

Übung 8 – Algorithmen II

Moritz Laupichler, Hans-Peter Lehmann — {moritz.laupichler, hans-peter.lehmann}@kit.edu http://algo2.iti.kit.edu/AlgorithmenII_WS22.php

Institut für Theoretische Informatik - Algorithmik II

```
sweath - current weight:
    PROPERTY STATE
or( idget0 eid = graph.edgeBegin( current ); eid != graph.edgeEnd( current ); ++eid ){
 const Edge & edge = graph.getEdge( eid );
 COUNTING( statistic data.inc( DijkstraStatisticData::TOUCHED EDGES ); )
if( edge. forward ){
   COUNTING( statistic data.inc( DijkstraStatisticData::RELAXED EDGES ); )
   Weight new weight = edge.weight + current weight;
  GUARANTEE( new weight >= current weight, std::runtime error, "Weight overflow detected
 if( !priority queue.isReached( edge.target ) ){
     COUNTING( statistic data.inc( DijkstraStatisticData::SUCCESSFULLY RELAXED EDGES )
    COUNTING( statistic data.inc( DijkstraStatisticData::REACHED MODES )
   priority queue.push( edge.target, new weight ):
} else {
  if( priority queue.getCurrentKey( edge.target ) > new wellphill
     COUNTING( Statistic data.inc( DijkstrastatisticData | tuccastaning v 480 Aven | Nove
     priority queue.decreasekey( edge target, new weight)
```

Organisatorisches



- Anmeldung zur Klausur im CAS ist freigeschaltet
- Besprechung ÜB 3 onächstes Mal

Themenübersicht



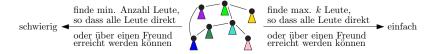
- Parameterisierte Algorithmen
- Parallele Algorithmen

fixed parameter tractable (FPT)



Warum Probleme parametrisieren?

- es gibt "schwierige" Probleme z.B. Minimum Independent Set
 - → allgemeine Instanzen haben zu lange Berechnungszeit
- Kann man Spezialfälle eventuell effizient berechnen?
 - → Identifizierung zusätzlicher Parameter k der Problemstellung
 - \rightarrow falls "Komplexität" in diesen Parametern k steckt, effiziente Lösungen für k=const. !



Parametrisierte Algorithmen Definition



Ein Problem heißt fixed parameter tractable, wenn es eine Laufzeit

$$T(n, k) = \mathcal{O}(f(k) \cdot p(n))$$

hat, mit $f(\cdot)$ berechenbar, $p(\cdot)$ Polynom.

 $f(\cdot)$ darf nicht von n abhängen und $p(\cdot)$ nicht von k; häufig Entscheidungsprobleme

Techniken



Tiefenbeschränkte Suche

- erschöpfendes Aufzählen und Testen aller Möglichkeiten
 - ightarrow mit geeignetem Suchbaum beschränkter Tiefe

k gibt Hinweis, wie weit man in die Tiefe gehen muss



Kernbildung

- Probleminstanz auf (schwierigen) Problemkern reduzieren
- Problemkern mit anderer Technik lösen



I



Schiebepuzzle

Problemstellung

- **g**egeben $n \times n$ Schiebepuzzle, $k \in \mathbb{N}$
- entscheide, ob das Puzzle in $\leq k$ Zügen gelöst werden kann
 - adas Puzzle ist gelöst, wenn die Teile sortiert sind
 - Loch wird pro Zug eine Position horizontal oder vertikal verschoben

Algorithmus A

- es gibt \leq 4 Möglichkeiten in jedem Zug, k Züge
 - baue Suchbaum (Höhe k, Verzweigungsgrad ≤ 4)
 → Baumgröße O(4^k)
 - teste jeden Knoten auf korrekte Lösung
 - ightarrow Aufwand $\mathcal{O}(\mathit{n}^2) \in \mathcal{O}(\mathit{poly}(\mathit{n}))$

24	8	13	12	20
11	2		17	21
7	15	14	19	5
6	10	3	9	1
4	23	11	18	22

\Rightarrow	Gesamtaufwand:	0	$(4^k n^2)$	$\Rightarrow FPT$
---------------	----------------	---	-------------	-------------------

$$T(n,k) = 4T(n,k-1) + poly(n)$$



Schiebepuzzle

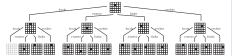
Problemstellung

- **g**egeben $n \times n$ Schiebepuzzle, $k \in \mathbb{N}$
- entscheide, ob das Puzzle in $\leq k$ Zügen gelöst werden kann
 - das Puzzle ist gelöst, wenn die Teile sortiert sind
 - Loch wird pro Zug eine Position horizontal oder vertikal verschoben

