

# PARALLELRECHNER

**Task System** = Menge an Tasks mit Abhängigkeit untereinander, lösen Problem

**Nebenläufig** = 2 Tasks nebenläufig wenn unabhängig voneinander

**Simultan, parallel** = 2 Tasks simultan / parallel ausgeführt wenn beide gestartet und noch nicht terminiert

**Parallelrechner** = Computer, der nebenläufige Tasks eines Task Systems parallel ausführt

**Parallelrechner** = Hochleistungsrechner = Höchstleistungsrechner

= Supercomputer = High Performance Computing System

- Verkürzte Ausführungszeit von Anwendungen
- Lösung von Problemen mit größerer Komplexität / feinerer Auflösung
- Potentiell bessere Fehlertoleranz
- Wissenschaftliches Interesse

Anwendungsbereiche:

- Forschung + Entwicklung:
  - große numerische Simulationsprogramme (Wetter, Klima, Bio, Chemie, ...)
  - Datenanalyse
  - verteilte Systeme
- Industrie
  - Simulationsaufgaben
  - Bildverarbeitung, Schrifterkennung
  - Banken (Börseninformationssystem)
  - Produktionsysteme (Qualitätskontrolle)
  - Datenbanken, Datenanalyse
- Parallelrechner besteht aus mehreren Verarbeitungselementen die koordiniert zusammen arbeiten

Verarbeitungselemente: zB

- gleichartige Rechenwerke
- spezialisierte Einheiten
- Prozessorknoten eines Multiprozessors
- Vollständige Rechner → virtueller Parallelrechner
- ganze Parallelrechner → gekoppelt als Metacomputer bzw. Hypercomputer

Abgrenzung

- Eingebettete Systeme = spezialisierte Parallelrechner
- Superskalaprozessoren: feinkörnige Parallelität durch Befehlspipelining
- Mikroprozessor als Hauptprozessor arbeitet gleichzeitig zu spezialisierten Einheiten z.B. GPU
- Ein-Chip-Multiprozessoren, Multi-core-Prozessoren

# Einordnung von Parallelrechnern Klassifikation nach Flynn

Charakterisierung von Rechnern als Operatoren auf 2 versch. Informationsströmen: Befehlstrom, Datenstrom

**SISD** = Single Instruction stream over a single Data stream  
→ von-Neumann-Architekturen (Einprozessorrechner)

**SIMD** = Single Instruction stream over Multiple Data streams  
→ Vektorrechner

**MISD** - Multiple instruction streams over a single Data stream  
→ Ø, Ausnahme: heterogene Systeme verarbeiten gleiche Daten, müssen zu übereinstimmenden Ergebnissen kommen

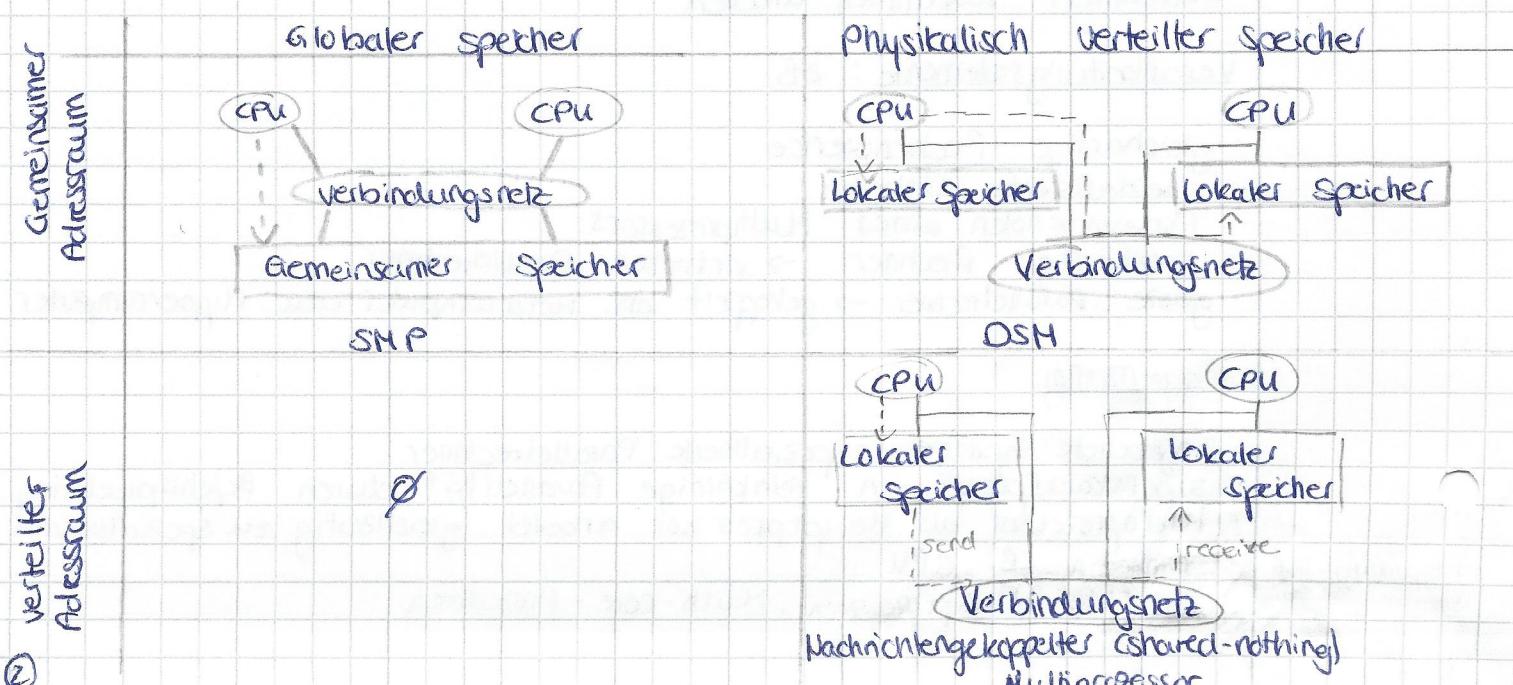
**MIMD** = Multiple instruction streams over Multiple Data streams  
→ Multiprozessorsysteme

**Vektorrechner:** Rechenwerk besteht aus mind. einer pipelineartig aufgebauten Funktionseinheit zur Verarbeitung von Arrays (Vektoren) von Gleitpunktzahlen  
↳ = Vektorpipeline

## Multiprocessorsysteme

Arten:

- **speichergekoppelt:** alle Prozessoren gleicher Adressraum, Kommunikation + Synchronisation über gemeinsame Variablen
  - symmetrischer Multiprocessor (SMP): globaler Speicher
  - distributed-shared-memory-system (DSM): gemeinsamer Adr. Raum trotz physikalisch verteilter Speichermodule
- **nachrichtengekoppelt:** alle Prozessoren besitzen nur physikalisch verteilte Speicher + processorlokale Adr. Räume Kommunikation durch Nachrichtenaustausch



## speichergekoppelte Multiprozessoren

- alle Prozessoren gemeinsamer Adressraum
- Kommunikation, Synchronisation über gemeinsame Variablen

### Uniform-memory-access-Modell UMA

- Zugriffszeit aller Prozessoren auf gemeinsamen Speicher gleich
- Jeder Prozessor kann zusätzlich lokalen Cache besitzen
- z.B. symmetrische Multiprozessoren SMP

### Non-uniform-memory-access-Modell NUMA

- Zugriffszeiten auf Speicherzellen variieren je nach Ort der Speicherzelle
  - Speichermodule des gem. Speichers physikalisch auf Prozessoren aufgeteilt
  - z.B. DSM
- CC-NUMA: (cache-coherent NUMA) Cache-Kohärenz über gesamtes System gewährleistet
- NCC-NUMA: (non-cache-coherent NUMA) Cache-Kohärenz nur innerhalb eines Knoten gewährleistet
- CO-MA: (cache-only-memory-architecture) gesamter Speicher des Rechners besteht nur aus Caches

## Nachrichtengekoppelte Systeme

- no-remote-memory-access-Modell (NORMA)
- Uniform-communication-architecture-Modell (UCA): zwischen allen Prozessoren können gleich lange Nachrichten mit einheitlicher Übertragungszeit geschickt werden
- Non-uniform-communication-architecture-Modell (NUCA): Übertragungszeit des Nachrichtentransfers zw. Prozessoren ist je nach Senden-, Empfänger-Prozessor verschieden lang

## Programmierbarkeit

### speichergekoppelt

- gemeinsame Variablen
- Synchronisation notwendig  
→ locks, mutex, ...
- Multithreading: leichtgewichtige Prozesse
- Standard: Posix Threads
- durchgängiges Konzept für Workstations, MPUs, ...

