Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-06-19

Kapitel

Sortieren

Quellen
Übliche, und:

+Comparison-Based In-Place Sorting with Cuda, GPU Computing Gems, Jade Edition

+GPU Gems 2 & 3

+Wikipedia

Allgemeines

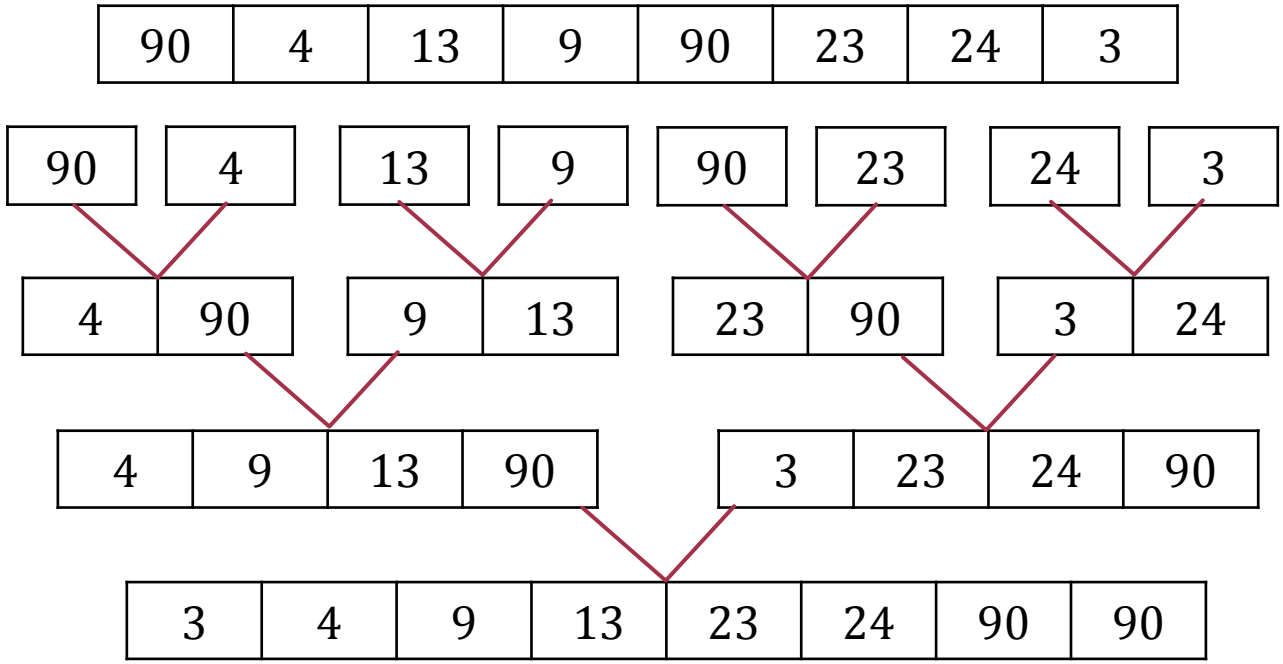
- Gegeben:
  - Folge a $= (a_0, a_1, \dots, a_{n-1})$ aus n Elementen
  - Jedem a_i sei Key x_i aus x $= (x_0, x_1, \dots, x_{n-1})$ zugeordnet
  - Für Keys ist eine Ordnung definierbar
  - Key und Element können identisch sein
  - Es kann mehrere Elemente mit gleichem Key geben
- Für sortierte Folge gilt:
  - $x_0 \leq x_1 \leq x_2 \leq \cdots \leq x_{n-1}$, oder:
  - $x_0 \geq x_1 \geq x_2 \geq \cdots \geq x_{n-1}$
- Sortieren: Überführe Folge durch permutieren der Elemente gemäß Keys in sortierte Folge
- Beispiel: Rendern mit Early-z-Test
  - a: n Raumschiffe (aus je max 5000 Vertices)
  - Key (x): z-Komponenten der Schwerpunkte der Raumschiffe
  - Sortiere gemäß x
  - Rendere Elemente aus a von vorne nach hinten

Algorithmus

Merge Sort

Divide-And-Conquer Prinzip

- Zerteile Problem so lange, bis einfach lösbar
- Hier: Ein elementige Folgen (sind bereits sortiert)
- Kombiniere dann Teillösungen rekursiv zu Gesamtlösung
- Hier: Vereinige je 2 sortierte Listen zu einer Sortierten (Merge)
- Parallel:
- Merge-Ops einer Rekursionsstufe
- Merge Op selbst?

