Karlsruhe Institute of Technology

Algorithmen II

Peter Sanders

Übungen:

Moritz Laupichler, Nikolai Maas

Institut für Theoretische Informatik

Web:

`algo2.iti.kit.edu/AlgorithmenII_WS23.php`

10 Parallele Algorithmen

Schnupperkapitel.

Mehr in der gleichnamigen Vorlesung

sowie im Vertiefungsfach Parallelverarbeitung

Warum Parallelverarbeitung

Geschwindigkeitsteigerung: p Computer, die gemeinsam an einem Problem arbeiten, lösen es bis zu p mal so schnell. Aber, viele Köche verderben den Brei $\rightsquigarrow$ gute Koordinationsalgorithmen

Energieersparnis: Zwei Prozessoren mit halber Taktfrequenz brauchen weniger als ein voll getakteter Prozessor. (Leistung$\approx$Spannung $\cdot$ Taktfrequenz)

Speicherbeschränkungen von Einzelprozessoren

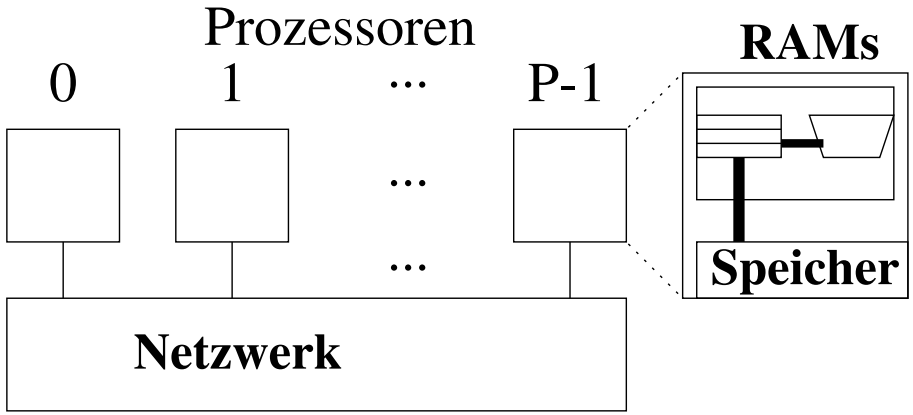
Kommunikationsersparnis: wenn Daten verteilt anfallen kann man sie auch verteilt (vor)verarbeiten

Parallelverarbeitung am ITI Sanders

- Massiv parallele Graph-Algorithmen, Niklas Uhl, Matthias Schimek, Daniel Seemaier
- Parallele Konstruktion von minimalen perfekten Hashfunktionen, Hans-Peter Lehmann
- Shared Memory Datenstrukturen, Marvin Williams
- (Hyper)Graphpartitionierung, Daniel Seemaier, Nikolai Maas, Lars Gottesbüren
- Fehlertoleranz, Lukas Hübner
- SAT-Solving und Planungsprobl., Dominik Schreiber
- Geometrische Algorithmen, Daniel Funke
- String-Algorithmen, Florian Kurpicz

Modell

Nachrichtengekoppelte Parallelrechner



- Prozessoren sind RAMs
- asynchrone Programmabarbeitung
- Interaktion durch Nachrichtenaustausch

Kostenmodell für Nachrichtenaustausch

Jedes PE kann

gleichzeitig maximal eine Nachricht senden und empfangen.

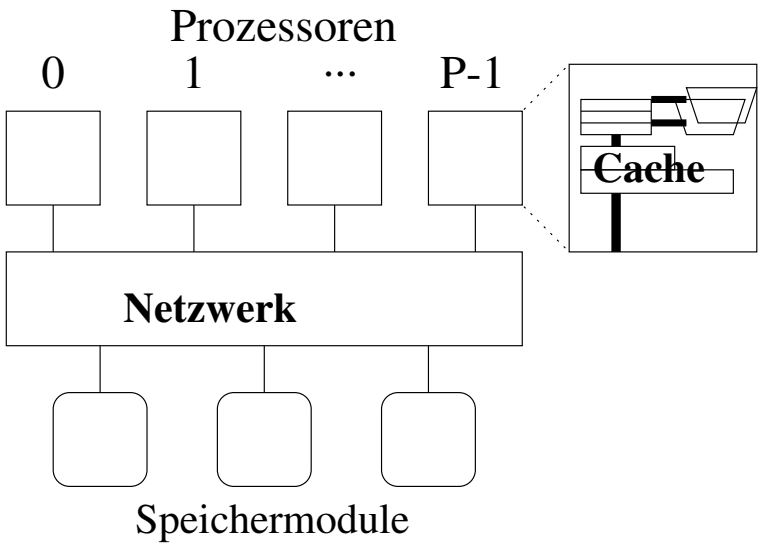
Bei Nachrichtenlänge ℓ dauert das

$$T_{\text{comm}}(\ell) = \alpha + \ell\beta$$

- ☐ vollduplex
- ☐ Punkt-zu-Punkt
- ☐ vollständige Verknüpfung
- ☐ i.allg. $\alpha \gg \beta$ – Alternative: Blockkommunikation analog Sekundärspeichermodell

Warum kein (shared memory) Multicore-Modell

- Unklar wie man damit skalierbaren Parallellismus erreicht
- Unklares Kostenmaß bei Speicherzugriffskonflikten
- Gute Strategie für Parallelprogrammierung: Verteilt entwerfen, vereint implementieren



⇝ mit verteiltem Speicher decken wir den ganzen Bereich vom Multicore-Smartphone zum Superrechner ab und sind noch einigermaßen nah an Cloud, Sensor- oder Peer-to-Peer-Netzen

Formulierung paralleler Algorithmen

Gleicher Pseudocode wie immer.

Single Program Multiple Data Prinzip.

