

# 1 Konzepte der Parallelverarbeitung

- Erhöhung der Rechenleistung
  - verbesserte Prozessorarchitekturen mit immer höheren Taktraten
  - Vektorrechner
  - Multiprozessorsysteme  
(Rechner mit wenigen Prozessoren)
  - Parallelrechner  
(*massively parallel processors*, MPP; Rechner mit einigen 100 bis einigen 1000 Prozessoren)
  - vernetzte (heterogene) Rechner für verteilte Verarbeitung  
(*distributed computing*, *virtual computer*, virtueller Rechner)
    - ◆ relativ preiswert, da vorhandene Rechner genutzt werden  
(die meisten *Workstations* werden in der Nacht im Allgemeinen nicht benutzt)
    - ◆ Rechenleistung kann "beliebig" erhöht werden
    - ◆ in das Konzept können auch Vektorrechner oder Parallelrechner integriert werden
- Unterstützung des Programmierers durch parallelisierende Compiler bzw. spezielle Bibliotheken und Programme zur Verteilung der Prozesse

- Datenaustausch zwischen den parallelen Prozessen
  - über gemeinsamen Speicher (*shared memory*)
  - über Nachrichtenaustausch (*message passing*)
  
- Eigenschaften eines virtuellen Rechners
  - vereint im Allgemeinen Rechner unterschiedlicher Hersteller
  - die einzelnen Rechner haben im Allgemeinen unterschiedliche Compiler
  - sie besitzen im Allgemeinen unterschiedliche Architekturen  
(es können Rechner von PCs über *Workstations* bis zu Höchstleistungsrechnern enthalten sein ⇒ unterschiedliche Binärformate)
  - sie haben im Allgemeinen unterschiedliche Datenformate  
(*big endian*, *little endian*, Darstellung von Gleitkommazahlen, ...)
  - sie sind im Allgemeinen unterschiedlich leistungsfähig
  - sie sind im Allgemeinen unterschiedlich stark ausgelastet
  - sie sind im Allgemeinen unterschiedlich vernetzt  
(*Gigabit-Ethernet*, *Fast-Ethernet*, *Myrinet*, *Fiber Channel*, ...)

- die Leistungsfähigkeit hängt im Allgemeinen auch von der Netzlast ab
- ⇒ jeder parallele Prozess muss auf jeder Rechnerarchitektur übersetzt werden
- ⇒ der Datenaustausch muss über eine "genormte" Schnittstelle erfolgen  
(Codierung der Nachricht vom internen Format in das "Netzformat" beim Sender und Decodierung beim Empfänger)
- ⇒ der Programmierer muss dafür sorgen, dass alle Rechner des virtuellen Rechners ausgelastet werden  
(schnelle Rechner sollten nicht auf Ergebnisse langsamer Rechner warten müssen)
- ⇒ abhängig von der Grundlast einzelner Rechner und des Netzes kann die Leistungsfähigkeit eines virtuellen Rechners sehr stark variieren
- ⇒ die Grundlast kann wesentlich sein, wenn der parallele Algorithmus sensitiv gegenüber Ankunftszeiten von Nachrichten ist

- Vorteile eines virtuellen Rechners
  - da vorhandene Hardware genutzt wird, sind die Kosten einer Berechnung im Allgemeinen sehr niedrig
  - die Leistung kann optimiert werden, indem jeder Prozess der am besten geeigneten Rechnerarchitektur zugewiesen wird
  - die Heterogenität kann gezielt ausgenutzt werden
  - die Programmentwicklung erfolgt in einer vertrauten Umgebung
  - der virtuelle Rechner kann schrittweise wachsen und der jeweiligen Aufgabenstellung angepasst werden

