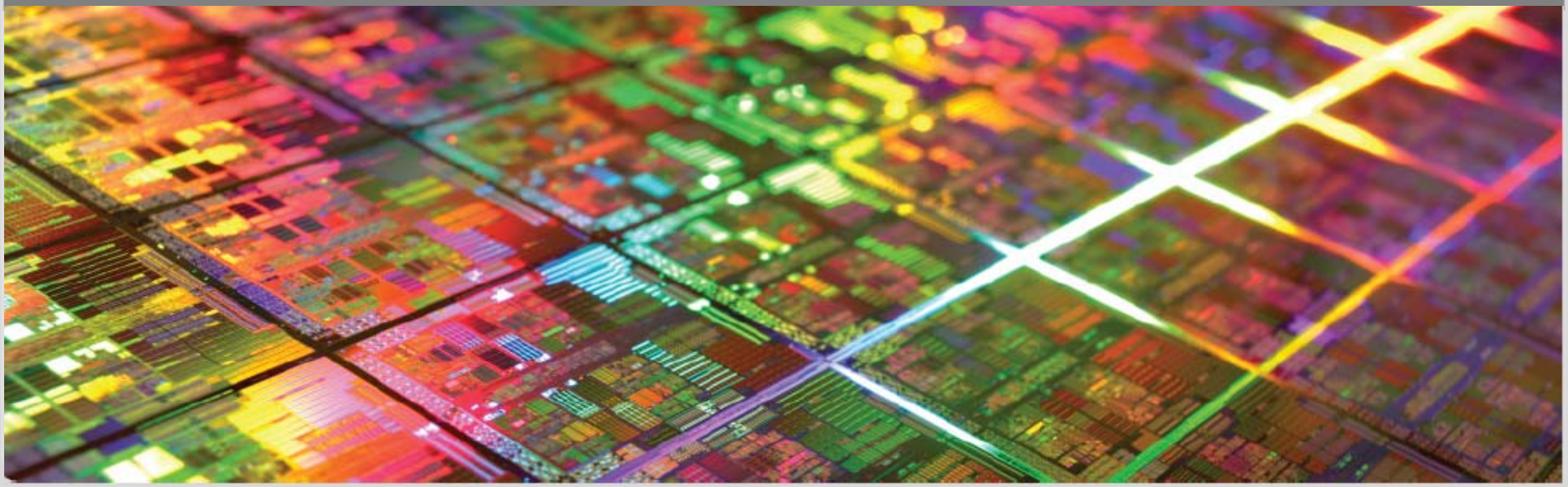
Karlsruher Institut für Technologie

Rechnerstrukturen

Vorlesung im Sommersemester 2010

Prof. Dr. Wolfgang Karl

Fakultät für Informatik – Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung



Vorlesung Rechnerstrukturen

- **Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene**
- 3.2 Allgemeine Grundlagen

Allgemeine Grundlagen

- **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

Parallele Architekturen

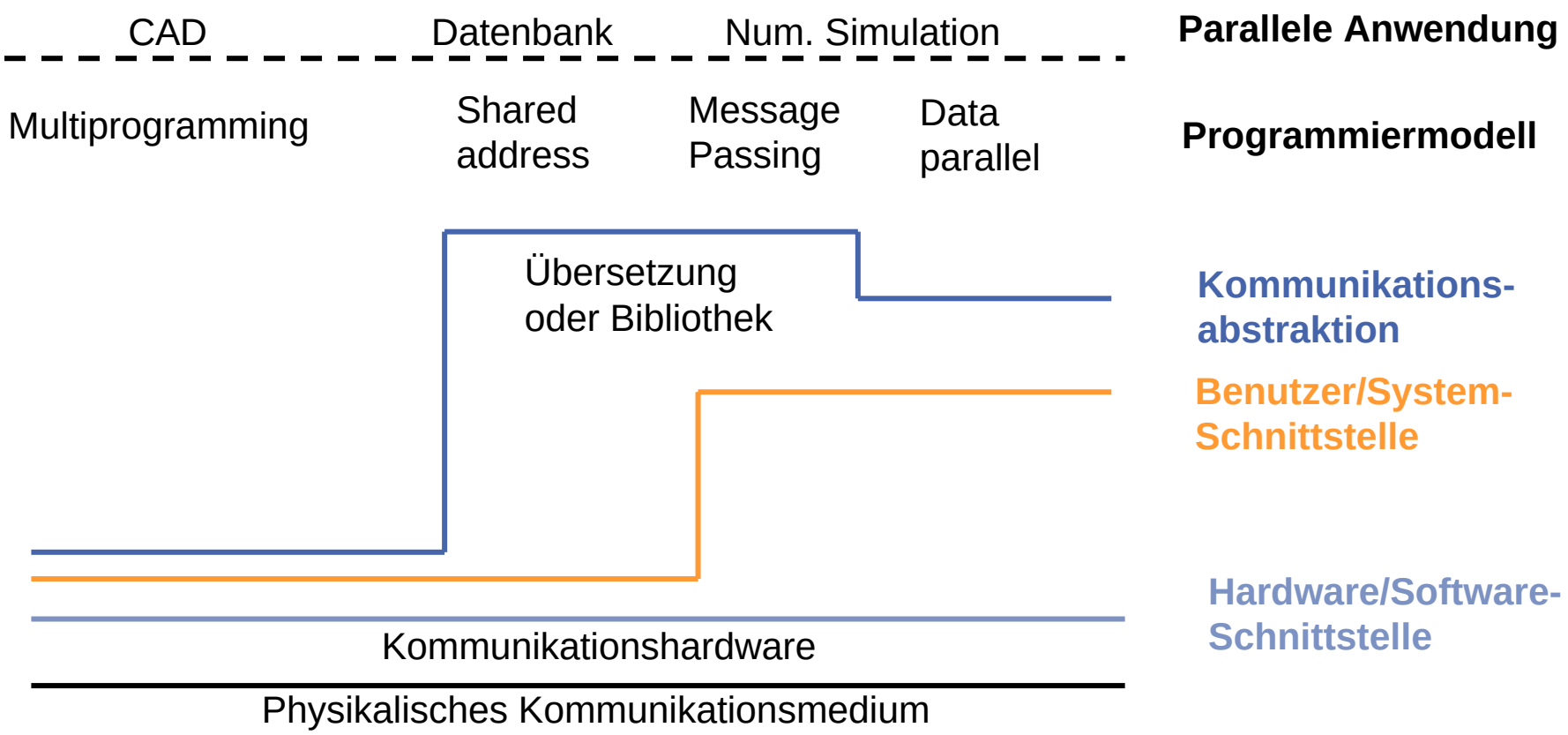
- **Rechnerarchitektur**
- Abstraktion
  - Benutzer-/System-Schnittstelle
  - Hardware-/Software-Schnittstelle
- Architektur
  - Spezifiziert die Menge der Operationen an den Schnittstellen und die Datentypen, auf denen diese operieren
- Organisation
  - Realisierung der Abstraktionen

