

Universität Karlsruhe (TH)

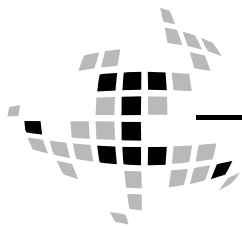
Forschungsuniversität · gegründet 1825

Parallele Algorithmen III

Prof. Dr. Walter F. Tichy

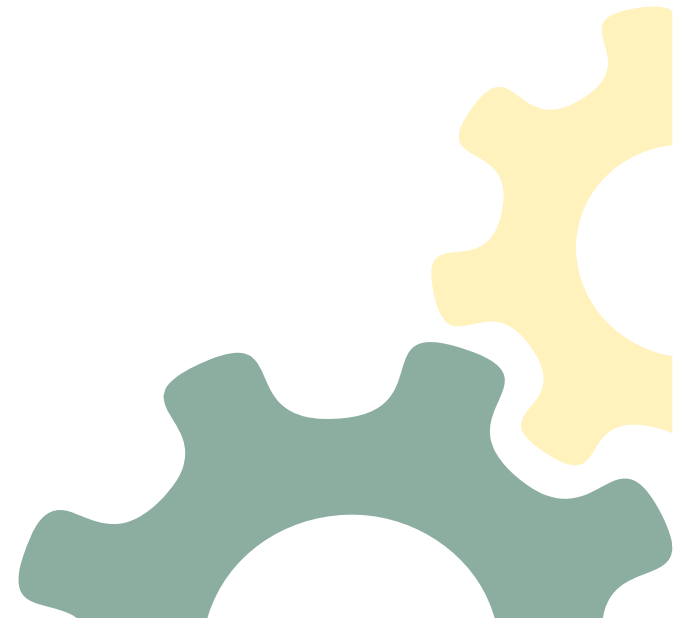
Dr. Victor Pankratius

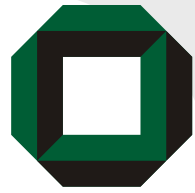
Ali Jannesari



Fakultät für **Informatik**

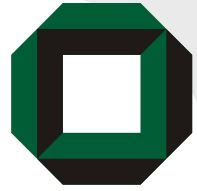
Lehrstuhl für Programmiersysteme





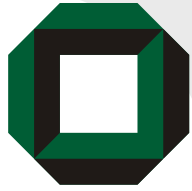
Minimaler Spannbaum (1)

- Auch: minimales Gerüst
- Sei $G=(V,E)$ ein ungerichteter, zusammenhängender Graph mit gewichteten Kanten.
- $T=(V,E')$ mit $E' \subseteq E$ heißt Spannbaum (Gerüst) von G , falls T ein Baum ist (d.h. ein Graph ohne Zyklen).

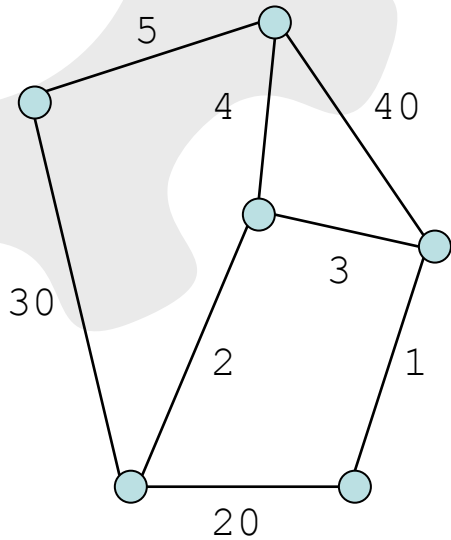


Minimaler Spannbaum (2)

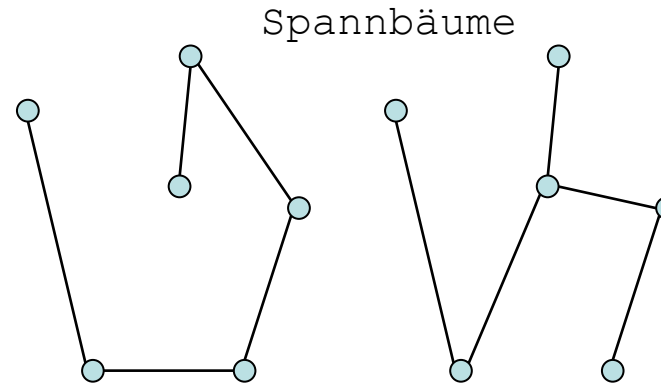
- Das Gewicht $w(T)$ des Spannbaums ist die Summe der Gewichte in E' .
- Ein minimales Gerüst oder ein minimaler Spannbaum ist ein Gerüst mit dem minimalen Gewicht.
- Wenn alle Kanten unterschiedliche Gewichte haben, dann ist der min. Spannbaum eindeutig.