(+) leicht programmierbar

### Nachrichtengekoppelt

- expliziter Nachrichtenaustausch
- Standard: MPI, früher PVM

(+) skalieren theoretisch unbegrenzt

## Supercomputer vs. Mainframe

### Supercomputer

- auf hohe Rechenleistung getrimmt
- wissenschaftliche, ingenieurtechnische Fragestellung
- Floating point operations per second (Flop/s). Addition, Multiplikation mit größtmöglicher Präzision
- nicht kosteneffizient zur Transaktionsverarbeitung

### Mainframe

- auf Zuverlässigkeit\* getrimmt
  - Aufgaben, die durch Datenverarbeitung mit I/O-Geräten, Zuverlässigkeit und parallel Verarbeitung von Geschäftsprozessen und Transaktionen limitiert sind
  - Millions of (integer) instructions (MIPS): Daten verschieben, Werte prüfen
- \* RAS (Reliability, Availability, Serviceability), hohe Sicherheit, I/O-Infrastruktur, Backwards-Compatibility, hohe Auslastung für hohen Durchsatz

## Leistungsfähigkeit Parallelrechner

- Maßzahl: Floating Point Operations per second Flop/s
- Anzahl Flop nicht zwingend proportional zur Taktrate des Prozessors
  - a) mehrere Taktzyklen bei einzelnen Prozessortypen benötigt evtl.
  - b) tausende Operationen gleichzeitig je Takt (Vektorprozessoren)
- Heute: LINPACK - Benchmark zur Messung

K	Kilo	$10^3$	Tausend
M	Mega	$(10^3)^2 = 10^6$	Million
G	Giga	$(10^3)^3 = 10^9$	Milliarde
T	Tera	$(10^3)^4 = 10^{12}$	Billion
P	Peta	$(10^3)^5 = 10^{15}$	Billiarde
E	Exa	$(10^3)^6 = 10^{18}$	Trillion

## Top-500 Liste

500 schnellsten Supercomputer → Messung mit LINPACK - Benchmark

↳ Green 500: energieeffizientesten Supercomputer

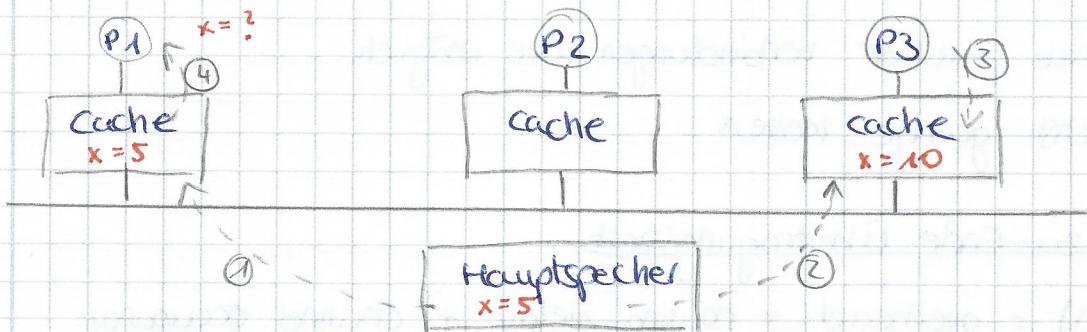
## Shared-Memory - Multiprozessoren

### Uniform Memory Architecture UMA

- UMA / symmetrische Multiprozessorsysteme (SMP) haben gemeinsamen Speicher
  - globaler phys. Adr. Raum
  - Komm./Synch. über gemeinsame Variablen
  - symmetrischer Zugriff (gleiche Latenz) zum ges. Hauptspeicher von jedem Prozessor aus
- einfache Programmierung
- dominieren Servermarkt, populär im Desktopbereich
  - Anwendungen mit großem Hauptspeicherbedarf
- nur auf SMPs: sequentielle / moderat-parallele Anwendungen mit sehr großem Hauptspeicherbedarf ausführbar
- hoher Kostenfaktor: Anbindung von Prozessoren / Speicherbänken an Bus
- durch multi-core Prozessoren: quasi alles SMP-System
- Cache-Kohärenz-Problem :-)

### Cache-Kohärenz

Replikat in den Caches versch. Prozessoren müssen kohärent sein



Cache **kohärent**: Lesezugriff liefert immer Wert des letzten Schreibzugriffs

System **konsistent**: alle Kopien eines Datums im Hauptspeicher und allen Caches identisch

System konsistent  $\Rightarrow$  kohärent

### Konsistenzmodelle

- Aussage über Ordnung der Speicherzugriffe durch parallele Prozessoren
- spezifiziert Reihenfolge, in der Speicherzugriffe eines Prozesses von anderen Prozessen gesehen werden

## Replikation

- = Halten einer / mehrerer Kopien eines Datums
  - ↳ Prozess kann auf jedes beliebige Replika zugreifen und erhält immer gleiches Ergebnis
  - ↳ Konsistenz der Kopien muss erreicht werden
- => Dilemma: bessere Skalierbarkeit + Leistung, aber Mechanismen zur Replika-Konsistenz verschlechtern Leistung
- Lösung: keine strikte Konsistenz

## Distributed-Memory-Multiprozessoren = Nachrichtengekoppelt, Shared-nothing

- keine gemeinsamen Speicher- / Adr. bereiche → lokale Speicher pro Prozessor
  - Kommunikation über Nachrichten über Verbindungsnetz
  - Prozessorknoten durch serielle Punkt-zu-Punkt-Verbindungen gekoppelt
  - Skalierbarkeit theoretisch unbegrenzt
- fast alle Supercomputer nach diesem Prinzip

## Verbindungsnetzwerke Ziele

- möglichst schnell
- so viele parallele Verbindungen wie möglich
- möglichst geringe Kosten

## Ende-zu-Ende Übertragungszeit

$$\text{Time}(n) = \text{overhead} + \text{routing delay} + \text{channel occupancy} \\ + \text{contention delay}$$

- overhead: Zeit um Nachrichten in und aus Netzwerk zu bekommen
- routing delay: in jedem Netzwerkbauteil erfährt Nachricht Verzögerung
- channel occupancy: Zeit um  $n$  Bytes über Netzwerkkanal zu übertragen

$$(n + n_e) / b$$

$b$  = Bandbreite des Kanals

$n_e$  = Bytes des Netzwerkprotokolls

- contention delay: Zeit in der Nachricht in jedem Netzwerkteil blockiert bis nächster Netzwerkkanal frei

→ tritt auf falls zwei Pakete gleichen Kanal nutzen wollen

⑥ → Lösungen: puffern, alternative Route verwenden, Nachricht löschen

## COTS-Cluster commodity - off - the - shelf

- Hochleistungsparallelrechner basierend auf günstiger PC-Standard-HW
- Vernetzung mit Ethernet
- freie Software für OS und Systemsoftware (zB Linux, OpenMPI)
- Haushaltslüfter, Mehrfachstecker, Regale, ...

## **IBM RS/6000 SP** Nachrichtengekoppelter Multiprozessor

- 2-512 Knoten, Knotenprozessor
- Verbindungsstruktur: High-Performance Switch  $4 \times 4$  bidir. Kreuzschienen
- Knoten als einzelne Workstations oder Multiprozessorknoten
- bis zu 16 Knoten = ein Frame, redundante Stromversorgung, Steuerung, Netzwerk

zB: Deep Blue Schachcomputer mit 32-Knoten

## Logical Partitions LPARS

- aus logischen Ressourcen physikalische Maschinen schaffen  
→ volle Flexibilität, volle Isolation
- auf jeder LPAR volle OS-Kopie ausgeführt
- Gruppierung entsprechend Funktion:
  - compute Partitionen für parallele Anwendungen
  - login Partitionen für interaktiven Benutzer Login, Daten-Management, ...
  - I/O - Partitionen für Festplatten-I/O, File System
  - TSM Partitionen für Band-I/O

## Distributed-Shared-Memory Multiprozessoren

DSM

- gemeinsamer Adr.raum, Speichermodule aber auf Prozessoren verteilt
- NUMA: Zugriff eines Prozessors auf Datum im lokalen Speicher schneller als Zugriff auf Datum im lokalen Speicher eines anderen Prozessors
- i.d.R zusätzliche Caches
- CC-NUMA, COMA: Cache-cohärenz
- NCC-NUMA: puffer Cache-Blöcke nur im Falle eines prozessorlokalen Speicherzugriffs

### Arten

- 1) Zugriff für Maschinenprogramm transparent  
↳ üblich bei CC-NUMAs: HW entscheidet anhand Adr. ob lokal oder entfernt
- 2) Zugriff durch explizite Befehle, die nur entfernten Speicherzugriff  
↳ üblich bei NCC-NUMAs: Maschinenbefehlssatz des Prozessors erweitert
- 3) Software DSMs

