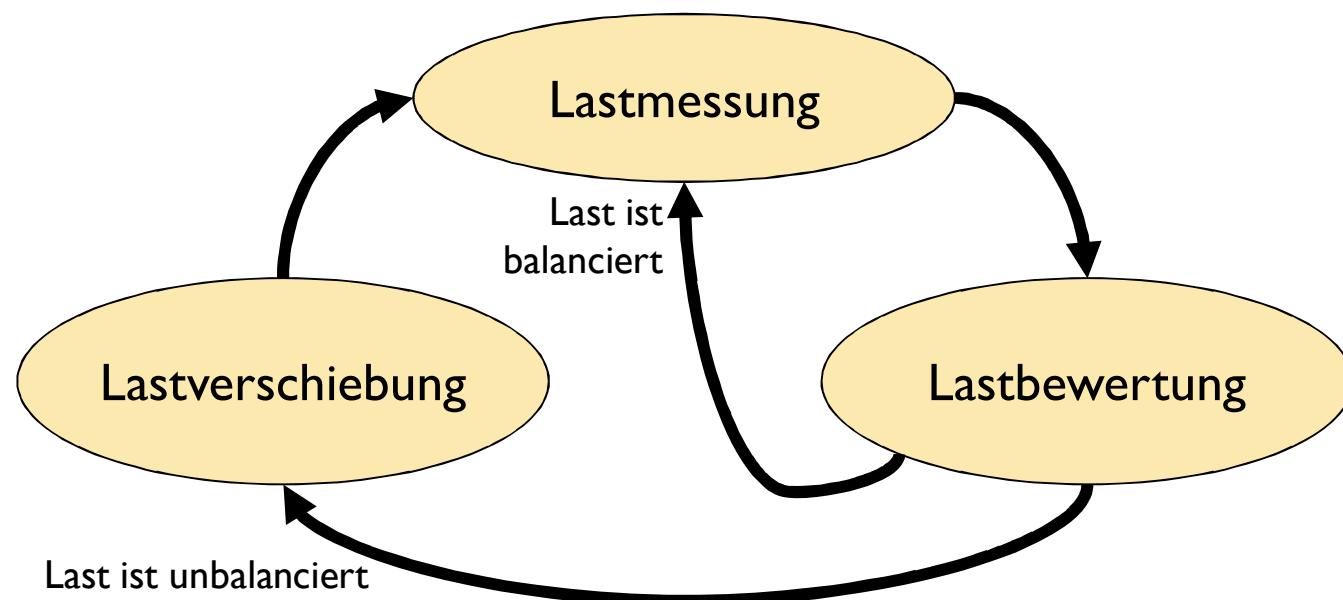
- ▶ Typisches Verhalten vieler paralleler und verteilter Anwendungen
- ▶ Statischer Ausgleich (bei Programmstart) nicht möglich, da Lastungleichheit daten- und zeitabhängig
- ▶ Lastungleichheit verringert den Systemdurchsatz

Ressourcen von Interesse

- ▶ Hauptsächlich der Prozessor
- ▶ Seltener: Speicher, Netzwerk, Platten

# Die grundsätzliche Lösung

## Regelkreis der Lastverwaltung



# Die grundsätzliche Lösung...

---

Es gelten die üblichen Regelkreischarakteristika

- ▶ Schnelle Reaktion
- ▶ Genaues Nachregeln
- ▶ Keine Überreaktion
- ▶ Kein Oszillation
- ▶ Keine Belastung des geregelten Systems

Technische Umsetzung der Anforderungen sehr  
komplex

Teilweise sich widersprechende Anforderungen

# Interessensbereiche

---

- ▶ Integrationstyp
  - ▶ Anwendungs- oder Systemintegration
- ▶ Lastmessung
  - ▶ Schnell, genau, umfassend, beeinflussungsarm
- ▶ Lastbewertung
  - ▶ Schnell, korrekt
  - ▶ Die Zukunft aus der Vergangenheit vorhersagen
- ▶ Lastverschiebung
  - ▶ Muss mehr bringen als kosten

# Integrationstyp

## Anwendungsintegriert

- ▶ In den Code der Anwendung integriert
- ▶ Überschaubare Implementierungskomplexität
- ▶ Wiederholter Implementierungsaufwand
- ▶ Meist nicht für andere Programme direkt übernehmbar
- ▶ Optimal an eine Anwendung angepasst
- ▶ Arbeitet mit allen Betriebssystemen zusammen (auch mit lastbalancierten)

# Integrationstyp...

## Systemintegriert

- ▶ In das Betriebssystem integriert oder mit ihm eng verbunden
- ▶ Hohe Implementierungskomplexität
- ▶ Einmaliger Implementierungsaufwand
- ▶ Optimal an das System angepasst, aber an keine spezifische Anwendung
- ▶ Arbeitet mit allen Anwendungen zusammen (auch mit lastbalancierten)

# Lastmessung

## Dynamisches Messen verschiedener Kenndaten

- ▶ Zunächst meist nur knotenbezogen
- ▶ Zur Verfeinerung auch prozessbezogen
- ▶ Meist sehr beeinflussungsarm

## Mechanismen

- ▶ Online-Werkzeuge wie z.B. Dyninst/Paradyn



# Lastbewertung

---

## Einfach

- ▶ Einzelwerte: Prozessorleerlaufzeit, Speichernutzung

## Komplex

- ▶ Systemkenndaten, Anwendungskenn Daten
- ▶ Auswertung mit neuronalen Netzen
- ▶ Global konsistenter Blick
- ▶ Metawissen (z.B. vergangene Entscheidungen)

Problem: alle Daten beschreiben Vergangenheit

# Lastbewertung...

## Konkretes System

- ▶ Zentrale Bewertungskomponente
- ▶ Erfasse alle knotenbezogenen Daten
- ▶ Bestimme einen überlasteten und einen unterlasteten Knoten
- ▶ Messe prozessbezogene Daten auf dem überlasteten Knoten
- ▶ Identifiziere geeigneten Kandidaten zur Verschiebung

# Lastbewertung...

## Probleme

- ▶ Zentrale Komponente in größeren Systemen nicht mehr praktikabel
- ▶ Daten müssen über ein Intervall erfasst werden, das nicht zu kurz sein darf
- ▶ Gleichzeitig aber schnelle Reaktion erwünscht
- ▶ Situationsabhängige Verfeinerung der Messungen
- ▶ Stabil gegen Oszillationen

# Lastverschiebung

---

## Verschiebeobjekte

- ▶ Einfacher: Anwendungsebene (feingranular)  
Datenpakete, Anfrage
- ▶ Komplex: Systemebene (grobgranular)  
Prozesse, Objekte, Dateien

## Allgemeine Probleme

- ▶ Bezug zu anderen Objekten muss gesichert sein
- ▶ Verschiebung muss Gewinn bringen

# Lastverschiebung...

---

## Beispiel: Prozessverschiebung

- ▶ Wie stoppt man einen laufenden Prozess?
- ▶ Wie verschiebt man ihn?
- ▶ Was verschiebt man konkret?
- ▶ Wie behandelt man offene Dateien?
- ▶ Wie behandelt man Signale?
- ▶ Was geschieht während der Verschiebung?
- ▶ Was passiert mit Nachrichten nach der Verschiebung?
- ▶ Wie finden sich die Kommunikationspartner nach der Verschiebung wieder?

# Lastverschiebung...

## Beispiel: Intel Hypercube Parallelrechner

- ▶ Verschiebung mittels Paging-Mechanismus über das Netzwerk
- ▶ Nur aktive Code-Seite verlagern; restliche Seiten werden durch auftretende Seitenfehler geholt
- ▶ Nachrichten an den Prozess bei den Sendern zwischengespeichert
- ▶ Quellknoten der Verschiebung gibt neuen Ort an Sender bekannt; diese korrigieren ihre Tabellen

# Lastverschiebung...

Beispiel: Verschiebung im Cluster mit CoCheck

- ▶ Alle Kommunikationen stoppen
- ▶ Prozessabbild durch Dump-Funktion erstellen
- ▶ Prozess zum Ziel kopieren
- ▶ Andere Prozesse weiterlaufen lassen; ggf. über neuen Ort informieren

# Lastverschiebung...

## Beispielproblem: Offene Dateien

- ▶ Mitprotokollieren **aller** Systemaufrufe durch zwischengeschaltete Bibliothek (wie MPI-Profiling-Bibliothek)
- ▶ Damit weiß man immer, wo der Prozess in einer Datei gerade liest
- ▶ Nach der Verschiebung Dateien wieder öffnen und die vermerkte Stelle wieder anfahren

# Zur Lage der Werkzeuge

---

## Benötigte Zusatzwerkzeuge

- ▶ Visualisierung der Messungen und Entscheidungen des Lastausgleichers

## Situation der aktuellen Werkzeuge

- ▶ Keines kann mit Prozessmigration umgehen
- ▶ Beispielproblem: Verschiebung eines Prozesses, für den ein Haltepunkt definiert ist
- ▶ Beispielproblem: Verschiebung eines Prozesses, dessen Code dynamisch instrumentiert ist

# Und weil das ganze so schwer ist...

---

... gibt es kaum Systeme mit dynamischem Lastausgleich

- ▶ Aber es gibt Tonnen von Literatur, bei der Simulationen des Verhaltens eines solchen Systems analysiert werden
- ▶ Und es gibt die initiale Lastplazierung, die meist banal ist, aber auch als Lastausgleich verkauft wird

Aber es gibt viele Anwendungen, bei denen die Programmierer mühevoll einen Lastausgleich eingebaut haben

