Hochleistungsrechnen

Vorlesung im Wintersemester 2010/11

Skriptversion 09.11.2010

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

Siehe: http://en.wikipedia.org/wiki/High_performance_computing

Der starke Bezug zur Mathematik ist dadurch gegeben, dass auf Hochleistungsrechnern größtenteils rechenintensive numerische Verfahren zum Einsatz kommen. Die computergestützte Simulation von Systemen beliebiger Natur steht dabei im Vordergrund.

Anwendungsbereiche

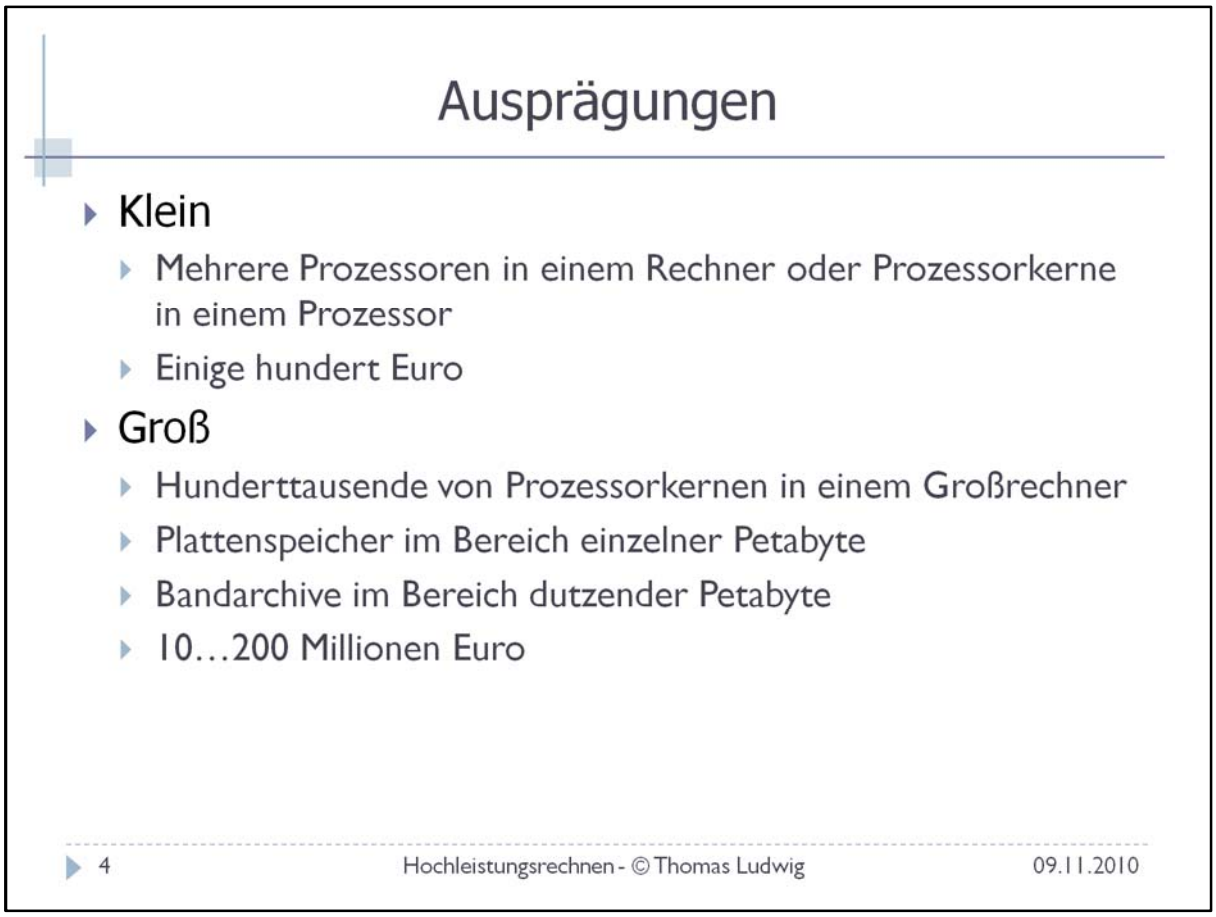
Anwendungsbereiche sind in allen Wissenschaften mit einem hohen Bedarf an Rechen- und Speicherleistung

Klassisch:

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

Neu dazugekommen:

- Finanzwirtschaft
- Sozialwissenschaften (Simulation von Gesellschaften)



• 1 Petabyte = 1024 Terabyte = 1024x1024 Gigabyte.
• Eine DVD fasst 4 Gigabyte.
• 2010: Eine aktuelle Festplatte speichert 1 Terabyte, ein aktuelles Magnetband speichert 1 Terabyte.

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 (16-45)
- Die TOP500-Liste (46-100)
- Vernetzungkonzepte (103-136)
- Hochleistungs-Eingabe/Ausgabe (137-170)
- Betriebssystemaspekte (171-199)

Teil II: Programmierung

- Parallele Programmierung (200-240)
- Programmiermodell Nachrichtenaustausch (241-276)
- Parallele Eingabe/Ausgabe (277-303)
- Programmierung mit Threads (304-340)
- Programmierung mit OpenMP (341-380)
- Leistungsanalyse
- Hybride Programmierung
- Grafikkartenprogrammierung
- Skalierbarkeit, Lastausgleich, Engpässe

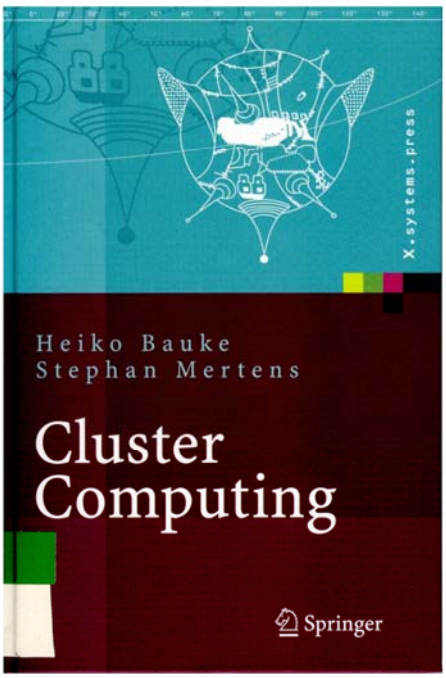
Teil III: Programmierwerkzeuge

- Werkzeugarchitekturen
- Fehlersuche
- Leistungsanalyse
- Leistungsoptimierung
- Lastausgleich
- Fehlertoleranz

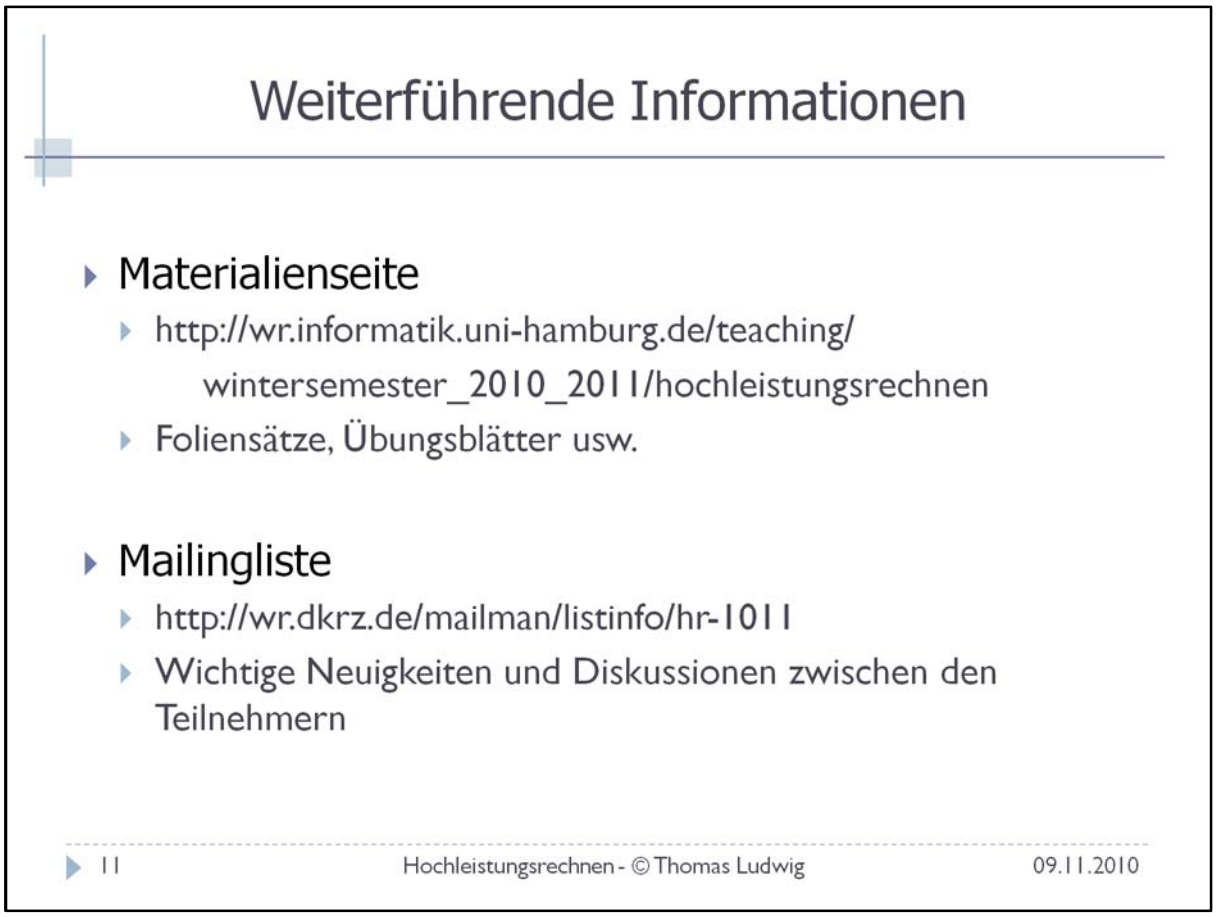
Teil IV: Aktuelle Fragestellungen

- Grid- und Cloud-Computing
- Energieeffizienz im Hochleistungsrechnen
- Simulation der Rechnerumgebung
- Die Technologische Singularität

Literatur



Heiko Bauke &
Stephan Mertens
Cluster Computing
Springer, 2006
460 Seiten, 50 Euro
ISBN-10: 3-540-42299-4



Siehe:

- http://wr.informatik.uni-hamburg.de/teaching/sommersemester_2010/hochleistungsrechnen
- http://wr.dkrz.de/mailman/listinfo/hr-10

Leistungsnachweis

- Für die Studierenden aus den verschiedenen Studiengängen ja nach Bedarf
  - Klausur ?
  - Mündliche Prüfung ?
  - Anwesenheitspflicht ?
  - Welche Studiengänge sind vertreten?
- Übungen
  - Organisiert durch Michael Kuhn und Julian Kunkel

Arbeitsbereich Wissenschaftliches Rechnen

- Im Department 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
- Personelle Ausstattung
  - Kuhn [UHH], Kunkel, [DKRZ] Lenhart, [UHH], Minartz [UHH], (Nerge [UHH])

Weitere Lehrveranstaltungen im WS 2010/11

- Projekt „Parallelrechnerevaluation“
  (Ludwig, Kuhn, Lenhart)
- Proseminar „Speicher- und Dateisysteme“
  (Ludwig, Kuhn)
- Seminar „Softwareentwicklung in der Wissenschaft“
  (Ludwig, Kunkel, Nerge, Lenhart)

Mitarbeit in der Arbeitsgruppe

- Forschungsthemen
  - Energieeffizienz von Hochleistungsrechnern
  - Speicherung großer Datenmengen
  - Simulation von Hochleistungsrechensystemen

Wir betreuen:

- Bachelor-, Master-, Diplomarbeiten
- Promotionsvorhaben

Wir stellen ein: ☺

- Studentische Hilfskräfte
- Mitarbeiter 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?

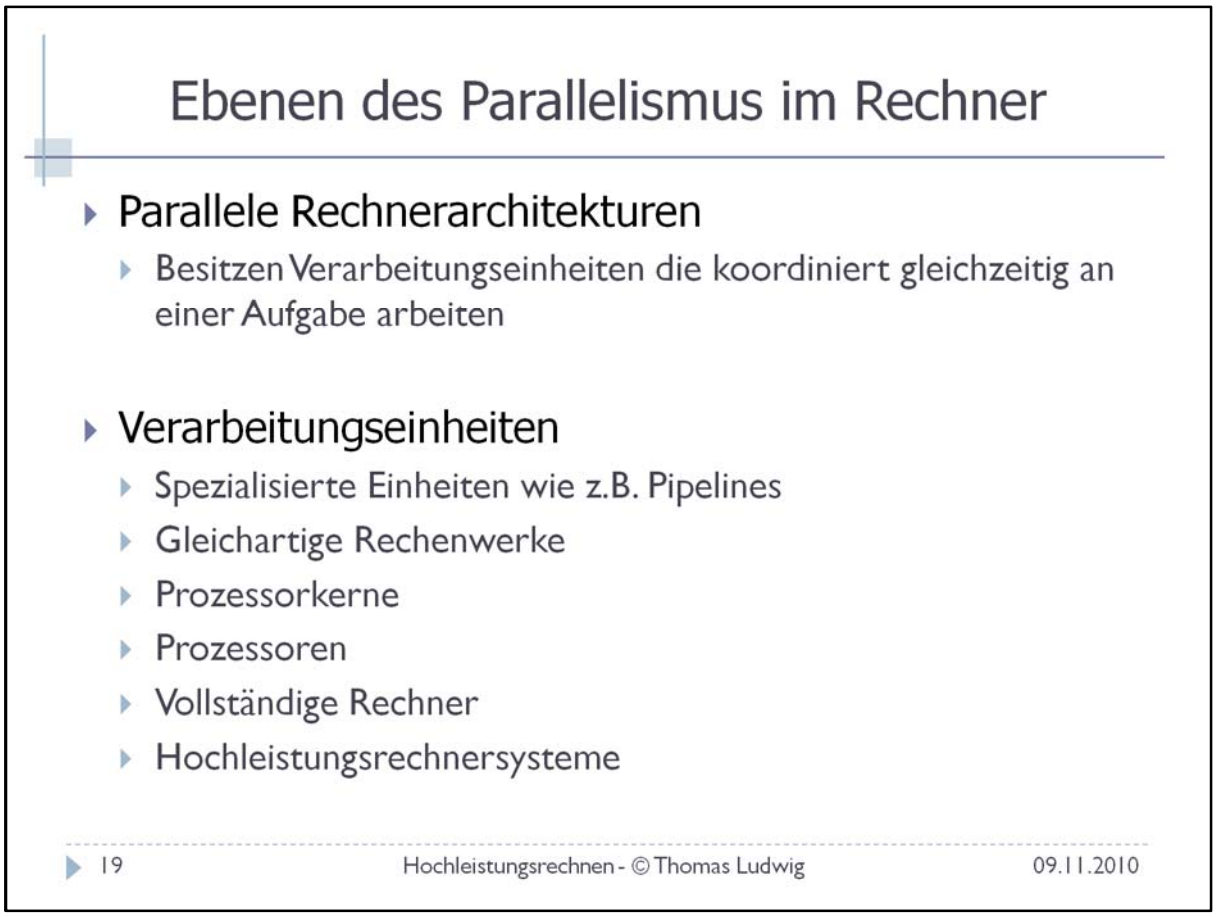
Seymour Cray (1925-1996)

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

Die beiden Zitate sind dem Buch von Bauke/Mertens über Cluster-Computing entnommen.



Im Speichersystem finden wir die Existenz vieler Festplatten und vieler Bandlesegeräte.

In der Vernetzung finden wir parallele Wege zwischen vielen Paaren von Kommunikationspartnern.

Flynnsche Klassifikation

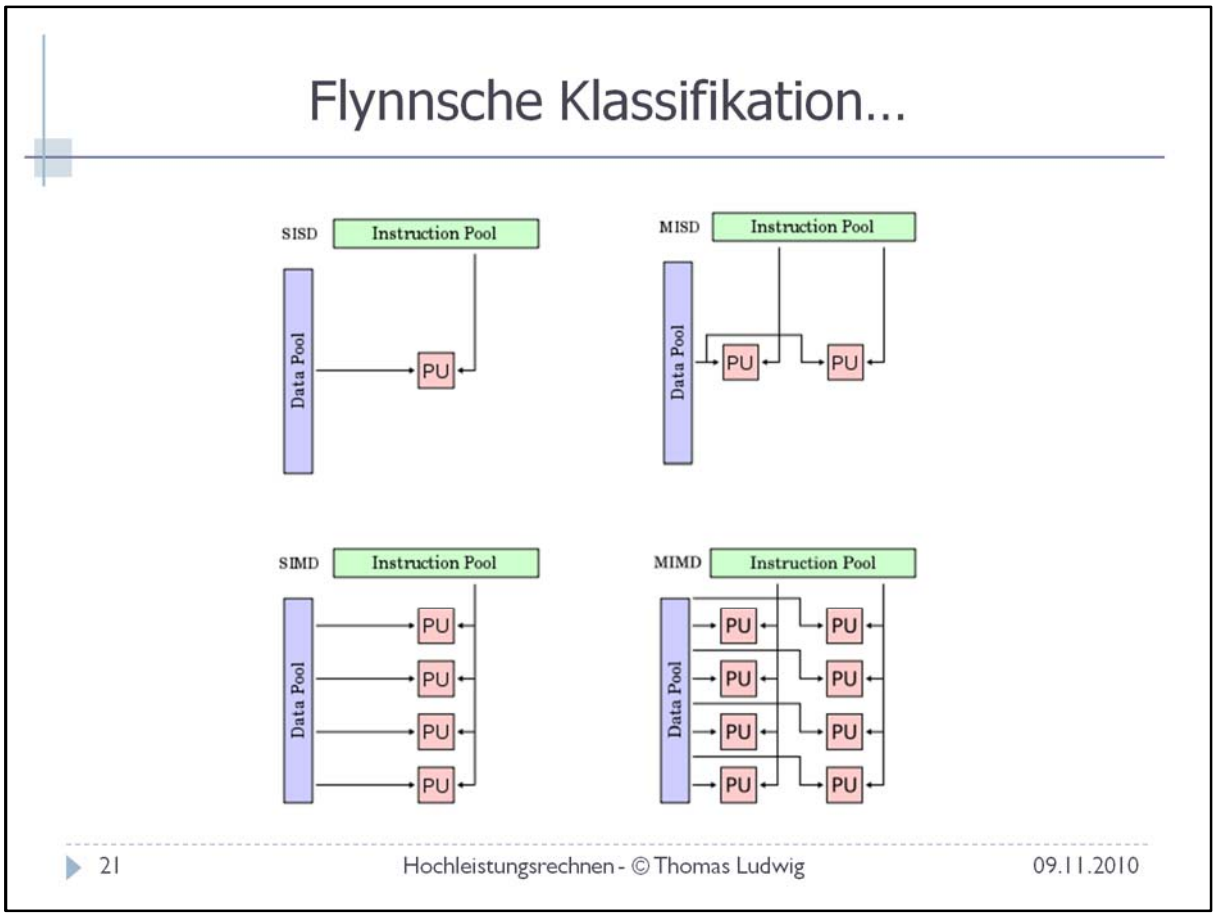
Klassifikation nach Flynn (1972)

- 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

Siehe: http://en.wikipedia.org/wiki/Flynn%20s_Taxonomy

Die Klassifikation ist historisch. Sie dient hier einer ersten Unterteilung unserer Rechner in verschiedene Klassen, genügt aber nicht den aktuellen Anforderungen und kann die aktuellen Systeme nicht adäquat erfassen.



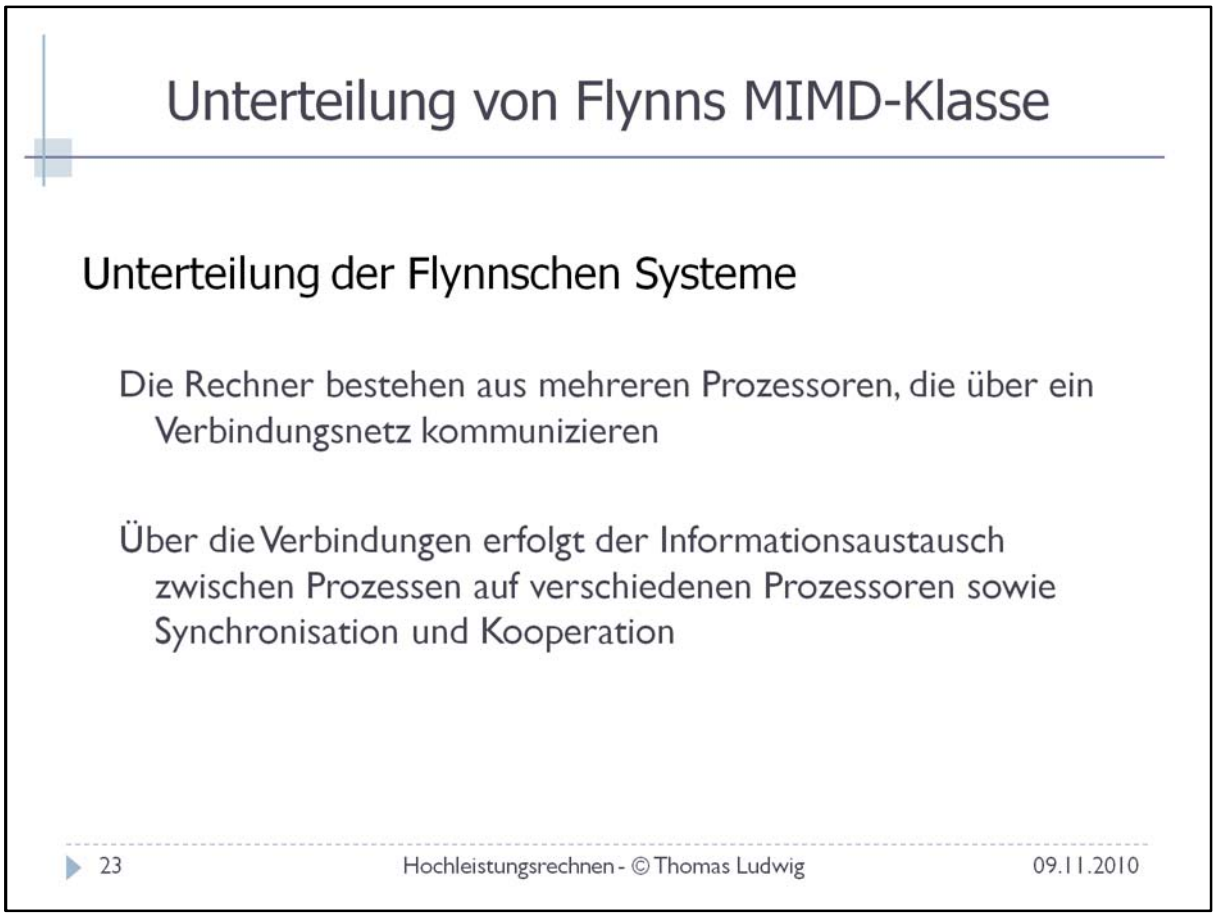
Quelle: Wikimedia Commons (http://en.wikipedia.org/wiki/File:SISD.svg, http://en.wikipedia.org/wiki/File:MISD.svg, http://en.wikipedia.org/wiki/File:SIMD.svg, http://en.wikipedia.org/wiki/File:MIMD.svg)

Flynnsche 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



Moderne Prozessoren enthalten jetzt fast immer mehrere Prozessorkerne, die wie Prozessoren geringerer Leistungsfähigkeit arbeiten. Dies verkompliziert unsere Betrachtungen. Später mehr dazu.

Siehe: http://en.wikipedia.org/wiki/Multi_core

Ein Prozeß ist ein auf einem Prozessor ablaufendes Programm. Ein Prozessor mit z.B. vier Prozessorkernen kann vier Prozesse echt gleichzeitig abarbeiten. Daneben werden ja auf jedem Einzelprozessor normalerweise auch mehrere Prozesse zeitlich verzahnt (also quasi-gleichzeitig) abgearbeitet.

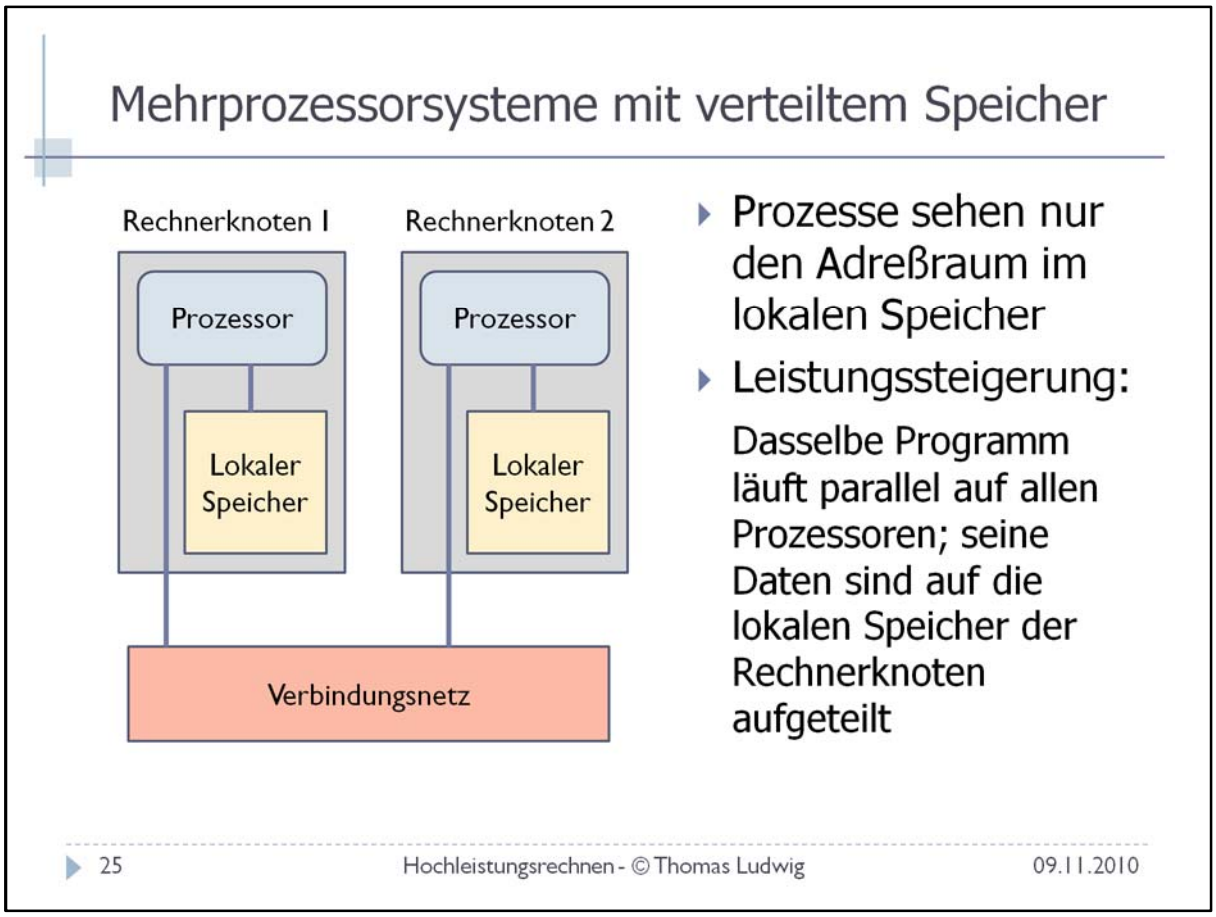
Unterteilung von Flynns MIMD-Klasse...

Neue Unterscheidungskriterien

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

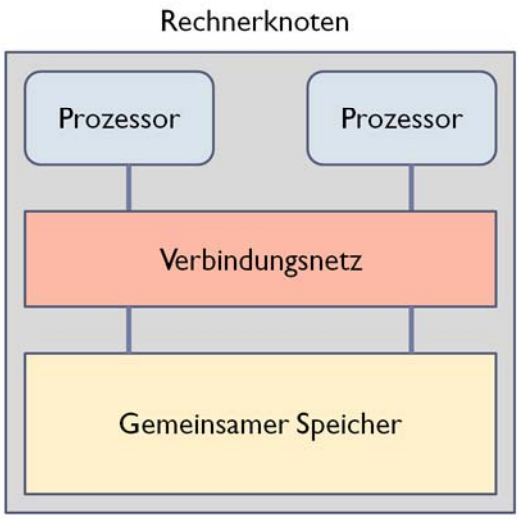
Neue Klassen

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



Im uns am besten bekannten Fall ist der Rechnerknoten ein einzelner Rechner (z.B. PC) und das Verbindungsnetz ein Ethernet-Netz. Bei Hochleistungsrechnern ist der Rechnerknoten ein Einschub in einem Rack (oder auch nur ein Teil eines solchen Einschubs) und das Verbindungsnetz ist etwas hochspezielles.

Mehrprozessorsysteme mit gemeinsamem Speicher

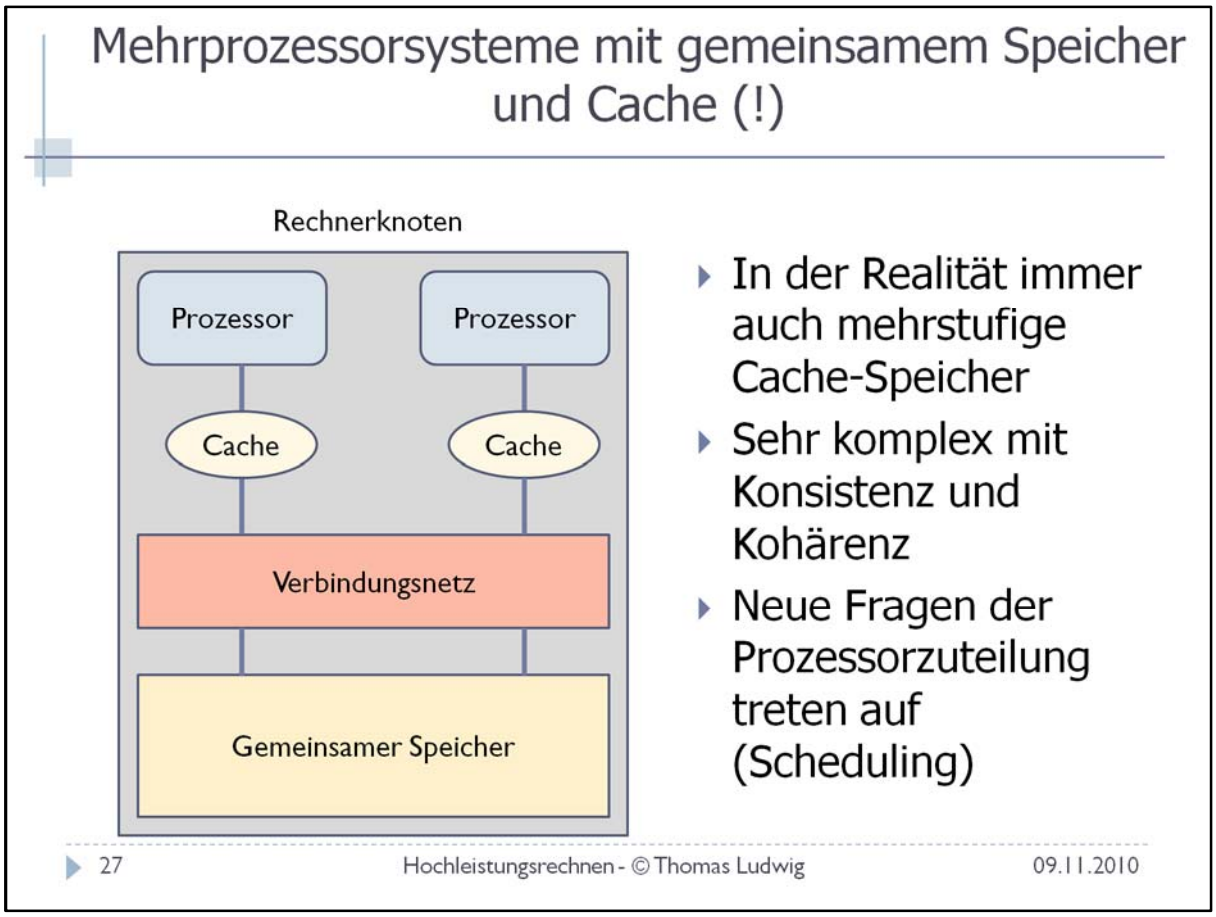


- Jeder Prozeß sieht den gesamten Adreßraum des gemeinsamen Speichers
- Leistungssteigerung:

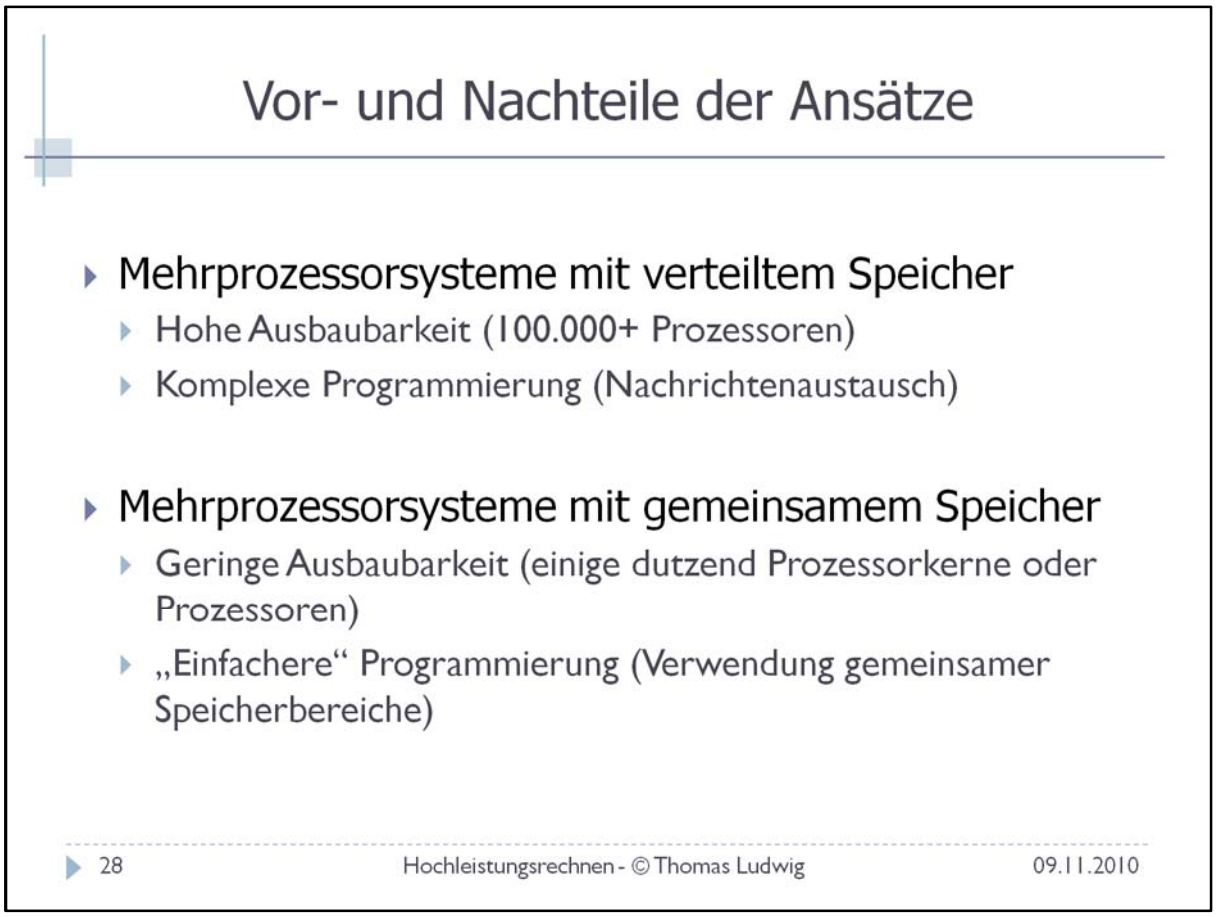
  Dasselbe Programm läuft parallel auf allen Prozessoren; seine Daten sind auf im gemeinsamen Speicher für alle zugreifbar

Siehe: http://en.wikipedia.org/wiki/Multi-core_processor

Im uns am besten bekannten Fall handelt es sich hier um einen Rechner mit einem Prozessor und z.B. zwei Prozessorkernen, wie wir ihn heute als normalen PC kaufen. Das Verbindungsnetz ist hier der Prozessor-Speicher-Bus im Rechner. Bei Hochleistungsrechnern finden wir komplexe Formen der Spezialhardware für das Verbindungsnetz.



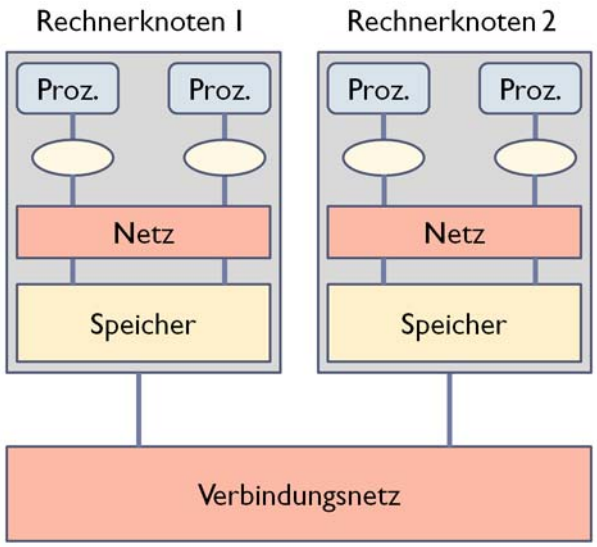
Im uns am besten bekannten Fall handelt es sich hier um einen Rechner mit einem Prozessor und z.B. zwei Prozessorkernen, wie wir ihn heute als normalen PC kaufen. Das Verbindungsnetz ist hier der Prozessor-Speicher-Bus im Rechner. Bei Hochleistungsrechnern finden wir komplexe Formen der Spezialhardware für das Verbindungsnetz.



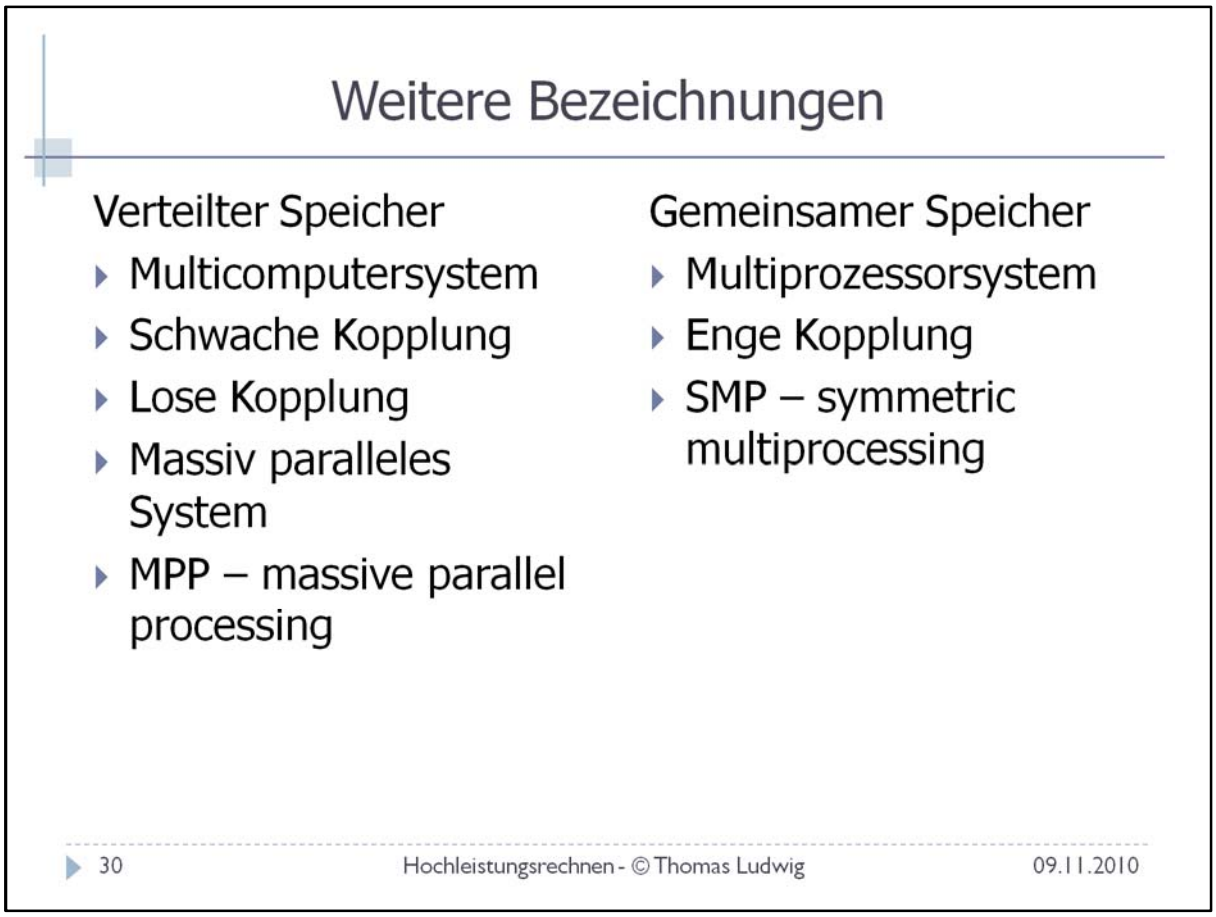
Die Programmierung von Systemen mit gemeinsamem Speicher ist nur auf den ersten Blick einfacher. Soll maximale Effizienz erzielt werden, so ist das auch beliebig schwierig. Allerdings gibt es in diesem Bereich halbwegs brauchbare automatische Ansätze durch Compiler-Unterstützung. Künftig wird man Kenntnisse der Programmierung dieser Architekturen vermehrt brauchen, wenn nämlich die Mehrkernprozessoren sich weiter verbreiten.

