Algorithm Engineering
Basic Toolbox Algorithms and Data Structures

Peter Sanders

What are the fastest implemented Algorithms
for the most basic algorithms:
lists, sorting, priority queues, sorted sequences, hash tables,
graph algorithms?

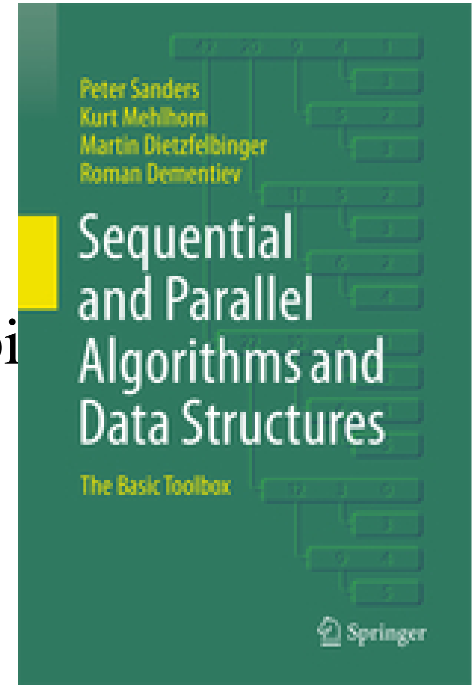
Useful Previouis Knowledge

- Algorithmen I
- Algorithmen II
- some computer architecture
- passive knowledge of C/C++

Vertiefungsgebiet: Algorithmik

Material

- Slides
- Scientific papers
  lecture homepage
- Basics: algorithms textbooks,
  z.B. Sanders et al., Cormen et al.
- Mehlhorn Näher: The LEDA Platform of Combi
  and Geometric Computing.
- Catherine McGeoch, A Guide to Experimental
  Algorithmics
- perhaps materials from a new book “Algorithm
  Engineering” Sanders et al.



Exercises

- overall 20% of the grade
- taught by Stefan Hermann and Sasch Witt
- detals later

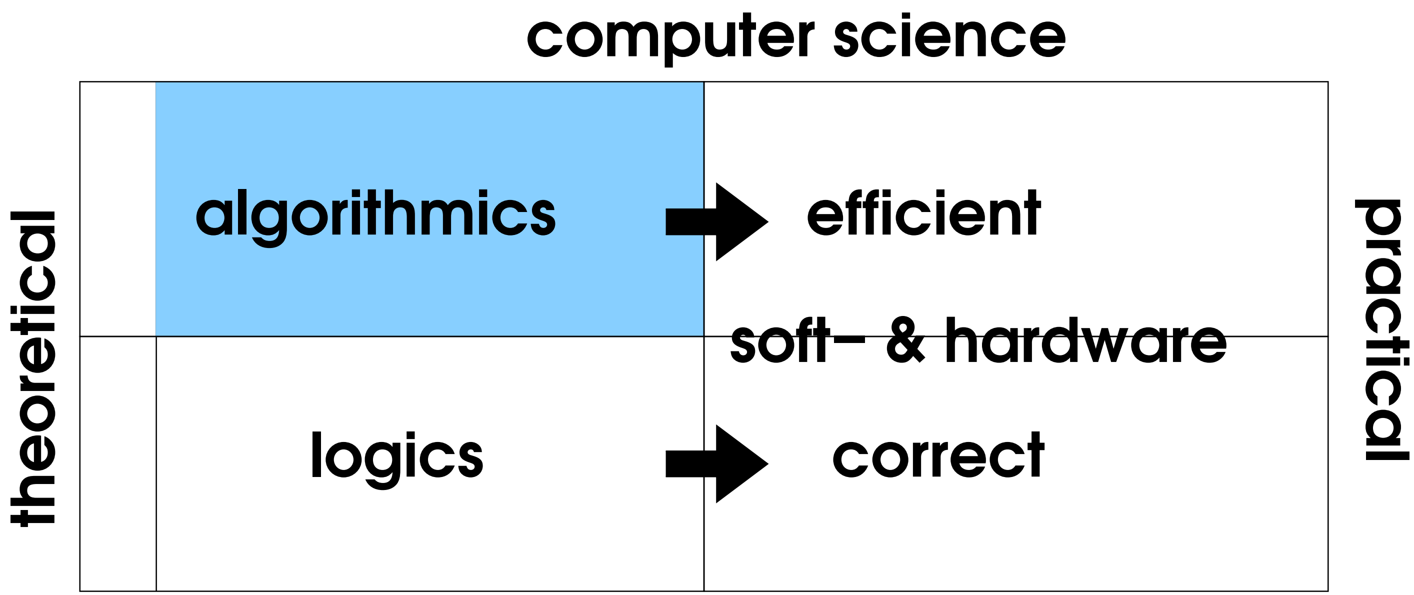
Überblick

- What is Algorithm Engineering, Modelle, ...
- First Steps: Arrays, verkettete Listen, Stacks, FIFOs,...
- Sorting
- Priority Queues
- Sortes sequences
- Hash tables
- Minimum spanning trees
- Shortest paths

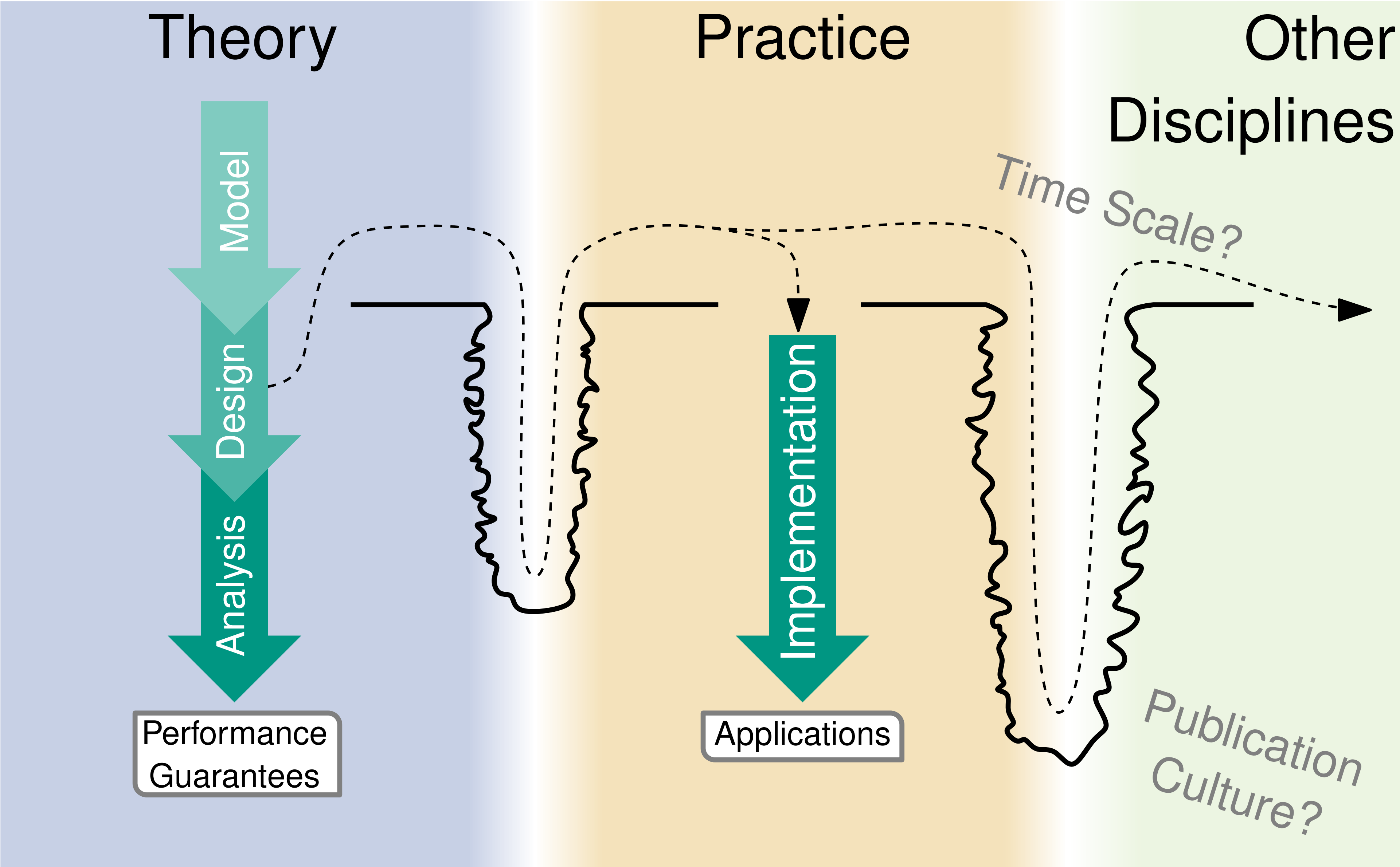
Methodology: mostly in digressions

Algorithmics

= the systematic design of efficient software and hardware

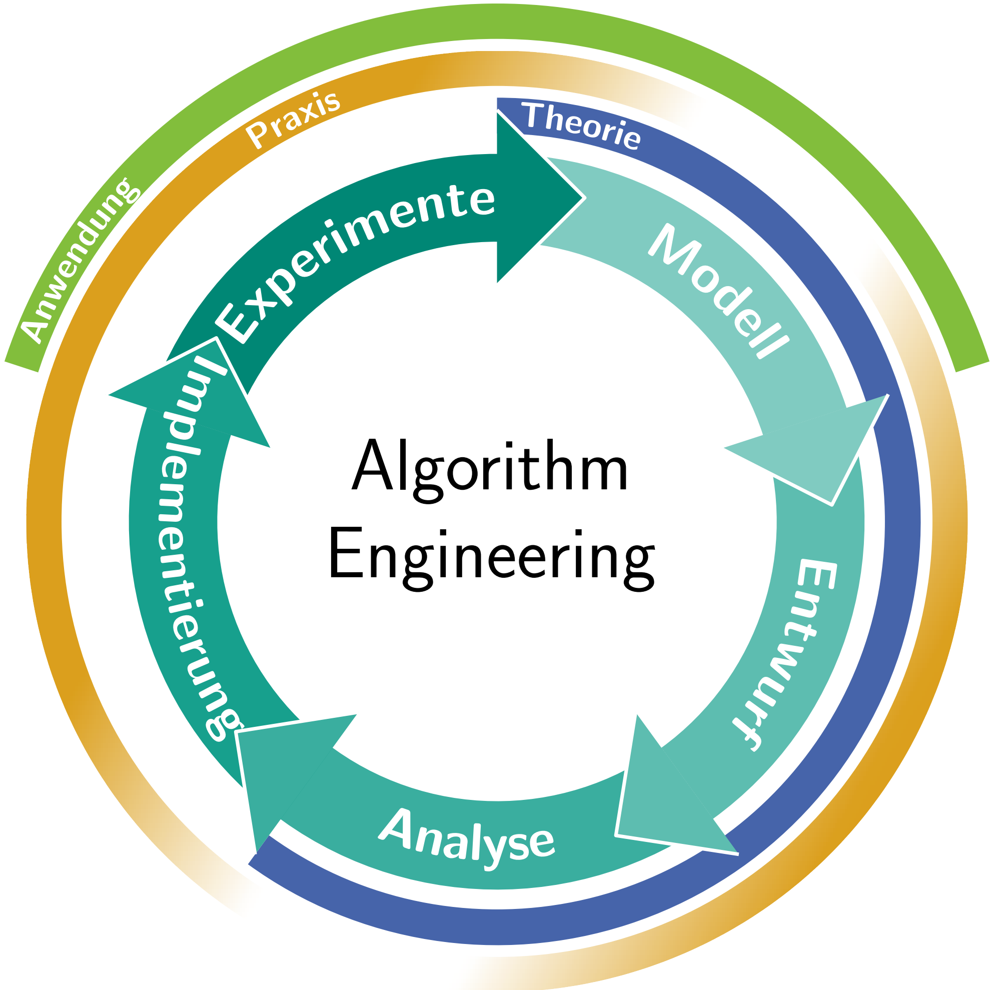


(Karikierte) traditionelle Sicht: Algorithmentheorie

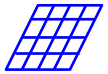
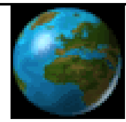
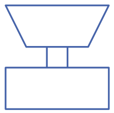
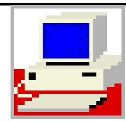
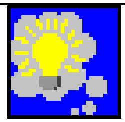
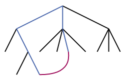
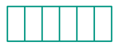
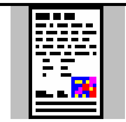


Algorithmics as Algorithm Engineering

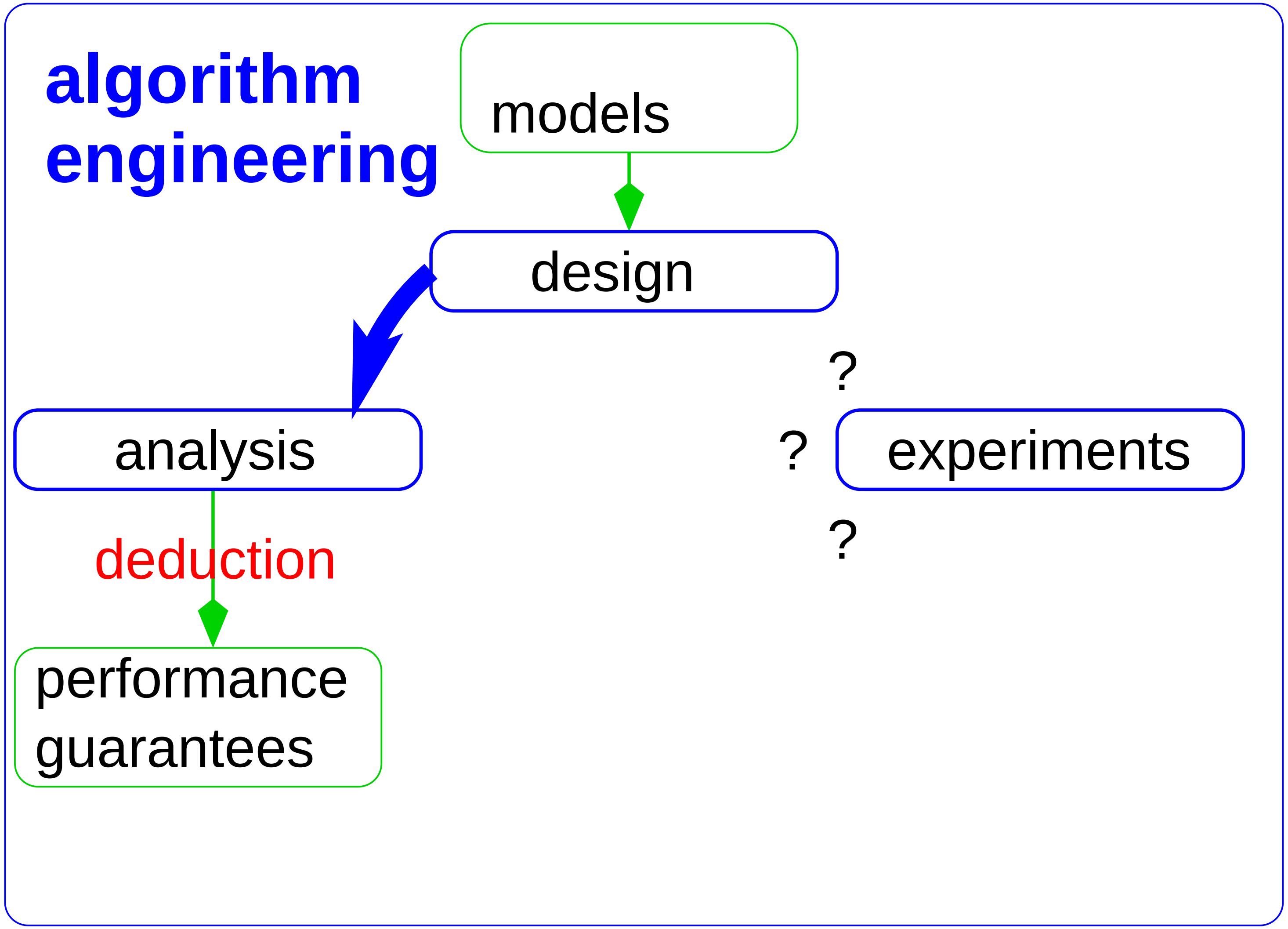
- bridge gaps
  betwee theory and practice
- integrate
  interdisziplinary
  Research



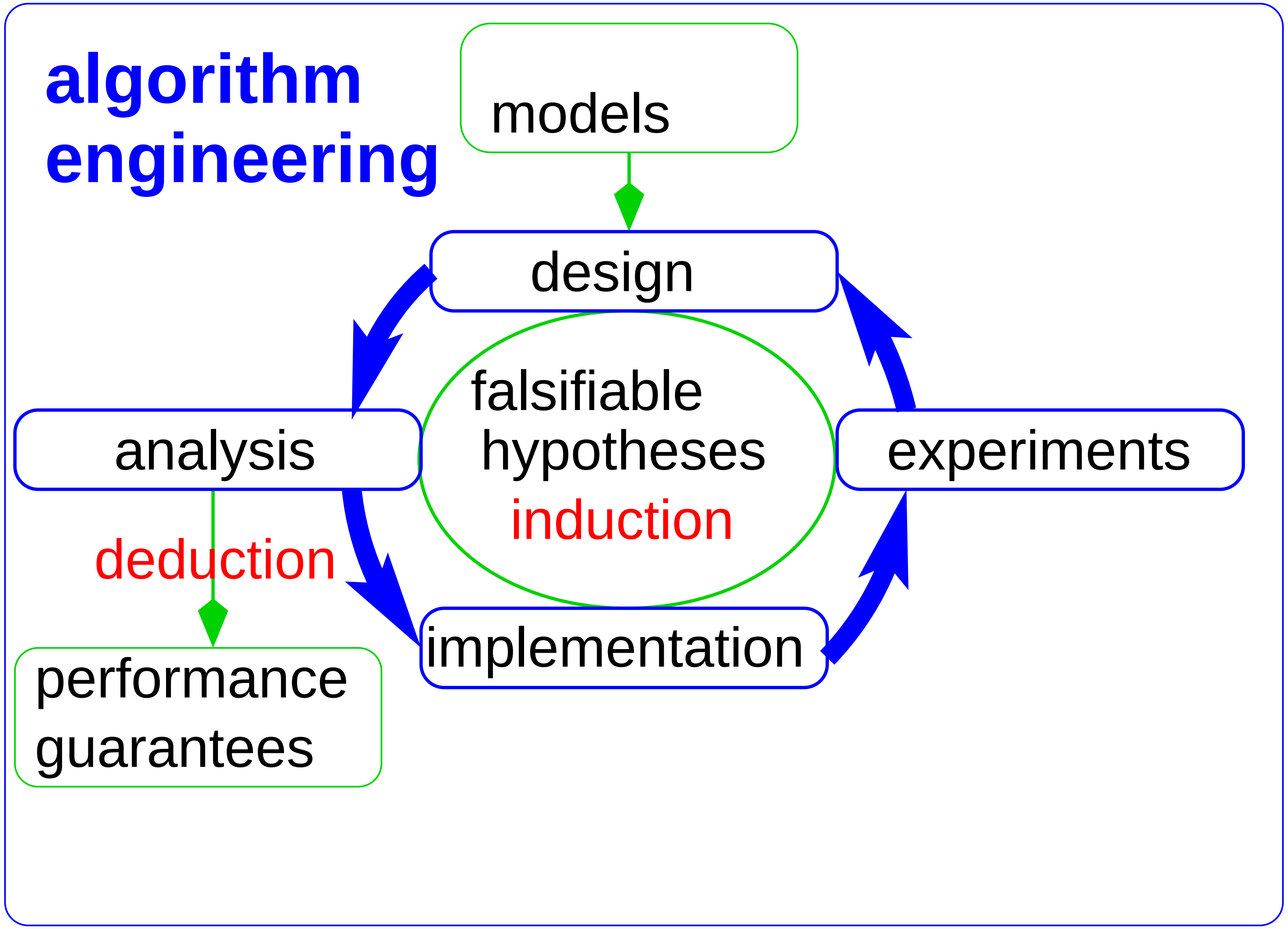
Gaps Between Theory & Practice

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

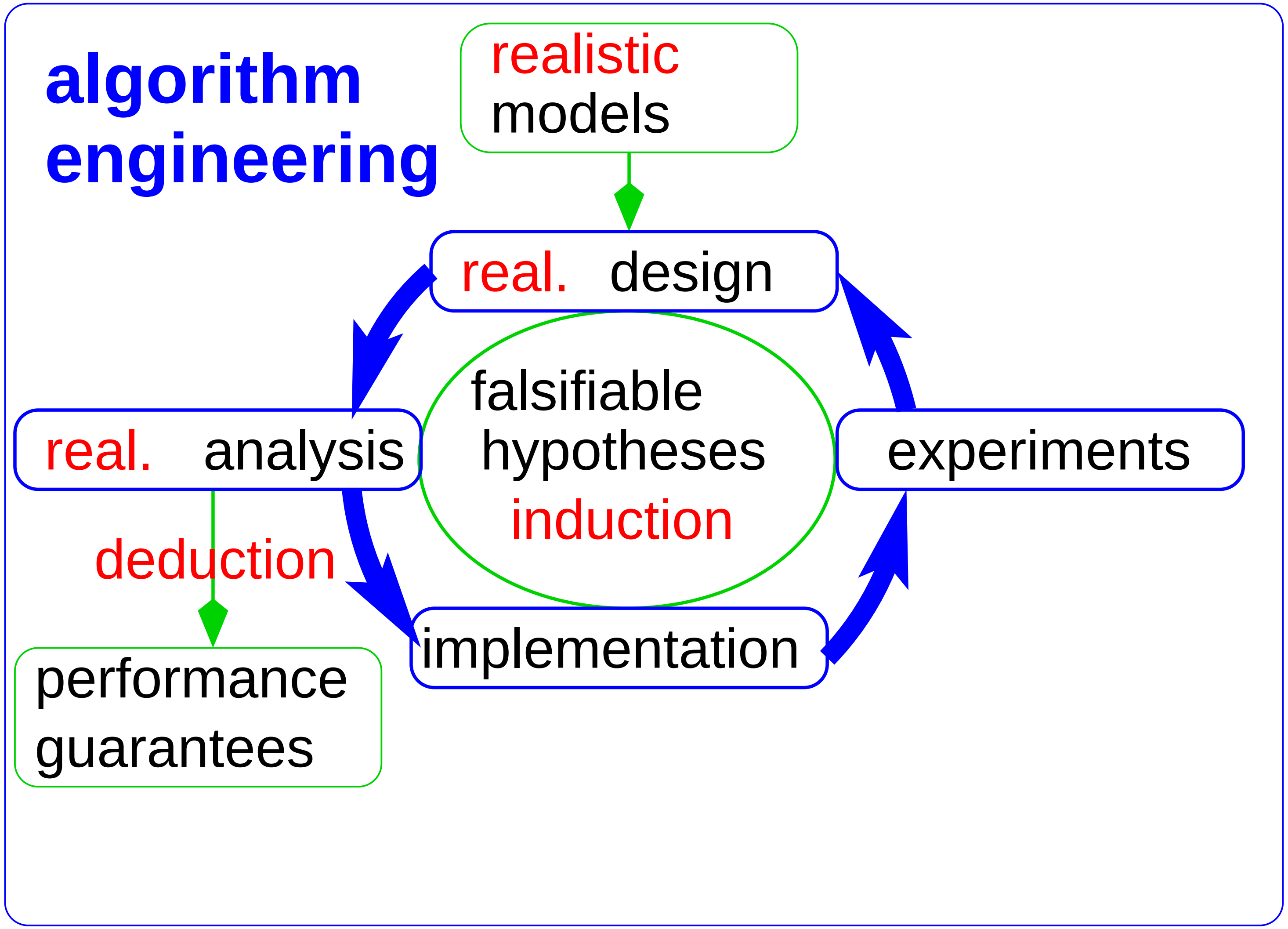
Algorithmics as Algorithm Engineering



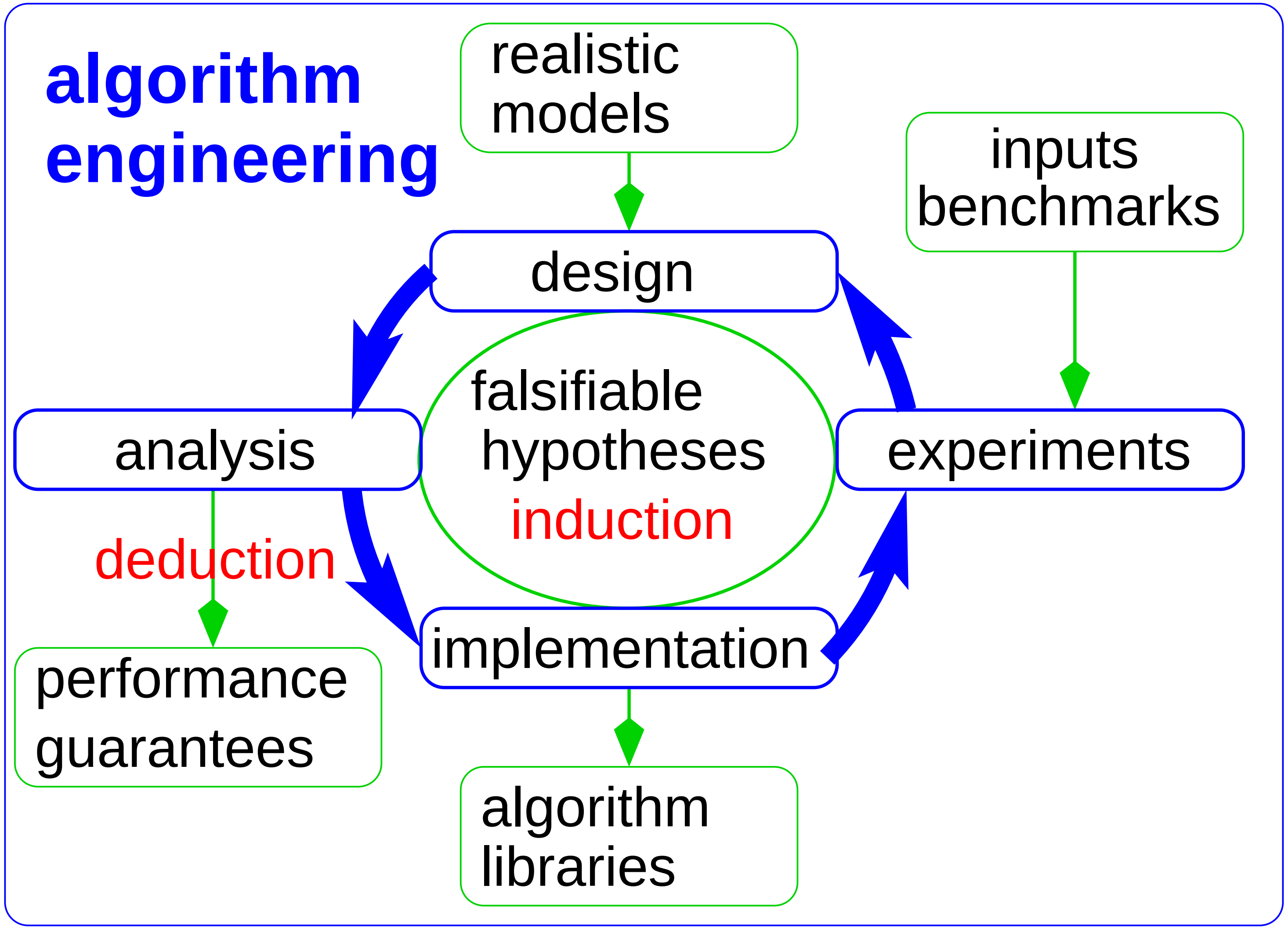
Algorithmics as Algorithm Engineering



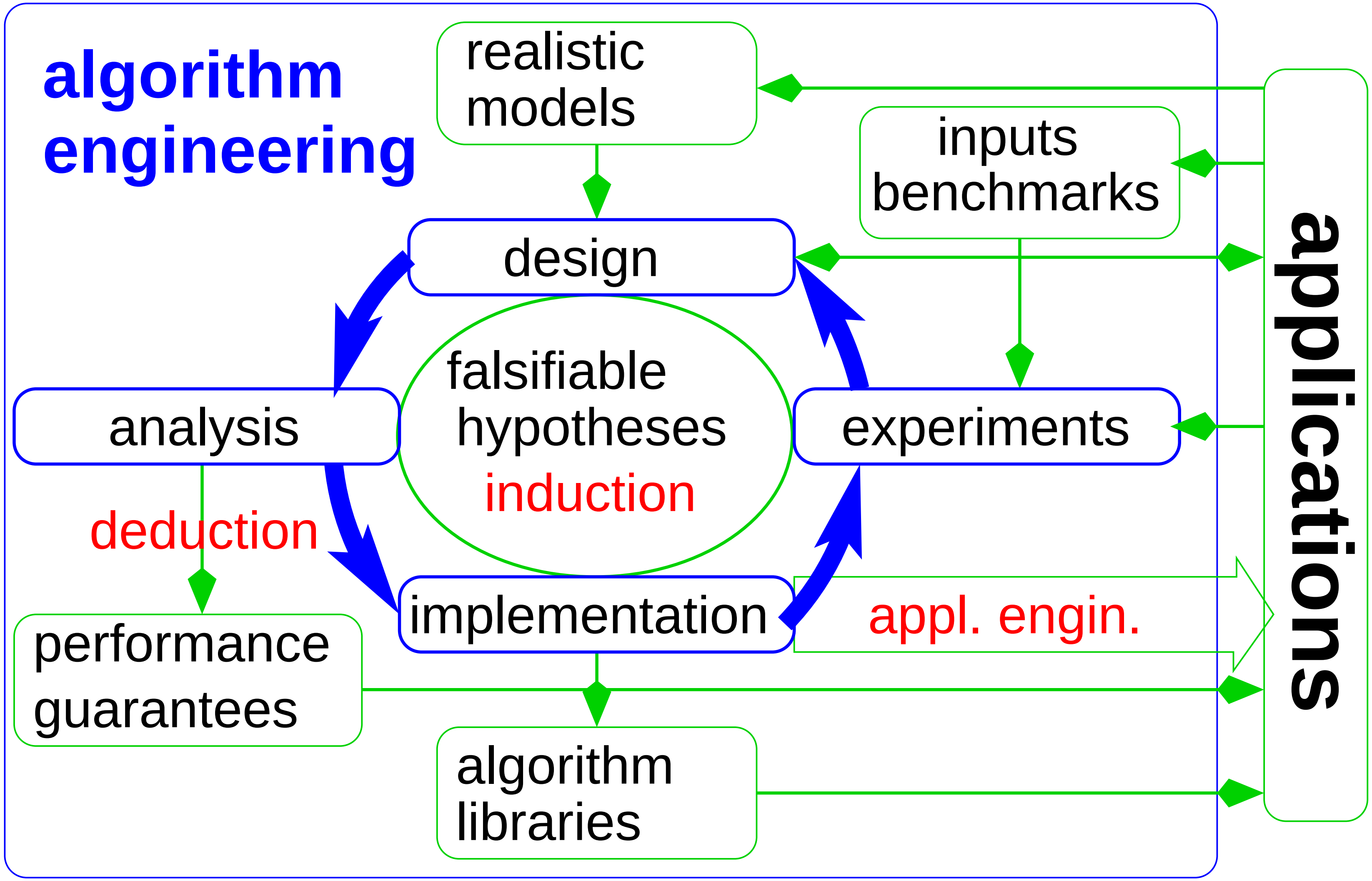
Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering

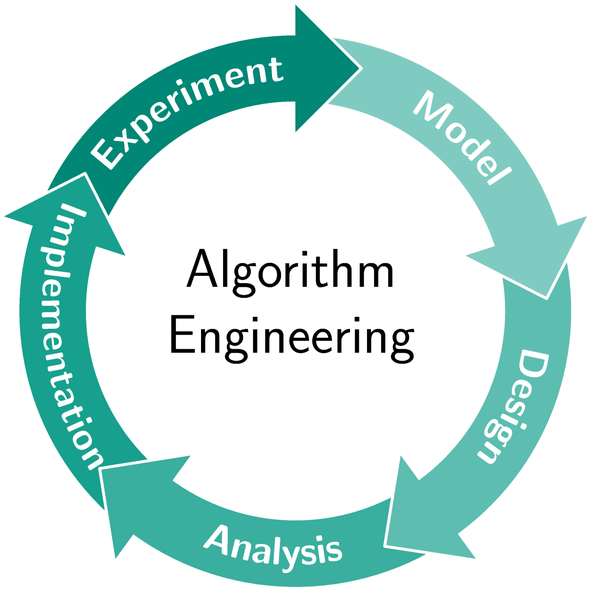


Algorithmics as Algorithm Engineering



Goals

- bridge gaps between theory and practice
- accelerate transfer of algorithmic results into applications
- keep the advantages of theoretical treatment:
  generality of solutions and
  reliabiltiy, predictabilty 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
$\rightsquigarrow$ ESA applied track [G. Italiano]

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

1998 Alex workshop in Italy $\rightsquigarrow$ ALENEX

Why this Lecture?

- Every computer scientist knows some textbook algorithms
  $\leadsto$ wir can start directly with algorithm engineering
- Many applications profit
- It is striking that there is so much new research possible
- Basis for bachelor and master theses

Was this Lecture is NOT:

Not Rehashed Algorithms I/II etc.

- Basic lectures often oversimplify
- Sometimes advanced algorithms
- Steeper learning curve
- Implementation details
- Emphasis on experiments

Was this Lecture is NOT:

Not a Theory Lecture

- few proofs
- actual performance before asymptotics

Was this Lecture is NOT:

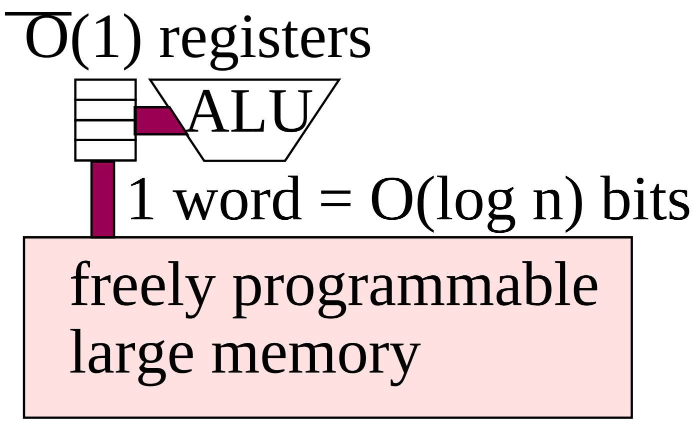
Not an Implementation Lecture

- Some algorithm analysis,...
- Little software engineering

Digression: Machine Models

RAM/von Neumann Model

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

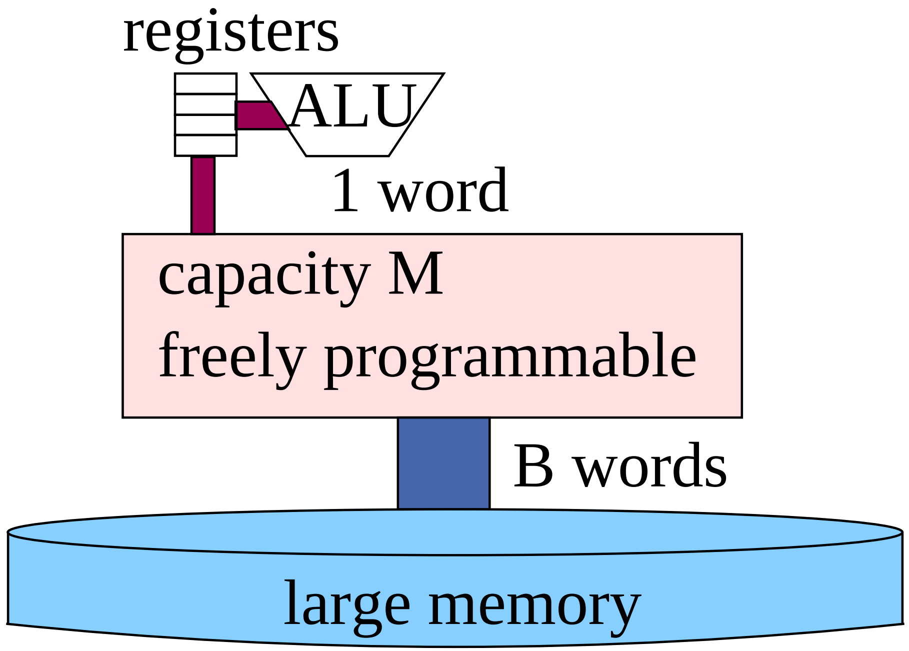


- simple
- very successful
- increasingly unrealistic
  because real hardware
  gets more and more complex

The External Memory Model

M: Fast memory of size *M*

B: Block size



Analysis: count (only?) block accesses (I/Os)

Interpretation of the External Memory Model

	external memory	Caches
large memory	disk(s)	main memory
M	main memory	one cache level
B	disk block (MBytes!)	cache block (16–256) bytes

possibly also two cache levels.

Variant: SSDs

More Model Aspekts

- Instruktion parallelism (Superscalar, VLIW, EPIC,SIMD,. . .)
- Pipelining
- Cost of branch misprediction?
- Multilevel caches (currenly 3 levels) $\leadsto$ “cache oblivious algorithms”
- Parallel processors, multithreading
- Communication networks
- . . .

1 Arrays, Linked Lists and derived data structures

Bounded Arrays

builtin data structure

size must be known in advance

Unbounded Array

e.g., `std::vector`

pushBack: append element

popBack: remove last element

Idea: double when space runs out

half when space gets wasted

If we do that right, n pushBack/popBack operations need time $\mathcal{O}(n)$

Algorithmese: pushBack/popBack have constant amortized complexity.

What can go wrong?

Doubly Linked Lists

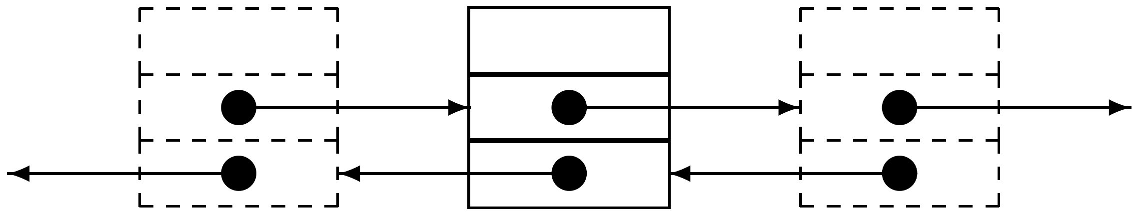
Class Item **of** Element // one link in a doubly linked list

e : Element

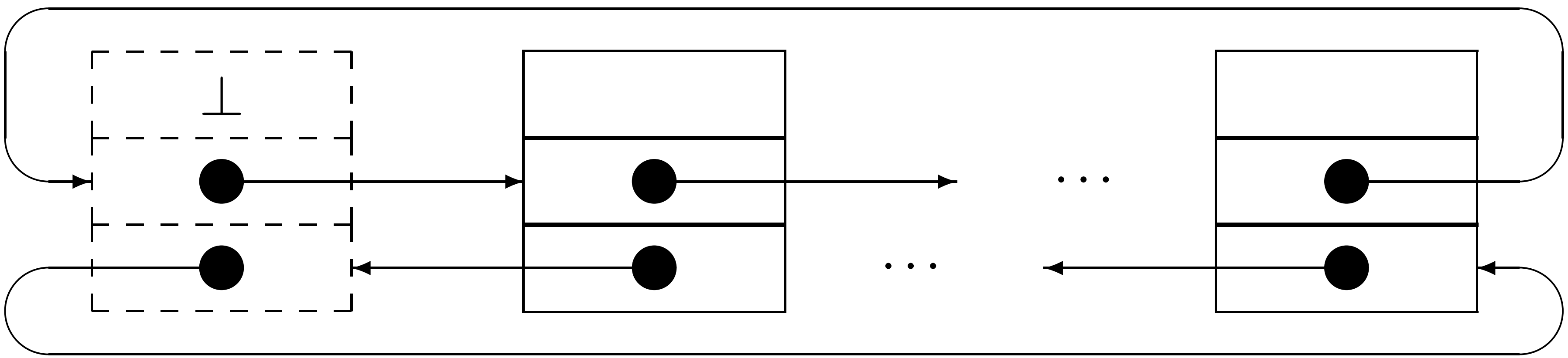
next : Handle //

prev : Handle

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



Trick: Use a dummy header



Procedure splice(a,b,t : Handle)

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

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

a' := a→prev

b' := b→next

a' →next := b' //

b' →prev := a' //

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

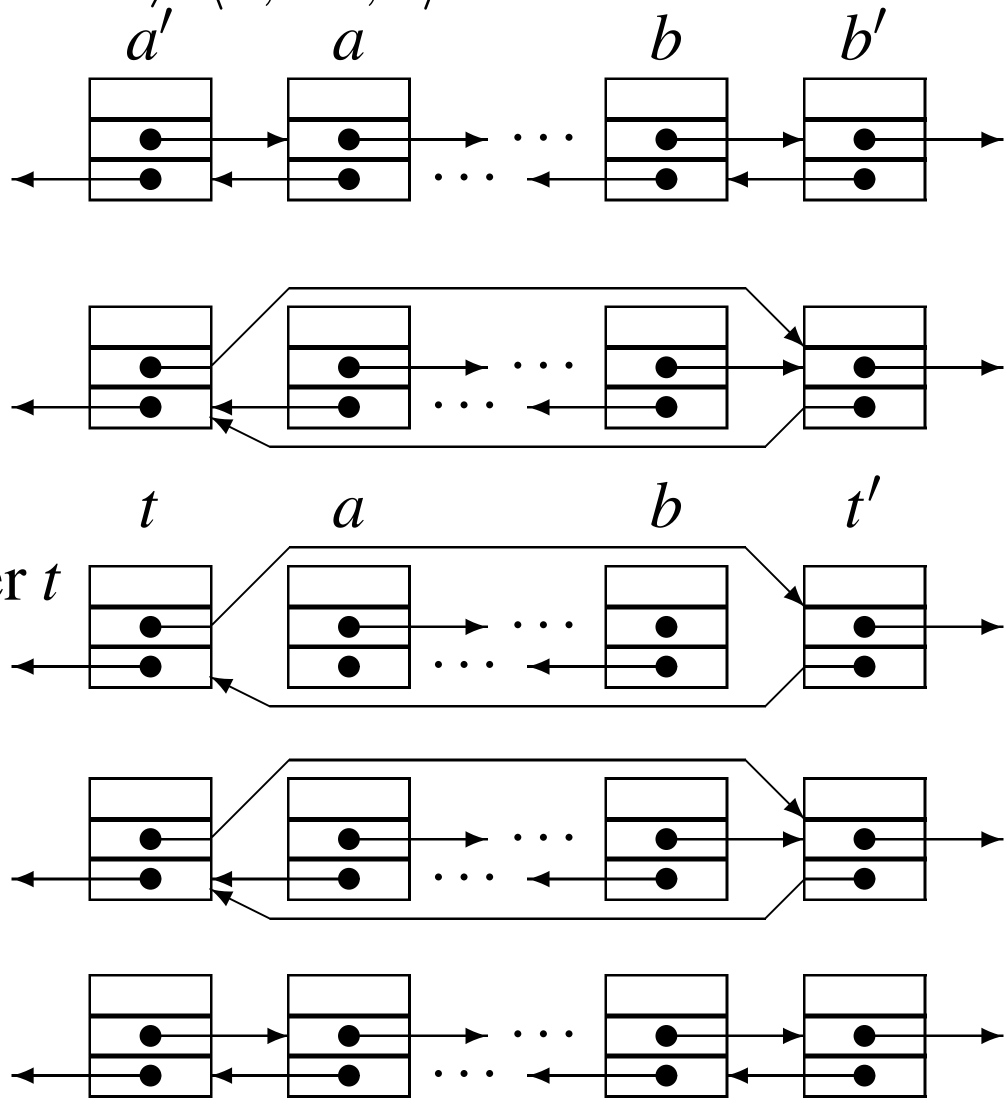
t' := t→next //

b→next := t' //

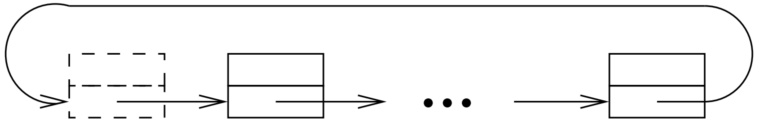
a→prev := t //

t →next := a //

t' →prev := b //



Singly Linked Lists



Comparison with doubly linked lists:

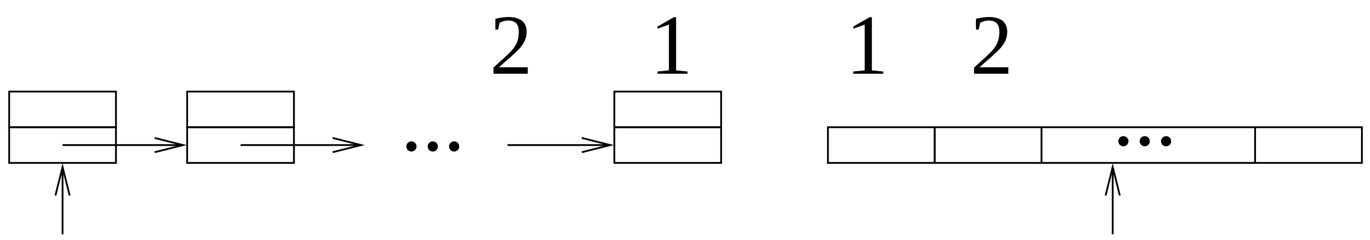
- Less space
- Space often implies time
- More restrictected, e.g., no delete
- Weird API, e.g., deleteAfter

Memory Management for Lists

- can easily cost 90 % of running time!
- Rather move elements between (Free)lists rather than actual `mallocs`
- Allocate many items at once
- Free together at the end?
- "parasitär" storage. e.g., graphs:
  node array. Each node stores a ListItem
  $\rightsquigarrow$ note partition can be represented as sth like linked lists
  $\rightsquigarrow$ MST, shortest path

Challenge: garbage collection, many data types
$\rightsquigarrow$ also a software engineering problem — not here

Example: Stack

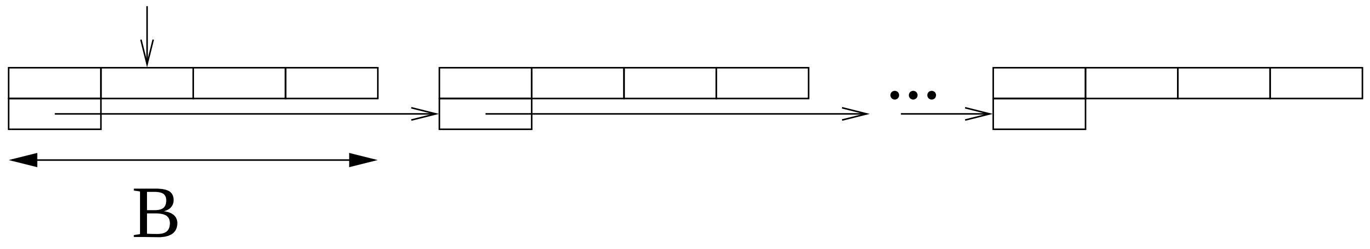


	SList	B-Array	U-Array
dynamic	+	−	+
space waste	pointer	too big?	too big?
	free?		
time waste	cache miss	+	copy
worst case time	(+)	+	−

Was that it?

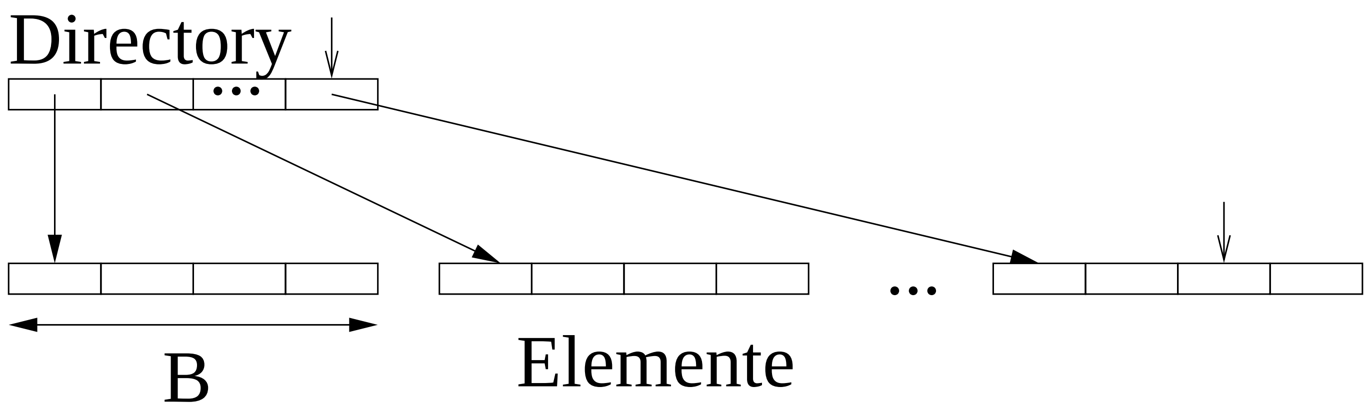
Every implementierung has serious weaknesses?

The Best From Both Worlds



	hybrid
dynamic	$+$
space waste	$n/B+B$
time waste	$+$
worst case time	$+$

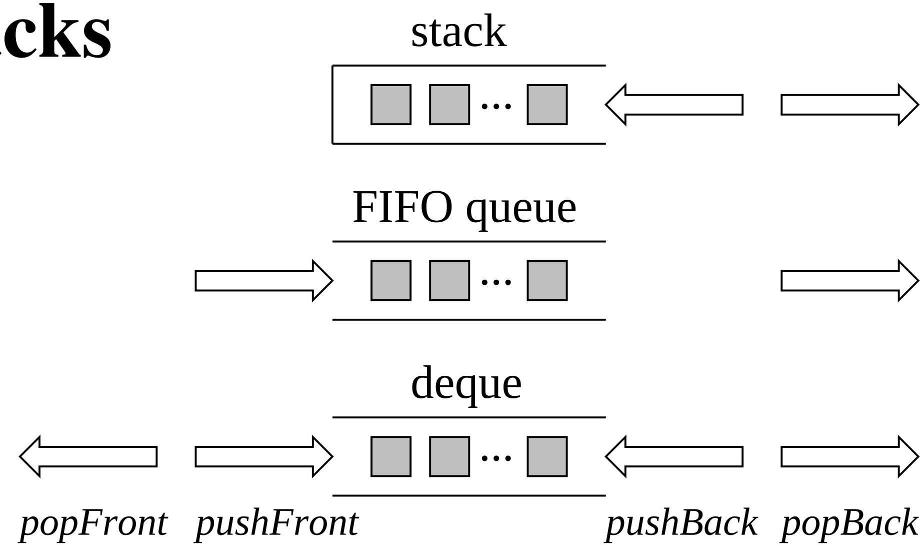
A Variant

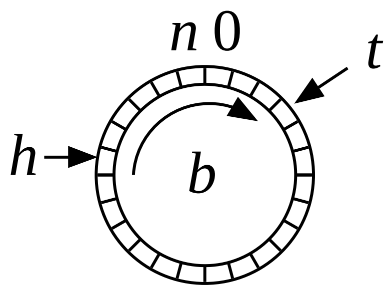


– Reallocations at the top level $\leadsto$ not worst case constant time

+ Indexed access to $S[i]$ in constant time

Beyond Stacks





FIFO: BArray $\longrightarrow$ cyclic array

Exercise: An array, that supports "[i]" in constant time and insert/delete in time $\mathcal{O}(\sqrt{n})$

Exercise: An external stack, that supports n push/pop operations with $\mathcal{O}(n/B)$ I/Os

Exercise: complete table for hybrid data structures

vervollständigen

Operation	List	SList	UArray	CArray	explanation of '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	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

What is Missing?

Facts Facts Facts

Measurements for

- Different implementation variants
- Different architectures
- Different input sizes
- Effects on actual applications
- Plots for all that
- Interpretation, possibly building a theory

Exercise: scan and array versus randomly allocated linked list

Algorithm Engineering
A Detailed View
Using Sorting as Guiding Example

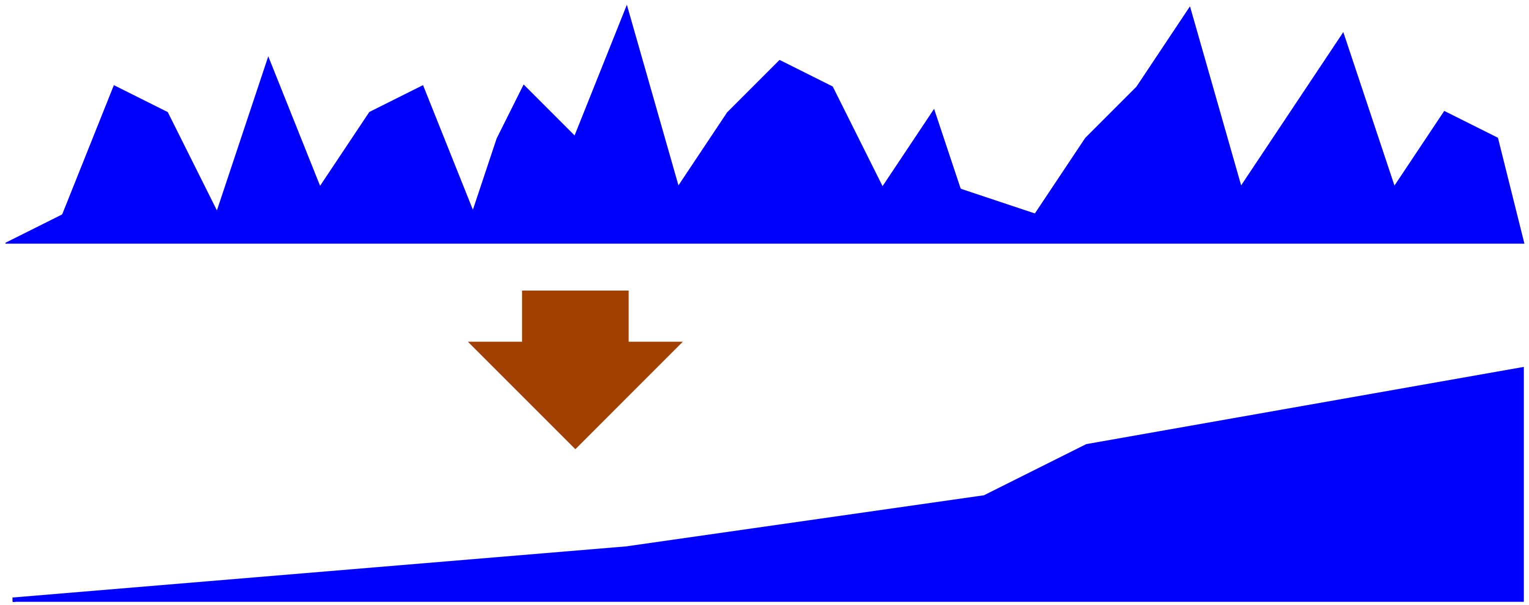
Sorting

Permute n elements of an array a such that

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

Efficient sequential, comparison based algorithms take time

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



Sorting – Model

Comparison based | arbitrary e.g. integer

true/false | full information

Why Sorting?

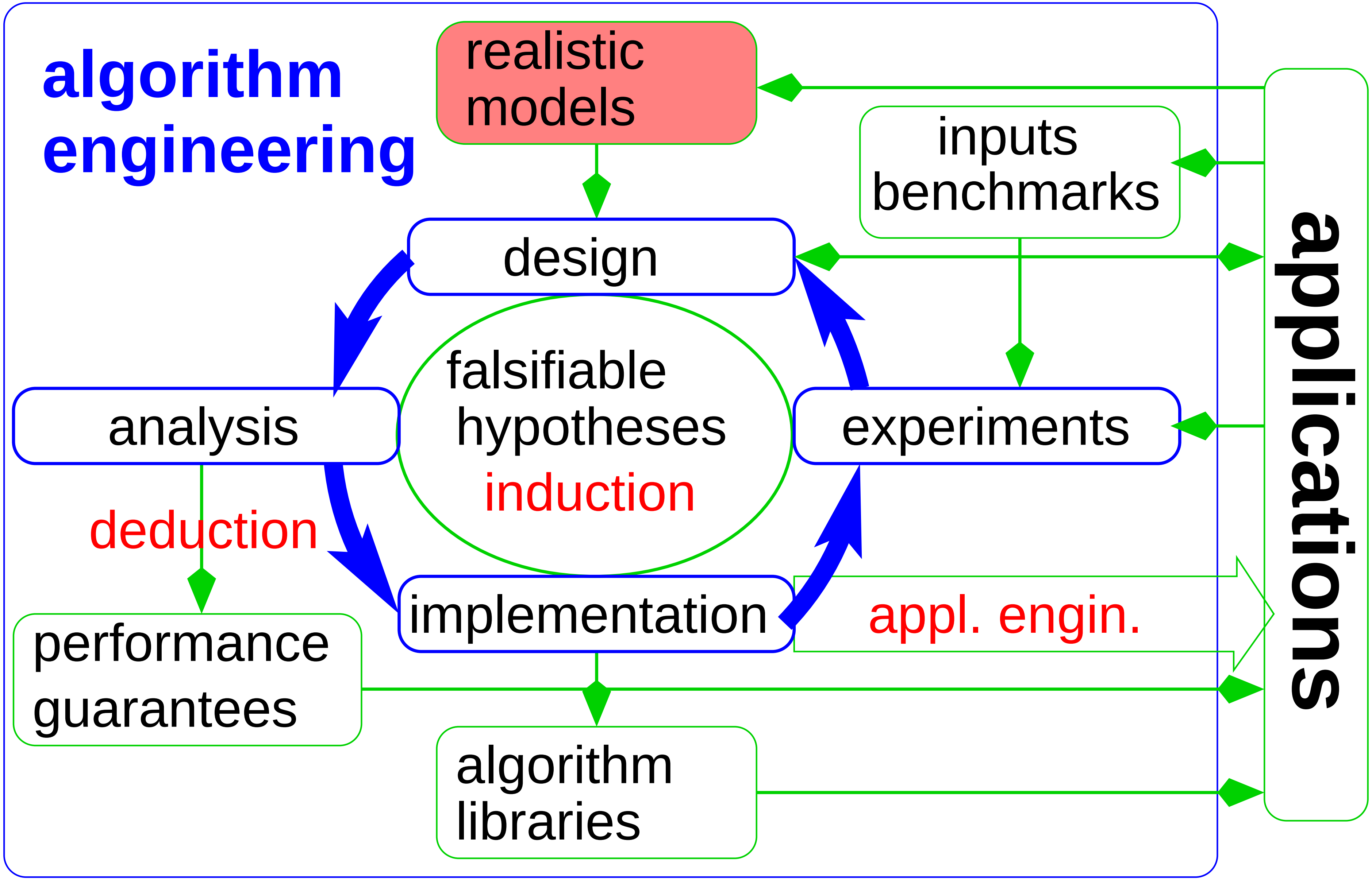
Teaching perspective:

- simple
- surprisingly nontrivial
- computer scientists know the basics

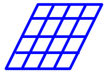
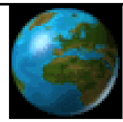
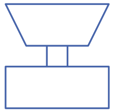
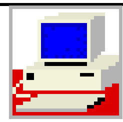
Application Perspective:

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

$\leadsto$ Bottleneck in many applications


algorithm
engineering
realistic
models
inputs
benchmarks
design
falsifiable
hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance
guarantees
algorithm
libraries
applications

Realistic Models

Theory	⟷	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**

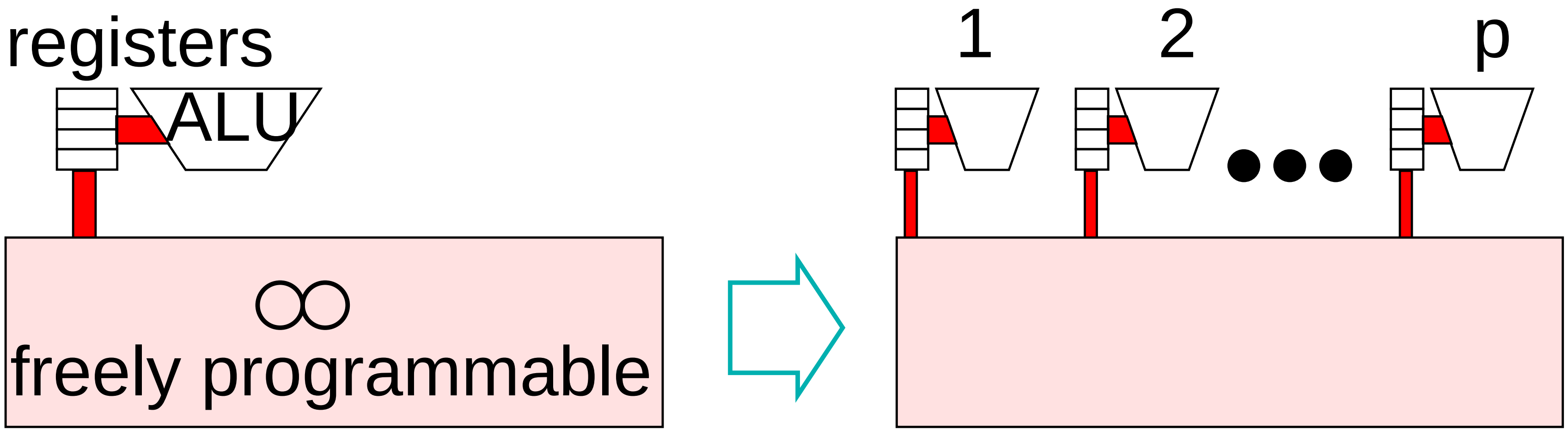
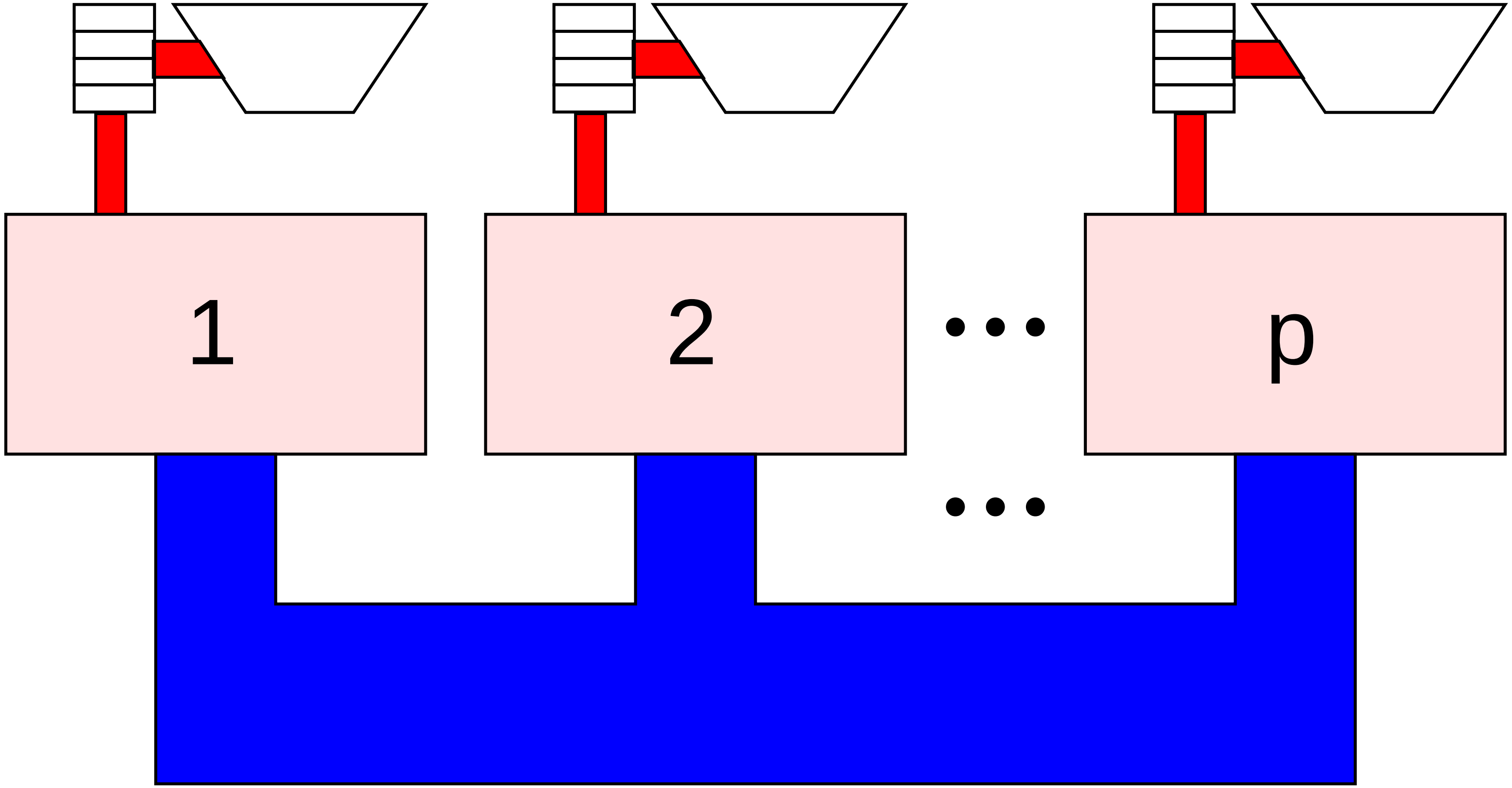


exhibit parallelism

Distributed Memory

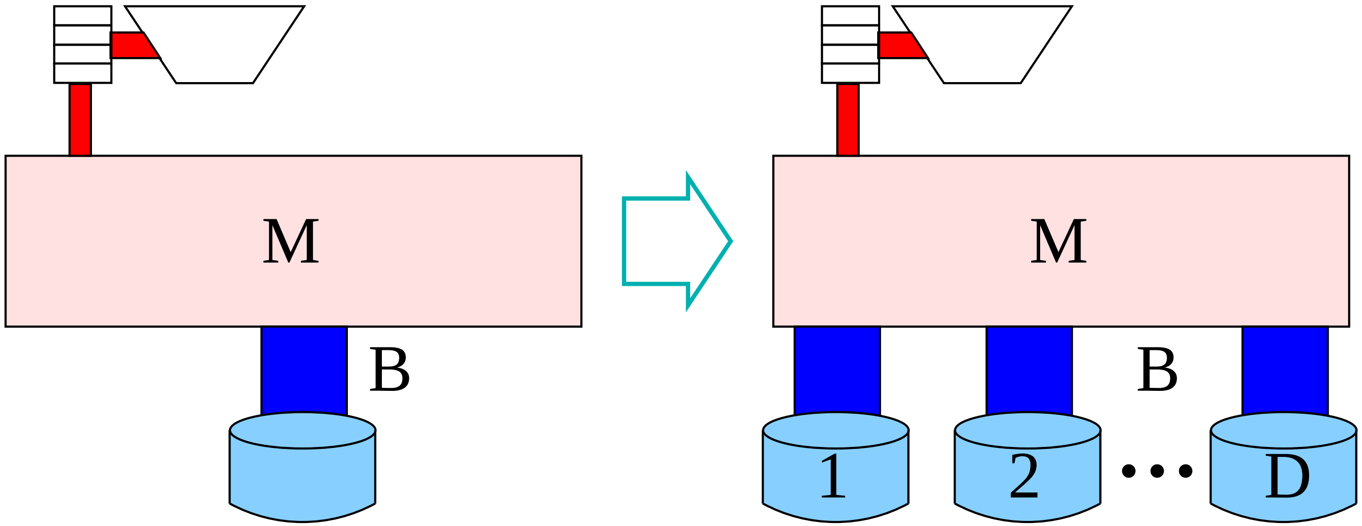
[1]



also consider communication

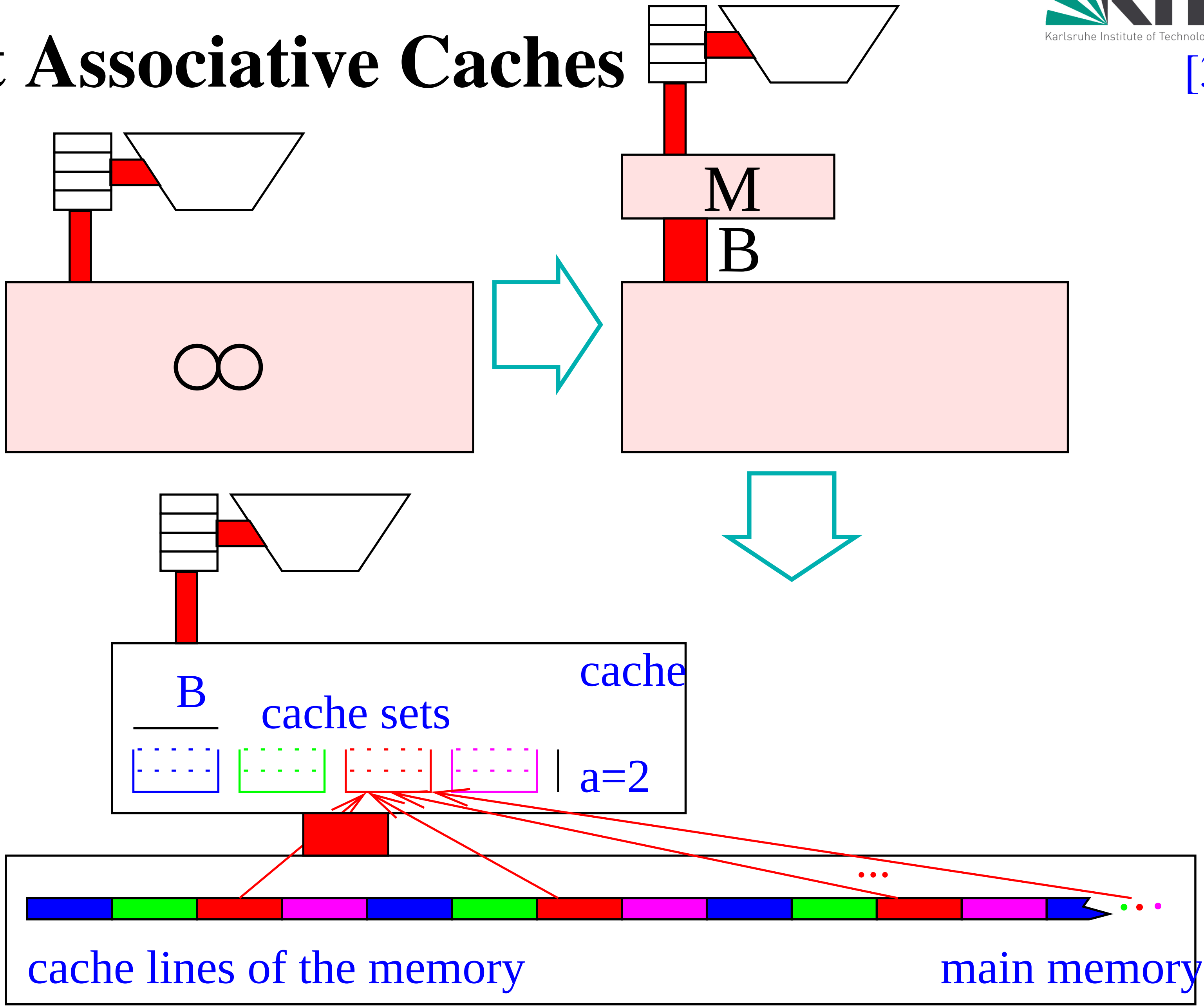
Parallel Disks

[2]



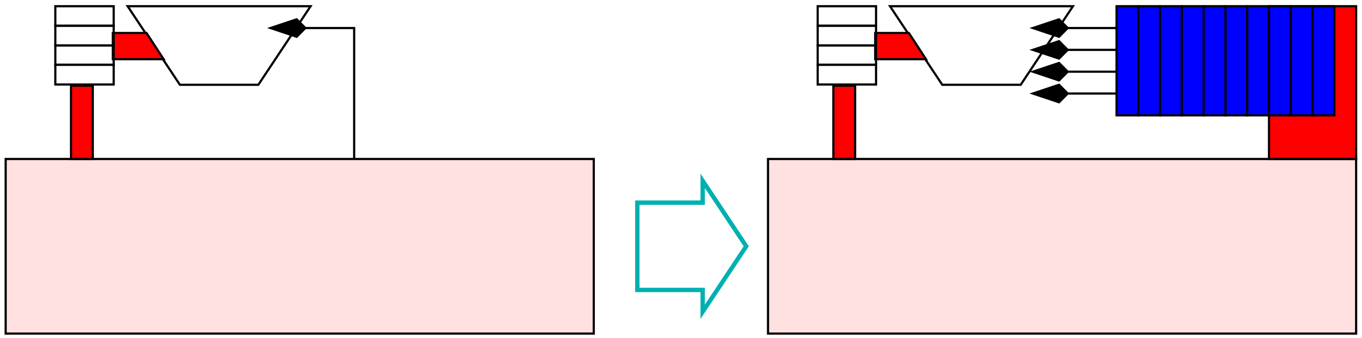
Set Associative Caches

[3]

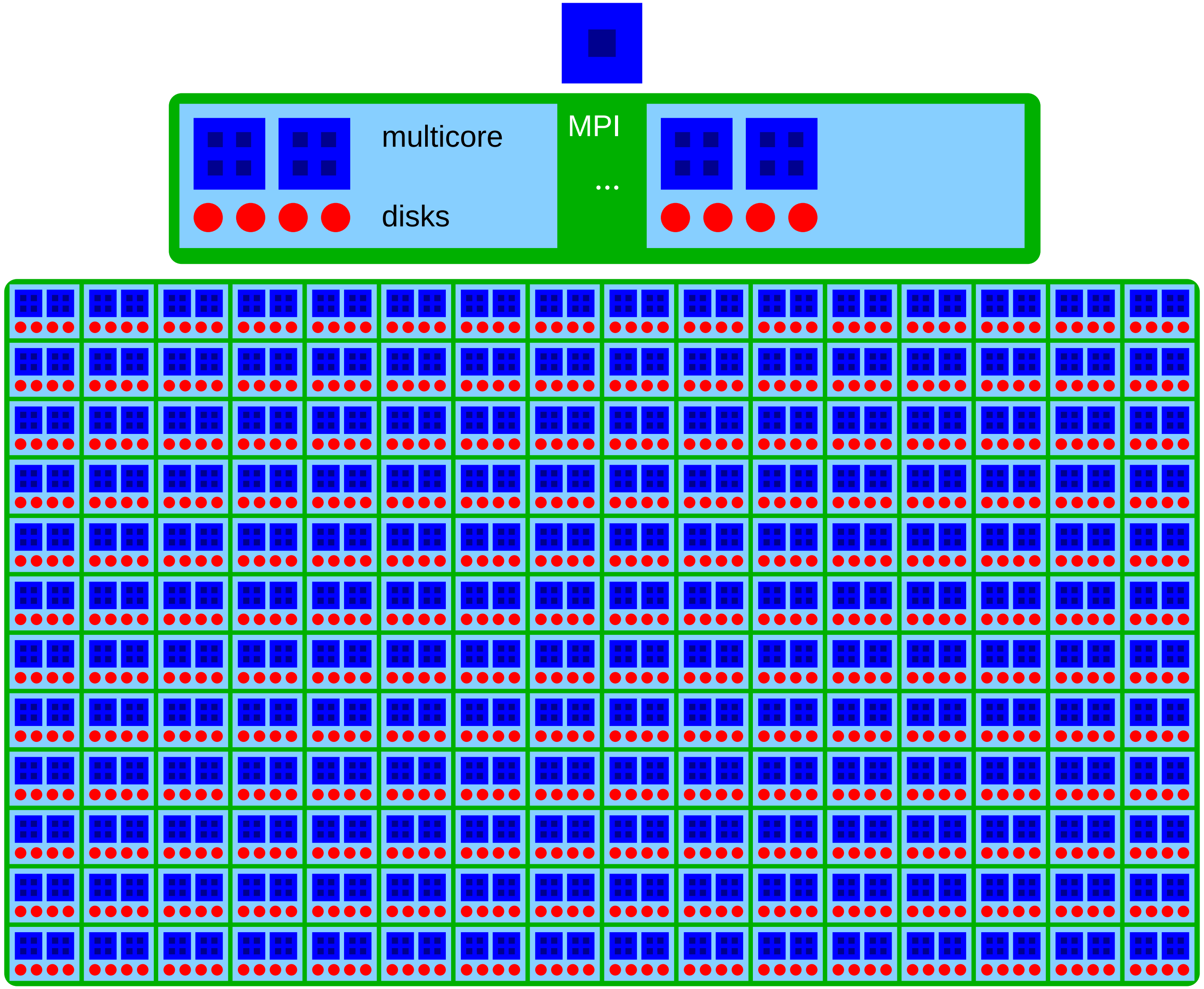


Branch Prediction

[4]

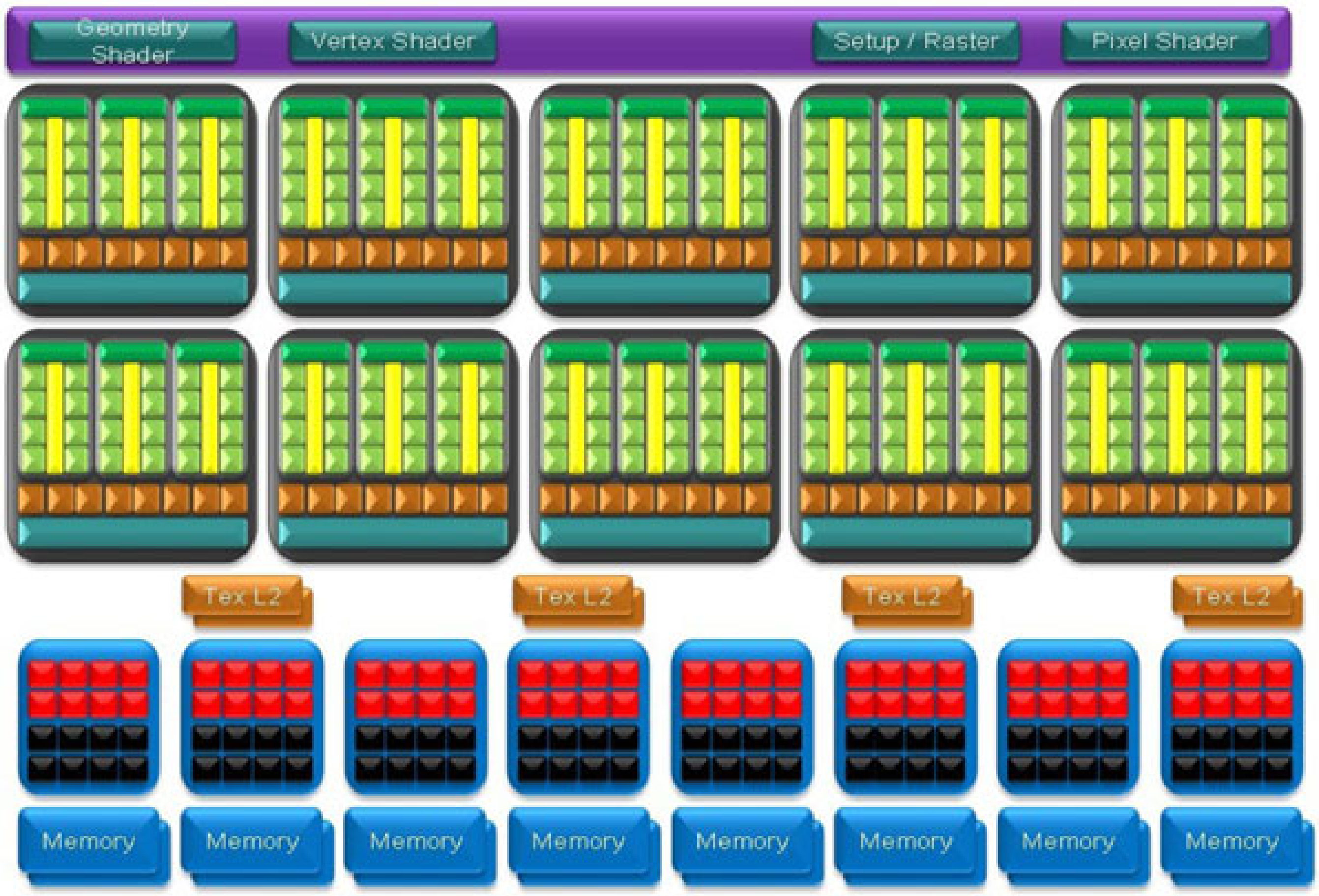


Hierarchical Parallel External Memory [5]



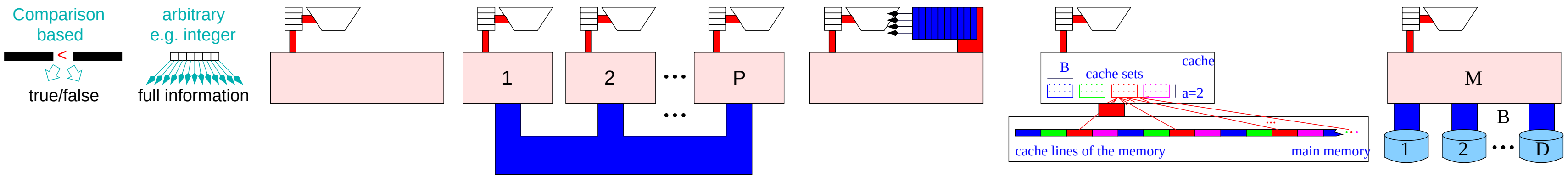
Graphics Processing Units

[6]

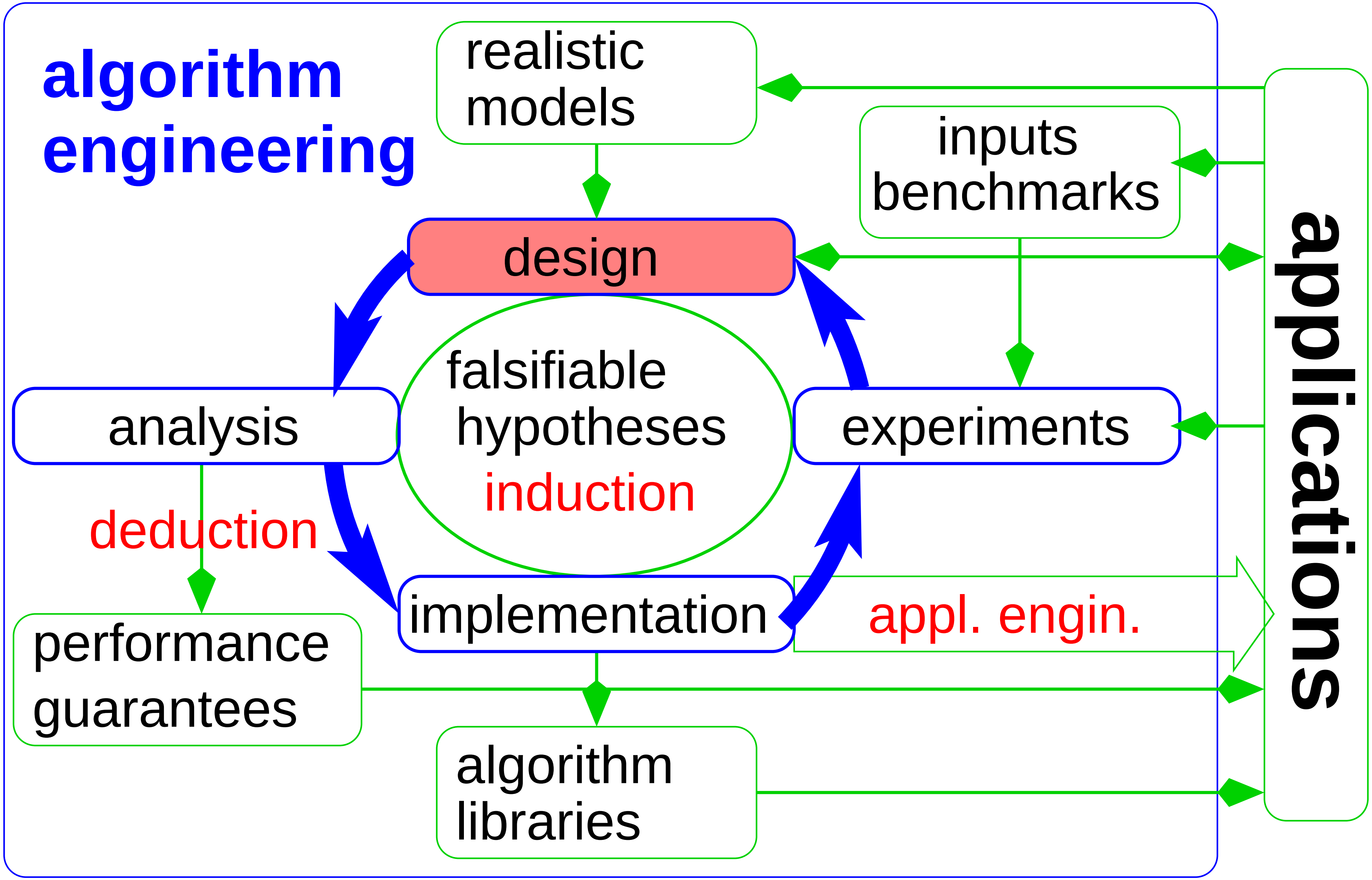


Combininig Models?

