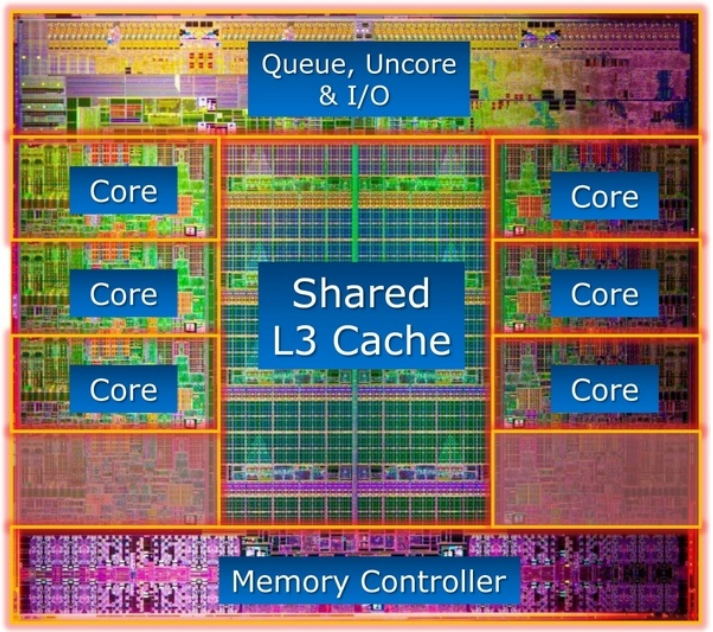
Parallele Algorithmen

Peter Sanders

Institut für Theoretische Informatik



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 eine voll getakteter Prozessor. (Leistung$\approx$Spannung $\cdot$ Taktfrequenz)

Speicherbeschränkungen von Einzelprozessoren

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

Thema der Vorlesung

Grundlegende Methoden der parallelen Problemlösung

- Parallelisierung sequentieller Grundtechniken: Sortieren, Datenstrukturen, Graphenalgorithmen,...
- Basiskommunikationsmuster
- Lastverteilung
- Betonung von beweisbaren Leistungsgarantien
- Aber Anwendbarkeit in „Blickweite“

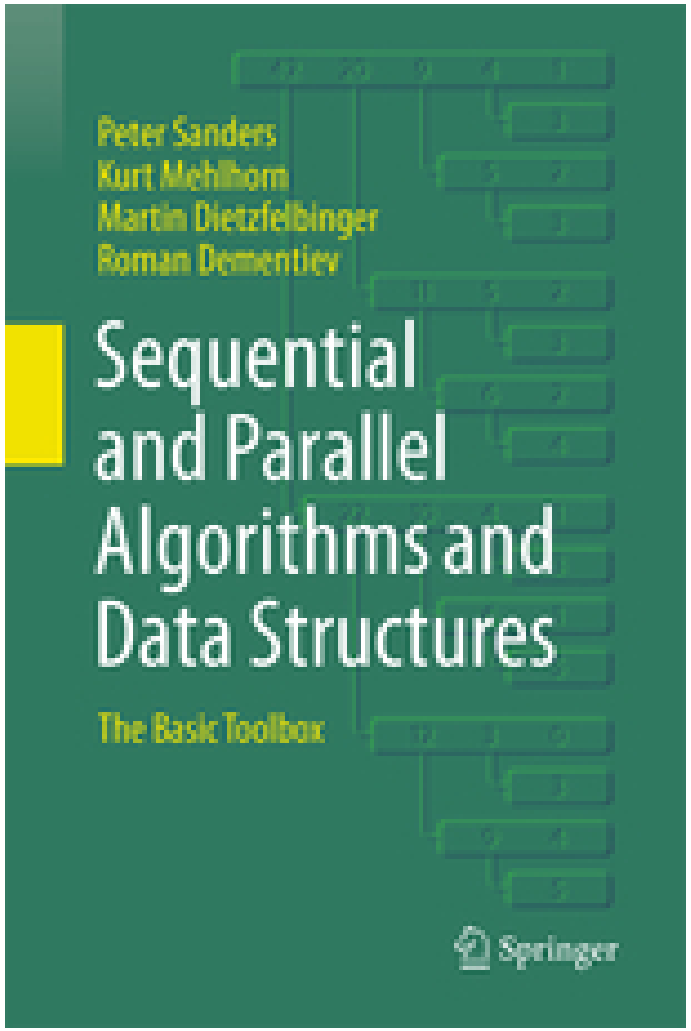
Überblick

- ☐ Modelle, Einfache Beispiele
- ☐ Matrixmultiplikation
- ☐ Broadcasting
- ☐ Sortieren
- ☐ Allgemeiner Datenaustausch
- ☐ Lastverteilung I,II,III
- ☐ Umwandlung verkettete Liste $\rightarrow$ Array
- ☐ Hashing, Prioritätslisten
- ☐ einfache Graphenalgorithmen
- ☐ Graphpartitionierung

Literatur

Skript

+



Viele Abbildungen etc. übernommen markiert durch [Book].

Mehr Literatur

[Kumar, Grama, Gupta und Karypis],

Introduction to Parallel Computing. Design and Analysis of Algorithms,

Benjamin/Cummings, 1994. Praktikerbuch

[Leighton], Introduction to Parallel Algorithms and Architectures,

Morgan Kaufmann, 1992.

Theoretische Algorithmen auf konkreten Netzwerken

[JáJá], An Introduction to Parallel Algorithms, Addison Wesley, 1992.

PRAM

[Sanders, Worsch],

Parallele Programmierung mit MPI – ein Praktikum, Logos, 1997.

Parallelverarbeitung am ITI Sanders

- Massiv paralleles Sortieren, Michael Axtmann
- Massiv parallele Graph-Algorithmen, Sebastian Lamm
- Fehlertoleranz, Demian Hespe
- Big-Data Framework Thrill, Timo Bingmann
- Shared Memory Datenstrukturen, Tobias Maier
- (Hyper)Graphpartitionierung, Tobias Heuer & Yaroslav Akhremtsev
- Kommunikationseff. Alg., Lorenz Hübschle-Schneider
- SAT-Solving und Planungsprobl., Dominik Schreiber
- Geometrische Algorithmen, Daniel Funke

Einbettung in das Informatik-Studium

- Wahlfach oder Mastervorzug im **Bachelorstudium!**
- Vertiefungsfach
  - Algorithmentechnik
  - Parallelverarbeitung
- Studienprofil daten-intensives Rechnen

Schwesterveranstaltungen

Parallelprogrammierung: Tichy, Karl, Streit

Modelle der Parallelverarbeitung: viel theoretischer,

Komplexitätstheorie,... Worsch

Algorithmen in Zellularautomaten: spezieller, radikaler, theoretischer

Worsch

Rechnerarchitektur: Karl

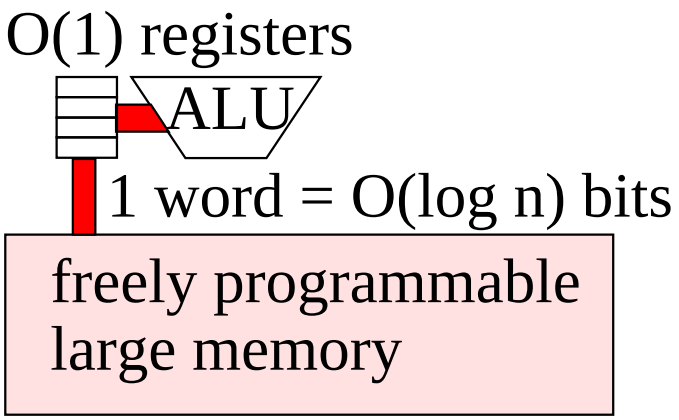
GPUs: Dachsbacher

+ andere Algorithmikvorlesungen

RAM/von Neumann Modell

Analyse: zähle Maschinenbefehle — load, store, Arithmetik, Branch,. . .

- Einfach
- Sehr erfolgreich



Algorithmenanalyse:

- Zyklen zählen: $T(I)$, für gegebene Probleminstanz I.
- Worst case in Abhängigkeit von Problemgröße:
  $T(n) = \max_{|I|=n} T(I)$
- Average case: $T_{\text{avg}}(n) = \frac{\sum_{|I|=n} T(I)}{|\{I : |I| = n\}|}$ Beispiel: Quicksort hat average case Ausführungszeit $\mathcal{O}(n \log n)$
- Probabilistische (randomisierte) Algorithmen: $T(n)$ (worst case) ist eine Zufallsvariable. Wir interessieren uns z.B. für deren Erwartungswert (später mehr).

  Nicht mit average case verwechseln.

  Beispiel: Quicksort mit zufälliger Pivotwahl hat erwarteten worst case Aufwand $\mathbb{E}[T(n)] = \mathcal{O}(n \log n)$

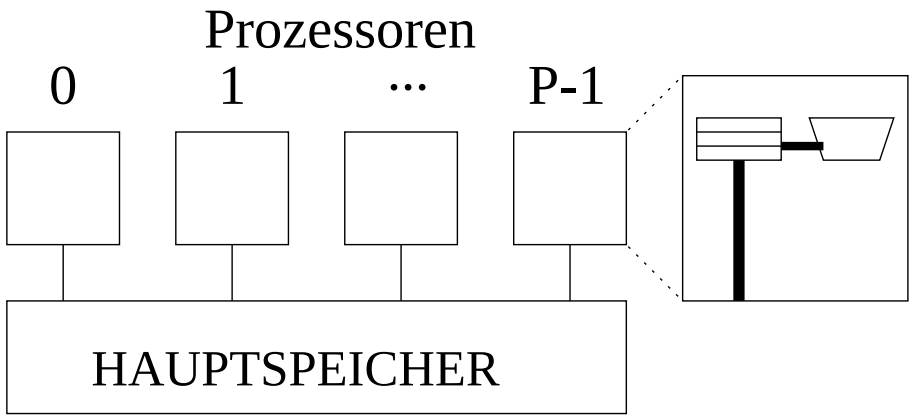
Algorithmenanalyse: Noch mehr Konventionen

- $\mathcal{O}(\cdot)$ plättet lästige Konstanten
- Sekundärziel: Speicherplatz
- Die Ausführungszeit kann von mehreren Parametern abhängen:
  Beispiel: Eine effiziente Variante von Dijkstra's Algorithmus für kürzeste Wege benötigt Zeit $\mathcal{O}(m + n \log n)$ wenn n die Anzahl Knoten und m die Anzahl Kanten ist. (Es muss immer klar sein, welche Parameter was bedeuten.)

Ein einfaches paralleles Modell: PRAMs

Idee: RAM so wenig wie möglich verändern.

- p Prozessoren (**P**rozessor**E**lemente); nummeriert $1..p$ (oder $0..p-1$). Jedes PE kennt p.
- Ein Maschinenbefehs pro Takt und Prozessor synchron
- Gemeinsamer globaler Speicher



Zugriffskonflikte?

EREW: Exclusive Read Exclusive Write. Gleichzeitige Zugriffe verboten

CREW: Concurrent Read Exclusive Write. Gleichzeitiges lesen OK.

Beispiel: Einer schreibt, andere lesen = „Broadcast“

CRCW: Concurrent Read Concurrent Write. Chaos droht:

common: Alle Schreiber müssen sich einig sein. Beispiel: OR in konstanter Zeit (AND?) ←

arbitrary: Irgendeiner setzt sich durch ←

priority: Schreiber mit kleinster Nummer setzt sich durch

combine: Alle Werte werden kombiniert. Zum Beispiel Summe.

Beispiel: Global Or

Eingabe in $x[1..p]$

Sei Speicherstelle Result$= 0$

Parallel auf Prozessor $i = 1..p$

```
if x[i] then Result := 1
```

Global And

Sei Speicherstelle Result$= 1$

```
if not x[i] then Result := 0
```

Beispiel: Maximum auf common CRCW PRAM

[JáJá Algorithmus 2.8]

Input: $A[1..n]$ // distinct elements

Output: $M[1..n]$ // $M[i] = 1$ iff $A[i] = \max_j A[j]$

forall $(i,j) \in \{1..n\}^2$ **dopar** $B[i,j] := A[i] \geq A[j]$

forall $i \in \{1..n\}$ **dopar**

$M[i] := \bigwedge_{j=1}^{n} B[i,j]$ // parallel subroutine

$\mathcal{O}(1)$ Zeit

$\Theta(n^2)$ Prozessoren (!)

i	A	B	1	2	3	4	5	<- j	M
1	3		*	0	1	0	1		1
2	5		1	*	1	0	1		1
3	2		0	0	*	0	1		1
4	8		1	1	1	*	1		1
5	1		0	0	0	0	*		1
	A		3	5	2	8	1		

---

i	A	B	1	2	3	4	5	<- j	M
1	3		*	0	1	0	1		0
2	5		1	*	1	0	1		0
3	2		0	0	*	0	1		0
4	8		1	1	1	*	1		1->maxValue=8
5	1		0	0	0	0	*		0

Formulierung paralleler Algorithmen

- Pascal-ähnlicher Pseudocode
- Explizit parallele Schleifen [JáJá S. 72]
- Single Program Multiple Data Prinzip. Der Prozessorindex wird genutzt um die Symmetrie zu brechen. $\neq$ SIMD !

Analyse paralleler Algorithmen

Im Prinzip nur ein zusätzlicher Parameter: p.

Finde Ausführungszeit $T(I,p)$.

Problem: Interpretation.

Work: $W = pT(p)$ ist ein Kostenmaß. (z.B. Max: $W = \Theta(n^2)$)

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!

(z.B. Maximum: $S = \Theta(n)$, $S_{\text{rel}} = \Theta(n^2)$)

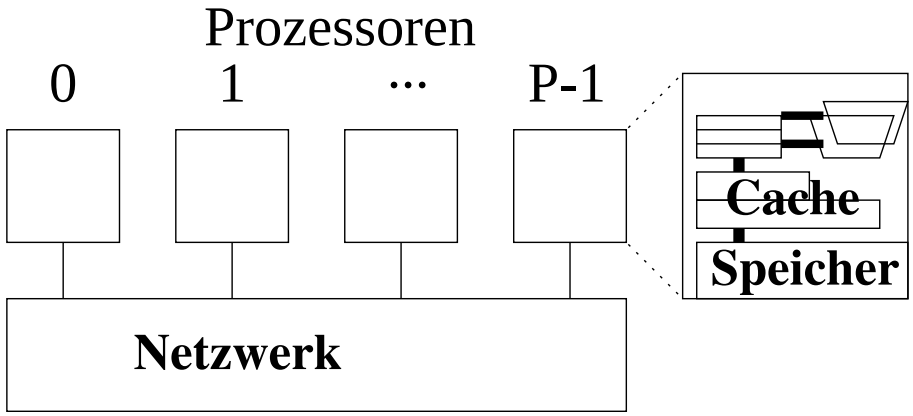
Effizienz: $E = S/p$. Ziel: $E \approx 1$ oder wenigstens $E = \Theta(1)$.

(Sinnvolles Kostenmaß?) „Superlineare Beschleunigung“: $E > 1$.

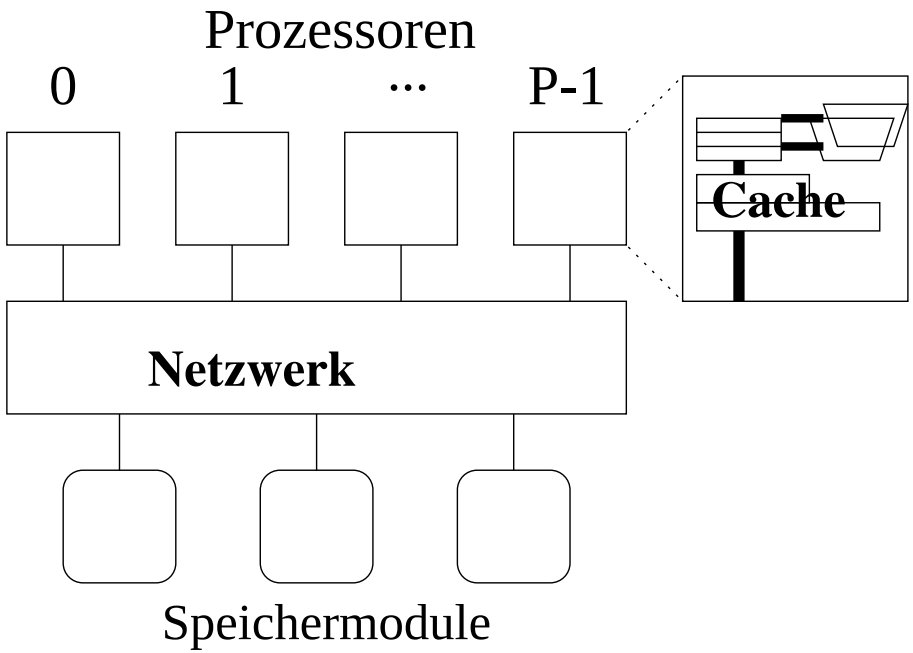
(möglich?). Beispiel Maximum: $E = \Theta(1/n)$.

PRAM vs. reale Parallelrechner

Distributed Memory



(Symmetric) Shared Memory



Probleme

- Asynchron $\rightsquigarrow$ Entwurf, Analyse, Implementierung, Debugging viele schwieriger als PRAM
- Contention (Stau) für gleiche Speichermodule/cache lines. Beispiel: Der $\Theta(1)$ PRAM Algorithmus für globales OR wird zu $\Theta(p)$.
- Lokaler/Cache-Speicher ist (viel) schneller zugreifbar als globaler Speicher
- Das Netzwerk wird mit zunehmendem p komplizierter und die Verzögerungen werden größer.
- Contention im Netzwerk
- Es interessiert der maximale lokale Speicherverbrauch und weniger die Summe der lokalen Speicherverbräuche

Realistic Shared Memory Models

- asynchronous
- aCRQW: asynchronous concurrent read queued write. When x PEs contend for the same memory cell, this costs time $\mathcal{O}(x)$.
- consistent write operations using atomic operations
- memory hierarchies

Why is concurrent read OK?

Atomare Instruktionen: Compare-And-Swap

Allgemein und weit verbreitet:

Function CAS$(a,$ expected, desired$)$: $\{0,1\}$

BeginTransaction

if $*a =$ expected **then** $*a$:= desired; **return** 1// success

else expected:= $*a$; **return** 0// failure

EndTransaction

Weitere Operationen für konsistenten Speicherzugriff:

- Fetch-and-add
- Hardwaretransaktionen

Function fetchAndAdd(a, Δ)

expected:= $*a$

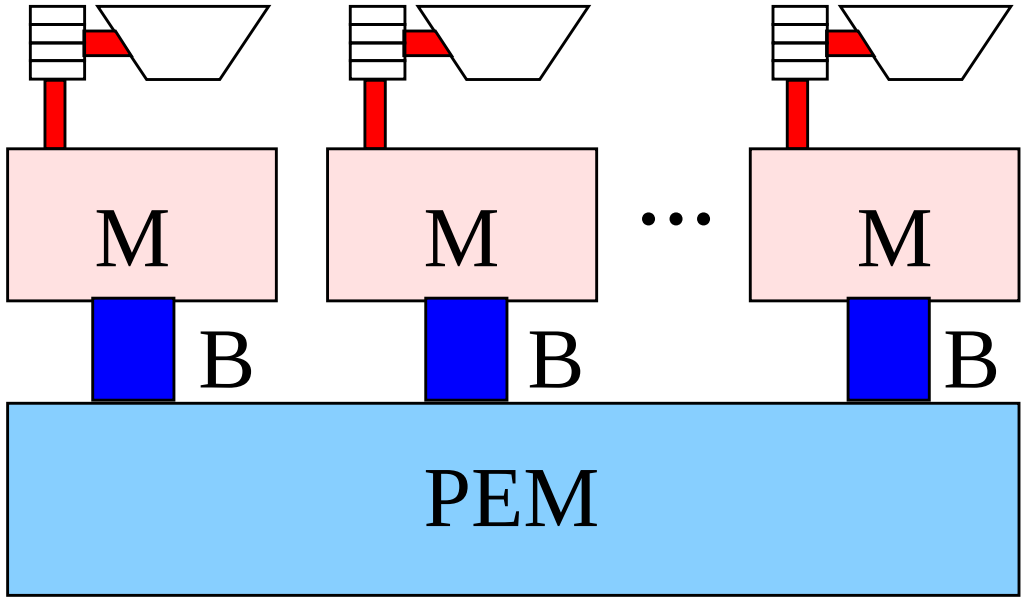
repeat

desired:= expected $+ \Delta$

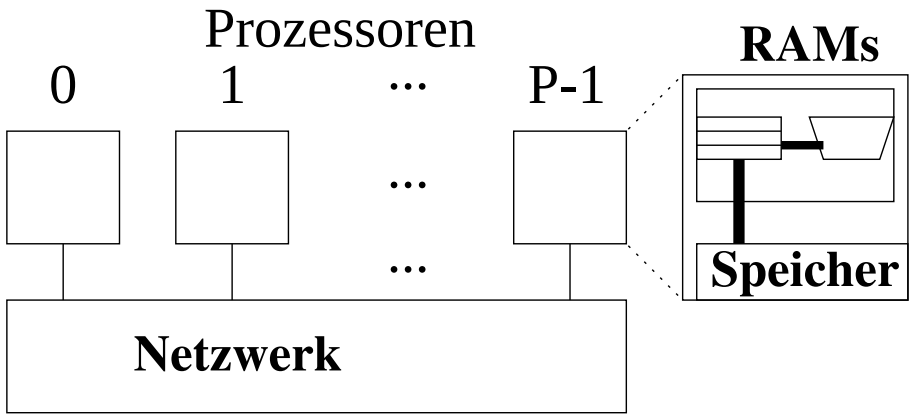
until CAS$(a,$ expected$,$ desired$)$

return desired

Parallel External Memory



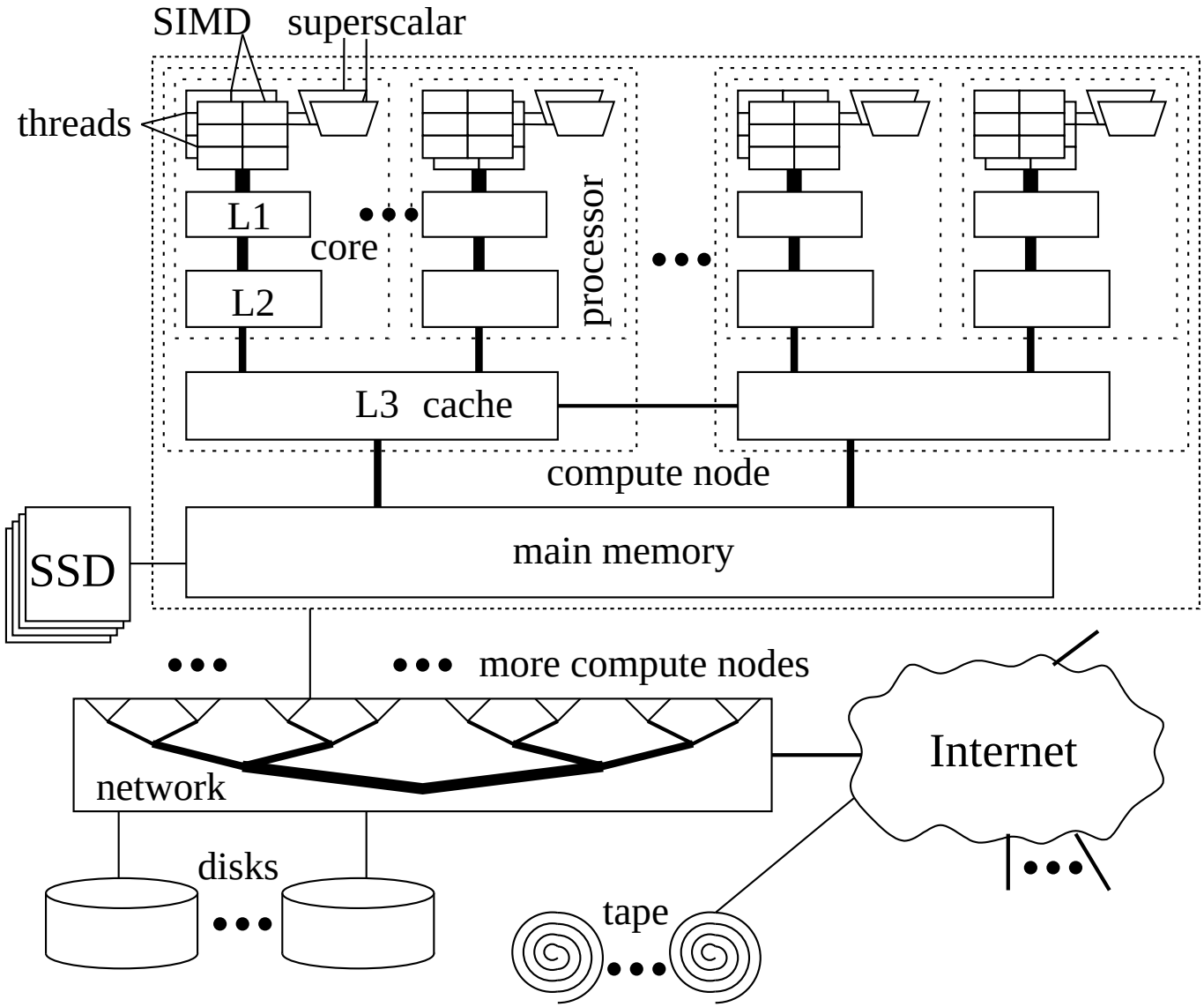
Modelle mit Verbindungsnetzwerken



- Prozessoren sind RAMs
- asynchrone Programmabarbeitung
- Interaktion durch Nachrichtenaustausch

Entscheidend ist das Kostenmodell für den Nachrichtenaustausch

Reale Maschinen Heute



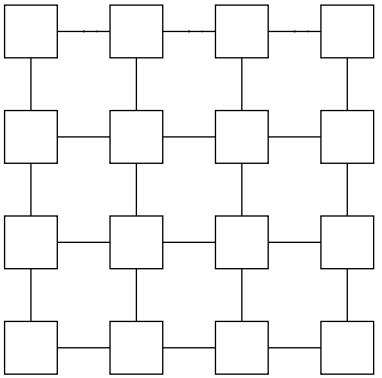
[Book]

Umgang mit komplexen Hierarchien

These: mit flachen Modellen, vor allem bei verteiltem Speicher kommen wir sehr weit.

- Entwerfe verteilt, implementiere hierarchieangepaßt
- Shared-Memory Unterprogramme auf Knoten

Explizites „Store-and-Forward“

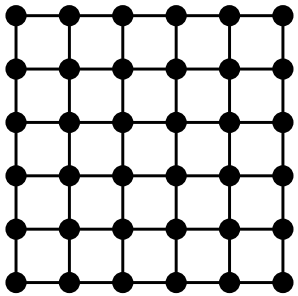


- Wir kennen die Struktur des Verbindungsgraphen ($V = \{1,\ldots,p\}, E \subseteq V \times V$). Varianten:
  - $V = \{1,\ldots,p\} \cup R$ mit zusätzlichen „dummen“ Routerknoten (ggf. mit Pufferspeicher).
  - Busse $\rightarrow$ Hyperedges
- Zu jeder Zeiteinheit kann jede Kante maximal k' Datenpakete konstanter Länge transportieren (meist $k' = 1$)
- In einer k-Port-Maschine kann jeder Knoten k Pakete gleichzeitig senden oder empfangen. $k = 1$ nennt sich single-ported.

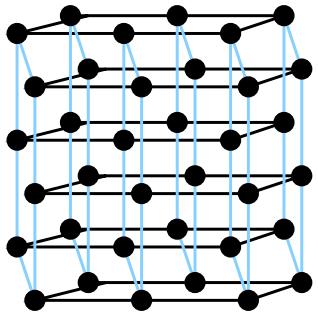
Diskussion

+ einfach formuliert
− low level $\Rightarrow$ „messy algorithms“
− Hardwarerouter erlauben schnelle Komm. wann immer ein Kommunikationspfad gefunden wird.

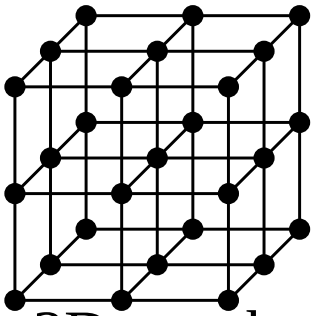
Typische Verbindungsnetzwerke



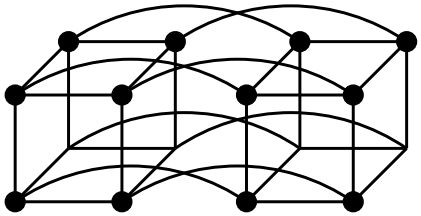
mesh



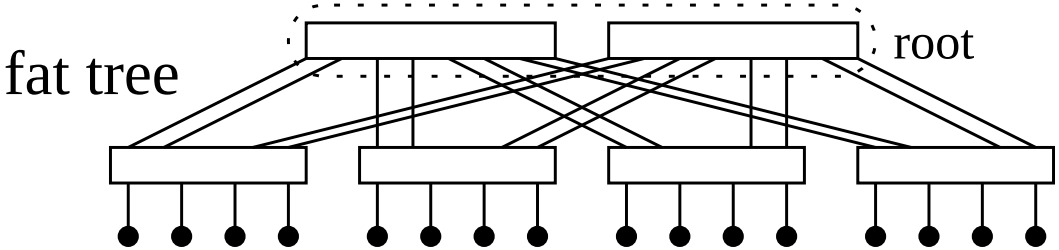
torus



3D−mesh



hypercube



[Book]

Vollständige Verknüpfung Punkt-zu-Punkt

- $E = V \times V$, single ported
- $T_{\text{comm}}(m) = \alpha + m\beta$. ($m =$ Nachrichtenl. in Maschinenwörtern)
- \+ Realistische Behandlung von Nachrichtenlängen
- \+ Viele Verbindungsnetze approximieren vollständige Verknüpfung
  $\Rightarrow$ sinnvolle Abstraktion
- \+ Keine überlasteten Kanten $\rightarrow$ OK für Hardwarerouter
- \+ „künstliches“ Vergrößern v. α, β
  $\rightarrow$ OK für „schwächliche“ Netzwerke
- \+ Asynchrones Modell
- − Etwas Händewedeln bei realen Netzwerken

Vollständige Verknüpfung: Varianten

Was tut PE i in Zeit $T_{\text{comm}}(m) = \alpha + m\beta$?

Nachrichtenlänge m.

halbduplex: $1\times$senden oder $1\times$empfangen (auch simplex)

Telefon: $1\times$senden an PE j und $1\times$empfangen von PE j

(voll)duplex: $1\times$senden und $1\times$empfangen.

Beliebige Kommunikationspartner

Auswirkung auf Laufzeit:

$$T^{\text{duplex}} \leq T^{\text{Telefon}} \leq T^{\text{duplex/2}} \leq 3T^{\text{duplex}}$$

BSP Bulk Synchronous Parallel

[McColl LNCS Band 1000, S. 46]

Maschine wir duch drei Parameter beschrieben: p, L und g.

L: Startup overhead für einen kollektiven Nachrichtenaustausch – an dem alle PEs beteiligt sind

g: gap$\approx \frac{\text{Rechengeschwindigkeit}}{\text{Kommunikationsbandbreite}}$

Superstep: Lokal arbeiten dann kollektiver global synchronisierter Austausch beliebiger Nachrichten.

w: max. lokale Arbeit (Taktzyklen)

h: max. number of machine words die ein PE sendet oder empfängt (h-relation)

Zeitaufwand: $w + L + gh$

BSP versus Point-to-Point

Mit naiver direkter Nachrichtenauslieferung:

Sei $H =$ max #Nachrichten eines PEs.

Dann $T \geq \alpha(H + \log p) + h\beta$.

Worst case $H = h$. Also $L \geq \alpha \log p$ and $g \geq \alpha$?

Mittels all-to-all und direkter Nachrichtenauslieferung:

Dann $T \geq \alpha p + h\beta$.

Also $L \geq \alpha p$ and $g \approx \beta$?

Mittels all-to-all und indirekter Nachrichtenauslieferung:

Dann $T = \Omega(\log p(\alpha + h\beta))$.

Also $L = \Omega(\alpha \log p)$ and $g = \Omega(\beta \log p)$?

BSP*

Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model,
Armin Bäumker and Friedhelm Meyer auf der Heide, ESA 1995.

Neudefinition von h zu # blocks der Größe B, z.B. $B = \Theta(\alpha/\beta)$.

Sei M_i Menge der Nachrichten, die PE i sendet oder empfängt.
Sei $h = \max_i \sum_{m \in M_i} \lceil |m|/B \rceil$.
Sei g gap zwischen Paktetsendungen der Größe B.
Dann ist wieder

$$w + L + gh$$

die Zeit für einen Superstep.

BSP* versus Point-to-Point

Mit naiver direkter Nachrichtenauslieferung:

$L \approx \alpha \log p$

$g \approx B\beta$

BSP$^+$

Wir erweitern BSP so dass die kollektiven Operationen

broadcast

(all-)reduce

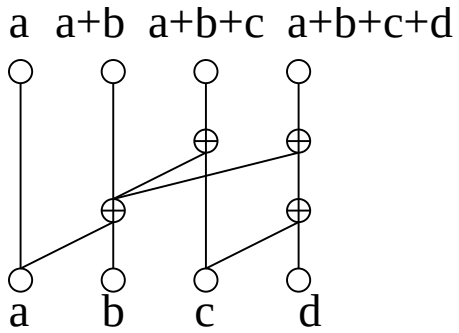
prefix-sum

mit Nachrichtenlänge h ebenfalls erlaubt sind.

Stay tuned für Algorithmen, die das rechtfertigen.

BSP*-Algorithmen sind bis zu einem Faktor $\Theta(\log p)$ langsamer als BSP$^+$-Algorithmen.

Graph- und Schaltkreisdarstellung v. Algorithmen



Viele Berechnungen können als

gerichteter azyklischer Graph dargestellt werden

- Eingabeknoten haben Eingangsgrad 0

  und eine feste Ausgabe

- Ausgabeknoten haben Ausgangsgrad 0

  und Eingangsgrad 1

- Der Eingangsgrad ist durch

  eine kleine Konstante beschränkt.

- Innere Knoten berechnen eine Funktion, die sich in konstanter Zeit berechnen läßt.

Schaltkreise

- Variante: Wenn statt Machinenworten, konstant viele bits verarbeitet werden spricht man von Schaltkreisen.
- Die Tiefe $d(S)$ des Berechnungs-DAG ist die Anzahl innerer Knoten auf dem längsten Pfad von einem Eingang zu einem Ausgang. Tiefe$\sim$Rechenzeit
- Wenn man für jede Eingabegröße (algorithmisch) einen Schaltkreis angibt, spricht man von Schaltkreisfamilien

Beispiel: Assoziative Operationen (=Reduktion)

Satz 1. *Sei $\oplus$ ein assoziativer Operator, der in konstanter Zeit berechnet werden kann. Dann läßt sich*

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

in Zeit $\mathcal{O}(\log n)$ auf einer PRAM berechnen und in Zeit $\mathcal{O}(\alpha \log n)$ auf einem linearen Array mit Hardwarerouter

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

Beweisskizze für $n = 2^k$ (oBdA?)

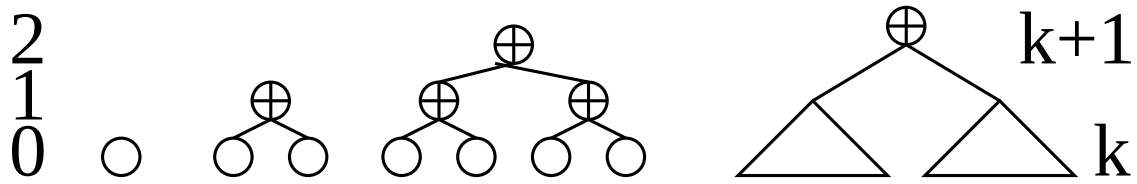
Induktionsannahme: $\exists$ Schaltkreis d. Tiefe k für $\bigoplus_{i<2^k} x_i$

$k = 0$: trivial

$k \rightsquigarrow k+1$:

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

■



PRAM Code

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

active := 1

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

  if active **then**

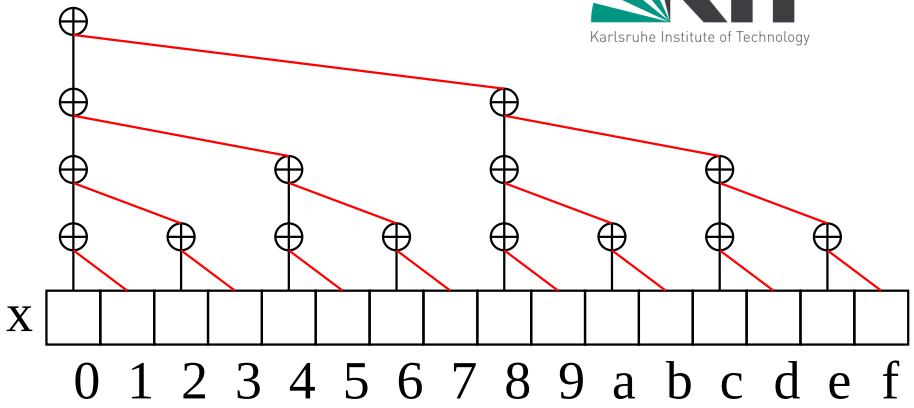
    if bit k of i **then**

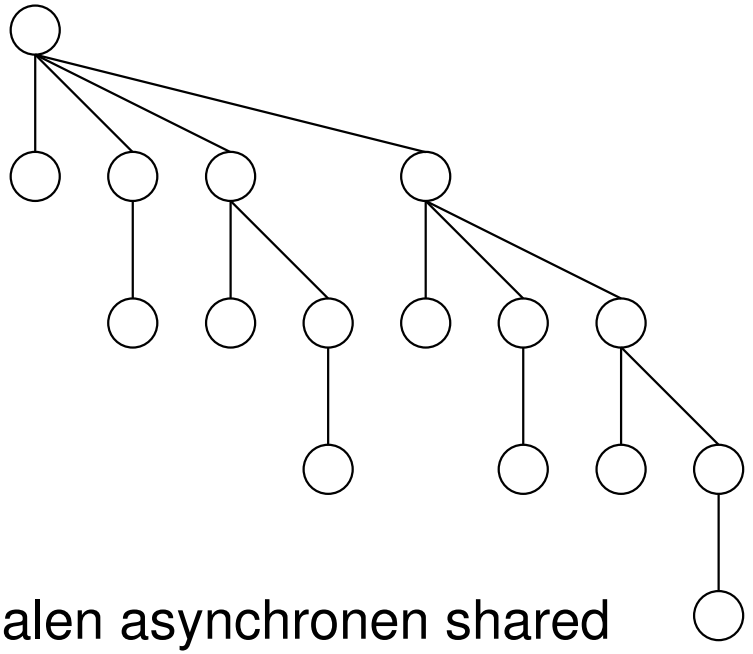
      active := 0

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

      $x_i := x_i \oplus x_{i+2^k}$

// result is in x_0





Vorsicht: Viel komplizierter auf einer realen asynchronen shared memory Maschine.

Speedup? Effizienz?

$\log x$ bei uns immer $\log_2 x$

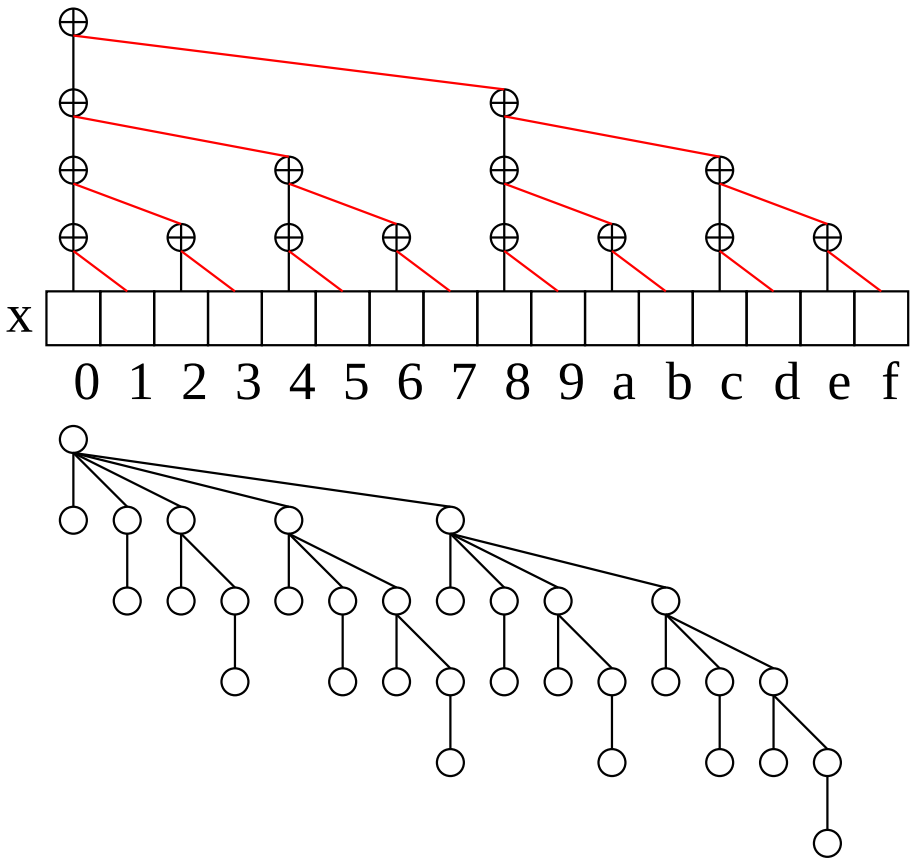
Analyse

n PEs

Zeit $\mathcal{O}(\log n)$

Speedup $\mathcal{O}(n/\log n)$

Effizienz $\mathcal{O}(1/\log n)$



Weniger ist Mehr (Brent's Prinzip)

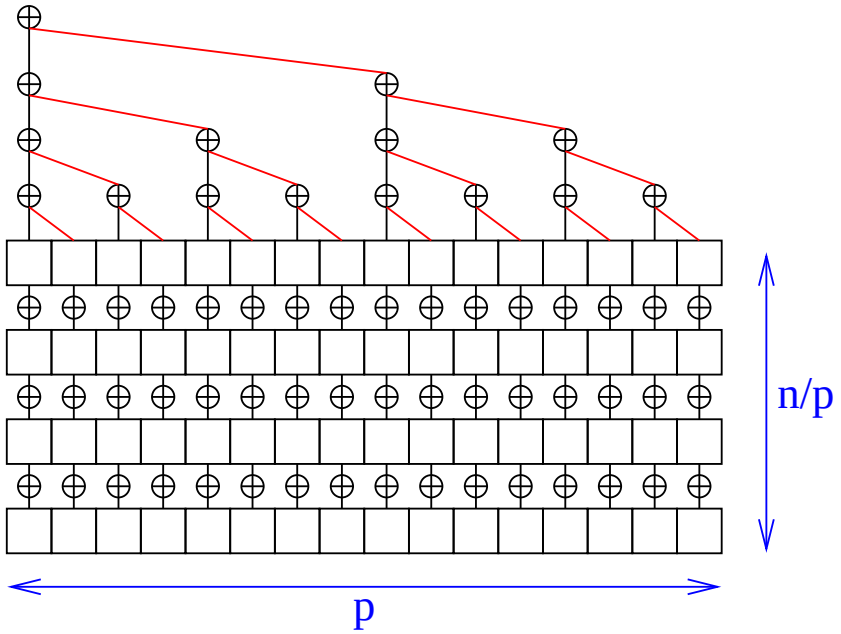
p PEs

Jedes PE addiert

n/p Elemente sequentiell

Dann parallele Summe

für p Teilsummen



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

Effizienz

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

falls $n \gg p\log p$

Distributed Memory Machine

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

//Input x_i located on PE i

active := 1

$s := x_i$

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

  if active **then**

    if bit k of i **then**

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

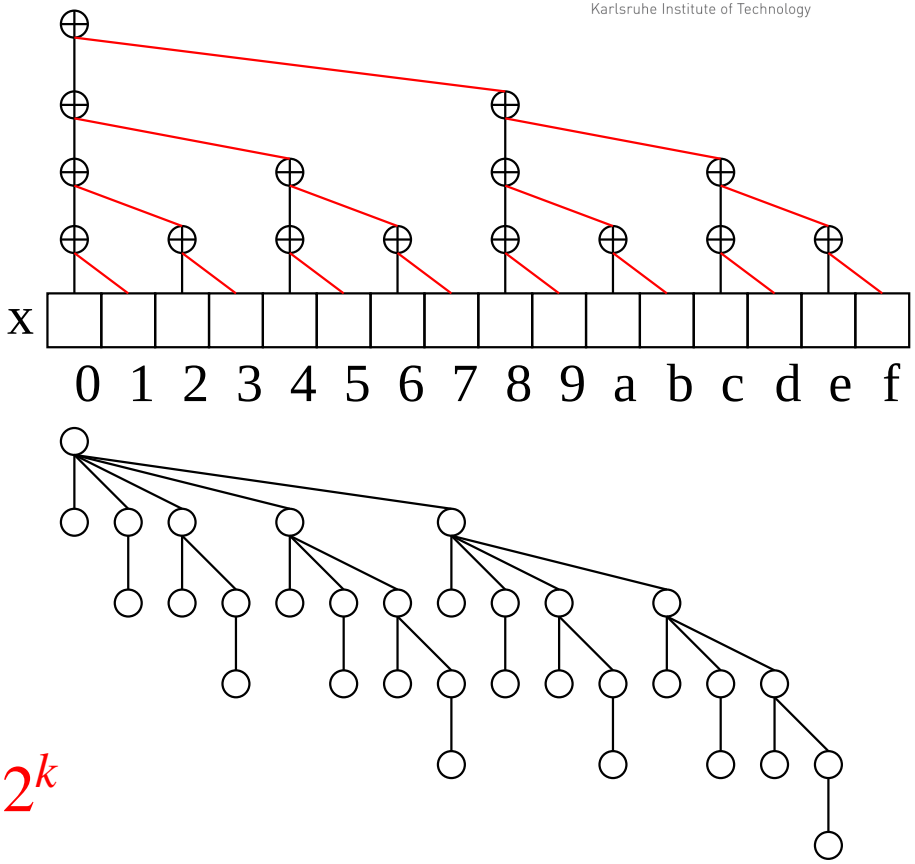
      active := 0

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

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

      $s := s \oplus s'$

//result is in s on PE 0



Analyse

vollständige Verknüpfung: $\Theta((\alpha+\beta)\log p)$

lineares Array: $\Theta(p)$: Schritt k braucht Zeit 2^k.

lineares Array mit Router: $\Theta((\alpha+\beta)\log p)$, weil edge congestion (Kantenlast) in jedem Schritt eins ist.

BSP $\Theta((l+g)\log p) = \Omega\left(\log^2 p\right)$

Beliebiges $n > p$: jeweils zusätzliche Zeit $T_{\text{seq}}(n/p)$

Diskussion Reduktionsoperation

- Binärbaum führt zu logarithmischer Ausführungszeit
- Nützlich auf den meisten Modellen
- Brent's Prinzip: Ineffiziente Algorithmen werden durch Verringerung der Prozessorzahl effizient
- Später: Reduktion komplexer Objekte. Zum Beispiel Vektoren, Matrizen

Matrixmultiplikation

Gegeben: Matrizen $A \in \mathbf{R}^{n\times n}$, $B \in \mathbf{R}^{n\times n}$

mit $A = ((a_{ij}))$ und $B = ((b_{ij}))$

$\mathbf{R}$: Halbring

$C = ((c_{ij})) = A \cdot B$ bekanntlich gemäß:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Arbeit: $\Theta\left(n^3\right)$ arithmetische Operationen

(bessere Algorithmen falls in R Subtraktion möglich)

Ein erster PRAM Algorithmus

n^3 PEs

for i:= 1 **to** n **dopar**

  for j:= 1 **to** n **dopar**

    $c_{ij} := \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$ // n PE parallel sum

Ein PE für jedes Teilprodukt $c_{ikj} := a_{ik} b_{kj}$

Zeit $\mathcal{O}(\log n)$

Effizienz $\mathcal{O}(1/\log n)$

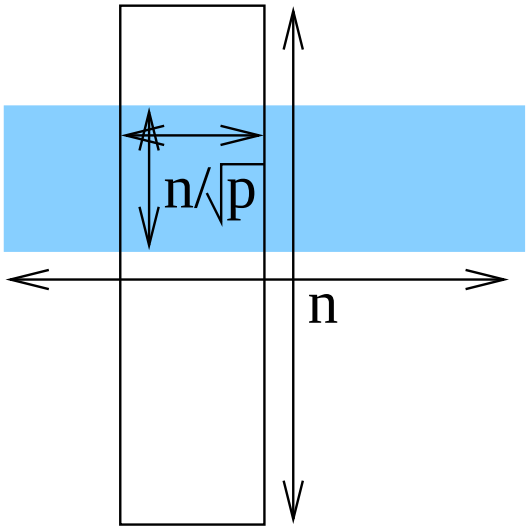
Verteilte Implementierung I

$p \leq n^2$ PEs

for i:= 1 **to** n **dopar**

for j:= 1 **to** n **dopar**

$$c_{ij} := \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$



Teile jedem PE n^2/p der c_{ij} zu

- Begrenzte Skalierbarkeit
- Hohes Kommunikationsvolumen. Zeit $\Omega\left(\beta \frac{n^2}{\sqrt{p}}\right)$

Verteilte Implementierung II-1

[Dekel Nassimi Sahni 81, KGGK Section 5.4.4]

Sei $p = N^3$, n ein Vielfaches von N

Fasse A, B, C als $N \times N$ Matrizen auf,

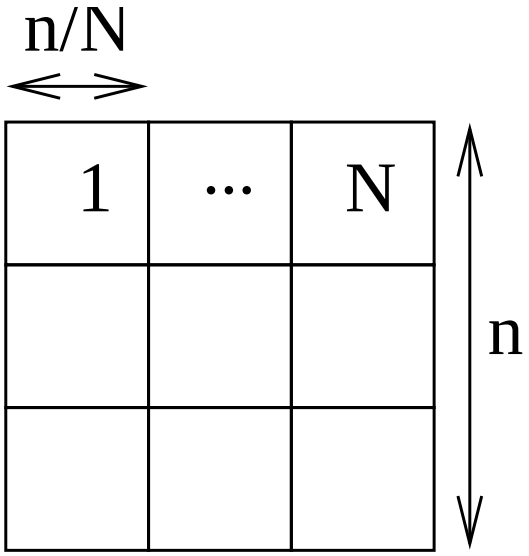
jedes Element ist $n/N \times n/N$ Matrix

for i:= 1 **to** N **dopar**

for j:= 1 **to** N **dopar**

$$c_{ij} := \sum_{k=1}^{N} a_{ik} b_{kj}$$

Ein PE für jedes Teilprodukt $c_{ikj} := a_{ik} b_{kj}$



Verteilte Implementierung II-2

store a_{ik} in PE $(i,k,1)$

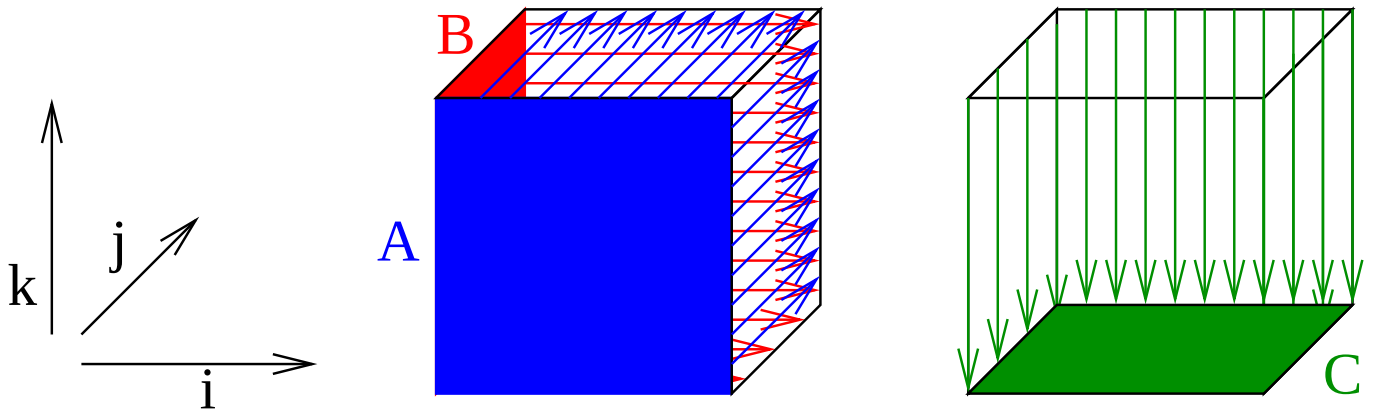
store b_{kj} in PE $(1,k,j)$

PE $(i,k,1)$ broadcasts a_{ik} to PEs (i,k,j) for $j \in \{1..N\}$

PE $(1,k,j)$ broadcasts b_{kj} to PEs (i,k,j) for $i \in \{1..N\}$

compute $c_{ikj} := a_{ik} b_{kj}$ on PE (i,k,j) // local!

PEs (i,k,j) for $k \in \{1..N\}$ compute $c_{ij} := \sum_{k=1}^{N} c_{ikj}$ to PE $(i,1,j)$



Analyse, Fully Connected u.v.a.m.

store a_{ik} in PE $(i,k,1)$ // free (or cheap)

store b_{kj} in PE $(1,k,j)$ // free (or cheap)

PE $(i,k,1)$ broadcasts a_{ik} to PEs (i,k,j) for $j \in \{1..N\}$

PE $(1,k,j)$ broadcasts b_{kj} to PEs (i,k,j) for $i \in \{1..N\}$

compute $c_{ikj} := a_{ik} b_{kj}$ on PE (i,k,j) // $T_{\text{seq}}(n/N) = \mathcal{O}\left((n/N)^3\right)$

PEs (i,k,j) for $k \in \{1..N\}$ compute $c_{ij} := \sum_{k=1}^{N} c_{ikj}$ to PE $(i,1,j)$

Kommunikation:

$$2T_{\text{broadcast}}(\overbrace{(n/N)^2}^{\text{Obj. size}}, \overbrace{N}^{\text{PEs}}) + T_{\text{reduce}}((n/N)^2, N) \approx 3T_{\text{broadcast}}((n/N)^2, N)$$

$$\overset{N=p^{1/3}}{\rightsquigarrow} \cdots \mathcal{O}\left(\frac{n^3}{p} + \beta\frac{n^2}{p^{2/3}} + \alpha \log p\right)$$

Diskussion Matrixmultiplikation

- PRAM Alg. ist guter Ausgangspunkt
- DNS Algorithmus spart Kommunikation braucht aber Faktor $\Theta\left(\sqrt[3]{p}\right)$ mehr Platz als andere Algorithmen

$\rightsquigarrow$ gut für kleine Matrizen (bei grossen ist Kommunikation eh egal)

- Pattern für vollbesetzte lineare Algebra:

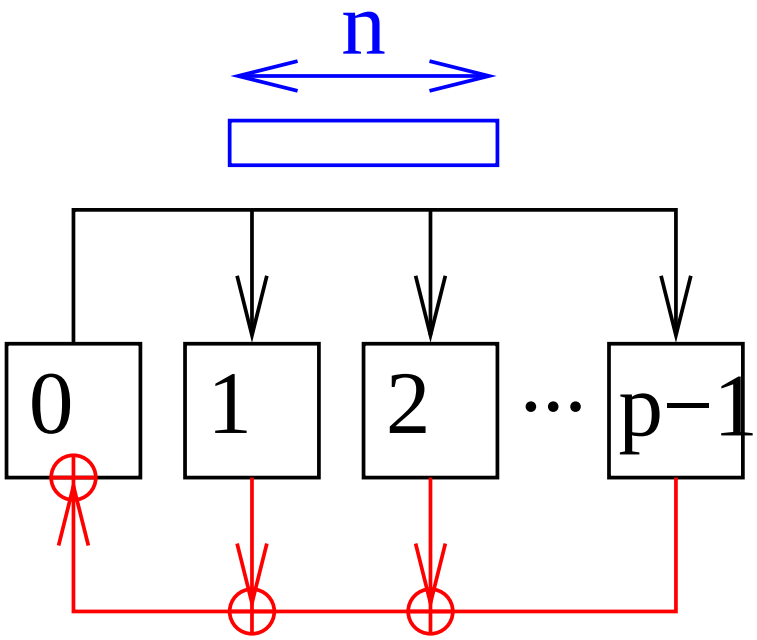
  Lokale Ops auf Teilmatrizen $+$ Broadcast $+$ Reduce

  z.B. Matrix-Vektor-Produkt, LGS lösen,...