# Lastausgleich

## Zusammenfassung

- ▶ Lastungleichheit ist die ungenügende Nutzung wichtiger Ressourcen
- ▶ Lastungleichheit variiert dynamisch und lässt sich deshalb nicht durch einen statischen Ansatz kontrollieren
- ▶ Die Lastverwaltung folgt dem Prinzip des Regelkreises
- ▶ Die Lastverwaltung kann in die Anwendung oder in das System integriert werden
- ▶ Eine effiziente Lastbewertung unterliegt vielen Einzelfragestellungen
- ▶ Die Lastverschiebung ist die technisch anspruchsvollste Komponente
- ▶ Aufgrund der Komplexität gibt es kaum Realisierungen auf Betriebssystemebene
- ▶ Häufig wird aber Lastausgleich in die Anwendung eingebaut

# Fehlertoleranz

---

- ▶ Einführung
- ▶ Begriffsbildung
- ▶ Techniken der Fehlertoleranz
  - ▶ Statisch
  - ▶ Dynamisch
- ▶ Quantitative Bewertung
- ▶ Systeme in der Praxis

# Fehlertoleranz

## Die zehn wichtigsten Fragen

- ▶ Wozu Fehlertoleranz?
- ▶ Warum sind Parallelrechner in diesem Bereich wichtige Architekturtypen
- ▶ Welche Kenngrößen werden verwendet?
- ▶ Welche Fehlerkategorien gibt es?
- ▶ Was ist statische Redundanz?
- ▶ Welches sind hierbei die typischen Konzepte?
- ▶ Was ist dynamische Redundanz?
- ▶ Welche Durchführungsphasen finden wir hierbei?
- ▶ Wie sieht die qualitative Bewertung eines Systems ohne Reparatur aus?
- ▶ Welche Umsetzung gibt es für Linux-Systeme?

# Warum als Thema Fehlertoleranz?

---

Vorlesungsstunde hätte auch „Ausfallsicherheit“ oder „Hochverfügbarkeit“ heißen können

„Fehlertoleranz“ ist der allgemeine Mechanismus, mit dem Ausfallsicherheit und Hochverfügbarkeit erzielt wird

# Motivation

---

## Wozu Fehlertoleranzmechanismen?

- ▶ Ausfälle im System werden verdeckt und das System läuft mit kurzer Unterbrechung oder mit verminderter Leistung weiter
- ▶ Schutz vor Verlust von Menschenleben und/oder Sachwerten

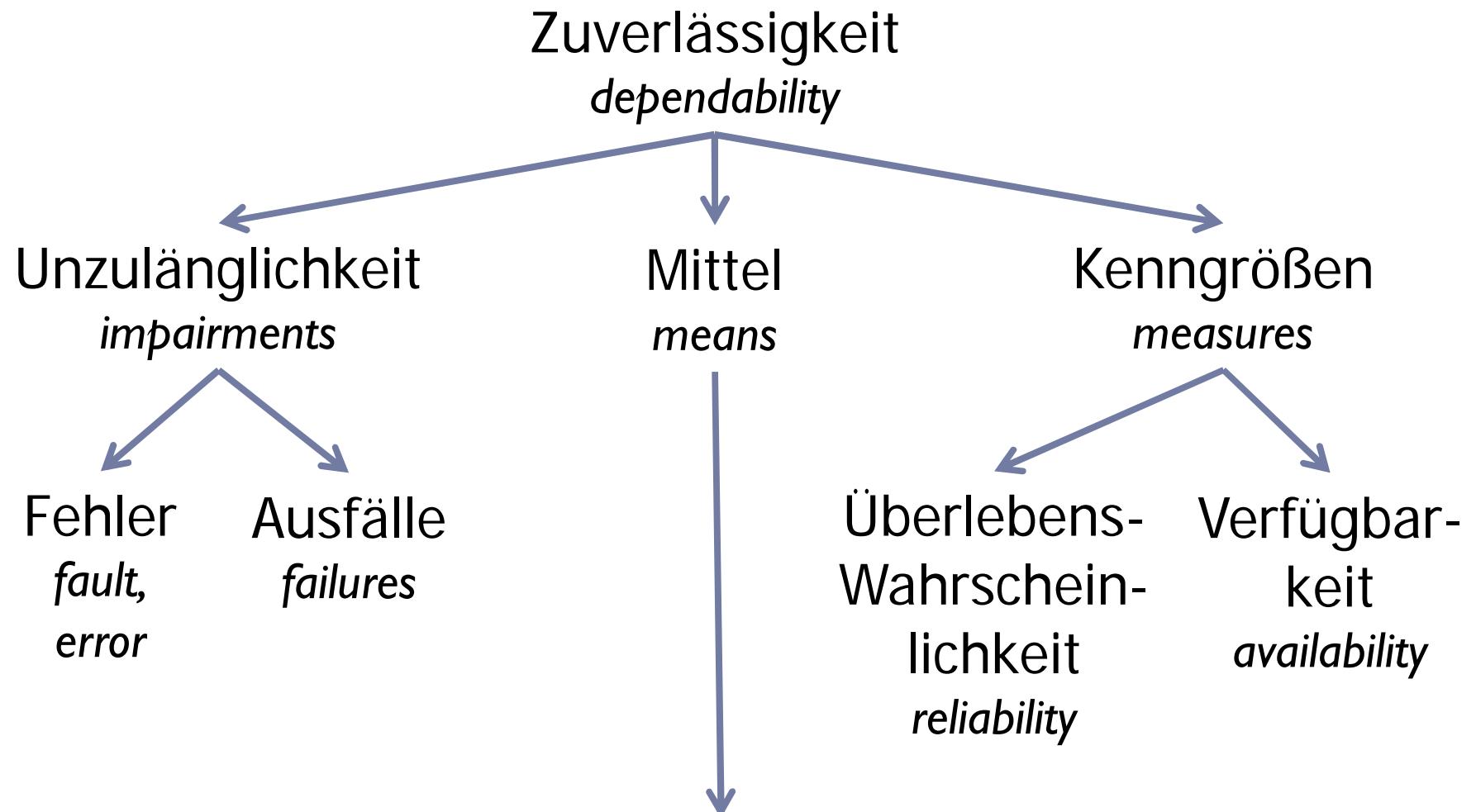
## Warum auf Parallelrechnern / in Clustern?

- ▶ Redundante Hardware-Strukturen eignen sich besonders gut zur Fehlererkennung und dynamischen Fehlerverdeckung

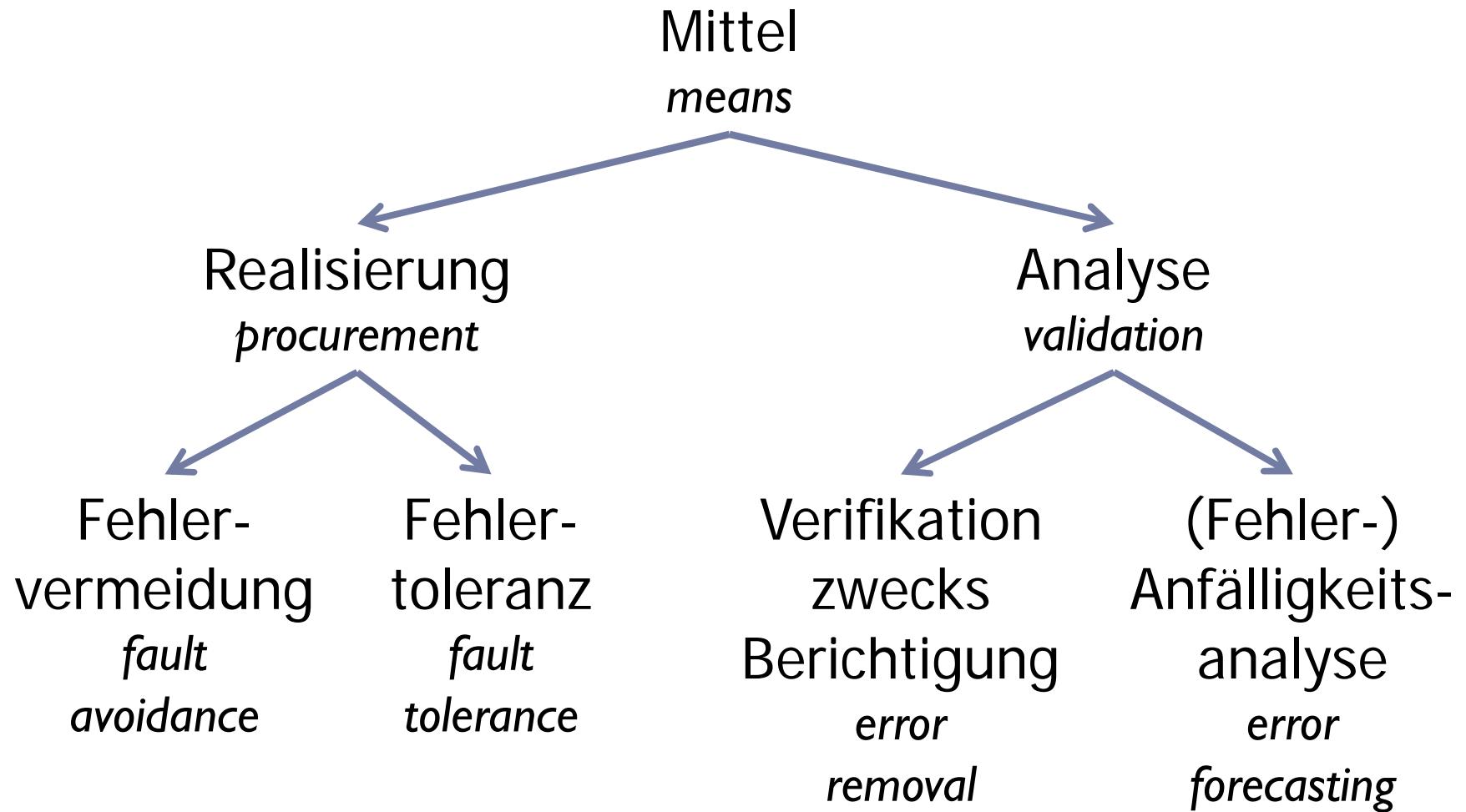
## Warum wirklich auf Parallelrechnern / in Clustern?