Parallele Architekturen

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

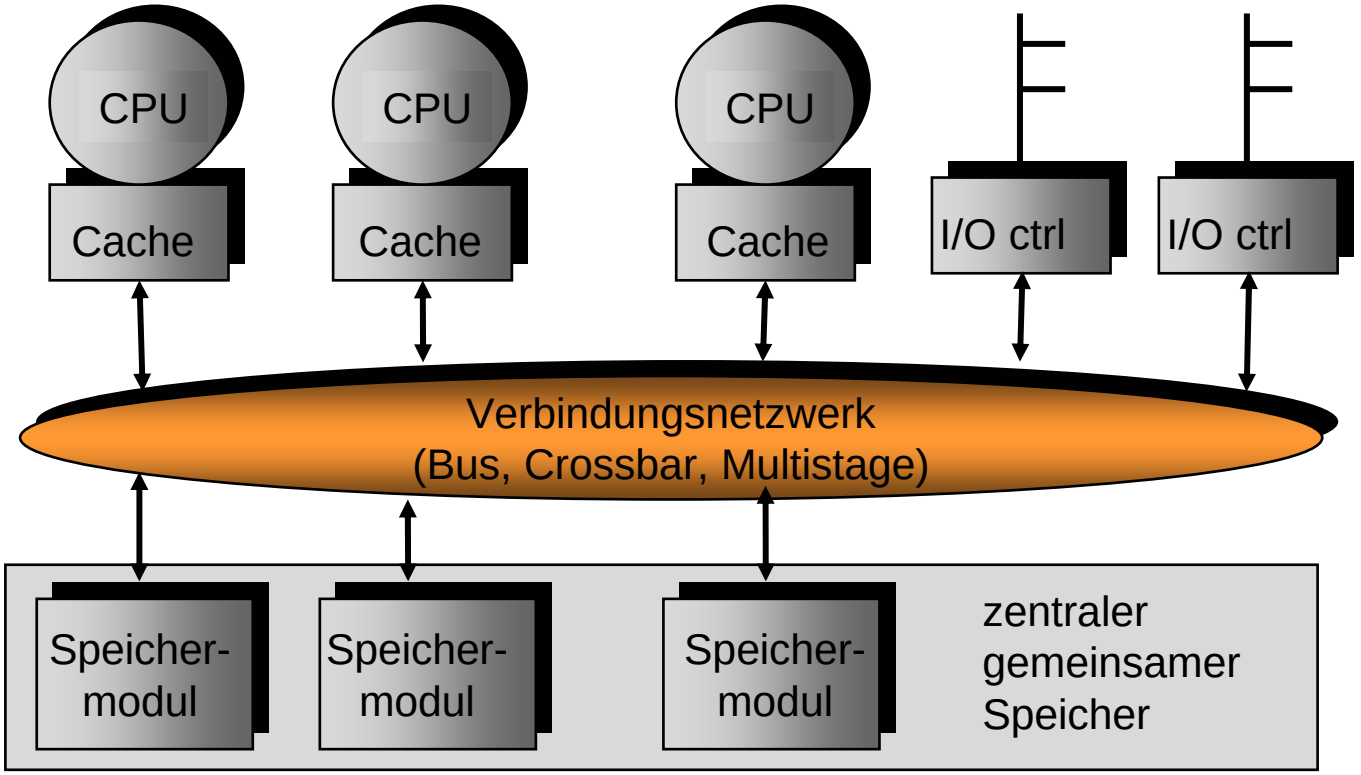
Parallele Architekturen

- **Abstraktion**



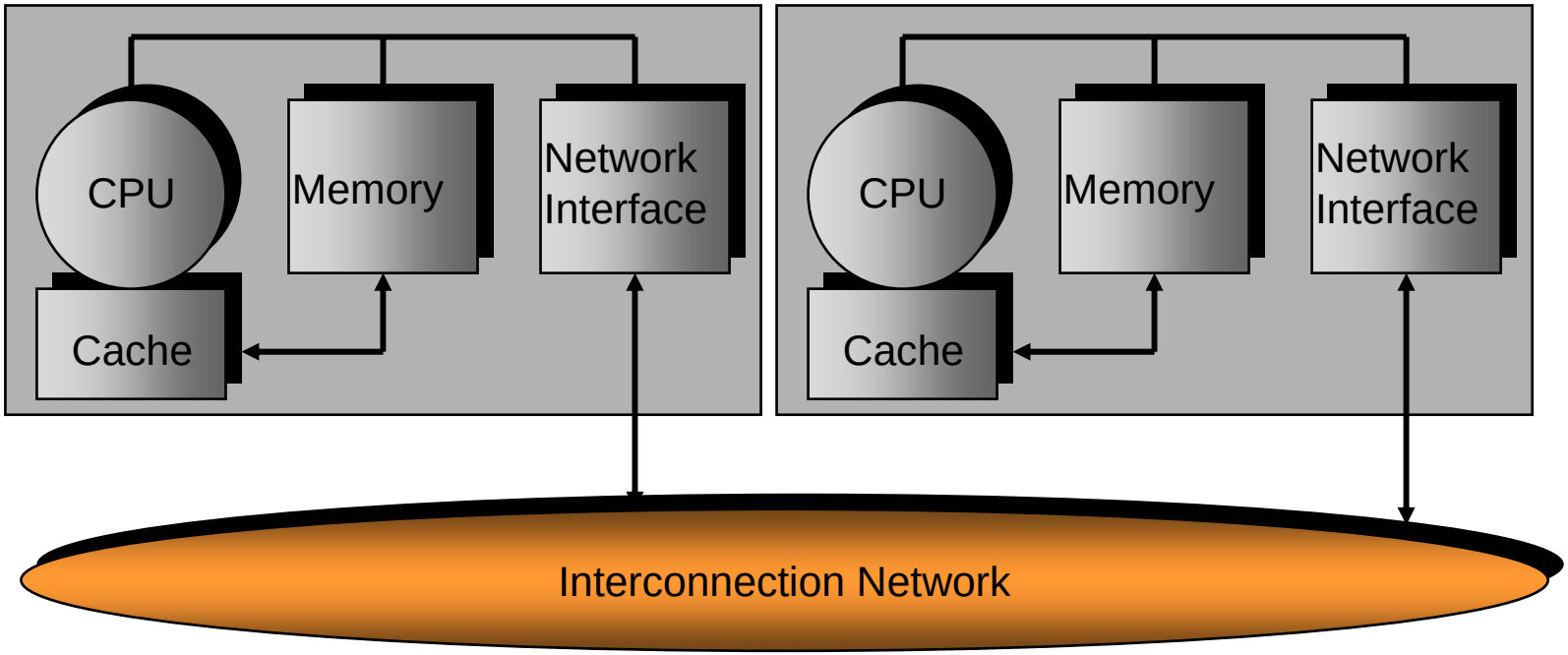
Parallele Architekturen

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



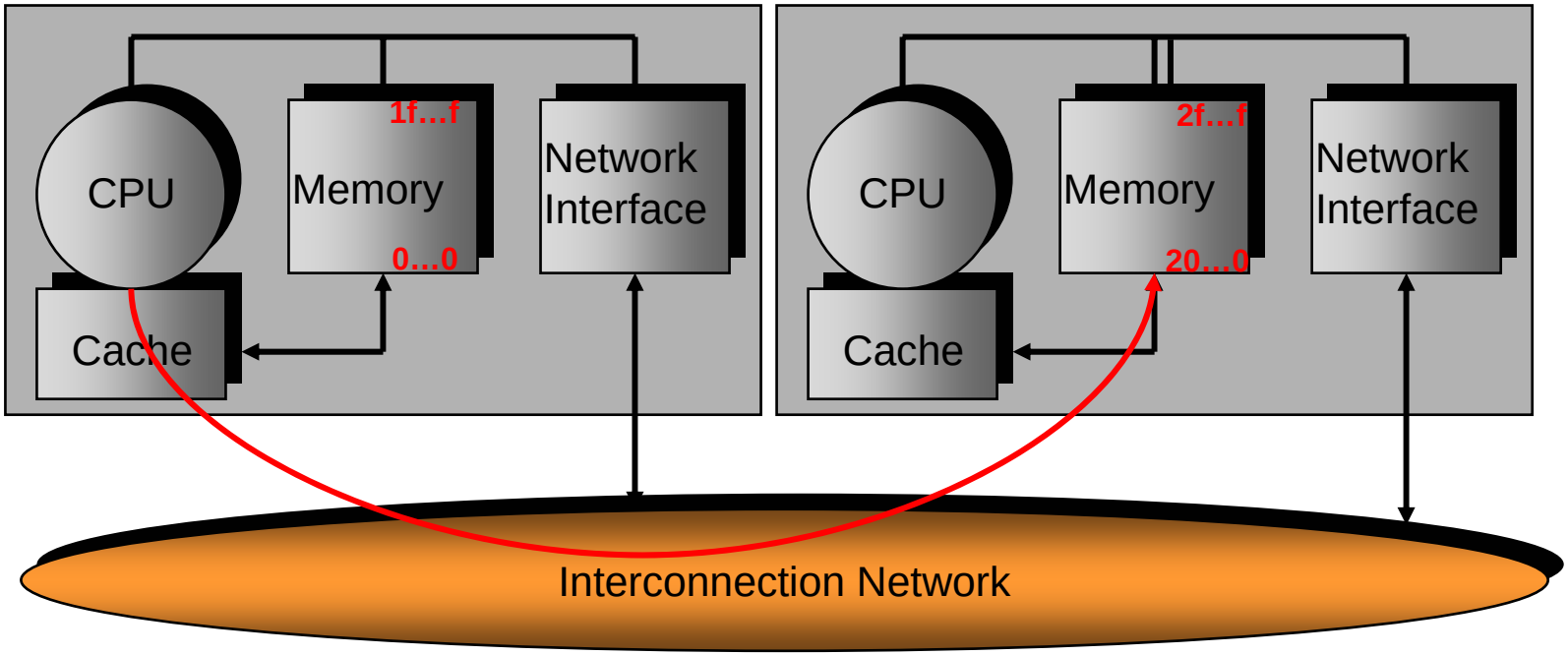
Parallele Architekturen

- **Multiprozessor mit verteiltem Speicher**
  - NORMA No Remote Memory Access
  - Beispiel: Cluster