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



Or: Model Agnostic Algorithm Design


algorithm engineering
realistic models
inputs benchmarks
design
falsifiable hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance guarantees
algorithm libraries
applications

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]

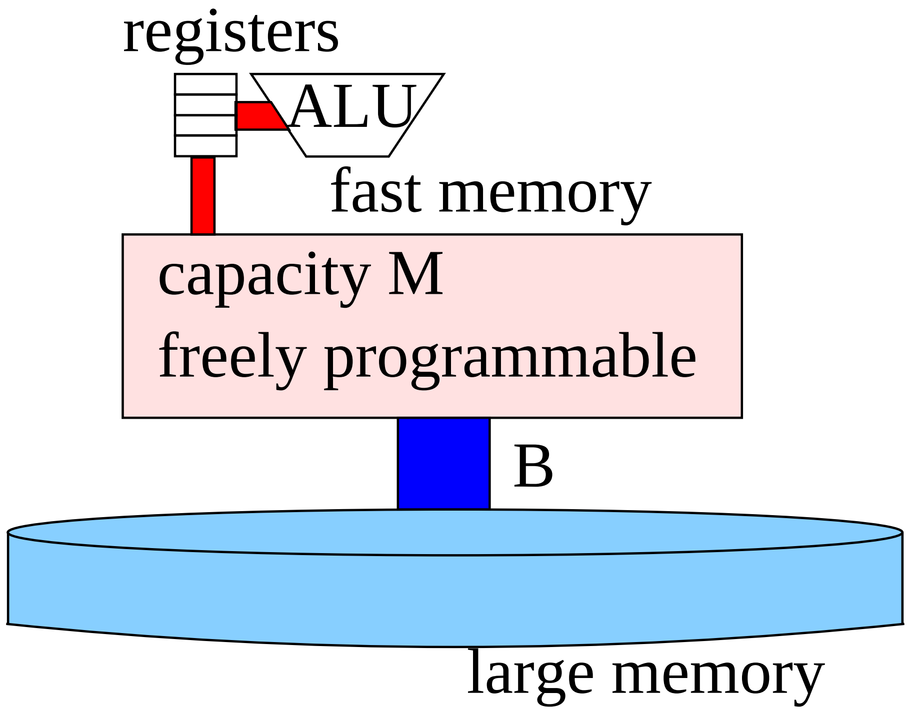
Example: External Sorting

[10]

n: input size

M: internal memory size

B: block size



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

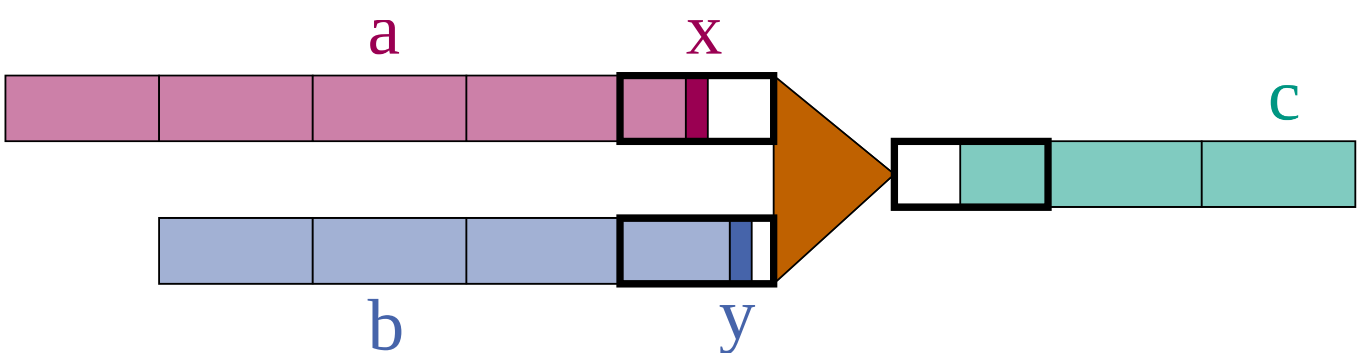
$x := a$.readElement // Assume emptyFile.readElement$= \infty$

$y := b$.readElement

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

if $x \leq y$ **then** c.writeElement(x); $x := a$.readElement

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



External Binary Merging

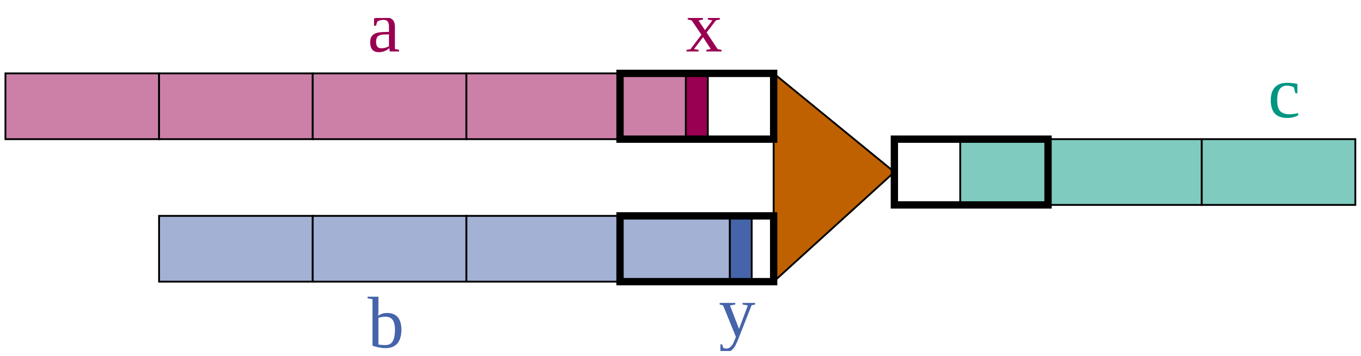
read file a: $\approx |a|/B$.

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

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

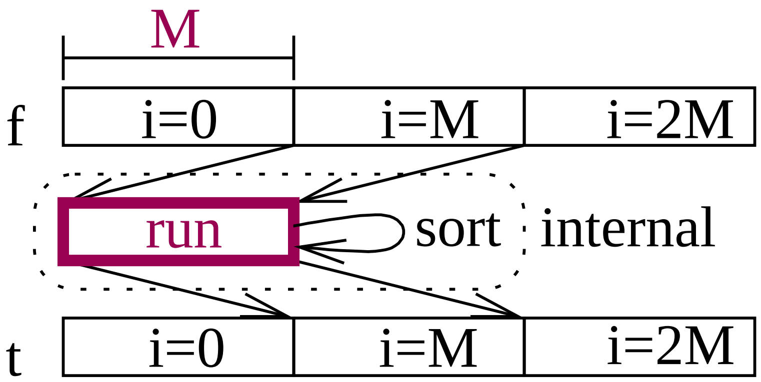
overall:

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



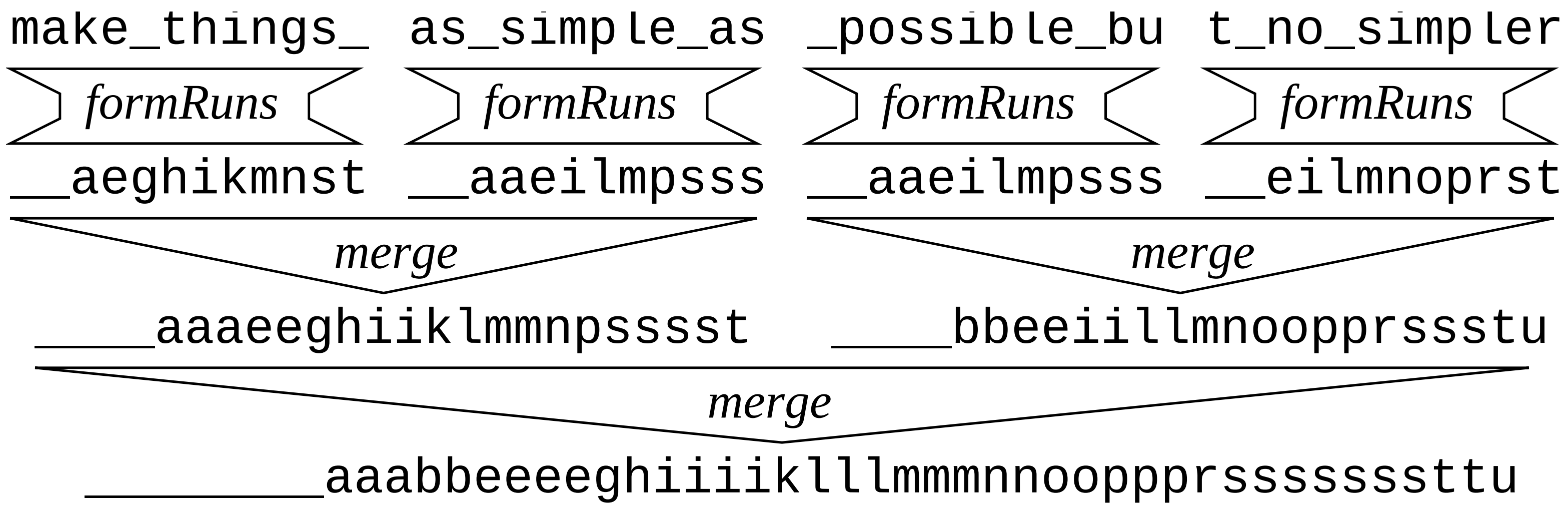
Run Formation

Sort input pieces of size M



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

Sorting by External Binary Merging



Procedure externalBinaryMergeSort // I/Os: $\approx$

  run formation // $2n/B$

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

    merge pairs of runs // 2n/B

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

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)

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

Idea: 8 passes $\rightsquigarrow$ 2 passes

Multiway Merging

Procedure multiwayMerge$(a_1, \ldots, a_k, c$:File **of** Element$)$

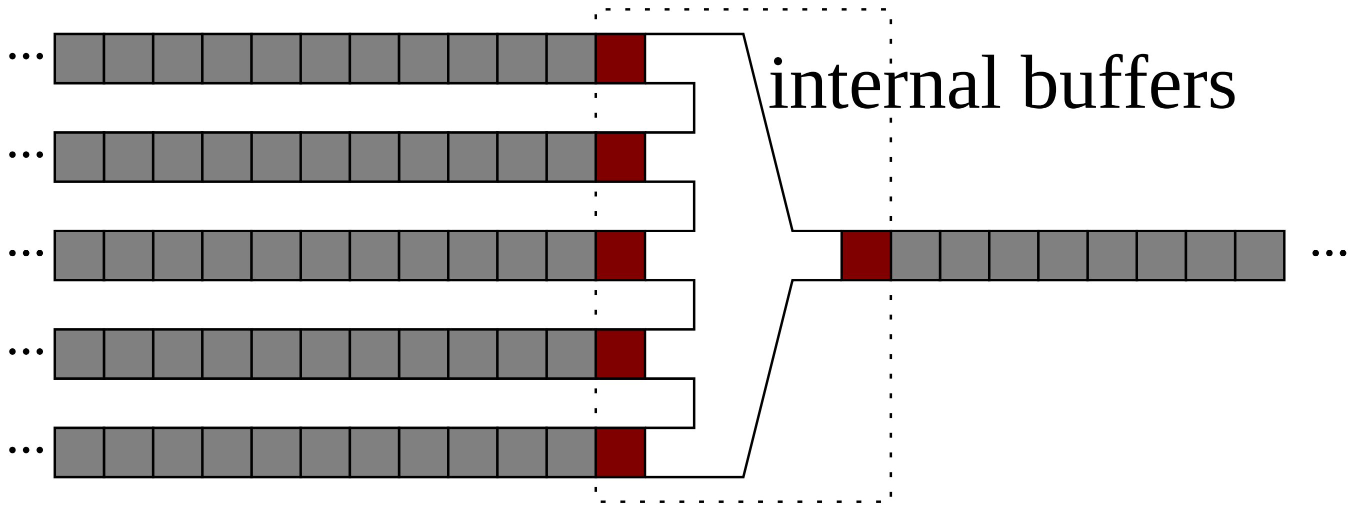
for $i := 1$ **to** k **do** $x_i := a_i$.readElement

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

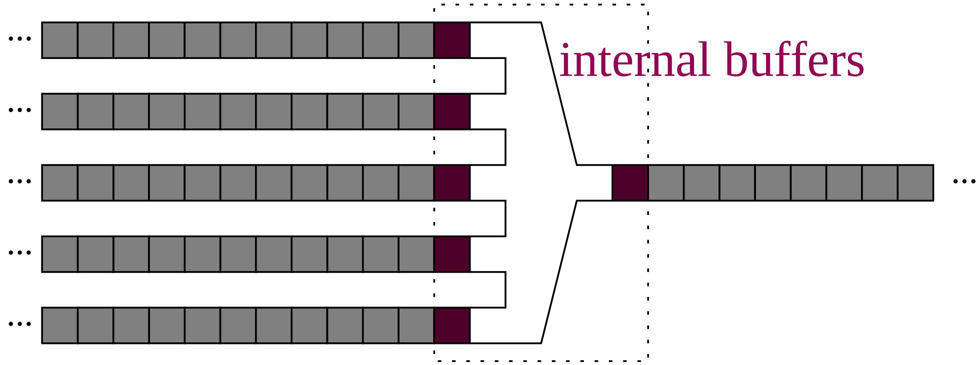
find $i \in 1..k$ that minimizes x_i// no I/Os!, $\mathcal{O}(\log k)$ time

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:

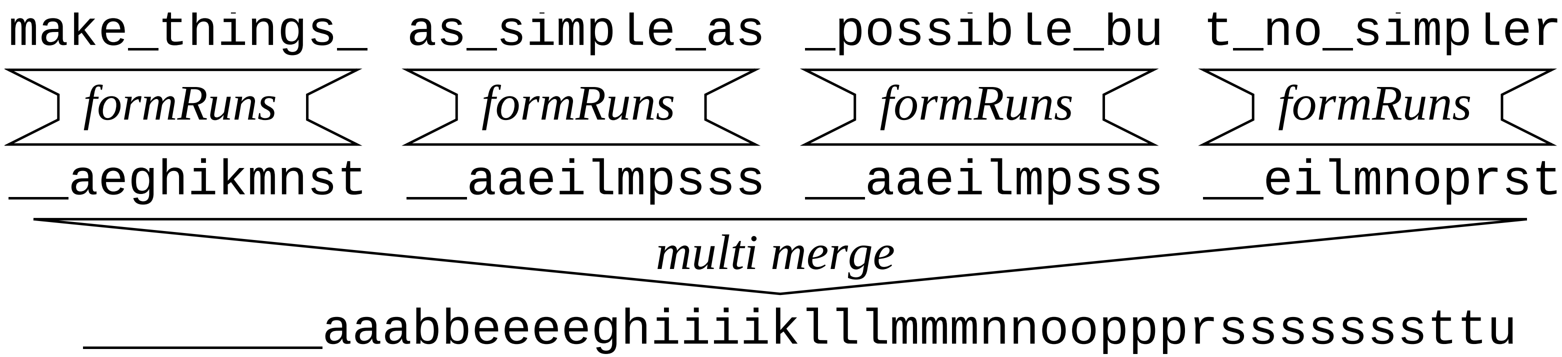
$$\leq\approx 2\frac{\sum_{i=1}^{k} |a_i|}{B}$$

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?:

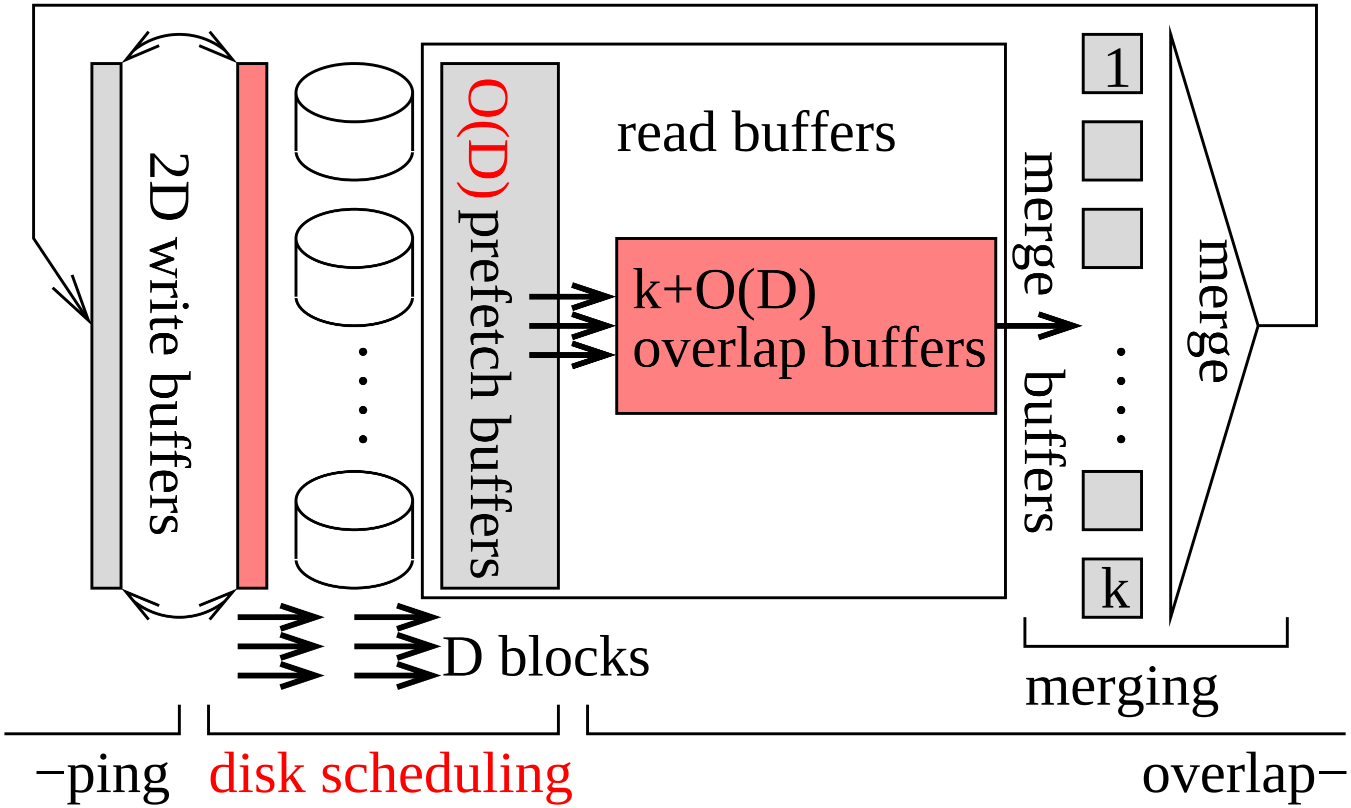
Not for the hierarchy main memory, hard disk.

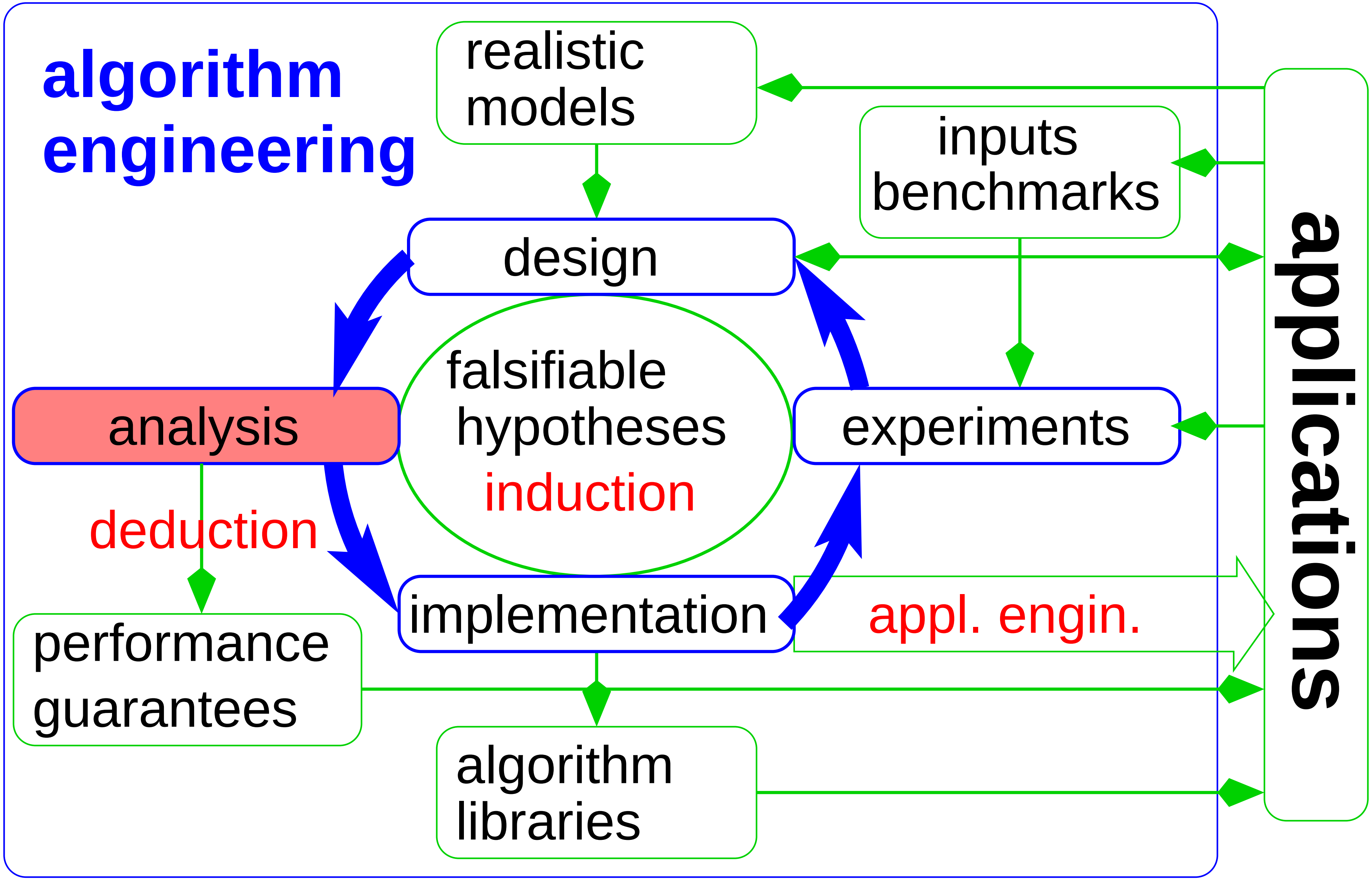
reason: $\overbrace{\frac{M}{B}}^{>4000} > \overbrace{\frac{\text{RAM Euro/bit}}{\text{Platte Euro/bit}}}^{\approx 207}$

Currently $4000 > 207$

More on Multiway Mergesort – Parallel Disks

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




algorithm
engineering
realistic
models
inputs
benchmarks
design
falsifiable
hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance
guarantees
algorithm
libraries
applications

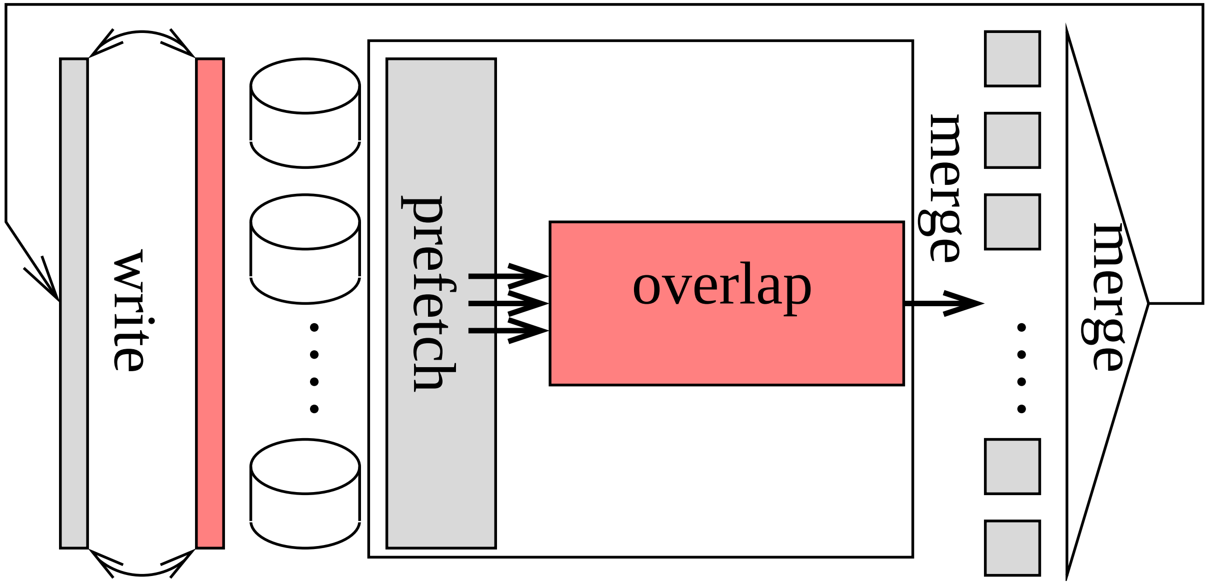
Analysis

- Constant factors matter

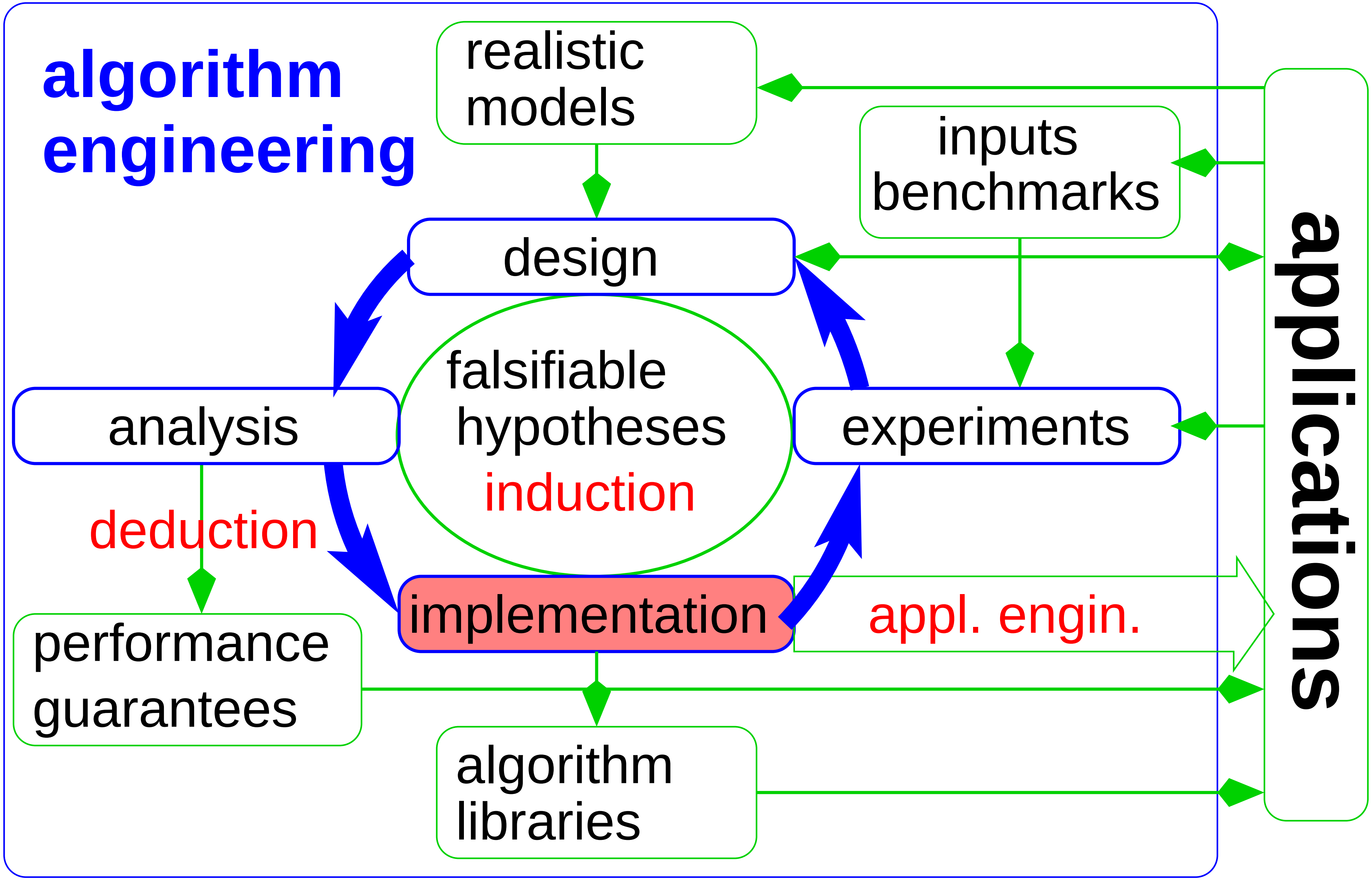
- Beyond worst case analysis

- 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]


algorithm
engineering
realistic
models
inputs
benchmarks
design
falsifiable
hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance
guarantees
algorithm
libraries
applications

Implementation

sanity check for algorithms !

Challenges

Semantic gaps:

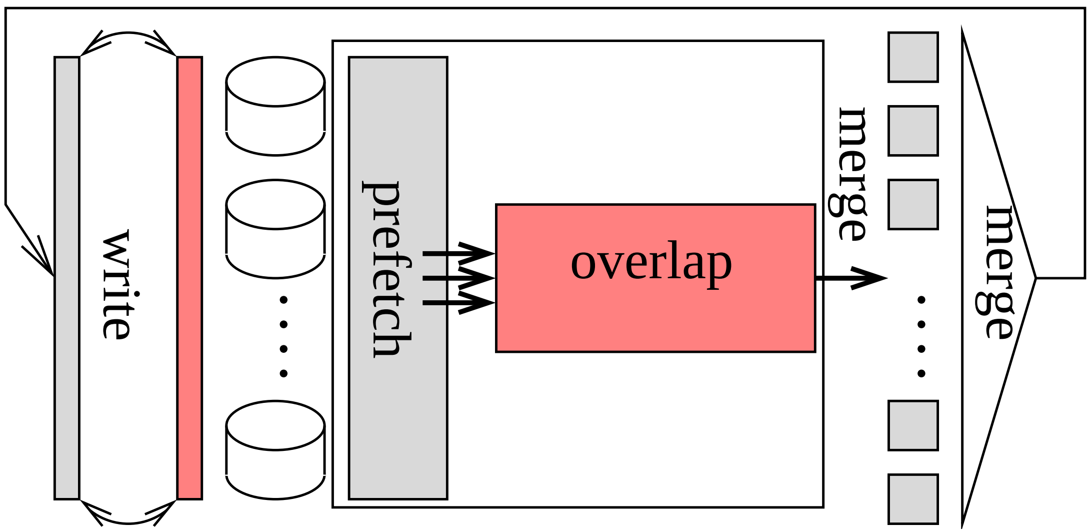
Abstract algorithm

$\leftrightarrow$

C++...

$\leftrightarrow$

hardware

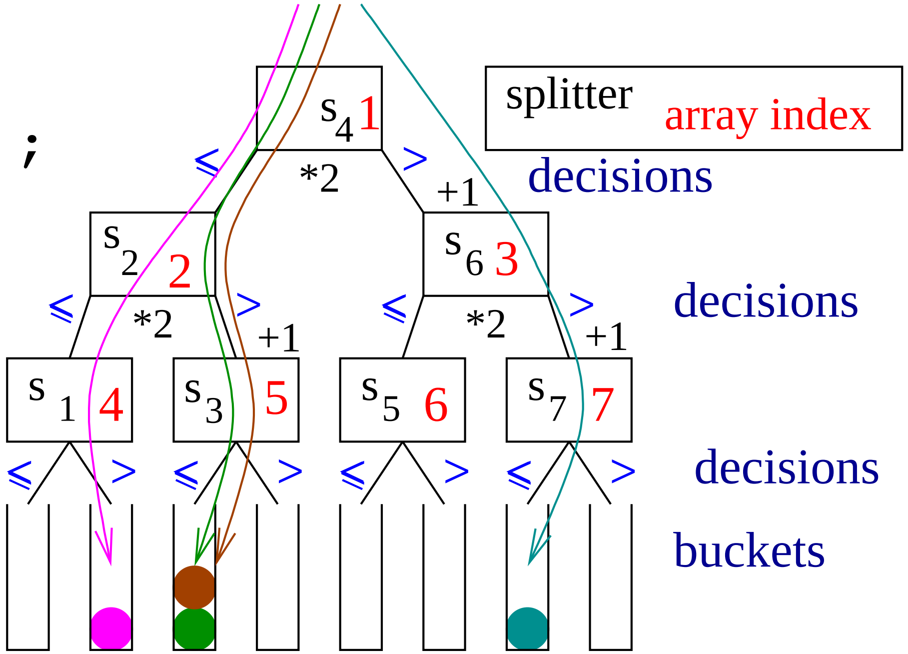


Small constant factors:

compare highly tuned competitors

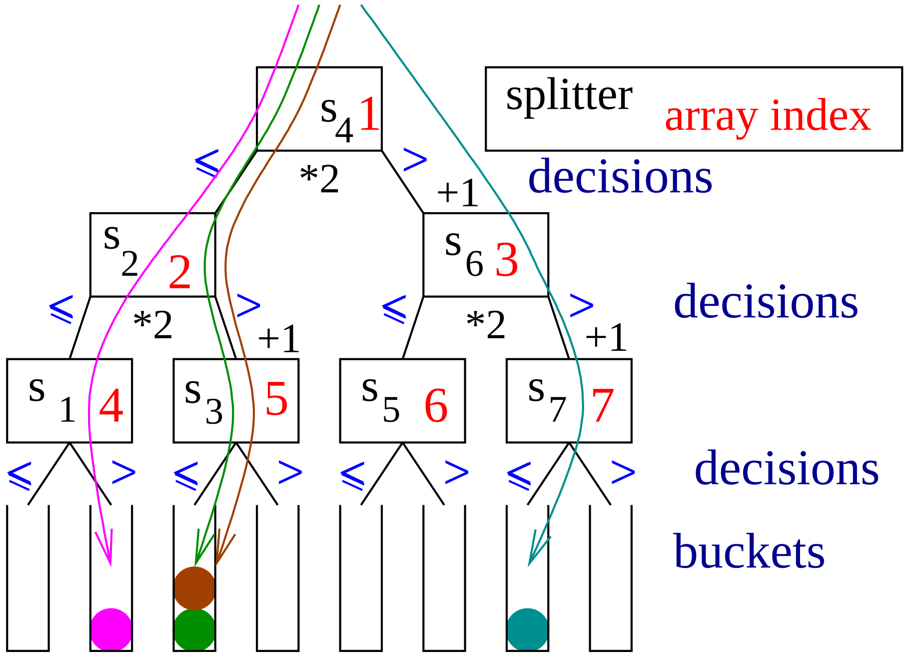
Example: Inner Loops Sample Sort [4]

```
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;
  }
}
```



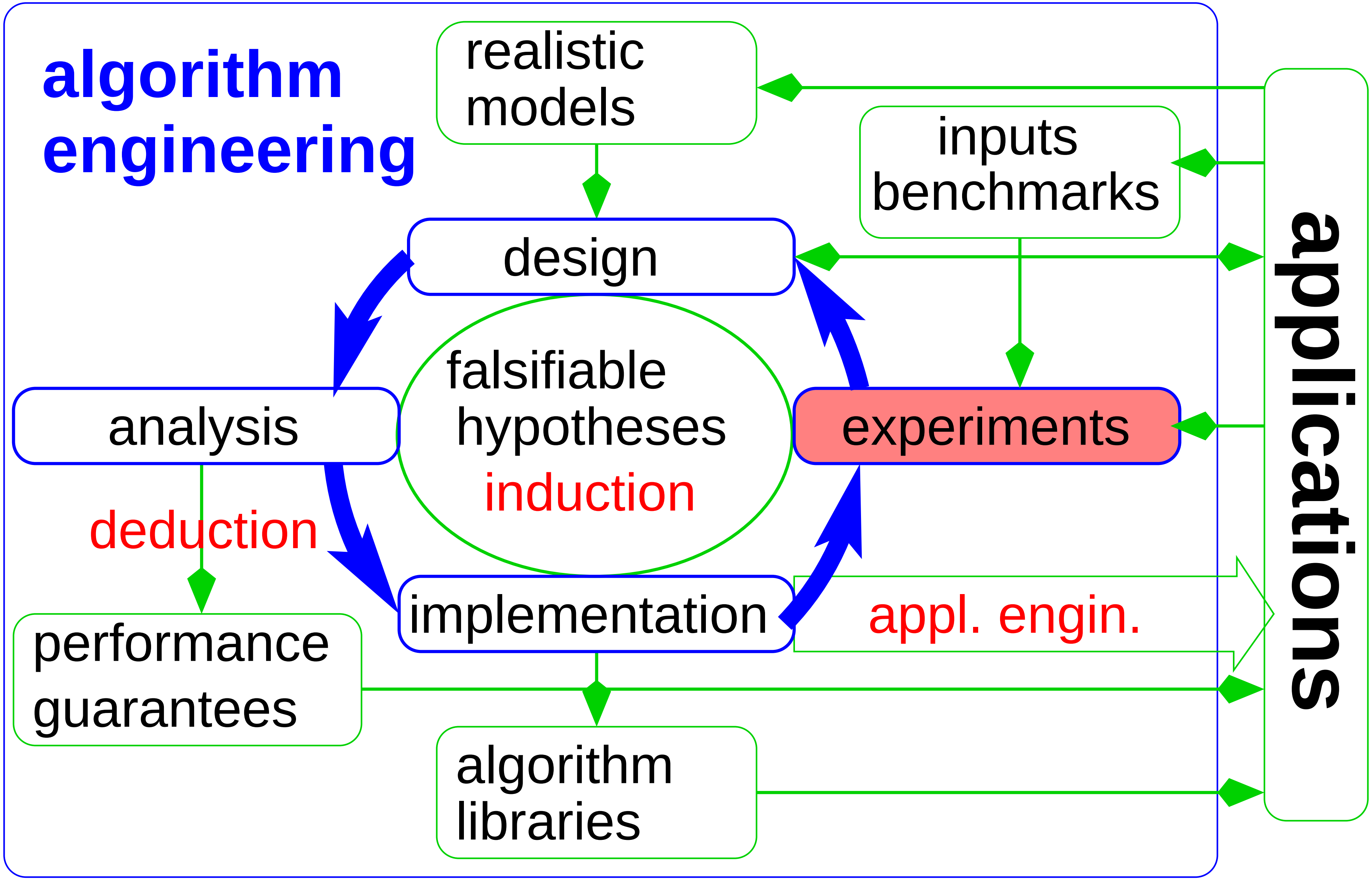
Example: Inner Loops Sample Sort [4]

```
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;
} }
```


algorithm engineering
realistic models
inputs benchmarks
design
falsifiable hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance guarantees
algorithm libraries
applications

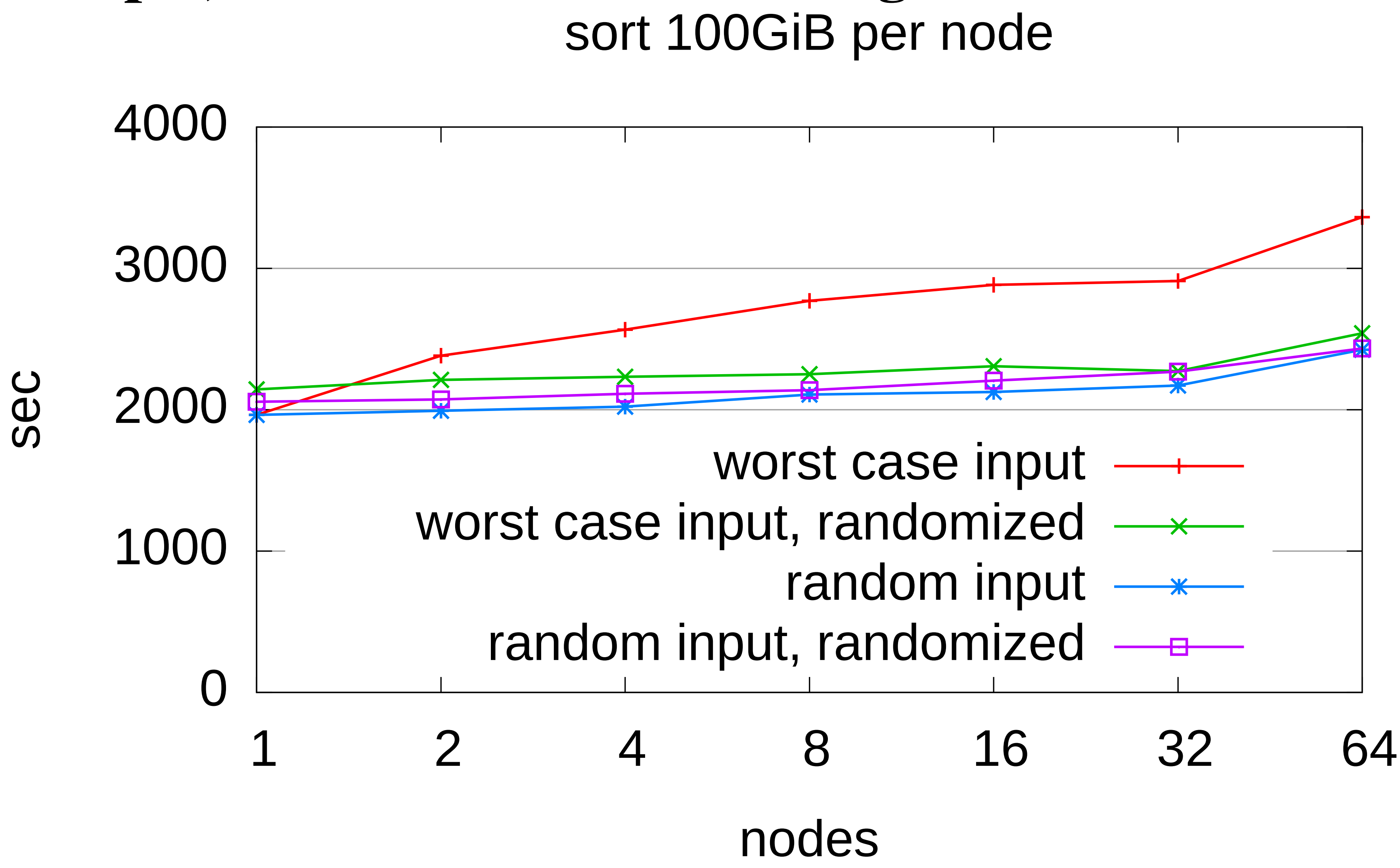
Experiments

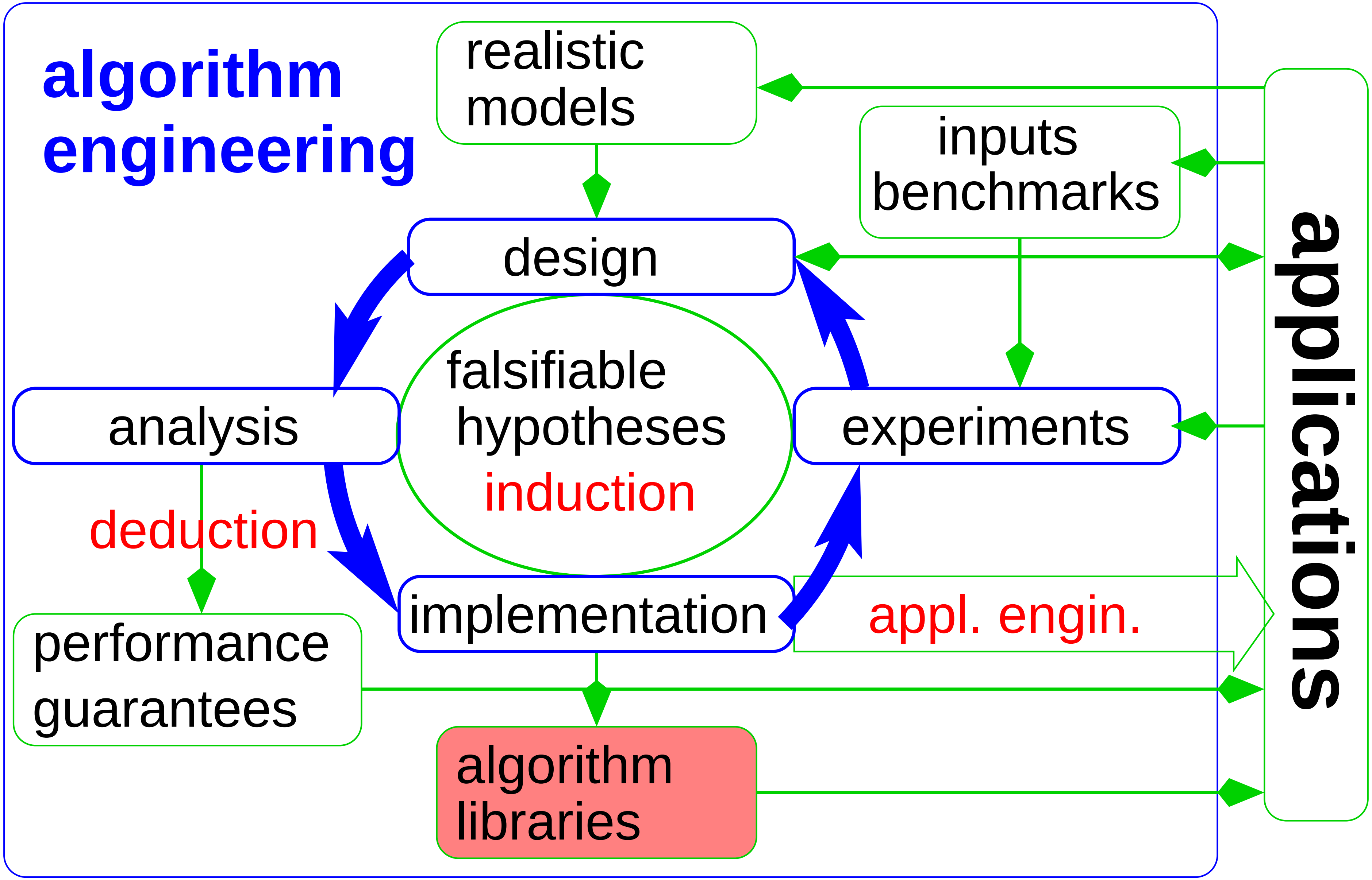
- central for AE in science
  reproducibility, careful comparisons, careful preparation of evidence
- also important in applications – just more informal
- careful planning
- careful interpretation
- close AE cycle fast

Experiments

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

Example, Parallel External Sorting [7]

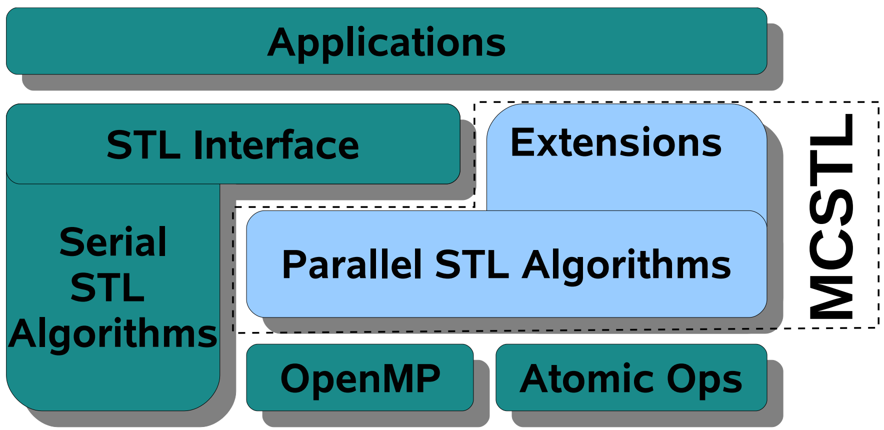


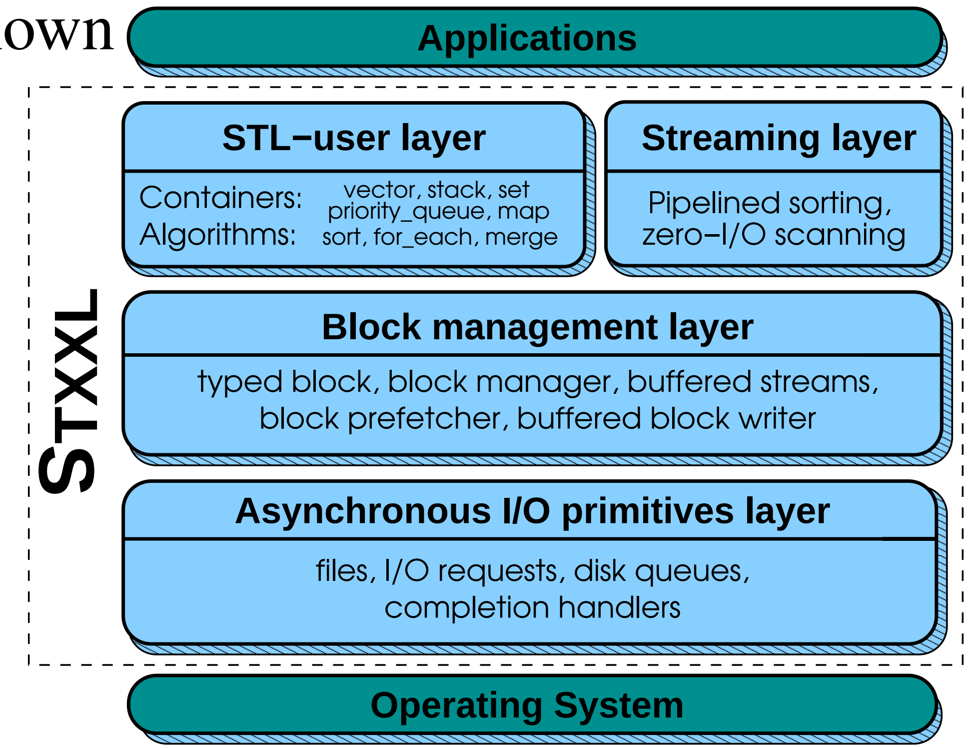


algorithm
engineering
realistic
models
inputs
benchmarks
design
falsifiable
hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance
guarantees
algorithm
libraries
applications

Algorithm Libraries — Challenges

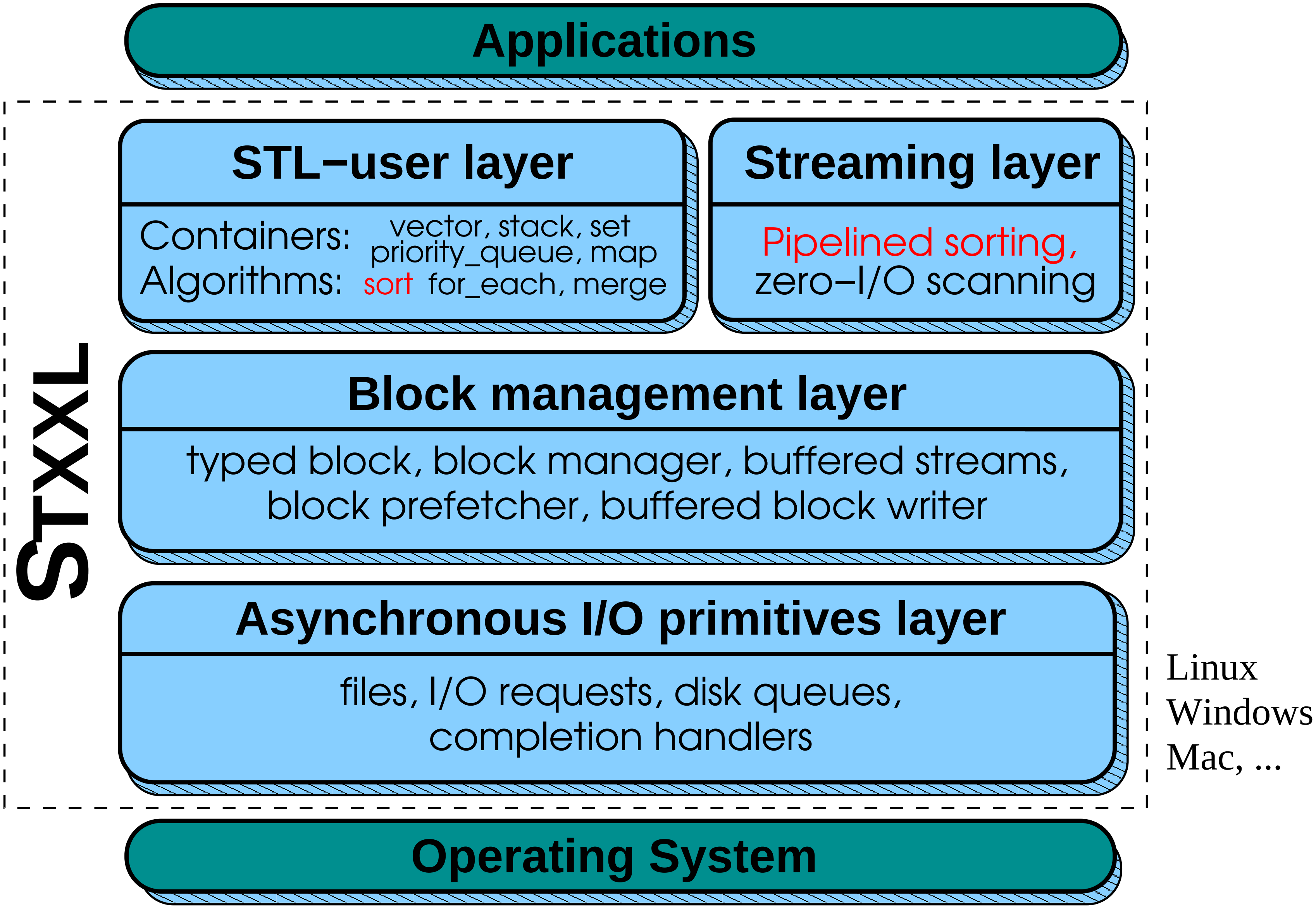
- software engineering , e.g. CGAL [www.cgal.org]
- standardization, e.g. java.util, C++ STL and BOOST
- performance $\leftrightarrow$ generality $\leftrightarrow$ simplicity
- applications are a priori unknown
- result checking, verification

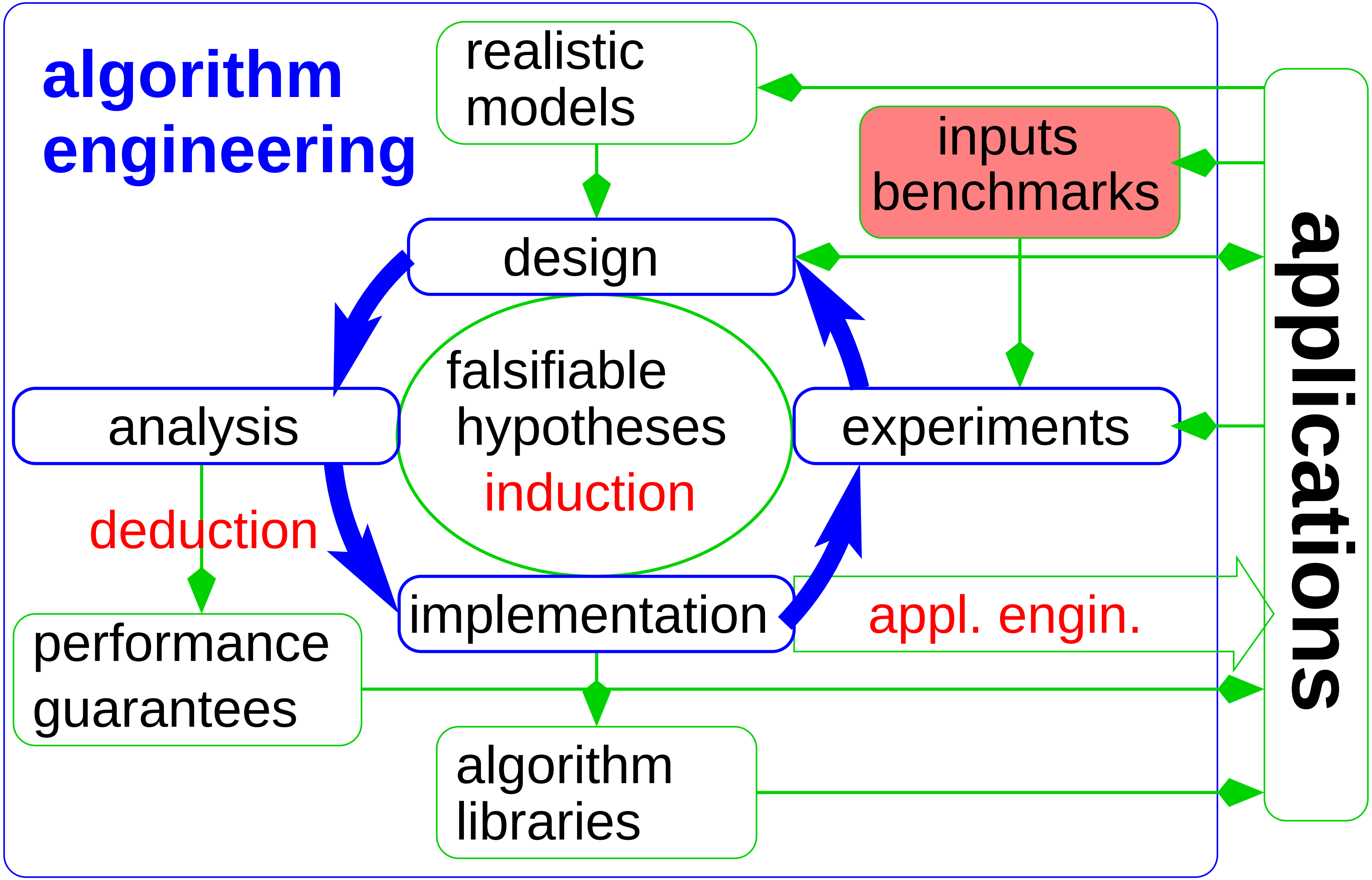




Example: External Sorting

[7, 11]

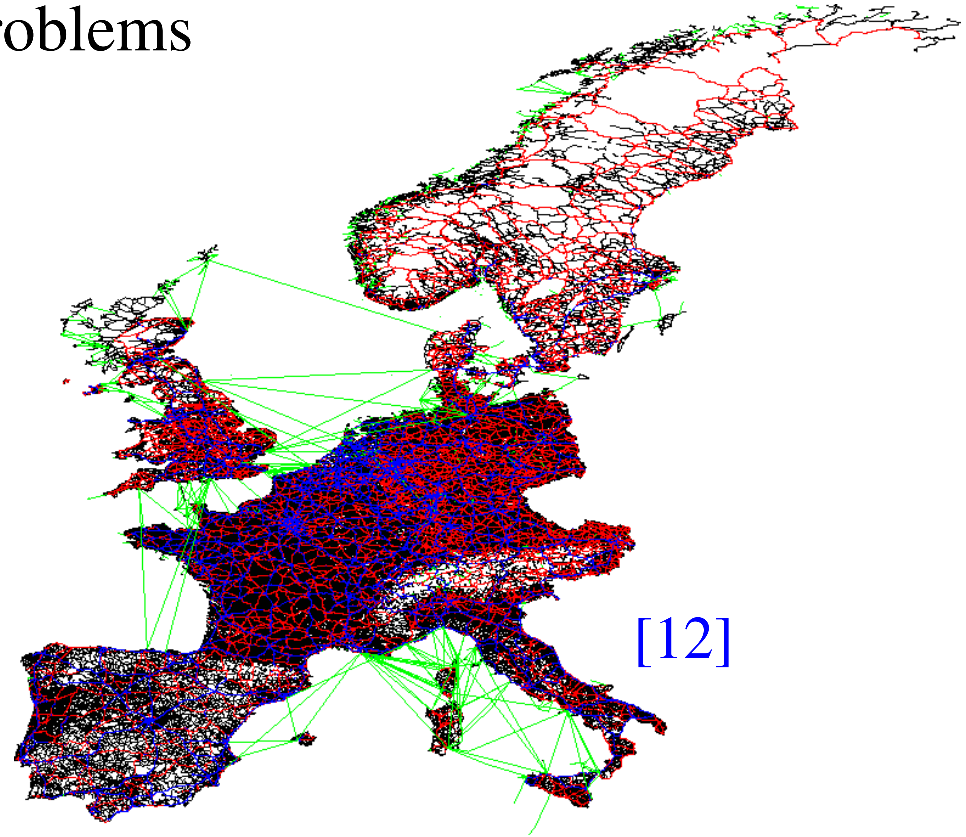



algorithm
engineering
realistic
models
inputs
benchmarks
design
falsifiable
hypotheses
induction
analysis
experiments
deduction
implementation
appl. engin.
performance
guarantees
algorithm
libraries
applications

Problem Instances

Benchmark instances for NP-hard problems

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



[12]

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)

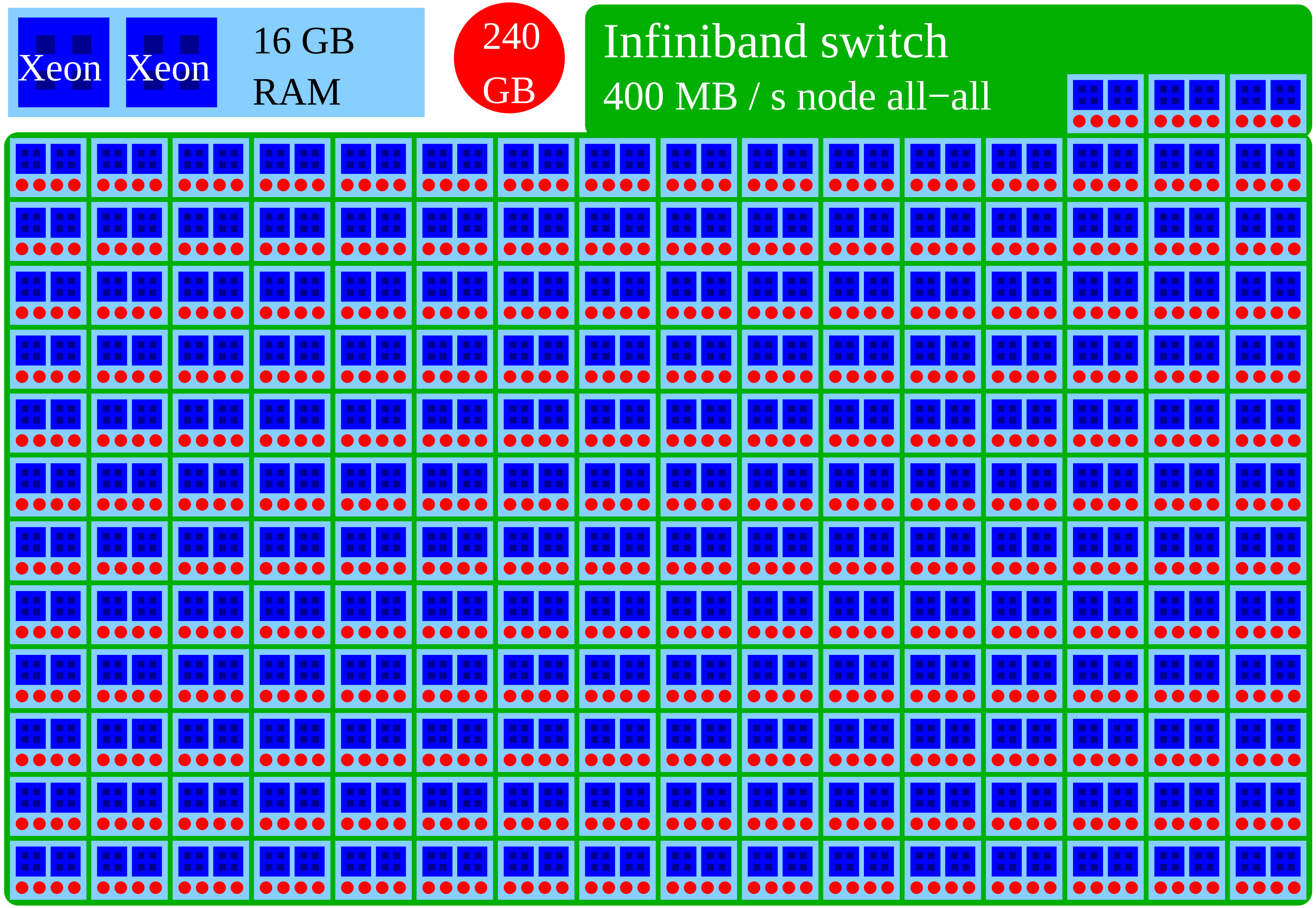
[5, 13]

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

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

Also: PennySort

GraySort: inplace multiway mergesort, exact splitting [5]

