

# **Synthese Eingebetteter Systeme**

Wintersemester 2012/13

## **14 – Abbildung von Anwendungen auf Multicore-Systeme**

Michael Engel  
Informatik 12  
TU Dortmund

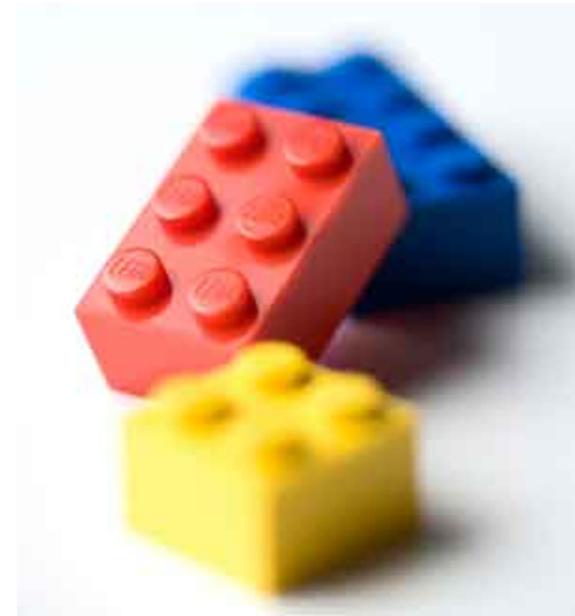
2012/12/19

---

# Abbildung von Anwendungen auf Multicores

---

- **Multicore-Architekturen**
  - Entwurfsraum
  - Grenzen der Parallelität
- Beispiele für Architekturen
- Parallelität
  - Granularität und Lastverteilung
  - Explizit vs. implizit
  - Beispiele
- Abbildung von Anwendungen
  - Überblick
  - Zielsetzungen



---

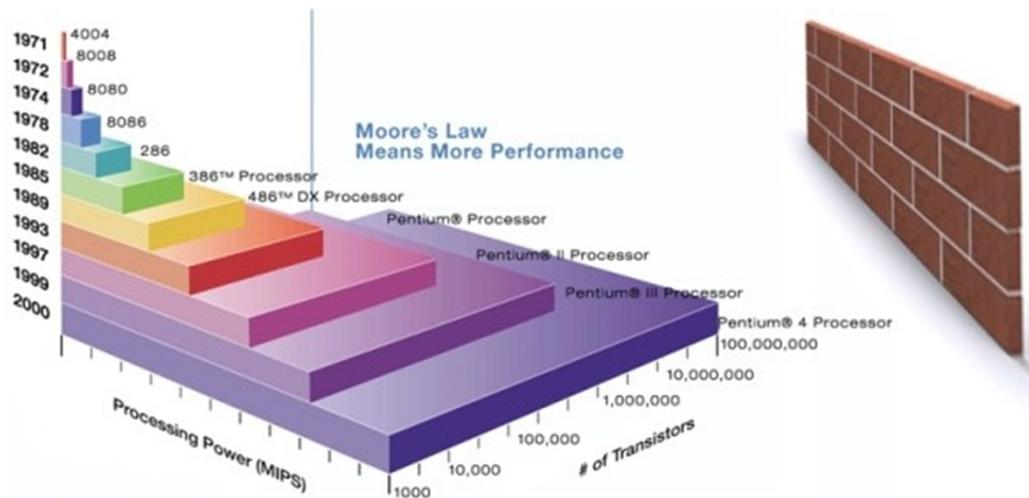
# Aktuelle Trends eingebetteter Systeme

---

- Problem
  - Immer höhere Rechenleistungen auch (und gerade) in eingebetteten Systemen benötigt
- z.B. in der Signalverarbeitung
  - Video-De- und Encoding
  - Bilderkennung (z.B. Automotive)
  - Spracherkennung

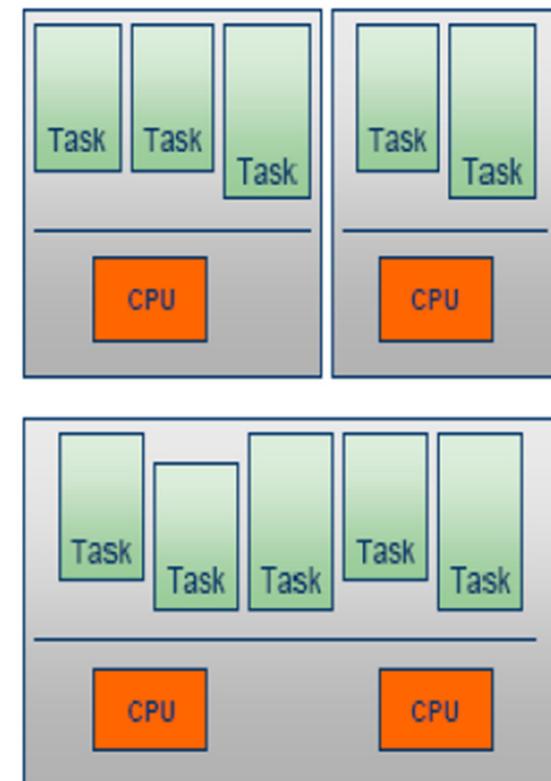
# Moore'sches Gesetz – die Lösung?

- **Moore's Law**
  - Verdoppelung der Anzahl der Transistoren alle 18 Monate
- Wie lange noch gültig?
  - Taktfrequenz erreicht physikalische Grenzen
- Wofür verwendet man diese große Menge Transistoren?
  - Klassisch: Ein Core, (sehr) viel Cache, Superskalar
  - Multicores: m Cores mit n/m MHz vs. 1 Core mit n MHz vs.



# Multicore: Begriffe

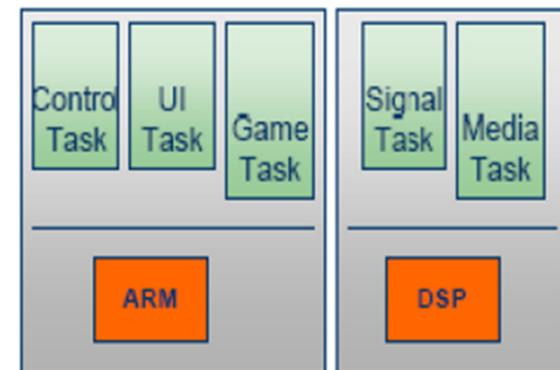
- AMP (Asymmetrischer MP)
  - Jeder Prozessor besitzt lokalen Speicher
  - Tasks meist statisch auf einen Prozessor abgebildet
- SMP (Symmetrischer MP)
  - Prozessoren teilen sich Speicher
  - Tasks auch dynamisch auf beliebige Prozessoren abbildbar



# Multicore: Begriffe

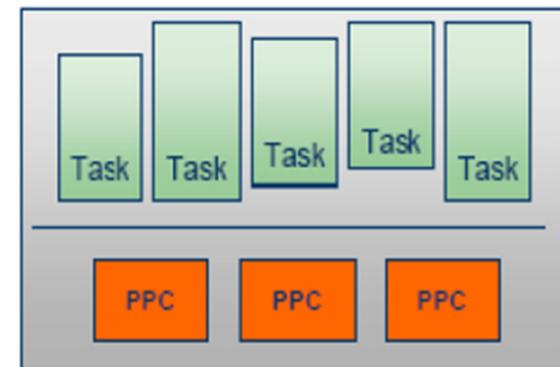
## Heterogen:

- Spezialisierung der Prozessoren
- Oft unterschiedliche Befehlssätze
- Meist asymmetrische MPs

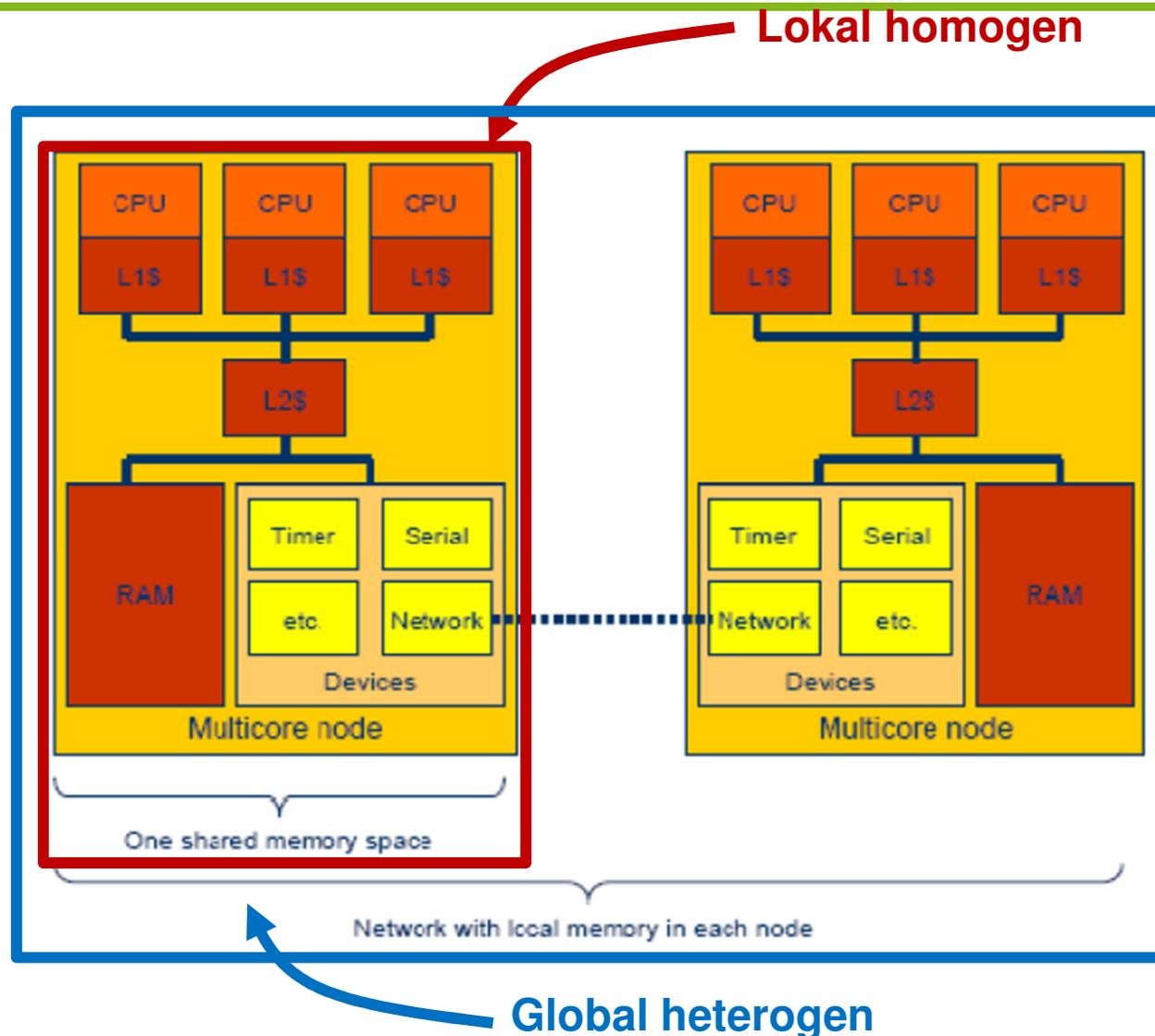


## Homogen:

- Alle Prozessoren haben gleichen Befehlssatz
- Prozessoren können jeden Task ausführen
- Meist symmetrische MPs



