Algorithm Engineering for Fundamental Data Structures and Algorithms

Peter Sanders

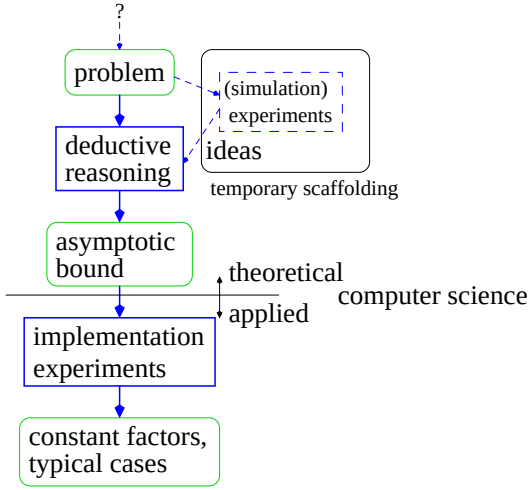
MPII

with

Rene Beier, Stefan Burkhardt, Roman Dementiev, Sebastian Egner, Dimitris Fotakis, Stefan Funke, David Hutchinson, Juha Kärkkäinen, Irit Katriel, Lutz Kettner, Domagoj Matijevic, Kurt Mehlhorn, Jens Mehnert, Uli Meyer, Thomas Novak, Rasmus Pagh, Dominik Schultes, Jop Sibeyn, Naveen Sivadasan, Paul Spirakis, Jesper Träff, Jeff Vitter, Berthold Vöcking, Sebastian Winkel

The Traditional Theoretical View?

A Waterfall Model of Algorithmics



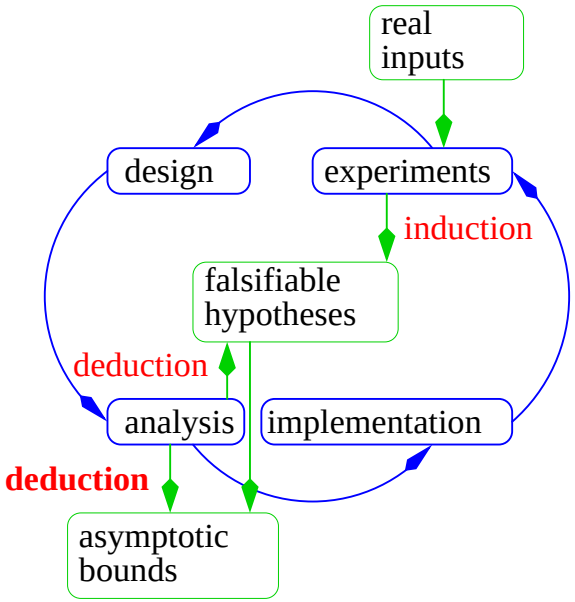
Algorithm Engineering

Scientific Method

The model

of natural science

[Popper 1934]



Goals

- Theory meets technology — machine models must cope with technologiscal developments
- Faster transfer of algorithmic result into applications
- Bridge existing gaps

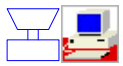
Symptoms of Gaps

theory		⟷		practice
simple		**problem model**		complex
simple		**machine model**		real
complex		**algorithms**	FOR	simple
advanced		**data structures**		arrays, . . .
worst case	max	**complexity measure**		real inputs
asymptotical	$\mathcal{O}(\cdot)$	**efficiency**	42%	constant factors

Why Bridge Gaps?

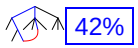
- With growing problems size asymptotics will eventually win
- Worst case bounds
  - ⇝ performance guarantees
  - ⇝ quality, real time properties
- Theory in the natural science means:
  Theory explains reality

Overview



Sorting: All kinds of machine models

- Why comparisons are (not) the bottleneck
- Parallel disk sorting
- Suffix array construction — an “all model algorithm”



Priority Queues: More memory hierarchies

Search Trees: What integer keys do for us

Hashing: Space efficiency + access time guarantees

Minimum Spanning Trees:

- Does the cycle property help?
- Huge sparse graphs

42%

Maximum Flows: Is the theoretically best algorithm practicable?

max

1 Super Scalar Sample Sort

Comparison Based Sorting

Large data sets (thousands to millions)

Theory: Lots of algorithms with $n \log n + \mathcal{O}(n)$ comparisons

Practice: Quicksort

- + $n \log n + \mathcal{O}(n)$ expected comparisons
  using sample based pivot seletction
- + sufficiently cache efficient
  $\mathcal{O}\left(\frac{n}{B} \log n\right)$ cache misses on all levels
- + Simple
- + Compiler writers are aware of it

Can we nevertheless beat quicksort?