Der Prozessorindex wird genutzt um die Symmetrie zu brechen.

Procedure helloWorldParallel

writeLineAtomic “Hallo, I am PE ” iProc “ out of ” p “processing elements”

```
Hallo, I am PE 0 out of 3 processing elements
Hallo, I am PE 2 out of 3 processing elements
Hallo, I am PE 1 out of 3 processing elements
```

Analyse paralleler Algorithmen

Im Prinzip nur ein zusätzlicher Parameter: p.

Finde Ausführungszeit $T(I,p)$.

Problem: Interpretation.

Work: $W = pT(p)$ ist ein Kostenmaß.

Span: $T_\infty = \inf_p T(p)$ misst Parallelisierbarkeit.

(absoluter) Speedup: $S = T_{\text{seq}}/T(p)$ Beschleunigung. Benutze besten bekannten sequentiellen Algorithmus. Relative Beschleunigung $S_{\text{rel}} = T(1)/T(p)$ ist i.allg. was anderes!

Effizienz: $E = S/p$. Ziel: $E \approx 1$ oder wenigstens $E = \Theta(1)$.

(Sinnvolles Kostenmaß?) "'Superlineare Beschleunigung"': $E > 1$.

(möglich?).

Beispiel: Assoziative Operationen (=Reduktion)

Theorem 1. *Sei* $\bigoplus$ *ein assoziativer Operator, der in konstanter Zeit berechnet werden kann. Dann lässt sich*

$$\bigoplus_{i<p} x_i := (\cdots((x_0 \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_{p-1})$$

in Zeit $O(\log p)$ *berechnen*

Beispiele: $+$, $\cdot$, max, min, ... (z.B. ? nichkommutativ?)

Grundidee für $p = 2^k$ (oBdA?)

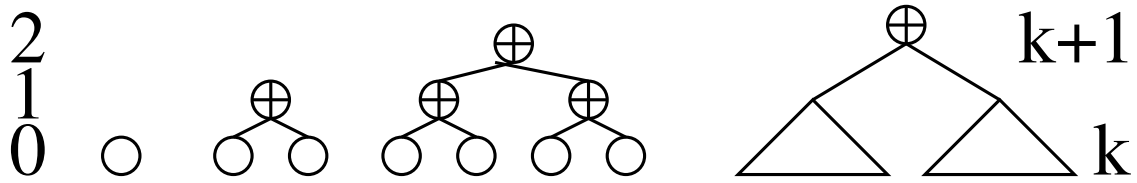
Induktion über k:

$k = 0$: trivial

$k \rightsquigarrow k+1$:

$$\bigoplus_{i<2^{k+1}} x_i = \underbrace{\overbrace{\bigoplus_{i<2^k} x_i}^{\text{Tiefe } k} \oplus \overbrace{\bigoplus_{i<2^k} x_{i+2^k}}^{\text{Tiefe } k \text{ (IA)}}}_{\text{Tiefe } k+1}$$