- ▶ Redundante Hardware-Strukturen fallen ständig aus ☹

# Begriffsbildung



# Begriffsbildung...



# Zuverlässigkeit

---

## Definition

Fähigkeit eines Systems, die spezifizierte Anwendungsfunktion während eines Einsatzzeitraumes zu erbringen

## Kenngrößen

- ▶ Überlebenswahrscheinlichkeit
- ▶ Verfügbarkeit

# Zuverlässigkeit...

## Überlebenswahrscheinlichkeit

- ▶ Wahrscheinlichkeit  $R(t)$  dafür, daß das System im Zeitintervall  $[0;t]$  fehlerfrei bleibt, wenn es in  $t=0$  intakt war  
[System ohne Reparatur]

## Verfügbarkeit

- ▶ Wahrscheinlichkeit  $A(t)$  dafür, daß das System zum Zeitpunkt  $t$  intakt ist  
[System mit Reparatur]
- ▶  $A(t) = \text{MTBF} / (\text{MTBF} + \text{MTTR})$   
**MTBF:** mean time between failures  
**MTTR:** mean time to repair

# Wichtige Kennzahlen der Verfügbarkeit

---

Man spricht z.B. von den fünf Neunen, gemeint ist 0,99999 % Verfügbarkeit

Was bedeutet das als Ausfallzeit im Jahr?

| Verfügbarkeit | Ausfallzeit pro Jahr |
|---------------|----------------------|
| 0,9           | 876 Stunden          |
| 0,99          | 87 Stunden           |
| 0,999         | 9 Stunden            |
| 0,9999        | 52 Minuten           |
| 0,99999       | 5 Minuten            |

# Unzulänglichkeiten

---

## Beeinträchtigung der Zuverlässigkeit

- ▶ Fehler
  - Unerwünschter (nicht die Spezifikation erfüllender) Zustand des Systems  
(unterscheide noch Fehlerursache/-zustand)
- ▶ Ausfall
  - Ist das Ereignis, daß eine benötigte Funktion nicht erbracht wird

# Fehler

---

Einteilung nach ihrem Entstehen im Lebenszyklus des Systems

- ▶ Entwurfsfehler
- ▶ Herstellungsfehler
- ▶ Betriebsfehler

Einteilung nach der Zeitdauer

- ▶ Permanent
- ▶ Transient
- ▶ Intermittierend (pseudotransient)

# Fehler...

## Entwurfsfehler

Vor Inbetriebnahme des Systems

- ▶ Spezifikationsfehler
- ▶ Implementierungsfehler
- ▶ Dokumentierungsfehler

## Herstellungsfehler

System entspricht nicht der entworfenen Implementierung

- ▶ Z.B. fehlerhafte Materialen oder Werkzeuge

# Fehler...

## Betriebsfehler

In der Nutzungsphase nach Inbetriebnahme

- ▶ Zufällige physikalische Fehler
- ▶ Verschleißfehler
- ▶ Störungsbedingte Fehler
- ▶ Bedienungsfehler
- ▶ Wartungsfehler
- ▶ Absichtliche Fehler (z.B. Sabotage)

Softwarefehler meist Entwurfs- oder Herstellungsfehler

# Fehlermodell

---

Aufgrund der Vielzahl möglicher Fehler:

- ▶ Einschränkung auf bestimmte Gruppen
- ▶ Fehlermodell nennt betroffene Subsysteme und beschreibt Fehlverhalten

Z.B. Fehlermodell für Rechner-Cluster

- ▶ Nur permanente Fehler in Knoten, Leitung, Switches. Zusätzlich evtl. Prozessoren, Speichermodule, Festplatten
- ▶ Fehlerzustände: intakt, defekt
- ▶ Systemzustand dargestellt durch Vektor

# Mittel zur Realisierung von Zuverlässigkeit

---

Ergänzen sich gegenseitig:

- ▶ Fehlervermeidung
- ▶ Fehlertoleranz

Fehlervermeidung (Fehlerintoleranz)

- ▶ Sorgfältige Konstruktion
- ▶ Ausführliche Tests

Nicht vermeidbare Fehler versuchen zu tolerieren

# Fehlertoleranz

---

Einsatz von Redundanz, um im Fehlerfall weiterarbeiten zu können

- ▶ Redundanz
  - ▶ HW-Redundanz (zusätzliche Bauteile)
  - ▶ SW-Redundanz (zusätzliche Programme)
  - ▶ Zeit-Redundanz (zusätzlicher Zeitaufwand)
- ▶ Aktivierung der Redundanz
  - ▶ Statische Redundanz (funktionsbeteiligte Redundanz)
  - ▶ Dynamische Redundanz (Reserveredundanz)

# Fehlertoleranz...

---

## Statische Redundanz

- ▶ Ressourcen ständig in Betrieb
- ▶ Im Fehlerfall direkter Zugriff auf diese Einheiten
- ▶ Z.B. Hamming Codes / Triple Modular Redundancy

## Dynamische Redundanz

- ▶ Aktivierung erst im Fehlerfall
- ▶ Bis zu diesem Zeitpunkt:
  - ▶ Ungenutzte Redundanz (Standby-Systeme)
  - ▶ Fremdgenutzte Redundanz (mit Verdrängung)
  - ▶ Gegenseitig nutzbare Redundanz (Fail-Soft-Systeme)

# Fehlertoleranztechniken

---

## Tolerierung verschiedener Fehlerarten

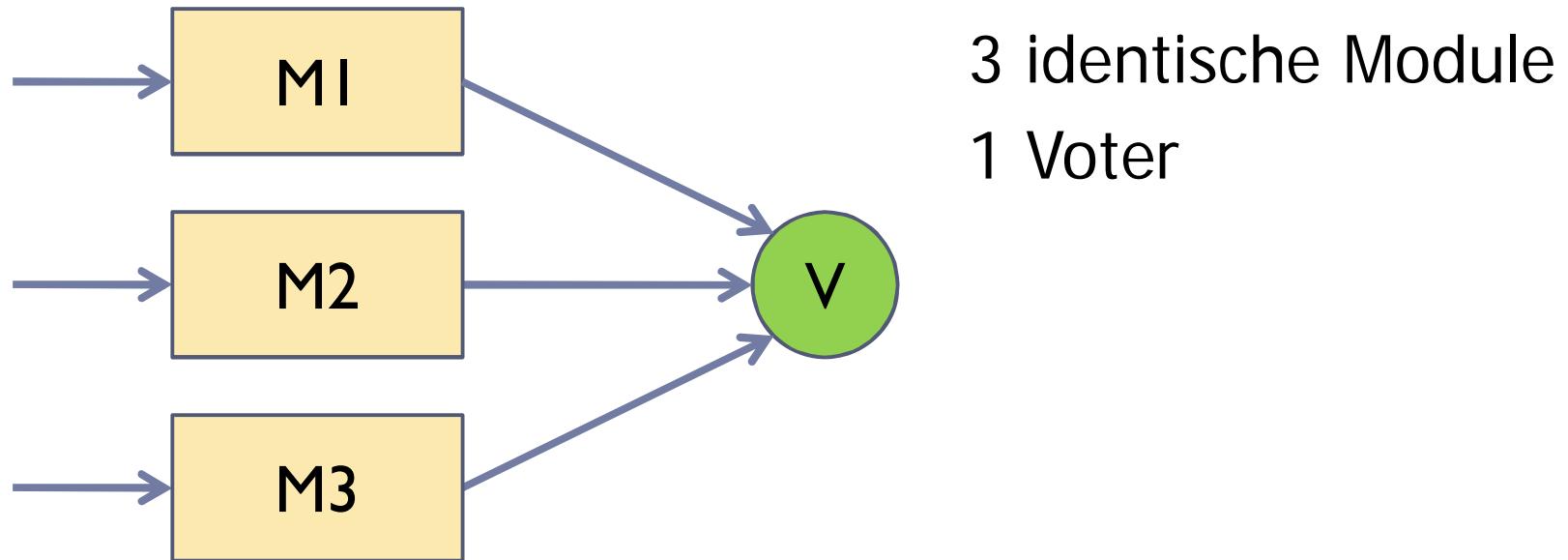
- ▶ HW-Fehler (siehe folgende Folien)
  - ▶ SW-Fehler
- N-Versionen-Programmierung (gut für Mehrprozessorsysteme)

## Betrachtungsebene

- ▶ Hier: Prozessoren, Speicher, Verbindungen
- ▶ Tiefere Ebenen auch möglich: z.B. fehlerkorrigierende Codes

# Statische Redundanz

Wichtigste Technik: Triple Modular Redundancy (TMR)



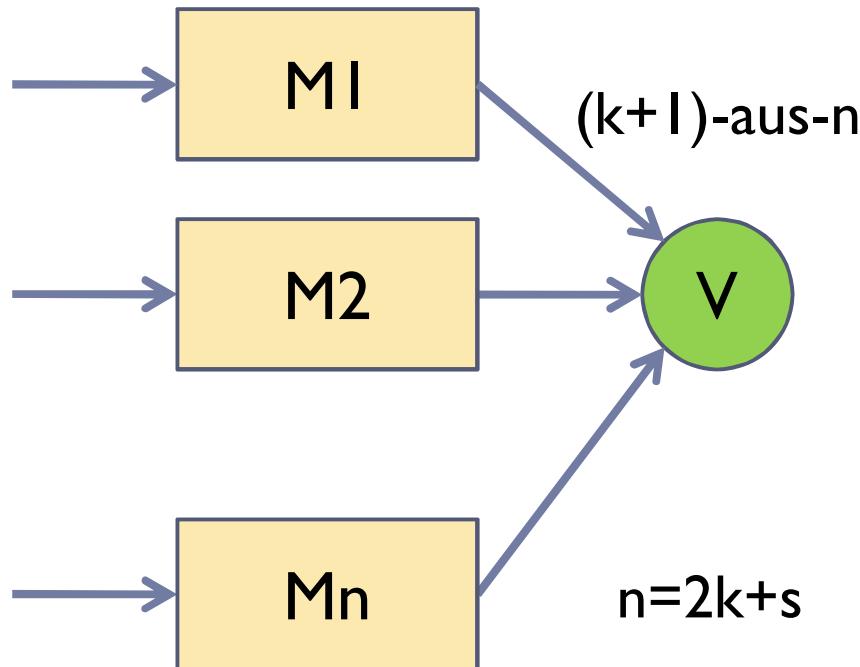
# Statische Redundanz...