Algorithmus A

- es gibt \leq 4 Möglichkeiten in jedem Zug, k Züge
 - baue Suchbaum (Höhe k, Verzweigungsgrad ≤ 4)
 → Baumgröße O(4^k)
 - teste jeden Knoten auf korrekte Lösung
 - ightarrow Aufwand $\mathcal{O}(n^2) \in \mathcal{O}(poly(n))$
- \Rightarrow Gesamtaufwand: $\mathcal{O}(4^k n^2) \Rightarrow \text{FPT}$

$$T(n,k) = 4T(n,k-1) + poly(n)$$

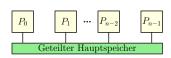


Parallelverarbeitung Modelle



PRAM (Shared Memory)

- synchrone Prozessoren
- gemeinsamer Speicher
- Speicherkonflikte



(symmetrisch) gemeinsamer Speicher

Verteilter Speicher (Distributed Memory)

BulkSynchronousParallel

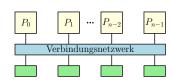
- kollektiver Nachrichtenaustausch aller Rechner
- BSP* berücksichtigt Nachrichtenlänge

Parallelverarbeitung Modelle



PRAM (Shared Memory)

- synchrone Prozessoren
- gemeinsamer Speicher
- Speicherkonflikte



(symmetrisch) gemeinsamer Speicher

Verteilter Speicher (Distributed Memory)

BulkSynchronousParallel

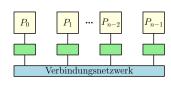
- kollektiver Nachrichtenaustausch aller Rechner
- BSP* berücksichtigt Nachrichtenlänge

Parallelverarbeitung



PRAM (Shared Memory)

- synchrone Prozessoren
- gemeinsamer Speicher
- Speicherkonflikte



(symmetrisch) gemeinsamer Speicher

Verteilter Speicher (Distributed Memory)

BulkSynchronousParallel

- kollektiver Nachrichtenaustausch aller Rechner
- BSP* berücksichtigt Nachrichtenlänge



Struktur

Vollverkabelt

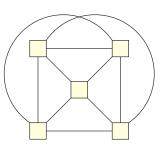
- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex
 - Telefon
 - Duplex

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn $* * * 1 * * \leftrightarrow * * * 0 * *$

$$***1** \leftrightarrow ***0**$$







Struktur

Vollverkabelt

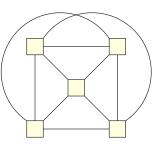
- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$







Struktur

Vollverkabelt

- nur für geringe Anzahl an Rechnern
- Varianten
 - **Simplex** $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$

0

Kosten



Struktur

Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - **Simplex** $i \rightarrow j$
 - **Telefon** $i \leftrightarrow i$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn



Kosten



Struktur

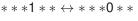
Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$



Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn





Kosten



Struktur

Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$



Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$



Kosten



Struktur

Vollverkabelt

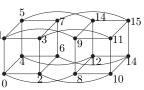
- nur für geringe Anzahl an Rechnern
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

↑ → **/** →

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$



Kosten



Struktur

Vollverkabelt

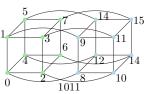
- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - **Simplex** $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

↑ → **/** →

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$



Kosten



Struktur

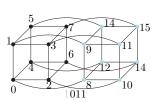
Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$



Kosten



Struktur

Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - **Simplex** $i \rightarrow j$
 - **Telefon** $i \leftrightarrow i$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn $* * * 1 * * \leftrightarrow * * * 0 * *$



Kosten

Kostenmaß Kommunikation $T_{comm} = T_{start} + I \cdot T_{byte}$

10

011



Struktur

Vollverkabelt

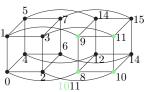
- nur für geringe Anzahl an Rechnern
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

↑ → **/** →

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$



Kosten



Struktur

Vollverkabelt

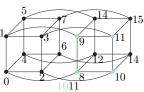
- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