■



Pseudocode

PE index $i \in \{0,\ldots,p-1\}$

//Input x_i located on PE i

active := 1

$s := x_i$

for $0 \leq k < \lceil \log p \rceil$ **do**

  if active **then**

    if bit k of i **then**

      sync-send s to PE $i - 2^k$

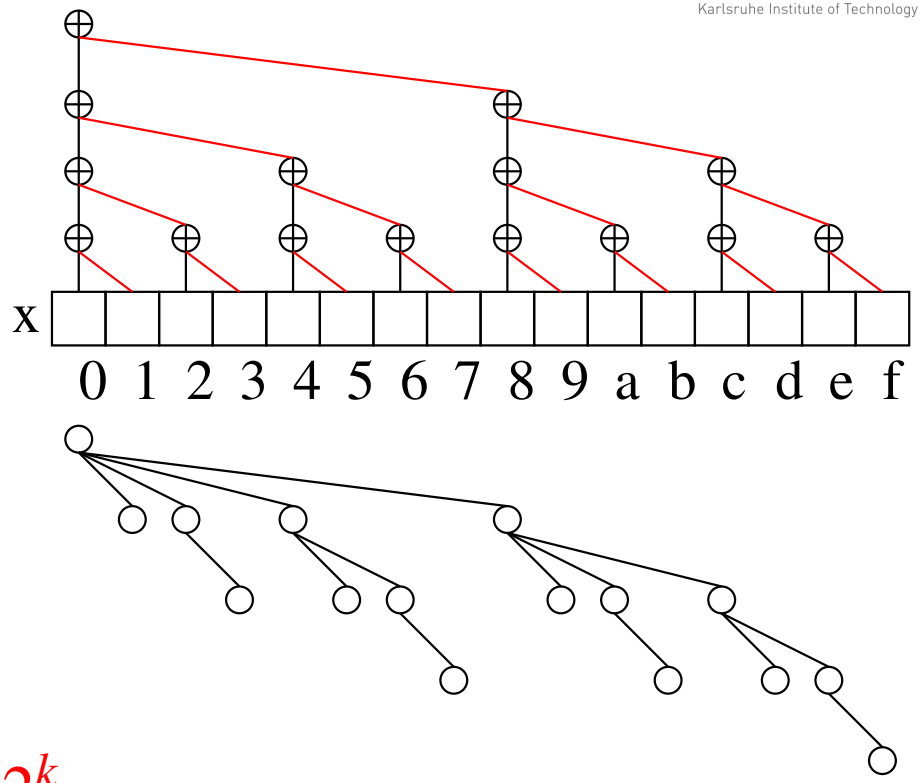
      active := 0

    else if $i + 2^k < p$ **then**

      receive s' from PE $i + 2^k$

      $s := s \oplus s'$

//result is in s on PE 0



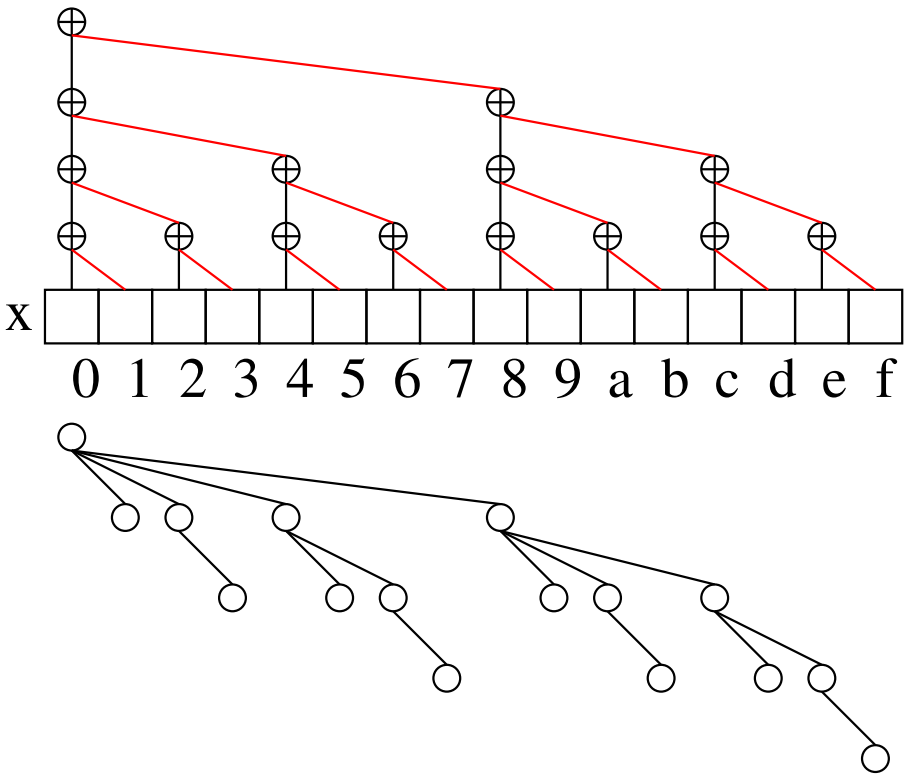
Analyse

n PEs

Zeit $O(\log n)$

Speedup $O(n/\log n)$

Effizienz $O(1/\log n)$



Weniger ist Mehr

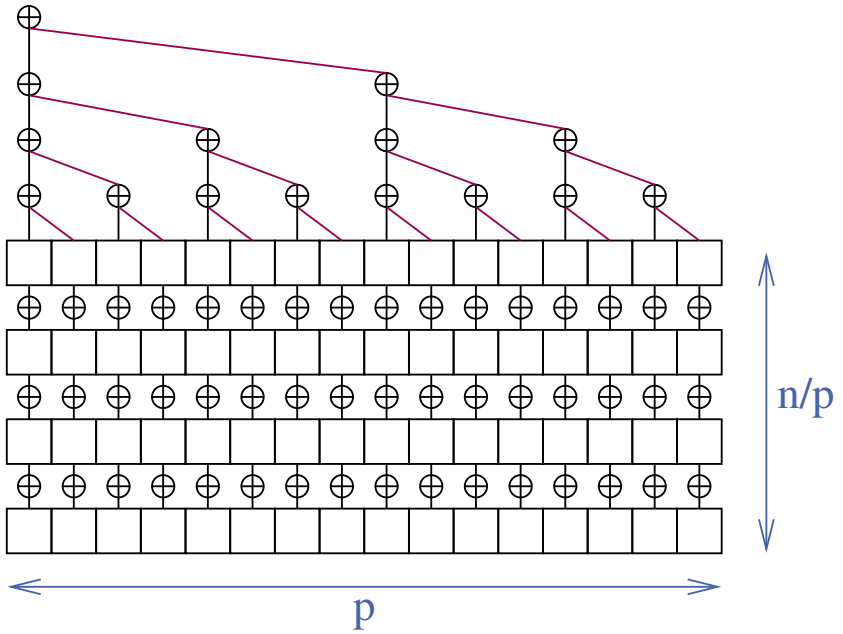
p PEs

Jedes PE addiert

n/p Elemente sequentiell

Dann parallele Summe

für p Teilsummen



Zeit $T_{\text{seq}}(n/p) + \Theta(\log p)$

Effizienz

