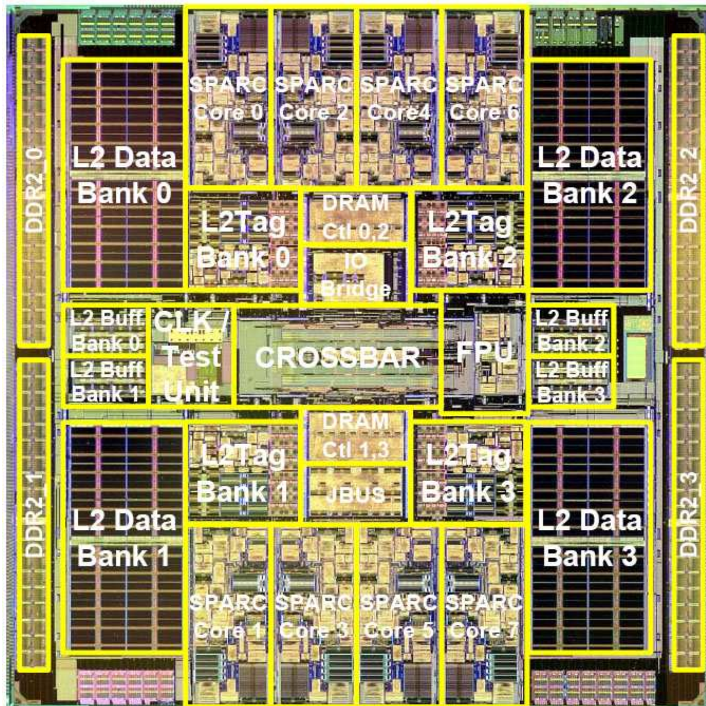
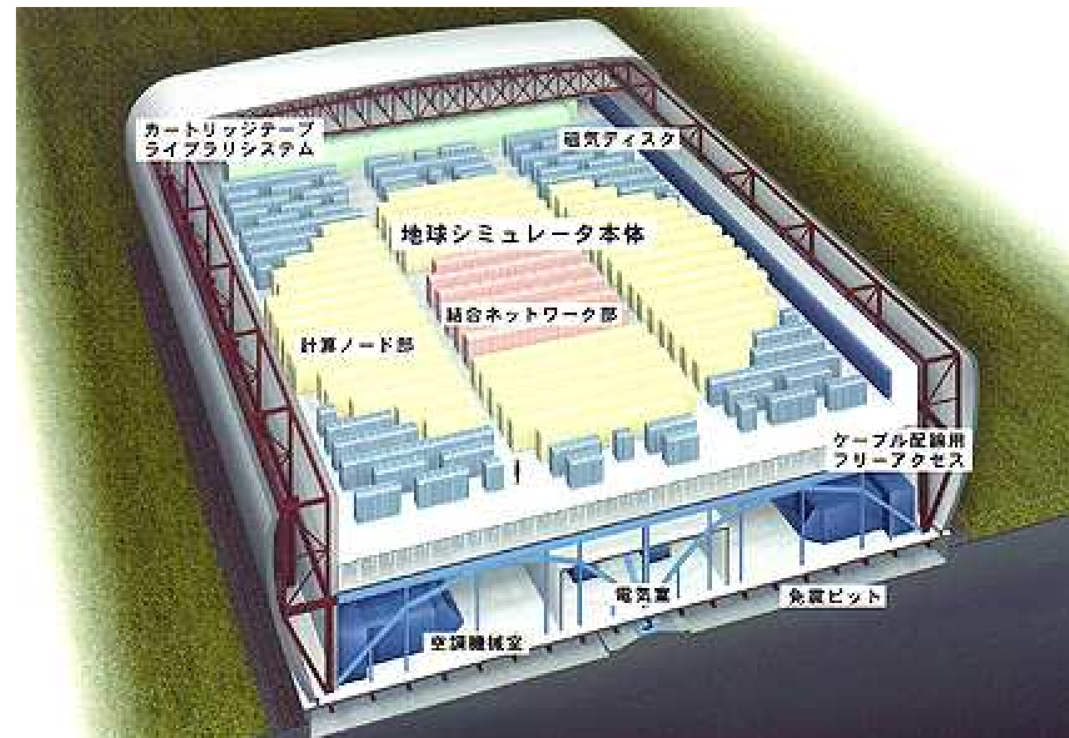
Parallele Algorithmen – Crashkurs

Peter Sanders

Institut für Theoretische Informatik —

Algorithmik II





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 $\leadsto$ 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 $\longrightarrow$ Array
- Prioritätslisten
- einfache Graphenalgorithmen
- Graphpartitionierung

Literatur

Skript

[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 Algorithmik II

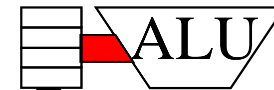
- Multicore Basisalgorithmen (Singler, EU-Projekt PEPPHER)
  $\rightsquigarrow$ z.B. g++ STL parallel mode
- geometrische Algorithmen für Multicore (Singler)
- GPU Algorithmen (Osipov, EU-Projekt PEPPHER)
  $\rightsquigarrow$ z.B. schnellster vergleichsbasierter Sortierer
- Parallele Externe Algorithmen (Daten auf Festplatte) (Osipov, Singler,. . .)
  $\rightsquigarrow$ Diverse Sortierbenchmarks
- Lastverteilungsalgorithmen (DFB Transregio SFB Invasic, Speck)
- Hauptspeicherbasierte Datenbanken (Kooperation SAP)
- Graphpartitionierung (Schulz, Osipov)

RAM/von Neumann Modell

Analyse: zähle Maschinenbefehle —

load, store, Arithmetik, Branch,. . .