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 ≈ 20 % speedup, and only for laaarge inputs

Sample Sort

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

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

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

sort S

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

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

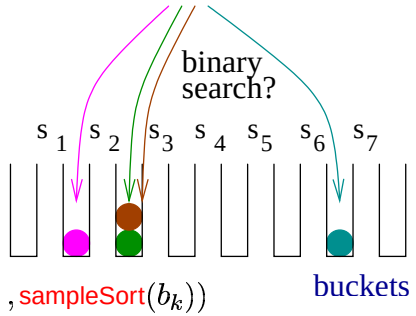
for $i := 1$ **to** n **do**

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 $\approx$ merge sort
– Binary search not much faster than merging
– complicated memory management

Super Scalar Sample Sort

☐ Binary search $\longrightarrow$ implicit search tree
☐ Eliminate 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** **(* unroll $\approx 4\times$ *)**

$j := 1$

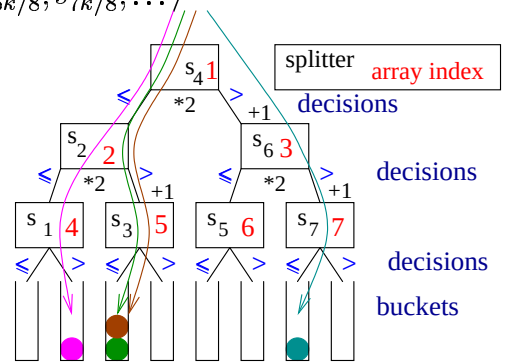
repeat $\log k$ times **(* unroll *)**

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

$j := j - k + 1$

$|b_j|$ ++

o(i):= j **(* oracle *)**

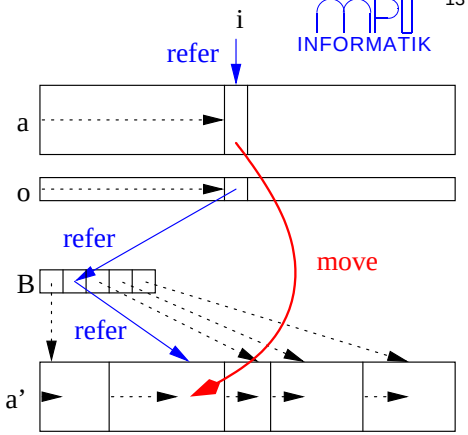


Now the compiler should:

☐ use predicated instructions
☐ interleave iterations of the for loop (software pipelining)

Distributing Elements

for $i := 1$ **to** n **do** $a'_{B[o(i)]++} := a_i$



Why Oracles?

