

Algorithmik kontinuierlicher Systeme

Matrizen - Datenstrukturen und praktische Verfahren

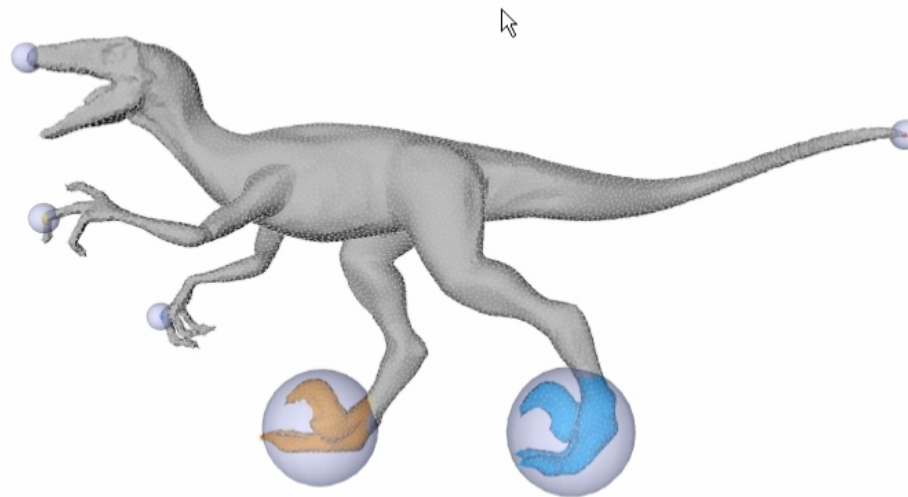


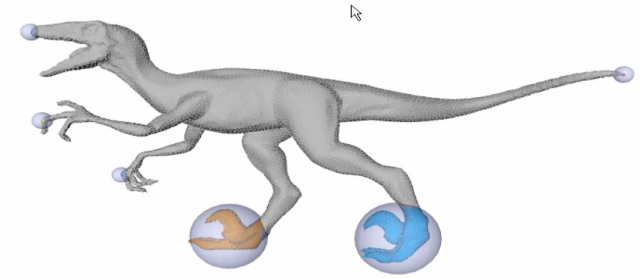
- Oft ist die Dimension von Matrizen und Vektoren ist bekannt und fix
- Elementare Geometrische Anwendungen:
 - Matrix beschreibt meist Transformationen von Vektoren im 2D bzw. 3D (d.h. Dimension klein (2x2, 3x3, 4x4,))
 - Matrizen in der Regel als 2D-Array dargestellt
 - Fixe Datenstruktur günstig
- Beispiel: Rotationsmatrix (um z-Achse)

$$\mathbf{J}_z = \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 \\ \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Anspruchsvolle geometrische Probleme:

z.B. Animation mit ARAP (= as-rigid-as-possible)





- Anspruchsvolle geometrische Probleme: z.B. Animation mit ARAP
- Gegeben
 - Ruhe-Positionen $M = \{v_0, v_1, \dots, v_n\}$ (5.000 – 50.000 3D-Punkte)
 - Ziel-Position für einige Punkte $\{\hat{v}_3 = c_3, \hat{v}_9 = c_9, \dots \hat{v}_l = c_l\}$ (50 – 500 3D-Punkte)
- Gesucht wird Ziel-Position der übrigen Punkte \hat{v}_j so dass

$$E(\hat{M}) = \sum_{v_j \in M} \sum_{v_i \in N(v_j)} \|(\hat{v}_i - \hat{v}_j) - \mathbf{R}_j(v_i - v_j)\|^2 = \min$$

- Dazu sind (u.a.) viele Gleichungssysteme mit bis zu $3 \cdot 49.500$ Unbekannten zu lösen
- Koeffizienten-Matrix ist extrem dünn besetzt
- Auch die orthogonalen Matrizen \mathbf{R}_j sind unbekannt

Matrizen und Vektoren in Gleichungssystemen

- Meist sehr große Dimensionen
z.B. Diskretisierung kontinuierlicher Strukturen
- 10^7 Unbekannte keine Seltenheit (Spitze bis 10^{13} , Stand 2016)
 -> Matrix hat dann $10^7 \times 10^7$ Elemente oder mehr
- Aber: immer spezielle Struktur:
 - z.B. symmetrisch, Band-Struktur, dünn besetzt
- Bei naiver Datenstruktur astronomischer Speicheraufwand:
 - schon bei $10^7 \times 10^7 = 10^{14}$ mit floating points benötigt man 800 Terrabyte (double precision)!
- Spezielle Matrixstruktur muss unbedingt ausgenützt werden

- Beispiel:
Temperaturverteilung (auf einer quadratischen Platte)
(bei festen Randbedingungen im Gleichgewichtszustand)
 - Die (noch unbekannte) Funktion $u(x,y)$ beschreibt die Temperatur um Plattenpunkt $x,y \in [0,a]$
 - Physik und Mathematik:
Im Innern gilt folgende *partielle Differentialgleichung* (Laplace):