## Software DSM

- Illusion eines gem. Speichers auf Rechen-Clustern / Multiprozessoren mit verteilten Speichern
  - jeder Prozessor kann gemeinsame Daten lesen / schreiben als ob globaler Speicher zur Verfügung
- Synchron. mit Schloss-, Semaphor-, Bedingungsvariablen
- Notwendige Netzwerkkommunikation macht DSM-System
- ④
- leichte Programmierung
  - leichte Fortierbarkeit von / zu eng gekoppelten Multiprozessoren
- ! Lokalität und Konsistenz der Daten wichtig

PROBLEM: false sharing

Datenverwaltung → Speicherkonsistenz auf Basis dieser Speichereinheiten

### Seitenbasiert

- nutzt virtuelle Speicherverwaltung des OS
  - explizite Platzierung der Daten im DSM
  - gemeinsamer virtueller Speicher aufgeteilt in Seiten mit untersch. Granularität
- ⑥ - geringere Effizienz des Seiten nachladens über Netzwerk  
Unordenes Feld von Speicherworten → keine Struktur
- ⑧ - false sharing ist noch Granularität

## Objektbasiert

- Speziell ausgewiesene Speicherbereiche
- explizite Bekanntgabe dieser Datenstrukturen
- große Blöcke aufteilen in kleinere, linear aufeinander folgende Bereiche

## Merkmale DSM

- Granularität: Größe der Speichereinheiten (Speicherseiten, Daten-Objekte)
- Einfügen der DSM-Zugriffe:
  - virtuelles Speichermanagement des OS
  - modifizierter Compiler
  - explizit im Quellcode
- Konsistenz-Modell: Vertrag zw. Anwendung und Speicher wann evtl. vorhandene Repliken von DSM-Daten konsistent

## False sharing, Flattern

- versch. Datenwörter innerhalb einer Seite werden von mehreren Prozessoren benötigt
- Kohärenzmechanismen betreffen ges. Seite  $\rightarrow$  vor Schreiben Seite anderer Prozessoren entziehen, dann neu übertragen
- mehrfache Schreibzugriffe: Blockiert häufig Prozessoren
  - \* Flattern = Seite muss immer wieder über Netz übertragen werden

Minderung:  $\rightarrow$  kleinere Seitengröße

- (+) wkt dass unabh. Datenwörter auf einer Seite kleiner
- (-) größerer Aufwand für Seitenverwaltung

## Cray T3E NCC-NUMA

- 8 - 2176 CPUs = Processing Elements (PEs)
- 64MB - 2GB Speicher pro PE

## Adressierung

- globale Adr. bei physisch verteiltem Speicher besteht aus Verarbeitungselement-Nr. (PE-Nr.) + Offset in lokalem Speicher
- 3 Ebenen der Adressierung (durch Partitionierung des Systems)
  - virtuelle PE-Nummer: Sicht der Anwendung unabh. von Lage der Partition
  - logische PE-Nummer: Sicht des OS
  - physische PE-Nummer: HW-Sicht
- $\rightarrow$  virtueller zu logischer zu physischer Adr.: Umsetzung in HW

## Verbindungsnetzwerk

- Topologie: dreidim. Torus = ringförmig geschlossenes Gitter
- jedes PE kann auf lokalem Speicher arbeiten ohne Netzwerk
- Daten und Nachrichten gleichzeitig über separate Pfade durch den Knoten bidirektional in alle Richtungen ( $x, y, z$ ) transportierbar ohne dass PE involviert  
→ kurze Verbindungswege, schnelle Übertragung

## SGI Altix CC-NUMA

- 19 Partitionen à 256 dual-core Prozessoren ( $\rightarrow$  512 cores) mit 1,6 GHz  
 $\hookrightarrow$  9728 cores, 39 TB Speicher

## Cluster Systeme "COTS-Cluster professionell machen"

- jeder Knoten eigenständiges System: eigenes OS!
- Früher: Interprozesskommunikation über Netzwerk + Nachrichten
- Heute: hybride Systeme: gemeinsamer Speicher in einem Knoten  
verteilter Speicher zw. den Knoten  
 $\rightarrow$  2 Ebenen der Parallelität

## HPC Cluster @ KIT

### BW Uni Cluster

- 512 dünne Knoten: 64 GB Hauptspeicher, 2 TB lokale Platten
- 8 fette Knoten: 1 TB Hauptspeicher, 8 TB lokale Platten
- BetWü Netz 10G Ethernet

ForHLR 1 in Top 500 Juni 2014  
in Green500 Juni 2014

### ForHLR 2

- 1152 HPC-Knoten
- 13 Knoten für Login, Management, DataMover zur LSDF } 24.048 cores  
95 TB Arbeitsspeicher
- Warmwasserkühlung direkt in den Knoten  
 $\hookrightarrow$  Wärme-Nutzung zur Gebäudeheizung

## Beschleuniger da Leistungszunahme der CPUs stagnierte

- IBM Roadrunner als erstes HPC System in top500 mit Beschleuniger
  - ↳ Field Programmable Gate Arrays als Beschleuniger

## Programmierung:

- Parallelrechner mit verteiltem Speicher in vielen Rechenknoten
  - ↳ Nachrichtenübertragung mit MPI Bibliothek
- Rechenknoten mit Multi-Core Prozessoren und gemeinsamen Adressraum und Speicher (SMP)
  - ↳ Sync., komm. über gemeinsame Variablen
- SMP Rechenknoten zusätzlich mit Beschleunigern (GPUs, FPGAs, ...)
  - ↳ Auslagern spezieller Programmteile auf Beschleuniger:  
CUDA, OpenCL, OpenACC

### OpenCL

- für C, C++: kann direkt auf OpenGL, DirectX Objekte zugreifen  
spezielle Datentypen + Operationen

### OpenACC

- Compiler Directiven im code

### Cuda

- APIs für GPUs

## Vektorrechner

Vektor = Array von Gleitpunktzahlen

- Rechner mit pipelineartig aufgebautem/n Rechenwerkten zur Verarbeitung von Vektoren
- Rechner besitzt im Rechenwerk einen Satz Vektorpipelines = Vektoreinheit
- Skalarverarbeitung = Verknüpfung einzelner Operanden
- Rechner enthält neben Vektoreinheit eine/mehrere Skalareinheiten
  - ↳ können parallel zueinander arbeiten

### Vektor-Addition:

$$A(j) = B(j) + C(j) \quad j=1, \dots, N$$

- addiere B, C also  $B(1), \dots, B(N)$  und  $C(1), \dots, C(N)$  komponentenweise mit einem Befehl

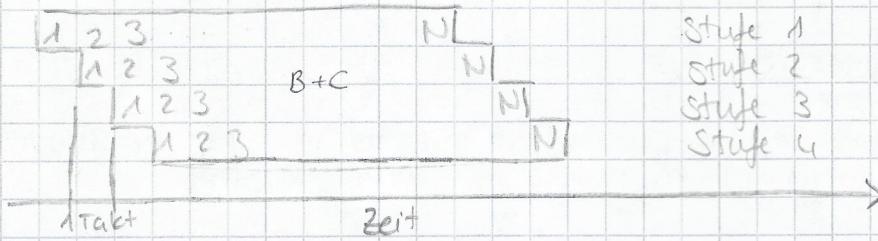
↳ Aufteilung in mehrere Schritte da zu komplex für 1 Takt

Bsp. 4 Stufen: jeder Schritt 1 Takt  $\Rightarrow$  1 Operation = 4 Takte

- Paar von Gleitkommazahlen aus Vektorregister laden
- Exponenten vergleichen, eine der Mantissen verschieben
- Ausgerichtete Mantissen addieren
- Ergebnis normalisieren, in Register zurückschreiben

→ Nächste Operation kann schon vor Beendigung der vorherigen gestartet werden → Pipeline-Prinzip

→ Berechne Vektoren sequentiell: zuerst  $B(1) + C(1)$ , dann  $B(2) + C(2), \dots$



### Besonderheiten

- Verarbeitung wird mit einem Vektorbefehl für zwei Vektoren durchgeführt  
↳ Adressrechnungen der skalaren Prozessoren entfallen
- bei ununterbrochenem Arbeiten: nach Einschwingzeit/Füllzeit nach jedem Pipeline-Takt ein Ergebnis
- Taktzeit = Dauer der längsten Teilverarbeitungszeit + Stufentransferzeit

### Verkettung

- Erweiterung des Pipeline-Prinzips auf Folge von Vektorops

→ Verkette spezialisierte Pipelines: Ergebnisse einer Pipeline direkt an nächste weitergeben

$$\text{Bsp.: } B(j) * C(j) + D(j)$$

### Parallelarbeit

#### Vektor-Pipeline-Parallelität

- durch Stufenzahl der Vektor-Pipeline gegeben

#### Mehrere Vektor-Pipelines in einer Vektorseinheit

- mehrere, meist funktional versch. Vektor-Pipelines in einer Vektorseinheit verkettet

