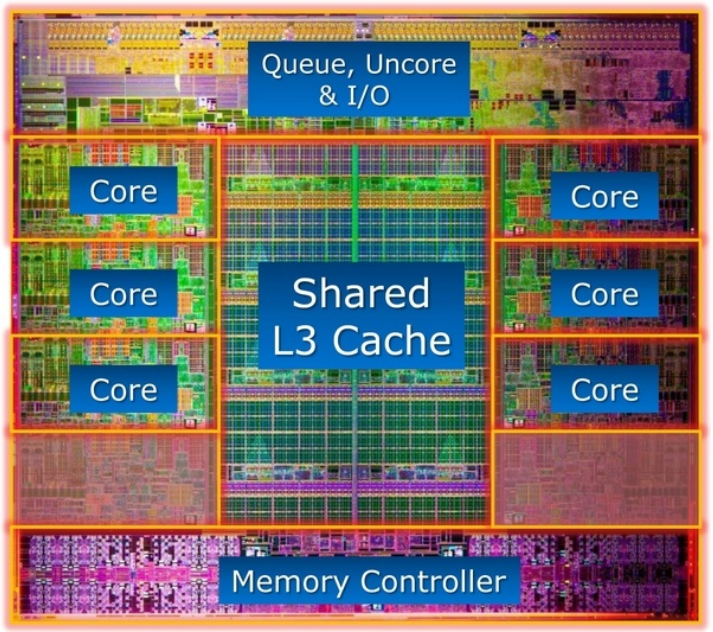
Parallel Algorithms

Peter Sanders

Institute of Theoretical Informatics



Why Parallel Processing

increase speed: p computers jointly working on a problem solve it up to p times faster. But, too many cooks spoil the broth. $\rightsquigarrow$ careful coordination of processors

save energy: Two processors with half the clock frequency need less power than one processor at full speed. (power$\approx$ voltage $\cdot$ clock frequency)

expand available memory by using aggregate memory of many processors

less communication: when data is distributed it can also be (pre)processed in a distributed way.

Subject of the Lecture

Basic methods for parallel problem solving

- Parallelization of basic sequential techniques: sorting, data structures, graph algorithms, ...
- Basic communication patterns
- load balancing
- Emphasis on provable performance guarantees
- But applicability always in view

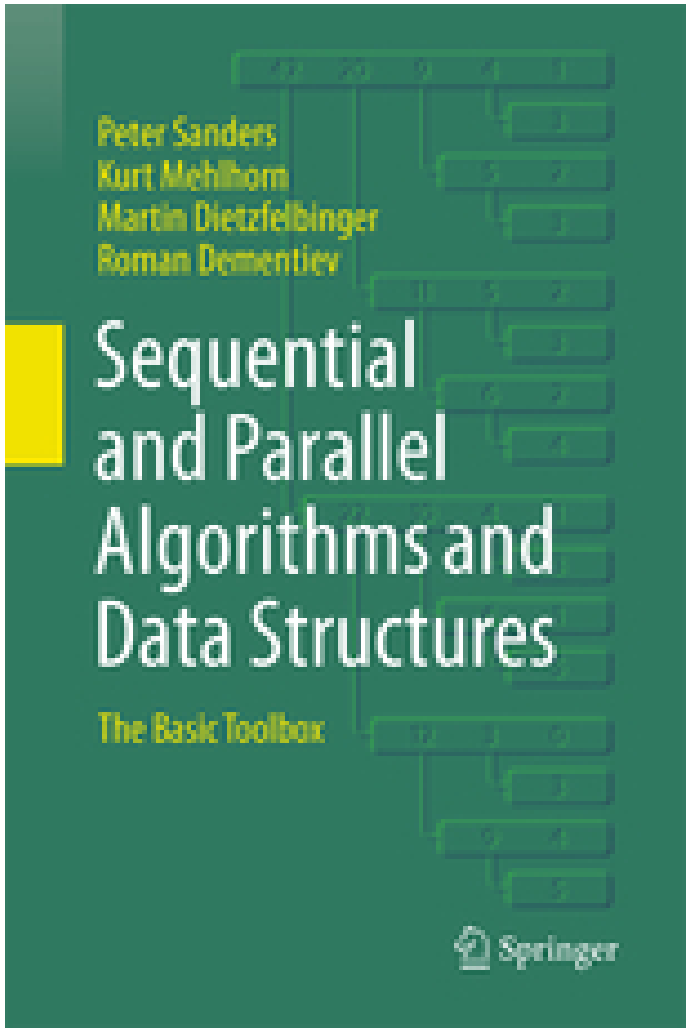
Overview

- ☐ Models, simple examples
- ☐ Matrix multiplikation
- ☐ Broadcasting
- ☐ Sorting
- ☐ General data exchange
- ☐ Load balanciong I, II, III
- ☐ List ranking (conversion list $\rightarrow$ array)
- ☐ Hashing, priority queues
- ☐ Simple graph algorithms

Literature

Script (in German)

+



Figures from the book marked as [Book].

More Literature

[Kumar, Grama, Gupta und Karypis],

Introduction to Parallel Computing. Design and Analysis of Algorithms,

Benjamin/Cummings, 1994. mostly practical, programming oriented

[Leighton], Introduction to Parallel Algorithms and Architectures,

Morgan Kaufmann, 1992.

Theoretical algorithms on concrete interconnection networks

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

PRAM

[Sanders, Worsch],

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

Parallel Computing at ITI Sanders

- Massively parallel sorting, — Michael Axtmann
- Massively parallel graph algorithms, — Sebastian Lamm
- Fault tolerance, — Lukas Hübner
- Shared-memory data structures, — Tobias Maier
- (Hyper)graph partitioning, — Tobias Heuer & Daniel Seemaier
- Comm. eff. alg., — Lorenz Hübschle-Schneider
- SAT-solving and planning, — Markus Iser and Dominik Schreiber
- Geometric algorithms, — Daniel Funke

Role in the CS curriculum

- Elective or “Mastervorzug” **Bachelor!**
- Vertiefungsfach
  - Algorithmentechnik
  - Parallelverarbeitung
- Studienprofil daten-intensives Rechnen

Related Courses

Parallel programming: Tichy, Karl, Streit

Modelle der Parallelverarbeitung: viel theoretischer,

Komplexitätstheorie,... Worsch

Algorithmen in Zellularautomaten: spezieller, radikaler, theoretischer

Worsch

Rechnerarchitektur: Karl

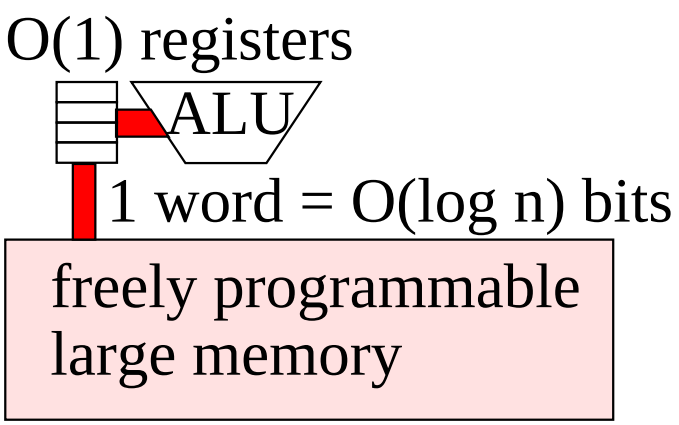
GPUs: Dachsbacher

+ other algorithms lectures

RAM/von Neumann Model

Analysis: count machine instructions — load, store, arithmetics, branch,. . .

- simple
- very successful



Algorithmenanalyse:

- Count cycles: $T(I)$, for given problem instance I.
- Worst case depending on problem size: $T(n) = \max_{|I|=n} T(I)$
- Average case: $T_{\text{avg}}(n) = \frac{\sum_{|I|=n} T(I)}{|\{I : |I| = n\}|}$

  Example: Quicksort has average case complexity $\mathcal{O}(n \log n)$
- Randomized Algorithms: $T(n)$ (worst case) is a random variable.

  We are interested, e.g. in its expectation (later more).

  Don't mix up with average case.

  Example: Quicksort with random pivot selection has expected worst case cost $\mathbb{E}[T(n)] = \mathcal{O}(n \log n)$

Algorithm Analysis: More Conventions

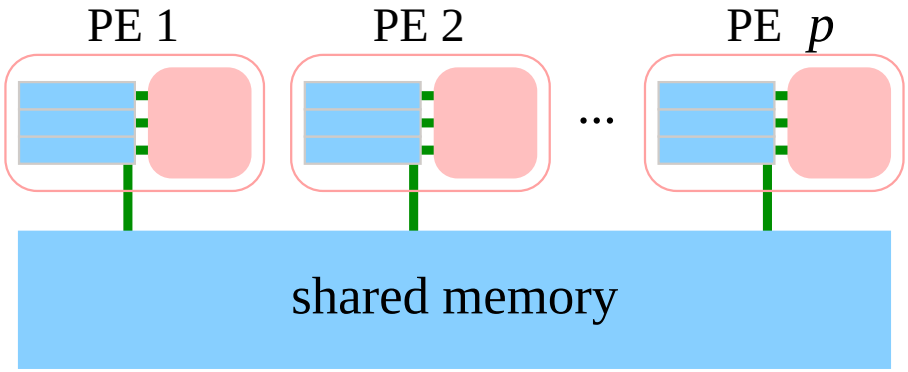
- $\mathcal{O}(\cdot)$ gets rid of cumbersome constants
- Secondary goal: memory
- Execution time can depend on several parameters:

  Example: An efficient variant of Dijkstra's Algorithm for shortest paths needs time $\mathcal{O}(m + n \log n)$ where n is the number of nodes and m is the number of edges. (It always has to be clear what parameters mean.)

A Simple Parallele Model: PRAM

Idea: change RAM as little as possible.

- p PEs (**P**rocessing **E**lements); numbered $1..p$ (or $0..p-1$).
  Every PE knows p.
- One machine instruction per clock cycle and PE synchronous
- Shared global memory



Access Conflicts?

EREW: Exclusive Read Exclusive Write. Concurrent access is forbidden

CREW: Concurrent Read Exclusive Write. Concurrent read OK.

Example: One PE writes, others read = „Broadcast“

CRCW: Concurrent Read Concurrent Write. avoid chaos:

- common: All writers have to agree
  Example: OR in constant time (AND?)
- arbitrary: someone succeeds ($\neq$ random!)
- priority: writer with smallest ID succeeds
- combine: All values are combined, e.g., sum

Example: Global Or

Input in $x[1..p]$

Initilize memory location Result$= 0$

Parallel on PE $i = 1..p$

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

Global And

Initilize memory location Result$= 1$

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

Example: Maximum on Common CRCW PRAM

[JáJá Algorithm 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

$\mathcal{O}(1)$ time

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

```
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
```

Describing Parallel Algorithms

- Pascal-like pseudocode
- Explicitly parallel loops [JáJá S. 72]
- Single Program Multiple Data principle. The PE-index is used to break the symmetry. $\neq$ SIMD !

Analysis of Parallel Algorithms

In principle – just another parameter: p.

Find execution time $T(I,p)$.

Problem: interpretation.

Work: $W = pT(p)$ is a cost measure. (e.g. Max: $W = \Theta(n^2)$)

Span: $T_\infty = \inf_p T(p)$ measures parallelizability.

(absolute) Speedup: $S = T_{\text{seq}}/T(p)$.

Use the best known sequential algorithm.

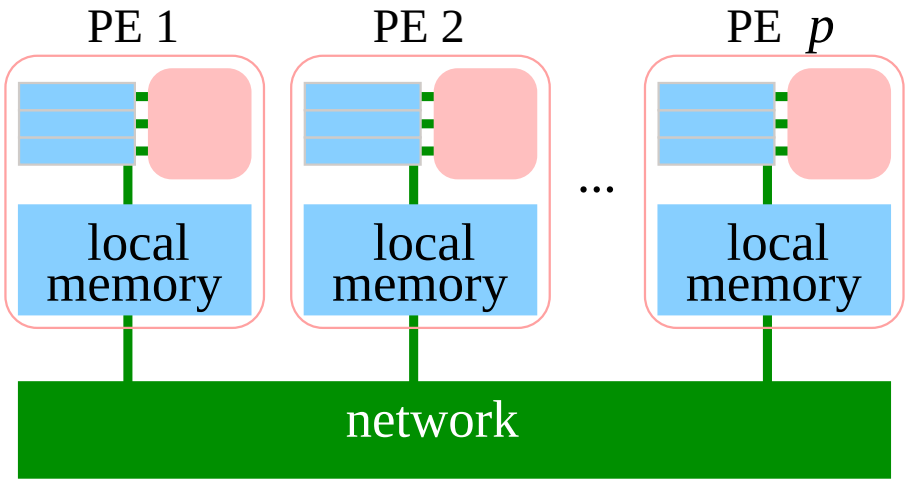
Relative Speedup $S_{\text{rel}} = T(1)/T(p)$ is usually different!

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

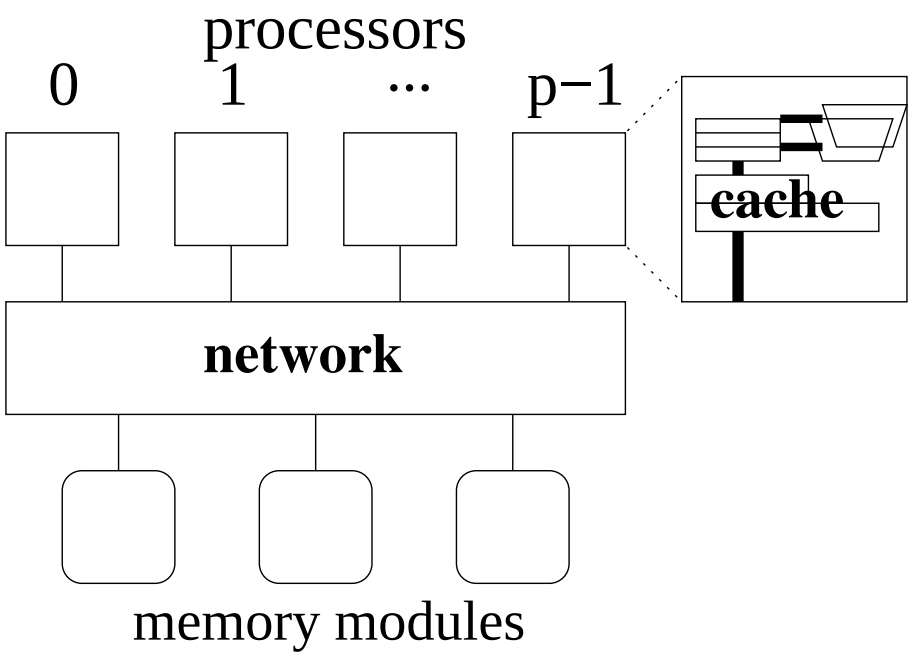
Efficiency: $E = S/p$. Goal: $E \approx 1$ or, at least $E = \Theta(1)$. „Superlinear speedup“: $E > 1$. (possible?). Example maximum: $E = \Theta(1/n)$.

PRAM vs. Real Parallel Computers

Distributed Memory



(Symmetric) Shared Memory



Problems

- Asynchronous $\rightsquigarrow$ design, analysis, implementation, and debugging are much more difficult than for PRAM
- Contention for memory module / cache line

  Example:

  The $\Theta(1)$ PRAM Algorithm for global OR becomes $\Theta(p)$.
- local/cache-memory is (much) faster than global memory
- The network gets more complex with growing p

  while latencies grow
- Contention in the network
- Maximum local memory consumption more important than total memory consumption

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?

Atomic Instuctions: Compare-And-Swap

General and widely available:

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

BeginTransaction

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

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

EndTransaction

Further Operations for Consistent Memory Access:

- Fetch-and-add
- Hardware transactions

Function fetchAndAdd(a, Δ)

expected:= $*a$

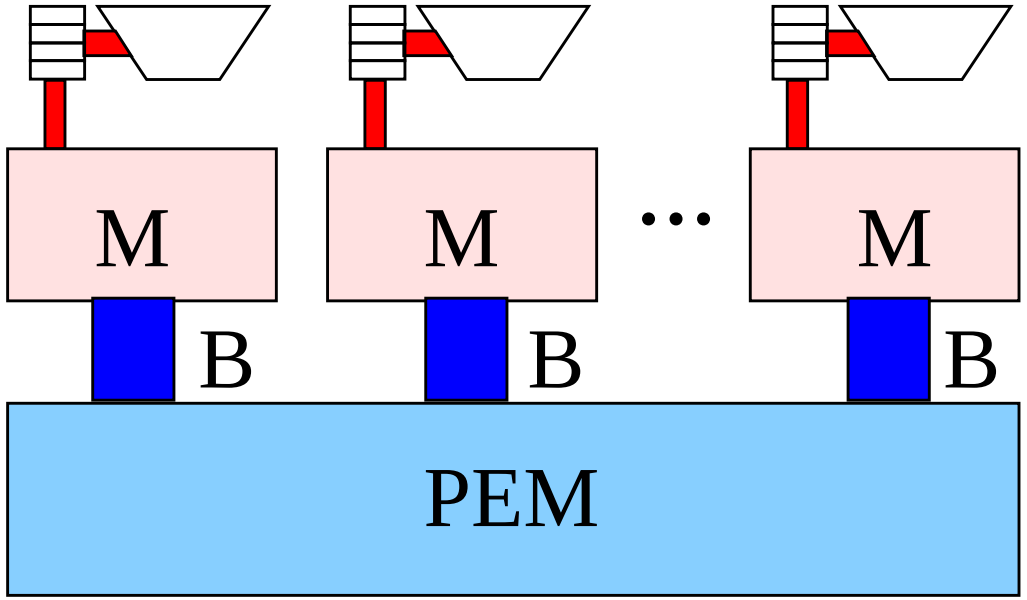
repeat

desired:= expected $+ \Delta$

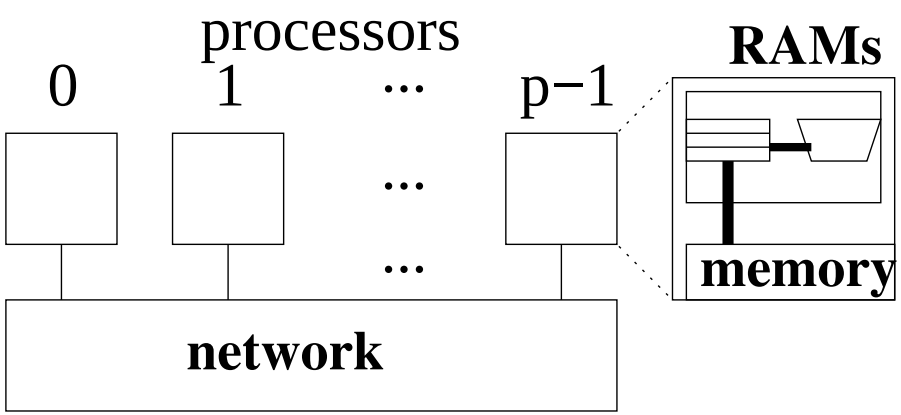
until CAS$(a,$ expected$,$ desired$)$

return desired

Parallel External Memory



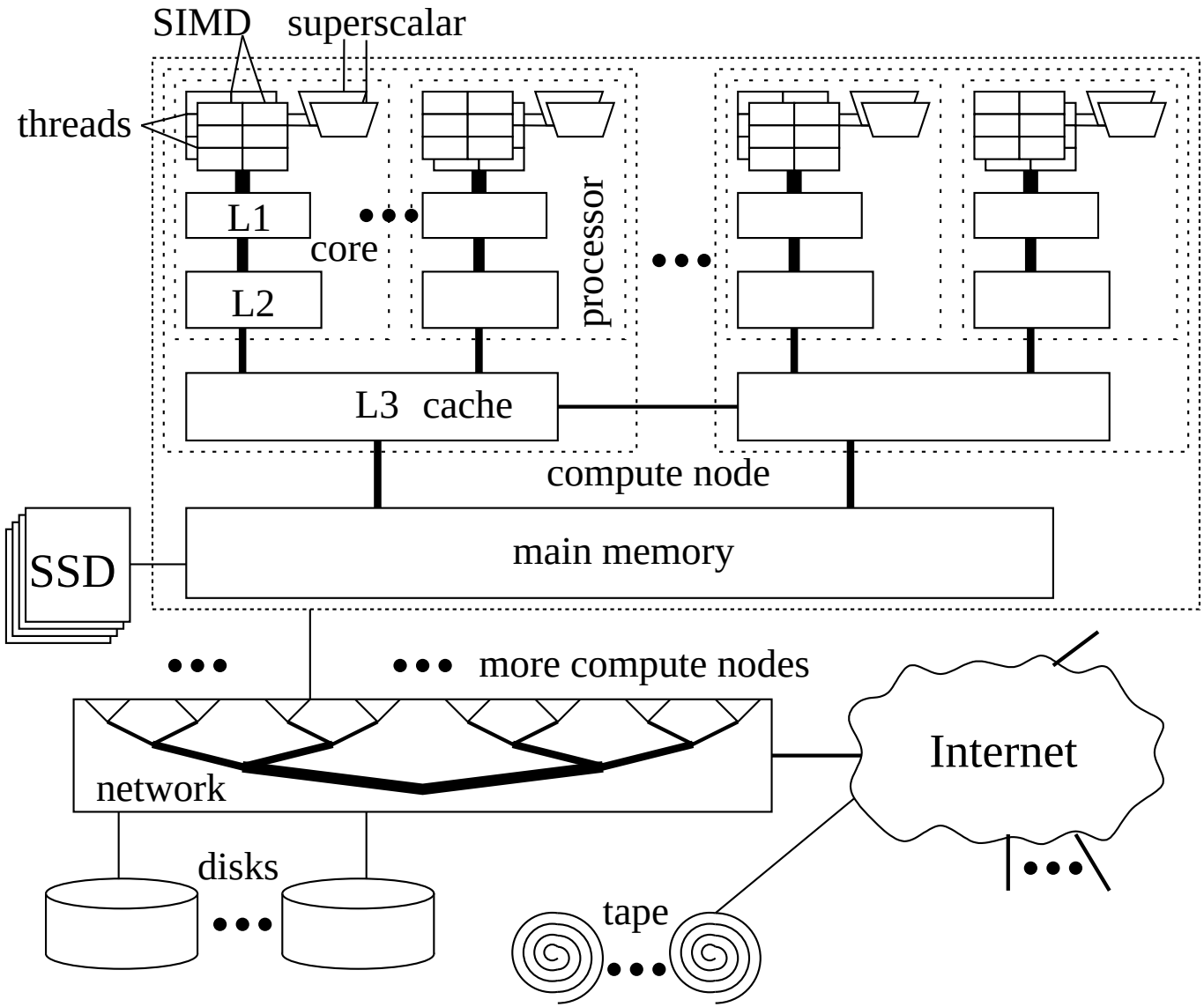
Models with Interconnection Networks



- PEs are RAMs
- asynchronous processing
- Interaction by message exchange

Important: cost model for data exchange

Real Maschines Today



[Book]

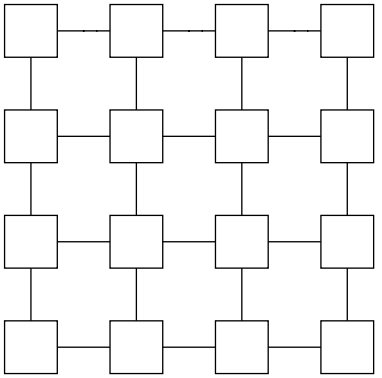
Handling Complex Hierarchies

Proposition: we get quite far with flat Models, in particular for shared memory.

- Design distributed, implement hierarchy adapted
- Shared-memory subroutines on nodes

Explicit „Store-and-Forward“

- We know the interconnection graph $(V = \{1, \ldots, p\}, E \subseteq V \times V)$. Variants:
  - $V = \{1, \ldots, p\} \cup R$ with additional „dumb“ router nodes (perhaps with buffer memory).
  - Buses $\rightarrow$ Hyperedges

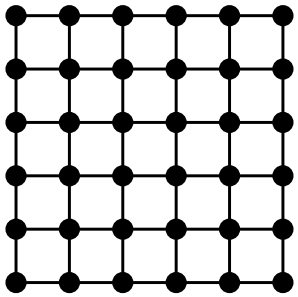


- In each time step each edge can transport up to k' data packets of constant length (usually $k' = 1$)
- On a k-port-machine, each node can simultaneously send or receive k Packets gleichzeitig senden oder empfangen. $k = 1$ is called single-ported.

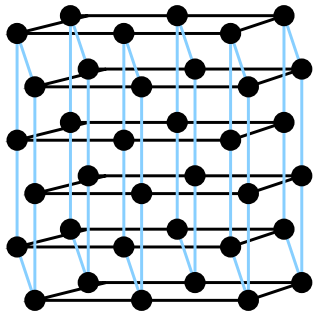
Discussion

+ simple formulation
- low level $\Rightarrow$ „messy algorithms“
- Hardware router allow fast communication whenever an unloaded communication path is found.

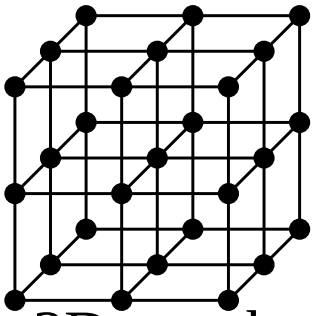
Typical Interconnection Networks



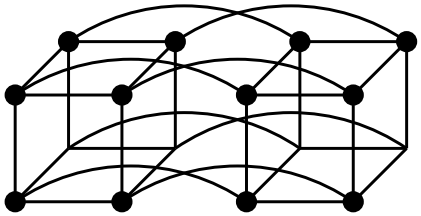
mesh



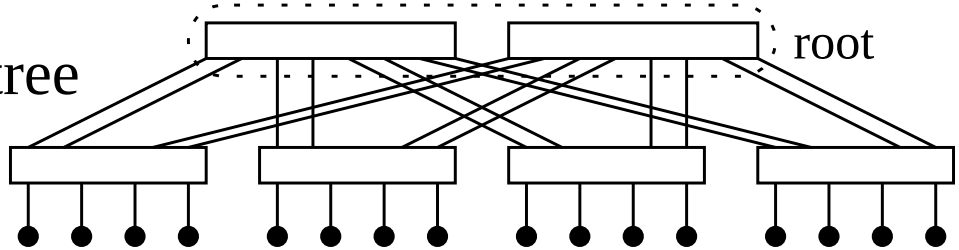
torus



3D−mesh



hypercube



[Book]

Fully Connected Point-to-Point

- $E = V \times V$, single ported
- $T_{\text{comm}}(m) = \alpha + m\beta$. ($m =$ message length in machine words)
- \+ Realistic handling of message lengths
- \+ Many interconnection networks approximate fully connected networks $\Rightarrow$ sensible abstraction
- \+ No overloaded edges $\rightarrow$ OK for hardware router
- \+ „artificially“ increasing α, β $\rightarrow$ OK for „weak“ networks
- \+ Asynchronous model
- − A bit of hand waving for real networks

Fully Connected: Variants

What can PE I do in time $T_{\text{comm}}(m) = \alpha + m\beta$?

message length m.

half duplex: $1\times$send or $1\times$receive (also called simplex)

Telephone: $1\times$send to PE j and $1\times$receive from PE j

(full)duplex: $1\times$send and $1\times$receive.

Arbitrary comunication partners

Effect on running time:

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

BSP Bulk Synchronous Parallel

[McColl LNCS Band 1000, S. 46]

Machine described by three parameters: p, L and g.

L: Startup overhead for one collective message exchange – involving all PEs

g: gap$\approx \frac{\text{computation speed}}{\text{communication bandwidth}}$

Superstep: Work locally. Then collective global synchronized data exchange with arbitrary messages.

w: max. local work (clock cycles)

h: max. number of machine words sent or received by a PE (h-relation)

Time: $w + L + gh$

BSP versus Point-to-Point

By naive direct data delivery:

Let $H =$ max #messages of a PEs.

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

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

By all-to-all and direct data delivery:

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

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

By all-to-all and indirect data delivery:

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

Thus $L = \Omega(\alpha \log p)$ 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.

Redefinition of h to # blocks of size B, e.g. $B = \Theta(\alpha/\beta)$.

Let M_i be the set of messages sent or received by PE i.
Let $h = \max_i \sum_{m \in M_i} \lceil |m|/B \rceil$.
Let g the gap between sending packets of size B.
Then, once more

$$w + L + gh$$

is the time for a super step.

BSP* versus Point-to-Point

With naive direct data delivery:

$L \approx \alpha \log p$

$g \approx B\beta$

BSP$^+$

We augment BSP by collective operations

broadcast

(all-)reduce

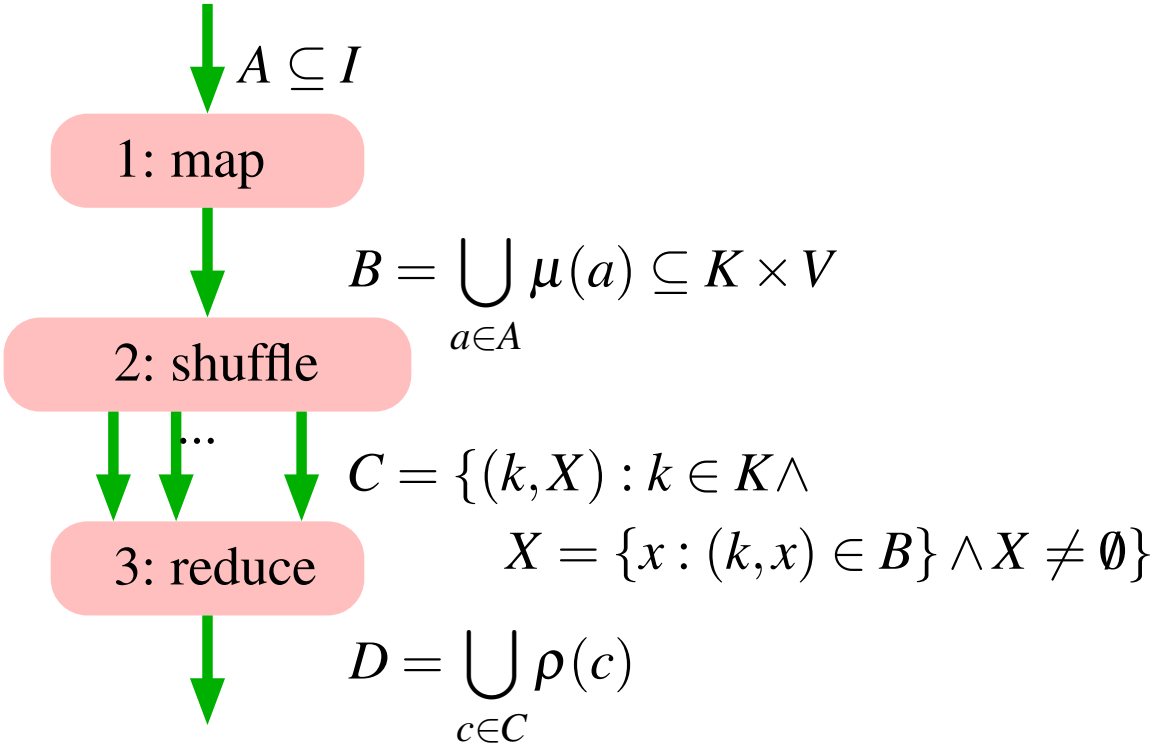
prefix-sum

with message lengths h.

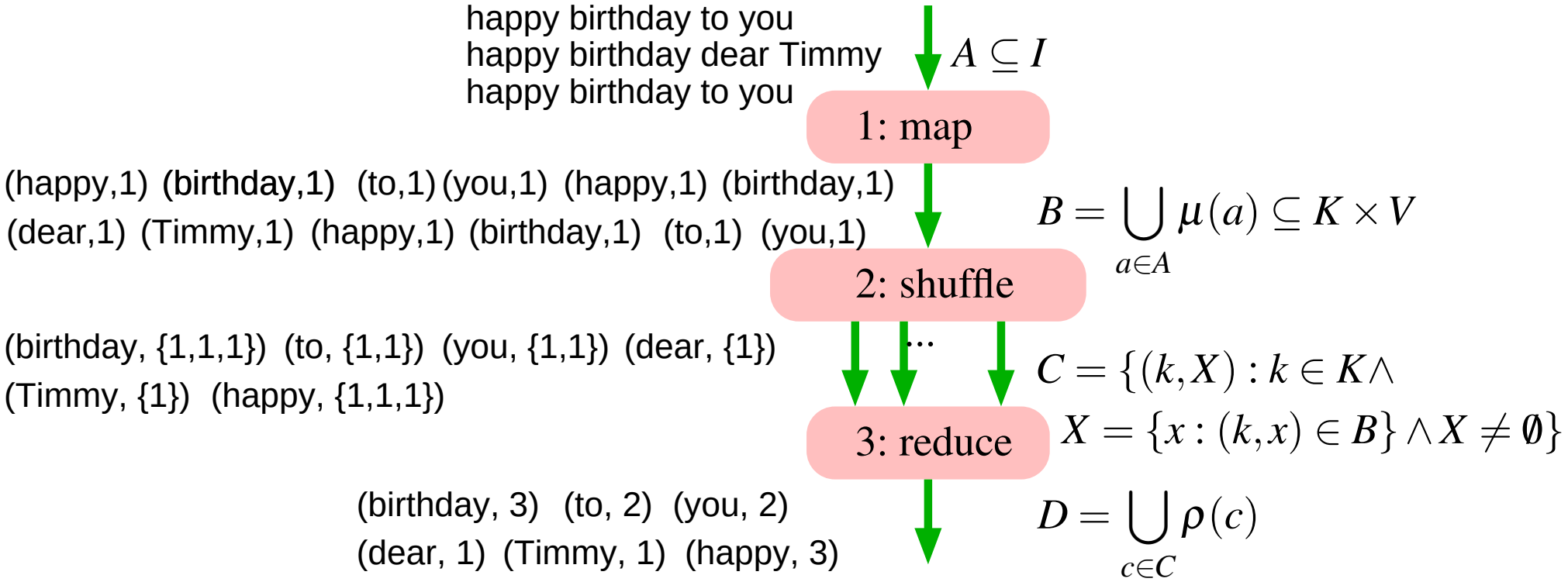
Stay tuned for algorithms which justify this.

BSP*-algorithms are up to a factor $\Theta(\log p)$ slower than BSP$^+$-algorithms.

MapReduce

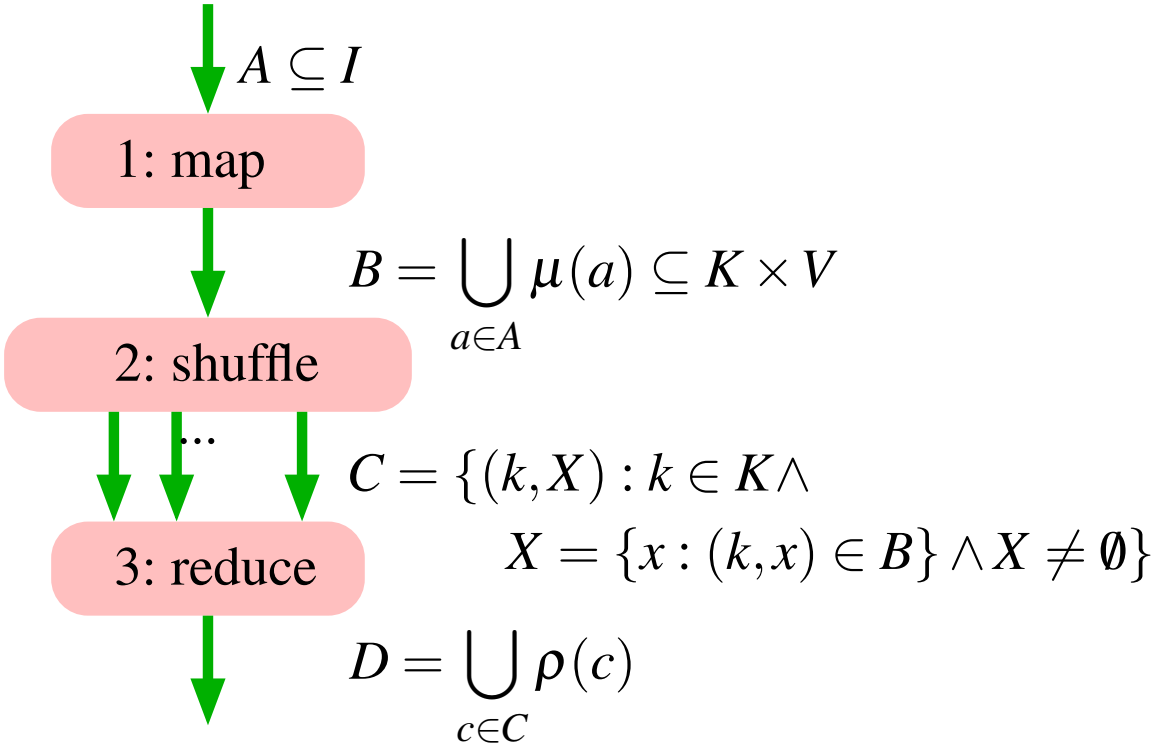

$A \subseteq I$
1: map
$B = \bigcup_{a \in A} \mu(a) \subseteq K \times V$
2: shuffle
...
$C = \{(k,X) : k \in K \wedge$
$X = \{x : (k,x) \in B\} \wedge X \neq \emptyset\}$
3: reduce
$D = \bigcup_{c \in C} \rho(c)$

MapReduce Example WordCount



MapReduce Discussion

+ Abstracts away difficult issues
  * parallelization
  * load balancing
  * fault tolerance
  * memory hierarchies
  * ...
- Large overheads
- Limited functionality



MapReduce – MRC Model

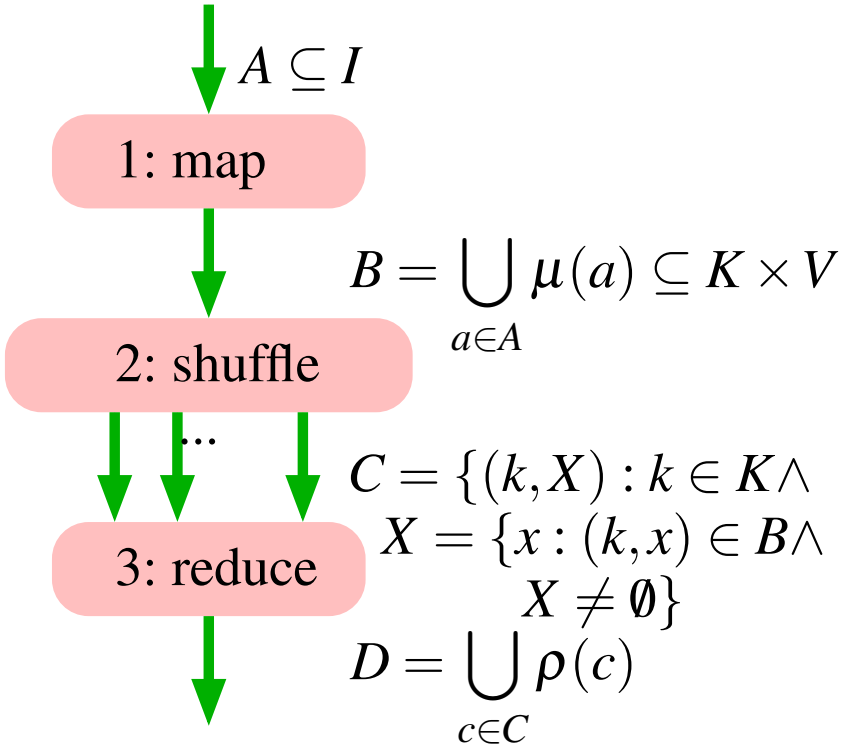
[Karloff Suri Vassilvitskii 2010]

A problem is in MRC iff for input of size n:

- solvable in $\mathcal{O}(\text{polylog}(n))$ MapReduce steps
- μ and ρ evaluate in time $\mathcal{O}(\text{poly}(n))$
- μ and ρ use space $\mathcal{O}\left(n^{1-\varepsilon}\right)$ ("substantially sublinear")
- overall space for B $\mathcal{O}\left(n^{2-\varepsilon}\right)$ ("substantially subquadratic")

Roughly: count steps,

very loose constraints on everything else



MapReduce – MRC Model

Criticism

A problem is in MRC iff for input of size n:

- solvable in $\mathcal{O}(\text{polylog}(n))$ MapReduce steps
- μ and ρ evaluate in time $\mathcal{O}(\text{poly}(n))$ — n^2? n^{42}? "big" data?
- μ and ρ use space $\mathcal{O}\left(n^{1-\varepsilon}\right)$ ("substantially sublinear")
- overall space for B $\mathcal{O}\left(n^{2-\varepsilon}\right)$ ("substantially subquadratic") — "big" data?

Roughly: count steps, — speedup? efficiency?

very loose constraints on everything else

MapReduce – MRC+ Model

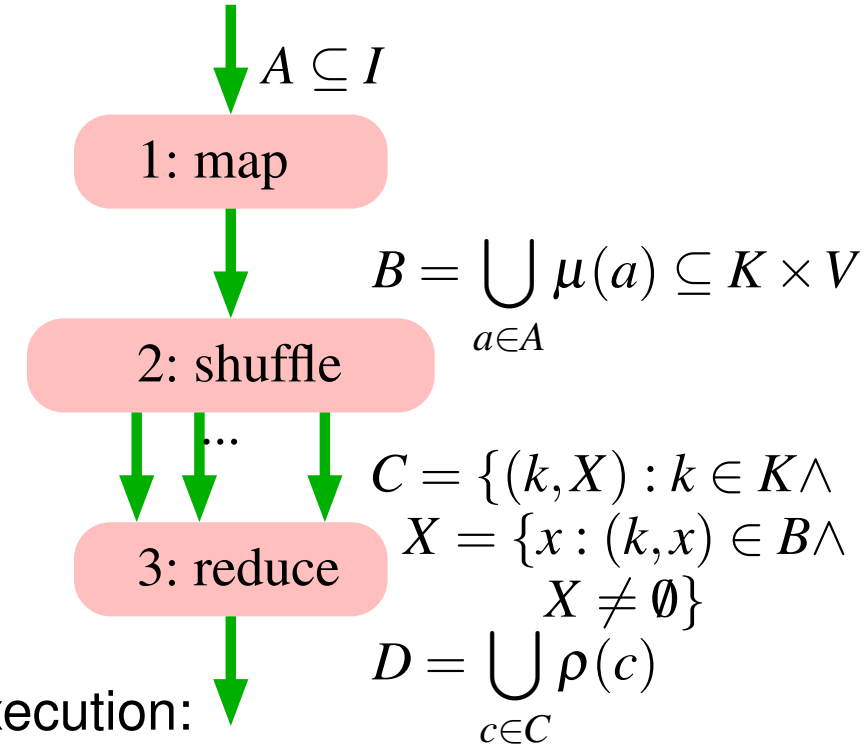
[Sanders IEEE BigData 2020]

w Total work for μ, ρ

$\hat{w}$ Maximal work for μ, ρ

m Total data volume for $A \cup B \cup C \cup D$

$\hat{m}$ Maximal object size in $A \cup B \cup C \cup D$,



Theorem: Lower and upper bound for parallel execution:

$\Theta\left(\frac{w}{p} + \hat{w} + \log p\right)$ parallel time,

$\Theta\left(\frac{m}{p} + \hat{m} + \log p\right)$ bottleneck communication volume.

Implementing MapReduce on BSP

2 Supersteps:

1. Map local data.

   Send $(k,v) \in B$ to PE $h(k)$ (hashing)

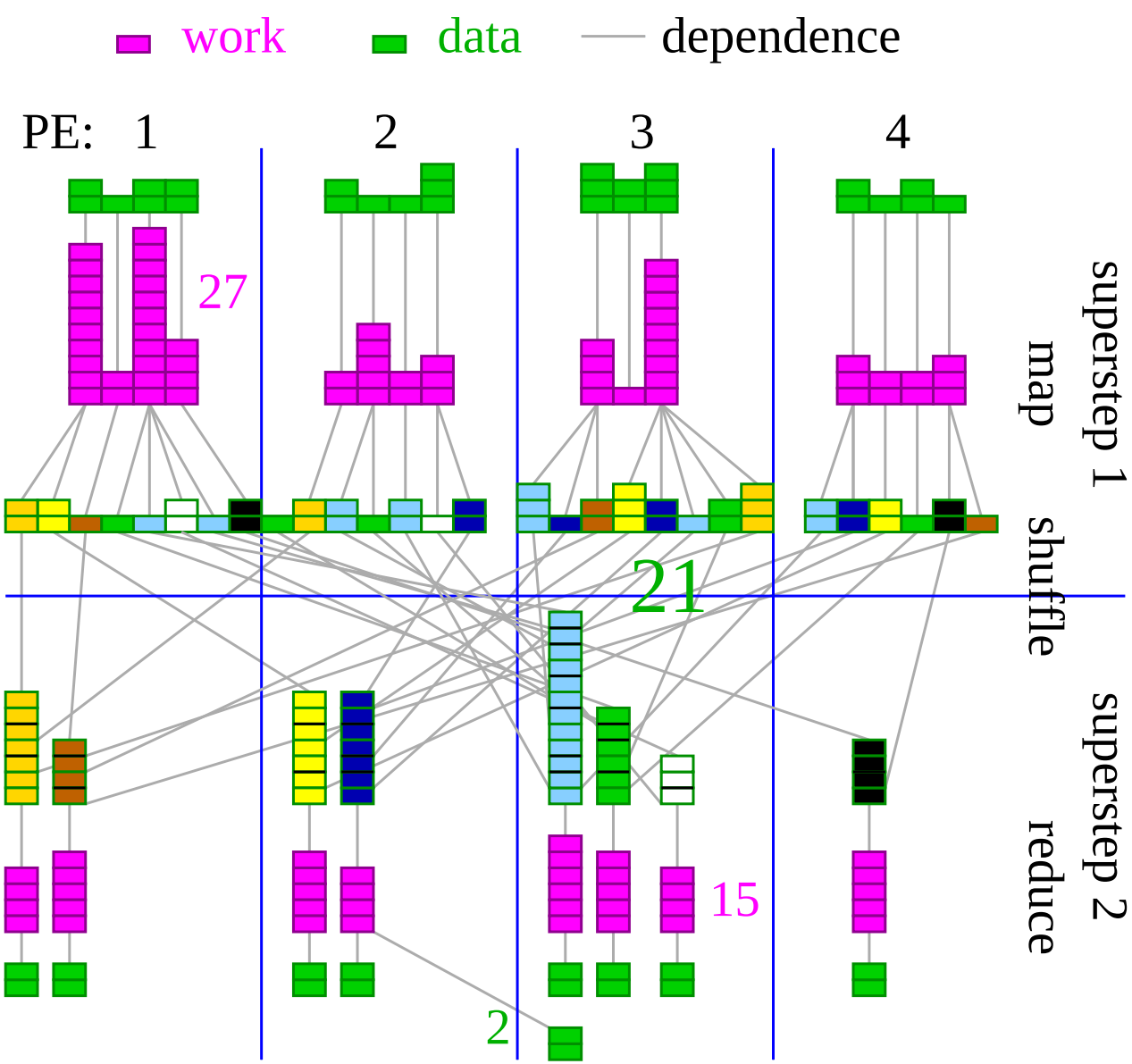
2. Receive elements of B.

   Build elements of C.

   Run reducer.

   Send all but first result of a reducer to a random PE.

MapReduce on BSP – Example



MapReduce on BSP – Analysis

Time

$$2L + \mathcal{O}\left(\frac{w}{p} + g\frac{m}{p}\right)$$

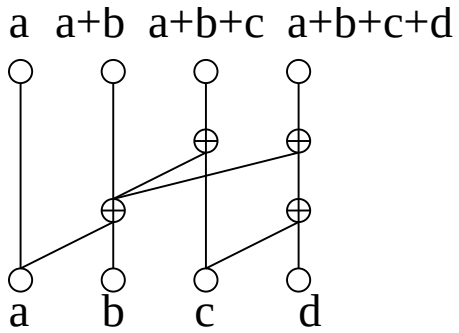
if

$$w = \Omega(\hat{w}p\log p) \text{ and } m = \Omega(\hat{m}p\log p)$$

Graph- und Circuit Representation of Algorithms

Many computations can be represented as a

directed acyclic graph



- Input nodes have indegree 0

  and a fixed output

- Ouput nodes have outdegree 0

  and indegree 1

- The indegree is bounded by

  a small constant.

- Inner nodes compute a function, that can be computed in constant time.

Circuits

- Variant: When a constant number of bits rather than machine words are processed, we get circuits.
- The depth $d(S)$ of the computation-DAG is the number of inner nodes on the longest path from an input to an output $\sim$ time
- One circuit for each input size (specified algorithmically) $\Rightarrow$ circuit family

Example: Associative Operations (=Reduction)

Satz 1. *Let $\oplus$ denote an associative operator that can be computed in constant time. Then,*

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

can be computed in time $\mathcal{O}(\log n)$ on a PRAM and in time $\mathcal{O}(\alpha \log n)$ on a linear array with hardware router

Example: $+$, $\cdot$, max, min, ... (example for non-commutative?)

Proof Outline for $n = 2^k$ (wlog?)

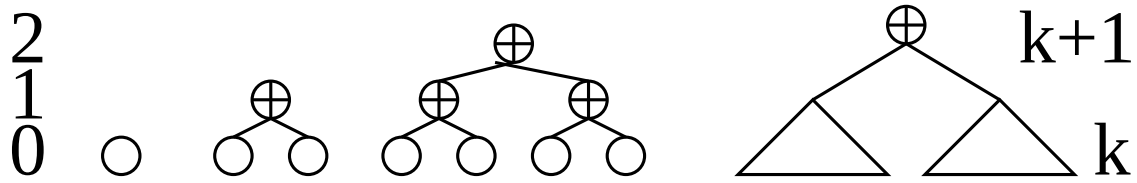
Induction hypothesis: $\exists$ circuit of depth k for $\bigoplus_{i<2^k} x_i$

$k = 0$: trivial

$k \rightsquigarrow k+1$:

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

■



PRAM Code

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

active := 1

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

  if active **then**

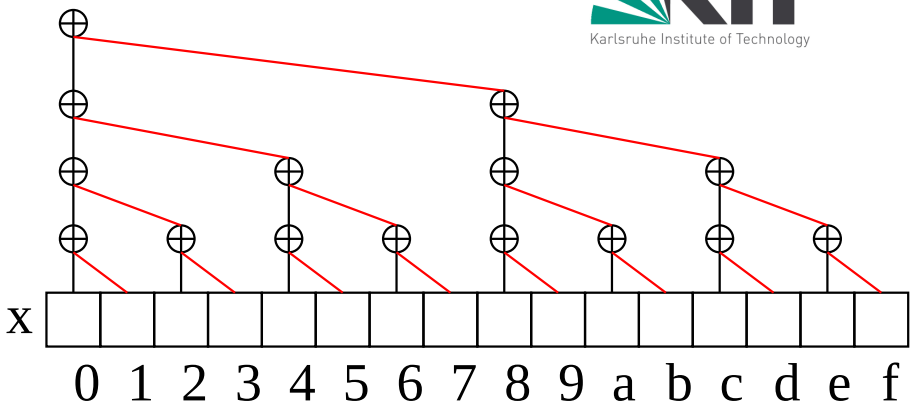
    if bit k of i **then**

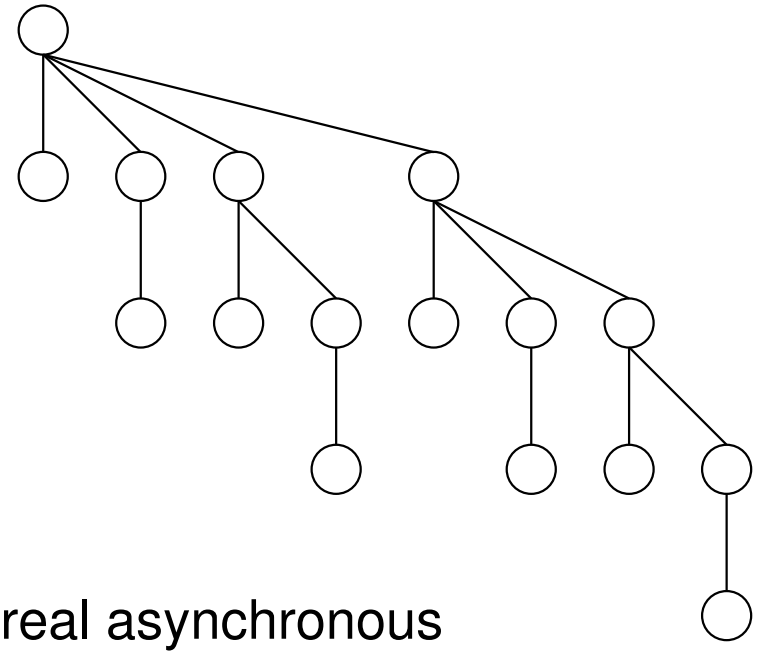
      active := 0

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

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

// result is in x_0





Careful: much more complicated on a real asynchronous shared-memory machine.