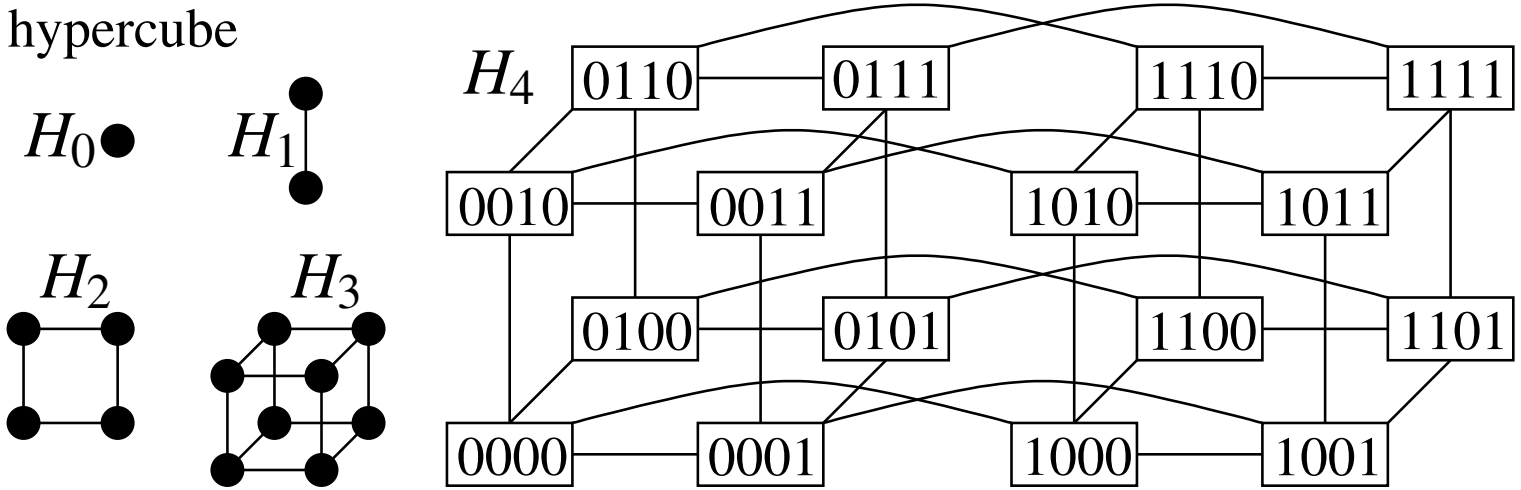
$$\frac{T_{\text{seq}}(n)}{p(T_{\text{seq}}(n/p)+\Theta(\log p))} = \frac{n}{p(n/p+\Theta(\log(p)))} = \frac{1}{1+\Theta(p\log(p))/n}$$

≈ 1 falls $n \gg p \log p$

Diskussion Reduktionsoperation

- Binärbaum führt zu logarithmischer Ausführungszeit
- Nützlich auf den meisten Modellen
- Brent's Prinzip: Ineffiziente Algorithmen werden durch Verringerung der Prozessorzahl effizient

Hyperwürfel

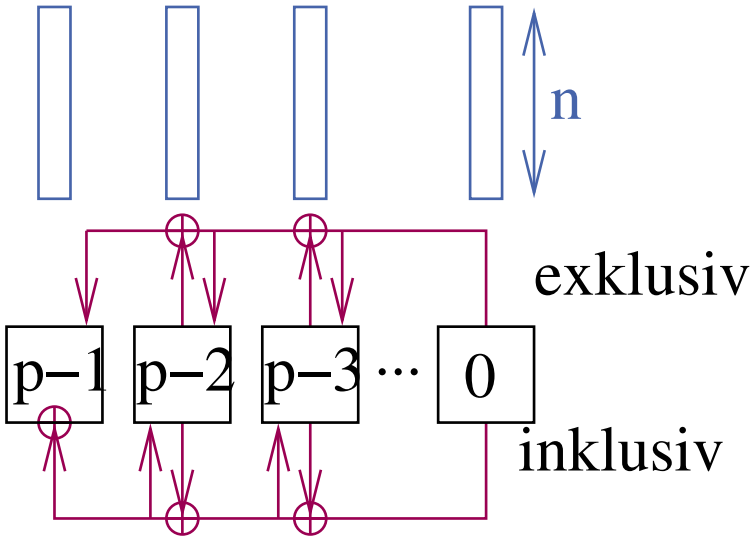


Präfixsummen

Gesucht

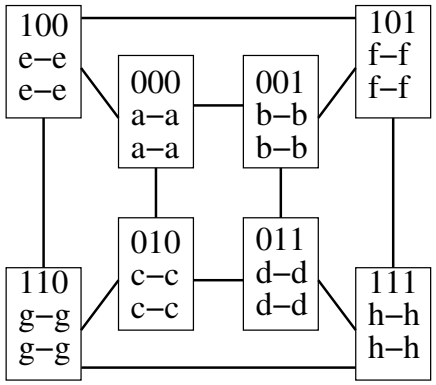
$$x@i := \bigoplus_{i' \leq i} m@i'$$

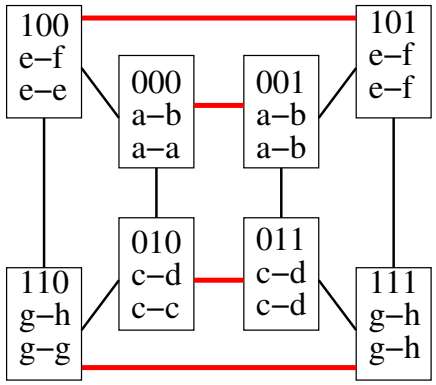
(auf PE i) Objekte der Länge ℓ

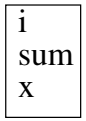


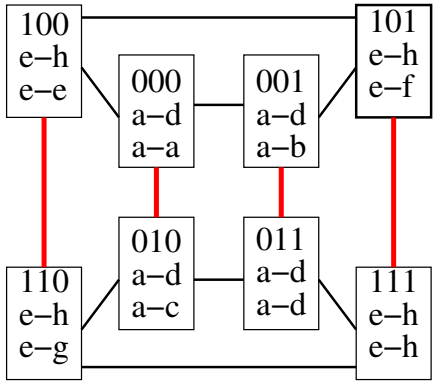
inhärent sequentiell ???

