

Algorithm Engineering für grundlegende Datenstrukturen und Algorithmen

Peter Sanders

Was sind die **schnellsten implementierten Algorithmen**
für das 1×1 der Algorithmik:

Listen, Sortieren, Prioritätslisten, Sortierte Listen, Hashtabellen,
Graphenalgorithmen?

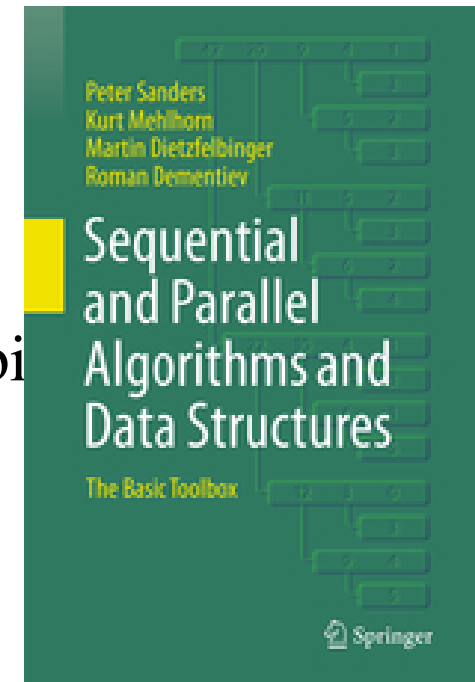
Nützliche Vorkenntnisse

- Algorithmen I
- Algorithmen II
- etwa Rechnerarchitektur (oder Ct lesen ;-)
- passive Kenntnisse von C/C++

Vertiefungsgebiet: Algorithmik

Material

- Folien
- wissenschaftliche Aufsätze.
Siehe Vorlesungshomepage
- Basiskenntnisse: Algorithmenlehrbücher,
z.B. Sanders et al., Cormen et al.
- Mehlhorn Näher: The LEDA Platform of Combinatorics
and Geometric Computing.
- Catherine McGeoch, A Guide to Experimental
Algorithmics
- ggf. Materialien aus neuem Buch “Algorithm Engineering”
Sanders et al.



Übungsbetrieb

- Insgesamt 20% der Note
- Leitung durch Moritz Laupichler und Matthias Schimek
- Möglicherweise Quiz (5% der Note) zur Semestermitte
- Hauptbeitrag: Wahl zwischen
 - Programmieraufgabe im Umfeld von minimalen Spannbäumen
 - Miniseminarvortrag gegen Ende des Semesters
 - Verbesserung/Erstellung einer Wikipediaseite zur Algorithmik

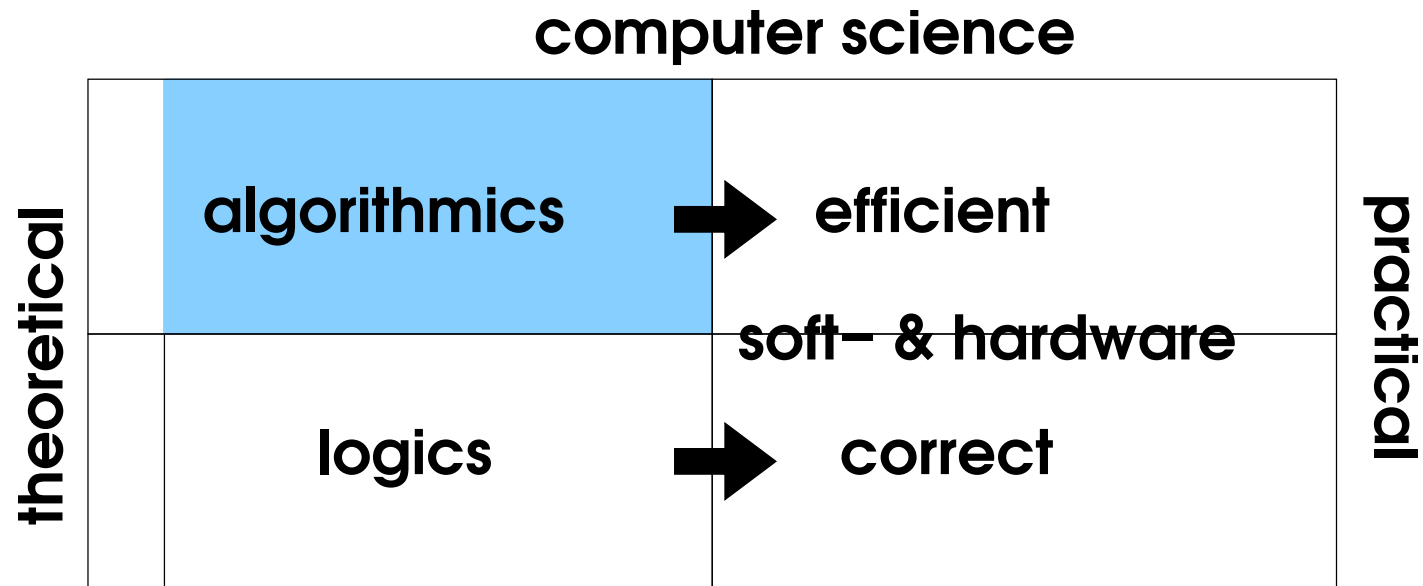
Überblick

- Was ist Algorithm Engineering, Modelle, ...
- Erste Schritte: Arrays, verkettete Listen, Stacks, FIFOs,...
- Sortieren rauf und runter
- Prioritätslisten
- Sortierte Listen
- Hashtabellen
- Minimale Spannbäume
- Kürzeste Wege
- Ausgewählte fortgeschrittene Algorithmen, z.B. maximale Flüsse

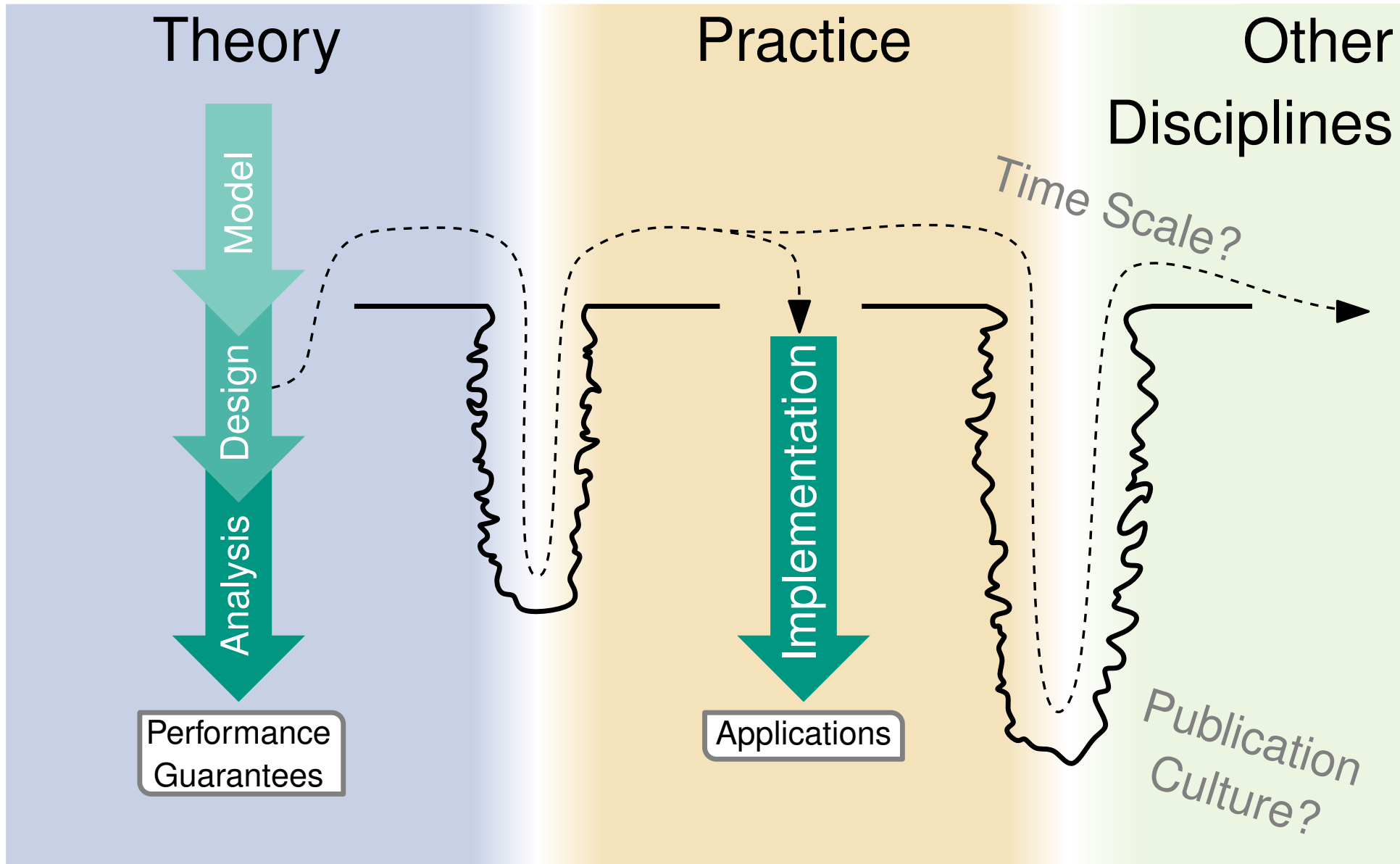
Methodik: in Exkursen

Algorithmics

= the **systematic** design of efficient software and hardware

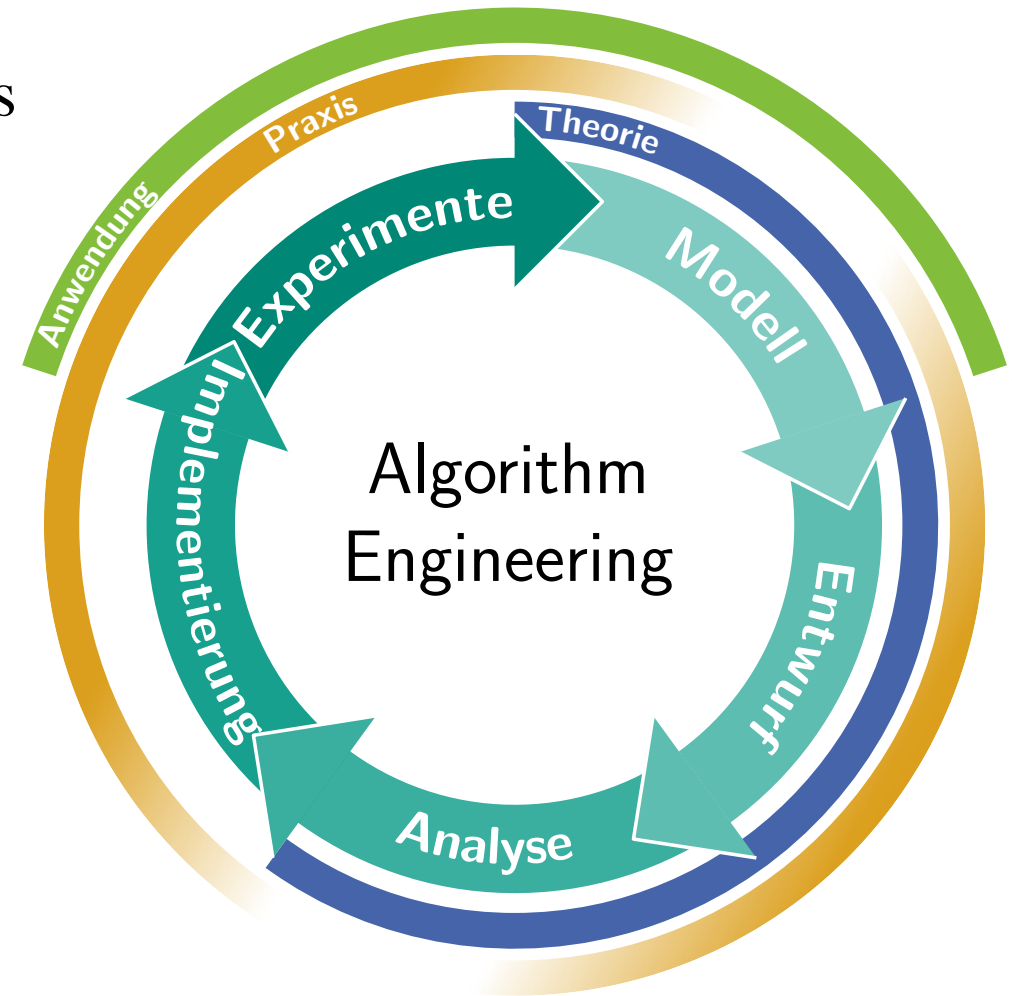


(Karikierte) traditionelle Sicht: Algorithmentheorie

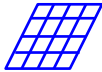

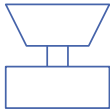

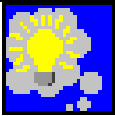
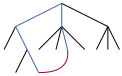




Algorithmik als Algorithm Engineering

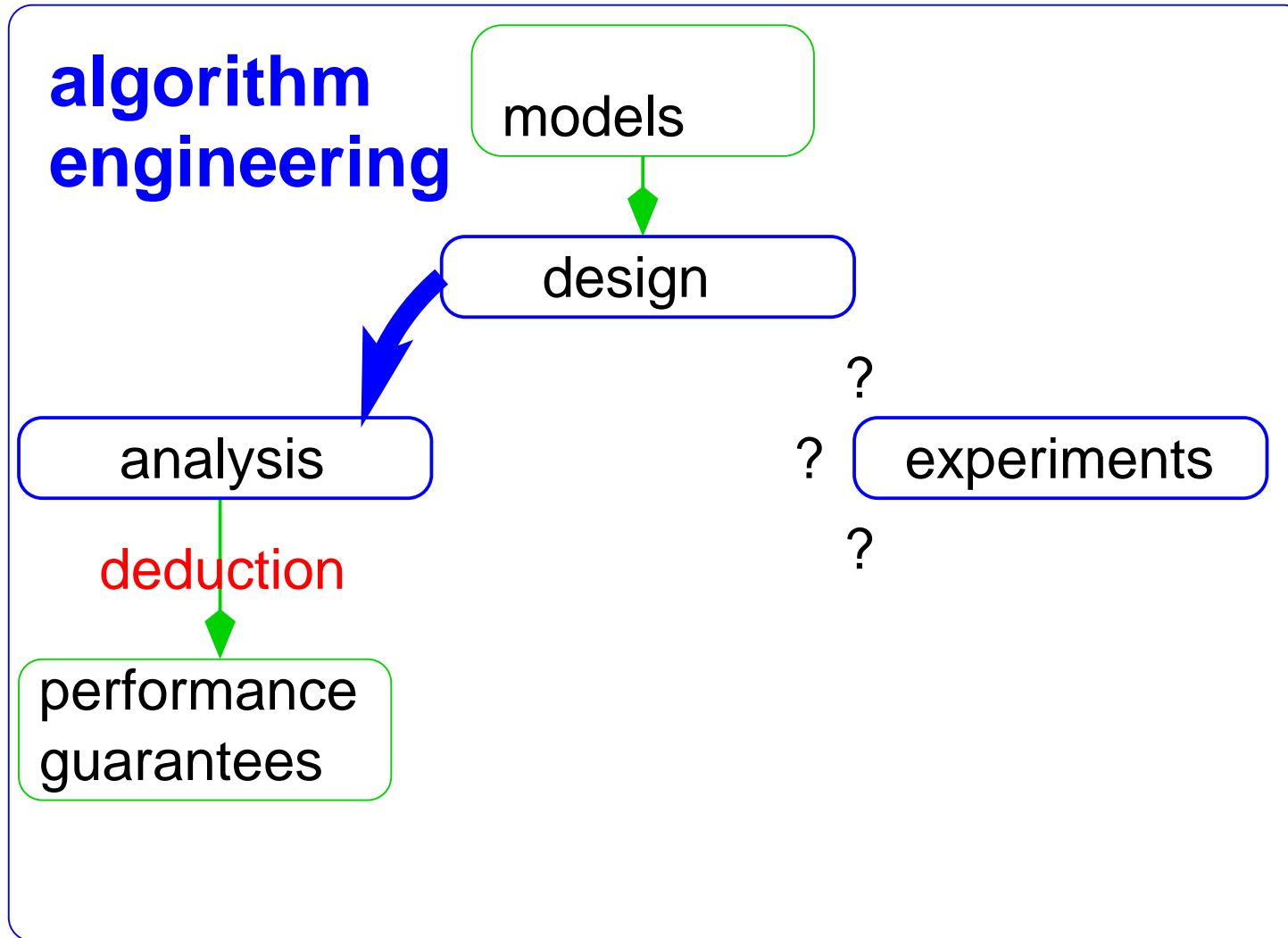
- überbrücke Lücken
zwischen Theorie und Praxis
- integrierte
interdisziplinäre
Forschung



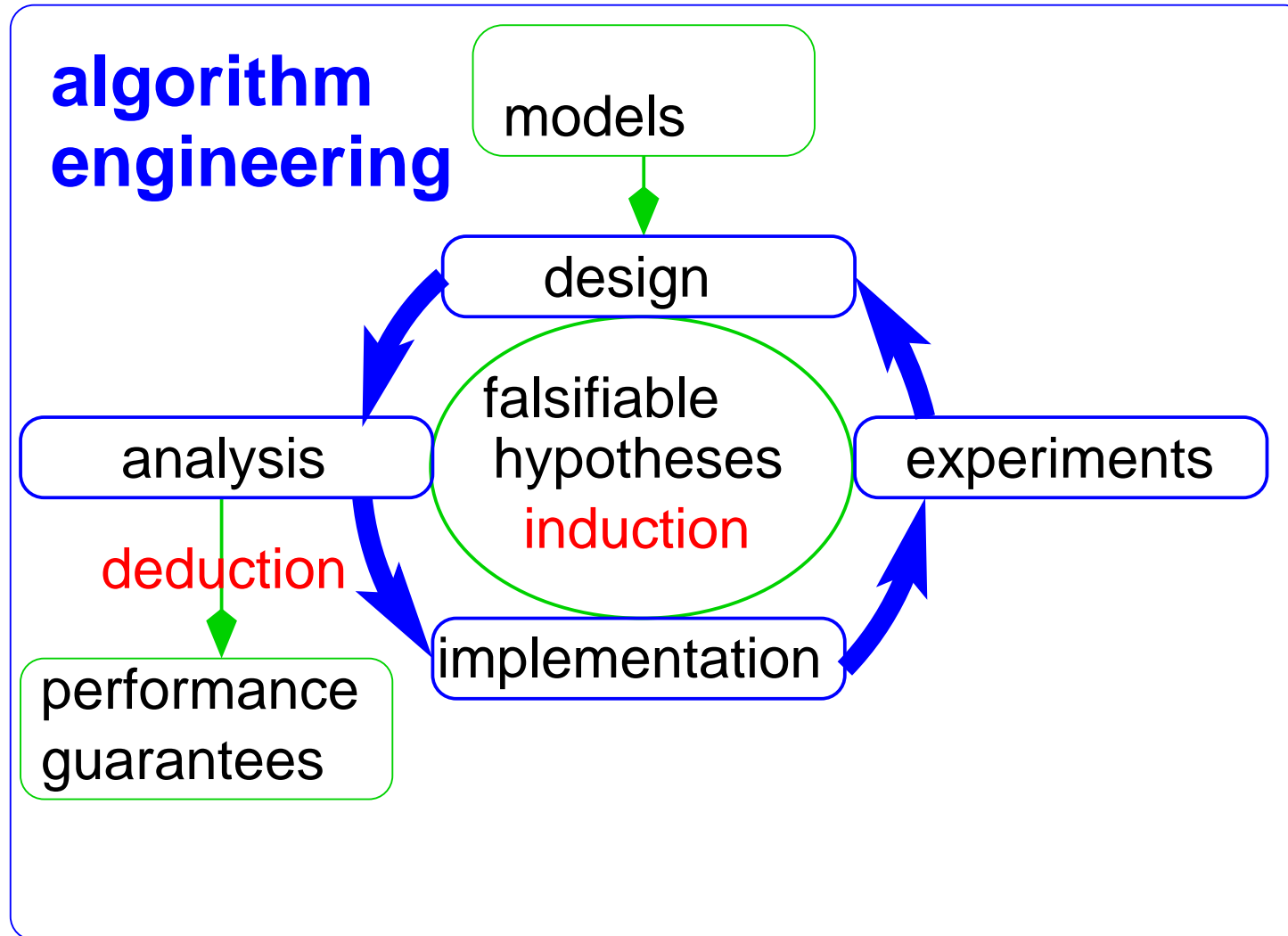
Gaps Between Theory & Practice

Theory		\longleftrightarrow	Practice	
simple		appl. model		complex
simple		machine model		real
complex		algorithms	<code>FOR</code>	simple
advanced		data structures		arrays,...
worst case	<code>max</code>	complexity measure		inputs
asympt.	<code>$\mathcal{O}(\cdot)$</code>	efficiency	<code>42%</code>	constant factors

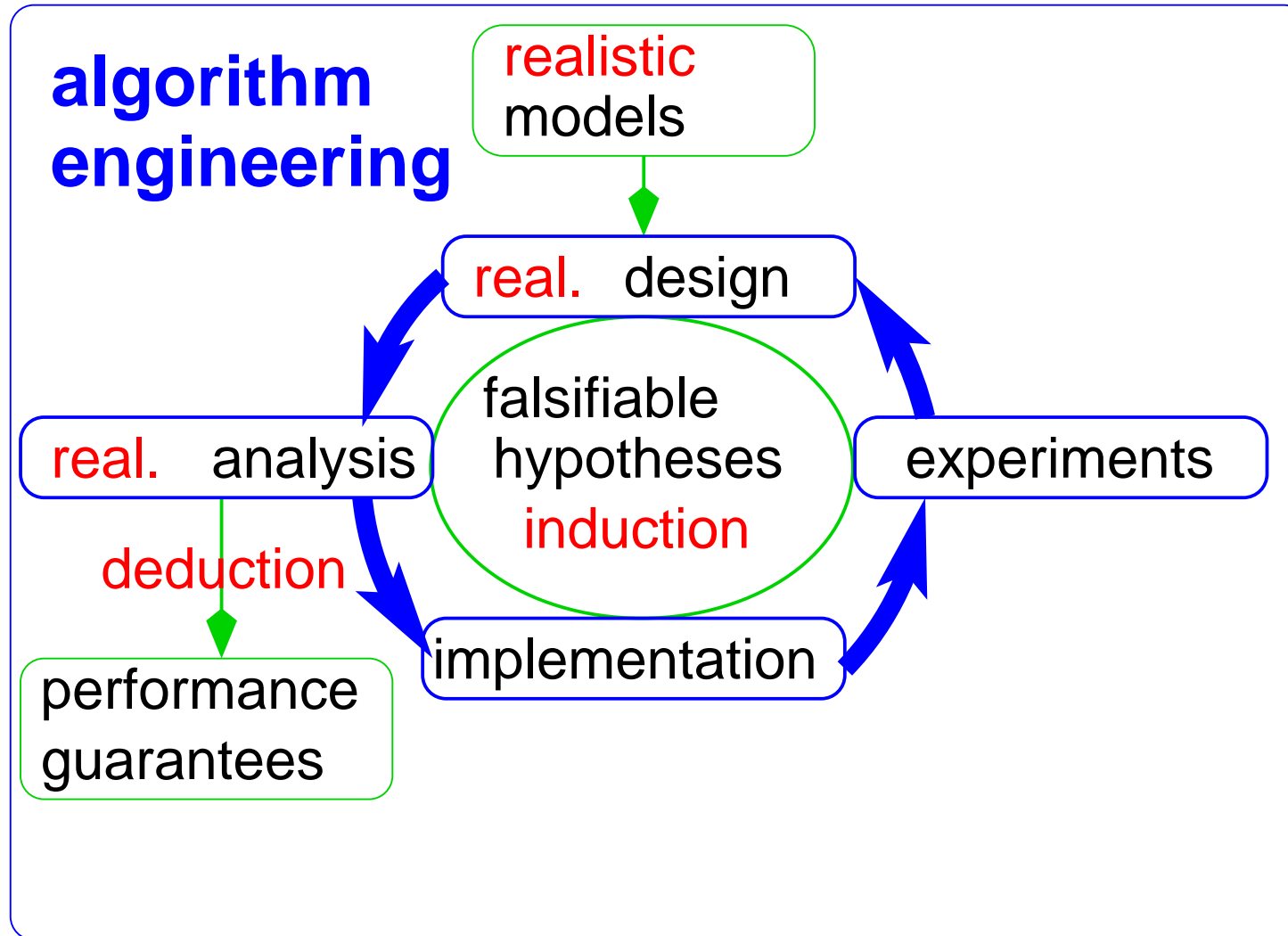
Algorithmics as Algorithm Engineering



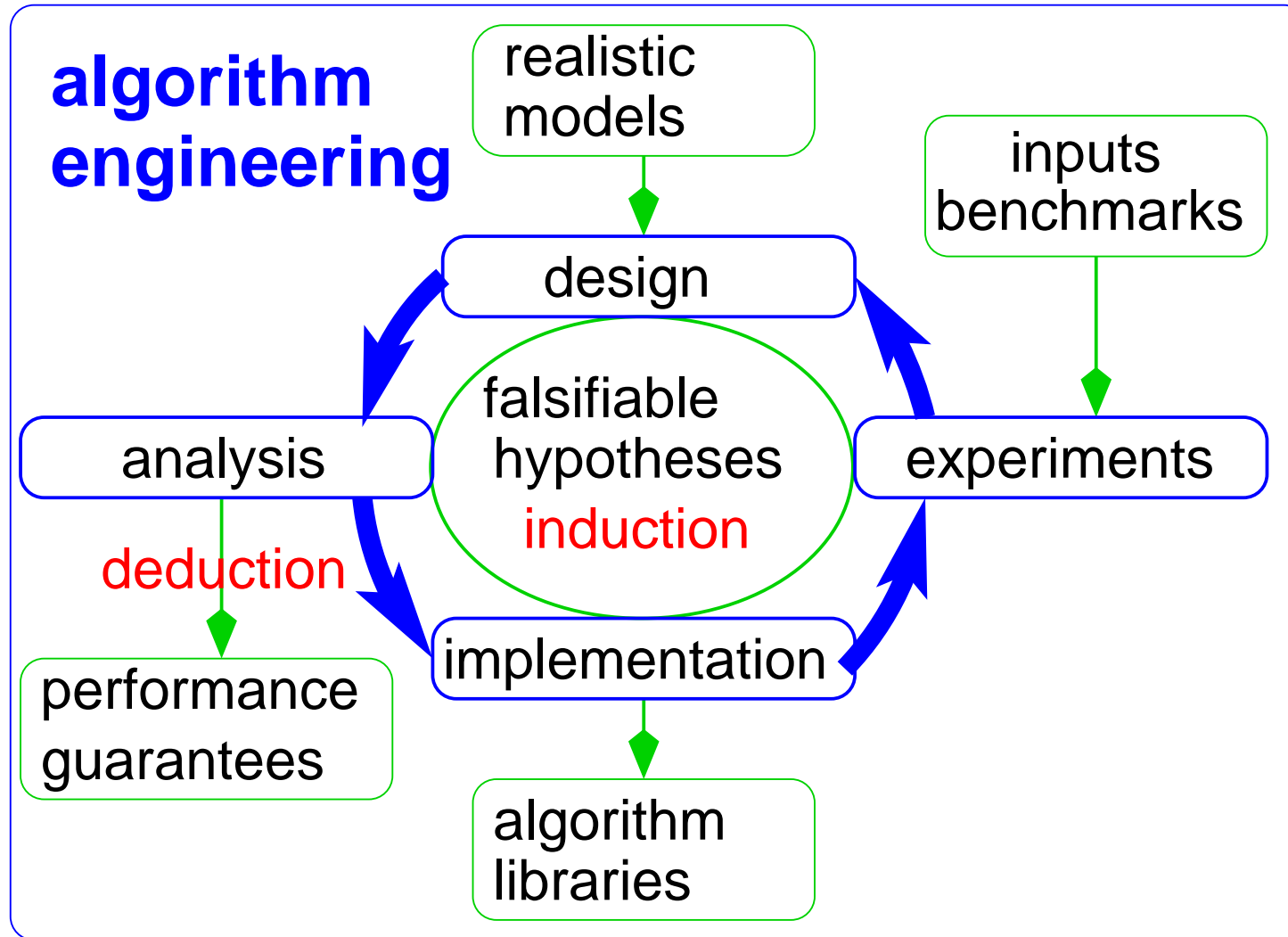
Algorithmics as Algorithm Engineering



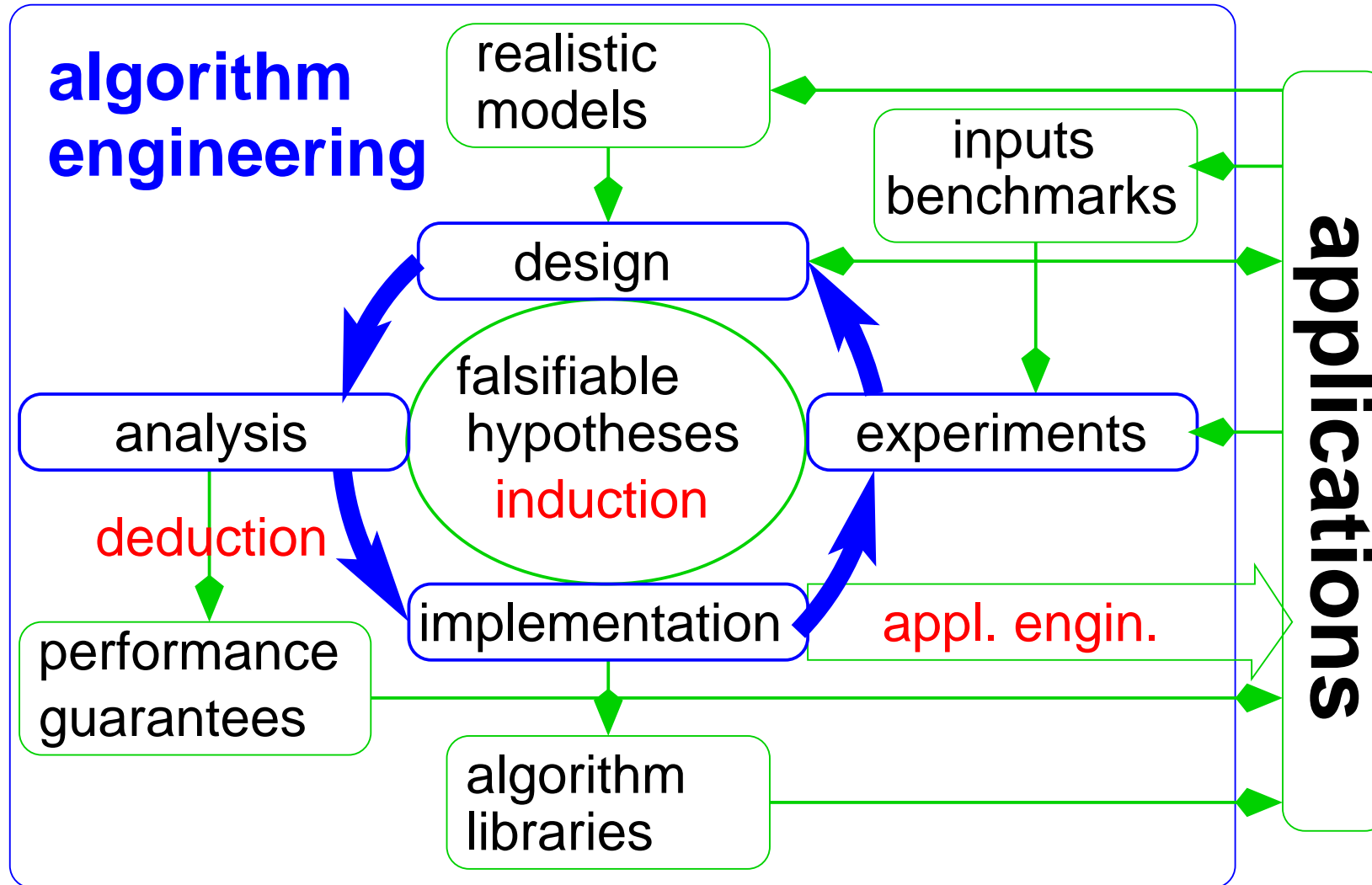
Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering

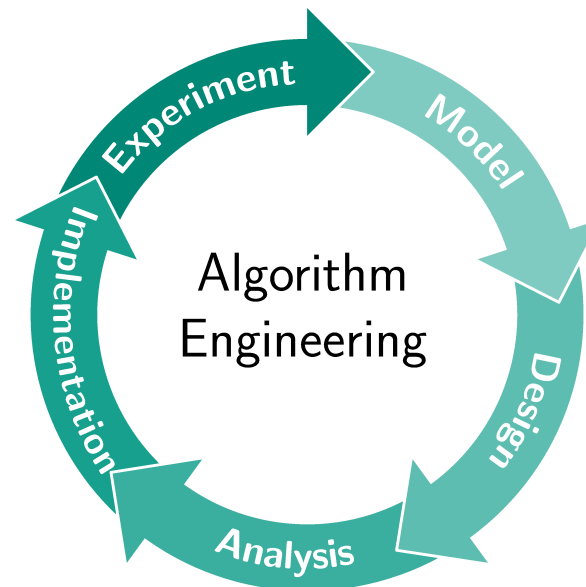


Algorithmics as Algorithm Engineering



Goals

- **bridge gaps** between theory and practice
- accelerate **transfer** of algorithmic results into **applications**
- keep the advantages of theoretical treatment:
generality of solutions and
reliability, predictability from performance guarantees



Bits of History

1843– Algorithms in theory and practice

1950s,1960s Still infancy

1970s,1980s Paper and pencil algorithm theory.

Exceptions exist, e.g., [J. Bentley, D. Johnson]

1986 Term used by [T. Beth],

lecture “**Algorithmentechnik**” in Karlsruhe.

1988– Library of Efficient Data Types and Algorithms (LEDA) [K. Mehlhorn]

1990– **DIMACS Implementation Challenges** [D. Johnson]

1997– **Workshop on Algorithm Engineering**

↪ ESA applied track [G. Italiano]

1997 Term used in US policy paper [Aho, Johnson, Karp, et. al]

1998 **Alex** workshop in Italy ↪ **ALENEX**



Warum diese Vorlesung?

- Jeder Informatiker kennt einige Lehrbuchalgorithmen
 \rightsquigarrow wir können gleich mit Algorithm Engineering loslegen
- Viele Anwendungen profitieren
- Es ist frappierend, dass es hier noch Neuland gibt
- Basis für Bachelor- Masterarbeiten

Was diese Vorlesung nicht ist:

Keine wiedergekäute Algorithmen I/II o.Ä.

- Grundvorlesungen “vereinfachen” die Wahrheit oft
- z.T. fortgeschrittene Algorithmen
- steilere Lernkurve
- Implementierungsdetails
- Betonung von Messergebnissen

Was diese Vorlesung nicht ist:

Keine Theorievorlesung

- keine (wenig?) Beweise
- Reale Leistung vor Asymptotik

Was diese Vorlesung nicht ist:

Keine Implementierungsvorlesung

- Etwas Algorithmenanalyse,...
- Wenig Software Engineering

Exkurs: Maschinenmodelle

RAM/von Neumann Modell

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

- Einfach
- Sehr erfolgreich
- zunehmend **unrealistisch**
weil reale Hardware
immer komplexer wird

$O(1)$ registers



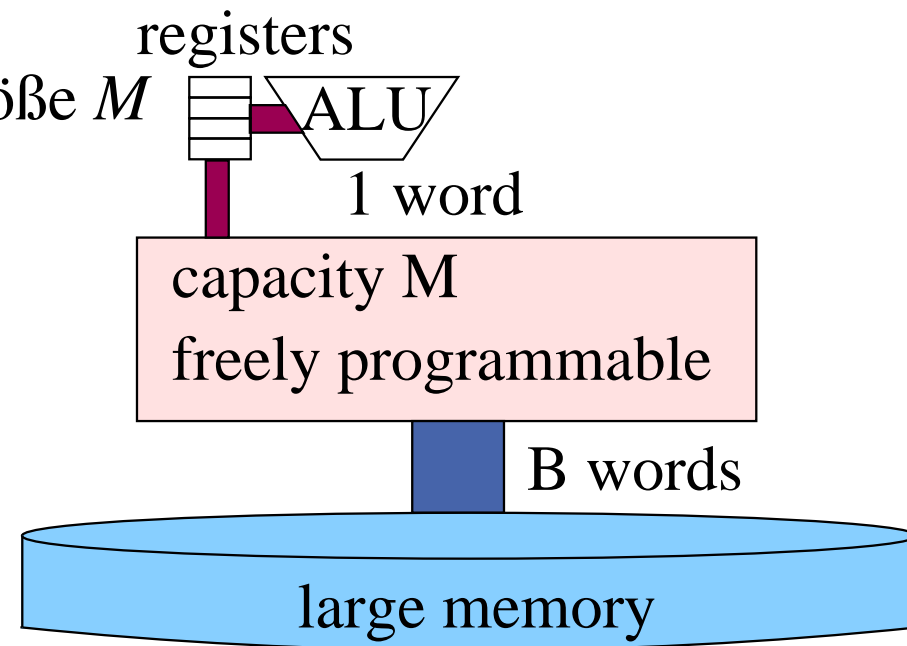
1 word = $O(\log n)$ bits

freely programmable
large memory

Das Sekundärspeichermodell

M : Schneller Speicher der Größe M

B : Blockgröße



Analyse: zähle (nur?) Blockzugriffe (I/Os)

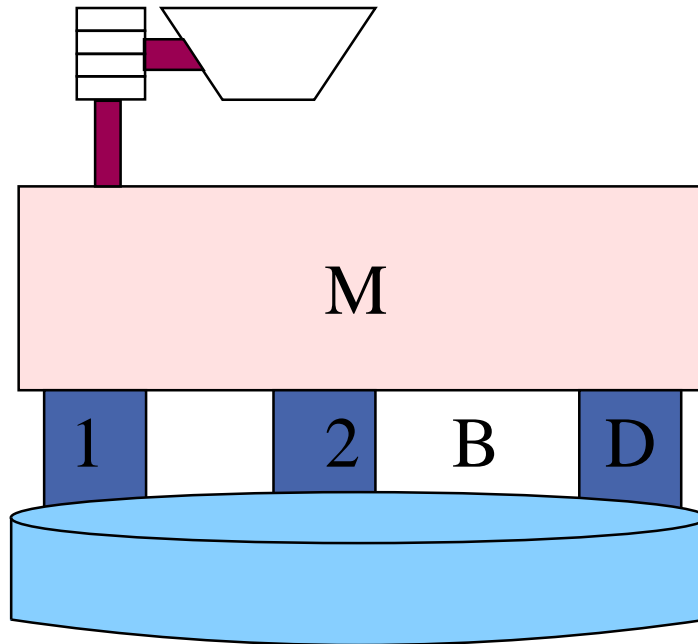
Interpretationen des Sekundärspeichermodells

	Externspeicher	Caches
großer Speicher	Platte(n)	Hauptspeicher
M	Hauptspeicher	ein cache level
B	Plattenblock (MBytes!)	Cache Block (16–256) Byte

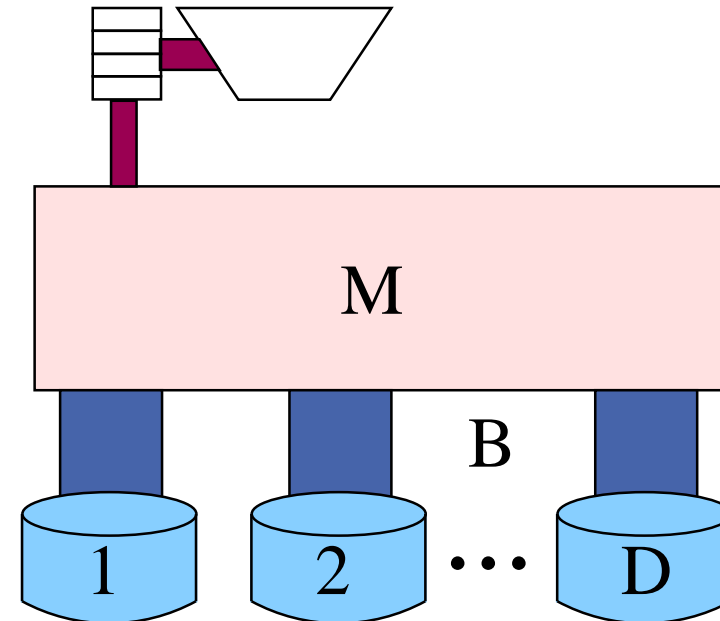
Ggf. auch zwei cache levels.

Variante: SSDs

Parallele Platten



Mehrkopfmodell
[Aggarwal Vitter 88]



unabhängige Platten
[Vitter Shriver 94]

Mehr Modellaspekte

- Instruktionsparallelismus (Superscalar, VLIW, EPIC, SIMD, ...)
- Pipelining
- Was kostet branch misprediction?
- Multilevel Caches (gegenwärtig 2–3 levels) \rightsquigarrow “cache oblivious algorithms”
- Parallele Prozessoren, Multithreading
- Kommunikationsnetzwerke
- ...

1 Arrays, Verkettete Listen und abgeleitete Datenstrukturen

Bounded Arrays

Eingebaute Datenstruktur.

Größe muss von Anfang an bekannt sein

Unbounded Array

z.B. `std::vector`

`pushBack`: Element anhängen

`popBack`: Letztes Element löschen

Idee: verdoppeln wenn der Platz ausgeht
halbiere wenn Platz verschwendet wird

Wenn man das **richtig** macht, brauchen

n `pushBack`/`popBack` Operationen Zeit $\mathcal{O}(n)$

Algorithme: `pushBack`/`popBack` haben konstante **amortisierte**
Komplexität. Was kann man falsch machen?

Doppelt verkettete Listen



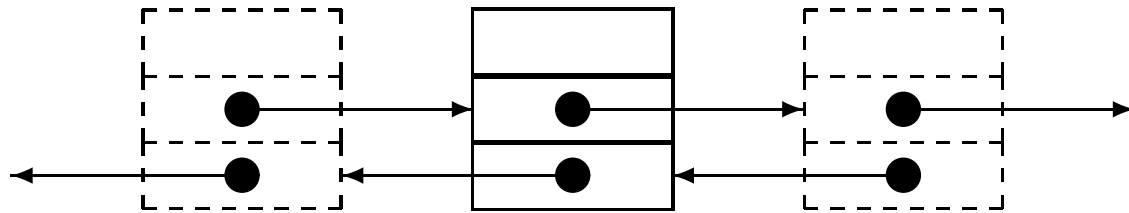
Class Item of Element // one link in a doubly linked list

e : Element

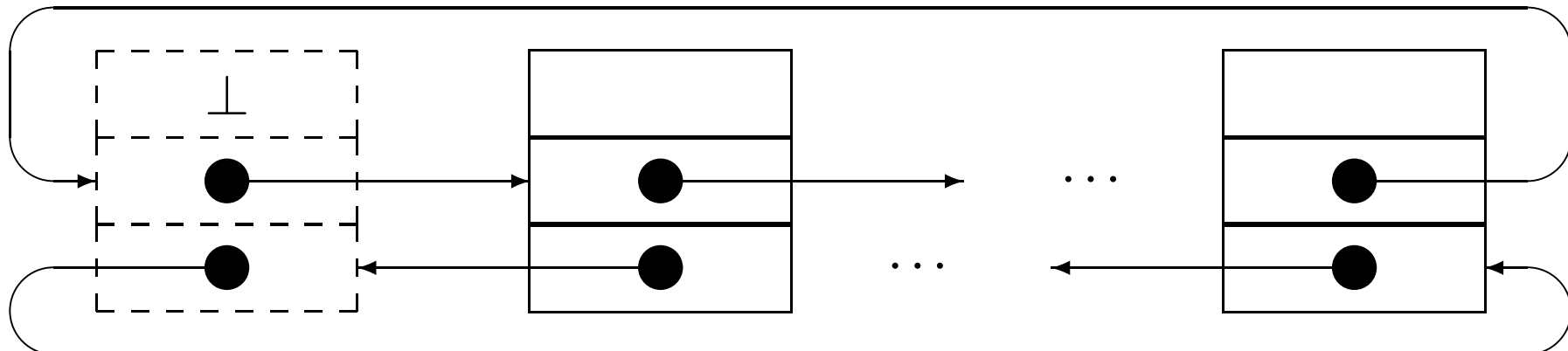
next : Handle //

prev : Handle

invariant next → prev = prev → next = **this**



Trick: Use a dummy header



Procedure splice(a,b,t : Handle)

assert b is not before $a \wedge t \notin \langle a, \dots, b \rangle$

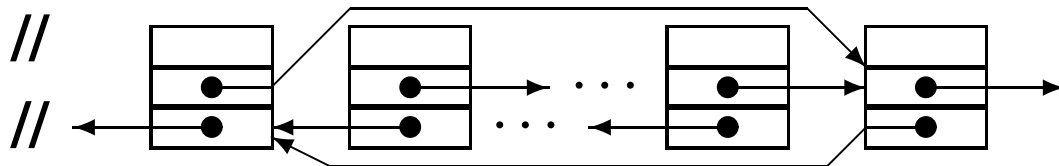
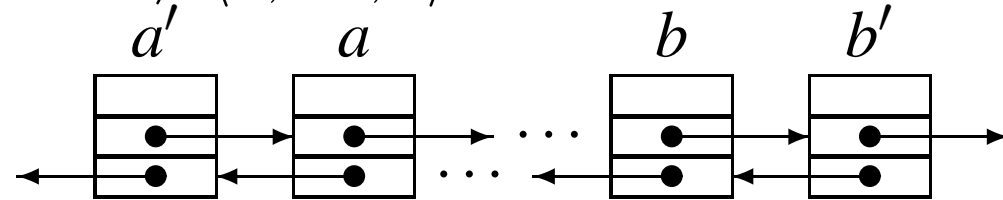
// Cut out $\langle a, \dots, b \rangle$

$a' := a \rightarrow \text{prev}$

$b' := b \rightarrow \text{next}$

$a' \rightarrow \text{next} := b'$

$b' \rightarrow \text{prev} := a'$



// insert $\langle a, \dots, b \rangle$ after t

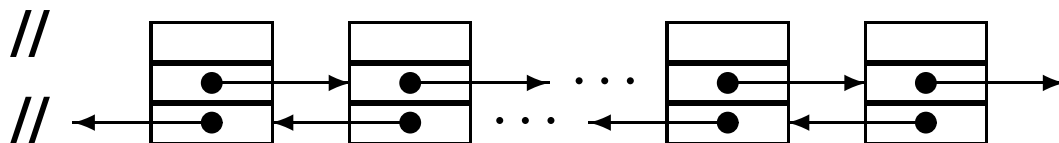
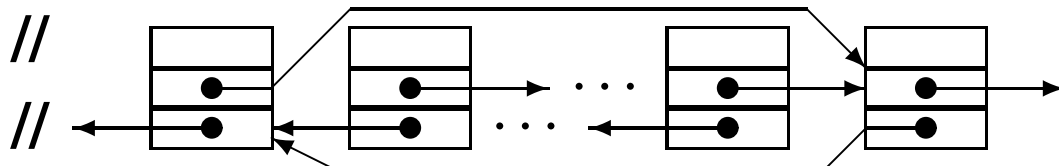
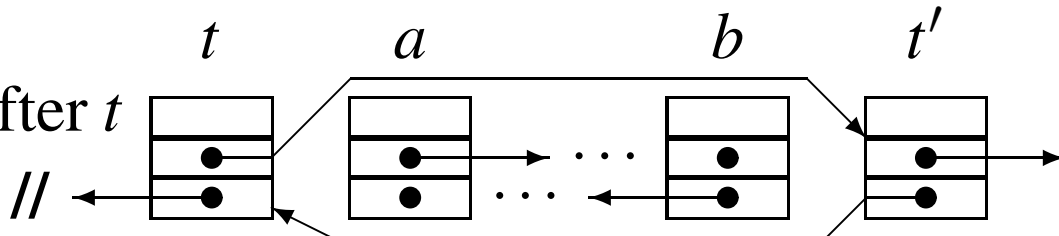
$t' := t \rightarrow \text{next}$

$b \rightarrow \text{next} := t'$

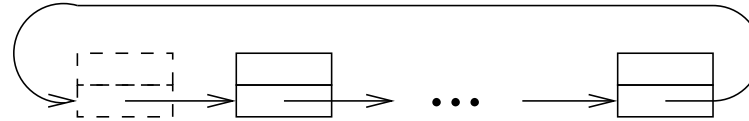
$a \rightarrow \text{prev} := t$

$t \rightarrow \text{next} := a$

$t' \rightarrow \text{prev} := b$



Einfach verkettete Listen



Vergleich mit doppelt verketteten Listen

- Weniger Speicherplatz
- Platz ist oft auch Zeit
- Eingeschränkter z.B. kein delete
- Merkwürdige Benutzerschnittstelle, z.B. deleteAfter

Speicherverwaltung für Listen

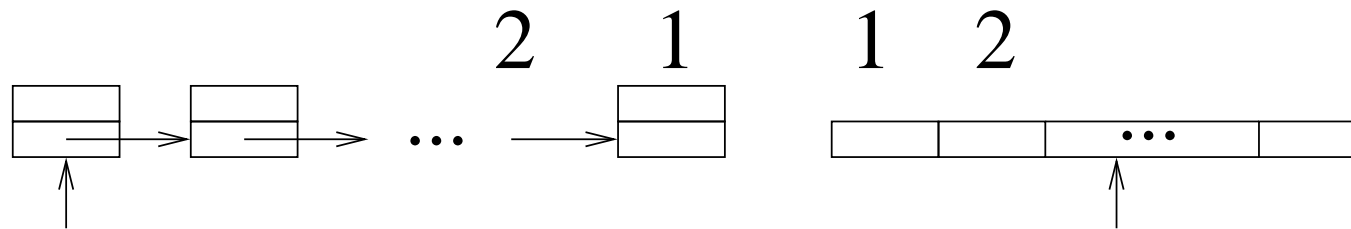
- kann leicht 90 % der Zeit kosten!
- Lieber Elemente zwischen (Free)lists herschieben als echte `mallocs`
- Alloziere viele Items gleichzeitig
- Am Ende alles freigeben?
- Speichere „parasitär“. z.B. Graphen:
 - Knotenarray. Jeder Knoten speichert ein ListItem
 - ~> Partition der Knoten kann als verkettete Listen gespeichert werden
 - ~> MST, shortest Path

Challenge: garbage collection, viele Datentypen

~> auch ein Software Engineering Problem

hier nicht

Beispiel: Stack

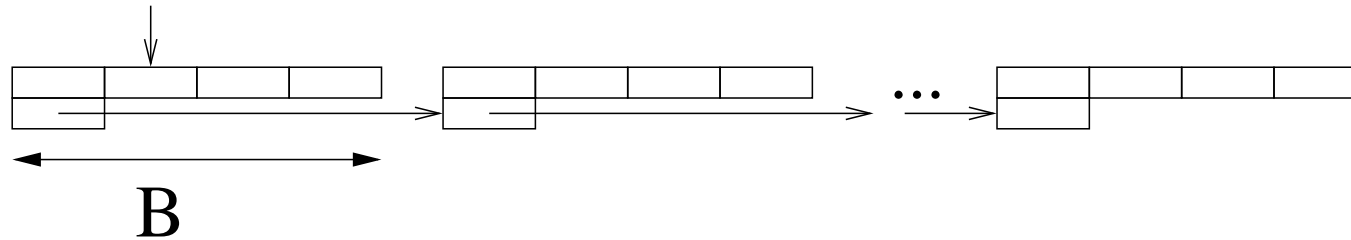


	SList	B-Array	U-Array
dynamisch	+	-	+
Platzverschwendung	pointer freigeben?	zu groß?	zu groß?
Zeitverschwendung	cache miss	+	umkopieren
worst case time	(+)	+	-

War es das?

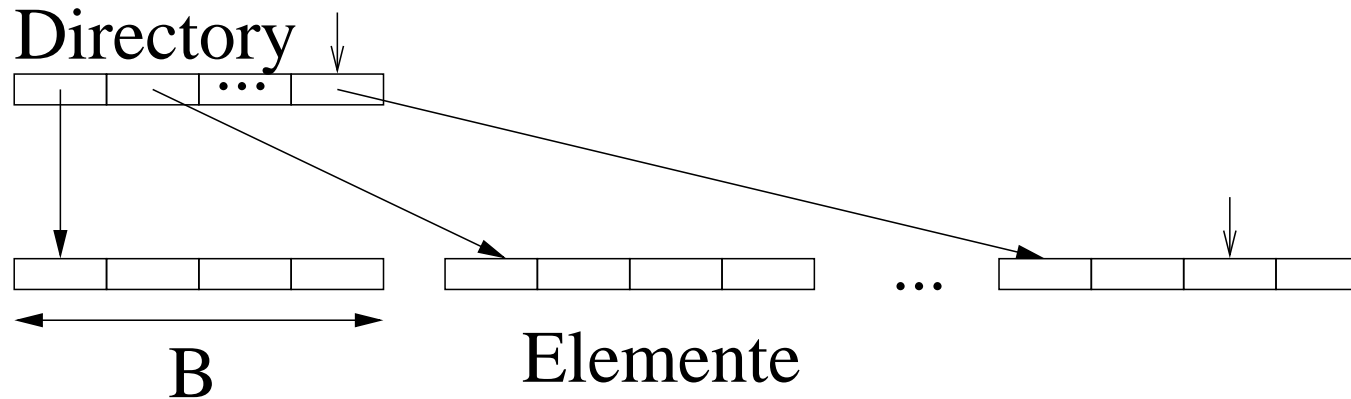
Hat jede Implementierung gravierende Schwächen?

The Best From Both Worlds



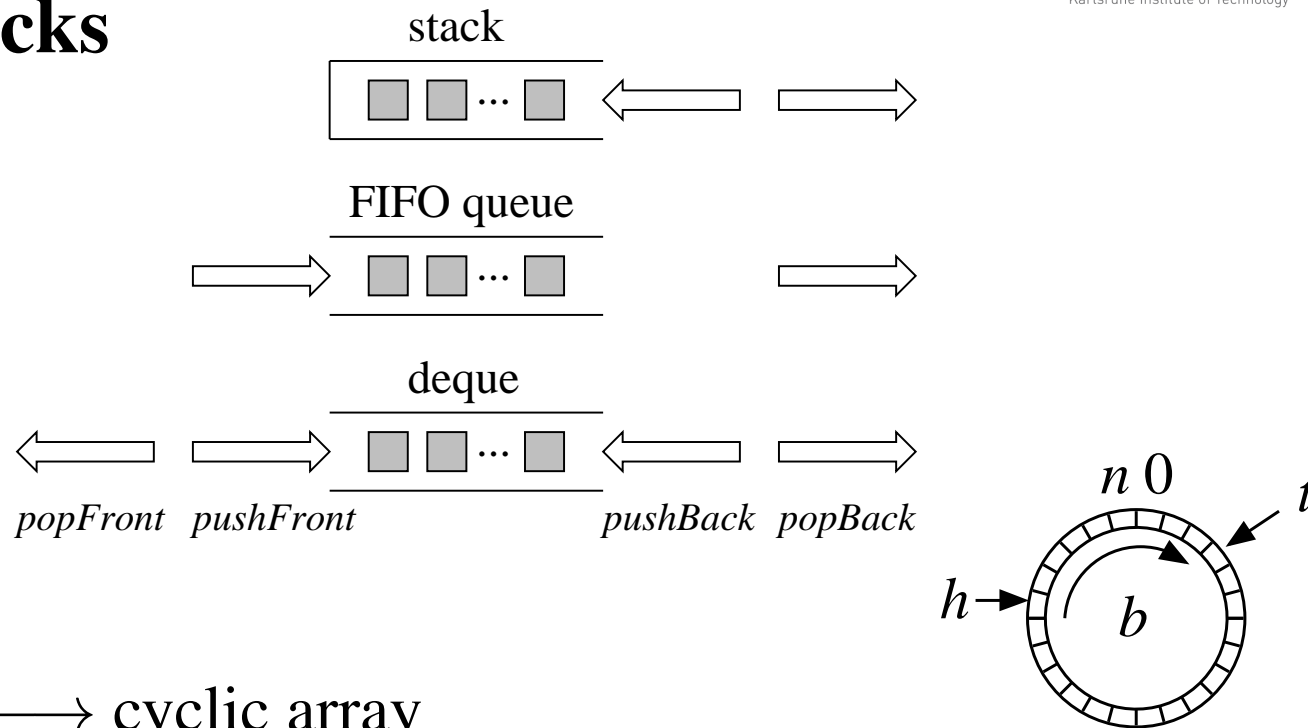
	hybrid
dynamisch	+
Platzverschwendung	$n/B + B$
Zeitverschwendung	+
worst case time	+

Eine Variante



- Reallozierung im top level \rightsquigarrow nicht worst case konstante Zeit
- + Indizierter Zugriff auf $S[i]$ in konstanter Zeit

Beyond Stacks



FIFO: BArray \longrightarrow cyclic array

Aufgabe: Ein Array, das “[i]” in konstanter Zeit und einfügen/löschen von Elementen in Zeit $\mathcal{O}(\sqrt{n})$ unterstützt

Aufgabe: Ein externer Stack, der n push/pop Operationen mit $\mathcal{O}(n/B)$ I/Os unterstützt

Aufgabe: Tabelle für hybride Datenstrukturen vervollständigen

Operation	List	SList	UArray	CArray	explanation of ‘*’
[·]	n	n	1	1	
·	1*	1*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1*	n	n	insertAfter only
remove	1	1*	n	n	removeAfter only
pushBack	1	1	1*	1*	amortized
pushFront	1	1	n	1*	amortized
popBack	1	n	1*	1*	amortized
popFront	1	1	n	1*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,.. .	n	n	n^*	n^*	cache efficient

Was fehlt?

Fakten Fakten Fakten

Messungen für

- Verschiedene Implementierungsvarianten
- Verschiedene Architekturen
- Verschiedene Eingabegrößen
- Auswirkungen auf reale Anwendungen
- Kurven dazu
- Interpretation, ggf. Theoriebildung

Aufgabe: Array durchlaufen versus zufällig allozierte verkettete
Liste

Algorithm Engineering

A Detailed View

Using Sorting as Guiding Example

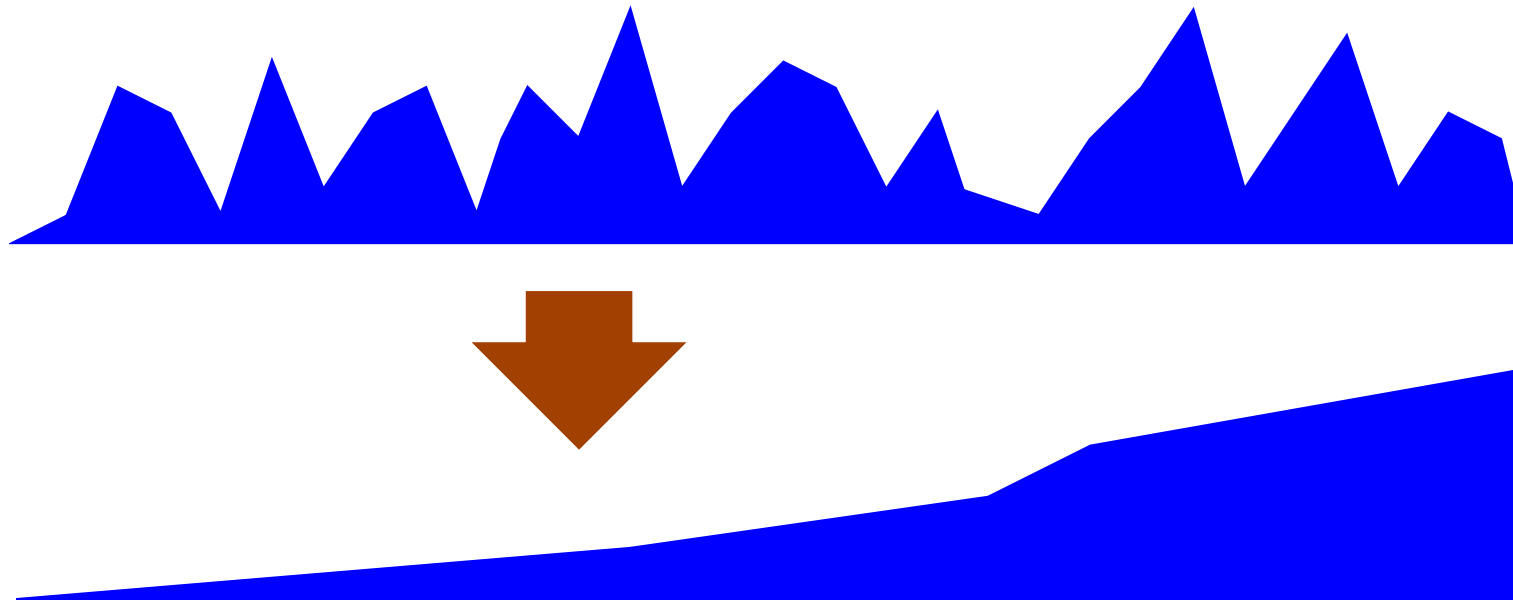
Sorting

Permute n elements of an array a such that

$$a[1] \leq a[2] \leq \dots \leq a[n]$$

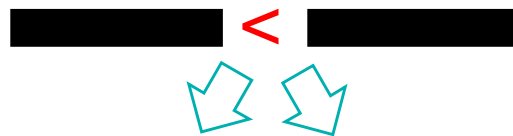
Efficient sequential, comparison based algorithms take time

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



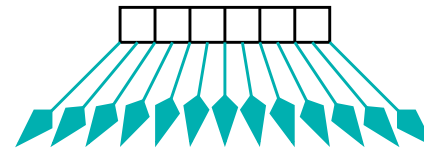
Sorting – Model

Comparison
based



true/false

arbitrary
e.g. integer



full information

Why Sorting?

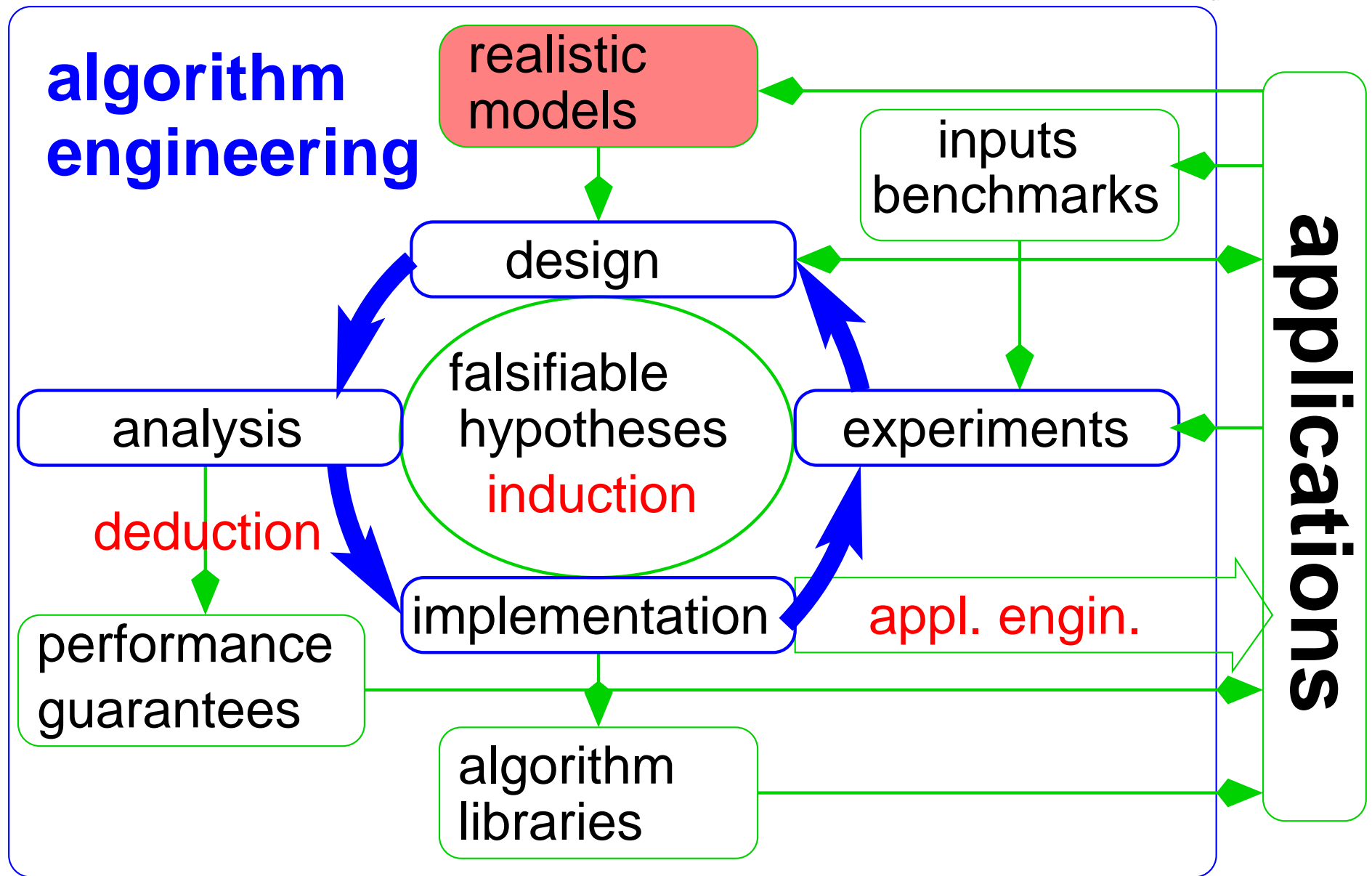
Teaching perspective:

- simple
- surprisingly nontrivial
- computer scientists know the basics

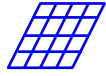

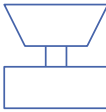

Application Perspective:

- Build index data structures
- Process objects in well defined order
- Group similar objects

↪ Bottleneck in many applications



Realistic Models

Theory	\longleftrightarrow	Practice
simple 	appl. model	 complex
simple 	machine model	 real

- Careful refinements
- Try to preserve (partial) analyzability / simple results



Advanced Machine Models

RAM / von Neumann

PRAM / shared memory

registers

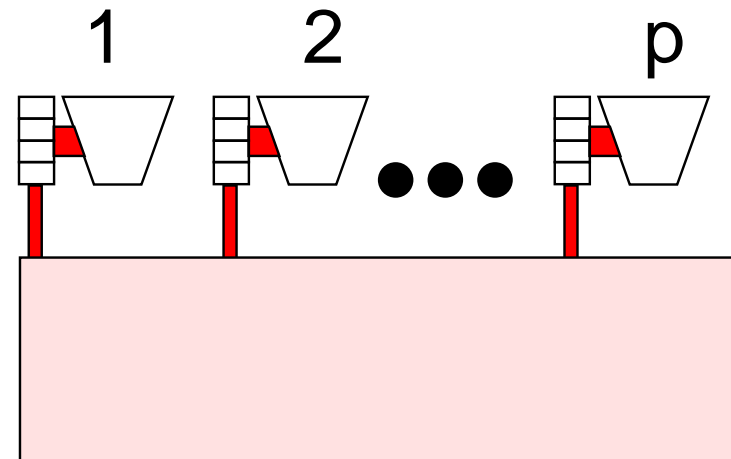
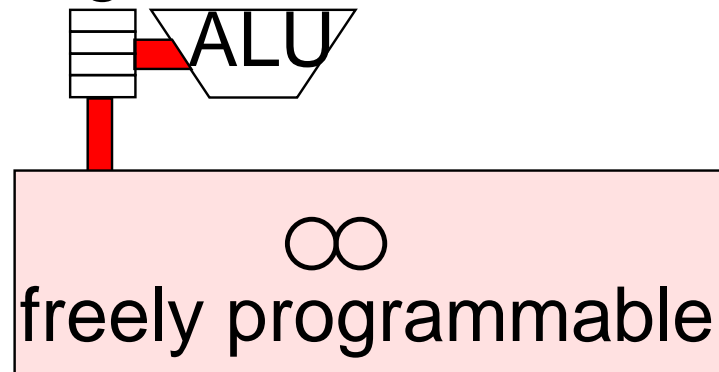
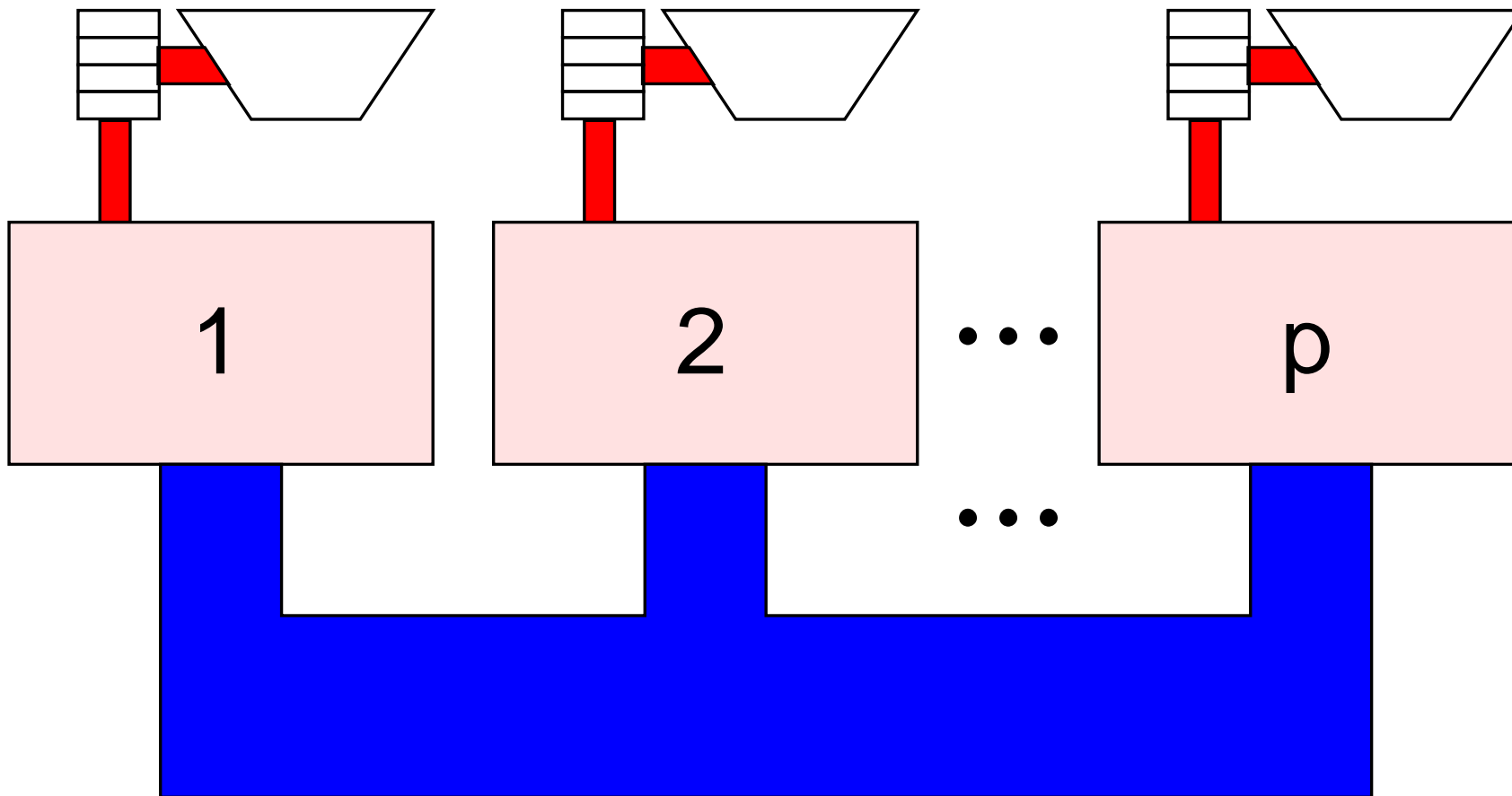


exhibit parallelism

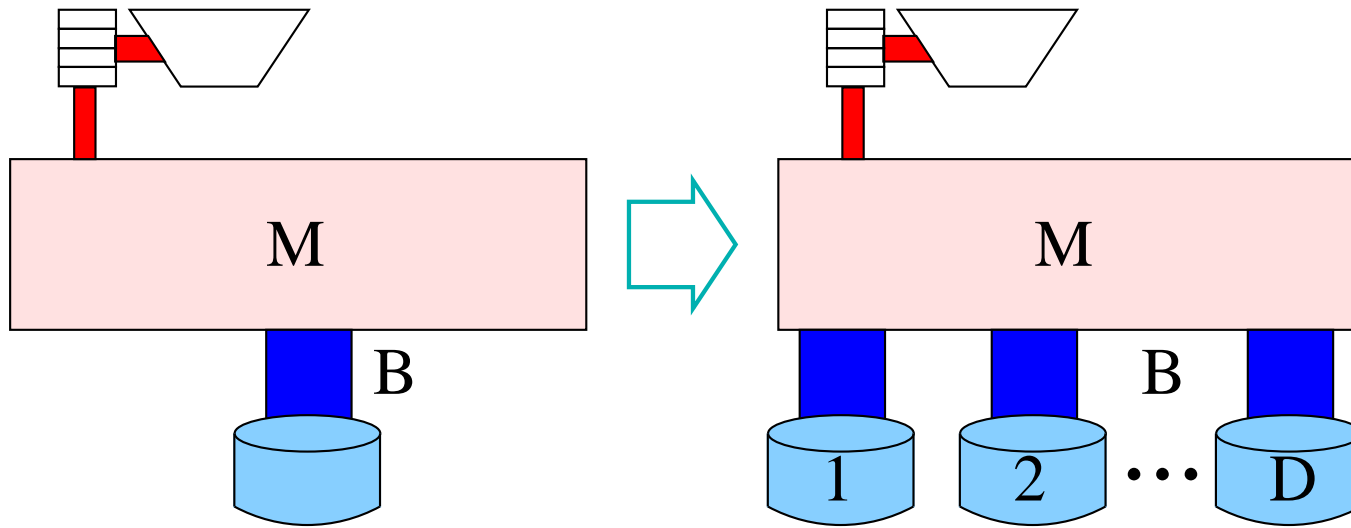
Distributed Memory

[1]



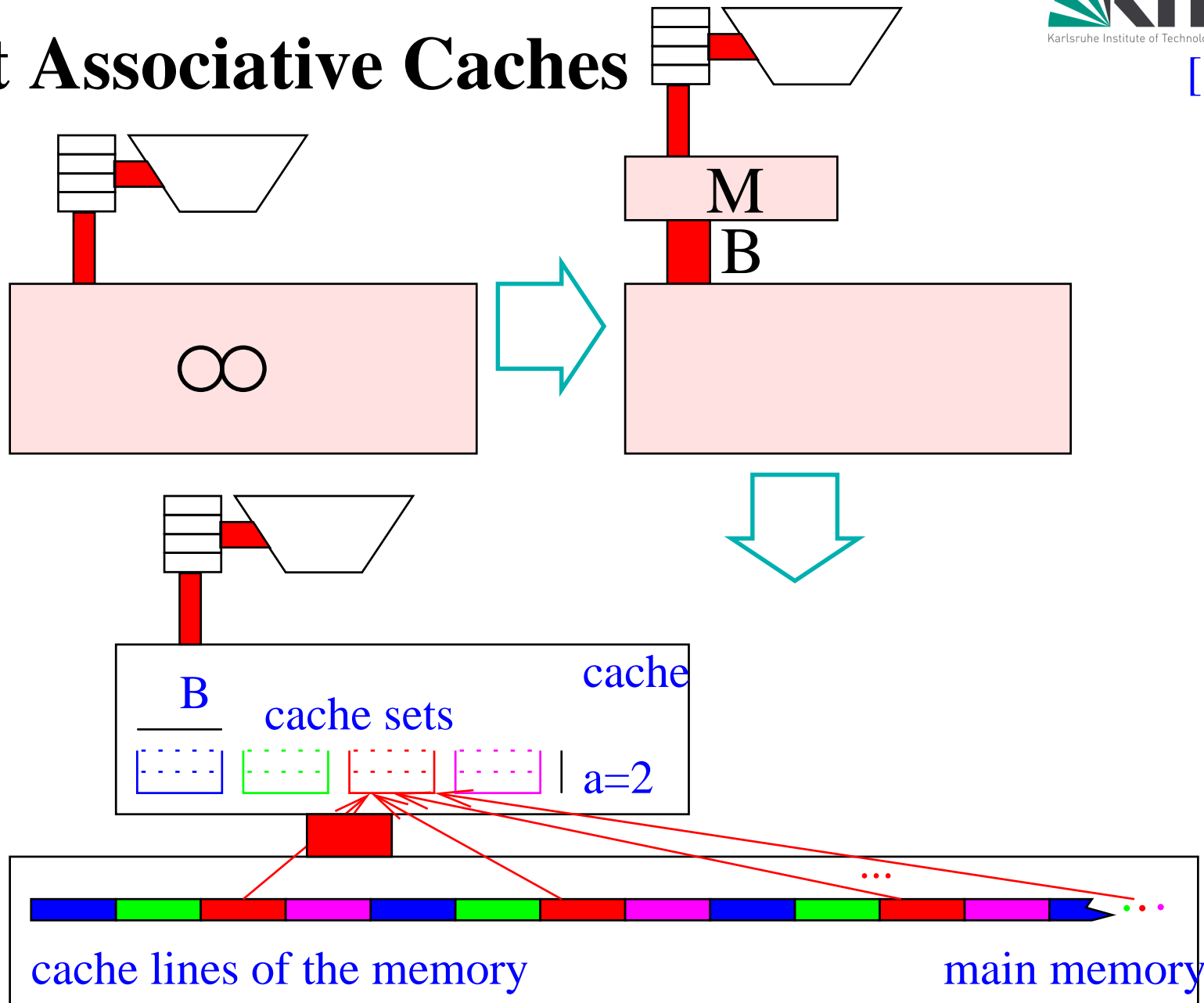
also consider communication

Parallel Disks



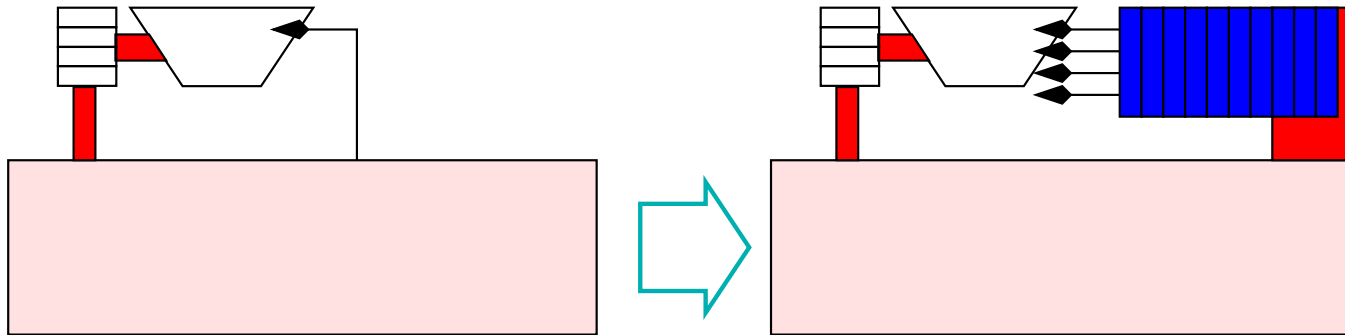
Set Associative Caches

[3]

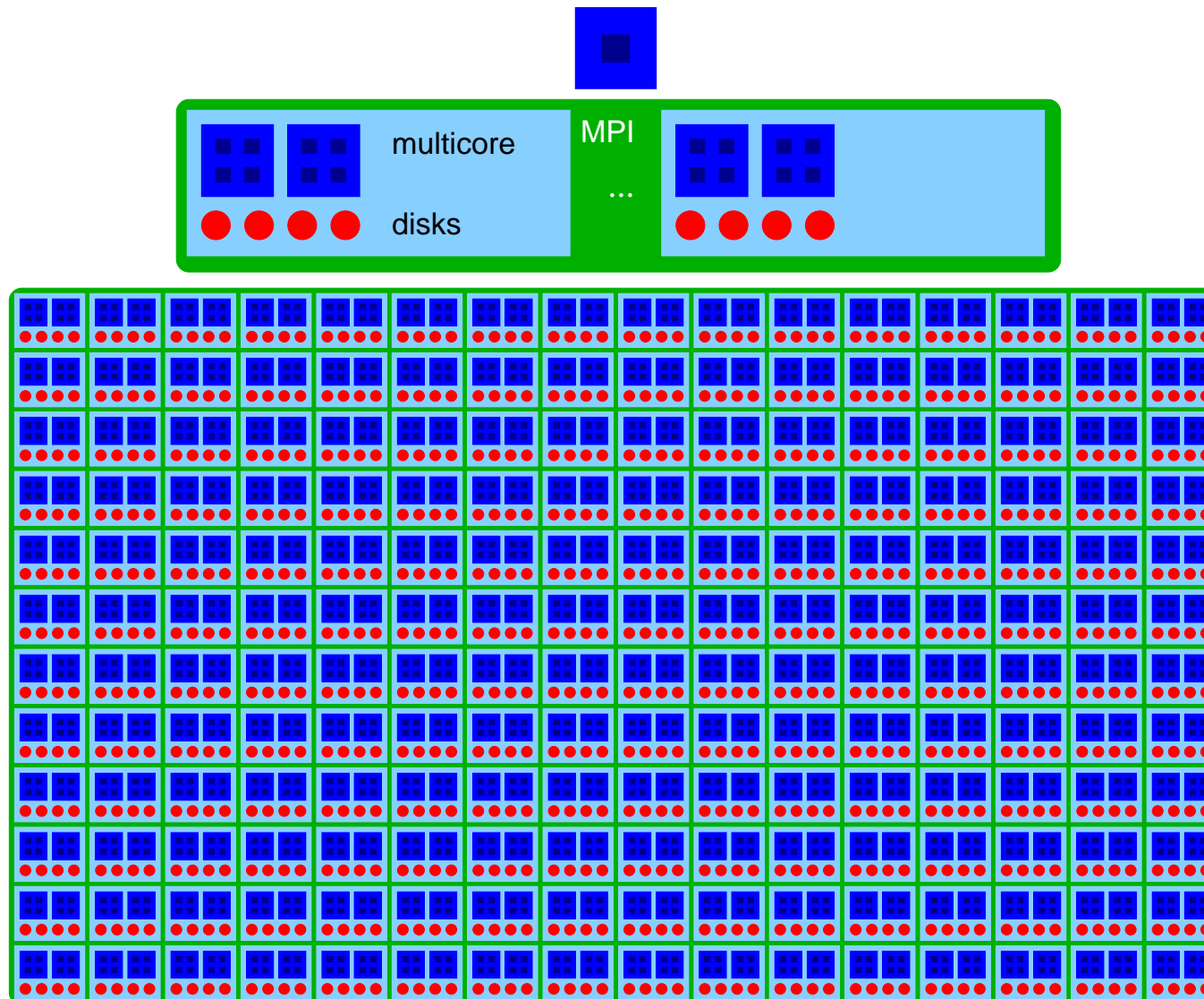


Branch Prediction

[4]

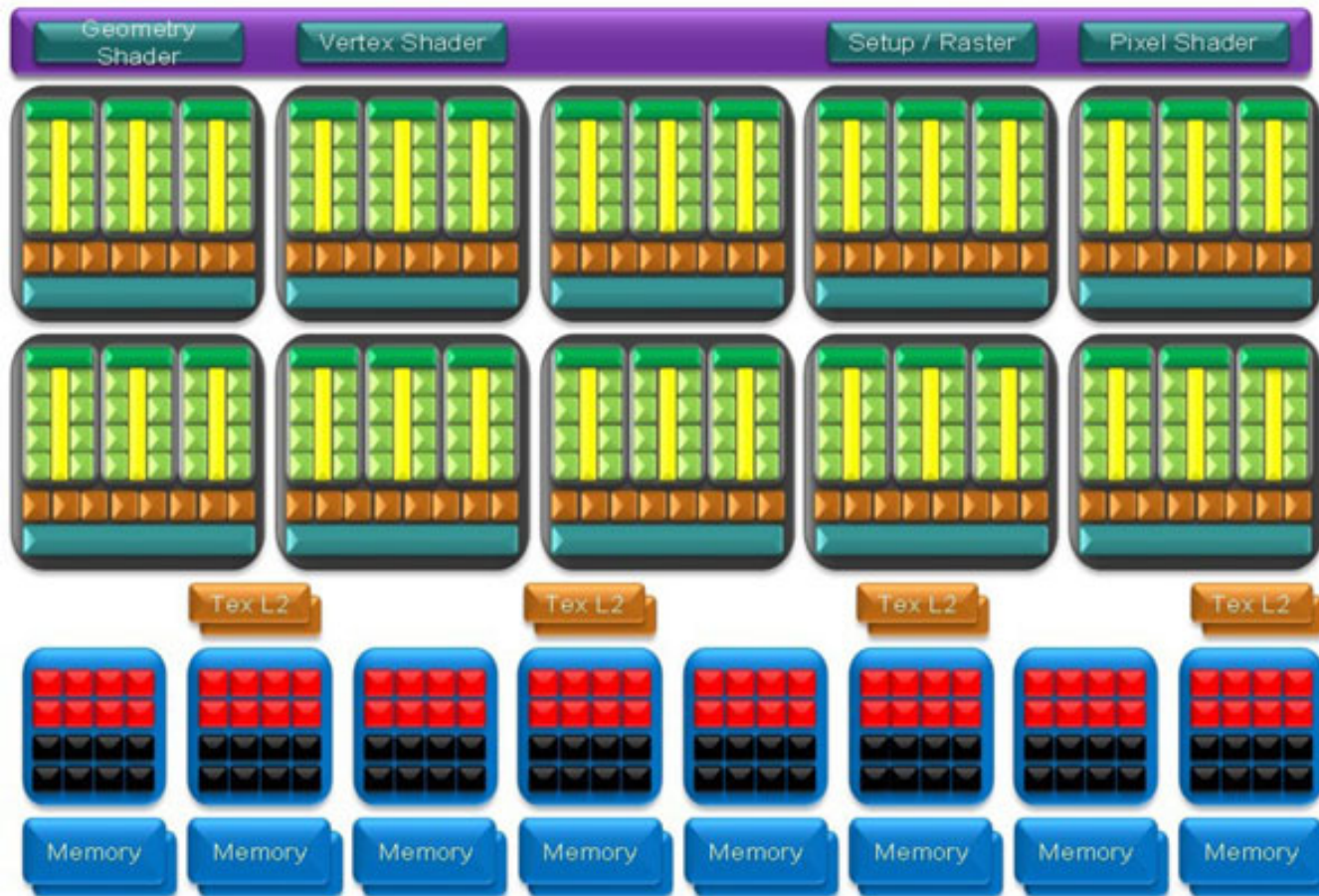


Hierarchical Parallel External Memory [5]



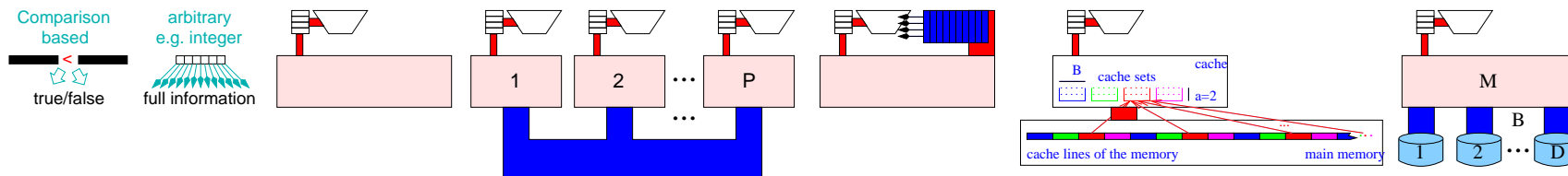
Graphics Processing Units

[6]

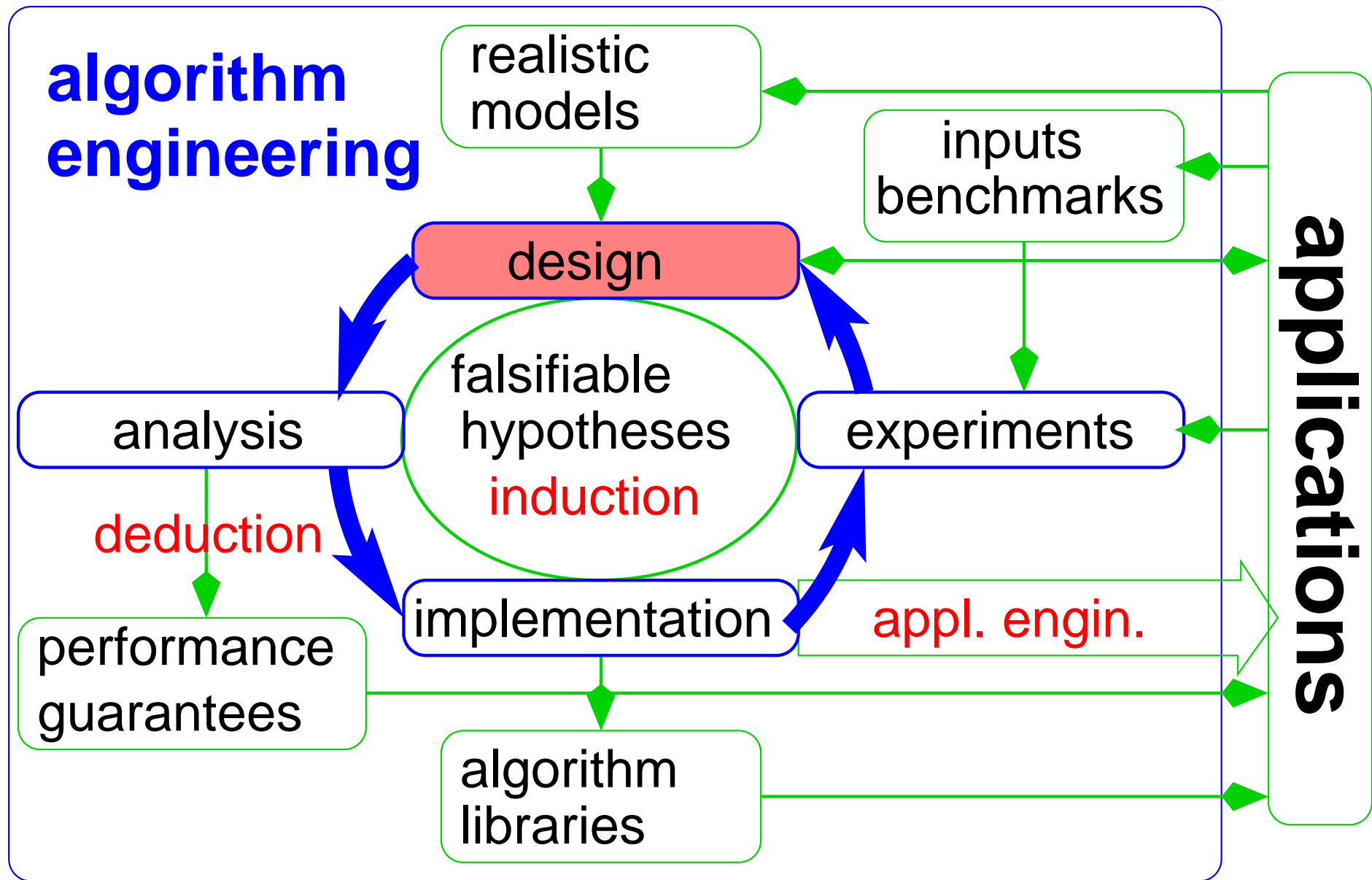


Combining Models?