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



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

Parallele Architekturen

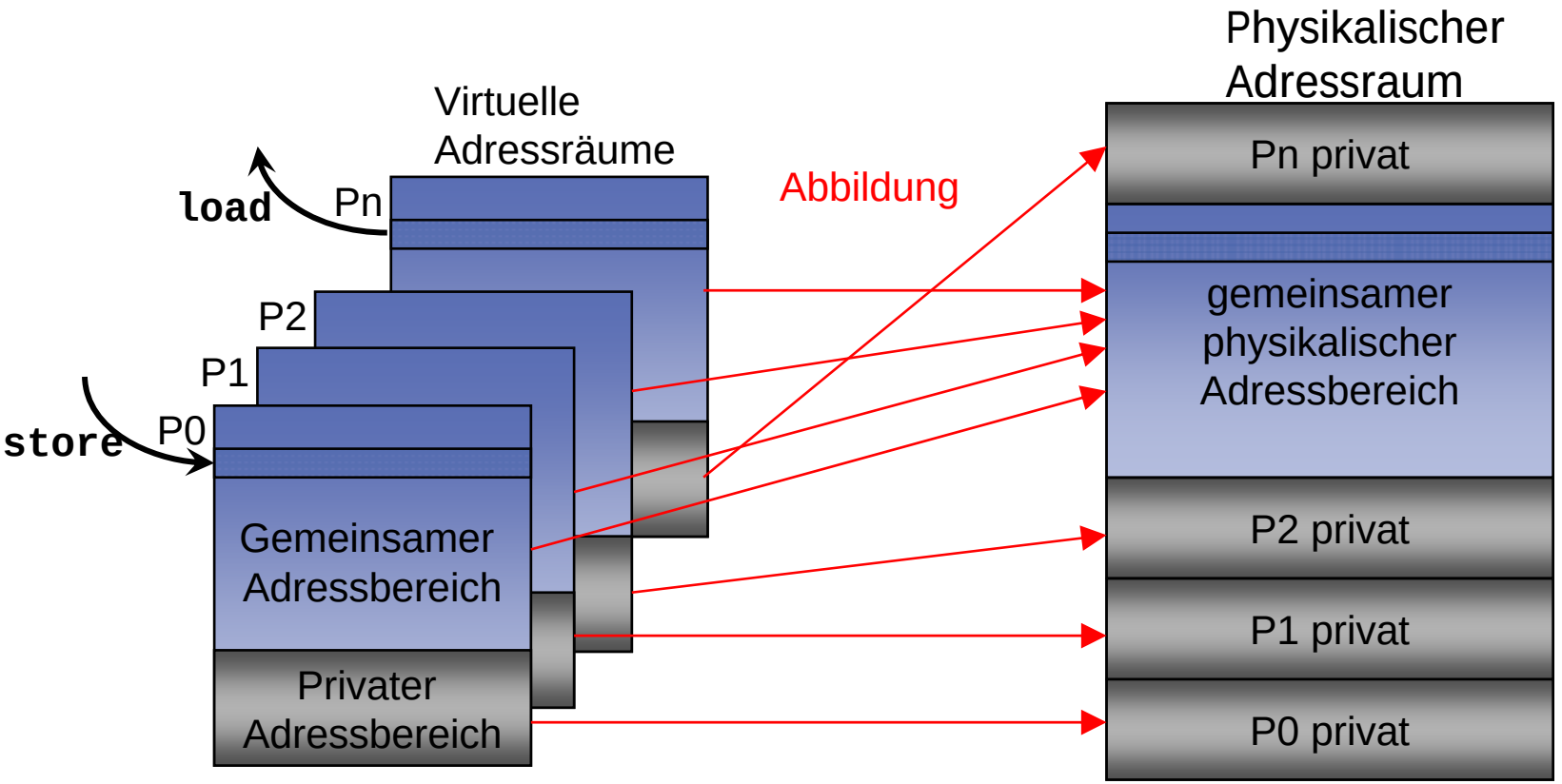
- **Programmiermodell**
- Multiprogramming
  - Menge von unabhängigen sequentiellen Programmen
  - Keine Kommunikation oder Koordination

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

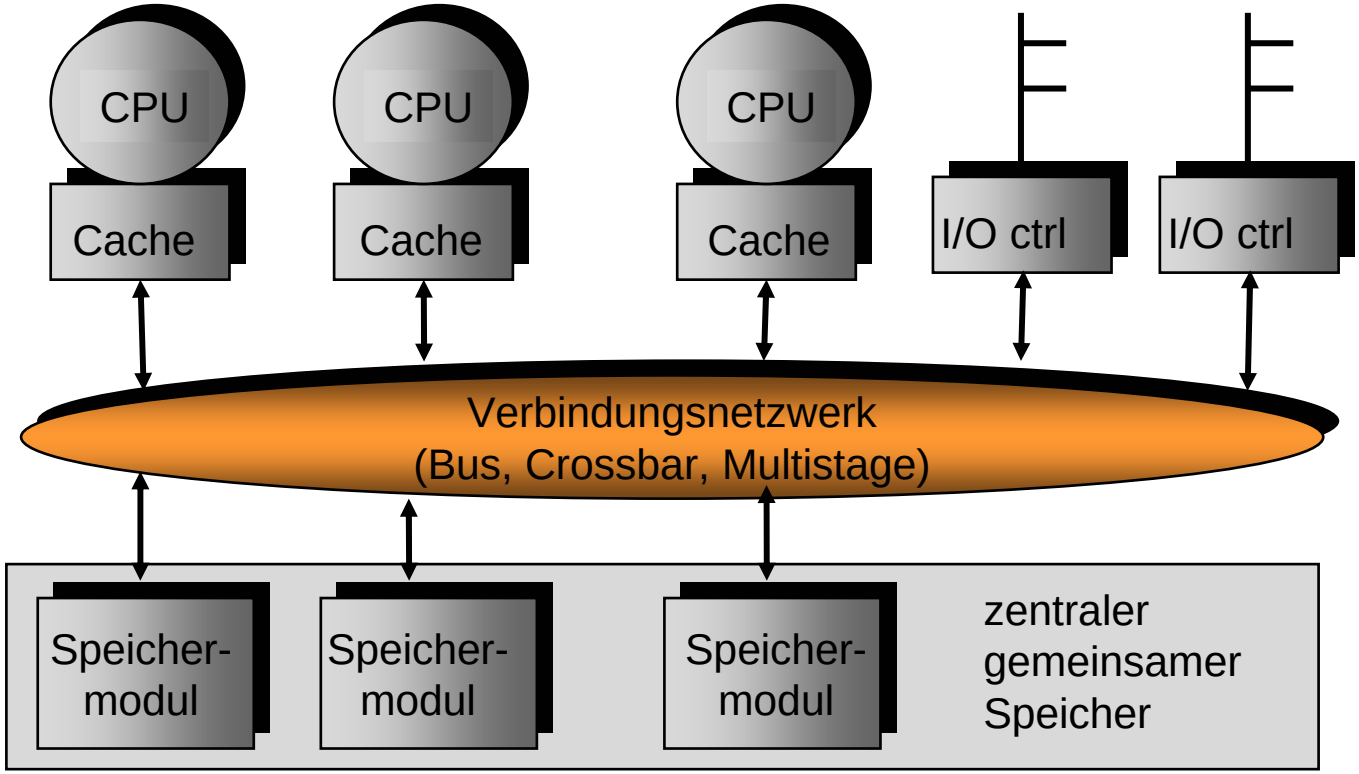
Parallele Architekturen

- **Programmiermodell**
- Gemeinsamer Speicher (Shared Memory)