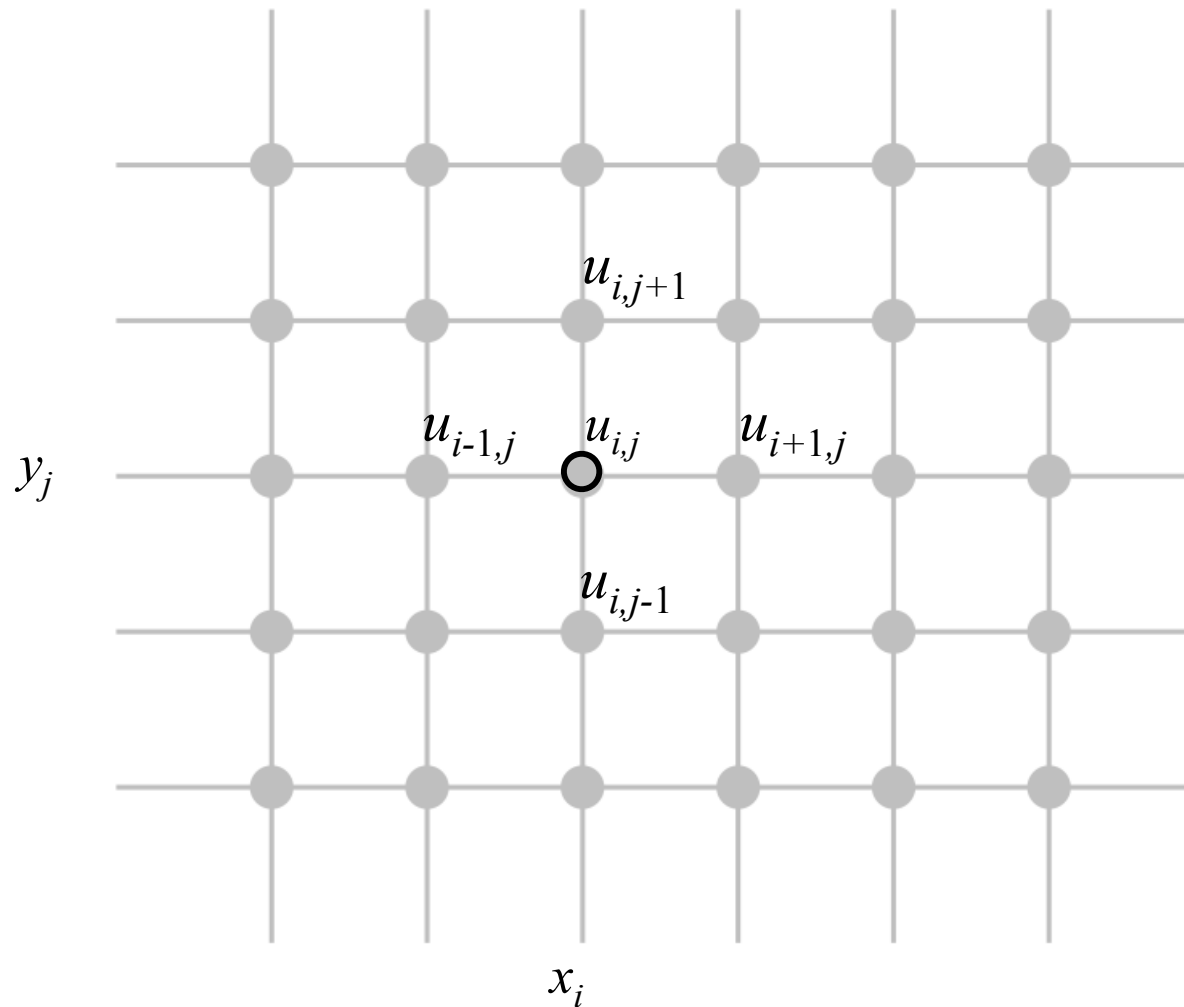
$$\Delta u = u_{xx} + u_{yy} = 0$$

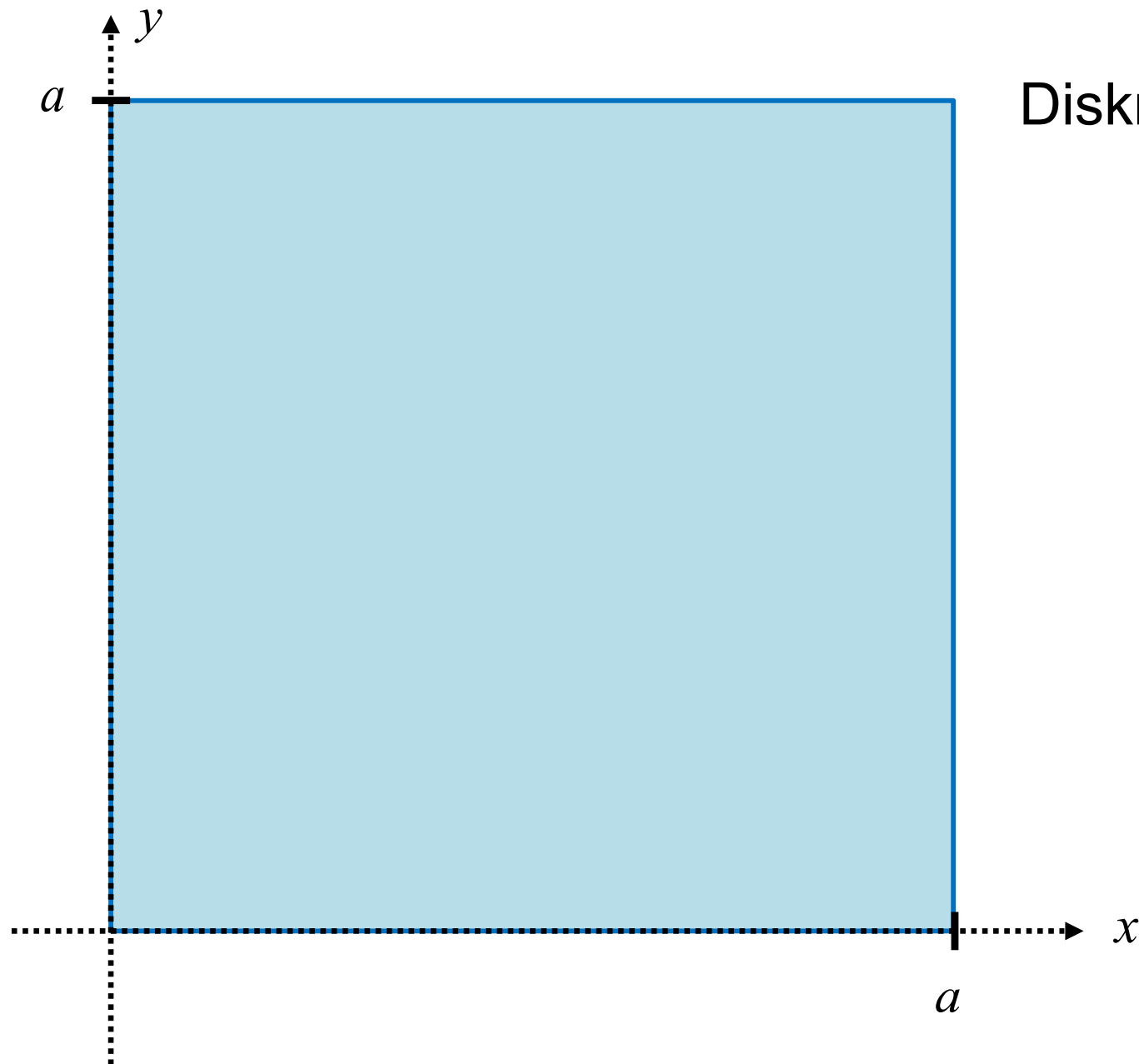
- Diskretisierung:
 - statt kontinuierlicher Funktion $u(x,y)$ betrachte man (viele) Sample-Punkte $u_{i,j} = u(x_i, y_j)$
 - Statt Differentialgleichung eine *Differenzengleichung*:

$$u_{i+1,j} + u_{i,j+1} + u_{i-1,j} + u_{i,j-1} - 4u_{i,j} = 0$$

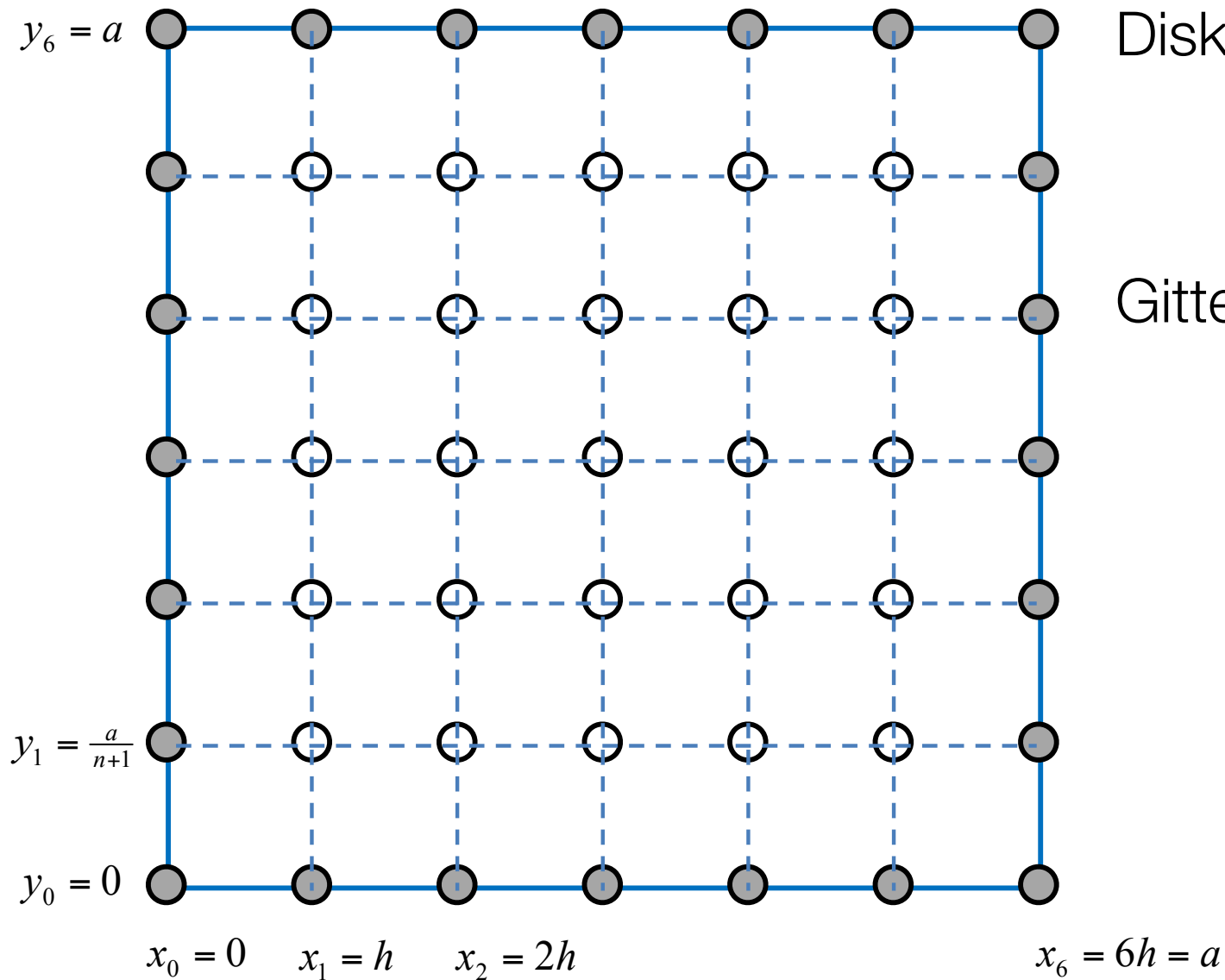
- Beispiel: Diskretisierung Temperaturverteilung auf Platte

