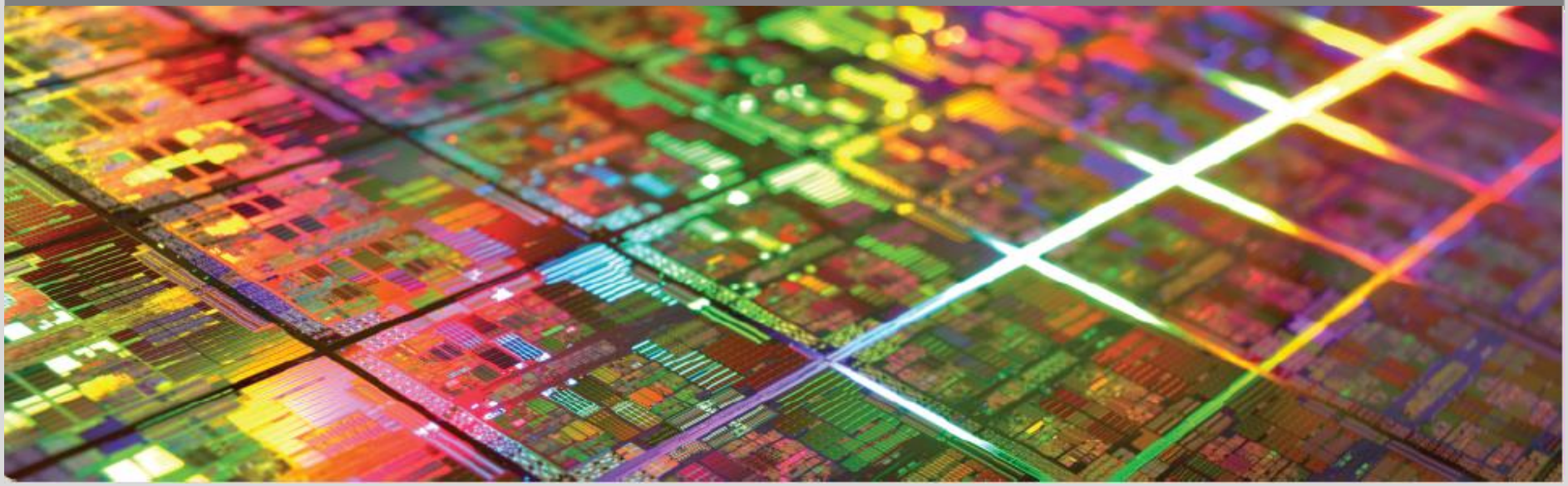
Karlsruher Institut für Technologie

# Rechnerstrukturen

## Vorlesung im Sommersemester 2013


## Prof. Dr. Wolfgang Karl

Fakultät für Informatik – Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung

# Vorlesung Rechnerstrukturen

## Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene

- 3.1 Motivation


© Prof. Dr. Wolfgang Karl

# Multiprozessorsysteme

## Motivation

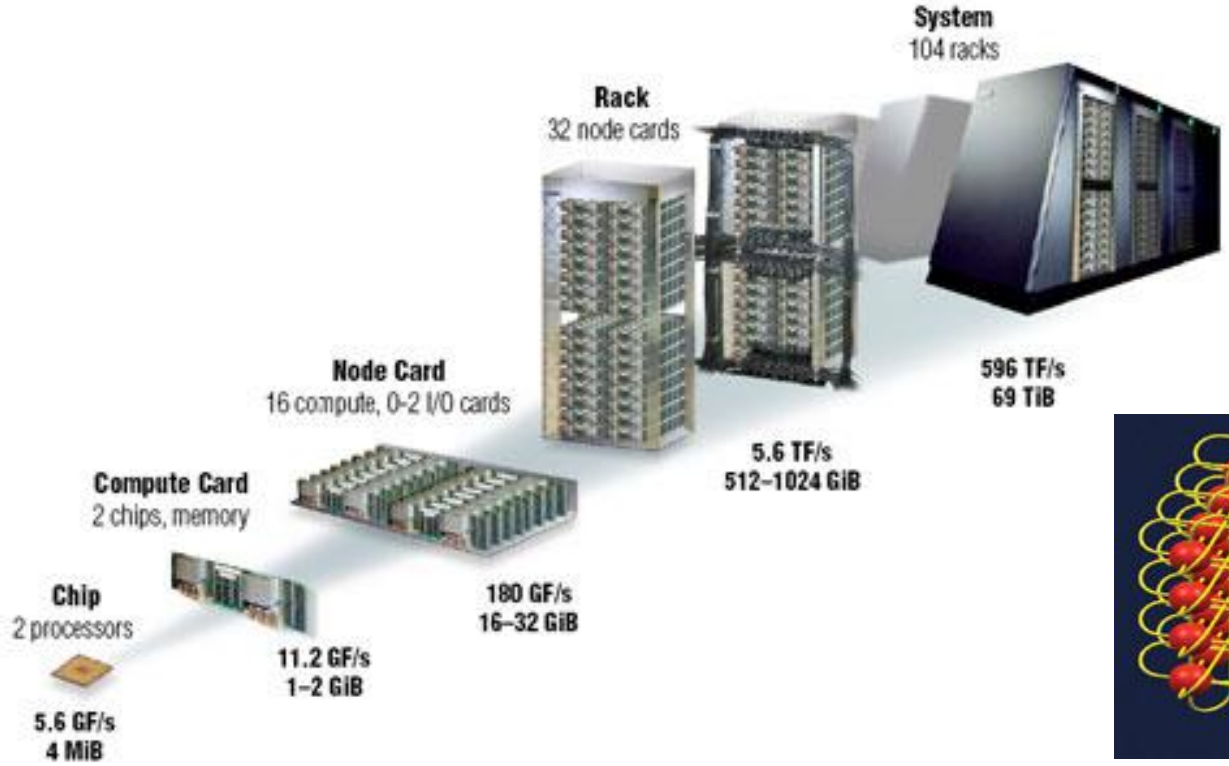
- Allgemeine Grundlagen, parallele Programmierung, Verbindungsstrukturen, Leistungsfähigkeit
- Speichergekoppelte Multiprozessoren: SMP und DSM, Cache-Kohärenz und Speicherkonsistenz, Rechnerbeispiele
- Nachrichtengekoppelte Multiprozessoren, Beispielrechner

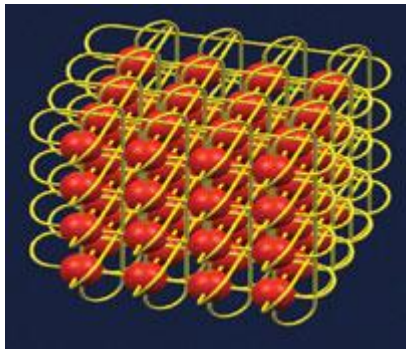

© Prof. Dr. Wolfgang Karl

# Multiprozessorsysteme

## Parallelrechner:

- **Beispiel: Multiprozessor mit verteiltem Speicher**



BlueGene/L: Interconnection Network

Quelle: https://asc.llnl.gov/computing_resources/bluegenel/photogallery.html


© Prof. Dr. Wolfgang Karl

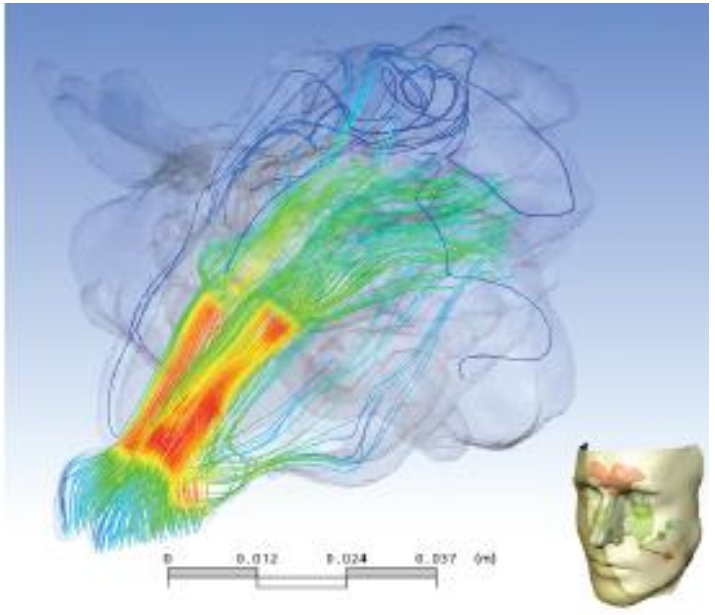
# Multiprozessorsysteme

## Anwendungsbereiche

- Hohe Anforderungen der Anwendungen an die Rechenleistung
  - Technisch-wissenschaftlicher Bereich
    - Rechnergestützte Simulation
    - Strömungsmechanik
    - Modellierung der globalen klimatischen Veränderungen
    - Struktur von Materialien
    - ...
    - Medizintechnik

Rechenzeit auf einer HP XC 4000 (15 TFLOPS): ~4 Tage

Rechenzeit auf einem Rechner im PFLOPS-Bereich: <40 min



Quelle: Persönliche Notiz: V. Heuveline


© Prof. Dr. Wolfgang Karl

# Vorlesung Rechnerstrukturen

## Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene

- 3.1 Motivation
- 3.2 Allgemeine Grundlagen


© Prof. Dr. Wolfgang Karl

## Parallele Architekturen

- Definition Parallelrechner:
  - *„A collection of processing elements that communicate and cooperate to solve large problems“* (Almase and Gottlieb, 1989)
  - Betrachtung einer parallelen Architektur als eine Erweiterung des Konzepts einer konventionellen Rechnerarchitektur um eine Kommunikationsarchitektur


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Rechnerarchitektur

- **Abstraktion**
  - Benutzer-/System-Schnittstelle
  - Hardware-/Software-Schnittstelle
- **Architektur**
  - Spezifiziert die Menge der Operationen an den Schnittstellen und die Datentypen, auf denen diese operieren
- **Organisation**
  - Realisierung der Abstraktionen