### Vervielfachung der Pipelines

- Vektor-Pipeline vervielfachen: pro Takt Operandenpaar in mehrere parallel arbeitende gleichartige Pipelines speisen

### Mehrere Vektorseinheiten

- nach Art eines JMP parallel arbeitend

• Traditionelle Vektorrechner kaum noch vorhanden

• Vektor & Pipeline Technologie in fast jedem Prozessor

## VEC SX Serie

- 1 Knoten: 16 Vektorprozessoren, 17B gem. Hauptspeicher
- 1 Vektorprozessor  $\hat{=} 8$  Vektorpipes mit je 2 Multiplizier- & 2 Additions-Einheiten
- größtmögls. System: 512 Knoten, 8192 Prozessoren, 512 TB

## Programmiergrundlagen

### Parallelitätsebenen / -techniken

Annahme: paralleles Programm bei dem Parallelität explizit vorgelegt

- Programm darstellbar als halbgeordnete Menge\* von Befehlen, Ordnung ggf. durch Abh. der Befehle untereinander
  - unabh. Befehle parallel ausführbar
  - totale Ordnung = sequentielle Folge
  - versch. seq. Folgen können unabh. voneinander sein

Halbgeordnete Menge: Reflexiv ( $x \leq x$ ), Antisymmetrisch ( $x \leq y \wedge y \leq x \Rightarrow x = y$ ), Transitiv ( $x \leq y, y \leq z \Rightarrow x \leq z$ )

### Fünf Ebenen:

- 1) Programmebene  
z.B. gleichzeitig ausgeführte Programme in einem OS
- 2) Prozessebene  
Tasks (schwergewichtige Prozesse)  
↳ gleicher Programm, gemeinsame Daten, Kommunikation
- 3) Blockebene  
Anweisungsblöcke, leichtgewichtige Prozesse  
z.B. parallele Schleifeniterationen
- 4) Anweisungsebene  
Einzelne Maschinenbefehle / elementare Anweisungen parallel ausgeführt
- 5) Suboperationsebene  
Elementare Anweisung durch Compiler zerlegt in Suboperationen und parallel ausgeführt  
z.B. Vektorops die von Vektorpipeline überlappend parallel ausgeführt

Körnigkeit / Granularität abh. von Verhältnis Rechenaufwand zu Kommunikations- oder Synchronisationsaufwand  
→ abh. von Anzahl Befehle in seq. Befehlsfolge bevor Kommunikation oder synchr. ausgeführt wird

- Ebene 1), 2), 3) = grobkörnige Parallelität
- 3) selten = mittelkörnig
- 5) noch feinkörniger als 4)

## Techniken

1) 2) 3) 4) 5)

### durch Rechnerkopplung

- Hyper- / Metacomputer X
- workstation- cluster X

X

X

3)

4)

5)

### durch Prozessorkopplung

- Nachrichtenkopplung X
- Speicherkopplung SMP X X X
- ... X X X

### durch Prozessorarchitektur

- Befehlspipelining X
- Superskalär X
- VLIW X

X

X

X

### SIMD - Techniken

- Vektorrechner X

X

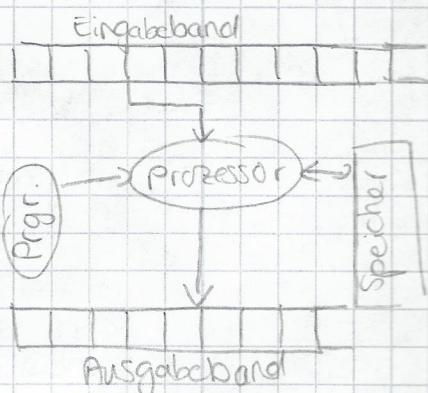
## Maschinenmodelle

### Verallgemeinerte Registermaschine (RAM)

Modell für Einprozessorechner bestehend aus:

- Recheneinheit
- Programm
- Schreib- / Lesespeicher → abzählbar viele Register
- Eingabeband
- Ausgabeband

→ nur Werte aus IN<sub>0</sub>



### PRAM (Parallel Random Access Machine)

Modell eines idealisierten, speicher gekoppelten Parallelrechner (ohne Synchronisations- / Speicherzugriffskosten)

- n - Prozessor - PRAM: n identische Prozessoren: können auf gem. Speicher zugreifen  
↳ gesteuert durch gem. Takt, führen zu Zeitpunkt dieselbe oder versch. Rechenops aus



- Zugriff auf Speicherzelle unabh. von verteilt oder zentral

Optionen Zugriffskonflikte PRAM bei gleichzeitigem Lesen/Schreiben einer Zelle

- Exclusive read (ER):  
max. 1 Prozessor kann pro Zyklus lesen
- Exclusive write (EW):  
max. 1 Prozessor kann pro Zyklus schreiben
- Concurrent read (CR):  
mehrere Prozessoren können pro Zyklus lesen
- Concurrent write (CW):  
mehrere Prozessoren können pro Zyklus schreiben  
→ Schreibkonflikte müssen gelöst werden

} kombinierbar

→ Kombinationen

- EREW-PRAM: gemeinsames Lesen-/Schreiben verboten
- CREW-PRAM: gem. Lesen erlaubt, gem. Schreiben verboten
- ER CW-PRAM: excl. lesen, concurrent schreiben
- CRCW-PRAM: conc. read., concurrent write
  - ↳ Schreibkonflikte:
    - common: gleichzeitiges Schreiben nur wenn alle gleichen Wert schreiben wollen
    - arbitrary: ein Prozessor gewinnt und schreibt, Rest ignoriert
    - priority: Prozessor mit kleinstem Index schreibt

BSF bulk synchronous parallel

- nachrichtengekoppelt
- besteht aus: mehrere Prozessoren, Kommunikationsnetz, Barrierensynchronisations-Mechanismus
- Prozessoren führen parallele Supersteps durch
- Superstep: feste Anzahl Berechnungsschritte auf lokalen Variablen  
Versand von Nachrichten über idealisiertes Komm.netz  
Barrierensync. aller Prozessoren vor nächstem Superstep

• 3 Parameter

$$p = \# \text{ Prozessoren}$$

$g = \text{Faktor Umrechnung Komm. Kosten in Berechnungskosten}$

$L = \text{min. Zeit zw. 2 Synchronisationen} = \text{Zeit für ein Superstep}$

→ ist nach Zeit  $L$  ein Superstep fertig kommt der nächste, ansonsten nochmal  $L$  warten

→ entkoppelt Komm. und Synch. ⇒ keine Deadlocks durch race conditions

→ nach Superstep durch Barriere Zustand wohldefiniert

## Log P

nachrichtengekoppelt

- Prozessoren arbeiten unabh., tauschen Daten durch Nachrichten über Netzwerk aus

### Parameter

- $L$  = Latenzzeit, max. Zeit zur Übertragung kleiner Nachrichten
- $O$  = Overhead, Zeit für Sende-/Empfangsvorgang  
↳ Annahme: Prozessor kann parallel nichts machen
- $g$  = Gap, untere Schranke für Zeit, die Prozessor braucht zw. 2 Übertragungsvorgängen
- $P$  = # Prozessoren
- Netzwerk hat endliche Bandbreite  $\rightarrow$  höchstens  $L/g$  Nachrichten können unterwegs sein

## Vergleich

### PRAM

- speichergekoppelt
- "takt" synchron
- "gut" zu programmieren

### BSP

- nachrichtengekoppelt
- Sync: Supersteps
- $\oplus$  keine deadlocks

### Log P

- nachrichtengekoppelt
- explizite Kommunikation
- gut für Laufzeitabschätzung
- $\ominus$  deadlocks

## Quantitative Maßzahlen für Parallelrechner

Ausführungszeit  $T$  = Zeit zw. Starten des parallelen Progr. bis Ende (letzter Prozessor beendet Arbeit)

$\hookrightarrow$  obzwischen alle Prozessoren in Zustand: Rechnend, kommunizierend, Wartend

$$T = T_{\text{comp}} + T_{\text{comm}} + T_{\text{idle}}$$

$T_{\text{comp}}$  Berechnungszeit = Zeit die für Rechenaufgaben verwendet wird

$T_{\text{comm}}$  Kommunikationszeit = Zeit für Sende-/Empfangsaufgaben

$T_{\text{idle}}$  Wartezeit / Inaktivitätszeit = Zeit für Warten (auf Senden/Empfang)

- $T(1) = \text{Ausführungszeit bei } 1 \text{ Prozessor}$
- $T(p) = -" - \frac{\text{P}}{p} \text{ Prozessoren}$

$$\text{Beschleunigung} = \text{Speed-up} \quad S(p) = \frac{T(1)}{T(p)}$$

$$\text{Effizienz} \quad E(p) = \frac{S(p)}{p}$$

## Relative Beschleunigung (relative Effizienz)

- Algorithmenabhängig
- Ermittlung  $T(1)$ : paralleler Algo so als sei er sequentiell und messe Laufzeit