Ausbauen kann man die Systeme natürlich beliebig – die Frage ist, wie lange die Leistungssteigerung den aufgewendeten Finanzen folgt.

Reale Systeme: Alles in Einem

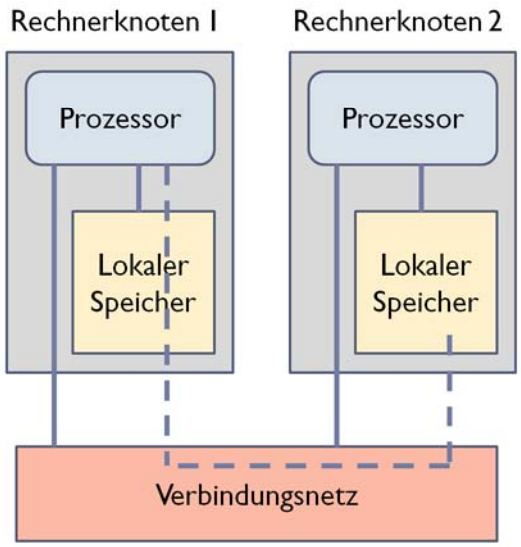


- Existierende HLR sind heute meist eine Kombination aus Rechnerknoten mit gemeinsamem Speicher, von den man viele verwendet und über ein Verbindungsnetz verbindet



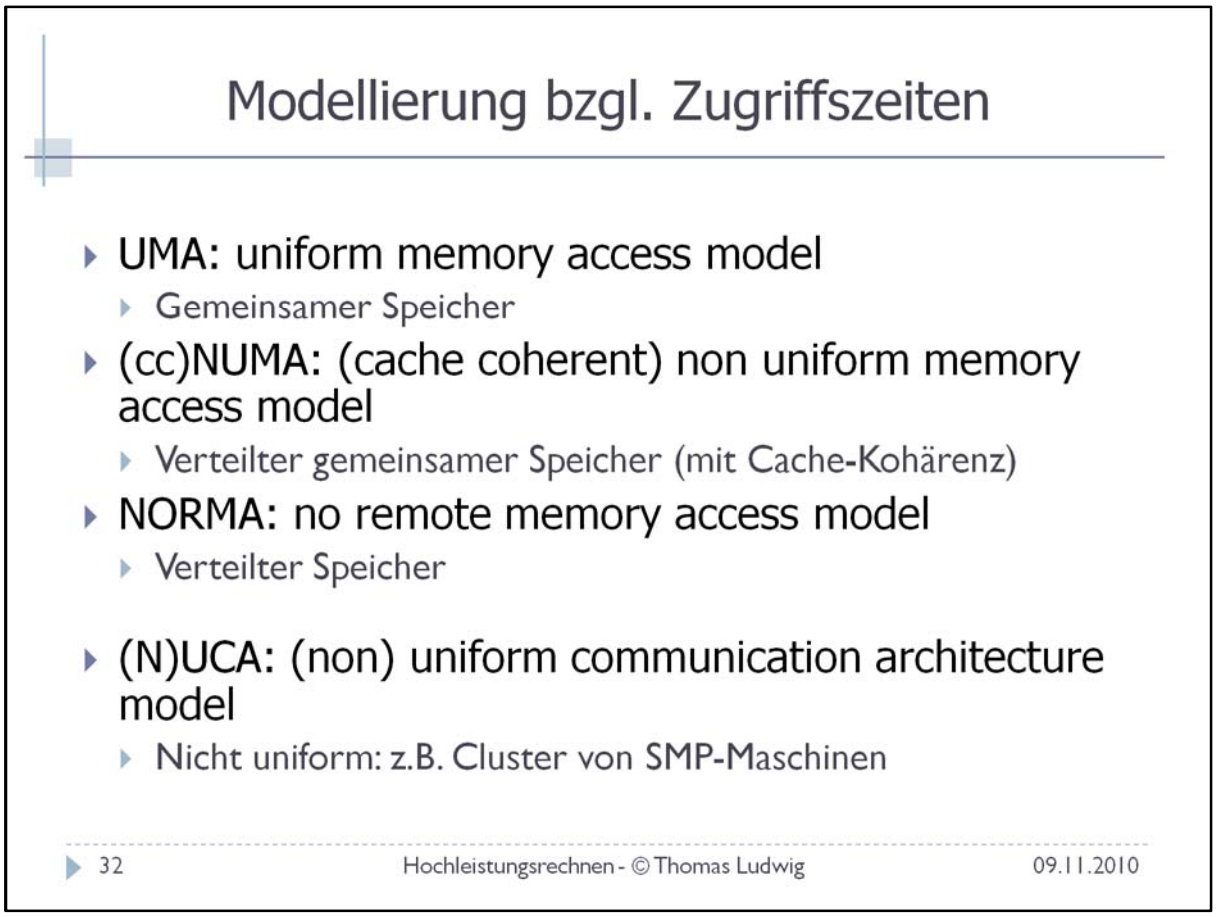
SMP ist eine Bezeichnung aus der Betriebssystemtechnik. Wir kommen später darauf zurück.

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)

DSM-Architekturen kommen im alltäglichen PC-Bereich nicht vor und spielen auch bei Hochleistungsrechnern keine Rolle mehr.



Am gebräuchlichsten in Prospekten ist die Bezeichnung (cc)NUMA. Die anderen Begriffe kommen seltener vor und dann zur Abgrenzung gegenüber NUMA.

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

Siehe: http://en.wikipedia.org/wiki/Scalability

Verbindungsnetze

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

Probleme

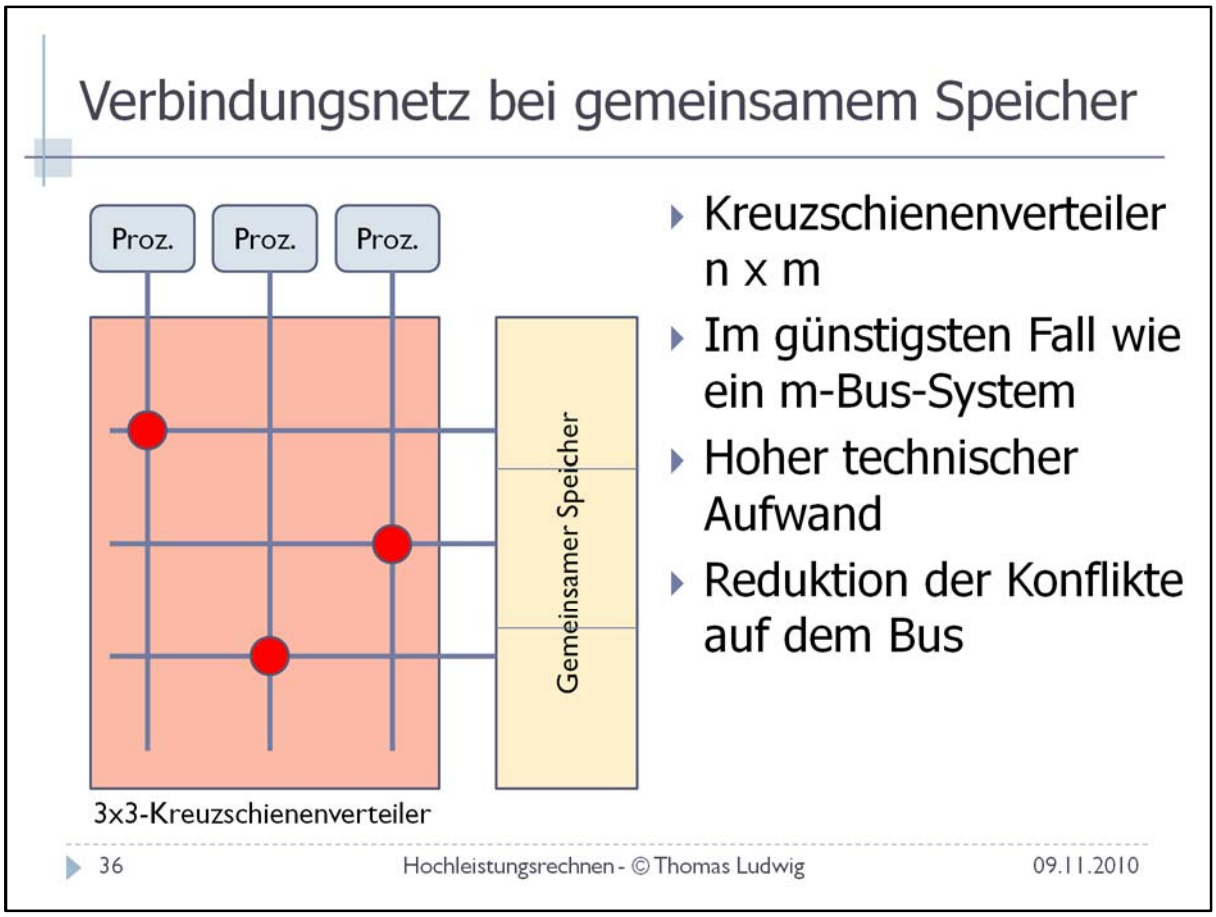
- Latenzzeiten, Übertragungszeiten
- Netzbelastung, Kollisionen

Beispiele von Verbindungsnetzen

Es gibt hier eine Vielzahl von Konzepten !

Wir greifen drei davon zur Illustration heraus:

- Kreuzschienenverteiler
- Zweidimensionaler Torus
- Hypercube

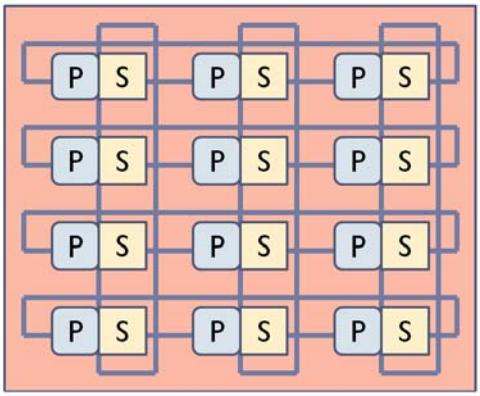


Engl.: crossbar switch

Siehe: http://en.wikipedia.org/wiki/Crossbar_switch

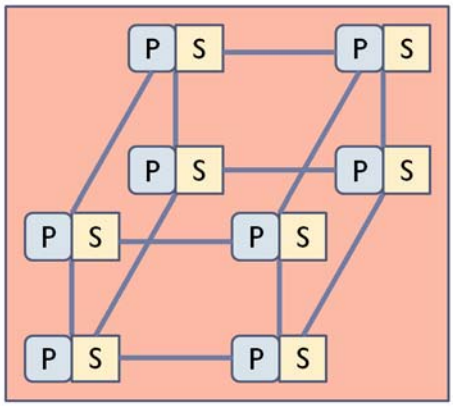
Verbindet Prozessoren mit Speichermodulen; die Module (hier drei) können unabhängig voneinander angesprochen werden.

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 Entfernungen
- 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

SSI hat in der Praxis keine Relevanz. Das Verbergen eines konkreten Ausführungsortes taucht aber als Forschungsziel immer wieder auf, zuletzt beim Cloud-Computing.

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)

Siehe: http://en.wikipedia.org/wiki/Flops,
http://en.wikipedia.org/wiki/Benchmark_%28computing%29

Zum Vergleich: Prozessoren

Beispiele der GFLOP-Werte an einigen CPUs[1]

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) [2]	64,45
Core i7, 3,33 GHz, 6 Kerne	80		
Itanium, 1,6 GHz	6,4	5,95	92,97

Siehe: http://de.wikipedia.org/wiki/Floating_Point_Operations_Per_Second#cite_note-0

Zum Vergleich: Anwendungen

- Rechnersystem Blizzard am DKRZ 2010
  - 110 TFLOPS LINPACK-Leistung
  - Bei ca. 8000 Prozessoren macht das ca. 13 GFLOPS/Proz
- 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_{peak} ist die theoretische Maximalleistung

Siehe: http://en.wikipedia.org/wiki/LINPACK

Die TOP500-Liste

Website 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: Internatinal Supercomputing Conference Deutschland
- November: Supercomputing Conference USA

Basiert auf dem LINPACK-Benchmark

Siehe: http://www.top500.org/

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
2	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50
3	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85	
4	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
5	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT	71680	563.10	1206.19	
6	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0 GHz/Nehalem EP 2.93 Ghz / 2009 SGI	56320	544.30	673.26	2348.00
7	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
8	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	458.61	557.06	1260.00

TOP500 Nov 2009 1-8 / Ein paar wichtige Daten sollten Sie kennen.

Nr 1: Mit über 220.000 Prozessorkernen verwendet Jaguar einen 6-Kern-AMD-Prozessoren. Braucht auch entsprechend viel Energie.

Nr 2: Roadrunner verwendet zwei Prozessortypen: Opteron und Cell-Prozessor

Nr 4: Das in Jülich stehende System JUGENE hat fast 300.000 Prozessoren vom Typ PowerPC 450 850 MHz.

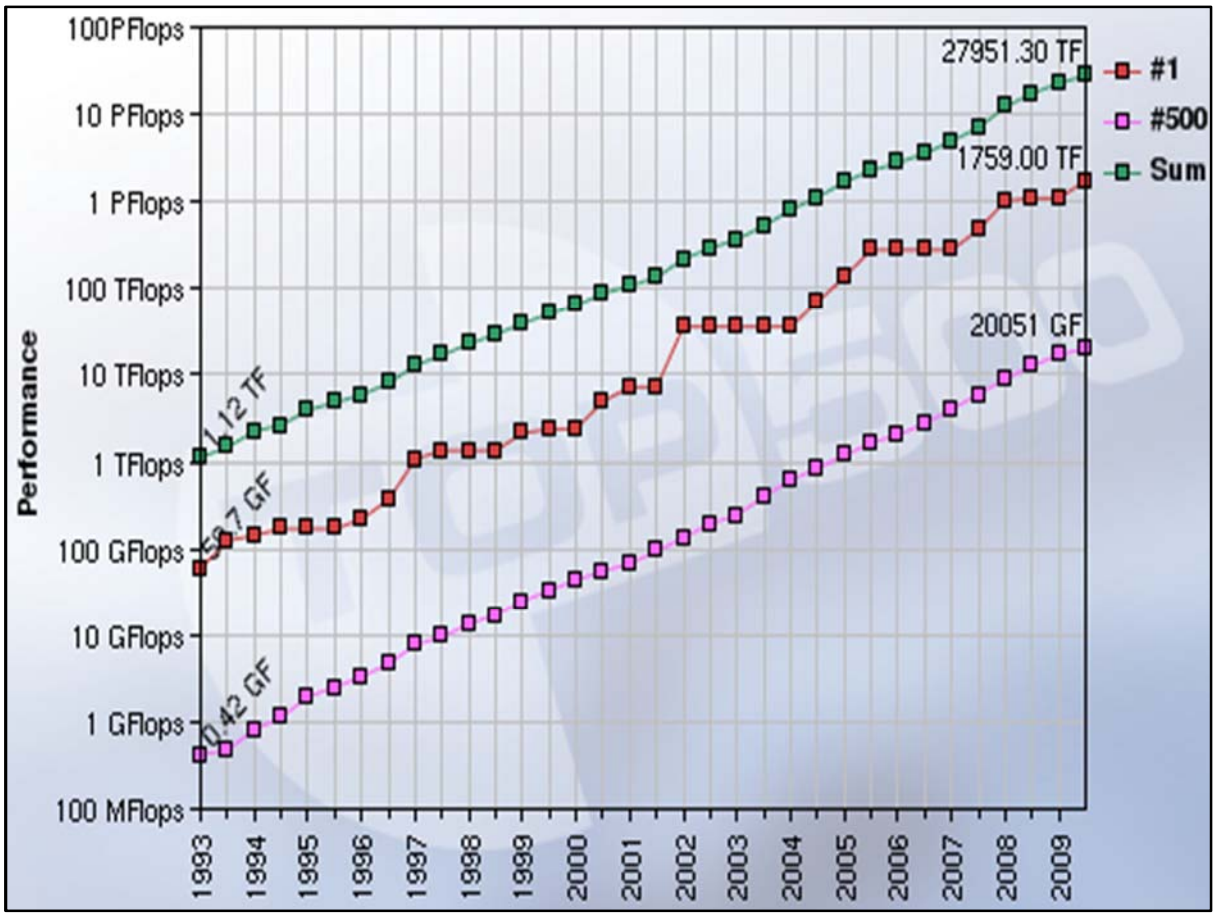
9	Texas Advanced Computing Center/Univ. of Texas United States	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband / 2008 Sun Microsystems	62976	433.20	579.38	2000.00
10	Sandia National Laboratories / National Renewable Energy Laboratory United States	Red Sky - Sun Blade x6275, Xeon X55xx 2.93 Ghz, Infiniband / 2009 Sun Microsystems	41616	423.90	487.74	
11	DOE/NNSA/LLNL United States	Dawn - Blue Gene/P Solution / 2009 IBM	147456	415.70	501.35	1134.00
12	Moscow State University - Research Computing Center Russia	Lomonosov - T-Platforms T-Blade2, Xeon 5570 2.93 GHz, Infiniband QDR / 2009 T-Platforms	35360	350.10	414.42	
13	Forschungszentrum Juelich (FZJ) Germany	JUROPA - Sun Constellation, NovaScale R422-E2, Intel Xeon X5570, 2.93 GHz, Sun M9/Mellanox QDR Infiniband/Partec Parastation / 2009 Bull SA	26304	274.80	308.28	1549.00
14	KISTI Supercomputing Center Korea, South	Tachyonll - Sun Blade x6048, X6275, IB QDR M9 switch, Sun HPC stack Linux edition / 2009 Sun Microsystems	26232	274.80	307.44	1275.96
15	NERSC/LBNL United States	Franklin - Cray XT4 QuadCore 2.3 GHz / 2008 Cray Inc.	38642	266.30	355.51	1150.00
16	Oak Ridge National Laboratory United States	Jaguar - Cray XT4 QuadCore 2.1 GHz / 2008 Cray Inc.	30976	205.00	260.20	1580.71

TOP500 Nov 2009 9-16 / Ein paar wichtige Daten sollten Sie kennen.

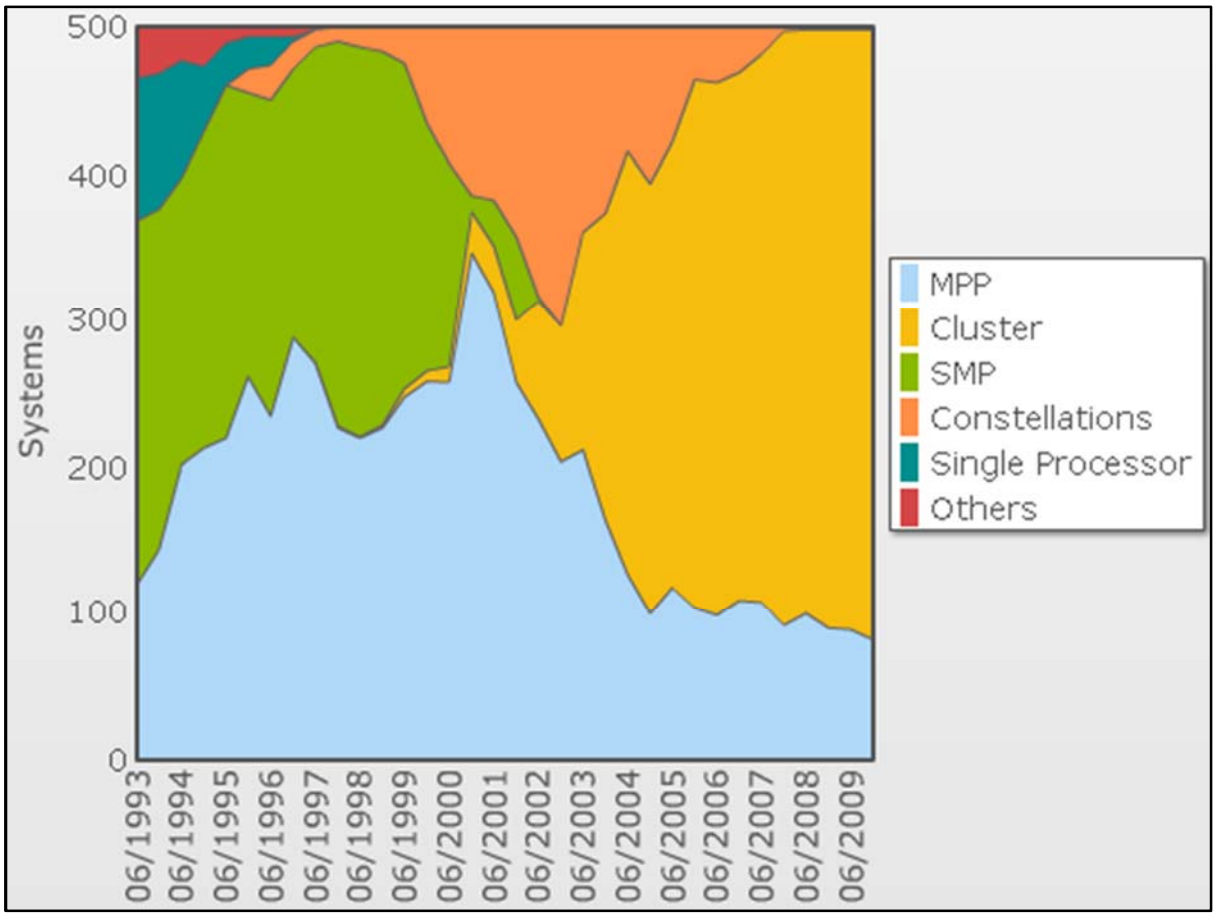
Rank	Site	System	Cores	R_{max}	R_{peak}
4	Forschungszentrum Juelich (FZJ) Germany	Blue Gene/P Solution IBM	294912	825.5	1002.7
13	Forschungszentrum Juelich (FZJ) Germany	Sun Constellation, NovaScale R422-E2, Intel Xeon X5570, 2.93 GHz, Sun M9/Mellanox QDR Infiniband/Partec Parastation Bull SA	26304	274.8	308.28
35	DKRZ - Deutsches Klimarechenzentrum Germany	Power 575, p6 4.7 GHz, Infiniband IBM	8064	115.9	151.6
39	HLRN at Universitaet Hannover / RRZN Germany	SGI Altix ICE 8200EX, Xeon QC E5472 3.0 GHz/X5570 2.93 GHz SGI	10240	107.1	120.73
40	HLRN at ZIB/Konrad Zuse-Zentrum fuer Informationstechnik Germany	SGI Altix ICE 8200EX, Xeon QC E5472 3.0 GHz/X5570 2.93 GHz SGI	10240	107.1	120.73
46	Max-Planck-Gesellschaft MPI/IPP Germany	Power 575, p6 4.7 GHz, Infiniband IBM	6848	98.42	128.74
50	IT Service Provider Germany	Cluster Platform 3000 BL2x220, E54xx 3.0 Ghz, Infiniband Hewlett-Packard	10240	94.74	122.88
82	Leibniz Rechenzentrum Germany	Altix 4700 1.6 GHz SGI	9728	56.52	62.26
94	HWW/Universitaet Stuttgart Germany	NEC HPC 140Rb-1 Cluster, Xeon X5560 2.8Ghz, Infiniband NEC	5376	50.79	60.21
102	Max-Planck-Gesellschaft MPI/IPP Germany	Blue Gene/P Solution IBM	16384	47.73	55.71
110	Forschungszentrum Juelich (FZJ) Germany	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus IBM	4608	43.01	55.71

TOP500 Nov 2009 Deutschland: 27 Systeme auf den Plätzen von 4 bis 481.

Das DKRZ steht auf Platz 3 in Deutschland.

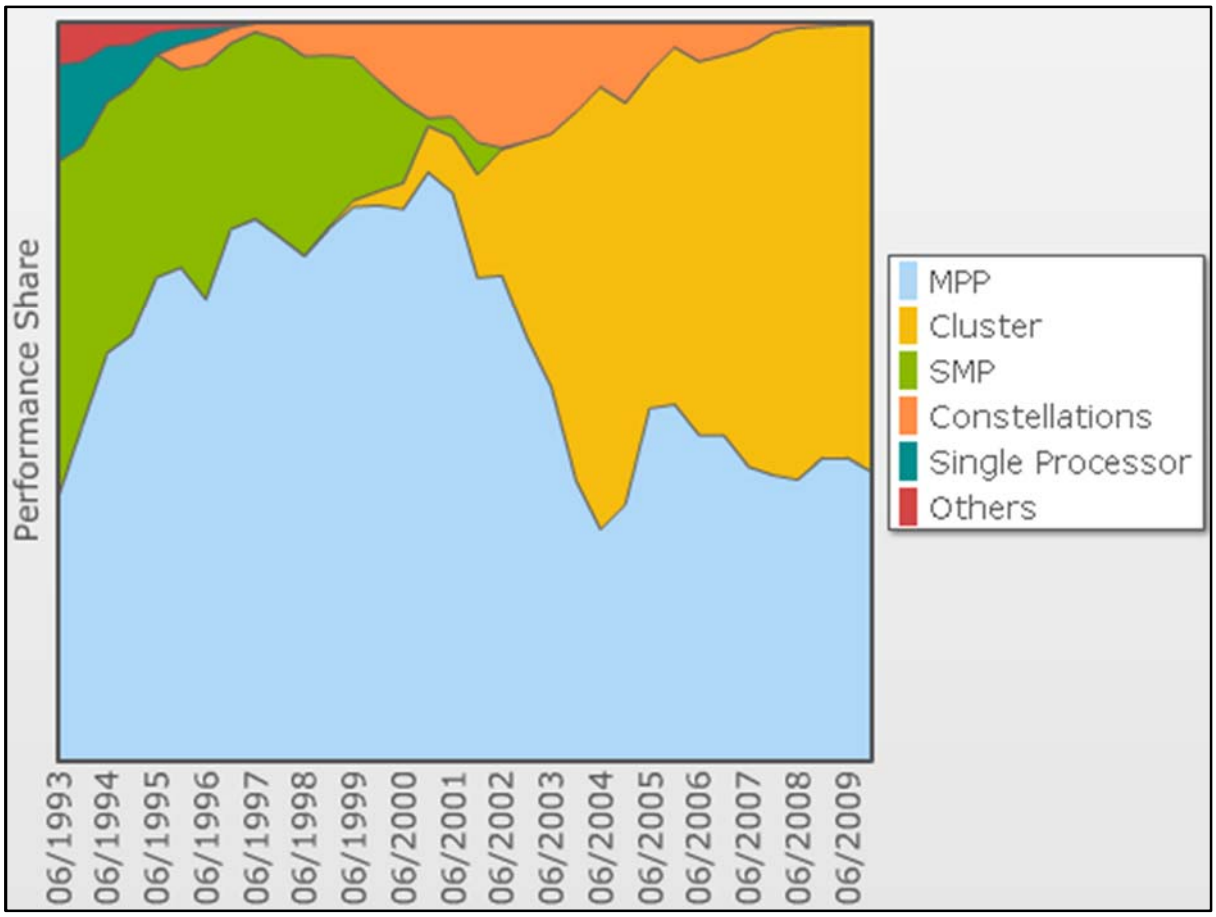


TOP500 Leistung: Die aktuellen Nr. 1-Rechner erzeugen immer ein Plateau über mehrere Listen hinweg (Z.B. ab 06/2002 5x der japanische Earth Simulator von NEC auf Platz 1). Die Leistung des Nr. 500-Rechners und die aggregierte Gesamtleistung aller 500 Rechner steigen in dieser Grafik linear, wobei die y-Achse logarithmisch aufgetragen ist.



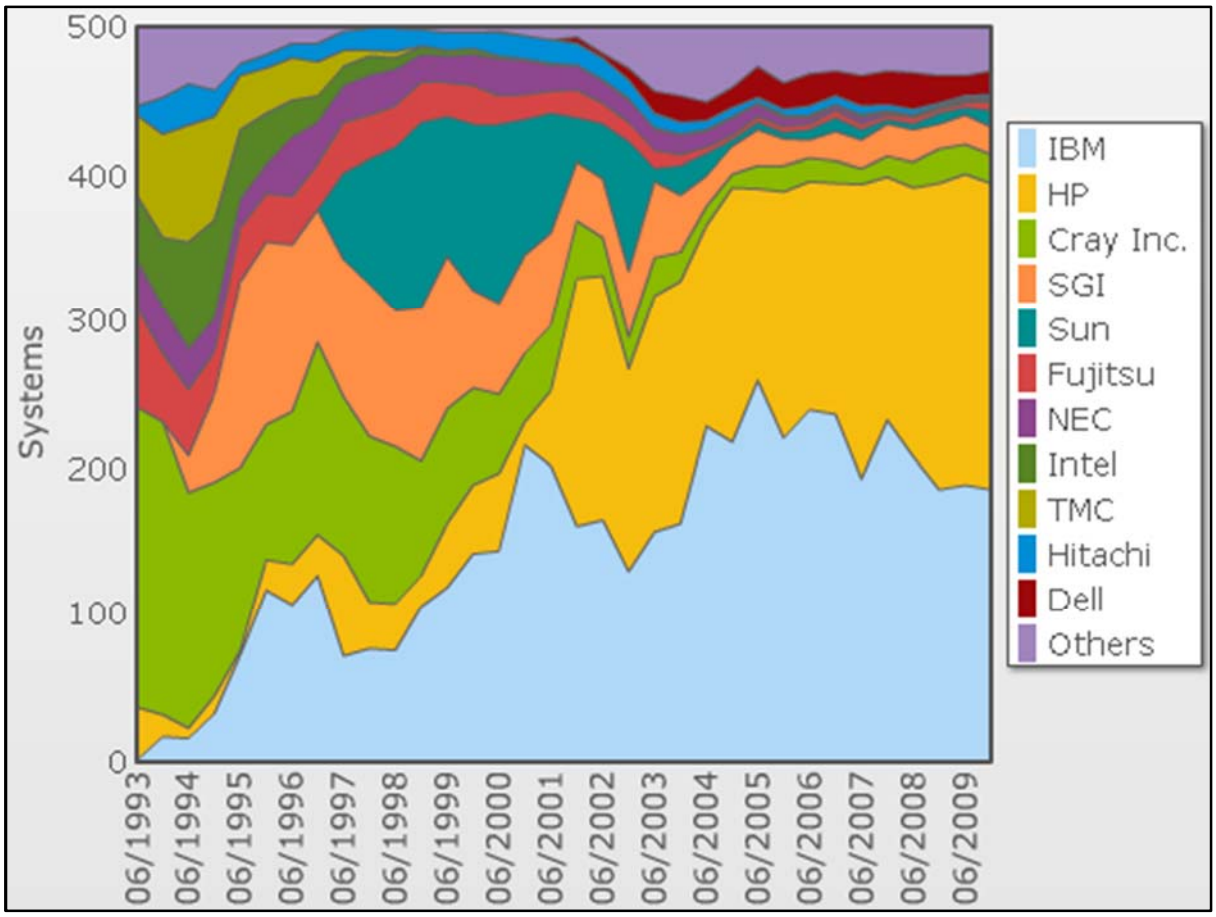
TOP500 Architekturen (# Systeme)

Die überwiegende Zahl der Installationen sind vom Typ Cluster. Beachten Sie die ausgestorbenen Architekturen: Einzelprozessor, SMP und Constellation.



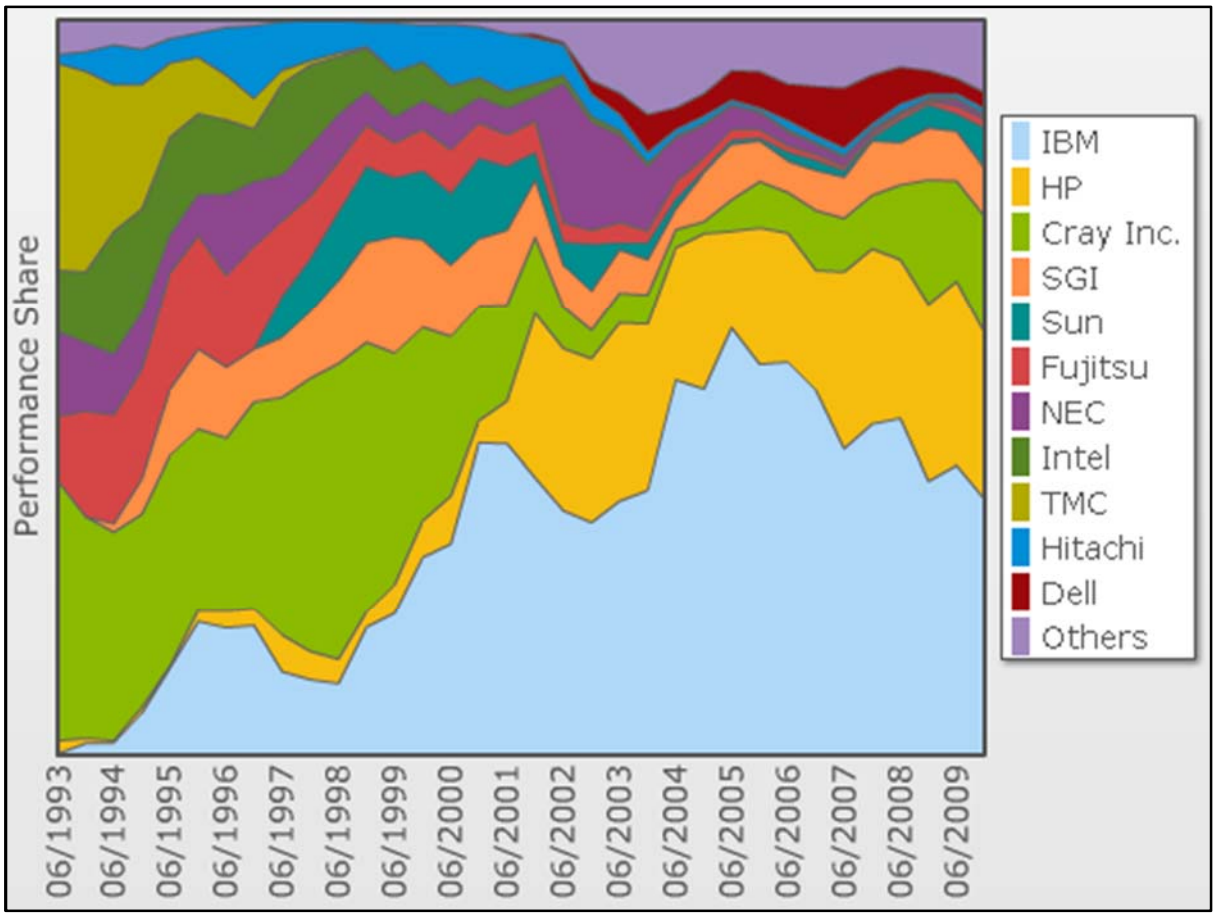
TOP500 Architekturen (% Leistung)

Der Leistungsanteil der Cluster ist geringer, weil die Menge der BlueGene-Systeme nicht vom Typ Cluster ist und insgesamt viel Leistung liefert.



TOP500 Hersteller (# Systeme)

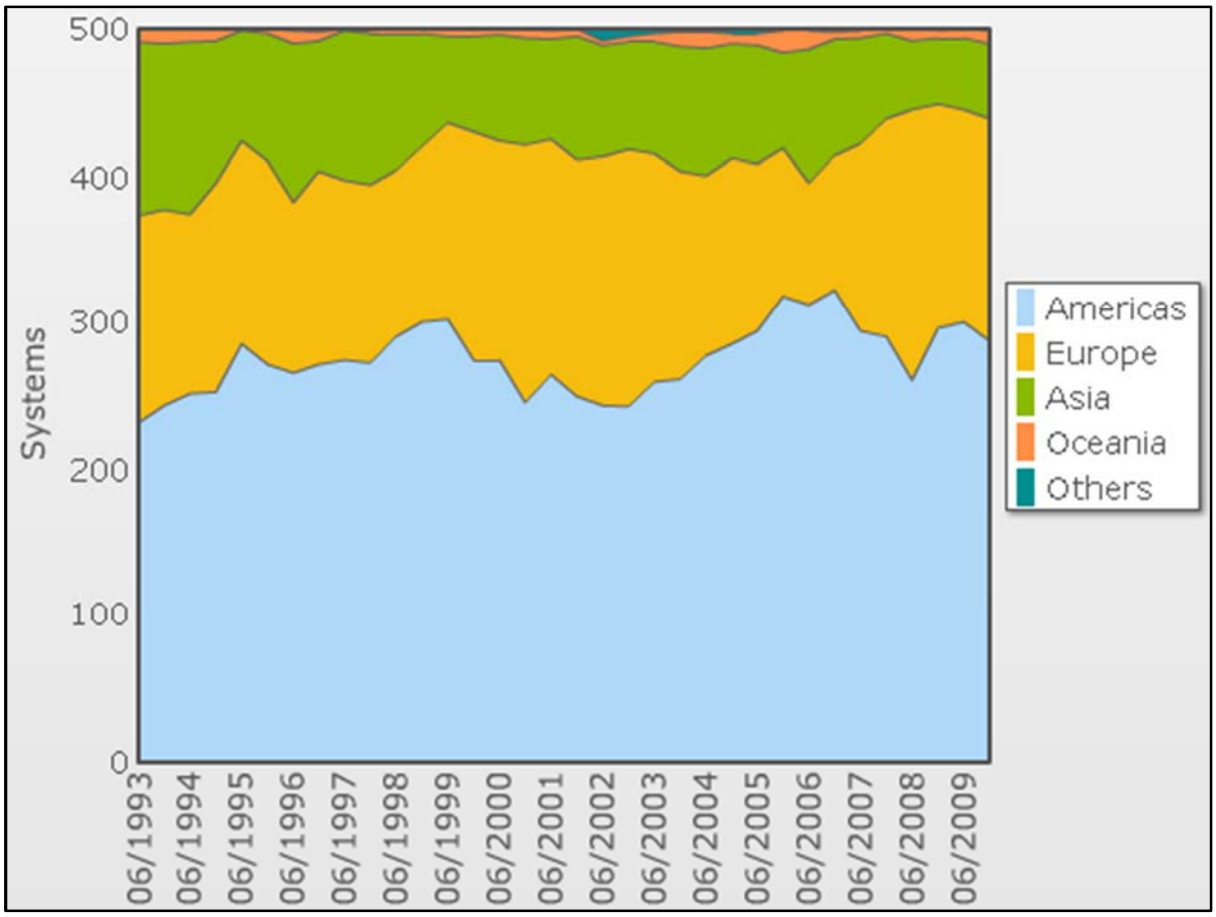
IBM und HP sind die Führer mit etwa gleich vielen Systemen am Markt.



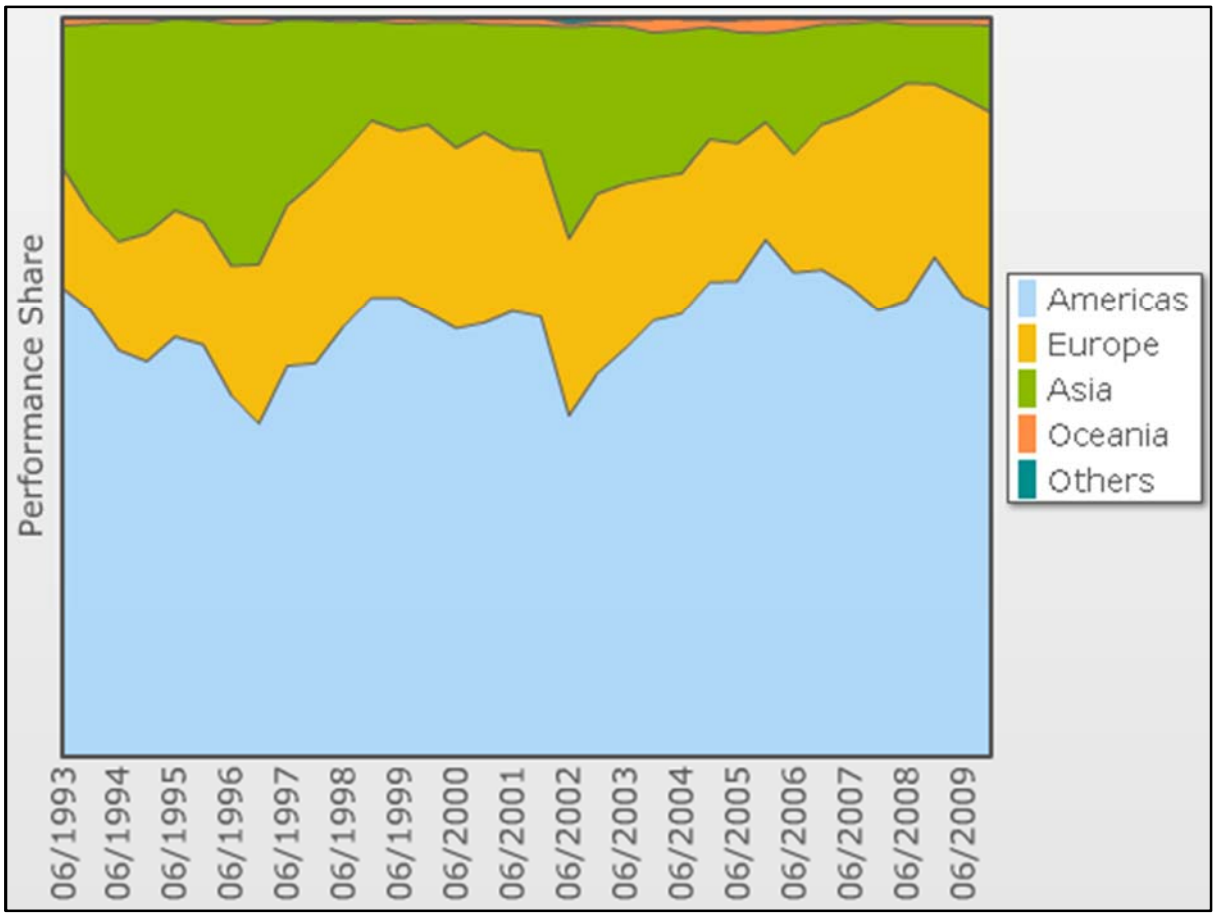
TOP500 Hersteller (% Leistung)

An der Leistung hat IBM aber wegen der BlueGene-Systeme einen viel größeren Gesamtanteil.

