3. Design paralleler Programme

1. Wichtige Begriffe und Definitionen
2. Dekompositionstechniken
3. Lastverteilungsverfahren
4. Parallele Algorithmenmodelle

3. Design paralleler Programme

1. **Wichtige Begriffe und Definitionen**
2. Dekompositionstechniken
3. Lastverteilungsverfahren
4. Parallele Algorithmenmodelle

Überblick: Design paralleler Programme

- Ein sequentielles Programm besteht aus einer Folge elementarer Schritte zur Lösung eines Problems.
- Ein paralleles Programm muss zusätzlich folgende Aspekte festlegen:
  - **Dekomposition:** Zerlegung in parallele Teile (**Tasks**)
  - **Mapping:** Zuordnung der Tasks zu Prozessen
  - **Kommunikation:** Austausch von Daten zwischen den Tasks
  - **Synchronisation:** Koordination der Berechnung
- Oft gibt es für jeden Aspekt mehrere Realisierungsmöglichkeiten, die auf unterschiedlichen Parallelrechnerarchitekturen zu unterschiedlichen Laufzeiten führen können.
- **Ziel:** Methodische Vorgehensweise

Eigenschaften von Tasks

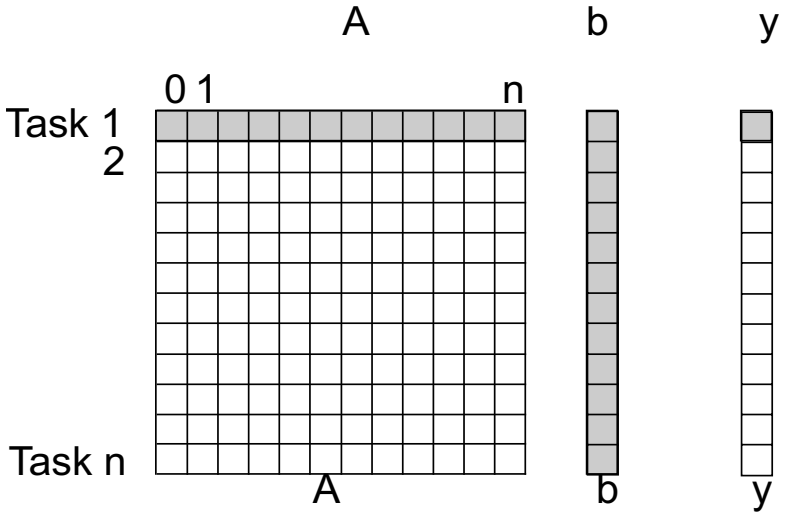
- **Tasks** sind Berechnungseinheiten, die aus dem Dekompositionsprozess eines Problems hervorgehen.
- Durch parallele Ausführung der Tasks wird eine Beschleunigung der Berechnung erzielt.
- Zwischen Tasks können **Datenabhängigkeiten** bestehen.
  - Ein Task benötigt zu seiner Abarbeitung Daten, die von einem anderen Task berechnet werden.
- **Statische Task Erzeugung:**
  - Tasks werden vor Beginn der Berechnung festgelegt.
- **Dynamische Task Erzeugung:**
  - Tasks werden (fortlaufend) während der Berechnung erzeugt.
  - Algorithmus legt fest, wann ein neuer Task mit welchen Eigenschaften erzeugt werden soll.

Eigenschaften von Tasks

- Die **Größe** eines Tasks ist definiert durch seine Berechnungsdauer.
- Oftmals besteht eine Berechnung aus Tasks von sehr unterschiedlicher Größe:
  - Dekompositionsverfahren erzeugt Tasks mit unterschiedlicher Größe.
  - Größe der Tasks ist a priori nicht bekannt.

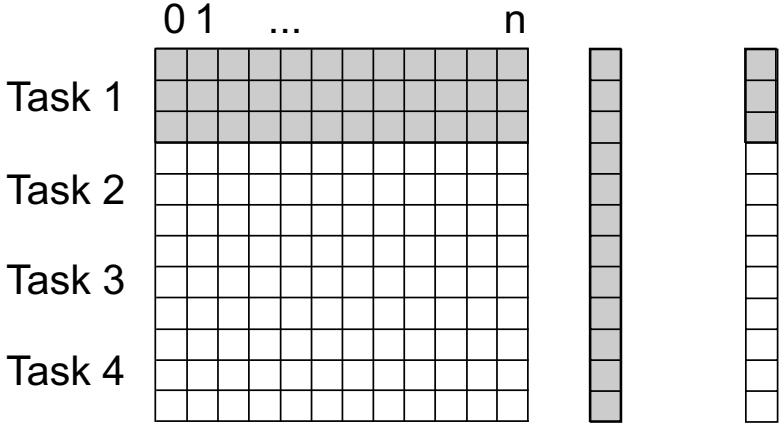
- Dekomposition ist durch ihre **Granularität** gekennzeichnet:
  - **fein-granular** (fine-grained): viele kleine Tasks
  - **grob-granular** (coarse-grained): wenige große Tasks

Beispiel: Matrix-Vektor Multiplikation



$$y[i] = \sum_{j=1}^{n} A[i,j].b[j]$$

Fein–granulare Dekomposition



Grob-granulare Dekomposition

Prozess vs. Prozessor

- Prozesse sind „Berechnungsagenten“ die Tasks ausführen.
  - Hier nicht umfassende Definition aus Betriebssystemkontext.
- Wesentliche Eigenschaften:
  - Prozess kann aus Programmcode und Daten eines Tasks in endlicher Zeit das Ergebnis des Tasks berechnen.
  - Die Prozesse einer Berechnung können untereinander kommunizieren und sich synchronisieren.
- Meistens wird eine 1-zu-1 Beziehung zwischen Prozessen und Prozessoren angenommen.
- Manchmal ist aber ein höherer Abstraktionsgrad nützlich:
  - z.B. bei mehrstufigem Designprozess für hierarchische Parallelrechnerarchitekturen

Task-Abhängigkeitsgraph

- Datenabhängigkeiten zwischen den Tasks werden durch den **Task-Abhängigkeitsgraph** angezeigt:
  - Gerichteter, azyklischer Graph
  - Knoten repräsentieren Tasks
  - Gewicht eines Knotens ist die Größe des Tasks
  - Kanten geben Datenabhängigkeiten an
  - Start-Knoten: Knoten ohne eingehende Kanten
  - End-Knoten: Knoten ohne ausgehende Kanten
- Der Task-Abhängigkeitsgraph bestimmt die Ausführungs-reihenfolge der Tasks: Ein Task kann dann ausgeführt werden, wenn alle Tasks ausgeführt wurden, die über eingehende Kanten mit ihm verbunden sind.

Eigenschaften von Task- Abhängigkeitsgraphen

- **Maximaler Grad der Nebenläufigkeit:**
  Maximale Anzahl an Tasks, die zu einem Zeitpunkt gleichzeitig ausgeführt werden können.
- **Kritischer Pfad:** Längster vorkommender gerichteter Pfad zwischen Start- und End-Knoten.
  - **Pfadlänge:** Summe der Gewichte der Knoten entlang des Pfades.
- **Durchschnittlicher Grad der Nebenläufigkeit:** Verhältnis des Gesamtgewichts der Tasks zur Länge des kritischen Pfads.

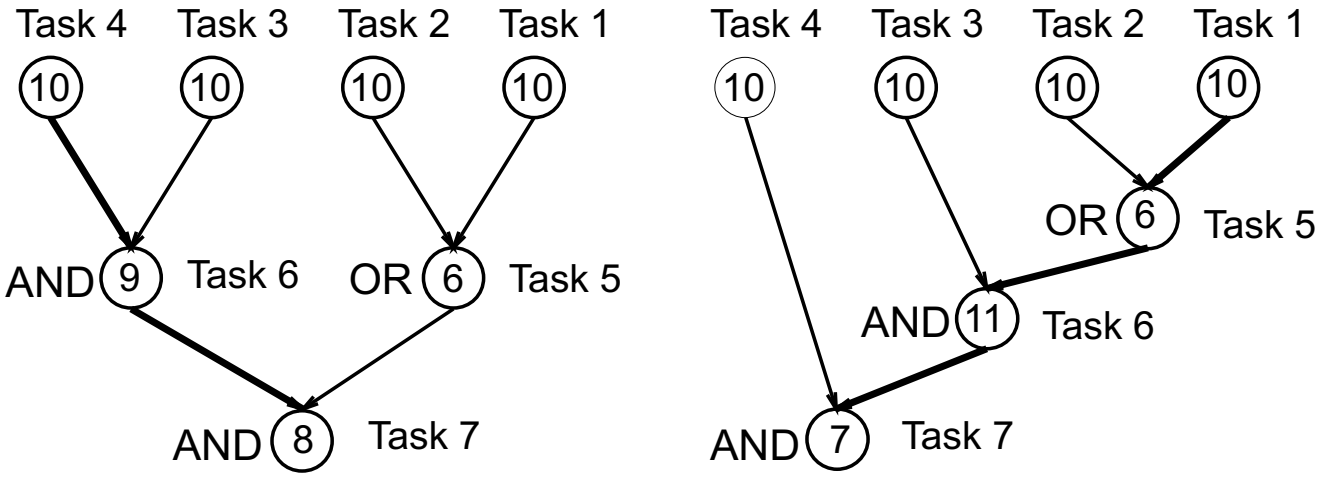
Beispiel: Datenbankanfrage

KFZ Datenbank

ID	Model	Year	Color	Price
4523	Civic	2002	Blue	$18,000
3476	Corolla	1999	White	$15,000
7623	Camry	2001	Green	$21,000
9834	Prius	2002	Green	$18,000
6734	Civic	2001	White	$17,000
5342	Altima	2001	Green	$19,000
3845	Maxima	2001	Blue	$22,000
8354	Accord	2000	Green	$18,000
4395	Civic	2001	Red	$17,000
7352	Civic	2002	Red	$18,000

Beispiel: Datenbankanfrage

MODEL=„CIVIC“ AND YEAR=„2001“ AND (COLOR=„WHITE“ OR COLOR=„GREEN“)



Task-Größe: Anzahl der Zugriffe auf einzelne Elemente

Dekomposition A

- Länge des kritischen Pfads: 27
- Durchschnittlicher Grad der Nebenläufigkeit: 63/27 = 2,33…

Dekomposition B

- Länge des kritischen Pfads: 34
- Durchschnittlicher Grad der Nebenläufigkeit: 64/34 = 1,88…