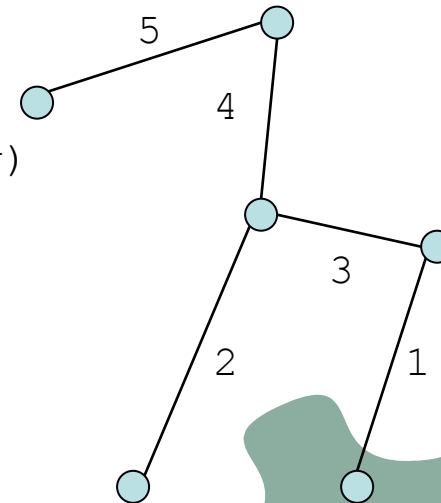
Min. Spannbaum, Beispiel



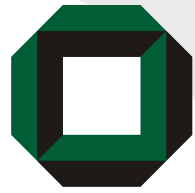
Ausgangsgraph



Min.
Spannbaum
(eindeutig)

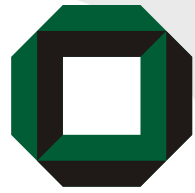


- Notation:
 - $w(v_i, v_j)$ sei das Gewicht der Kante (v_i, v_j) .
 - $w(v_i, v_j) = \infty$, wenn die Kante (v_i, v_j) nicht existiert.



MSB: Vorüberlegungen (1)

- Falls $V = \{v_0, v_1, \dots, v_{n-1}\}$, dann hat der MSB $n-1$ Kanten. Insgesamt kann es $n(n-1)/2$ Kanten geben. Daher ist $\Omega(n^2)$ eine untere Schranke für die Operationen, die zur Berechnung des MSB erforderlich sind (jede Kante muss wenigstens einmal inspiziert werden).

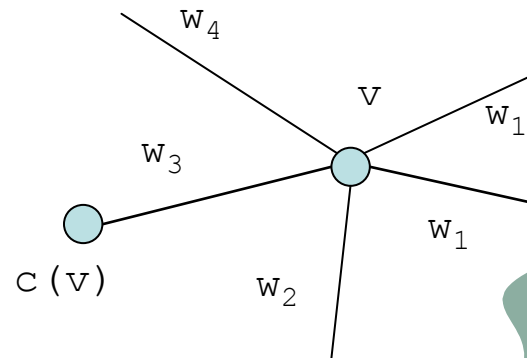


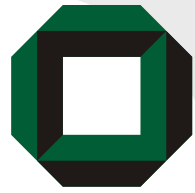
MSB: Vorüberlegungen (2)

- Lemma

- $G=(V,E)$ sei gewichteter, ungerichteter Graph.
- $\forall v \in V$ definiere $c(v) \in V$ so, dass $(v,c(v))$ die Kante mit minimalem Gewicht ist, die auf v auftrifft (beliebige Wahl bei gleichen Gewichten).
- Behauptung: alle Kanten $(v,c(v))$ gehören zum MSB.

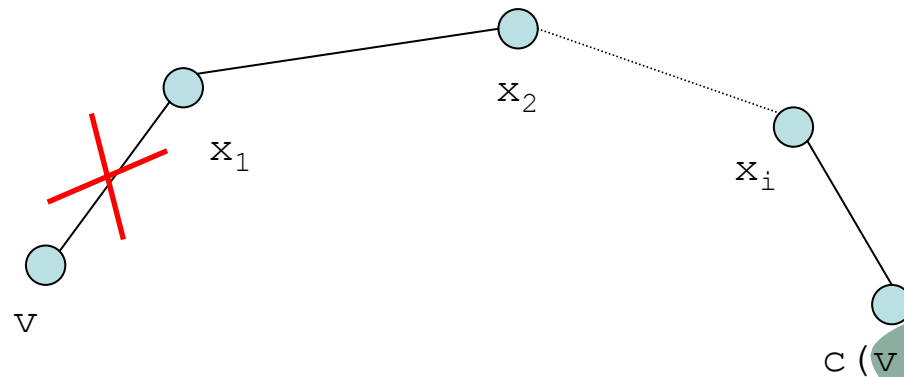
$$w_3 = \min \{ w_1 \dots w_4 \}$$

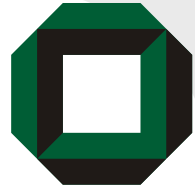




MSB: Vorüberlegungen (3)

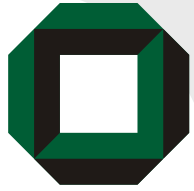
- Lemma (Forts.)
 - Beweis durch Widerspruch:
($v, c(v)$) gehöre nicht zum MSB.
Wenn man nun (v, x_1) entfernt und dafür ($v, c(v)$) einsetzt, erhält man einen MSB mit geringerem Gewicht \Leftarrow