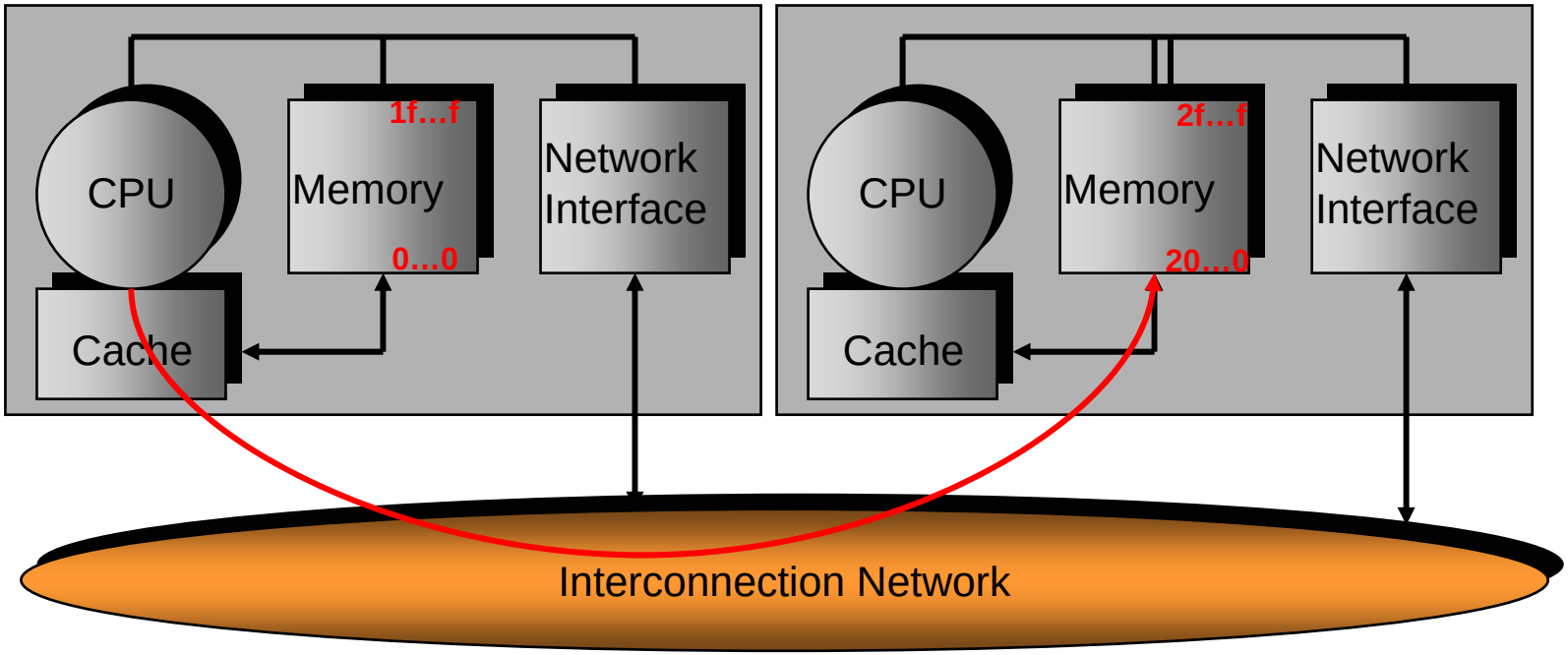
Parallele Architekturen

- **Multiprozessor mit gemeinsamem Speicher**
  - UMA: Uniform Memory Access



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

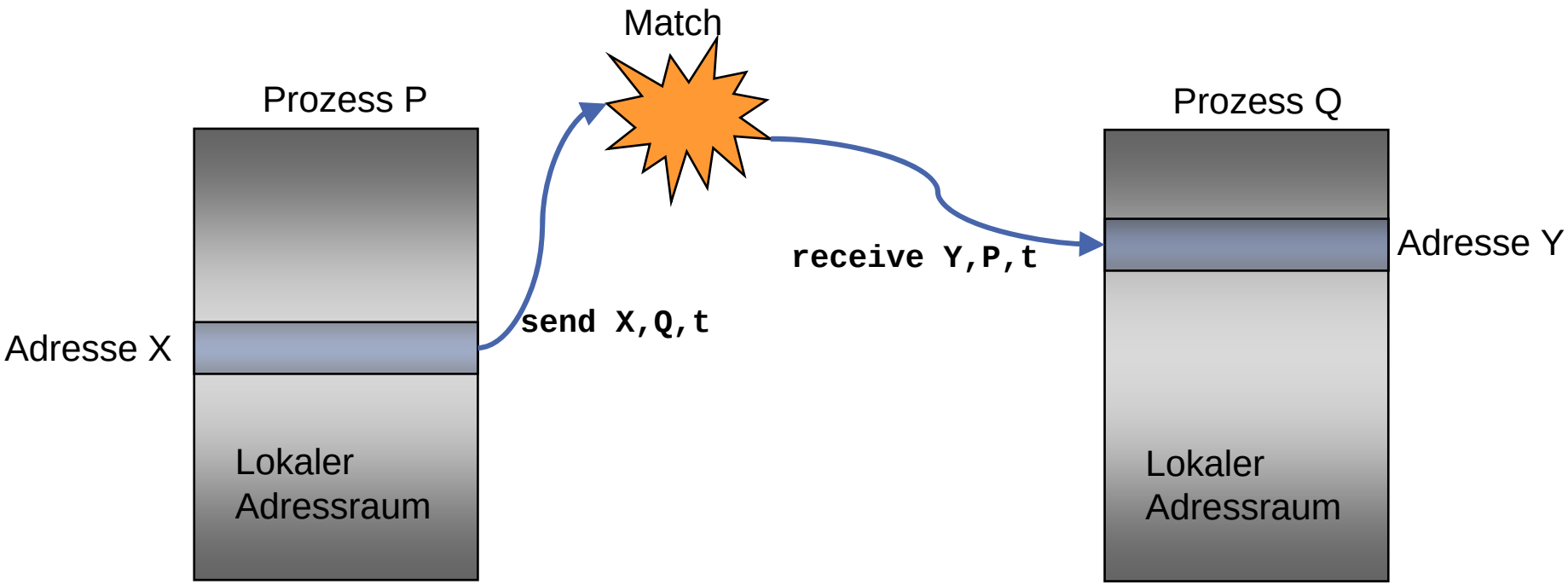


Parallele Architekturen

- **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

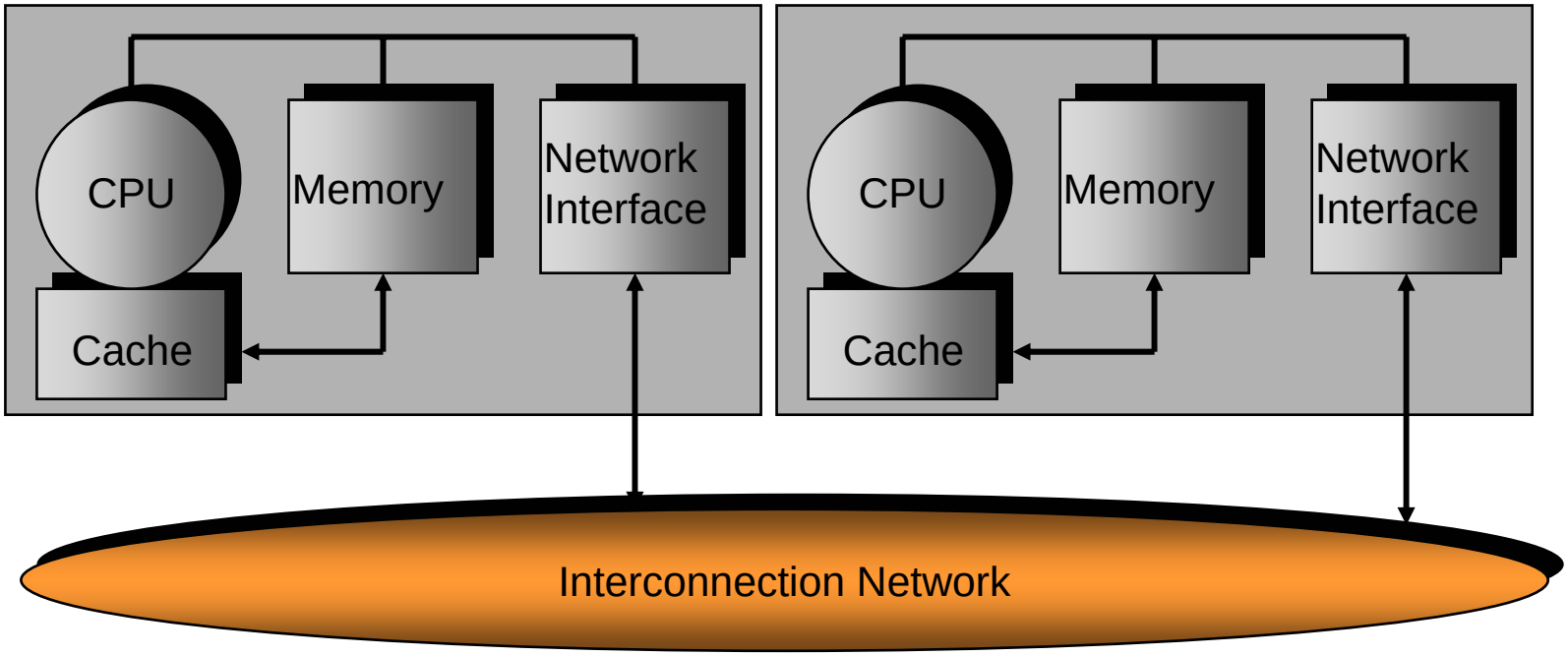
Parallele Architekturen

- **Programmiermodell**
- Nachrichtenorientiertes Programmiermodell (Message Passing)



Parallele Architekturen

- **Multiprozessor mit verteiltem Speicher**
  - NORMA No Remote Memory Access



Parallele Architekturen

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

Vorlesung Rechnerstrukturen

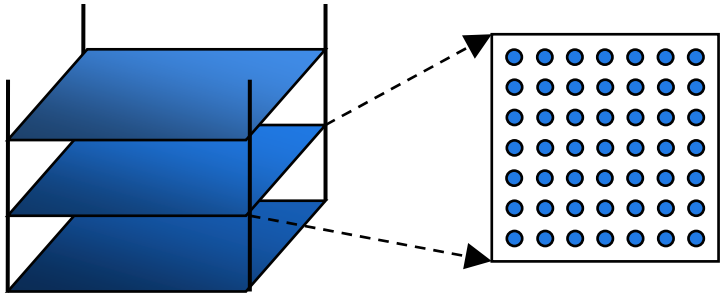
- **Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene**
- 3.3 Parallele Programmierung