- design / analyze **one aspect at a time**
- hierarchical combination
- autotuning ?



Or: Model Agnostic Algorithm Design



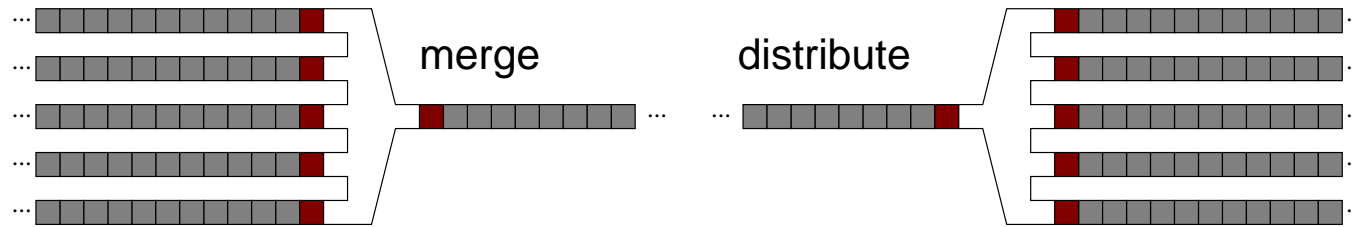
Design

of algorithms that work well in **practice**

- simplicity**
- reuse**
- constant** factors
- exploit **easy** instances

Design – Sorting

simplicity



reuse

disk scheduling, prefetching,
load balancing, sequence partitioning [7, 2, 8, 5]

constant factors

detailed machine model.
(caches, TLBs, registers, branch prediction, ILP) [9, 4]

instances

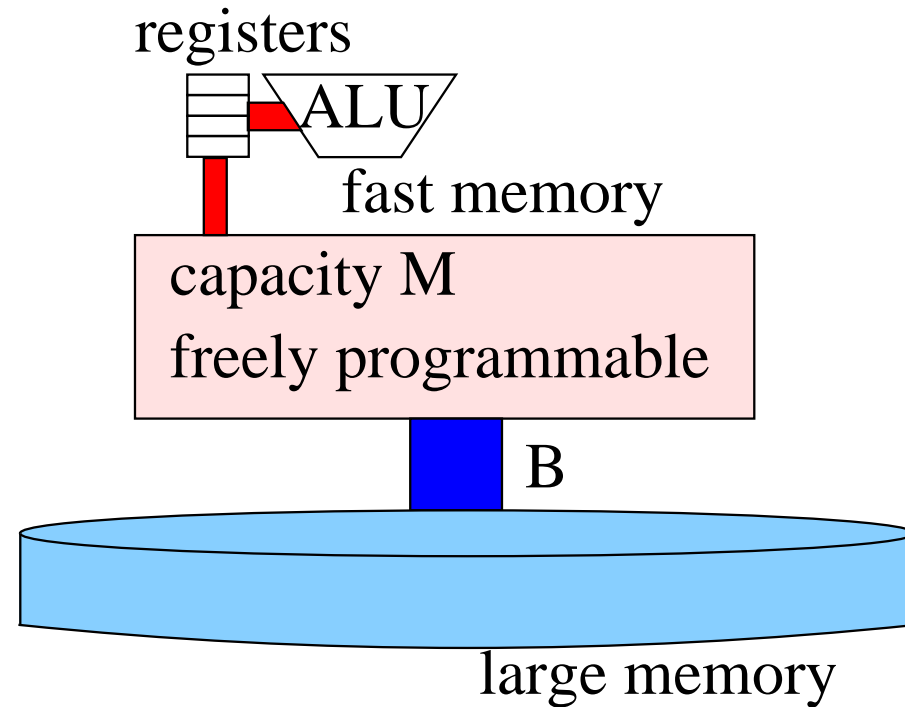
randomization for difficult instances [2, 5]

Example: External Sorting

n : input size

M : internal memory size

B : block size



Procedure externalMerge(*a*, *b*, *c* :File of Element)

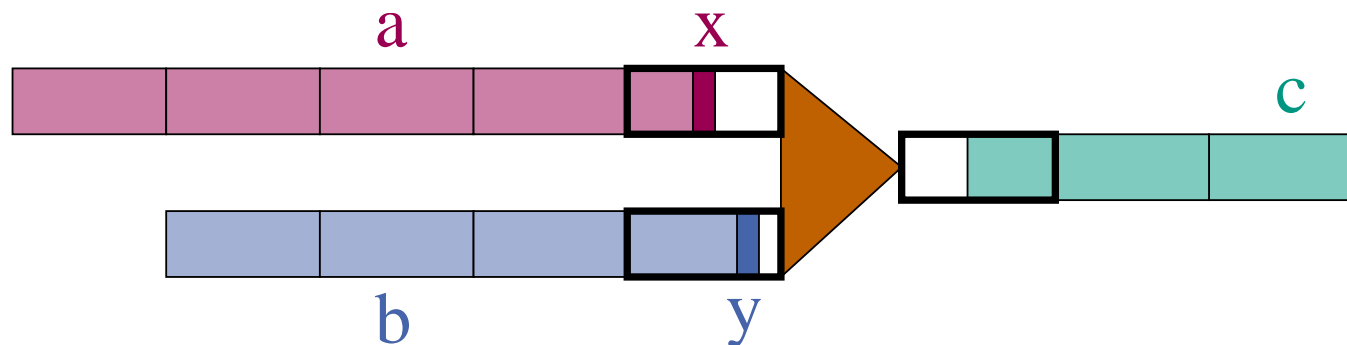
x := *a*.readElement // Assume emptyFile.readElement = ∞

y := *b*.readElement

for *j* := 1 **to** |*a*| + |*b*| **do**

if *x* ≤ *y* **then** *c*.writeElement(*x*); *x* := *a*.readElement

else *c*.writeElement(*y*); *y* := *b*.readElement



External Binary Merging

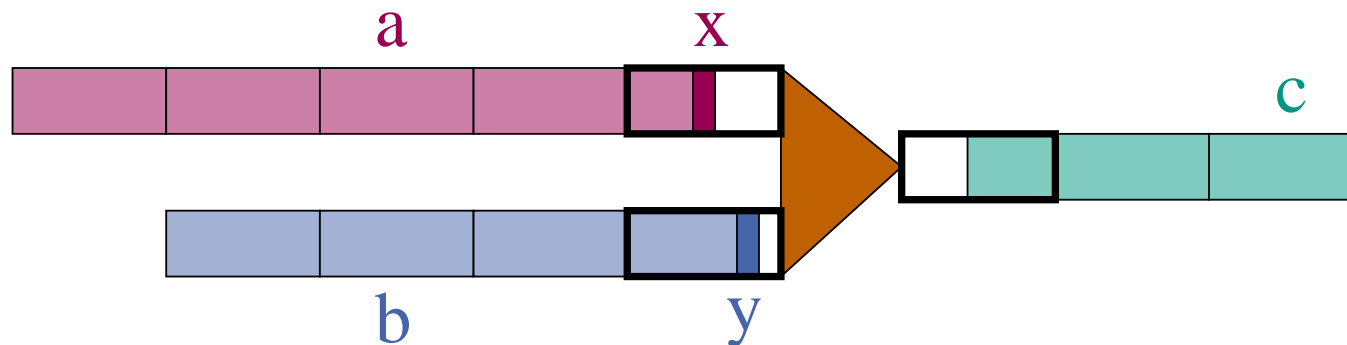
read file a : $\approx |a|/B$.

read file b : $\approx |b|/B$.

write file c : $\approx (|a| + |b|)/B$.

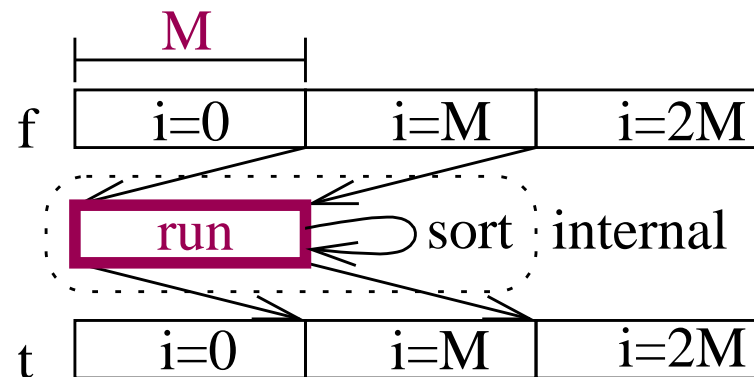
overall:

$$\approx 2 \frac{|a| + |b|}{B}$$



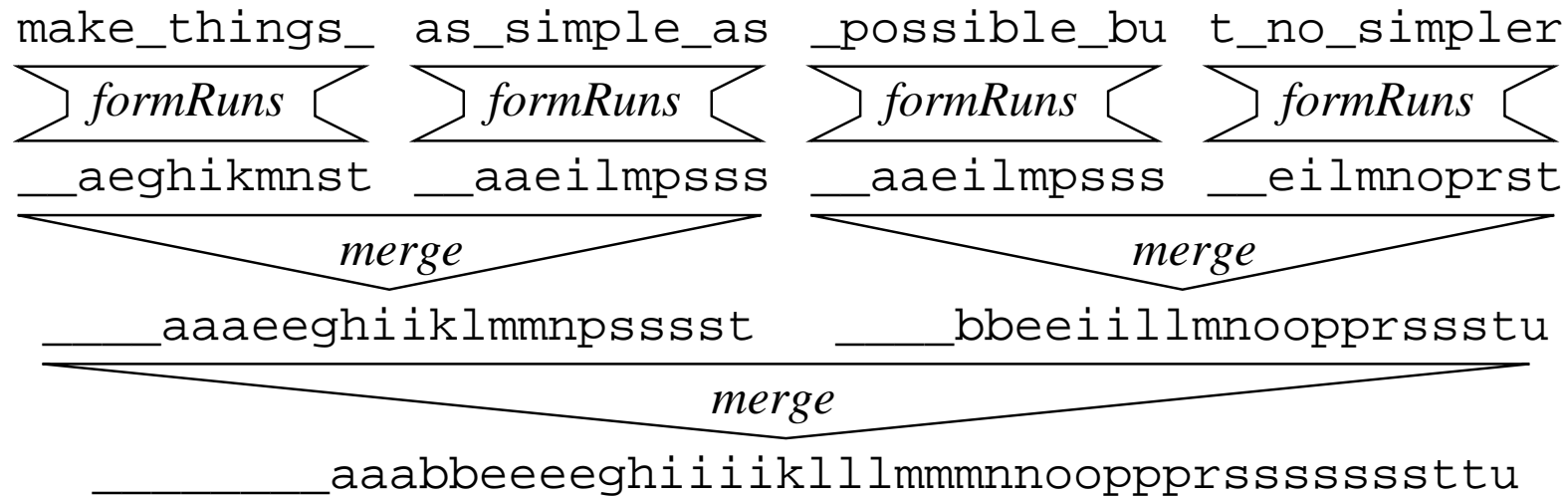
Run Formation

Sort input pieces of size M



$$\text{I/Os: } \approx 2 \frac{n}{B}$$

Sorting by External Binary Merging



Procedure externalBinaryMergeSort

run formation

while more than one run left **do**

merge pairs of runs

output remaining run

// I/Os: \approx

// $2n/B$

// $\lceil \log \frac{n}{M} \rceil \times$

// $2n/B$

// $\Sigma : 2 \frac{n}{B} \left(1 + \lceil \log \frac{n}{M} \rceil \right)$

Example Numbers: PC 2019

$n = 2^{41}$ Byte (2 TB) , i.e., 4 TB HDD capacity

$M = 2^{34}$ Byte (16 GB)

$B = 2^{22}$ Byte (4 MB)

one I/O needs 2^{-5} s (31.25 ms)

$$\begin{aligned}\text{time} &= 2 \frac{n}{B} \left(1 + \left\lceil \log \frac{n}{M} \right\rceil \right) \cdot 2^{-5} \text{s} \\ &= 2 \cdot 2^{19} \cdot (1 + 7) \cdot 2^{-5} \text{s} = 2^{18} \text{s} \approx 18\text{h}\end{aligned}$$

Idea: 8 passes \rightsquigarrow 2 passes

Multiway Merging

Procedure multiwayMerge(a_1, \dots, a_k, c :File of Element)

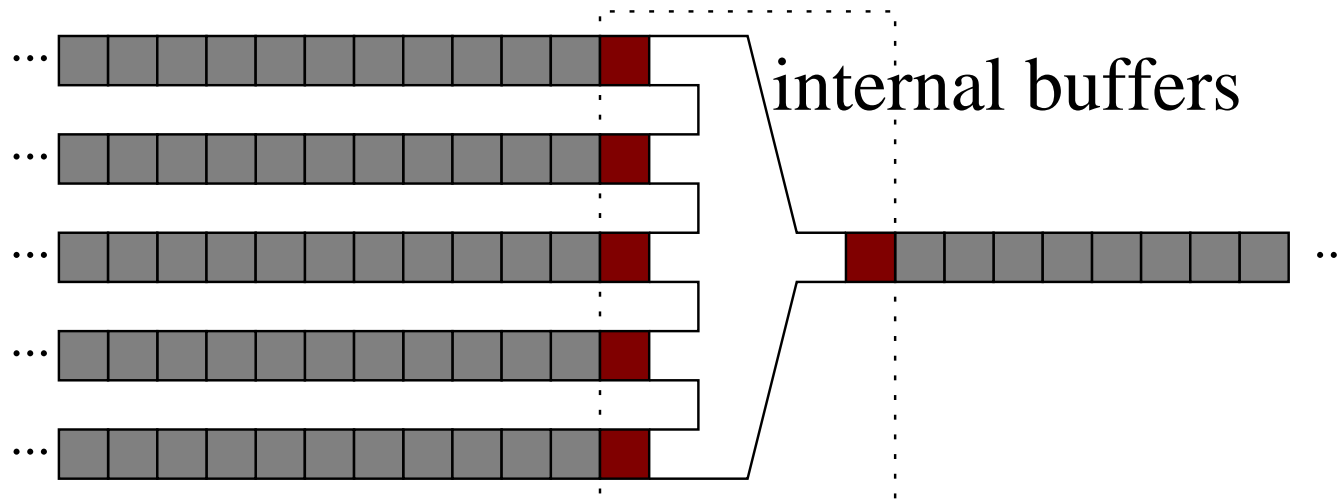
for $i := 1$ **to** k **do** $x_i := a_i$.readElement

for $j := 1$ **to** $\sum_{i=1}^k |a_i|$ **do**

 find $i \in 1..k$ that minimizes x_i // no I/Os!, $\mathcal{O}(\log k)$ time

c .writeElement(x_i)

$x_i := a_i$.readElement



Multiway Merging – Analysis

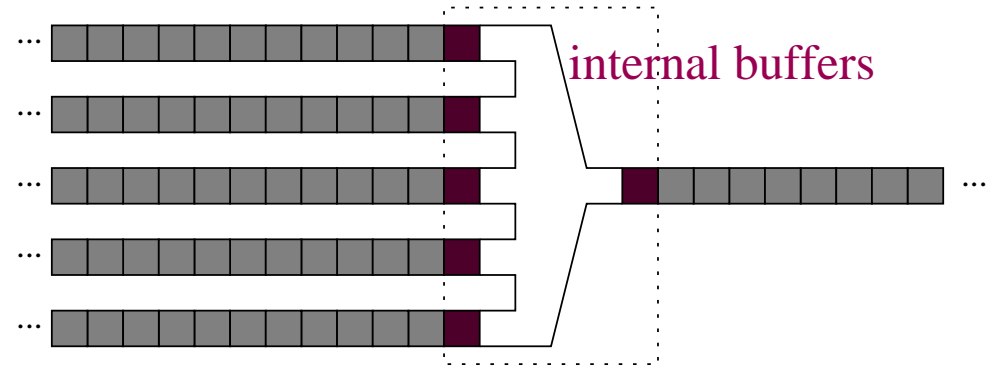
I/Os: read file a_i : $\approx |a_i|/B$.

write file c : $\approx \sum_{i=1}^k |a_i|/B$

overall:

$$\leq \approx 2 \frac{\sum_{i=1}^k |a_i|}{B}$$

constraint: We need $k + 1$ buffer blocks, i.e., $k + 1 < M/B$

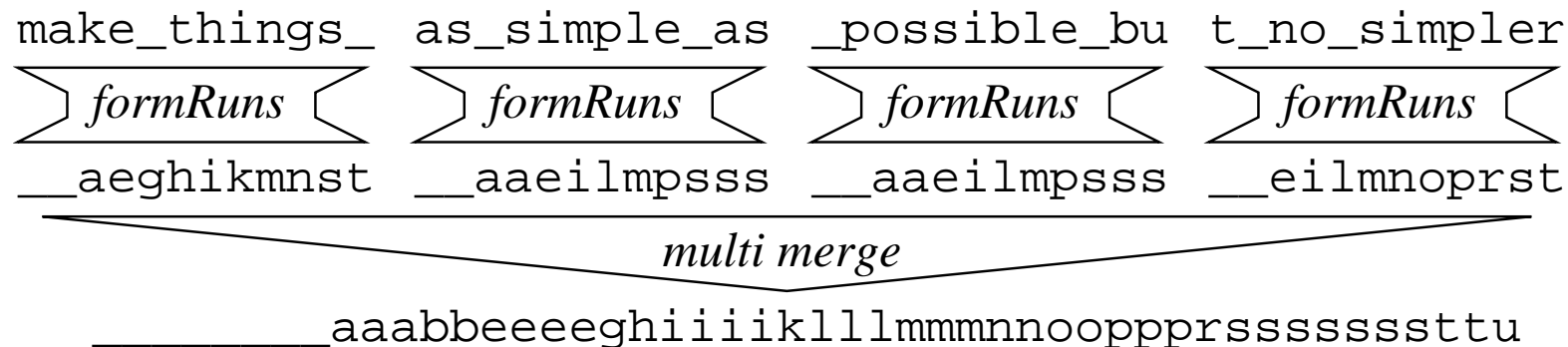


Sorting by Multiway-Merging

- sort $\lceil n/M \rceil$ runs with M elements each $2n/B$ I/Os
- merge M/B runs at a time $2n/B$ I/Os
- until a single run remains $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ merging phases

overall

$$\text{sort}(n) := \frac{2n}{B} \left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right) \text{ I/Os}$$



External Sorting by Multiway-Merging

More than one merging phase?:

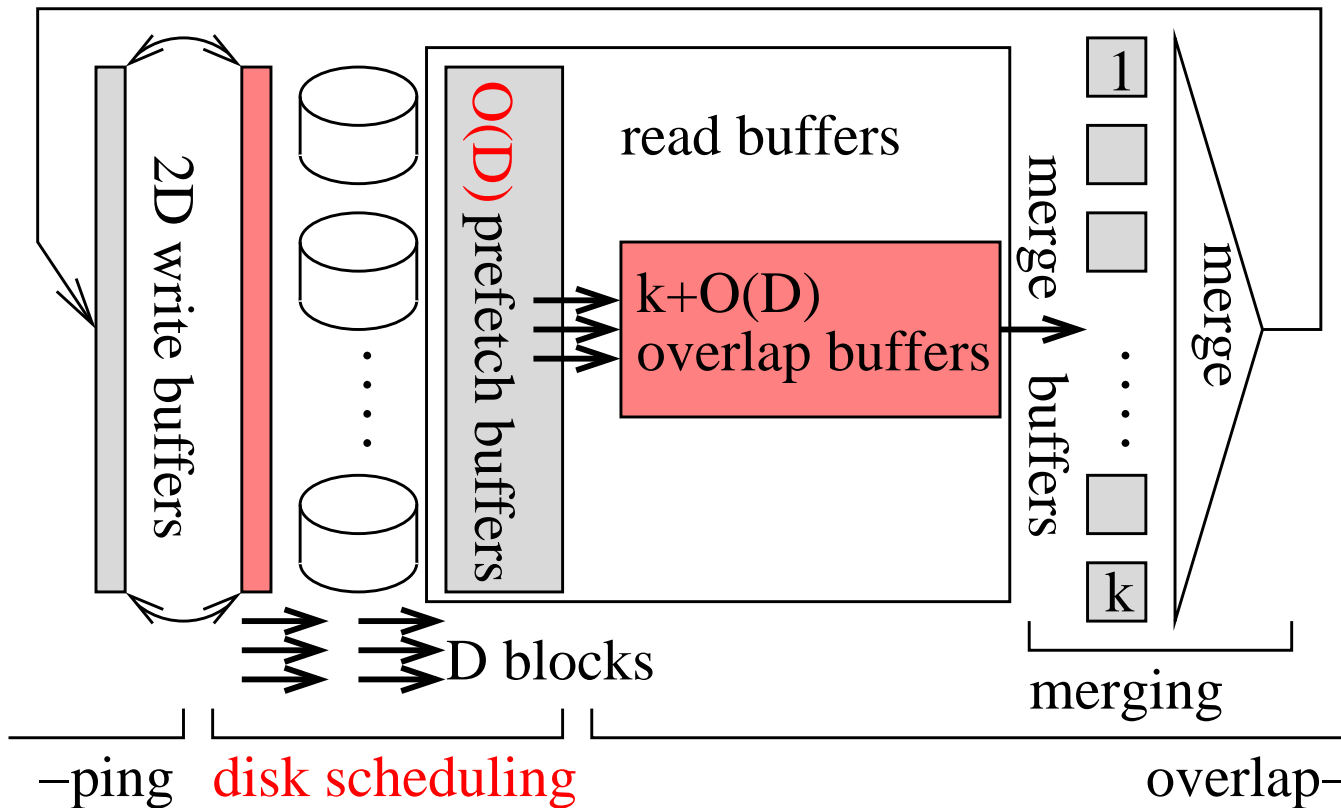
Not for the hierarchy main memory, hard disk.

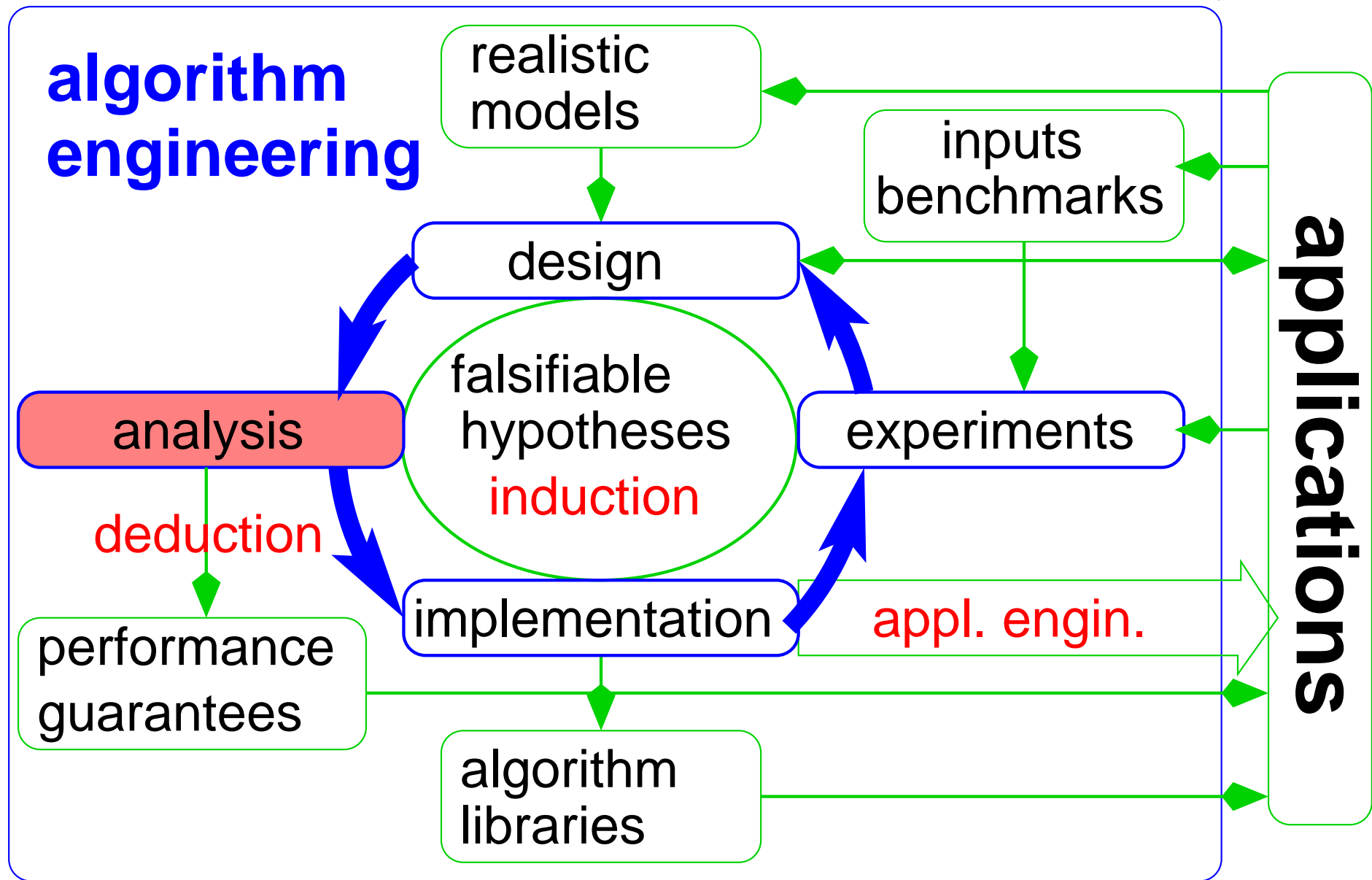
$$\text{reason: } \frac{\overbrace{M}^{>4000}}{B} > \frac{\overbrace{\text{RAM Euro/bit}}{\approx 207}}{\text{Platte Euro/bit}}$$

Currently $4000 > 207$

More on Multiway Mergesort – Parallel Disks

- Randomized Striping [2]
- Optimal Prefetching [2]
- Overlapping of I/O and Computation [7]

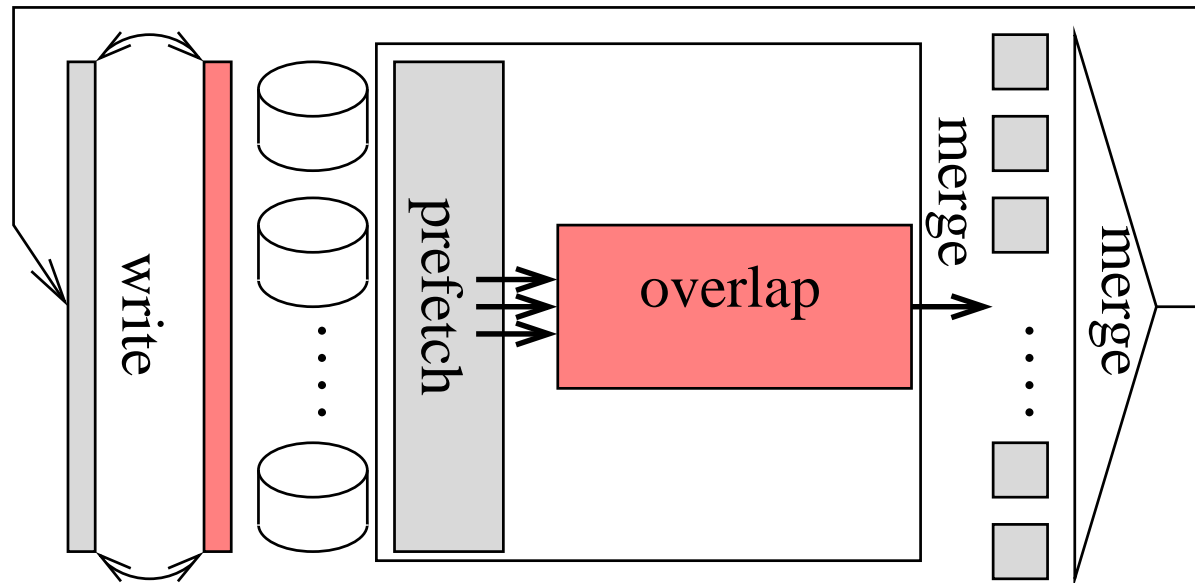




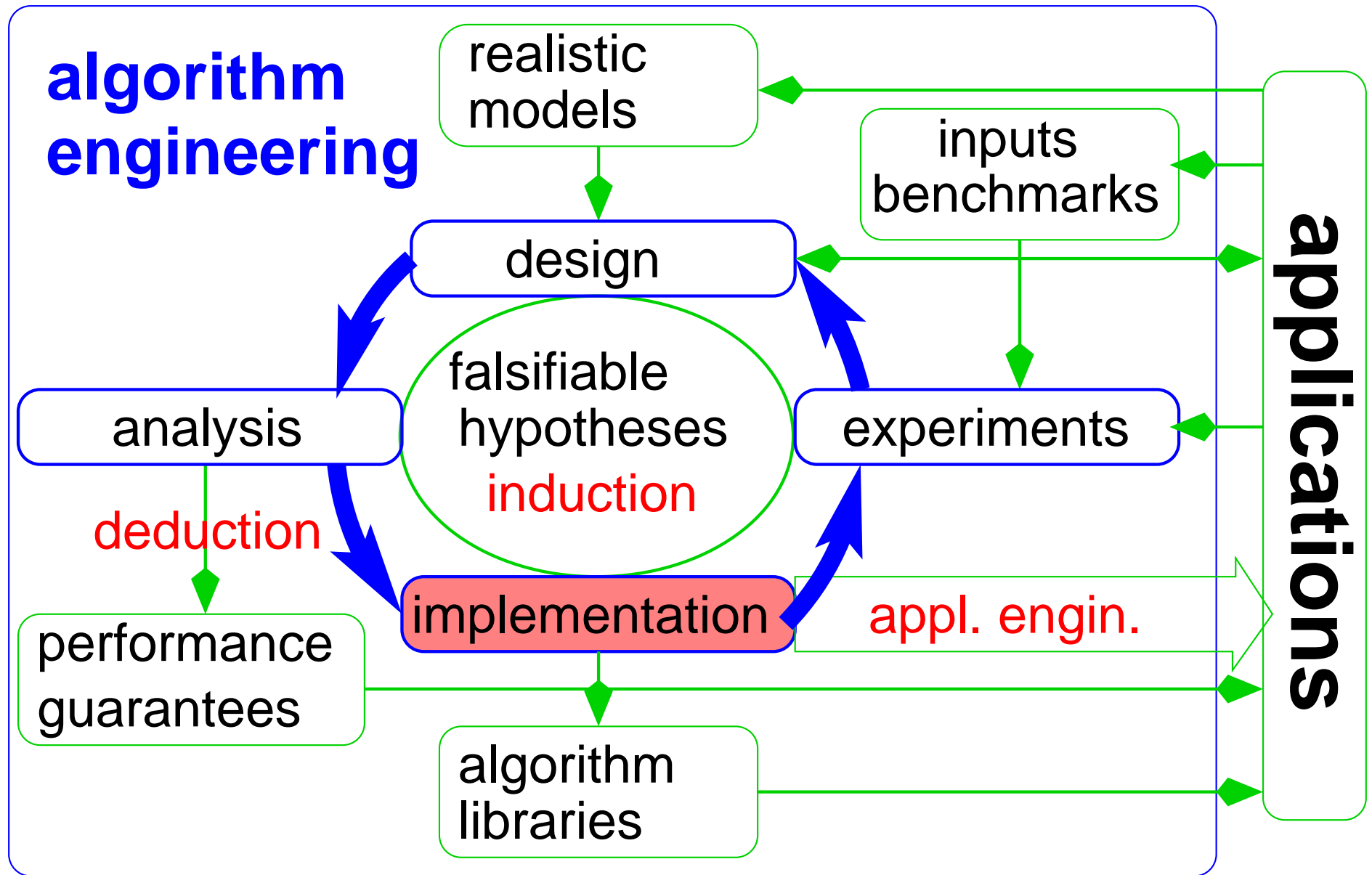
Analysis

- Constant factors matter
- Beyond worst case analysis
- Practical algorithms might be difficult to analyze
(randomization, meta heuristics, ...)

Analysis – Sorting



- Constant factors matter** $(1 + o(1)) \times$ lower bound
[2, 5] I/Os for parallel (disk) external sorting
- Beyond worst case analysis** adaptive sorting
- Practical algorithms** might be difficult to analyze **Open:**
[2] greedy algorithm for parallel disk prefetching
[Knuth@48]



Implementation

sanity check for algorithms !

Challenges

Semantic gaps:

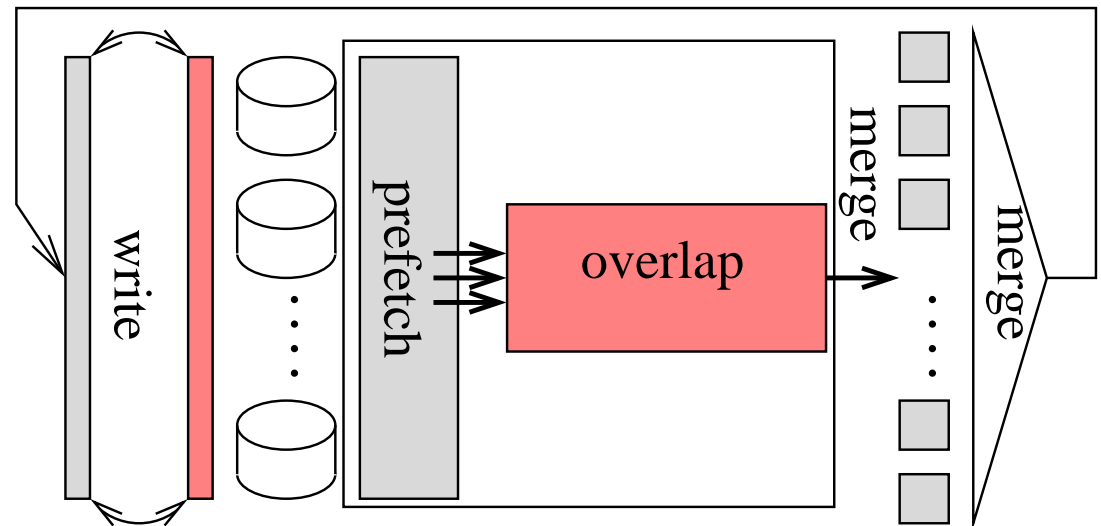
Abstract algorithm

↔

C++...

↔

hardware



Small constant factors:

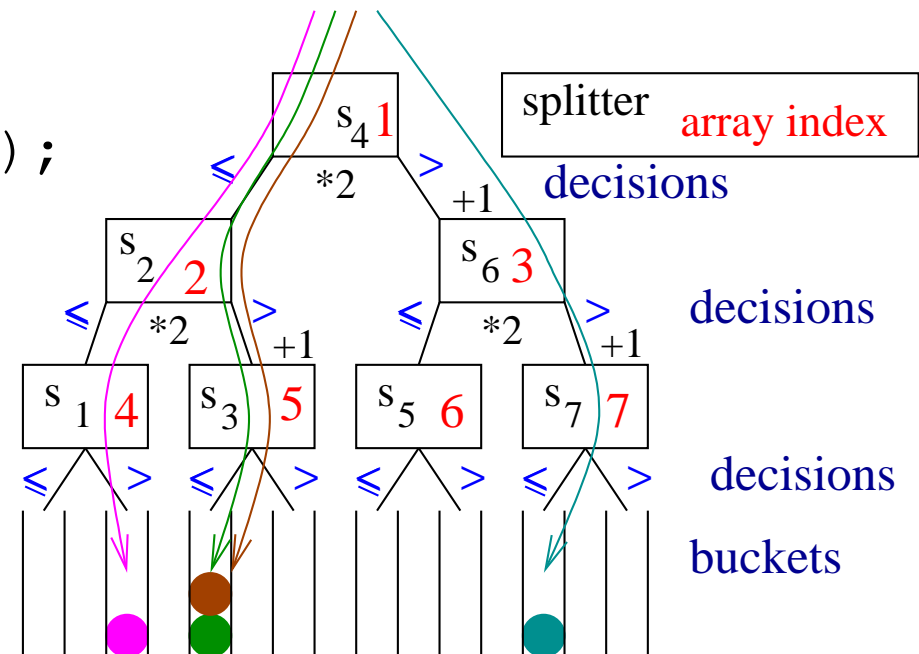
compare highly tuned competitors

Example: Inner Loops Sample Sort

[4]

```

template <class T>
void findOraclesAndCount (const T* const a,
    const int n, const int k, const T* const s,
    Oracle* const oracle, int* const bucket) {
{ for (int i = 0; i < n; i++)
    int j = 1;
    while (j < k) {
        j = j*2 + (a[i] > s[j]);
    }
    int b = j-k;
    bucket[b]++;
    oracle[i] = b;
}
}
    
```

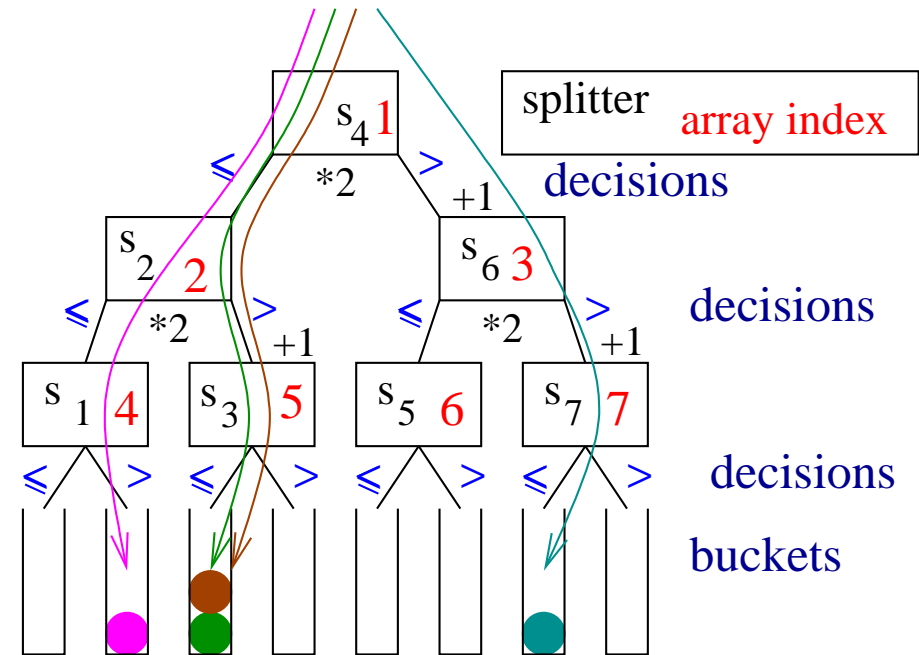


[4]

Example: Inner Loops Sample Sort

```

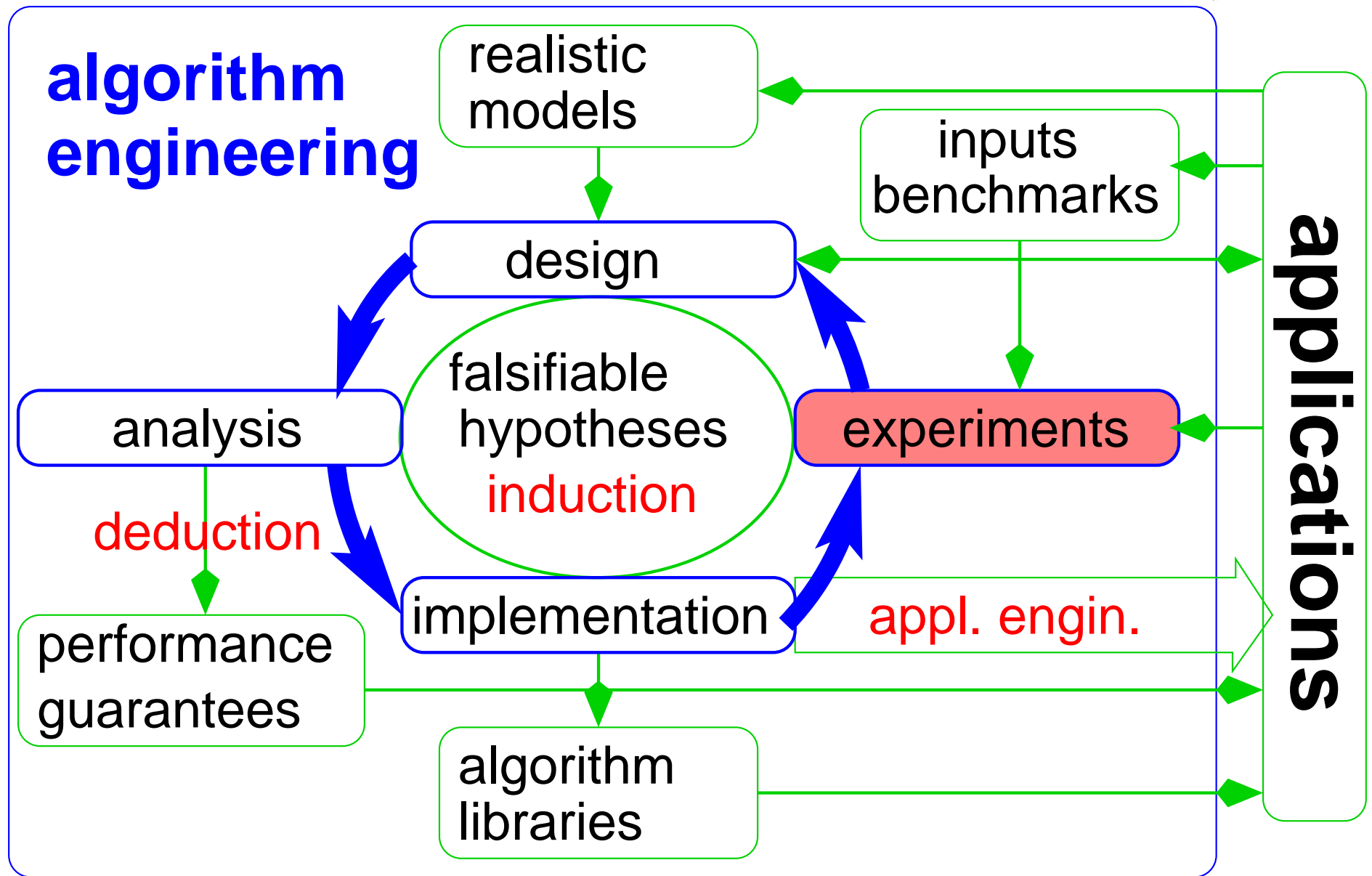
template <class T>
void findOraclesAndCountUnrolled(...) {
    for (int i = 0; i < n; i++)
        int j = 1;
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        int b = j-k;
        bucket[b]++;
        oracle[i] = b;
    } }
    
```



Example: Inner Loops Sample Sort

[4]

```
template <class T>
void findOraclesAndCountUnrolled2 ([...]) {
    for (int i = n & 1; i < n; i+=2) {\
        int j0 = 1;                int j1 = 1;
        T ai0 = a[i];              T ai1 = a[i+1];
        j0=j0*2+(ai0>s[j0]);        j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);        j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);        j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);        j1=j1*2+(ai1>s[j1]);
        int b0 = j0-k;              int b1 = j1-k;
        bucket[b0]++;               bucket[b1]++;
        oracle[i] = b0;             oracle[i+1] = b1;
    } }
```



Experiments

- central for AE in science
reproducibility, careful comparisons, careful preparation of evidence
- also important in applications – just more informal
- careful **planning**
- careful **interpretation**
- close AE cycle fast**

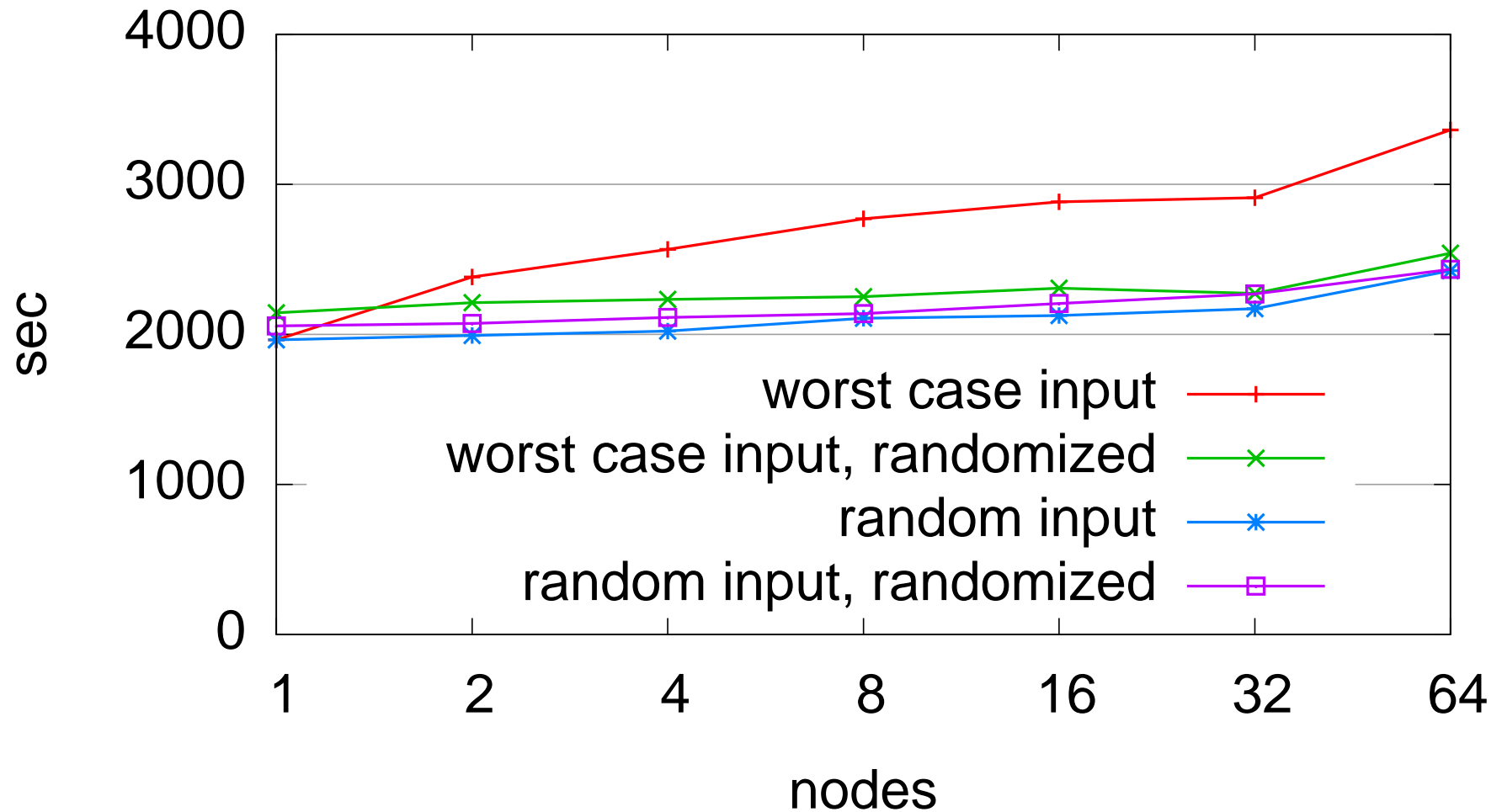
Experiments

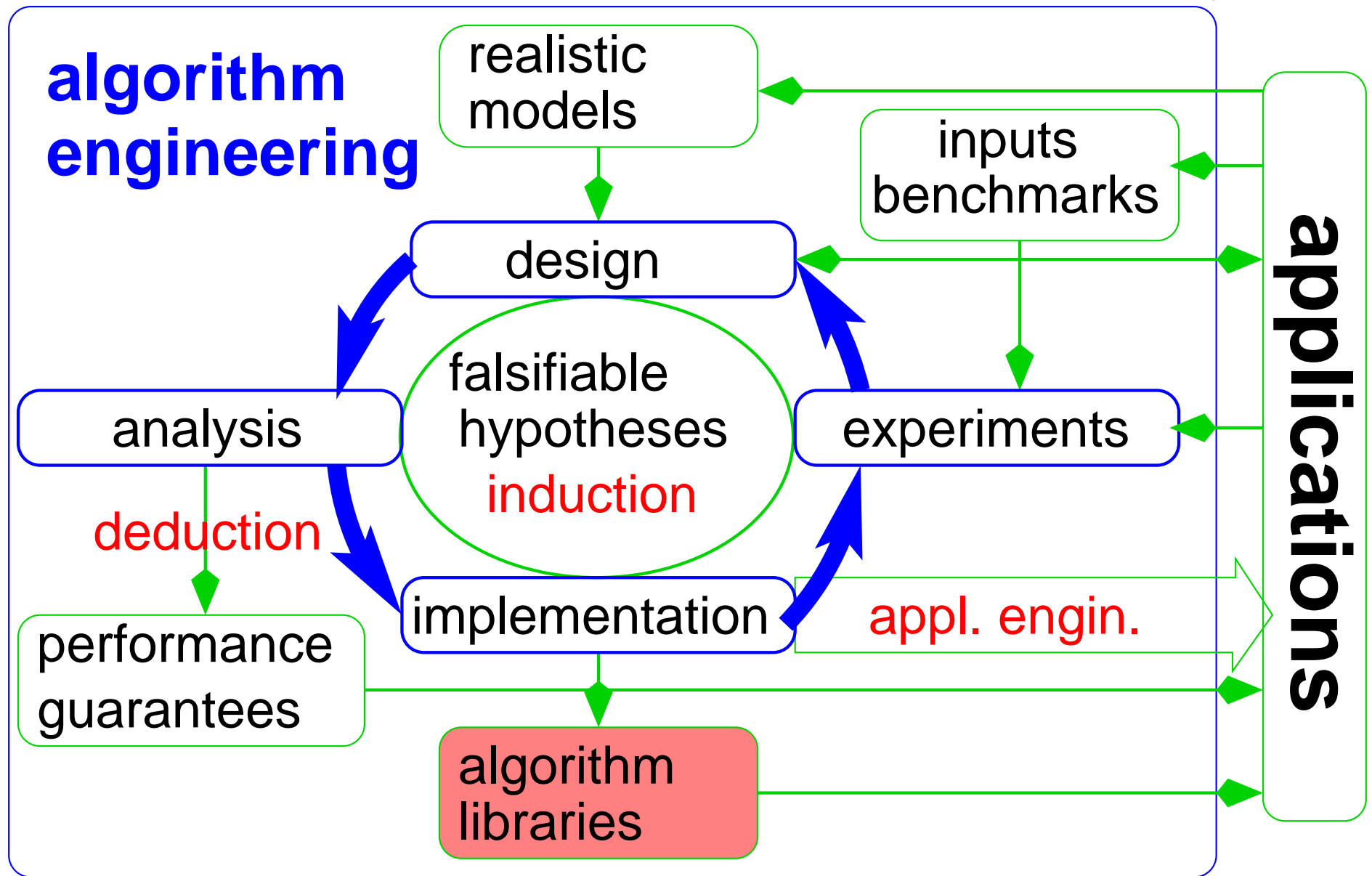
- sometimes a good **surrogate for analysis**
- too much** rather than too little **output data**
- reproducibility** (10 years!)
- software engineering**

Example, Parallel External Sorting

sort 100GiB per node

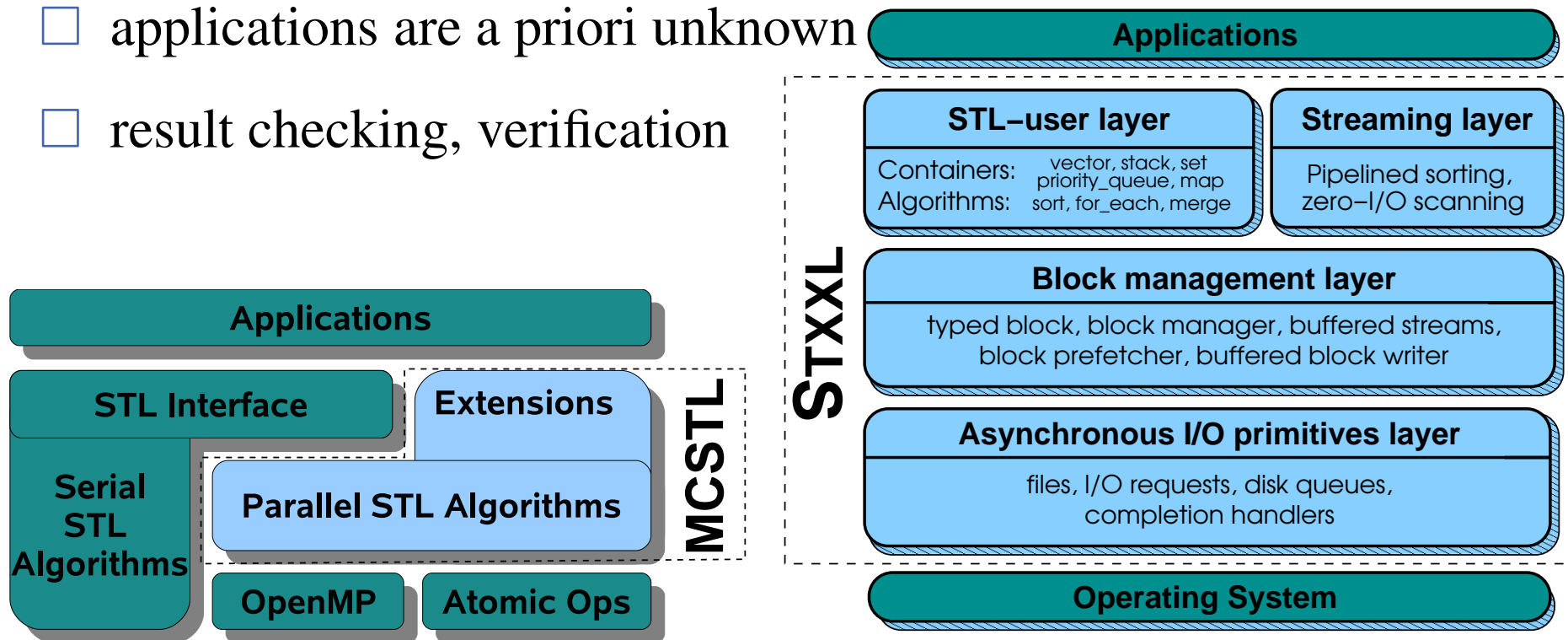
[7]



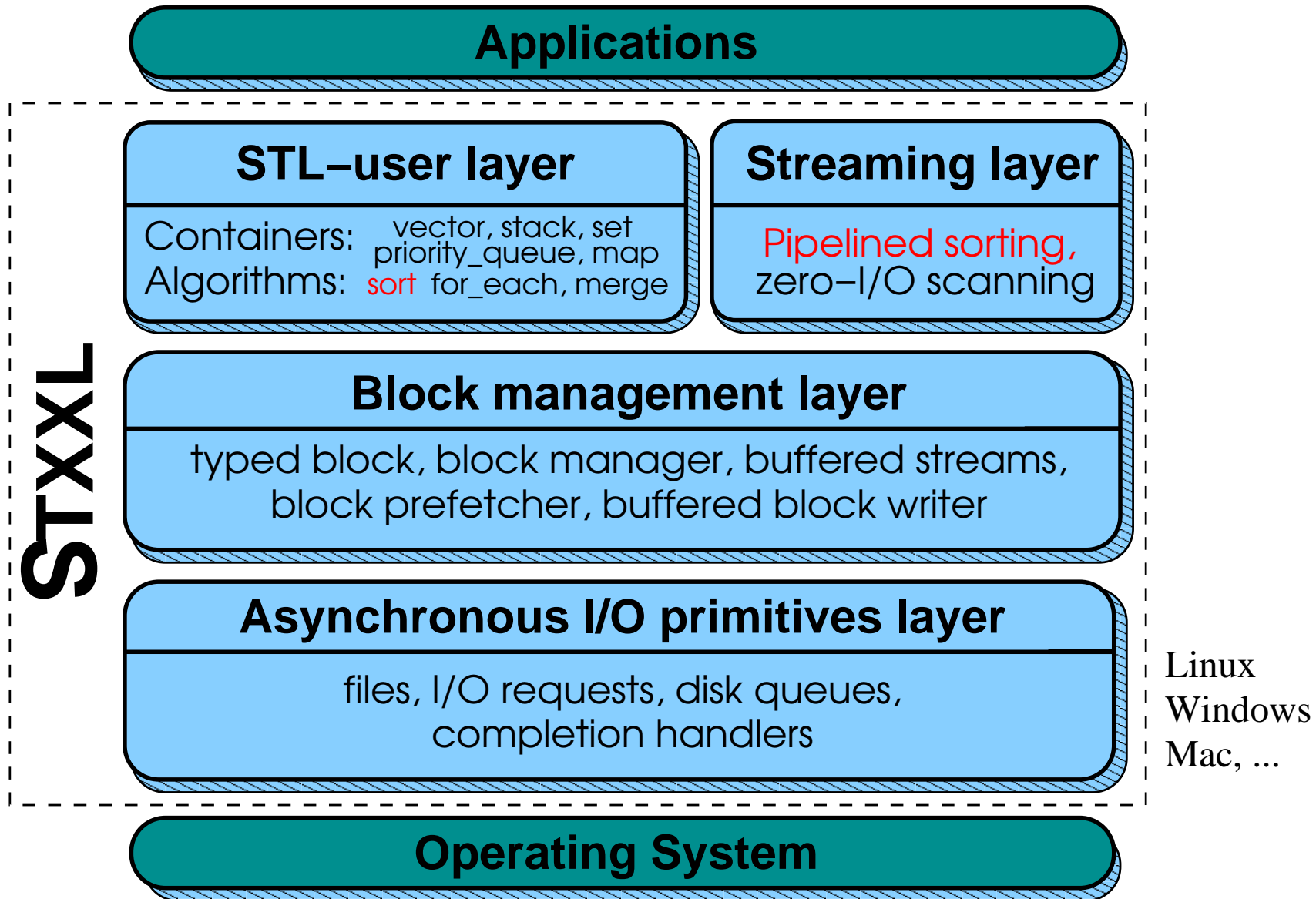


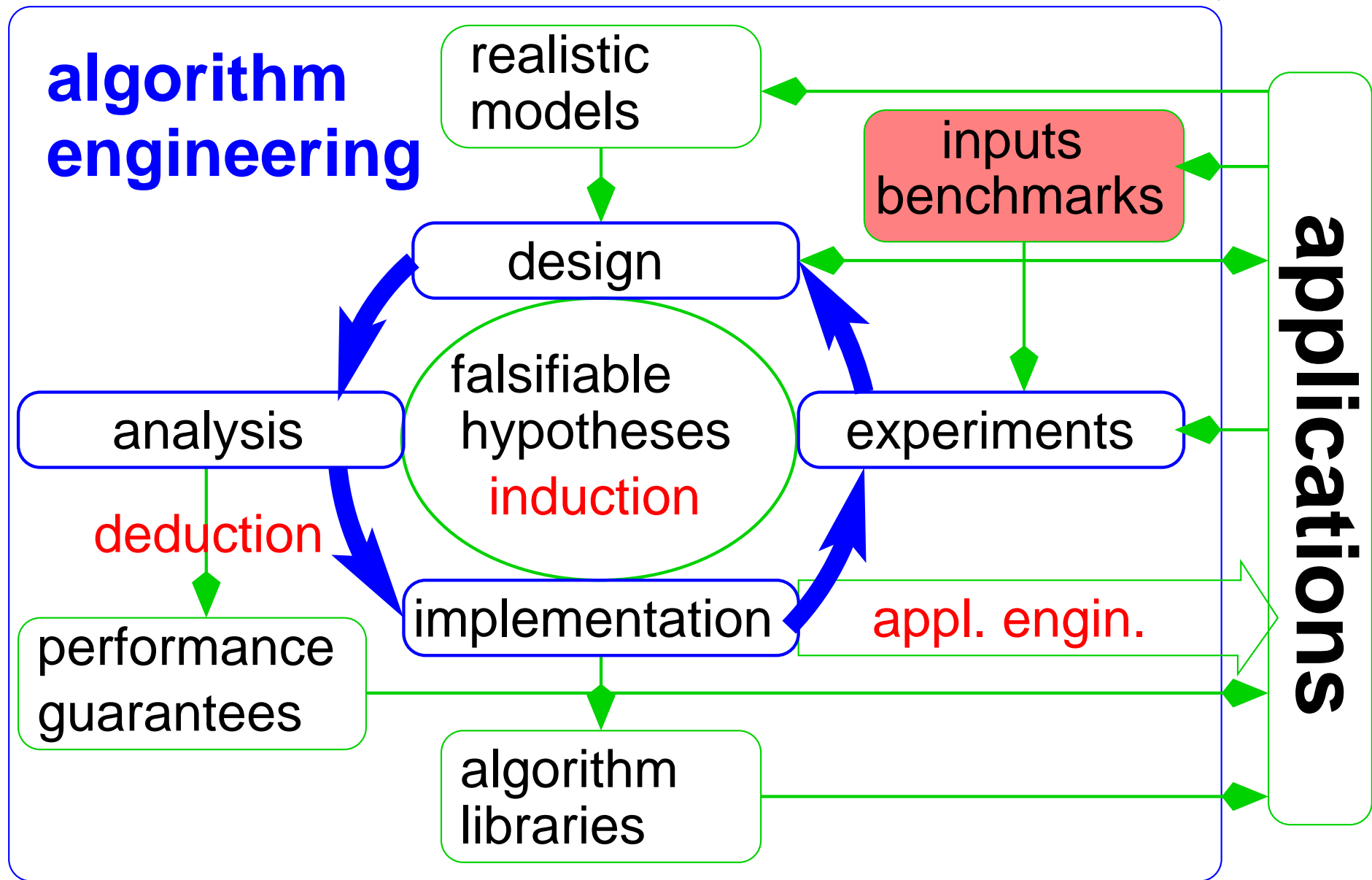
Algorithm Libraries — Challenges

- software engineering , e.g. CGAL [www.cgal.org]
- standardization, e.g. java.util, C++ STL and BOOST
- performance ↔ generality ↔ simplicity
- applications are a priori unknown
- result checking, verification



Example: External Sorting

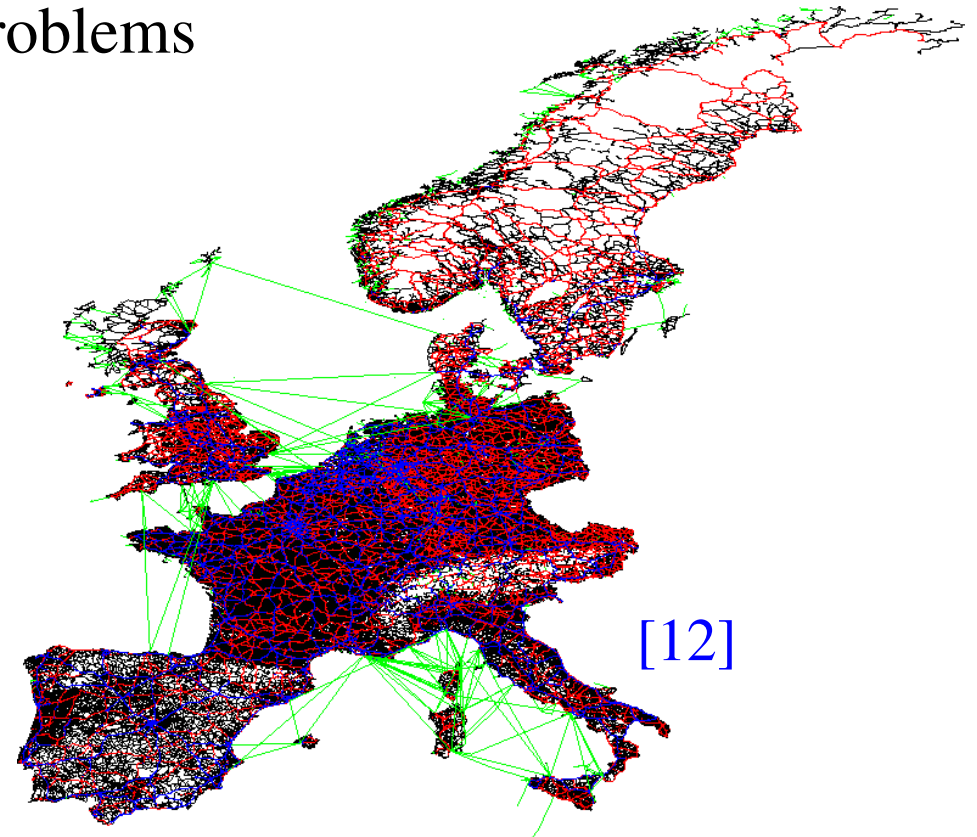




Problem Instances

Benchmark instances for **NP-hard** problems

- TSP
- Steiner-Tree
- SAT
- set covering
- graph partitioning
- ...



have proved essential for development of practical algorithms

Strange: much less real world instances for **polynomial problems** (MST, shortest path, max flow, matching...)