† → / →

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn





Kosten



Struktur

Vollverkabelt

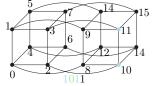
- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

→ / **→**

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

$$***1** \leftrightarrow ***0**$$



Kosten



Struktur

Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - **Simplex** $i \rightarrow j$
 - Telefon $i \leftrightarrow j$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn







Struktur

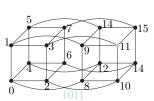
Vollverkabelt

- nur für geringe Anzahl an Rechnern
- $\frac{p \cdot (p-1)}{2}$ Verbindungen nötig
- Varianten
 - Simplex $i \rightarrow j$
 - Telefon $i \leftrightarrow j$
 - **Duplex** $i \rightarrow j, k \rightarrow i$

Hyperwürfel

- p log p Verbindungen
- klare Nummerierung von Nachbarn

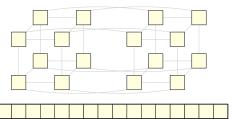
$$***1** \leftrightarrow ***0**$$



Kosten



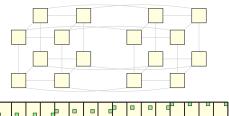
- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - 2. Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow$ gehört zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID

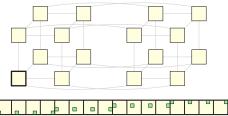


- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört zur Präfixsumme$
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID

 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



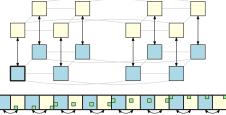
- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört zur Präfixsumme$
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



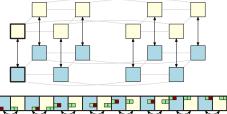
- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow$ gehört zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID

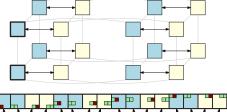


- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID

 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



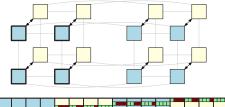
- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



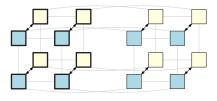
- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$



- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID

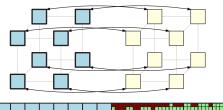


- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme
 - ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$

Präfixsumme - Hypercube



- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



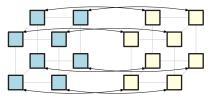
- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme

- ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$

Präfixsumme - Hypercube



- Jede CPU speichert zwei Werte
 - 1. Summe aller bekannten Elemente
 - Summe aller bekannten Elemente von CPUs mit kleinerer ID



- In Schritt k tauschen CPUs entlang der k-ten Dimension ihre Summen aus.
 - ID[k] = 1: Summe aller Elemente beim Kommunikationspartner kommt von CPUs mit kleinerer $ID \rightarrow gehört$ zur Präfixsumme

- ID[k] = 0: Erhaltene Daten gehören zu CPUs größerer ID
 → gehört nicht zur Präfixsumme
- $T(n, p) = (T_{start} + I \cdot T_{byte}) \cdot \log p$

PRAM Präfixsumme



- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme



- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

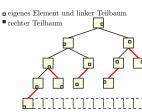
- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts



Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

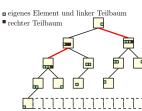
- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts



Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

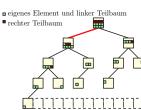
- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts



Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

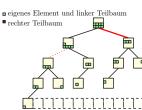
- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts



Aufwärtsphase

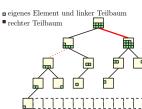
- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts



Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

n eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

PRAM Präfixsumme

Karlsruher Institut für Technologie

- viele komplexere Algorithmen nutzen Reduktionsschemata in Baumform
- hier Beispiel Präfixsumme Fibonacci-Baum
- zweiphasig, aufwärts und abwärts

eigenes Element und linker Teilbaum rechter Teilbaum

Aufwärtsphase

- Prozessoren aggregieren Daten aus Teilbäumen
- speichere Summe kleinerer linker Teilbaum und größerer rechter Teilaum Elemente getrennt voneinander
- leite Summe aller Elemente an Vorgängerknoten

- gesammelte Daten werden verteilt
- bisher in Abwärtsphase empfangene Daten; eigene Daten und Daten des linken Teilbaums nur nach rechts

