Leistungsmodellierung und Analyse

Leistungsmodellierung und Analyse

- Amdahls Gesetz + Speedup
- Ursache von Leistungsverlusten
- Hardware Charakteristika
- Programmanalyse + Optimierung
- Engpässe

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?

Amdahls Gesetz

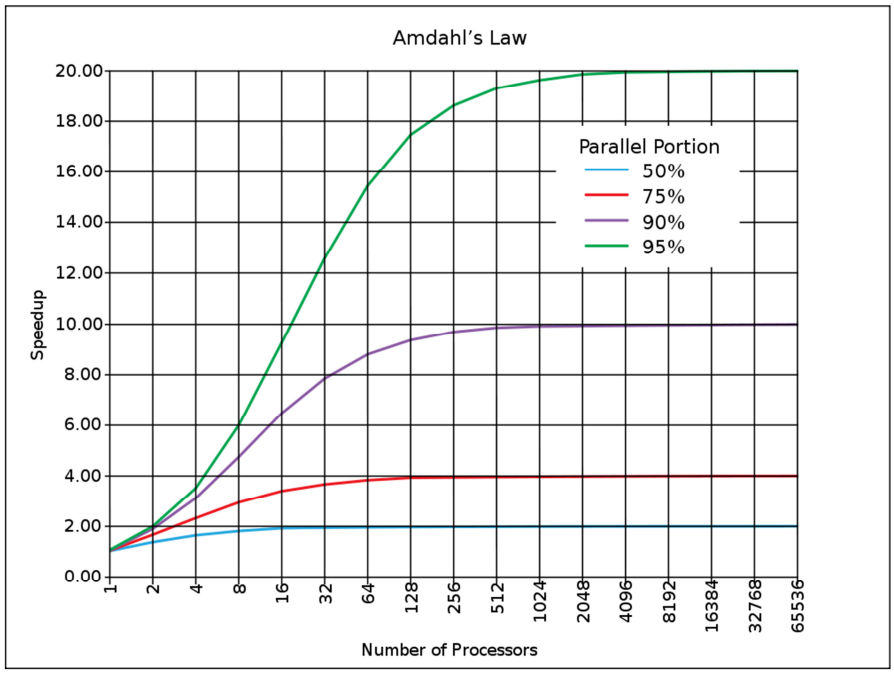
- Ausgangspunkt: Jedes Programm enthält einen Bruchteil f an Operationen, die nur sequentiell ausgeführt werden können
- Es gilt daher für den Speedup

  $S <= 1/(f+(1-f)/p) \Rightarrow S_{max} <= 1/f$
- Beispiel: $f=0.01 \Rightarrow S_{max}=100$
- Praktische Erfahrung

  Sequentieller Anteil meist sehr gering
- Bewertung: Amdahls Gesetz gilt! Man muß versuchen, den sequentiellen Anteil klein zu halten

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

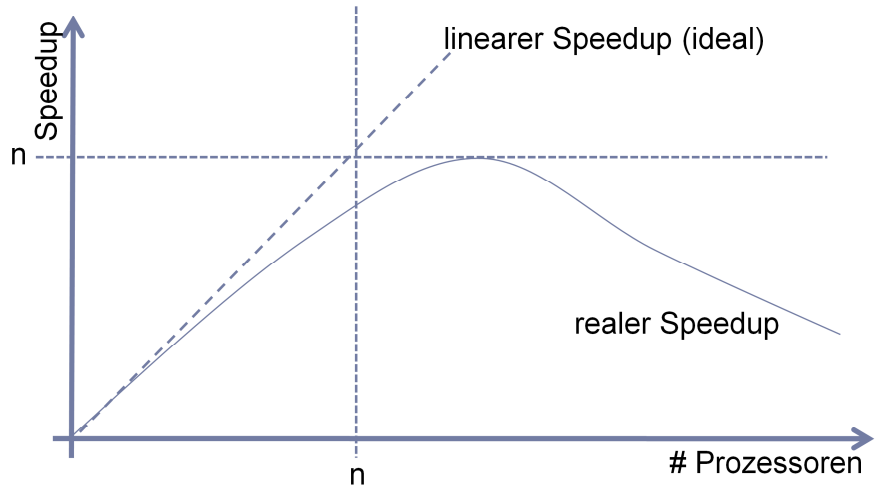
Amdahls Gesetz...