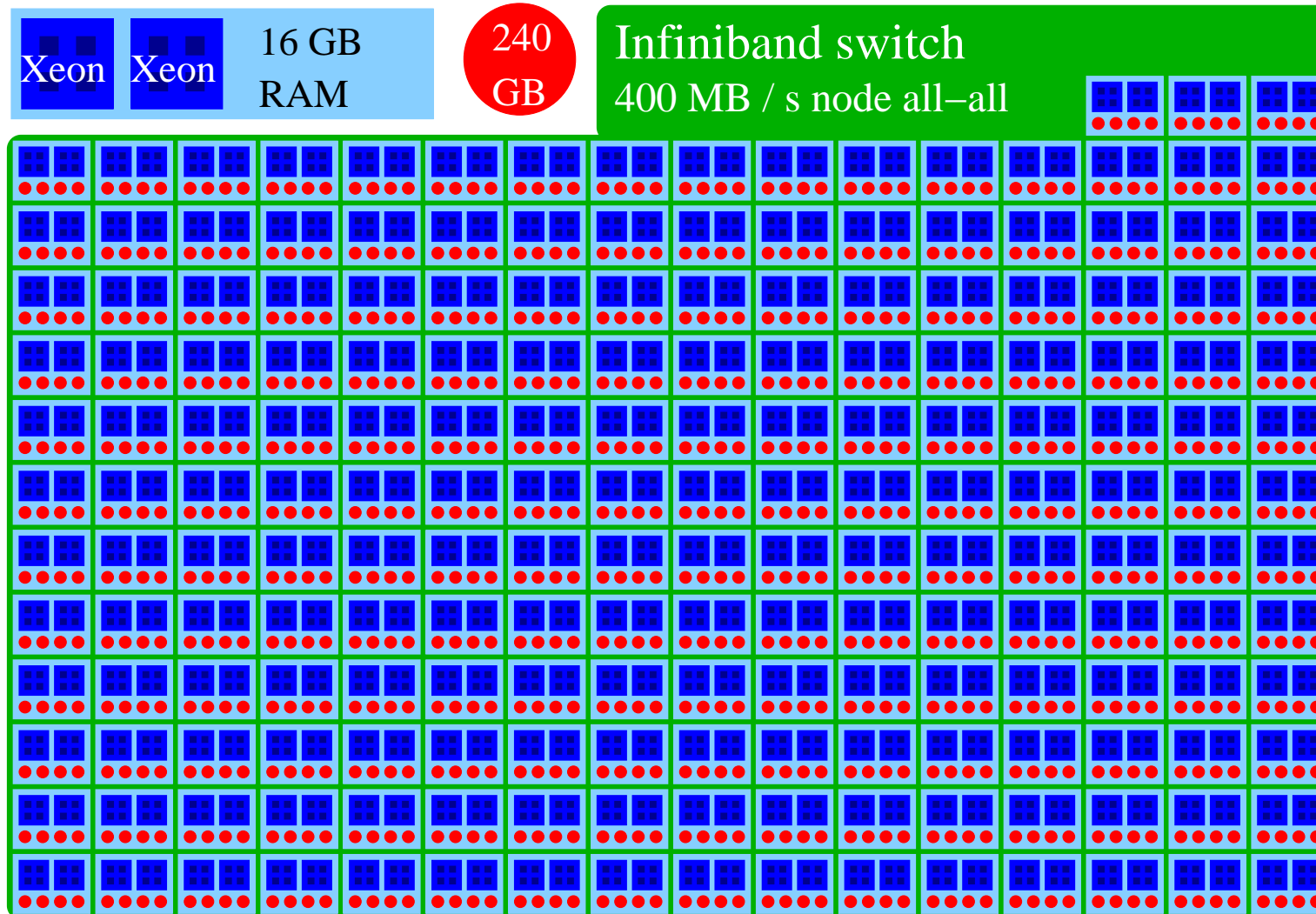
Example: Sorting Benchmark (Indy)

100 byte records, 10 byte random keys, with file I/O

Category	data volume	performance	improvement
GraySort	100 TB	564 GB / min	17×
MinuteSort	955 GB	955 GB / min	> 10×
JouleSort	1 000 GB	13 400 Recs/Joule	4×
JouleSort	100 GB	35 500 Recs/Joule	3×
JouleSort	10 GB	34 300 Recs/Joule	3×

Also: PennySort

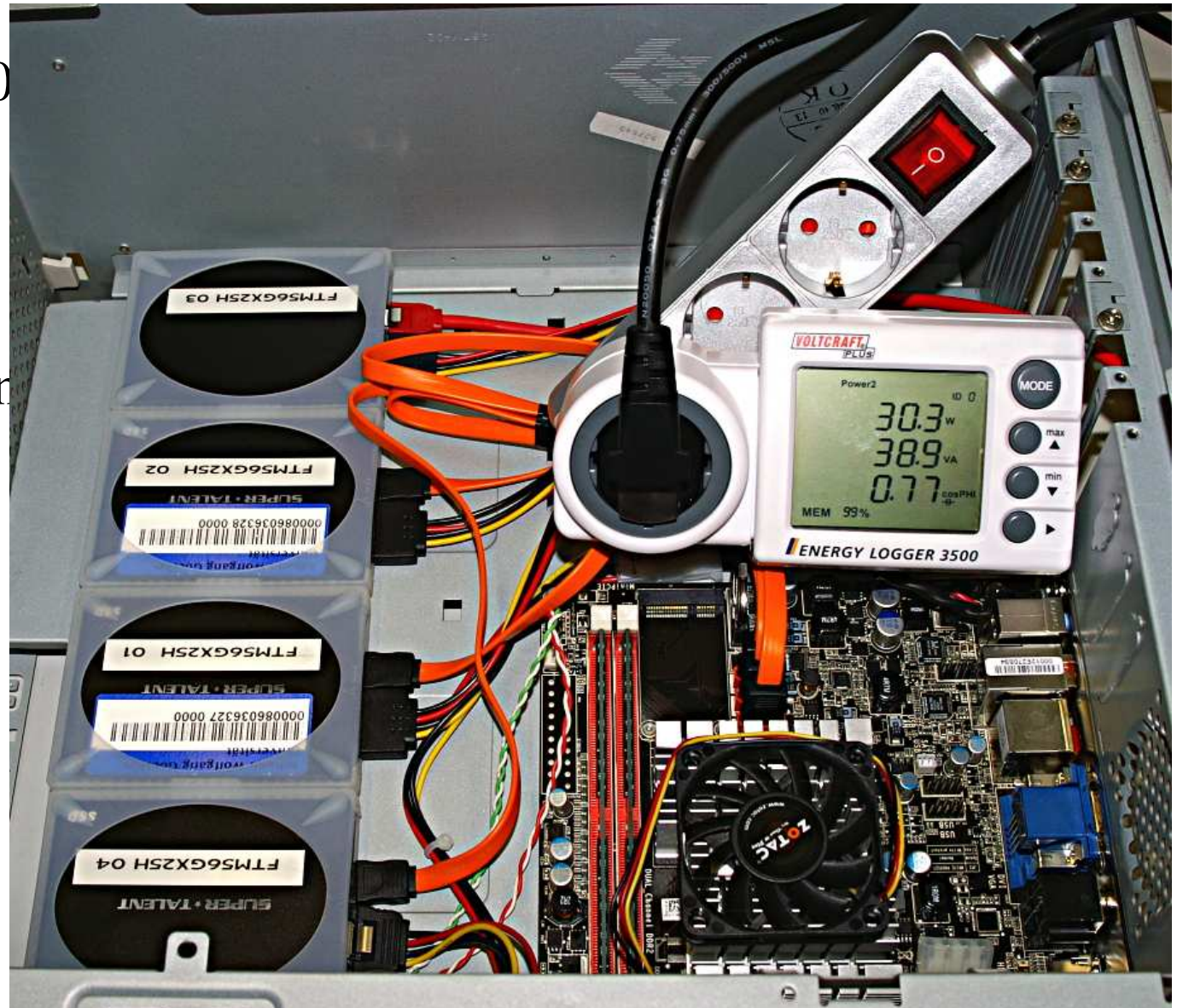
GraySort: *inplace* multiway mergesort, *exact splitting* [5]



JouleSort

- Intel Atom N330
- 4 GB RAM
- 4 × 256 GB
SSD (SuperTalent)

Algorithm similar to
GraySort



More Sorting Work in my Group

Model	mmerge	sample s	quicks.	radixs.
GPU		[14]		[14]
distributed memory	[5]	[15, 16]	[17, 16]	
cache	[8]	[4, 18]		[19]
parallel disks	[7]	[20]		
branch mispredictions		[4, 18]	[21]	
parallel string s.	[22, 23]	[24, 22]	[24, 22]	[24, 22]
massively parallel	[15]	[15, 16]	[16, 25]	

Sorting — Overview

- You think you understand **quicksort**?
- Avoiding **branch mispredictions**: Super Scalar Sample Sort
- (Parallel disk) **external** sorting

Quicksort

Function quickSort(s : Sequence of Element) : Sequence of Element

if $|s| \leq 1$ **then return** s // base case

pick $p \in s$ uniformly at random // pivot key

$a := \langle e \in s : e < p \rangle$

$b := \langle e \in s : e = p \rangle$

$c := \langle e \in s : e > p \rangle$

return concatenate(quickSort(a), b , quickSort(c))

Engineering Quicksort

- array
- 2-way-Comparisons
- sentinels** for inner loop
- inplace swaps
- Recursion on **smaller** subproblems
→ $\mathcal{O}(\log n)$ additional space
- break recursion** for small (20–100) inputs, insertion sort
(**not** one big insertion sort)

```

Procedure qSort( $a$  : Array of Element;  $\ell, r$  :  $\mathbb{N}$ ) // Sort  $a[\ell..r]$ 
  while  $r - \ell \geq n_0$  do // Use divide-and-conquer
     $j := \text{pickPivotPos}(a, \ell, r)$ 
    swap( $a[\ell], a[j]$ ) // Helps to establish the invariant
     $p := a[\ell]$ 
     $i := \ell; j := r$ 
    repeat //  $a: \ell \quad i \rightarrow \leftarrow j \quad r$ 
      while  $a[i] < p$  do  $i++$  // Scan over elements (A)
      while  $a[j] > p$  do  $j--$  // on the correct side (B)
      if  $i \leq j$  then swap( $a[i], a[j]$ );  $i++$  ;  $j--$ 
    until  $i > j$  // Done partitioning
    if  $i < \frac{\ell+r}{2}$  then qSort( $a, \ell, j$ );  $\ell := j$ 
    else qSort( $a, i, r$ ) ;  $r := i$ 
  insertionSort( $a[\ell..r]$ ) // faster for small  $r - \ell$ 

```

Picking Pivots Painstakingly — Theory

probabilistically: Expected $1.4n \log n$ element comparisons

median of three: Expected $1.2n \log n$ element comparisons

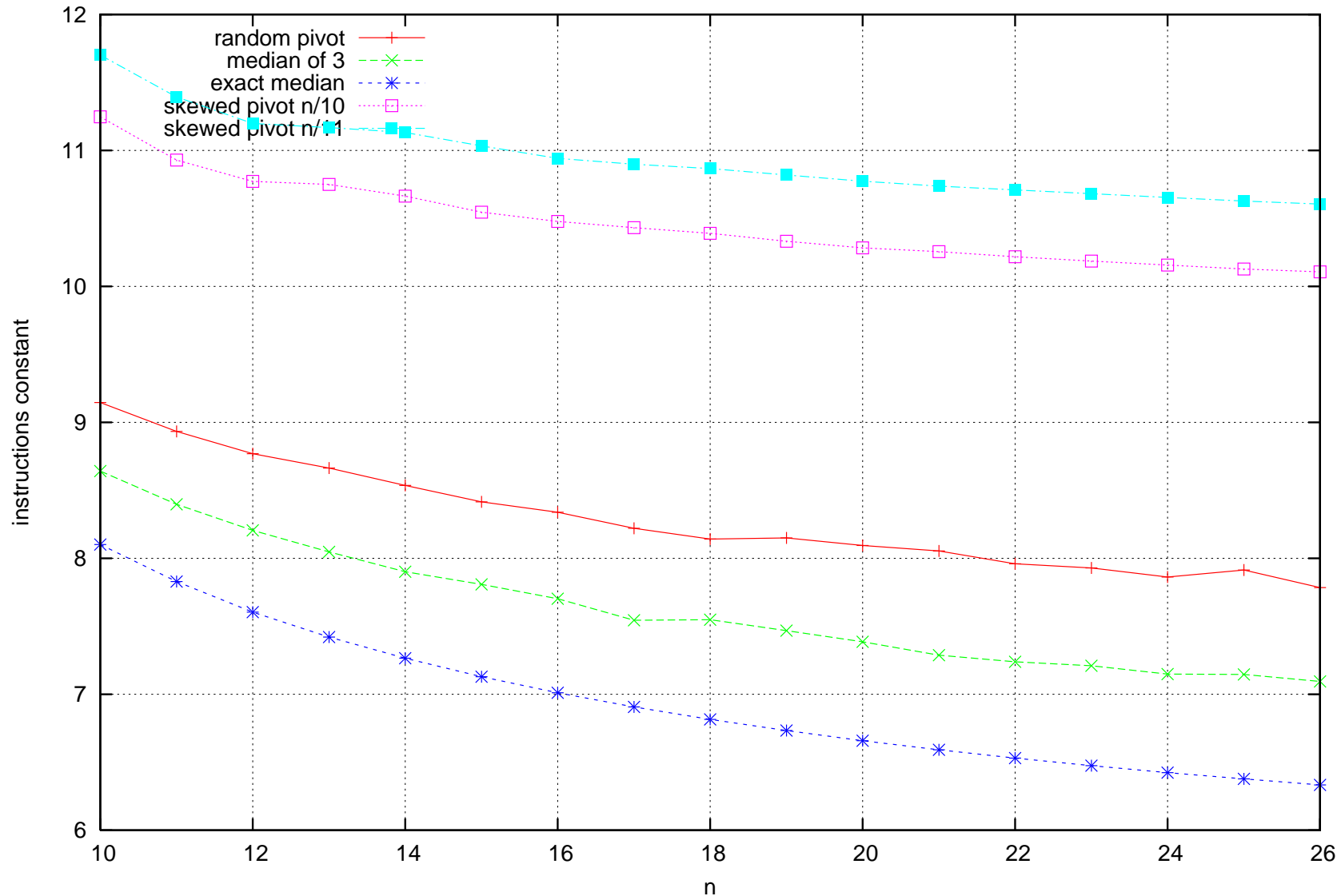
perfect: $\longrightarrow n \log n$ element comparisons
(approximate using large samples)

Practice

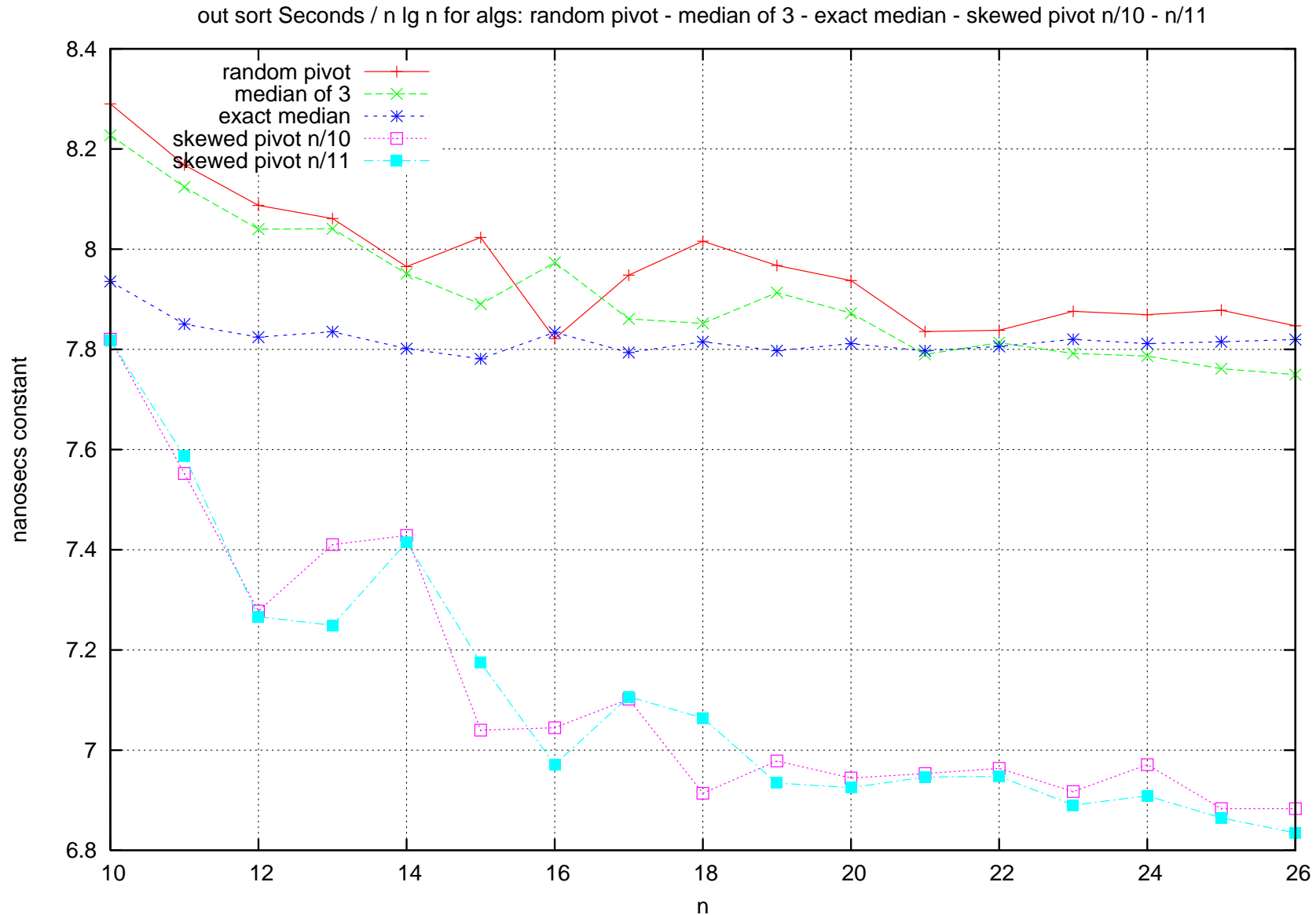
3GHz Pentium 4 Prescott, g++

Picking Pivots Painstakingly — Instructions

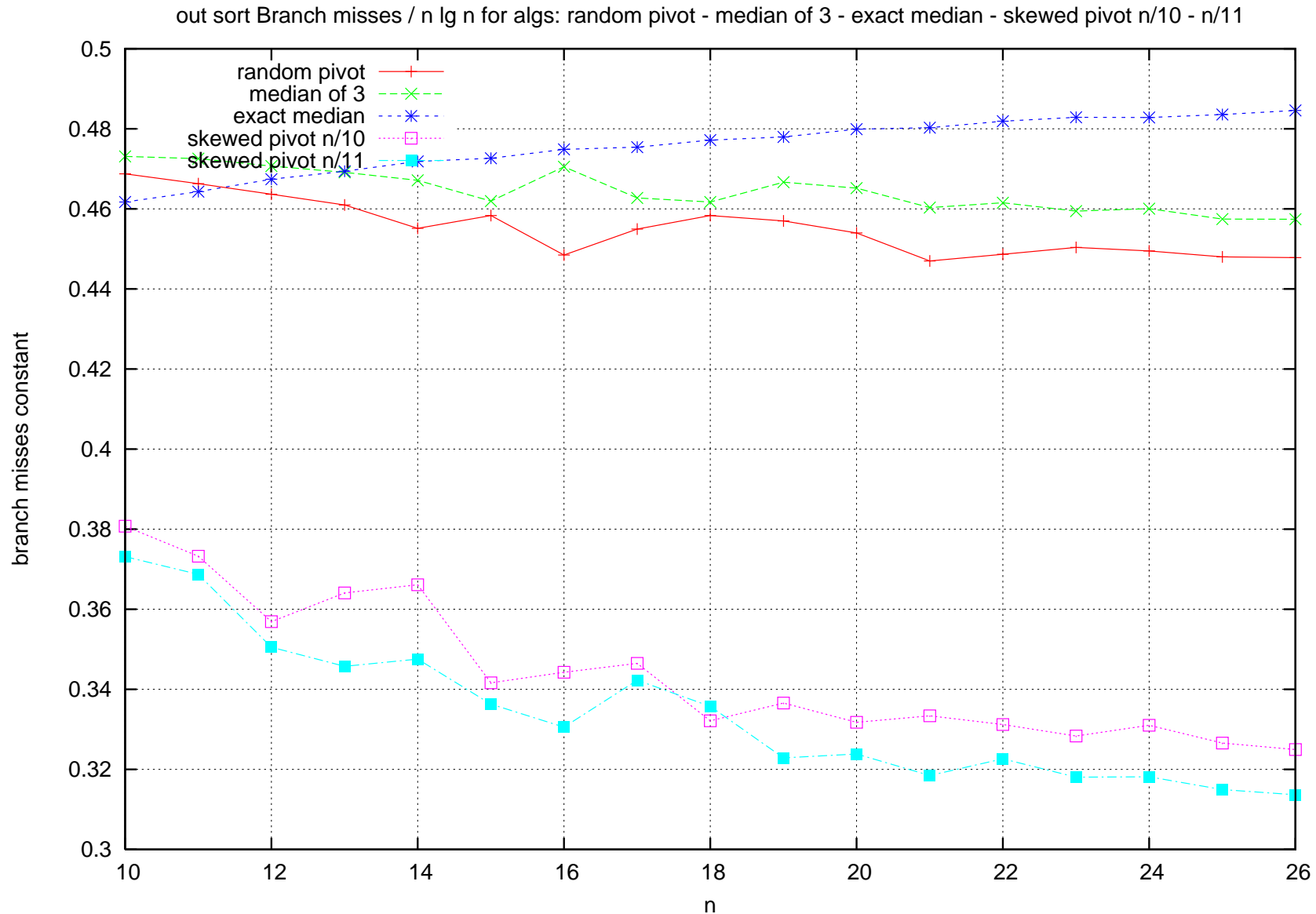
out sort Instructions / $n \lg n$ for algs: random pivot - median of 3 - exact median - skewed pivot $n/10$ - $n/11$



Picking Pivots Painstakingly — Time



Picking Pivots Painstakingly — Branch Misses



Can We Do Better? Previous Work

Integer Keys

- + Can be 2 – 3 times faster than quicksort
- Naive ones are cache inefficient and **slower** than quicksort
- Simple ones are **distribution** dependent.

Cache efficient sorting

k-ary merge sort

[Nyberg et al. 94, Arge et al. 04, Ranade et al. 00, Brodal et al. 04]

- + Faktor $\log k$ less cache faults
- Only $\approx 20\%$ speedup, and only for large inputs

Sample Sort

Function $\text{sampleSort}(e = \langle e_1, \dots, e_n \rangle, k)$

if n/k is “small” **then return** $\text{smallSort}(e)$

let $S = \langle S_1, \dots, S_{ak-1} \rangle$ denote a random **sample** of e

sort S

$\langle s_0, s_1, s_2, \dots, s_{k-1}, s_k \rangle :=$

$\langle -\infty, S_a, S_{2a}, \dots, S_{(k-1)a}, \infty \rangle$

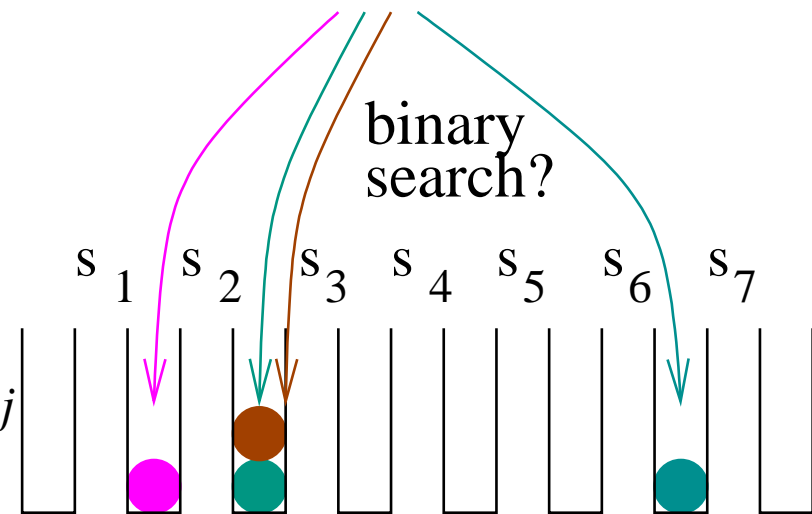
for $i := 1$ **to** n **do**

find $j \in \{1, \dots, k\}$

such that $s_{j-1} < e_i \leq s_j$

place e_i in bucket b_j

return concatenate($\text{sampleSort}(b_1), \dots, \text{sampleSort}(b_k)$) **buckets**



Why Sample Sort?

- traditionally: **parallelizable** on coarse grained machines
- + Cache efficient \approx merge sort
- **Binary search** not much faster than merging
- complicated **memory management**

Super Scalar Sample Sort

- Binary search \longrightarrow **implicit search tree**
- Eliminate all conditional **branches**
- \rightsquigarrow Exploit **instruction parallelism**
- \rightsquigarrow **Cache efficiency** comes to bear
- “steal” memory management from **radix sort**

Classifying Elements

$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \dots \rangle$

for $i := 1$ **to** n **do**

$j := 1$

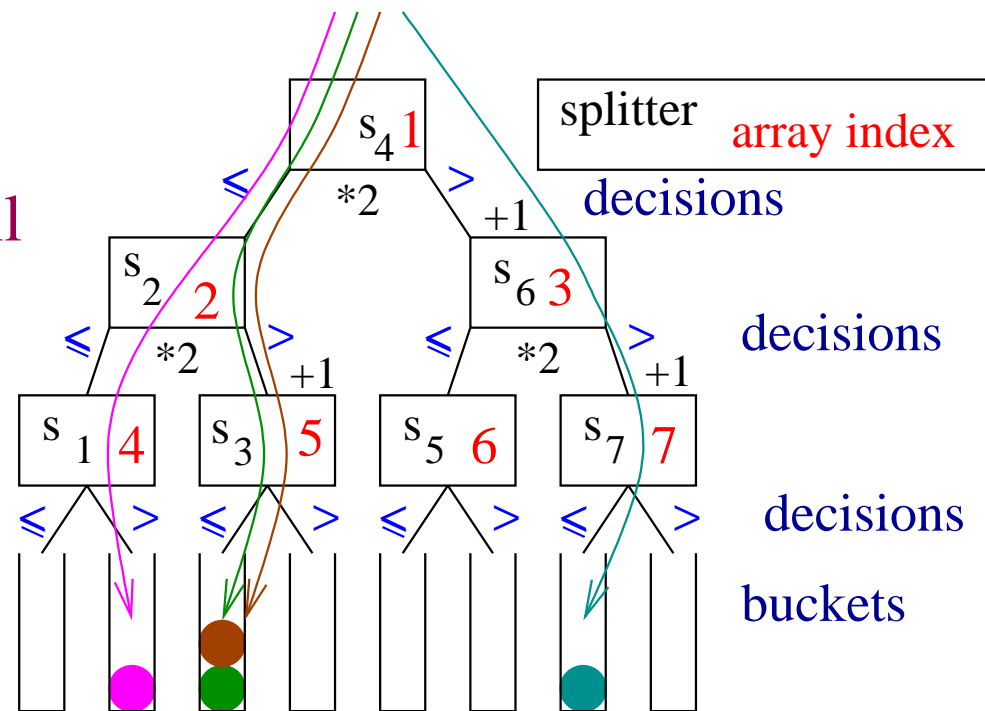
repeat $\log k$ **times** // unroll

$j := 2j + (a_i > t_j)$

$j := j - k + 1$

$|b_j| ++$

 oracle[i] := j // oracle



Now the **compiler** should:

- use **predicated instructions**
- interleave for-loop iterations (**unrolling** \vee **software pipelining**)

```
template <class T>
void findOraclesAndCount(const T* const a,
    const int n, const int k, const T* const s,
    Oracle* const oracle, int* const bucket) {
{ for (int i = 0; i < n; i++)
    int j = 1;
    while (j < k) {
        j = j*2 + (a[i] > s[j]);
    }
    int b = j-k;
    bucket[b]++;
    oracle[i] = b;
}
}
```

Predication

Hardware mechanism that allows instructions to be **conditionally executed**

- Boolean **predicate registers** (1–64) hold condition codes
- predicate registers p are additional inputs of **predicated instructions** I
- At runtime, I is executed if and only if p is true
- + Avoids branch misprediction penalty
- + More flexible instruction scheduling
- Switched off instructions still take time
- Longer opcodes
- Complicated hardware design

Example (IA-64)

Translation of: $\text{if } (r1 > r2) \text{ } r3 := r3 + 4$

With a **conditional branch**:

```
    cmp.gt  p6, p7=r1, r2
(p7) br.cond .label
    add  r3=4, r3
.label:
```

(...)

Via **predication**:

```
    cmp.gt  p6, p7=r1, r2
(p6) add  r3=4, r3
```

Other Current Architectures:

Conditional moves only

Unrolling ($k = 16$)

```
template <class T>
void findOraclesAndCountUnrolled([...]) {
    for (int i = 0; i < n; i++)
        int j = 1;
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        int b = j-k;
        bucket[b]++;
        oracle[i] = b;
    } }
```

More Unrolling $k = 16, n$ even

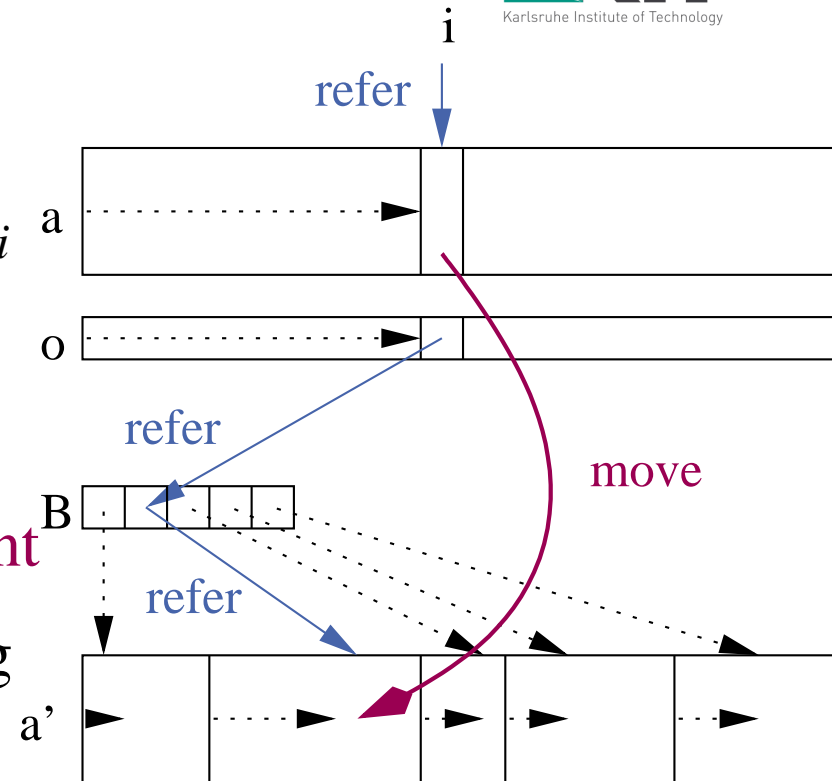
```
template <class T>
void findOraclesAndCountUnrolled2 ([...]) {
    for (int i = n & 1; i < n; i+=2) {\
        int j0 = 1;           int j1 = 1;
        T ai0 = a[i];        T ai1 = a[i+1];
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
        int b0 = j0-k;       int b1 = j1-k;
        bucket[b0]++;       bucket[b1]++;
        oracle[i] = b0;     oracle[i+1] = b1;
    } }
```


Distributing Elements

for $i := 1$ to n do $a'_{B[\text{oracle}[i]]++} := a_i$

Why Oracles?

- simplifies **memory management**
- no **overflow tests** or re-copying
- simplifies software **pipelining**
- separates **computation** and **memory access** aspects
- small** (n bytes)
- sequential, predictable** memory access
- can be **hidden** using prefetching / write buffering



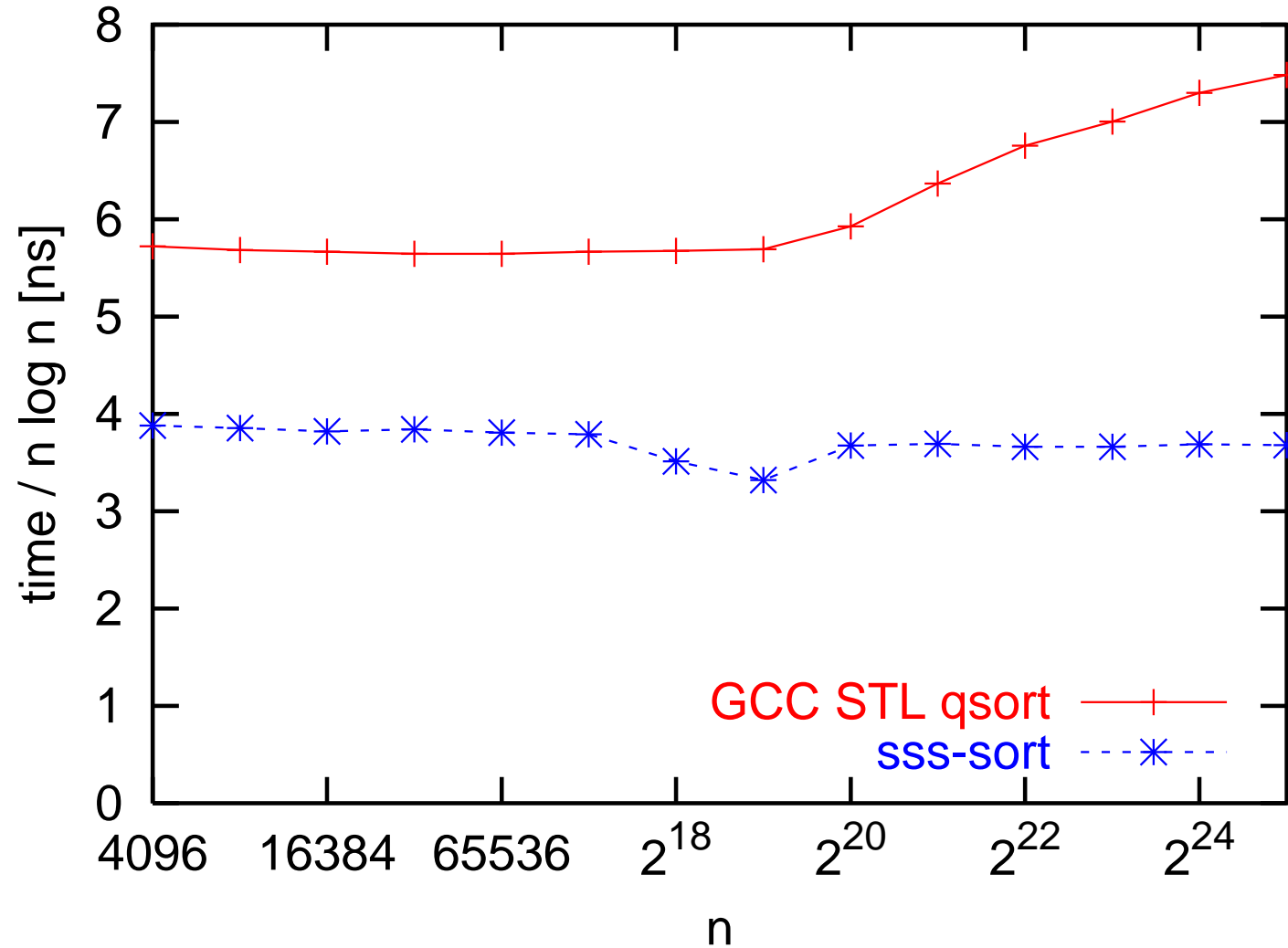
Distributing Elements

```
template <class T> void distribute(  
    const T* const a0, T* const a1,  
    const int n, const int k,  
    const Oracle* const oracle, int* const bucket)  
{ for (int i = 0, sum = 0; i <= k; i++) {  
    int t = bucket[i]; bucket[i] = sum; sum += t;  
}  
for (int i = 0; i < n; i++) {  
    a1[bucket[oracle[i]]++] = a0[i];  
}  
}
```

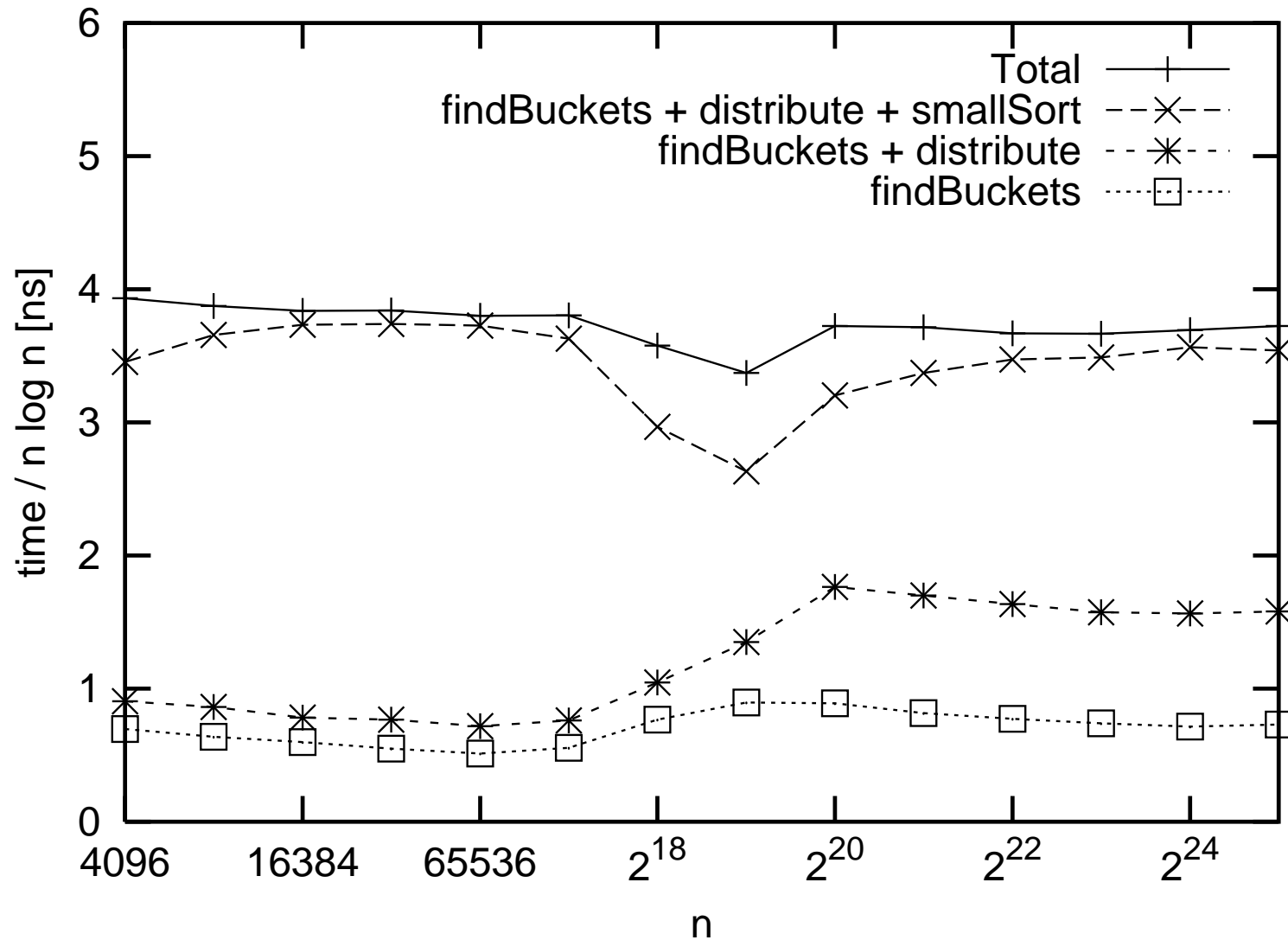
Experiments: 1.4 GHz Itanium 2

- `restrict` keyword from ANSI/ISO C99 to indicate nonaliasing
- Intel's C++ compiler v8.0 uses **predicated instructions** automatically
- Profiling** gives **9%** speedup
- $k = 256$ splitters
- Use `std::sort` from **g++** ($n \leq 1000$)!
- insertion sort for $n \leq 100$
- Random 32 bit integers in $[0, 10^9]$

Comparison with Quicksort



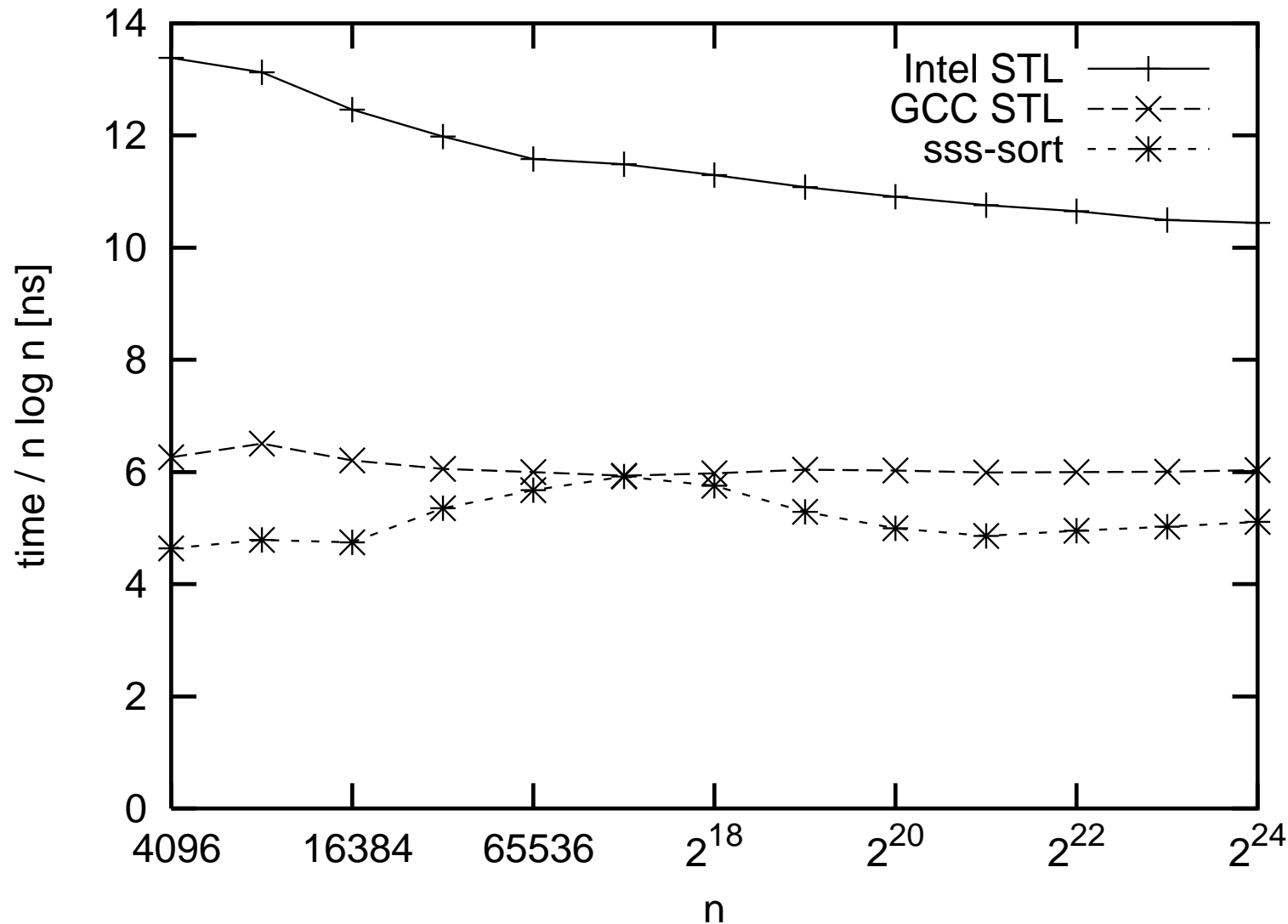
Breakdown of Execution Time



A More Detailed View

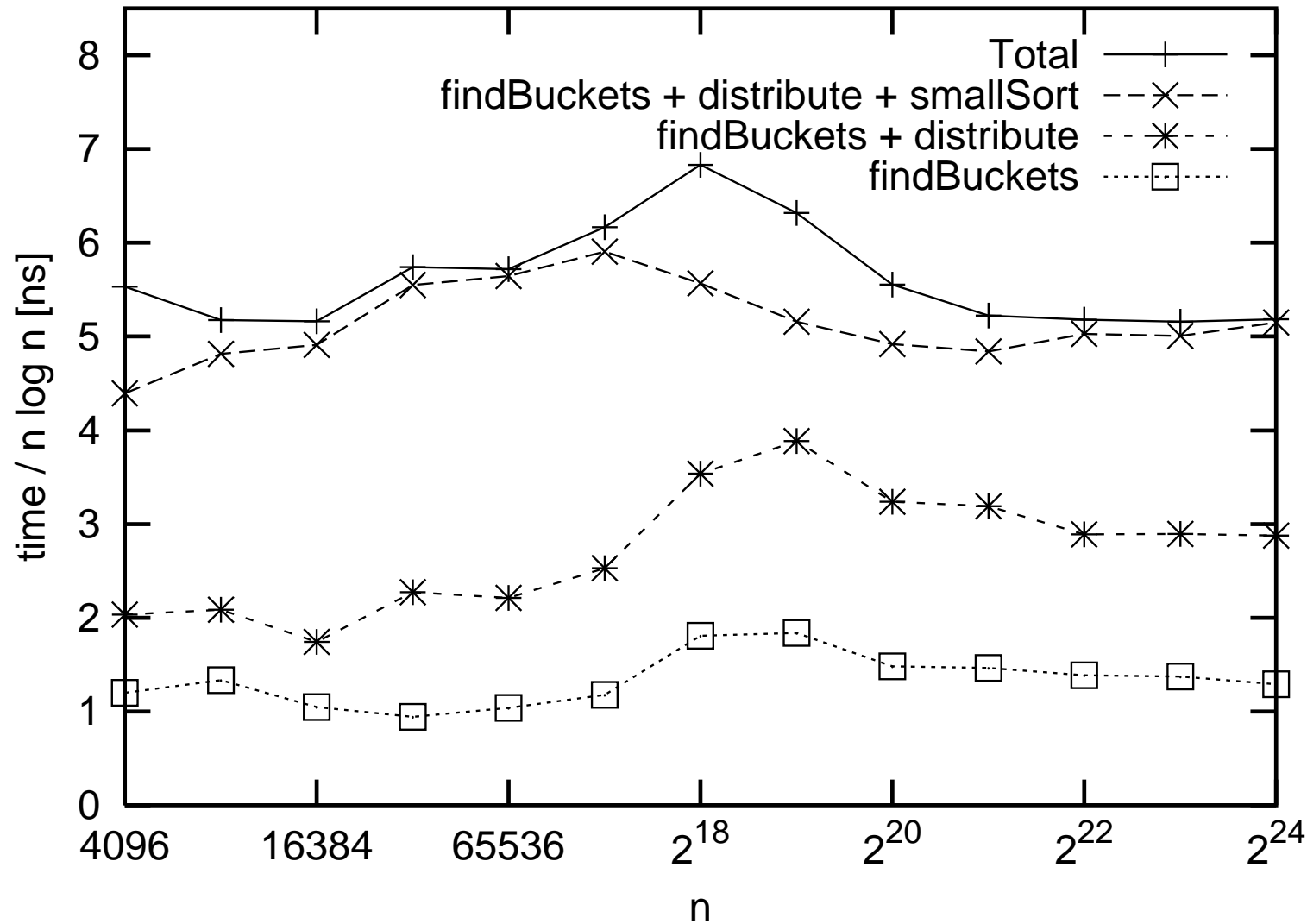
	instr.	cycles	dynamic IPC small n	dynamic IPC $n = 2^{25}$
findBuckets, 1 × outer loop	63	11	5.4	4.5
distribute, one element	14	4	3.5	0.8

Comparison with Quicksort Pentium 4



Problems: few registers, one condition code only, compiler needs “help”

Breakdown of Execution Time Pentium 4



Analysis

	mem. acc.	branches	data dep.	I/Os	registers	instructions
<i>k</i> -way distribution:						
sss-sort	$n \log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\geq 3.5n/B$	$3 \times \text{unroll}$	$\mathcal{O}(\log k)$
IS^4_o	$n \log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$4n/B$	$3 \times \text{unroll}$	$\mathcal{O}(\log k)$
quicksort $\log k$ lvls.	$2n \log k$	$n \log k$	$\mathcal{O}(n \log k)$	$2 \frac{n}{B} \log k$	4	$\mathcal{O}(1)$
<i>k</i> -way merging:						
memory	$n \log k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	7	$\mathcal{O}(\log k)$
register	$2n$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	k	$\mathcal{O}(k)$
funnel $k'^{\log_{k'} k}$	$2n \log_{k'} k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	$2k' + 2$	$\mathcal{O}(k')$

Conclusions

- sss-sort up to **twice** as fast as quicksort on Itanium
- comparisons \neq conditional branches
- algorithm analysis is not just instructions and caches

More results: **GPU-Sample-Sort** is (was) best comparison based sorting algorithm on graphics hardware

[Leischner/Osipov/Sanders 2009]

Parallel String Sample-Sorting is best string sorting algorithm

[Bingmann/Sanders 2013]

AMS Sort scales to 2^{15} PEs [AxtmannBSS SPAA 2015]

Criticism I

Why only random keys?

Answer I

Sample sort hardly depends on input distribution

Criticism I'

What if there are many equal keys?

They all end up in the same bucket

Answer I'

Its not a bug its a feature:

$s_i = s_{i+1} = \dots = s_j$ indicates a frequent key!

Set $s_i := \max \{x \in \text{Key} : x < s_i\}$,

(optional: drop s_{i+2}, \dots, s_j)

Now bucket $i + 1$ need not be sorted!

Exercise: Explain how to support equality buckets using a single additional comparison per element.

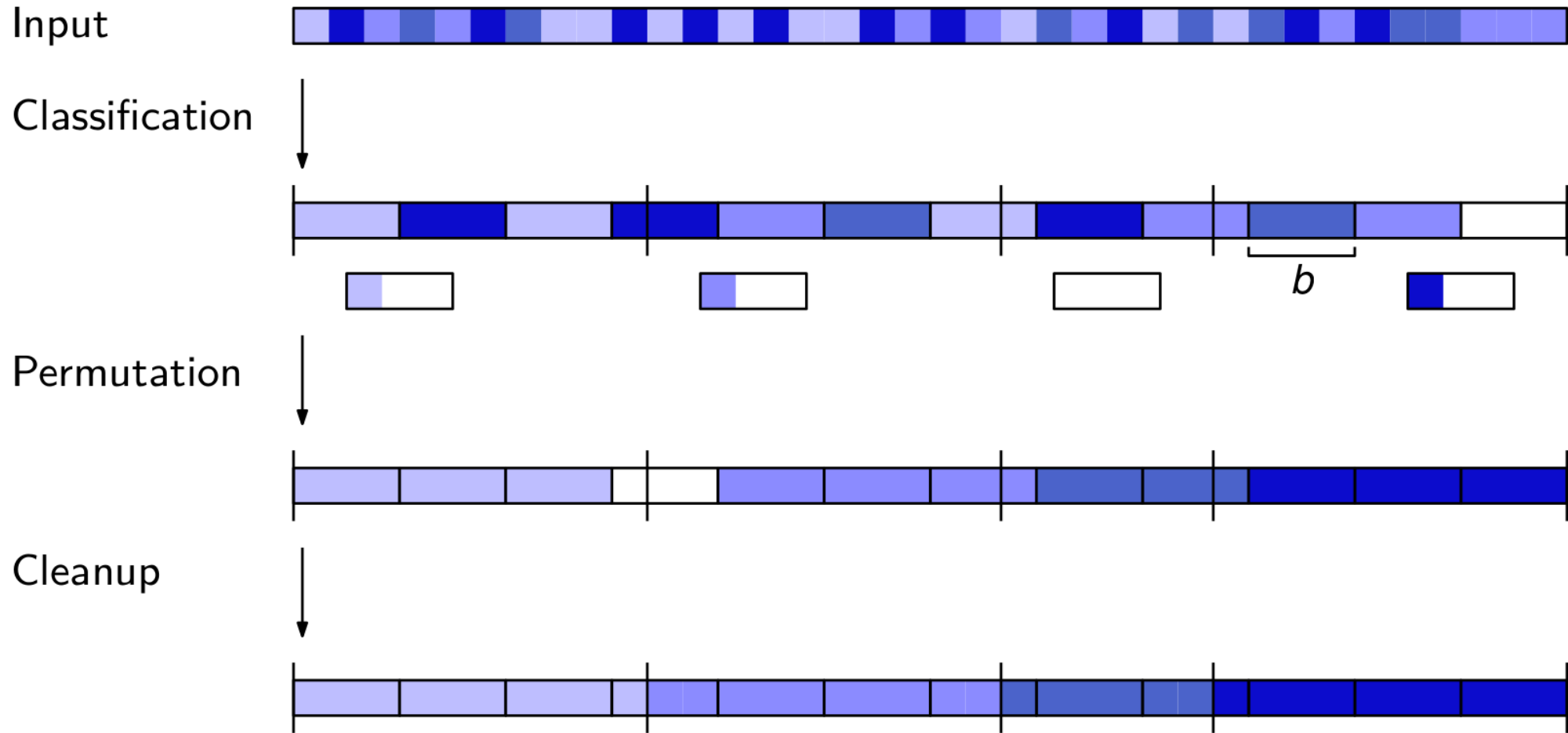
Criticism II

Quicksort is inplace

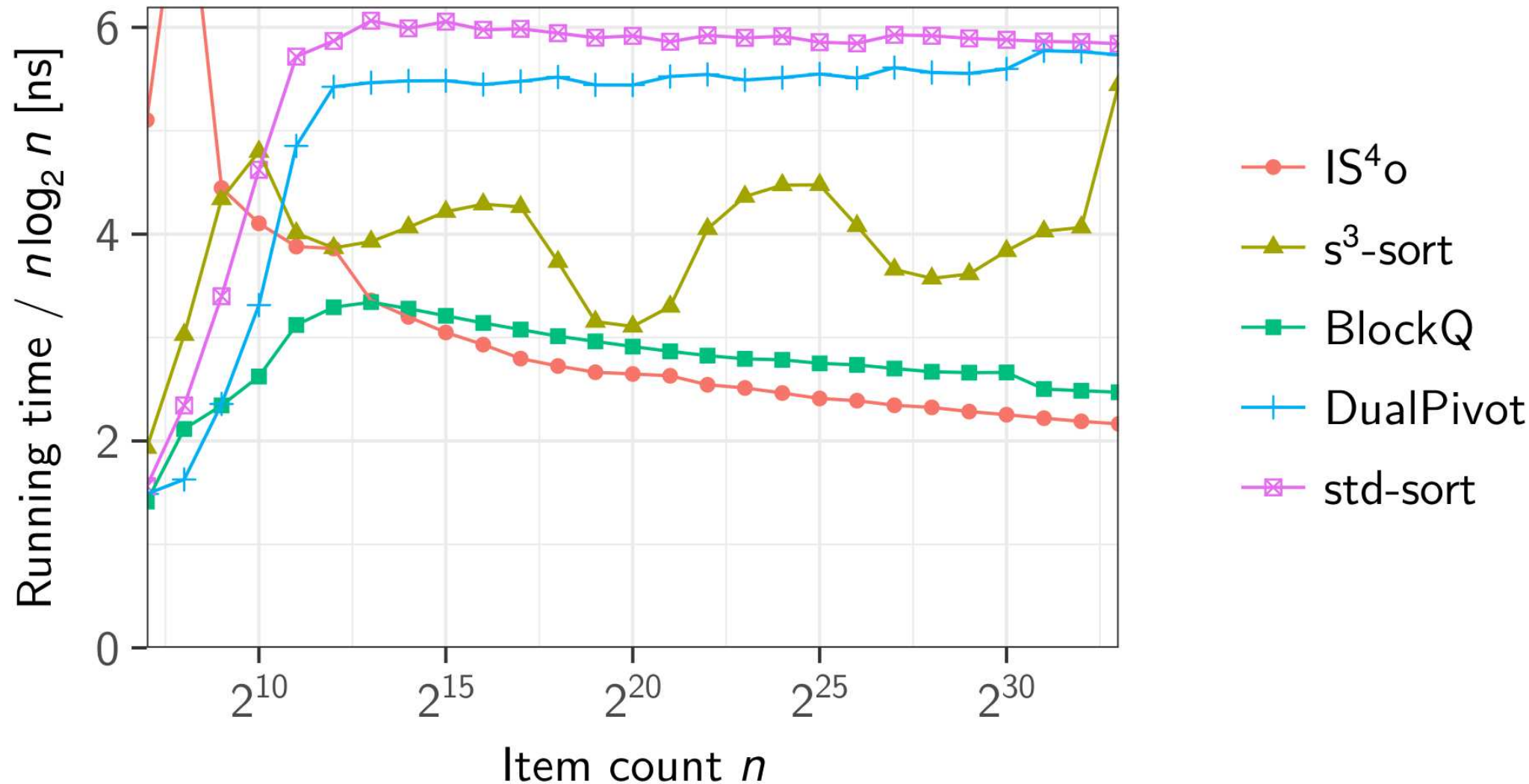
Answer II

inplace super scalar sample sort [\[18\]](#).

Inplace Super Scalar Sample Sort



Inplace Super Scalar Sample Sort



Why is inplace **faster**

- memory management
- allocation misses
- associativity misses
- oracles

versus writing all the data one additional times

Future Work

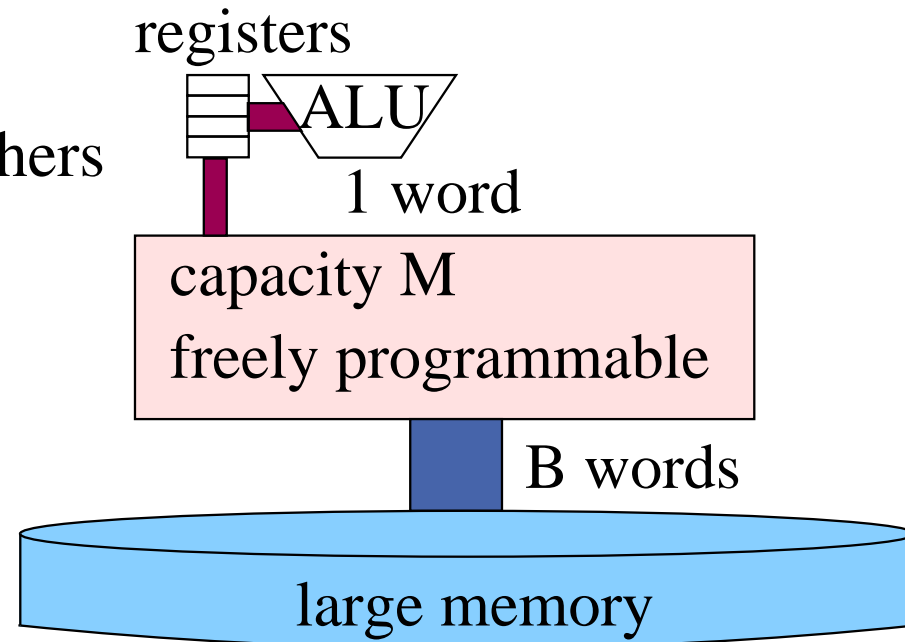
- better small case sorter** for arbitrary keys/comparators
(for small numbers, SIMD, sorting networks etc. give good base case sorters)
- SIMD**-instructions for distribution
- multilevel** cache-aware or cache-oblivious generalization
- thorough testing** \rightsquigarrow **verification?** or back to simplicity?
- Save a pass for IS^4o using virtual memory tricks (**remap rather than move** blocks)

Externes Sortieren

n : Eingabegröße

M : Größe des schnellen Speichers

B : Blockgröße



Procedure externalMerge(*a*, *b*, *c* :File of Element)

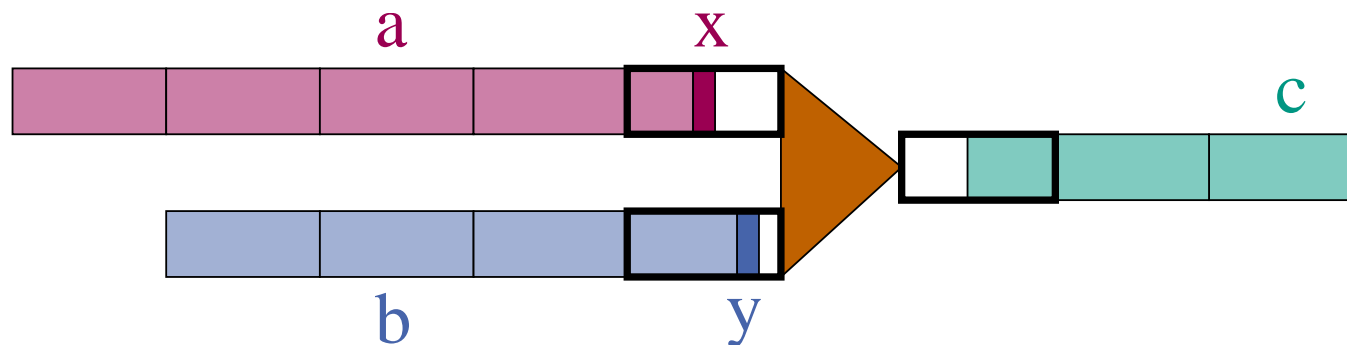
x := *a*.readElement // Assume emptyFile.readElement = ∞

y := *b*.readElement

for *j* := 1 **to** |*a*| + |*b*| **do**

if *x* ≤ *y* **then** *c*.writeElement(*x*); *x* := *a*.readElement

else *c*.writeElement(*y*); *y* := *b*.readElement



Externes (binäres) Mischen – I/O-Analyse

Datei a lesen: $\lceil |a|/B \rceil \leq |a|/B + 1$.

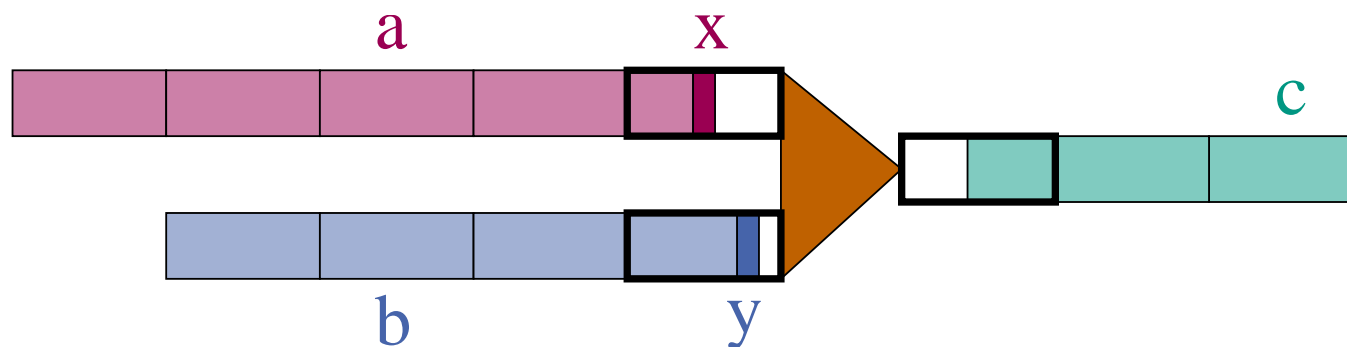
Datei b lesen: $\lceil |b|/B \rceil \leq |b|/B + 1$.

Datei c schreiben: $\lceil (|a| + |b|)/B \rceil \leq (|a| + |b|)/B + 1$.

Insgesamt:

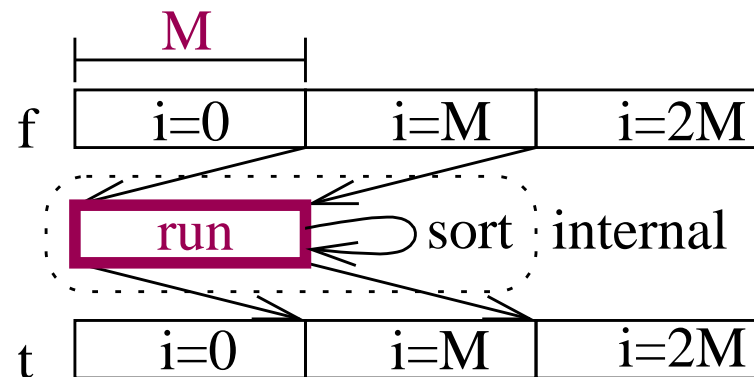
$$\leq 3 + 2 \frac{|a| + |b|}{B} \approx 2 \frac{|a| + |b|}{B}$$

Bedingung: Wir brauchen 3 Pufferblöcke, d.h., $M > 3B$.



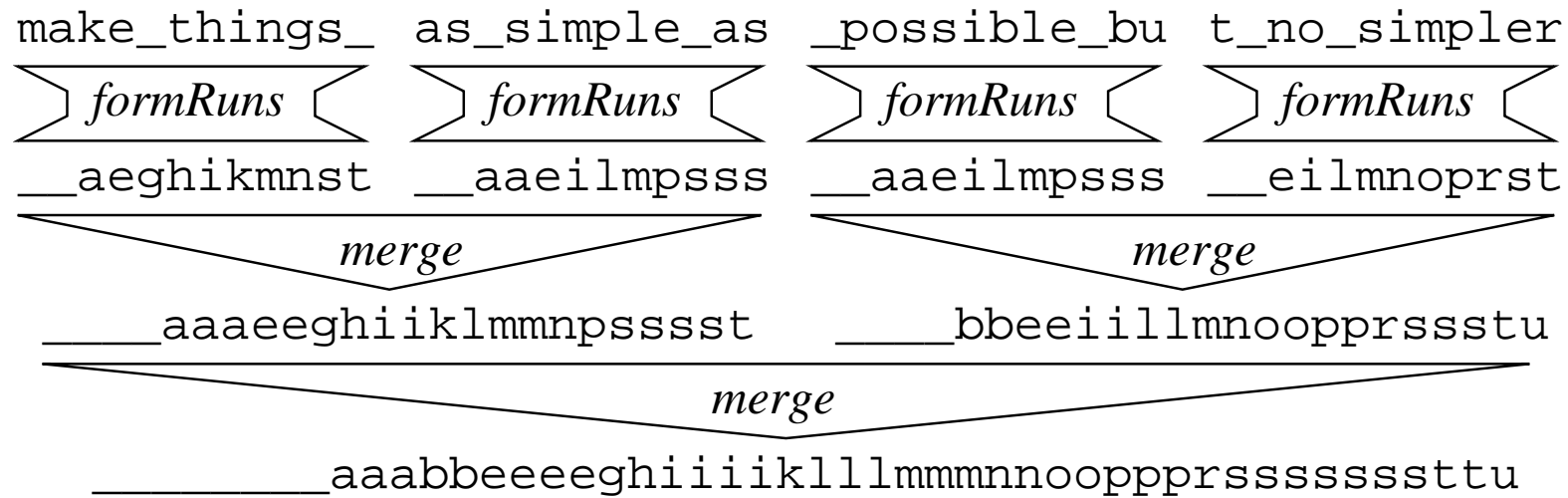
Run Formation

Sortiere Eingabeportionen der Größe M



$$\text{I/Os: } \approx 2 \frac{n}{B}$$

Sortieren durch Externes Binäres Mischen



Procedure externalBinaryMergeSort // I/Os: \approx

 run formation // $2n/B$

while more than one run left **do** // $\lceil \log \frac{n}{M} \rceil \times$

 merge pairs of runs // $2n/B$

 output remaining run // $\Sigma : 2 \frac{n}{B} \left(1 + \lceil \log \frac{n}{M} \rceil \right)$

Zahlenbeispiel: PC 2007

$$n = 2^{38} \text{ Byte}$$

$$M = 2^{31} \text{ Byte}$$

$$B = 2^{20} \text{ Byte}$$

I/O braucht 2^{-6} s

$$\text{Zeit: } 2 \frac{n}{B} \left(1 + \left\lceil \log \frac{n}{M} \right\rceil \right) = 2 \cdot 2^{18} \cdot (1 + 7) \cdot 2^{-6} \text{ s} = 2^{16} \text{ s} \approx 18 \text{ h}$$

Idee: 8 Durchläufe \rightsquigarrow 2 Durchläufe

Zahlenbeispiel: PC 2007 → 2019

$$n = 2^{38 \rightarrow 41} \text{ Byte}$$

$$M = 2^{31 \rightarrow 34} \text{ Byte}$$

$$B = 2^{20 \rightarrow 22} \text{ Byte}$$

I/O braucht 2^{-5} s

$$\text{Zeit: } 2 \frac{n}{B} \left(1 + \left\lceil \log \frac{n}{M} \right\rceil \right) = 2 \cdot 2^{18} \cdot (1 + 7) \cdot 2^{-5} \text{ s} = 2^{16} \text{ s} \approx 73 \text{ h}$$

Mehrwegemischen

Procedure multiwayMerge(a_1, \dots, a_k, c :File of Element)

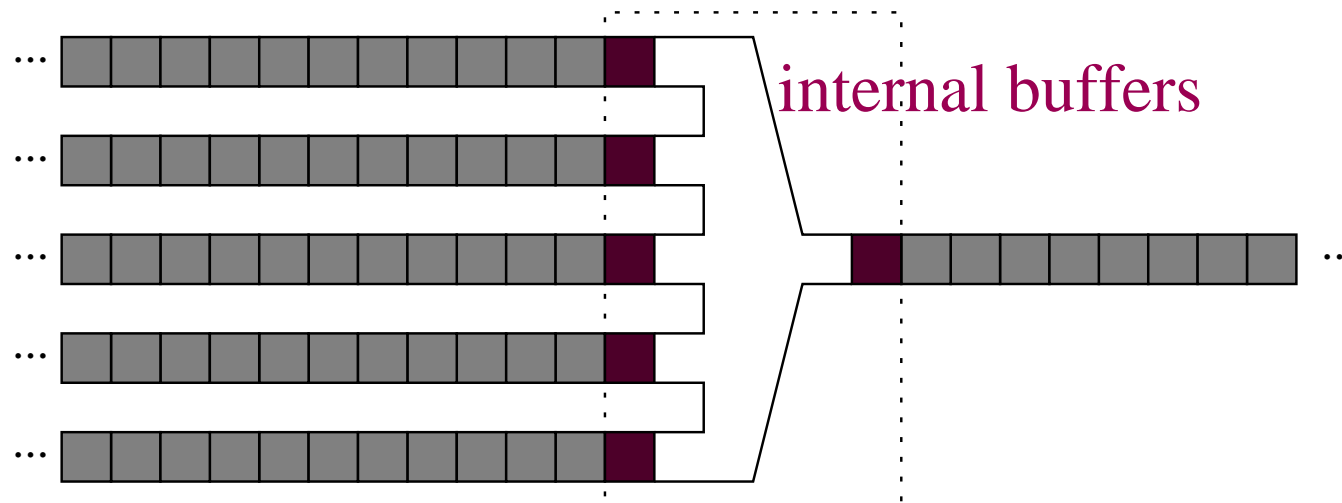
for $i := 1$ **to** k **do** $x_i := a_i$.readElement

for $j := 1$ **to** $\sum_{i=1}^k |a_i|$ **do**

 find $i \in 1..k$ that minimizes x_i // no I/Os!, $\mathcal{O}(\log k)$ time

c .writeElement(x_i)

$x_i := a_i$.readElement



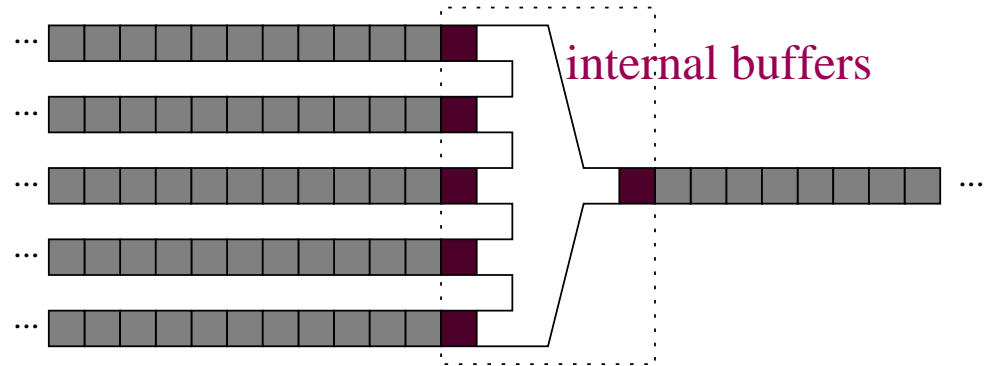
Mehrwegemischen – Analyse

I/Os: Datei a_i lesen: $\approx |a_i|/B$.

Datei c schreiben: $\approx \sum_{i=1}^k |a_i|/B$

Insgesamt:

$$\leq \approx 2 \frac{\sum_{i=1}^k |a_i|}{B}$$



Bedingung: Wir brauchen k Pufferblöcke, d.h., $k < M/B$.

Interne Arbeit: (benutze Prioritätsliste !)

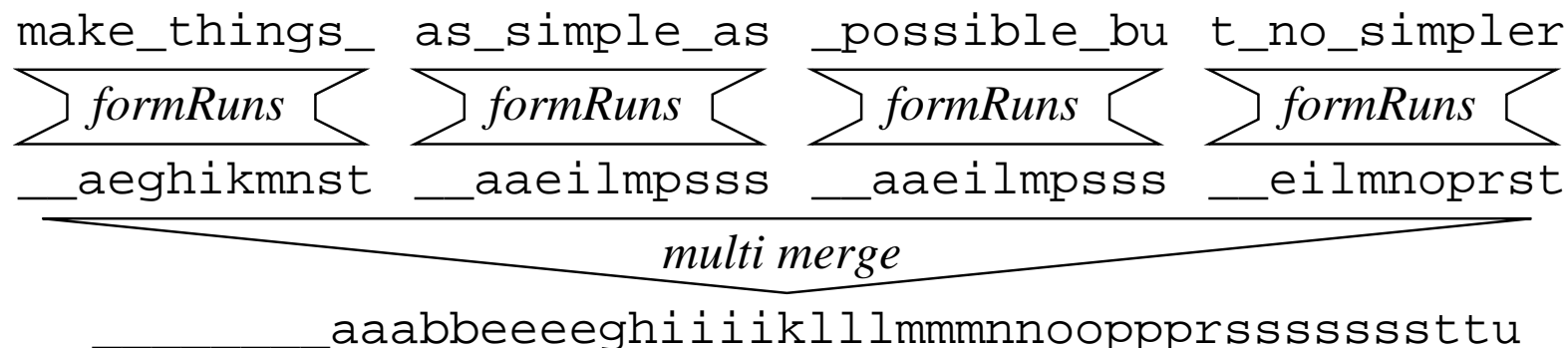
$$\mathcal{O} \left(\log k \sum_{i=1}^k |a_i| \right)$$

Sortieren durch Mehrwege-Mischen

- Sortiere $\lceil n/M \rceil$ runs mit je M Elementen $2n/B$ I/Os
- Mische jeweils M/B runs $2n/B$ I/Os
- bis nur noch ein run übrig ist $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ Mischphasen

Insgesamt

$$\text{sort}(n) := \frac{2n}{B} \left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right) \text{ I/Os}$$



Sortieren durch Mehrwege-Mischen

Interne Arbeit:

$$\mathcal{O} \left(\underbrace{n \log M}_{\text{run formation}} + \underbrace{n \log \frac{M}{B}}_{\text{PQ access per phase}} \overbrace{\left\lceil \log_{M/B} \frac{n}{M} \right\rceil}^{\text{phases}} \right) = \mathcal{O}(n \log n)$$

Mehr als eine Mischphase?:

Nicht für Hierarchie Hauptspeicher, Festplatte.

$$\text{Grund } \frac{\overbrace{M}^{>4000}}{B} > \frac{\overbrace{\text{RAM Euro/bit}}^{<1000}}{\text{Platte Euro/bit}}$$

$$2019: \frac{16GB}{4MB} = 4096 > \frac{88/16GB}{99/4TB} \approx 207$$

Mehr zu externem Sortieren

Untere Schranke $\approx \frac{2^{(?)n}}{B} \left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$ I/Os

[Aggarwal Vitter 1988]

Oberer Schranke $\approx \frac{2n}{DB} \left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$ I/Os (erwartet)

für D parallele Platten

[Hutchinson Sanders Vitter 2005, Dementiev Sanders 2003]

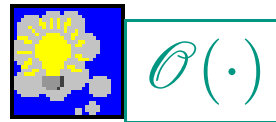
Offene Frage: deterministisch?

Sorting with Parallel Disks

I/O Step := Access to a single physical block per disk

Theory: Balance Sort [Nodine Vitter 93].

Deterministic, complex
asymptotically optimal



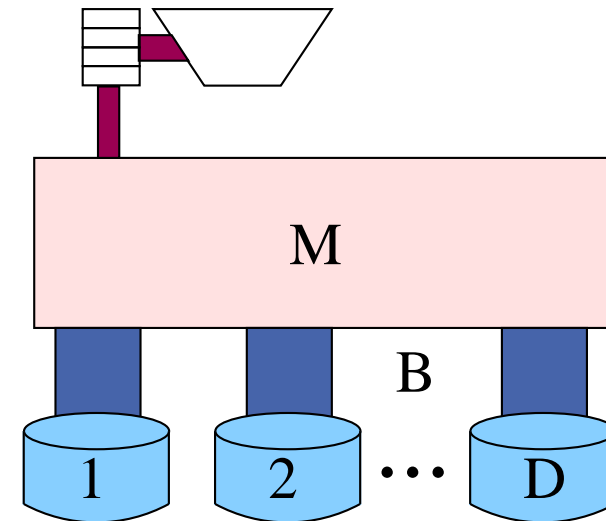
Multiway merging

“Usually” factor 10? less I/Os.

Not asymptotically optimal.

42%

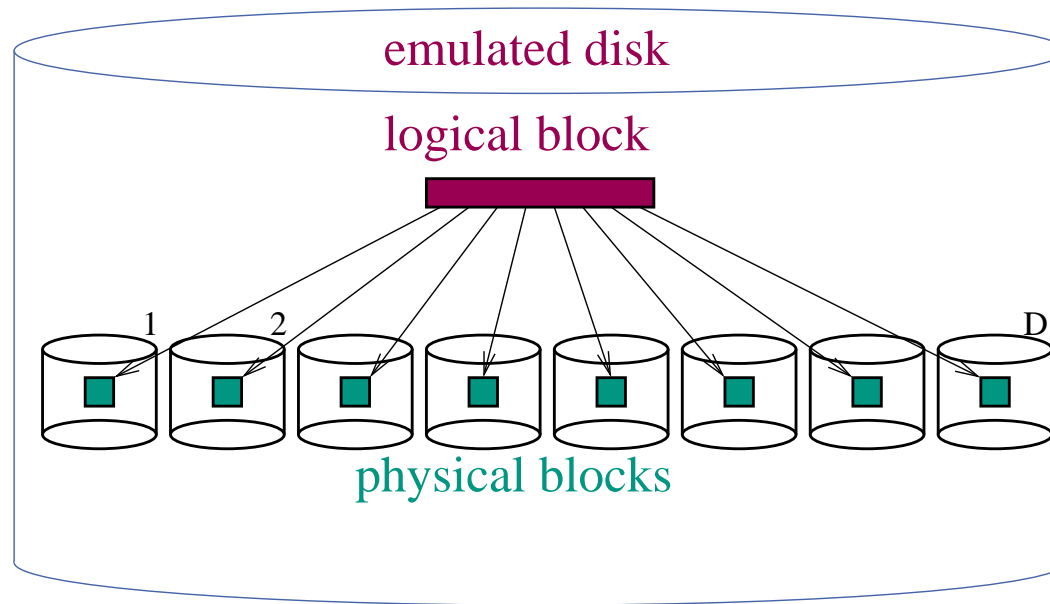
Basic Approach: Improve Multiway Merging



independent disks

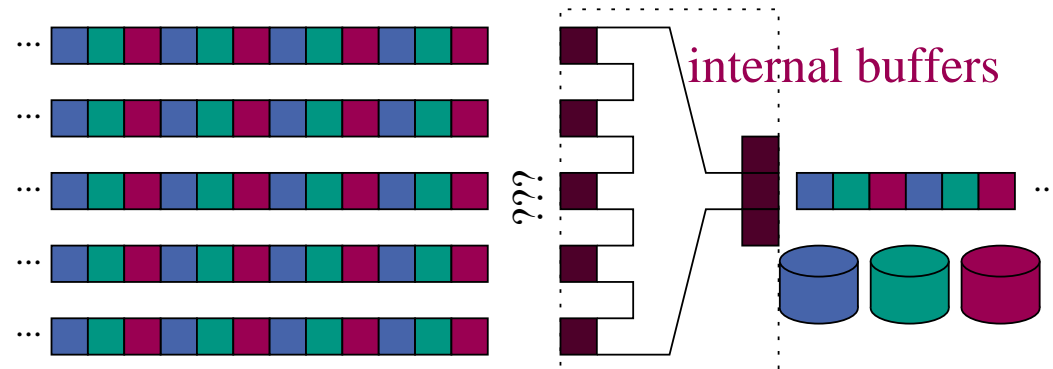
[Vitter Shriver 94]

Striping



That takes care of **run formation**
and writing the **output**

But what about **merging**?



