Leistungsmodellierung

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

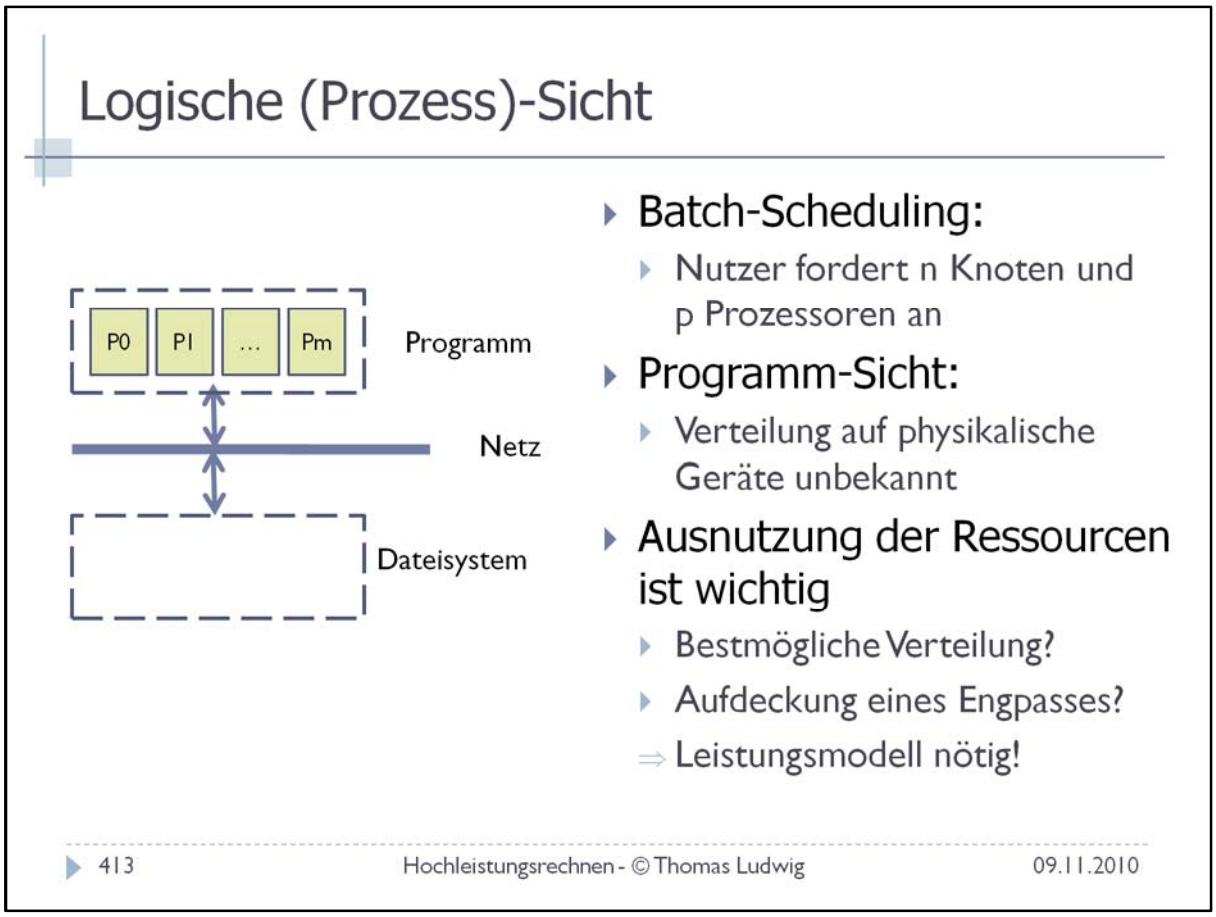
Leistungsanalyse: Analytisch/Mathematisch, Modellierung oder Hands-On, Grundgedanken zur möglichen Leistung eines Programms.

Modellierung: Hardware/Software-Verhalten ist komplex, daher modellieren/abstrahieren wir. In der Vorlesung modellieren wir das Clustersystem.

Messung: Sehr praktisch.

Leistungsmodellierung
Die 6 wichtigsten Fragen

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



In MPI ist es aber durchaus möglich eine sinnvolle Verteilung auf die Hardware zu erreichen, dies wird mit Topologien erreicht. Die Implementierungen unterscheiden sich allerdings darin sinnvolle Topologien zu ermitteln, daher wird oftmals von Hand platziert.

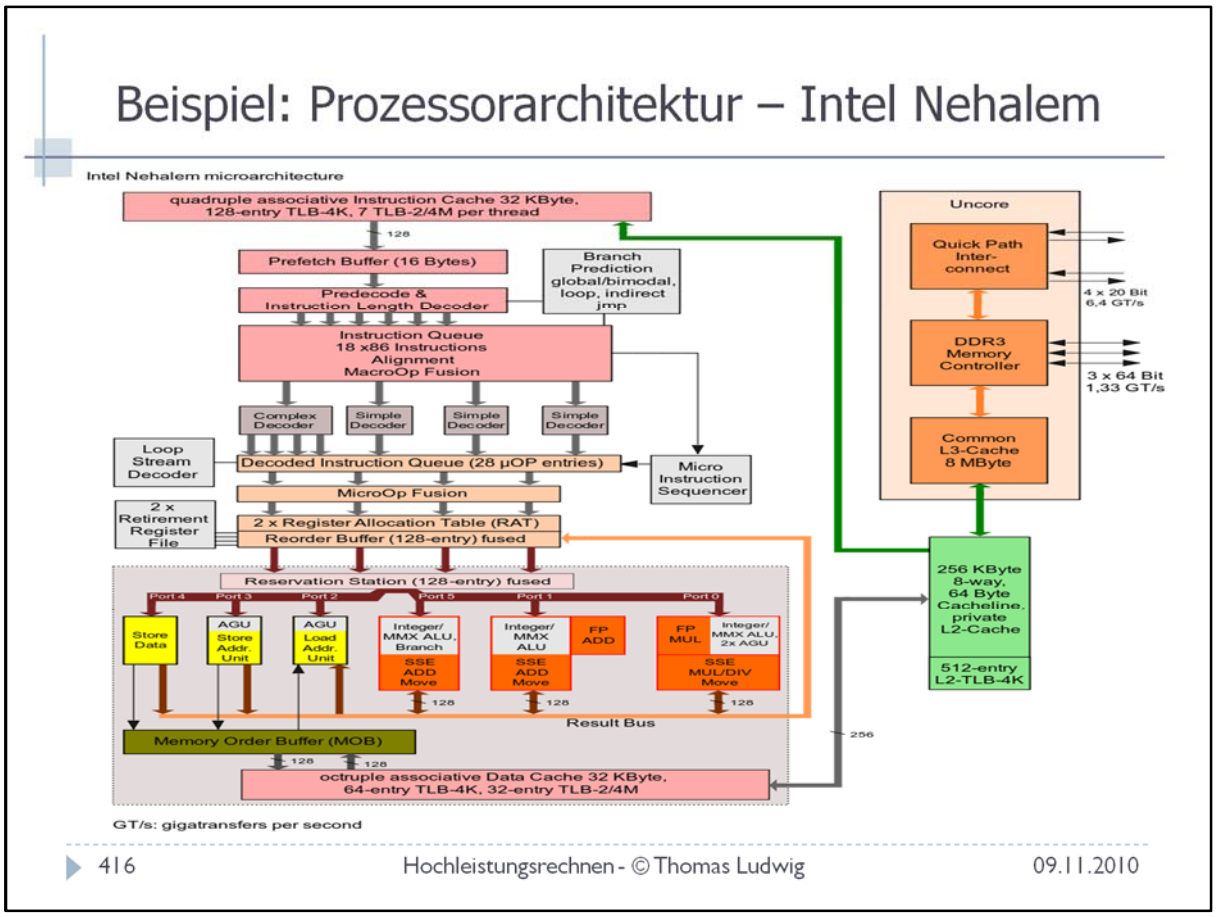
Leistungscharakteristika

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

Woher kommen die Modell-Referenzwerte?

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

Herstellerangaben sind typischerweise optimistisch.
Benchmarks wollen richtig programmiert sein, ebenfalls muss die Leistung ermittelbar sein, ohne Zwischenschichten mit zu erfassen.



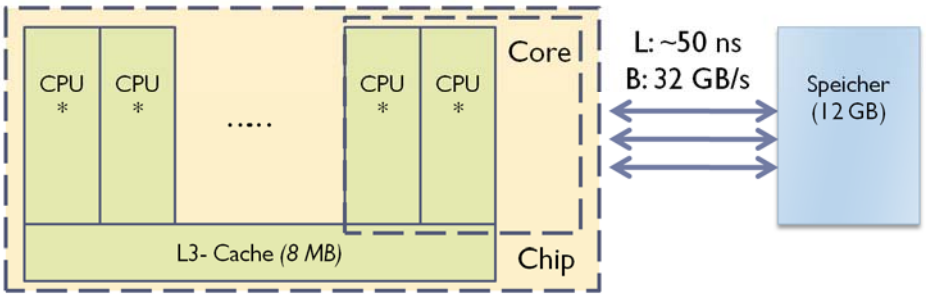
Komplexe Architektur des Intel Nehalem.
Intel Nehalem microarchitecture von: Appaloosa (von Wikipedia) unter Creative-Commons-„Namensnennung-Weitergabe unter gleichen Bedingungen 3.0 Unported“-Lizenz.

Inklusive Cache, http://www.anandtech.com/show/2594/9
Siehe auch: http://www.notur.no/notur2009/files/semin.pdf

Relevante Leistungscharakteristika

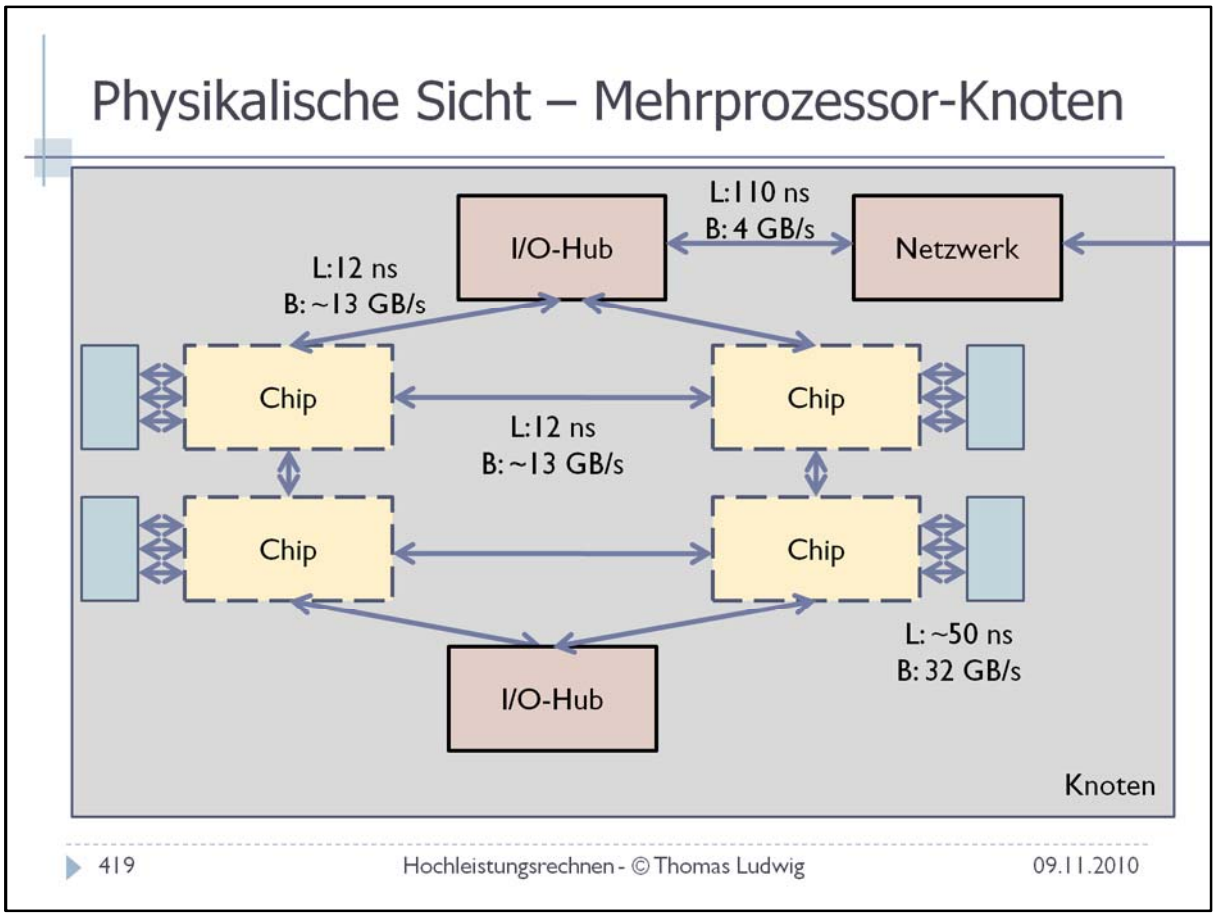
- Prozessorleistungsfähigkeit
  - Instruktionen/Sekunde, Simultaneous Multithreading (SMT)?
  - Größe der L1-, L2-, L3-Caches
- Speicheranbindung
  - Topologie: Einzelner Bus, Bus/Chip (z. B. Nehalem), Interconnect
  - Latenz und Bandbreite
- E/A-Leistungsfähigkeit
  - Bandbreite (pro Knoten und Server)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

Physikalische Sicht – Prozessor (mit SMT)



CPU kann pro Takt z. B. 2 FLOP ausführen

Pro Core:	*Latenz:*
L1: 32K Instruktion/32K Daten	*4 Zyklen*
L2: 256K	*10 Zyklen*



Cache/Speicher-Größen in Klammer
http://de.wikipedia.org/wiki/DDR-SDRAM#Latenzzeiten_im_Vergleich
DDR3-1333 CL-8-8-8 12 ns

IOHub - früher Northbridge – für PCIe – PCIe 2.0 / 8x erzielt 4000 MB/s

PCIe-Infrastruktur enthält ebenfalls Switches, PLXs Altair-Switch mit 110 ns Latenz.