## Absolute Beschleunigung (absolute Effizienz)

- Algorithm unabhangig
- Ermittlung  $T(1)$ : bester bekannter sequentieller Algo und messe Laufzeit

## Skalierbarkeit

- Hinzufügen weiterer Verarbeitungselement  $\Rightarrow$  kürzere Gesamtausführungszeit ohne Programm zu ändern
- lineare Steigerung der Beschleunigung mit Effizienz  $\approx 1$
- Wichtig: angemessene Problemgröße
  - $\hookrightarrow$  bei fester Problemgröße + steigender Prozessorzahl irgendwann Sättigung
  - Skalierbarkeit ist überschränkt
- = Strong scaling
- Skaliert man mit # Prozessoren auch Problemgröße  $\Rightarrow$  keine Sättigung
  - $\hookrightarrow$  Speed-up nicht direkt messbar
  - $\hookrightarrow$  Gesamtausführungszeit & max. Problemgröße
- = weak scaling

## Amdahls Gesetz Abschätzung Speedup

$q$  - Anteil Ops die parallel ausführbar

$$\Rightarrow T(p) = \underbrace{T(1)}_{\text{Paralleler Anteil}} \frac{q}{p} + \underbrace{T(1)(1-q)}_{\text{Sequentieller Anteil}}$$

$$S(p) = \frac{T(1)}{T(p)} = \frac{1}{\frac{q}{p} + (1-q)}$$

$$\Rightarrow \text{Amdahls Gesetz: } S(p) \leq \frac{1}{1-q}$$

Ideal:  $S(p) = p$ , aber  $1 \leq S(p) \leq p$  da nicht alles parallel ausführbar

ABER: gibt super-linearen Speedup  $S(p) > p$   
 $\hookrightarrow$  Cache-Effekte: ganzer Datensatz im Cache  
 $\rightarrow$  Reduzierung Speicherzugriffe  
 $\rightarrow$  kürzere Laufzeit bei paralleler Ausführung

## Prozesse und Threads

- Parallelität = spezielle Form der Nebenläufigkeit
  - ↳ parallel = von mehreren Prozessoren ausführbar
  - nebenläufig ≈ in beliebiger Reihenfolge sequentiell ausführbar

**Prozess** = funktionelle Einheit aus:

- zeitlich invariantem Programm
  - Satz von Daten zur Initialisierung
  - zeitlich variantem Zustand
- Programme aus denen mehrere Prozesse resultieren = parallele Programmsysteme

**Prozessumgebung** = geschützte Adr. Bereiche eines Prozesses

- ↳ eigener Code-, Datenbereich im Speicher
- ↳ als Seiten-/Segmenttabelle hinterlegt
- ↳ Prozesswechsel bedeutet Umgebungswechsel

**Prozesskontext** = Registerwerte des mit der Prozessausführung beschäftigten Prozessors

- Befehlszähler
- Zeiger auf Stack oder Aktivierungssatz
- Zustandsinfos, Datenwerte (nötig zur Wiederaufnahme nach Wechsel)
- ↳ Prozesswechsel bedeutet Kontextwechsel

## Laufzeitbetrachtung

Prozesse = Zeitaufwändig

- ↳ Erzeugen => OS Aufrufe
- ↳ Zugriffsschutzmechanismen falls Prozess für mehrere Nutzer gleichzeitig
- ↳ Prozesswechsel langsam
- ↳ Prozesskommunikation erfordert Konm.-Wege + Synchr. Operationen

## Schwergewichtige Prozesse

Prozesse im klassischen Sinne → z.B. Prozesse von Unix-OS

Leichtgewichtige Prozesse = Threads

Ziel: bilden Programmeinheit die parallel zu anderen Programmeinheiten arbeiten kann

- mehrere Threads teilen sich Prozessumgebung
- ↳ Threadwechsel nur Kontextwechsel, kein Umgebungswechsel

## Multi-Threaded OS

Programmierer können Threads als parallele Kontrollfäden nutzen

- ↳ Standard: POSIX P-Threads

## Synchronisation, Kommunikation über gem. Variable

- Prozesse / Threads **interdependent** = voneinander abh., müssen in bestimmter Reihenfolge ausgeführt werden

Gründe:

- Kooperation: Prozesse / Threads erfüllen Teilaufgaben einer Gesamtaufgabe

zB Producer / consumer  
Client / Server

- Konkurrenz: Aktivitäten eines Prozesses / Threads drohen andere zu behindern

=> Koordination zw. Prozessen / Threads = **Synchronisation**

↳ bringt Aktivitäten versch. Proz. / Thr. in Reihenfolge

↳ Kommunikation: Datenaustausch über Proz.- / thr. Grenzen hinweg

### Synchronisation Kommunikation durch gemeinsame Variablen

- Reihenfolge der Lese- / Schreibzugriffe je nach dem welcher Prozess der schnelleste bei Zugriffsoperationen

↳ wettkämpfen = race

- Am Ende sollte deterministisches Ergebnis sein unabh. von Zugriffsfolge durch wettkämpfen

↳ keine race conditions

→ Synch. schränkt wettkämpfen ein um Fehler zu vermeiden

### Einsitzige Sync.

A1, A2 abhängig, A1 Voraussetzung für A2

↳ A1 vor A2 ausführen

### Mehrseitige Sync.

Abfolge von A1, A2 egal, aber Schreib- / Schreib bzw.

Schreib- / Lese - Konflikte: A1 & A2 nicht gleichzeitig

↳ mutual exclusion (= gegenseitiger Ausschluss)

**critical sections** = Anweisungen deren Ausführung gegenseitiger Ausschluss erfordern

### Verklemmung, Aussperrung

- Deadlock = alle Prozesse warten auf Ergebnis das nicht mehr eintreten kann

- Aussperrung = Prozess wird undefinierbar lang verzögert

- Deadlock Voraussetzung:

1) umstrittene Ressource nur exklusiv nutzbar

2) umstrittene Ressource kann nicht entzogen werden

3) Prozesse belegen zugew. Ressourcen obwohl sie auf andere warten

4) Zyklische Abhängigkeit: Prozesse besitzen Ressourcen auf die nächster in Kette wartet

- Vermeidung: - eine der Normen. brechen
  - Bedarfssanalyse: zukünftige Ressourcen Nutzung analysieren, entspr. Zustände verbieten
  - Erkennung: versuchen Deadlocks zu erkennen + beseitigen

Schlüsselvariablen "verriegelt" kritische Abschnitte

- abstrakte Datenstruktur auf die Prozess zugreifen muss um kritischen Abschnitt zu betreten
  - lock & unlock
  - lock = Prozess wartet bis kritischer Abschnitt frei (lock offen), betrifft Abschnitt und verriegelt Schloss
  - unlock = Prozess beendet kritischen Abschnitt, gibt ihn für andere Prozesse frei
- ! unlock muss vom gleichen Prozess kommen wie lock
- ! lock = prüfen + setzen  $\rightarrow$  2 Ops  $\rightarrow$  kritischer Abschnitt  
 $\hookrightarrow$  Ops die nicht unterbrechbar sind verwenden (atomare Ops)

- z.B. test-and-set (a,b)
- swap wird solange wiederholt bis kritischer Abschnitt frei  $\rightarrow$  spin lock  $\therefore$  verbraucht Ressourcen  
 $\hookrightarrow$  mehrere Warten: zufällige Auswahl
- suspend lock: OS verwaltet wartende Prozesse, aktiviert einen statt spin lock

Semaphore Sync. aller Prozesse

- abstrakter Datentyp, nichtneg. Integer-Variable + zwei Ops (P, V)
- Semaphorzähler bekommt bei init von S Wert zugewiesen
- P(S): wenn  $S > 0$  dann  $S = S - 1$  belege krit. Abschnitt  
 ansonsten muss Prozess warten
- V(S):  $S = S + 1$  gebe krit. Abschnitt frei

$\rightarrow$  P(S), V(S) unteilbare Operationen, V kann aber von anderem ausgeführt werden als Prozess selbst

- binäre Semaphore: Wert kann nur 0 oder 1 sein (max. 1 Prozess im krit. Abschnitt)  
 ansonsten: allgemeine /zählende Semaphore

⊕ lösen Sync. problem auf niedriger Ebene

⊖Semaphore op falsch programmiert  $\rightarrow$  Systemabsturz  $\therefore$

## bedingter krit. Abschnitt

- Eintritt eines Prozesses abh. davon dass kein weiterer Prozess bereits betreten  
→ max. 1 Prozess in krit. Abschnitt
- zusätzl. abh. evtl. von Erfüllung weiterer Bedingung b
  - ↳ nur wenn  $b = \text{true}$  kann Prozess um krit. Abschnitt werben
  - ↳ sonst warten
- sobald Abschnitt frei: alle Prozesse prüfen erneut b, werben ggf. um Eintritt
  - ↳ Implementierung zur Verteilung muss fair sein