Man sieht auch schön den Zugewinn bei NEC nach Installation des Earth Simulators.

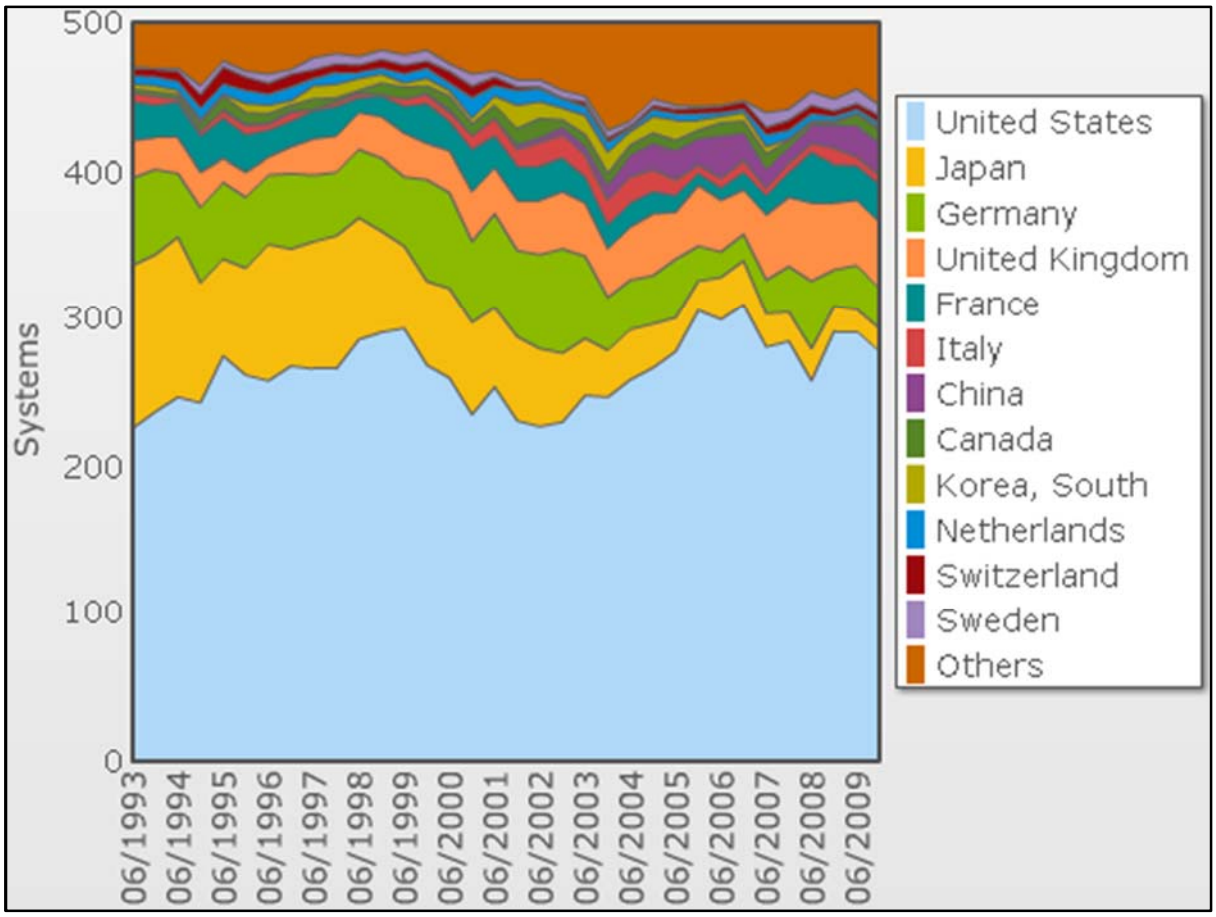


TOP500 Kontinente (# Systeme)

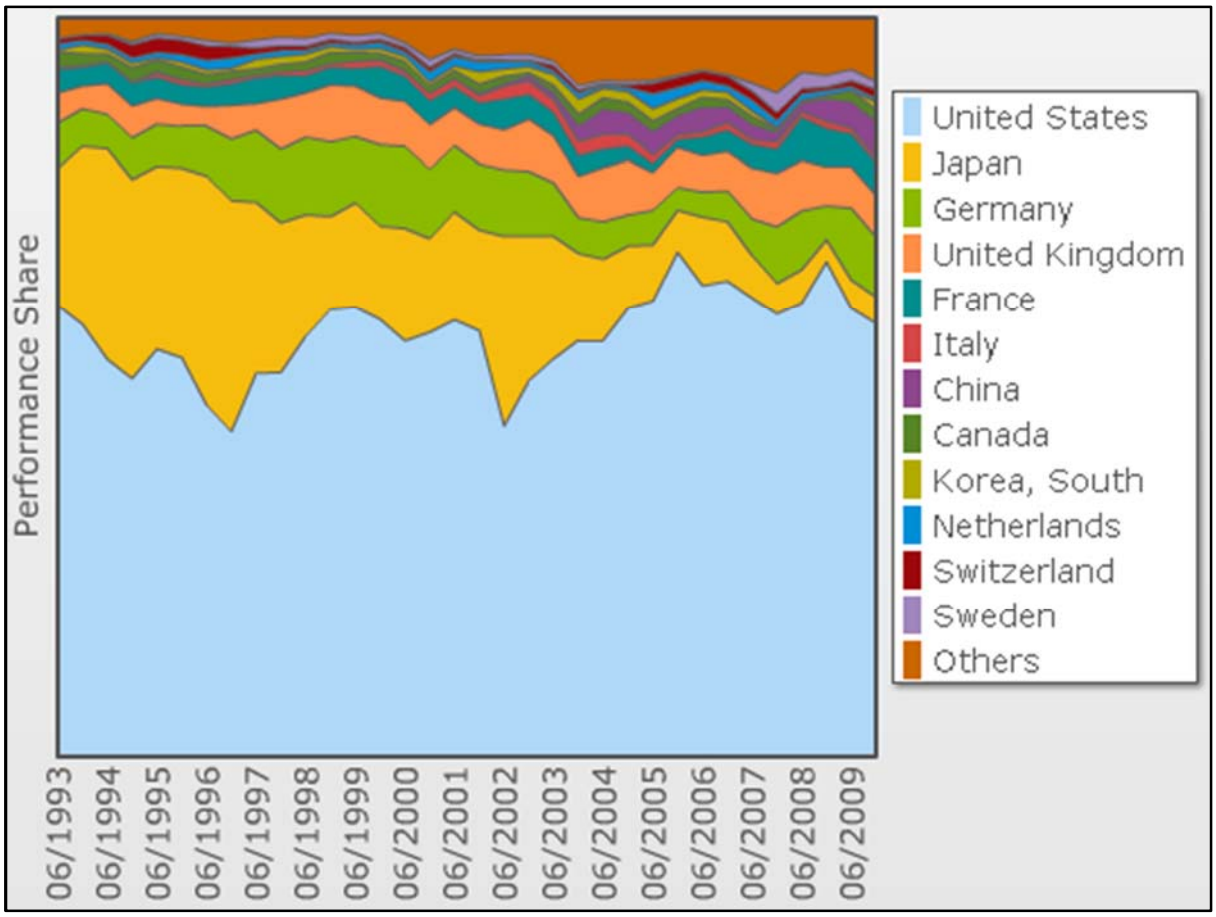


TOP500 Kontinente (% Leistung)

Man sieht Juni 2002 schön den Einfluß des japanischen Earth-Simulator von NEC: ein System mit sehr hoher Leistung, das Japan nach vorne bringt (grün!). (Ebenso Juni 1996 eine starke japanische Maschine von Hitachi).

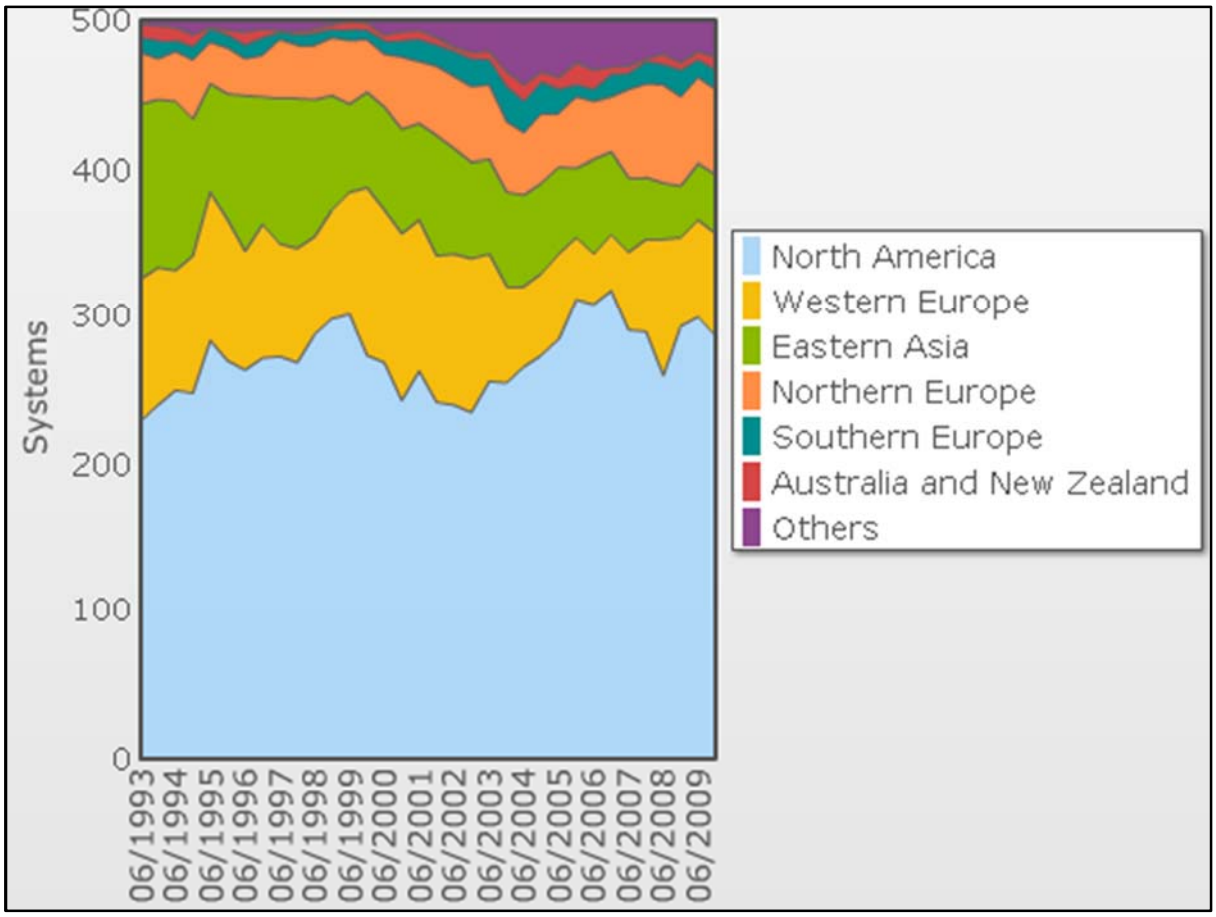


TOP500 Länder (# Systeme)

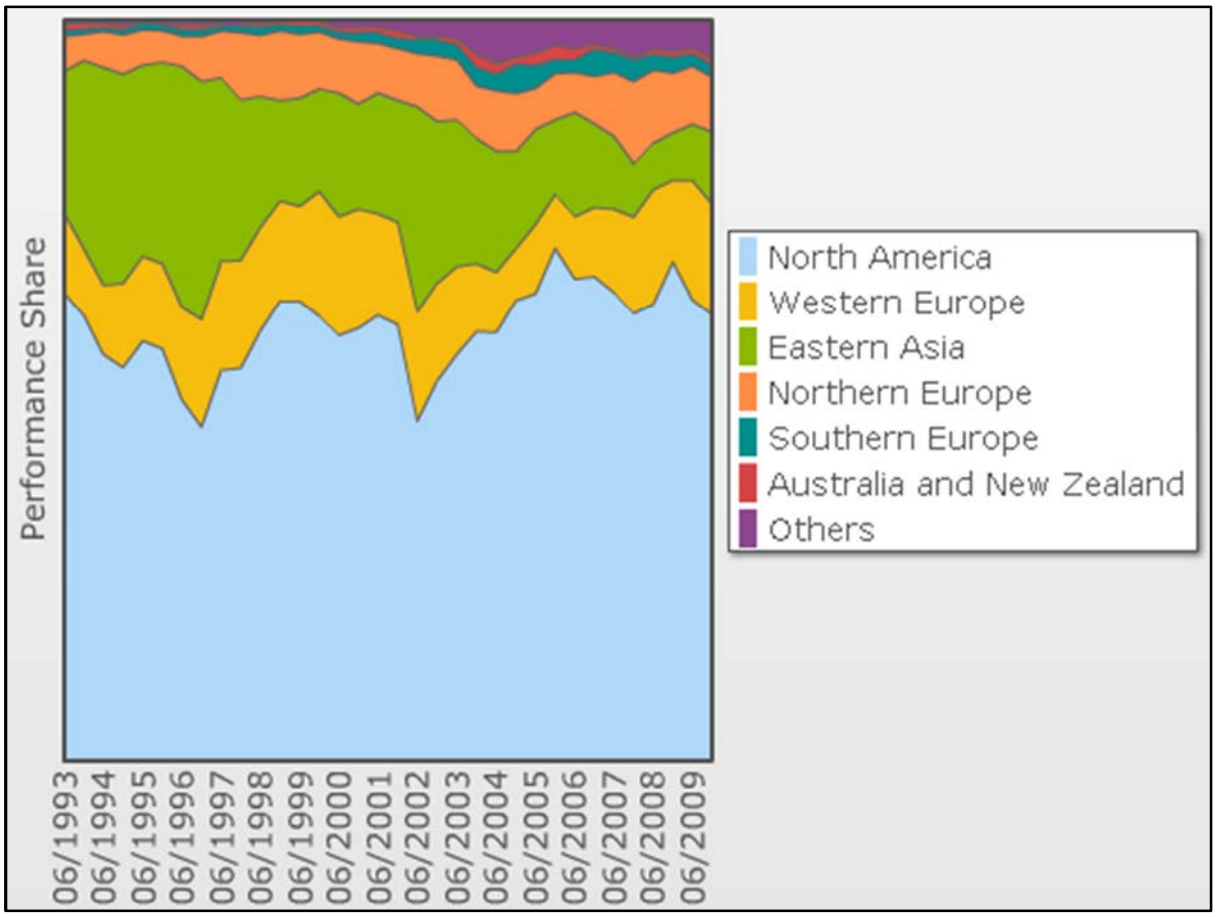


TOP500 Länder (% Leistung)

Wieder sieht man die Spitzen der Hitachi und des NEC Earth Simulator (gelb!).

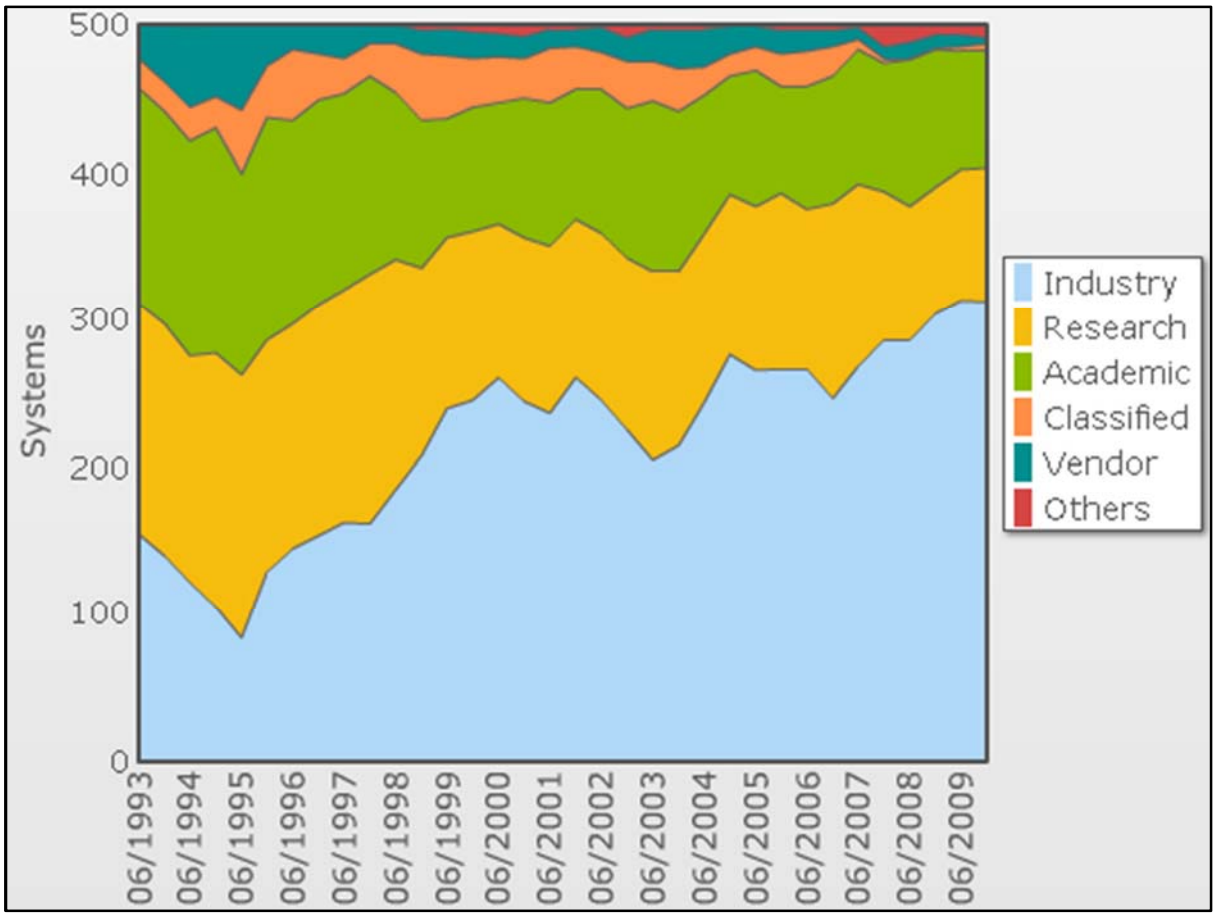


TOP 500 Geographische Regionen (# Systeme)



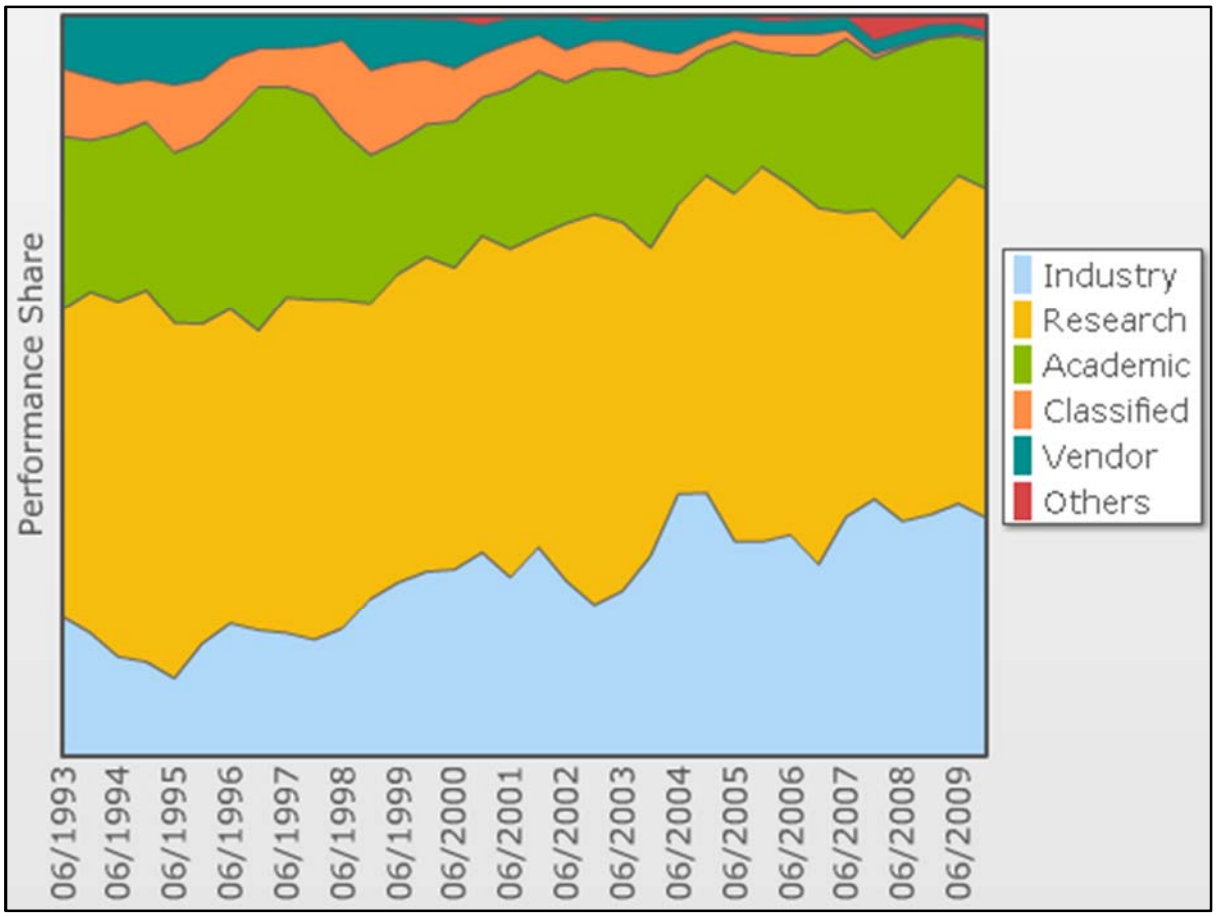
TOP500 Geographische Regionen (% Leistung)

Japan deutlich zu sehen (grün!).



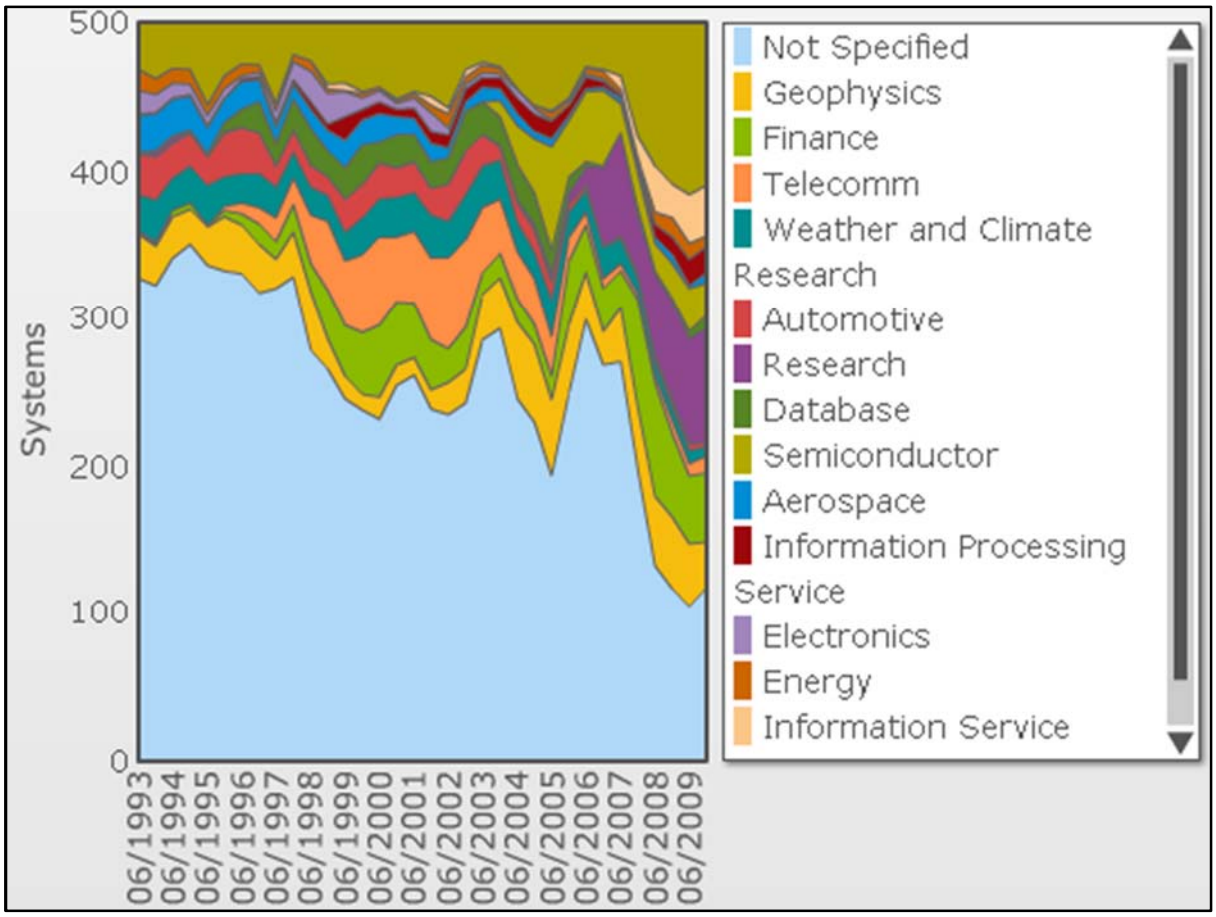
TOP500 Anwendergebiet (# Systeme)

Es werden zunehmend mehr Systeme in der Industrie installiert, d.h. das Hochleistungsrechnen findet Eingang in die Praxis. Insbesondere Ingenieurwissenschaften und Finanzwirtschaft.

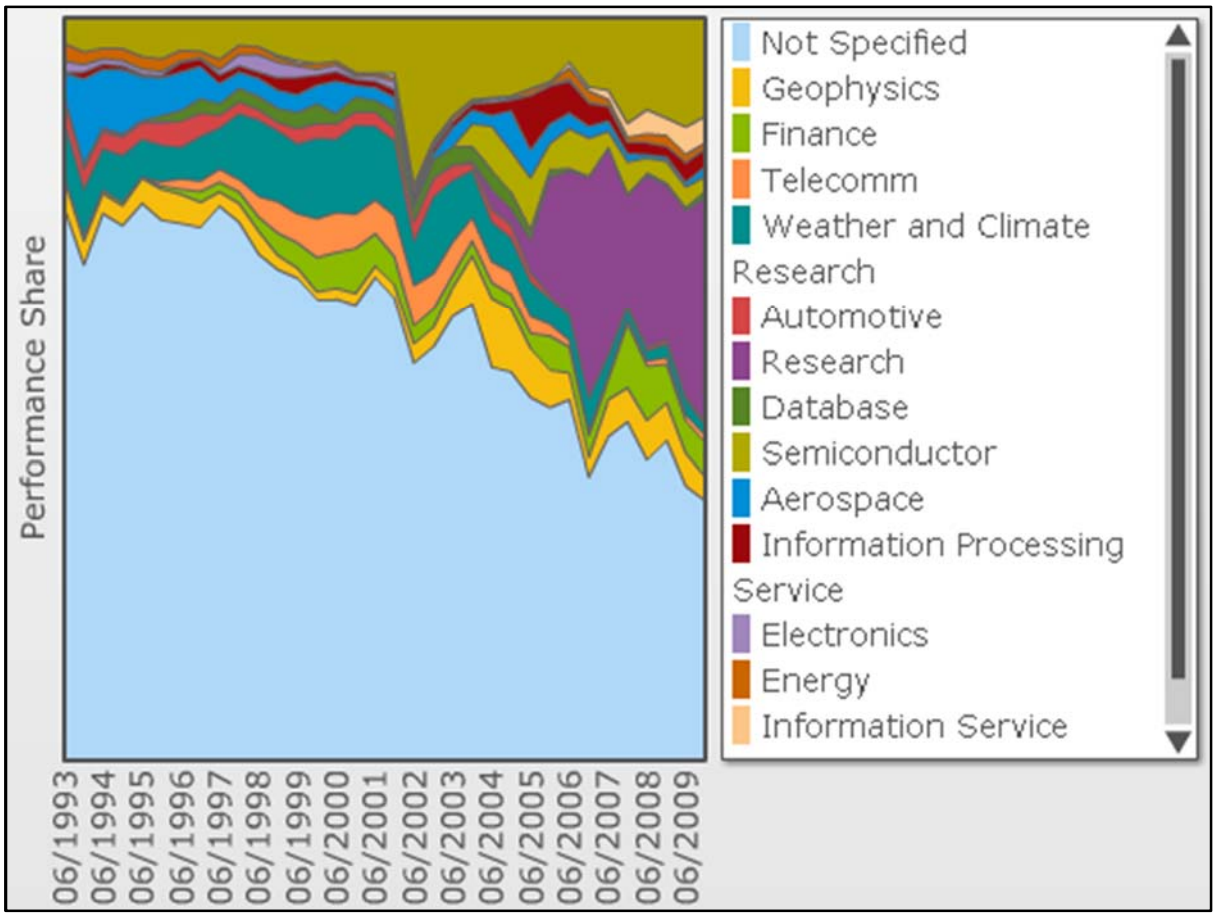


TOP500 Anwendergebiet (% Leistung)

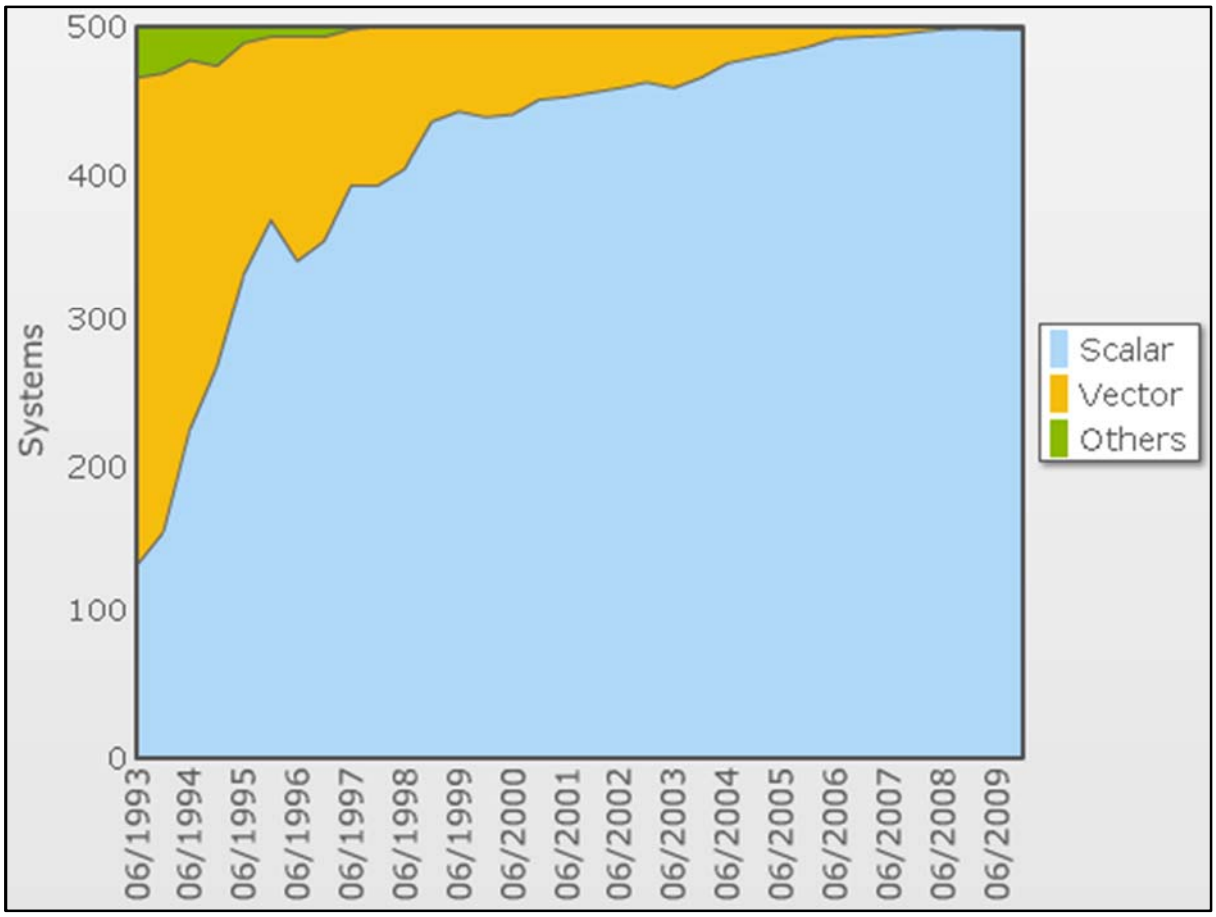
Das leistungsstärkste Gebiet bleibt aber die Forschung. D.h. die Industrie installiert eher kleine bis mittlere Systeme.



TOP500 Anwendungsgebiet (# Systeme)

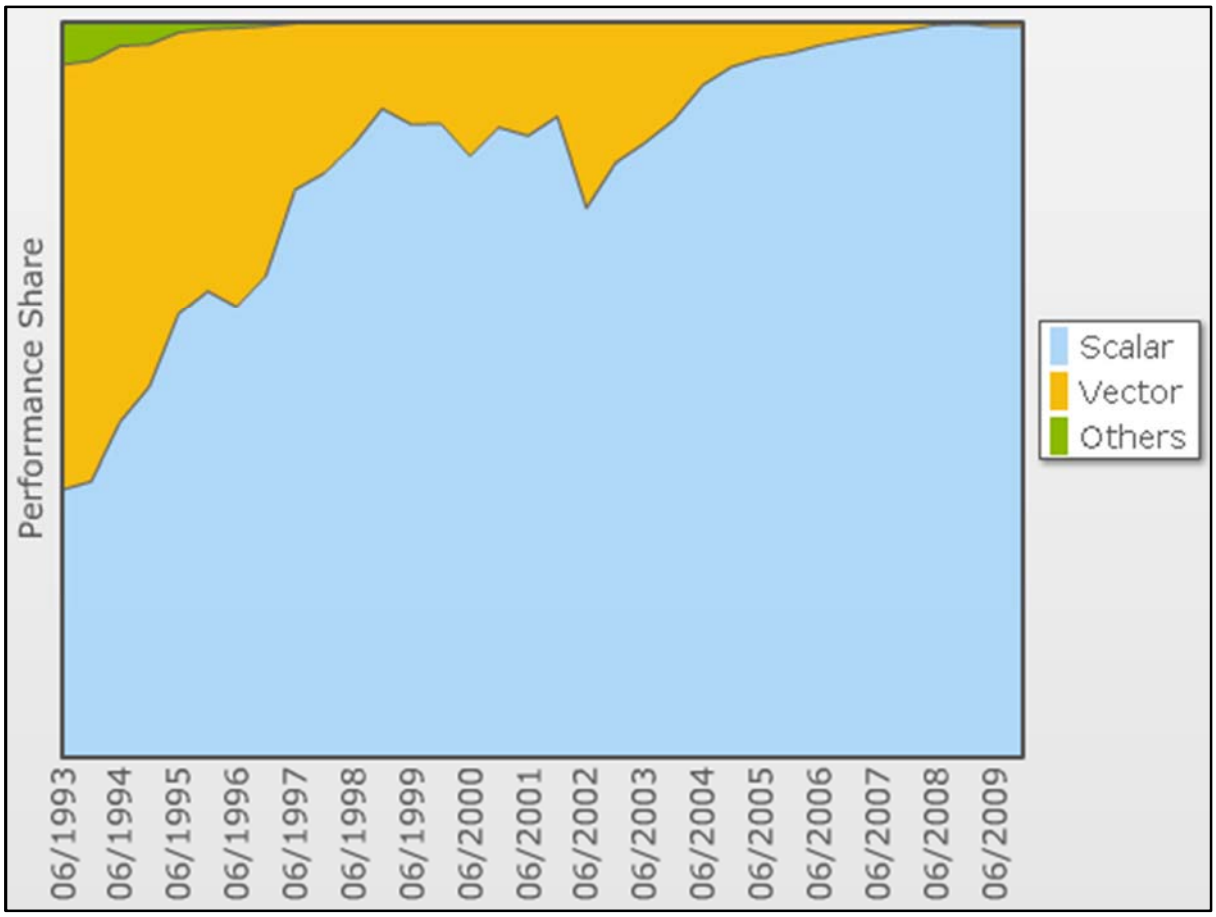


TOP500 Anwendungsgebiet (% Leistung)



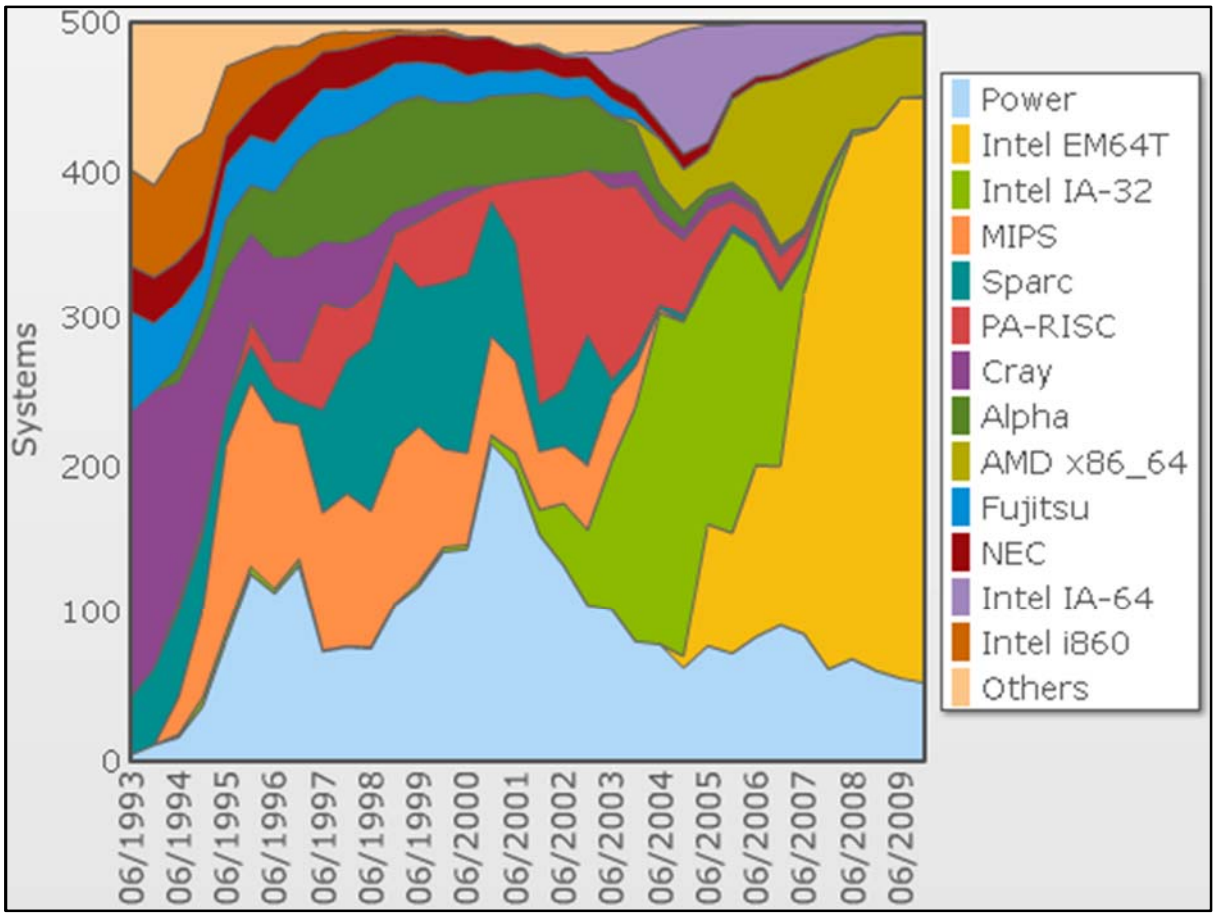
TOP500 Prozessorarchitektur (# Systeme)

Nur noch Skalarprozessoren.