Parallele Programmierung

- **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

Parallele Programmierung

- **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

Parallele Programmierung

- **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

Parallele Programmierung

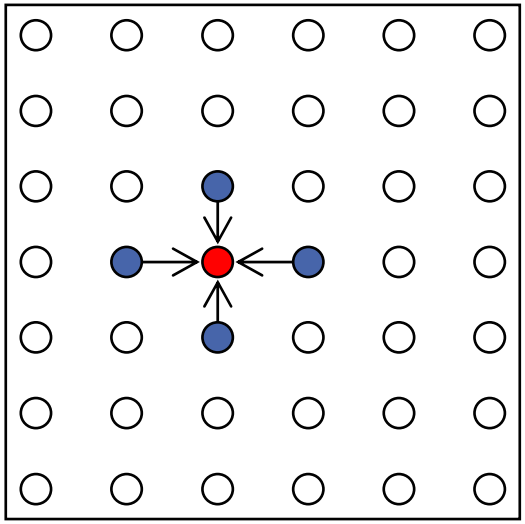
- **Fallstudie: OCEAN - Simulation der Ozean-Strömung**
- Lösung der Bewegungsgleichungen:
  - Je mehr Gitterpunkte verwendet werden, desto feiner ist die Auflösung der Diskretisierung und desto genauer ist die Simulation
  - Für einen Ozean wie den Atlantik, der etwa eine Fläche von 2000km x 2000km 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

Parallele Programmierung

- **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

Parallele Programmierung

- **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

Parallele Programmierung

- **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
```

Parallele Programmierung

- **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
```

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

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!

Parallele Programmierung

- **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!