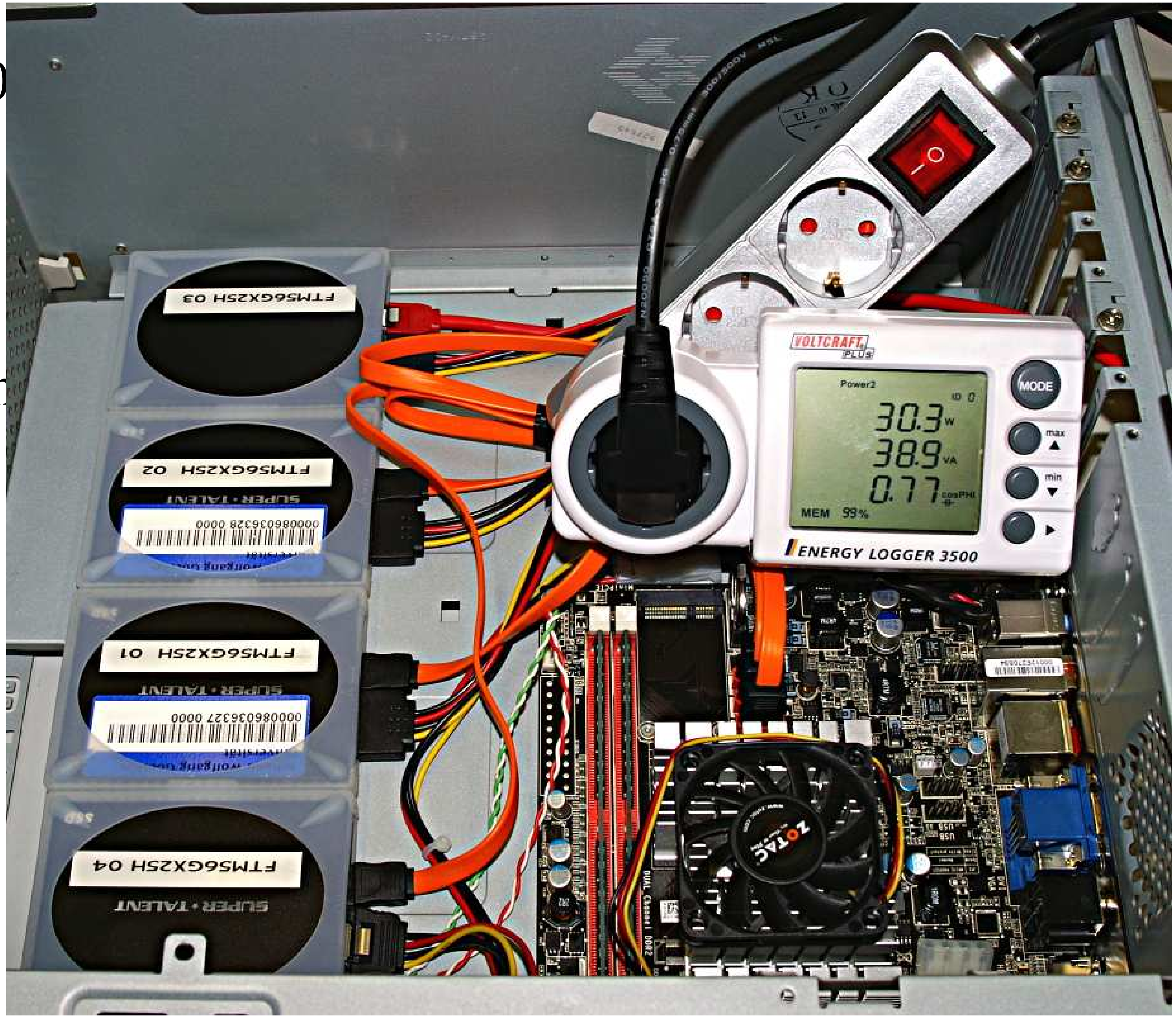


JouleSort

[13]

- Intel Atom N330
- 4 GB RAM
- 4×256 GB SSD (SuperTalen

Algorithm similar to GraySort



2 Sorting

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, 22]	
parallel string s.	[23, 24]	[25, 23]	[25, 23]	[25, 23]
massively parallel	[15]	[15, 16]	[16, 26]	

Sorting — Overview

- You think you understand quicksort?
- Avoiding branch mispredictions: Super Scalar Sample Sort
- (Parallel disk) external sorting. Perhaps not in detail this year

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$(\text{quickSort}(a), b, \text{quickSort}(c))$

Engineering Quicksort

- array
- 2-way-Comparisons
- sentinels for inner loop
- inplace swaps
- Recursion on smaller subproblems
  $\rightarrow \mathcal{O}(\log n)$ 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
        $j :=$ pickPivotPos(a, l, r)
        swap$(a[\ell], a[j])$ // Helps to establish the invariant
        $p := a[\ell]$
        $i := \ell$; $j := r$
        repeat // a: ℓ $i\rightarrow$ $\leftarrow j$ 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 $i < \frac{l+r}{2}$ **then** qSort(a, ℓ, j); $\ell := j$
        else qSort(a, i, r) ; $r := i$
    insertionSort$(a[l..r])$ // faster for small $r - \ell$

Picking Pivots Painstakingly — Theory

"How branch mispredictions affect quicksort" [21]

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

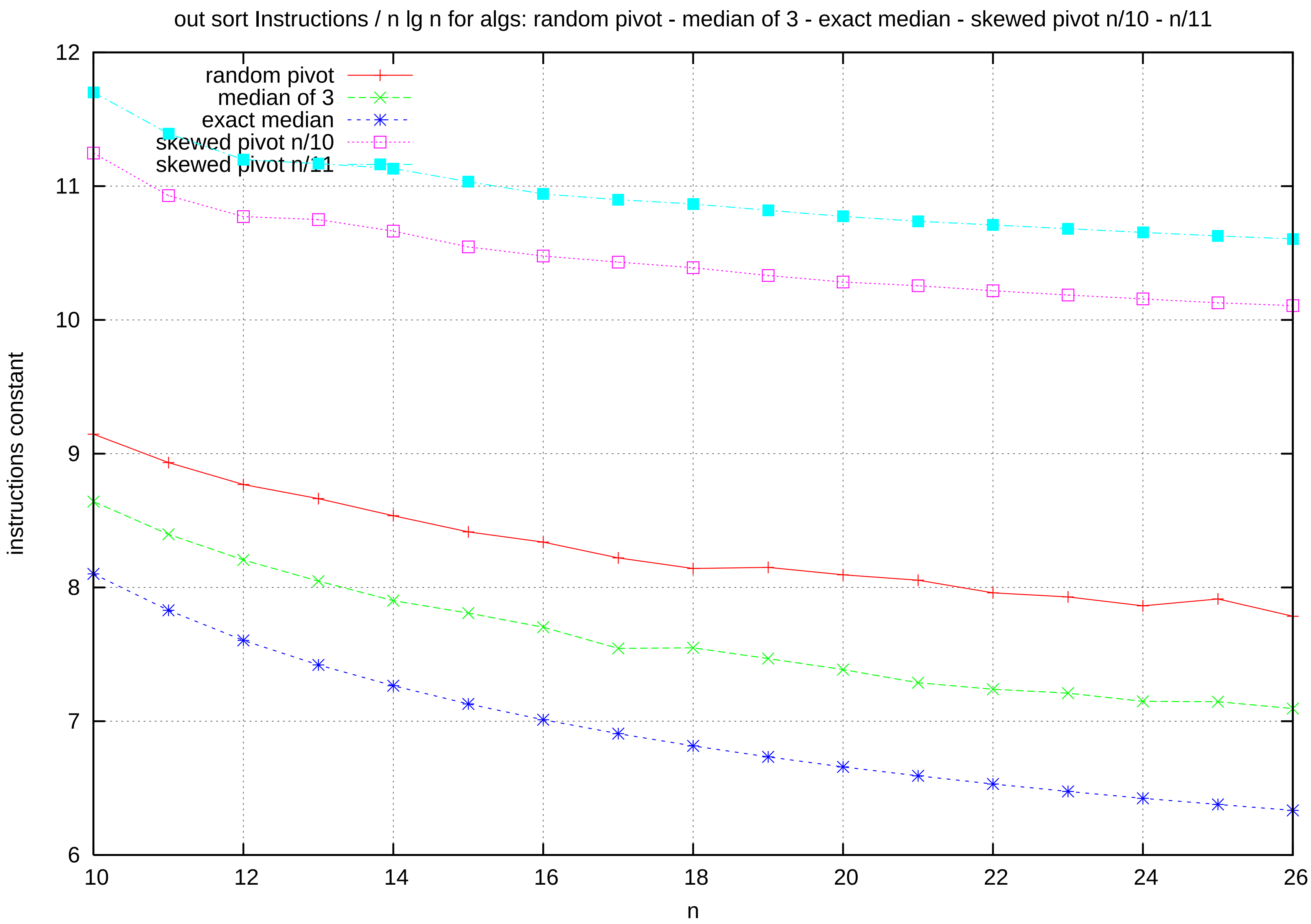
median of three: Expected $1.2n\log n$ element comparisons

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

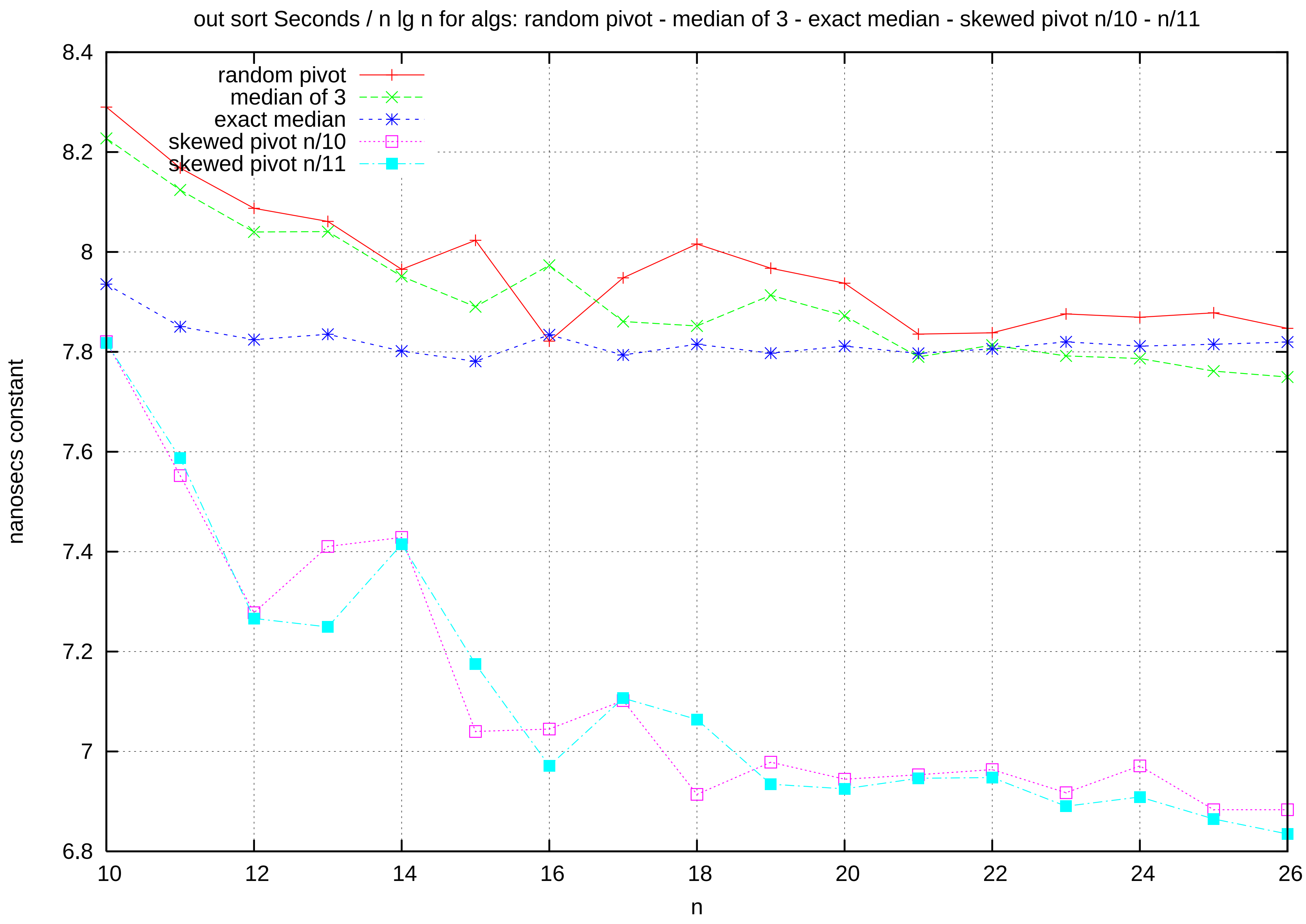
Practice

3GHz Pentium 4 Prescott, g++

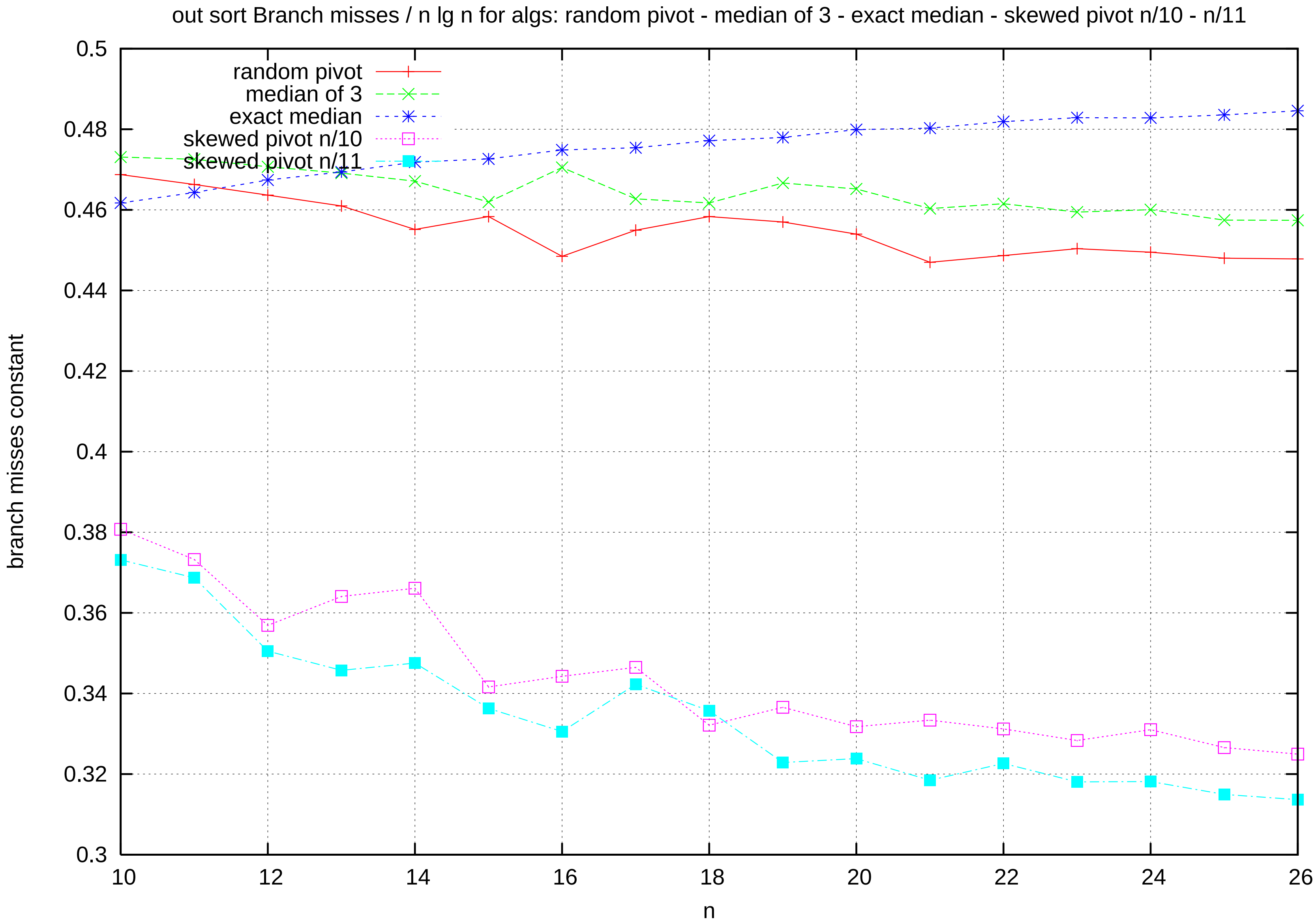
Picking Pivots Painstakingly — Instructions



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 laaarge inputs

Can We Do Better? Subsequent Work

Blockquicksort: Avoiding branch mispredictions in quicksort [27] (arxiv version is titled “BlockQuicksort: How Branch Mispredictions don’t affect Quicksort”

Sample Sort

Function sampleSort$(e = \langle e_1, \ldots, e_n \rangle, k)$

if n/k is “small” **then return** smallSort(e)

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

sort S

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

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

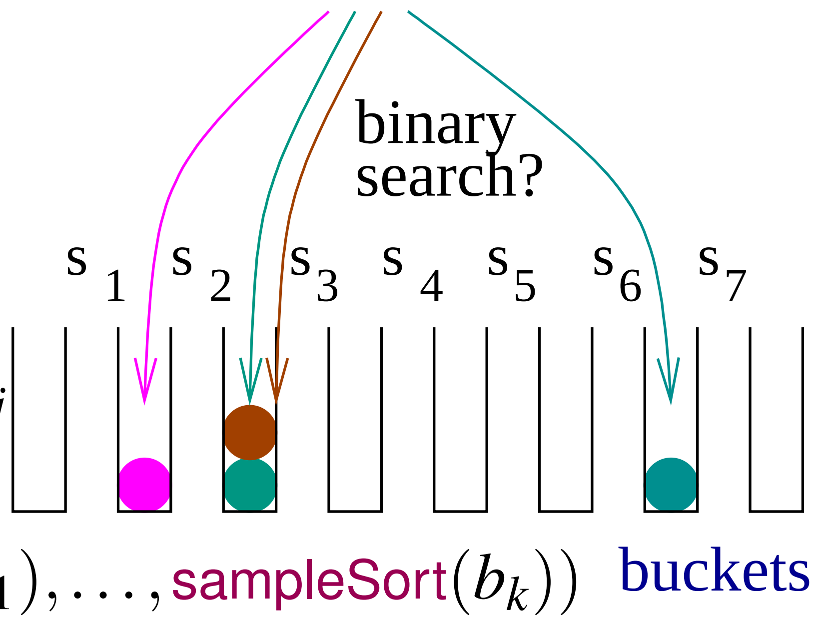
for $i := 1$ **to** n **do**

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

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

place e_i in bucket b_j

return concatenate(sampleSort$(b_1), \ldots,$ sampleSort(b_k))



Why Sample Sort?

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

Super Scalar Sample Sort

- □ Binary search ⟶ implicit search tree
- □ Eliminate all conditional branches
- ⇝ Exploit instruction parallelism
- ⇝ Cache efficiency comes to bear
- □ "steal" memory management from radix sort

Classifying Elements

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

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

  $j := 1$

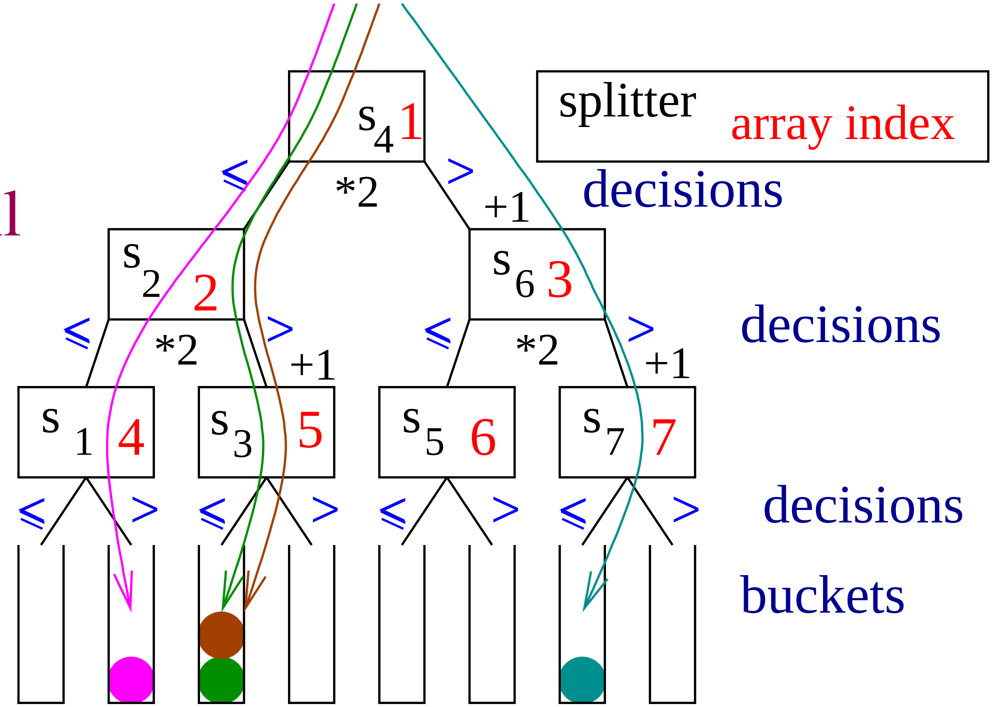
  repeat $\log k$ times // unroll

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

  $j := j - k + 1$

  $|b_j|{+}{+}$

  $\text{oracle}[i] := j$ // oracle



Now the compiler should:

- use predicated instructions
- interleave for-loop iterations (unrolling ∨ 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: if (r1 > r2) 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

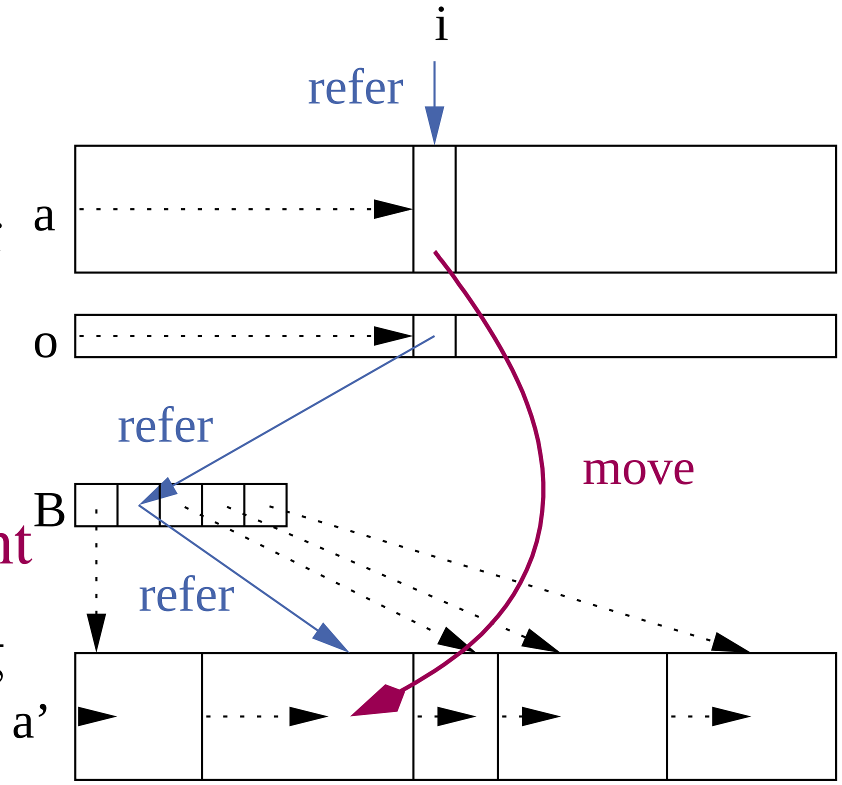
Unrolling ($k = 16$)

```
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** $a'_{B[\mathsf{oracle}[i]]++} := a_i$

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

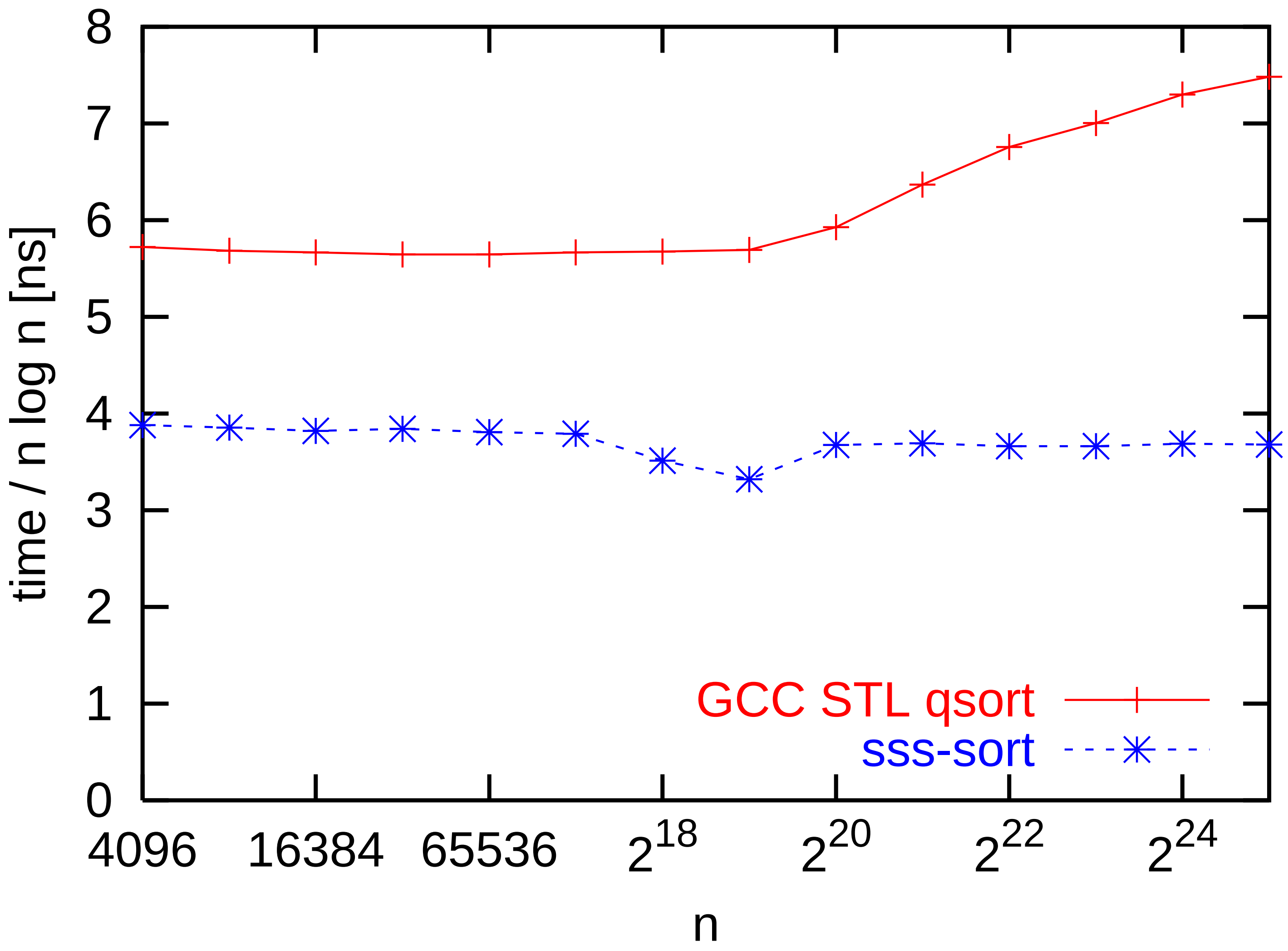
Distributing Elements

```
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];
  }
}
```

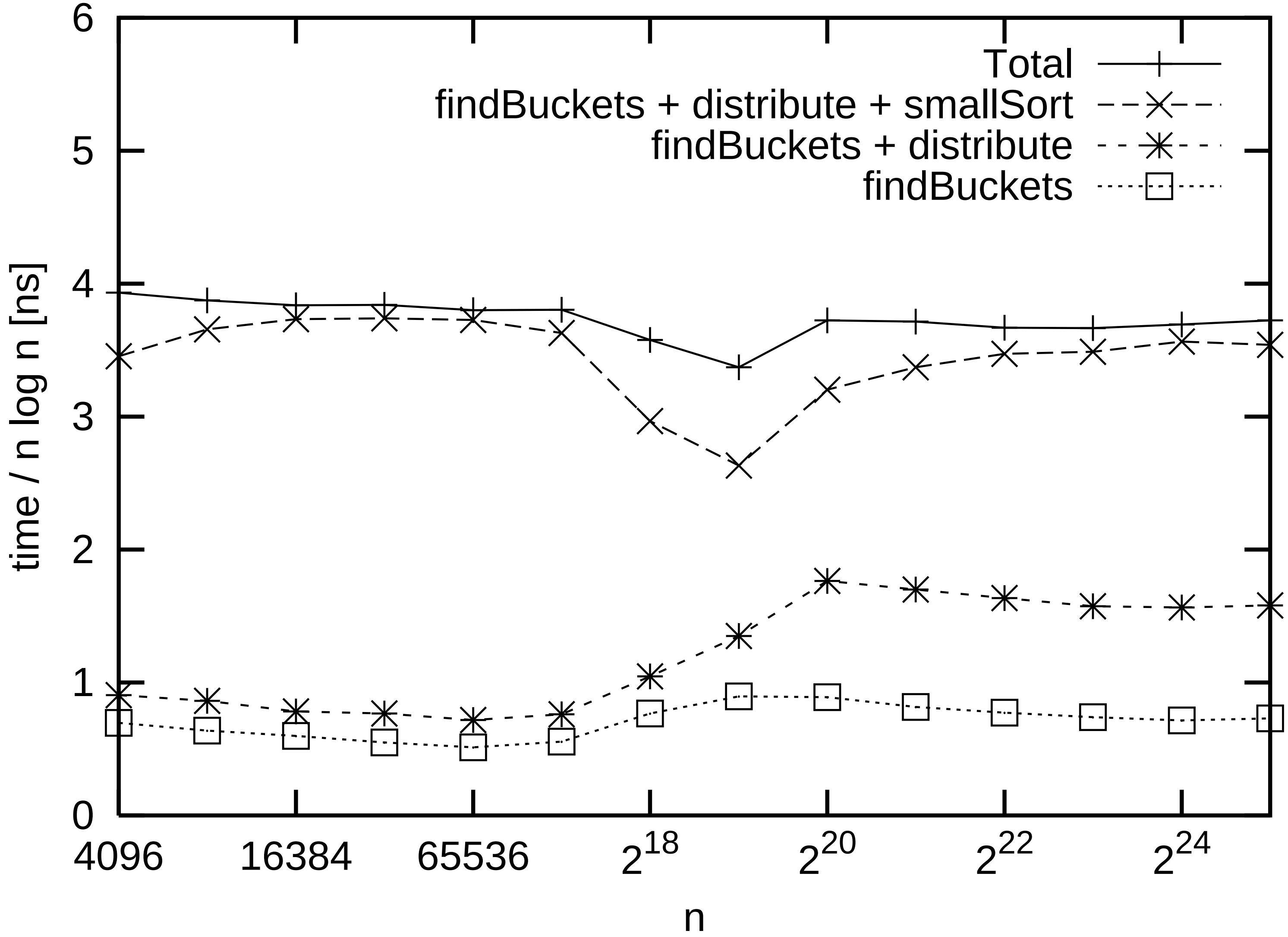
Experiments: 1.4 GHz Itanium 2

- `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 `stl:sort` from g++ ($n \leq 1000$)!
- insertion sort for $n \leq 100$
- Random 32 bit integers in $[0, 10^9]$

Comparison with Quicksort



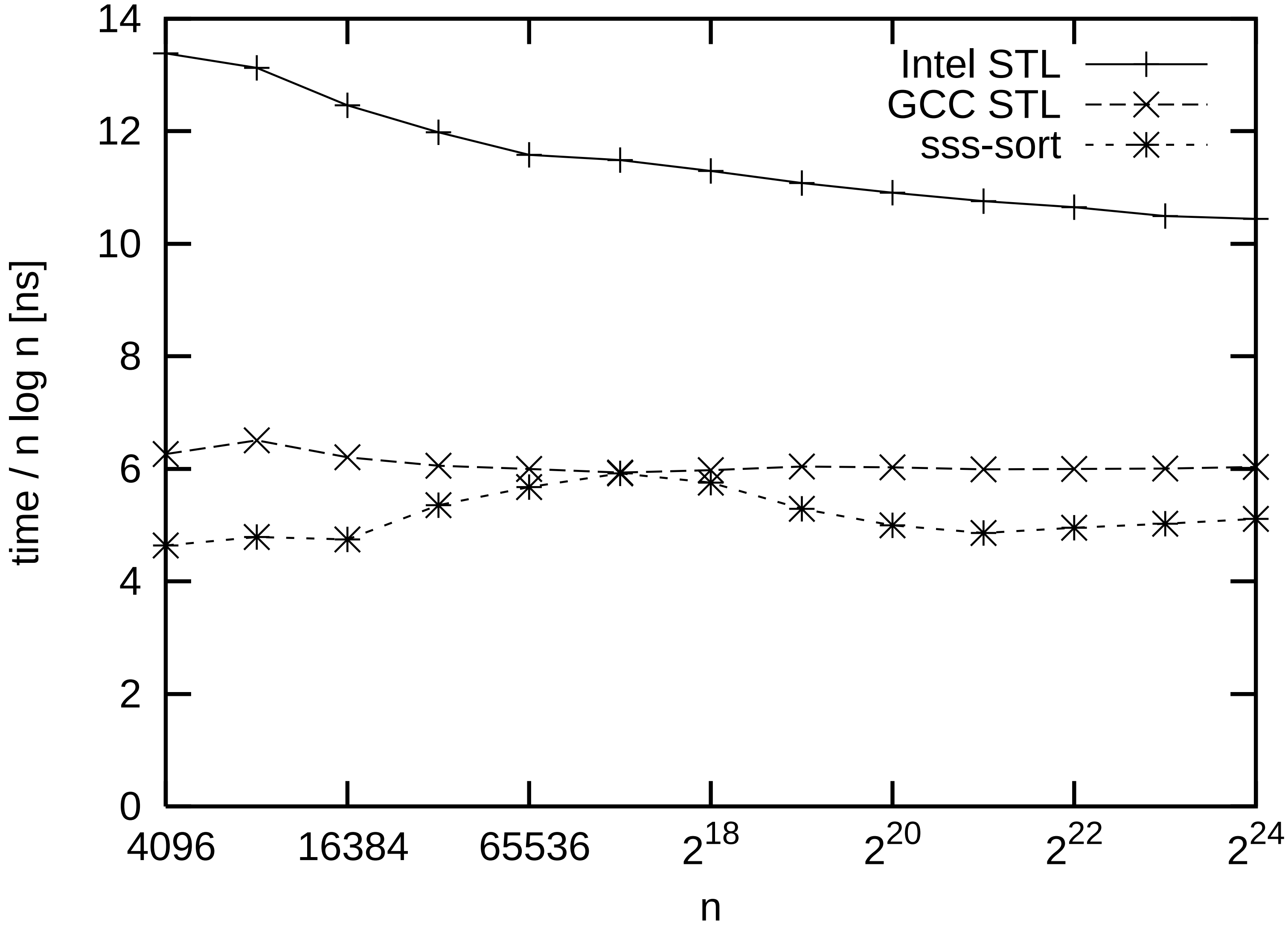
Breakdown of Execution Time



A More Detailed View

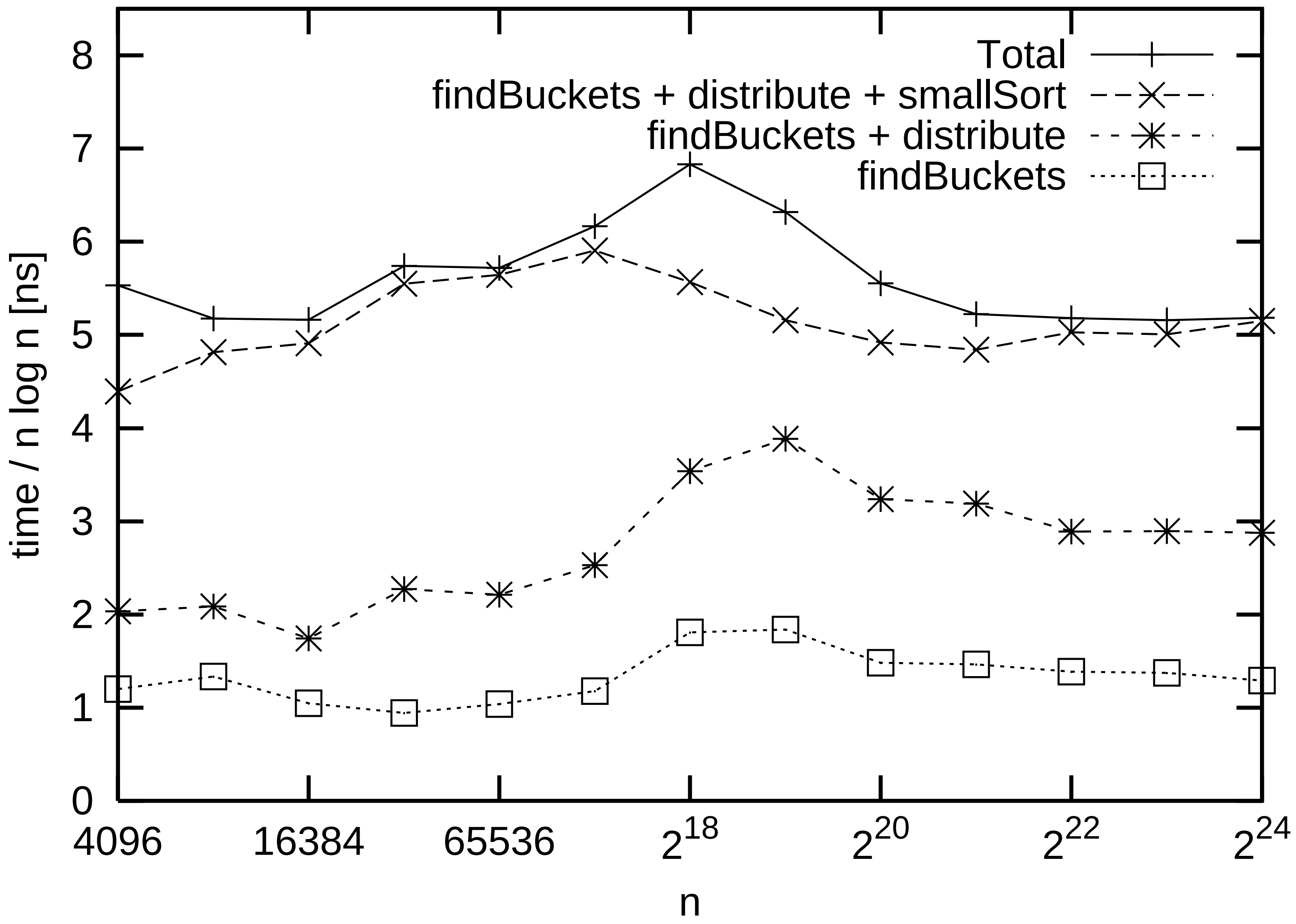
	instr.	cycles	dynamic IPC small n	dynamic IPC $n = 2^{25}$
findBuckets, $1\times$ outer loop	63	11	5.4	4.5
distribute, one element	14	4	3.5	0.8

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
k-way distribution:						
sss-sort	$n\log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\geq 3.5n/B$	$3\times$unroll	$\mathcal{O}(\log k)$
IS^4o	$n\log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$4n/B$	$3\times$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	$\mathcal{O}(1)$
k-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:

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

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

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

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

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

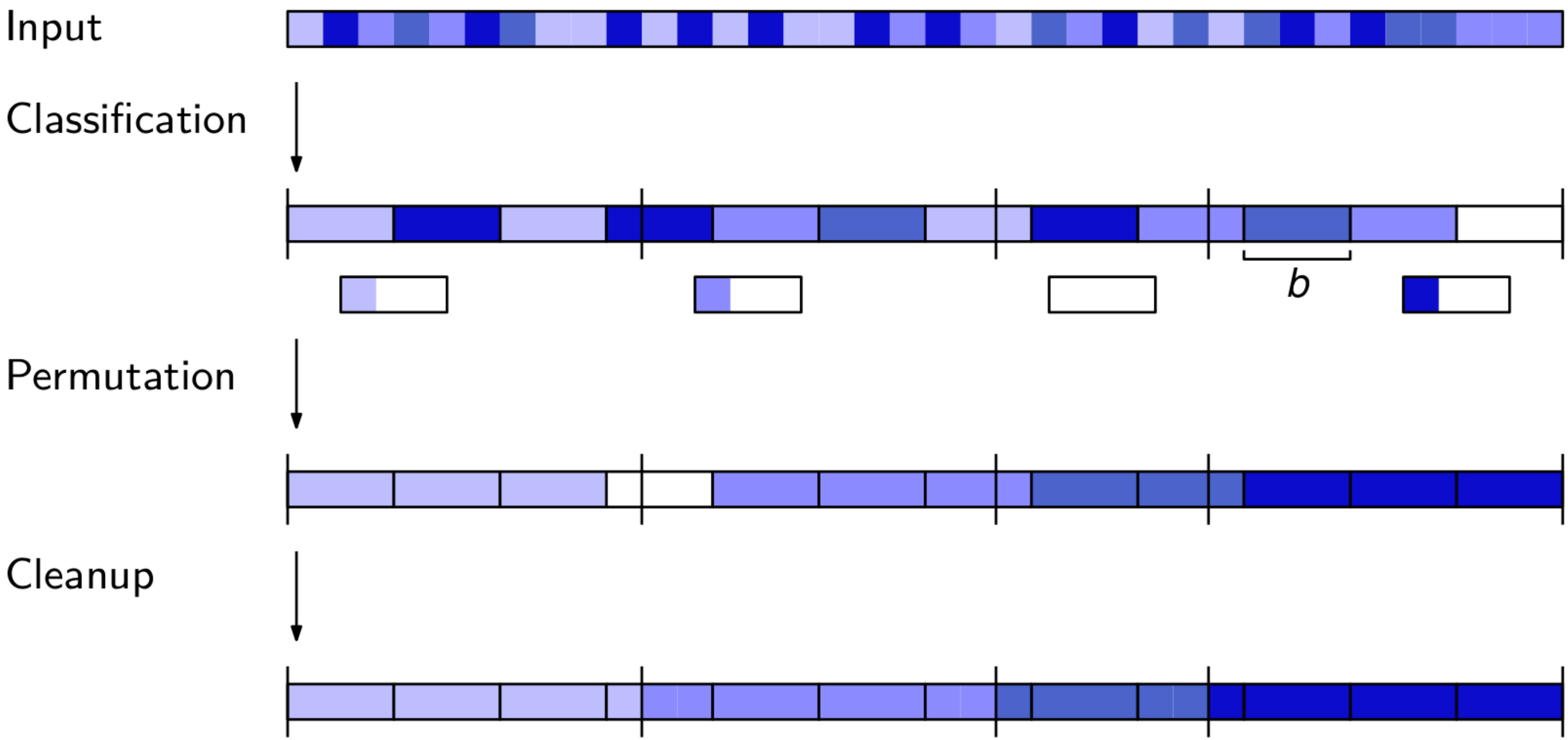
Criticism II

Quicksort is inplace

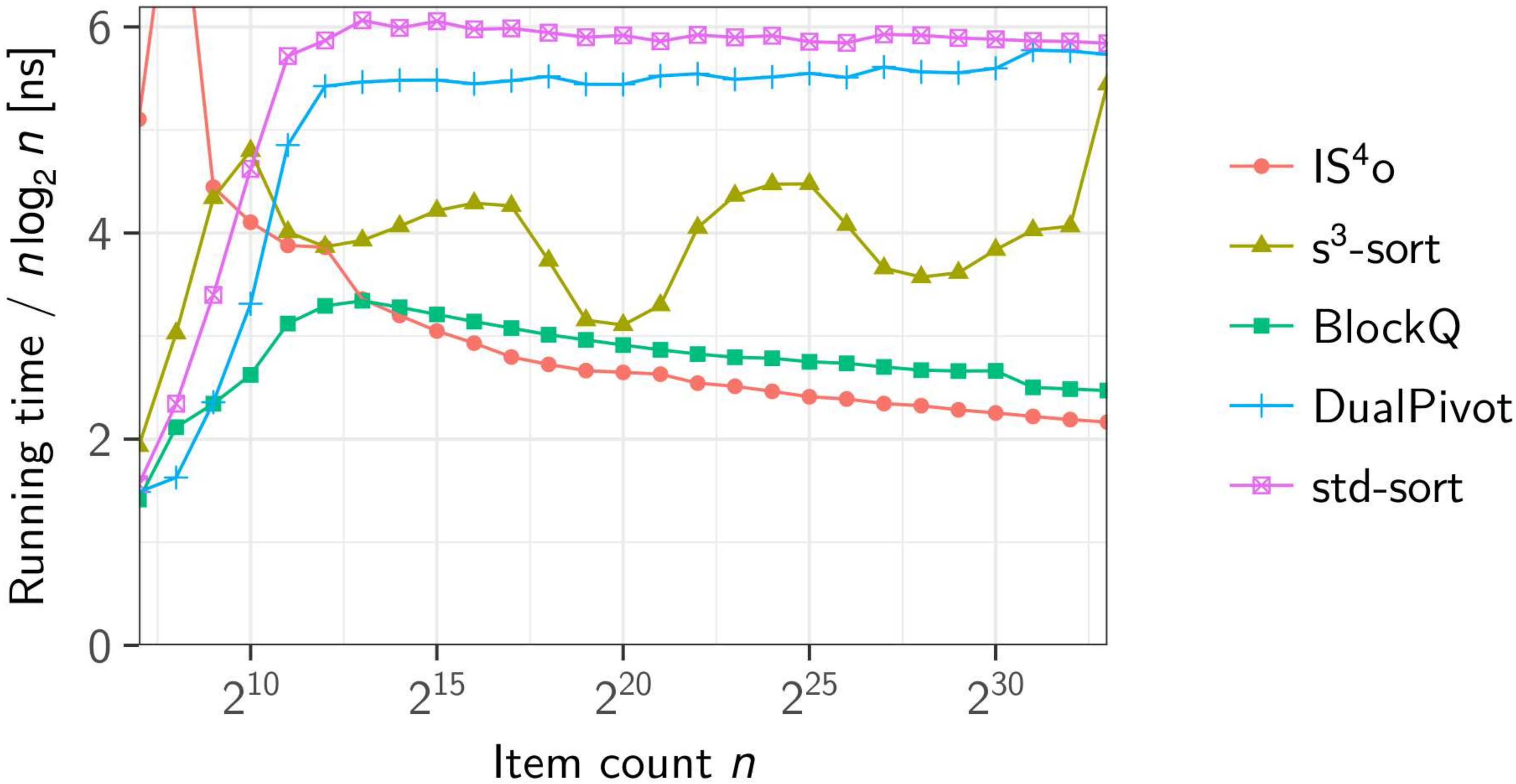
Answer II

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

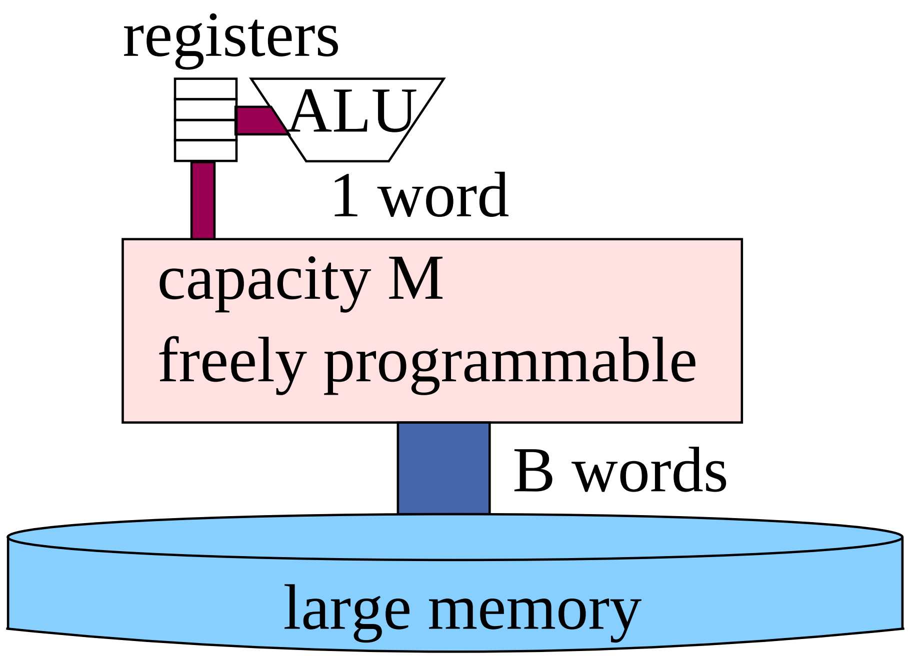
- better small case sorter for arbitrary keys/comparators (for small numbers, SIMD, sorting networks etc. give good base case sorters)
- SIMD-instructions for distribution
- multilevel cache-aware or cache-oblivious generalization
- thorough testing $\rightsquigarrow$ verification? or back to simplicity?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

Externes Sortieren

n: input size

M: size of fast memory

B: block size



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

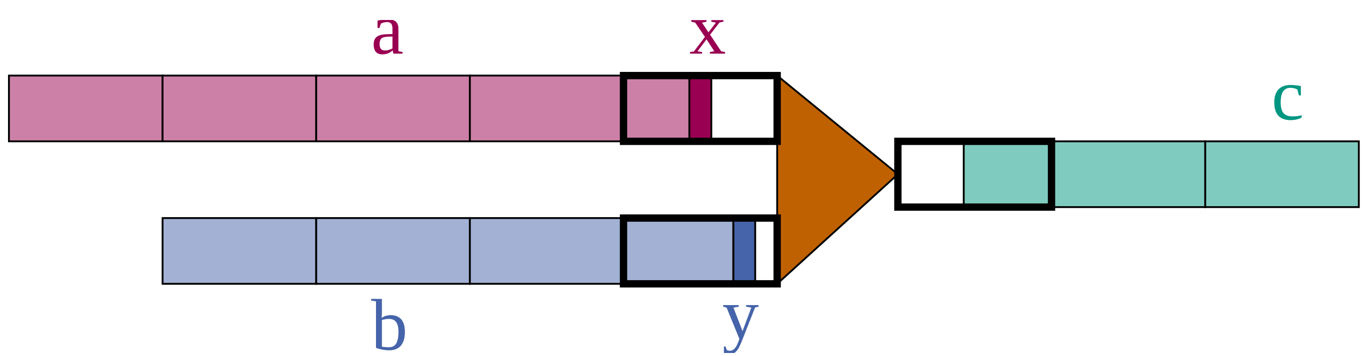
$x := a$.readElement // Assume emptyFile.readElement$= \infty$

$y := b$.readElement

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

  if $x \leq y$ **then** c.writeElement(x); $x := a$.readElement

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



External (binary) Merging– I/O-Analysis

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

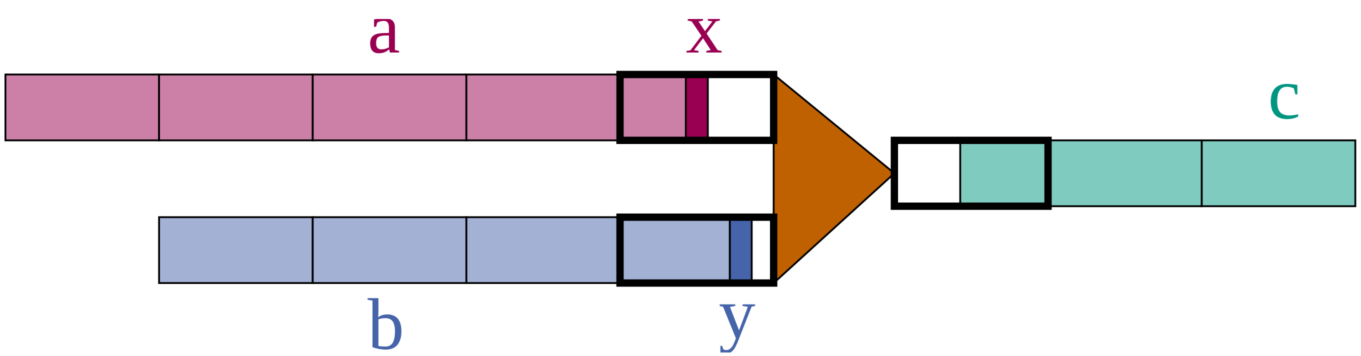
Datei b lesen: $\lceil |b|/B \rceil \leq |b|/B + 1$.

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

All together:

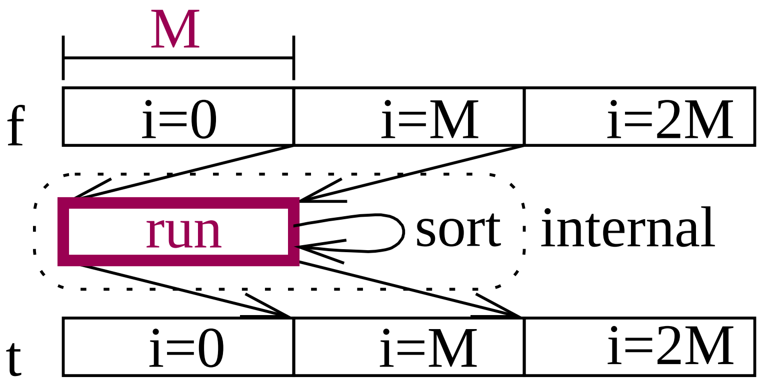
$$\leq 3 + 2\frac{|a| + |b|}{B} \approx 2\frac{|a| + |b|}{B}$$

Constraint: We need 3 buffer blocks, i.e., $M > 3B$.



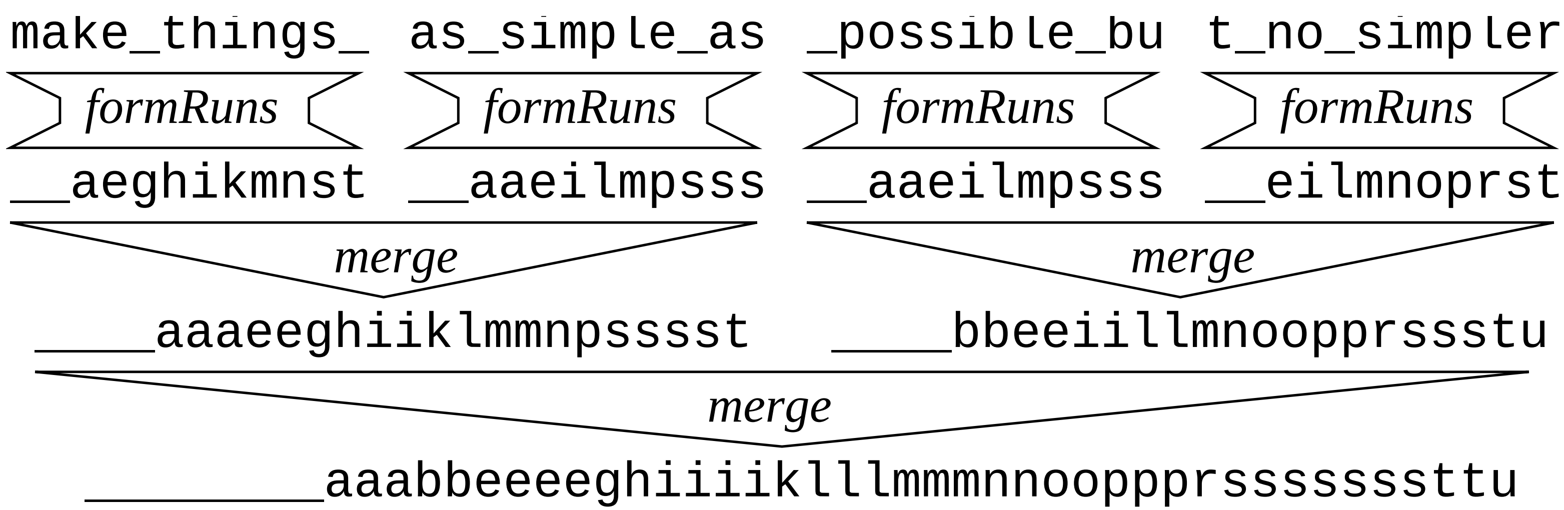
Run Formation

Sortiere Eingabeportionen der Größe M



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

Zahlenbeispiel: PC 2007

$n = 2^{38}$ Byte

$M = 2^{31}$ Byte

$B = 2^{20}$ Byte

I/O braucht 2^{-6} s

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

$n = 2^{38 \to 41}$ Byte

$M = 2^{31 \to 34}$ Byte

$B = 2^{20 \to 22}$ Byte

I/O braucht 2^{-5} s

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($a_1, \ldots, a_k, c$:File **of** Element)

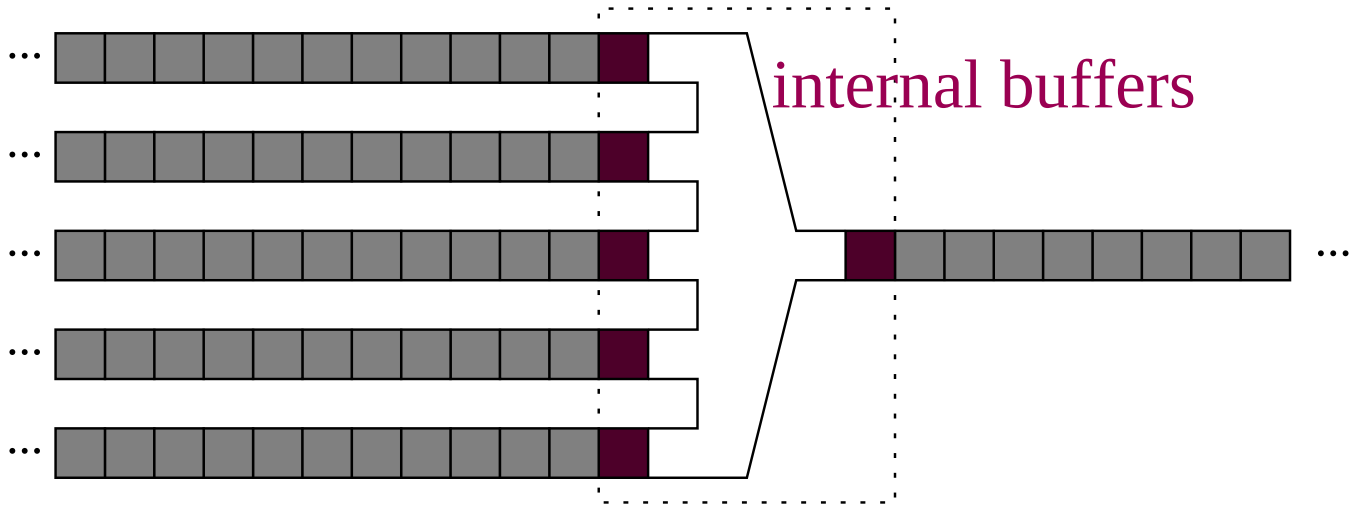
for $i := 1$ **to** k **do** $x_i := a_i$.readElement

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

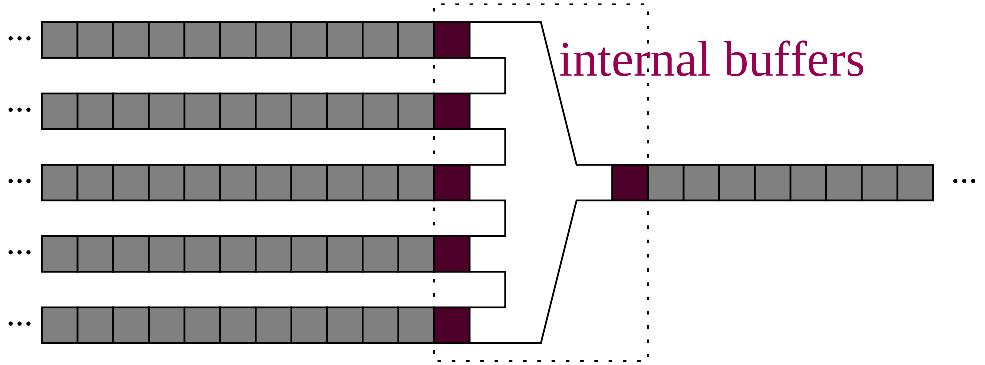
find $i \in 1..k$ that minimizes x_i// no I/Os!, $\mathcal{O}(\log k)$ time

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