Broadcast (Rundruf?) und Reduktion

Broadcast: Einer für alle

Ein PE (z.B. 0) schickt Nachricht der Länge n an alle



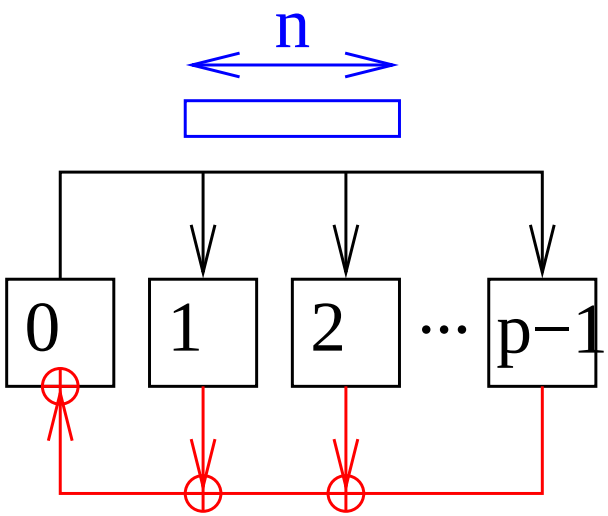
Reduktion: Alle für einen

Ein PE (z.B. 0) empfängt Summe v. p Nachrichten der Länge n

(Vektoraddition$\neq$ lokale Addition!)

Broadcast $\rightsquigarrow$ Reduktion

- Kommunikationsrichtung umdrehen
- Korrespondierende Teile ankommender und eigener Nachrichten addieren



Alle folgenden Broadcastalgorithmen ergeben Reduktionsalgoithmen für kommutative und assoziative Operationen.

Die meisten (ausser Johnsson/Ho und speziellen Einbettungen) funktionieren auch bei nichtkommutativen Operationen.

Modellannahmen

- fully connected
- vollduplex – paralleles Senden und Empfangen

Varianten: halbduplex also senden oder empfangen, BSP, Einbettung in konkrete Netzwerke

Naiver Broadcast [KGGK Abschnitt 3.2.1]

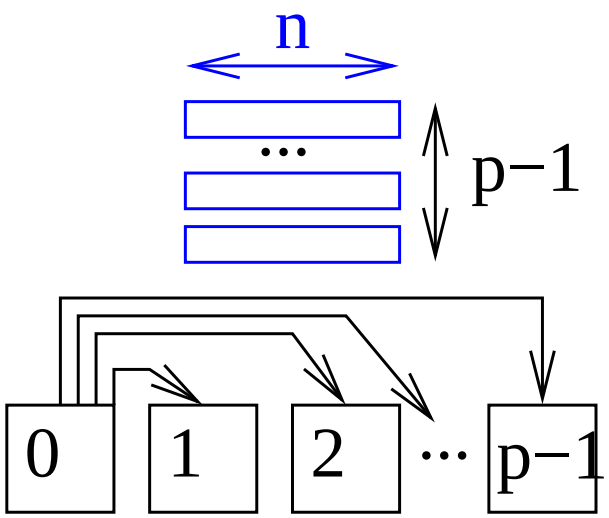
Procedure naiveBroadcast$(m[1..n])$

PE 0: **for** $i := 1$ **to** $p-1$ **do** send m to PE i

PE $i > 0$: receive m

Zeit: $(p-1)(n\beta+\alpha)$

Alptraum bei der Implementierung skalierbarer Algorithmen



Binomialbaum-Broadcast

Procedure binomialTreeBroadcast$(m[1..n])$

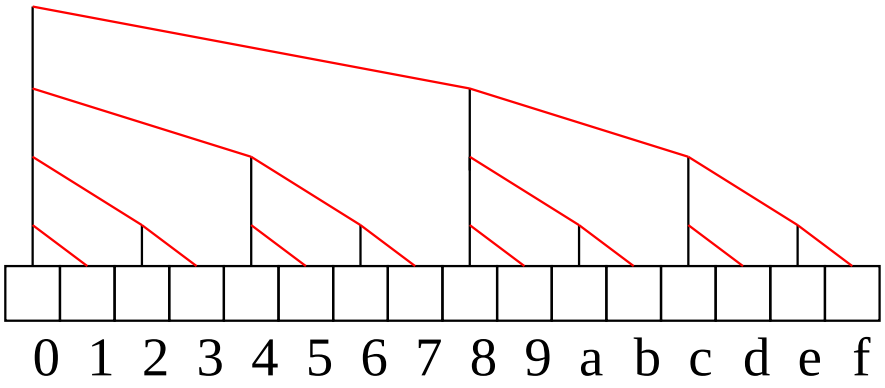
PE index $i \in \{0,\ldots,p-1\}$

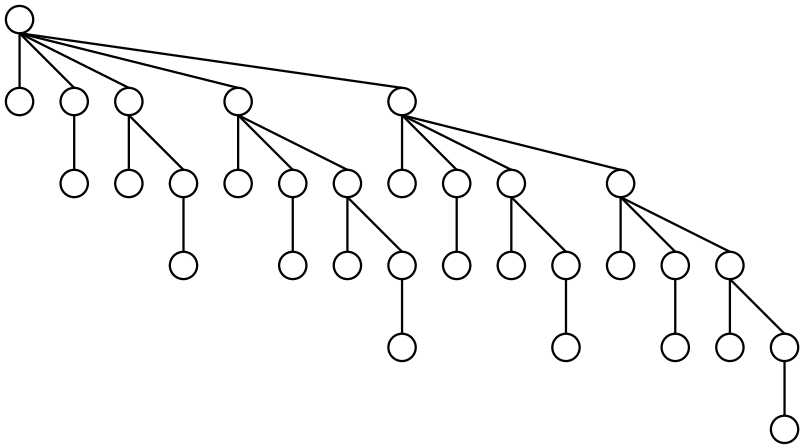
//Message m located on PE 0

if $i > 0$ **then** receive m

for $k := \min\{\lceil \log n \rceil, \text{trailingZeroes}(i)\} - 1$ **downto** 0 **do**

send m to PE $i + 2^k$ // noop if receiver $\geq p$

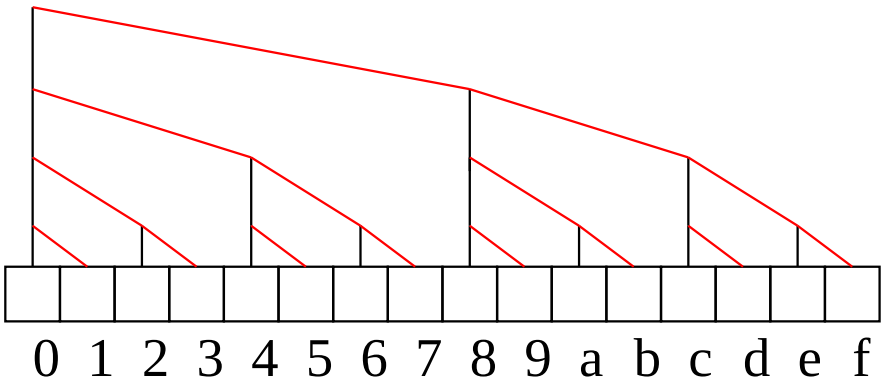


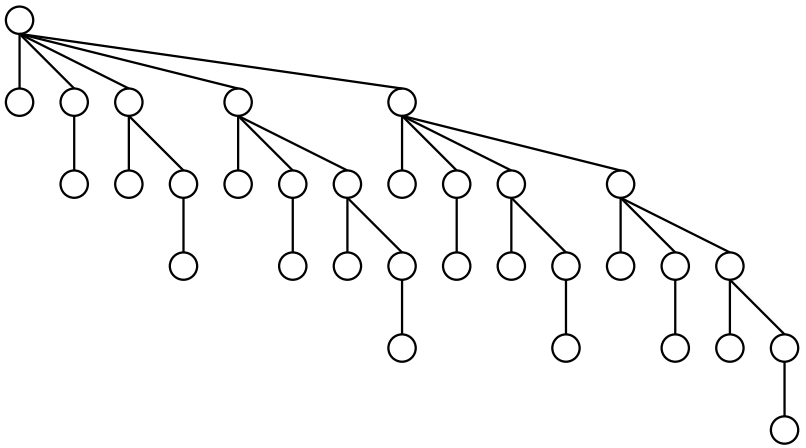


Analyse

- Zeit: $\lceil \log p \rceil \, (n\beta + \alpha)$
- Optimal für $n = 1$
- Einbettbar in lineares Gitter

$n \cdot f(p) \rightsquigarrow n + \log p$?





Lineare Pipeline

Procedure linearPipelineBroadcast$(m[1..n], k)$

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

// Message m located on PE 0

// assume k divides n

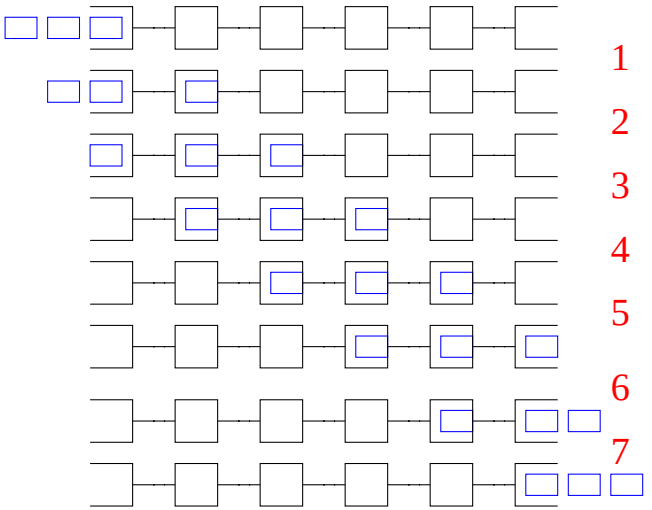
define piece j as $m[(j-1)\frac{n}{k}+1..j\frac{n}{k}]$

for $j := 1$ **to** $k+1$ **do**

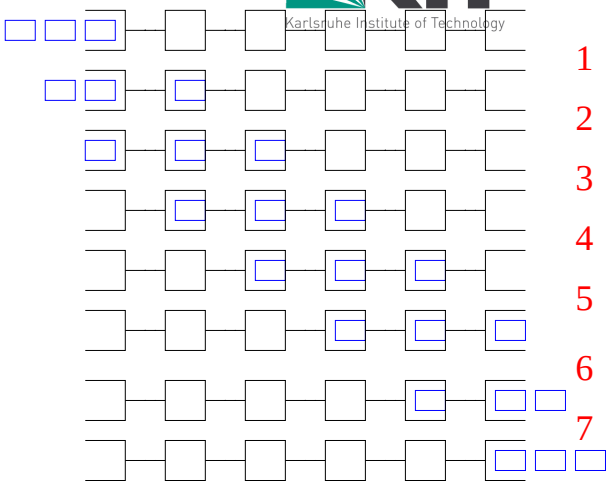
receive piece j from PE $i-1$ // noop if $i=0$ or $j=k+1$

and, concurrently,

send piece $j-1$ to PE $i+1$ // noop if $i=p-1$ or $j=1$



Analyse

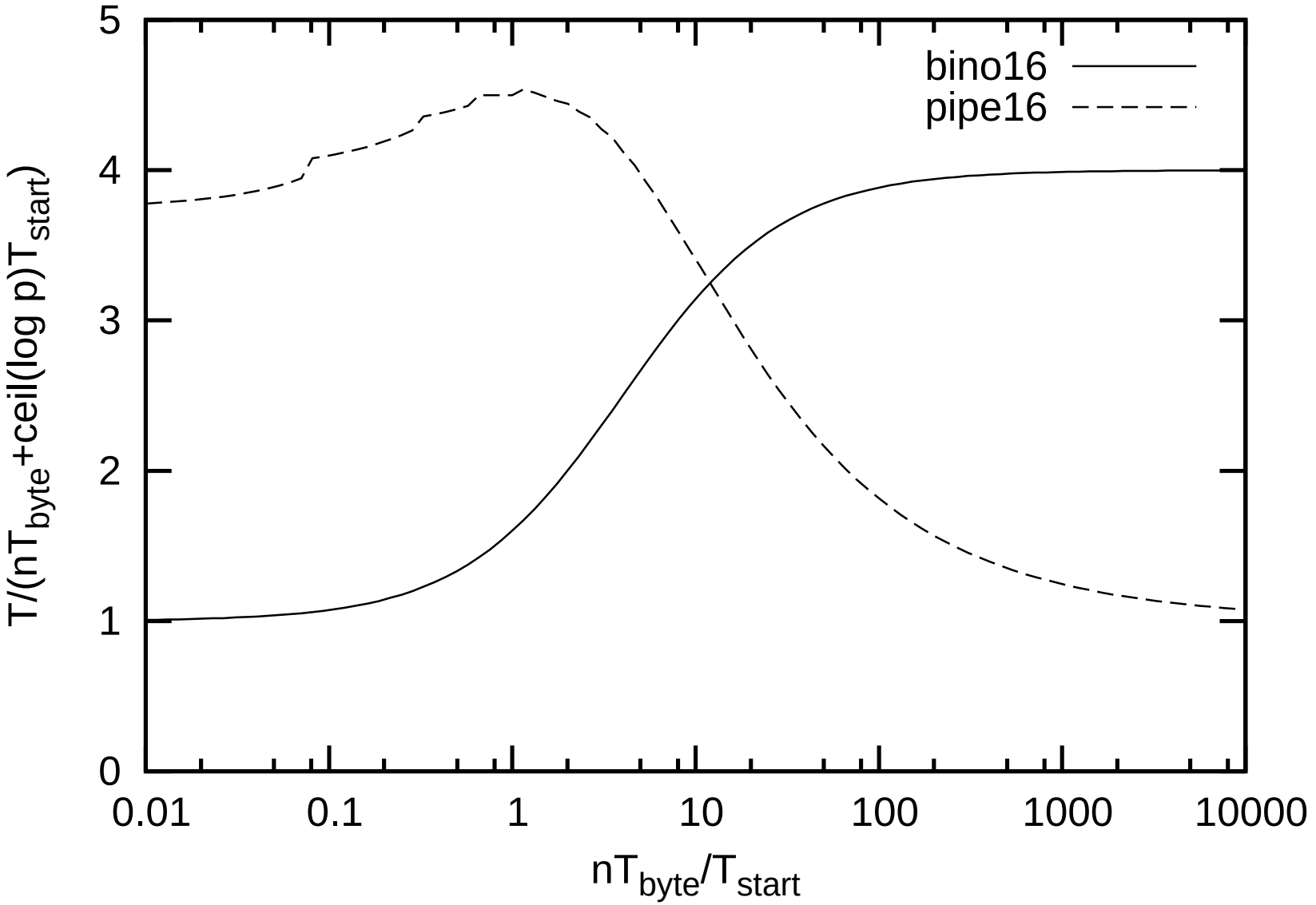


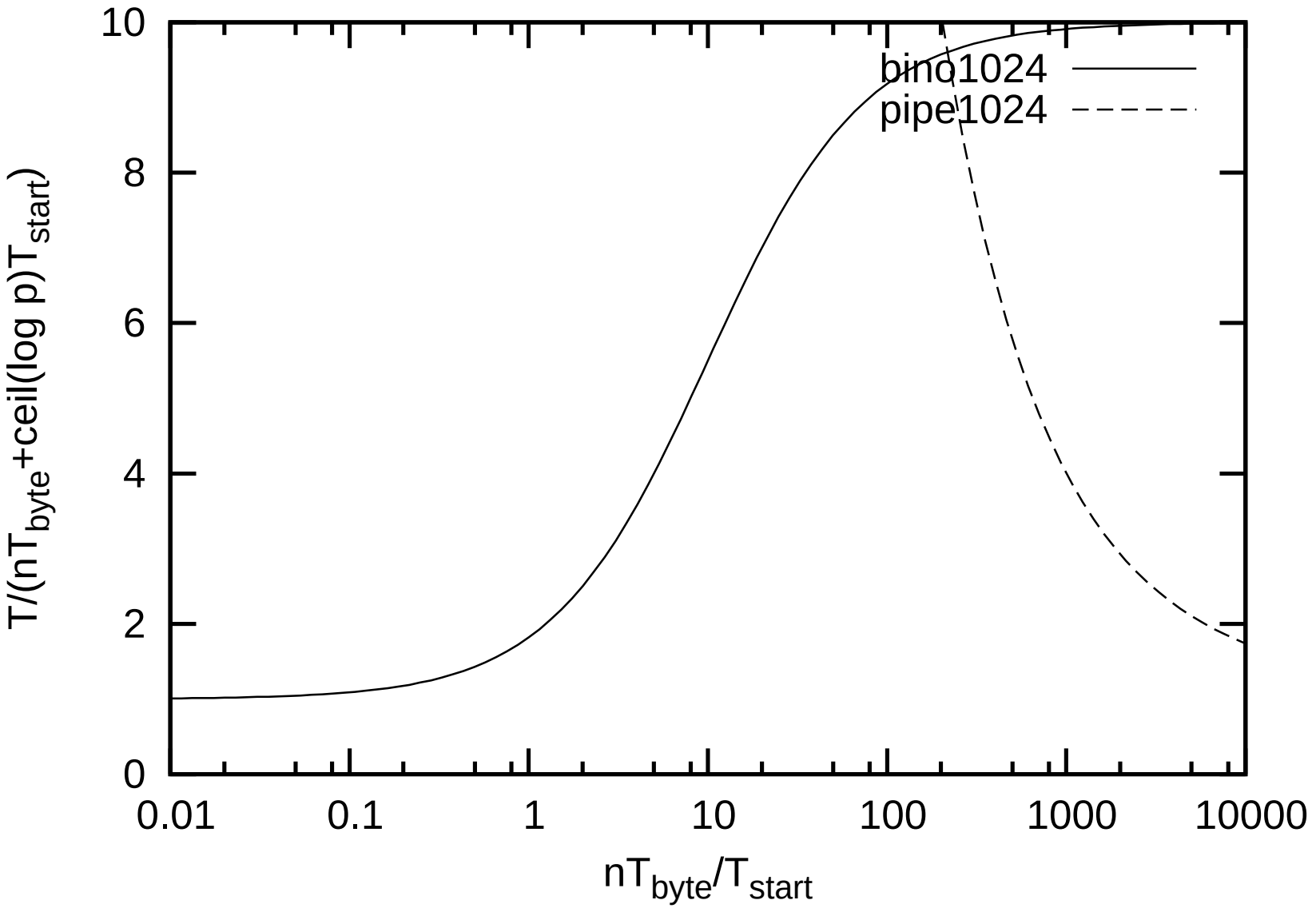
- Zeit $\frac{n}{k}\beta + \alpha$ pro Schritt
  ($\neq$ Iteration)
- $p-1$ Schritte bis erstes Paket ankommt
- Dann 1 Schritte pro weiteres Paket

$T(n,p,k)$: $\left(\frac{n}{k}\beta + \alpha\right)(p+k-2))$

optimales k: $\sqrt{\frac{n(p-2)\beta}{\alpha}}$

$T^*(n,p)$: $\approx n\beta + p\alpha + 2\sqrt{np\alpha\beta}$


bino16
pipe16
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
0
1
2
3
4
5
0.01
0.1
1
10
100
1000
10000


10
8
6
4
2
0
0.01
0.1
1
10
100
1000
10000
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
bino1024
pipe1024

Diskussion

- Lineares Pipelining ist optimal für festes p und $n \to \infty$
- Aber für großes p braucht man extrem grosse Nachrichten

$\alpha p \rightsquigarrow \alpha \log p$?

Procedure binaryTreePipelinedBroadcast$(m[1..n], k)$

// Message m located on root, assume k divides n

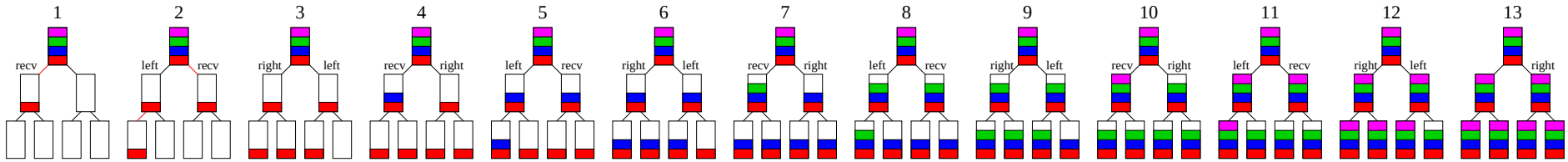
define piece j as $m[(j-1)\frac{n}{k}+1..j\frac{n}{k}]$

for $j := 1$ **to** k **do**

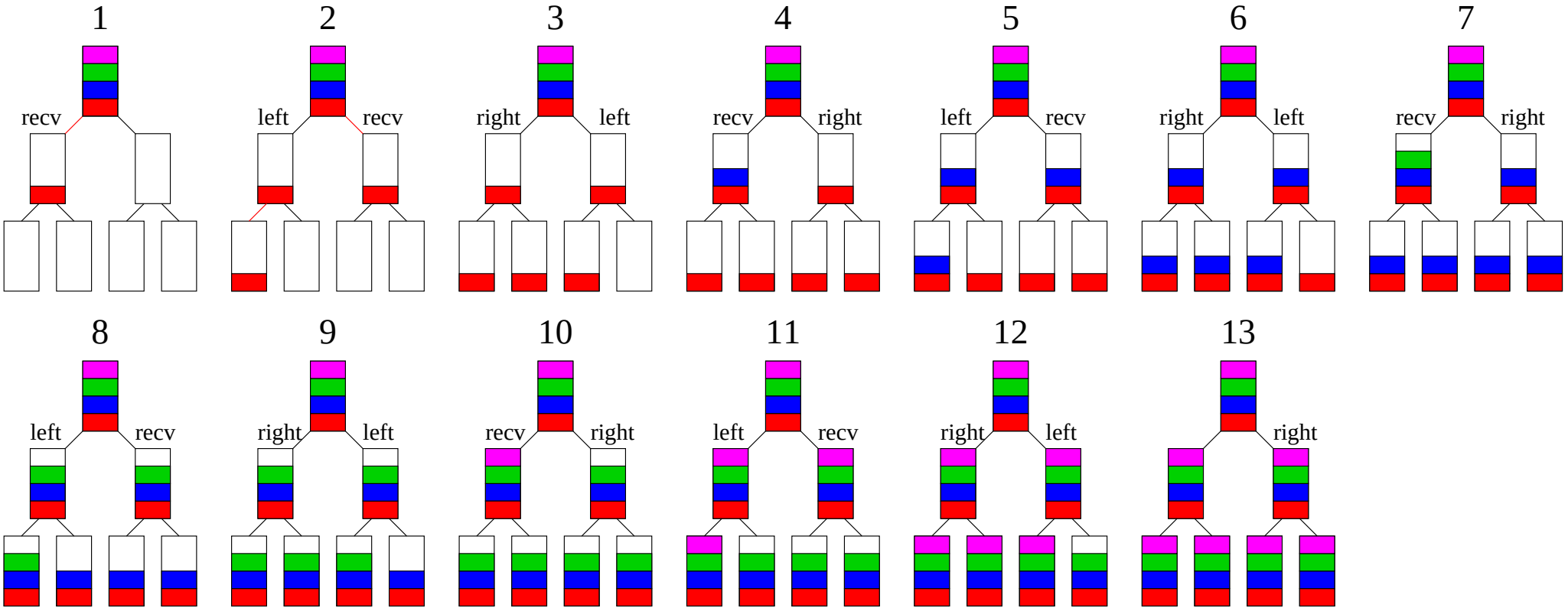
if parent exists **then** receive piece j

if left child ℓ exists **then** send piece j to ℓ

if right child r exists **then** send piece j to r



Beispiel


1
recv
2
left
recv
3
right
left
4
recv
right
5
left
recv
6
right
left
7
recv
right
8
left
recv
9
right
left
10
recv
right
11
left
recv
12
right
left
13
right

Analyse

- Zeit $\frac{n}{k}\beta + \alpha$ pro Schritt ($\neq$ Iteration)
- $2j$ Schritte bis erstes Paket Schicht j erreicht
- Wieviele Schichten? $d := \lfloor \log p \rfloor$
- Dann 3 Schritte pro weiteres Paket

Insgesamt: $T(n,p,k) := (2d + 3(k-1)) \left(\frac{n}{k}\beta + \alpha\right)$

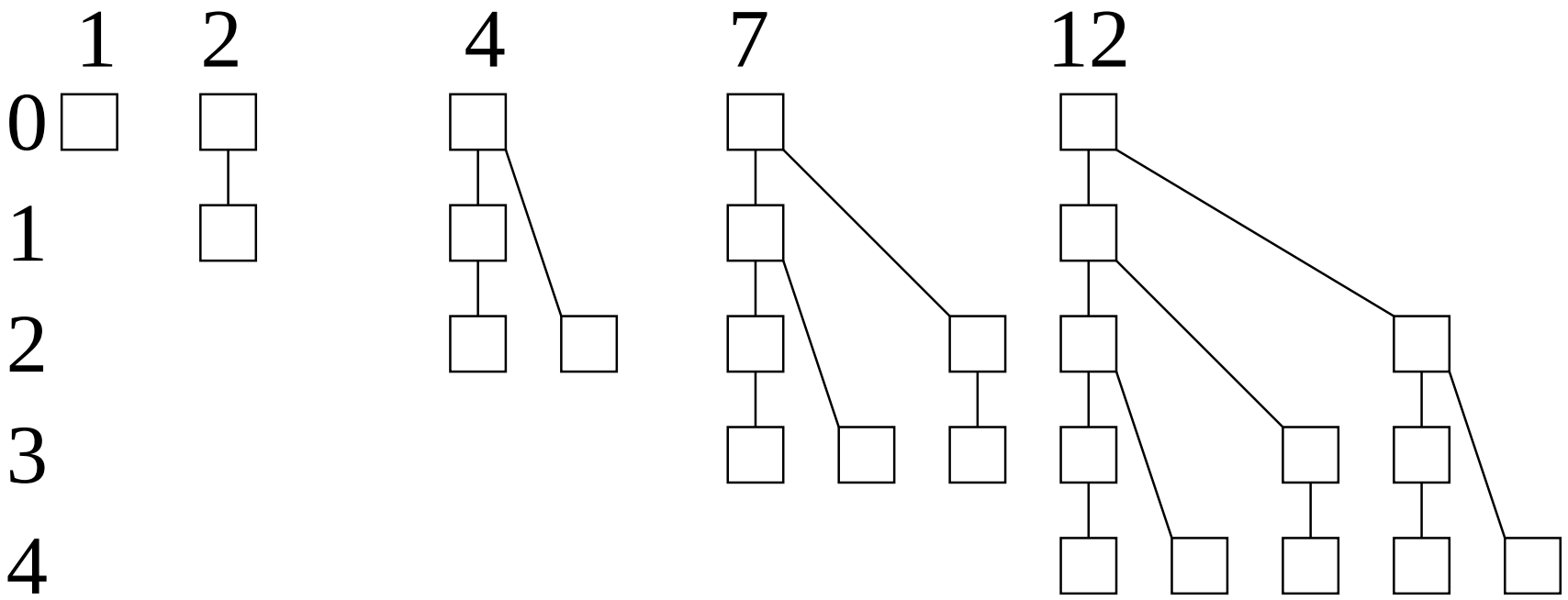
optimales k: $\sqrt{\frac{n(2d-3)\beta}{3\alpha}}$

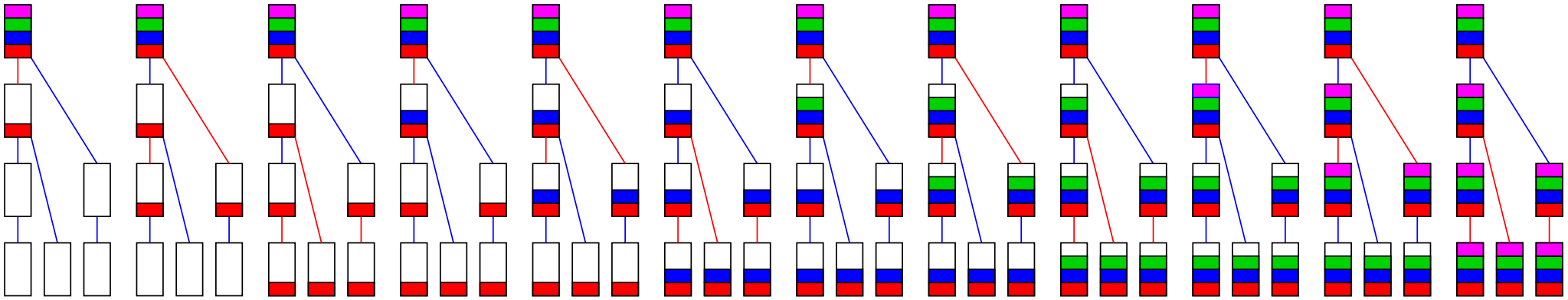
Analyse

- Zeit $\frac{n}{k}\beta + \alpha$ pro Schritt ($\neq$ Iteration)
- $d := \lfloor \log p \rfloor$ Schichten
- Insgesamt: $T(n,p,k) := (2d + 3(k-1))\left(\frac{n}{k}\beta + \alpha\right)$
- optimales k: $\sqrt{\frac{n(2d-3)\beta}{3\alpha}}$

eingesetzt: $T^*(n,p) = 2d\alpha + 3n\beta + \mathcal{O}\left(\sqrt{nd\alpha\beta}\right)$

Fibonacci-Bäume





active connection passive connection

Analyse

- Zeit $\frac{n}{k}\beta + \alpha$ pro Schritt ($\neq$ Iteration)
- j Schritte bis erstes Paket Schicht j erreicht
- Wieviele PEs p_j mit Schicht $0..j$?

  $p_0 = 1, p_1 = 2, p_j = p_{j-2} + p_{j-1} + 1$ $\rightsquigarrow$ask Maple,

  ```
  rsolve(p(0)=1,p(1)=2,p(i)=p(i-2)+p(i-1)+1,p(i));
  ```

  $$p_j \approx \frac{3\sqrt{5}+5}{5(\sqrt{5}-1)}\Phi^j \approx 1.89\Phi^j$$

  mit $\Phi = \frac{1+\sqrt{5}}{2}$ (goldener Schnitt)

$\rightsquigarrow$ $d \approx \log_\Phi p$ Schichten

insgesamt: $T^*(n,p) = d\alpha + 3n\beta + \mathcal{O}\left(\sqrt{nd\alpha\beta}\right)$

Procedure fullDuplexBinaryTreePipelinedBroadcast$(m[1..n], k)$

//Message m located on root, assume k divides n

define piece j as $m[(j-1)\frac{n}{k}+1..j\frac{n}{k}]$

for $j := 1$ **to** $k+1$ **do**

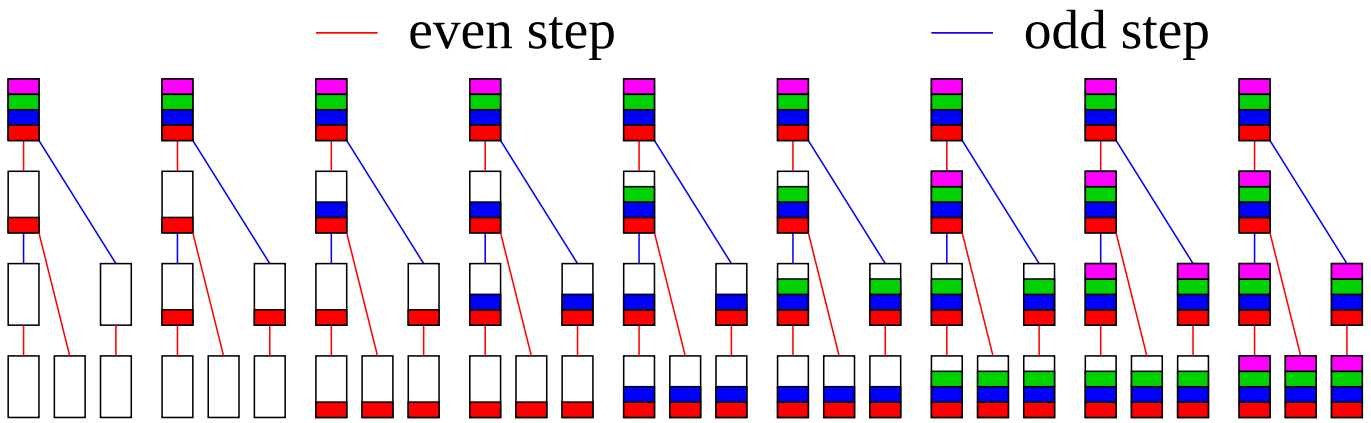
receive piece j from parent // noop for root or $j = k+1$

and, concurrently, send piece $j-1$ to right child

// noop if no such child or $j = 1$

send piece j to left child

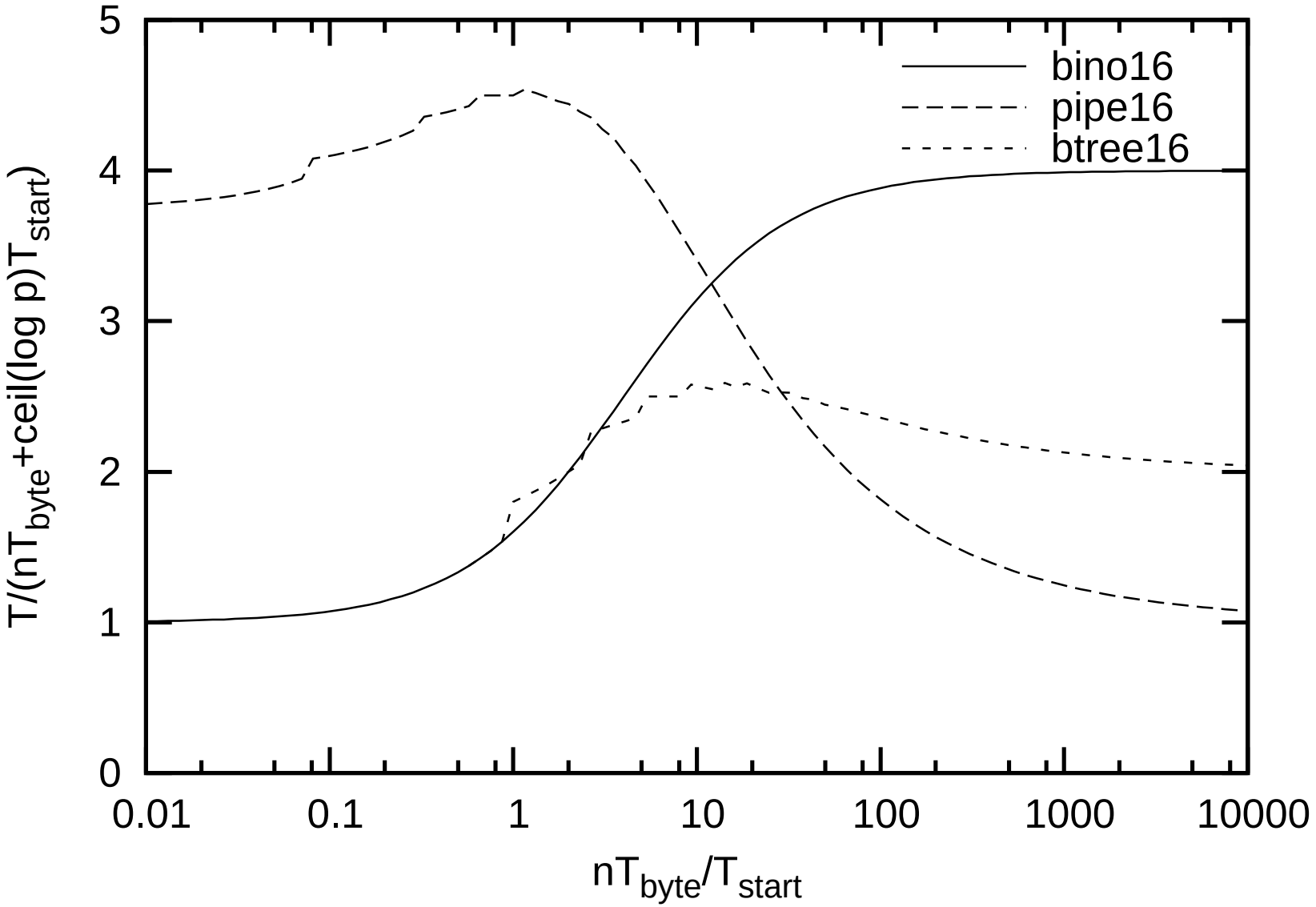
// noop if no such child or $j = k+1$

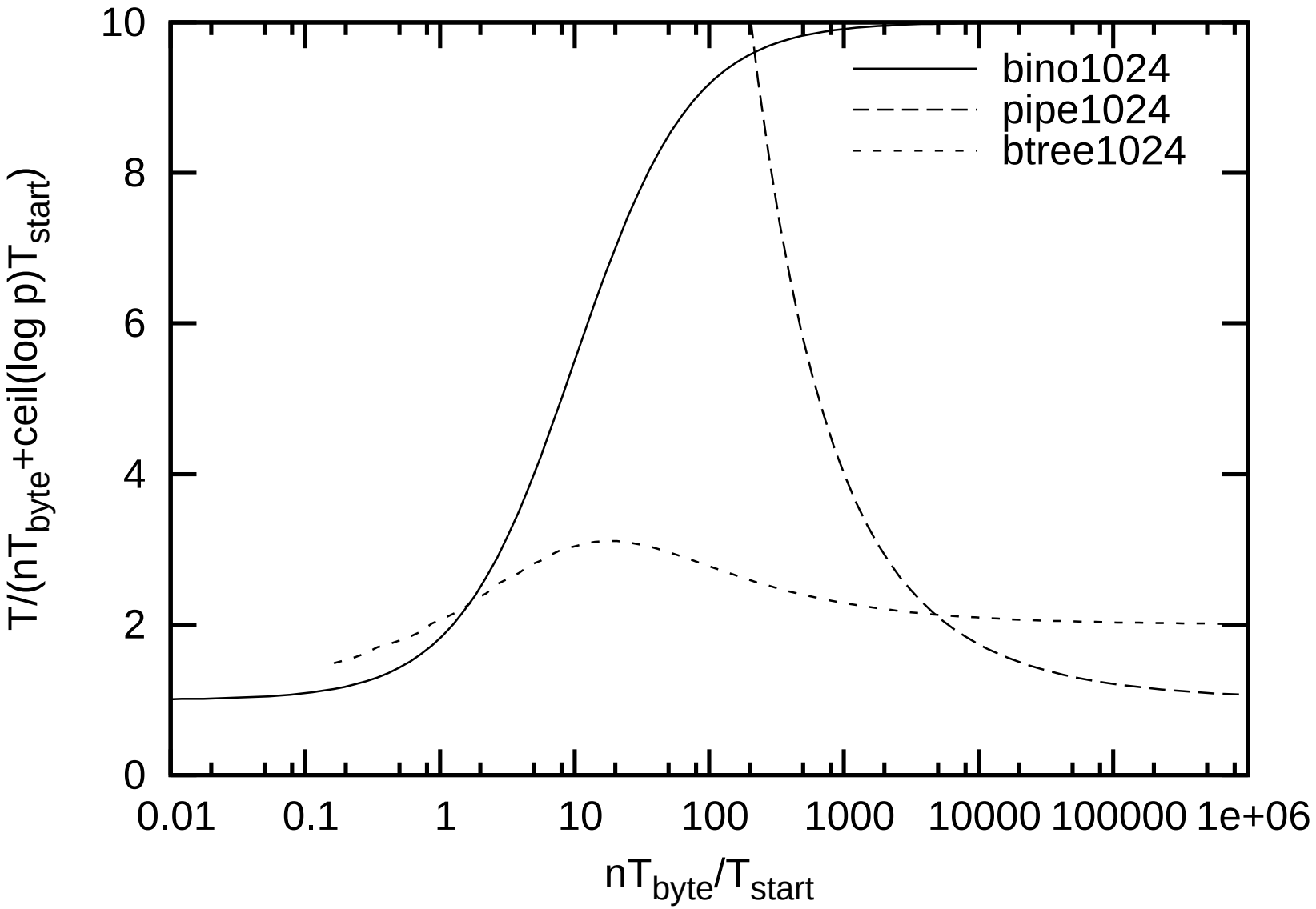


Analyse

- Zeit $\frac{n}{k}\beta + \alpha$ pro Schritt
- j Schritte bis erstes Paket Schicht j erreicht
- $d \approx \log_\Phi p$ Schichten
- Dann 2 Schritte pro weiteres Paket

insgesamt: $T^*(n,p) = d\alpha + 2n\beta + \mathcal{O}\left(\sqrt{nd\alpha\beta}\right)$


bino16
pipe16
btree16
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
0
1
2
3
4
5
0.01
0.1
1
10
100
1000
10000


bino1024
pipe1024
btree1024
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
0
2
4
6
8
10
0.01
0.1
1
10
100
1000
10000
100000
1e+06

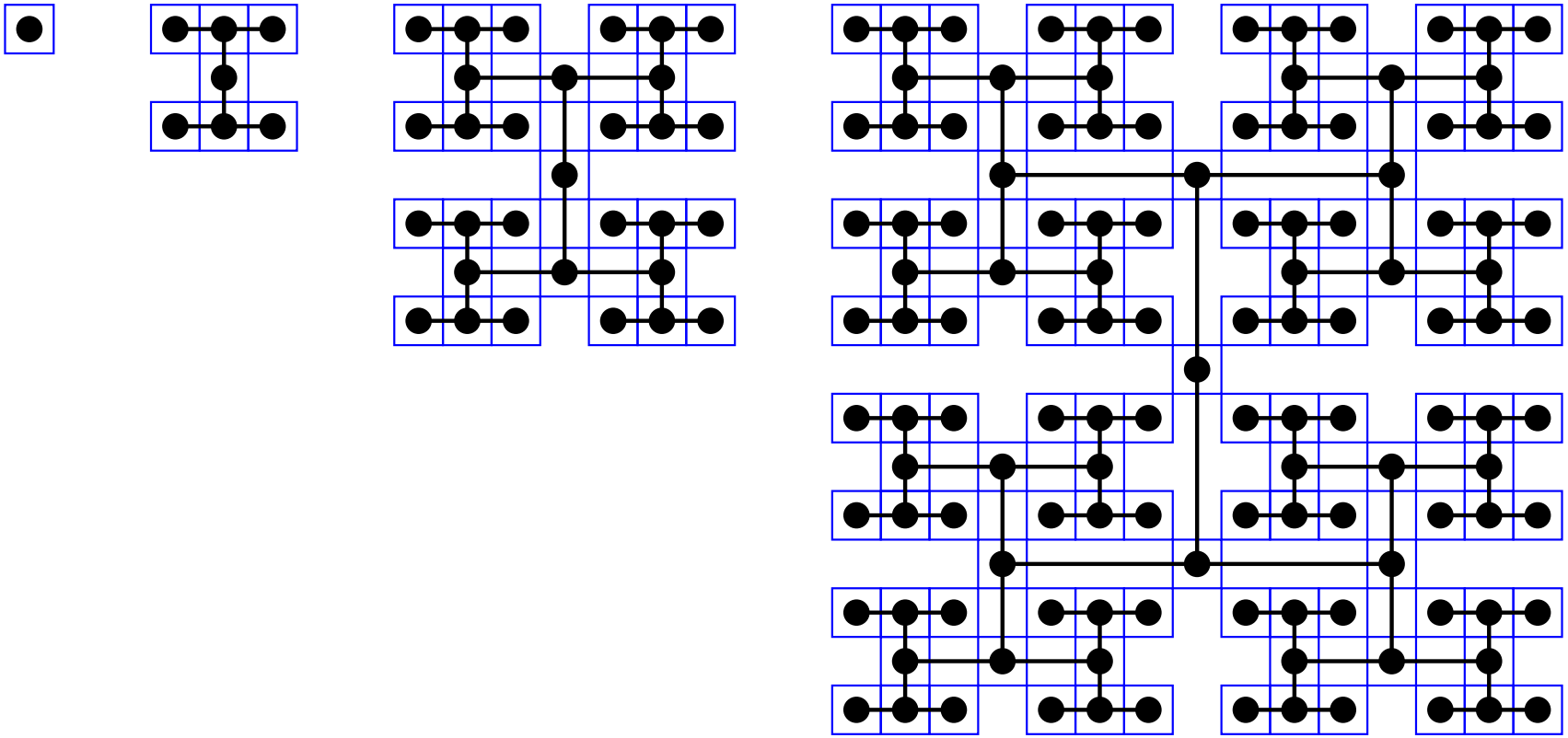
Diskussion

Fibonacci trees sind ein guter Kompromiss für alle n, p.

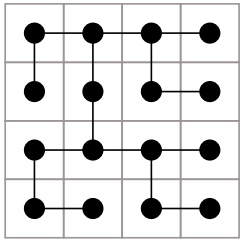
Allgemeine p:

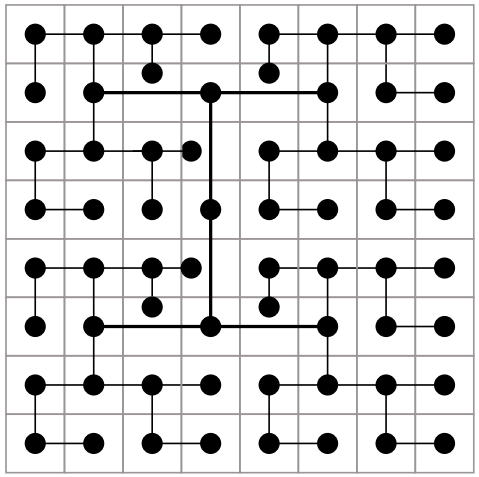
nächstgößeren Baum nehmen und dann Teilbaum weglassen.