Quelle: Wikimedia Commons (http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg)

Speedup-Bestimmung

Verwendet in wissenschaftlichen Artikeln, in denen die Verfahren parallelisiert wurden



Ursachen von Leistungsverlusten

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

Ursachen von Leistungsverlusten...

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

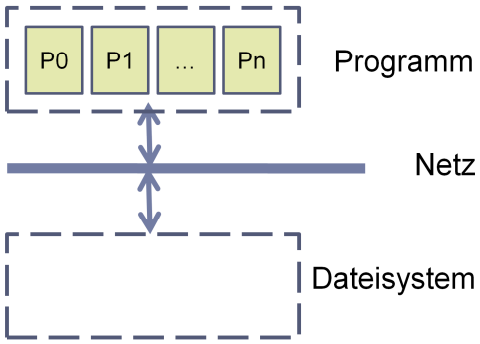
Ursache von trügerischen Leistungsgewinnen

- Geänderte Ressourcennutzung
  - Bei fester Problemgröße und steigender Anzahl Prozessoren: relevante Daten- und Code-Anteile passen auf einmal wieder in den Cache

    => deutlich schnellere Abarbeitung

Möglicherweise entsteht hier superlinearer Speedup. Das ist dann leicht zu erkennen. Aber meistens wird einfach nur die Kurve nach oben gedrückt, was sich eigentlich erstmal gar nicht erkennen läßt. Es sieht dann womöglich aus wie guter Speedup.

Logische (Prozess)-Sicht



- Batch-Scheduling:
  - Nutzer fordert N Knoten und P Prozessoren an
- Programm-Sicht:
  - Verteilung auf physikalische Geräte unbekannt
- Ausnutzung der Ressourcen ist wichtig
  - Bestmögliche Verteilung?
  - Aufdeckung eines Engpasses?

  ⇒ Leistungsmodell nötig!

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

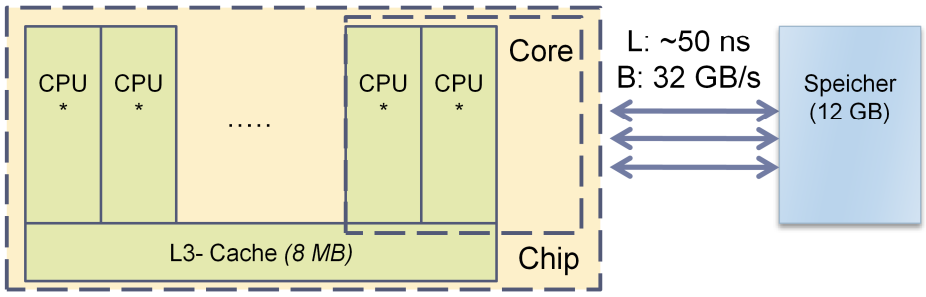
- Einzelne Komponenten sehr komplex
- Ohne Kenntnis des Systems suboptimale Leistung
  - Aber bis zu welchem Detailgrad brauchen wir es?
- Daher hier einfaches Modell für Kerncharakteristika
  - Modell bildet Realität nicht exakt ab
  - Aber hinreichend gut um Resultate zu bewerten
  - Ausblick auf komplexere 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?
- Vergleich mit bestehenden Systemen

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

Physikalische Sicht – Prozessor (mit SMT)



CPU kann pro Takt z. B. 2 FLOP ausführen, entsprechend Rechenwerken!