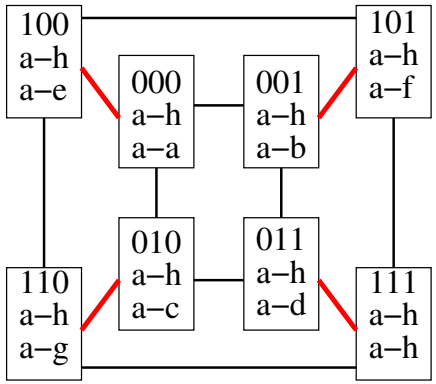
Hyperwürfelalgorithmus











// view PE index i as a

// d-bit bit array

Function hcPrefix(m)

$x := \sigma := m$

for $k := 0$ **to** $d-1$ **do**

invariant $\sigma = \bigoplus_{j=i[k..d-1]0^k}^{i[k..d-1]1^k} \text{m@j}$

invariant $x = \bigoplus_{j=i[k..d-1]0^k}^{i} \text{m@j}$

$y := \sigma@(i \text{ xor } 2^k)$ // sendRecv

$\sigma := \sigma \oplus y$

if $i[k] = 1$ **then** $x := x \oplus y$

return x

Analyse

$$T_{\text{prefix}} = \mathrm{O}((\alpha + \ell\beta)\log p)$$

Problemchen:

Nichtoptimal bei $\ell\beta > \alpha$ (analog schon bei Reduktion)

siehe Spezialvorlesung

$\rightsquigarrow \mathrm{O}(\alpha \log P + \ell\beta)$

Sortieren

- Paralleles Quicksort
- Paralleles Mehrwege-Mergesort
- Hier nicht: binäres Mergesort, Radixsort,...

Paralleles Quicksort

Sequentiell (vereinfacht)

Procedure qSort$(d[], n)$

if $n = 1$ **then return**

select a pivot v

reorder the elements in d such that

$$d_0 \cdots d_{k-1} \leq v < d_k \cdots d_{n-1}$$

qSort$([d_0, \ldots, d_{k-1}], k)$

qSort$([d_{k+1}, \ldots, d_{n-1}], n-k-1)$

Anfänger-Parallelisierung

Parallelisierung der rekursiven Aufrufe.

$$T_{\text{par}} = \Omega(n)$$

- ☐ Sehr begrenzter Speedup
- ☐ Schlecht für distributed Memory

Theoretiker-Parallelisierung

Zur Vereinfachung: $n = p$.

Idee: Auch die Aufteilung parallelisieren.

1. Ein PE stellt den Pivot (z.B. zufällig).
2. Broadcast
3. Lokaler Vergleich
4. “Kleine” Elemente durchnummerieren (Präfix-Summe)
5. Daten umverteilen
6. Prozessoren aufspalten
7. Parallele Rekursion

Theoretiker-Parallelisierung

// Let $i \in 0..p-1$ and p denote the 'local' PE index and partition size

Procedure theoQSort(d, i, p)

if $p = 1$ **then return**

$r :=$ random element from $0..p-1$ // same value in entire partition

$v := d@r$ // broadcast pivot

$f := d \leq v$ // 1 iff d is on left side, 0 otherwise

$j := \sum_{k=0}^{i} f@k$ // prefix sum, count elements on left side

$p' := j@(p-1)$ // broadcast, result is border index

if f **then** send d to PE $j-1$

else send d to PE $p'+i-j$ // $i-j = \sum_{k=0}^{i} d@k > v$

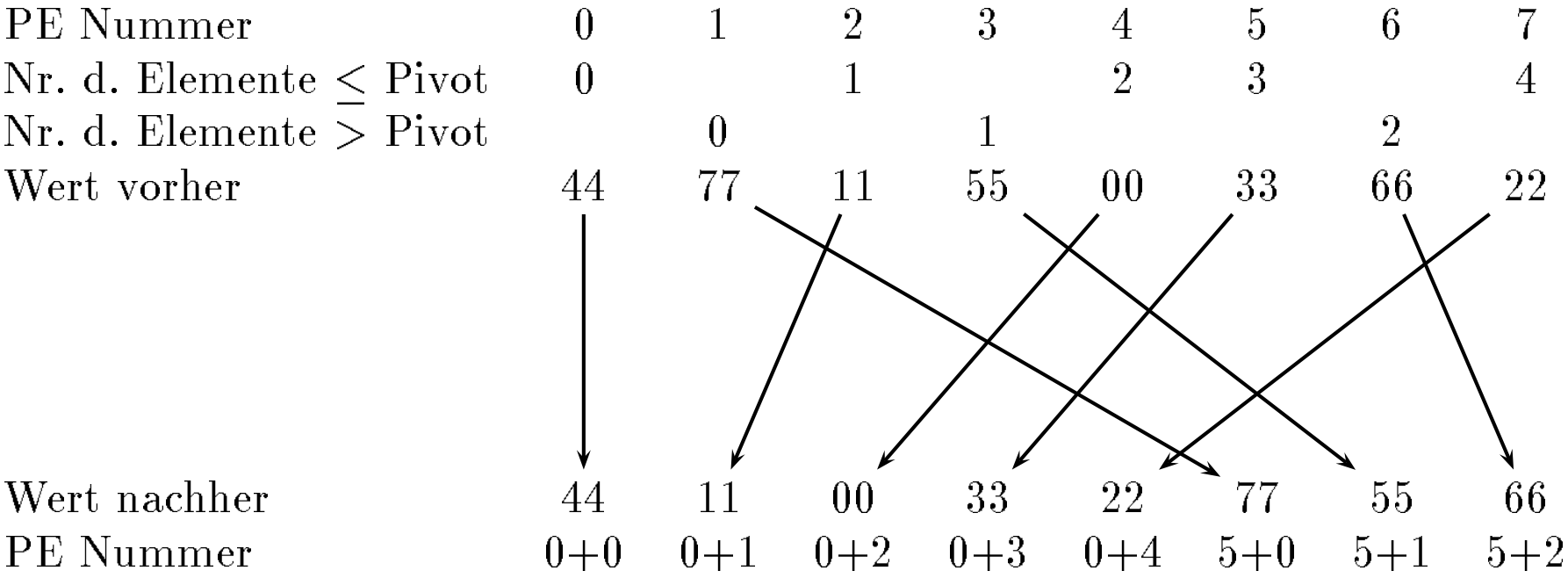
receive d

if $i < p'$ **then** join left partition; qsort(d, i, p')