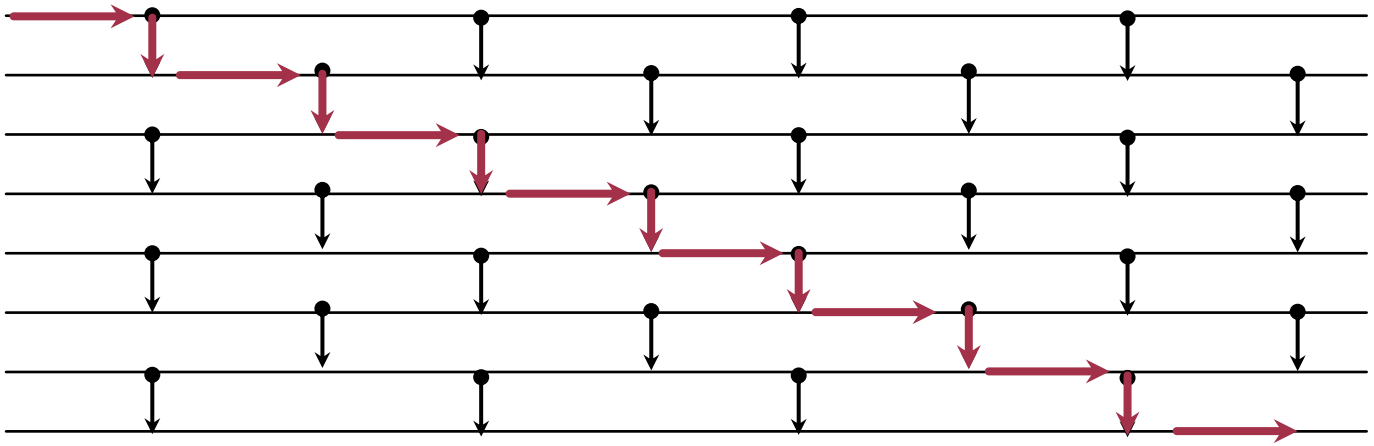


Algorithmus

Bitonic (Merge) Sort

Sortiernetz

- Datenunabhängiges Sortierverfahren
  - Verarbeitungsreihenfolge nicht von behandelten Daten abhängig
  - Best Case = Average Case = Worst Case
  - Gut parallelisierbar: Festes Datenladeschema & vorher bekannte Synchronisationspunkte
  - Kann durch Sortiernetz beschrieben werden
- Daten von links nach rechts entlang der Linien
- Datum trifft auf Pfeil:
  - Zeigt Pfeil auf größeren Wert?
  - Nein: Werte tauschen
  - Beispiel: Größter Wert oben
- Dieses Netz: Total Complexity $O(n^2)$