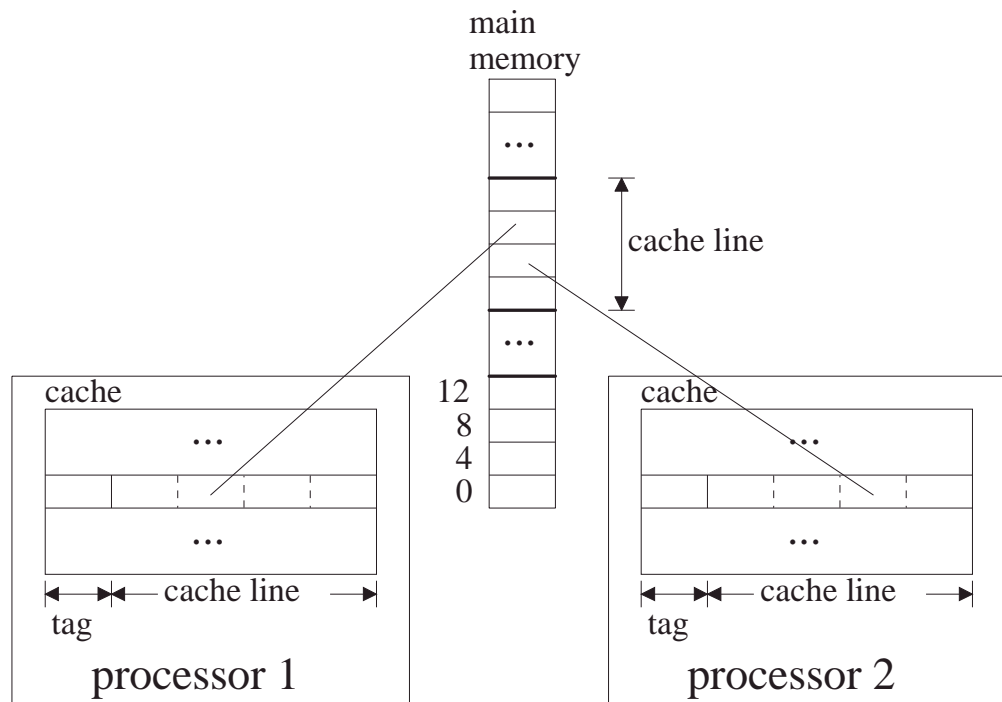
(Falls ausreichend viele gleichartige Rechner über ein schnelles Netz zu einem virtuellen Rechner verbunden werden, erhält man eine Konfiguration, die wirtschaftlich und technisch nur sehr schwer mit Höchstleistungsrechnern erbracht werden kann.

**Beispiel:** Die *Rendering-Farm* zur Erstellung des Films *Toy Story* (100 Sun Sparc 20 Doppelprozessorsysteme, 1995) bzw. *Toy Story II* (100 Sun UltraSparc E4500 mit je 14 Prozessoren, 1999). Die *Rendering-Farm* von 1999 ist ca. 20-mal leistungsfähiger als die von 1995; Communications of the ACM, Vol. 43, Nr. 1, S. 25-29, Januar 2000)

  - eine Erweiterung des virtuellen Rechners um die neueste Rechner- und Netzwerktechnik ist "einfach" möglich
  - die einzelnen Rechner des virtuellen Rechners laufen im Allgemeinen sehr stabil und sind einfach zu administrieren

- Leistungsfähigkeit eines Multiprozessorsystems ist abhängig vom
  - *Scheduling*
    - ◆ jeder Prozess/*Thread* wird nach Möglichkeit immer auf demselben Prozessor ausgeführt  
(processor affinity, processor binding)
    - ◆ *Scheduler* weist einem Prozess/*Thread* einen beliebigen freien Prozessor zu  
(*Cache*-Speicher muss jedes Mal neu gefüllt werden)
  - *Cache*-Speicher
    - ◆ gemeinsamer L2-*Cache*
      - kann zum *Streit* beim Speicherinhalt führen
      - führt im Allgemeinen zu einem Leistungsverlust, wenn unabhängige, sequentielle Programme parallel ausgeführt werden  
(*Cache* enthält immer die falschen Daten)
      - führt u. U. zu einer Leistungssteigerung, wenn ein paralleles Programm ausgeführt wird  
(falls mehrere Prozessoren auf denselben Bereich im gemeinsamen Speicher zugreifen, muss nur der erste Prozessor die *Cache-line* füllen und alle anderen finden die Daten bereits vor ⇒ geringe Busaktivität)
      - ein schlechtes *Scheduling* (keine Prozessor-Affinität) hat geringere Auswirkungen