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

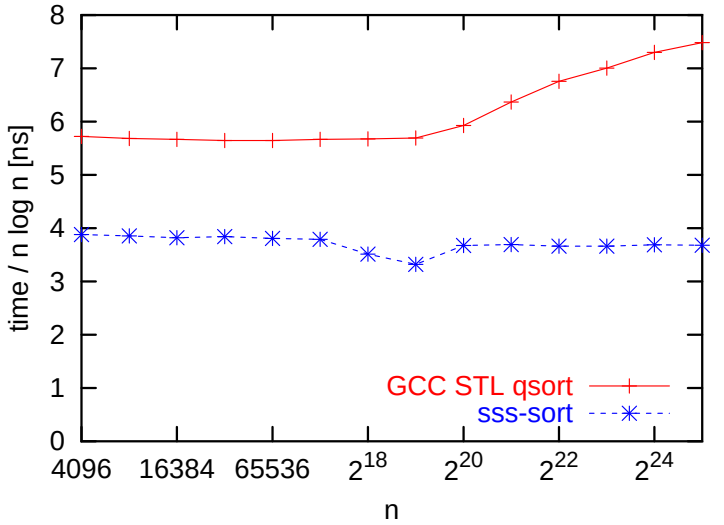
Analysis

	mem. acc.	branches	data dep.	I/Os	registers	instructions
k-way distribution:						
sss-sort	$n \log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$3.5n/B$	3×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 $k'^{\log_{k'} k}$	$2n \log_{k'} k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	$2k'+2$	$\mathcal{O}(k')$

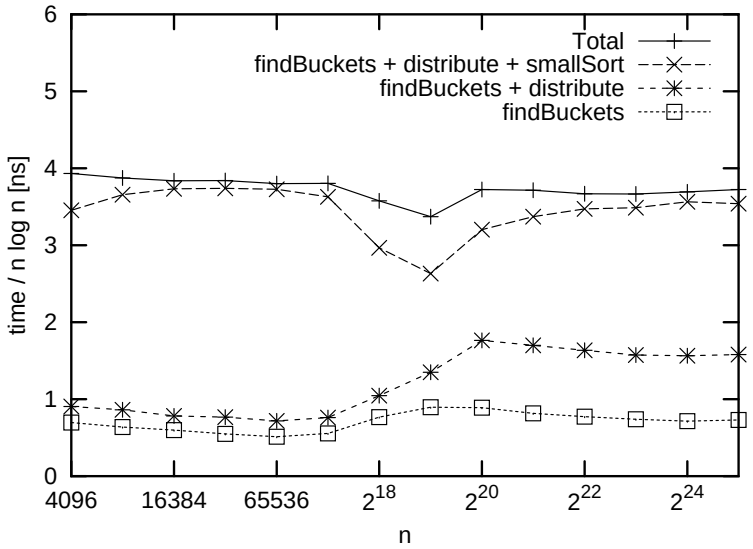
Experiments: 1.4 GHz Itanium 2

- `restrict` keyword from ANSI/ISO C99
  to indicate nonaliasing
- Intel's C++ compiler v8.0 uses predicated instructions automatically
- Profiling gives 9% speedup
- $k = 256$
- 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