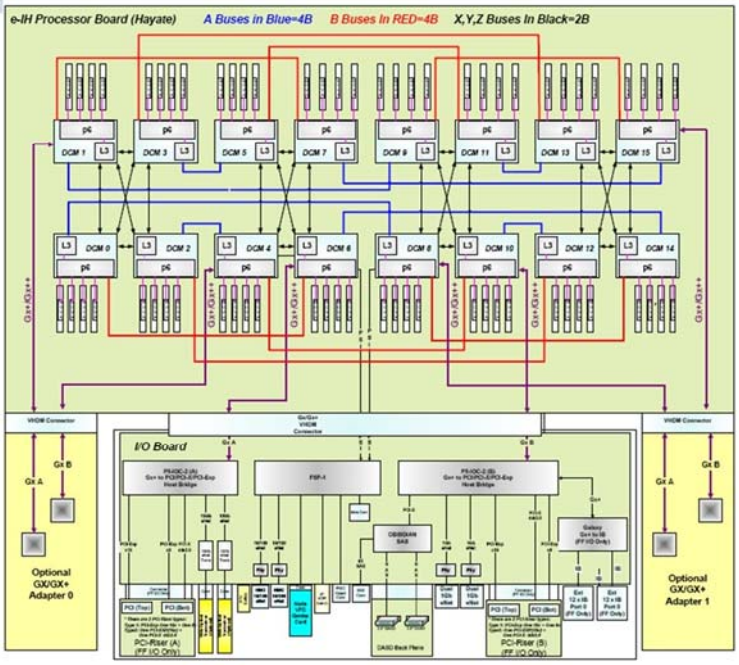
http://www.notur.no/notur2009/files/semin.pdf

Dieses Beispiel enthält zwischen manchen Chips keine Verbindung, dies hängt auch stark von der Architektur ab.
Werte sind vom Nehalem genommen.

Beobachtung

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

Praxisbeispiel: IBM-Power6-Server des DKRZ



16 Node-Cards a 2 Cores Power6 mit 4.7 GHz

256 Knoten
Infiniband-Vernetzung

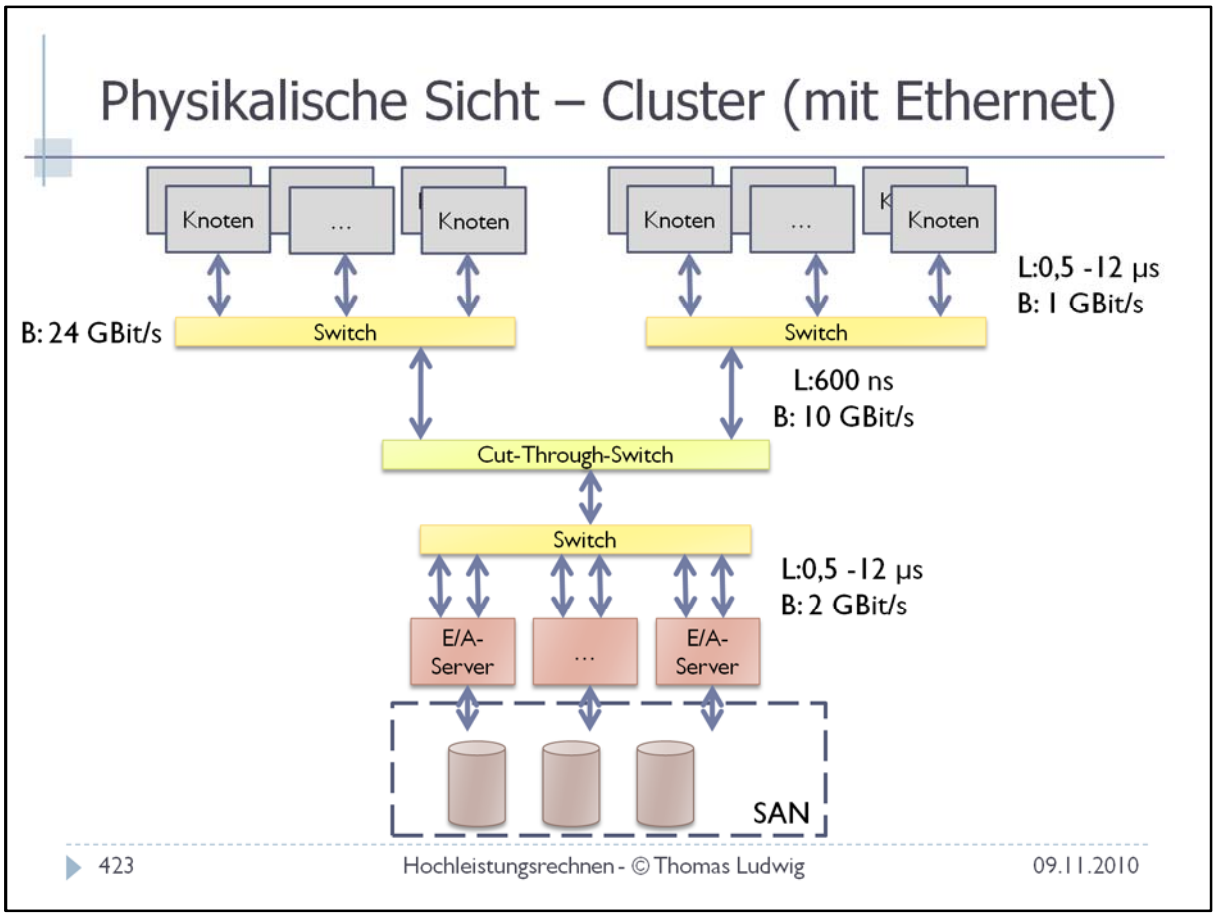
Betriebssystemeinflüsse

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

Prozessumschaltung == Context Switching
http://www.linuxjournal.com/article/2941
und http://www.cs.rochester.edu/u/cli/research/switch.pdf

Daten von einem 200-MHz-PC, Linux 2.0.30, Prozessumschaltung 19µs.
Mit einem 2 GHz Intel Xeon 2.6.17 mehr als 4.2 µs, falls die Daten in den L2 passen (anderer Test).

Swapping ist zu vermeiden.



Beispielwerte für Ethernet mit Store-and-Forward. Die Latenz ist abhängig von der Paketgröße, z. B. 1500 Bytes bei 1 GBit/s haben eine Latenz von 12 µs, ein 64-Byte-Paket nur 0,5 µs.
Bei Store-and-Forward wird das ganze Paket erstnächst in einem Puffer gespeichert, dann der Zielport bestimmt und das Paket weitergeschickt.
Die Zeit im Switch wird hier mit 0 angegeben.
Cut-Through-Switches im Vergleich inspizieren lediglich den Paketheader und können, im Falle dass keine Kollision vorliegt, dann die Daten direkt weiterschicken.

Die angegebene messbare Latenz beinhaltet ebenfalls den Mehraufwand der Kommunikation im Betriebssystem, z. B. Interrupt-Verarbeitung.

Die Bandbreite der Switches kann auch über mehrere Ports beschränkt sein, d.h. die Switche haben nicht die volle (und erwartete) Bisektionsbandbreite.
Je nach Anzahl der angeschlossenen Knoten kann die Bandbreite durch den Switch limitiert sein.
Oftmals wird die Anzahl der vermittelbaren Pakete auch in pps (packets per second) angegeben und das Nachrechnen der verfügbaren Bandbreite ist erforderlich.

Das SAN mit FiberChannel vertiefen wir hier einmal nicht.

Im Netzwerk gibt es auch Konflikte aufgrund der Topologie.

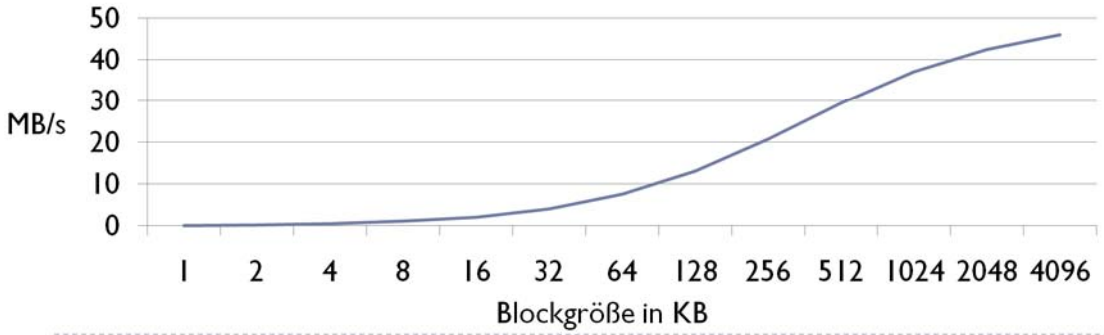
E/A-Leistung

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

In der Literatur zu finden unter „spatial und temporal access pattern“.
Die Optimierung in den zwischenliegenden Bibliotheken kann auch kontraproduktiv sein.

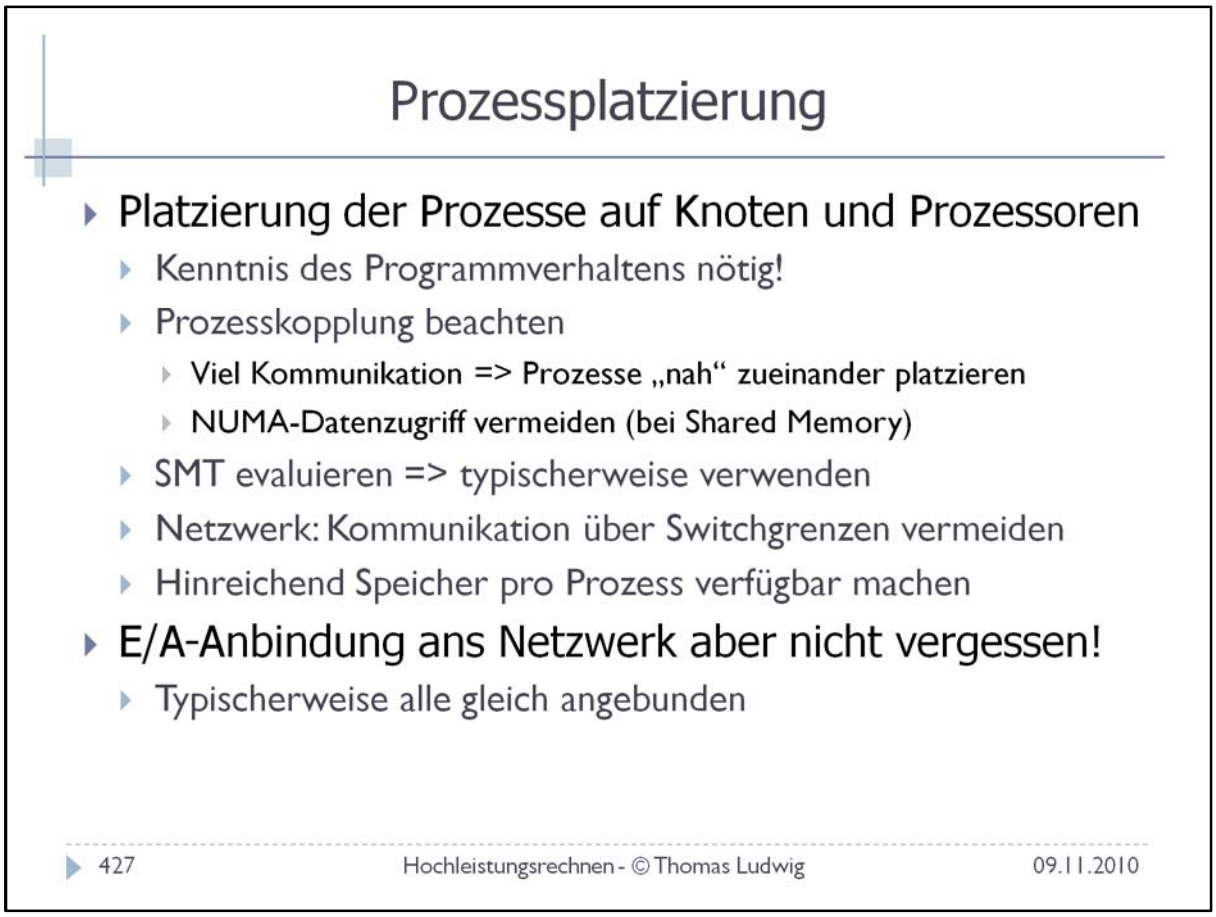
Charakteristika einer Festplatte

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



Praktische Anwendung

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



Auf die E/A-Anbindung gehen wir hier nicht ein, typischerweise sind die Knoten auf gleiche Weise an die Dateisysteme angebunden, daher kann dieser Faktor bei der Platzierung ignoriert werden.
NUMA-Datenzugriff: Normalerweise reserviert das Betriebssystem Speicher auf dem Speicher des Prozessors, welcher die Daten allokiert hat. Daher ist es bei Shared-Memory-Programmierung wichtig, dass jeder Thread seinen Speicher allokiert.