Speedup? Efficiency?

$\log x$ here always $\log_2 x$

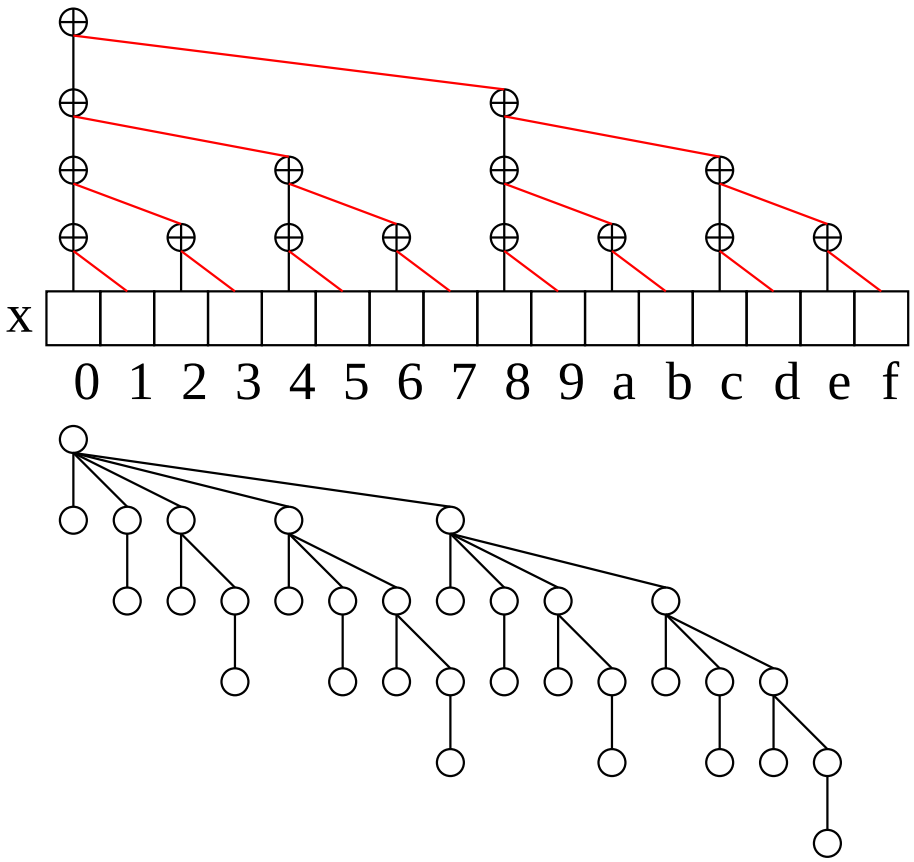
Analysis

n PEs

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

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

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



Less is More (Brent's Principle)

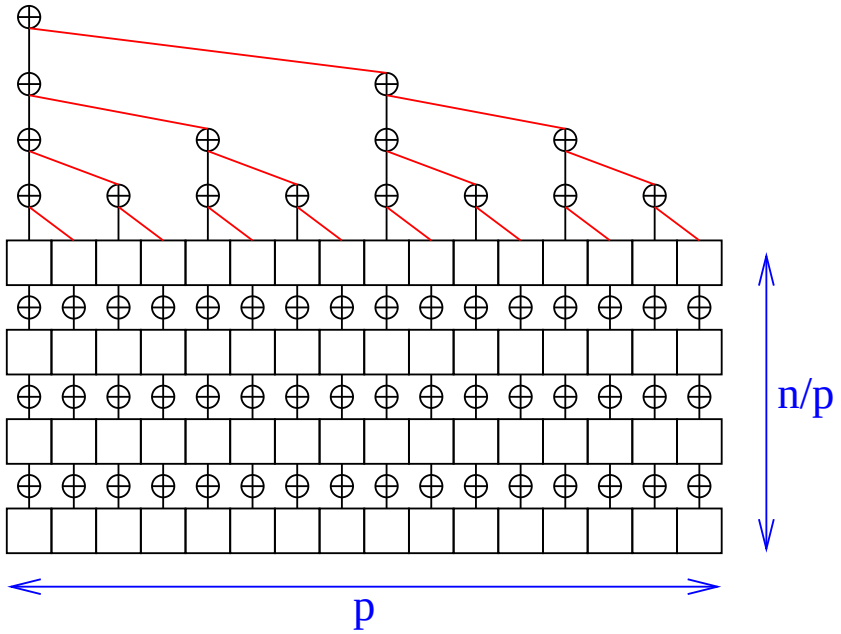
p PEs

Each PE adds

n/p elements sequentially.

Then parallel sum

for p subsums



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

Efficiency

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

if $n \gg p \log p$

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 $i - 2^k$

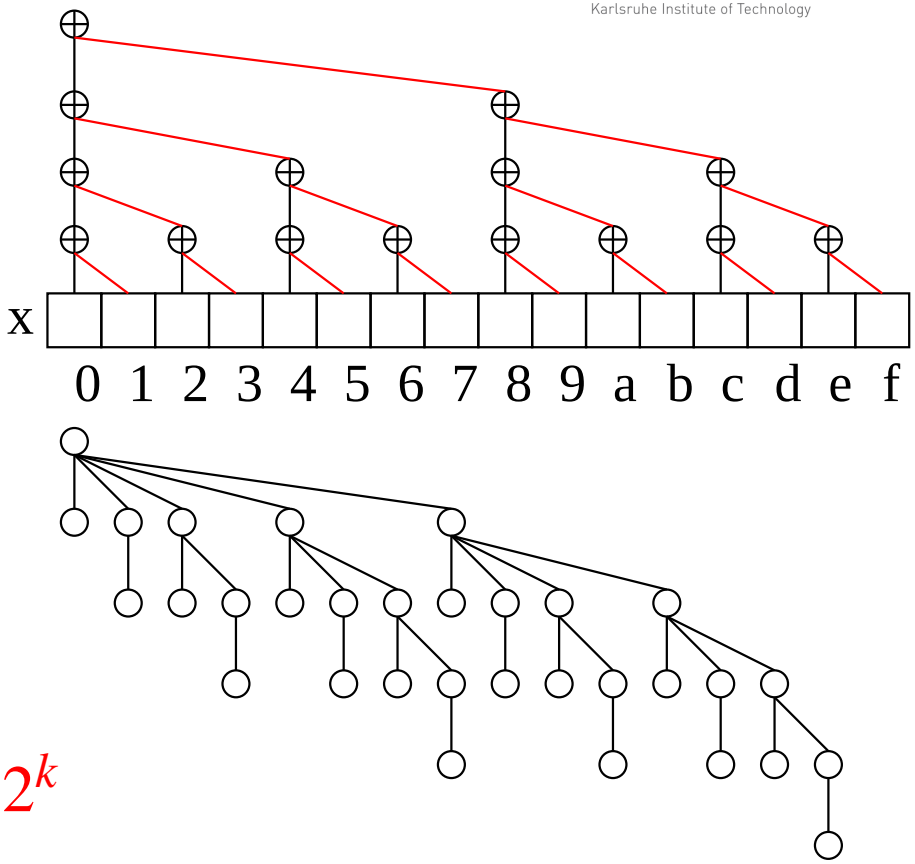
      active := 0

    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



Analysis

fully connected: $\Theta((\alpha+\beta)\log p)$

linear array: $\Theta(p)$: step k needs time 2^k.

linear array with router: $\Theta((\alpha+\beta)\log p)$, since edge congestion is one in every step

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

Arbitrary $n > p$: additional time $T_{\text{seq}}(n/p)$

Discussion Reductions Operation

- Binary tree yields logarithmic running time
- Useful for most models
- Brent's principle: inefficient algorithms get efficient by using less PEs
- Later: reduction of complex objects, e.g., vectors, matrices

Matrix Multiplikation

Given: Matrices $A \in \mathbf{R}^{n\times n}$, $B \in \mathbf{R}^{n\times n}$

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

$\mathbf{R}$: semiring

$C = ((c_{ij})) = A \cdot B$ well-known:

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

work: $\Theta\left(n^3\right)$ arithmetical operations

(better algorithms if R allows subtraction)

A first PRAM Algorithm

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

One PE für each product $c_{ikj} := a_{ik} b_{kj}$

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

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

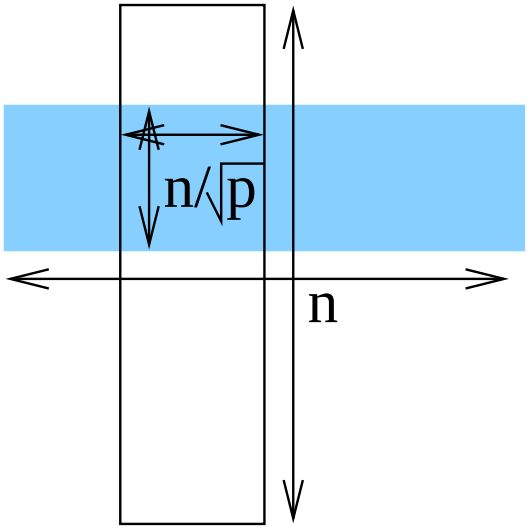
Distributed Implementation I

$p \leq n^2$ PEs

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

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

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



Assign n^2/p of the c_{ij} to each PE

- limited scalability
- high communication volume. Time $\Omega\left(\beta \frac{n^2}{\sqrt{p}}\right)$

Distributed Implementation II-1

[Dekel Nassimi Sahni 81, KGGK Section 5.4.4]

Assume $p = N^3$, n is a multiple of N

View A, B, C as $N \times N$ matrices,

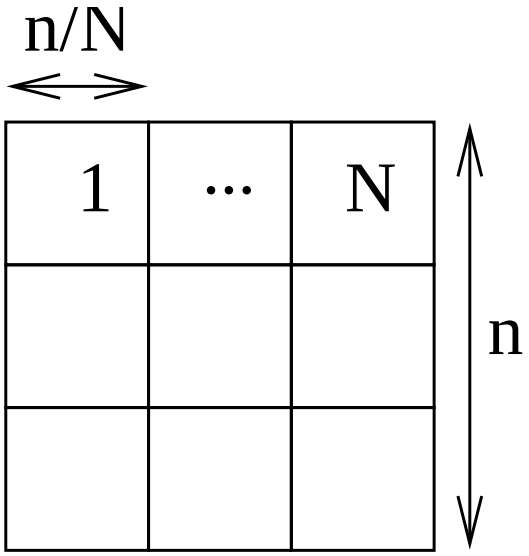
each element is a $n/N \times n/N$ matrix

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

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

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

One PE for each product $c_{ikj} := a_{ik} b_{kj}$



Distributed Implementation II-2

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

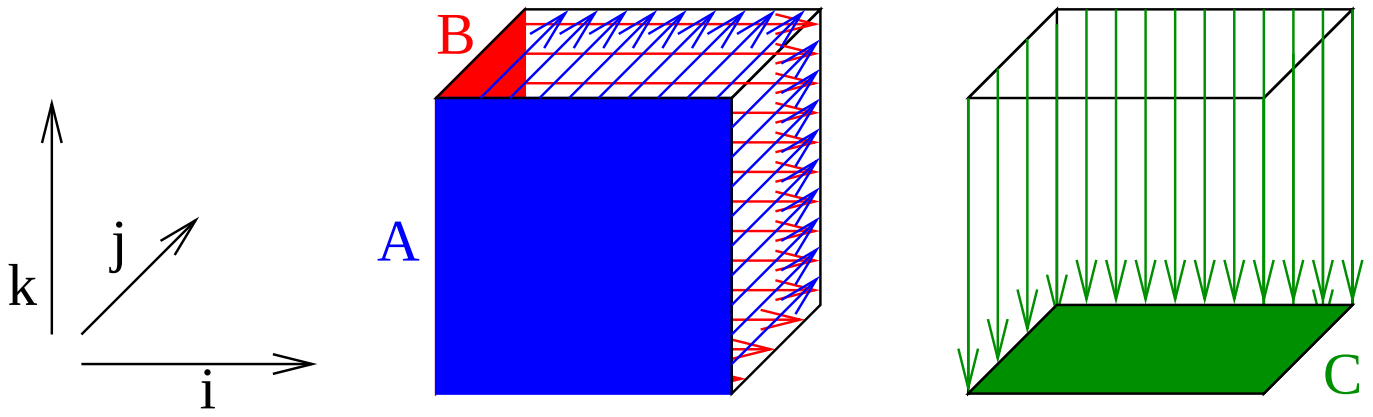
store b_{kj} in PE $(1,k,j)$

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 $(i,1,j)$



Analysis, Fully Connected, etc.

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 $(i,1,j)$

Communication:

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

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

Discussion Matrix Multiplikation

- PRAM Alg. is a good starting point
- DNS algorithm saves communication but needs factor $\Theta\left(\sqrt[3]{p}\right)$ more space than other algorithms

$\rightsquigarrow$ good for small matrices (for big ones communication is irrelevant)

- Pattern for dense linear algebra:

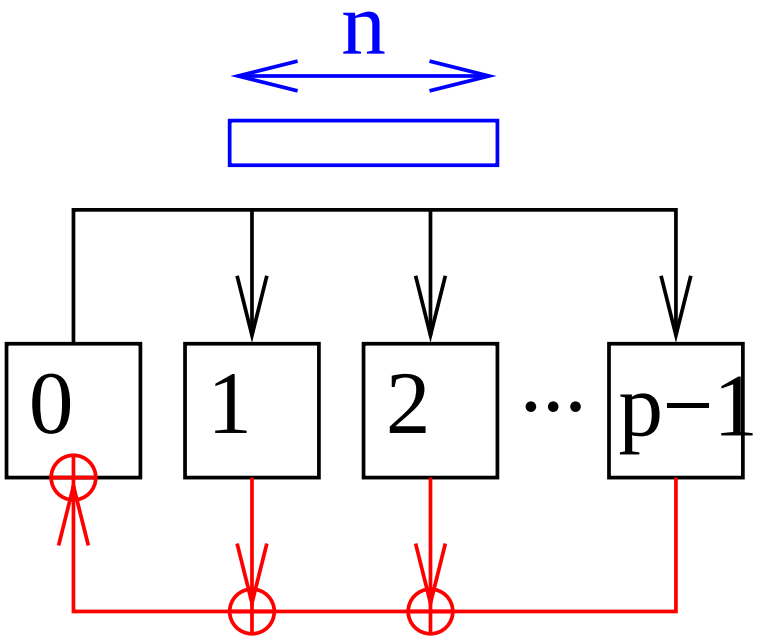
  Local Ops on submatrices $+$ Broadcast $+$ Reduce

  e.g. matrix-vector-product, solve lin. eq. syst.,...

Broadcast and Reduction

Broadcast: One for all

One PE (e.g. 0) sends message of length n to all PEs



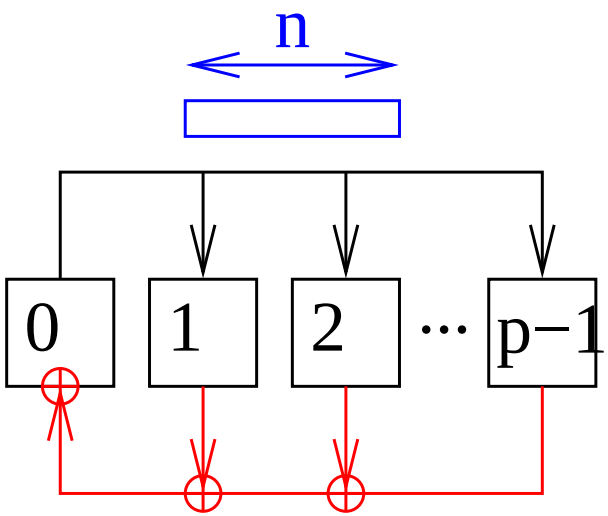
Reduction: One for all

One PE (e.g. 0) receives sum of p messages of length n

(vector addition$\neq$ local addition)

Broadcast $\rightsquigarrow$ Reduction

- turn around direction of communication
- add corresponding parts of arriving and own messages



All the following broadcast algorithms yield reduction algorithms for commutative and associative operations.

Most of them (except Johnsson/Ho and some network embeddings) also work for noncommutative operations.

Modelling Assumptions

- fully connected
- fullduplex – parallel send and receive

Variants: halfduplex, i.e., send or receive, BSP, embedding into concrete networks

Naive Broadcast [KGGK Section 3.2.1]

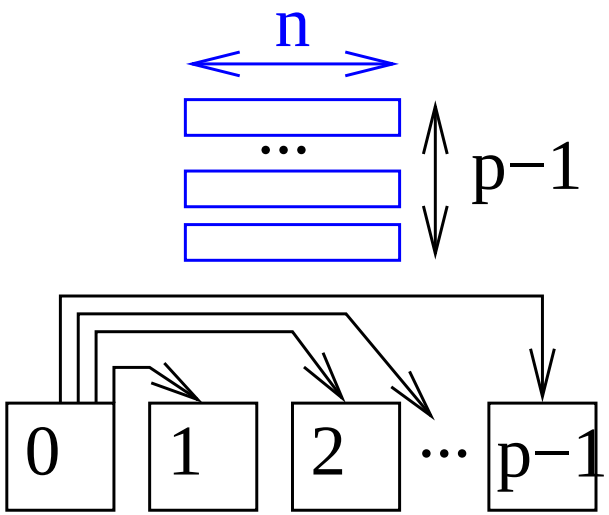
Procedure naiveBroadcast$(m[1..n])$

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

PE $i > 0$: receive m

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

nightmare for implementing a scalable algorithm



Binomial Tree Broadcast

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

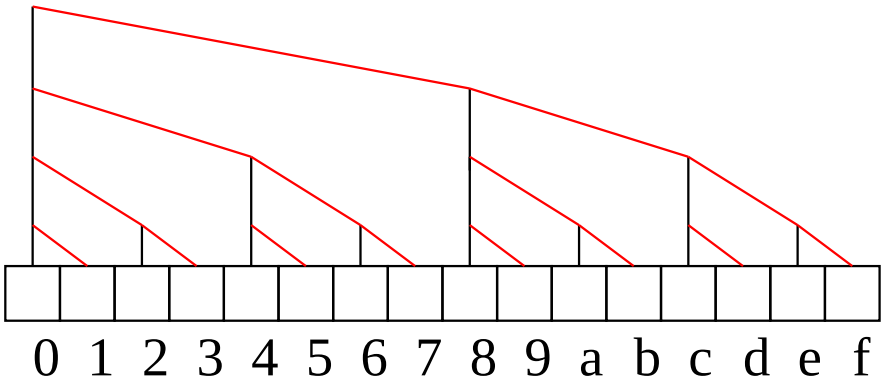
PE index $i \in \{0,\ldots,p-1\}$

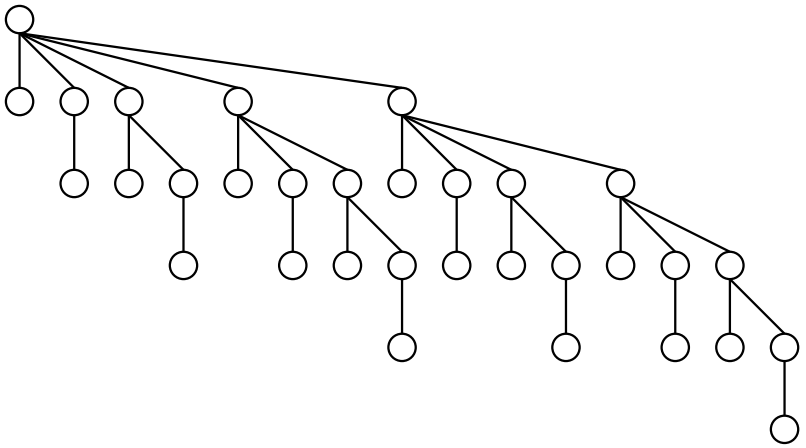
//Message m located on PE 0

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 $\geq p$

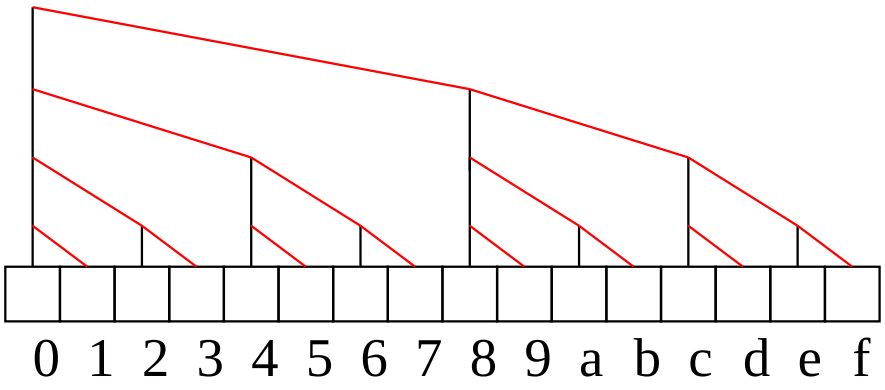


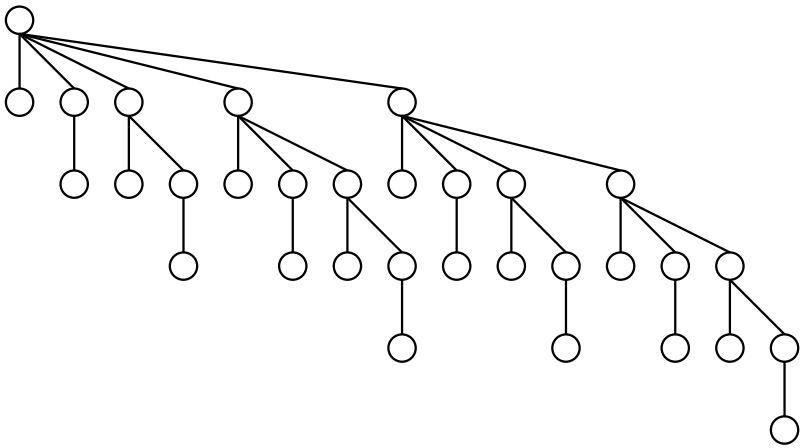


Analysis

- Time: $\lceil \log p \rceil \, (n\beta + \alpha)$
- Optimal for $n = 1$
- Embeddable into linear array

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





Linear 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

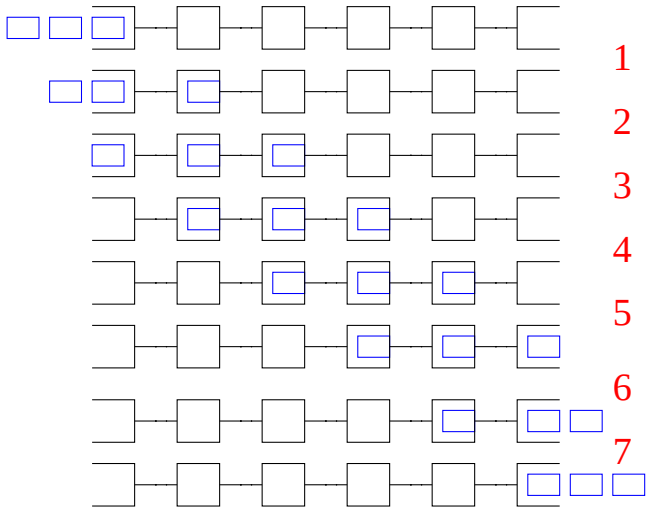
define piece j as $m[(j-1)\frac{n}{k}+1..j\frac{n}{k}]$

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

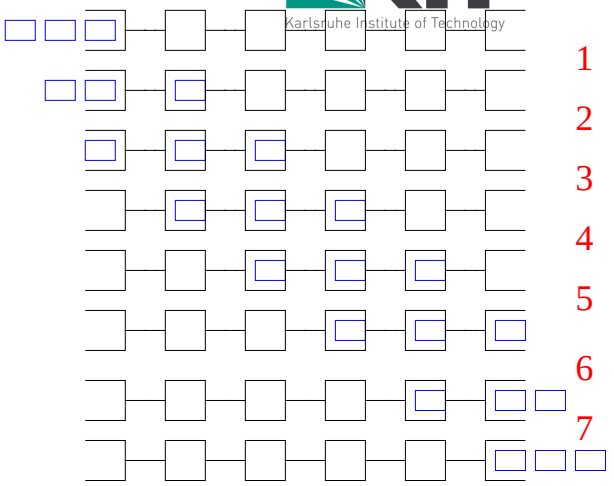
receive piece j from PE $i-1$ // noop if $i = 0$ or $j = k+1$

and, concurrently,

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



Analysis

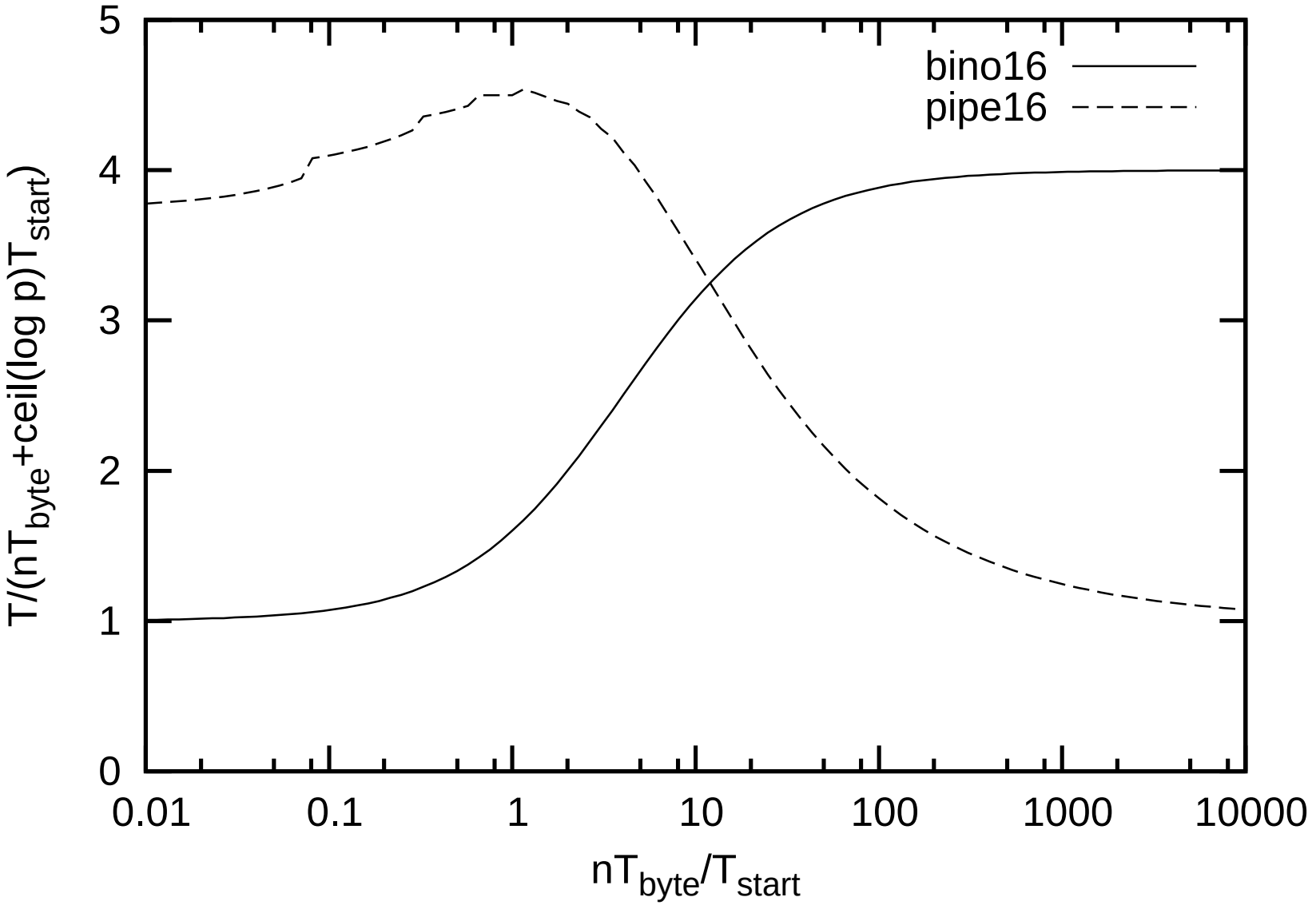


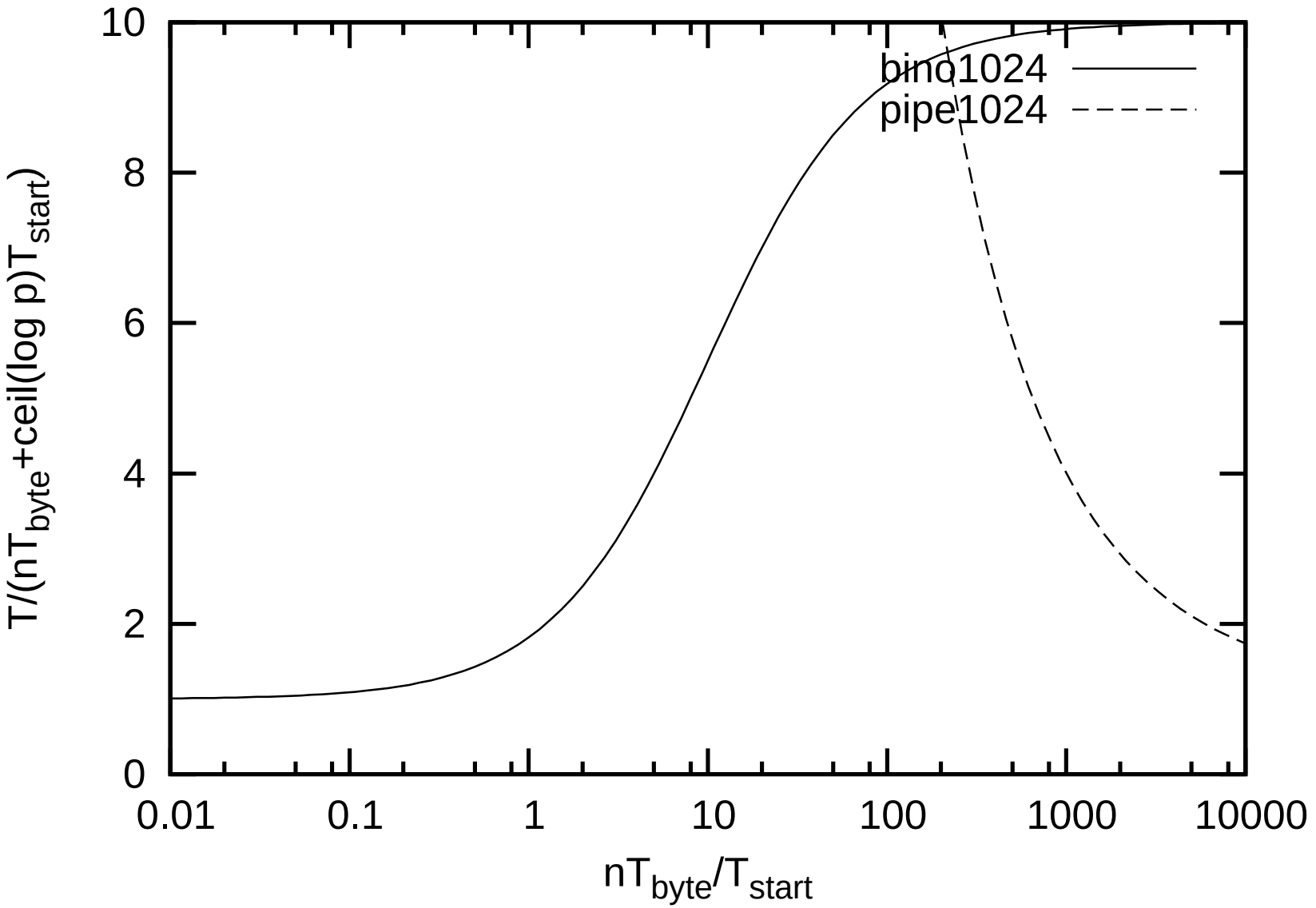
- Time $\frac{n}{k}\beta + \alpha$ per step
  ($\neq$ Iteration)
- $p-1$ steps until first packet arrives
- Then 1 step per further packet

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

optimal 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

Discussion

- Linear pipelining is optimal for fixed p und $n \to \infty$
- But for large p extremely large messages needed

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

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

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

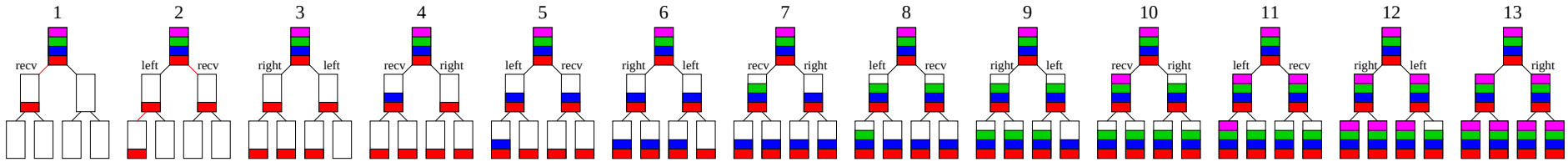
define piece j as $m[(j-1)\frac{n}{k}+1..j\frac{n}{k}]$

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

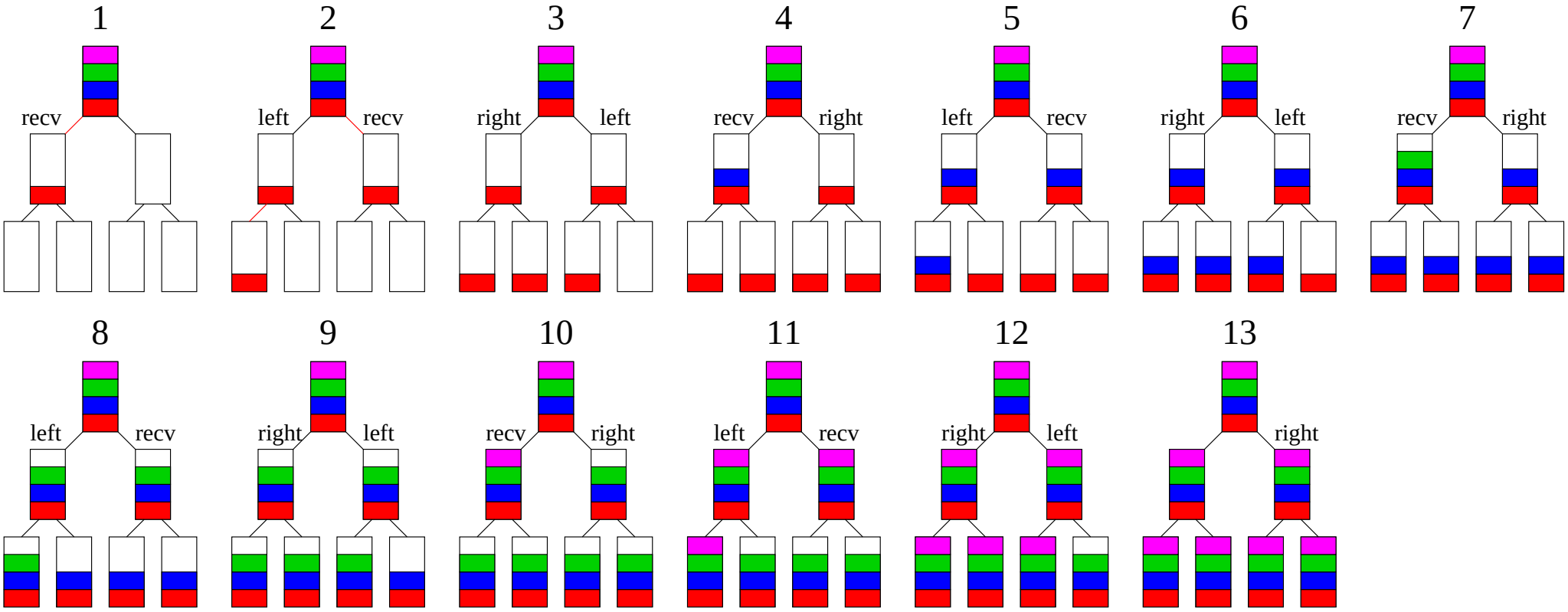
if parent exists **then** receive piece j

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

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



Example


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

Analysis

- time $\frac{n}{k}\beta + \alpha$ per step ($\neq$ iteration)
- $2j$ steps until first packet reaches level j
- how many levels? $d := \lfloor \log p \rfloor$
- then 3 steps for each further packet

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

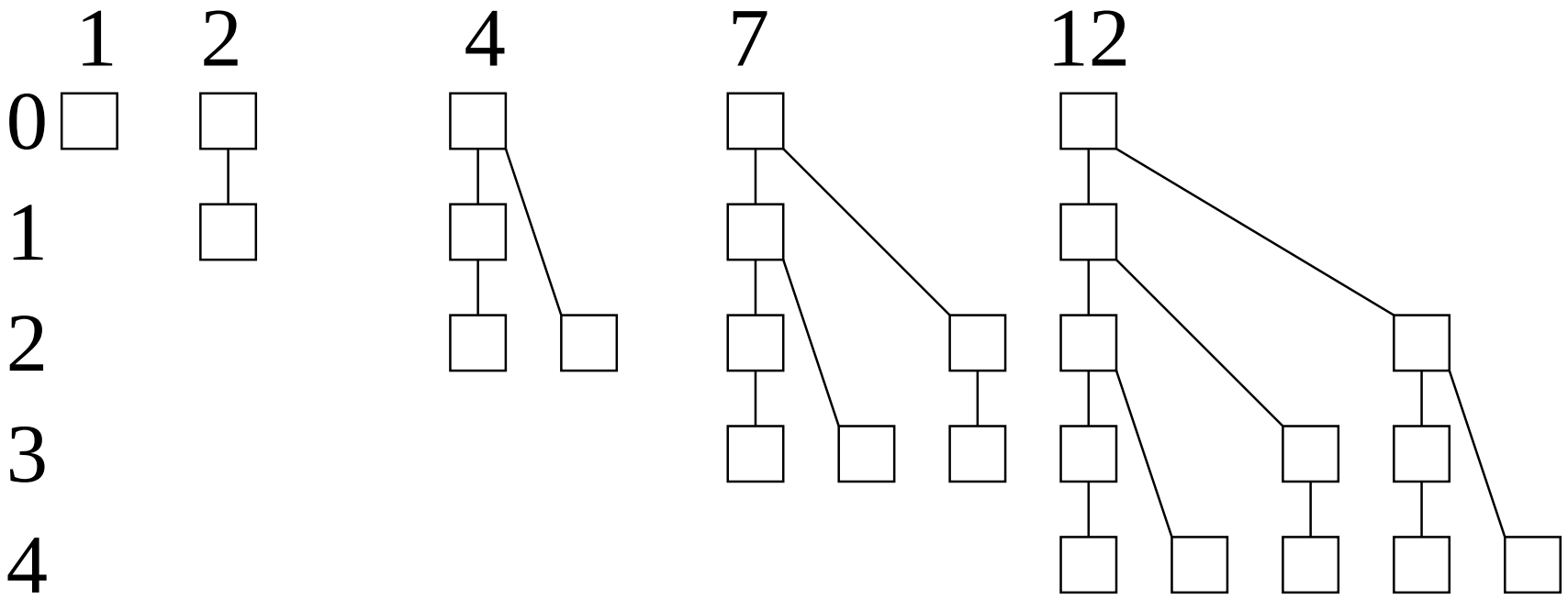
optimal k: $\sqrt{\frac{n(2d-3)\beta}{3\alpha}}$

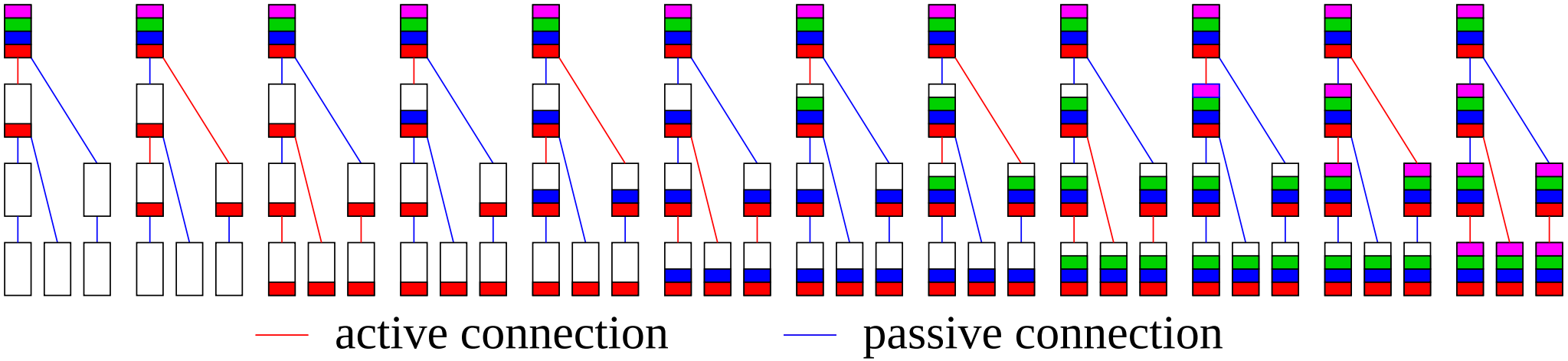
Analysis

- Time $\frac{n}{k}\beta + \alpha$ per step ($\neq$ iteration)
- $d := \lfloor \log p \rfloor$ levels
- Overall: $T(n,p,k) := (2d + 3(k-1))\left(\frac{n}{k}\beta + \alpha\right)$
- optimal k: $\sqrt{\frac{n(2d-3)\beta}{3\alpha}}$

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

Fibonacci-Trees





Analysis

- Time $\frac{n}{k}\beta + \alpha$ per step ($\neq$ iteration)
- j steps until first packet reaches level j
- How many PEs p_j with level $0..j$?

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

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

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

  with $\Phi = \frac{1+\sqrt{5}}{2}$ (golden ratio)

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

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

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**

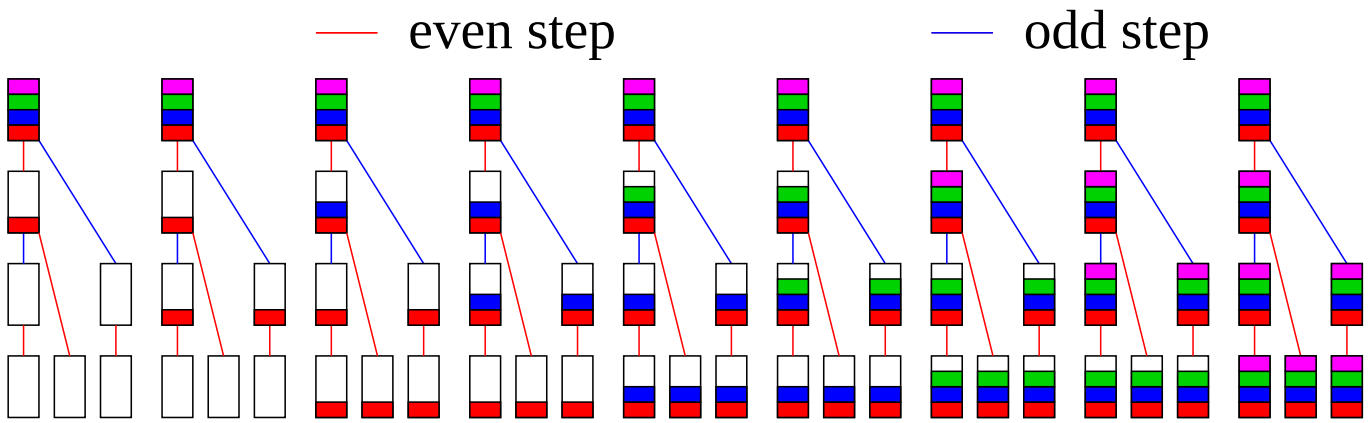
receive piece j from parent // noop for root or $j = k+1$

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

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

send piece j to left child

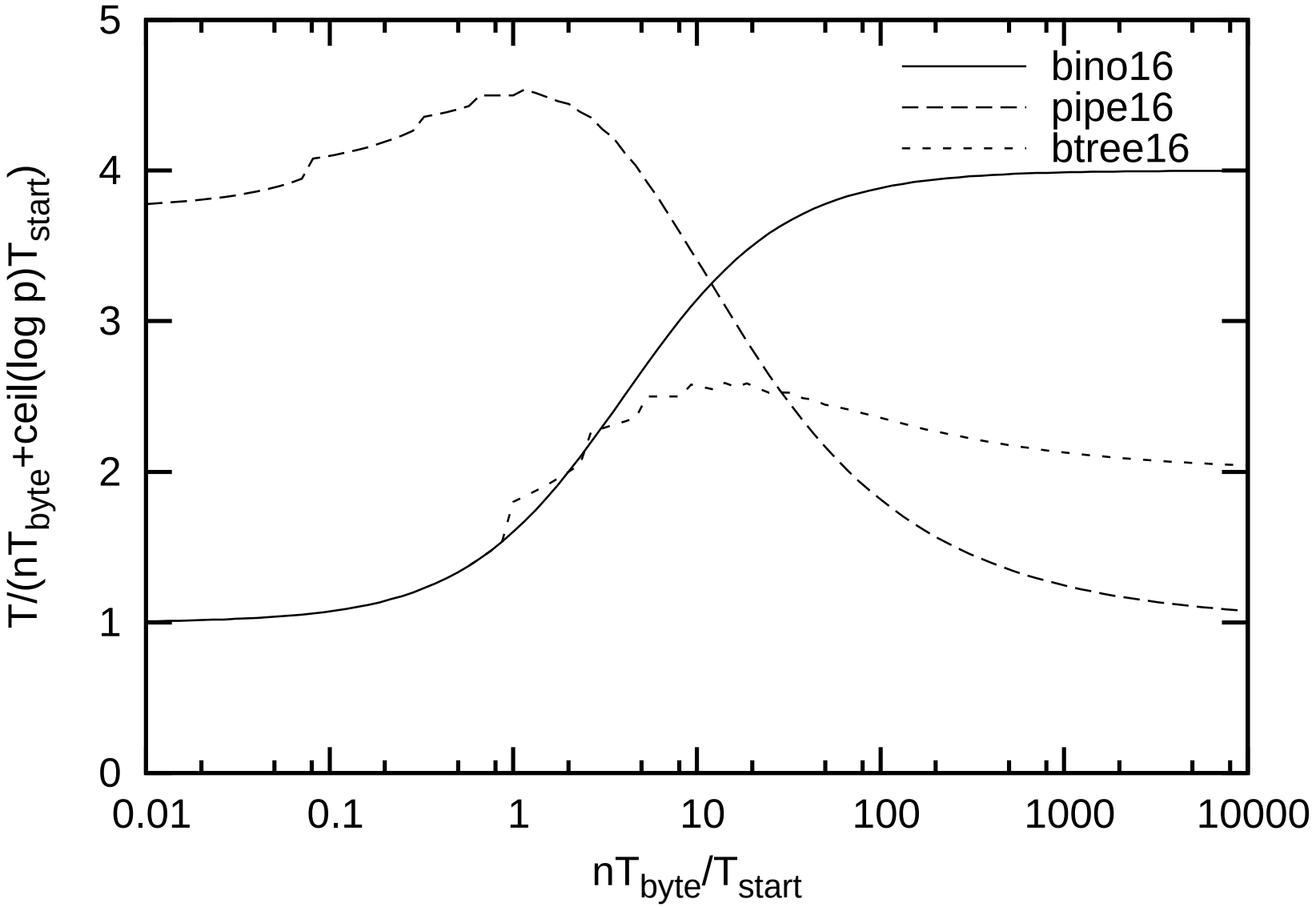
// noop if no such child or $j = k+1$

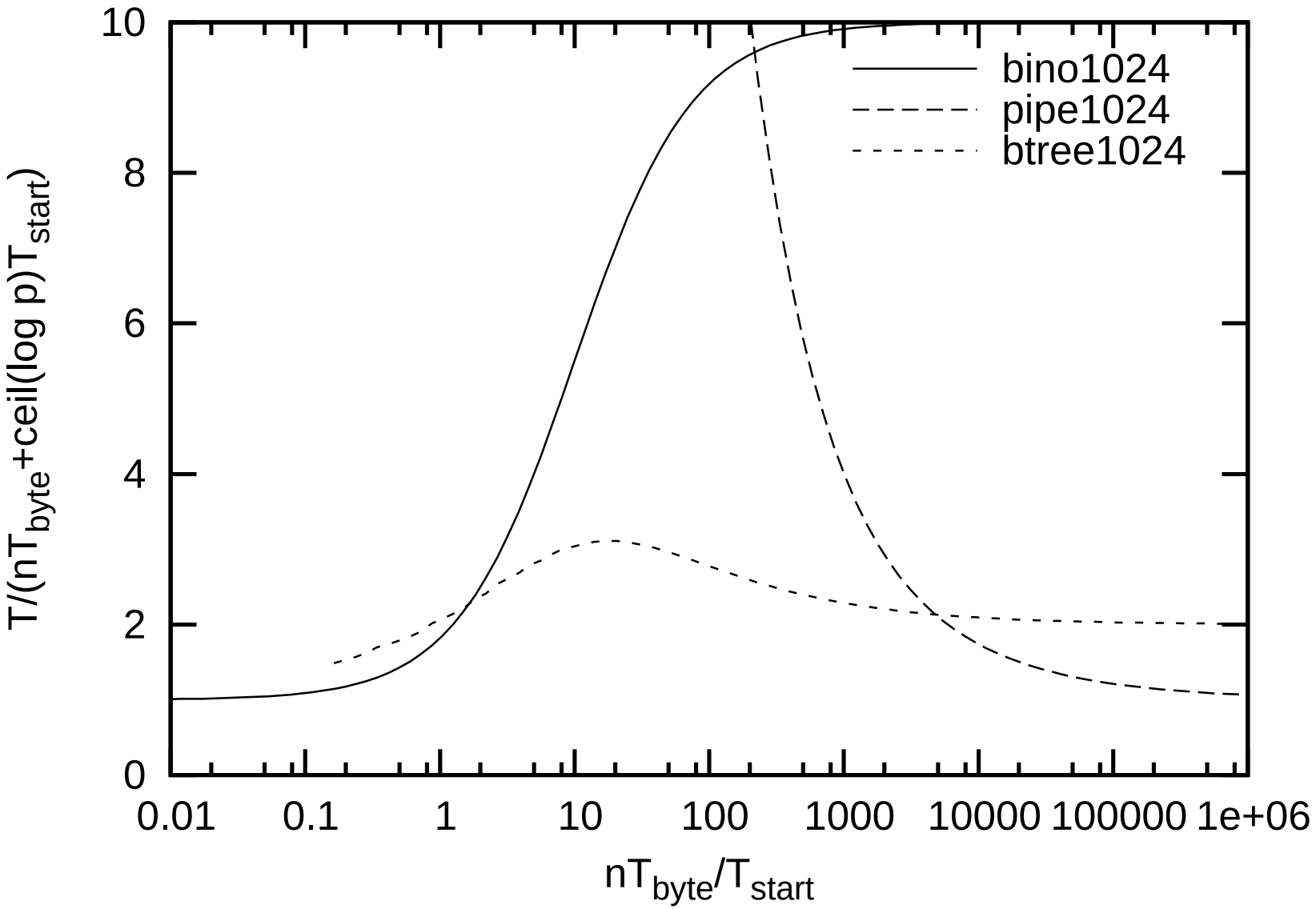


Analysis

- Time $\frac{n}{k}\beta + \alpha$ per step
- j steps until first packet level j reaches
- $d \approx \log_\Phi p$ levels
- Then 2 steps for each further packet

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


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


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

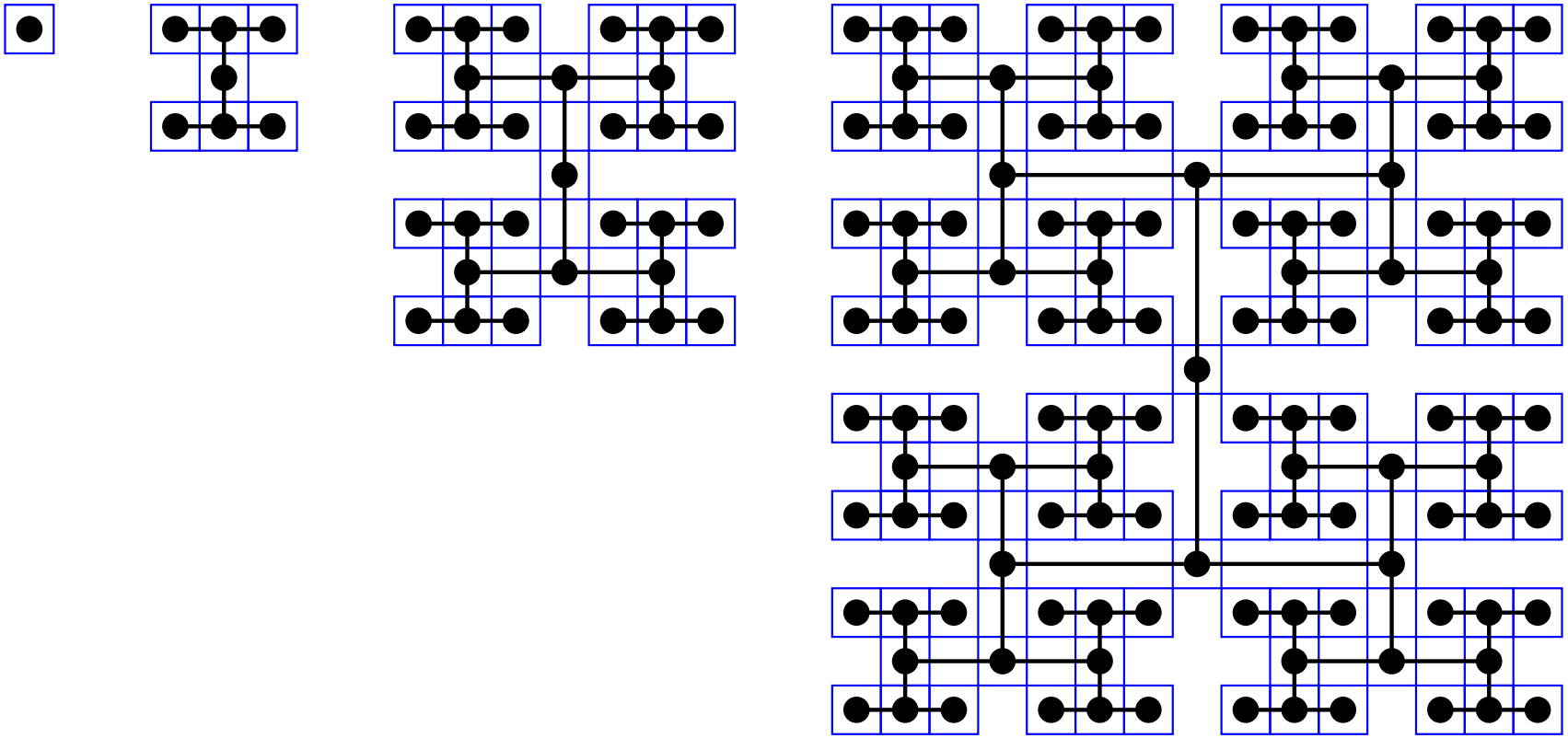
Discussion

Fibonacci trees are a good compromise for all n, p.