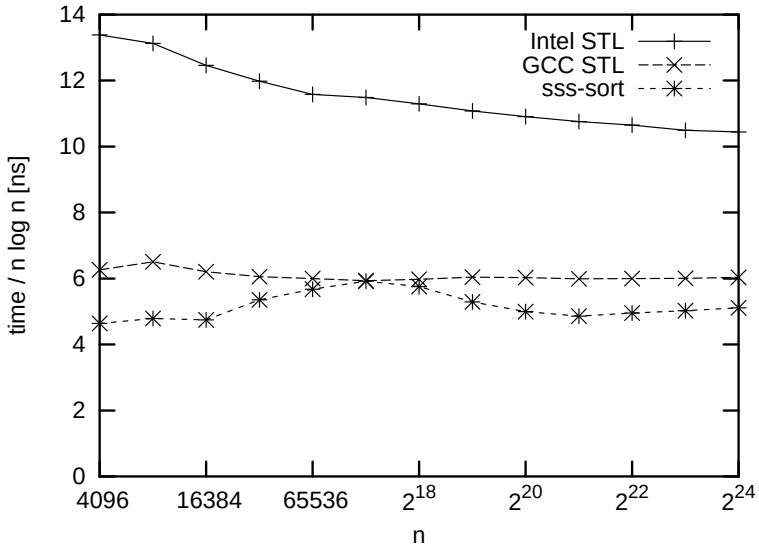
Breakdown of Execution Time



A More Detailed View

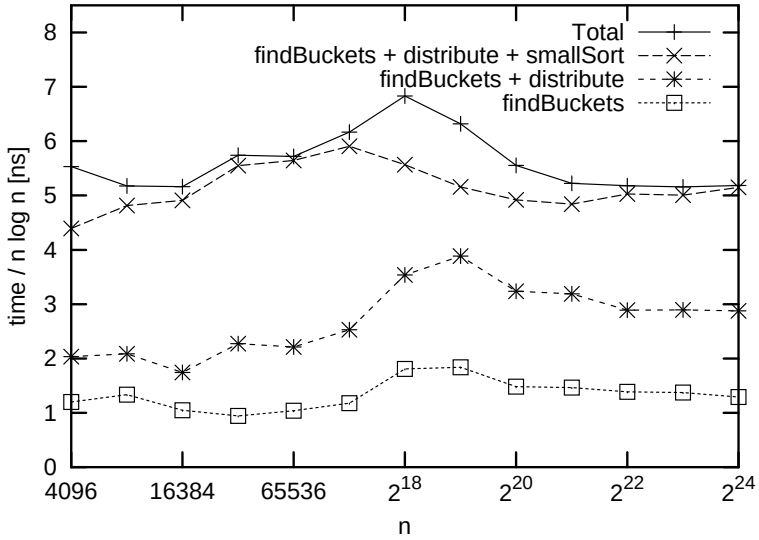
	instr.	cycles	IPC in L3	IPC $n = 2^{25}$
findBuckets, $1\times$ outer loop	63	11	5.4	4.5
distribute, one element	14	4	3.5	0.8

Comparison with Quicksort Pentium 4



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

Breakdown of Execution Time Pentium 4



Conclusions

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

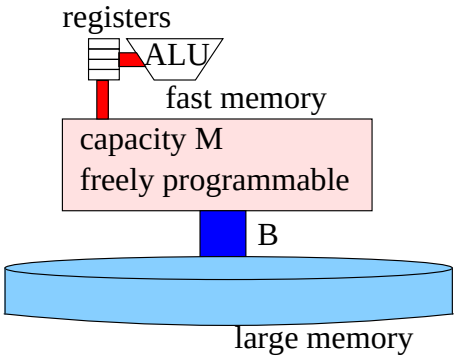
Future Work

- high level fine-tuning, e.g., clever choice of k
- other architectures, e.g., Opteron, PowerPC
- almost in-place implementation
- multilevel cache-aware or cache-oblivious generalization (oracles help)

The Secondary Memory Model

M: Fast memory of size M

B: Block size (?)



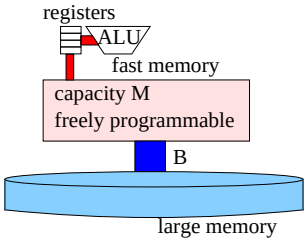
Analysis: count (only?) block accesses (I/Os)

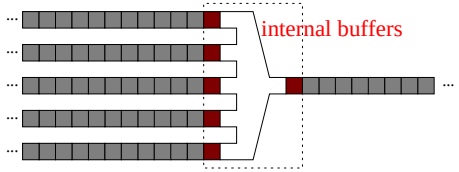
Sorting by Multiway Merging

- 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 only one run is left — $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ merge phases

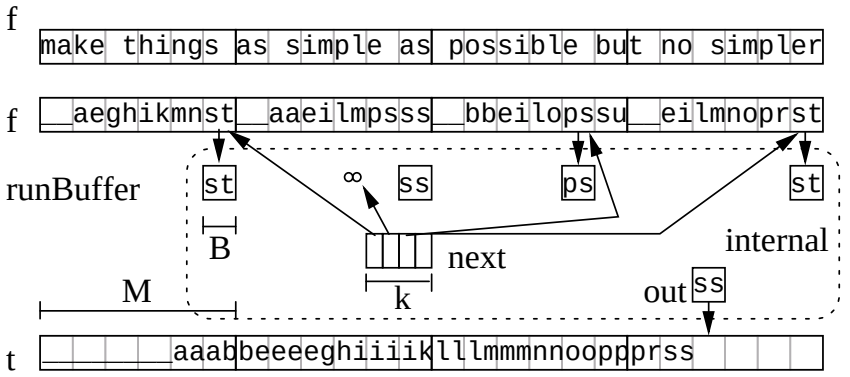
In total

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





Example, $B = 2$, run size $= 6$



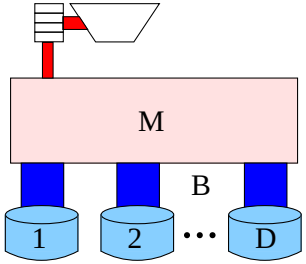
2 Sorting with Parallel Disks

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

Theory: Balance Sort [Nodine Vitter 93].