# Zukünftige Manycores



---

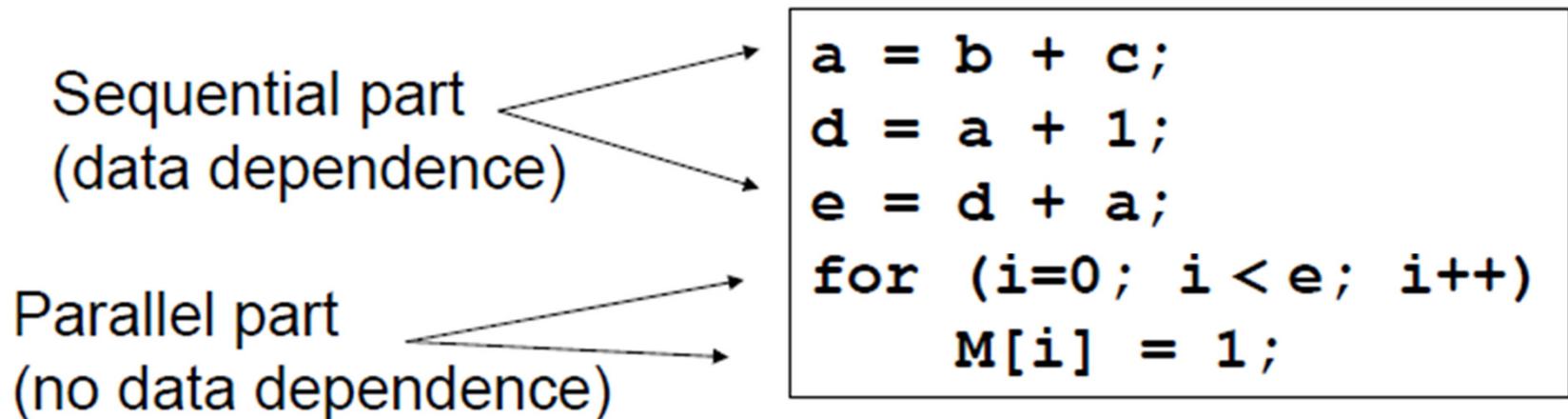
# Gründe für Multicores

---

- Mehr Parallelität! Auf Instruktionsebene schwer zu finden
- Kleine Verbesserungen erfordern komplexe Entwürfe
  - Ausnutzung von grobgranularerer Parallelität
- Steigerung der Taktfrequenzen verlangsamt sich
  - Zu großer Energieverbrauch, zu viel Abwärme
- Kommunikation auf Chip nicht schnell genug
  - Entwurf kleiner, lokaler Einheiten mit kurzen Pfaden
  - Effiziente Ausnutzung von Milliarden Transistoren
  - Vielfache Wiederverwendung von Basiskomponenten
  - Gute Skalierungsmöglichkeiten
    - *Einfach Cores für höhere Leistung hinzufügen?*

# Abdeckung

- Nicht alle Programme sind trivial (“embarrassingly”) parallel
- Programme bestehen aus sequentiellen und parallelen Abschnitten



---

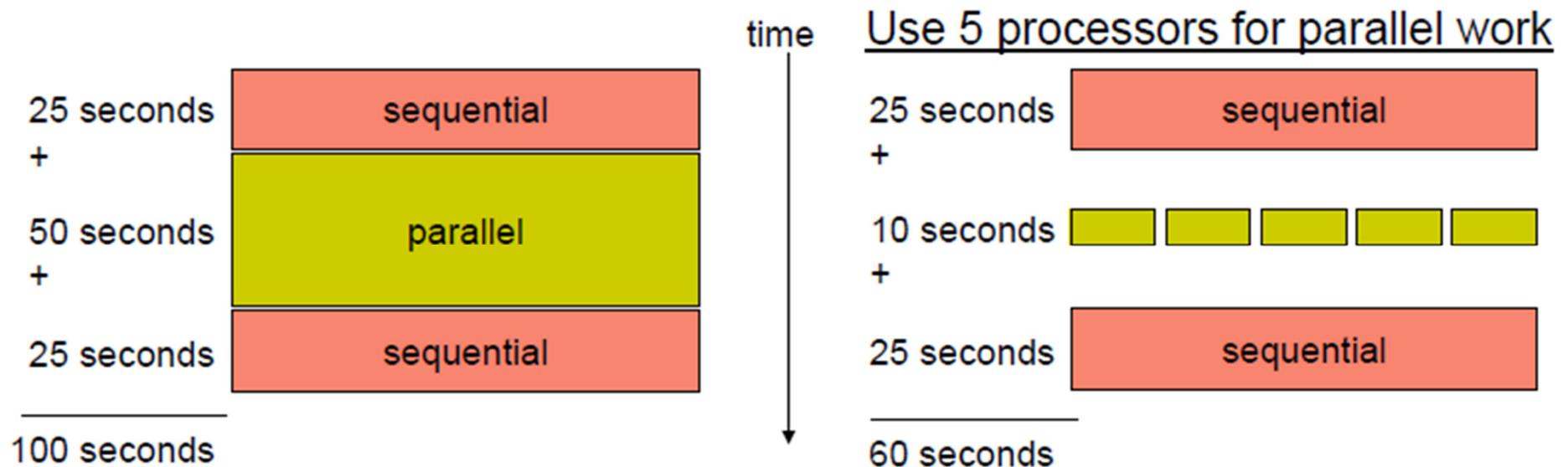
# Amdahl's Law

---

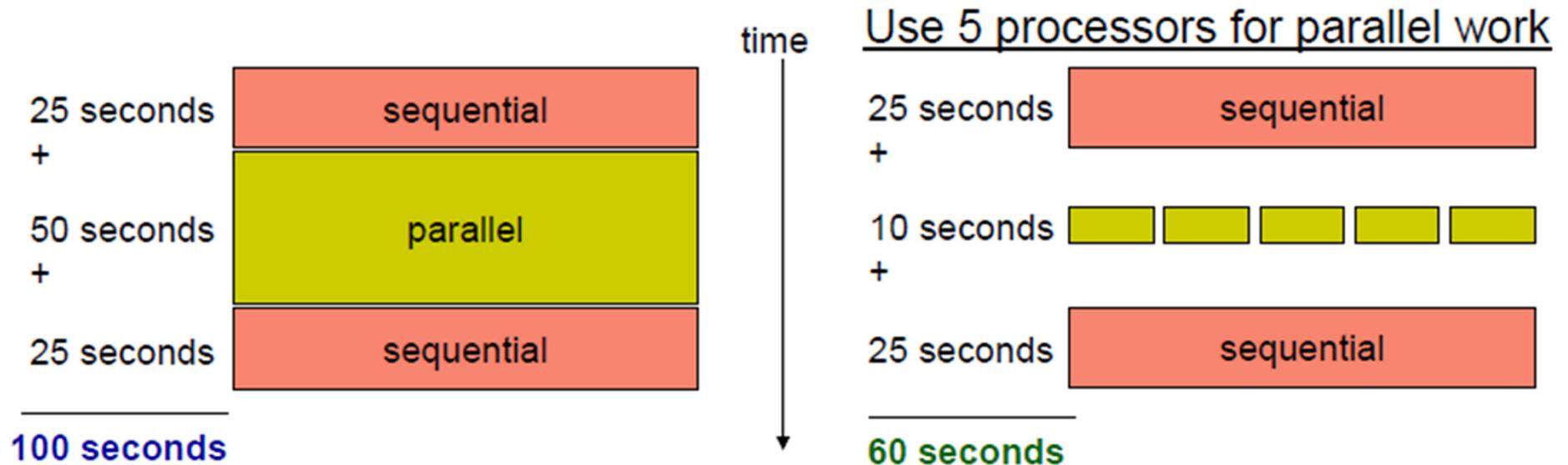
- **Amdahl's Law:** *Die Leistungssteigerung, die durch einen schnellen Modus der Ausführung erreichbar ist, ist durch den Anteil der Zeit begrenzt, in der dieser schnellere Modus der Ausführung verwendet werden kann.*

# Amdahl's Law

- Erzielbare Programmbeschleunigung (speedup) ist begrenzt durch den Anteil an parallelisierbarem Code eines Programms



# Amdahl's Law



- Speedup = alte Laufzeit / neue Laufzeit

$$= 100 \text{ s} / 60 \text{ s}$$

$$= 1.67$$

(parallele Version ist 1,67-mal schneller)