Interaktion zwischen Tasks

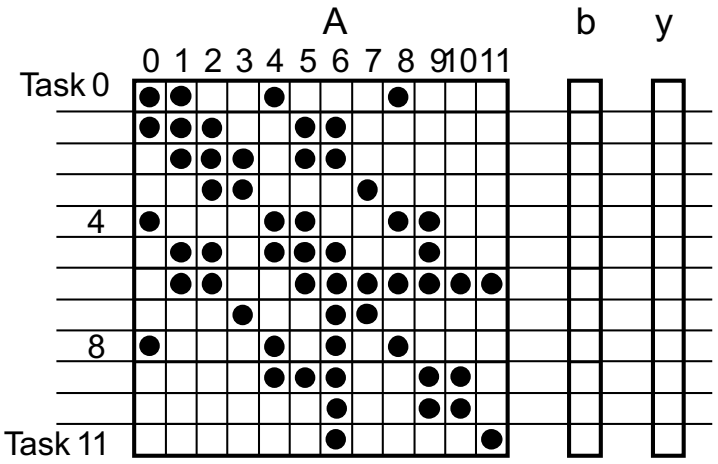
- Die maximal erzielbare Beschleunigung wird bestimmt von
  - dem durchschnittlichen Grad der Nebenläufigkeit
    - der Granularität der Dekomposition
    - der Länge des kritischen Pfades
  - **und der Interaktion der Tasks**
- Oftmals sind Interaktionen zwischen Tasks nicht im Task-Abhängigkeitsgraph berücksichtigt.
  - Interaktionen sind abhängig vom Programmiermodell und/oder der Architektur des Parallelrechners.
  - Beispiel: der Eingabevektor b bei einer Matrix-Vektor Multiplikation muss allen Prozessen zur Verfügung stehen.

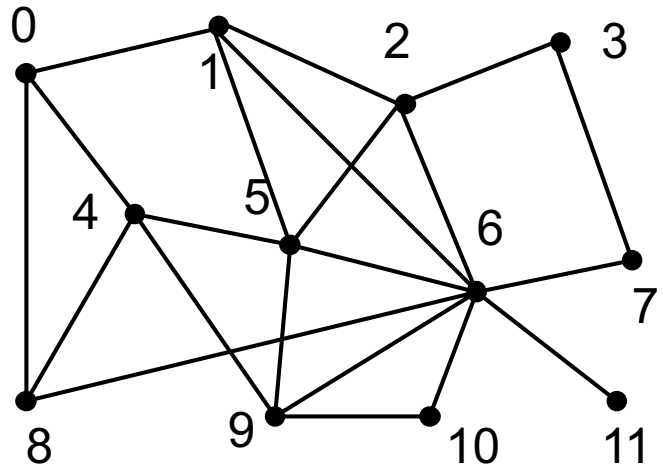
Task-Interaktionsgraph

- Der **Task-Interaktionsgraph** stellt das Interaktionsmuster zwischen den Tasks dar.
  - Knoten repräsentieren Tasks
  - Kanten zeigen Interaktionen zwischen den Tasks an
- Die Kantenmenge des Task-Interaktionsgraphs ist eine Obermenge der Kantenmenge des Task- Abhängigkeitsgraphs.
- **Task-Abhängigkeitsgraph erfasst Problem spezifische Aspekte.**
- **Task-Interaktionsgraph erfasst (zusätzlich) Aspekte der Abbildung auf eine konkrete Parallelrechnerarchitektur.**

Beispiel: Sparse Matrix-Vektor Multiplikation

- Dünnbesetzte (sparse) Matrix: viele Einträge sind 0.
- Daten-Dekomposition auf Nachrichten basierter Architektur:
  Task i berechnet y[i] und **speichert** A[i,*] und b[i].
- Task-Abhängigkeitsgraph enthält keine Kanten.





Task-Interaktionsgraph

Eigenschaften von Task-Interaktionen

- **Statisches Interaktionsmuster**
  - Interagierende Tasks stehen vor Beginn der Berechnung fest.
  - Interaktionen treten zu vordefinierten Zeitpunkten auf.
- **Dynamisches Interaktionsmuster**
  - Interagierende Tasks und/oder Zeitpunkte der Interaktion können nicht vorherbestimmt werden.
- Dynamische Interaktionsmuster sind im Message-Passing Programmiermodell schwierig zu realisieren:
  - Sinnvolle Platzierung der send/receive Paare im Programmtext schwierig.
  - Zusätzliche Synchronisation oder Polling erforderlich.

Eigenschaften von Task-Interaktionen

- **Reguläres Interaktionsmuster**
  - Struktur des Interaktionsmusters kann für effiziente Implementierung genutzt werden.
  - Interagierende Tasks werden so auf Prozesse abgebildet, dass sie effizient kommunizieren können.
  - Beispiel: Sparse Matrix-Vektor Multiplikation, bei der die von 0 verschiedenen Elemente der Matrix ein Muster aufweisen. (z.B. Bandmatrix: Nicht-Null Elemente liegen auf Diagonale)
- **Irreguläres Interaktionsmuster**
  - Interaktionsmuster weist keine verwertbare Struktur auf.
  - Beispiel: Sparse Matrix-Vektor Multiplikation, bei der die 0 Elemente der Matrix zufällig verteilt sind.

Eigenschaften von Task-Interaktionen

- **Two-Way Interaktion**
  - Die von einem Task benötigten Daten werden explizit von einem (oder mehreren) anderen Task(s) zur Verfügung gestellt.
  - Typischerweise Producer/Consumer Beziehung.
- **One-Way Interaktion**
  - Ein Task initiiert die Kommunikation, ohne die Ausführung von anderen Tasks zu beeinflussen.
  - Typischerweise Read-Only Kommunikation.
- Im Message-Passing Programmiermodell müssen One-Way Interaktionen immer zu Two-Way Interaktionen umstrukturiert werden.

3. Design paralleler Programme

1. Wichtige Begriffe und Definitionen
2. **Dekompositionstechniken**
3. Lastverteilungsverfahren
4. Parallele Algorithmenmodelle

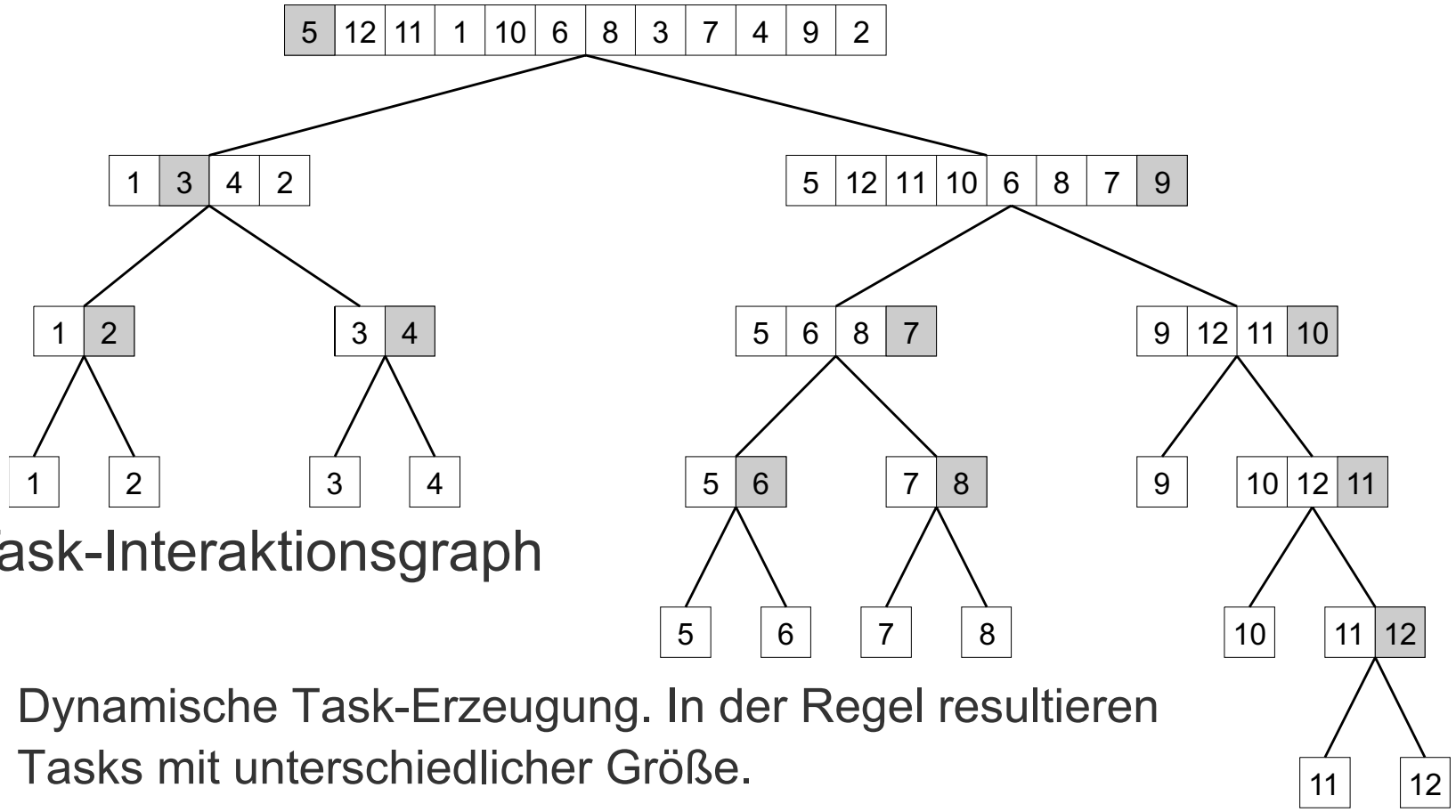
Dekompositionstechniken

- Klassen von Dekompositionstechniken:
  - Rekursive Dekomposition
  - Daten-Dekomposition
  - Explorative Dekomposition
  - Spekulative Dekomposition
- Dekompositionstechniken können als Ausgangspunkt verwendet werden; oftmals ist Abwandlung oder Kombination erforderlich.

Rekursive Dekomposition

- **Divide-and-Conquer** Schema wird benutzt, um Nebenläufigkeit zu induzieren.
  - Aufteilung in Menge von unabhängigen Subproblemen (Divide-Schritt).
  - Nach Lösung der Subprobleme werden die Ergebnisse zusammengeführt (Conquer-Schritt).
  - Jedes Subproblem wird gelöst, indem es rekursiv weiter unterteilt wird, bis Trivialfall erreicht ist.
- Oftmals können Algorithmen neu strukturiert werden, um sie für rekursive Dekomposition zugänglich zu machen.