Deterministic, complex

asymptotically optimal 



independent disks

[Vitter Shriver 94]

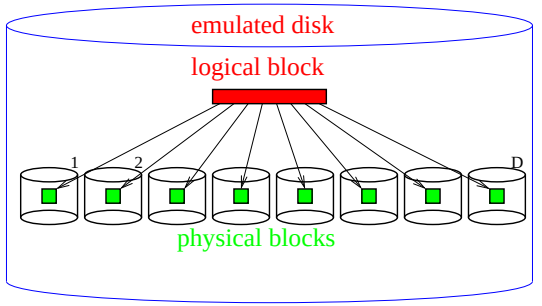
Multiway merging

"Usually" factor 10? less I/Os.

Not asymptotically optimal. 42%

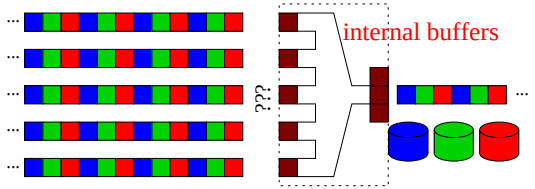
Basic Approach: Improve Multiway Merging

Striping



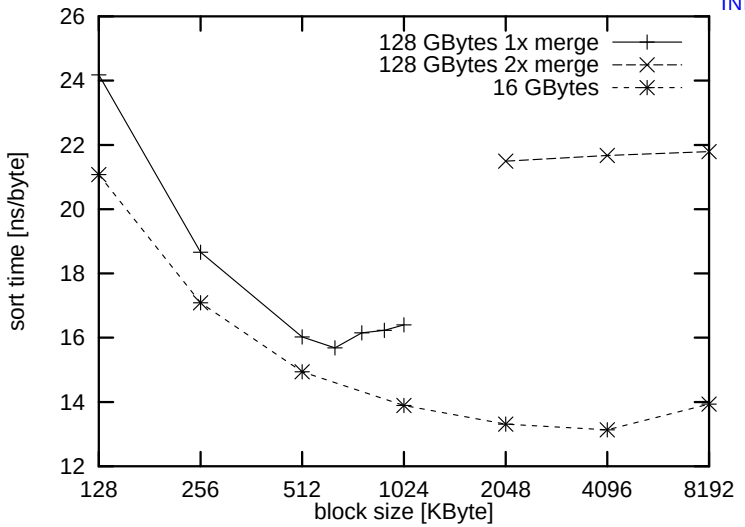
That takes care of run formation

and writing the output



But what about merging?

What are good block sizes (8 disks)?



Much larger than previously assumed.

Using merge buffers for entire stripes would be bad

Prediction

[Folklore, Knuth]

Smallest Element

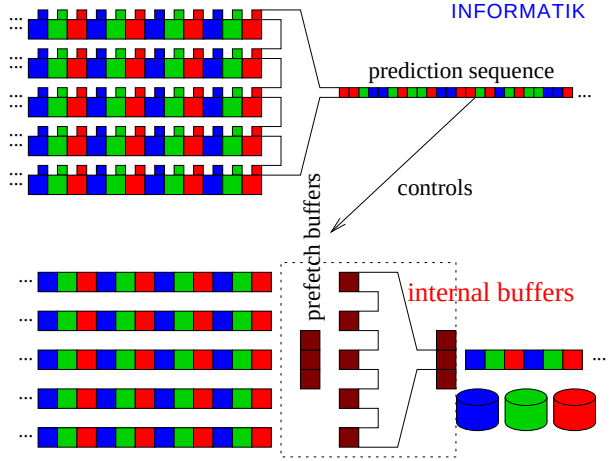
of each block

triggers fetch.