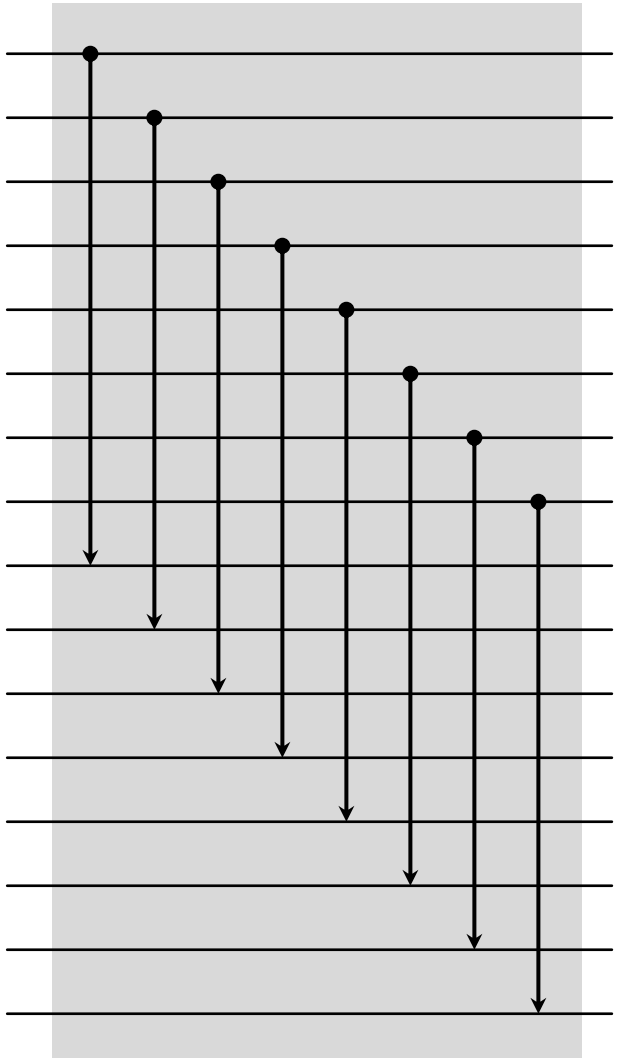


Bitonische Folge

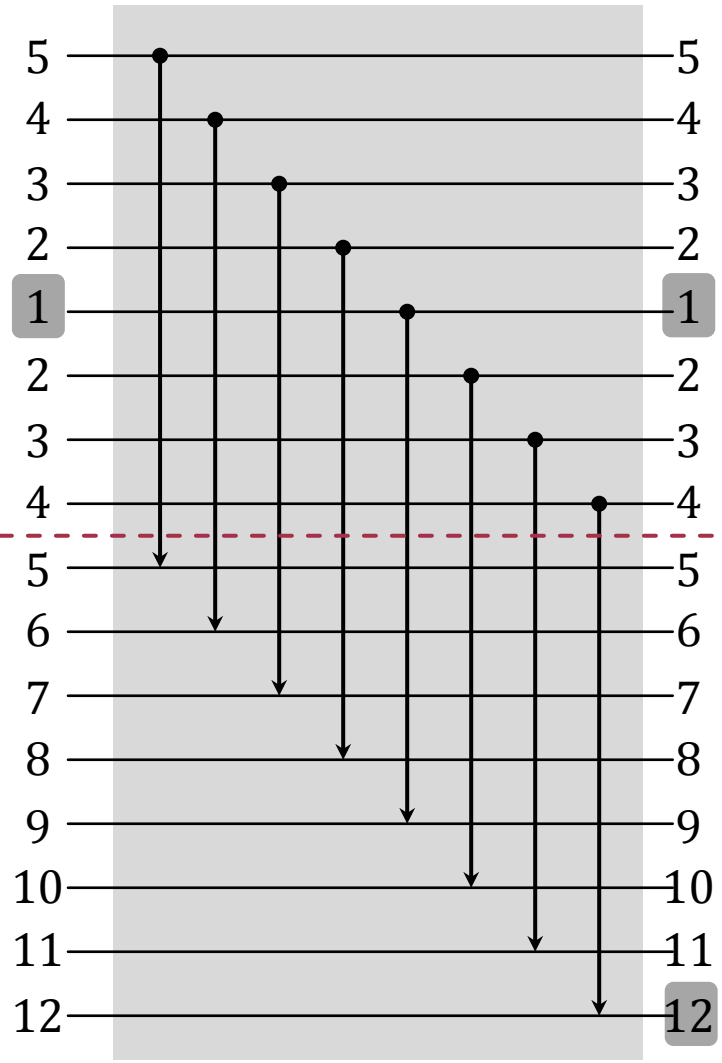
- Folge $x = (x_0, x_1, x_2, \dots, x_{n-1})$ aus n Zahlen ist bitonisch, falls eine der Bedingungen gilt:
  - $x_0 \leq x_1 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$, für ein k, mit: $0 \leq k < n$
  - $x_0 \geq x_1 \geq \cdots \geq x_k \leq \cdots \leq x_{n-1}$, für ein k, mit: $0 \leq k < n$
  - x kann in zwei Teile aufgeteilt werden, die vertauscht obige Bedingung erfüllen
- Beispiele (xk unterlegt):
  - (8, 6, 4, 2, 1, 3, 5, 7), oder umgestellt: (1, 3, 5, 7, 8, 6, 4, 2)
  - (3, 2, 1, 6, 8, 24, 15): vertausche (3, 2, 1) und (6, 8, 24, 15):
  - (6, 8, 24, 15, 3, 2, 1)
  - (1, 2, 3, 4, 5, 6, 7, 8)
  - (1, 2, 1, 2, 1, 2, 1, 2)

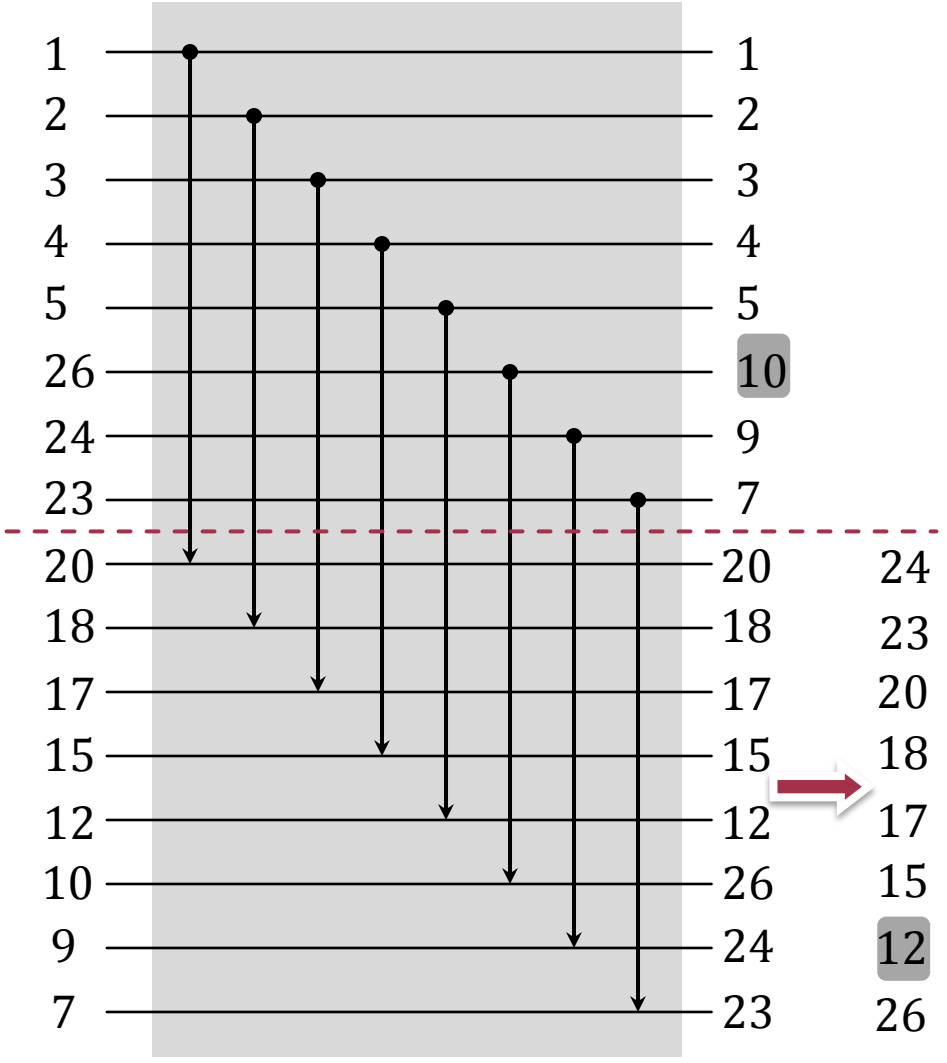
„InnerBox“

- Vergleicht jedes Element der oberen Hälfte mit entsprechendem der unteren Hälfte
- Falls Eingabe bitonische Folge
- … dann Ausgabe:
  - Obere Hälfte: Bitonische Folge
  - Untere Hälfte: Bitonische Folge
  - Alle Elemente der oberen Hälfte ≤ alle der Unteren