```
make_things_  as_simple_as  _possible_bu  t_no_simpler
  formRuns      formRuns      formRuns      formRuns
__aeghikmnst  __aaeilmpsss  __aaeilmpsss  __eilmnoprst
                       multi merge
    ________aaabbeeeeghiiiiklllmmmnnooppprsssssssttu
```

Sortieren durch Mehrwege-Mischen

Interne Arbeit:

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

Mehr als eine Mischphase?:

Nicht für Hierarchie Hauptspeicher, Festplatte.

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

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

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]

Obere 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 Sanders2003]

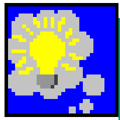
Offene Frage: deterministisch?

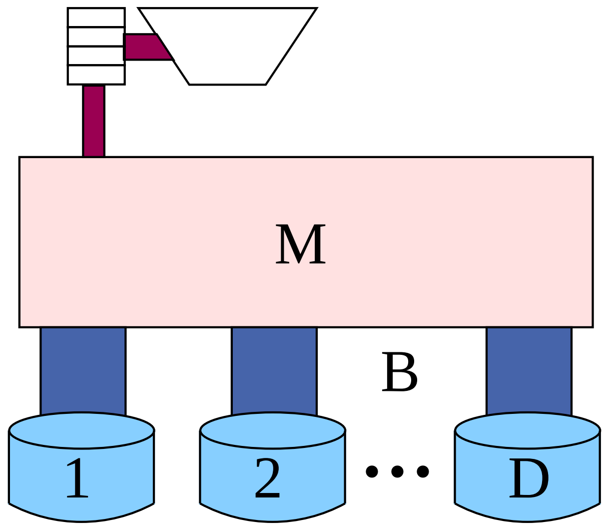
Sorting with Parallel Disks

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

Theory: Balance Sort [Nodine Vitter 93].

Deterministic, complex

asymptotically optimal  $\mathcal{O}(\cdot)$



independent disks

[Vitter Shriver 94]

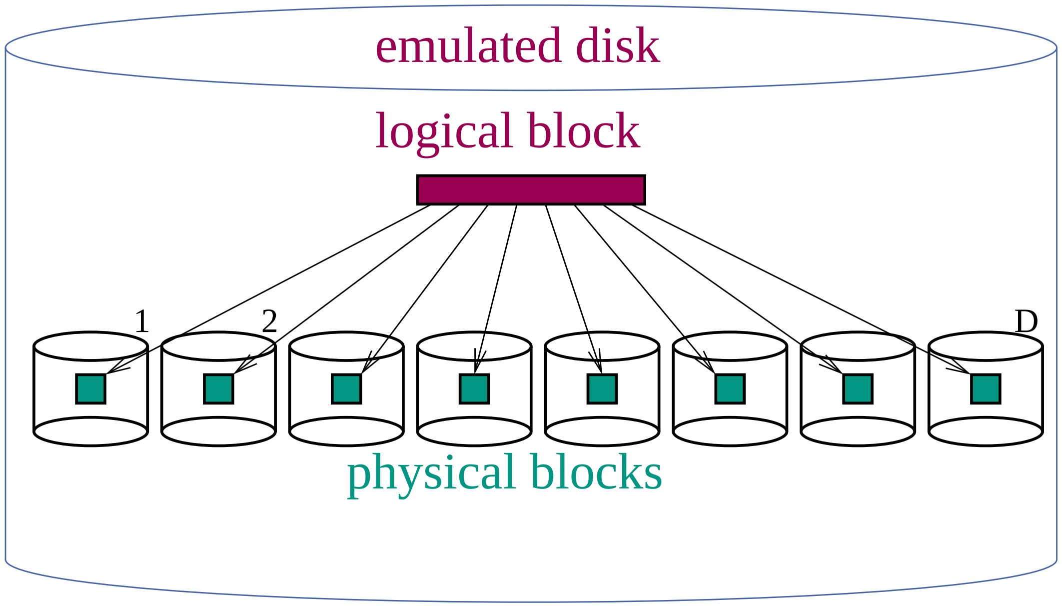
Multiway merging

"Usually" factor 10? less I/Os.

Not asymptotically optimal. 42%

Basic Approach: Improve Multiway Merging

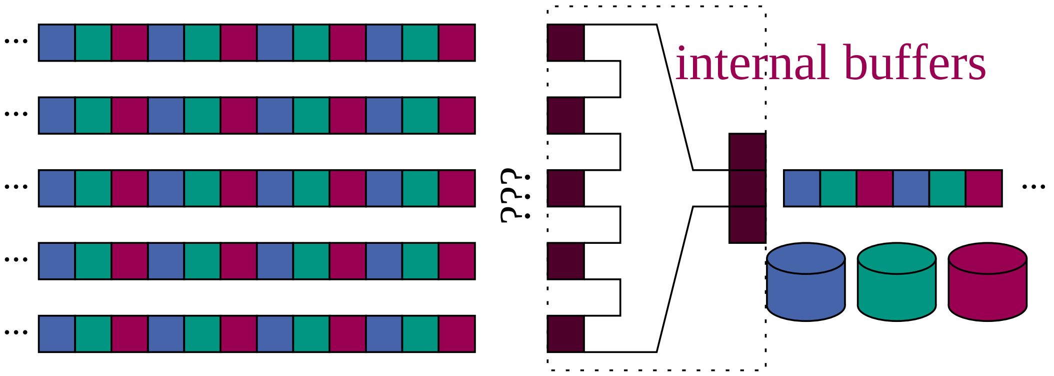
Striping



That takes care of run formation

and writing the output

But what about merging?



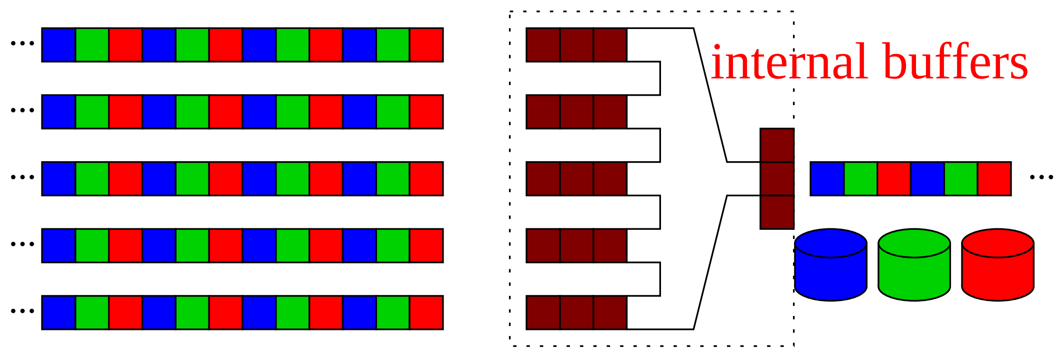
Naive Striping

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 $D \approx M/B$

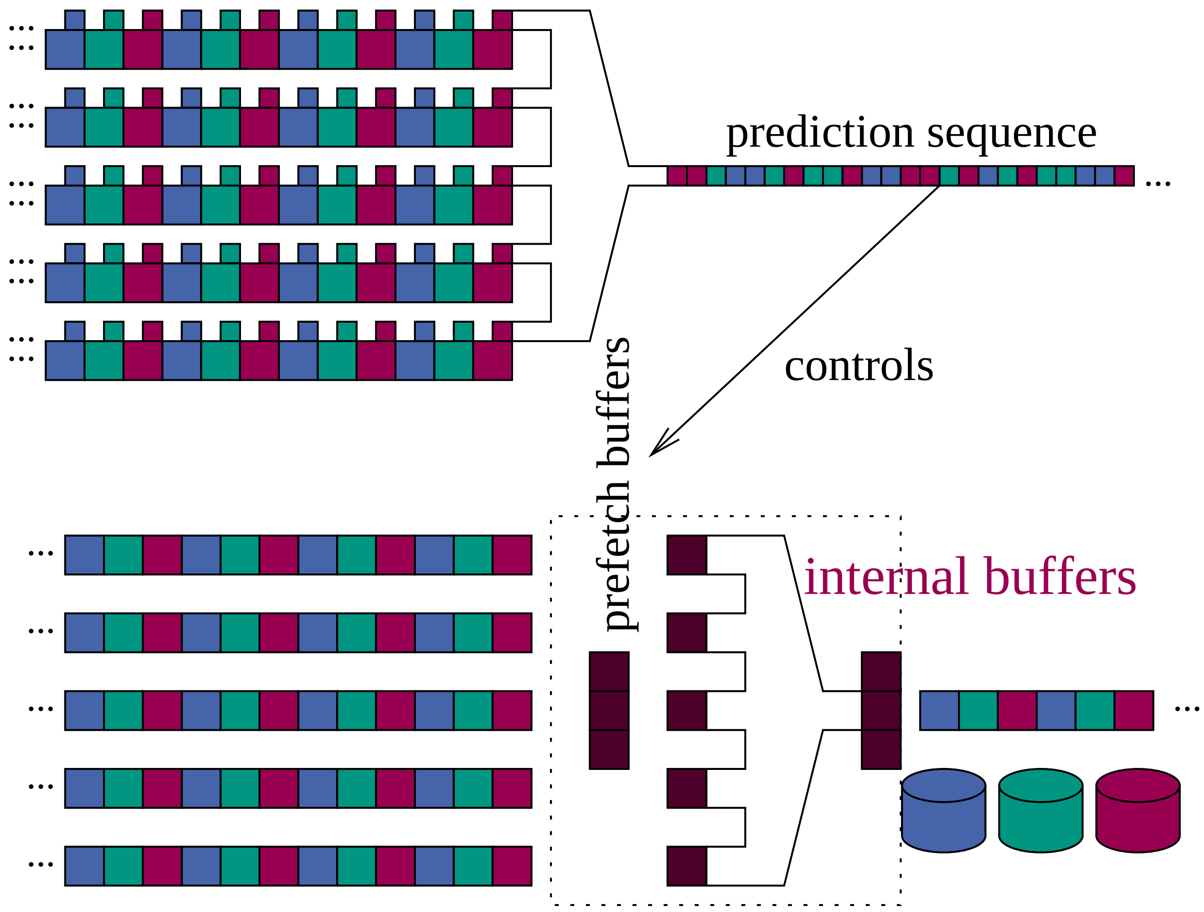
Practice: $2 \rightarrow 3$ passes in some cases



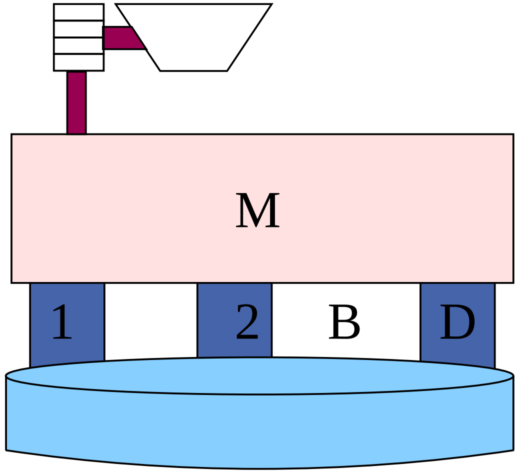
Prediction

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

Prefetch buffers
allow parallel access
of next blocks

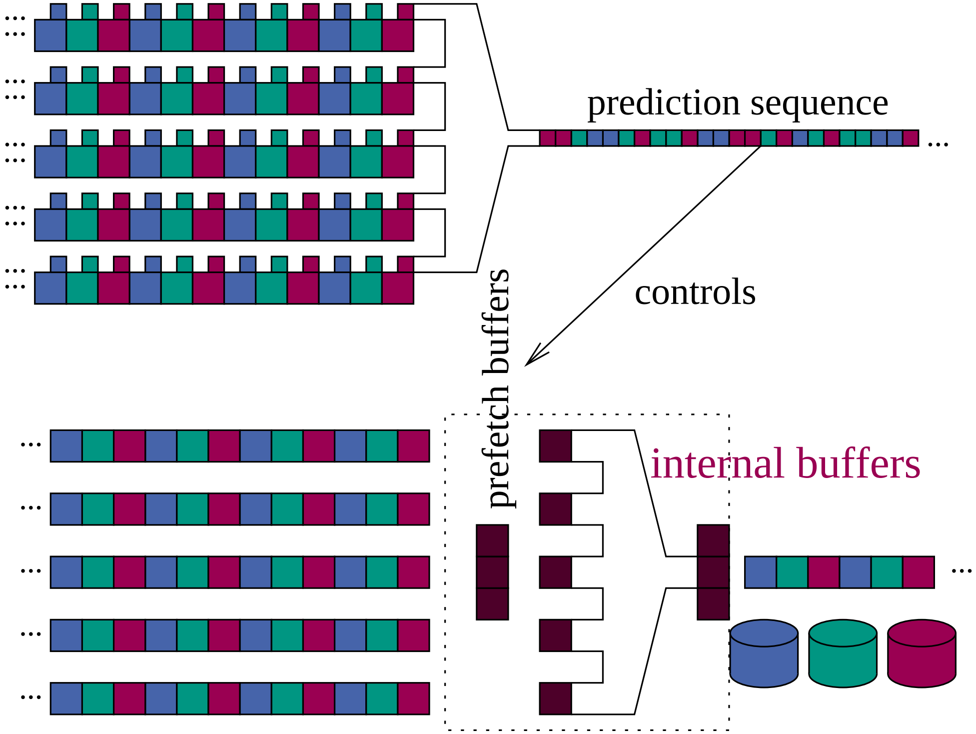


Warmup: Multihead Model



Multihead Model

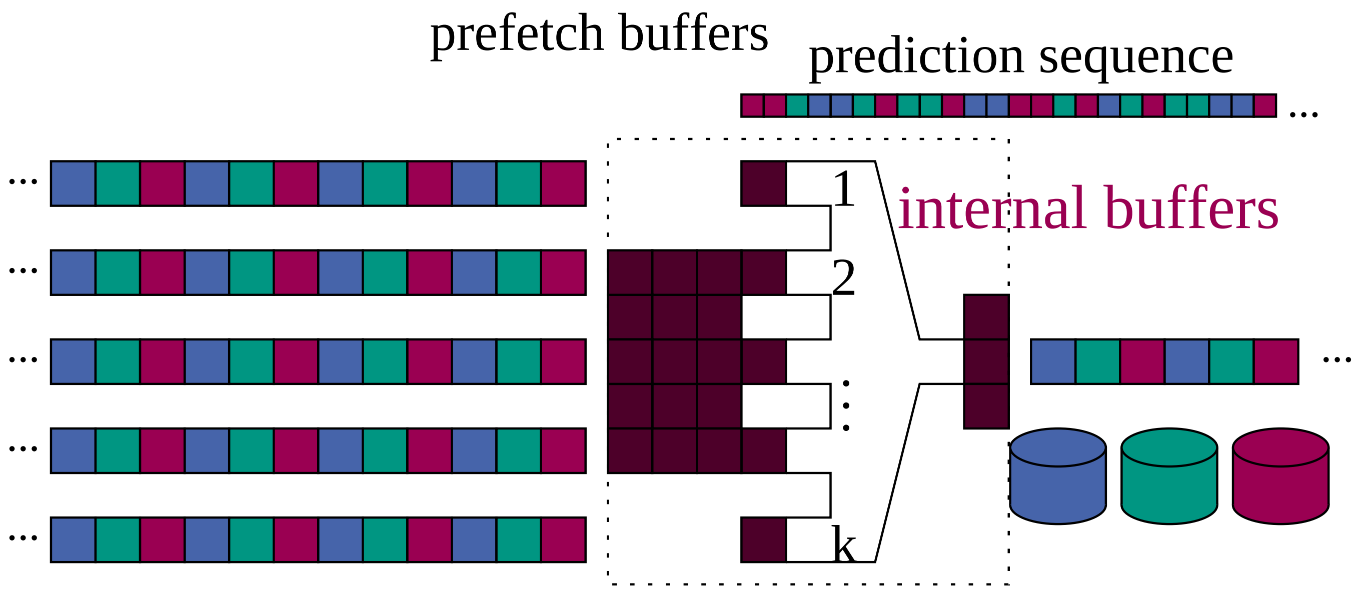
[Aggarwal Vitter 88]



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

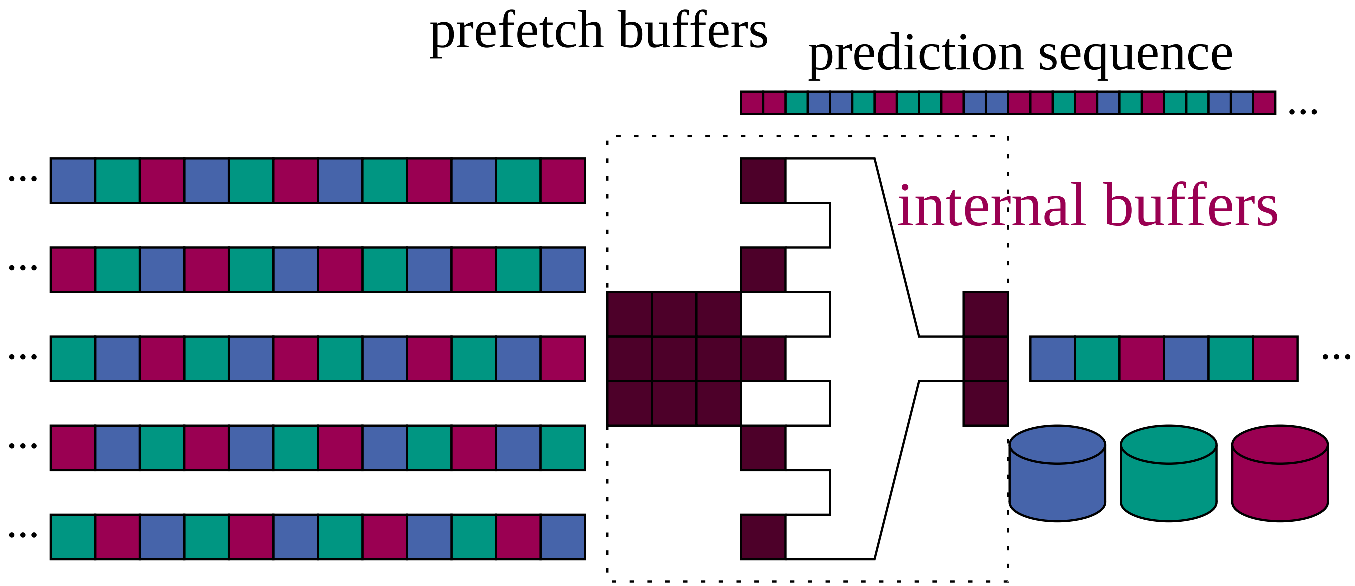
$\mathcal{O}(D)$ would yield an optimal algorithm.

Possible?

Randomized Cycling

[Vitter Hutchinson 01]

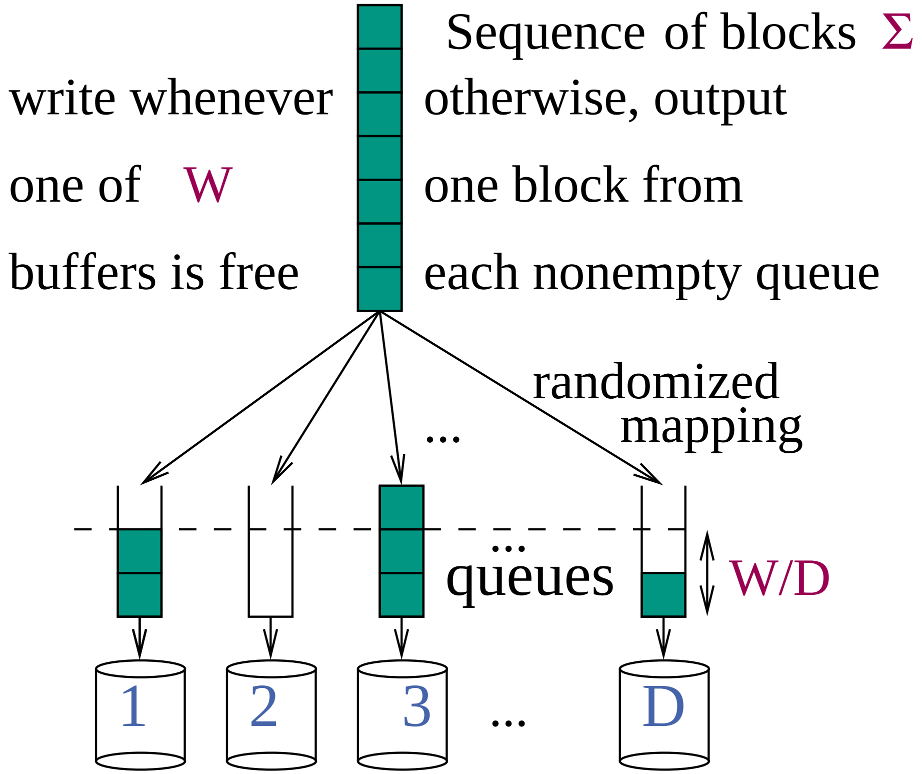
Block i of stripe j goes to disk $\pi_j(i)$ for a rand. permutation π_j



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

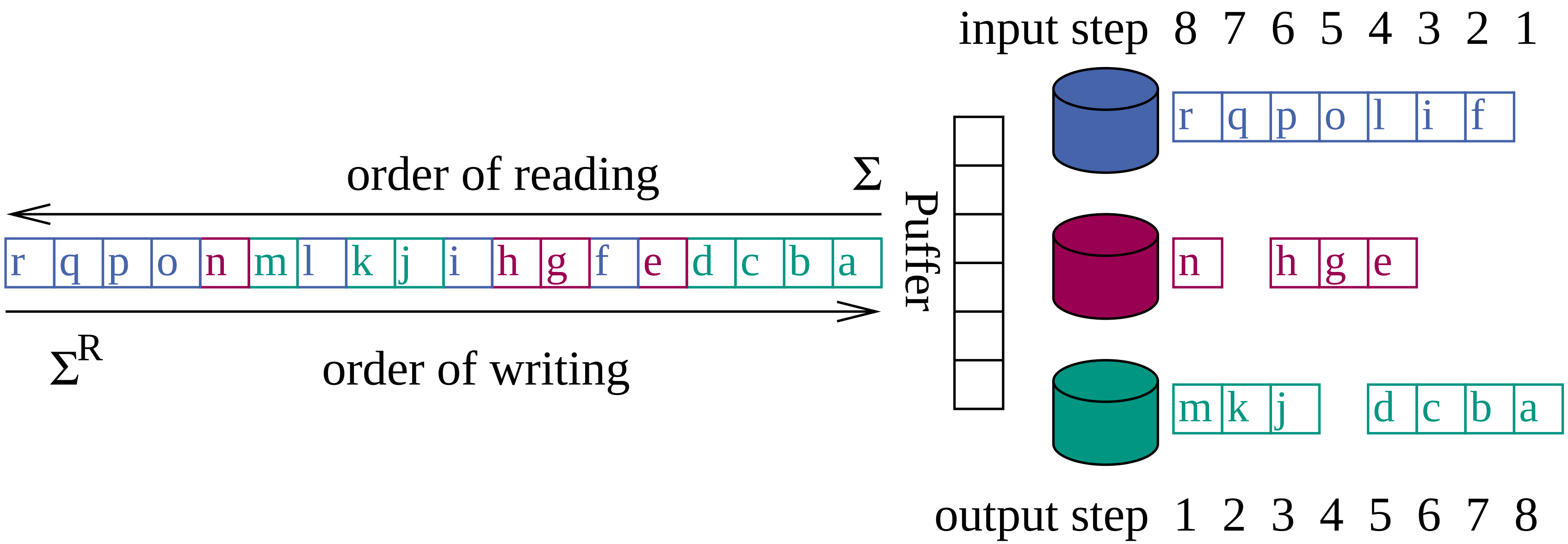
Theorem:

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

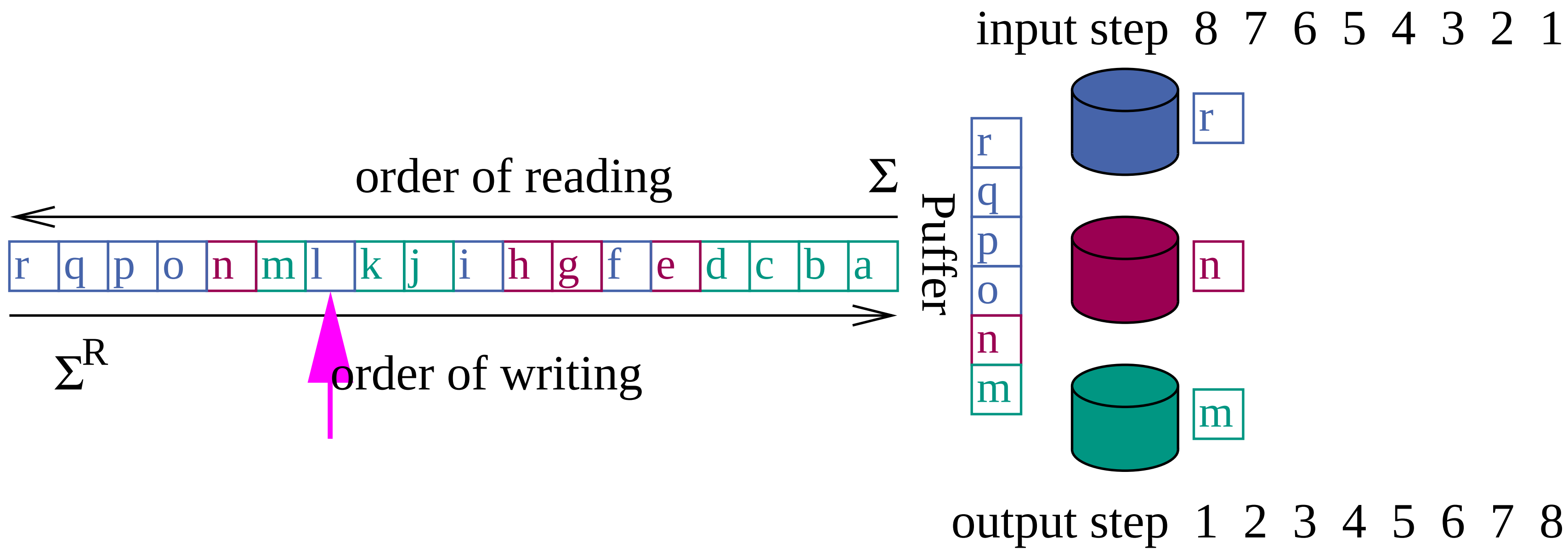
Theorem:

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

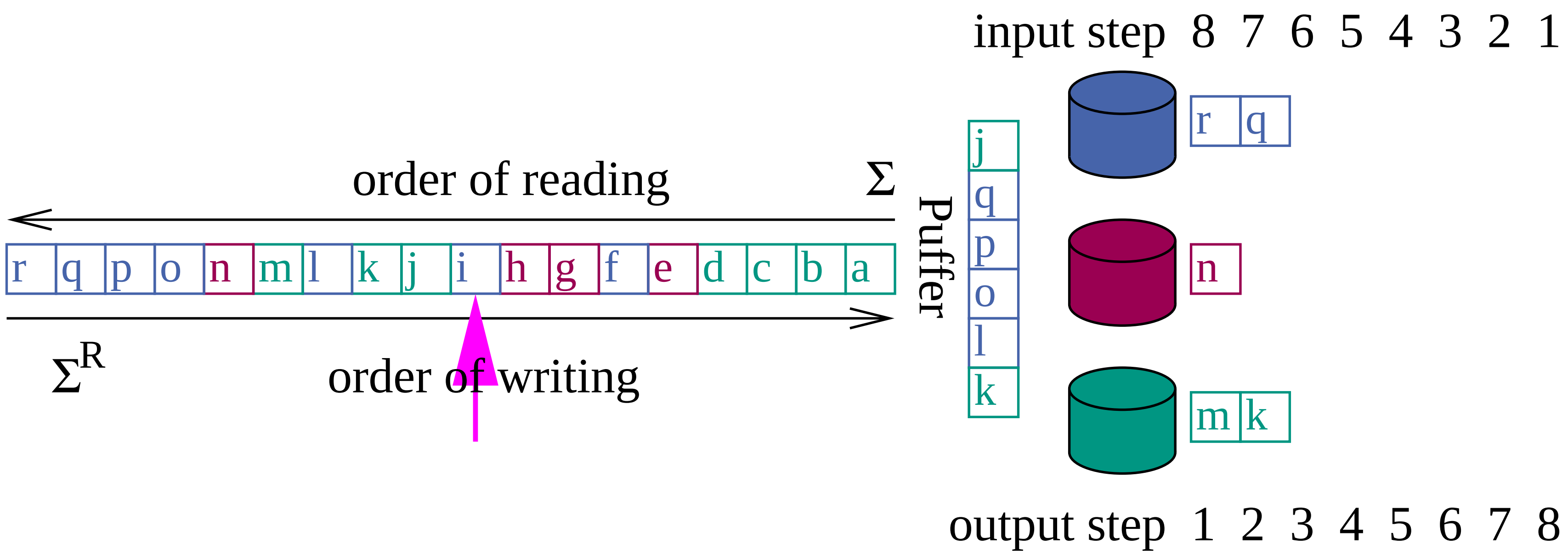
Theorem:

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

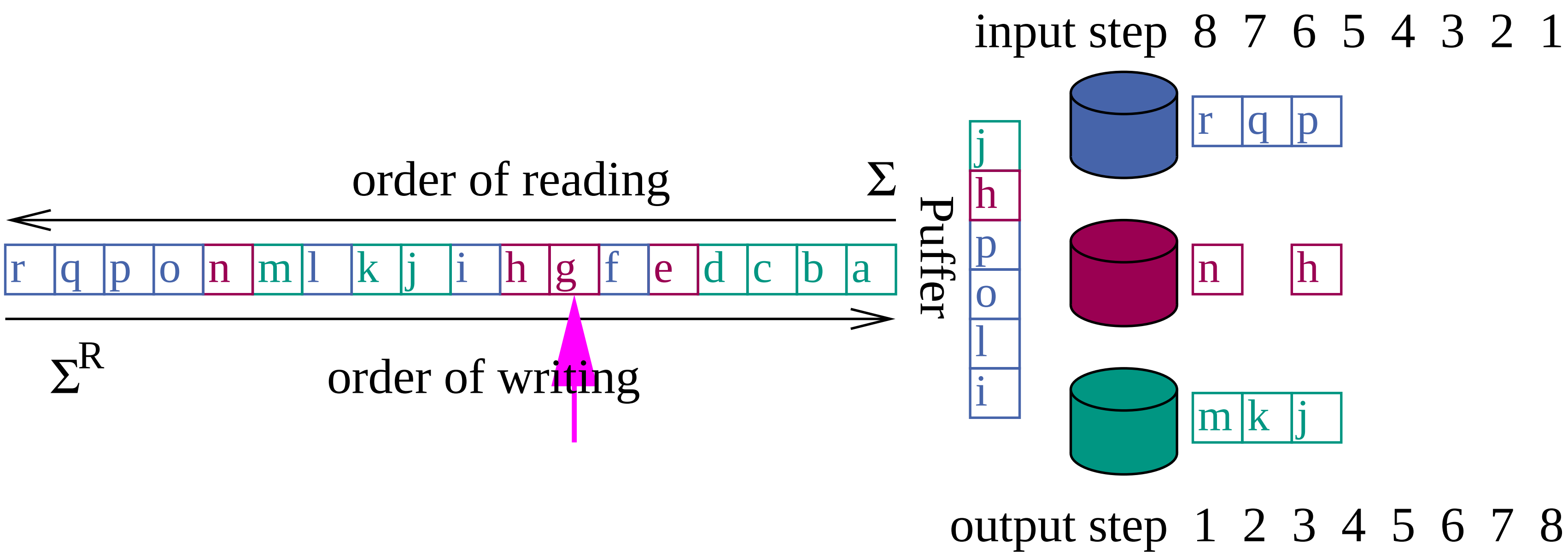
Theorem:

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

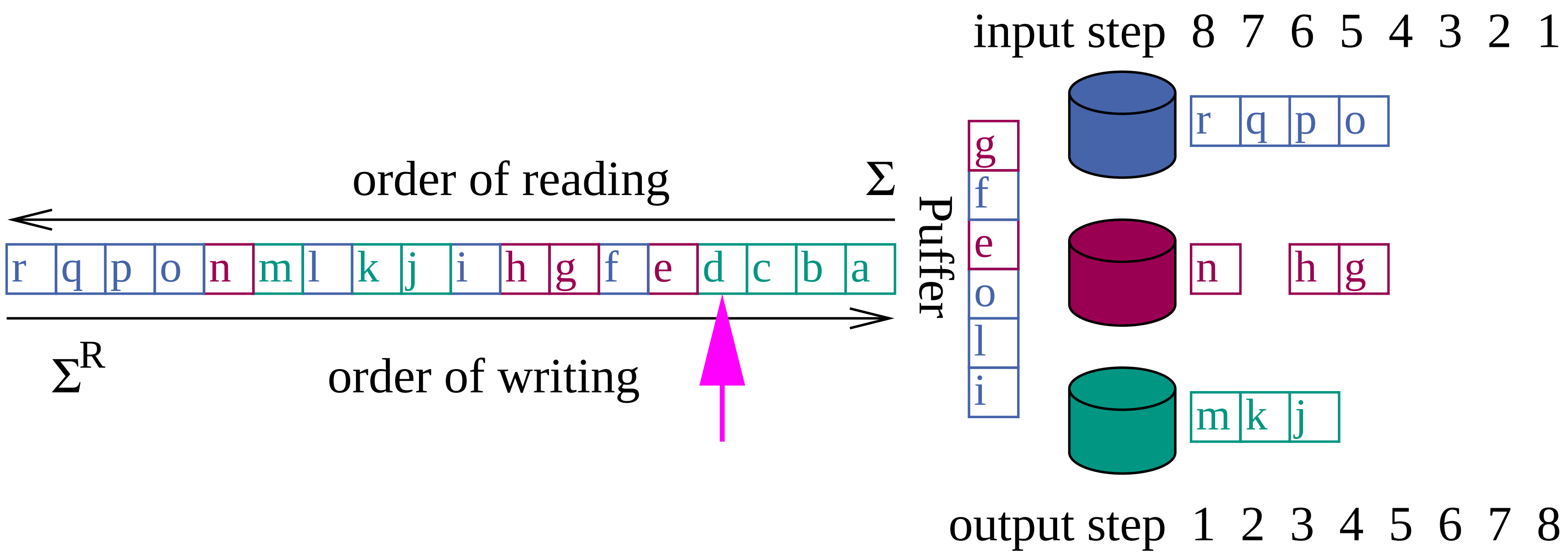
Theorem:

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

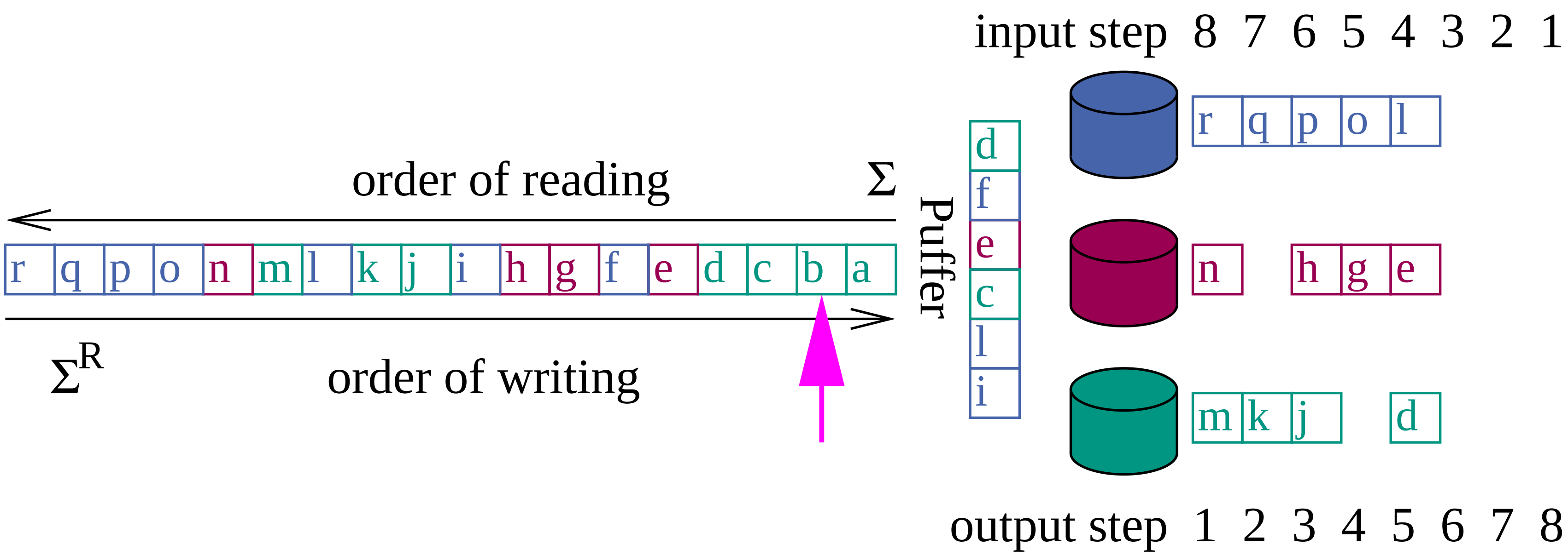
Theorem:

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

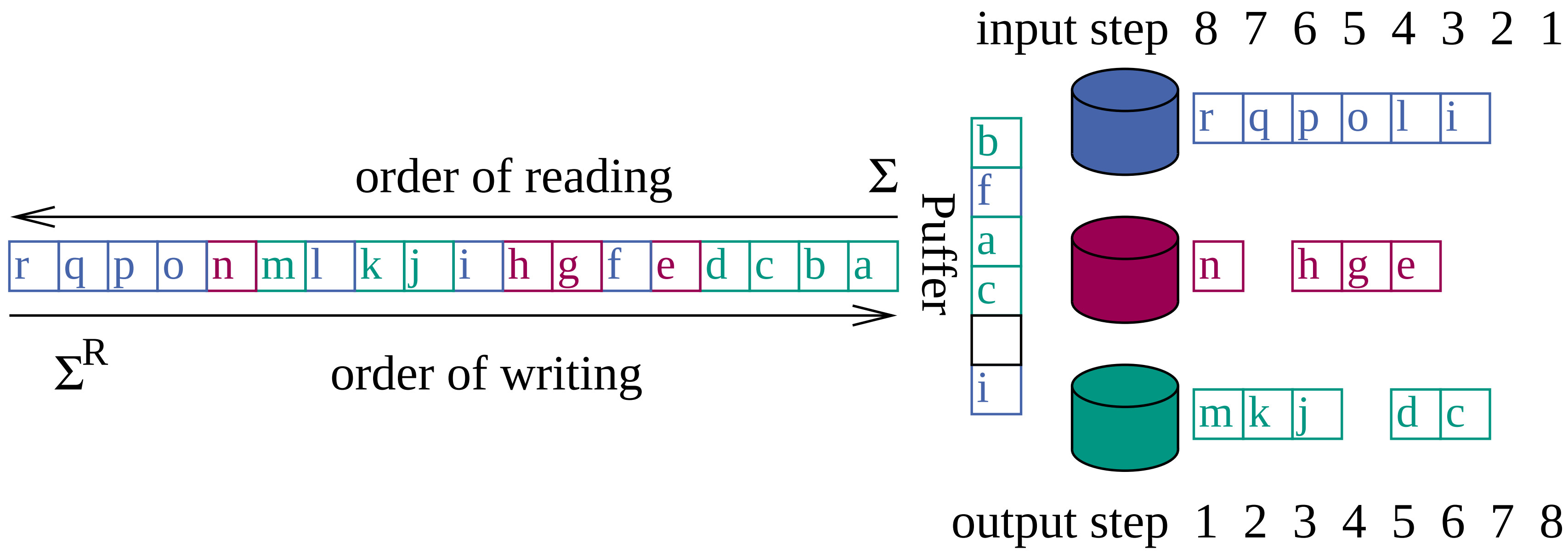
Theorem:

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

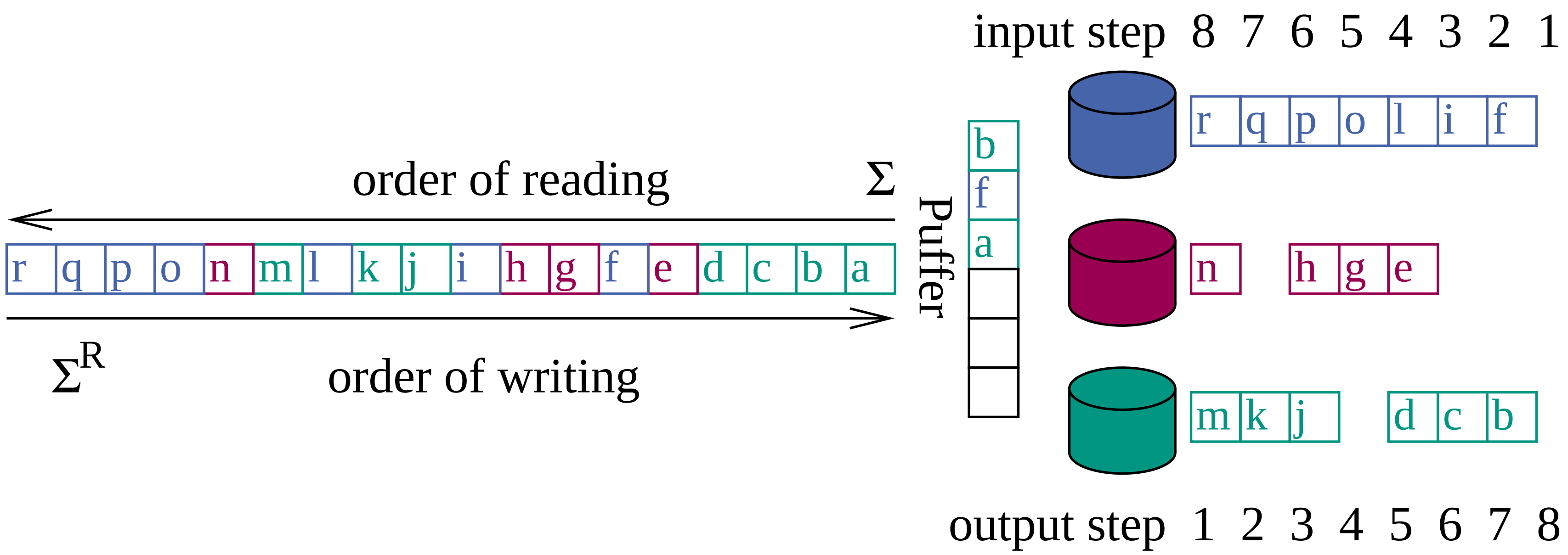
Theorem:

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

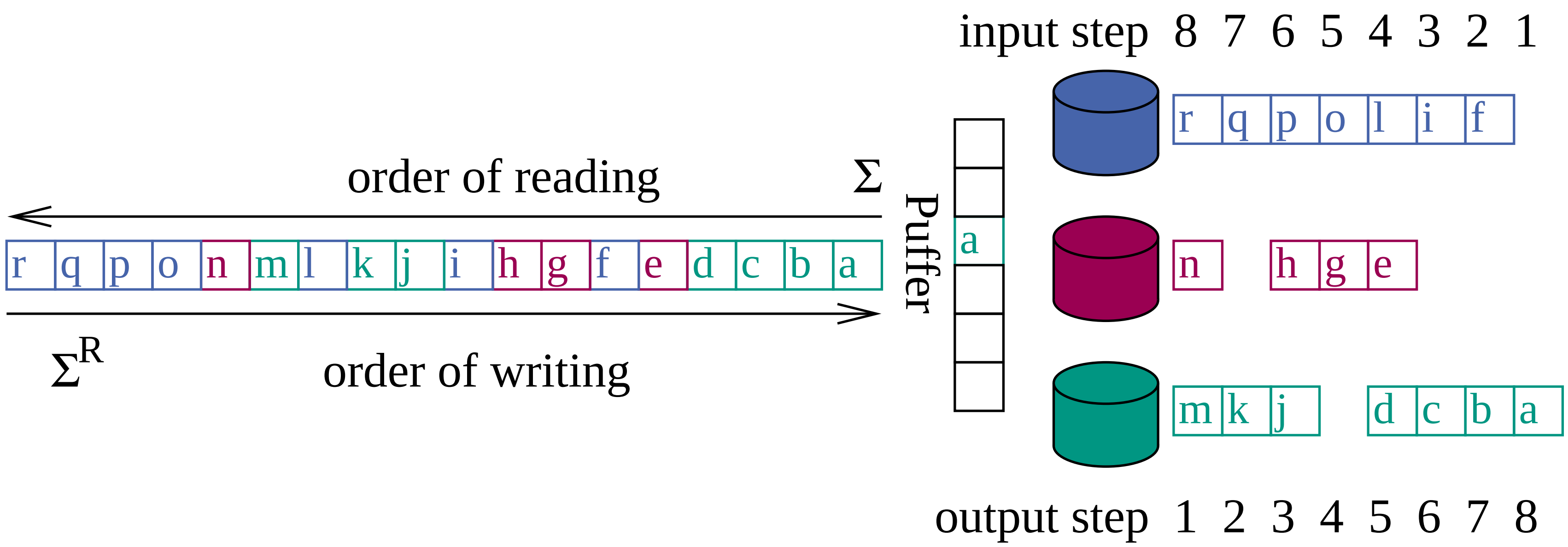
Theorem:

For buffer size W:

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

$\Leftrightarrow$

$\exists$ (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]

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

$\rightsquigarrow$"answers" question in [Knuth 98];

difficulty 48 on a 1..50 scale.

We are not done yet!

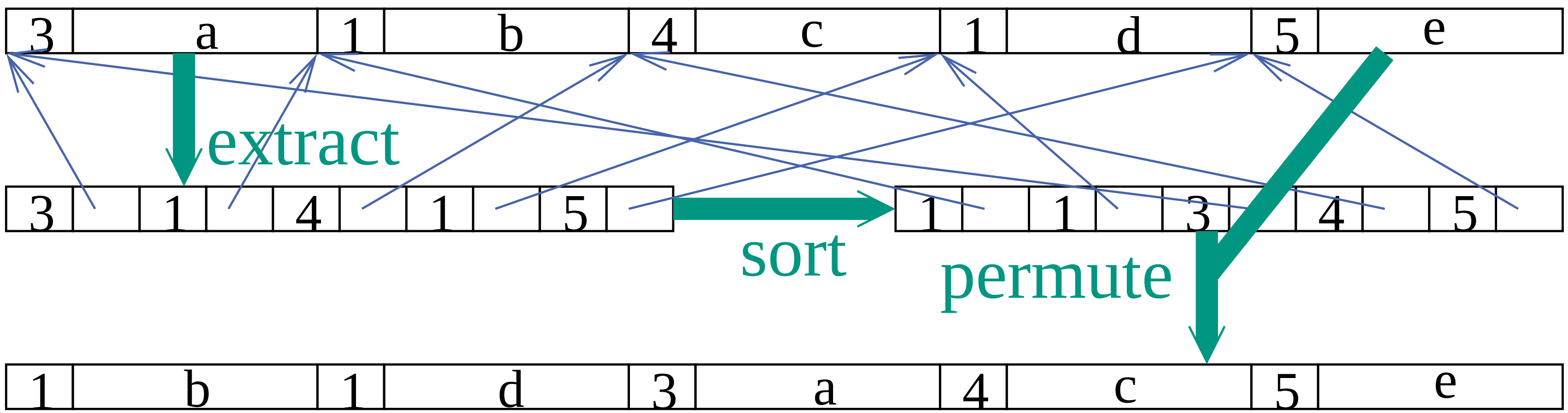
- Internal work
- Overlapping I/O and computation
- Reasonable hardware
- Interfacing with the Operating System
- Parameter Tuning
- Software engineering
- Pipelining

Key Sorting

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

$\rightsquigarrow$ If key size $\ll$ element size

sort key pointer pairs to save memory bandwidth during run formation



Tournament Trees for Multiway Merging

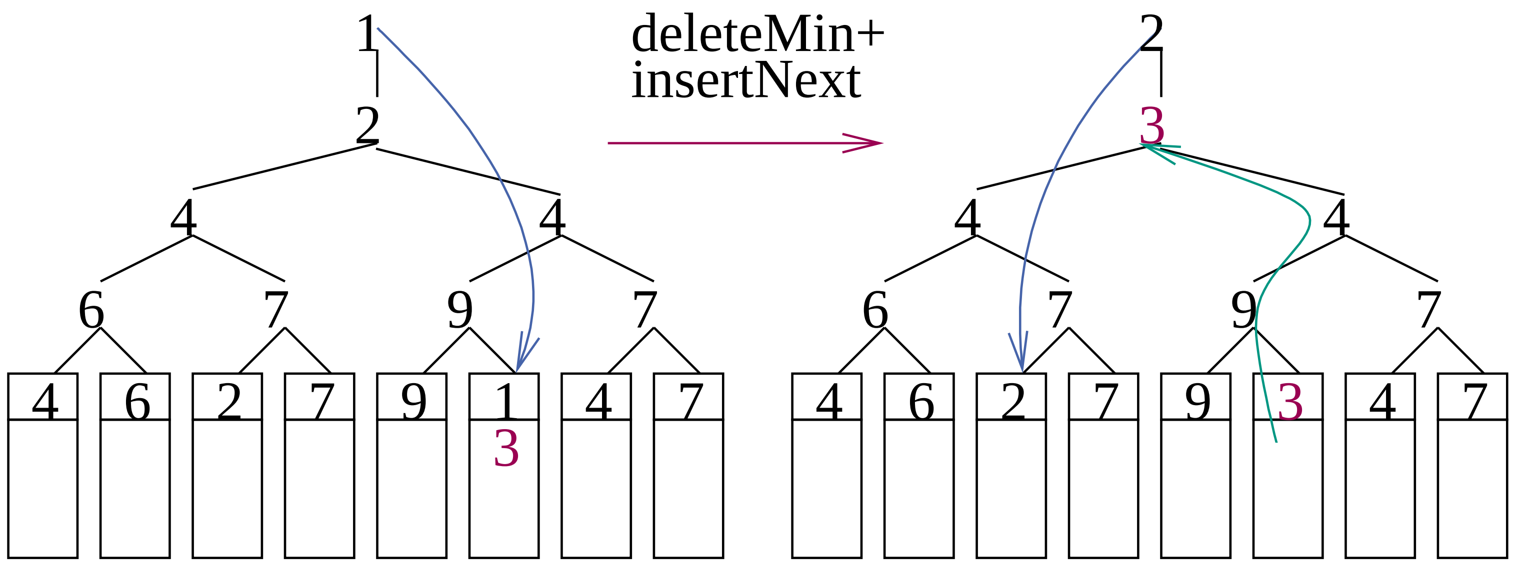
Assume $k = 2^K$ runs

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 $\leadsto$ 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

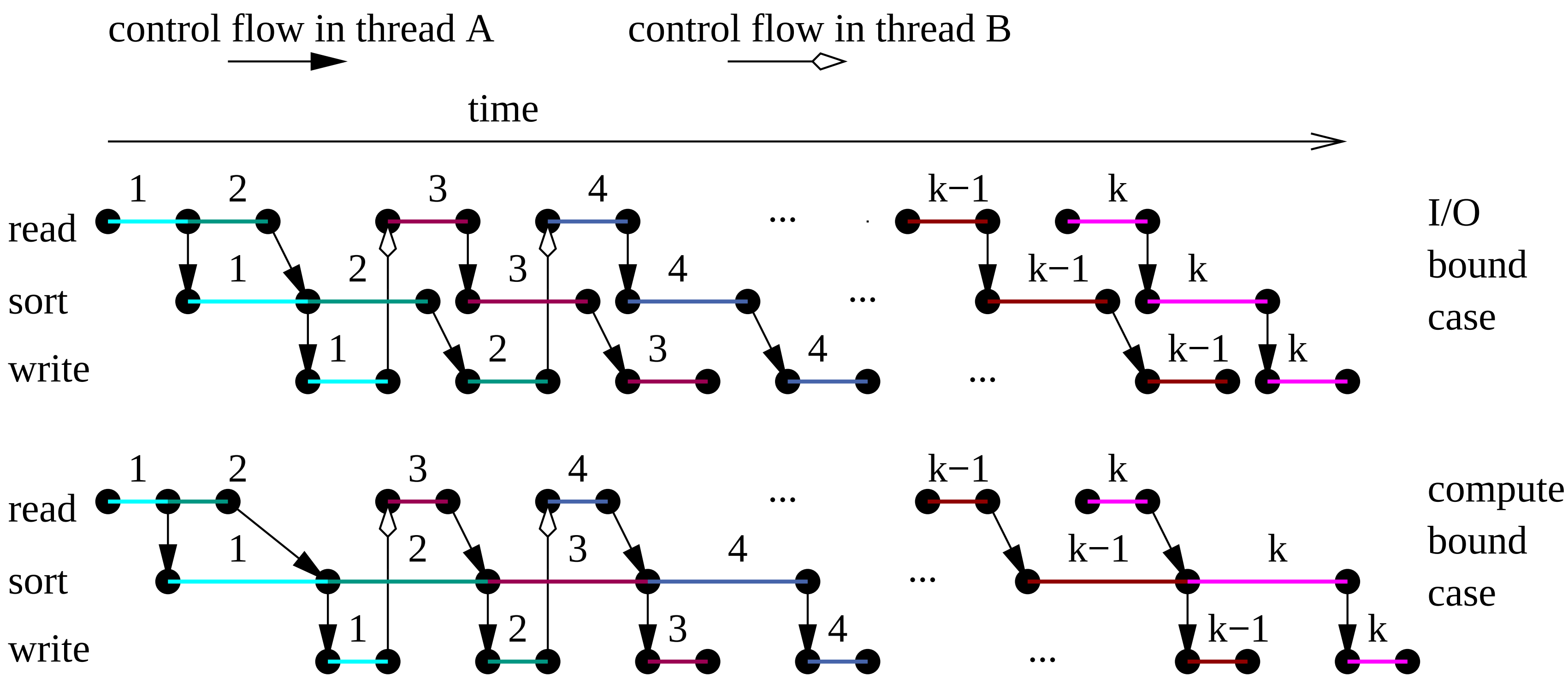
- One thread for each disk (or asynchronous I/O)
- Possibly additional threads
- Blocks filled with elements are passed by references between different buffers

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

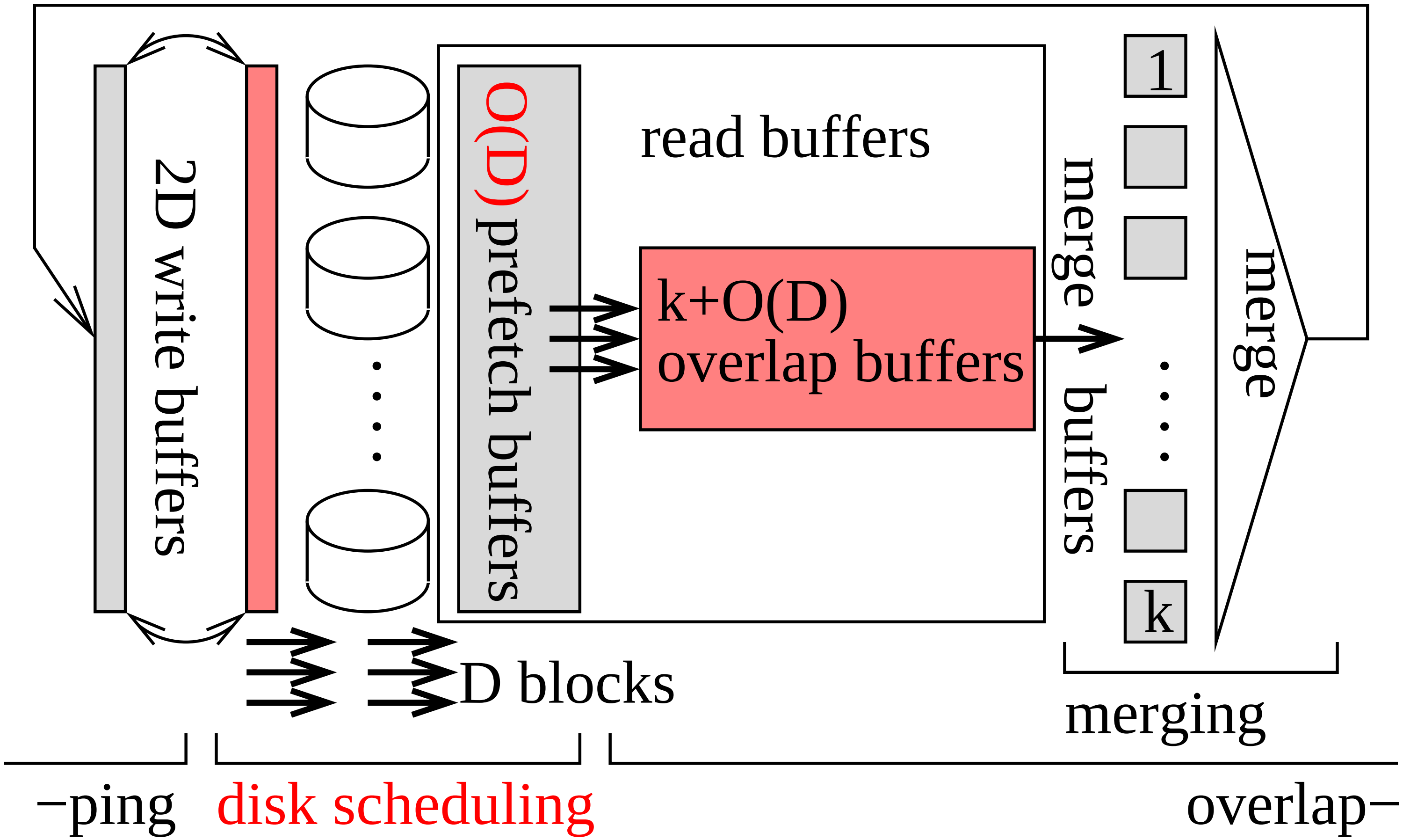
Bad example:

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

$$\cdots$$

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

Overlapping During Merging

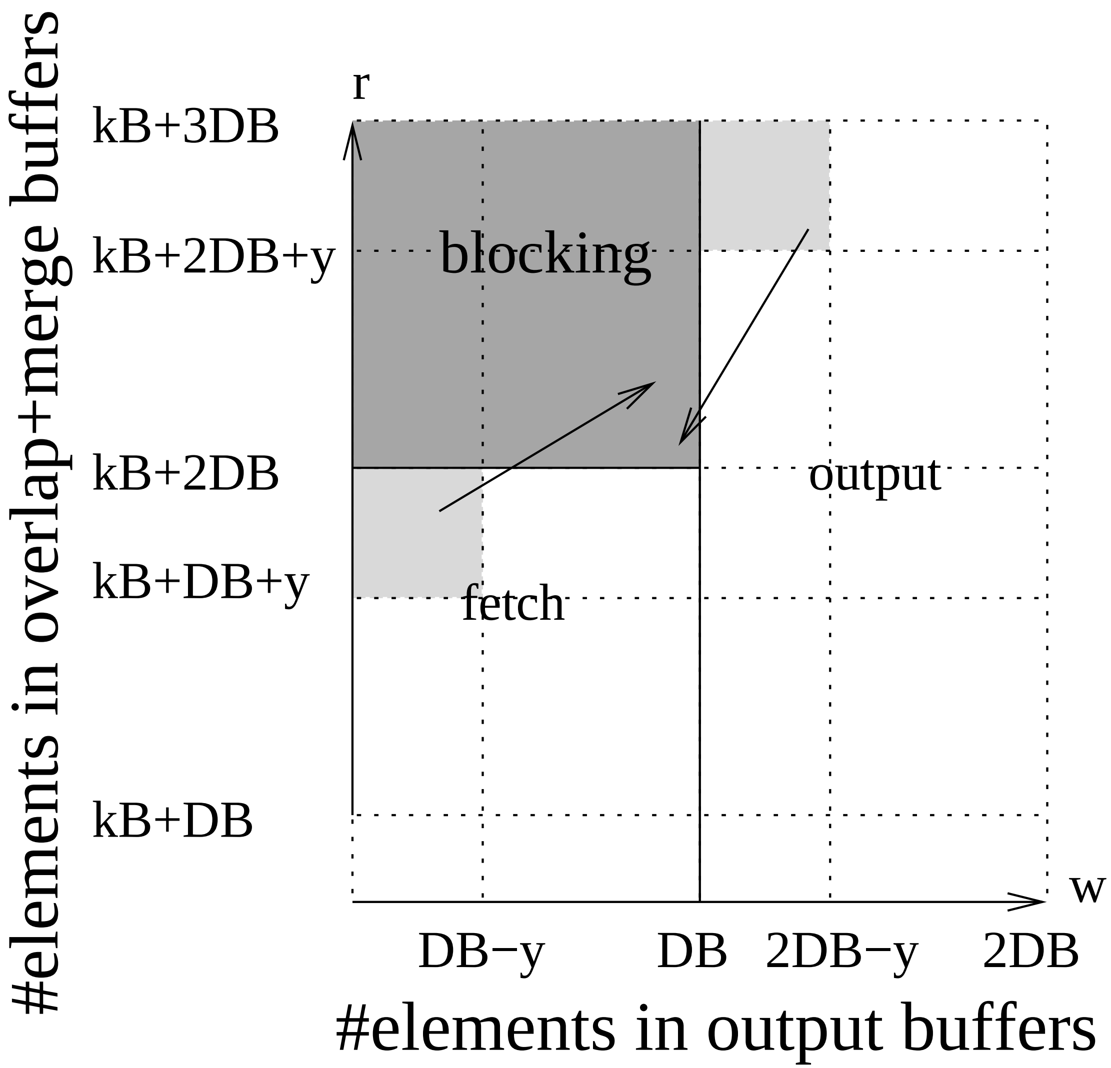


I/O Threads: Writing has priority over reading

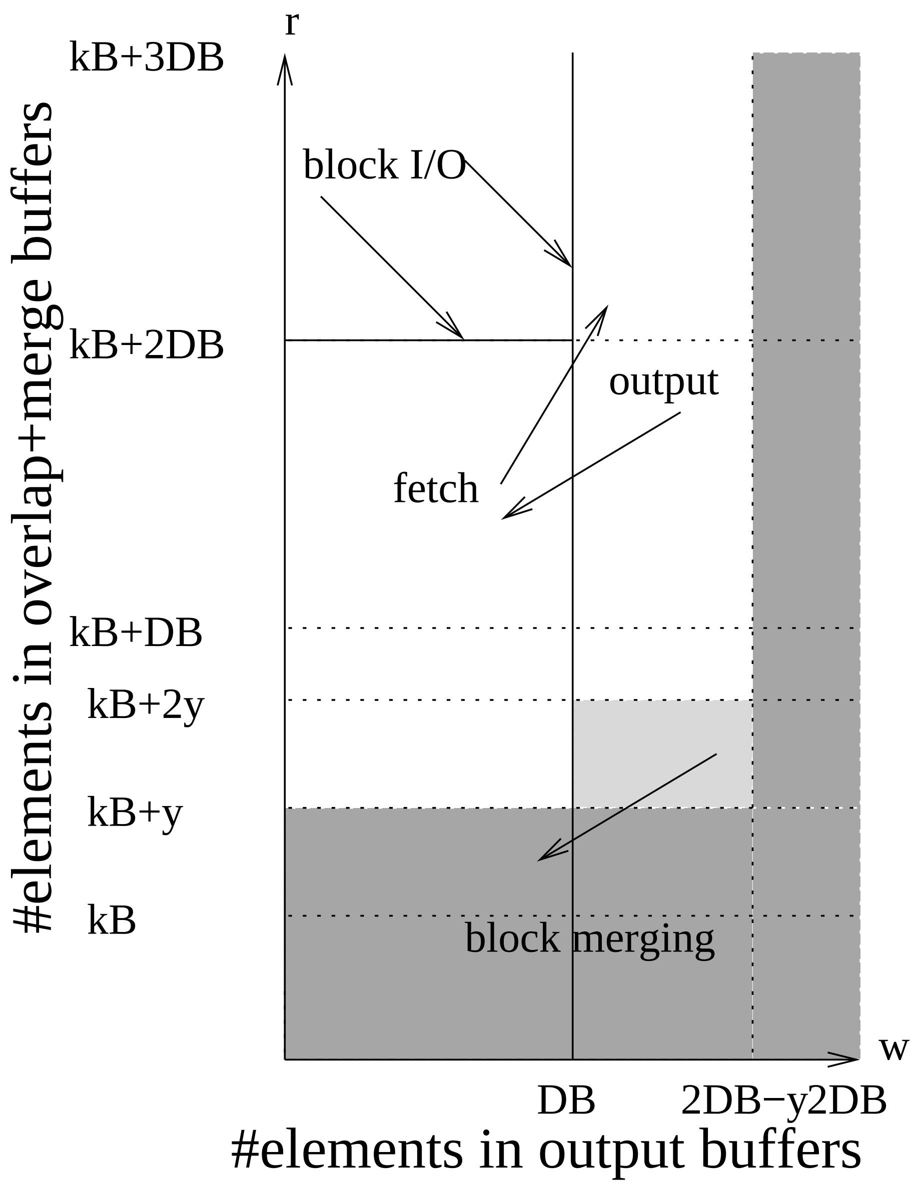
I/O bound case: prefetch thread never blocks

$y = \#$ 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

($2 \times$ 2GHz Xeon $\times$ 2 Threads)

Several 66 MHz PCI-buses

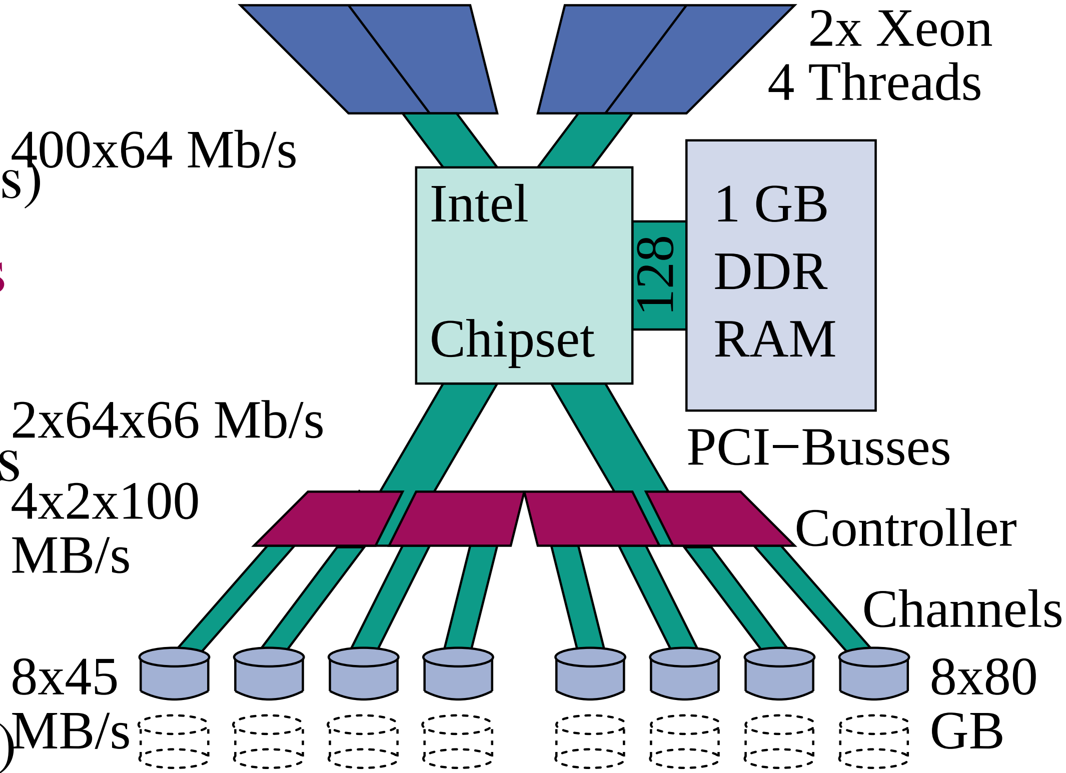
(SuperMicro P4DPE3)

Several fast IDE controllers

($4 \times$ Promise Ultra100 TX2)

Many fast IDE disks

($8\times$ IBM IC35L080AVVA07)



---

cost effective I/O-bandwidth (real 360 MB/s for $\approx$ 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

$\approx$3000 Euro for 32 512 GB SATA SSDs a 93 Euro:
$\leadsto$ 16TB capacity, and
$\leadsto$ 16GB/s read bandwidth ?

64 GB/RAM 800 Euro

500 Euro Motherboard

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

Hardware 2017 geschätzt

$\approx$2000 Euro for 4 1TB M.2 SSD a 500 Euro:

$\leadsto$ 4TB capacity, and

$\leadsto$ 14GB/s read bandwidth ?

128 GB/RAM 1000 Euro

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

Hardware 2019 geschätzt

$\approx$1720 Euro for 8 2TB M.2 SSD a 215 Euro:

$\leadsto$ 16TB capacity, and

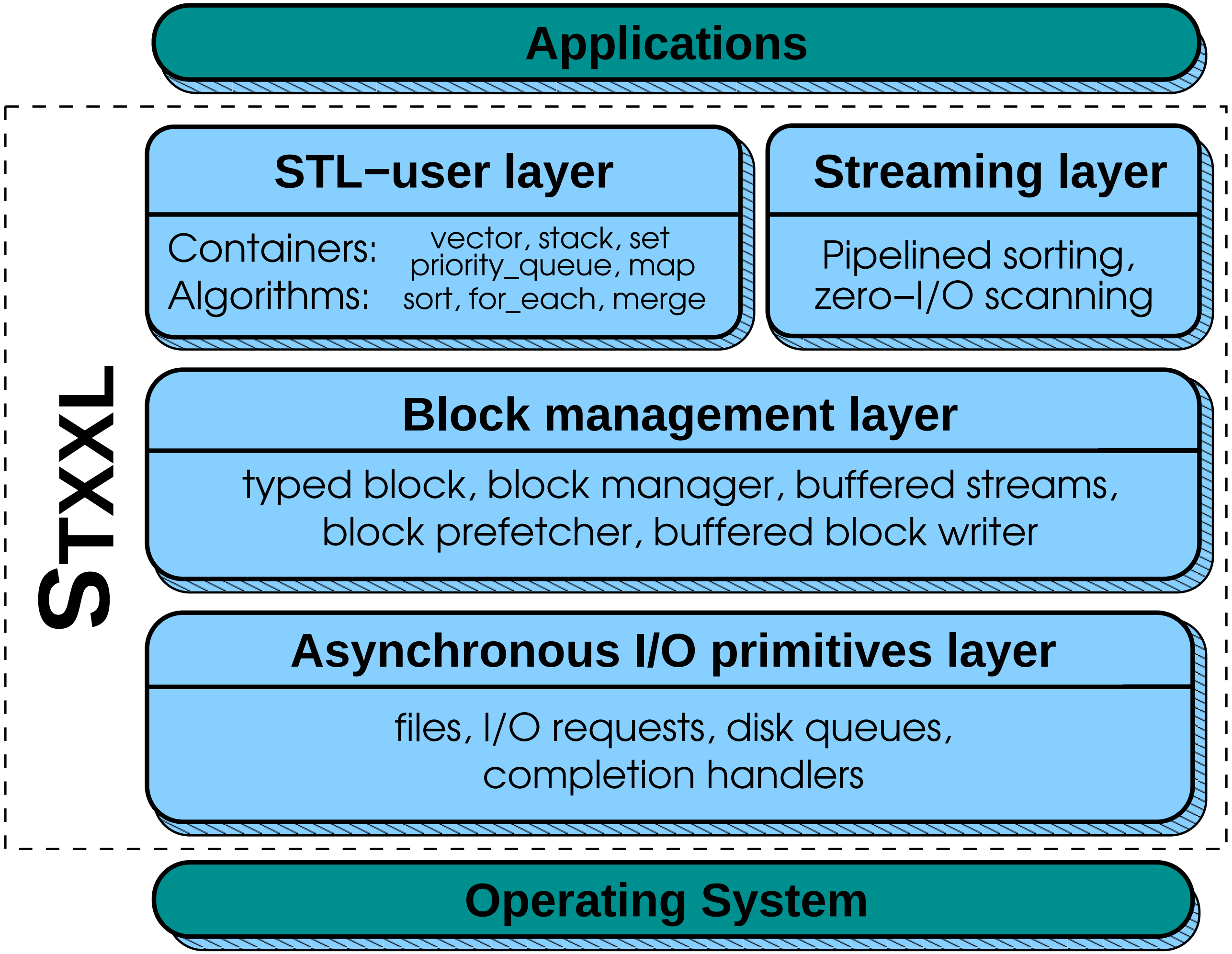
$\leadsto$ 14.4GB/s read bandwidth ?

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

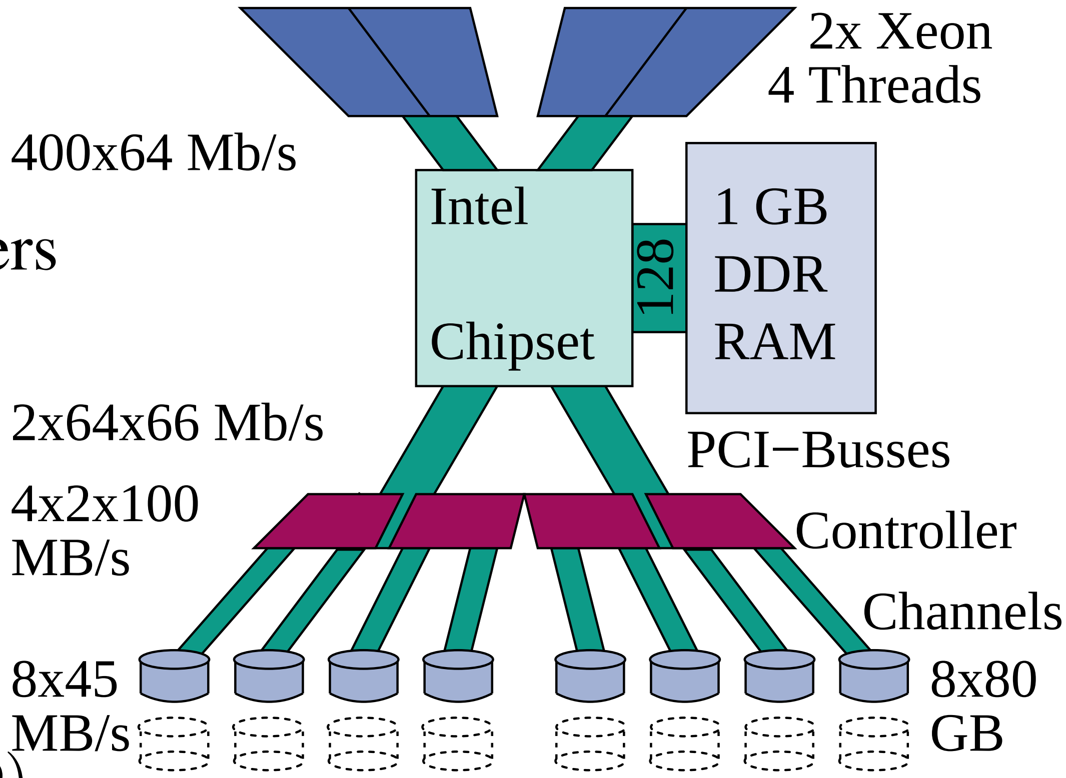
Run Size: 256 MByte

Block size B: 2 MByte

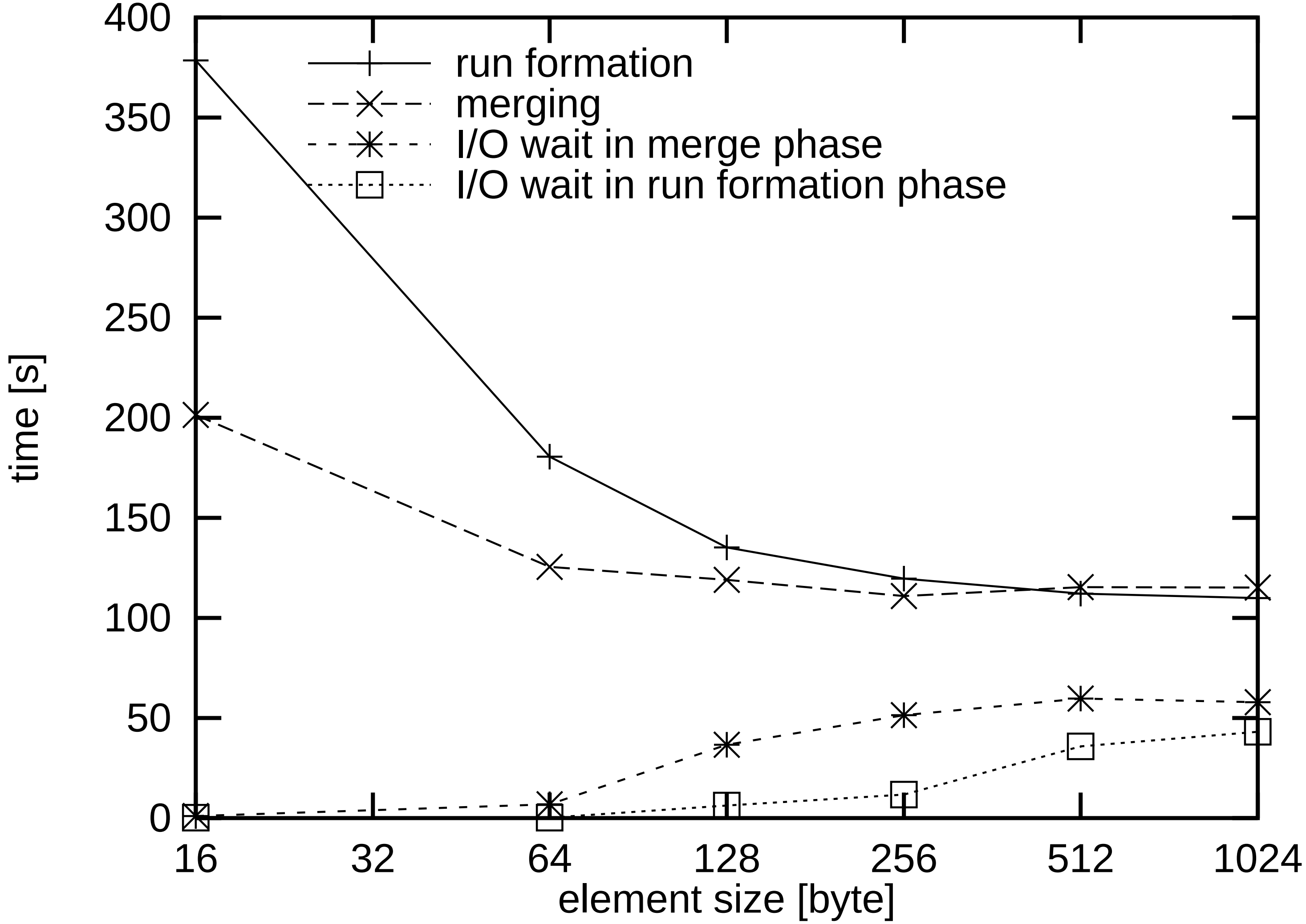
Compiler: g++ 3.2 -O6

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

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



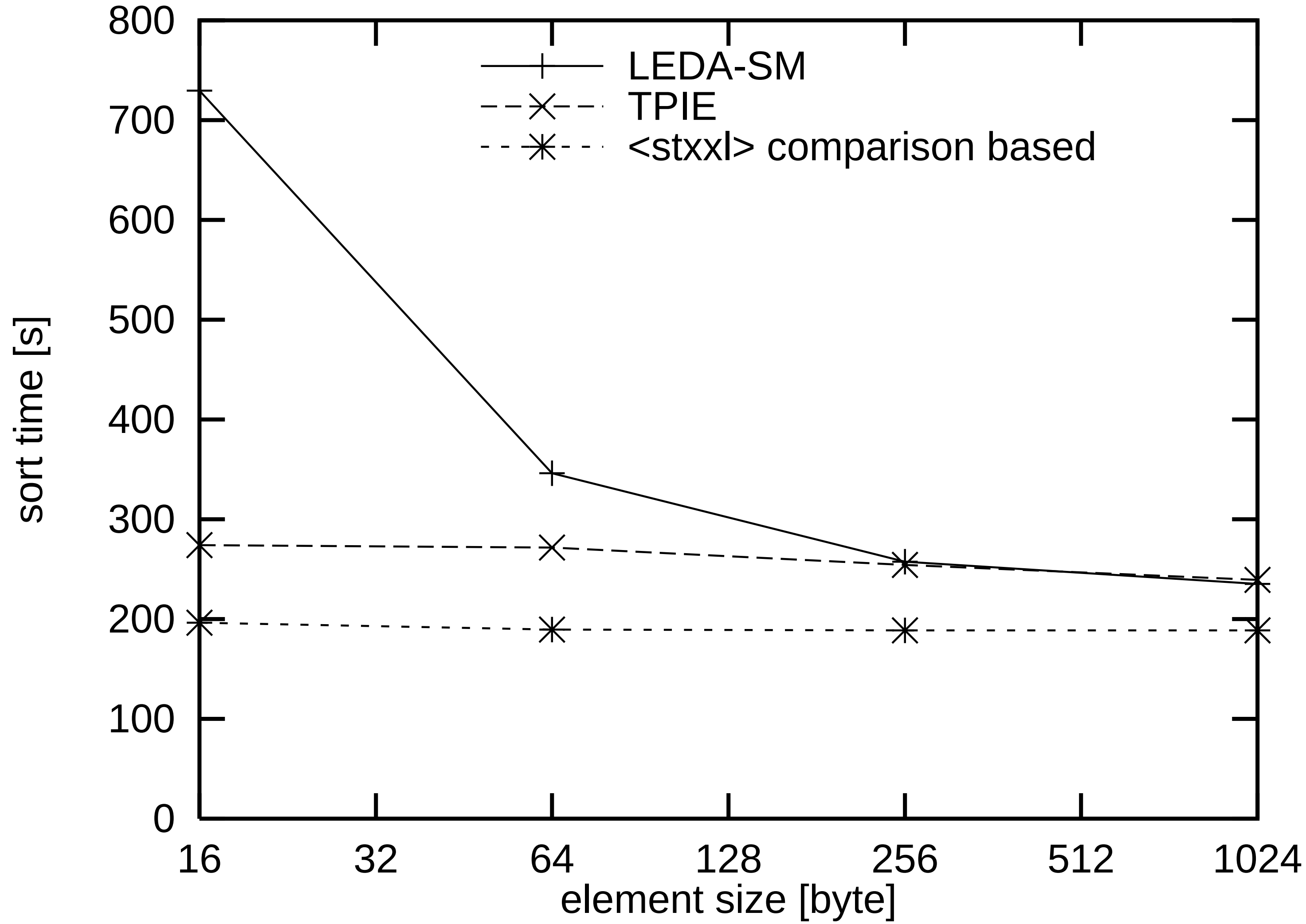
Element sizes (16 GByte, 8 disks)



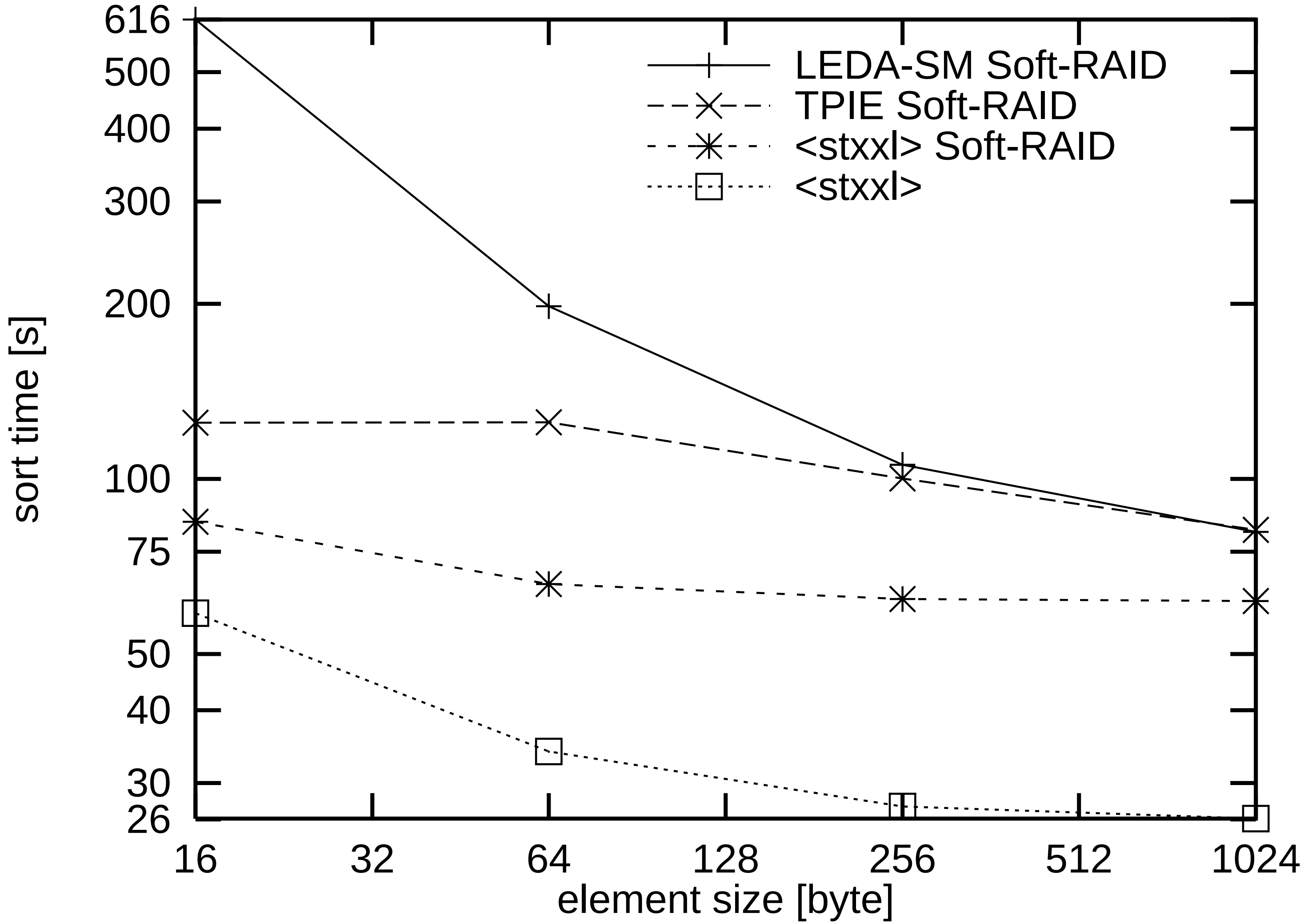
parallel disks ⇝ bandwidth "for free" ⇝ internal work, overlapping are relev

Earlier Academic Implementations

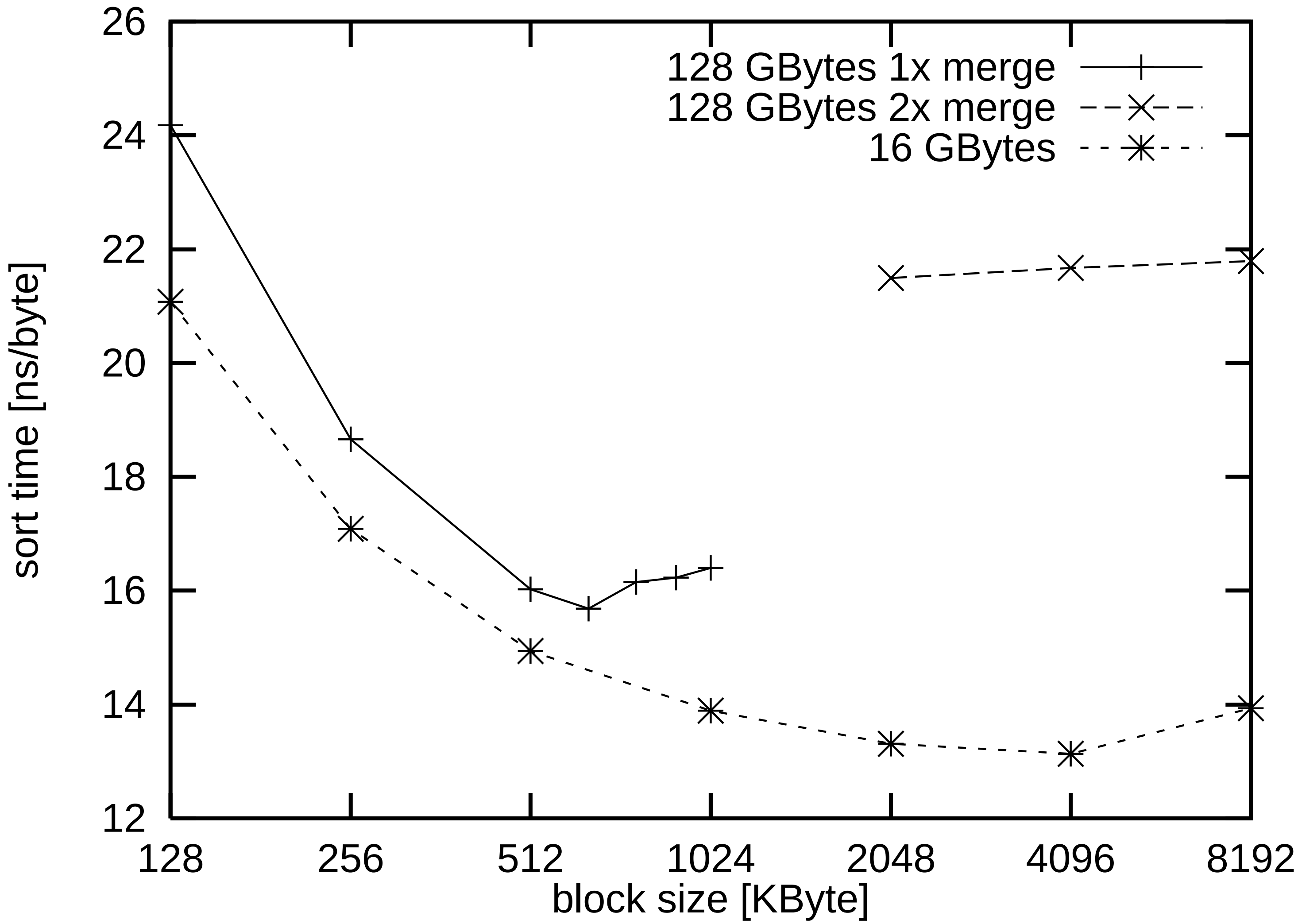
Single Disk, at most 2 GByte, old measurements use artificial *M*



Earlier Acad. Implementations: Multiple Disks



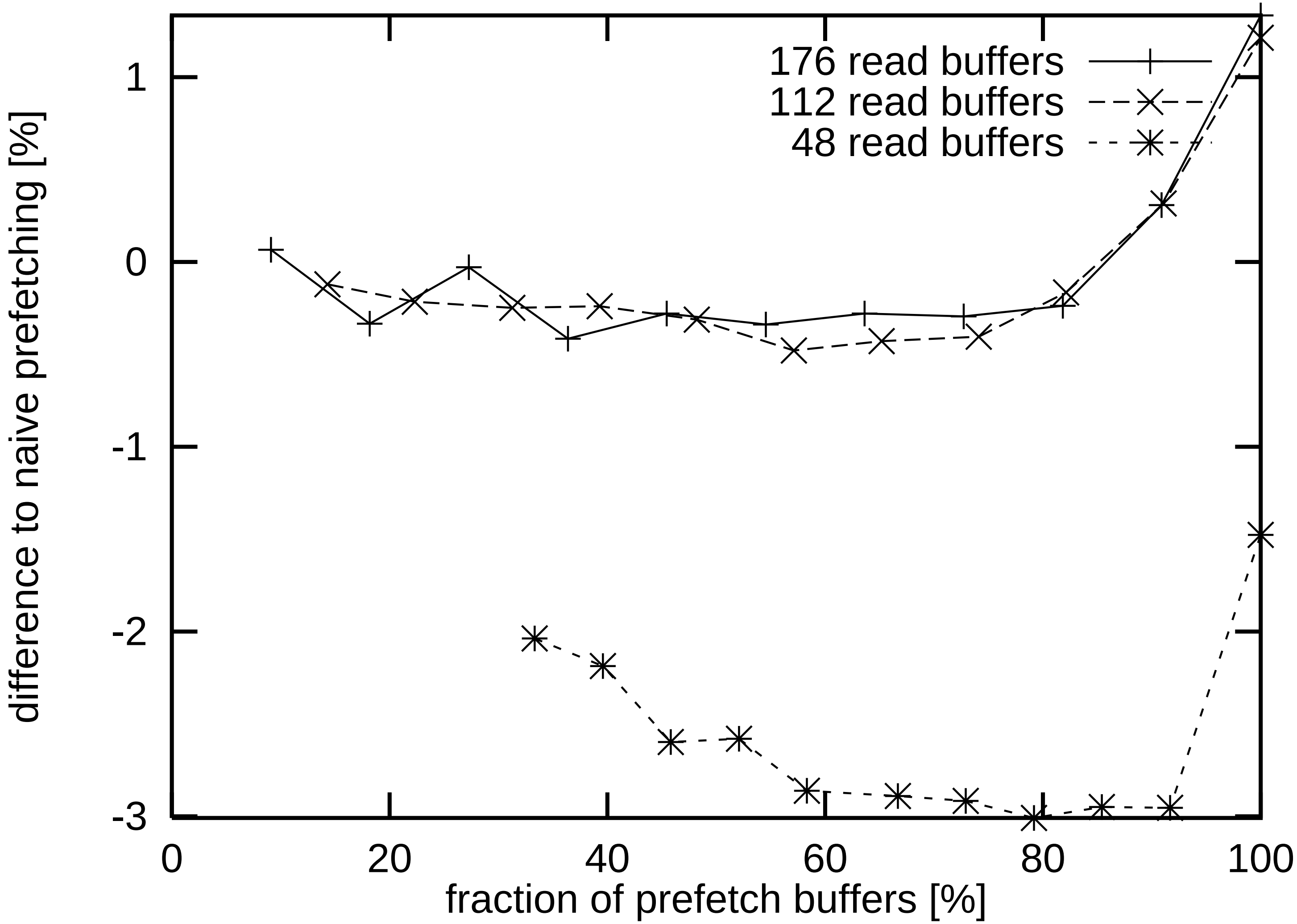
What are good block sizes (8 disks)?



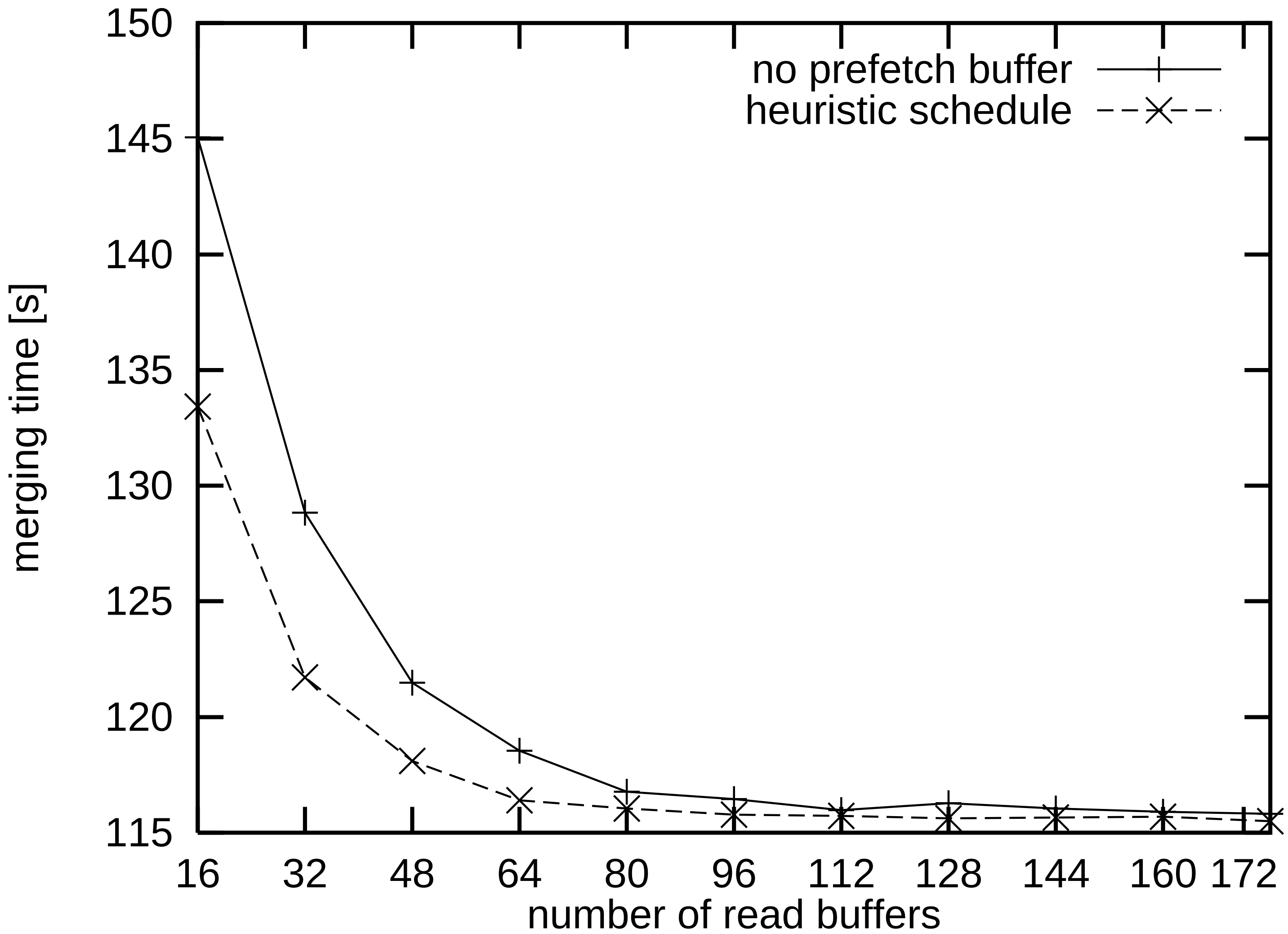
B is not a technology constant

Optimal Versus Naive Prefetching

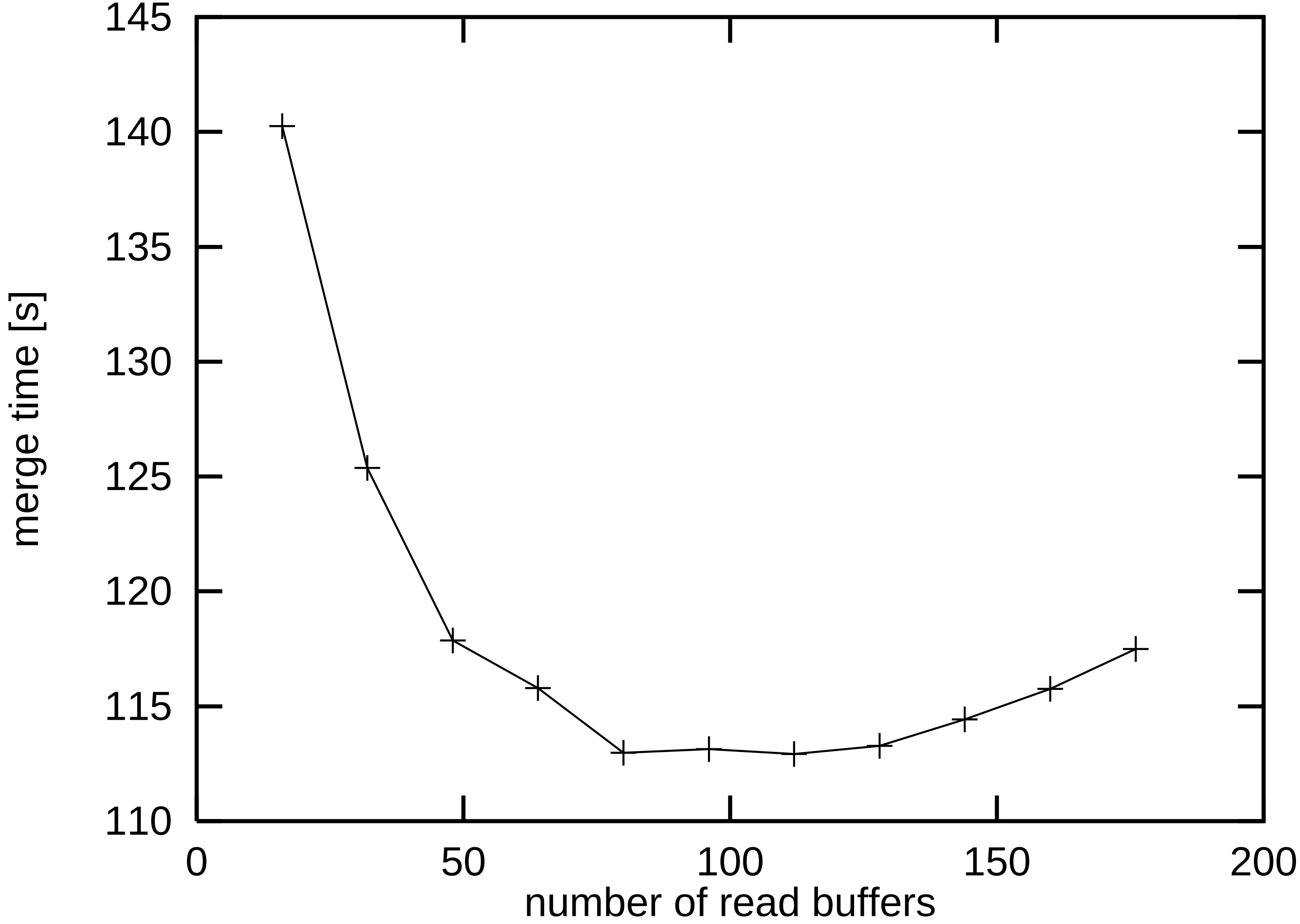
Total merge time



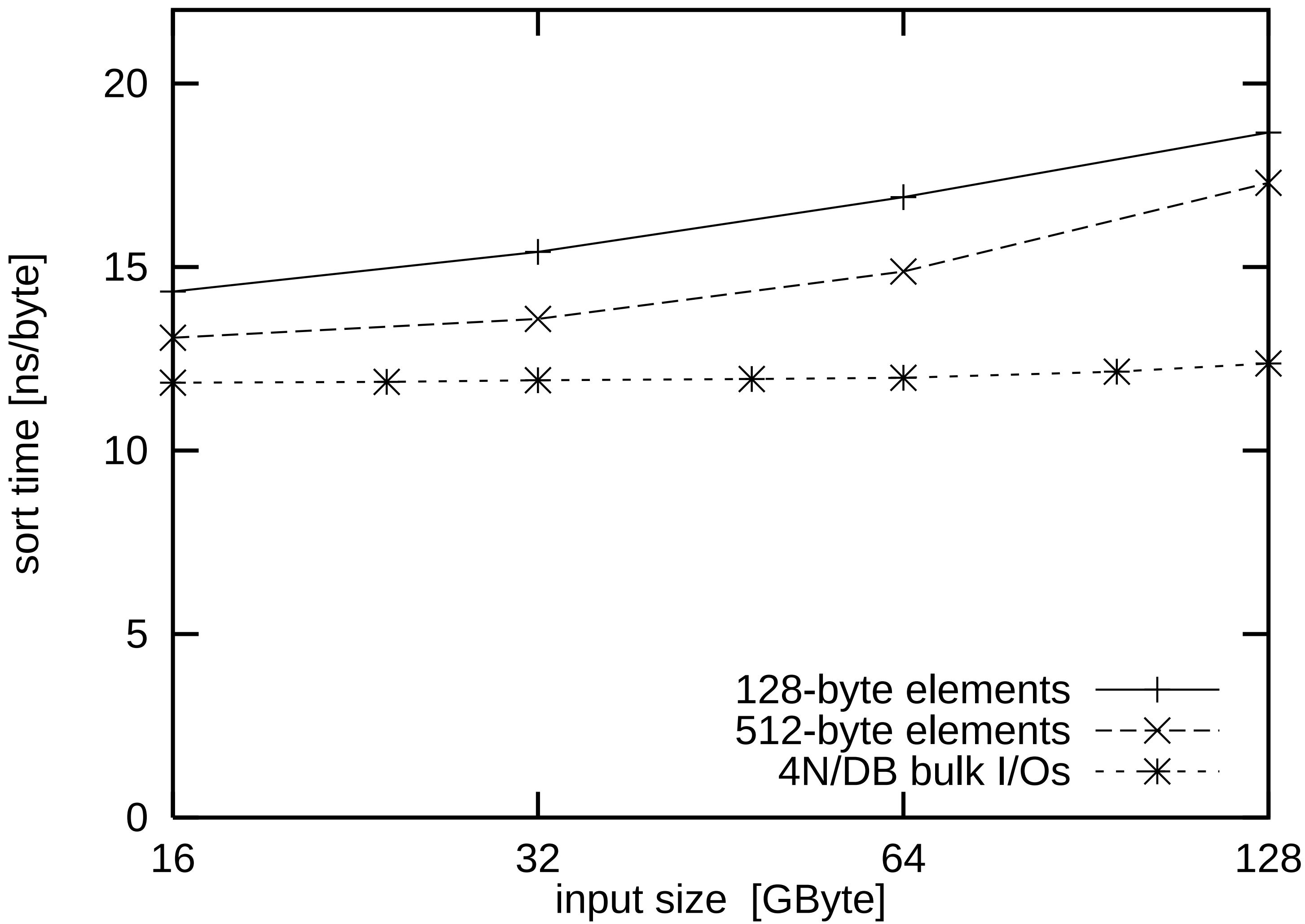
Impact of Prefetch and Overlap Buffers



Tradeoff: Write Buffer Size Versus Read Buffer Size



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

$\leadsto$ only half the I/Os for sorting

$\leadsto$ 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: ssssort, 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 $e' :=$ 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 $2M$

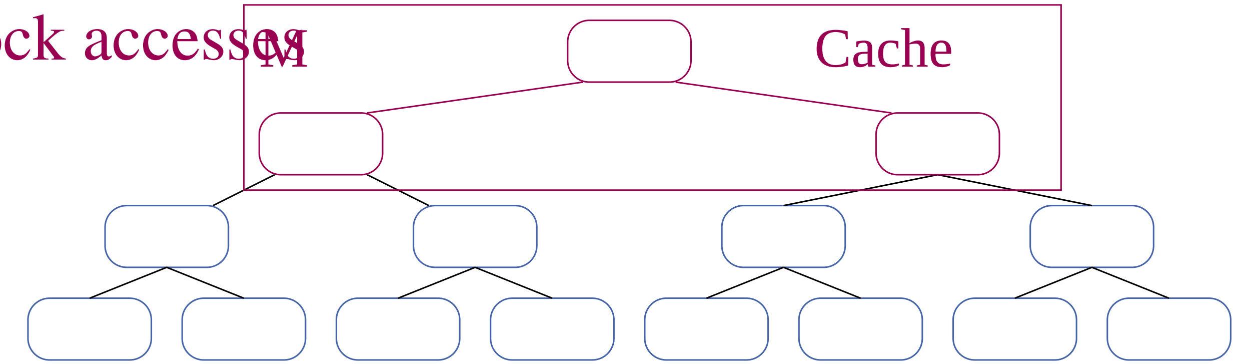
todo: cache-effiziente Implementierung

3 Priority Queues (insert, deleteMin)

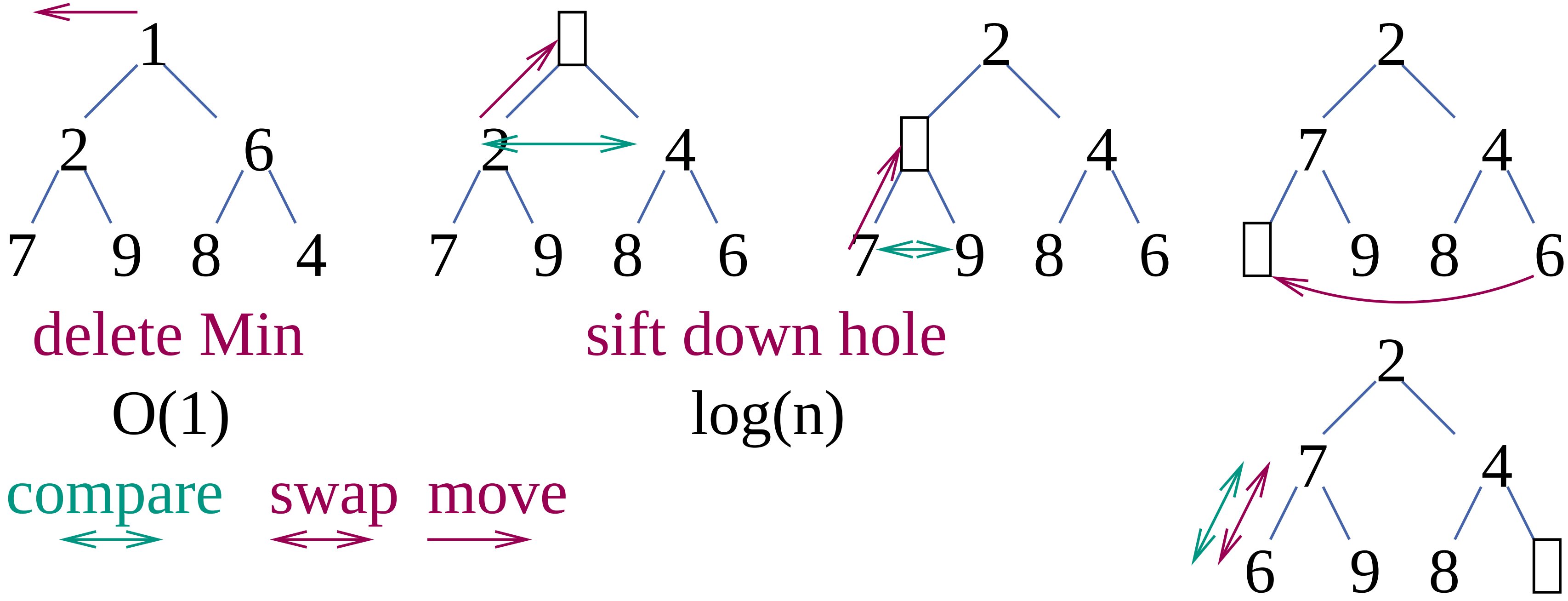
Binary Heaps best comparison based "flat memory" algorithm

+ 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

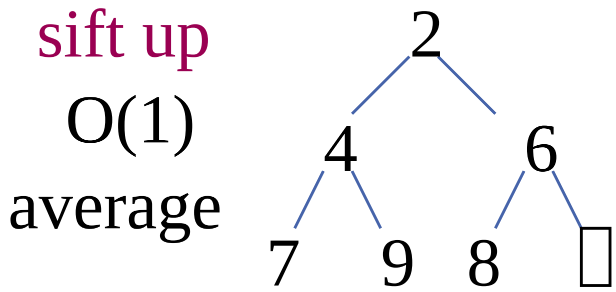


Bottom Up Heuristics



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

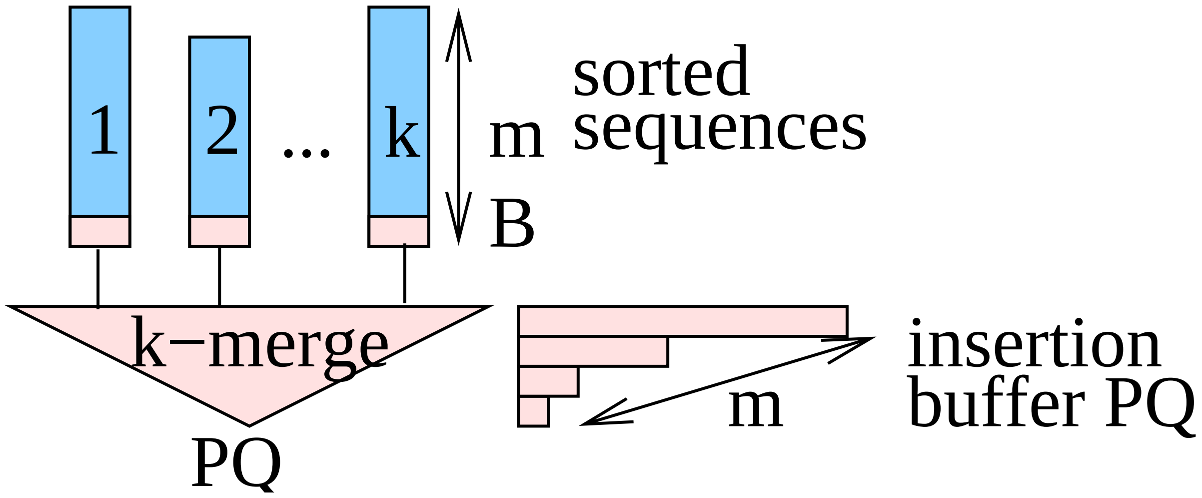
Procedure buildHeapBackwards
    for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** siftDown(i)

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.

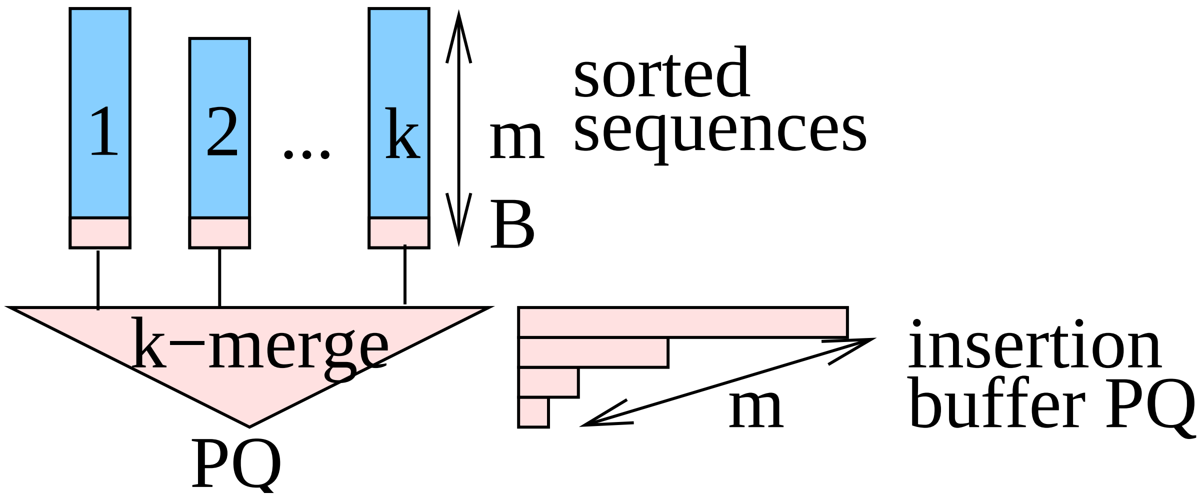
Überlauf $\longrightarrow$

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.



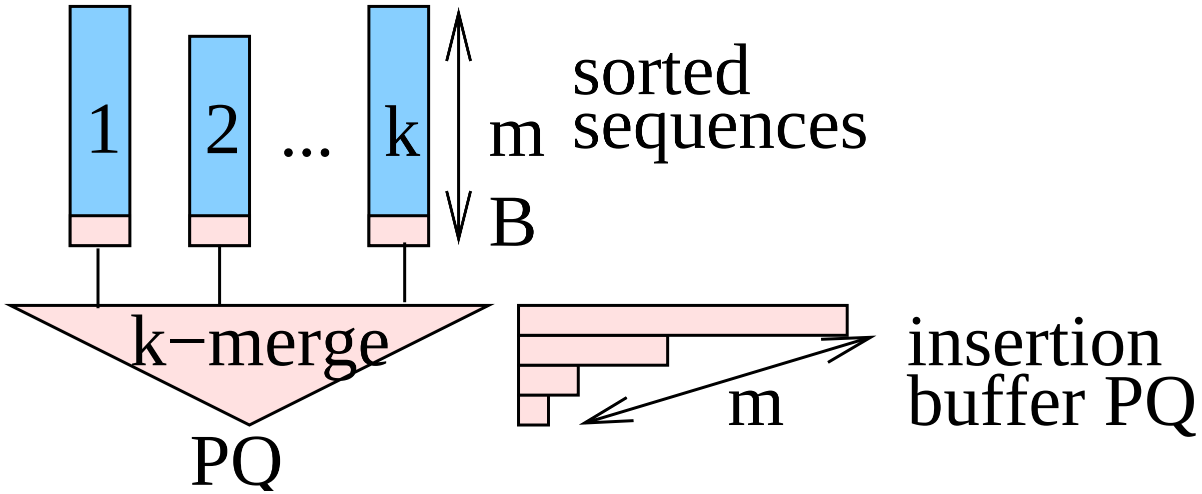
Analyse – Vergleiche (Maß für interne Arbeit)

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

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

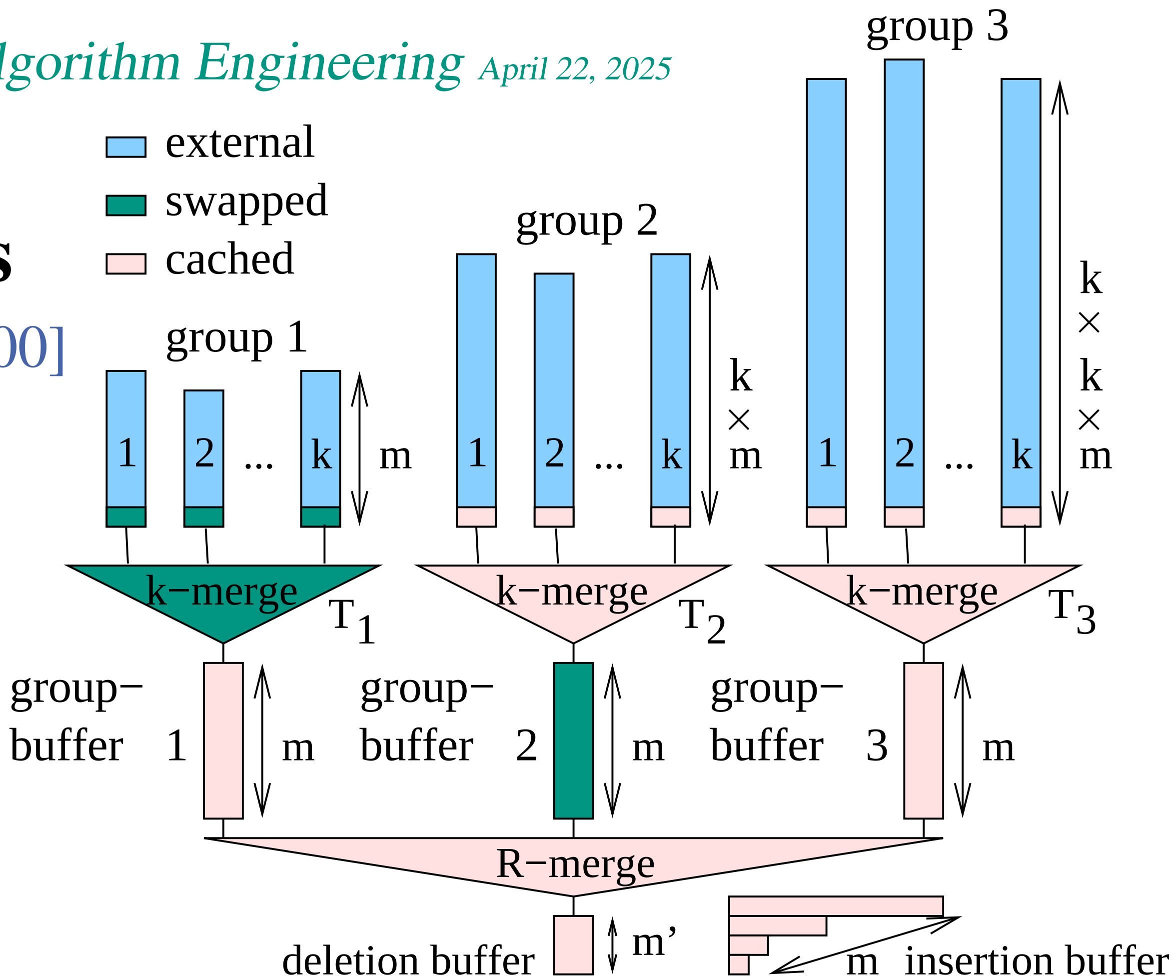