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

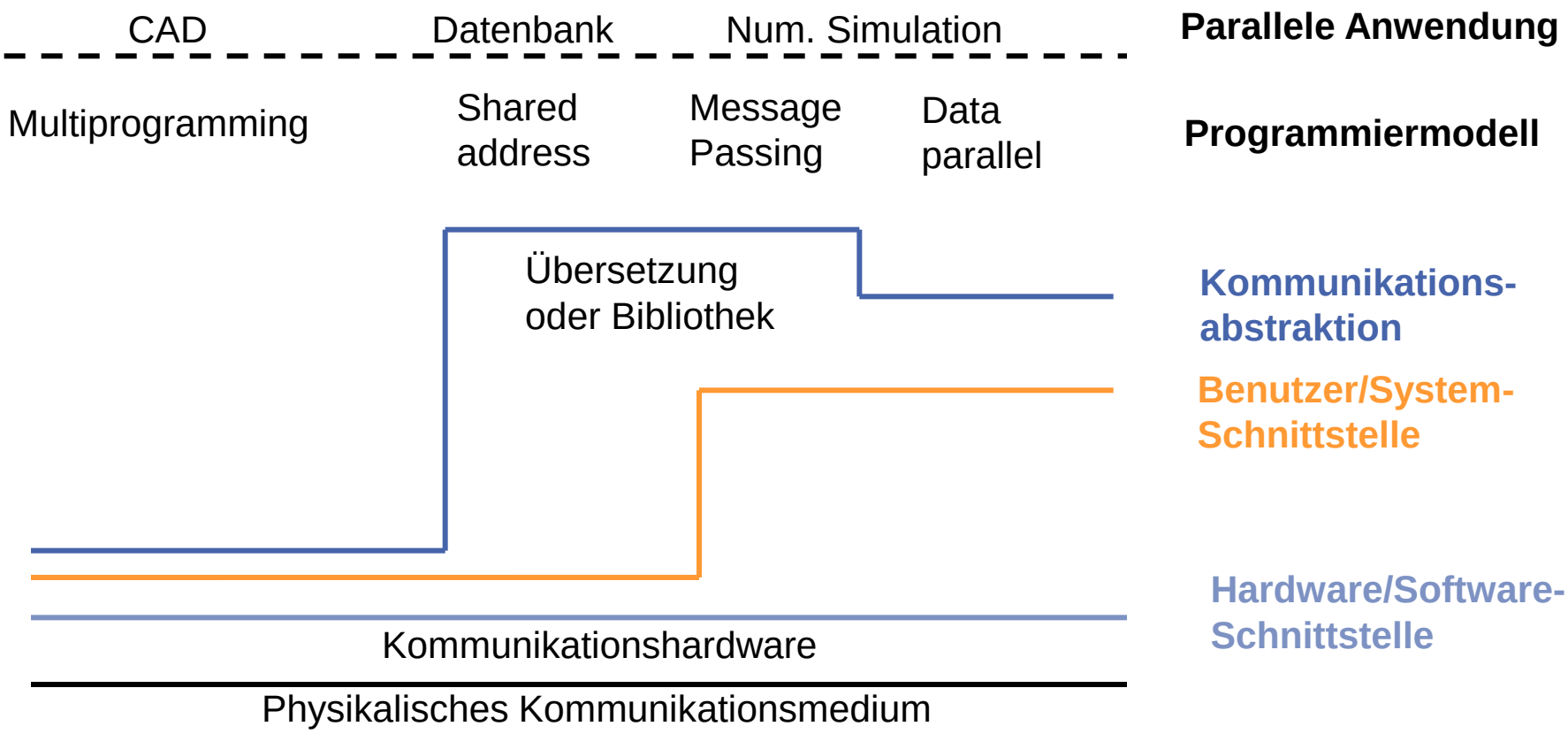
## Kommunikationsarchitektur

- **Abstraktion**
  - Benutzer-/System-Schnittstelle
  - Hardware-/Software-Schnittstelle
- **Architektur**
  - Spezifiziert die Kommunikations- und Synchronisationsoperationen
- **Organisation**
  - Realisierung dieser Operationen


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Abstraktion



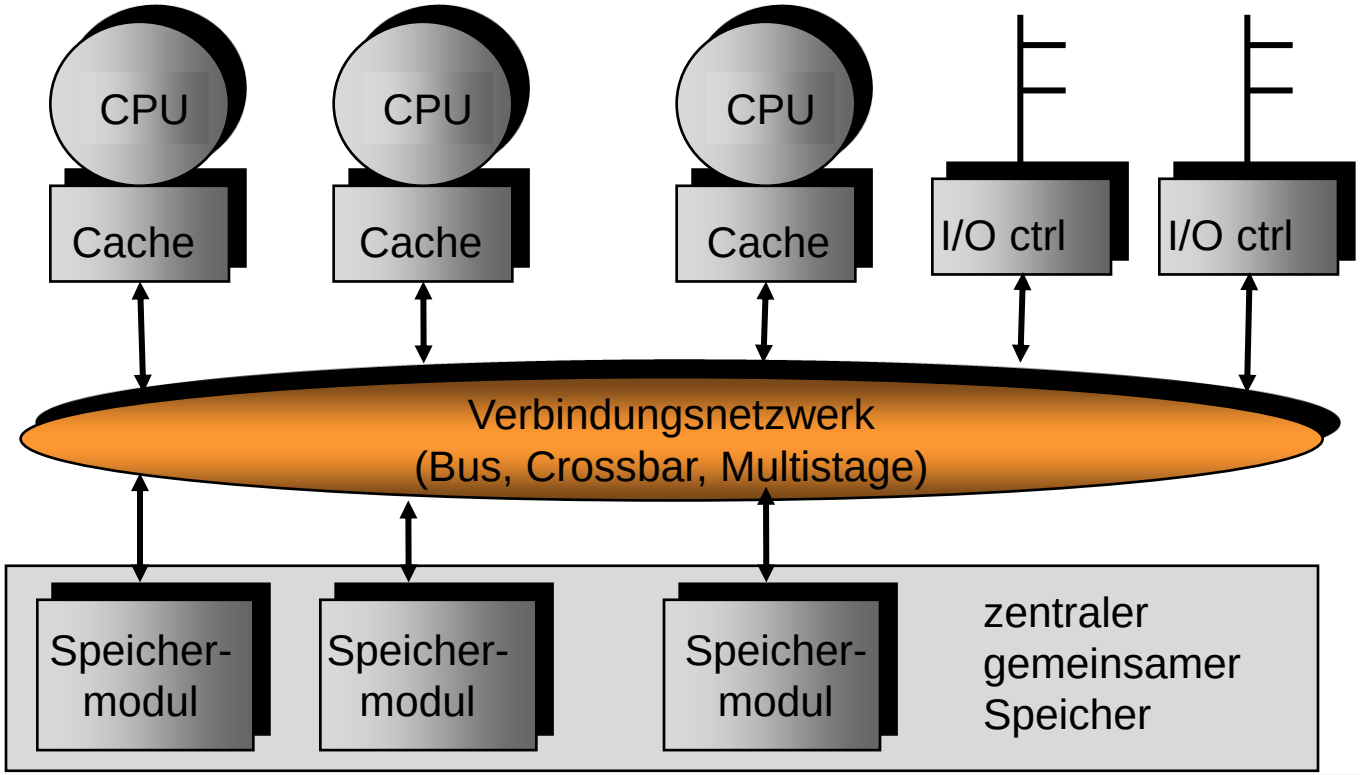


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Multiprozessor mit gemeinsamem Speicher

- **UMA**: Uniform Memory Access
- Beispiel: **symmetrischer Multiprozessor (SMP)**
  - Gleichberechtigter Zugriff der Prozessoren auf die Betriebsmittel



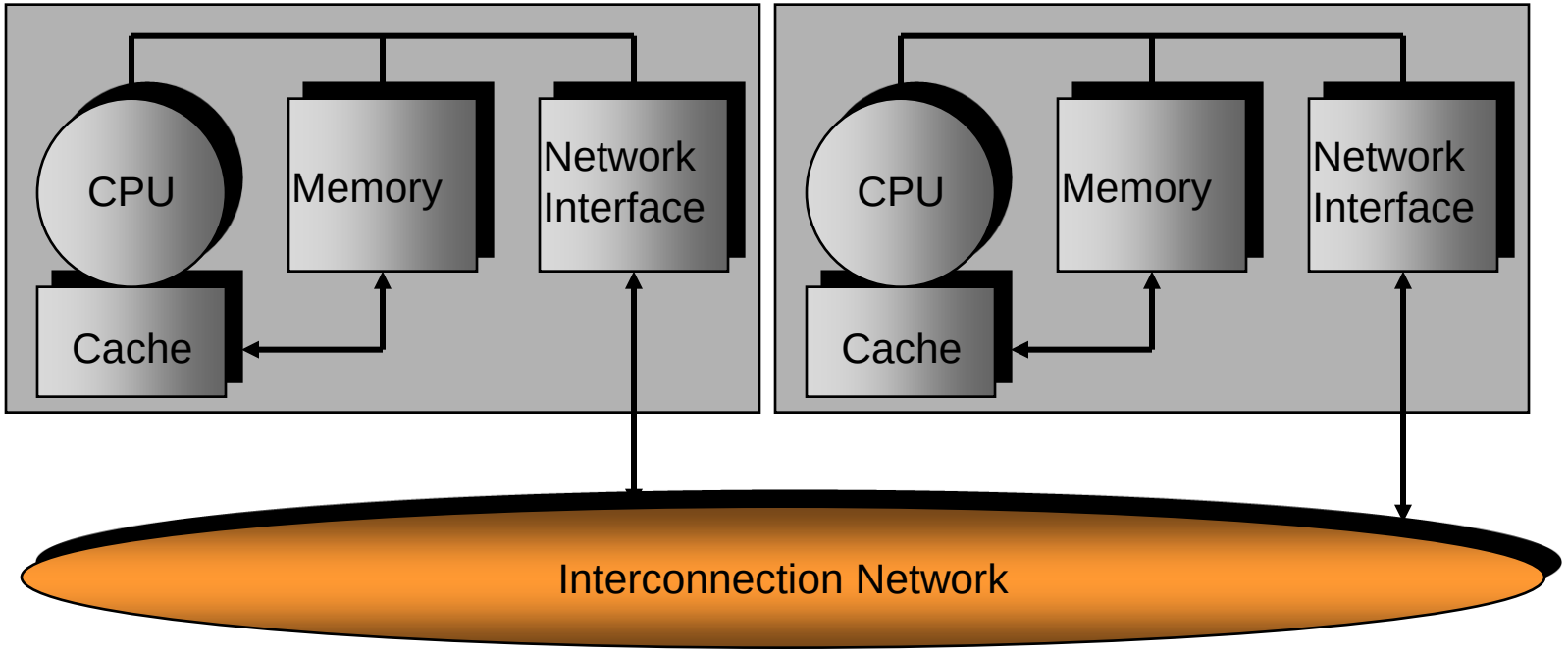


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Multiprozessor mit verteiltem Speicher