Beispiele

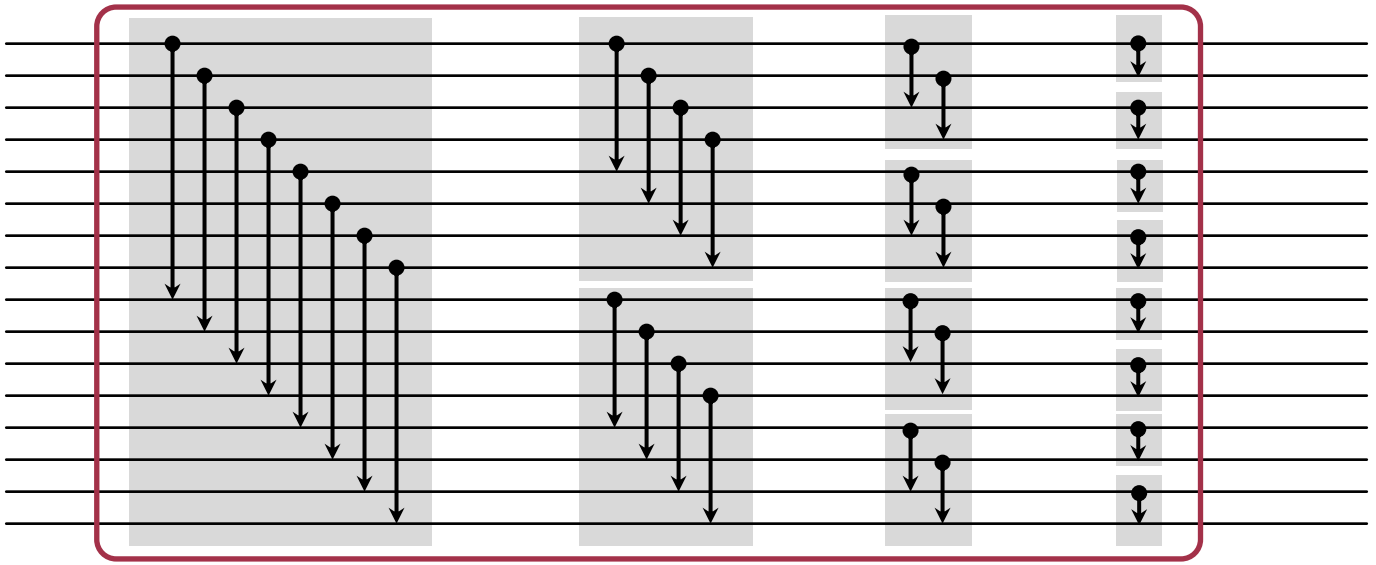




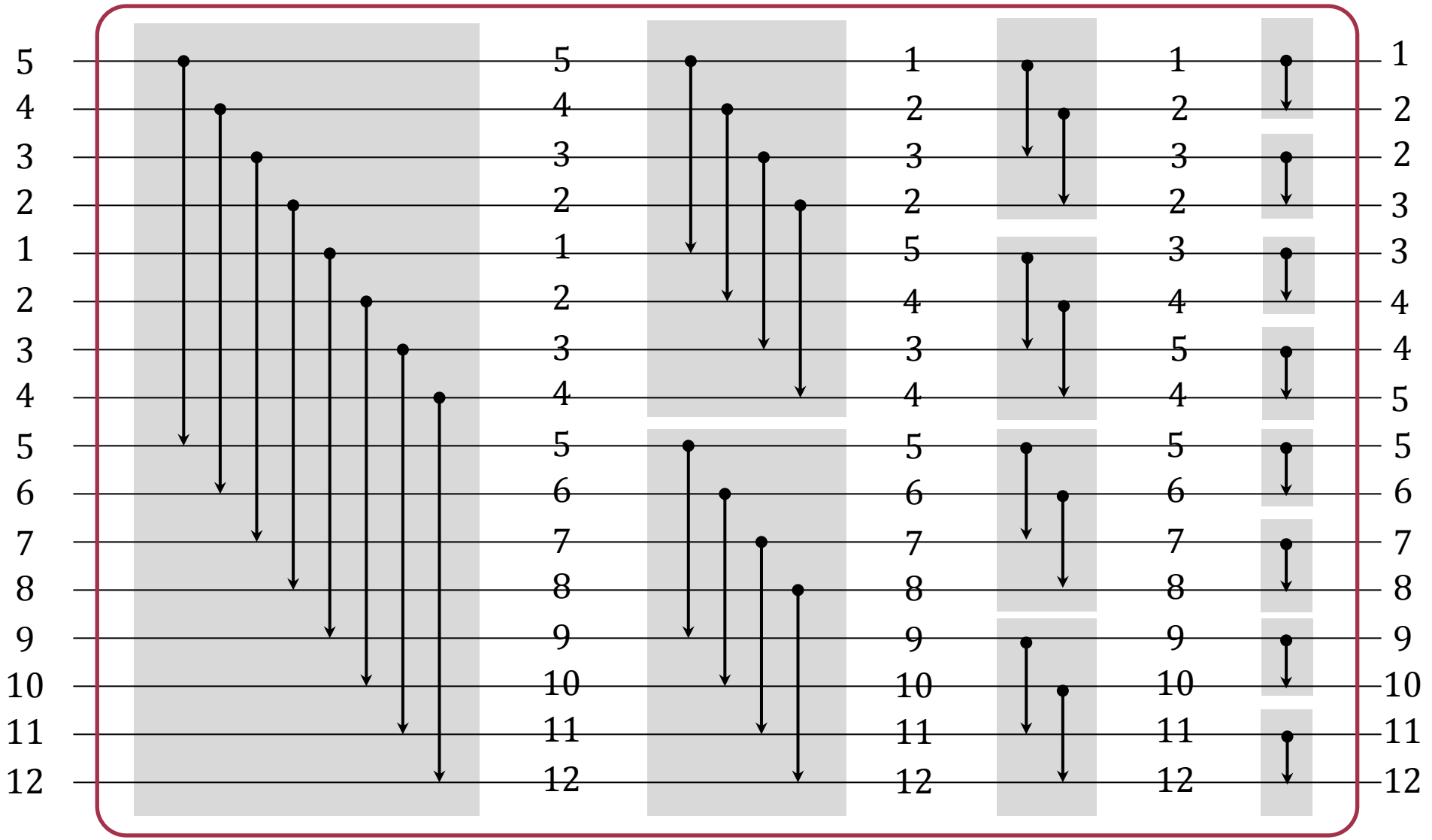
Umstellen. Nur zur Veranschaulichung, passiert nicht