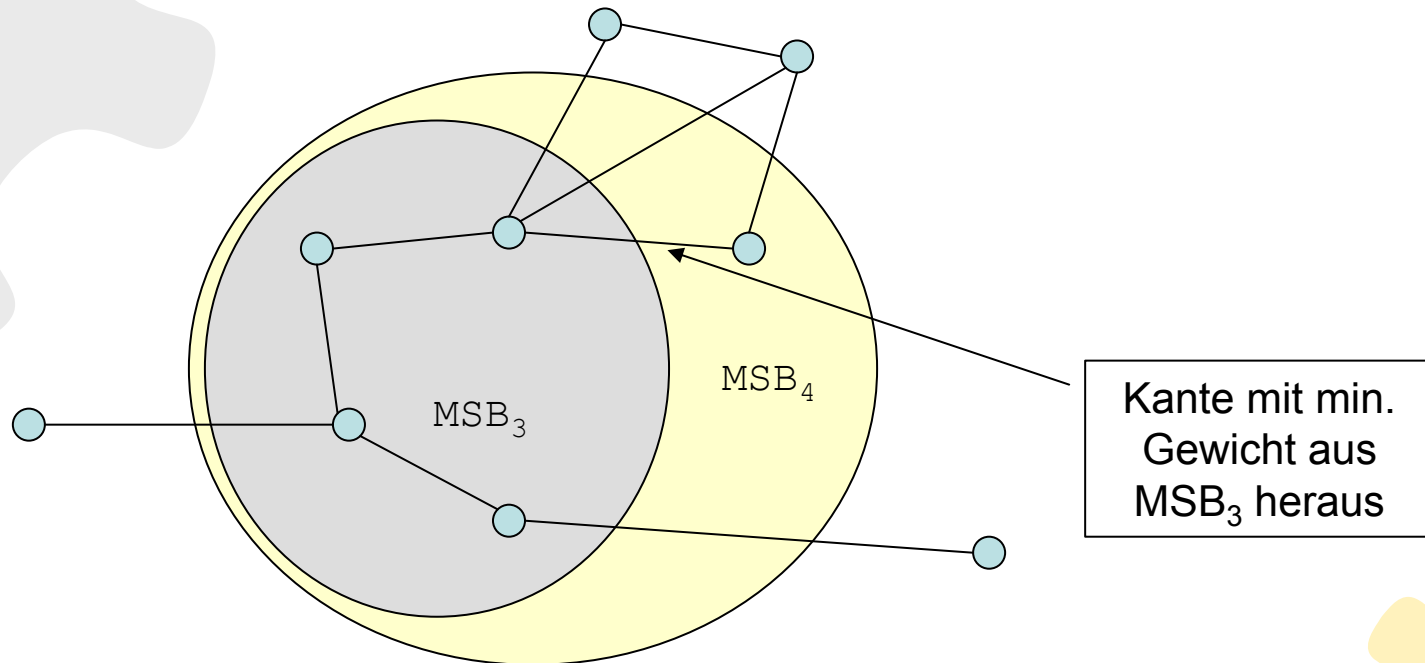


Prims sequentieller Alg. (1)

- Beginne mit einem bel. Knoten, um den Baum aufzubauen.
- In jedem Schritt
 - Finde die minimale Kante, die aus dem Baum herausführt und füge diese Kante und den zugeh. Knoten hinzu.
 - Korrigiere die Kanten, die aus dem Baum herausführen (da ein neuer Knoten hinzukam).

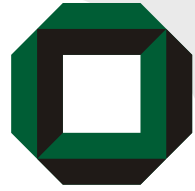


Prims sequentieller Alg. (2)



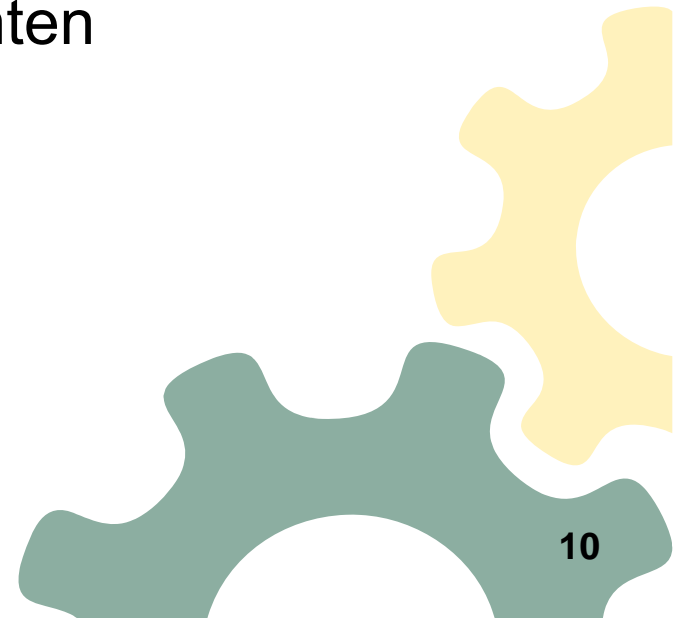
- Laufzeit: (k Knoten im Baum)

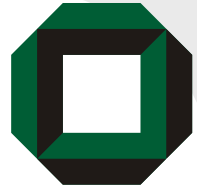
$$\sum_{k=1}^{n-1} (n - k) = O(n^2)$$



Prims Algorithmus (3)

- Idee zur Parallelisierung
 - Auffinden der minimalen, herausführenden Kante gleichzeitig
 - Hinzufügen von Knoten und Kante
 - Rundruf der hinzugefügten Kante
 - Korrektur der herausführenden Kanten

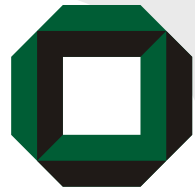




MSB: paralleler Algorithmus (1)

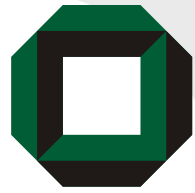
- Annahmen für den parallelen Algorithmus:
 - Gewichtsmatrix $W^{(n \times n)}$, wobei
$$W(j,i) = W(i,j) = \begin{cases} \text{Gewicht der Kante } (i,j) \\ \infty, \text{ falls Kante } (i,j) \text{ nicht ex.} \\ \infty, \text{ falls } i=j \end{cases}$$
 - N Prozessoren P_0, P_1, \dots, P_{N-1} mit $1 < N < n$
 - Wir schreiben $N = n^{1-x}$, mit $0 < x < 1$.
 - Anzahl Knoten, die jedem Prozessor zugeteilt werden:

$$\left\lceil \frac{n}{n^{1-x}} \right\rceil = \lceil n^x \rceil$$



MSB: paralleler Algorithmus (2)