## Monitor

- abstrakte Datenstruktur mit impliziten Sync. Eigenschaften
  - Implementierungsdetails vor Nutzer verborgen
  - Zugriffsoperationen als mutual exclusion → keine gleichzeitige Benutzung der Variablen möglich
  - Monitor = # Monitorvariablen + # Prozeduren, Funktionen + Monitorkörper
    - Monitoprozeduren haben nur Zugriff auf monitorgeb. Variablen
    - Monitorkörper = Befehle zur init der Monitorvars
  - Prozess betritt Monitor durch Aufruf einer Monitoprozedur
    - dadurch Ausschluss aller anderer Prozesse
- (+) ggüber Semaphore: einmal korrekt implementiert unabh. von neuen Prozessen

ggüber bed. Abschn.: kann nicht nur Befehlsfolge beinhalten sondern ganze Prozeduren inkl. Params und Kvar

## Barriere

- Sync.-punkt für mehrere Prozesse
- Gruppe von Prozessen wartet bis alle Prozesse Barriere erreicht
- Barriere als Variable angelegt, init vor erstem Benutzen 'init-barrier' → # Prozesse auf die zu warten ist
- 'wait-barrier' muss von allen diesen Prozessen ausgeführt werden
  - ↳ Prozessanzahl aus init dekrementieren, falls noch > 0 muss Prozess warten
- Alle angekommen: wartende Prozesse aufwachen, Variable auf Startwert zurücksetzen
- Cray T3E: Oder-Barriere = tureka-Mechanismus zusätzlich
- nachrichtengekoppelte Modelle: globale Ops  $\hat{=}$  Barrieren zB aufsummieren global

## Sync., komm. über Nachrichten

- Verbrauchbarkeit: Nachricht existiert i.d.R. nur zw. senden und empfangen
- Sender muss angeben:
  - Ziel: Prozess / Knoten / Komm.kanal
  - Nachrichtenr.: zur Identifikation
  - Speicherbereich dessen Inhalt gesendet werden soll
  - Anzahl, Datentyp der Elemente (Nachrichtenlänge berechnen)
- Sequentialitätsbeziehung: Nachricht kann erst empfangen werden wenn vorher gesendet

## Sendoptionen

- Asynchron: Prozess wird bis Empfang nicht blockiert
- Synchron: Prozess blockiert bis Empfangsbestätigung erhalten
- Gebuffert: Nachrichteninhalt wird aus Sendepuffer im Anwendungsprogr. in Systempuffer kopiert
- Ungebuffert: Nachrichteninhalt wird direkt auf Verbindungsnetz geschickt
- Nichtblockierend: stößt Verschicken an, gibt Kontrolle sofort zurück an nächsten Befehl des sendenden Prozesses
- Blockierend: warte bis Nachricht komplett versendet

## Empfansoptionen

- Empfang:
  - konsumierend (zerstörend)
  - konserierend (non-destructive)
- synchron (üblich): warte bis Nachricht für Prozess eingetroffen
- asynchron: Empfangsop. liefert Nachricht oder "keine Nachricht"  
↳ kann dann zunächst weiter arbeiten

## Nachrichtenaustausch

synchron: Sender + Empfänger arbeiten synchron

asynchron: mind. einer arbeitet asynchron → Puffer nötig

## Addressierungsarten

- direkte Benennung: Prozessname als Bezeichnung von Sender + Empfänger
- Briefkasten: globaler Speicher in den Nachrichten gelegt/geholt werden  
↳ Name bekannt
- Port: Name global bekannt, an Prozess gebunden  
↳ nur in eine Richtung nutzbar
- Verbindungen / Kanal: verbinde Ports versch. Prozesse

## Aktionorientierte Komm.

Grundschema: Client / Server

- Client sendet Anforderung an Server
- Server arbeitet, Client wartet
- Server sendet Rückmeldung (+ Daten)
- Client empfängt, arbeitet weiter

## Rendezvous

- Prozesse "treffen" sich wenn beide bereit, führen Prozedur aus, trennen sich

## Parallele Programmstrukturen

- von paralleler Progr. Sprache oder Progr. Schnittstelle vorgegebene Programmschemata
- Progr. Schemata bedingen zur Laufzeit die (dynamische) Ablaufstruktur

### Fork-Join-Modell

- Beginn des parallelen Teils: erzeuge  $n$  Threads mit fork-Operation
  - $T_1, \dots, T_n$  arbeiten unabh. weiter, erreichen am Ende eine join-Operation  
 $\hookrightarrow$  Sync. mit  $T_1$  und Terminierung aller Threads
- ⌚ Erzeugungsaufwand  $T_1, \dots, T_n$

### SPMD Modell

- Erzeuge  $n$  Threads bei Programmstart, terminiere alle am Ende  
 $\rightarrow$  Kein Erzeugungsaufwand
- sequentielle Teile von allen ausgeführt
- parallele Teile jeweils von einem Thread

### Reusable-Thread-Pool Modell

Verknüpfung fork-join und SPMD

- erzeuge neue Threads wenn das erste Mal benötigt
- nach parallelen Codeblöcken: zusätzliche Threads auf idle setzen  
 $\hookrightarrow$  sequentieller Code wird nur einmal ausgeführt

# ENTWURF PARALLELER PROGRAMME

Problem: welche Progr. Abläufe parallelisieren / parallelisierbar

↳ Lösung idealerweise aus Problem entwickeln  
typischerweise aber basierend auf bekannten Grundstrukturen

Erstellung:

- manuell: explizit durch Programmierer
- halbautomatisch: Tools unterstützen Programmierer
- automatisch: parallelisierende Compiler

Prinzip:

- Datenverteilung so, dass Datenzugriffe größtenteils als lokale Speicherzugriffe  
↳ lokal = schneller
- computation/communication ratio: wieviele Befehle mind. sequenziell in parallelen Programmen ausführen bis nächster Synch./Kommunikationszeit

## phasen der Entwicklung nach Foster

4 Phasen von Problemspezifikation bis paralleler Progr.

### ① Partitionierungsphase

- Berechnungsschritte + Daten in kleine Aufgaben aufteilen
- ignoriert Systemspezifika
- Ziel: möglichst viel Parallelität entfalten

### ② Kommunikationsphase

- notwendige Komm. zur Aufgabenkoord. feststellen
  - ↳ Kommunikationsstrukturen + Algos definieren
- ⇒ Fokus von ①, ②: Entfaltung von Parallelität + gute Skalierbarkeit  
↳ finde entspr. Algo

### ③ Bündelungsphase

- evaluierter Aufgaben + Komm.strukturen bzgl. Leistungsanforderungen + Impl. Kosten
- ↳ ggf. zusammenfassen zu Aufgabenbereichen um Leistung zu erhöhen / Kosten zu sparen

### ④ Abbildungsphase

- Prozessoren Aufgaben zuordnen ( beachte Max. Prot. Auslastung, Min. Komm. Kosten)
- Zuordnung statisch vom Compiler oder dynamisch zur Laufzeit durch Lastbalancierungsalges

⇒ Fokus von ③, ④: Berücksichtigung Lokalität + leistungsorientierte Eigenschaften

## ① Partitionierungsphase

- finde Möglichkeiten der parallelen Ausführung
  - ↳ suche feingranulare Aufteilung mit möglichst vielen Tasks
  - gute Partitionierung teilt Berechnungsschritte und zugeh. Daten
  - 2 Möglichkeiten: Gebietszerlegung, Funktionszerlegung

### Gebietszerlegung

- Ausgehend von Datenpartitionierung, danach Berechnungsschritte
  - ↳ zerlegen in Bereiche mit min. Komm. + gute Lastbalance
  - Anweisungsblock, Funktion, ganzes Programm gleichzeitig auf unterteilten Daten partitionieren angewendet
- statische Struktur: Compiler / Programmierer legt Aufteilung fest
- dynamische Struktur: zur Laufzeit: Last-Balancierung

Datenmengen: Konzentration auf größte Datenstruktur od. Datenstruktur mit meistem Zugriff

### Funktionszerlegung

- Ausgehend von Berechnungsschritten, danach Daten
  - ↳ zusammenfassen zu Programmkomponenten die parallelisierbar
- für jede Komponente muss eigenes Progr. geschrieben werden

### Idealfall

- Daten können disjunkt den Tasks zugewiesen werden
  - ↳ ansonsten Kommunikation nötig

## ② Kommunikationsphase

- Bestimme Informationsfluss: welcher Task braucht welche Berechnung eines anderen Tasks?
- abstrahiere komm. zw. Tasks durch Kanal (gerichtet od. ungerichtet)