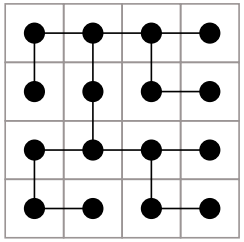
General p:

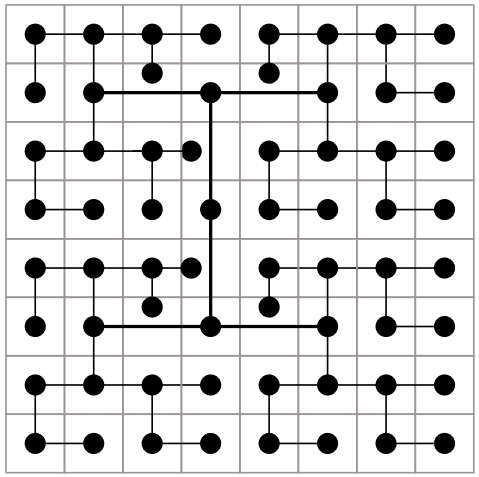
use next larger tree. Then drop a subtree

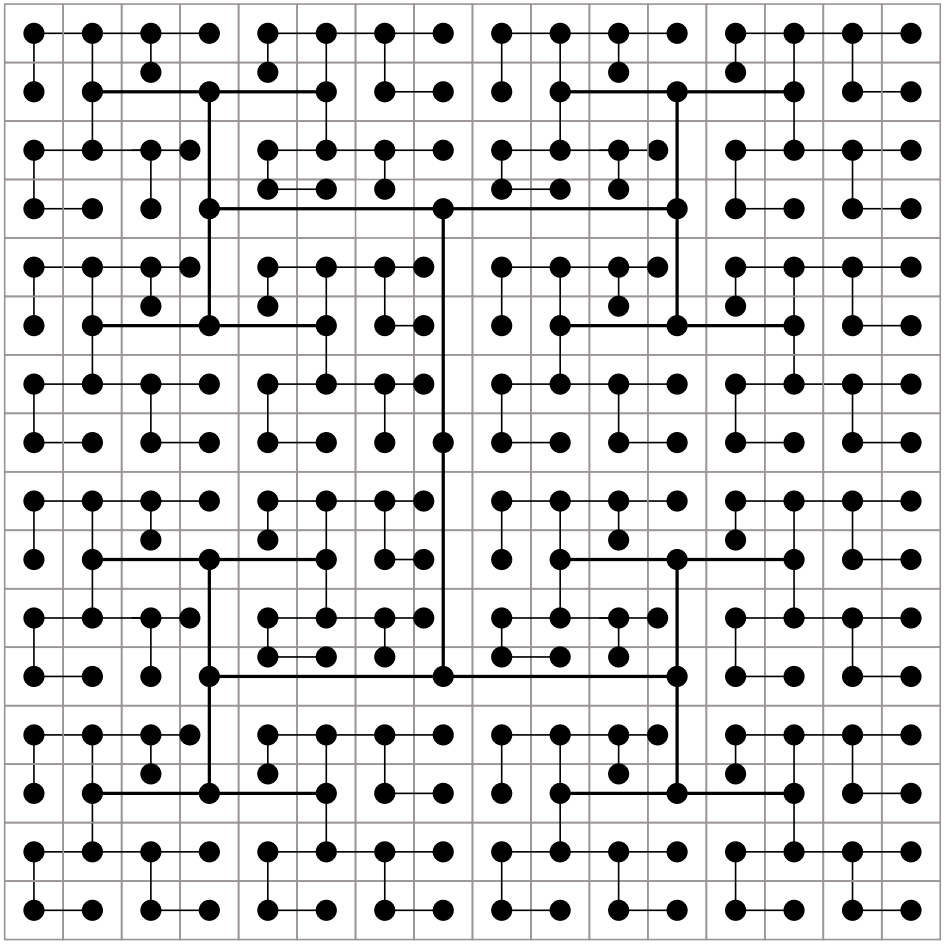
H-Trees



H-Trees







Disadvantages of Tree-based Broadcasts

- Leaves only receive their data. Otherwise they are unproductive
- Inner nodes send more than they receive

  $\rightsquigarrow$ full-duplex communication not fully exploited

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

- Binary-Tree-Broadcasts using two trees A und B at once
- Inner nodes of A are

  leaves of B

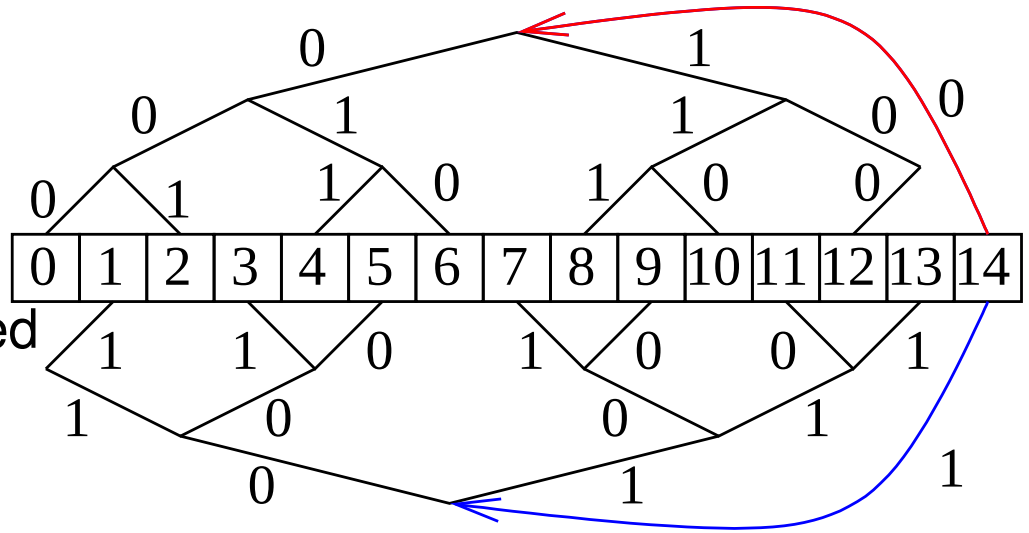
  and vice versa
- Per double step:

  One packet received as a leaf +

  one packet received and forwarded

  as inner node

  i.e., 2 packets sent and received

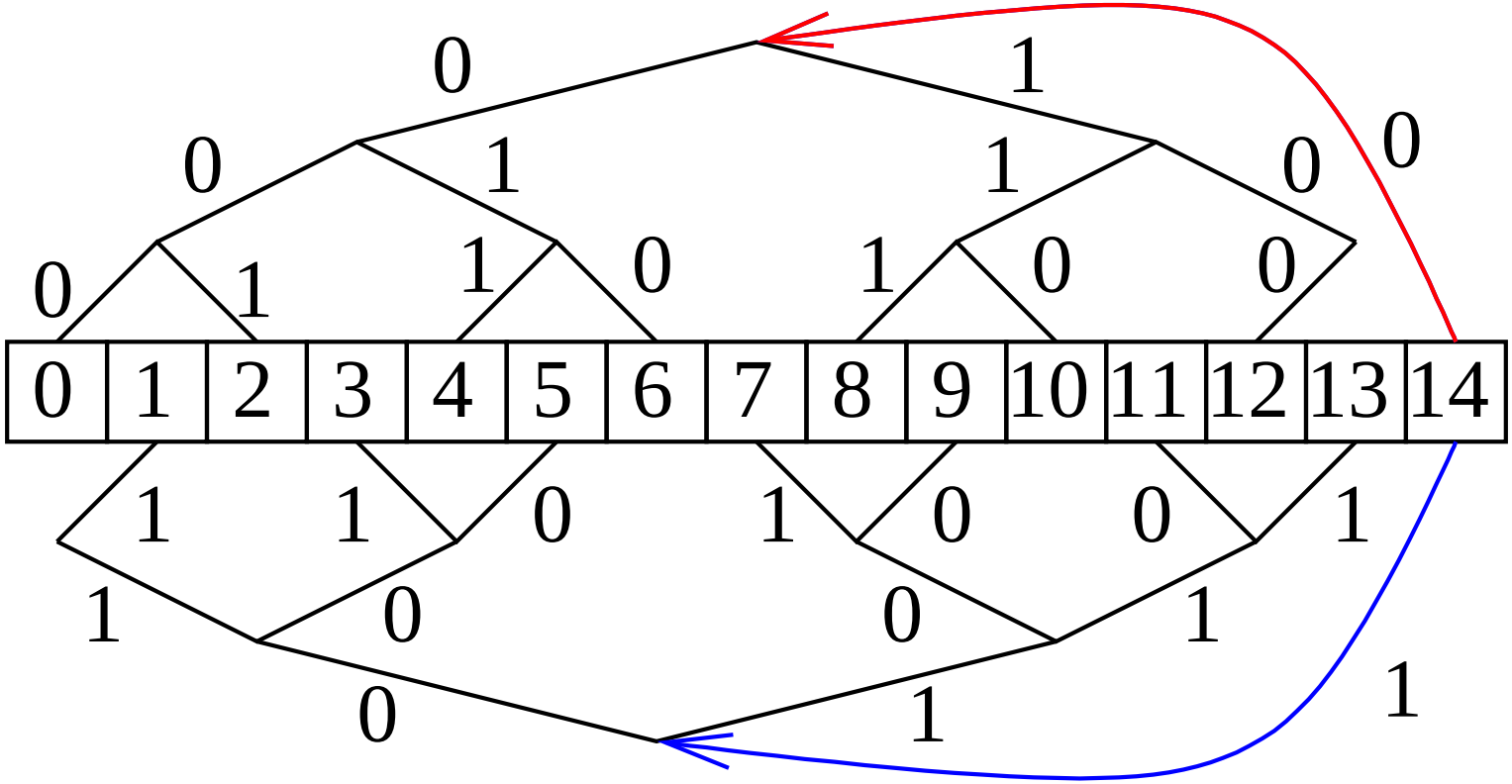


Root Process

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

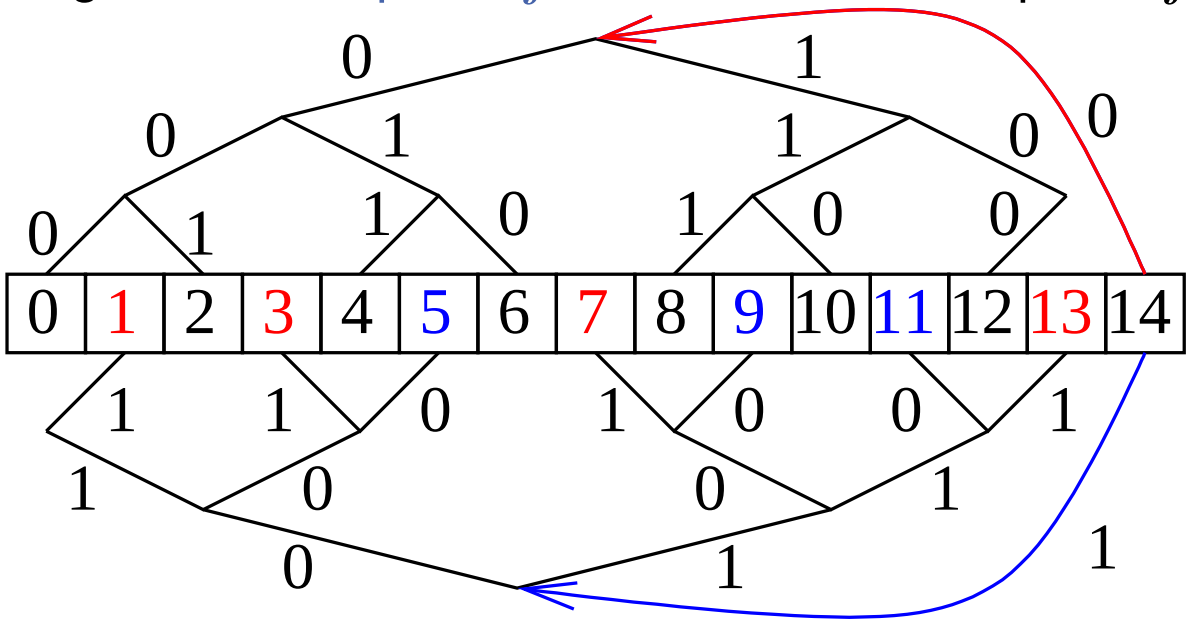
if it comes from the upper tree over an edge labelled b **then**

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

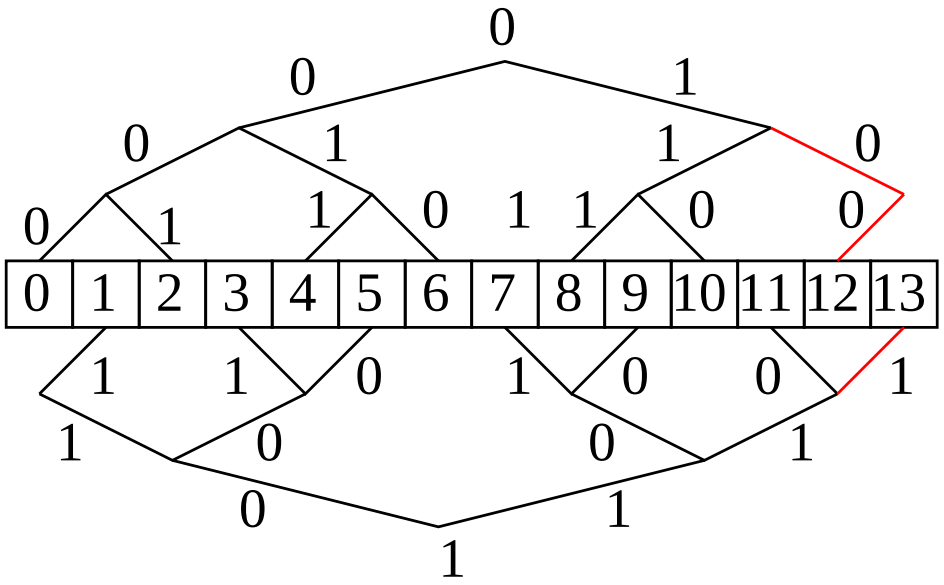
for $j := 1$ **to** $k + \Delta$ **step** 2 **do**

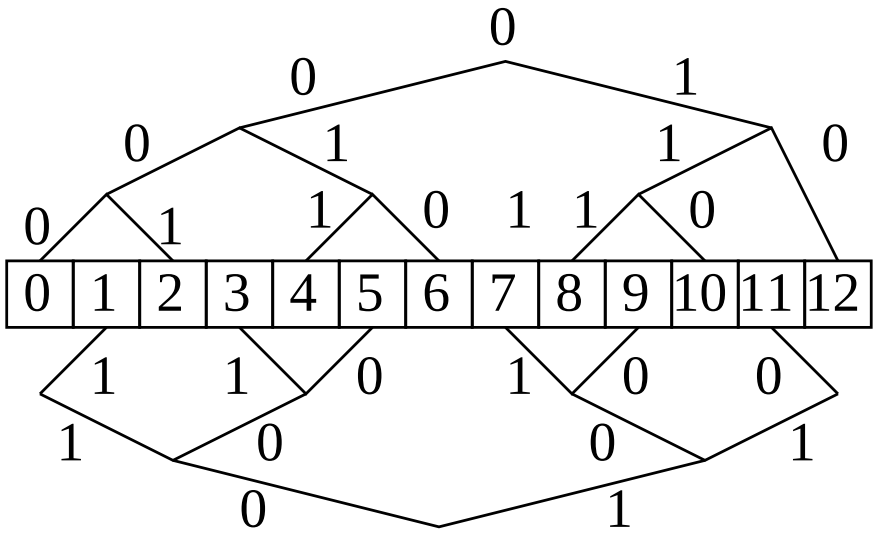
along b-edges: receive piece j and send piece $j-2$

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

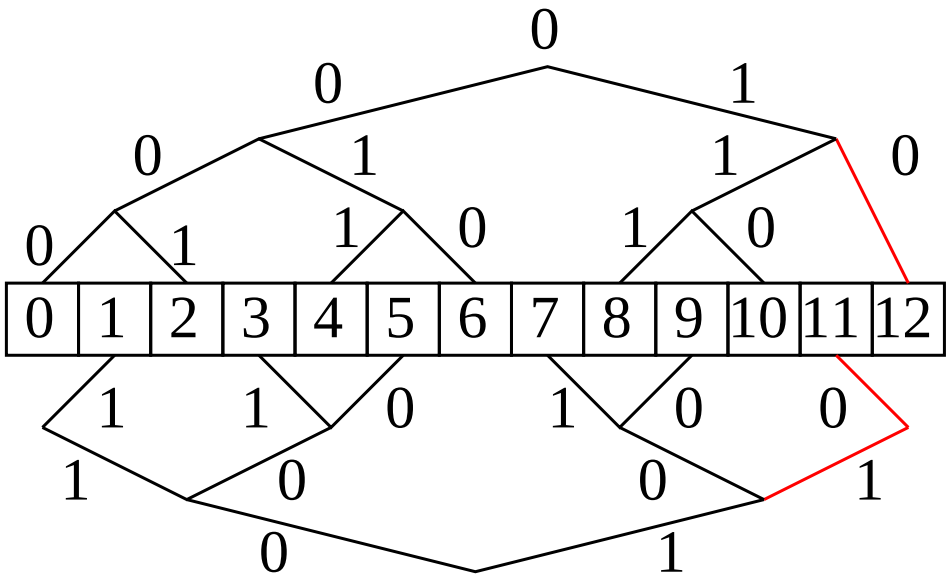


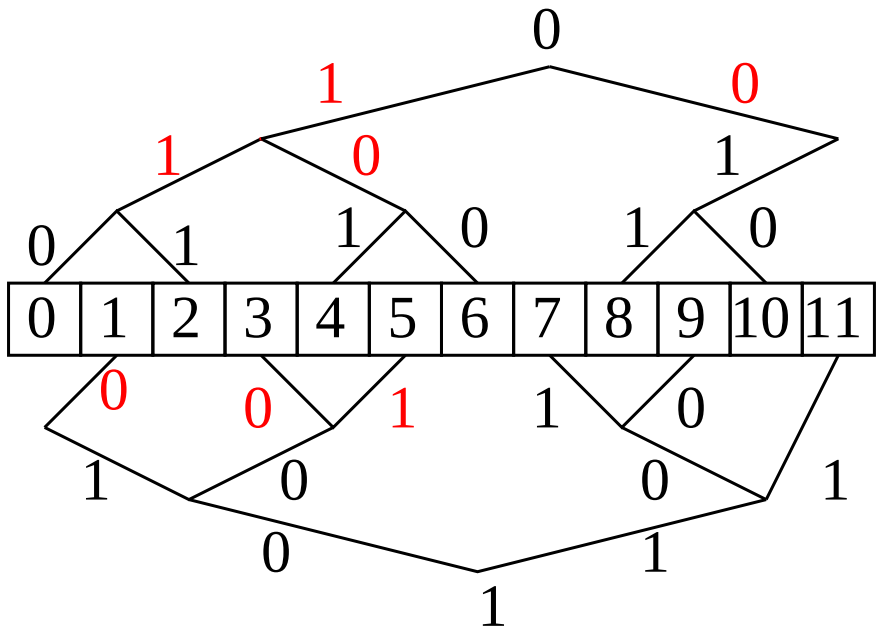
Arbitrary Number of Processors



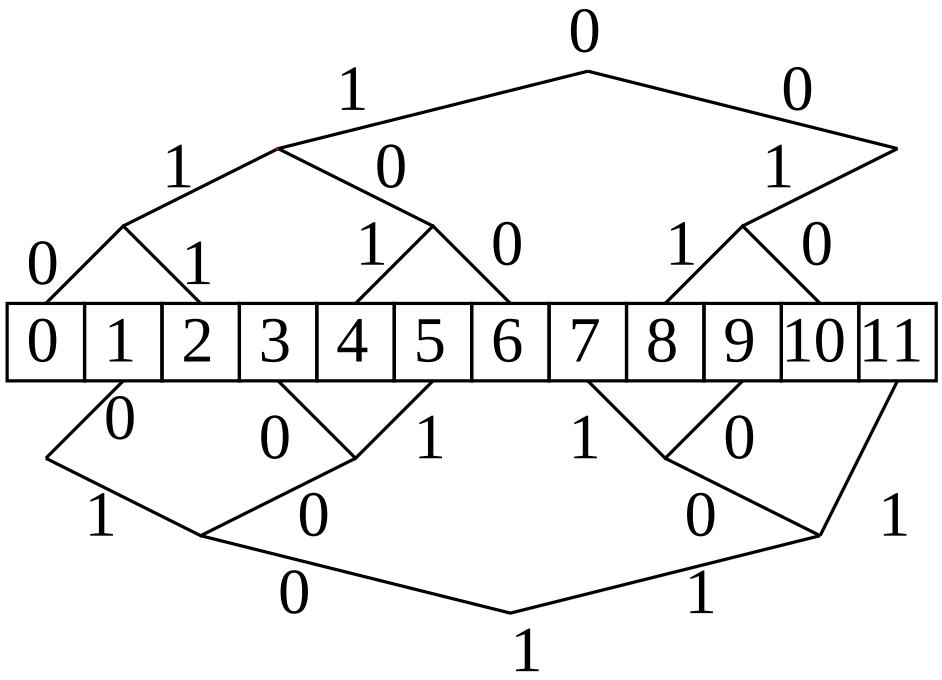


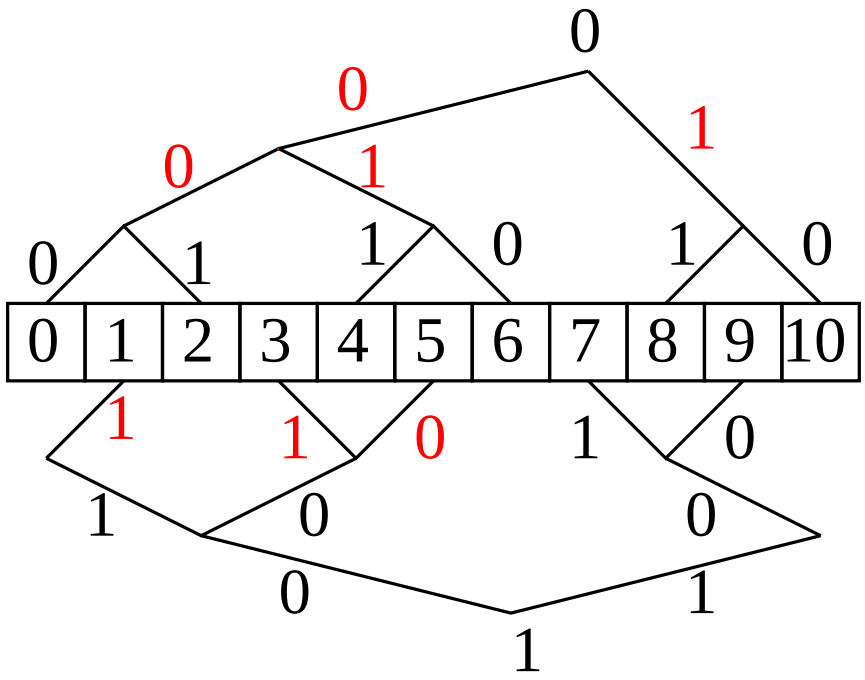
Arbitrary Number of Processors



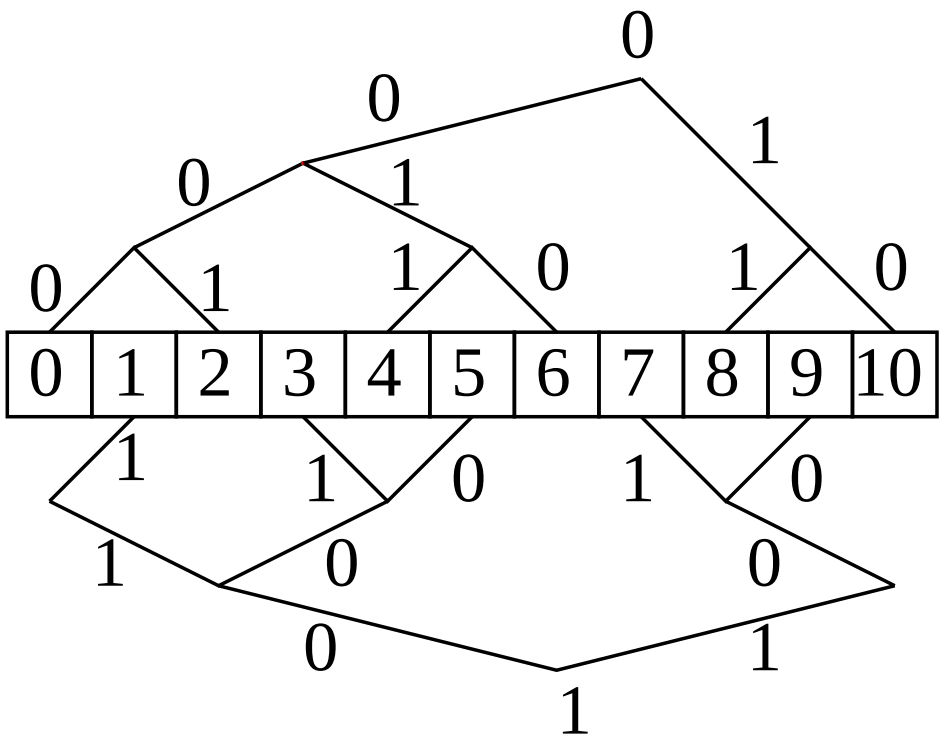


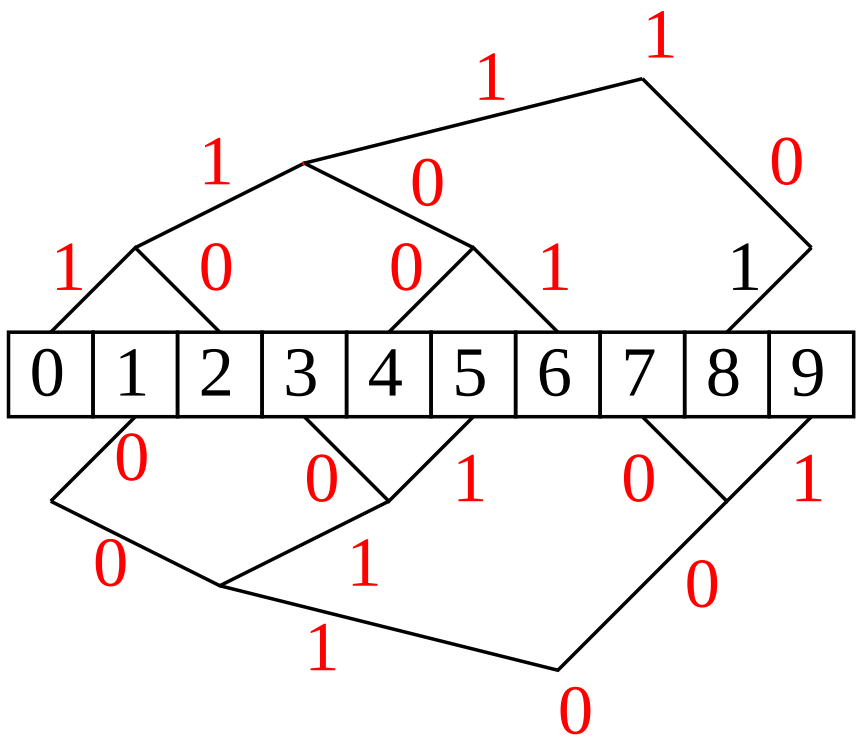
Arbitrary Number of Processors



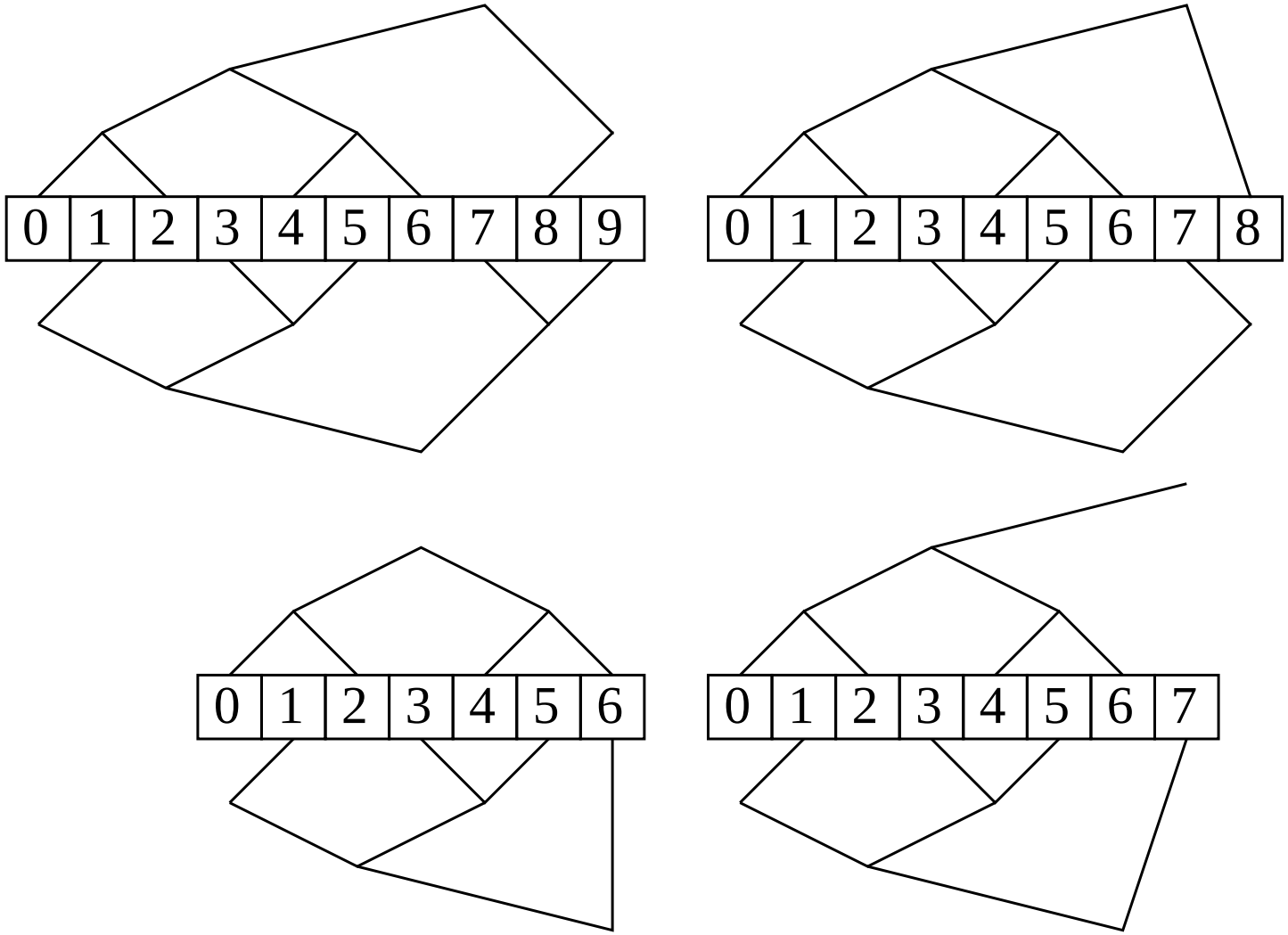


Arbitrary Number of Processors





Arbitrary Number of Processors



Constructing the Trees

Case $p = 2^h - 1$: Upper tree + lower tree + root

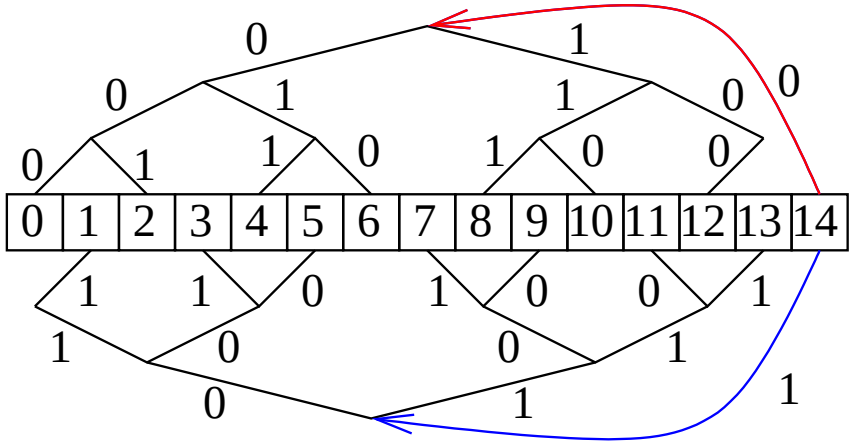
Upper tree: complete binary tree of height $h - 1$, − right leaf

Lower tree: complete binary tree of height $h - 1$, − left leaf

Lower tree ≈ Upper tree shifted by one.

Inner nodes upper tree = leaves of lower tree.

Inner nodes lower tree = leaves of upper tree.



Building Smaller Trees (without root)

invariant : last node has outdegree 1 in tree x

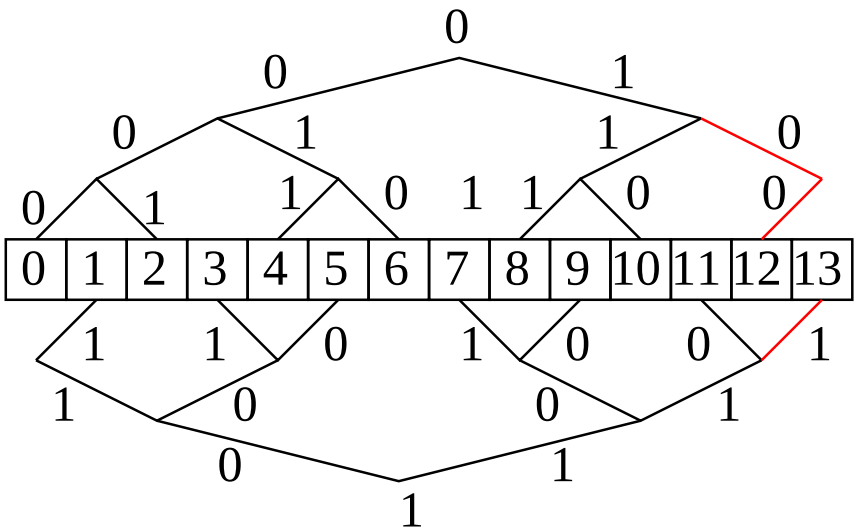
invariant : last node has outdegree 0 in tree $\bar{x}$

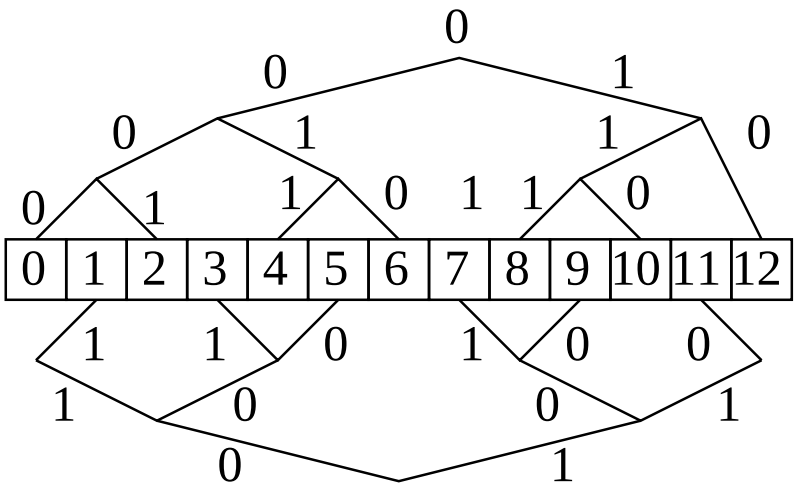
$p \rightsquigarrow p-1$:

Remove last node:

right node in x now has degree 0

right node in $\bar{x}$ now has degree 1





Coloring Edges

Consider bipartite graph:

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

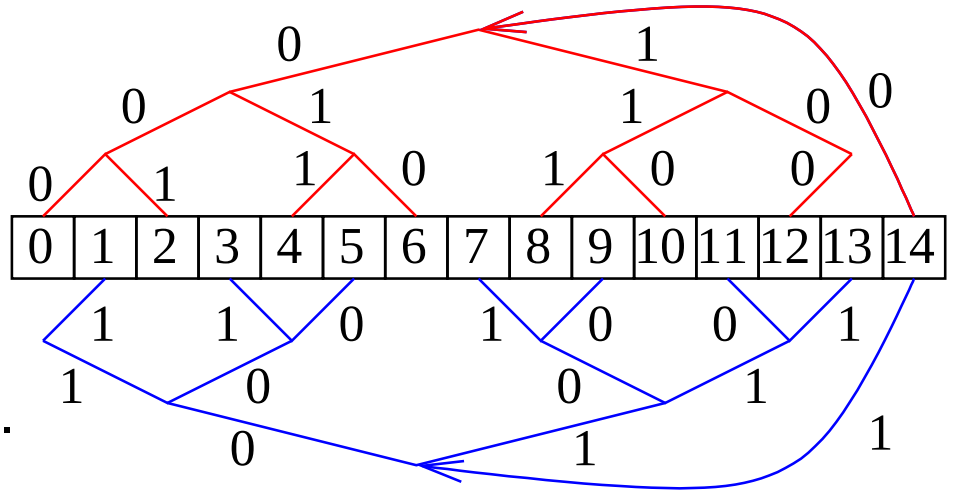
s_i: Sender role of PE i.

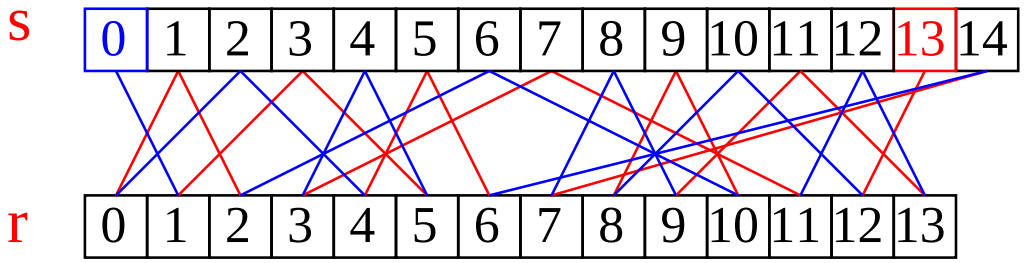
r_i: Receiver role of PE i.

$2\times$ degree 1. all other degree 2.

$\Rightarrow B$ is a path plus even circles.

color edges alternatingly with 0 and 1.

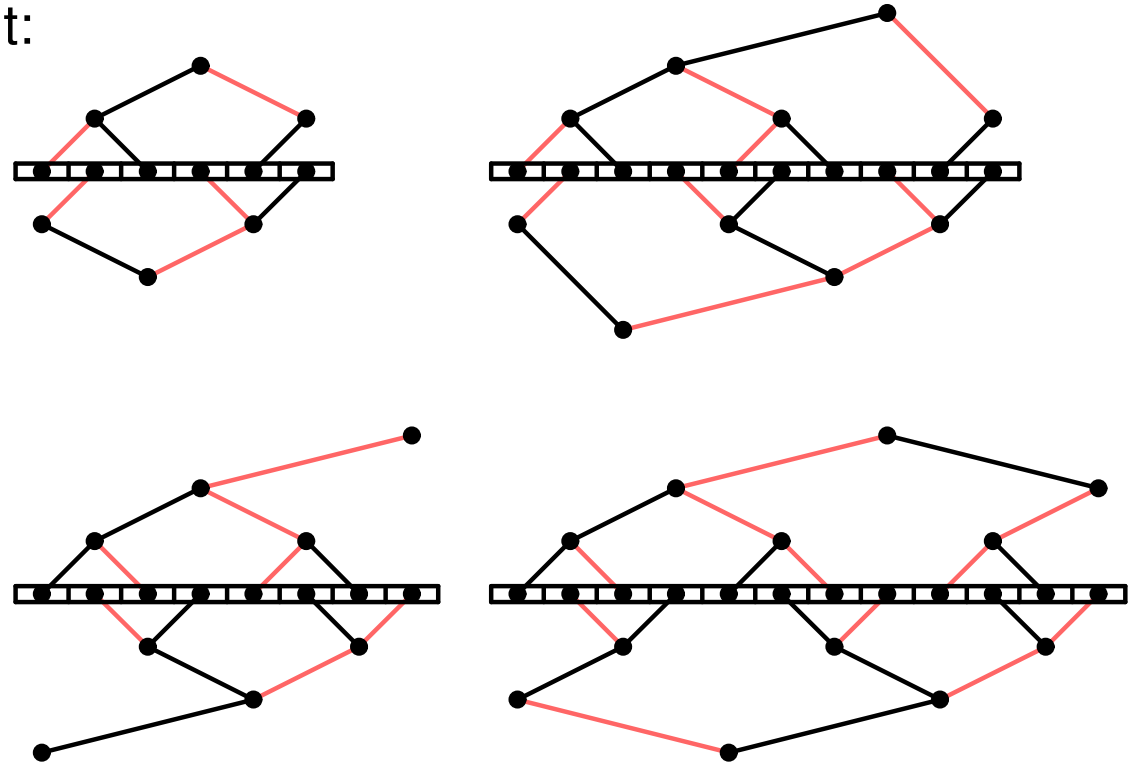




Open Question: Parallel Coloring ?

- In Time Polylog(p) using list ranking.
  (unfortunately impractical for small inputs)
- Fast explicit calculation color(i, p) without communication ?