- **NORMA** No Remote Memory Access
- Beispiel: **Cluster**



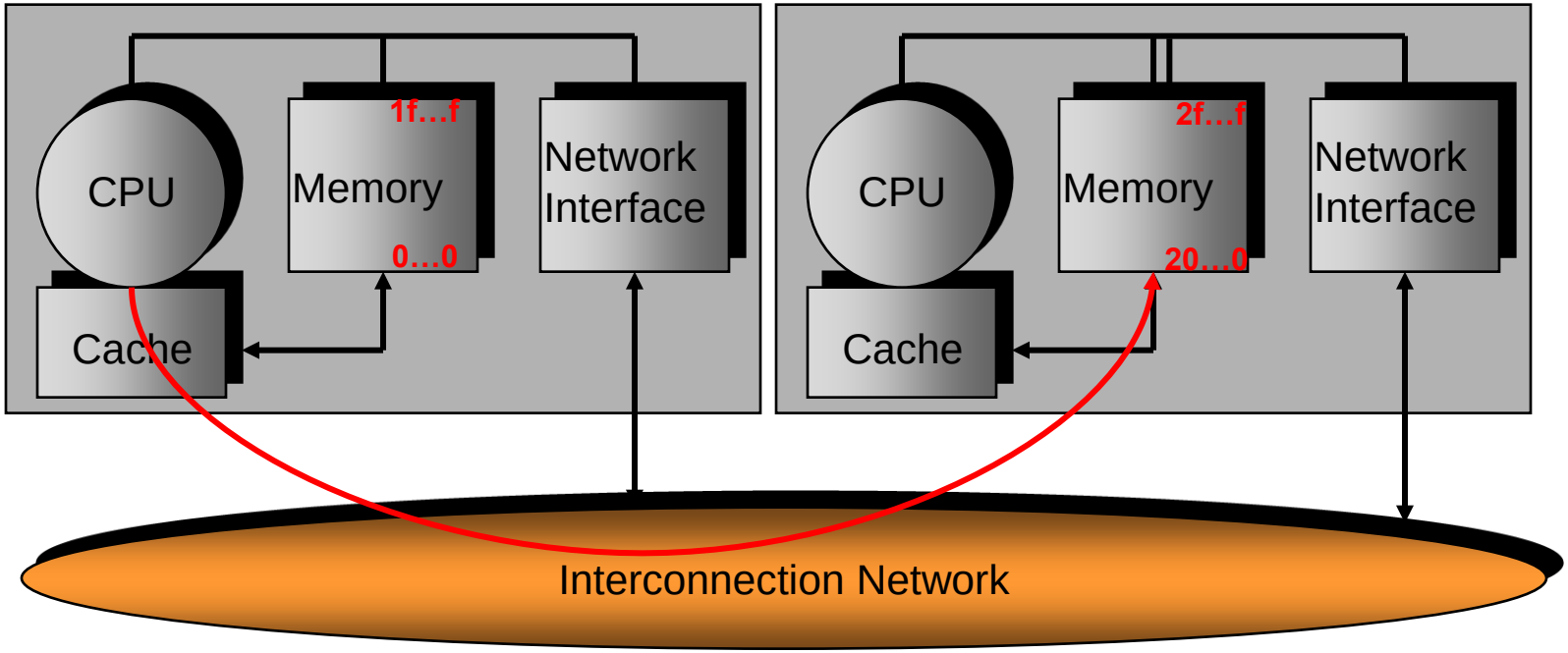


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Multiprozessor mit verteiltem gemeinsamen Speicher

- **NUMA**: Non-Uniform Memory Access
- **CC-NUMA**: Cache-Coherent Non-Uniform Memory Access
  - Globaler Adressraum: Zugriff auf entfernten Speicher





© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

- Abstraktion einer parallelen Maschine, auf der der Anwender sein Programm formuliert
- Spezifiziert, wie Teile des Programms parallel abgearbeitet werden, wie Informationen ausgetauscht werden und welche Synchronisationsoperationen verfügbar sind, um die Aktivitäten zu koordinieren
- Anwendungen werden auf der Grundlage eines parallelen Programmiermodells formuliert


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

- **Multiprogramming**
  - Menge von unabhängigen sequentiellen Programmen
  - Keine Kommunikation oder Koordination


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

- **Gemeinsamer Speicher (Shared Memory)**
  - Kommunikation und Koordination von Prozessen (Threads) über **gemeinsame Variablen** und Zeiger, die gemeinsame Adressen referenzieren

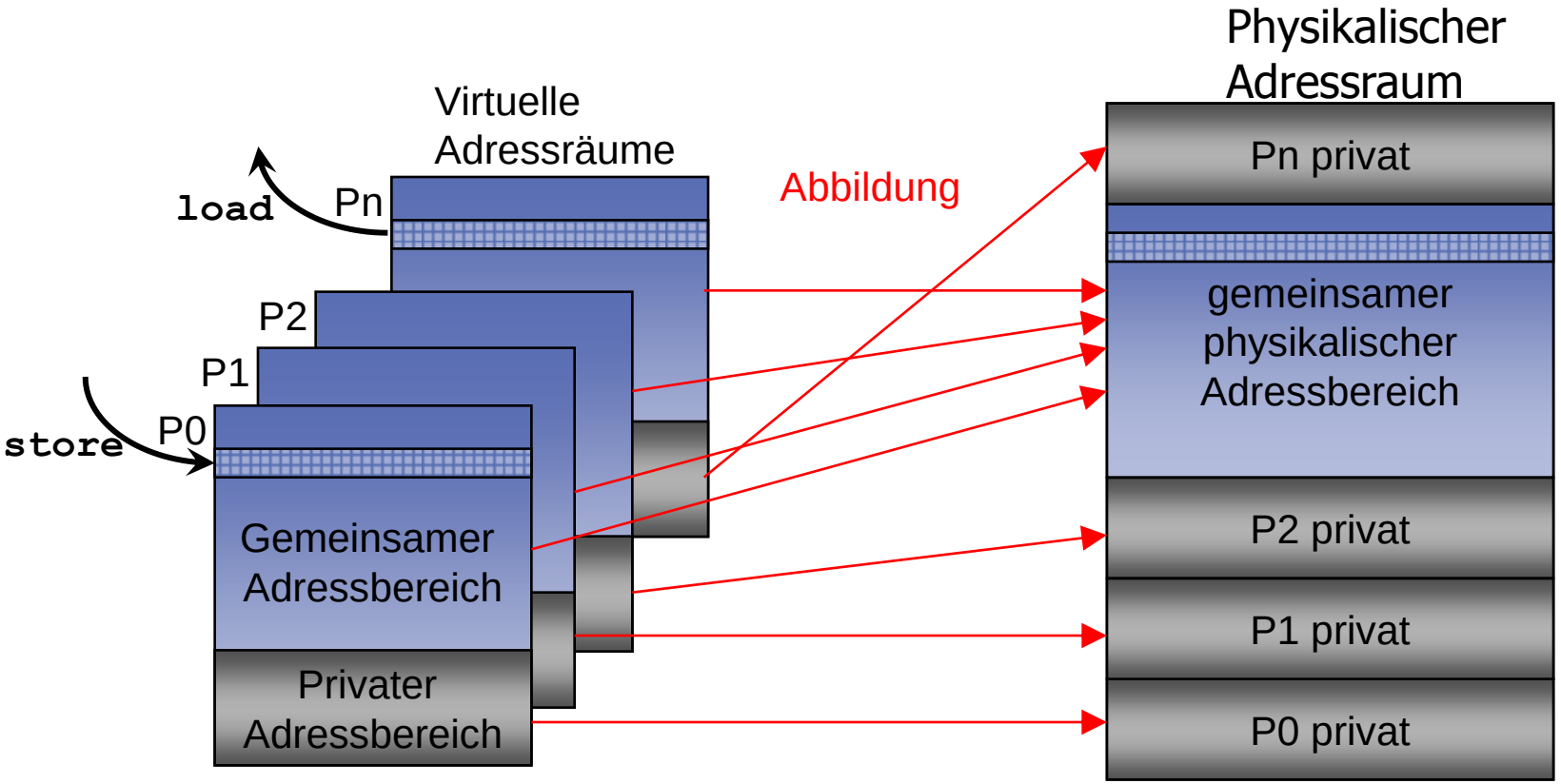
  - **Kommunikationsarchitektur**
    - Verwendung konventioneller Speicheroperationen für die Kommunikation über gemeinsame Adressen
    - Atomare Synchronisationsoperationen


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

- **Gemeinsamer Speicher (Shared Memory)**



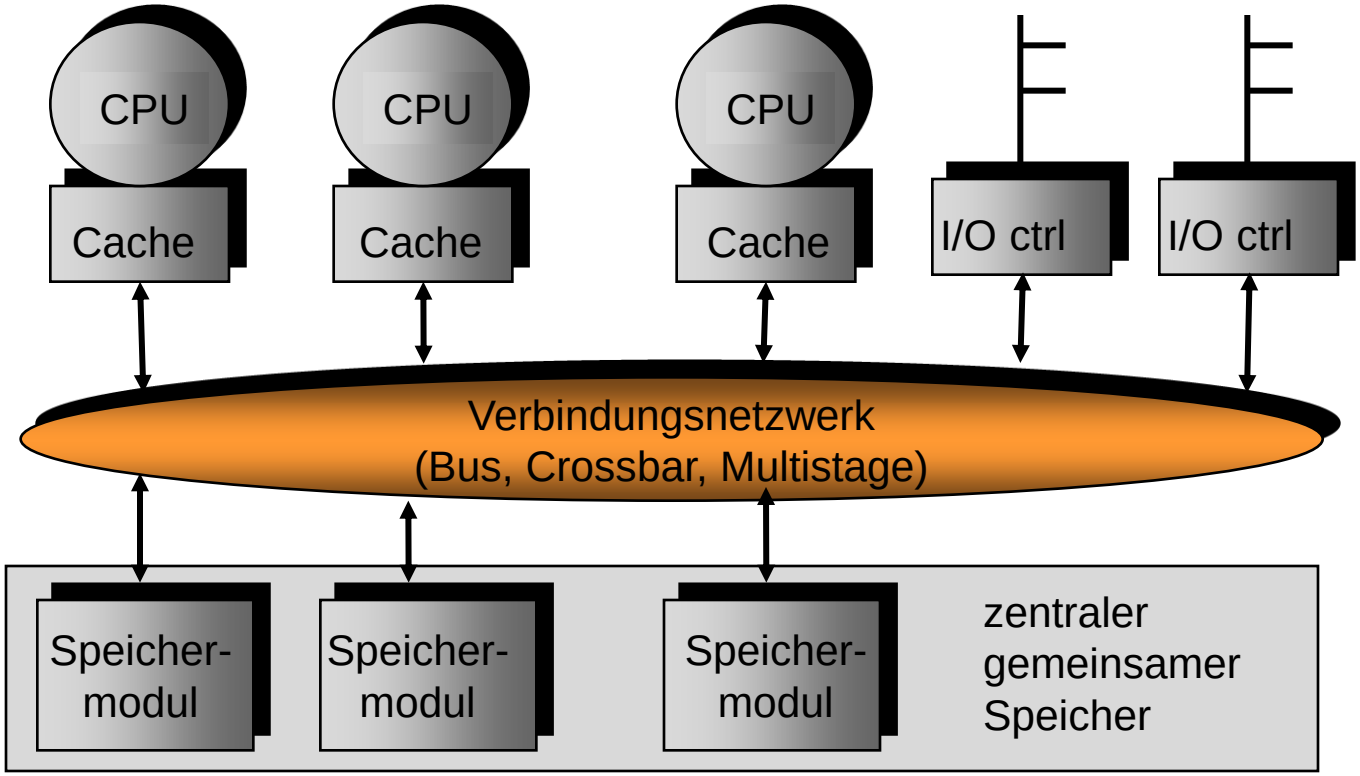


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Multiprozessor mit gemeinsamem Speicher