### Lokale Kommunikation

- Kommunikation nur mit direkten Nachbarn

### Globale Kommunikation

- Kommunikation mit allen möglich

### Strukturierte Komm.

- Komm.-Anforderung erzeugen regelmäßiges Muster der Kanäle

### Unstrukturierte Komm.

- erzeugen beliebigen Graphen → verkompliziert Phase ③

### Statische Komm.

- komm. partner während ges. Berechnung gleich

### Dynamische Komm.

- Komm. partner abh. von Berechnungen, ändern sich ggf. zur Laufzeit

### Synchrones Komm. Muster

- produzierende und konsumierende Tasks koordiniert
- ↳ komm. gleichzeitig

### Asynchrones Komm. Muster

- komm. anforderungen unregelmäßig
- Tasks erkennen nicht wann andere Tasks ihre Daten brauchen

## ③ Bündelungsphase

- konkrete Entscheidung unter Berücksichtigung der Zielarchitektur
  - ↳ ggf. # Tasks = # Prozessoren
- 3 konfliktierende Ziele
  - 1) Reduziere komm.kosten durch Erhöhen der Berechnungs- / komm.granularität
  - 2) Schaffe Flexibilität bzg. Skalierbarkeit
  - 3) Reduziere Softwareengineering Kosten

### Senkung der komm. kosten - Heuristiken

- Warten + Empfangen teuer
  - ↳ Senkung durch
    - weniger + seltener Datenübertragung
    - verschmelzen von Daten
- Oberfläche -vs- Volumen - Effekt:  
komm.anforderungen des tasks prop. zur Oberfläche des Teilbereichs auf dem er arbeitet, während Berechnungsanforderungen prop. zum Volumen des Teilbereichs
- Replizierte Berechnungen:  
Spalte komm.kosten wenn Berechnungen von versch. Tasks mehrfach ausgeführt
- verschmelzen Nacheinander auszuführender Tasks
  - ↳ spart komm. kosten, aber weniger Parallelität
  - ↳ kann Flexibilität zerstören ;)

## ④ Abbildungsphase

- Prozessor-Task Zuordnung
- Ziel: min. Gesamt ausführungszeit, Strategien:
  - ↳ 1) parallele Tasks auf versch. Prozessoren
  - ↳ 2) stark kommunizierende Tasks auf gleichem / benachbarten Prozessoren

### Heuristiken:

- Interprozesskomm. minimieren → Gebietszerlegung bei fester Anzahl / etwa gleich großer Tasks + strukturierte komm.

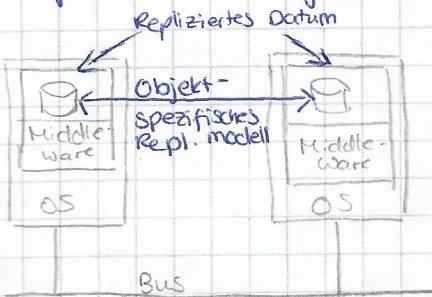
- dynamischer Lastbalancierungsalgo bei ungleichm. großen Tasks / unstrukturier. Komm. Muster / ändernde # Tasks
- Funktionszerlegung: Task-Scheduling Algo  $\rightarrow$  viele kurzlebige Tasks die viel kommunizieren

## Programmierung speicher gekoppelter Multiprozessoren

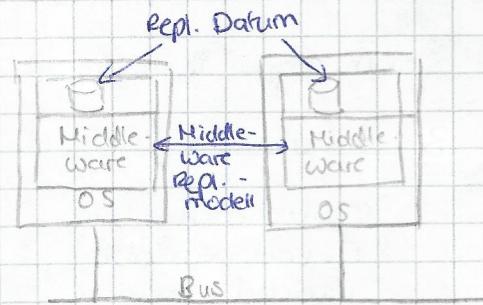
### Cache-Kohärenz

- Lesezugriff liefert immer Wert der als letztes geschrieben wurde
  - konsistent = alle Kopien im Speicher + Caches identisch  
 $\Rightarrow$  Kohärenz
  - Inkonsistenz wenn nur im Cache geschrieben wird, nicht im Hauptspeicher
- $\rightarrow$  write-update-Protokoll: wird eine Kopie in einem Cache verändert müssen alle Kopien (spätestens beim nächsten Zugriff) auch verändert werden
- write-invalidate-Protokoll: vor Verändern einer Kopie müssen alle Kopien als ungültig erklärt werden

### Replikations-Intelligenz



- Programmierer für Repl. - management verantwortlich



- System (Middleware) für Repl. - man. verantwortlich

### Konsistenzmodelle

Vertrag zw. Datenspeicher und Prozessoren  $\rightarrow$  Regeln für Zugriff

#### Strikt konsistenz einfachstes Modell

- jedes read liefert letzten write Wert
- $\hookrightarrow$  globale Zeit
- $\hookrightarrow$  nicht impl. im Multiprozessorsystem ::

#### Sequentielle Konsistenz

- jede Permutation von gültigen read-/write-Ops erlaubt, sondern alle Prozessoren sie so wahrnehmen
  - keine globale Zeit
- $\rightarrow$  zwei gleiche Programm läufe können untersch. Ergebnisse liefern

## Schwache Konsistenz

- Konsistenz nur zu gezielten Synchr. Punkten gewährleistet  
↳ kritische Bereiche ohne konkurrenzbed. Lesen/Schreiben
- Bedingungen:
  - Bevor lesen/Schreiben bzgl. beliebigen Prozessors: alle vorherigen Synchr. Punkte müssen erreicht worden sein
  - bei var Synchr. bzgl. beliebigen Prozessors: alle vorherigen Schreib-/Lesezugriffe müssen ausgeführt worden sein
  - Sync-Punkte müssen sequentiell konsistent sein

## Variablenanalyse Ziel: Schleifeniterationen parallel ausführen

### 1. Unterscheidung lokale vs. gemeinsame Variable

- > Lokale Var: neu init. vor jeder Iteration der parallelen Schleife
- > gemeinsame Var: von mehreren Iterationen gelesen/geschrieben

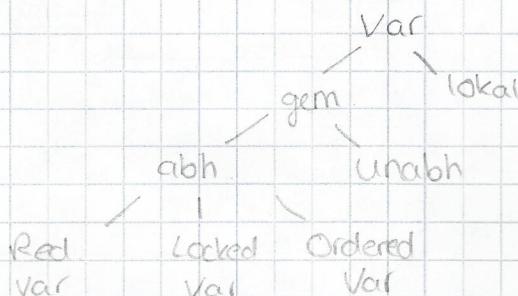
### 2. Unterscheidung (bei gemeinsamer Var)

- > unabhängig: Variable nur gelesen von allen Iterationen ODER Variable = Array, jede Iteration greift auf versch. Elemente zu

- > abhängig: krit. nicht erfüllt

### 3. Unterscheidung (bei abh. Var.)

- > Reduktionsvariable
- > Locked - Variable
- > Ordered - Variable



## Reduktionsvariable

- Array-Var oder Skalare Var
- wird nur in einer einzigen assoziativen und kommutativen Operation benutzt (+, \*, ^, v, xor)
- OP ist der Form  $\text{var} = \text{var} \text{ op } \text{expr}$ , expr enthält var nicht

## Locked-Variable

- Array-Var oder Skalare Var, kann von mehreren Iterationen gelesen/geschrieben werden  
↳ aber Reihenfolge egal

## Ordered-Variable

- Array-Var oder Skalare Var, Reihenfolge verändert Ergebnis  
↳ erzwingt sequentielle Abarbeitung der Iterationen

Optimierung: pro Ordered-Var zusätzliche Wächter Variable

## Parallelisierung von Iterationen

- möglichst über äußere Schleife
- behandle abh. Var.
  - Red. var: globale Reduktions-Ops verwendbar
  - Locked-var: kritischer Bereich lock/unlock
  - ordered-var: guards

## Programmierung nachrichtengekoppelter Multiprozessoren

Kommunikation über Nachrichten:

- Punkt - zu - Punkt: 2 Prozesse als Sender/Empfänger 1:1
- Kollektive Kommunikation: einige/lalle Prozesse nehmen teil 1:M

Nachrichtentypen:

- Daten-Nachrichten
- Kontroll-Nachrichten

### Kollektive Kommunikation

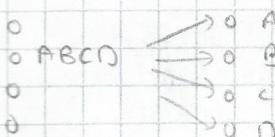
> Broadcast: einer sendet gleiche Nachricht an alle Teilnehmer



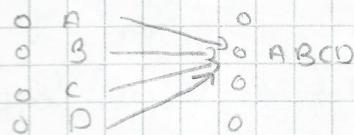
> Multicast: einer sendet gleiche Nachricht an Teil aller Teilnehmer



> Scatter: Daten von einem werden auf alle Teilnehmer verteilt



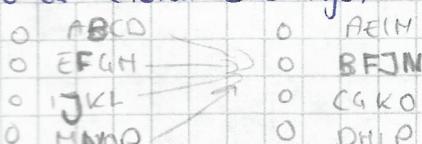
> Gather: ein Prozess sammelt Daten aller



> Gather-to-all: alle Sammeln Daten aller



> All-to-all (total exchange): Daten von allen auf alle verteilt



> Reduce : Daten von allen in einen reduziert, zB. glob. Summe/min...