Parallele Programmierung

- **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

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

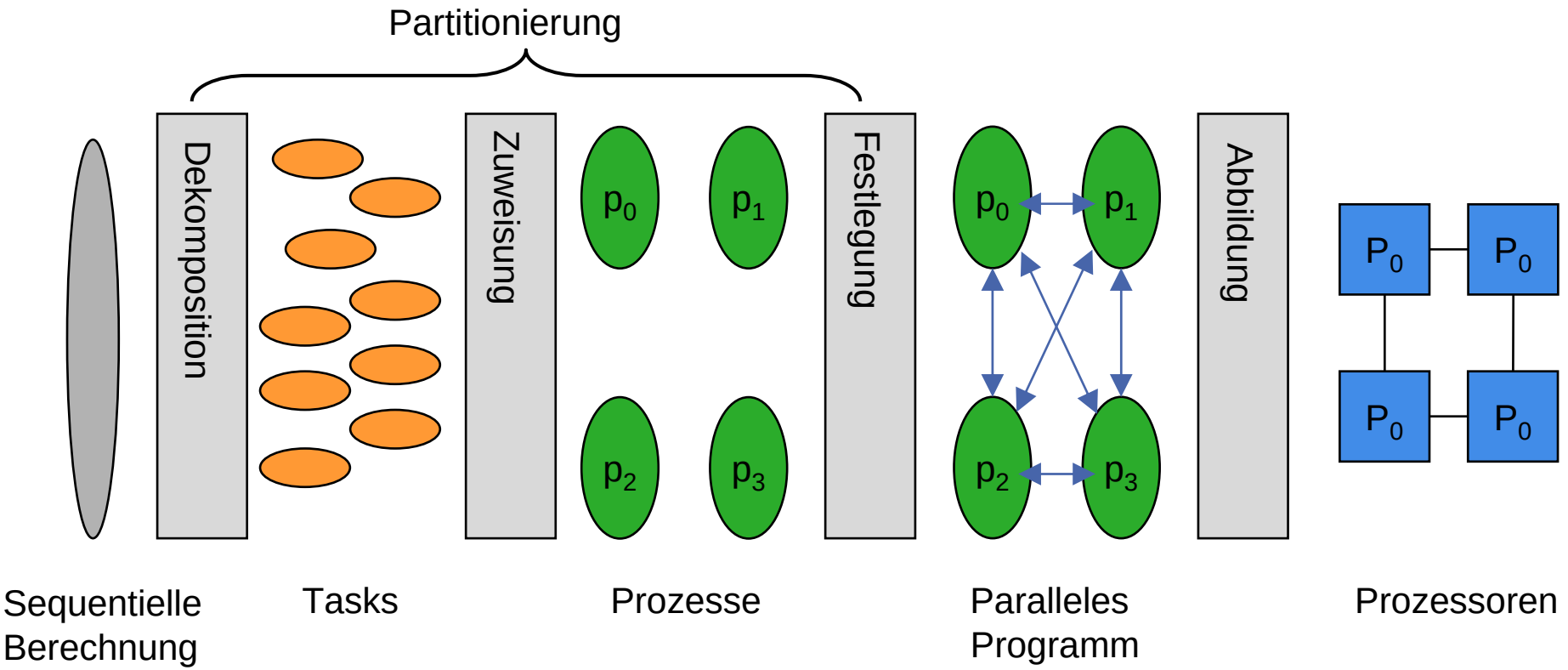
Parallele Programmierung

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

Partitionierung

Parallele Programmierung

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



Parallele Programmierung

- **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

Parallele Programmierung

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

Parallele Programmierung

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

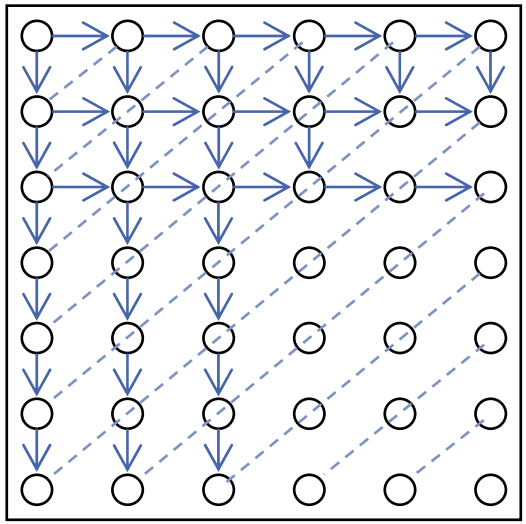
```
(1) while (!done) do
(2)    diff=0;
(3)    for i←1 to n do
(4)       for j←1 to n do
(5)          temp=A[i,j];
(6)          A[i,j]←0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j]);
(7)          diff += abs(A[i,j]–temp);
(8)       end for
(9)     end for
(10)     if (diff/(n*n) < TOL) then done=1;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

Parallele Programmierung

- **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

Parallele Programmierung

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



Abhängigkeiten

Verbinden Punkte, zwischen den keine Abhängigkeiten bestehen

Parallele Programmierung

- **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!

Parallele Programmierung

- **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

Parallele Programmierung

- **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

Parallele Programmierung

- **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

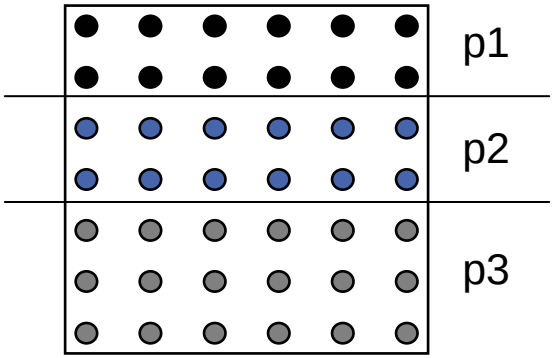
```
(1) while (!done) do
(2)     diff=0;
(3)     for_all i←1 to n do
(4)        for_all j←1 to n do
(5)           temp=A[i,j];
(6)           A[i,j]←0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j]);
(7)           diff += abs(A[i,j]–temp);
(8)        end for_all
(9)      end for_all
(10)     if (diff/(n*n)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

Parallele Programmierung

- **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

Parallele Programmierung

- **Parallelisierungsprozess**
- Zuweisung
  - Beispiel:



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

Parallele Programmierung

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