# Amdahl's Law

- $p$  = fraction of work that can be parallelized
- $n$  = the number of processor

$$\text{speedup} = \frac{\text{old running time}}{\text{new running time}}$$

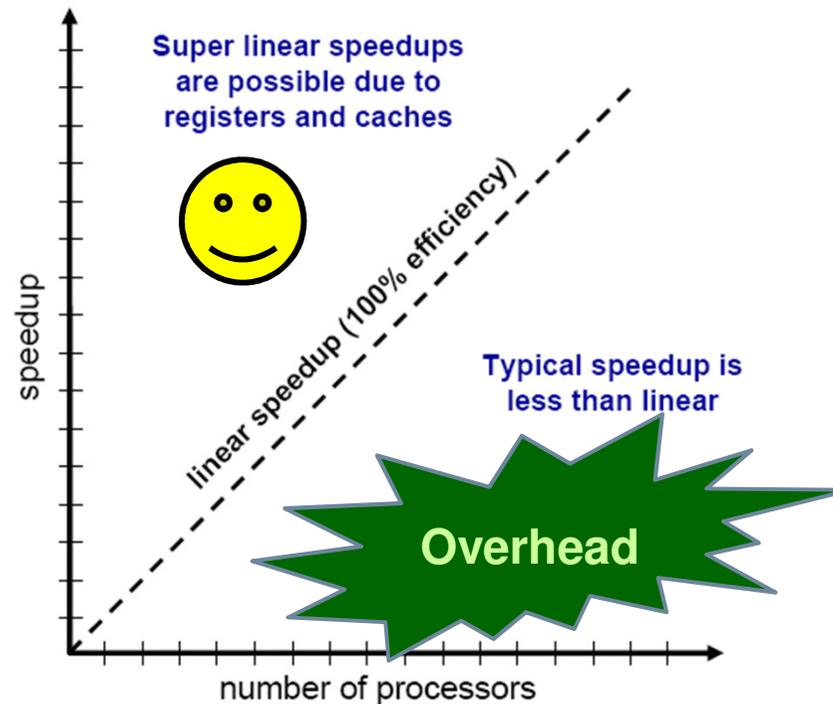
$$= \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

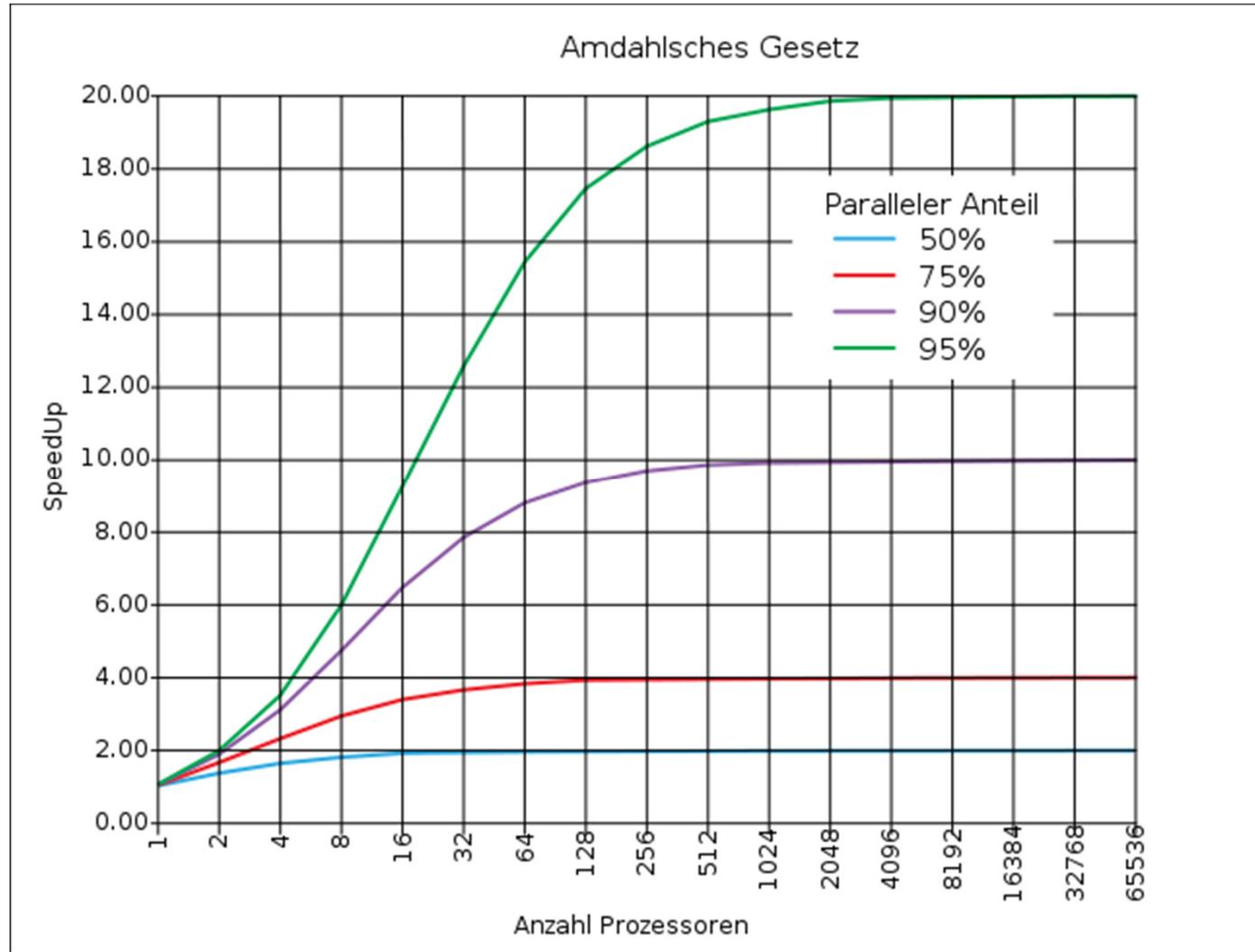
fraction of time to complete parallel work

# Amdahl's Law

- Speedup nähert sich  $1/(1-p)$ , wenn Anzahl der Prozessoren gegen  $\infty$  geht
- Parallele Programmierung lohnt sich, wenn Programme viele parallele Aufgaben zu erfüllen haben



# Grenzen der Parallelität

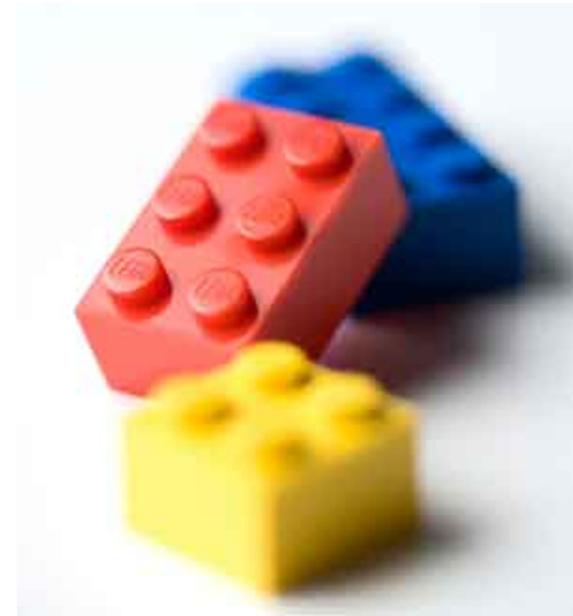


---

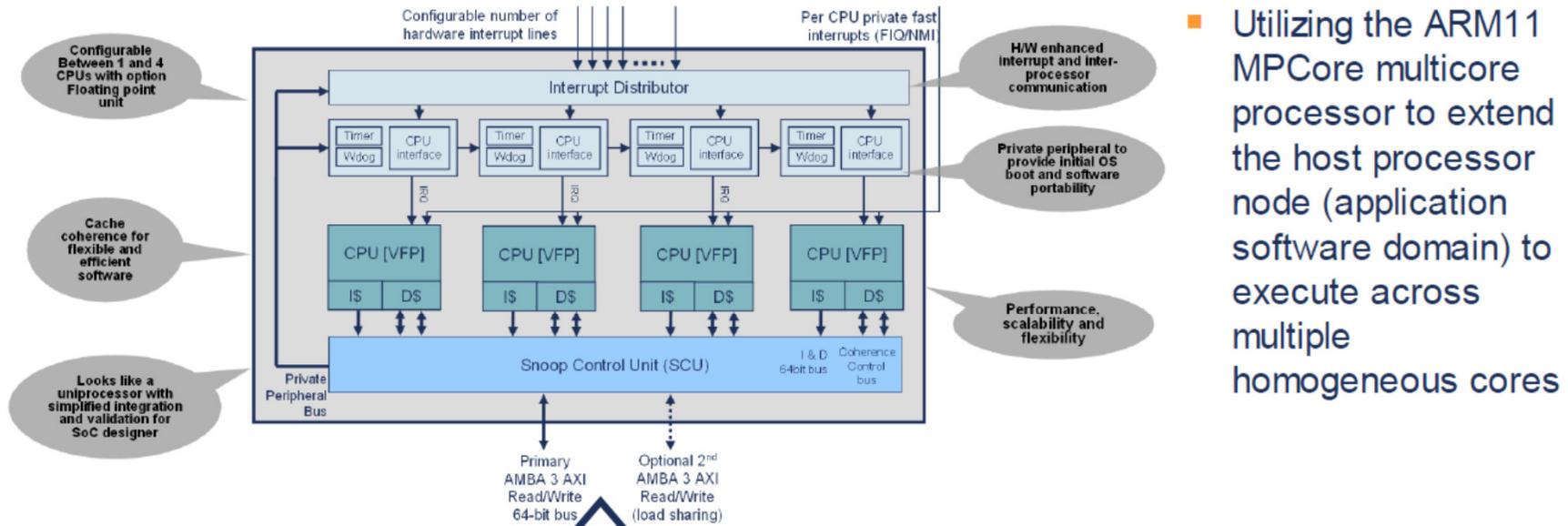
# Abbildung von Anwendungen auf Multicores

---

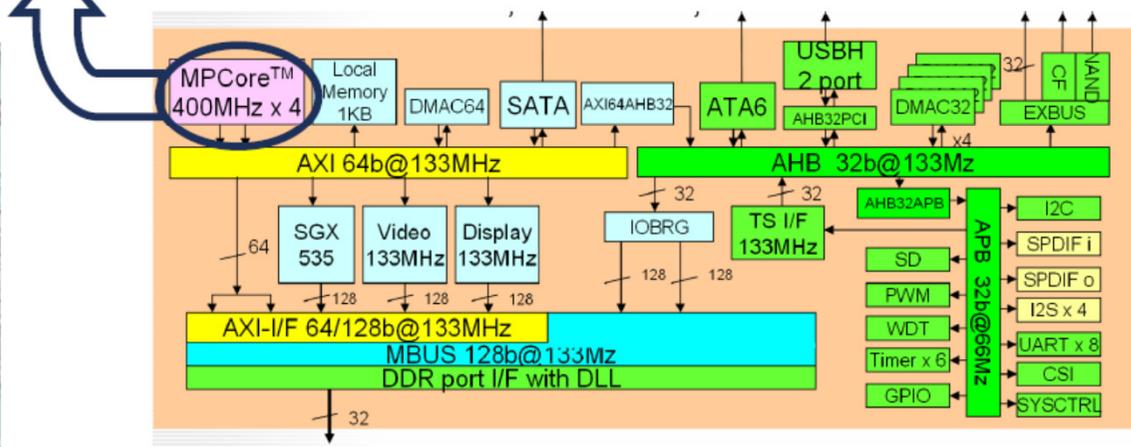
- Multicore-Architekturen
  - Entwurfsraum
  - Grenzen der Parallelität
- Beispiele für Architekturen
- Parallelität
  - Granularität und Lastverteilung
  - Explizit vs. implizit
  - Beispiele
- Abbildung von Anwendungen
  - Überblick
  - Zielsetzungen