> All-reduce = reduce + all-to-all

> Reduce - scatter = reduce + scatter

> parallel-prefix = alle erhalten partielle reduce Ergebnis

## OpenMP

üblicherweise um Schleifen zu parallelisieren

### Shared Memory System

• ges. System verhält sich wie ein einzelnes Comp. System

↳ alle Prozessoren zugriff auf shared mem.

↳ nur eine Kopie des OS

• Parallelisierung über shared memory oder message passing

↳ OpenMP

MPI

### Merkmale OpenMP

• Daten gespeichert im shared mem.

• ein Prozess beim Progr. start → parallele Threads zur Laufzeit erzeugt/zerstört  
↳ Threads können auf gem. oder private Daten zugreifen

• grob- oder feinkörnige Parallelisierung

Parallel Region = Code darin wird von allen ersteuerten Threads ausgeführt

Work Sharing Konstrukte :  
- Do-Schleifen : Schleifenindizes von versch. Threads  
- Codesegmente : omp sections sind gegeneinander abgegrenzt → werden von versch. Threads abgearbeitet

↳ work sharing Konstrukte liegen innerhalb parallel regions!

• Aufruf über Direktiven #pragma omp parallel { ... } [parallel region]  
Laufzeitbibliotheken  
Umgebungsvariablen

• Parallel Region OMP PARALLEL [clauses] → zB private/critical/...

### Schedule - Klausel:

- static: teile Iterationen in Stücke, statische Zuteilung zu Threads in einem Team
- dynamic: -"-, sobald Thread fertig mit einem Stück, bekommt nächstes
- guided: -"-, jeder Thread bekommt ein Stück, nachdem fertig immer kleinere
- runtime: Schedule bestimmt durch Umgebungsvariable

- OMP DO: folgende Schleife wird aufgeteilt auf alle threads des parallelen Teams  
→ Anzahl Iters. muss bei Start der Schleife bekannt sein
- OMP SECTIONS: verhandelte sections werden zwischen parallelen Teams aufgeteilt, jede section wird von einem Thread einmal durchlaufen.
- OMP Task: erzeugt Task (Code + Daten) → wird von Thread gepackt

### Gültigkeitsbereich Variablen

- die meisten shared (shared mem. modell)
- globale Vars gemeinsam unter Threads
- privat:
  - automatische Vars innerhalb Anweisungsblock
  - Stackvars in unterprogr., die von parallel regions aus aufgerufen
  - einige Schleifenindizes

### Programmierfehler shared mem.

- race conditions
- deadlocks
- False sharing  
mehrere Threads schreiben auf Daten der gleichen CacheLine ↳  
↳ CacheLine muss zw. den CPUs der threads bewegt werden  
↳ zeitintensiv

### Mutual exclusion Sync. Klausel critical

nur ein Thread kann zu Zeitpunkt Abschnitt betreten

atomic spezialfall  
Skalare Vars bekommen neuen Wert in einem Schritt

## MPI

### Distributed Memory System

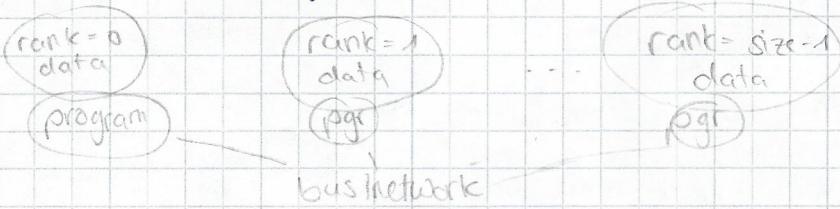
- jeder Knoten = eigenständiges Computersys.  
↳ eigenes OS  
↳ Zugriff nur auf lokalen Speicher
- Parallelisierung über Message Passing Interface

### Message Passing Paradigma

- auf jedem Prozessor in einem message-passing-Progr. läuft genau ein Prozess  
→ typischerweise gleiches Progr. auf allen, mit lokalen Daten
- Kommunikation über send & receive (MPI)

## Daten-, Arbeitsverteilung

- size: Prozessoren → Wert von MPI-Routine festgelegt  
↳ rank ebenfalls davon Prozessoren zugordnet
- Alle Prozesse durch MPI Init gestartet.
- Verteilungsentscheidungen (Daten) abh. von rank



## SPMD Single Program, Multiple Data

→ gleiches Progr. auf allen Prozessoren über versch. Datensätze

• MPI erlaubt auch MPMD → kann auch durch SPMD emuliert werden

## Messages Datenpakete von einem Prozess zum anderen

notwendige Infos:

- |                     |                   |
|---------------------|-------------------|
| • Sender rank       | - Empfänger rank  |
| • Quelllokation     | - Ziellokation    |
| • Datentyp Quelle   | - Datentyp Ziel   |
| • Datengröße Quelle | - Datengröße Ziel |

## P2P Kommunikation

• Prozess sendet Botschaft gezielt an anderen Prozess

→ synchron: Sender bekommt Empfangsbestätigung

→ Asynchron (gepuffert): Sender weiß nur ob Nachricht versandt

## Blockierendes Senden/Emfangen

- sende-/Empfangsunterprogramm wird erst verlassen wenn Operation beendet
- > synchroner Fall: wenn Empfangsbestätigung } senden
- > asynch.: wenn Daten vollständig verschickt }
- > Empfangen: wenn Daten vollst. im Anwendungsspeicher

## Nichtblockierend

- stößt komm. an, arbeitet direkt weiter
  - ggf. warten / testen ob op. beendet
  - WAIT / TEST implementieren!

## Verwendung MPI

- Erste Routine: MPI-Init, danach erst Kommandos,  
Letzte Routine: MPI-Finalize
- Alle Prozesse hängen an MPI-COMM-WORLD
- Jeder Prozess hat eigenen Rang (rank 0...size-1)
- neuer Kommunikator erzeugbar mit beliebigen Prozessen  
(auch mit angehängter Topologieinfo möglich)
- Rang ist Basis für parallelen Code + Datenverarbeitung
- Kommunikation immer innerhalb eines Kommunikators (im Zweifel WORLD)  
P2P
- Kommunikation nur im gleichen Kommunikator

## Kommunikationsarten Regeln

- Standard send:
  - min. Transferzeit
  - kann blockieren
  - Risiko: bei 'synchronous send'
- Buffered send:
  - geringe Latenzzeit / schlechte Bandbreite
- Synchronous send
  - Risiko deadlock, Serialisierung
  - Warte Risiko -> idle-Zeit
  - hohe Latenzzeit / beste Bandbreite
- Ready send
  - nicht nutzen außer Recv wurde definitiv schon aufgerufen

## Abgeleitete strukturierte Daten

Möglichkeit: Struktur Element für Element in Puffer  
→ versenden  
→ auspacken

Alternativ: Bytes am Stück versenden

## Verteiltes Rechnen

- keine einheitliche Definition
- Wikipedia: components interact to achieve common goal  
parallel comp. and distributed computing overlap
- VL: HW + SW die über geogr. Distanzen hinweg zusammen arbeiten als Problemlösungsumgebung
- Warum verteiltes Rechnen?
  - Anwendung braucht verteilte Ressourcen zB. Daten die lokal nicht verfügbar
  - zB kosteneffizienter, zuverlässiger, ...
- Architekturen:
  - Client/Server
  - Peer-to-Peer (responsibility distributed, peer can be server and client)

## Metacomputing

- Metacomputer - logische Integration eigenständiger, über HP-WAN gekoppelte (heterogene) Parallelrechner zu einem System
- Progr. Ablauf verteilt → anwendungsbezogene Selektion der ausführenden Instanzen
    - einzelne Anwendung über versch. Parallelrechner so verteilen, dass heterogene Ansammlung der Rechner GesamtAufg. bearbeiten
    - löse Probleme die nicht auf einem einzelnen Supercomp. lösbar
    - löse Probleme schneller als auf einzelnen Supercomp.
  - Gekoppelte Simulationen: jede Simulation auf der optimalen Plattform
  - Erhöhte Nutzbarkeit + Auslastung von Systemen

## Programmierung

- Nachrichtenkopplung (MPI)
- Aufwand um Progr. für Metacomp. kompatibel zu machen so gering wie möglich
  - ↳ spezielle MPI-Umgebung für Metacomp.

## Grid-Konzept

- Infrastruktur basierend auf Internet
- verteilte Ressourcen skalierbar, sicher, high-perfomant nutzbar
- virtuelle Organisationen: Verteilte Gruppen können zusammen arbeiten