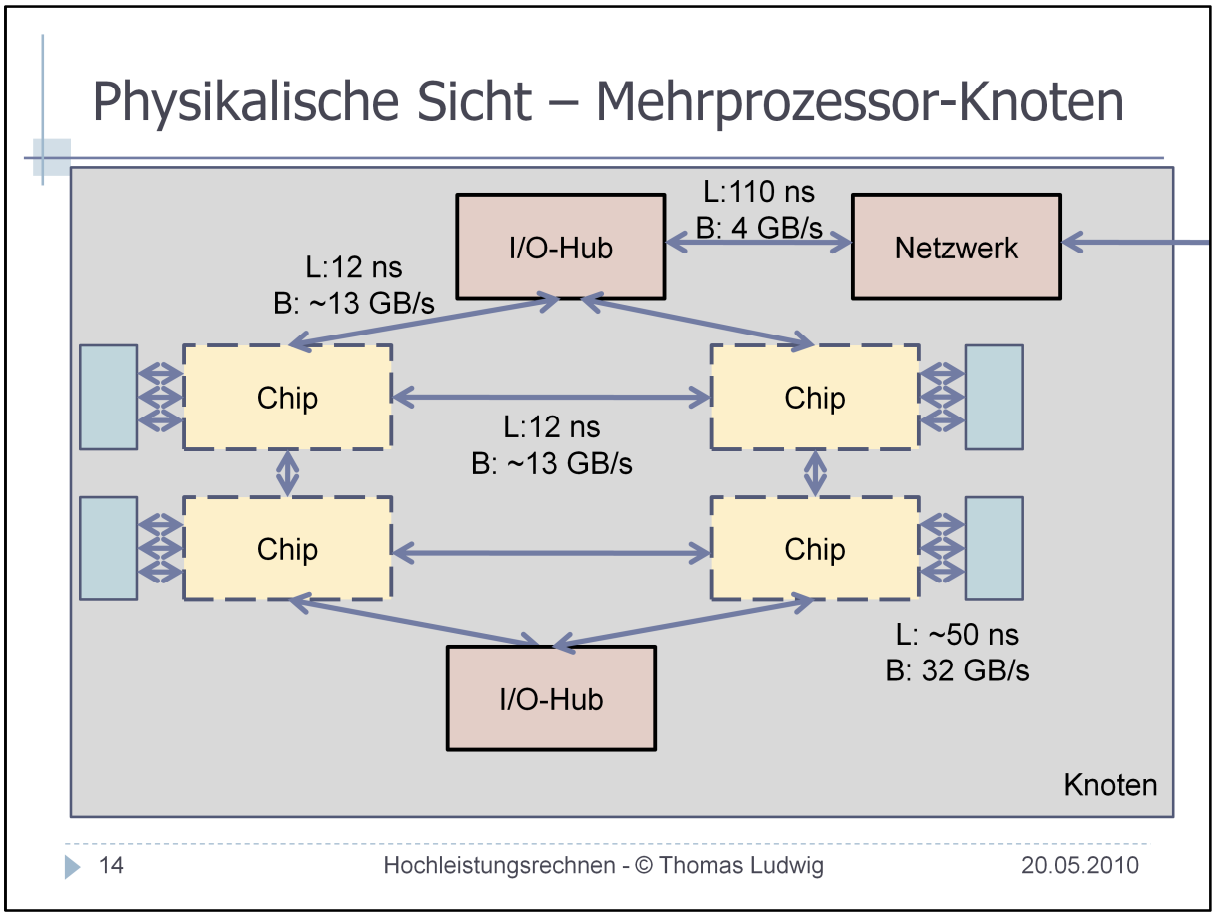
Pro Core:	*Latenz:*
L1: 32K Instruktion/32K Daten	*4 Zyklen*
L2: 256K	*10 Zyklen*

Eine Abschätzung der Leistungsfähigkeit des Codes ist schwierig. Compiler haben ebenfalls einen Einfluss auf Leistungsfähigkeit.

Mit SMT; normalerweise pro Core nur eine CPU. Eigentlich eine CPU / Core. Tatsächlich gibt es aber mehrere Verarbeitungseinheiten pro Prozessor, so dass zwei Threads effizient laufen können.
Bspw. können in der Nehalem-Architektur pro Takt 1 Load, 1 Store Data, 1 Store (von Adresse) und 3 Verarbeitungen stattfinden, allerdings oftmals verschiedene. FP Add, FP Multiply z. B. können parallel betrieben werden.

Speicheranbindung (exemplarisch), 50 ns (bei 3 GHz) == 150 Zyklen

Pro Nehalem 4 Cores
Chip / Sockel oftmals synonym



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.