# Homogene Architektur



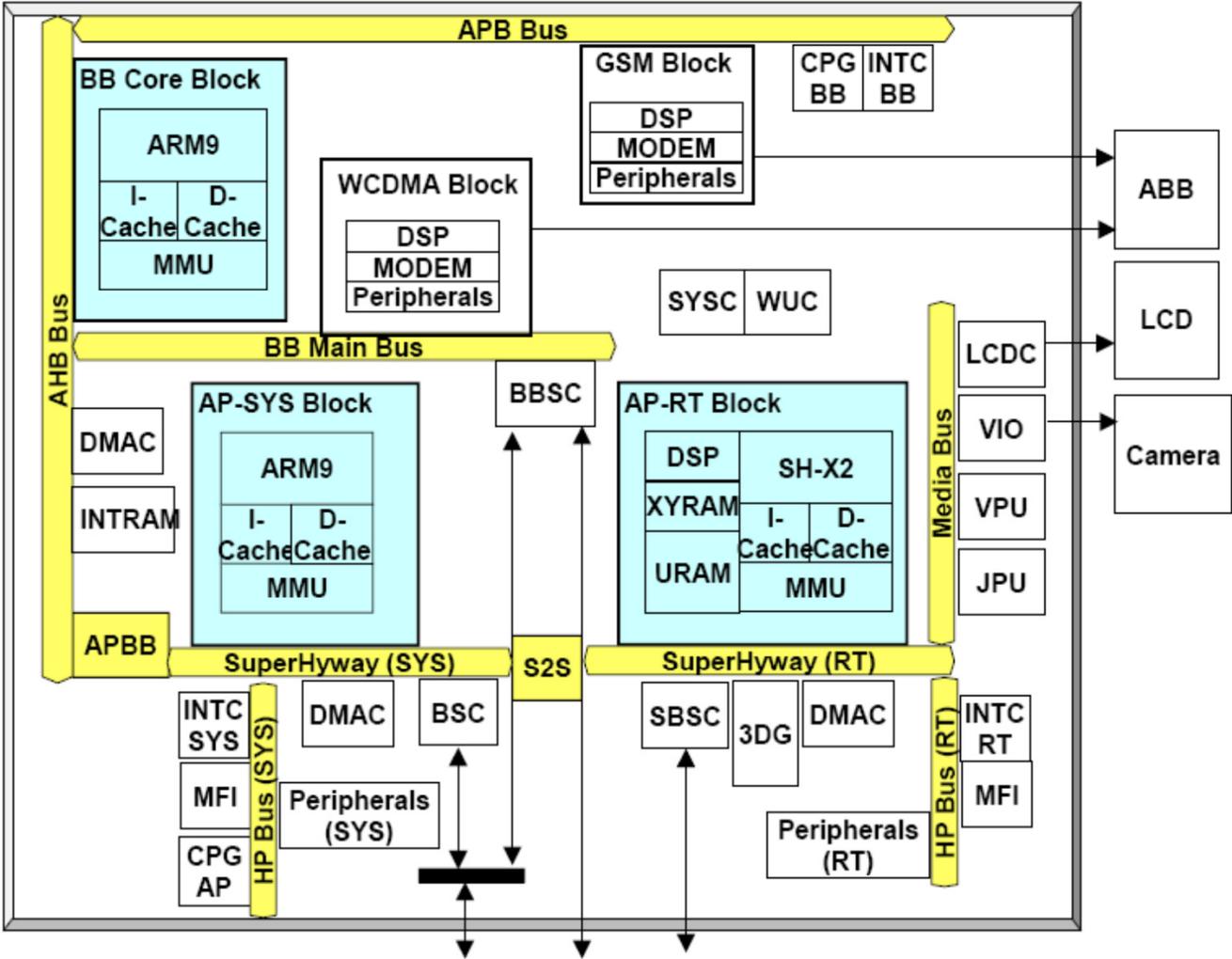
- Utilizing the ARM11 MPCore multicore processor to extend the host processor node (application software domain) to execute across multiple homogeneous cores



NaviEngine®の特徴 NEC

# Heterogene Architecturen

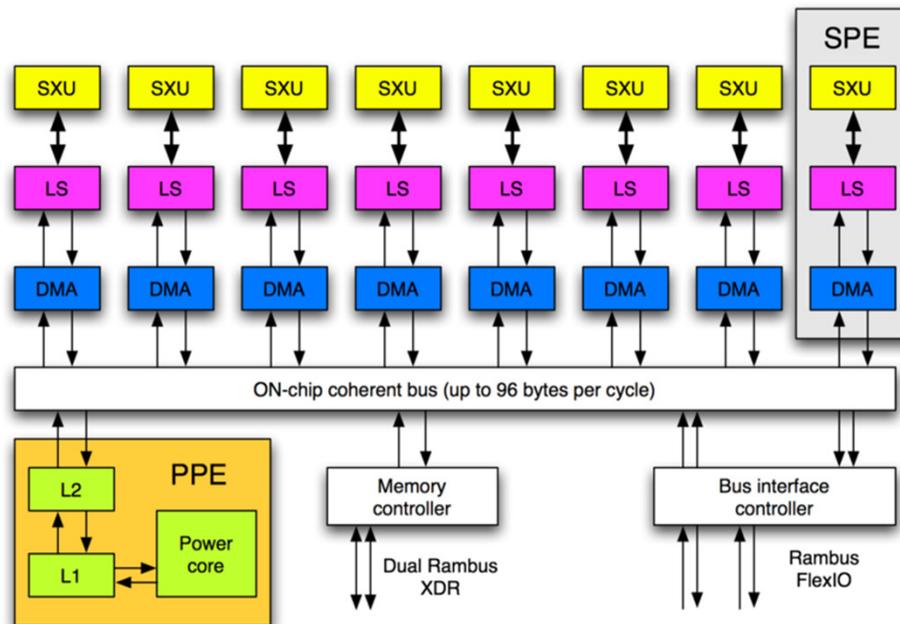
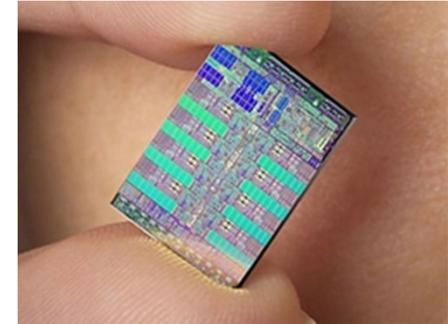
## G1 Module Diagram



<http://www.mpsoc-forum.org/2007/slides/Hattori.pdf>

# Heterogenes MPSoC: CELL

- IBM/Toshiba/Sony CELL-Prozessor
  - Entwicklungsstart: März 2001
  - Entwicklungsdauer: ca. 4 Jahre
  - > 400 beteiligte Entwickler
  - Entwicklungskosten:  
**400 Millionen US\$**



---

# CELL: Details

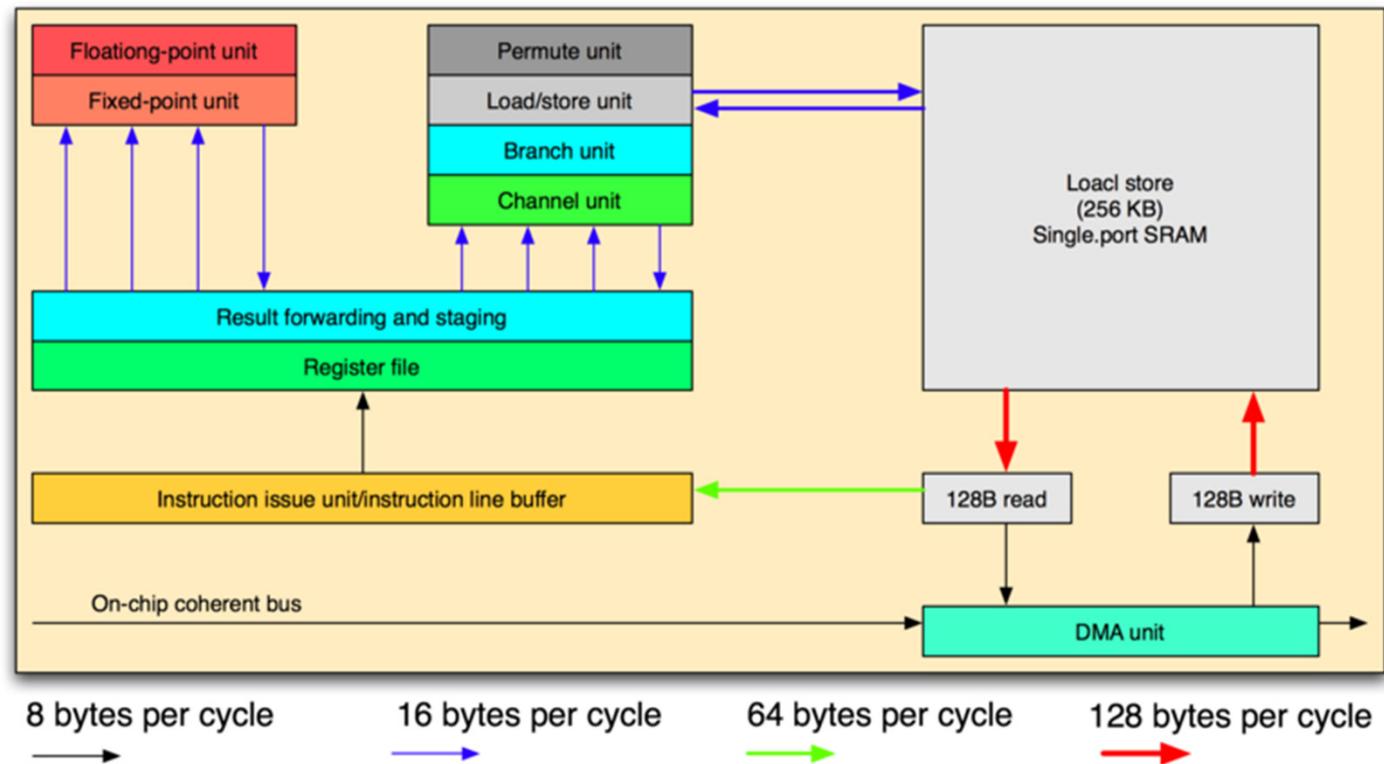
---

Technische Daten des CELL:

- **241 Millionen Transistoren** auf 235mm<sup>2</sup> Chipfläche (90nm), 3.2 GHz
- Acht Synergistic Processing Elements (SPE)
  - Je eine Recheneinheit (ALU) mit vierfachem SIMD
  - 128 Register, die jeweils 128 Bit groß sind
  - Memory Flow Controller (MFC): DMA-Transfers
  - Lokaler Speicher von 256 kB
  - Je **21 Millionen Transistoren**, davon ~2/3 für RAM
- Ein PowerPC Processing Element (PPE)
  - 64-Bit-PowerPC-Architektur von IBM
  - In-Order-Ausführung, zwei Threads gleichzeitig
  - 512 kB L2-Cache
- Element Interconnect Bus (EIB)
  - Ringstruktur
  - Bis zu 96 Byte pro Takt übertragbar

# CELL: Details (2)

- Struktur der CELL SPU:

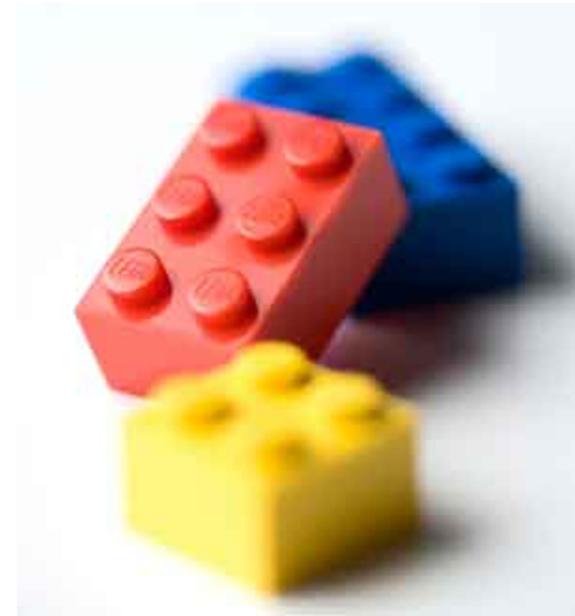


---

# Abbildung von Anwendungen auf Multicores

---

- Multicore-Architekturen
  - Entwurfsraum
  - Grenzen der Parallelität
- Beispiele für Architekturen
- Parallelität
  - Granularität und Lastverteilung
  - Explizit vs. implizit
  - Beispiele
- Abbildung von Anwendungen
  - Überblick
  - Zielsetzungen



---

# Granularität

---

- Granularität ist ein qualitatives Maß, das das Verhältnis von Berechnung zu Kommunikation ausdrückt
- Berechnungsphasen sind meist von Kommunikationsphasen durch Synchronisationsereignisse getrennt

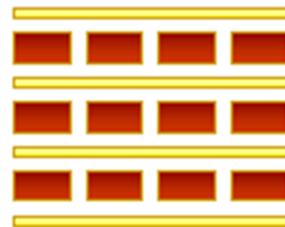
---

# Granularität

---

- Feingranulare Parallelität

- Niedriges Verhältnis von Berechnung zu Kommunikation
- Kurze Berechnungsphasen zwischen Kommunikationsphasen
- Geringere Leistungssteigerung möglich
- Hoher Kommunikations-overhead



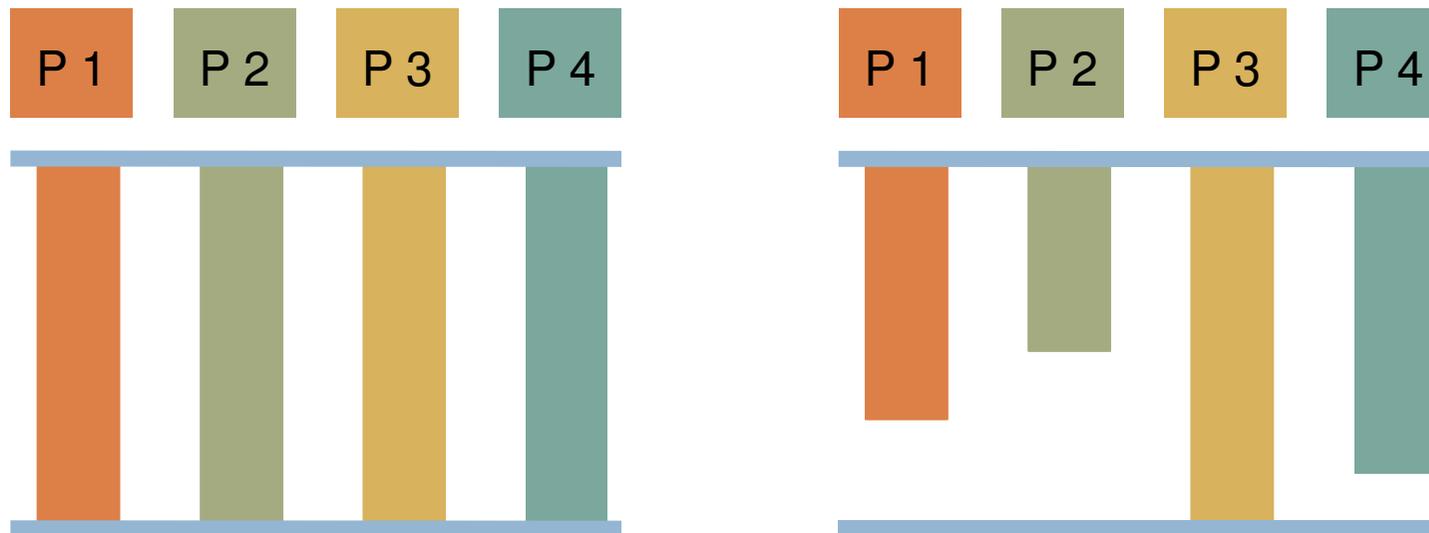
- Grobgranulare Parallelität

- Hohes Verhältnis von Berechnung zu Kommunikation
- Lange Berechnungsphasen zwischen Kommunikationsphasen
- Mehr Möglichkeiten zur Leistungssteigerung
- Schwieriger zu balancieren



# Lastausgleich

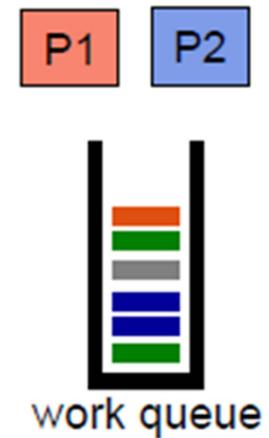
- Prozessoren, die ihre Aufgabe früher beenden, müssen auf den Prozessor mit der längsten Berechnung warten
  - Folge: Leerlaufzeiten, schlechtere Ausnutzung



***Langsamster Core (P3) gibt  
gesamte Ausführungszeit vor***

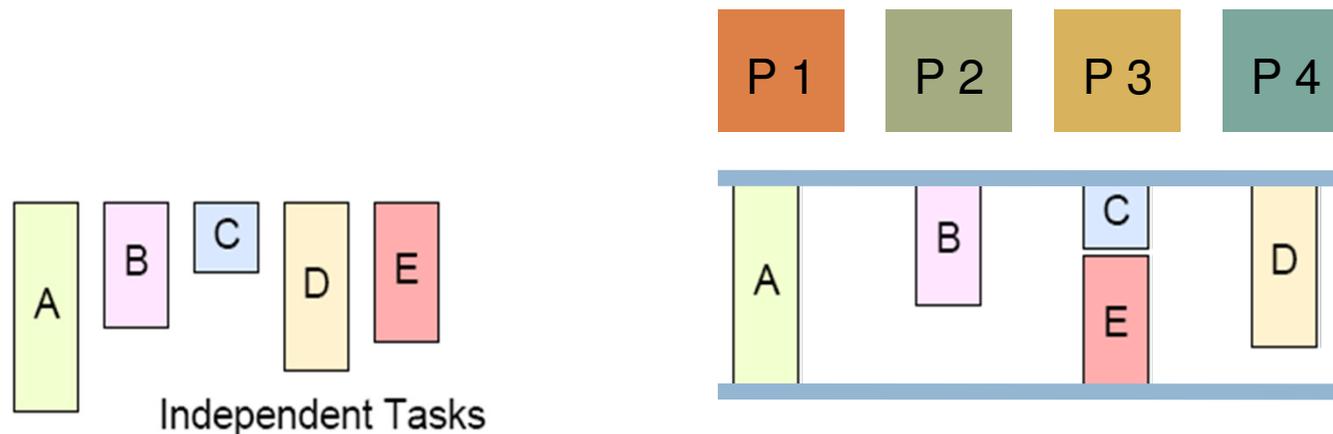
# Statischer Lastausgleich

- Entwickler trifft Entscheidungen zur Entwurfszeit: Zuteilung einer festen Menge an Arbeit zu jedem Prozessorkern
- Gut geeignet für homogene Multicores
  - Alle Cores identisch
  - Wenig Aufwand für Zuteilung
- Gut geeignet für eingebettete Systeme
  - Vorhersagbar und statisch
  - Aber: bei heterogenen Systemen aufwendig