- UMA: Uniform Memory Access



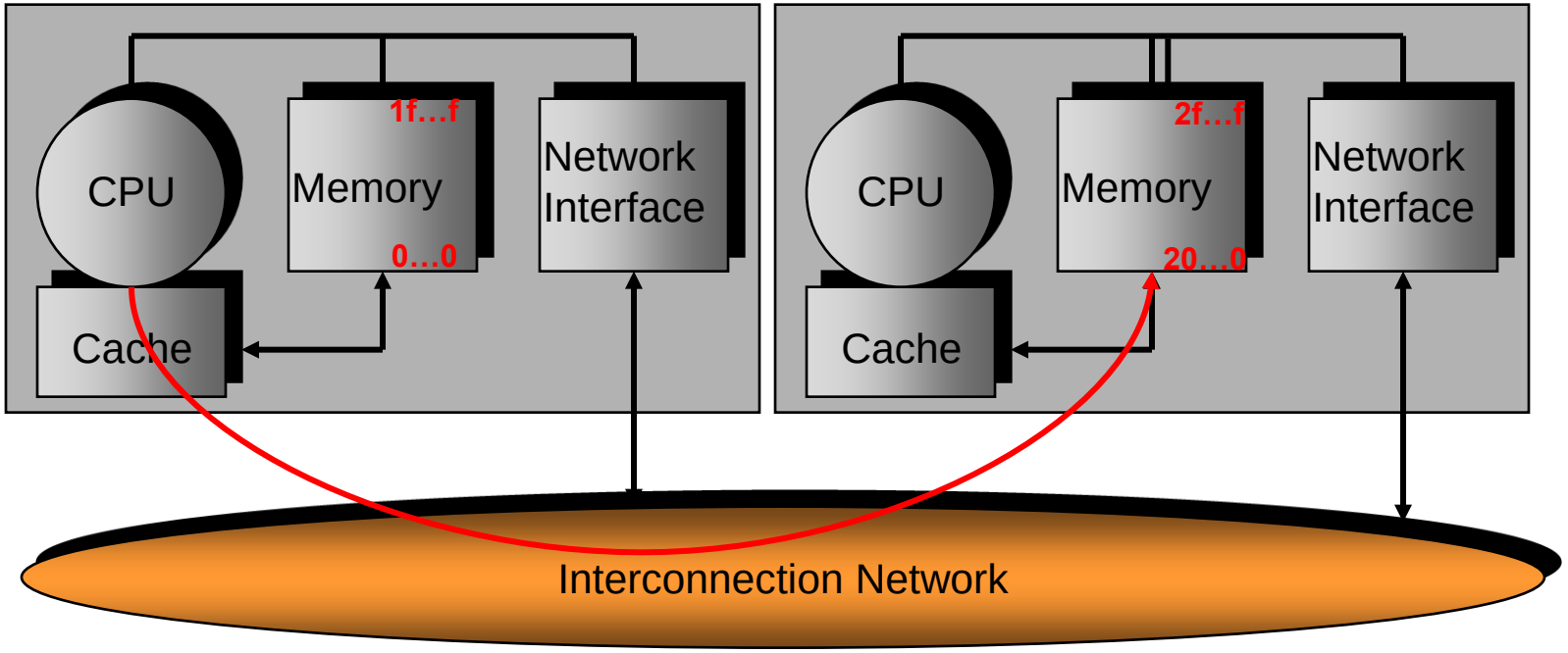


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Multiprozessor mit verteiltem gemeinsamen Speicher

- NUMA: Non-Uniform Memory Access
- CC-NUMA: Cache-Coherent Non-Uniform Memory Access
  - Globaler Adressraum: Zugriff auf entfernten Speicher





© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

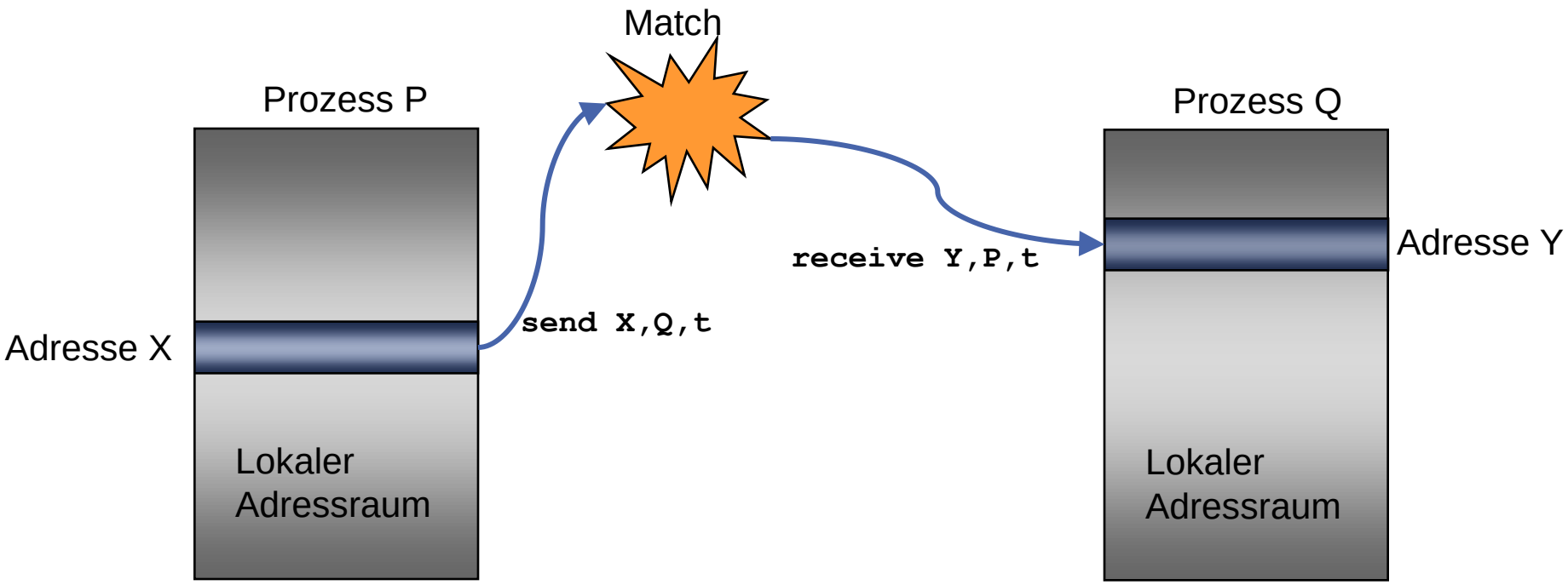
- **Nachrichtenorientiertes Programmiermodell (Message Passing)**
  - Kommunikation der Prozesse (Threads) mit Hilfe von **Nachrichten**
    - Kein gemeinsamer Adressbereich
  - **Kommunikationsarchitektur**
    - Verwendung von korrespondierenden Send- und Receive-Operationen
    - **Send**: Spezifikation eines lokalen Datenpuffers und eines Empfangsprozesses (auf einem entfernten Prozessor)
    - **Receive**: Spezifikation des Sende-Prozesses und eines lokalen Datenpuffers, in den die Daten ankommen


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

- **Nachrichtenorientiertes Programmiermodell (Message Passing)**



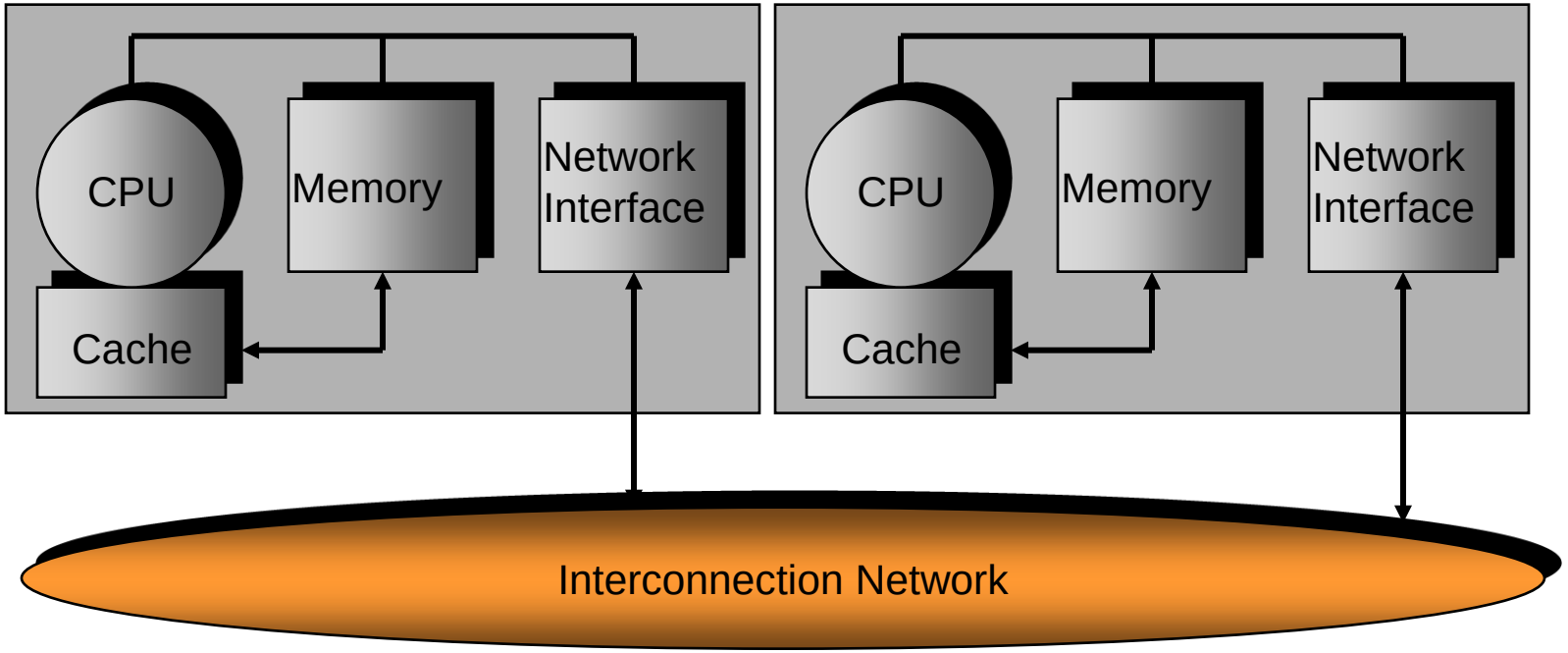


© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Multiprozessor mit verteiltem Speicher