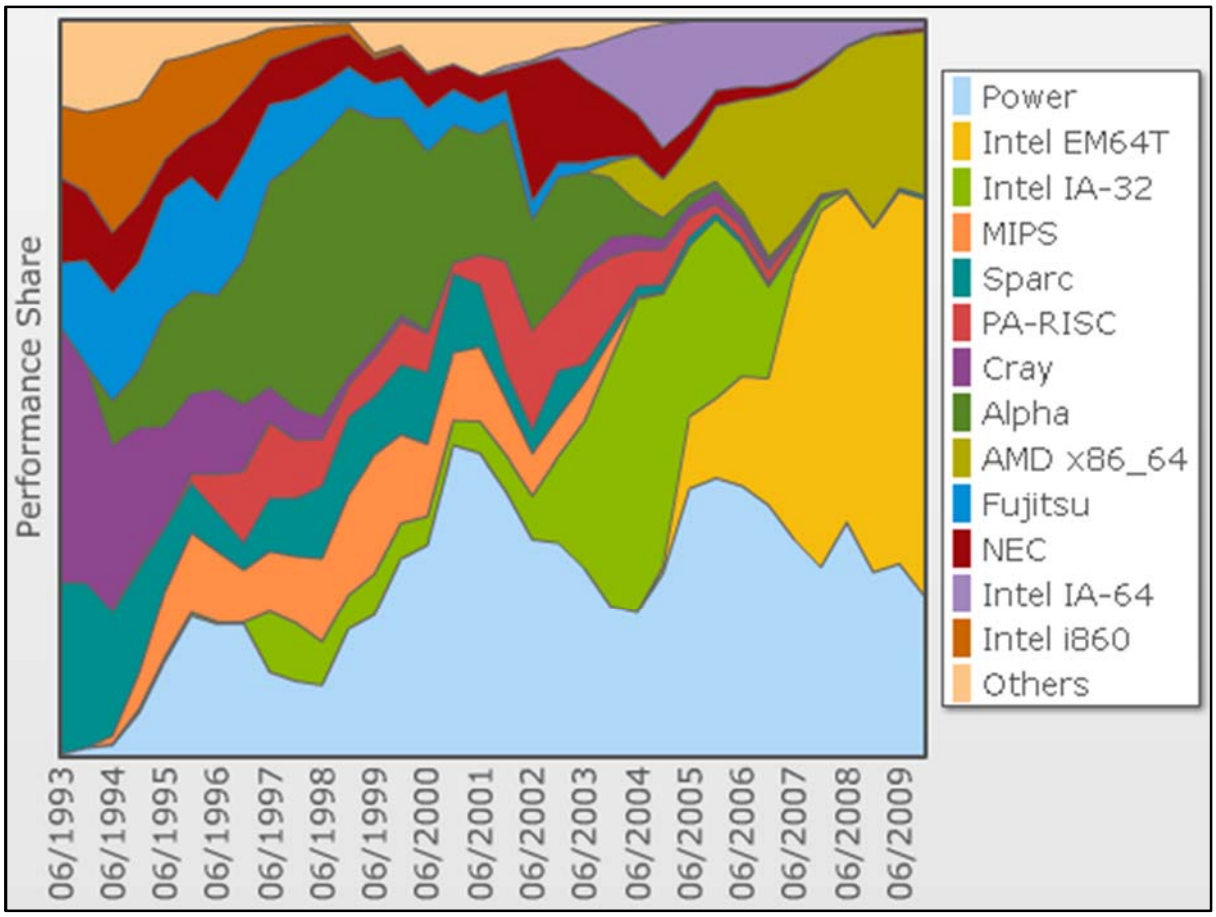
TOP500 Prozessorarchitektur (% Leistung)

06/2002 ein letztes Aufzucken der Vektorprozessoren im NEC Earth-Simulator.



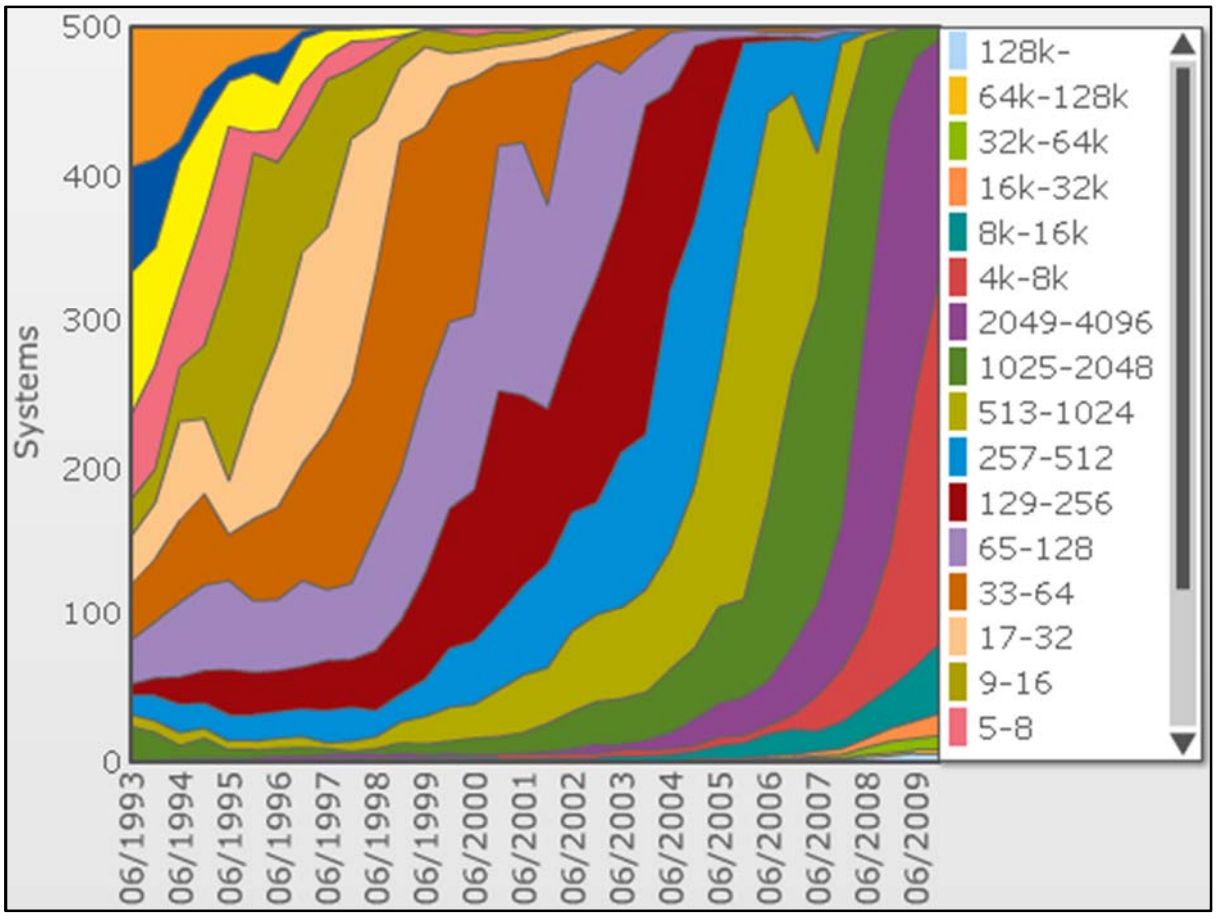
TOP500 Prozessorfamilie (# Systeme)

Intel dominiert die Anzahl der installierten Systeme vor Power und AMD.



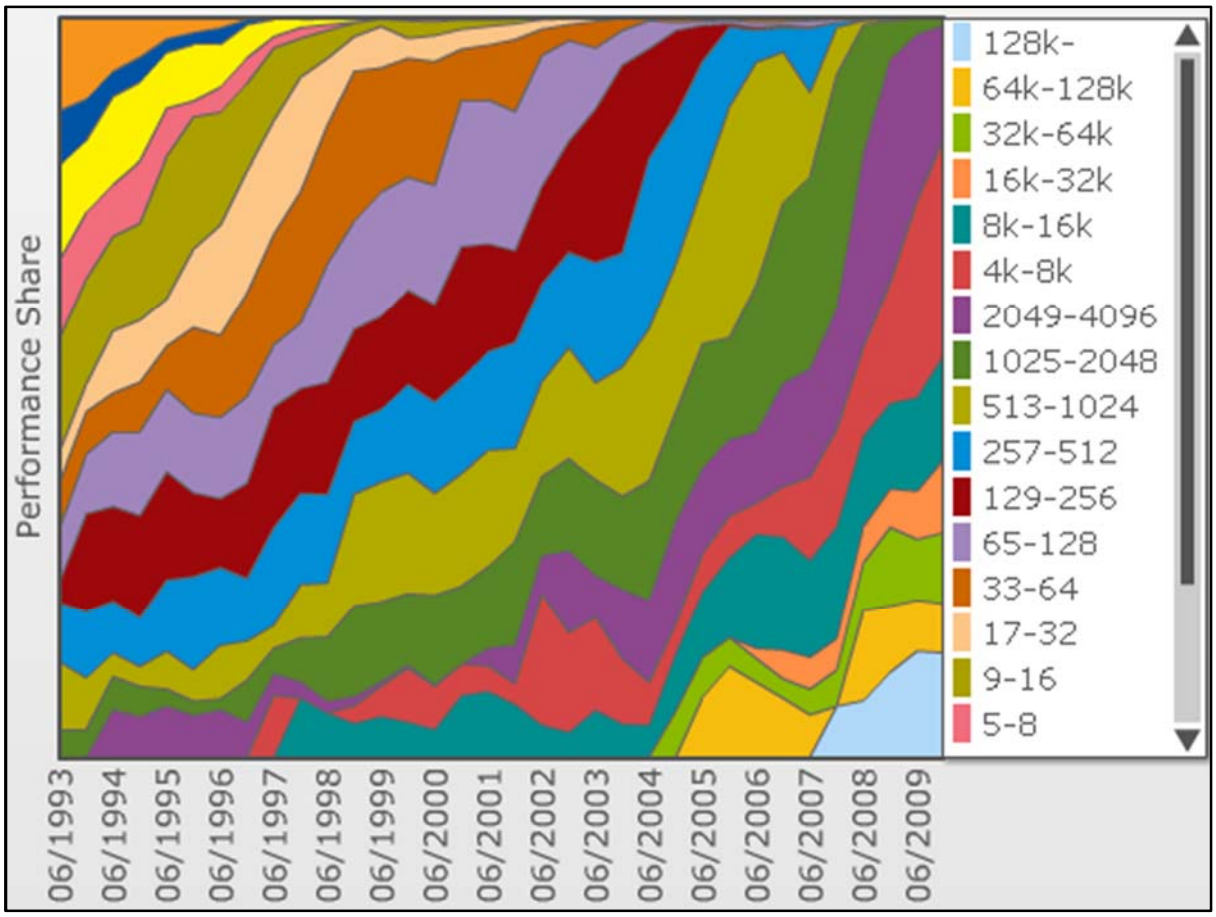
TOP500 Prozessorfamilie (% Leistung)

Im Leistungsanteil ist Power allerdings deutlich höher wegen der BlueGene-Systeme.



TOP500 Prozessoranzahl (# Systeme)

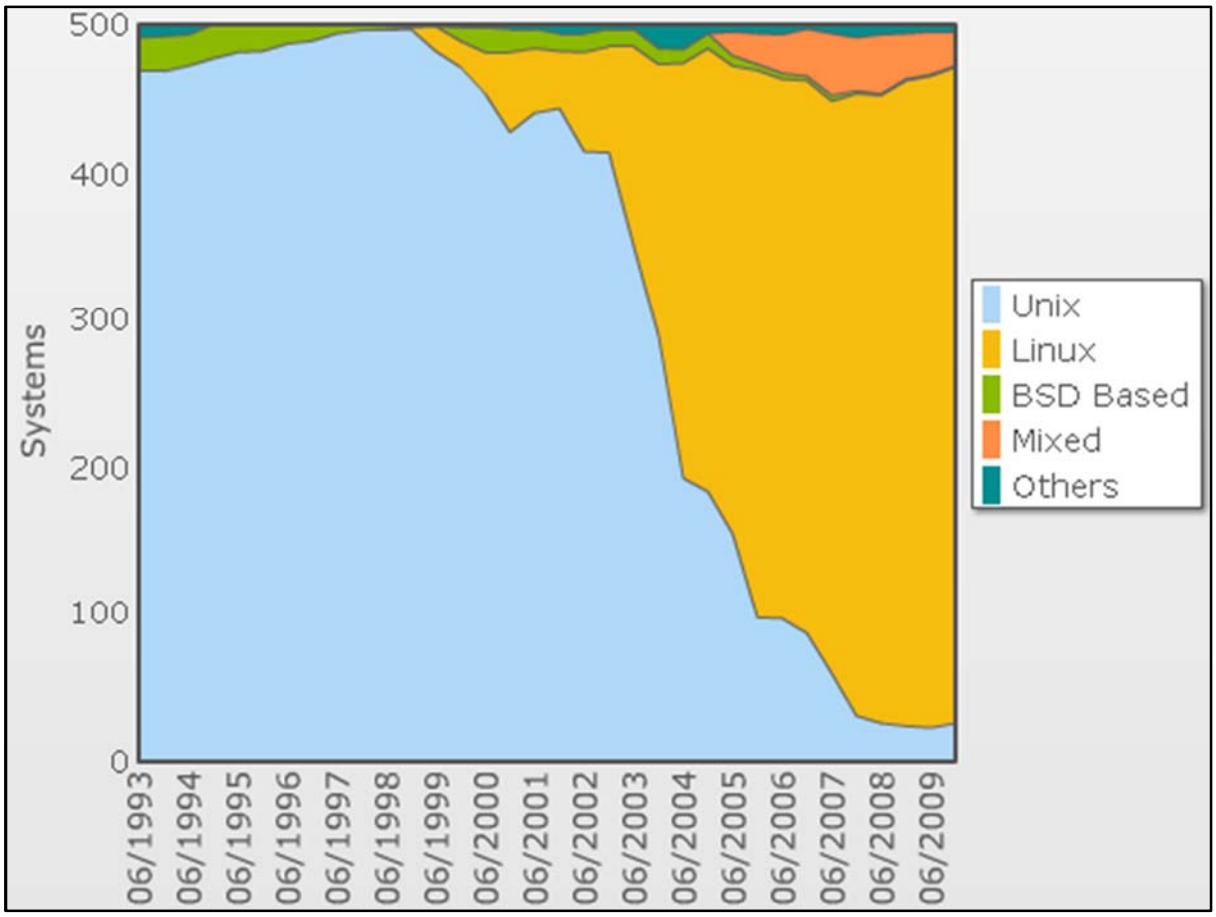
Nur wenige Systeme mit über 128K Prozessorkernen …



TOP500 Prozessoranzahl (% Leistung)

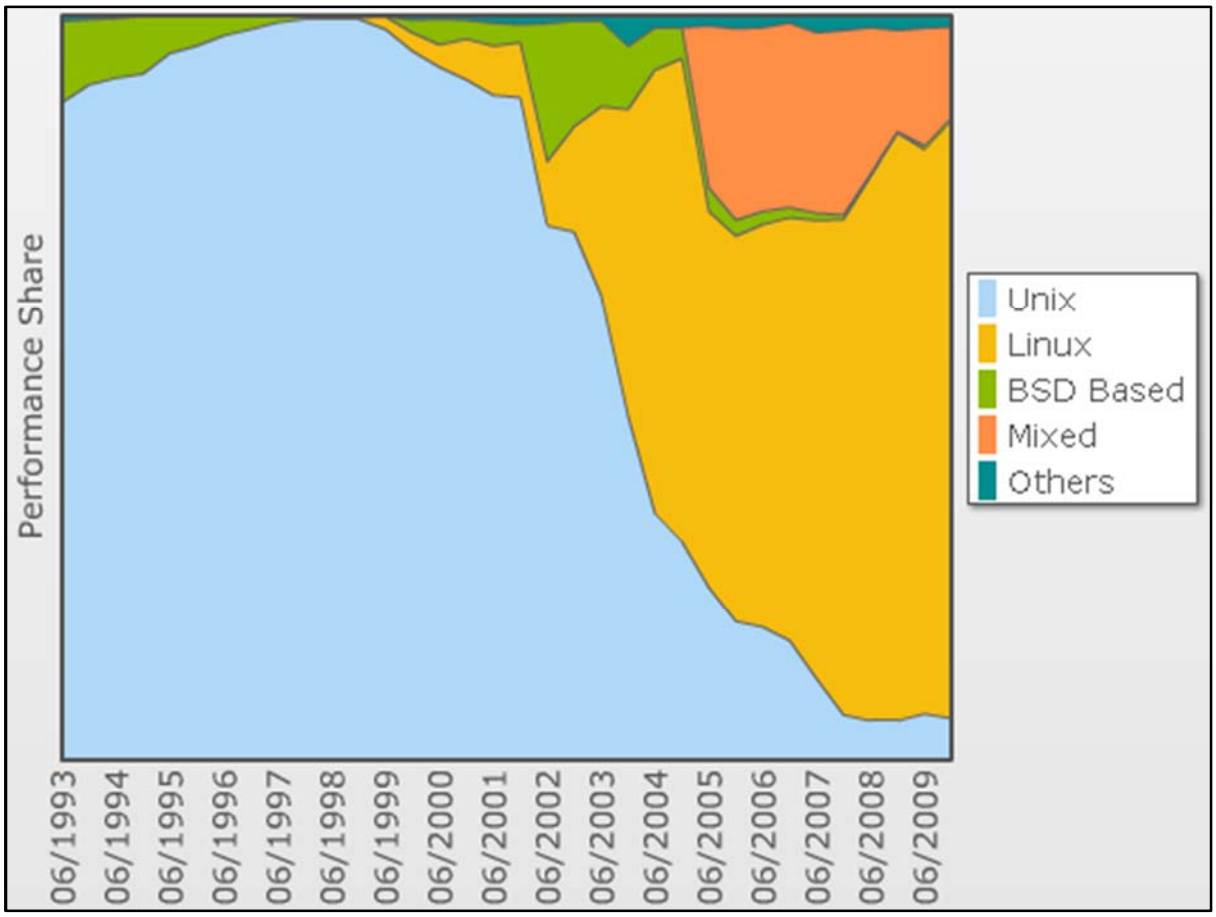
… aber die wenigen bringen einen deutlich sichtbaren Leistungsanteil (fast alles BlueGene-Systeme).

Interessant wäre eine Projektion für die kommenden zehn Jahre.



TOP500 Betriebssystemfamilie (# Systeme)

Linux dominiert die Betriebssysteme.



TOP500 Betriebssystemfamilie (% Leistung)

Gemischte Systeme anteilig stark wegen des Einsatzes auf BlueGene.

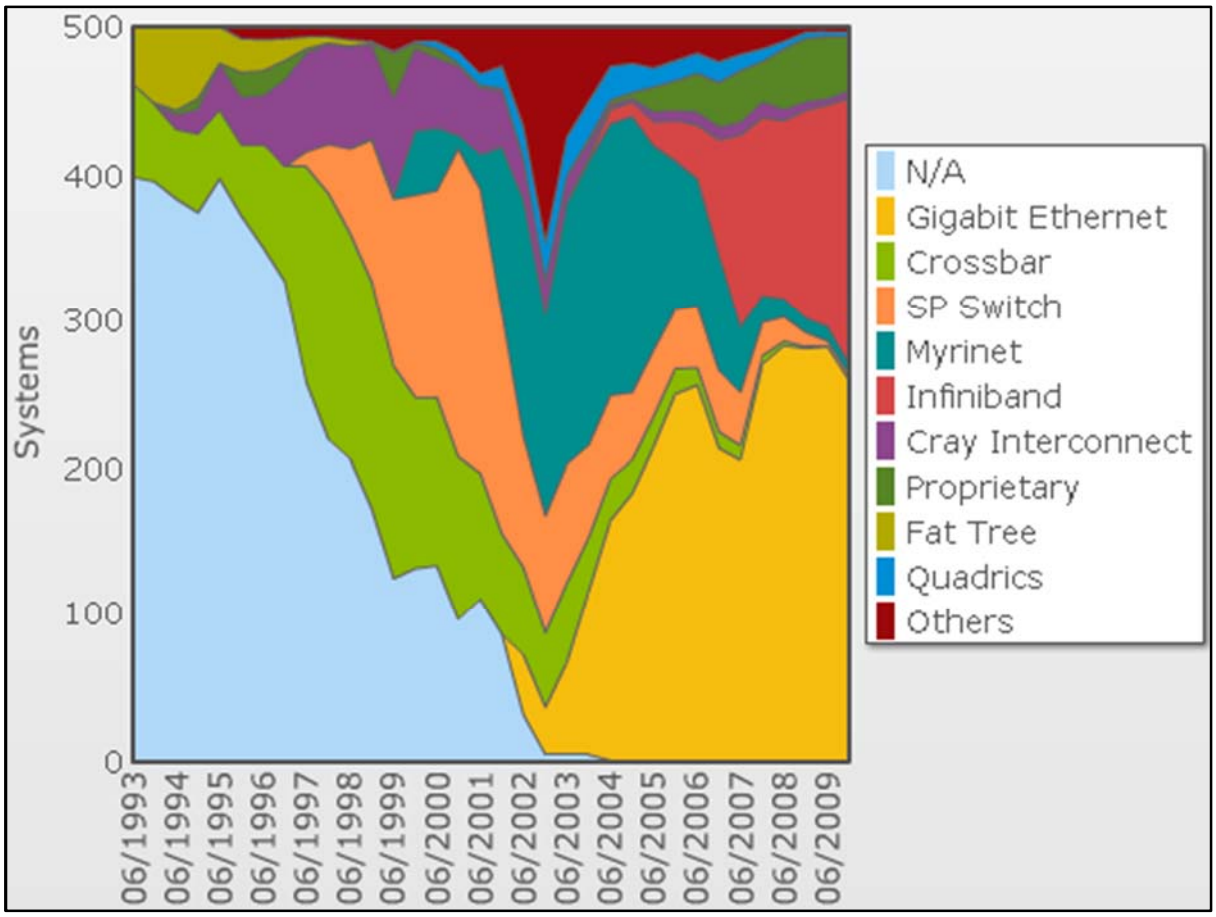
Verbindungstechnologien

In dedizierten Parallelrechnern

- Spezielle Hochleistungsnetze mit verschiedenen Topologien

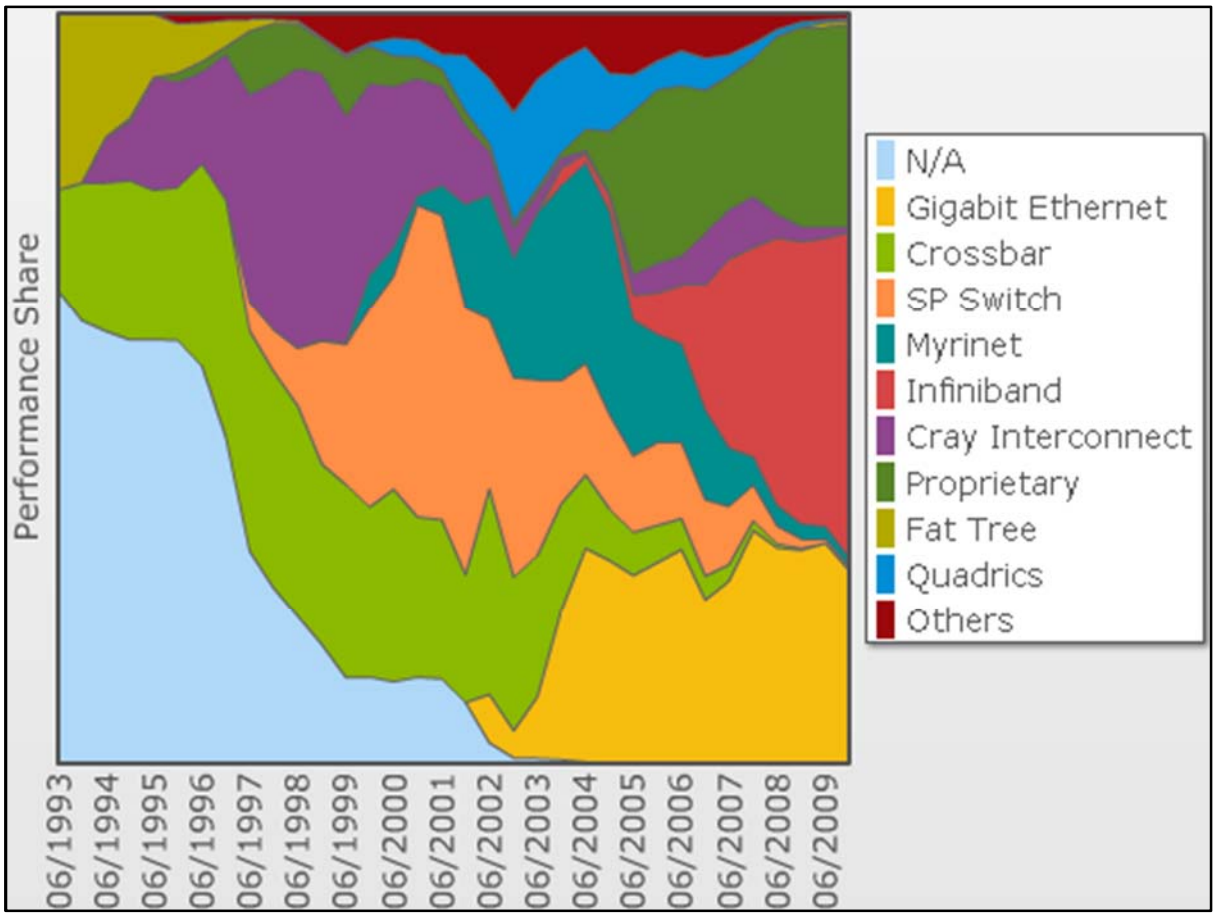
In Cluster-Architekturen

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



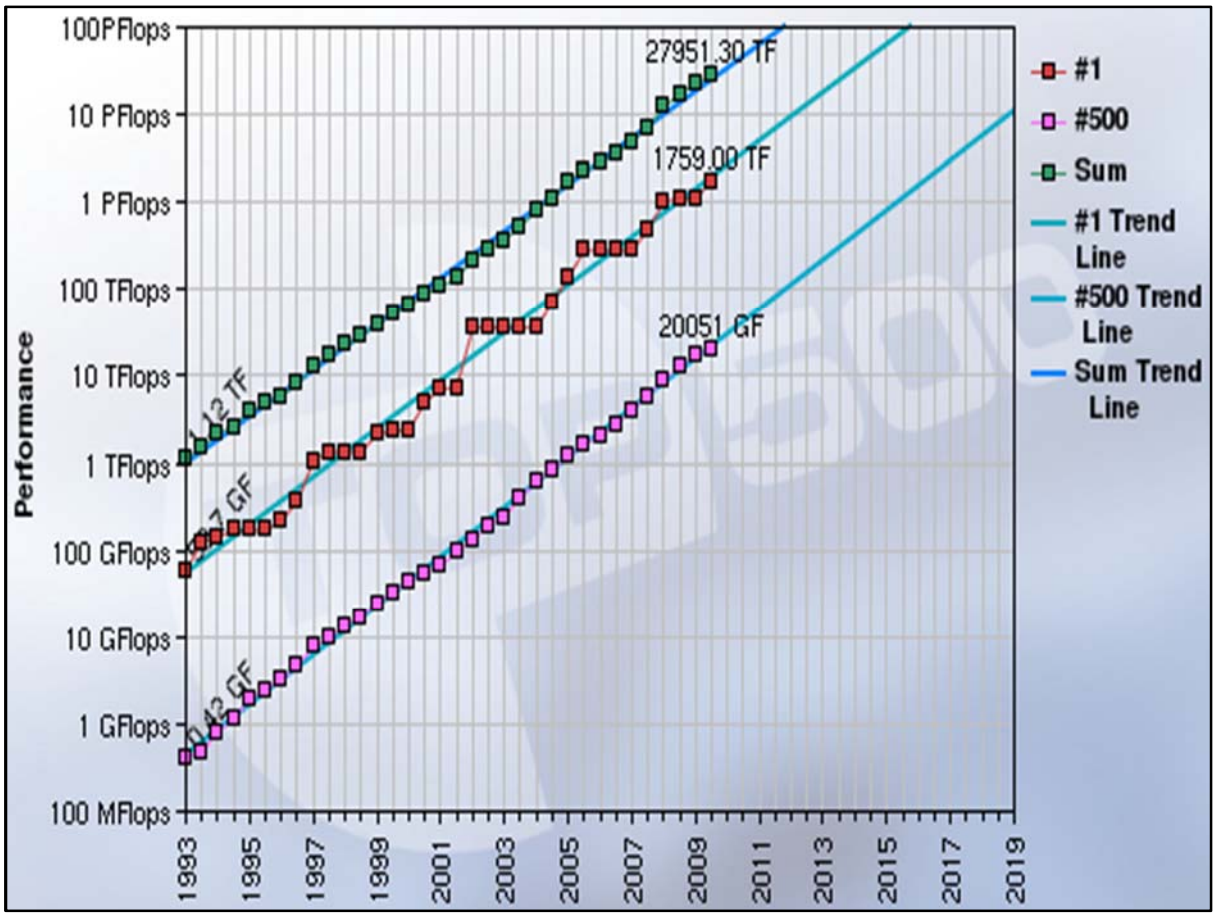
Verbindungstechnologie / Systeme

Gigabit-Ethernet dominiert die Installationen.

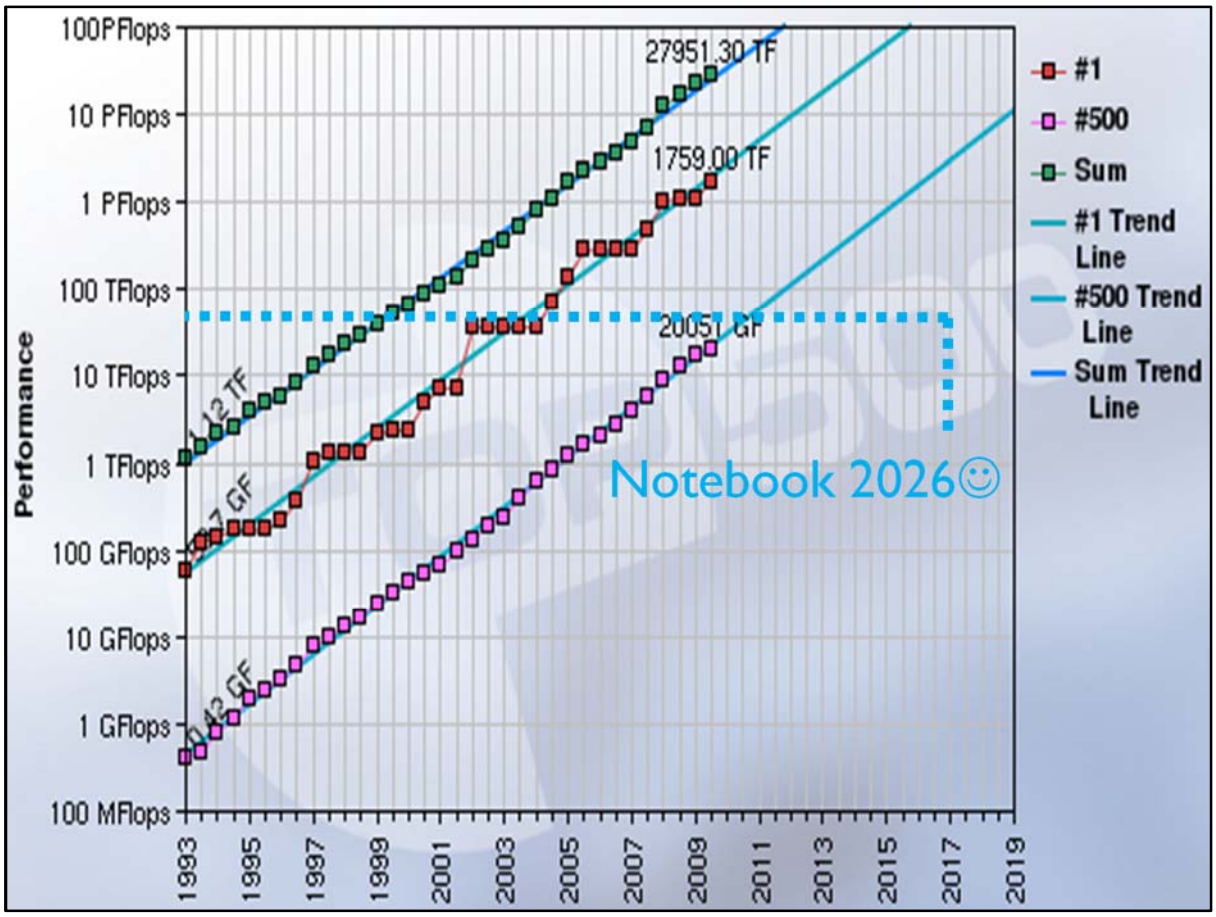


Verbindungstechnologie / Leistung

Allerdings hält Gigabit einen kleinen Leistungsanteil, weil es meist in kleineren und mittleren Systemen verwandt wird. Infiniband ist das wichtigste Netz für leistungssfähige Systeme. BlueGene hat ein proprietäres Netz, wie deutlich erkennbar ist.



Wir sind auf dem Weg zum Exaflops-Computing.



Notebook 2026: Faktor 30.000 über dem von 2010

Leistungsentwicklung bis November 2009

Moore´s Law

„Verdopplung der Transistorzahl alle 18 Monate“

(entspricht evtl. Leistungsverdopplung)

Jun93–Nov09: 16,5 Jahre = 11x18 Monate

Faktor 2^{11}=2048

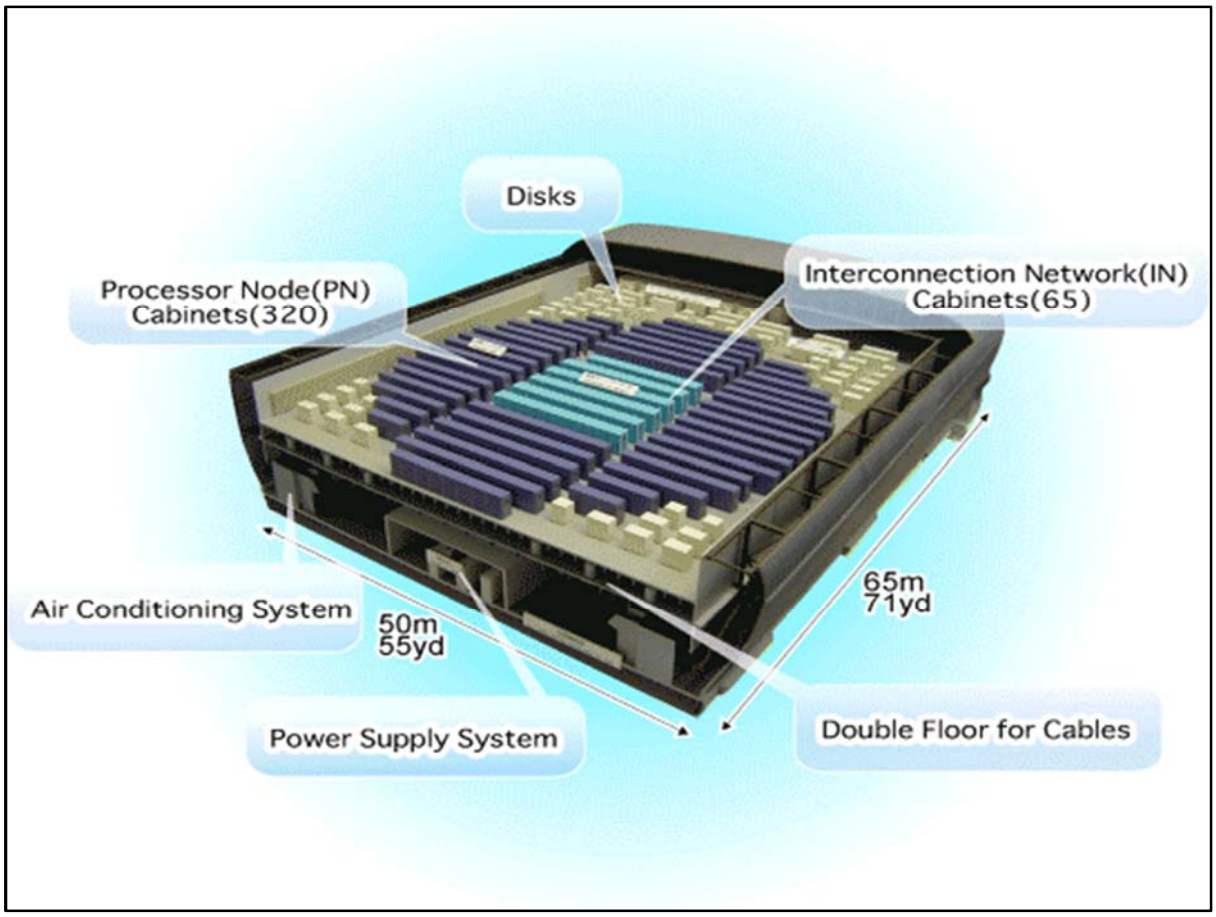
Leistung Summe: 1 TFlop/s – 27.951 TFlop/s (x 27.951)

Leistung #1: 60 GFlop/s – 1.759 TFlop/s (x 29.300)

Leistung #500: 0,4GFlop/s – 20.041 GFlop/s (x 50.100)

Siehe: http://en.wikipedia.org/wiki/Moore%27s_law

Die Installation des Rechners traf die Amerikaner gleichermaßen schwer wie damals, als die Russen den Sputnik ins All schossen. Daher die Bezeichnung Computenik-Shock.



Quelle: http://www.jamstec.go.jp/e/

Erdbebensicheres Gebäude des Earth Simulator von NEC.

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

Siehe: http://en.wikipedia.org/wiki/Bluegene

Historische Sicht

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

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

TOP500 im Juni 1993

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

TOP500 Deutschland im Juni 1993

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

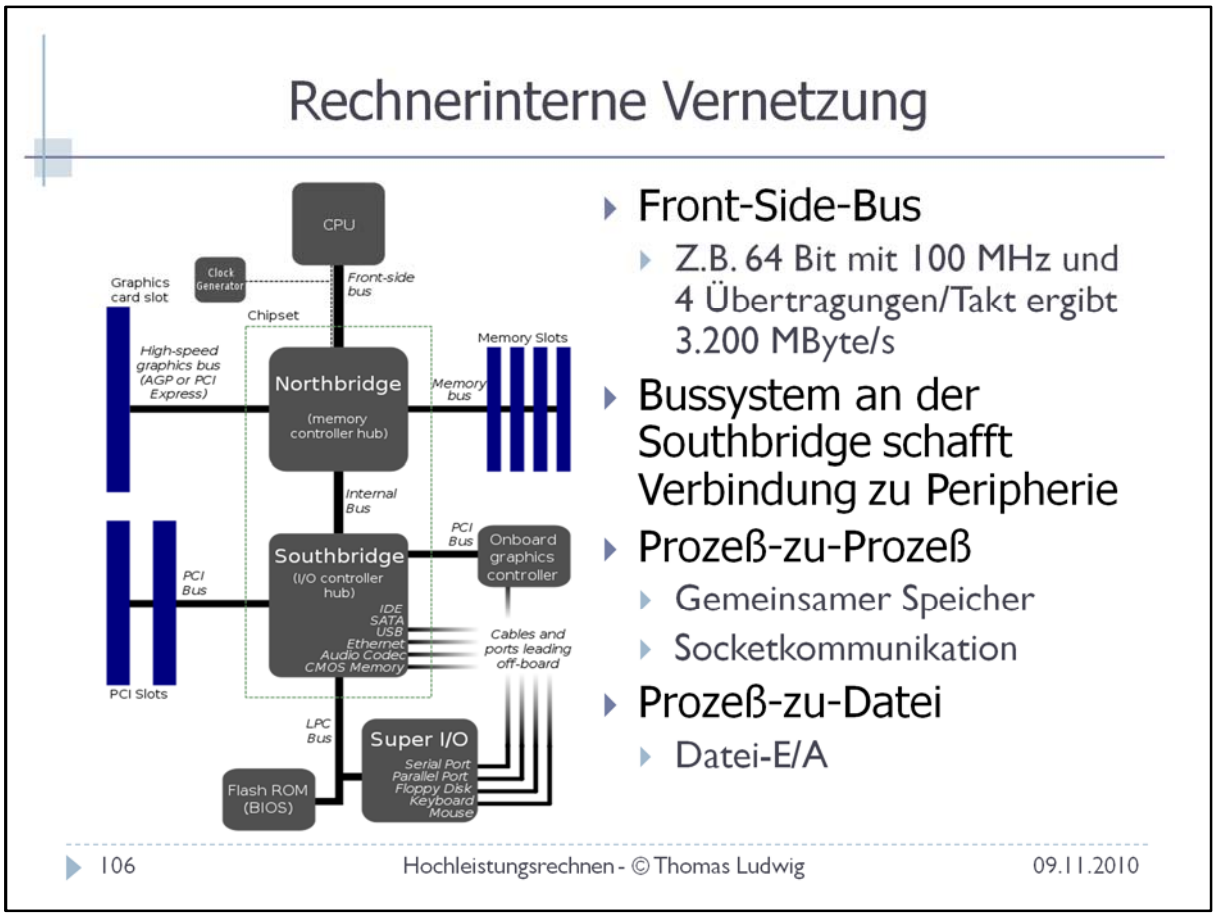
- Welche Aufgaben hat die Vernetzung?
- Welche Komponenten weist eine Vernetzung auf?
- Welche Fragen finden wir bei der Pufferverwaltung?
- Was bedeutet Überlagerung von Berechnung und Kommunikation?
- Was versteht man unter Hardware-Realisierung von Software?
- Welche Bandbreiten finden wir bei aktuellen 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

Die Vernetzung (Hardware und Software) unterstützt die Interprozeßkommunikation und die Prozeß-E/A.



Quelle: http://en.wikipedia.org/wiki/File:Motherboard_diagram.svg

Im Rechner finden wir verschiedene Bussysteme, die die Komponenten in Verbindung bringen. Softwareseitig verwenden wir zur Interprozeßkommunikation Socketprogrammierung basierend auf TCP/IP. Alternativ können gemeinsame Speicherbereiche verwandt werden. Hier gibt es aber keine genormte Programmierschnittstelle.