insert: $\approx m \log m$ alle m Ops. Amortisiert $\log m$

Insgesamt nur $\log km$ amortisiert !



Large Queues

[Sanders 00]



insert:

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

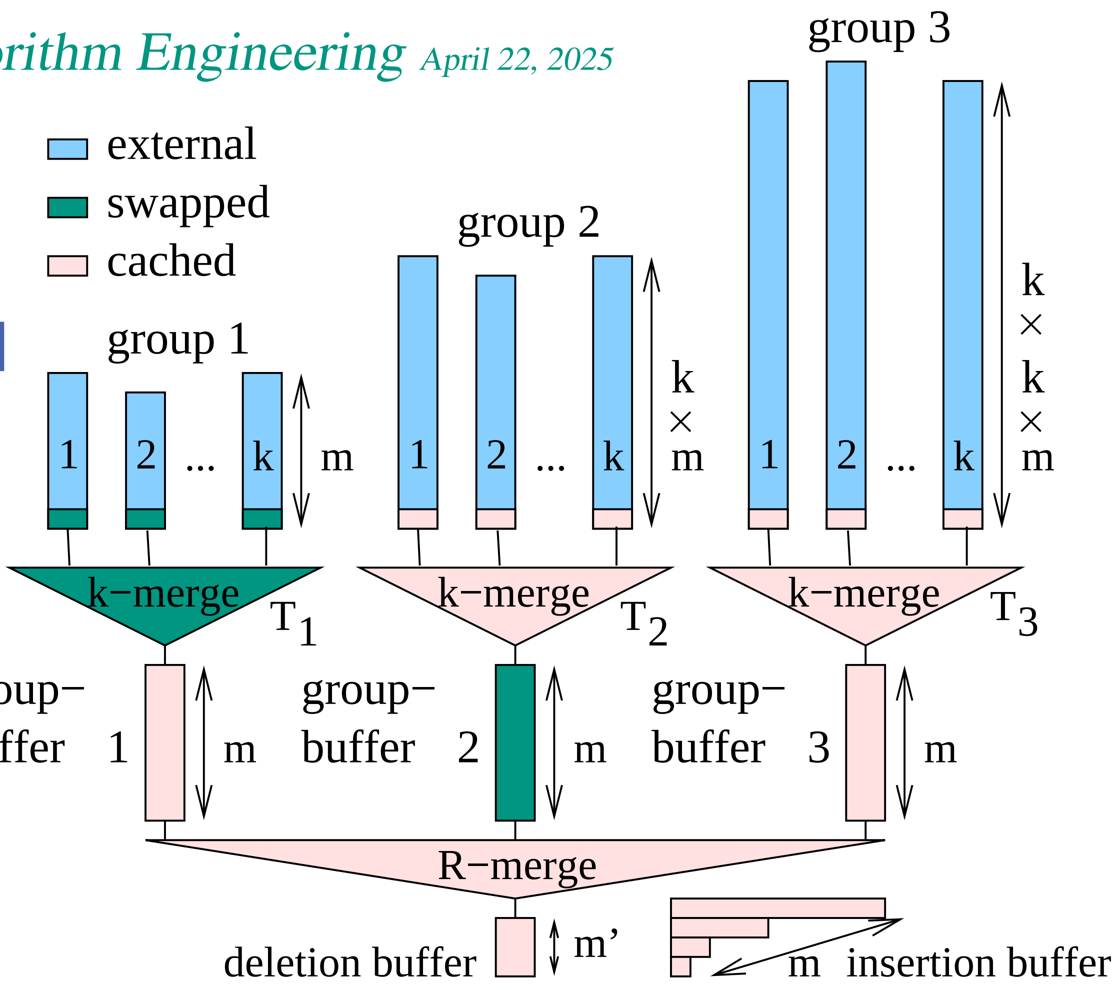
m' smallest into deletion buffer, next m into group buffer one, rest into group 1.

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

merge invalid group buffers and move them into group 1.

Large Queues

[Sanders 00]

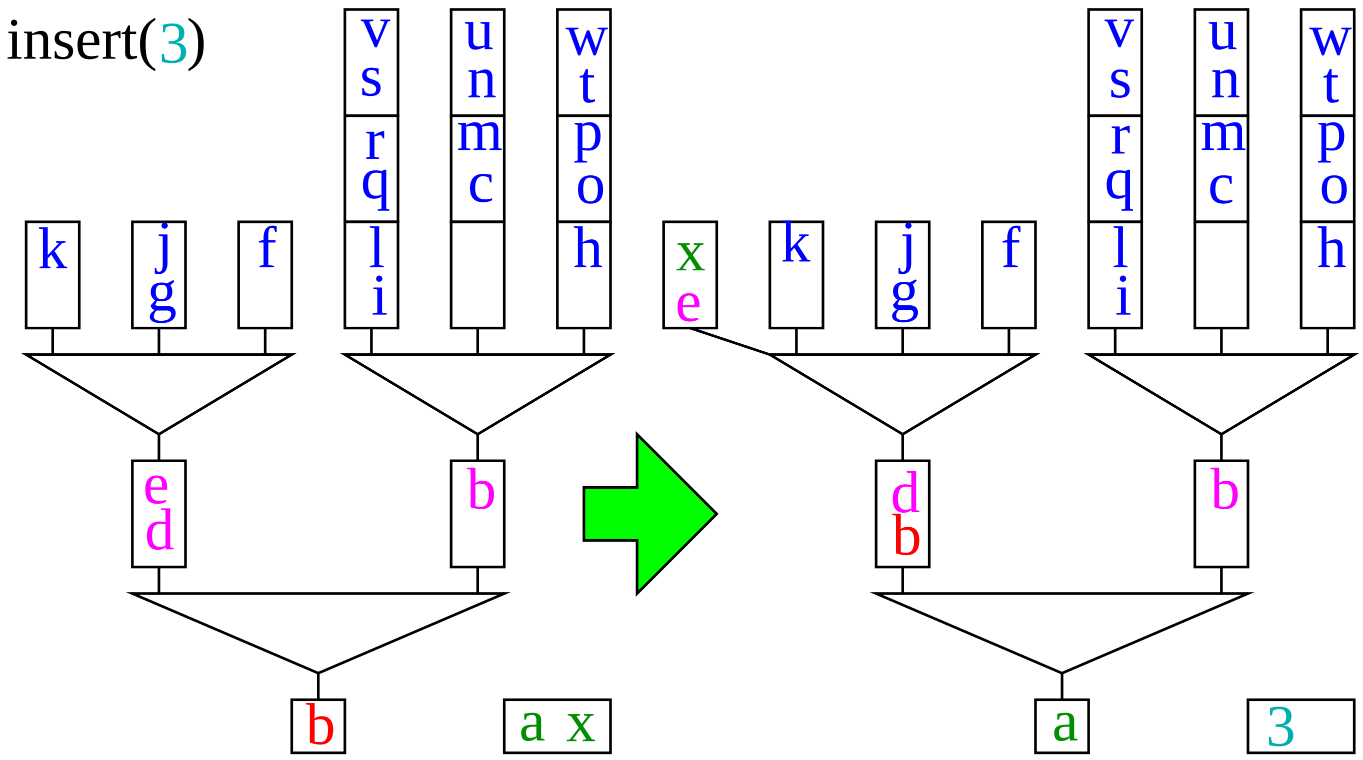


Delete-Min:

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

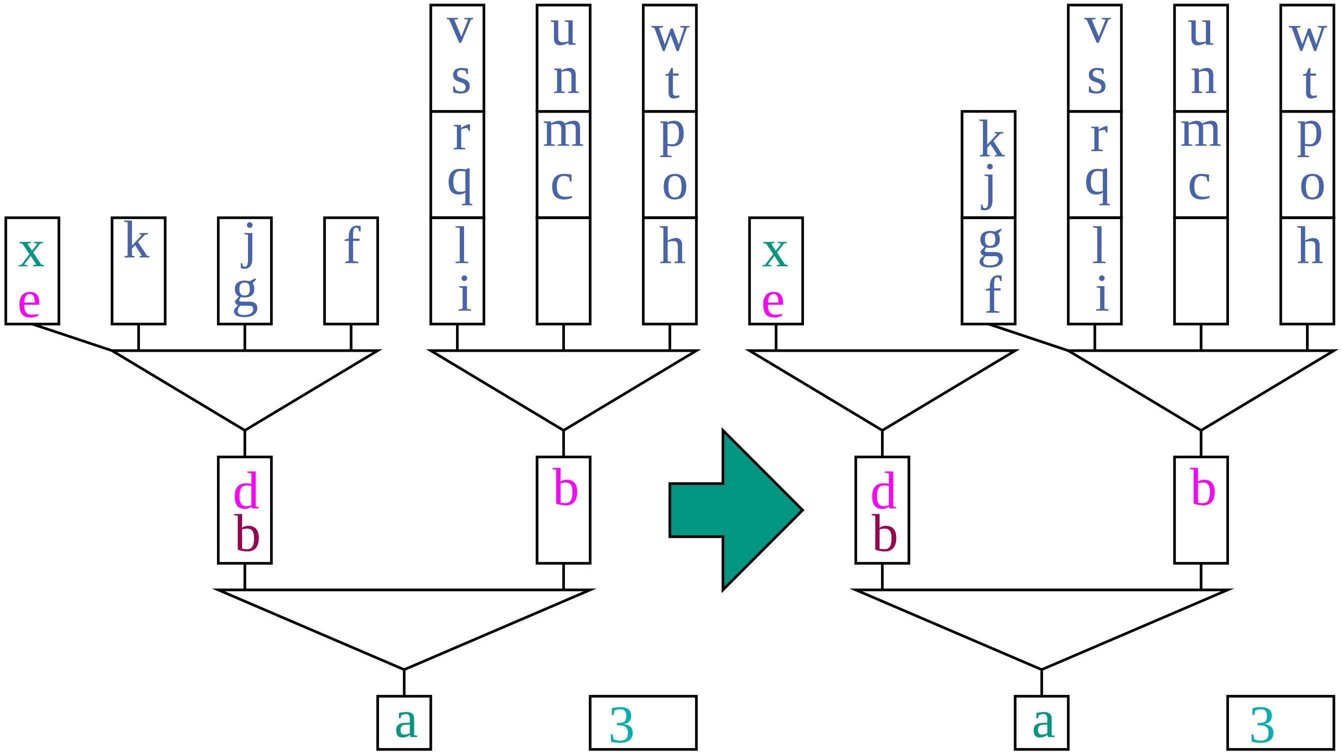
Example

Merge insertion buffer, deletion buffer, and leftmost group buffer



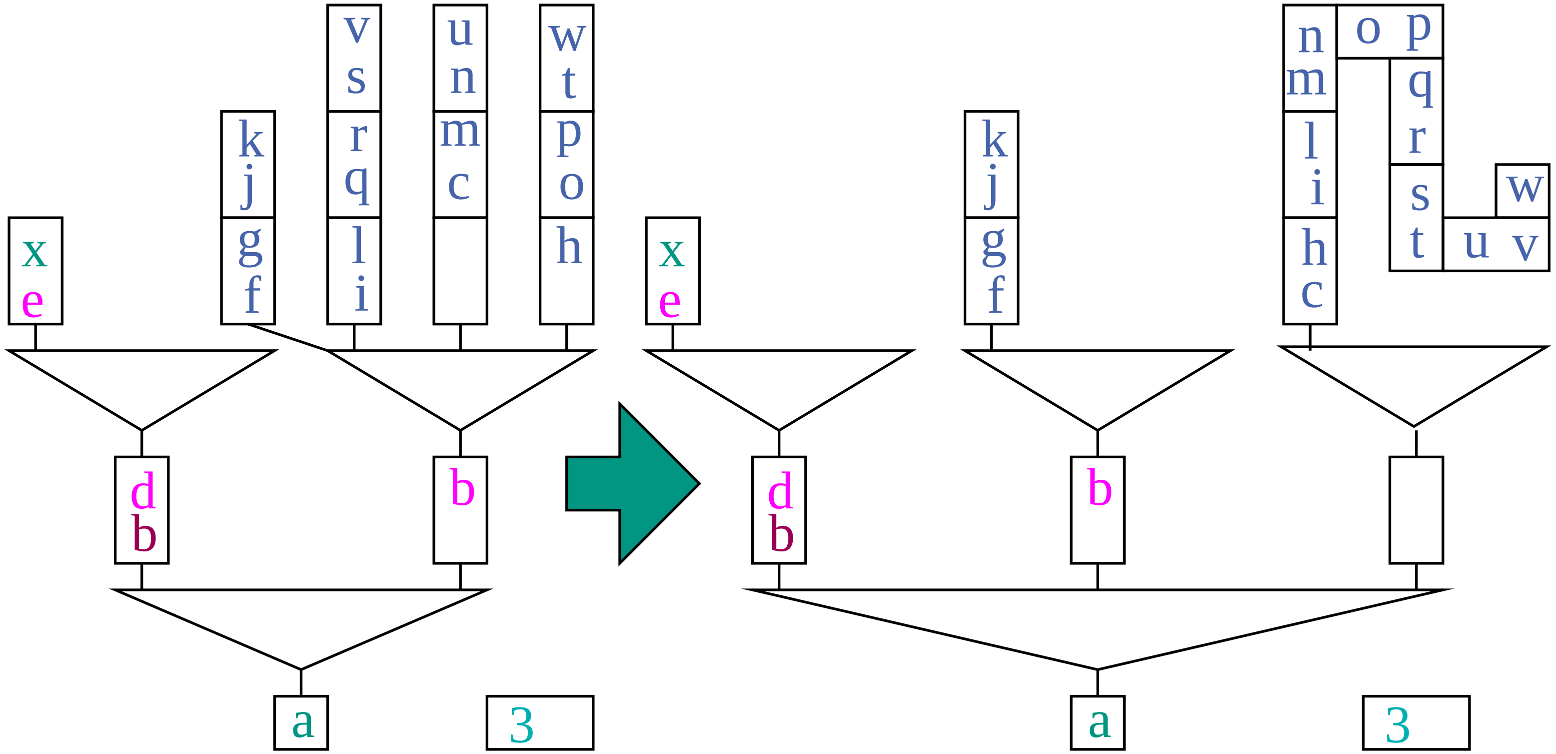
Example

Merge group 1



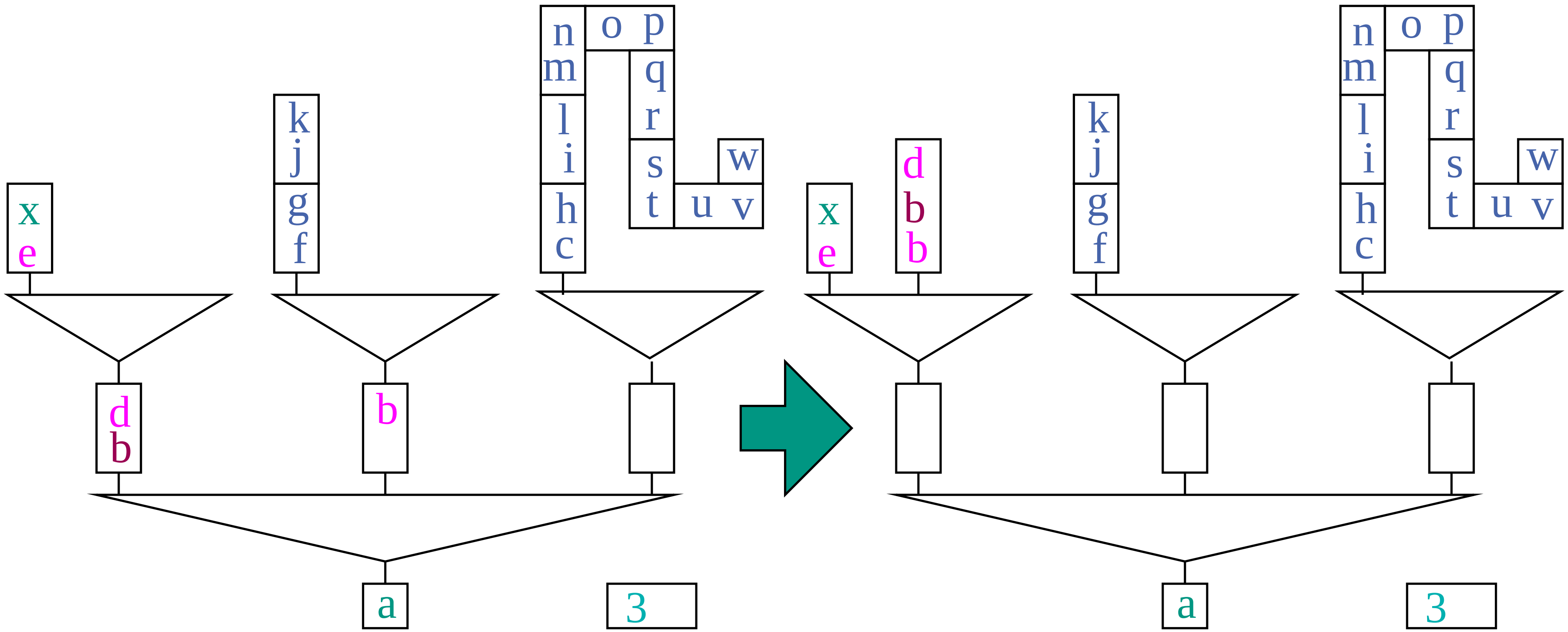
Example

Merge group 2



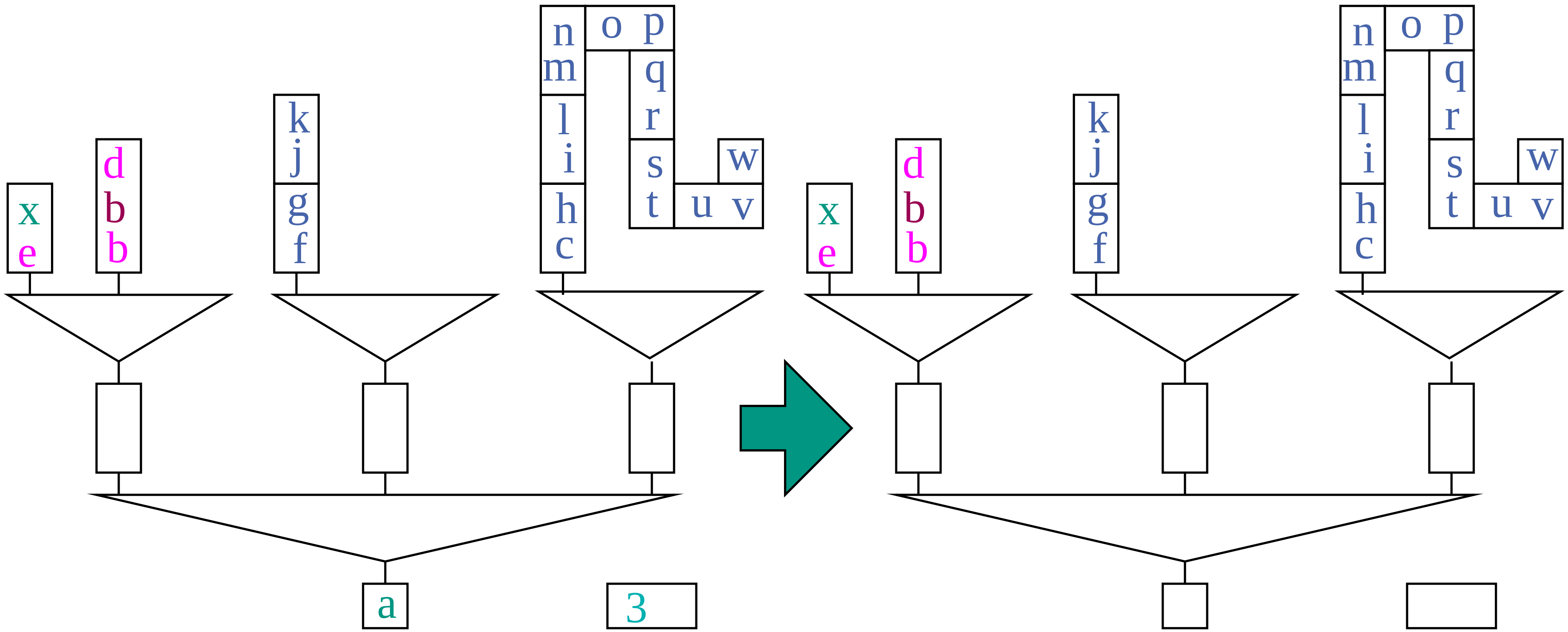
Example

Merge group buffers



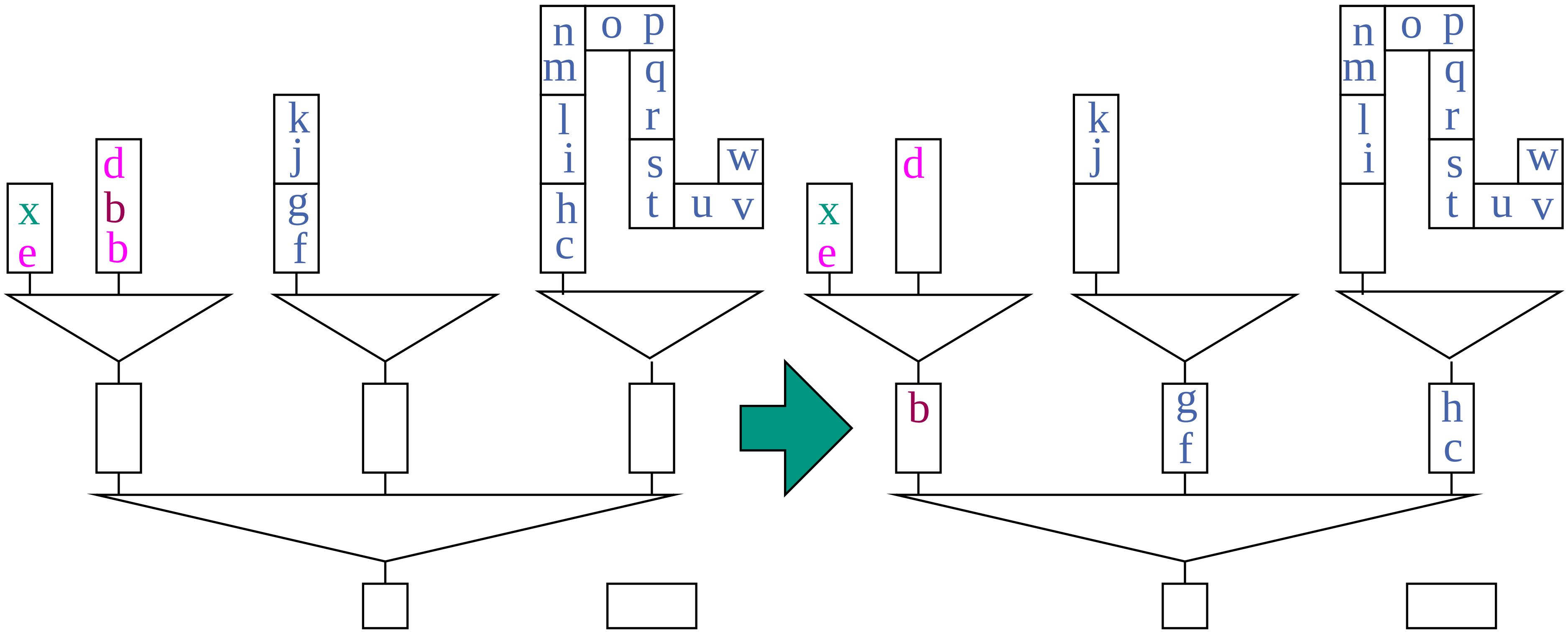
Example

DeleteMin $\rightsquigarrow$ 3; DeleteMin $\rightsquigarrow$ a;



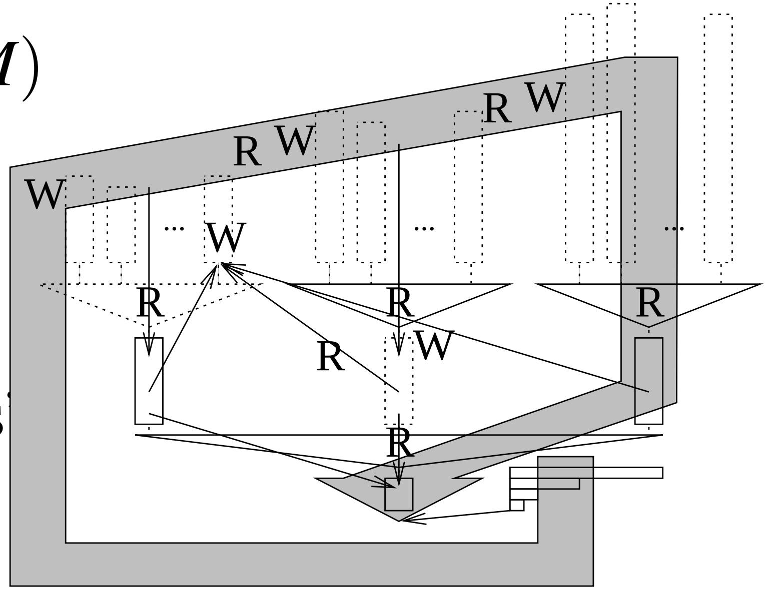
Example

DeleteMin $\rightsquigarrow$ b



Analysis

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



block accesses: $\mathrm{sort}(I)$+"small terms"

key comparisons: $I \log I$ + "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

Group buffer size: 256

Merging degree k: 128

Compiler flags: Highly optimizing, nothing advanced

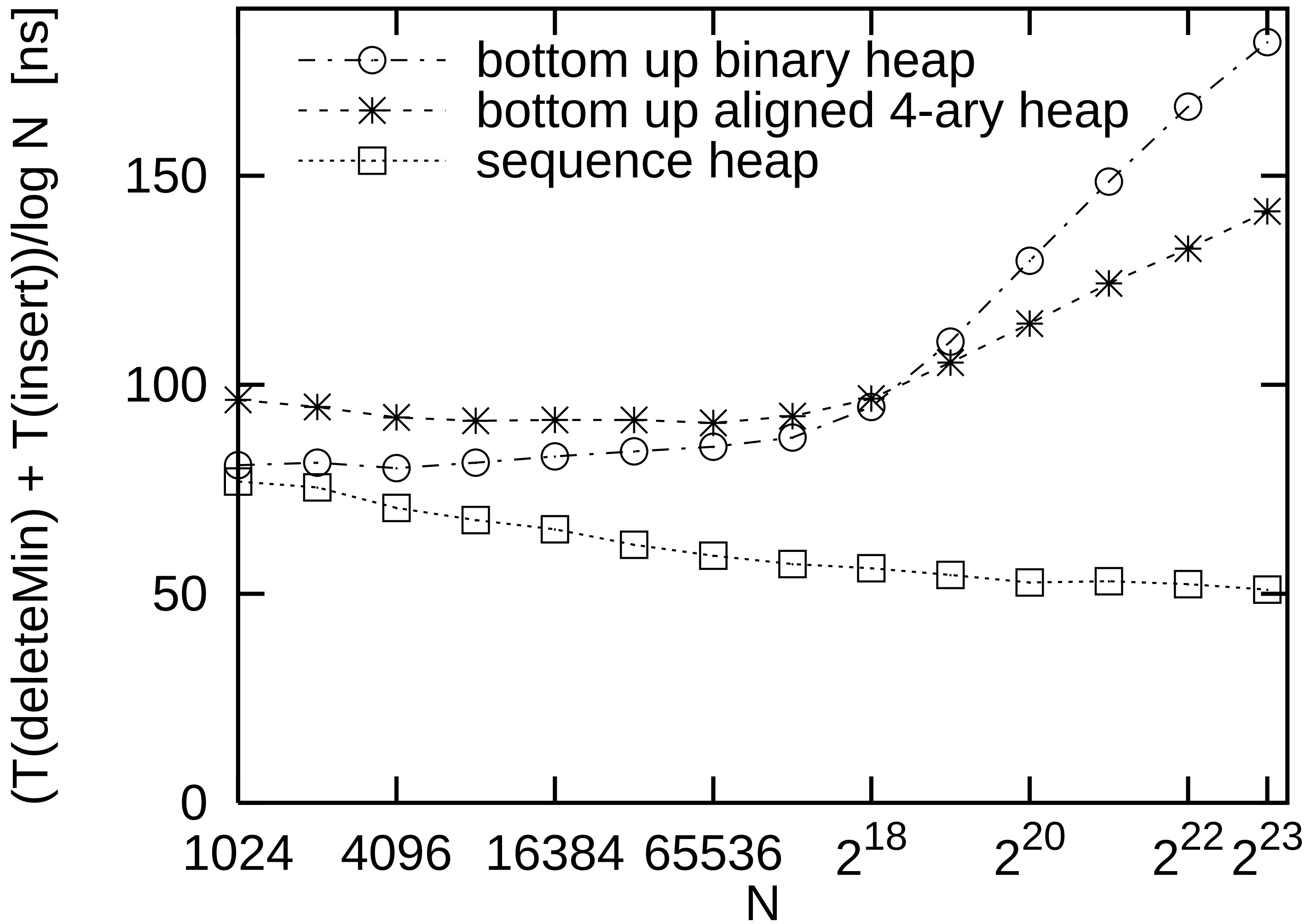
Operation Sequence:

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

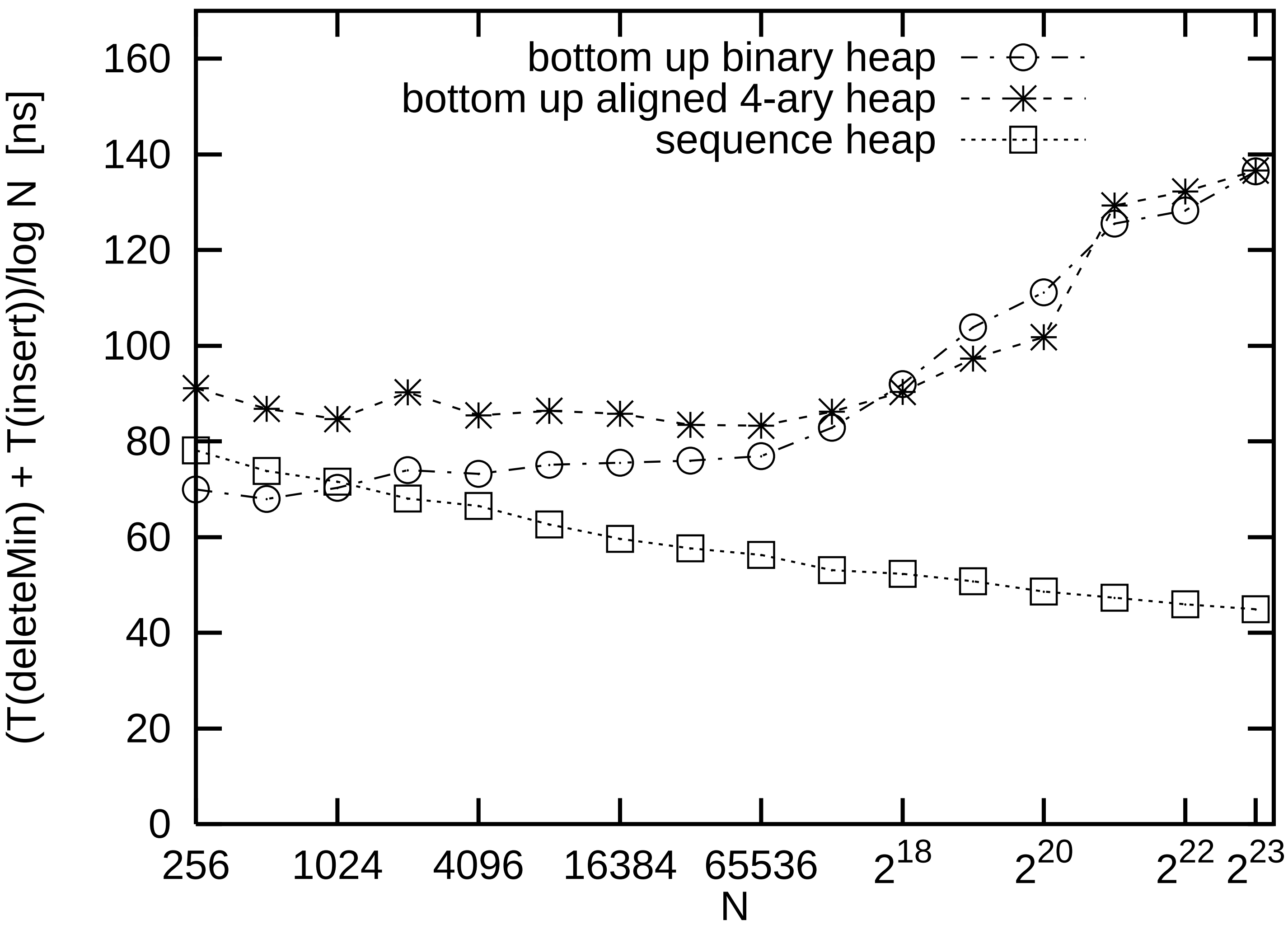
Near optimal
: performance on
all machines tried!

Near optimal performance on all machines tried!

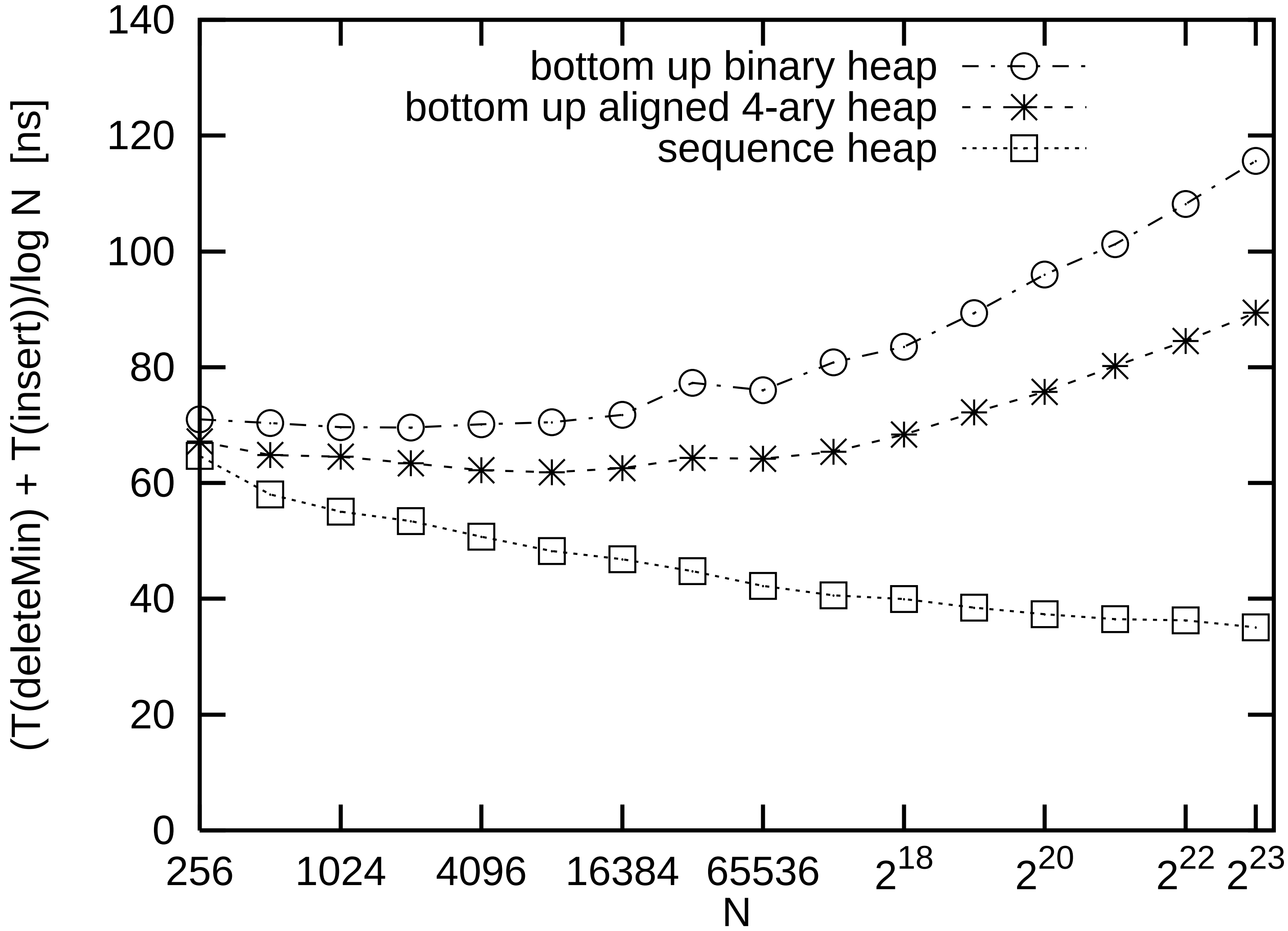
MIPS R10000, 180 MHz



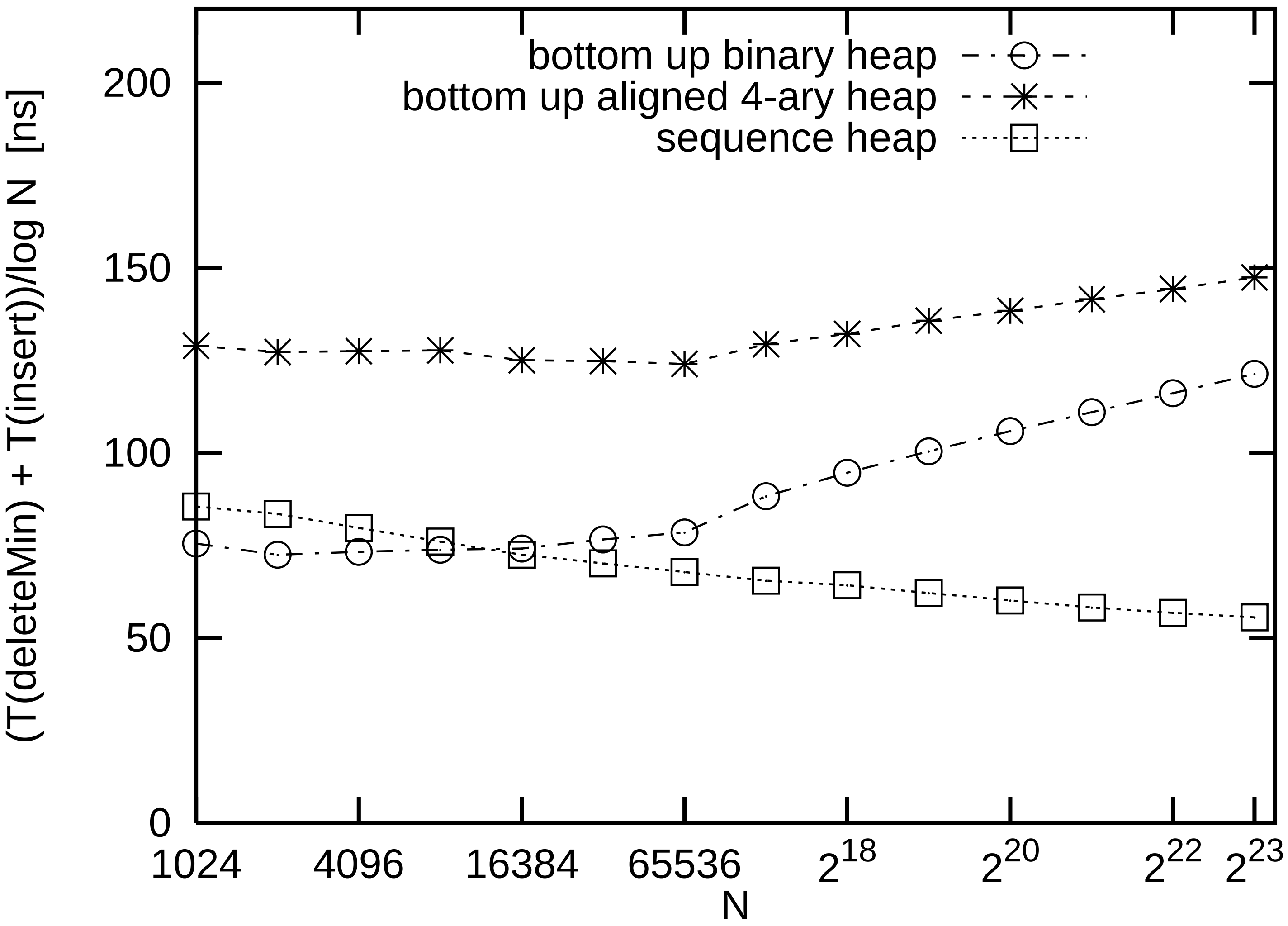
Ultra-SparcIIi, 300 MHz



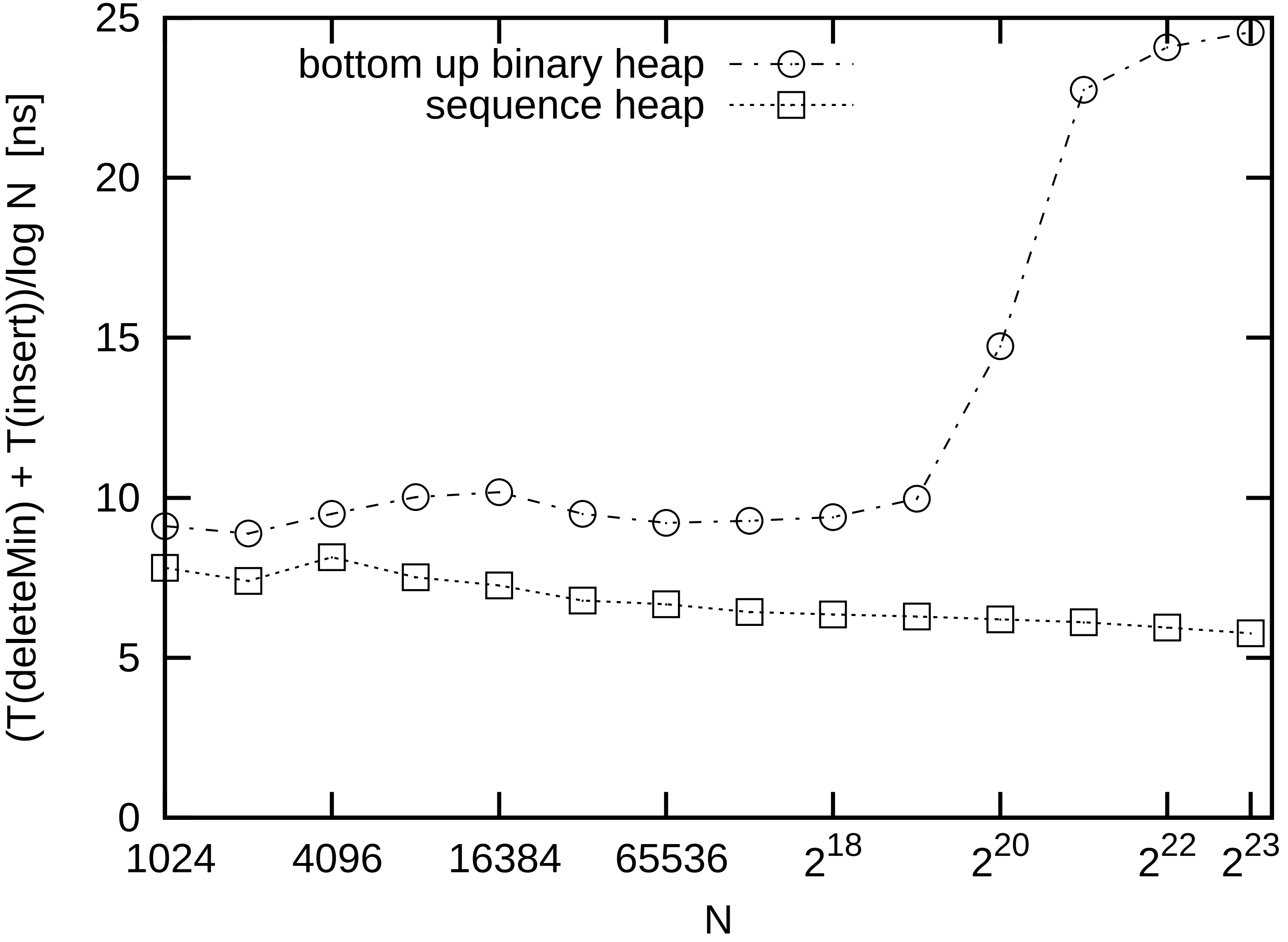
Alpha-21164, 533 MHz



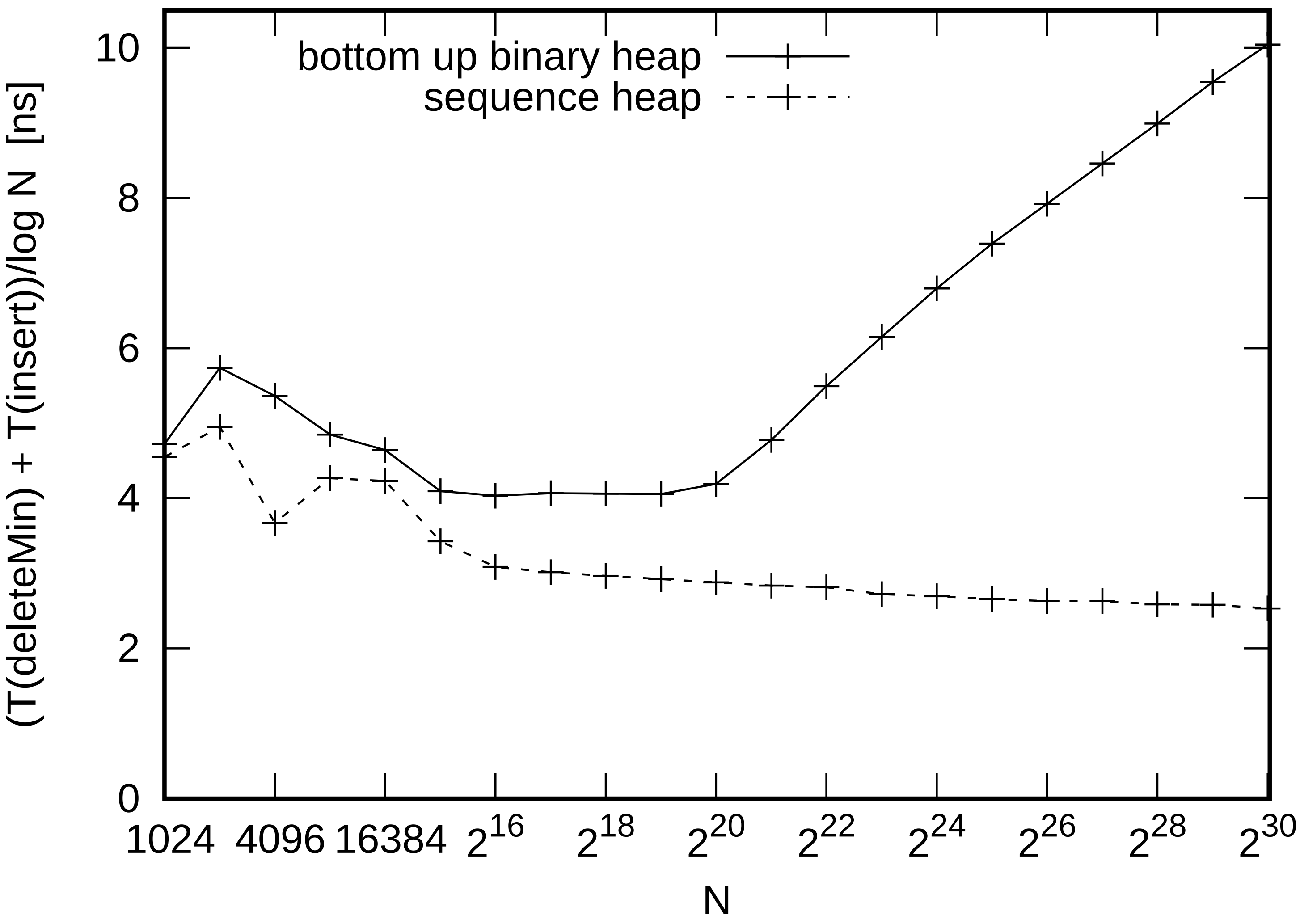
Pentium II, 300 MHz



Core2 Duo Notebook, 1.??? GHz

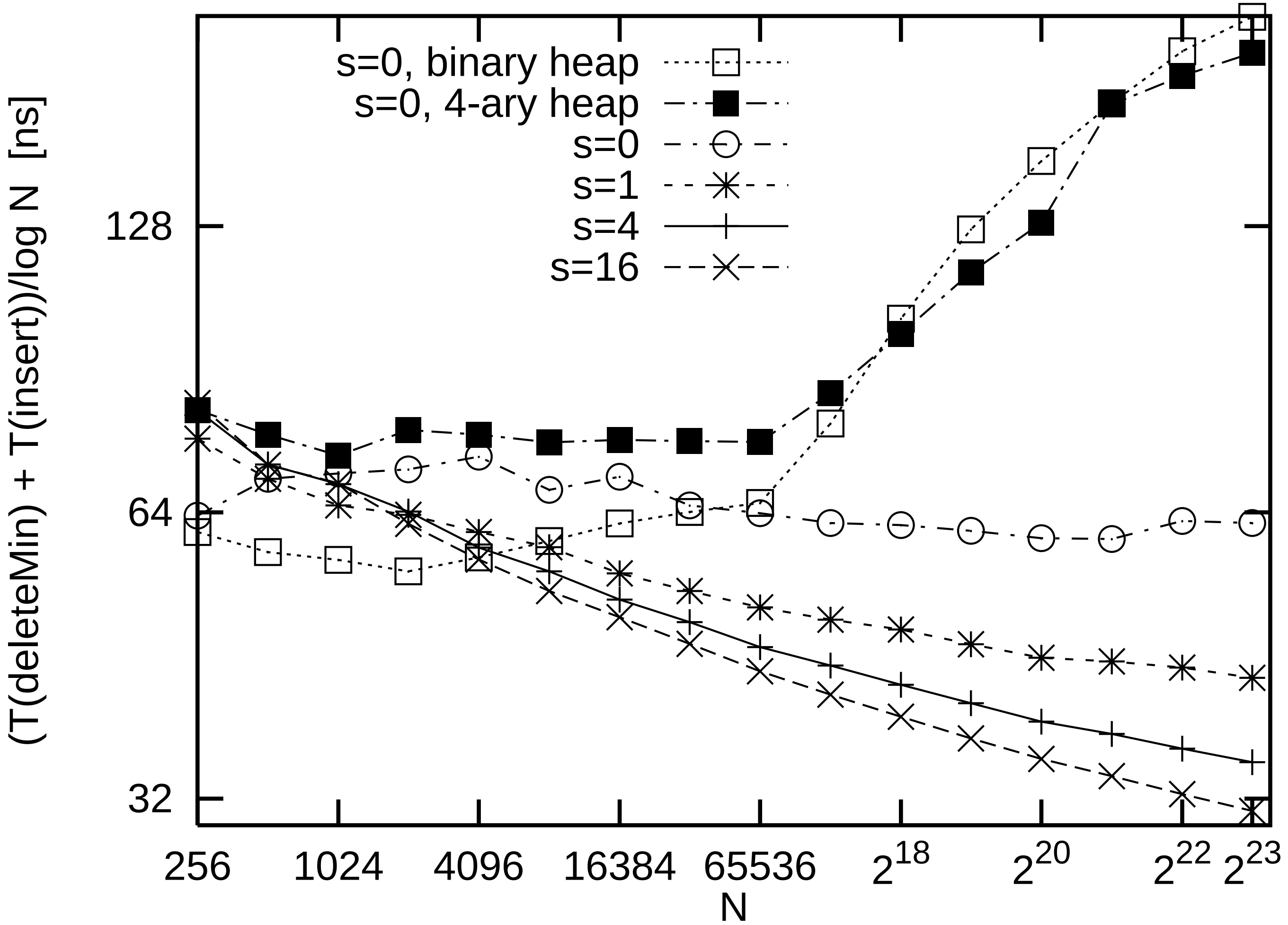


AMD Ryzen 1800X, 16MB L3, 3.6 GHz, 2017



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

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



Methodological Lessons

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

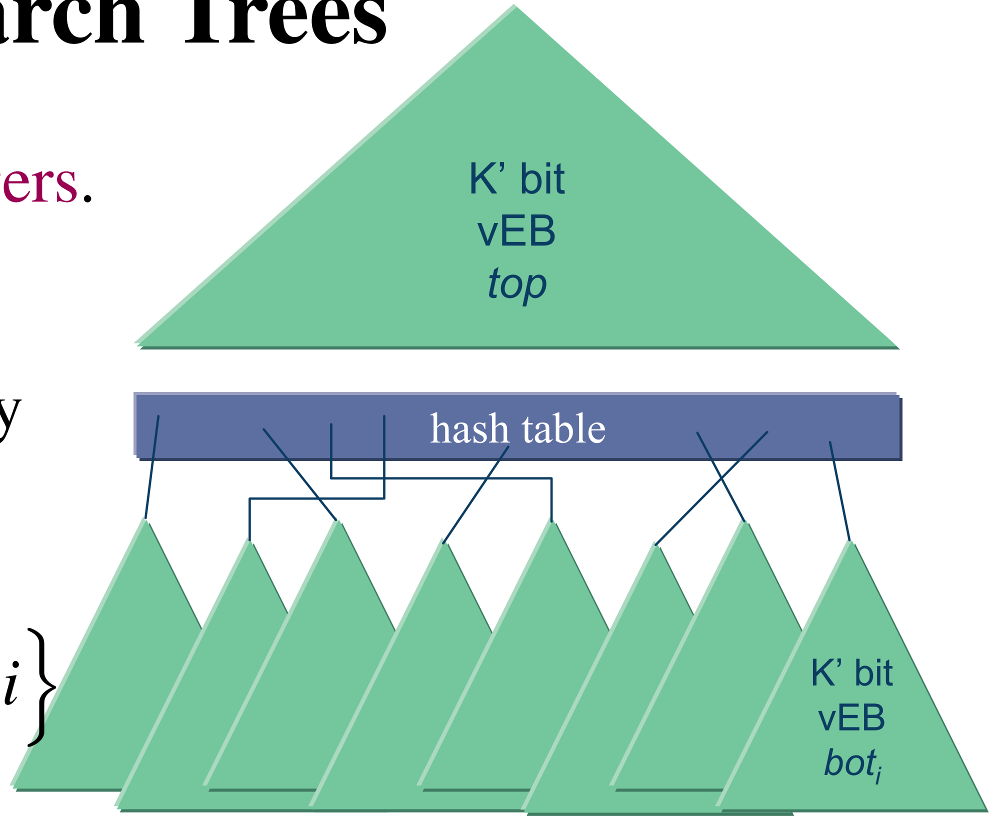
- Reproducibility demands publication of source codes (4-ary heaps, old study in Pascal)
- Highly tuned codes in particular for the competitors (binary heaps have factor 2 between good and naive implementation).
  How do you compare two mediocre implementations?
- Careful choice/description of inputs
- 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

4 van Emde-Boas Search Trees

- Store set M of $K = 2^k$-bit integers.
  later: associated information
- $K = 1$ or $|M| = 1$: store directly
- $K' := K/2$
- $M_i := \left\{ x \bmod 2^{K'} : x \operatorname{div} 2^{K'} = i \right\}$
- 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

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

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

    $i :=$ top.locate($i + 1$)

    $j := \min M_i$ // precomputed!

  else $j := M_i$.locate(j)

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

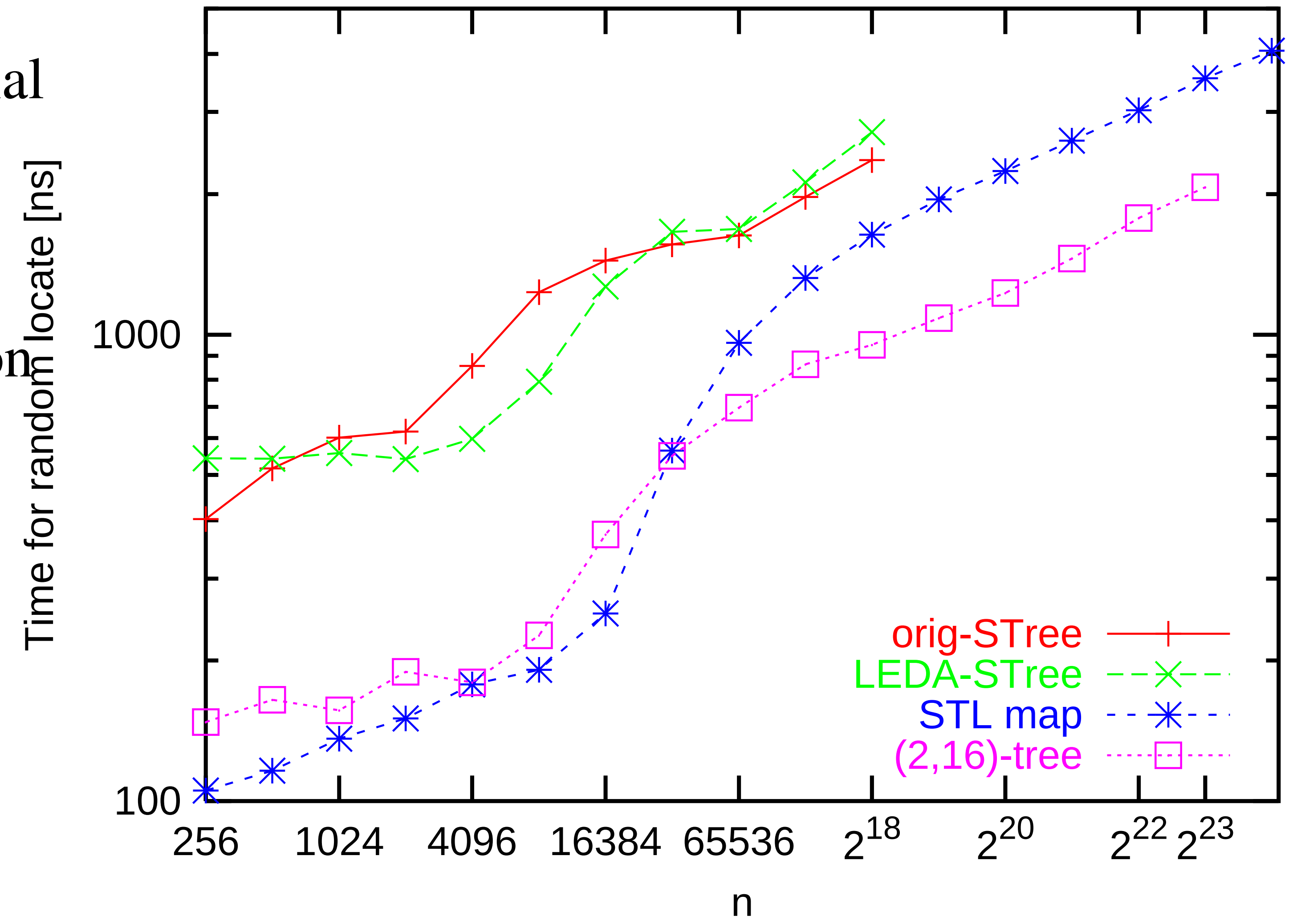
Comparison with Comparison Based Search Trees

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

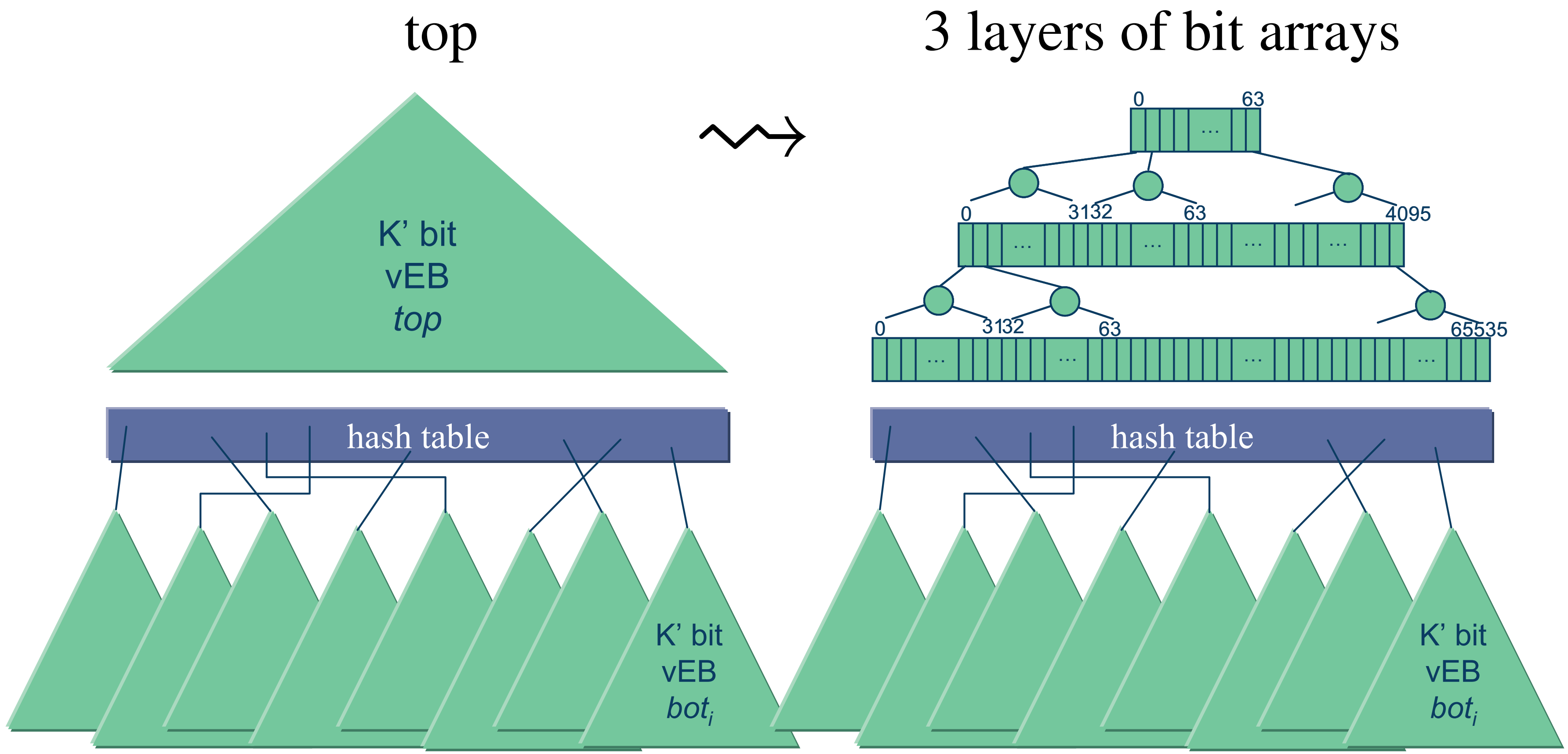
Problems:

Many special case tests

High space consumption



Efficient 32 bit Implementation



Layers of Bit Arrays

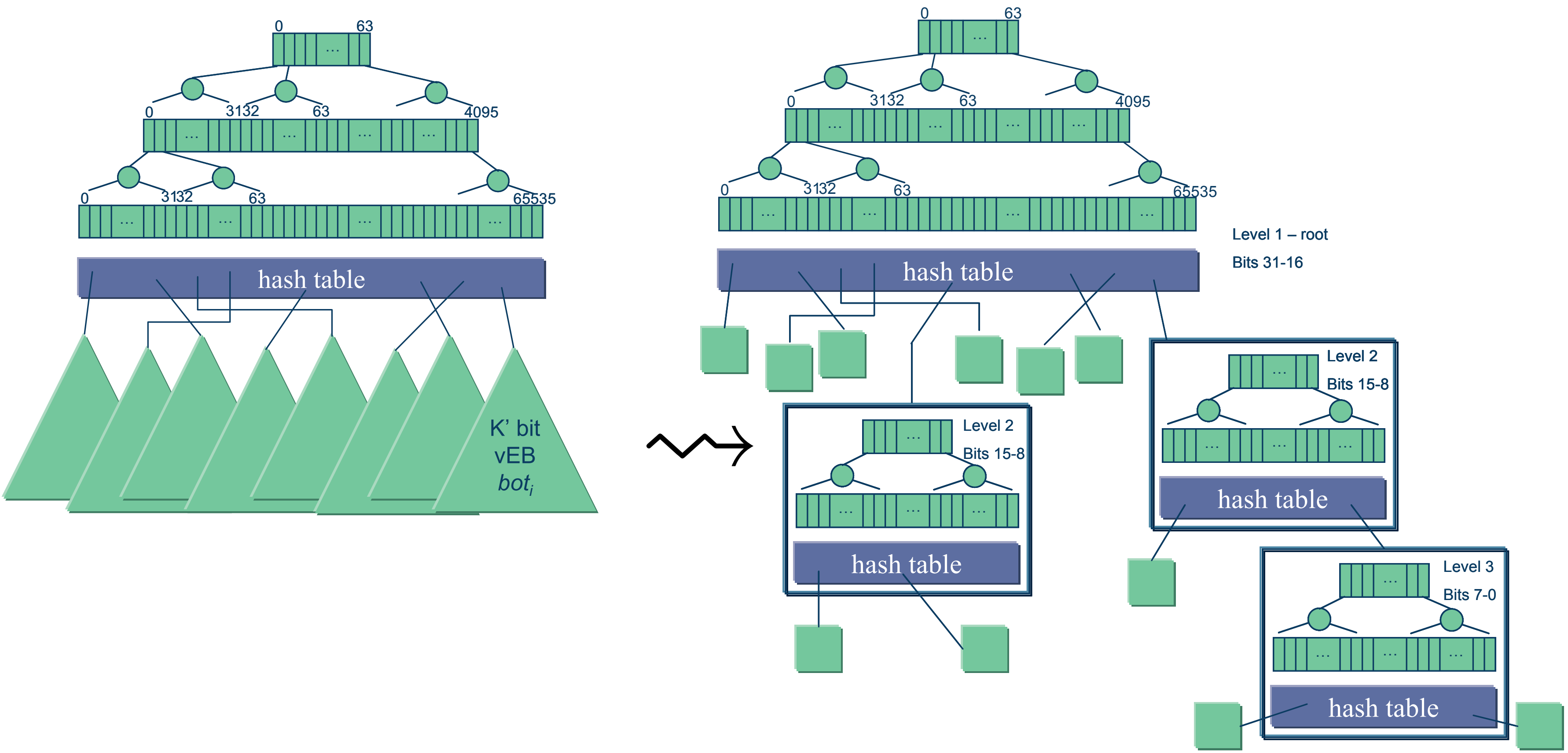
$t^1[i] = 1 \text{ iff } M_i \neq \emptyset$

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

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

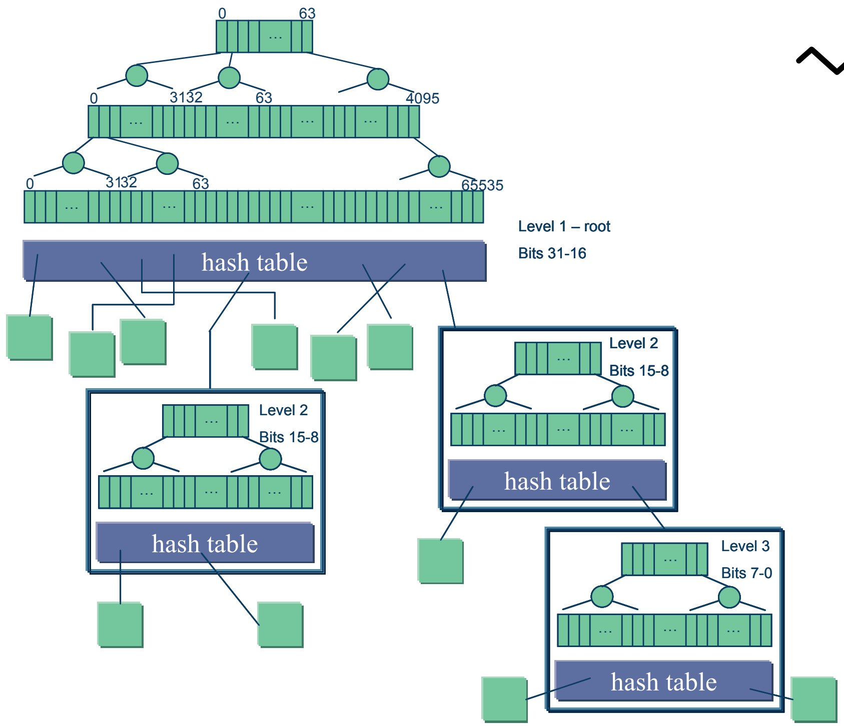
Efficient 32 bit Implementation

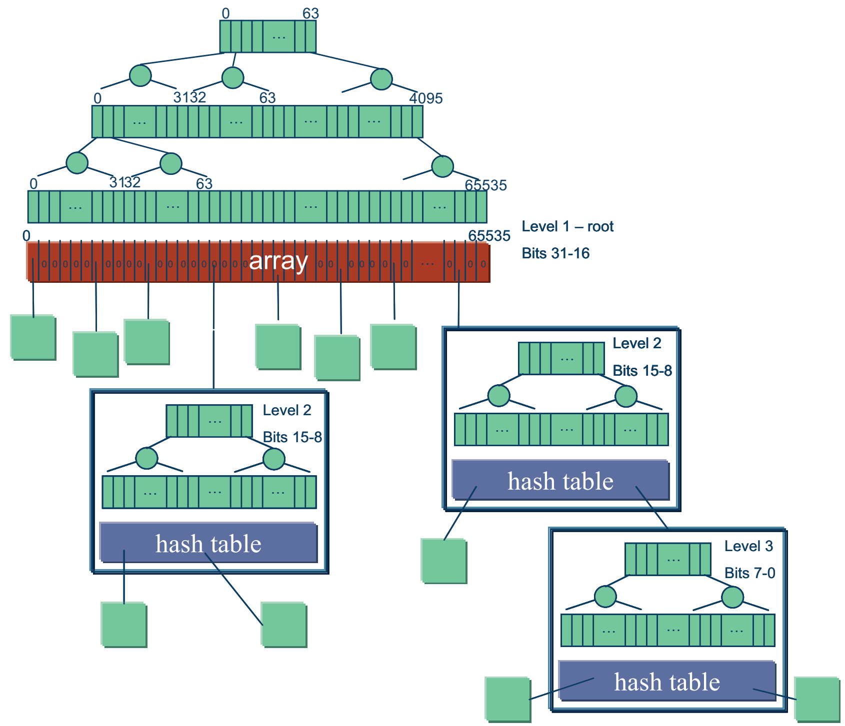
Break recursion after 3 layers



Efficient 32 bit Implementation

root hash table $\rightsquigarrow$ root array





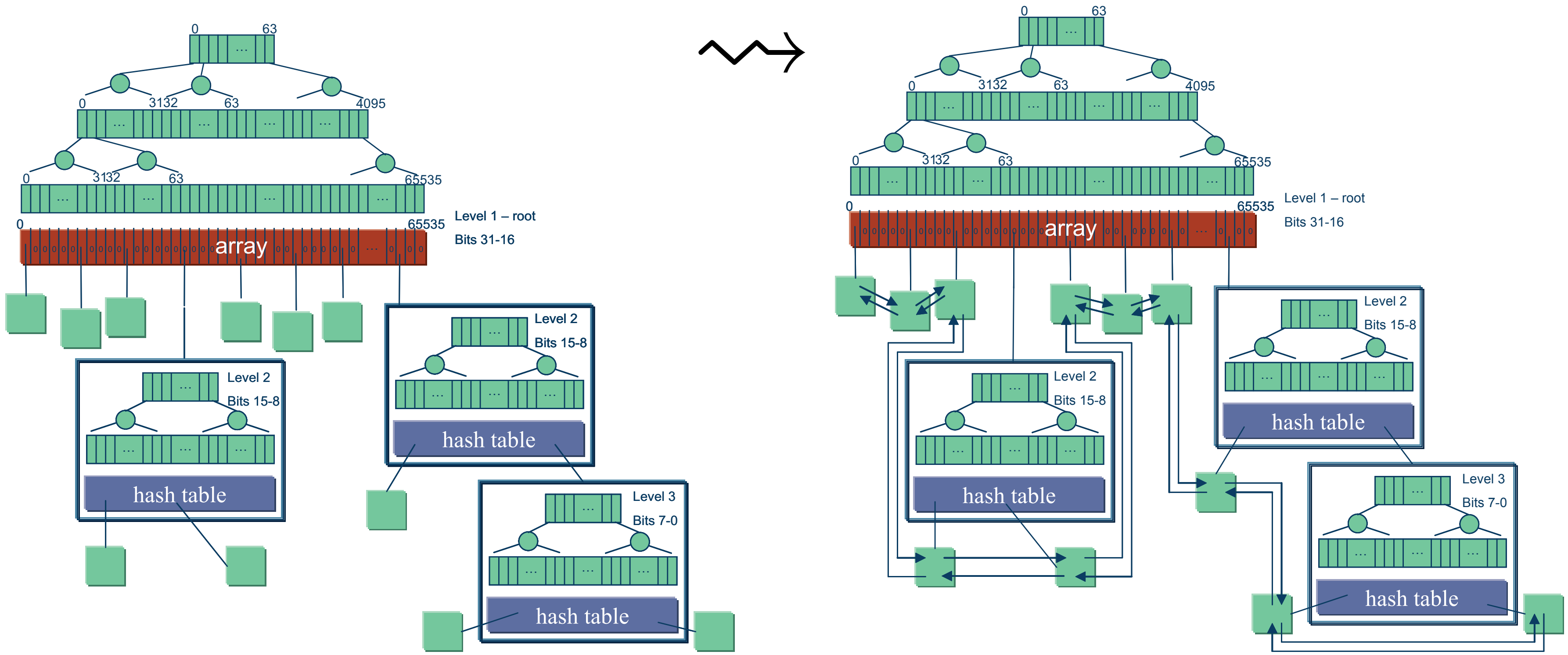
Efficient 32 bit Implementation

Tuned small hash tables with 8-bit keys:

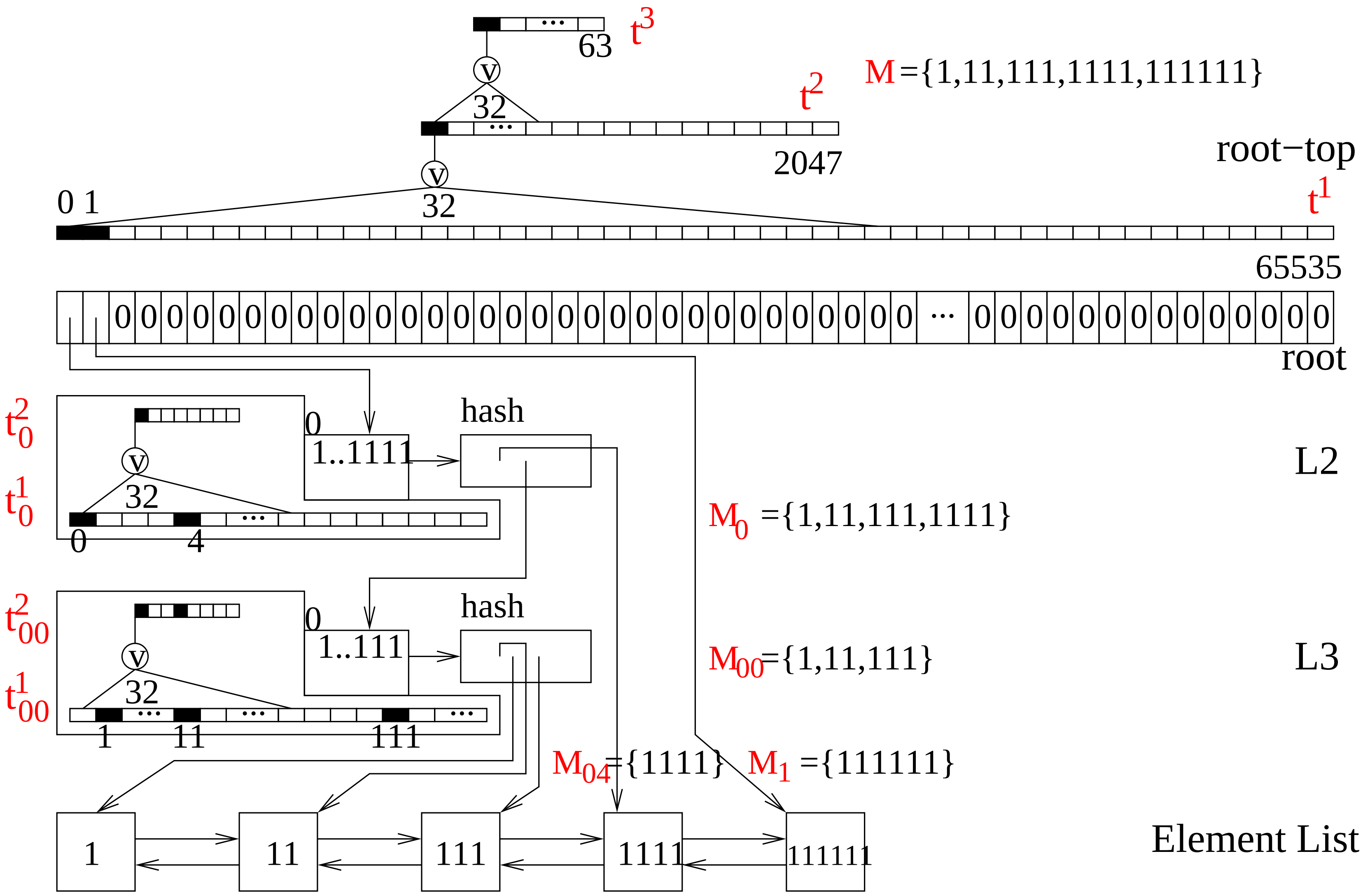
- Tabulate hash function (256 entries)
  $\rightarrow$ very fast
- Make it a random permutation
  $\rightarrow$ reduces collisions

Efficient 32 bit Implementation

Sorted doubly linked lists for associated information and range queries



Example


63
t3
32
2047
t2
M ={1,11,111,1111,111111}
root−top
0 1
32
t1
65535
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ··· 0 0 0 0 0 0 0 0 0 0 0 0 0 0
root
t0 2
t0 1
32
0
4
0
1..1111
hash
L2
M0 ={1,11,111,1111}
t00 2
t00 1
32
1
11
111
0
1..111
hash
M00={1,11,111}
L3
M04={1111}
M1 ={111111}
1
11
111
1111
111111
Element List

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

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

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

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

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

return $r_{ij}[t_{ij}^1$.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 $t^1, t^2, t^1_i, t^1_{ij}$; $n = 64$ for t^3; $n = 8$ for t^2_i, t^2_{ij}

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 // $n-1 \cdots i \bmod n \cdots 0$

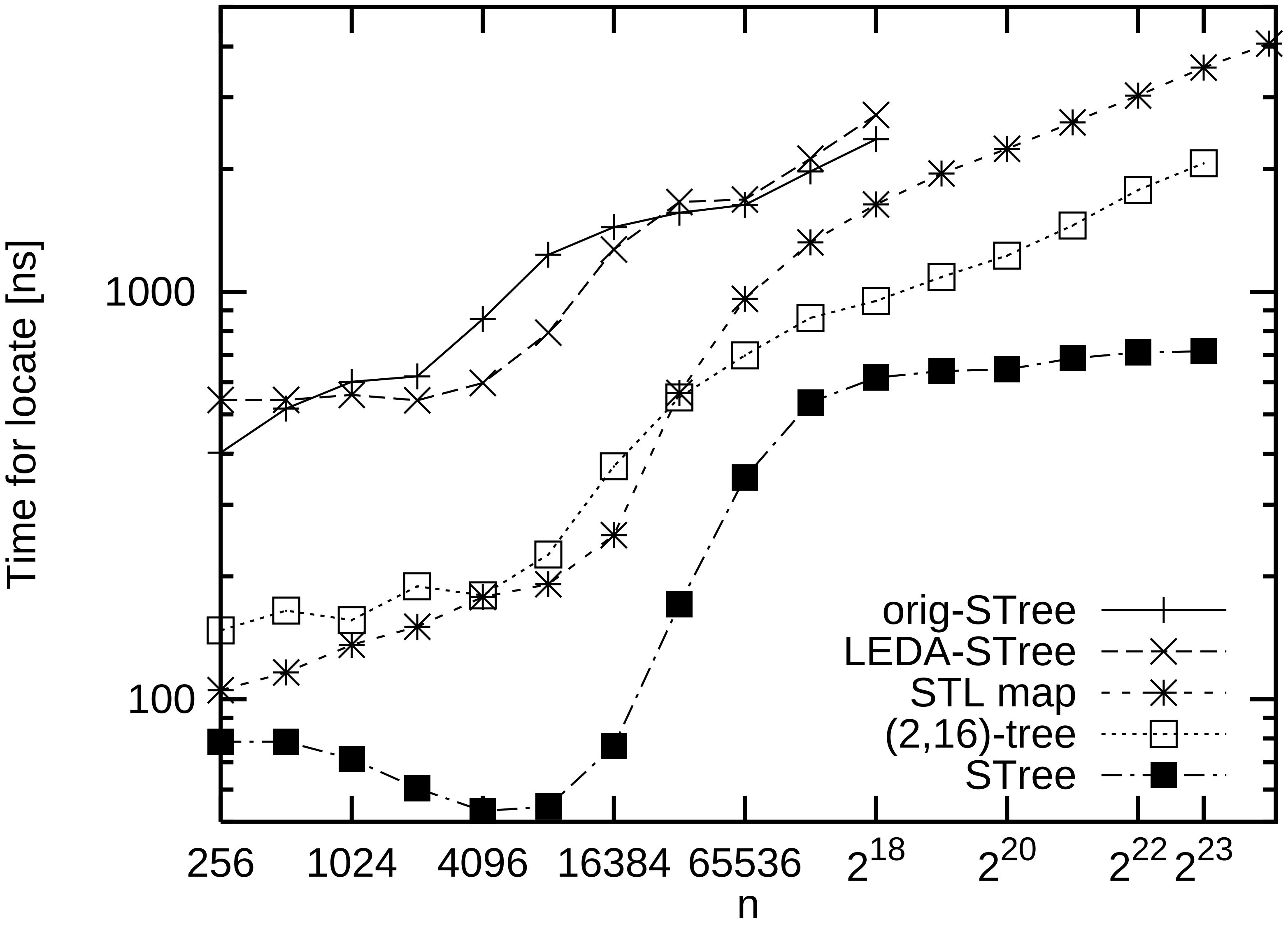
if $a = 0$ **then**

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

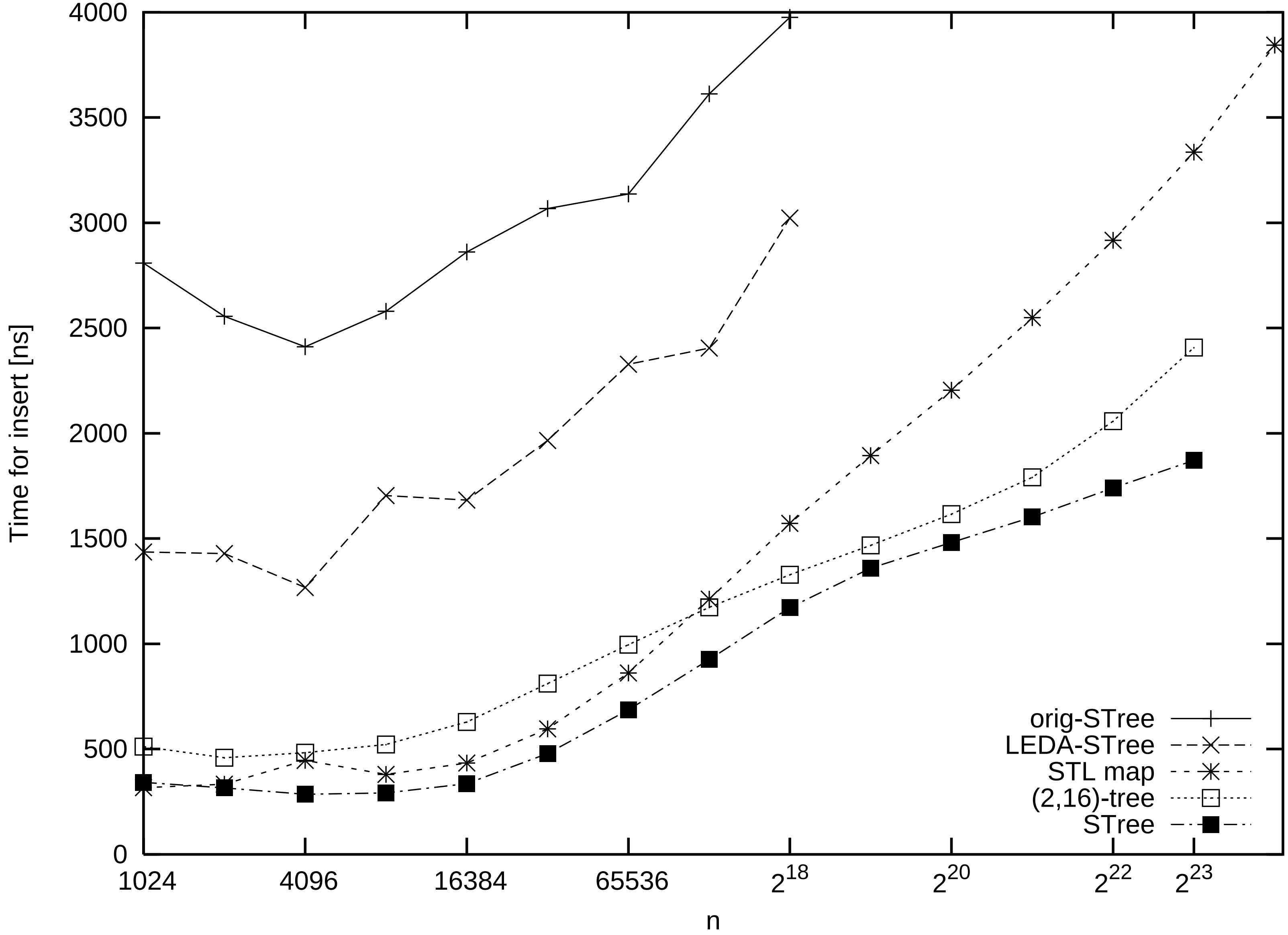
  $a := t^k[nj..nj+n-1]$

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

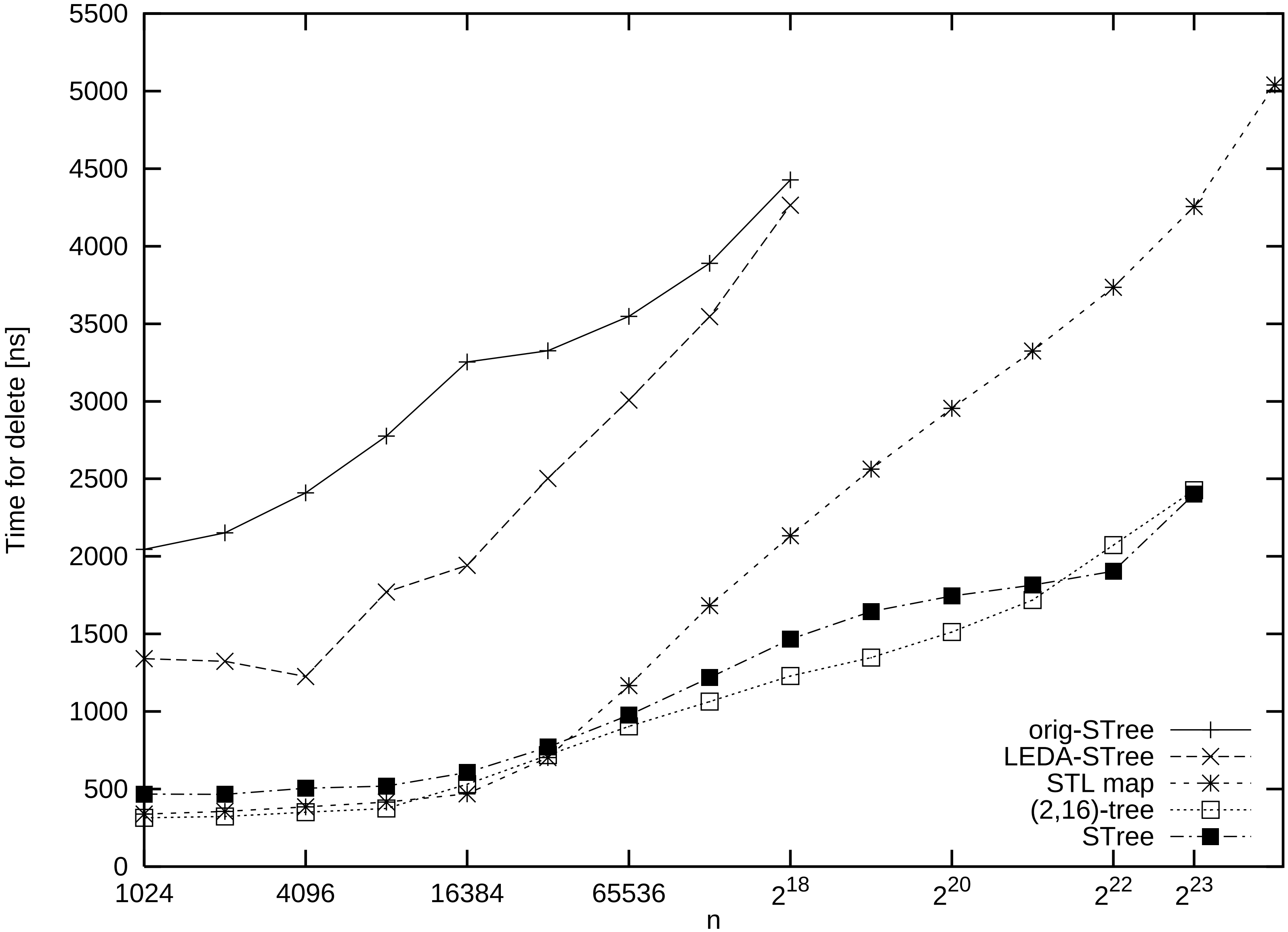
Random Locate



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

5 Hashing

"to hash" ≈ "to bring into complete disorder"

paradoxically, this helps us to find things more easily!

store set $M \subseteq \mathsf{Element}$.
$\mathsf{key}(e)$ is unique for $e \in M$.
support dictionary operations in $\mathcal{O}(1)$ time:

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

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

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

(Convention: key is <u>implicit</u>), e.g. $e = k$ iff $\mathsf{key}(e) = k$)

More Hash Table Operations

insertOrUpdate(e,u): If element e' with key(e) = key(e') is already present then update it to $u(e',e)$

build: from given elements

doAll: Iterate through all elements in the set, possibly updating or deleting them.

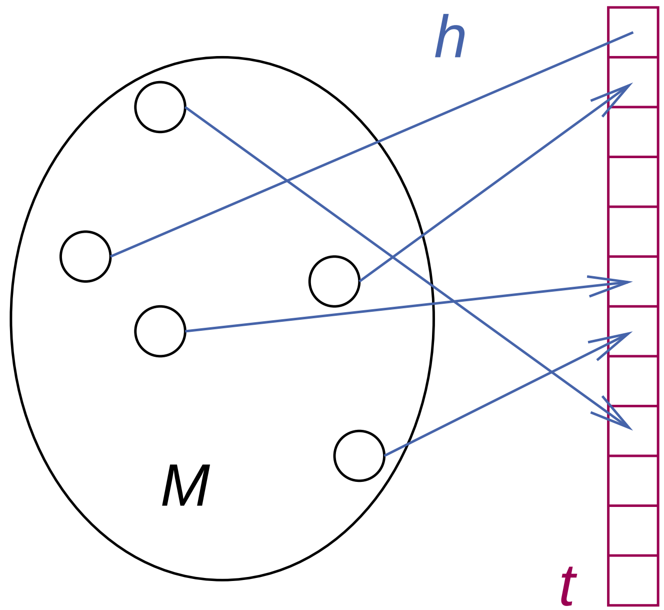
also init, find, contains, size, sample, clear, join, set operations.

Bulk operations can be faster and more cache efficient.

Deprecated: exposing buckets.

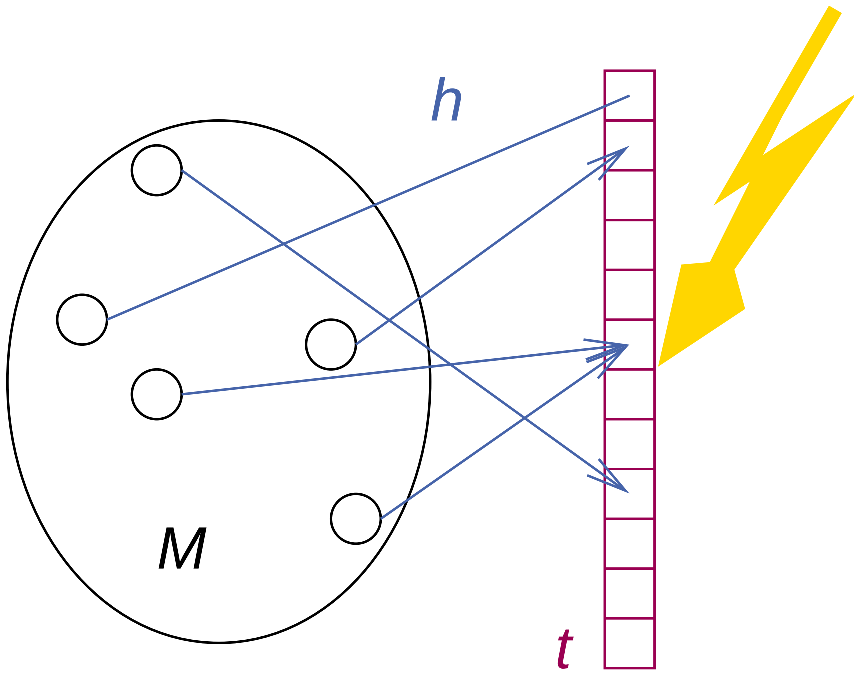
An (Over)optimistic approach

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



Collisions

perfect hash functions are difficult to obtain

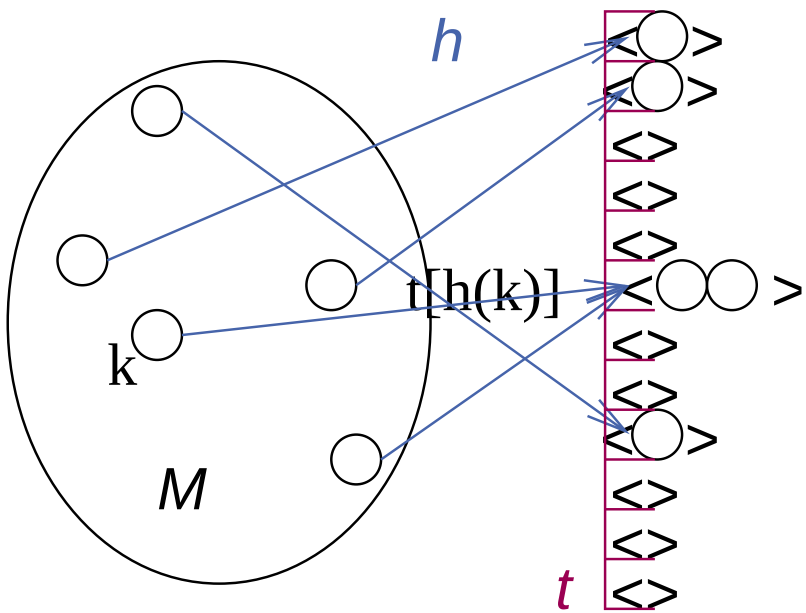


Example: Birthday Paradoxon

Collision Resolution

for example by closed hashing

entries: elements $\rightsquigarrow$ sequences of elements



Hashing with Chaining

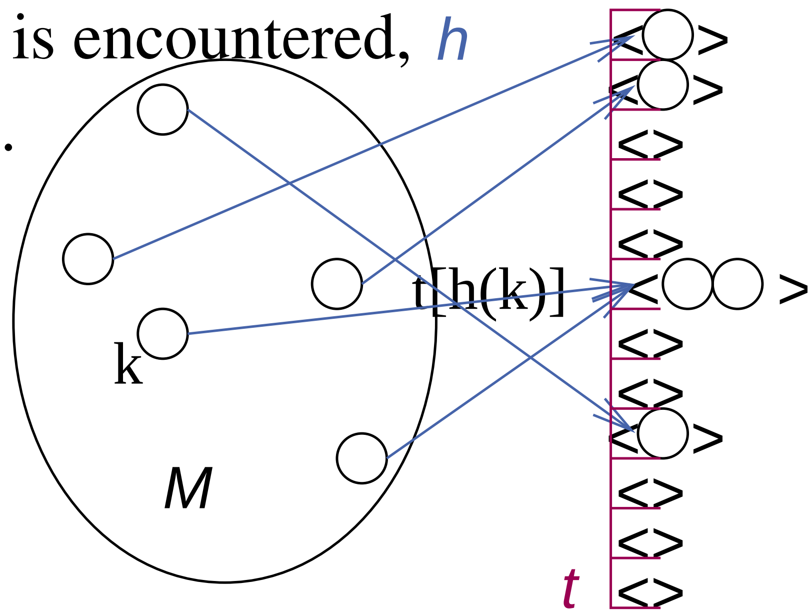
Implement sequences in closed hashing by singly linked lists

insert(e): Insert e at the beginning of $t[h(e)]$. constant time

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

find(k) : Scan through $t[h(k)]$.
If an element e with $h(e) = k$ is encountered, return it. Otherwise, return $\perp$.

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



Hashing with Linear Probing

Open hashing: go back to original idea.

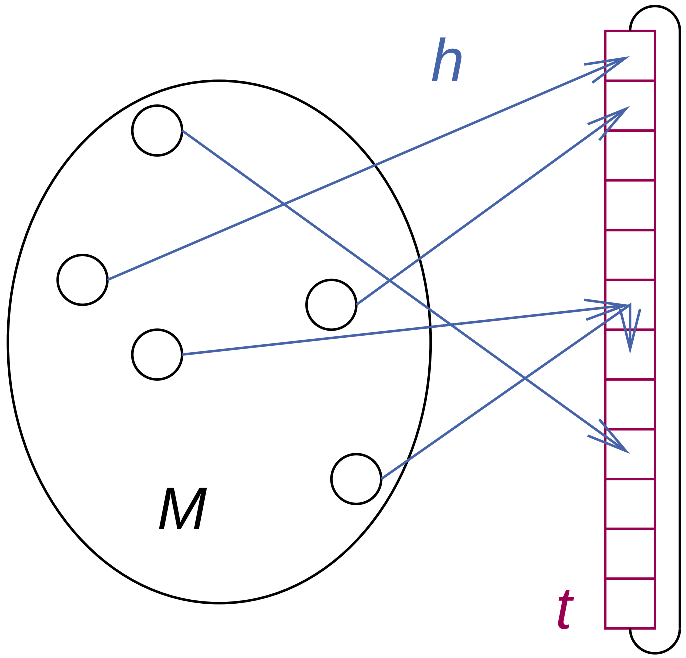
Elements are directly stored in the table.

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$(m, m' : \mathbb{N};\ h : \mathsf{Key} \to 0..m-1)$

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

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

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

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

assert $i < m + m' - 1$

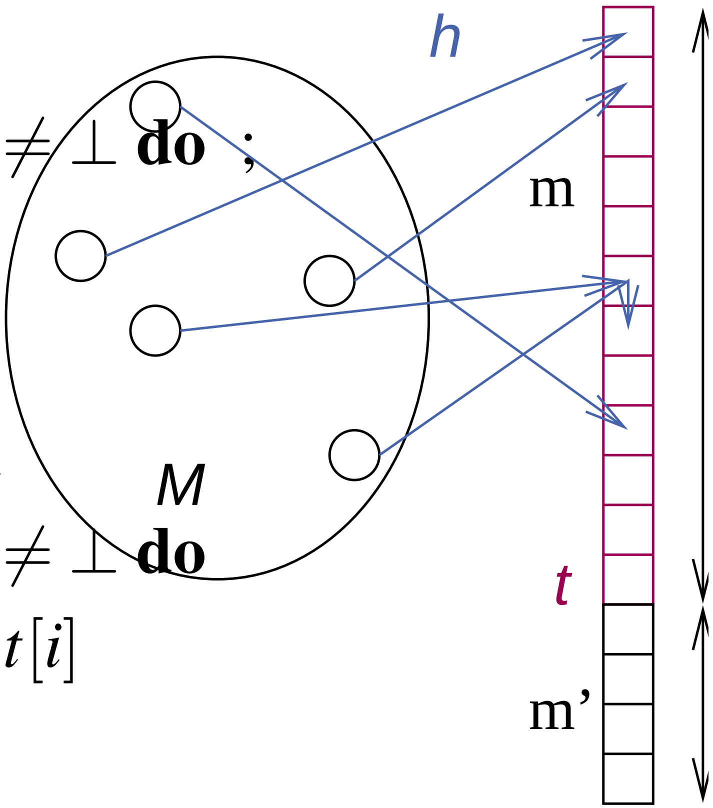
$t[i] := e$

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

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

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

return $\bot$



Remove

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

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