Mirror layout:



Jochen Speck's Solution

// 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 $[i' > i]$

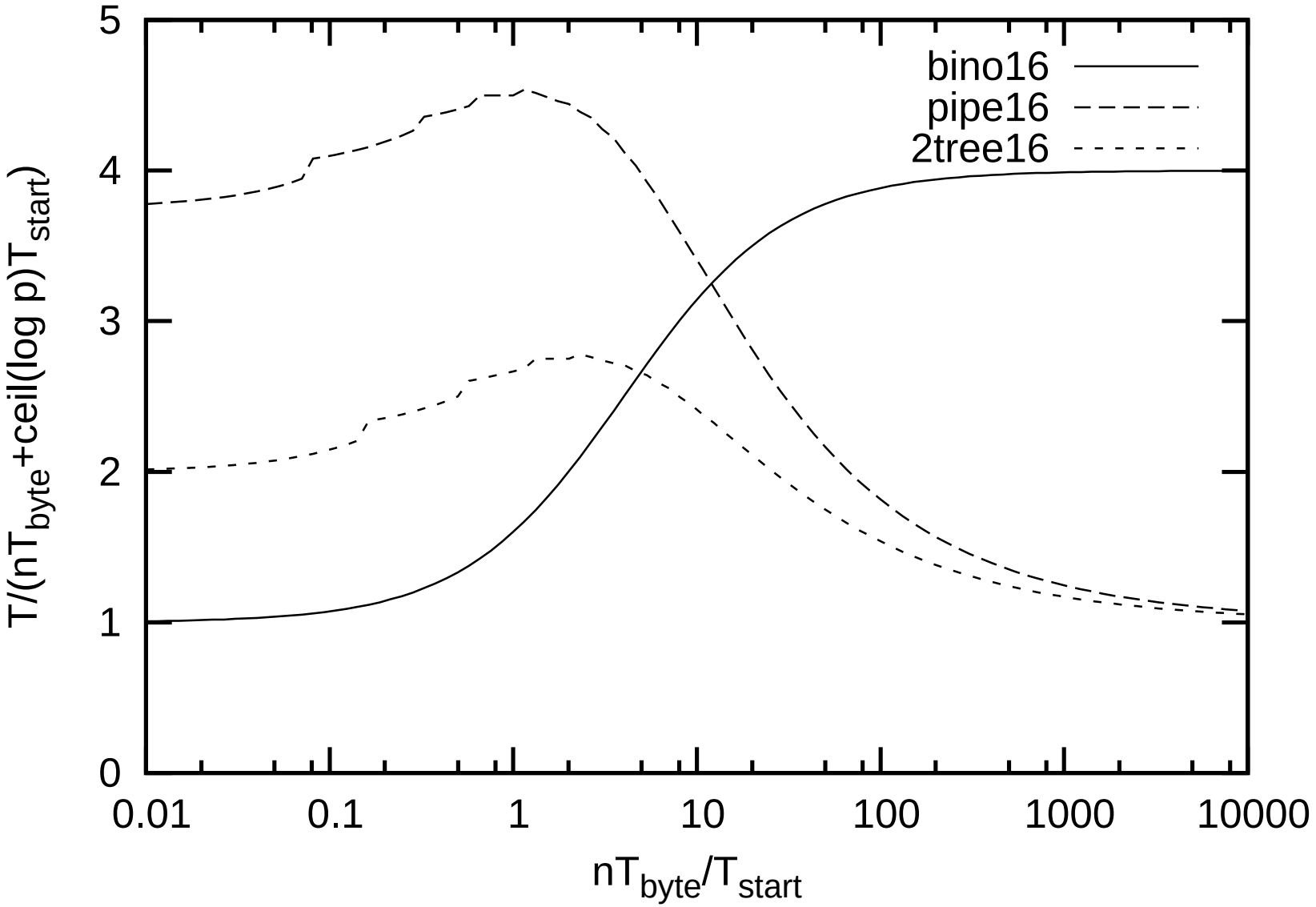
Analysis

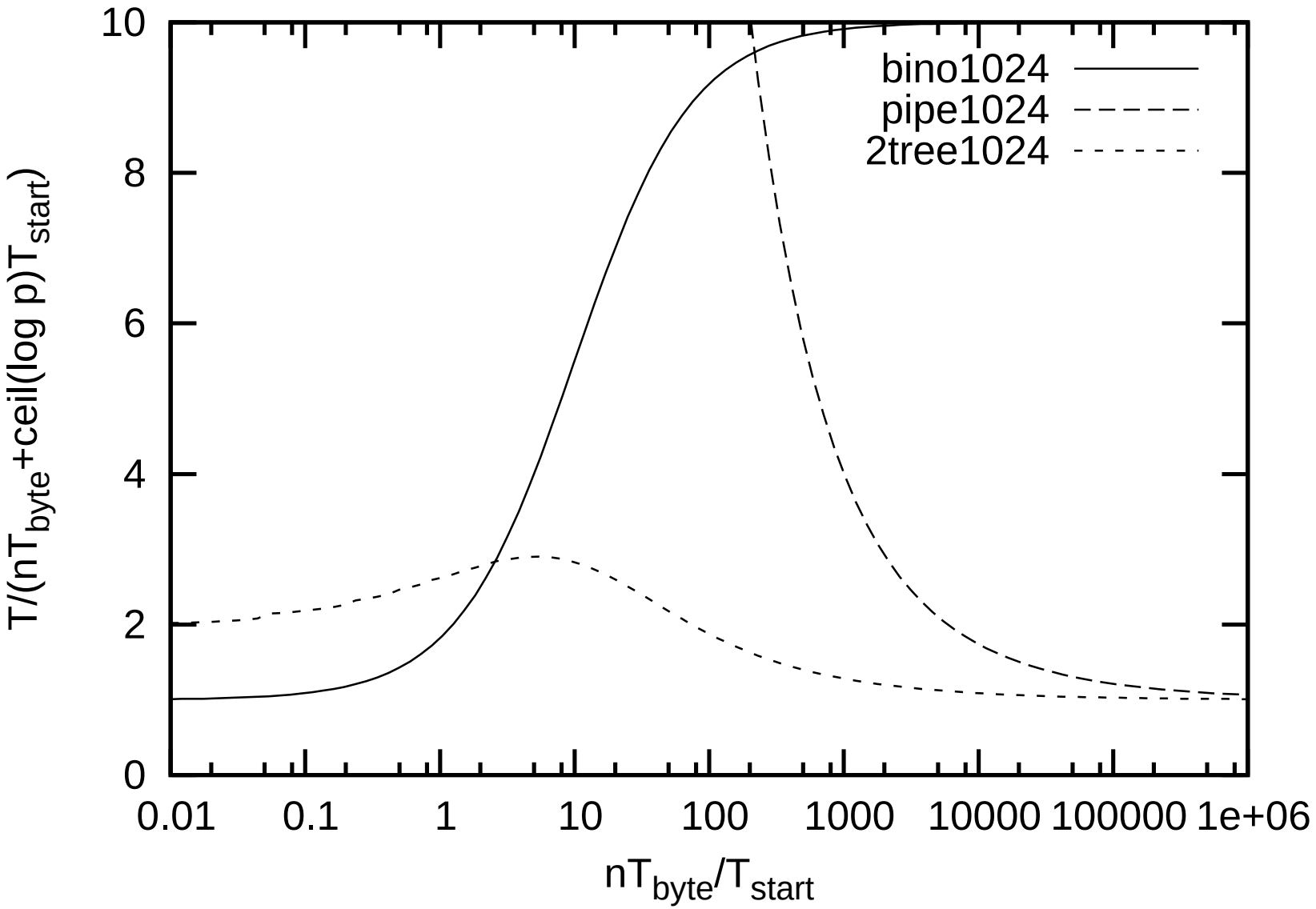
- Time $\frac{n}{k}\beta + \alpha$ pro step
- $2j$ steps until all PEs in level j are reached
- $d = \lceil \log(p+1) \rceil$ levels
- Then 2 steps for 2 further packets

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

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

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

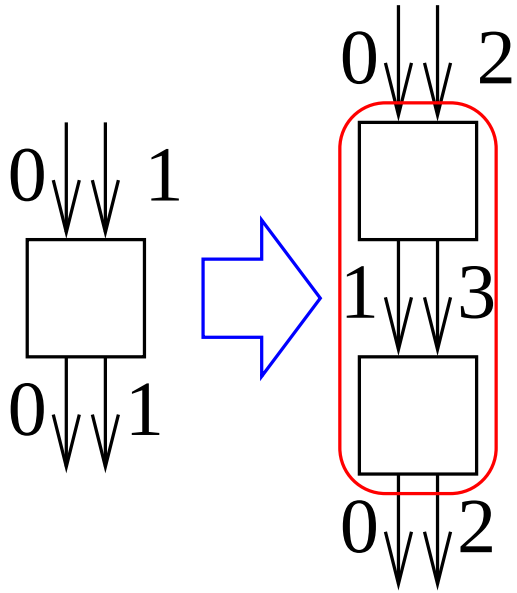

bino16
pipe16
2tree16
T/(nT_byte+ceil(log p)T_start)
nT_byte/T_start
0
1
2
3
4
5
0.01
0.1
1
10
100
1000
10000


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

Implementation in the Simplex-Model

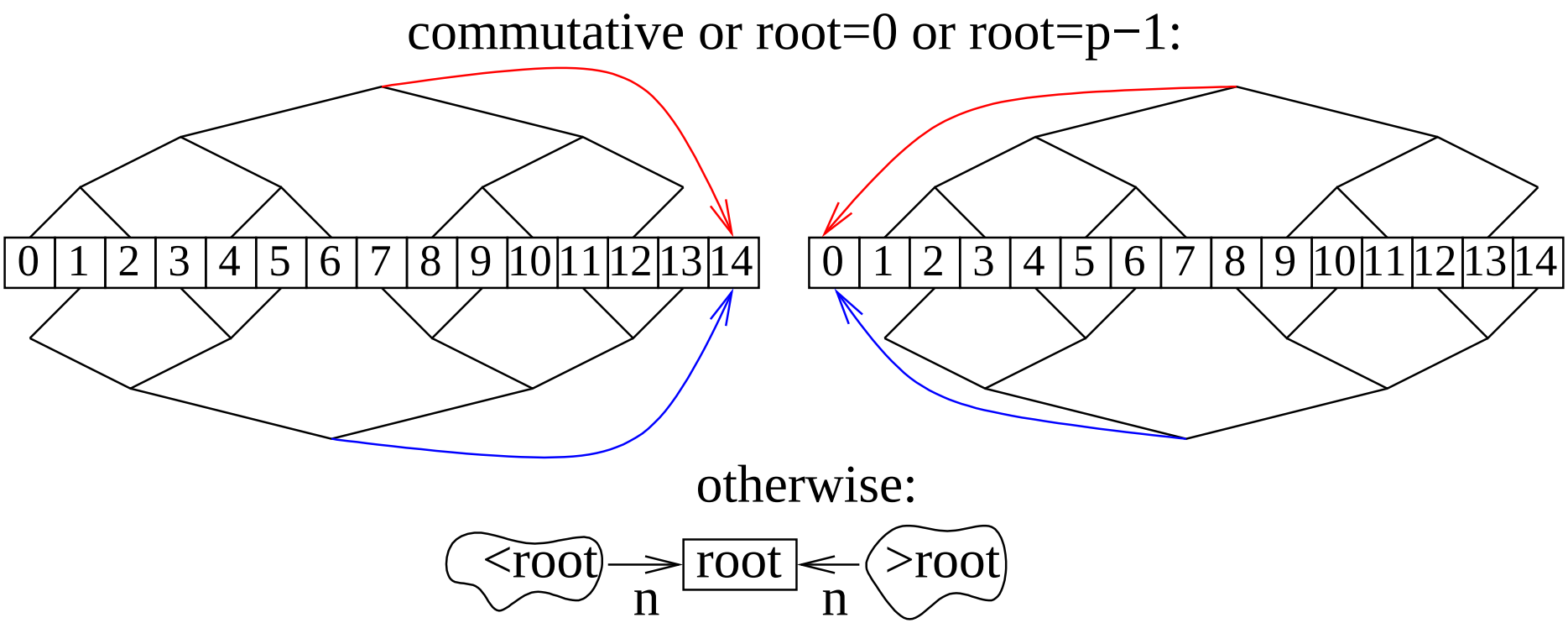
2 steps duplex $\rightsquigarrow$ 4 steps simplex.

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



23-Reduktion

Numbering is an inorder-numbering for both trees !



Another optimal Algorithm

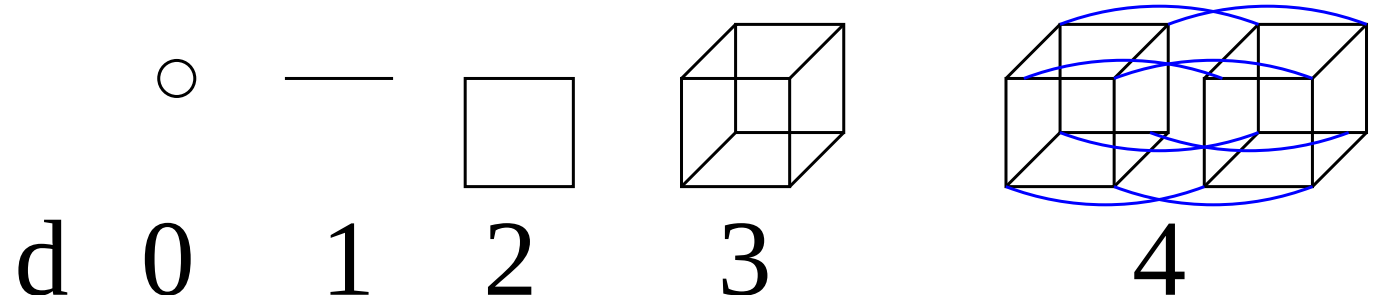
[Johnsson Ho 85: Optimal Broadcasting and Personalized Communication in Hypercube, IEEE Transactions on Computers, vol. 38, no.9, pp. 1249-1268.]

Model: full-duplex limited to a single edge per PE (telephone model)

Adaptation half-duplex: everything $\times 2$

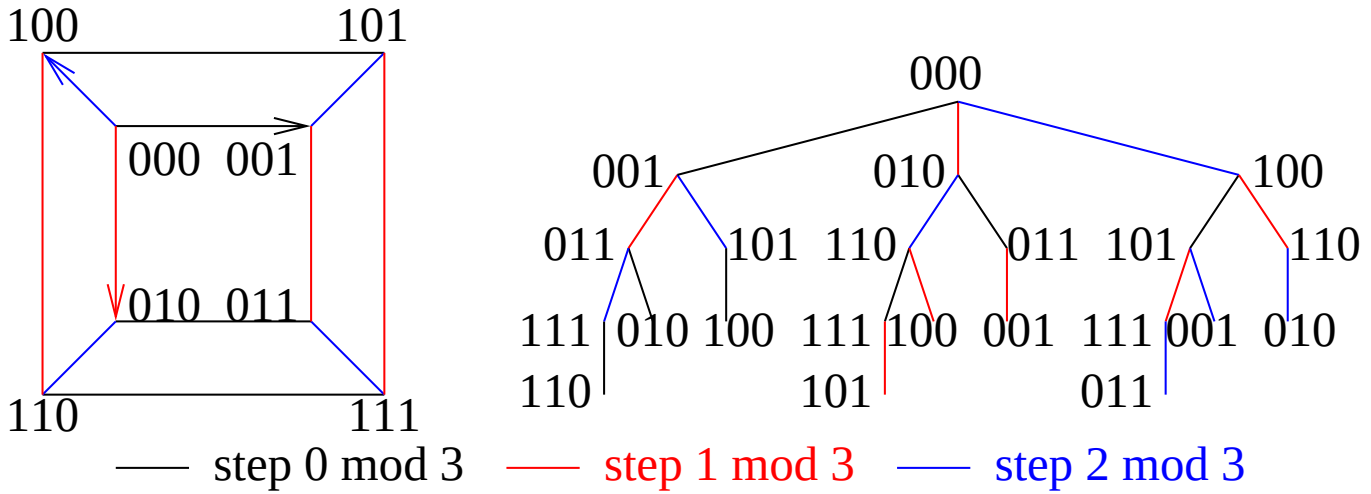
Hypercube H_d

- $p = 2^d$ PEs
- nodes $V = \{0,1\}^d$, i.e., write node number in binary
- edges 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 step i communication along dimension $i \bmod d$
- Decompose H_d into d Edge-disjoint Spanning Binomial Trees
- 0^d cyclically distributes packet to roots of the ESBTs
- ESBT-roots perform binomial tree broadcasting
  (except missing smallest subtree 0^d)



Analysis, Telephone model

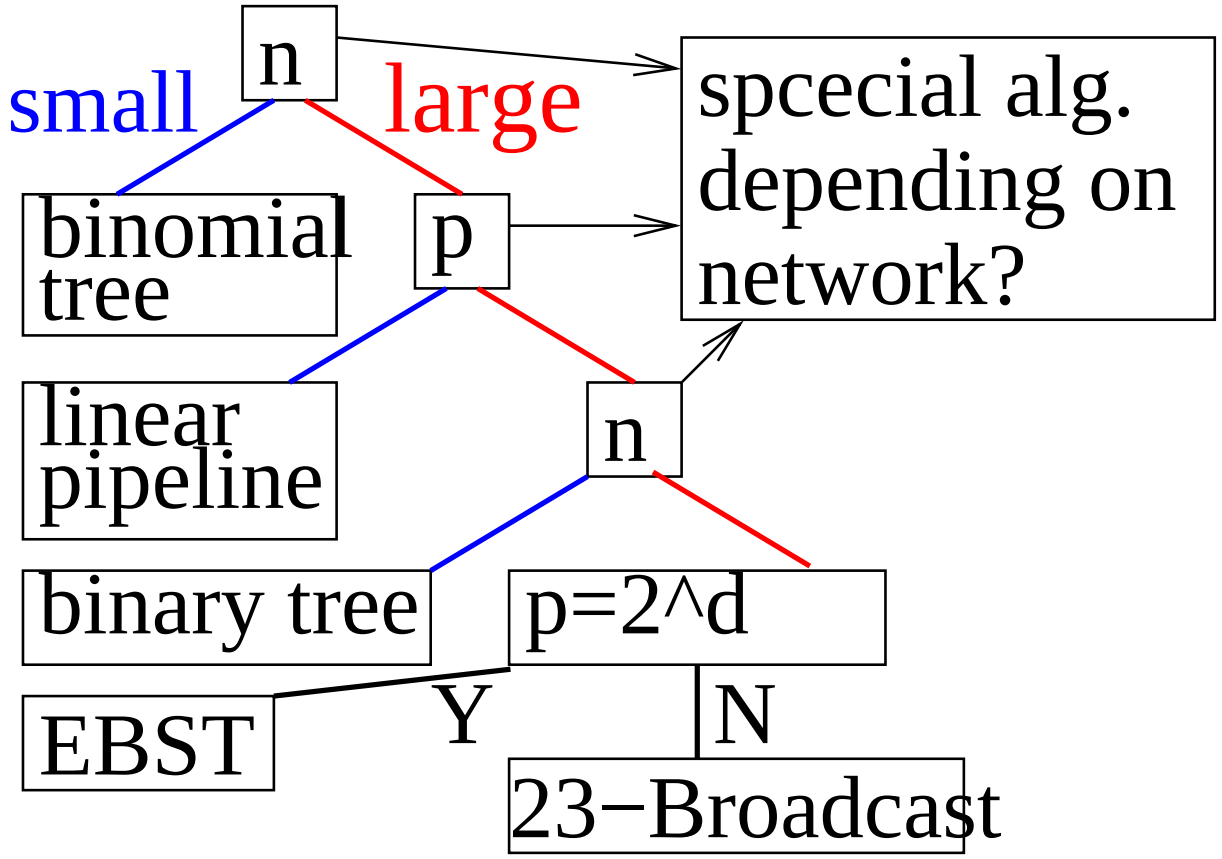
- k packets, k divides n
- k steps until last packet left PE 0
- d steps until it has reached the last leaf
- Overall $d+k$ steps

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

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

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

Discussion


n
small
large
spcecial alg. depending on network?
binomial tree
p
linear pipeline
n
binary tree
p=2^d
EBST
Y
N
23−Broadcast

Reality Check

- Libraries (z.B. MPI) often do not have a pipelined implementation of collective operations $\rightsquigarrow$ your own broadcast may be significantly faster than a library routine
- choosing k is more complicated: only piece-wise linear cost function for point-to-point communication, rounding
- Hypercube gets slow when communication latencies have large variance
- perhaps modify Fibonacci-tree etc. in case of asynchronous communication (Sender finishes before receiver). Data should reach all leaves about at the same time.

Broadcast for Library Authors

- **One** Implementation? $\rightsquigarrow$ 23-Broadcast
- Few, simple Variants? $\{$binomial tree, 23-Broadcast$\}$ or $\{$binomial tree, 23-Broadcast, linear pipeline$\}$

Beyond Broadcast

- Pipelining is important technique for handling large data sets
- hyper-cube algorithms are often elegant and efficient. (and often simpler than ESBT)
- Parameter tuning (e.g. for. k) is often important.

Sorting

- Fast inefficient ranking
- Quicksort
- Sample Sort
- Multiway Mergesort
- Selection
- More on sorting

Fast Inefficient Ranking

n Elements, n^2 processors:

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

Output: $M[1..m]$ // $M[i]$ =rank of $A[i]$

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

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

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

Sorting Larger Data Sets

- n input elements. Initially n/p per PE
- Perhaps more general initial state
- Output is globally sorted

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

$$\Downarrow \pi$$

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

- Comparison based model
- $T_{\text{seq}} = T_{\text{compr}} n \log n + \mathcal{O}(n)$

Careful: different notation in Skript $n \leftrightarrow n/p$

Back to Fast 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$)$

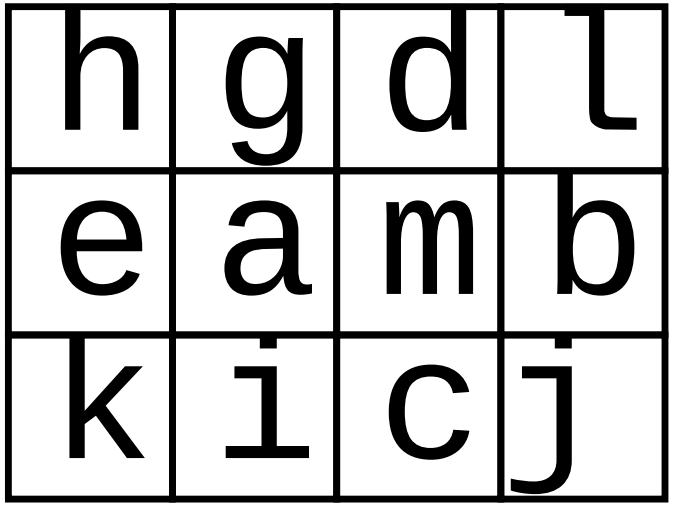
ranks:= $\langle |\{x \in c : x \leq y\}| : y \in r \rangle$ // merge

reduce-by-rows(ranks)

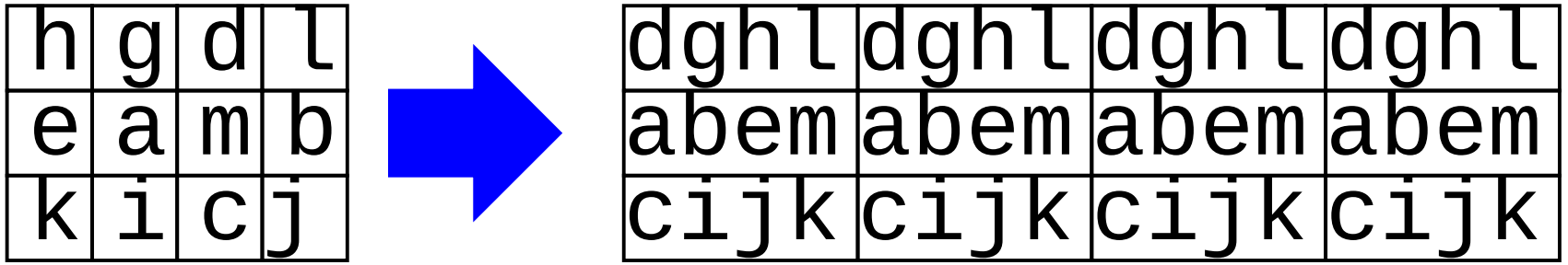
Time

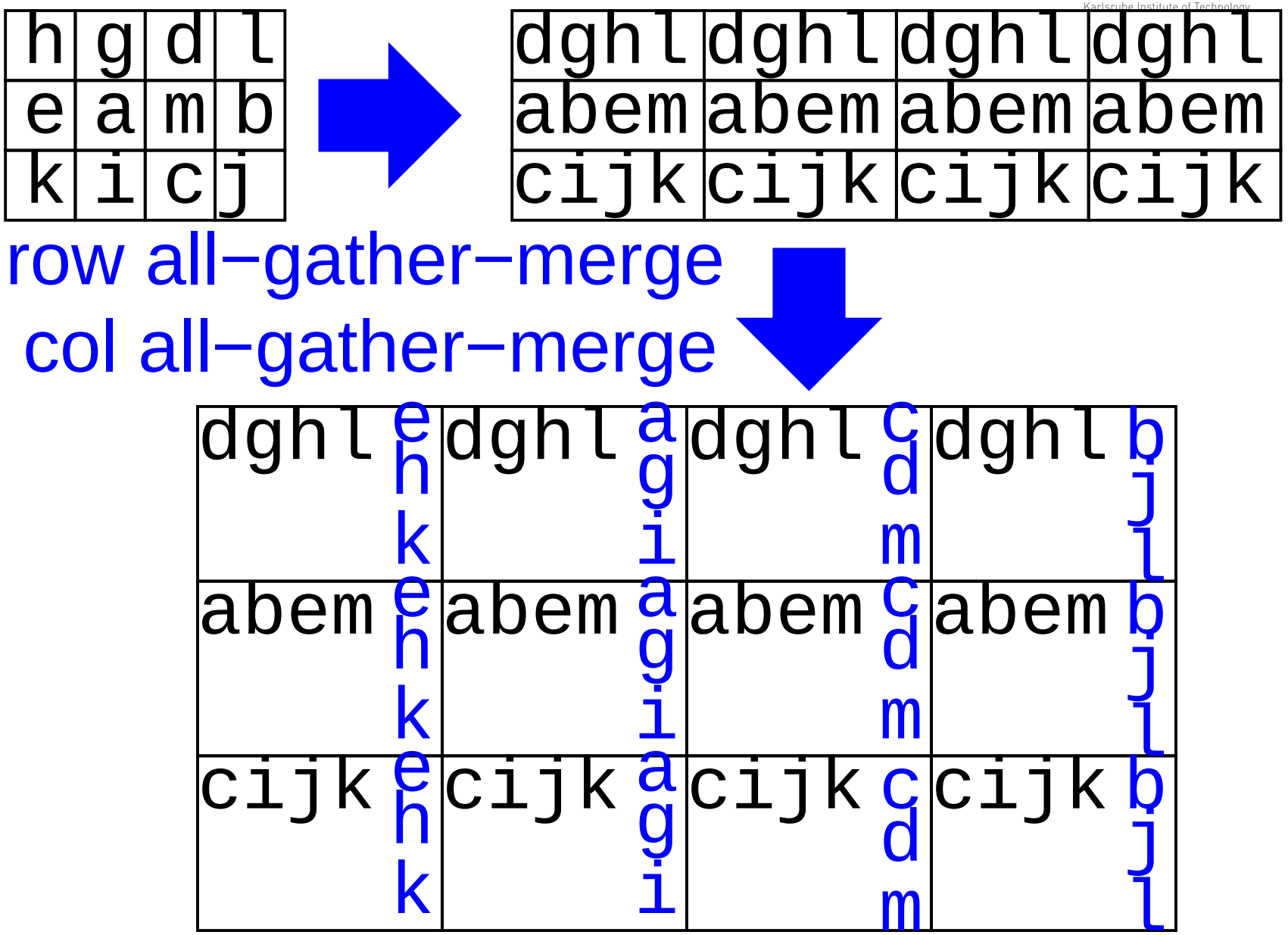
$$\mathcal{O}\left(\alpha \log p + \beta \frac{n}{\sqrt{p}} + \frac{n}{p} \log \frac{n}{p}\right) \quad . \tag{1}$$

Example



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 0123 ehk	dghl 1223 agi	dghl 2222 cdm	dghl 1113 bjl
abem 0013 ehk	abem 1113 agi	abem 0023 cdm	abem 0113 bjl
cijk 0223 ehk	cijk 1333 agi	cijk 1222 cdm	cijk 1122 bjl

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

More Accurate Analysis

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 EDSBT-Algorithm:

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

Overall:

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

Numerical Example:

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

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

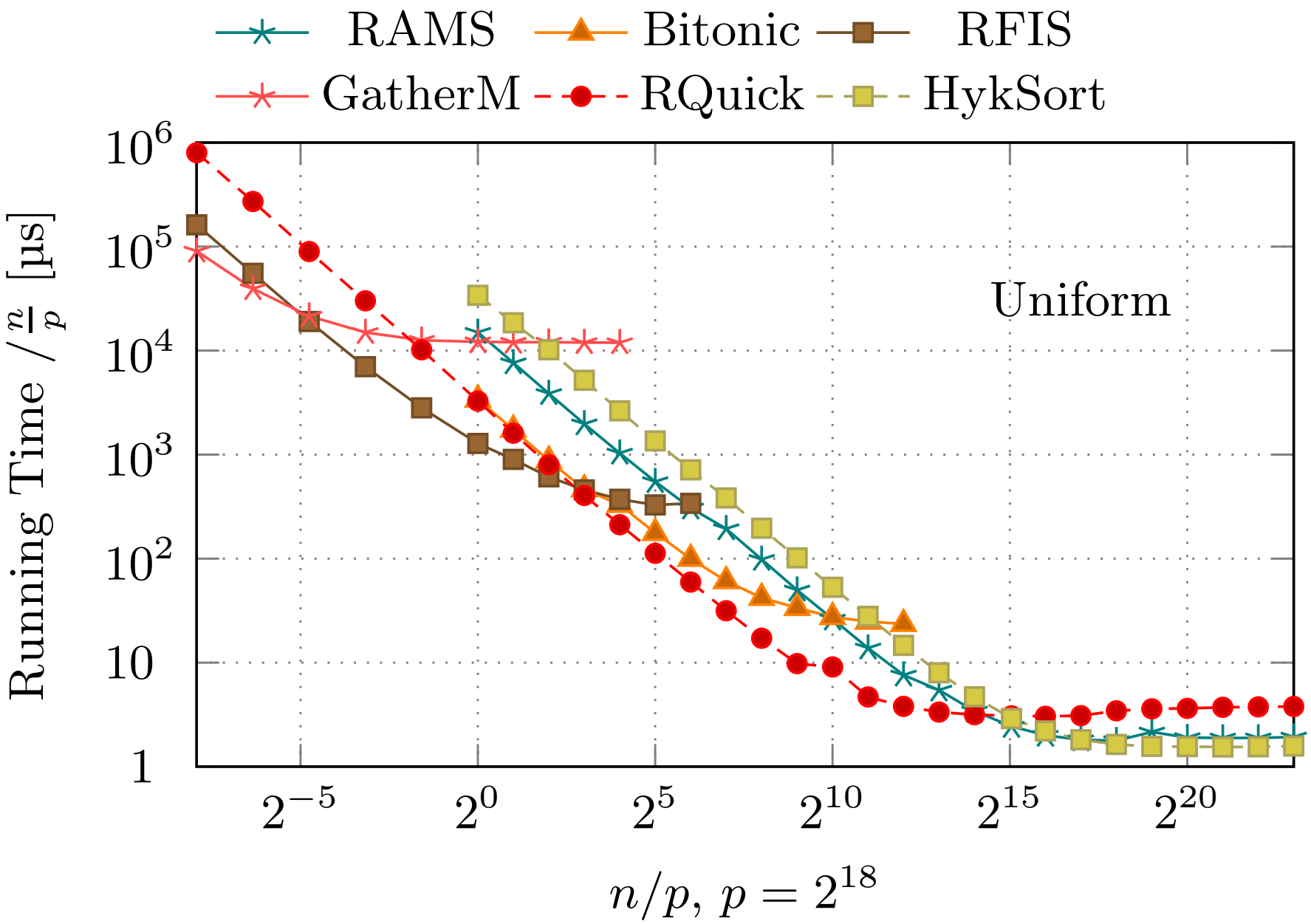
Time≈ 0.200ms.

In comparison: efficient Gather+seq. sort:

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

even larger difference with naive gather

Mearements Axtmann Sanders ALENEX 2017



Quicksort

Sequential

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

Parallelization for Beginners

Parallelization of recursive calls.

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

- Very limited speedup
- Bad for distributed memory

Parallelization for Theoreticians

For simplicity: $n = p$.

Idea: Also parallelize partitioning

1. One PE provides the pivot (e.g. random choice).
2. Broadcast
3. Local comparison
4. Enumerate „small“ elements (prefix-sum)
5. Redistribute data
6. Split PEs
7. Parallel recursion

Parallelization for Theoreticians

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

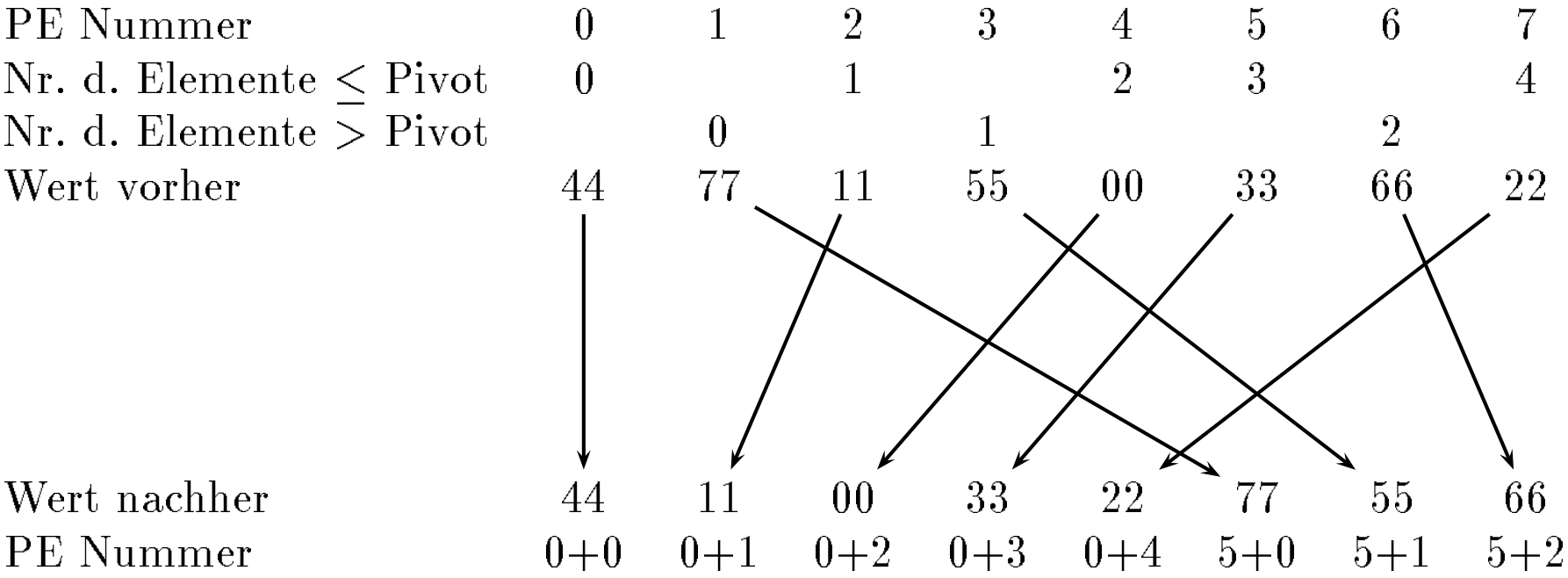
receive d

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

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

Example

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);
}
```

Analysis

- Per recursion level:
  - $2\times$ Broadcast
  - $1\times$ Prefix sum ($\rightarrow$later)

  $\rightsquigarrow$ Time $\mathcal{O}(\alpha \log p)$
- Expected recursion depth: $\mathcal{O}(\log p)$

  ($\rightarrow$ lecture randomized algorithms)

Expected overall time: $\mathcal{O}(\alpha \log^2 p)$

Generalization for $n \gg p$ by Standard Procedure?

- Each PE has, in general, „large“ und „small“ Elements.
- These numbers are not multiples of n/p
- Prefix sums remain useful
- On PRAMs we get a $\mathcal{O}\left(\frac{n \log n}{p} + \log^2 p\right)$ algorithm
- For distributed memory its bad that many elements get moved $\Omega(\log p)$ times.

  $\rightsquigarrow \cdots \rightsquigarrow$ Time $\mathcal{O}\left(\frac{n}{p}(\log n + \beta \log p) + \alpha \log^2 p\right)$

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)

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

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

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

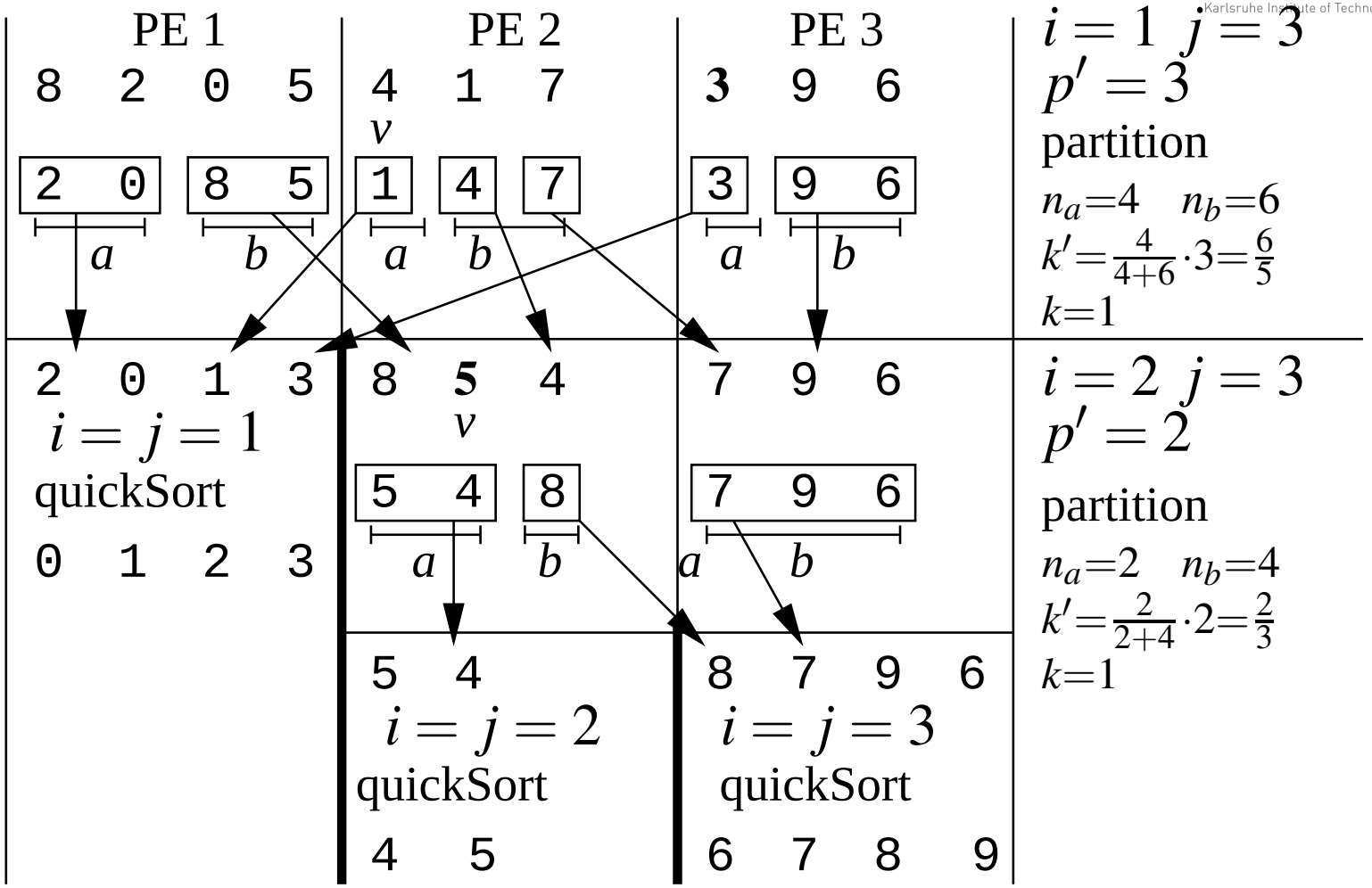
choose $k \in \{\lfloor k' \rfloor, \lceil k' \rceil\}$ such that $\max\left\{\left\lceil \frac{n_a}{k} \right\rceil, \left\lceil \frac{n_b}{p'-k} \right\rceil\right\}$ is minimized

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

Load Balance

Simplified scenario: splitting always with ratio 1:2

larger subproblem gets one PE-load too much.

Imbalance-factor:

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

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

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

The good News:

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

Better Balance?

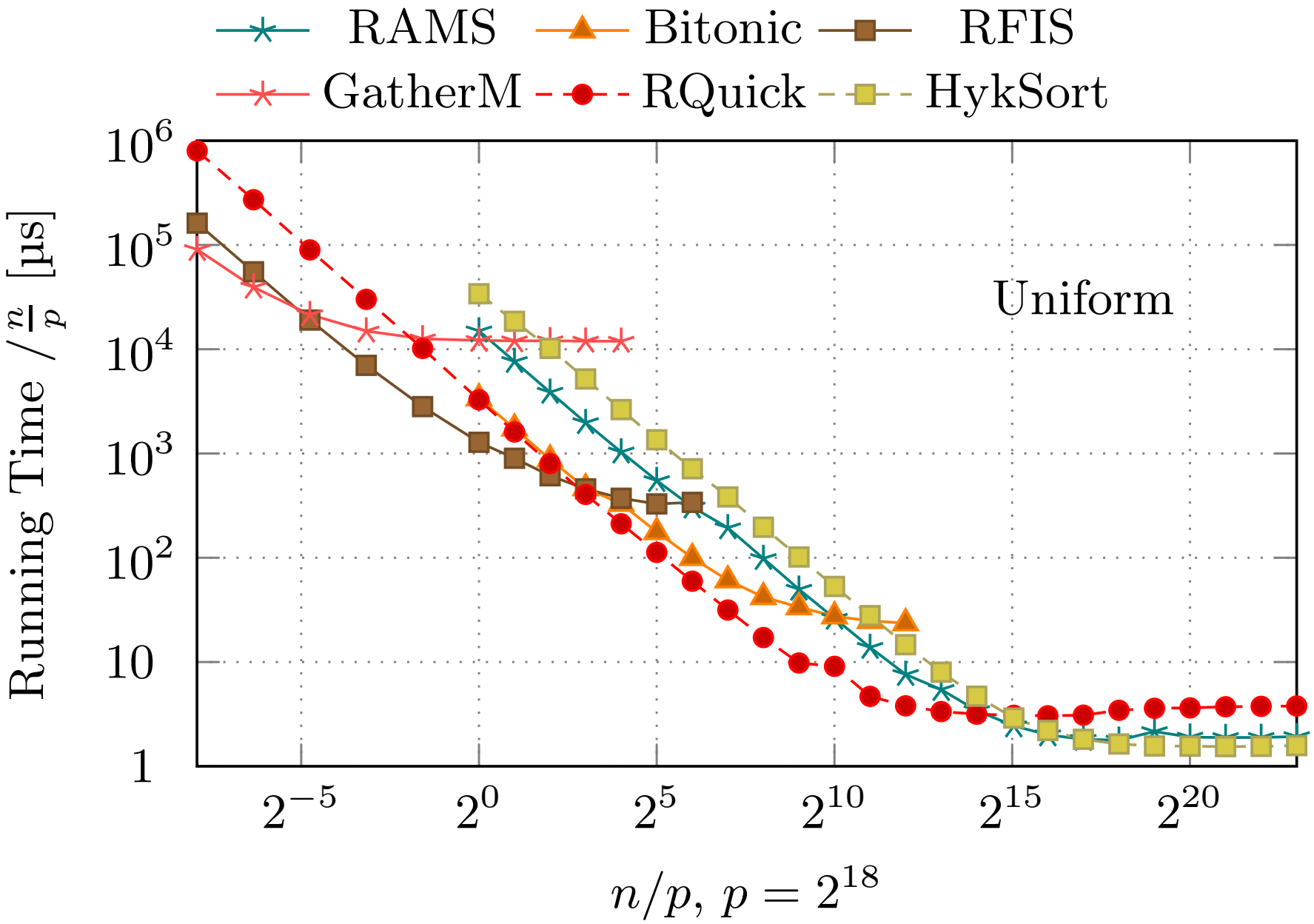
- Janus-quicksort? Axtmann, Wiebigke, Sanders, IPDPS 2018
- for small p' choose pivot carefully
- for small p' ($\Theta(\log p)$) switch to sample sort?

Alternative: always halve PEs, randomizaztion, careful choice of pivot

Axtmann, Sanders, ALENEX 2017

Results

Axtmann Sanders ALENEX 2017



Multi-Pivot Methods

Simplifying assumption: perfect splitters are available for free

// Für $0 < k < p$ let v_k the element with rank $k \cdot n/p$
// Set $v_0 = -\infty$ and $v_p = \infty$.
initialize p empty messages N_k, $(0 \leq k < p)$
for $i := 0$ **to** $n/p - 1$ **do**
    determine k, such that $v_k < d_i \leq v_{k+1}$
    put d_i into message N_k
send N_i to PE i and // All-to-all
receive p messages // personalized communication
sort received data

Analysis

$$T_{\text{par}} = \overbrace{\mathcal{O}\left(\frac{n}{p}\log p\right)}^{\text{distribution}} + \overbrace{T_{\text{seq}}(n/p)}^{\text{local sorting}} + \overbrace{T_{\text{all-to-all}}(p, n/p)}^{\text{data exchange}}$$

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

Idealizing assumption is realistic for permutation.

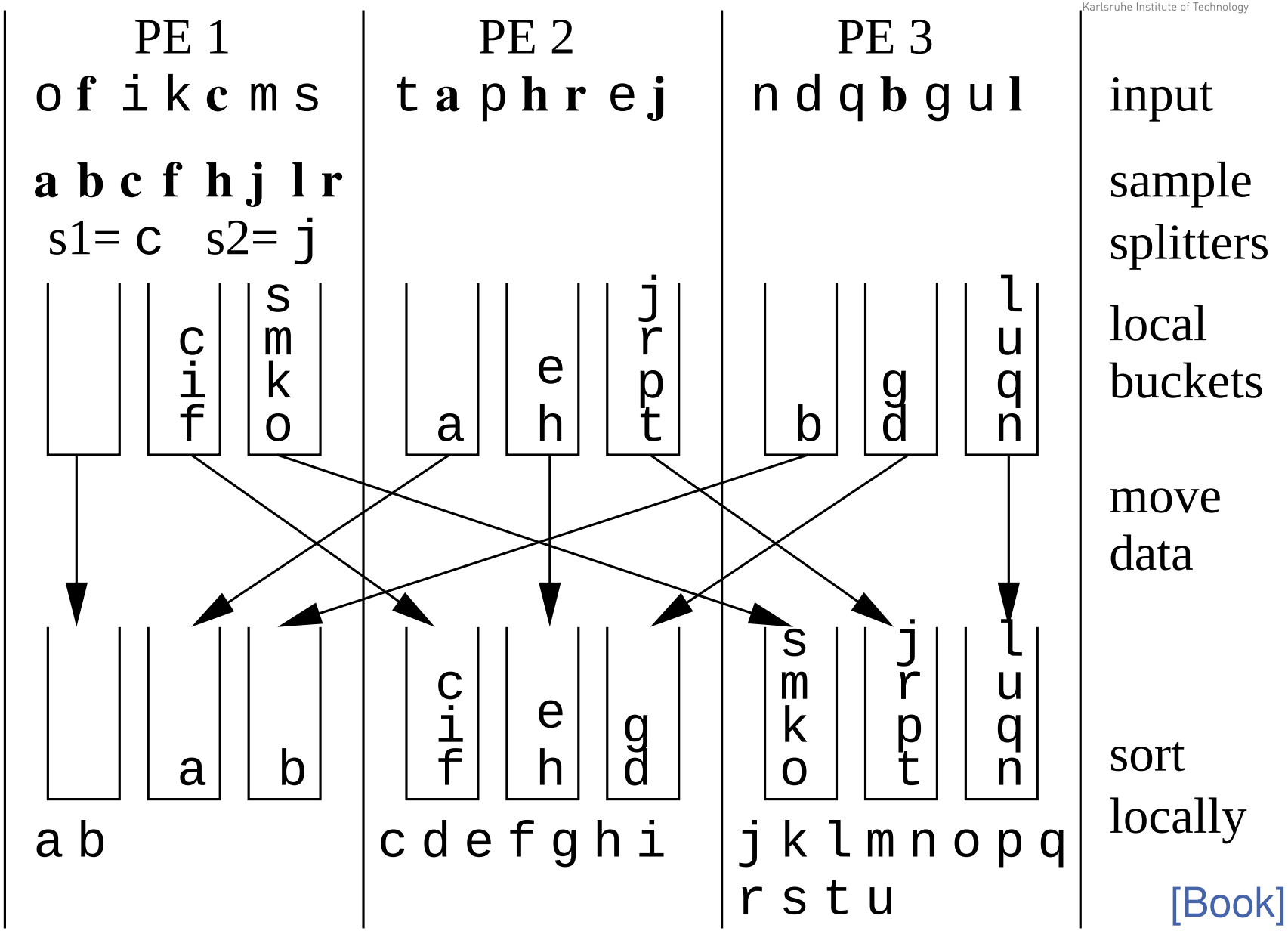
Sample Sort

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

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

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

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


PE 1
PE 2
PE 3
o f i k c m s
t a p h r e j
n d q b g u l
input
a b c f h j l r
sample
s1= c s2= j
splitters
local buckets
move data
sort locally
a b
c d e f g h i
j k l m n o p q r s t u
[Book]

Lemma 2. $a = \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ *suffices such that with probability* $\geq 1 - \frac{1}{n}$ *no PE gets more than* $(1+\varepsilon)n/p$ *elements.*

Lemma:
$a = \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ suffices such that with probability $\geq 1 - \frac{1}{n}$ no PE gets more than $(1+\varepsilon)n/p$ elements.

Proof idea: We analyze and algorithm that choses global samples with replacement.

Let $\langle e_1, \ldots, e_n \rangle$ denote the input in sorted order.

fail: Some PE gets more than $(1+\varepsilon)n/p$ elements

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

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

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

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

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

Chernoff-Bound

Lemma 3. *Let $X = \sum_i X_i$ denote the sum of independent 0-1 random variables.*

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

Applied to our problem:

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

$$\Leftrightarrow a \geq \frac{4}{\varepsilon^2}\ln n$$

■

Analysis of Sample Sort

$$T_{\text{sampleSort}}(p,n) = \overbrace{\overbrace{T_{\text{fastsort}}(p, \mathcal{O}\left(\frac{p \log n}{\varepsilon^2}\right))}^{\text{sort sample}} + \overbrace{T_{\text{allgather}}(p)}^{\text{collect/distribute splitters}}}^{\text{small if } n \gg p^2 \log p}$$
$$+ \underbrace{\mathcal{O}\left(\frac{n}{p} \log p\right)}_{\text{partition}} + \underbrace{T_{\text{seq}}((1+\varepsilon)\frac{n}{p})}_{\text{local sorting}} + \underbrace{T_{\text{all-to-all}}(p, (1+\varepsilon)\frac{n}{p})}_{\text{data exchange}}$$

Sorting Samples

- Using gather/gossiping
- Using gather–merge
- Fast ranking
- Parallel quicksort
- Recursively using sample sort

Sorting Samples

efficient if $n \gg$

- Using gather/gossiping — $\frac{p^2 \log p T_{\text{compr}}}{\varepsilon^2}$
- Using gather–merge — $\frac{p^2 \beta}{\varepsilon^2 T_{\text{compr}}}$
- Fast ranking — $\frac{p^2 \beta}{\log p T_{\text{compr}}}$
- Parallel quicksort — $\frac{p^2 \beta}{\log p T_{\text{compr}}}$
- Recursively using 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
MPI_Allgather(locS.data(), locS.size(), mpiType,
   s.data(), locS.size(), mpiType, comm);


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

Partition Locally

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

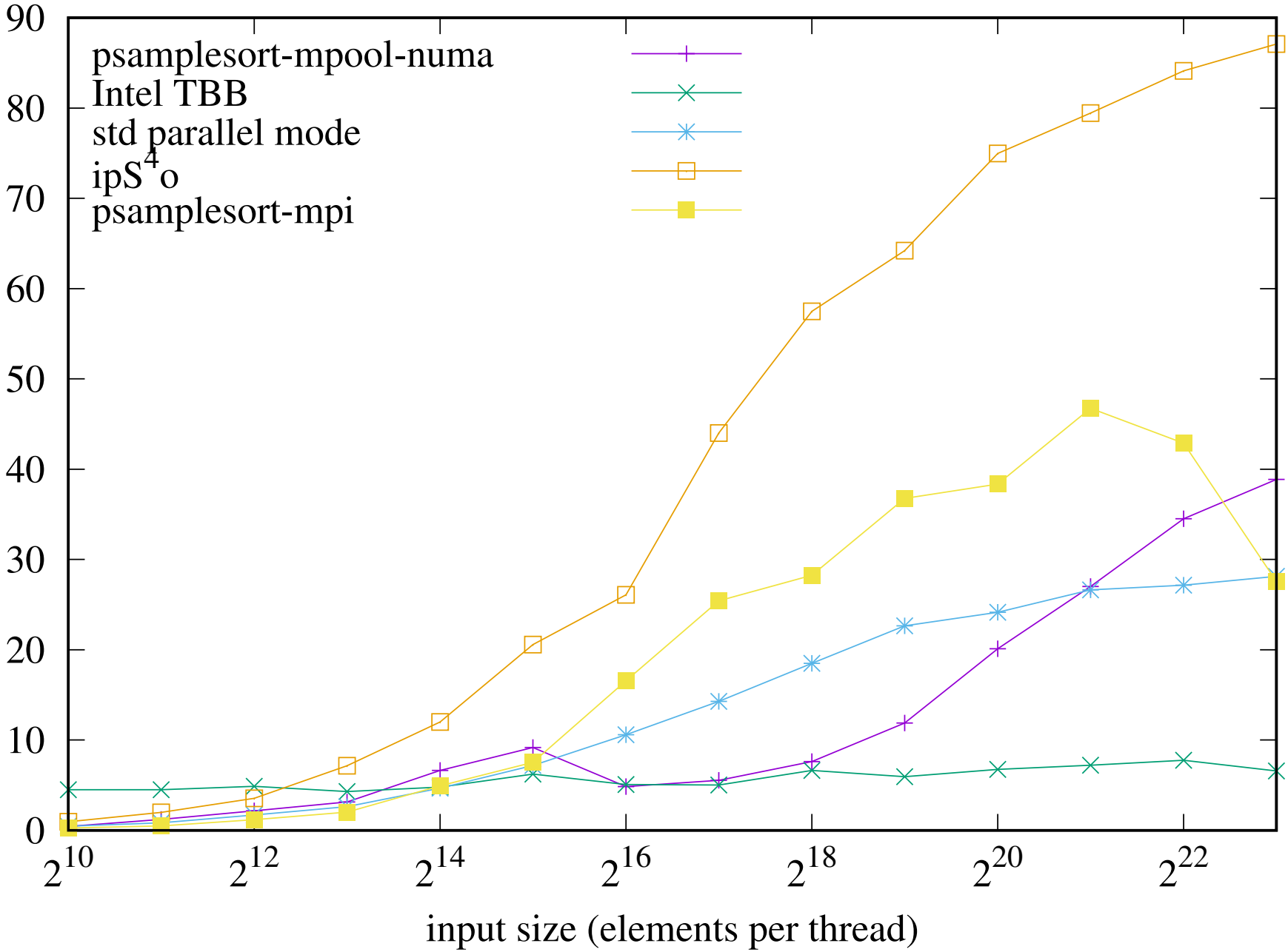
Find Message Sizes

```
// 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



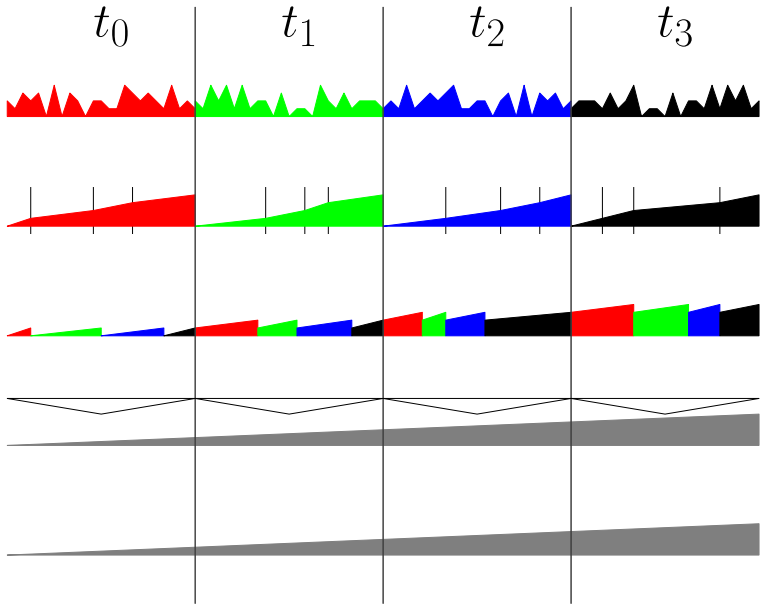
Sorting by multiway merging

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

Idea: each PE determines a splitter with appropriate global Rank

(shared memory)

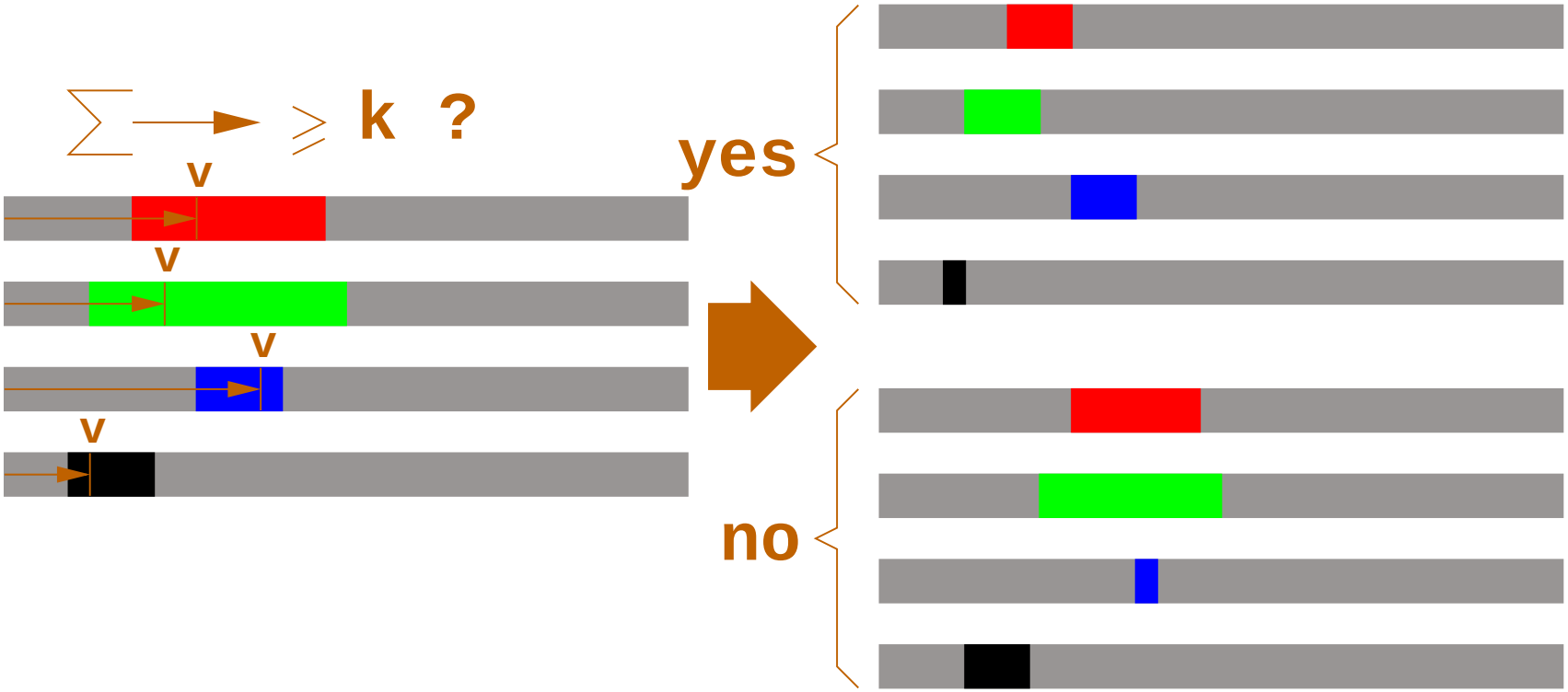
Comparison based lower bound: $\Omega\left(p \log \frac{n}{p}\right)$

We present an algorithmus with $\mathcal{O}\left(p \log n \log \frac{n}{p}\right)$

Splitter Selection

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

Simple algorithm: quickSelect exploiting sortedness of the sequences.