◆ unabhängige L2-Cache-Speicher



- jeder Prozessor hat seinen eigenen L2-Cache
- führt im Allgemeinen zu einer Leistungssteigerung, wenn unabhängige, sequentielle Programme ausgeführt werden
- kann u. U. zu einem Leistungsverlust führen, wenn ein paralleles Programm ausgeführt wird
- führt zu häufigem Laden des Pufferspeichers, falls der *Scheduler* die Zuordnung von Prozessen/*Threads* zu Prozessoren oft ändert ⇒ starke Busaktivität

- falls ein Prozessor eine *Cache-line* ändert, müssen alle Kopien in den andern Pufferspeichern ungültig gesetzt oder aktualisiert werden  
(falls zwei oder mehr Prozessoren Änderungen in derselben *Cache-line* vornehmen, wird das Problem noch verschärft)
  - ⇒ sehr hohe *Cache*- und Busaktivität  
(bekannt als *false sharing*; dieses Problem kann auch bei einem gemeinsamen L2-*Cache* auftreten, da jeder Prozessor immer noch seinen privaten L1-*Cache* hat)
  - ⇒ die Daten sollten so angeordnet werden, dass alle Prozesse/*Threads* in verschiedenen *Cache-lines* arbeiten
- ◆ falls sich eine kleine Datenstruktur über eine *Cache-line*-Grenze erstreckt, müssen bei jedem Zugriff u. U. zwei *Cache-lines* aktualisiert werden

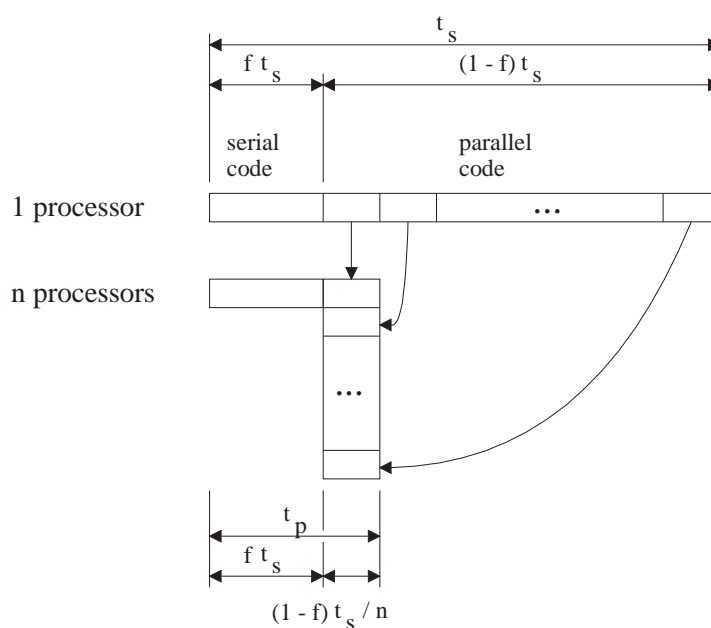
- Organisation der parallelen Prozesse
  - *Master/Slave*
    - ◆ Aufgaben des *Masters*
      - zuständig für die Initialisierung
      - erzeugt alle erforderlichen Prozesse/*Threads*
      - versorgt die *Slaves* mit Arbeit
      - sammelt ggf. alle Ergebnisse und gibt sie aus
      - synchronisiert u. U. den zeitlichen Ablauf
    - ◆ Aufgaben der *Slaves*
      - erledigen die Arbeit (z. B. Berechnungen)
      - empfangen ihre Arbeit vom *Master*  
(statisch oder dynamisch in Abhängigkeit von ihrer Leistung)
    - ◆ manchmal ist die Anzahl der parallelen Prozesse/*Threads* nicht vorhersagbar, so dass eine Leistungsminderung eintreten kann, wenn alle *Slaves* um dieselben Ressourcen "kämpfen"
    - ◆ Beispiele:
      - Internet-*Server*
      - parallele Matrix-Multiplikation