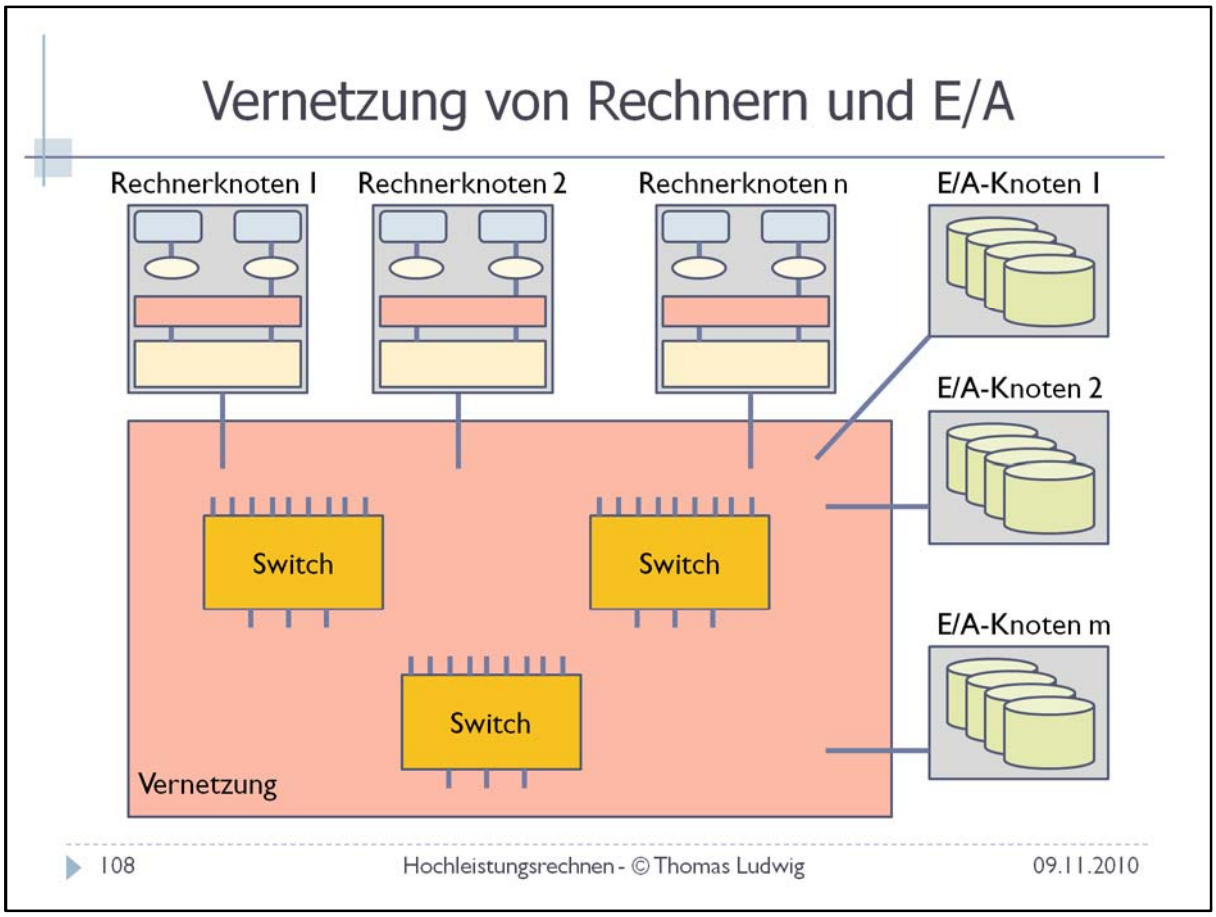
Vernetzung von Rechnern und E/A

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



Die Rechner und die E/A-Komponenten werden über Netze verbunden. Dies können getrennte oder gemeinsam genutzte Netzwerke sein. Die Verknüpfung erfolgt über Switche.

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

Quelle: Tutorial von Panda, Balaji, Koop: InfiniBand and 10-Gigabit Ethernet for Dummies, Supercomputing 2009.

Von 1998 bis 2009 in 12 Jahren ein Faktor von nicht mehr als 25. Bei Rechnern mehr als Faktor 1000.

Siehe auch: http://en.wikipedia.org/wiki/List_of_device_bandwidths - für verschiedenste Gerätetypen

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)

Quelle: Tutorial von Panda, Balaji, Koop: InfiniBand and 10-Gigabit Ethernet for Dummies, Supercomputing 2009.

In den 9 Jahren von 2001-2009 sehen wir bei Infiniband einen Faktor von 16. Bei den Rechnern grob einen Faktor von 300.

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

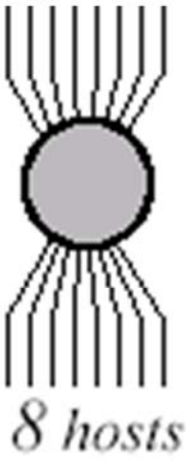
Warum interessiert uns das?

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

Wir betrachten einen Switch mit 64 Ports und 1 Gbit/s pro Port. Wenn 32 Sender mit 32 Empfängern kommunizieren wollen, benötigen wir 32 Gbit/s an Bandbreite durch den Switch hindurch. Bidirektional das doppelte. Die interne Topologie des Switches sollte also diese Gesamtleistung ermöglichen. In der Praxis ist sie meist geringer.

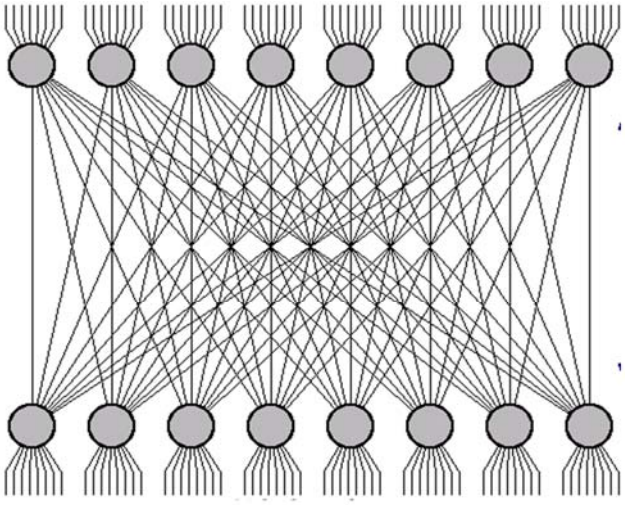
Bisektion am Beispiel Myrinet

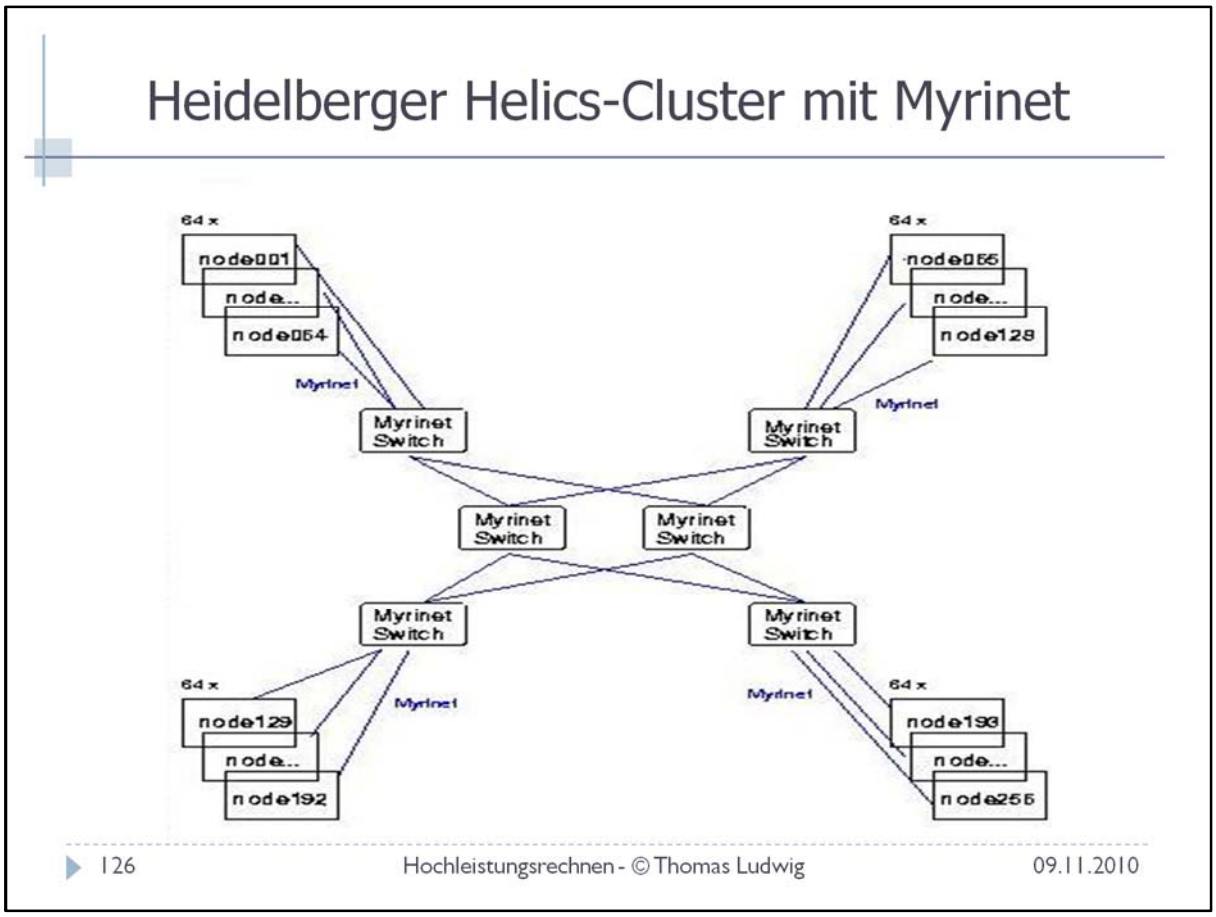
- Basisbaustein: Kreuzschiene mit 16 Anschlüssen
- Topologie benannt nach Charles Clos (1952)



Myrinet-Switch

Neue Basiskomponente aus 16 des bisherigen Typs

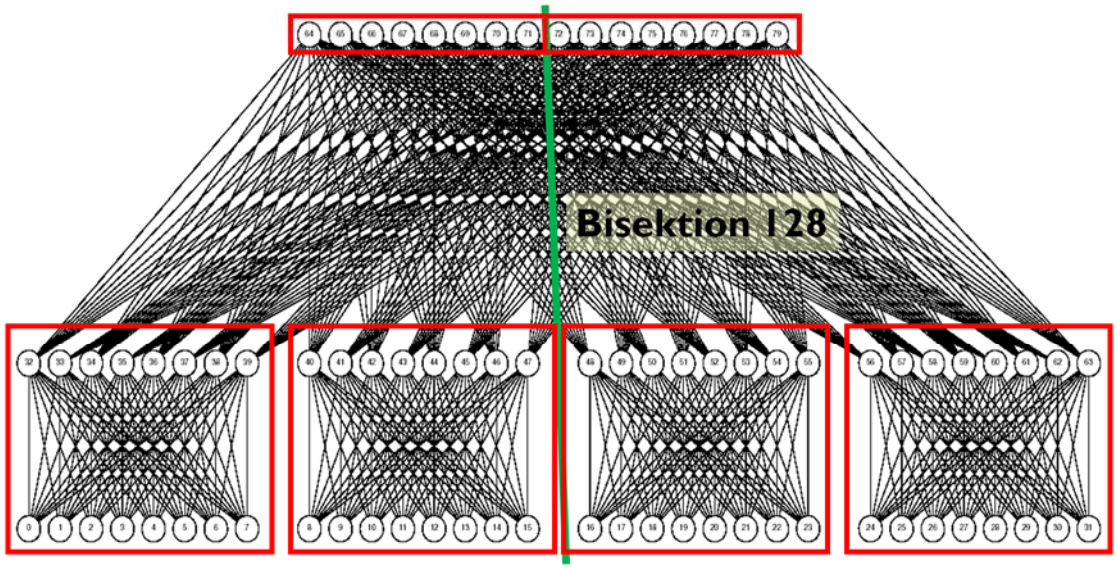




Es werden 6 Switches vom vorgestellten Typ verwandt. Je einer ist mit 64 Rechnerknoten verbunden. Die 4 Switches für 256 Knoten sind wiederum über 2 Switches miteinander verbunden.

Myrinet im Detail...

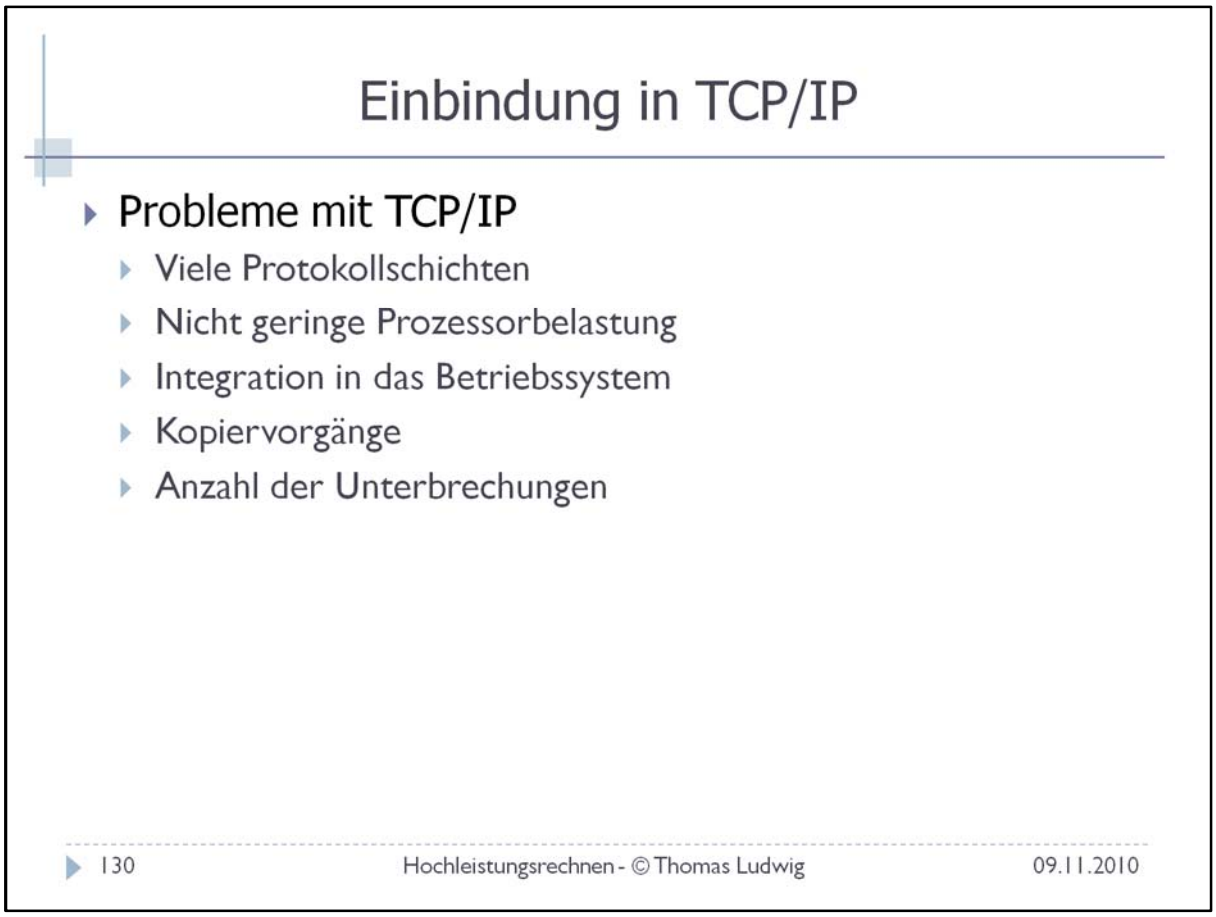
4 Clos64-Bausteine und 2 Switche für 256 Rechner



... 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



In den zahlreichen Protokollschichten werden Aktivitäten ausgeführt, die in einem lokalen Cluster ohne Bedeutung sind, aber Software-Aufwand hervorrufen. Z.B. die Überlaststeuerung zwischen entfernteren Knoten.

Die umfangreichen Software-Schichten bedeuten auch eine nicht vernachlässigbare Belastung des Prozessor, der diese ja abarbeiten muß.

Die Integration in das Betriebssytem ist nachteilig, da die Anwendung ja immer erst durch dieses hindurch auf die NIC zugreifen muß, anstatt einen eigenen Weg zu haben.

Beim Durchlaufen der Schichten und beim Übergang zwischen Betriebssystem und Anwenderprogramm kommt es immer wieder zu zeitaufwendigen Kopiervorgängen.

Aufgrund der maximalen MTU (maximum transmission unit) von 1500 Byte kommt es auch bei GigaBit-Ethernet zu hohen Raten von Unterbrechungen. Größere Frames (Jumbo-Frames) schaffen hier ein wenig Abhilfe.

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

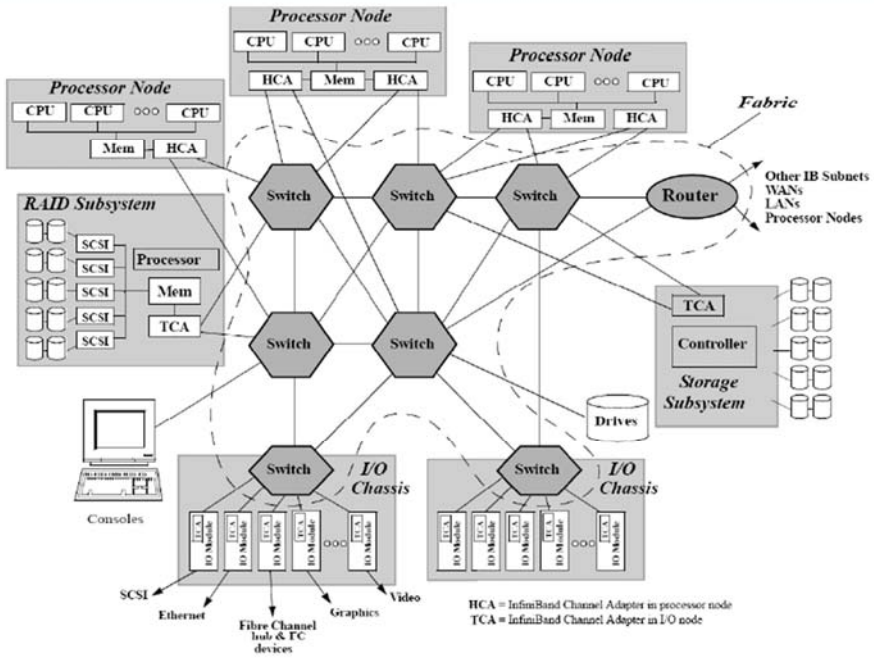
- Lösungsansätze...
  - Remote Direct Memory Access (RDMA)
    - Die NIC kann direkt in den Speicher eines entfernten Rechners schreiben
  - Zero-Copy Networking
    - Vermeide Kopien zwischen Betriebssystem und Anwendungprogramm
    - Verwende stattdessen DMA und MMU
  - Interrupt Mitigation
    - Fasse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

Siehe: http://de.wikipedia.org/wiki/InfiniBand

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

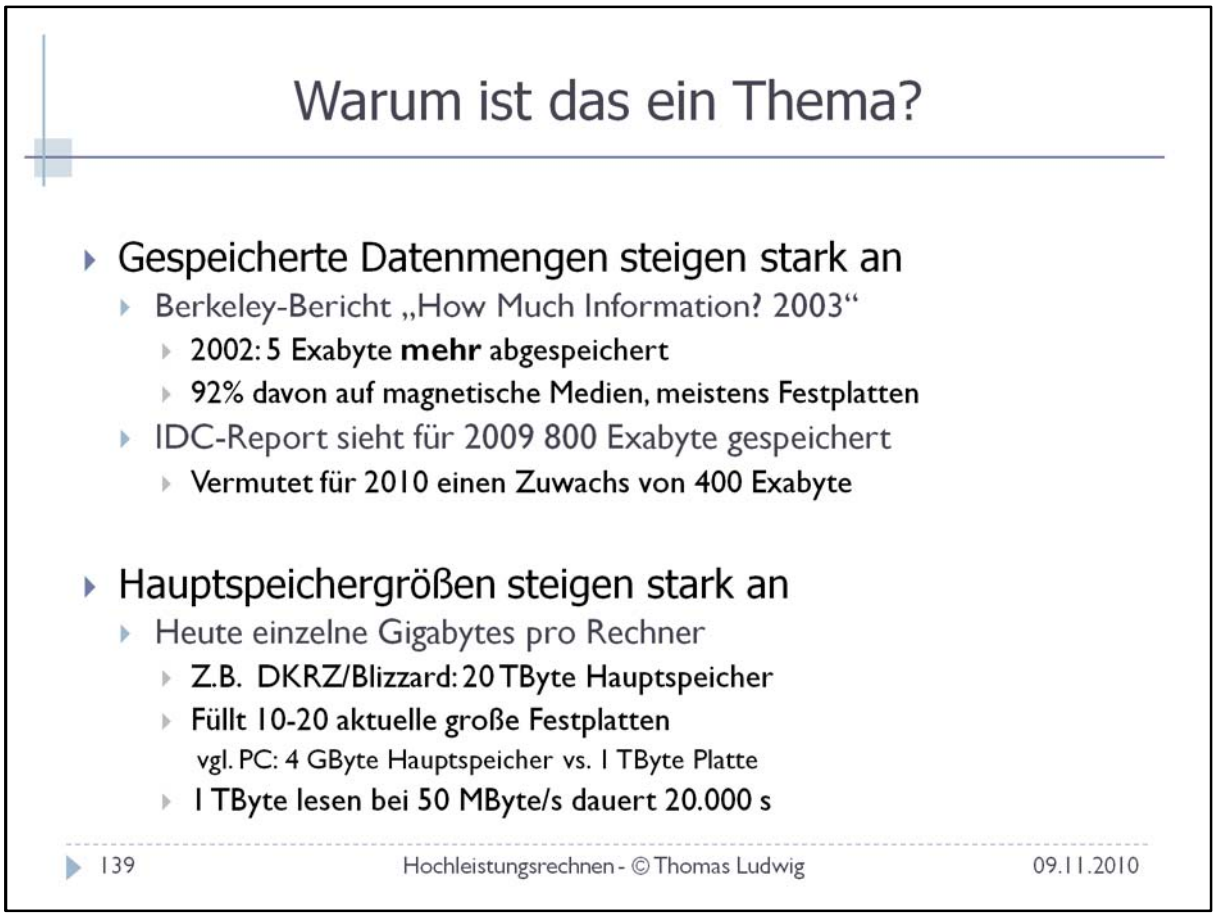
- Die Vernetzung bringt Rechner miteinander und mit den E/A-Geräten in Verbindung
- Vernetzungshardware sind Netzadapter, Kabel, Switches
- Wir möchten hohe Datenraten und geringe Latenzen bei gleichzeitiger guter Skalierbarkeit
- Effiziente Kommunikation umfaßt viele Einzelaspekte
- In den letzten 10 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

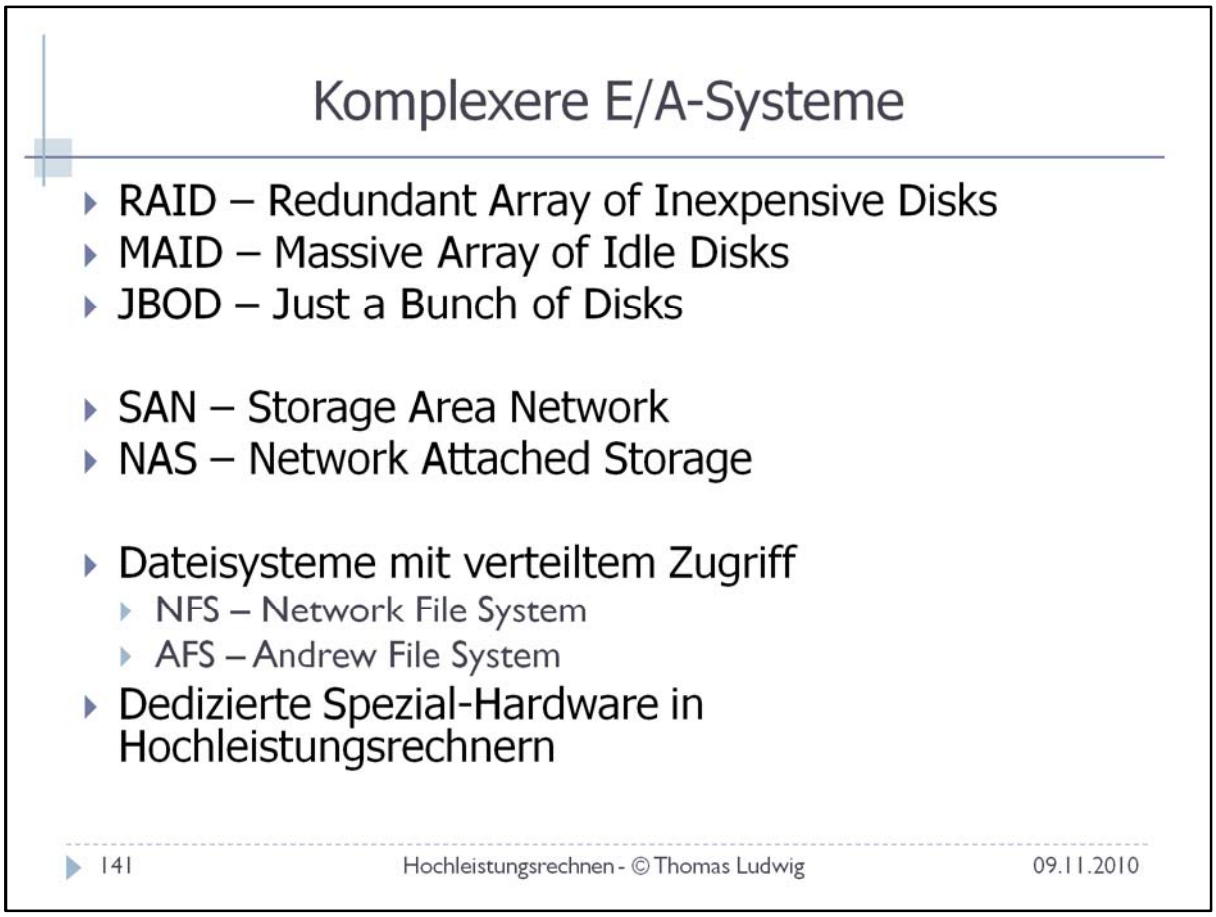
- Warum ist E/A auf einmal ein Thema?
- Wie hantiert man mit Daten in Dateien?
- Welche Abstraktionsschichten unterscheidet man?
- Welche Varianten der E/A finden wir bei numerischen Anwendungen?
- Welche Benutzungsschnittstellen gibt es?
- Wie funktioniert E/A im Cluster?
- Wie funktioniert E/A im Hochleistungsrechner?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



Siehe:
• http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/
• http://insidehpc.com/2010/10/14/zettabytes-petabytes-and-all-that-the-intersection-of-supercomputing-and-all-the-data-we-create/

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



Siehe: http://en.wikipedia.org/wiki/RAID , http://en.wikipedia.org/wiki/MAID , http://en.wikipedia.org/wiki/JBOD#JBOD

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

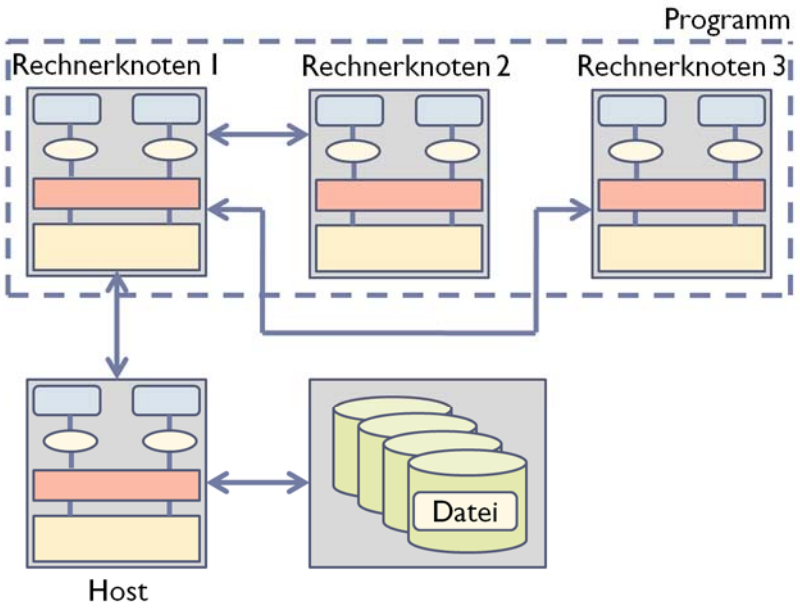
- Traditionell
  - E/A nur über Hostrechner
  - E/A nur durch festgelegten Prozeß

  Bewertung
  - Aus Effizienzgründen nicht mehr möglich

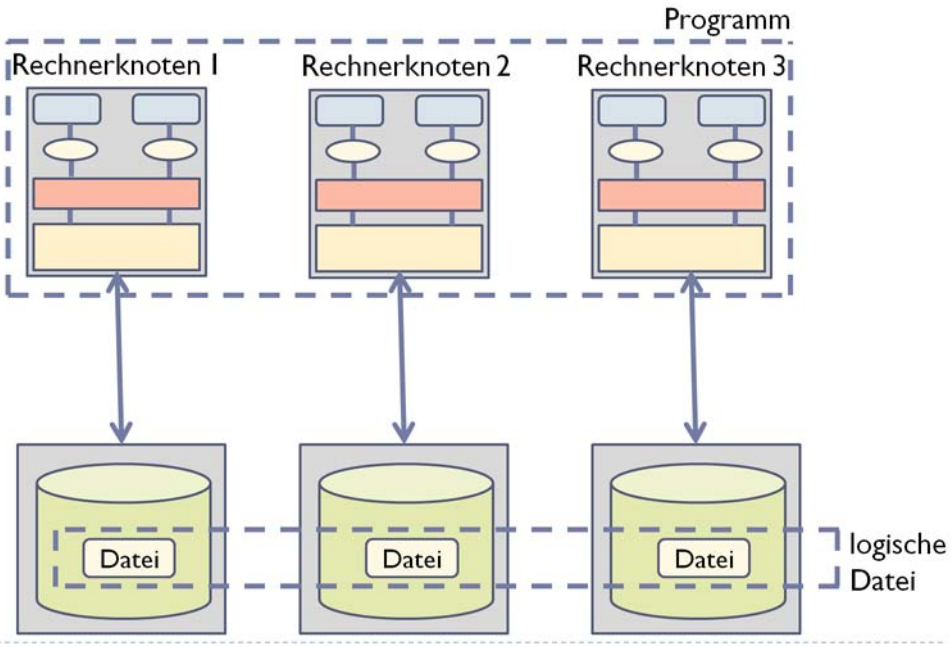
- Modern
  - E/A über viele Knoten
  - E/A durch alle Prozesse

  Bewertung
  - Verwendung moderner Techniken

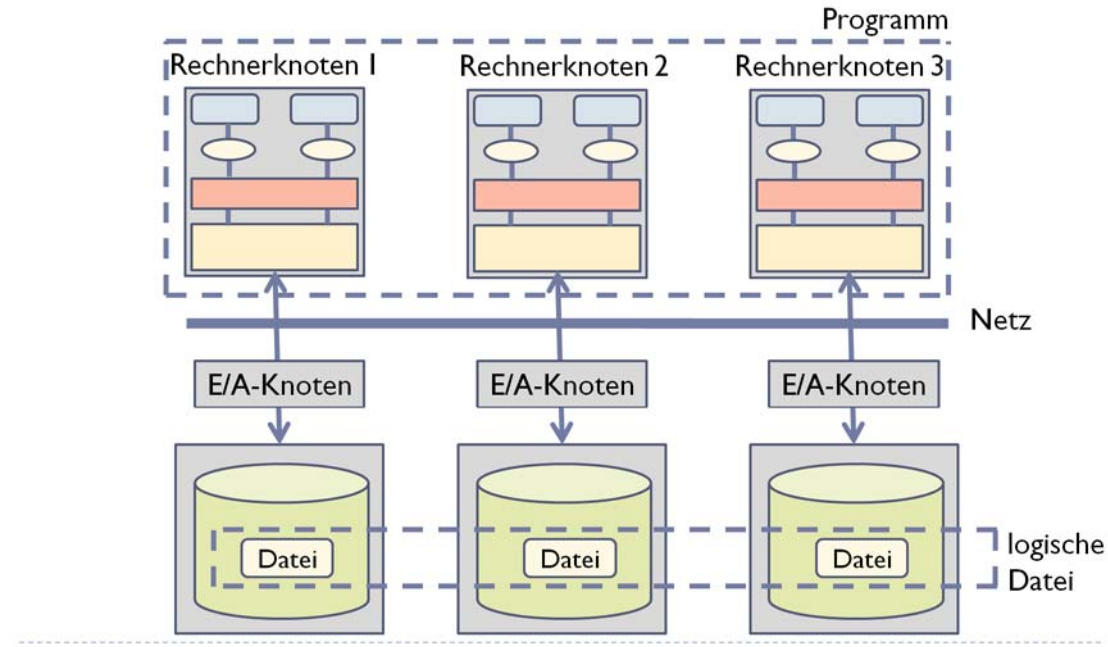
Traditionelle E/A



Moderne E/A


Programm
Rechnerknoten 1
Rechnerknoten 2
Rechnerknoten 3
Datei
Datei
Datei
logische
Datei

Moderne E/A...


Programm
Rechnerknoten 1
Rechnerknoten 2
Rechnerknoten 3
Netz
E/A-Knoten
E/A-Knoten
E/A-Knoten
Datei
Datei
Datei
logische
Datei

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 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

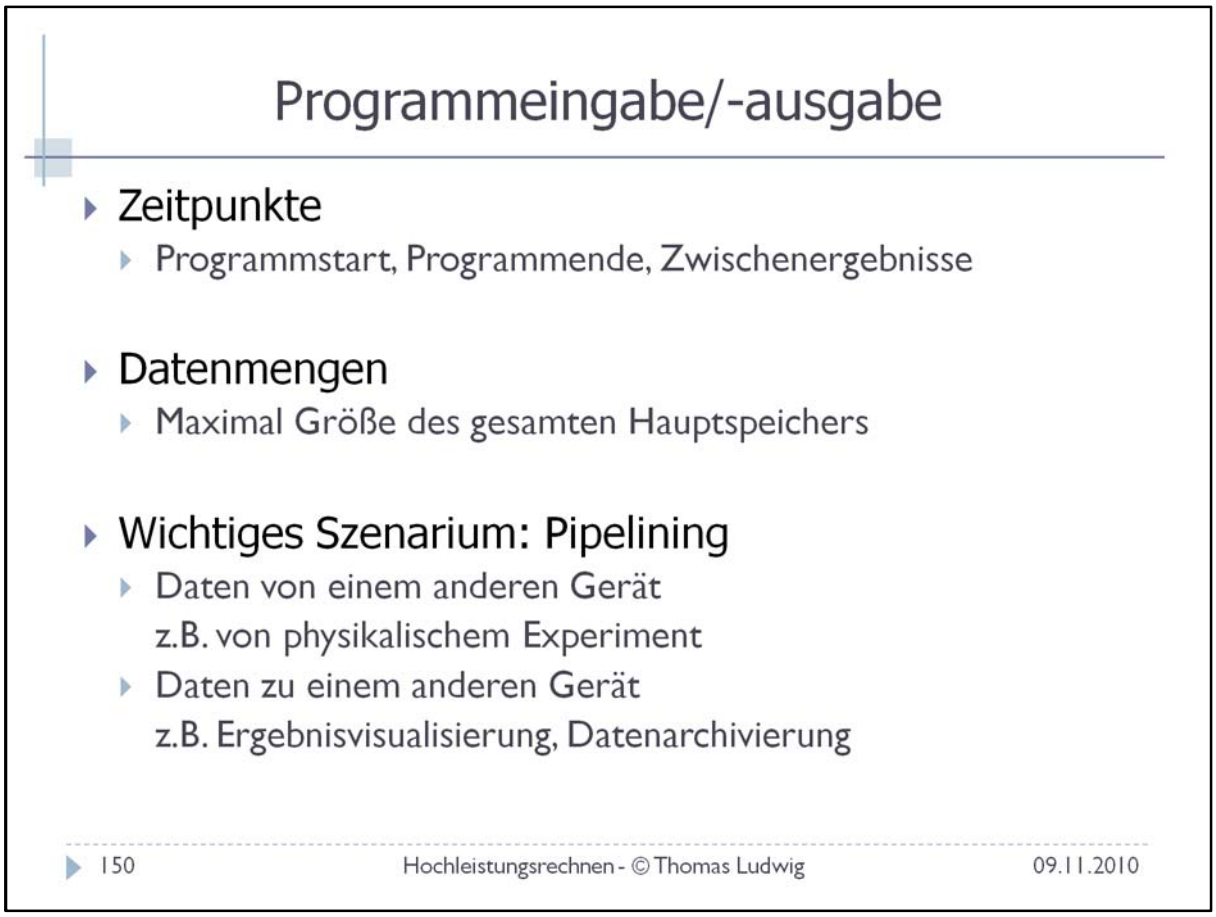
    z.B. Multimedia-Anwendungen

  Beides hier nicht weiter betrachtet
- Numerische Anwendungen
  - Regelmäßig strukturierte Daten

    z.B. Vektoren, Arrays mit großen Dimensionen

E/A-Klassen bei numerischen Anwendungen

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



Pipelining-Modus mit massiven Datenmengen ist ein wichtiger künftiger Anwendungsfall.

Sicherungspunkte

- Sichern aller wichtigen Daten
- Verwendet zur Programmfortsetzung
  - Nach Absturz oder Unterbrechung
- 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

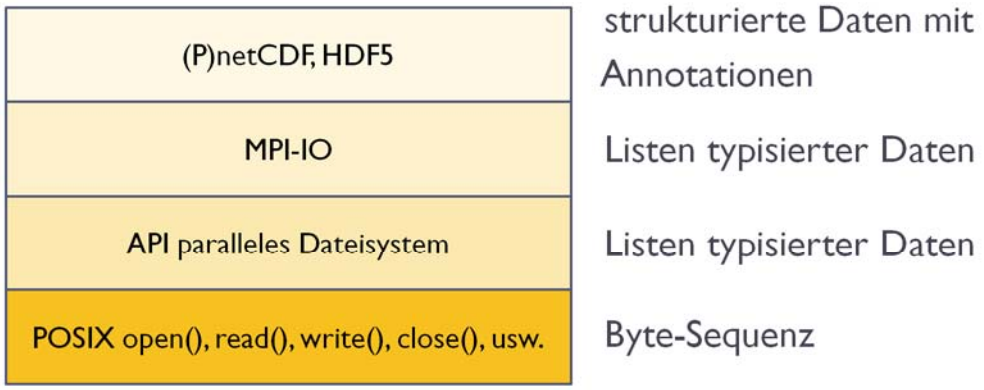
- Wichtig für Fall 1: E/A bei Programmen
- Fragestellung
  - Wann wird E/A durchgeführt?
  - Welche Mengen werden transferiert?
  - Welche Byte einer Datei werden angesprochen?
- Ausführliche Studien aus den 90ern
- Wir müssen hier lernen, was die Klimaforscher tun
- Wichtig um die Abbildung der logischen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

Entspricht der Fragestellung der Datenaufteilung bei parallelen Programmen und Rechnern mit verteiltem Speicher: Welche Daten sollen wo liegen?

Benutzungsschnittstellen

- Hierarchie von Schnittstellen

  Im Moment nur paralleles Dateisystem betrachtet



Auf den unterschiedlichen Ebenen werden Objekte verschiedener Abstraktion übertragen. Ganz unten nur einzelne Byte. In den Schichten darüber schreibt/liest man mit einem Aufruf ganze Listen typisierter Daten. Ganz oben gleich komplexe Daten samt ihrer Annotationen.

E/A im Rechnercluster

- Häufig: 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

Ein paralleles Dateisystem stellt somit für die Hintergrunddaten so etwas ähnliches dar, wie ein virtuell gemeinsamer Speicher für die Hauptspeicherdaten.