$$u_{i+1,j} + u_{i,j+1} + u_{i-1,j} + u_{i,j-1} - 4u_{i,j} = 0$$



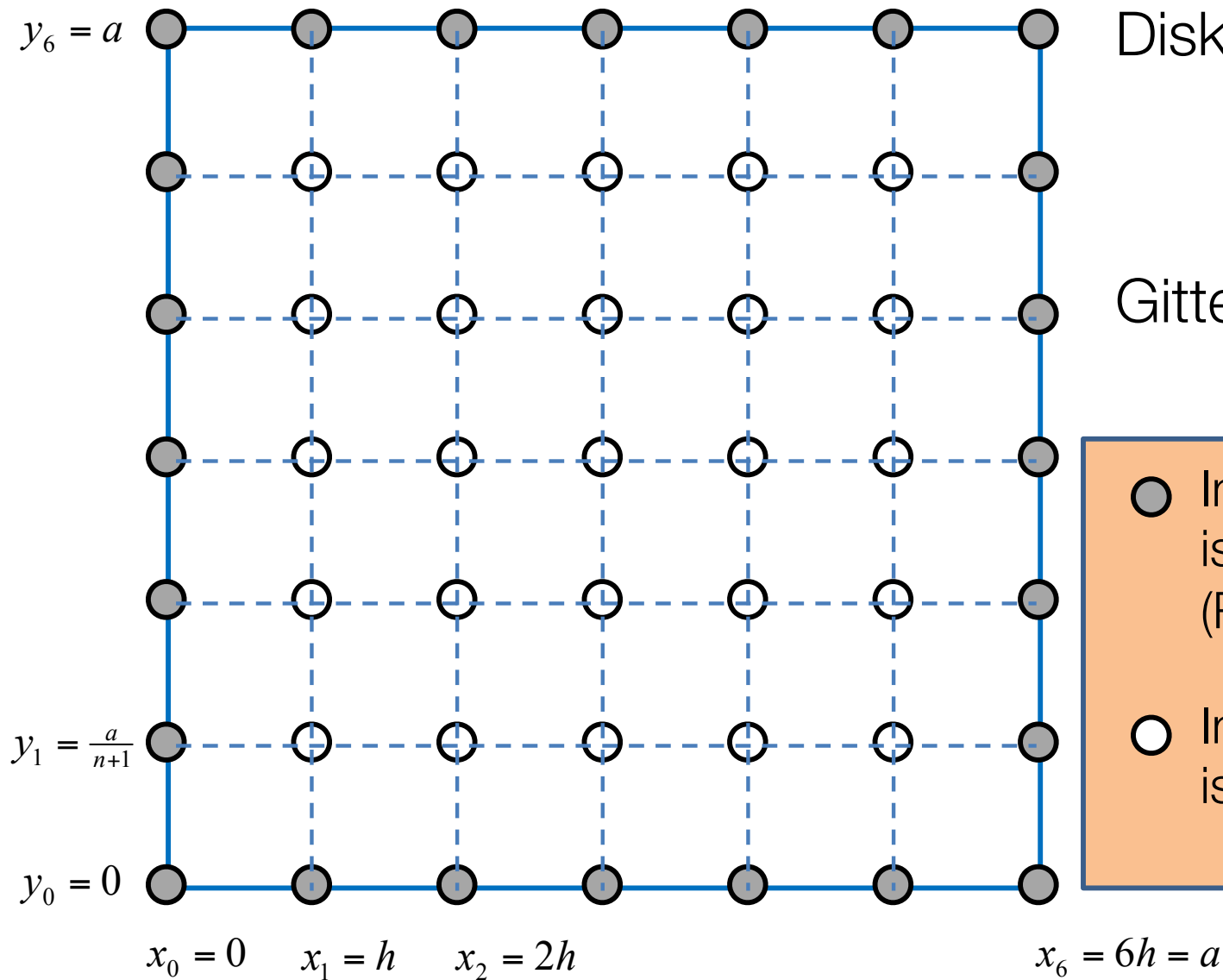


Diskretisierung für $n=5$



Diskretisierung für $n=5$

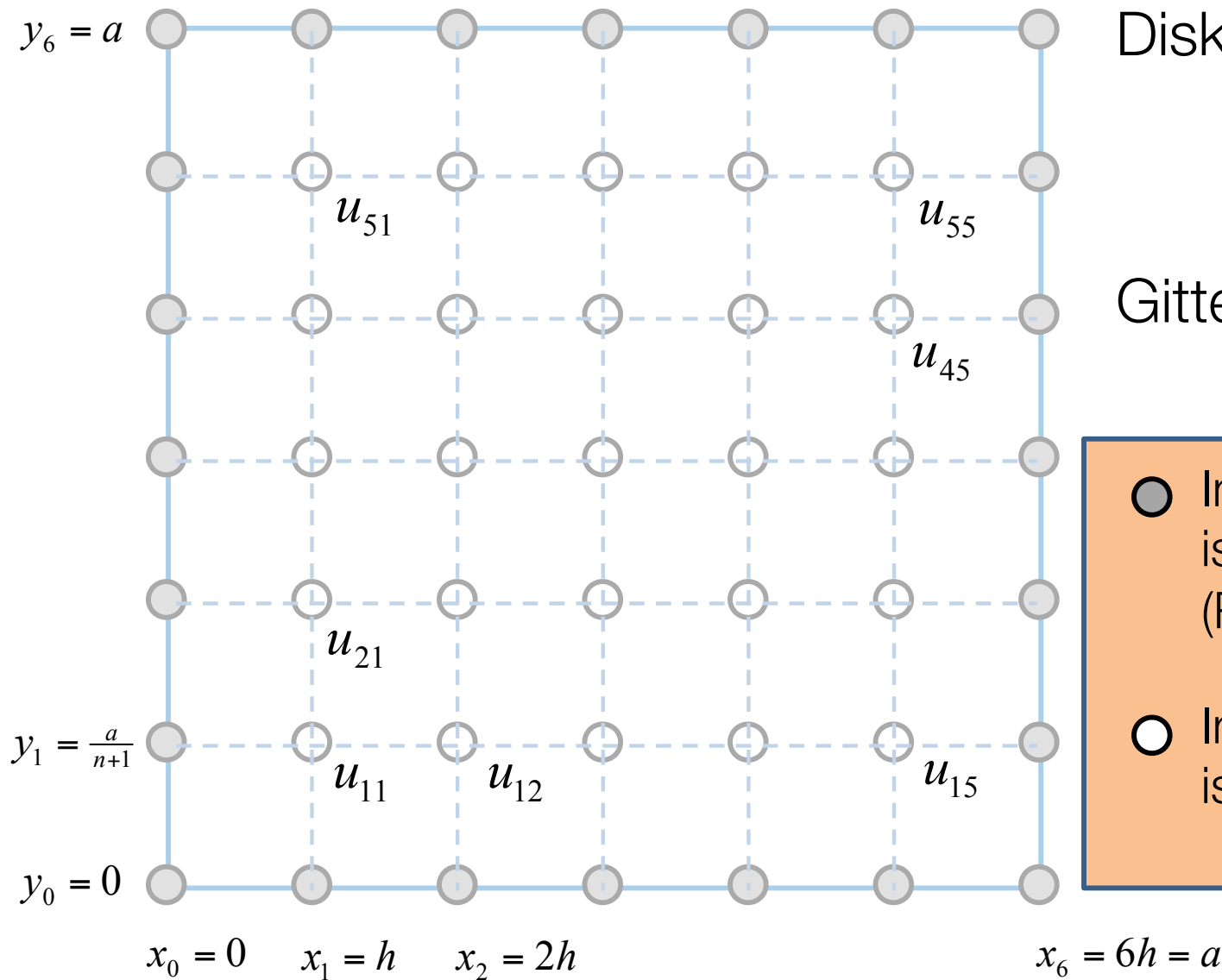
Gitterweite : $h = \frac{a}{n+1}$



Diskretisierung für $n=5$

Gitterweite : $h = \frac{a}{n+1}$

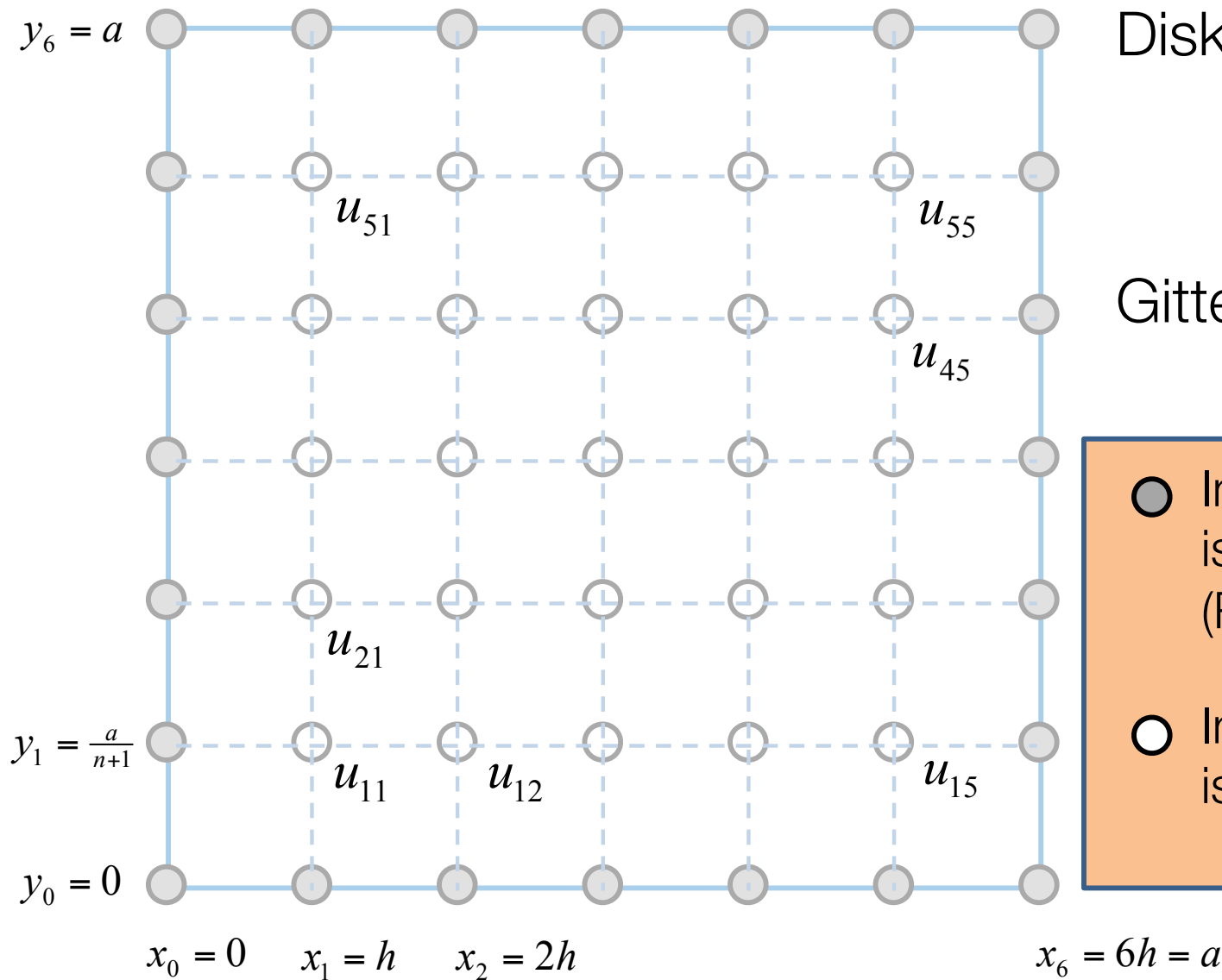
- In diesen Punkten ist u bekannt (Randbedingungen)
- In diesen Punkten ist u unbekannt