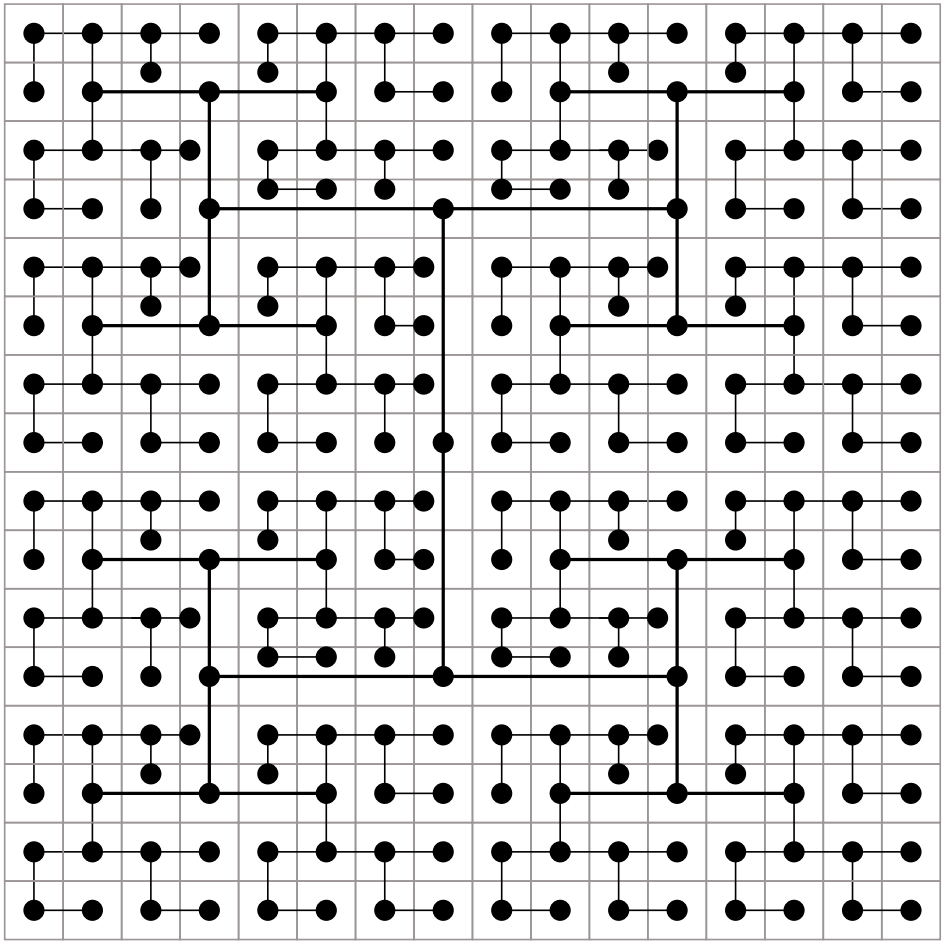
H-Trees



H-Trees







Nachteile baumbasierter Broadcasts

- Blätter empfangen nur Ihre Daten

  und tragen sonst nichts zur Verbreitung der Daten bei

- Innere Knoten senden mehr als sie empfangen

  $\rightsquigarrow$ full-duplex Kommunikation nicht ausgereizt

23-Broadcast: Two T(h)rees for the Price of One

- Binary-Tree-Broadcasts über zwei Bäume A und B gleichzeitig
- Innere Knoten von A sind

  Blätter von B

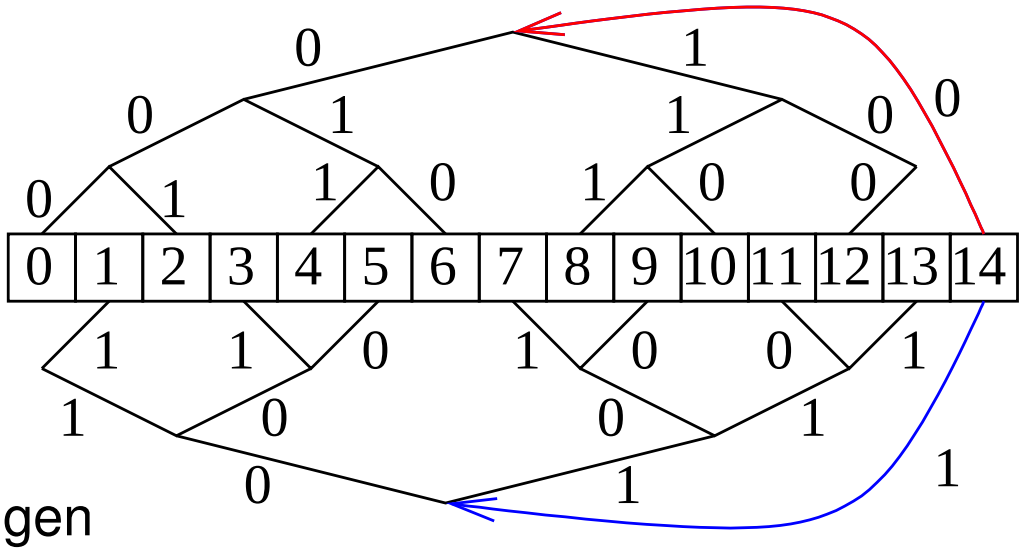
  und umgekehrt
- Pro Doppelschritt:

  Ein Paket als Blatt empfangen +

  Ein Paket als innerer Knoten

  empfangen und weiterleiten.

  d.h. 2 Pakete senden und empfangen

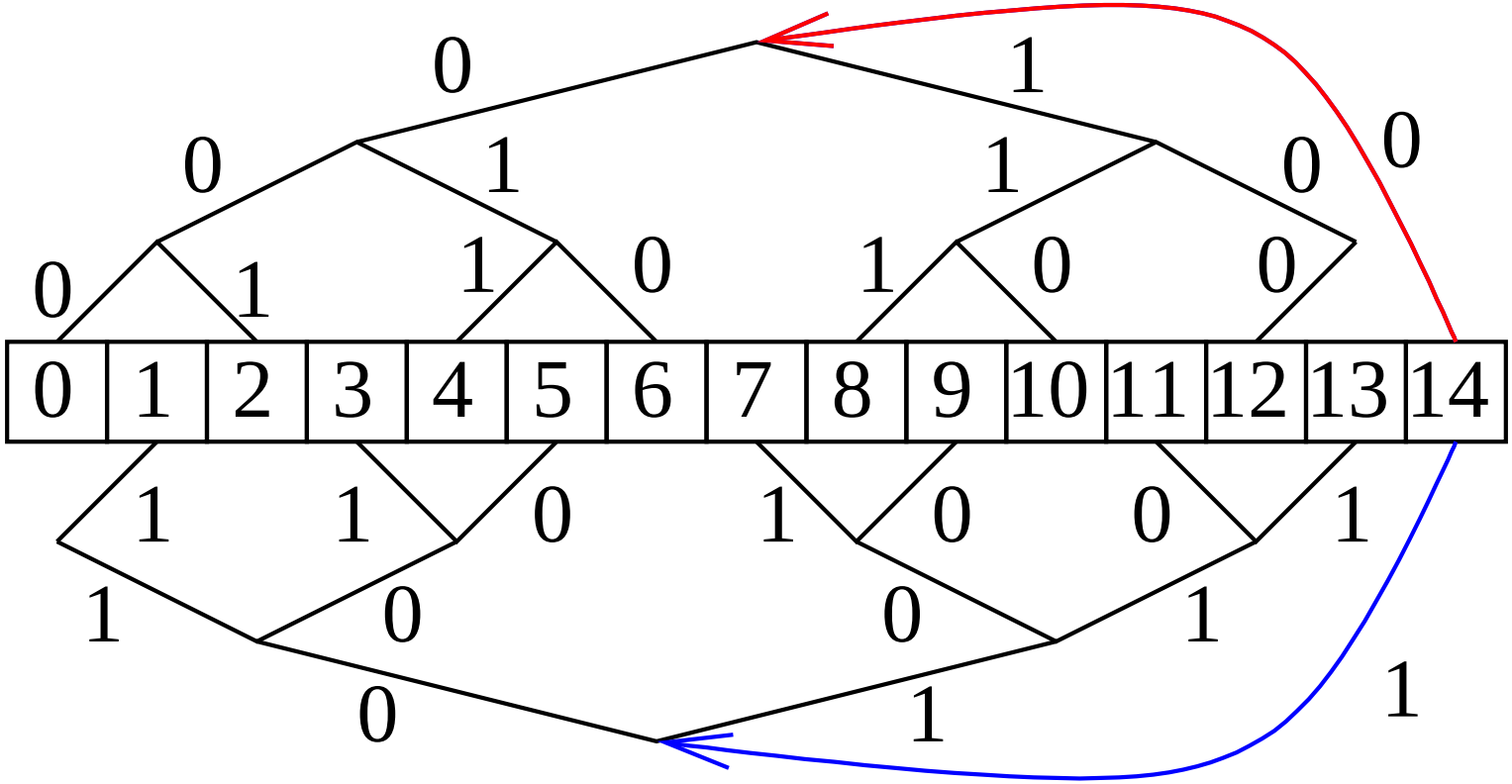


Root Process

for $j := 1$ **to** k **step** 2 **do**

send piece $j+0$ along edge labelled 0

send piece $j+1$ along edge labelled 1



Other Processes,

Wait for first piece to arrive

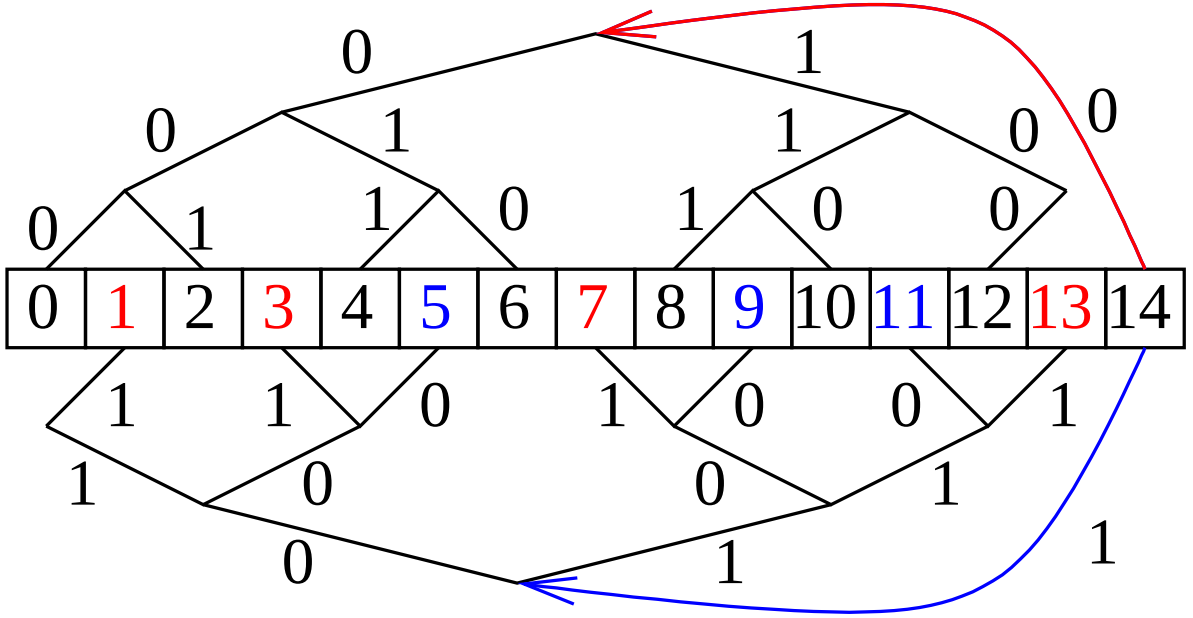
if it comes from the upper tree over an edge labelled b **then**

$\Delta := 2 \cdot$ distance of the node from the bottom in the upper tree

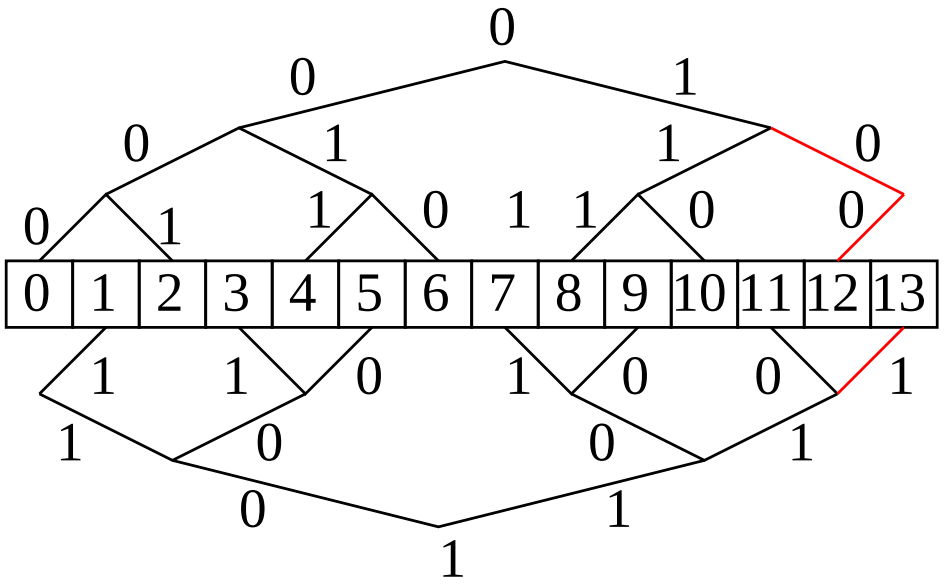
for $j := 1$ **to** $k + \Delta$ **step** 2 **do**

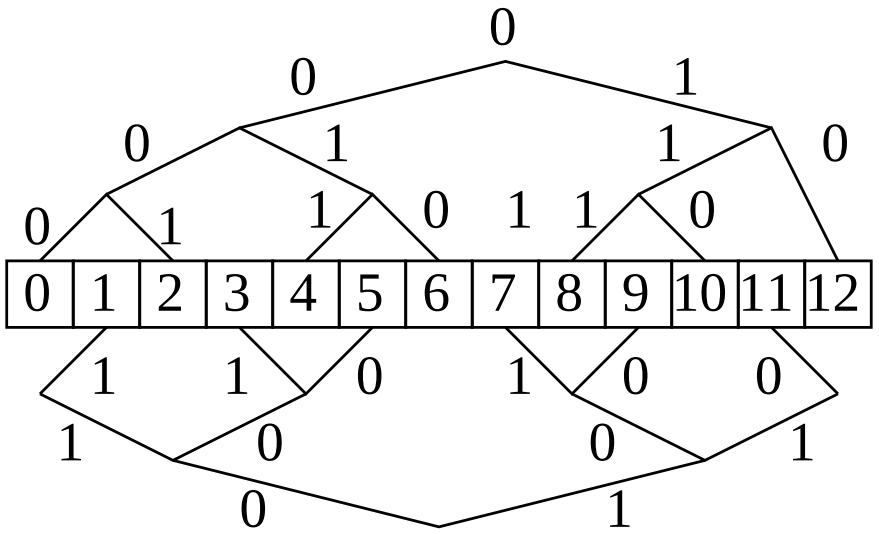
along b-edges: receive piece j and send piece $j - 2$

along $1 - b$-edges: receive piece $j + 1 - \Delta$ and send piece j

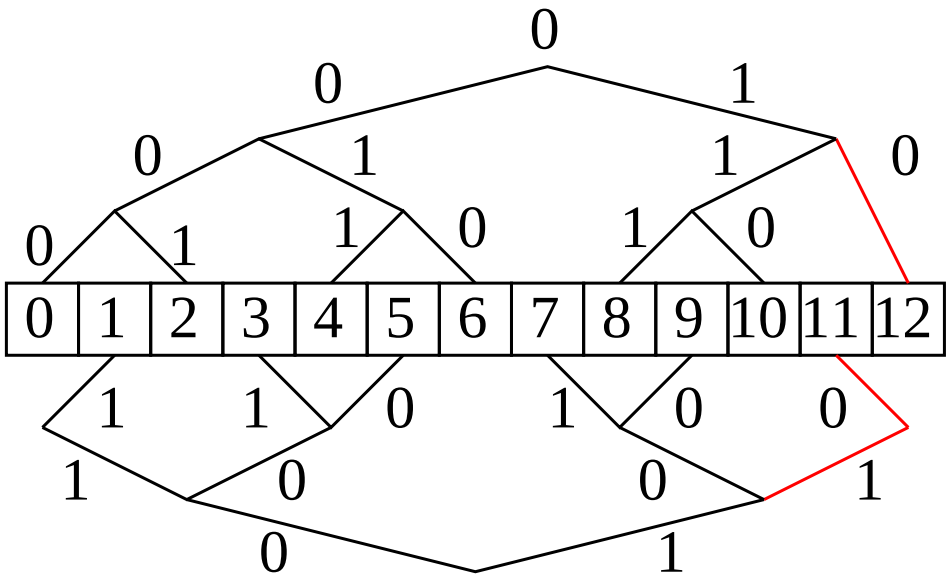


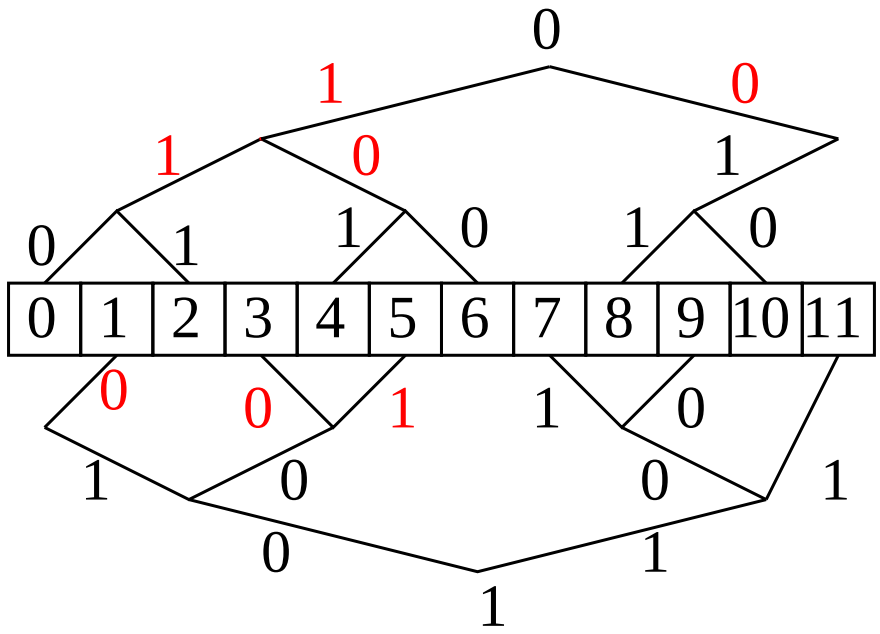
Beliebige Prozessorzahl



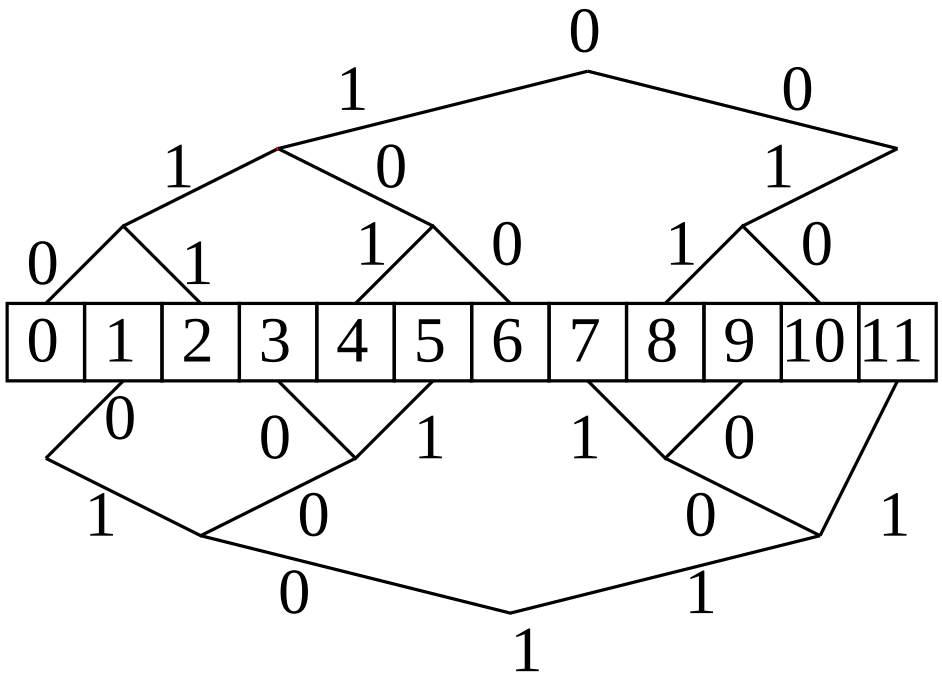


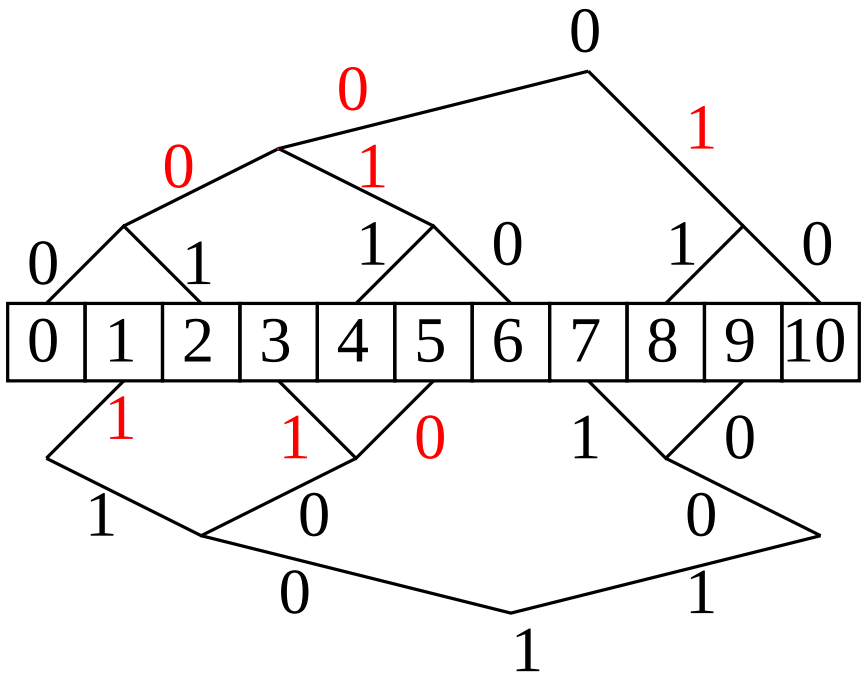
Beliebige Prozessorzahl



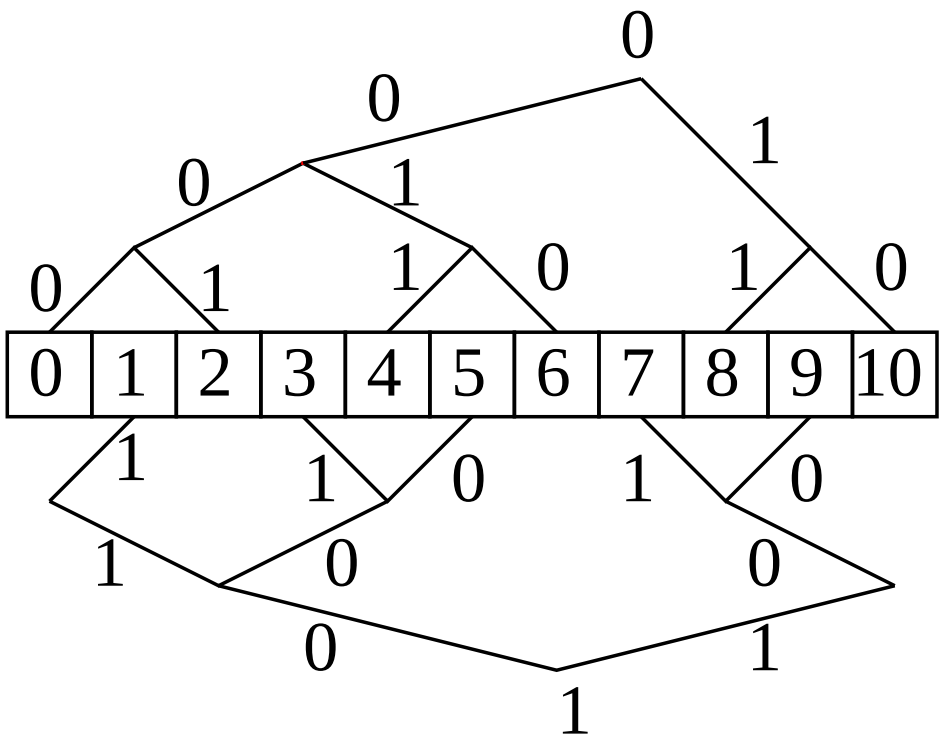


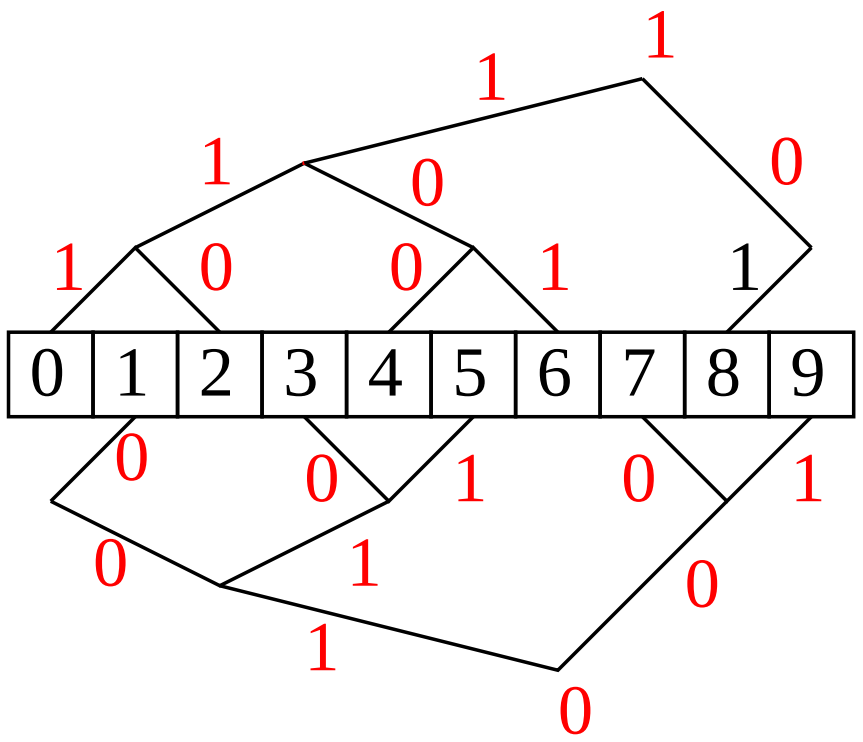
Beliebige Prozessorzahl



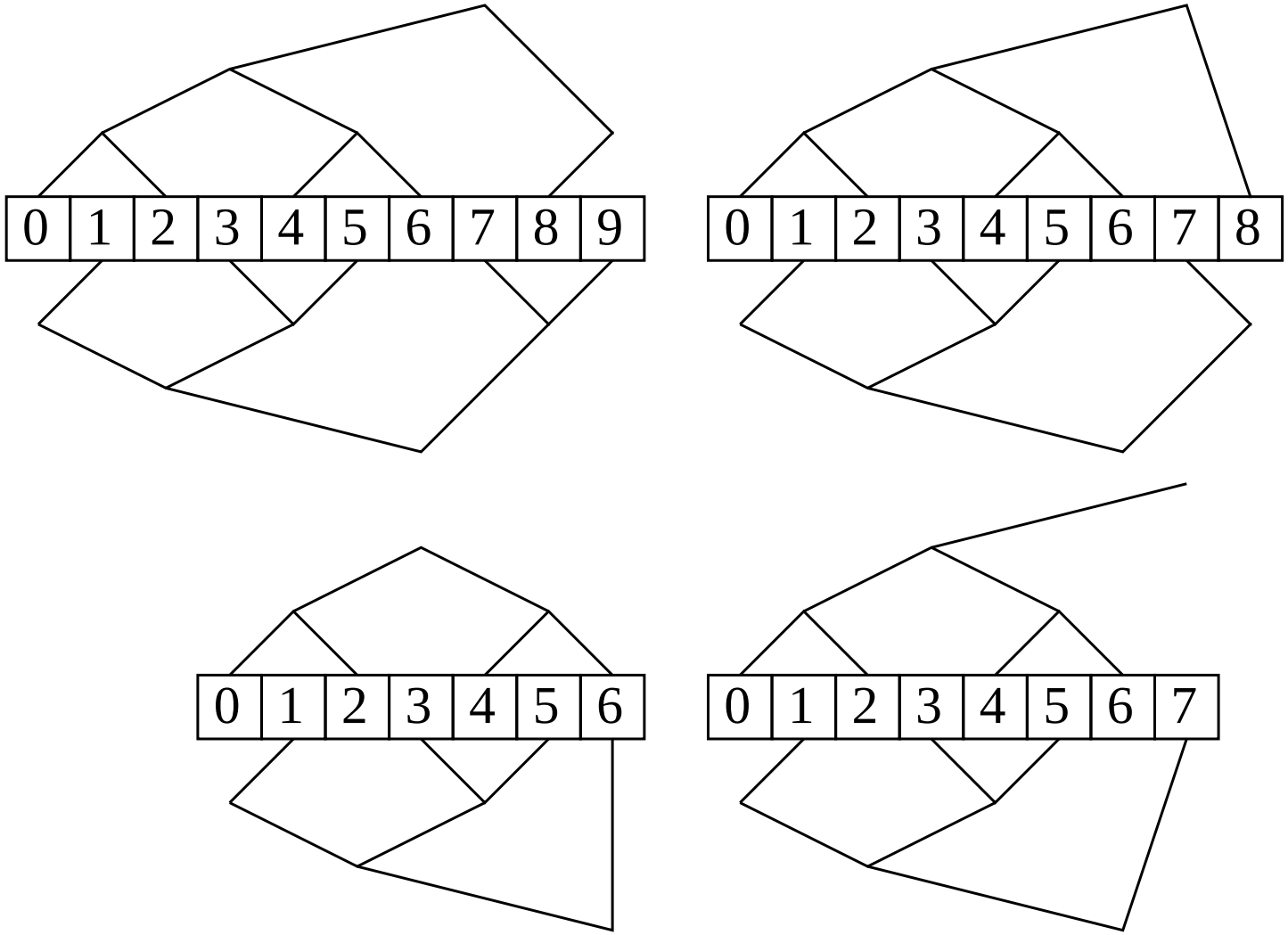


Beliebige Prozessorzahl





Beliebige Prozessorzahl



Aufbau der Bäume

Fall $p = 2^h - 1$: Oberer Baum + Unterer Baum + Wurzel

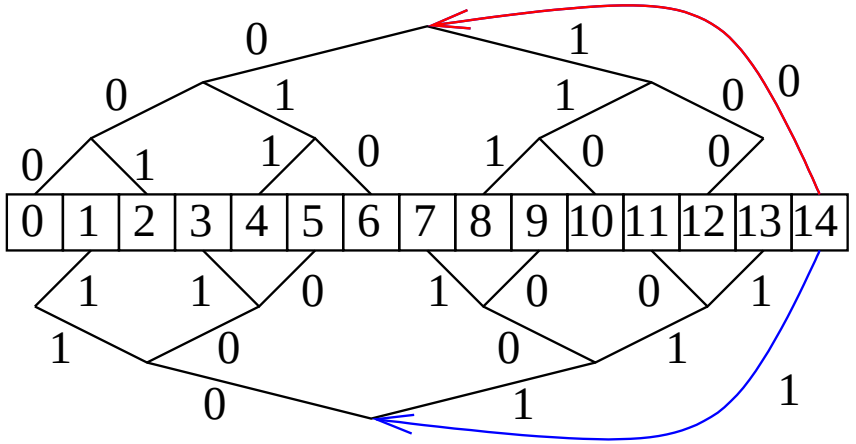
Oberer Baum: Vollst. Binärbaum der Höhe $h-1$, − rechtes Blatt

Unterer Baum: Vollst. Binärbaum der Höhe $h-1$, − linkes Blatt

Unterer Baum ≈ Oberer Baum um eins verschoben

Innere Knoten oberer Baum = Blätter unterer Baum.

Innere Knoten unterer Baum = Blätter oberer Baum.



Aufbau kleinerer Bäume (ohne Wurzel)

invariant : letzter Knoten hat Ausgangsgrad 1 in Baum x

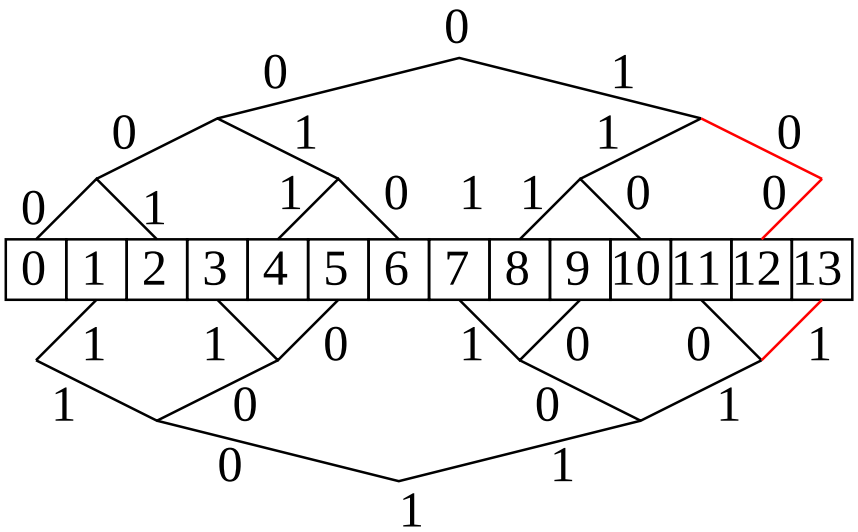
invariant : letzter Knoten hat Ausgangsgrad 0 in Baum $\bar{x}$

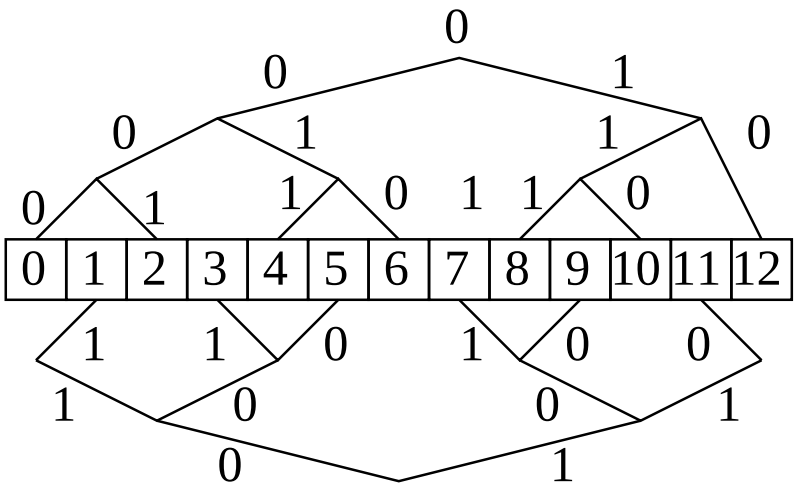
$p \rightsquigarrow p-1$:

Entferne letzten Knoten:

rechter Knoten in x hat jetzt Grad 0

rechter Knoten in $\bar{x}$ hat jetzt Grad 1





Kanten färben

Betrachte den bipartiten Graphen

$B = (\{s_0, \ldots, s_{p-1}\} \cup \{r_0, \ldots, r_{p-2}\}, E)$.

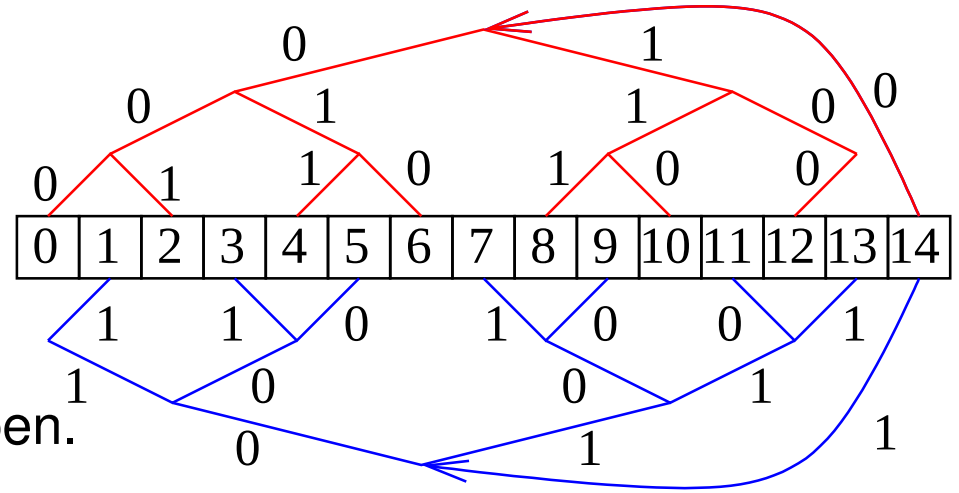
s_i: Senderrolle von PE i.

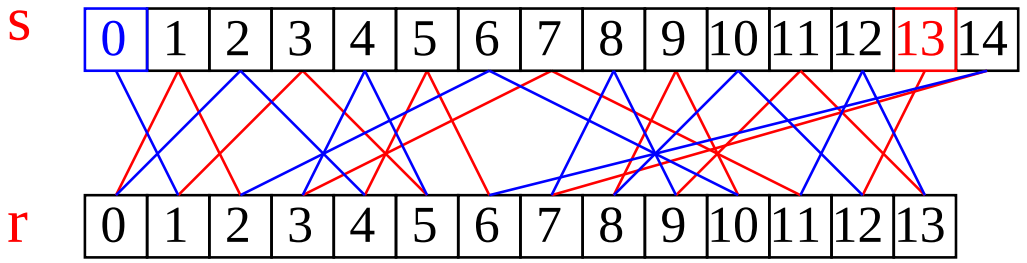
r_i: Empfängerrolle von PE i.

$2\times$ Grad 1. Sonst alles Grad 2.

$\Rightarrow B$ ist ein Pfad plus gerade Kreise.

Kanten abwechselnd mit 0 und 1 färben.

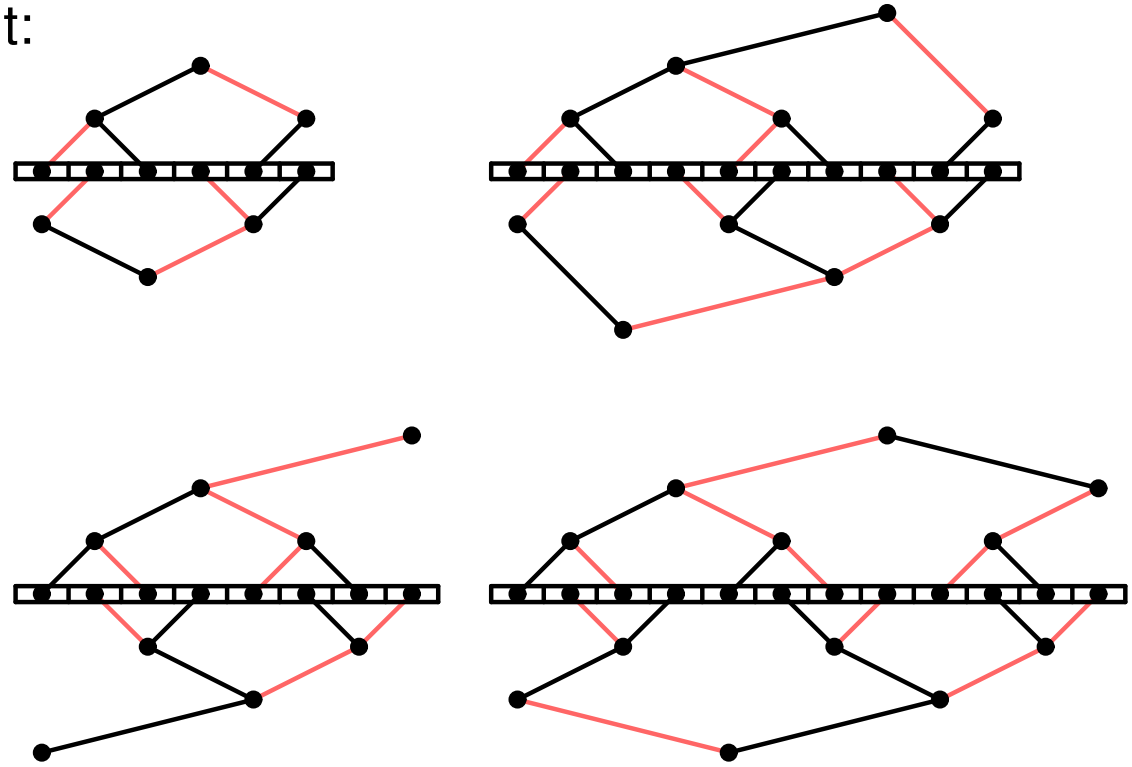




Offene Frage: Parallele Färbung ?

- In Zeit Polylog(p) mittels list ranking.
  (leider nicht praktikabel für kleine Eingaben)
- Schnelle explizite Berechnung color(i, p) ohne Kommunikation ?

Mirror layout:



Jochen Speck's Lösung

// Compute color of edge entering node i in the upper tree.

// h is a lower bound on the height of node i.

Function inEdgeColor(p,i,h)

if i is the root of T_1 **then return** 1

while i bitand $2^h = 0$ **do** $h{+}{+}$ // compute height

$$i' := \begin{cases} i - 2^h & \text{if } 2^{h+1} \text{ bitand } i = 1 \vee i + 2^h > p \\ i + 2^h & \text{else} \end{cases}$$ // compute parent of i

return inEdgeColor(p,i',h) xor $(p/2 \bmod 2)$ xor $[i' > i]$

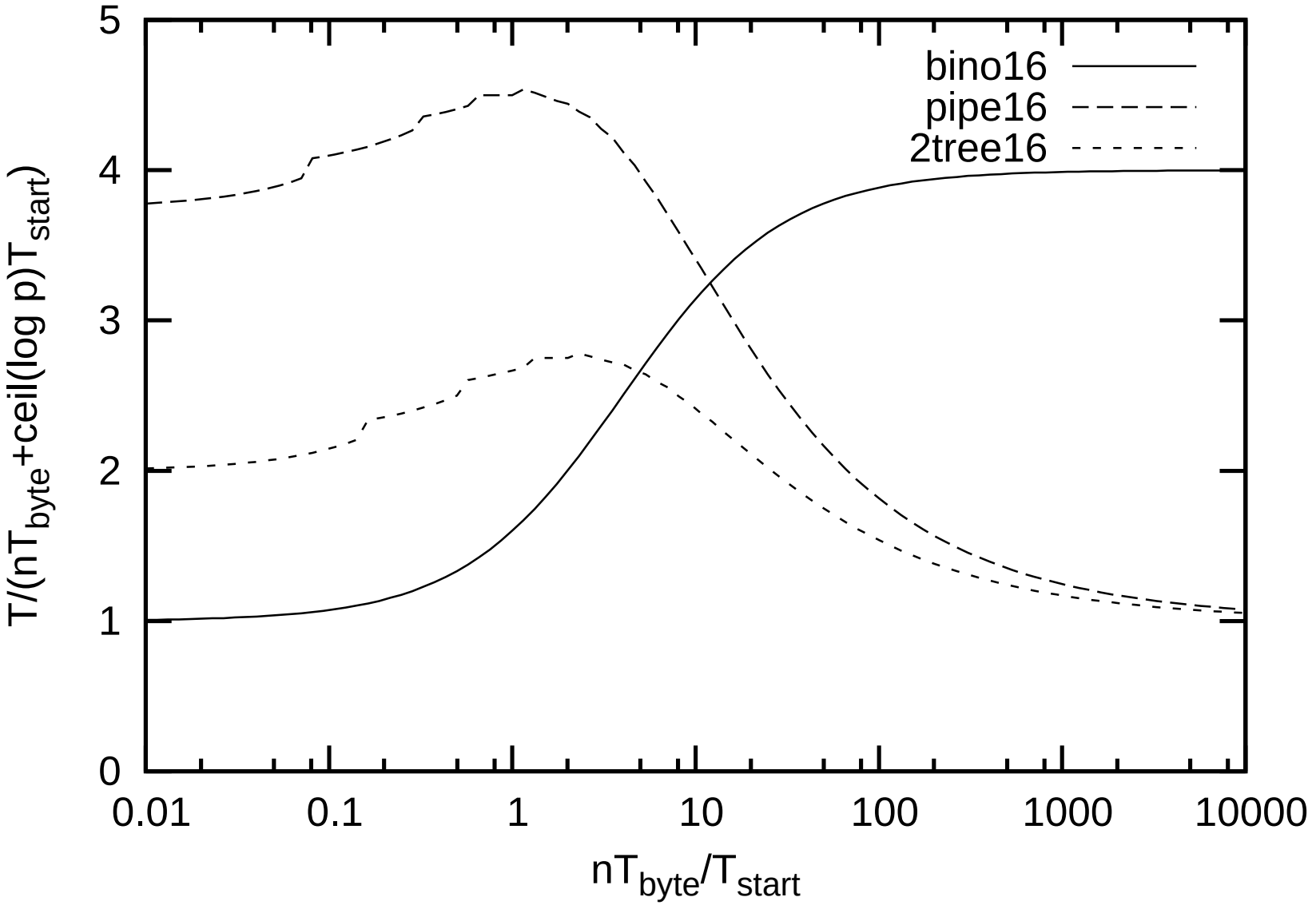
Analyse

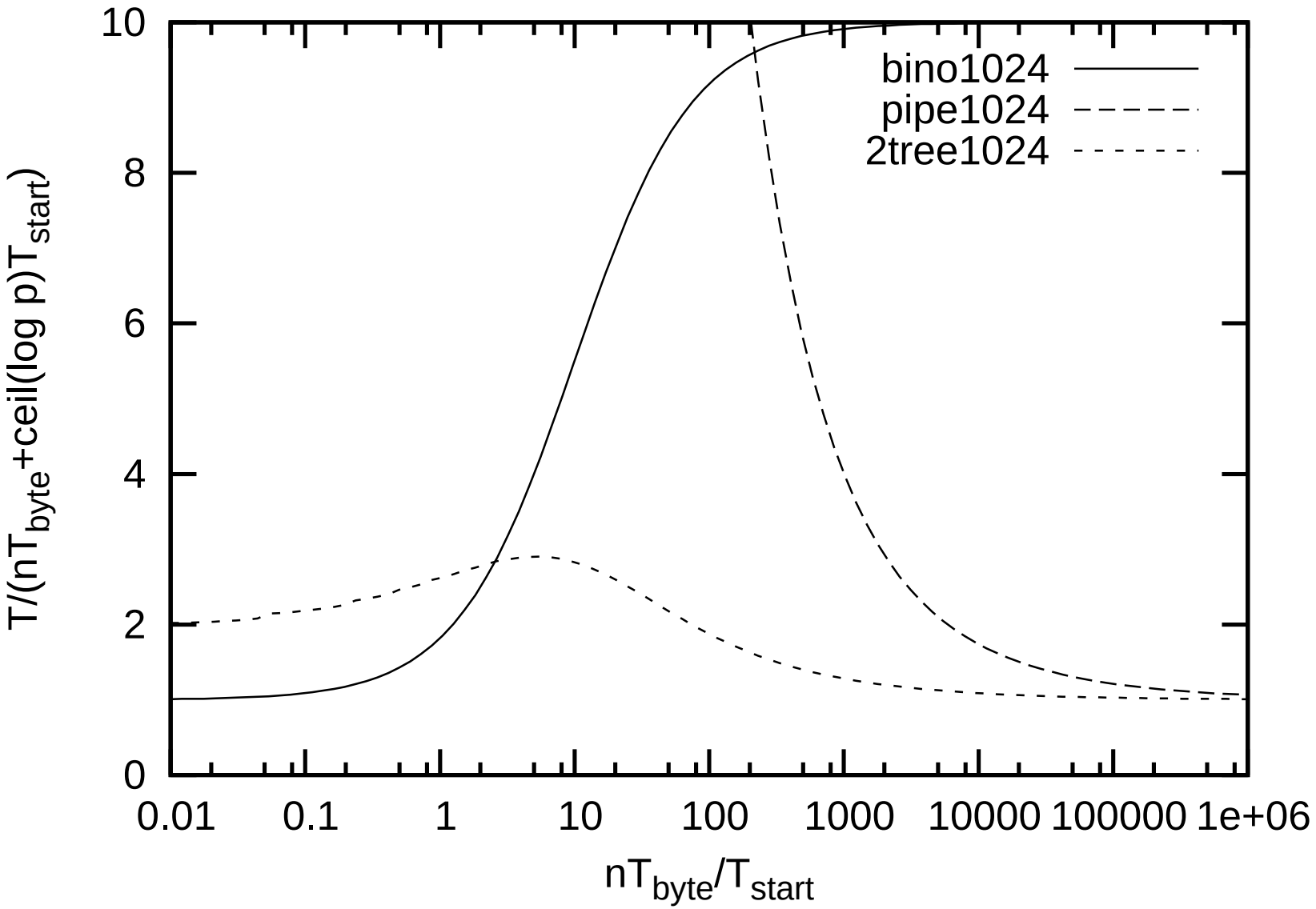
- Zeit $\frac{n}{k}\beta + \alpha$ pro Schritt
- $2j$ Schritte bis alle PEs in Schicht j erreicht
- $d = \lceil \log(p+1) \rceil$ Schichten
- Dann 2 Schritte pro weitere 2 Pakete

$$T(n,p,k) \approx \left(\frac{n}{k}\beta + \alpha\right)(2d + k - 1)), \text{ mit } d \approx \log p$$

optimales k: $\sqrt{\frac{n(2d-1)\beta}{\alpha}}$

$$T^*(n,p) \approx n\beta + \alpha \cdot 2\log p + \sqrt{2n\log p\alpha\beta}$$

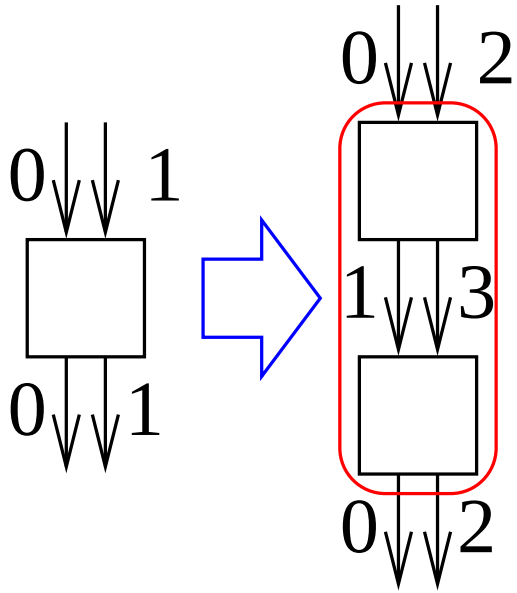

bino16
pipe16
2tree16
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
0
1
2
3
4
5
0.01
0.1
1
10
100
1000
10000


bino1024
pipe1024
2tree1024
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
0
2
4
6
8
10
0.01
0.1
1
10
100
1000
10000
100000
1e+06

Implementierung im Simplex-Modell

2 Zeitschritt duplex $\rightsquigarrow$ 4 Zeitschritt simplex.

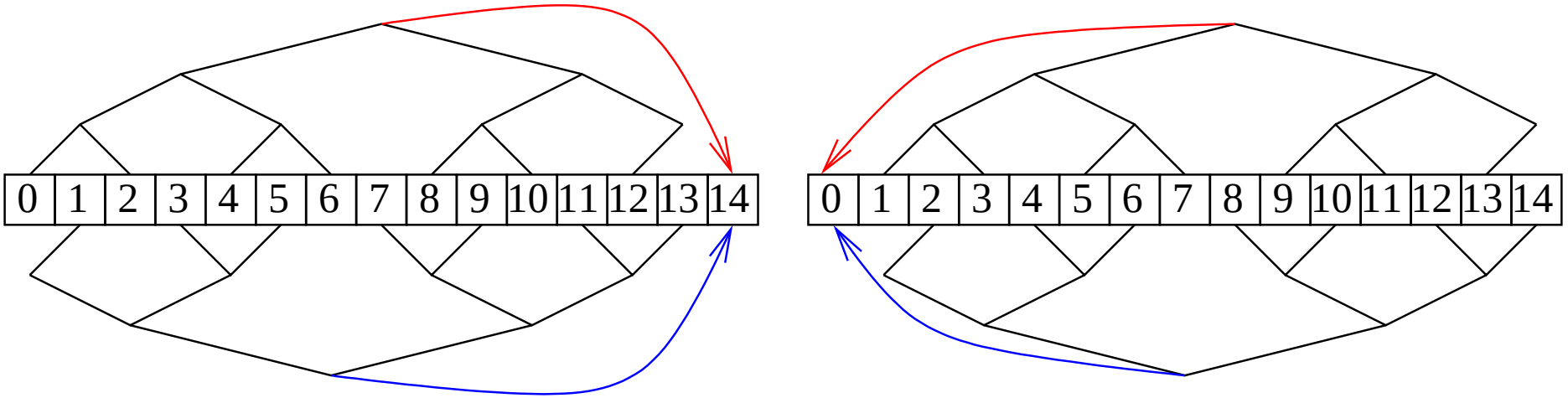
1 PE duplex $\rightsquigarrow$ 1 simplex couple $=$ sender + receiver.



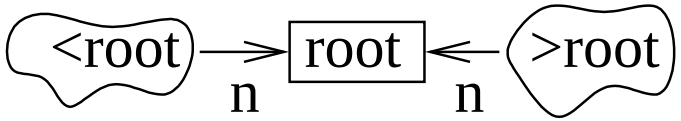
23-Reduktion

Nummerierung ist Inorder-Nummerierung für beide Bäume !

kommutativ oder root=0 oder root=p−1:



sonst:



Noch ein optimaler Algorithmus

[Johnsson Ho 85: Optimal Broadcasting and Personalized Communication in Hypercube, IEEE Transactions on Computers, vol. 38, no.9, pp. 1249-1268.]

Idee: getrennt marschieren — vereint schlagen

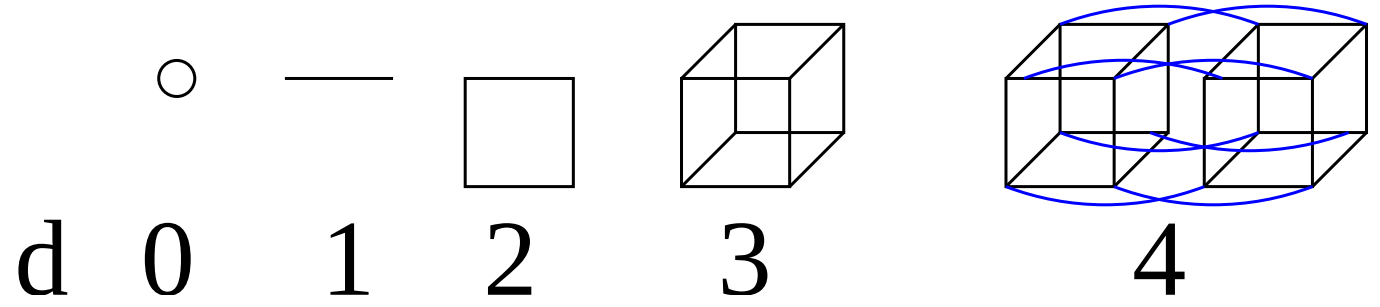
Modell: voll-duplex eingeschränkt auf einzige Kante pro PE

(Telefonmodell)

Anpassung halb-duplex: alles $\times 2$

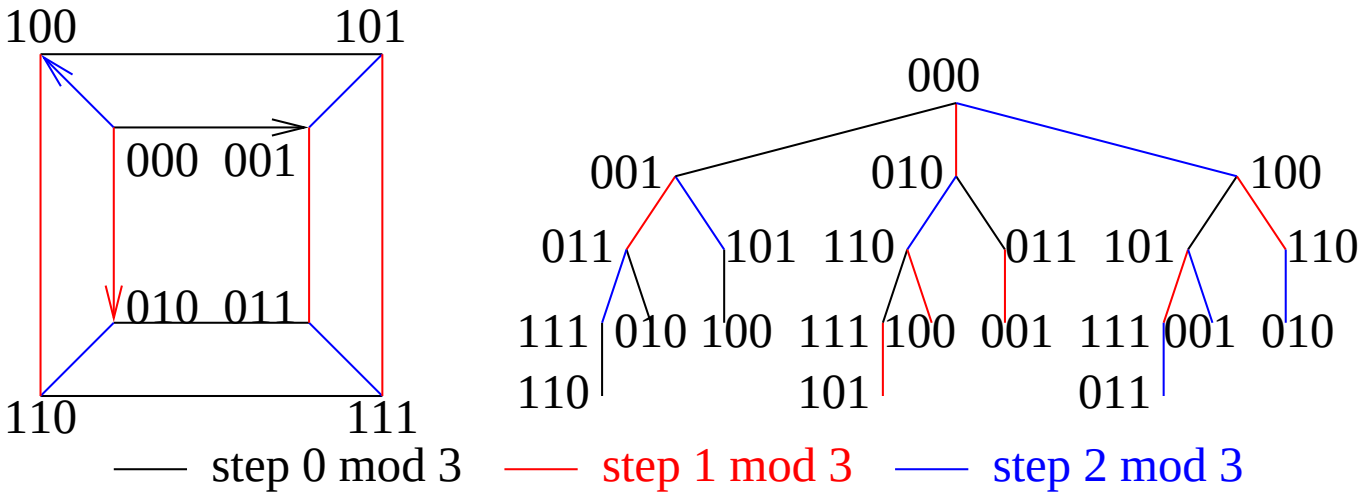
Hyperwürfel H_d

- $p = 2^d$ PEs
- Knoten $V = \{0,1\}^d$, also Knotennummern binär aufschreiben
- Kanten in Dimension i: $E_i = \left\{(u,v) : u \oplus v = 2^i\right\}$
- $E = E_0 \cup \cdots \cup E_{d-1}$



ESBT-Broadcasting

- In Schritt i Kommunikation entlang Dimension $i \bmod d$
- Zerlege H_d in d Edge-disjoint Spanning Binomial Trees
- 0^d verteilt zyklisch Pakete an Wurzeln der ESBTs
- ESBT-Wurzeln machen binomial tree broadcasting
  (außer fehlender kleinster Unterbaum 0^d)



Analyse, Telefonmodell

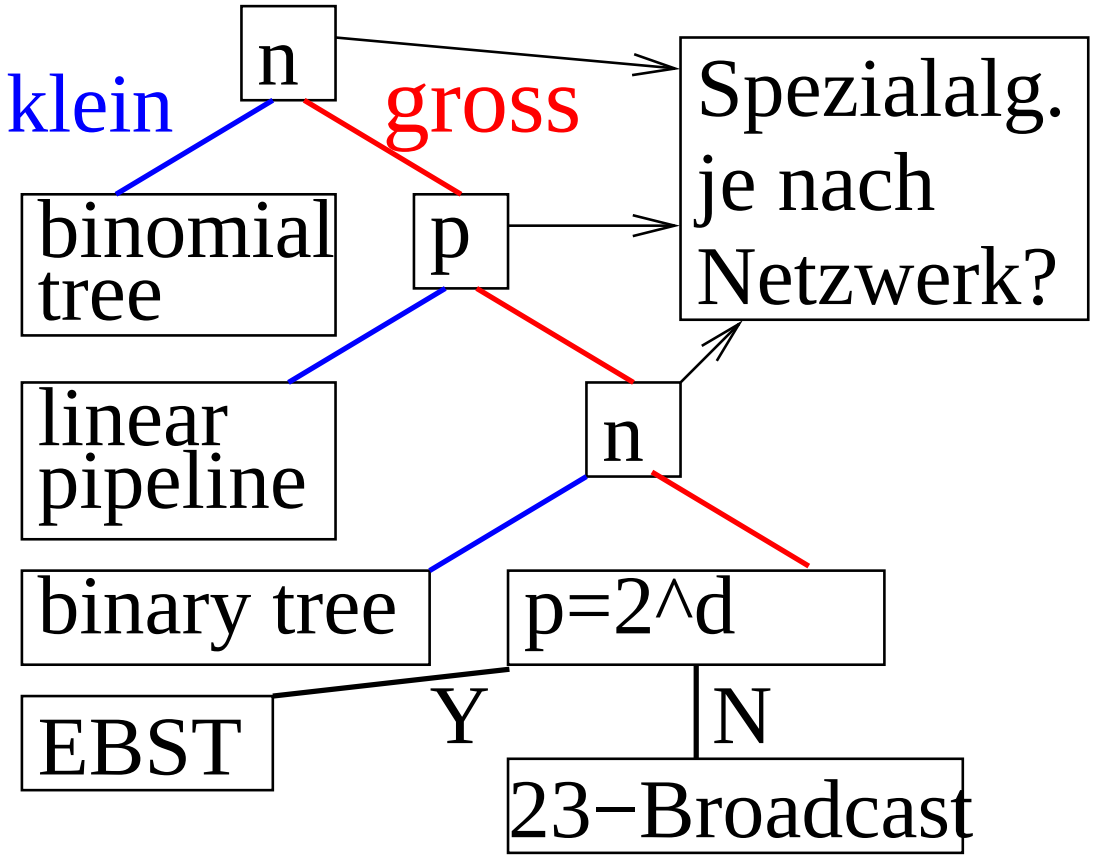
- k Pakete, k teilt n
- k Schritte bis letztes Paket PE 0 verlassen hat
- d Schritte bis es das letzte Blatt erreicht hat
- Insgesamt $d+k$ Schritte

$$T(n,p,k) = \left(\frac{n}{k}\beta + \alpha\right)(k+d)$$

optimales k: $\sqrt{\frac{nd\beta}{\alpha}}$

$T^*(n,p) := n\beta + d\alpha + \sqrt{nd\alpha\beta}$

Diskussion


n
klein
gross
Spezialalg.
je nach
Netzwerk?
binomial
tree
p
linear
pipeline
n
binary tree
p=2^d
EBST
Y
N
23−Broadcast

Reality Check

- Libraries (z.B. MPI) haben oft keine pipelined Implementierungen von kollektiven Operationen $\rightsquigarrow$ eigener Broadcast kann deutlich schneller sein als Bibliotheksfunktion.
- k einstellen ist komplizierter: nur abschnittweise lineare Kostenfunktion für Punkt-zu-Punkt-Kommunikation, Rundung
- Hyperwürfel werden langsam wenn Kommunikationslatzenzen eine große Varianz haben
- Fibonacci-Baum etc. bei asynchroner Kommunikation ggf. modifizieren (Sender ist eher fertig als Empfänger). Daten sollen an allen Blättern ungefähr gleichzeitig ankommen.