- NORMA No Remote Memory Access





© Prof. Dr. Wolfgang Karl

# Parallele Architekturen

## Programmiermodell

- **Datenparallelismus**
  - Gleichzeitige Ausführung von Operationen auf getrennten Elementen einer Datenmenge (Feld, Matrix)
  - Typischerweise in Vektorprozessoren


© Prof. Dr. Wolfgang Karl

# Vorlesung Rechnerstrukturen

## Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene

- 3.1 Motivation
- 3.2 Allgemeine Grundlagen
- 3.3 Parallele Programmierung


© Prof. Dr. Wolfgang Karl

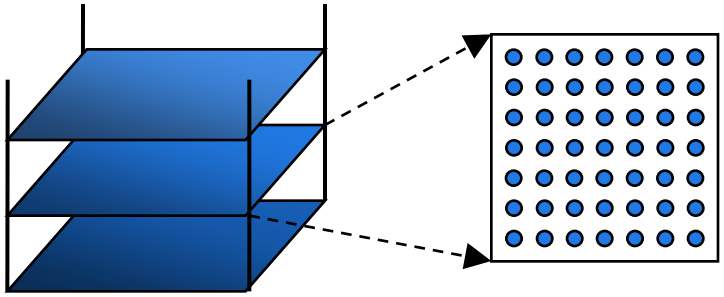
## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- Benchmark-Programm aus der SPLASH-Benchmark-Suite
  - **Modellierung des Erdklimas**
    - Gegenseitige Beeinflussung der Atmosphäre und der Ozeane, die ¾ der Erdoberfläche ausmachen
  - **Simulation der Bewegung der Wasserströmung im Ozean**
    - Strömung entwickelt sich unter dem Einfluss mehrerer physikalischer Kräfte, einschließlich atmosphärischer Effekte, dem Wind und der Reibung am Grund des Ozeans;
    - Vertikale Reibung an den „Rändern“: führt zu Wirbelströmung
    - Ziel: Simulation dieser Wirbelströme über der Zeit


© Prof. Dr. Wolfgang Karl

## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- **Diskretisierung:**
  - **Raum**:
    - Modellierung des Ozeansbeckens als ein Gitter von diskreten Punkten
    - Jede Variable (Druck, Geschwindigkeit, …) hat einen Wert an jedem Gitterpunkt
    - Zweidimensionales Gitter:



Modellierung des Ozeans
in einem rechteckigen Becken

  - **Zeit**:
    - Endliche Folge von Zeitschritten


© Prof. Dr. Wolfgang Karl

## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- **Lösung der Bewegungsgleichungen:**
  - An allen Gitterpunkten in einem Zeitschritt
  - In jedem Zeitschritt werden die Variablen neu berechnet
  - Wiederholung der Berechnung mit jedem Zeitschritt
  - Jeder Zeitschritt besteht aus mehreren Phasen


© Prof. Dr. Wolfgang Karl

## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- **Lösung der Bewegungsgleichungen:**
  - Je mehr Gitterpunkte verwendet werden, desto feiner ist die Auflösung der Diskretisierung und desto genauer ist die Simulation
  - Für einen Ozean wie den Atlantik, der etwa eine Fläche von 2000km x 2000km umspannt bedeutet ein Gitter mit 100 x 100 Punkten eine Distanz von 20 km in jeder Dimension
  - Kürzere physikalische Intervalle zwischen den Zeitschritten führen zu einer höheren Simulationsgenauigkeit
  - Simulation der Ozeanbewegung über einen Zeitraum von 5 Jahren mit einer Aktualisierung des Zustands alle 8 Stunden erfordert 5500 Zeitschritte


© Prof. Dr. Wolfgang Karl

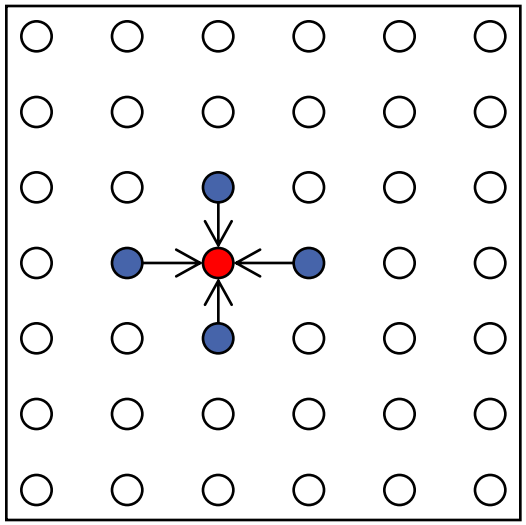
## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- **Lösung der Bewegungsgleichungen**
  - Lösung einer einfachen partiellen Differentialgleichung auf einem Gitter mit Hilfe einer finiten Differenzenmethode (Gauss-Seidel-Verfahren)
  - Reguläres zweidimensionales Gitter mit *(n+2)*(n+2)* Punkten (eine Ebene des Ozeanbeckens
  - Randwerte sind fest
  - Die inneren *n*n* Gitterpunkte werden mit Hilfe des Lösers berechnet, ausgehend von Anfangswerten


© Prof. Dr. Wolfgang Karl

## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- **Lösung der Bewegungsgleichungen**
  - Gitter



Berechnungsvorschrift für einen Gitterpunkt:

```
A[i,j]=0,2  x  (A[i,j]+
               +A[i,j-1]+A[i-1,j]
               +A[i,j+1]+A[i+1,j]
```

Wiederholte Berechnung, bis Verfahren konvergiert


© Prof. Dr. Wolfgang Karl

## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- **Sequentielle Version des Lösers:**

```
(1) int n,                                  /*size of matrix: (n+2)x(n+2)
(2) float **A, diff=0;
(3) main()
(4) begin
(5)   read(n)                               /*read input parameters*/
(6)   A←malloc (2-d array of size n+2 by n+2 doubles)
(7)   initialize(A);                        /*initialize matrix A*/
(8)   Solve (A);
(9)  end main
```


© Prof. Dr. Wolfgang Karl

## Fallstudie: OCEAN - Simulation der Ozean-Strömung

- Sequentielle Version des Lösers:

```
(1) procedure Solve (A)                    /*solve equation system*/
(2)   float **A;
(3) begin
(4)  int i,j,done=0
(5)  float temp;
(6)  while (!done) do                      /*outermost loop over sweeps*/
(7)    diff=0;                             /*initialize maximum diff*/
(8)    for i←1 to n do                     /*sweep over nonborder points*/
(9)       for j←1 to n do
(10)          temp=A[i,j];
(11)          A[i,j]←0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j]);
(12)          diff += abs(A[i,j]-temp);
(13)       end for
(14)     end for
(15)     if (diff/(n*n) < TOL) then done=1;
(16)   end while
(17) end procedure
```


© Prof. Dr. Wolfgang Karl

# Parallele Programmierung

## Parallelisierungsprozess

- Festlegen der Aufgaben, die parallel ausgeführt werden kann
  - Aufteilen der Aufgaben und der Daten auf Verarbeitungsknoten
    - Berechnung
    - Datenzugriff
    - Ein-/Ausgabe
  - Verwalten des Datenzugriffs, der Kommunikation und der Synchronisation
- Ziel: Hohe Leistung
  - Schnellere Lösung der parallelen Version gegenüber der sequentiellen Version
    - Ausgewogene Verteilung der Arbeit unter den Verarbeitungsknoten
    - Reduzierung des Kommunikations- und Synchronisationsaufwandes


© Prof. Dr. Wolfgang Karl

# Parallele Programmierung

## Parallelisierungsprozess

- Ausführung der Schritte bei der Parallelisierung
  - Durch den Programmierer
  - Auf den verschiedenen Ebenen der Systemsoftware
    - Compiler
    - Laufzeitsystem
    - Betriebssystem
  - Ideal: Automatische Parallelisierung
    - Sequentielles Programm wird automatisch in ein effizientes paralleles Programm transformiert
    - Parallelisierende Compiler
    - Parallele Programmiersprachen
    - Noch nicht vollständig möglich!


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Definitionen**
  - **Task**:
    - Beliebige Aufgabe, die durch ein Programm auszuführen ist
    - Kleinste Parallelisierungseinheit
    - Möglichkeiten beim Beispiel Ocean:
      - Berechnung eines Gitterpunkts in jeder Berechnungsphase,
      - die Berechnung einer Reihe von Gitterpunkten,
      - die Berechnung einer beliebigen Teilmenge von Gitterpunkten
    - **Granularität**
      - grobkörnig
      - feinkörnig!