Diskretisierung für $n=5$

Gitterweite : $h = \frac{a}{n+1}$

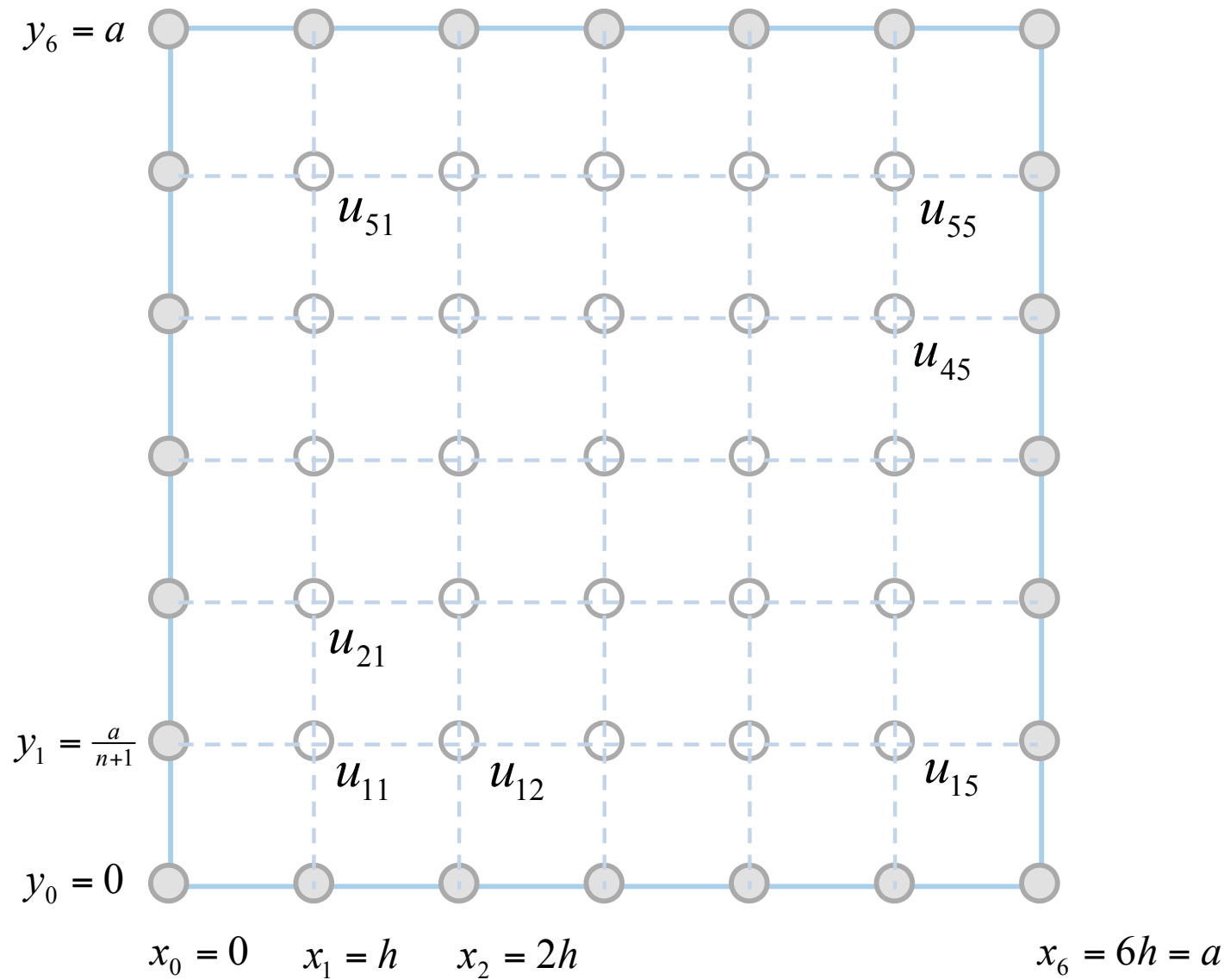
- In diesen Punkten ist u bekannt (Randbedingungen)
- In diesen Punkten ist u unbekannt



Diskretisierung für $n=5$

Gitterweite : $h = \frac{a}{n+1}$

- In diesen Punkten ist u bekannt (Randbedingungen)
- In diesen Punkten ist u unbekannt



$$\begin{bmatrix} u_{11} \\ \vdots \\ u_{15} \\ u_{21} \\ \vdots \\ u_{25} \\ \vdots \\ \vdots \\ u_{51} \\ \vdots \\ u_{55} \end{bmatrix}$$

$$\begin{bmatrix}
 \begin{bmatrix} 4 & -1 \\ -1 & 4 \end{bmatrix} & \begin{bmatrix} -1 \\ -1 \end{bmatrix} \\
 & \ddots \\
 & & \begin{bmatrix} 4 & -1 \\ -1 & 4 \end{bmatrix} & \begin{bmatrix} -1 \\ -1 \end{bmatrix} \\
 & & & \ddots \\
 & & & & \begin{bmatrix} 4 & -1 \\ -1 & 4 \end{bmatrix}
 \end{bmatrix}
 \begin{bmatrix}
 u_{11} \\
 \vdots \\
 u_{1n} \\
 u_{21} \\
 \vdots \\
 u_{2n} \\
 \vdots \\
 u_{nn}
 \end{bmatrix}
 =
 \begin{bmatrix}
 * \\
 \vdots \\
 * \\
 * \\
 \vdots \\
 * \\
 \vdots \\
 *
 \end{bmatrix}$$

$-(u_{i+1,j} + u_{i,j+1} + u_{i-1,j} + u_{i,j-1} - 4u_{i,j}) = 0$

Rechte Seite ergibt sich aus den Randbedingungen

- Um zuverlässige Ergebnisse zu erhalten muss man mit Schrittweiten 10^{-1} bis 10^{-2} oder kleiner wählen.
- Somit hat der unbekannte Vektor die Größe 100 bis 10.000 oder größer
- Die Koeffizienten-Matrix explizit abgespeichert (mit 8 Bytes pro Eintrag) benötigt bis zu 800 MB
- Aber es gibt nur 50.000 von Null verschiedenen Einträge!
- Dünn besetzte Matrizen mit oder ohne Mustern treten bei Gleichungssysteme häufig auf!

- Hintergrund: Lösung partieller Differentialgleichungen, (hier konkret die Poissongleichung, Laplace-Operator)

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \quad \text{in } \Omega =]0,1[^2$$

zusätzlich geeignete Randbedingungen, z.B.:

$$u(x,0) = u_S(x), u(x,1) = u_N(x), u(0,y) = u_O(y), u(1,y) = u_W(y),$$

- Weitere Verallgemeinerungen
Skalare, lineare, *elliptische partielle Differentialgleichung* in 2D (oder 3D)
- Näherung der partiellen Ableitungen mit *Finiten Differenzen* auf einem uniformen Gitter

- Probleme dieser Art treten in zahlreichen Anwendungen auf, z.B.:
 - Temperaturverteilung in inhomogenen Medien,
 - Physik: Elektrostatik, Gravitationspotential, ...
 - Fluidmechanik (Druck-Korrektur),
 - Umwelttechnik (Grundwasserströmungen),
 - Medizin (bioelektrische Felder), u.v.m.
 - Diffusionsprozesse (in Materialwissenschaften, Verfahrenstechnik, biomedizin, Umwelttechnik)

- Einfache dünn besetzte, strukturierte Matrizen

$$\begin{bmatrix} * & & & & & & \\ & * & & & & & \\ & & * & & & & \\ & & & * & & & \\ & & & & * & & \\ & & & & & * & \\ & & & & & & * \end{bmatrix}$$

Diagonal

$$\begin{bmatrix} * & * & & & & & \\ * & * & * & & & & \\ & * & * & * & & & \\ & & * & * & * & & \\ & & & * & * & * & \\ & & & & * & * & * \\ & & & & & * & * & * \end{bmatrix}$$

Tridiagonal

$$\begin{bmatrix} * & * & * & & & & \\ * & * & * & * & & & \\ * & * & * & * & * & & \\ & * & * & * & * & * & \\ & & * & * & * & * & * \\ & & & * & * & * & * & * \\ & & & & * & * & * & * & * \\ & & & & & * & * & * & * & * \end{bmatrix}$$

Bandmatrix der Bandbreite b

$$\begin{bmatrix} * & * & * & & & & \\ * & * & * & & & & \\ * & * & * & & & & \\ & & & * & * & * & \\ & & & * & * & * & \\ & & & * & * & * & \\ & & & & & & * & * & * \\ & & & & & & * & * & * \\ & & & & & & * & * & * \end{bmatrix}$$

Blockdiagonal

- Idee: Speichere nur Einträge die von 0 verschieden sind

$$A = \begin{bmatrix} 7 & 0 & 0 & 13 & 0 \\ 0 & 0 & 4 & 0 & 8 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{bmatrix}$$

- Zeilenweise ausgelesen der von 0 verschiedenen Werte:

$$val = [7, 13, 4, 8, 11, 3]$$

- Wir müssen noch wissen an welchen Stellen diese Werte stehen!

- Also speichern wir auch den zugehörigen Zeilen- und Spaltenindex

$$A = \begin{bmatrix} 7 & 0 & 0 & 13 & 0 \\ 0 & 0 & 4 & 0 & 8 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{bmatrix}$$

$$val = [7, 13, 4, 8, 11, 3]$$

$$col_ind = [1, 4, 3, 5, 2, 3]$$

$$row_ind = [1, 1, 2, 2, 3, 4]$$

Beachte:
Indizierung beginnt bei 1

- Wir brauchen nur noch $3 \cdot \#(non_zero)$ Werte speichern

- Noch effizienter: Statt *row_ind* einen Pointer *row_ptr* der den Zeilenanfang jeder Zeile markiert

$$A = \begin{bmatrix} 7 & 0 & 0 & 13 & 0 \\ 0 & 0 & 4 & 0 & 8 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{bmatrix}$$

$$val = [7, 13, 4, 8, 11, 3]$$

$$col_ind = [1, 4, 3, 5, 2, 3]$$

$$row_ptr = [1, 3, 5, 6, 7]$$

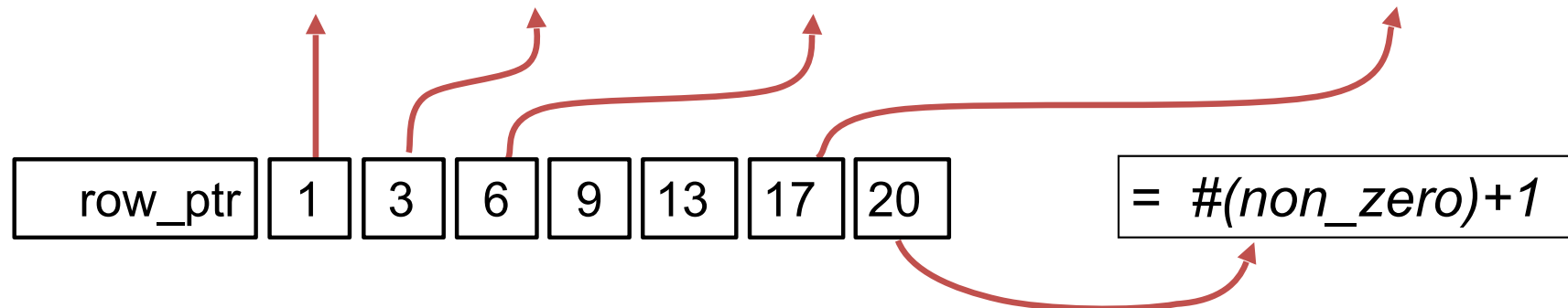
Beachte:
Indizierung beginnt bei 1

- *row_ptr* zeigt auf die Elemente der anderen Arrays
- das *row_ptr* Array hat $\#(rows)+1$ Einträge
- der letzte Eintrag in *row_ptr* ist stets $\#(non_zero)+1$
- Das ganze heißt *Compressed Row Storage (CRS)*