„Outer Box“

- Eine InnerBox verarbeitet Eingangsfolge…
- … zwei InnerBoxen halber Größe die Ergebnishälften davon..
- … usw. bis zwei Elemente verglichen werden
- Sei Eingabe bitonische Folge. Dann befindet sich jedes Element
  - Nach Schritt 1
  - Nach Schritt 2
  - Nach Schritt 3
  - Nach Schritt 4
  - …
- OuterBox sortiert bitonische Folge aufsteigend

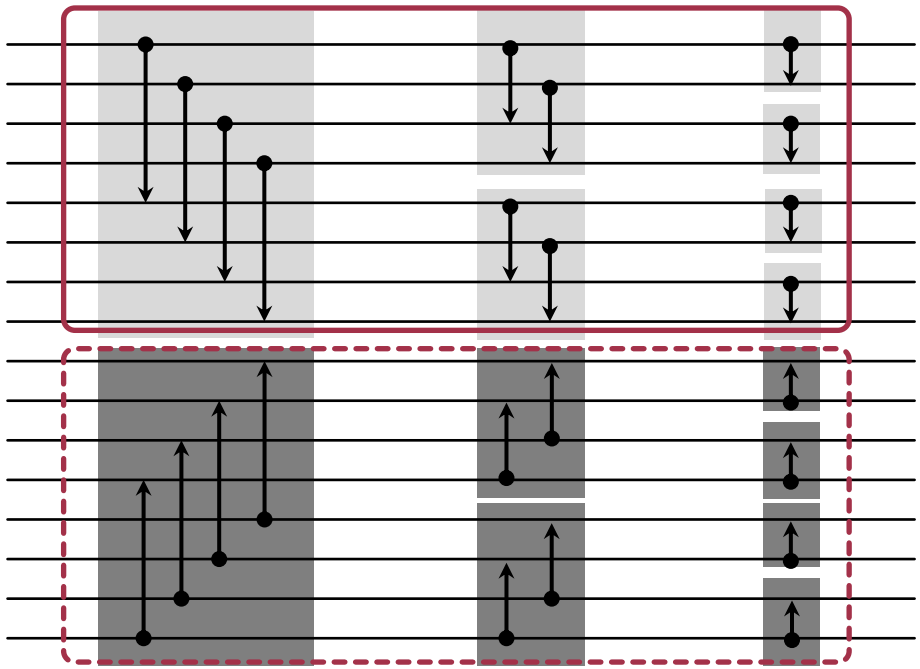


Beispiel


5 4 3 2 1 2 3 4 5 6 7 8 9 10 11 12
5 4 3 2 1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 2 5 4 3 4 5 6 7 8 9 10 11 12
1 2 3 2 3 4 5 4 5 6 7 8 9 10 11 12