- Die Knotenmenge des Prozessors i sei V_i .
- Zum Beispiel bekommt Prozessor i die Knoten $i \cdot n^x, \dots, (i+1) \cdot n^x - 1$ zugewiesen.
- Die Prozessoren halten den Vektor c aktuell.
 - Ein Eintrag pro Knoten; der Vektor c wird partitioniert wie die Knotenmenge
 - Falls Knoten v_k außerhalb des gerade im Aufbau befindlichen MSB, dann ist $c(v_k)$ der Knoten v_x innerhalb des MSB, mit $w(v_k, v_x)$ minimal.

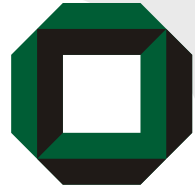


MSB: paralleler Algorithmus (3)

- Minimaler Spannbaum($[n][n]W$, $[n]$ Baum)
 - Berechnet min. Spannbaum aus Gewichtsmatrix $W^{(n \times n)}$.
 - Die Kanten des MSB erscheinen im Ausgabe-Parameter Baum.
- Schritt 1
 - (1.1) Markiere v_0 als Knoten, der bereits zum Baum gehört
 - (1.2) forall $i: 0..N-1$ in parallel
foreach $j \in V_j$ do
 $c(v_j) := v_0$
end foreach
end forall

$O(1)$

$O(n^x)$



MSB: paralleler Algorithmus (4)

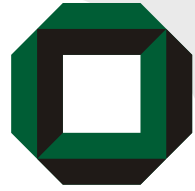
- Schritt 2: for $i: 1..n-1$ do // füge Kante ein
 - (2.1) forall $j: 0..N-1$ in parallel
 - (a) P_j bestimmt das Minimum $w(v_p, c(v_p))$, wobei v_p ein Knoten außerhalb des MSB ist, der P_j zugeordnet ist
 - (b) $w_j = w(r_j, t_j)$ sei das Minimum aus (a)
 P_j gibt das Tripel (w_j, r_j, t_j) zurückend forall
 - (2.2) Bestimme das Minimum der w_j für $0 \leq j < N-1$ mit Minimumreduktion; das Ergebnis sei (w_s, r_s, t_s) .
(r_s ist außerhalb, t_s innerhalb des Baumes)
 - (2.3) P_0 fügt die Kanten (r_s, t_s) dem Baum hinzu

$O(n^x)$

$O(1)$

$O(\log N)$

$O(1)$



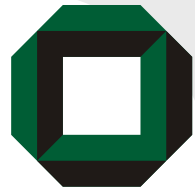
MSB: paralleler Algorithmus (5)

- Schritt 2: for $i: 1..n-1$ do (Fortsetzung)
 - (2.4) r_s wird an alle Prozessoren verteilt
 - (2.5) forall $j: 0..N-1$ in parallel
 - (a) if r_s in V_j then
 - P_j markiert r_s als zum Baum gehörig
 - end if
 - (b) foreach $v_p \in V_j$
 - if v_p nicht im Baum then
 - if $w(v_p, r_s) < w(v_p, c(v_p))$ then
 - $c(v_p) := r_s$
 - endif
 - endif
 - end foreach
 - end forall

$O(\log N)$

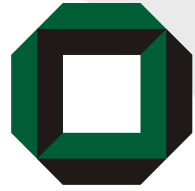
$O(1)$

$O(n^x)$



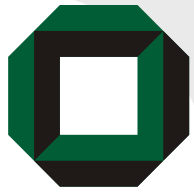
MSB: paralleler Algorithmus (6)

- Datenverteilung
 - Jeder Prozessor speichert $n \times$ **Zeilen** der Gewichtsmatrix W .
 - Zu berechnen:
 - in Schritt (2.1a): $w(i, c(i))$: lokal
 - in Schritt (2.5b): $w(i, r_s)$: lokal; $w(i, c(i))$: lokal
- ⇒Keine zusätzliche Kommunikation nötig.

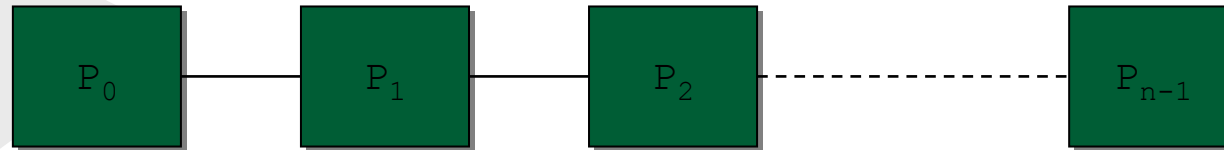


MSB: paralleler Algorithmus (7)

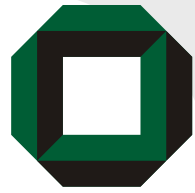
- Zeitaufwand
 - Iterationszeit: $T_{it} = O(n^x)$
 - Insgesamt: $T = O(n^{1+x})$
- Kosten
 - Prozessoren * Laufzeit
 - $K = n^{1-x} \cdot O(n^{1+x}) = O(n^2)$
 - Algorithmus ist kostenoptimal und adaptiv.
 - $n^x > \log n$ gilt nur für große n , Optimalität ist nur für $N = n^{1-x} = n/n^x < n/\log n$ gegeben.