Beispiel: Quicksort



Task-Interaktionsgraph

- Dynamische Task-Erzeugung. In der Regel resultieren Tasks mit unterschiedlicher Größe.
- Dynamisches, reguläres, two-way Interaktionsmuster

Beispiel: Kleinstes Element

Standard Algorithmus

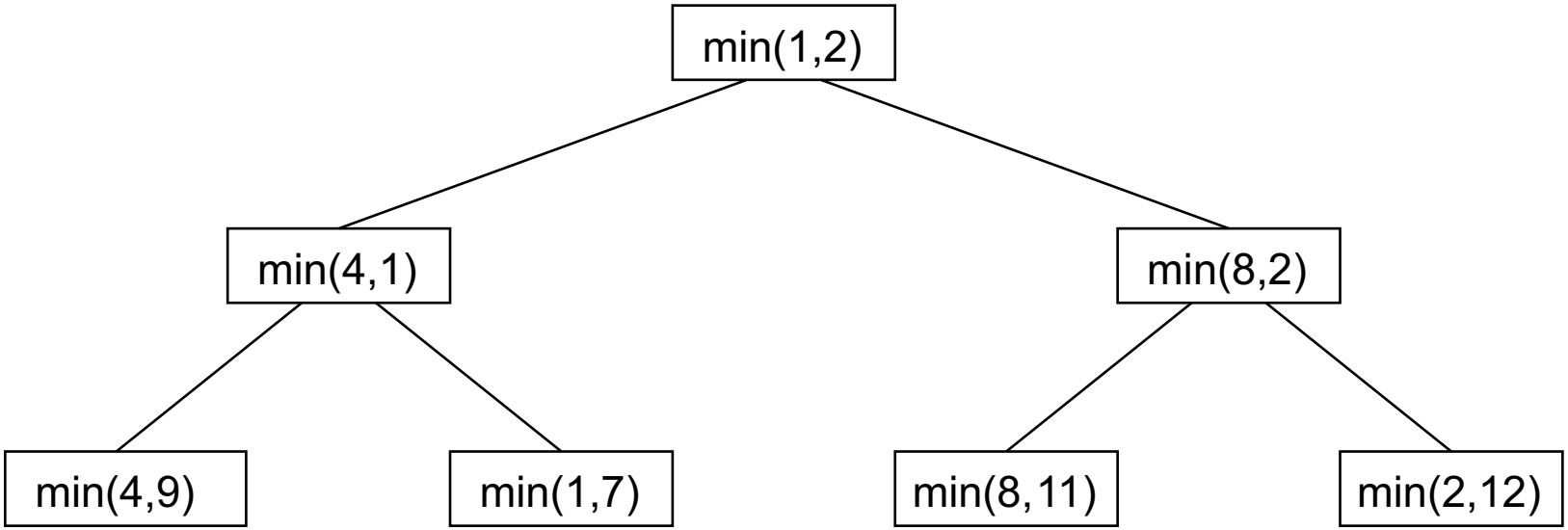
```
procedure SERIAL_MIN (A, n)
begin
  min := A[0];
  for i := 1 to n - 1 do
    if (A[i ] < min) then
      min := A[i ];
  endfor;
  return min;
end
```

Divide-and-Conquer Algorithmus

```
procedure RECURSIVE_MIN (A, n)
begin
  if (n = 1) then
    min := A[0];
  else
    lmin := RECURSIVE_MIN (A, n/2);
    rmin := RECURSIVE_MIN (&(A[n/2]), n
   - n/2);
    if (lmin < rmin) then
      min := lmin;
    else
      min := rmin;
    endelse;
  endelse;
  return min;
end
```

Beispiel: Kleinstes Element

Task-Interaktionsgraph für Eingabe: {4,9,1,7,8,11,2,12}



Daten-Dekomposition

- Daten-Dekomposition wird gewöhnlich bei Algorithmen eingesetzt, die große Datenstrukturen manipulieren.
- Dekomposition erfolgt in 2 Schritten:
  1. Datenstrukturen werden partitioniert
  2. Partition induziert Dekomposition des Problems in verschiedene Tasks
- **Owner-Computes Regel**
  Ein Task führt alle Berechnungen auf dem ihm zugewiesenen Datenbereich aus.
- Partitionierung kann auf Eingabe-, Ausgabe- und/oder Zwischen-Datenstrukturen vorgenommen werden.
  - Kombination führt oft zu fein-granularer Dekomposition

Partitionierung der Ausgabedaten

- Anwendbar, wenn die Elemente der Ausgabedatenstruktur unabhängig voneinander berechnet werden können.
- Owner-Computes Regel bedeutet hier: Jeder Task berechnet einen Teil der Ausgabe.
- Beachte: Eine Partitionierung der Ausgabedaten kann zu unterschiedlichen Dekompositionen in Tasks führen (siehe Beispiel auf Folie 27).

Beispiel: Matrix Multiplikation

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Formulierung mit Blockoperationen

$$\begin{aligned}
\text{Task 1: } C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
\text{Task 2: } C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
\text{Task 3: } C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
\text{Task 4: } C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}$$

Beispiel: Matrix Multiplikation

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Dekomposition 1	Dekomposition 2
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

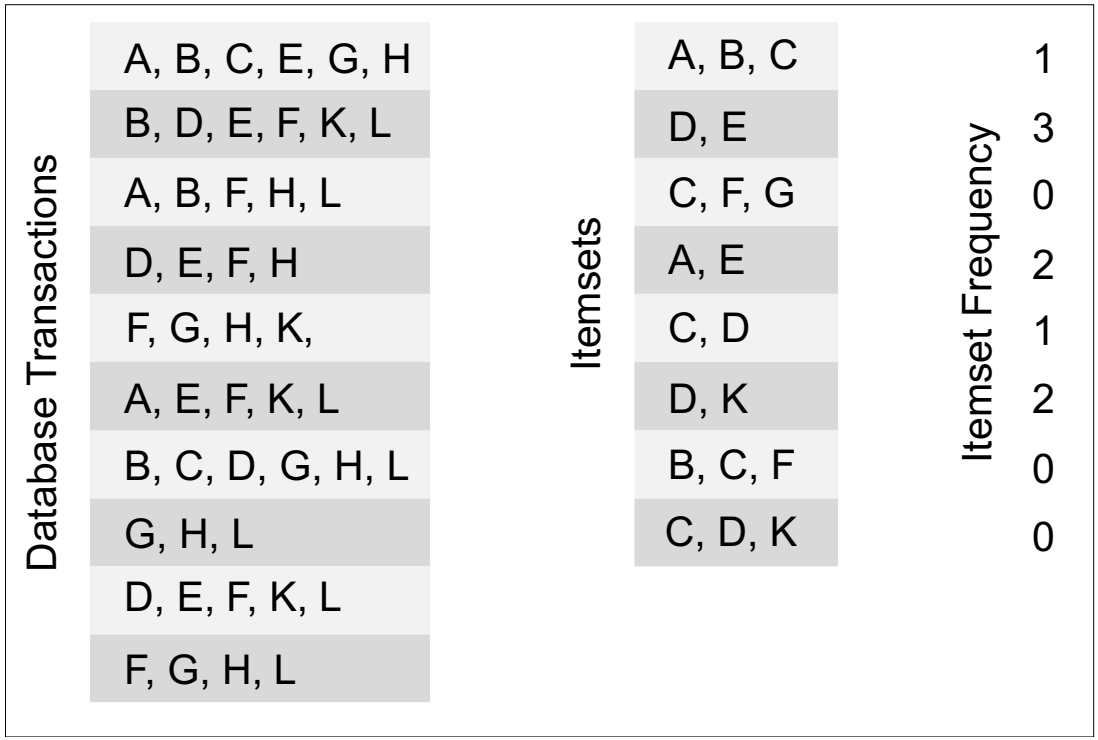
Partitionierung der Eingabedaten

- Jeder Task führt zunächst Berechnung auf (lokalem) Eingabedatenbereich aus.
- Anschlussberechnung erforderlich, um die Teilergebnisse zusammenzufügen.
- Owner-Computes Regel bedeutet hier: Jeder Task berechnet soviel wie möglich auf den zugewiesenen Eingabedaten.

Beispiel: Data-Mining

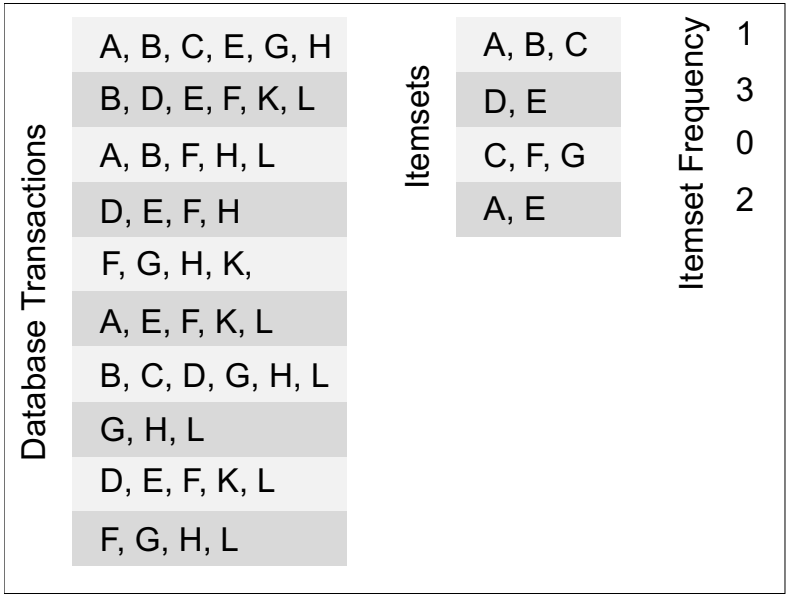
- Gegeben:
  - Menge T von n Datenbank-Transaktionen
  - Menge S mit m Itemsets
  - Jede Transaktion und jedes Itemset besteht aus Elementen einer Menge I = {A,B,C,...} von Items.
- Problem: Finde für jedes Itemset in I die Häufigkeit seines Vorkommens in den Transaktionen aus T.
- Beispiel:
  - Jede Transaktion in T stellt einen Kassenbon dar.
  - Die Itemsets in S sind verschiedene Warengruppen.