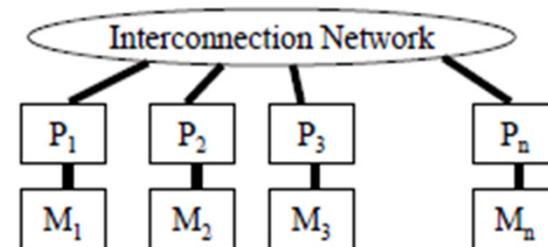
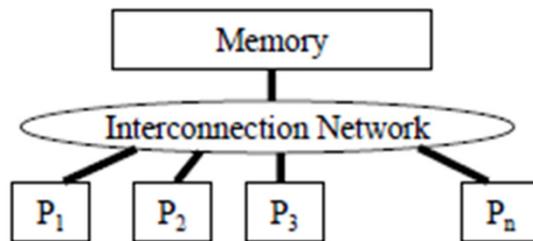
# Dynamischer Lastausgleich

- Sobald ein Core die ihm zugewiesene Aufgabe erledigt hat, übernimmt er Aufgaben des am meisten belasteten Cores
- Ideal für Cores mit ungleicher Arbeitsverteilung und heterogene Multicores
- Problematisch für eingebettete Systeme wegen schlechter Vorhersagbarkeit



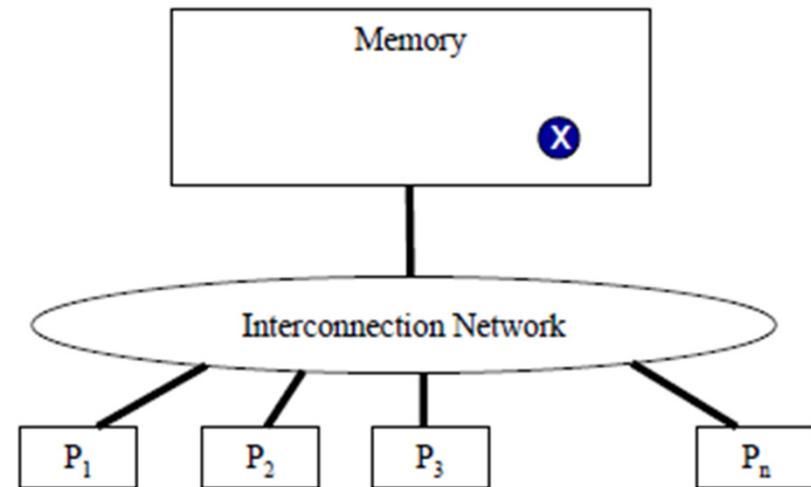
# Entwurfsparadigmen von Multicores

- Zwei grundlegende Muster für den Entwurf von Multicore-Architekturen
  - **Shared memory**
    - z.B. Intel Core i3/i5/i7
    - Eine Instanz von Daten über viele Cores gemeinsam verwendet
    - Atomicität, Locking and Synchronisation essential für die Korrektheit
    - Skalierbarkeitsprobleme
  - **Distributed memory**
    - z.B.: Cell
    - Cores arbeiten hauptsächlich auf lokalem Speicher
    - Expliziter Datenaustausch zwischen Cores
    - Planung der Datenverteilung und Kommunikation essentiell für Performance



# Programmieren mit Shared Memory

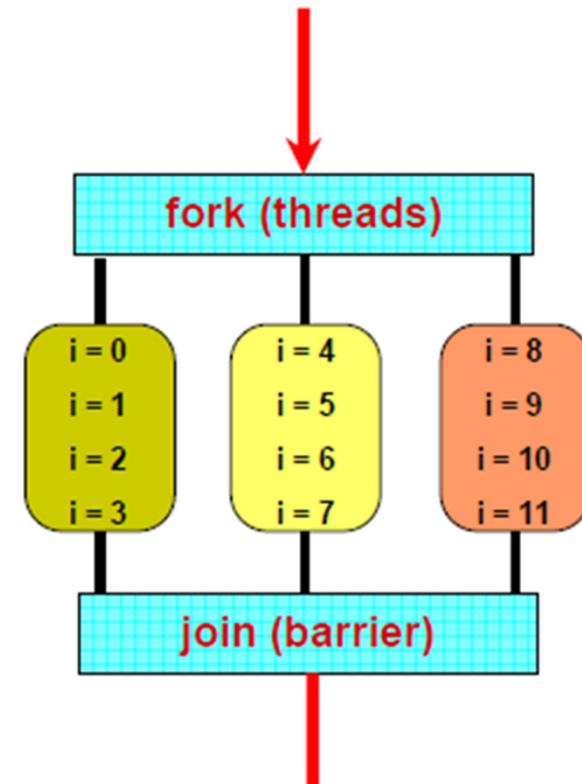
- Processor 1..n fordert X an
- Es gibt nur einen Ort für X
- Kommunikation über gemeinsame Variable
- *race conditions* möglich
  - Synchronisation kann Konflikte vermeiden
  - Synchronisation kann minimiert werden, wenn Daten anders gespeichert werden



# Beispiel

```
for (i = 0; i < 12; i++)  
    C[i] = A[i] + B[i];
```

- Datenparallel
  - Führt die selbe Berechnung auf unterschiedlichen Daten aus
- Ein Prozess kann mehrere nebenläufige Threads erzeugen
  - Jeder Thread besitzt eigenen Ausführungspfad
  - Jeder Thread besitzt lokalen Zustand und gemeinsame Ressourcen
  - Threads kommunizieren über gemeinsame Ressourcen wie z.B. globalen Speicher

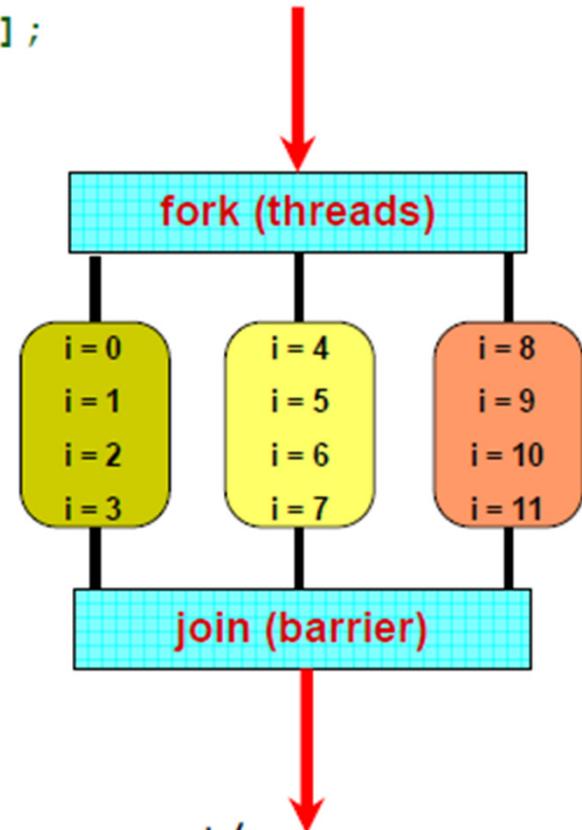


# Pthreads

```
int A[12] = {...}; int B[12] = {...}; int C[12];

void add_arrays(int start)
{
    int i;
    for (i = start; i < start + 4; i++)
        C[i] = A[i] + B[i];
}

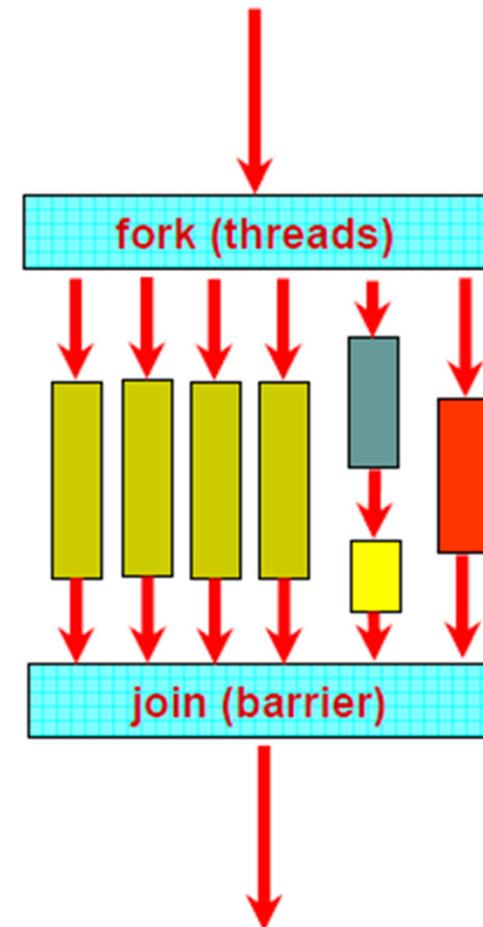
int main (int argc, char *argv[])
{
    pthread_t threads_ids[3];
    int rc, t;
    for(t = 0; t < 4; t++) {
        rc = pthread_create(&thread_ids[t],
                           NULL /* attributes */,
                           add_arrays /* function */,
                           t * 4 /* args to function */);
    }
    pthread_exit(NULL);
}
```



# Arten von Parallelität

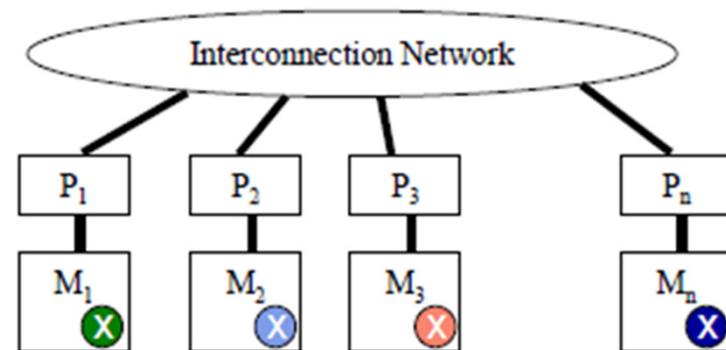
- Datenparallelität
  - Die selbe Berechnung auf unterschiedlichen Daten ausführen
- Task-(Steuerungs-)parallelität
  - Ausführung unterschiedlicher Funktionen

```
pthread_create(/* thread id  
              /* attributes  
              /* any function  
              /* args to function
```



# Programmieren mit verteiltem Speicher

- Prozessor 1..n fordert X an
- Es gibt n mögliche Orte
  - Jeder Prozessor hat sein “eigenes” X im Speicher
  - Werte von X können sich unterscheiden