else join right partition; qsort$(d, i-p', p-p')$

Beispiel

pivot $v = 44$




```
int pQuickSort(int item, MPI_Comm comm)
{ int iP, nP, small, allSmall, pivot;
  MPI_Comm newComm; MPI_Status status;
  MPI_Comm_rank(comm, &iP); MPI_Comm_size(comm, &nP);

  if (nP == 1) { return item; }
  else {
    pivot = getPivot(item, comm, nP);
    count(item <= pivot, &small, &allSmall, comm, nP);
    if (item <= pivot) {
      MPI_Bsend(&item,1,MPI_INT,   small - 1     ,8,comm);
    } else {
      MPI_Bsend(&item,1,MPI_INT,allSmall+iP-small,8,comm);
    }
    MPI_Recv(&item,1,MPI_INT,MPI_ANY_SOURCE,8,comm,&status);
    MPI_Comm_split(comm, iP < allSmall, 0, &newComm);
    return pQuickSort(item, newComm);}}
```

```
/* determine a pivot */
int getPivot(int item, MPI_Comm comm, int nP)
{  int pivot   = item;
   int pivotPE = globalRandInt(nP);/* from random PE */
   /* overwrite pivot by that one from pivotPE */
   MPI_Bcast(&pivot, 1, MPI_INT, pivotPE, comm);
   return pivot;
}

/* determine prefix-sum and overall sum over value */
void
count(int value,int *sum,int *allSum,MPI_Comm comm,int nP)
{  MPI_Scan(&value, sum, 1, MPI_INT, MPI_SUM, comm);
   *allSum = *sum;
   MPI_Bcast(allSum, 1, MPI_INT, nP - 1, comm);
}
```

Analyse

- pro Rekursionsebene:
  - $2\times$ broadcast
  - $1\times$ Präfixsumme

  $\rightsquigarrow$ Zeit $O(\alpha \log p)$
- erwartete Rekursionstiefe: $O(\log p)$

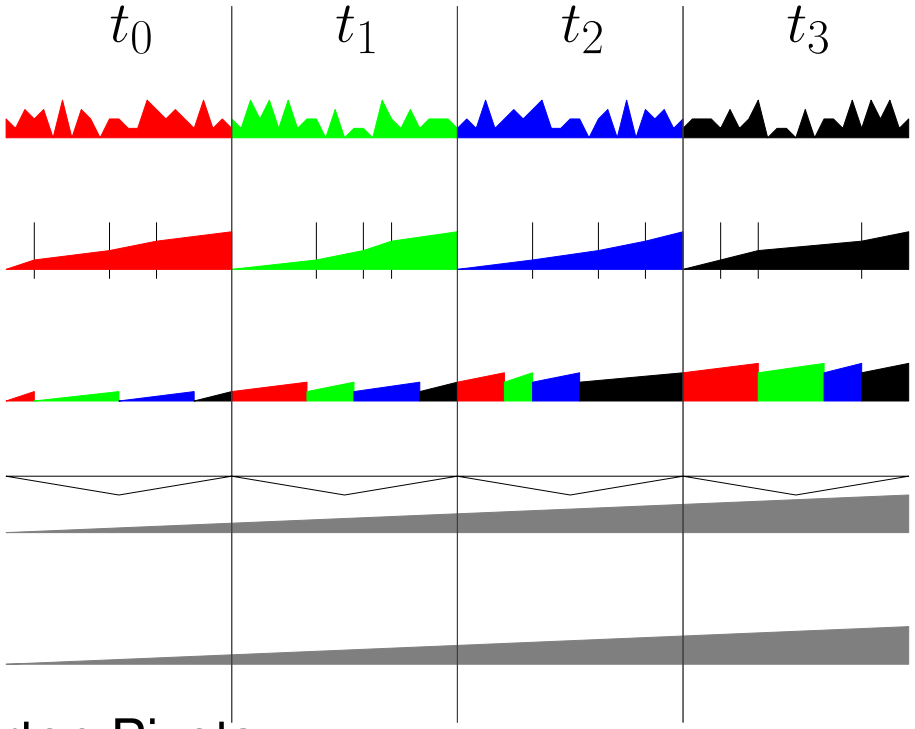
Erwartete Gesamtzeit: $O(\alpha \log^2 p)$

Verallgemeinerung für $n \gg p$ nach Schema F?

- Jedes PE hat i.allg. “große” und “kleine” Elemente.
- Aufteilung geht nicht genau auf
- Präfixsummen weiterhin nützlich
- Unterm Strich ist Zeit $\mathrm{O}\left(\frac{n \log n}{p} + \log^2 p\right)$ möglich
- Bei verteiltem Speicher stört, dass jedes Element $\Omega(\log p)$ mal transportiert wird.
  $\rightsquigarrow \cdots \rightsquigarrow$ Zeit $\mathrm{O}\left(\frac{n}{p}(\log n + \beta \log p) + \alpha \log^2 p\right)$

Paralleles Sortieren durch Mehrwegemischen

1. p Prozessoren sortieren
   je n/p Elemente lokal
2. Finde pivots so dass
   n/p Elemente zwischen
   zwei benachbarten pivots liegen
3. Jeder Prozessor macht
   Mehrwegemischen für alle
   Elemente zwischen zwei benachbarten Pivots.