Beispiel: Data-Mining

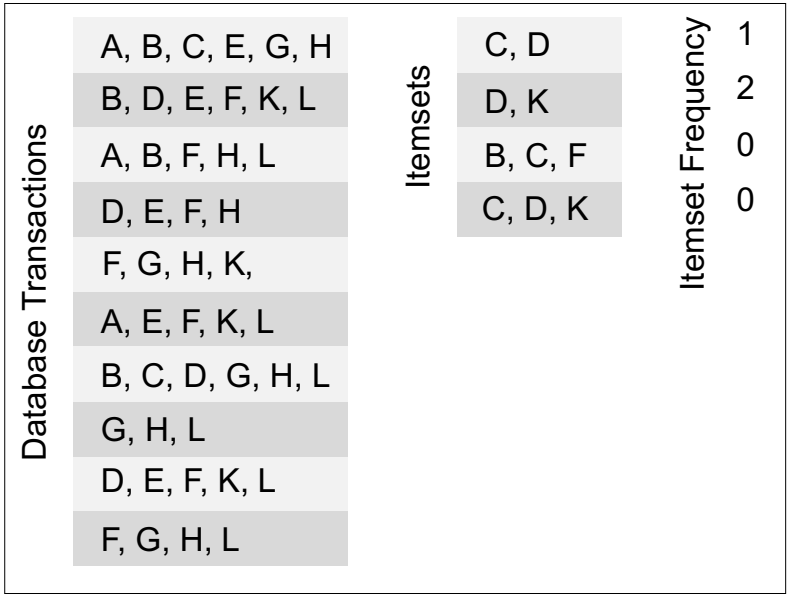


Beispiel: Data-Mining

Partitionierung der Ausgabedaten



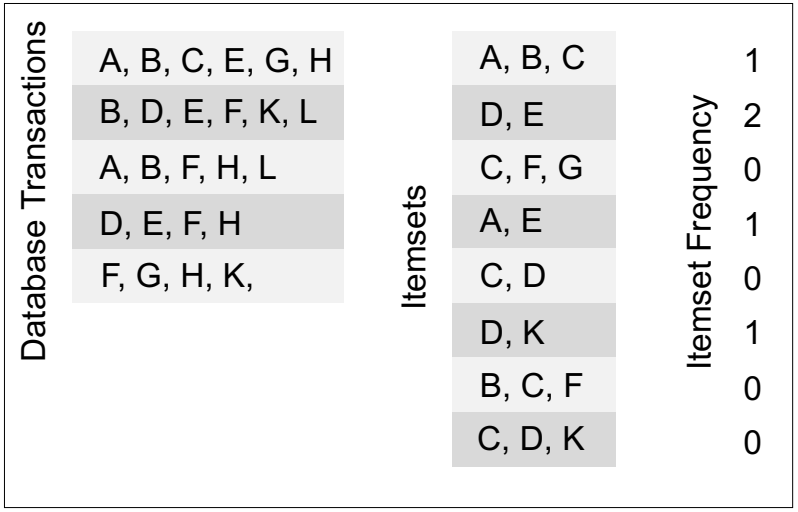
Task 1



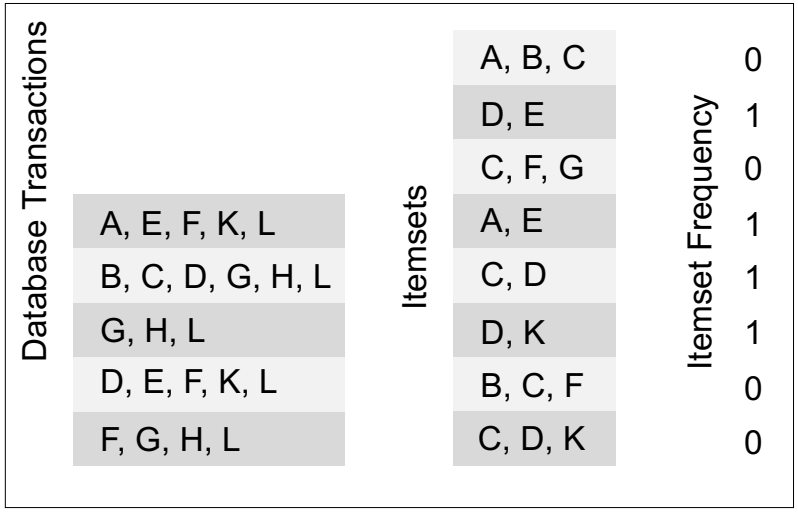
Task 2

Beispiel: Data-Mining

Partitionierung der Eingabedaten



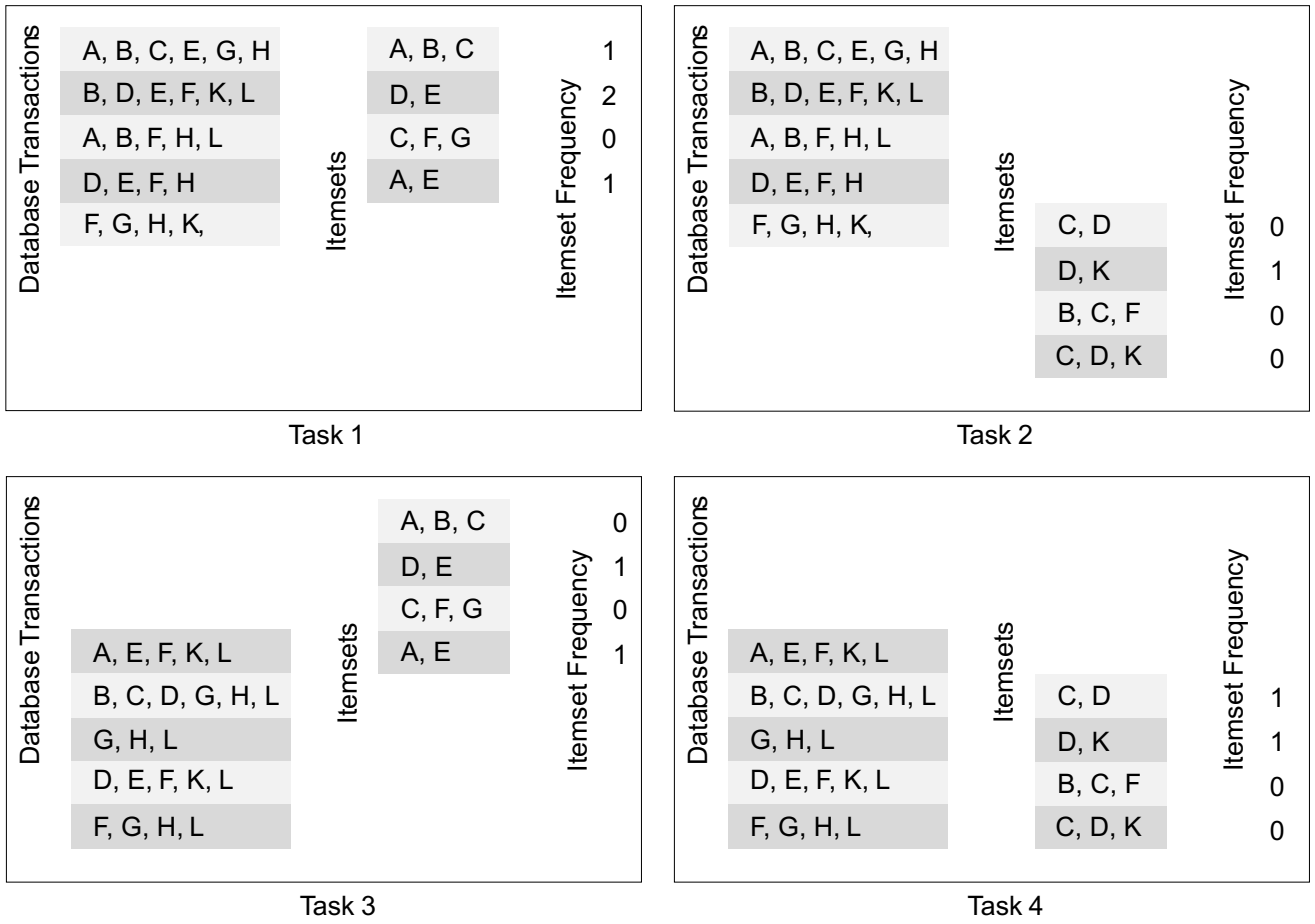
Task 1



Task 2

Beispiel: Data-Mining

Partitionierung der Ein- und Ausgabedaten



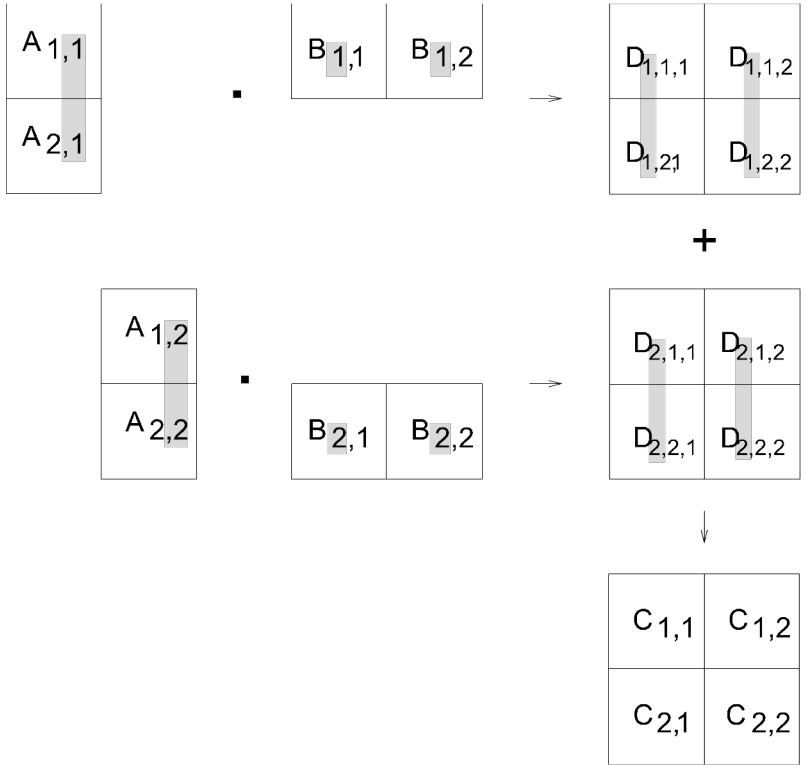
Partitionierung von Zwischendaten

- Anwendbar bei mehrstufigen Algorithmen
  - Zwischen-Datenstruktur ist Ausgabe einer Stufe bzw. Eingabe der nachfolgenden Stufe.
- Auch explizite Einführung von Zwischen-Datenstrukturen möglich, die im sequentiellen Algorithmus nicht gebraucht werden.
- **Vorteil:** Partitionierung der Zwischen-Datenstruktur induziert häufig zusätzliche Nebenläufigkeit.
- **Nachteil:** Explizite Zwischen-Datenstruktur benötigt zusätzlich Speicherplatz.