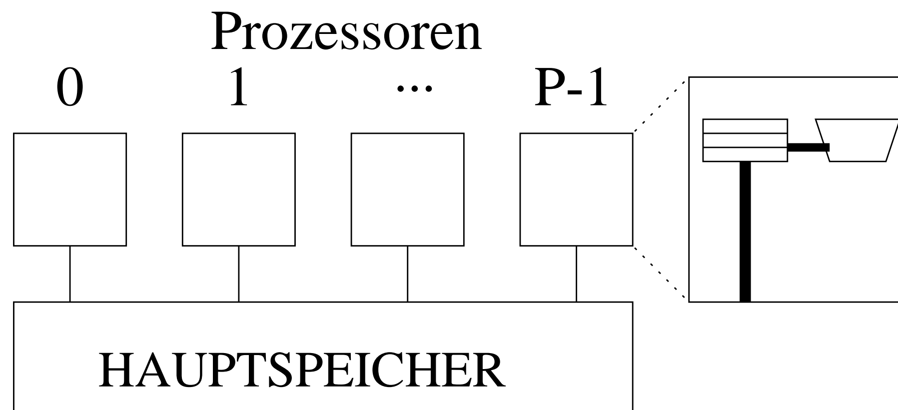
- Einfach
- Sehr erfolgreich



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?) $\longleftarrow$
- arbitrary: Irgendeiner setzt sich durch $\longleftarrow$
- 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
```

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 (!)

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 !

Synchron versus asynchron

PE index is j

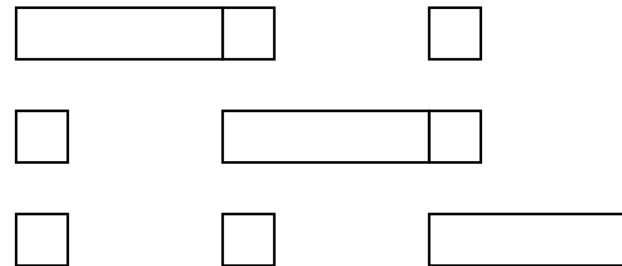
for $i = 1$ **to** p **do** (a)synchronously

  if $i = j$ **then**

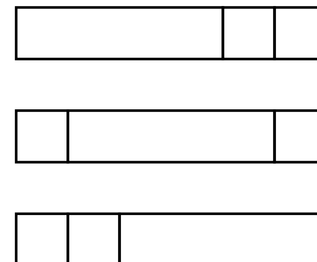
    for $k = 1$ **to** l **do** foo

  else

    foo



synchron



asynchron

pl foos versus $p + l - 1$ foos.

Analyse paralleler Algorithmen

Im Prinzip nur ein zusätzlicher Parameter: p.

Finde Ausführungszeit $T(I,p)$.

Problem: Interpretation.

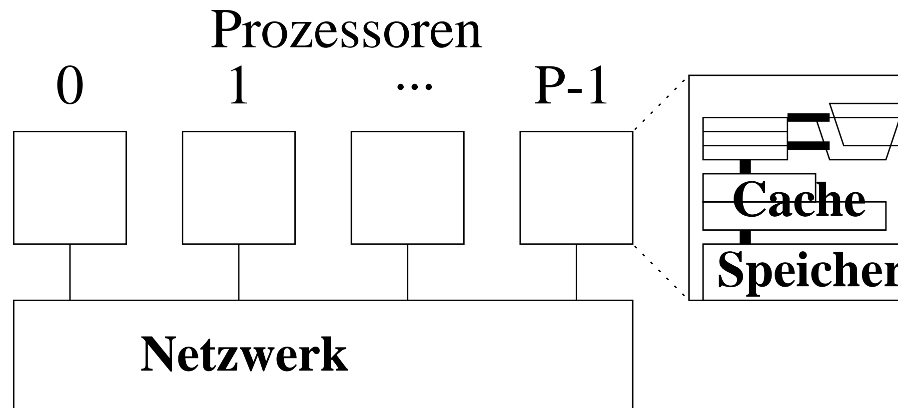
Work: $W = pT(p)$ ist ein Kostenmaß. (z.B. Max: $W = \Theta(n^2)$)

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

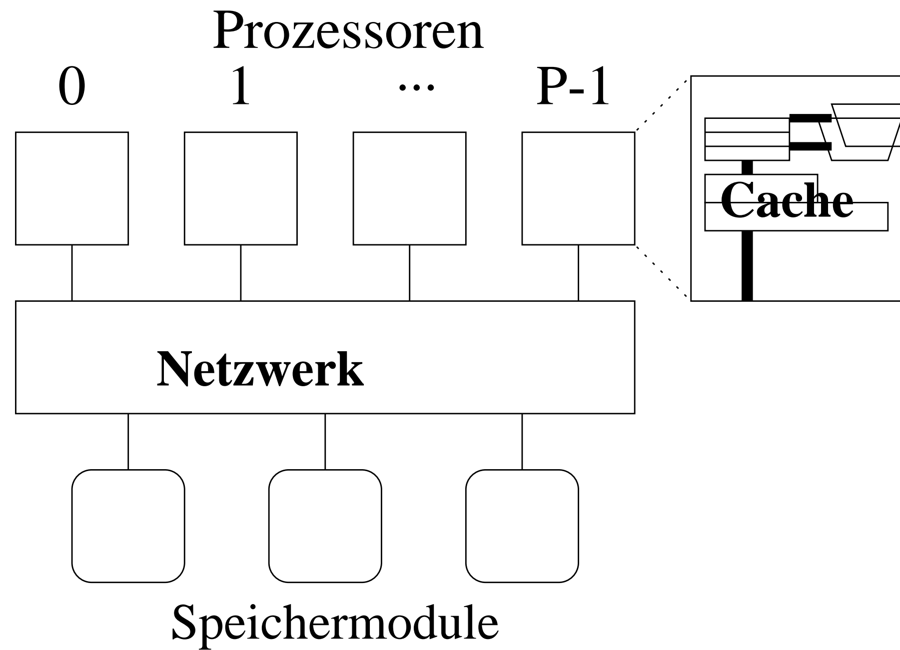
Effizienz: $E = S/p$. Ziel: $E \approx 1$ oder wenigstens $E = \Theta(1)$.
(Sinnvolles Kostenmaß?) „Superlineare Beschleunigung“: $E > 1$.
(möglich?). Beispiel Maximum: $E = \Theta(1/n)$.

PRAM vs. reale Parallelrechner

Distributed Memory



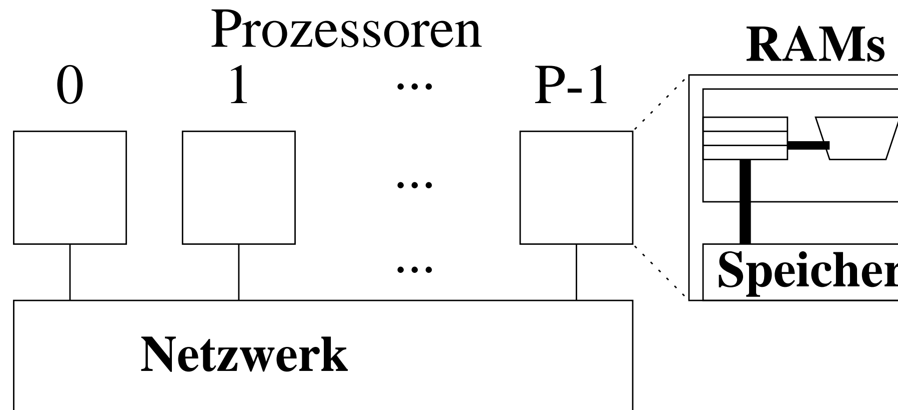
(Symmetric) Shared Memory



Probleme

- Asynchron $\rightsquigarrow$ Entwurf, Analyse, Implementierung, Debugging viele schwieriger als PRAM
- Contention (Stau) für gleiche Speichermodule. 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

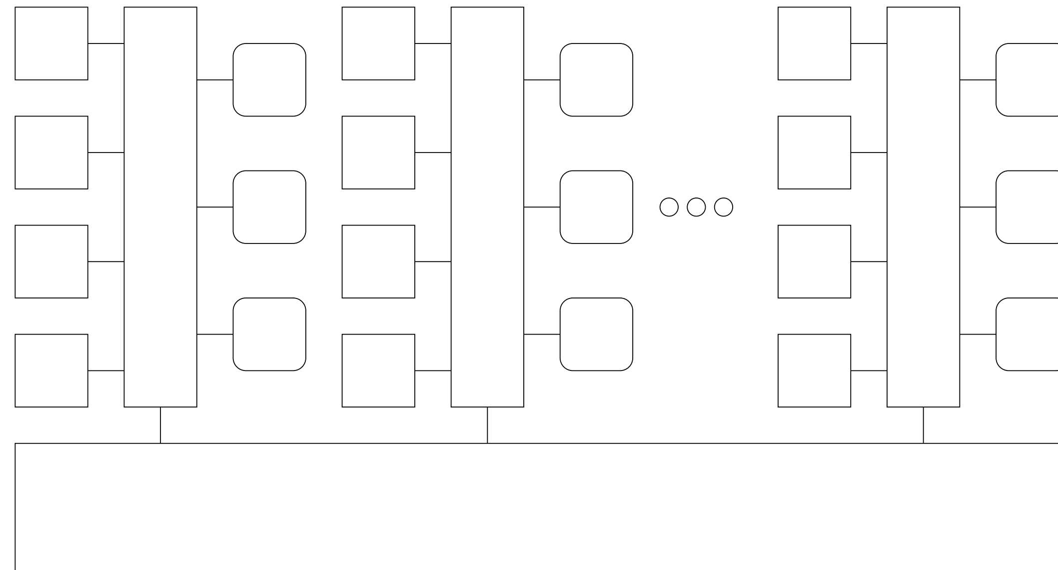
Modelle mit Verbindungsnetzwerken



- Prozessoren sind RAMs
- asynchrone Programmabarbeitung
- Interaktion durch Nachrichtenaustausch

Entscheidend ist das Kostenmodell für den Nachrichtenaustausch

Reale Maschinen Heute



Komplexe Hierarchien.

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

- Entwerfe verteilt, implementiere hierarchieangepaßt, ggf. mit shared memory

Vollständige Verknüpfung

- □ $E = V \times V$, single ported
- □ $T_{\text{comm}}(m) = T_{\text{start}} + mT_{\text{byte}}$, “vollduplex” – gleichzeitiges Senden und Empfangen möglich
- \+ Realistische Behandlung von Nachrichtenlängen
- \+ Viele Verbindungsnetze approximieren vollständige Verknüpfung
  $\Rightarrow$ sinnvolle Abstraktion
- \+ Keine überlasteten Kanten $\longrightarrow$ OK für Hardwarerouter
- \+ „künstliches“ Vergrößern v. $T_{\text{start}}, T_{\text{byte}}$
  $\longrightarrow$ OK für „schwächliche“ Netzwerke
- \+ Asynchrones Modell
- − Etwas Händewedeln bei realen Netzwerken

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* $\mathcal{O}(T_{\text{start}} \log n)$ *auf einem nachrichtengekoppelten Parallelrechner*

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

PRAM Code

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

active := 1

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

if active **then**

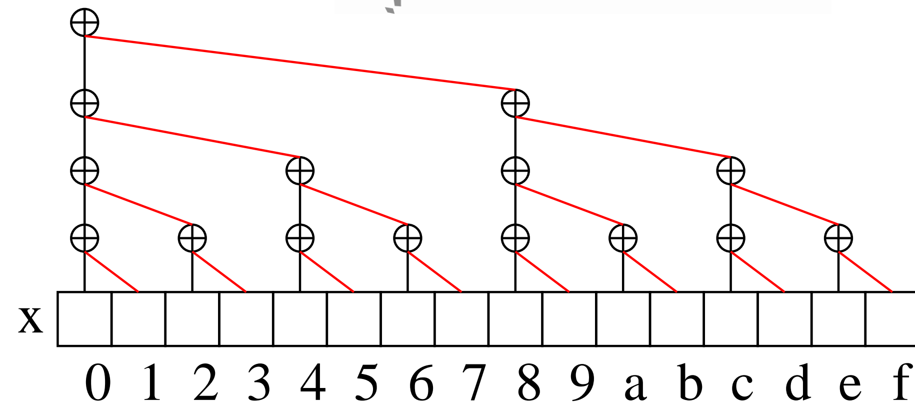
if bit k of i **then**

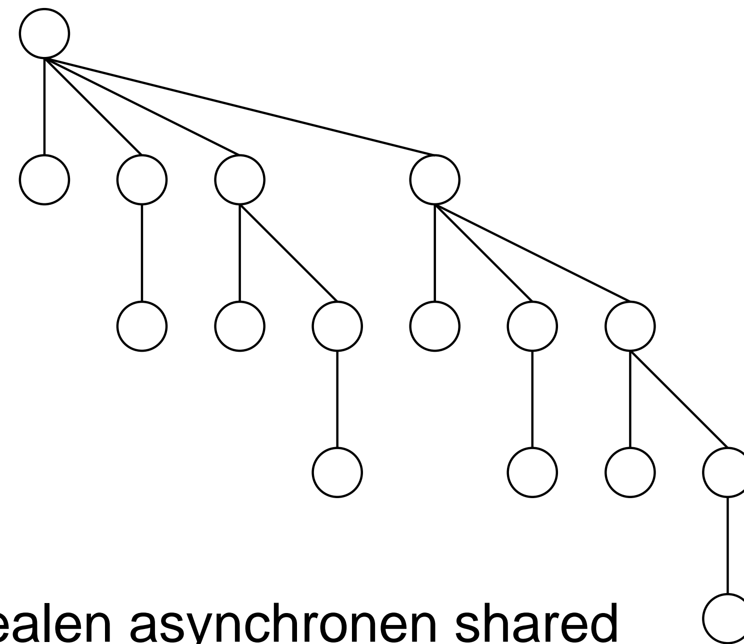
active := 0

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 $\log_2 x$

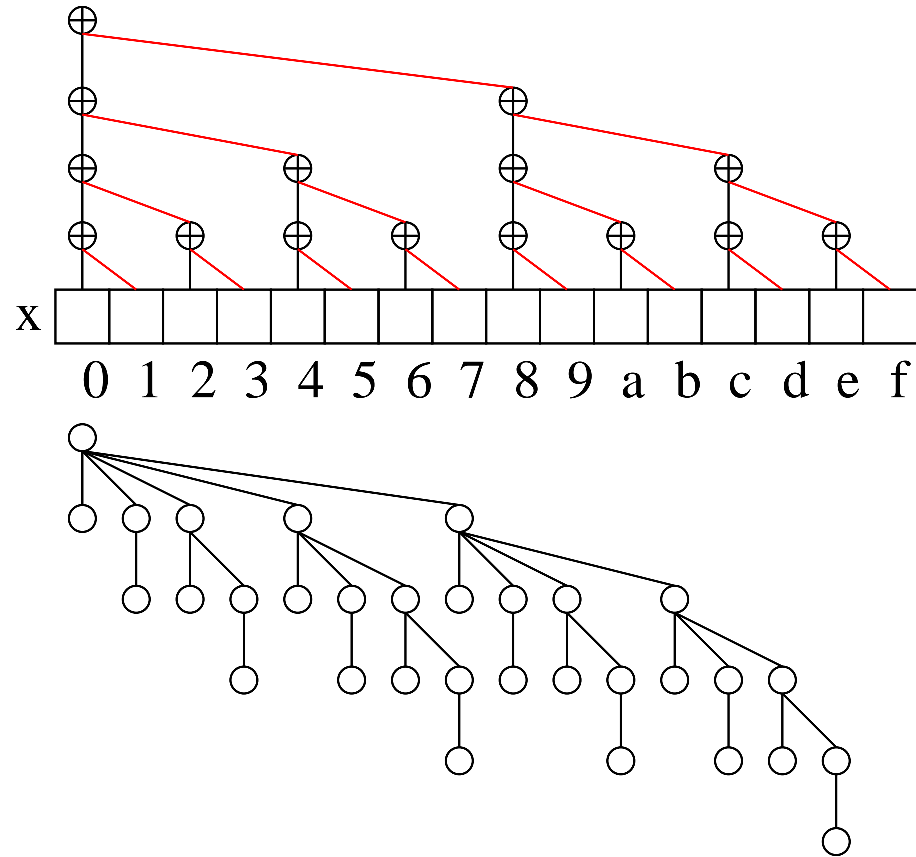
Analyse

n PEs

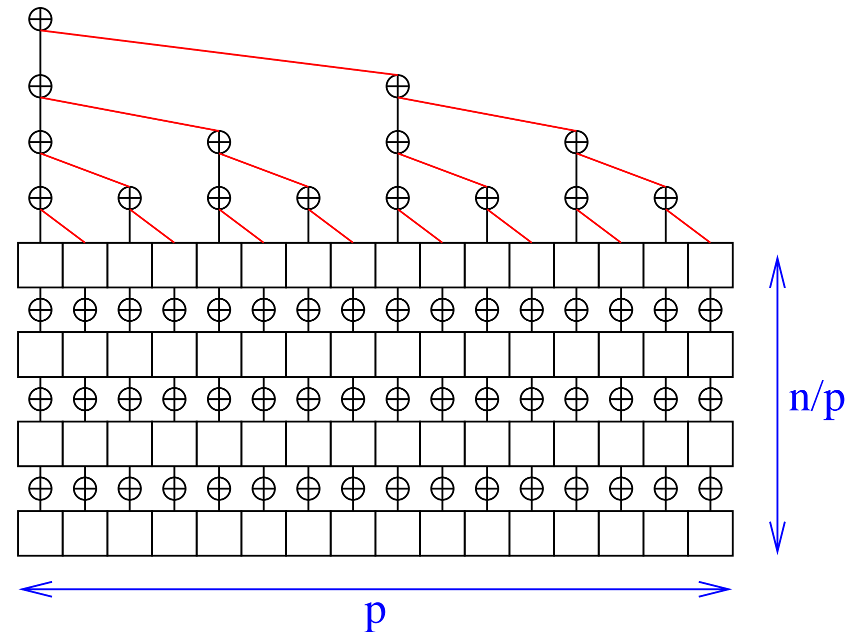
Zeit $\mathcal{O}(\log n)$

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

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



Weniger ist Mehr



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 $i - 2^k$

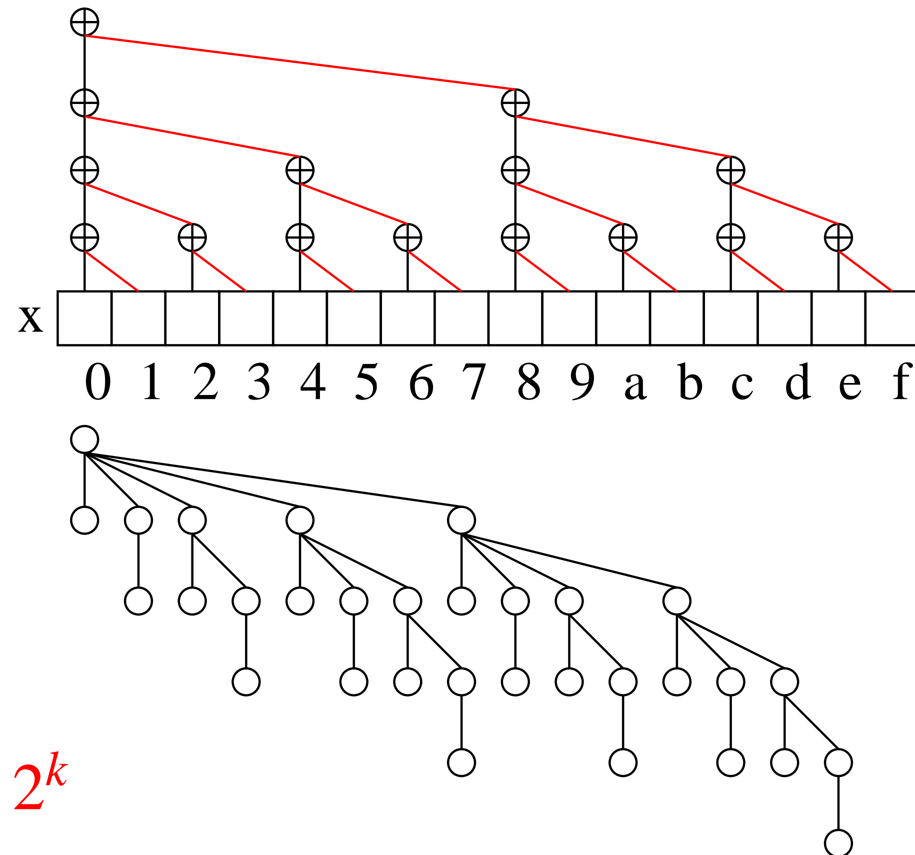
            active := 0

        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: $\Theta\left((T_{\text{start}} + T_{\text{byte}}) \log p\right)$

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**