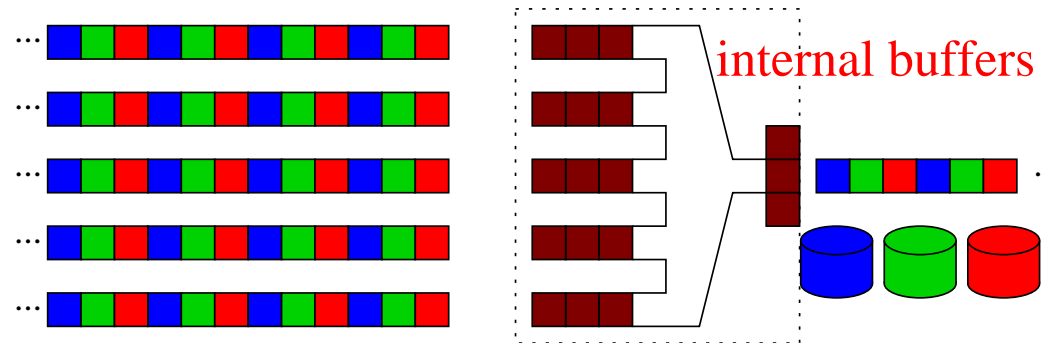
Naive Striping

Run single disk merge-sort on striped logical disk:

$$\frac{2n}{DB} \left(1 + \left\lceil \log_{M/DB} \frac{n}{M} \right\rceil \right) \text{ I/Os}$$

Theory: $\Theta(\log M/B)$ worse when $D \approx M/B$

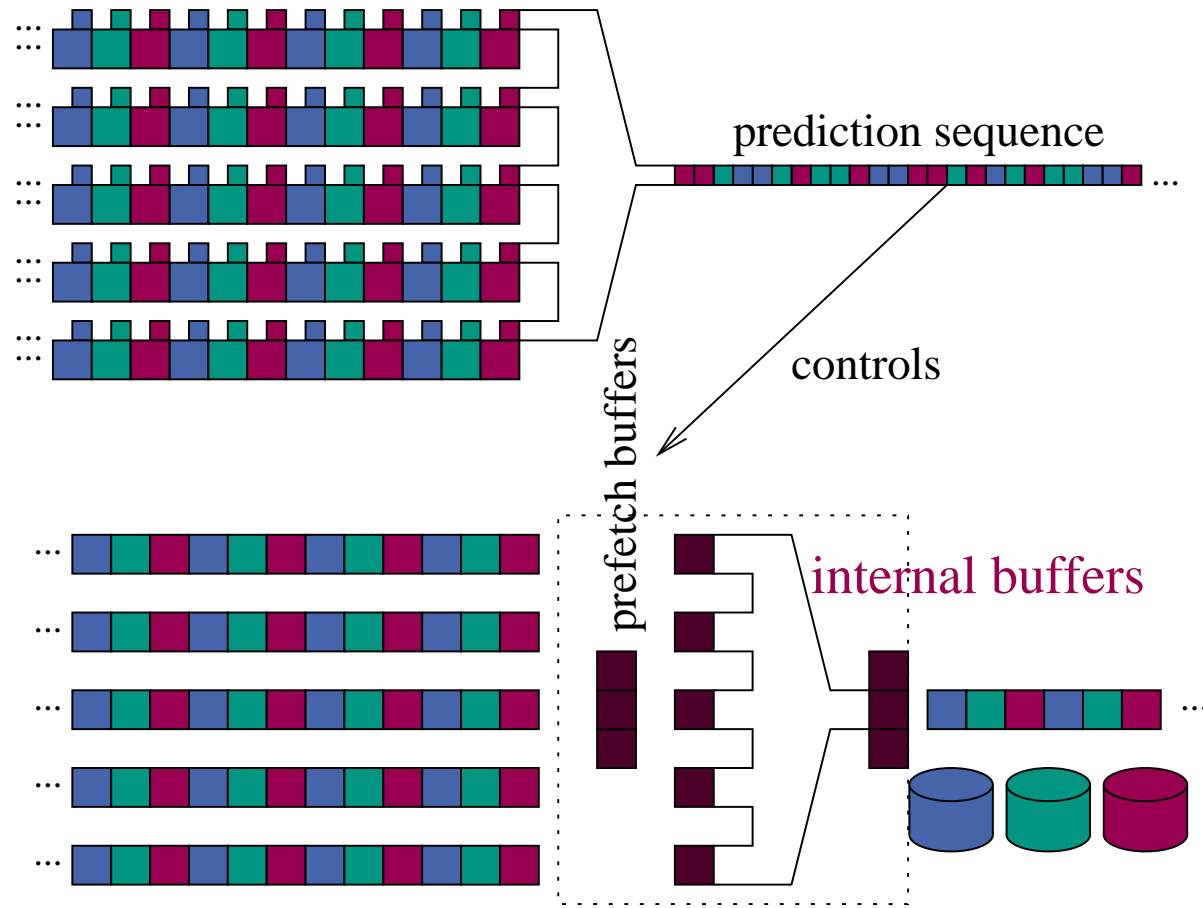
Practice: 2 \rightarrow 3 passes in some cases



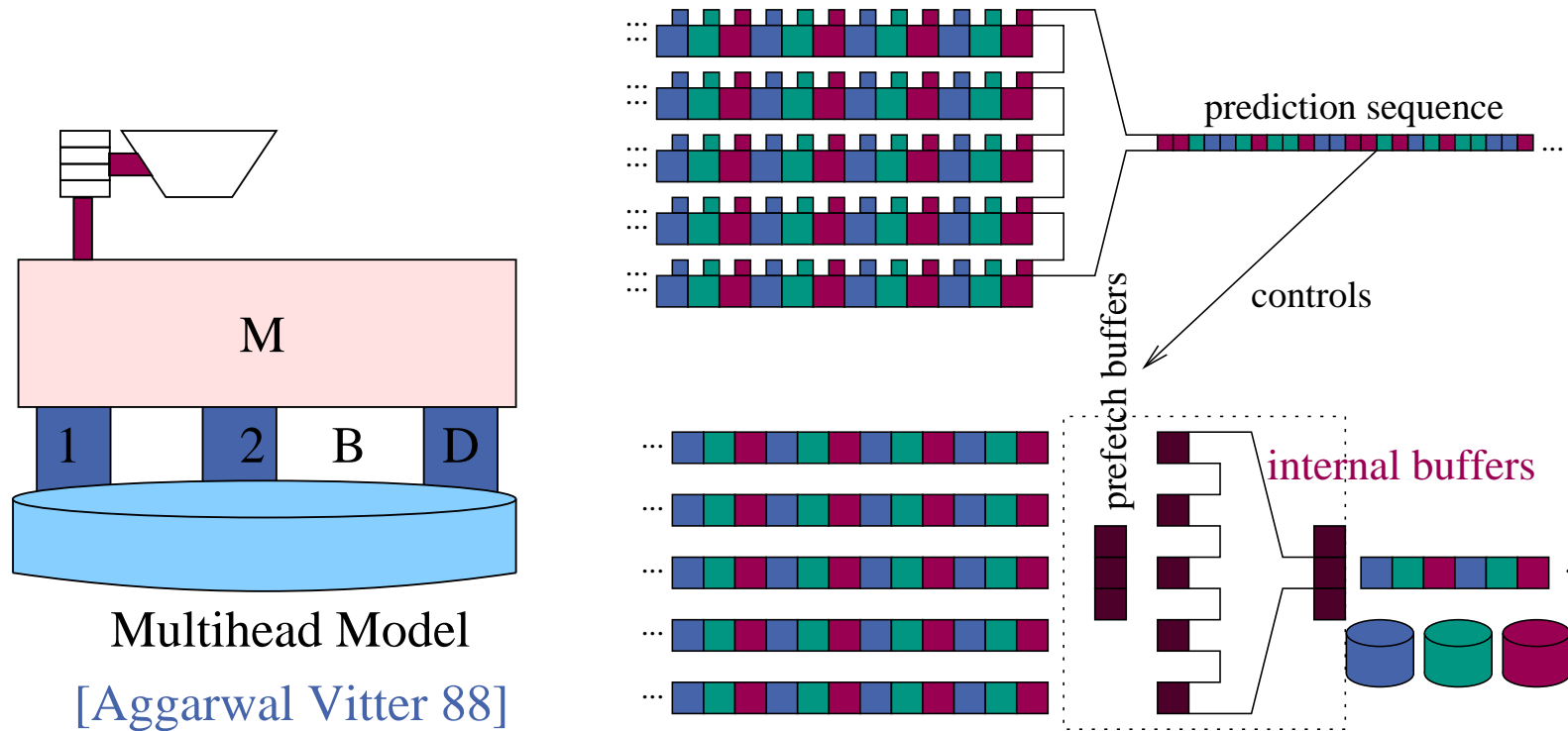
Prediction

[Folklore, Knuth]
Smallest Element
of each block
triggers fetch.

Prefetch buffers
allow parallel access
of next blocks



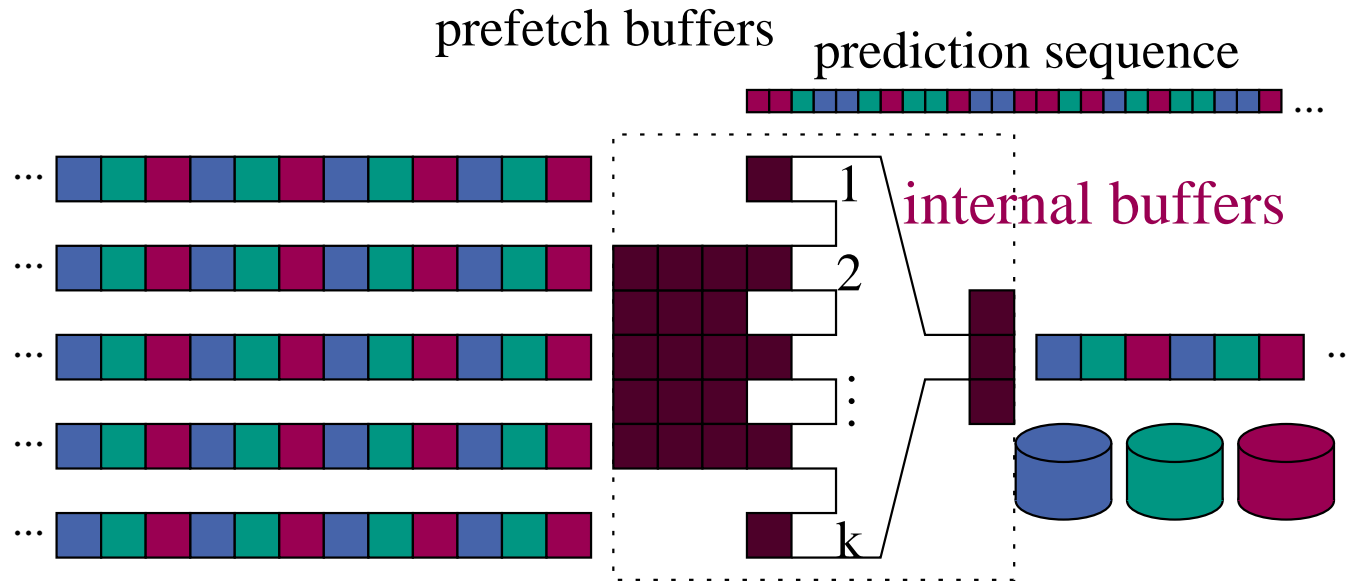
Warmup: Multihead Model



D prefetch buffers yield an optimal algorithm

$$\text{sort}(n) := \frac{2n}{DB} \left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right) \text{ I/Os}$$

Bigger Prefetch Buffer



$Dk \rightsquigarrow$ good **deterministic** performance

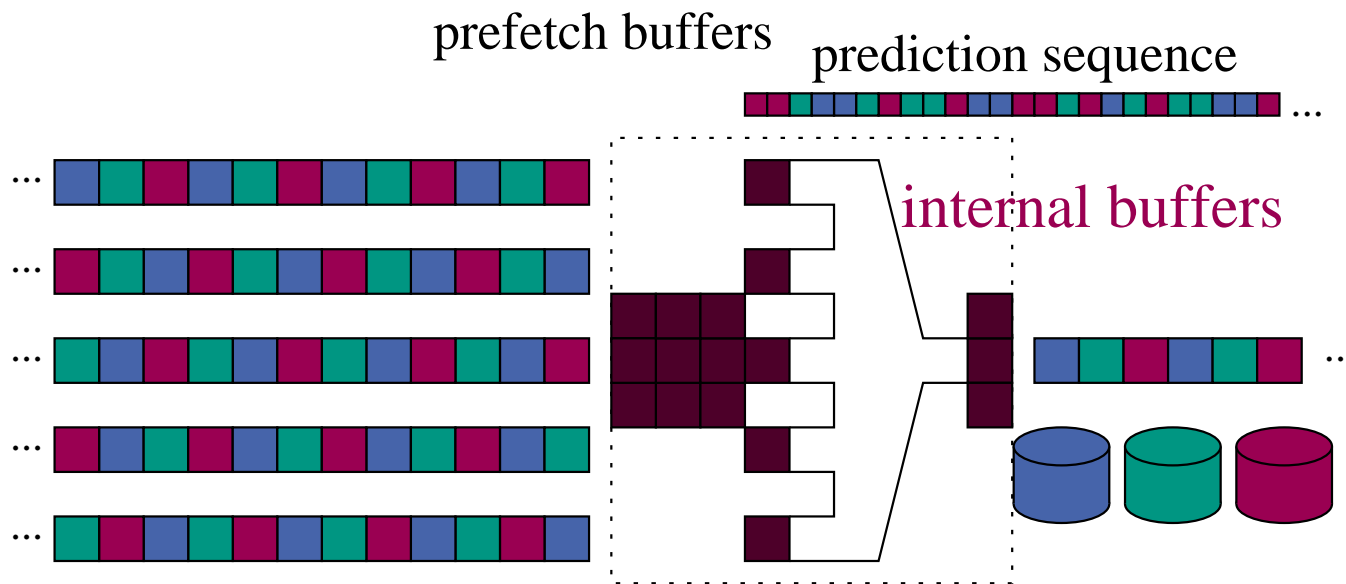
$\mathcal{O}(D)$ would yield an optimal algorithm.

Possible?

Randomized Cycling

[Vitter Hutchinson 01]

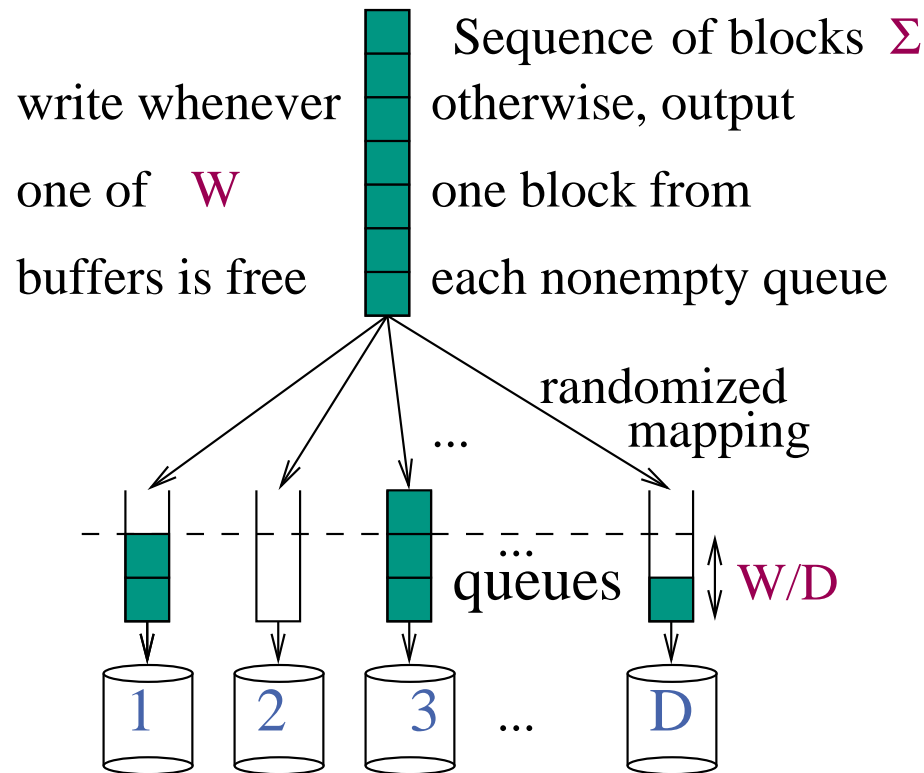
Block i of stripe j goes to disk $\pi_j(i)$ for a **rand. permutation** π_j



Good for **naive prefetching** and $\Omega(D \log D)$ buffers

Buffered Writing

[S-Egner-Korst SODA00, Hutchinson-S-Vitter ESA 01]



Theorem:
Buffered Writing
is **optimal**

...

But

how good is optimal?

Theorem: Rand. cycling achieves **efficiency** $1 - \mathcal{O}(D/W)$.

Analysis: **negative association** of random variables,
application of **queueing theory** to a “throttled” Alg.

Optimal Offline Prefetching

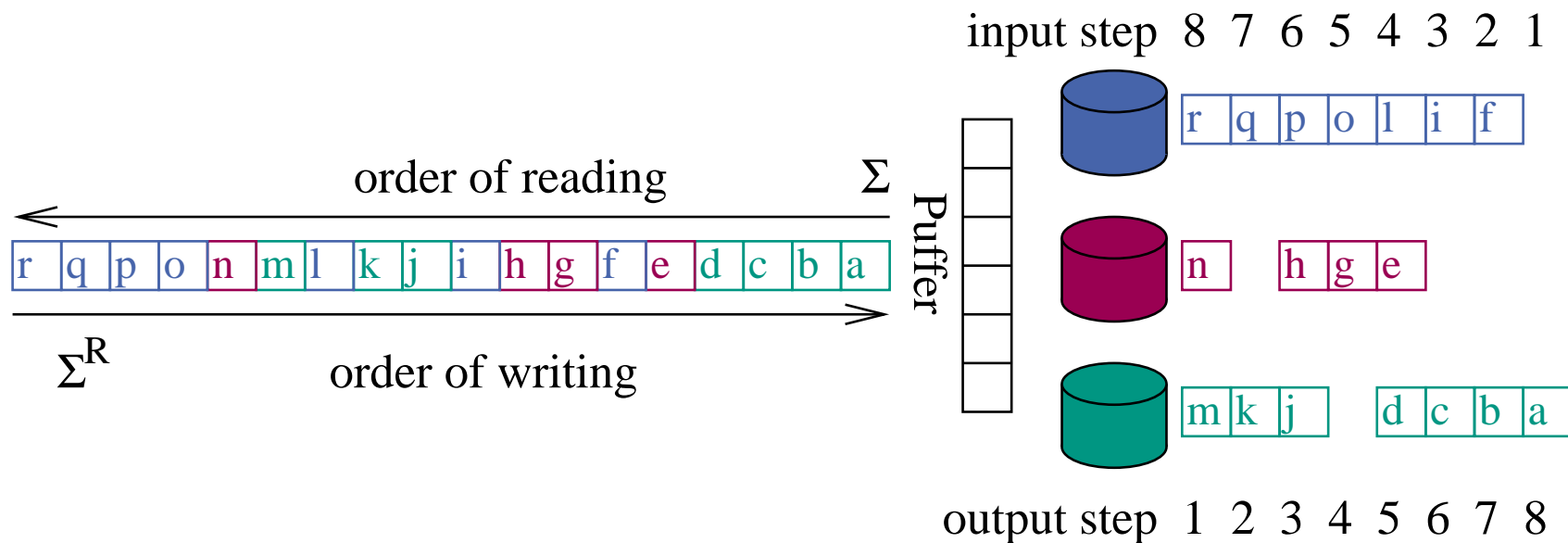
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

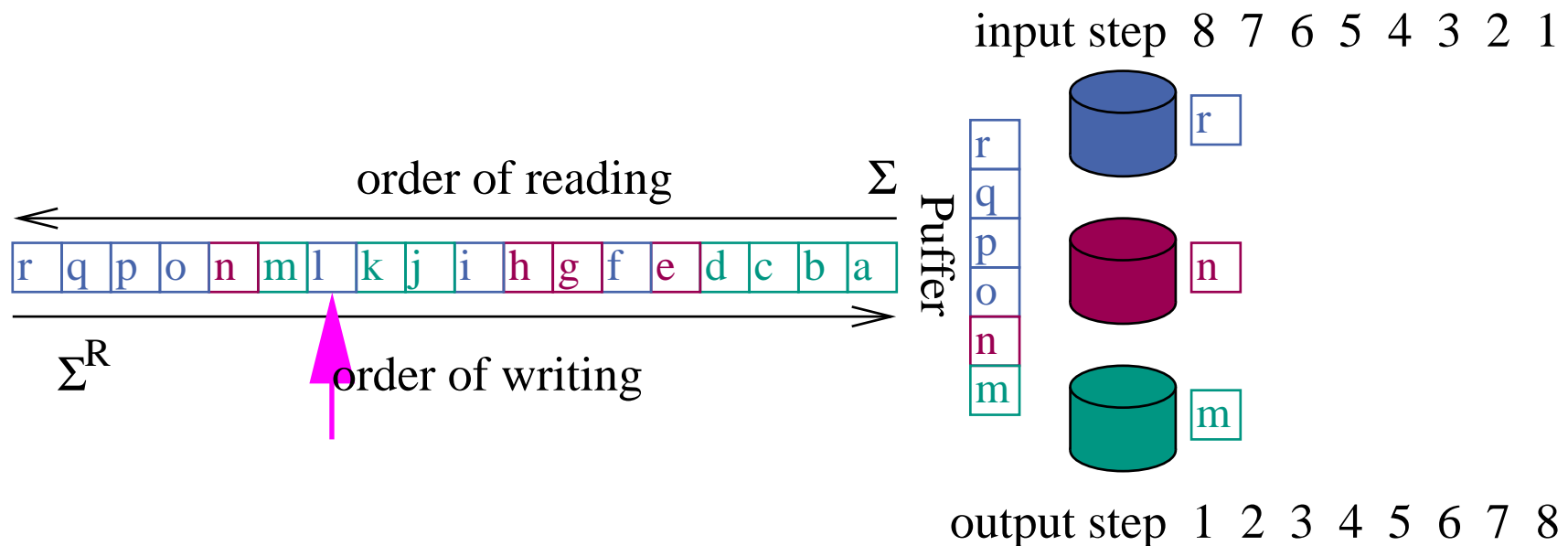
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

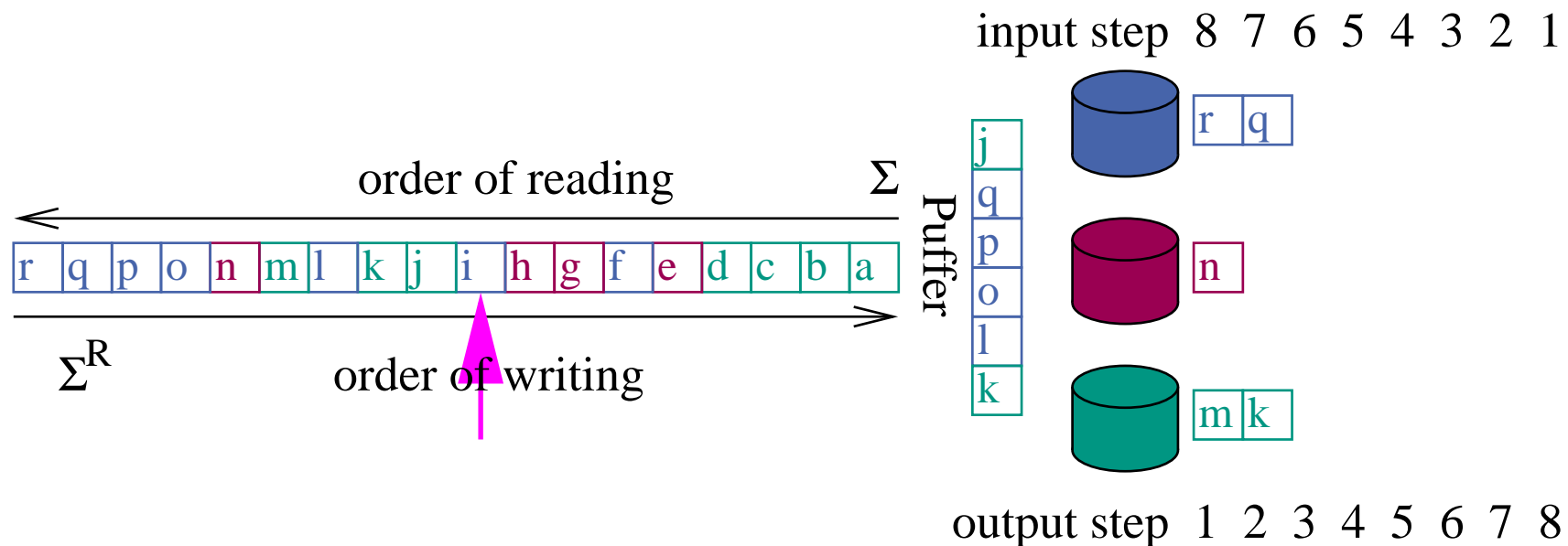
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

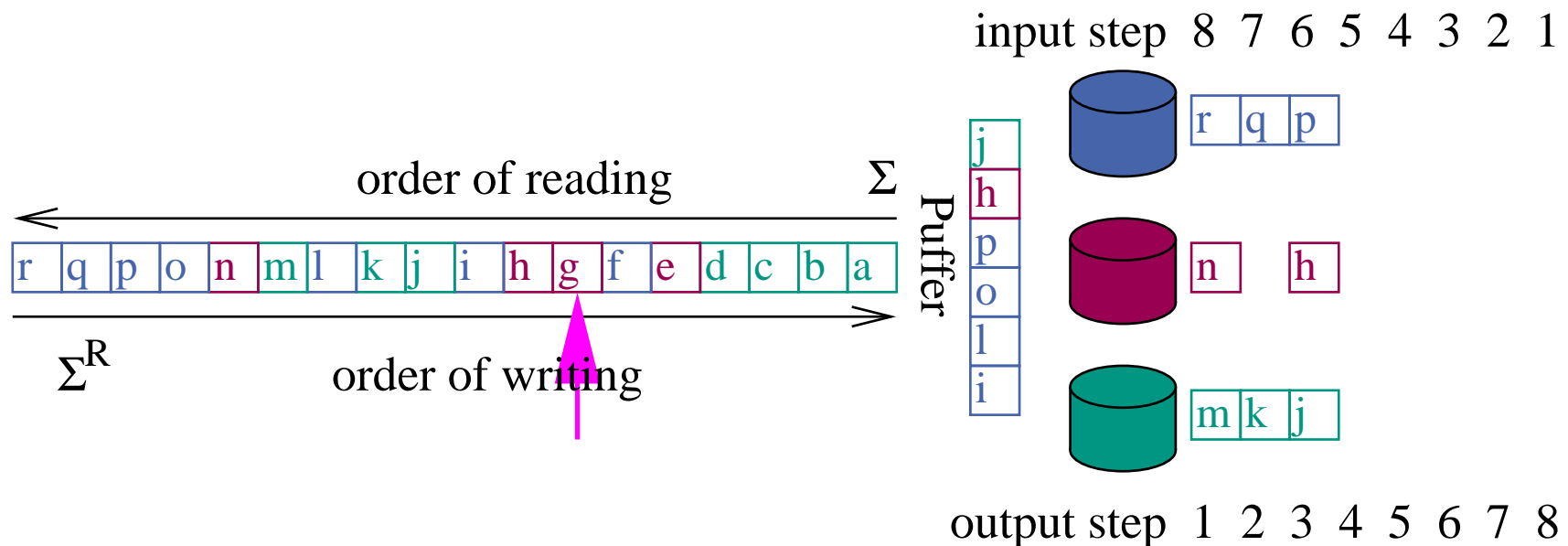
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

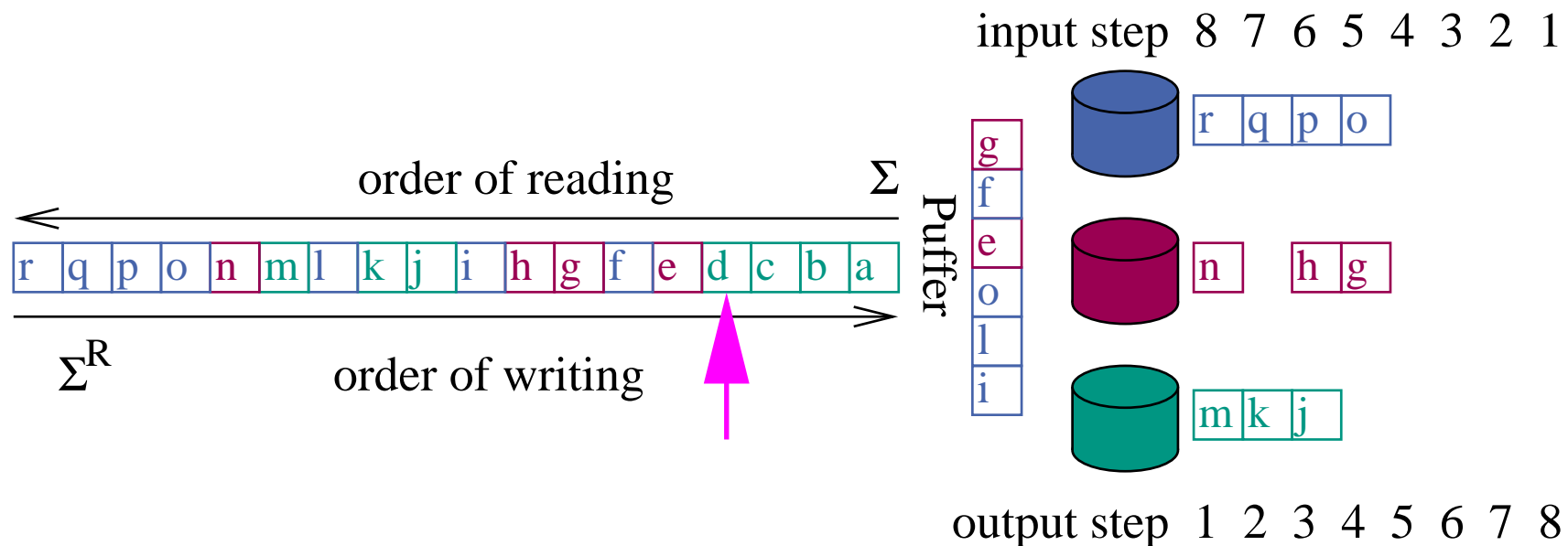
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

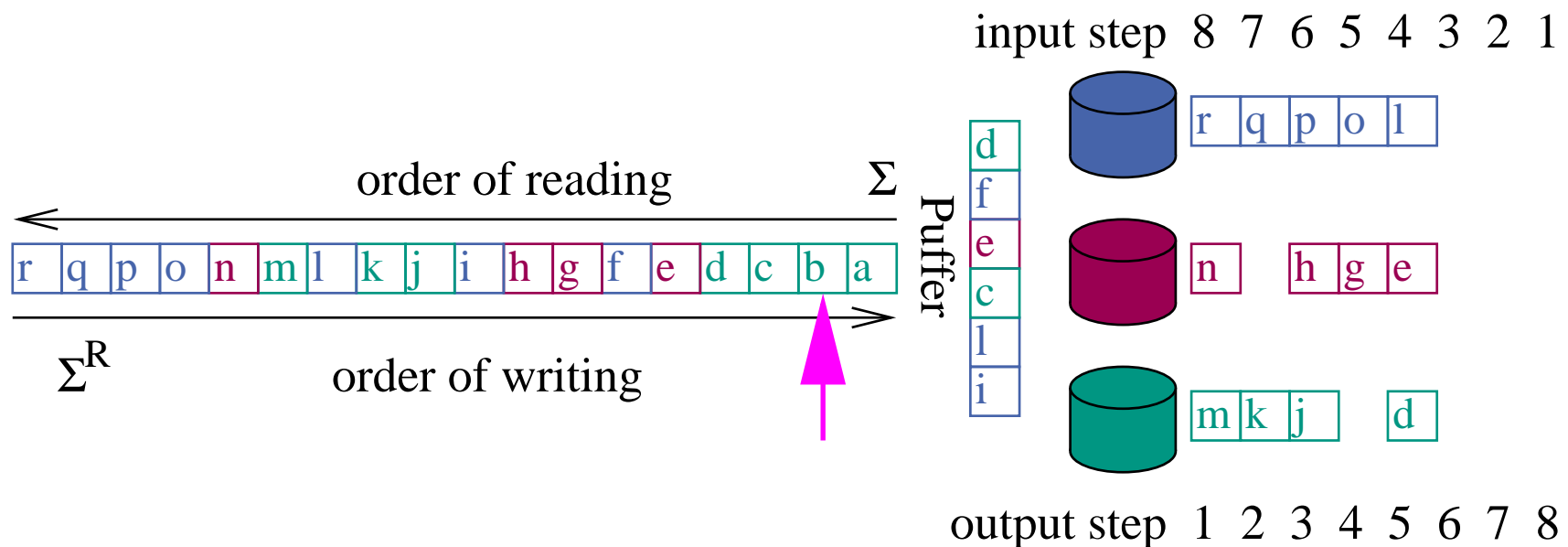
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

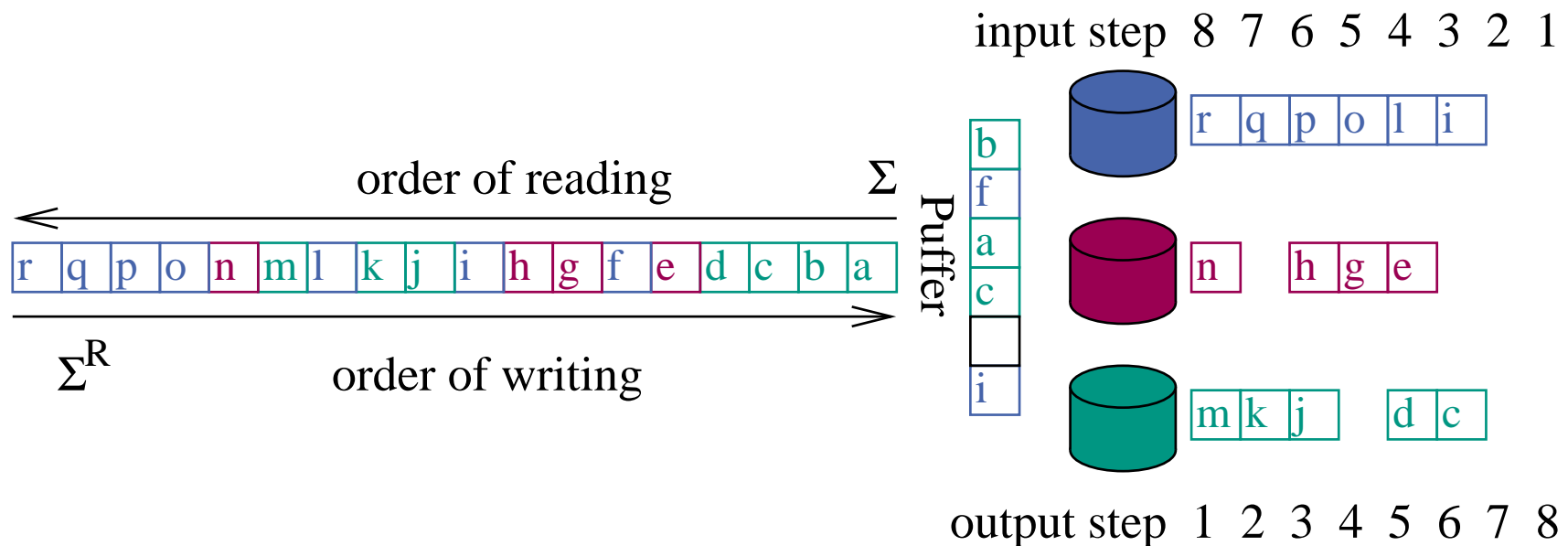
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

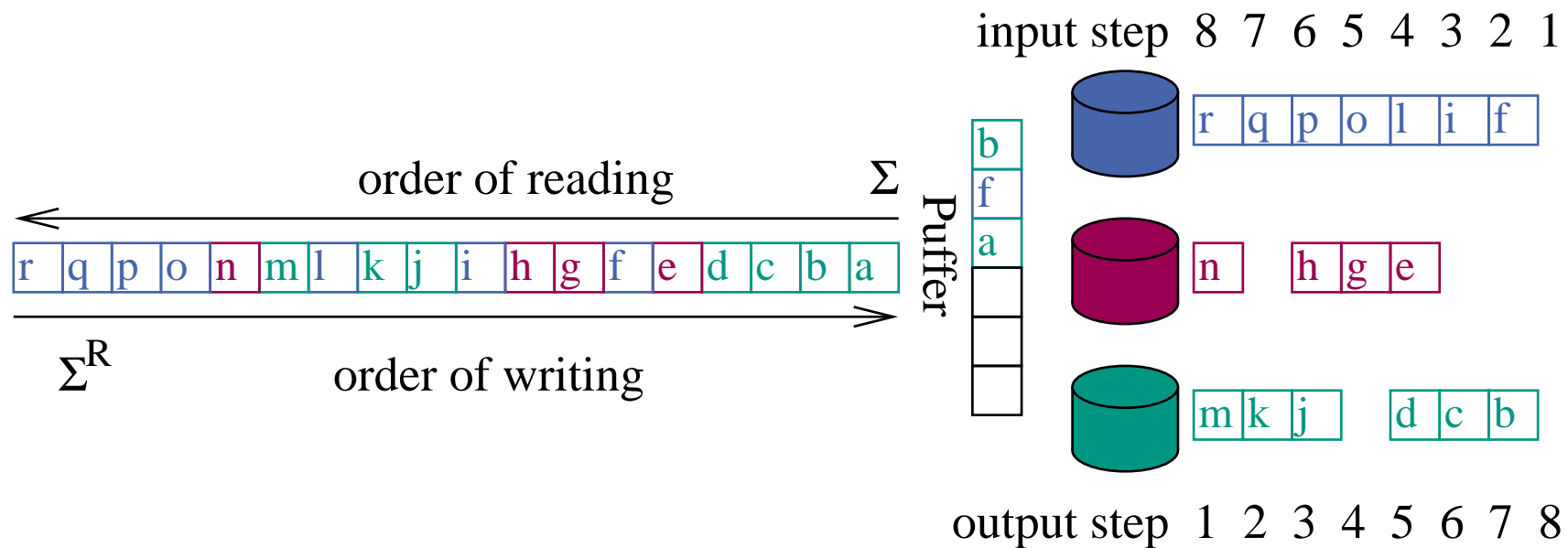
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps

\Leftrightarrow

\exists (online) **write** schedule for Σ^R with T output steps



Optimal Offline Prefetching

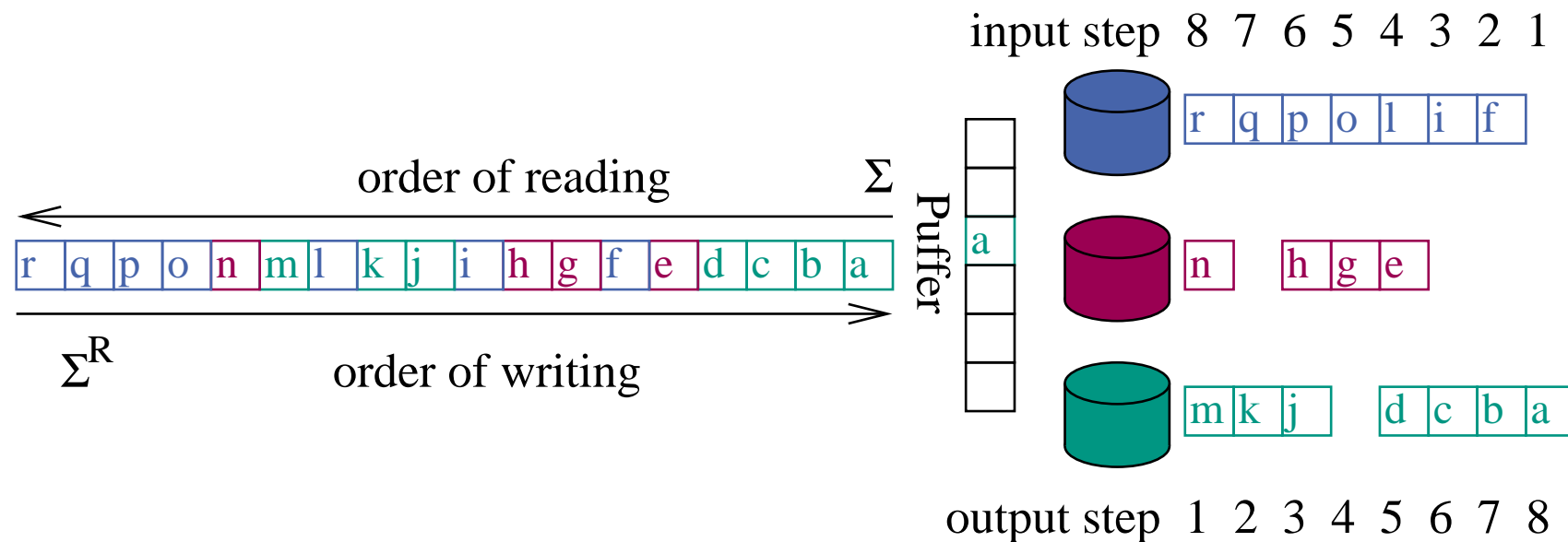
Theorem:

For buffer size W :

\exists (offline) **prefetching** schedule for Σ with T input steps



\exists (online) **write** schedule for Σ^R with T output steps



Synthesis

Multiway merging

+ prediction

[60s Folklore]

+ **optimal (randomized) writing**

[S-Egner-Korst SODA 2000]

+ randomized cycling

[Vitter Hutchinson 2001]

+ **optimal prefetching**

[Hutchinson-S-Vitter ESA 2002]

$\rightsquigarrow (1 + o(1)) \cdot \text{sort}(n)$ I/Os

\rightsquigarrow “answers” question in [Knuth 98];

difficulty 48 on a 1..50 scale.

We are not done yet!

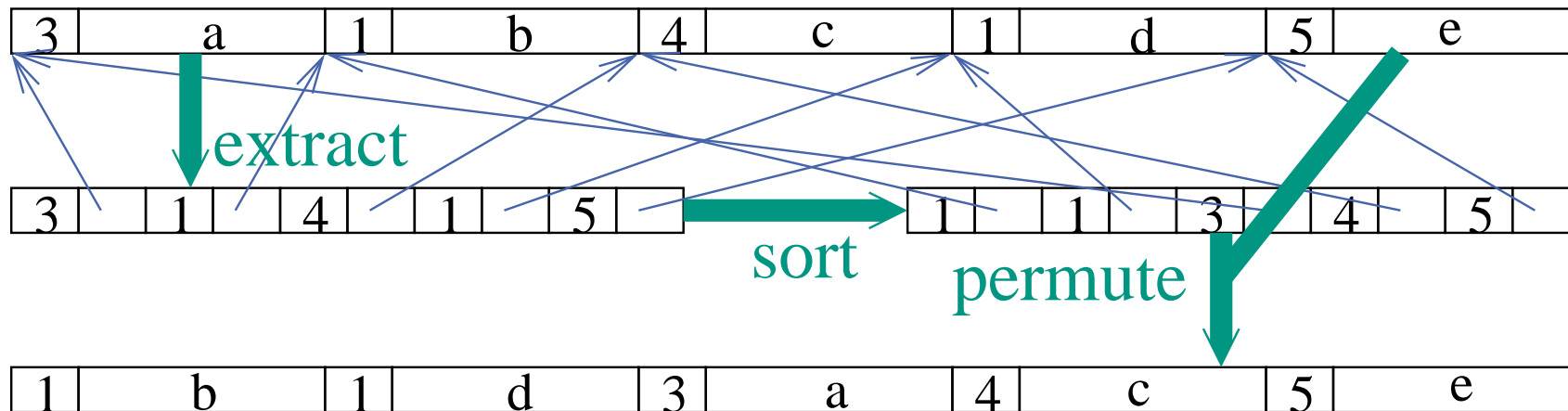
- Internal work
- Overlapping I/O and computation
- Reasonable hardware
- Interfacing with the Operating System
- Parameter Tuning
- Software engineering
- Pipelining

Key Sorting

The **I/O bandwidth** of our machine is about **1/3** of its **main memory** bandwidth

↪ If key size \ll element size

sort key pointer pairs to save memory bandwidth during run formation



Tournament Trees for Multiway Merging

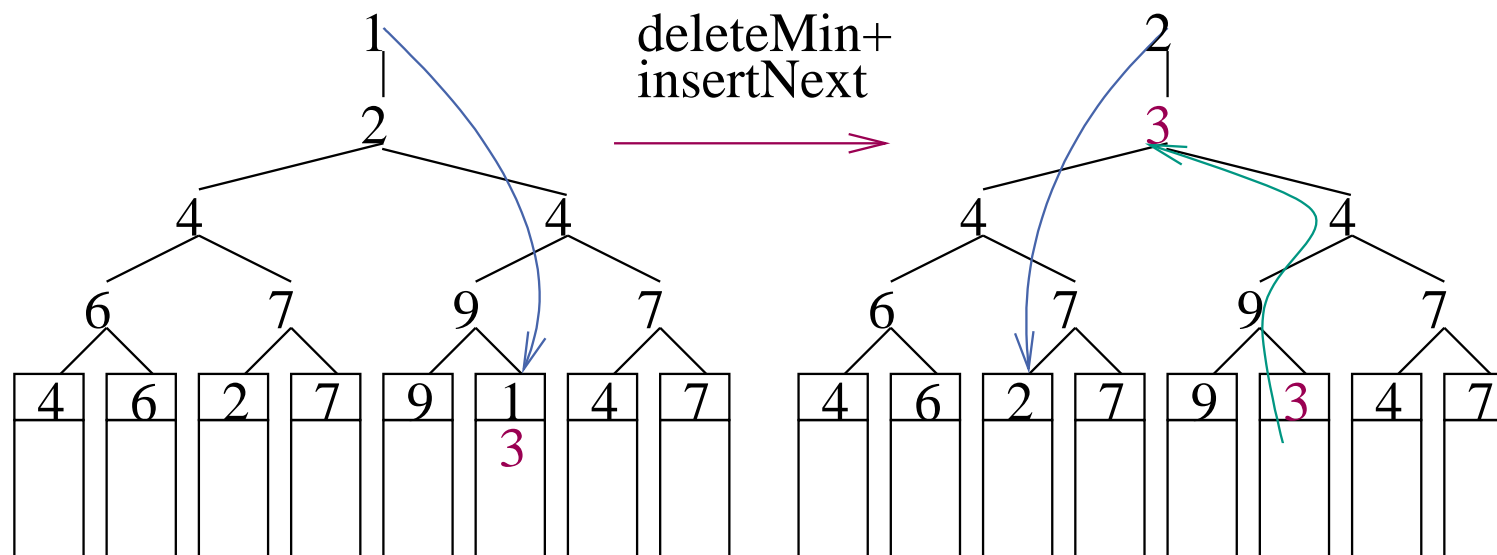
Assume $k = 2^K$ runs

K level complete binary tree

Leaves: smallest current element of each run

Internal nodes: loser of a competition for being smallest

Above root: global winner



Why Tournament Trees

- Exactly $\log k$ element comparisons
- **Implicit layout** in an array \rightsquigarrow simple index arithmetics (shifts)
- **Predictable** load instructions and index computations

(Unrollable) inner loop:

```
for (int i=(winnerIndex+kReg)>>1; i>0; i>>=1) {  
    currentPos = entry + i;  
    currentKey = currentPos->key;  
    if (currentKey < winnerKey) {  
        currentIndex = currentPos->index;  
        currentPos->key = winnerKey;  
        currentPos->index = winnerIndex;  
        winnerKey = currentKey;  
        winnerIndex = currentIndex; } }
```

Overlapping I/O and Computation

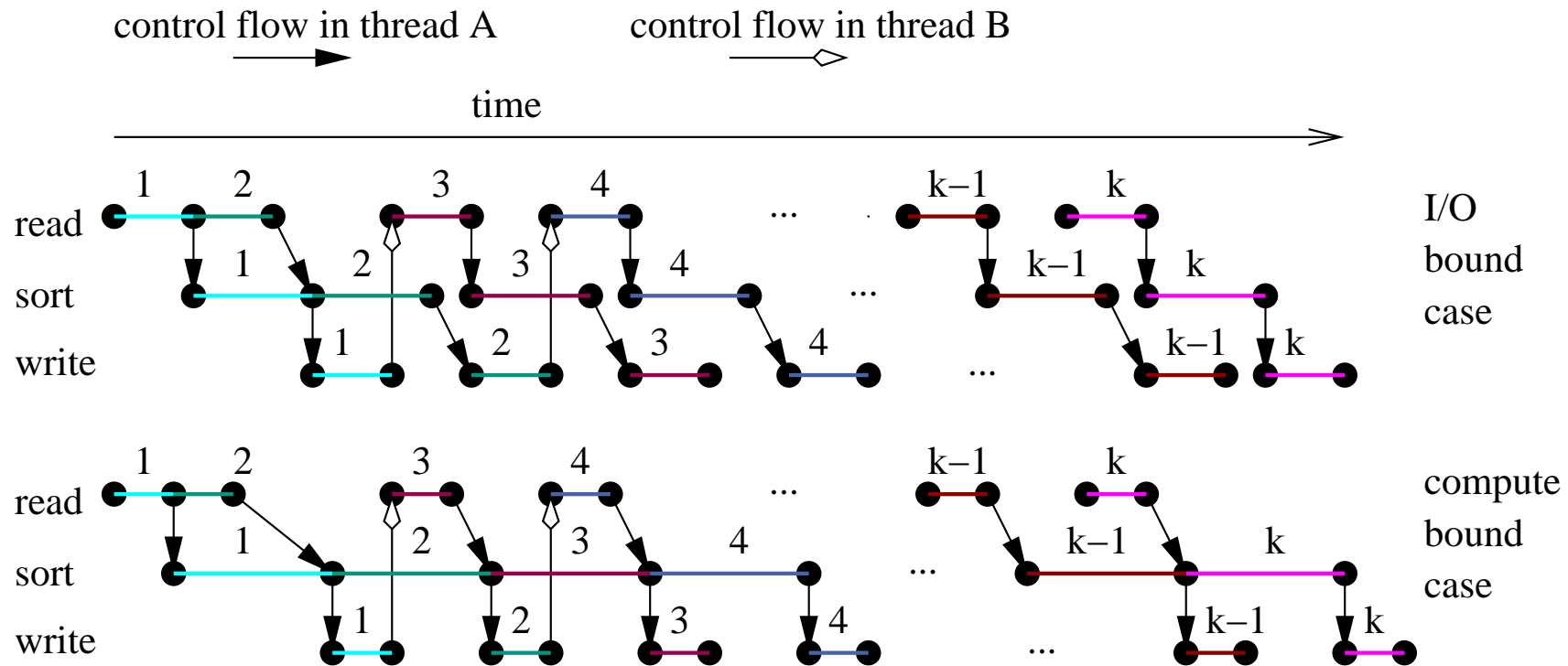
- One thread for each disk (or asynchronous I/O)
- Possibly additional threads
- Blocks filled with elements are passed **by references** between different buffers

Overlapping During Run Formation

First post **read** requests for runs 1 and 2

Thread A: Loop { wait-read i ; sort i ; post-write i }; sorting thread

Thread B: Loop { wait-write i ; post-read $i + 2$ }; prefetch thread



Overlapping During Merging

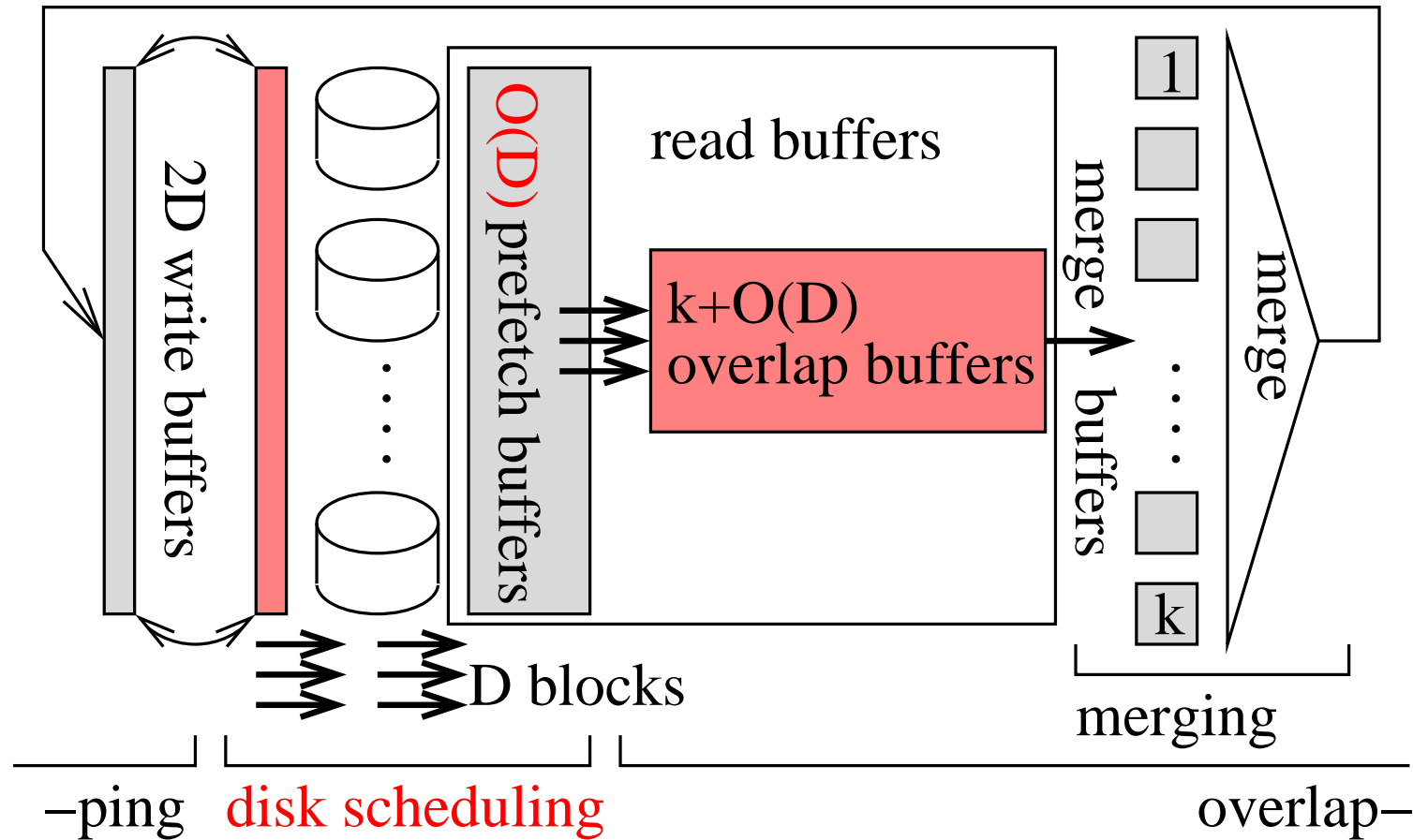
Bad example:

$$\boxed{1^{B-1}2} \boxed{3^{B-1}4} \boxed{5^{B-1}6} \dots$$

...

$$\boxed{1^{B-1}2} \boxed{3^{B-1}4} \boxed{5^{B-1}6} \dots$$

Overlapping During Merging



I/O Threads: Writing has priority over reading

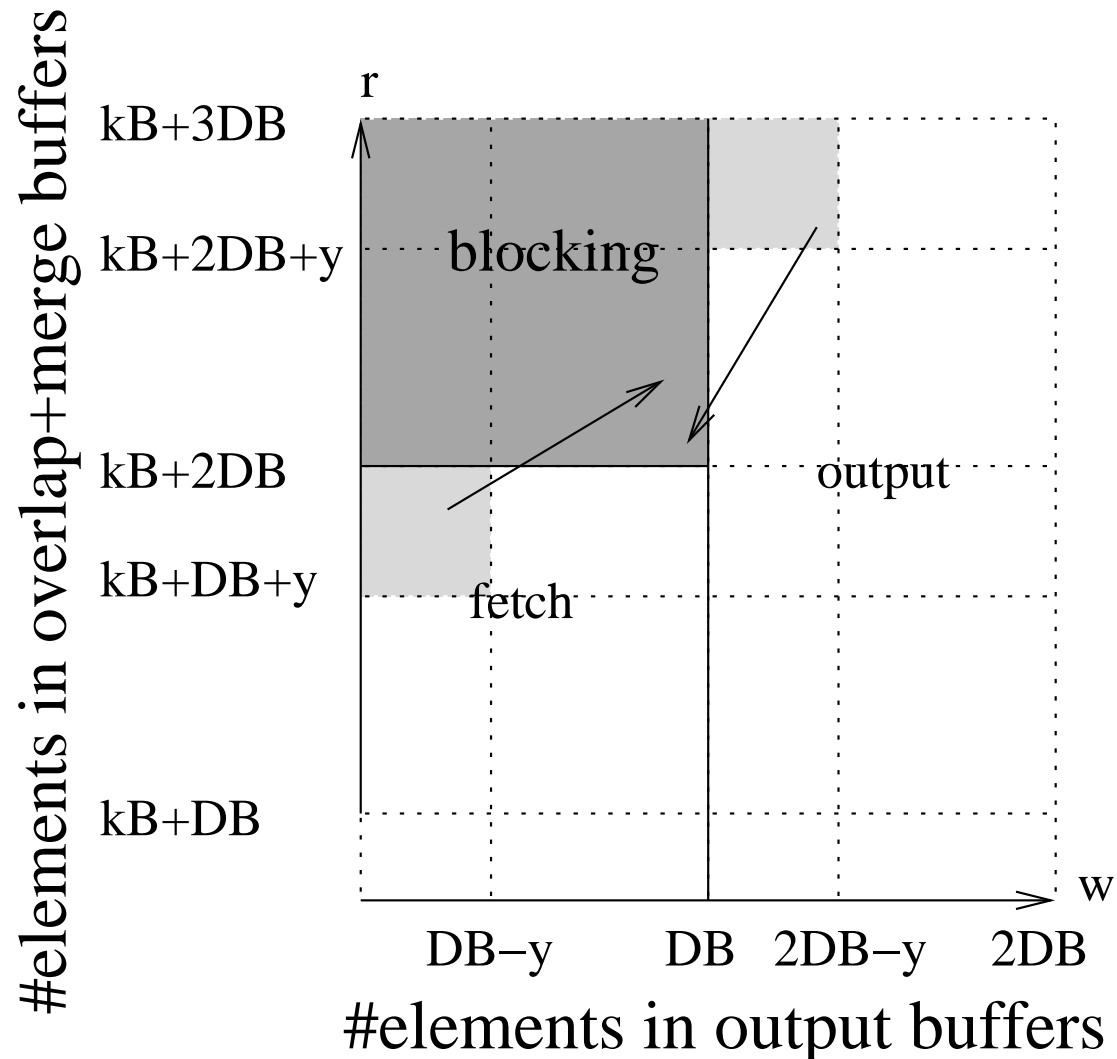
I/O bound case: prefetch thread never blocks

$y = \#$ of elements merged during one I/O step.

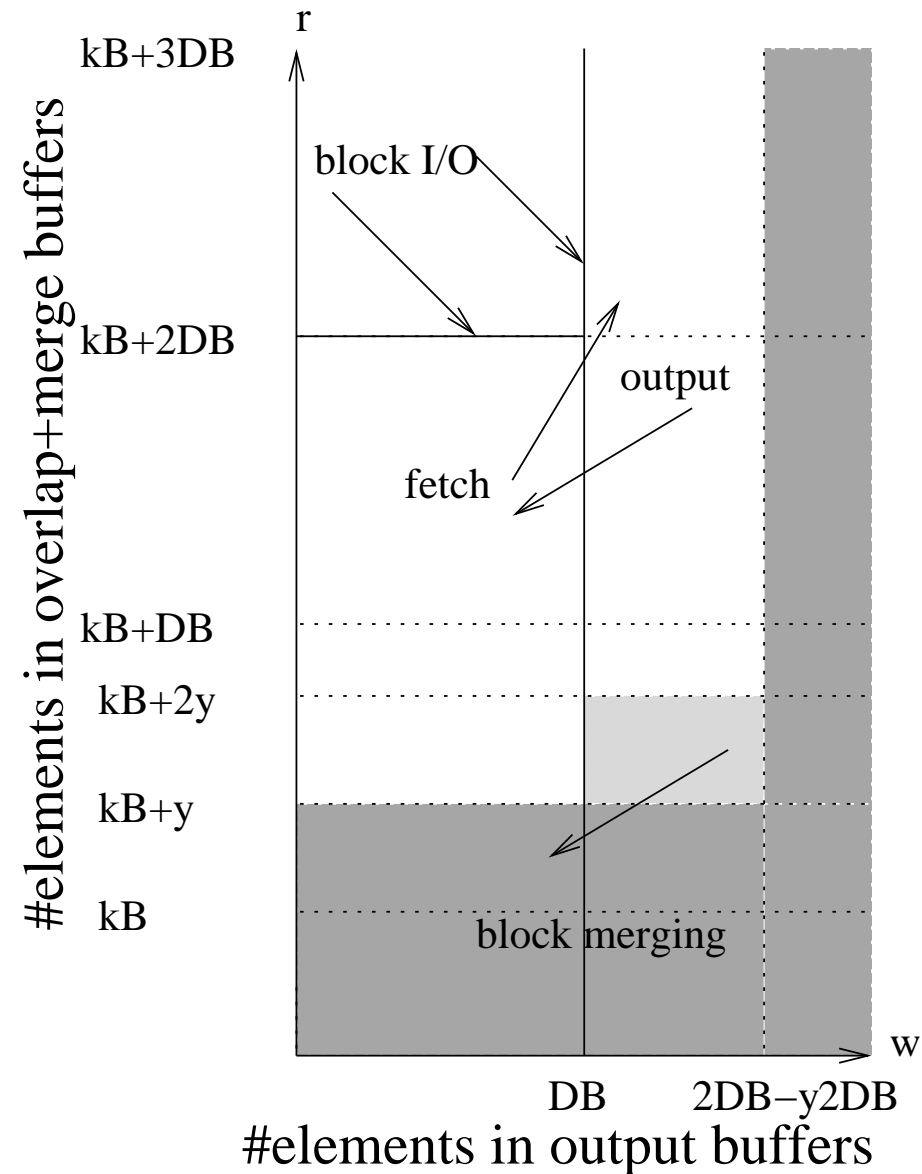
I/O bound \rightsquigarrow

$$y > \frac{DB}{2}$$

$$y \leq DB$$



Compute bound case: The merging thread never blocks



Hardware (mid 2002)

Linux

(2 × 2GHz Xeon × 2 Threads) ^{400x64 Mb/s}

Several 66 MHz PCI-buses

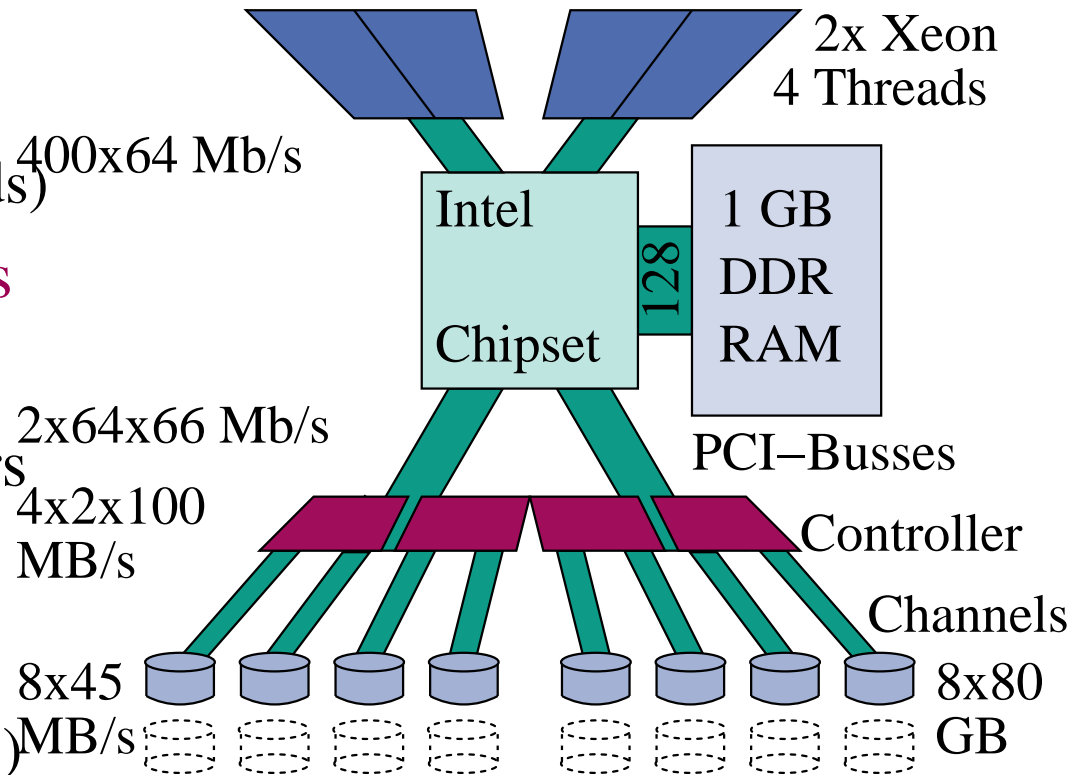
(SuperMicro P4DPE3)

Several fast IDE controllers

(4 × Promise Ultra100 TX2) ^{4x2x100 MB/s}

Many fast IDE disks

(8 × IBM IC35L080AVVA07) ^{8x45 MB/s}



cost effective I/O-bandwidth

(real 360 MB/s for ≈ 3000) €

Hardware (end 2009) geschätzt

Linux

(2 × 2.4 GHz Xeon E5530 × 4 Cores × 2 Threads)

PCIe x8 SATA controller

16–24 1.5 TByte SATA disks

(8 × IBM IC35L080AVVA07)

24 GByte RAM

cost effective I/O-bandwidth

(real 2 GB/s for ≈ 6000) €

Hardware 2015 geschätzt

≈3000 Euro for 32 512 GB SATA SSDs a 93 Euro:

↪ 16TB capacity, and

↪ 16GB/s read bandwidth ?

64 GB/RAM 800 Euro

500 Euro Motherboard

2x8 cores Intel Xeon E5-2603v3 (a 200 Euro)

Hardware 2017 geschätzt

≈2000 Euro for 4 1TB M.2 SSD a 500 Euro:

↪ 4TB capacity, and

↪ 14GB/s read bandwidth ?

128 GB/RAM 1000 Euro

2x6 cores Intel Xeon E5-2603vv (a 240 Euro)

Hardware 2019 geschätzt

≈1720 Euro for 8 2TB M.2 SSD a 215 Euro:

↪ 16TB capacity, and

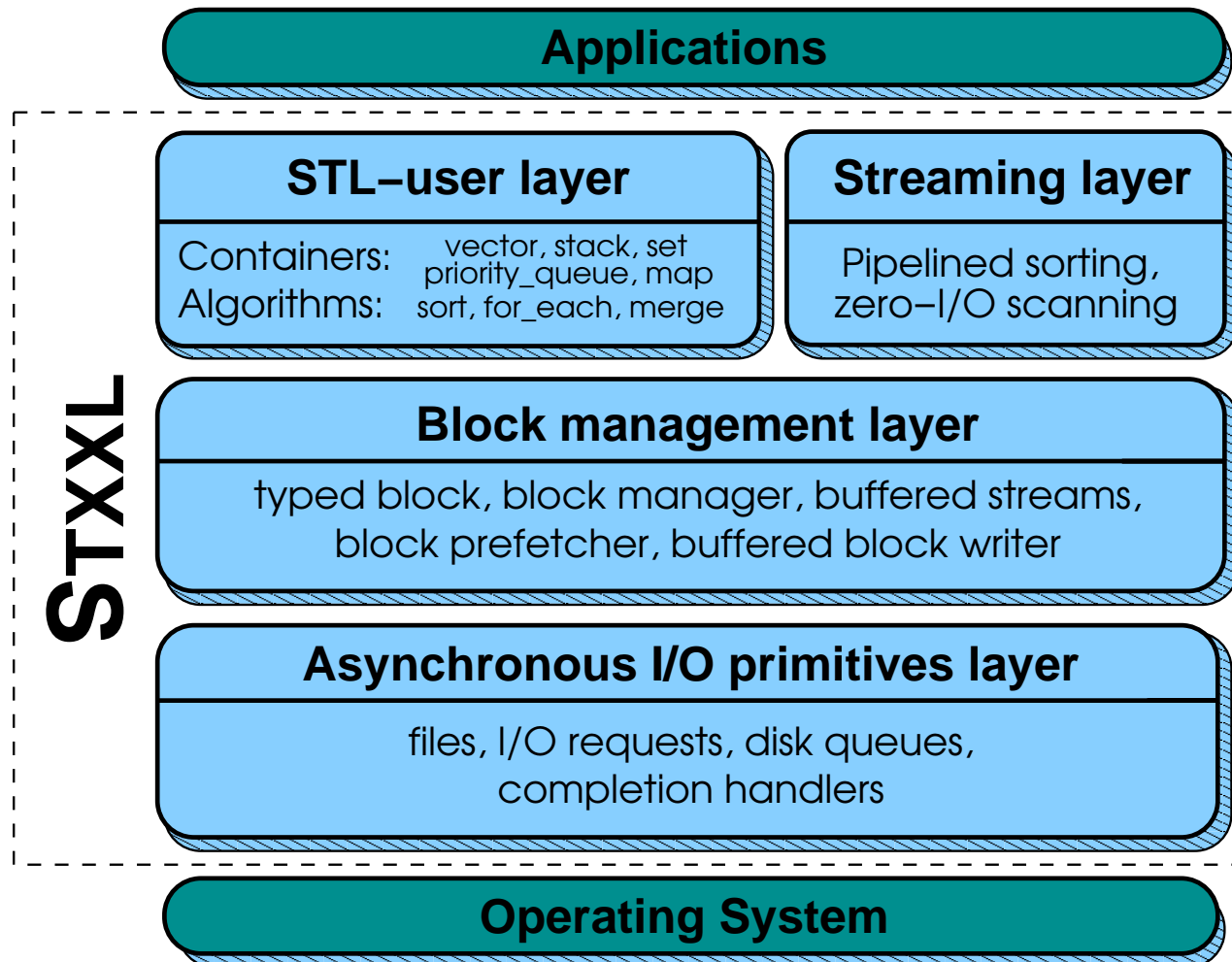
↪ 14.4GB/s read bandwidth ?

128 GB/RAM 550 Euro

8 cores AMD EPYC 7251 (555 \$)

Software Interface

Goals: **efficient** + **simple** + **compatible**



Default Measurement Parameters

$t :=$ number of available buffer blocks

Input Size: 16 GByte

Element Size: 128 Byte

Keys: Random 32 bit integers

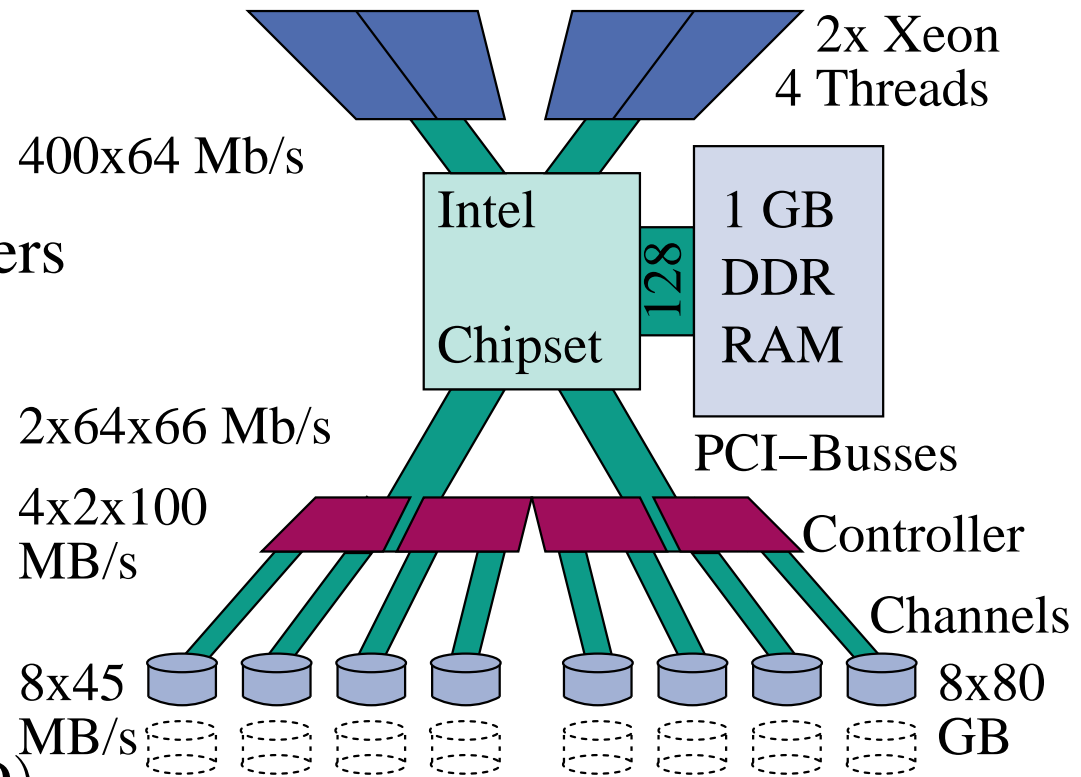
Run Size: 256 MByte

Block size B : 2 MByte

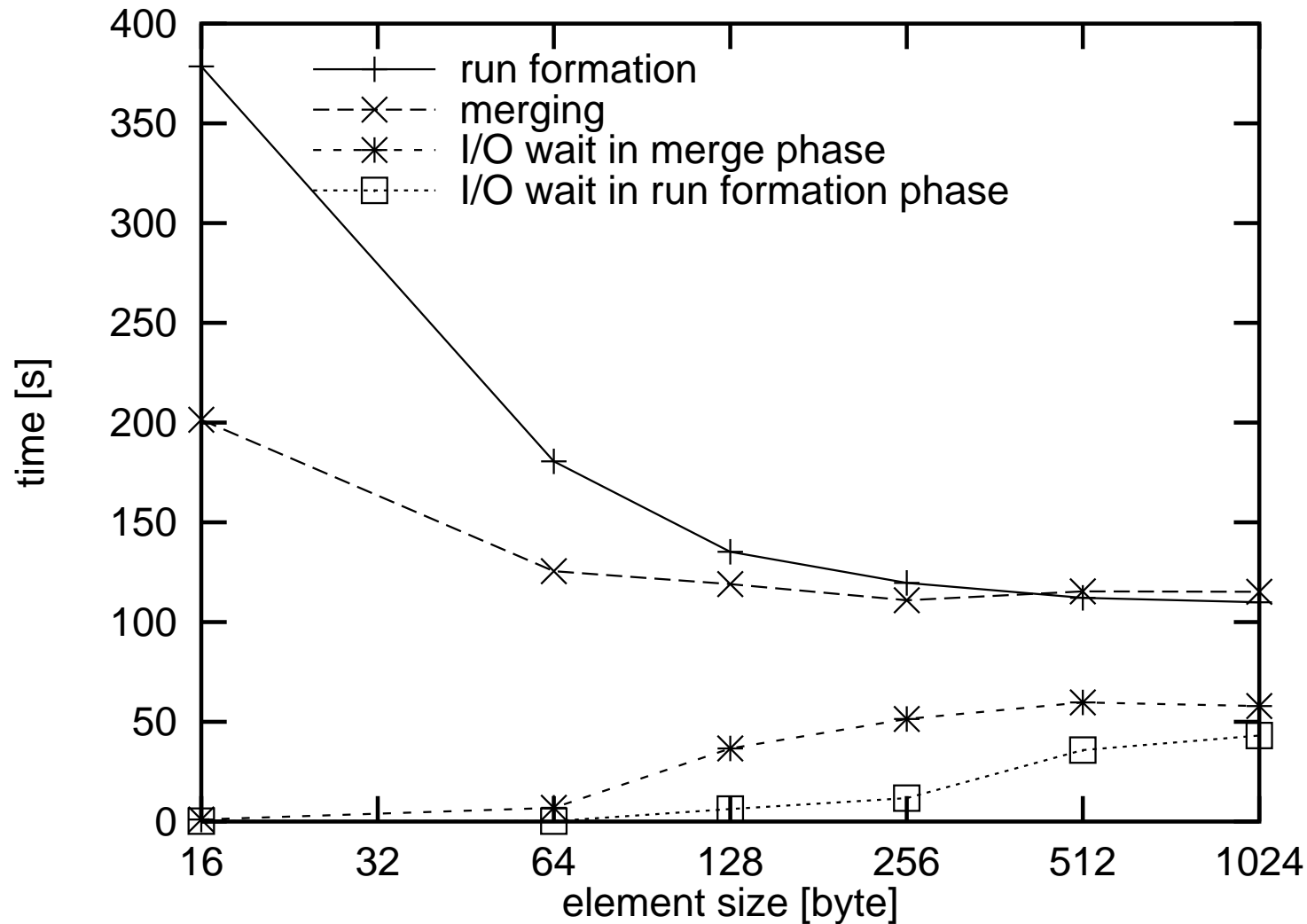
Compiler: g++ 3.2 -O6

Write Buffers: $\max(t/4, 2D)$

Prefetch Buffers: $2D + \frac{3}{10}(t - w - 2D)$



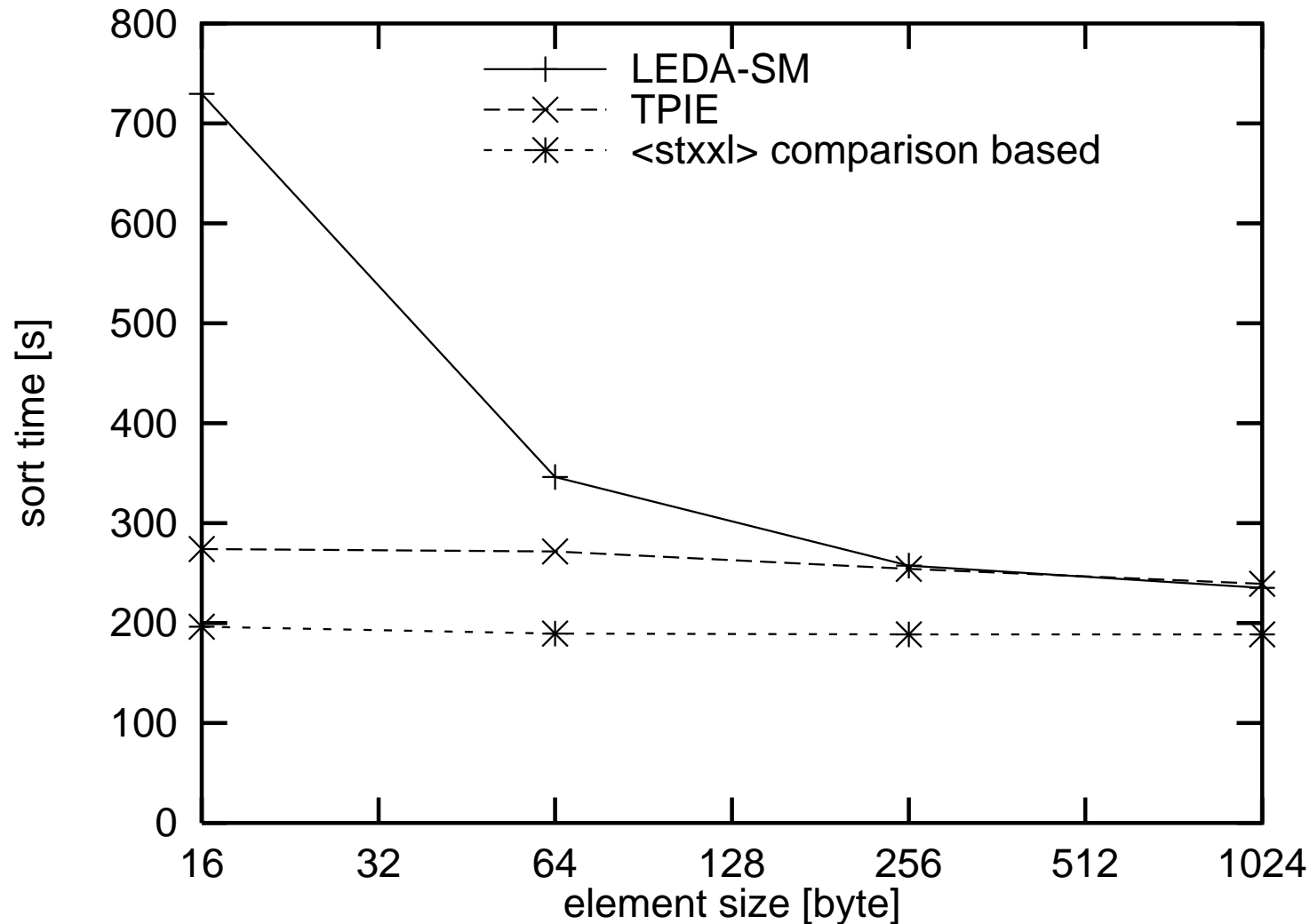
Element sizes (16 GByte, 8 disks)



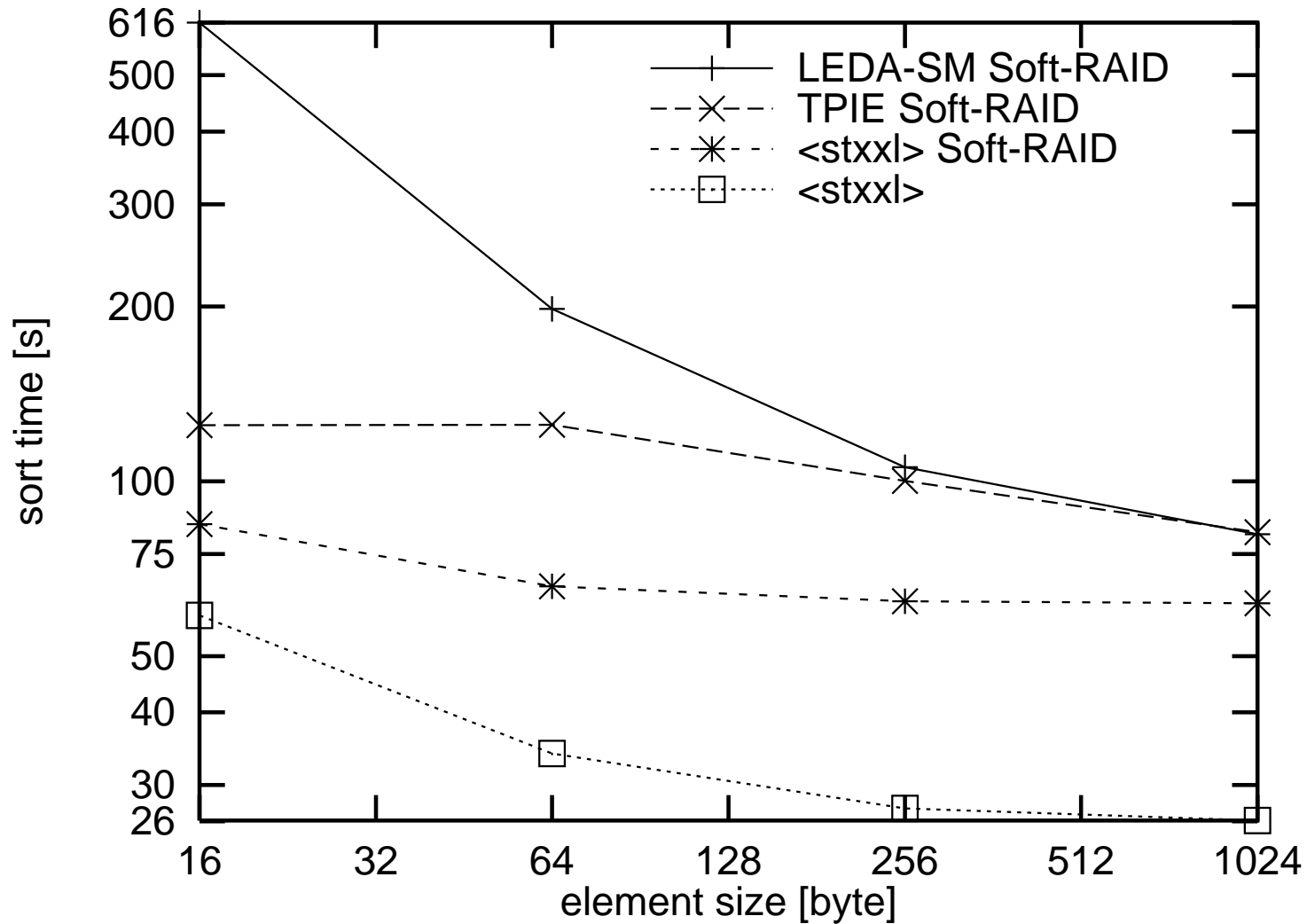
parallel disks \rightsquigarrow **bandwidth** “for free” \rightsquigarrow internal work, overlapping are relev

Earlier Academic Implementations

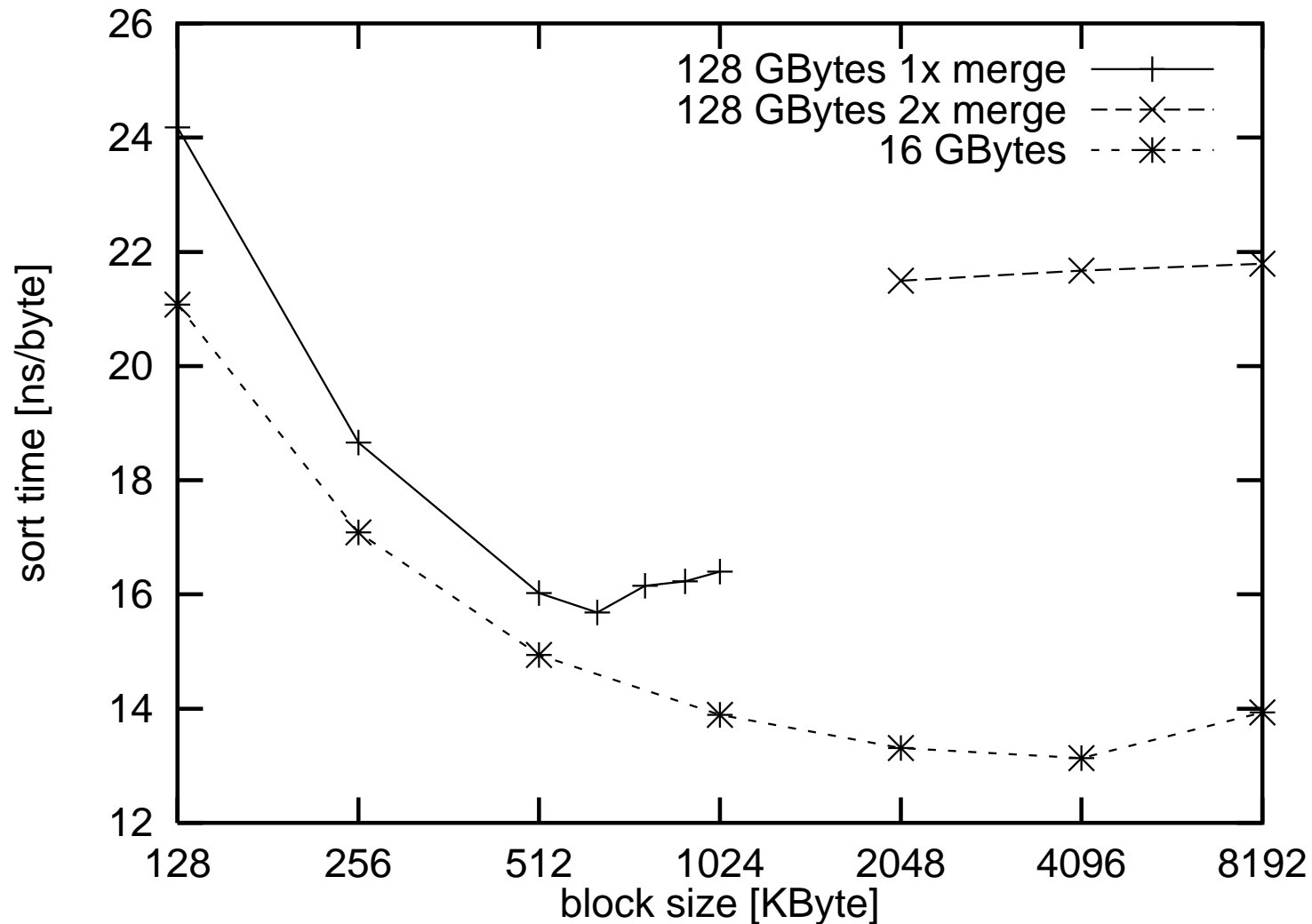
Single Disk, at most 2 GByte, old measurements use artificial M



Earlier Acad. Implementations: Multiple Disks



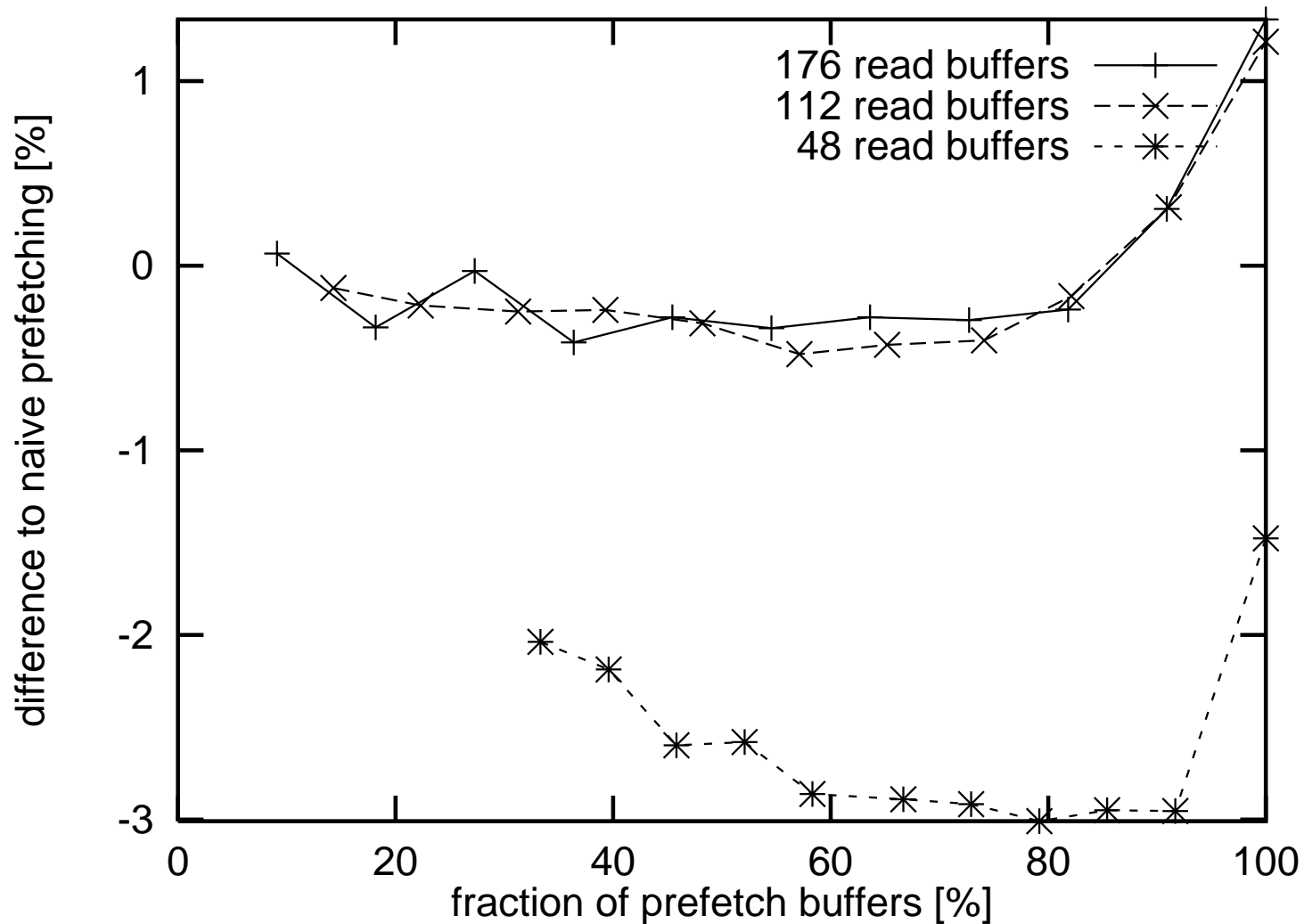
What are good block sizes (8 disks)?