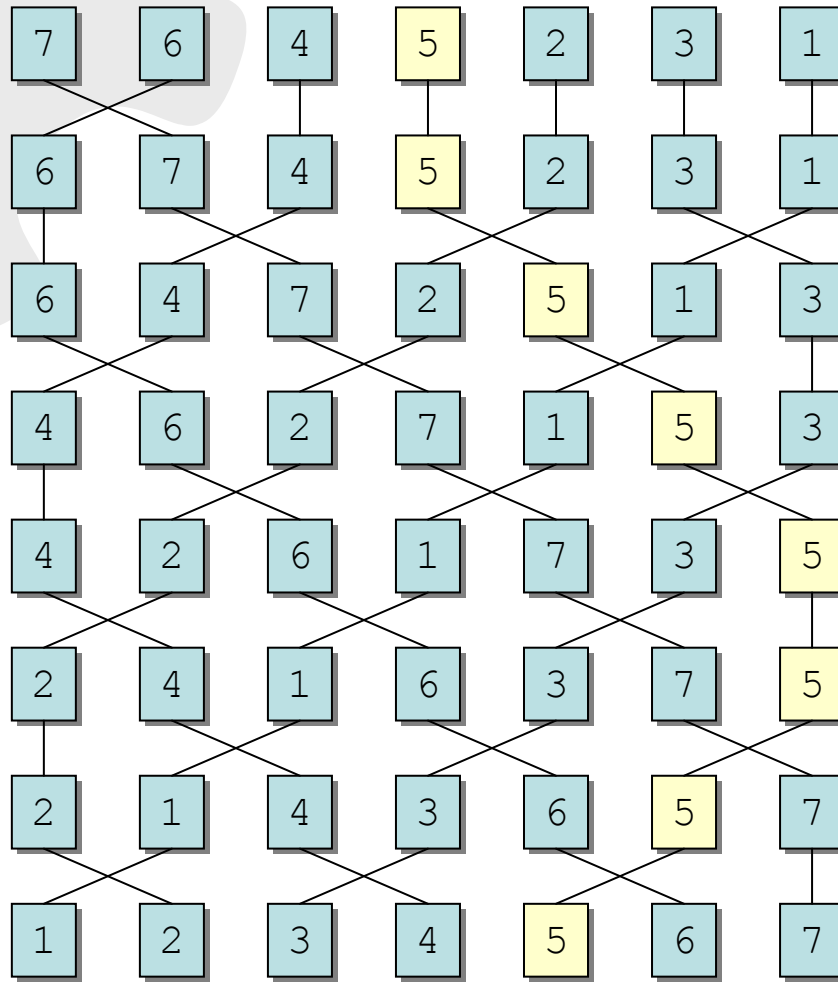
Odd-Even Transposition Sort (1)



- Annahmen
 - n Prozessoren in linearer Verbindung
 - Benachbarte Prozessoren können Elemente austauschen; jeder Prozessor kann aber nur mit einem Nachbar gleichzeitig zu tauschen
- Idee
 - Erst tauschen Prozessoren mit gradzahligem Index, dann die mit ungradzahligem.

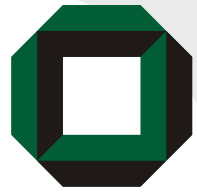


Odd-Even Transposition Sort (2)



- Beispiel: Sortiere 7,6,4,5,2,3,1

- Ein Element bleibt nicht notwendigerweise auf seiner endgültigen Position stehen, wenn es sie das erste Mal erreicht.

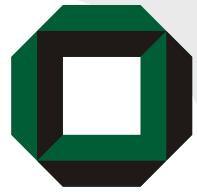


Odd-Even Transposition Sort (3)

- Algorithmus ($A[0..n-1]$ bereits verteilt)

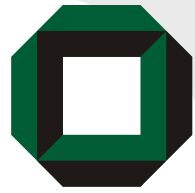
```
for k=1 to ceil(n/2) do
  forall i in 0..n-2 in sync do
    if even(i) and  $A[i] > A[i+1]$  then
       $A[i] := A[i+1]$ 
    else if odd(i) and  $A[i] > A[i+1]$  then
       $A[i] := A[i+1]$ 
    endif
  end forall
end for
```

- Laufzeit: $O(n)$; Prozessoren: n ; Kosten: $O(n^2)$.
- Optimal für lineare Verknüpfung.



Odd-Even Transposition Sort (4)

- Satz: Odd-Even Transp. Sort sortiert eine n-elementige Folge in höchstens n Schritten
- Beweis (Induktion)
 - Für n: 1..3 durch Fallunterscheidung zu zeigen.
 - Induktionsannahme: Alg. sortiere in m Schritten eine m-elementige Folge
 $S_{m-1} = \{x_0, x_1, \dots, x_{m-1}\}$.