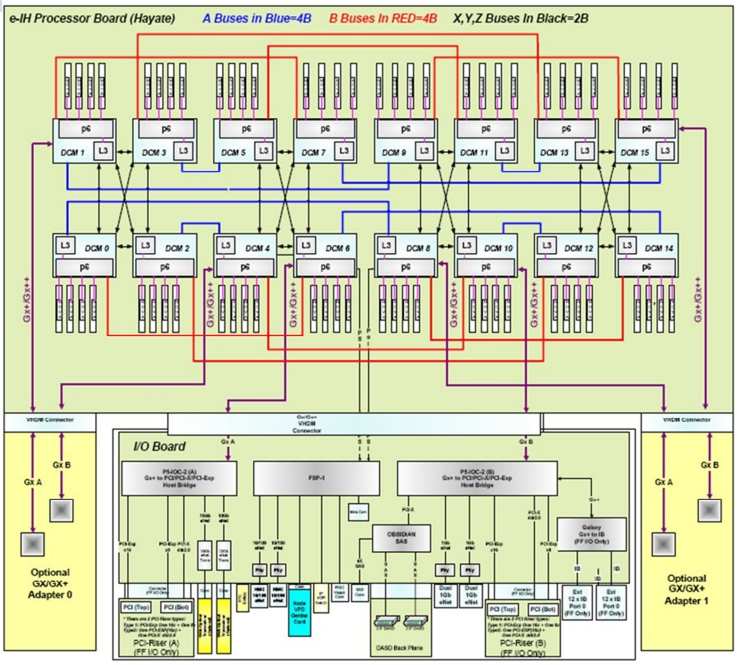
http://www.notur.no/notur2009/files/semin.pdf

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

NUMA-Charakteristik
- Speicherorganisation beachten
- Prozessmigration zwischen CPUs - CPU-Pinning

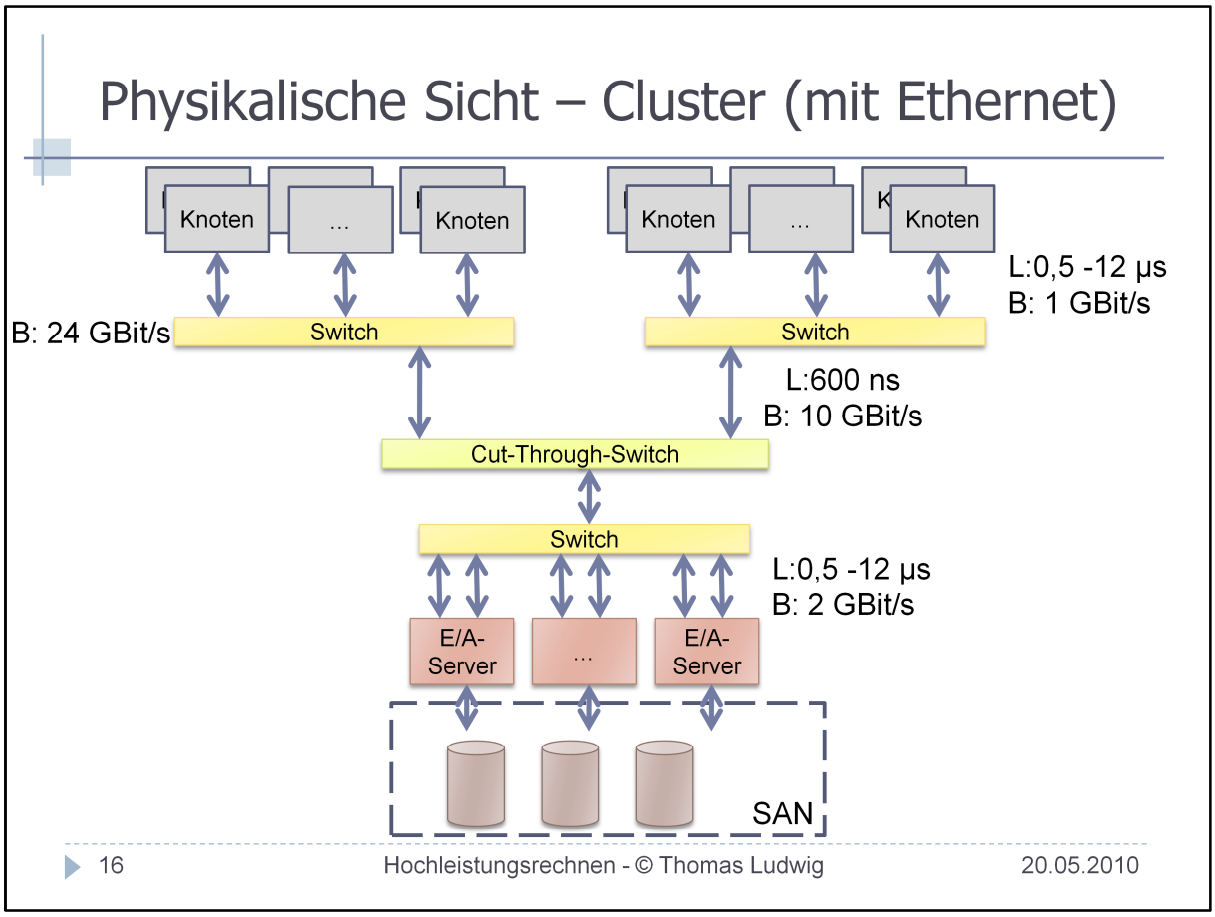
Netzwerkeffizienz – oberen beiden Chips benutzen
Caches verschiedener Größe und Latenzen
Netzwerk kann u. U. nur von mehreren Prozessoren saturiert werden