Idea:

Ordinary select but $p\times$ binary search instead of partitioning

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

Analysis of p-way mergesort

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

- efficient if $n \gg p^2 \log p$
- deterministic (almost)
- perfect load balance
- somewhat worse constant factors than sample sort

Distributed Multisequence Selection

Owner computes paradigm

$\mathcal{O}(\log n)$ global levels of recursion.

Gather + Broadcast for finding pivot / distribution (vector length $p-1$).

$p-1$ local searches everywhere.

Reduction for finding partition sizes (Vector length $p-1$).

Expected time

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

Distributed 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** $\ell_i := m_i$

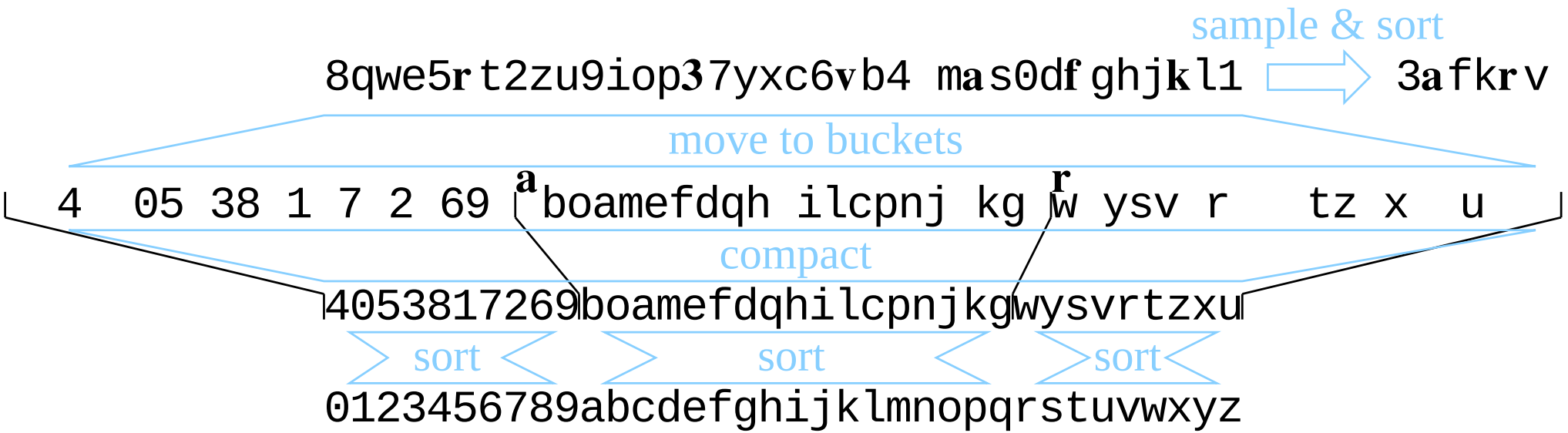
return ℓ

CRCW Sorting in logarithmic time

Consider case $n = p$.

- sample of size $\sqrt{p}$
- $k = \Theta\left(\sqrt{p}/\log p\right)$ splitters
- Buckets have size $\leq cp/k$ elements whp
- Allocate buckets of size $2cp/k$
- Write elements to random free positions within their bucket
- Compactify using prefix sums
- Recursion

Example


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

More on Sorting I

Cole's merge sort: [JáJá Section 4.3.2]

Time $\mathcal{O}\left(\frac{n}{p}+\log p\right)$ deterministic, EREW PRAM (CREW in [JáJá]). Idea: Pipelined parallel merge sort. Use (deterministic) sampling to predict where data comes from.

Sorting Networks: nodes sort 2 elements. Simple networks have $\mathcal{O}\left(\log^2 n\right)$ depth (e.g. bitonic sort). They yield reasonable deterministic sorting algorithms (2 elements $\rightsquigarrow$ merge-and-split of two subsequences). Very complicated ones with depth $\mathcal{O}\left(\log n\right)$.

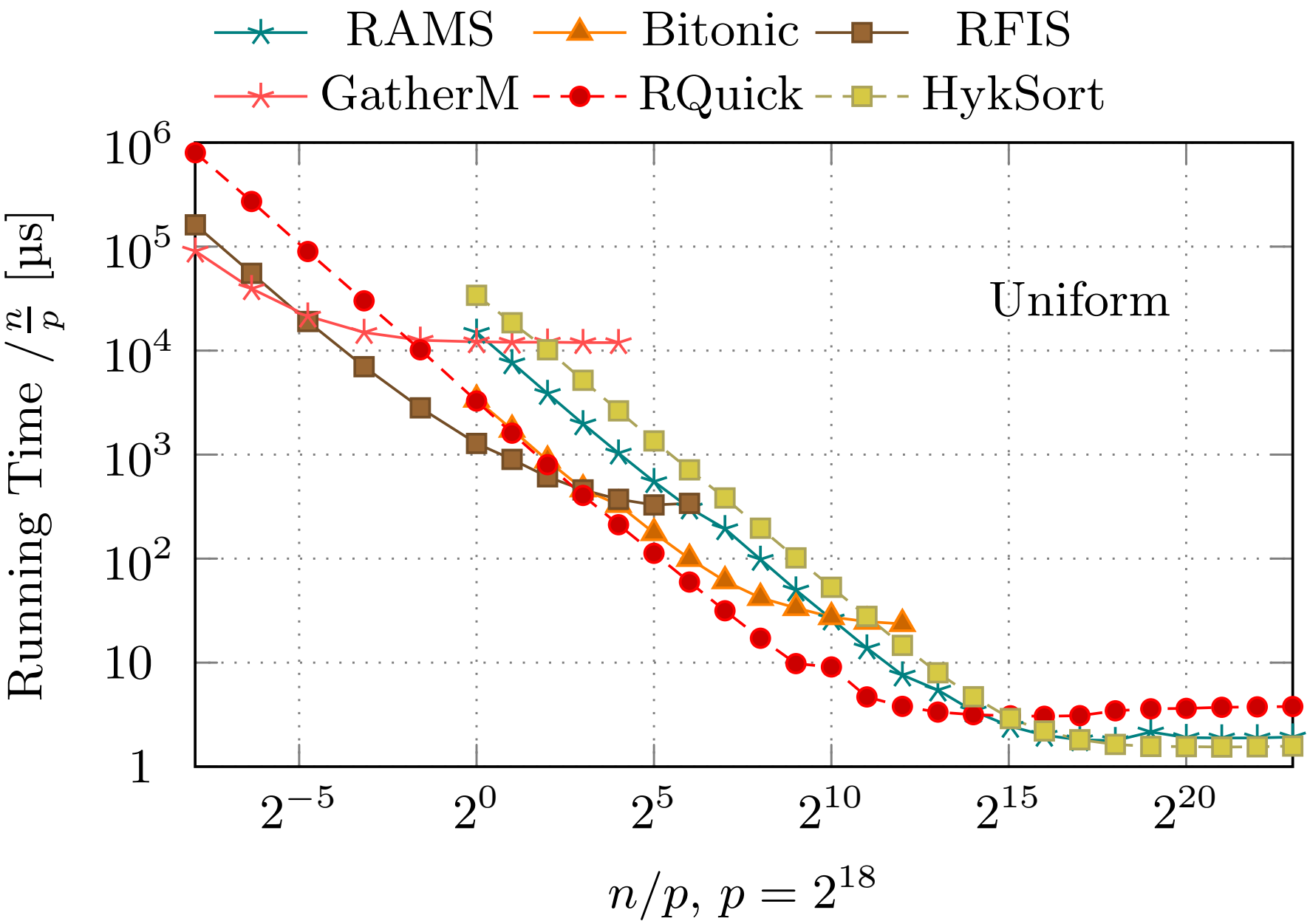
More on Sorting II

Integer Sorting: (Close to) linear work. Very fast algorithms on CRCW PRAM.

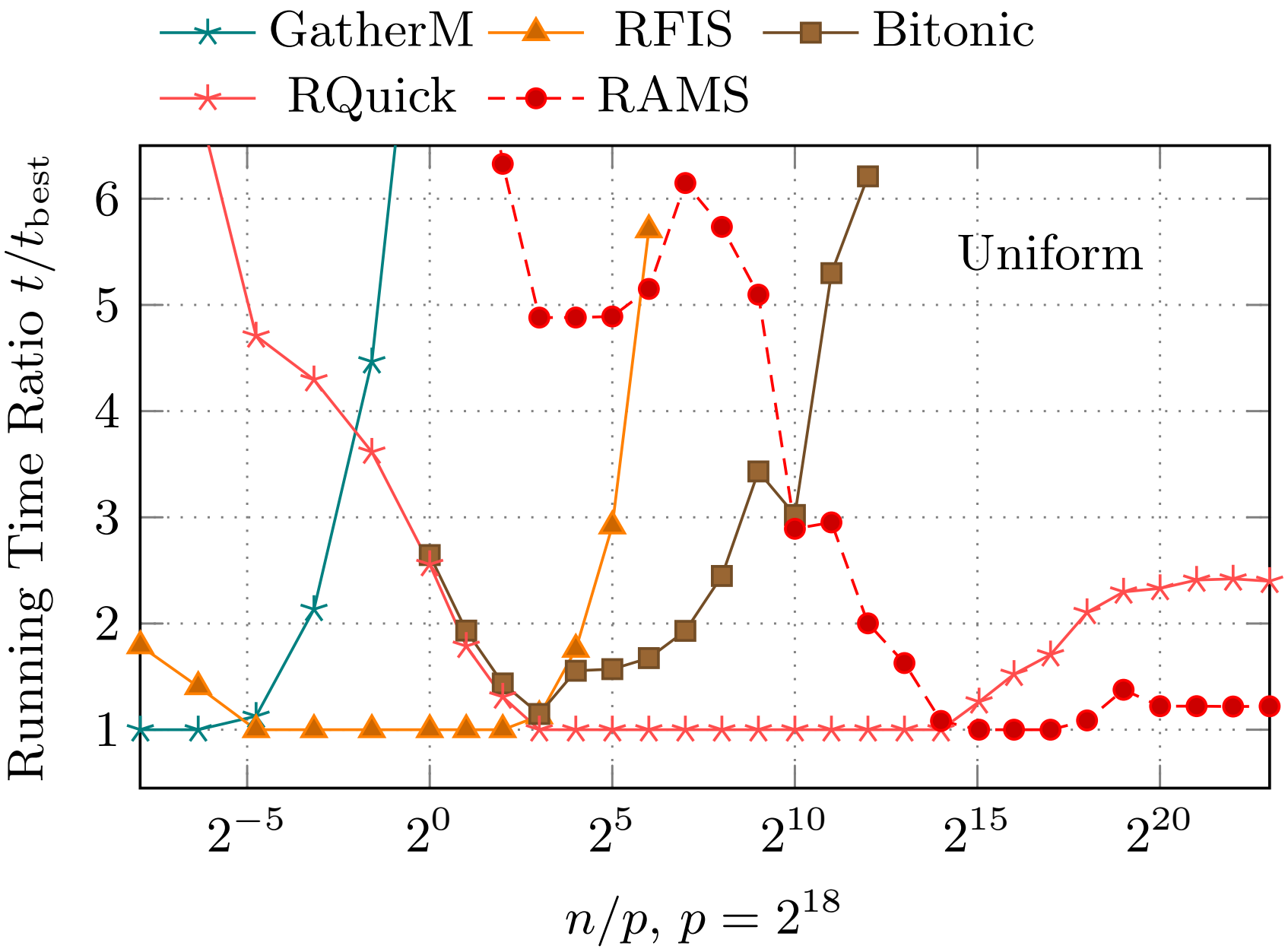
Multi-Pass-Sample/Merge-Sort: more general compromise between latency and communication volume, e.g. AMS-Sort Axtmann, Bingmann, Schulz, Sanders SPAA 2015

Measurements

Axtmann Sanders ALENEX 2017



Slowdown wrt Fastest Algorithm



Collective Communication

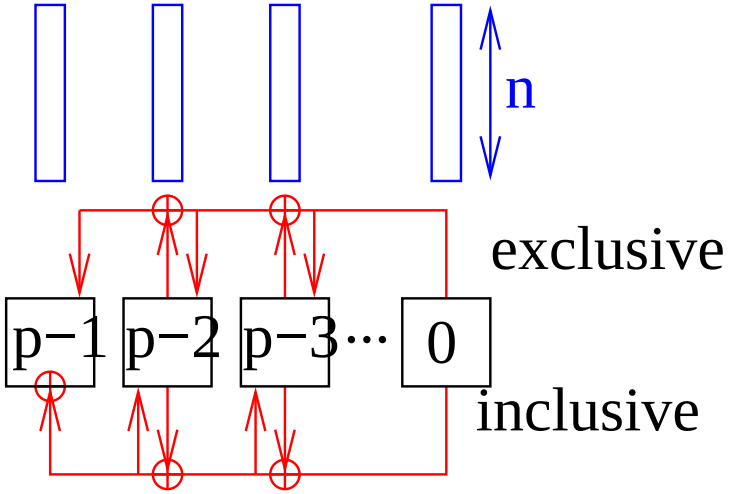
- Broadcast
- Reduction
- Prefix sum
- Not here hier: gather / scatter
- Gossiping ($=$ all-gather $=$ gather $+$ broadcast)
- All-to-all personalized communication
  - equal message lengths
  - arbitrary message lengths, $= h$-relation

Prefix sums

[Leighton 1.2.2] Compute

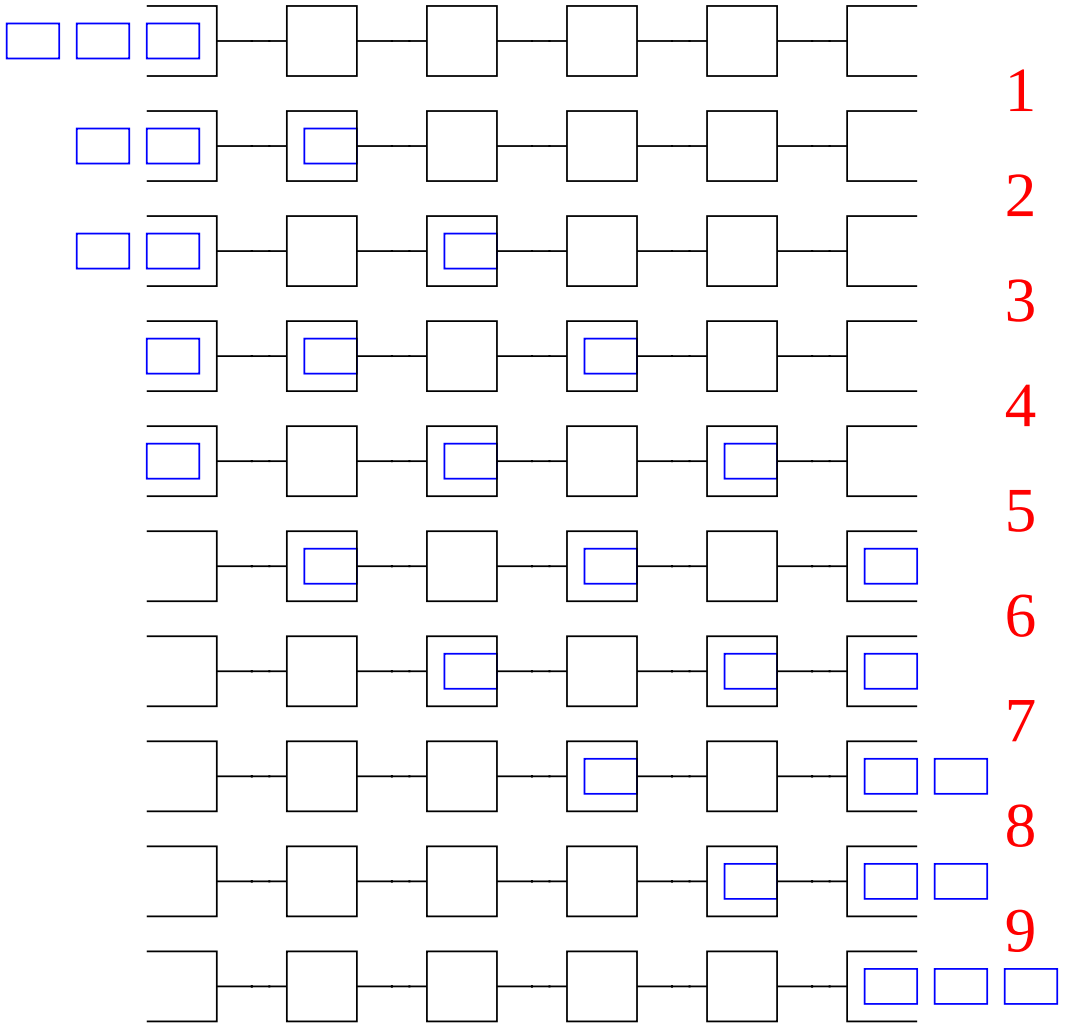
$$x@i := \bigotimes_{i' \leq i} m@i'$$

(on PE i, m may be a vector with n bytes length.)



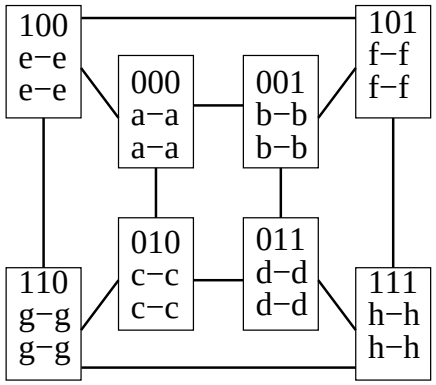
Plain Pipeline

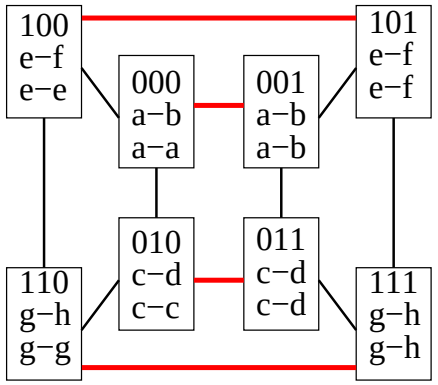
As in broadcast

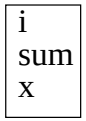


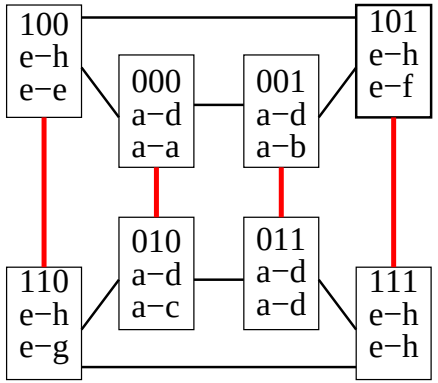
Hypercube Algorithm

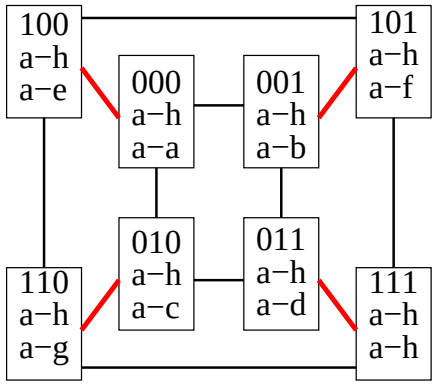
// view PE index i as a
// d-bit bit array
Function hcPrefix(m)
  $x := \sigma := m$
  for $k := 0$ **to** $d-1$ **do**
    invariant $\sigma = \bigotimes_{j=i[k..d-1]0^k}^{i[k..d-1]1^k}$ m@j
    invariant $x = \bigotimes_{j=i[k..d-1]0^k}^{i}$ m@j
    $y := \sigma@(i \oplus 2^k)$ // sendRecv
    $\sigma := \sigma \otimes y$
    if $i[k] = 1$ **then** $x := y \otimes x$
  return x











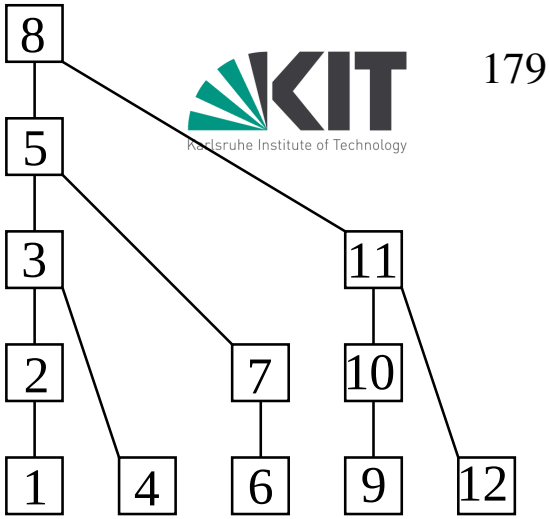
Analysis

Telephone model:

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

Pipelining does not work since all PEs are busy.

Pipelined Binary Tree Prefix Sums



Inorder numbering of the nodes

Upward phase: as with reduction but

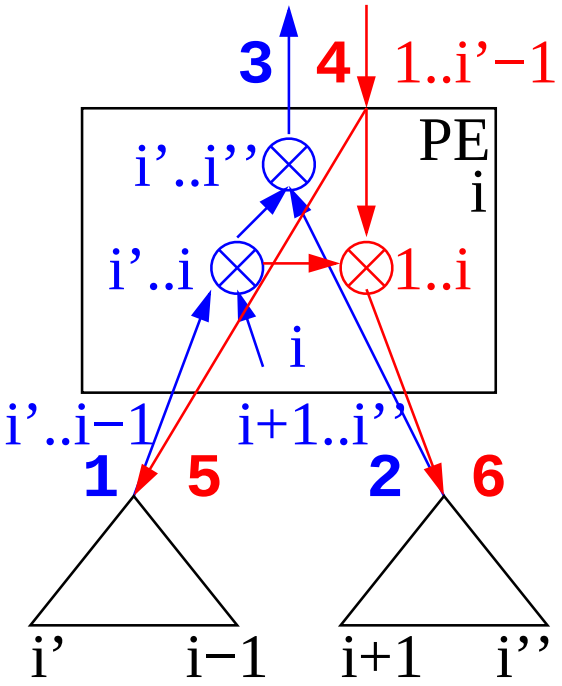
PE i stores $\sum_{j=i'}^{i} x@j$

Downward phase: PE i receives $\sum_{j=1}^{i'-1} x@j$

(root: $= 0$!)

and forwards this to the left.

right subtree gets $\sum_{j=1}^{i} x@j$

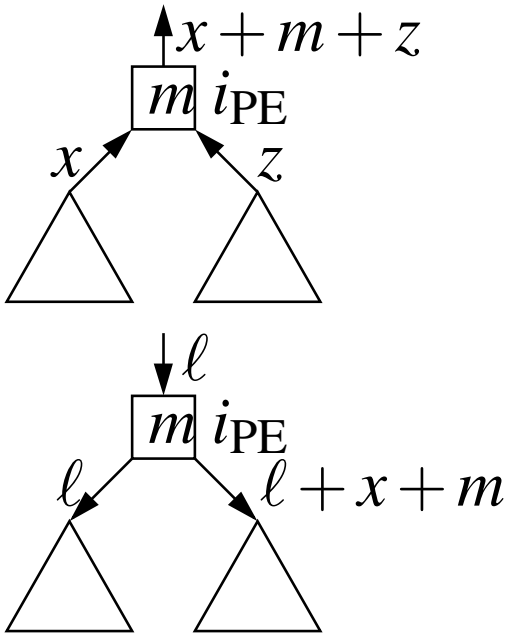


Each PE only active once per phase. $\rightarrow$ pipelining OK

Pseudocode

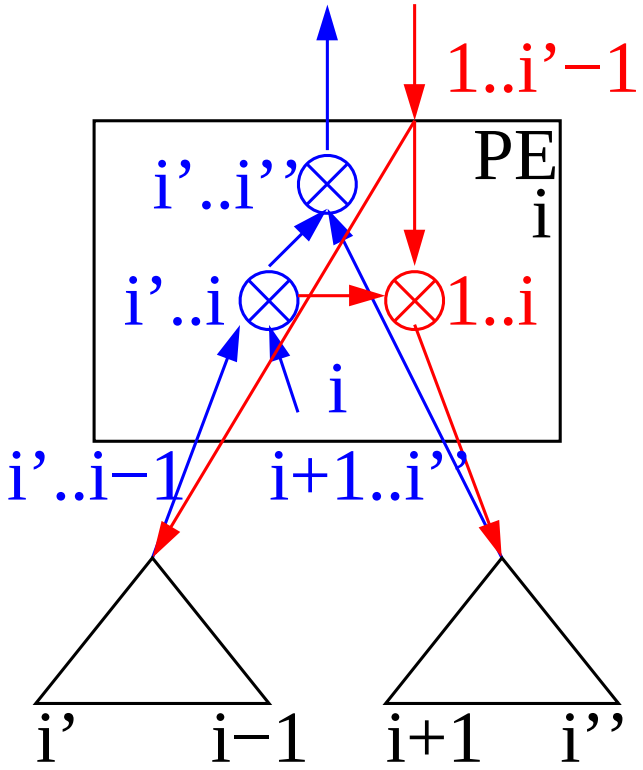
Function InOrderTree::prefixSum(m)

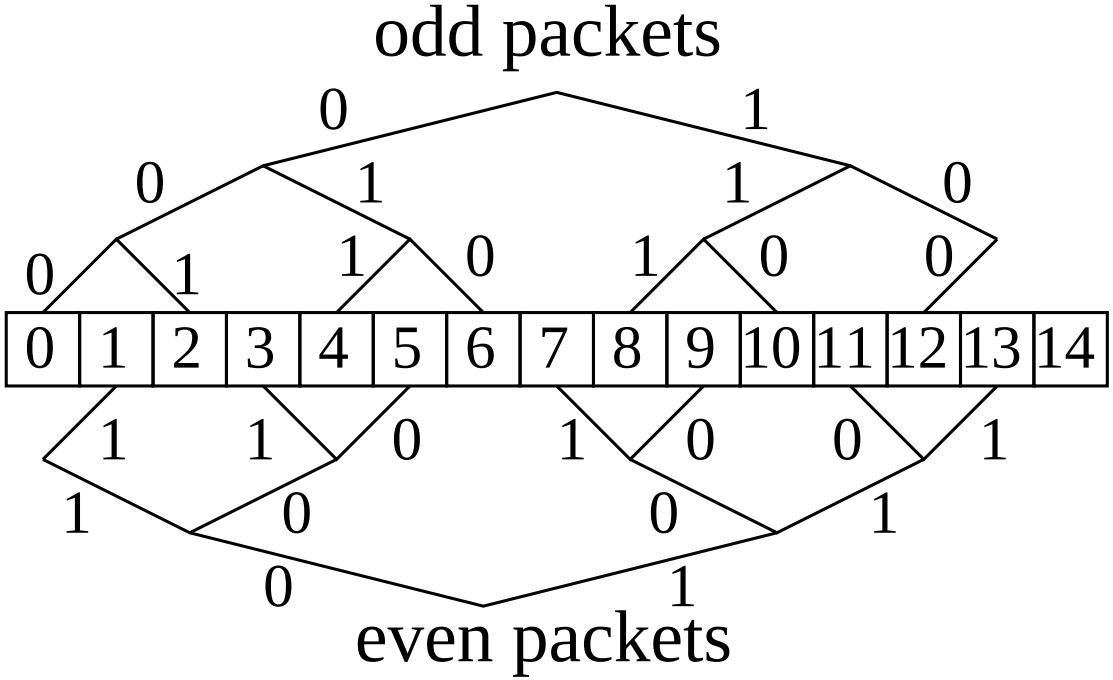
```
// upward phase:
x := 0;   receive(leftChild, x)
z := 0;   receive(rightChild, z)
send(parent, x + m + z)

// downward phase:
ℓ := 0;   receive(parent, ℓ)
send(leftChild, ℓ)
send(rightChild, ℓ + x + m)

return ℓ + x + m
```



23-Prefix Sums

Numbering is inorder-numbering for both trees!





Analysis

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

Generalization:

- Applies to any algorithm based on inorder numbered trees

⇝ ESBT does not work?

Gossiping

Each PE has a message m of length n.

At the end, each PE should know all messages.

Hypercube Algorithm

Let '$\cdot$' denote the concatenation operation; $p = 2^d$

PE i

$y := m$

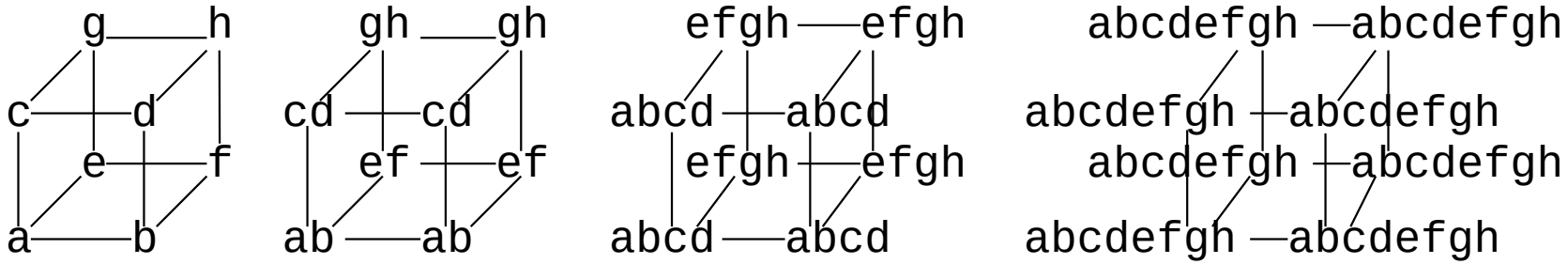
for $0 \leq j < d$ **do**

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

$y := y \cdot y'$

return y

Example


g h
c d
e f
a b
gh gh
cd cd
ef ef
ab ab
efgh efgh
abcd abcd
efgh efgh
abcd abcd
abcdefgh abcdefgh
abcdefgh abcdefgh
abcdefgh abcdefgh
abcdefgh abcdefgh

Analysis

Telephone model, $p = 2^d$ PEs, n byte per 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

Reduction instead of concatenation.

Advantage: Factor two less startups than reduction plus broadcast

Disadvantage: $p \log p$ messages.

This a disadvantage for congenstion-prone networks.

All-to-all Personalized Communication

Each PE has $p-1$ messages of length n. One for each other PE.

$m[i]@i$ is for PE i itself

Hypercube Algorithm

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

Analysis, Telephone model:

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

Fully Connected:

When n is large, rather send messages individually

(Factor $\log p$ less communication volume)

1-Factor-Algorithm

[König 1936]

p odd, i is PE-index:

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

$k := 2r \bmod p$

$j := (k-i) \bmod p'$

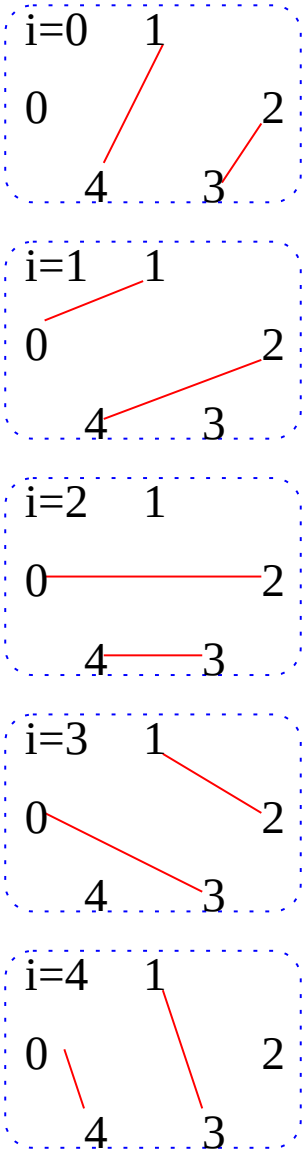
send(j, m_{ij}) || recv(j, m_{ji})

pairwise communication (telephone model):

The partner of the partners of j in round i is

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

Time: $p(n\beta+\alpha)$ optimal for $n \to \infty$



1-Factor Algorithm

p even:

//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

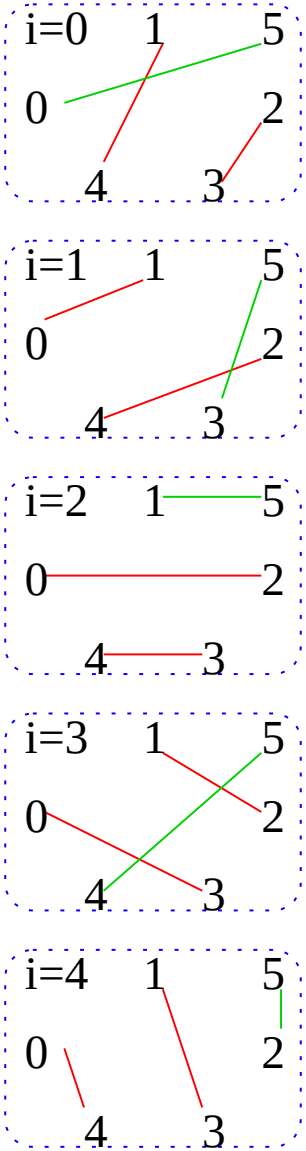
if $j =$ idle **then**

exchange data with PE $p-1$

else

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

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



Data exchange with irregular message lengths

- Particularly interesting with all-to-all $\rightarrow$ sorting
- similar problems in inhomogeneous interconnection networks or competition by other jobs.

The "Ostrich"-Algorithm

Push all messages using asynchronous send operations.

Receive what comes in.

Ostrich-Analysis:

BSP-Model: Time $L + gh$

But what is L and g in our single-ported models?

h-Relation

$h_{\text{in}}(i)$:= # packets received by PE i

$h_{\text{out}}(i)$:= # packets sent by PE i

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

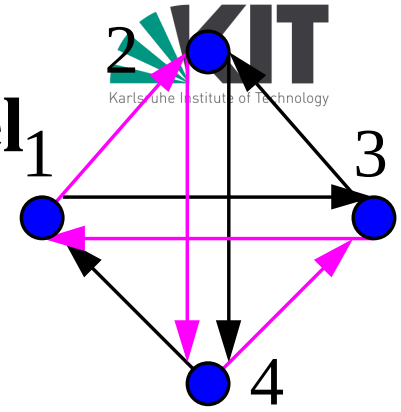
duplex: $h := \max_{i=1}^{p} \max(h_{\text{in}}(i), h_{\text{out}}(i))$

Lower bound for packet-wise delivery:

h steps, i.e.,

Time $h(\alpha + |\text{packet}|\beta)$

Offline h-Relations in the Duplex Model

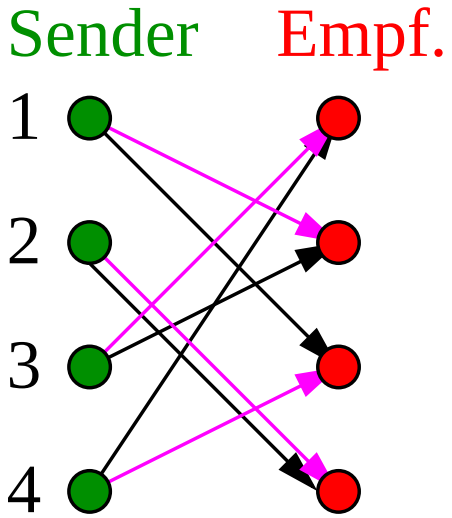


[König 1916]

Consider the bipartite multigraph

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

$|\{(s_i, r_j) \in E\}| =$ # packets from PE i to PE j.



Theorem: $\exists$ edge coloring $\phi : E \to \{1..h\}$, i.e.,

no two equal-colored edges are incident to any node.

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

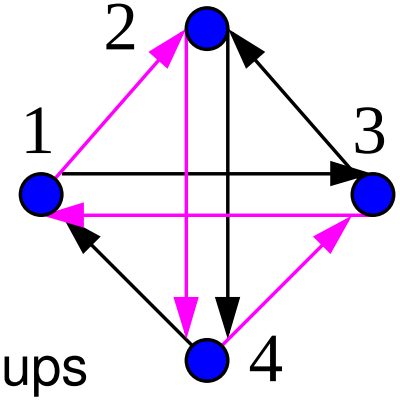
send messages with color j

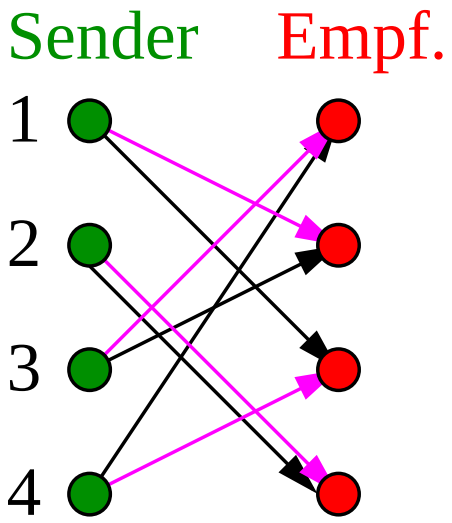
optimal when postulating packet-wise delivery

Offline h-Relations in the Duplex Model

Problems:

- Computing edge colorings online is complicated and expensive
- Shredding messages into packets increases # startups





Offline h-Relations in the Simplex-Model

[Petersen 1891? Shannon 1949?]

Consider the multigraph $G = (\{1,\ldots,p\}, E)$

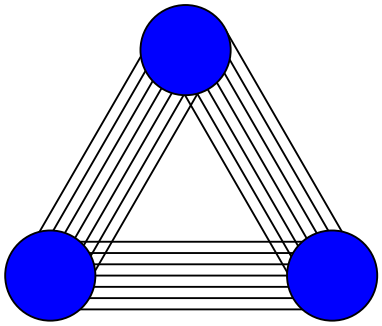
with $|\{\{i,j\} \in E\}| =$ # packets between PE i and PE j (both directions).

Theorem: $\exists$ edge coloring $\phi : E \to \{1..3\lfloor h/2 \rfloor + h \bmod 2\}$

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

Send messages of color j

optimal?? ?



How Helper Hasten h-Relations

[Sanders Solis-Oba 2000]

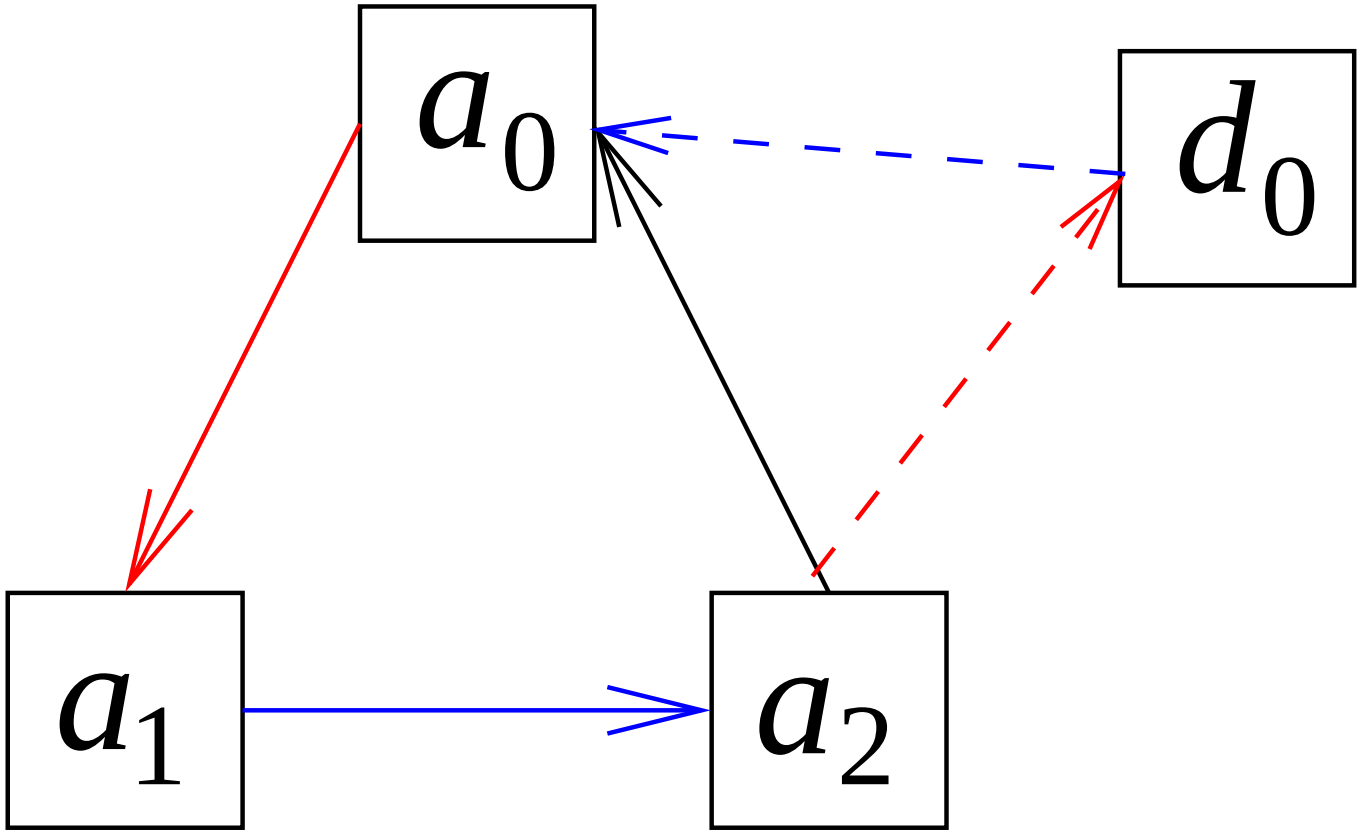
Satz 4. *For h-relations in the simplex model,*

$$\#\text{steps} = \begin{cases} \frac{6}{5}(h+1) & \text{if } p \text{ even} \\ (\frac{6}{5}+\frac{2}{p})(h+1) & \text{if } p \text{ odd} \end{cases} .$$

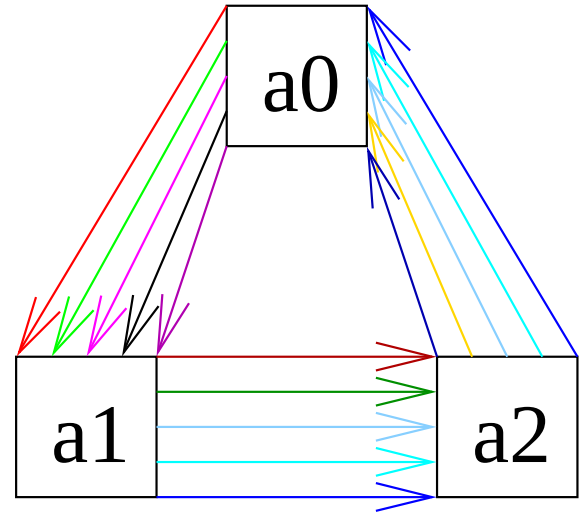
On the other hand, there is a lower bound

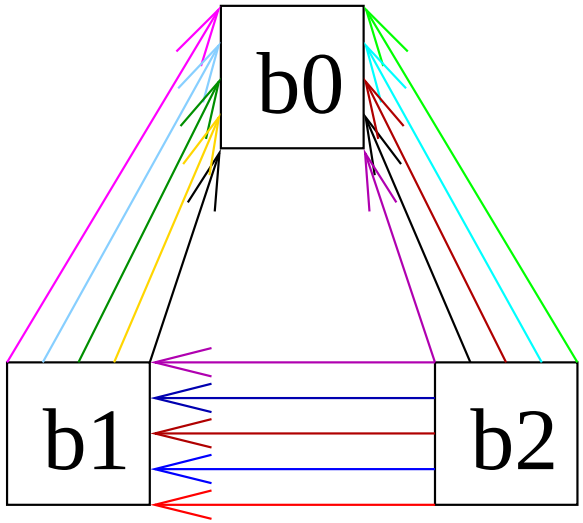
$$\#\text{steps} \geq \begin{cases} \frac{6}{5}h & \text{if } p \text{ even} \\ (\frac{6}{5}+\frac{18}{25p})h & \text{if } p \text{ odd} \end{cases}$$

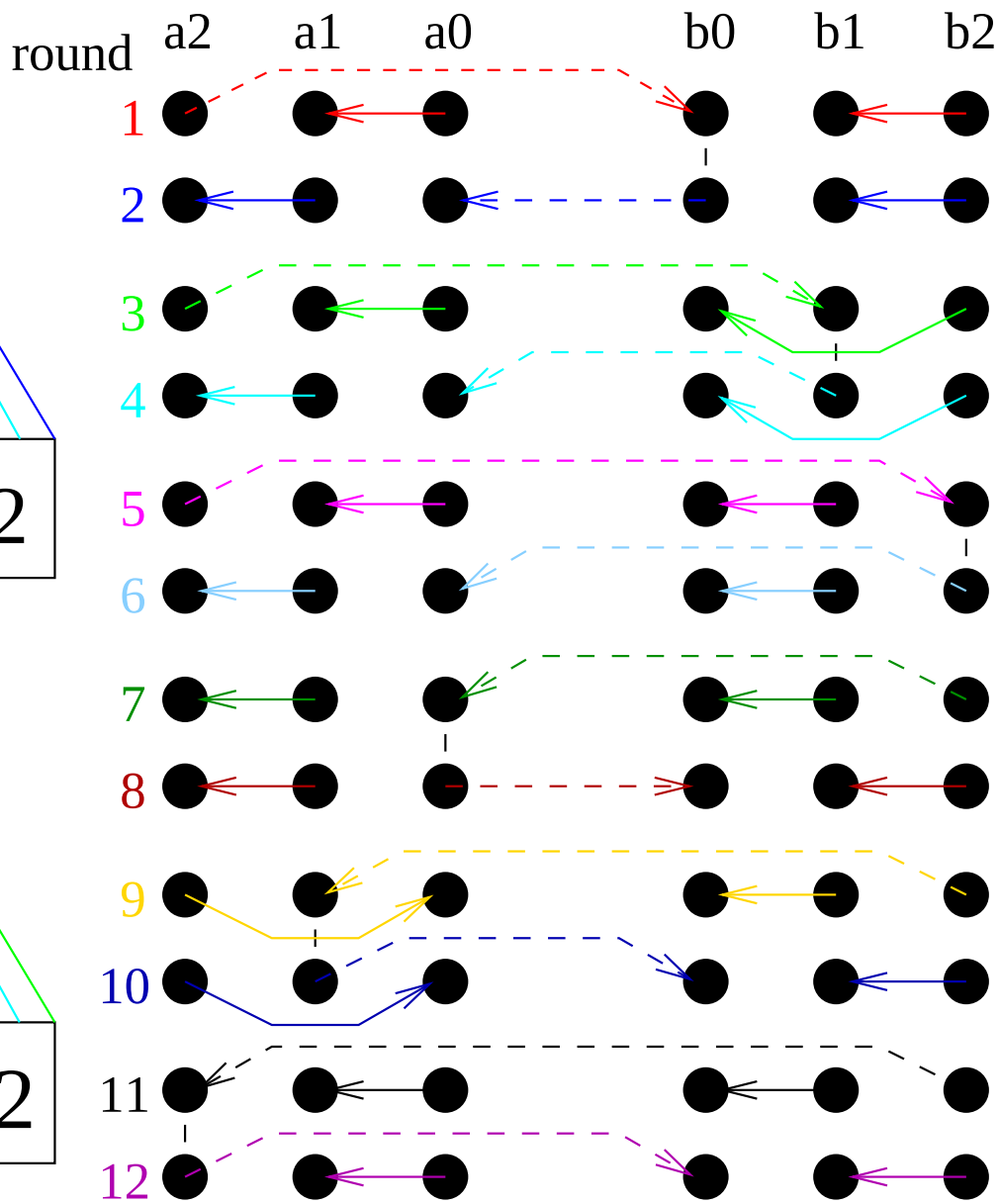
A Very Simple Case



Two Triangles







Reduction h-Relation $\rightsquigarrow \left\lceil \frac{h}{2} \right\rceil$ 2-Relations

- Ignore direction of communications for now
- Connect nodes with odd degree

  $\rightsquigarrow$ all nodes have even degree
- Eulertour-technique: decompose the graph into edge disjoint cycles
- Direct cycles in clockwise direction

  $\rightsquigarrow$ indegree and outdegree $\leq \lceil h/2 \rceil$
- Build bipartite graph (as before)
- Color bipartite graph
- Color classes in bipartite graph $\rightsquigarrow$ edge disjoint simple cycles in the input graph (2-relations)
- Reinstate orginal communication direction

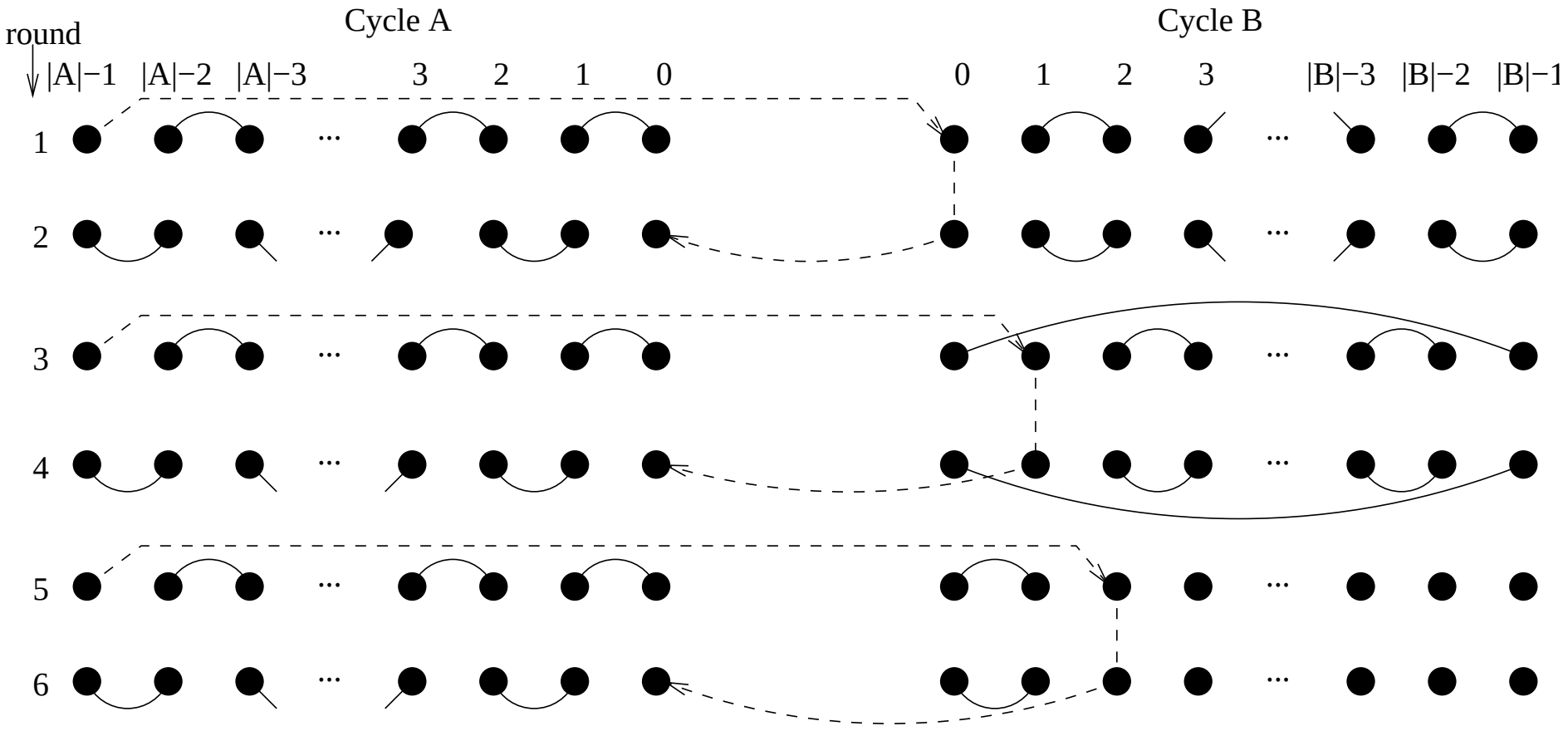
Routing 2-Relations for Even p

Pair odd cycles.