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

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

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

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

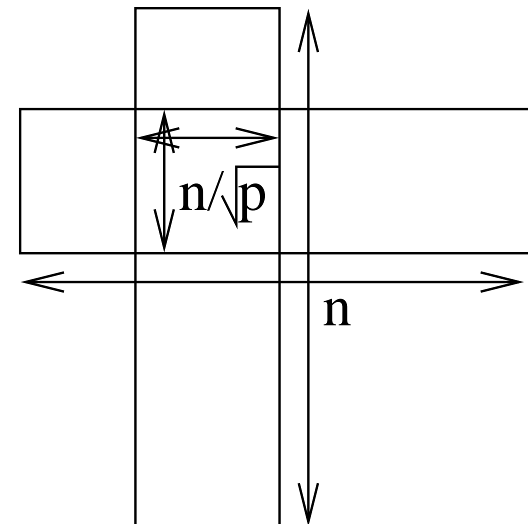
Verteilte Implementierung I

$p \leq n^2$ 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 $\Omega\left(T_{\text{byte}}\frac{n^2}{\sqrt{p}}\right)$

Verteilte Implementierung II-1

[Dekel Nassimi Sahni 81, KGGK Section 5.4.4]

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

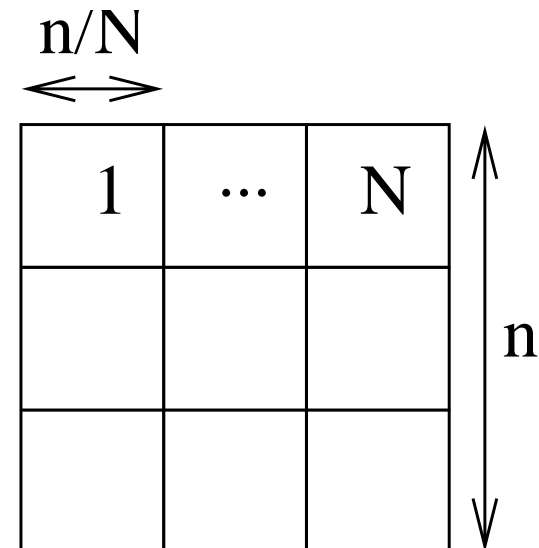
Fasse A, B, C als $N \times N$ Matrizen auf,

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 $(i,k,1)$

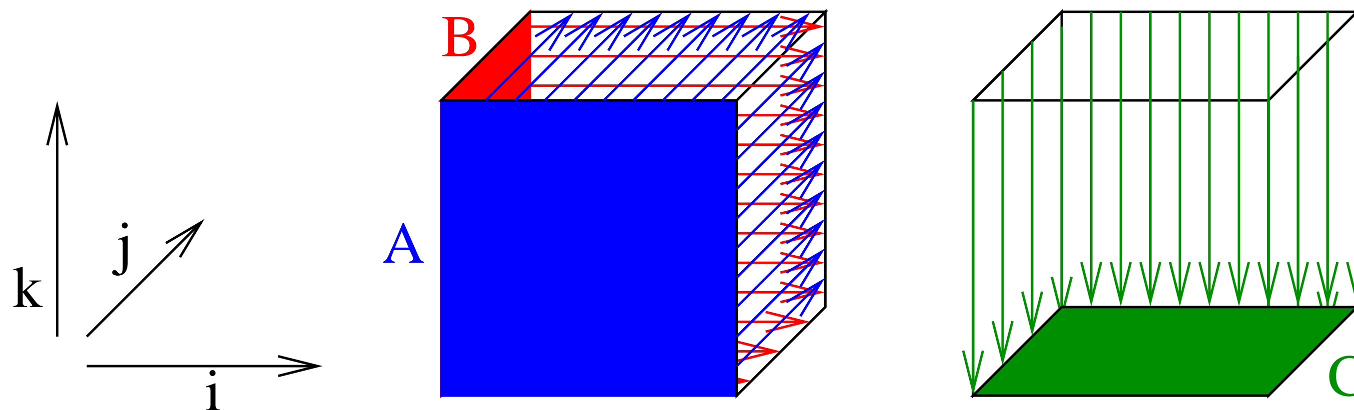
store b_{kj} in PE $(1,k,j)$

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:

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

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

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,...

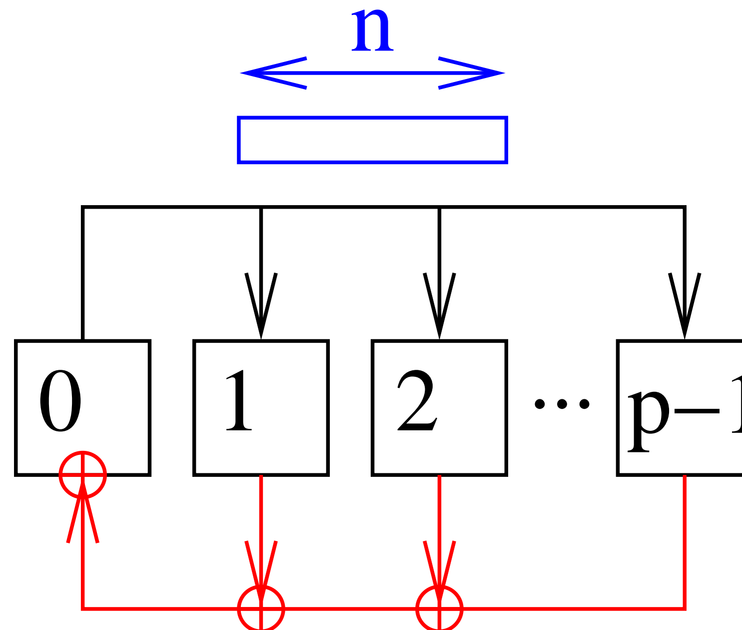
Beispiel $T_{\text{byte}} = 2^{-29}s$, $T_{\text{start}} = 2^{-17}s$, $p = 2^{12}$, $n = 2^{12}$,

8GFLOPS/PE

Broadcast (Rundruf?) und Reduktion

Broadcast: Einer für alle

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

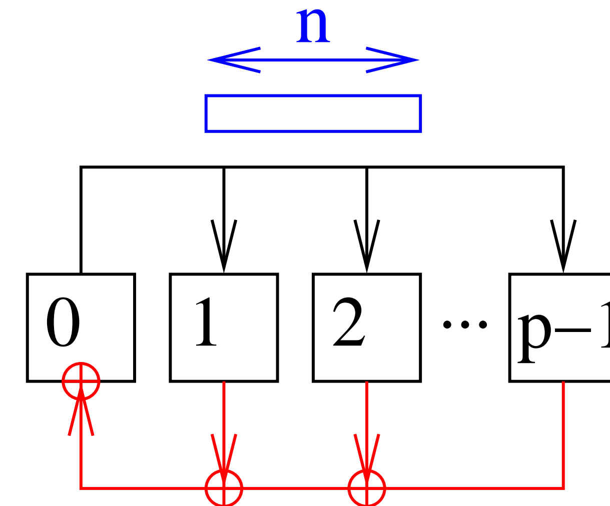


Reduktion: Alle für einen

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.

Naiver Broadcast [KGGK Abschnitt 3.2.1]

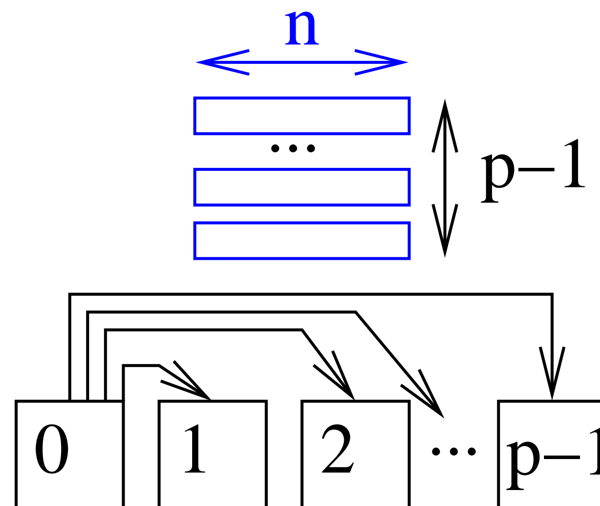
Procedure naiveBroadcast$(m[1..n])$

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

PE $i > 0$: receive m

Zeit: $(p-1)(nT_{\text{byte}} + T_{\text{start}})$

Alptraum bei der Implementierung skalierbarer Algorithmen



Binomialbaum-Broadcast

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

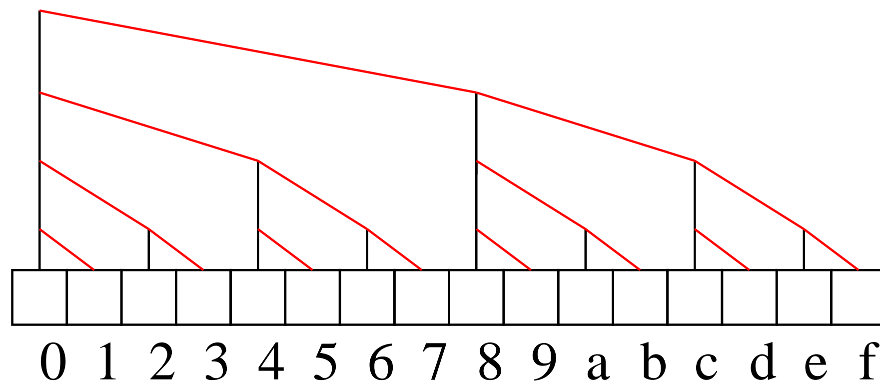
PE index $i \in \{0, \ldots, p-1\}$

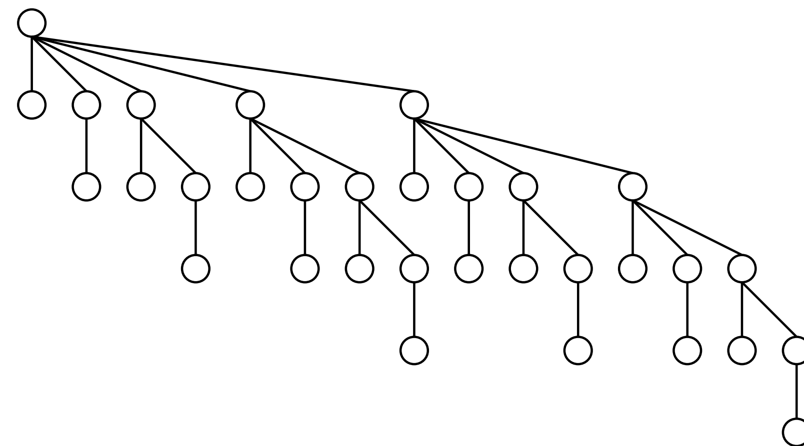
//Message m located on PE 0

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 $\geq p$

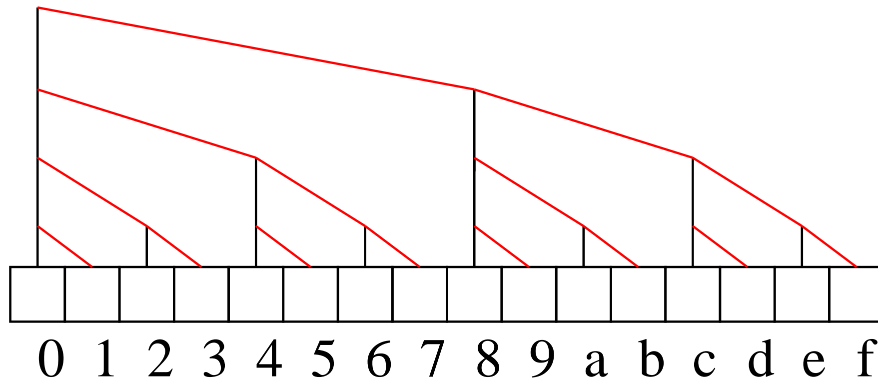


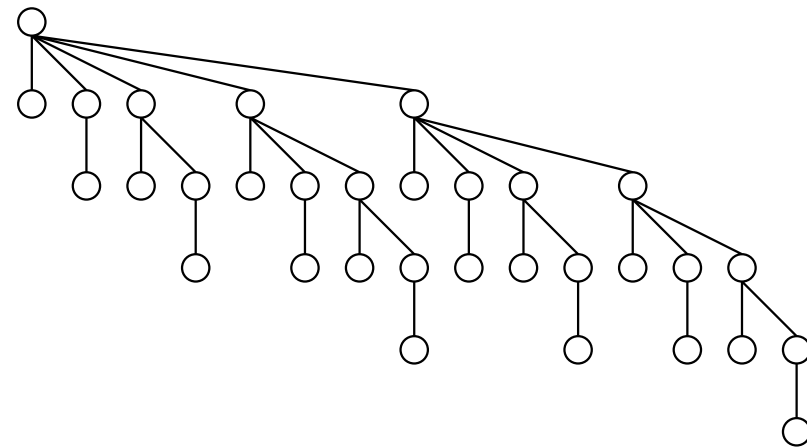


Analyse

- Zeit: $\lceil\log p\rceil\,(nT_{\text{byte}} + T_{\text{start}})$
- Optimal für $n = 1$

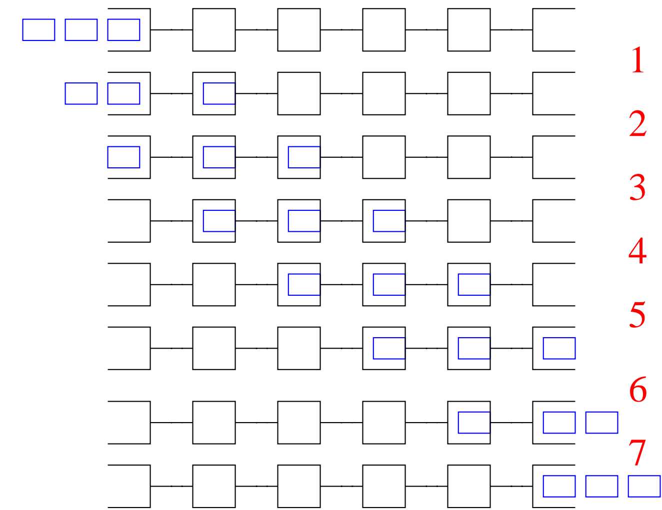
$n \cdot f(p) \leadsto n + \log p$?





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

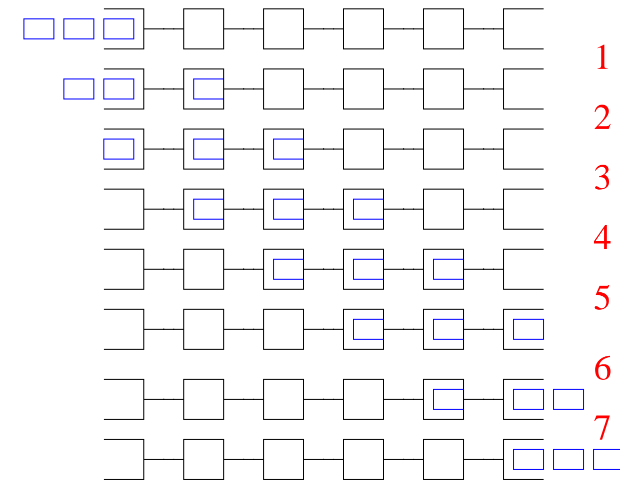
for $j := 1$ **to** $k+1$ **do**

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 $j=0$

Analyse

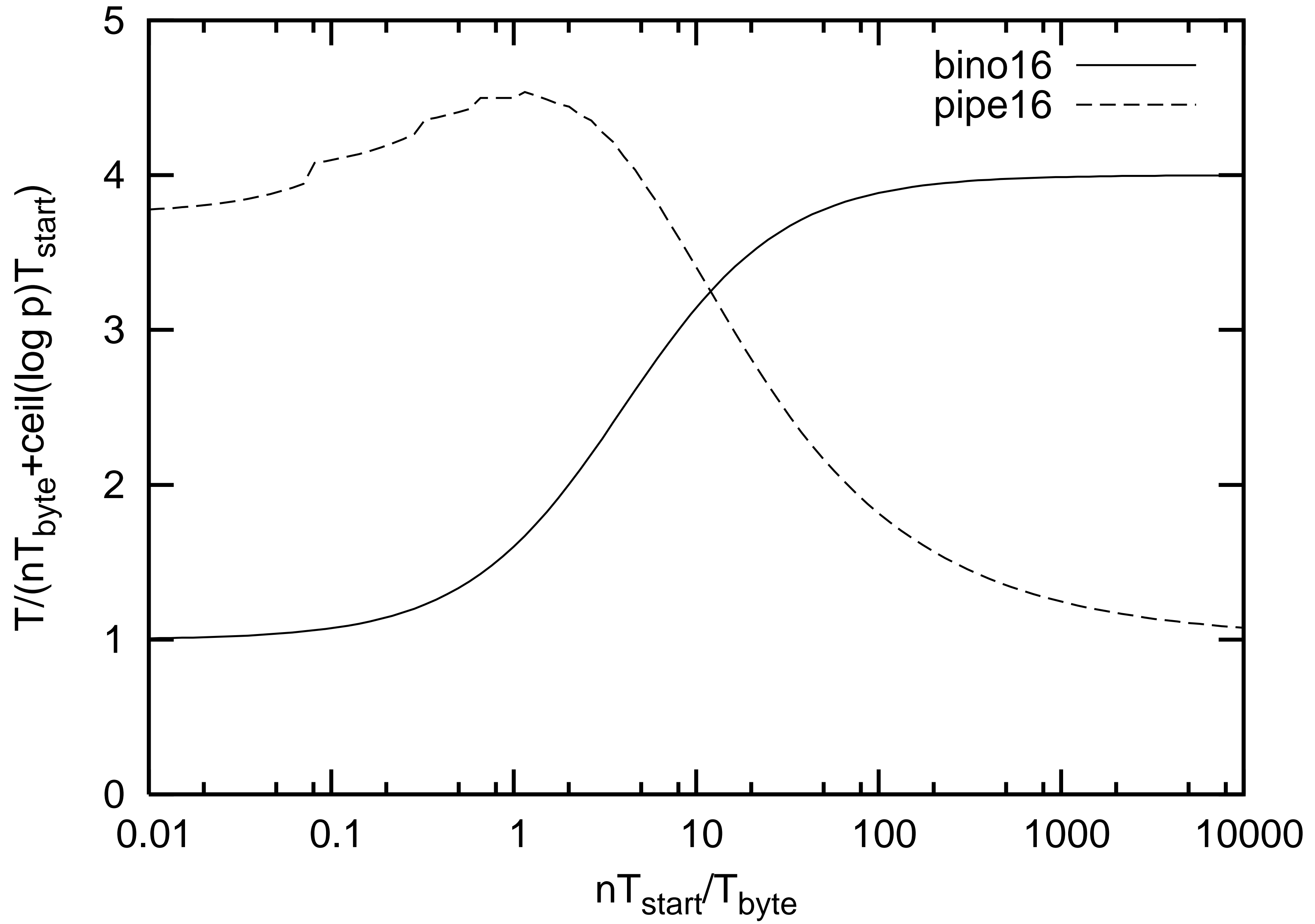


- Zeit $\frac{n}{k}T_{\text{byte}} + T_{\text{start}}$ pro *Schritt* ($\neq$ Iteration)
- $p-1$ Schritte bis erstes Paket ankommt
- Dann 1 Schritte pro weiteres Paket

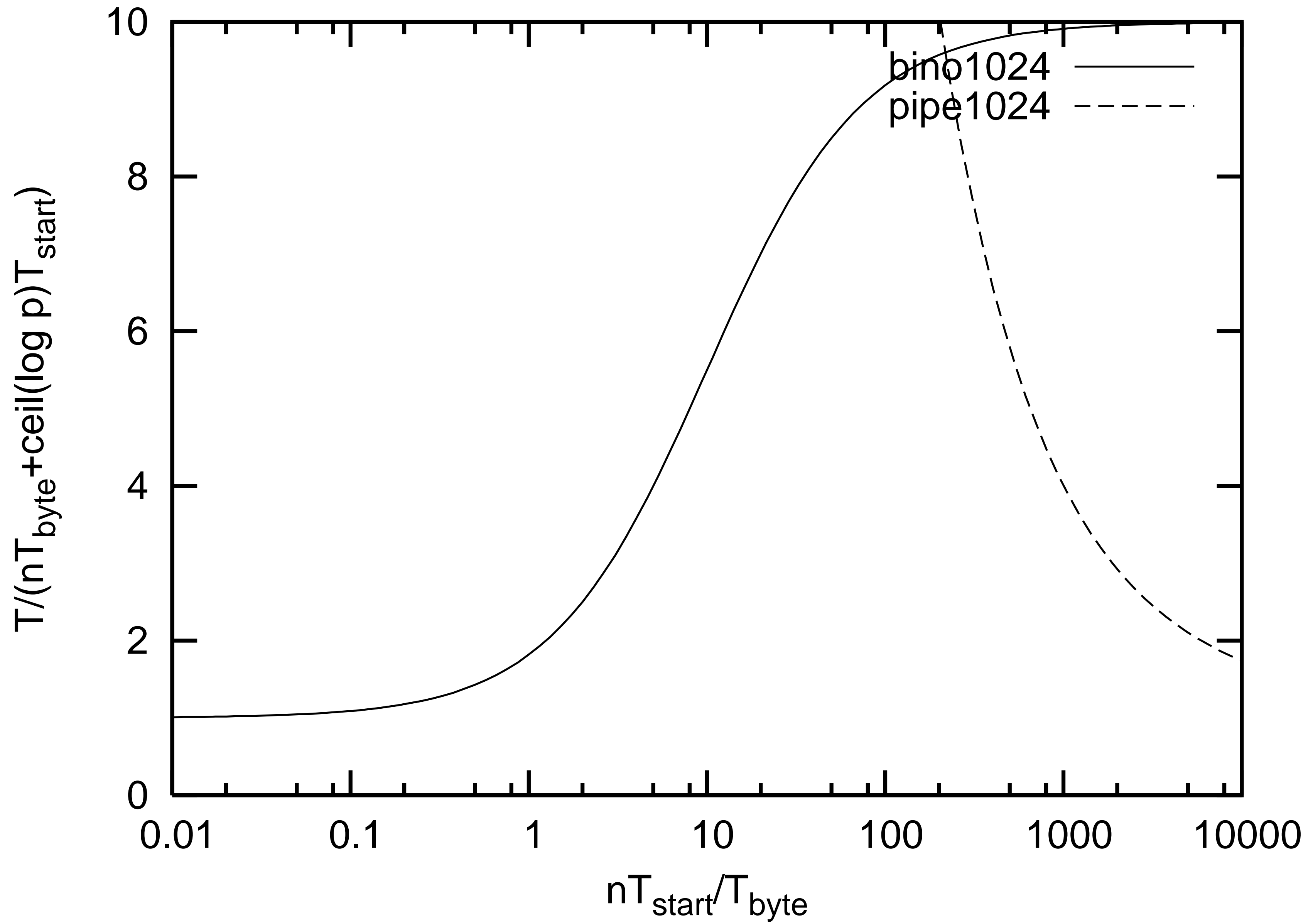
$$T(n,p,k)\colon \left(\frac{n}{k}T_{\text{byte}} + T_{\text{start}}\right)(p+k-2))$$

optimales k: $\sqrt{\frac{n(p-2)T_{\text{byte}}}{T_{\text{start}}}}$

$$T^*(n,p)\colon \approx nT_{\text{byte}} + pT_{\text{start}} + 2\sqrt{npT_{\text{start}}T_{\text{byte}}}$$



bino16
pipe16
5
4
3
2
1
0
0.01
0.1
1
10
100
1000
10000
$T/(nT_{byte}+ceil(log p)T_{start})$
nT_{start}/T_{byte}


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

Diskussion

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

$T_{\text{start}} p \rightsquigarrow T_{\text{start}} \log p$?

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

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

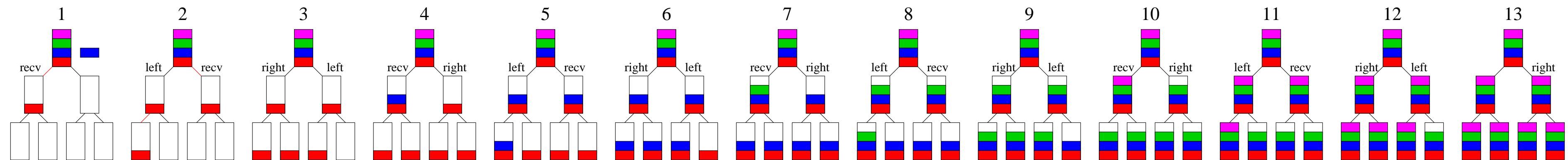
define piece j as $m[(j-1)\frac{n}{k}+1..j\frac{n}{k}]$

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

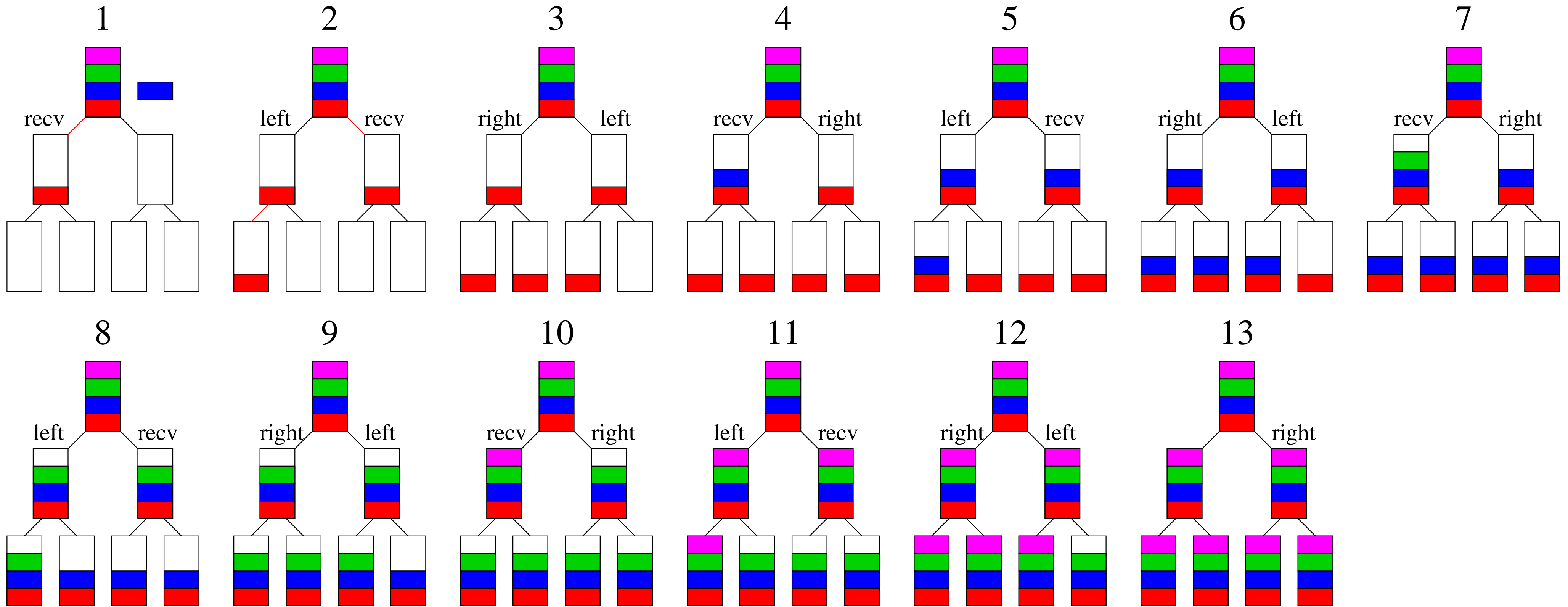
if parent exists **then** receive piece j

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

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