Broadcast für Bibliotheksimplementierer

- **EINE** Implementierung? $\rightsquigarrow$ 23-Broadcast
- Wenig, einfache Varianten? $\{$binomial tree, 23-Broadcast$\}$ oder $\{$binomial tree, 23-Broadcast, lineare Pipeline$\}$

Jenseits Broadcast

- Pipelining ist wichtige Technik zu Umgang mit großen Datenmengen.
- Hyperwürfelalgorithmen sind oft elegant und effizient. (Und oft einfacher als ESBT)
- Parametertuning (z.B. v. k) ist oft wichtig.

Sortieren

- Schnelles ineffizientes Ranking
- Quicksort
- Sample Sort
- Multiway Mergesort
- Selection
- Mehr zu Sortieren

Schnelles ineffizientes Ranking

m Elemente, m^2 Prozessoren:

Input: $A[1..m]$ // distinct elements

Output: $M[1..m]$ // $M[i] =$rang von $A[i]$

forall $(i,j) \in \{1..n\}^2$ **dopar** $B[i,j] := A[i] \geq A[j]$

forall $i \in \{1..n\}$ **dopar**

$$M[i] := \sum_{j=1}^{n} B[i,j]$$ // parallel subroutine

Ausführungszeit: $\approx T_{\text{broadcast}}(1) + T_{\text{reduce}}(1) = \mathcal{O}(\alpha \log p)$

i	A	B	1	2	3	4	5	<-	j	M
1	3		1	0	1	0	1			1
2	5		1	1	1	0	1			1
3	2		0	0	1	0	1			1
4	8		1	1	1	1	1			1
5	1		0	0	0	0	1			1
	A		3	5	2	8	1			

---

i	A	B	1	2	3	4	5	<-	j	M
1	3		1	0	1	0	1			3
2	5		1	1	1	0	1			4
3	2		0	0	1	0	1			2
4	8		1	1	1	1	1			5
5	1		0	0	0	0	1			1

Sortieren größerer Datenmengen

- n Eingabewerte. Anfangs n/p pro PE
- u.U. allgemeiner
- Ausgabe global sortiert

$$\boxed{d_{0,0},\ldots,d_{0,n/p-1}} \quad ,\ldots, \quad \boxed{d_{p-1,0},\ldots,d_{p-1,n/p-1}}$$

$$\Downarrow \pi$$

$$\boxed{s_{0,0} \leq \cdots \leq s_{0,n_1-1}} \leq \cdots \leq \boxed{s_{p-1,0} \leq \cdots \leq s_{p-1,n_{p-1}-1}}$$

- Vergleichsbasiertes Modell
- $T_{\text{seq}} = T_{\text{compr}} \frac{n}{p} \log \frac{n}{p} + \mathcal{O}\left(\frac{n}{p}\right)$

Vorsicht: abweichende Notation im Skript $n \leftrightarrow n/p$

Zurück zum schnellen Ranking

// Assume $p = a \times b$ PEs, PE Index is (i, j)

Procedure matrixRank(s)

sort(s) // locally

r:= all-gather-by-rows$(s,$ merge$)$

c:= all-gather-by-cols$(s,$ merge$)$

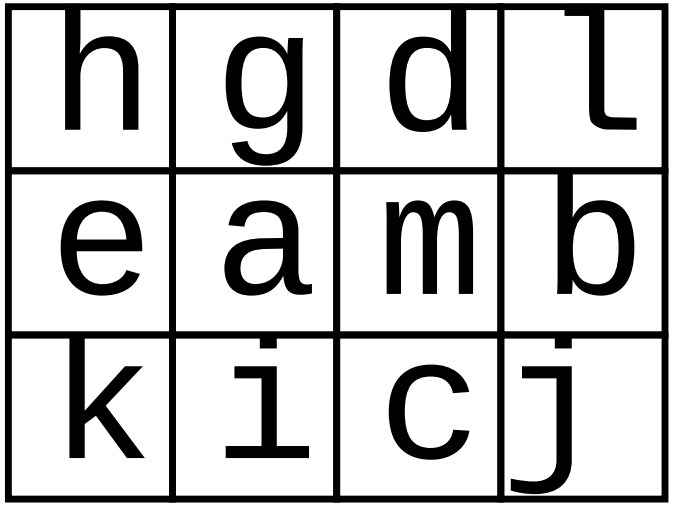
ranks:= $\langle |\{x \in c : x \leq y\}| : y \in r \rangle$ // merge

reduce-by-rows(ranks)

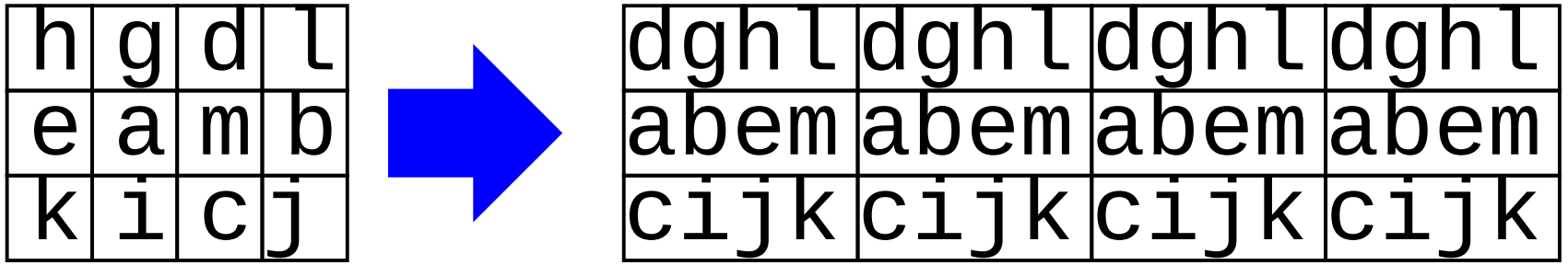
Time

$$\mathcal{O}\left(\alpha \log p + \beta \frac{n}{\sqrt{p}} + \frac{n}{p} \log \frac{n}{p}\right) \quad . \tag{1}$$

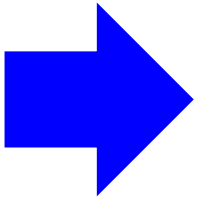
Beispiel



row all–gather–merge



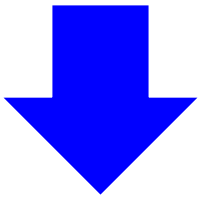
h	g	d	l
e	a	m	b
k	i	c	j



dghl	dghl	dghl	dghl
abem	abem	abem	abem
cijk	cijk	cijk	cijk

row all−gather−merge

col all−gather−merge



dghl ehk	dghl agi	dghl cdm	dghl bjl
abem ehk	abem agi	abem cdm	abem bjl
cijk ehk	cijk agi	cijk cdm	cijk bjl

h	g	d	l
e	a	m	b
k	i	c	j

dghl e 0123 h k	dghl a 1223 g i	dghl c 2222 d m	dghl b 1113 j l
abem e 0013 h k	abem a 1113 g i	abem c 0023 d m	abem b 0113 j l
cijk e 0223 h k	cijk a 1333 g i	cijk c 1222 d m	cijk b 1122 j l

h	g	d	l
e	a	m	b
k	i	c	j

dghl 0123	e h k	dghl 1223	a g i	dghl 2222	c d m	dghl 1113	b j l
abem 0013	e h k	abem 1113	a g i	abem 0023	c d m	abem 0113	b j l
cijk 0223	e h k	cijk 1333	a g i	cijk 1222	c d m	cijk 1122	b j l

d	g	h	l
4	6	7	11

a	b	e	m
1	2	5	12

c	i	j	k
3	8	9	10

Genauere Analyse (1 Maschinenwort/PE)

local sorting: $\frac{n}{p}\log\frac{n}{p}T_{\text{compr}}$

2× all-gather: $2\left(\beta n/\sqrt{p}+\frac{1}{2}\alpha\log p\right)$

local ranking: $2T_{\text{compr}}n/\sqrt{p}$

reduce JoHo-Algorithm:

$$\beta n/\sqrt{p}+\frac{1}{2}\alpha\log p+\sqrt{\alpha\beta n/\sqrt{p}\frac{1}{2}\log p}$$

Overall:

$$\frac{3}{2}\log p\alpha+3\beta n/\sqrt{p}+\sqrt{\alpha\beta 0.5n/\sqrt{p}\log p}+\frac{n}{p}\log\frac{n}{p}T_{\text{compr}}$$

Rechenbeispiel:

$p = 1024$, $\alpha = 10^{-5}$s, $\beta = 10^{-8}$s, $T_{\text{compr}} = 10^{-8}$s, $n/p = 32$.

$$\frac{3}{2}\log p\alpha + 3n\sqrt{p}\beta + \sqrt{0.5n\sqrt{p}\log p\alpha\beta} + n\log nT_{\text{compr}}$$

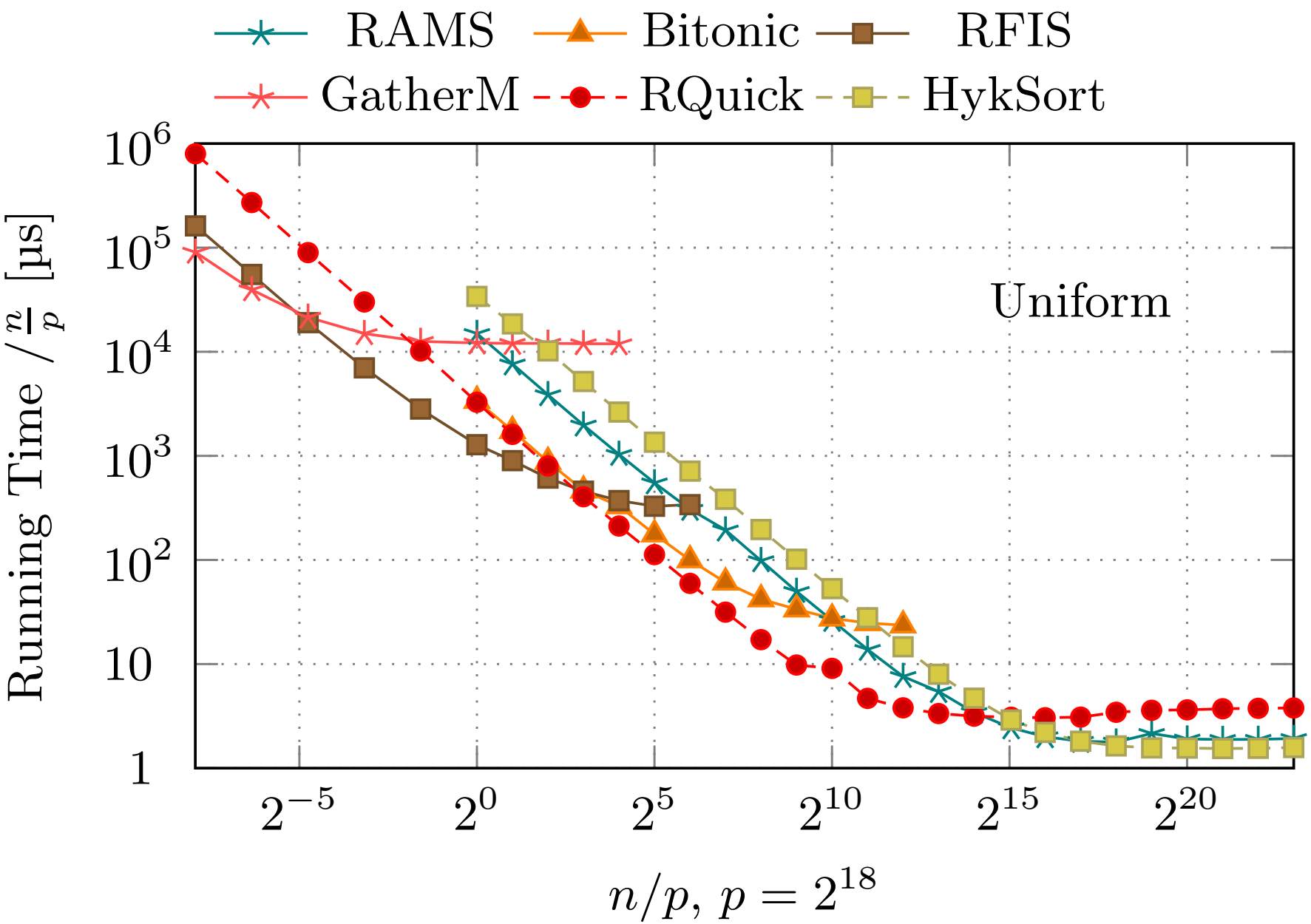
Zeit≈ 0.200ms.

Zum Vergleich: effizienter Gather+seq. sort:

$2 \cdot 32000 \cdot 10^{-8} + 10 \cdot 10^{-5} + 32000 \cdot 15 \cdot 10^{-8} \approx 5.6$ms

noch größerer Unterschied bei naivem gather

Messungen Axtmann Sanders ALENEX 2017



Quicksort

Sequentiell

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**

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

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

$f := d \leq v$

$j := \sum_{k=0}^{i} f@k$ // prefix sum

$p' := j@(p-1)$ // broadcast

if f **then** send d to PE j

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

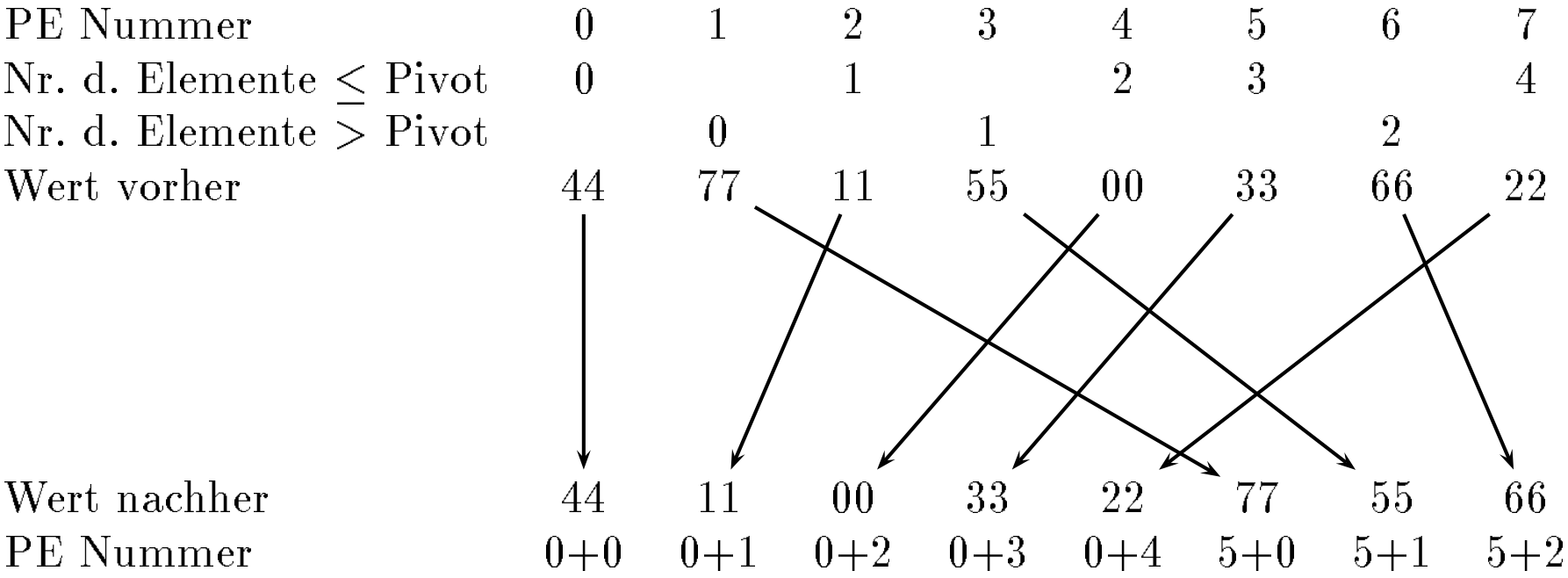
receive d

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 ($\rightarrow$später)

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

  ($\rightarrow$ Vorlesung randomisierte Algorithmen)

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

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

- Jedes PE hat i.allg. „große“ und „kleine“ Elemente.
- Aufteilung geht nicht genau auf
- Präfixsummen weiterhin nützlich
- Auf PRAM ergibt sich ein $\mathcal{O}\left(\frac{n \log n}{p} + \log^2 p\right)$ Algorithmus
- Bei verteiltem Speicher stört, dass jedes Element $\Omega(\log p)$ mal transportiert wird.
  $\rightsquigarrow \cdots \rightsquigarrow$ Zeit $\mathcal{O}\left(\frac{n}{p}(\log n + \beta \log p) + \alpha \log^2 p\right)$

Distributed memory parallel quicksort

Function parQuickSort(s : Sequence **of** Element, $i, j : \mathbb{N}$) : Sequence **of** Element

$p' := j - i + 1$

if $p' = 1$ **then** quickSort(s) ; **return** s // sort locally

$v :=$ pickPivot(s, i, j)

$a := \langle e \in s : e \leq v \rangle$; $b := \langle e \in s : e > v \rangle$

$n_a := \sum_{i \leq k \leq j} |a|@k$; $n_b := \sum_{i \leq k \leq j} |b|@k$

$k' := \frac{n_a}{n_a + n_b} p'$

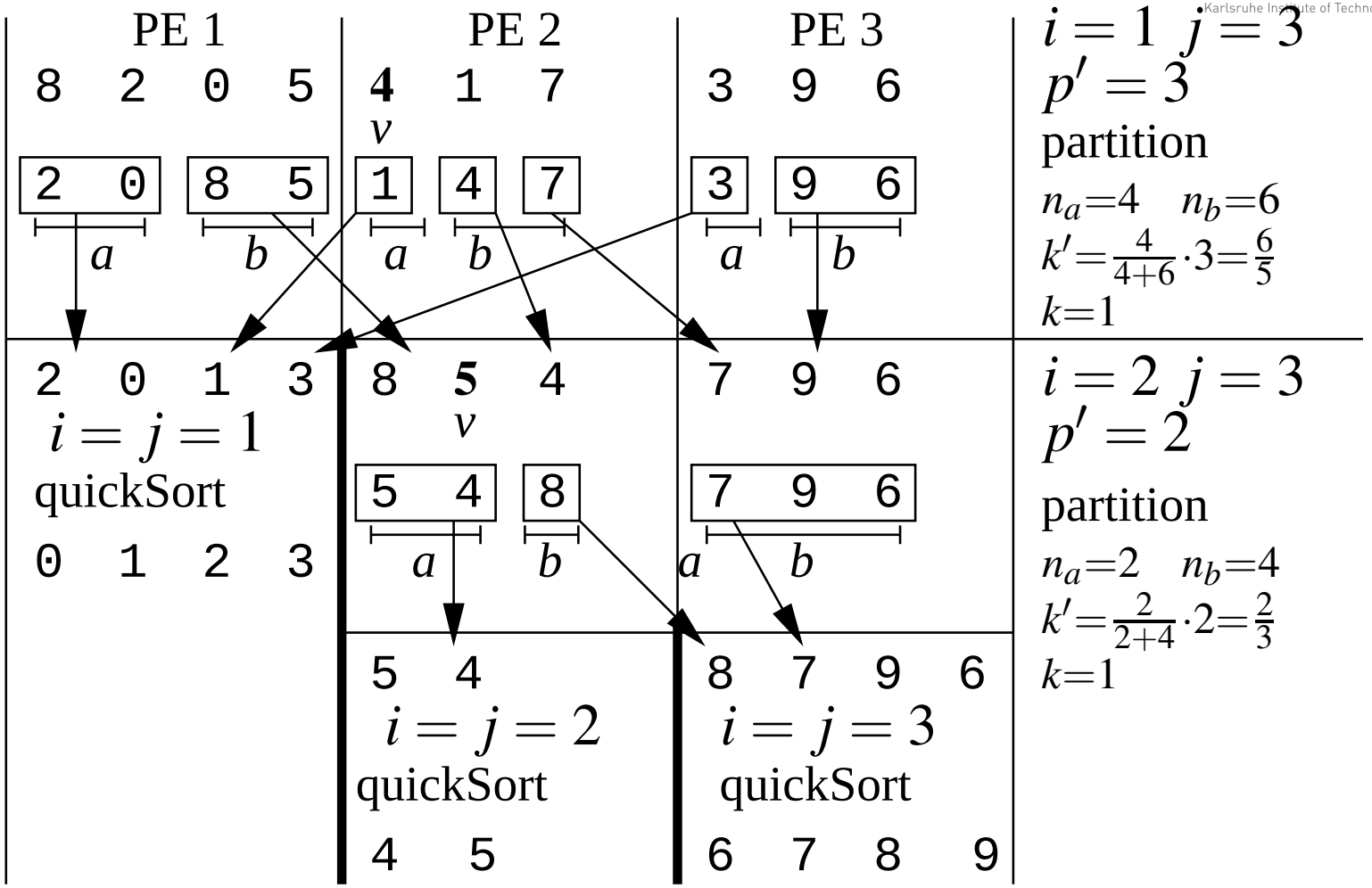
choose $k \in \{\lfloor k' \rfloor, \lceil k' \rceil\}$ such that $\max\left\{\left\lceil \frac{n_a}{k} \right\rceil, \left\lceil \frac{n_b}{p'-k} \right\rceil\right\}$ is minimized

send the a-s to PEs $i..i+k-1$ ($\leq \left\lceil \frac{n_a}{k} \right\rceil$ per PE)

send the b-s to PEs $i+k..j$ ($\leq \left\lceil \frac{n_b}{p'-k} \right\rceil$ per PE)

receive data sent to PE i_{PE} into s

if $i_{\text{PE}} < i + k$ **then** parQuickSort$(s, i, i+k-1)$ **else** parQuickSort$(s, i+k, j)$


PE 1
PE 2
PE 3
8 2 0 5
4 1 7
3 9 6
v
2 0
8 5
1
4
7
3
9 6
a
b
a
b
a
b
$i = 1\ j = 3$
$p' = 3$
partition
n_a=4 n_b=6
$k' = \frac{4}{4+6} \cdot 3 = \frac{6}{5}$
k=1
2 0 1 3
8 5 4
7 9 6
$i = j = 1$
quickSort
0 1 2 3
v
$i = 2\ j = 3$
$p' = 2$
partition
5 4
8
7 9 6
a
b
a
b
n_a=2 n_b=4
$k' = \frac{2}{2+4} \cdot 2 = \frac{2}{3}$
k=1
5 4
8 7 9 6
$i = j = 2$
quickSort
$i = j = 3$
quickSort
4 5
6 7 8 9

Load Balance

Vereinfachtes Szenario: Splitting immer im Verhältnis 1:2

größeres Teilproblem kriegt ein PE-Load zu viel.

Imbalance-Faktor:

$$\prod_{i=1}^{k} 1+\frac{1}{p\left(\frac{2}{3}\right)^i} = e^{\sum_{i=1}^{k} \ln\left(1+\frac{1}{p\left(\frac{2}{3}\right)^i}\right)}$$

$$\leq e^{\sum_{i=1}^{k} \frac{1}{p\left(\frac{2}{3}\right)^i}} = e^{\frac{1}{p}\sum_{i=0}^{k}\left(\frac{3}{2}\right)^i} \text{ geom. Summe}$$

$$= e^{\frac{1}{p}\frac{\left(\frac{3}{2}\right)^{k+1}-1}{\frac{3}{2}-1}} \leq e^{\frac{1}{p}3\left(\frac{3}{2}\right)^k} = e^3 \approx 20.1 \ .$$

Die gute Nachricht:

Zeit $\mathcal{O}\left(\frac{n}{p}\log\frac{n}{p}+\log^2 p\right)$

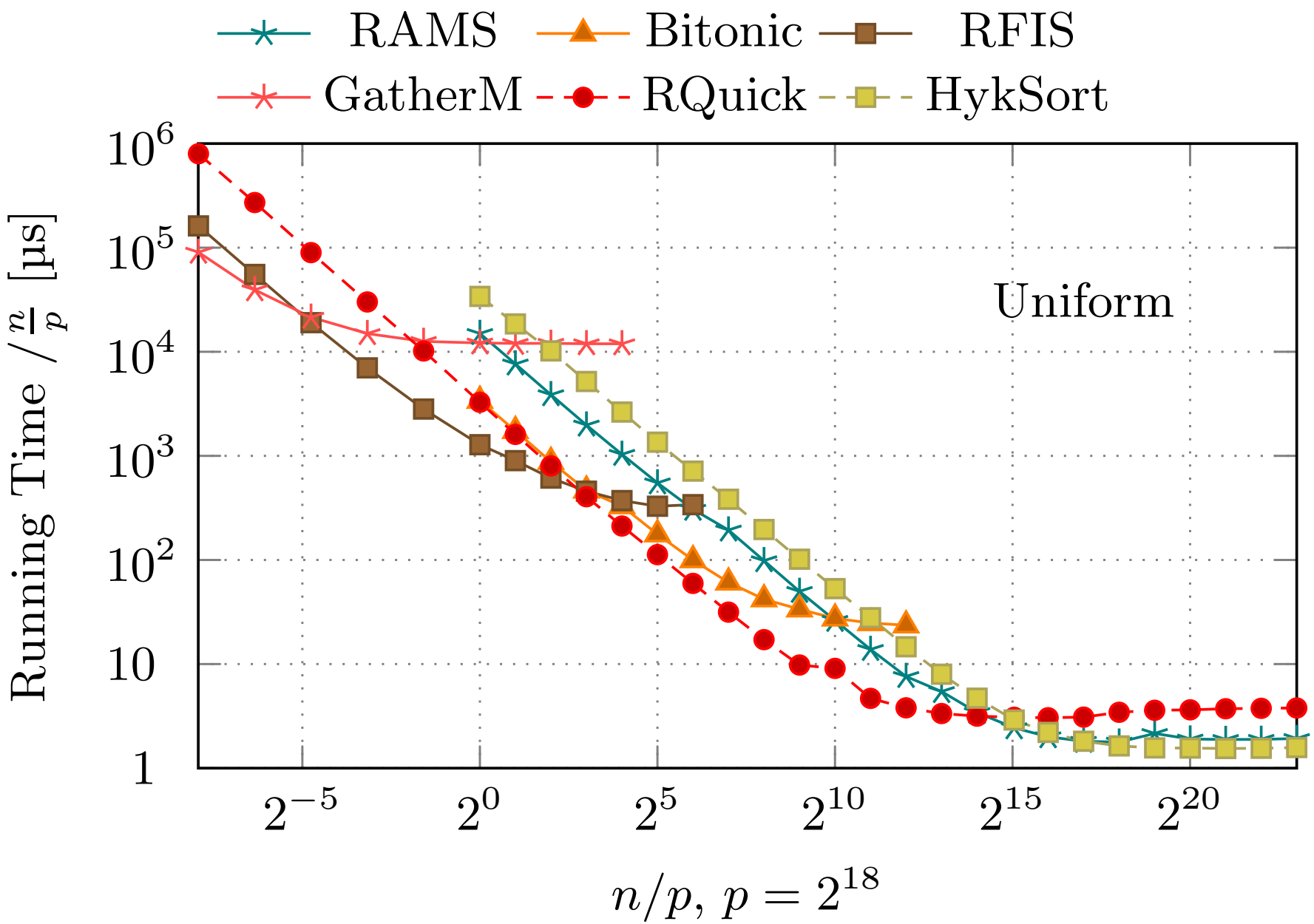
Bessere Lastbalancierung?

- Janus-quicksort? Axtmann, Wiebigke, Sanders, IPDPS 2018
- bei kleinem p' pivot sorgfältig wählen
- bei kleinem p' ($\Theta(\log p)$) auf sample sort umsteigen?

Alternative: immer Halbierung der Prozessoren, Randomisierung, sorgfältige Pivot-Wahl.

Axtmann, Sanders, ALENEX 2017

Messungen Axtmann Sanders ALENEX 2017



Multi-Pivot Verfahren

Vereinfachende Annahme: Splitter fallen vom Himmel

// Für $0 < k < p$ sei v_k das Element mit Rang $k \cdot n/p$
// Außerdem setzen wir $v_0 = -\infty$ und $v_P = \infty$.
initialisiere p leere Nachrichten N_k, $(0 \leq k < p)$
for $i := 0$ **to** $n-1$ **do**
    bestimme k, so daß $v_k < d_i \leq v_{k+1}$
    nimm d_i in Nachricht N_k auf
schicke N_i an PE i und // All-to-all
empfange p Nachrichten // personalized communication
sortiere empfangene Daten

Analyse

$$T_{\text{par}} = \overbrace{\mathcal{O}\left(\frac{n}{p}\log p\right)}^{\text{verteilen}} + \overbrace{T_{\text{seq}}(n/p)}^{\text{lokal sortieren}} + \overbrace{T_{\text{all-to-all}}(p, n/p)}^{\text{Datenaustausch}}$$

$$\approx \frac{T_{\text{seq}}(n)}{p} + 2\frac{n}{p}\beta + p\alpha$$

Idealisierende Annahme ist realistisch für Permutation.

Sample Sort

choose a total of Sp random elements s_k, (S per PE) ($1 \leq k \leq Sp$)

sort $[s_1, \ldots, s_{Sp}]$ // or only

for $i := 1$ **to** $p-1$ **do** $v_i := s_{Si}$ // multiple selection

$v_0 := -\infty$; $v_P := \infty$

unsortierte Eingangsdaten

19	7	12	6	30	17	3	20	14
1	9	13	13	10	11	18	5	16
25	4	2	16	27	22	15	21	8

zufälliges Sample

7 13 25 | 6 17 10 | 20 18 21

Sample sortiert und aufgeteilt

6 7 10 | 13 17 18 | 20 21 25

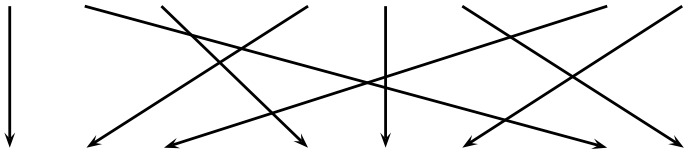
Broadcast der Pivotelemente

$(p_0 = -\infty)$ $p_1 = 10$ $p_2 = 18$ $(p_3 = \infty)$

Elemente klassifiziert

I_0	I_1	I_2	I_0	I_1	I_2	I_0	I_1	I_2
1	12	25	6	17	30	3	14	20
4	13	19	10	13	27	5	18	21
2				11	22	8	16	
7				16			15	
9								

Umverteilung



lokal sortierte Daten

I_0			I_1			I_2		
1	2	3	11	12	13	19	20	21
4	5	6	13	14	15	22	25	27
7	8	9	16	16	18	30		
10								

Lemma 2. *$S = \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ genügt damit mit Wahrscheinlichkeit $\geq 1 - \frac{1}{n}$ kein PE mehr als $(1+\varepsilon)n/p$ Elemente erhält.*

Lemma:
$S = \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ genügt damit mit Wahrscheinlichkeit $\geq 1 - \frac{1}{n}$ kein PE mehr als $(1+\varepsilon)n/p$ Elemente erhält.

Beweisansatz: Wir analysieren einen Alg. bei dem global samples mit Zurücklegen gewählt werden.

Sei $\langle e_1, \ldots, e_n \rangle$ die Eingabe in sortierter Reihenfolge.

fail: Ein PE kriegt mehr als $(1+\varepsilon)n/p$ Elemente

$\rightarrow \exists j :\leq S$ samples aus $\langle e_j, \ldots, e_{j+(1+\varepsilon)n/p} \rangle$ (Ereignis $\mathcal{E}_j$)

$\rightarrow \mathbb{P}[\text{fail}] \leq n\mathbb{P}\left[\mathcal{E}_j\right]$, j fest.

Sei X_i:= $\begin{cases} 1 & \text{falls } s_i \in \langle e_j, \ldots, e_{j+(1+\varepsilon)n/p} \rangle \\ 0 & \text{sonst} \end{cases}$, $X := \sum_i X_i$

$\mathbb{P}\left[\mathcal{E}_j\right] = \mathbb{P}[X < S] = \mathbb{P}[X < 1/(1+\varepsilon)\mathbb{E}[X]] \approx \mathbb{P}[X < (1-\varepsilon)\mathbb{E}[X]]$

$\mathbb{E}[X_i] = \mathbb{P}[X_i = 1] = \frac{1+\varepsilon}{p}$

Chernoff-Schranke

Lemma 3. *Sei $X = \sum_i X_i$ die Summe unabhängiger 0-1 Zufallsvariablen.*

$$\mathbb{P}[X < (1-\varepsilon)\mathbb{E}[X]] \leq \exp\left(-\frac{\varepsilon^2 \mathbb{E}[X]}{2}\right) \quad .$$

Angewandt auf unser Problem:

$$\mathbb{P}[X < S] \leq \exp\left(-\frac{\varepsilon^2(1+\varepsilon)S}{2}\right) \leq \exp\left(-\frac{\varepsilon^2 S}{2}\right) \overset{!}{\leq} \frac{1}{n^2}$$

$$\leftrightarrow S \geq \frac{4}{\varepsilon^2} \ln n$$

■

Analyse von Sample Sort

$$T_{\text{sampleSort}}(p,n) = \overbrace{\overbrace{T_{\text{fastsort}}(p, \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right))}^{\text{sample sortieren}} + \overbrace{T_{\text{allgather}}(p)}^{\text{splitter sammeln/verteilen}}}^{\text{klein wenn } n \gg p^2 \log p}$$

$$+ \underbrace{\mathcal{O}\left(\frac{n}{p}\log p\right)}_{\text{verteilen}} + \underbrace{T_{\text{seq}}((1+\varepsilon)\frac{n}{p})}_{\text{lokal sortieren}} + \underbrace{T_{\text{all-to-all}}(p,(1+\varepsilon)\frac{n}{p})}_{\text{Datenaustausch}}$$

Samples Sortieren

- Mit Gather/Gossiping
- Mit Gather–Merge
- Schnelles Ranking
- Paralleles Quicksort
- Rekursiv mit Sample-Sort

Samples Sortieren

effizient falls $n \gg$

- ☐ Mit Gather/Gossiping — $\frac{p^2 \log p T_{\text{compr}}}{\varepsilon^2}$
- ☐ Mit Gather–Merge — $\frac{p^2 \beta}{\varepsilon^2 T_{\text{compr}}}$
- ☐ Schnelles Ranking — $\frac{p^2 \beta}{\log p T_{\text{compr}}}$
- ☐ Paralleles Quicksort — $\frac{p^2 \beta}{\log p T_{\text{compr}}}$
- ☐ Rekursiv mit Sample-Sort

MPI Sample Sort – Init and Local Sample

Many thanks to Michael Axtmann

```
template<class Element>                                                    1
void parallelSort(MPI_Comm comm, vector<Element>& data,                    2
                  MPI_Datatype mpiType, int p, int myRank)                 3
{ random_device rd;                                                        4
  mt19937 rndEngine(rd());                                                 5
  uniform_int_distribution<size_t> dataGen(0, data.size() − 1);            6
  vector<Element> locS; // local sample of elements from input <data>      7
  const int a = (int)(16∗log(p)/log(2.)); // oversampling ratio            8
  for (size_t i=0; i < (size_t)(a+1); ++i)                                 9
    locS.push_back(data[dataGen(rndEngine)]);                              10
```

Find Splitters

```
vector<Element> s(locS.size() ∗ p); // global samples                    1
MPI_Allgather(locS.data(), locS.size(), mpiType,                         2
   s.data(), locS.size(), mpiType, comm);                                3


sort(s.begin(), s.end()); // sort global sample                          5
for (size_t i=0; i < p−1; ++i) s[i] = s[(a+1) ∗ (i+1)]; //select splitters  6
s.resize(p−1);                                                           7
```

Partition Locally

```
vector<vector<Element>> buckets(p); // partition data                    1
for(auto& bucket : buckets) bucket.reserve((data.size() / p) * 2);       2
for( auto& el : data) {                                                  3
  const auto bound = upper_bound(s.begin(), s.end(), el);                4
  buckets[bound - s.begin()].push_back(el);                              5
}                                                                        6
data.clear();                                                            7
```

Find Message Sizes

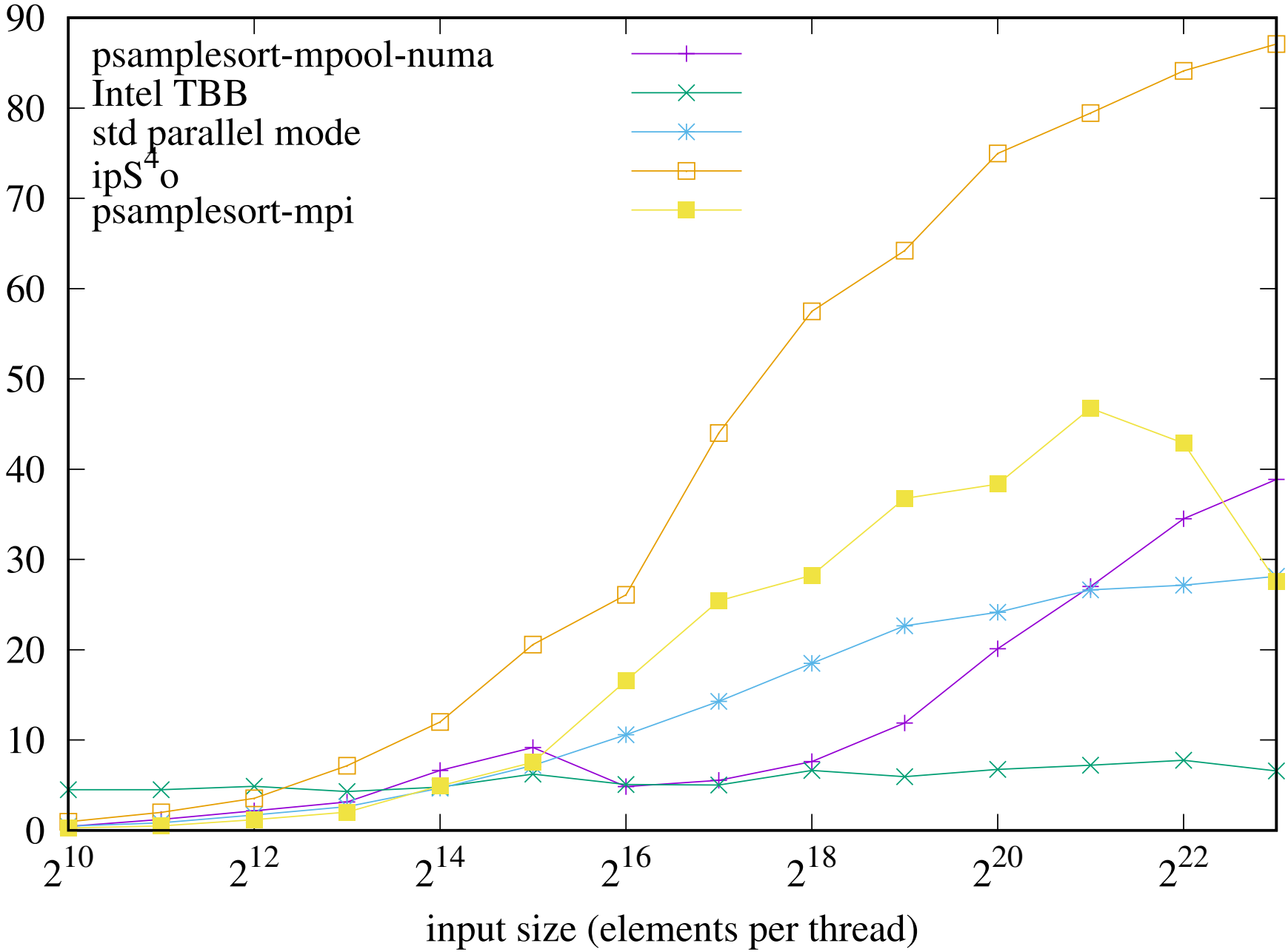
```
// exchange bucket sizes and calculate send/recv information
vector<int> sCounts, sDispls, rCounts(p), rDispls(p + 1);
sDispls.push_back(0);
for (auto& bucket : buckets) {
  data.insert(data.end(), bucket.begin(), bucket.end());
  sCounts.push_back(bucket.size());
  sDispls.push_back(bucket.size() + sDispls.back());
}
MPI_Alltoall(sCounts.data(),1,MPI_INT,rCounts.data(),1,MPI_INT,comm);
// exclusive prefix sum of recv displacements
rDispls[0] = 0;
for(int i = 1; i <= p; i++) rDispls[i] = rCounts[i-1]+rDispls[i-1];
```

Data Exchange and Local Sorting

```
    vector<Element> rData(rDispls.back()); // data exchange                    1
    MPI_Alltoallv(data.data(), sCounts.data(), sDispls.data(), mpiType,        2
      rData.data(), rCounts.data(), rDispls.data(), mpiType, comm);            3


    sort(rData.begin(), rData.end());                                          5
    rData.swap(data);                                                          6
}                                                                              7
```

Experiments Speedup on 4× Intel E7-8890 v3



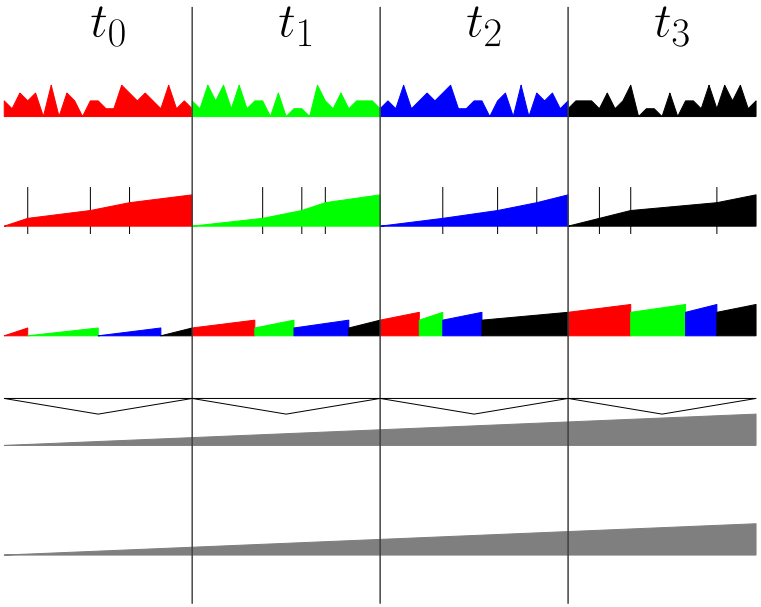
Sortieren durch Mehrwegemischen

Function mmSort(d, n, p) // shared memory not SPMD

PE i sorts $d[in/p..(i+1)n/p]$; barrier synchronization

PE i finds v_i with rank in/p in d; barrier synchronization

PE i merges p subsequences with $v_k \leq d_j < v_{k+1}$



Multisequence Selection

Idee: jedes PE bestimmt einen Splitter mit geeignetem globalem Rang (shared memory)

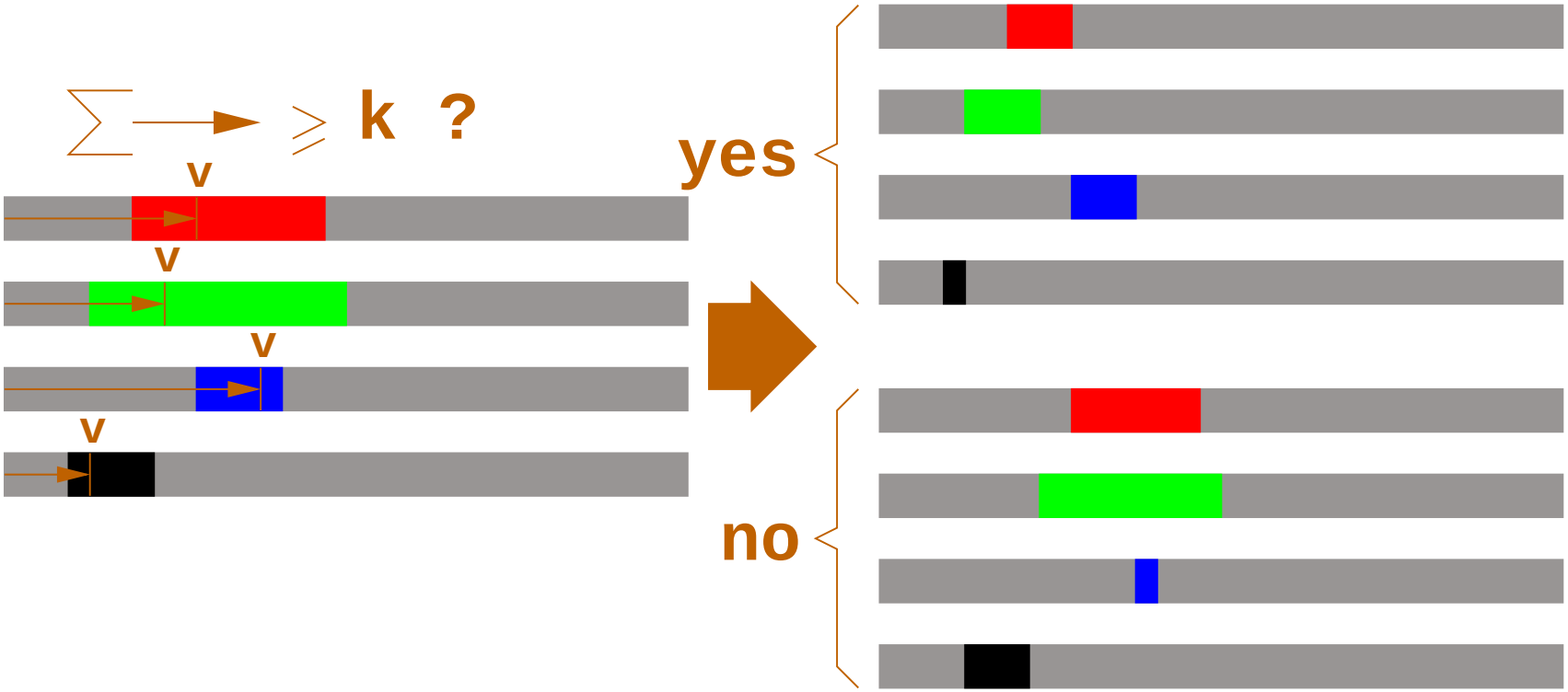
Vergleichsbasierte untere Schranke: $\mathcal{O}\left(p \log \frac{n}{p}\right)$

Wir geben Algorithmus mit $\mathcal{O}\left(p \log n \log \frac{n}{p}\right)$

Splitter Selection

Processor i selects the element with global rank $k = \frac{in}{p}$.

Simple algorithm: quickSelect exploiting sortedness of the sequences.



Idee:

Normales select aber $p\times$ binäre Suche statt Partitionierung

Function msSelect(S : Array **of** Sequence **of** Element; k : $\mathbb{N}$) : Array **of** $\mathbb{N}$

for $i := 1$ **to** $|S|$ **do** $(\ell_i, r_i) := (0, |S_i|)$

invariant $\forall i$: $\ell_i..r_i$ contains the splitting position of S_i

invariant $\forall i, j : \forall a \leq \ell_i, b > r_j : S_i[a] \leq S_j[b]$

while $\exists i : \ell_i < r_i$ **do**

$v :=$ pickPivot(S, ℓ, r)

for $i := 1$ **to** $|S|$ **do** $m_i :=$ binarySearch$(v, S_i[\ell_i..r_i])$

if $\sum_i m_i \geq k$ **then** $r := m$ **else** $\ell := m$

return ℓ

Analyse von p-way Mergesort

$$T_{\text{pMergeSort}}(p,n) = \mathcal{O}\left(\underbrace{\frac{n}{p}\log\frac{n}{p}}_{\text{lokal sortieren}} + \underbrace{p\log n\log\frac{n}{p}}_{\text{ms-selection}} + \underbrace{\frac{n}{p}\log p}_{\text{merging}} \right)$$

- effizient falls $n \gg p^2 \log p$
- deterministisch (fast)
- perfekte Lastbalancierung
- etwas schlechtere konstante Faktoren als sample sort

Verteilte Multisequence Selection

Owner computes Paradigma

$\mathcal{O}(\log n)$ globale Rekursionslevel.

Gather + Broadcast für Pivotbestimmung/Verteilung (Vektorlänge $p-1$).

überall $p-1$ lokale Suchen.

Reduktion für Bestimmung der Partionsgrößen (Vektorenlänge $p-1$).

Erwartete Zeit

$$\mathcal{O}\left(\log n\left(p(\log\tfrac{n}{p}+\beta)+\log p\alpha\right)\right)$$

Verteilte Multisequence Selection

Function dmSelect(s : Seq **of** Elem; k : Array$[1..p]$ **of** $\mathbb{N}$) : Array$[1..p]$ **of** $\mathbb{N}$

ℓ, r, m, v, σ : **Array** $[1..p]$ **of** $\mathbb{N}$

for $i := 1$ **to** p **do** $(\ell_i, r_i) := (0, |s|)$ // initial search ranges

while $\exists i, j : \ell_i @ j \neq r_i @ j$ **do** // or-reduction

$v :=$ pickPivotVector(s, ℓ, r) // reduction, prefix sum, broadcast

for $i := 1$ **to** p **do** $m_i :=$ binarySearch$(v_i, s[\ell_i..r_i])$

$\sigma := \sum_i m@i$ // vector valued reduction

for $i := 1$ **to** p **do** **if** $\sigma_i \geq k_i$ **then** $r_i := m_i$ **else** $\ell_i := m_i$

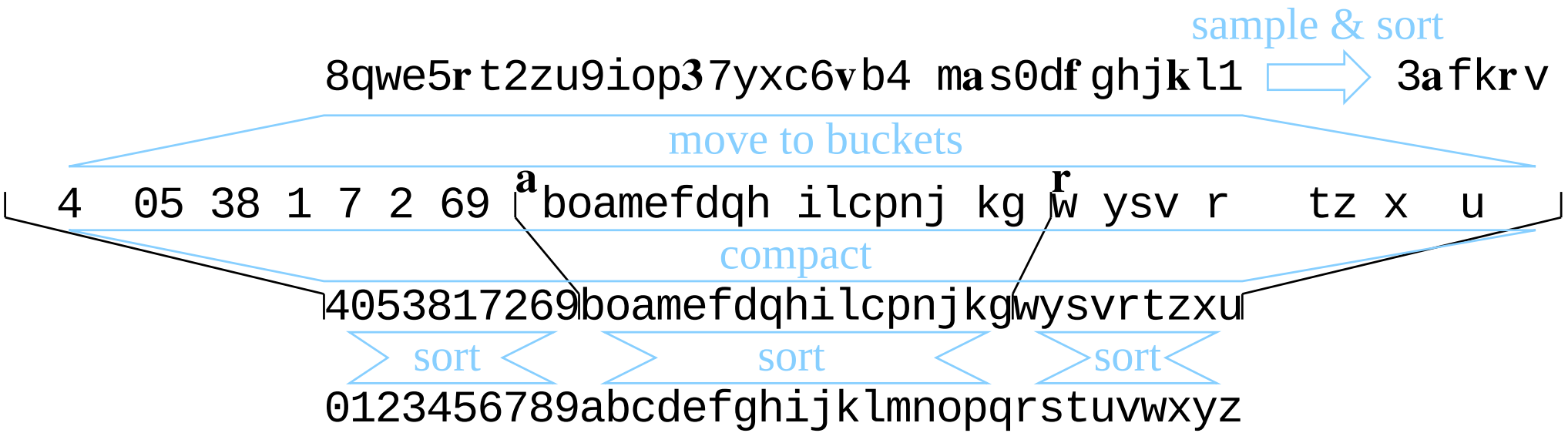
return ℓ

CRCW Sortieren in logarithmischer Zeit

Sei $n = p$.

- sample der Größe $\sqrt{p}$
- $k = \Theta\left(\sqrt{p}/\log p\right)$ splitter
- Buckets haben Größe $\leq cp/k$ Elements mhW
- Alloziere Buckets der Größe $2cp/k$
- Schreibe Elemente an zufällige freie Position in ihrem Bucket
- Kompaktiere mittels Präfixsummen
- Rekursion

Beispiel


sample & sort
8qwe5rt2zu9iop37yxc6vb4 mas0dfghjkl1
3afkrv
move to buckets
4 05 38 1 7 2 69 aboamefdqh ilcpnj kg rw ysv r tz x u
compact
4053817269boamefdqhilcpnjkgwysvrtzxu
sort
sort
sort
0123456789abcdefghijklmnopqrstuvwxyz

Mehr zu Sortieren I

Cole's merge sort: [JáJá Section 4.3.2]

Zeit $\mathcal{O}\left(\frac{n}{p}+\log p\right)$ deterministisch, EREW PRAM (CREW in [JáJá]). Idee: Pipelined parallel merge sort. Nutze (deterministisches) sampling zur Vorhersage wo die Daten herkommen.

Sorting Networks: Knoten sortieren 2 Elemente. Einfache Netzwerke $\mathcal{O}\left(\log^2 n\right)$ (z.B. bitonic sort) ergeben brauchbare deterministische Sortieralgorithmen (2 Elemente $\rightsquigarrow$ merge-and-split zweier sortierter Folgen). Sehr komplizierte mit Tiefe $\mathcal{O}(\log n)$.