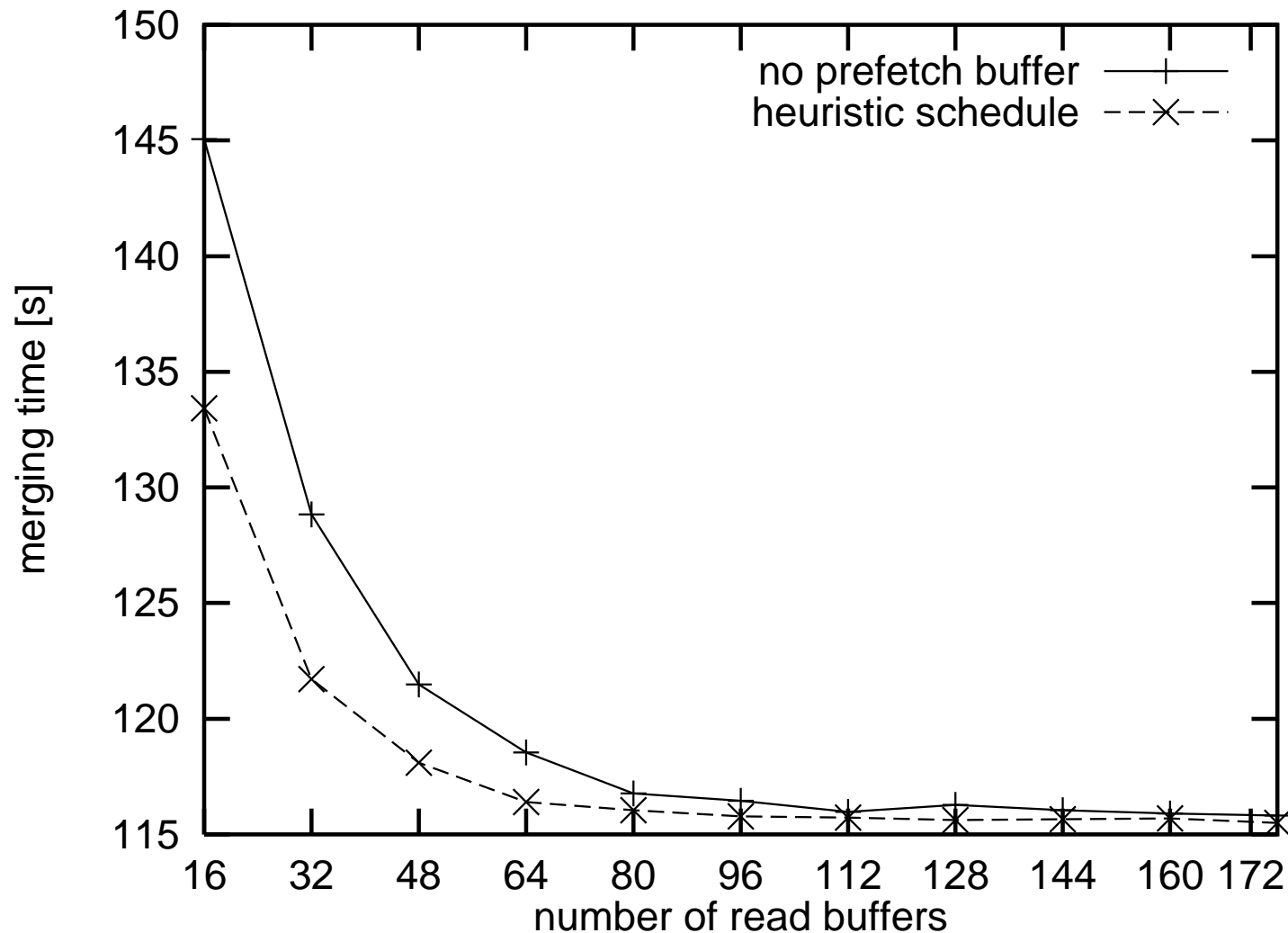
B is **not** a technology constant

Optimal Versus Naive Prefetching

Total merge time

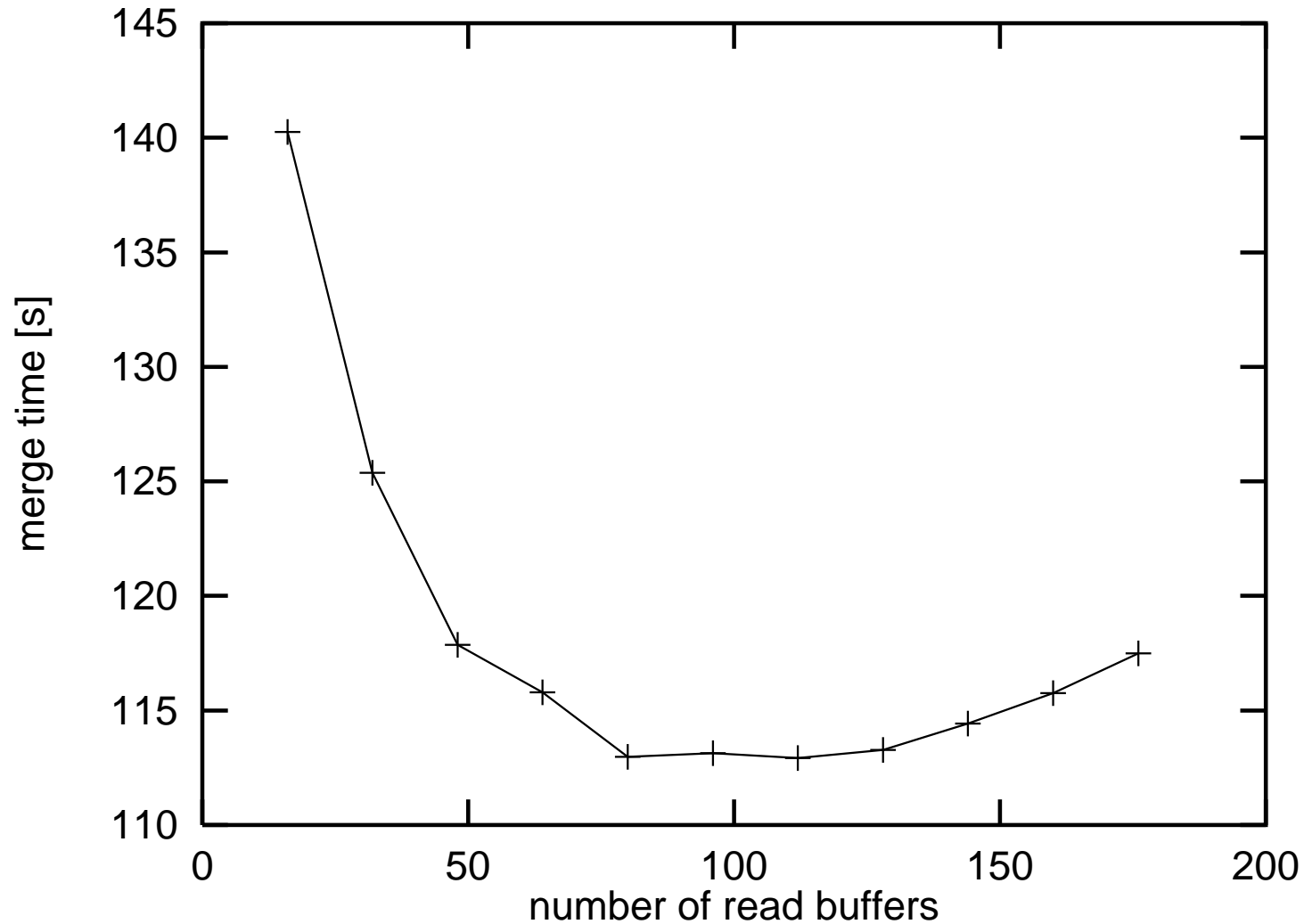


Impact of Prefetch and Overlap Buffers

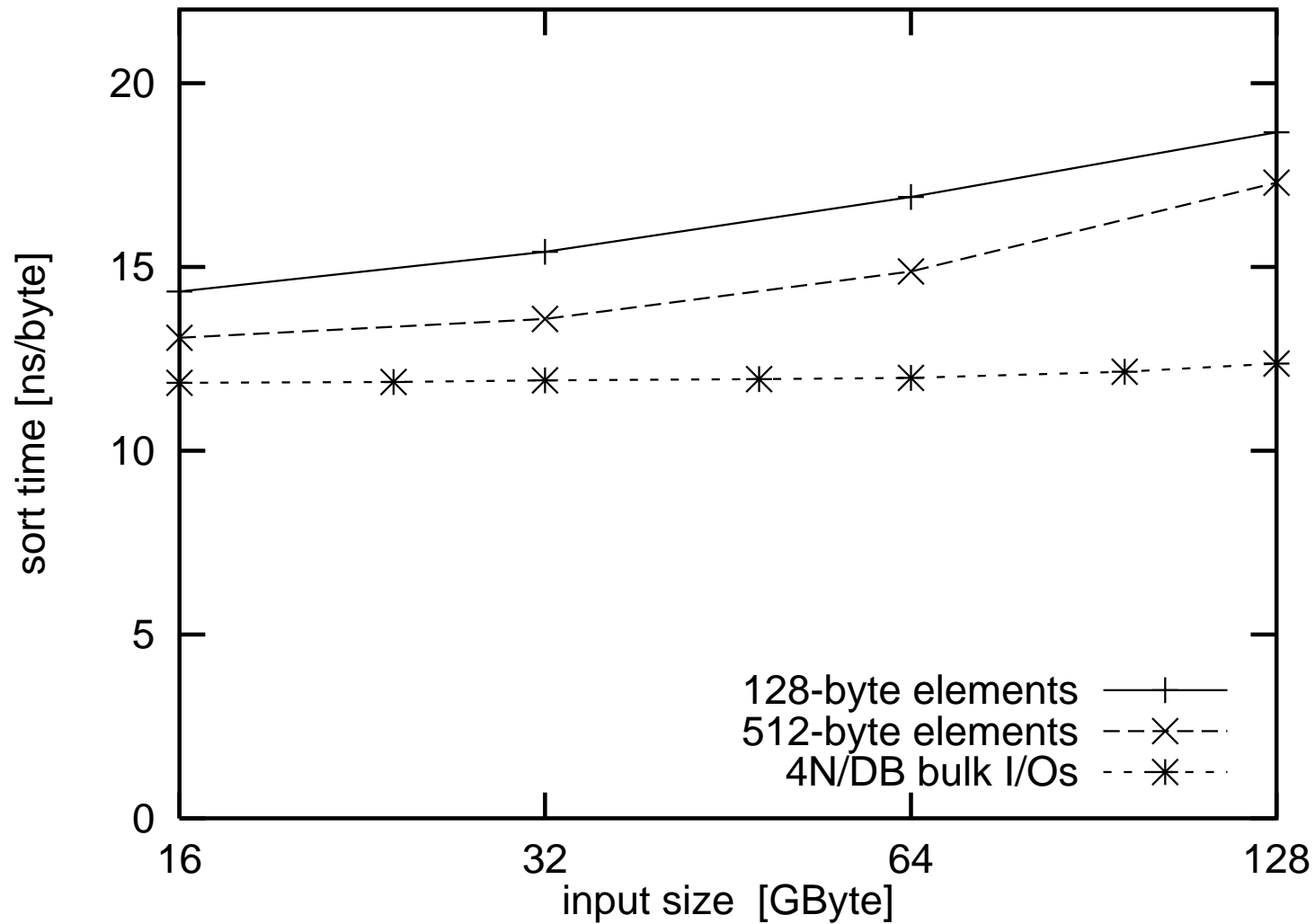


Tradeoff: Write Buffer Size Versus Read Buffer Size

Size



Scalability



Discussion

- Theory and practice harmonize
- No expensive server hardware necessary (SCSI,...)
- No need to work with artificial M
- No 2/4 GByte limits
- Faster than academic implementations
- (Must be) as fast as commercial implementations but with performance guarantees
- Blocks are much larger than often assumed. Not a technology constant
- Parallel disks \rightsquigarrow
bandwidth “for free” \rightsquigarrow don’t neglect internal costs

More Parallel Disk Sorting?

Pipelining: Input does not come from disk but from a logical input stream. Output goes to a logical output stream
~> only half the I/Os for sorting
~> often **no I/Os** for scanning **todo: better overlapping**

Parallelism: This is the only way to go for **really many** disks

Tuning and Special Cases: sssort, permutations, balance work between merging and run formation?...

Longer Runs: not done with guaranteed overlapping, fast internal sorting !

Distribution Sorting: Better for seeks etc.?

Inplace Sorting: Could also be faster

Determinism: A practical and theoretically efficient algorithm?

Procedure formLongRuns

q, q' : PriorityQueue

for $i := 1$ **to** M **do** q .insert(readElement)

invariant $|q| + |q'| = M$

loop

while $q \neq \emptyset$

writeElement($e := q$.deleteMin)

if input exhausted **then** break outer loop

if $e' := \text{readElement} < e$ **then** q' .insert(e')

else q .insert(e')

$q := q'$; $q' := \emptyset$

output q in sorted order; output q' in sorted order

Knuth: average run length $2M$

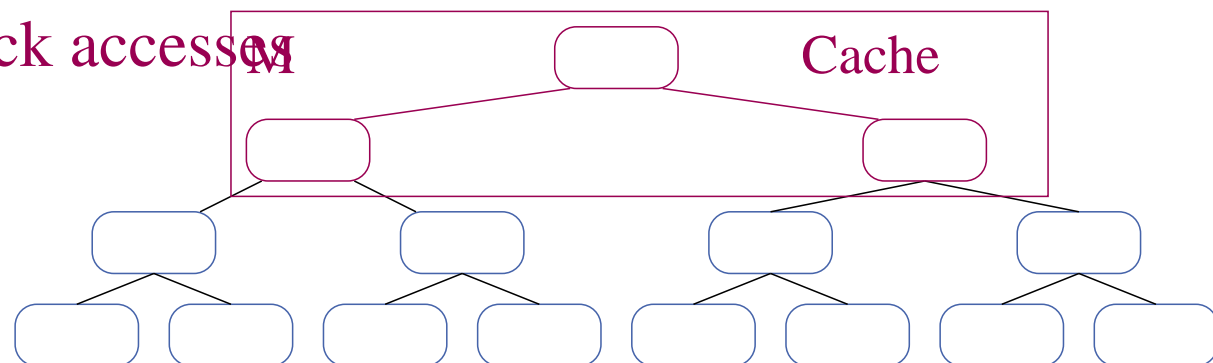
todo: cache-effiziente Implementierung

2 Priority Queues (insert, deleteMin)

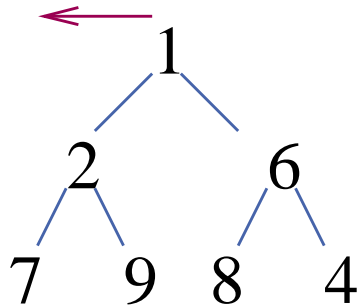
Binary Heaps best comparison based “flat memory” algorithm

- + On average **constant** time for **insertion**
- + On average $\log n + \mathcal{O}(1)$ key comparisons per delete-Min using the “bottom-up” heuristics [Wegener 93].

- $\approx \log(n/M)$ block accesses
per delete-Min



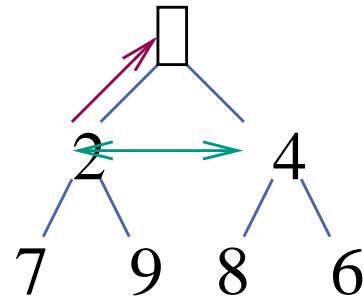
Bottom Up Heuristics



delete Min

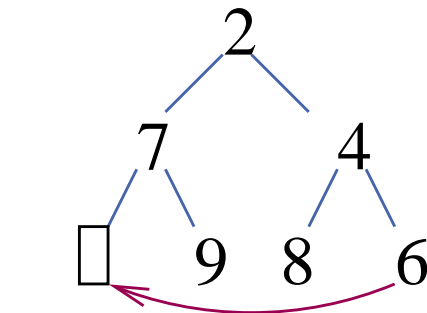
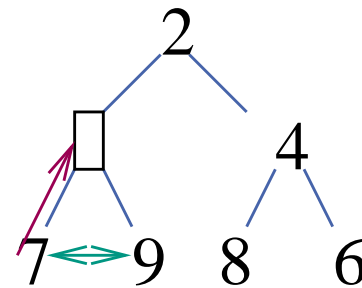
$O(1)$

compare swap move



sift down hole

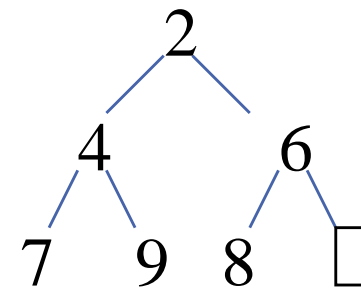
$\log(n)$



sift up

$O(1)$

average



Factor two faster
than naive implementation

Der Wettbewerber fit gemacht:

```
int i=1, m=2, t = a[1];
m += (m != n && a[m] > a[m + 1]);
if (t > a[m]) {
    do { a[i] = a[m];
        i = m;
        m = 2*i;
        if (m > n) break;
        m += (m != n && a[m] > a[m + 1]);
    } while (t > a[m]);
    a[i] = t; }
```

Keine signifikanten Leistungsunterschiede auf meiner Maschine
(heapsort von random integers)

Vergleich

Speicherzugriffe: $\mathcal{O}(1)$ weniger als top down.

$\mathcal{O}(\log n)$ worst case. bei **effizienter Implementierung**

Elementvergleiche: $\approx \log n$ weniger für bottom up (average case) aber die sind **leicht vorhersagbar**

Aufgabe: siftDown mit **worst case** $\log n + \mathcal{O}(\log \log n)$

Elementvergleichen

Heapkonstruktion

Procedure buildHeapBackwards

for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** siftDown(i)

Procedure buildHeapRecursive($i : \mathbb{N}$)

if $4i \leq n$ **then**

 buildHeapRecursive($2i$)

 buildHeapRecursive($2i + 1$)

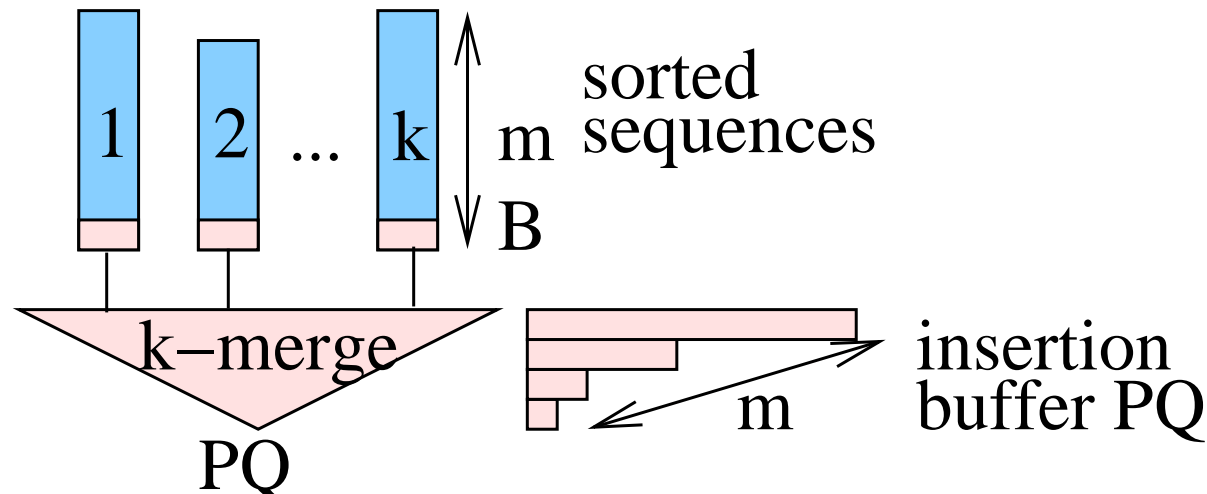
siftDown(i)

Rekursive Funktion für große Eingaben $2 \times$ schneller!

(Rekursion abrollen für 2 unterste Ebenen)

Aufgabe: Erklärung

Mittelgroße PQs – $km \ll M^2 / B$ Einfügungen



Insert: Anfangs in **insertion buffer**.

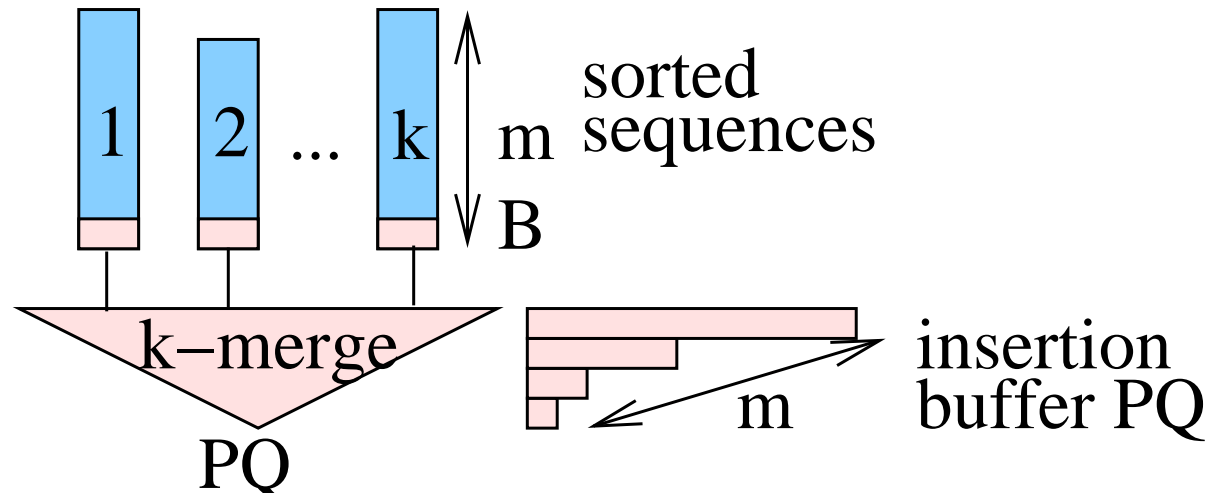
Überlauf \longrightarrow

sort; flush; kleinster Schlüssel in merge-PQ

Delete-Min: deleteMin aus der PQ mit kleinerem min

Analyse – I/Os

deleteMin: jedes Element wird $\leq 1 \times$ gelesen, zusammen mit B anderen – **amortisiert** $1/B$ penalty für **insert**.



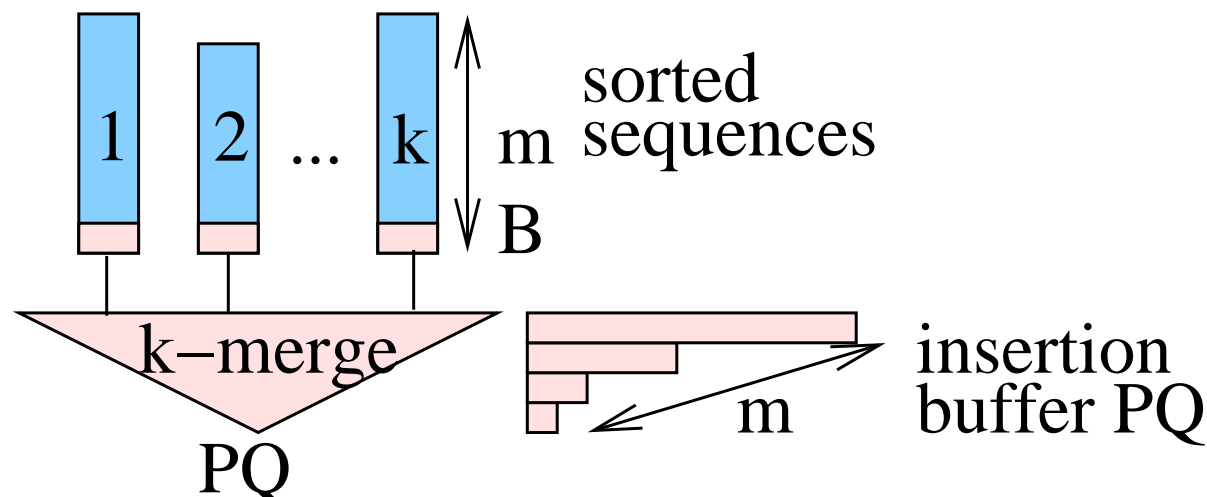
Analyse – Vergleiche (Maß für interne Arbeit)

deleteMin: $1 + \mathcal{O}(\max(\log k, \log m)) = \mathcal{O}(\log m)$

genauere Argumentation: amortisiert $1 + \log k$ bei geeigneter PQ

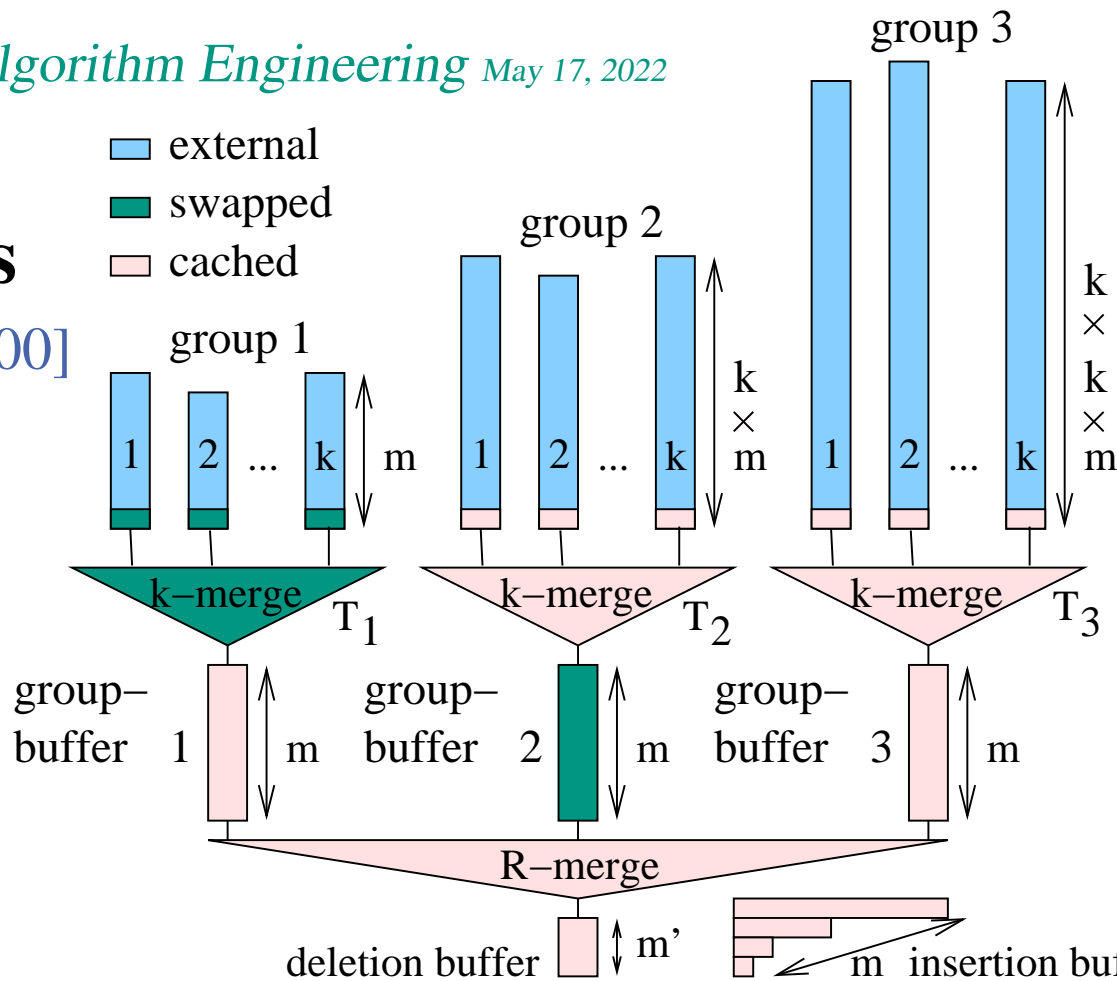
insert: $\approx m \log m$ alle m Ops. **Amortisiert** $\log m$

Insgesamt nur $\log km$ amortisiert !



Large Queues

[Sanders 00]



insert:

insert buffer full \longrightarrow merge ins-buf with del-buf·group-buf-1.

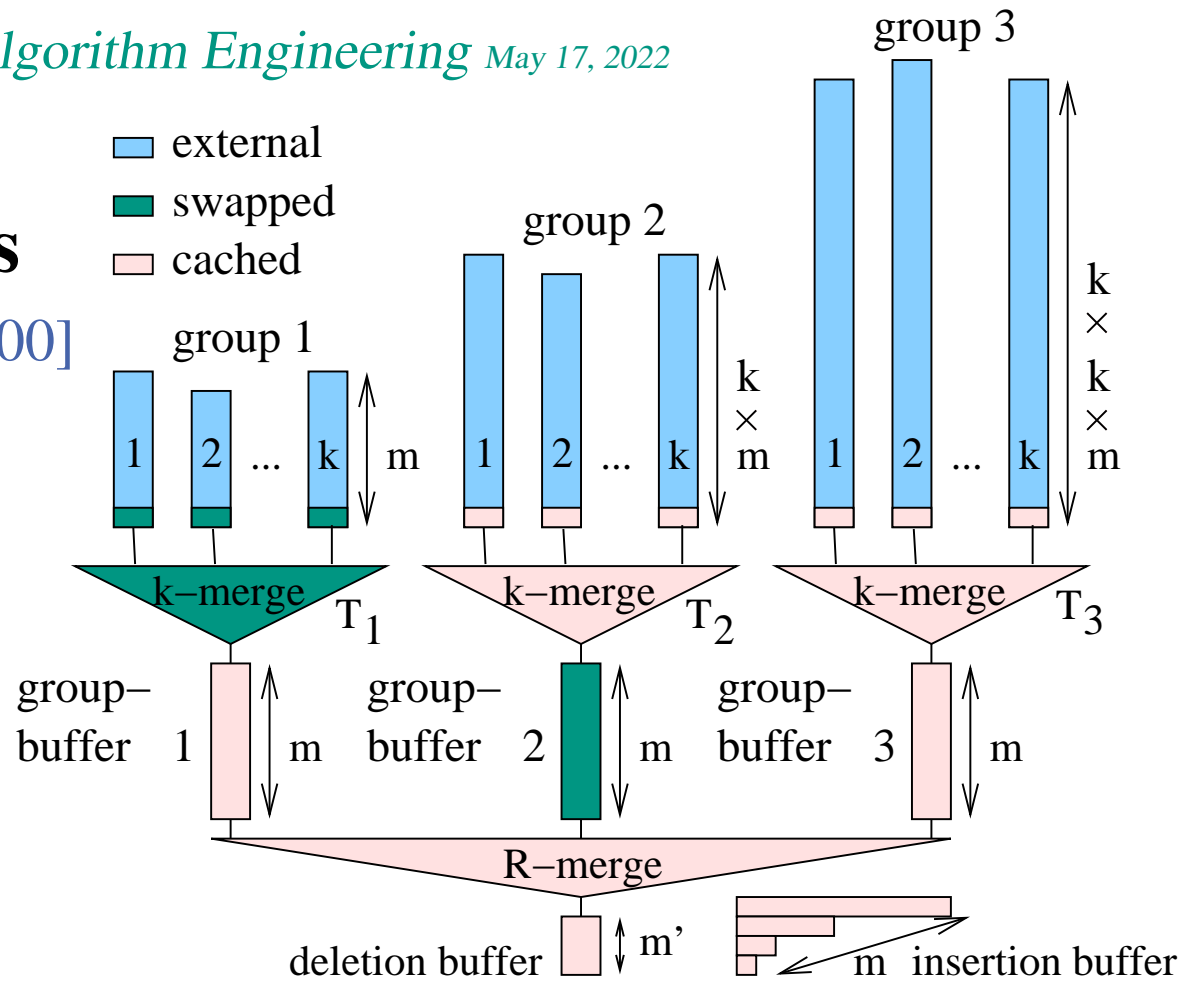
m' smallest into deletion buffer, next m into group buffer one, rest into group 1.

group full \longrightarrow merge group; shift into next group.

merge invalid group buffers and move them into group 1.

Large Queues

[Sanders 00]

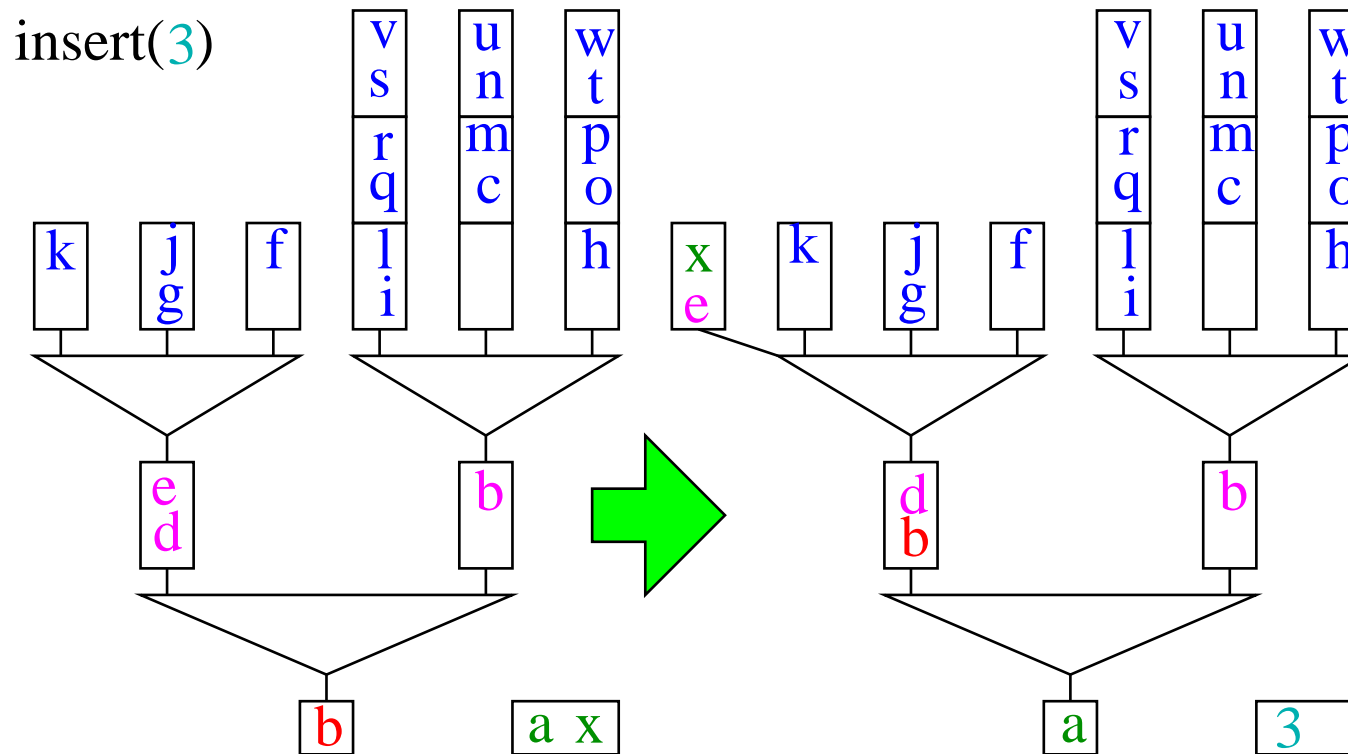


Delete-Min:

Refill. $m' \ll m$. nothing else

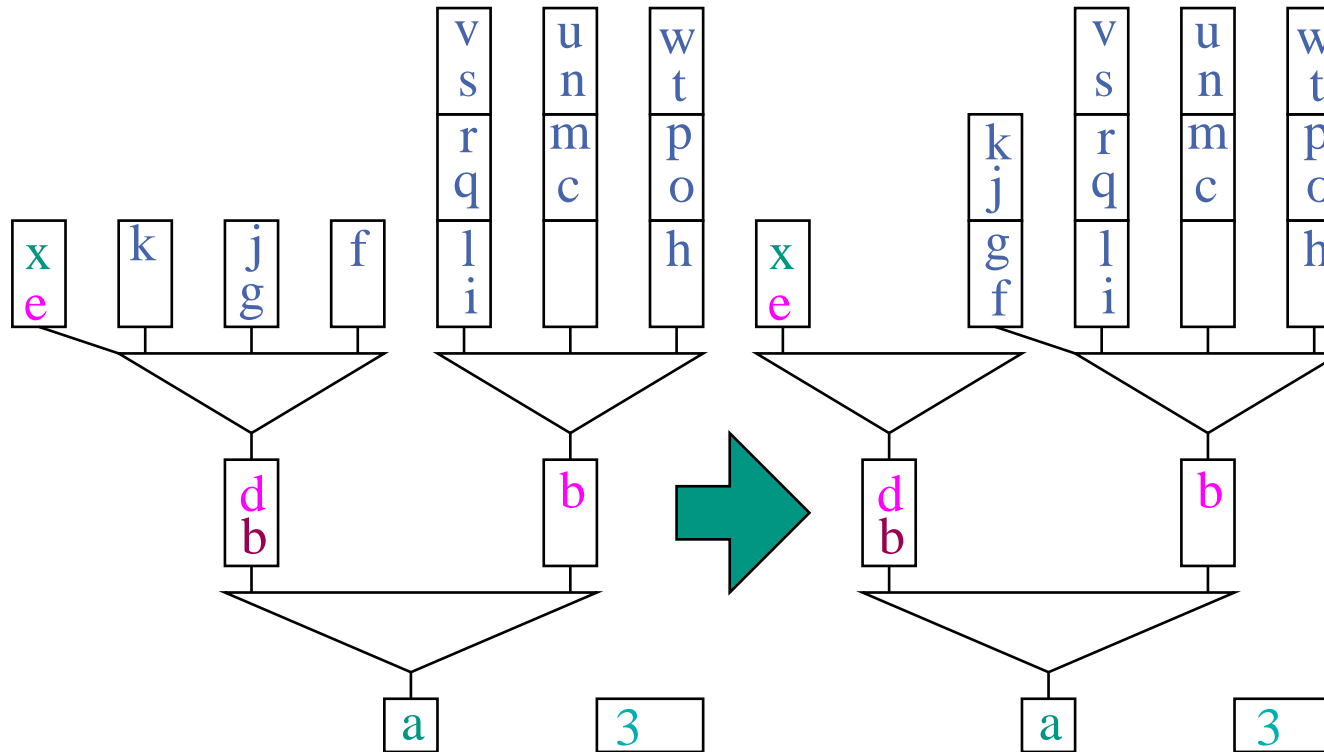
Example

Merge insertion buffer, deletion buffer, and leftmost group buffer



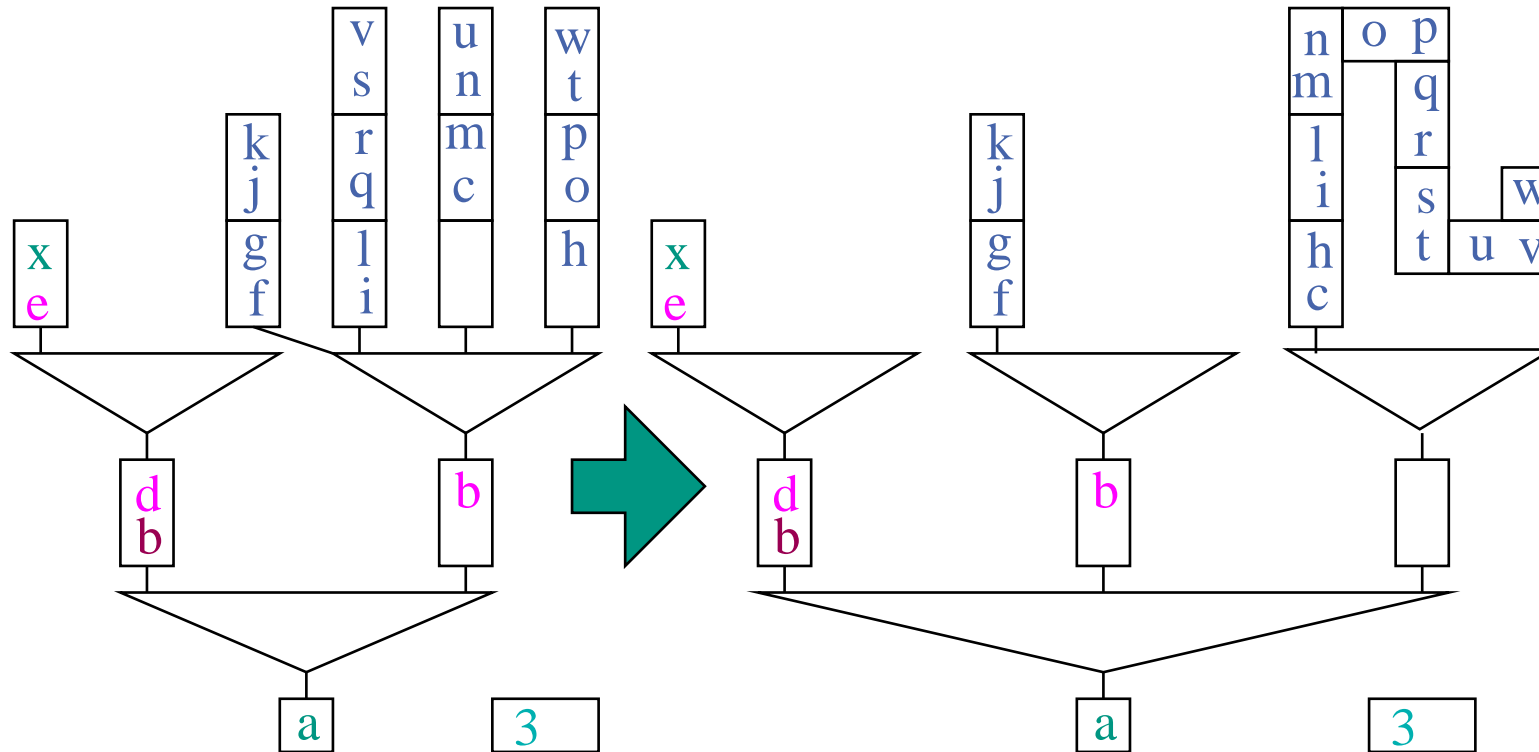
Example

Merge group 1



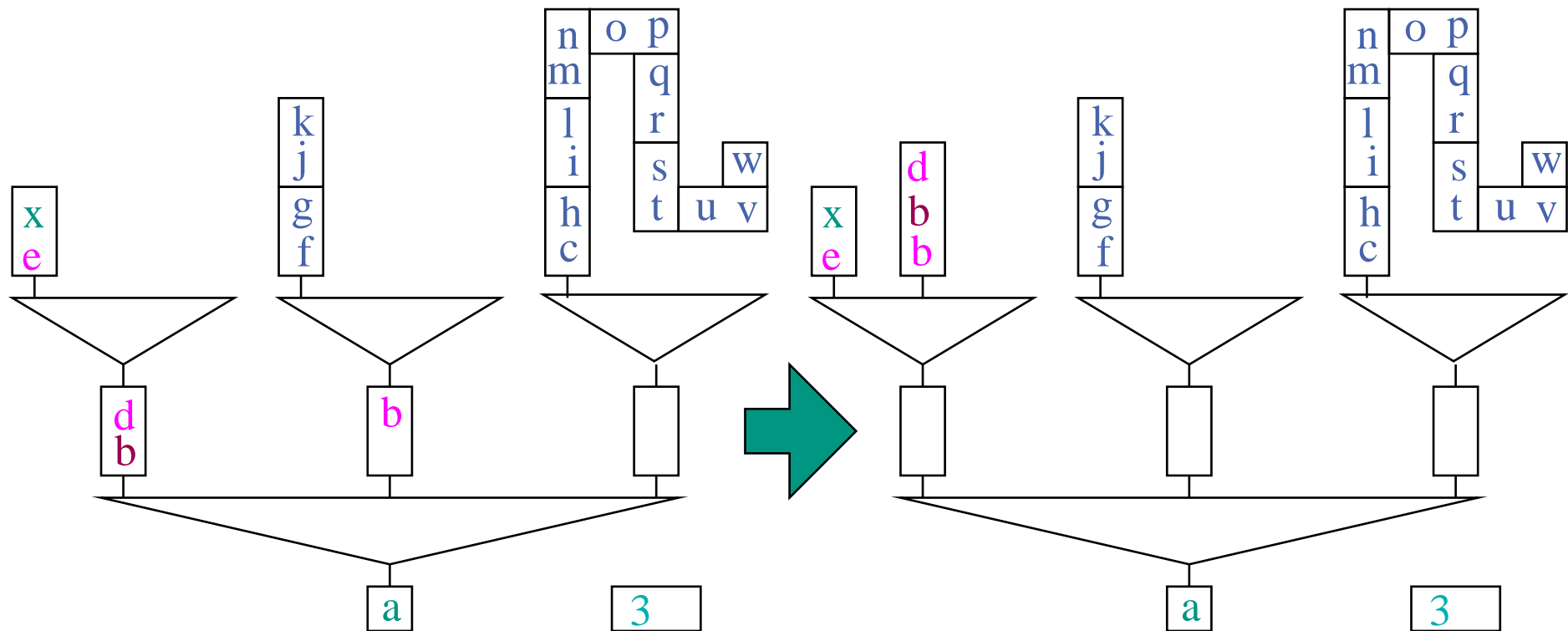
Example

Merge group 2



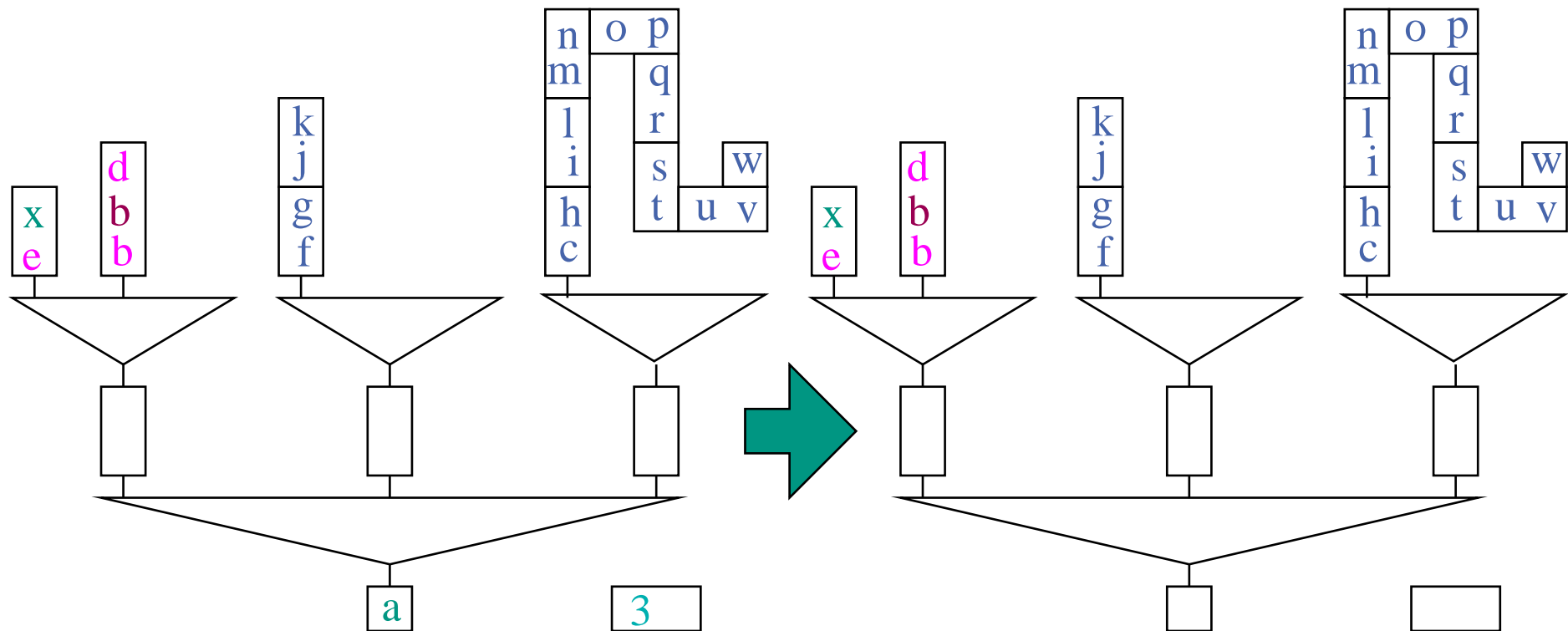
Example

Merge group buffers



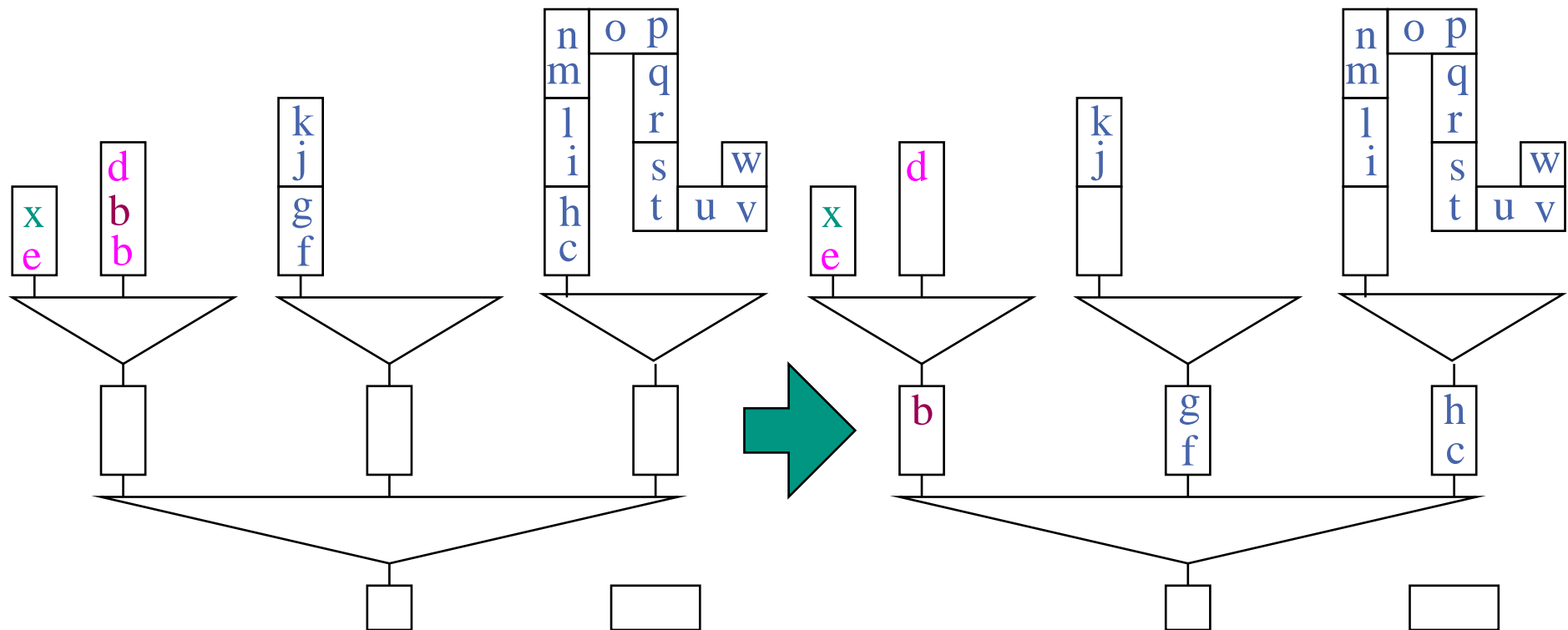
Example

DeleteMin \rightsquigarrow 3; DeleteMin \rightsquigarrow a;



Example

DeleteMin \rightsquigarrow b

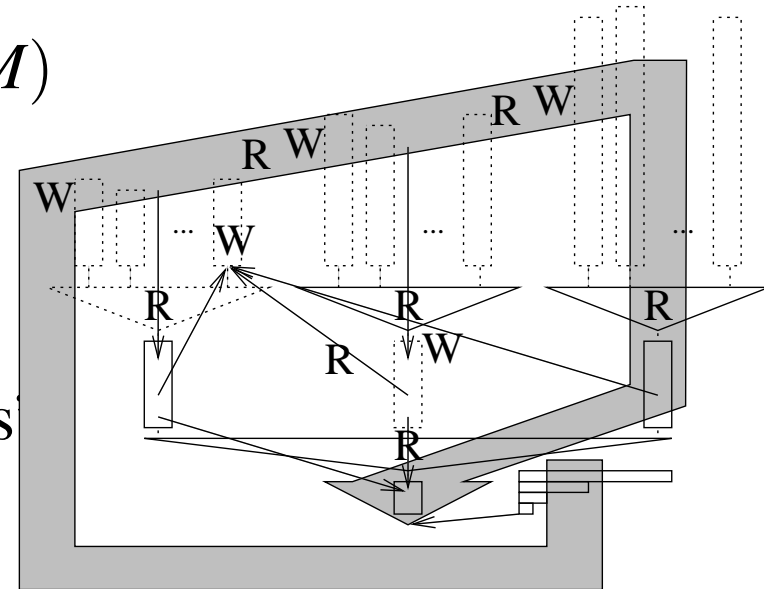


Analysis

- I insertions, buffer sizes $m = \Theta(M)$
- merging degree $k = \Theta(M/B)$

block accesses: $\text{sort}(I) + \text{“small terms”}$

key comparisons: $I \log I + \text{“small terms”}$
(on average)



Other (similar, earlier) [Arge 95, Brodal-Katajainen 98, Brengel et al. 99, Fadel et al. 97] data structures spend a factor ≥ 3 more I/Os to replace I by queue size.

Implementation Details

- Fast routines for 2–4 way merging keeping smallest elements in **registers**
- Use sentinels to avoid special case treatments (empty sequences, ...)
- Currently heap sort for sorting the insertion buffer
- $k \neq M/B$: multiple levels, limited associativity, TLB

Experiments

Keys: random 32 bit integers

Associated information: 32 dummy bits

Deletion buffer size: 32

Near optimal

Group buffer size: 256

: performance on

Merging degree k : 128

all machines tried!

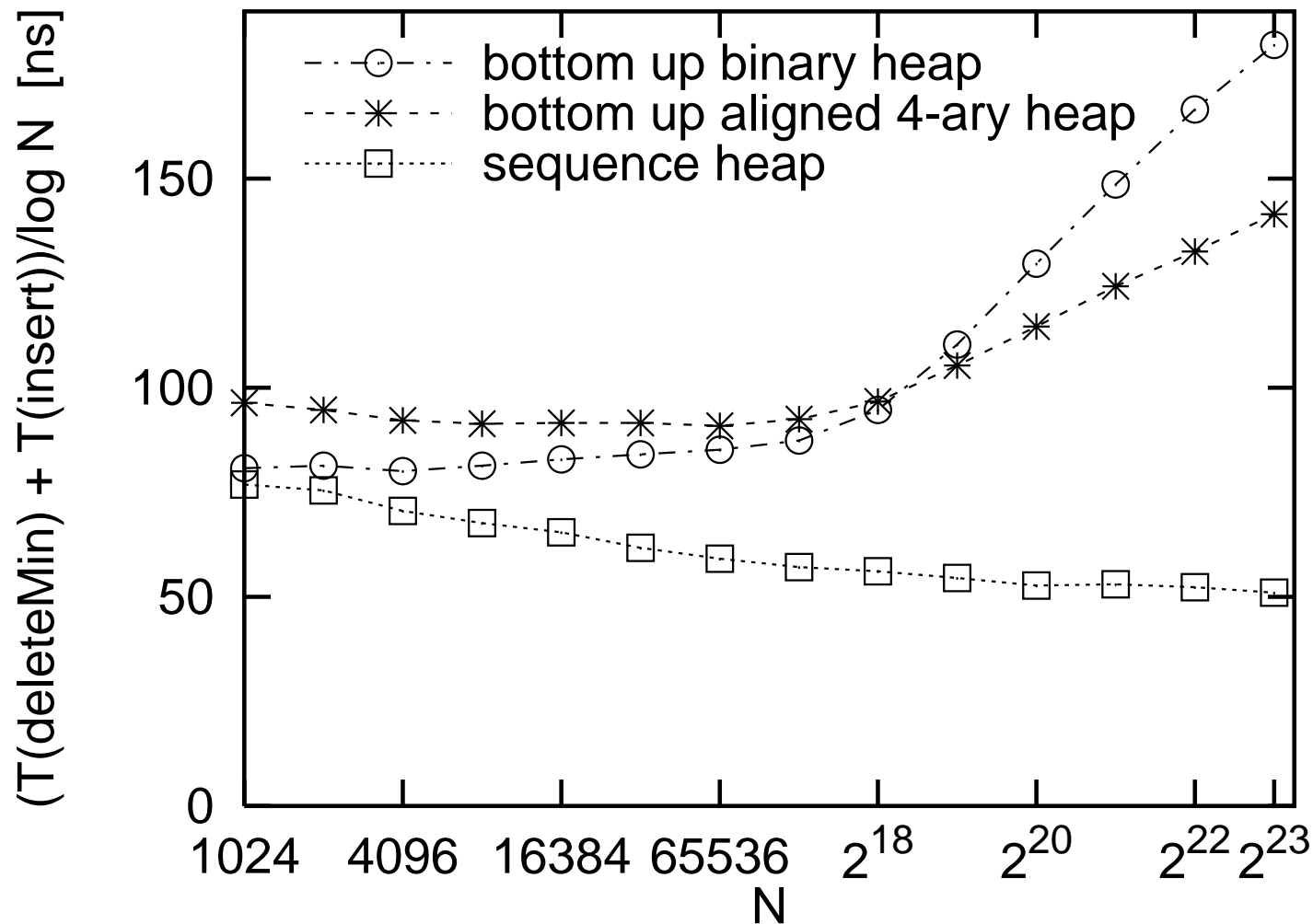
Compiler flags: Highly optimizing, nothing advanced

Operation Sequence:

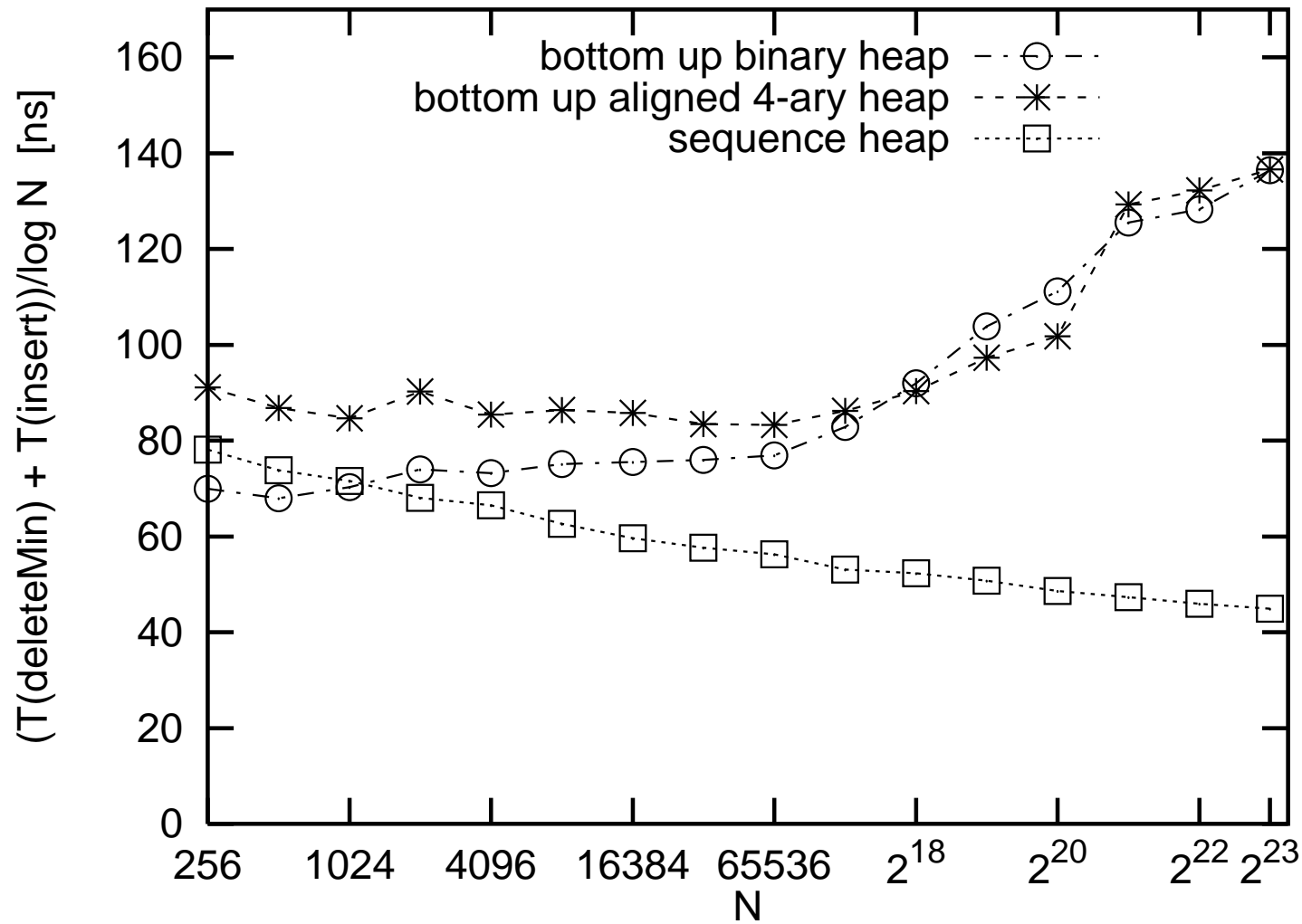
$(\text{Insert-DeleteMin-Insert})^N (\text{DeleteMin-Insert-DeleteMin})^N$

Near optimal performance on all machines tried!