Odd-Even Transposition Sort (5)

- Beweis (Fortsetzung)

- Induktionsschritt:

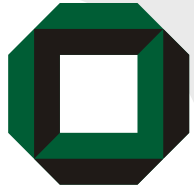
- Betrachte $S_m = \{x_0, x_1, \dots, x_m\}$ und M , das Maximum von S_m (bei Duplikaten das rechteste Maximum von S_m).

- Verfolge M im Sortiernetz.

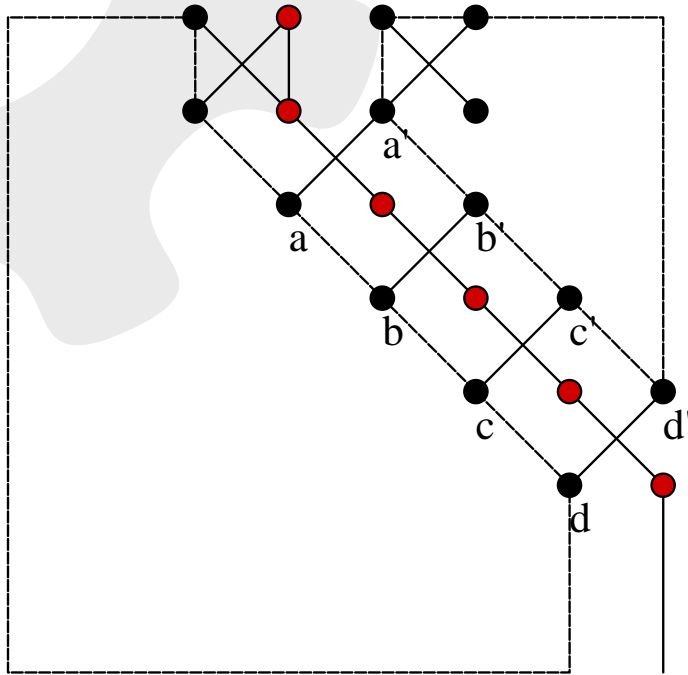
- Entferne M und „nähe“ Teile des Netzes zusammen.

⇒ Sortierung von m Elementen erfolgt in m Schritten; M ist an der richtigen Stelle.

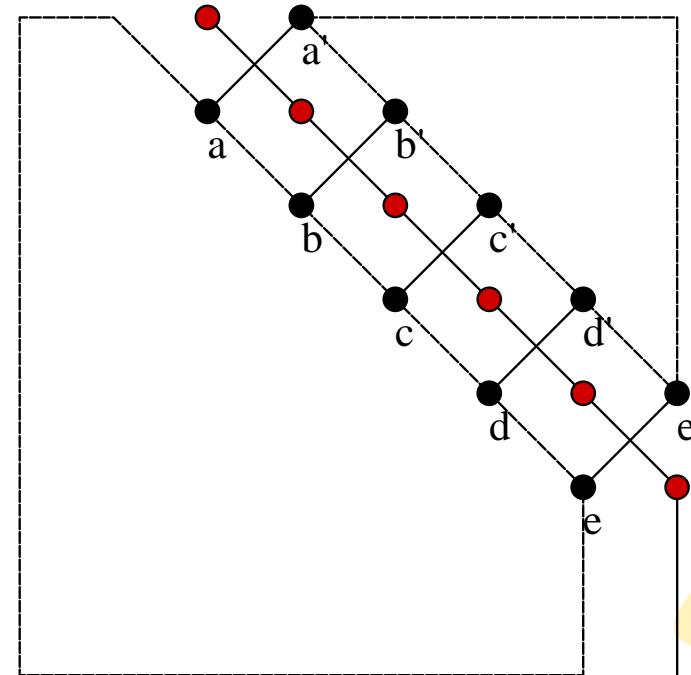
⇒ QED.



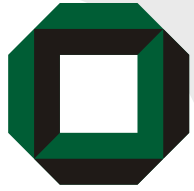
Odd-Even Transposition Sort (6)



Weg des größten Elementes (rot), bei ungerader Startposition



Weg des größten Elementes (rot), bei gerader Startposition



Odd-Even Transposition Sort (7)