Beispiel



Analyse

- Zeit $\frac{n}{k}T_{\text{byte}}+T_{\text{start}}$ 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: $T(n,p,k) := (2d+3(k-1))\left(\frac{n}{k}T_{\text{byte}}+T_{\text{start}}\right)$

optimales k: $\sqrt{\frac{n(2d-3)T_{\text{byte}}}{3T_{\text{start}}}}$

eingesetzt: $T^*(n,p) = 2dT_{\text{start}}+3nT_{\text{byte}}+\mathcal{O}\left(\sqrt{ndT_{\text{start}}T_{\text{byte}}}\right)$

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**

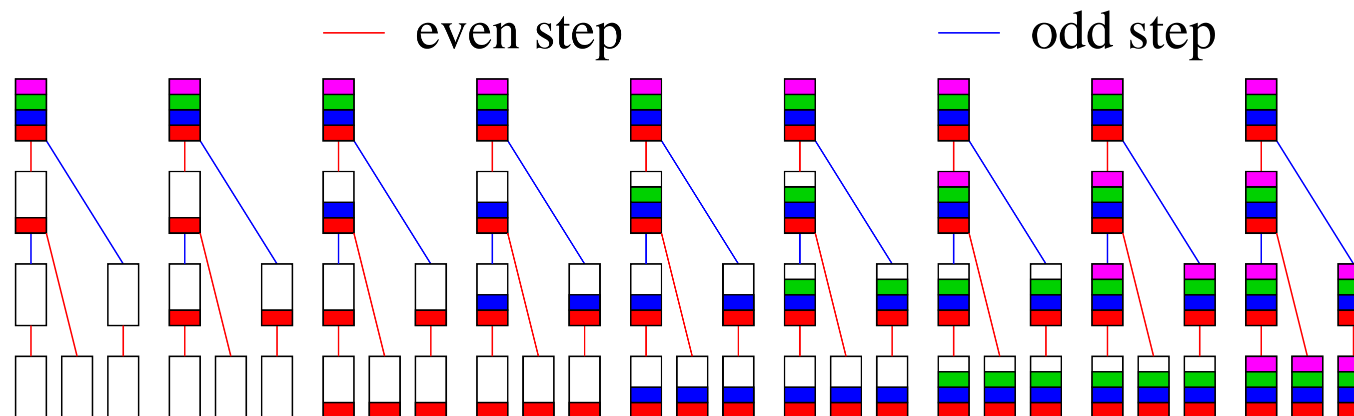
receive piece j from parent // noop for root or $j = k+1$

and, concurrently, send piece $j-1$ to child with color of parent

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

send piece j to child with color 1 color of parent

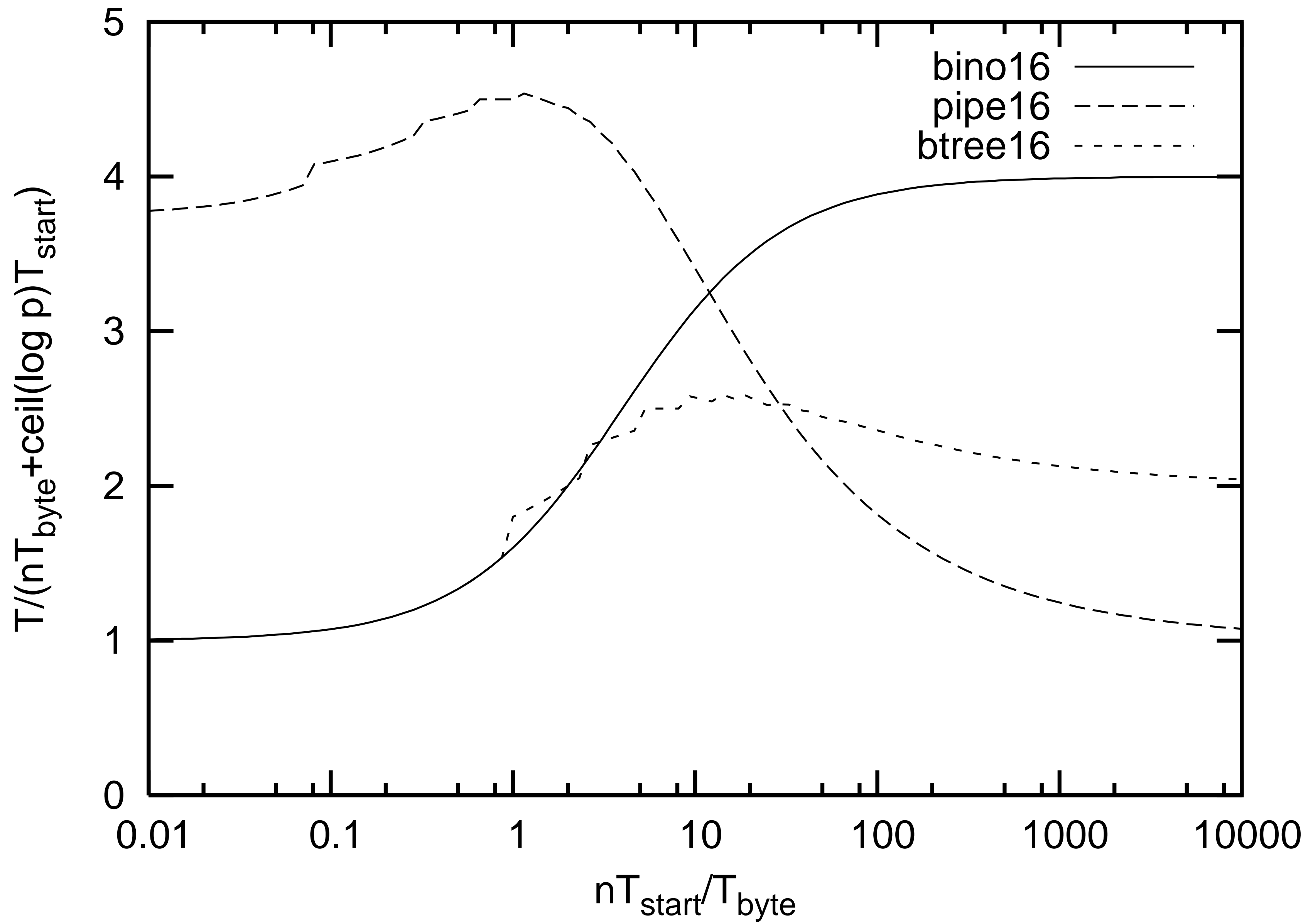
// noop if no such child or $j = k+1$

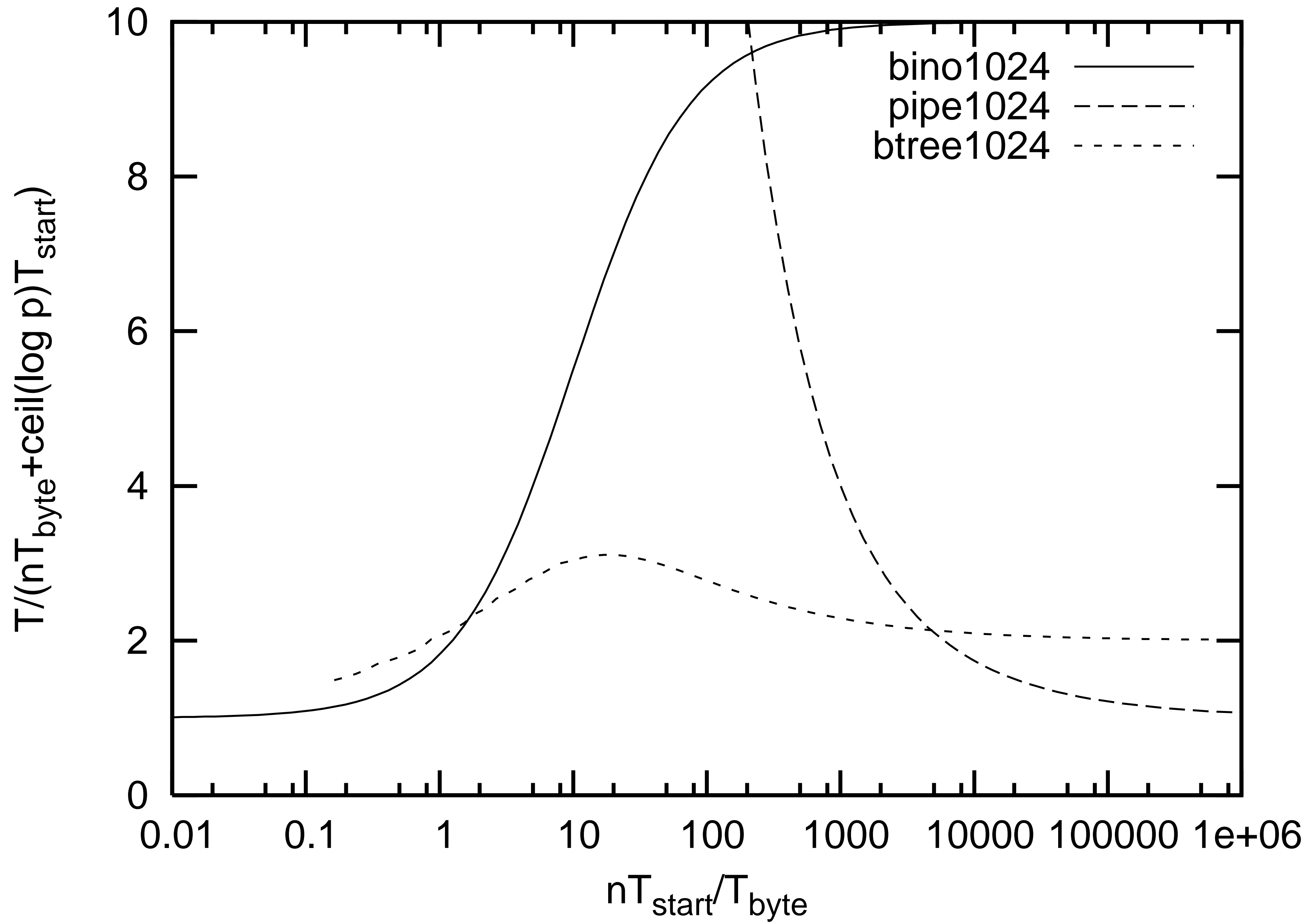


Analyse

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

insgesamt: $T^*(n,p) = dT_{\text{start}} + 2nT_{\text{byte}} + \mathcal{O}\left(\sqrt{ndT_{\text{start}}T_{\text{byte}}}\right)$

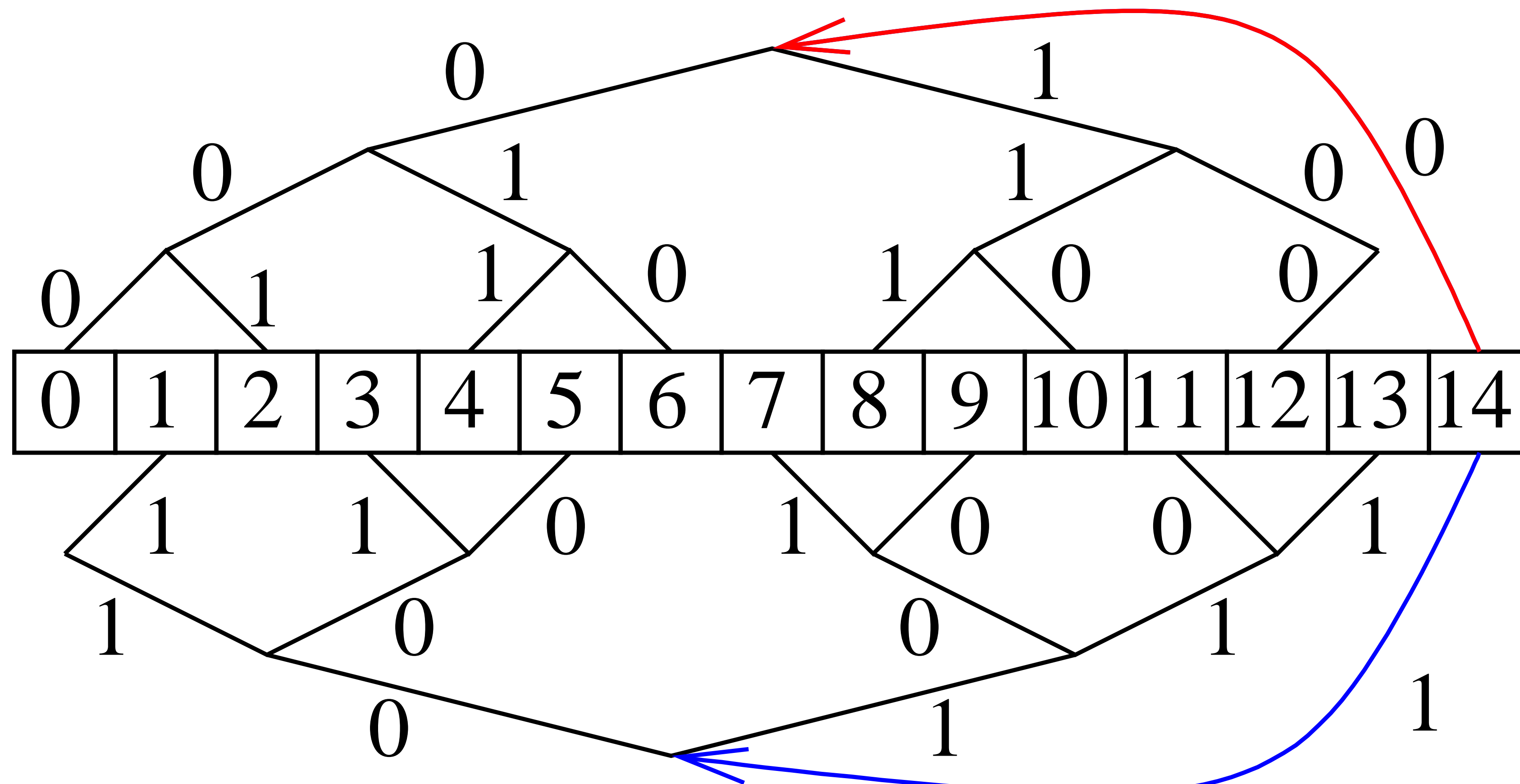

bino16
pipe16
btree16
0
1
2
3
4
5
0.01
0.1
1
10
100
1000
10000
T/(nT_byte+ceil(log p)T_start)
nT_start/T_byte


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

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

Idee: Spalte Nachricht in zwei Hälften.

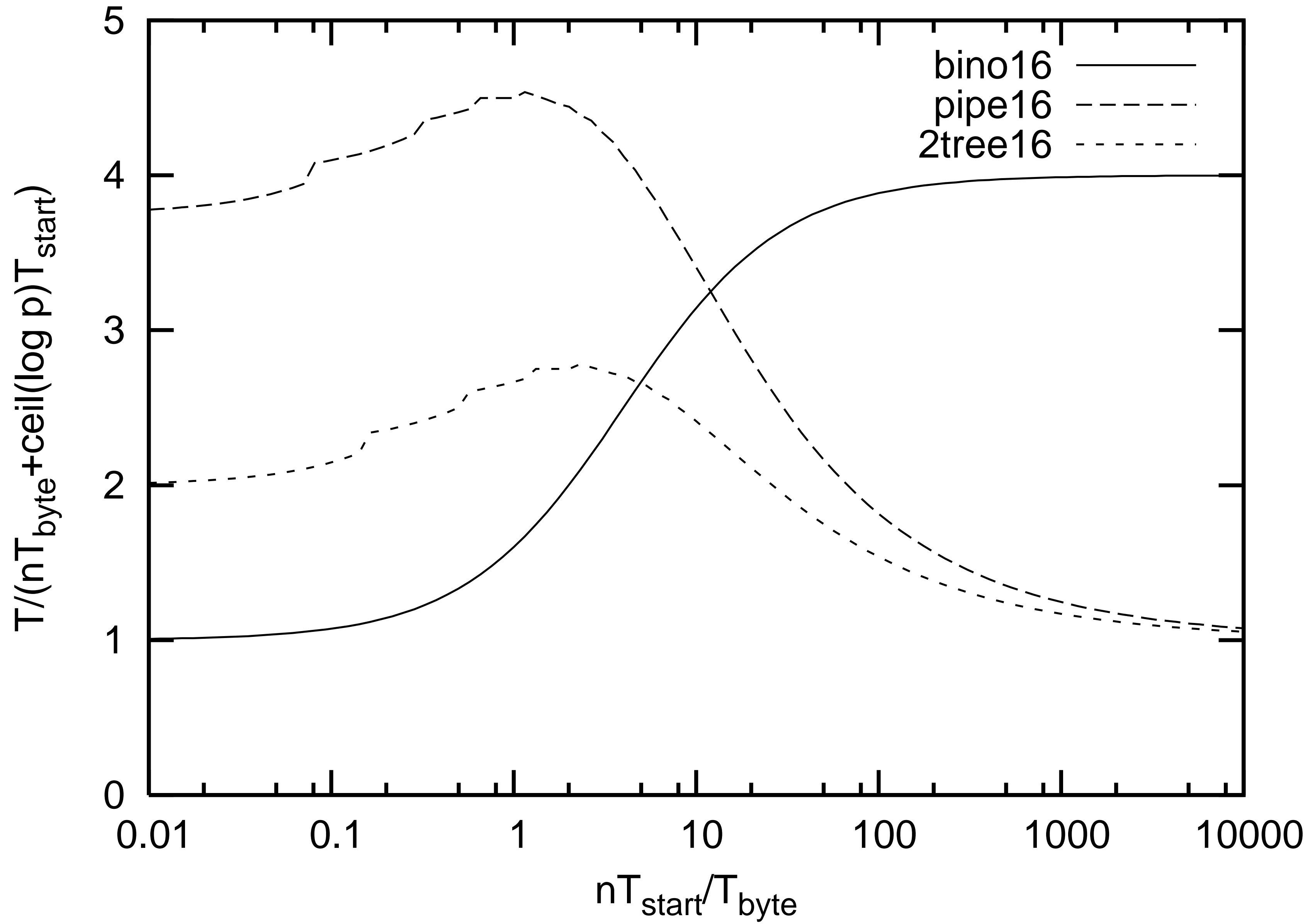
Zwei Binary-Tree-Broadcasts gleichzeitig.

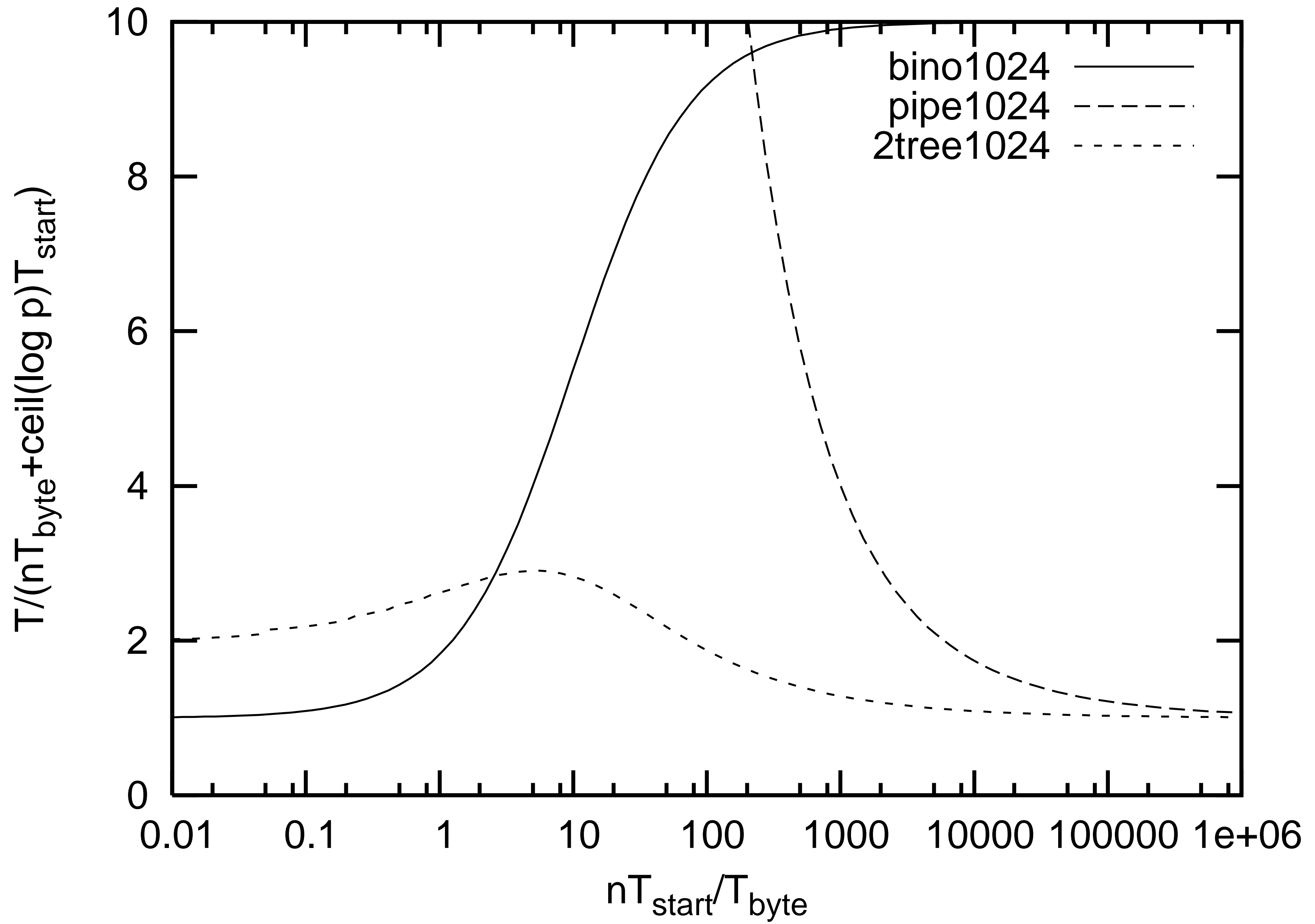


Analyse

$$\dots$$

$$T^*(n,p) :\approx nT_{\text{byte}} + T_{\text{start}} \cdot 2\log p + \sqrt{2n\log pT_{\text{start}}T_{\text{byte}}}$$


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


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

Jenseits Broadcast

- Pipelining ist wichtige Technik zu Umgang mit großen Datenmengen.
- Parametertuning (z.B. v. k) ist oft wichtig.

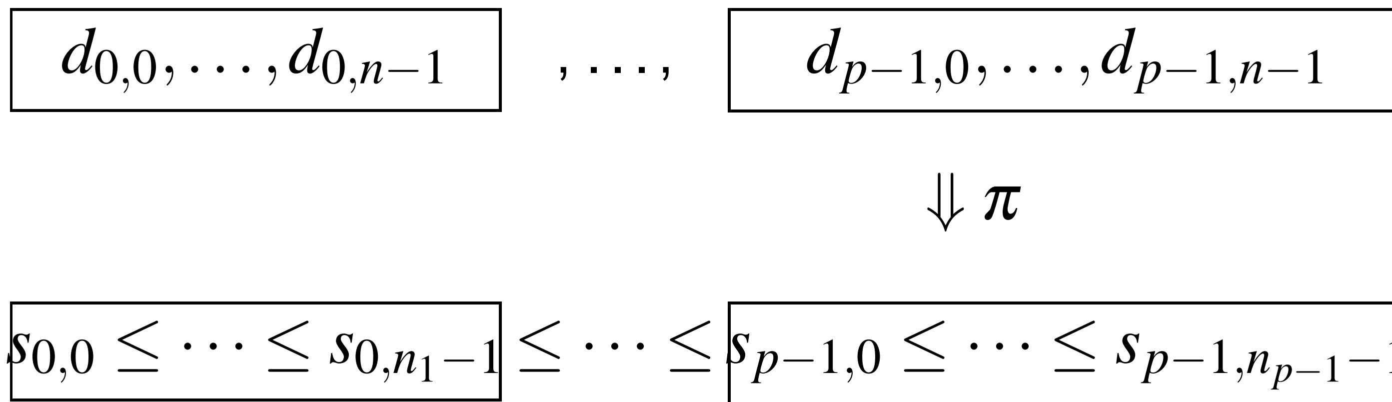
Sortieren

[Sanders Worsch Kapitel 6]

- Quicksort
- Sample Sort
- Multiway Mergesort
- Selection
- Mehr zu Sortieren

Sortieren größerer Datenmengen

- $m = np$ Eingabewerte. Anfangs n pro PE
- u.U. allgemeiner
- Ausgabe global sortiert



- Vergleichsbasiertes Modell
- $T_{\text{seq}} = T_{\text{compr}} m \log m + \mathcal{O}(m)$

Quicksort

Sequentiell

Procedure qSort($d[], p'$)

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

select a pivot v

reorder the elements in d such that

$$d_0 \leq \cdots \leq d_k = v \leq d_{k+1} \leq \cdots \leq d_{p'-1}$$

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

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

Anfänger-Parallelisierung

Parallelisierung der rekursiven Aufrufe.

$$T_{\text{par}} = \Omega\left(m\right)$$

- Sehr begrenzter Speedup
- Schlecht für distributed Memory

Theoretiker-Parallelisierung

Zur Vereinfachung: $m = p$.

Idee: Auch die Aufteilung parallelisieren.

1. Ein PE stellt den Pivot (z.B. zufällig).
2. Broadcast
3. Lokaler Vergleich
4. „Kleine“ Elemente durchnumerieren (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

$f := d \le v$

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

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

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

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

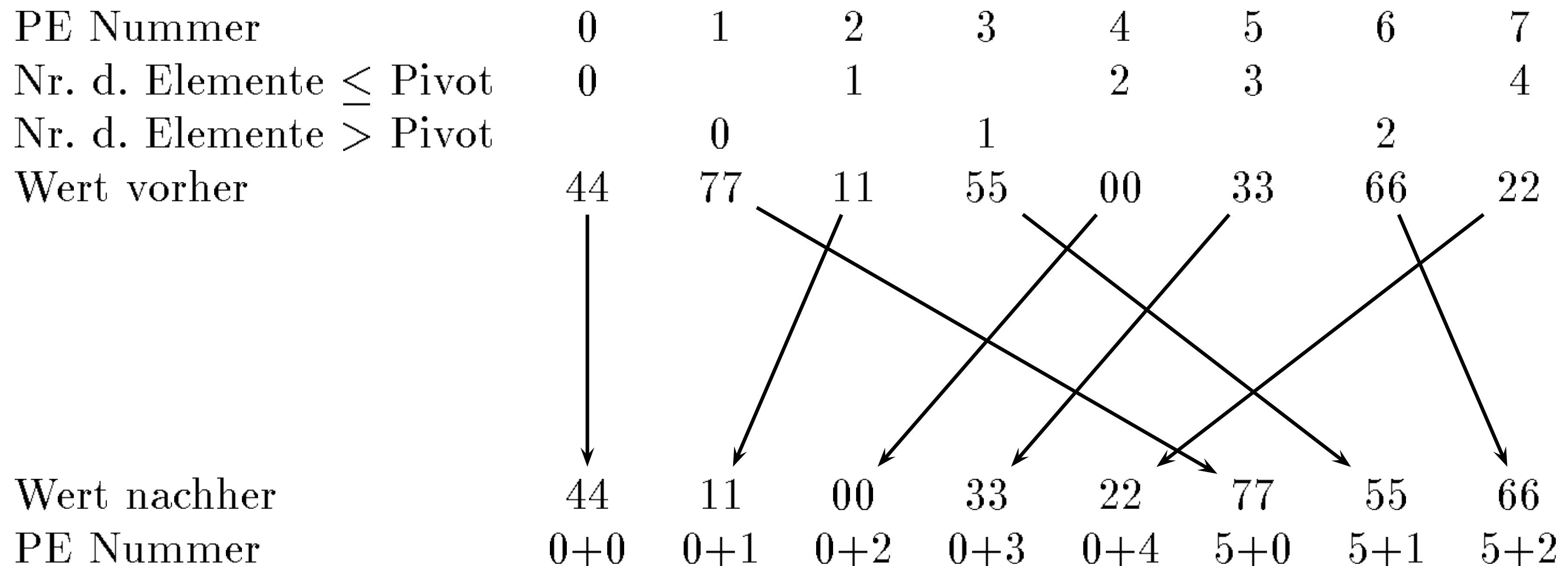
receive d

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

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

Beispiel

pivot $v = 44$



PE Nummer	0	1	2	3	4	5	6	7
Nr. d. Elemente $\leq$ Pivot	0		1		2	3		4
Nr. d. Elemente $>$ Pivot		0		1			2	
Wert vorher	44	77	11	55	00	33	66	22
Wert nachher	44	11	00	33	22	77	55	66
PE Nummer	0+0	0+1	0+2	0+3	0+4	5+0	5+1	5+2