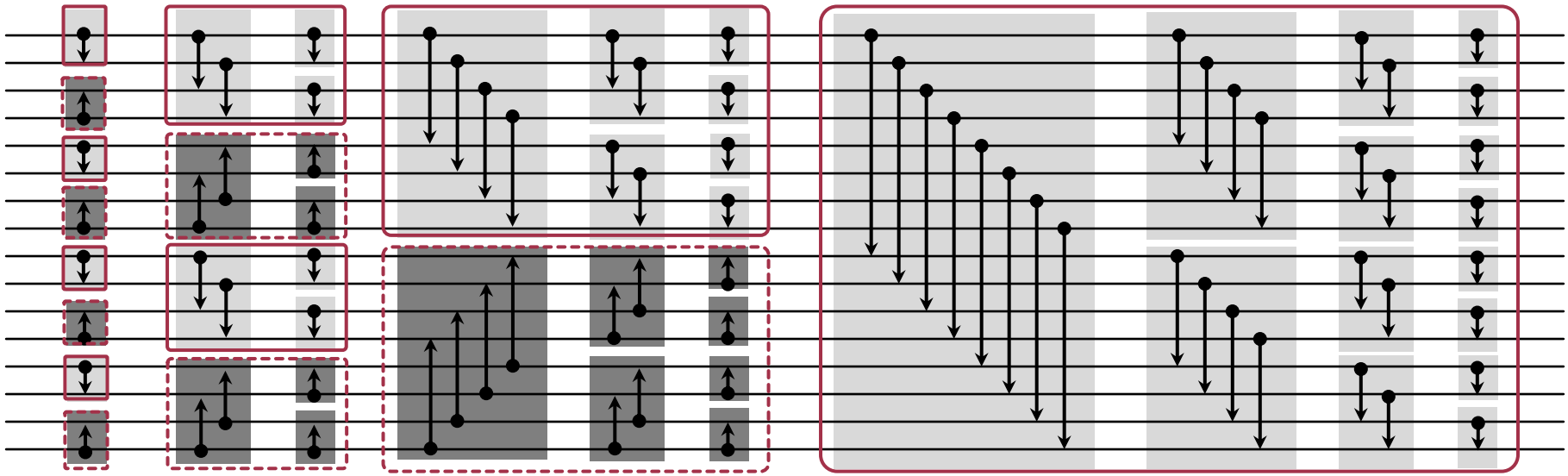
„-InnerBox“ / „-OuterBox“

- -InnerBox (dunkel unterlegt): Wie helle InnerBox, aber kleinere Elemente in unterer Ergebnishälfte
- -OuterBox (gestrichelt umrandet): Wie durchgezogen umrandete OuterBox, liefert aber absteigende Folge von Werten
- Bilden zusammen „Boxenpaar“



Sortiernetz für n=16 Keys

- Dieses Sortiernetz nennt man Butterfly-Network
- In:
  - n Sortierte Folgen aus je einem Element, bzw.
  - n/2 bitonische Folgen aus je zwei Elementen
- Out: Aufsteigend sortierte Folge