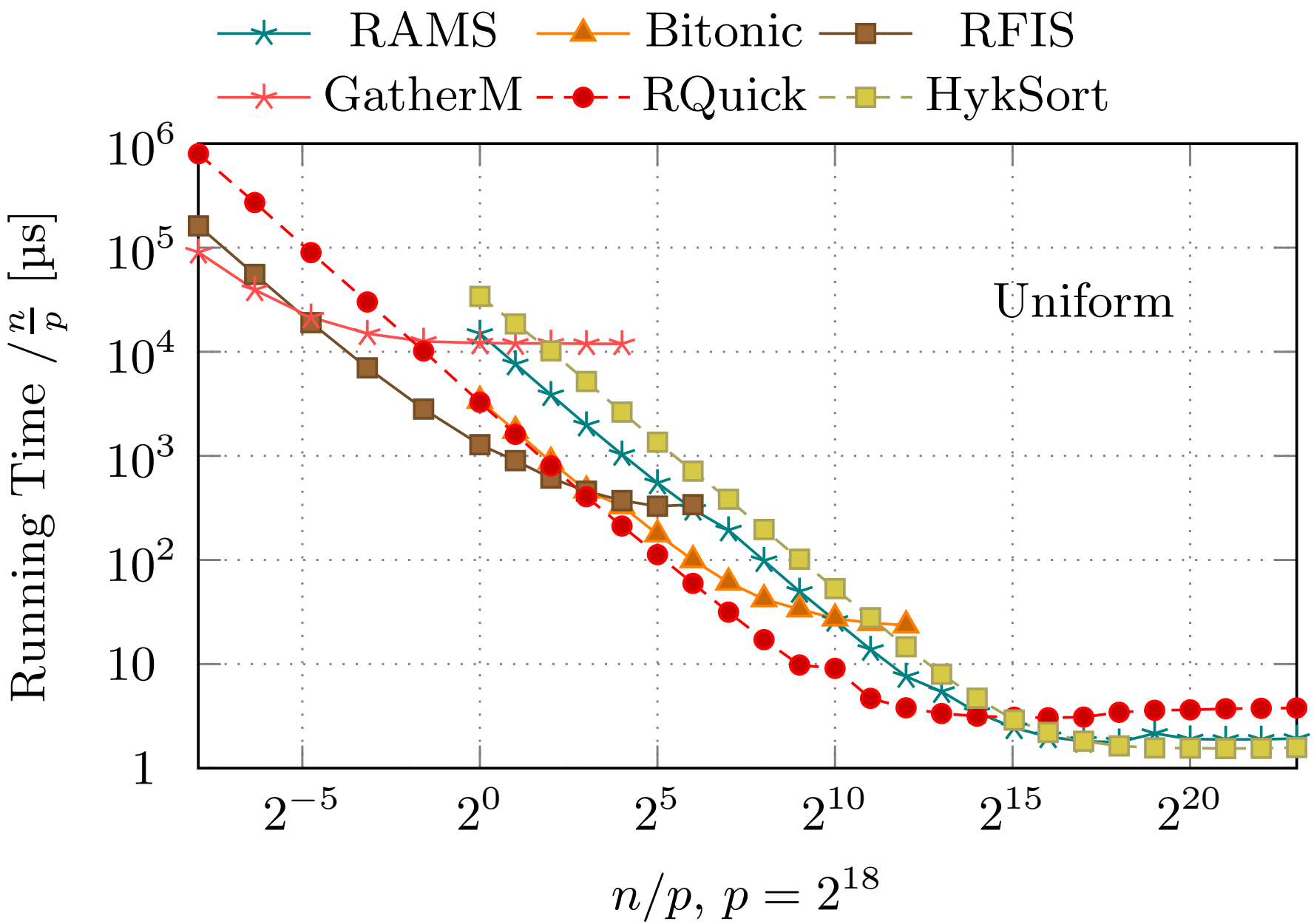
Mehr zu Sortieren II

Integer Sorting: (Annähernd) lineare Arbeit. Sehr schnelle Algorithmen auf CRCW PRAM.

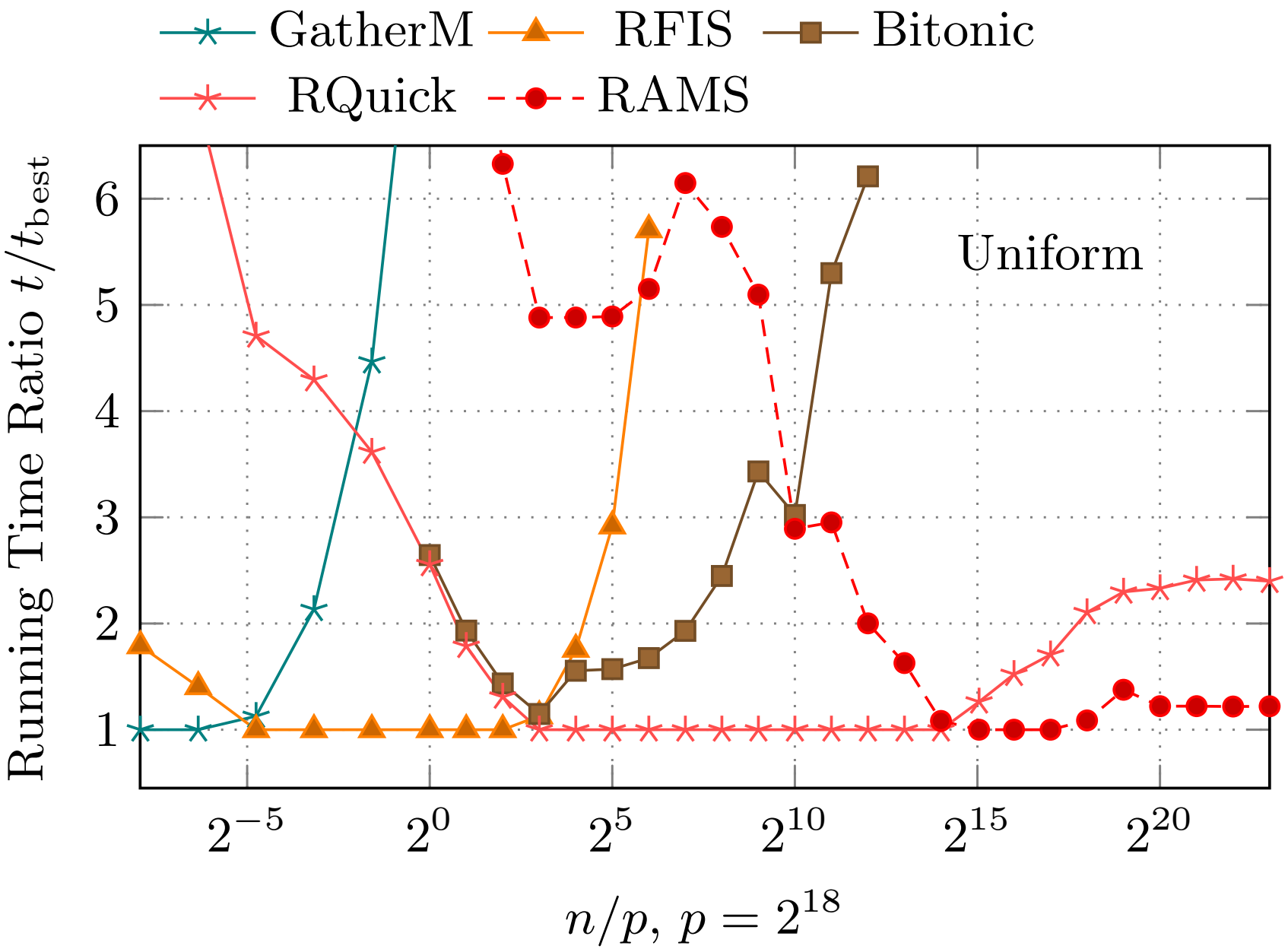
Mehr-Phasen-Sample/Merge-Sort: allgemeinerer Kompromiss zwischen Latenz und Kommunikationsvolumen, z.B. AMS-Sort

Axtmann, Bingmann, Schulz, Sanders SPAA 2015

Messungen Axtmann Sanders ALENEX 2017



Slowdown wrt Fastest Algorithm



Programmieraufgabe

Implementieren Sie einen parallelen Sortieralgorithmus.

Kombinationen aus $A \times B$ mit.

A: binary mergesort, multiway mergesort, hypercube quicksort, quicksort + sample-sort, shizophrenic quicksort, fast inefficient sort, CRCW logarithmic, sample-sort, Batcher sort, radix sort

B: MPI, C++11, Java, TBB, Cilk, OpenMP, CUDA, OpenCL, ...

Vergleichen Sie codes auf vergleichbaren Plattformen.

Separates Projekt: Vergleich mit existierenden parallelen Sortierern (TBB, stdlib,...).

Kollektive Kommunikation

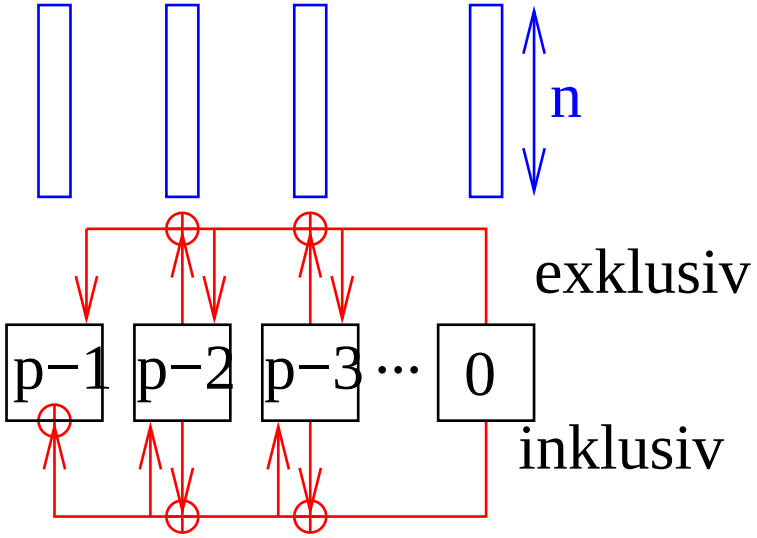
- Broadcast
- Reduktion
- Präfixsummen
- nicht hier: Sammeln / Austeilen (Gather / Scatter)
- Gossiping ($=$ All-Gather $=$ Gather $+$ Broadcast)
- All-to-all Personalized Communication
  - gleiche Nachrichtenlängen
  - ungleiche Nachrichtenlängen, $=h$-Relation

Präfixsummen

[Leighton 1.2.2] Gesucht

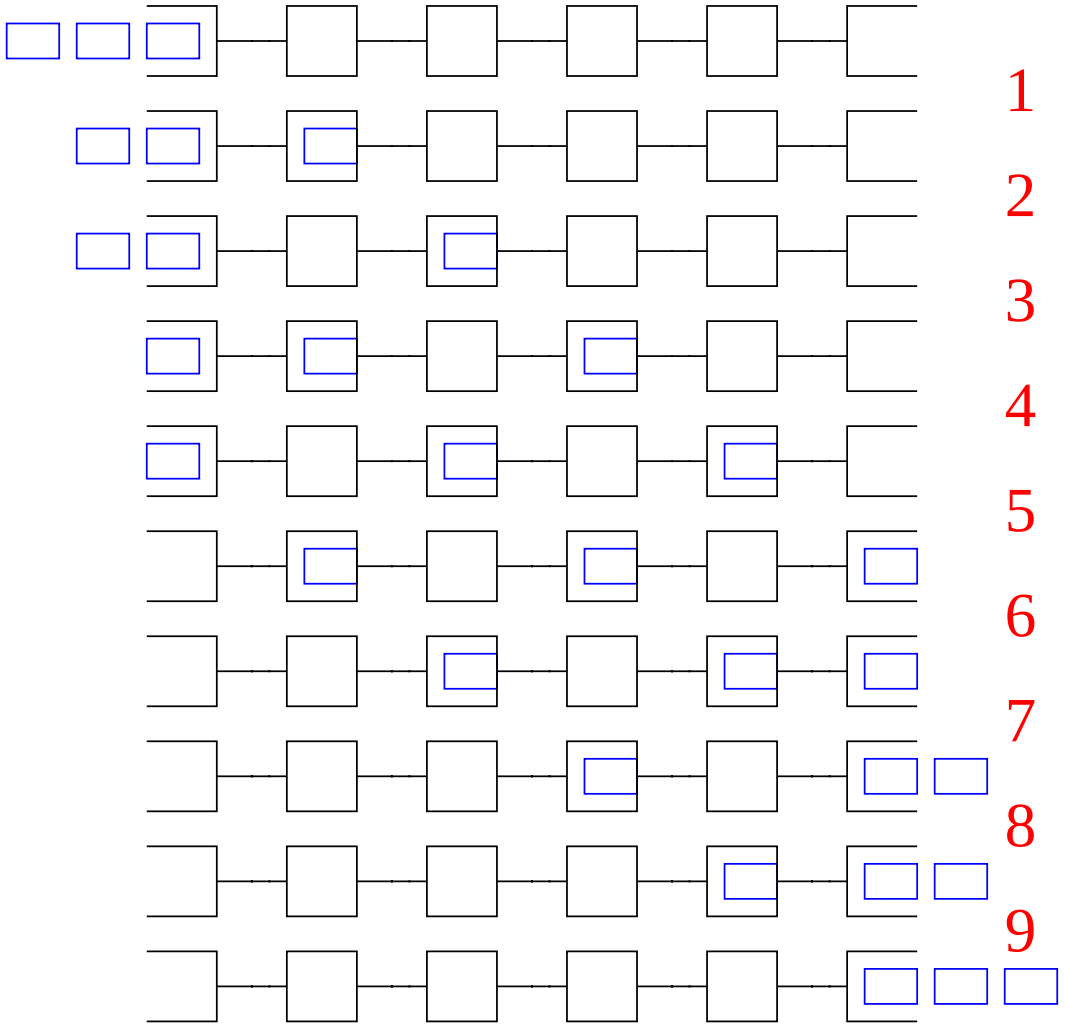
$$x@i := \bigotimes_{i' \leq i} m@i'$$

(auf PE i, m kann ein Vektor mit n Bytes sein.)

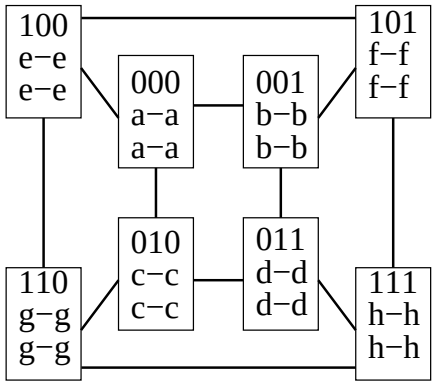


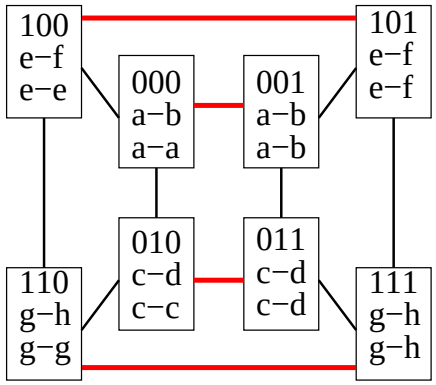
Einfache Pipeline

Wie bei Broadcast

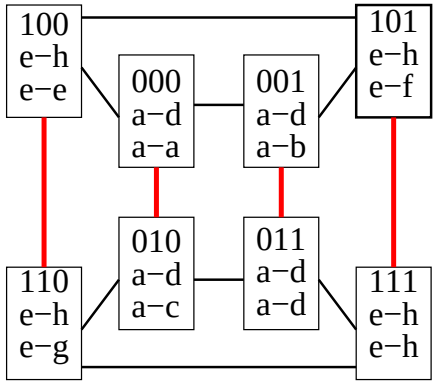


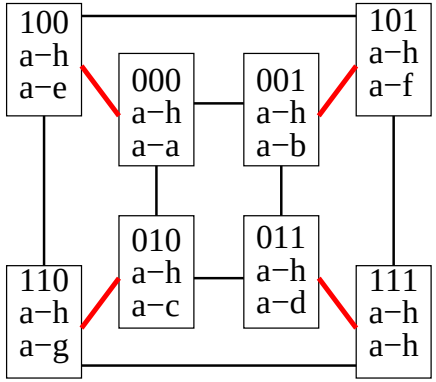
Hyperwürfelalgorithmus





i
sum
x





// view PE index i as a

// d-bit bit array

Function hcPrefix(m)

$x := \sigma := m$

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

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

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

$y := \sigma@(i \oplus 2^k)$ // sendRecv

$\sigma := \sigma \otimes y$

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

return x

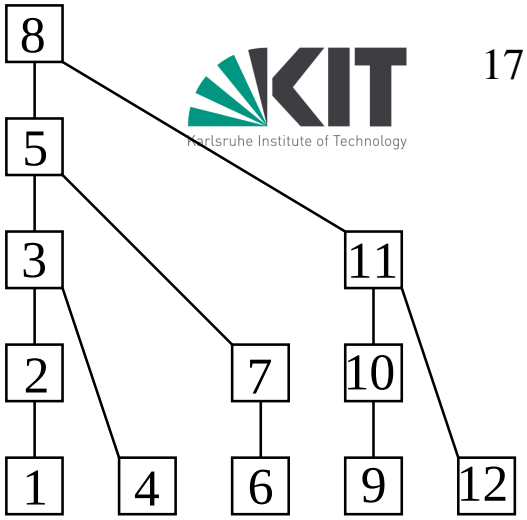
Analyse

Telefonmodell:

$$T_{\text{prefix}} = (\alpha + n\beta)\log p$$

Pipelining klappt nicht, da alle PEs immer beschäftigt.

Pipeline-Binärbaum-Präfixsummen



Infix Nummerierung (in order) der Knoten

Aufwärtsphase: wie bei Reduktion aber

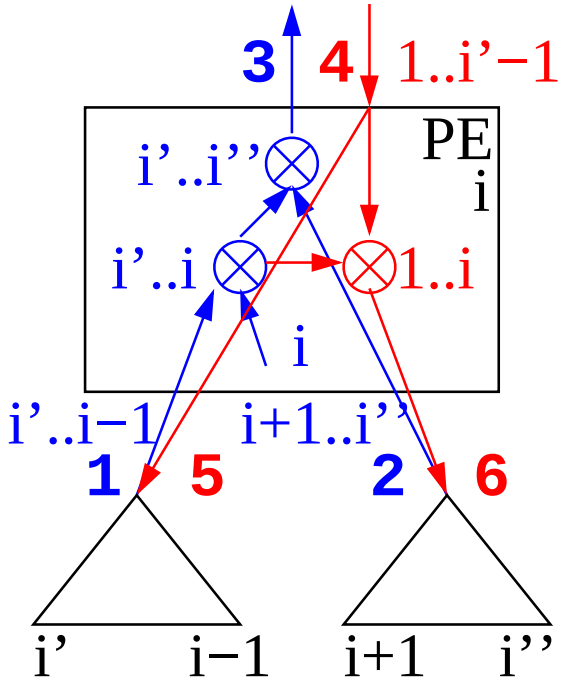
PE i speichert $\sum_{j=i'}^{i} x@j$

Abwärtsphase: PE i empfängt $\sum_{j=1}^{i'-1} x@j$

(Wurzel: $= 0$!)

und reicht das nach links weiter.

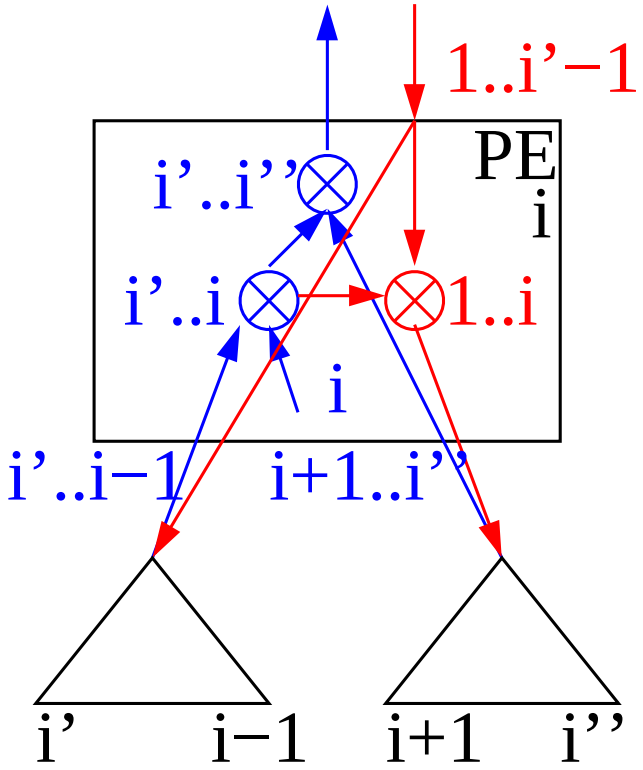
rechter Teilbaum kriegt $\sum_{j=1}^{i} x@j$

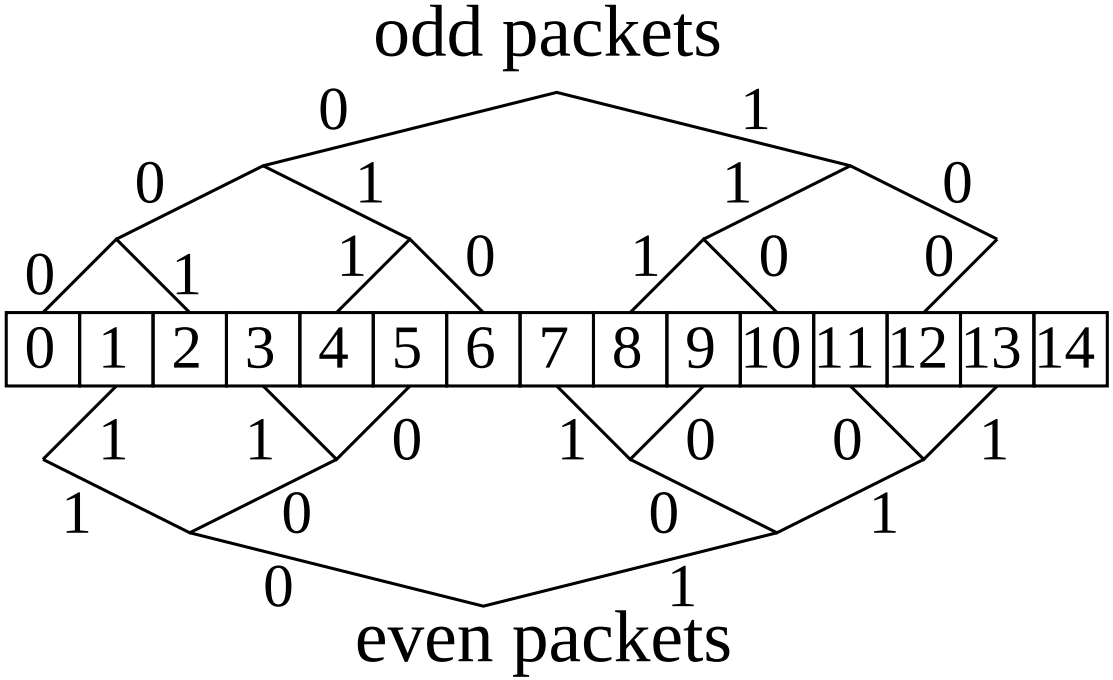


Jedes PE nur $1\times$ je Phase aktiv. $\rightarrow$ Pipelining OK

23-Präfixsummen

Nummerierung ist Inorder-Nummerierung für beide Bäume !





Analyse

$$T_{\text{prefix}} \approx T_{\text{reduce}} + T_{\text{broadcast}} \approx 2T_{\text{broadcast}} =$$
$$2n\beta + \alpha \cdot 4\log p + \sqrt{8n\log p\alpha\beta}$$

Latenz senken durch überlappen von Aufwärts und Abwärtsphase?

Verallgemeinerung:

- Beliebige auf inorder nummerierten Bäumen arbeitende Algorithmen einsetzbar

$\rightsquigarrow$ ESBT funktioniert nicht?

Gossiping

Jedes PE hat eine Nachricht m der Länge n.

Am Ende soll jedes PE alle Nachrichten kennen.

Hyperwürfelalgorithmus

Sei '$\cdot$' die Konkatenationsoperation; $p = 2^d$

PE i

$y := m$

for $0 \leq j < d$ **do**

$y' :=$ the y from PE $i \oplus 2^j$

$y := y \cdot y'$

return y

Analyse

Telefonmodell, $p = 2^d$ PEs, n Byte pro PE:

$$T_{\text{gossip}}(n,p) \approx \sum_{j=0}^{d-1} \alpha + n \cdot 2^j \beta = \log p \alpha + (p-1) n \beta$$

All-Reduce

Reduktion statt Konkatenation.

Vorteil: Faktor zwei weniger Startups als Reduktion plus Broadcast

Nachteil: $p \log p$ Nachrichten.

Das ist ungünstig bei stauanfälligen Netzwerken.

All-to-all Personalized Communication

Jedes PE hat $p-1$ Nachrichten der Länge n. Eine für jedes andere PE. Das lokale $m[i]$ ist für PE i

Hyperwürfelalgorithmus

PE i

for $j := d-1$ **downto** 0 **do**

Get from PE $i \oplus 2^j$ all its messages

destined for my j-D subcube

Move to PE $i \oplus 2^j$ all my messages

destined for its j-D subcube

Analyse, Telefonmodell:

$$T_{\text{all-to-all}}(p,n) \approx \log p(\frac{p}{2}n\beta + \alpha)$$

vollständige Verknüpfung:

Bei großem n Nachrichten lieber einzeln schicken

(Faktor $\log p$ weniger Kommunikationsvolumen)

Der 1-Faktor-Algorithmus

[König 1936]

p ungerade:

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

for $i := 0$ **to** $p-1$ **do**

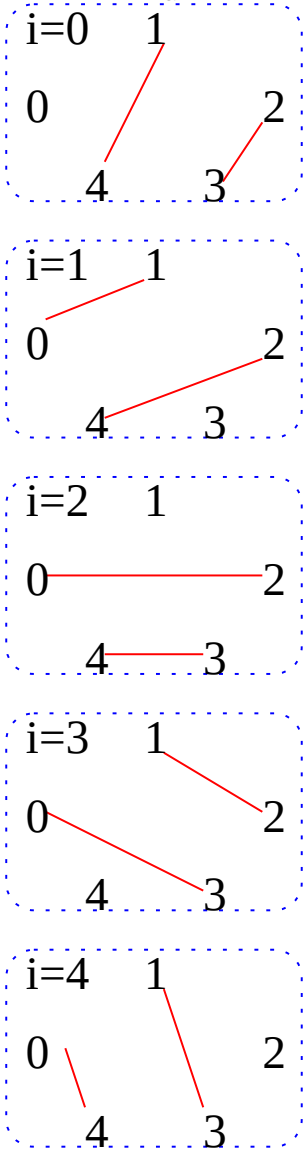
Exchange data with PE $(i-j) \bmod p$

Paarweise Kommunikation (Telefonmodell):

Der Partner des Partners von j in Runde i ist

$i-(i-j) \equiv j \bmod p$

Zeit: $p(n\beta+\alpha)$ optimal für $n \to \infty$

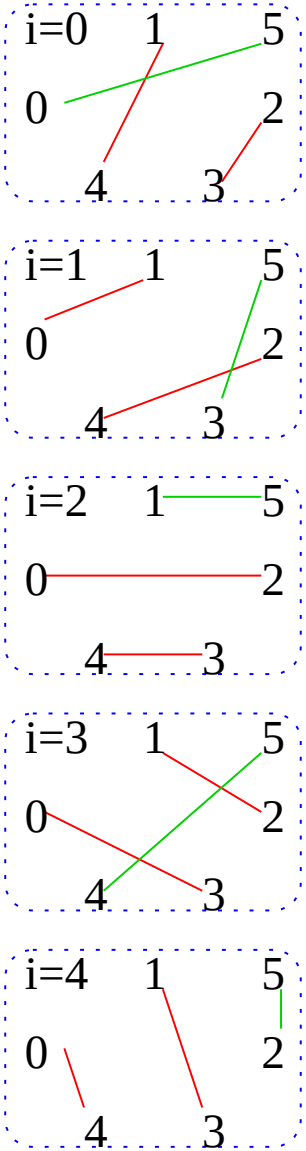


Der 1-Faktor-Algorithmus

p gerade:

//PE index $j \in \{0,\ldots,p-1\}$
for $i := 0$ **to** $p-2$ **do**
  idle:= $\frac{p}{2} i \bmod (p-1)$
  if $j = p-1$ **then** exchange data with PE idle
  else
    if $j =$ idle **then**
      exchange data with PE $p-1$
    else
      exchange data with PE $(i-j) \bmod (p-1)$

Zeit: $(p-1)(n\beta+\alpha)$ optimal für $n \to \infty$



Datenaustausch bei unregelmäßigen Nachrichtenlängen

- Vor allem bei all-to-all interessant $\rightarrow$ Sortieren
- Ähnliche Probleme bei inhomogenen Verbindungsnetzwerken oder Konkurrenz durch andere Jobs.

Der Vogel-Strauß-Algorithmus

Alle Nachrichten mit asynchronen Sendeoperationen

“ins Netz stopfen”.

Alles Ankommende empfangen

Vogel-Strauß-Analyse:

BSP-Modell: Zeit $L + gh$

Aber was ist L und g in Single-Ported Modellen?(jetzt)

Oder gleich in realen Netzwerken? (später)

h-Relation

$h_{\text{in}}(i)$:= Anzahl empfangener Pakete von PE i

$h_{\text{out}}(i)$:= Anzahl gesendeter Pakete von PE i

simplex: $h := \max_{i=1}^{p} h_{\text{in}}(i) + h_{\text{out}}(i)$

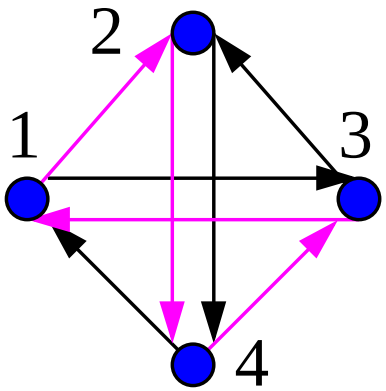
duplex: $h := \max_{i=1}^{p} \max(h_{\text{in}}(i), h_{\text{out}}(i))$

Untere Schranke bei paketweiser Auslieferung:

h Schritte, d.h.,

Zeit $h(\alpha + |\text{Paket}|\beta)$

Offline h-Relationen im duplex Modell

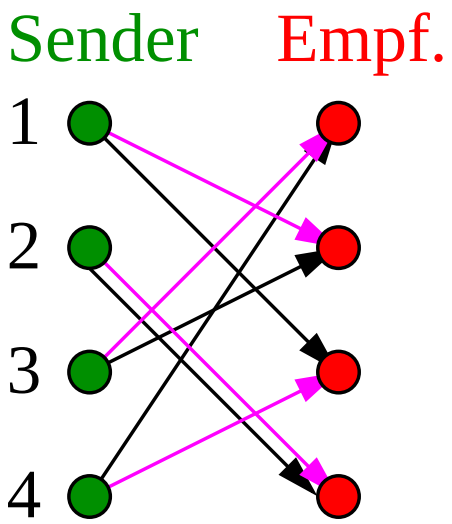


[König 1916]

Betrachte den bipartiten Multigraph

$G = (\{s_1, \ldots, s_p\} \cup \{r_1, \ldots, r_p\}, E)$ mit

$|\{(s_i, r_j) \in E\}| =$ # Pakete von PE i nach PE j.



Satz: $\exists$ Kantenfärbung $\phi : E \to \{1..h\}$, d.h.,

keine zwei gleichfarbigen Kanten

inzident zu einem Knoten.

for $j := 1$ **to** h **do**

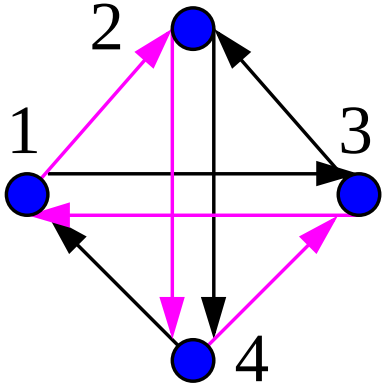
Sende Nachrichten der Farbe j

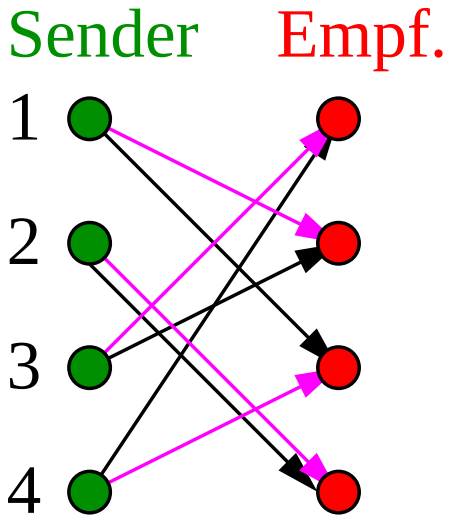
optimal wenn man paketweise Auslieferung postuliert

Offline h-Relationen im duplex Modell

Probleme:

- Kantenfärbung online berechnen
  ist kompliziert und teuer
- Aufteilung in Pakete erhöht Anzahl Startups





Offline h-Relationen im Simplex-Modell

[Petersen 1891? Shannon 1949?]

Betrachte den Multigraph $G = (\{1,\ldots,p\}, E)$

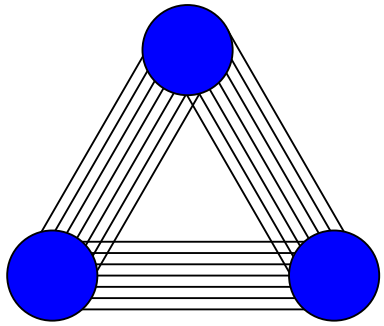
mit $|\{\{i,j\} \in E\}| =$ # Pakete zwischen PE i und PE j (beide Richtungen).

Satz: $\exists$ Kantenfärbung $\phi : E \to \{1..3\lfloor h/2 \rfloor + h \bmod 2\}$

for $j := 1$ **to** h **do**

Sende Nachrichten der Farbe j

optimal??



How Helper Hasten h-Relations

[Sanders Solis-Oba 2000]

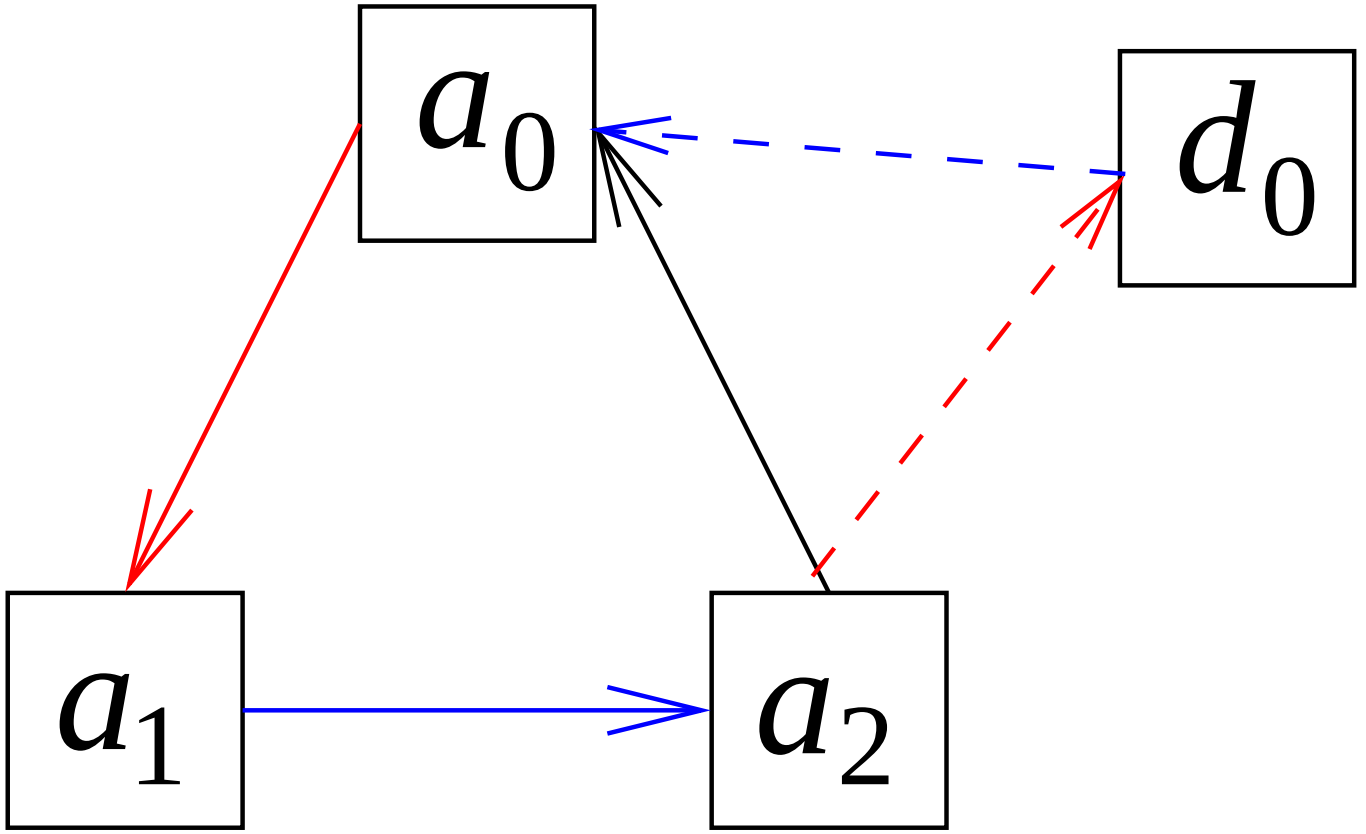
Satz 4. *Für h-Relationen im Simplexmodell gilt*

$$\#\text{steps} = \begin{cases} \frac{6}{5}(h+1) & \text{falls } P \text{ gerade} \\ (\frac{6}{5}+\frac{2}{P})(h+1) & \text{falls } P \text{ ungerade} \end{cases} .$$

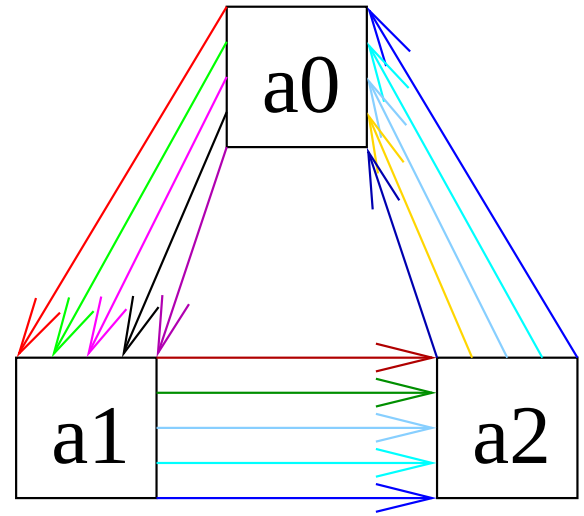
Andererseits gibt es eine untere Schranke

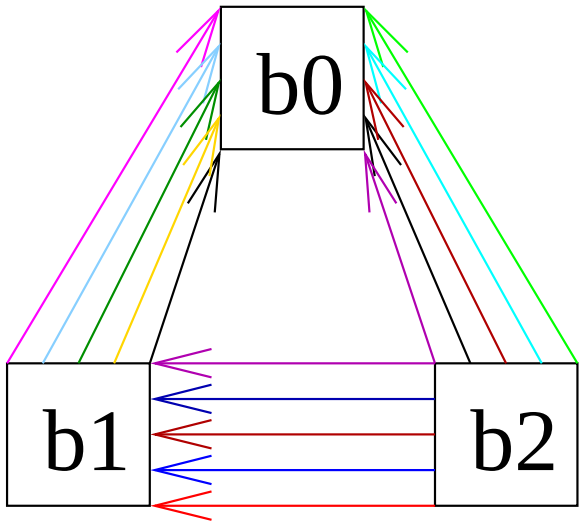
$$\#\text{steps} \geq \begin{cases} \frac{6}{5}h & \text{falls } P \text{ gerade} \\ (\frac{6}{5}+\frac{18}{25P})h & \text{falls } P \text{ ungerade} \end{cases}$$

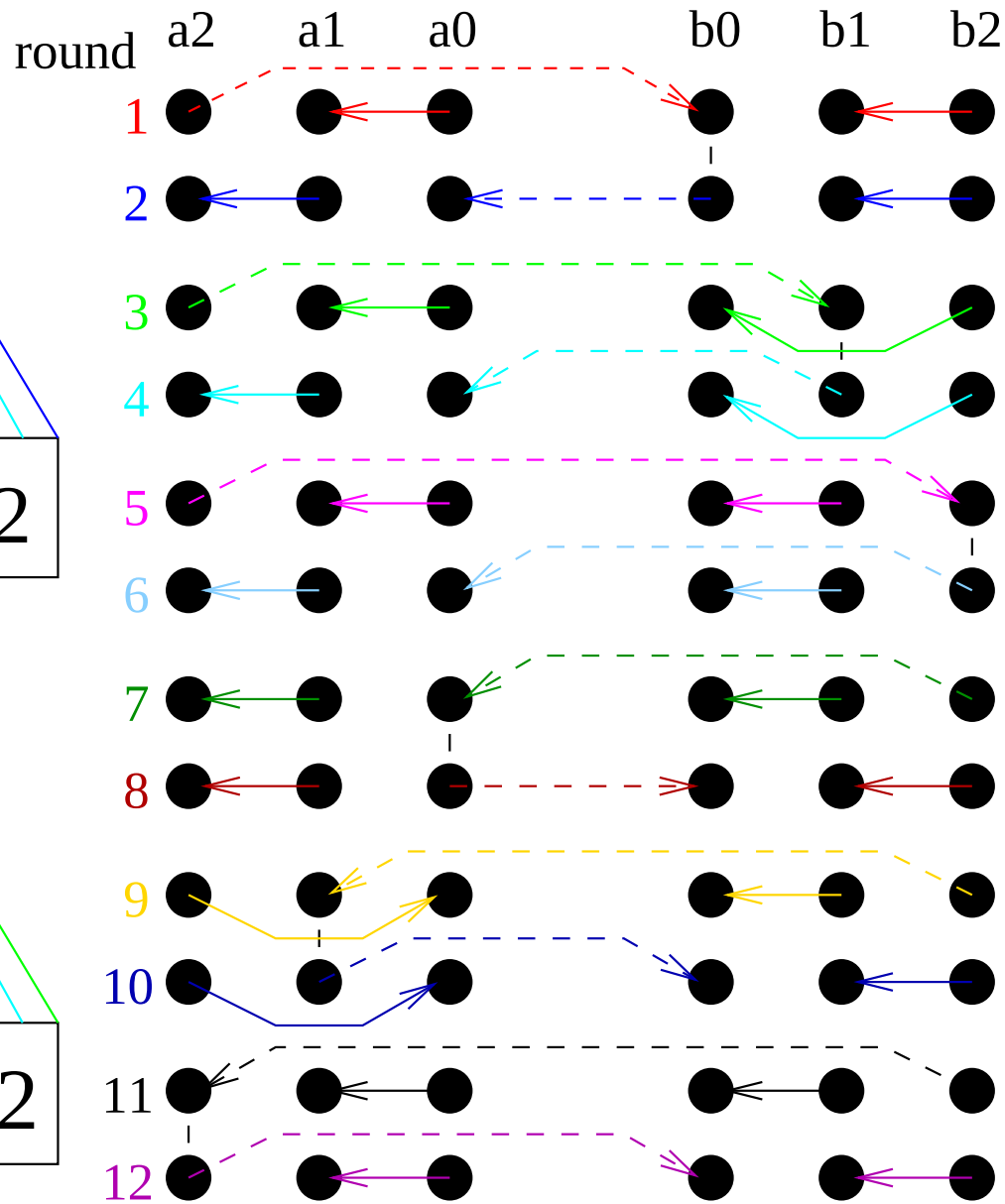
Ein ganz simpler Fall



Zwei Dreiecke







Reduktion h-Relation$\rightsquigarrow \left\lceil \frac{h}{2} \right\rceil$ 2-Relationen

- Kommunikationsrichtung erstmal ignorieren
- Verbinde Knoten mit ungeradem Grad $\rightsquigarrow$ alle Knoten haben geraden Grad
- Eulertourtechnik: Zerlege Graph in kantendisjunkte Kreise
- Kreise im Urzeigersinn ausrichten $\rightsquigarrow$ Eingangsgrad und Ausgangsgrad $\leq \lceil h/2 \rceil$
- Baue bipartiten Graphen (wie gehabt)
- Färbe bipartiten Graphen
- Farbklasse in bipartitem Graph $\rightsquigarrow$ kantendisjunkte einfache Kreise im Ursprungsgraphen (2-Relationen)
- Ursprüngliche Kommunikationsrichtung wiederherstellen

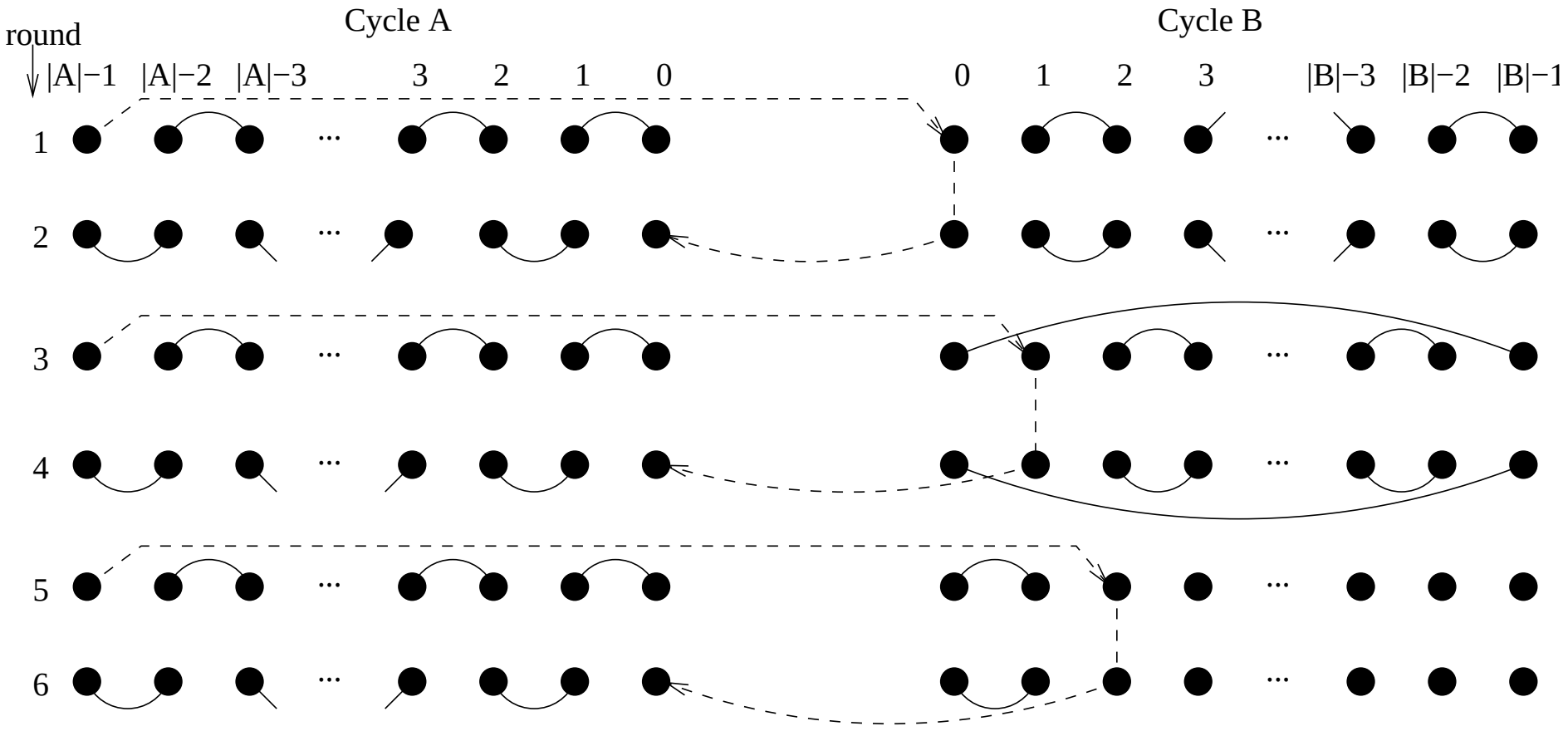
2-Relationen routen für gerade p

Paare ungerade Kreise.

1 Kreise haben nichts zu tun $\rightsquigarrow$ einfachster Fall

Zwei Ungerade Kreise mit ≥ 3 Knoten

Spalte Pakete in 5 Teilpakete



Dann das ganze umdrehen

Ungerade p

Idee: Lösche in jedem 2-Faktor eine Kante.

Tu dies “Immer woanders”

Sammle $\Theta(P)$ gelöschte Kanten in einem Matching

$\rightsquigarrow$ ein zusätzlicher Schritt pro $\Theta(P)$ 2-Faktoren.

Offene Probleme

- Aufspaltung in 5 Teilpakete loswerden?
- Vermutung:

  Eine h-Relation mit $\leq \frac{3}{8}hP$ Paketen kann in $\approx h$ Schritten ausgeliefert werden.
- Startupoverheads explizit berücksichtigen.
- Verbindungsnetzwerk explizit berücksichtigen?
- Verteiltes Scheduling

Ein einfacher verteilter Algorithmus — Der Zweiphasenalgorithmus

Idee: Irreg. All-to-all$\rightarrow 2\times$ regular All-to-all

Vereinfachende Annahmen:

- Alle Nachrichtenlängen durch p teilbar
  (Im Zweifel aufrunden)
- Kommunikation “mit sich selbst” wird mitgezählt
- Alle PEs senden und empfangen genau h Byte
  (Im Zweifel “padding” der Nachrichten)

// $n[i]$ is length of message $m[i]$

Procedure alltoall2phase$(m[1..p], n[1..p], p)$

  for $i := 1$ **to** p **do** $a[i] := \langle\rangle$

    for $j := 1$ **to** p **do** $a[i] := a[i] \odot m[j][(i-1)\frac{n[j]}{p} + 1..i\frac{n[j]}{p}]$

  $b :=$ regularAllToAll(a, h, p)

  $\delta := \langle 1, \ldots, 1 \rangle$

  for $i := 1$ **to** p **do** $c[i] := \langle\rangle$

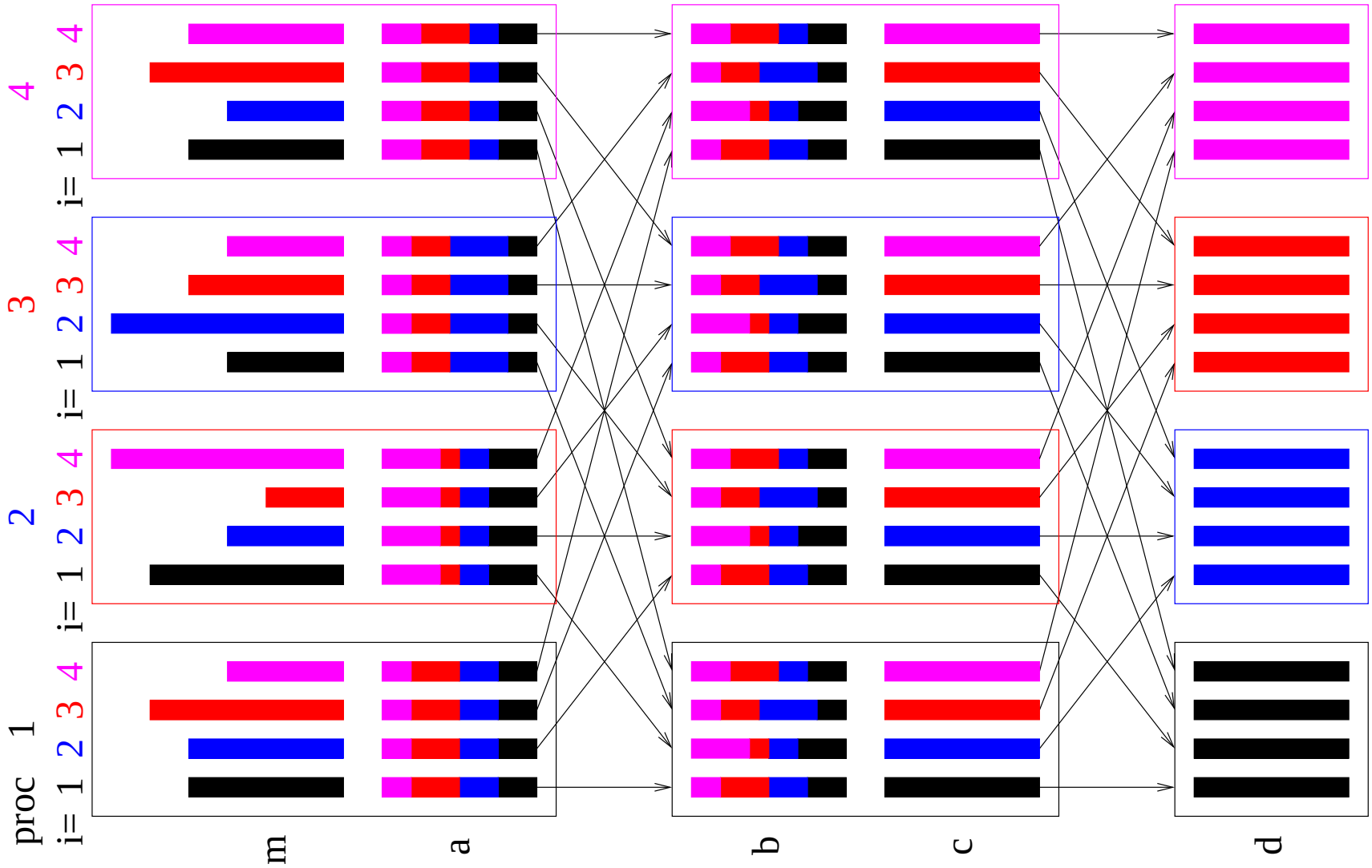
    for $j := 1$ **to** p **do**

      $c[i] := c[i] \odot b[j][\delta[j]..\delta[j] + \frac{n[i]@j}{p} - 1]$ // Use All-

      $\delta[j] := \delta[j] + \frac{n[i]@j}{p}$ // gather to implement '@'

  $d :=$ regularAllToAll(c, h, p)

  permute d to obtain the desired output format


proc
i= 1 2 3 4
1
i= 1 2 3 4
2
i= 1 2 3 4
3
i= 1 2 3 4
4
m
a
b
c
d

Mehr zum Zweiphasenalgorithmus

- Grosses p, kleine Nachrichten $\rightsquigarrow$
  lokale Daten in $\mathcal{O}(p \log p)$ Stücke aufteilen (nicht p^2) und zufällig verteilen.
- Aufspaltung des Problems in regelmäßigen und unregelmäßigen Teil $\rightsquigarrow$ nur ein Teil der Daten wir Zweiphasenprotokoll unterzogen.
  $\rightsquigarrow$ offenes Problem: wie aufspalten?

Ein nichtpräemptiver offline Algorithmus (simplex)

[Sanders Solis-Oba 99, unveröffentlicht]

Ziel: alle Nachrichten direkt, als Ganzes ausliefern.

Sei k:= Max. # Nachrichten an denen ein PE beteiligt ist.

Zeit für Ausführung des Schedule $k\alpha + 2h\beta$

hier ist h in Byte gemessen!

Abstrakte Beschreibung

s:= empty schedule

M:= set of messages to be scheduled

while $M \neq \emptyset$ **do**

$t := \min\{t : \exists m \in M : m$'s src and dest are idle at time $t\}$

$s := s \cup$ "start sending m at time t"

$M := M \setminus \{m\}$

Kann implementiert werden, so dass pro Nachricht Zeit für $\mathcal{O}(1)$ Prioritätslistenoperationen und eine p-bit Bitvektoroperation anfällt.

$\rightsquigarrow$ praktikabel für Nachrichtenlängen $\gg p$ und moderate p.

Offene Probleme zum nichtpräemptiven offline Algorithmus

- implementieren, ausmessen, verwenden, z.B. sortieren, Konstruktion v. Suffix-Arrays
- Bessere Approximationsalgorithmen?
- Parallele Scheduling-Algorithmen

Zusammenfassung: All-to-All

Vogel-Strauss: Abwälzen auf online, asynchrones Routing.

Gut wenn das gut implementiert ist.

Regular+2Phase: Robustere Lösung. Aber, Faktor 2 stört, viel Umkopieraufwand.