Beispiel: Matrix Multiplikation

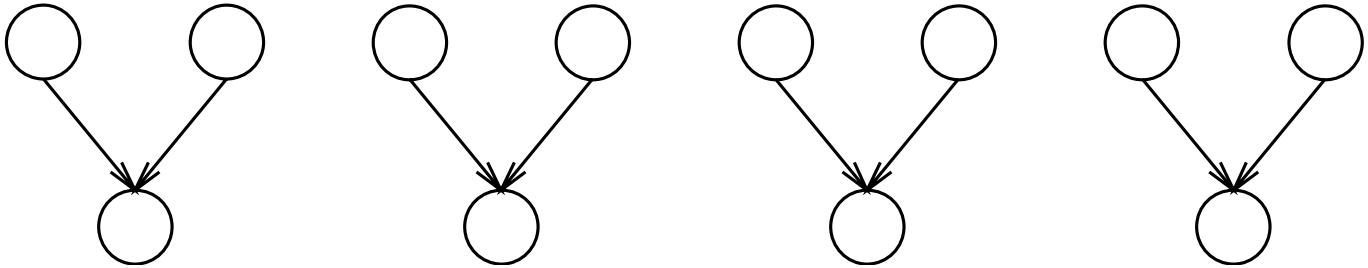
Explizite Einführung einer Zwischen-Datenstruktur D:

$D_{k,i,j}$ ist Produkt von $A_{i,k}$ und $B_{k,j}$



Beispiel: Matrix Multiplikation

Task-Interaktionsgraph:



Maximaler Grad der Nebenläufigkeit: 8

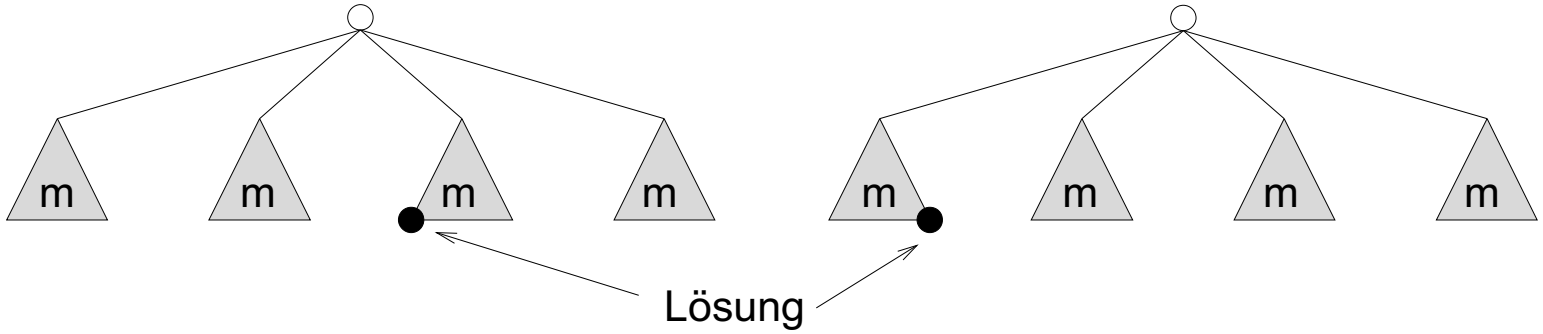
Durchschnittlicher Grad der Nebenläufigkeit: 6

Explorative Dekomposition

- Anwendbar zur Dekomposition von Suchproblemen:
  - Suchraum wird in disjunkte Teilräume aufgeteilt.
  - Parallele Tasks bearbeiten jeweils einen Teilraum.
- Unterschiede zu Daten-Dekomposition:
  - Suchraum wird meistens dynamisch aufgebaut, z.B. als Suchbaum.
  - Falls ein Task eine Lösung gefunden hat, kann die Arbeit der anderen Tasks verworfen werden.

Explorative Dekomposition

Bei paralleler Suche treten oft Anomalien auf:



Finden der Lösung:

- Seriell: 2m+1 Schritte
- Parallel: 1 Schritt

Finden der Lösung:

- Seriell: m Schritte
- Parallel: m Schritte

Gesamtaufwand:

- Seriell: m Schritte
- Parallel: 4m Schritte

Spekulative Dekomposition

- Anwendbar bei bedingten Programmverzweigungen, z.B. if-then-else oder switch-case Konstrukten.
- Ansatz: Auswertung der Bedingung und Ausführung aller möglichen Programmverzweigungen erfolgen parallel.
- Nachdem das Resultat der Auswertung der Bedingung vorliegt werden die falsch ausgeführten Tasks verworfen.
- Oft wird auch nur der wahrscheinlichste Teil der Programmverzweigung parallel zur Auswertung der Bedingung ausgeführt.

3. Design paralleler Programme

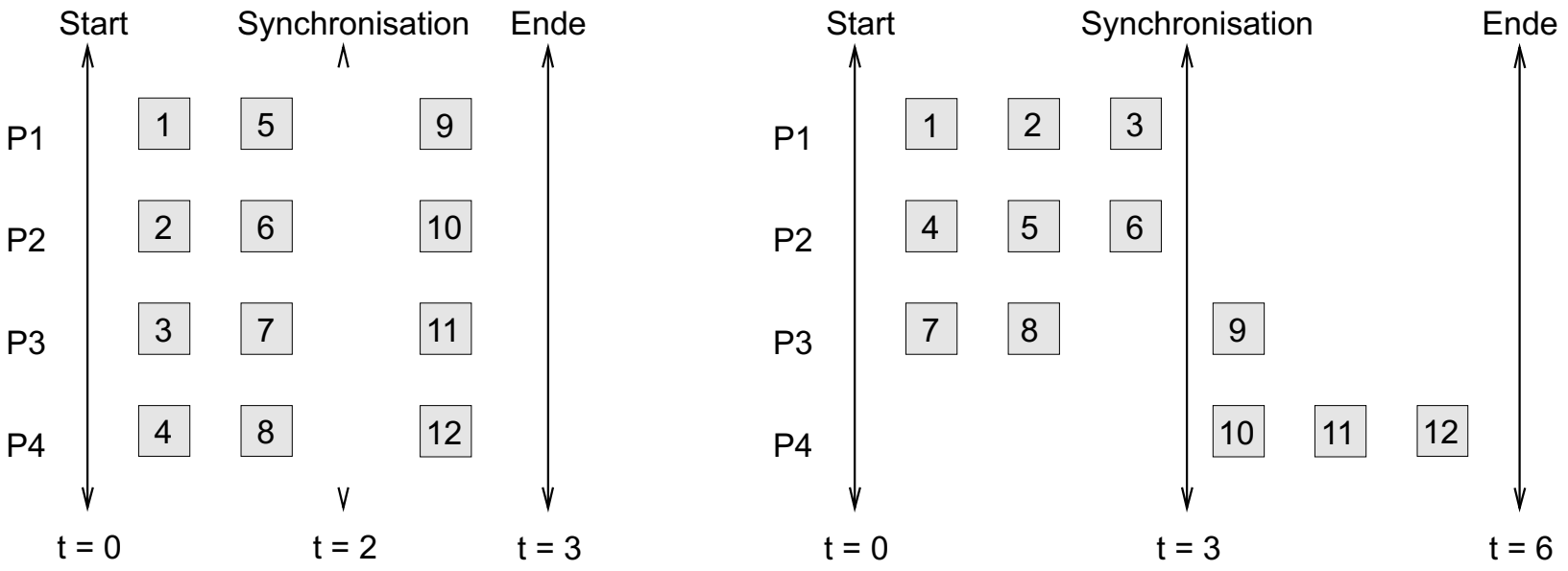
1. Wichtige Begriffe und Definitionen
2. Dekompositionstechniken
3. **Lastverteilungsverfahren**
4. Parallele Algorithmenmodelle

Lastverteilung

- Quellen von **Overhead** bei paralleler Ausführung:
  - Overhead durch Task Interaktion
    - Latenz, beschränkte Bandbreite
  - Overhead durch Leerlauf von Prozessen
    - Unterschiede in der Größe der Tasks
    - Datenabhängigkeiten blockieren die Ausführung von Tasks
- **Lastverteilung:** Zuordnung von Tasks zu Prozessen
- **Ziel der Lastverteilung:** Minimierung des Overheads der parallelen Ausführung der Tasks.
  - Minimierung der Zeit für Task Interaktion
  - Minimierung der Leerlaufzeit von Prozessen
- Oft können nicht beide Teilziele gleichzeitig erreicht werden.

Beispiel

- 12 Tasks werden gleichmäßig auf 4 Prozesse (also 3 pro Prozess) verteilt.
- Datenabhängigkeiten: Tasks 9-12 können erst nach Beendigung von Tasks 1-8 gestartet werden.



Klassifizierung von Lastverteilungsverfahren

- **Statische Lastverteilungsverfahren**
  - Die Zuordnung von Tasks zu Prozessen ist vor der Programmausführung bekannt.
  - Meistens integraler Bestandteil des Algorithmus.
  - Statische Task-Dekomposition erforderlich.
- **Dynamische Lastverteilungsverfahren**
  - Tasks werden während der Programmausführung den Prozessen zugeordnet.
  - Benötigt zusätzliche Systemkomponente zur Task-Migration.
  - Bei dynamischer Task-Dekomposition erforderlich.

Statische vs. dynamische Lastverteilungsverfahren

- **Statische Verfahren**
  - Technisch einfacher zu realisieren.
  - Erfordern Kenntnis über die Größe der Tasks und die vorkommenden Task-Interaktionen.
  - Optimale Zuordnung ist bei unterschiedlicher Task-Größe NP-vollständig, aber es gibt gute Heuristiken.

- **Dynamische Verfahren**
  - Erforderlich, wenn die Größe der Tasks stark unterschiedlich und/oder unbekannt ist.
  - Oft ineffizient, falls die Übertragungszeit der Tasks im Vergleich zu deren Berechnungszeit groß ist.