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

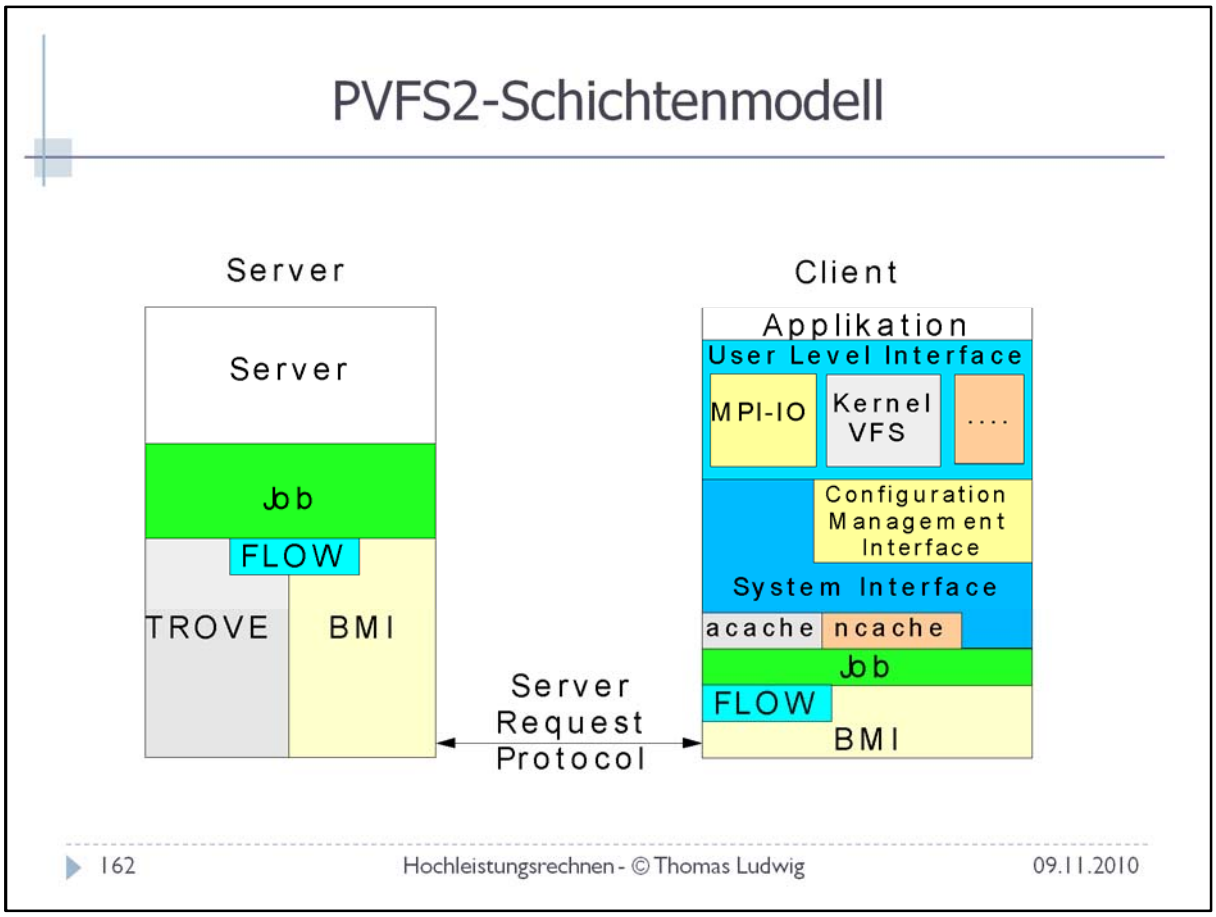
Beispiel: Parallel Virtual File System

- 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

Siehe: http://www.pvfs.org/

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



Die Schicht Job kontrolliert die Abarbeitung einzelner Anfragen in all ihren Phasen. Es sind immer mehrere Schritte in den darunterliegenden Schichten auszuführen.

Flow: Steuert den Floß von Anfragen und Antworten zwischen den tieferen Schichten Trove und BMI.

BMI (Buffered Message Interface): Netzwerk-Abstraktionsschicht. Ermöglicht die Verwendung verschiedener Geräte und Protokolle. Zur Zeit für TCP/IP (Ethernet), Myrinet´s GM und Infiniband.

Trove: Abtraktionsschicht für die persistente Datenspeicherung.

Namens-Cache (ncache) und Attribute-Cache (acache).

Arbeitsbereich Wissenschaftliches Rechnen und PVFS

Der Arbeitsbereich Wissenschaftliches Rechnen (ehemals Arbeitsgruppe „Parallele und verteilte Systeme" an der Universität Heidelberg) verwendet das „Parallel Virtual File System" 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:

- Umstieg auf GPFS
- Kombination mit Bandarchiv und Hierarchical Storage Management (HSM)

Als HSM-System verwendet das DKRZ HPSS (High Performance Storage System)

Siehe: http://www.hpss-collaboration.org/

Forschungsthema Nutzung

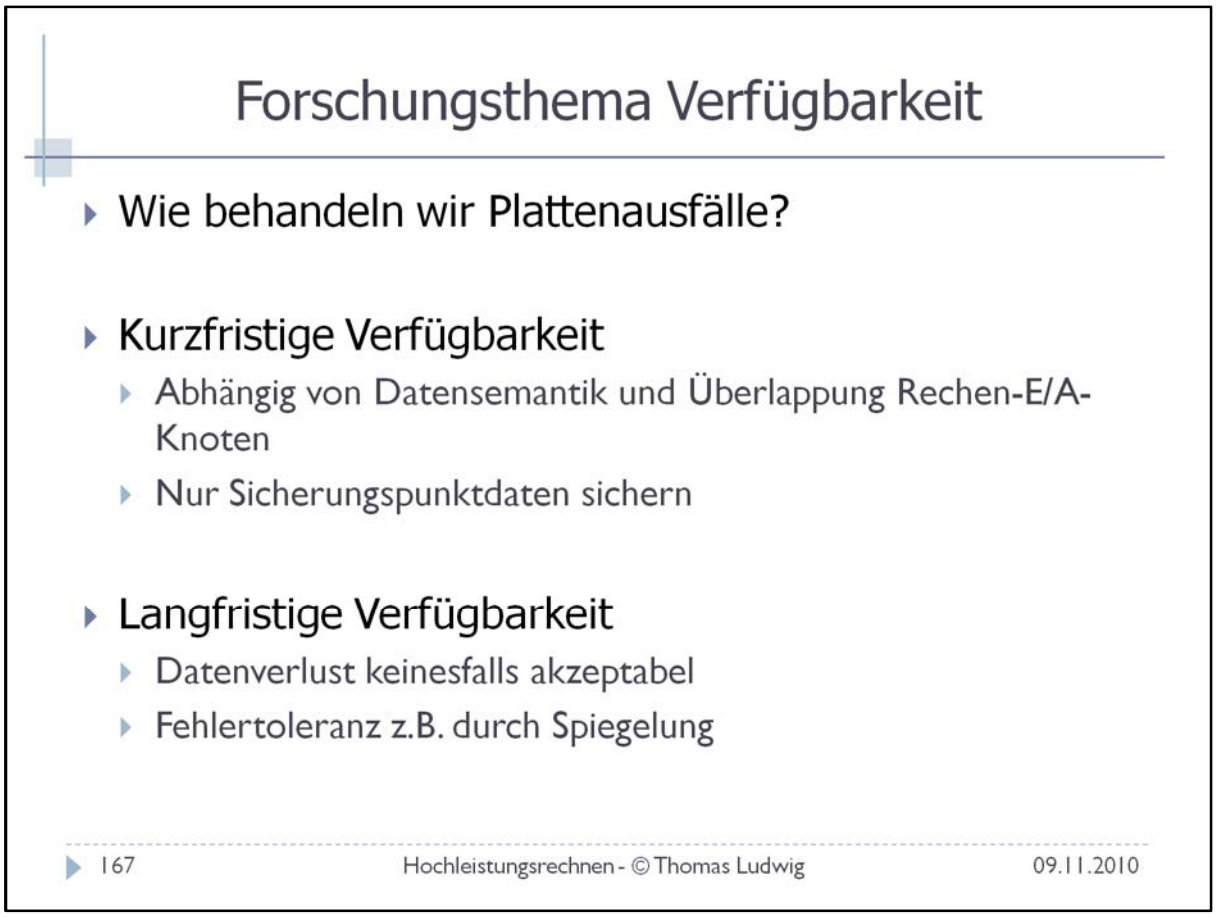
- Die Frage der Schnittstelle zum Programm ist noch offen
  - Welche Zugriffsvarianten (Semantiken)?
  - Schnittstellen auf welchem Abstraktionsniveau?
- Details im Vortrag zu MPI-IO

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

Kollektive Operationen werden bei MPI-IO besprochen.



Bei Datensemantik „Sicherungspunkt“ müssen wir Fehlertoleranz einbauen, wohingegen bei „temporäre Daten“ dies nicht notwendig ist.

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

Mitarbeit ist willkommen!

Hochleistungs-Eingabe/Ausgabe
Zusammenfassung

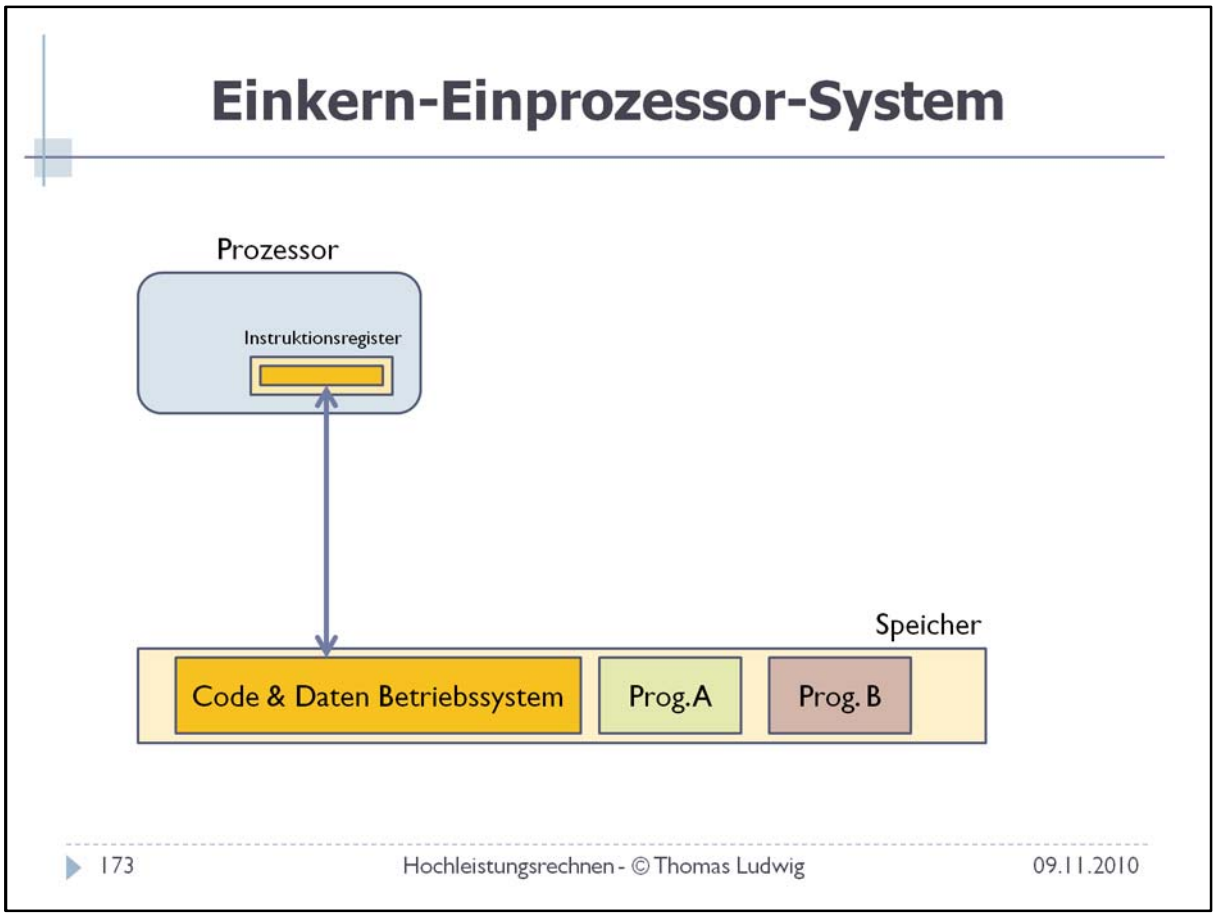
- Traditionelle Varianten der E/A jetzt ungenügend
- Parallelisierung der E/A notwendig und möglich
- Parallele E/A etwa ab Mitte der 90er entwickelt
- Uns interessieren hier nur numerische Anwendungen
- E/A gliedert sich in verschiedene Klassen auf
- Benutzerschnittstellen auf verschiedenen Ebenen
- 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

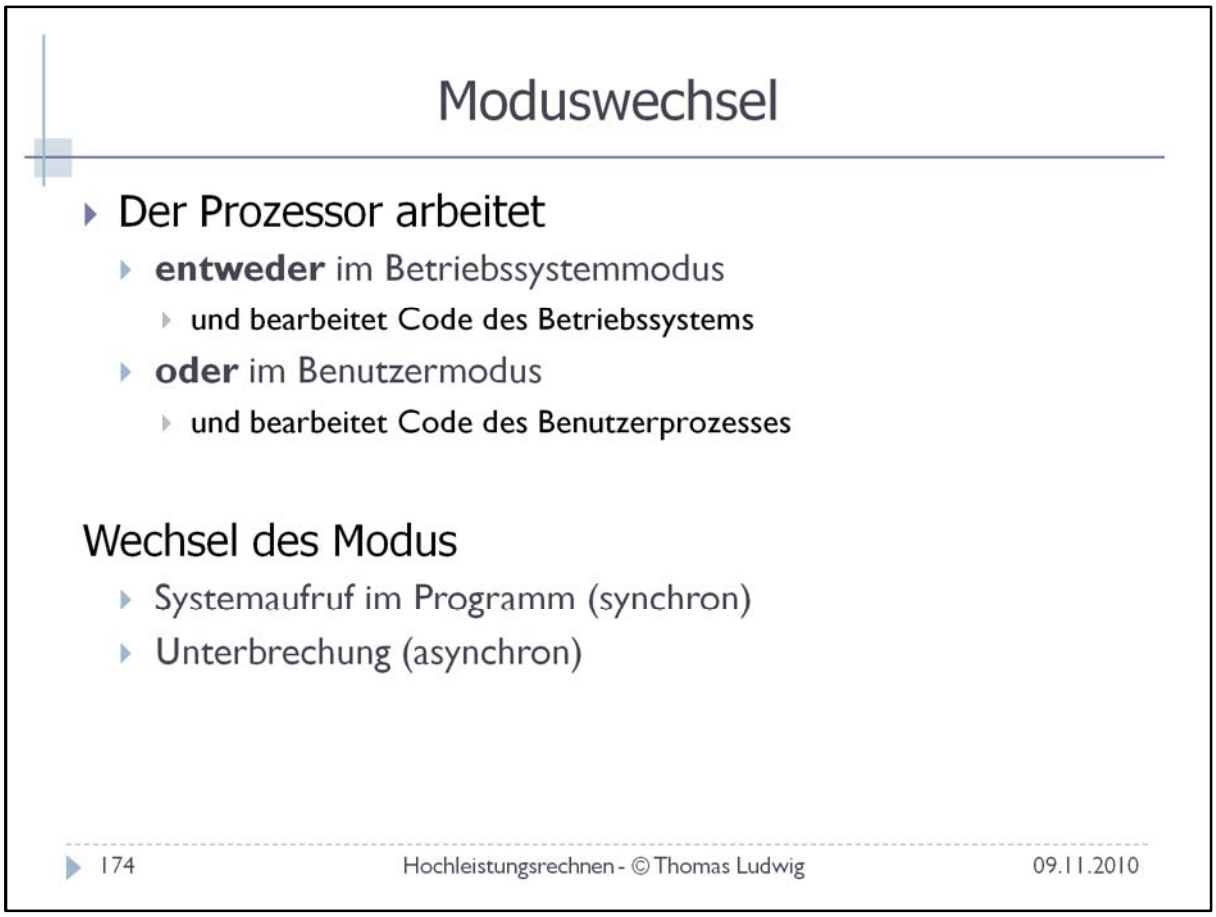
- Einprozessor-Einkern-Systeme
- Mehrprozessor/Mehrkern-Systeme
- SMP-Hardware
- SMP-Betriebssystem
- Synchronisationsmechanismen
- Eweiterte Betriebssystemfunktionalität

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?



Der Cache wird hier immer vernachlässigt, da er zunächst nur auf der Ebene der Prozessorhardware eine Rolle spielt.



Siehe:
• http://en.wikipedia.org/wiki/Usermode
• http://en.wikipedia.org/wiki/System_call
• http://en.wikipedia.org/wiki/Interrupt

Bei modernen Betriebssystemen ist die Wahrheit ein bißchen komplizierter, insbesondere, welche Adressen der Prozessor im Betriebssystemmodus bearbeitet. Der Code des Betriebssystem wird hier in den Adreßraum des Anwendungsprozeß eingeblendet und da im Systemmodus abgearbeitet. Davon wollen wir hier abstrahieren.

Moduswechsel sind teuer und sollten vermieden werden, wenn es geht. Dummerweise entsteht bei jeder E/A-Operation ein Moduswechsel.

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

175 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

Siehe: http://en.wikipedia.org/wiki/Reentrant_%28subroutine%29

Man beachte: Bibliotheken werden natürlich nur einmal in den Hauptspeicher geladen, möglicherweise aber von mehreren Prozessen verwendet. Auch hier muß der Code wiedereintrittsfähig (reentrant) sein.

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 ...

Siehe: http://en.wikipedia.org/wiki/Preemption_%28computing%29

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

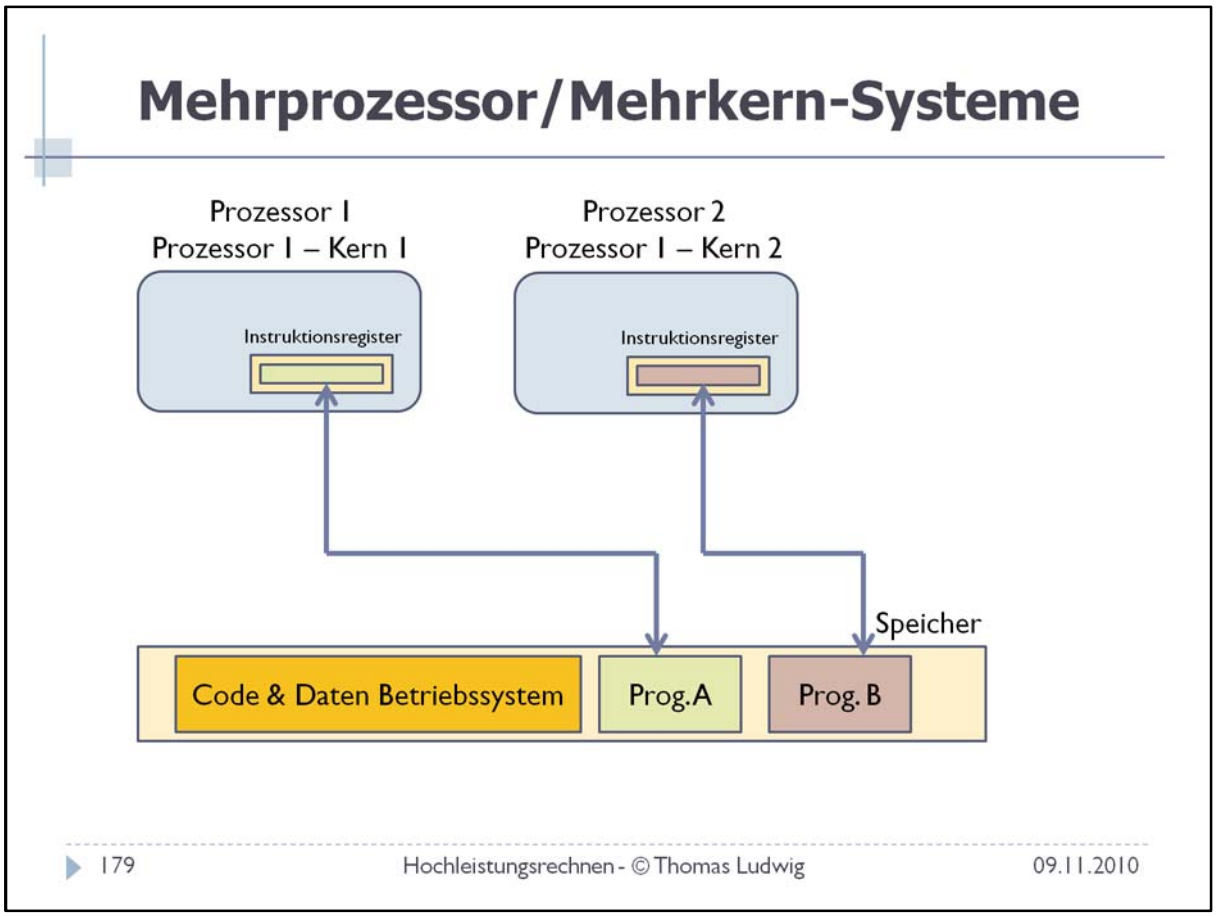
Synchronisation im traditionellen UNIX-Kern ...

Beim Einprozessorsystem

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

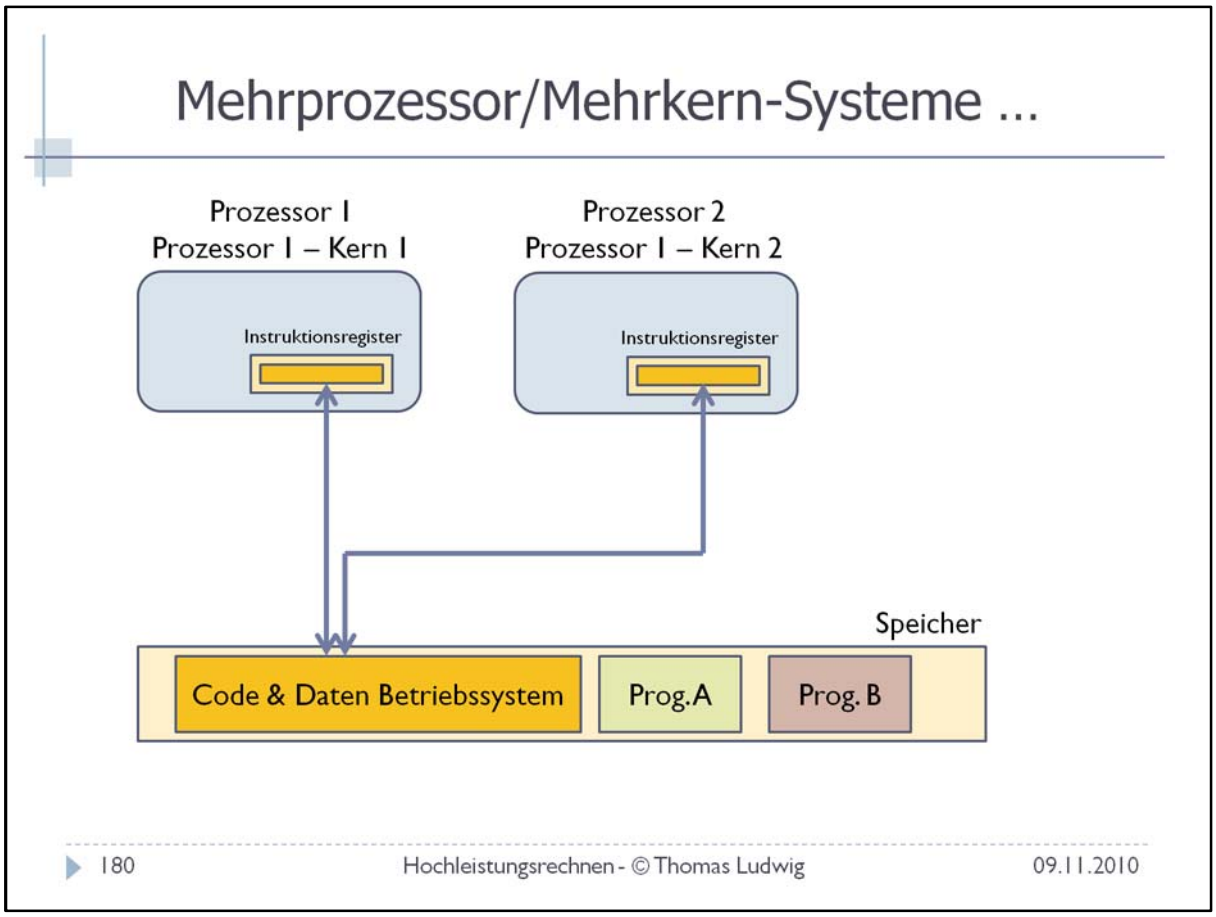
(Bei Echtzeitbetriebssystemen ist die Ununterbrechbarkeit des Kern nicht mehr gegeben – damit neue Probleme)

Siehe: http://en.wikipedia.org/wiki/Realtime_operating_system



Siehe: http://en.wikipedia.org/wiki/Multi-core_processor

Unkritischer Fall: Prozessoren bearbeiten Code von verschiedenen Programmen/Prozessen.



Kritischer Fall: Beide Prozessoren arbeiten gleichzeitig im Code des Betriebssystem.

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**

Siehe: http://en.wikipedia.org/wiki/Thread_%28computer_science%29

„Nebenläufig“ (concurrent) bedeutet, daß zwei Aktionsstränge gleichzeitig ablaufen, diese aber nicht notwendigerweise miteinander zu tun haben. Von „parallel“ spricht man, wenn die Abläufe zu einem Programm gehören und logisch miteinander verknüpft sind, indem z.B. Synchronisationen erfolgen.

SMP-Hardware

SMP Symmetric Multiprocessing

- Variante des Betriebssytems, 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

Siehe:
- http://en.wikipedia.org/wiki /Symmetric_multiprocessing
- http://en.wikipedia.org/wiki/MultiProcessor_Specification
- http://en.wikipedia.org/wiki/Intel_APIC_Architecture

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

185 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

In einem **verteilten** Betriebssystem werden Funktionen des BS auf **unterschiedlichen** Rechnern abgewickelt.

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

Siehe: http://en.wikipedia.org/wiki/Giant_lock

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 sie 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

Siehe: http://en.wikipedia.org/wiki/Thundering_herd_problem

Thundering-herd-problem: Beim Freiwerden einer Ressource werden viele Threads gleichzeitig deblockiert, nur um alle bis auf einen sofort wieder blockiert zu werden.

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

Siehe: http://en.wikipedia.org/wiki/Atomic_operation

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

Siehe: http://en.wikipedia.org/wiki/Semaphore_%28programming%29

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

Siehe: http://en.wikipedia.org/wiki/Spinlock

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 (?)

Siehe: http://en.wikipedia.org/wiki/Linux_kernel#Architecture

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 Prozessor fortgesetzt werden, auf dem er zuletzt lief

  Grund: Gültiger Inhalt im L2-Cache

- Gang-Scheduling

  Alle Threads eines Prozesses sollten zusammen zugeteilt werden

  Grund: Verzögerungen an gemeinsam genutzten Sperren vermeiden

Siehe:
- http://en.wikipedia.org/wiki/Processor_affinity
- http://en.wikipedia.org/wiki/Gang_scheduling

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 Architekturen sind diese Paradigmen jeweils optimal 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

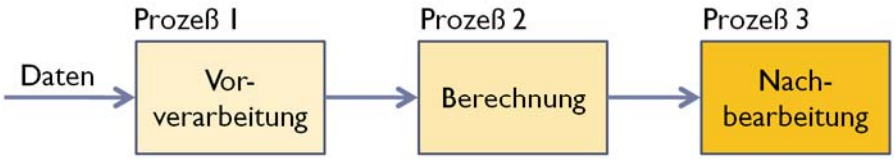
- Zusätzlich zur sequentiellen Software
  - Aufteilung des Programms in kleine Teile (*partitioning*)
  - Hinzufügen von Koordination und Kommunikation
  - Abbildung der Teile auf die Komponenten des Computers (*mapping*)
- Probleme
  - Fehlersuche (neue Fehlertypen)
  - Leistungsanalyse (Programmbeeinflussung)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



Siehe: http://en.wikipedia.org/wiki/Task_parallelism

Paradigmen der Parallelisierung...

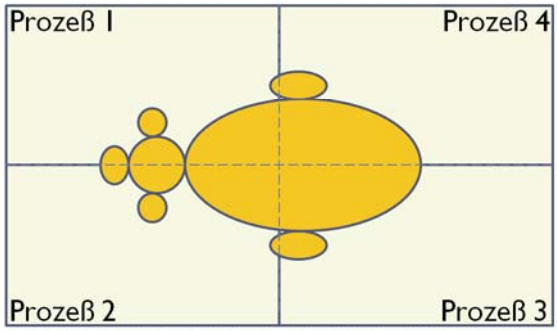
- **Vorteile** der Code-Aufteilung
  - Manchmal sehr einfach zu entwerfen
  - Passende Algorithmen existieren (z.B. FFT)
- **Nachteil** der Code-Aufteilung
  - Mehrere Quellcode-Dateien nötig
  - Schwierig an Zielmaschine anpaßbar
  - Komplexe Kommunikationsschemata
  - Schwierige Fehlersuche
  - Schwieriger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



Siehe: http://en.wikipedia.org/wiki/Data_parallelism

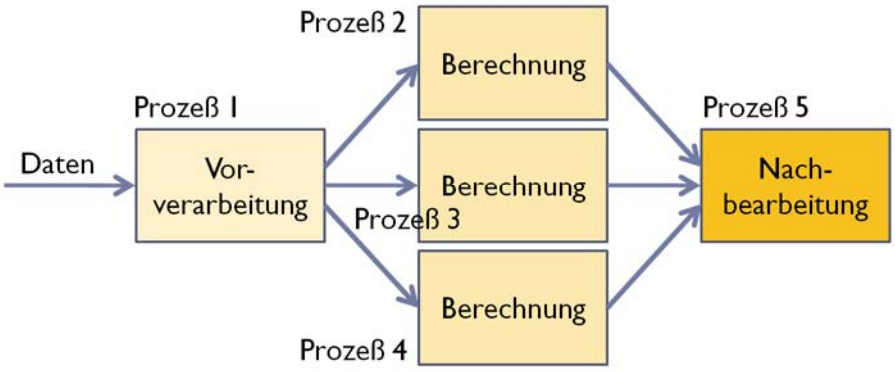
Paradigmen der Parallelisierung...

- **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

Siehe: http://en.wikipedia.org/wiki/Automatic_parallelization

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 übergibt 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

Siehe: http://en.wikipedia.org/wiki/Parallel_programming_model

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

Siehe:
- http://en.wikipedia.org/wiki/Grand_Challenge
- http://en.wikipedia.org/wiki/Theorem_prover

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

Siehe: http://en.wikipedia.org/wiki/Computational_fluid_dynamics

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

Siehe: http://en.wikipedia.org/wiki/Paradigm

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)

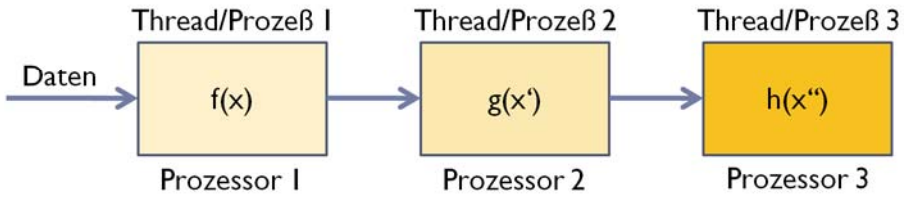
- Drei Funktionen f(), g() und h()
- Wende Funktionen auf eine Menge von Werten an und berechne Ergebnis h(g(f(x)))

- Wir betrachten vier Fälle:
  - Code-Aufteilung / Datenaufteilung
  - Gemeinsamer Speicher / verteilter Speicher

Beispiel 1: Numerik (2/11)

Code-Aufteilung

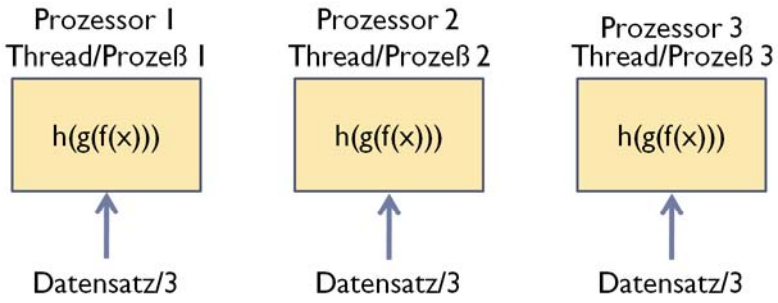
- Verteile drei Funktionen auf drei Knoten
- Arbeitet im Makro-Pipelining-Modus
- Wertemenge ist Eingabedatensatz
- Nur drei Prozessoren sinnvoll nutzbar



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
  - Potentieller Zugriffskonflikt auf Binärcode der Threads

Beispiel 1: Numerik (11/11)

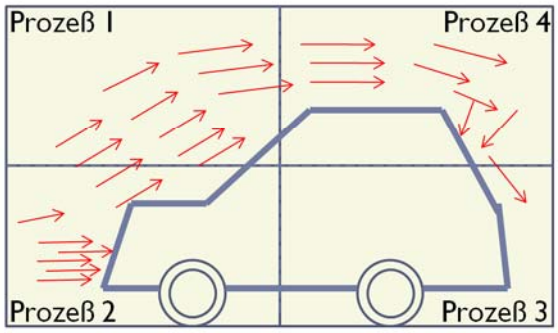
Datenaufteilung mit verteiltem Speicher

- 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)

Simulation eines Windkanals

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



Beim molekulardynamischen Verfahren werden die Flugbahnen und Kollisionen der Teilchen exakt berechnet. Dies ist extrem aufwendig.
Bei den Monte Carlo-Verfahren werden Flugbahnen und Kollisionen nicht exakt berechnet sondern quasi ausgewürfelt (mittels Zufallszahlen). Für bestimmte Konfigurationen des Experiments kommen hier nahezu exakte Ergebnisse heraus, wobei aber die Rechenzeiten dramatisch kürzer sind.

Für den makroskopischen Ansatz kommen sogenannte Gitterfahren zum Einsatz. Hier werden feinmaschige Gitter über das Gebiet gelegt und die Werte an den Kreuzungspunkten der Gitterlinien oder für die Gitterzellen berechnet.

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)

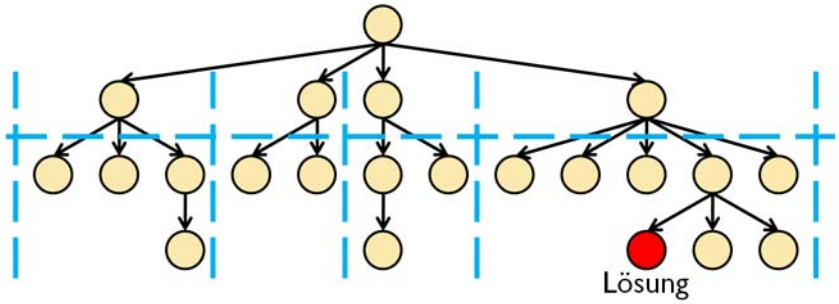
Mikroskopischer Ansatz:

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)

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



Siehe: http://en.wikipedia.org/wiki/Search_tree

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 verteiltem 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 Ende-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

Literatur:

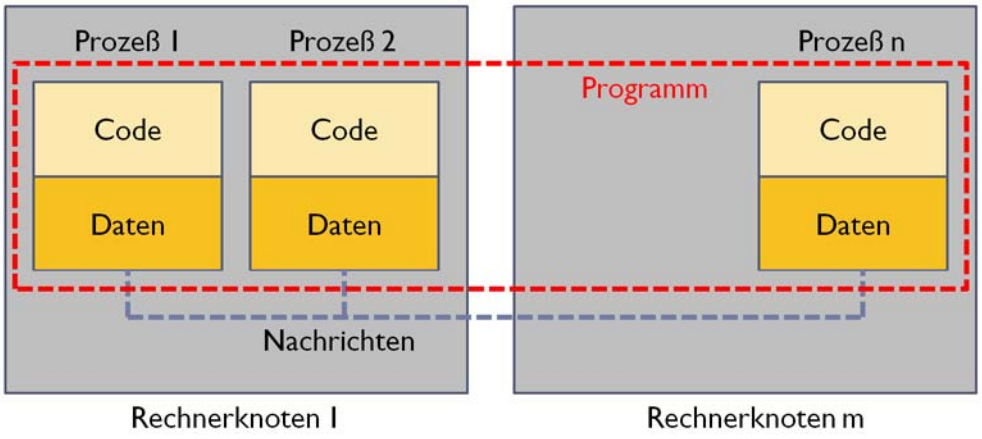
• Mark Snir et al.: MPI – The Complete Reference. Volume 1, The MPI Core. Second Edition. MIT-Press, 1998.
• William Gropp et al.: Using MPI – Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT-Press, 1999.

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



243 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

Nochmal zum Begriff des Rechnerknotens:
• Früher war ein Rechnerknoten ein einzelner Rechner, meist so eine Art PC, von denen man mehrere in einem Cluster zusammengebaut hat. D.h.: ein Knoten = ein Prozessor (=ein Core). Darauf bringt man einen Prozeß der Anwendung zum Ablauf.
• Heute ist ein Knoten z.B. am System „Blizzard“ des DKRZ ein Einschub mit 16 Prozessoren zu je 2 Cores, also ein 32-Prozessor-System.

Die Zuweisung von Jobs zum Rechner erfolgt durch die Angabe der Anzahl von Knoten, die man haben möchte. Es werden den Benutzern immer ganzzahlige Vielfache dieser Knoten zugewiesen.

Auf einem Knoten, der dann gemeinsamen Speicher hat, kann ein Prozeß nochmal in Threads unterteilt werden, die dann wiederum Code-Aufteilung und/oder Datenaufteilung folgen.

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>,...);
```

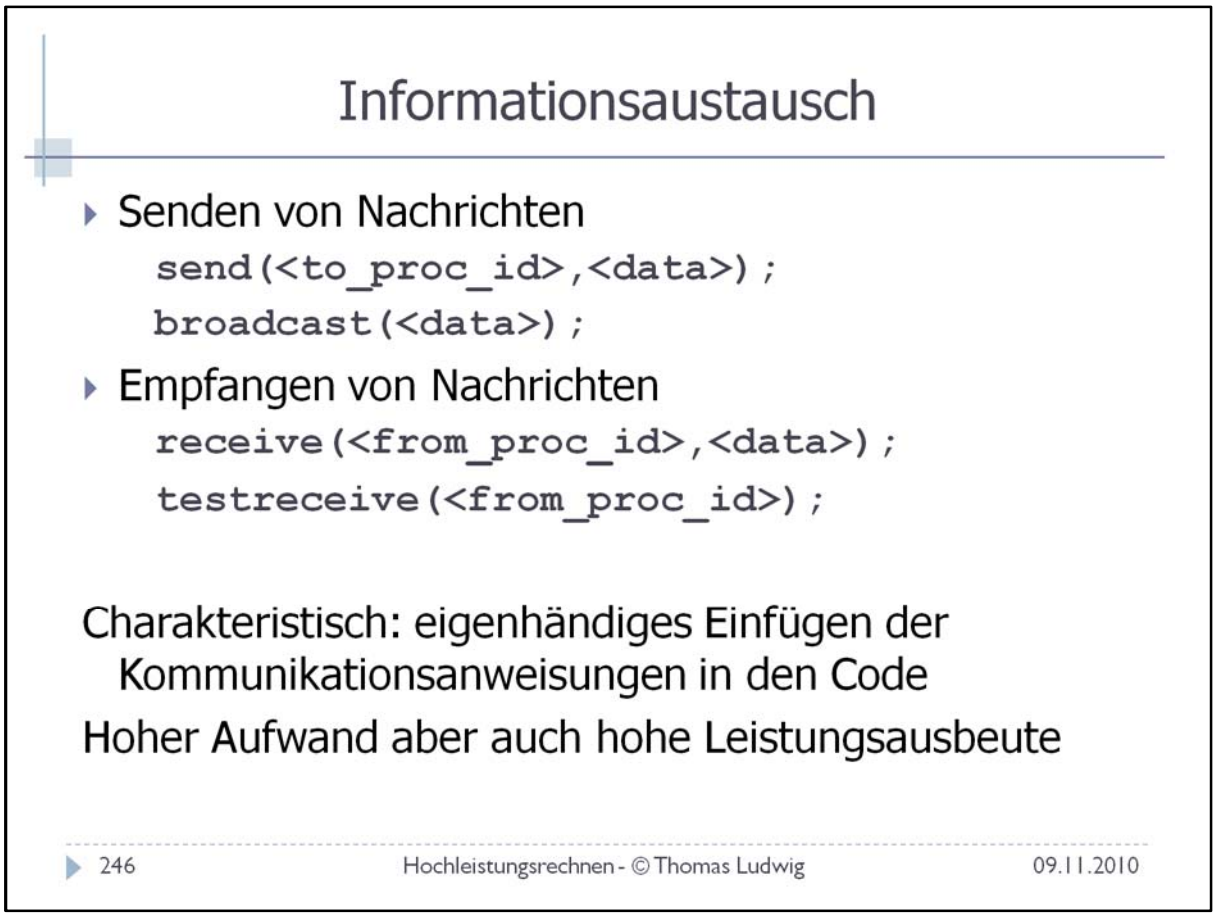
  - Ist ähnlich wie bei einer Thread-Erzeugung

- Wenn nur ein Programmcode existiert

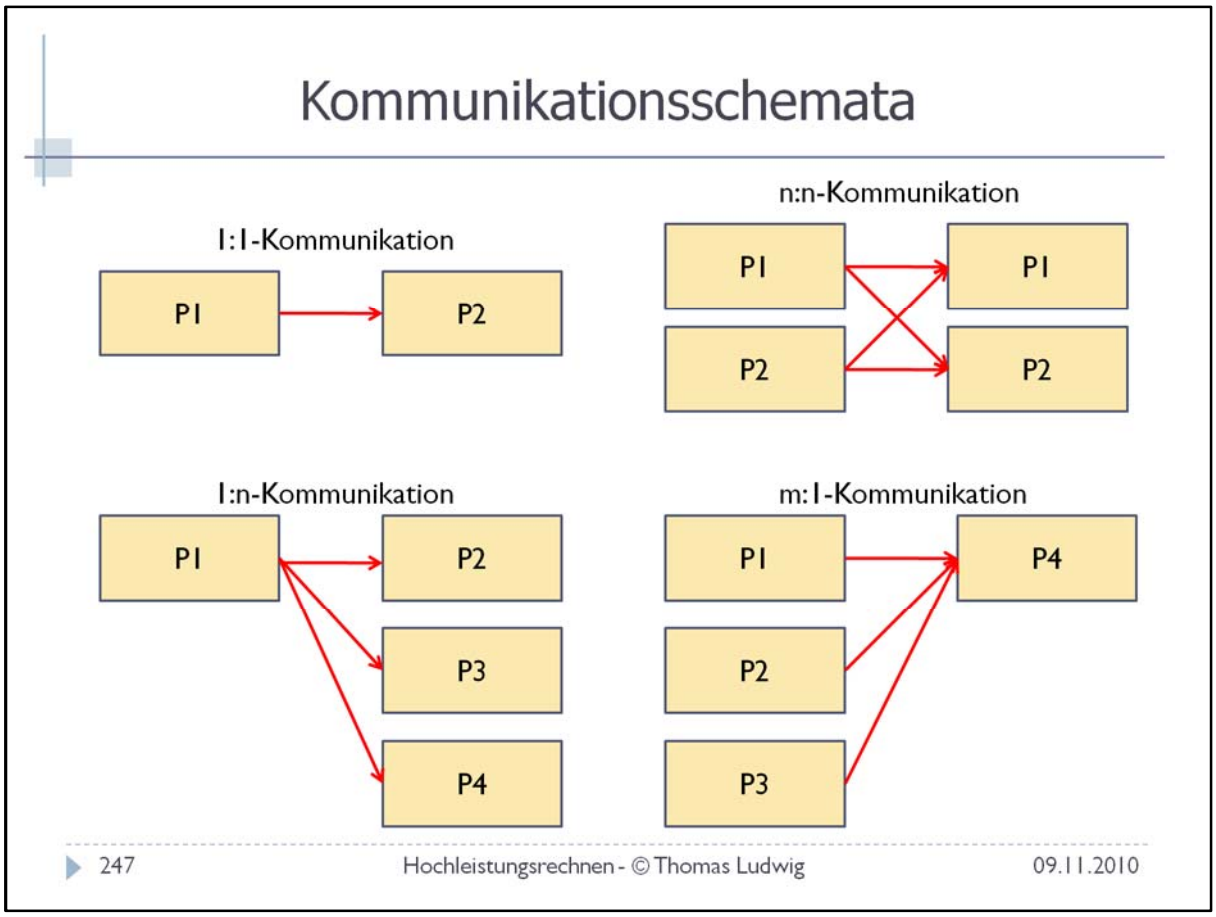
```
if (myid()==0)
then /* I´m the first */
  spawn(...); /* if others do not exist */
  send(init_data);
else /* I was spawned */
  receive(init_data);
fi
```

Nicht notwendigerweise nur ein Prozeß pro Prozessor

Bei z.B. MPI werden die beteiligten n Prozesse von 0 bis n-1 durchnumeriert und das definiert quasi ihren Namen. Prozeß 0 ist dann ein natürlicher Kandidat für Organisationsaufgaben. Bei anderen Bibliotheken können die Benennungen aber unterschiedlich funktionieren!



Die Programmierung des Nachrichtenaustausch wird auch als die Maschinenprogrammierung des parallelen Rechnens bezeichnet, weil sie auf einer sehr niedrigen Abstraktionsebene ansetzt.



1:1-Kommunikation: Ein Sender-Prozeß sendet an genau einen Empfänger-Prozeß. Der Sender kennt die Adresse vom Empfänger, der Empfänger die vom Sender.

1:n-Kommunikation: Ein Sender-Prozeß sendet an viele Empfänger-Prozesse mit einem einizigen Aufruf. Geht die Nachricht an alle anderen, so spricht man von Broadcast, geht sie an eine echte Teilmenge, so spricht man von Multicast.

m:1-Kommunikation: Insgesamt m Sender-Prozesse senden an ein und denselben Empfängerprozeß. Man unterscheidet syntaktisch Fälle, bei denen der Empfänger die Sender kennt und solche, wo das nicht der Fall ist. Ersterer wird verwandt, wenn z.B. Ergebnisse korrekt zusammengefügt werden müssen, letzterer, wenn man z.B. Teilergebnisse nur aufaddiert. Der zweite Fall kann meist mit besserer Effizienz implementiert werden.

n:n-Kommunikation wird verwendet, wenn eine Menge von Prozessen untereinander Daten austauschen, so daß jeder an Sende- und Empfangsoperationen beteiligt ist.

Alle komplexen Kommunikationsschemata können immer durch eine Menge 1:1-Kommunikationen dargestellt werden. Manchmal wird das auch so implementiert, das ist dann schnell und ineffizient. Vielfach wird aber gerade hier sehr viel Aufwand in interne Optimierungen der Bibliotheken gelegt.

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

Siehe: http://en.wikipedia.org/wiki/Parallel_Virtual_Machine

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 unterscchiedlich
- Probleme: Standardkonformität der Implementierungen

Siehe:
- http://www.mpi-forum.org/
- http://en.wikipedia.org/wiki/Message_Passing_Interface

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
- Kommunikationskontexte
- 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)
   IN buf       Adresse des Sendepuffers
   IN count     Anzahl der Elemente im Puffer
   IN datatype  Datentyp des Elements
   IN dest      Rangangabe des Ziels
   IN tag       Nachrichtenkennung
   IN comm      Kommunikator(Gruppe,Kontext)
```

Datentypen: int, long int, float, char, ...
Nachrichten bestehen aus Inhalt und Umschlag

Der Datentyp kann ein elementarer sein, z.B. Byte, Integer, Float, oder auch ein zusammengesetzter komplexer wie z.B. ein Feld komplexer Zahlen mit zehn Zeichenketten hintendran.