1-cycles have nothing to do $\rightsquigarrow$ most simple case

Two odd cycles with ≥ 3 nodes

Split packets into 5 subpackets



Then turn this around

Odd p

Idea: Delete one edge in each 2-factor

Do that “always somewhere else”

Collect $\Theta(p)$ removed edges into a matching

$\rightsquigarrow$ one additional step every $\Theta(p)$ 2-factors.

Open Problems

- Get rid of splitting into 5 subpackages?
- Conjecture:

  One h-Relation with $\leq \frac{3}{8}hp$ packets can be delivered in $\approx h$ steps.
- Explicitly account for startup overheads
- Explicitly account for interconnection network?
- Distributed scheduling

A Simple Distributed Algorithm — The Two-Phase-Algorithm

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

Simplifying assumptions:

- All message lengths are divisible by p
  (in doubt round up)
- communication “with oneself” is accounted for
- All PEs send and receive exactly h bytes
  (In doubt “pad” the messages)

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

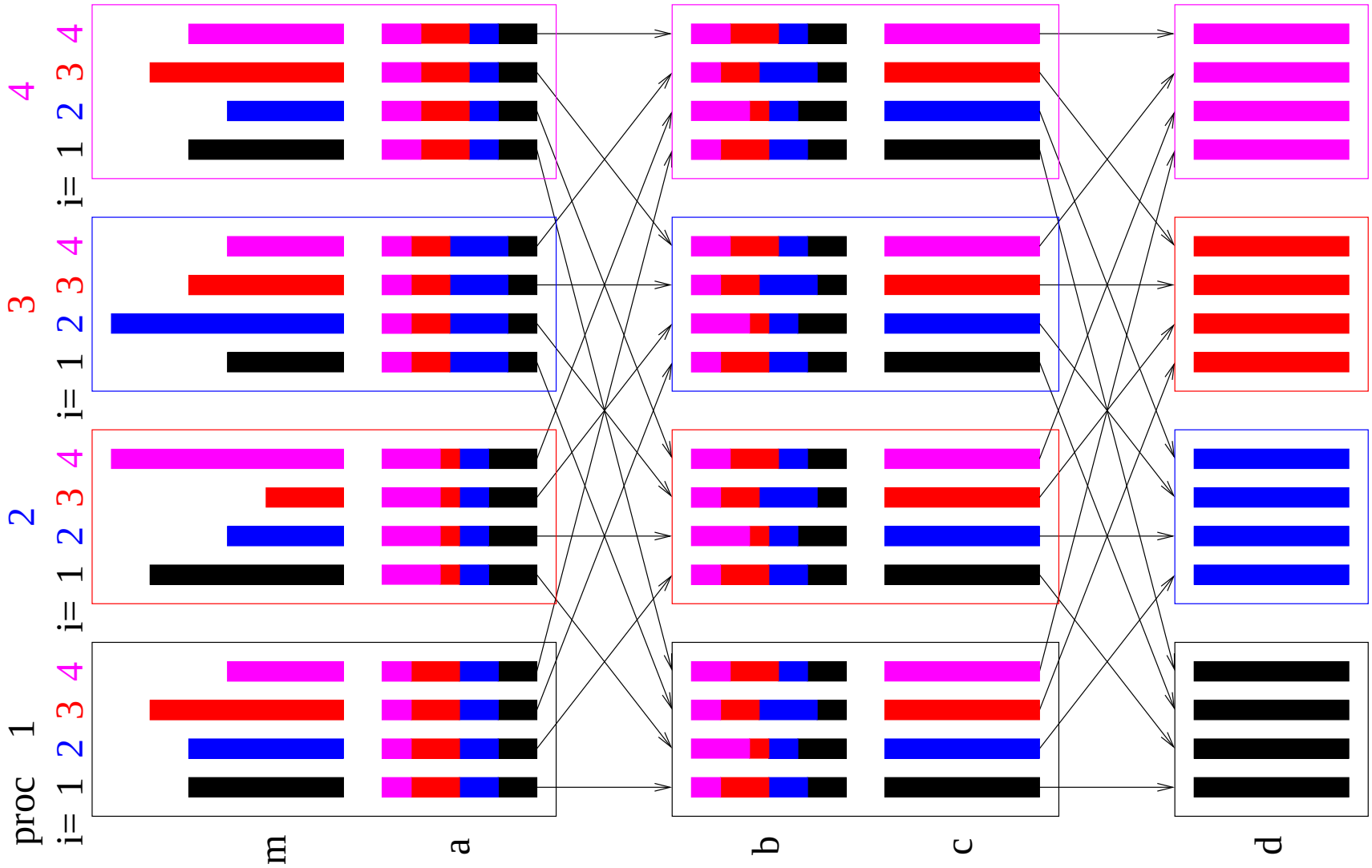
for $j := 1$ **to** p **do**

$c[i] := c[i] \odot b[j][\delta[j]..\delta[j] + \frac{n[i]@j}{p} - 1]$ // 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 1 2 3 4
i= 1 2 3 4
m
a
b
c
d

More on the Two-Phase-Algorithm

- Large p, small messages $\rightsquigarrow$
  split local data into $\mathcal{O}(p \log p)$ pieces (not p^2) and disperse randomly.
- Split the problems into regular and irregular part $\rightsquigarrow$ two-phase protocol only applied to a part of the data.
  $\rightsquigarrow$ open problem: how to split?

A Non-Preemptive Offline Algorithm (simplex)

[Sanders Solis-Oba 99, unpublished]

Goal: deliver all messages directly, as a whole.

Let k:= max. # messages one PE is involved in.

Time for executing the schedule $k\alpha + 2h\beta$.

Here h is measured in bytes!

Abstract Description

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"

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

Can be implemented such that per message, the time for $\mathcal{O}(1)$ priority-queue operationen and one p-bit bit-vector operation is used.

$\rightsquigarrow$ practicable for message lengths $\gg p$ and moderate p.

Open Problems for Non-Preemptive Offline Algorithms

- implement, measure, use, e.g. sorting, construction of suffix-arrays
- Better approximation algorithms?
- Parallel scheduling algorithms

Summary: All-to-All

Ostrich: Delegate to online, asynchronous routing.

Good when that is implemented well.

Regular+2Phase: more robust. But, factor 2 is troubling. A lot of copying overhead.

Non-preemptive: Minimizes startups and communication volume. Faktor 2 (worst case). Centralized Scheduling is troubling.

Good for repeated identical problems.

Coloring-based algorithms: Almost optimal for large packets. Complex.

Distributed Implementation? Splitting into packets is troubling.

Comparison of approaches?

Parallel Priority Queues

Manage a set M of elements. $n = |M|$. Initially empty

Binary Heaps (sequential)

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

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

Parallel Priority Queues, Goal

insert*: Each PE inserts a constant number of elements,

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

deleteMin*: delete the p smallest elements,

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

Asynchronous Variant (later): Each PE can insert or delete at any time.

Semantics: $\exists$ temporal ordering of the operations consistent with sequential execution.

Applications

- Priority-driven scheduling
- Best first branch-and-bound:

  Find best solution in a large, implicitly defined tree. (later more)
- Discrete event simulation

Naive Implementation

PE 0 manages a sequential PQ

All others send requests

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

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

Branch-and-Bound

H: tree (V,E) with bounded node degree

$c(v)$: node costs — growing when descending a path

v^*: leaf with minimal costs

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

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

h: Depth of $\tilde{H}$ (subgraph of H induced by $\tilde{V}$).

T_x: Time for generating the successors of a node

T_{coll}: Upper bound for broadcast, min-reduction, prefix-sum, routing one element from/to random partner.
$\mathcal{O}(\alpha \log p)$ on many networks.

Sequential Branch-and-Bound

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

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

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

  v:= Q.deleteMin

  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

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

Parallel Branch-and-Bound

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

Analysis

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

Case 1:

(at most m/p Iterations):

All processed nodes are in $\tilde{V}$

Case 2:

(at most h Iterations):

Some nodes outside $\tilde{V}$

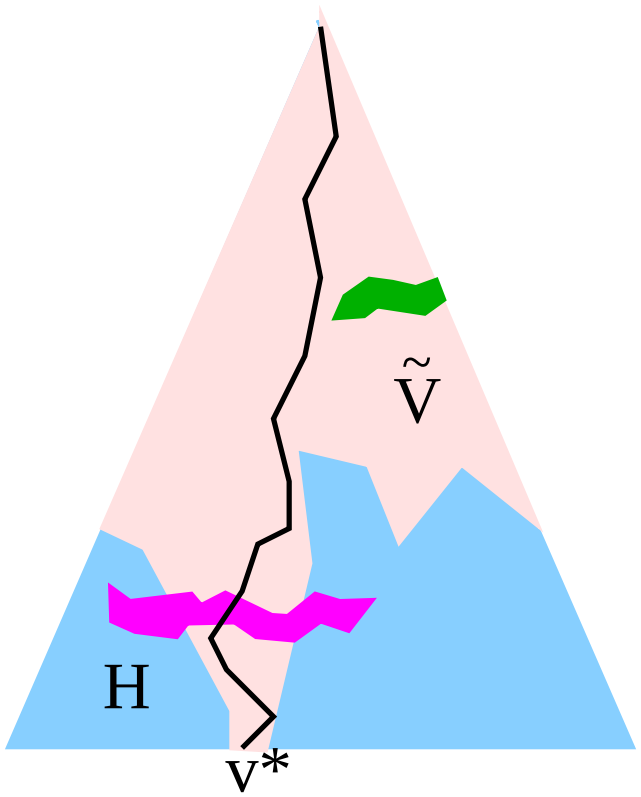
are processed

$\rightarrow$ maximal path length

from a node in Q

to the optimal solutions

is being reduced.



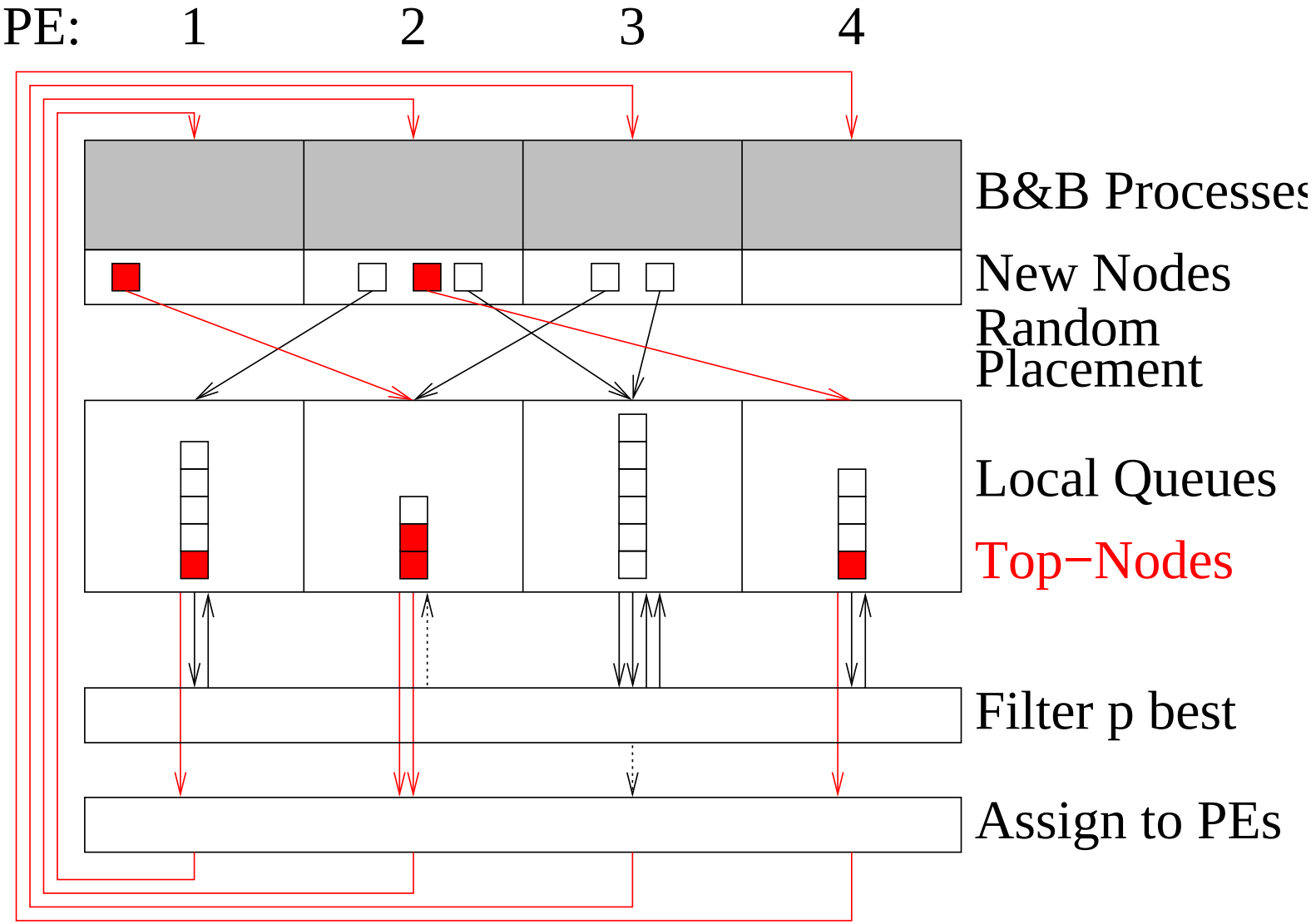
□

The Karp–Zhang Algorithm

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

Our Approach



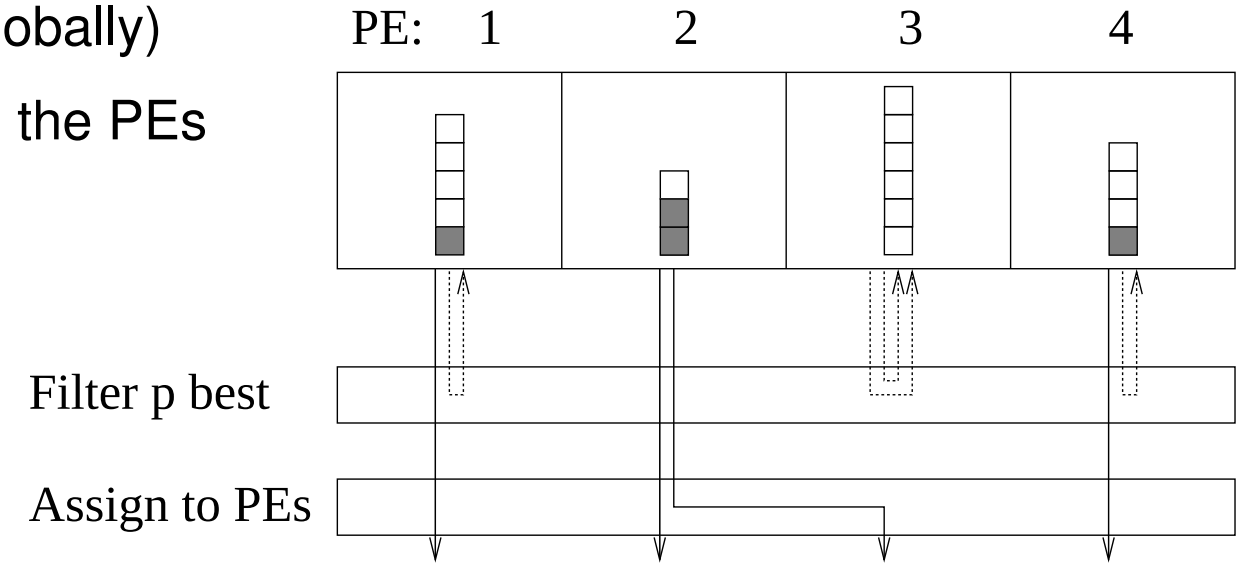
Parallel Priority Queues: Approach

- The queue is the union of local queues
- insert sends new elements to random local queues

  Intuition: each PE gets a representative view on the data
- deleteMin* finds the globally smallest elements

  (act locally think globally)

  distributes them to the PEs



Simple Probabilistic Properties

With high probability (whp):

here $\geq 1 - p^{-c}$ for a constant c we can choose

- whp only $\mathcal{O}\left(\frac{\log p}{\log\log p}\right)$ elements per local queue during insertion
- whp, the $\mathcal{O}(\log p)$ smallest elements of each local queue together contain the p globally best elements
- whp no local queue contains more then $\mathcal{O}(n/p + \log p)$ elements

Proof: Chernoff-bounds again-and-again.

(standard setting, balls-into-bins)

Parallel Implementation I

Insert

Sending: T_{coll}

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

(Better with “advanced” local queues Careful: amortized bounds are not sufficient.)

Parallel Implementation I

deleteMin*

Procedure deleteMin*(Q_1, p)

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

M:= select(Q_0, p) // later

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

Analysis

Remove locally: $\mathcal{O}\left(\log p \log \frac{n}{p}\right)$

Selection: $\mathcal{O}(T_{\text{coll}})$ whp todo

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

Deliver results: $\mathcal{O}(T_{\text{coll}})$ (random sources)

Verify: $\mathcal{O}(T_{\text{coll}}) +$

(sth polynomial in p)$\cdot$(a polynomially small probability)

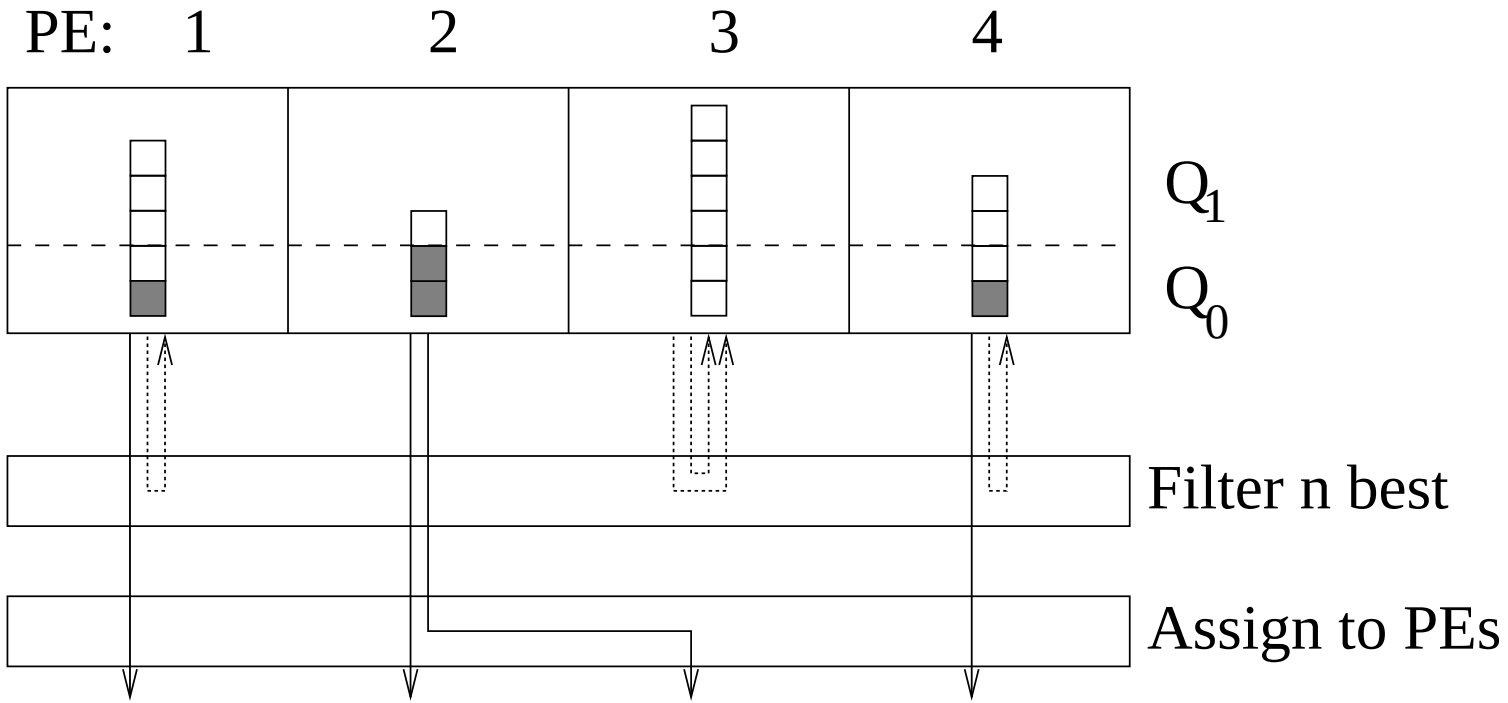
Insert locally: $\mathcal{O}\left(\log p \log \frac{n}{p}\right)$

Parallel Implementation II

Idea: Avoid ping-pong of the $\mathcal{O}(\log n)$ smallest elements.

Split the queue into Q_0 and Q_1.

Invariant: whp $|Q_0| = \mathcal{O}(\log p)$



Parallel Implementation II

Insert

Send: T_{coll}

Insert locally: merge Q_0 and new elements

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

Cleanup: Empty Q_0 every $\log p$ iterations.

$\rightsquigarrow$ cost $\mathcal{O}\left(\log p \log \frac{n}{p}\right)$ per $\log p$ iterations

$\rightsquigarrow$ average costs $\mathcal{O}\left(\log \frac{n}{p}\right)$

Parallel Implementation 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) // later

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

  assign e_i to PE i // use prefix sums

Analysis

Remove locally: $\mathcal{O}(1)$ expected iterations $\rightsquigarrow \mathcal{O}\left(T_{\text{coll}} + \log \frac{n}{p}\right)$

Selection: $\mathcal{O}(T_{\text{coll}})$ whp todo

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

Deliver results: $\mathcal{O}(T_{\text{coll}})$ (random sources)

Result

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

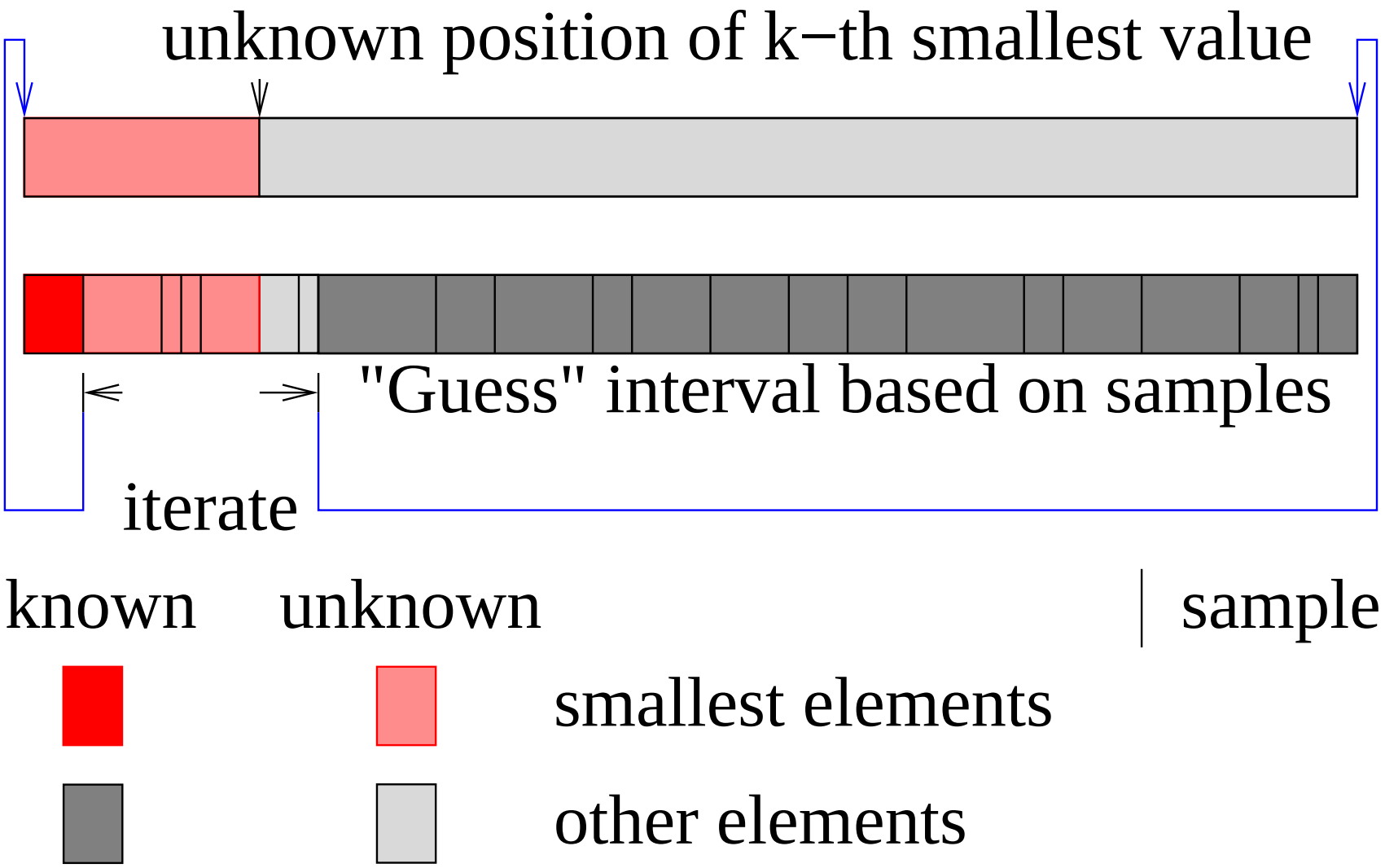
deleteMin*: expected $\mathcal{O}\left(T_{\text{coll}} + \log \frac{n}{p}\right)$

Randomized Selection [Blum et al. 1972]

Given n (randomly allocated) elements Q, find the k smallest ones.

- choose a sample $\mathbf{s}$
- u:= element with rank $\frac{k}{n}|\mathbf{s}| + \Delta$ in $\mathbf{s}$.
  ℓ:= element with rank $\frac{k}{n}|\mathbf{s}| - \Delta$ in $\mathbf{s}$
- Partition Q into
  $Q_< := \{q \in Q : q < \ell\}$,
  $Q_> := \{q \in Q : q > u\}$,
  $Q' := Q \setminus Q_< \setminus Q_>$
- If $|Q_<| < k$ and $|Q_<| + |Q'| \geq k$, output $Q_<$ and find the $k - |Q_<|$ smallest elements of Q'
- All other cases are unlikely if $|\mathbf{s}|$, Δ are sufficiently large.

Randomized Selection [Blum et al. 1972]



Parallel Implementation

- $|\mathbf{s}| = \sqrt{p} \rightsquigarrow$ sample can be sorted in time $\mathcal{O}(T_{\text{coll}})$.
- $\Delta = \Theta\left(p^{1/4+\varepsilon}\right)$ for a small constant ε.
  This makes difficult cases unlikely.
- No elements are redistributed. Random initial distribution guarantees good load balance whp.
- Whp a constant number of iterations suffices until only $\sqrt{p}$ elements are left. $\rightsquigarrow$ Then sort directly.

Overall expected time $\mathcal{O}\left(\frac{n}{p} + T_{\text{coll}}\right)$

Parallel Priority Queues – Refinements

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) // later

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

  assign e_i to PE i // use prefix sums

Or just use sufficiently many locally smallest elements and check later

Parallel Priority Queues – Refinements

- Mergable priority queues?
- Bulk delete after flush?
- Larger samples
- Remove larger batches?
- Only a subset of the PEs work as PQ-server?

Selection by pruned merging:

A reduction with vector length $\mathcal{O}\left(\sqrt{p \log p}\right)$

Asynchronous Variant

Accept insertions but do not immediately insert.

Batched deleteMin in a buffer.

Access buffer using an asynchronous FIFO.

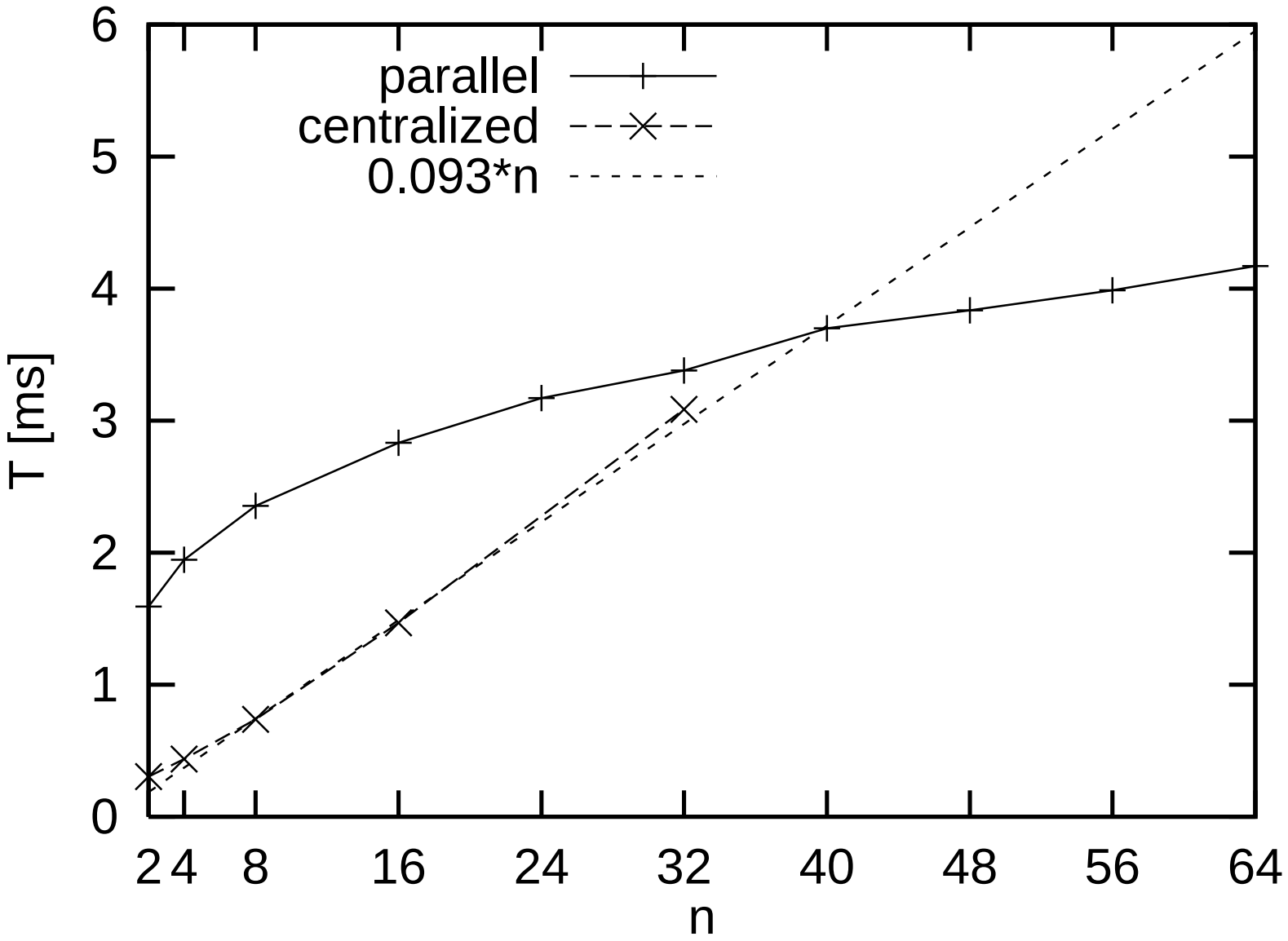
Sometimes:

Invalidate FIFO,

commit inserted elements

refill buffer

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



Implementation on Cray T3D, $m = 2^{24}$

$p = 256$

insert 256 elements and a deleteMin*:

centralized: > 28.16ms

parallel: 3.73ms

break-even at 34 PEs

More on parallel Priority Queues – History

Different approach starts with a binary heap:

nodes with p sorted Elements.

Invariant: All elements $>$ all elements in parent node

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

Elegant but expensive

Parallelization of a single access $\rightsquigarrow$ constant time with $\log n$ PEs.

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

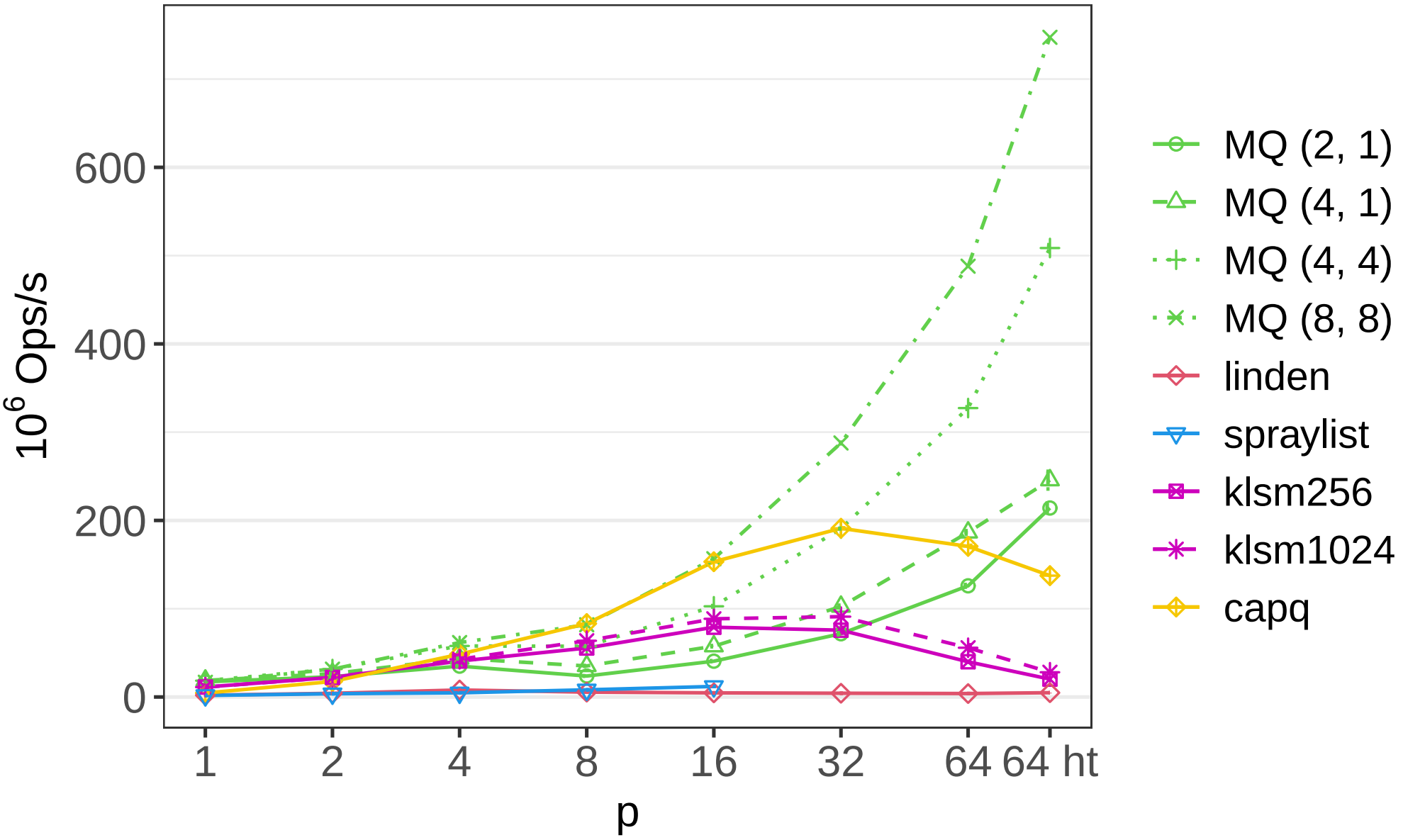
MultiQueues: Simple Relaxed Concurrent Priority Queues

With Roman Dementiev and Hamza Rihani, Marvin Williams

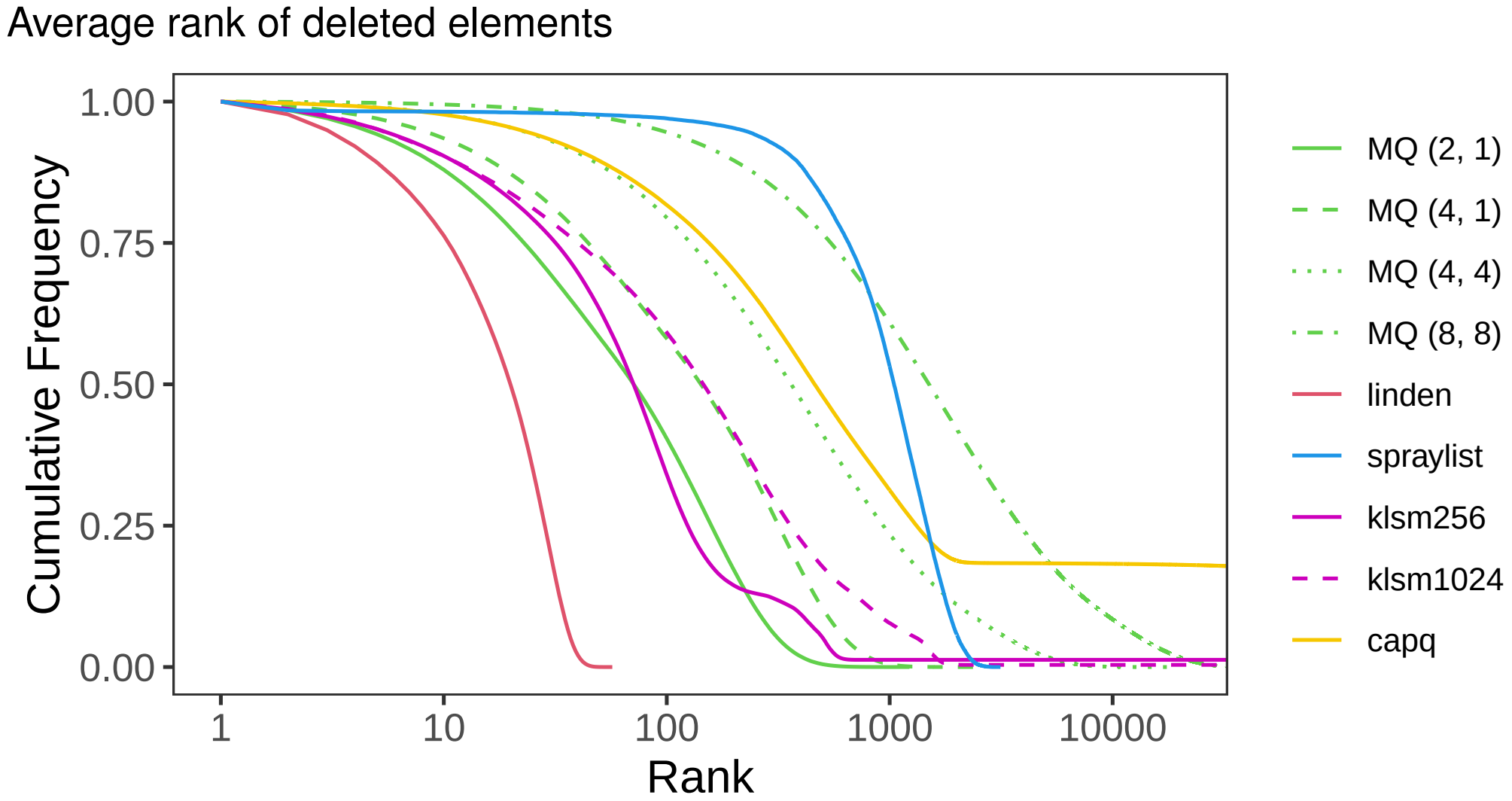
SPAA 2015, ESA 2021

- cp local queues $Q[1], \ldots, Q[cp]$, constant $c > 1$, e.g., $c = 2$
- Insert into random local queues (“wait-free” locking)
- Delete smallest elements from c randomly chosen queues
- Improve locality using insertion/deletion buffers
- Optional: Stickyness – stick to the same set of queues for s consecutive operations

Throughput

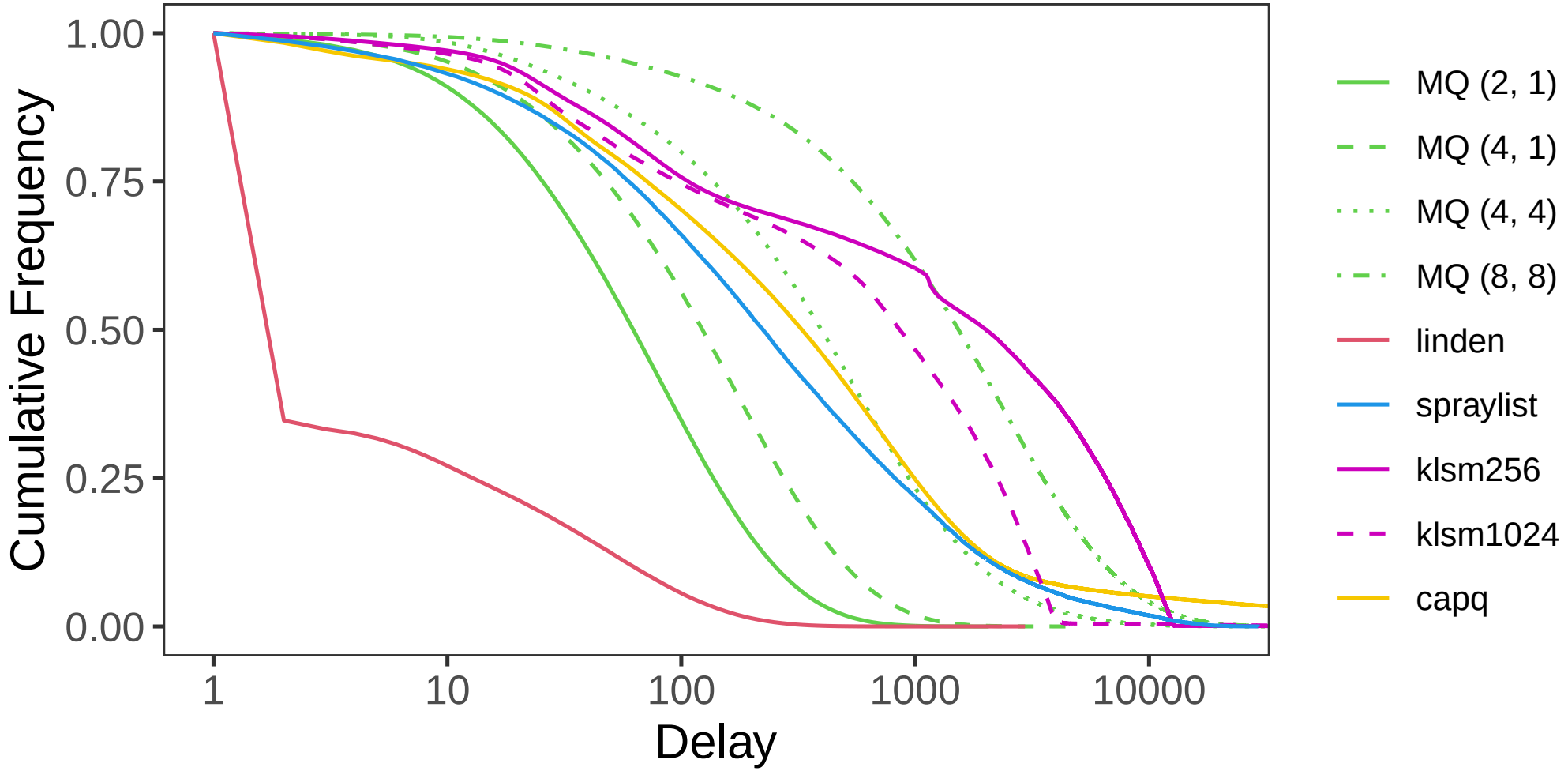

600
400
200
0
10^6 Ops/s
1
2
4
8
16
32
64
64 ht
p
MQ (2, 1)
MQ (4, 1)
MQ (4, 4)
MQ (8, 8)
linden
spraylist
klsm256
klsm1024
capq

Rank Error


Average rank of deleted elements
Cumulative Frequency
1.00
0.75
0.50
0.25
0.00
1
10
100
1000
10000
Rank
MQ (2, 1)
MQ (4, 1)
MQ (4, 4)
MQ (8, 8)
linden
spraylist
klsm256
klsm1024
capq

Delay

Average number of larger elements deleted before a removed element



List Ranking

Motivation:

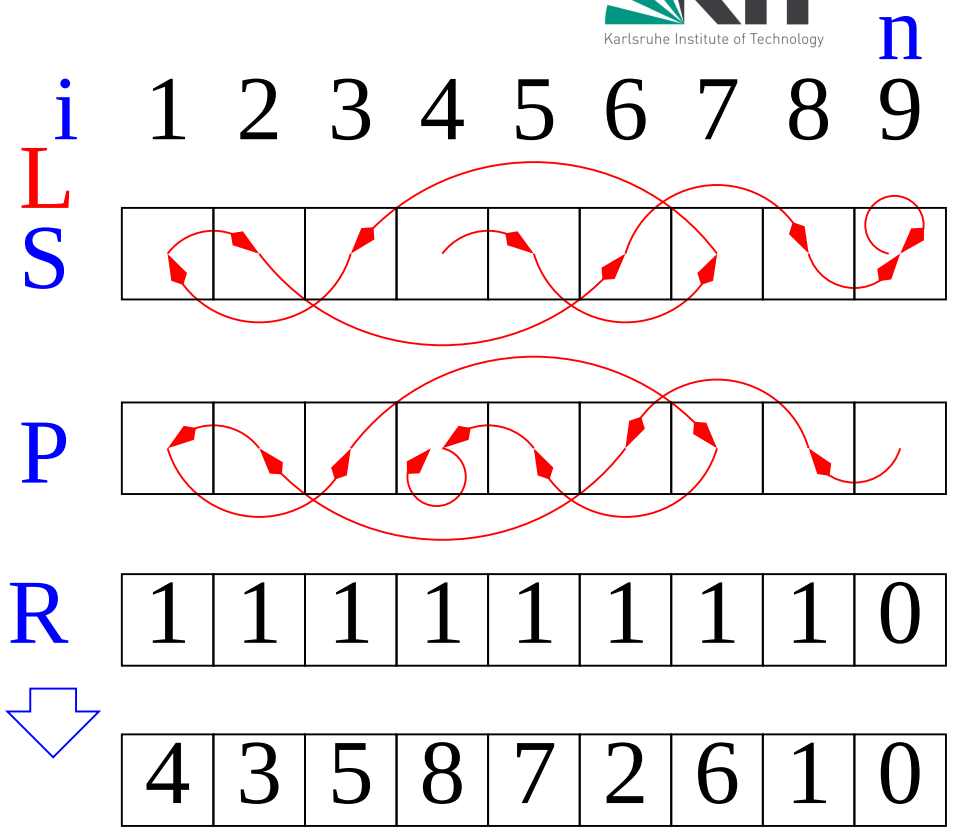
With arrays $a[1..n]$ we can do many things in parallel

- PE i processes $a[(i-1)\frac{n}{p}+1..i\frac{n}{p}]$
- Prefix sums
- ...

Can we do the same with linked lists?

Yes! For example, convert into an array

List Ranking



L: List

n: Elements

$S(i)$: Successor of element i

(unordered)

$S(i) = i$: at end of list

$P(i)$: Predecessor of element i

Exercise: compute in constant time for n PE PRAM

$R(i)$: Rank. Initially 1, 0 for last element.

Output: $R(i) =$ distance to the end (when following the chain).