```
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 ($\longrightarrow$später)

  $\rightsquigarrow$ Zeit $\mathcal{O}(T_{\text{start}} \log p)$
- erwartete Rekursionstiefe: $\mathcal{O}(\log p)$
  ($\longrightarrow$ Vorlesung randomisierte Algorithmen)

Erwartete Gesamtzeit: $\mathcal{O}\left(T_{\text{start}} \log^2 p\right)$

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 $\mathcal{O}\left(\frac{m \log m}{p} + \log^2 p\right)$ Algorithmus
- Bei verteiltem Speicher stört, dass jedes Element $\Omega(\log p)$ mal transportiert wird.

  $\rightsquigarrow \cdots \rightsquigarrow$ Zeit $\mathcal{O}\left(\frac{m}{p}(\log n + T_{\text{byte}} \log p) + T_{\text{start}} \log^2 p\right)$

hier nicht: inplace parallel quicksort

Multi-Pivot Verfahren

Vereinfachende Annahme: Splitter fallen vom Himmel

//Für $0 < k < p$ sei v_k das Element mit Rang $k \cdot m/p$

//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

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

$$\approx \frac{T_{\text{seq}}(np)}{p} + 2nT_{\text{byte}} + pT_{\text{start}}$$

Idealisierende Annahme ist realistisch für Permutation.

Sample Sort

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

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

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

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

unsortierte Eingangsdaten

19	7	12
1	9	13
25	4	2

6	30	17
13	10	11
16	27	22

3	20	14
18	5	16
15	21	8

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
1	12	25
4	13	19
2		
7		
9		

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

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

Umverteilung

lokal sortierte Daten

I_0		
1	2	3
4	5	6
7	8	9
10		

I_1		
11	12	13
13	14	15
16	16	18

I_2		
19	20	21
22	25	27
30		

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

Beweis: hier nicht. (Chernoff-Schranken)

Analyse von Sample Sort

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

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

Samples Sortieren

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

Mehrwegemischen

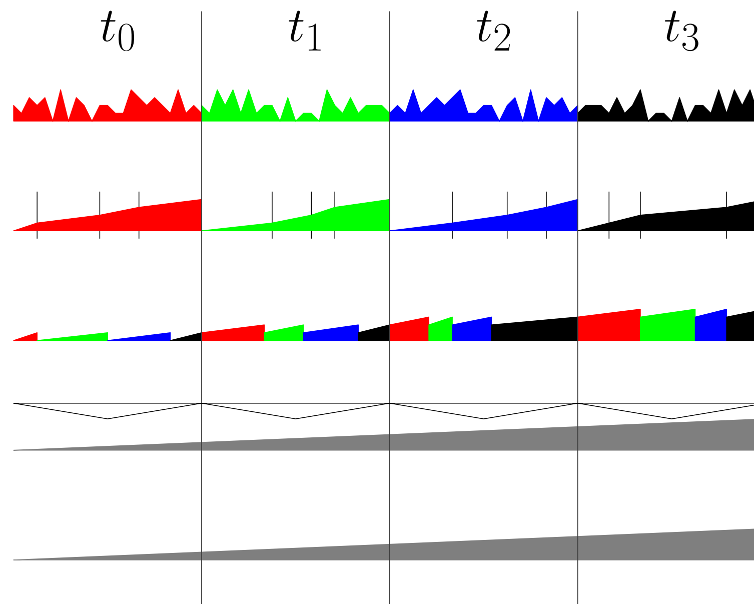
sort locally

v_k:= the element with rank $k \cdot m/p$ // multisequence selection

forall $k \in 1..P$ **do**

send $\langle d_j : v_k \leq d_j < v_{k+1} \rangle$ to PE k // All-to-all

merge received data // P-way merging



Multisequence Selection

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

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

Einfacher Algorithmus: $\mathcal{O}\left(p \log m \log \frac{m}{p}\right)$

...aber nicht hier

Mehr zu Sortieren

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

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

Kollektive Kommunikation

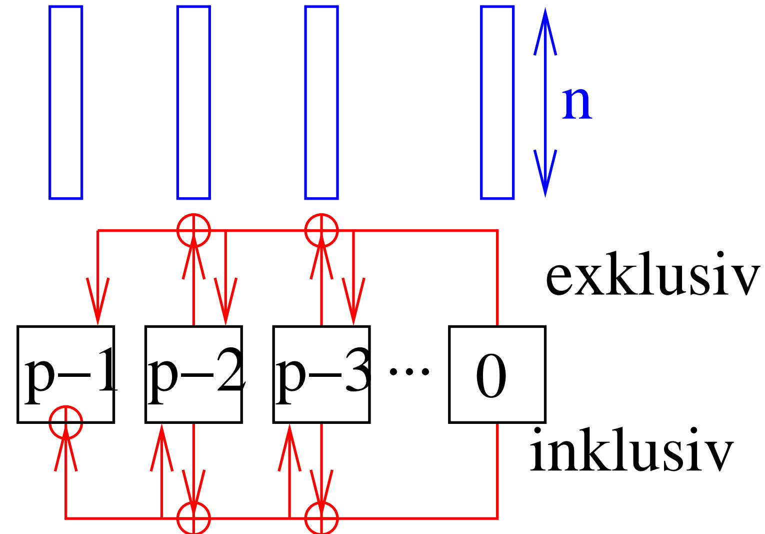
- Broadcast
- Reduktion
- Präfixsummen
- nicht hier: Sammeln / Austeilen (Gather / Scatter)
- Gossiping ($=$ All-Gather $=$ Gather $+$ Broadcast)
- All-to-all Personalized Communication
  - gleiche Nachrichtenlängen
  - ungleiche Nachrichtenlängen, $= h$-Relation

Präfixsummen

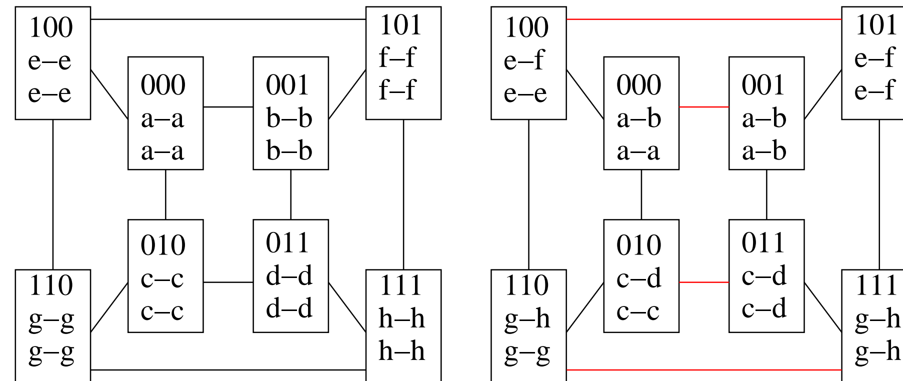
[Leighton 1.2.2] Gesucht

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

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



Hyperwürfelalgorithmus



// view PE index i as a

// d-bit bit array

Function hcPrefix(m)

$x := \sigma := m$

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

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

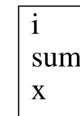
invariant $x = \bigotimes_{j=i[k..d-1]0^k}^{i} \text{m@j}$

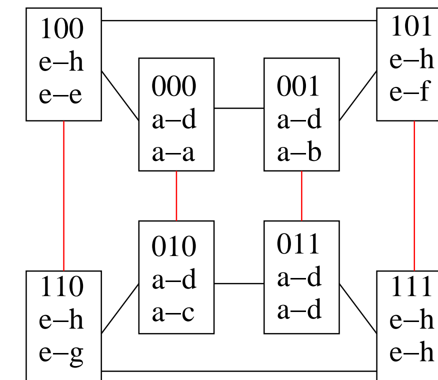
$y := \sigma @ (i \oplus 2^k)$ // sendRecv

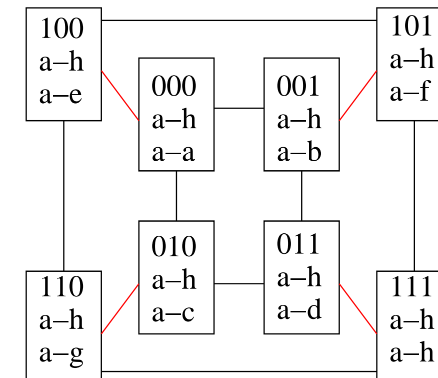
$\sigma := \sigma \otimes y$

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

return x







Analyse

$$T_{\text{prefix}} = (T_{\text{start}} + nT_{\text{byte}}) \log p$$

Pipelining klappt nicht, da alle PEs immer beschäftigt.

$\rightsquigarrow$

Baumbasierte Algorithmen, z.B. 23-tree:

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

Gossiping

Jedes PE hat eine Nachricht m der Länge n.

Am Ende soll jedes PE alle Nachrichten kennen.

Hyperwürfelalgorithmus

Sei '·' 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

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

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

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:

$$T_{\text{all}-\text{to}-\text{all}}(p,n) \approx \log p\left(\frac{p}{2} n T_{\text{byte}} + T_{\text{start}}\right)$$

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**

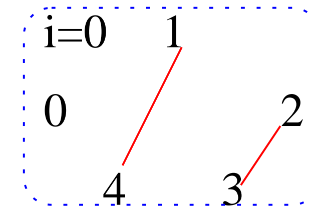
Exchange data with PE $(i - j) \bmod p$

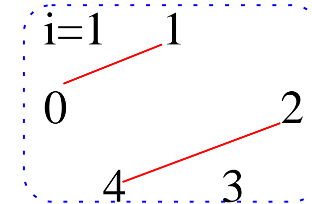
Paarweise Kommunikation:

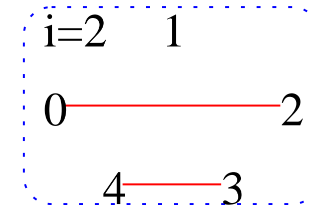
Der Partner des Partners von j in Runde i ist

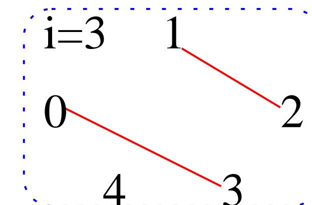
$i - (i - j) \equiv j \bmod p$

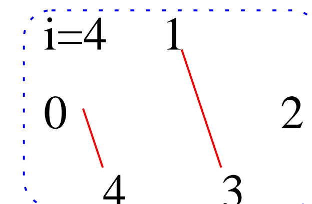
Zeit: $p(nT_{\text{byte}} + T_{\text{start}})$ optimal für $n \rightarrow \infty$











Der 1-Faktor-Algorithmus

p gerade:

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

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

idle:= $\frac{p}{2} i \bmod (p-1)$

if $j = p-1$ **then** exchange data with PE idle

else

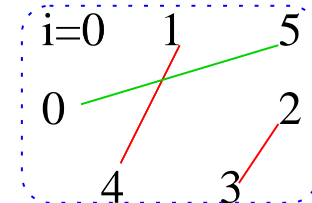
if $j =$ idle **then**

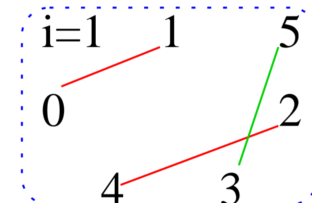
exchange data with PE $p-1$

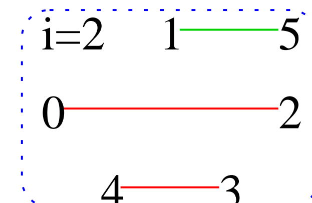
else

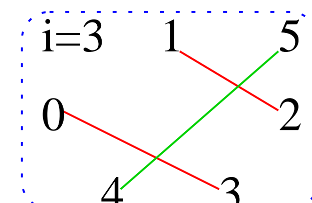
exchange data with PE $(i-j) \bmod (p-1)$

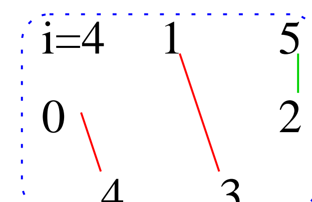
Zeit: $p(nT_{\text{byte}} + T_{\text{start}})$ optimal für $n \to \infty$











Datenaustausch bei unregelmäßigen Nachrichtenlängen

- Vor allem bei all-to-all interessant $\longrightarrow$ 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

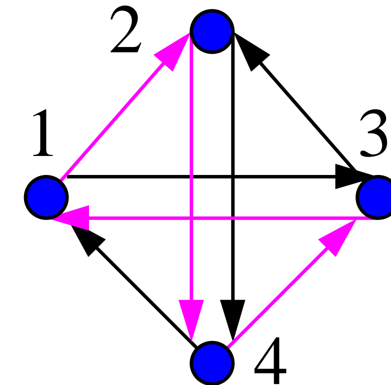
$h := \max_{i=1}^{p} h_{\text{in}}(i) + h_{\text{out}}(i)$ $h := \max_{i=1}^{p} \max(h_{\text{in}}(i), h_{\text{out}}(i))$

Untere Schranke bei paketweiser Auslieferung:

h Schritte, d.h.,

Zeit $h(T_{\text{start}} + |\text{Paket}| T_{\text{byte}})$

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.,

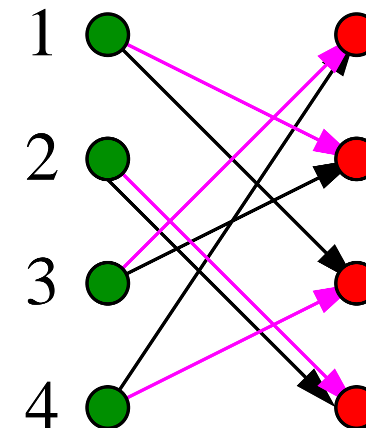
keine zwei gleichfarbigen Kanten

inzident zu einem Knoten.

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

Sende Nachrichten der Farbe j

optimal wenn man paketweise Auslieferung postuliert



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** $c[i] := \langle\rangle$

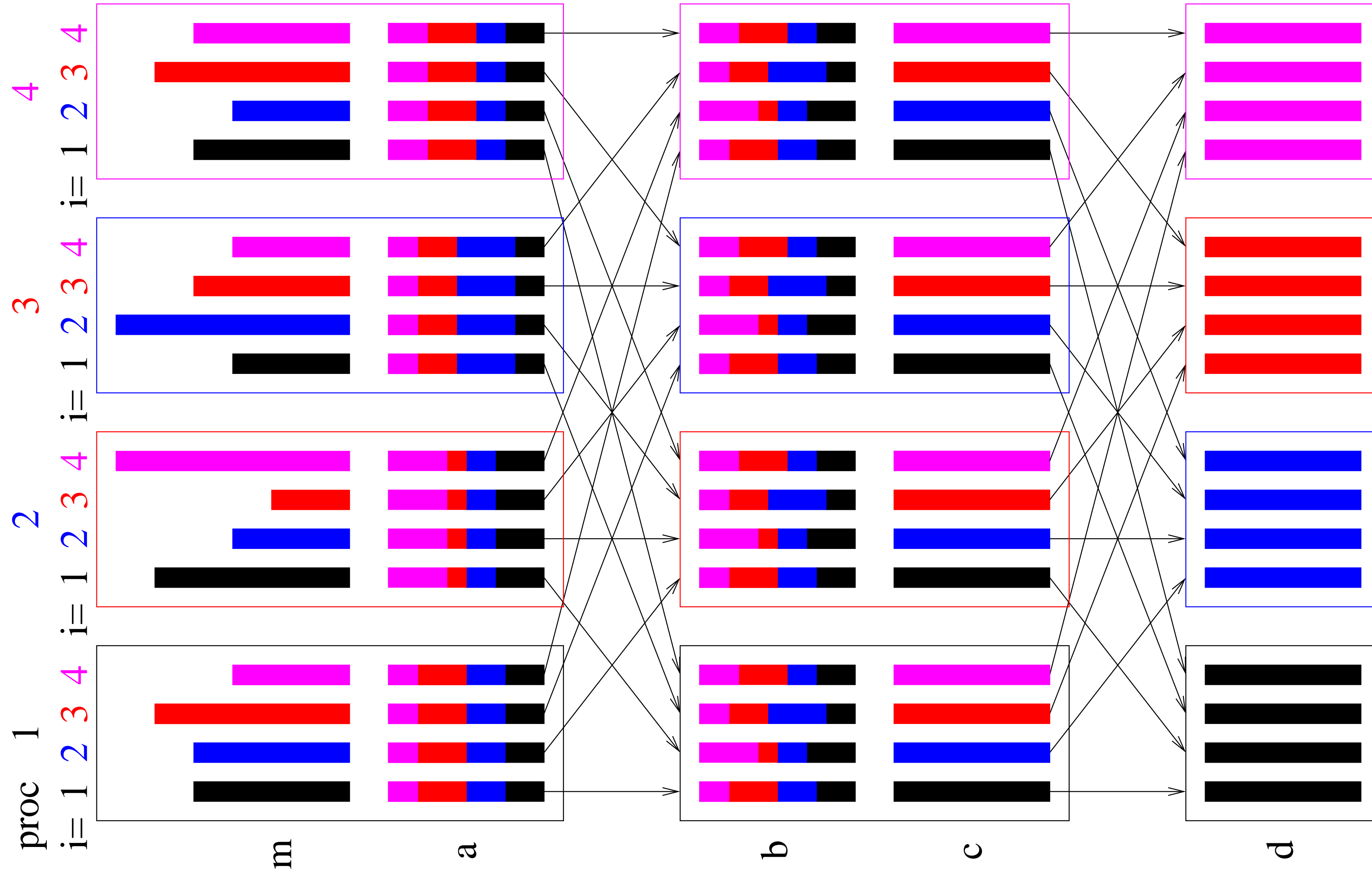
for $j := 1$ **to** p **do**

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

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.