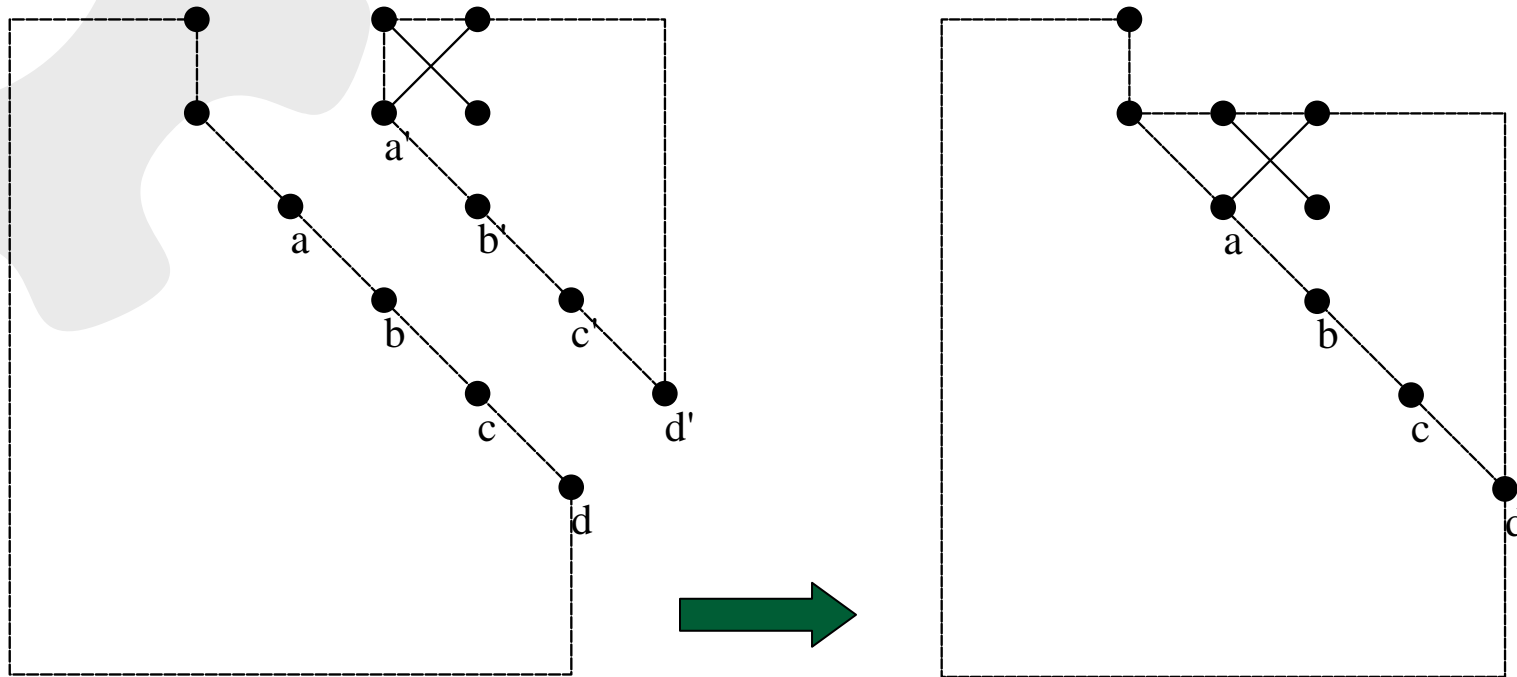
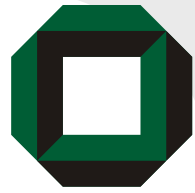
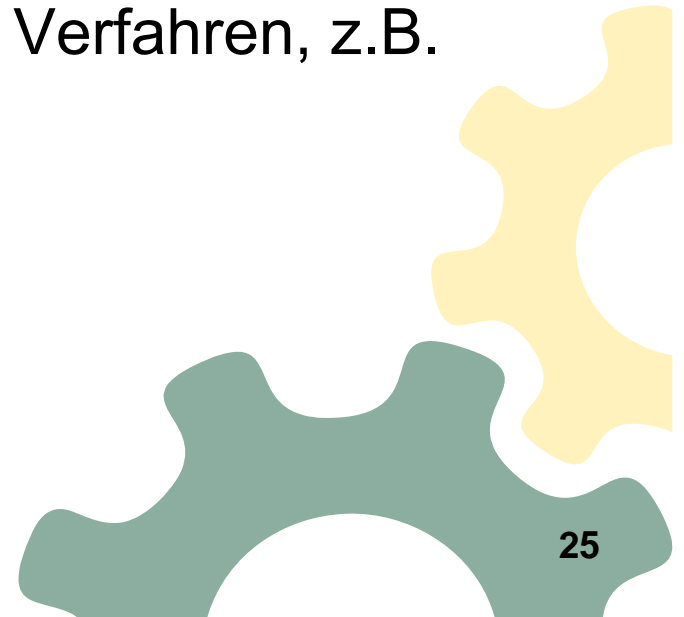


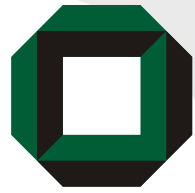
Diagramme zum Induktionsschritt:
Das Maximum wird entfernt und die beiden Teile des Sortiernetzes zusammengefügt.



Erweiterung für $p < n$ (1)

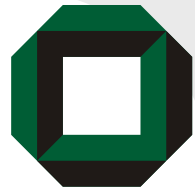
- Jeder Prozessor erhält $k = \text{ceil}(n/p)$ Elemente, der letzte Prozessor bekommt evtl. einige Dummy-Elemente.
- Vorverarbeitung:
 - Jeder Prozessor sortiert sein Segment von k Elementen mit einem sequentiellen Verfahren, z.B. Haldensortierung: $T(k) = O(k \log k)$.





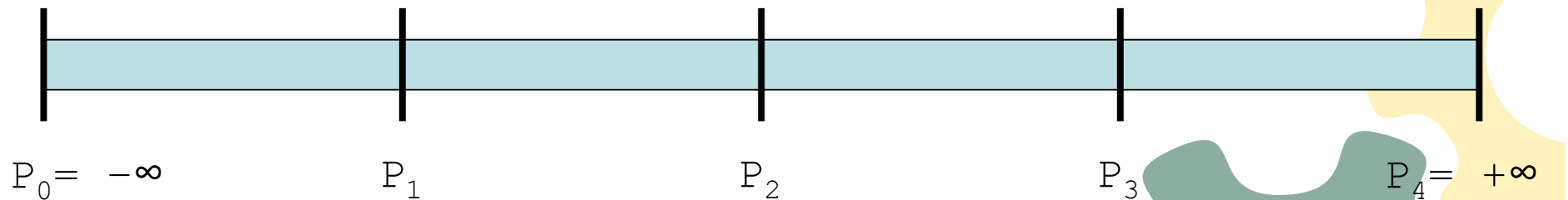
Erweiterung für $p < n$ (2)