- Beispiel
()

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

val	10	-2	3	9	3	7	8	7	3 ... 9	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	1 ... 5	6	2	5	6



- Speicheraufwand nur $2 \cdot \#(non_zero) + \#(rows) + 1$
- Erlaubt effiziente Matrix-Vektor Multiplikation:

```
for i = 1..n
  y[i] = 0
  for j = row_ptr[i]..row_ptr[i+1]-1
    y[i] = y[i] + val[j] * x[col_ind[j]]
```

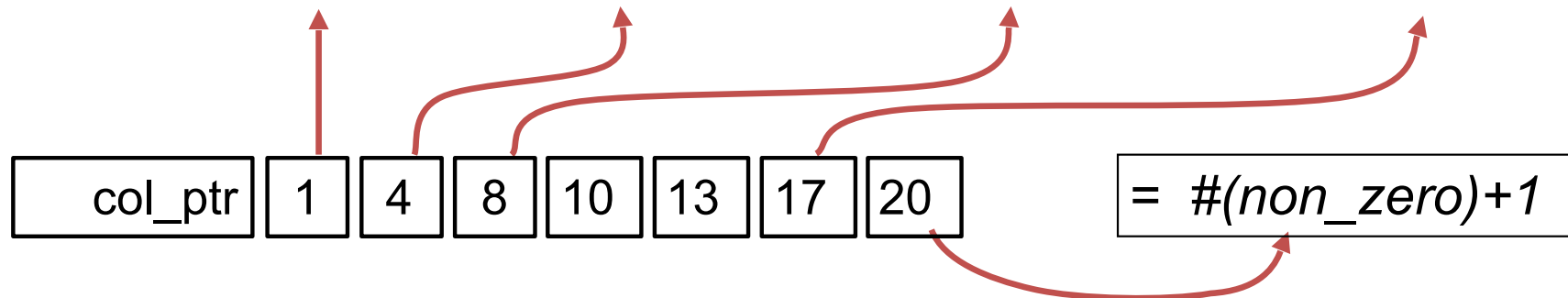
- Nachteil:
 - Einfügen und Löschen schwierig!
 - Arrays müssen umstrukturiert werden; ggf. Implementierung mit verketteten Listen, dafür wird Suchen allerdings teuer
 - Kein Random Access auf einzelne Elemente
 - In der jeweiligen Zeile: Suche auf sortiertem (Teil-)Array; $\log(n)$ mit *binary search*

- CRS für Blockmatrizen:
 - Block Compressed Row Storage (BCRS)
 - Zusammenfassen von dicht besetzten Gebieten in Blöcke
 - Jeder nicht-null Block wird intern normal gespeichert
 - Behandlung von Blöcken als Elemente für CRS
- Vorteil gegenüber CRS:
 - Falls Blöcke sehr dicht besetzt sind ist normale Speicherung effizienter
 - Trotzdem werden null-Gebiete nicht gespeichert!
 - Speziell für große Blockgrößen ist BCRS besser als CRS

- Analog zu CRS

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

val	10	3	3	9	7	8	4	8	8 ... 9	2	3	13	-1
row_ind	1	2	4	2	3	5	6	3	4 ... 5	6	2	5	6



- Speicherformate für dünn besetzte Matrizen:
 - Compressed Row Storage (CRS)
 - Compressed Column Storage (CCS)
 - Block Compressed Row Storage (BCRS)
 - Compressed Diagonal Storage (CDS)
 - Jagged Diagonal Storage (JDS)
 - Skyline Storage (SKS)
- Details siehe:
 - http://www.netlib.org/linalg/html_templates/node89.html
- Aktuelle Forschung dazu in Erlangen (im Schwerpunkt SPPEXA):

M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for modern processors with wide SIMD units*. SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: [10.1137/130930352](https://doi.org/10.1137/130930352). Preprint: [arXiv:1307.6209](https://arxiv.org/abs/1307.6209)

M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein: *GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems*. International Journal of Parallel Programming (2016). DOI: [10.1007/s10766-016-0464-z](https://doi.org/10.1007/s10766-016-0464-z). Preprint: [arXiv:1507.08101](https://arxiv.org/abs/1507.08101)

- Die „Besetzungs-“Struktur einer Matrizen kann durch Graphen dargestellt werden!
- Betrachtet wird ein Graph zu einer $n \times n$ -Matrix $A = (a_{ij})$

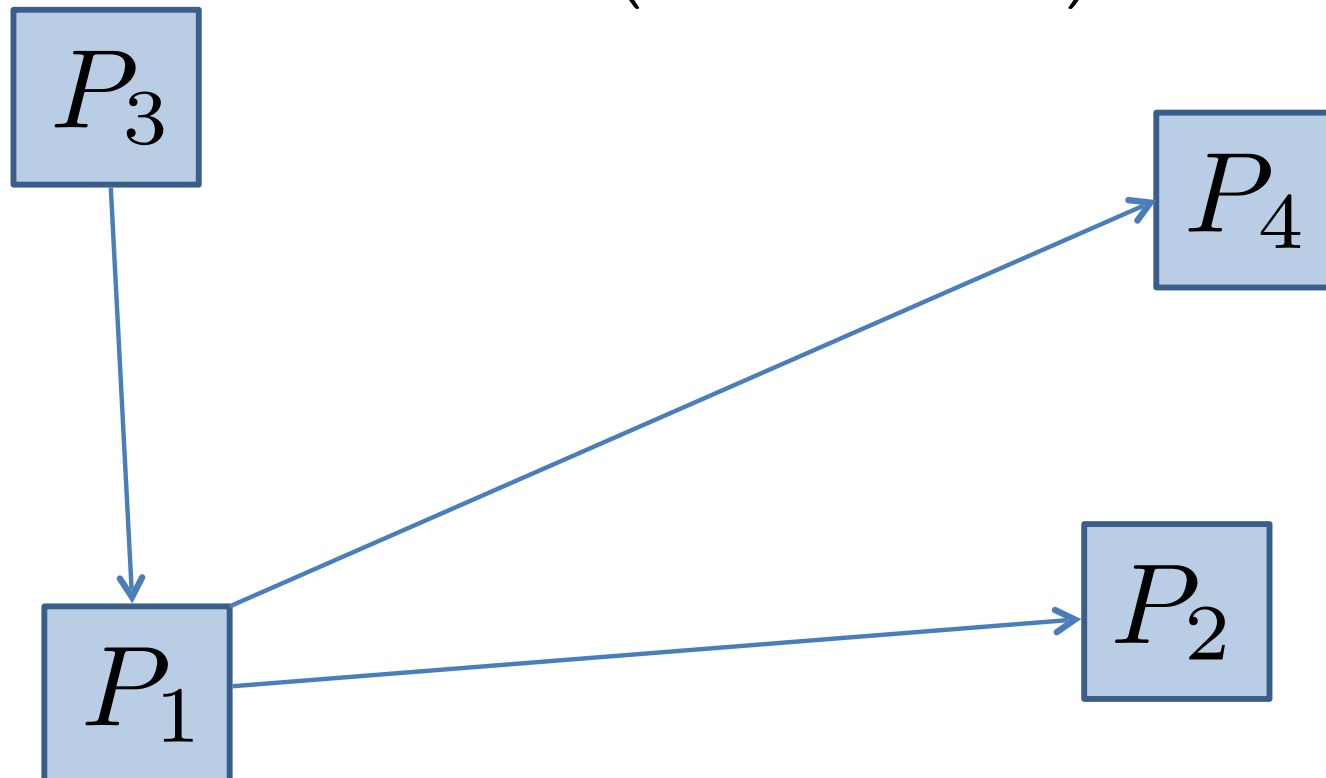
$G(V,E)$ ein Graph mit n Knoten $V = \{ P_i, i=1, \dots, n \}$;
eine gerichtete Kante P_i nach P_j existiert, wenn $a_{ij} \neq 0$.