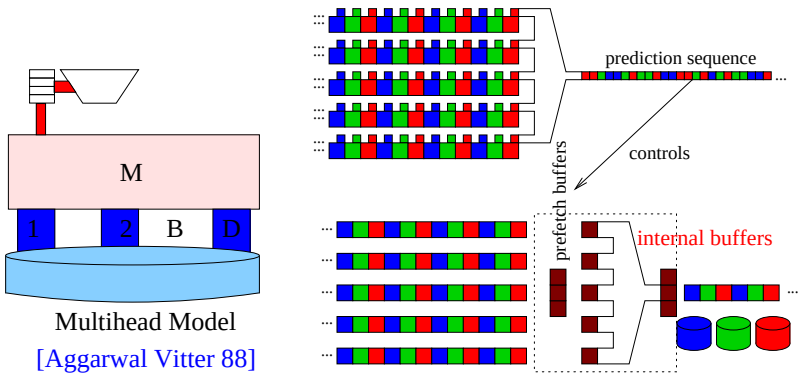
Prefetch buffers

allow parallel access

of next blocks



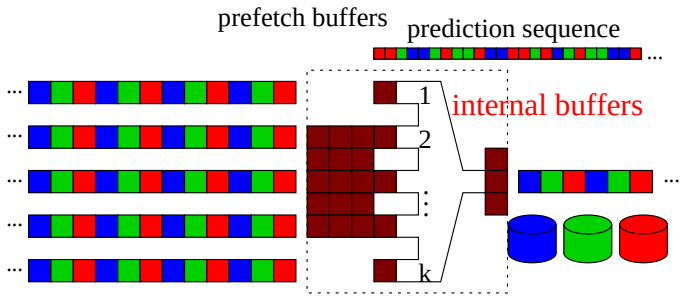
Warmup: Multihead Model



D prefetch buffers yield an optimal algorithm

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

Bigger Prefetch Buffer



$Dk \rightsquigarrow$ good deterministic performance

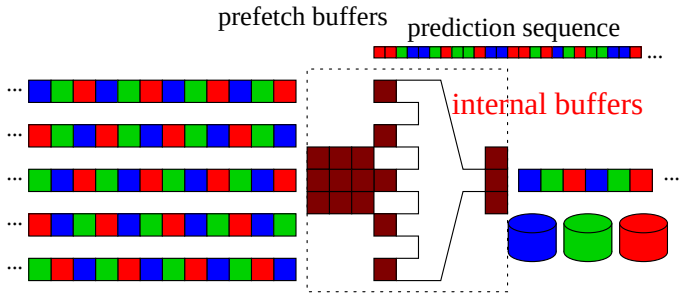
$\mathcal{O}(D)$ would yield an optimal algoritihm.

Possible?

Randomized Cycling

[Vitter Hutchinson 01]

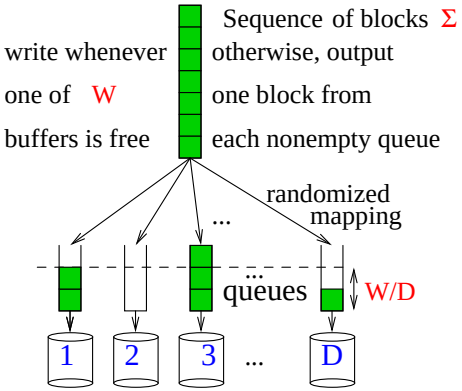
Block i of stripe j goes to disk $\pi_j(i)$ for a random permutation π_j



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

Buffered Writing

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



Theorem:
Buffered Writing
is optimal

...

But
how good is optimal?