## Konzept TMR

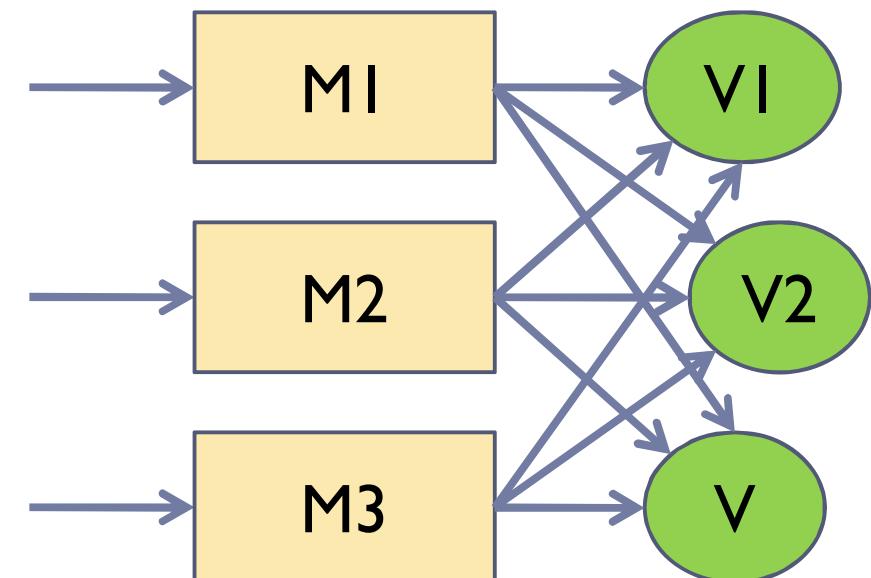
- ▶ Drei identische Module führen nebenläufig gleiche Funktion aus (kann HW oder SW sein)
- ▶ Voter fällt 2-aus-3-Mehrheitsentscheid
- ▶ Erster Ausfall kann toleriert werden
- ▶ Zweiter Ausfall kann noch erkannt werden
- ▶ Voter kann per Hardware oder Software realisiert werden

# Statische Redundanz...

N-Modular-Redundancy  
(NMR)



Tolerierung von Voter-Ausfällen



# Statische Redundanz...

## Vorteile TMR/NMR

- ▶ Toleriert permanente und transiente Fehler
- ▶ Fehlerbehebung kostet keine Zeit

## Nachteile

- ▶ Hoher Aufwand in HW oder SW begrenzt Einsatzbereich

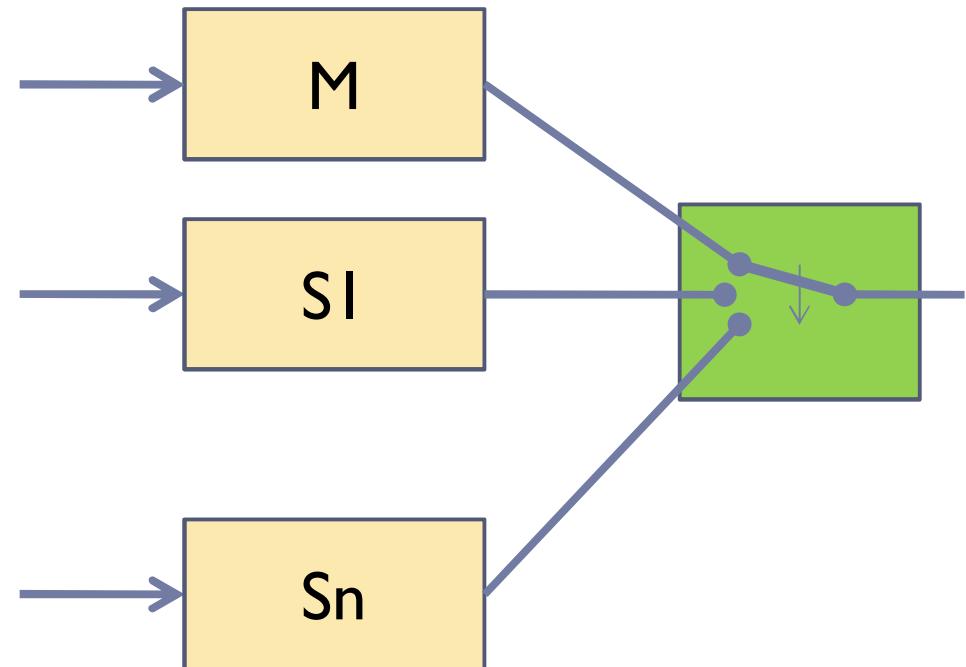
# Dynamische Redundanz

## Standby-Systeme

Zusätzlich zu den Systemkomponenten noch Reservekomponenten

### Nachteile:

- ▶ Kein verzögerungsfreies Umschalten
- ▶ Ungenutzte Ressourcen (cold/hot standby)



# Dynamische Redundanz...

## Fail-Soft-Systeme (Graceful Degradation)

- ▶ Mehrfach vorhandene Subsysteme werden als Redundanzen genutzt (Eigenredundanz)
  - ▶ Defekte Systemkomponenten werden deaktiviert
  - ▶ Mit verminderter Leistung kann die Aufgabe weiterbearbeitet werden
- 
- ▶ Ausgangsbedingungen und Verfahren durch ein Rechner-Cluster optimal abgedeckt

# Dynamische Redundanz...

## Vorgehensweise bei beiden Verfahren

- ▶ Fehlererkennung und –lokalisierung (Fehlerdiagnose)
- ▶ Rekonfiguration
- ▶ Fehlerbehandlung (Recovery)

# Dynamische Redundanz...

## (1) Fehlerdiagnose

- ▶ Aufgabe: Erkennen defekter Knoten
- ▶ Ebene: Komponenten- und Systemebene
- ▶ SRU (Smallest Replacable Unit)
  - Ebene der Rekonfiguration
  - Keine Lokalisation auf Komponentenebene
- ▶ Fehlererkennung mit Einprozessormethoden
  - Fehlererkennende Codes, Selbsttestprogramme
- ▶ Zur Rekonfiguration noch Diagnose auf Systemebene

# Dynamische Redundanz...

---

- ▶ Systemdiagnose
  - ▶ Fremddiagnose
    - ▶ Separater Prozessor für Diagnose (und i.a. auch für Rekonfiguration und Recovery)
    - ▶ Nachteile
      - Diagnose- und Wartungsprozessor als perfekt zuverlässig angenommen
      - Zusätzliche Prozessortypen (Heterogenität)
  - ▶ Eigendiagnose
    - ▶ Zentral: Testrunde in bestimmten zeitlichen Abständen  
Koordinatoreinheit ermittelt defekte Komponenten  
Problem: Wahl des Koordinatorknotens
    - ▶ Dezentral: Alle intakten Einheiten haben Diagnosebild  
Problem: Konsistenz der Einzelbilder

# Dynamische Redundanz...

## (2) Rekonfiguration

- ▶ Bildet aus den intakten Einheiten ein lauffähiges System
- ▶ Ausführung
  - ▶ Externer Wartungsprozessor
  - ▶ Zentraler Koordinator (unkritisch)
  - ▶ Dezentral
- ▶ Schwierigkeit
  - ▶ Suche einer neuen effizienten Abbildung der SW-Komponenten auf die HW-Komponenten

# Dynamische Redundanz...

## (3) Fehlerbehebung

- ▶ Versetzt das System in einen korrekten Zustand
- ▶ Setzt die Anwendung mit korrekten Daten wieder auf (Wiederanlauf)
- ▶ Methoden
  - ▶ Rückwärtsfehlerbehebung
  - ▶ Vorwärtsfehlerbehebung

# Dynamische Redundanz...

- ▶ Rückwärtsfehlerbehebung
  - ▶ System zurücksetzen in früheren konsistenten Zustand (*rollback*)
  - ▶ Rücksetzpunkte (*recovery points*)
  - ▶ Abspeicherung in Haupt- oder Hintergrundspeicher
  - ▶ Zeitintervalle durch analytische Methoden
- ▶ Vorwärtsfehlerbehebung
  - ▶ Zusätzliche Operationen bringen das System in einen korrekten Zustand
  - ▶ Problem: Genaue Kenntnis der Anwendung notwendig

# Dynamische Redundanz...

## Bewertung

- ▶ Bei dynamischer Fehlertoleranz kein unterbrechungsfreier Betrieb
  - ▶ Für „normale“ Anwendungen akzeptabel, nicht jedoch bei Echtzeitanwendungen
- ▶ Je nach Variante können aber alle Komponenten produktiv genutzt werden