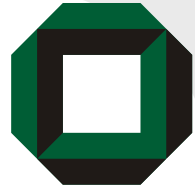
- Die zwei Vertauschungsschritte des Odd-Even-Transposition Sort werden durch je einen Verschmelze-Zerteile-Schritt ersetzt:
 - Prozessor $i+1$ sendet seine k Elemente an Prozessor i .
 - Dieser verschmelzt die beiden Folgen ($O(k)$) und sendet die größere Hälfte (sortiert) an Prozessor $i+1$.
- Zeitaufwand und Kosten:
 - $T_p(n) = O(n/p \cdot \log n/p) + O(p \cdot n/p) = O(n/p \cdot \log n) + O(n)$
 - $C_p(n) = p \cdot T_p(n) = O(n \cdot \log n) + O(np)$
(optimal für $p \leq \log n$)



Sortieren mit Stichproben (1)

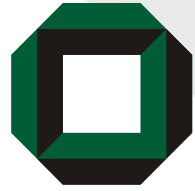
- Problem der Vertausche-Algorithmen
 - Die Daten werden zu oft bewegt.
 - Wie könnte man die Daten gleich an den richtigen Prozessor schicken?
- Angenommen,
 - wir haben N Prozessoren, $n > N$ Datenelemente zu sortieren.
 - wir hätten ferner $N+1$ „Angelpunkte“, die die Daten schön in N Segmente aufteilen.





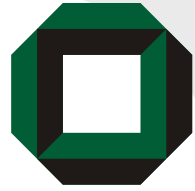
Sortieren mit Stichproben (2)

- Wenn bei 4 Prozessoren diese 5 Partitionierungspunkte im Voraus bekannt wären, dann könnte das Sortieren folgendermaßen von sich gehen:
 1. Alle Prozessoren erhalten alle Partitionierungspunkte.
 2. Die lokalen Daten werden in die Partitionierungen einsortiert.
 3. Die jeweilige Partitionierung wird an den Zielprozessor geschickt.
 4. Die Zielprozessoren nehmen „ihre“ Segmente entgegen und sortieren sie.



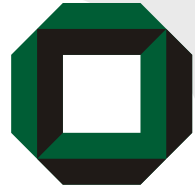
Sortieren mit Stichproben (3)

- Problem:
 - Die Angelpunkte so zu bestimmen, dass gleich große Partitionen herauskommen.
 - Wenn dies nicht gewährleistet ist, kann ein Prozessor fast alle Daten bekommen und eine Parallelisierung findet nicht statt.
- Idee:
 - Ziehe eine Stichprobe aus den Daten, sortiere sie. Die Stichprobe ist *größer* als die Anzahl der Partitionen.
 - Bestimme die Partitionen aus der Stichprobe.



Sortieren mit Stichproben (4)

- Satz:
 - Für alle $\varepsilon > 0$ gibt es ein $m \in O(1/\varepsilon^2 \cdot \log P)$, so dass im Mittel und mit hoher Wahrscheinlichkeit kein Prozessor mehr als $(1+\varepsilon) \cdot n/N$ Datenelemente erhält.
 - Wenn also $n/N \gg \log P$, genügt eine kleine Stichprobe.
- Beispiel
 - Auf den beiden folgenden Folien ist ein Beispiel mit $N=3$, $P=3$, $n=27$, $m=9$ dargestellt.



Sortieren mit Stichproben (5)

Unsortierte
Eingangsdaten

19	7	12
1	9	13
25	4	2

6	30	17
13	10	11
16	27	22

3	20	14
18	5	16
15	21	8

Zufällige Stichprobe

7	13	25
---	----	----

6	17	10
---	----	----

20	18	21
----	----	----

sortiert und aufgeteilt

6	7	10
---	---	----

13	17	18
----	----	----

20	21	25
----	----	----

Broadcast der
Pivotelemente

$(P_0 = -\infty)$

$P_1 = 10$

$P_2 = 18$

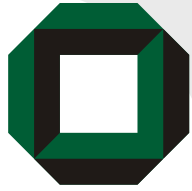
$(P_3 = +\infty)$

Elemente
klassifiziert

I_0	I_1	I_2
1	12	25
4	13	19
2		
7		
9		

I_0	I_1	I_2
6	17	30
10	13	27
	11	22
	16	

I_0	I_1	I_2
3	14	20
5	18	21
8	16	
	15	



Sortieren mit Stichproben (6)

Elemente
klassifiziert

I_0	I_1	I_2
1	12	25
4	13	19
2		
7		
9		

I_0	I_1	I_2
6	17	30
10	13	27
	11	22
	16	

I_0	I_1	I_2
3	14	20
5	18	21
8	16	
	15	

Umverteilung

I_0	I_1	I_2
1	2	3
4	5	6
7	8	9
10		

I_1	I_2	
11	12	13
13	14	15
16	16	17
18		

I_2		
19	20	21
22	25	27
30		

Lokal sortierte
Daten