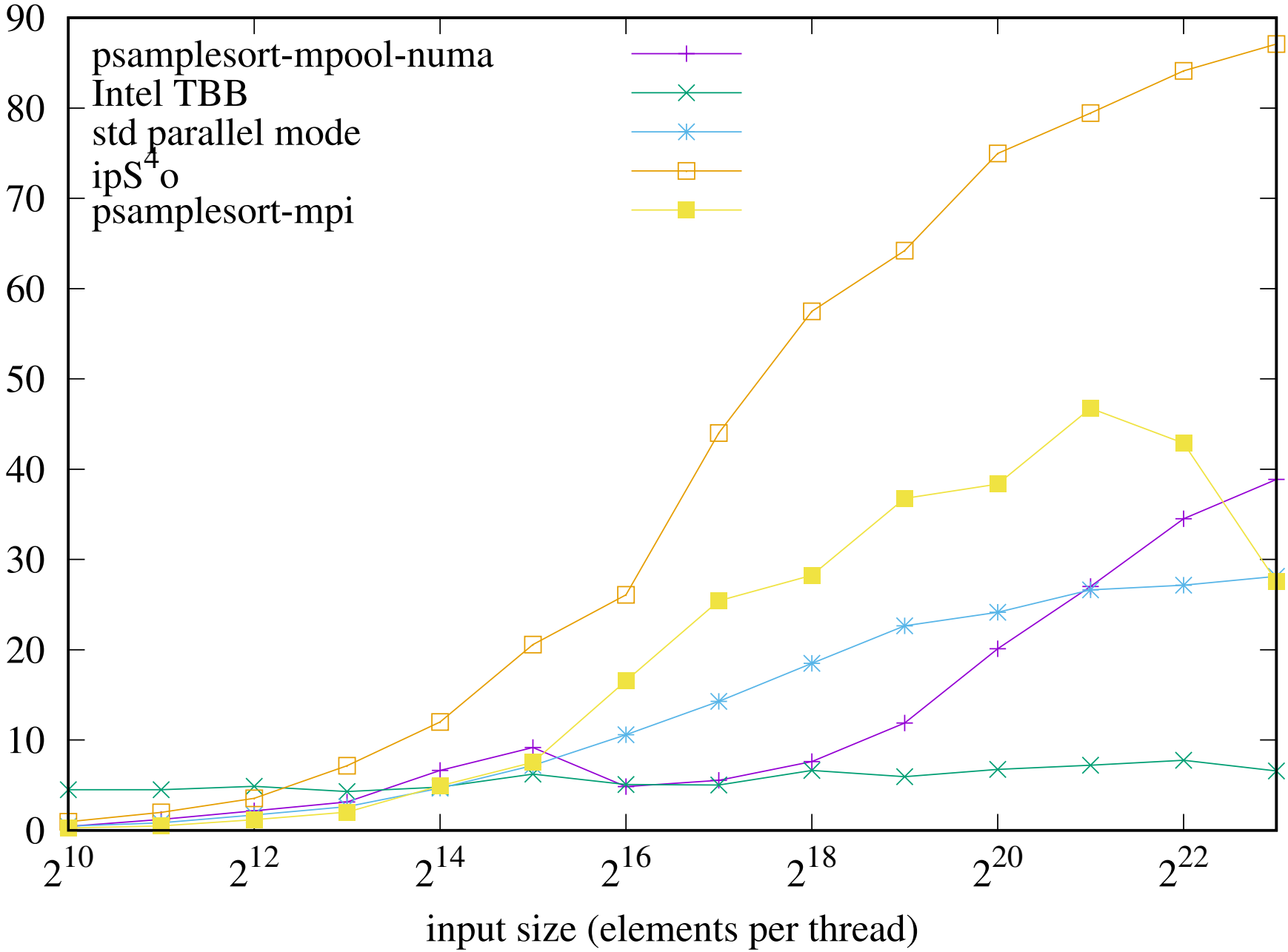
Mehr in “Parallele Algorithmen”