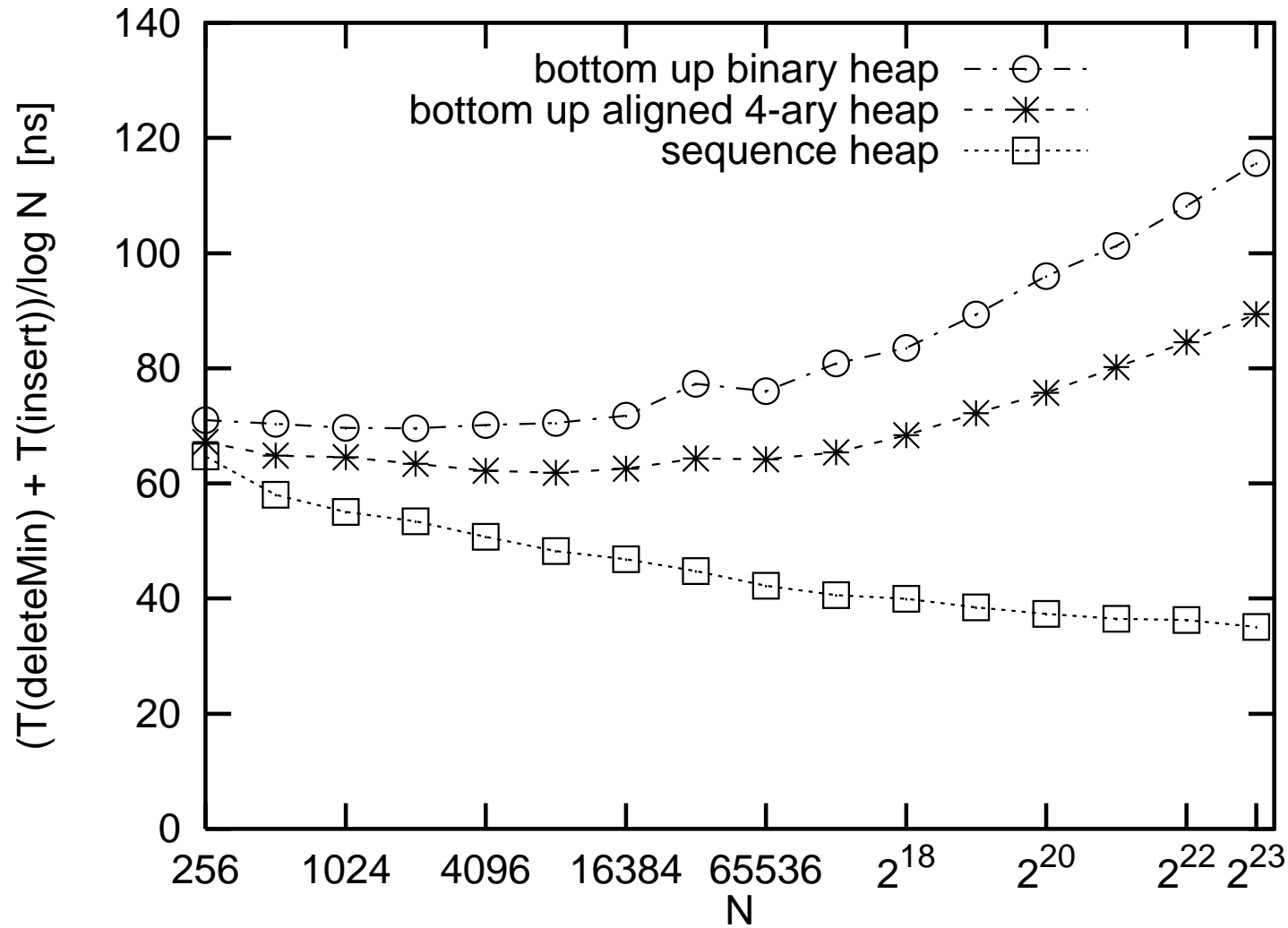
MIPS R10000, 180 MHz



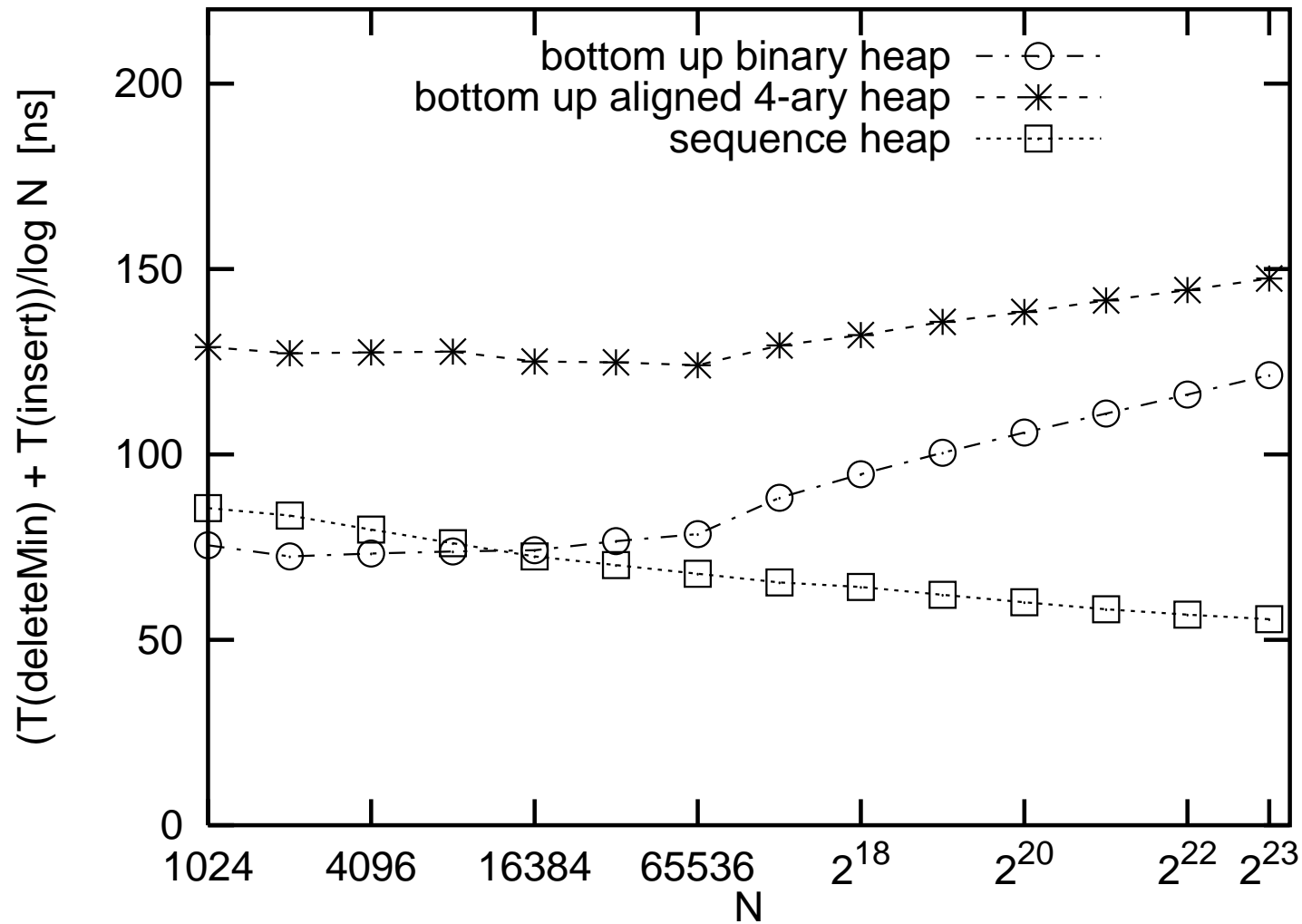
Ultra-SparcIIi, 300 MHz



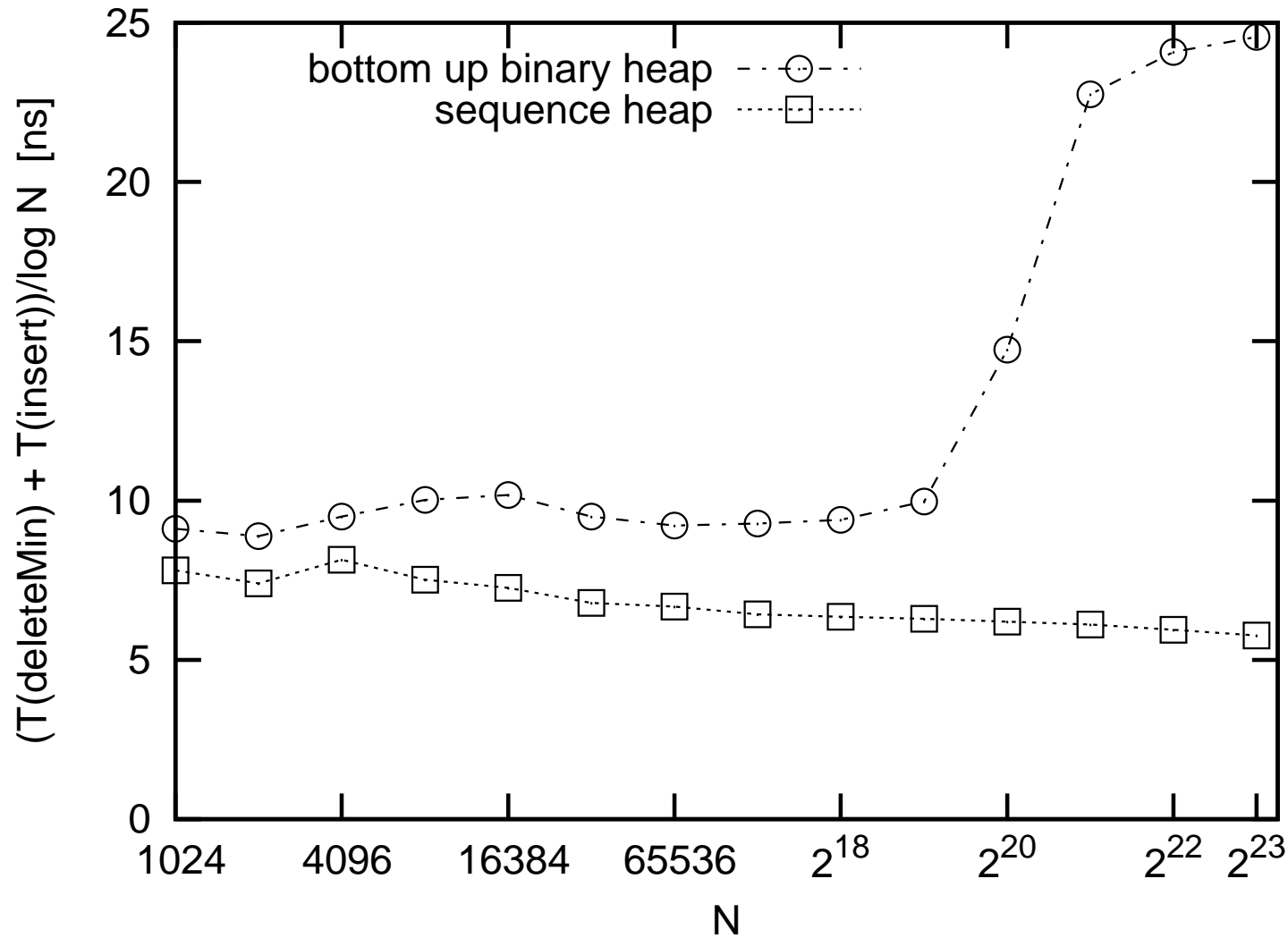
Alpha-21164, 533 MHz



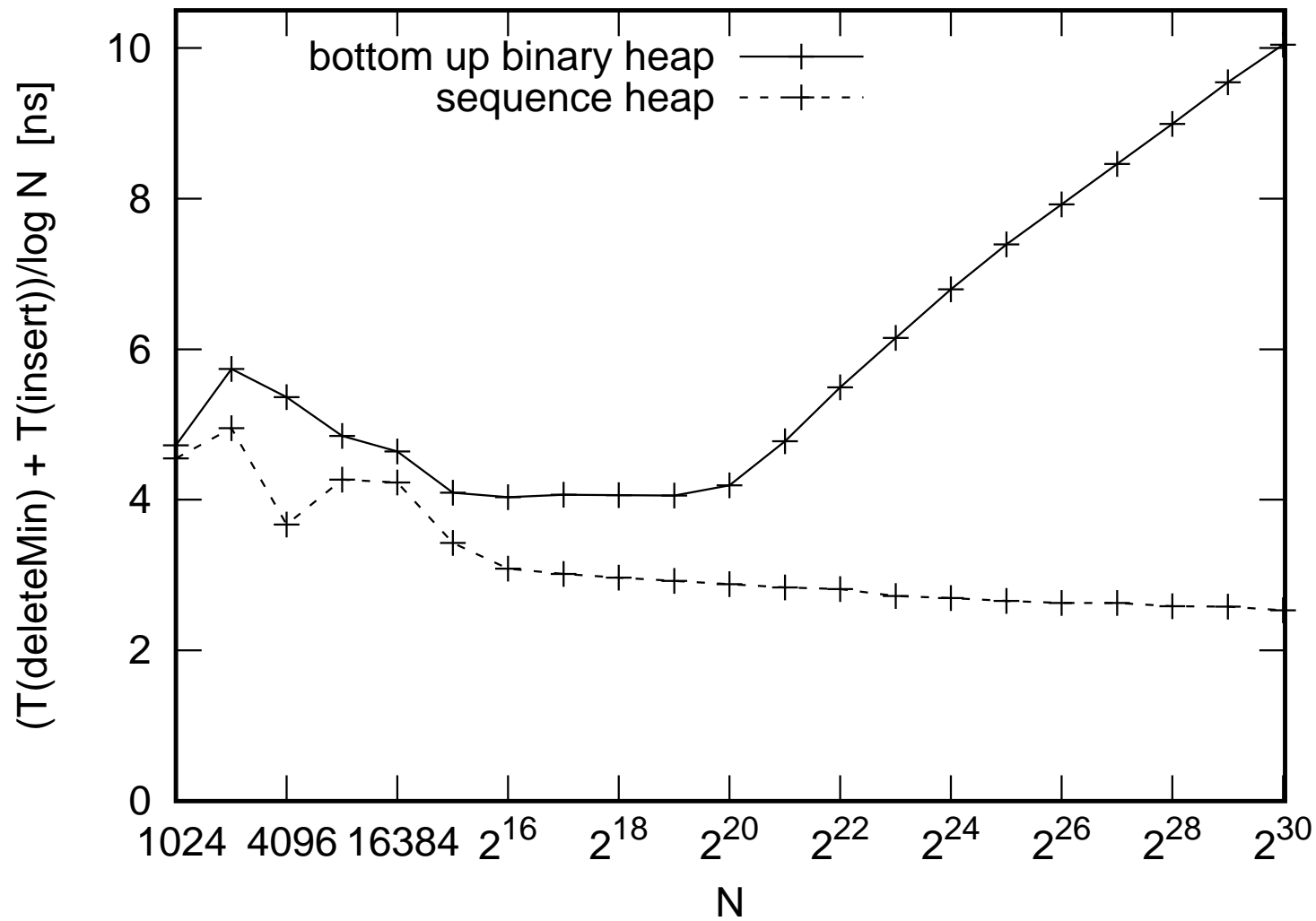
Pentium II, 300 MHz



Core2 Duo Notebook, 1.??? GHz

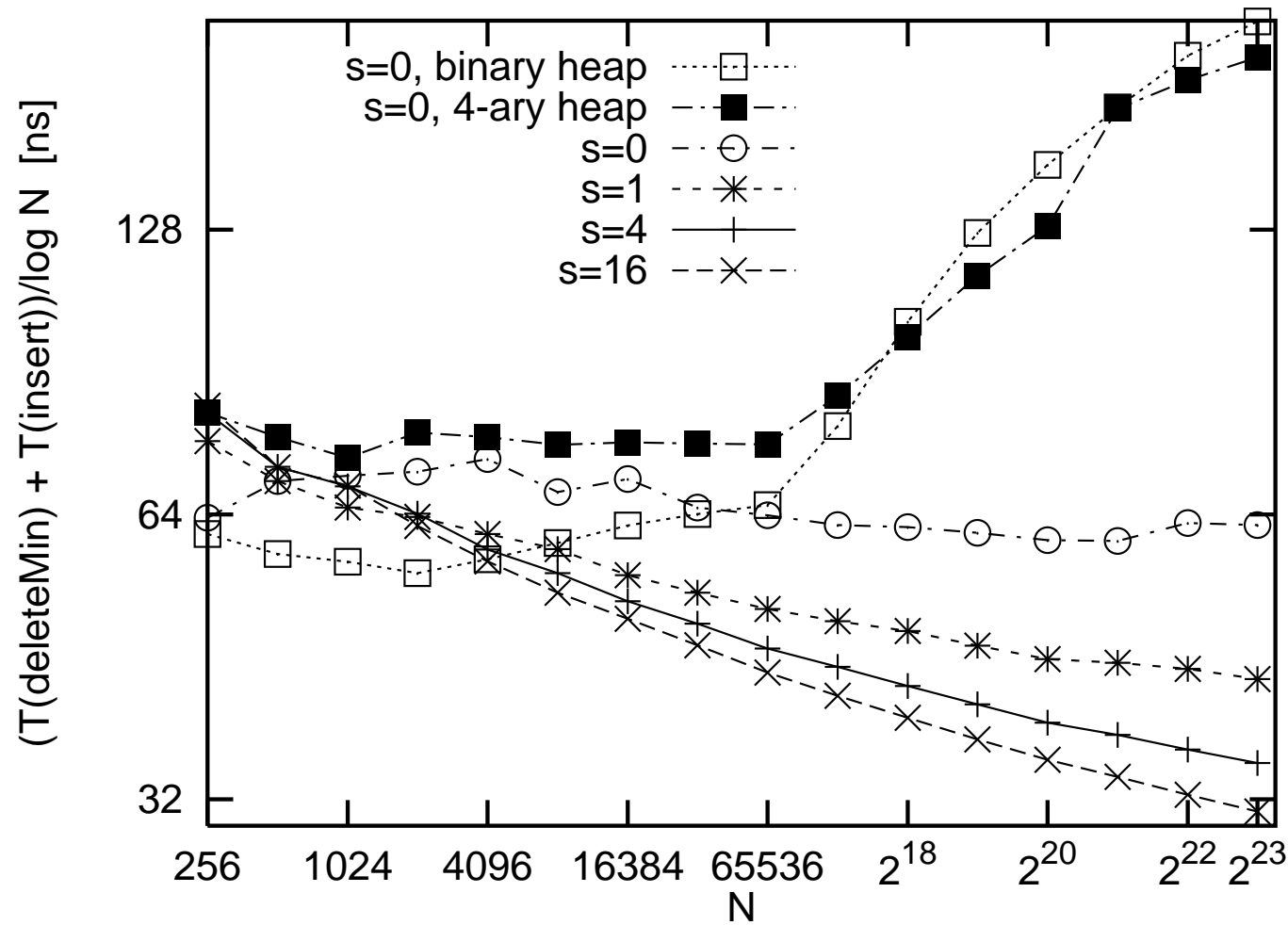


AMD Ryzen 1800X, 16MB L3, 3.6 GHz, 2017



$$(\text{insert} (\text{deleteMin insert})^s)^N$$

$$(\text{deleteMin} (\text{insert deleteMin})^s)^N$$



Methodological Lessons

If you want to compare **small** constant factors in **execution time**:

- **Reproducibility** demands **publication of source codes**
(4-ary heaps, old study in Pascal)

- Highly **tuned codes in particular** for the competitors
(binary heaps have factor 2 between good and naive implementation).

How do you compare two mediocre implementations?

- Careful choice/description of **inputs**

- Use multiple different hardware **platforms**

- Augment with **theory** (e.g., comparisons, data dependencies, cache faults, locality effects ...)

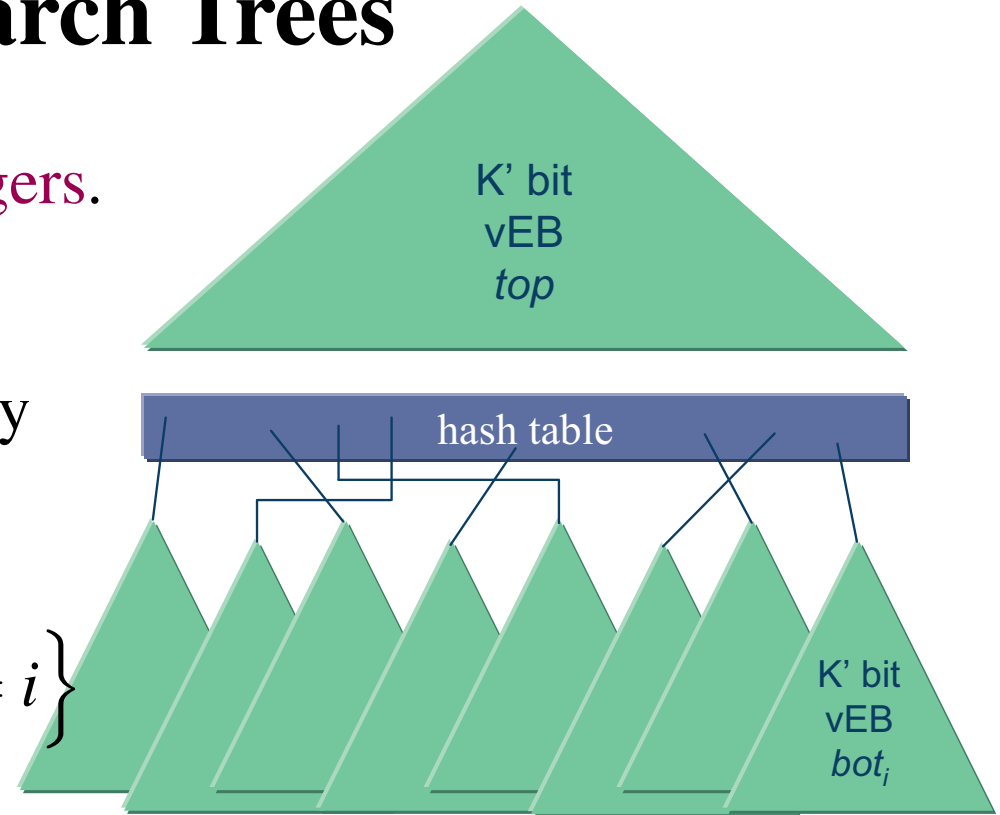
Open Problems

- Dependence on **size** rather than number of insertions
- Parallel disks**
- Space efficient** implementation
- Multi-level** cache aware or cache-oblivious variants
- Eliminate **branch mispredictions**

new: Master thesis v. d. Grün: did that vor insertion buffer,
first results on PQs based on distribution principle and the
inner loop of super scalar sample sort

3 van Emde-Boas Search Trees

- Store set M of $K = 2^k$ -bit integers.
later: associated information
- $K = 1$ or $|M| = 1$: store directly
- $K' := K/2$
- $M_i := \{x \bmod 2^{K'} : x \operatorname{div} 2^{K'} = i\}$
- **root** points to nonempty M_i -s
- **top** $t = \{i : M_i \neq \emptyset\}$
- insert, delete, search in $\mathcal{O}(\log K)$ time



Locate

// $\min x \in M : y \leq x$

Function locate($y : \mathbb{N}$) : ElementHandle

if $y > \max M$ **then return** ∞

// precomputed!

if $K = 1$ **then return** locateLocally(y)

if $M = \{x\}$ **then return** x

$(i, j) := (y \text{ div } 2^{K/2}, y \text{ mod } 2^{K/2})$

if $M_i = \emptyset \vee j > \max M_i$ **then**

$i := \text{top.locate}(i + 1)$

$j := \min M_i$

// precomputed!

else $j := M_i.\text{locate}(j)$

return $i2^{K/2} + j$

Comparison with Comparison Based Search Trees

Ideally: $\log n \rightsquigarrow \log \log n$

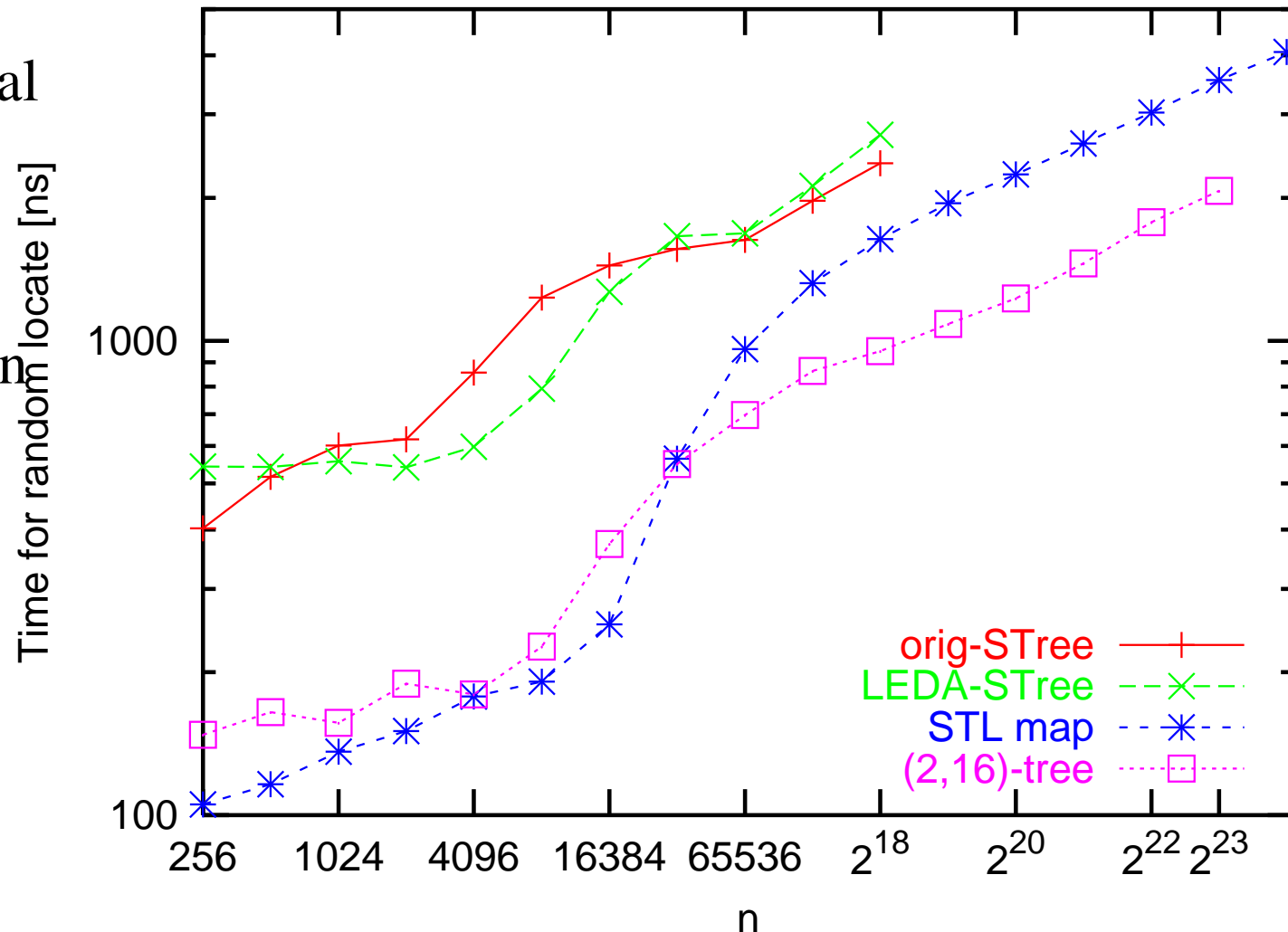
Problems:

Many special

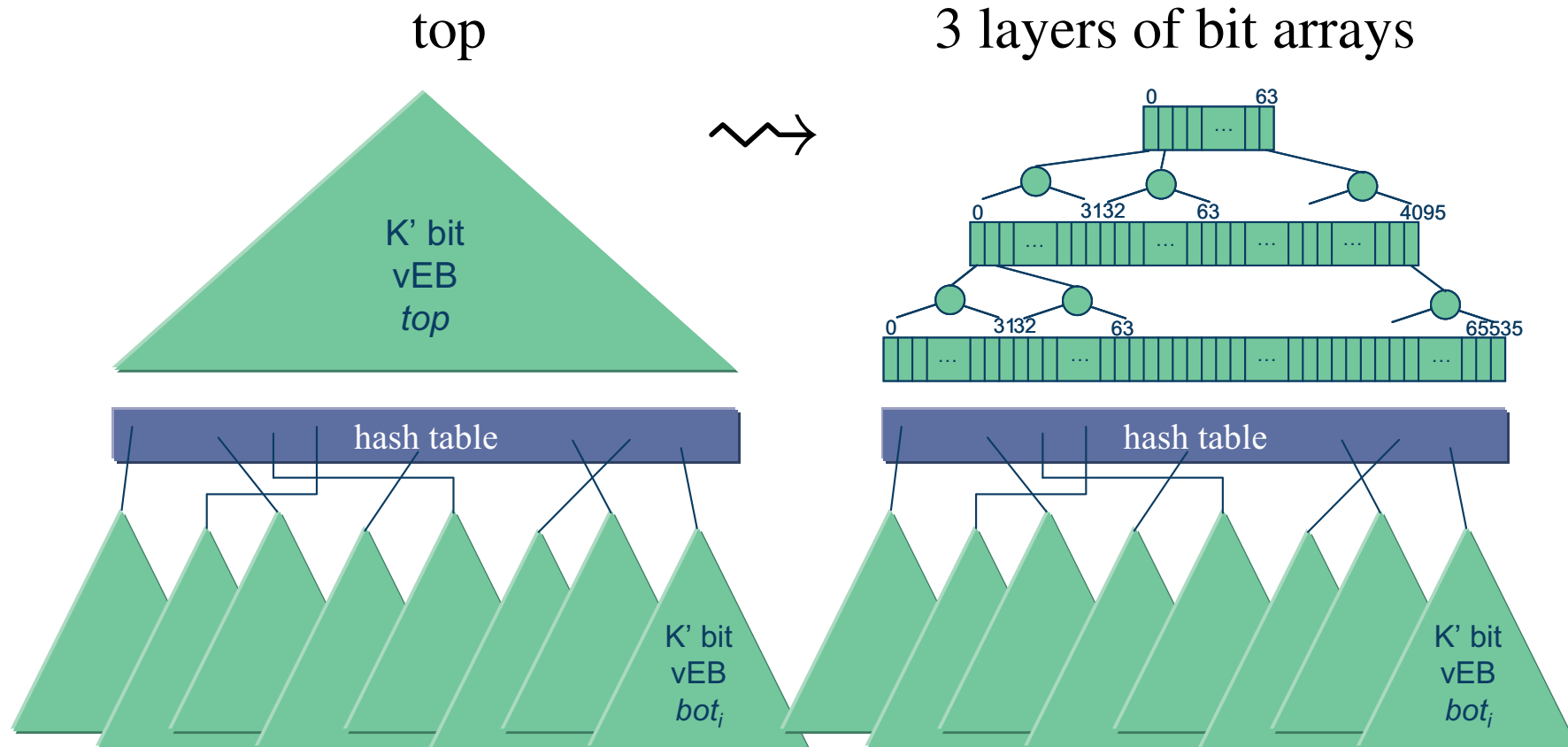
case tests

High space

consumption



Efficient 32 bit Implementation



Layers of Bit Arrays

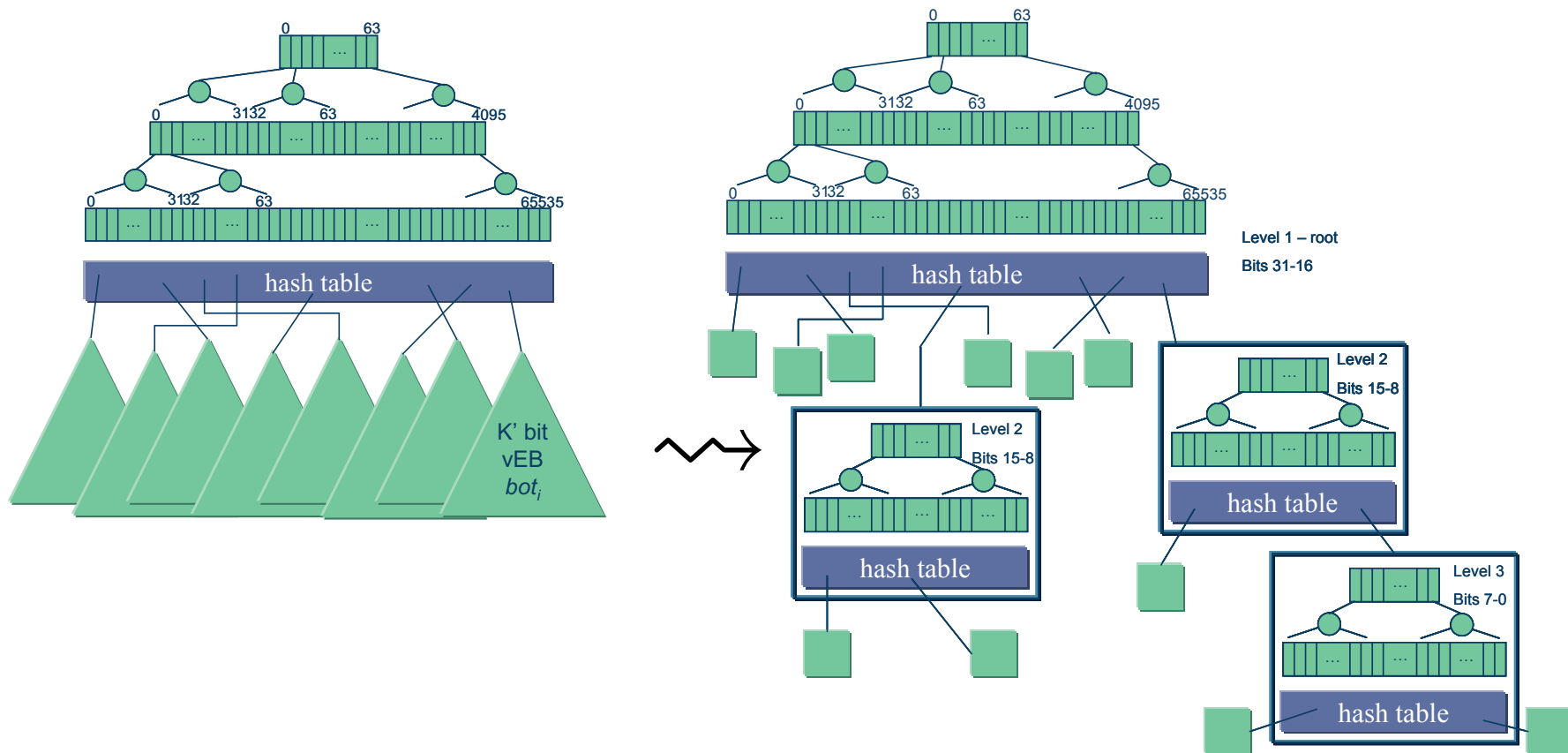
$$t^1[i] = 1 \text{ iff } M_i \neq \emptyset$$

$$t^2[i] = t^1[32i] \vee t^1[32i + 1] \vee \dots \vee t^1[32i + 31]$$

$$t^3[i] = t^2[32i] \vee t^2[32i + 1] \vee \dots \vee t^2[32i + 31]$$

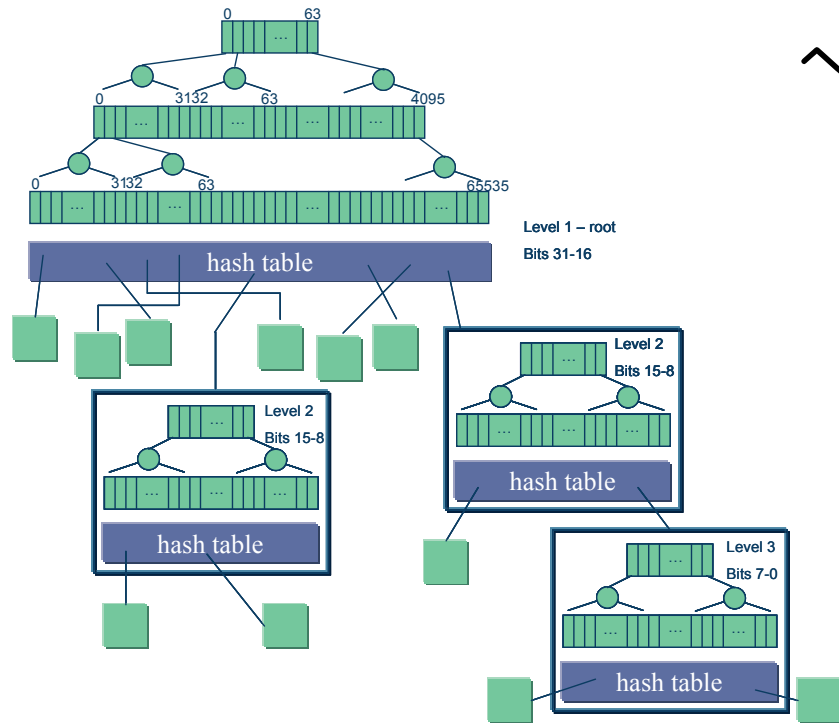
Efficient 32 bit Implementation

Break recursion after 3 layers

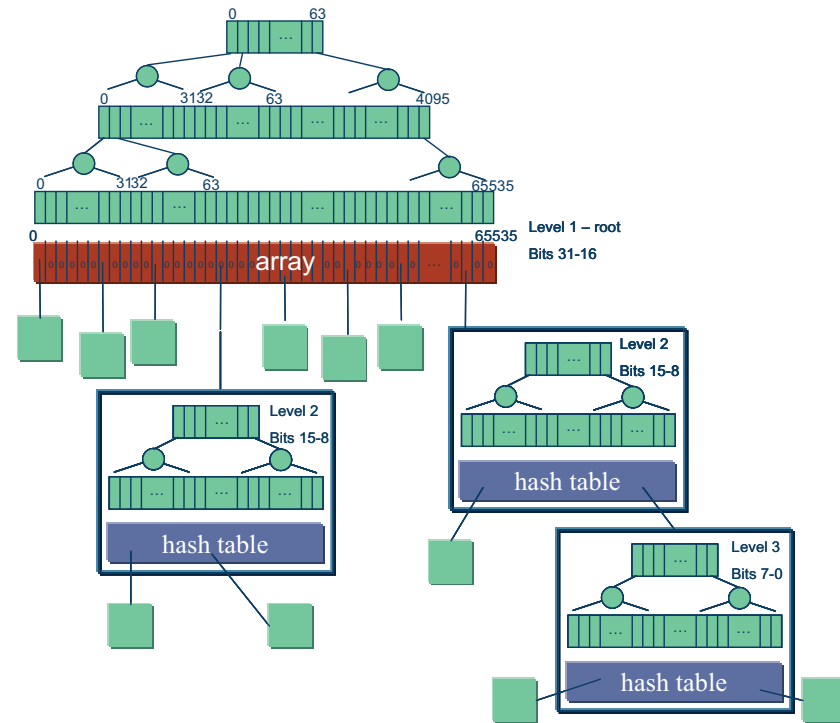


Efficient 32 bit Implementation

root hash table



root array



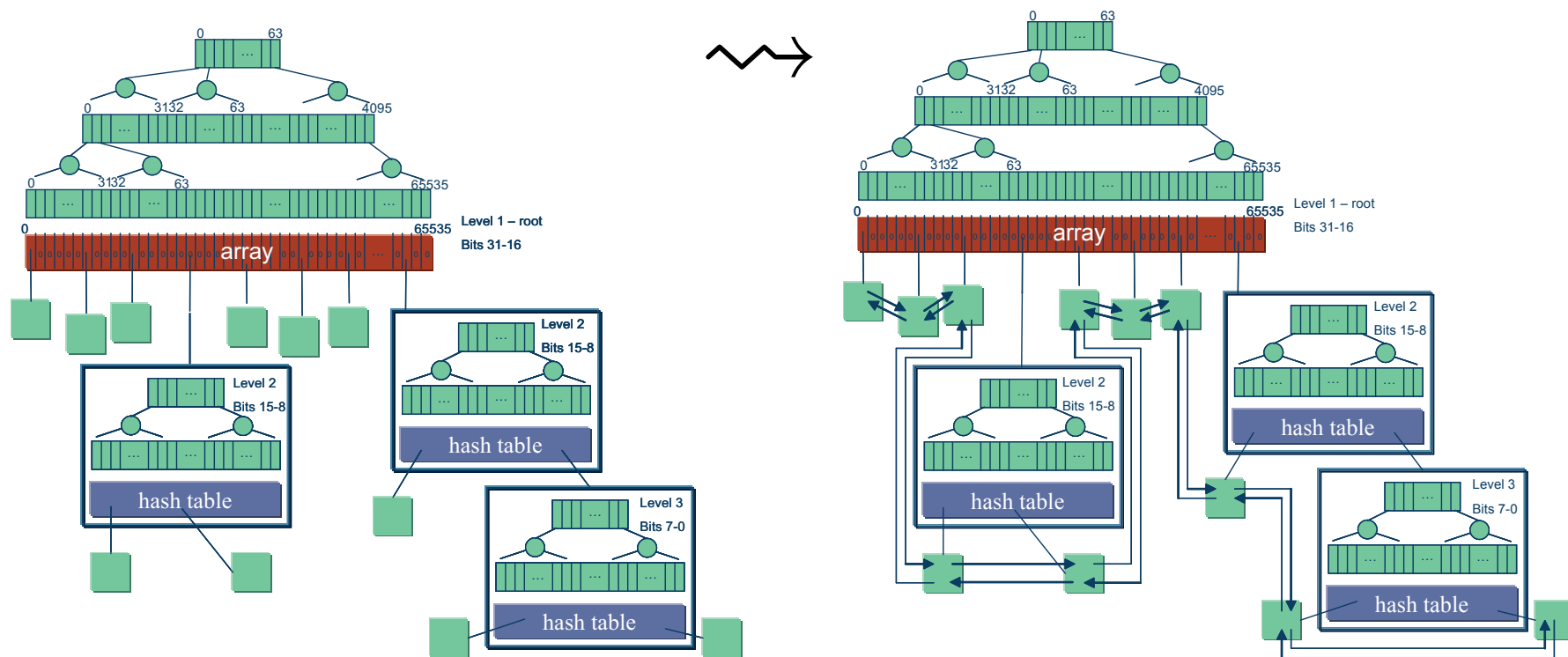
Efficient 32 bit Implementation

Tuned small hash tables with 8-bit keys:

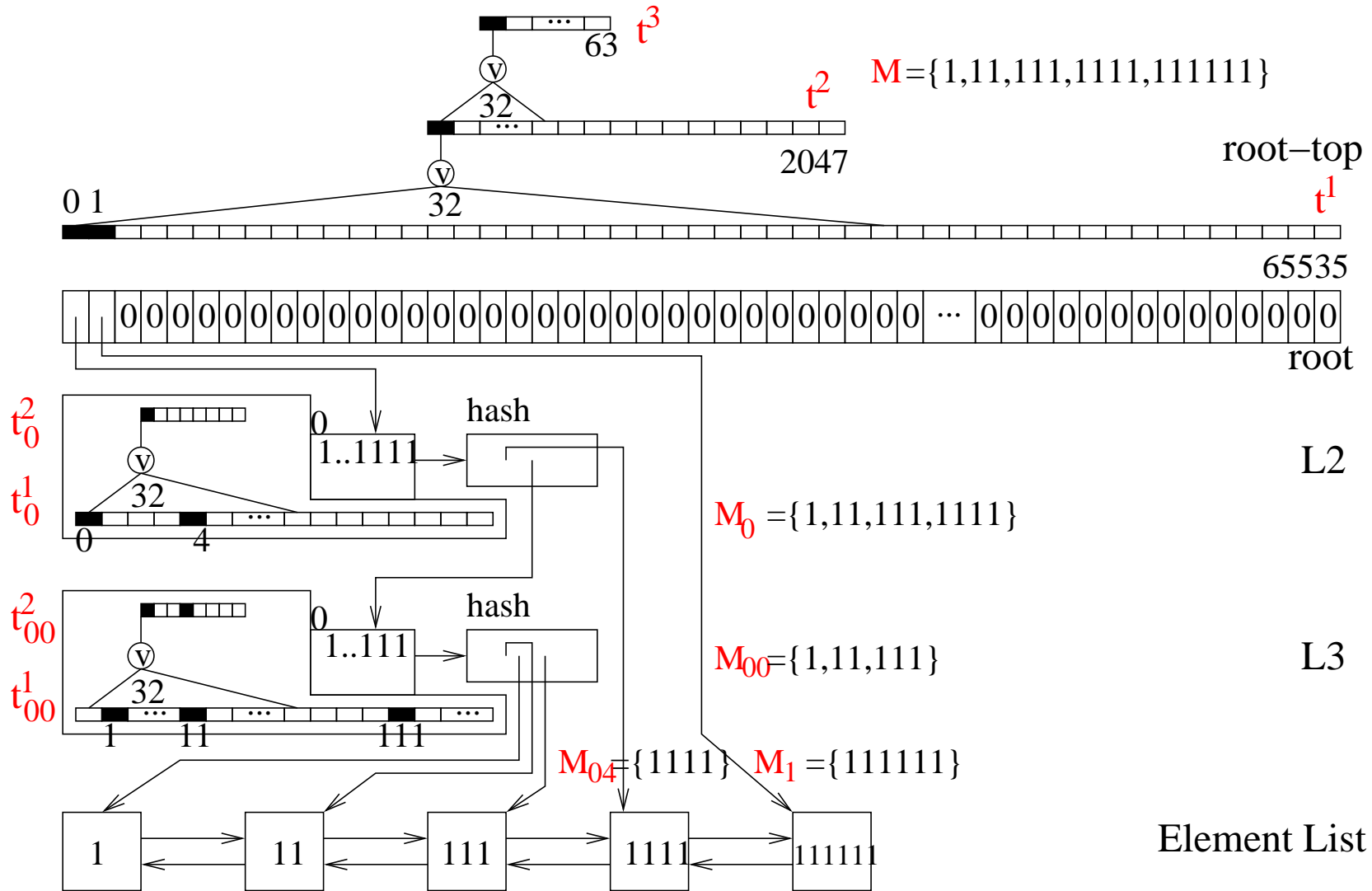
- Tabulate hash function (256 entries)
→ very fast
- Make it a random permutation
→ reduces collisions

Efficient 32 bit Implementation

Sorted doubly linked lists for **associated information** and **range queries**



Example



Locate High Level

//return handle of $\min x \in M : y \leq x$

Function `locate`($y : \mathbb{N}$) : ElementHandle

if $y > \max M$ **then return** ∞

$i := y[16..31]$ // Level 1

if $r[i] = \text{nil} \vee y > \max M_i$ **then return** $\min M_{t^1}.\text{locate}(i+1)$

if $M_i = \{x\}$ **then return** x

$j := y[8..15]$ // Level 2

if $r_i[j] = \text{nil} \vee y > \max M_{ij}$ **then return** $\min M_{i,t_i^1}.\text{locate}(j+1)$

if $M_{ij} = \{x\}$ **then return** x

return $r_{ij}[t_{ij}^1.\text{locate}(y[0..7])]$ // Level 3

Locate in Bit Arrays

//find the smallest $j \geq i$ such that $t^k[j] = 1$

Method `locate`(i) for a bit array t^k consisting of n bit words

// $n = 32$ for $t^1, t^2, t_i^1, t_{ij}^1$; $n = 64$ for t^3 ; $n = 8$ for t_i^2, t_{ij}^2

assert some bit in t^k to the right of i is nonzero

$j := i \text{ div } n$ // which word?

$a := t^k[nj..nj + n - 1]$

set $a[(i \bmod n) + 1..n - 1]$ to zero // $n - 1 \dots i \bmod n \dots 0$

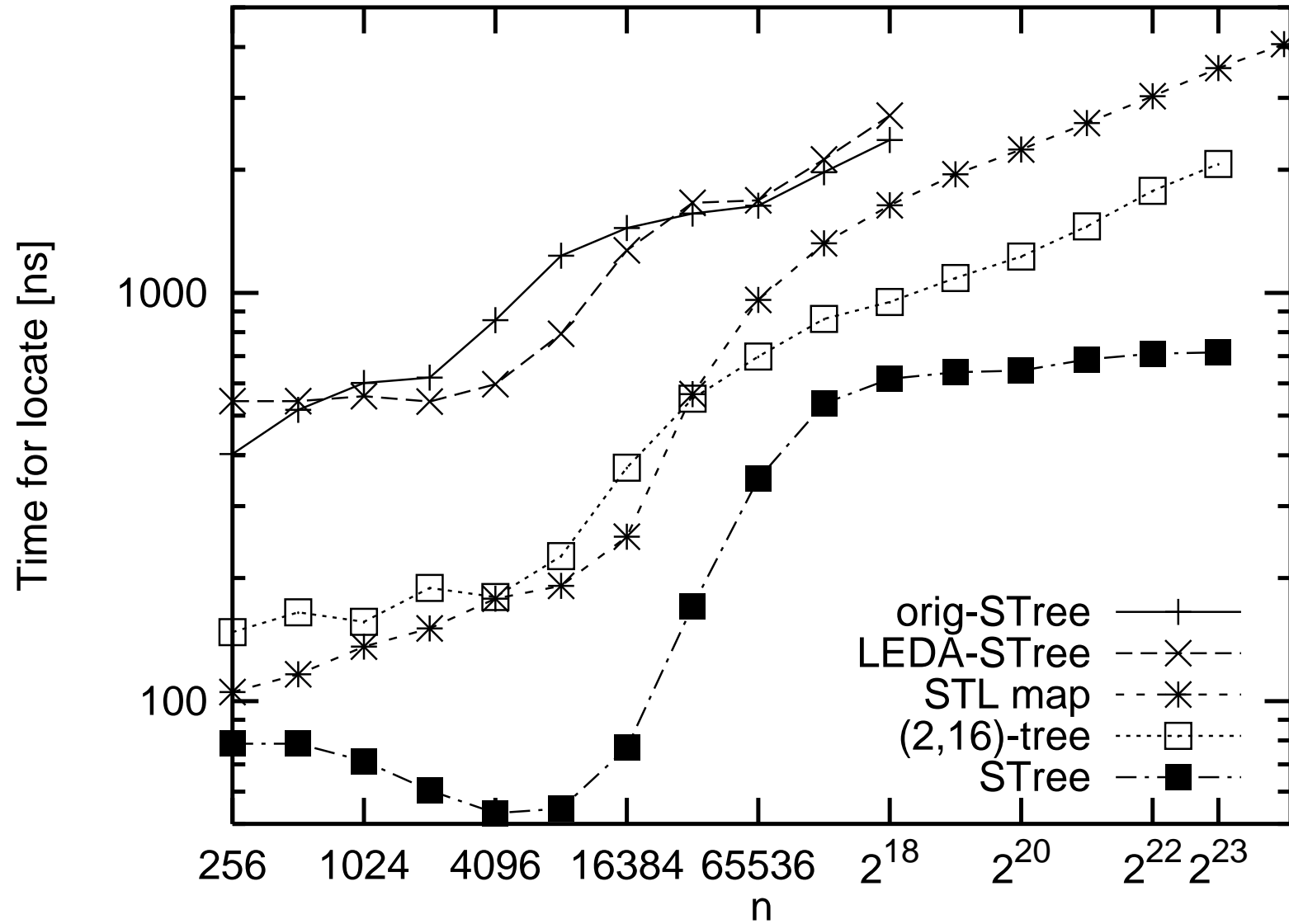
if $a = 0$ **then**

$j := t^{k+1}.\text{locate}(j)$

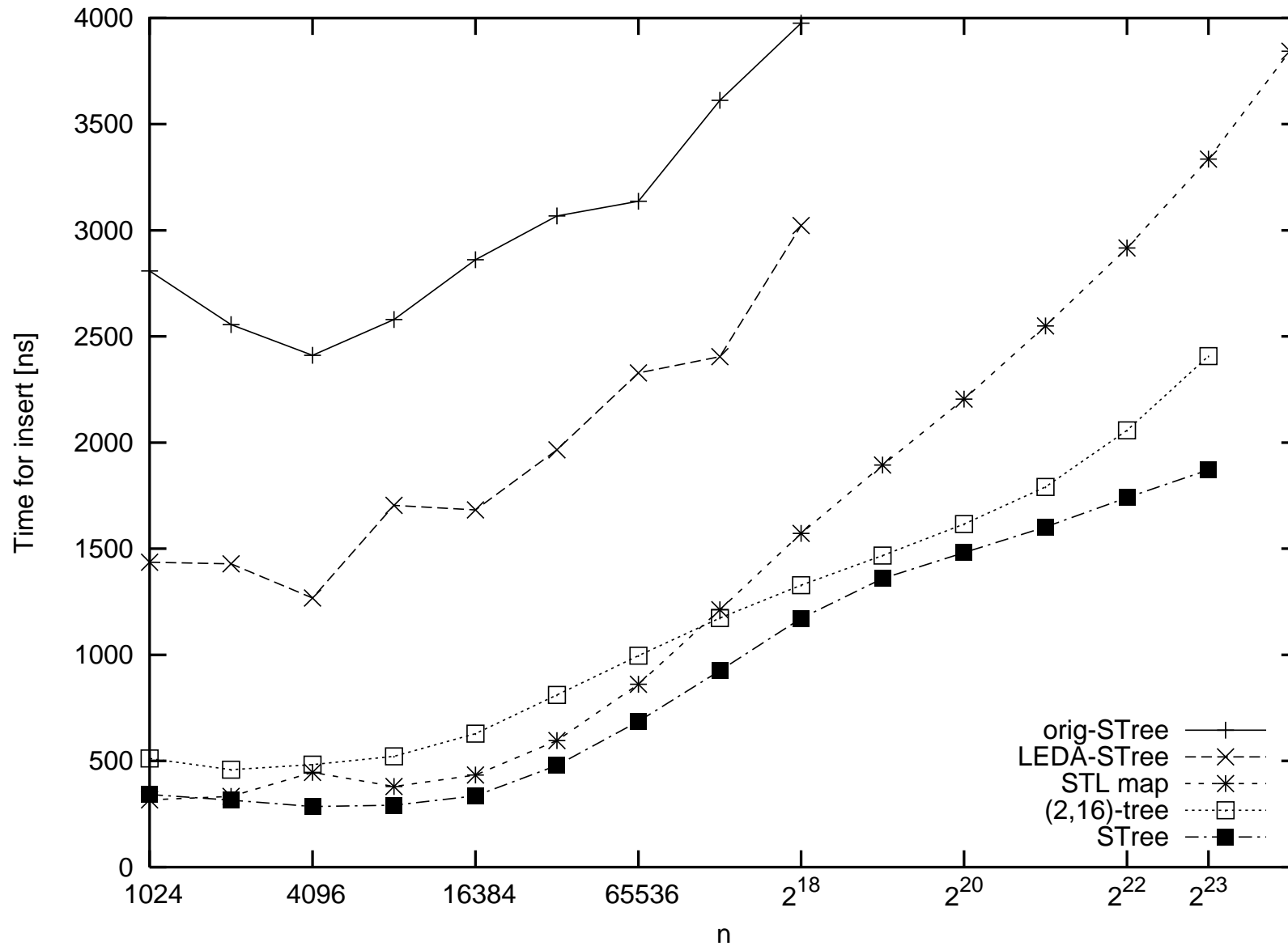
$a := t^k[nj..nj + n - 1]$

return $nj + \text{msbPos}(a)$ // e.g. floating point conversion

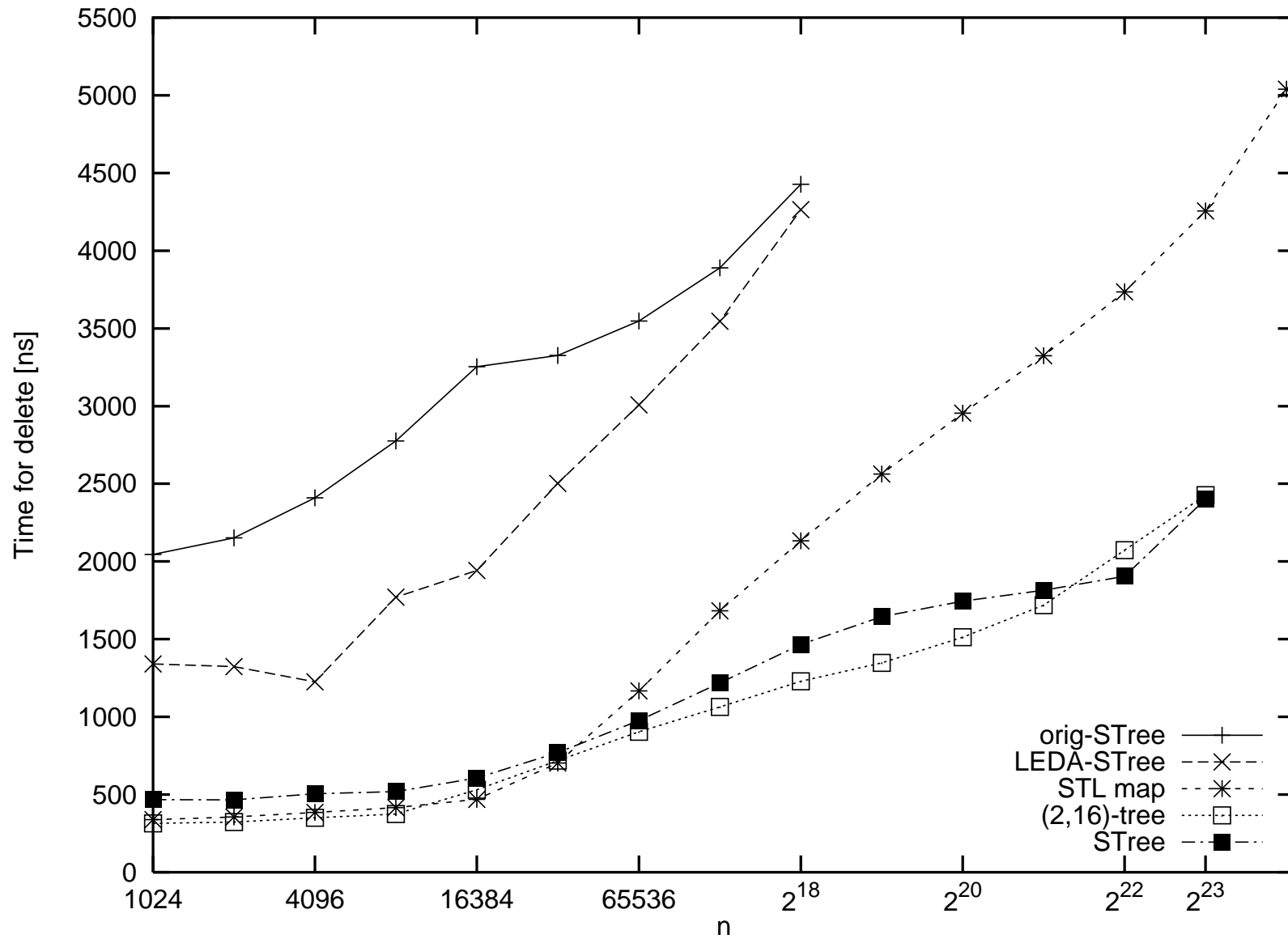
Random Locate



Random Insert



Delete Random Elements



Open Problems

- Measurement for “**worst case**” inputs
- Measure Performance for **realistic inputs**
 - **IP lookup** etc.
 - **Best first** heuristics like, e.g., bin packing
- More **space efficient** implementation
- (A few) **more bits**

4 Hashing

“to **hash**” \approx “to bring into complete **disorder**”

paradoxically, this helps us to find things more **easily**!

store set $M \subseteq \text{Element}$.

key(e) is unique for $e \in M$.

support **dictionary** operations in $\mathcal{O}(1)$ time:



$M.\text{insert}(e : \text{Element})$: $M := M \cup \{e\}$

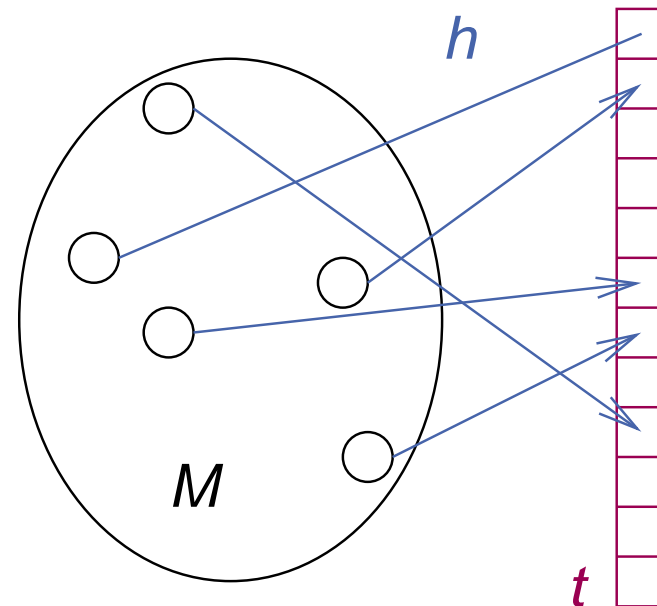
$M.\text{remove}(k : \text{Key})$: $M := M \setminus \{e\}, e = k$

$M.\text{find}(k : \text{Key})$: return $e \in M$ with $e = k$; \perp if none present

(Convention: key is implicit), e.g. $e = k$ iff $\text{key}(e) = k$)

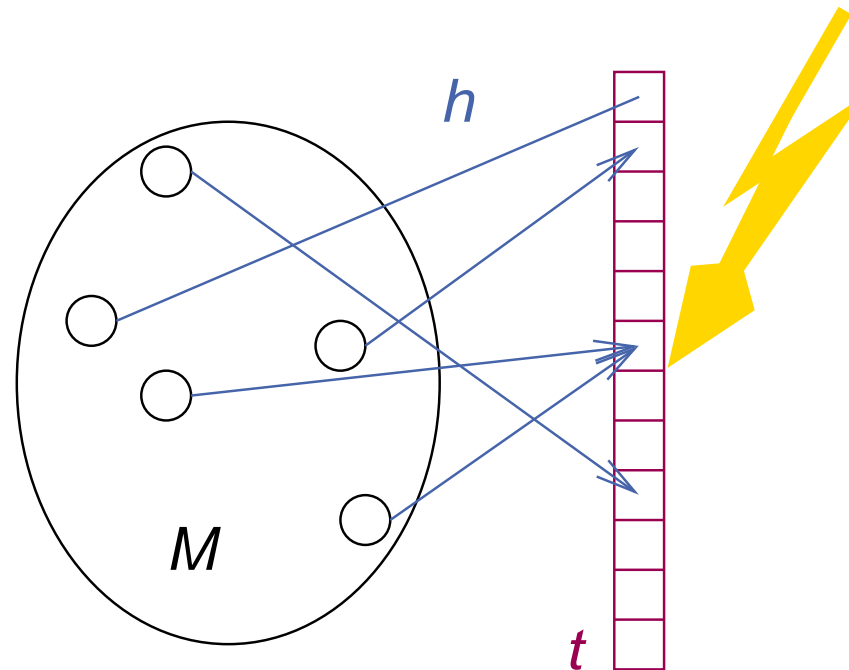
An (Over)optimistic approach

A (perfect) hash function h
maps elements of M to
unique entries of table $t[0..m-1]$, i.e.,
 $t[h(\text{key}(e))] = e$



Collisions

perfect hash functions are difficult to obtain

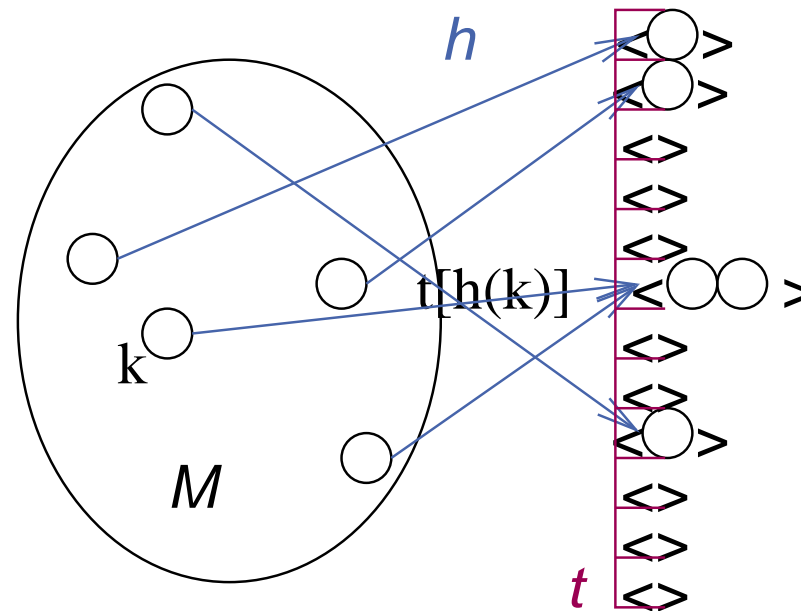


Example: Birthday Paradoxon

Collision Resolution

for example by **closed hashing**

entries: elements \rightsquigarrow **sequences** of elements



Hashing with Chaining

Implement sequences in closed hashing by singly linked lists

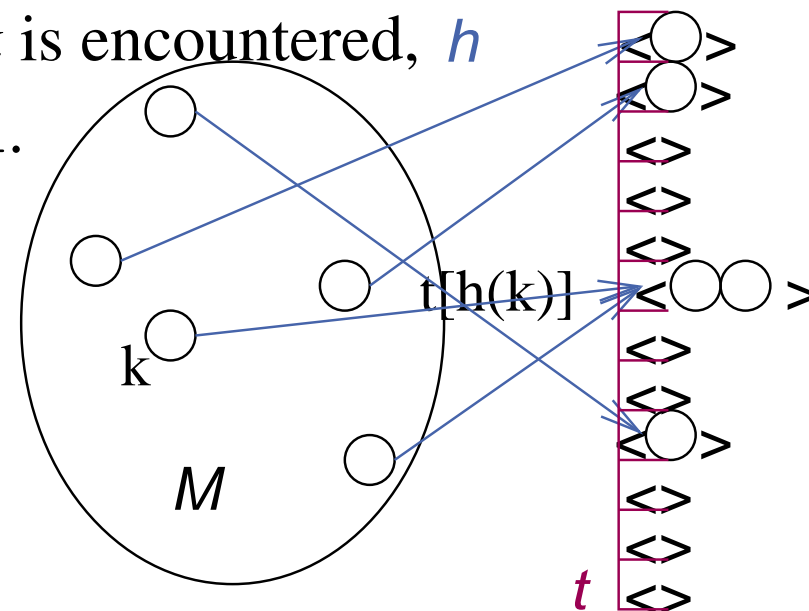
$\text{insert}(e)$: Insert e at the beginning of $t[h(e)]$. **constant time**

$\text{remove}(k)$: Scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, remove it and return.

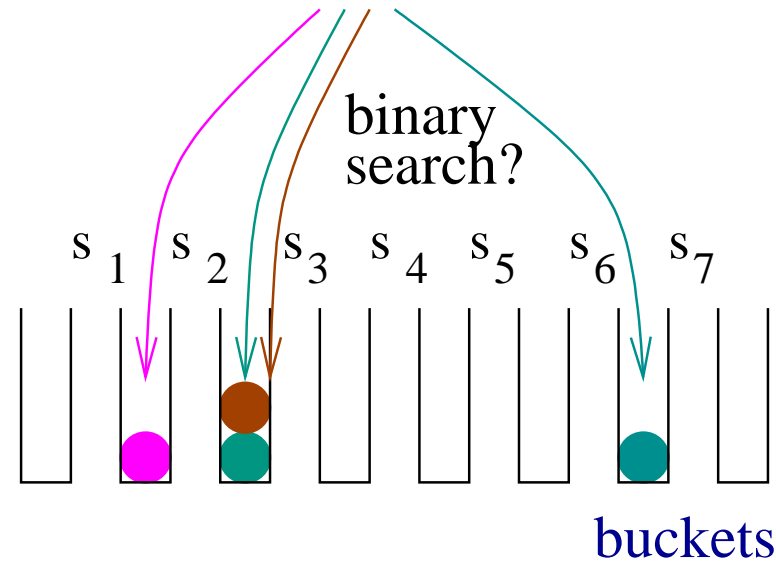
$\text{find}(k)$: Scan through $t[h(k)]$.

If an element e with $h(e) = k$ is encountered, h return it. Otherwise, return \perp .

$\mathcal{O}(|M|)$ worst case time for
remove and find



A Review of Probability



Example from hashing

sample space Ω

random hash functions $\{0..m-1\}^{\text{Key}}$

events: subsets of Ω

$$\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$$

p_x = probability of $x \in \Omega$

uniform distr. $p_h = m^{-|\text{Key}|}$

$$\mathbb{P}[\mathcal{E}] = \sum_{x \in E} p_x$$

$$\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$$

random variable $X_0 : \Omega \rightarrow \mathbb{R}$

$$X = |\{e \in M : h(e) = 0\}|.$$

expectation $E[X_0] = \sum_{y \in \Omega} p_y X(y)$

$$E[X] = \frac{|M|}{m} (*)$$

Linearity of Expectation: $E[X + Y] = E[X] + E[Y]$

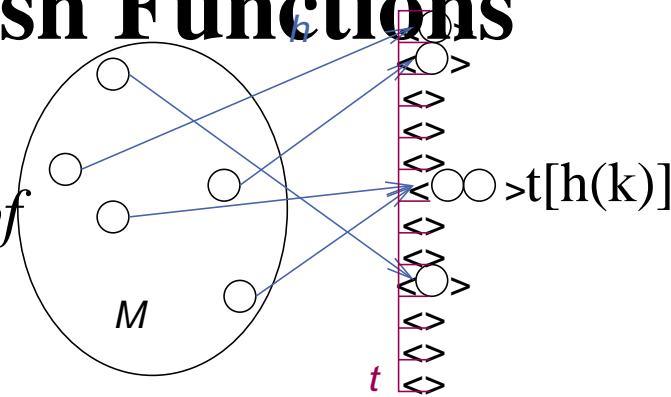
Proof of (*):

Consider the 0-1 RV $X_e = 1$ if $h(e) = 0$ for $e \in M$ and $X_e = 0$ else.

$$E[X_0] = E\left[\sum_{e \in M} X_e\right] = \sum_{e \in M} E[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = |M| \cdot \frac{1}{m}$$

Analysis for Random Hash Functions

Satz 1. The expected *execution time* of $\text{remove}(k)$ and $\text{find}(k)$ is $\mathcal{O}(1)$ if $|M| = \mathcal{O}(m)$.



Beweis. Constant time plus the *time for scanning* $t[h(k)]$.

$$X := |t[h(k)]| = |\{e \in M : h(e) = h(k)\}|.$$

Consider the 0-1 RV $X_e = 1$ if $h(e) = h(k)$ for $e \in M$ and $X_e = 0$ else.

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in M} X_e\right] = \sum_{e \in M} \mathbb{E}[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = \frac{|M|}{m} \\ &= \mathcal{O}(1) \end{aligned}$$

This is *independent of the input set* M . □

Universal Hashing

Idea: use only certain “easy” hash functions

Definition:

$\mathcal{U} \subseteq \{0..m-1\}^{\text{Key}}$ is *universal*

if for all x, y in Key with $x \neq y$ and random $h \in \mathcal{U}$,

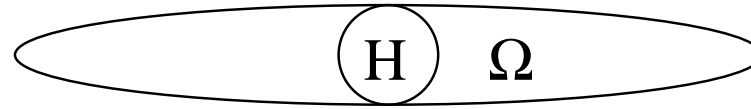
$$\mathbb{P}[h(x) = h(y)] = \frac{1}{m} .$$

Satz 2. *Theorem 1 also applies to universal families of hash functions.*

Beweis. For $\Omega = \mathcal{U}$ we still have $\mathbb{P}[X_e = 1] = \frac{1}{m}$.

The rest is as before.





A Simple Universal Family

Assume m is **prime**, $\text{Key} \subseteq \{0, \dots, m-1\}^k$

Satz 3. For $\mathbf{a} = (a_1, \dots, a_k) \in \{0, \dots, m-1\}^k$ define

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \bmod m, H = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k \right\}.$$

H is a universal family of hash functions

$$\left(\begin{array}{|c|c|c|} \hline x_1 & x_2 & x_3 \\ \hline * & * & * \\ \hline a_1 & a_2 & a_3 \\ \hline \end{array} \begin{array}{l} + \\ + \\ + \end{array} \bmod m \right) = h_{\mathbf{a}}(\mathbf{x})$$

Beweis. Consider $\mathbf{x} = (x_1, \dots, x_k)$, $\mathbf{y} = (y_1, \dots, y_k)$ with $x_j \neq y_j$
 count \mathbf{a} -s with $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.

For each choice of a_i s, $i \neq j$, \exists exactly one a_j with
 $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\begin{aligned} \sum_{1 \leq i \leq k} a_i x_i &\equiv \sum_{1 \leq i \leq k} a_i y_i \pmod{m} \\ \Leftrightarrow a_j (x_j - y_j) &\equiv \sum_{i \neq j, 1 \leq i \leq k} a_i (y_i - x_i) \pmod{m} \\ \Leftrightarrow a_j &\equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \leq i \leq k} a_i (y_i - x_i) \pmod{m} \end{aligned}$$

m^{k-1} ways to choose the a_i with $i \neq j$.

m^k is total number of \mathbf{a} s, i.e.,

$$\mathbb{P}[h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}.$$



Bit Based Universal Families

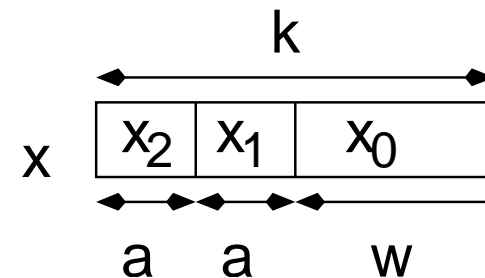
Let $m = 2^w$, Key = $\{0, 1\}^k$

Bit-Matrix Multiplication: $H^\oplus = \{h_{\mathbf{M}} : \mathbf{M} \in \{0, 1\}^{w \times k}\}$

where $h_{\mathbf{M}}(\mathbf{x}) = \mathbf{M}\mathbf{x}$ (arithmetics mod 2, i.e., xor, and)

Table Lookup: $H^{\oplus \square} = \{h_{(t_1, \dots, t_b)}^\oplus : t_i \in \{0..m-1\}^{\{0..w-1\}}\}$

where $h_{(t_1, \dots, t_b)}^\oplus((x_0, x_1, \dots, x_b)) = x_0 \oplus \bigoplus_{i=1}^b t_i[x_i]$



Hashing with Linear Probing

Open hashing: go back to original idea.

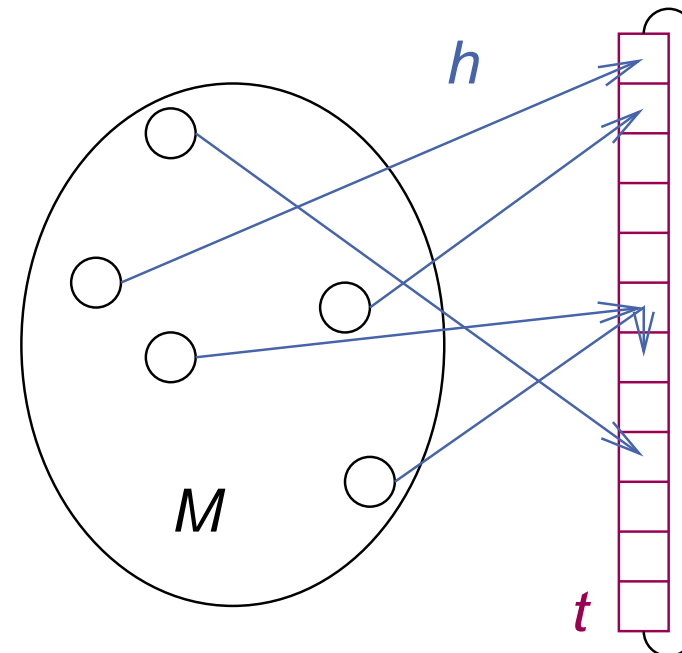
Elements are directly stored in the table.

Collisions are resolved by finding other entries.

linear probing: search for next free place by scanning the table.

Wrap around at the end.

- simple
- space efficient
- cache efficient



The Easy Part

Class BoundedLinearProbing($m, m' : \mathbb{N}; h : \text{Key} \rightarrow 0..m - 1$)

$t = [\perp, \dots, \perp] : \mathbf{Array} [0..m + m' - 1]$ **of** Element

invariant $\forall i : t[i] \neq \perp \Rightarrow \forall j \in \{h(t[i])..i - 1\} : t[j] = \perp$

Procedure insert($e : \text{Element}$)

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \perp$ **do** ;

assert $i < m + m' - 1$

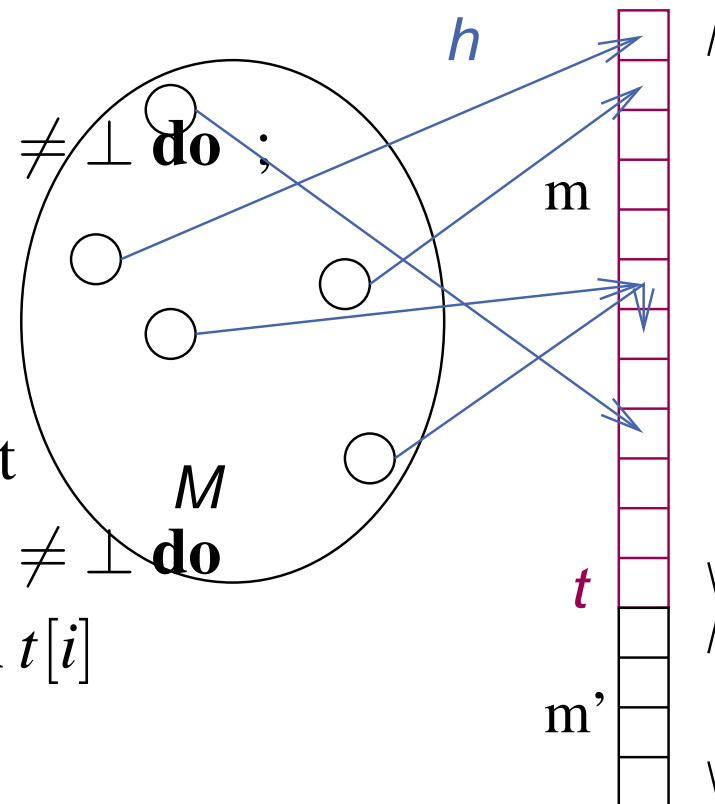
$t[i] := e$

Function find($k : \text{Key}$) : Element

for $i := h(k)$ **to** ∞ **while** $t[i] \neq \perp$ **do**

if $t[i] = k$ **then return** $t[i]$

return \perp



Remove

example: $t = [\dots, \underset{h(z)}{x}, y, z, \dots]$, $\text{remove}(x)$

invariant $\forall i : t[i] \neq \perp \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[j] \neq \perp$

Procedure $\text{remove}(k : \text{Key})$

for $i := h(k)$ **to** ∞ **while** $k \neq t[i]$ **do** // search k

if $t[i] = \perp$ **then return** // nothing to do

// we plan for a **hole at i** .

for $j := i + 1$ **to** ∞ **while** $t[j] \neq \perp$ **do**

// Establish invariant for $t[j]$.

if $h(t[j]) \leq i$ **then**

$t[i] := t[j]$ // Overwrite removed element

$i := j$ // move planned hole

$t[i] := \perp$ // erase freed entry

More Hashing Issues

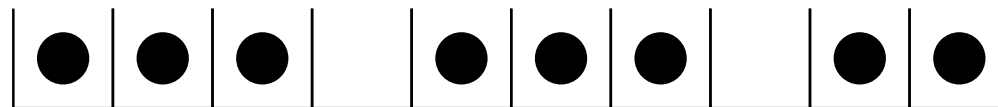
- High probability and **worst case** guarantees
 \rightsquigarrow more requirements on the hash functions
- Hashing as a means of load balancing in parallel systems,
 e.g., storage servers
 - Different disk sizes and speeds
 - Adding disks / replacing failed disks without much copying

Space Efficient Hashing with Worst Case Constant Access Time

Represent a set of n elements (with associated information) using space $(1 + \epsilon)n$.

Support operations **insert**, **delete**, **lookup**, (doall) efficiently.

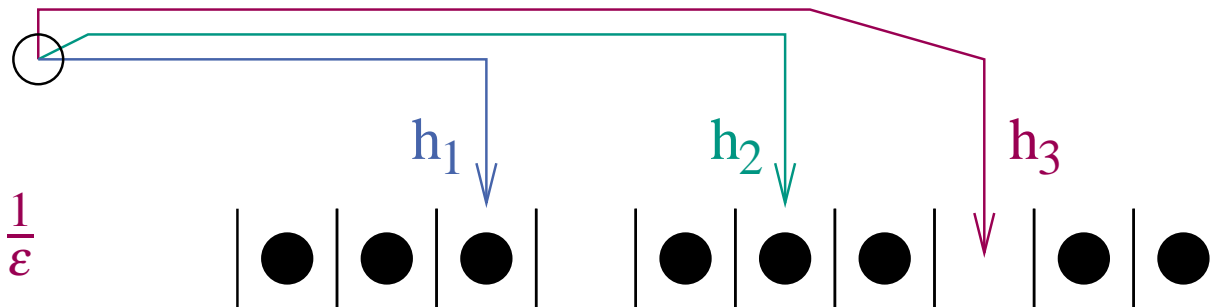
Assume a truly random hash function h



Related Work

Uniform hashing:

Expected time $\approx \frac{1}{\epsilon}$

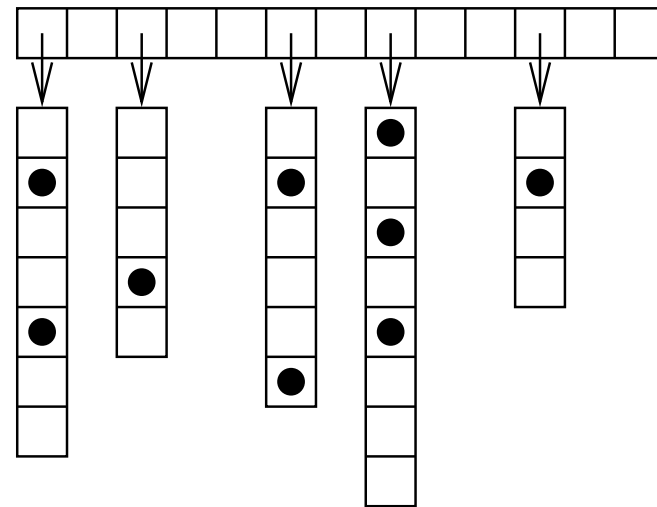


Dynamic Perfect Hashing,

[Dietzfelbinger et al. 94]

Worst case constant time

for lookup but ϵ is not small.



Approaching the Information Theoretic Lower Bound:

[Brodnik Munro 99, Raman Rao 02]

Space $(1 + o(1)) \times$ lower bound without associated information

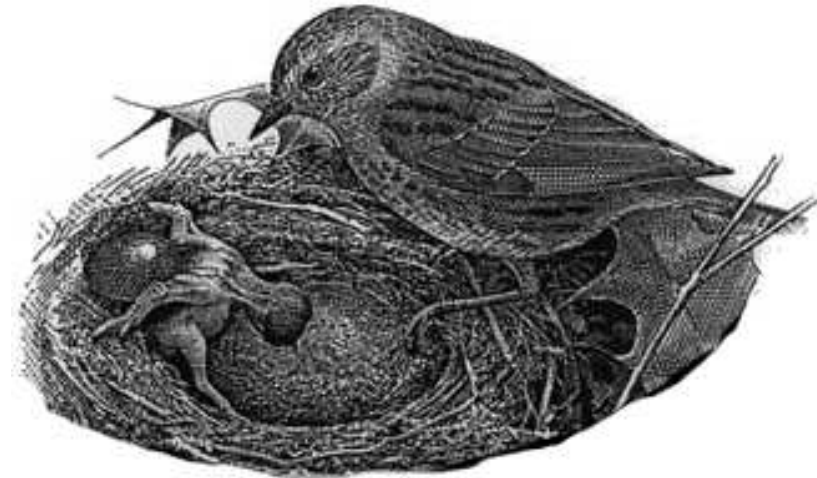
[Pagh 01] static case.

Cuckoo Hashing

[Pagh Rodler 01] Table of size $2 \cdot n$

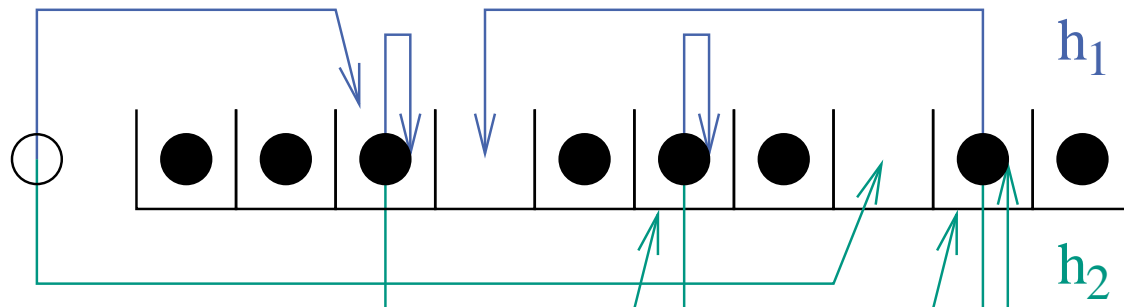
Two choices for each element.

Insert moves elements;
rebuild if necessary.



Very fast lookup and insert.

Expected constant insertion time.



d -ary Cuckoo Hashing

d choices for each element.

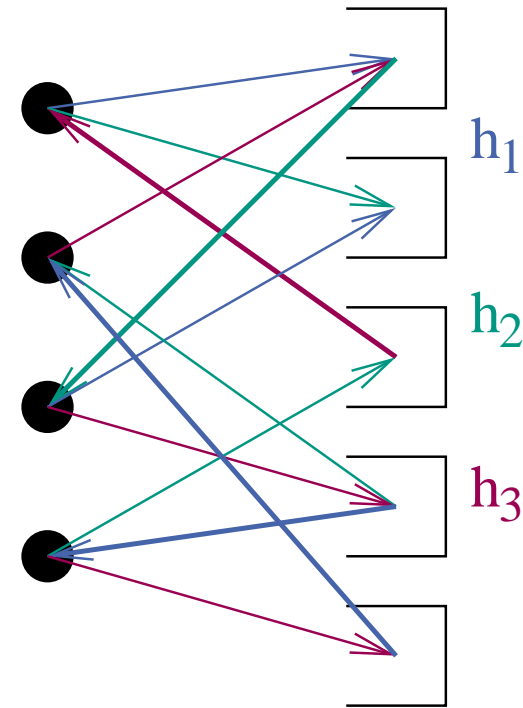
Worst case d probes for **delete** and **lookup**.

Task: maintain **perfect matching**

in the **bipartite graph**

($L = \text{Elements}$, $R = \text{Cells}$, $E = \text{Choices}$),

e.g., **insert** by **BFS** of **random walk**.