Färbungsbasierte Algorithmen: Fast optimal bei großen Paketen.

Komplex. Verteilte Implementierung? Aufspalten in Pakete stört.

Vergleich von Ansätzen?

List Ranking

[JáJá Section 3.1]

Motivation:

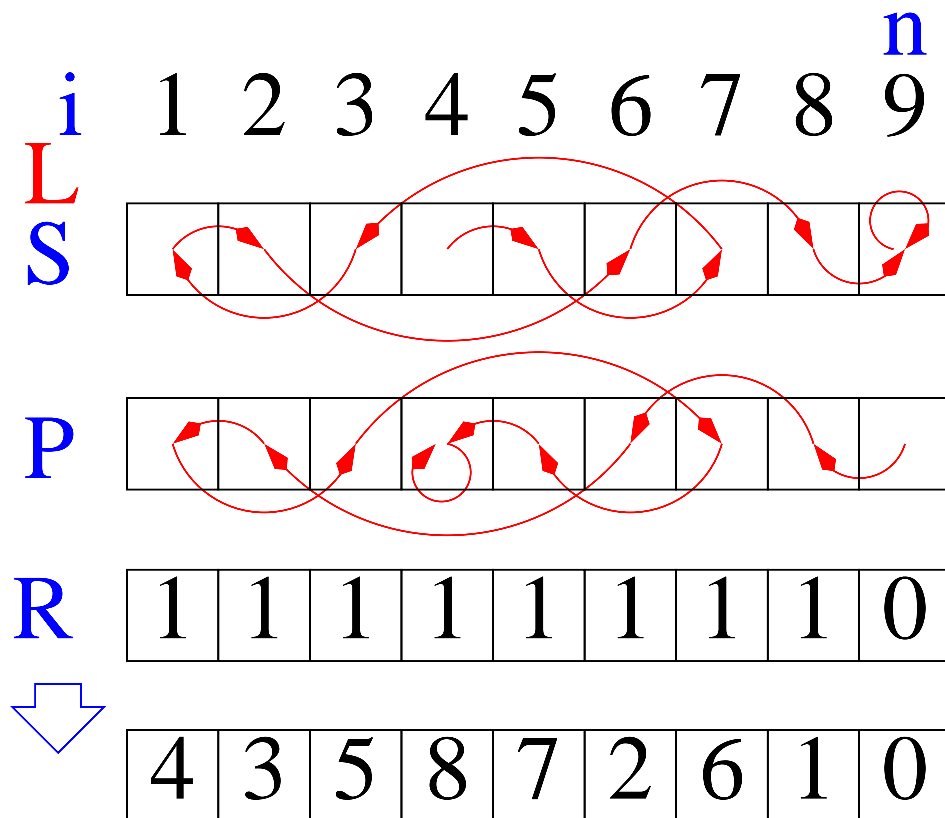
mit Arrays $a[1..n]$ können wir viele Dinge parallel machen

- 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

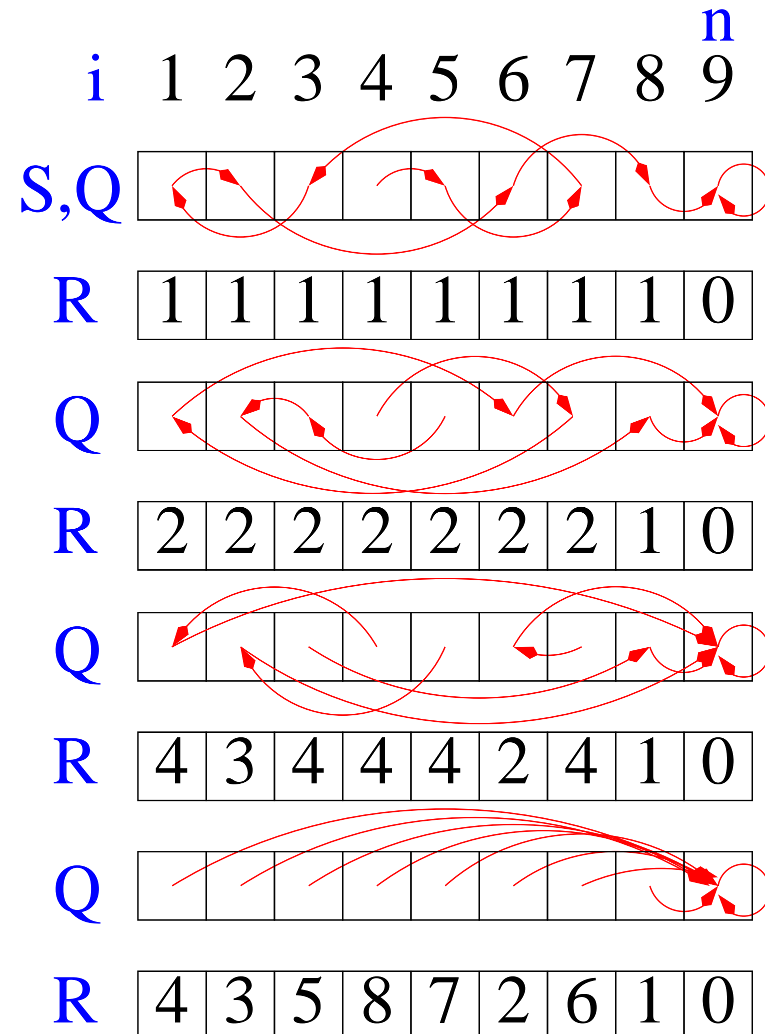
Q_i is the positions given by

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** $f(P(i))$

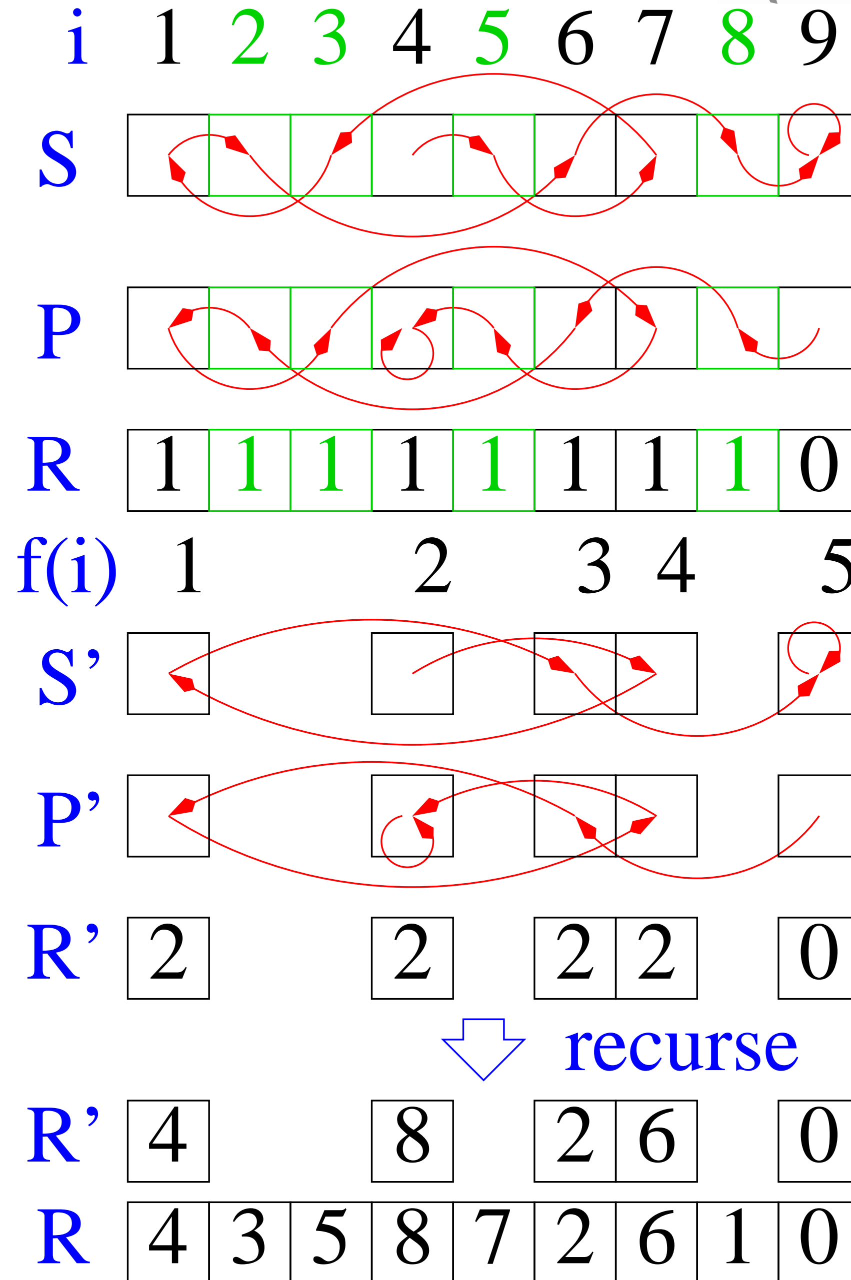
$R'(f(i)) :=$ **if** $S(i) \in I$ **then** $R(i) + R(S(i))$ **else** $R(i)$

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

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

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

Fakultät für Informatik

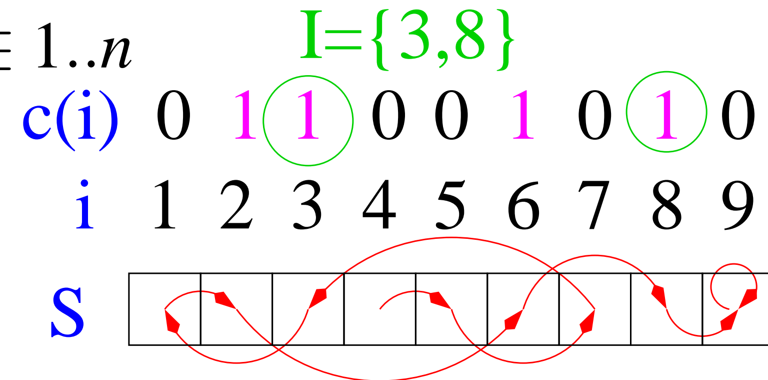

I={2,3,5,8}
n
i 1 2 3 4 5 6 7 8 9
S
P
R 1 1 1 1 1 1 1 1 0
f(i) 1 2 3 4 5
S'
P'
R' 2 2 2 2 0
recurse
R' 4 8 2 6 0
R 4 3 5 8 7 2 6 1 0

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 Carlso 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.

Übung: $|I| \geq 0.4n$ in Zeit $\mathcal{O}\left(\frac{n}{p} + \log n\right)$?

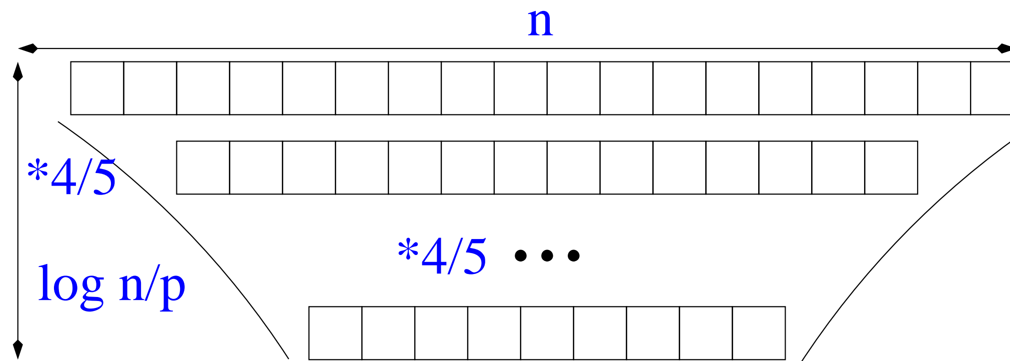
Finden einer bijektiven Abbildung

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

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

[teilweise in JáJá Abschnitt 5.2]

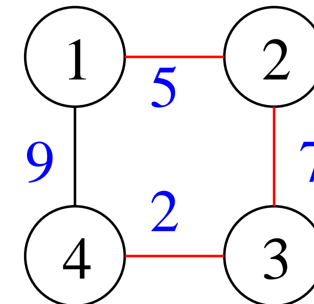
undirected Graph $G = (V, E)$.

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.

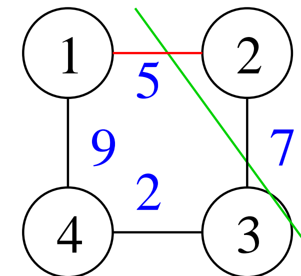
Selecting and Discarding MST Edges

The Cut Property

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

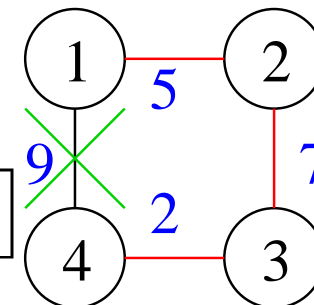
$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$

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

$T := \emptyset$

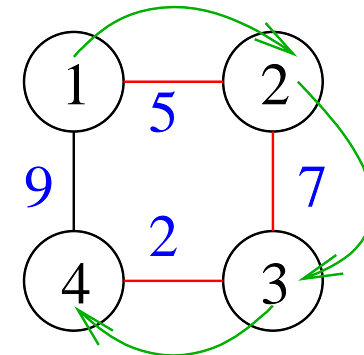
$S := \{s\}$ for arbitrary start node s

repeat $n-1$ times

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

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

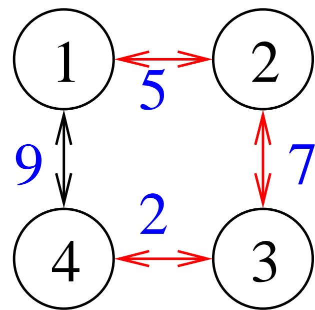
$T := T \cup \{(u,v)\}$



Graph Representation for Jarník-Prim

Adjacency Array

We need node $\longrightarrow$ incident edges



	1			n	5=n+1
V	1	3	5	7	9

	1					m		8=m+1
E	2	4	1	3	2	4	1	3
c	5	9	5	7	7	2	2	9

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 := ∅                                              // subforest of the MST
foreach (u,v) ∈ E in ascending order of weight do
    if u and v are in different subtrees of T then
        T := T ∪ {(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

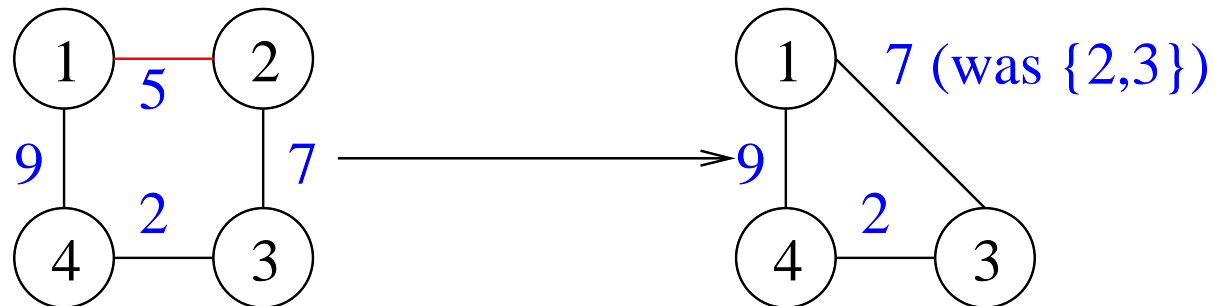
Edge Contraction

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]

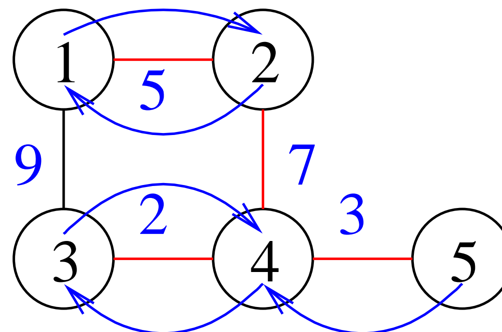
For each node find the lightest incident edge.