Mehr zu parallelem Sortieren

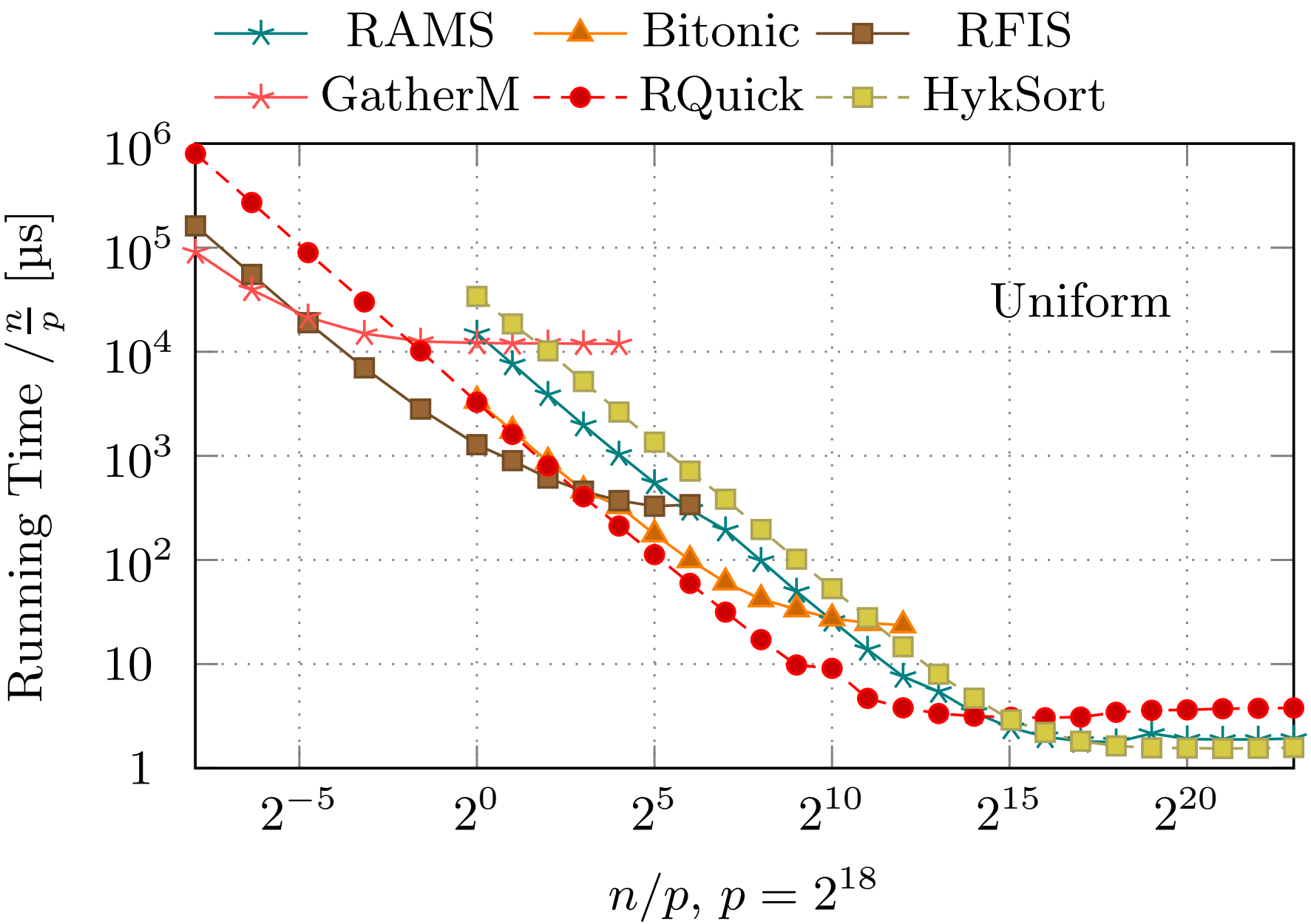
- Theoretikeralgorithmen mit Laufzeit $\mathrm{O}(\log p)$ bei $p = n$
- Praktikable Algorithmen mit Laufzeit $\mathrm{O}(\log p)$ für $n = \mathrm{O}\left(\sqrt{p}\right)$ – wenn schnell wichtiger als effizient ist, z.B. base case
- Sample sort: k-Wegeverallgemeinerung von quicksort
- Paralleles externes Sortieren

Karlsruhe Institute of Technology

Experiments Speedup on 4× Intel E7-8890 v3 – 72 cores



Messungen Axtmann Sanders ALENEX 2017



Mehr zu parallelen Algorithmen – Parallelisierung der Basic Toolbox ?

Verteilte Hashtabellen: geht. Oft teuer $\rightsquigarrow$ [Maier S Dementiev 16]

Prioritätslisten, Suchbäume: ähnliches Problem. Am ehesten batched updates$\rightsquigarrow$ [Hübschle-Schn. S Müller 16][Akhremtsev S 16][Williams D S 2017], aktuelle Masterarbeit

BFS: OK bei kleinem Graphdurchmesser

DFS: $\approx$ inhärent nichtparallelisierbar

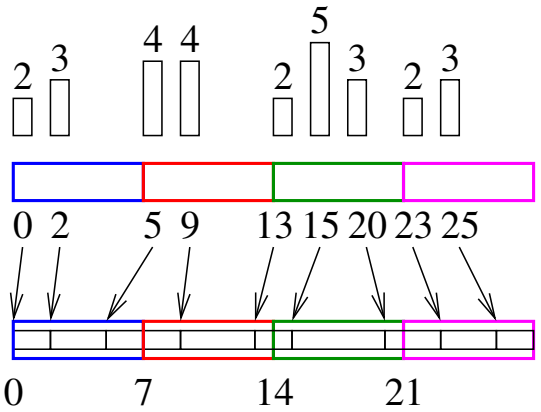
kürzeste Wege: ähnlich BFS. Aber all-to-all, Vorberechnungen OK, multiobjective OK [Mandow S 13]

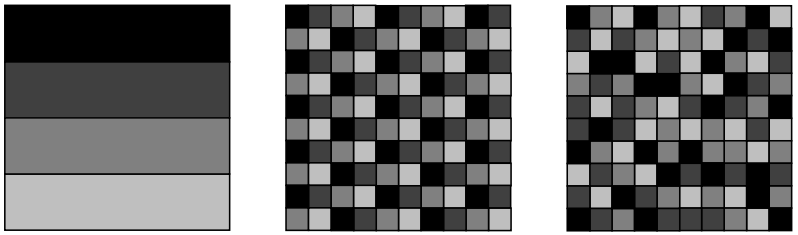
MST: Ja! U.U. große konstante Faktoren [S. Schimek IPDPS 2023]

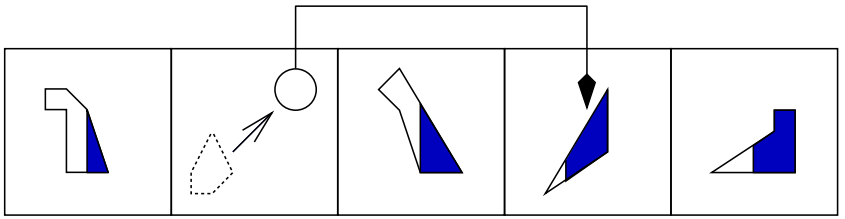
Optimierungstechniken: dynamische Programmierung OK. Greedy??? LP schwierig, Metaheuristiken teils OK

Mehr zu parallelen Algorithmen – Die parallele Basic Toolbox

- Kollektive Kommunikation: Reduktion, Präfixsumme, allg. Nachrichtenaustausch, Gossiping,. . .
- Lastverteilung
  - statisch/dynamisch
  - bekannte/unbekannte Jobgrößen
  - zentralisiert/vollverteilt







processed by: =PE 0 =PE 1 =PE 2 =PE 3