- *Workpile (work queue, Erzeuger/Verbraucher)*
  - ◆ einige Prozesse/*Threads* legen Aufträge in einer Warteschlange ab
  - ◆ andere Prozesse entnehmen der Warteschlange Aufträge und bearbeiten sie
  - ◆ im Allgemeinen gibt es eine feste Anzahl Prozesse/*Threads*
  - ◆ Auftragsspitzen werden über die Warteschlange abgefangen
  - ◆ bei zu hoher Dauerlast müssen Aufträge abgewiesen werden, damit die Warteschlange nicht zu groß wird
  - ◆ Beispiel: Berechnung einer Mandelbrotmenge
  
- baumartige Berechnung (*tree computation*)
  - ◆ Prozesse/*Threads* werden dynamisch mit dem Fortschritt der Berechnung in einer Baumstruktur erzeugt
  - ◆ besonders gut geeignet, wenn die Arbeitslast vorher nicht bekannt ist (z. B. *Branch-and-Bound* oder *Divide-and Conquer*-Algorithmen)
  - ◆ Beispiel: *Branch-and-Bound*: Problem des Handlungsreisenden, *Divide-and Conquer*: Quicksort

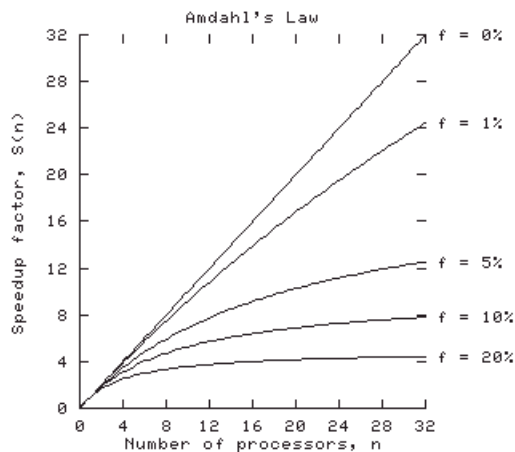
- Organisation der Arbeit (*workload allocation*)
  - Partitionierung der Daten (*data decomposition, data parallelism*)
    - ◆ identische Prozesse bearbeiten verschiedene Datenbereiche (*Single-Program-Multiple-Data, SPMD*)
    - ◆ Arbeit wird dynamisch oder statisch verteilt  
(Bei virtuellen Rechnern ist eine dynamische Zuteilung im Allgemeinen besser, damit alle Rechner trotz schwankender Grund- und Netzlasten gleichmäßig ausgelastet werden.)
    - ◆ Beispiel:
      - Berechnung einer Mandelbrotmenge
      - Addition von zwei n-dimensionalen Vektoren  
(bei p Prozessen/*Threads* führt jeder Prozess/*Thread* n/p Additionen aus)
  - Partitionierung der Funktionen (*function decomposition, function parallelism*)
    - ◆ verschiedene Prozesse/*Threads* führen verschiedene Operationen aus  
(*Multiple-Program-Single-Data, MPSD*)
    - ◆ Beispiel: *Grafik-Pipeline*

- Kombination der beiden Verfahren
  - ◆ verschiedene Prozesse führen verschiedene Operationen aus, wobei die einzelnen Operationen u. U. parallel auf verschiedenen Datenbereichen ausgeführt werden  
(*Multiple-Program-Multiple-Data*, MPMD)
  
- Granulation der Prozesse
  - grobkörnig
    - ◆ viele sequentielle Anweisungen
    - ◆ Ausführung der Anweisungen dauert sehr lange
  
  - feinkörnig
    - ◆ wenige sequentielle Anweisungen
    - ◆ Ausführung der Anweisungen erfolgt sehr schnell
  
  - Granulation wird oft als Zeit zwischen zwei Kommunikations- oder Synchronisationspunkten definiert
    - ⇒ grobkörnige Granulation reduziert die anteiligen Kosten zur Prozesserzeugung und Kommunikation
  
    - ⇒ feinkörnige Granulation erhöht die mögliche Parallelität
  
    - ⇒ in verteilten Systemen muss sichergestellt sein, dass die Kommunikationszeit nicht über die Rechenzeit dominiert