Überblick: Statische Lastverteilungsverfahren

- Statische Lastverteilungsverfahren werden meistens im Zusammenhang mit Daten-Dekompositionsverfahren oder Problemen mit statischem Task-Interaktionsgraph verwendet.
- **Lastverteilung mittels Daten-Partitionierung**
  - Blockverteilung, zyklische Blockverteilung, randomisierte Blockverteilung
- **Lastverteilung mittels Task-Partitionierung**
  - Partitionierung des Task-Interaktionsgraphs

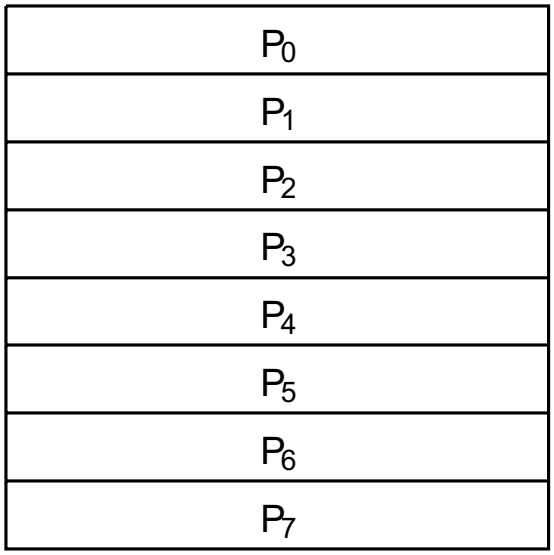
Block-Verteilungsverfahren

- Block-Verteilungsverfahren sind besonders gut geeignet, wenn die Interaktionen der Berechnung hohe Lokalität aufweisen, z.B.
  - alle Elemente lassen sich unabhängig berechnen.
  - die Berechnung eines Elements hängt nur von seinen Nachbarelementen ab.
- Wir betrachten im Folgenden beispielhaft 2-dimensionale Arrays der Größe n x n.
- Beachte: Owner-Computes Regel assoziiert Tasks und Daten
  - Abbildung von Daten zu Prozessen ist hier gleichbedeutend mit Abbildung von Tasks zu Prozessen.

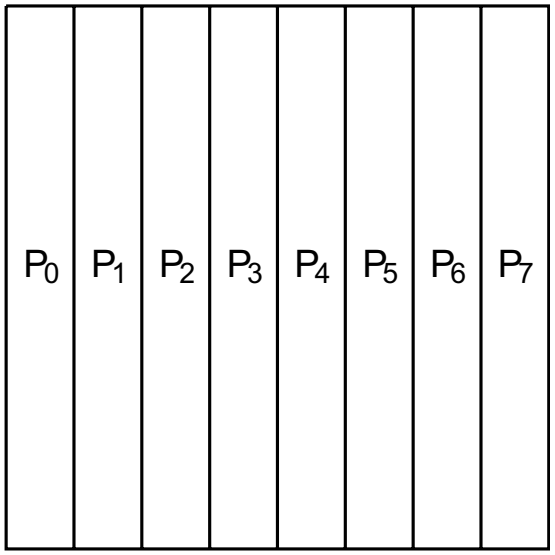
1-dimensionale Blockverteilung

- Jeder Prozess erhält zusammenhängenden Datenblock aus n/p Zeilen bzw. Spalten.
- Beispiel (p=8):

Zeilenweise Verteilung



Spaltenweise Verteilung



2-dimensionale Blockverteilung

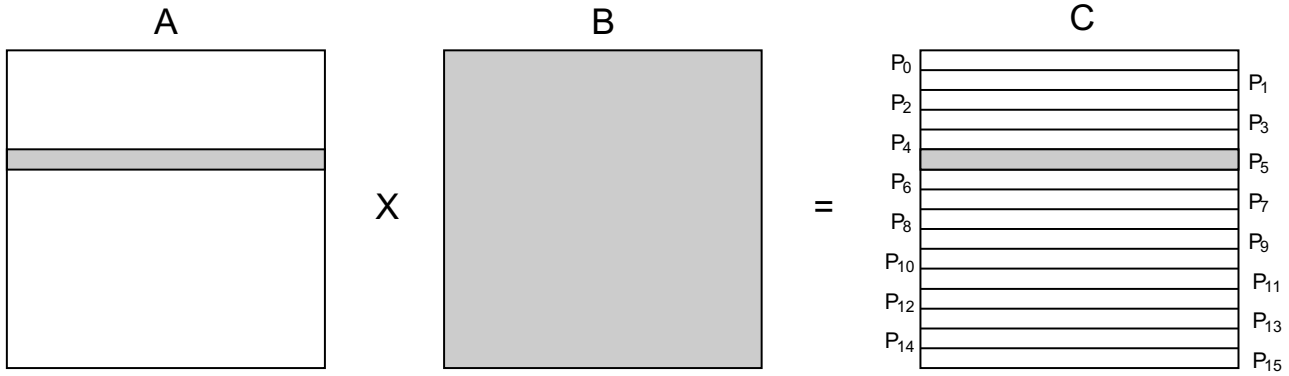
- Jeder Prozess erhält zusammenhängenden Datenblock der Größe n/p_1 x n/p_2 mit $p = p_1$ x p_2
- Beispiel ($p = 4$ x 4 und $p = 2$ x 8):

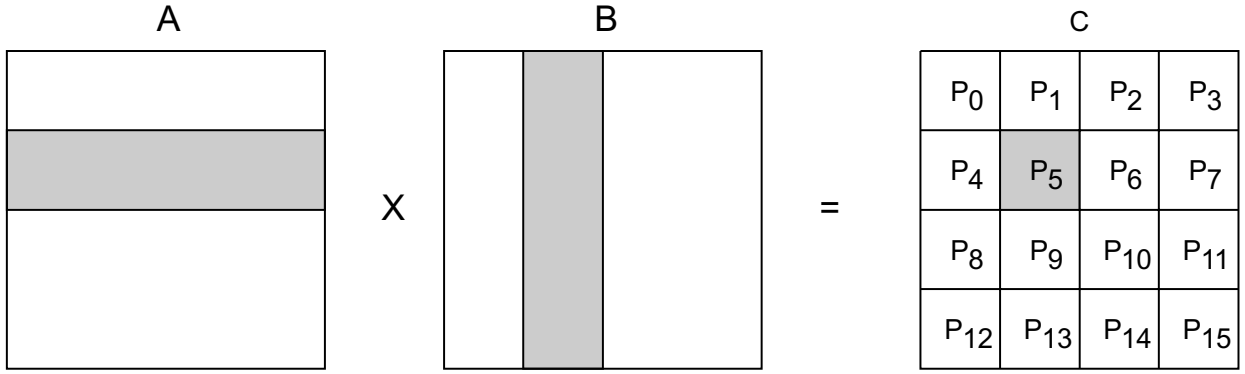
P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

Beispiel: Matrix Multiplikation

Partitionierung der Ausgabe-Matrix





Beispiel: Matrix Multiplikation

- Höher-dimensionale Partitionierung/Verteilung ermöglicht die Verwendung einer größeren Anzahl von Prozessen
  - 1 dimensional: max. n Prozesse
  - 2 dimensional: max. n^2 Prozesse
- Höher-dimensionale Partitionierung/Verteilung reduziert die Anzahl der Interaktionen
  - 1 dimensional:
    - Jeder Prozess greift auf alle Elemente der Matrix B zu
    - Gemeinsamer Datenbereich hat die Größe $O(n^2)$
  - 2 dimensional:
    - Gemeinsamer Datenbereich hat die Größe $O(n^2 / \sqrt{p})$

Zyklische Blockverteilung

- **Problem:** Falls die Berechnung der Elemente des Arrays unterschiedliche Zeit erfordert, kann durch Blockverteilung eine ungleichmäßige Lastverteilung resultieren.
- **Ansatz:** Zyklische Verteilung
  - Array wird in wesentlich mehr Blöcke partitioniert als Prozesse vorhanden sind.
  - Blöcke werden reihum auf Prozesse verteilt, so dass jeder Prozess mehrere nicht-zusammenhängende Blöcke erhält.

Zyklische Blockverteilung

1-dimensional:

αp Blöcke aus n/(αp) Zeilen/Spalten

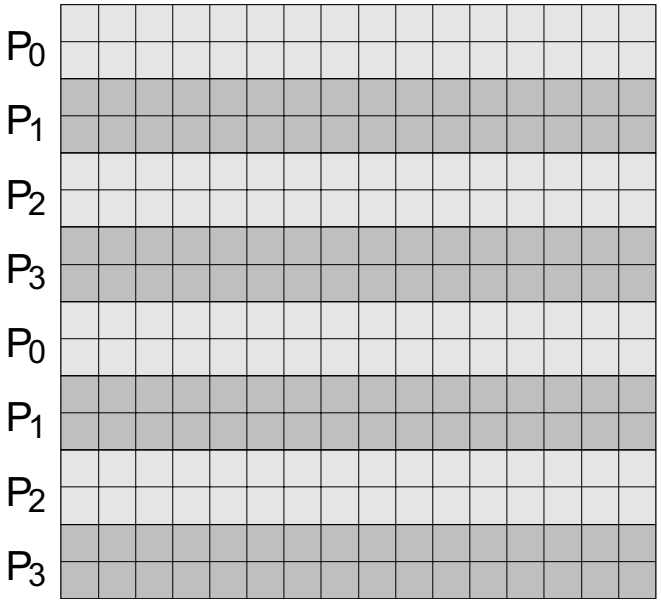
mit $1 \leq \alpha \leq n/p$

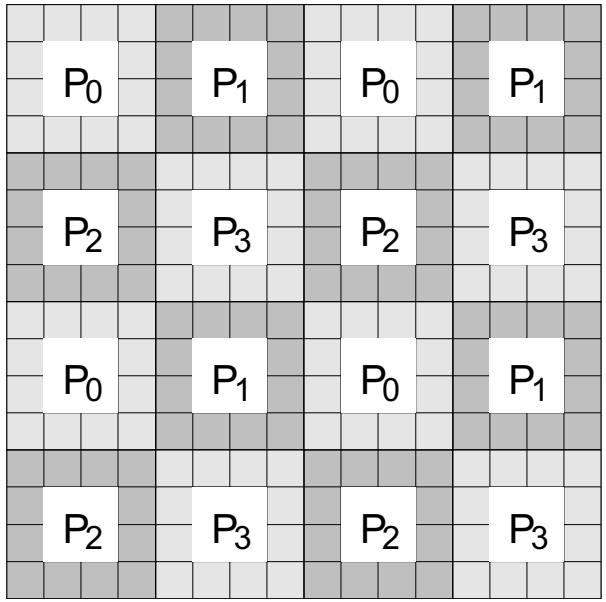
Block b_i wird Prozess $P_{i\%p}$ zugew.