Praxisbeispiel: IBM-Power-6-Server des DKRZ



16 Node-Cards a 2 Cores Power 6 mit 4.7 GHz

256 Knoten Infiniband-Vernetzung



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 erstmal in einem Puffer gespeichert, dann der Zielport bestimmt und das Paket weitergeschickt.
Die Zeit im Switch wird hier mal 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.

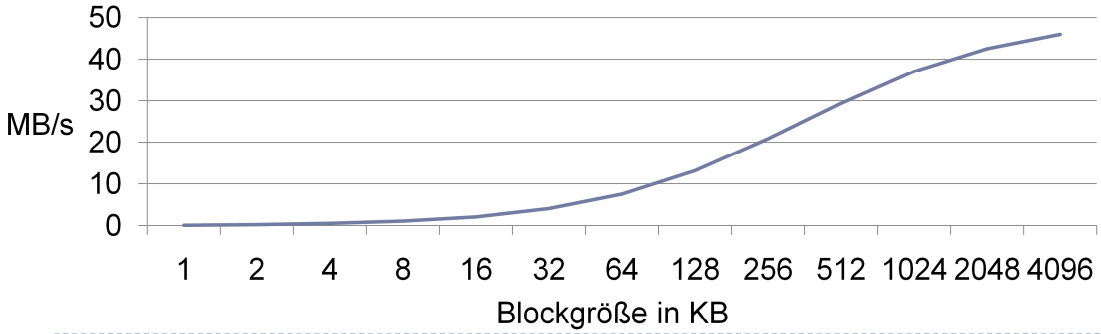
E/A-Leistung

- 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!

Prozessplatzierung

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

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:

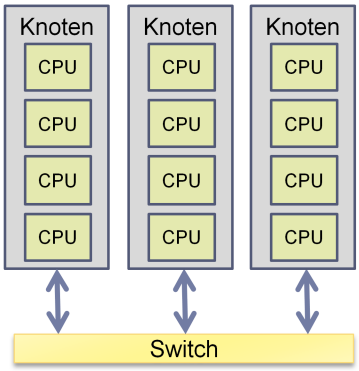
Fakten:
- Datenaufteilung
- 9 Prozesse

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:

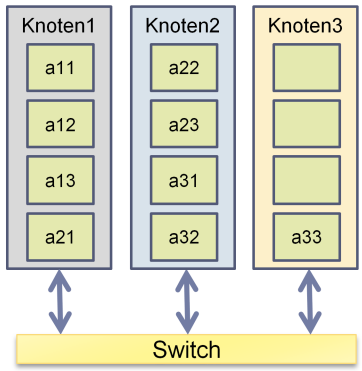


In dem Beispiel nehmen wir an, dass die Daten der Matrix räumlich so partitioniert sind wie dargestellt.
Als Beispiel sei die Aufteilung eines 2D-Gebietes, z. B. Landstrich in dem sich Objekte bewegen. Über die Grenzen der Gebiete muss Information ausgetauscht werden, bspw. Objekte die zwischen den Objekten 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.

Platzierung 1:



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

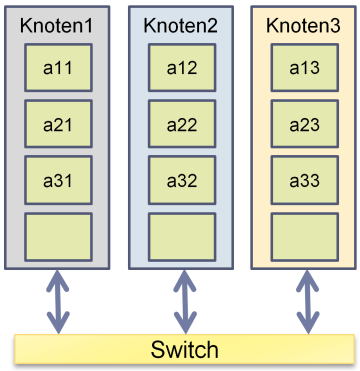
Beobachtungen:
- Knoten 3 ist nicht voll ausgelastet
- Kommunikationslast:
  - Innerhalb der Knoten:
    - Knoten 1 – 3 Grenzen
    - Knoten 2 – 3 Grenzen
    - Knoten 3 – 0 Grenzen
  - Zwischen Knoten:
    - Knoten1 ⇔ Knoten2 – 4 Grenzen
    - Knoten2 ⇔ Knoten3 – 2 Grenzen
  - Knoten 2 an 6 Kommunikationen beteiligt
- Aufteilung der Berechnung
  - Im Falle von Multicore?
  - Im Fall von SMT? Suboptimal!