# Quantitative Bewertung

---

Überlebenswahrscheinlichkeit  $R(t)$

Exponentiell verteilte Lebensdauer  $R(t) = e^{-\lambda t}$

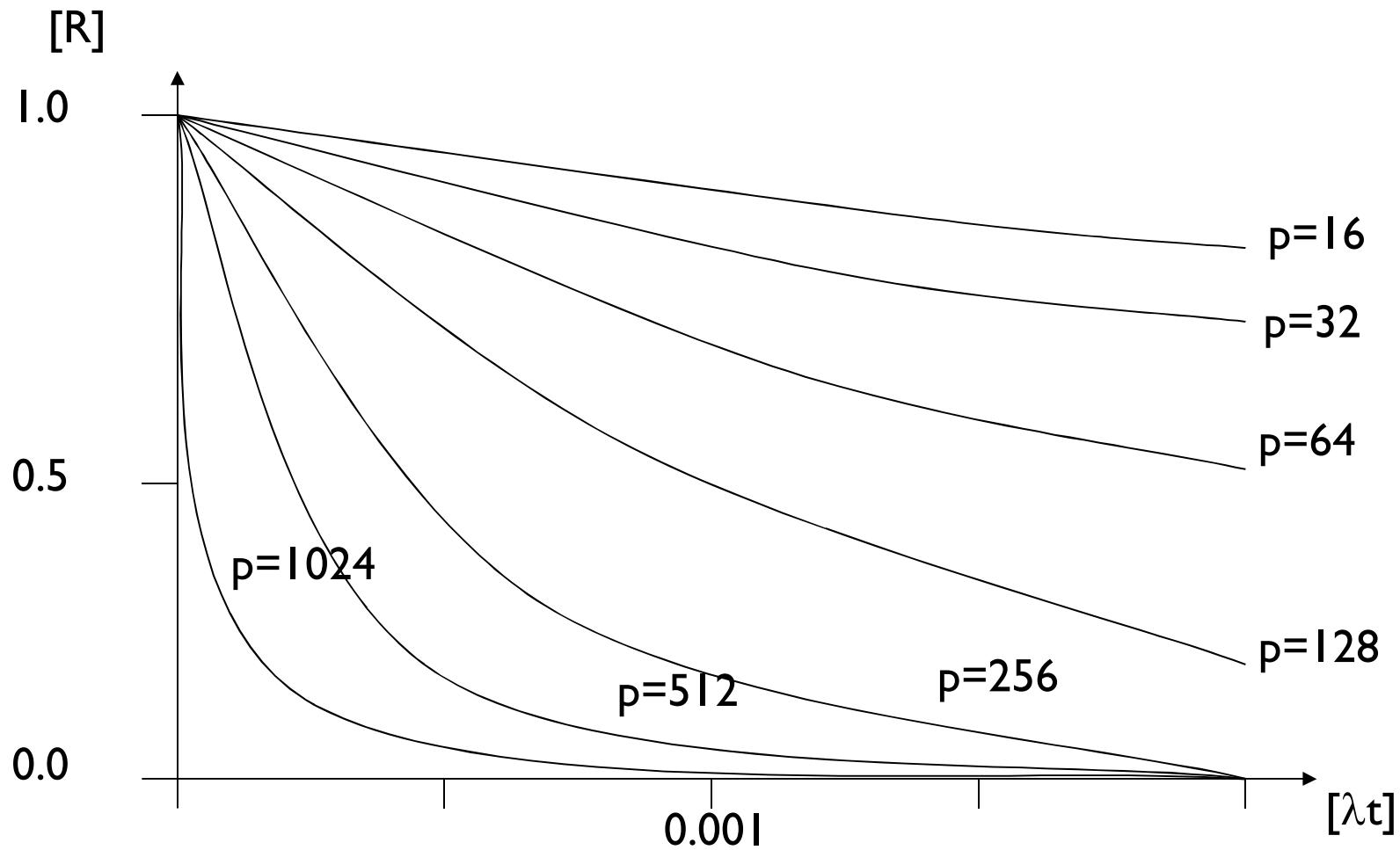
MTTF (mean time to failure) =  $1/\lambda$

Zunächst das System ohne Fehlertoleranz:

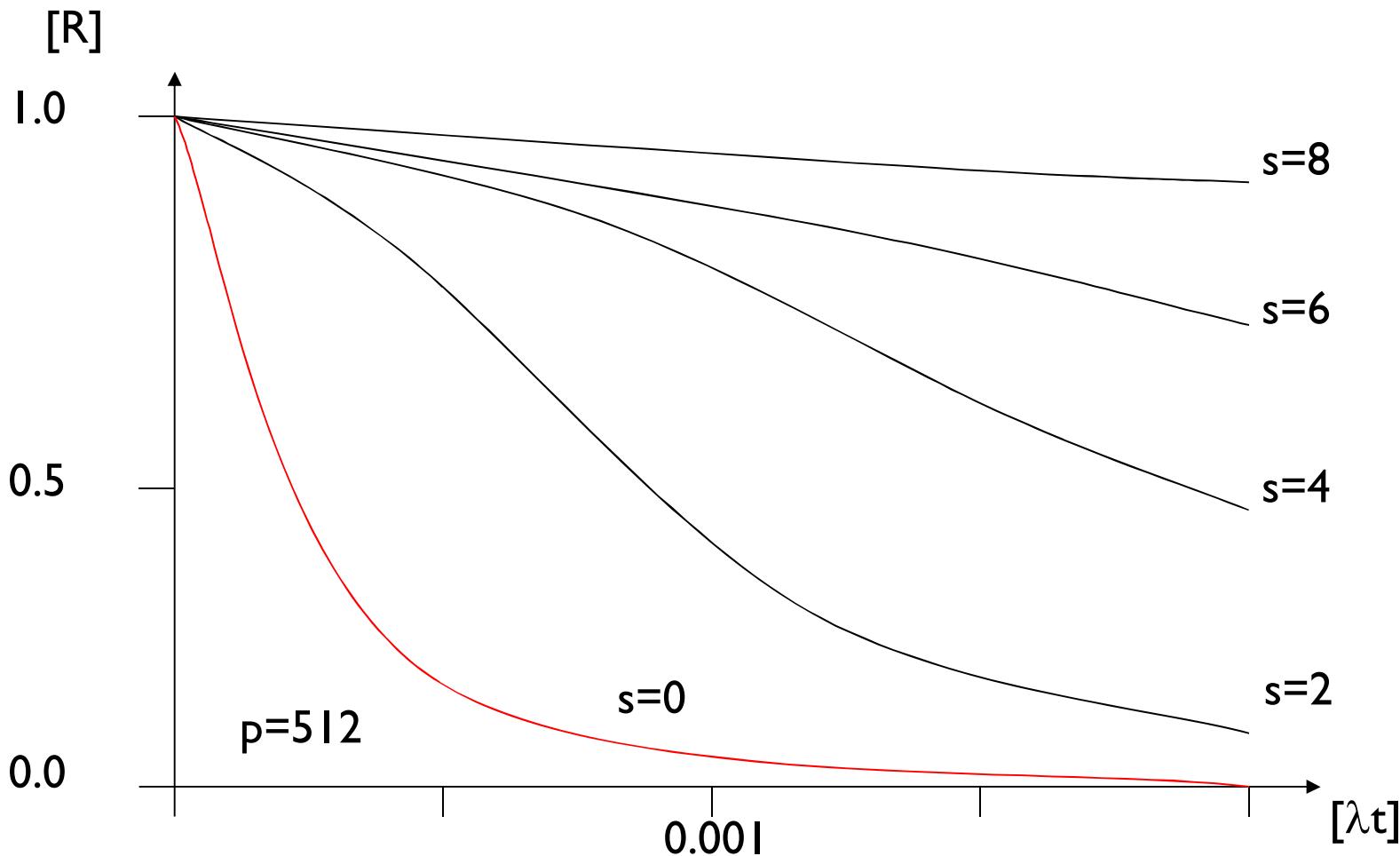
$$R(t)p = e^{-p\lambda t}$$

Anzahl  $p$  PMU's (*processor memory unit*)

# Quantitative Bewertung...



# Quantitative Bewertung...



# Quantitative Bewertung...

System kann bis zu s ausgefallene PMU's tolerieren:

$$R_{(p-s) \text{ aus } p}(t) = \sum_{i=0}^s c^i (1-R(t))^i R(t)^{p-i}$$

$c = P(\text{Fehler wird behandelt} \mid \text{Fehler tritt auf})$

(coverage factor - Überdeckungsfaktor)

# Systeme in der Praxis

---

- ▶ Parallelrechner und Hochleistungsrechnen
  - ▶ Nahezu keine Realisierungen
  - ▶ Gründe
    - ▶ Systeme zu aufwendig
    - ▶ Nutzen zu gering, da Schäden zu gering
- ▶ Parallelrechner und kommerzielle Anwendungen
  - ▶ High-Availability-Computing
    - ▶ Hochverfügbare Systeme
    - ▶ Normalerweise keine parallelen Programme
    - ▶ Bereich Datenbanken, Systemsteuerungen u.ä.