Gerade bei neueren Prozessorarchitekturen ist SMT in der Lage langsame Speicherzugriffe zu kaschieren und die Rechenwerke besser zu beschäftigen. 20% Leistungsgewinn sind keine Seltenheit. Dafür wird aber Cache-Speicher für die Ausführung des zweiten Prozesses benötigt.

Swapping ist zu vermeiden, daher muss genug Speicher pro Prozess zur Verfügung stehen.

Platzierungsbeispiel:

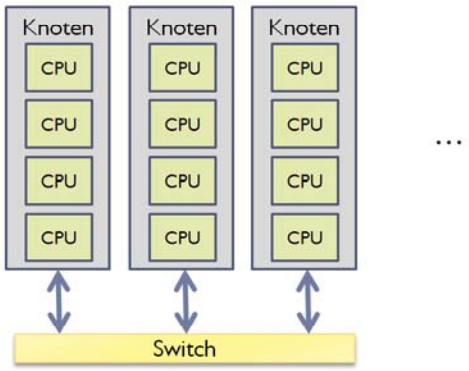
Fakten:
- Datenaufteilung
- 9 Prozesse

Eingabedaten:

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

Austausch erfolgt über die Linien

Vorhandene Hardware:

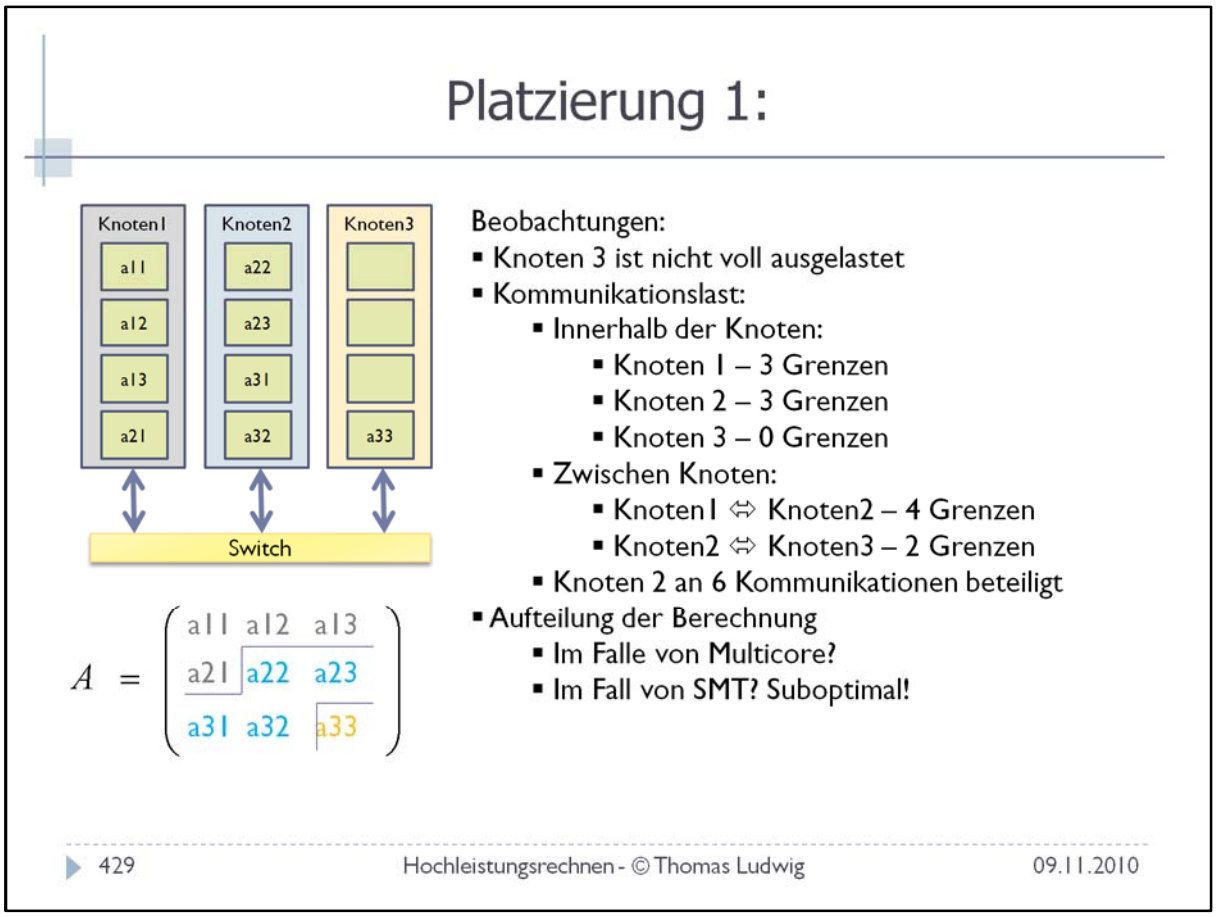


428 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

In dem Beispiel nehmen wir an, dass die Daten der Matrix räumlich so partitioniert sind wie dargestellt.
Als Beispiel sei die Aufteilung eines 2D-Gebietes, z. B. Landstrich in dem sich Objekte bewegen. Über die Grenzen der Gebiete muss Information ausgetauscht werden, bspw. Objekte die zwischen den Objekten wechseln, hierbei macht es keinen Sinn, dass Objekte von a11 nach a33 wandern, stattdessen wandern die Objekte erst nach a12 und a22.
Es stehen N Knoten mit jeweils 4 CPUs zur Verfügung.
Wie verteilen wir die 9 Prozesse auf die bestehende Hardware?
Der Algorithmus sei nicht in der Lage mit 12 Prozessen zu funktionieren.

Im Beispiel seien die CPUs echte Prozessoren ohne SMT.

Die Netzwerklast bestmöglich zu verteilen, hierfür muss eine Platzierung gefunden werden wo das Datenvolumen bzw. die Paketanzahl, die zwischen den Prozessen ausgetauscht wird minimal ist.
Das kann als ein Problem der Graphentherorie dargestellt und gelöst werden, Knoten sind Prozesse, Kanten werden mit Gewichten entsprechend des Datenvolumens versehen.
Das Problem ist NP-hart.

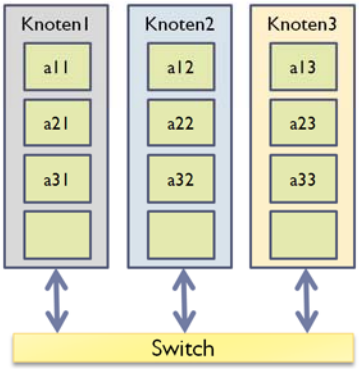


Die farbliche Darstellung verdeutlicht die Platzierung der Daten in der Matrix.
Über die Grenzen zwischen den einzelnen Datenbereichen muss Kommunikation erfolgen.
Wie viel der Kommunikation innerhalb der Knoten erfolgen kann und wie viel zwischen den Knoten ist für die Kommunikationslast entscheidend.

Im Multicore-Fall:
Die Prozesse, die miteinander kommunizieren, sollten auf einem Chip untergebracht werden, auf keinen Fall bspw. a11 und a13 auf einem Chip und a12 und a21 auf dem zweiten Chip rechnen lassen.
Prozess auf Knoten 3 hat mehr L3-Cache zur Verfügung.
I/O-Bandbreite steht ebenfalls dem Prozess a33 mehr zur Verfügung. Falls es einen Masterprozess geben sollte, so könnte dieser bspw. auf Knoten 3 platziert werden. Aber sequentieller Anteil sollte gering sein.

Falls bspw. SMT-fähig mit zwei Threads, so läuft Prozess a33 wesentlich schneller. Das kann zum Lastausgleich genutzt werden.

Platzierung 2:



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

$$A = \begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix}$$

430 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

Bewertung ob Platzierung 1 oder Platzierung 2 besser ist hängt von vielen Faktoren ab. Die Charakteristika der einzelnen Hardware-Komponenten und des Algorithmus sind endscheidend.
Die Balance der Prozesse auf die Komponenten wie in der Platzierung 2 ist vermutlich in den meisten Fällen vorteilhafter als Platzierung 1.

Falls die Bandbreite zwischen den Knoten der beschränkende Faktor ist, d.h. es wird sehr viel kommuniziert, so ist der Austausch zwischen Knoten 2 und den beiden anderen die Beschränkung. Beide Platzierungen müssen jeweils die Information von 6 Gebieten auf Knoten2 akzeptieren.
Innerhalb der Knoten erfolgt typischerweise der Austausch zwar schneller, aber auch vorhanden somit ist diese Konfiguration vermutlich in dem Fall auch etwas besser.

Programmanalyse/Optimierung

Optimierungszyklus:

1. Leistung erfassen
2. Vergleichen zur Modellvorstellung
3. Bewerten ob das Programm effizient läuft:
   1. Engpässe identifizieren
   2. Abschätzung des Leistungsgewinns durch Behebung
   3. Aufwandabschätzung für Tuning durchführen
   ⇒ fertig, falls das Programm „hinreichend“ gut ist
4. Tuning (evtl. Algorithmus), goto 1

Hinreichend gut, hängt vom Aufwand ab, der erzielt werden muss um ein Programm zu identifizieren.

Leistungsgewinn durch Optimierung

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

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

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

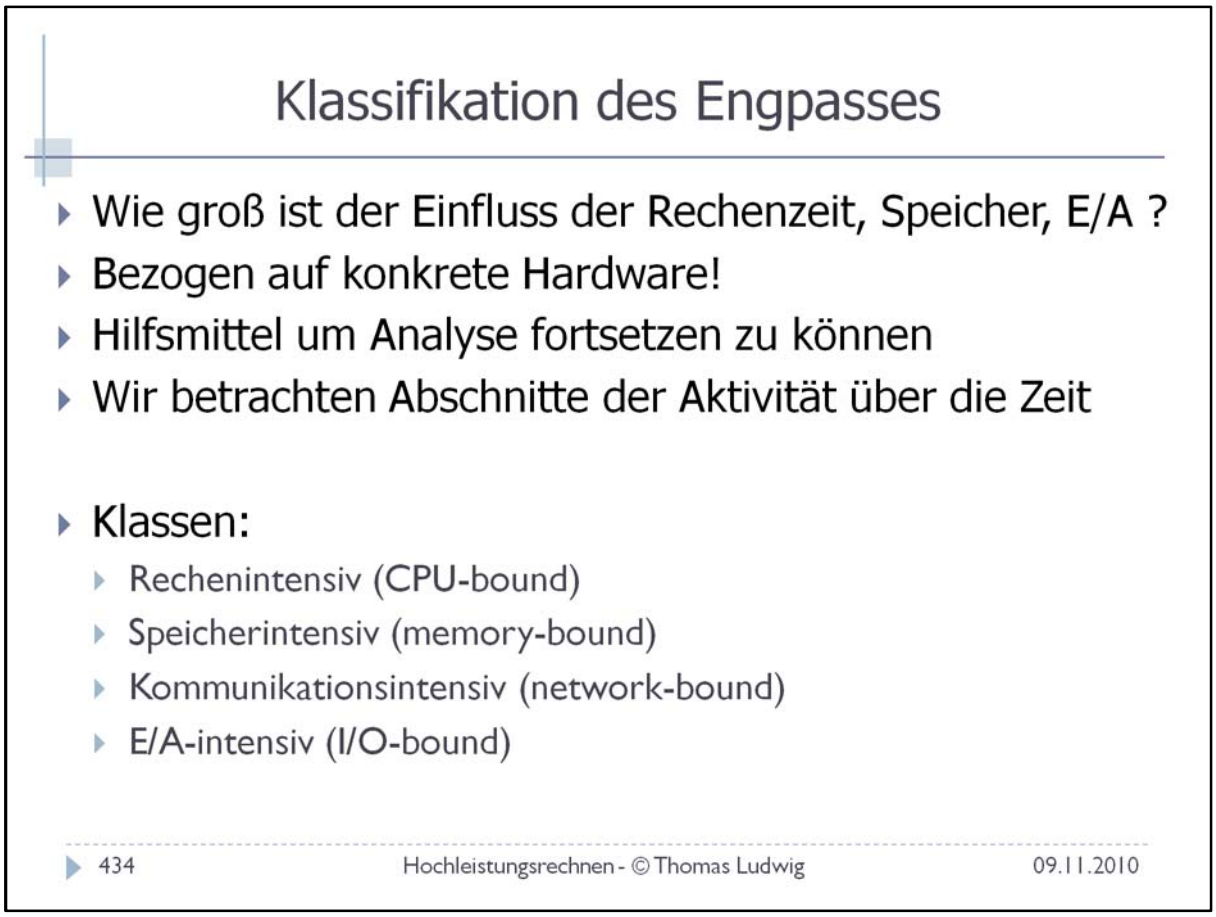
Wenn wir wissen, dass ein Programm 10% der Zeit in Kommunikation verbringt, so können wir maximal um den Faktor 1/0,9 durch die Optimierung schneller werden. Aber Skalierbarkeit kann stark zunehmen!