- Umgekehrt, ist die Adjazenzmatrix eines (gerichteten) Graphen $G(V,E)$ mit n Knoten eine $n \times n$ -Matrix mit

$$a_{ij} = \begin{cases} 1, & \text{falls Kante von } P_i \text{ nach } P_j \\ \text{sonst} & \end{cases}$$

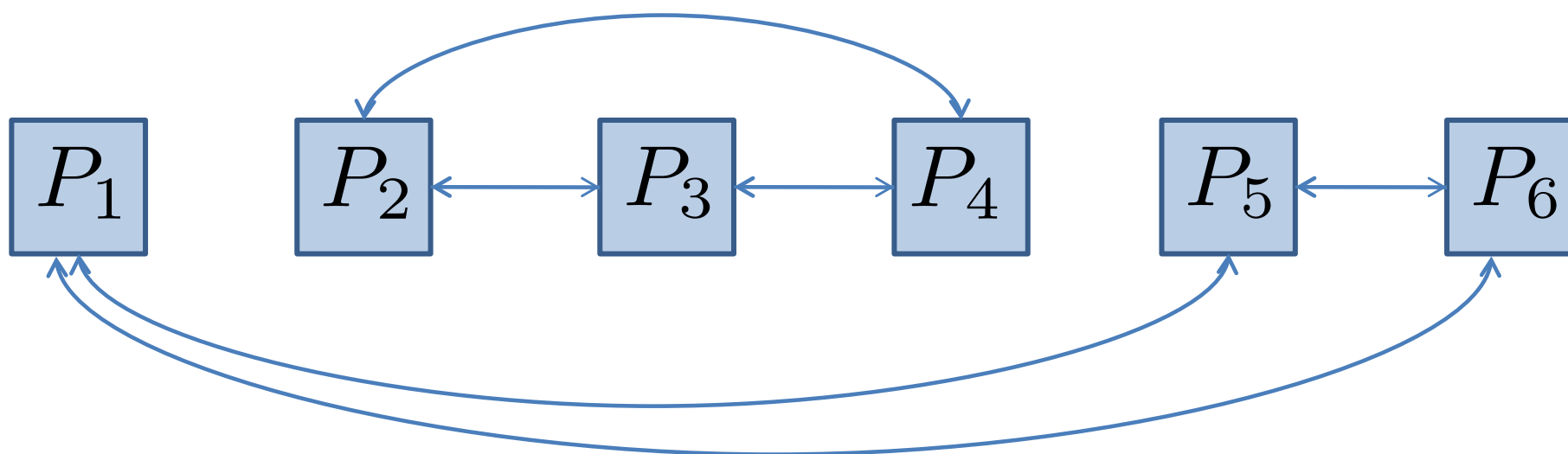
- Beispiel

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

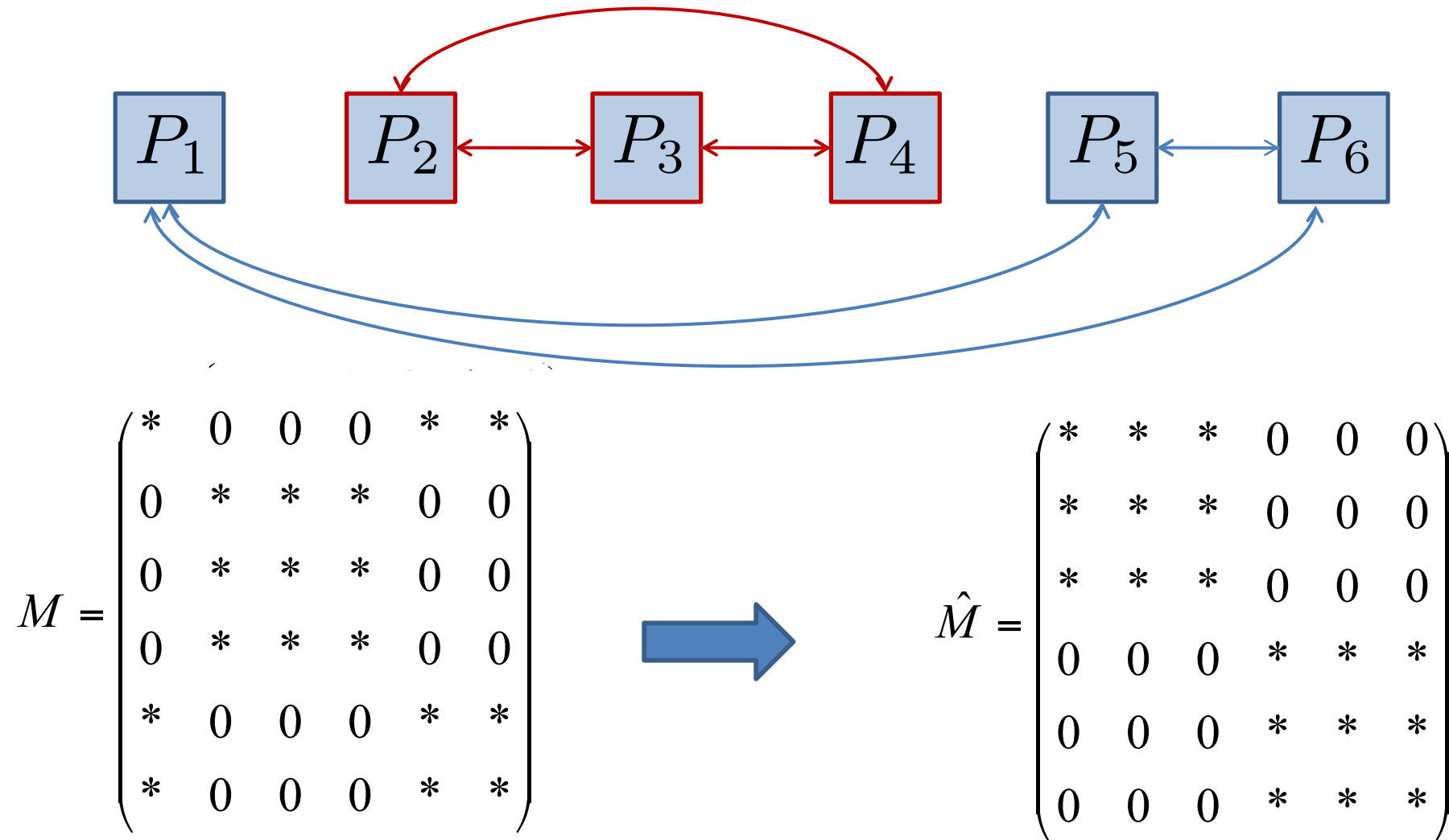


- Anwendung:
Graph zu beliebiger Matrix

$$M = \begin{pmatrix} * & 0 & 0 & 0 & * & * \\ 0 & * & * & * & 0 & 0 \\ 0 & * & * & * & 0 & 0 \\ 0 & * & * & * & 0 & 0 \\ * & 0 & 0 & 0 & * & * \\ * & 0 & 0 & 0 & * & * \end{pmatrix}$$

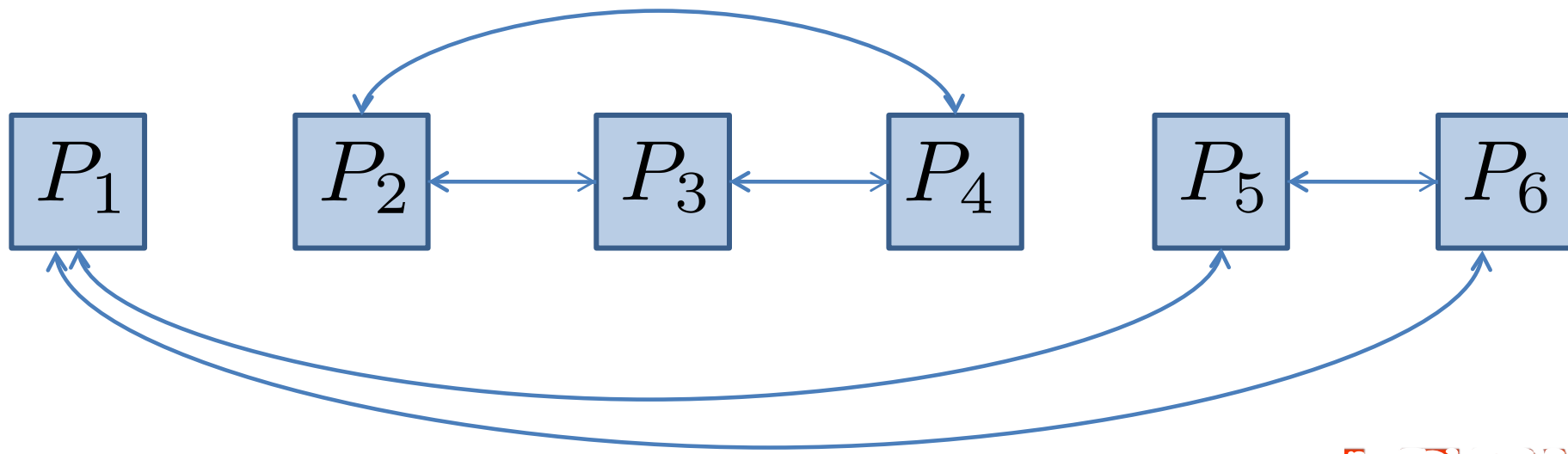


Beispiel: man die Knoten geschickt umsortieren, d.h. Spalten und Zeilen permutieren und erhält