# Systeme in der Praxis...

---

- ▶ High-Availability (HA) Linux Project
  - ▶ Definition von Verfahren, um Parallelrechner ausfallsicher zu machen
  - ▶ Cluster hier: Menge von HW-Komponenten, die als wechselseitige Redundanz dienen
- ▶ Linux High Availability HOWTO
- ▶ Viele kommerzielle HA-Cluster

# Fehlertoleranz

## Zusammenfassung

- ▶ Fehlertoleranz ist ein wichtiges Thema bei allen Systemen, die aus sehr vielen Einzelkomponenten bestehen
- ▶ Wir unterscheiden im allgemeinen Systeme mit und ohne Reparatur
- ▶ Wir unterscheiden verschiedene Klassen von Fehlern
- ▶ Ein Fehlermodell beschreibt das System analytisch
- ▶ Redundanz ist notwendig, um Fehler tolerierbar zu machen
- ▶ Wir unterscheiden statische und dynamische Redundanzverfahren
- ▶ Statische Redundanz: Triple-Modular-Redundancy-Verfahren
- ▶ Dynamische Redundanz: Fail-Soft-Verfahren
- ▶ In der Praxis Anwendungen nur im Bereich des sogenannten High-Availability-Computing

# Grid- und Cloud-Computing

---

- ▶ Motivation für Grid-Computing
  - ▶ Grid-Computing
  - ▶ Anwendungsbereiche
  - ▶ Benutzung / Dienste
  - ▶ Projekte
- 
- ▶ Motivation für Cloud-Computing
  - ▶ Varianten
  - ▶ Cloud-Computing und HPC

# Grid- und Cloud-Computing

## Die zehn wichtigsten Fragen

- ▶ Welche prinzipiellen Probleme gibt es bei der Nutzung von Clustern und Parallelrechnern?
- ▶ Wie faßt man solche Architekturen zu größeren Einheiten zusammen?
- ▶ Was bezeichnete man als Internetcomputing und Metacomputing?
- ▶ Was ist die Grundidee des Grid-Computing?
- ▶ Welche Fragestellungen finden wir hier?
- ▶ Welche Klassen von Anwendungen gibt es hier?
- ▶ Was ist die Grundidee des Cloud-Computing?
- ▶ Welche Dienste werden angeboten?
- ▶ Welche Vor- und Nachteile erkennen wir hier?
- ▶ Wie tauglich ist Cloud-Computing für HPC?

# Motivation für Grid-Computing

---

## Ausgangssituation

- ▶ Viele Cluster und Parallelrechner in verteilten Rechenzentren in der ganzen Welt
- ▶ Viel Knowhow in unterschiedlichen Gruppen verteilt

## Aber:

- ▶ Ressourcen sind oft nicht da, wo die Benutzer sind
- ▶ Das Wissen ist nicht da, wo der Benutzer ist

# Motivation für Grid-Computing...

---

## Situation bei Clustern und Parallelrechnern

- ▶ Verschiedene Hardware-Architekturen
- ▶ Verschiedene Hersteller
- ▶ Verschiedene Betriebssysteme und Basis-SW
- ▶ Verschiedene Benutzungsmodelle (Stapelbetrieb, interaktiv)
- ▶ Verschiedene Formen der Systemverwaltung
- ▶ Verschiedene Formen der Datenverwaltung
- ▶ Verschiedene Sicherheitsmechanismen

# Motivation für Grid-Computing...

---

## Konsequenzen für den Benutzer

- ▶ Arbeitet nur mit vertrautem Rechner
- ▶ Benutzt nur Rechner mit passenden Ressourcen
- ▶ Vermeidet neue Rechner

## Konsequenzen für den Systembetreiber

- ▶ Teure Ressourcen nicht optimal genutzt
- ▶ Probleme nicht optimal gelöst
- ▶ Lösbare Probleme zum Teil gar nicht gelöst

# Motivation für Grid-Computing...

---

## Neue Benutzungsmethodik

- ▶ Wir betrachten das Cluster, den Parallelrechner oder das Rechenzentrum als „einzelnen“ Rechner
- ▶ Zusammenfassen mehrerer solcher „Rechner“ zu einem neuen „Rechnersystem“
- ▶ Programme werden auf diesem „Rechnersystem“ ausgeführt

## „Rechnersystem“

- ▶ Früher vernetzte Einzelrechner
- ▶ Heute vernetzte Hochleistungsrechner

# Motivation für Grid-Computing...

---

Nutzungsmethode der neuen „Rechnersysteme“ in Analogie zu früher

- ▶ Zuweisung eines Jobs zu einer freien Ressource  
Einzeljob nicht parallel bzgl. neuen Rechnersystems  
Bis ca. 1999 genannt: Internetcomputing  
Frühes Beispiel: seti@home
- ▶ Parallelisierung eines Jobs über mehrere Ressourcen  
Paralleler Job bzgl. neuen Rechnersystems  
Bis ca. 1999 genannt: Metacomputing

# Motivation für Grid-Computing...

---

## Nutzungsmethode...

- ▶ Hardware- und Software-Konzepte  
Vernetzung, Sicherheit, Ein-/Ausgabe usw.
- ▶ Programmkonstruktion  
Individuelle Jobs für Einzelsysteme oder Verbundsysteme  
(aber intern alles parallelisiert)
- ▶ Nutzungsmethodik  
Stapelbetrieb und/oder interaktiver Betrieb

# Motivation für Grid-Computing...

---

Alles schon bekannt, aber jetzt eine Abstraktionsstufe höher

- ▶ Ressourcenverwaltung, Internetcomputing
  - Die einzelne Anwendung ist parallelisiert und läuft an einem auswählbaren Ort
  - Anbindung über Web-Frontend
- ▶ Metacomputing
  - Die Einzelanwendung ist parallelisiert für verschiedene parallele Umgebungen und läuft parallel an verschiedenen Orten

Neuer Begriff seit 1998: GRID-Computing

# Eine wichtige Ressource

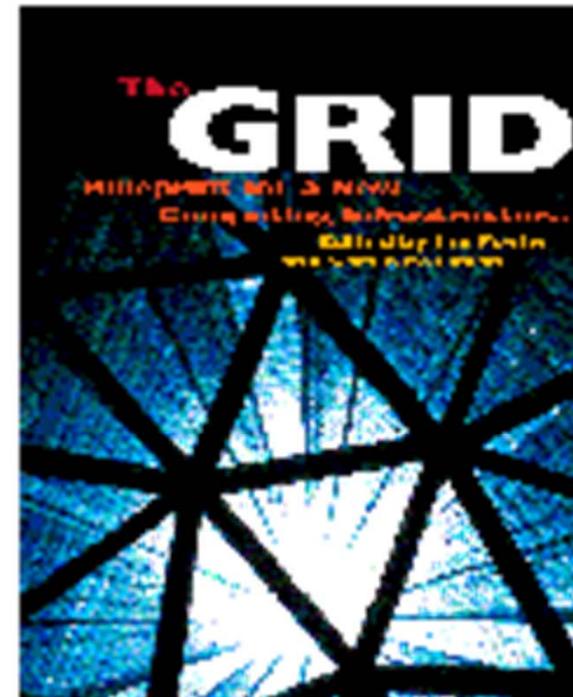


## The Grid: Blueprint for a New Computing Infrastructure

I. Foster, C. Kesselman (Eds), Morgan Kaufmann, 1999

- Available July 1998;  
ISBN 1-55860-475-8
- 22 chapters by expert  
authors including Andrew  
Chien, Jack Dongarra, Tom  
DeFanti, Andrew  
Grimshaw, Roch Guerin,  
Ken Kennedy, Paul  
Messina, Cliff Neuman, Jon  
Postel, Larry Smarr, Rick  
Stevens, and many others

*"A source book for the history  
of the future" -- Vint Cerf*



<http://www.mkp.com/grids>

Introduction 34

# Grid-Computing

---

## Idee

„Rechenleistung an jedem Ort“

Analogie zum „power grid“, dem Stromnetz

Einspeisung an verschiedene Stellen, Nutzung an beliebigen anderen Stellen

- ▶ Begriff des „Grid“ mittlerweile sehr umfassend
- ▶ Kein Konsens über Anwendung und Konzepte

# Grid-Computing...

---

Warum kommt das gerade jetzt auf?

- ▶ Zugriff auf Ressourcen erwünscht, die lokal nicht vorhanden sind; Replikation zu teuer
- ▶ Vernetzung immer leistungsfähiger und gleichzeitig immer billiger
- ▶ Erfahrung mit dem parallelen Rechnen brachte Erkenntnis zum Thema Verteilung von Programmen

# Grid-Computing...

## Fragestellungen

- ▶ Auswahl der Rechnerressourcen
- ▶ Auswahl der Parallelisierung
- ▶ Auswahl der Sprachkonzepte
- ▶ Sicherheitsaspekte
- ▶ Lastausgleich
- ▶ Fehlertoleranz

Insgesamt nichts neues!

Jetzt alles in großen Dimensionen betrachtet



# Grid-Computing...

---

## Die Ziele

- ▶ Weltweite Nutzung freier Rechenleistung
- ▶ Aggregation hoher Rechenleistung
- ▶ Einfacher Zugriff auf Rechenleistung
- ▶ Bewältigung großer Aufgabenstellungen
- ▶ Kollaboration zwischen Orten

# Grid-Computing...

## Voraussetzungen

- ▶ Hochgeschwindigkeitsvernetzung
  - Gigabit-Vernetzung, ATM-Vernetzung
  - Satellitenkopplung
- ▶ Geeignete Protokolle auf verschiedenen Ebenen

## Problem

- ▶ Kommunikationslatenz
  - Beim Grid-Computing deutlich höher, d.h. nur grobgranulare Parallelisierung gewinnbringend

# Anwendungsbereiche

---

## Vier wichtige Gebiete

- ▶ *Desktop Supercomputing*  
Zugang zu hoher Rechenleistung von überall aus
- ▶ *Smart Instruments*  
Verbindet Radioteleskope, Kernbeschleuniger, Tomographen usw. mit geeigneten Hochleistungsrechnern
- ▶ *Collaborative Environments*  
Verbindet Benutzungsumgebungen miteinander und mit Supercomputern zu Simulationsläufen usw.
- ▶ *Distributed Supercomputing*  
Erzielt Summe der Leistungen der Einzelrechner

# Umgebungscharakteristika

---

- ▶ Wachsende Systemgrößen
  - Jedoch meist nur noch Teilsysteme verwendet
- ▶ Heterogenität auf allen Ebenen
  - HW, Betriebssysteme, Sprachen, Compiler usw.
- ▶ Wechselnde Strukturen
  - Sowohl des Rechnersystems als auch des Programmsystems
- ▶ Dynamisches Verhalten
  - Gemeinsam genutzte Ressourcen schwer kontrollierbar
- ▶ Verschiedene Administrationsprogramme
  - Authentifizierung, Autorisierung usw.