2-dimensional:

$\alpha\sqrt{p}$ x $\alpha\sqrt{p}$ Blöcke der Größe $n/(\alpha\sqrt{p})$

mit $1 \leq \alpha \leq n/\sqrt{p}$





Beispiel: LU Faktorisierung

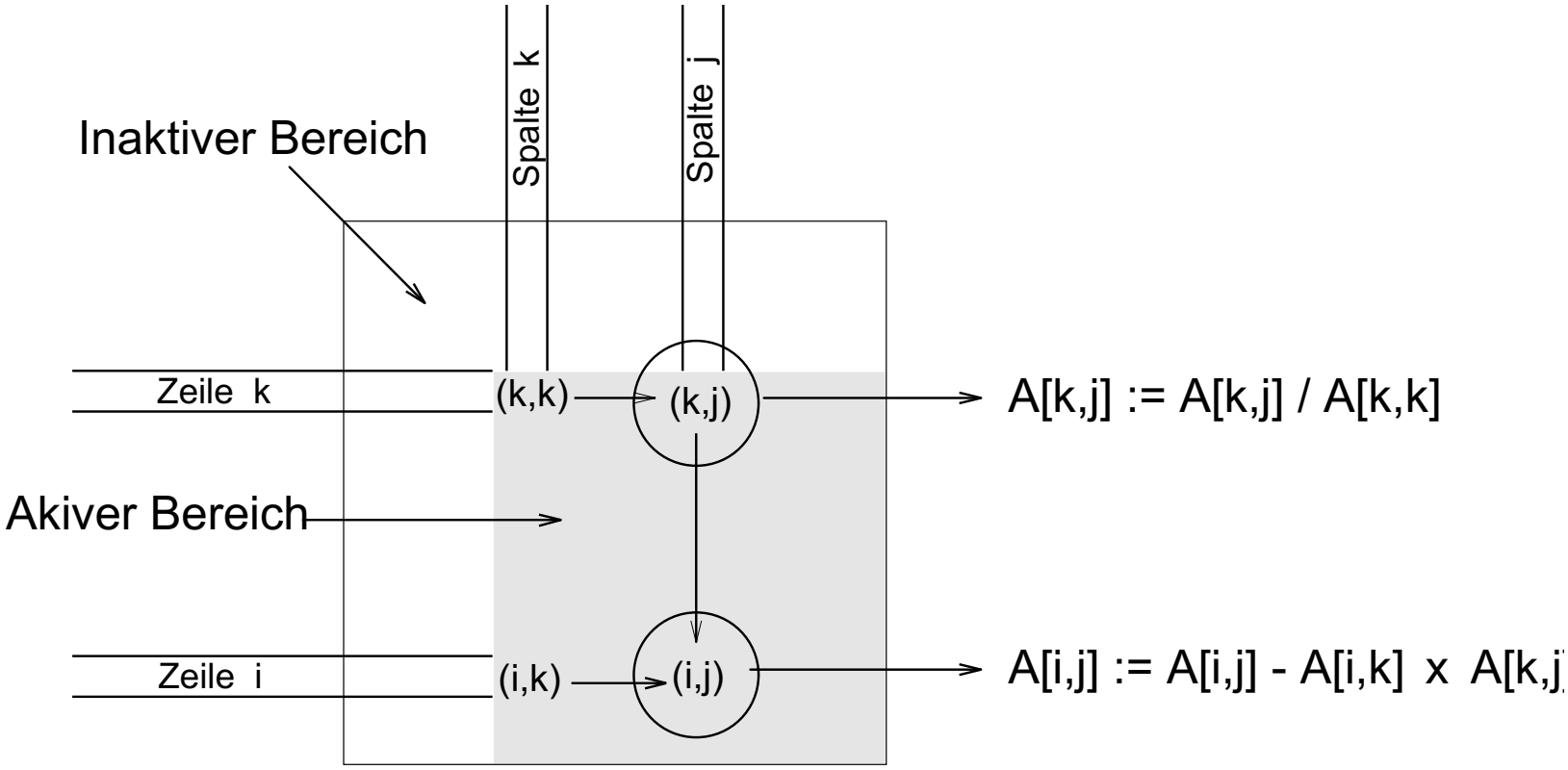
- Lösen eines linearen Gleichungssystems Ax=b
- Verfahren:
  - Bestimme Matrix L und Matrix U mit
    - A = L U
    - L untere Dreiecksmatrix mit Einheitendiagonale
    - U obere Dreiecksmatrix
  - Löse zunächst Ly=b und dann Ux=y
    - Lösungen lassen sich „ablesen“ (Dreicksmatrizen)

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ L_{2,1} & 1 & 0 \\ L_{3,1} & L_{3,2} & 1 \end{pmatrix} \bullet \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

Beispiel: LU Faktorisierung

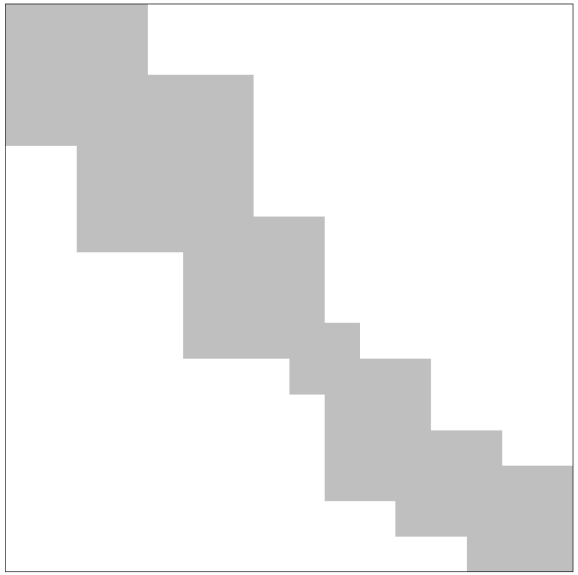
```
procedure LU Factorization (A)
begin
   for k := 1 to n do
      for j := k to n do
         A[ j, k] := A[ j, k]/A[k, k];
      endfor;
      for j := k + 1 to n do
         for i := k + 1 to n do
            A[i, j] := A[i, j] - A[i, k] × A[k, j ];
         endfor;
      endfor;
   /* After this iteration, column A[k + 1 : n, k] is logically the kth
   column of L and row A[k, k : n] is logically the kth row of U. */
   endfor;
end
```

Beispiel: LU Faktorisierung



Randomisierte Blockverteilung

- In manchen Fällen erzeugt auch eine zyklische Blockverteilung eine ungleichmäßige Lastverteilung.
- **Beispiel:** Prozesse auf Diagonale (P_0, P_5, P_{10} und $P_{15)}$ erhalten mehr Tasks als die anderen Prozesse.



P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

- **Lösung:** Randomisierte Verteilung: Zufallspermutation der Blöcke.

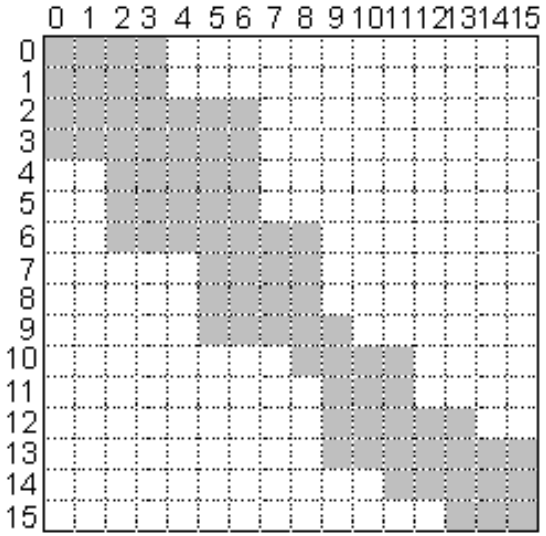
Randomisierte Blockverteilung

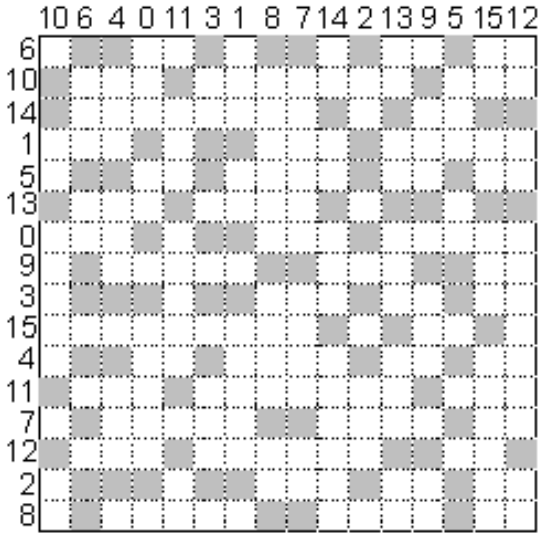
V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

random(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]

Zuordnung = 8 2 6 0 3 7 11 1 9 5 4 10

P_0 P_1 P_2 P_3





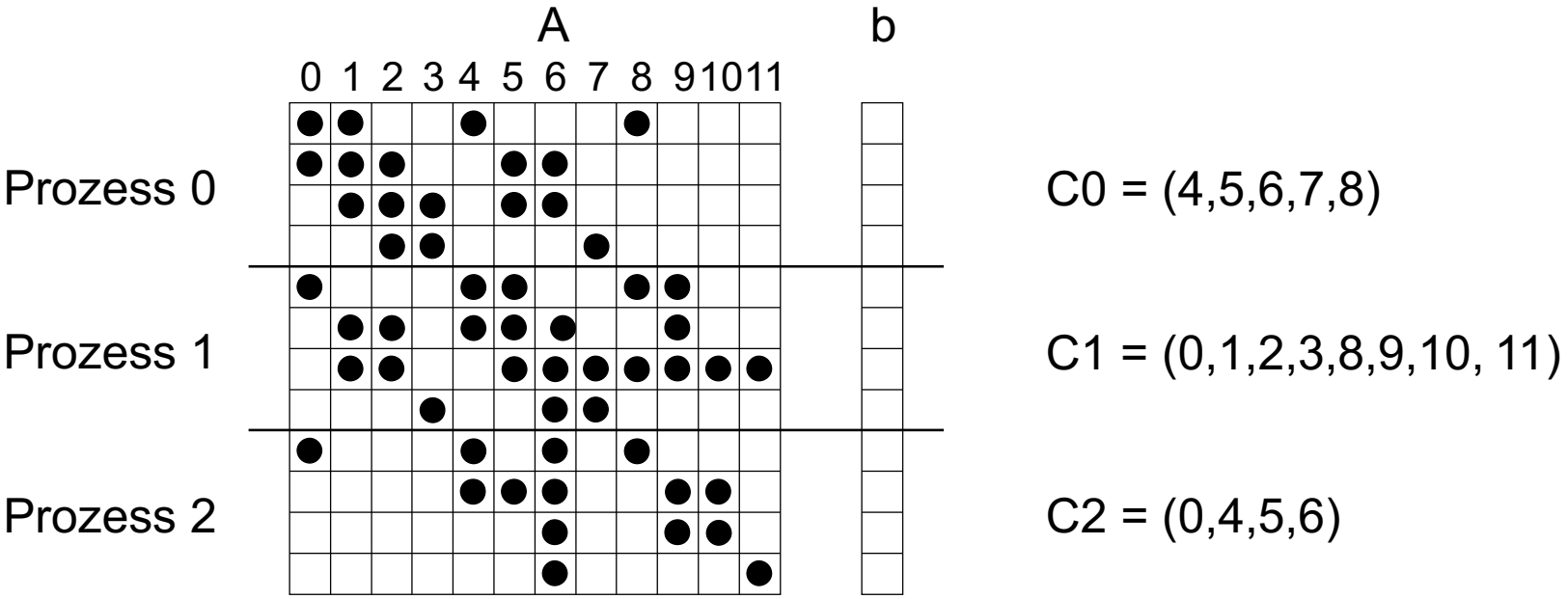
P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