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Definitionen**
  - **Prozess** oder **Thread**
    - Paralleles Programm setzt sich aus mehreren kooperierenden Prozessen zusammen, von denen jeder eine Teilmenge der Tasks ausführt
    - Tasks werden über Prozessen zugewiesen
    - Beispiel Ocean:
      - Falls die Berechnung einer Reihe von Gitterpunkten als Task angesehen wird, dann kann eine feste Anzahl von Reihen einem Prozess zugewiesen werden
      - Aufteilung einer Ebene in mehrere Streifen
    - Kommunikation der Prozesse untereinander und Synchronisation
  - **Prozessor**
    - Ausführung eines Prozesses


© Prof. Dr. Wolfgang Karl

# Parallele Programmierung

## Parallelisierungsprozess

- **Definitionen**
  - Unterscheidung Prozess und Prozessor
  - Prozessor:
    - Physikalische Ressource
  - Prozess
    - Abstraktion, Virtualisierung von einem Multiprozessor
    - Anzahl der Prozesse muss nicht gleich der Anzahl der Prozessoren eines Multiprozessorsystems sein


© Prof. Dr. Wolfgang Karl

# Parallele Programmierung

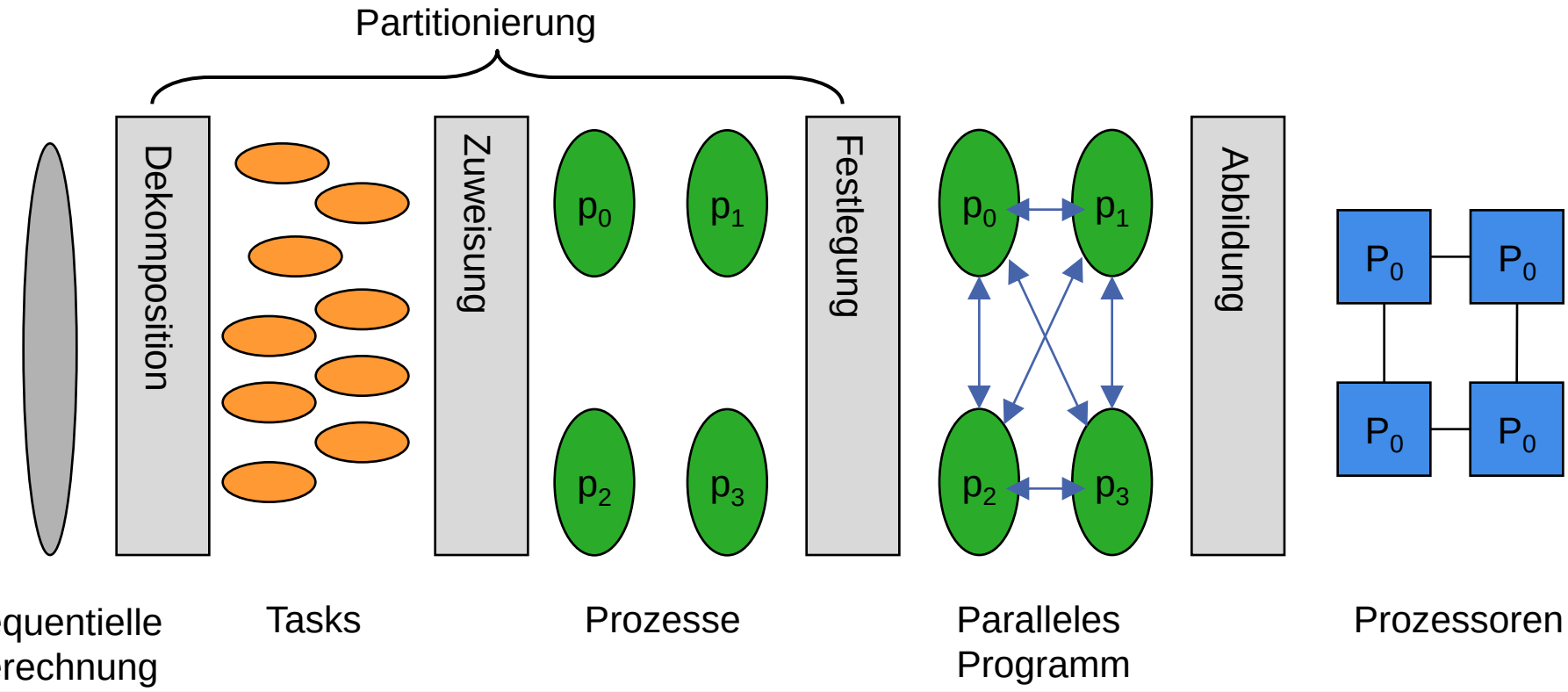
## Parallelisierungsprozess

- **Schritte bei der Parallelisierung**
  - Ausgangspunkt ist ein sequentielles Programm
  - **Aufteilung** oder **Dekomposition**
    - der Berechnung in Tasks
  - **Zuweisung**
    - der Tasks zu Prozessen
  - **Zusammenführung (Orchestration)**
    - Festlegung des notwendigen Datenzugriffs, der Kommunikation und der Synchronisation zwischen den Prozessen
  - **Abbildung**
    - der Prozesse an die Prozessoren


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- Schritte bei der Parallelisierung und die Beziehung zwischen Tasks, Prozessen und Prozessoren





© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition**
  - Aufteilung der Berechnung in eine Menge von Tasks
    - Tasks können dynamisch während der Ausführung generiert werden
    - Anzahl der Tasks kann während der Ausführung variieren
    - Maximale Anzahl der Tasks, die zu einem Zeitpunkt zur Ausführung verfügbar sind, ist eine obere Grenze für die Anzahl der Prozesse, die effektiv genutzt werden können
  - Ziel:
    - Finden von parallel ausführbaren Anteilen
    - Verwaltungsaufwand (Overhead) gering halten


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition**
  - **Beispiel: Ocean**
    - Programm strukturiert in geschachtelten Schleifen
      - Betrachtung einzelner Schleifen oder der geschachtelten Schleifen
      - Können Iterationen parallel ausgeführt werden?
      - Betrachtung über Schleifengrenzen


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition**
  - **Beispiel: Ocean**
    - Betrachtung einzelner Schleifen oder der geschachtelten Schleifen

```
(1) while (!done) do
(2)    diff=0;
(3)    for i←1 to n do
(4)       for j←1 to n do
(5)          temp=A[i,j];
(6)          A[i,j]←0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j]);
(7)          diff += abs(A[i,j]-temp);
(8)       end for
(9)     end for
(10)     if (diff/(n*n) < TOL) then done=1;
(11) end while
```


© Prof. Dr. Wolfgang Karl

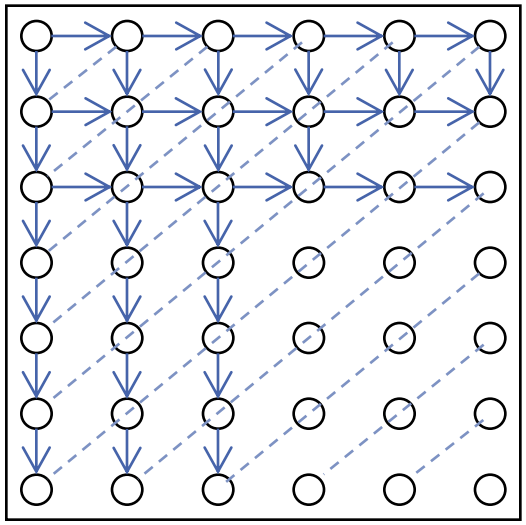
## Parallelisierungsprozess

- **Dekomposition**
  - **Beispiel: Ocean**
    - Betrachtung einzelner Schleifen oder der geschachtelten Schleifen
      - Äußere Schleife (Zeile 1-11) durchläuft das gesamte Gitter → Iterationen sind nicht unabhängig, da Daten, die in einer Iteration geändert werden, in der nächsten Iteration gebraucht werden
      - Innere Schleifen (Zeile 3-9)→Iterationen sind sequentiell abhängig, da in jeder inneren Schleife A[i,j-1] gelesen wird, der in der vorherigen Iteration geschrieben wurde


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition**
  - **Beispiel: Ocean**
    - Betrachtung der Abhängigkeiten (Granularität Gitterpunkte)



Abhängigkeiten

Verbinden Punkte, zwischen den keine Abhängigkeiten bestehen


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition am Beispiel: Ocean**
  - Aufteilen der Arbeit in einzelne Gitterpunkte, so dass die Aktualisierung eines Gitterpunktes eine Task ist
  - 1. Möglichkeit:
    - Beibehalten der Schleifenstruktur
    - erfordert Punkt-zu-Punkt-Synchronisation wegen Beachtung der Abhängigkeiten
    - Der neue Wert eines Gitterpunktes in einem Durchlauf ist berechnet, bevor er von dem westlichen oder südlichen Punkten verwendet wird
    - Verschiedene Schleifenschachtelungen und verschiedene Durchläufe können gleichzeitig ausgeführt werden
    - Hoher Aufwand!


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition am Beispiel: Ocean**
  - Aufteilen der Arbeit in einzelne Gitterpunkte, so dass die Aktualisierung eines Gitterpunktes eine Task ist
  - 2. Möglichkeit:
    - Ändern der Schleifenstruktur
    - Erste FOR-Schleife (Zeile3) geht über Anti-Diagonale und die innere Schleife geht über die Elemente der Anti-Diagonalen
      - Parallele Ausführung der inneren Schleife
      - Globale Synchronisation zwischen den Interationen der äußeren Schleife
    - Hoher Aufwand
      - Globale Synchronisation findet immer noch häufig statt
    - Lastungleichheit
      - wegen unterschiedlich vielen Elementen auf den Anti-Diagonalen


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition am Beispiel: Ocean**
  - 3. Möglichkeit: Asynchrone Methode
    - Ignorieren der Abhängigkeiten zwischen den Gitterpunkten für einen Durchlauf
    - Globale Synchronisation zwischen den Iterationen, aber keine Änderung der Durchlaufordnung
    - Prozess aktualisiert alle Punkte, sequentielle Ordnung
    - Punkte können auf mehrere Prozesse aufgeteilt werden, dann ist die Ordnung nicht vorhersagbar, sondern hängt von der Zuteilung der Punkte zu Prozessen, der Anzahl der Prozesse und wie schnell die verschiedenen Prozesse relativ zueinander während der Laufzeit ausgeführt werden, ab
    - Ausführung ist nicht deterministisch!
    - Anzahl der Durchläufe bis zur Konvergenz kann von der Anzahl der Prozesse abhängen


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Dekomposition am Beispiel: Ocean**
  - 3. Möglichkeit: Asynchrone Methode
    - Dekomposition in individuelle innere Schleifeniterationen
    - for_all: weist darunter liegende HW/SW an, dass Schleifeiterationen parallel ausgeführt werden können

```
(1) while (!done) do
(2)     diff=0;
(3)     for_all i←1 to n do
(4)        for_all j←1 to n do
(5)           temp=A[i,j];
(6)           A[i,j]←0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j]);
(7)           diff += abs(A[i,j]-temp);
(8)        end for_all
(9)      end for_all
(10)     if (diff/(n*n) < TOL) then done=1;
(11)   end while
```