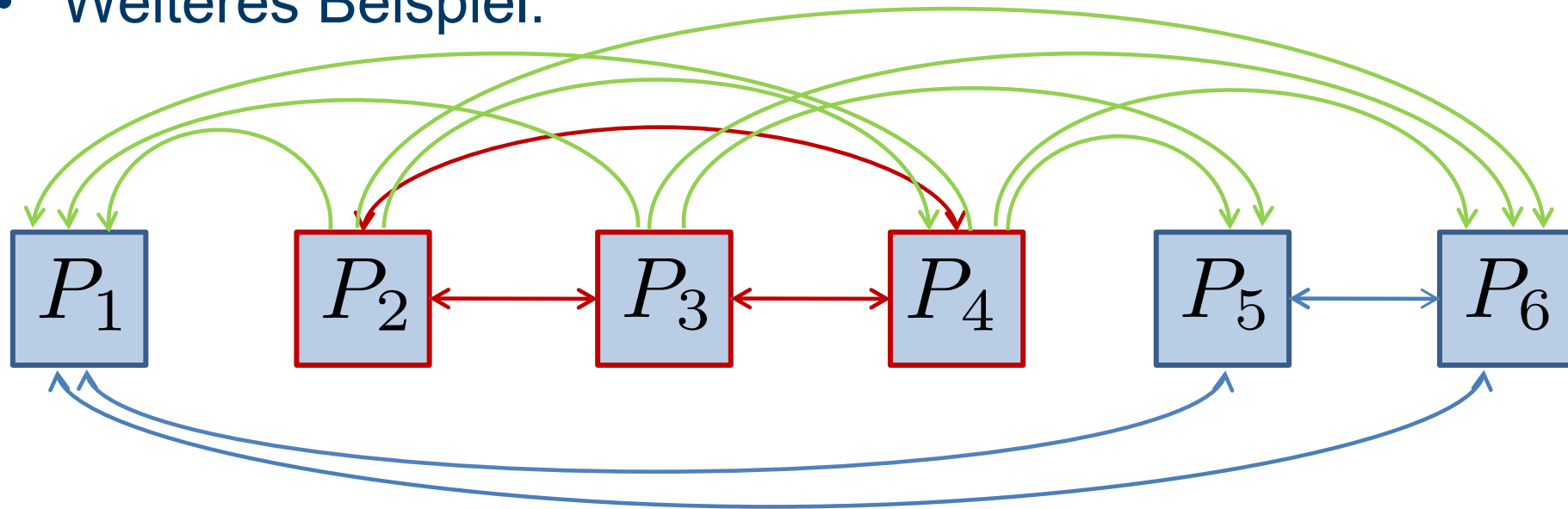


Eine sog. Blockmatrix

- Man sieht, dass der Graph nicht zusammenhängend ist
- Das bedeutet: Das Gleichungssystem von M lässt sich in zwei unabhängige Teilprobleme unterteilen!
 - Diese Erkenntnis kann enorm wichtig sein!
- zB für LR-Zerlegung: Statt $\frac{2}{3}n^3$ nur $2 \cdot \frac{2}{3} \left(\frac{n}{2}\right)^3 = \frac{1}{4} \cdot \frac{2}{3}n^3$



- Weiteres Beispiel:



man die Knoten wie oben umsortiert, d.h. Spalten und Zeilen permutieren und erhält folgende Blockmatrix

$$M = \begin{pmatrix} * & 0 & 0 & 0 & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & 0 & 0 & 0 & * & * \\ * & 0 & 0 & 0 & * & * \end{pmatrix} \quad \longrightarrow \quad \hat{M} = \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \end{pmatrix}$$

- Man kann Matrizen in einzelne Blöcke zerlegen

Dies vereinfacht z.B. die Schreibweise, vor allem wenn gleiche Blöcke mehrfach vorkommen.

Einfacher:

$$M = \begin{pmatrix} \begin{matrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{matrix} & & \\ & \begin{matrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{matrix} & \\ & & \ddots \end{pmatrix} \quad M = \begin{pmatrix} B & & \\ & B & \\ & & \ddots \end{pmatrix} \quad \text{mit}$$

$$B = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{pmatrix}$$

- Beispiel: Wärmeverteilung auf Platte (Poissonsgleichung)

$$= \begin{bmatrix} C & -Id & & \\ -Id & C & \ddots & \\ & \ddots & \ddots & -Id \\ & & -Id & C \end{bmatrix}$$

- Tridiagonale Blockmatrix
- tridiagonale Einträgen

- Mit Blockmatrizen (gleicher Struktur) kann ganz normal gerechnet werden:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1m} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{m1} & \cdots & \mathbf{A}_{mm} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \cdots & \mathbf{B}_{1m} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{m1} & \cdots & \mathbf{B}_{mm} \end{bmatrix}$$

wobei \mathbf{A}_{ij} bzw. \mathbf{B}_{ij} selbst Matrizen $\in \mathbb{R}^{k \times k}$ mit $n=m \cdot k$ sind

- Addition/Subtraktion: Addieren/subtrahieren der entsprechenden Einträge
- Multiplikation: (Matrizen-)Multiplikation der Einträge und aufsummieren
- Inverse ?

- Beispiel: Matrix-Vektor-Multiplikation:

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array} \right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right)$$

$$\vec{b} = \left(\begin{array}{c} b_1 \\ b_2 \\ \hline b_3 \\ b_4 \end{array} \right) = \left(\begin{array}{c} B_1 \\ B_2 \end{array} \right)$$

$$A \cdot \vec{b} = \left(\begin{array}{c} A_{11}B_1 + A_{12}B_2 \\ A_{21}B_1 + A_{22}B_2 \end{array} \right)$$

Umfassende Literatur:

The Matrix Cookbook

http://www.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf

- Beispiel: Gleichungssystem lösen $\mathbf{A}\mathbf{x} = \mathbf{b}$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

- Löse mittels Rückwärtseinsetzen:

- zunächst $\mathbf{A}_{22}\mathbf{X}_2 = \mathbf{B}_2$
- und dann $\mathbf{A}_{11}\mathbf{X}_1 = \mathbf{B}_1 - \mathbf{A}_{12}\mathbf{X}_2$

Das Splitting in Blockmatrizen kann wichtig sein

- Falls ein Problem nicht komplett in den Speicher passt, muss das Problem in Teilschritten bearbeitet werden
 - Man passt die Blockgröße dem verfügbaren Speicher an
 - Analog: Optimierung für CacheArchitekturen
- Zur effizienten Bereitstellung der Daten
 - Man passt die Blockgröße dem gegebenen Cache an
 - Erlaubt das Nutzen von Pipelining
 - Dies ist besonders bei GPGPU (General-purpose computing on graphics processing units) entscheidend
 - Ergibt ggf. bis zu mehrere Größenordnungen an Performancegewinn

- Matrix-Matrix-Multiplikation (nach V. Strassen 1970):

Berechne $C = A \cdot B$, wobei A, B, C 2×2 Block-Matrizen sind

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad A_{ij} \text{ und } B_{ij} \text{ sind } n \times n \text{ Matrizen}$$

- Zunächst berechnen wir sieben Hilfsmatrizen

$$H_1 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$H_2 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$H_3 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$H_4 = (A_{11} + A_{12})B_{22}$$

$$H_5 = A_{11}(B_{12} - B_{22})$$

$$H_6 = A_{22}(B_{21} - B_{11})$$

$$H_7 = (A_{21} + A_{22})B_{11}$$

- Matrix-Matrix-Multiplikation:
- C ergibt sich dann aus den Hilfsmatrizen wie folgt:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} H_1 + H_2 - H_4 + H_6 & H_4 + H_5 \\ H_6 + H_7 & H_2 - H_3 + H_5 - H_7 \end{bmatrix},$$

- Man benötigt
 - 7 Matrixmultiplikationen
 - 18 Matrixadditionen
- $$H_1 = (A_{12} - A_{22})(B_{21} + B_{22})$$
- $$H_2 = (A_{11} + A_{22})(B_{11} + B_{22})$$
- $$H_3 = (A_{11} - A_{21})(B_{11} + B_{12})$$
- $$H_4 = (A_{11} + A_{12})B_{22}$$
- $$H_5 = A_{11}(B_{12} - B_{22})$$
- $$H_6 = A_{22}(B_{21} - B_{11})$$
- $$H_7 = (A_{21} + A_{22})B_{11}$$

Matrix-Matrix-Multiplikation:

- Auf die 7 Matrixmultiplikationen können wir rekursiv wieder das gleiche Partitionierungsschema anwenden
- Pro Rekursionsschritt reduziert sich die Matrixdimension bei der Multiplikation
- Seien M_n, A_n Kosten der Multiplikation bzw. Addition.

Dann gilt:

$$M_{2n} = 7 M_n + 18 A_n = 7 M_n + 18 n^2$$

Die Multiplikation einer $n \times n$ Matrix benötigt $m = \log_2 n$ Rekursionsschritte. Falls $n = 2^m$ erhält man für die Kosten $F_m = M_n$:

$$F_{m+1} = 7F_m + 18 \cdot (2^m)^2 = 7(7F_{m-1} + 18 \cdot (2^{m-1})^2) + 18 \cdot 4^m = \dots$$

$$= 7^m F_1 + 18 \cdot \sum_{j=0}^{m-1} 7^j \cdot 4^{m-j} = O(7^m)$$

und da $m = \log_2 n$:

$$M_n = F_m = O(7^m) = O(7^{\log_2 n}) = O(n^{\log_2 7}) \approx O(n^{2,807})$$

Matrix-Matrix-Multiplikation:

- Der rekursive Blockmatrix Algorithmus ist schneller als die naive Matrix-Matrix-Multiplikation !

$$O(n^{\log_2 7}) \approx O(n^{2.807})$$

vs. $O(n^3)$

n	1.000	10.000	100.000	10 ⁶	10 ⁹
n ³	10 ⁹	10 ¹²	10 ¹⁵	10 ¹⁸	10 ²⁷
n ^{2.807}	2,64·10 ⁸	1.70·10 ¹¹	1,08·10 ¹⁴	6,98·10 ¹⁶	18,4· 10 ²⁵
Faktor	3,78	5,90	9,19	14,32	54,18
Co-Wi	74,5	313,3	1.318	5.546	4.13·10 ⁵

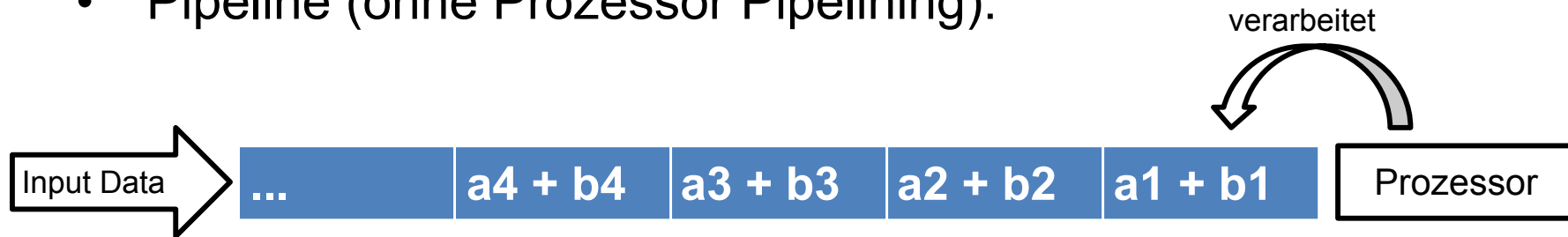
- Der Algorithmus ist auch bekannt als **Strassen-Multiplikation**
- Die numerische Stabilität ist etwas schlechter als die der naiven Variante
 - siehe: N. Higham: *Accuracy and Stability of Numerical Algorithms*
- Es geht sogar noch schneller: **Coppersmith–Winograd**: $O(n^{2.376})$
 - siehe: Coppersmith: *Matrix Multiplication via Arithmetic Progression*
- Besser als $O(n^2)$ kann es nicht gehen! Warum?