In manchen Gebieten sind Details tatsächlich relevant, beim Handeln an der Börse zählt jede Mikrosekunde. Spekulanten zahlen viel Geld um ihren Computer möglichst nah an dem System platzieren zu können, welches die Kurse festlegt.

Engpässe identifizieren

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

Eine grobe Bestimmung der Ursache eines Engpasses ist mittels Werkzeugen schnell machbar (siehe nächsten Vortrag). Die Identifikation der Ursache im Quellcode und die Behebung dagegen schwierig.

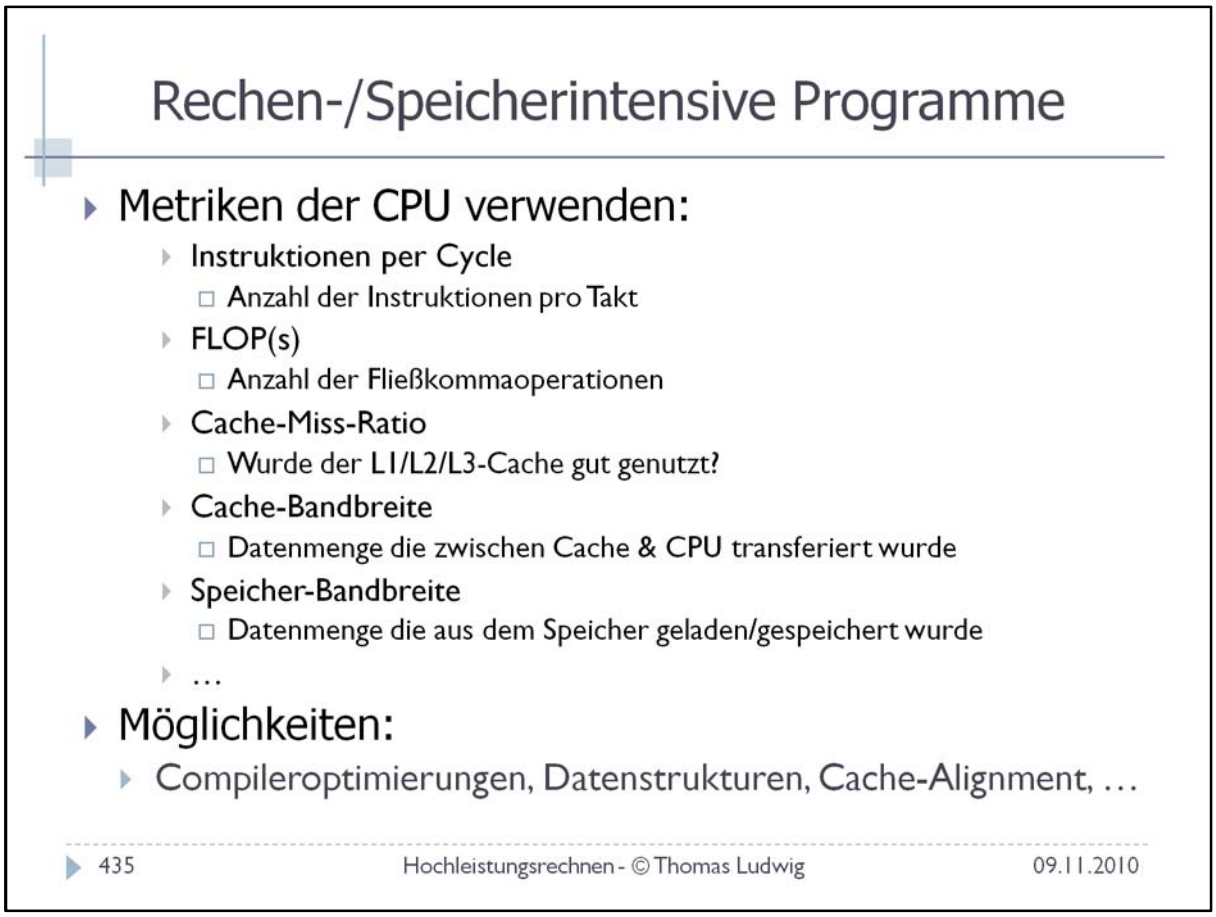


Memory-bound – zuwenig Speicher => Swapping, oder Zugriffszeit, d. h. Cache in CPU reicht nicht aus um Working-Set zu beinhalten.

Ein Programm kann rechen-intensiv sein, ebenso können wir Abschnitte der Programmlaufzeit als CPU-intensiv, andere als netzwerk-intensiv betrachten.

Die Klassifikation bezieht sich immer auf konkrete Hardware, d. h. wir können die Hardware nicht effizient ausnutzen, weil wir an einen Engpass des Systems gekommen sind.

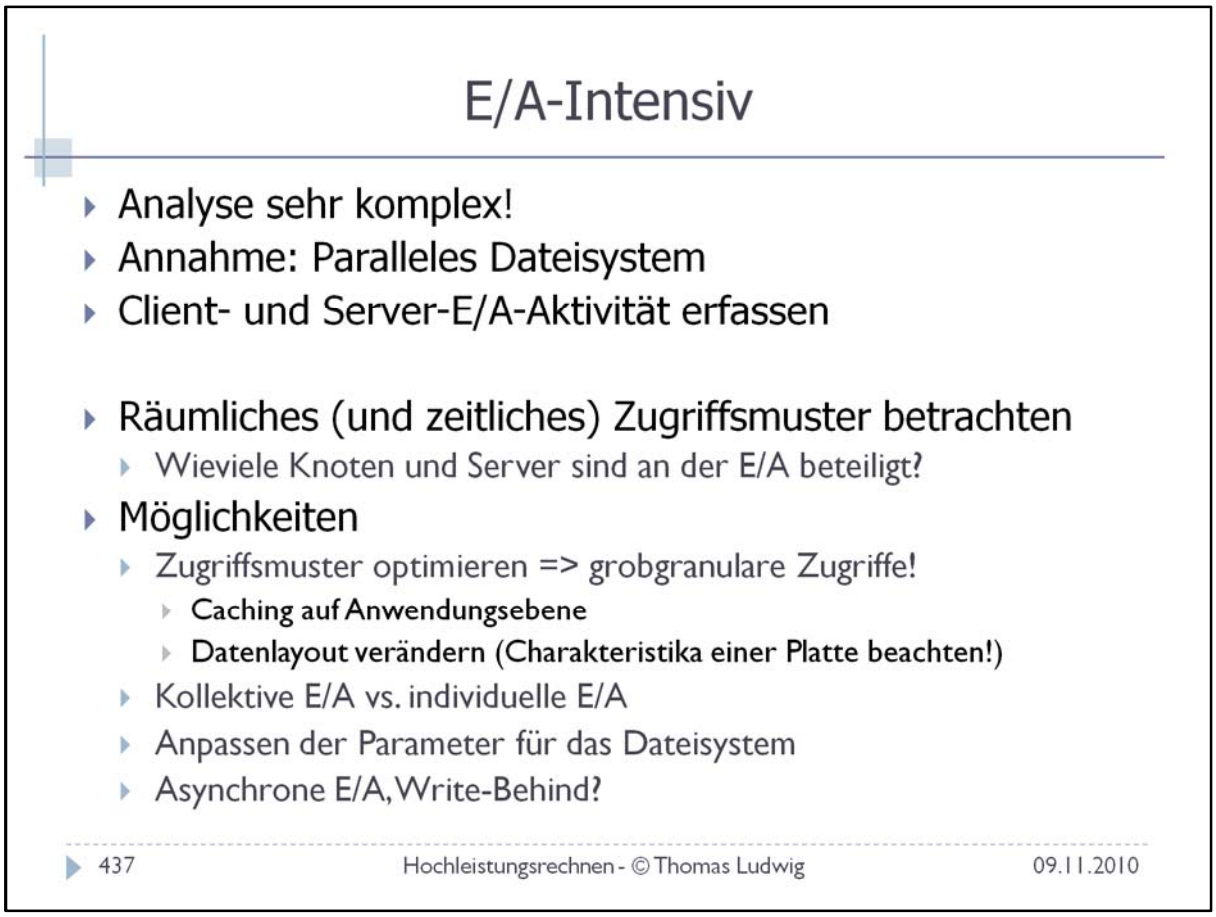
Wie optimieren wir den „Abschnitt“, den wir als wichtig identifiziert und klassifiziert haben? => Nächste Folien!



Hier exemplarisch ein paar relevante Daten.
Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors zu finden hier:
http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Kommunikationsintensive Programme

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



Die Parameter für das Dateisystem können bspw. mit MPI über Hints angepasst werden. In einigen Dateisystemen kann somit die Stripe-Größe festgelegt werden in der die Daten zwischen Servern/Platten aufgeteilt werden.

Write-Behind sollte von der genutzten E/A-Bibliothek oder der Implementierung des parallelen Dateisystems zur Verfügung gestellt werden. Oftmals entfällt dadurch die Notwendigkeit asynchrone E/A zu verwenden.

Der erste Lösungssatz sollte immer sein in der Anwendung die E/A so grobgranular wie möglich durchzuführen und alle Daten auf einmal anzufordern bzw. zu schreiben.

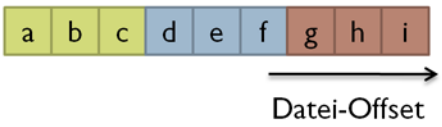
Ein Datenlayout könnte sein eine Matrix zeilenweise zu in die Datei schreiben zu wollen, die Ergebnisse aber spaltenweise berechnen und jeweils schreiben, dies ist sicherlich ineffizient und ähnelt der zufälligen E/A.

Dateilayout

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

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

- Variante 1:



- Variante 2:

438 Hochleistungsrechnen - © Thomas Ludwig 09.11.2010

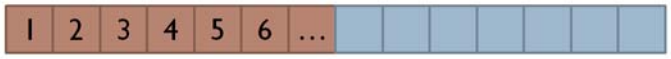
Die Matrix sollte gespeichert werden. Nehmen wir an die Ergebnisse werden spaltenweise berechnet und gespeichert.
Die Datei ist zeilenweise aufgebaut.

Variante 2 wäre hier vorzuziehen.

Zugriffsmuster und E/A-Durchsatz

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

  - Angenommene physikalische Abbildung auf zwei Servern/Platten:



- Lastungleichheit auf Servern meist schwer zu identifizieren!

Vermeiden sollte man, dass auf die Daten so zugegriffen wird, dass zeitlich immer nur eine Teilmenge der Server aktiv sein können. (Im Beispiel könnten die Server ebenfalls Platten sein.)
Im Beispiel wird von allen Programmen zunächst nur der erste Server verwendet und dann der zweite. Keine wirkliche parallelen Zugriffe auf die Server und Verlust der Leistung.
Die Reihenfolge in der die Leseanfragen an das Dateisystem übergeben werden ist in den Blöcken angegeben, jeder Prozess braucht die zusammenhängenden Bereiche, die farbig markiert sind.

In der Praxis sind derlei Lastungleichheiten deutlich schwerer zu identifizieren, E/A-Bibliotheken und zeitliche Abfolgen verändern das Zugriffsmuster.
Das Problem tritt normalerweise nicht auf, wenn der Zugriff auf hinreichend große zusammenhängende Blöcke erfolgt, da das parallele Dateisystem die Daten zwischen den Servern aufteilt.

Leistungsmodellierung
Zusammenfassung

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

Werkzeugarchitekturen

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

Leider keine Literatur. ☹

Werkzeugarchitekturen
Die zehn wichtigsten Fragen

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

Was sind Werkzeuge?

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

- Fehlersuche
- Optimierung
- Wartung
- Produktionsbetrieb

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

Was sind Werkzeuge?

Begriffe

- Offline-Werkzeuge

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

  Zeigen Informationen zum Programm gleichzeitig zum Ablauf an

Was sind Werkzeuge?

Begriffe...

- Interaktive Werkzeuge

  Gestatten eine direkte Interaktion mit dem Programm
- Automatische Werkzeuge

  Bearbeiten ohne Benutzereinwirkung das Programm zur Laufzeit

- Interaktiv und automatisch nur bei Online-Werkzeugen von Bedeutung

Was sind Werkzeuge?

Begriffe...