Punkt-zu-Punkt-Kommunikation...

Empfangen

```
MPI_RECV(buf,count,datatype,source,tag,
        comm,status)
 OUT buf     Adresse des Empfangspuffers
 IN count    Anz. der Elemente im Puffer
 IN datatype  Datentyp des Elements
 IN source     Rangangabe der Quelle
 IN tag      Nachrichtenkennung
 IN comm     Kommunika. (Gruppe,Kontext)
OUT status    Ergebnis des Empfangens
```

Punkt-zu-Punkt-Kommunikation...

Empfangen...

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

Aufgabe: In MPI gibt es zum Senden kein `MPI_ANY_DEST`. Überlegen Sie, wie die Semantik hiervon sein sollte, wenn es das gäbe. Wofür könnte man das gebrauchen? Wie könnte man es trotzdem implementieren?

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

```
MPI_ISEND(...)              immediate send
MPI_IRECV(...)              immediate receive
MPI_TEST(request,flag,status)
                            nichtblockierend
MPI_WAIT(request)           blockierend
MPI_CANCEL(request)
```

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;
}
```

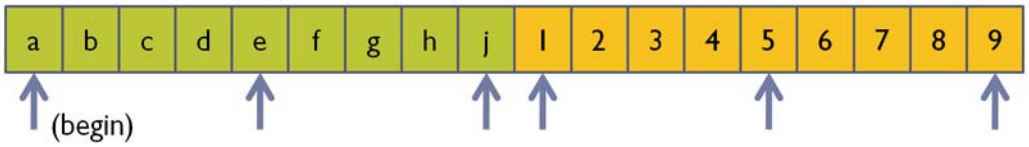

`MPI_Init()` und `MPI_Finalize()` rahmen den parallelen Teil eines Programmes eines. Davor und dahinter darf noch Code stehen.

Abgeleitete Datentypen

- Verwendungszweck
  - Nachrichten mit gemischten Datentypen
  - Nachrichten mit nichtzusammenhängenden Bereichen
- Ein-/Auspacken 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)
```

264 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

MPI_TYPE_VECTOR hängt n Blöcke zusammen (hier: 3), die die Blocklänge l (hier: 1) haben und im Abstand von s Elementen des Ausgangsdatentyps (stride, Versatz) (hier: 4) liegen. Ausgangsdatentyp ist hier MPI_COMPLEX, der neue Datentyp ist hier diag.

MPI_TYPE_CREATE_HVECTOR mißt den Versatz in Byte.

Das Konzept stützt sich sehr auf die Kenntnis der speicherinternen Organisation der einzelnen Datentypen.

Kollektive Kommunikationen

Kollektive Kommunikationen werden immer von allen Mitgliedern einer Gruppe durchgeführt

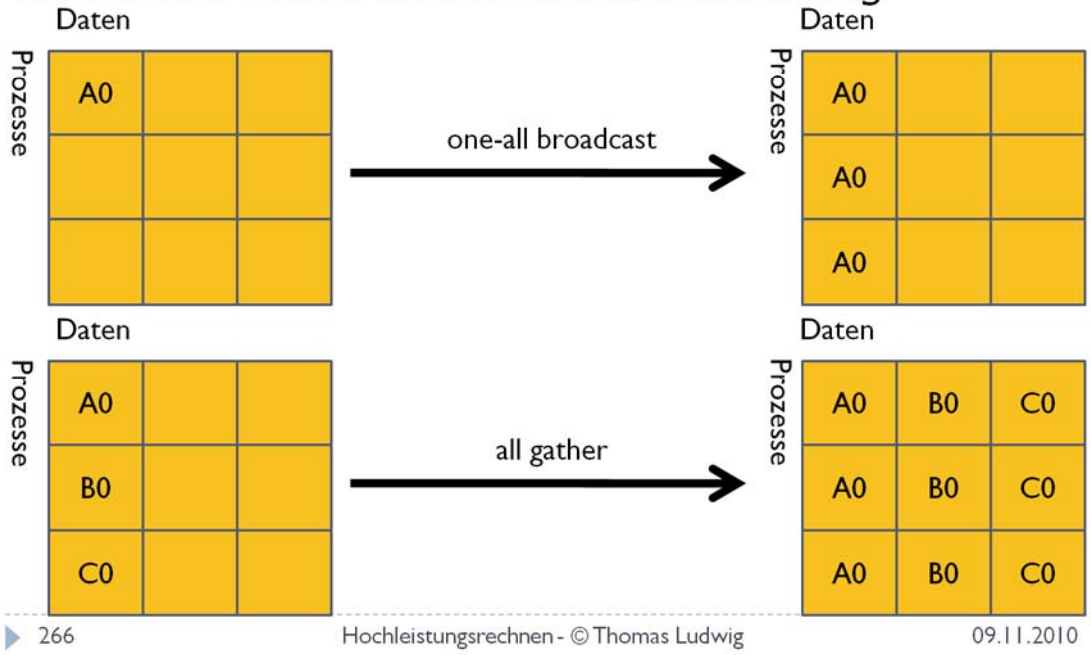
- Broadcast von einem an alle
- Barrierensynchronisation
- Daten einsammeln / verteilen
- Globale Berechnung von Funktionen

Möglicherweise durch spezielle Hardware unterstützt

Spielraum für Implementierungsoptimierungen

Kollektive Kommunikationen...

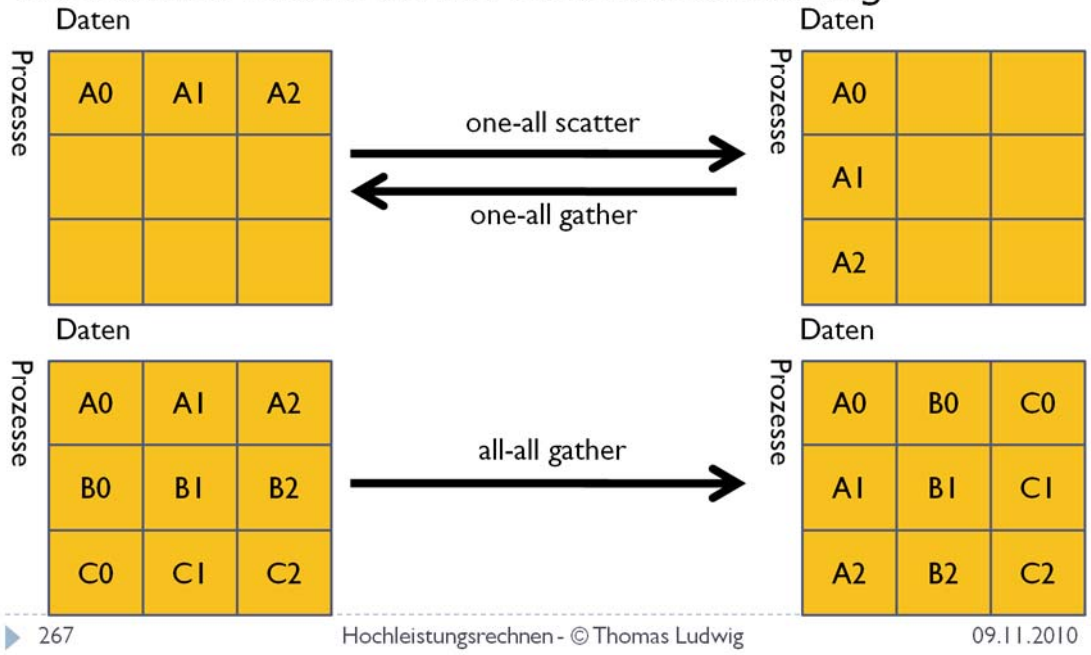
Kollektive Funktionen zur Datenverschiebung



266 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

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)

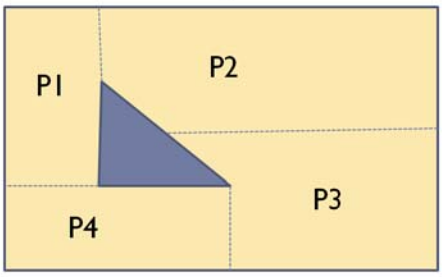
Die Operation op wird immer als assoziativ angenommen, d.h. a op (b op c) = (a op b) op c. Außerdem sind alle vordefinierten Funktionen auch kommutativ, d.h. a op b = b op a . Z.B. ist aber eine Gleitkommaaddition im Rechner nicht absolut kommutativ und assoziativ wegen der begrenzten Rechengenauigkeit.

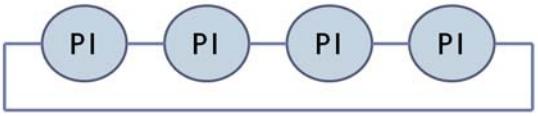
Gruppen, Kontexte, Kommunikatoren

- 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

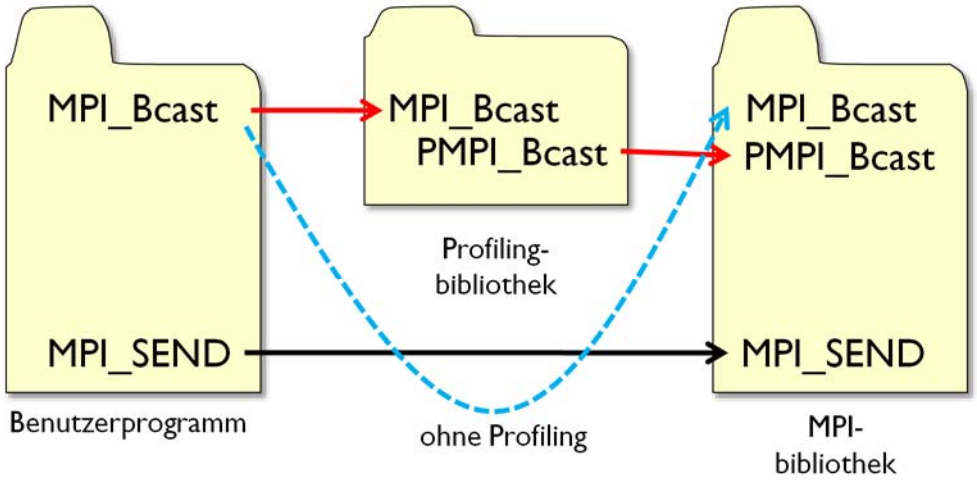
```
int MPI_Bcast(...)
  { int result;
    write_log_entry(...);
    start_timer();
    result=PMPI_Bcast(...);
    stop_timer(); write_log_entry(...);
    return result;
  }
```

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

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

Siehe:
• http://www-unix.mcs.anl.gov/mpi
• http://www-unix.mcs.anl.gov/mpi/mpich

Bücher:
• W. Gropp, R. Thakur, E. Lusk: Using MPI-2 – Advanced Features of the Message-Passing Interface. MIT-Press, 1999.
• W. Gropp, B. Nitzberg, E. Lusk: MPI – The Complete Reference Volume 2. MIT-Press, 1998.

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 Nachrichten-austausch 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?

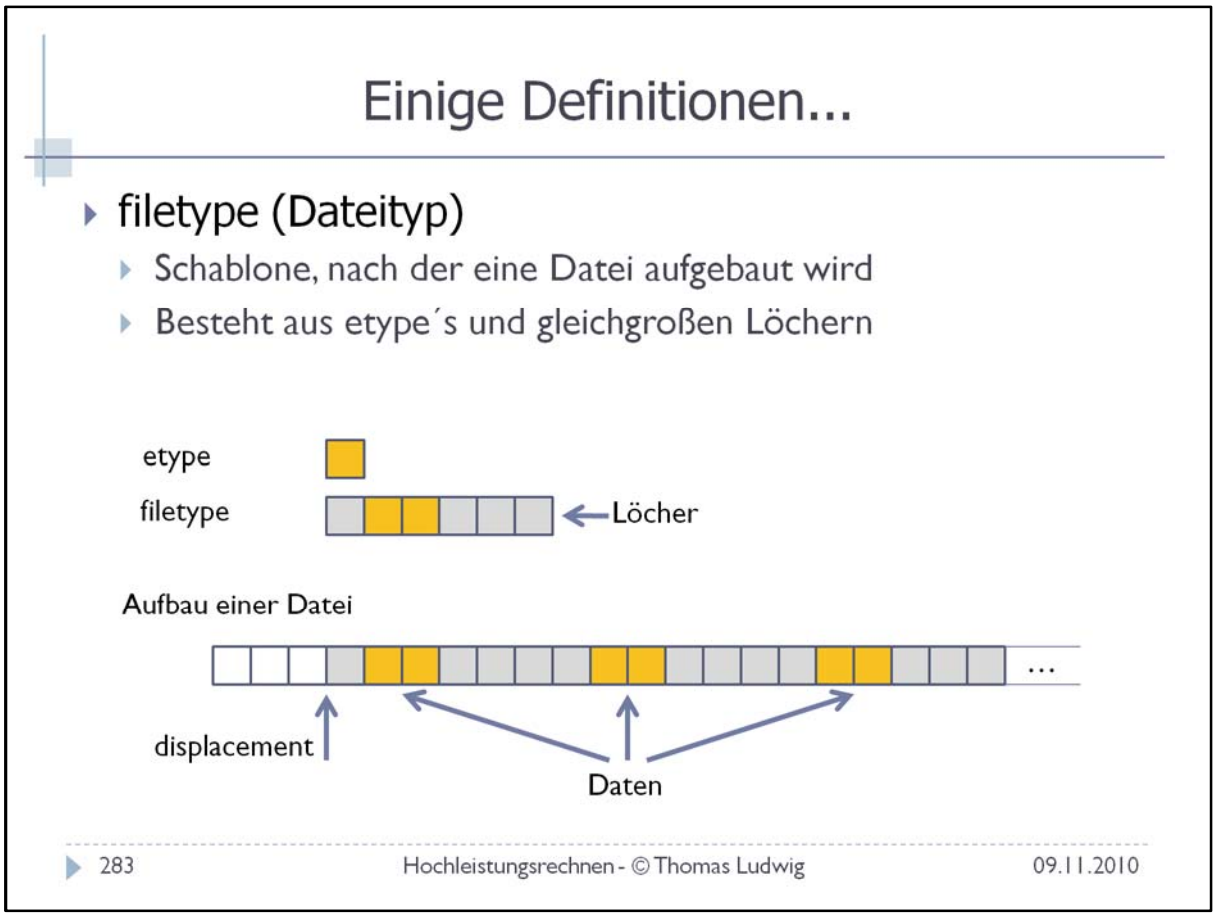
- Leistungsgewinn
  - Z.B. durch kollektive Aufrufe
  - Z.B. durch asynchrone E/A
- Einfacherer Zugriff durch Problemanpassung
  - Z.B. abgeleitete Datentypen bei irregulären Daten
  - Dadurch auch Portabilität in heterogenen Umgebungen

Konzepte der MPI-I/O

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

Einige Definitionen

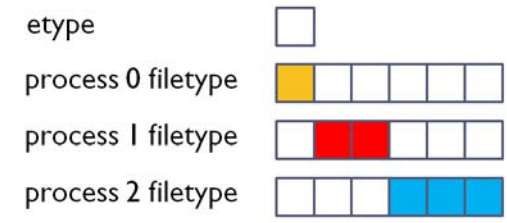
- file (Datei)
  - Eine geordnete Sammlung typisierter Daten
  - Zugriff erfolgt wahlfrei oder sequentiell
  - Kollektives Öffnen durch eine Gruppe von Prozessen
- displacement (Versatz)
  - Eine absolute Byte-Position relativ zum Dateianfang, an der eine Dateisicht beginnt
- etype (elementary datatype)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



Der Trick ist natürlich, daß jeder Prozeß nur seine Daten sieht, dabei dann aber das, was alle in der Summe sehen, die gesamte Datei ergibt!

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)
```


Bei MPI_File_seek bestimmt whence, wie die Position aus dem Offset berechnet wird: relativ zum Dateianfang, zur aktuellen Position oder zum Dateiende.

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 of 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

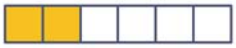
  `MPI_TYPE_CREATE_RESIZED`

Nichtzusammenhängende Zugriffe: Beispiel

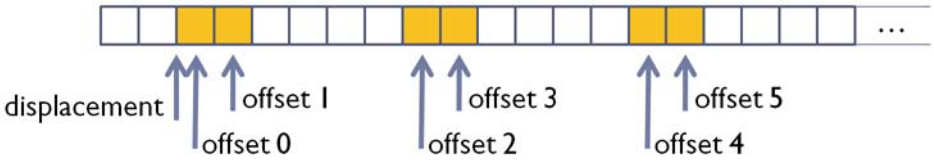
```
/* 2 MPI_INT zusammenhängend als derived data type /*
MPI_Type_contiguous(2,MPI_INT,&contig);

/* 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 wird gemäß dem Rang der Prozesse serialisiert

  `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 durchzureichen
- 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

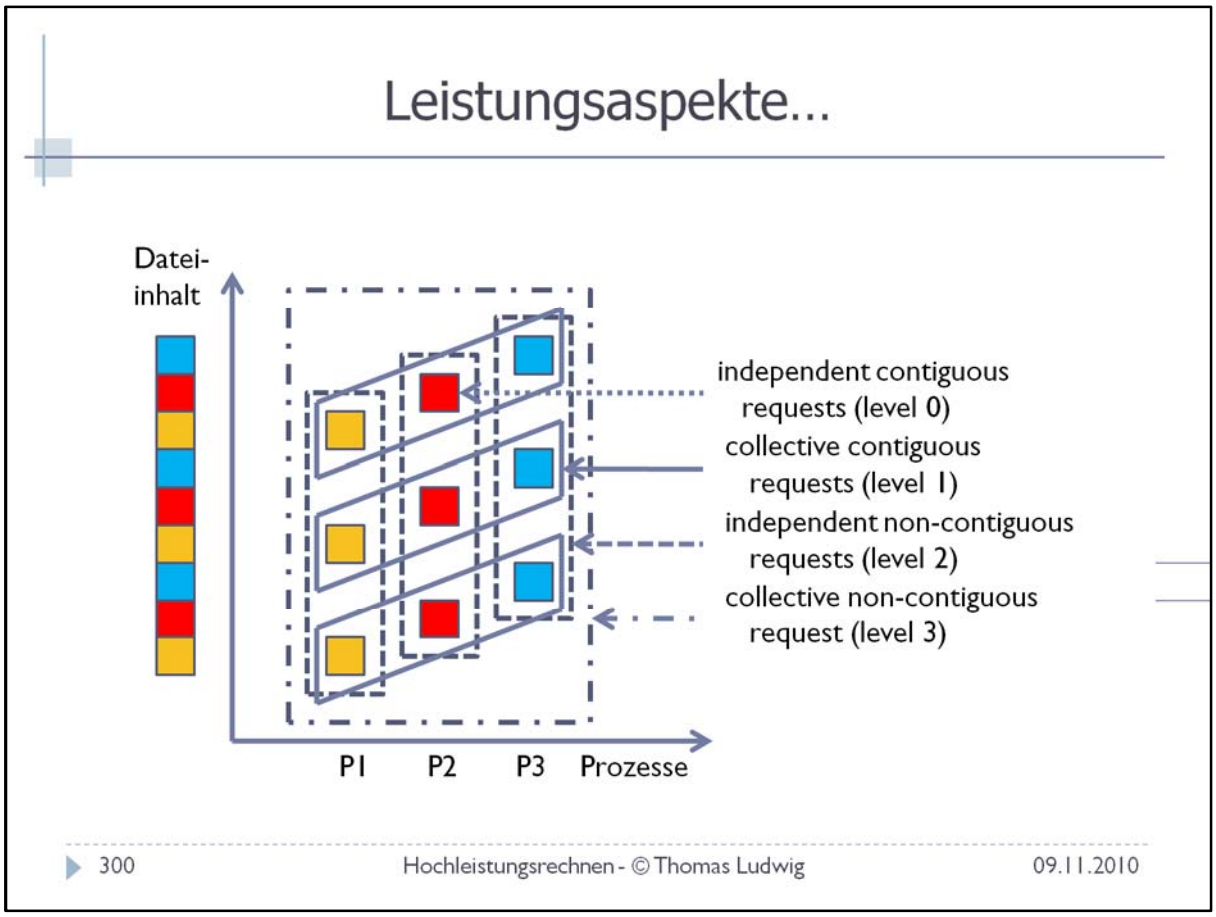
- Dateien werden als Folge von Bytes gesehen

  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 geeignetsten E/A-Methode bestimmt die erzielbare E/A-Bandbreite
  - Zusammenhängend / nichtzusammenhängend
  - Kollektiv / nichtkollektiv
- Beispiel
  - Datei einer 3x3-Matrix komplexer Datentypen



Je nach Level nimmt die Bibliothek verschiedene Optimierungen vor. Z.B. wird bei level 2 und lesenden Zugriffen auf nichtzusammenhängende Datenmenge ggf. eine größere Datenmenge in einem Zugriff gelesen und die nicht benötigten Teilabschnitte werden weggeworfen. Die anderen Optimierungen sind zu komplex, als daß wir sie hier diskutieren könnten.

Wichtig ist: je mehr Information wir der Bibliothek geben können (in Form von Kenntnis über den gesamten Ablauf aller Zugriffe), desto mehr kann sie evtl. optimieren. Sie muß es allerdings nicht.

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 im Betriebssystem
- Typen von Threads
- Einsatzbereiche Threads
- Die Pthreads-Schnittstelle

Siehe: http://en.wikipedia.org/wiki/Thread_%28computer_science%29

Programmierung mit Threads
Die zehn wichtigsten Fragen

- Worin unterscheiden sich Threads und Prozesse?
- Was sind Threads auf Benutzerebene und auf Systemebene?
- Was ist eine hybride Thread-Realisierung?
- Warum sind Threads ein Thema beim Hochleistungsrechnen?
- Wie funktioniert Thread-Scheduling?
- Was ist ein Prioritätswechselprotokoll?
- Wann programmieren wir mit Threads?
- Welche Grundoperationen bietet die Pthreads-Schnittstelle?
- Wie funktioniert ein Mutex?
- Wie arbeitet man mit Bedingungsvariablen?

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...

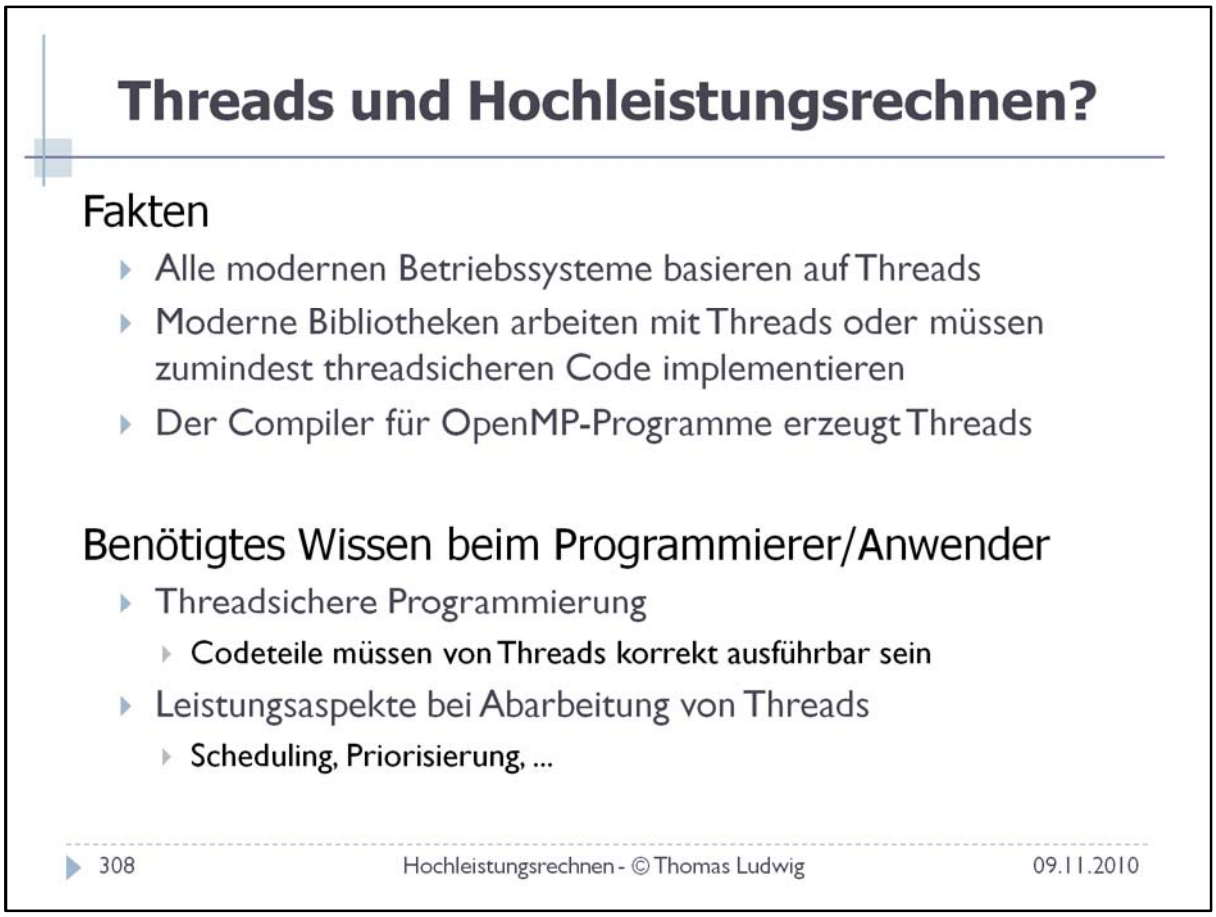
Eigenschaften von Threads

- 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)
  - Kellergröße (feste maximale Größe)
- 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
- Strukturierung von Programmen



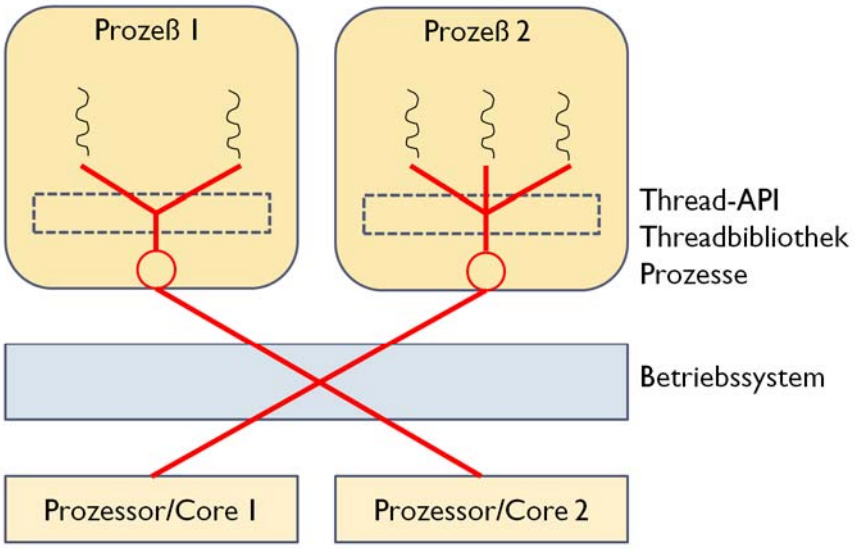
Siehe: http://en.wikipedia.org/wiki/Threadsafe

„Thread safety is a key challenge in multi-threaded programming. It was once only a concern of operating system programmers but since the late 1990s has become a commonplace issue. In a multi-threaded program, several threads execute simultaneously in a shared address space. Every thread has access to virtually all the memory of every other thread. Thus the flow of control and the sequence of accesses to data often have little relation to what would be reasonably expected by looking at the text of the program, violating the principle of least astonishment. Thread safety is a property aimed at minimizing surprising behavior by re-establishing some of the correspondences between the actual flow of control and the text of the program.” [http://en.wikipedia.org/wiki/Threadsafe] (Siehe auch: http://en.wikipedia.org/wiki/Principle_of_least_astonishment)

„A subroutine is reentrant, and thus thread-safe, if 1) the only variables it uses are from the stack, 2) execution depends only on the arguments passed in, and 3) the only subroutines it calls have the same properties. Such a sub-routine is sometimes called a "pure function", and is much like a mathematical function.”

Arten von Threads

Threads auf Benutzerebene (user threads)



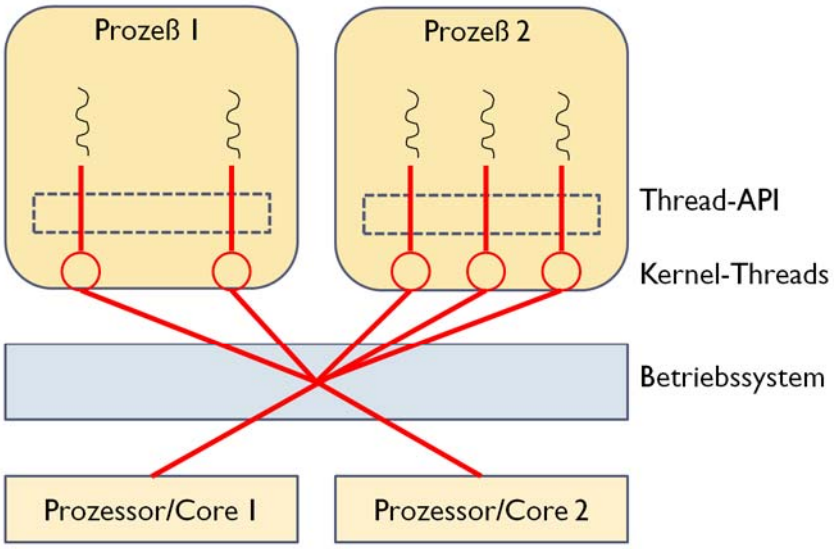
Arten von Threads...

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 bei blockierendem Systemaufruf
- Keine Koordination zwischen BS- und Thread-Scheduler

Arten on Threads...

Threads auf Systemebene (kernel threads)



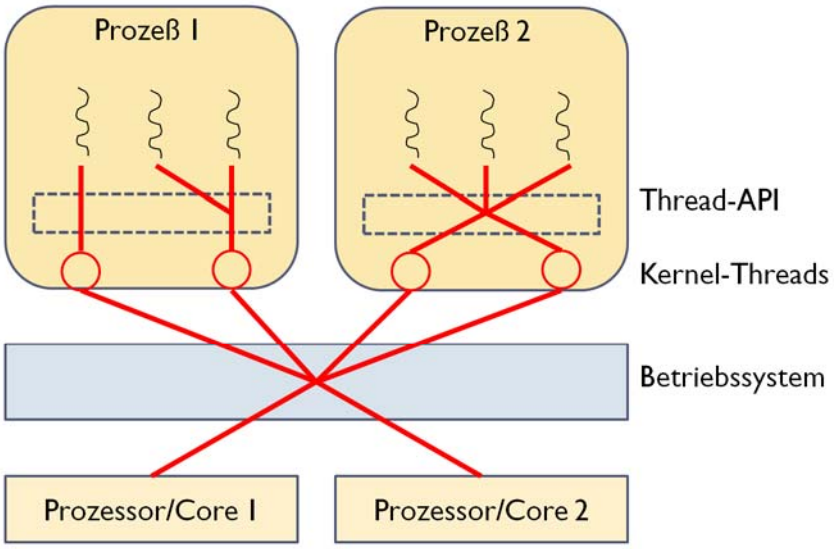
Arten von Threads...

Threads auf Systemebene...

- Langsamer als User Threads wg. Systemaufrufen
  - Z.B. Zeit zur Erzeugung auf SparcStation2
    U´Thread: 50µs / K´Thread: 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

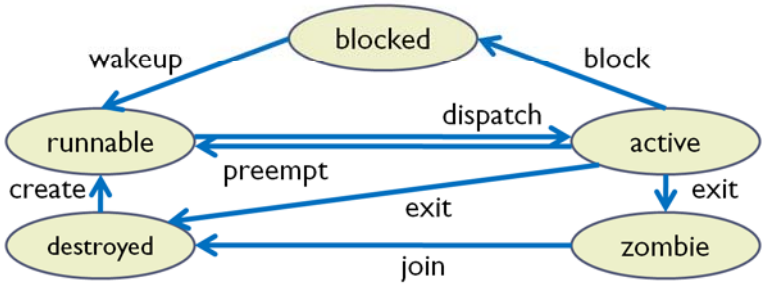
- Anzahl der Kernel-Threads von der Thread-Bibliothek bestimmt

  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
  - Max. Zahl gleichzeitig blockierender Systemaufrufe

Thread-Scheduling

- Scheduling: Zuteilung rechenbereiter Threads an Prozessoren
- Präemptives Scheduling: rechnender Thread kann unterbrochen werden
- Threadzustände

  (POSIX-Implementierungsmodell)



Thread-Scheduling...

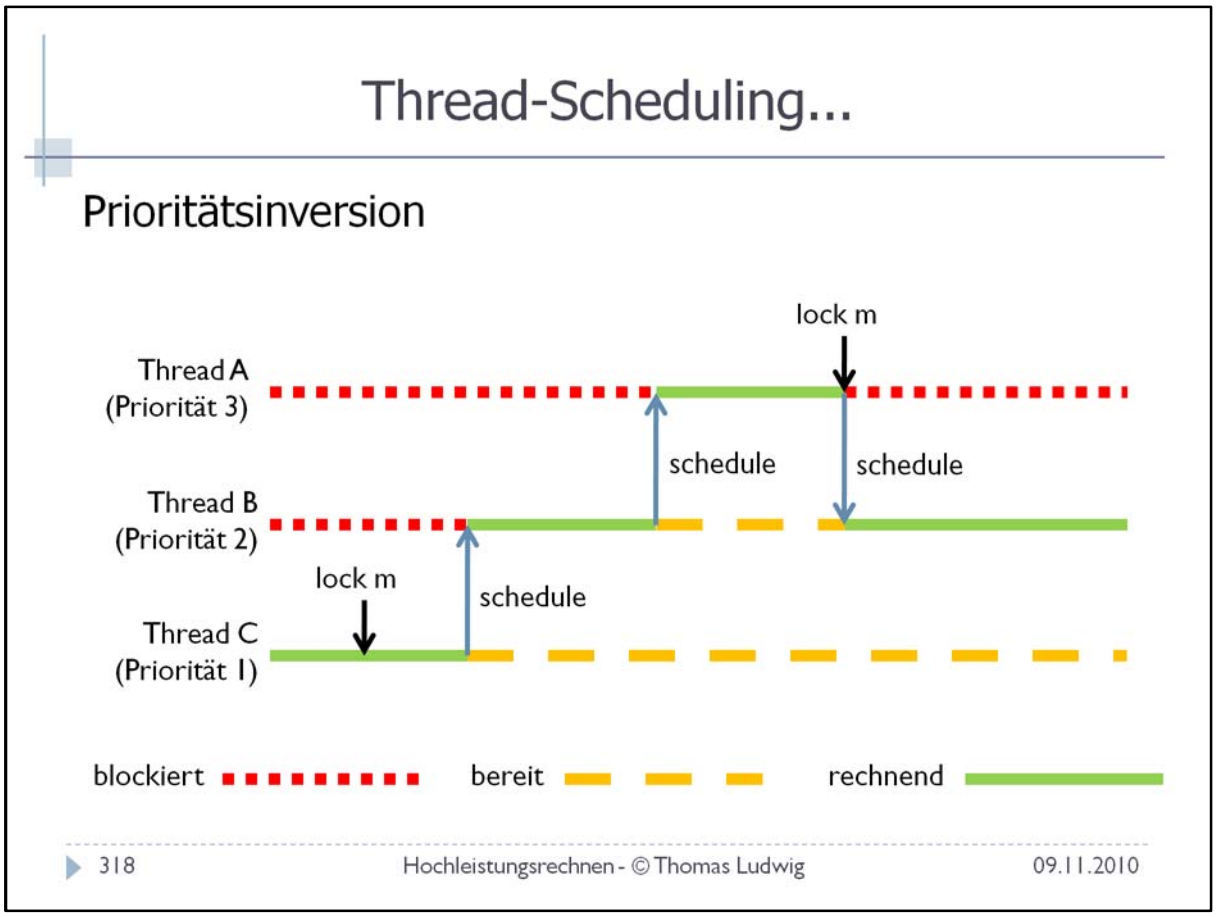
Schedulingverfahren

- 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

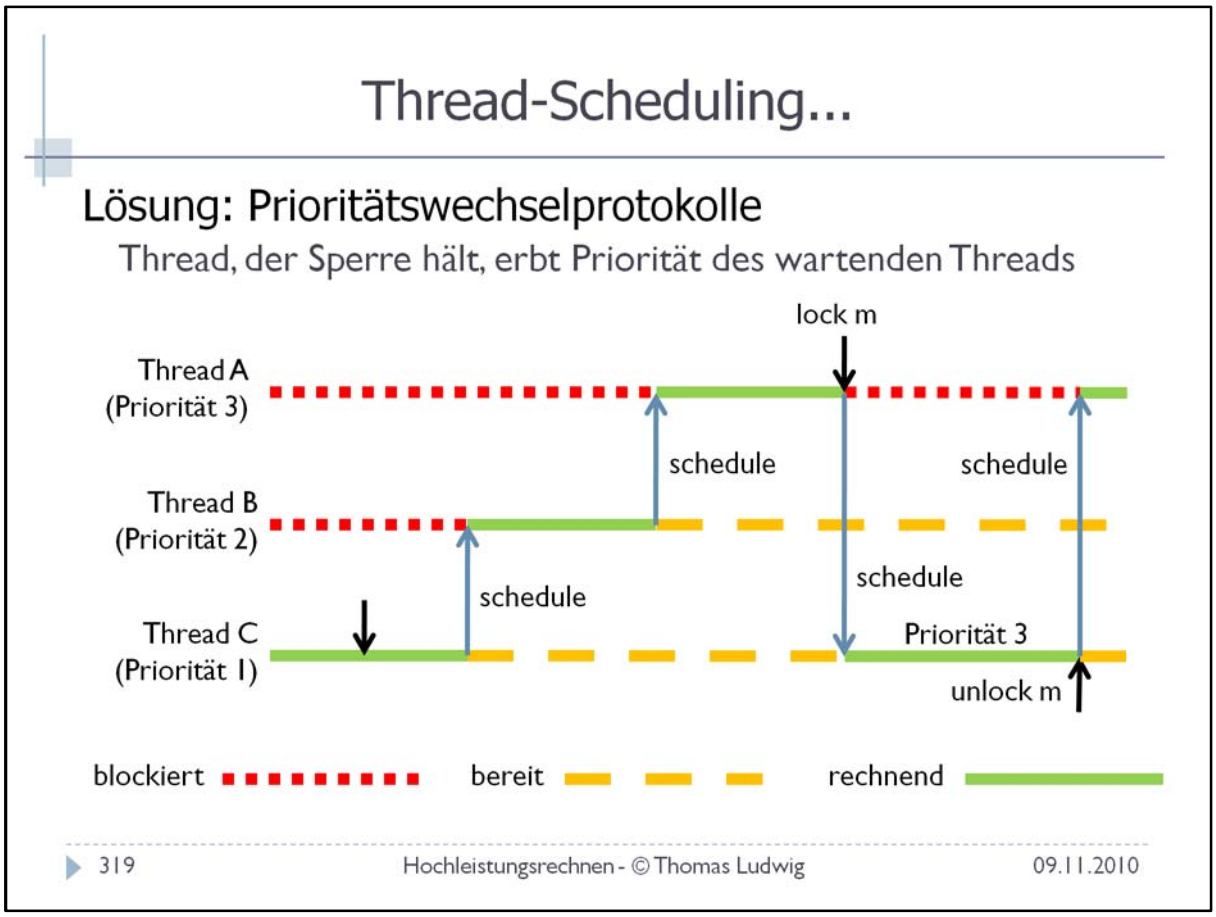
Thread-Scheduling...