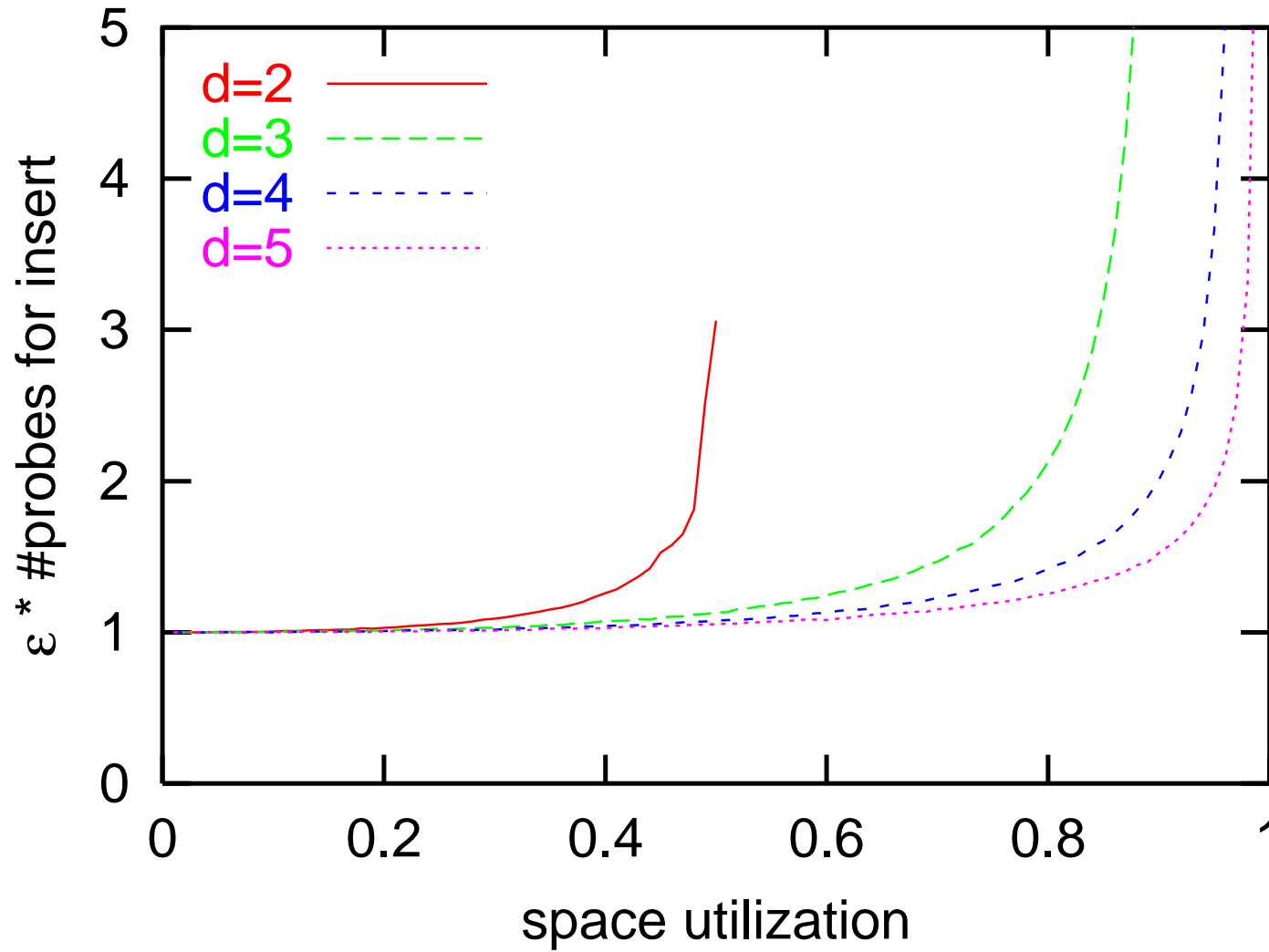


Tradeoff: Space \leftrightarrow Lookup/Deletion Time

Lookup and Delete: $d = \mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ probes

Insert: $\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\log(1/\varepsilon))}$, (experiments) $\longrightarrow \mathcal{O}(1/\varepsilon)$?

Experiments



Open Questions and Further Results

- Tight analysis of **insertion**
- Two choices with d slots each [Dietzfelbinger et al.]
 \rightsquigarrow cache efficiency

Good Implementation?

- Automatic rehash
- Always **correct**

5 Minimum Spanning Trees

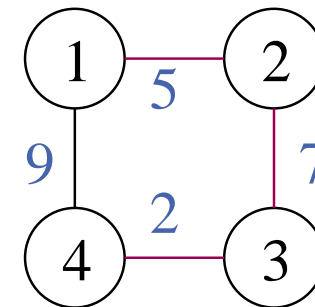
undirected Graph $G = (V, E)$.

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

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

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

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.

MST: Overview

- Basics: Edge property and cycle property
- Jarník-Prim Algorithm
- Kruskals Algorithm
- Filter-Kruskal
- Comparison
- (Advanced algorithms using the cycle property)
- External MST

Applications

- Clustering
- Subroutine in combinatorial optimization, e.g., Held-Karp lower bound for TSP.
Challenging real world instances???
- Image segmentation → [Diss. Jan Wassenberg]

Anyway: almost ideal “fruit fly” problem

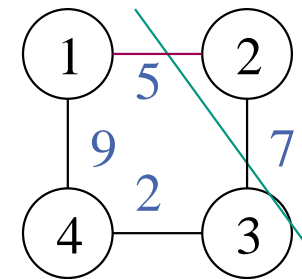
Selecting and Discarding MST Edges

The Cut Property

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

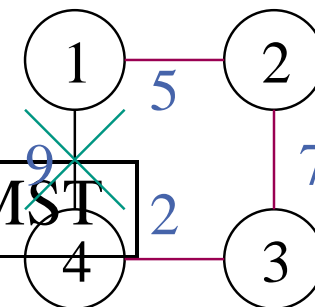
$$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$$

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

$T := \emptyset$

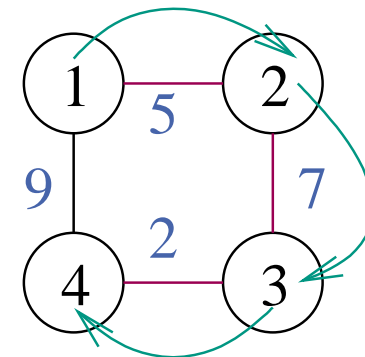
$S := \{s\}$ for arbitrary start node s

repeat $n - 1$ times

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

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

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



Implementation Using Priority Queues

Function $\text{jpMST}(V, E, w) : \text{Set of Edge}$

dist = $[\infty, \dots, \infty]$: **Array** $[1..n]$ // $\text{dist}[v]$ is distance of v from the tree

pred : **Array of Edge** // $\text{pred}[v]$ is shortest edge between S and v

q : **PriorityQueue of Node** with **dist** $[\cdot]$ as priority

$\text{dist}[s] := 0$; $q.\text{insert}(s)$ for any $s \in V$

for $i := 1$ **to** $n - 1$ **do do**

$u := q.\text{deleteMin}()$ // new node for S

$\text{dist}[u] := 0$

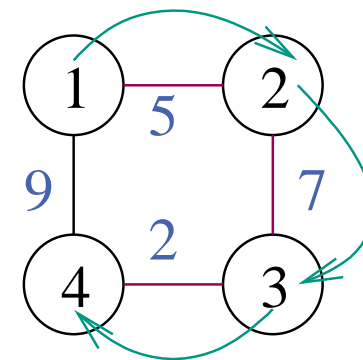
foreach $(u, v) \in E$ **do**

if $c((u, v)) < \text{dist}[v]$ **then**

$\text{dist}[v] := c((u, v)); \text{pred}[v] := (u, v)$

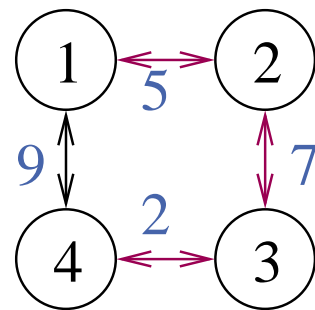
if $v \in q$ **then** $q.\text{decreaseKey}(v)$ **else** $q.\text{insert}(v)$

return $\{\text{pred}[v] : v \in V \setminus \{s\}\}$



Graph Representation for Jarník-Prim

We need node \rightarrow incident edges



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

- + fast (cache efficient)
- + more compact than linked lists
- difficult to change
- Edges are stored twice

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**

practical implementation using simpler **pairing heaps**.

But analysis is still partly **open**!

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  $T$ 
```

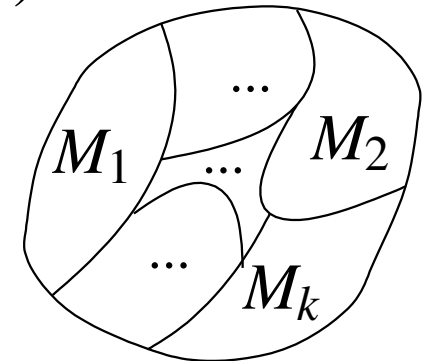
Union-Find Datenstruktur

Verwalte **Partition** der Menge $1..n$, d. h., Mengen (Blocks)

M_1, \dots, M_k mit

$$M_1 \cup \dots \cup M_k = 1..n,$$

$$\forall i \neq j : M_i \cap M_j = \emptyset$$



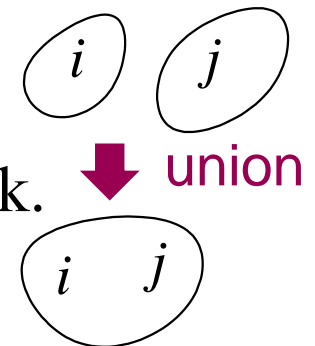
Class UnionFind($n : \mathbb{N}$)

Procedure union($i, j : 1..n$)

join the blocks containing i and j to a single block.

Function find($i : 1..n$) : $1..n$

return a unique identifier for the block containing i .

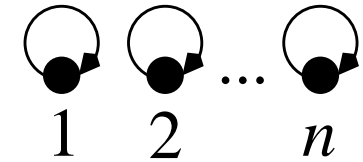


Union-Find Datenstruktur – Erste Version

Class UnionFind($n : \mathbb{N}$)

parent = $\langle 1, 2, \dots, n \rangle$: **Array** $[1..n]$ of $1..n$

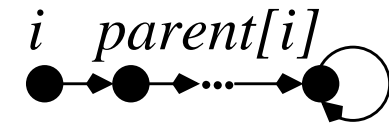
invariant parent-refs lead to unique **Partition-Reps**



Function find($i : 1..n$) : $1..n$

if parent[i] = i **then return** i

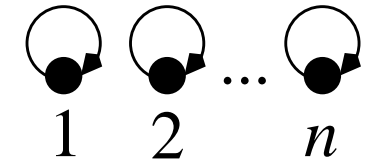
else return find(parent[i])



Union-Find Datenstruktur – Erste Version

Class UnionFind($n : \mathbb{N}$)

parent = $\langle 1, 2, \dots, n \rangle$: **Array** $[1..n]$ of $1..n$

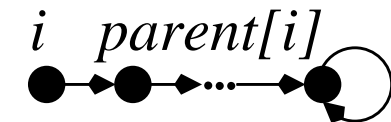


invariant parent-refs lead to unique **Partition-Reps**

Function find($i : 1..n$) : $1..n$

if parent[i] = i **then return** i

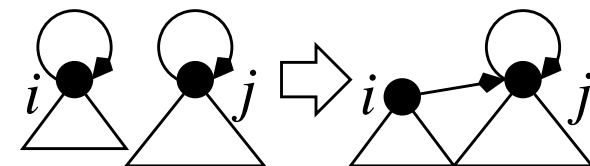
else return find(parent[i])



Procedure link($i, j : 1..n$)

assert i and j are representatives of different blocks

parent[i] := j



Procedure union($i, j : 1..n$)

if find(i) \neq find(j) **then** link(find(i), find(j))

Union-Find Datenstruktur – Erste Version

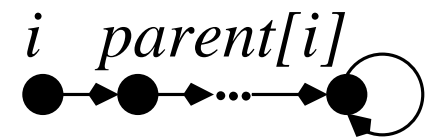
Analyse:

+: **union** braucht konstante Zeit

–: **find** braucht Zeit $\Theta(n)$ im schlechtesten Fall !

zu langsam.

Idee: **find-Pfade kurz halten**



Pfadkompression

Class UnionFind($n : \mathbb{N}$)

parent = $\langle 1, 2, \dots, n \rangle$: **Array** [1..n] of 1..n

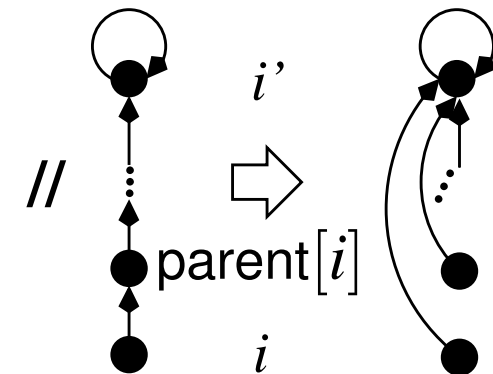
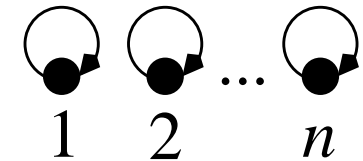
Function find($i : 1..n$) : 1..n

if parent[i] = i **then return** i

else $i' :=$ find(parent[i])

parent[i] := i'

return i'

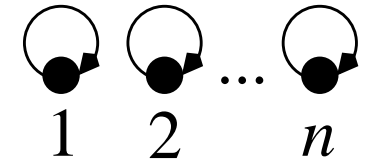


Union by Rank

Class UnionFind($n : \mathbb{N}$)

parent = $\langle 1, 2, \dots, n \rangle$: **Array** [1..n] of 1..n

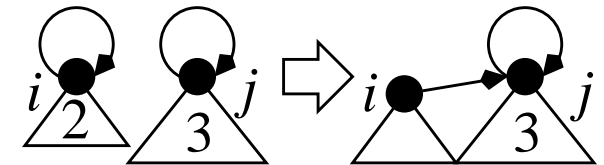
rank = $\langle 0, \dots, 0 \rangle$: **Array** [1..n] of 0..log n



Procedure link($i, j : 1..n$)

assert i and j are representatives of different blocks

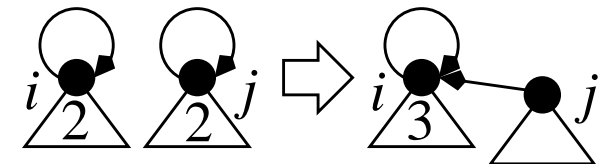
if rank[i] < rank[j] **then** parent[i] := j



else

parent[j] := i

if rank[i] = rank[j] **then** rank[i] ++



Space Efficient Union by Rank

Class UnionFind($n : \mathbb{N}$) // Maintain a partition of $1..n$

parent = $[n + 1, \dots, n + 1]$: **Array** $[1..n]$ of $1..n + \lceil \log n \rceil$

Function find($i : 1..n$) : $1..n$

if $\text{parent}[i] > n$ then return i

else $i' := \text{find}(\text{parent}[i])$

parent[i] := i'

return i'

Procedure link($i, j : 1..n$)

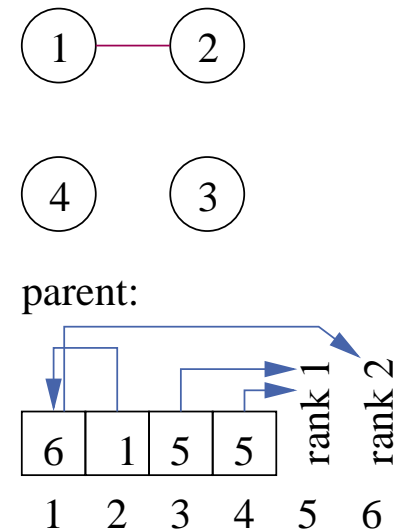
assert i and j are leaders of different subsets

if $\text{parent}[i] < \text{parent}[j]$ then $\text{parent}[i] := j$

else if $\text{parent}[i] > \text{parent}[j]$ then $\text{parent}[j] := i$

else $\text{parent}[j] := i$; $\text{parent}[i]++$ // next generation

Procedure union(i, j) if $\text{find}(i) \neq \text{find}(j)$ then link($\text{find}(i), \text{find}(j)$)



Kruskal Using Union Find

$T : \text{UnionFind}(n)$

sort E in ascending order of weight

$\text{kruskal}(E)$

Procedure $\text{kruskal}(E)$

foreach $(u, v) \in E$ **do**

$u' := T.\text{find}(u)$

$v' := T.\text{find}(v)$

if $u' \neq v'$ **then**

output (u, v)

$T.\text{link}(u', v')$

Graph Representation for Kruskal

Just an edge sequence (array) !

- + very fast (cache efficient)
- + Edges are stored only once
- ↪ more compact than adjacency array

Analysis

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

Kruskal versus Jarník-Prim I

- Kruskal wins for very sparse graphs
- Prim seems to win for denser graphs
- Switching point is **unclear**
 - How is the input **represented**?
 - How many **decreaseKeys** are performed by JP?
(average case: $n \log \frac{m}{n}$ [Noshita 85])
 - Experimental studies are quite **old** [Moret Shapiro 91],
use **slow** graph **representation** for both algs,
and **artificial inputs**

see attached slides.

5.1 Filtering by Sampling Rather Than Sorting

$R :=$ random sample of r edges from E

$F := \text{MST}(R)$ // Wlog assume that F spans V

$L := \emptyset$ // “light edges” with respect to R

foreach $e \in E$ **do** // Filter

$C :=$ the unique cycle in $\{e\} \cup F$

if e is not heaviest in C **then**

$L := L \cup \{e\}$

return $\text{MST}((L \cup F))$

5.1.1 Analysis

[Chan 98, KKK 95]

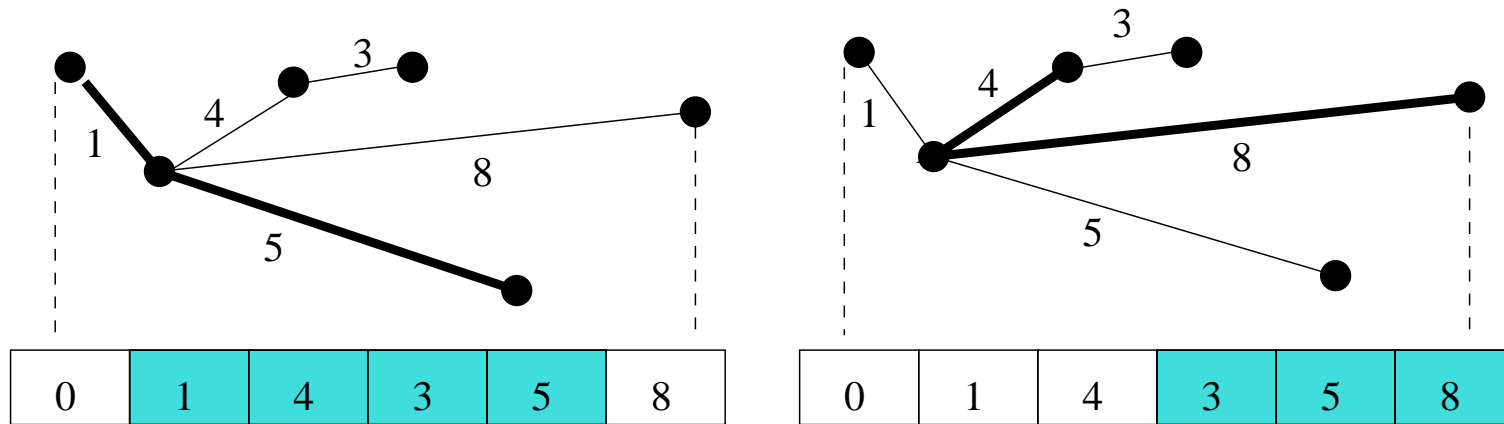
Observation: $e \in L$ only if $e \in \text{MST}(R \cup \{e\})$.

(Otherwise e could replace some heavier edge in F).

Lemma 4. $E[|L \cup F|] \leq \frac{mn}{r}$

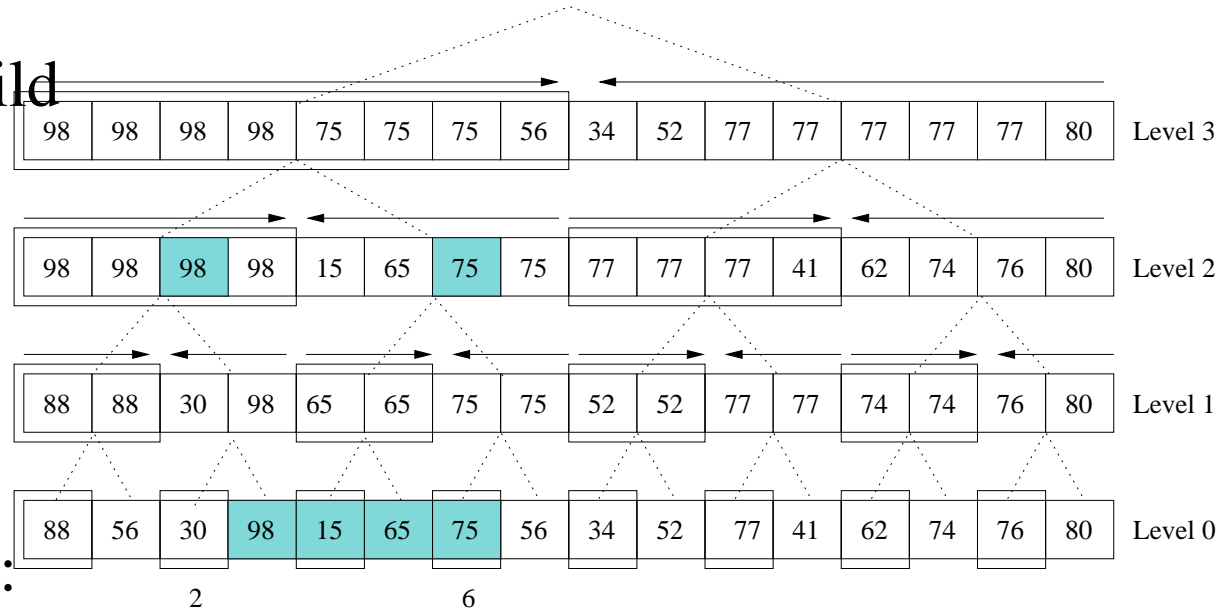
MST Verification by Interval Maxima

- Number the nodes by the order they were added to the MST by Prim's algorithm.
- $w_i =$ weight of the edge that inserted node i .
- Largest weight on path(u, v) = $\max\{w_j \mid u < j \leq v\}$.



Interval Maxima

Preprocessing: build
 $n \log n$ size array
 PreSuf.



To find $\max a[i..j]$:

- Find the level of the LCA: $\ell = \lfloor \log_2(i \oplus j) \rfloor$.
- Return $\max(\text{PreSuf}[\ell][i], \text{PreSuf}[\ell][j])$.
- Example: $2 \oplus 6 = 010 \oplus 110 = 100 \Rightarrow \ell = 2$

A Simple Filter Based Algorithm

Choose $r = \sqrt{mn}$.

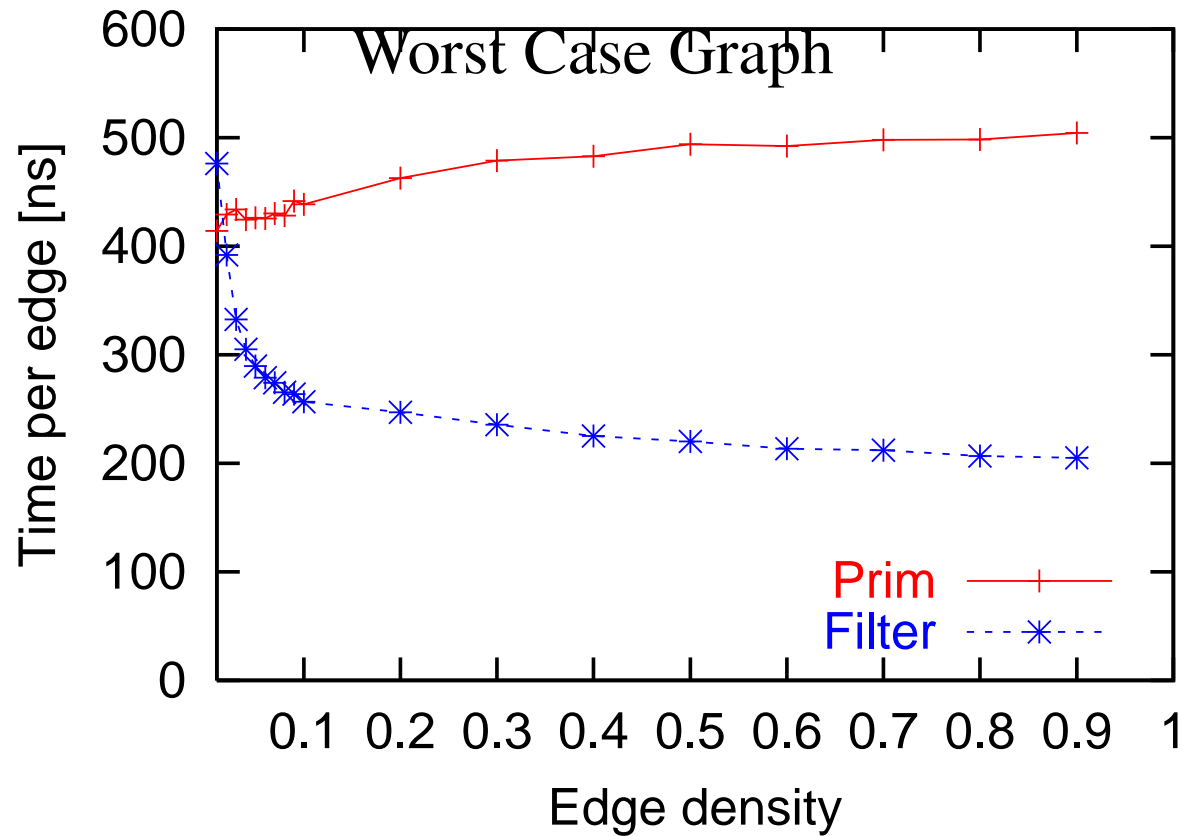
We get expected time

$$T_{\text{Prim}}(\sqrt{mn}) + \mathcal{O}(n \log n + m) + T_{\text{Prim}}\left(\frac{mn}{\sqrt{mn}}\right) = \mathcal{O}(n \log n + m)$$

The constant factor in front of the m is very small.

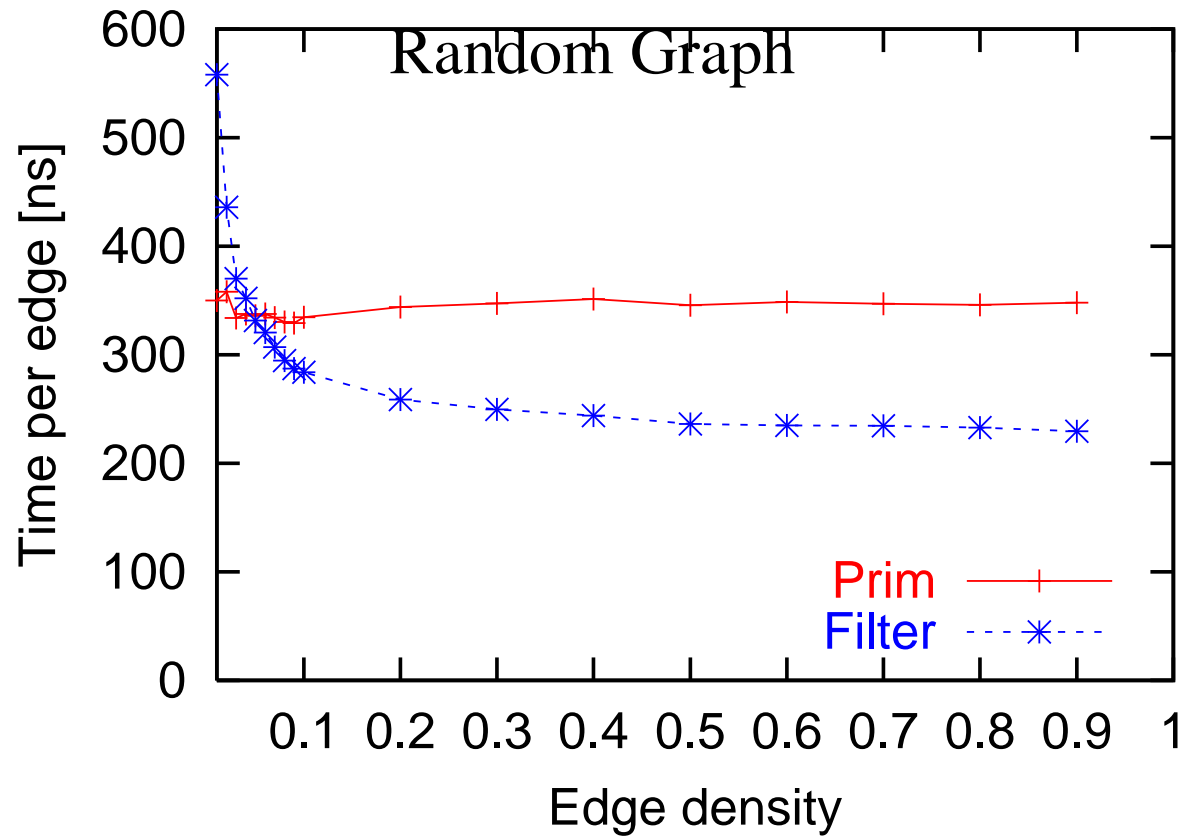
Results

10 000 nodes, SUN-Fire-15000, 900 MHz UltraSPARC-III+

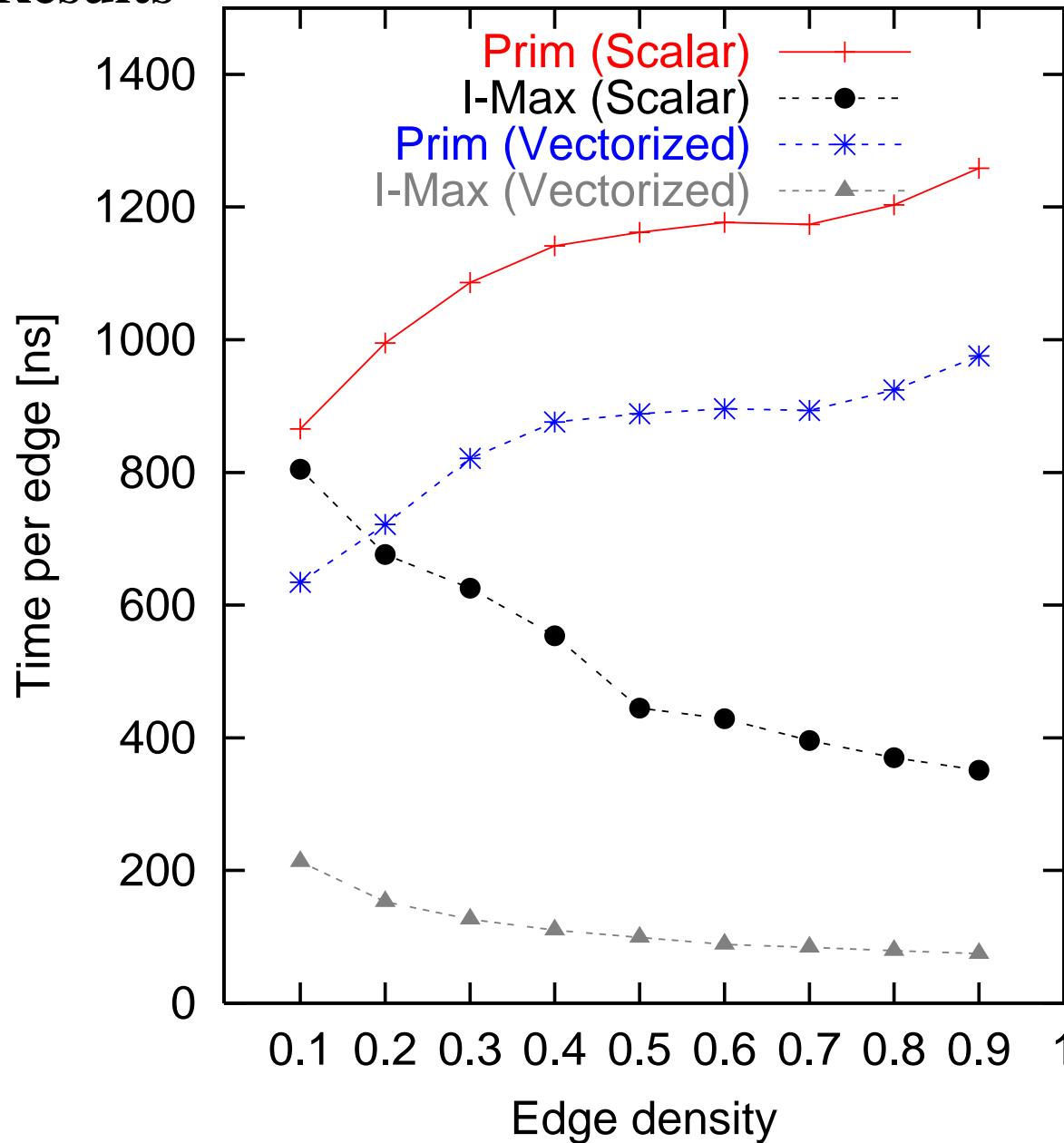


Results

10 000 nodes, SUN-Fire-15000, 900 MHz UltraSPARC-III+



Results



10 000 nodes,
NEC SX-5
Vector Machine
“worst case”

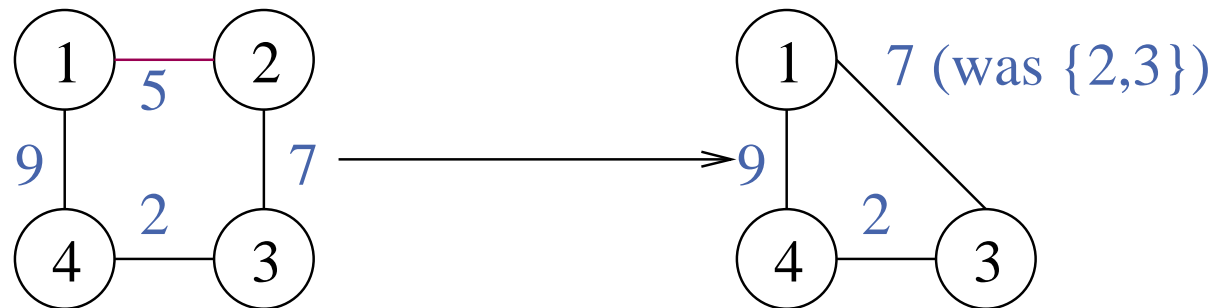
Edge Contraction

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 original terminals

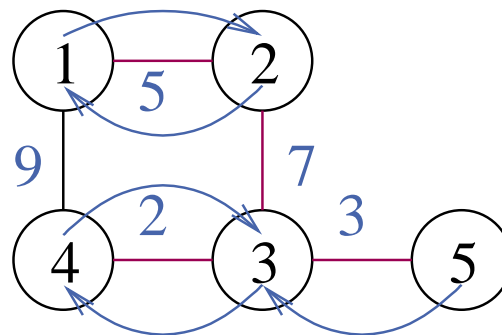


Boruvka's Node Reduction Algorithm

For each node find the lightest incident edge.
Include them into the MST (cut property)
contract these edges,

Time $\mathcal{O}(m)$

At least halves the number of remaining nodes



5.2 Simpler and Faster Node Reduction

for $i := n$ **downto** $n' + 1$ **do**

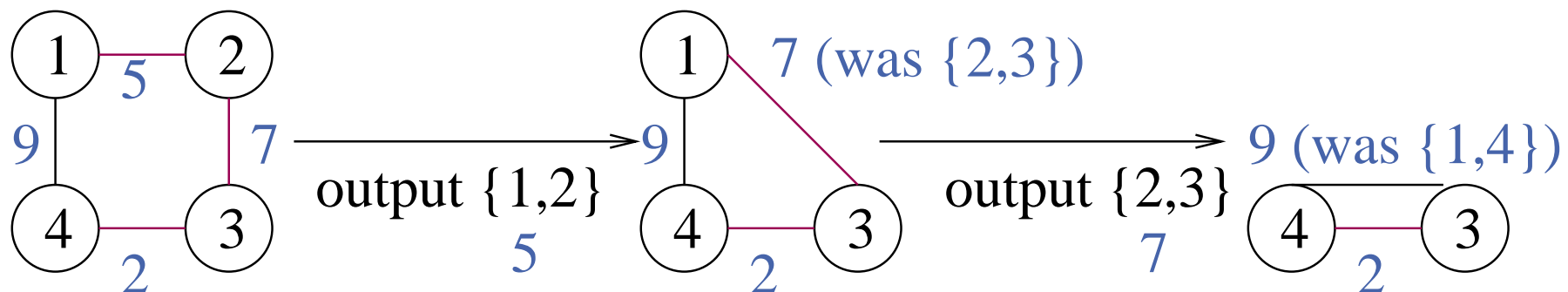
 pick a random node v

 find the **lightest** edge (u, v) out of v and output it

 contract (u, v)

$$E[\text{degree}(v)] \leq 2m/i$$

$$\sum_{n' < i \leq n} \frac{2m}{i} = 2m \left(\sum_{0 < i \leq n} \frac{1}{i} - \sum_{0 < i \leq n'} \frac{1}{i} \right) \approx 2m(\ln n - \ln n') = 2m \ln \frac{n}{n'}$$



5.3 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). $\mathcal{O}(m + n)$

$$\mathbf{E}[|L|] \leq \frac{mn/8}{m/2} = n/4.$$

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.

5.4 External MSTs

Semiexternal Algorithms

Assume $n \leq M - 2B$:

run **Kruskal's algorithm** using **external sorting**

Streaming MSTs

If M is yet a bit larger we can even do it with m/B I/Os:

```
 $T := \emptyset$  // current approximation of MST  
while there are any unprocessed edges do  
    load any  $\Theta(M)$  unprocessed edges  $E'$   
     $T := \text{MST}(T \cup E')$  // for any internal MST alg.
```

Corollary: we can do it with **linear** expected **internal work**

Disadvantages to Kruskal:

Slower in practice

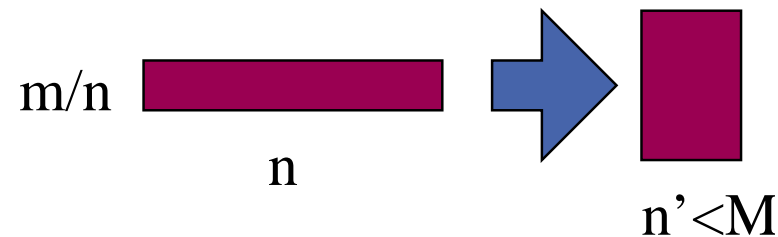
Smaller max. n

General External MST

while $n > M - 2B$ **do**

 perform some node reduction

use semi-external Kruskal



Theory: $\mathcal{O}(\text{sort}(m))$ expected I/Os by externalizing the linear time algorithm.

(i.e., node reduction + edge reduction)

External Implementation I: Sweeping

π : random permutation $V \rightarrow V$

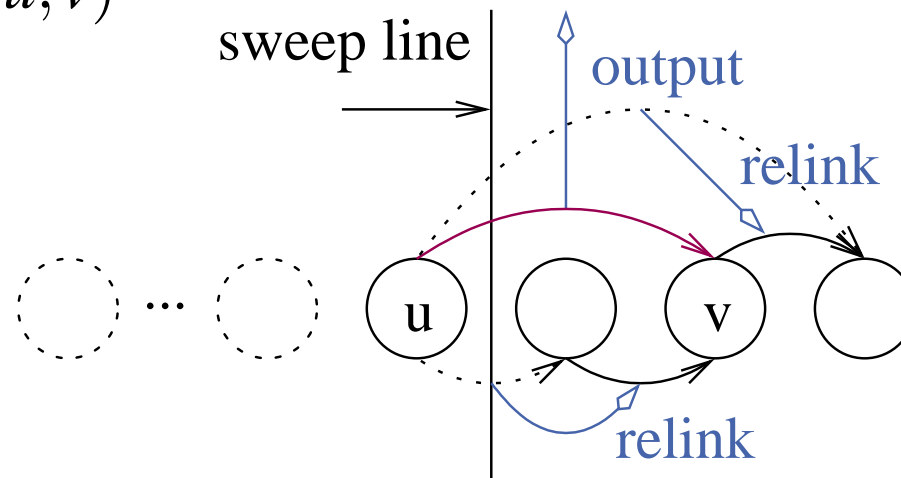
sort edges (u, v) by $\max(\pi(u), \pi(v))$

for $i := n$ **downto** $n' + 1$ **do**

 pick the node v with $\pi(v) = i$

 find the **lightest** edge (u, v) out of v and output it

 contract (u, v)



Problem: how to implement relinking?

Relinking Using Priority Queues

Q: priority queue // Order: **max node**, then **min edge weight**

foreach $(\{u, v\}, c) \in E$ **do** Q.insert($(\{\pi(u), \pi(v)\}, c, \{u, v\})$)

current := $n + 1$

loop

$(\{u, v\}, c, \{u_0, v_0\}) :=$ Q.deleteMin()

if current \neq max $\{u, v\}$ **then**

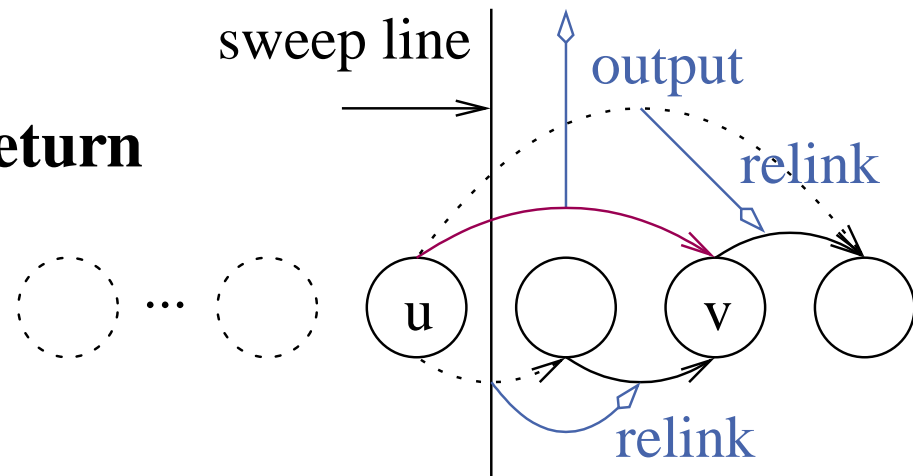
if current = $M + 1$ **then return**

output $\{u_0, v_0\}, c$

current := max $\{u, v\}$

connect := min $\{u, v\}$

else Q.insert($(\{\min \{u, v\}, \text{connect}\}, c, \{u_0, v_0\})$)



\approx $\text{sort}(10m \ln \frac{n}{M})$ I/Os with opt. priority queues

[Sanders 00]

Problem: **Compute** bound

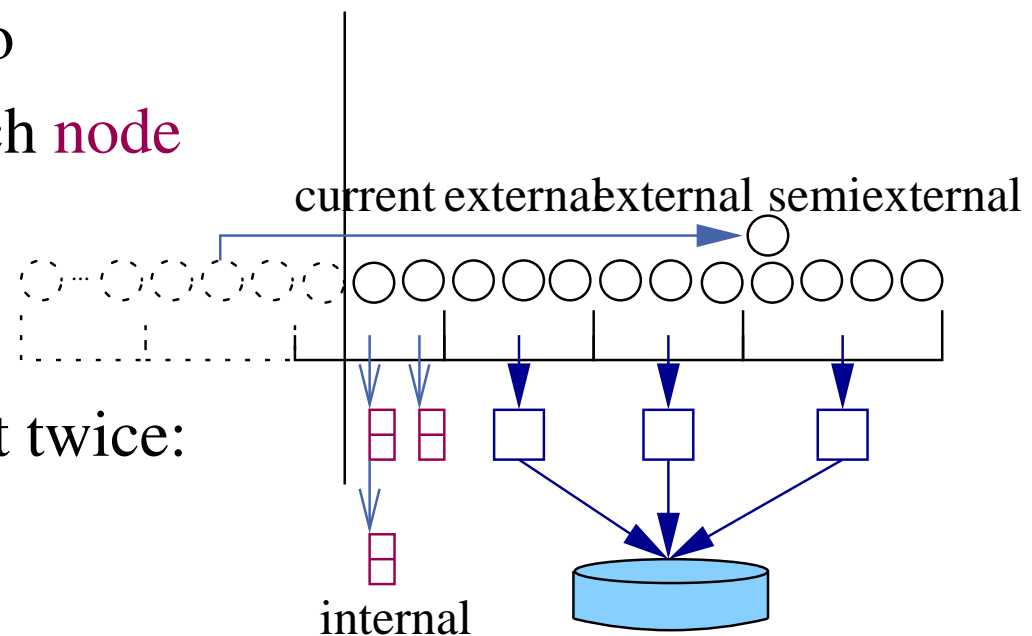
Sweeping with linear internal work

- Assume $m = \mathcal{O}(M^2/B)$
- $k = \Theta(M/B)$ external buckets with n/k nodes each
- M nodes for last “semiexternal” bucket
- split current bucket into internal buckets for each node

Sweeping:

Scan current internal bucket twice:

1. Find minimum
2. Relink



New external bucket: scan and put in internal buckets

Large degree nodes: move to semiexternal bucket

Experiments

Instances from “classical” MST study [Moret Shapiro 1994]

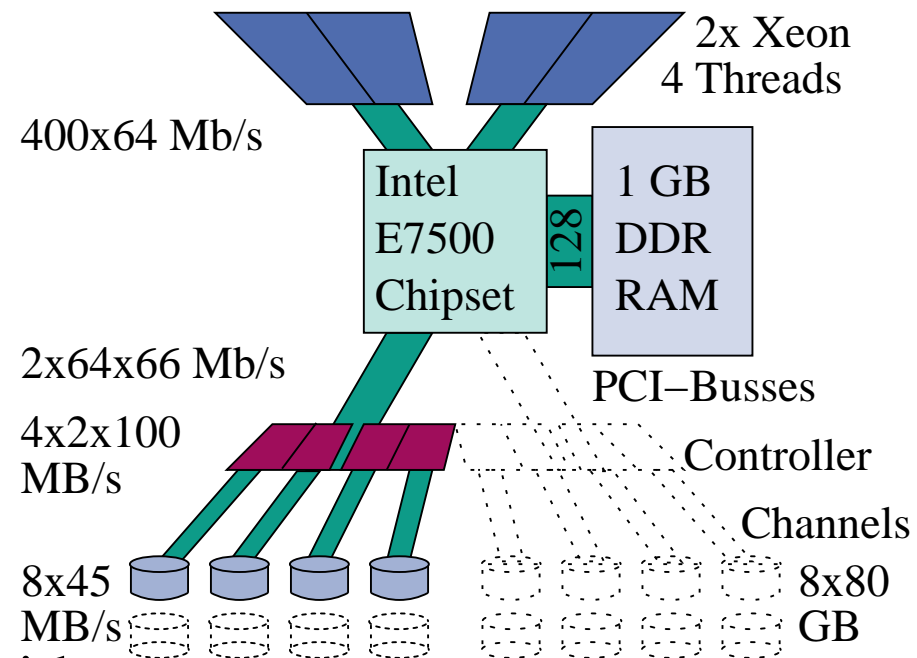
- sparse random graphs
- random geometric graphs
- grids

$\mathcal{O}(\text{sort}(m))$ I/Os

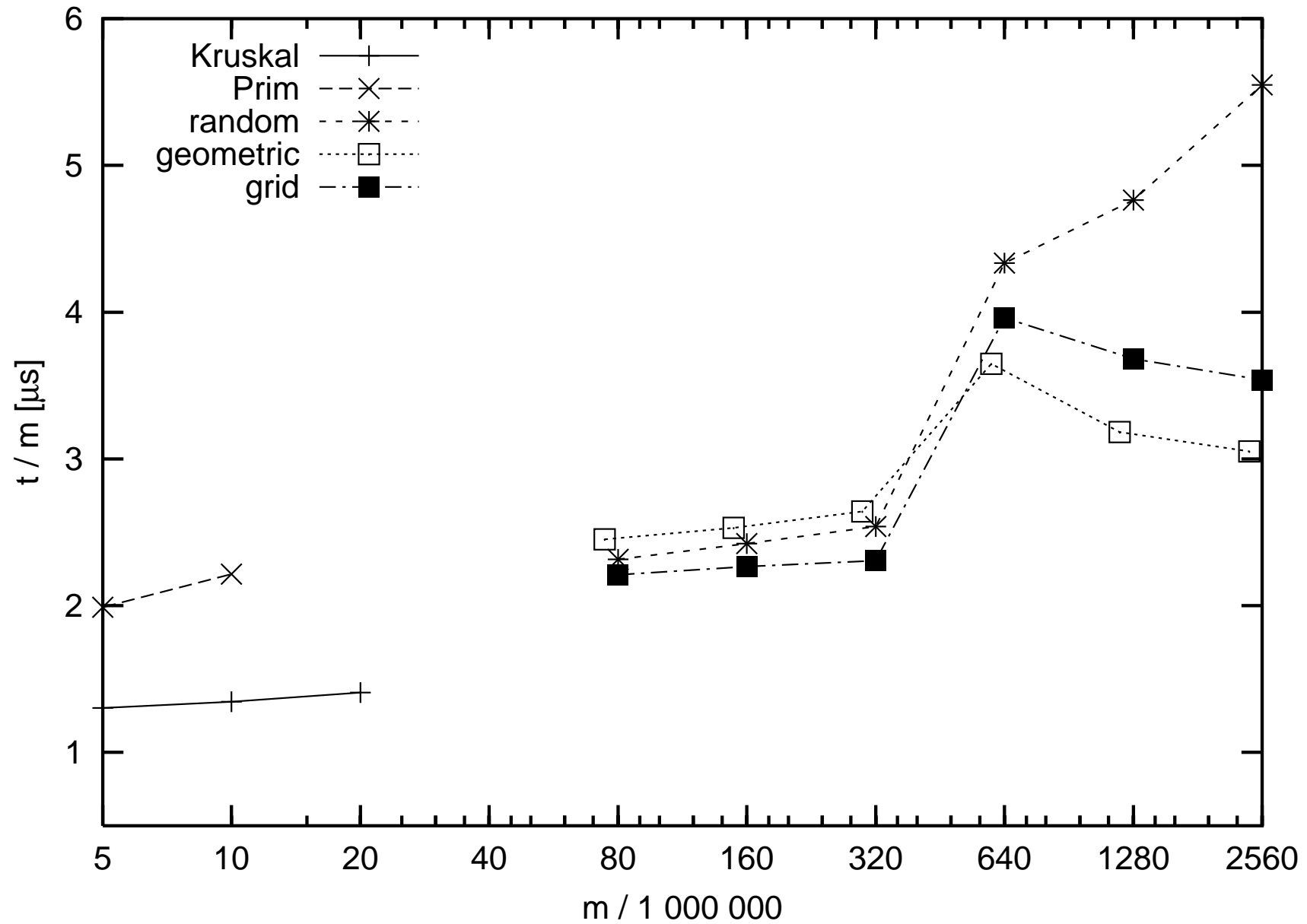
for planar graphs by

removing parallel edges!

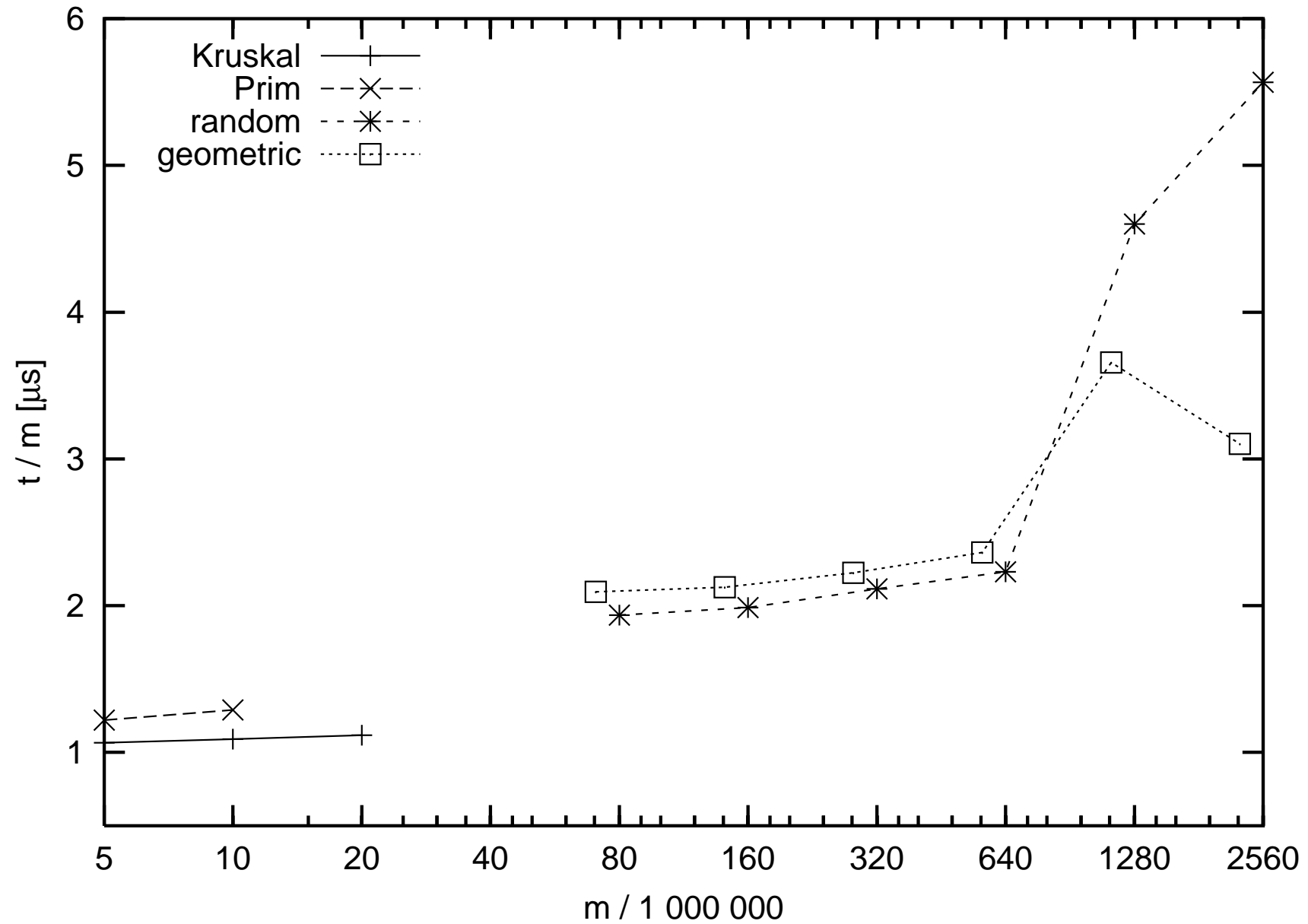
Other instances are rather dense or designed to fool specific algorithms.



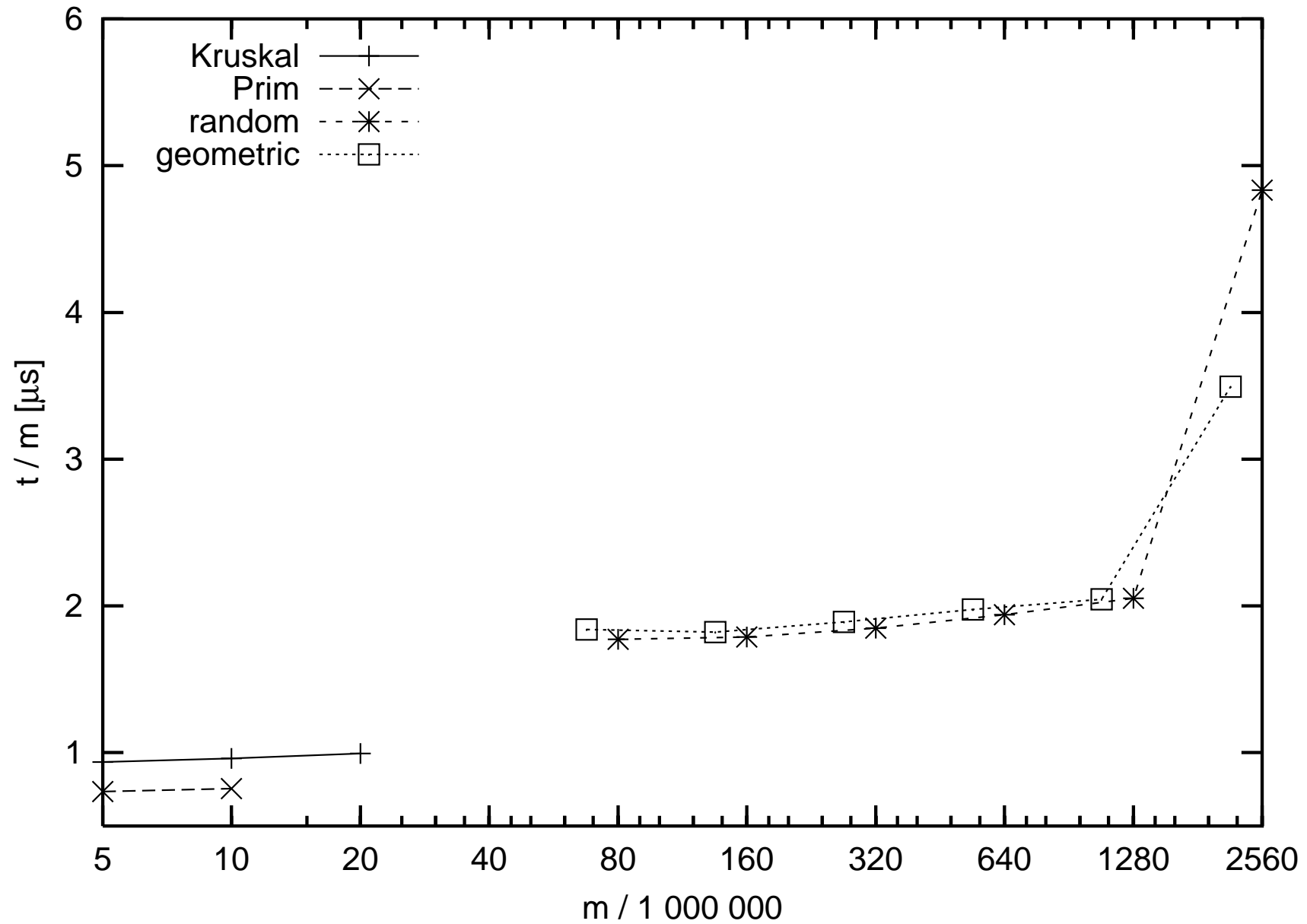
$$m \approx 2n$$



$$m \approx 4n$$



$$m \approx 8n$$



MST Summary

- Edge reduction helps for very dense, “hard” graphs
- A fast and simple **node reduction** algorithm
 \rightsquigarrow $4\times$ less I/Os than previous algorithms
- Refined semiexternal MST, use as **base case**

- Simple pseudo random permutations (no I/Os)
- A fast **implementation**
- Experiments with huge graphs (up to $n = 4 \cdot 10^9$ nodes)

External MST is feasible

Conclusions

- Even fundamental, “simple” algorithmic problems still raise interesting questions
- Implementation and experiments are important and were neglected by parts of the algorithms community
- Theory** an (at least) equally important, essential component of the algorithm design process

Open Problems

- New experiments for (improved) Kruskal versus Jarník-Prim
- Realistic (huge) inputs
- Parallel and/or external algorithms
Matthias Schimek just started on this
- A practical linear time Algorithm (Hagerup 2010?)
- Implementations for other graph problems

More Algorithm Engineering on Graphs

- Parallel algorithms
- Graph partitioning \rightsquigarrow KaHiP
- Hypergraph partitioning \rightsquigarrow KaHyPar
- Graph generators \rightsquigarrow KaGen
- Independent sets
- Route planning

Maximal Flows

Theory: $\mathcal{O}(m\Lambda \log(n^2/m) \log U)$ binary blocking flow-algorithm mit $\Lambda = \min\{m^{1/2}, n^{2/3}\}$ [Goldberg-Rao-97].

Problem: best case \approx worst case

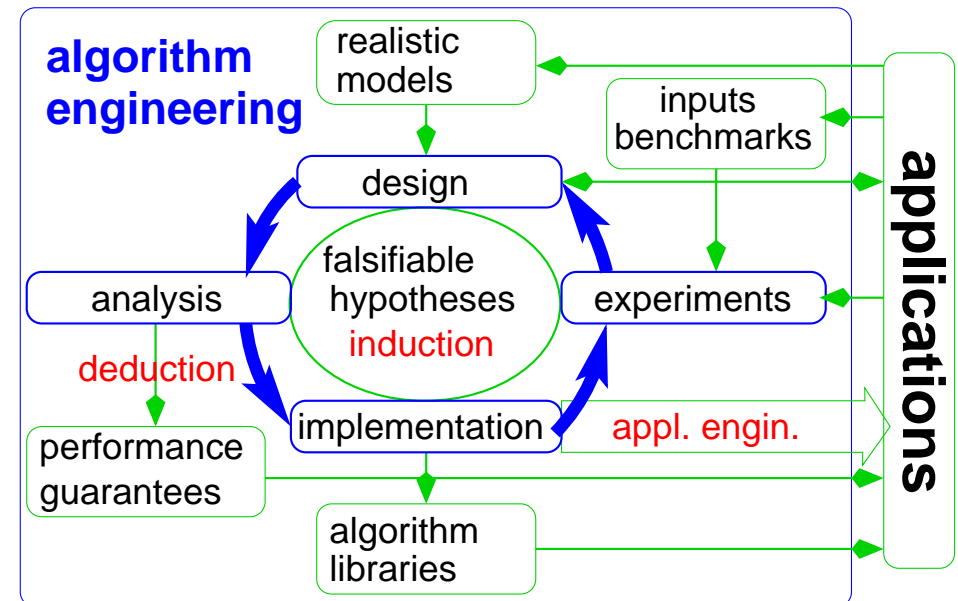
[Hagerup Sanders Träff WAE 98]:

- Implementable generalization
- best case \ll worst case
- best algorithms for some “difficult” instances

More On Experimental Methodology

Scientific Method:

- Experiment need a possible outcome that **falsifies** a hypothesis
- Reproducible
 - keep data/code for at least 10 years
 - clear and detailed description in papers / TRs
 - share instances and code



Quality Criteria

- Beat the state of the art, globally – (not your own toy codes or the toy codes used in your community!)
- Clearly** demonstrate this !
 - both codes use same data ideally from accepted benchmarks (not just your favorite data!)
 - comparable machines or fair (conservative) scaling
 - Avoid uncomparabilities like: “Yeah we are worse but twice as fast”
 - real world data wherever possible
 - as much different inputs as possible
 - its fine if you are better just on some (important) inputs

Not Here but Important

- describing the setup
- finding sources of measurement errors
- reducing measurement errors (averaging, median, unloaded machine...)
- measurements in the **creative** phase of experimental algorithmics.

The Starting Point

- (Several) Algorithm(s)
- A few quantities to be measured: time, space, solution quality, comparisons, cache faults, ... There may also be **measurement errors**.
- An unlimited number of potential inputs. \rightsquigarrow condense to a few characteristic ones (size, $|V|$, $|E|$, ... or problem instances from applications)

Usually there is not a lack but an **abundance** of data \neq many other sciences

The Process

Waterfall model?

1. Design
2. Measurement
3. Interpretation

Perhaps the paper should at least look like that.

The Process

- Eventually stop asking questions (Advisors/Referees listen !)
- build measurement tools
- automate (re)measurements
- Choice of experiments driven by risk and opportunity
- Distinguish mode
 - explorative:** many different parameter settings, interactive, short turnaround times
 - consolidating:** many large instances, standardized measurement conditions, batch mode, many machines

Of Risks and Opportunities

Example: Hypothesis = my algorithm is the best

big risk: untried main competitor

small risk: tuning of a subroutine that takes 20 % of the time.

big opportunity: use algorithm for a new application

~> new input instances

Presenting Data from Experiments in Algorithmics

Restrictions

- black and white \rightsquigarrow easy and cheap printing
- 2D (stay tuned)
- no animation
- no realism desired

Basic Principles

- Minimize nondata ink
(form follows function, not a beauty contest,...)
- Letter size \approx surrounding text
- Avoid clutter and overwhelming complexity
- Avoid boredom (too little data per m^2).
- Make the conclusions evident

Tables

- + easy
- easy \rightsquigarrow overuse
- + accurate values (\neq 3D)
- + more compact than bar chart
- + good for unrelated instances (e.g. solution quality)
- boring
- no visual processing

rule of thumb that “tables usually outperform a graph for small data sets of 20 numbers or less” [Tufte 83]

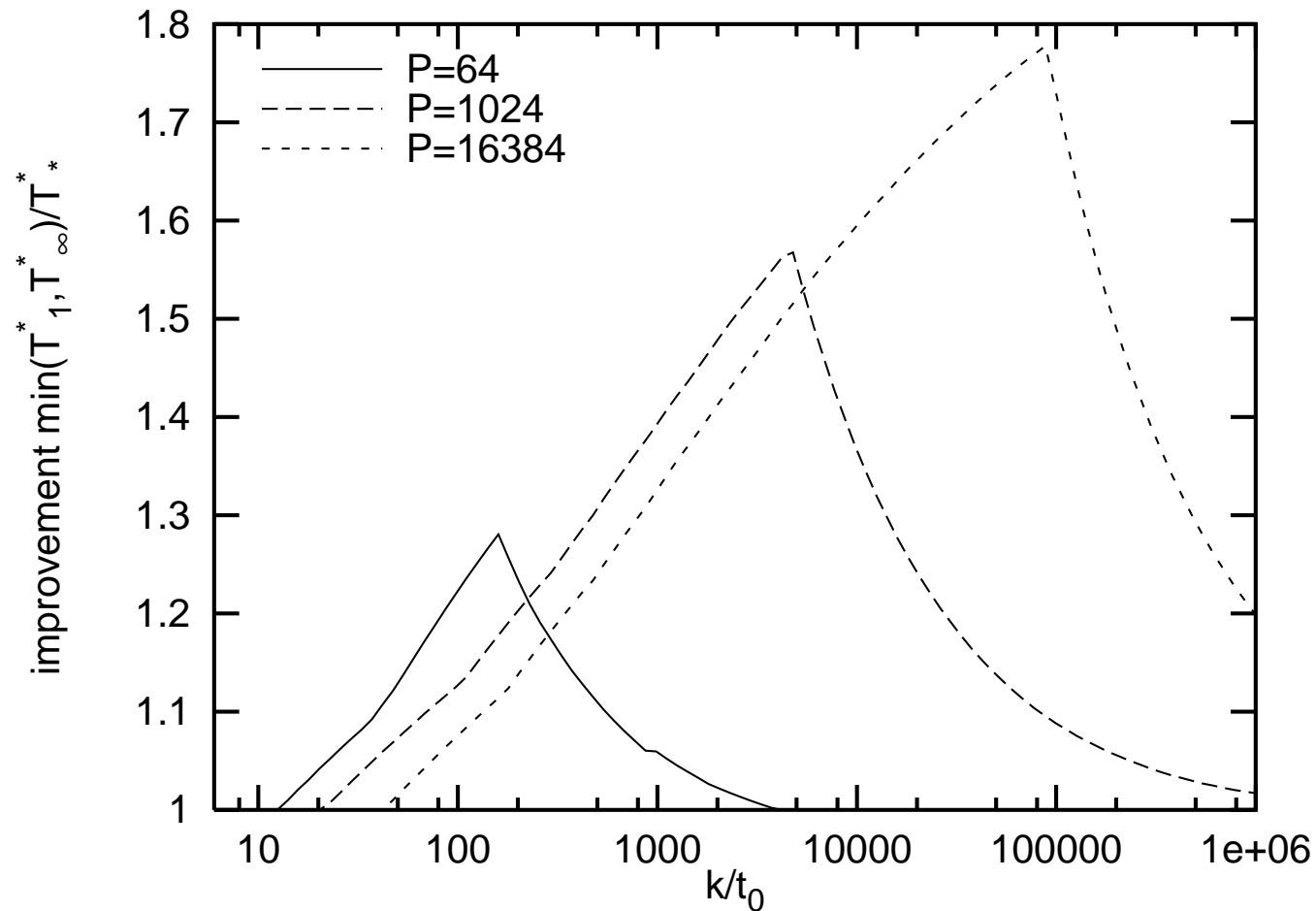
Curves in main paper, tables in appendix?

2D Figures

default: $x = \text{input size}$, $y = f(\text{execution time})$

x Axis

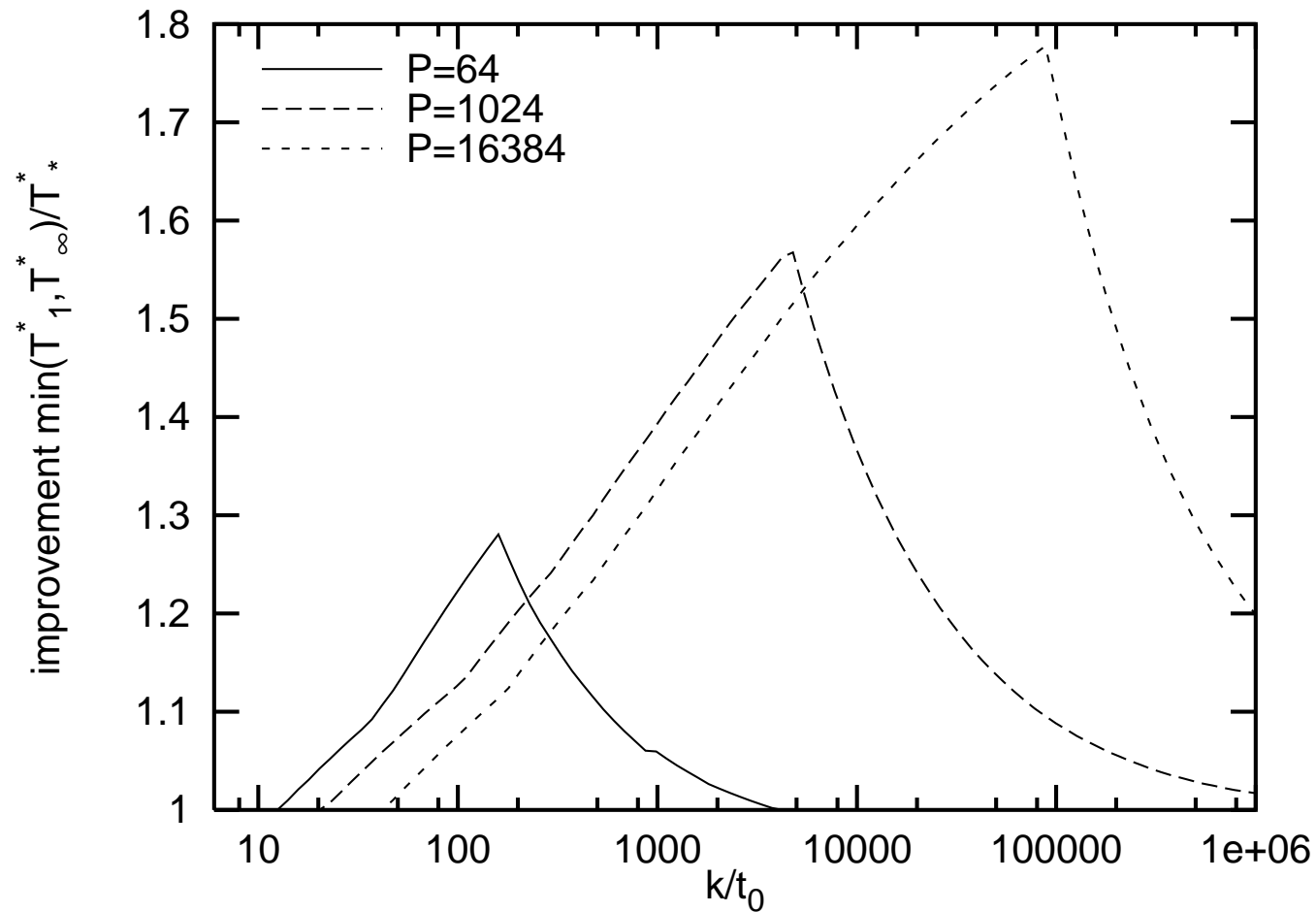
Choose unit to eliminate a parameter?



length k fractional tree broadcasting. latency $t_0 + k$

x Axis

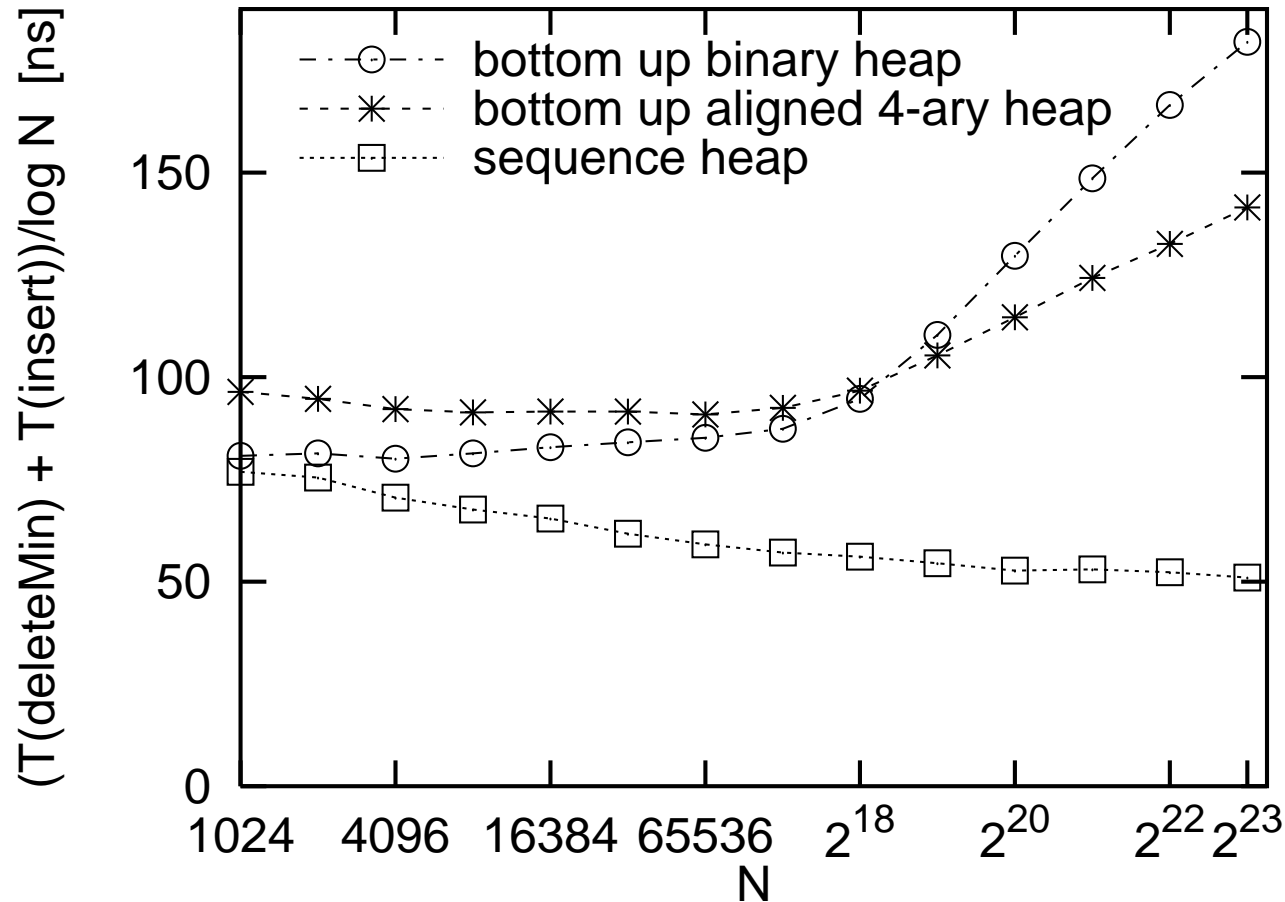
logarithmic scale?



yes if x range is wide

x Axis

logarithmic scale, powers of two (or $\sqrt{2}$)



with tic marks, (plus a few small ones)

gnuplot

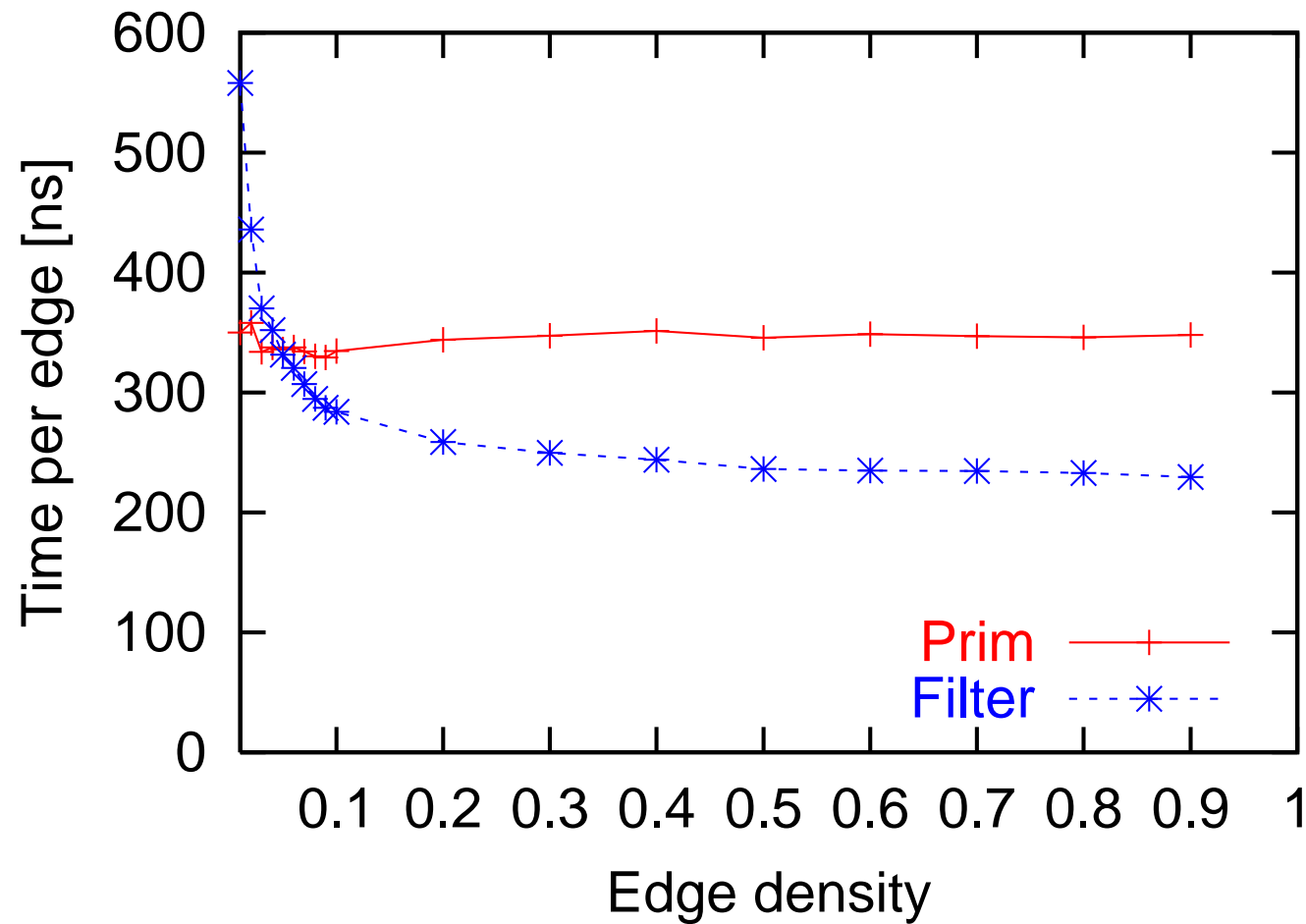
```
set xlabel "N"
set ylabel "(time per operation)/log N [ns]"
set xtics (256, 1024, 4096, 16384, 65536, "2^{18}" 262144)
set size 0.66, 0.33
set logscale x 2
set data style linespoints
set key left
set terminal postscript portrait enhanced 10
set output "r10000timenew.eps"
plot [1024:10000000][0:220] \
    "h2r10000new.log" using 1:3 title "bottom up binary heap"
    "h4r10000new.log" using 1:3 title "bottom up aligned 4-a"
    "knr10000new.log" using 1:3 title "sequence heap" with l
```

Data File