Lastverteilung durch Task- Partitionierung

- Lastverteilung durch Task-Partitionierung beruht auf Partitionierung des Task-Interaktionsgraphs mittels Graphpartitionierungsverfahren.
  - Knotenmenge des Graphs soll so in p Teile partitioniert werden, dass
    - alle Partitionen möglichst gleich groß sind (bzgl. Der Summe der Task Größen) und
    - die Anzahl der durchtrennten Kanten minimiert wird.
  - NP vollständiges Problem, es gibt aber gute Heuristiken.
- Statischer Task-Interaktionsgraph erforderlich.
- Task Größe muss bekannt sein.

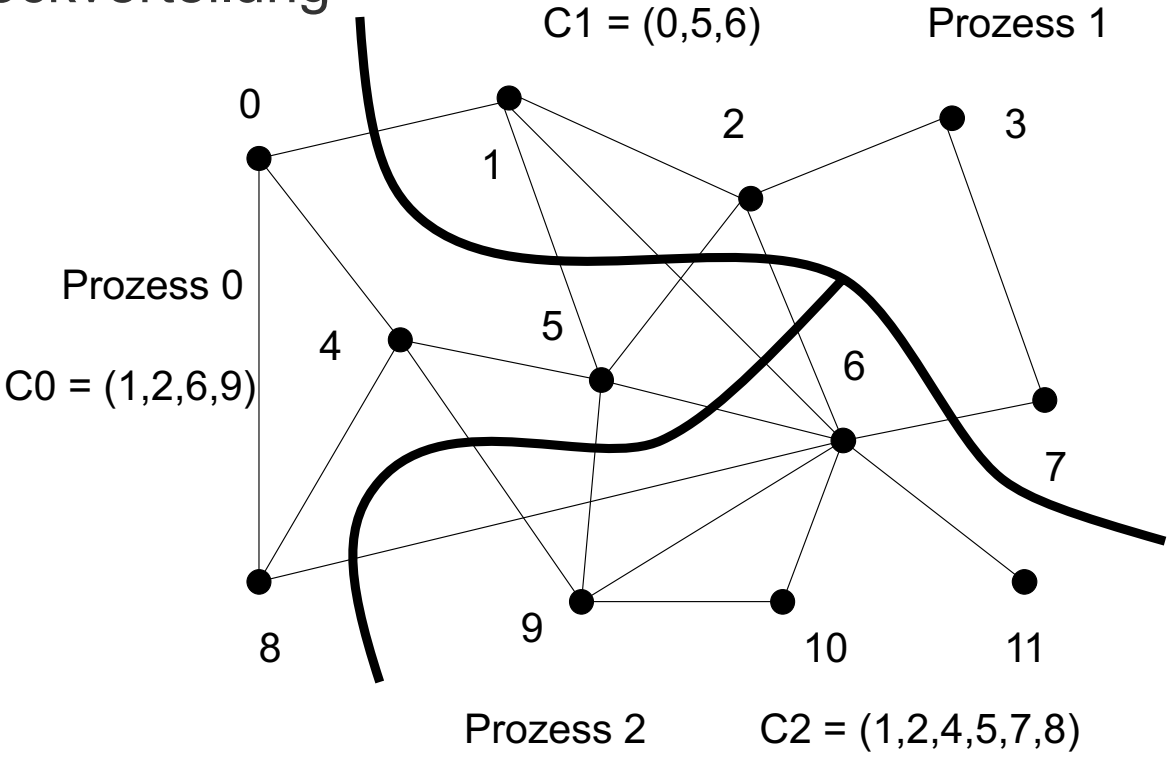
Beispiel: Sparse Matrix-Vektor Multiplikation

- Lastverteilung durch 1D Blockverteilung
- Liste C_i zeigt Interaktionen der Tasks von Prozess i mit Tasks die auf andere Prozesse abgebildet sind.



Beispiel: Sparse Matrix-Vektor Multiplikation

- Lastverteilung durch Task-Partitionierung
- Task-Interaktion über Prozessgrenzen ist geringer als bei Blockverteilung



Überblick: Dynamische Lastverteilungsverfahren

- Dynamische Lastverteilungsverfahren sind erforderlich, falls
  - statische Verfahren zu einer ungleichmäßigen Lastverteilung führen oder
  - der Task-Interaktionsgraph nicht statisch bekannt ist.
- Dynamische Lastverteilungsverfahren können
  - **zentral** oder
  - **verteilt**

  realisiert werden.

Zentraler Ansatz für dynamische Lastverteilung

- Alle ausführbaren Tasks werden in einer zentralen Datenstruktur (**Task-Pool**) gehalten.
  - Wenn ein Prozess keinen Task zur Ausführung verfügbar hat, entnimmt er einen Task aus dem Task-Pool.
  - Dynamisch erzeugte Tasks werden in den Task-Pool eingestellt.
- Ist für die Verwaltung des Pools ein spezieller Prozess zuständig, so heißt dieser **Master-Prozess**, die ausführenden Prozesse heißen dann **Slave-Prozesse**.

Zentraler Ansatz für dynamische Lastverteilung

- Beispiel: Sortieren der Zeilen einer nxn Matrix

```
for (i=0; i<n; i++)
    sort(A[i], n);
```

- Je nach den Werten der Einträge kann das Sortieren der Zeilen unterschiedlich lange dauern.
  - Statische Zuordnung führt dann zu ungleichmäßiger Lastverteilung.
- Dynamischer Ansatz: **Self-Scheduling** von Schleifen.
  - Task-Pool enthält Indizes von noch nicht sortierten Zeilen.
  - Prozesse entnehmen Indizes aus Task-Pool und führen den zugehörigen Sortier-Task aus.

Zentraler Ansatz für dynamische Lastverteilung

- Problem bei zentralem Ansatz: Schlechte Skalierbarkeit
  - Bei einer großen Anzahl von Prozessen wird der Zugriff auf den zentrale Task-Pool zum Flaschenhals.
- Abhilfe: **Chunk-Scheduling**
  - Es wird pro Anfrage eine Gruppe von Tasks (Chunk) aus dem Task-Pool entnommen.
  - Bei vielen Tasks pro Chunk kann wiederum eine ungleichmäßige Lastverteilung auftreten.
  - Vermeidung von ungleichmäßiger Lastverteilung durch dynamische Reduzierung der Chunk-Größe zum Ende der Berechnung.

Verteilter Ansatz für dynamische Lastverteilung

- Prinzip: Jeder Prozess hat lokalen Task-Pool.
- Tasks können von anderen Prozessen empfangen, bzw. zu anderen Prozessen geschickt werden.
- Parameter
  - Wer initiiert einen Task-Transfer?
    - Sender oder Empfänger
  - Wann wird ein Task-Transfer vorgenommen?
    - Schwellenwert für Größe des lokalen Task Pools
  - Wie werden Sender- und Empfängerprozess gepaart?
    - z.B. round-robin oder randomisiert
  - Wie viele Tasks werden auf einmal transferiert?

3. Design paralleler Programme

1. Wichtige Begriffe und Definitionen
2. Dekompositionstechniken
3. Lastverteilungsverfahren
4. **Parallele Algorithmenmodelle**

Parallele Algorithmenmodelle

- Parallele Algorithmenmodelle stellen typische Kombi-nationen von Dekompositions- und Lastverteilungs-verfahren dar.
- Beispiele:
  - Daten paralleles Modell
  - Task paralleles Modell
  - Worker-Pool Modell
  - Master-Slave Modell

Daten paralleles Modell

- Task-Dekomposition: statisch (Daten-Dekomposition)
- Lastverteilung: statisch
- Typische Eigenschaften:
  - Alle Tasks führen ähnliche Operationen auf unterschiedlichen Daten aus.
  - Berechnung verläuft in Phasen.
    - Zwischen den Phasen erfolgt Synchronisation der Berechnung und Kommunikation von Daten.
  - Grad der Nebenläufigkeit steigt mit zunehmender Problemgröße.
- Beispiel: Dense Matrix-Vektor Multiplikation

Task paralleles Modell

- Task-Dekomposition: statisch (führt zu statischem Task-Interaktionsgraph)
- Lastverteilung: statisch durch Partitionierung des Task Interaktionsgraphen.
- Typische Eigenschaften:
  - Geeignet für Probleme, bei denen die Größe der Daten eines Tasks vergleichsweise groß ist gegenüber seiner Größe (Berechnungsdauer).
- Beispiel: Sparse Matrix-Vektor Multiplikation

Worker-Pool Modell

- Task-Dekomposition: statisch oder dynamisch
- Lastverteilung: dynamisch (zentral oder verteilt)
- Typische Eigenschaften:
  - Jeder Task kann von jedem Prozess ausgeführt werden.
  - Für Probleme, bei denen die Größe der Daten eines Tasks vergleichsweise klein ist gegenüber seiner Größe (Berechnungsdauer).
- Beispiel:
  - Statische Task-Dekomposition: Self Scheduling von Schleifen.
  - Dynamische Task-Dekomposition: Parallele Baumsuche.

Master-Slave Modell

- Auch **Manager-Worker** Modell genannt.
- Ein oder mehrere Manager Prozesse erzeugen Tasks und ordnen sie Worker Prozessen zu.
- Lastverteilung:
  - semi-statisch, falls Task-Größe bekannt.
  - dynamisch bei unbekannter Task-Größe oder wenn die Erzeugung der Task zeitaufwändig ist.
- Hierarchische Organisation möglich
  - Top-level Manager verteilen Tasks an untergeordnete Manager.
  - Untergeordnete Manager führen weitere Dekomposition durch
- Beispiel: Seti@Home