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 C kommt erst einmal nicht an die Reihe und blockiert damit das Weiterlaufen von Thread A, der ja die höhere Priorität hätte.



Thread C kann weiterarbeiten und gibt in der Folge die Sperre wieder frei.

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

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

- 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

Siehe:
• http://en.wikipedia.org/wiki/Pthreads
• https://www.llnl.gov/computing/tutorials/pthreads/ – sehr gutes Tutorial zum Thema

Tippe unter Linux: `man pthreads`

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)
```


Das Kreieren eines Thread liefert eine ID zurück. Man übergibt als Parameter die Attribute für diesen Thread, einen Zeiger auf eine Funktion, die den Thread-Code darstellt und die Parameter für diesen Thread.

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 Abschlußsynchronisation

Die Pthread-Schnittstelle...

Thread-Attribute

- **`pthread_attr_init`**
  Attributstruktur initalisieren
- **`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)

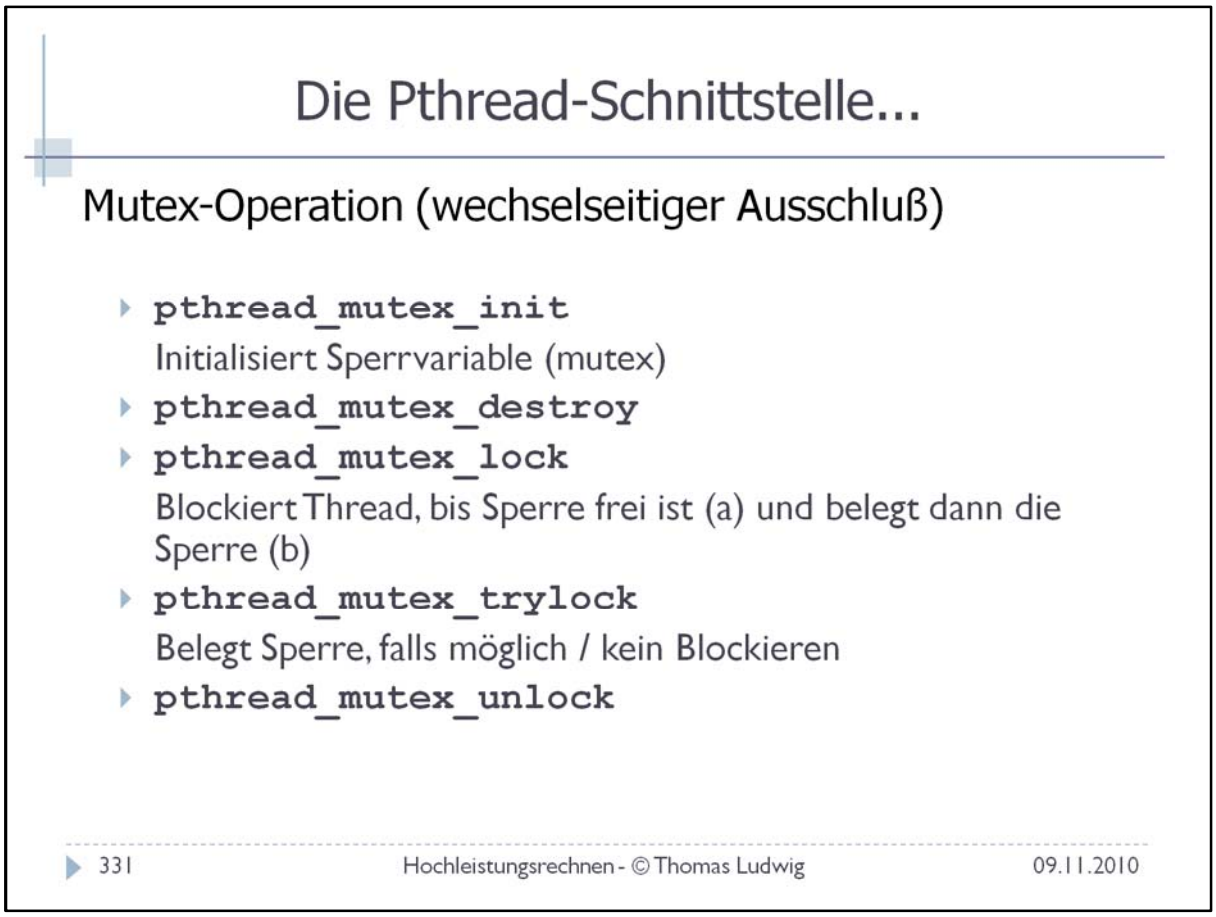
- **`pthread_sigmask`**

  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

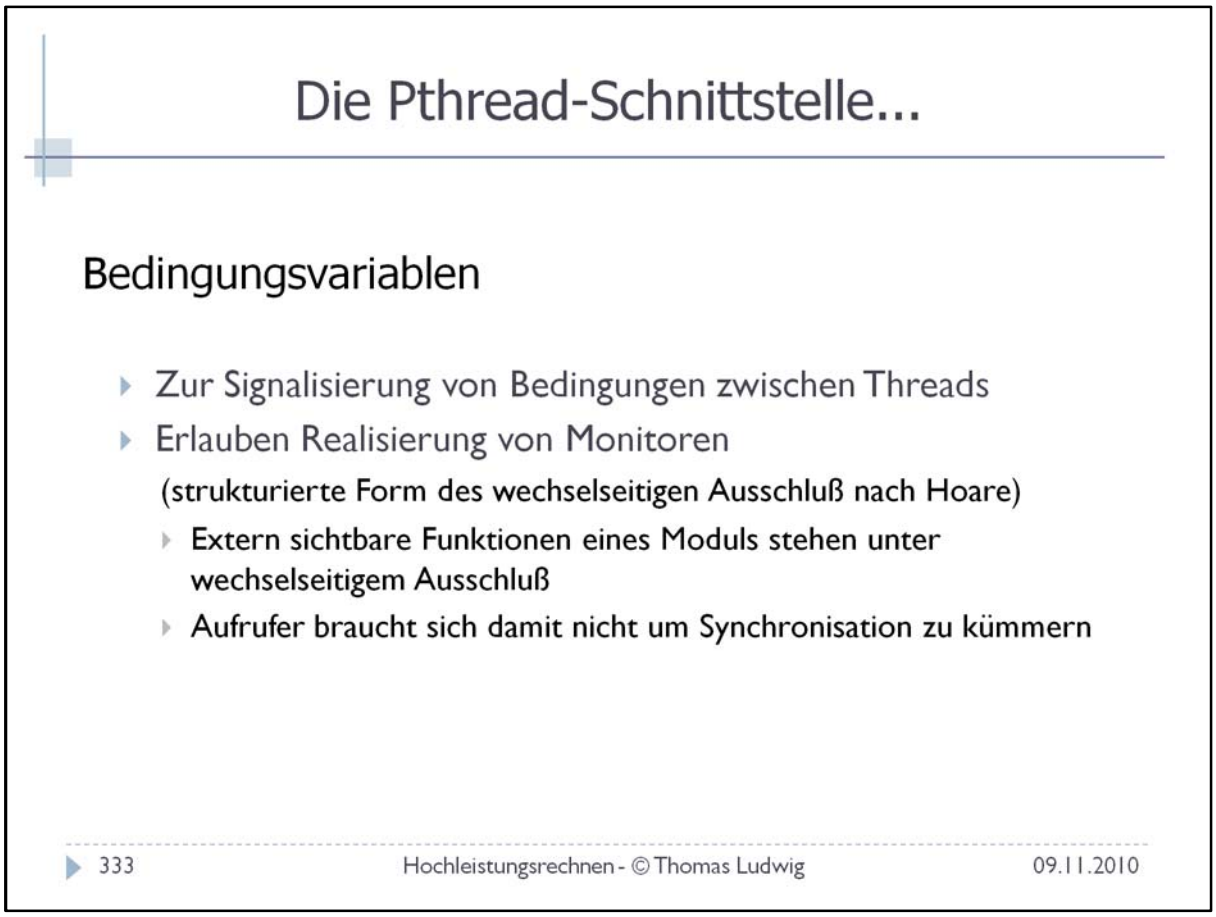


Mutex wird verwendet, wenn z.B. mehrere Schreiber auf eine Variable zugreifen. Hierdurch wird die Variable geschützt, so daß sie immer nur ein Schreiber zu einem Zeitpunkt manipulieren kann.

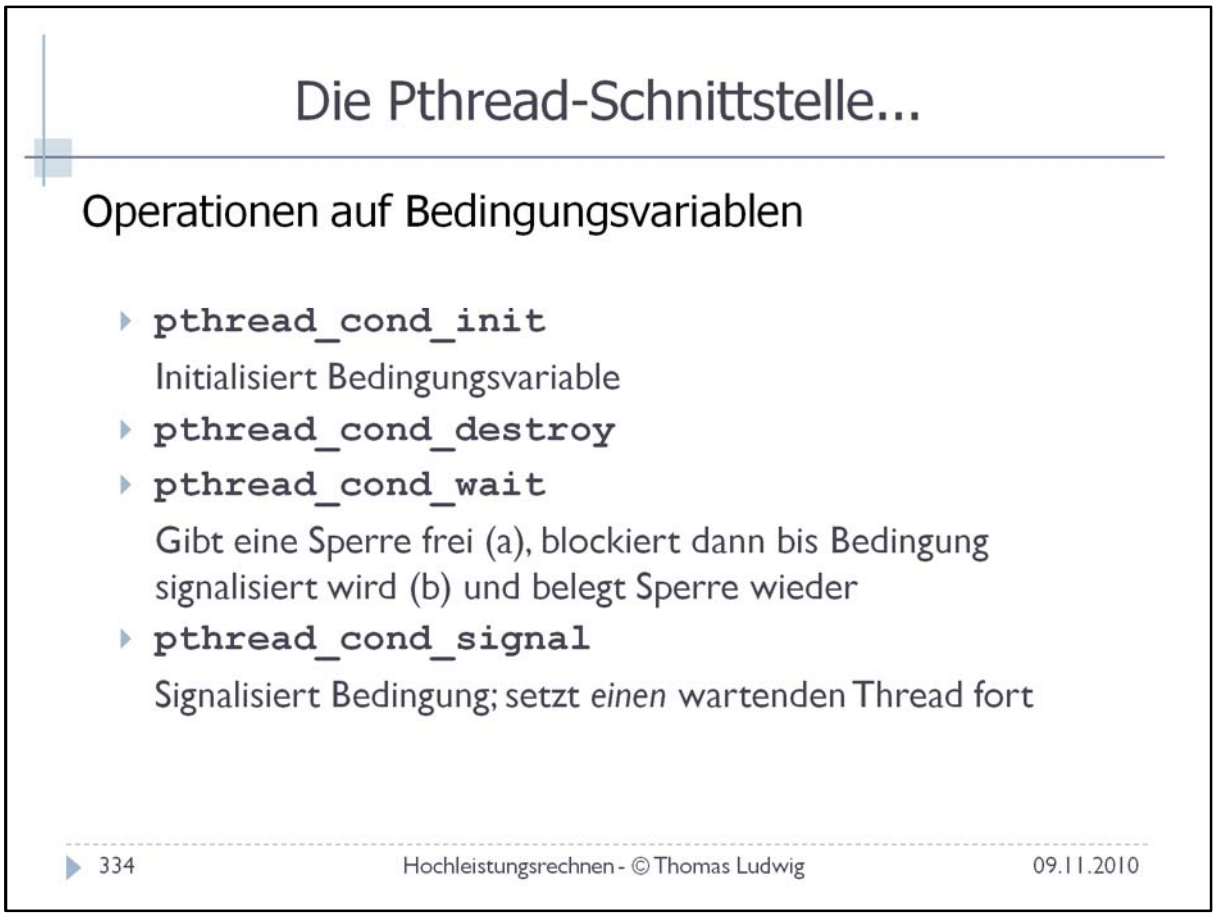
Die Pthread-Schnittstelle...

Anmerkungen zu Mutex-Operationen

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



Ein Monitor verwaltet z.B. Daten und eine Zugriffsfunktion (so etwa wie eine Klasse bei OO-Programmierung). Intern wird vom Monitor durch einen Thread sichergestellt, daß nur jeweils ein Aufrufer zu einem Zeitpunkt die internen Daten manipulieren kann.



Die zu lösende Situation besteht z.B. darin, daß ein Thread eine Aktivität ausführen soll, wenn eine bestimmte Bedingung erfüllt ist, z.B. der Wert einer Variablen überschreitet einen Grenzwert.

Die Lösung mit Mutexen sieht so aus, daß dieser Thread immer wieder Zugriff auf die Variable erwirbt und dann prüft und gegebenenfalls seine Aktionen ausführt. Das ist sehr kostspielig.

Mit Bedingungsvariablen wartet er blockierend darauf, daß ihm **ein anderer** Thread signalisiert, daß die Bedingung erfüllt ist. Die Signalisierung wird nur dort potentiell ausgelöst, wo der Wert der Variablen durch einen anderen Thread erhöht wird.

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

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

Siehe: http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

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 Betriebssytemmodus
- Sind allerdings dem Scheduler unterworfen
- Zur Strukturierung von Aktivitäten im Kernel

Vergleich der Ansätze

	Pthreads	MPI
Skalierbarkeit	Begrenzt	Ja
Fortran / C	Ja? / Ja	Ja / Ja
Hohe Abstraktion	Nein	Nein
Leistungsorientierung	Nein	Ja
Portierbarkeit	Ja	Ja
Herstellerunterstützung	Unix/SMP	Verbreitet
Inkrem. Parallelisierung	Nein	Nein

Inkrementelle Parallelisierung bedeutet: man kann das Programm stückweise von einem sequentiellen in ein paralleles Programm umbauen. Z.B. je nach Zeit, die einem zur Verfügung steht. Dies geht hier in beiden Fällen nicht. Man kann nur entweder ein richtiges voll parallelisiertes Programm erstellen, oder man arbeitet mit dem sequentiellen weiter.

Programmierung mit Threads
Zusammenfassung

- Threads trennen Einheiten der Ressourcen-belegung von Einheiten der Prozessorzuteilung
- 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
- Threads werden für echt parallele Programme aber auch als Strukturierungsmittel eingesetzt
- Die Pthreads-Schnittstelle definiert einen Standard zur Nutzung von Threads

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

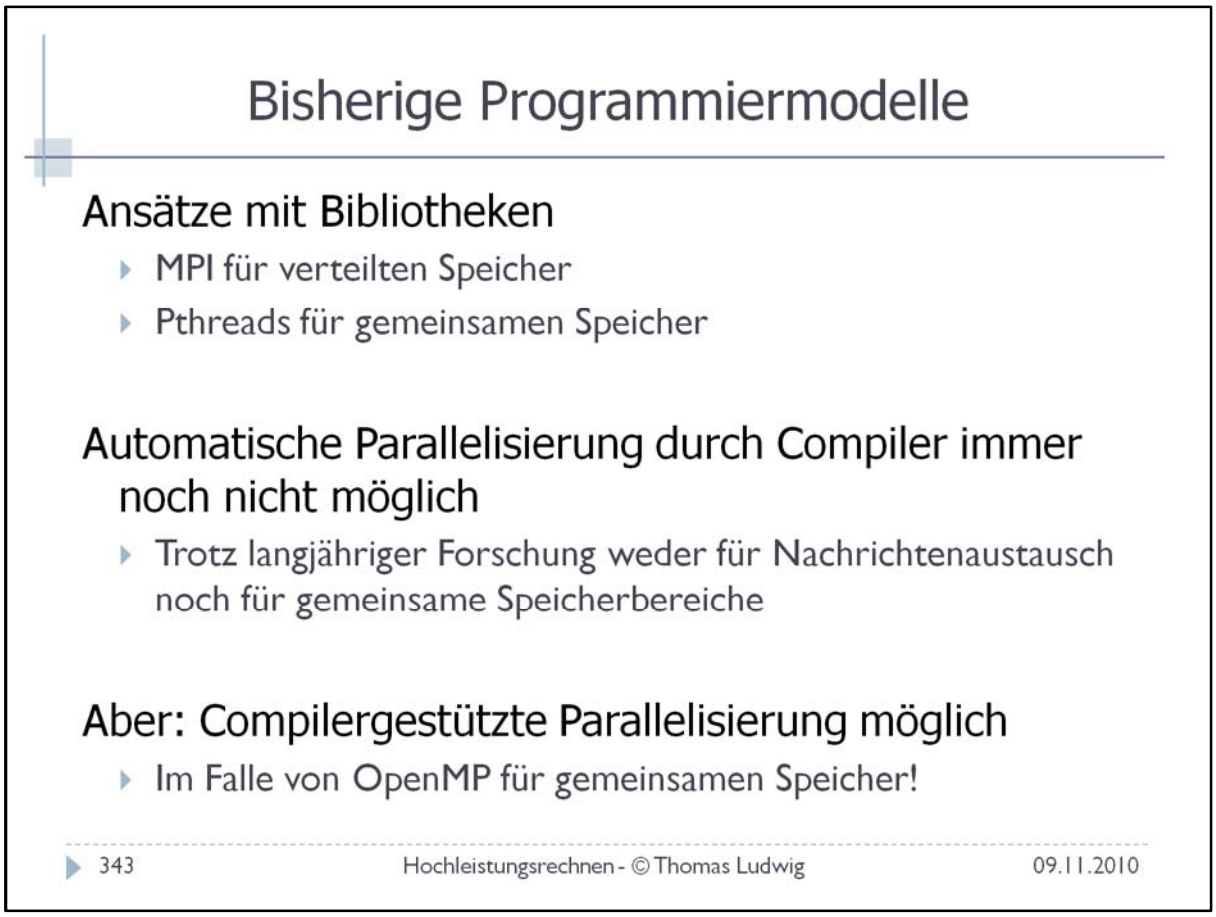
Siehe:
• http://www.openmp.org/
• http://en.wikipedia.org/wiki/OpenMP
• https://computing.llnl.gov/tutorials/openMP/ - sehr gutes Tutorial zum Thema

Bücher:
• R. Chandra et al.: Parallel Programming in OpenMP. Academic Press, London, UK. 2001. 230 Seiten.
• B. Chapman et al.: Using OpenMP: Portable Shared Memory Parallel Programming. Mit Pr, 2007, 353 Seiten.

Was bedeutet OpenMP?
• Kurze Version: Open Multi-Processing
• Lange Version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

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?



Der letzte Versuch für automatisch parallelisierende Compiler für Architekturen mit verteiltem Speicher, High Performance Fortran (HPF), wurde Ende der 90er erfolglos beendet.

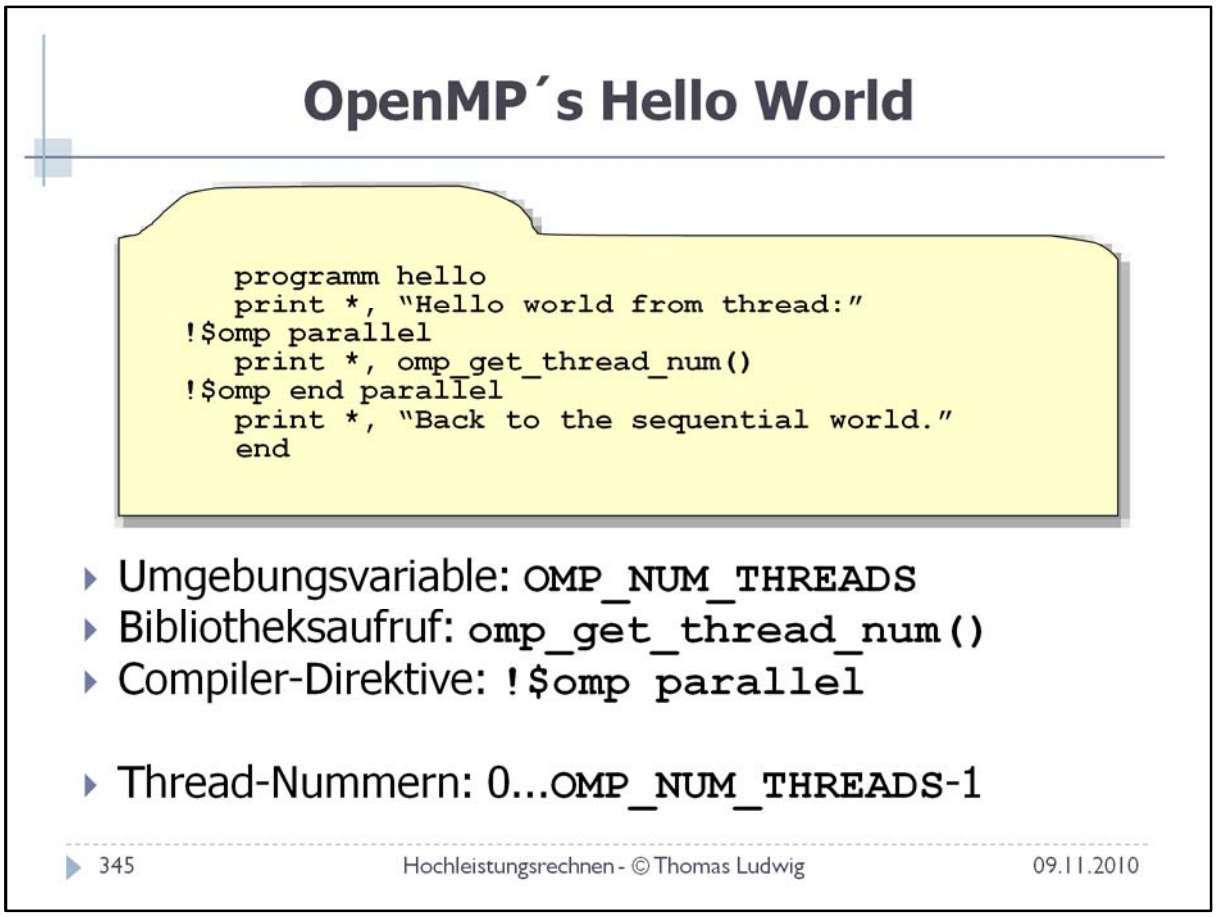
Siehe: http://en.wikipedia.org/wiki/High_Performance_Fortran

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!



Hier sieht man auch sofort alle drei Konstrukte, die OpenMP beinhaltet: Als wichtigstes die Compiler-Direktive, dann ein OpenMP-Bibliotheksaufruf und eine Verwendung von OpenMP-Umgebungsvariablen.

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

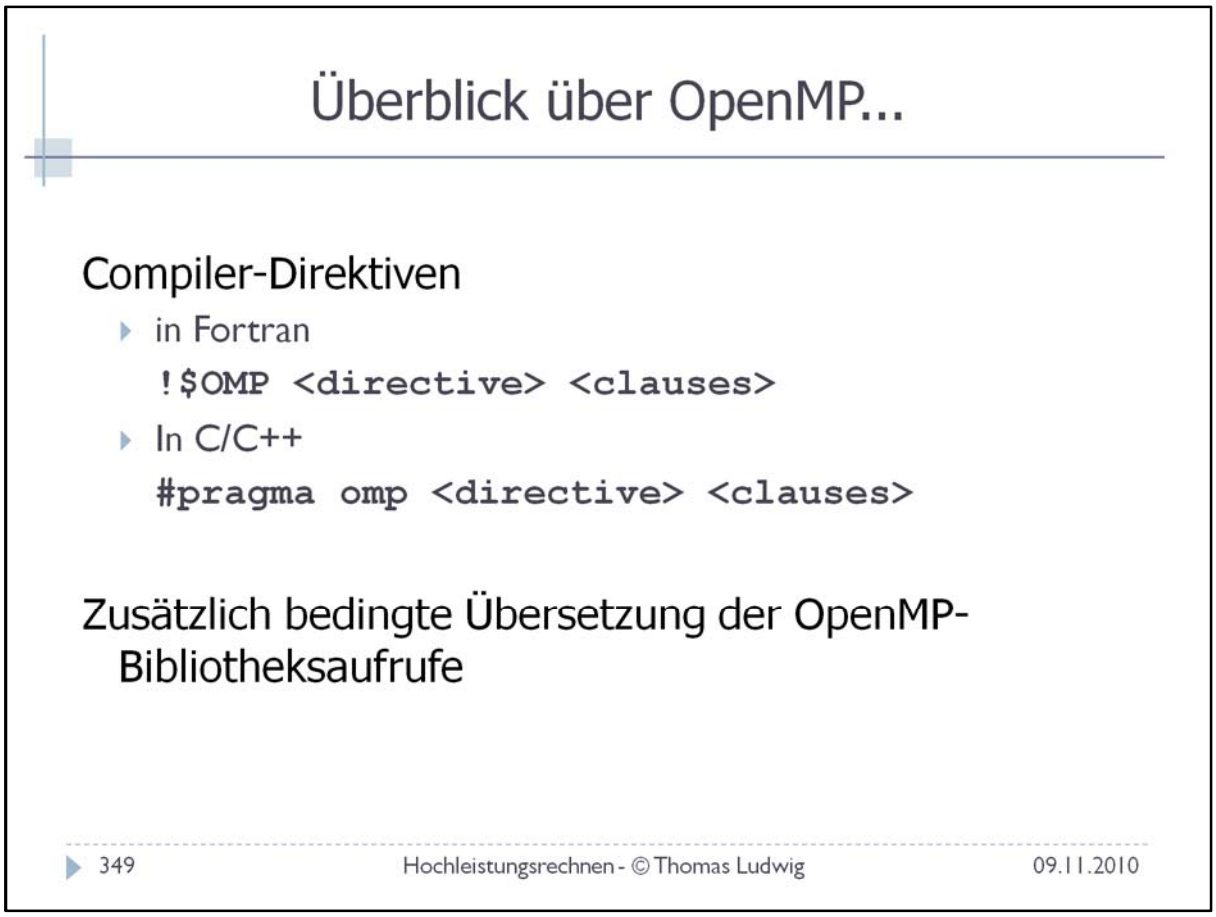
- OpenMP jetzt von allen Herstellern akzeptiert
- Von unabhängiger Organisation gefördert

Überblick über OpenMP

Portabilität: Neuübersetzung reicht aus

Kategorien der Spracherweiterungen

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



Die Compiler-Direktiven sind für einen nicht OpenMP-Compiler nicht wirksam.

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

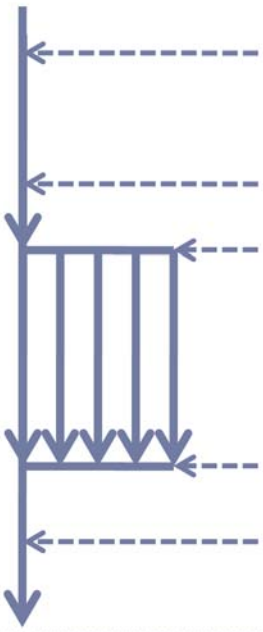
Parallelisierung einer Schleife...

```
      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 wiedereintrittsfähig sein (*reentrant*)

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**

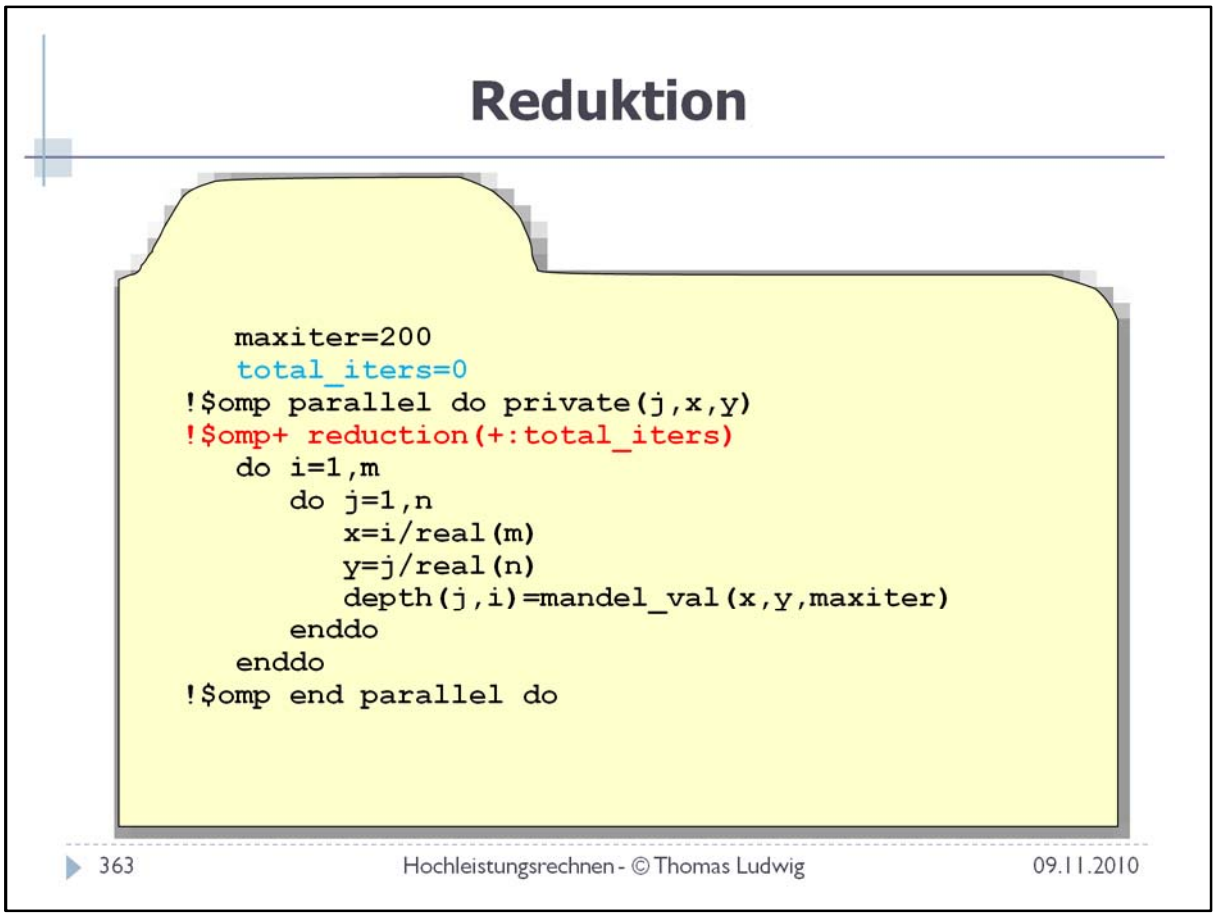
Eine Verkomplizierung...

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!



Die Variable total_iters wird hierbei erst als private-Variable behandelt. Ihre Aktualisierung ist somit unkritisch. Nach Abschluß der Schleife erfolgt die Reduktion, bei der allen privaten Einzelvariablen zur gemeinsam genutzten außerhalb der Schleife aufaddiert werden.

Das ´+´ bei ´!$omp+´ kennzeichnet die Fortsetzung der vorhergehenden Zeile. Fortran-Notation.

Schleifenparallelisierung

Hauptproblem der Praxis:

Datenabhängigkeiten zwischen Schleifenindizes

Bespiel:

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

Lösung des Problems

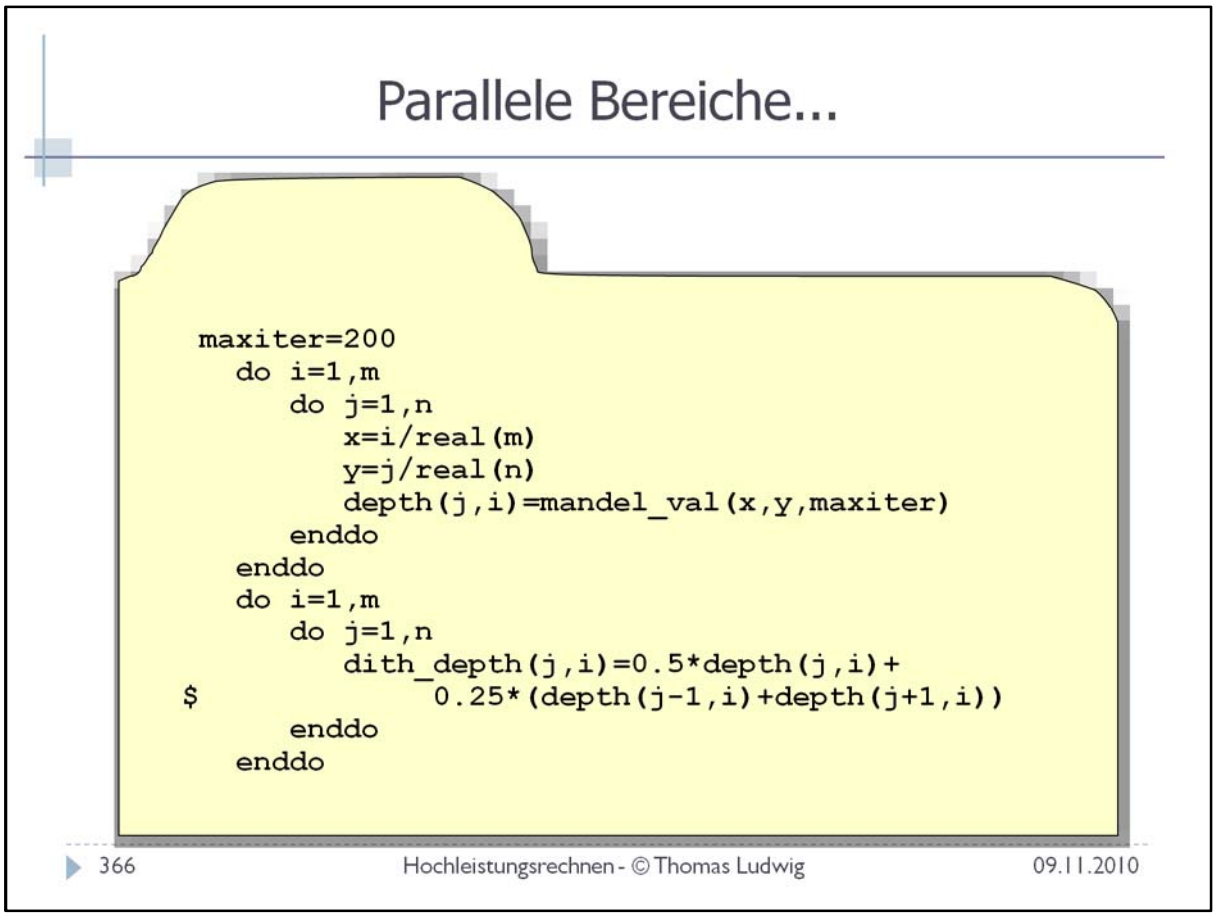
- Komplizierte Methoden zum Finden der Abhängigkeiten
  - Großes Forschungsgebiet seit vielen Jahren
- Verschiedene Methoden zu ihrer Beseitigung
  - Zusätzliche Variable
  - Zugriffskoordination mit kritischem Bereich

Parallele Bereiche

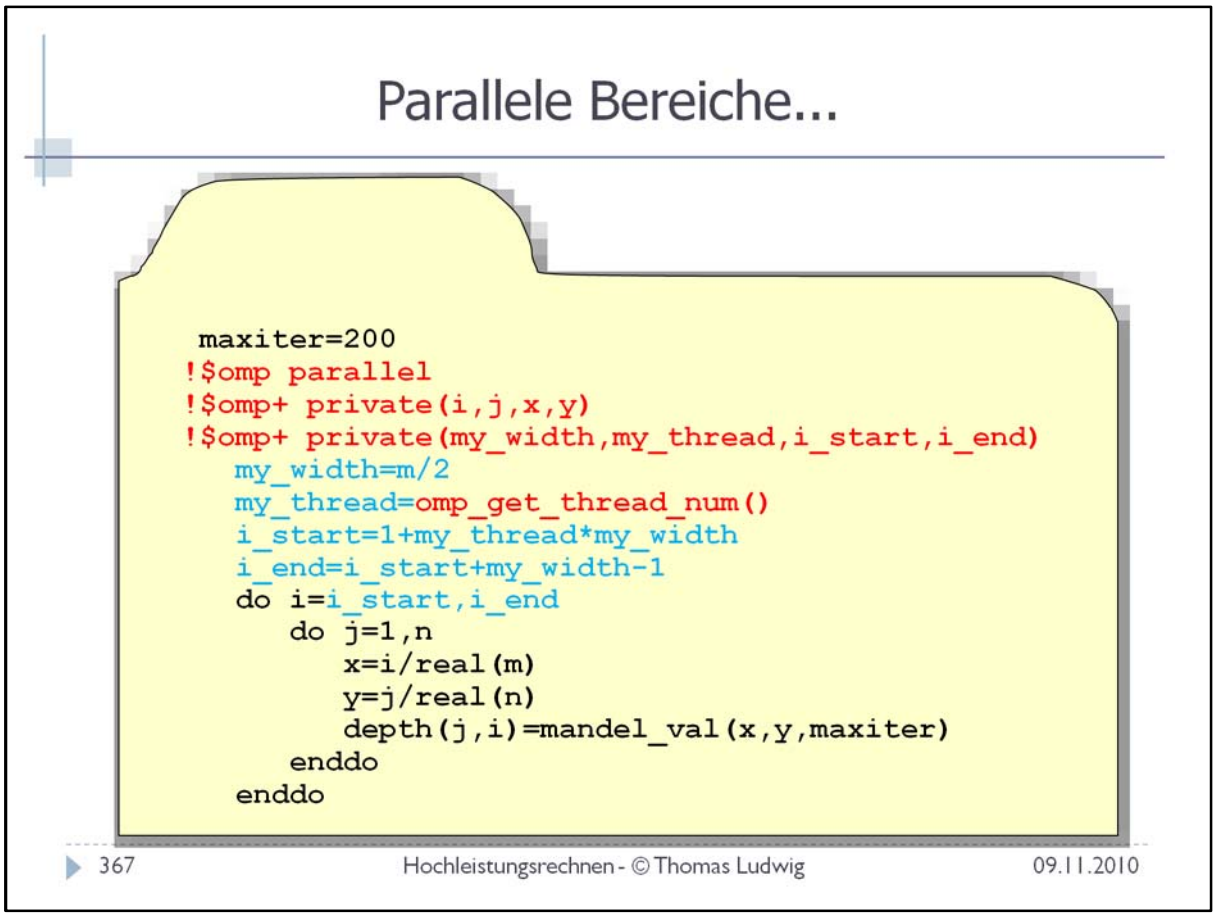
Bisher nur parallele Schleifen (feingranular)
Jetzt auch grobgranularer Parallelismus

Konstrukt: `parallel / end parallel`

- eingeschlossener Code wird mit mehreren Threads nebenläufig bearbeitet



Zwei Einzelberechnungen in Schleifen nacheinander ausgeführt.



Das Ganze wird hier auf 2 Threads aufgeteilt. Jeder Thread ermittelt seine Nummer und bestimmt daraus den Block von Indizes, die er durchlaufen muß.

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

Parallelisierung nicht 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 Schleifen-beginn
- `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

```
!$opm section [clause [,] [clause...]]
  [!$omp section]
     code for the first section
  [!$omp section
     code for the second section
     ...
  ]
  !$omp end sections [nowait]
```


Mit Klauseln kann man wieder z.B. die Verwaltung der Variablen oder das Scheduling steuern.

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

Siehe: http://openmp.org/wp/openmp-compilers/

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 übersetzbar
- 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