© Prof. Dr. Wolfgang Karl

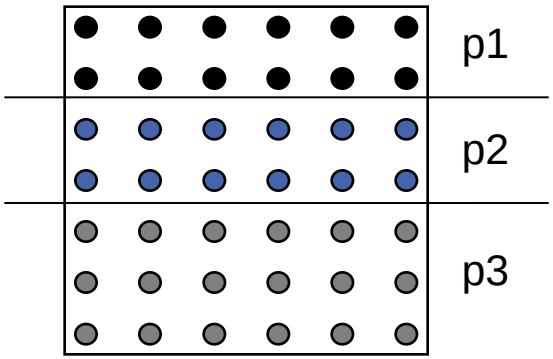
## Parallelisierungsprozess

- **Zuweisung**
  - Spezifikation des Mechanismus, mit dessen Hilfe die Tasks auf Prozesse aufgeteilt werden
  - Ziel:
    - ausgewogene Lastverteilung: Lastbalanzierung
    - Reduzierung der Interprozess-Kommunikation
    - Reduzierung des Aufwands zur Laufzeit
    - Statische oder dynamische Zuweisung


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Zuweisung**
  - Beispiel:





© Prof. Dr. Wolfgang Karl

# Parallele Programmierung

## Parallelisierungsprozess

- **Festlegung**
  - Architektur und Programmiermodell sowie die Programmiersprache spielen eine Rolle
    - Um die zugewiesenen Tasks ausführen zu können, benötigen die Prozesse Mechanismen
      - für den Zugriff auf die Daten,
      - für die Kommunikation (Austausch von Daten)
    - für die Synchronisation untereinander
  - Fragen
    - Organisation der Datenstrukturen
    - Ablauf der Tasks
    - Explizite oder implizite Kommunikation
  - Ziel:
    - Reduzierung des Kommunikations- und Synchronisationsaufwandes (aus der Sicht des Prozessors)
    - Erhalten der Lokalität der Datenzugriffe, soweit möglich
    - Reduzierung des Parallelisierungsaufwandes
  - Rechnerarchitekt: bereitstellen effizienter Mechanismen, die die Festlegung vereinfachen


© Prof. Dr. Wolfgang Karl

# Parallele Programmierung

- **Parallelisierungsprozess**
- **Festlegung**
  - Programmiermodelle:
    - Shared-Memory-Programmiermodell
    - Nachrichten-orientiertes Programmiermodell
    - Datenparalleles Programmiermodell


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Festlegung**
  - **Shared-Memory-Programmiermodell: Primitive**

| Name | Syntax | Funktion |
|---|---|---|
| `CREATE` | `CREATE(p,proc,args)` | Generiere Prozess, der die Ausführung bei der Prozedur **proc** mit den Argumenten **args** startet |
| `G_MALLOC` | `G_MALLOC(size)` | Allokation eines gemeinsamen Datenbereichs der Größe **size** Bytes |
| `LOCK` | `LOCK(name)` | Fordere wechselseitigen exklusiven Zugriff an |
| `UNLOCK` | `UNLOCK(name)` | Freigeben des Locks |


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Festlegung**
  - **Shared-Memory-Programmiermodell: Primitive**

| Name | Syntax | Funktion |
|---|---|---|
| `BARRIER` | `BARRIER(name,number)` | Globale Synchronisation für `number` Prozesse |
| `WAIT_FOR_END` | `WAIT_FOR_END(number)` | Warten, bis `number` Prozesse terminieren |
| `WAIT_FOR_FLAG` | `while (!flag); or WAIT(flag)` | Warte auf gesetztes flag; entweder wiederholte Abfrage (spin) oder blockiere; |
| `SET FLAG` | `flag=1; or SIGNAL(flag)` | Setze flag; weckt Prozess auf, der `flag` wiederholt abfragt |


© Prof. Dr. Wolfgang Karl

## Parallelisierungsprozess

- **Festlegung**
  - **Message Passing: Primitive**

| Name | Syntax | Funktion |
|---|---|---|
| **CREATE** | **CREATE(procedure)** | Erzeuge Prozess, der bei **procedure** startet |
| **SEND** | **SEND(src_addr,size,dest,tag)** | Sende **size** Bytes von Adresse **src_addr** an **dest** Prozess mit **tag** Identifier |
| **RECEIVE** | **RECEIVE(buffer_addr, size,src,tag)** | Empfange eine Nachricht mit der Kennung **tag** vom **src**-Prozess und lege **size** Bytes in Puffer bei **buffer_addr** ab |
| **BARRIER** | **BARRIER(name,number)** | Globale Synchronisation von **number** Prozessen |


© Prof. Dr. Wolfgang Karl

# Vorlesung Rechnerstrukturen

## Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene

- 3.1 Motivation
- 3.2 Allgemeine Grundlagen
- 3.3 Parallele Programmierung
- 3.4 Quantitative Maßzahlen


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

## Parallelitätsprofil

- misst die entstehende Parallelität in einem parallelen Programm bzw. bei der Ausführung auf einem Parallelrechner.
- Gibt eine Vorstellung von der inhärenten Parallelität eines Algorithmus/Programms und deren Nutzung auf einem realen oder ideellen Parallelrechner

- Grafische Darstellung:
  - Auf der x-Achse wird die Zeit und auf der y-Achse die Anzahl paralleler Aktivitäten angetragen.
  - ➔ Perioden von Berechnungs- Kommunikations- und Untätigkeitszeiten sind erkennbar.

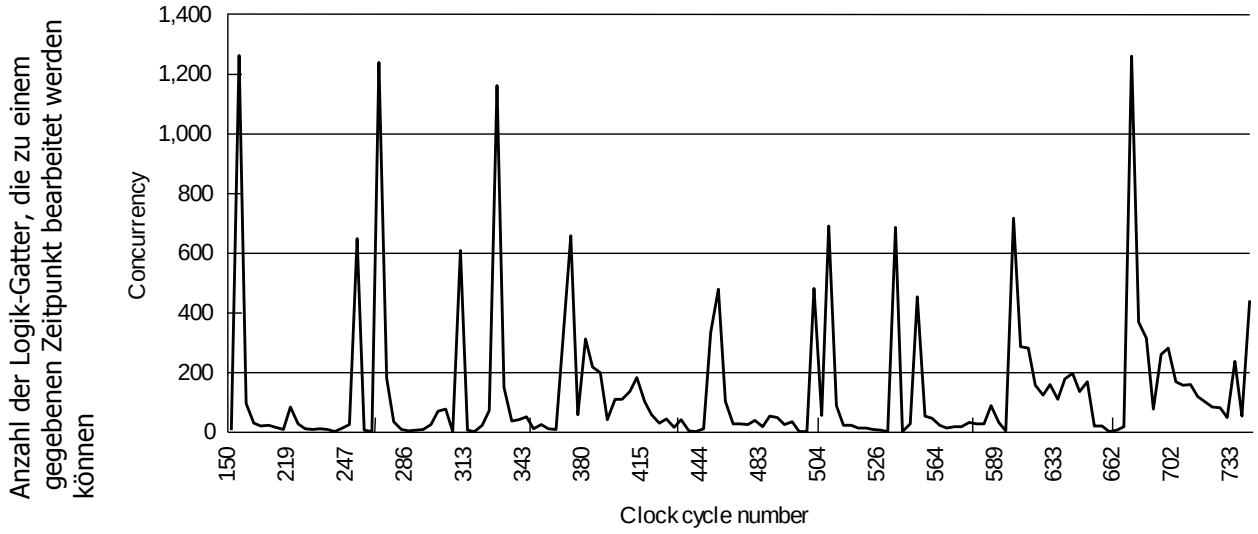

© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

## Parallelitätsprofil

- Zeigt an, wie viele Tasks einer Anwendung zu einem Zeitpunkt parallel ausgeführt werden können
- Parallelitätsgrad PG(t):
  - Anzahl der Tasks, die zu einem Zeitpunkt parallel bearbeitet werden können

Beispiel: Parallele ereignisgesteuerte Simulation der Logik-Synthese




Quelle: D. Culler: Parallel Computer Architecture. Morgan Kaufmann Publishers, 1999, p.87


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

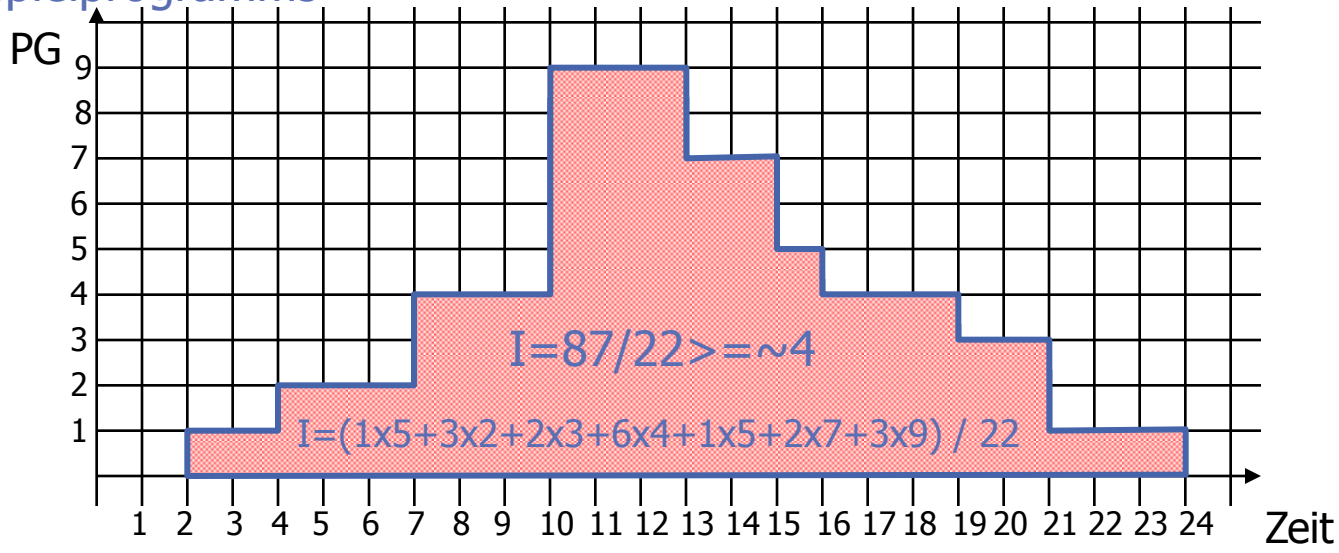
## Parallelitätsprofil

- **Parallelindex I** (Mittlerer Grad des Parallelismus):

$$I = \frac{1}{t_2 - t_1} \int_{t1}^{t2} PG(t)dt \qquad I = \left( \sum_{i=1}^{m} i * t_i \right) \Big/ \left( \sum_{i=1}^{m} t_i \right)$$

PG Bereich  Ausführungszeit

Parallelitätsprofil eines Beispielprogramms





© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

## Vergleich von Multiprozessorsystemen zu Einprozessorsystemen

- Leistungsangaben zu Multiprozessorsystemen werden mit Leistungsangaben zu Einprozessorsystemen in Beziehung gesetzt
- Notwendig:
  - Programm das auf beiden zu vergleichenden Systemen ablaufen kann


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Definitionen:**
  - P(1): Anzahl der auszuführenden (Einheits-) Operationen (Tasks) des Programms auf einem Einprozessorsystem.
  - P(n): Anzahl der auszuführenden (Einheits-) Operationen (Tasks) des Programms auf einem Multiprozessorsystem mit n Prozessoren.
  - T(1): Ausführungszeit auf einem Einprozessorsystem in Schritten (oder Takten).
  - T(n): Ausführungszeit auf einem Multiprozessorsystem mit n Prozessoren in Schritten (oder Takten).
- **Vereinfachende Voraussetzungen:**
  - T(1) = P(1),
    - da in einem Einprozessorsystem (Annahme: einfacher Prozessor) jede (Einheits-) Operation in genau einem Schritt ausgeführt werden kann.
  - $T(n) \leq P(n)$,
    - da in einem Multiprozessorsystem mit n Prozessoren ($n \geq 2$) in einem Schritt mehr als eine (Einheits-)Operation ausgeführt werden kann.