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

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

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

  // we plan for a hole at i.

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

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

Robin Hood Hashing

like linear probing but keep elements sorted by their hash function value.

Advantage: Minimizes maximum search distance.

Disadvantage: More expensive insertion

AE Details of Linear Probing

- Usually wrap-around rather than m' “blind” elements.
- We need a specialized empty element $\perp$. There are tricks to circumvent that
- Insert and unsuccessful find are slow for high load factors α

$$T_{\text{fail}} \approx \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$$

$\Rightarrow$ keep α small when space is not at a premium

- That may be add odds with a fast clear operation. There are tricks to circumvent that.
- Also careful when table is supposed to fit into cache.

More Hashing Issues

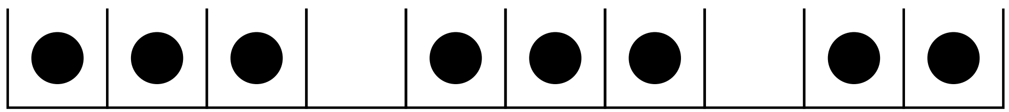
- High probability and worst case guarantees
  $\rightsquigarrow$ more requirements on the hash functions
- Space efficiency I: Avoid empty cells, pointers, ...
- Space efficiency II: Succinctness – approach lower bound
- Adaptive space: space efficiency at all times as the table grows or shrinks
- Referential integrity – allow pointers to elements
- Concurrent access
- Memory hierarchies
- Fast, provably effective hash functions
- Resilience agains DoS attacks? Encryption?

Space Efficient Hashing with Worst Case Constant Access Time

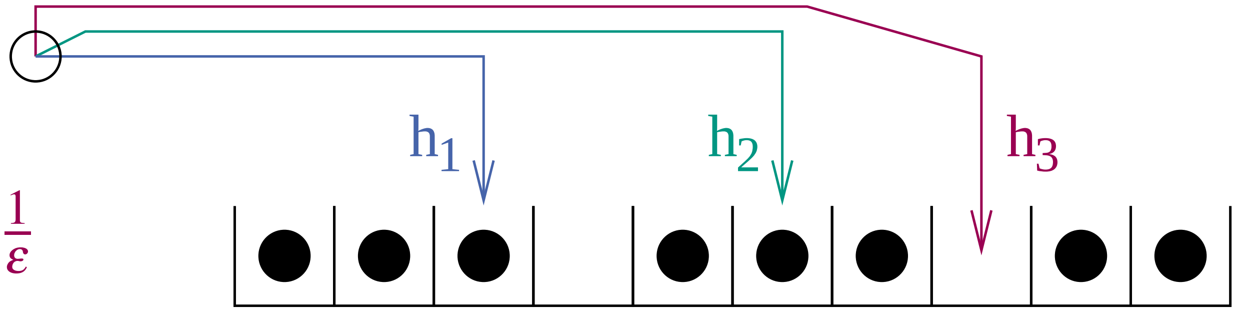
Represent a set of n elements (with associated information) using space $(1+\varepsilon)n$.

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

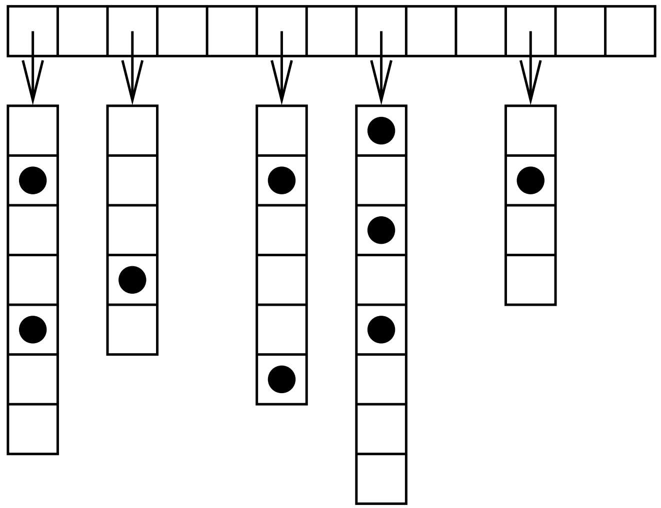
Assume a truly random hash function h



Related Work



Uniform hashing:
Expected time $\approx \frac{1}{\varepsilon}$



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.

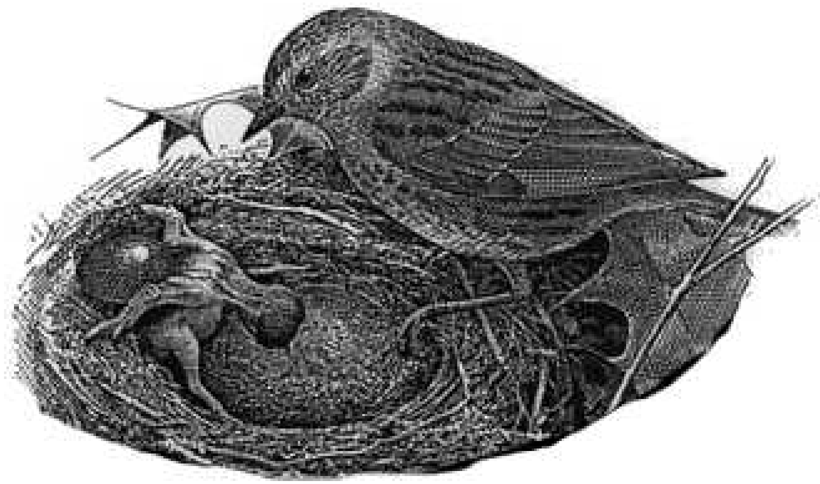
Cuckoo Hashing

[Pagh Rodler 01] Table of size $2 + \varepsilon$.

Two choices for each element.

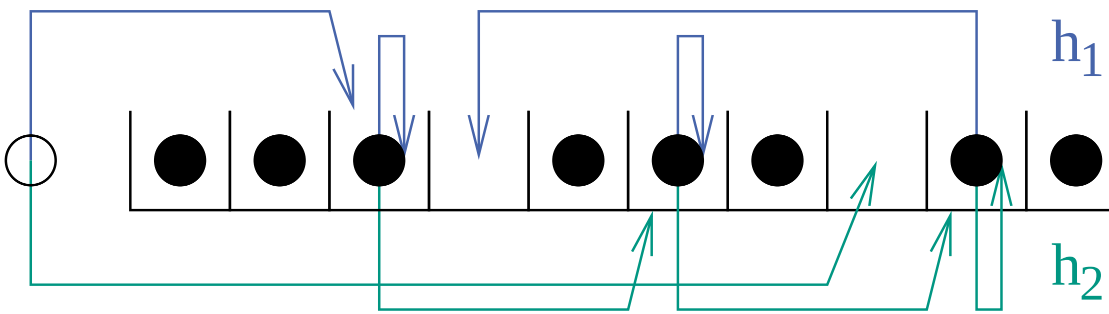
Insert moves elements;

rebuild if necessary.



Very fast lookup and insert.

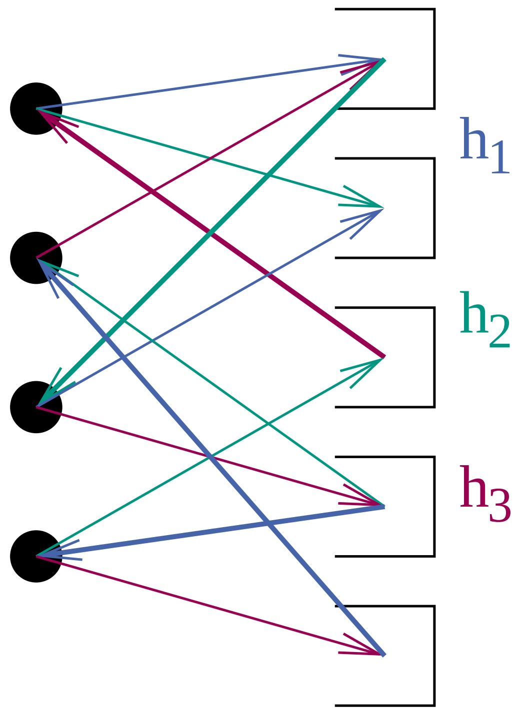
Expected constant insertion time.



H-ary Cuckoo Hashing [28]

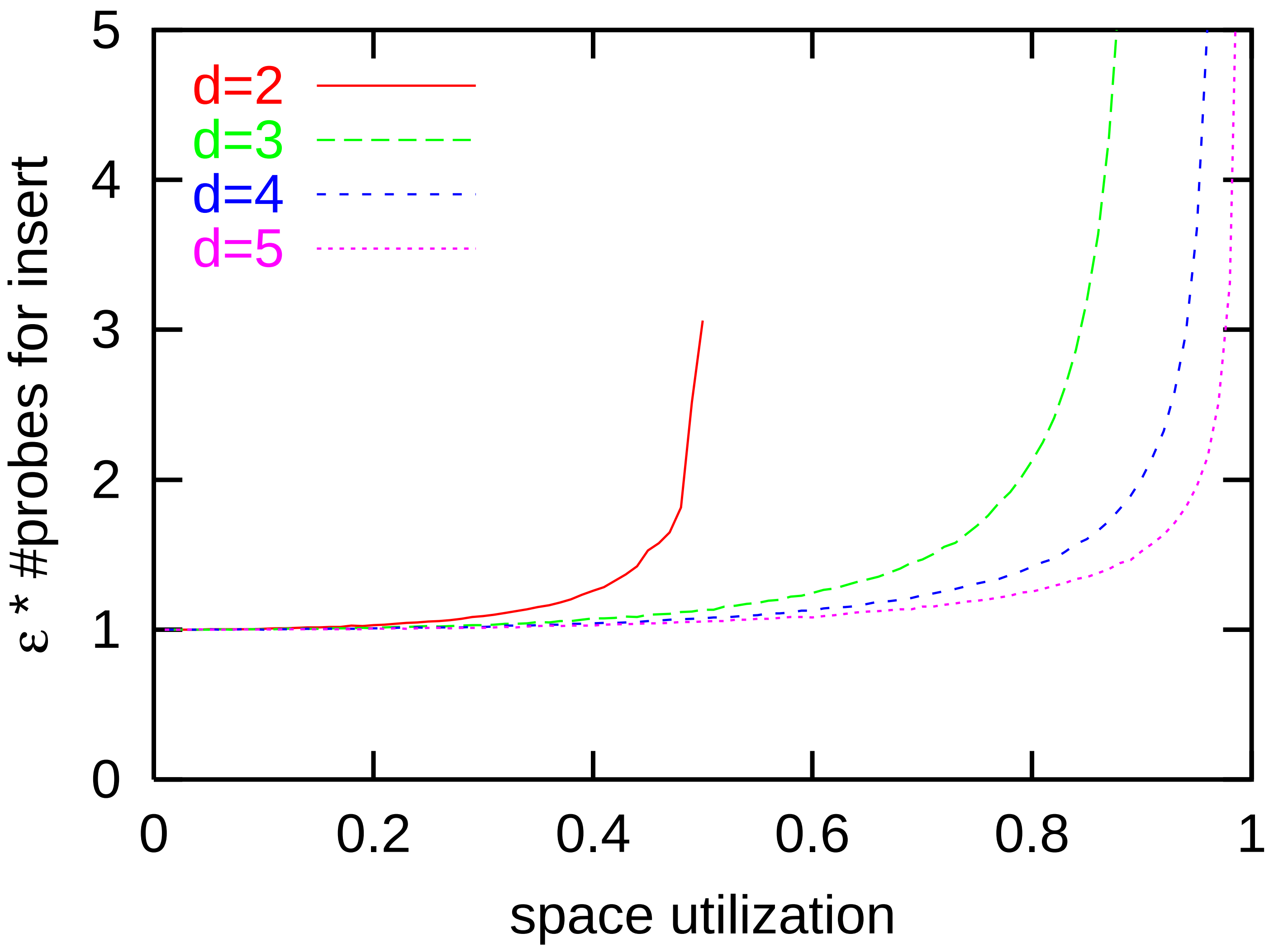
H choices for each element.

Worst case *H* probes for delete and lookup.



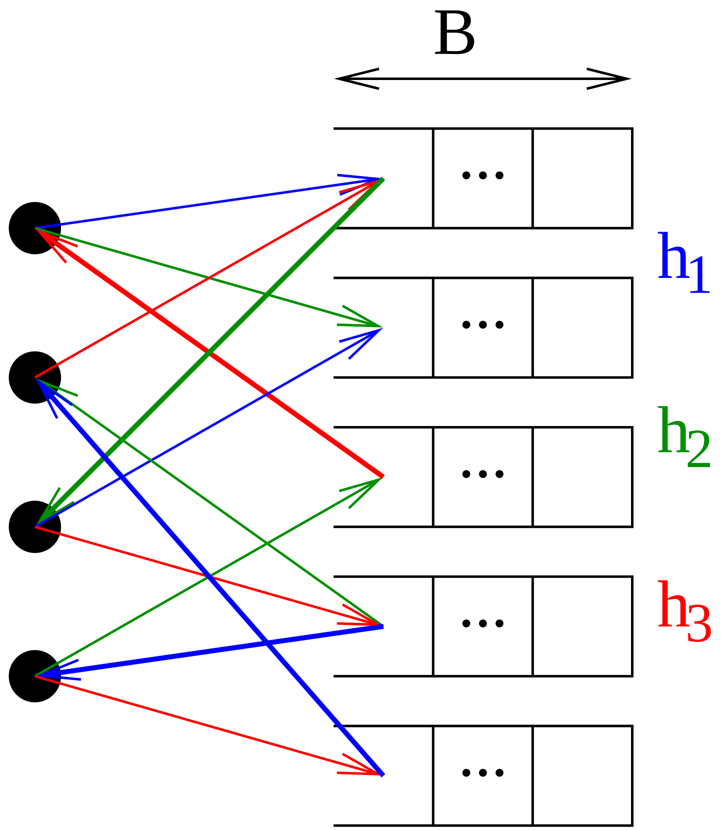
Task: maintain perfect matching
in the bipartite graph
$(L = \text{Elements}, R = \text{Cells}, E = \text{Choices})$,
e.g., insert by BFS of random walk.

Experiments



Blocked Cuckoo Hashing

Map elements to H blocks of size B.



Better space and cache efficiency

Threshold Values

$H \backslash B$	1	2	3	4	5	6	7	8
2	.5	.897	.959	.980	.989	.994	.996	.998
3	.918	.988	.997	.9992				
4	.977	.998	.9998	.99997				

Random Walk Based insert(x)

pick any hash function h_i

repeat patience times

k:= $h_i(x)$

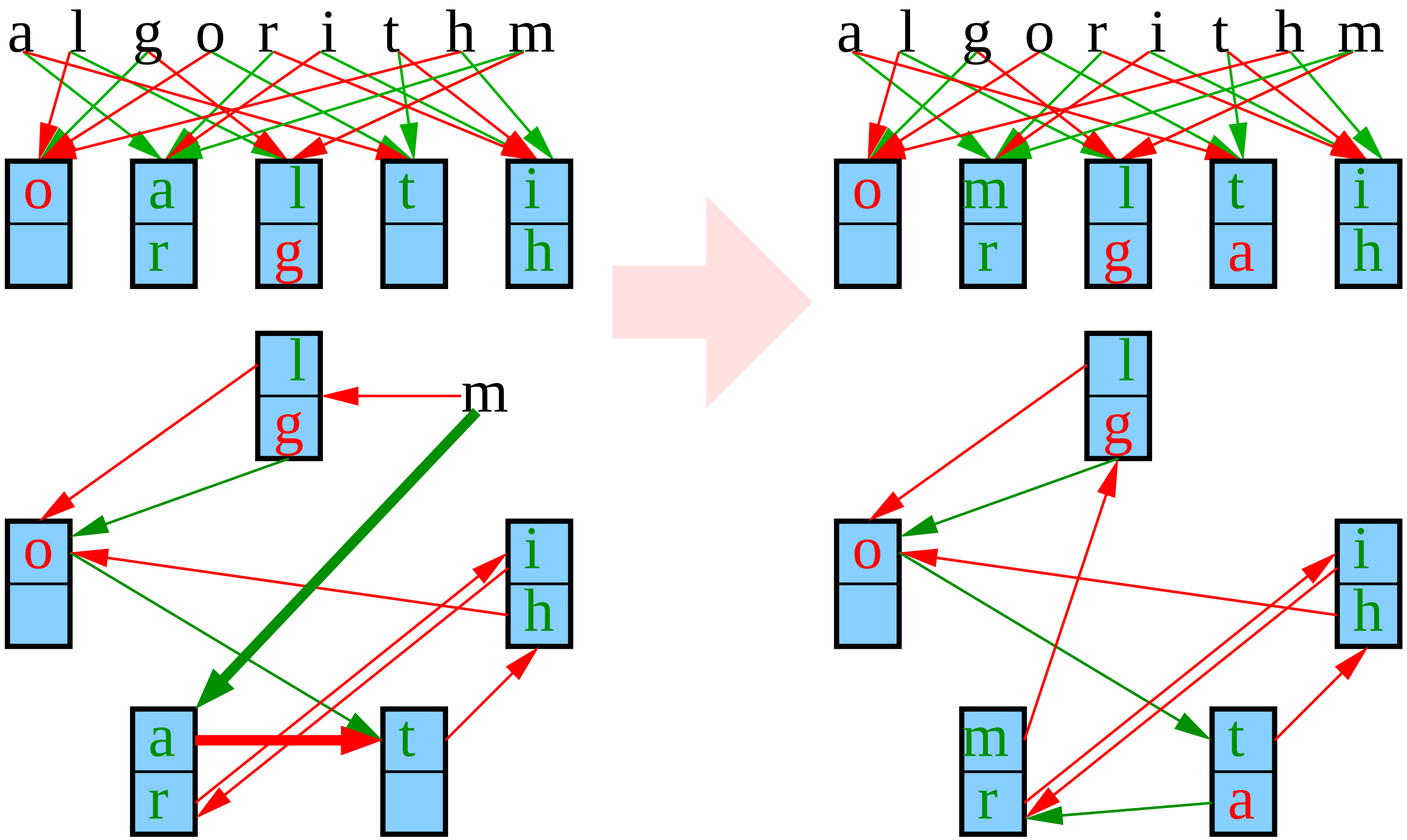
if $t[k]$ has a free slot **then** store x there; **return**

swap x and $t[k][j]$ for random $j \in 0..B-1$

pick a random h_i with $h_i(x) \neq k$

give up // exception, rehash or grow table

Cuckoo Insert Example



BFS Based insert(x)

Use BFS to find shortest path to a free slot.
Variant: Only store queue of explorable blocks without removing duplicates (rare anyway)

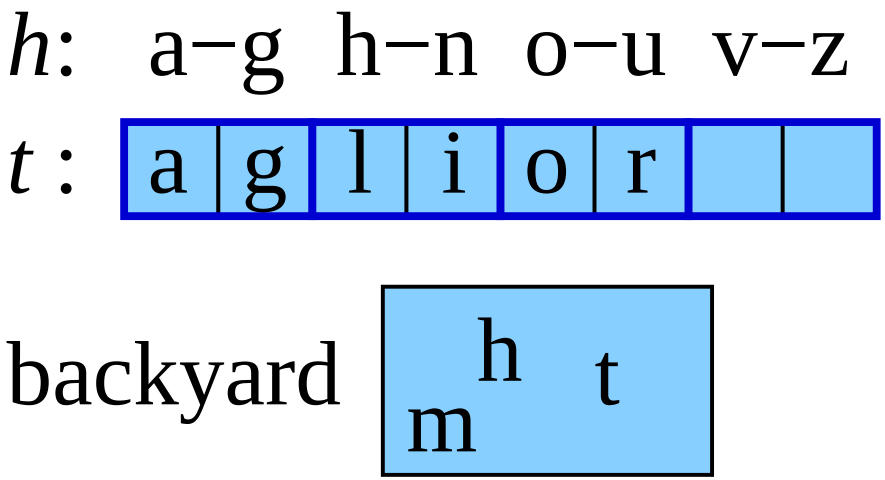
\+ Less write operations

\+ Allows optimal exploitation of space (without duplicate removal)

– Additional space for maintaining search frontier

Blocking and Backyards

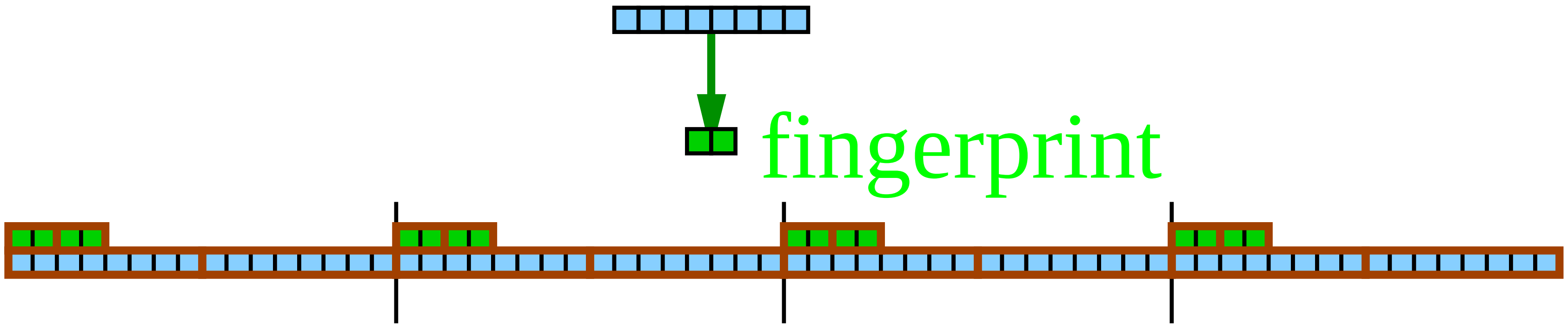
Consider Cuckoo Hashing with $H = 1$. How to insert when a block is full? Idea: bump something to another level of the data structure – the backyard



Fingerprints

In a block of size B, use $\approx \log B$ hash bits of each elements as fingerprint – say 8 bits.

Bit-parallel or SIMD-parallel search in fingerprints accelerates search.



Succinct Hash Tables

Simplification for now:

- Keys are random
- No associated information
  (Easy to add back. Just messes up notation here.)

Information theoretic lower bound for storing n elements from a domain of size U:

$$\log \binom{U}{n} \approx n \log \frac{U}{n} \text{ bits.}$$

Quotienting for "Succinctization"

Suppose also for now that there are no hash collisions (h is *perfect*).

Store x div m in $t[x \bmod m]$.

Retrieve $x = t[i]m + i$.

Allowing Collisions

Derive "some" information from storage location.

Blocks and Backyards: With M blocks,
store x div M somewhere in block x mod M (or bump).
Yields $\log M = \log \frac{m}{B}$ bits of quotient information.

Slick Hash: (see below) Similar to Blocking.

Cuckoo Hashing: Use H-partite hashing with on subtable of size $\frac{m}{H}$ for each hash function. Continue as above. Yields $\log \frac{m}{HB}$ bits of quotient information.

Linear Probing: Cleary's trick [29]. Use 2–3 bits per table entry of metadata to track hash values of stored elements.
(also in "Quotient Filters" [30, 31])

Nonrandom Keys

Rather than a hash function h,
use an invertible pseudorandom permutation π.
For blocked case:
Store $\pi(x)$ div M in block $\pi(x)$ mod M.
Retrieve $x = \pi^{-1}(ym+i)$ for a value y stored somewhere in block i.

Fast Pseudorandom Permutations

Linear congruential:

$\pi(x) := ax + c \bmod U$ for a relatively prime to U.

$\pi^{-1}(y) = a^{-1}(y - c)$ where a is a multiplicative inverse of a

(can be computed using the Extended Euclidian Algorithm).

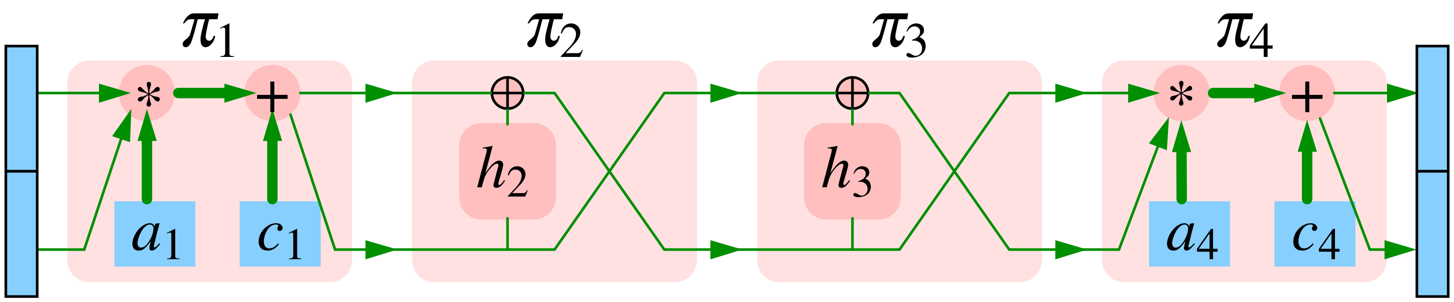
Feistel Permutations

Consider a hash function $h : \mathbb{Z}_u \to \mathbb{Z}_u$ and

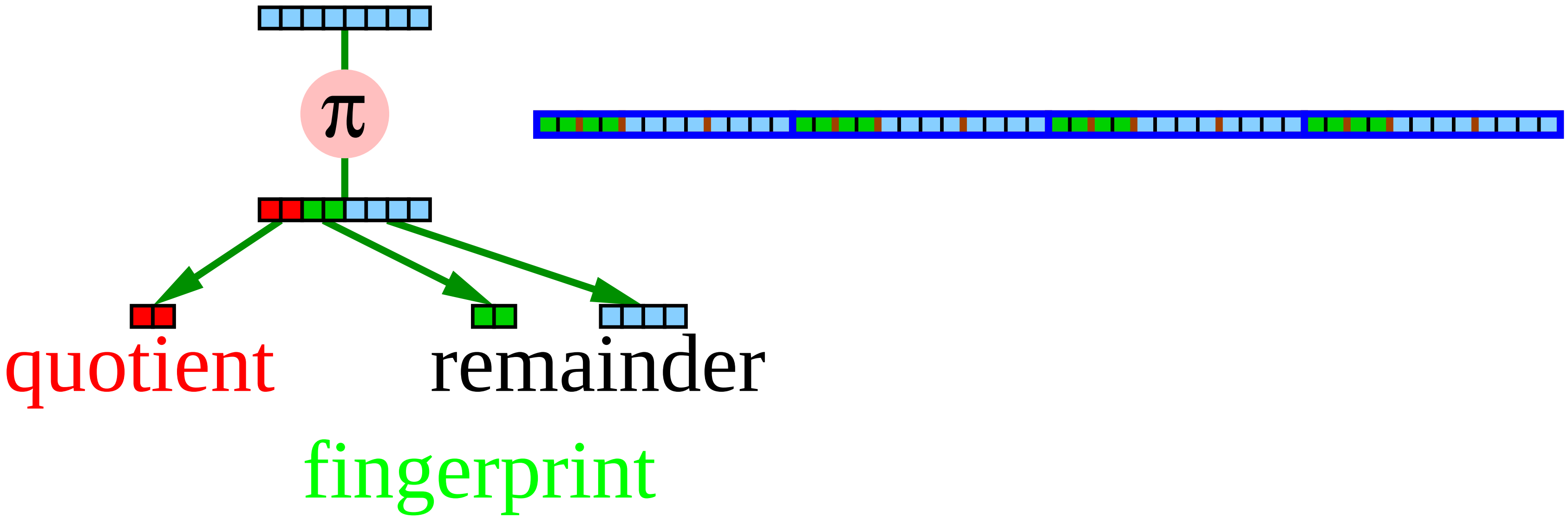
$$\pi_h : \mathbb{Z}_{u^2} \to \mathbb{Z}_{u^2} \text{ with } \pi_h(x,y) = (y, x + h(y) \bmod u) \quad .$$

$$\pi_h^{-1} : \mathbb{Z}_{u^2} \to \mathbb{Z}_{u^2} \text{ with } \pi_h^{-1}(y,z) = (z - h(y), y) \bmod u) \quad .$$

[32, 33, 34]: Chaining 4 Feistel permutations or linear∘Feistel∘Feistel∘linear is cryptographically safe if the hs are cryptographically safe.



Combining Succinctness and Fingerprints



Permutations allow us to use a part of the keys as fingerprint.

⇒No space overhead for fingerprints.

Adaptive Growing (and Shrinking)

Idea: use only little more space then necessary to store the elements, any time.

see separate slides

Possible Mini-Projects

- Concentrate on space-efficient Slick
- Concentrate on fast (unsuccessful) search for Slick
- Concentrate on fast build for Slick
- Concentrate on fast insert for Slick (SIMD instructions?)
- Concentrate on fast backyard cleaning for Slick
- Rudimentary succinct Slick?
- Rudimentary adaptively growing Slick?
- Cuckoo with large B and fast fingerprint-based search?
  Also Succinct?
- Bumbed Robin-Hood Hashing

- Bumped Block Hashing
- Linear Cuckoo Hashing
- …; your idea here

Summary Hashing

- Versatile data structure
- Often performance critical
- Various space-time-simplicity tradeoffs
- Shopping list (considered harmful?)
- Also relevant: Special cases and relaxation – Retrieval, perfect static hashing, approximate membership filters (AMQs aka Bloom filters)
- Still active area of research (SIMD; GPU, succinct, adaptive growing, special cases. . .)

6 Minimum Spanning Trees

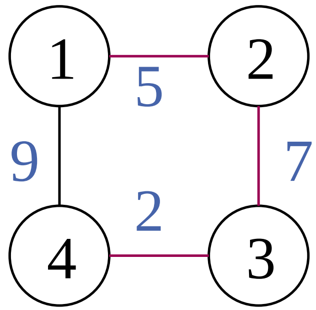
undirected Graph $G = (V, E)$.

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

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

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

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 segementation $\longrightarrow$ [Diss. Jan Wassenberg]

Anyway: almost ideal "fruit fly" problem

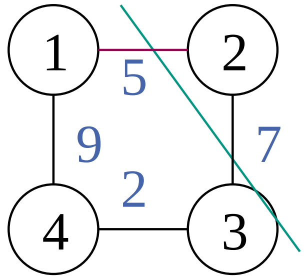
Selecting and Discarding MST Edges

The Cut Property

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

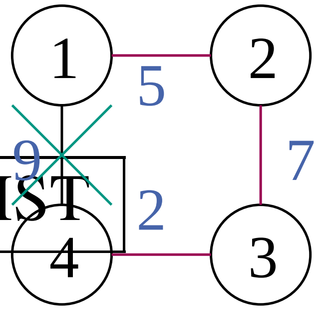
$C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$

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

$T := \emptyset$

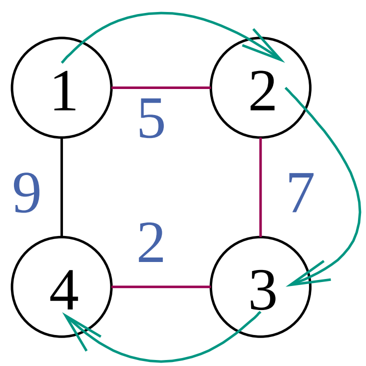
$S := \{s\}$ for arbitrary start node s