- Integrierte Werkzeugumgebungen

  Mehrere Werkzeuge unter einer einheitlichen GUI vereint und meist gemeinsam nutzbar

- Interoperable Werkzeuge

  Unabhängig voneinander entworfene und lauffähige Werkzeuge, die nun zusammenarbeiten

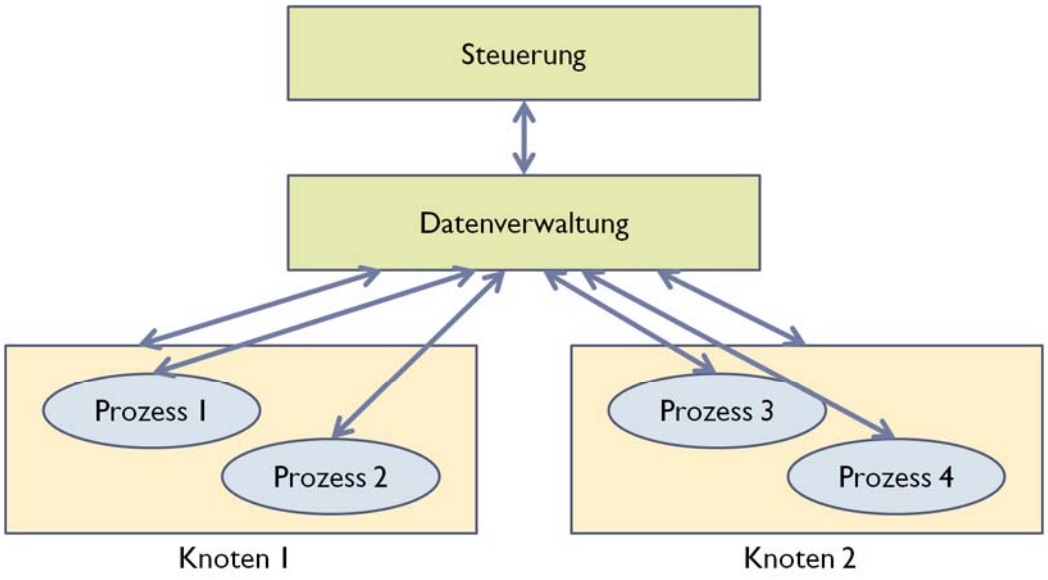
  Sehr schwer zu realisieren; Forschungsziel

Was sind Werkzeuge?

Hier nicht betrachtete Werkzeuge

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

Struktur von Werkzeugen



Struktur von Werkzeugen...

Abstraktes Strukturbild klar

Aber: Realität sehr komplex
Gründe:

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

Struktur von Werkzeugen...

Begriffe

- Monitorsystem

  Summe der Komponenten im System zur Beobachtung und Manipulation von HW und SW der Zielmaschine
- Monitor (knotenlokal)

  Komponenten auf einem Rechnerknoten, die HW und SW des Knotens überwachen können

Struktur von Werkzeugen...

Begriffe...

- Instrumentierung

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

    Primitivste Variante: Einbau von `printf(...)`
  - Binärcode-Instrumentierung

    Dynamisch eingebaute Sprünge in den Code des Monitorsystems

Struktur von Werkzeugen...

Offline-Werkzeuge

- Erfassung von Kenndaten zur Laufzeit des Programms
- Visualierung nach Programmende
- Keine Beinflussungsmöglichkeit

- Überwachung wird aktiviert und Monitorsystem speichert Spurdaten ab (*trace*)
- Visualisierung der Spur nach Programmende

Struktur von Werkzeugen...

Online-Werkzeuge

- Erfassung von Kenndaten zur Laufzeit
- Sofortige Visualisierung
- Sofortige Beeinflussung möglich

- Direkte Interaktion zwischen Steuerung und Datenverwaltung
- Optional Generierung von Spurdaten

Struktur von Werkzeugen...

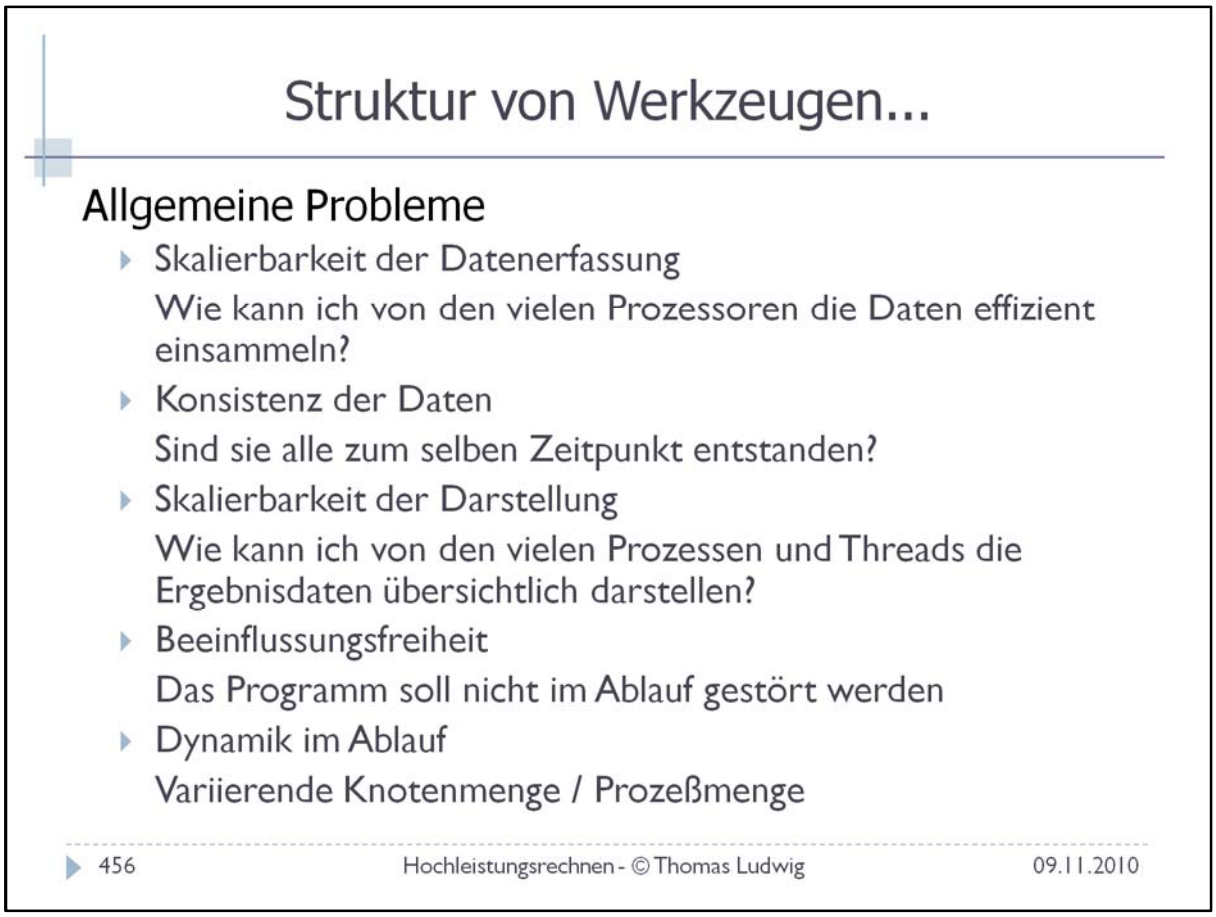
Interaktive (Online-)Werkzeuge

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

Struktur von Werkzeugen...

Automatische (Online-)Werkzeuge

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



Es gibt nur wenige Programme, bei denen die Anzahl der eingesetzten Knoten sich dynamisch ändert. Es gibt etwas mehr, aber immer noch wenige Programme, bei denen sich die Anzahl der Prozesse dynamisch ändert. Bei Einsatz von Threads ändert sich deren Anzahl naturgemäß laufend.

Werkzeuge

Die typischen interaktiven Werkzeuge:

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

Werkzeuge...

Die typischen automatischen Werkzeuge

- Ressourcenverwaltung
- Lastausgleich
- Sicherungspunktverwaltung
- Fehlertoleranzmechanismen

Werkzeuge zur Fehlersuche

Zweck

- Erkennen von Fehlerzuständen
- Auffinden von Fehlerursachen

Probleme

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

Werkzeuge zur Leistungsanalyse

Zweck

- Visualisierung wichtiger Leistungsdaten
- Erkennen von Leistungsengpässen

Probleme

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

Werkzeuge zur Programmvisualisierung

Zweck

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

Probleme

- Skalierung der graphischen Darstellung
- Zeitnähe

Werkzeuge zur Ergebnisvisualisierung

Zweck

- Visualisierung wichtiger Datenstrukturen
  Vor allem bei numerischen Anwendungen

Probleme

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

Werkzeuge zur Ablaufsteuerung

Zweck

- Manipulation algorithmischer Kenngrößen zur Laufzeit

  Z.B. Algorithmus konvergiert schlecht; dann Korrektur einzelner Parameter
- Wichtig bei langlaufenden Programmen

Probleme

- Wie bei Online-Ergebnisvisualisierung und Fehlersuche zusammen

Problem Solving Environments

Zweck

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

Probleme

- Vielfältig, deshalb noch kaum Vertreter dieser Gruppe

Werkzeuge zur Ressourcenverwaltung

Zweck

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

Probleme

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

Werkzeuge zum Lastausgleich

Zweck

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

Probleme

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

Werkzeuge zur Sicherungspunktverwaltung

Zweck

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

Probleme

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

Werkzeuge zur Fehlertoleranz

Zweck

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

Probleme

- Hoher Implementierungsaufwand

Erste Zusammenfassung

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

Konzepte zum effizienten Werkzeugentwurf

Wir betrachten jetzt Monitoring-Systeme für Online-Werkzeuge

Ziele:

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

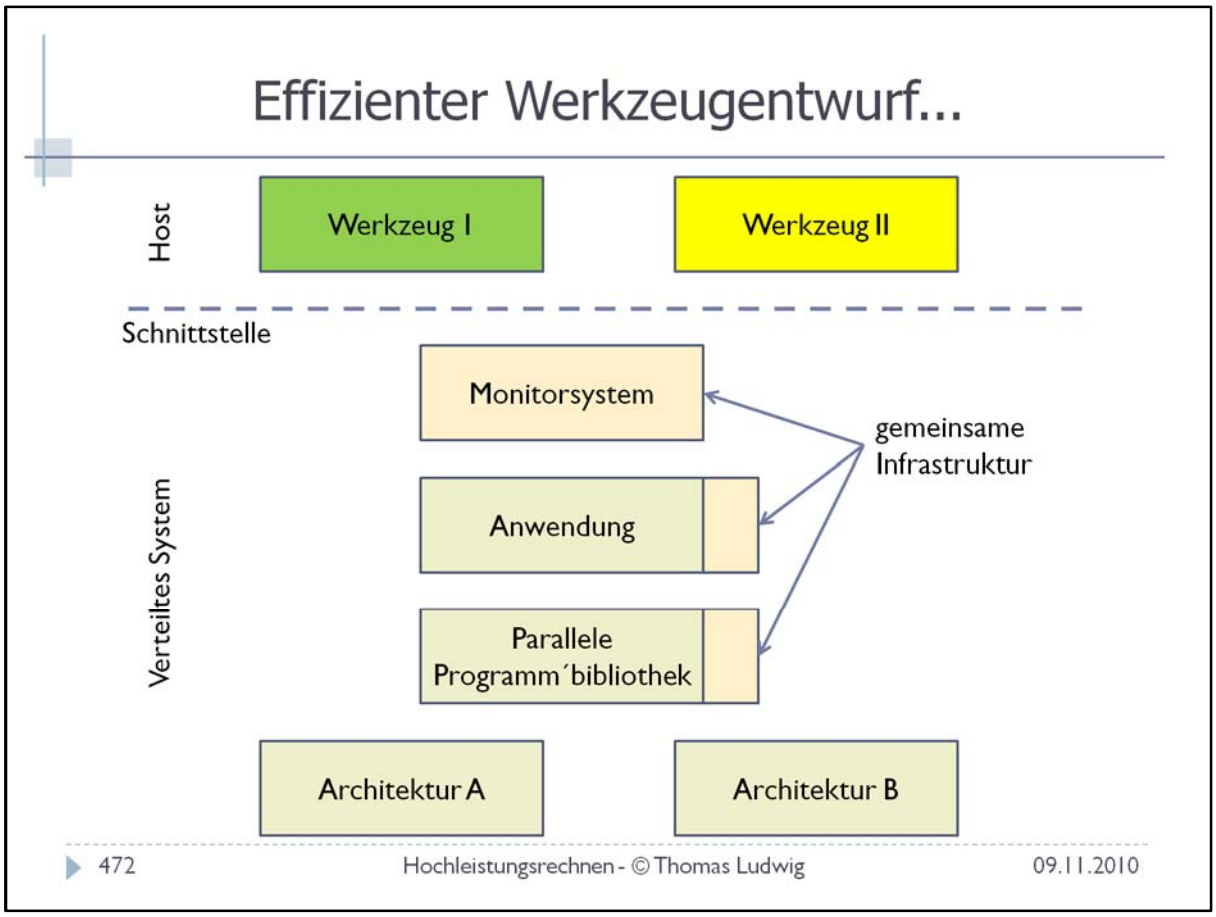
Warum geht das überhaupt?