Paralleler Quicksort



- - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme

 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der

Paralleler Quicksort



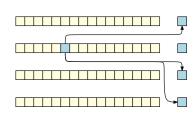
- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerierer
 - ightarrow Präfixsumme
- 5. umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwer
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- 6. Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion



Paralleler Quicksort



- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerierer
 - → Präfixsumme
- 5. umverteiler
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwer
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion



Paralleler Quicksort



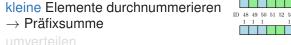
- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerierer
 - → Präfixsumme
- 5. umverteiler
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwer
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion



Paralleler Quicksort

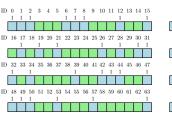


- 1. ein PE stellt Pivot zufällig
- Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren



- - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme

 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der



Paralleler Quicksort



- 1. ein PE stellt Pivot zufällig
- Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
 - → Präfixsumme
- - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme

 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der



Paralleler Quicksort



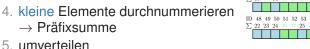
- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
 - → Präfixsumme
- 5. umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwert
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion



Paralleler Quicksort



- 1. ein PE stellt Pivot zufällig
- Broadcast
- 3. lokaler Vergleich



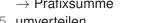
- umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Flemente
 - Position kleine Flemente ist Präfixsummenwert
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente



Paralleler Quicksort

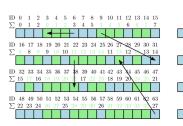


- 1. ein PE stellt Pivot zufällig
- Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
 - → Präfixsumme





- Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Flemente
- Position kleine Flemente ist Präfixsummenwert
- Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente



Paralleler Quicksort



Rekursives Verfahren

- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
 - → Präfixsumme

eren m

- ID 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- $\mathrm{ID}\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31$
- ID 32 33 34 35 36 37 38 30 40 41 42 43 44 45 46 47
- ID 48 40 50 51 52 53 54 55 56 57 58 50 60 61 62 63

- 5. umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwert
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- 6. Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion

Paralleler Quicksort



Rekursives Verfahren

- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
 - → Präfixsumme

eren m

- ID 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- $\mathrm{ID}\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31$
- ID 32 33 34 35 36 37 38 30 40 41 42 43 44 45 46 47
- ID 48 40 50 51 52 53 54 55 56 57 58 50 60 61 62 63

- 5. umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwert
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- 6. Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion

Paralleler Quicksort



4 5 6 7 8 9 10 11 12 13 14 15

19 20 21 22 23 24 25 26 27

- 1. ein PE stellt Pivot zufällig
- Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
- - → Präfixsumme
- umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Flemente
 - Position kleine Flemente ist Präfixsummenwert
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- 6. Prozessoren aufspalten Problematisch bei unbalancierter Verteilung

Paralleler Quicksort



- 1. ein PE stellt Pivot zufällig
- 2. Broadcast
- 3. lokaler Vergleich
- 4. kleine Elemente durchnummerieren
 - → Präfixsumme
- 5. umverteilen
 - Präfixsumme für große Elemente folgt direkt aus ID und Präfixsumme kleiner Elemente
 - Position kleine Elemente ist Präfixsummenwert
 - Position großer Elemente ist Anzahl kleiner Elemente plus Wert der Präfixsumme für große Elemente
- 6. Prozessoren aufspalten Problematisch bei unbalancierter Verteilung
- 7. parallele Rekursion





Parallele Programmierung



Ein Einstieg

Einstieg in parallele Programmierung?

- OpenMP
 - www.openmp.org
 - enthalten im GCC Compiler
 - Parallelität über Preprozessorflags #pragma omp parallel
- Intel Thread Building Block Library (TBB)
 - https://software.intel.com/en-us/tbb
 - mehr objektorientiert als OpenMP
 - enthält konkurrente Datenstrukturen und viele parallele Primitive
 - Konkurrente Queues, Arrays, . . .
 - Parallel Sort, For, While, ...
 - Komplexe Dataflow-Graphen
 - Geschachtelter und rekursiver Parallelismus
- Message Passing Interface (MPI)
 - https://www.mcs.anl.gov/research/projects/mpi/
 - Standard f
 ür verteilte Programmierung
 - implementiert die g\u00e4ngisten Kommunikationsprimitiven (c-Style Interface)

Ende!





Feierabend!