Array-Conversion: store item i in $a(n - R(i))$

Motivation II

Lists are very simple graphs

$\rightsquigarrow$ warmup for graph algorithms

$\rightsquigarrow$ long paths are a main obstacle for parallelization.

I.e., solutions might help for more general graphs also (at least trees)?

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 $\mathcal{O}(n)$
- Time $\Theta(n)$

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

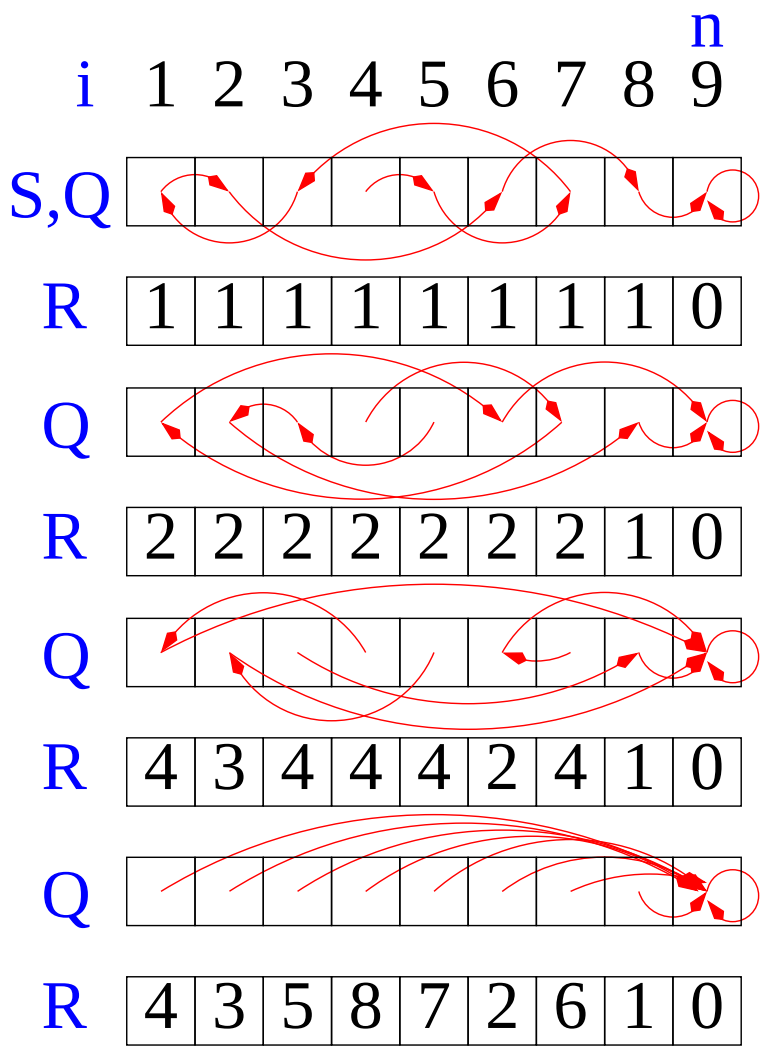
Q_i is the positions given by

chasing Q-pointers from pos i

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

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

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



Analysis

Induction Hypothesis: After k iterations

- $R(i) = 2^k$ or
- $R(i) =$ final result

Proof: True for $k = 0$.

$k \rightsquigarrow k+1$:

Case $R(i) < 2^k$: already final value (IH)

Case $R(i) = 2^k, R(Q(i)) < 2^k$: now final value (Invariant, IH)

Case $R(i) = R(Q(i)) = 2^k$: Now 2^{k+1}

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

Independent Set Removal

// 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** $f(S(i))$

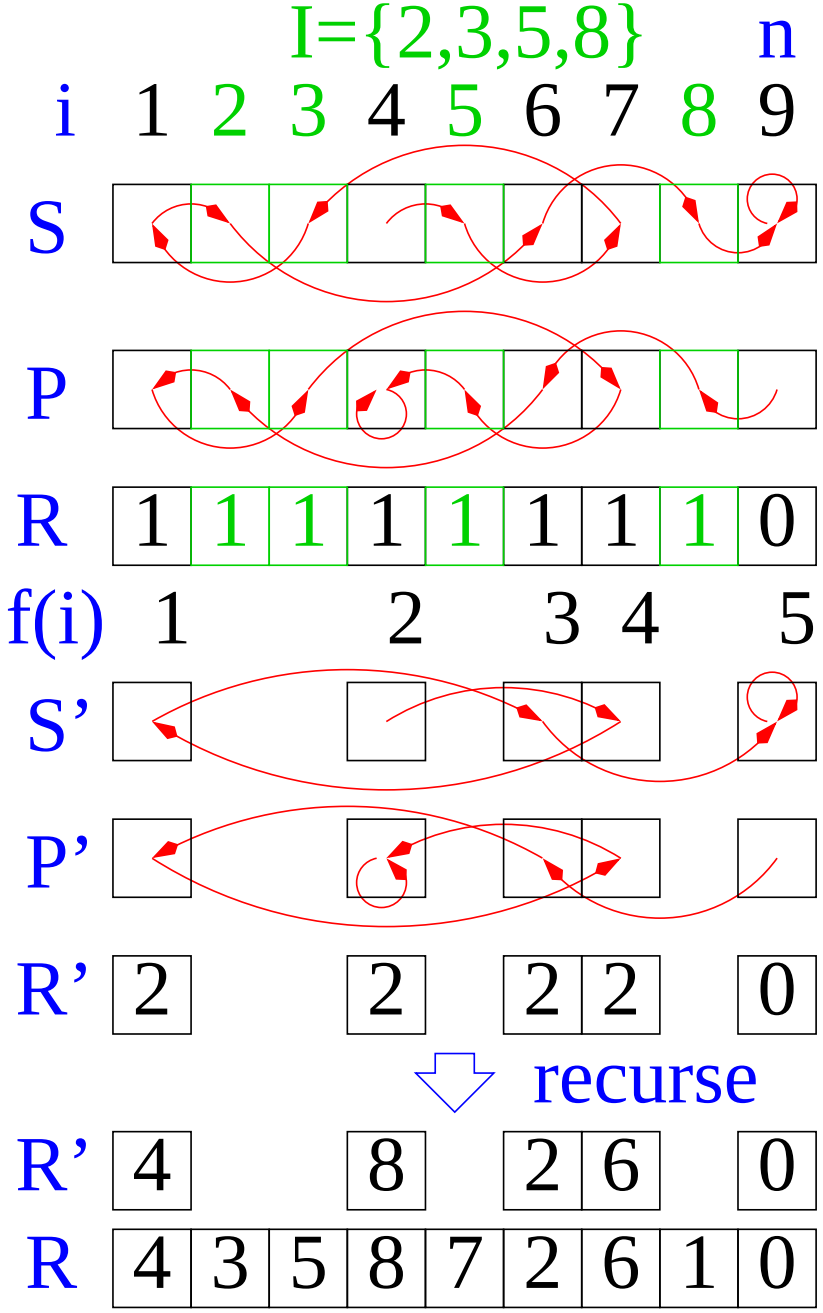
$P'(f(i)) := $ **if** $P(i) \in I$ **then** $f(P(P(i)))$ **else** $f(P(i))$

$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** $R(i) := R(i) + R'(f(S(i)))$

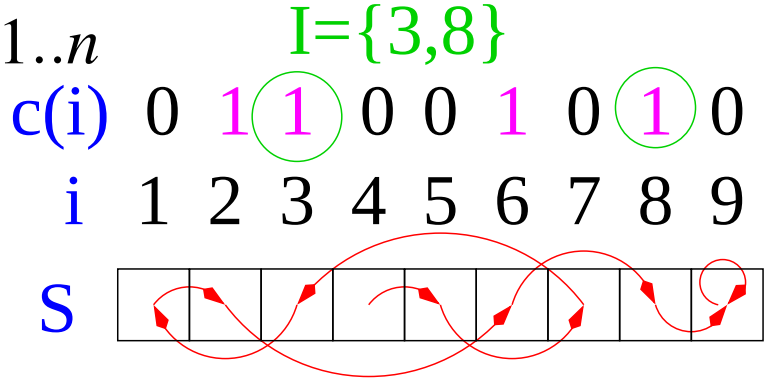

I={2,3,5,8}
n
i 1 2 3 4 5 6 7 8 9
S
P
R 1 1 1 1 1 1 1 1 0
f(i) 1 2 3 4 5
S'
P'
R' 2 2 2 2 0
recurse
R' 4 8 2 6 0
R 4 3 5 8 7 2 6 1 0

Finding Independent Sets

"Throw a coin" $c(i) \in \{0,1\}$ for each $i \in 1..n$

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

Expected size $|I| \approx \frac{n}{4}$



Monte Carlo algorithm $\rightsquigarrow$ Las Vegas algorithm:

repeat until $|I| > \frac{n}{5}$.

Expected time: $\mathcal{O}(n/p)$

Neither start nor end of list are in I.

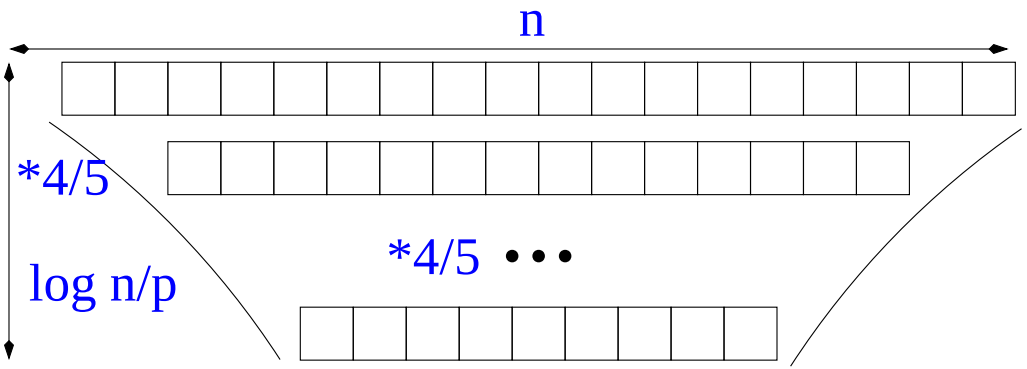
Finding a Bijective Mapping

Prefix sum over the indicator function of $\{1..n\} \setminus I$:

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

Analysis

- $T(n) = \mathcal{O}\left(\frac{n}{p} + \log p\right) + T\left(\frac{4}{5}n\right)$ in expectation
- $\mathcal{O}\left(\log \frac{n}{p}\right)$ levels of recursion
- Sum: $\mathcal{O}\left(\frac{n}{p} + \log \frac{n}{p} \log p\right)$ geometric sum
- Linear work, time $\mathcal{O}(\log n \log\log n)$ with $\frac{n}{\log n \log\log n}$ PEs



More on List Ranking

- Simple algorithm with expected time $\mathcal{O}(\log n)$
- Complicated algorithm with worst case Time $\mathcal{O}(\log n)$
- Many “applications” in PRAM algorithms
- Implementation on distributed-memory parallel computers [Sibeyn 97]: $p = 100$, $n = 10^8$, speedup 30.
- Generalization for segmented lists, trees
- Generalization for general graphs:
  contract nodes or edges
- Example for multilevel algorithms

Newer Implementation Results

- Cut the list at s random places
- Sequential algorithm for each sublist
- Recursive solution on instance of size s

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

Parallel Graph Algorithms

The „canonical“ „easy“ graph algorithms:

Main interest, sparse, polylog. execution time, efficient

- DFS
- BFS
- Shortest paths
  (nonnegative SSSP $\mathcal{O}(n)$ par. time. interesting for $m = \Omega(np)$)
  (what about APSP?)
- Topological sorting
+ Connected components (but not strongly connected)
+ Minimal spanning trees
+ Graph partitioning

Minimum Spanning Trees

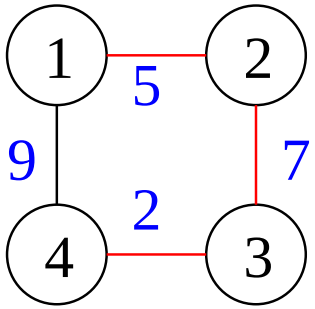
Undirected graph $G = (V, E)$

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.

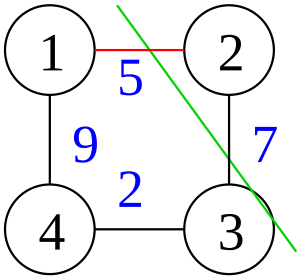
Selecting and Discarding MST Edges

The Cut Property

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

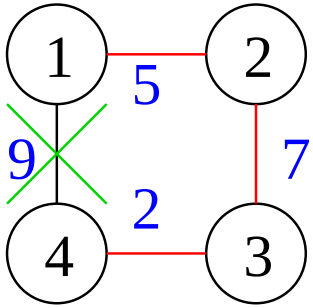
$$C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$$

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

$T := \emptyset$

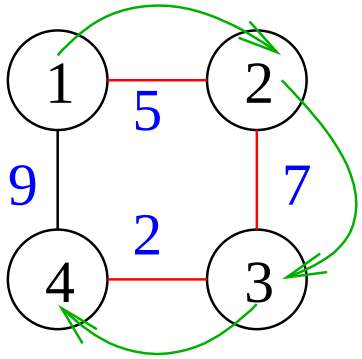
$S := \{s\}$ for arbitrary start node s

repeat $n-1$ times

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

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

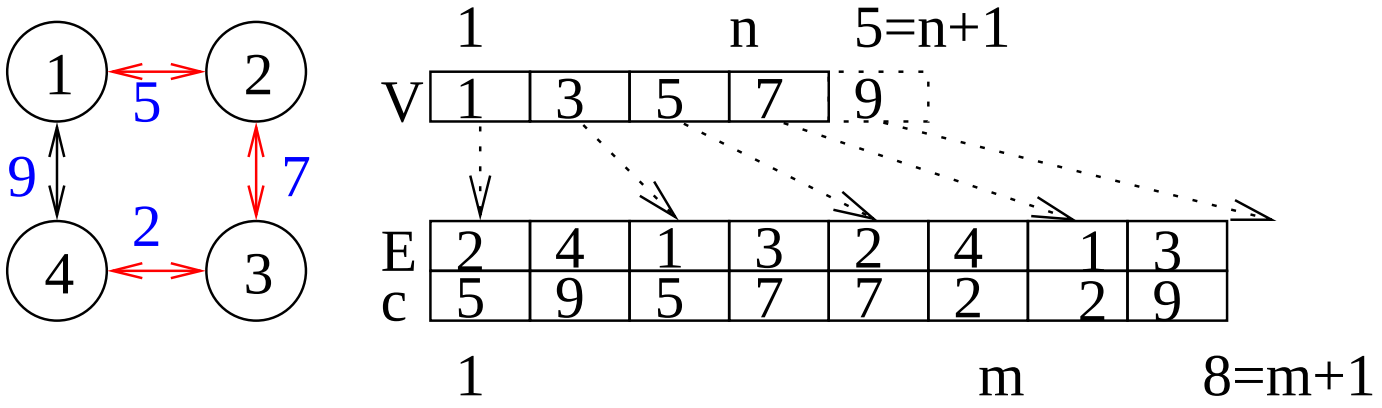
$T := T \cup \{(u,v)\}$



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

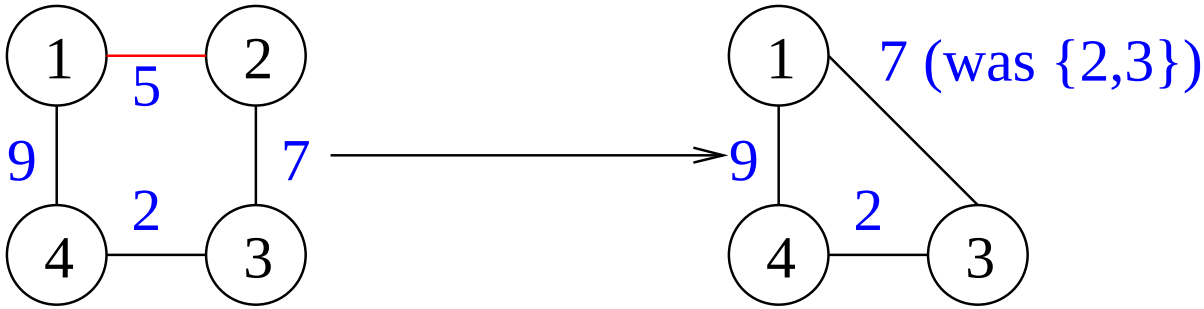
Edge Contraction

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]

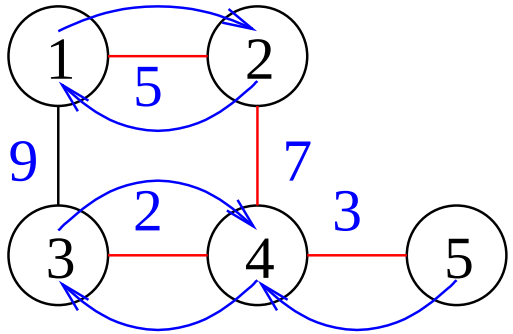
For each node find the lightest incident edge.

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**

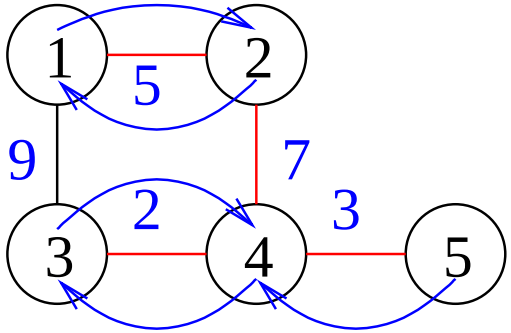
allocate $|\Gamma(v)|\frac{p}{2m}$ processors to node v // prefix sum

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

output original edge corresponding to (v,w)

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

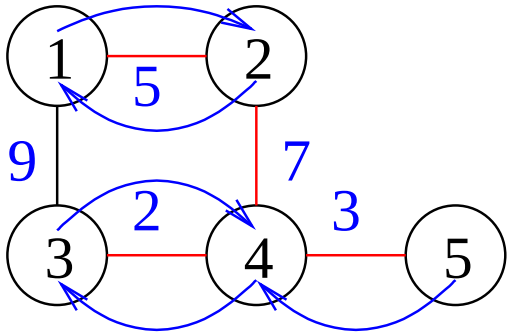
Time $\mathcal{O}\left(\frac{m}{p} + \log p\right)$



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



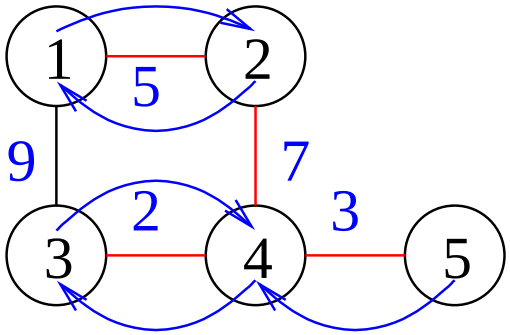
Pseudotrees $\rightarrow$ Rooted Trees

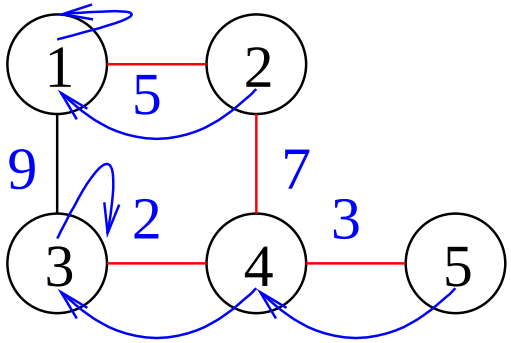
forall $v \in V$ **dopar**

w:= pred(v)

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

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



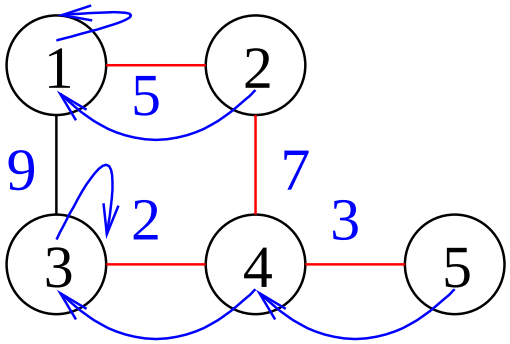


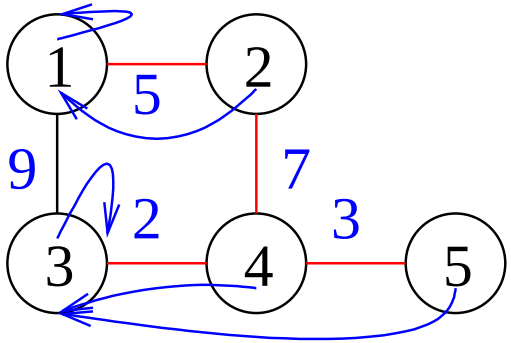
Rooted Trees $\rightarrow$ Rooted Stars by Doubling

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

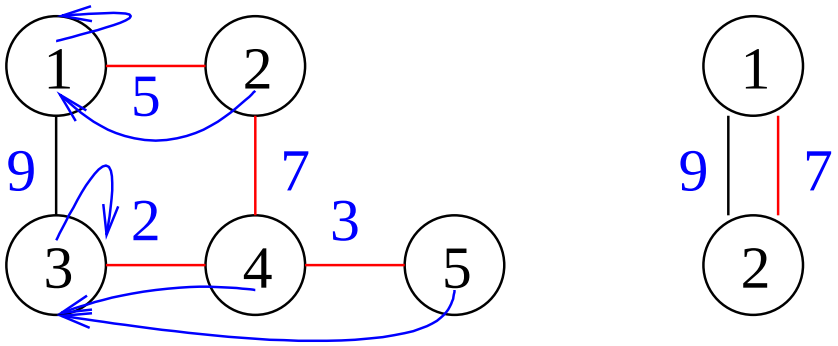
k:= #components

$V' = 1..k$

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:

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

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

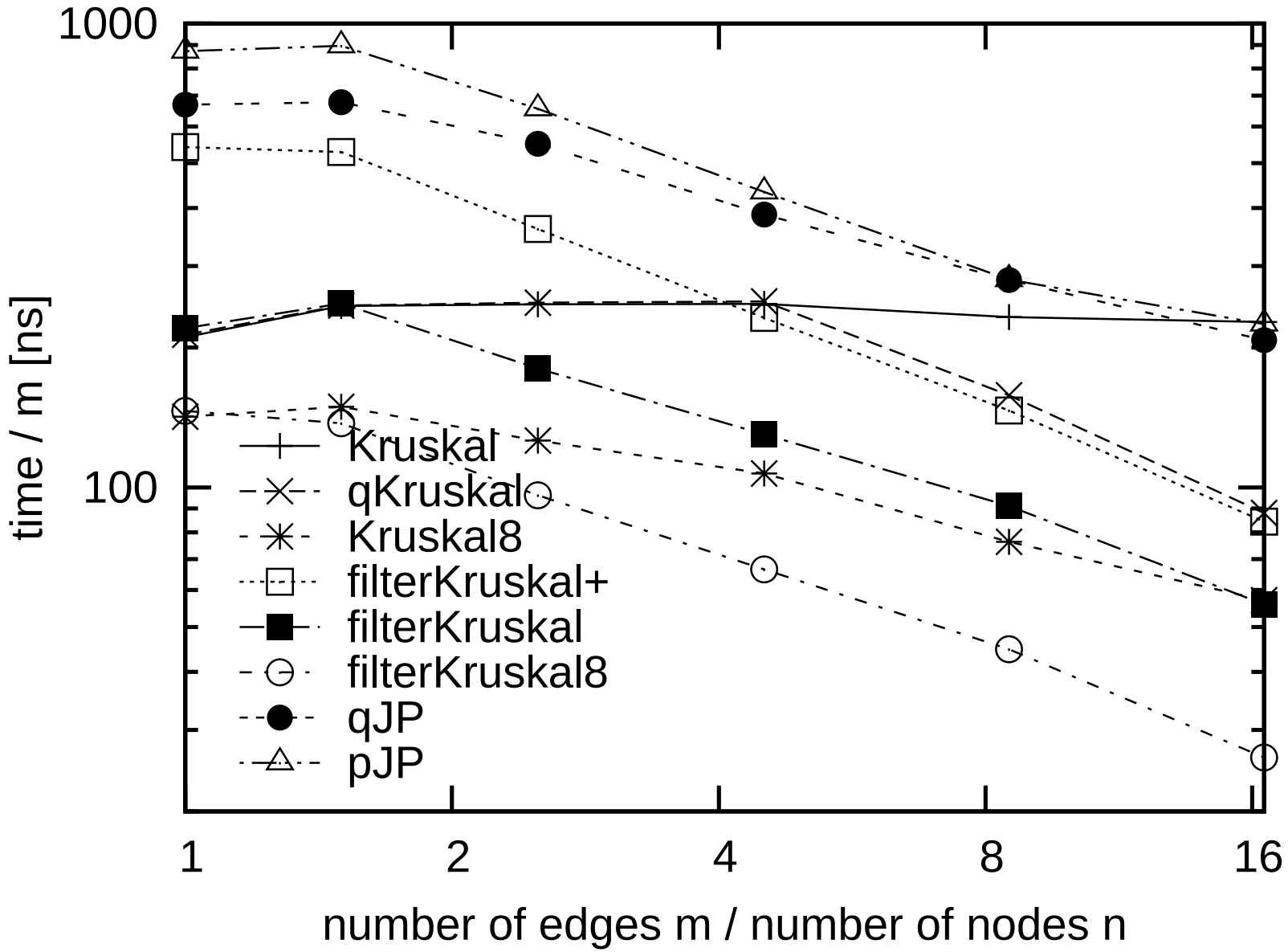
    filterKruskal$(E_{\leq}, T, P)$

    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.

Master thesis topic: communication-efficient distributed-memory MST

Load Balancing

[Sanders Worsch 97]

Given

- Work to be done
- PEs

Load balancing = assigning work $\rightarrow$ PEs

Goal: minimize parallel execution time

What we Already saw

- Estimating load using sampling — sample sort
- Assign approx. equal sized pieces — sample sort
- Multi sequence selection balances multiway merging
- Dynamic load balancing for quicksort and doall
- Prefix sums

  quicksort, parPQ, list ranking, MSTs,...
- Parallel priority queues — branch-and-bound

Measuring Cost

- Maximal Load: $\max_{i=1}^{p} \sum_{j \in \text{jobs @ PE } i} T(j, i, \ldots)$
- Time for finding the assignment
- Executing the assignment
- Cost for redistribution
- communication between jobs? (volume, locality?)

What is Known About the Jobs?

- ☐ Exact size
- ☐ Approximate size
- ☐ (Almost) nothing
- ☐ Further subdivisible?

Similar for communication costs

What is Kown About the Processors?

- All equal?
- Different?
- Fluctuating external load
- Tolerate faults?

Similar for communication resources

In this Lecture

- Independent jobs
  - Sizes exactly kown — fully parallel implementation
  - Sizes unknown or inaccurate — random assignment, master-worker-scheme, random polling

A very simple modell

- n jobs, $\mathcal{O}(n/p)$ per PE, independent, splittable, description with size $\mathcal{O}(1)$
- Size ℓ_i exactly known

Sequential 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

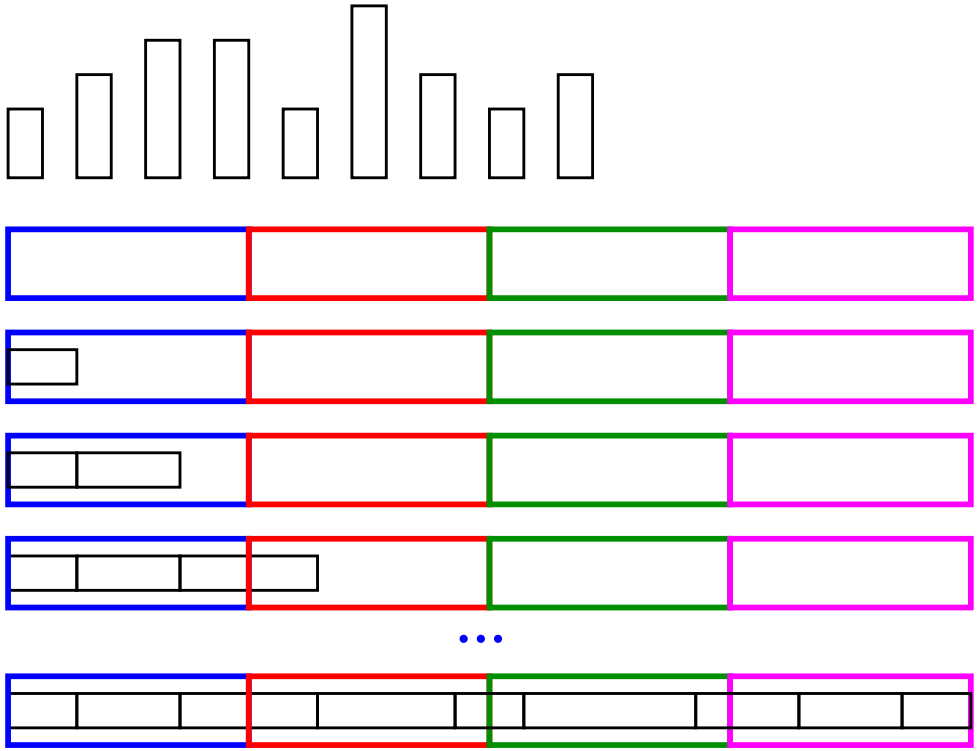
  $f := f - c$

  $\ell := \ell - c$

  if $f = 0$ **then** $i{+}{+}$; $f := C$ // next PE

  if $\ell = 0$ **then** $j{+}{+}$; $\ell := \ell_j$ // next job

Sequential Next Fit [McNaughton 59]



Parallelization of Next Fit (Sketch)

// 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

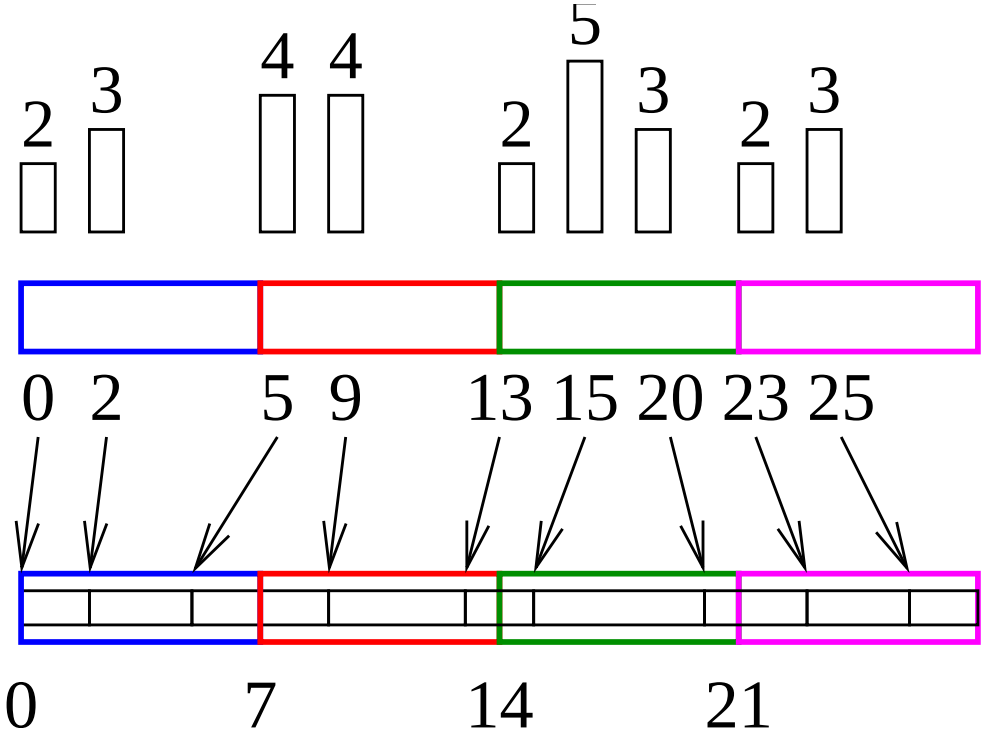
assign job j to PEs $\left\lfloor \frac{\text{pos}}{C} \right\rfloor .. \left\lfloor \frac{\text{pos}+\ell_j}{C} \right\rfloor$ // segmented broadcast

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

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

Time $C + \mathcal{O}\left(\frac{n}{p} + \log p\right)$ if jobs are initially distributed randomly.

Parallelization of Next Fit: Example



Atomic Jobs

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

Maximum load $\leq C + \max_j \ell_j \leq 2\text{opt}$

Better sequential approximation:

Assign largest jobs first

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

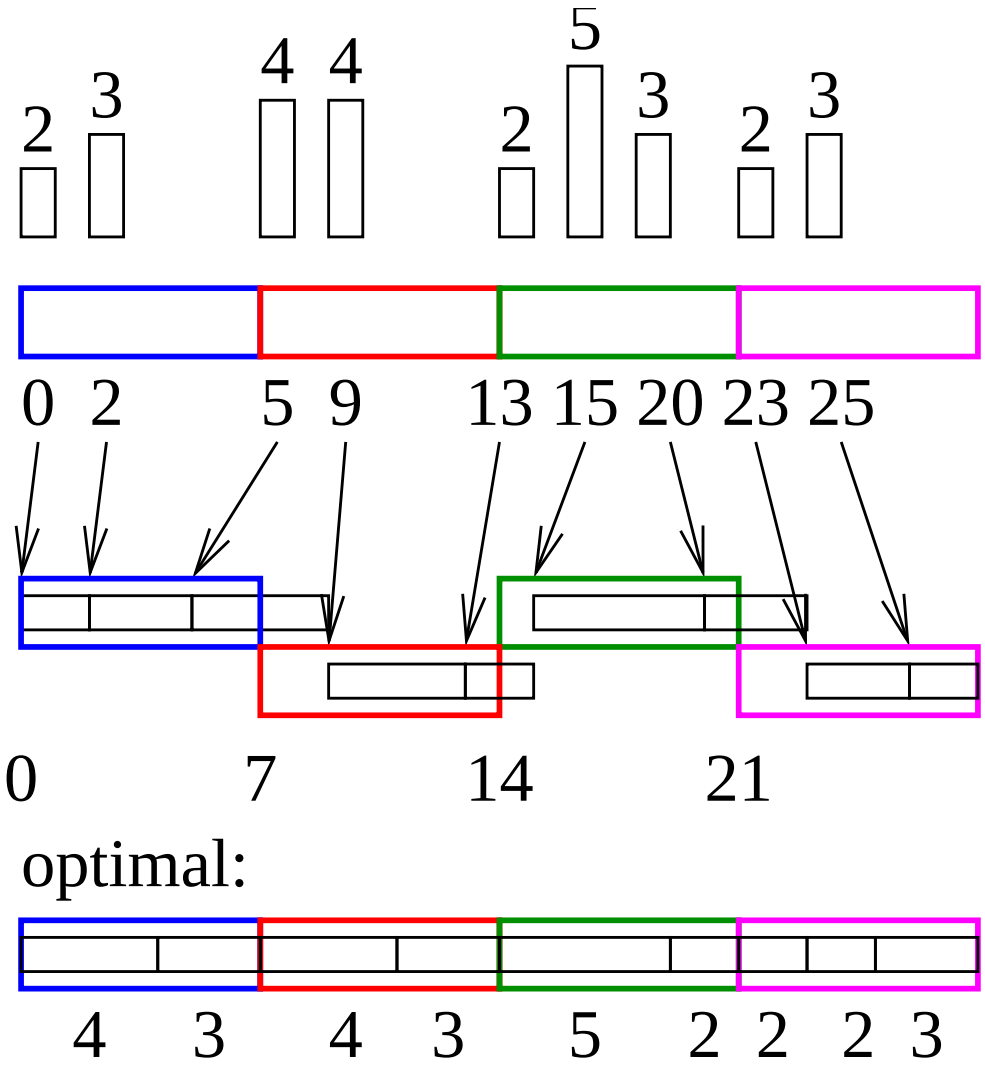
probably not parallelizable

Parallel

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

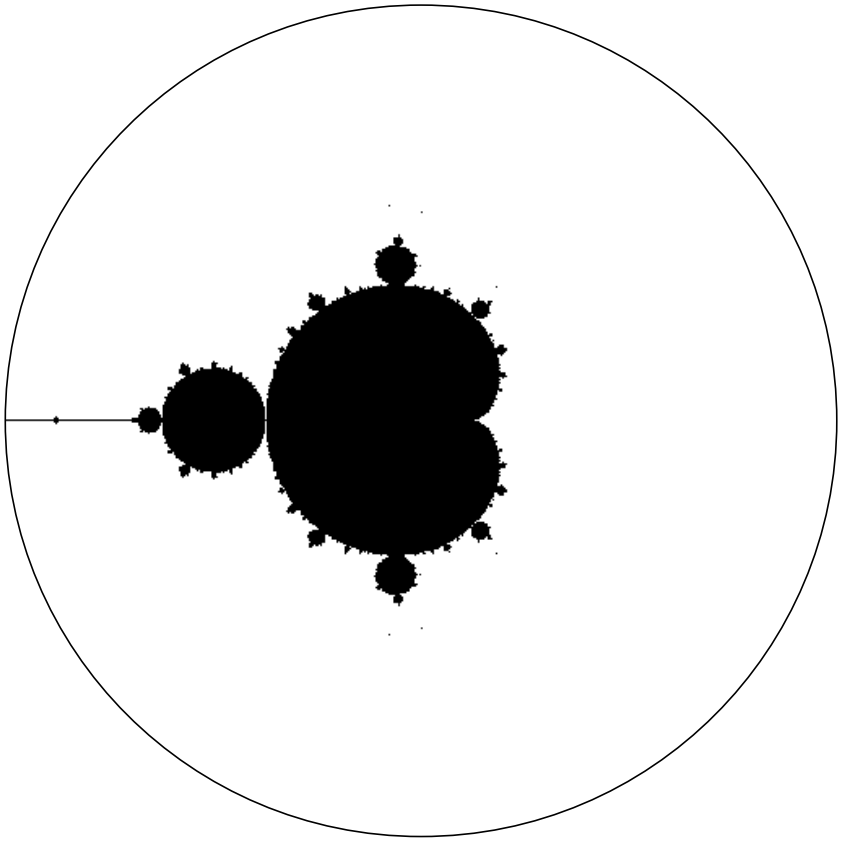
[Anderson, Mayr, Warmuth 89]

Atomic Jobs: Example



Example Mandelbrot Set

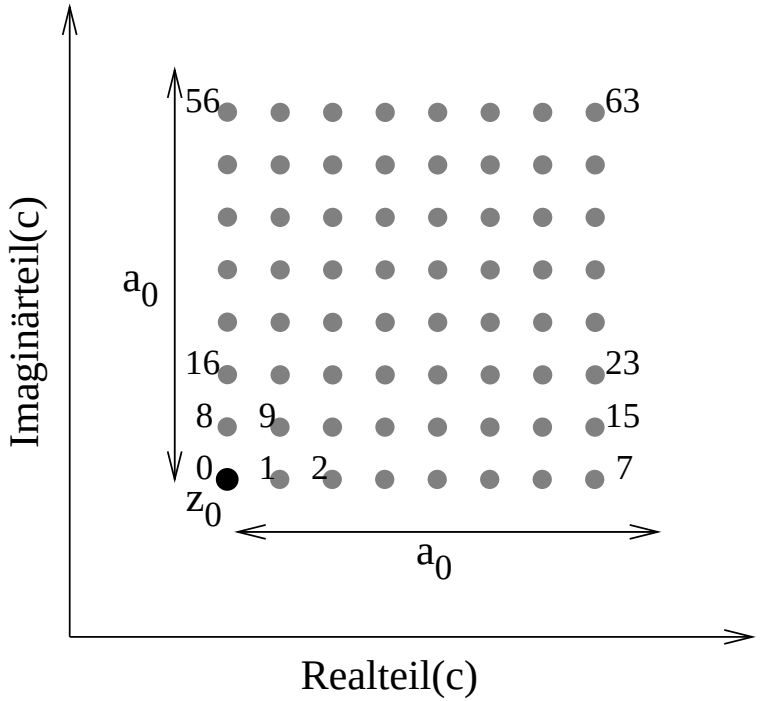
$$z_c(m) : \mathbb{N} \to \mathbb{C}$$
$$z_c(0) := 0, \quad z_c(m+1) := z_c(m)^2 + c$$
$$M := \{c \in \mathbb{C} : z_c(m) \text{ is bounded}\} \ .$$



Approximate Computation

- Computation only for a quadratic subset of the complex plane
- Computation only for a discrete grid of points
- z_c unbounded if $|z_c(k)| \geq 2$
- Stop after $m_{\max}$ iterations

Where is the load balancing problem?