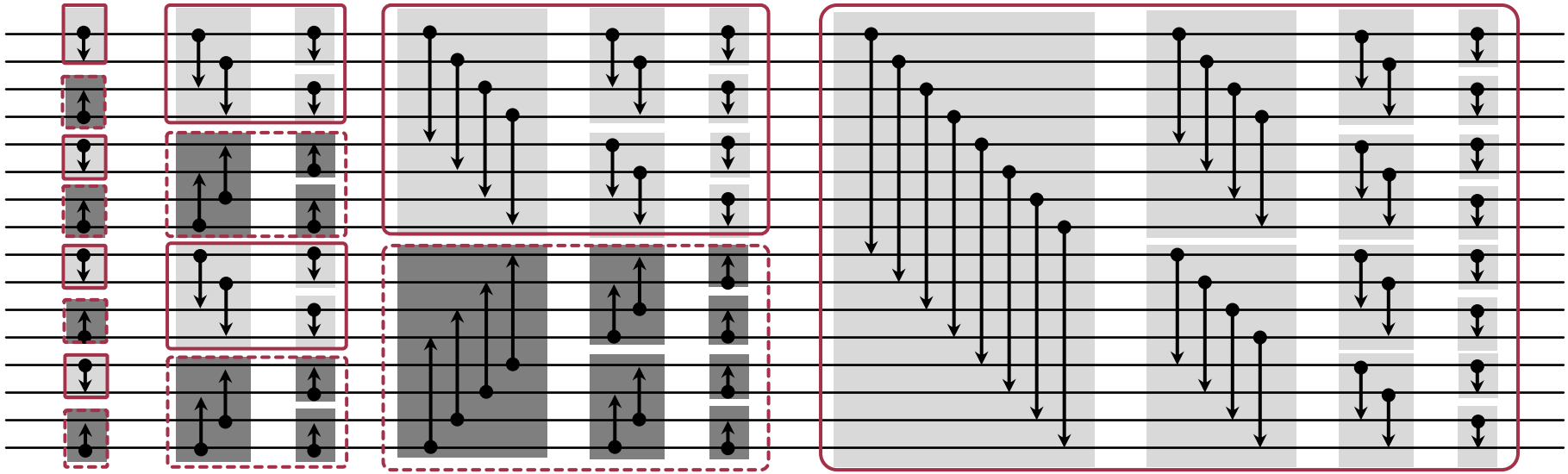


Sortiernetz II

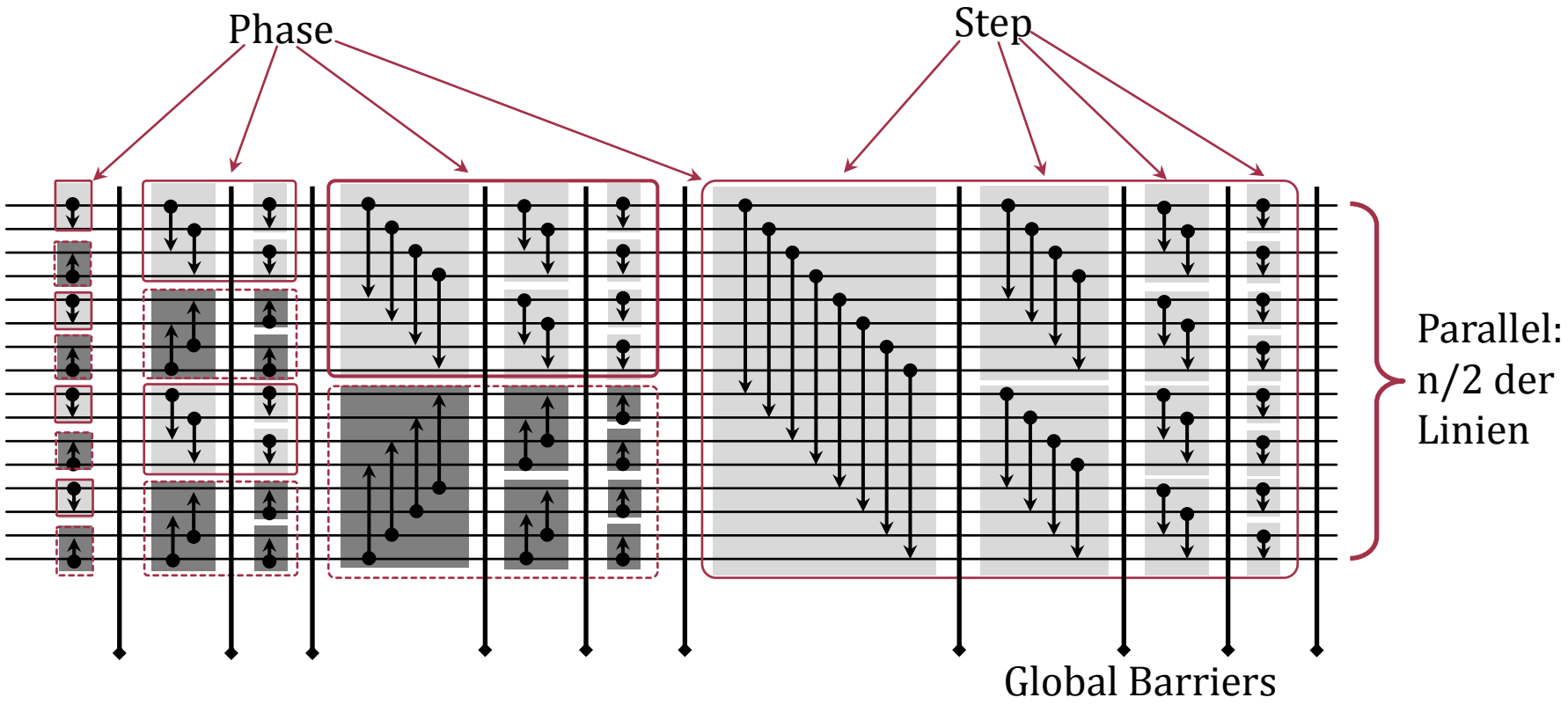
- Jedes Boxenpaar…

  … liefert zwei umgekehrt sortierte Folgen

- Jede OuterBox…
  - Erhält als Eingabe 2 umgekehrt sortierte Folgen
  - Hängt diese aneinander (merge)
    ⇒ Bitonische Folge
  - Liefert sortierte Folge zurück (-sort)
- Letzte Box: Liefert sortierte Folge



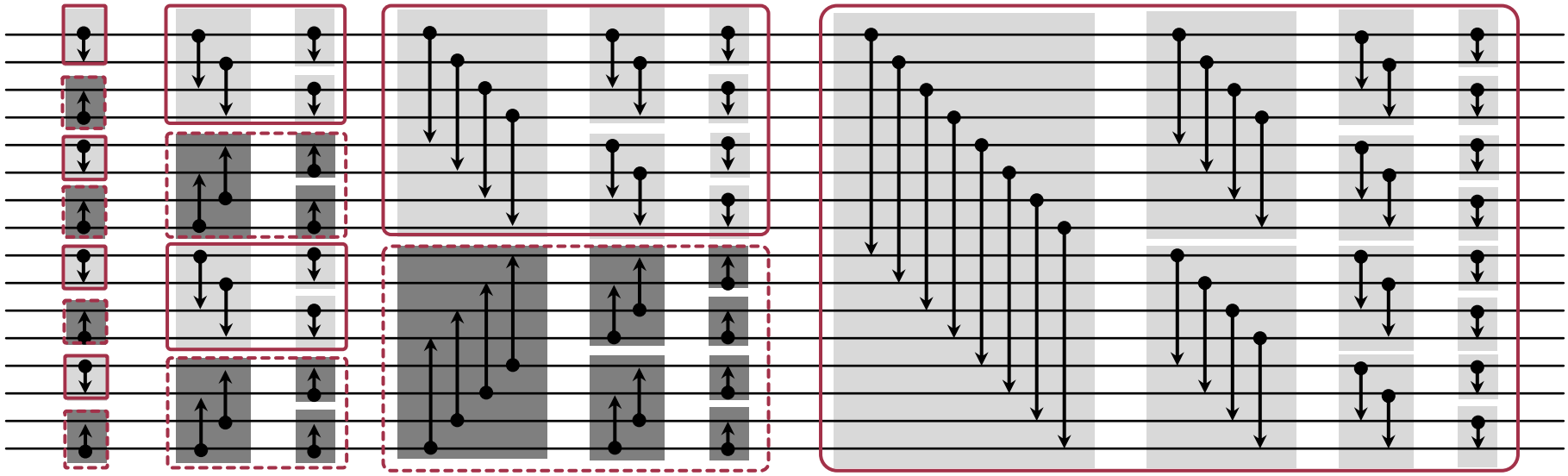
Parallelität



- Parallellität: n/2
- Sequentiell: Entlang der Linien
  - $log_2(n)$ Phases
  - Bis zu $\log_2(n)$ Steps je Phase
- Sicherstellen der Fertigstellung des vorherigen Schrittes mit globaler barrier nach jedem Step

Eigenschaften

- Parallel Running-Time: $O(log(n)^2)$
- Total Cost: $O(n \cdot log(n)^2)$
- Etwas schlechter als Merge-Sort
- Aber: Festes Vergleichsschema gut für parallele Hardware geeignet