Nichtpräemptiv: Minimiert Startups, Kommunikationsvolumen. Faktor 2 (worst case). Zentralisierte Berechnung stört.

Gut bei wiederholten identischen Problemen.

Färbungsbasierte Algorithmen: Fast optimal bei großen Paketen.

Komplex. Verteilte Implementierung? Aufspalten in Pakete stört.

Vergleich von Ansätzen?

Parallele Prioritätslisten

Verwalte eine Menge M von Elementen. $n = |M|$. Anfangs leer

Binary Heaps (sequentiell)

Procedure insert(e) $M := M \cup \{e\}$ // $\mathcal{O}(\log n)$

Function deleteMin $e := \min M$; $M := M \setminus \{e\}$; **return** e // $\mathcal{O}(\log n)$

Parallele Prioritätslisten, Ziel

insert*: Jedes PE fügt konstant viele Elemente ein,

Zeit $\mathcal{O}(\log n + \log p)$?

deleteMin*: lösche die p kleinsten Elemente,

Zeit $\mathcal{O}(\log n + \log p)$?

Nicht hier: asynchrone Variante: Jeder kann jederzeit einfügen oder deleteMin machen.

Semantik: $\exists$ zeitliche Anordnung der Operationen, die mit der sequentiellen Queue übereinstimmt.

Anwendungen

- Prioritätsgestriebenes Scheduling von unabhängigen Jobs
- Best first Branch-and-bound:

  Finde beste Lösung in einem großen, implizit definierten Baum.

  (später mehr)
- Simulation diskreter Ereignisse

Naive Implementierung

PE 0 verwaltet eine sequentielle Prioritätsliste

Alle anderen stellen Anfragen

insert: $\Omega\left(p(\alpha+\log n)\right)$

deleteMin: $\Omega\left(p(\alpha+\log n)\right)$

Branch-and-Bound

H: Baum (V,E) mit beschränktem maximalen Knotengrad

$c(v)$: Knotenkosten — steigen auf jedem Abwärtspfad monoton an

v^*: Blatt mit minimalen Kosten

$\tilde{V}$: $\{v \in V : v \leq v^*\}$

m: $|\tilde{V}|$ Vereinfachung: $\Omega(p \log p)$

h: Tiefe von $\tilde{H}$ (durch $\tilde{V}$ knoteninduzierter Teilgraph von H).

T_{x} Zeit für Generierung der Nachfolger eines Knotens

T_{coll} obere Schranke für Broadcast, Min-Reduktion, Prefix-Summe, routing ein Element von/zu zufälligem Partner.
$\mathcal{O}(\alpha \log p)$ auf vielen Netzwerken.

Sequentielles Branch-and-Bound

$Q = \{\text{root node}\}$: PriorityQueue // frontier set

$c^* = \infty$ // best solution so far

while $Q \neq \emptyset$ **do**

select some $v \in Q$ and remove it

if $c(v) < c^*$ **then**

if v is a leaf node **then** process new solution; $c^* := c(v)$

else insert successors of v into Q

$T_{\text{seq}} = m(T_{\text{x}} + \mathcal{O}(\log m))$

Paralleles Branch-and-Bound

$Q = \{$root node$\}$: ParallelPriorityQueue

$c^* = \infty$ // best solution so far

while $Q \neq \emptyset$ **do**

  v:= Q.deleteMin$*$ // SPMD!

  if $c(v) < c^*$ **then**

    if v is a leaf node **then**

      process new solution

      update c^* // Reduction

    else insert successors of v into Q

Analyse

Satz: $T_{\text{par}} = (\frac{m}{p} + h)(T_{\text{x}} + \mathcal{O}\left(T_{\text{queueOp}}\right))$

Fall 1 (höchstens m/p Iterationen): Alle bearbeiteten Knoten sind in $\tilde{V}$

Fall 2 (höchstens h Iterationen): Knoten ausserhalb von $\tilde{V}$ werden bearbeitet $\rightarrow$ die maximale Pfadlänge von einem Knoten in Q zur optimalen Lösung wird reduziert.

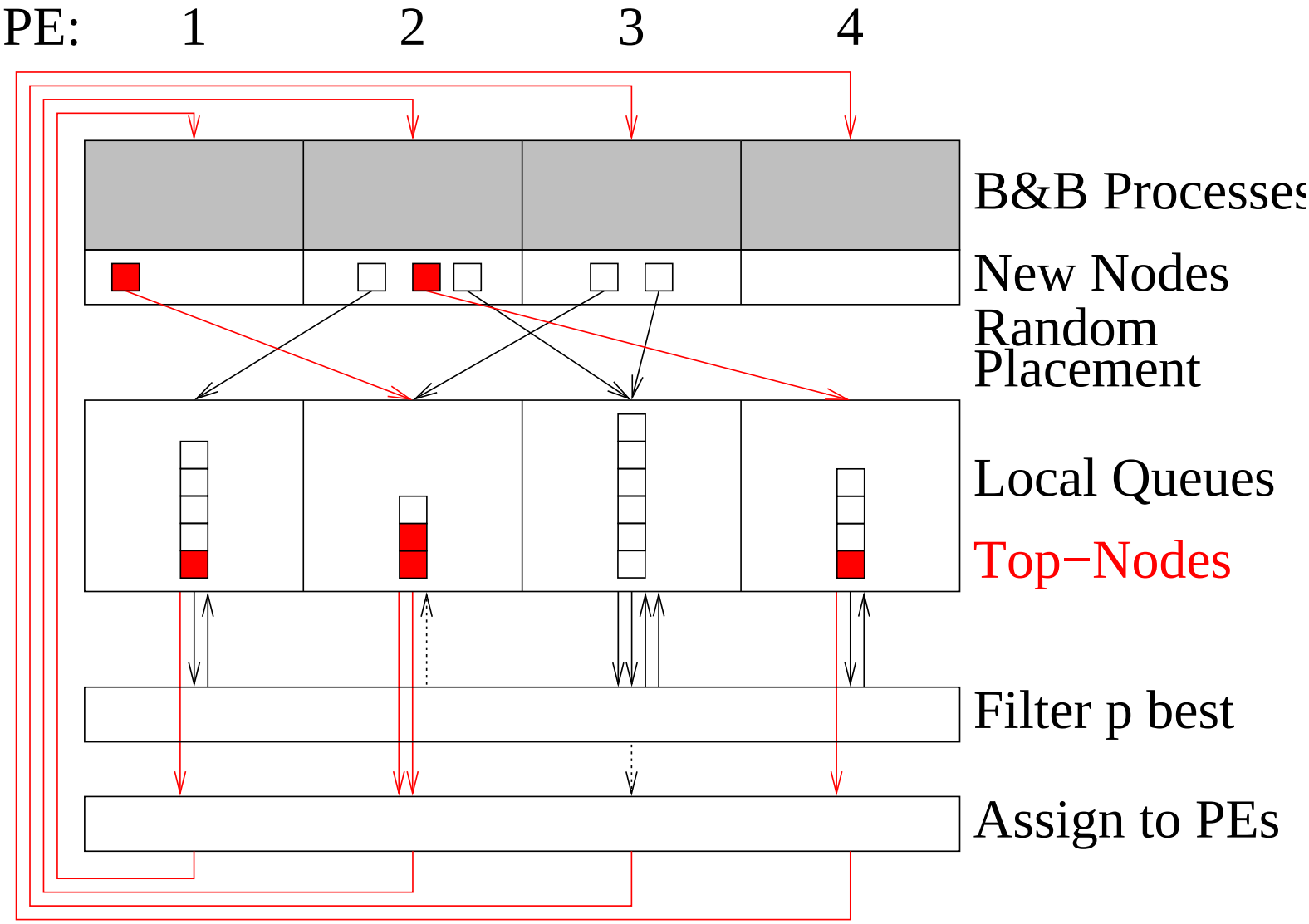
□

Der Algorithmus von Karp und Zhang

$Q = \{\text{root node}\}$: PriorityQueue // local!
$c^* = \infty$ // best solution so far
while $\exists i : Q@i \neq \emptyset$ **do**
    v:= Q.deleteMin$*$ // local!
    if $c(v) < c^*$ **then**
        if v is a leaf node **then**
            process new solution
            $c^* := \min_i c(v)@i$ // Reduction
        else for each successor v' of v **do**
            insert v into $Q@i$ for random i

Satz: Expected time is asymptotically optimal

Unser Ansatz



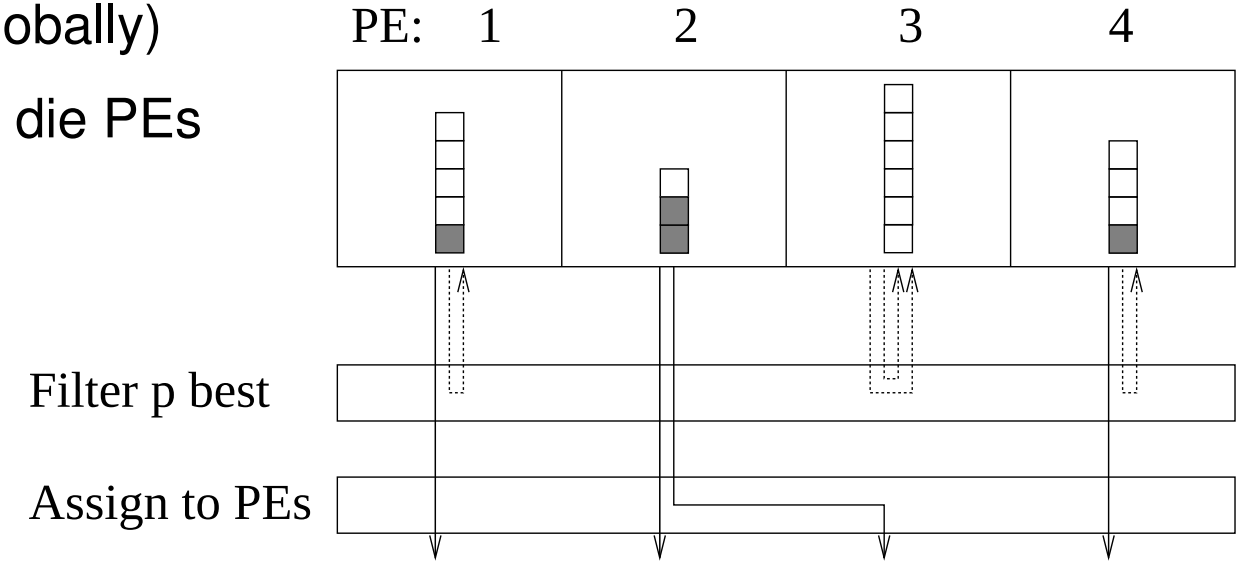
Parallele Prioritätslisten: Ansatz

- Die Queue ist die Vereinigung lokaler queues
- Einfügen schickt neue Elemente an zufällige lokale Queues

  Intuition: jedes PE braucht eine repräsentative Sicht der Dinge
- deleteMin* sucht die global kleinsten Elemente

  (act locally think globally)

  und verteilt sie auf die PEs



Einfache Probabilistische Eigenschaften

Mit hoher Wahrscheinlichkeit (mhw):

hier $\geq 1 - p^{-c}$ für eine Konstante c unserer Wahl)

- mhw nur $\mathcal{O}\left(\frac{\log p}{\log\log p}\right)$ Elemente pro lokaler Queue beim Einfügen
- mhw enthalten die $\mathcal{O}(\log p)$ kleinsten Elemente jeder lokalen queue die p global besten Elemente
- mhw enthält keine lokale queue mehr als $\mathcal{O}(n/p + \log p)$ Elemente

Beweis: Chernoff-Schranken rauf und runter.

(Standardsituation. Bälle in Kisten)

Parallele Realisierung I

Sei $T_{\text{coll}} :=$ obere Schranke für

Broadcast, Min-Reduktion, Prefix-Summe, routing ein Element von/zu zufälligem Partner.

$\mathcal{O}(\alpha \log p)$ auf vielen Netzwerken.

Einfügen

Verschicken: T_{coll}

Lokal einfügen: $\mathcal{O}\left(\frac{\log p}{\log\log p} \cdot \log \frac{n}{p}\right)$.

(Besser mit “fortgeschrittenen” lokalen queues. Vorsicht: amortisierte Schranken reichen nicht.)

Parallele Realisierung I

deleteMin*

Procedure deleteMin*(Q_1, p)

Q_0:= the $\mathcal{O}(\log p)$ smallest elements of Q_1

M:= select(Q_0, p) // später

enumerate $M = \{e_1, \ldots, e_p\}$

assign e_i to PE i // use prefix sums

if $\max_i e_i > \min_j Q_1 @ j$ **then** expensive special case treatment

empty Q_0 back into Q_1

Analyse

Lokal entfernen: $\mathcal{O}\left(\log p \log \frac{n}{p}\right)$

Selektion: $\mathcal{O}(T_{\text{coll}})$ mhw todo

M aufzählen: $\mathcal{O}(T_{\text{coll}})$

Ergebnisse ausliefern: $\mathcal{O}(T_{\text{coll}})$ (zufällige Quellen)

Verifizieren: $\mathcal{O}(T_{\text{coll}}) +$(etwas polynomiell in p)$\cdot$(eine polynomiell kleine Wahrscheinlichkeit)

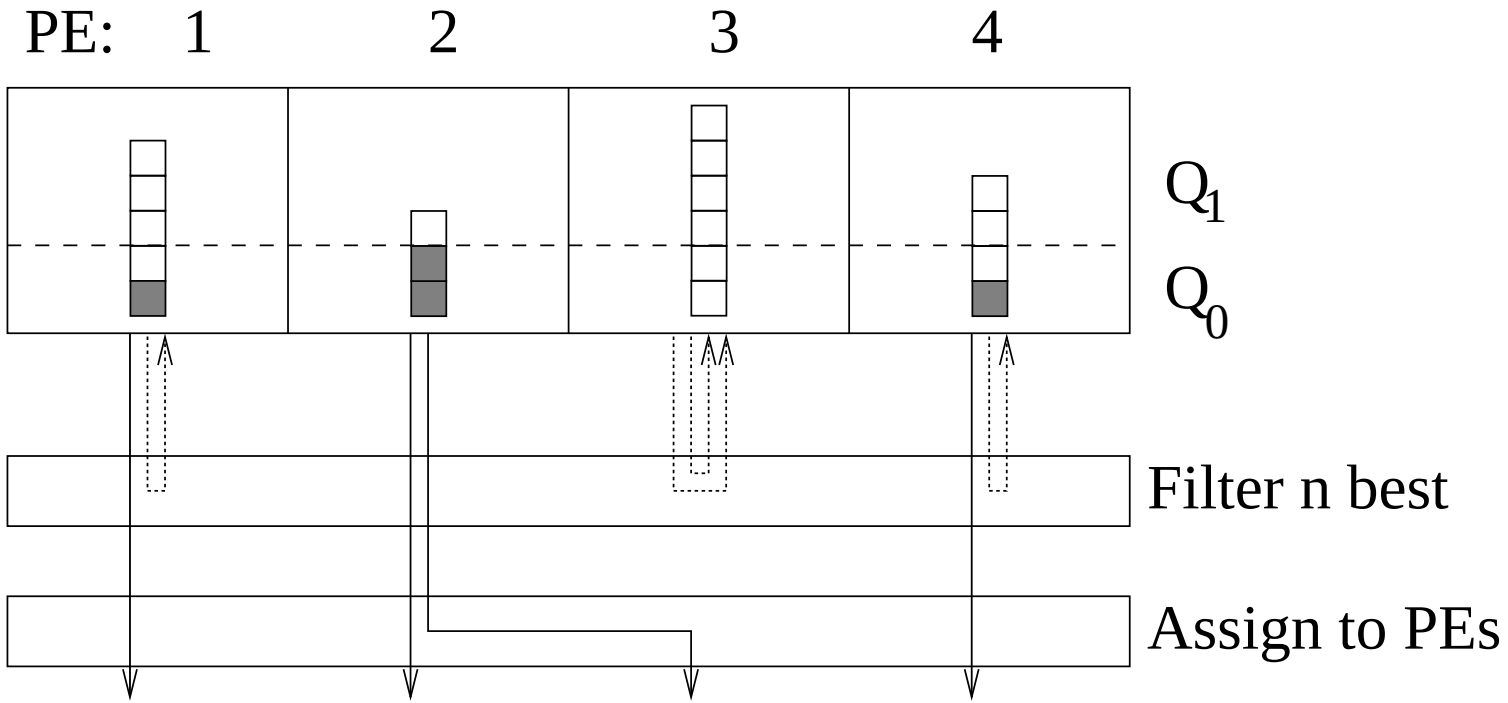
Lokal einfügen: $\mathcal{O}\left(\frac{\log p}{\log \log p} \log \frac{n}{p}\right)$

Parallele Realisierung II

Idee vermeide Ping-Pong der $\mathcal{O}(\log n)$ kleinsten Elemente.

Zweiteilung der queue in Q_0 und Q_1, $|Q_0| = \mathcal{O}(\log p)$.

Invariante: mhw $|Q_0| = \mathcal{O}(\log p)$



Parallele Realisierung II

Einfügen

Verschicken: T_{coll}

Lokal einfügen: mischen von Q_0 und neuen Elementen

$\mathcal{O}(\log p)$ mhw.

Aufräumen: Alle $\log p$ Iterationen Q_0 leeren.

$\rightsquigarrow$ Kosten $\mathcal{O}\left(\log p \log \frac{m}{p}\right)$ pro $\log p$ Iterationen

$\rightsquigarrow$ mittlere Kosten $\mathcal{O}\left(\log \frac{m}{p}\right)$

Parallele Realisierung II

deleteMin*

Procedure deleteMin*(Q_0, Q_1, p)

  while $\left|\{e \in \breve{Q}_0 : e < \min \breve{Q}_1\}\right| < p$ **do**

    Q_0 := $Q_0 \cup \{\text{deleteMin}(Q_1)\}$

  M:= select(Q_0, p) // später

  enumerate $M = \{e_1, \ldots, e_p\}$

  assign e_i to PE i // use prefix sums

Analyse

Lokal entfernen: erwartet $\mathcal{O}(1)$ Iterationen $\rightsquigarrow \mathcal{O}\left(T_{\text{coll}} + \log \frac{n}{p}\right)$

Selektion: $\mathcal{O}(T_{\text{coll}})$ mhw todo

M aufzählen: $\mathcal{O}(T_{\text{coll}})$

Ergebnisse ausliefern: $\mathcal{O}(T_{\text{coll}})$ (zufällige Quellen)

Ergebnis

insert*: erwartet $\mathcal{O}\left(T_{\text{coll}} + \log \frac{n}{p}\right)$

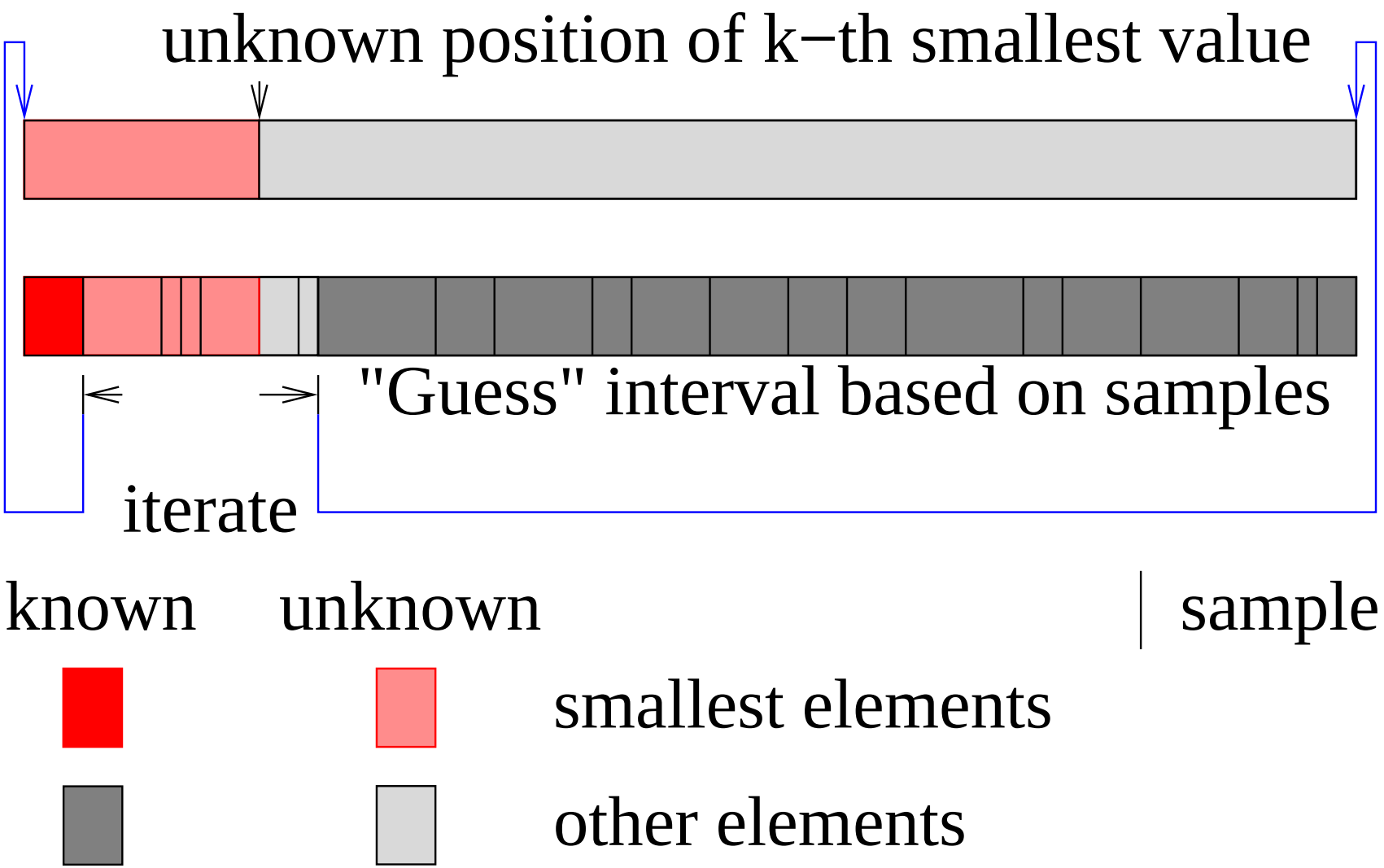
deleteMin*: erwartet $\mathcal{O}\left(T_{\text{coll}} + \log \frac{n}{p}\right)$

Randomisierte Selektion [Blum et al. 1972]

Gegeben n (zufällig allozierte) Elemente Q, finde die k kleinsten.

- wähle ein sample $\mathbf{s}$
- u:= Element mit Rang $\frac{k}{n}|\mathbf{s}| + \Delta$ in $\mathbf{s}$.
  ℓ:= Element mit Rang $\frac{k}{n}|\mathbf{s}| - \Delta$ in $\mathbf{s}$
- Partitioniere Q in
  $Q_< := \{q \in Q : q < \ell\}$,
  $Q_> := \{q \in Q : q > u\}$,
  $Q' := Q \setminus Q_< \setminus Q_>$
- Falls $|Q_<| < k$ und $|Q_<| + |Q'| \geq k$, gib $Q_<$ aus und finde die $k - |Q_<|$ kleinsten Elemente von Q'
- Alle anderen Fälle unwahrscheinlich falls $|\mathbf{s}|$, Δ hinreichend groß.

Randomisierte Selektion [Blum et al. 1972]



Parallele Implementierung

- $|\mathbf{s}| = \sqrt{p} \rightsquigarrow$ Sample kann in Zeit $\mathcal{O}(T_{\text{coll}})$ sortiert werden.
- $\Delta = \Theta\left(p^{1/4+\varepsilon}\right)$ für kleine Konstante ε macht die schwierigen Fälle unwahrscheinlich.
- Keine Elemente werden umverteilt. Zufällige Anfangsverteilung garantiert gute Lastverteilung mhw.
- mhw reichen konstant viele Iterationen bis nur noch $\sqrt{p}$ Elemente übrig $\rightsquigarrow$ direkt sortieren.

Insgesamt erwartete Zeit $\mathcal{O}\left(\frac{n}{p} + T_{\text{coll}}\right)$

Parallele Prioritätslisten, Verfeinerungen

Procedure deleteMin*(Q_0, Q_1, p)
  while $|\{e \in \breve{Q}_0 : e < \min \breve{Q}_1\}| < p$ **do**
    Q_0 := $Q_0 \cup \{\text{deleteMin}(Q_1)\}$ // select immediately
  M:= select(Q_0, p) // später
  enumerate $M = \{e_1, \ldots, e_p\}$
  assign e_i to PE i // use prefix sums

Or just use sufficiently many locally smallest els and check later

Parallel Prioritätslisten, Verfeinerungen

- mergable priority queues?
- bulk delete after flush?
- Größere samples
- größere Batches löschen?
- Nur Teilmenge der PEs spielen PQ-server?

Selection by pruned merging: Eine Reduktion mit Vektorlänge $\mathcal{O}\left(\sqrt{p \log p}\right)$

Asynchrone Variante

Einfügungen akzeptieren aber nicht fertig ausführen.

Batched deleteMin in einen Puffer.

Den mittels asynchroner FIFO zugreifen.

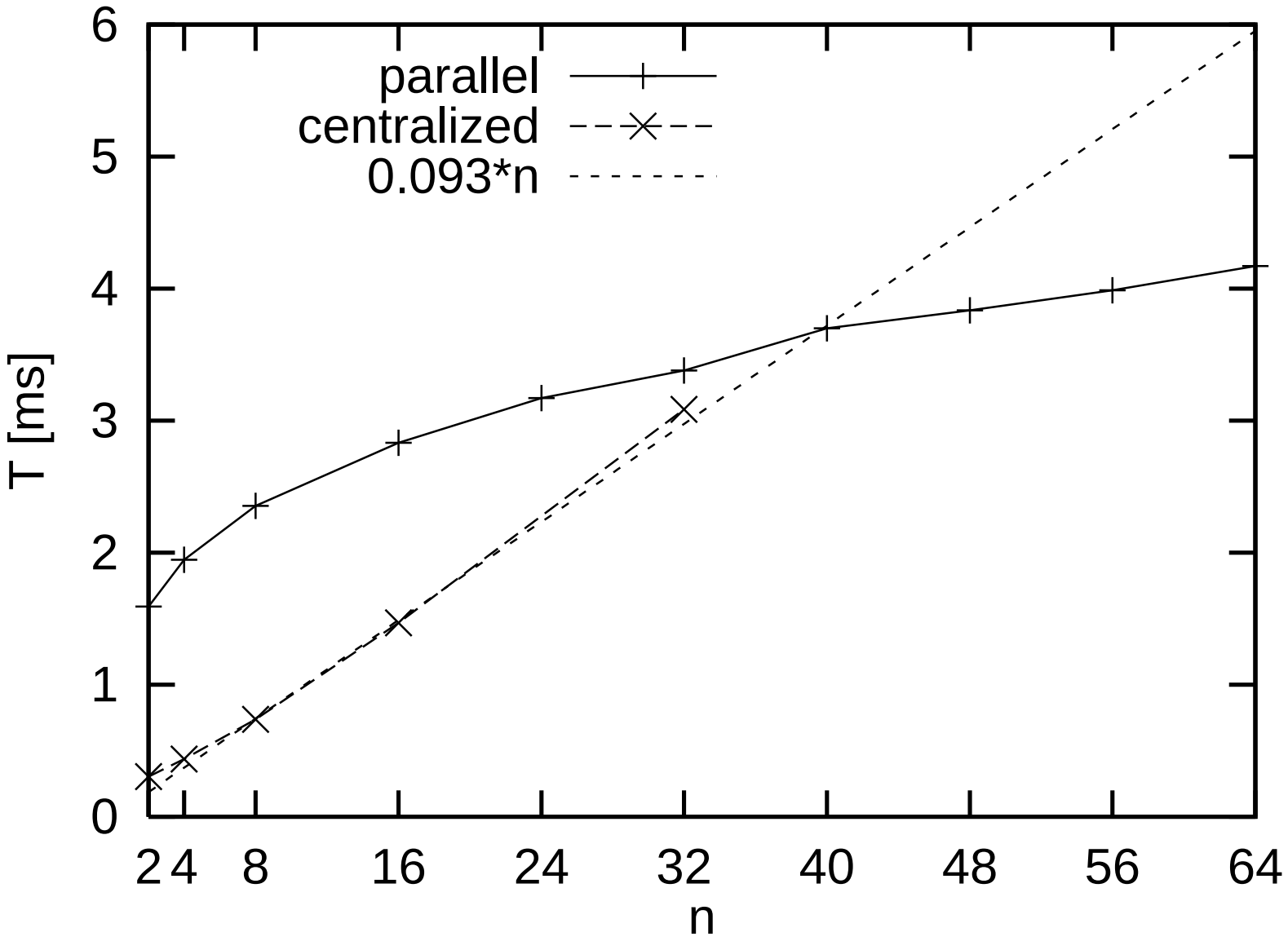
Gelegentlich:

FIFO invalidieren,

commit aller eingefügten Elemente

Puffer neu füllen

Implementierung IBM SP-2, $m = 2^{24}$



Implementierung Cray T3D, $m = 2^{24}$

$p = 256$

256 Els Einfügen plus deleteMin*:

zentralisiert: > 28.16ms

parallel: 3.73ms

break-even bei 34 PEs

Mehr zu parallelen Priority Queues – Geschichte

Anderer Ansatz beginnt mit binary heap:

Knoten mit p sortierten Elementen.

Invariante: Alle Elemente $>$ alle Elemente in Elterknoten

Compare-and-swap $\rightsquigarrow$ merge-and-split

Elegant aber teuer

Parallelisierung des sequentiellen Zugriffs $\rightsquigarrow$ konstante Zeit mit $\log n$ Prozessoren.

Communication Efficient Priority Queues

Each PE stores a search tree augmented with subtree sizes. $\rightsquigarrow$

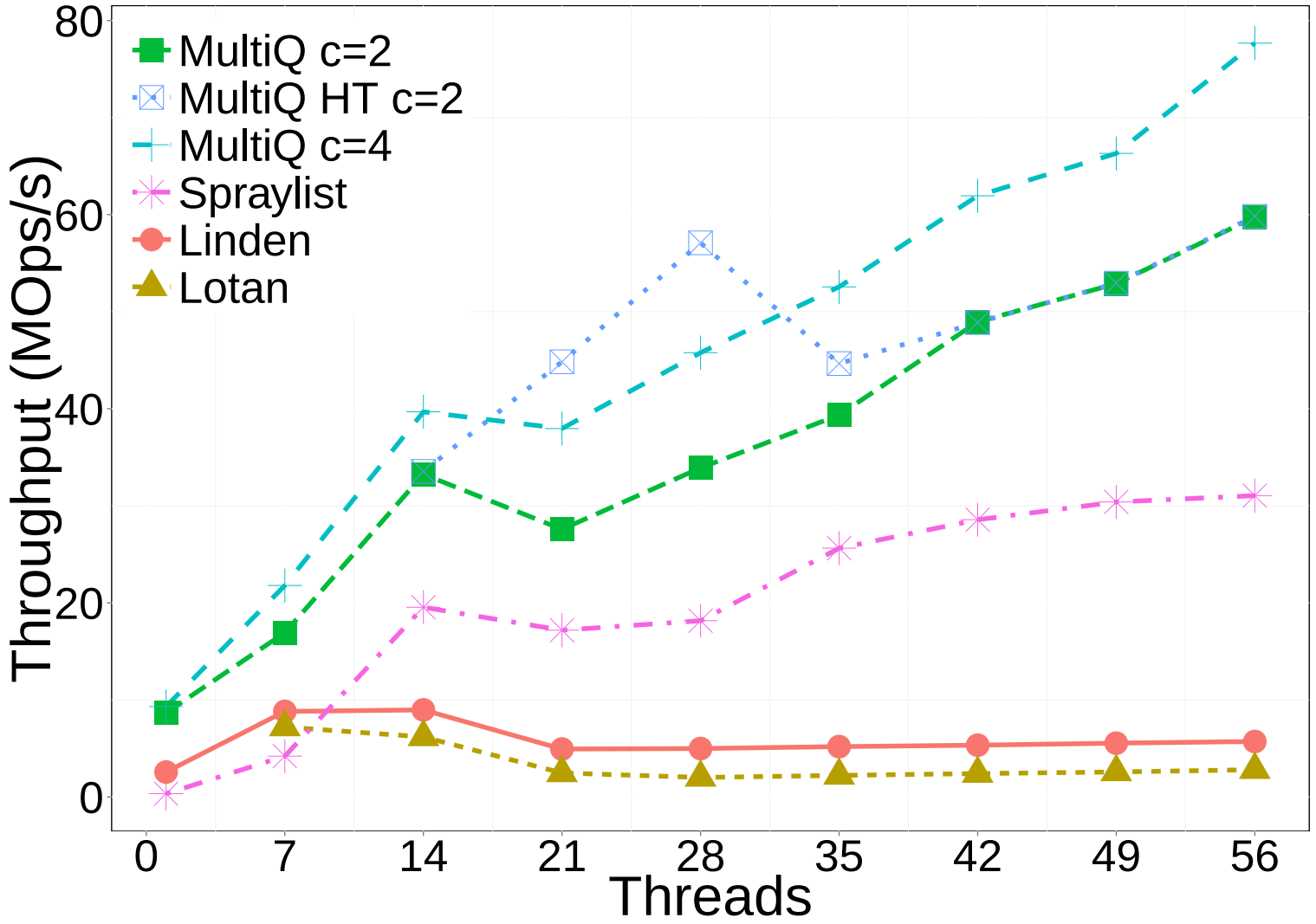
- local insert – $\mathcal{O}(\log n)$ time.
- find k smallest elements in time $\mathcal{O}(\log^2 n)$
  (similar to multi-sequence selection for mergesort)
- find $\Theta(k)$ smallest elements in time $\mathcal{O}(\log n)$

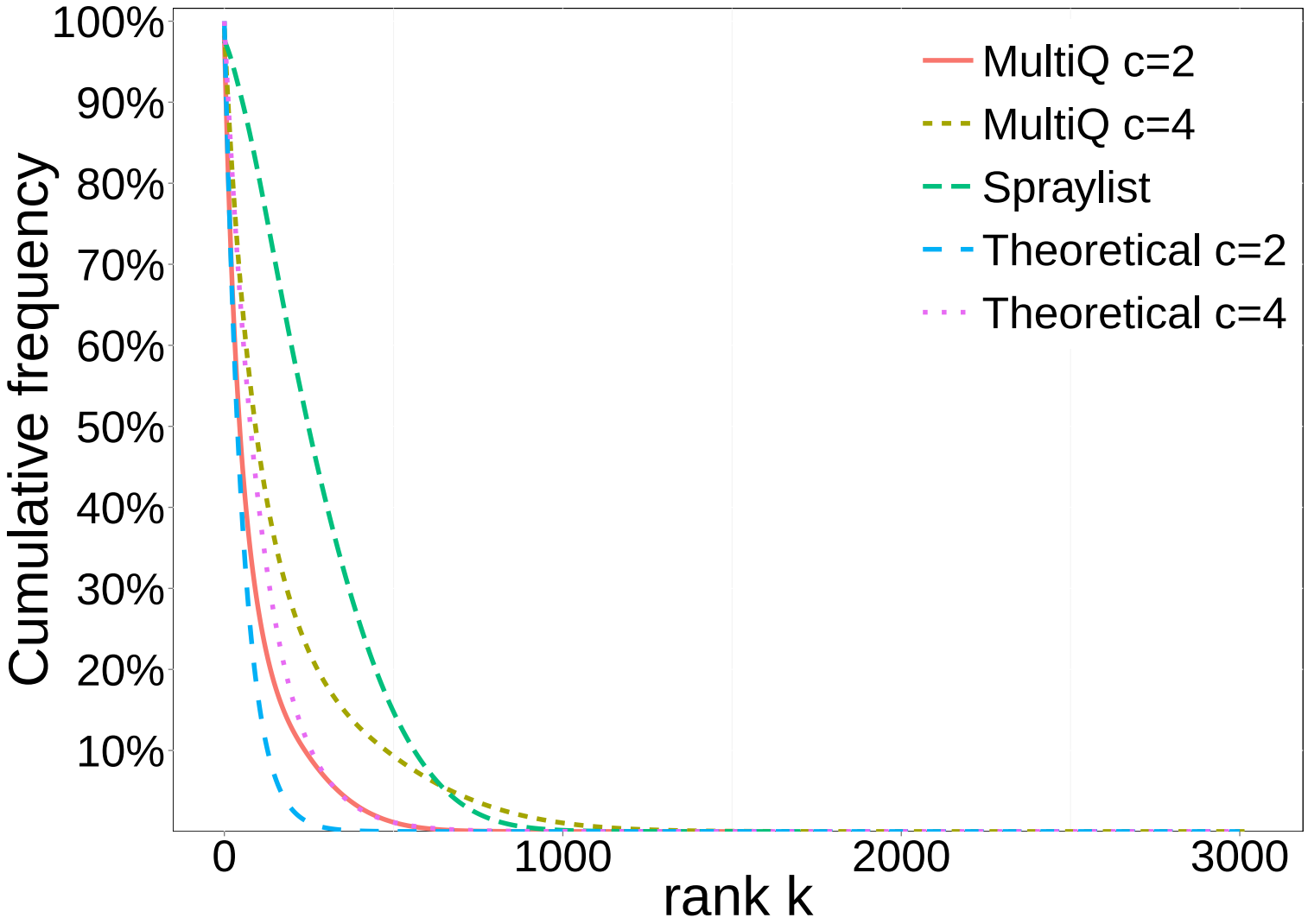
Communication Efficient Algorithms for Top-k Selection Problems, with Lorenz Hübschle-Schneider, IPDPS 2016

MultiQueues: Simple Relaxed Concurrent Priority Queues

with Roman Dementiev and Hamza Rihani, SPAA 2015

- $2p$ local queues $Q[1], \ldots, Q[p]$
- insert into random local queues (“wait-free” locking)
- delete smallest elements from two randomly chosen queues


MultiQ c=2
MultiQ HT c=2
MultiQ c=4
Spraylist
Linden
Lotan
Throughput (MOps/s)
0
20
40
60
80
0
7
14
21
28
35
42
49
56
Threads


100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
Cumulative frequency
0
1000
2000
3000
rank k
MultiQ c=2
MultiQ c=4
Spraylist
Theoretical c=2
Theoretical c=4

List Ranking

Motivation:

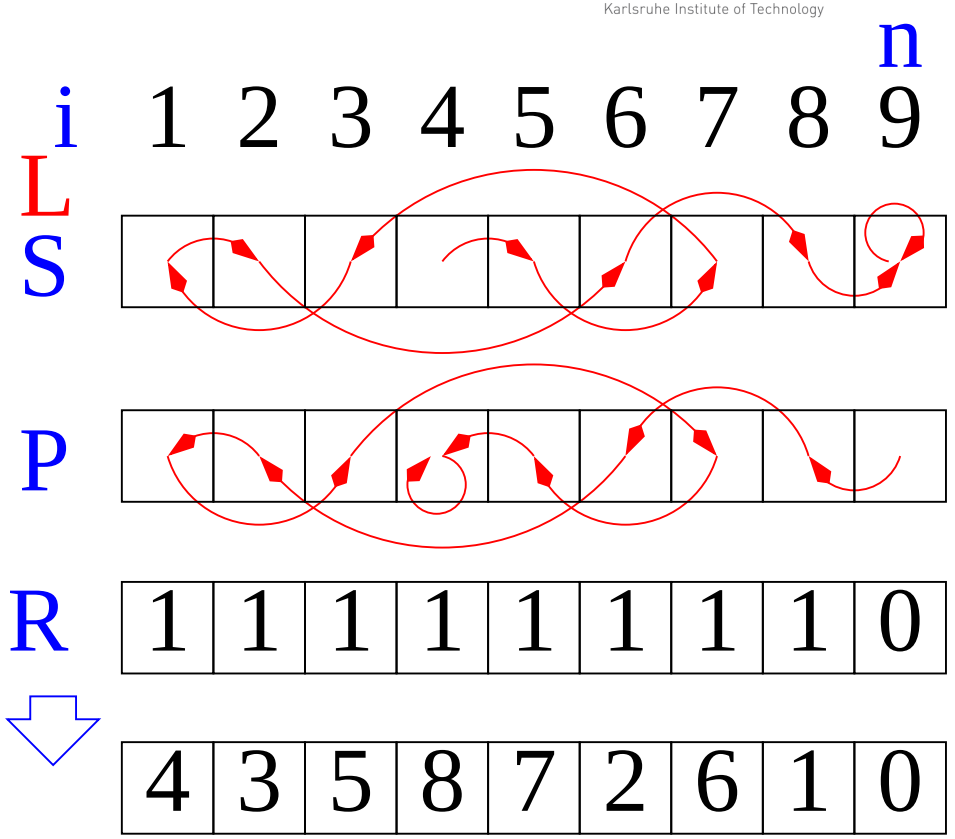
mit Arrays $a[1..n]$ können wir viele Dinge parallel machen

- PE i bearbeitet $a[(i-1)\frac{n}{p}+1..i\frac{n}{p}]$
- Prefixsummen
- ...

Können wir das gleiche mit verketteten Listen?

Ja! in Array konvertieren

List Ranking



L: Liste

n: Elemente

$S(i)$: Nachfolger von Element i

(ungeordnet)

$S(i) = i$: Listenende

$P(i)$: Vorgänger von Element i

Übung: berechne in konstanter Zeit für n PE PRAM

$R(i)$: Anfangs 1, 0 für letztes Element.

Ausgabe:$R(i) =$ Abstand von $S(i)$ vom Ende, rank

Array-Konvertierung: speichere $S(i)$ in $a(n - R(i))$

Motivation II

Listen sind einfache Graphen

$\rightsquigarrow$ warmup für Graphenalgorithmen

$\rightsquigarrow$ lange Pfade sind ein Parallelisierungshindernis

Pointer Chasing

find i such that $S(i) = i$ // parallelizable

for $r := 0$ **to** $n-1$ **do**

$R(i) := r$

$i := P(i)$ // inherently sequential?

- Work $\mathcal{O}(n)$
- Zeit $\Theta(n)$

Doubling using CREW PRAM, $n = p$

$Q(i) := S(i)$ // SPMD. PE index i

invariant $\sum_{j \in Q_i} R(j) =$ rank of item i

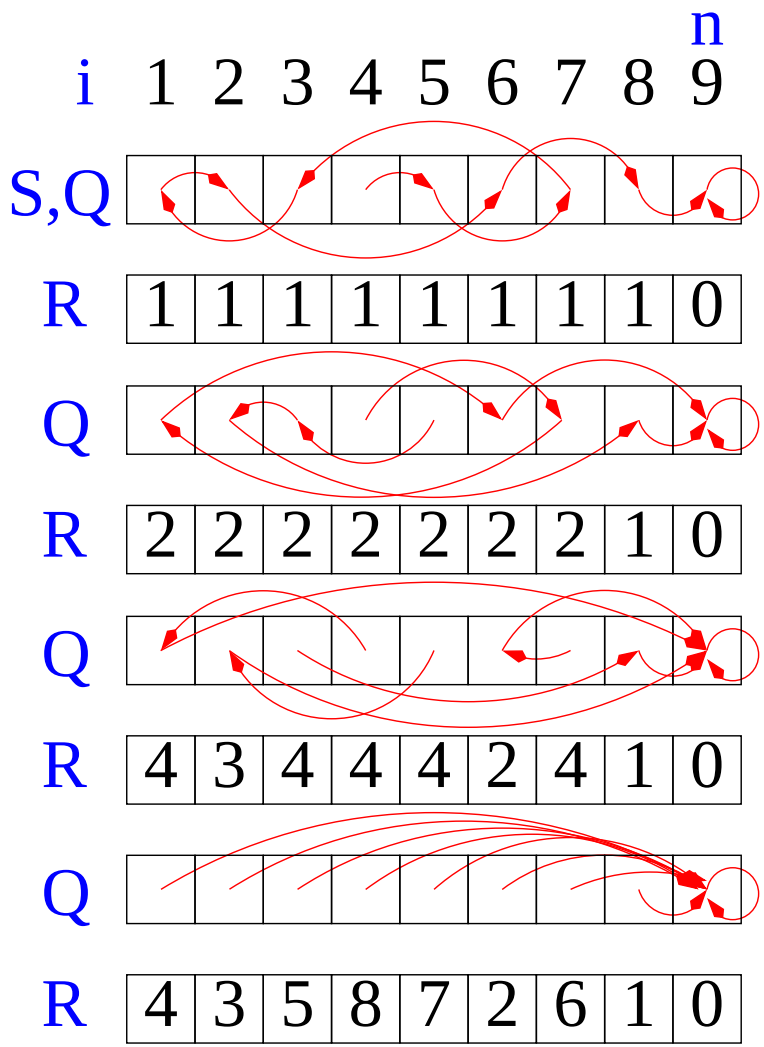
Q_i is the positions given by

chasing Q-pointers from pos i

while $R(Q(i)) \neq 0$ **do**

$R(i) := R(i) + R(Q(i))$

$Q(i) := Q(Q(i))$



Analyse

Induktionsannahme: Nach k Iterationen gilt

- $R(i) = 2^k$ oder
- $R(i) =$ Endergebnis

Beweis: Stimmt für $k = 0$.

$k \rightsquigarrow k+1$:

Fall $R(i) < 2^k$: Bereits Endwert (IV)

Fall $R(i) = 2^k, R(Q(i)) < 2^k$: Nun Endwert (Invariante, IV)

Fall $R(i) = R(Q(i)) = 2^k$: Nun 2^{k+1}

- Work $\Theta(n \log n)$
- Zeit $\Theta(\log n)$

Entfernung unabhängiger Teilmengen

// Compute the sum of the $R(i)$-values when following the $S(i)$ pointers

Procedure independentSetRemovalRank(n, S, P, R)

if $p \geq n$ **then** use doubling; **return**

find $I \subseteq 1..n$ such that $\forall i \in I : S(i) \notin I \wedge P(i) \notin I$

find a bijective mapping $f : \{1..n\} \setminus I \to 1..n - |I|$

foreach $i \notin I$ **dopar** // remove independent set I

$S'(f(i)) := $ **if** $S(i) \in I$ **then** $f(S(S(i)))$ **else** $f(S(i))$

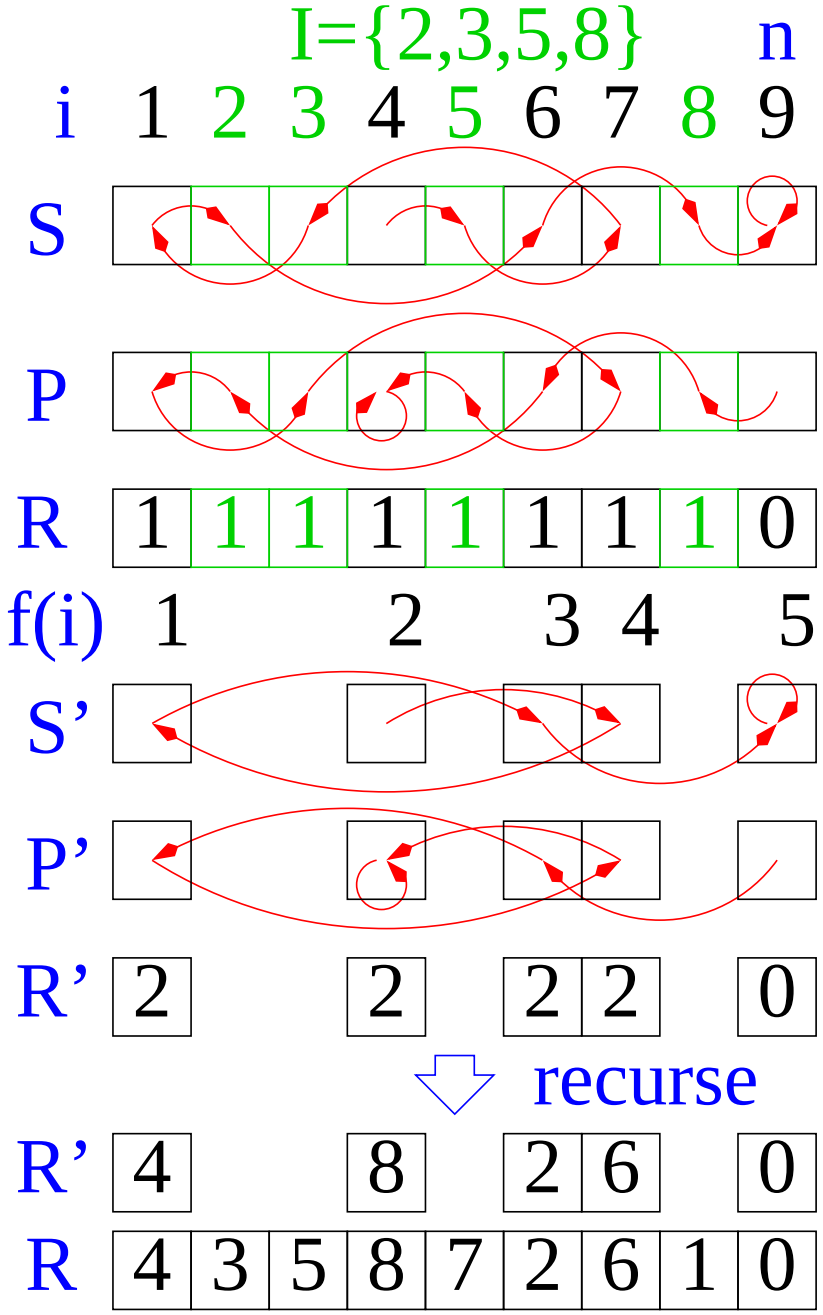
$P'(f(i)) := $ **if** $P(i) \in I$ **then** $f(P(P(i)))$ **else** $f(P(i))$

$R'(f(i)) := $ **if** $S(i) \in I$ **then** $R(i) + R(S(i))$ **else** $R(i)$

independentSetRemovalRank$(n - |I|, S', P', R')$

foreach $i \notin I$ **dopar** $R(i) := R'(f(i))$

foreach $i \in I$ **dopar** $R(i) := R(i) + R'(f(S(i)))$

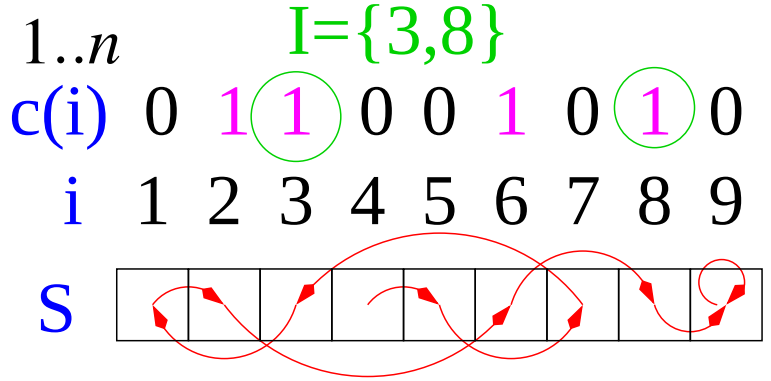

I={2,3,5,8}
n
i 1 2 3 4 5 6 7 8 9
S
P
R 1 1 1 1 1 1 1 1 0
f(i) 1 2 3 4 5
S'
P'
R' 2 2 2 2 0
recurse
R' 4 8 2 6 0
R 4 3 5 8 7 2 6 1 0

Finden unabhängiger Teilmengen

"Werfe Münze" $c(i) \in \{0,1\}$ für jedes $i \in 1..n$

$i \in I$ falls $c(i) = 1 \wedge c(S(i)) = 0$

Erwartete Größe $|I| \approx \frac{n}{4}$



Monte Carlo Algorithmus $\rightsquigarrow$ Las Vegas Algorithmus:

wiederhole so lange bis $|I| > \frac{n}{5}$.

Erwartete Laufzeit: $\mathcal{O}(n/p)$

Weder Anfang noch Ende der Liste sind in I.

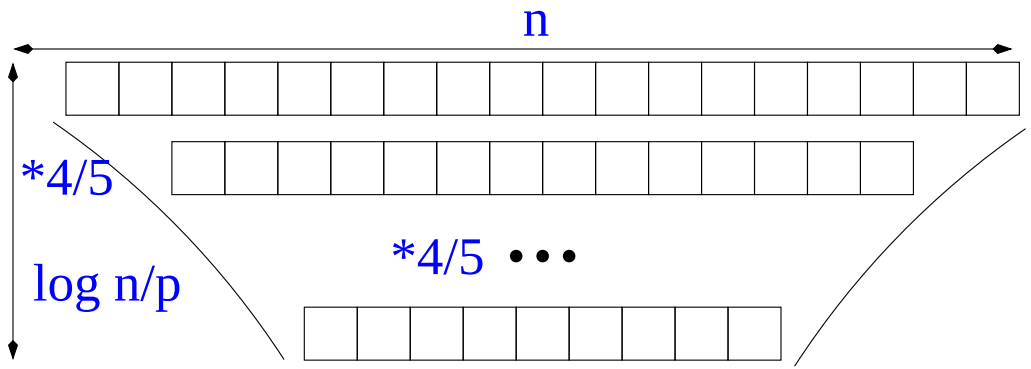
Finden einer bijektiven Abbildung

Prefixsumme über die charakteristische Funktion von $\{1..n\} \setminus I$:

$$f(i) = \sum_{j \leq i} [j \notin I]$$

Analyse

- $T(n) = \mathcal{O}\left(\frac{n}{p} + \log p\right) + T\left(\frac{4}{5}n\right)$ erwartet
- $\mathcal{O}\left(\log \frac{n}{p}\right)$ Rekursionsebenen
- Summe: $\mathcal{O}\left(\frac{n}{p} + \log \frac{n}{p} \log p\right)$ geometrische Summe
- Lineare Arbeit, Zeit $\mathcal{O}(\log n \log\log n)$ mit $\frac{n}{\log n \log\log n}$ PEs



Mehr zu List Ranking

- Einfacher Algorithmus mit erwarteter Zeit $\mathcal{O}(\log n)$
- Komplizierter Algorithmus mit worst case Zeit $\mathcal{O}(\log n)$
- viele “Anwendungen” in PRAM-Algorithmen
- Implementierung auf nachrichtengekoppelten Parallelrechnern
  [Sibeyn 97]: $p = 100$, $n = 10^8$, Speedup 30.
- Verallgemeinerungen für segmentierte Listen, Bäume
- Verallgemeinerungen für allgemeine Graphen:
  kontrahiere Knoten oder Kanten
- Beispiel für Multilevel-Algorithmus

Neuere Implementierungsergbnisse

- Zerschneide Liste an s zufälligen Stellen
- Sequentieller Algorithmus für jede Teilliste
- Rekursive Lösung auf Instanz der Größe s

Speedup ≈ 10 über 8-core CPU (???) [Wei JaJa 2010]

Parallele Graphenalgorithmen

Der „Kanon“ „einfacher“ Graphprobleme:

Hauptinteresse, dünn, polylog. Ausführungszeit, effizient

- DFS
- BFS
- kürzeste Wege
  (nonnegative SSSP $\mathcal{O}(n)$ par. Zeit. interessant für $m = \Omega(np)$)
  (wie ist es mit APSP?)
- topologisches Sortieren
+ Zusammenhangskomponenten (aber nicht starker Zus.)
+ Minimale Spannbäume
+ Graphpartitionierung

Minimum Spanning Trees

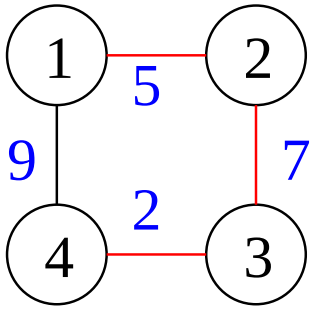
undirected Graph $G = (V, E)$.

nodes V, $n = |V|$, e.g., $V = \{1, \ldots, n\}$

edges $e \in E$, $m = |E|$, two-element subsets of V.

edge weight $c(e)$, $c(e) \in \mathbb{R}_+$ wlog all different.