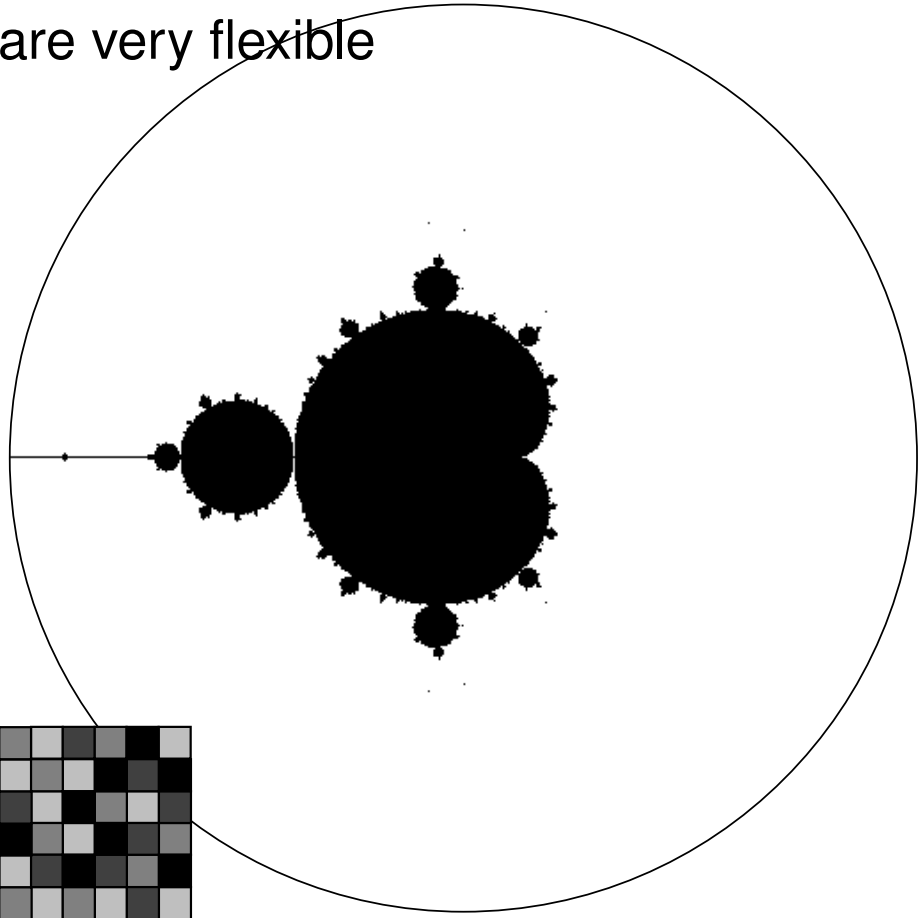


Code

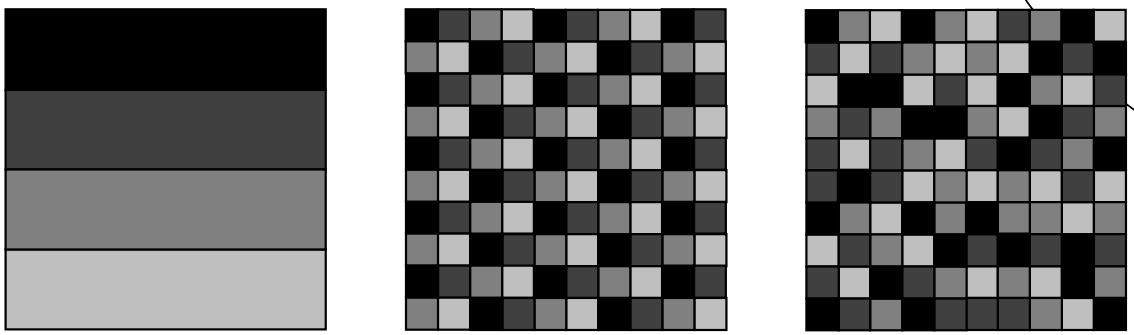
```
int iterate(int pos, int resolution, double step)
{ int iter;
  complex c =
    z0+complex((double)(pos % resolution) * step,
               (double)(pos / resolution) * step);
  complex z = c;
  for (iter = 1;
       iter < maxiter && abs(z) <= LARGE;
       iter++) {
    z = z*z + c;
  }
  return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

Static Apple Distribution

Since there is little communication, we are very flexible



- Split into strips
  - Why attractive?
  - Why rather not?
- Cyclic. Good. But provable??
- Random



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

Parallelize Assignment Phase

- If the jobs are arbitrarily distributed over the PEs: Random permutation via all-to-all. (see also sample sort)
- Implicit generation of jobs
  - Jobs can be generated based on a number $1 \ldots n$.
  - Problem: Parallel generation of a (pseudo) random permutation

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 safe in some cryptographical sense

Random Assignment

- Given: n jobs of size $\ell_1, \ldots, \ell_n$
- Let $L := \sum_{i \le n} \ell_i$
- Let $l_{\max} := \max_{i \le n} \ell_i$
- Assign the jobs to random PEs

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

then the maximum Load is at most $(1+\varepsilon)\frac{L}{p}$

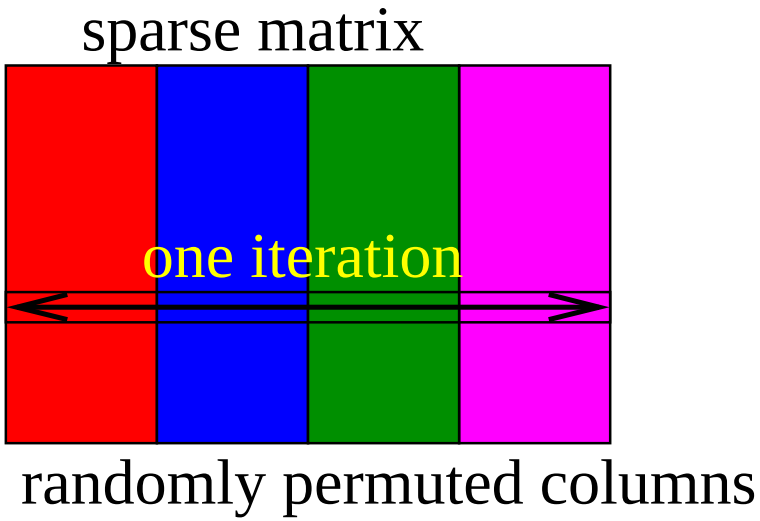
with probability at least $1 - p^{-\beta}$. Proof: ... Chernoff-Bounds...

Discussion

- \+ Job sizes need not been known at all
- \+ It is irrelevant where the jobs come from (distributed generation possible)
- − Inacceptable with big $l_{\max}$
- − Very good load balance only with large $L/l_{\max}$ (quadratic in $1/\varepsilon$, logarithmic in p).

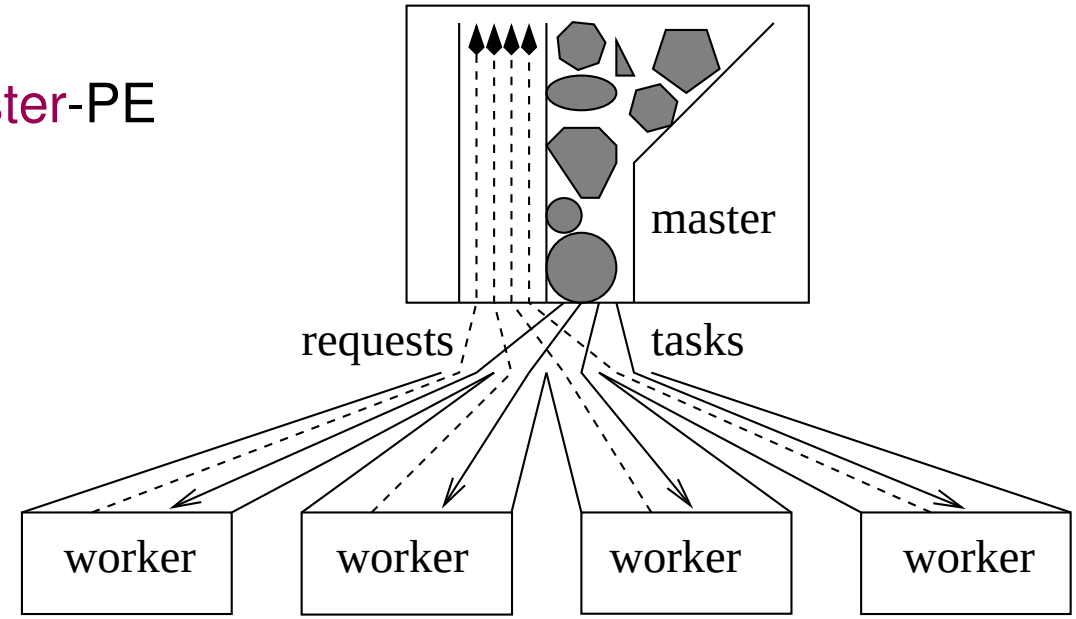
Application Example: Airline Crew Scheduling

A single random assignment solves k simultaneous load balancing problems. (Deterministically, this is probably a difficult problem.)



The Master-Worker-Scheme

- Initially all jobs are on a master-PE
- Job sizes can be estimated but is not exactly known
- Once a job is assigned, they cannot be further subdivided (nonpreemptive)

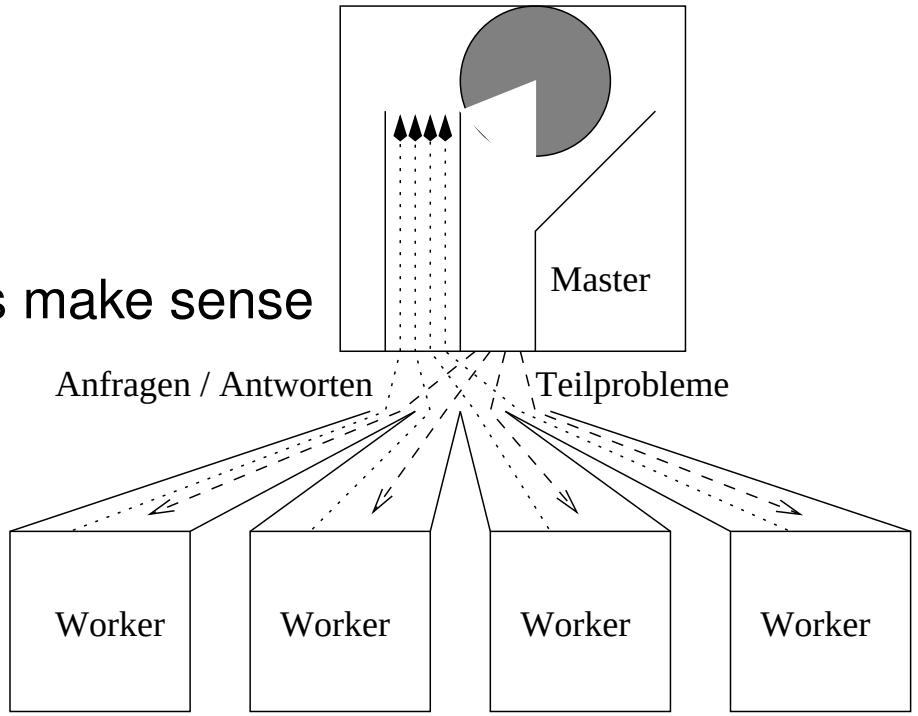


Discussion

- \+ Simple
- \+ Natural input-output scheme (but perhaps a separate disk slave)
- \+ Suggests itself when the job generator is not parallelized
- \+ Easy to debug
- − Communication bottleneck $\Rightarrow$ tradeoff communication cost versus imbalance
- − How to split?
- − Multi-level schemes are complicated and of limited help

Size of the jobs

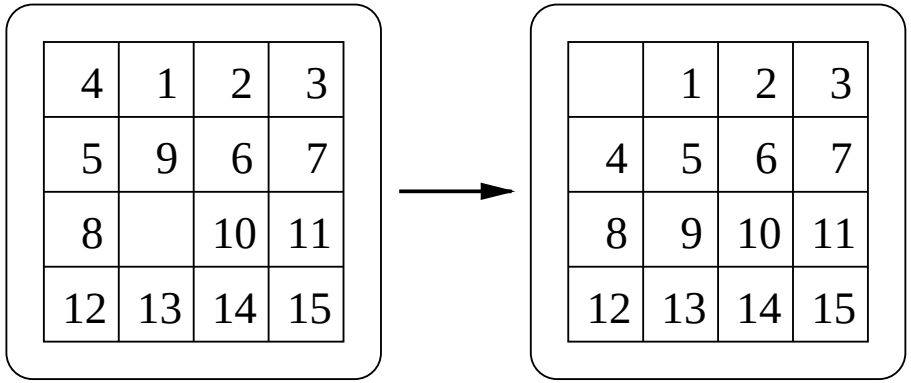
- Deal out jobs that are as large as possible as long as balance is not in danger. Why?
- Conservative criterion: upper bound for the size of the delivered jobs $\leq$
  $1/P$-th part of lower bound
  for system load.
  - Where to get size estimate?
- More aggressive approaches make sense



Work Stealing

- (Almost) arbitrarily subdivisible load
- Initially all the work on PE 0
- Almost nothing is known on job sizes
- Preemption is allowed. (Successive splitting)

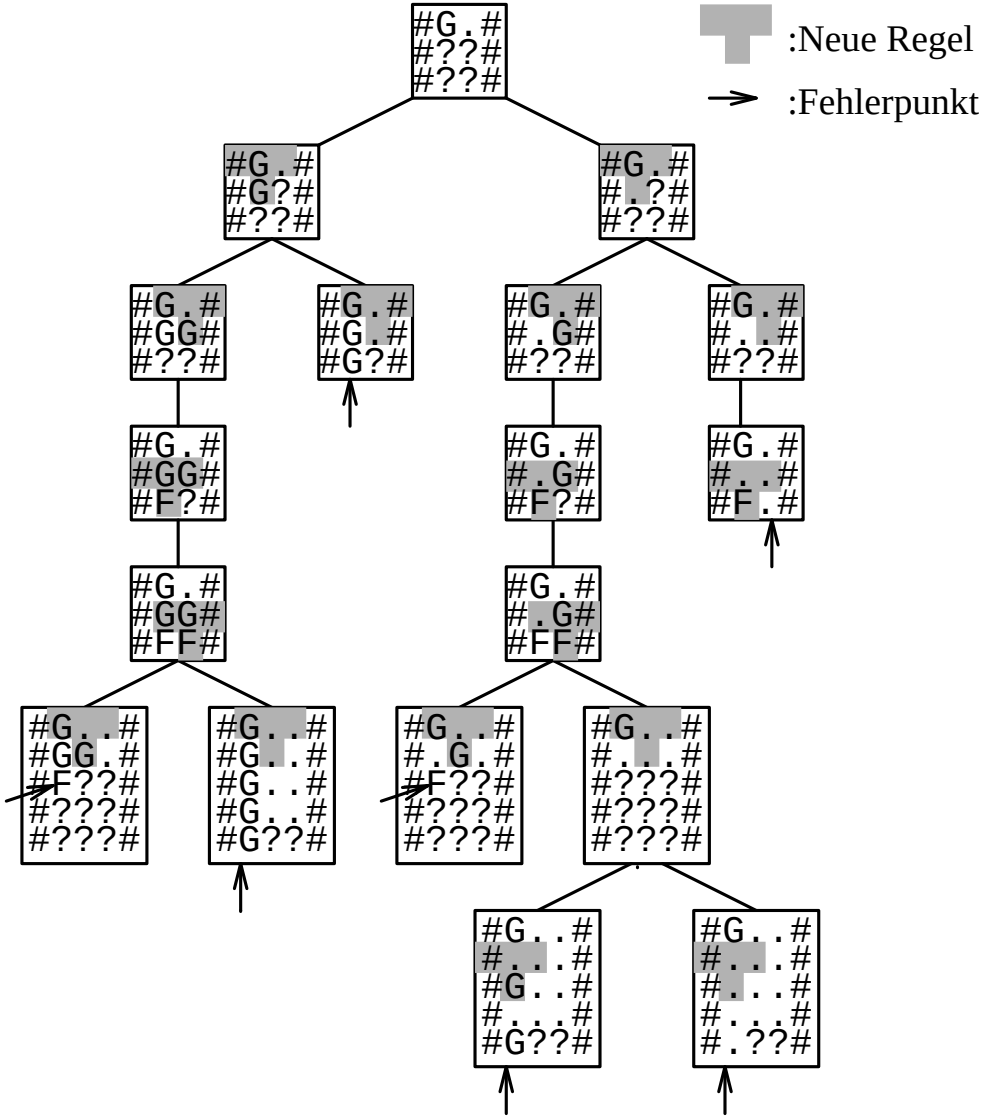
Example: The 15-Puzzle



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

Example: Firing Squad Synchronization Problem

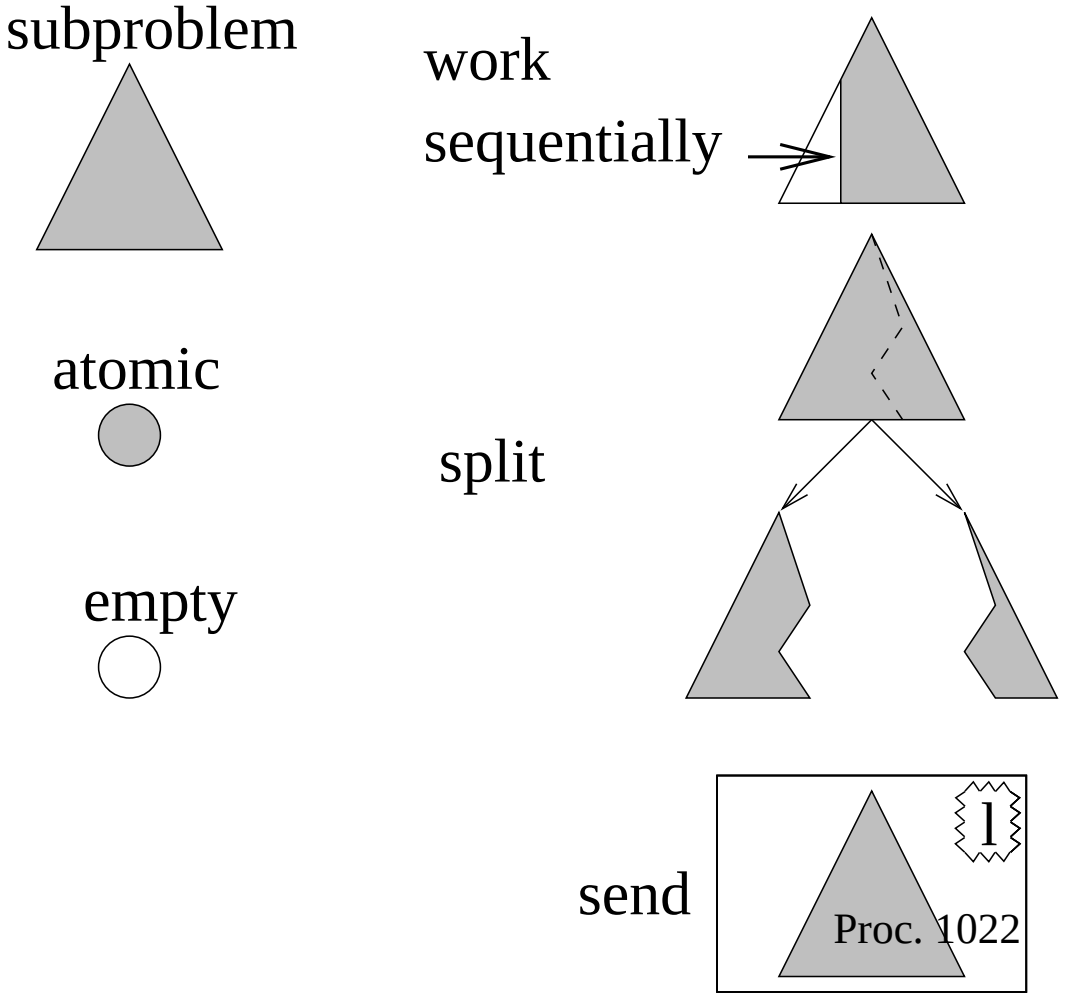
Backtracking over Transition Functions



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

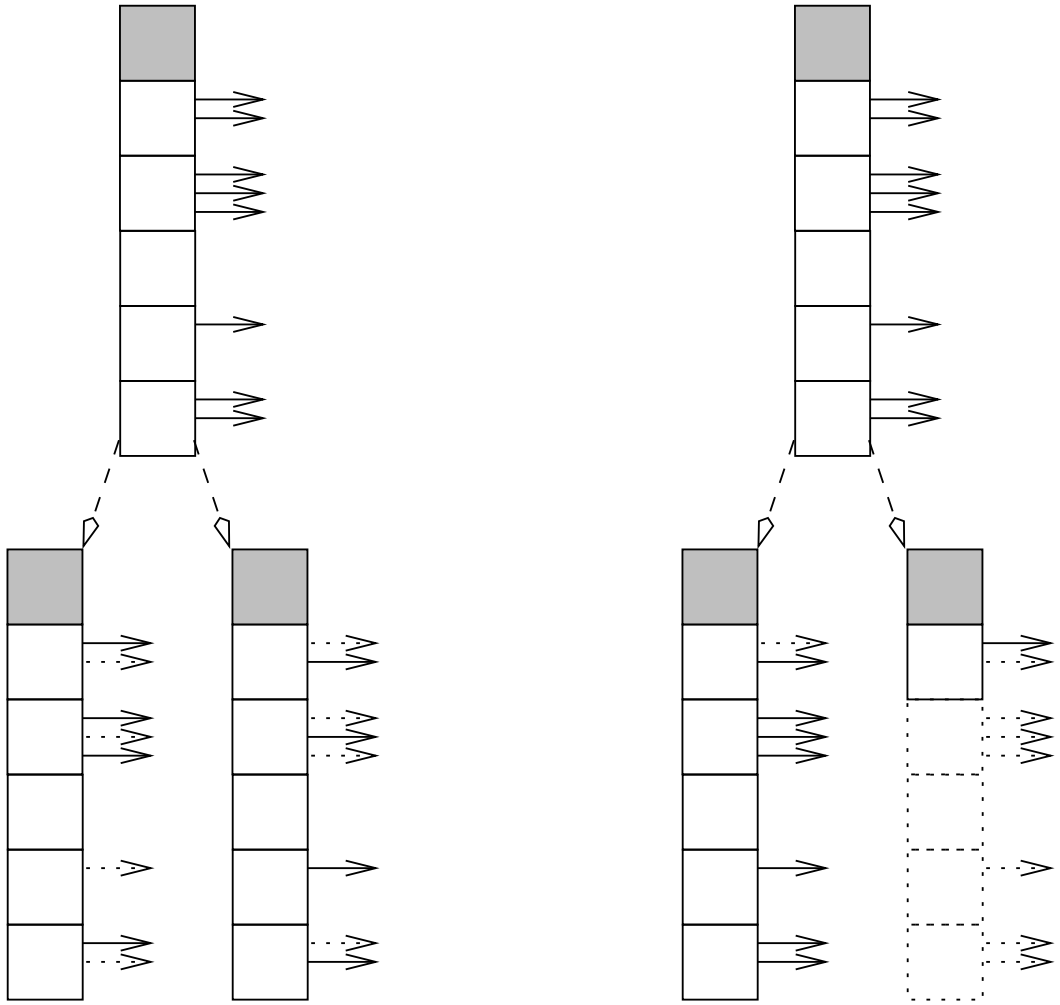
T_{rout}: time needed for communicating a subproblem $(\alpha + \ell\beta)$

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. $h \rightsquigarrow T_\infty$

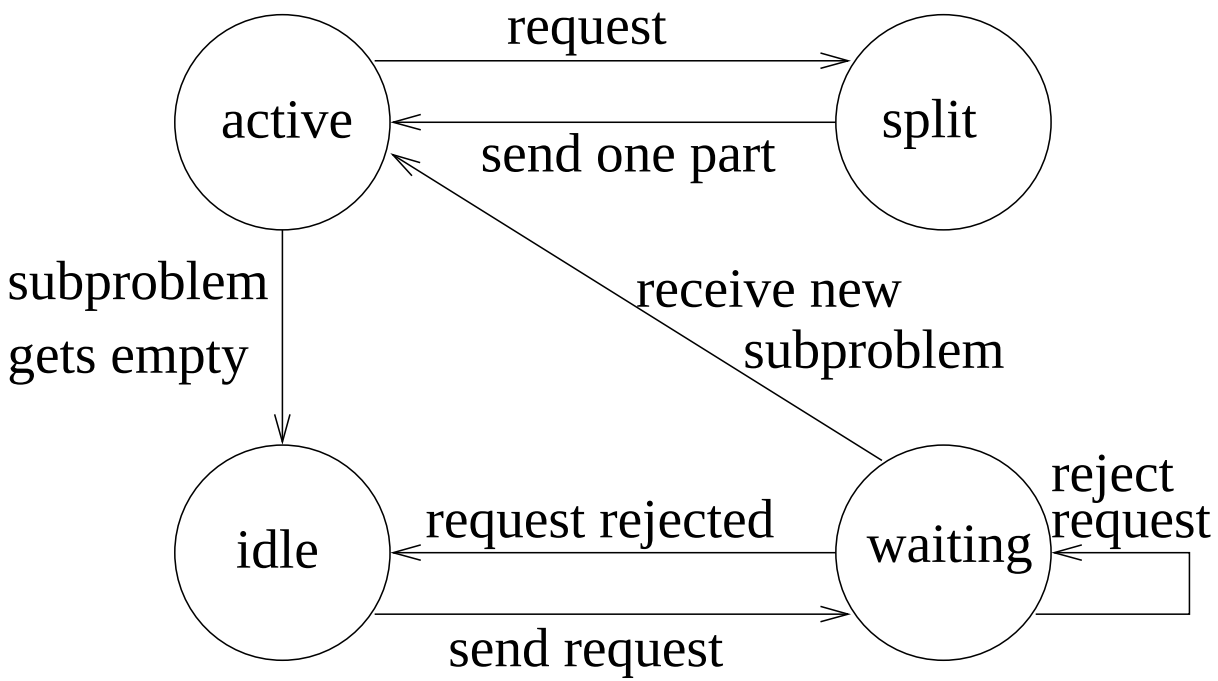
An Application List

- 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

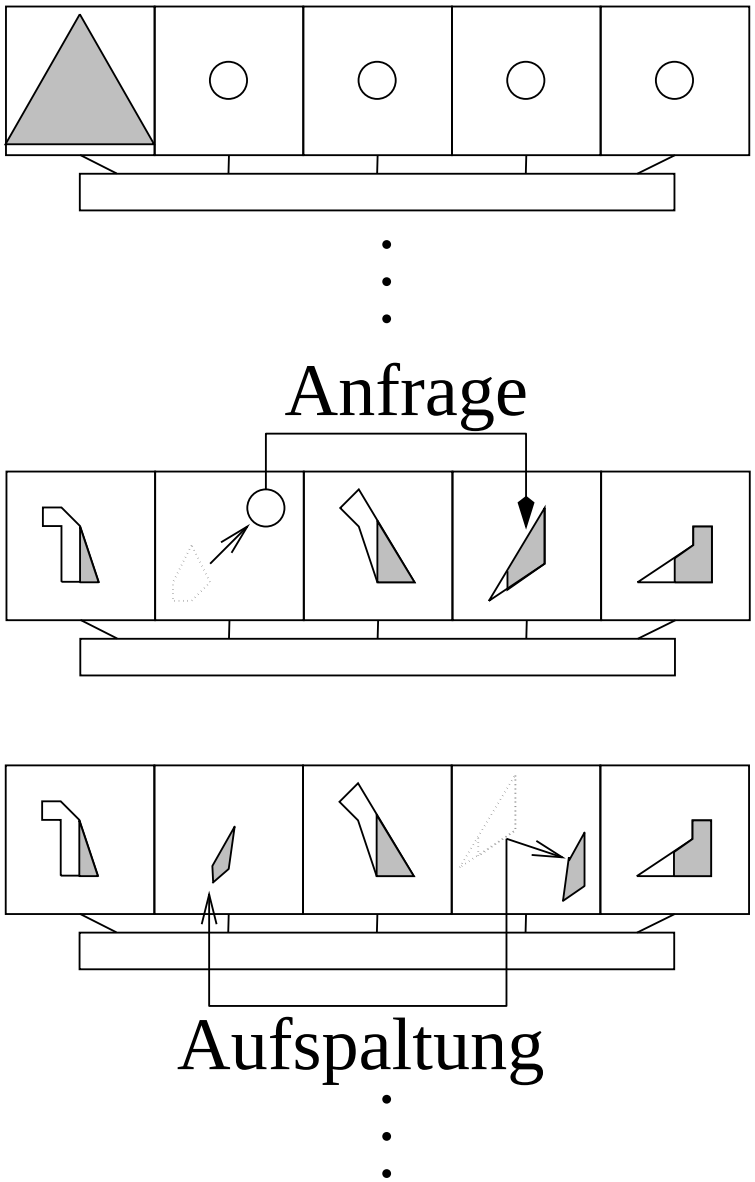
Limits of the Model

- 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

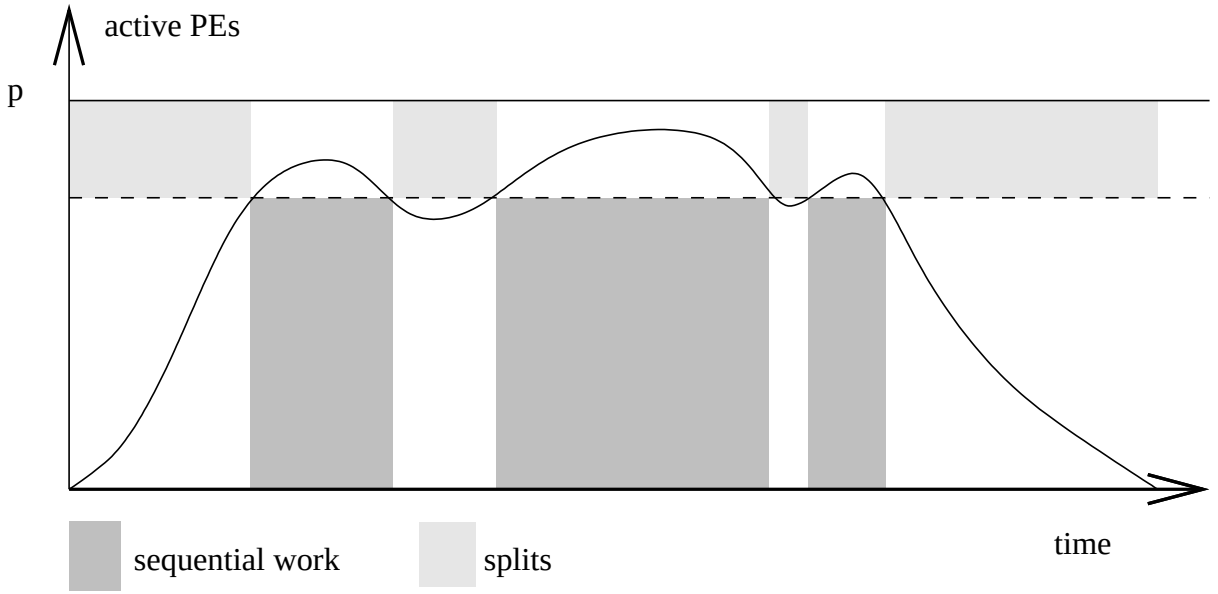
if $T(P) = 0$ **then** receive P

Analysis

Theorem:

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

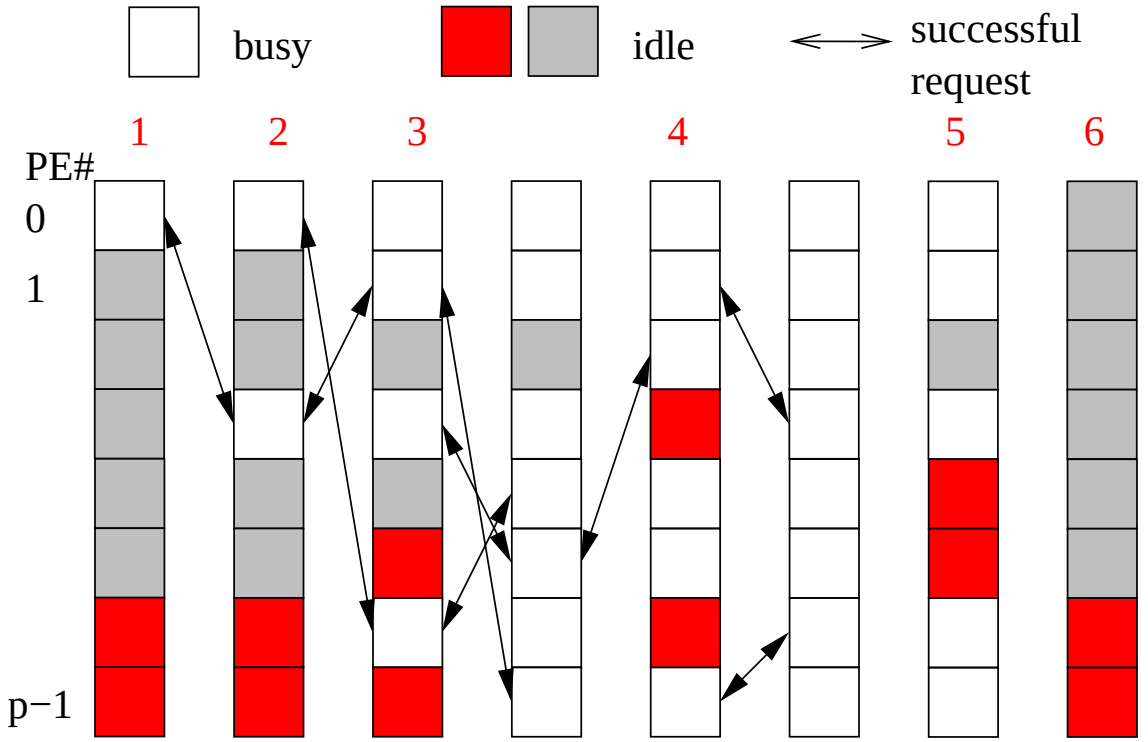
$$T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{p} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}})\right) \quad .$$



Bounding Idleness

Lemma 6.

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 7. *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 8. *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 9.

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

for an appropriate choice of Δt.

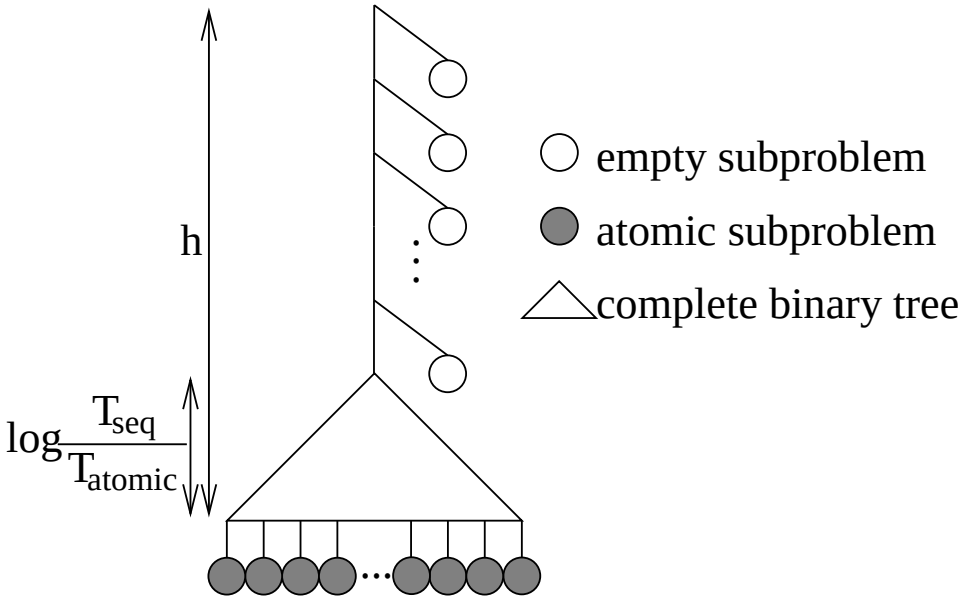
A Trivial Lower Bound

Satz 10. *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

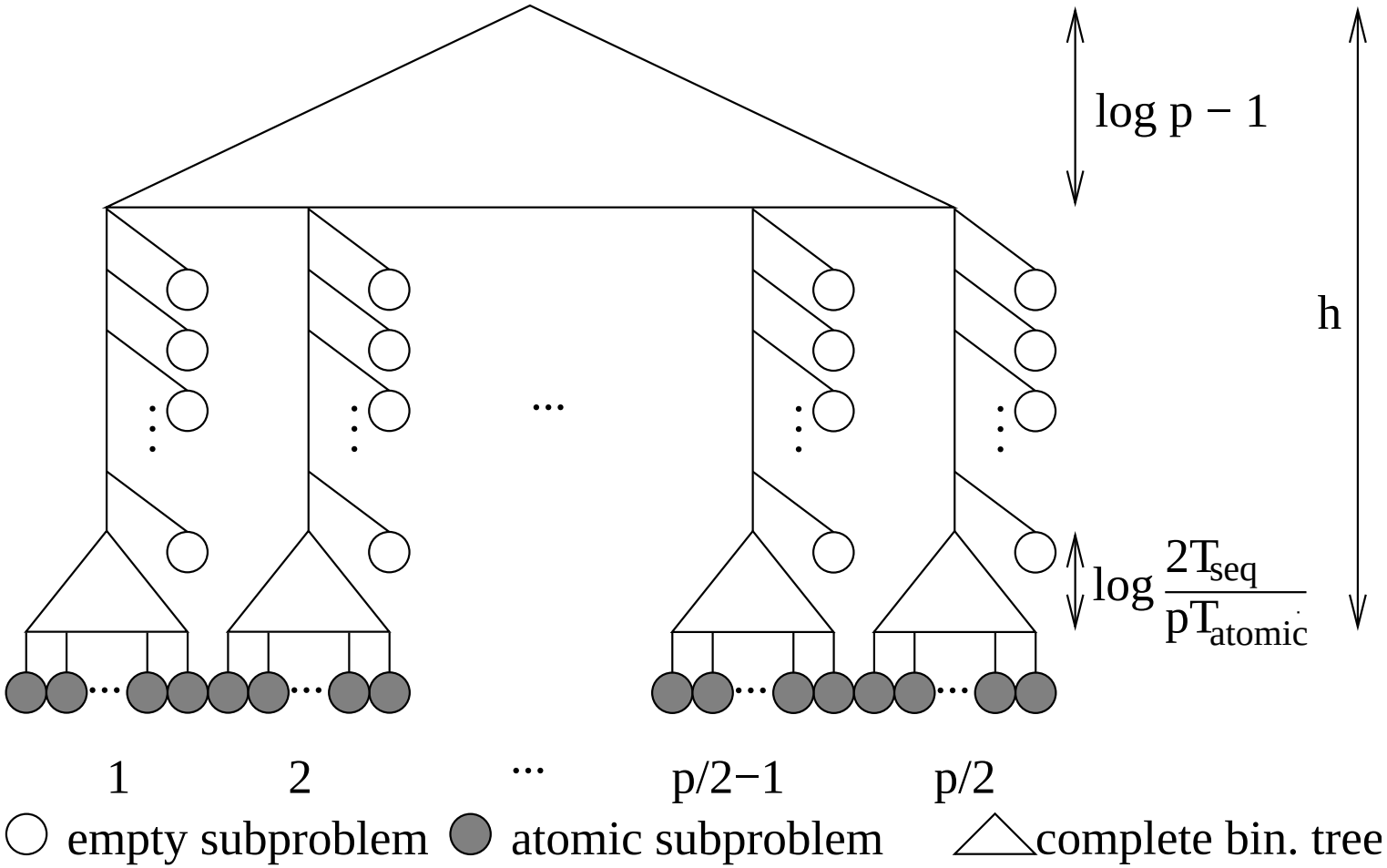


Additional

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

term.

Many Splits Overall



Satz 11. *Some problems need at least*

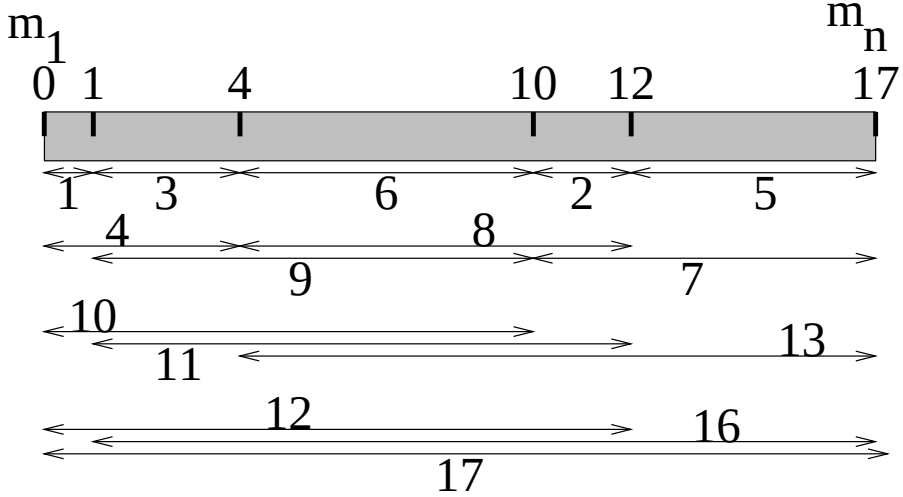
$$\frac{p}{2}\left(h - \log\frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)$$

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

Korollar 12. *Receiver initiated algorithms need a corresponding number of communications.*

Satz 13 (Wu and Kung 1991)**.** *A similar bound holds for all deterministic load balancers.*

Golomb-Rulers

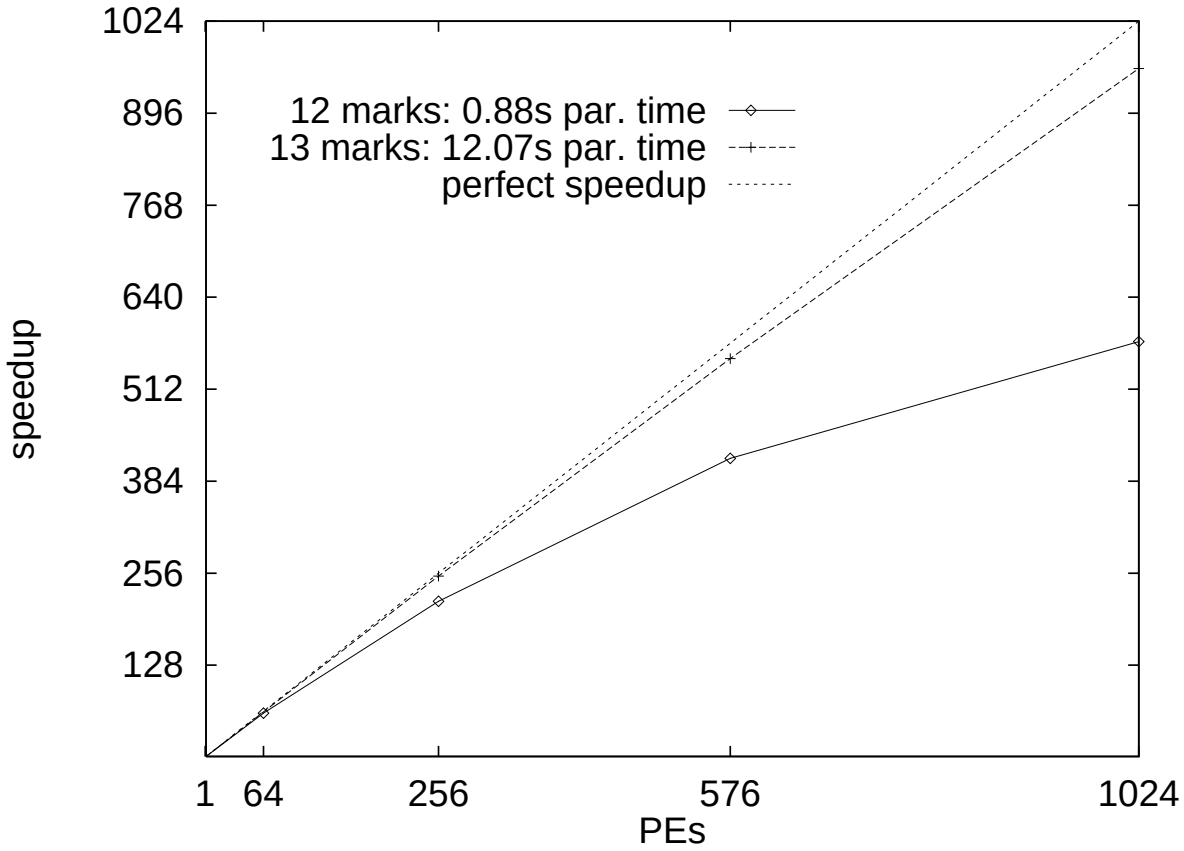


- Total length m
- find n marks $\{m_1, \ldots m_n\} \subseteq \mathbb{N}_0$
- $m_1 = 0, m_n = m$
- $|\{m_j - m_i : 1 \leq i < j \leq n\}| = n(n-1)/2$

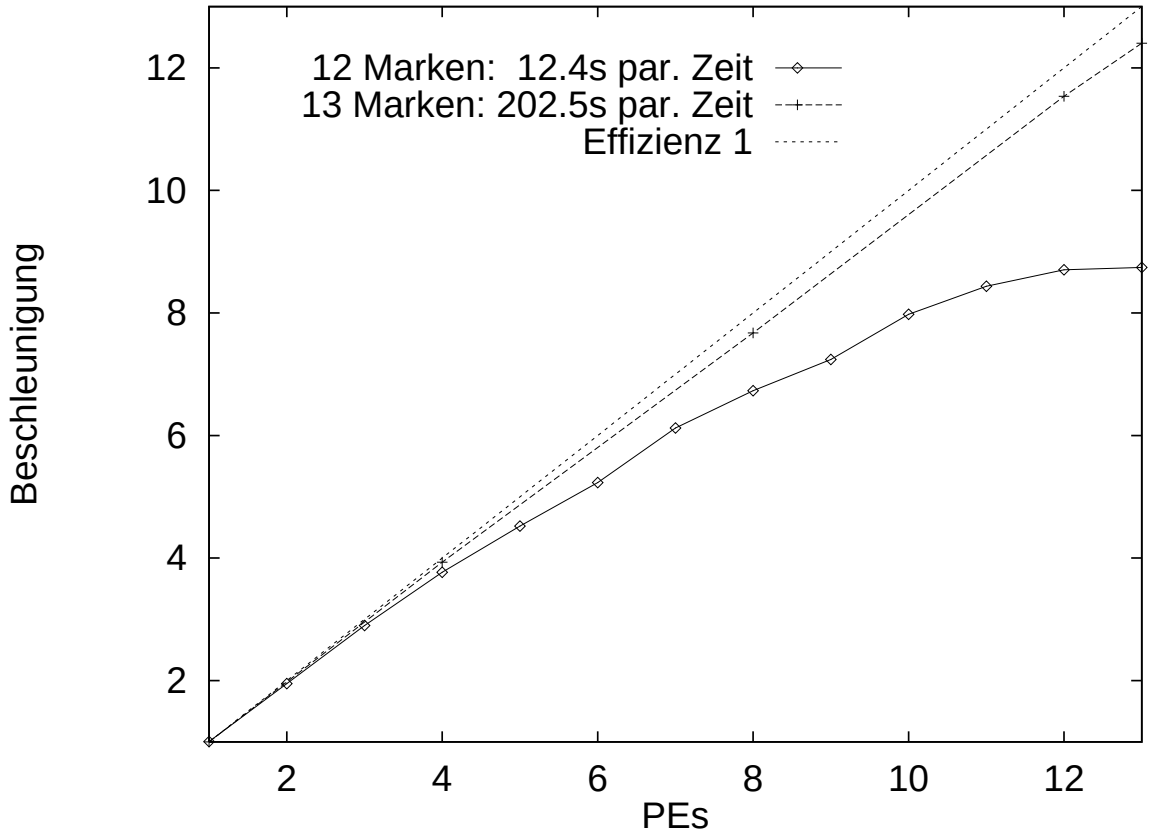
Applications: Radar astronomy, codes, ...

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

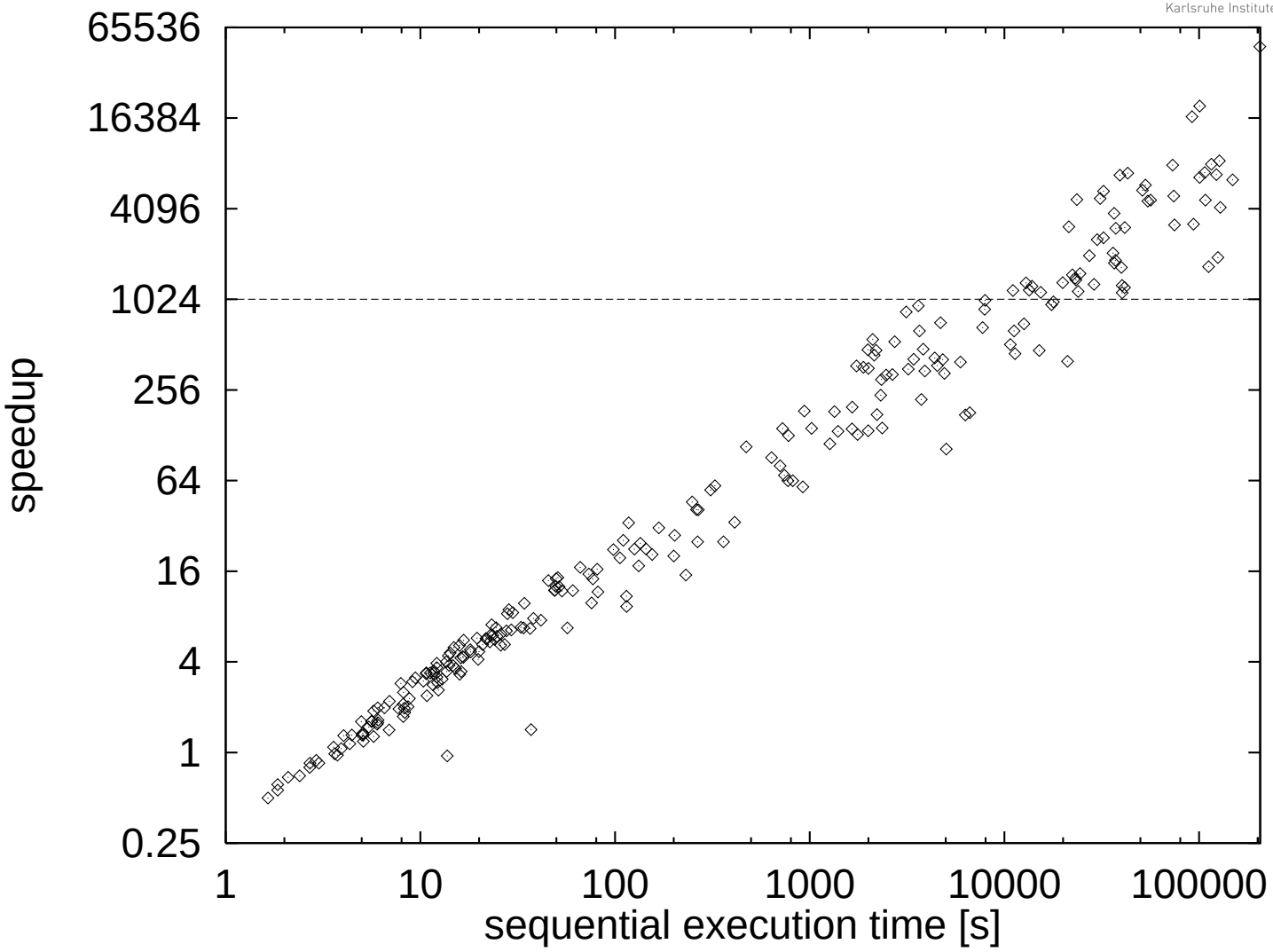
- m items
- Maximum knapsack weight M
- Item weights w_i
- Item profits p_i
- Find $x_i \in \{0,1\}$ such that
  - $\sum w_i x_i \leq M$
  - $\sum p_i x_i$ is maximized

Best known approach for large m:

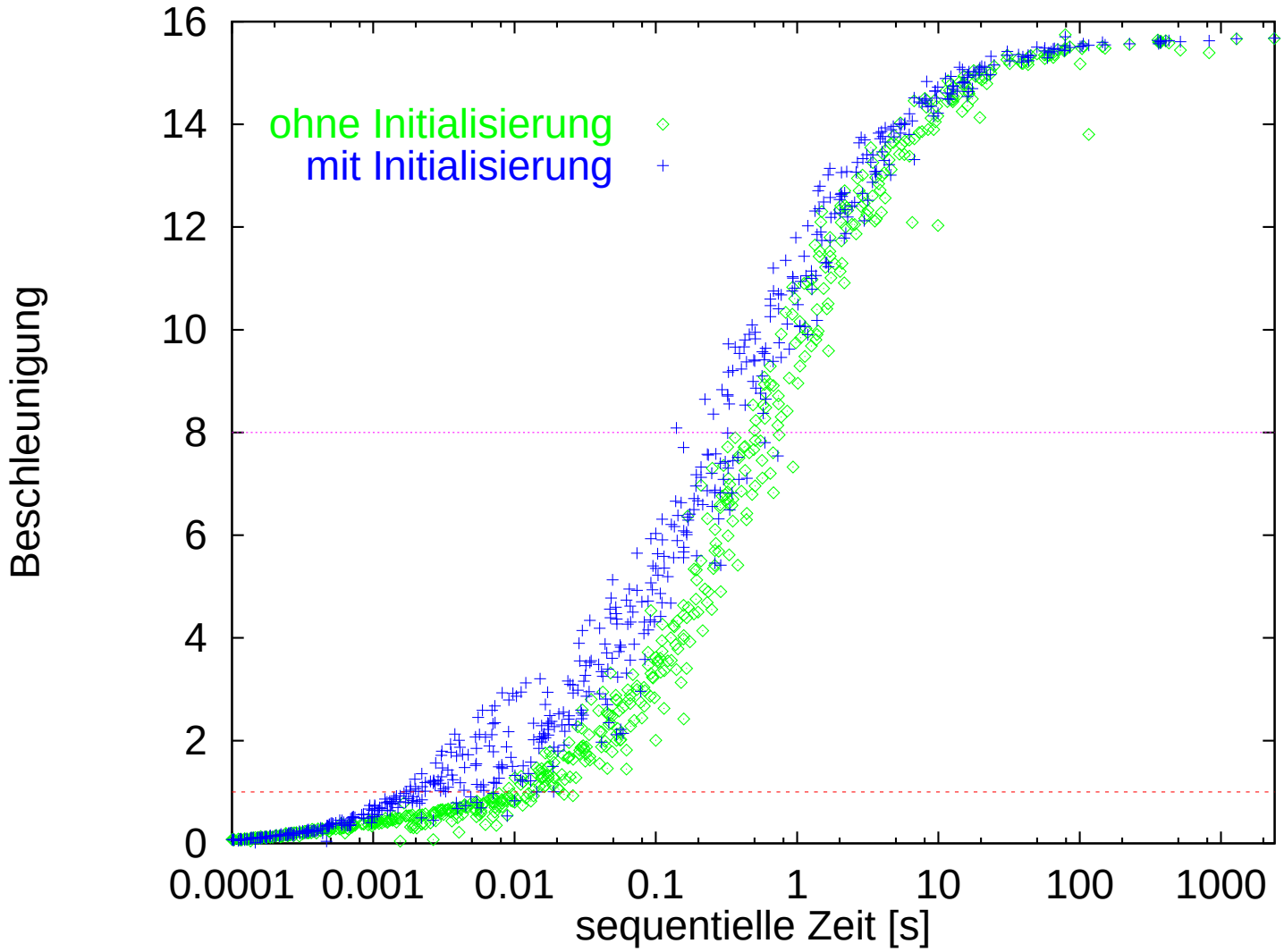
- Depth-first branch-and-bound
- Bounding function based on a the relaxation $x_i \in [0,1]$. (Can be computed in $\mathcal{O}(\log m)$ steps.)

Superlinear Speedup

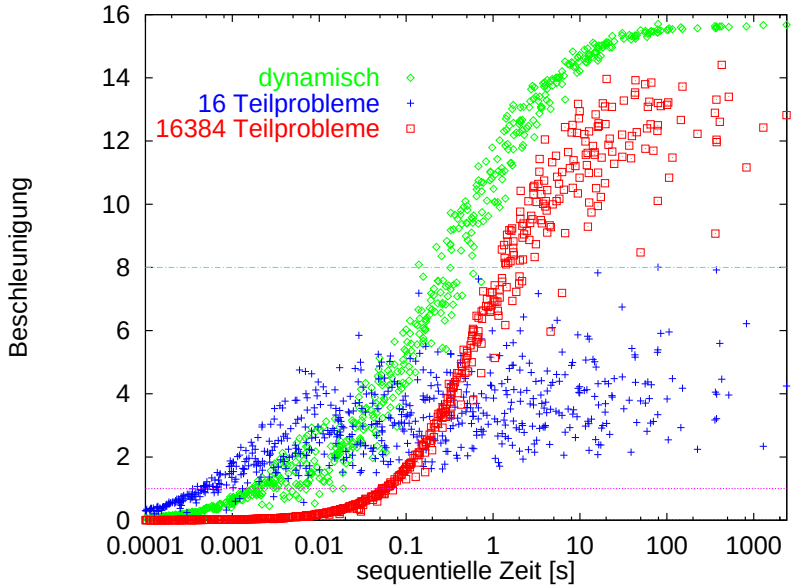
- Parsytec GCel-3/1024 under COSY (PB)
- 1024 processors
- 2000 items
- Splitting on all levels
- 256 random instances at the border between simple and difficult
- Overall $1410\times$ faster than seq. computation!

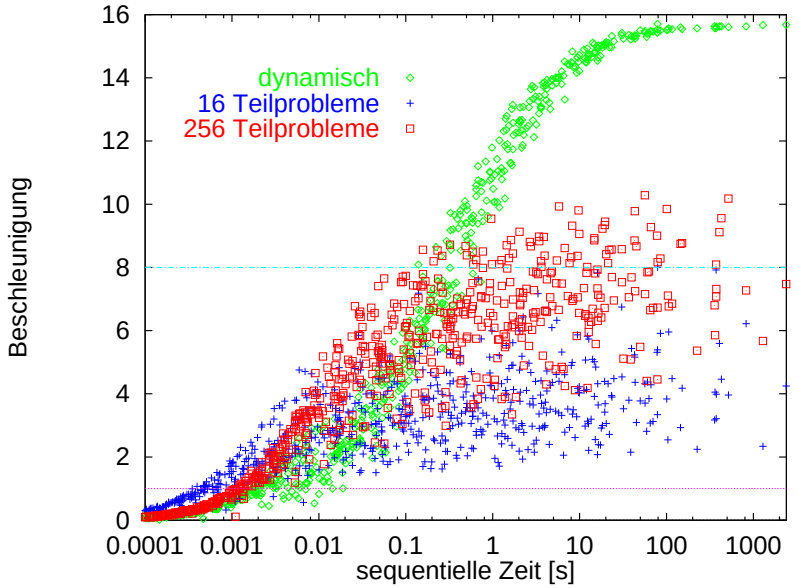

65536
16384
4096
1024
256
64
16
4
1
0.25
speedup
1
10
100
1000
10000
100000
sequential execution time [s]

Fast Initialization



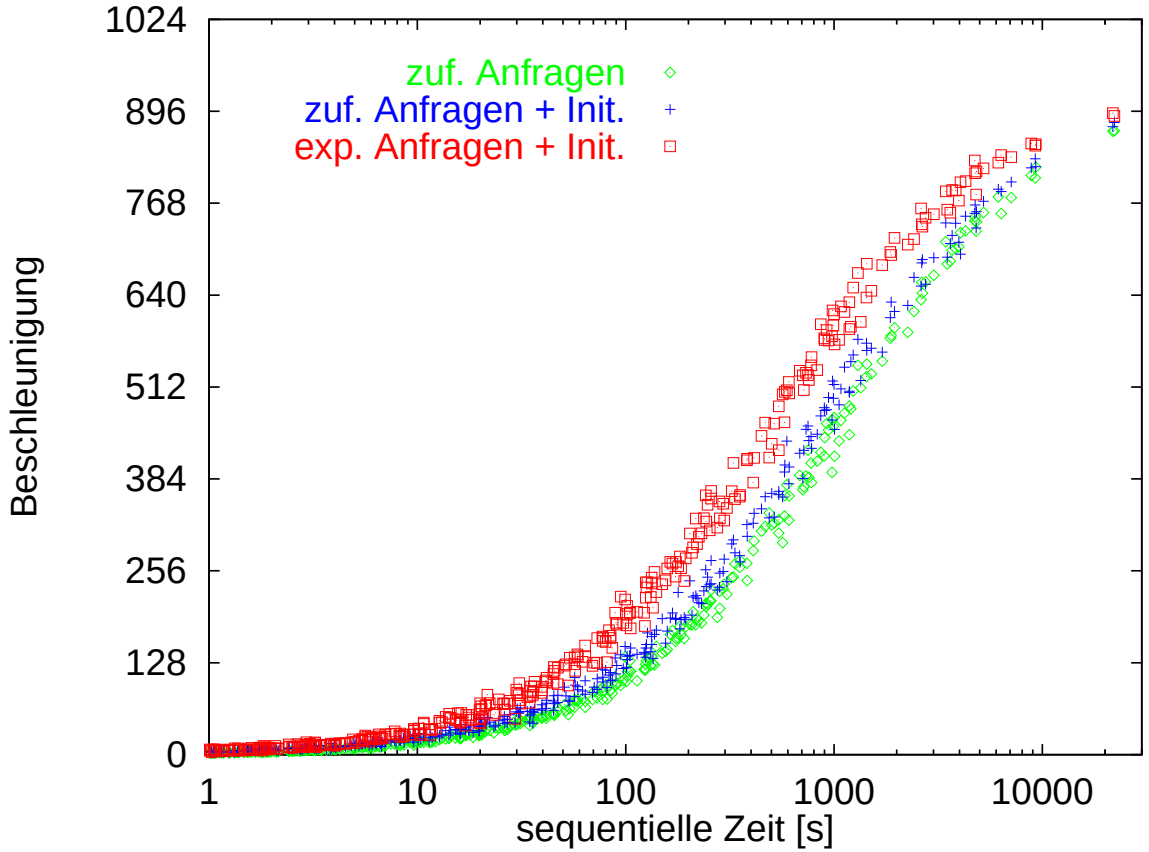
Static vs Dynamic LB





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] see Wikipedia

Framework for processing multisets of $(\text{key}, \text{value})$ Pairs.

// $M \subseteq K \times V$

// 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