- Die meisten Werkzeuge benötigen identische Funktionen des Monitorsystems

Effizienter Werkzeugentwurf

Vorgehensweise

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


Effizienter Werkzeugentwurf...
Host
Werkzeug I
Werkzeug II
Schnittstelle
Monitorsystem
gemeinsame
Infrastruktur
Verteiltes System
Anwendung
Parallele
Programm´bibliothek
Architektur A
Architektur B
472
Hochleistungsrechnen - © Thomas Ludwig
09.11.2010

Struktur der Schnittstelle

Aus Sicht des Werkzeugs

- Monitorsystem ist Server, der Dienste an Objekten anbietet

Diensteklassen

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

Objektklassen

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

Arbeitsprinzip der Schnittstelle

Ereignis/Aktions-Modell

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

Dienstanforderungen

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

Dienste der Schnittstelle (Beispiele)

Prozesse

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

Nachrichtenpuffer und Nachrichten

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

Programmierung der Werkzeuge

Vorgehensweise

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

Wichtiger Aspekt

- Verschiedene Werkzeuge benutzen gleiche Ereignisdefinitionen und ähnliche Aktionen

Beispiel: Leistungsanalyse

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

Wenn Prozeß p_2 einen Sendeaufruf startet: aktiviere einen integrierenden Zähler und addiere die Nachrichtenlänge ($par5) auf einen Zähler

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

Wenn der Prozeß den Aufruf beendet: halte den Zähler an

Resultat: der integrierende Zähler hält die Verweilzeit in Sendeaufrufen, der normale Zähler die gesendete Nachrichtenmenge.

Werkzeugarchitekturen
Zusammenfassung

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

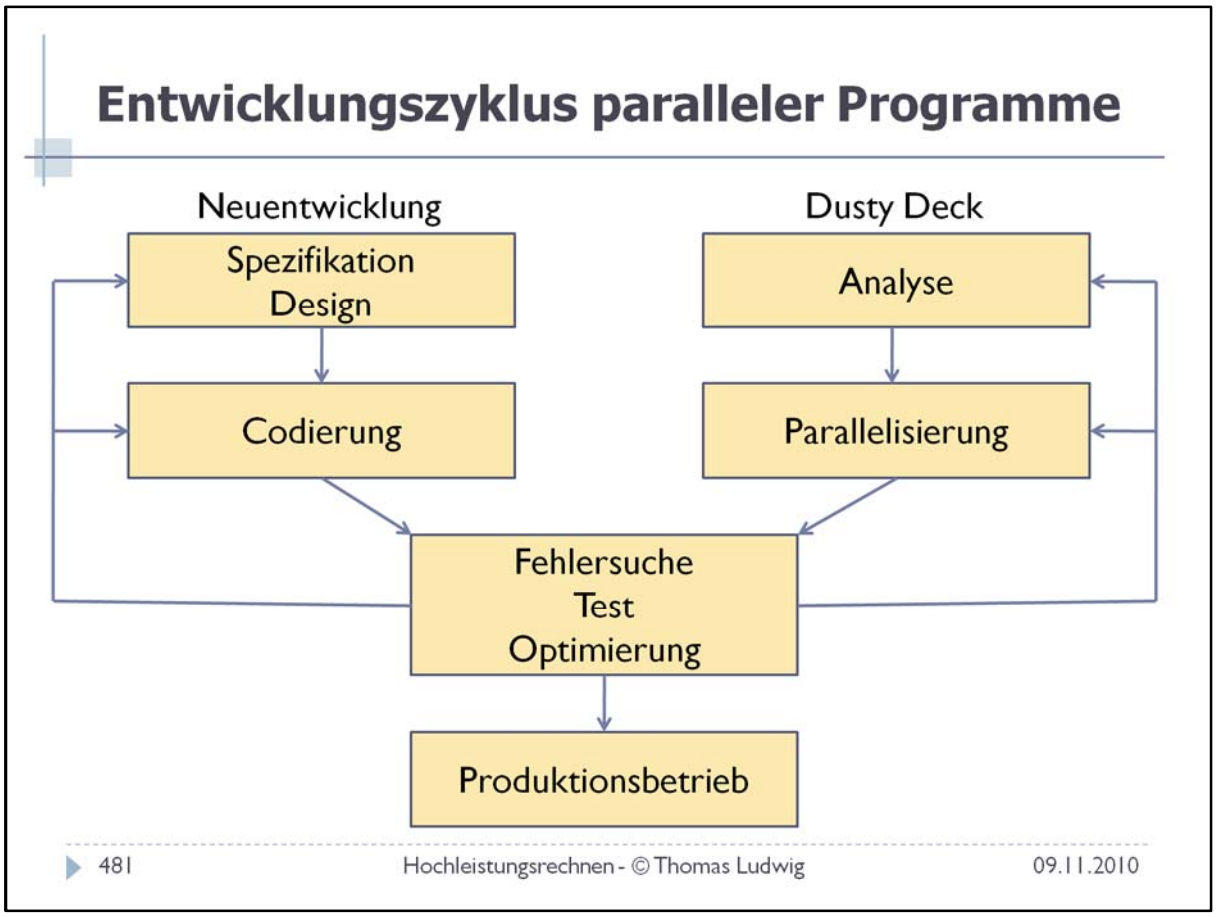
Fehlersuche

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

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

Fehlersuche
Die zehn wichtigsten Fragen

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



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

Als Dusty-Deck (von ´deck´, Lochkartenstapel), oder Legacy-Programme bezeichnet man solche, die schon sehr alt sind, aber immer noch verwendet werden. Die Programmierer sind längst woanders und niemand kennt sich mit dem Code aus. Bei Parallelisierungen gehören die allermeisten Projekte in diese Kategorie.

Fehlersuche (Debugging)

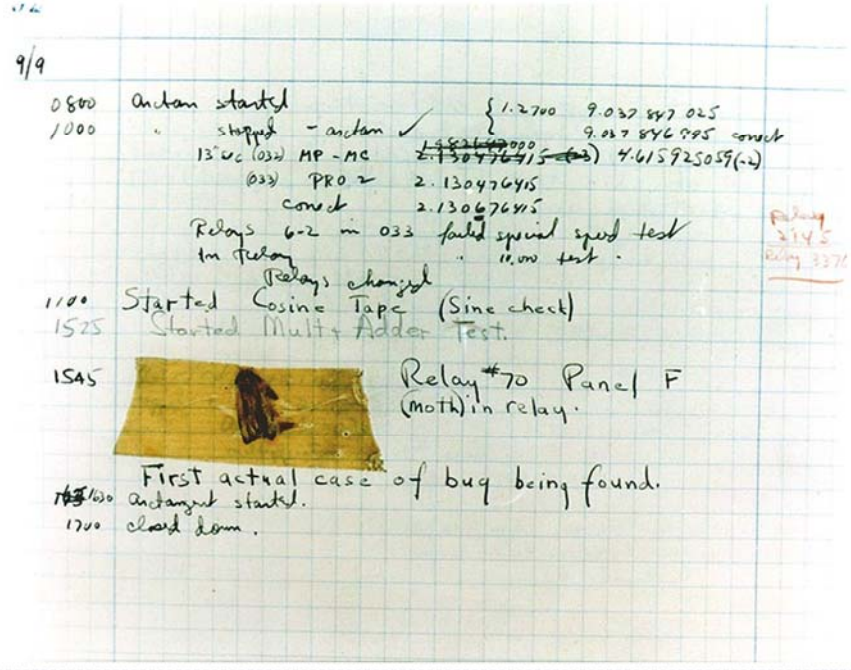
Aufspüren von Fehlerzuständen und die Beseitigung ihrer Ursachen

4 Schritte

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

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

Debugging?



Quelle: http://en.wikipedia.org/wiki/File:H96566k.jpg#filelinks

The First "Computer Bug" Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program".

Beispiel

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

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

Häufigste Fehlerquellen

Sequentielle Programmierung

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

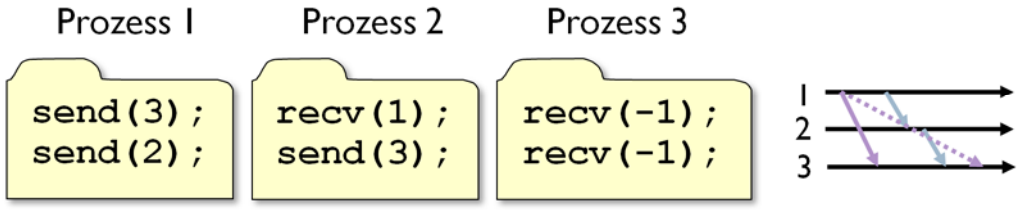
Parallele Programme

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

Häufigste Fehlerquellen: Überholvorgänge

Definition: Ein Überholvorgang entsteht durch unsynchronisierte, modifizierende Zugriffe auf gemeinsame Objekte (Adreßbereiche, Nachrichten-puffer)

Beispiel:



Konsequenz: Nichtdeterminismus, Nichtreproduzierbarkeit (schwer feststellbar)

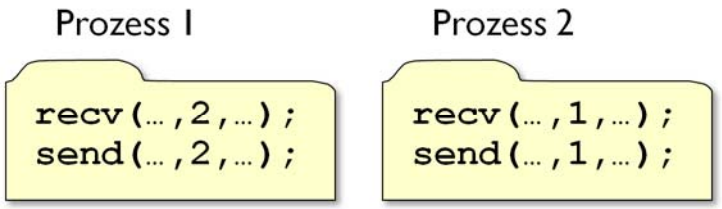
Siehe: http://en.wikipedia.org/wiki/Race_condition

Prozeß 3 empfängt von beliebigen Sender.

Häufigste Fehlerquellen: Verklemmungen

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

Beispiel:



Konsequenz: Programm bleibt hängen (leicht feststellbar)

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

Kann bei parallelen Programmen leicht passieren, da wir oft nur einen Quellcode verwenden. Der von Prozeß 1 und Prozeß 2 ist also sowieso identisch. Man sieht es aber dem Code nicht so leicht an, da man ihn ja nur einmal vor sich liegen hat und außerdem die Aktualparameter meist Variablen sind.

Problemstellungen

Zusätzlich zu den normalen Problemen der Fehlersuche

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

Erkennen einer Fehlerwirkung

Normalerweise

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

Bei parallelen Programmen

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

=> Falsche Berechnungen schwer erkennbar

Werkzeugunterstützung

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

Statische Analyse

Analyse des Programmtextes vor/zur Übersetzungszeit

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

  = Forschungsthemen (bisher ungelöst)

printf() und WRITE

Die Werkzeuge zur Fehlersuche schlechthin!

Bei parallelen Programmen aber:

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

Laufzeit-Werkzeuge

Automatische Fehlerprüfung zur Laufzeit

Sequentielle Aspekte

- Dynamische Speicherverwaltung

Parallele Aspekte

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

Vorbereitung der Anwendung (Alternativen)

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

Spurbasierte Werkzeuge

Merkmale

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

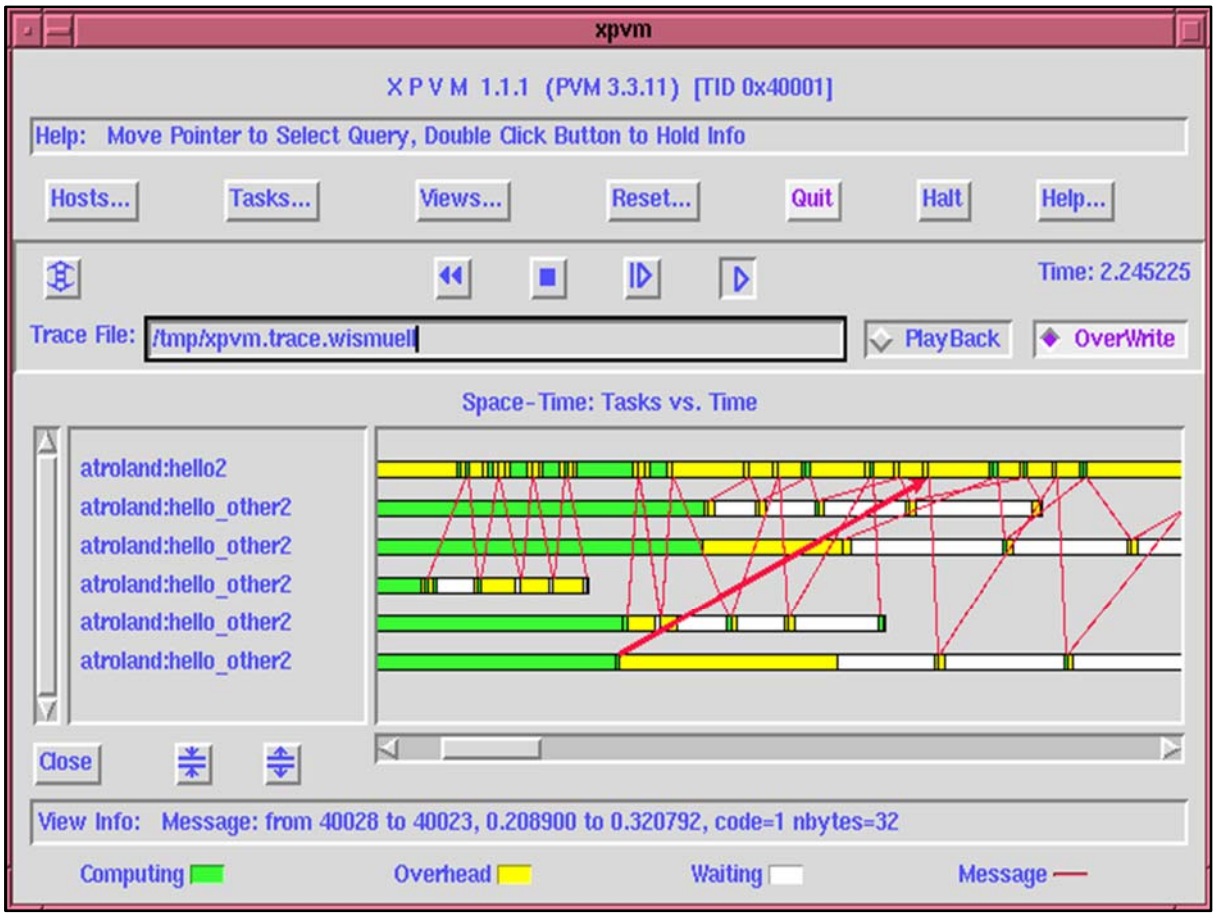
Aufgezeichnete Ereignisse

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

Spurbasierte Werkzeuge...