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Beschleunigung S(n) (Speedup):**

$$S(n) = \frac{T(1)}{T(n)}$$

  - Gibt die Verbesserung in der Verarbeitungsgeschwindigkeit an
  - Wert bezieht sich auf das jeweils bearbeitete Programm oder kann als Mittelwert eine Menge von Programmen angesehen werden
  - Üblicherweise gilt:


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Effizienz E(n)**

$$E(n) = \frac{S(n)}{n}$$

  - Gibt die relative Verbesserung in der Verarbeitungsgeschwindigkeit an
  - Leistungssteigerung wird mit der Anzahl der Prozessoren n normiert

  - Üblicherweise gilt:


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Beschleunigung (Speed-Up), Effizienz:**
  - Algorithmenunabhängige Definition
  - Man setzt den besten bekannten sequentiellen Algorithmus für das Einprozessorsystem in Beziehung zum vergleichbaren parallelen Algorithmus für das Multiprozessorsystem.
    - Absolute Beschleunigung
    - Absolute Effizienz


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Beschleunigung (Speed-Up), Effizienz:**
  - Algorithmenabhängige Definition
  - Man benutzt den parallelen Algorithmus so, als sei er sequentiell, und misst dessen Laufzeit auf einem Einprozessorsystem.
  - Der für die Parallelisierung erforderliche Zusatzaufwand an Kommunikation und Synchronisation kommt „ungerechterweise“ auch für den sequentiellen Algorithmus zum Tragen.
    - Relative Beschleunigung
    - Relative Effizienz


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Mehraufwand R(n) für die Parallelisierung:**

$$R(n) = \frac{P(n)}{P(1)}$$

  - Beschreibt den bei einem Multiprozessorsystem erforderlichen Mehraufwand für die Organisation, Synchronisation und Kommunikation der Prozessoren
  - Es gilt:

$$1 \leq R(n)$$

    - Anzahl der auszuführenden Operationen eines parallelen Programms größer ist als diejenige des vergleichbaren sequentiellen Programms


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- **Auslastung U(n):**

$$U(n) = \frac{I(n)}{n} = R(n) \cdot E(n) = \frac{P(n)}{n \cdot T(n)}$$

  - Entspricht dem normierten Parallelindex
  - Gibt an, wie viele Operationen (Tasks) jeder Prozessor im Durchschnitt pro Zeiteinheit ausgeführt hat


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- Folgerungen
  - Alle definierten Ausdrücke haben für n = 1 den Wert 1.
  - Der Parallelindex gibt eine obere Schranke für die Leistungssteigerung:

$$1 \leq S(n) \leq I(n) \leq n$$

  - Die Auslastung ist eine obere Schranke für die Effizienz:

$$\frac{1}{n} \leq E(n) \leq U(n) \leq 1$$


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- Zahlenbeispiel:
  - Ein Einprozessorsystem benötige für die Ausführung von 1000 Operationen 1000 Schritte.
  - Ein Multiprozessorsystem mit 4 Prozessoren benötige dafür 1200 Operationen, die aber in 400 Schritten ausgeführt werden können.
  - Damit gilt:

$$P(1) = T(1) = 1000, P(4) = 1200, T(4) = 400$$

  - Daraus ergibt sich:

$$S(4) = 2,5 \text{ und } E(4)=0,625$$

  - Die Leistungssteigerung verteilt sich als im Mittel zu 62,5% auf alle Prozessoren


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Vergleich von Multiprozessorsystemen zu Einprozessorsystemen**
- Zahlenbeispiel:
  - Parallelindex und Auslastung:

$$I(4) = 3 \text{ und } U(4) = 0,75$$

  - Es sind im Mittel drei Prozessoren gleichzeitig tätig, d.h., jeder Prozessor ist nur zu 75% der Zeit aktiv.
  - Mehraufwand:

$$R(4) = 1,2$$

  - Bei Ausführung auf dem Multiprozessorsystem sind 20% mehr Operationen als bei Ausführung auf einem Einprozessorsystem notwendig.


© Prof. Dr. Wolfgang Karl

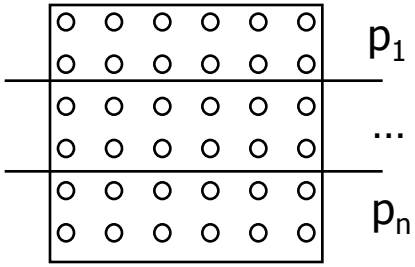
# Quantitative Maßzahlen

- **Skalierbarkeit eines Parallelrechners**
  - das Hinzufügen von weiteren Verarbeitungselementen führt zu einer kürzeren Gesamtausführungszeit, ohne dass das Programm geändert werden muss.
  - Insbesondere meint man damit eine lineare Steigerung der Beschleunigung mit einer Effizienz nahe bei Eins.
  - Wichtig für die Skalierbarkeit ist eine angemessene Problemgröße.
  - Bei fester Problemgröße und steigender Prozessorzahl wird ab einer bestimmten Prozessorzahl eine Sättigung eintreten. Die Skalierbarkeit ist in jedem Fall beschränkt.
  - Skaliert man mit der Anzahl der Prozessoren auch die Problemgröße (scaled problem analysis), so tritt dieser Effekt bei gut skalierenden Hardware- oder Software-Systemen nicht auf.


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Gesetz von Amdahl**
- Beispiel:
  - Gegeben: ein zweidimensionales *k* x *k*-Gitter
  - 1. Berechnungsphase: Ausführung einer Operation auf allen Gitterpunkten
    - Annahme: keine Abhängigkeiten zwischen den Gitterpunkten
    - Parallele Berechnung auf n Prozessoren




  - 2. Berechnungsphase: Berechnung der Summe der $k^2$ berechneten Werte der Gittersumme
    - Jeder Prozessor addiert seine $k^2/n$ berechneten Werte zur globalen Summe


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Gesetz von Amdahl**
- Beispiel:
  - Problem:
    - Akkumulation der globalen Summe muss serialisiert werden!
    - 2. Phase benötigt $k^2$ Zeiteinheiten unabhängig von *n*
    - Ausführungszeit des parallelen Programms: $k^2/n + k^2$
    - Ausführungszeit des sequentiellen Programms: $2k^2$
    - Möglicher Speedup S:

$$\frac{2k^2}{\frac{k^2}{n} + k^2} = \frac{2n}{n+1}$$

    - Selbst bei einer hohen Anzahl Prozessoren nicht mehr als 2.


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Gesetz von Amdahl**
  - Gesamtausführungszeit T(n)

$$T(n) = \underbrace{T(1) \cdot \frac{1-a}{n}}_{\text{Ausführungszeit des parallel ausführbaren Programmteils 1-a}} + \underbrace{T(1) \cdot a}_{\text{Ausführungszeit des sequentiell ausführbaren Programmteils a}}$$

a: Anteil des Programmteils, der nur sequentiell ausgeführt werden kann

  - Beschleunigung

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a} = \frac{1}{\frac{1-a}{n} + a}$$

  - Für $n \to \infty$ : $S(n) = \frac{1}{a}$


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Gesetz von Amdahl**
- Beispiel:
  - Erhöhung der Parallelität
    - Aufteilung der 2. Berechnungsphase in zwei weiteren Teilphasen:
    - 1. Teilphase: Jeder Prozessor berechnet die Summe seiner berechneten Werte
      - Kann vollständig parallel abgearbeitet werden
    - 2. Teilphase: Akkumulation der Teilsummen
      - Weiterhin seriell!
    - Ausführungszeit $T(n)=k^2/n+k^2/n+n$
    - Beschleunigung $S(n)=n*2k^2/(2k^2+n^2)$
      - Wenn n groß genug, dann nahezu linear!


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Gesetz von Amdahl**
- Diskussion
  - Amdahls Gesetz zufolge kann eine kleine Anzahl von sequentiellen Operationen die mit einem Parallelrechner erreichbare Beschleunigung signifikant begrenzen.
  - Beispiel: $a$ = 1/10 des parallelen Programms kann nur sequenziell ausgeführt werden,
    ➔ das gesamte Programm kann maximal zehnmal schneller als ein vergleichbares, rein sequenzielles Programm sein.
  - Jedoch: viele parallele Programme haben einen sehr geringen sequenziellen Anteil (a«1)


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Synergetischer Effekt und superlinearer Speedup**
- Theorie : einen „**superlinearen Speedup**“ kann es nicht geben:
  - Jeder parallele Algorithmus lässt sich auf einem Einprozessorsystem simulieren, indem in einer Schleife jeweils der nächste Schritt jedes Prozessors der parallelen Maschine emuliert wird.

- Ein „**superlinearer Speed-up**“ kann real beobachtet werden bei
  - parallelem Backtracking (depth-first search)
  - Beim Programmlauf auf einem Rechner passen die Daten nicht in den Hauptspeicher des Rechners (häufiger Seitenwechsel), aber:
    bei Verteilung auf die Knoten des Multiprozessors können die parallelen Programme vollständig in den Cache- und Hauptspeichern der einzelnen Knoten ablaufen.


© Prof. Dr. Wolfgang Karl

# Quantitative Maßzahlen

- **Weitere grundsätzliche Probleme bei Multiprozessoren**
  - Verwaltungsaufwand (Overhead)
    - Steigt mit der Zahl der zu verwaltenden Prozessoren
  - Möglichkeit von Systemverklemmungen (deadlocks)
  - Möglichkeit von Sättigungserscheinungen
    - können durch Systemengpässe (bottlenecks) verursacht werden.


© Prof. Dr. Wolfgang Karl