Algorithmus

Radix Sort

Key für Radix Sort

- Zum sortieren der Daten nötige Information
  - Muss aus Zeichen eines endlichen Alphabets bestehen
  - Ordnung muss auf Alphabet definiert sein
  - Länge der Keys muss begrenzt sein
  - Typisch: Integer
- Digit:
  - Eine Stelle der Keys
  - Besteht aus je genau einem Zeichen des Alphabets
- Beispiel für Key / Alphabet:
  - Ganze Zahlen 0, … , 999
  - Alphabet: 0, 1, …, 9
  - Key Länge / Stellenzahl: 3
- Weiteres Beispiel für Key / Alphabet
  - 32 Bit Unsigned int
  - Alphabet: Bit (0, 1)
  - Key Länge: 32

Ein Verfahren

- Sortiere nacheinander gemäß je eines Digits …
  - … von rechts nach links: (L)east (S)ignificant (D)igit
  - Oder umgekehrt: Most Significant Digit (MSD)
- Ein Verfahren für LSD Radix Sort:
  - Es gebe Alphabet-Größe viele Buckets
  - GroupKeys(k, s, b): Gruppiere Keys k gemäß Stelle s unter Beibehaltung der Unterordnung in Buckets b
  - Gather(k, b): Sammle Keys k aufsteigend aus Buckets b wieder ein

```
Keys k
Buckets b // (Alphabetgröße viele)
For(Stelle s ← LSD; s ≤ MSD; s eins signifikanter) do
  | call GroupKeys(k, s, b)
  | call Gather(k, b)
End
```

Beispiel

Key: Ganze Zahlen 0, … , 99, Alphabet: 0, 1, …, 9

Iteration I

Keys	90	4	13	9	90	23	24	3	90	0

Buckets	0	1	2	3	4	5	6	7	8	9
	90 90 90 0			13 23 3	4 24					9

Iteration II

Keys	90	90	90	0	13	23	3	4	24	9

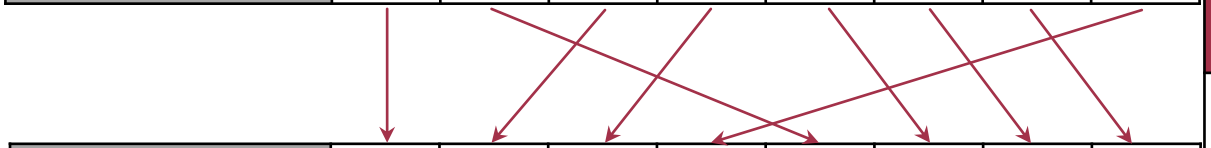
Buckets	0	1	2	3	4	5	6	7	8	9
	0 3 4 9	13	23 24							90 90 90

Keys	0	3	4	9	13	23	24	90	90	90

Paralleles Radix Sort (LSD, 1. Iteration)

Index „i“	0	1	2	3	4	5	6	7	
Keys „in“	100	111	010	110	011	101	001	000	3Bit uint, n=8, Alphabet: Bits
LSB „b“	0	1	0	0	1	1	1	0	
int[] „falses“ int[] „falsesS“	1	0	1	1	0	0	0	1	**setFalses()** b[i] = 0? Ja: falses[i] = falsesS[i] <- 1 Sonst: falses[i] = falsesS[i] <- 0
int[] „falsesS“	0	1	1	2	3	3	3	3	**scanFalses()** Wende (Exclusive Scan, +) auf falsesS an
									getTotalFalses() tf <- falsesS[n-1] + falses[n-1] = 4
int[] „address“	0	4	1	2	5	6	7	3	falses[i] = 1? True: address[i] <- falsesS[i] False: address[i] <- i - falsesS[i] + tf

In	100	111	010	110	011	101	001	000
Out	100	010	110	000	111	011	101	001



Scatter()

out[address[i]] <- in[i]

Algorithmus für k-Bit unsigned Integer

```
uint[] keys                                          // size: n, initialisiert
uint[] sortedKeys, falses, falsesS                   // size: n
for (int bit = 0; bit < k; bit++) do //k z.B. 32 (uint) oder 64 (ulong)
    uint bitMask ← 1 << bit
    if (bit % 2 = 0)
        in ← keys, out ← sortedKeys
    else
        out ← keys, in ← sortedKeys
    end
    for each (key k) in parallel do
        call setFalses(in, bitMask, falses, falsesS, k) // b lokal ber.
    end

    execute parallelExclusiveScanPlus(falsesS)

    int tf ← call getTotalFalses(falses, falsesS)

    for each (key k) in parallel do
        call scatter(in, out, falses, falsesS, tf, k) // Addr. lokal ber.
    end
end
sortedKeys <- out
```

Bewertung

- n Keys, Key Länge: k
- Parallelität:
  - Scan: n/2, n/4, …, 1
  - Rest: n
  - Außer: getTotalFalses, aber redundant berechenbar
  - Scalable
- Gesamtalgorithmus mit k Iterationen
  - Total Complexity: $O(k \cdot n)$
  - Parallel Running Time: $O(k \cdot \log(n))$
  - Speicher: $O(n)$
- Beispiel: Datentyp Integer
  - k fest, z.B.: 4, 16, 32 oder 64
  - Total Complexity: $O(n)$
  - Parallel Running Time: $O(\log(n))$
- Kein vergleichsbasiertes Sortierverfahren
  $\Rightarrow$ untere Schranke $O(n \cdot \log(n))$ gilt nicht