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 jpMST(V, E, w) : Set **of** Edge

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

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

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

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

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

  $u :=$ q.deleteMin() // new node for S

  dist$[u]$:= 0

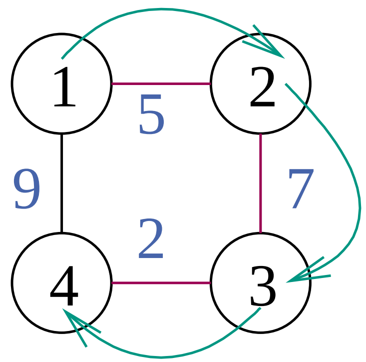
  foreach $(u, v) \in E$ **do**

    if $c((u,v)) <$ dist$[v]$ **then**

      dist$[v]$:= $c((u,v))$; pred$[v]$:= (u,v)

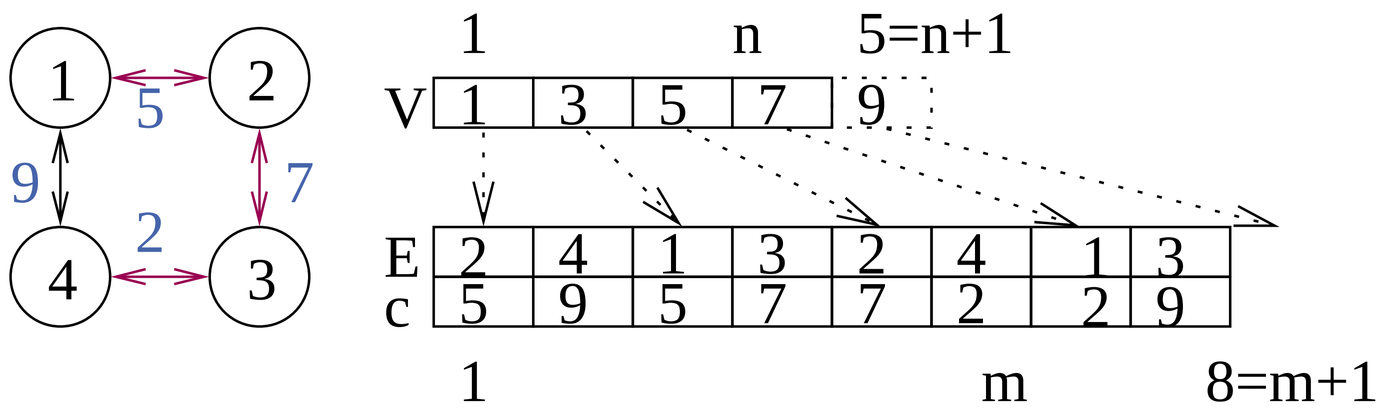
      if $v \in q$ **then** q.decreaseKey(v) **else** q.insert(v)

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



Graph Representation for Jarník-Prim

We need node $\rightarrow$ incident edges



+ 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.

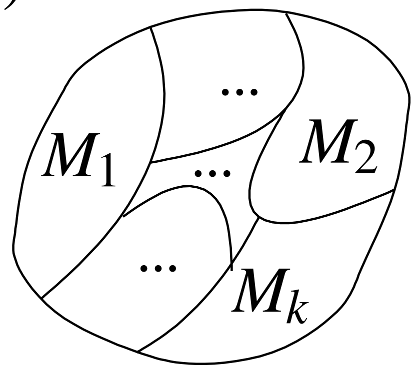
But analysis is still partly open!

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) $M_1, \ldots, M_k$ mit



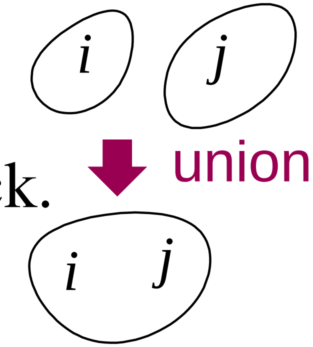
$$M_1 \cup \cdots \cup M_k = 1..n,$$

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

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

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

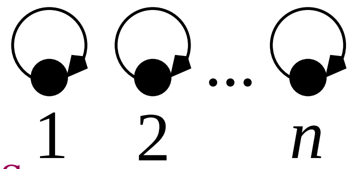
join the blocks containing i and j to a single block.



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

return a unique identifier for the block containing i.

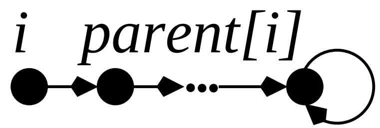
Union-Find Datenstruktur – Erste Version



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

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

  invariant parent-refs lead to unique Partition-Reps



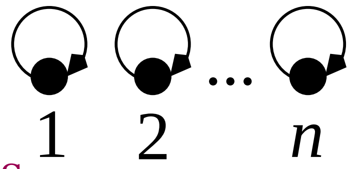
  Function find($i : 1..n$) : $1..n$

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

    else return find(parent$[i]$)

Union-Find Datenstruktur – Erste Version

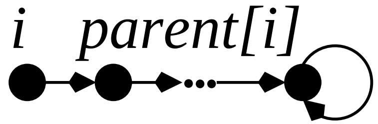
Class UnionFind($n : \mathbb{N}$)



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

invariant parent-refs lead to unique Partition-Reps

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

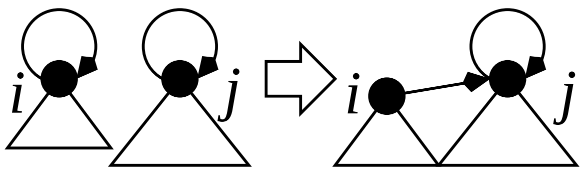


if parent$[i] = i$ **then return** i

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

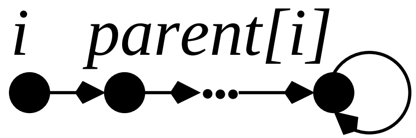
if find(i) $\neq$ find(j) **then** link(find(i), find(j))

Union-Find Datenstruktur – Erste Version

Analyse:

+: union braucht konstante Zeit

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



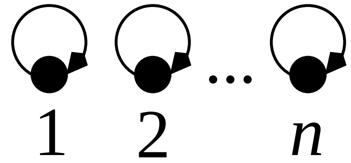
zu langsam.

Idee: find-Pfade kurz halten

Pfadkompression

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

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



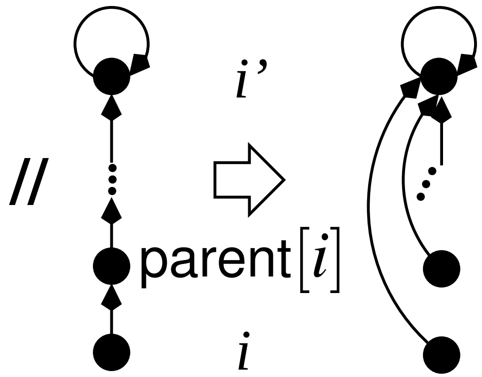
  Function find($i : 1..n$) : $1..n$

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

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

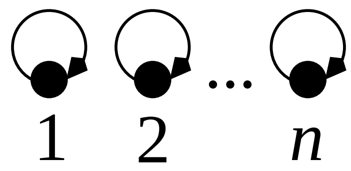
      parent$[i]$:= i'

      return i'



Union by Rank

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



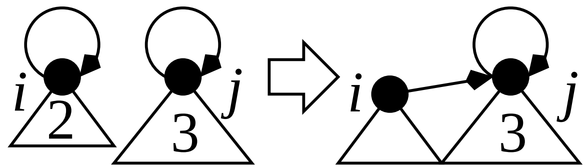
parent$=\langle 1,2,\ldots,n\rangle$: **Array** $[1..n]$ **of** $1..n$

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

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

assert i and j are representatives of different blocks

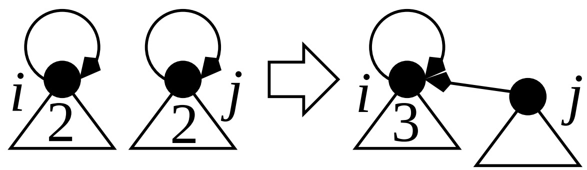
if rank$[i] <$ rank$[j]$ **then** parent$[i] := j$



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

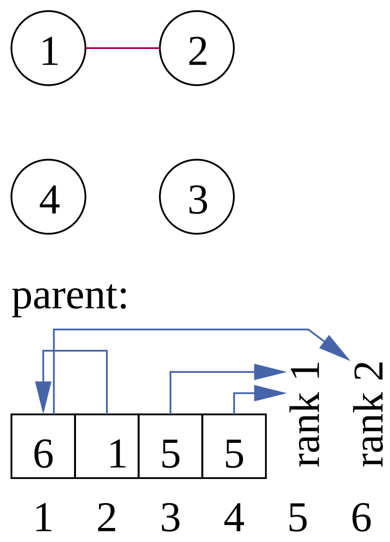
Function find($i : 1..n$) : $1..n$

if parent$[i] > n$ **then return** i

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

parent$[i]$:= i'

return i'



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

assert i and j are leaders of different subsets

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

else if parent$[i] >$ parent$[j]$ **then** parent$[j]$:= i

else parent$[j]$:= i; parent$[i]$++ // next generation

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

Kruskal Using Union Find

T : UnionFind(n)
sort E in ascending order of weight
kruskal(E)

Procedure kruskal(E)
  foreach $(u,v) \in E$ **do**
    $u' := T.\mathsf{find}(u)$
    $v' := T.\mathsf{find}(v)$
    if $u' \neq v'$ **then**
      output (u,v)
      $T.\mathsf{link}(u',v')$

Graph Representation for Kruskal

Just an edge sequence (array) !

+ very fast (cache efficient)

+ Edges are stored only once

$\rightsquigarrow$ more compact than adjacency array

Analysis

$\mathcal{O}(\mathsf{sort}(m) + m\alpha(m,n)) = \mathcal{O}(m\log m)$ 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.

6.1 Filtering by Sampling Rather Than Sorting

R:= random sample of r edges from E

F :=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 MST(($L \cup F$))

6.1.1 Analysis

[Chan 98, KKK 95]

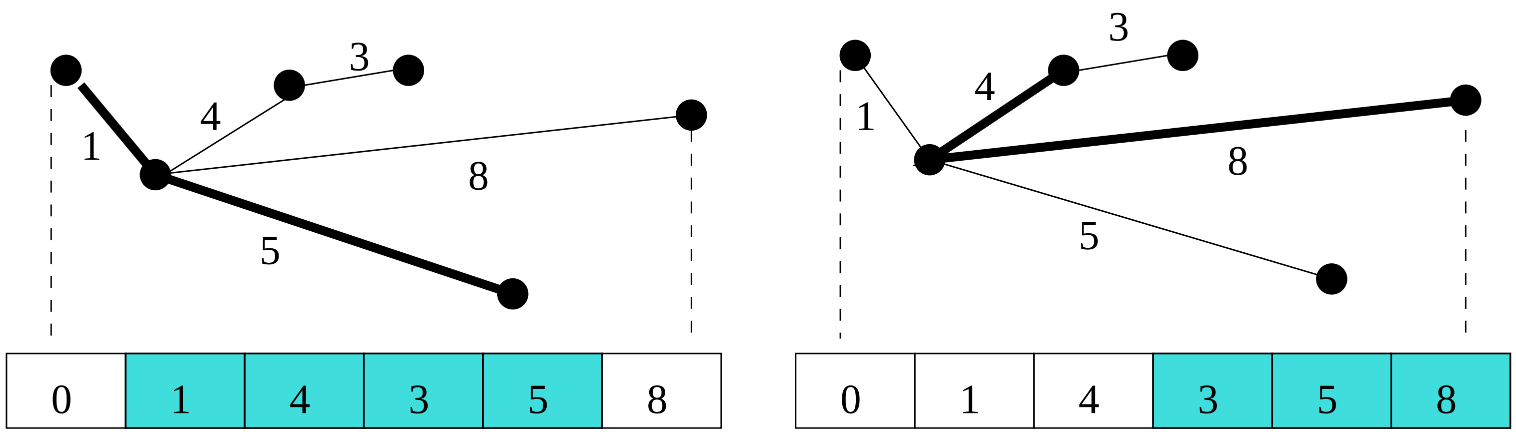
Observation: $e \in L$ only if $e \in \mathrm{MST}(R \cup \{e\})$.

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

Lemma 1. $\mathrm{E}[|L \cup F|] \leq \frac{mn}{r}$

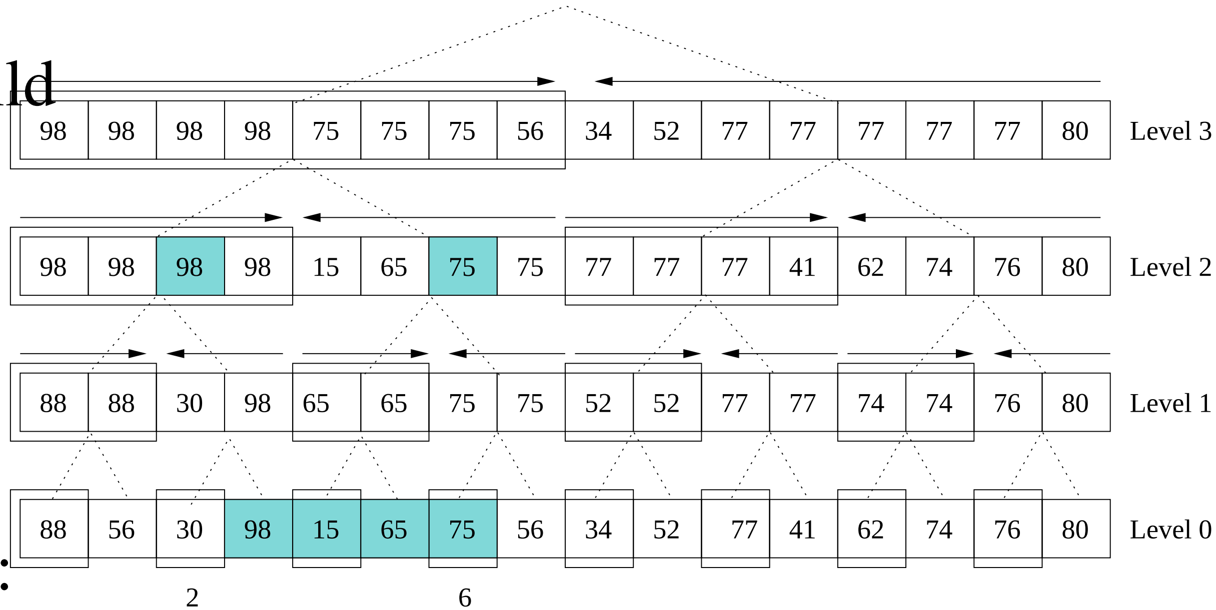
MST Verification by Interval Maxima

- 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) $= \max\{w_j : u < j \leq v\}$.



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 $\max(\mathrm{PreSuf}[\ell][i], \mathrm{PreSuf}[\ell][j])$.
- Example: $2 \oplus 6 = 010 \oplus 110 = 100 \Rightarrow \ell = 2$

A Simple Filter Based Algorithm

Choose $r = \sqrt{mn}$.

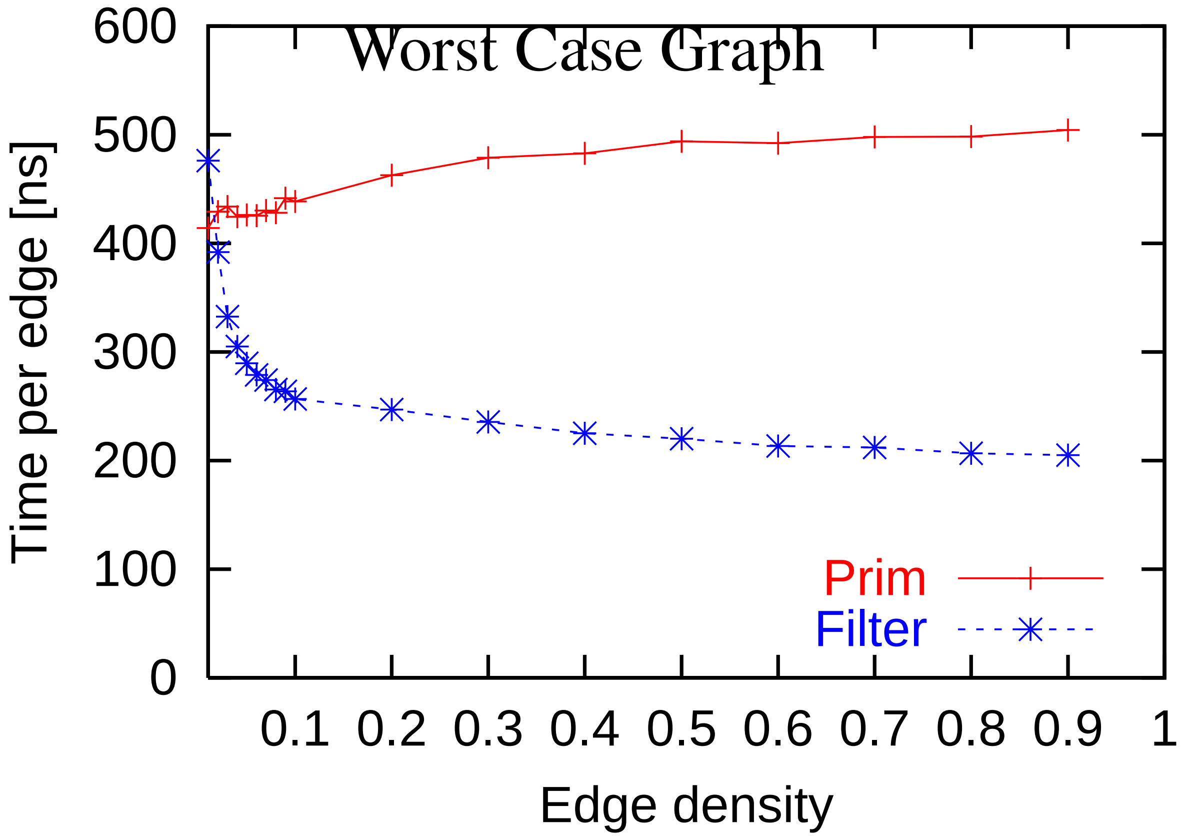
We get expected time

$$\begin{aligned} T &= T_{\text{Prim}}(\sqrt{mn}) + \mathcal{O}(n\log n + m) + T_{\text{Prim}}(\frac{mn}{\sqrt{mn}}) \\ &= T_{\text{Prim}}(\sqrt{mn}) + \mathcal{O}(n\log n + m) \\ &= \mathcal{O}(n\log n + \underbrace{\sqrt{mn}}_{o(n\log n + m)}) + \mathcal{O}(n\log n + m) \end{aligned}$$

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

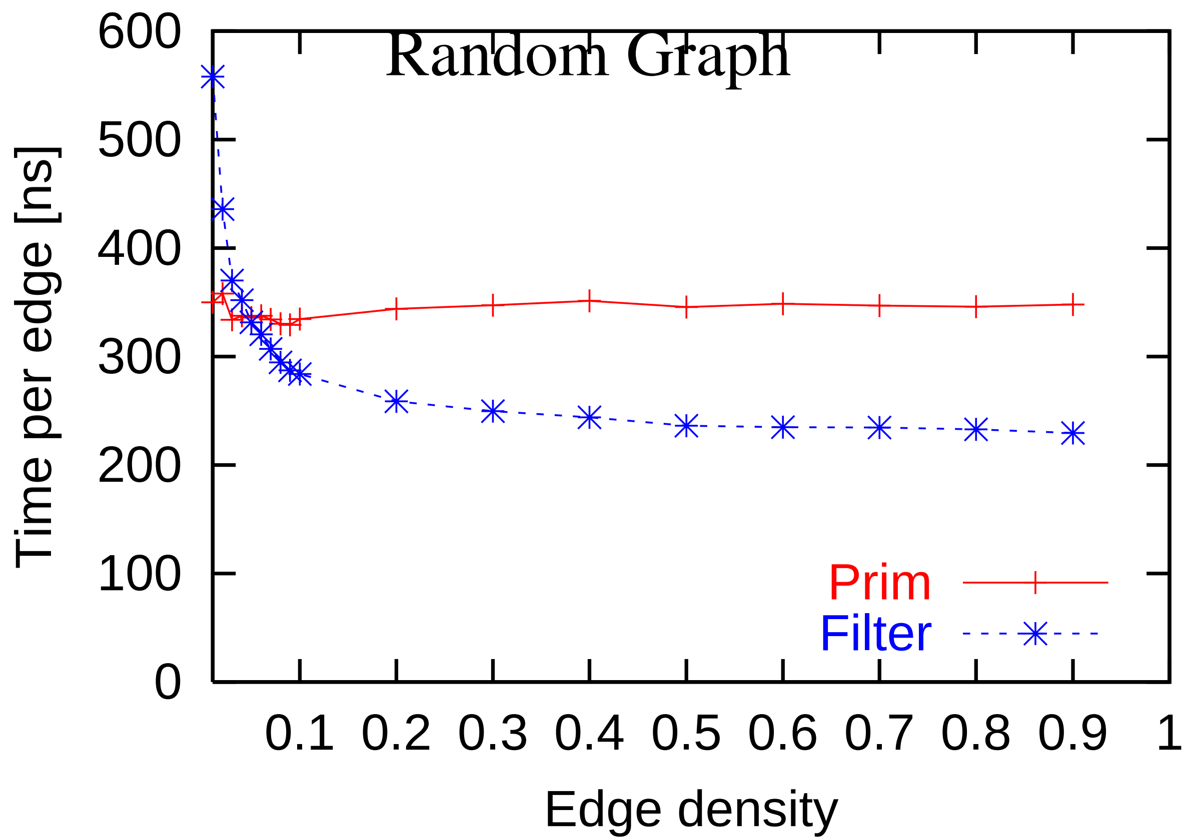
Results

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

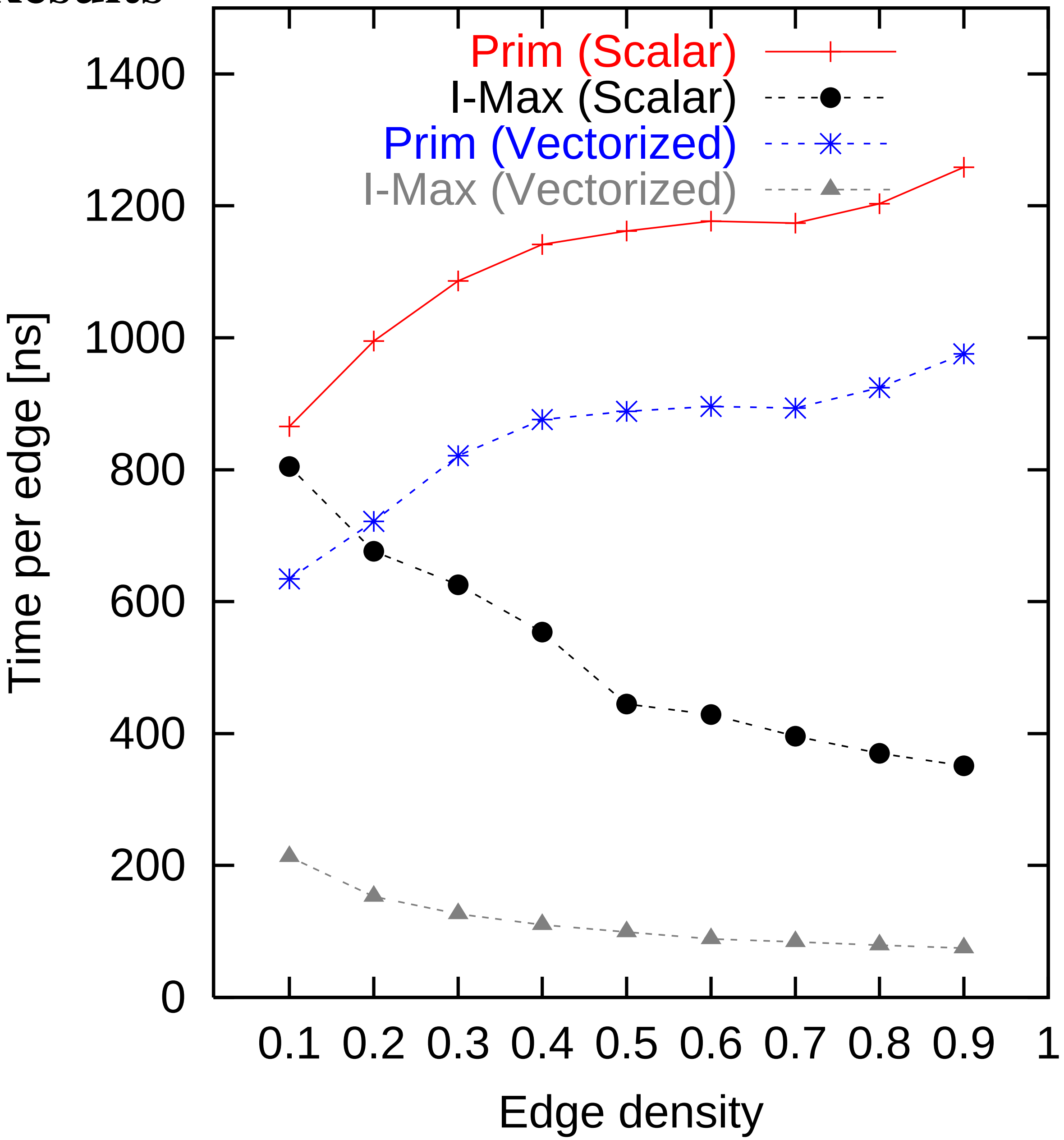


Results

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



Results



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

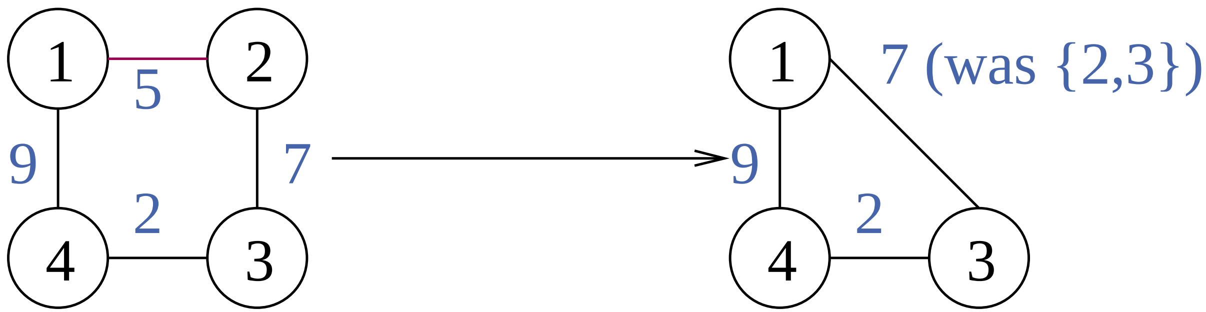
Edge Contraction

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

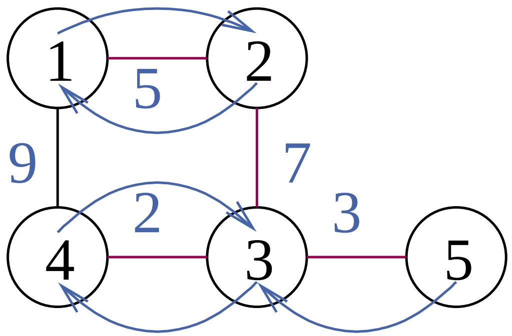


Boruvka's Node Reduction Algorithm

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



6.2 Simpler and Faster Node Reduction

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

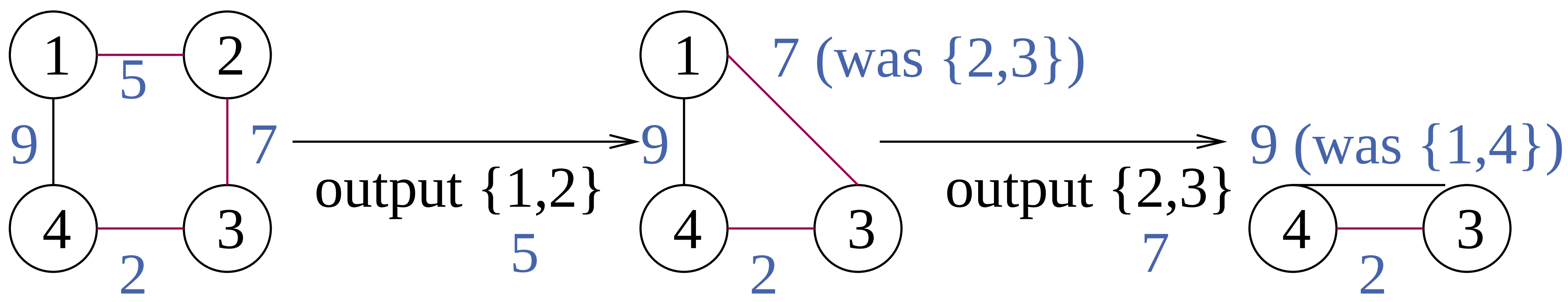
pick a random node v

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

contract (u, v)

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

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



6.3 Randomized Linear Time Algorithm

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

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

6.4 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'
  $T := \mathsf{MST}(T \cup E')$ // for any internal MST alg.

Corollary: we can do it with linear expected internal work

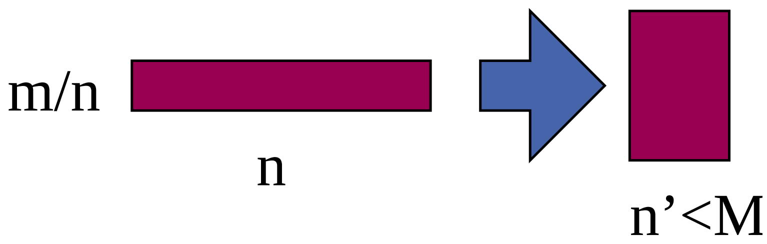
Disadvantages to Kruskal:
Slower in practice
Smaller max. n

General External MST

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

  perform some node reduction

use semi-external Kruskal



Theory: $\mathcal{O}(\mathsf{sort}(m))$ expected I/Os by externalizing the linear time algorithm.
(i.e., node reduction + edge reduction)

External Implementation I: Sweeping

π : random permutation $V \to V$

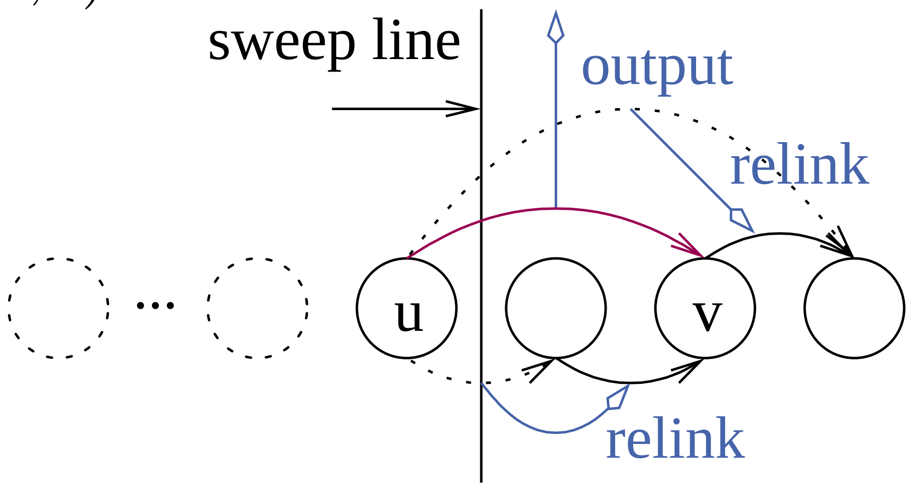
sort edges (u,v) by $\max(\pi(u),\pi(v))$

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**

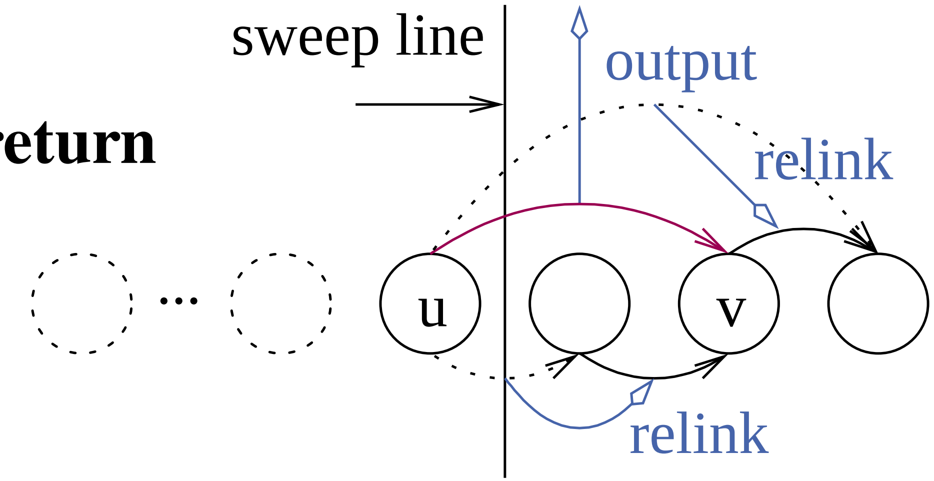
    if current$= M+1$ **then return**

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

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

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

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

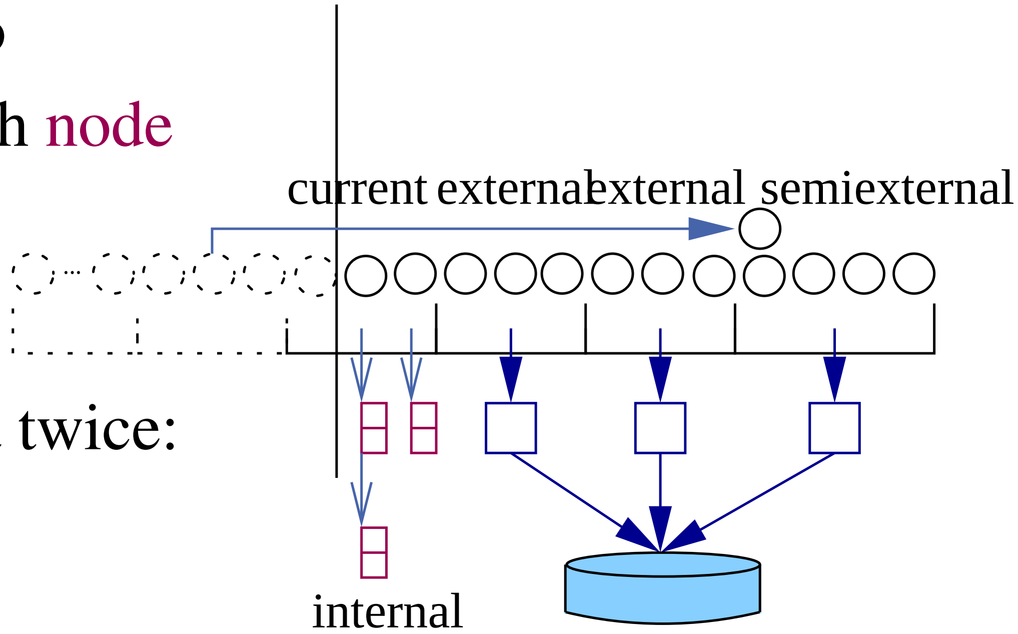


$\approx \mathsf{sort}(10m\ln\frac{n}{M})$ I/Os with opt. priority queues [Sanders 00]