- welche Leistungssteigerung ist möglich ?
  - im Allgemeinen kann die Leistung höchstens linear gesteigert werden (superlineare Leistungssteigerung ist in Ausnahmefällen möglich)
  - einige begrenzende Faktoren
    - ◆ einige Teile des Programms können nur sequentiell ausgeführt werden  
(Initialisierung des Systems, Erzeugung paralleler Prozesse/*Threads*, ...)
    - ◆ Interprozesskommunikation kostet Zeit (z. B. Nachrichten senden)
    - ◆ Synchronisation der Prozesse/*Threads* kostet Zeit
    - ◆ u. U. sind zusätzliche Berechnungen in der parallelen Version erforderlich  
(z. B. lokale Berechnung eines Wertes in jedem Prozess)
  - **Annahme:** alle sequentiellen Programmteile sind am Anfang konzentriert und alle parallelen Teilen werden ohne zusätzlichen Verwaltungsaufwand parallel ausgeführt  
( $f$  sei der Anteil, der sequentiell ausgeführt werden muss)



- Leistungserhöhung  $S(n) = \frac{t_s}{t_p} = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1 + (n-1)f}$
- Gleichung ist als das *Amdahl'sche Gesetz* bekannt



- Begrenzung der Leistungssteigerung auf  $S(n) = \frac{1}{f}$  ( $n \rightarrow \infty$ )  
(bei 5% sequentiell und 95% parallelem Code kann die Leistung maximal auf das 20-fache gesteigert werden)
- das *Amdahl'sche Gesetz* gilt für Probleme fester Größe, deren Rechenzeit durch mehr Prozessoren reduziert werden soll
- *Gustafson* hat festgestellt, dass mehr Rechenleistung im Allgemeinen dazu genutzt wird, größere Probleme zu lösen
  - ♦ die Problemgröße wird so skaliert, dass die parallele Rechenzeit konstant bleibt
  - ♦ der sequentielle Programmanteil wächst im Allgemeinen nicht mit der Problemgröße

– *Gustafson's* skalierte Leistungserhöhung  $S_s(n)$

- ◆ **Annahme:**  $s$  bezeichne die Zeit um den sequentiellen Teil des Programms auszuführen und  $p$  die für den parallelen Teil

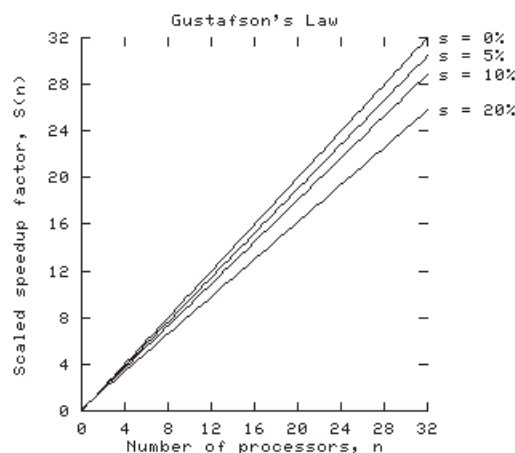
⇒ Ausführungszeit auf parallelem Rechner mit  $n$  Prozessoren:  $s + p$

(Gesamtlaufzeit ist fest; zur algebraischen Vereinfachung:  $s + p = 1$ )

⇒ Ausführungszeit auf sequentiellm Rechner:  $s + n p$

(alle parallelen Teile müssen sequentiell ausgeführt werden)

- ◆ 
$$S_s(n) = \frac{s + np}{s + p} \underset{s+p=1}{\Rightarrow} \frac{s + n(1-s)}{s + 1-s} = n + (1-n)s$$



(ein Multiprozessorsystem mit 32 Prozessoren kann bei 5% sequentiellm Code nach *Amdahl* eine 12,55-fache Leistungssteigerung erreichen und nach *Gustafson* eine 30,45-fache)

⇒ das *Gustafson'sche Gesetz* scheint den heutigen Intentionen näher zu kommen, da im Allgemeinen komplexere Probleme innerhalb einer festen Zeit gelöst werden sollen