- Will Prozessor 1 Wert von X von Pr. 2's X erfahren
  - Prozessor 1 muss X von Prozessor 2 anfordern
  - Prozessor 2 sendet Kopie seines X an Prozessor 1
  - Prozessor 1 empfängt Kopie
  - Prozessor 1 speichert Wert in seinem eigenen Speicher

---

# Nachrichtenaustausch

---

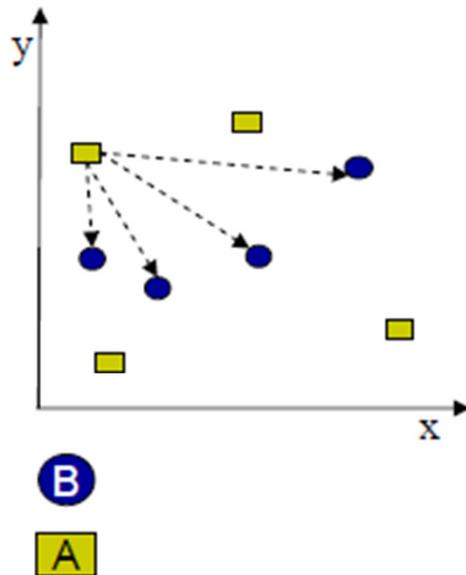
- Architekturen mit verteiltem Speicher nutzen explizite Kommunikation zum Datenaustausch
  - Datenaustausch erfordert Synchronisation (Kooperation) zwischen Sender und Empfänger



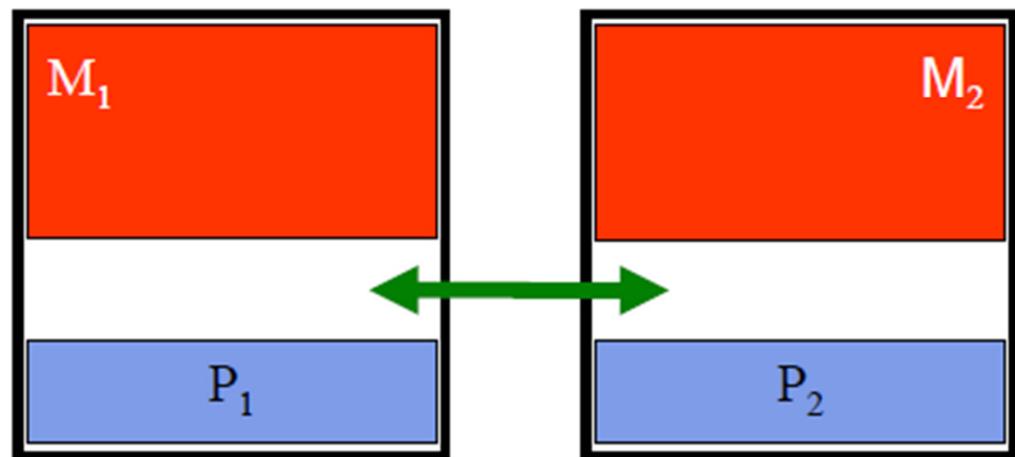
- How is "data" described
- How are processes identified
- Will receiver recognize or screen messages
- What does it mean for a send or receive to complete

# Beispiel

- Calculate the distance from each point in  $A[1..4]$  to every other point in  $B[1..4]$  and store results to  $C[1..4][1..4]$

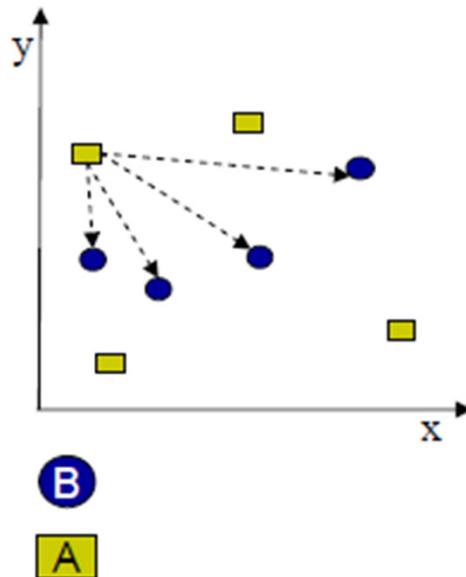


```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

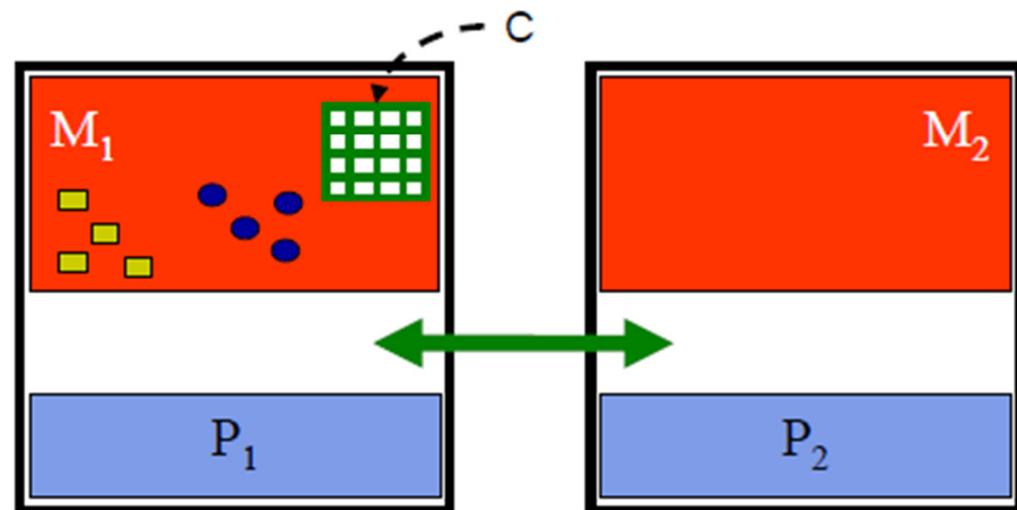


# Beispiel

- Calculate the distance from each point in  $A[1..4]$  to every other point in  $B[1..4]$  and store results to  $C[1..4][1..4]$



```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



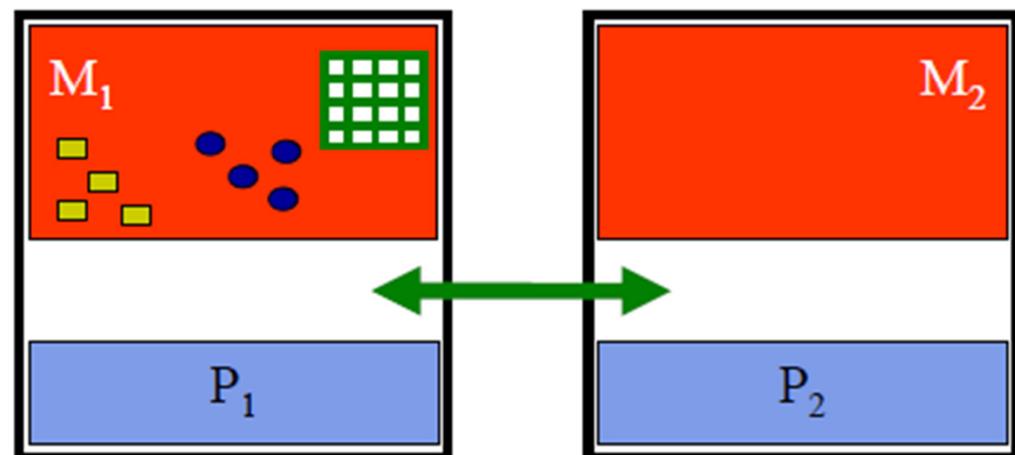
# Beispiel

- Calculate the distance from each point in  $A[1..4]$  to every other point in  $B[1..4]$  and store results to  $C[1..4][1..4]$

- Can break up work between the two processors

- $P_1$  sends data to  $P_2$

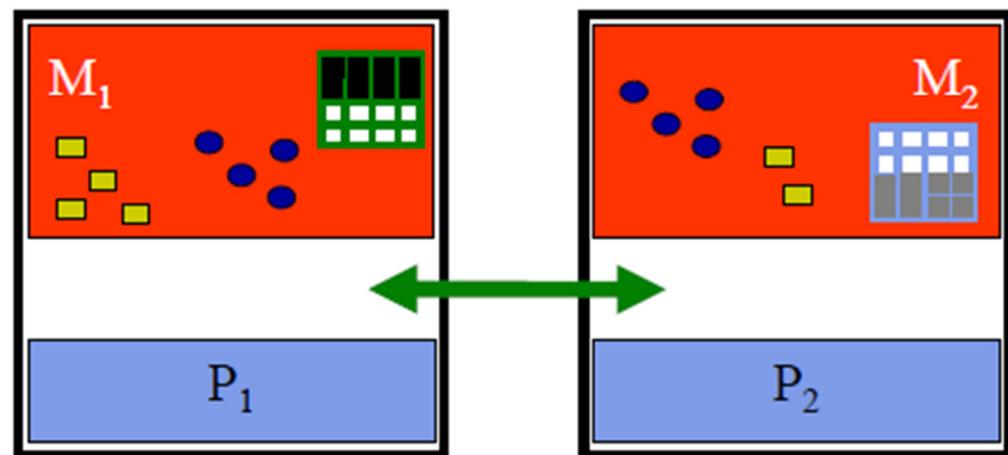
```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



# Beispiel

- Calculate the distance from each point in  $A[1..4]$  to every other point in  $B[1..4]$  and store results to  $C[1..4][1..4]$
- Can break up work between the two processors
  - $P_1$  sends data to  $P_2$
  - $P_1$  and  $P_2$  compute

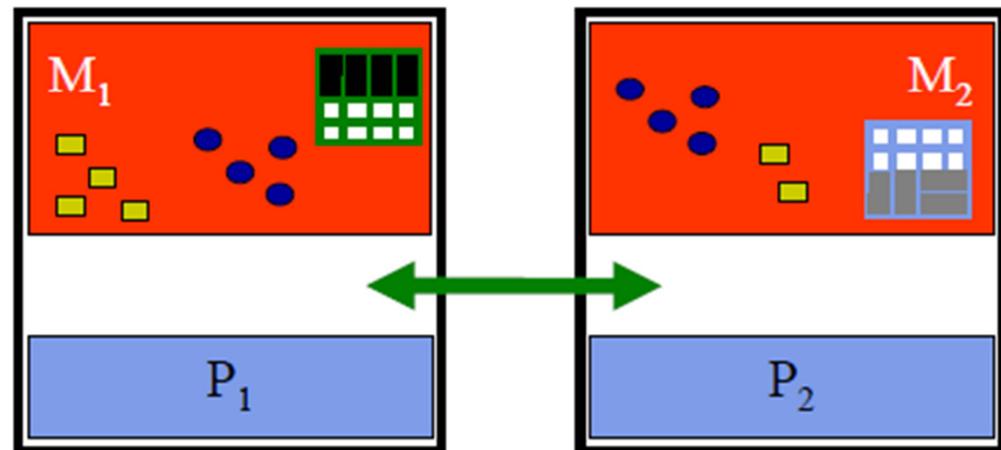
```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