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 $\mathcal{O}(\mathrm{Polylog}(m))$ time parallelization

Finding lightest incident edges

Assume the input is given in adjacency array representation

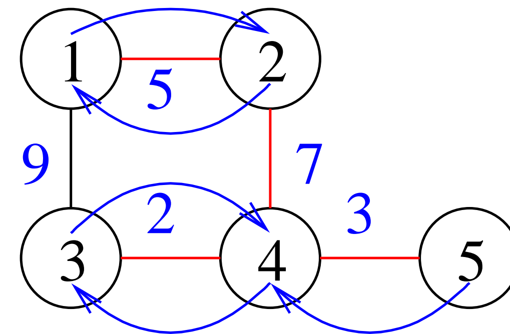
forall $v \in V$ **dopar**

allocate degree$(v)\frac{p}{n}$ 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)

pred$(v) := w$



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

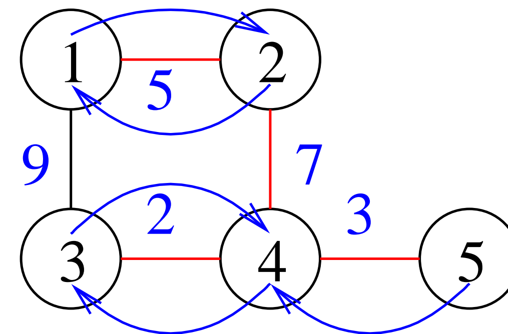
Structure of Resulting Components

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

- 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



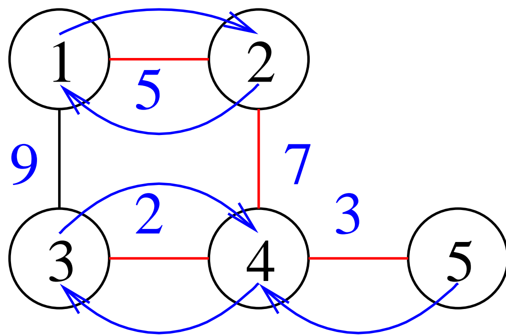
Pseudotrees → Rooted Trees

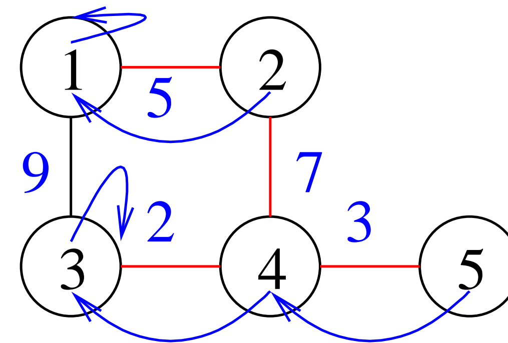
forall $v \in V$ **dopar**

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

if $v < w \land \mathsf{pred}(w) = v$ **then** $\mathsf{pred}(v) := v$

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



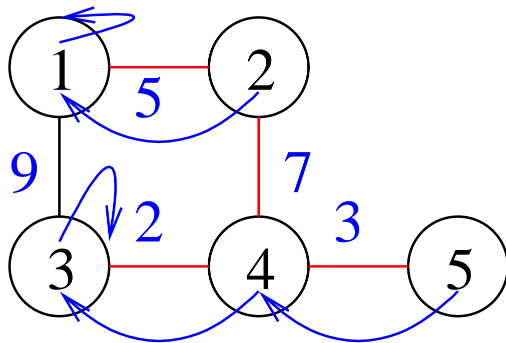


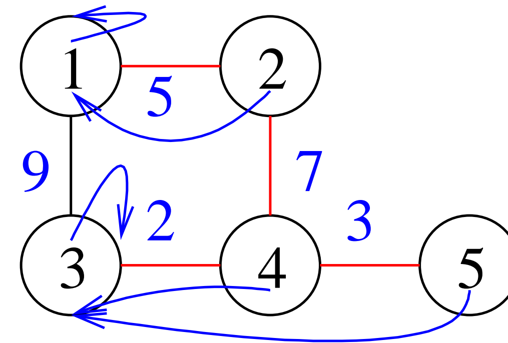
Rooted Trees → Rooted Stars by Doubling

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

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

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

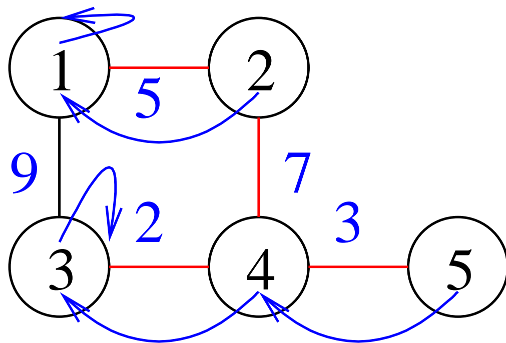


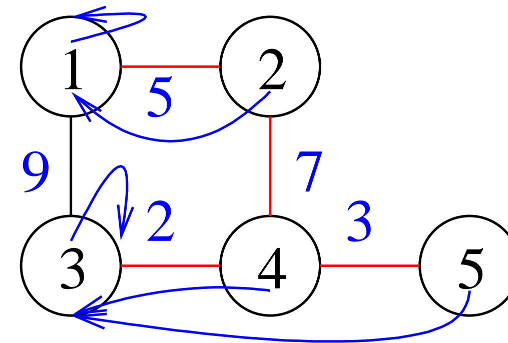


Efficient: Rooted Trees $\longrightarrow$ Rooted Stars

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

Algorithm: not here. Similar ideas as in list ranking





Contraction

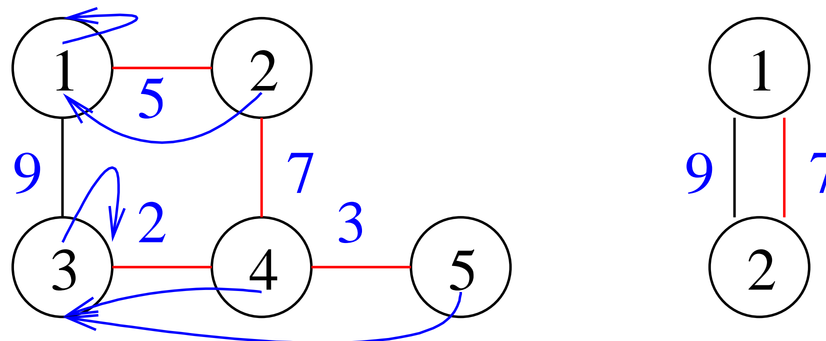
k:= #components

$V' = 1..k$

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

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

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

expected time bound $T(m,n) = \mathcal{O}\left(\frac{m}{p} + \log n\right) + T(m, \frac{n}{2})$, i.e.,

$T(m,n) = \mathcal{O}\left(\log n(\frac{m}{p} + \log n)\right)$

relativ effizient für $m = \Omega\left(p \log^2 p\right)$

absolut effizient falls außerdem $m = \mathcal{O}(n)$

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

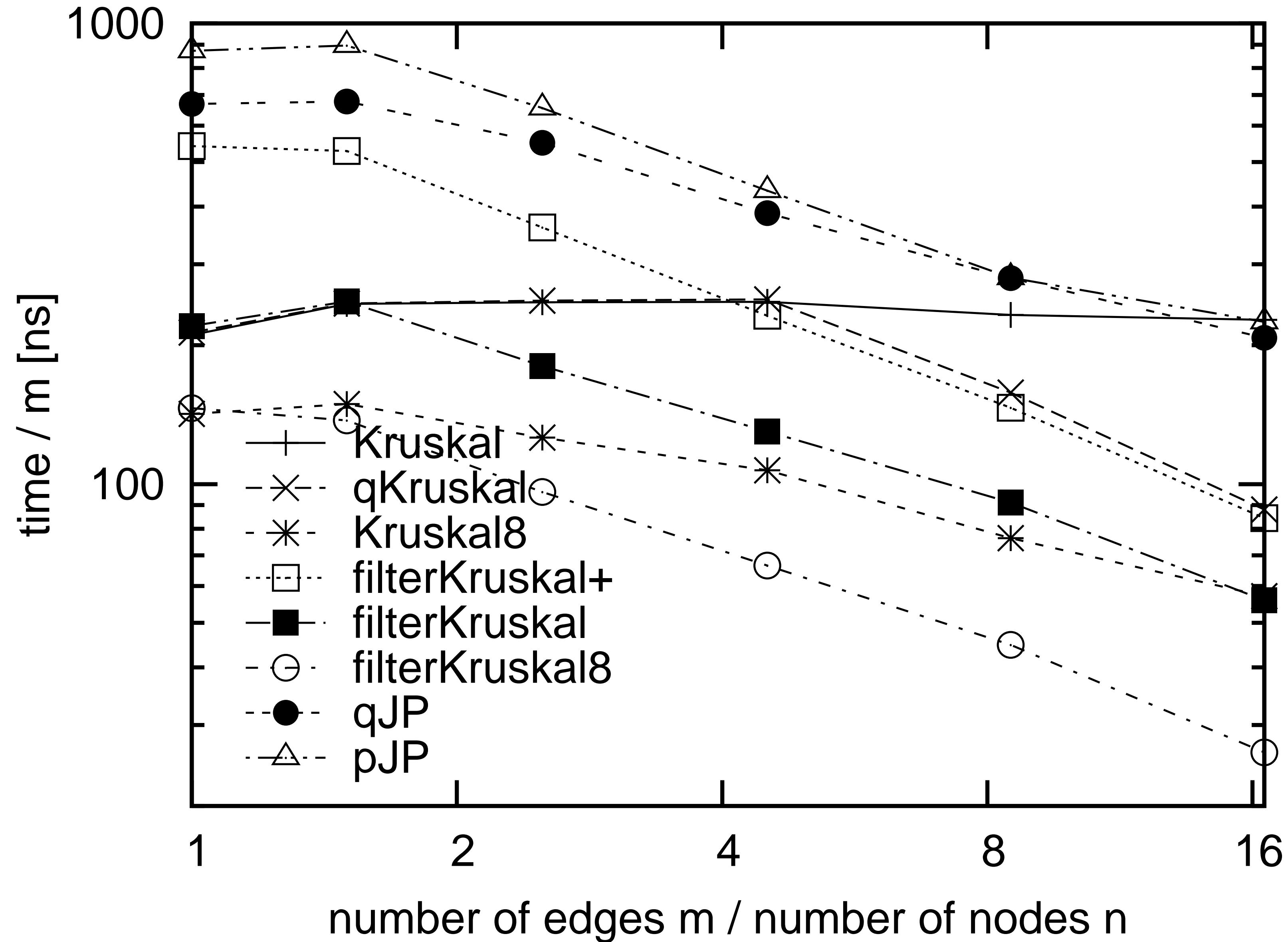
    qKruskal($E_{\leq}, T, P$)

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

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

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

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.

[Bader Cong 04] report speedup ≈ 4 on sparse graphs and a shared memory machine

Lastverteilung

[Sanders Worsch 97]

Gegeben

- zu verrichtende Arbeit
- PEs

Lastverteilung = Zuordnung Arbeit $\longrightarrow$ 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, 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
- Graphpartitionierung (falls Zeit)

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

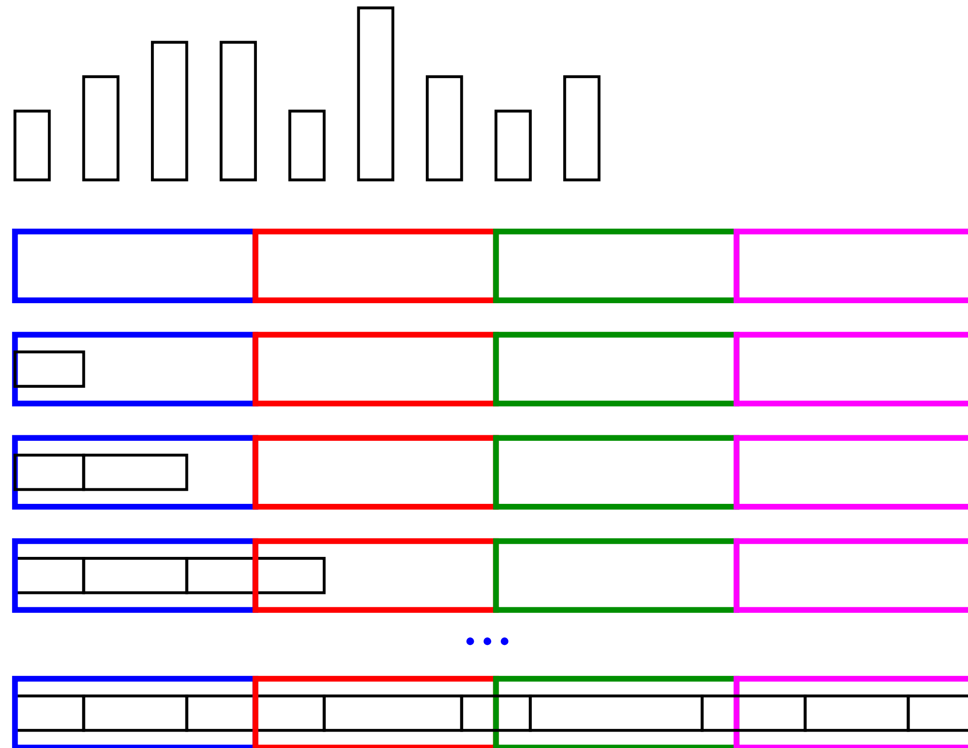
  $f := f - c$

  $\ell := \ell - c$

  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 $j_i..j'_i$

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

forall $j \leq n$ **dopar**

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

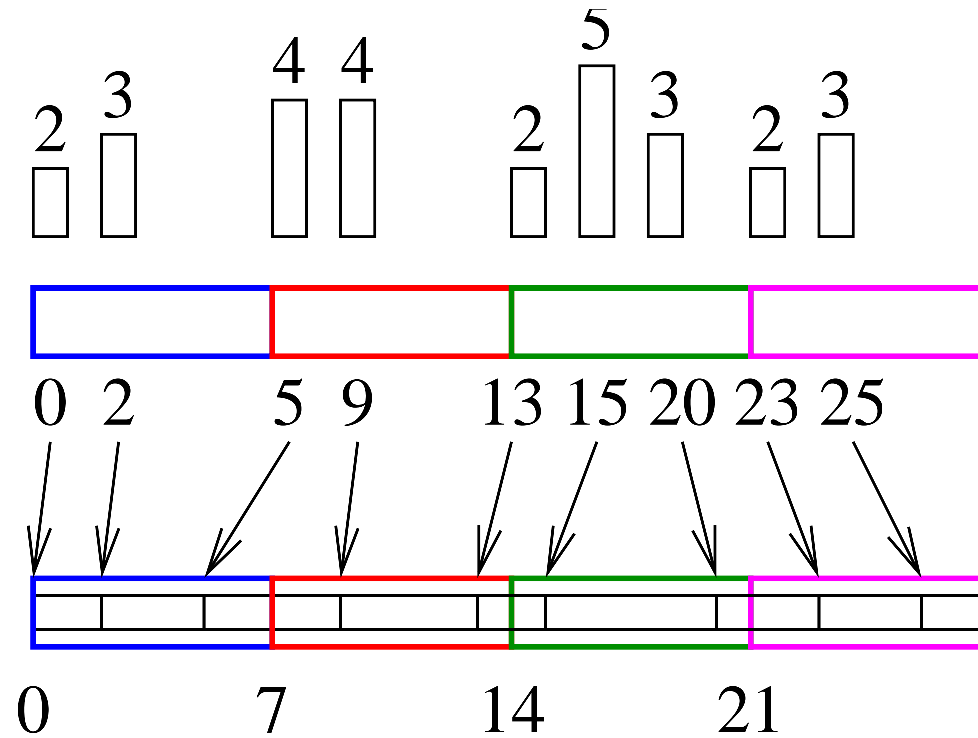
assign job j to PEs $\left\lfloor\frac{\text{pos}}{C}\right\rfloor..\left\lfloor\frac{\text{pos}+\ell_j}{C}\right\rfloor$ // segmented broadcast

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

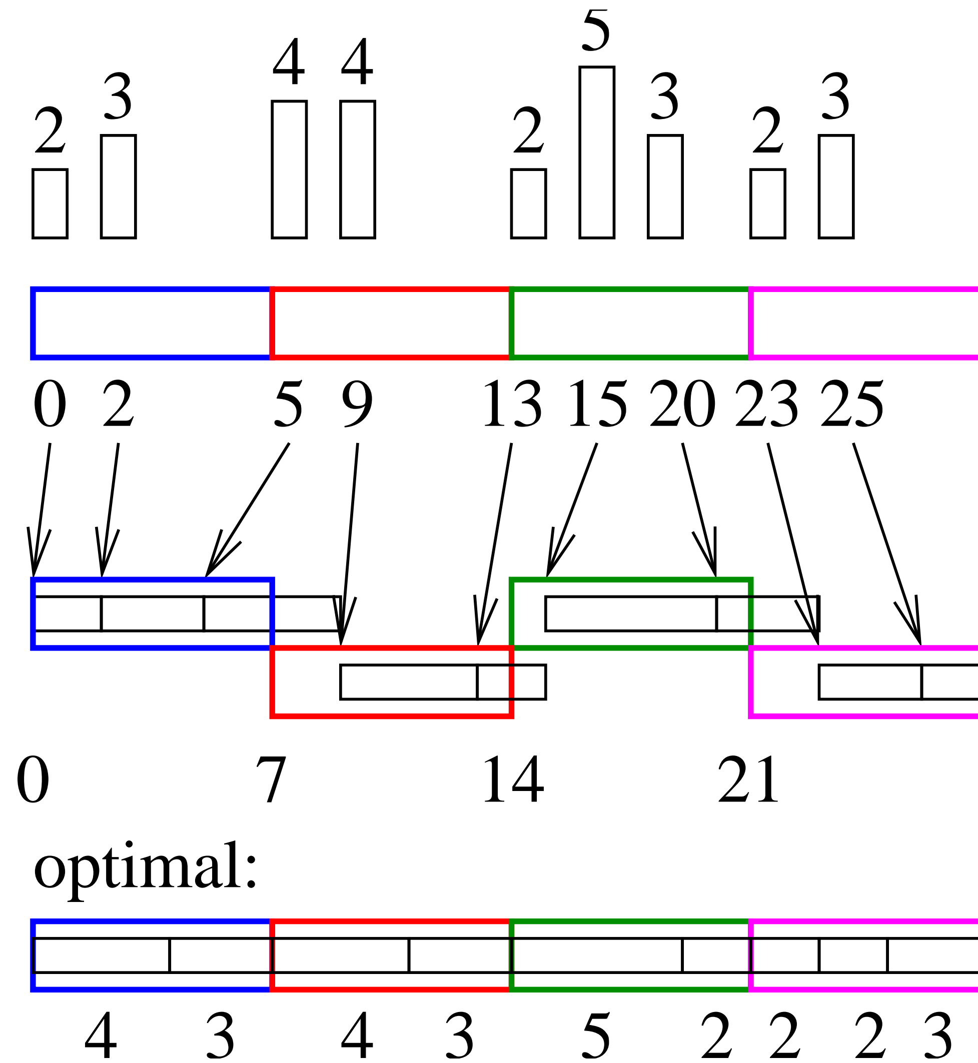
vermutlich nicht parallelisierbar

Parallel

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

[Anderson, Mayr, Warmuth 89]

Atomare Jobs: Beispiel

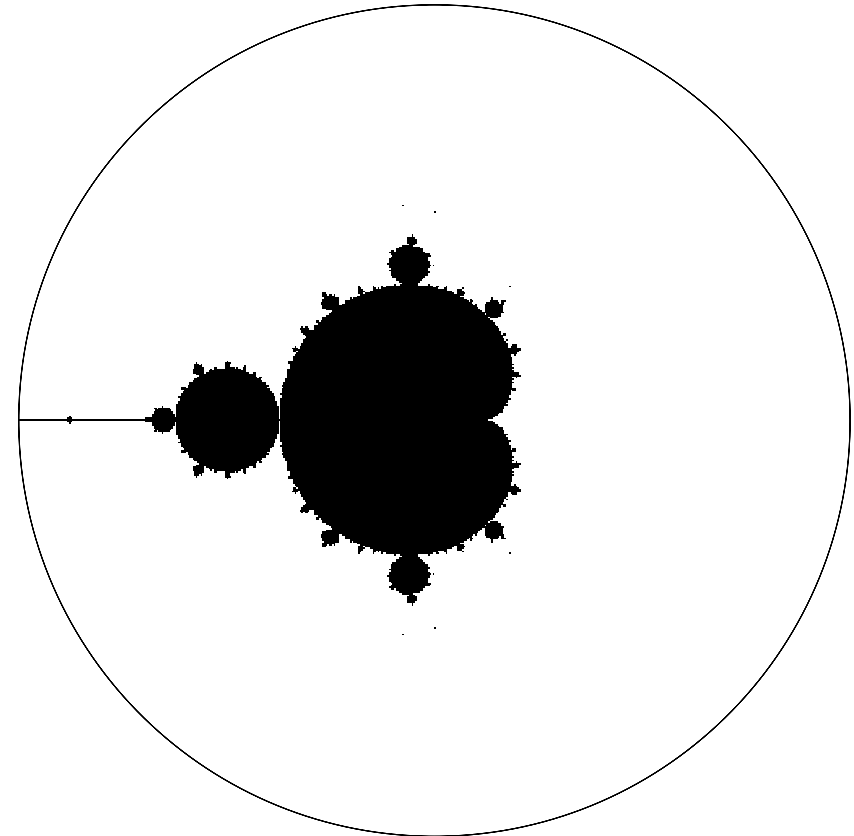


Beispiel Mandelbrotmenge

$z_c(m) : \mathbb{N} \to \mathbb{C}$

$z_c(0) := 0, \quad z_c(m+1) := z_c(m)^2 + c$

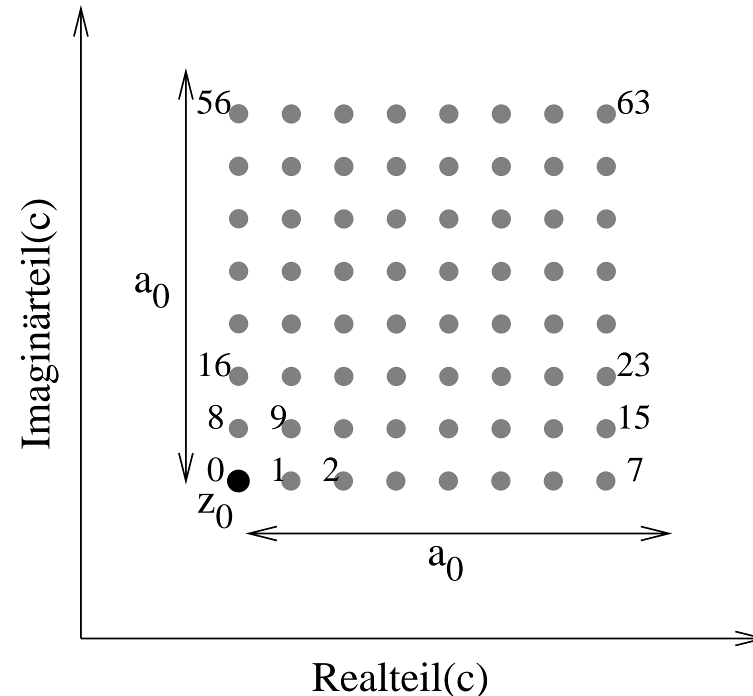
$M := \{c \in \mathbb{C} : z_c(m) \text{ ist beschränkt}\}$.