G is connected, i.e., $\exists$ path between any two nodes.



Find a tree (V, T) with minimum weight $\sum_{e \in T} c(e)$ that connects all nodes.

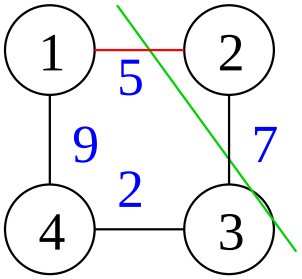
Selecting and Discarding MST Edges

The Cut Property

For any $S \subset V$ consider the cut edges

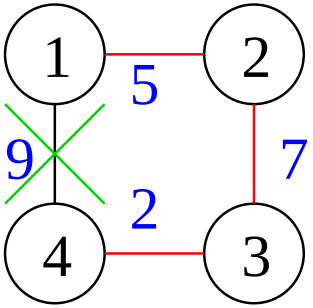
$C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$

The lightest edge in C can be used in an MST.



The Cycle Property

The heaviest edge on a cycle is not needed for an MST



The Jarník-Prim Algorithm

[Jarník 1930, Prim 1957]

Idea: grow a tree

$T := \emptyset$

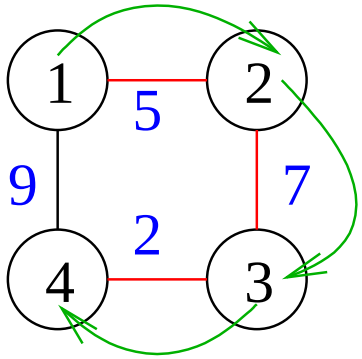
$S := \{s\}$ for arbitrary start node s

repeat $n-1$ times

find (u,v) fulfilling the cut property for S

$S := S \cup \{v\}$

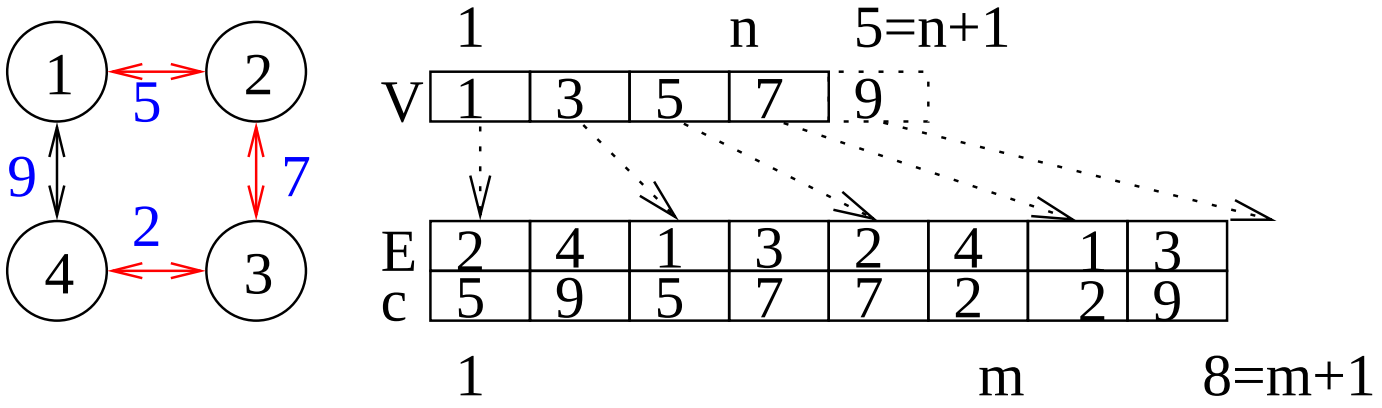
$T := T \cup \{(u,v)\}$



Graph Representation for Jarník-Prim

Adjacency Array

We need node $\rightarrow$ incident edges



Analysis

- $\mathcal{O}(m+n)$ time outside priority queue
- n deleteMin (time $\mathcal{O}(n \log n)$)
- $\mathcal{O}(m)$ decreaseKey (time $\mathcal{O}(1)$ amortized)

$\rightsquigarrow$ $\mathcal{O}(m+n\log n)$ using Fibonacci Heaps

Problem: inherently sequential.

Best bet: use $\log n$ procs to support $\mathcal{O}(1)$ time PQ access.

Kruskal's Algorithm [1956]

$T := \emptyset$ // subforest of the MST

foreach $(u,v) \in E$ in ascending order of weight **do**

  if u and v are in different subtrees of T **then**

    $T := T \cup \{(u,v)\}$ // Join two subtrees

return T

Analysis

$\mathcal{O}(\text{sort}(m) + m\alpha(m,n)) = \mathcal{O}(m \log m)$ where α is the inverse Ackermann function

Problem: still sequential

Best bet: parallelize sorting

Idea: grow tree more aggressively

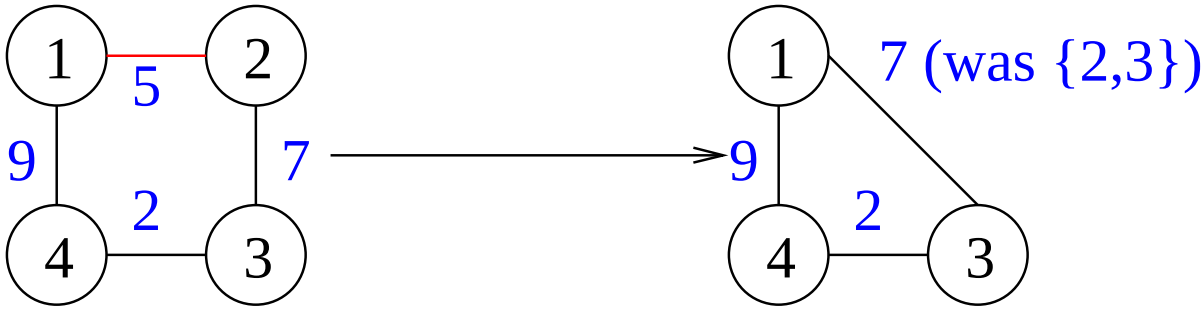
Edge Contraction

Let $\{u, v\}$ denote an MST edge.

Eliminate v:

forall $(w, v) \in E$ **do**

$E := E \setminus (w, v) \cup \{(w, u)\}$ // but remember orignal terminals



Boruvka's Algorithm

[Boruvka 26, Sollin 65]

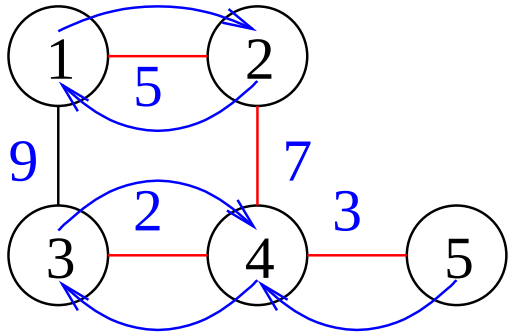
For each node find the lightest incident edge.

Include them into the MST (cut property)

contract these edges,

Time $\mathcal{O}(m)$ per iteration

At least halves the number of remaining nodes



Analysis (Sequential)

$\mathcal{O}(m \log n)$ time

asymptotics is OK for sparse graphs

Goal: $\mathcal{O}(m \log n)$ work $\mathcal{O}(\text{Polylog}(m))$ time parallelization

Finding lightest incident edges

Assume the input is given in adjacency array representation

forall $v \in V$ **dopar**

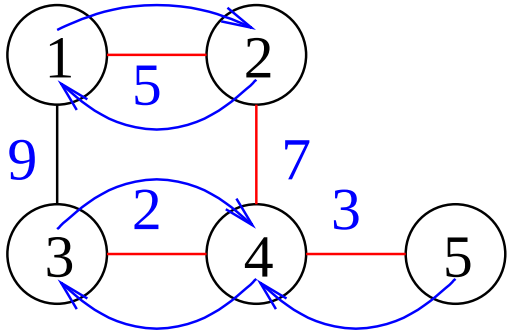
allocate $|\Gamma(v)|\frac{p}{2m}$ processors to node v // prefix sum

find w such that $c(v,w)$ is minimized among $\Gamma(v)$ // reduction

output original edge corresponding to (v,w)

$\text{pred}(v) := w$

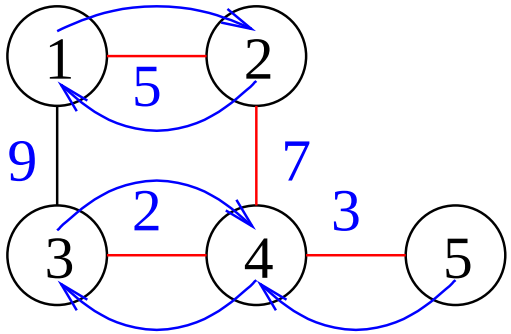
Time $\mathcal{O}\left(\frac{m}{p} + \log p\right)$



Structure of Resulting Components

Consider a component C of the graph $(V, \{(v, \text{pred}(v)) : v \in V\})$

- out-degree 1
- $|C|$ edges
- pseudotree,
  i.e. a tree plus one edge
- one two-cycle at the
  lightest edge (u, w)
- remaining edges lead to u or w



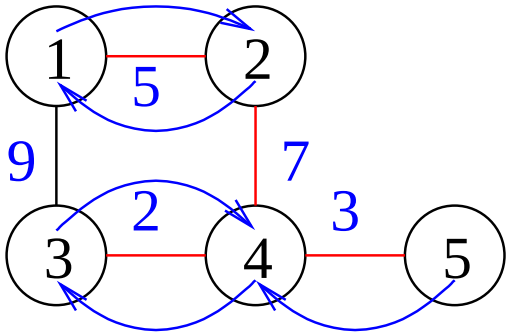
Pseudotrees $\rightarrow$ Rooted Trees

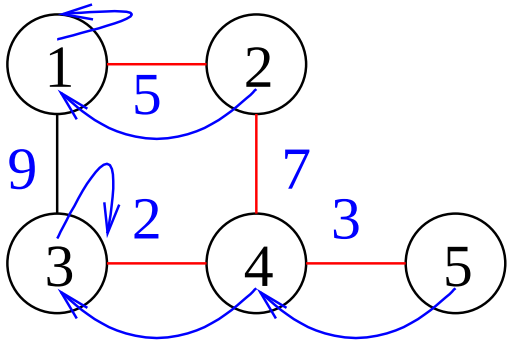
forall $v \in V$ **dopar**

w:= pred(v)

if $v < w \wedge$ pred$(w) = v$ **then** pred(v):= v

Time $\mathcal{O}\left(\frac{n}{p}\right)$



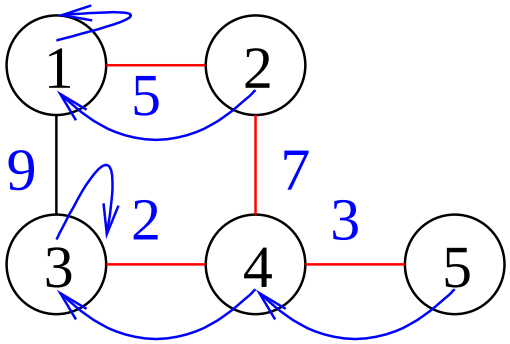


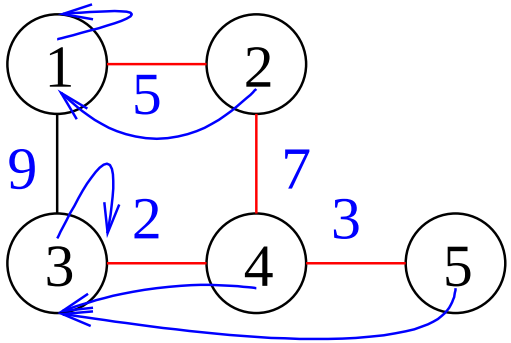
Rooted Trees $\rightarrow$ Rooted Stars by Doubling

while $\exists v \in V : \text{pred}(\text{pred}(v)) \neq \text{pred}(v)$ **do**

forall $v \in V$ **dopar** $\text{pred}(v) := \text{pred}(\text{pred}(v))$

Time $\mathcal{O}\left(\frac{n}{p}\log n\right)$





Contraction

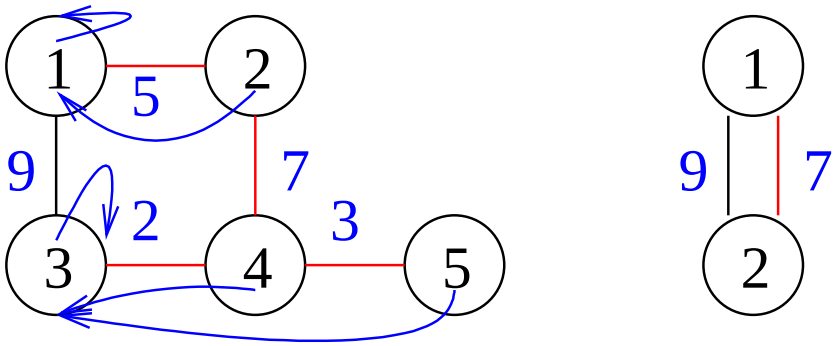
k:= #components

$V' = 1..k$

find a bijective mapping f : star-roots$\rightarrow 1..k$ // prefix sum

E':= $\{(f(\text{pred}(u)), f(\text{pred}(v)), c, e_{\text{old}}) :$
$(u, v, c, e_{\text{old}}) \in E \wedge \text{pred}(u) \neq \text{pred}(v)\}$

Time $\mathcal{O}\left(\frac{m}{p} + \log p\right)$



Recursion

convert $G' = (V', E')$ into adjacency array representation // integer sorting

optional: remove parallel edges // retain lightest one

recurse on G'

Expected sorting time $\mathcal{O}\left(\frac{m}{p} + \log p\right)$ CRCW PRAM

[Rajasekaran and Reif 1989]

practical algorithms for $m \gg p$

Analysis

Satz 5. *On a CRCW-PRAM, parallel Borůvka can be implemented to run in expected time*

$$\mathcal{O}\left(\frac{m}{p}\log n + \log^2 n\right) \quad .$$

- $\leq \log n$ iterations
- sum costs determined above
- for root finding:

$$\sum_i \frac{n}{2^i}\log\frac{n}{2^i} \leq n\log n\sum_i 2^{-i} = \mathcal{O}(n\log n)$$

A Simpler Algorithm (Outline)

Alternate

- Find lightest incident edges of tree roots (grafting)
- One iteration of doubling (pointer jumping)
- Contract leaves

As efficient as with more complicated “starification”

Randomized Linear Time Algorithm

1. Factor 8 node reduction ($3\times$ Boruvka or sweep algorithm)
   $\mathcal{O}(m+n)$.
2. $R \Leftarrow m/2$ random edges. $\mathcal{O}(m+n)$.
3. $F \Leftarrow MST(R)$ [Recursively].
4. Find light edges L (edge reduction). $\mathcal{O}(m+n)$
   $\mathrm{E}[|L|] \leq \frac{mn/8}{m/2} = n/4$.
5. $T \Leftarrow MST(L \cup F)$ [Recursively].

$T(n,m) \leq T(n/8, m/2) + T(n/8, n/4) + c(n+m)$
$T(n,m) \leq 2c(n+m)$ fulfills this recurrence.

Parallel Filter Kruskal

Procedure filterKruskal$(E, T$: Sequence **of** Edge$, P$: UnionFind$)$

if $m \leq$ kruskalThreshold$(n, m, |T|)$ **then**

kruskal(E, T, P) // parallel sort

else

pick a pivot $p \in E$

$E_{\leq} := \langle e \in E : e \leq p \rangle$ // parallel

$E_{>} := \langle e \in E : e > p \rangle$ // partitioning

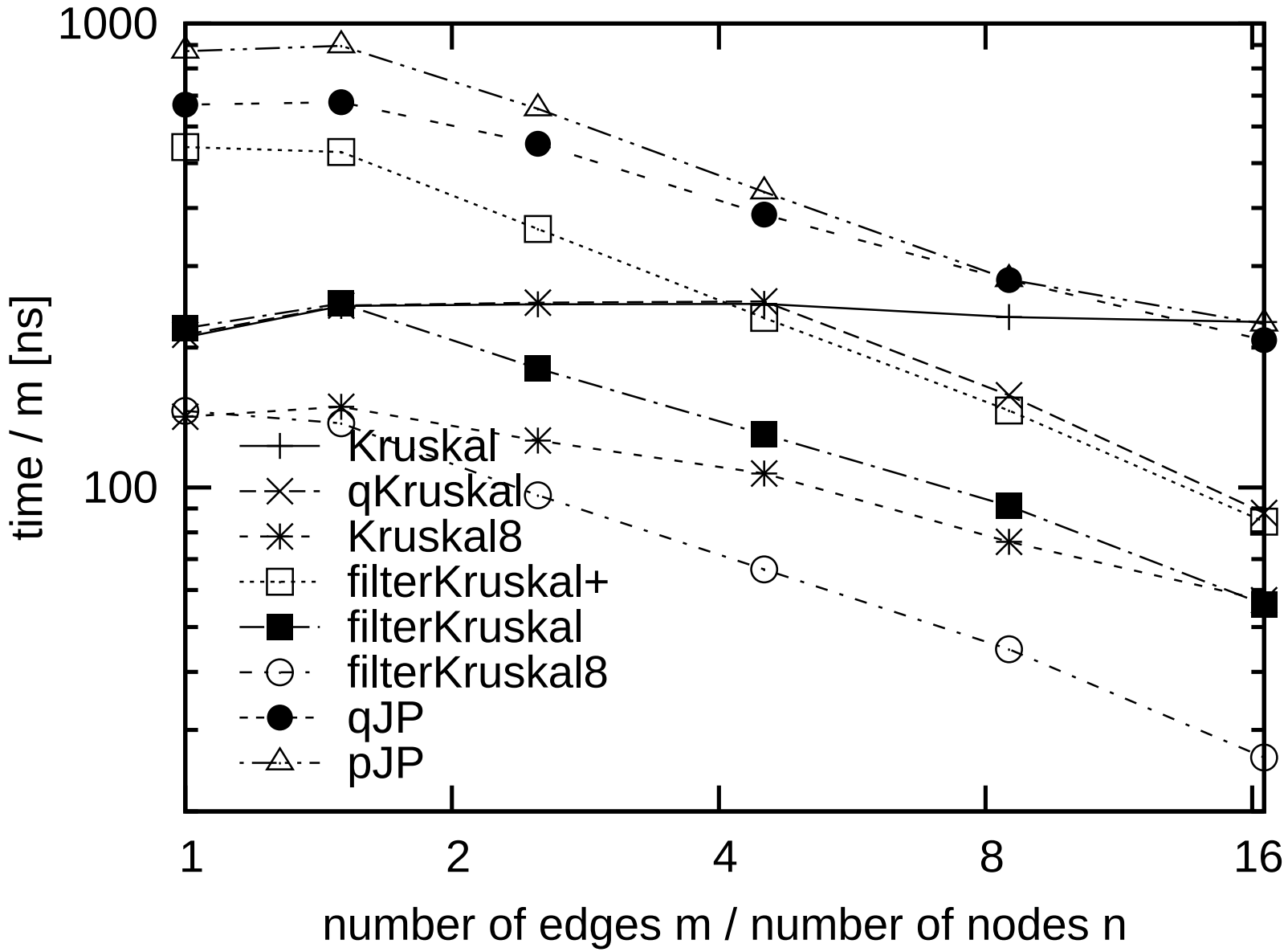
filterKruskal$(E_{\leq}, T, P)$

if $|T| = n - 1$ **then** exit

$E_{>} :=$ filter$(E_{>}, P)$ // parallel removeIf

filterKruskal$(E_{>}, T, P)$

Running Time: Random graph with 2^{16} nodes



More on Parallel MST

[Pettie Ramachandran 02] $\mathcal{O}(m)$ work, $\mathcal{O}(\log n)$ expected time randomized EREW PRAM algorithm.

[Masterarbeit Wei Zhou 17:]Parallel Borůvka $+$ filtering.

Use edge list representation, union-find.

Speedup up to 20 on 72 cores.

Lastverteilung

[Sanders Worsch 97]

Gegeben

- zu verrichtende Arbeit
- PEs

Lastverteilung = Zuordnung Arbeit $\rightarrow$ PEs

Ziel: minimiere parallele Ausführungszeit

Was wir schon gesehen haben

- Lastabschätzung mittels Sampling — sample sort
- Zuteilung ungefähr gleich grosser Stücke — sample sort
- Multisequence selection balanciert multiway merging
- Dynamische Lastbalancierung für quicksort und doall
- Präfixsummen

  quicksort, parPQ, list ranking, MSTs,...
- Parallele Prioritätslisten — branch-and-bound

Kostenmaß

- Maximale Last: $\max_{i=1}^{p} \sum_{j \in \text{jobs @ PE } i} T(j, i, \ldots)$
- Berechnungzeit der Zuteilung
- Durchführung der Zuteilung
- Kosten einer Umverteilung
- Kommunikation zwischen Jobs? (Umfang, Lokalität?)

Was wissen wir über die Jobs?

- genaue Größe
- ungefähre Größe
- (fast) nichts
- weiter aufspaltbar?

dito für Kommunikationskosten

Was wissen wir über die Prozessoren?

- alle gleich?
- unterschiedlich?
- schwankende Fremdlast
- Ausfälle sind zu tolerieren?

dito für Kommunikationsfähigkeiten

In dieser Vorlesung

- Unabhängige Jobs
  - Größen genau bekannt — voll parallele Implementierung
  - Größen nicht oder ungenau bekannt — zufällige Zuordnung, Master Worker Schema, Random Polling

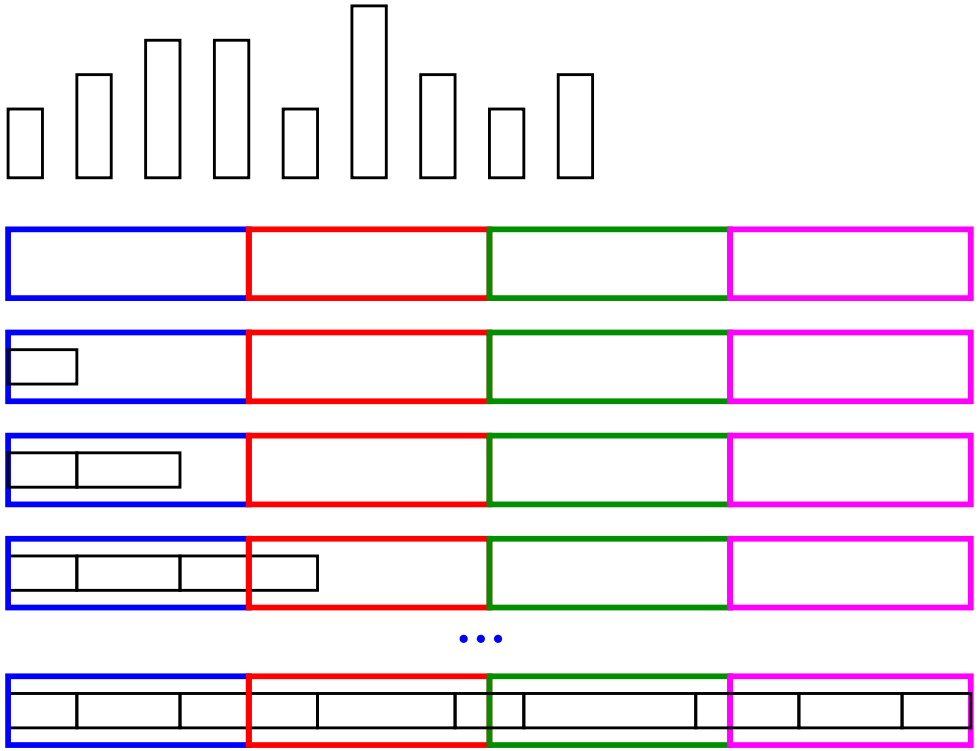
Ein ganz einfaches Modell

- n Jobs, $\mathcal{O}(n/p)$ pro Prozessor, unabhängig, aufspaltbar, Beschreibung mit Platz $\mathcal{O}(1)$
- Größe ℓ_i genau bekannt

Sequentielles Next Fit [McNaughton 59]

$C := \sum_{j \le n} \frac{\ell_i}{p}$ // work per PE
$i := 0$ // current PE
$f := C$ // free room on PE i
$j := 1$ // current Job
$\ell := \ell_1$ // remaining piece of job j
while $j \le n$ **do**
    $c := \min(f, \ell)$ // largest fitting piece
    assign a piece of size c of job j to PE i
    $f := f - c$
    $\ell := \ell - c$
    if $f = 0$ **then** $i{+}{+}$; $f := C$ // next PE
    if $\ell = 0$ **then** $j{+}{+}$; $\ell := \ell_j$ // next job

Sequentielles Next Fit [McNaughton 59]



Parallelisierung von Next Fit (Skizze)

// Assume PE i holds jobs $j_i..j'_i$

$C := \sum_{j \le n} \frac{\ell_i}{p}$

forall $j \le n$ **dopar**

pos:= $\sum_{k<i} \ell_k$ // prefix sums

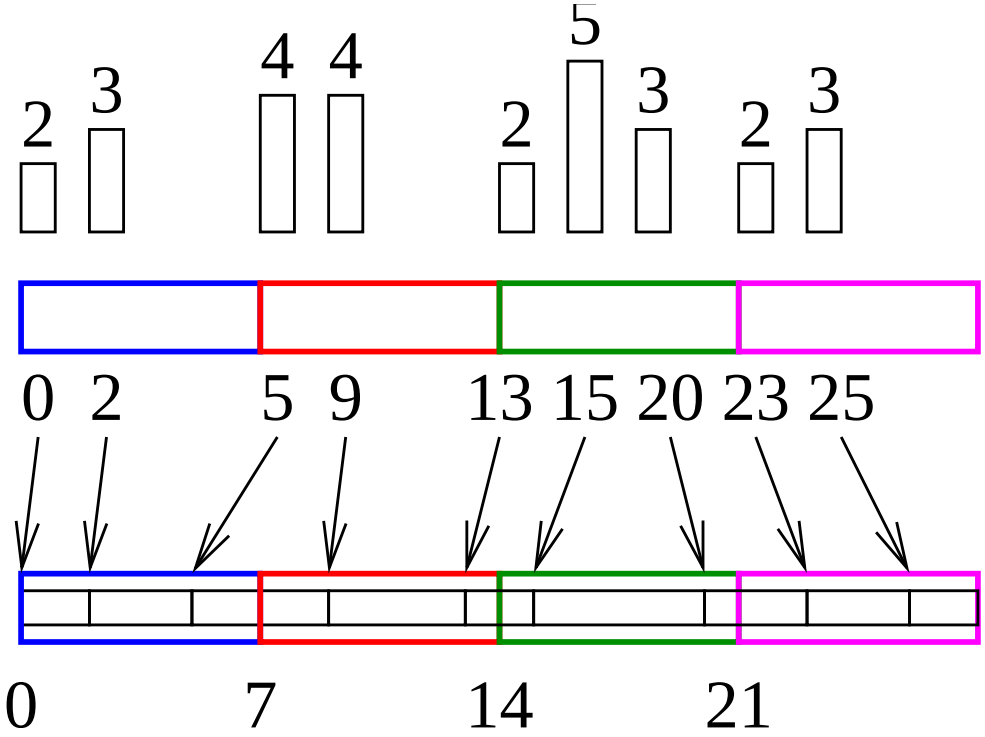
assign job j to PEs $\left\lfloor \frac{\text{pos}}{C} \right\rfloor .. \left\lfloor \frac{\text{pos}+\ell_j}{C} \right\rfloor$ // segmented broadcast

piece size at PE $i = \left\lfloor \frac{\text{pos}}{C} \right\rfloor : (i+1)C - \text{pos}$

piece size at PE $i = \left\lfloor \frac{\text{pos}+\ell_j}{C} \right\rfloor : \text{pos} + \ell_j - iC$

Zeit $C + \mathcal{O}\left(\frac{n}{p} + \log p\right)$ falls Jobs am Anfang zufällig verteilt.

Parallelisierung von Next Fit: Beispiel



Atomare Jobs

assign job j to PE $\left\lfloor \frac{\text{pos}}{C} \right\rfloor$

Maximale Last $\leq C + \max_j \ell_j \leq 2\text{opt}$

Bessere sequentielle Approximation:

Zuteilung nach abnehmender Jobgröße

(shortest queue, first fit, best fit) in Zeit $\mathcal{O}(n \log n)$

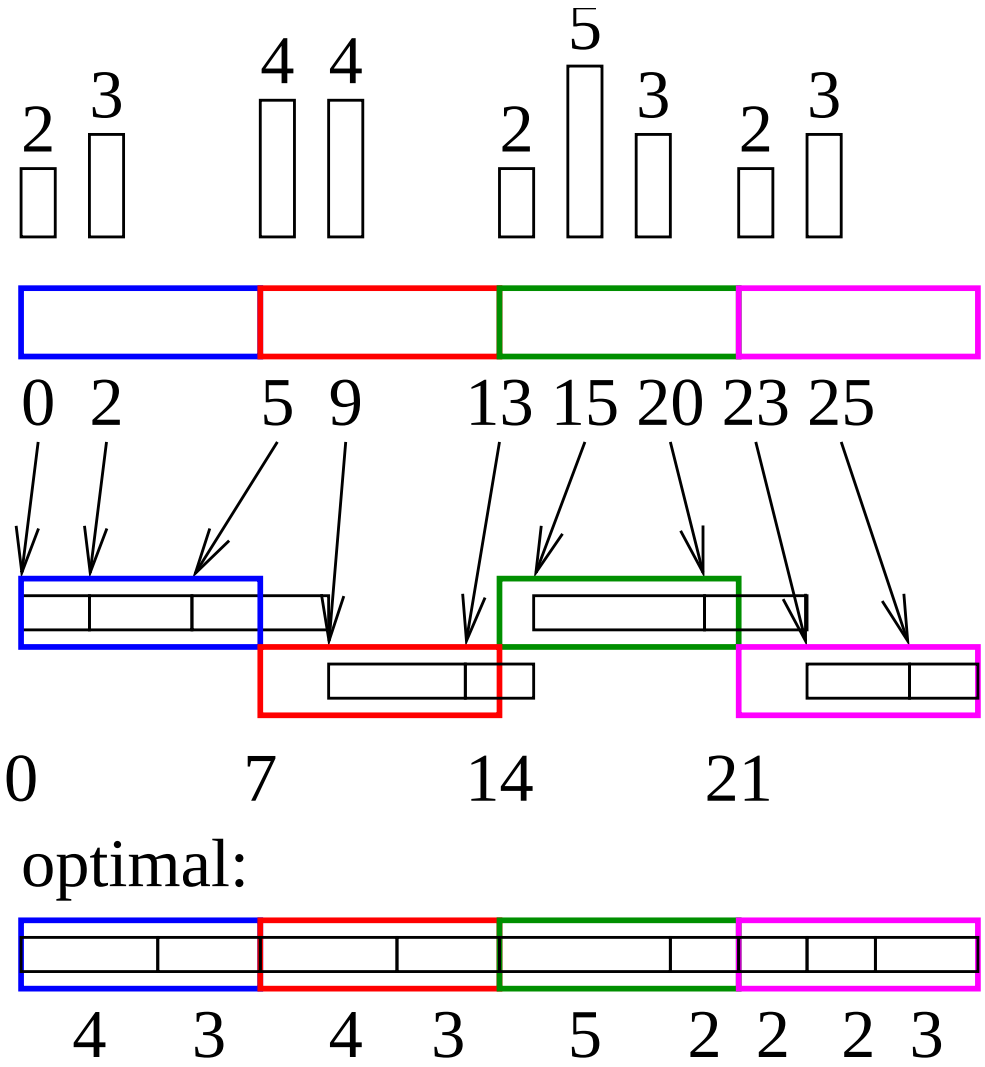
vermutlich nicht parallelisierbar

Parallel

$$\frac{11}{9} \cdot \text{opt}$$

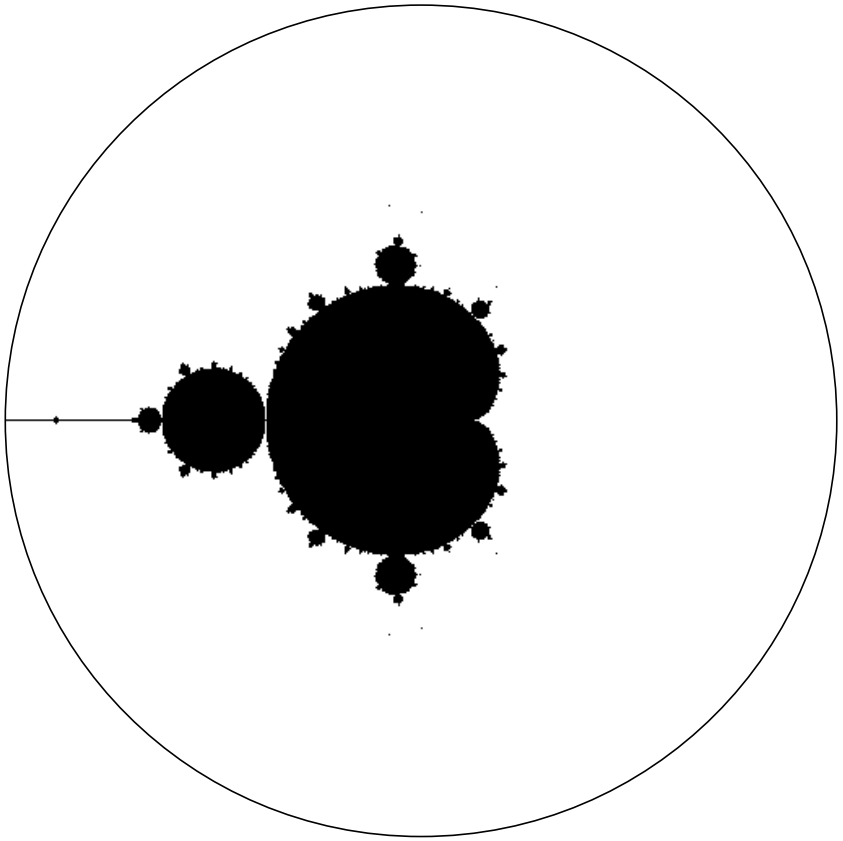
[Anderson, Mayr, Warmuth 89]

Atomare Jobs: Beispiel



Beispiel Mandelbrotmenge

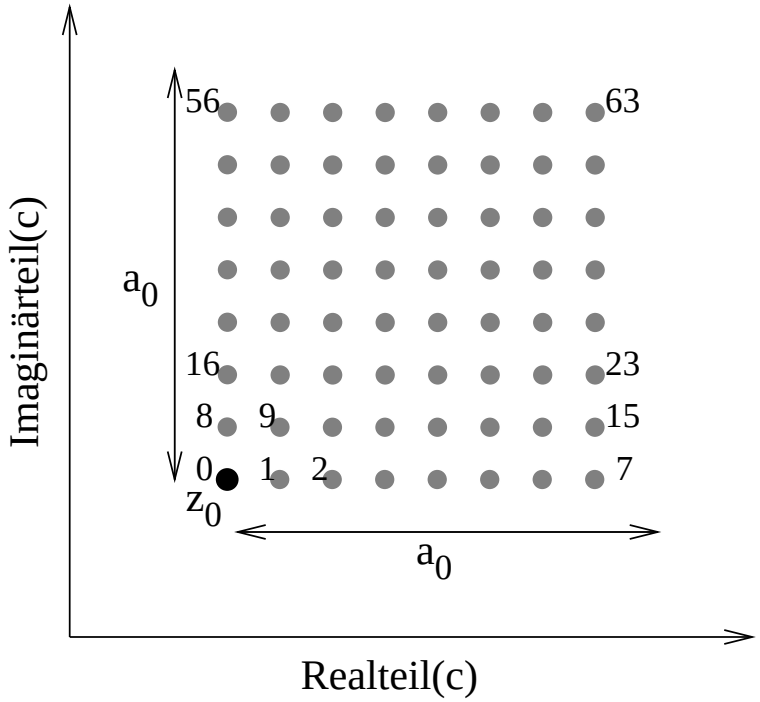
$$z_c(m) : \mathbb{N} \to \mathbb{C}$$
$$z_c(0) := 0, \quad z_c(m+1) := z_c(m)^2 + c$$
$$M := \{c \in \mathbb{C} : z_c(m) \text{ ist beschränkt}\} \ .$$



Angenäherte Berechnung

- Berechung nur für quadratischen Ausschnitt der komplexen Zahlenebene
- Berechnung nur für diskretes Gitter von Punkten
- z_c unbeschränkt falls $|z_c(k)| \geq 2$
- Abbruch nach $m_{\max}$ Iterationen

Wo liegt das Lastverteilungsproblem?



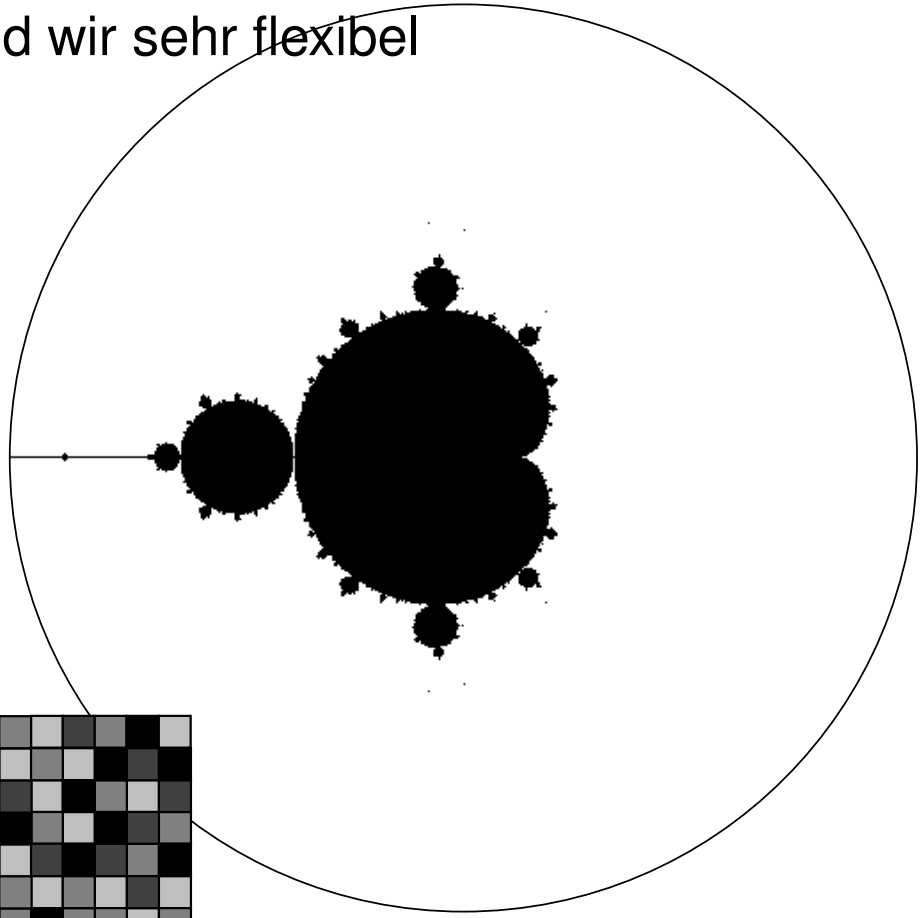
Code

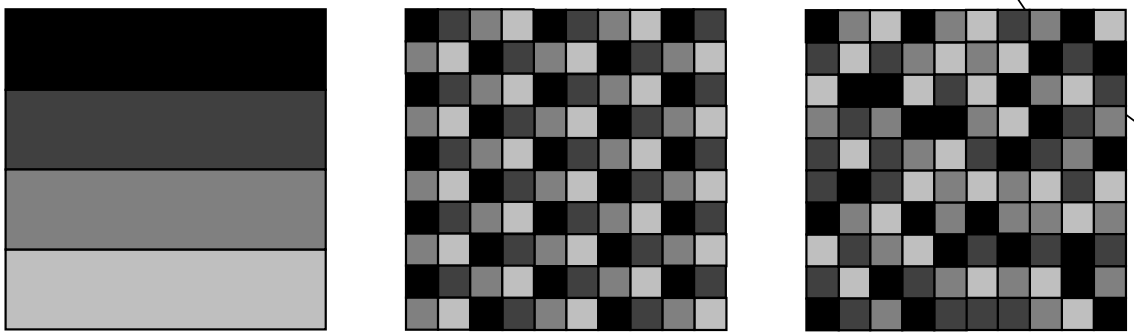
```
int iterate(int pos, int resolution, double step)
{ int iter;
  complex c =
    z0+complex((double)(pos % resolution) * step,
               (double)(pos / resolution) * step);
  complex z = c;
  for (iter = 1;
       iter < maxiter && abs(z) <= LARGE;
       iter++) {
    z = z*z + c;
  }
  return iter; }
```

Statische Äpfelverteilung

Da kaum Kommunikation stattfindet sind wir sehr flexibel

- Streifenweise Zerlegung
  - Warum attraktiv?
  - Warum besser nicht?
- zyklisch. Gut. Aber beweisbar?
- Zufällig





Bearbeitet von: =PE 0 =PE 1 =PE 2 =PE 3

Parallelisierung der Zuordnungsphase

- Wenn die Teilprobleme irgendwie auf die PEs verteilt sind: Zufallspermutation via all-to-all. (Siehe auch sample sort)
- Implizites Erzeugen der Einzelteile
  - Teilproblem läßt sich allein aus seiner Nummer $1 \ldots n$ erzeugen.
  - Problem: Parallele Berechnung einer (Pseudo)Zufallspermutation

Pseudorandom Permutations $\pi : 0..n-1 \to 0..n-1$

Wlog (?) let n be a square.

- Interpret numbers from $0..n-1$ as pairs from $\{0..\sqrt{n}-1\}^2$.
- $f : 0..\sqrt{n}-1 \to 0..\sqrt{n}-1$ (pseudo)random function
- Feistel permutation: $\pi_f((a,b)) = (b, a + f(b) \bmod \sqrt{n})$
  $(\pi_f^{-1}(b,x) = (x - f(b) \bmod \sqrt{n}, b))$
- Chain several Feistel permutations
- $\pi(x) = \pi_f(\pi_g(\pi_h(\pi_l(x))))$ is even save in some cryptographical sense

Zufälliges Zuordnen

- Gegeben: n Teilprobleme der Größe $\ell_1, \ldots, \ell_n$
- Sei $L := \sum_{i \leq n} \ell_i$
- Sei $l_{\max} := \max_{i \leq n} \ell_i$
- Ordne die Teilprobleme zufälligen PEs zu

Satz: Falls $L \geq 2(\beta+1)pl_{\max}\frac{\ln p}{\varepsilon^2} + O(1/\varepsilon^3))$

dann ist die maximale Last höchstens $(1+\varepsilon)\frac{L}{p}$

mit Wahrscheinlichkeit mindestens $1 - p^{-\beta}$. Beweis:

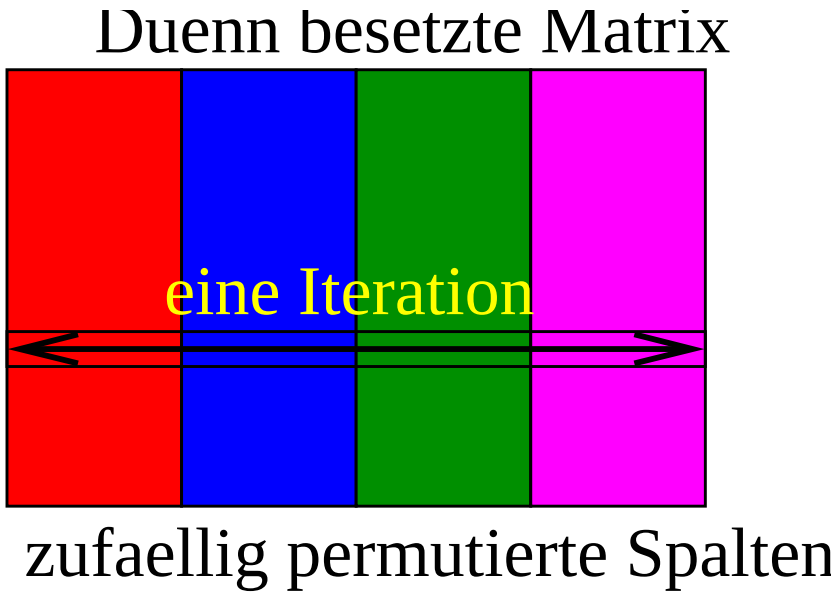
. . . Chernoff-Schranken. . .

Diskussion

+ Teilproblemgrößen müssen überhaupt nicht bekannt sein
+ Es ist unerheblich wo die Teilprobleme herkommen
  (verteilte Erzeugung möglich)
− inakzeptabel bei großem $l_{\max}$
− Sehr gute Lastverteilung nur bei sehr großem $L/l_{\max}$
  (quadratisch in $1/\varepsilon$,logarithmisch in p).

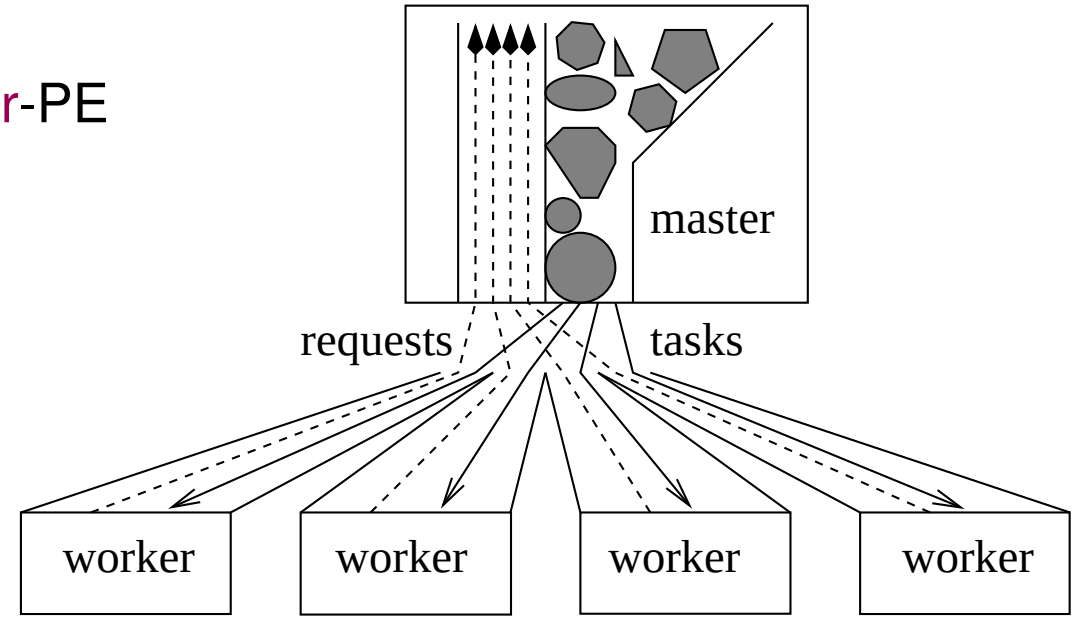
Anwendungsbeispiel: Airline Crew Scheduling

Eine einzige zufällige Verteilung löst k simultane Lastverteilungsprobleme. (Deterministisch vermutlich ein schwieriges Problem.)



Das Master-Worker-Schema

- Anfangs alle Jobs auf Master-PE
- Jobgrößen sind abschätzbar aber nicht genau bekannt
- Einmal abgegebene Jobs können nicht weiter unterteilt werden (nichtpreemptiv)

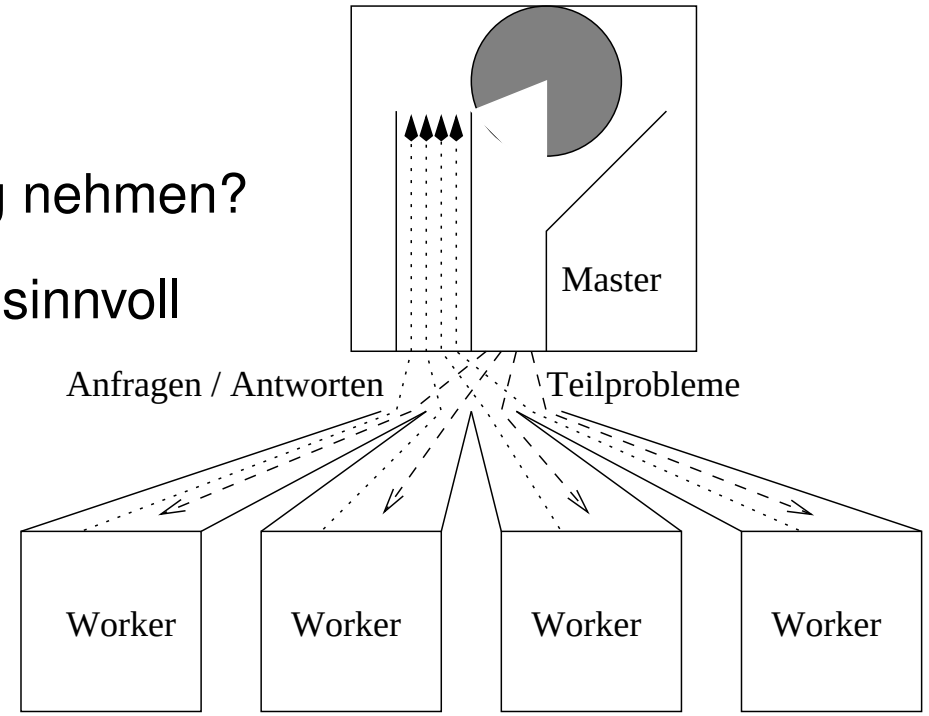


Diskussion

+ Einfach
+ Natürliches Ein- Ausgabeschema (aber u.U. gesonderter Plattensklave)
+ Naheliegend wenn Jobgenerator nicht parallelisiert
+ Leicht zu debuggen
− Kommunikationsengpaß $\Rightarrow$ Tradeoff Kommunikationsaufwand versus Imbalance
− Wie soll aufgespalten werden?
− Multilevelschemata sind kompliziert und nur begrenzt hilfreich

Größe der Teilprobleme

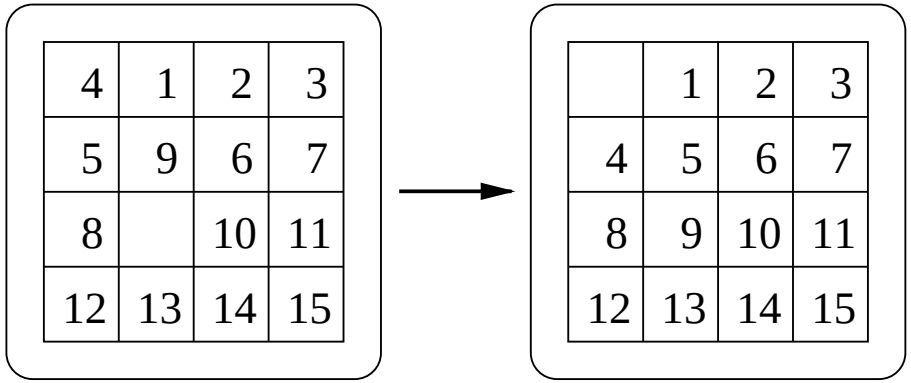
- Möglichst grosse Probleme abgeben solange Lastverteilung nicht gefährdet. Warum?
- Konservatives Kriterium: obere Schranke für die Größe des abgegebenen Teilproblems $\leq$ $1/P$-tel untere Schranke für Systemlast.
  - Woher Grössenabschätzung nehmen?
- Aggressivere Verfahren ggf. sinnvoll



Work Stealing

- (Fast) Beliebig unterteilbare Last
- Anfangs alles auf PE 0
- Fast nichts bekannt über Teilproblemgrößen
- Preemption erlaubt. (Sukzessives aufspalten)

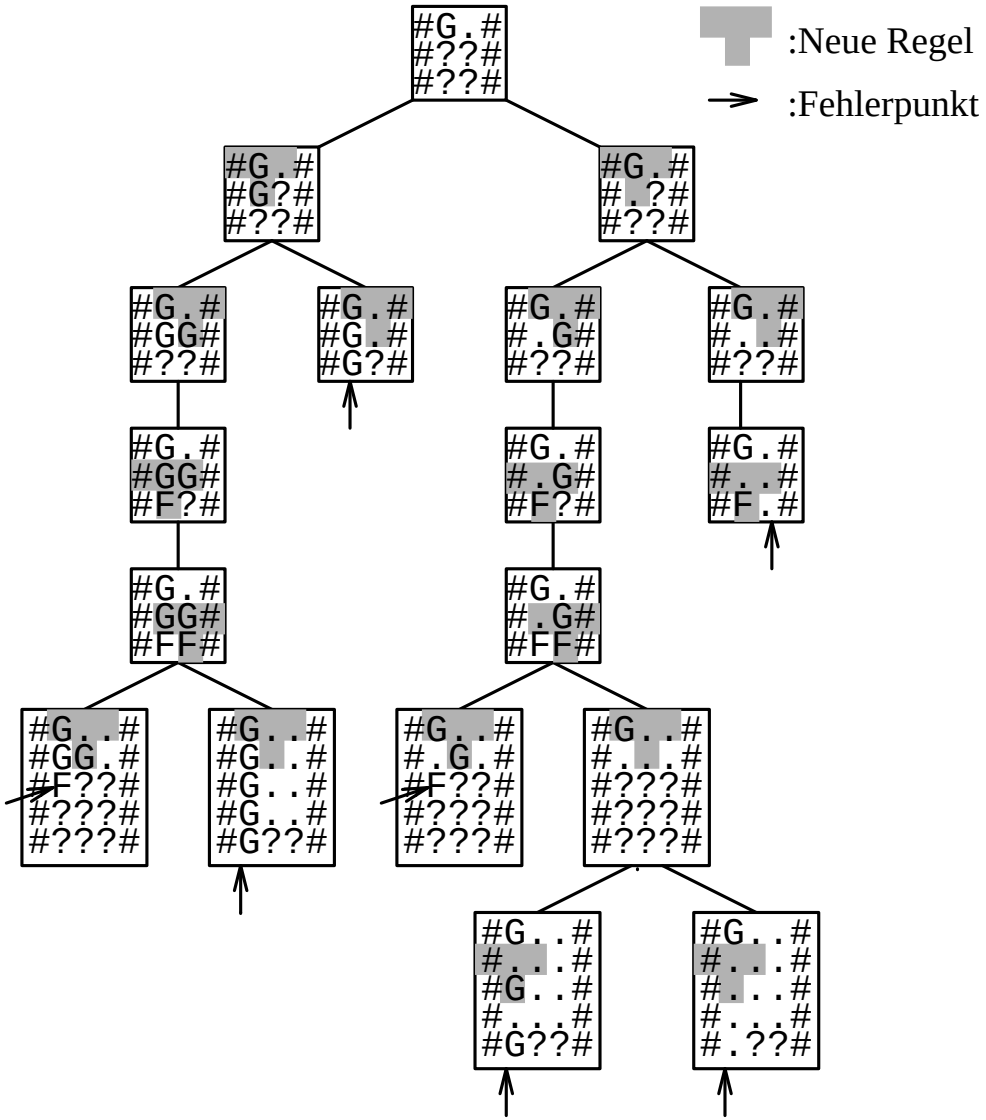
Example: The 15-Puzzle



Korf 85: Iterative deepening depth first search with $\approx 10^9$ tree nodes.