# Benutzung / Dienste

---

Das Grid ist nicht nur eine Ansammlung von Ressourcen sondern auch von Diensten

- ▶ Scheduling
- ▶ Ressourcen-Verwaltung
- ▶ Sicherungspunkt-Verwaltung
- ▶ Sicherheit und Abrechnung
- ▶ Weitere Dienste...

# Benutzung / Dienste...

## Scheduler

- ▶ Job-Scheduler
  - ▶ Optimierung auf Job-Durchsatz
- ▶ Ressourcen-Scheduler
  - ▶ Optimierung auf beste Ressourcen-Nutzung
- ▶ Anwendungs-Scheduler
  - ▶ Optimierung auf z.B. minimale Ausführungszeit

## Fragestellungen

- ▶ Die üblichen:  
Was soll wann wo ausgeführt werden?

# Benutzung / Dienste...

Im Grid heterogene Leistungs-Charakteristik

- ▶ Software, Hardware, Vernetzung
- ▶ Mitbenutzung durch andere Nutzer
- ▶ Keine zentrale Kontrolle über alle Ressourcen möglich
- ▶ Verfügbare Ressourcen sind dynamisch
- ▶ Leistung der Anwendung  $\neq$  Leistung des Systems

Scheduling hier maximal schwer

- ▶ Nur heuristische Verfahren sinnvoll

# Benutzung / Dienste...

## Sicherheit und Abrechnung

- ▶ Bekannte Mechanismen auf höherer Ebene
  - ▶ Kerberos
  - ▶ Public-Key-Systeme
  - ▶ Zertifikate
  - ▶ Firewalls
- ▶ Zusätzliche Probleme
  - ▶ Sicherheitsvereinbarungen zwischen Organisationen
  - ▶ Verteilter Zugang zum System
  - ▶ Verteilte Abrechnung

# Infrastrukturen

---

## Wichtigste Infrastruktur

- ▶ Globus Toolkit (Argonne National Laboratory)

Globus-Alliance: [www.globus.org](http://www.globus.org)

## Das Konzept

- ▶ Globus stellt ein Toolkit zur Verfügung, das Basismechanismen bereitstellt (Kommunikation, Autorisierung usw.)
- ▶ Hierauf lassen sich höhere Dienste des Grid-Computing abstützen (Programmierwerkzeuge, Scheduler usw.)
- ▶ Langfristziel: AWARE (*Adaptive Wide Area Resource Environment*)

# Deutsche Grid-Initiative (D-Grid)

The screenshot shows the homepage of the D-Grid Initiative. At the top, there's a navigation bar with links for Startseite, News, Veranstaltungen, Kontakt, Sitemap, and language switches (German and English). Below the header, there's a main content area with a sidebar on the left containing links for Startseite, Über D-Grid, Integrationsprojekt, Projektübersicht, D-Grid im eScience, Service, and Veranstaltungen. The main content area features sections for "Die Deutsche Grid-Initiative (D-Grid)", "D-Grid 1, 2005-2008", and "D-Grid 2, 2007-2010". A sidebar on the right lists "Veranstaltungen" (with a sub-section for AHM III 2010, Dresden), "Weitere Veranstaltungen", and "Aktuelles". A footer at the bottom includes a "GEFÖRDERT VOM" section featuring the German flag and the text "Bundesministerium für Bildung und Forschung".

## Die Deutsche Grid-Initiative (D-Grid)

Als gemeinsame Initiative mit der deutschen Wissenschaft und Wirtschaft fördert das Bundesministerium für Bildung und Forschung (BMBF) den Aufbau des D-Grids.

Die ersten D-Grid-Projekte starteten im September 2005 mit dem Entwickeln einer verteilten, integrierten Ressourcenplattform für Hochleistungsrechnen, grosse Mengen von Daten und Informationen und entsprechende Dienstleistungen. Der Aufbau und Betrieb dieser Grid-Plattform erfolgt in mehreren Stufen:

### D-Grid 1, 2005-2008

IT-Dienste für die Wissenschaftler, entwickelt und implementiert von erfahrenen Grid-Forschern und Anwendern. Sogenannte Grid-Communitys testen diese globale Dienste-Infrastruktur inzwischen mit ihren rechen- und daten-intensiven Anwendungen aus den Gebieten der Hochenergiephysik, Astrophysik, alternative Energien, Medizin, Klimaforschung, Ingenieuranwendungen und Geisteswissenschaften.

### D-Grid 2, 2007-2010

IT-Dienste für Wissenschaft und Industrie, die auf der D-Grid-Integrationsschicht aufbauen, wie zum Beispiel Bauindustrie, Finanzwirtschaft, Automobilindustrie, Luft- und Raumfahrt, Betriebsinformations- und Betriebsmittel-Systeme und geographische Datenverarbeitung.

### Veranstaltungen

#### AHM III 2010, Dresden

Im Rahmen des All-Hands-Meeting trafen sich die verschiedenen D-Grid-Projekte, um sich miteinander bekannt zu machen und für einen regen Informationsaustausch innerhalb der Community zu sorgen.

[mehr...]  
[Bildergalerien...]

#### Weitere Veranstaltungen

Informationen zu weiteren Events und Workshops entnehmen Sie bitte dem Veranstaltungskalender

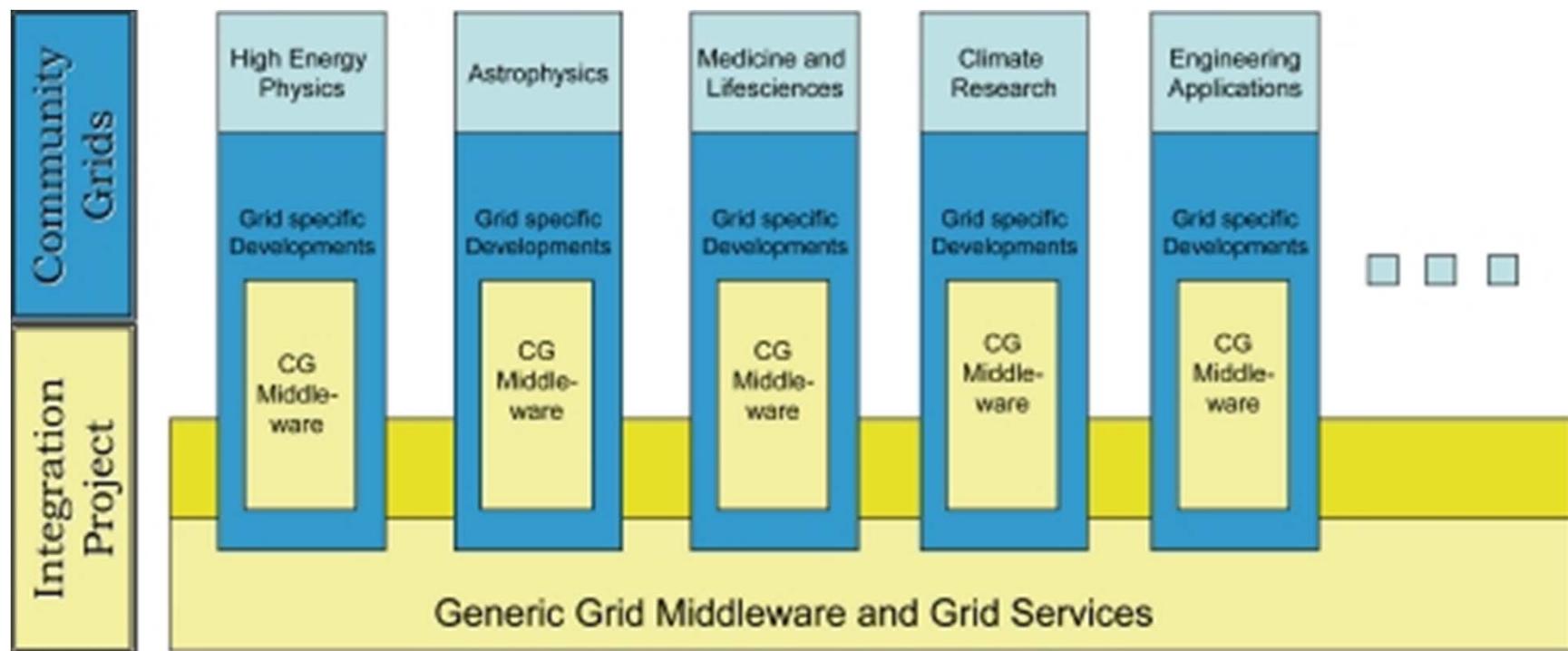
[mehr...]

#### Aktuelles

30.11.2009

# Deutsche Grid-Initiative (D-Grid)...

## Integration Project + Community Grids



# Gauss Centre for Supercomputing

The screenshot shows the GCS homepage. At the top left is the GCS logo with the text "Gauss Centre for Supercomputing" and a small European Union flag. To the right is a photograph of server racks. On the far right is a "Print" link. The main content area features the heading "Supercomputing at the Leading Edge" and "A Key Technology for Science and Engineering". Below this, a text block states: "The Gauss Centre for Supercomputing (GCS) provides the most powerful high-performance computing infrastructure in Europe." Another text block below says: "The GCS is the alliance of the three German national supercomputing centres:". To the left of the main content is a sidebar with a blue background containing a navigation menu:

- Homepage
- About GCS
- Resources
- Science
- Publications
- Events
- Links
- News
- Imprint

Below the sidebar, the three alliance partners are listed with their logos:

- John von Neumann-Institut für Computing  
Jülich 
- Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften  
Garching 
- Höchstleistungsrechenzentrum Stuttgart 

# PRACE



Home      Administrator log in

**Welcome to PRACE**

The Partnership for Advanced Computing in Europe, PRACE, is a unique persistent pan-European Research Infrastructure for High Performance Computing (HPC). PRACE is a project funded in part by the EU's 7th Framework Programme.