Darstellungsarten

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



Beispiel: XPVM. In seiner Darstellung sehr typisch für viele andere spurbasierte Werkzeuge.

Spurbasierte Werkzeuge...

Steuerung

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

Vorbereitung

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

Bewertung

- Für globalen Überblick und zur Überwachung der Kommunikation

Haltepunktbasierte Debugger

Vorgehen

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

Bei erkanntem Fehler

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

Haltepunktbasierte Debugger...

Typischer Funktionsumfang

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

Konzepte paralleler Debugger

Eigenschaften paralleler Programme

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

Berücksichtigung dieser Eigenschaften sehr unterschiedlich

Kein Standard auf dem Gebiet in Sicht

Es gibt zur Zeit zwei gebräuchliche parallele Debugger:
- The Distributed Debugging Tool DDT der Firma Allinea
- Totalview der Firma Rogue Wave.

Umgang mit mehreren Prozessen

Zwei Methoden:

Fenstertechnik und Prozeßmengen

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

DETOP: Debugging Tool für Parallel Programs, Eigenentwicklung an der TU München.

Skalierbarkeit und Dynamik

Problem: Umgang mit höheren Prozeßanzahlen

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

Problem: Debugging dynamisch generierter Prozesse

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

Interaktion

Überwachung von Kommunikation und Synchronisation

- Möglich durch Haltepunkte auf Bibliotheksfunktionen
- Meist keine weitergehende Unterstützung, wie z.B.
  - Ausgabe wartender Prozesse
  - Status von Nachrichtenwarteschlangen
  - Haltepunkte auf Nachrichten
- Ausweg

  Gleichzeitige Benutzung spurbasierter Werkzeuge (i.a. nur lesender Zugriff) und von Spezialwerkzeugen (message queue manager, mqm)

Verteiltheit

Daten der Anwendung sind verteilt

- Spezialwerkzeuge liefern globale Sicht

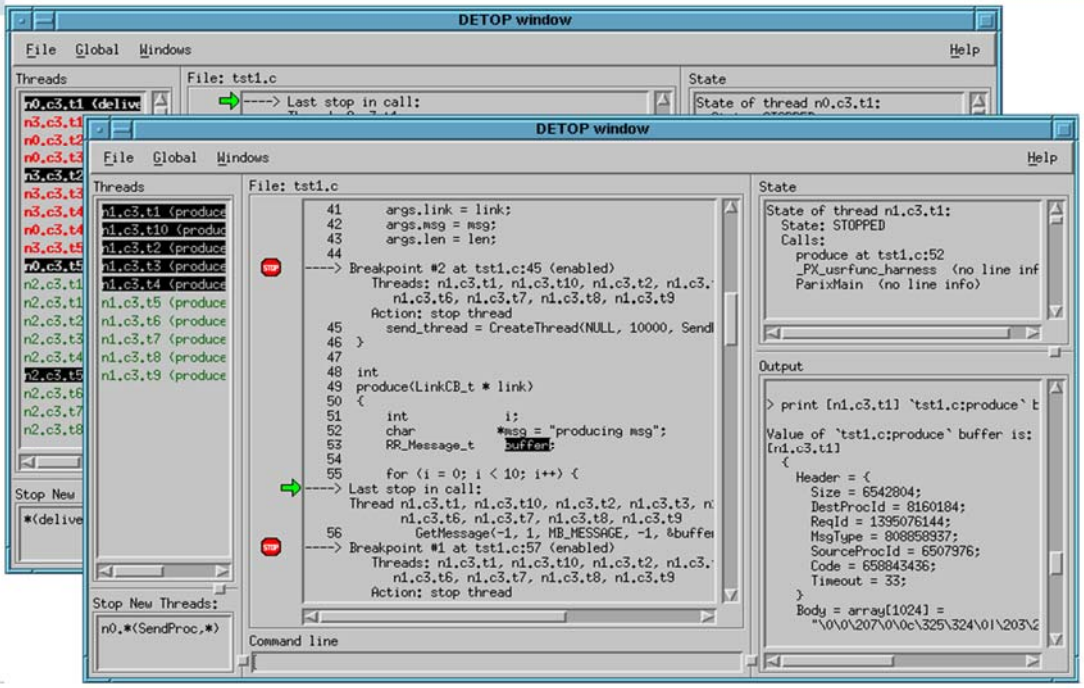
Gemeinsame Daten verteilter Prozesse

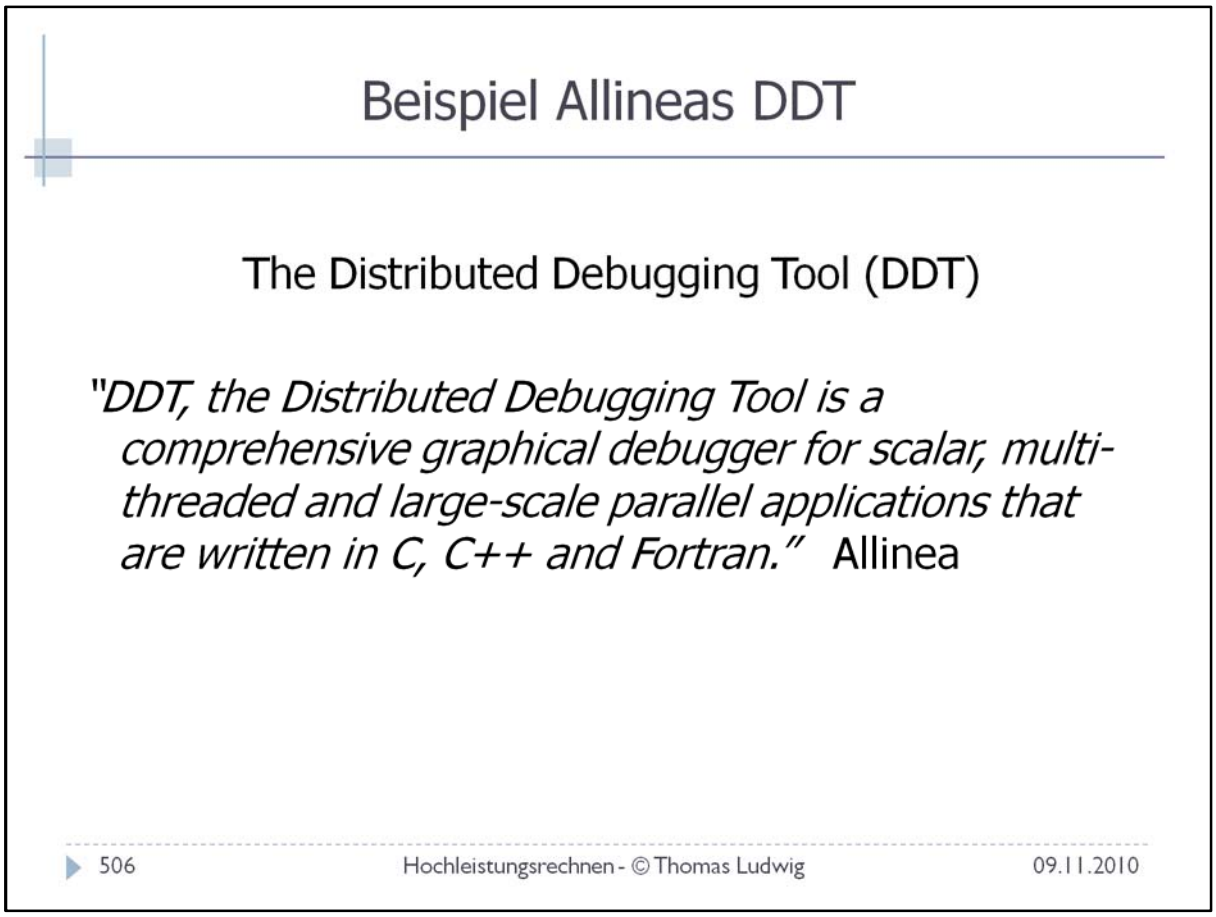
- Beispiele: MPI-Gruppen, gemeinsame Speichersegmente
- Problem: Einfrieren des Zustandes bei Erreichen des Haltepunktes
- Meist nur Anhalten eines Prozesses unterstützt
- Wenn globales Anhalten unterstützt, dann nie sofort(!)
  => Zustandveränderungen sind möglich

Globale Ereigniserkennung (Forschungsthema)

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

Beispiel DETOP





Siehe: http://www.allinea.com/index.php?page=48

“For multi-threaded or OpenMP development DDT allows threads to be controlled individually and collectively, with advanced capabilities to examine data across threads.

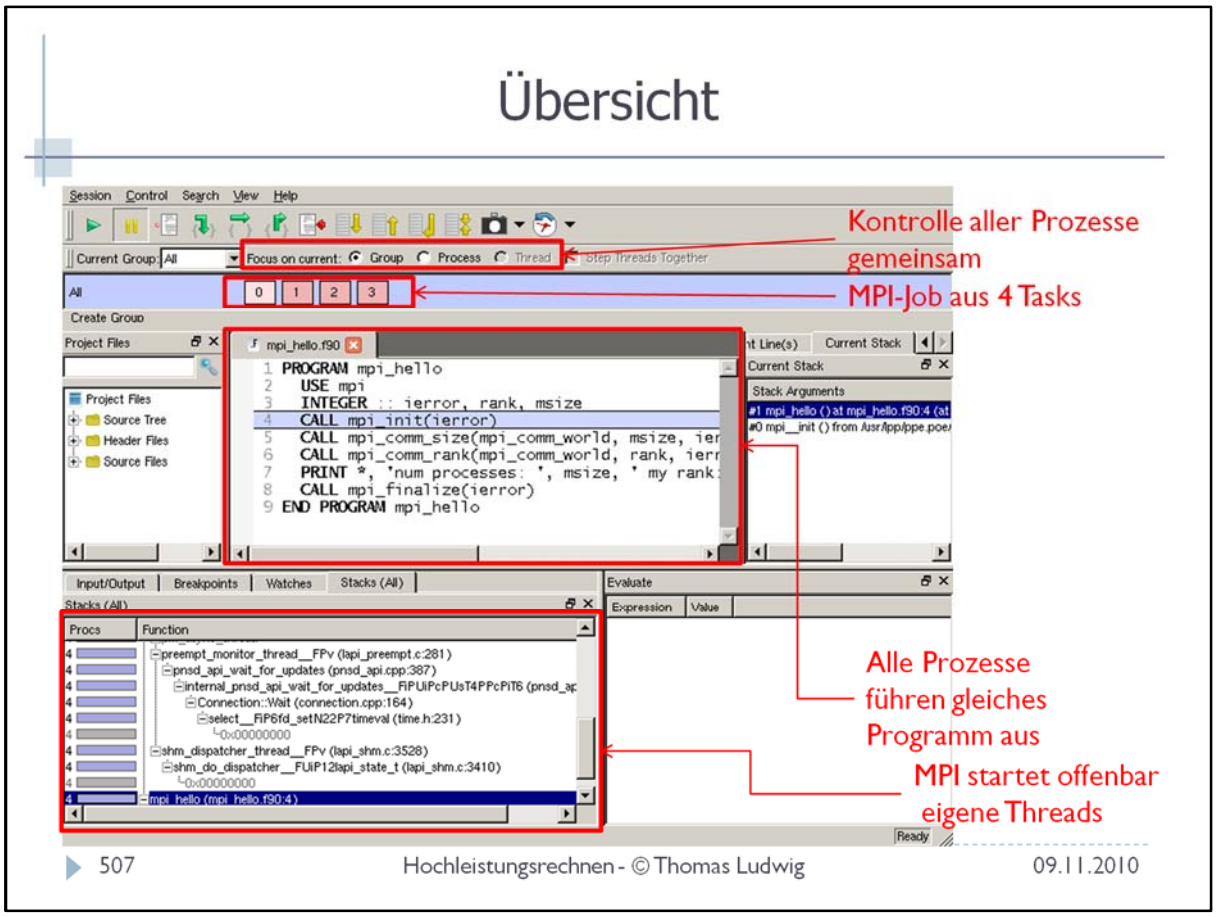
The Parallel Stack Viewer is a unique way to see the program state of all processes and threads at a glance - and developers can easily spot rogue processes or threads, and even define new control groups from it, meaning massively parallel programs are easy to manage.

DDT's interface scales amazingly - providing the same clarity of information at thousands of processes as at a handful - highlighting commonality and differences with summary views and data comparison to focus your attention.