- Eine Operation ist gut parallelisierbar wenn
 - sie in mehrere Teiloperationen zerlegt werden kann
 - ohne, dass Nebenläufigkeiten oder Abhängigkeiten entstehen
- Beispiel: Vektoraddition

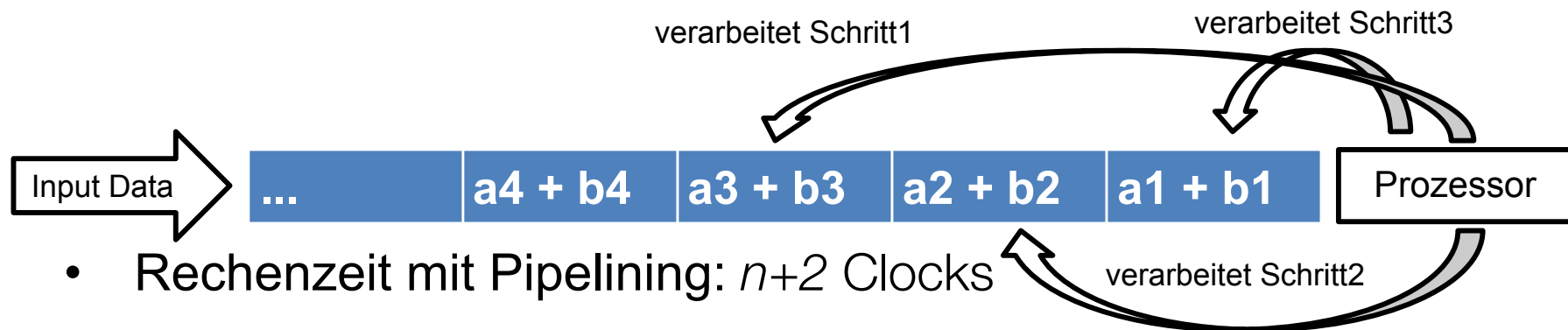
$$a_i + b_i \quad \text{for} \quad i = 1 \dots n$$

- Hier kann jeder Teilschritt kann einzeln
- und voneinander unabhängig abgearbeitet werden!

- Beispiel: Vektoraddition $a_i + b_i$ for $i = 1..n$
 - Pipeline (ohne Prozessor Pipelining):

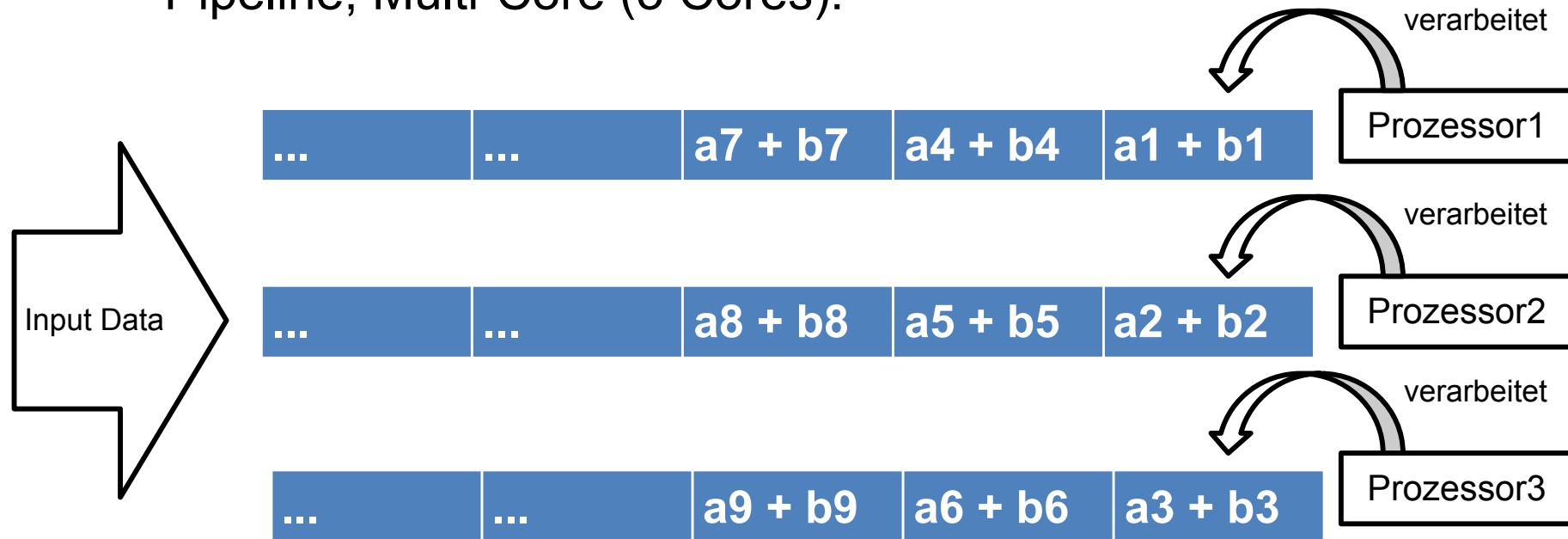


- Annahme: Prozessor braucht 3 Taktzyklen für eine Addition
- Rechenzeit: $3n$ Clocks!
- Pipeline mit Prozessor-Pipelining (im Taktzyklus 3):



- Rechenzeit mit Pipelining: $n+2$ Clocks

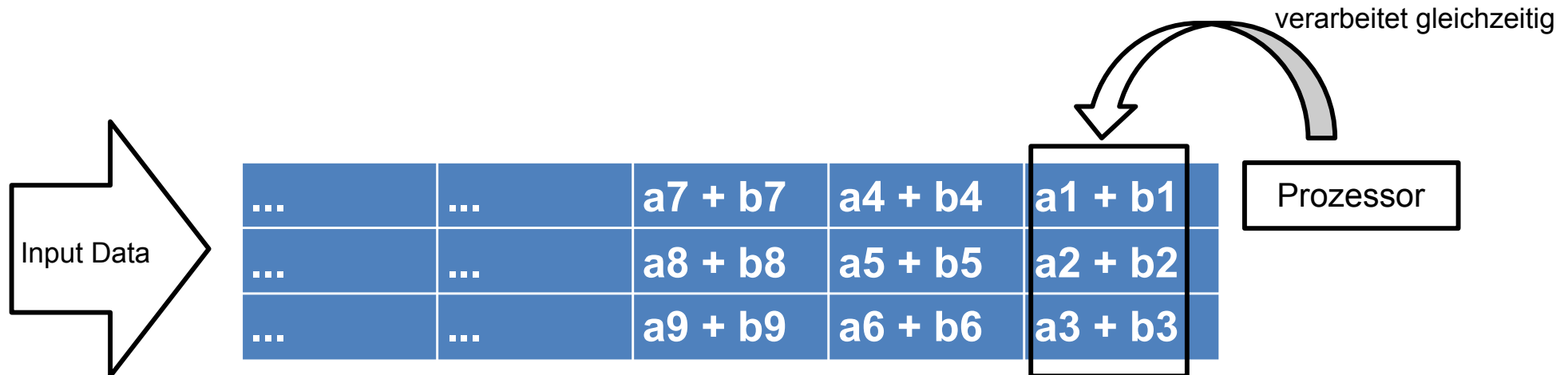
- Beispiel: Vektoraddition $a_i + b_i$ for $i = 1..n$
 - Pipeline, Multi-Core (3 Cores):



- Arbeit wird nun durch Anzahl der Prozessoren geteilt.
- Jeder einzelne Prozessor kann wiederum Prozessor-Pipelining unterstützen

dann ergibt sich ein Aufwand von $(n+2)/(\#num_cores)$ Clocks

- Beispiel: Vektoraddition $a_i + b_i$ for $i = 1..n$
 - Pipeline, SIMD (Single-Instruction-Multiple-Data), SIMD-Breite 3



- Vektoreinheit führt eine Operation gleichzeitig an mehreren Daten durch
- Verschiedene Operationen in einem Schritt sind nicht möglich
zB. $a1 + b1$ und $a2 * b2$ müsste sequentialisiert werden.
- Bei gleicher Operation selbe Performance wie Multi-Core

- Multi-Cores und SIMD ist typisch für heutige Hardware!
 - Aktuelle Grafikkhardware (NVIDIA GTX Titan):
 - 15 Prozessoren (1 standardmäßig deaktiviert)
 - 192 Kerne / Prozessor
 - $14 \times 192 = 2688$ Kerne
 - Aktuelle CPUs haben ähnliches: SSE (Streaming SIMD Extension)
 - SSE4 hat 128 Bit breite Register (= SIMD-Breite 4 single precision)
 - AVX: 256 Bit
 - AVX-512 hat Vektorregister mit 512 Bit Breite (Knights Landing) = 16 Werte in single precision
- Zurück zu Blockmatrizen:
 - Blocksplitting erlaubt Pipelining und Parallelisierbarkeit bei Matrixoperationen
 - Fazit: Blockgröße immer an jeweilige Hardware anpassen, um optimale Performanz zu erzielen, sonst liegt Rechenleistung brach
 - Algorithmen können/müssen auch dahingehend optimiert werden!

- Beispiel: Diskretisierung der Poisson-Gleichung (des Laplace-Operators)
- Spezielle Matrix Strukturen
- Datenstrukturen für dünn besetzte Matrizen
 - CRS
 - BCRS
 - CCS
- Blockmatrizen, schnelle MM-Multiplikation (Strassen)
- Beschleunigung: Pipelining, Parallelisierung