```
256 703.125 87.8906
512 729.167 81.0185
1024 768.229 76.8229
2048 830.078 75.4616
4096 846.354 70.5295
8192 878.906 67.6082
16384 915.527 65.3948
32768 925.7 61.7133
65536 946.045 59.1278
131072 971.476 57.1457
262144 1009.62 56.0902
524288 1035.69 54.51
1048576 1055.08 52.7541
2097152 1113.73 53.0349
4194304 1150.29 52.2859
8388608 1172.62 50.9836
```

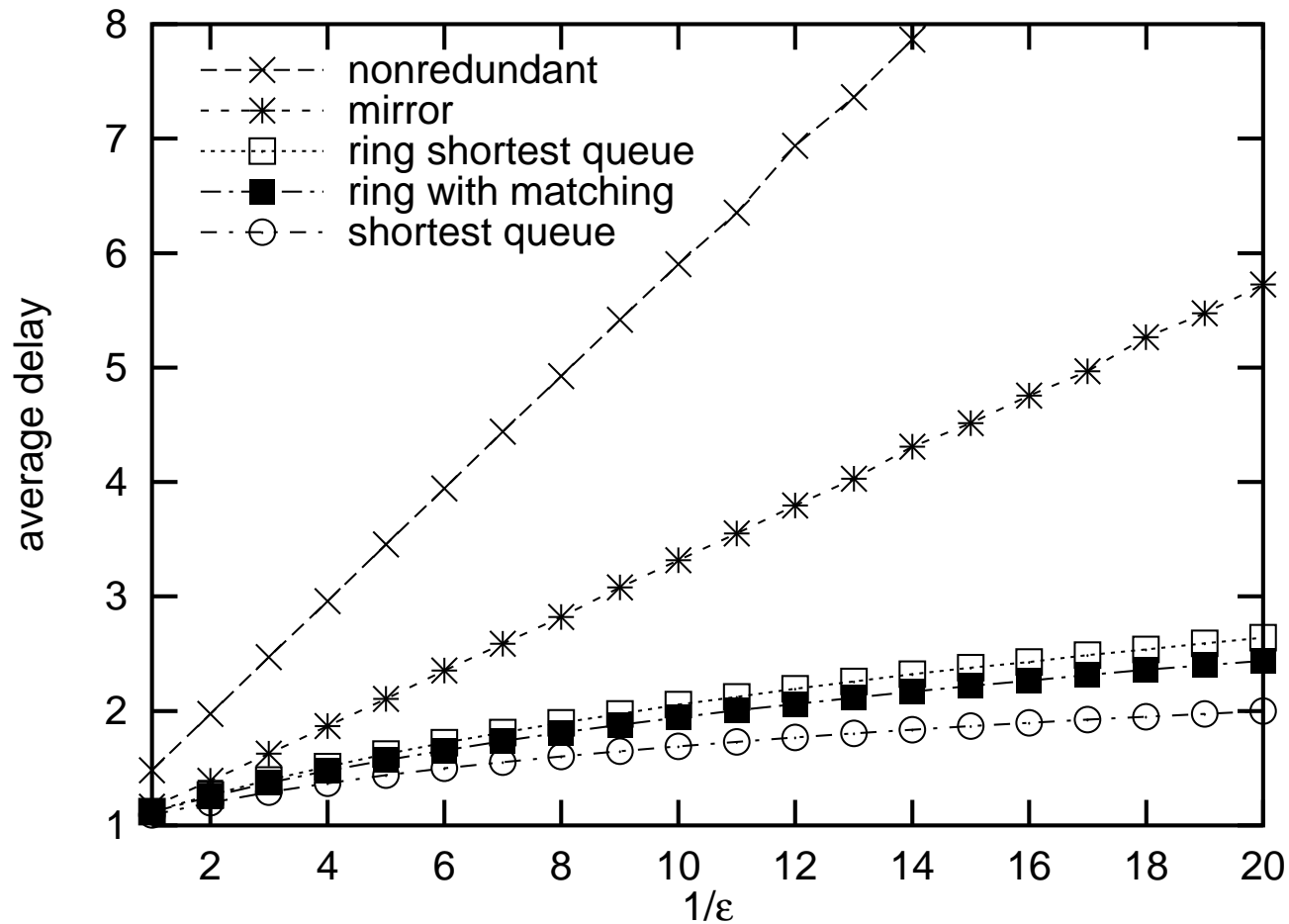
x Axis

linear scale for ratios or small ranges (#processor,...)



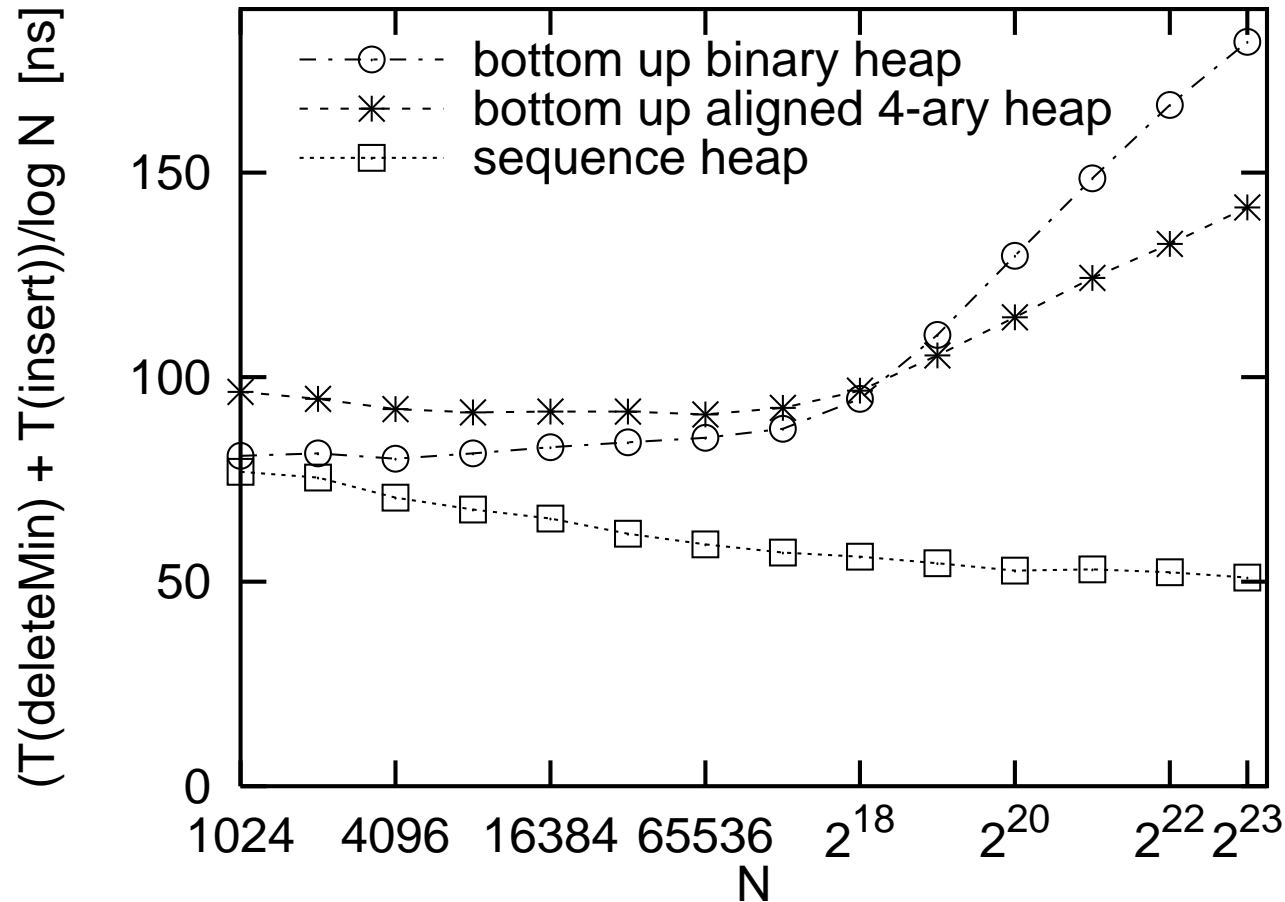
x Axis

An exotic scale: arrival rate $1 - \epsilon$ of saturation point



y Axis

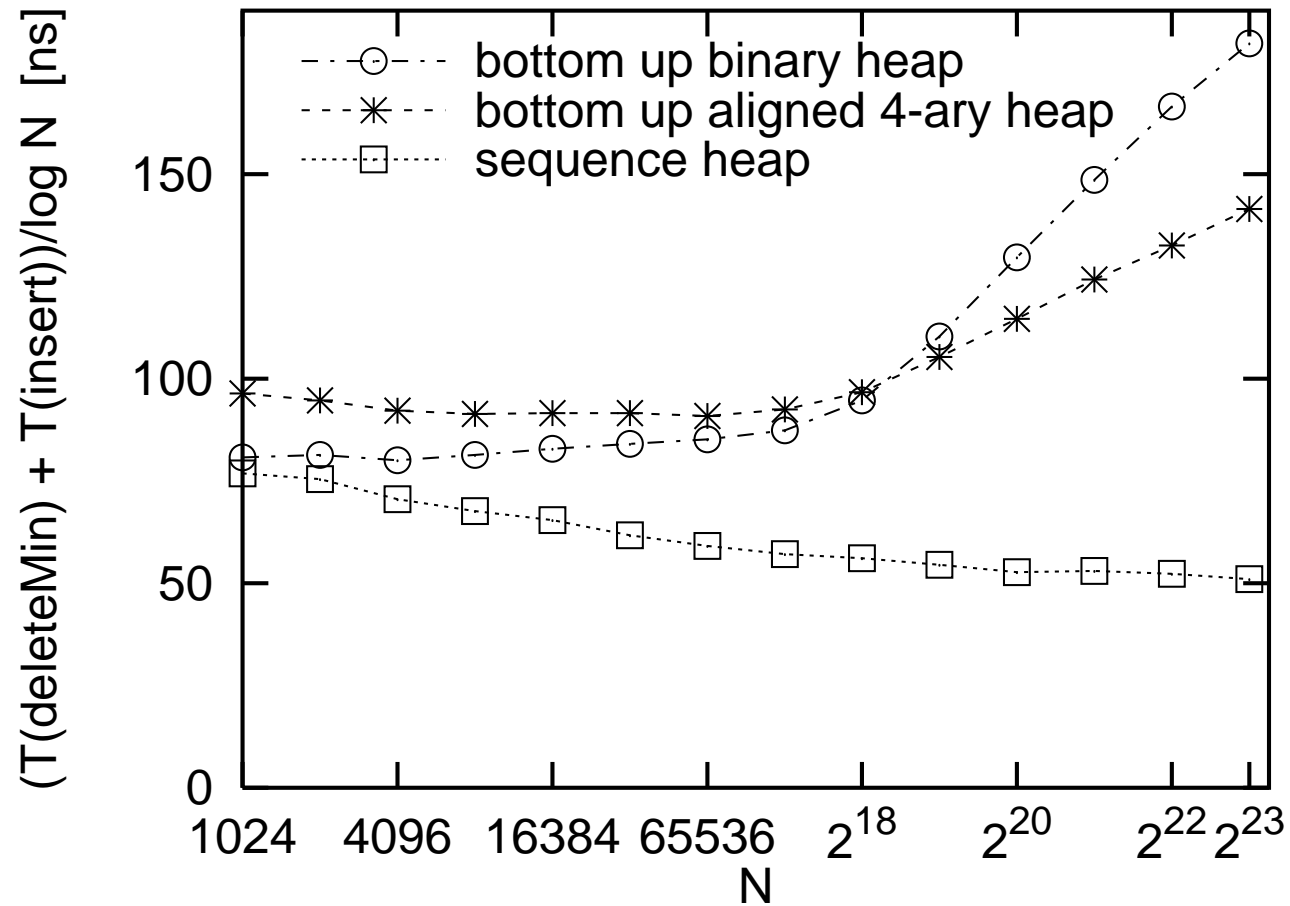
Avoid log scale ! scale such that theory gives \approx horizontal lines



but give easy interpretation of the scaling function

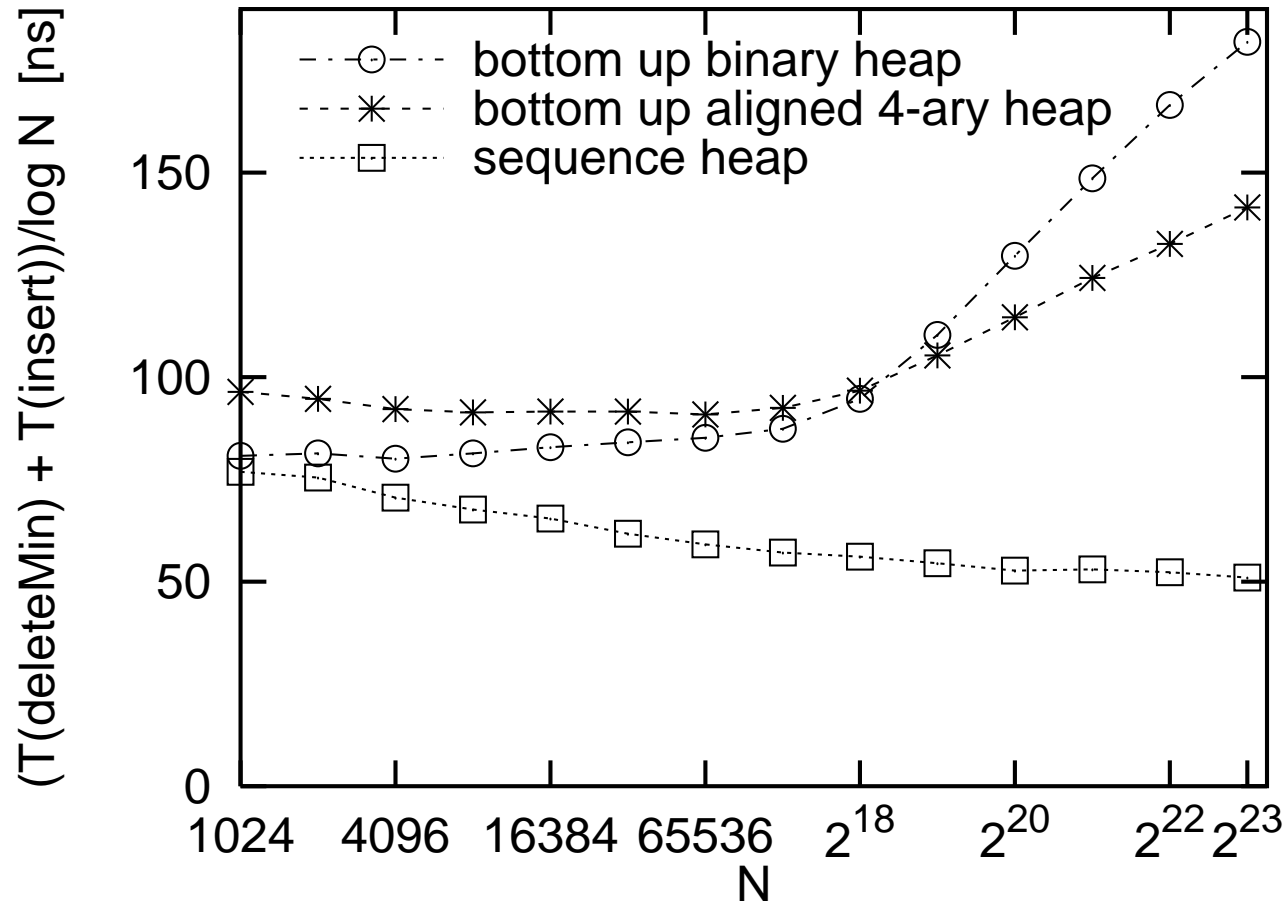
y Axis

give units



y Axis

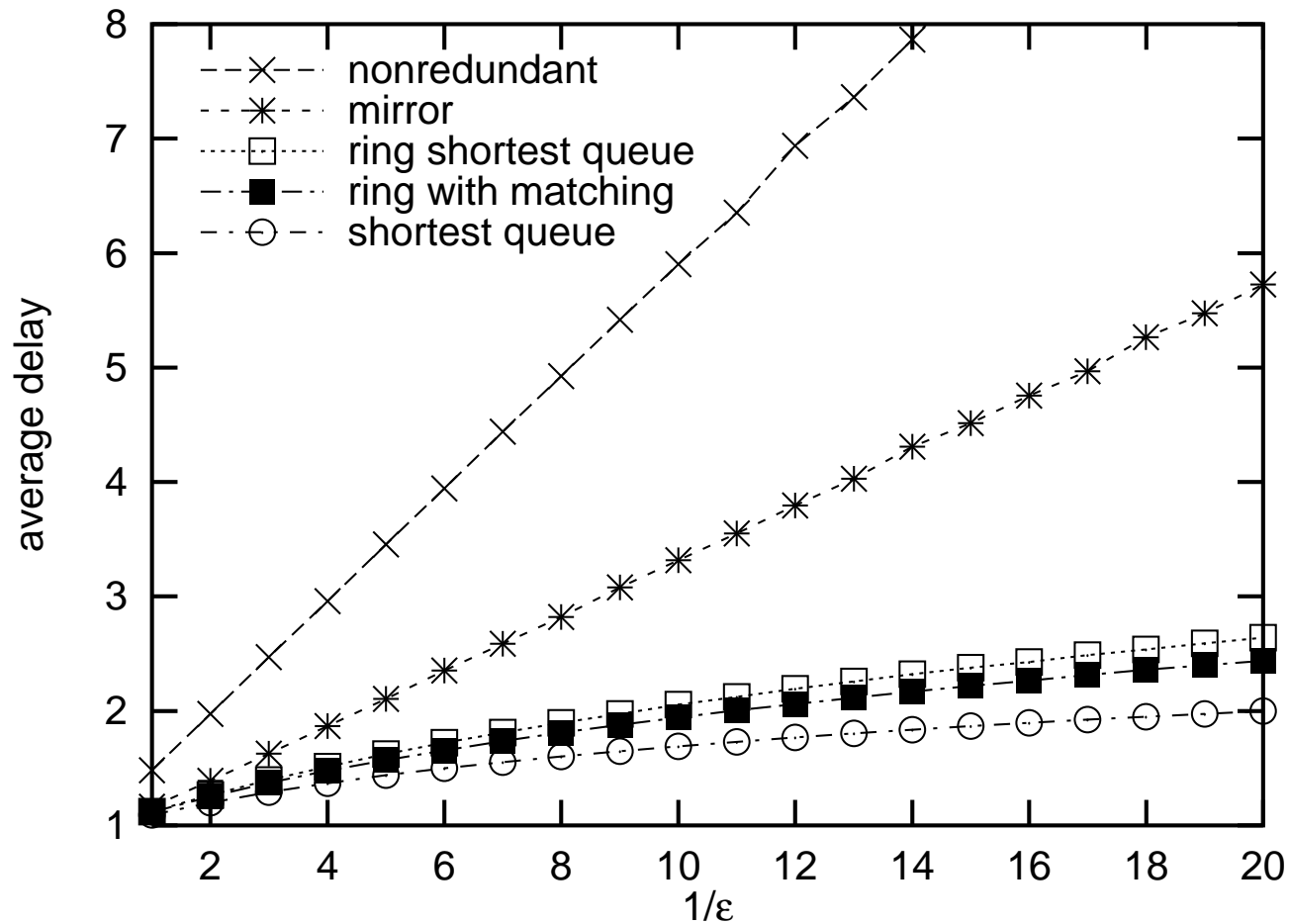
start from 0 **if** this does not waste too much space



you may assume readers to be out of Kindergarten

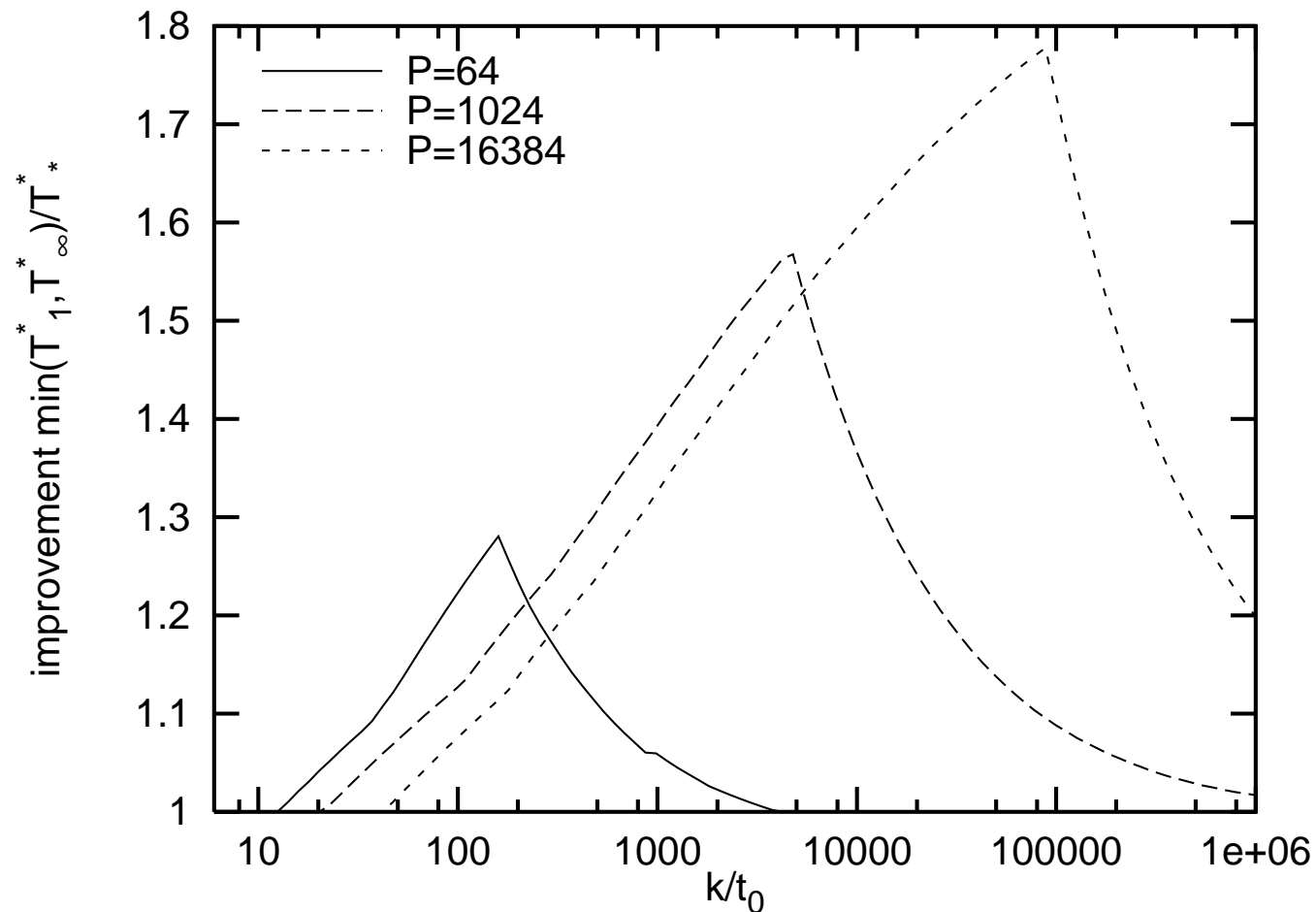
y Axis

clip outclassed algorithms



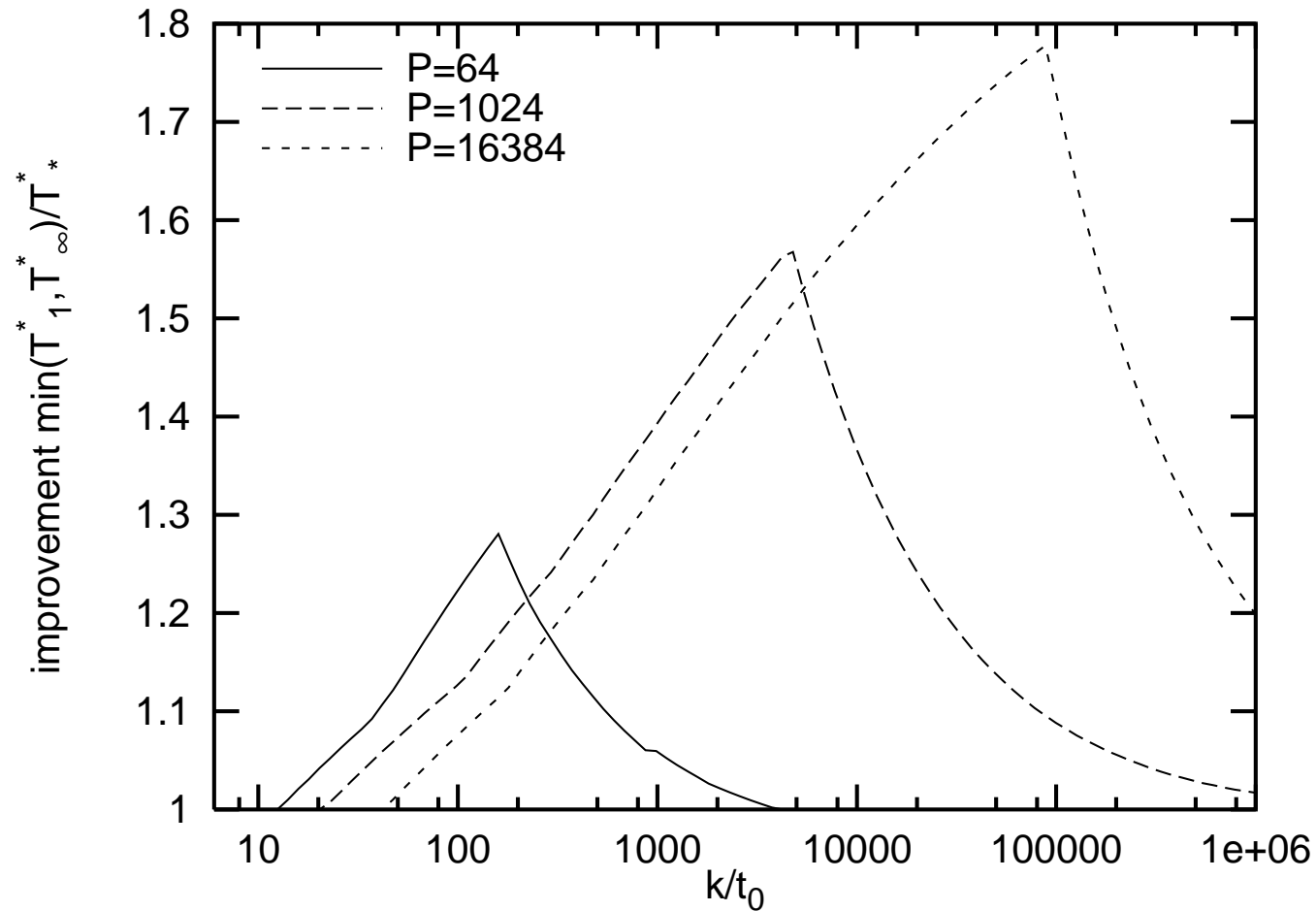
y Axis

vertical size: weighted average of the slants of the line segments
in the figure should be about 45° [Cleveland 94]



y Axis

graph a bit wider than high, e.g., golden ratio [Tufte 83]



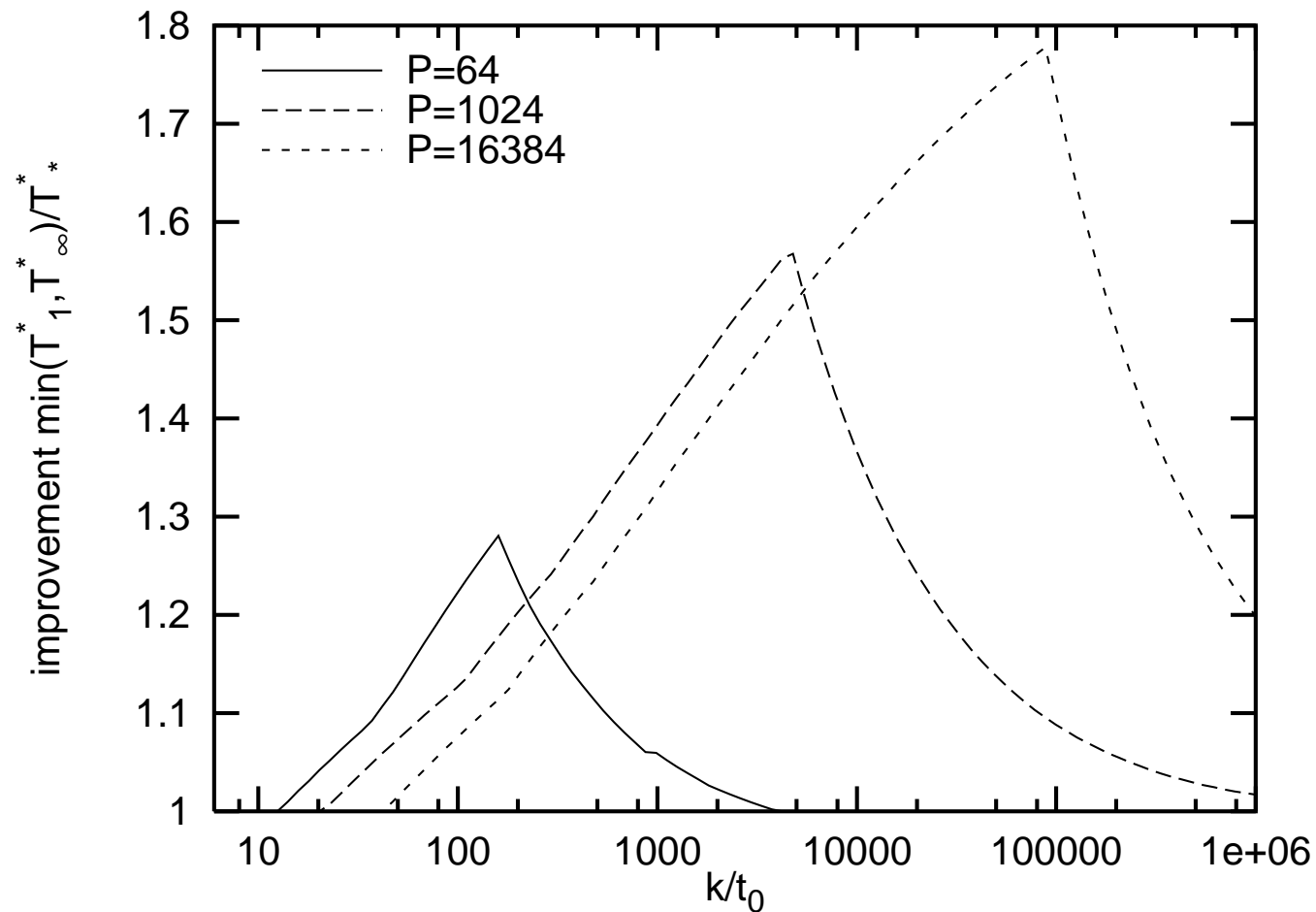
Multiple Curves

- + high information density
- + better than 3D (reading off values)
- Easily overdone

≤ 7 smooth curves

Reducing the Number of Curves

use ratios



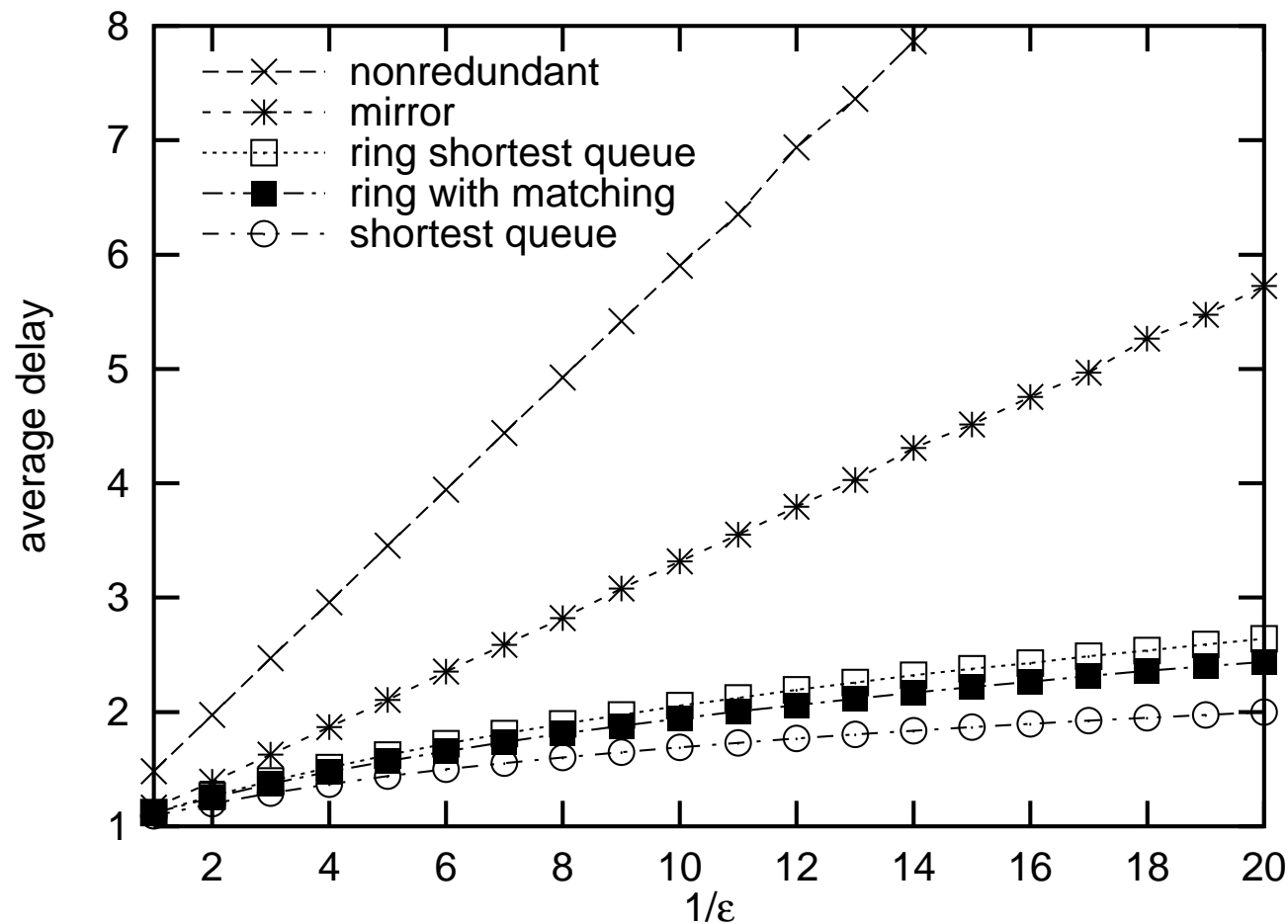
Reducing the Number of Curves

omit curves

- outclassed algorithms (for case shown)
- equivalent algorithms (for case shown)

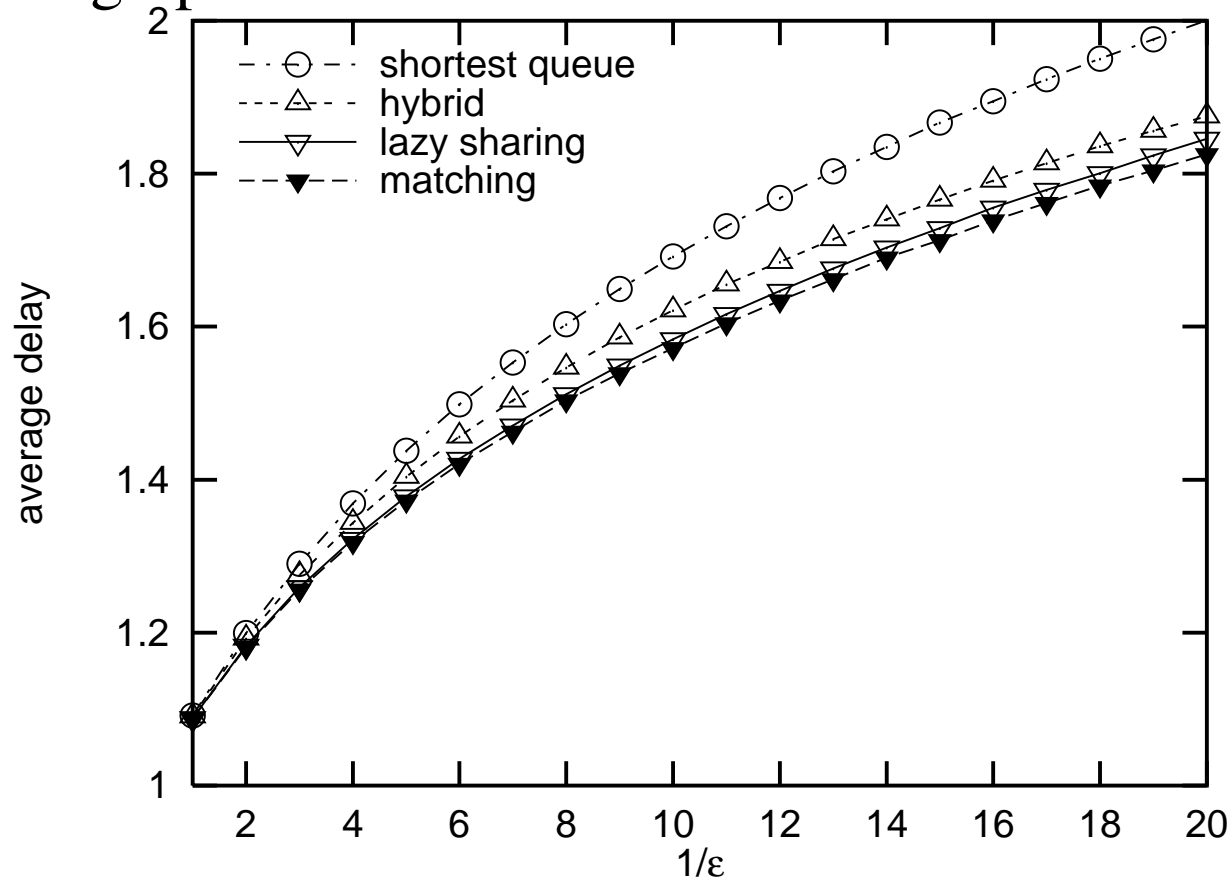
Reducing the Number of Curves

split into two graphs

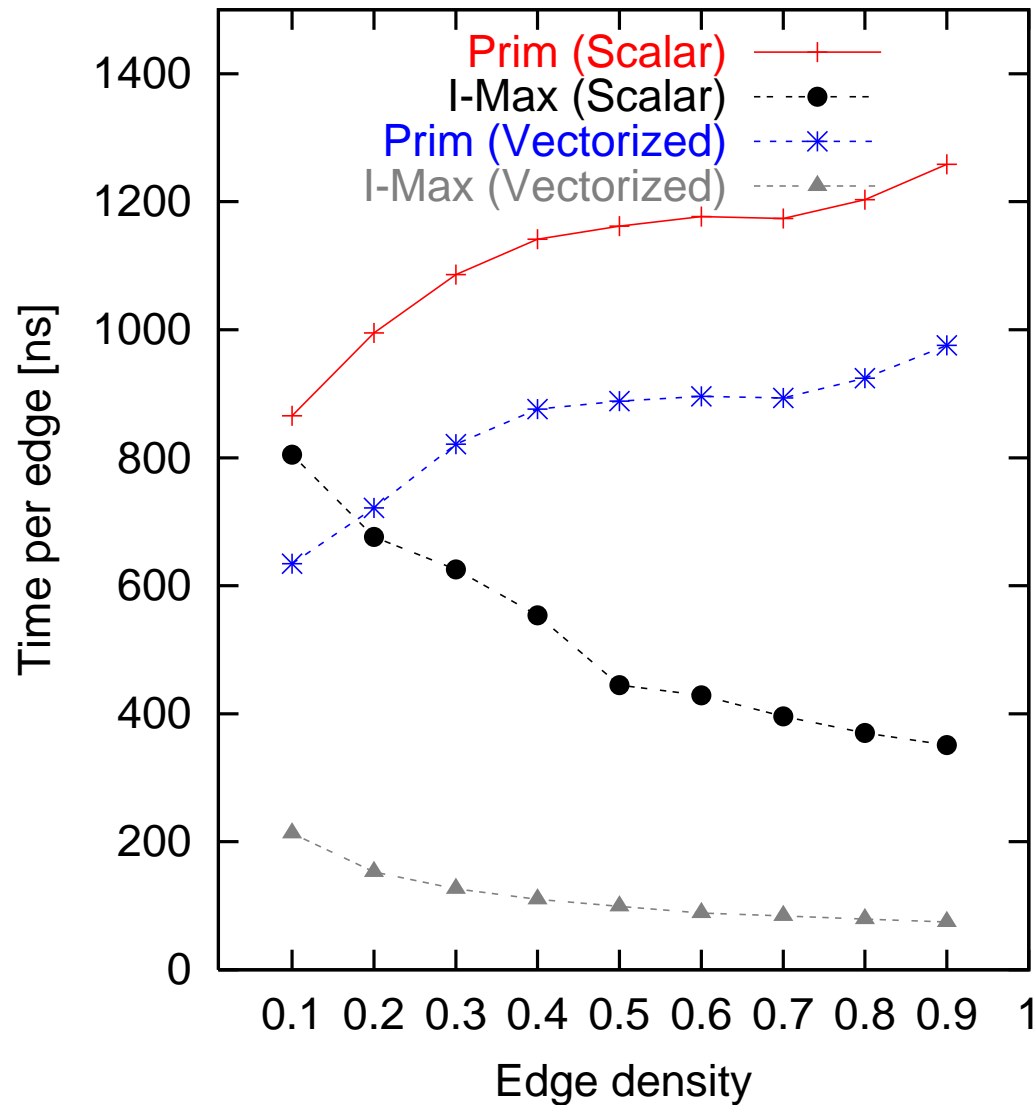


Reducing the Number of Curves

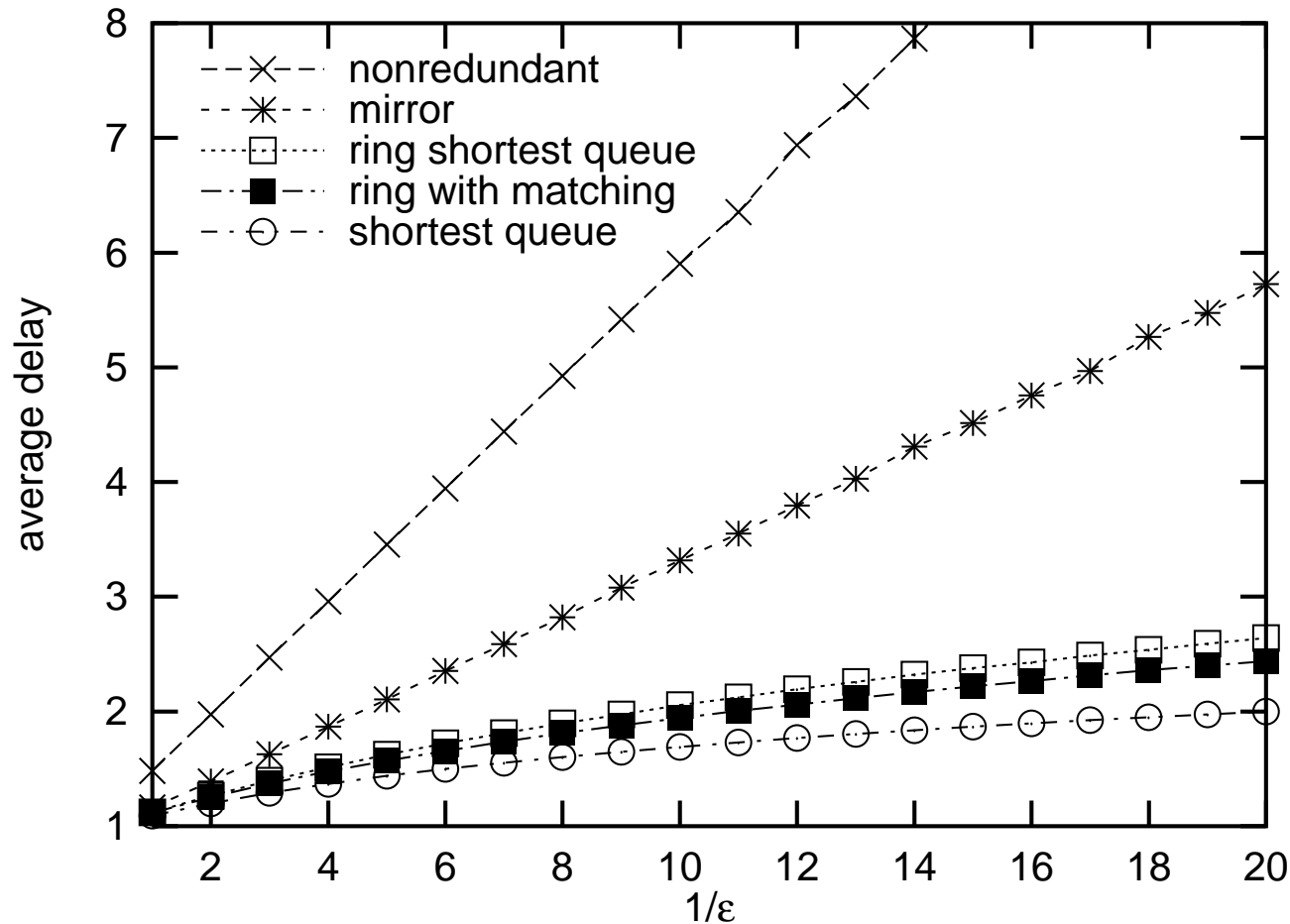
split into two graphs



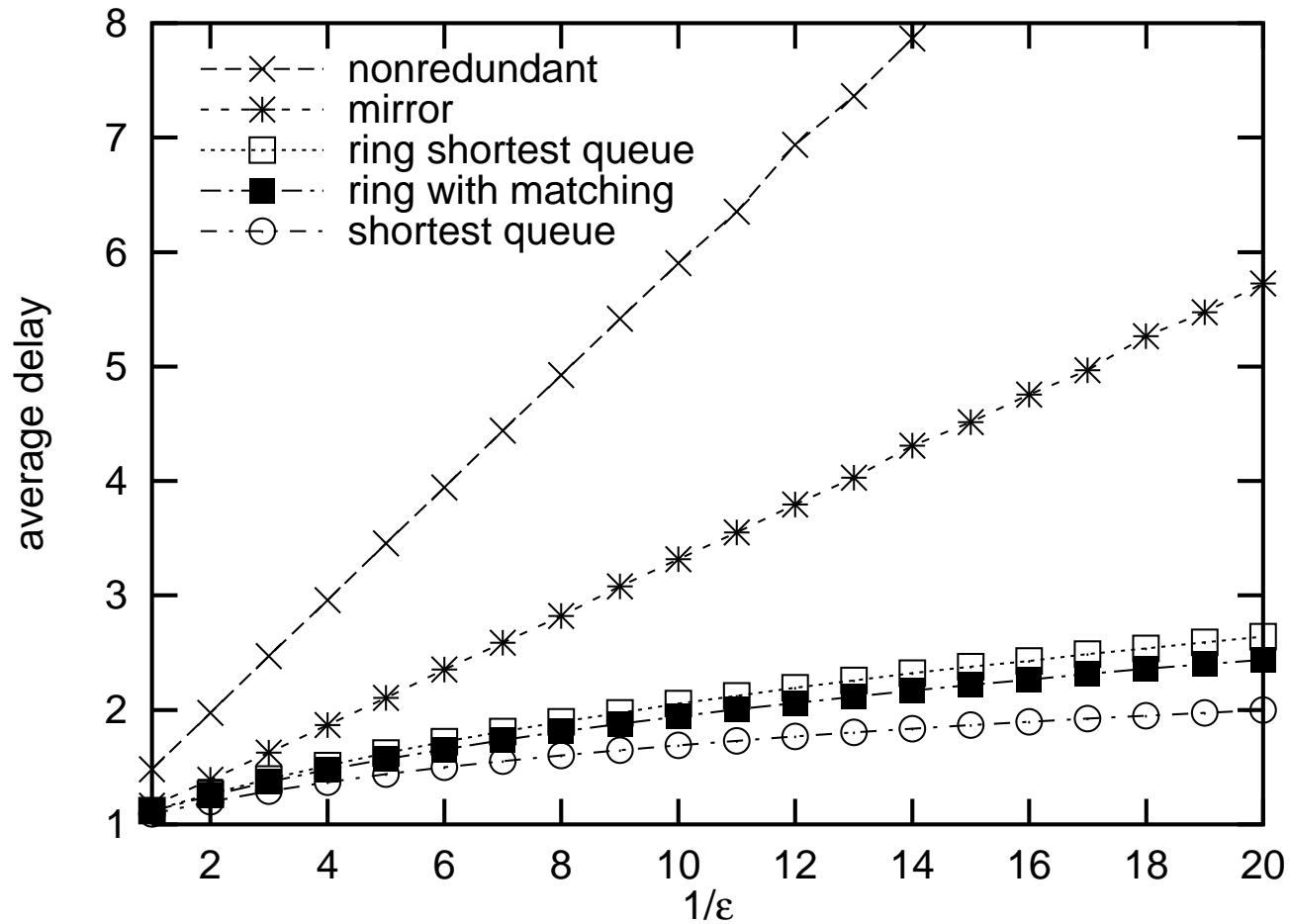
Keeping Curves apart: log y scale



Keeping Curves apart: smoothing



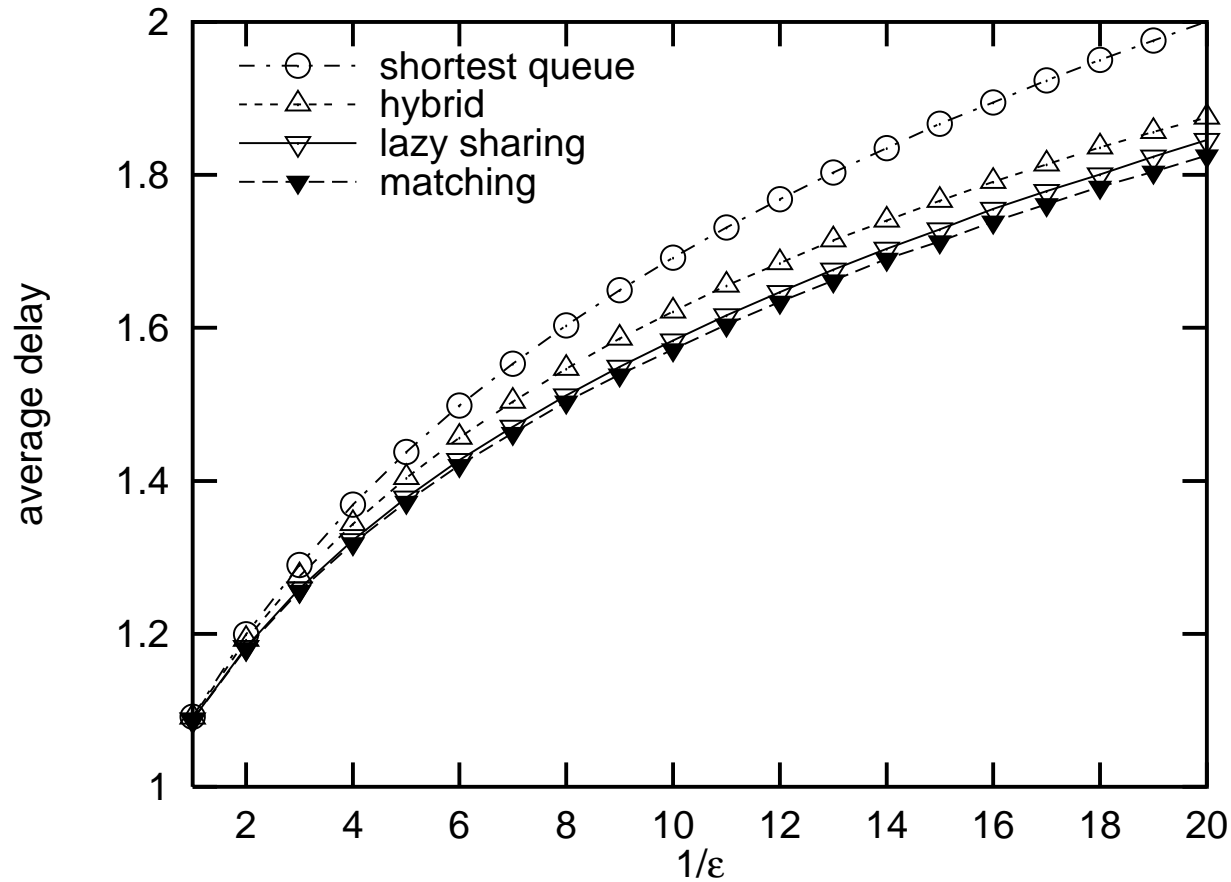
Keys



same order as curves

Keys

place in white space



consistent in different figures

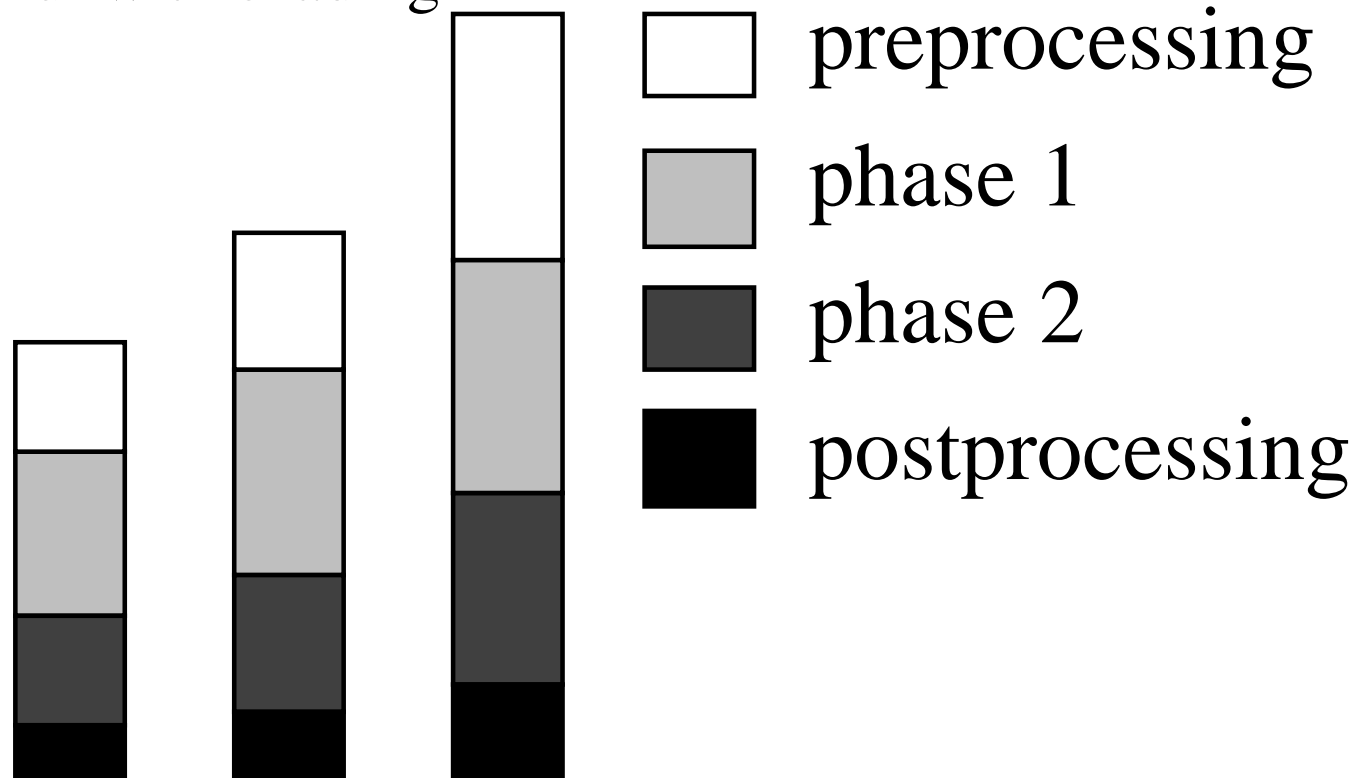
Todsünden

1. forget explaining the **axes**
2. **connecting unrelated** points by lines
3. mindless use/overinterpretation of **double-log plot**
4. cryptic **abbreviations**
5. microscopic **lettering**
6. excessive **complexity**
7. **pie charts**



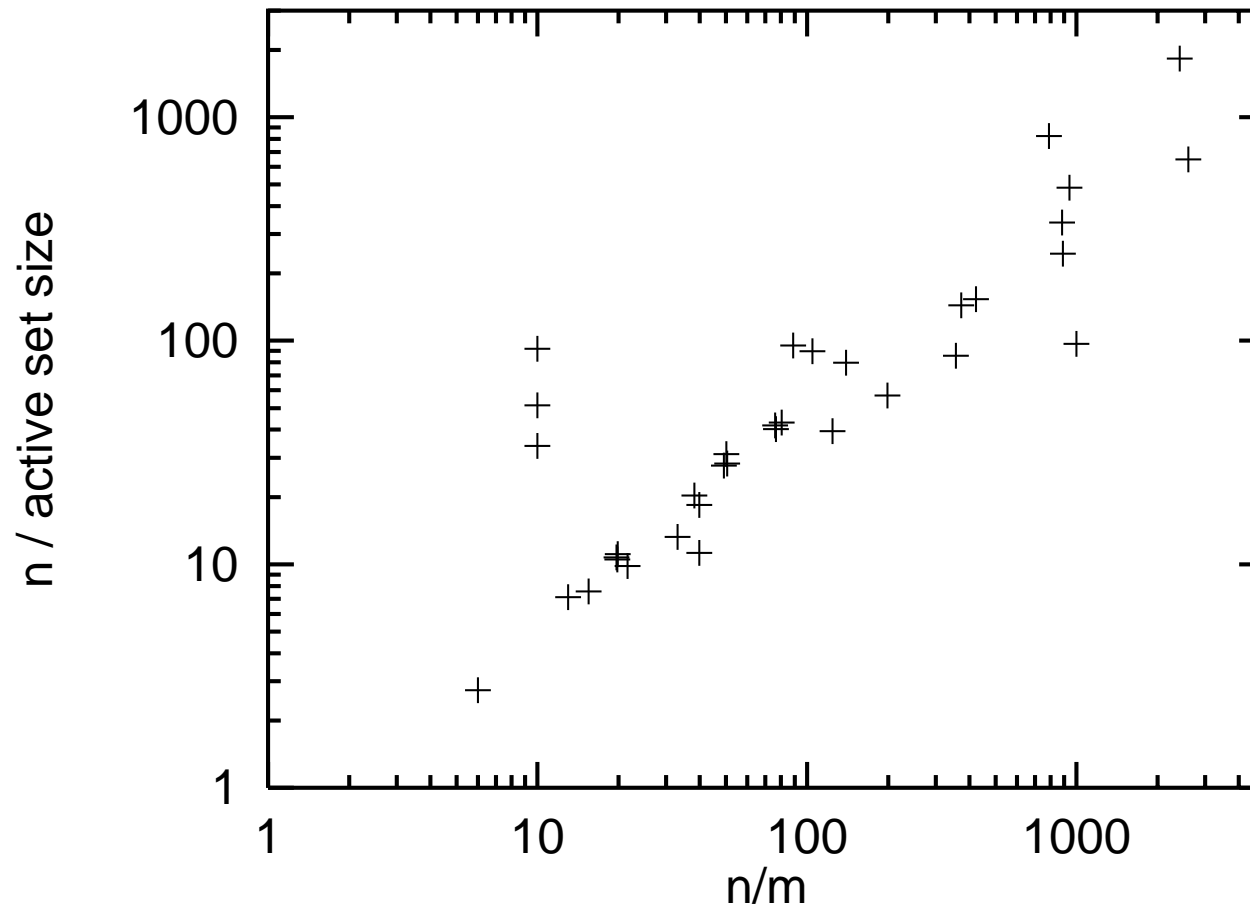
Arranging Instances

- bar charts
- stack components of execution time
- careful with shading



Arranging Instances

scatter plots

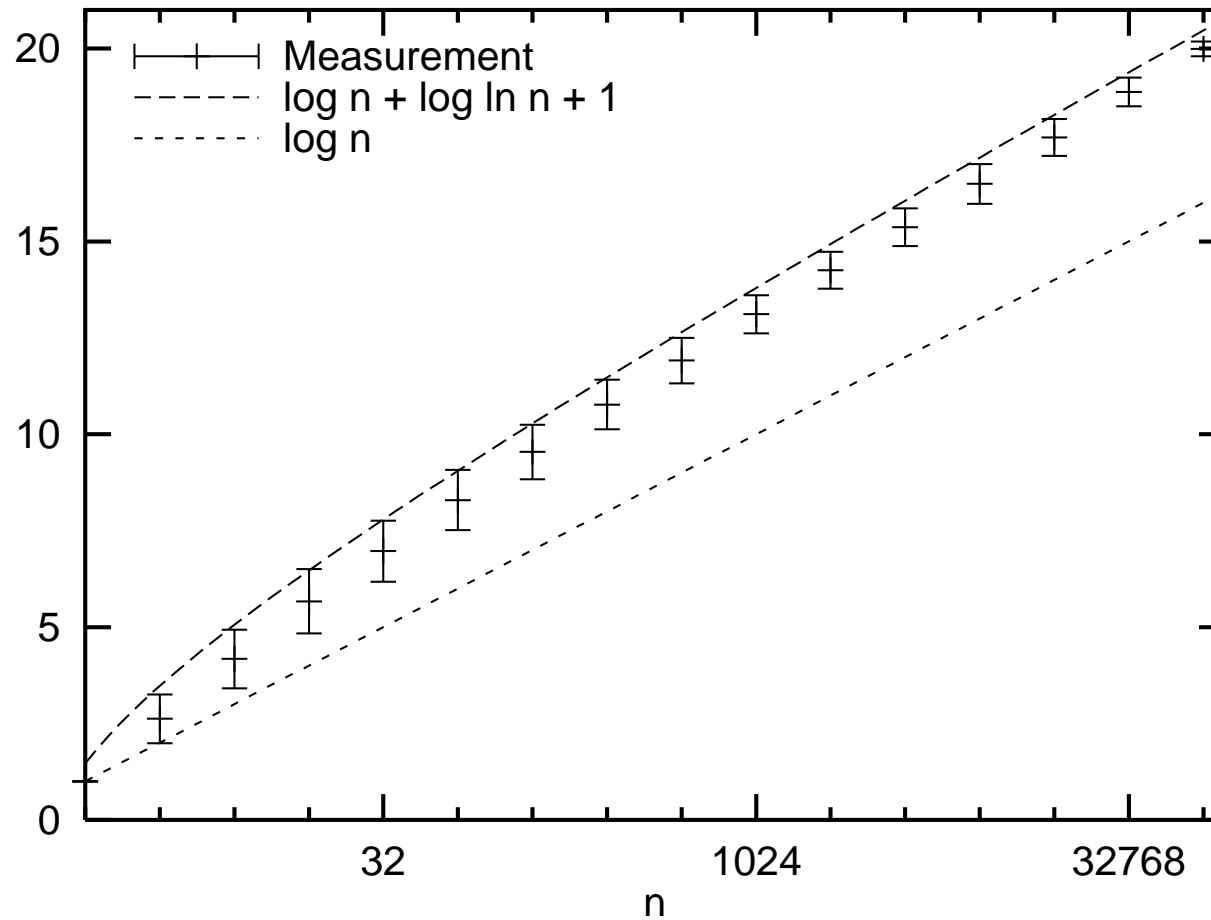


Measurements and Connections

- straight line between points do not imply claim of linear interpolation
- different with higher order curves
- no points imply an even stronger claim. Good for very dense smooth measurements.

Grids and Ticks

- Avoid grids or make it light gray
- usually round numbers for tic marks!
- sometimes plot important values on the axis



errors may **not** be of statistical nature!

3D

- you cannot read off absolute values
- interesting parts may be hidden
- only one surface
- + good impression of shape

Caption

what is displayed

how has the data been obtained

surrounding text has more.

Check List

- Should the experimental setup from the exploratory phase be redesigned to increase conciseness or accuracy?
- What parameters should be varied? What variables should be measured? How are parameters chosen that cannot be varied?
- Can tables be converted into curves, bar charts, scatter plots or any other useful graphics?
- Should tables be added in an appendix or on a web page?
- Should a 3D-plot be replaced by collections of 2D-curves?
- Can we reduce the number of curves to be displayed?
- How many figures are needed?

- Scale the x -axis to make y -values independent of some parameters?
- Should the x -axis have a logarithmic scale? If so, do the x -values used for measuring have the same basis as the tick marks?
- Should the x -axis be transformed to magnify interesting subranges?
- Is the range of x -values adequate?
- Do we have measurements for the right x -values, i.e., nowhere too dense or too sparse?
- Should the y -axis be transformed to make the interesting part of the data more visible?
- Should the y -axis have a logarithmic scale?

- Is it be misleading to start the y -range at the smallest measured value?
- Clip the range of y -values to exclude useless parts of curves?
- Can we use banking to 45° ?
- Are all curves sufficiently well separated?
- Can noise be reduced using more accurate measurements?
- Are error bars needed? If so, what should they indicate?
Remember that measurement errors are usually not random variables.
- Use points to indicate for which x -values actual data is available.
- Connect points belonging to the same curve.

- Only use splines for connecting points if interpolation is sensible.
- Do not connect points belonging to unrelated problem instances.
- Use different point and line styles for different curves.
- Use the same styles for corresponding curves in different graphs.
- Place labels defining point and line styles in the right order and without concealing the curves.
- Captions should make figures self contained.
- Give enough information to make experiments reproducible.

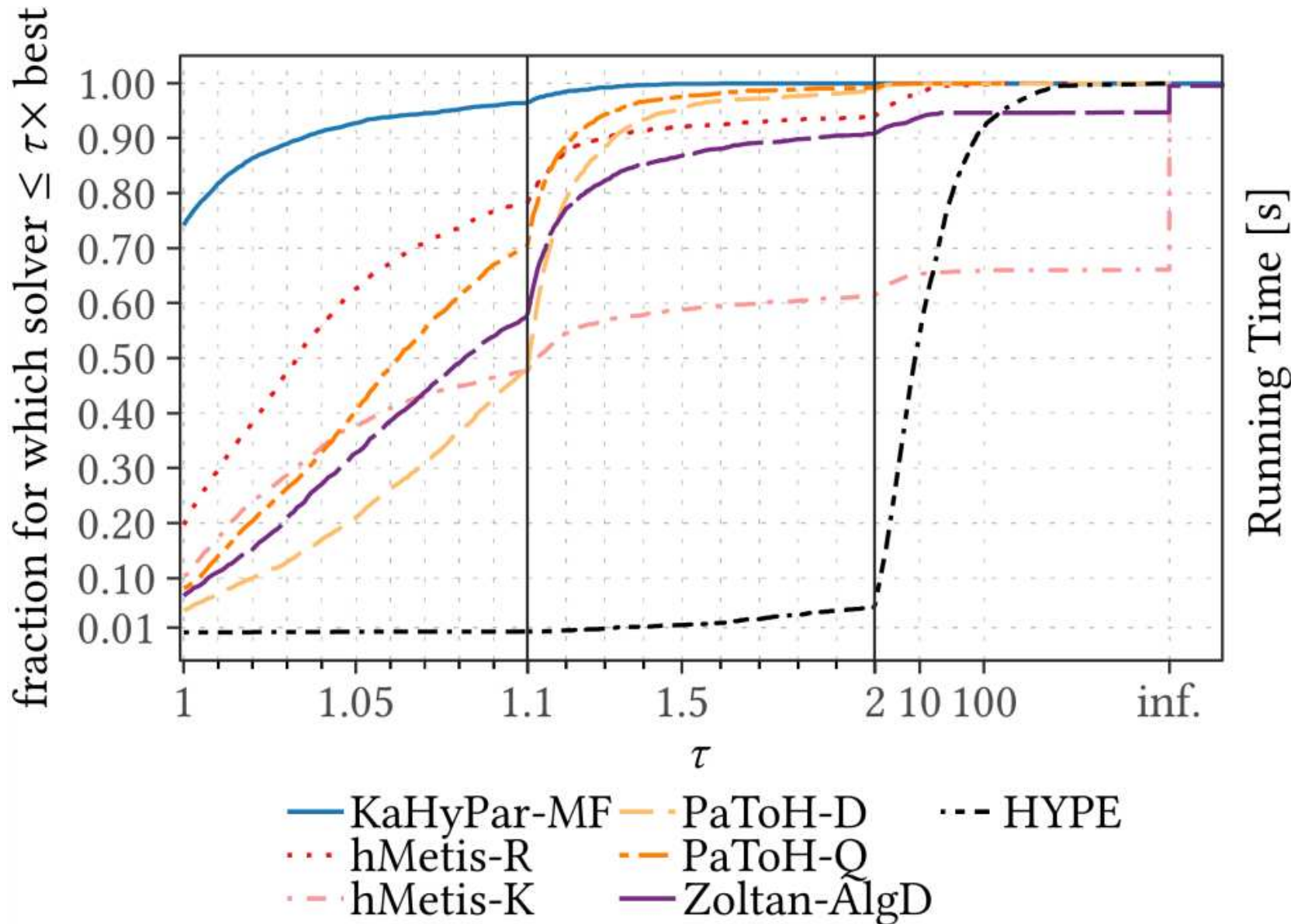
6 Comparing Apples and Oranges

In optimization problems we compare **running time** and **solution quality** for many different **instances**.

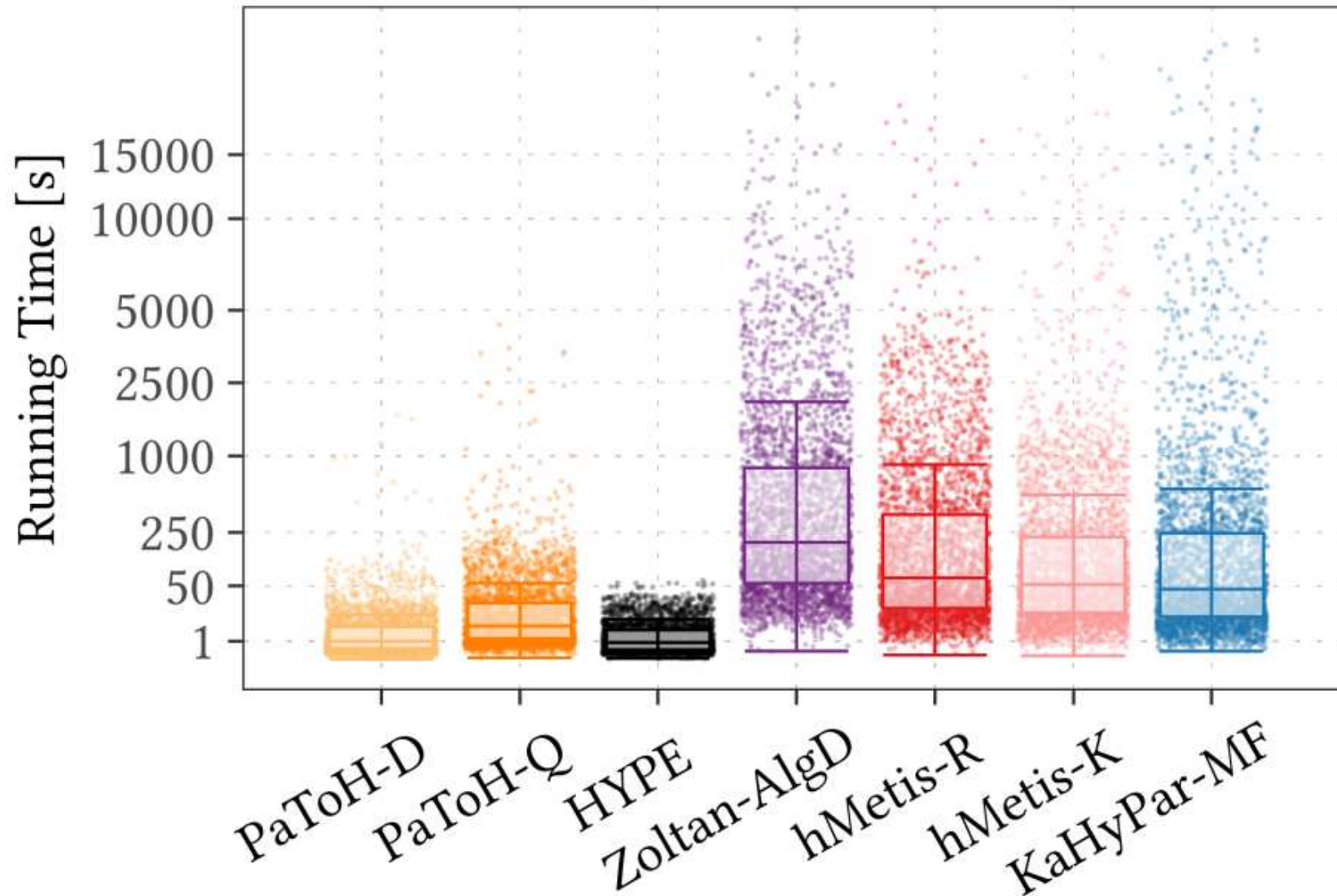
What is the better algorithm???

- Do it separately
- Quality and running time at once?

Performance Profiles (Hypergraph Partitioning)



Corresponding running times



Quality and running time at once?

We solve a special case:

- Times not too far apart
- Restarts or other means of varying time help

Idea: give both algorithms the same amount of time

Virtual instances

Compare some repetitions of algorithms A and B .

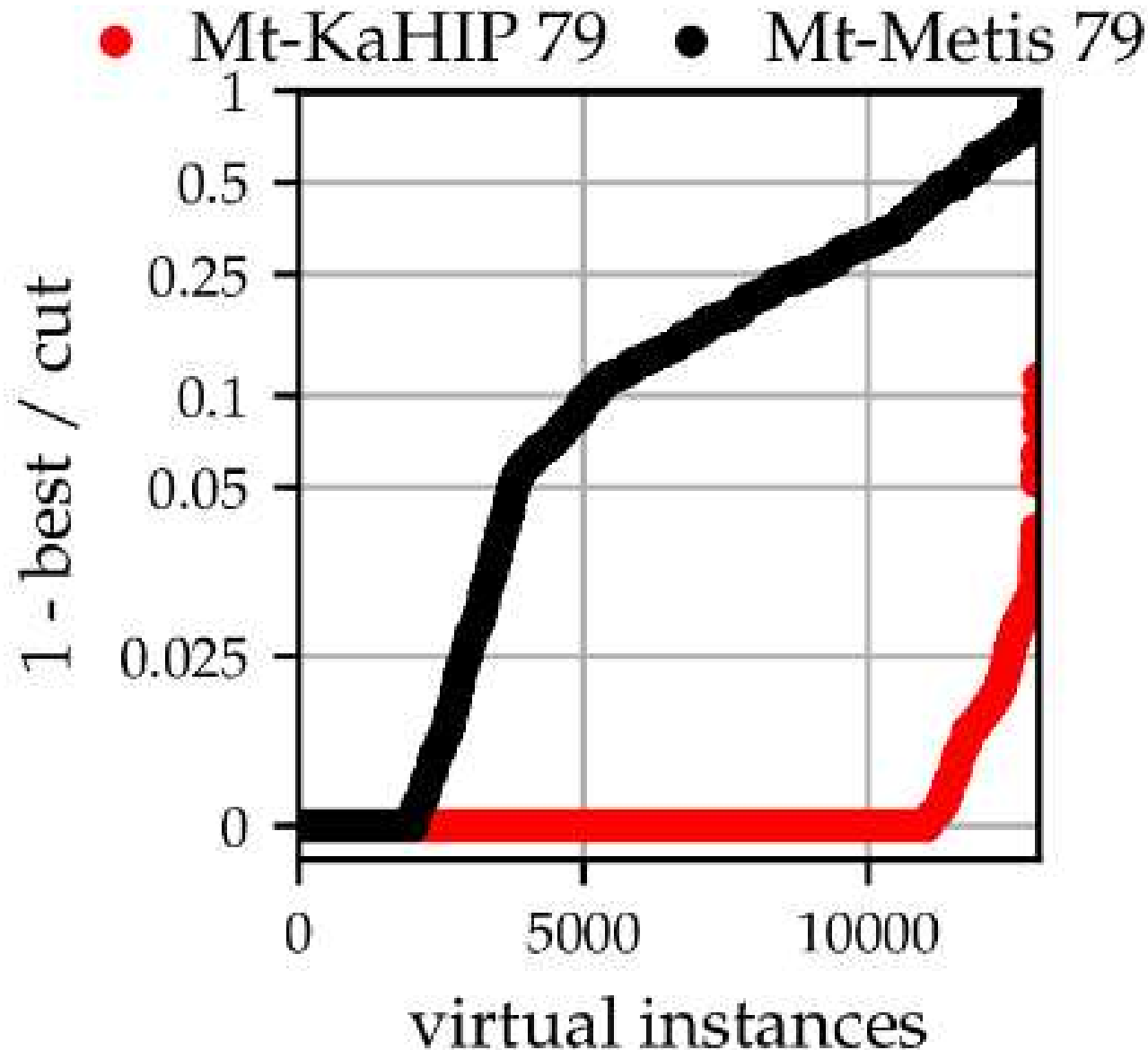
Yields several **virtual instances**

- Sample one repetition of each algorithm. Wlog assume $t_A^1 \geq t_B^1$.
- Sample (without replacement) additional repetitions of algorithm B until the total running time accumulated for algorithm B exceeds t_A^1 .
- Accept the last sample with probability

$$\frac{t_A^1 - \sum_{1 < i < \ell} t_B^i}{t_B^\ell}$$

Return first result for A and best result for B

Applied to multi-threaded graph partitioning



Literatur

- [1] Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In IEEE Int. Conf. on Big Data, 2013.
- [2] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. SIAM Journal on Computing, 34(6):1443–1463, 2005.
- [3] K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. Algorithmica, 35(1):75–93, 2003.
- [4] P. Sanders and S. Winkel. Super scalar sample sort. In 12th European Symposium on Algorithms, volume 3221 of LNCS, pages 784–796. Springer, 2004.
- [5] M. Rahn, P. Sanders, and J. Singler. Scalable distributed-memory external sorting. In 26th IEEE International Conference on Data Engineering, pages 685–688, 2010.
- [6] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. CoRR, abs/0909.5649, 2009. submitted for publication.
- [7] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In 15th ACM Symposium on Parallelism in Algorithms and Architectures, pages 138–148, San Diego, 2003.
- [8] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In 13th Euro-Par, volume 4641 of LNCS, pages 682–694. Springer, 2007.
- [9] U. Meyer, P. Sanders, and J. Sibeyn, editors. Algorithms for Memory Hierarchies, volume 2625 of LNCS Tutorial. Springer, 2003.
- [10] K. Mehlhorn and P. Sanders. Algorithms and Data Structures — The Basic Toolbox. Springer, 2008.
- [11] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL data sets. Software Practice & Experience, 38(6):589–637, 2008.
- [12] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, volume 588 of Contemporary Mathematics. AMS, 2013.

- [13] A. Beckmann, U. Meyer, P. Sanders, and J. Singler. Energy-efficient sorting using solid state disks. In 1st International Green Computing Conference, pages 191–202. IEEE, 2010.
- [14] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In 24th IEEE International Parallel and Distributed Processing Symposium, 2010. see also arXiv:0909.5649.
- [15] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In 27th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA), 2015.
- [16] Michael Axtmann and Peter Sanders. Robust massively parallel sorting. In 19th Workshop on Algorithm Engineering and Experiments (ALENEX), pages 83–97. SIAM, 2017.
- [17] P. Sanders and T. Hansch. On the efficient implementation of massively parallel quicksort. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, 4th International Symposium on Solving Irregularly Structured Problems in Parallel, number 1253 in LNCS, pages 13–24. Springer, 1997.
- [18] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort (ipssso). In 25th European Symposium on Algorithms (ESA), 2017.
- [19] Peter Sanders and Jan Wassenberg. Engineering a multi-core radix sort. In Euro-Par, volume 6853 of LNCS, pages 160–169. Springer, 2011.
- [20] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In 11th ACM-SIAM Symposium on Discrete Algorithms, pages 849–858, 2000.
- [21] K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In 14th European Symposium on Algorithms (ESA), volume 4168 of LNCS, pages 780–791, 2006.
- [22] Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. Algorithmica, pages 1–52, 2015.
- [23] Timo Bingmann, Peter Sanders, and Matthias Schimek. Communication-efficient string sorting. In 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020.
- [24] Timo Bingmann and Peter Sanders. Parallel string sample sort. In 21st European Symposium on Algorithms (ESA), volume 8125 of LNCS, pages 169–180. Springer, 2013.

- [25] M. Axtmann, A. Wiebigke, and P. Sanders. Lightweight mpi communicators with applications to perfectly balanced quicksort. In 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 254–265, May 2018.