# Beispiel

- Calculate the distance from each point in  $A[1..4]$  to every other point in  $B[1..4]$  and store results to  $C[1..4][1..4]$
- Can break up work between the two processors
  - $P_1$  sends data to  $P_2$
  - $P_1$  and  $P_2$  compute
  - $P_2$  sends output to  $P_1$

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



# Beispiel

processor 1

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

sequential

parallel with messages

processor 1

```
A[n] = {...}
B[n] = {...}

Send (A[n/2+1..n], B[1..n])

for (i = 1 to n/2)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Receive (C[n/2+1..n][1..n])
```

processor 2

```
A[n] = {...}
B[n] = {...}

Receive (A[n/2+1..n], B[1..n])

for (i = n/2+1 to n)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

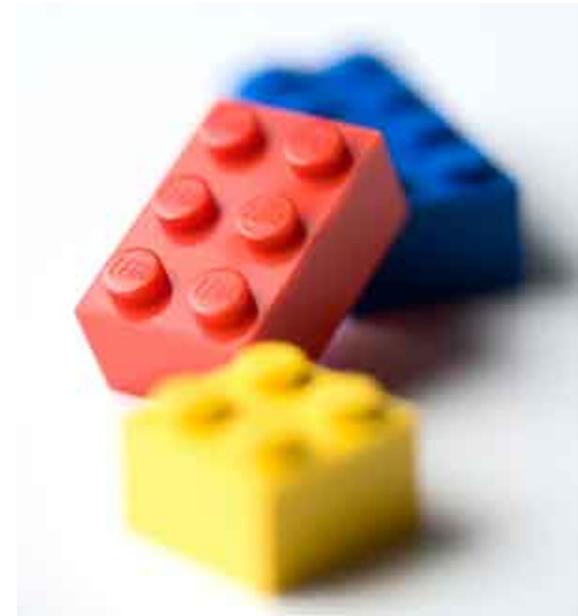
Send (C[n/2+1..n][1..n])
```

---

# Abbildung von Anwendungen auf Multicores

---

- Multicore-Architekturen
  - Entwurfsraum
  - Grenzen der Parallelität
- Beispiele für Architekturen
- Parallelität
  - Granularität und Lastverteilung
  - Explizit vs. implizit
  - Beispiele
- **Abbildung von Anwendungen**
  - Überblick
  - Zielsetzungen



---

# Abbildungsproblem

**Tools urgently needed!**

## Gegeben

- Eine Menge an Tasks
- Szenarien für die Nutzung der Tasks
- Eine Menge möglicher Architekturen mit
  - (Evtl. heterogene) Prozessoren
  - (Evtl. heterogene) Kommunikationsarchitekturen
  - Mögliche Schedulingverfahren

**Not many contributions yet!**

## Finde

- Eine Abbildung von Anwendungen auf Prozessoren
- Geeignete Schedulingverfahren (wenn nicht vorgegeben)
- Eine Zielarchitektur (wenn DSE gefordert ist)

## Ziele

- Einhaltung von Deadlines und/oder maxim. Performance
- Minimierung von Kosten und Energieverbrauch

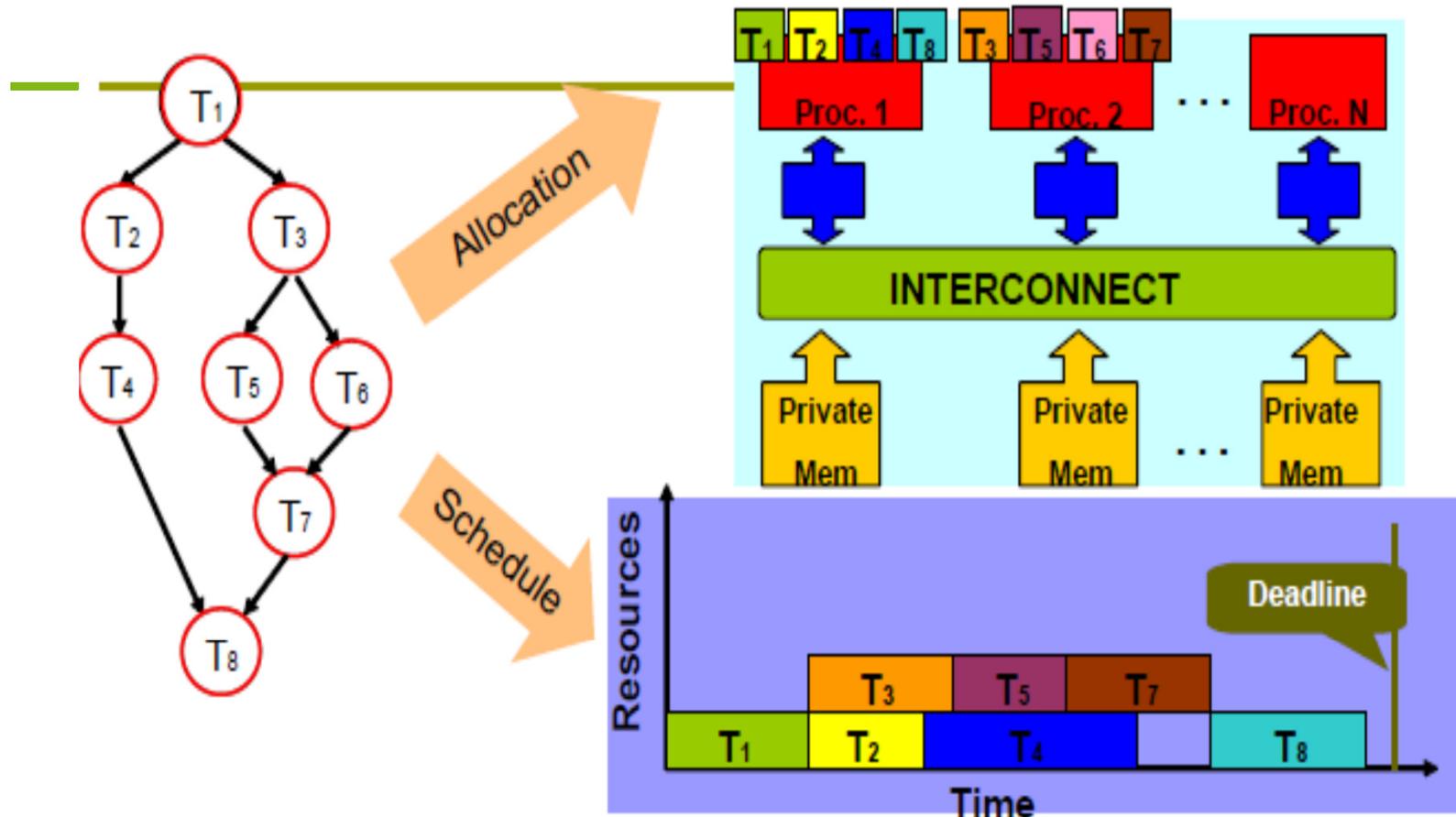
# Unterschiede bei parallelen Anwendungen für eingebettete und PC-ähnliche Systeme

	Eingebettet	PC-ähnlich
<b>Architekturen</b>	Oft heterogen, sehr kompakt	Meist homogen, nicht kompakt (x86 usw.)
<b>x86-Kompatibilität</b>	Nicht relevant	Sehr wichtig
<b>Archit. vorgegeben?</b>	Manchmal nicht	Ja
<b>Berechnungsmodelle</b>	C+versch. MoCs (SDF, ...)	Meist von Neumann
<b>Anwendungen</b>	Mehrere nebenläufige Anw.	Meist einzelne Anwendung
<b>Anw. zur Entwurfszeit bekannt?</b>	Die meisten oder alle	Nur einige (WORD, ..)
<b>Ziele</b>	Mehrere (Energie, Größe, ...)	Üblicherweise durchschnittliche Leistung
<b>Echtzeitkritisch</b>	Ja (!)	Seltenst

# Eine einfache Klassifikation

Architektur vorgegeben/ <b>Automatische Parallelisierung</b>	Vorgegebene Architektur	Architektur soll entworfen werden
<b>Von vorgegebenem Modell ausgehend</b>	Abbildung auf CELL, HOPES, ETHAM	COOL codesign tool; <b>EXPO/SPEA2</b>
<b>Automatische Parallelisierung</b>	Franke, O'Boyle et al., Mneme MAPS	Daedalus

# Beispiel für feste Architektur: Abbildung auf Cell



- The problem of allocating and scheduling task graphs on processors in a distributed real-time system is **NP-hard**.

- Martino Ruggiero, Luca Benini: Mapping task graphs to the CELL BE processor, *1st Workshop on Mapping of Applications to MPSoCs, Rheinfels Castle, 2008*

---

# Verwandte Gebiete

---

- Scheduling-Theorie:  
Grundlagen für Abbildung *Tasks* → *Startzeitpunkte*
- Hardware-/Software-Partitionierung:  
Anwendbar, wenn mehrere Prozessoren unterstützt werden
- Hochleistungsrechnen (HPC)  
Automatisch Parallelisierung, aber nur für
  - Einzelne Anwendungen,
  - Vorgegebene Architekturen,
  - Keine Unterstützung von Scheduling,
  - Meist abweichende Speicher- und Kommunikationsmodelle
- High-level-Synthese  
Nützliche Konzepte wie Scheduling, Allokation, Zuweisung
- Optimierungstheorie

---

# Zusammenfassung

---

- Klarer Trend hin zu Multiprozessorsystemen für eingebettete Systeme
- Strukturen und Kommunikation bei Multicores
- Abbildung von Anwendungen