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?



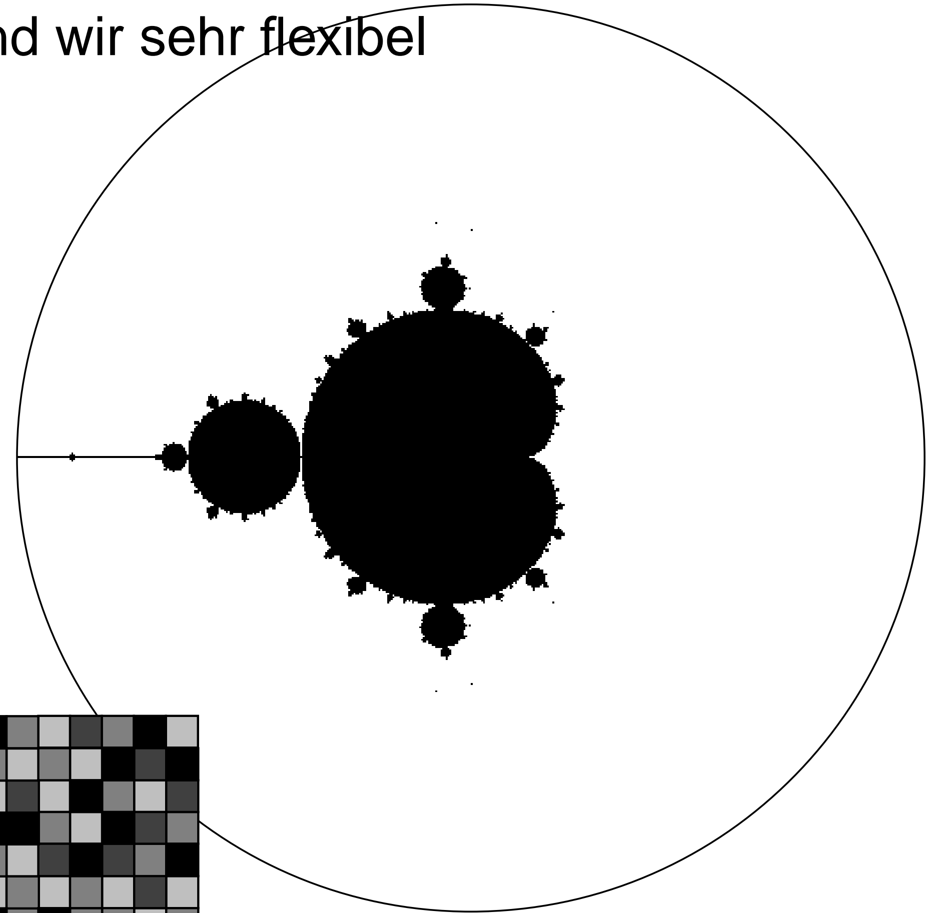
Code

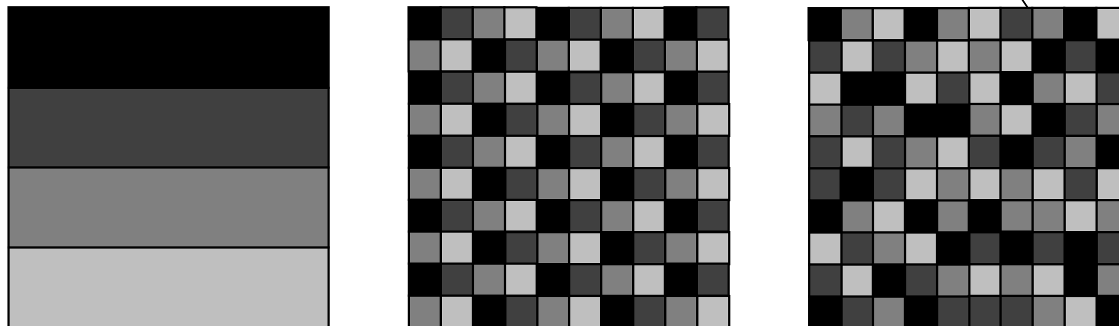
```
int iterate(int pos, int resolution, double step)
{ int iter;
  complex c =
    z0+complex((double)(pos % resolution) * step,
               (double)(pos / resolution) * step);
  complex z = c;
  for (iter = 1;
       iter < maxiter && abs(z) <= LARGE;
       iter++) {
    z = z*z + c;
  }
  return 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 $\pi : 0..n-1 \rightarrow 0..n-1$

Wlog (?) let n be a square.

- Interpret numbers from $0..n-1$ as pairs from $\{0..\sqrt{n}-1\}^2$.
- $f : 0..\sqrt{n}-1 \rightarrow 0..\sqrt{n}-1$ (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 $L \geq 2(\beta+1) p l_{\max} \frac{\ln p}{\varepsilon^2} + O(\varepsilon^3))$

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

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

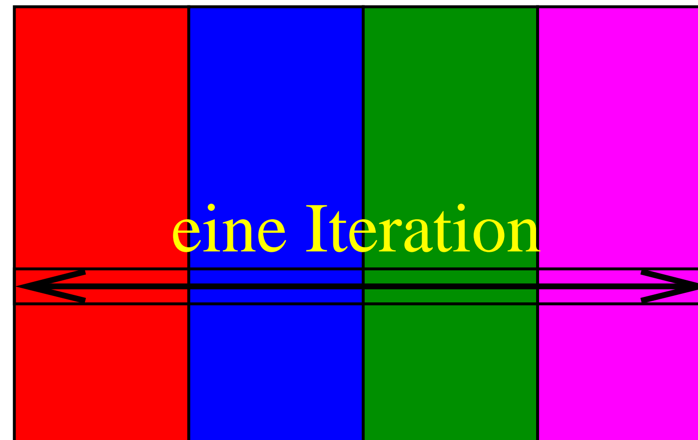
…Chernoff-Schranken…

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 $1/\varepsilon$).

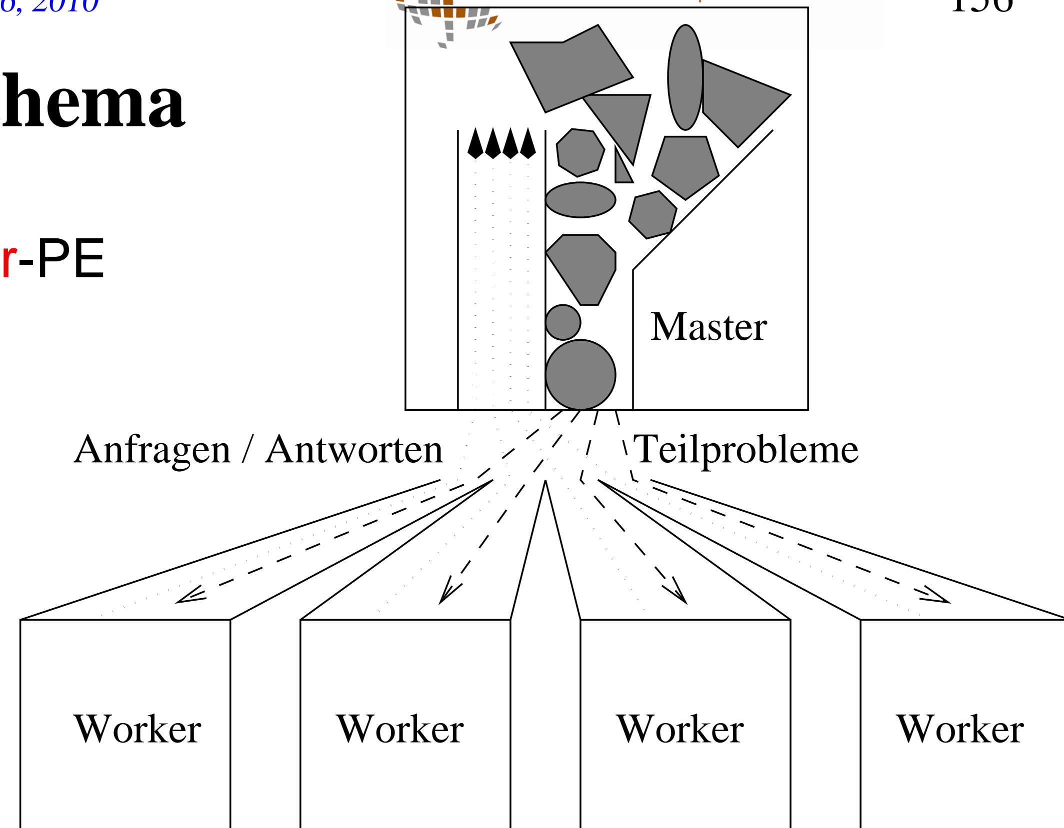
Anwendungsbeispiel: Airline Crew Scheduling

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)



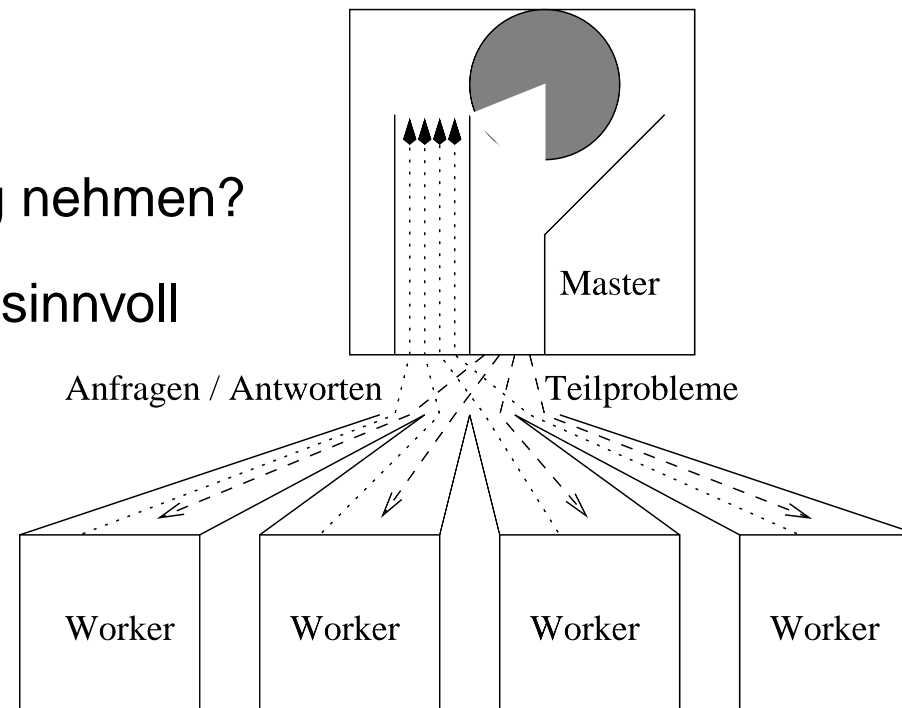
Diskussion

+ 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

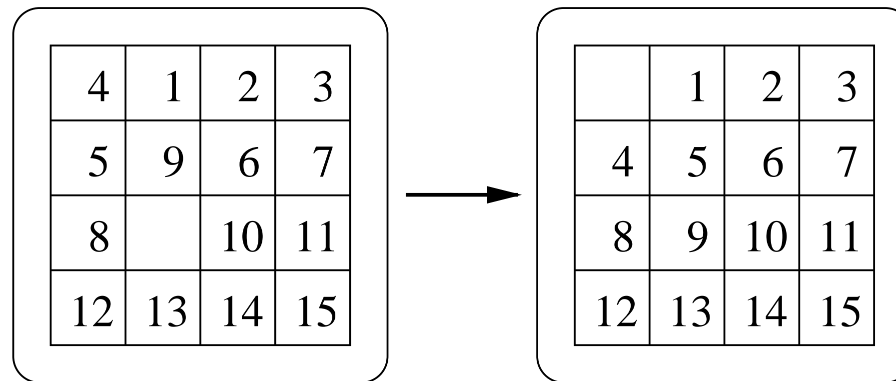
- Möglichst grosse Probleme abgeben solange Lastverteilung nicht gefährdet. Warum?
- Konservatives Kriterium: obere Schranke für die Größe des abgegebenen Teilproblems $\leq$ $1/P$-tel untere Schranke für Systemlast.
  - Woher Grössenabschätzung nehmen?
- Aggressivere Verfahren ggf. sinnvoll



Work Stealing

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

Example: The 15-Puzzle



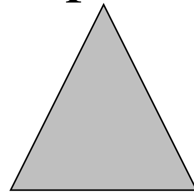
Korf 85: Iterative deepening depth first search with $\approx 10^9$ tree nodes.

Goal for the analysis

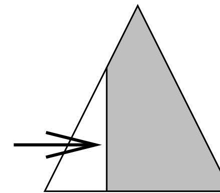
$$T_{\text{par}} \leq (1+\varepsilon)\frac{T_{\text{seq}}}{p} + \text{lower order terms}$$

An Abstract Model: Tree Shaped Computations

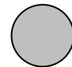
subproblem



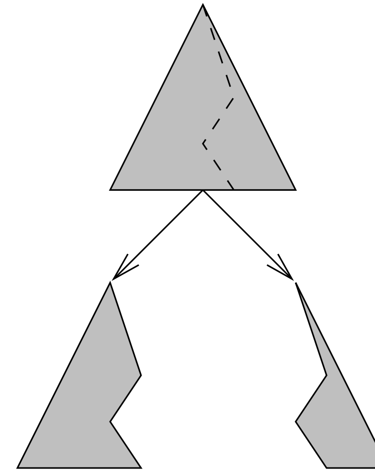
work sequentially



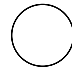
atomic



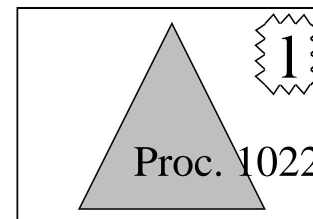
split



empty



send



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 ($T_{\text{start}} + \ell T_{\text{byte}}$)

T_{coll}: time for a reduction

Other Problems Categories

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

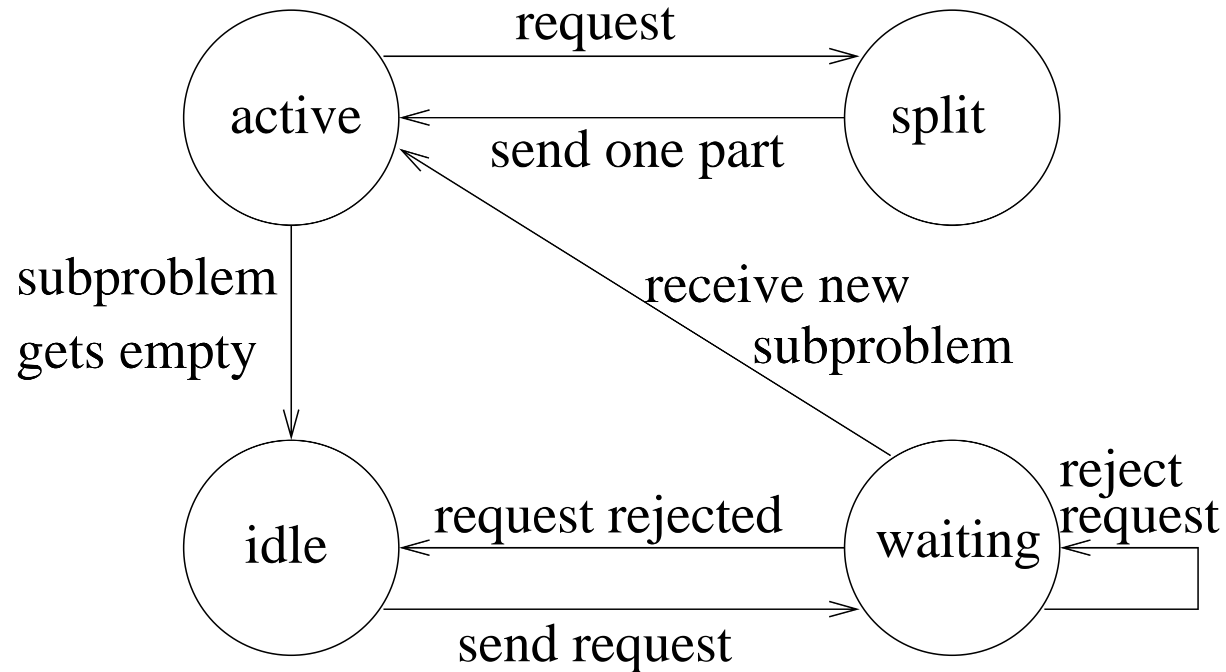
An Application List

- 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

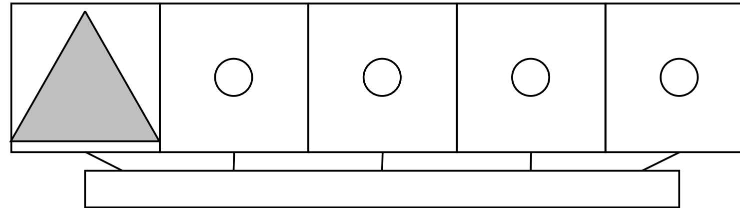
Limits of the Model

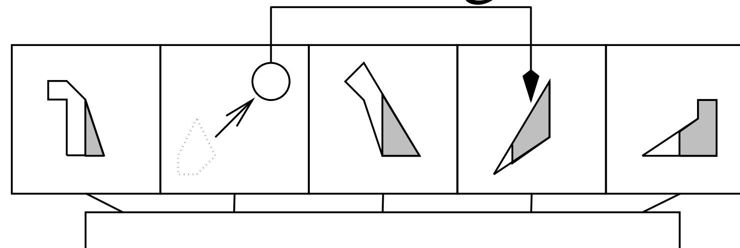
- Quicksort and similar divide-and-conquer algorithms (shared memory OK $\rightsquigarrow$ Cilk, MCSTL, Intel TBB)
- Finding the first Solution (often OK)
- Branch-and-bound
  - Verifying bounds OK
  - Depth-first often OK
- Subtree dependent pruning
  - FSSP OK
  - Game tree search tough (load balancing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

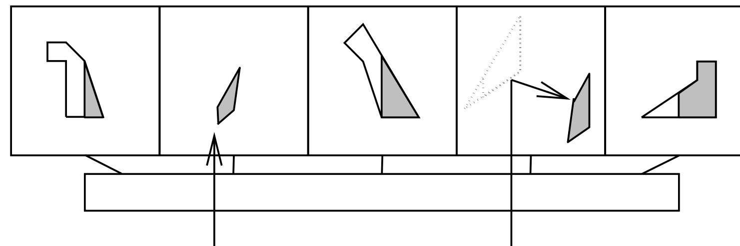
Receiver Initiated Load Balancing



Random Polling







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 3.

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

for an appropriate choice of Δt.

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