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?

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

23 Hochleistungsrechnen - © Thomas Ludwig 20.05.2010

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.

Klassifikation des Engpasses

- Wie groß ist der Einfluss der Rechenzeit, Speicher, E/A ?
- Bezogen auf konkrete Hardware!
- Hilfsmittel um Analyse fortsetzen zu können
- Wir betrachten Abschnitte der Aktivität über die Zeit

- Klassen:
  - Rechenintensiv (CPU-bound)
  - Speicherintensiv (memory-bound)
  - Kommunikationsintensiv (network-bound)
  - E/A-intensiv (I/O-bound)

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!

Rechen-/Speicherintensive Programme

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

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?

E/A-Intensiv

- Analyse sehr komplex!
- Annahme: Paralleles Dateisystem
- Client- und Server-E/A-Aktivität erfassen

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

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.

Relevante Leistungscharakteristika

- Prozessorleistungsfähigkeit
  - Instruktionen / Sekunde, Simultaneous Multithreading (SMT)?
  - Größe der 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

Zusammenfassung

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

Parallele Ein/Ausgabe

Wozu parallele E/A in MPI?

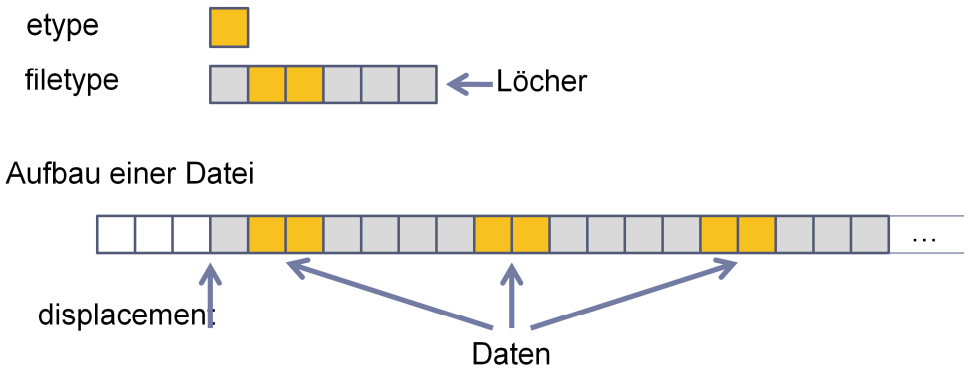
- Leistungsgewinn
  - Z.B. durch kollektive Aufrufe
  - Z.B. durch asynchrone E/A
- Einfacherer Zugriff durch Problemanpassung
  - Z.B. abgeleitete Datentypen bei irregulären Daten
  - Dadurch auch Portabilität in heterogenen Umgebungen

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...

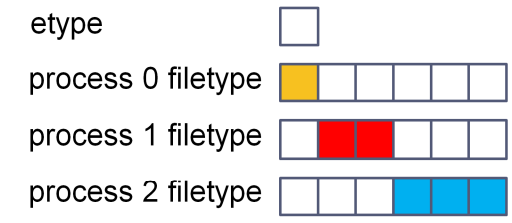
- filetype (Dateityp)
  - Schablone, nach der eine Datei aufgebaut wird
  - Besteht aus elementaren Datentypen und gleichgroßen Löchern



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`

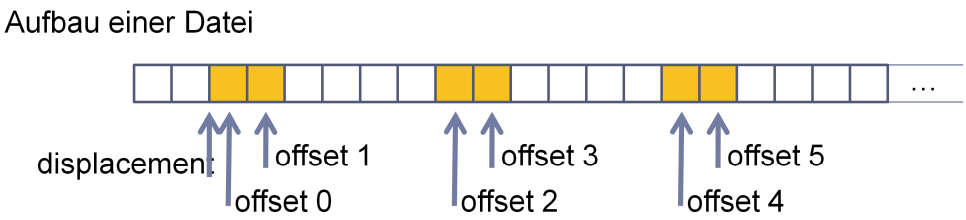
Nichtzusammenhängende Zugriffe: Beispiel

```
/* 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



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