Supercomputers are indispensable tools for solving the most challenging and complex scientific and technological problems through simulations. To remain internationally competitive, European scientists and engineers must be provided with leadership-class supercomputer systems.

PRACE forms the top level of the European HPC ecosystem. The partnership was established through the close collaboration of the European countries that prepared the legal, financial, and technical basis in a project funded in part by the European Commission. PRACE provides Europe with world-class systems for world-class science and strengthens Europe's scientific and industrial competitiveness. PRACE will maintain a pan-European HPC service consisting of up to six top of the line leadership systems (Tier-0) well integrated into the European HPC ecosystem. Each system will provide computing power of several Petaflop/s (one quadrillion operations per second) in midterm. On the longer term (2019) Exaflop/s (one quintillion) computing power will be targeted by PRACE. This infrastructure is managed as a single European entity.

**News** [more...](#)

- » Recruitment of the PRACE Director 2010-06-30
- » PRACE calls for One Year Project Grants on Europe's fastest Computer 2010-06-15
- » PRACE Research Infrastructure inaugurated: World-class Supercomputing Service for European Science 2010-06-09
- » Press release by the EC: Digital Agenda: Commission welcomes launch of supercomputing infrastructure for European researchers 2010-06-09
- » Presentations from ISC'10 available 2010-06-01

**Events** [more...](#)

- » PRACE 1IP Kick-Off Meeting, August 30-31, Garching, Germany
- » ICT 2010, September 27-29,

**PRACE newsletter**

Your e-mail address

HTML  
 Text

# Motivation für Cloud-Computing

---

Weltweit existieren sehr viele Rechner- und Speichersysteme und auf ihnen viele Softwaresysteme

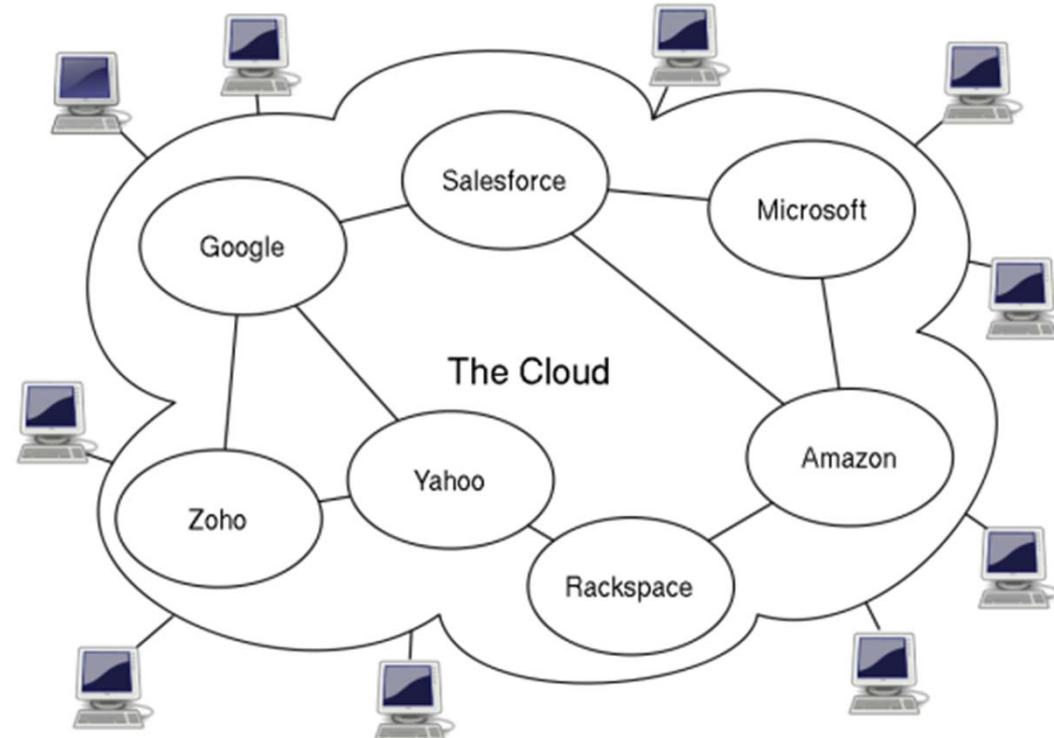
Nutzer an beliebigen Orten könnten vielleicht über eine Vernetzung auf diese Ressourcen zugreifen

Hätte für Anbieter und Nutzer einige Vorteile

# Cloud-Computing

„Cloud-Computing“ = Rechnen in der Wolke

Wolke? Abgeleitet vom Wolkensymbol für das Internet



# Charakter des Cloud-Computing

---

Konzepte und Systeme werden über das Netz zur Verfügung gestellt  
Bezahlung je nach Nutzung

- ▶ Infrastructure as a Service (IaaS)
  - ▶ Rechenkapazität, Speicherkapazität bereitstellen
- ▶ Software as a Service (SaaS)
  - ▶ Nutzung fertiger Programmpakete
- ▶ Platform as a Service (PaaS)
  - ▶ Programmierumgebungen bereitstellen
- ▶ Archive as a Service (AaaS)
  - ▶ Archivierungsdienste bereitstellen

# Vorteile / Nachteile

---

## Vorteile

- ▶ Aus Nutzersicht
  - ▶ Keine Investitionskosten, Wartungskosten usw.
  - ▶ Flexibler Einkauf von Diensten
  - ▶ Billige Abwicklung von Lastspitzen
- ▶ Aus Betreibersicht
  - ▶ Effizientes Anbieten von Diensten
  - ▶ Effizientes Verwalten der angebotenen Dienste

## Nachteile

- ▶ Aus Nutzersicht
  - ▶ Datensicherheit
  - ▶ Anbieterbindung

# Abgrenzungen

---

## Grid-Computing

- ▶ Gemeinschaftliche Nutzung der gemeinsamen Ressourcen, meist im Bereich des Hochleistungsrechnens
  - ▶ Cloud hat wenige einzelne Anbieter und die sind zentral gesteuert

## Peer-to-Peer-Computing

- ▶ P2P: Verteilung von Rechenlast auf viele Rechner
- ▶ Cloud: Verlagerung von Rechenlast auf andere Rechner

# Cloud-Dienstanbieter



Bei AWS Management Console anmelden



Legen ↗

▼ AWS

▼ Produkte

▼ Entwickler

▼ Community

▼ Sup

Produkte & Dienstleistungen ▾

## Amazon Elastic Compute Cloud (Amazon EC2)

### Amazon EC2 Details

- [EC2 Überblick](#)
- [Häufig gestellte Fragen](#)
- [EC2 Preisgestaltung](#)
- [Amazon EC2 SLA \(Englisch\)](#)
- [Amazon EC2-Instanztypen](#)
- [Kaufoptionen für EC2-Instanzen](#)
- [Reserved Instances](#)
- [Spot Instances](#)

Amazon Elastic Compute Cloud (Amazon EC2) ist ein Webservice, der die Anpassung der Rechenkapazität in der Cloud ermöglicht. Mit diesem Service wird die Web-Skalierung der Rechenleistung für Entwickler einfacher.

Mit der einfachen Web-Service-Oberfläche von Amazon EC2 können Sie mühelos Kapazität erhalten und konfigurieren. Sie ermöglicht Ihnen die vollständige Kontrolle über Ihre Rechenressourcen sowie die Ausführung auf der bewährten Rechenumgebung von Amazon. Amazon EC2 verringert die zum Erwerben und Booten neuer Server-Instanzen benötigte Zeit auf wenige Minuten. So können Sie die Kapazität entsprechend den Änderungen Ihrer Rechenanforderungen schnell in beide Richtungen skalieren. Indem Sie nur für die Kapazität zahlen, die Sie auch tatsächlich nutzen, verändert Amazon EC2 die wirtschaftlichen Rahmenbedingungen von Rechenoperationen. Amazon EC2 bietet Entwicklern die Tools, um ausfallsichere Anwendungen zu erstellen und diese von üblichen Fehlerszenarien zu isolieren.

z

# Cloud und HPC ?

Email not displaying correctly? [View it in your browser](#)



## HPC *In the Cloud*

### Weekly Update

Dedicated to Covering Enterprise & Scientific Large Scale Cloud Computing  
Jun 29, 2010

#### Sponsored Content

Windows + Supercomputing = Accelerated Results

Accelerating your workloads. Read the [case studies](#) now and see the results.

#### Feature Articles

#### Will Public Clouds Ever Be Suitable for HPC?

Since the primary consideration in HPC is performance, it stands to reason that it's no easy task to convince the scientific computing community that the public cloud is a viable option. Accordingly, a handful of traditional HPC vendors are refining their solutions to bridge the cloud performance chasm that exists in EC2, making the cloud more hospitable for HPC. [Read More...](#)

# Grid- und Cloud-Computing

## Zusammenfassung

- ▶ Die heute weltweit verfügbare Rechenleistung soll beim Grid-Computing an verschiedenen Orten nutzbar sein
- ▶ Cluster, Parallelrechner, Rechenzentren werden zu einem Grid zusammengefaßt
- ▶ Wir kennen verschiedene Nutzungsmethoden
- ▶ Die Fragestellungen bleiben dieselben, allerdings auf einem anderen Abstraktionsniveau
- ▶ Grid-Computing: Rechenleistung aus der Steckdose
- ▶ Globus ist eine Infrastruktur in Form einer Sammlung von Diensten
- ▶ Die großen Supercomputing-Zentren in Europa formieren sich zu einer Grid-Infrastruktur
- ▶ Cloud-Computing stellt verschiedene Dienste bereit
- ▶ Cloud-Computing zur Zeit (noch) nicht gut für HPC geeignet