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

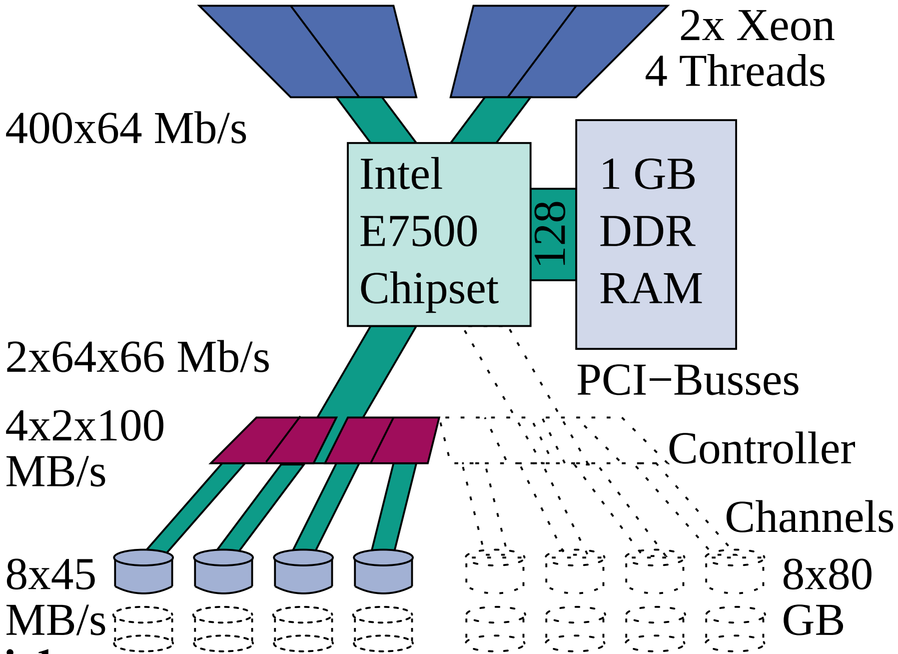
New external bucket: scan and put in internal buckets

Large degree nodes: move to semiexternal bucket

Experiments

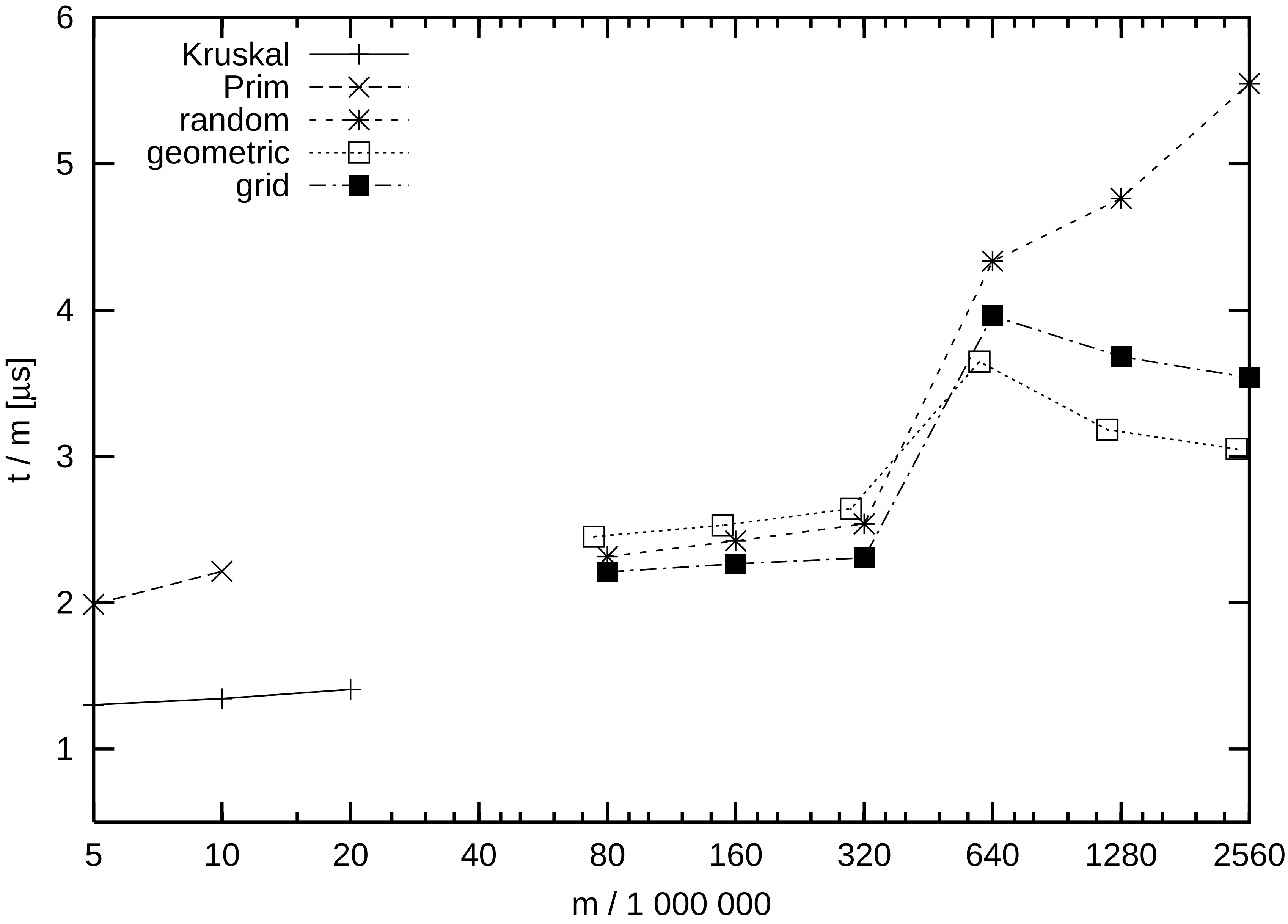
Instances from "classical" MST study [Moret Shapiro 1994]

- sparse random graphs
- random geometric graphs
- grids
  $\mathcal{O}(\mathrm{sort}(m))$ I/Os
  for planar graphs by
  removing parallel edges!

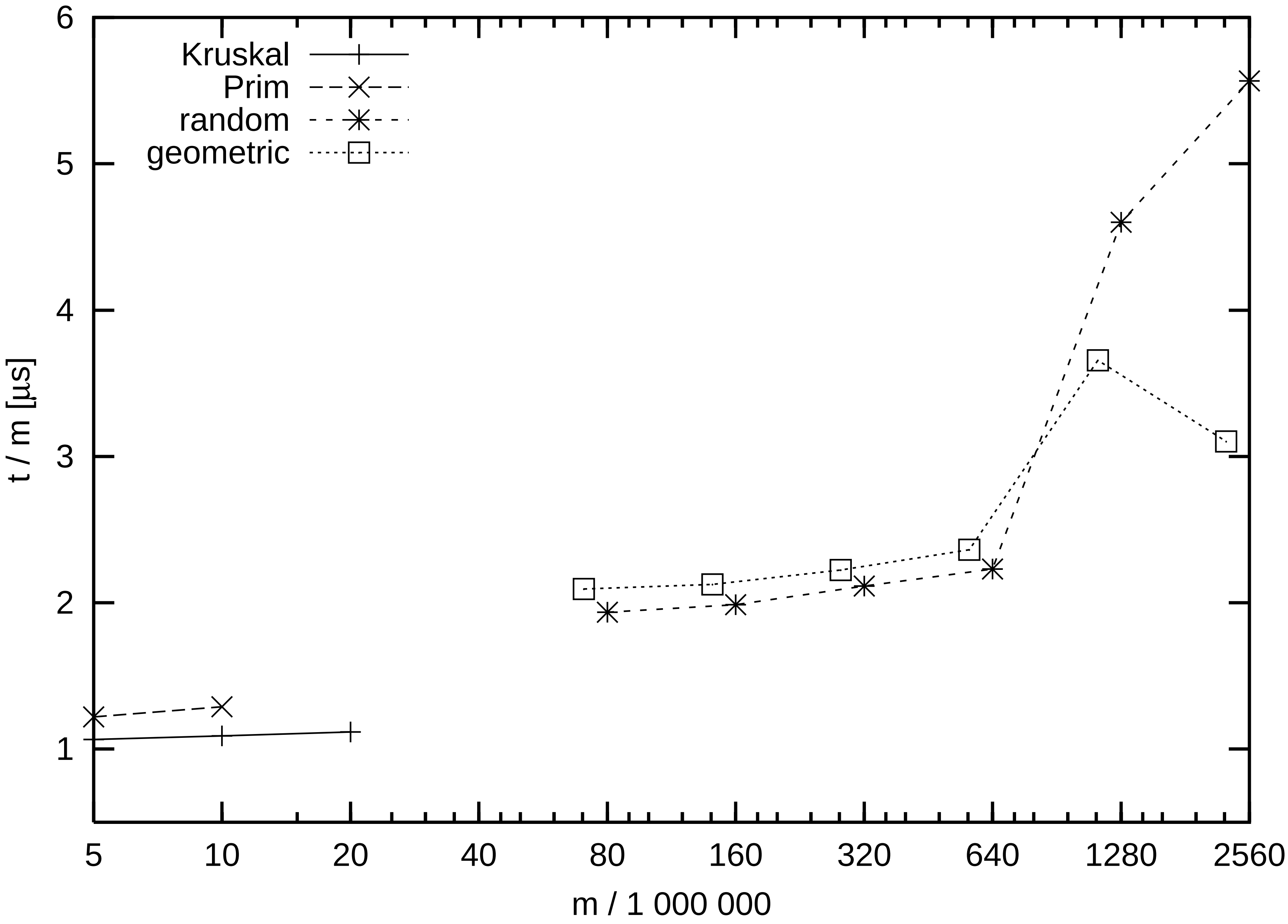


Other instances are rather dense
or designed to fool specific algorithms.

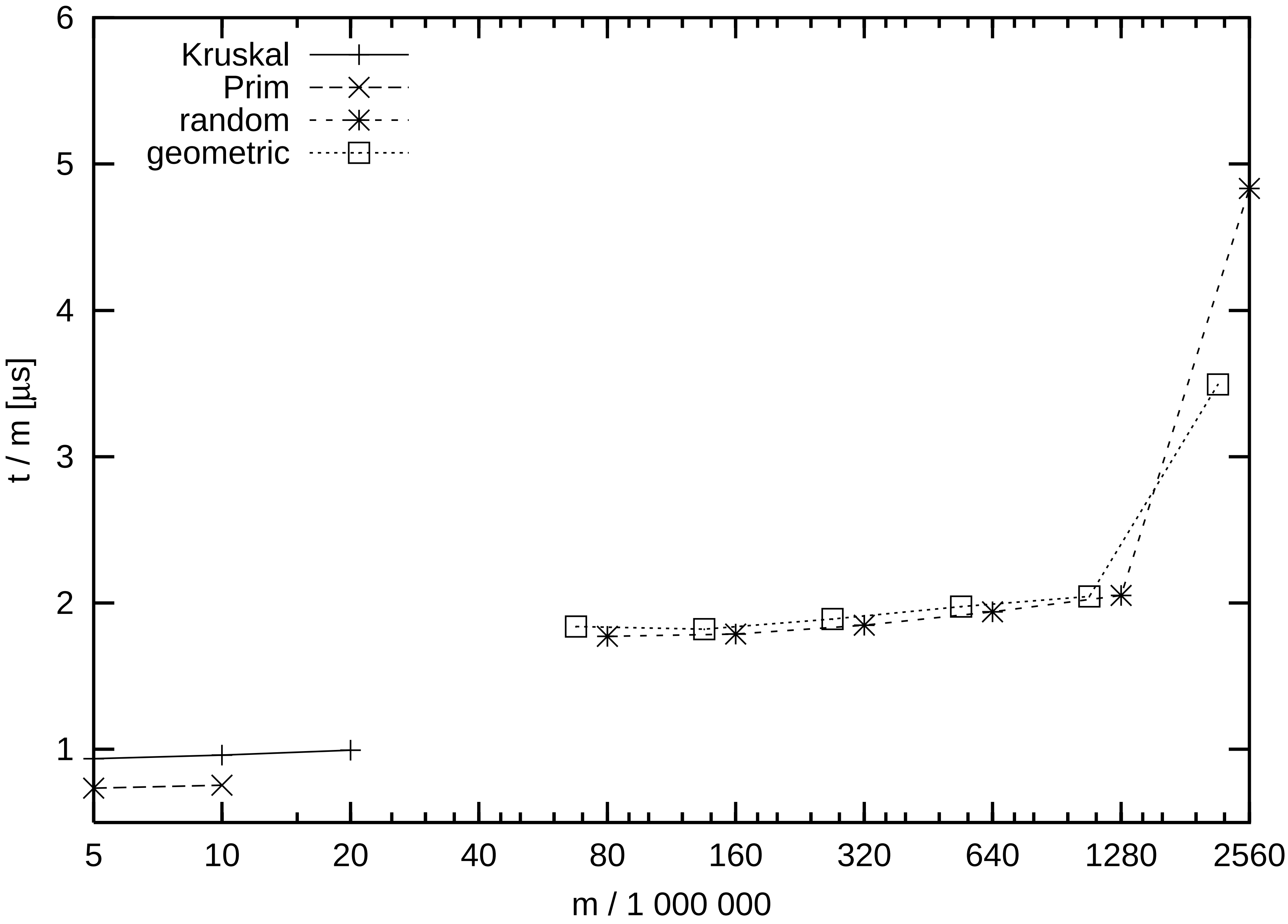
$m \approx 2n$


Kruskal
Prim
random
geometric
grid
t / m [µs]
1
2
3
4
5
6
5
10
20
40
80
160
320
640
1280
2560
m / 1 000 000

$m \approx 4n$


Kruskal
Prim
random
geometric
t / m [µs]
6
5
4
3
2
1
5
10
20
40
80
160
320
640
1280
2560
m / 1 000 000

$m \approx 8n$


Kruskal
Prim
random
geometric
t / m [µs]
m / 1 000 000
6
5
4
3
2
1
5
10
20
40
80
160
320
640
1280
2560

External MST Summary

- Edge reduction helps for very dense, “hard” graphs
- A fast and simple node reduction algorithm
  $\leadsto$ $4\times$ 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 (at that time) 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 did this
- A practical linear time Algorithm
- 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: $\mathcal{O}\left(m\Lambda\log(n^2/m)\log U\right)$ binary blocking flow-algorithm mit $\Lambda = \min\{m^{1/2}, n^{2/3}\}$ [Goldberg-Rao-97].

Problem: best case $\approx$ worst case

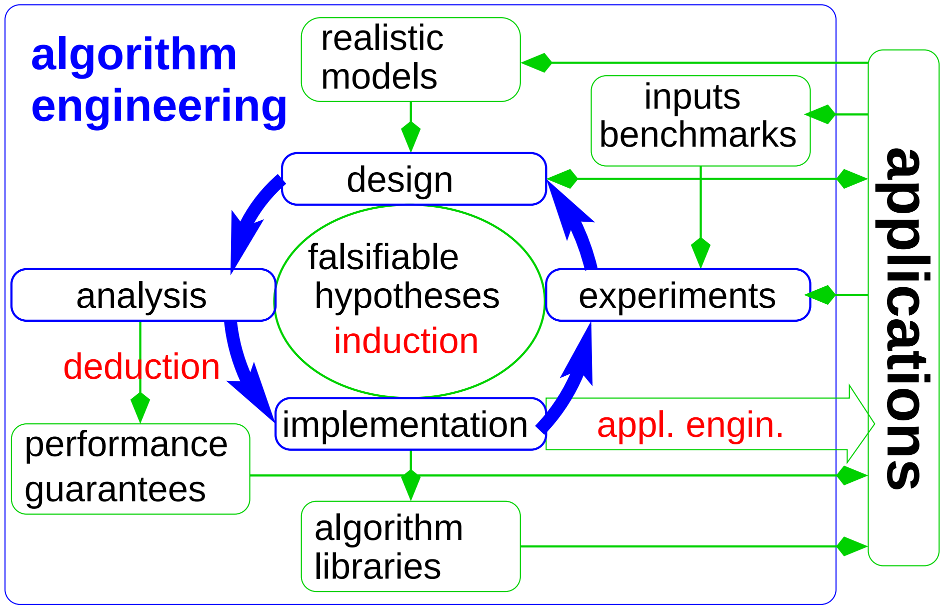
[Hagerup Sanders Träff WAE 98]:

- 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, $|V|$, $|E|$, $\ldots$ or problem instances from applications)

Usually there is an abundance of data (was: $\neq$ 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

$\leadsto$ new input instances

Presenting Data from Experiments in Algorithmics

Restrictions

- black and white $\leadsto$ easy and cheap printing
  (Now: Few colors, distinguishable on different beamers or screen, ideally readable when printed b/w)
- 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]

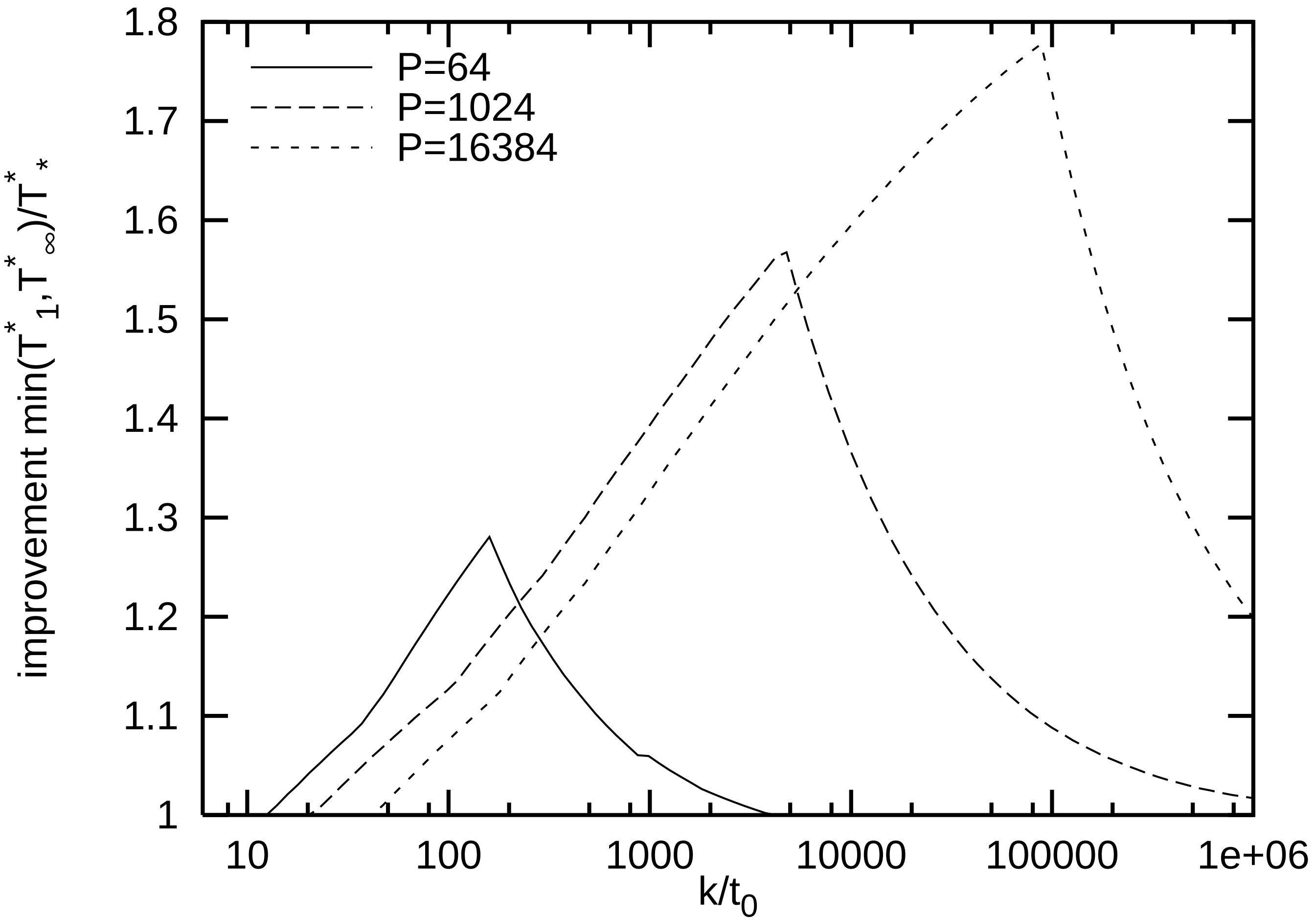
Curves in main paper, tables in appendix?

2D Figures

default: $x =$ input size, $y = f$(execution time)

x Axis

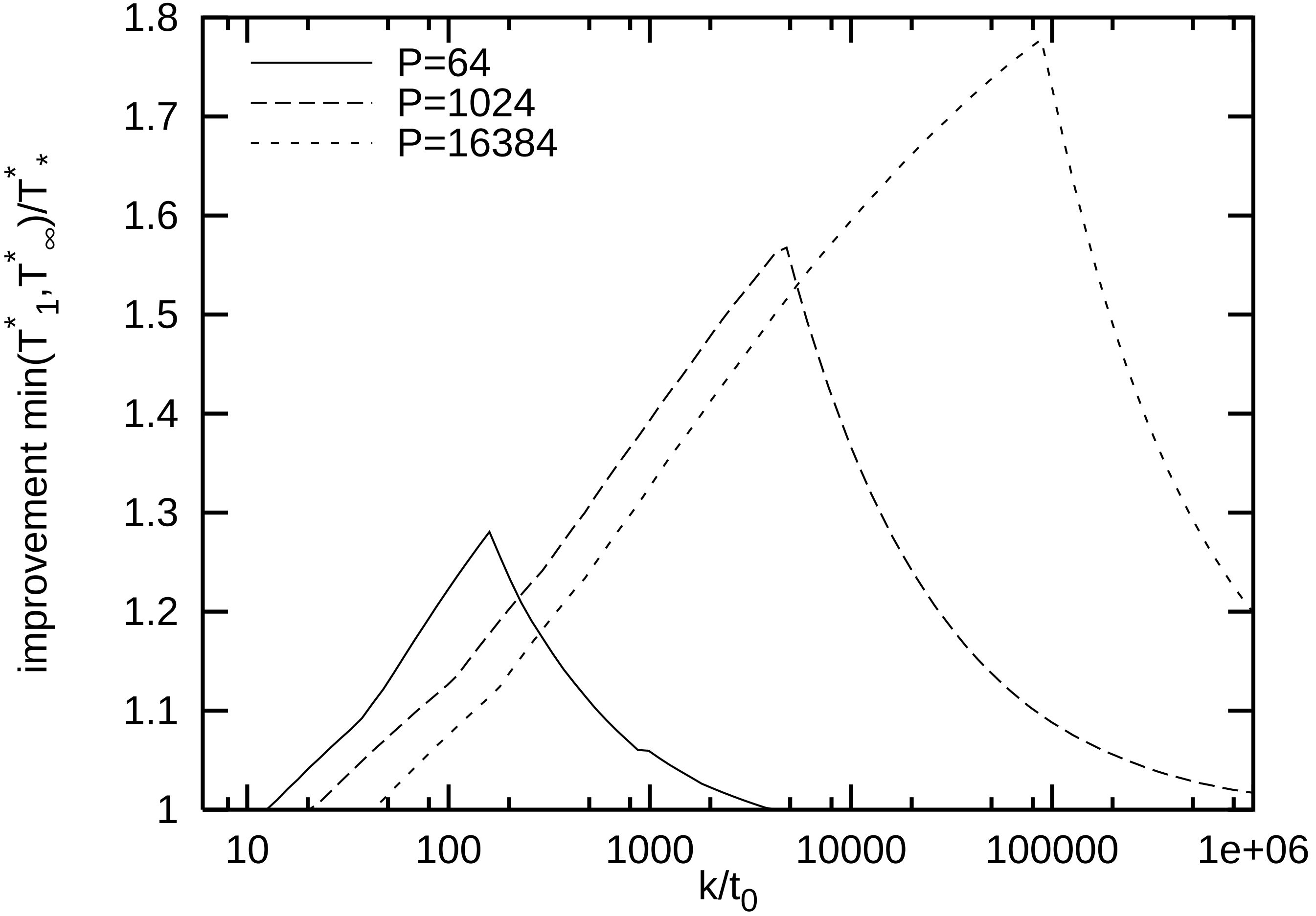
Choose unit to eliminate a parameter?



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

x Axis

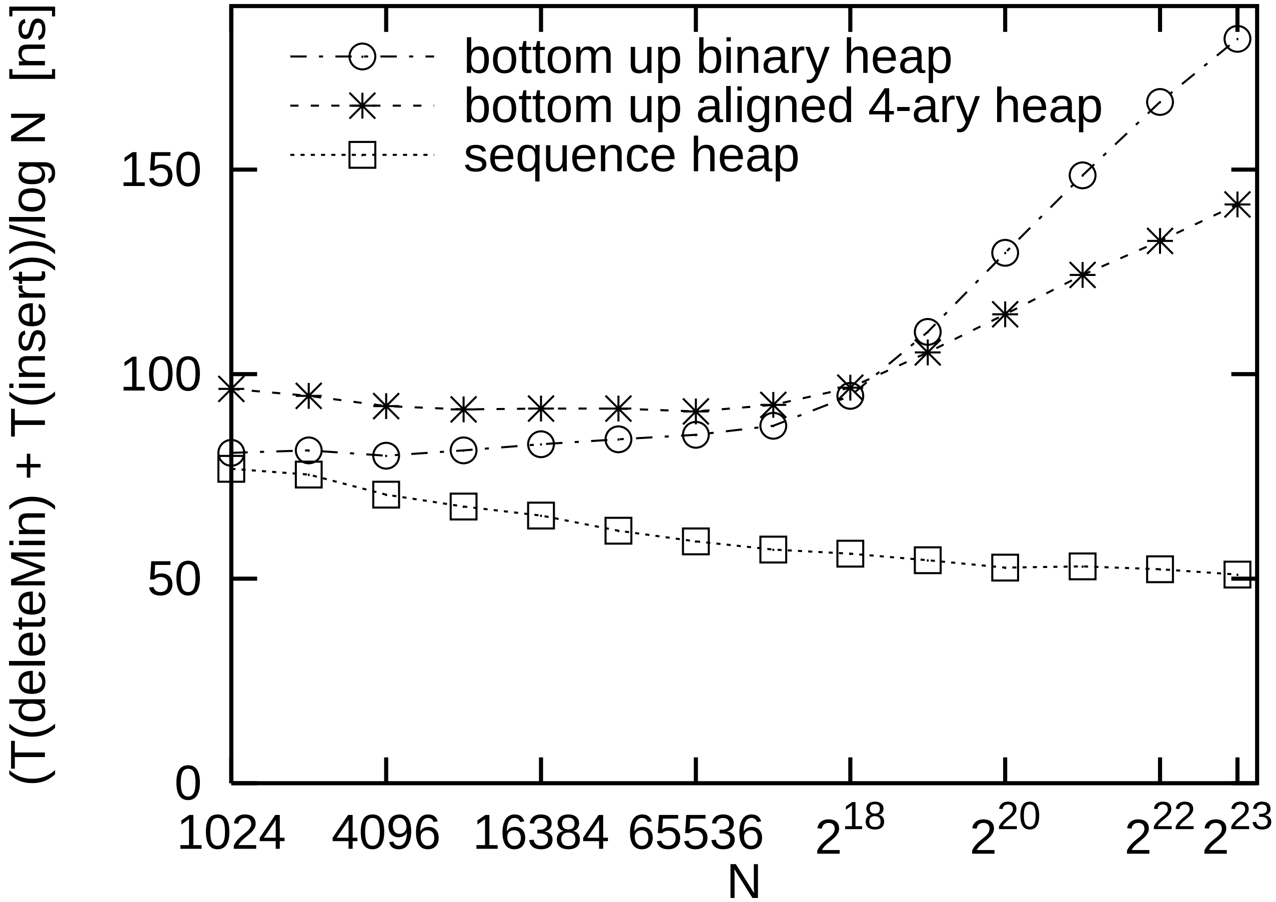
logarithmic scale?



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

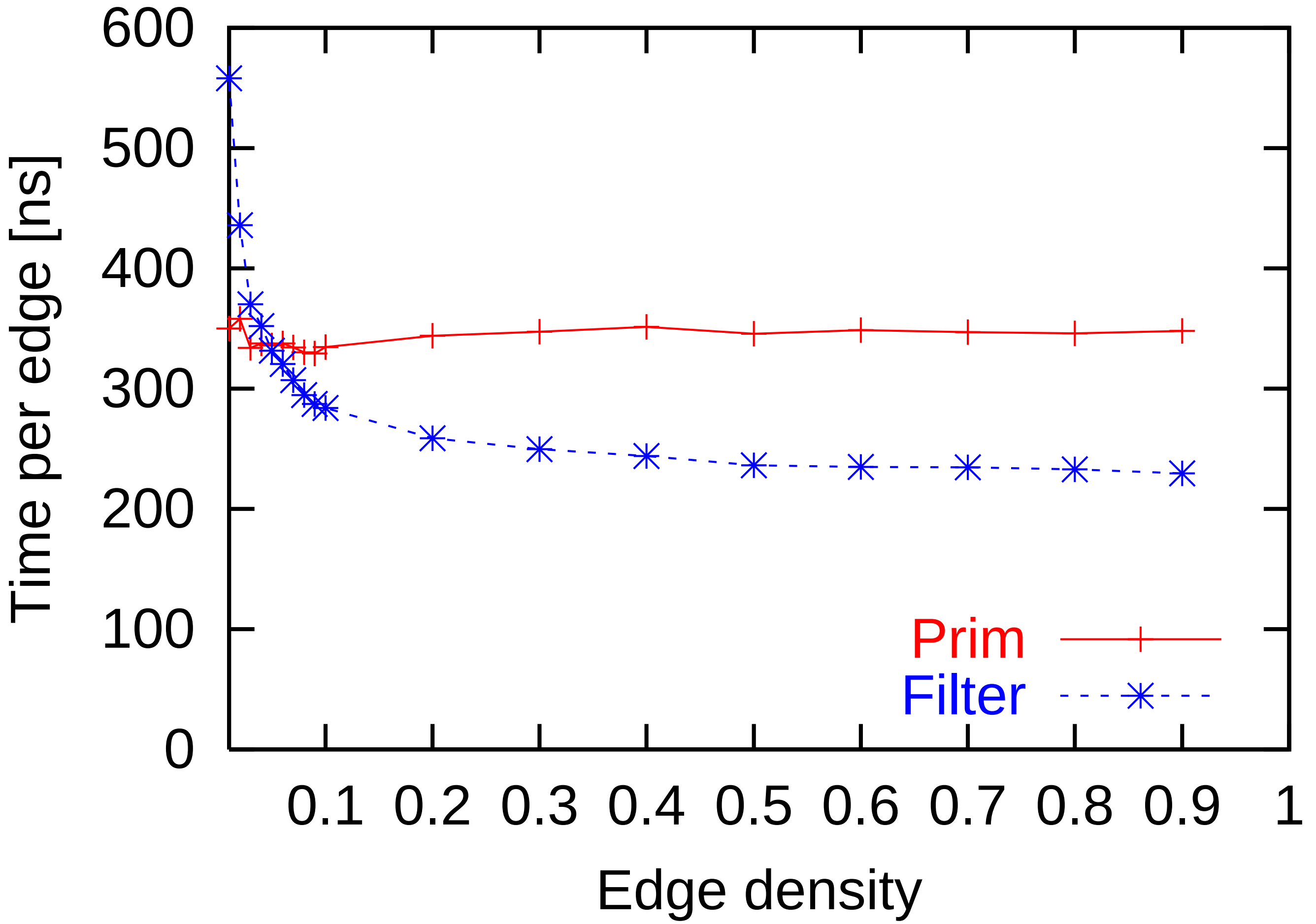
```
set xlabel "N"
set ylabel "(time per operation)/log N  [ns]"
set xtics (256, 1024, 4096, 16384, 65536, "2^{18}" 262144
set size 0.66, 0.33
set logscale x 2
set data style linespoints
set key left
set terminal postscript portrait enhanced 10
set output "r10000timenew.eps"
plot [1024:10000000][0:220]\
 "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
```

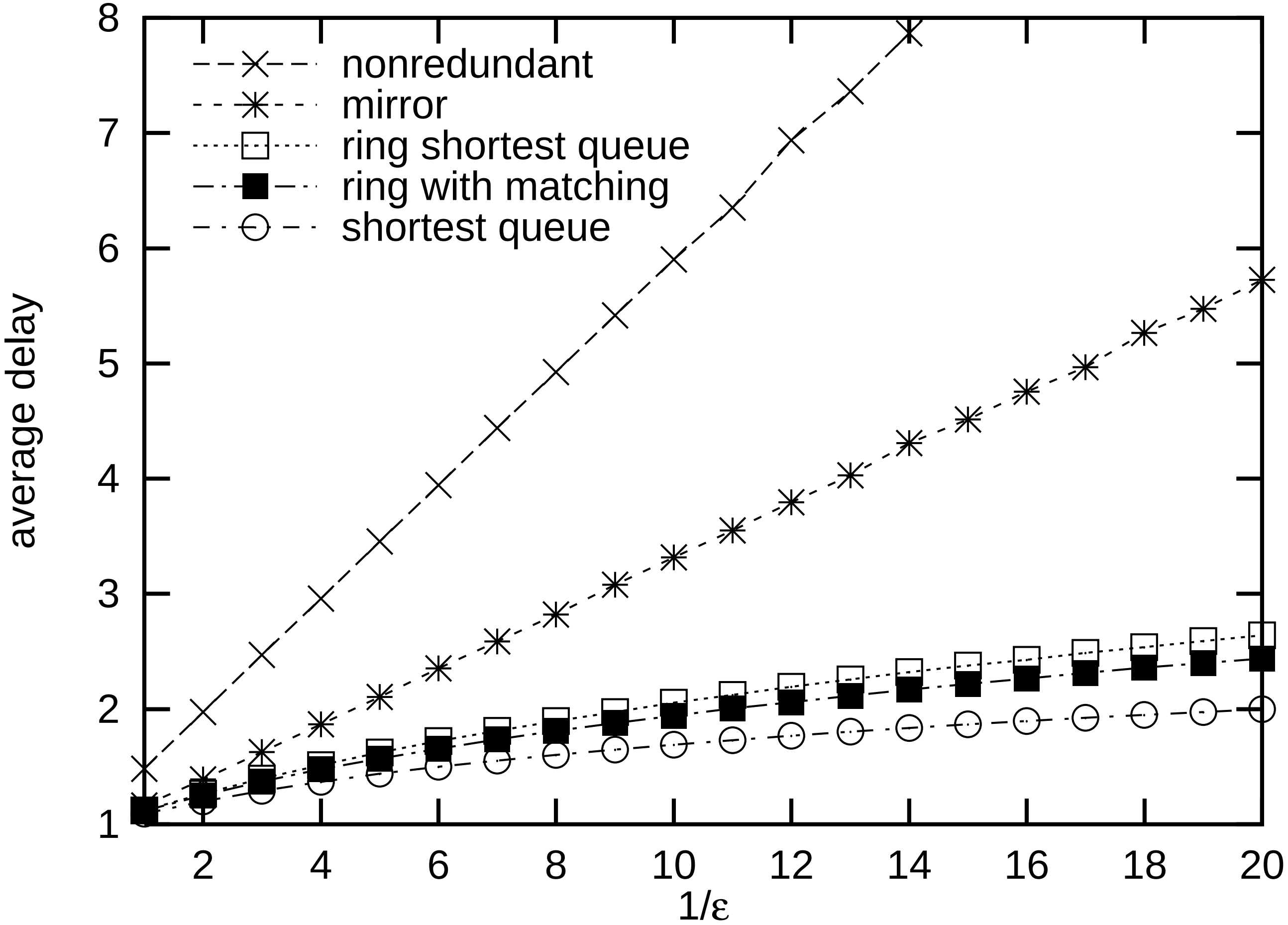
x Axis

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



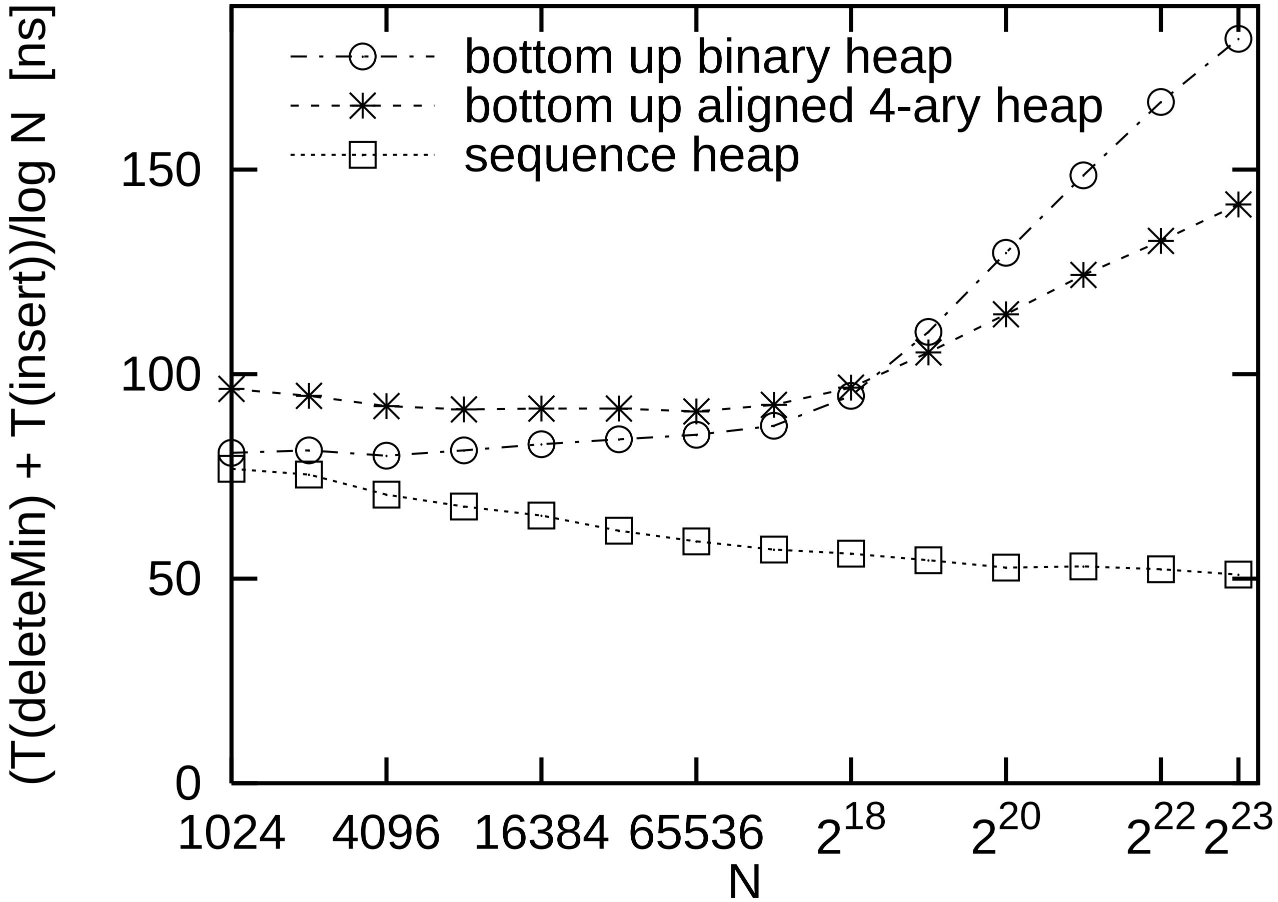
x Axis

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



y Axis

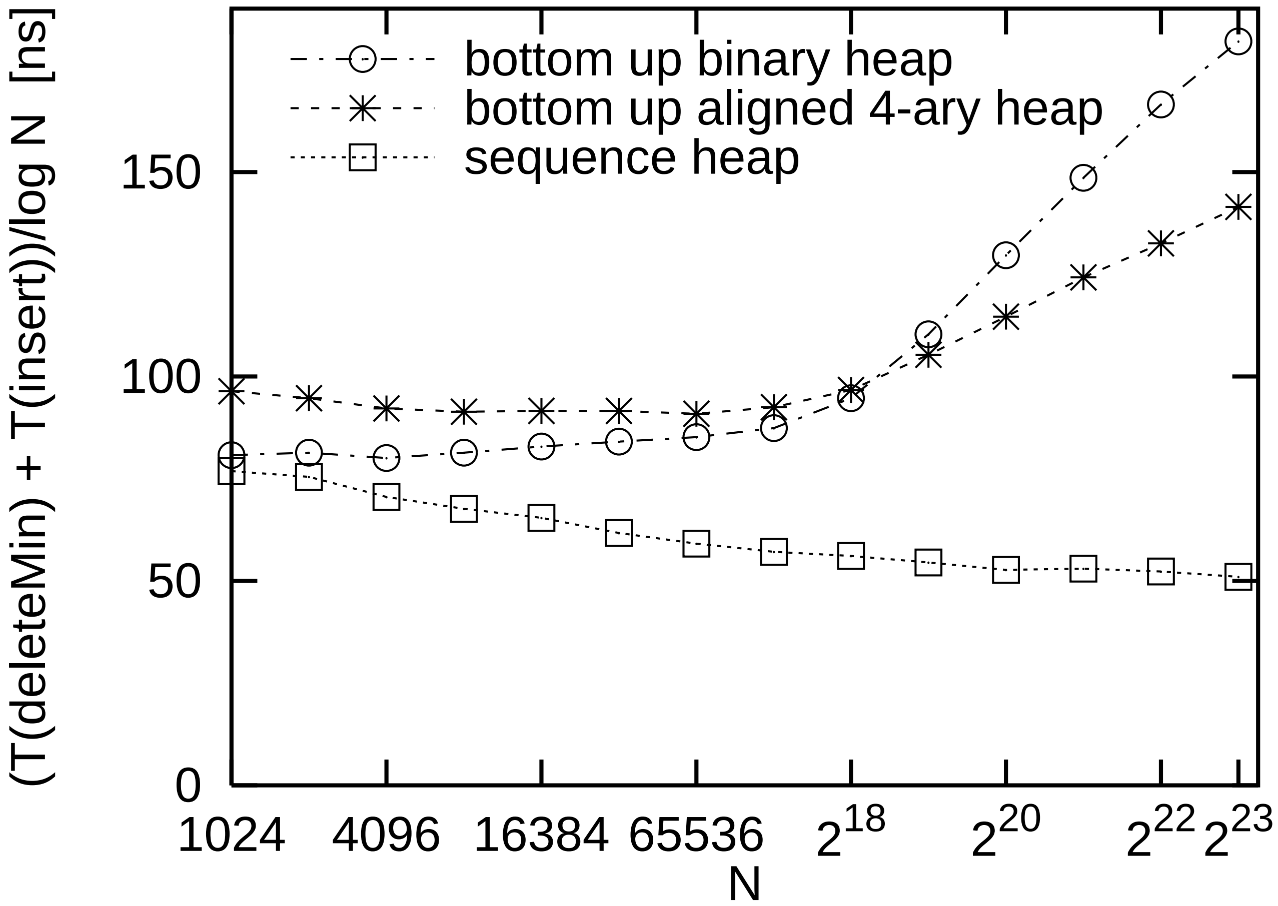
Avoid log scale ! scale such that theory gives ≈ horizontal lines



but give easy interpretation of the scaling function

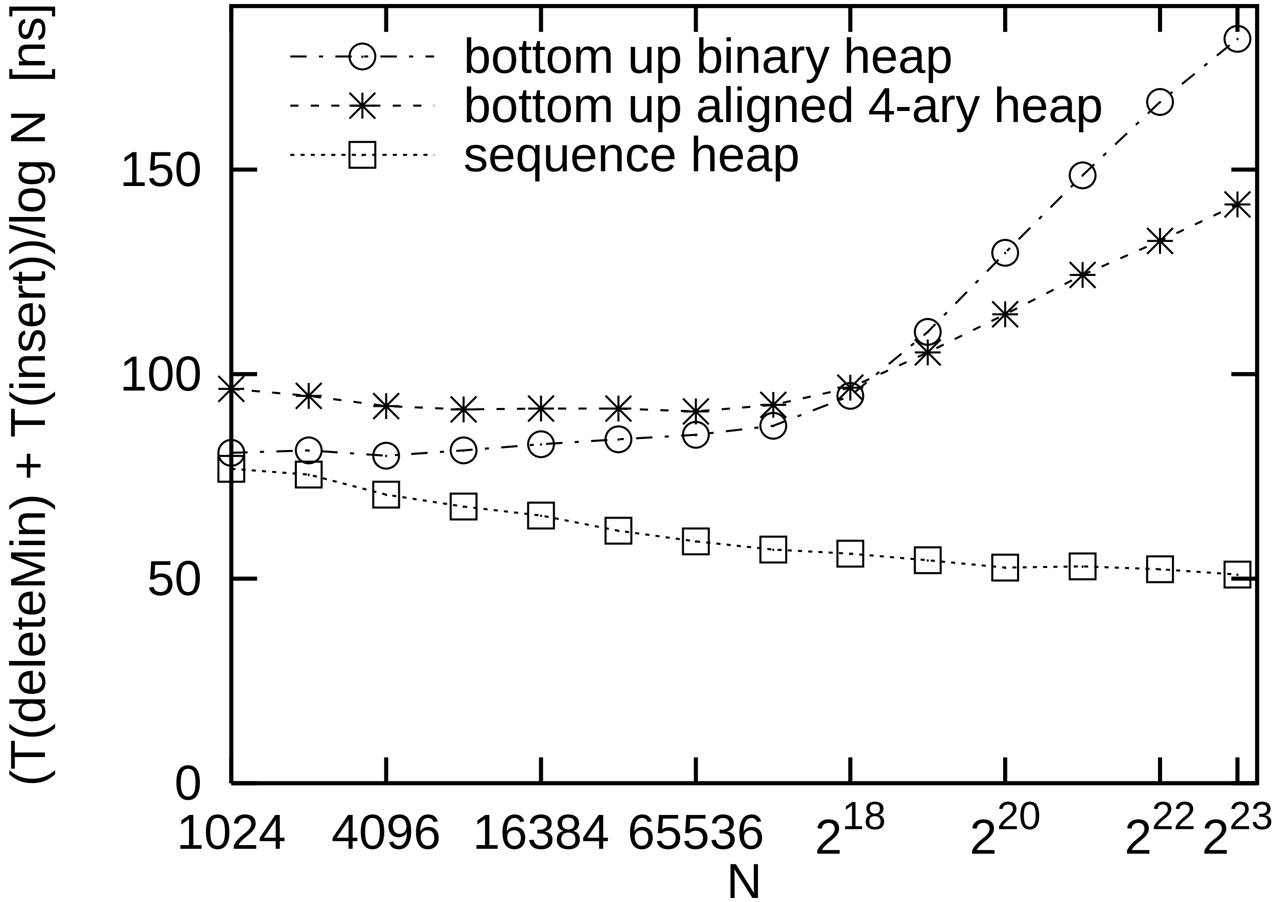
y Axis

give units



y Axis

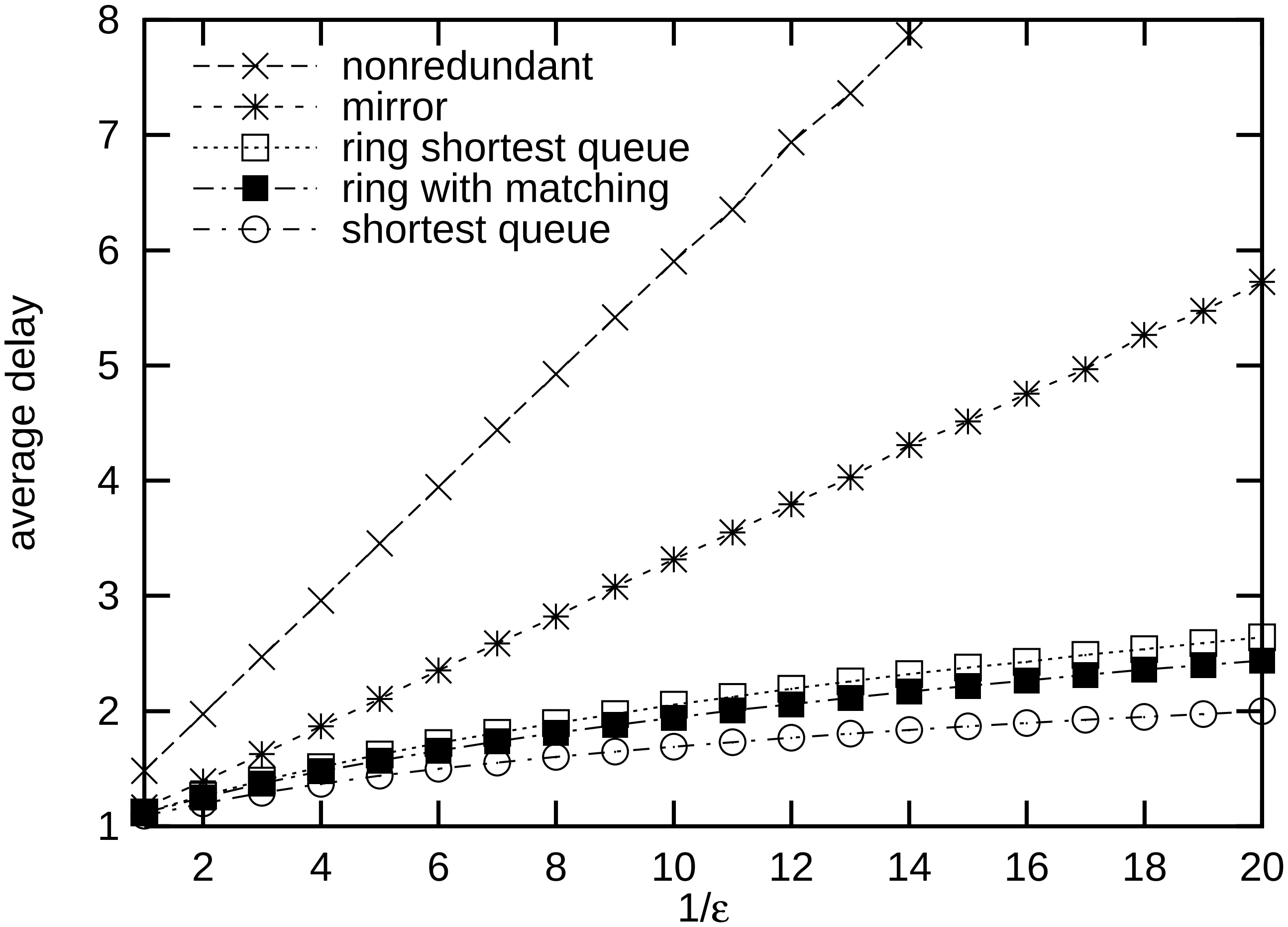
start from 0 if this does not waste too much space



you may assume readers to be out of Kindergarten

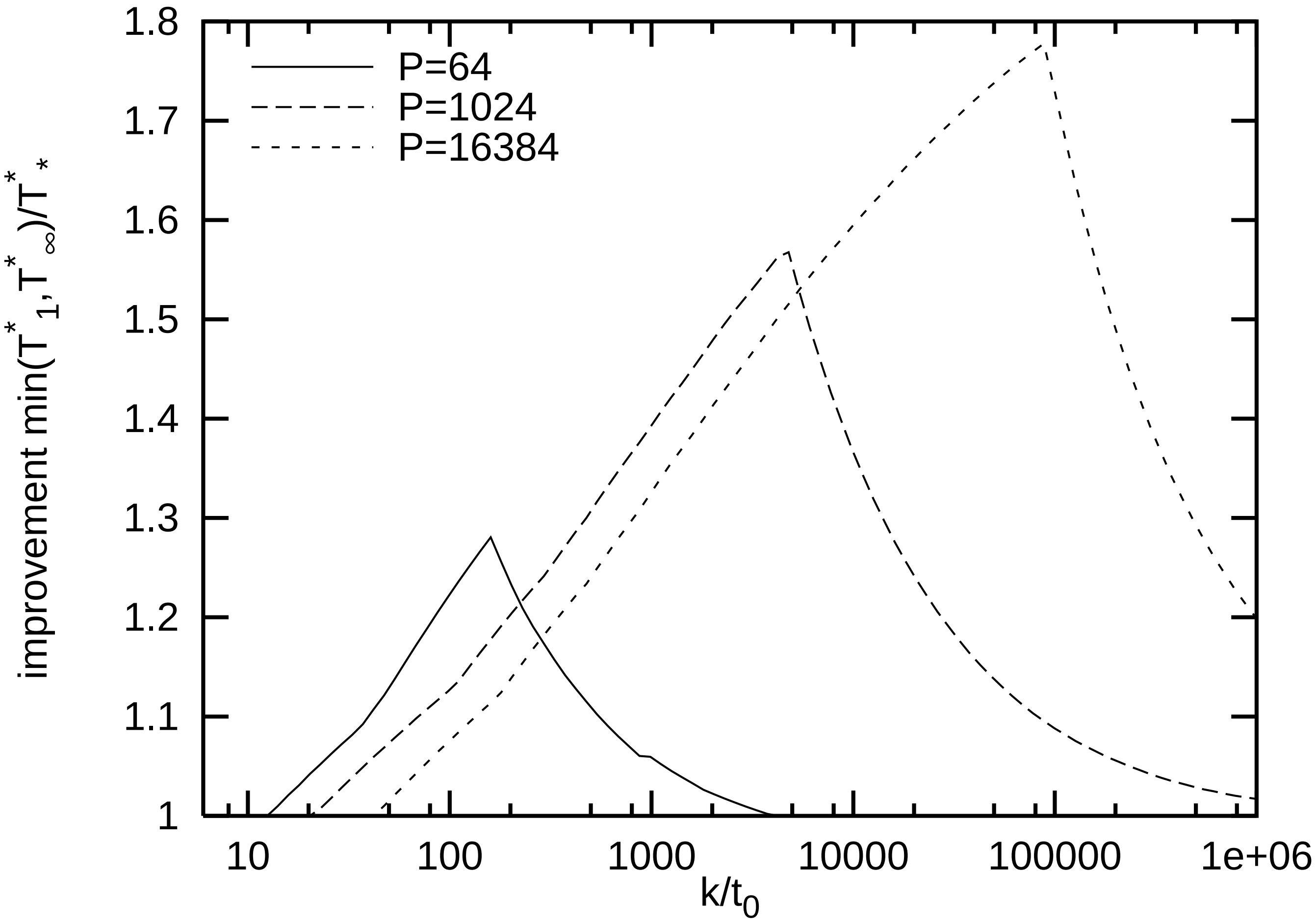
y Axis

clip outclassed algorithms



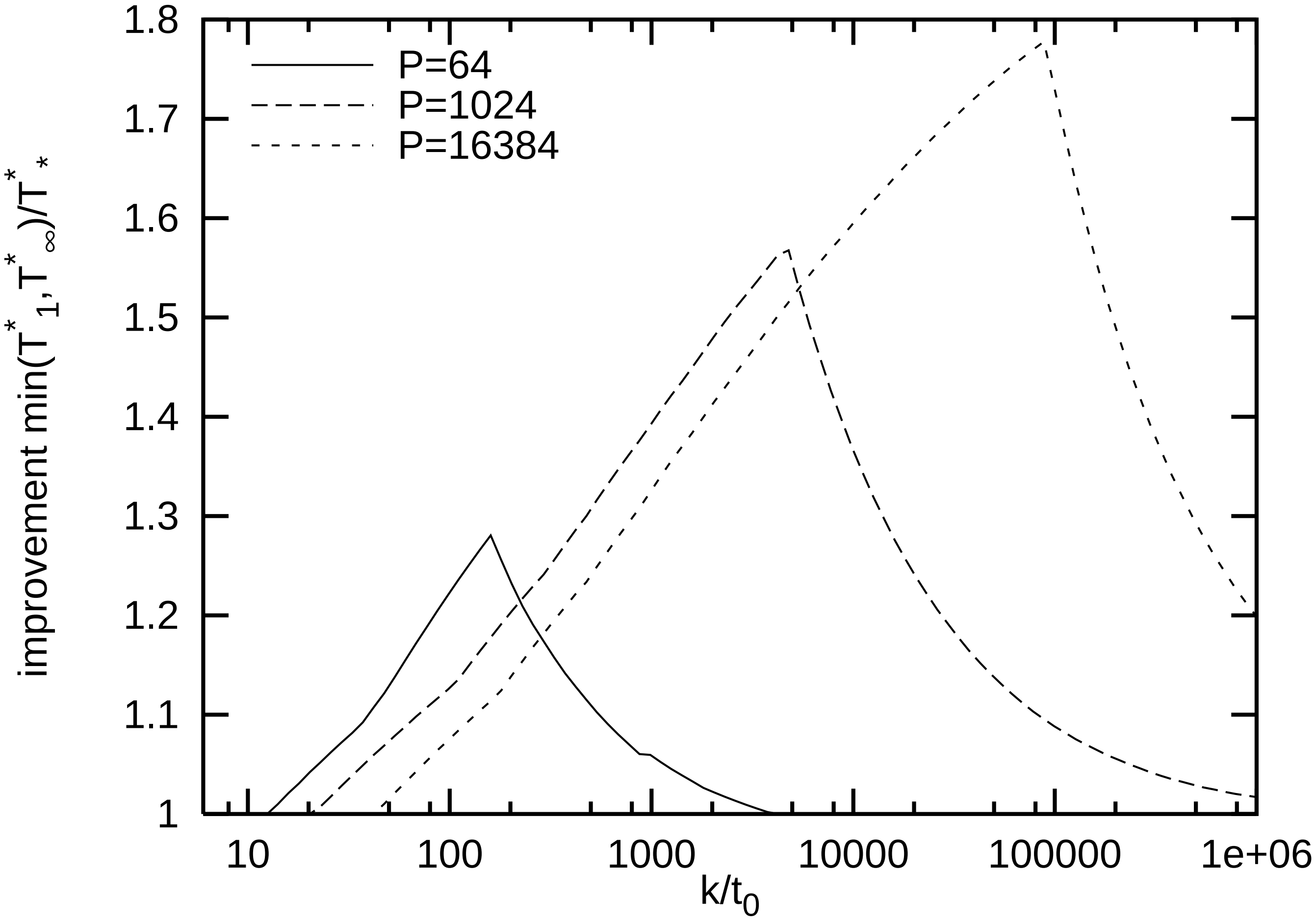
y Axis

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

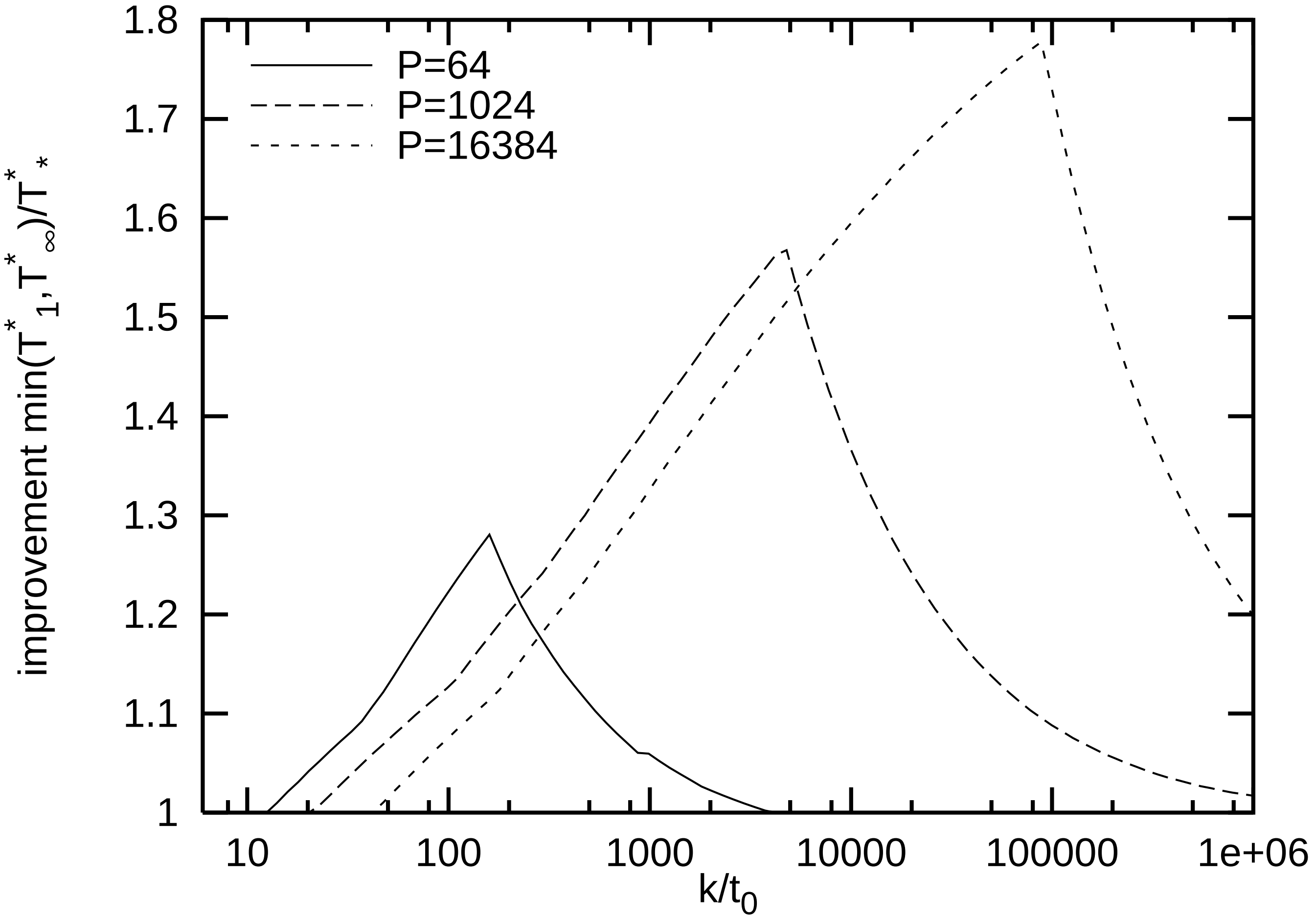
+ better than 3D (reading off values)

− Easily overdone

≤ 7 smooth curves

Reducing the Number of Curves

use ratios



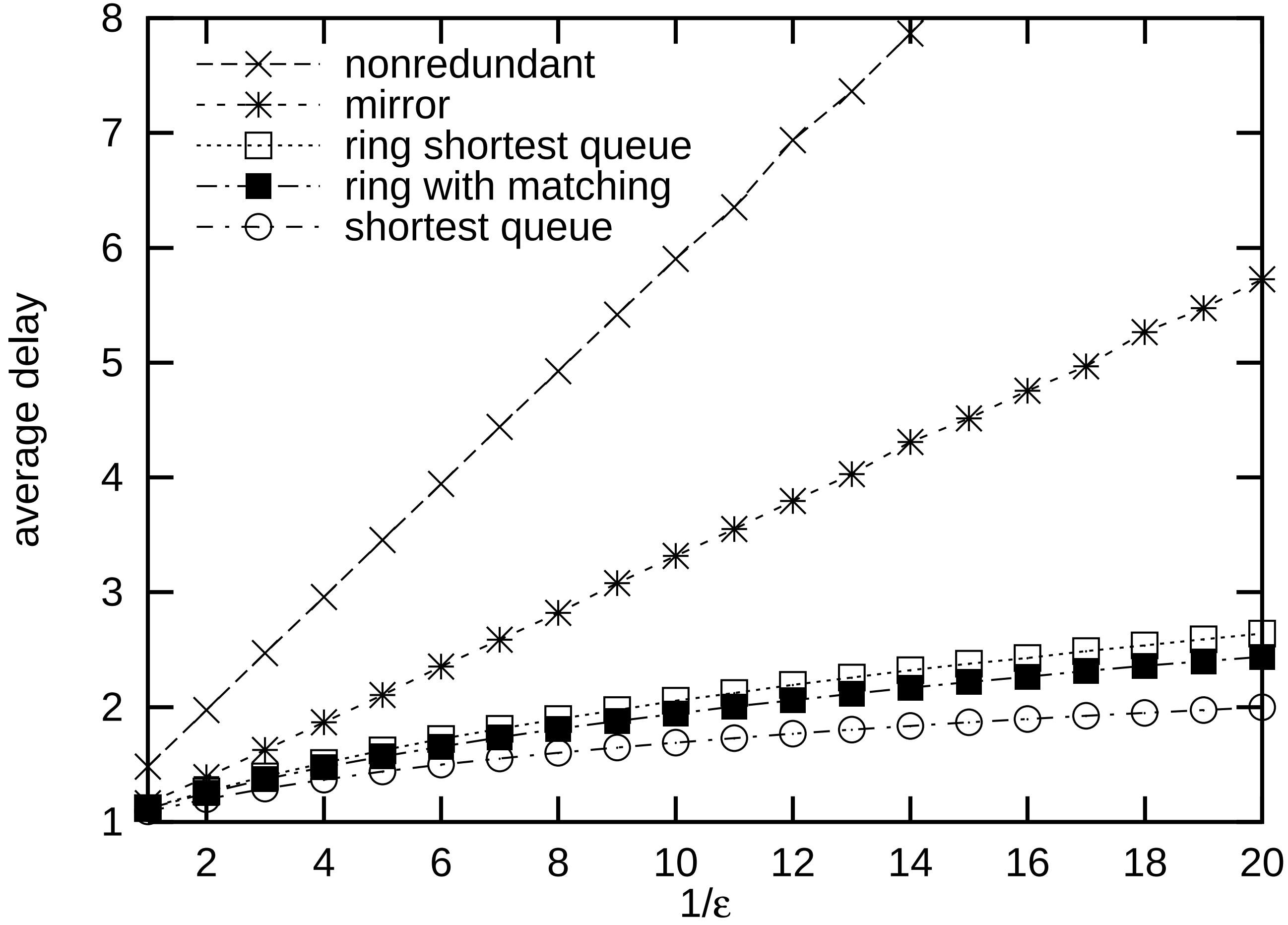
Reducing the Number of Curves

omit curves

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

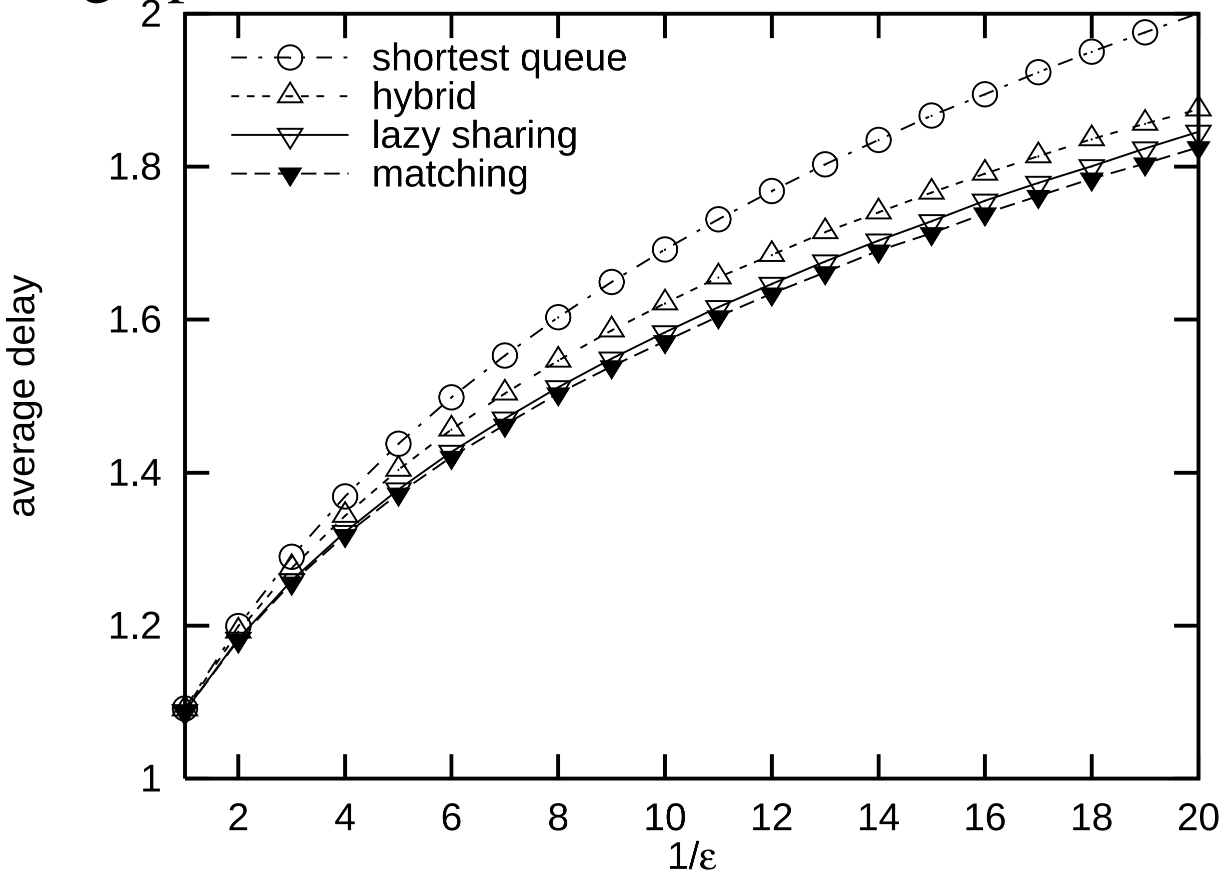
Reducing the Number of Curves

split into two graphs

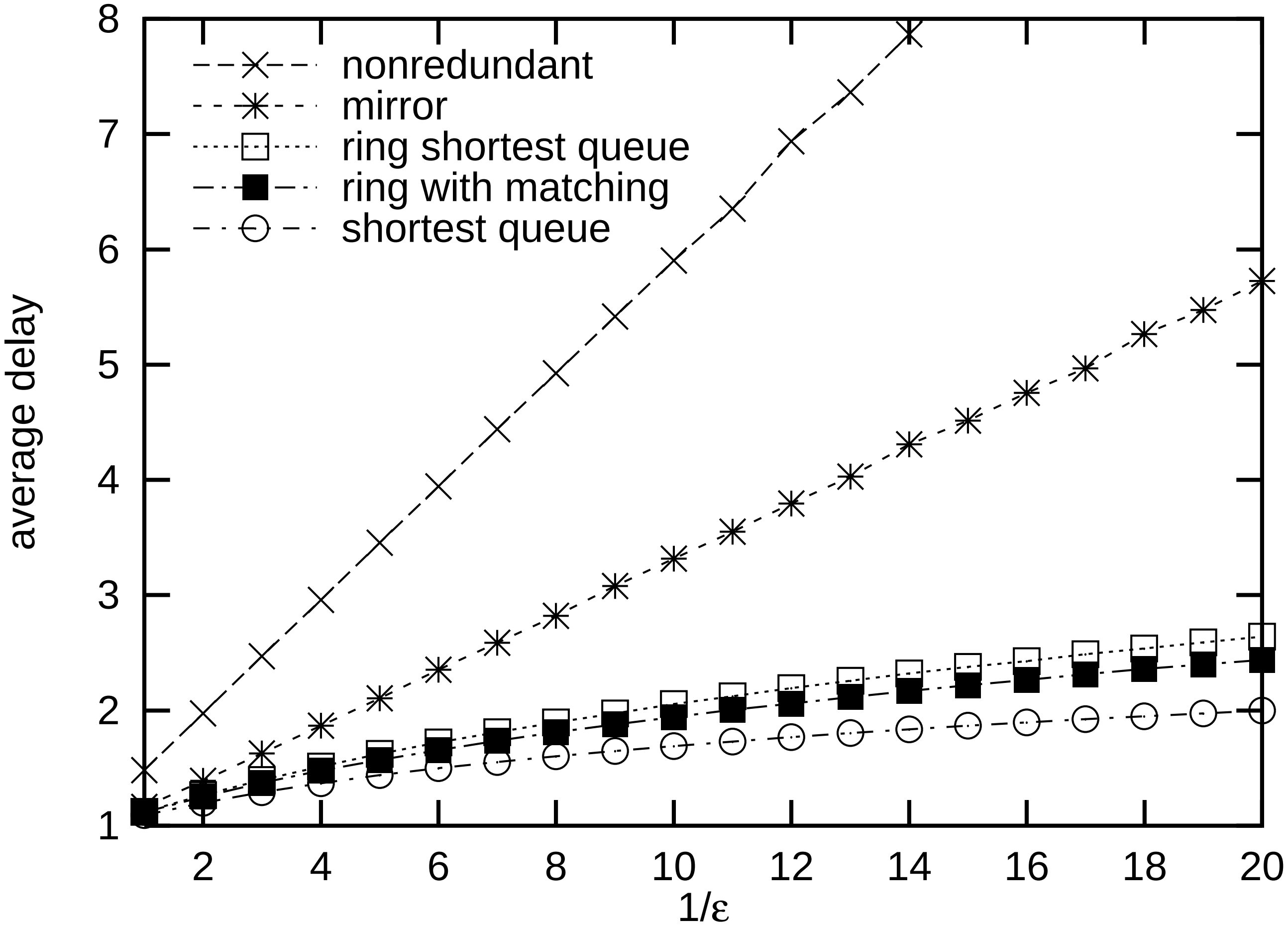


Reducing the Number of Curves

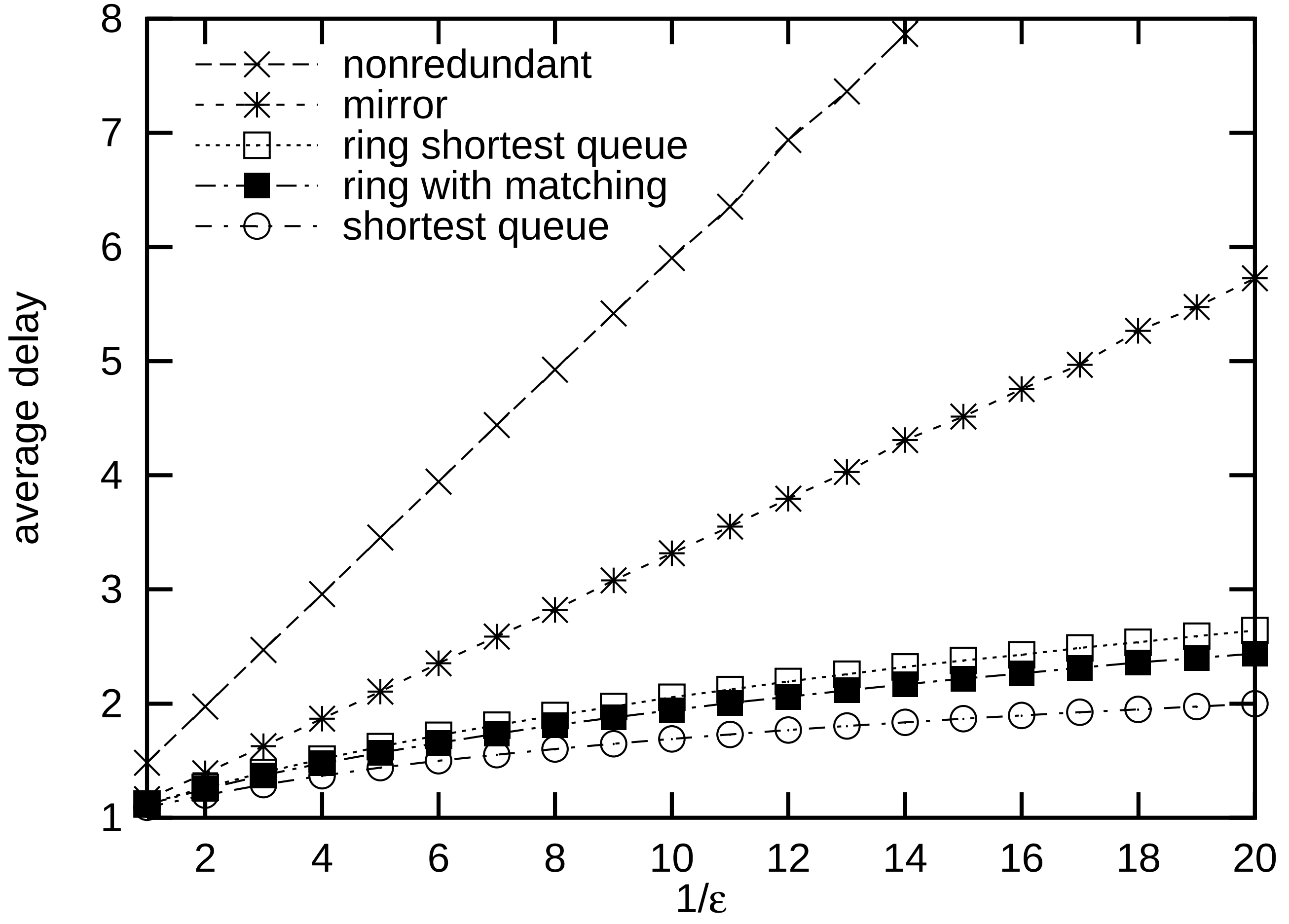
split into two graphs



Keeping Curves apart: smoothing



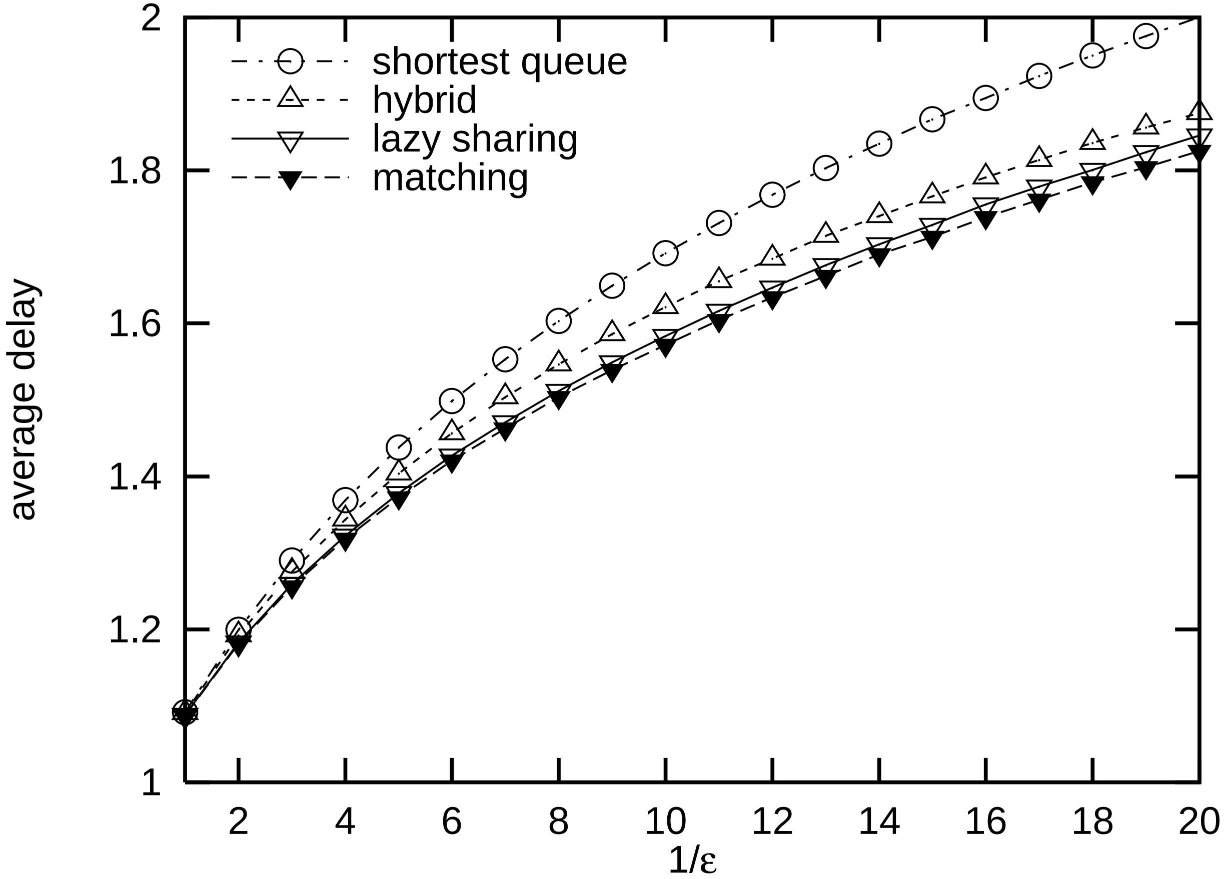
Keys



same order as curves

Keys

place in white space



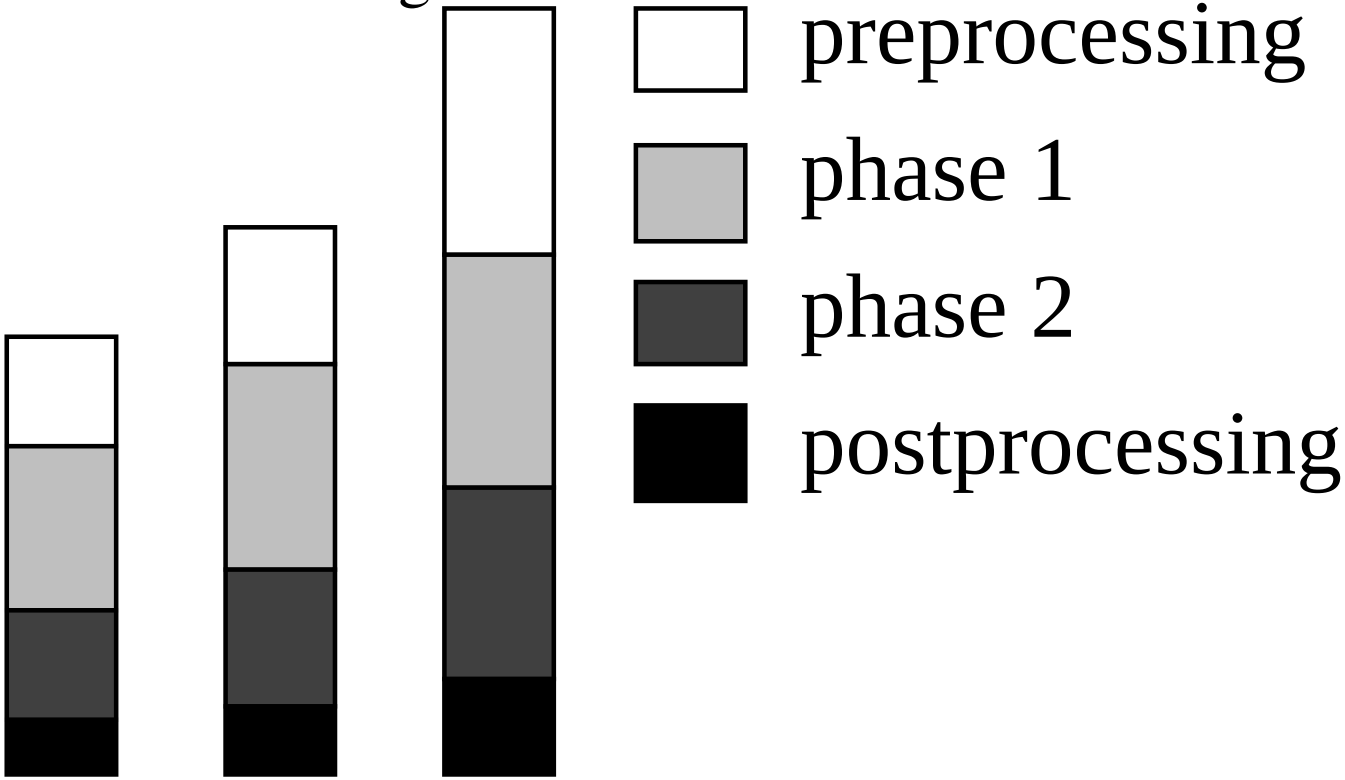
consistent in different figures

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

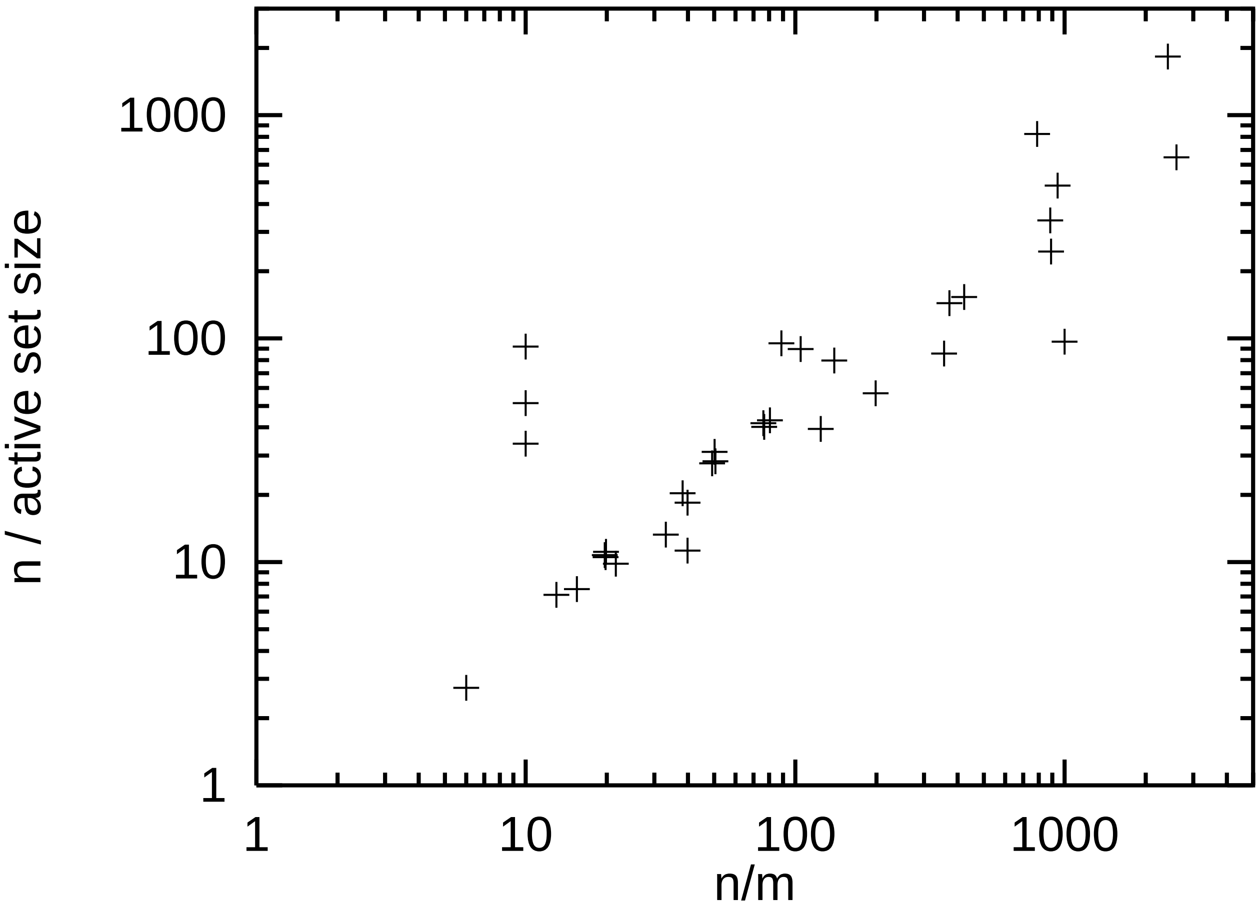
Arranging Instances

- 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.

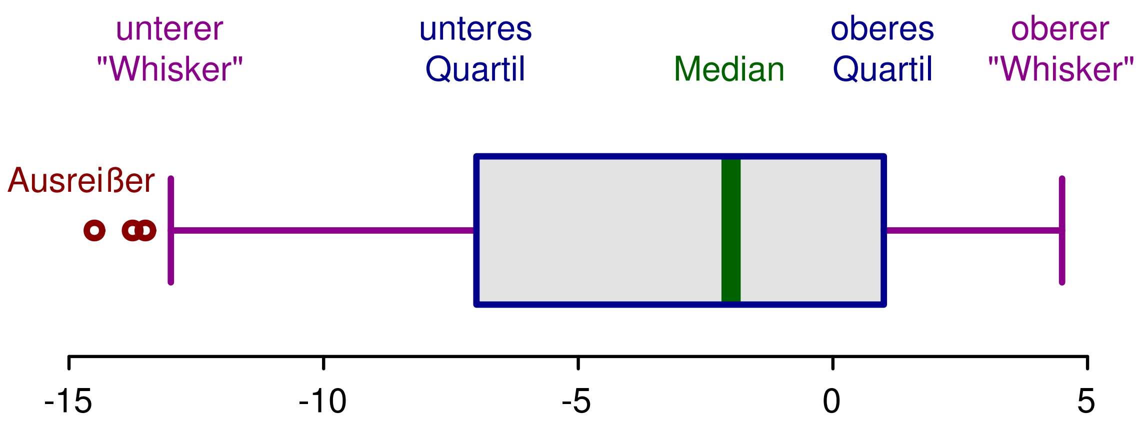
Grids and Ticks

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

Representing Distributions

e.g., when measurements are repeated. Levels of “Escalation”

- Just Average or Median
- Average/Median and Min/Max or empirical variance
- Box-Whisker-Plot: Median, Quartile, “Whiskers”, outlier
- Violin plot or histogram



3D

− you cannot read off absolute values

− interesting parts may be hidden

− only one surface

\+ good impression of shape

□ Perhaps good in an interactive context?

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 <u>not</u> random variables.
- Use points to indicate for which x-values actual data is available.
- Connect points belonging to the same curve.

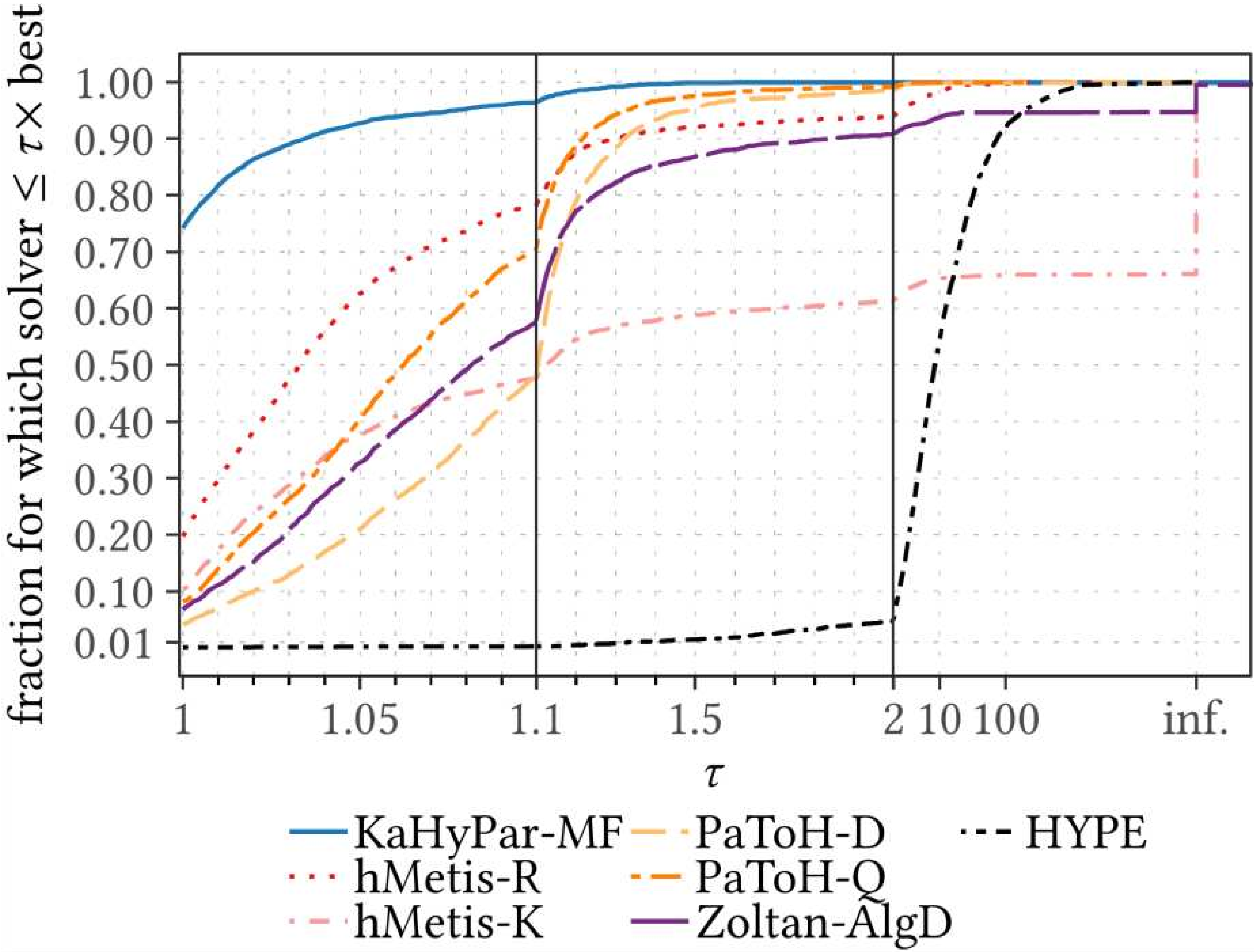
- ☐ Only use splines for connecting points if interpolation is sensible.
- ☐ Do not connect points belonging to unrelated problem instances.
- ☐ Use different point and line styles for different curves.
- ☐ Use the same styles for corresponding curves in different graphs.
- ☐ Place labels defining point 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.

Comparing Apples and Oranges

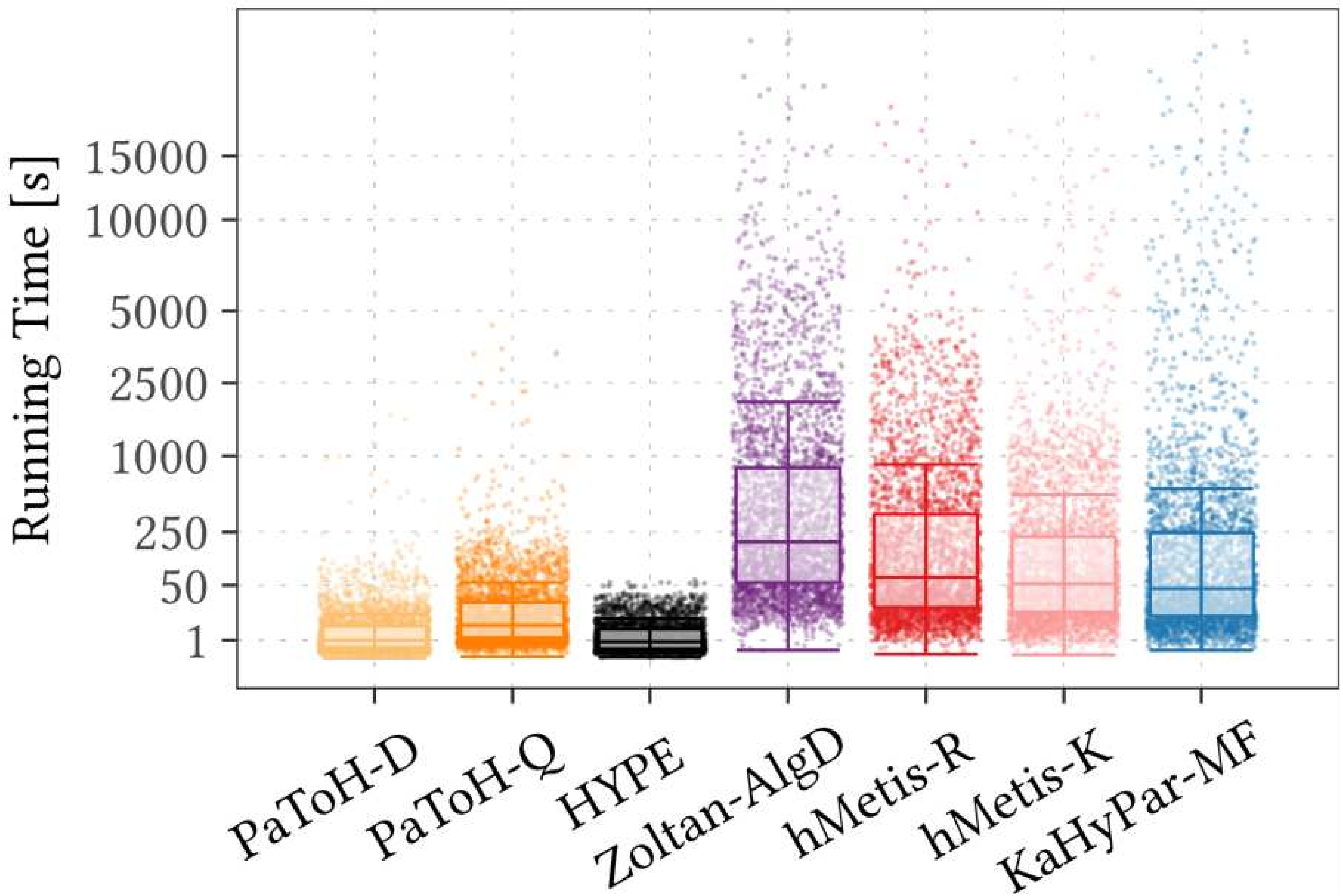
In optimization problems we compare
running time and solution quality
for many different instances.
What is the better algorithm???

- 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

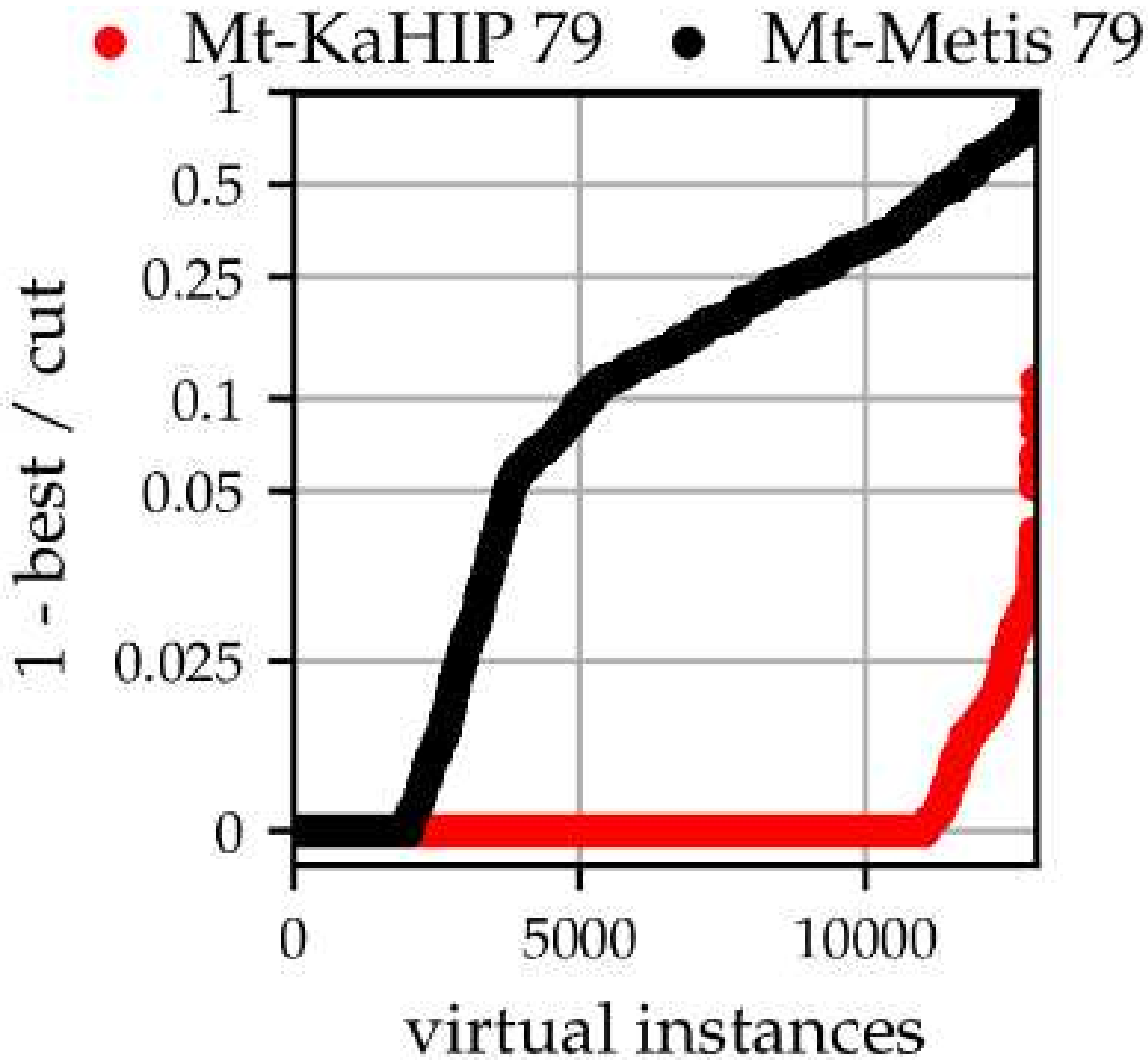
Compare some repetitions of algorithms A and B.
Yields several virtual instances

- 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

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

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 (ipsssso). 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] Jan Wassenberg, Mark Blacher, Joachim Giesen, and Peter Sanders. Vectorized and performance-portable quicksort. Softw. Pract. Exp., 52(12):2684–2699, 2022.

[23] Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. Algorithmica, pages 1–52, 2015.

[24] Timo Bingmann, Peter Sanders, and Matthias Schimek. Communication-efficient string sorting. In 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020.

[25] 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.

[26] 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.

[27] Stefan Edelkamp and Armin Weiß. Blockquicksort: Avoiding branch mispredictions in quicksort. Journal of Experimental Algorithmics (JEA), 24:1–22, 2019.

[28] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. Theory of Computing Systems, 38(2):229–248, 2005.

[29] J. G. Cleary. Compact hash tables using bidirectional linear probing. IEEE Transactions on Computers, C-33(9):828–834, 1984.

[30] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. Proc. VLDB Endow., 5(11):1627–1637, 2012.

[31] Tobias Maier, Peter Sanders, and Robert Williger. Concurrent expandable AMQs on the basis of quotient filters. In 18th Symposium on Experimental Algorithms (SEA), LIPIcs, 2020.

[32] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. SIAM Journal on Computing, 17(2):373–386, 1988.

[33] M. Naor and O. Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. Journal of Cryptology: the journal of the International Association for Cryptologic Research, 12(1):29–66, 1999.

[34] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In 2010 IEEE 51st Annual symposium on foundations of computer science, pages 787–796. IEEE, 2010.