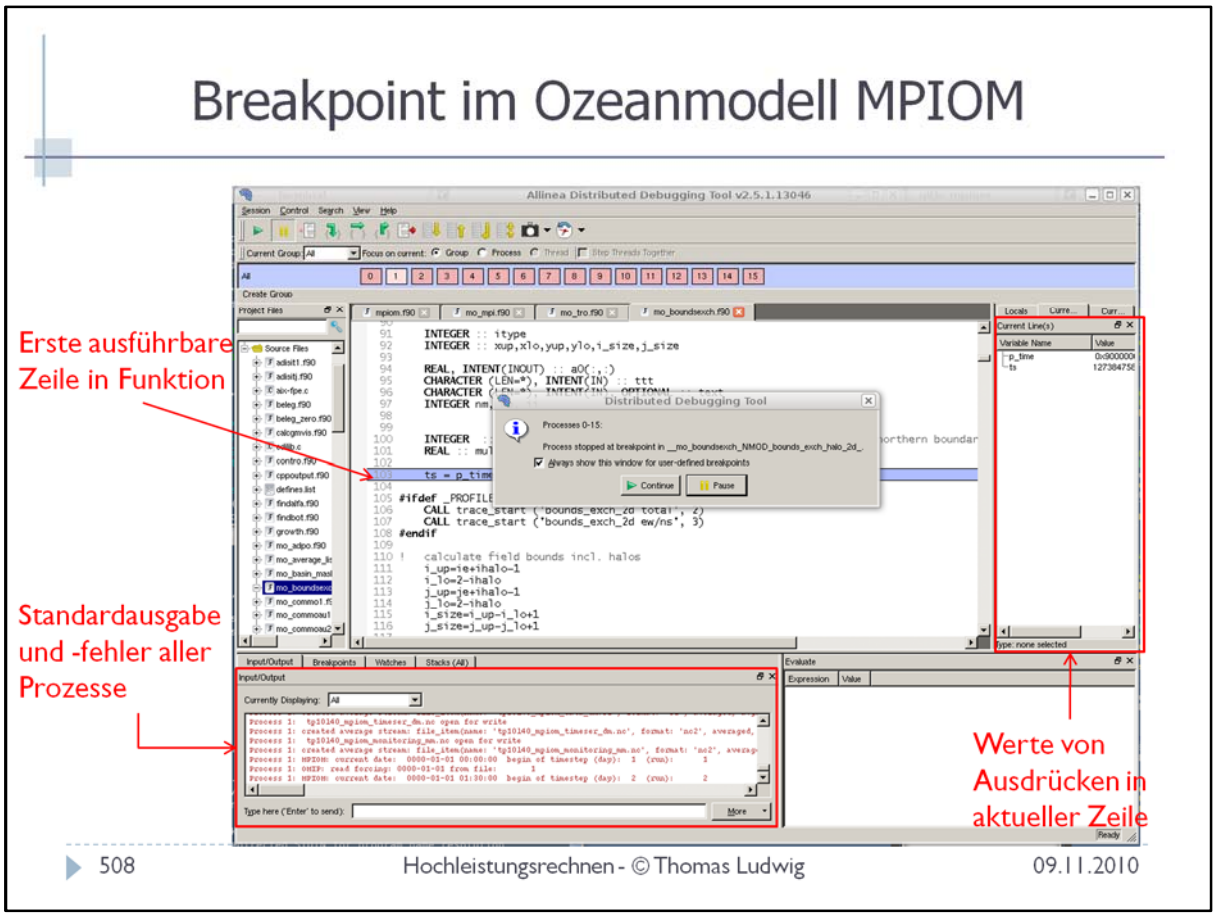
DDT is proven at scale on the most powerful systems - including debugging applications at over 200,000 cores simultaneously.

DDT's advanced memory debugging capability brings tremendous benefits to developers of scalar and parallel applications. DDT can find memory leaks, and detect common memory usage errors before your program crashes.

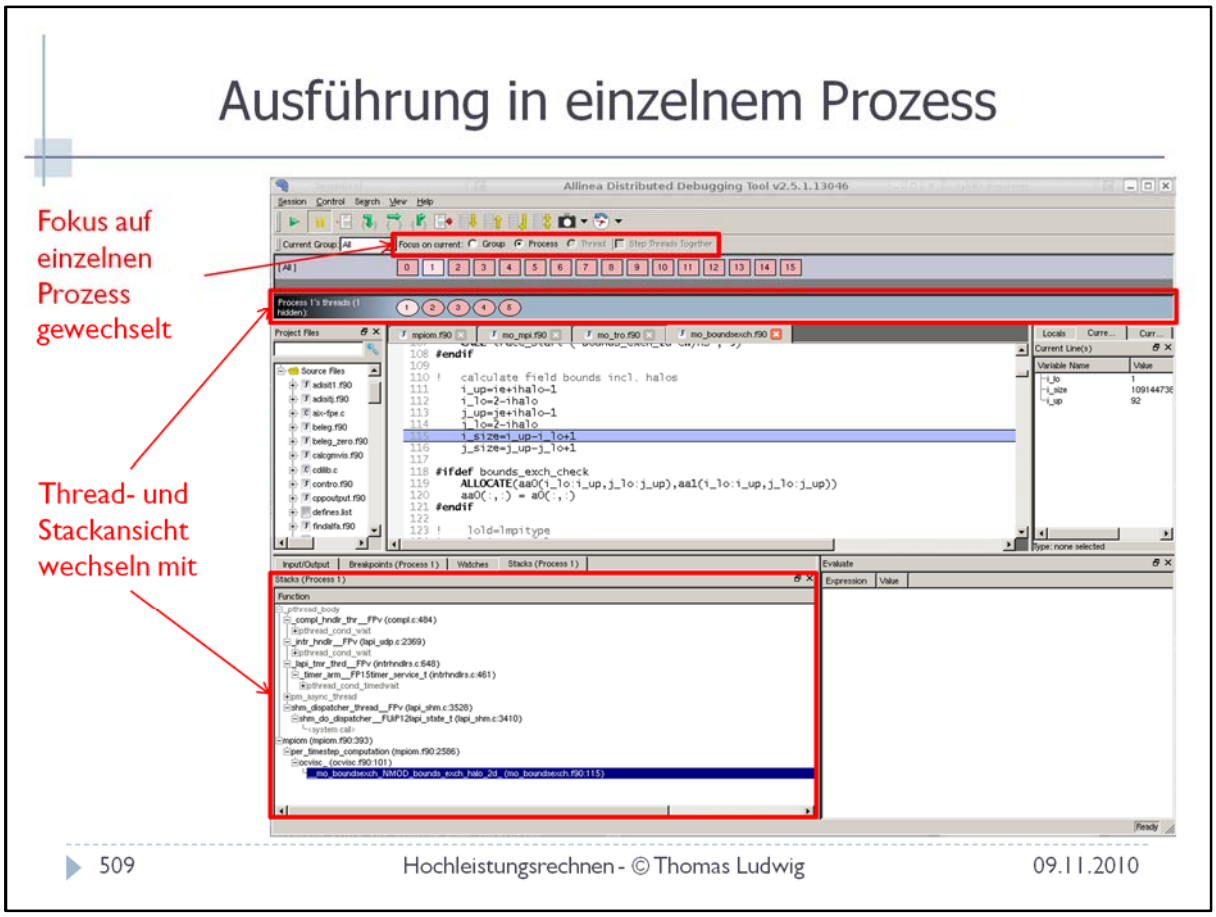
With DDT, you can check a pointer is valid or find the stack when it was allocated - a fantastic boost for any developer. Reading or writing beyond the ends of allocated data can also be detected - instantly.”



Erklärungsbedürftig sind noch: einzelne Kontrollelemente in zweiter Menüleiste (v.l.): Fortfahren, Unterbrechen, Breakpoint setzen, Step-In, Step-Over, Until

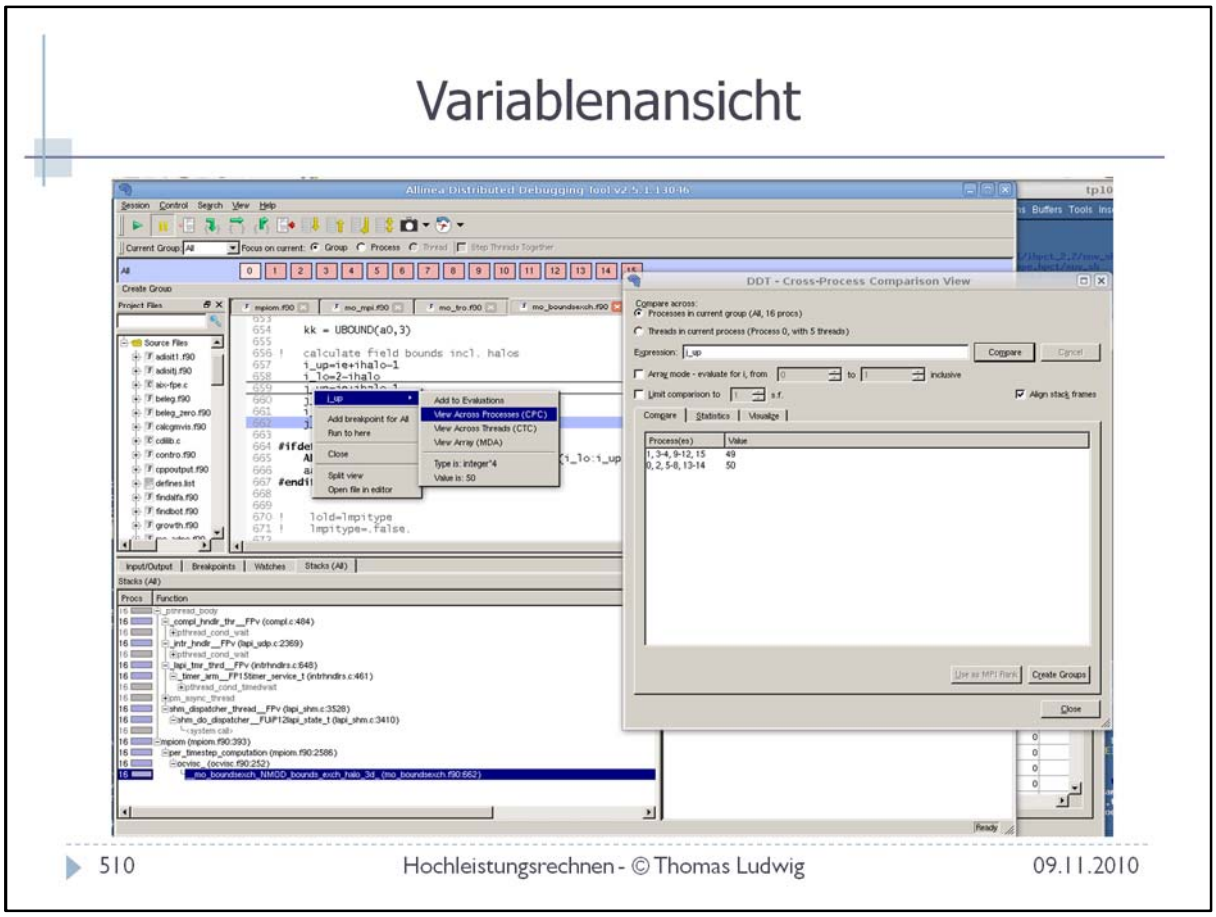


Gezeigt ist ein Lauf von MPIOM mit 16 Prozessen.
Gestoppt wurde durch einen Breakpoint auf die Kommunikationsfunktion boundsexch_halo_2d, es wird die erste ausführbare Zeile fokussiert, die erste Zeile der Funktionsdeklaration ist Fortran-üblich bereits viele Zeilen vorher und deshalb nicht im Bild.
Wie am Dialog in der Bildmitte zu erkennen wartet der Debugger typischerweise einen kurzen Moment bis alle Prozesse in der fokussierten Gruppe am Breakpoint ankommen.



Nennenswert:

- Stackansicht wechselt auf den ausgewählten Prozess
- Fortlaufende Aktualisierung der Ausdrücke in aktueller Zeile
- Automatische Anzeige der auswählbaren Threads unter Prozessen in der Gruppe



Die automatische Variable j_up hat auf mehreren Tasks verschiedene Werte die sich in der Vergleichsansicht aufzeigen lassen.
Komplexere Ansichten von Feldvariablen sind möglich (zeitaufwendig).

Deterministische Ablaufkontrolle

Problem des Nichtdeterminismus

- Erzwingen einer deterministischen Abarbeitungsreihenfolge der Kommunikation

  Programm dadurch evtl. verlangsamt

  Varianten der Reihenfolgen systematisch testbar

  (Bei sequentiellen Programmen kein Problem!)

Deterministische Ablaufkontrolle...

Funktionsweise eines Werkzeugs hierzu:

- Erster Programmlauf

  Aufzeichnen der Reihenfolge des Eintreffens von Nachrichten bei Empfängern

- Weitere Programmläufe (*deterministic replay*)

  Verwendung der Informationen aus dem ersten Programmlauf

  Die Reihenfolge, wie Nachrichten beim Empfänger ankommen, wird gesteuert: zu früh eintreffende werden zurückgestellt

Sicherungspunkte

Problem der Zykluszeit

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

Vorgehensweise

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

Fehlersuche
Zusammenfassung

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