Parallele Programmierung

- **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

Parallele Programmierung

- **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

Parallele Programmierung

- **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

Vorlesung Rechnerstrukturen

- **Kapitel 3: Multiprozessoren – Parallelismus auf Prozess-/ Blockebene**
- 3.4 Quantitative Maßzahlen

Quantitative Maßzahlen

- **Ausführungszeit**
  - Die (Gesamt-)Ausführungszeit T (Execution Time) eines parallelen Programms ist die Zeit zwischen dem Starten der Programmausführung auf einem der Prozessoren bis zu dem Zeitpunkt, an dem der letzte Prozessor die Arbeit an dem Programm beendet hat.
  - Zu beachten:
    - Prozessorzustand:
    - Während der Programmausführung sind alle Prozessoren in einem der drei Zustände:
      - rechnend
      - kommunizierend
      - untätig

Quantitative Maßzahlen

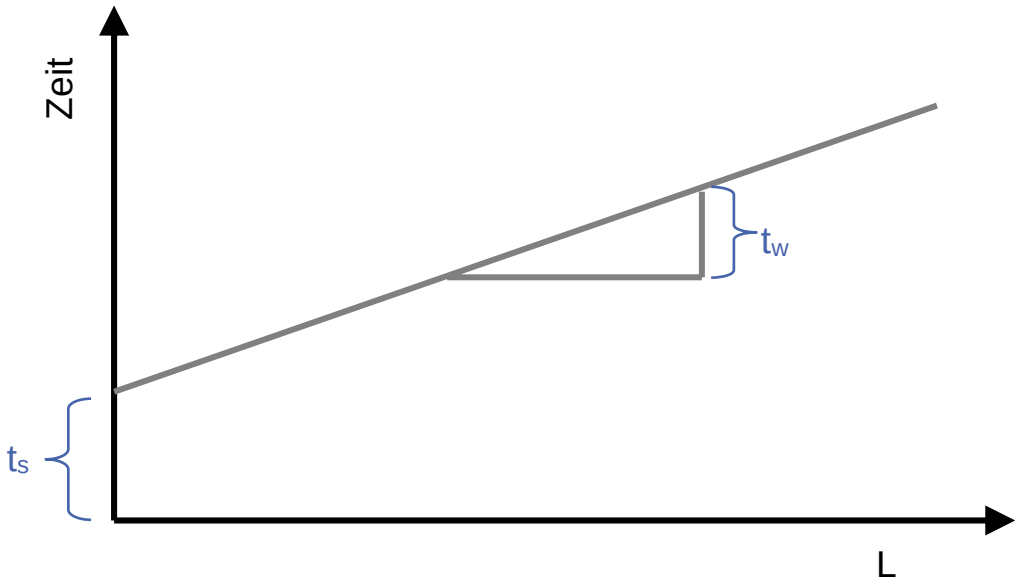
- **Ausführungszeit**
- Die Ausführungszeit eines parallelen Programms auf einem dediziert zugeordneten Parallelrechner setzt sich zusammen aus:
  - Berechnungszeit T_{comp} (Computation Time)
    - Die Zeit, die für die Rechenoperationen verwendet wird
  - Kommunikationszeit T_{comm} (Communication Time)
    - Die Zeit, die für Sende- und Empfangsoperationen verwendet wird
  - Untätigkeitszeit T_{idle} (Idle Time)
    - Die Zeit, die mit Warten (auf zu empfangene Nachrichten oder und zu versendende) verbraucht wird
- Es gilt:
  - $T = T_{comp} + T_{comm} + T_{idle}$

Quantitative Maßzahlen

- **Übertragungszeit einer Nachricht T_{msg}**
  - die Zeit, die für das Verschicken einer Nachricht von einer bestimmten Länge zwischen zwei Prozessoren benötigt wird
  - Die Übertragungszeit setzt sich zusammen aus:
    - der Startzeit t_s (Message Startup Time):
      - Die Zeit, die benötigt wird, um die Kommunikation zu initiieren
    - Transferzeit t_w pro übertragenem Datenwort:
      - hängt von der physikalischen Bandbreite des Kommunikationsmediums ab.
    - Voraussetzung:
      - Verbindungsnetz ist konfliktfrei, da sonst die Übertragungszeit nicht fest berechnet werden kann

Quantitative Maßzahlen

- **Übertragungszeit einer Nachricht T_{msg}**
- Formel:
  - $T_{msg} = t_s + t_w * L$, mit L: Anzahl der Datenwörter



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.