Example: Firing Squad Synchronization Problem

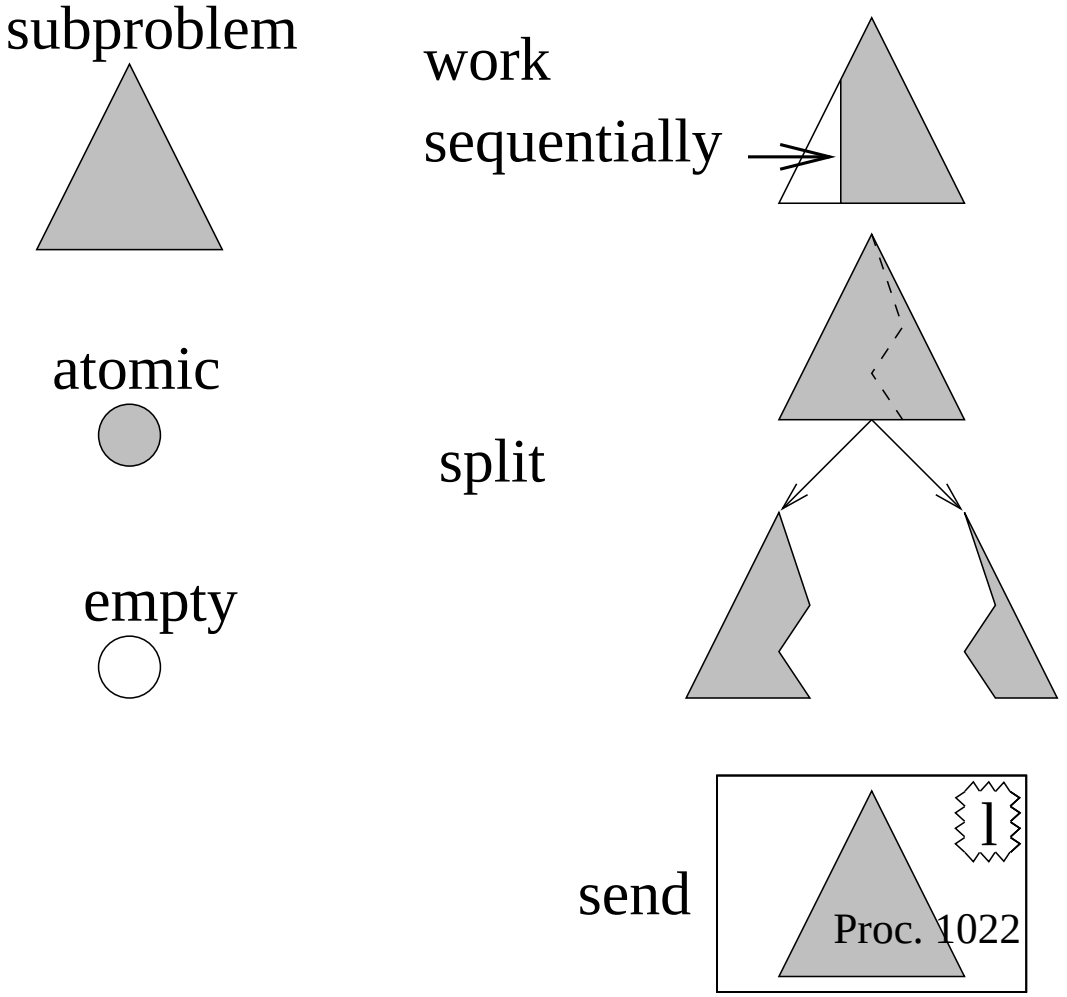
Backtracking over Transition Functions



Goal for the analysis

$$T_{\text{par}} \leq (1+\varepsilon)\frac{T_{\text{seq}}}{p} + \text{lower order terms}$$

An Abstract Model:
Tree Shaped Computations



Tree Shaped Computations: Parameters

T_{atomic}: max. time for finishing up an atomic subproblem

T_{split}: max. time needed for splitting

h: max. generation $\text{gen}(P)$ of a nonatomic subproblem P

ℓ: max size of a subproblem description

p: no. of processors

T_{rout}: time needed for communicating a subproblem $(\alpha + \ell\beta)$

T_{coll}: time for a reduction

Relation to Depth First Search

let stack consist of root node only

while stack is not empty **do**

    remove a node N from the stack

    if N is a leaf **then**

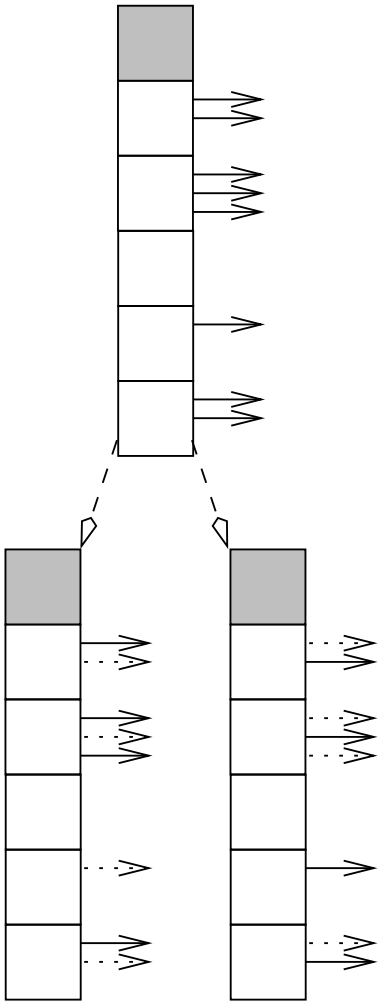
        evaluate leaf N

    else

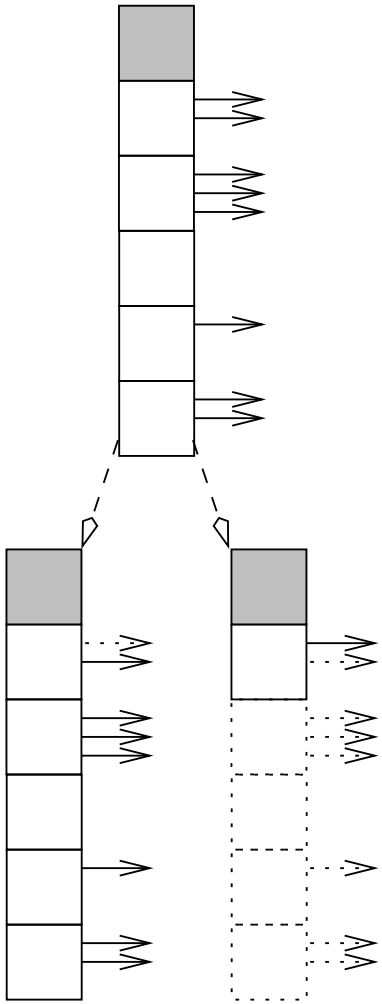
        put successors of N on the stack

    fi

Splitting Stacks



a)



b)

Other Problem Categories

- Loop Scheduling
- Higher Dimensional Interval Subdivision
- Particle Physics Simulation
- Generalization: Multithreaded computations. $h \rightsquigarrow T_\infty$

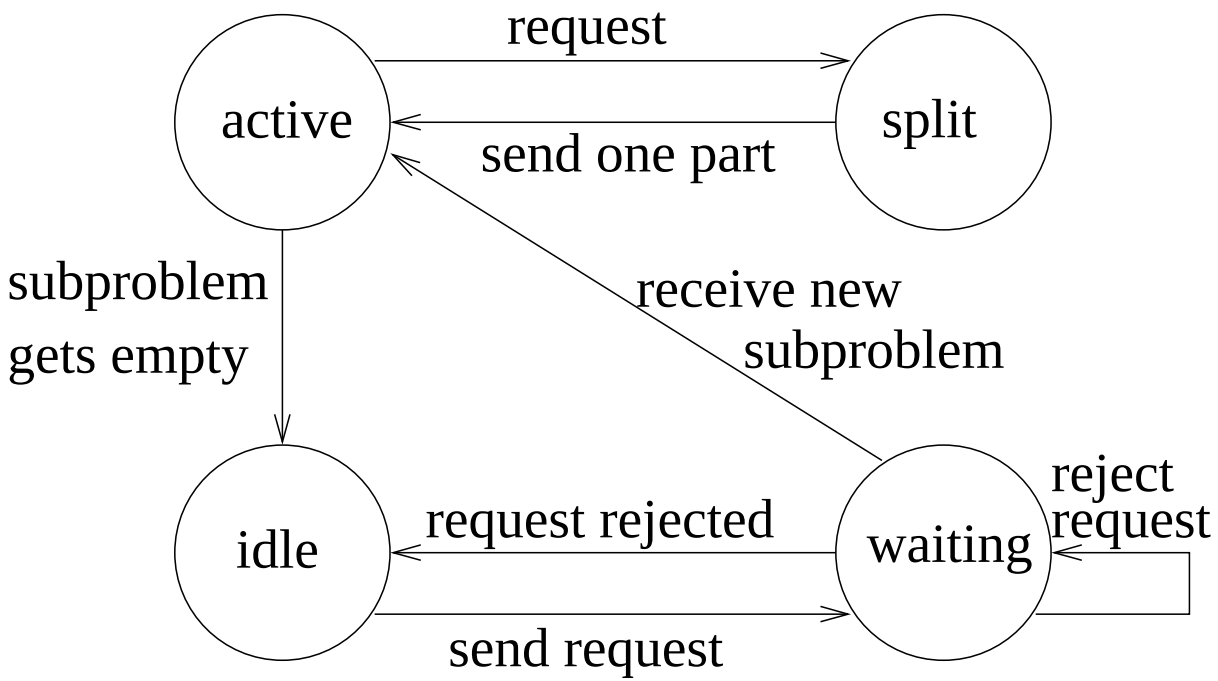
An Application List

- Discrete Mathematics (Toys?):
  - Golomb Rulers
  - Cellular Automata, Trellis Automata
  - 15-Puzzle, n-Queens, Pentominoes ...
- NP-complete Problems (nondeterminism $\rightsquigarrow$ branching)
  - 0/1 Knapsack Problem (fast!)
  - Quadratic Assignment Problem
  - SAT
- Functional, Logical Programming Languages
- Constraint Satisfaction, Planning, ...
- Numerical: Adaptive Integration, Nonlinear Optimization by Interval Arithmetics, Eigenvalues of Tridiagonal Matrices

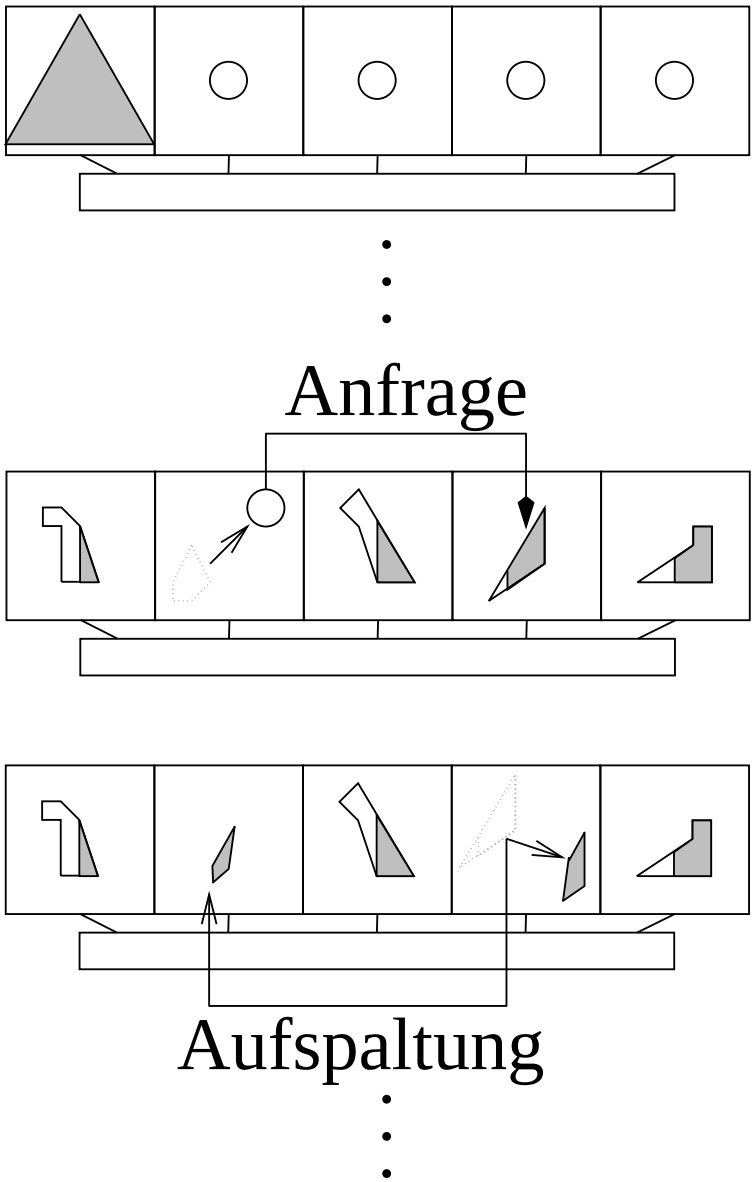
Limits of the Model

- Quicksort and similar divide-and-conquer algorithms (shared memory OK $\rightsquigarrow$ Cilk, MCSTL, Intel TBB, OpenMP 3.0?)
- Finding the first Solution (often OK)
- Branch-and-bound
  - Verifying bounds OK
  - Depth-first often OK
- Subtree dependent pruning
  - FSSP OK
  - Game tree search tough (load balancing OK)

Receiver Initiated Load Balancing



Random Polling



$\tilde{\mathrm{O}}(\cdot)$ Calculus

$X \in \tilde{\mathrm{O}}(f(n))$ – iff $\forall \beta > 0$:

$$\exists c > 0, n_0 > 0 : \forall n \geq n_0 : \mathbb{P}[X > cf(n)] \leq n^{-\beta}$$

Advantage: simple rules for sum and maximum.

Termination Detection

not here

Synchronous Random Polling

P, P' : Subproblem

P := **if** $i_{\text{PE}} = 0$ **then** P_{root} **else** $P_\emptyset$

loop $P := \text{work}(P, \Delta t)$

$m' := |\{i : T(P@i) = 0\}|$

if $m' = p$ **then** exit loop

else if $m' \geq m$ **then**

if $T(P) = 0$ **then** send a request to a random PE

if there is an incoming request **then**

$(P, P') := \text{split}(P)$

send P' to one of the requestors

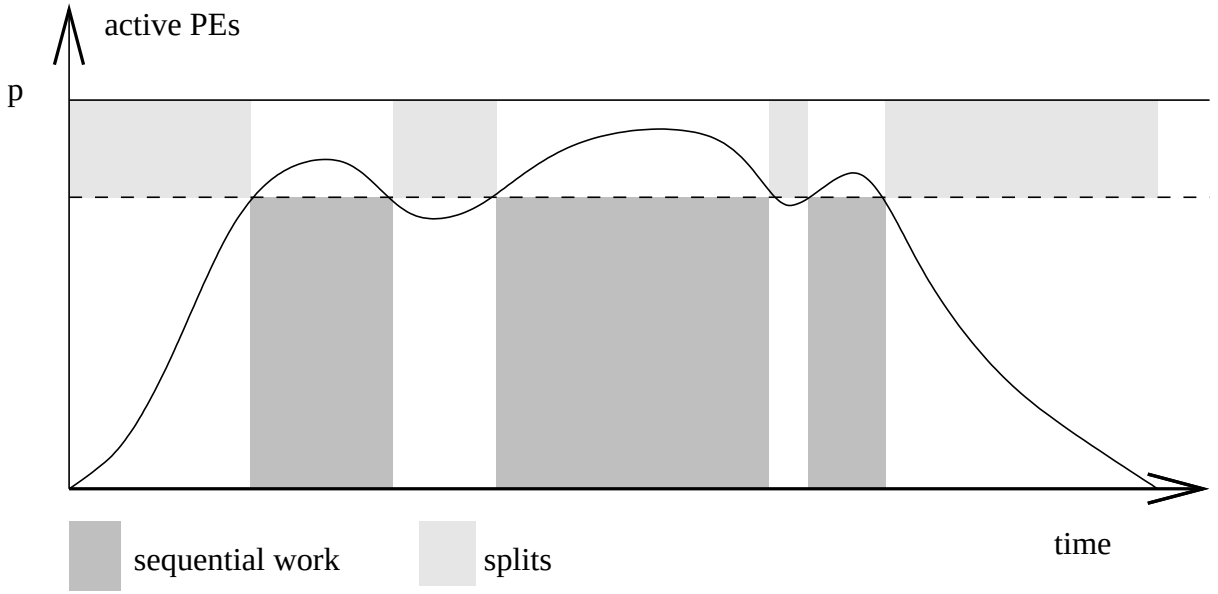
send empty subproblems the rest

if $T(P) = 0$ **then** receive P

Analysis

Satz 6. *For all $\varepsilon > 0$ there is a choice of Δt and m such that*

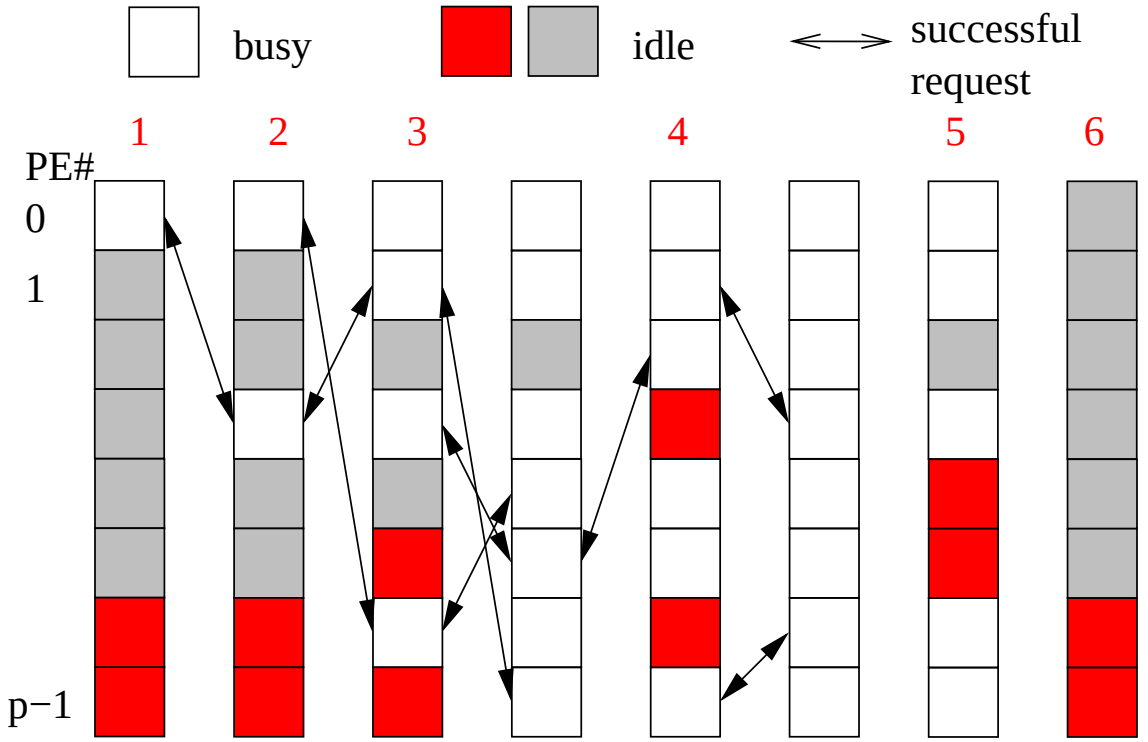
$$T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{p} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h\left(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}\right)\right) \quad .$$



Bounding Idleness

Lemma 7.

Let $m < p$ with $m \in \Omega(p)$.
Then $\tilde{\mathrm{O}}(h)$ iterations
with at least
m empty subproblems
suffice to ensure
$\forall P : \mathrm{gen}(P) \geq h$.



Busy phases

Lemma 8. *There are at most* $\frac{T_{\text{seq}}}{(p-m)\Delta t}$ *iterations with* $\leq m$ *idle PEs at their end.*

A Simplified Algorithm

P, P' : Subproblem

P := **if** $i_{\text{PE}} = 0$ **then** P_{root} **else** $P_\emptyset$

while not finished

P := work$(P, \Delta t)$

select a global value $0 \leq s < n$ uniformly at random

if $T(P@i_{\text{PE}} - s \bmod p) = 0$ **then**

$(P, P@i_{\text{PE}} - s \bmod p)$:= split(P)

Satz 9. *For all $\varepsilon > 0$ there is a choice of Δt and m such that*

$$T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{p} + \tilde{\text{O}}\left(T_{\text{atomic}} + h(T_{\text{rout}}(l) + T_{\text{split}})\right) \quad .$$

Asynchronous Random Polling

P, P' : Subproblem

P := **if** $i_{\text{PE}} = 0$ **then** P_{root} **else** $P_\emptyset$

while no global termination yet **do**

  if $T(P) = 0$ **then** send a request to a random PE

  else $P := \text{work}(P, \Delta t)$

  if there is an incoming message M **then**

    if M is a request from PE j **then**

      $(P, P') := \text{split}(P)$

      send P' to PE j

    else

      $P := M$

Analysis

Satz 10.

$$\mathbb{E}T_{\text{par}} \leq (1+\varepsilon)\frac{T_{\text{seq}}}{p} + \mathcal{O}\left(T_{\text{atomic}} + h\left(\frac{1}{\varepsilon} + T_{\text{rout}} + T_{\text{split}}\right)\right)$$

for an appropriate choice of Δt.

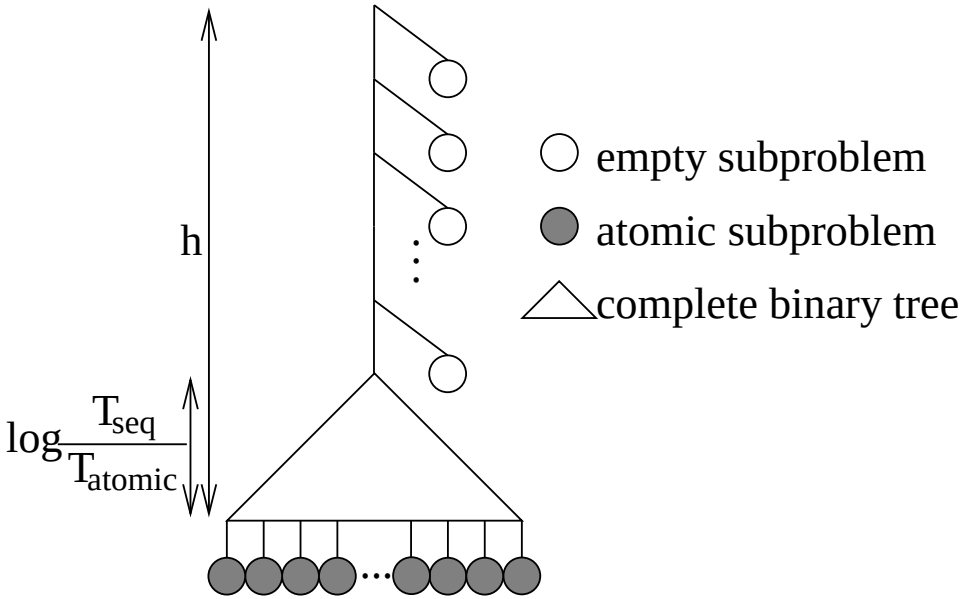
A Trivial Lower Bound

Satz 11. *For* **all** *tree shaped computations*

$$T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{p} + T_{\text{atomic}} + T_{\text{coll}} + T_{\text{split}} \log p\right) \quad .$$

if efficiency in $\Omega(1)$ *shall be achieved.*

Many Consecutive Splits

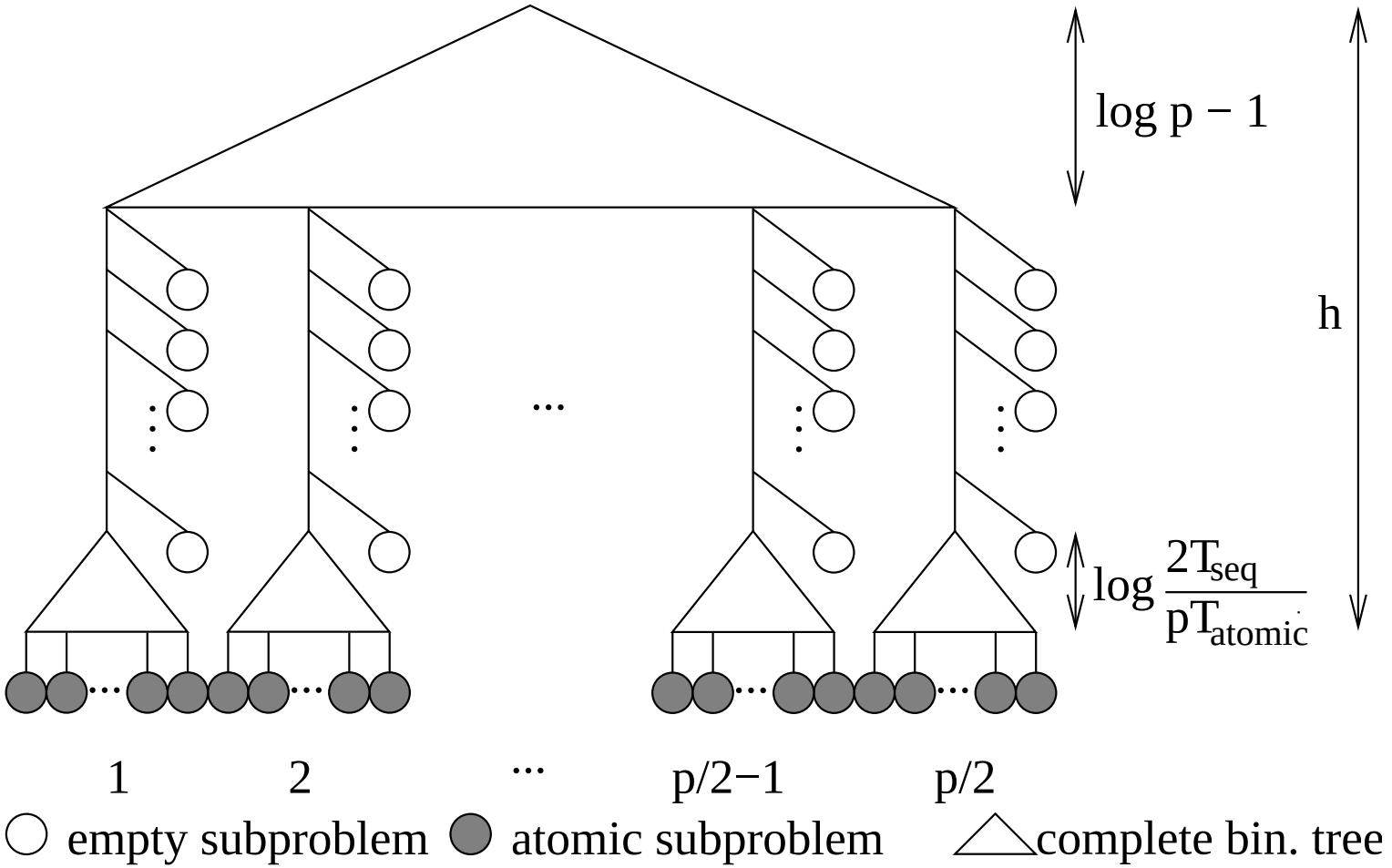


Additional

$$h - \log \frac{T_{\text{seq}}}{T_{\text{atomic}}}$$

term.

Many Splits Overall



Satz 12. *Some problems need at least*

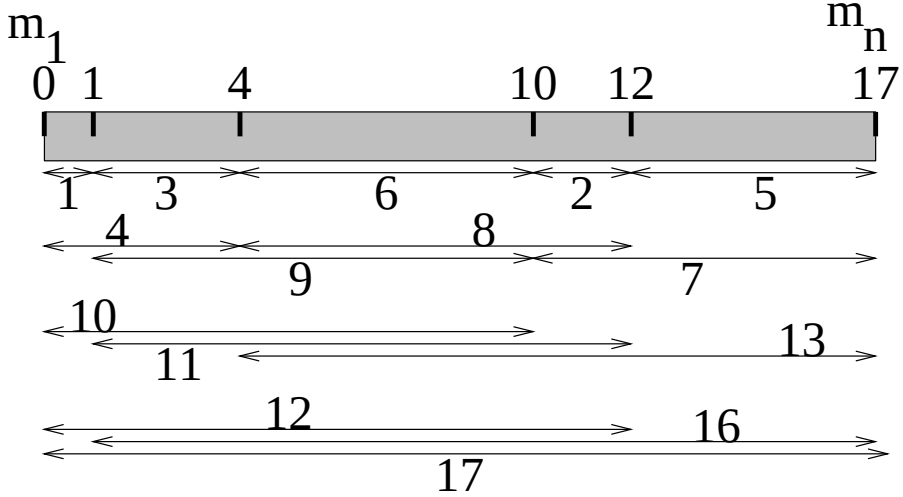
$$\frac{p}{2}\left(h - \log\frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)$$

splits for efficiency $\geq \frac{1}{2}$.

Korollar 13. *Receiver initiated algorithms need a corresponding number of communications.*

Satz 14 (Wu and Kung 1991)**.** *A similar bound holds for all deterministic load balancers.*

Golomb-Rulers

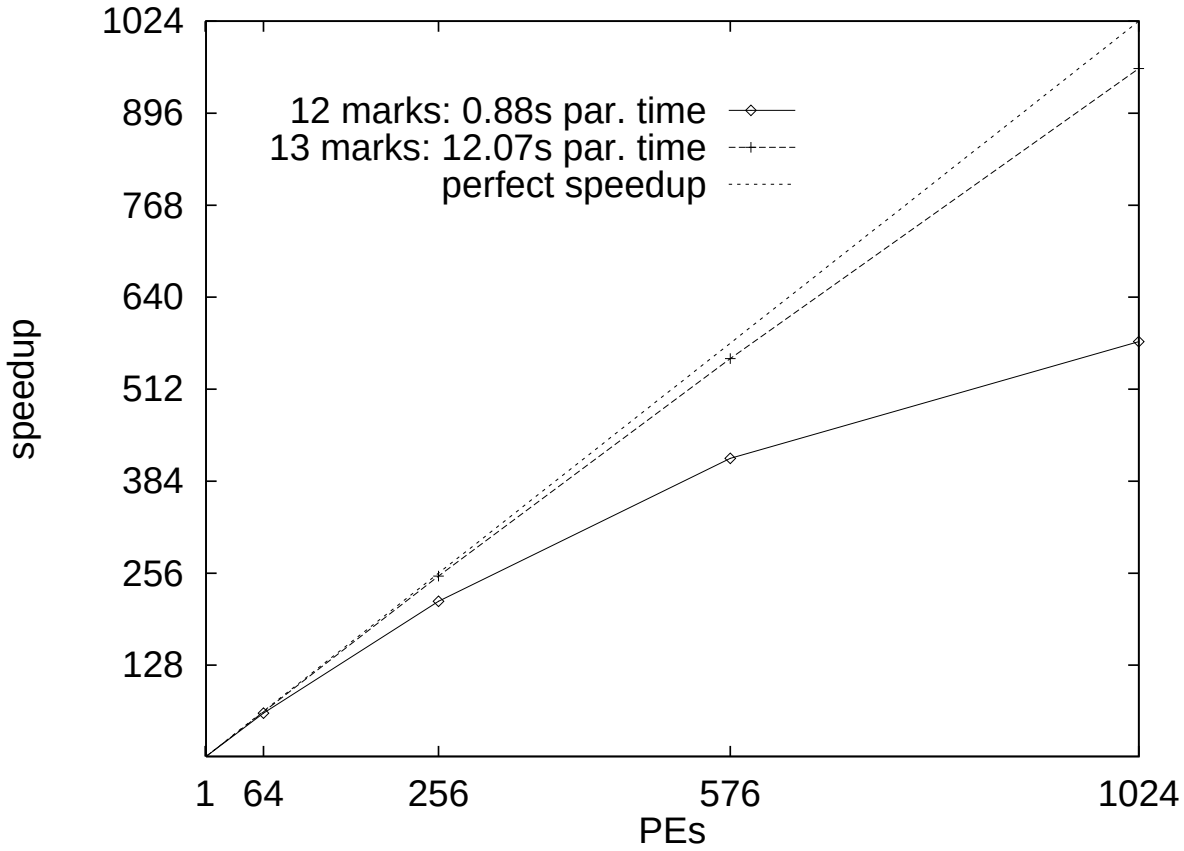


- Total length m
- find n marks $\{m_1, \ldots m_n\} \subseteq \mathbb{N}_0$
- $m_1 = 0, m_n = m$
- $|\{m_j - m_i : 1 \leq i < j \leq n\}| = n(n-1)/2$

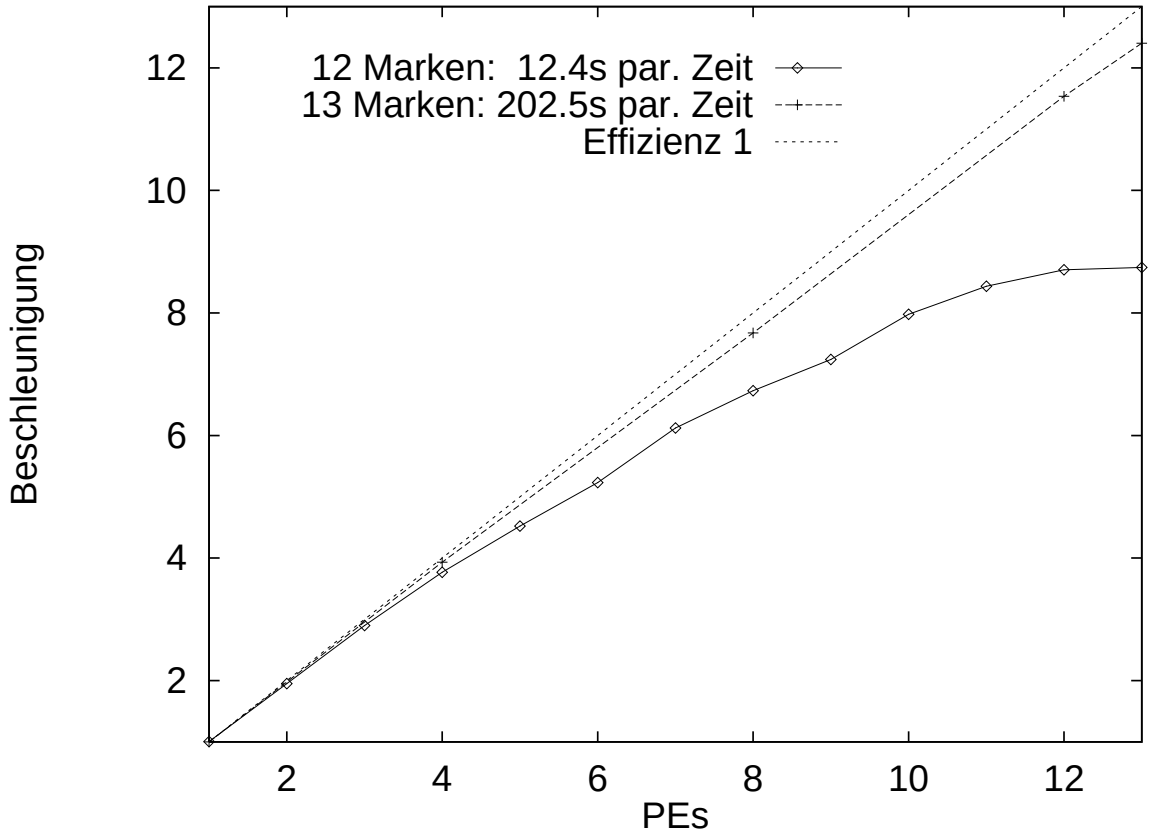
Applications: Radar astronomy, codes, ...

Many Processors

- Parsytec GCel-3/1024 with COSY (PB)
- Verification search



LAN



- Differing PE-Speeds (even dynamically) are unproblematic.
- Even complete suspension OK as long as requests are answered.

The $0/1$-Knapsack Problem

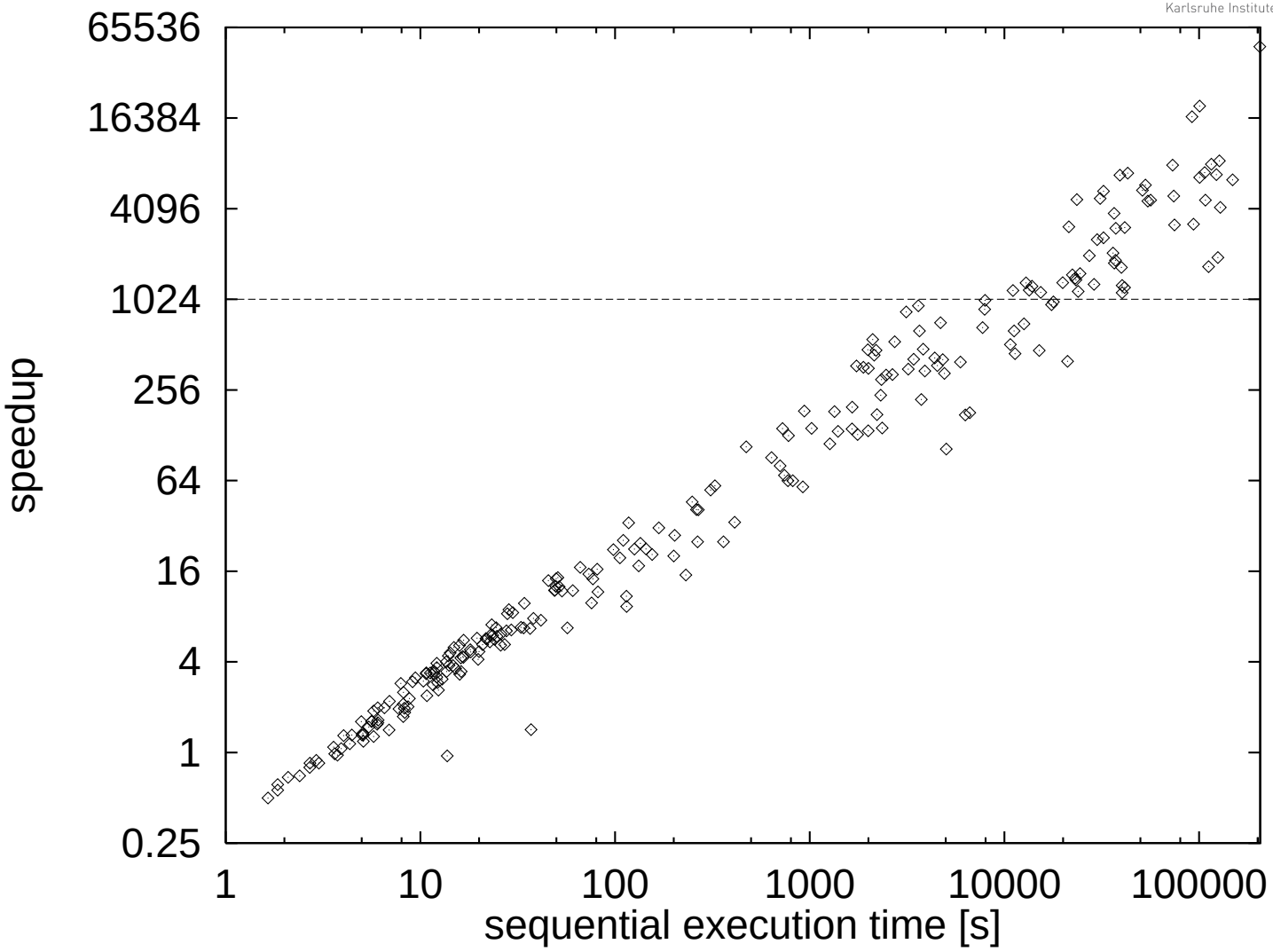
- m items
- maximum knapsack weight M
- item weights w_i
- item profits p_i
- Find $x_i \in \{0,1\}$ such that
  - $\sum w_i x_i \leq M$
  - $\sum p_i x_i$ is maximized

Best known approach for large m:

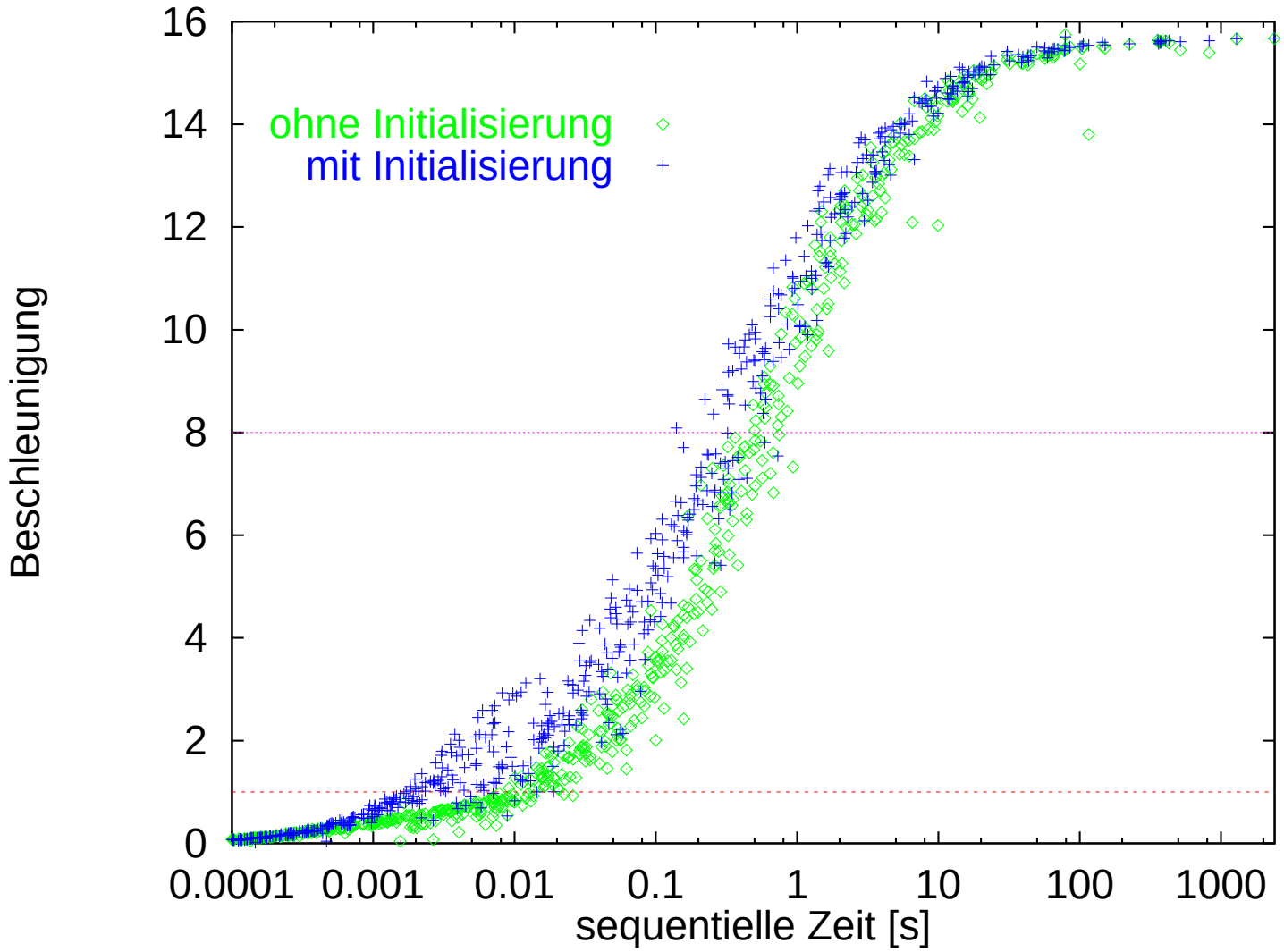
- Depth-first branch-and-bound
- Bounding function based on a the relaxation $x_i \in [0,1]$. (Can be computed in $\mathcal{O}(\log m)$ steps.)

Superlinear Speedup

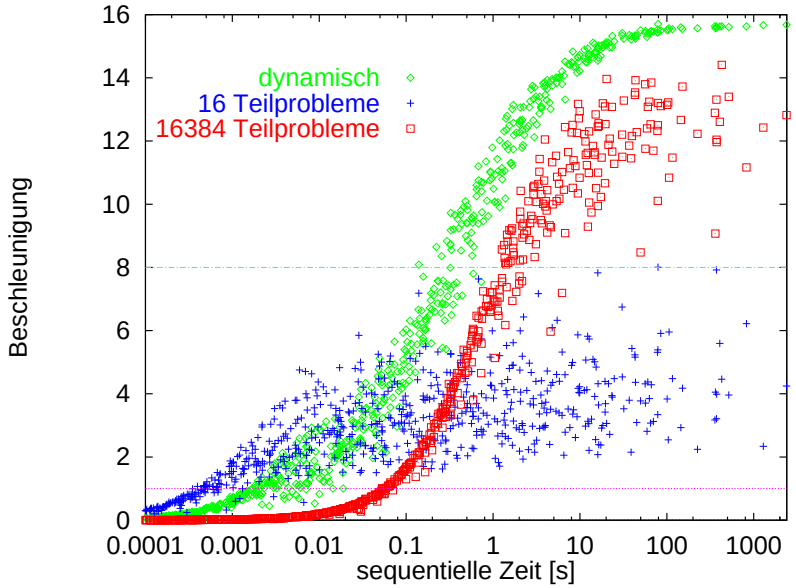
- Parsytec GCel-3/1024 under COSY (PB)
- 1024 processors
- 2000 items
- Splitting on all levels
- 256 random instances at the border between simple and difficult
- overall $1410\times$ faster than seq. computation!

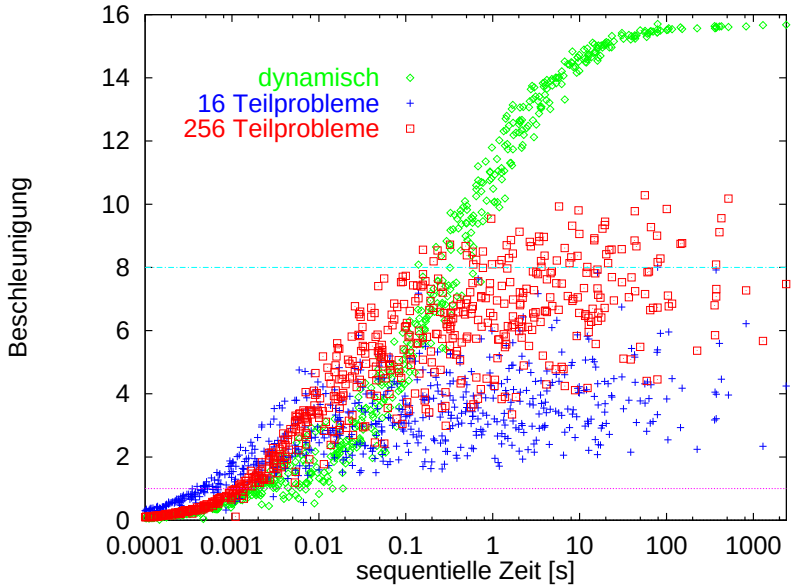

65536
16384
4096
1024
256
64
16
4
1
0.25
speedup
1
10
100
1000
10000
100000
sequential execution time [s]

Fast Initialization



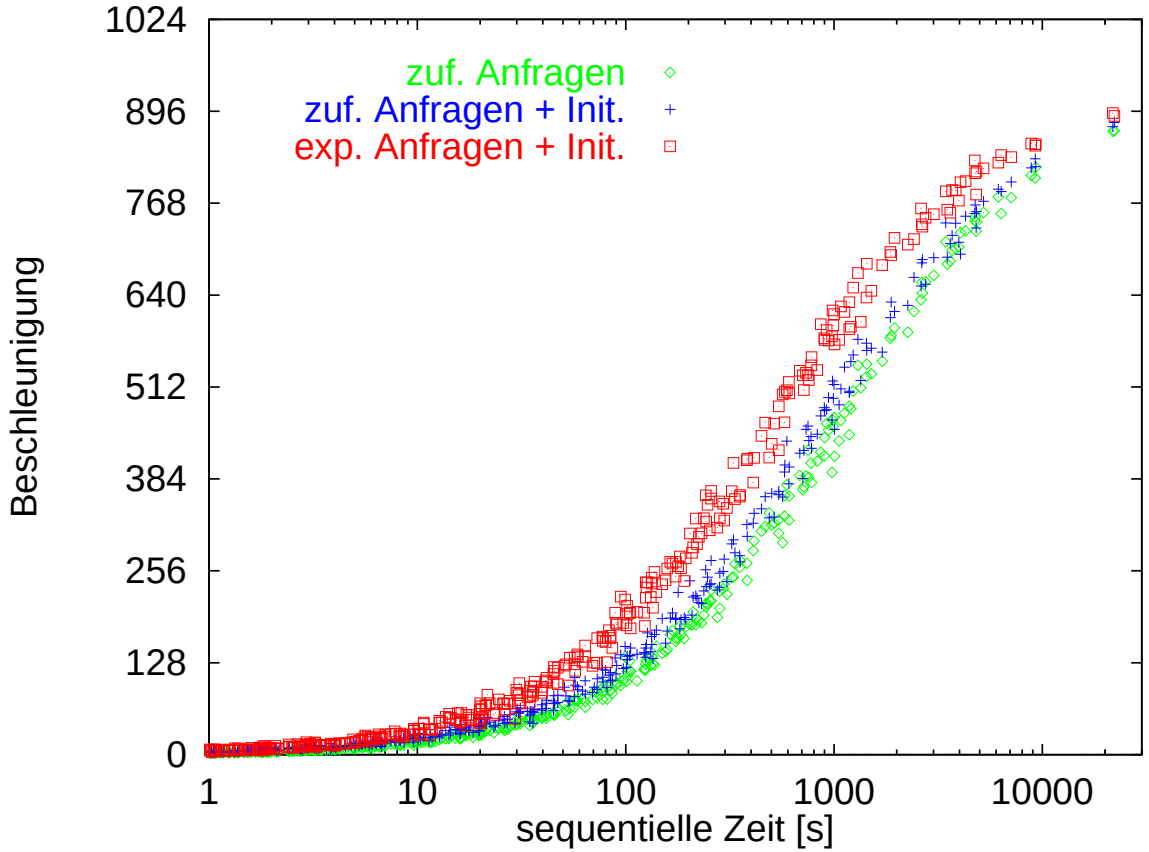
Static vs Dynamic LB





Beyond Global Polling

- Randomized Initialization
- Asynchronously increase polling range (exponentially)



Scalability Comparison Independ. Jobs

$\bar{t}$ =average job size

$\hat{t}$ =maximum job size

T =required total work for parallel execution time $(1+\varepsilon)\frac{\text{total work}}{p}$

Algorithm	$T=\Omega(\cdots)$	Remarks
prefix sum	$\frac{p}{\varepsilon}(\hat{t}+\alpha\log p)$	known task sizes
master-worker	$\frac{p}{\varepsilon}\cdot\frac{\alpha p}{\varepsilon}\cdot\frac{\hat{t}}{\bar{t}}$	bundle size $\sqrt{\frac{m\alpha}{\hat{t}}}$
randomized static	$\frac{p}{\varepsilon}\cdot\frac{\log p}{\varepsilon}\cdot\hat{t}$	randomized
work stealing	$\frac{p}{\varepsilon}(\hat{t}+\alpha\log p)$	randomized

Game Tree Search

- Naive Parallelization yields only Speedup $\mathcal{O}(\sqrt{n})$.
- Young Brother Wait Concept (Feldmann et al.)
- Tradeoff between Speculativity and Sequentialization
- Propagate window updates
- Combine with global transposition table

MapReduce in 10 Minutes

[Google, DeanGhemawat OSDI 2004] siehe auch Wikipedia

Framework zur Verarbeitung von Multimengen von $(\text{key}, \text{value})$ Paaren.

// $M \subseteq K \times V$

// MapF : $K \times V \to K' \times V'$

// ReduceF : $K' \times 2^{V'} \to V''$

Function mapReduce$(M, \text{MapF}, \text{ReduceF}) : V''$

$M' := \{\text{MapF}((k,v)) : (k,v) \in M\}$ // easy (load balancing?)

sort(M') // basic toolbox

forall k' with $\exists (k',v') \in M'$ **dopar** // easy

$s := \{v' : (k',v') \in M'\}$

$S := S \cup (k',s)$

return $\{\text{reduceF}(k',s) : (k',s) \in S\}$ // easy (load balancing?)

Refinements

- Fault Tolerance
- Load Balancing using hashing (default) und Master-Worker
- Associative commutative reduce functions

Examples

- Grep
- URL access frequencies
- build inverted index
- Build reverse graph adjacency array

Graph Partitioning

Contraction

while $|V| > c \cdot k$ **do**

find a matching $M \subseteq E$

contract M // similar to MST algorithm (more simple)

save each generated level

Finding a Matching

Find approximate max. weight matching wrt edge rating

$$\text{expansion}(\{u,v\}) := \frac{\omega(\{u,v\})}{c(u)+c(v)}$$

$$\text{expansion}^*(\{u,v\}) := \frac{\omega(\{u,v\})}{c(u)c(v)}$$

$$\text{expansion}^{*2}(\{u,v\}) := \frac{\omega(\{u,v\})^2}{c(u)c(v)}$$

$$\text{innerOuter}(\{u,v\}) := \frac{\omega(\{u,v\})}{\text{Out}(v)+\text{Out}(u)-2\omega(u,v)}$$

Approx. max. weighted Matching

todo

Graph Partitioning Future Work

- Understand edge ratings
- Scalable parallel weighted Matching code
- Hypergraph partitioning
- Handling exact balance
- Max. Flow. based techniques
- Parallel external, e.g., partitioning THE web graph