Theorem: Randomized 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

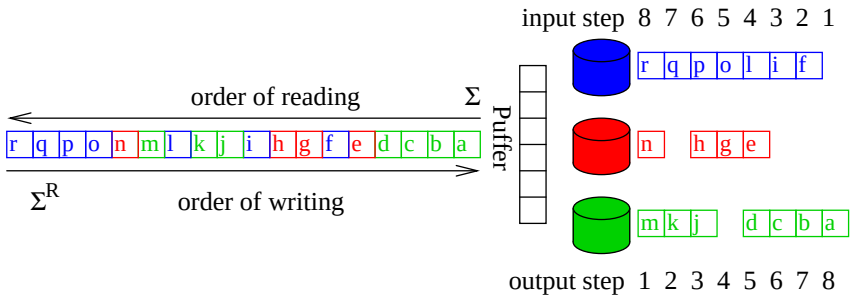
Theorem:

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

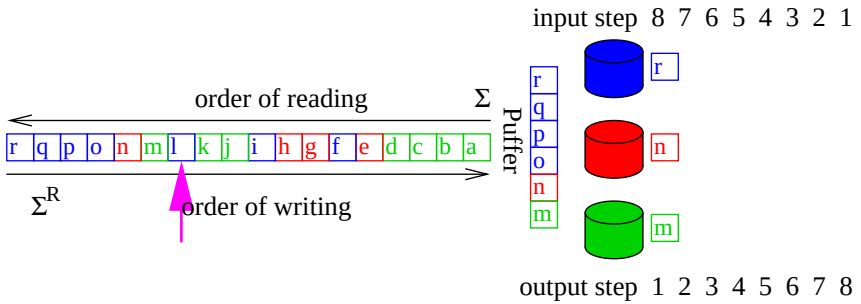
Theorem:

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

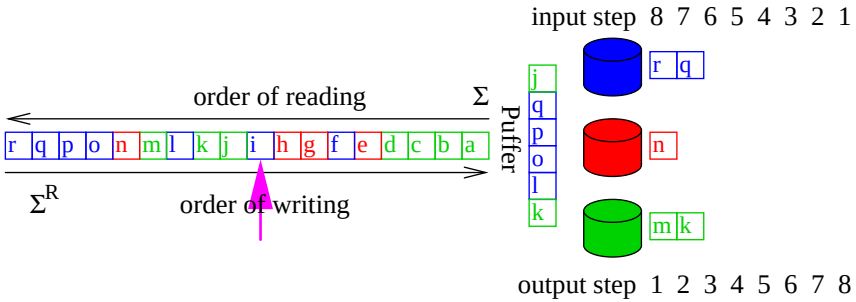
Theorem:

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

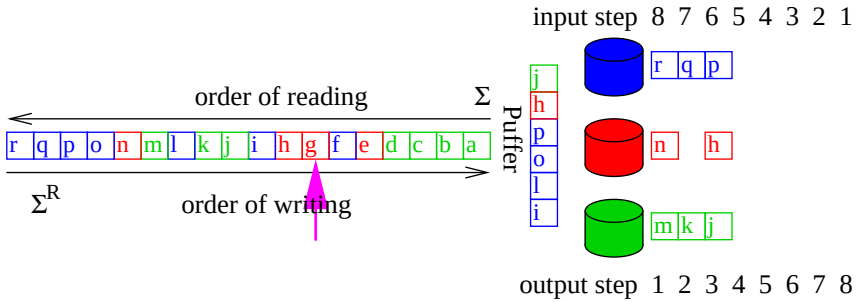
Theorem:

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

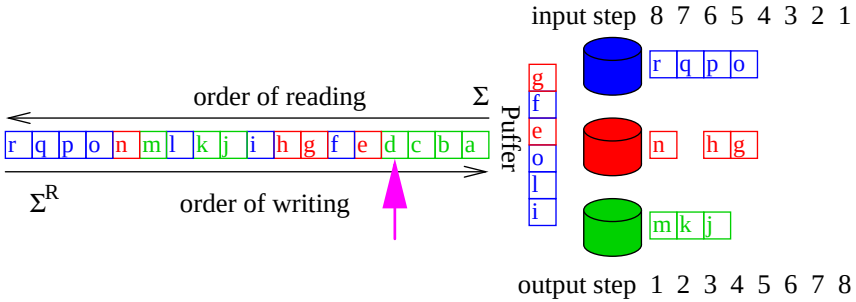
Theorem:

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

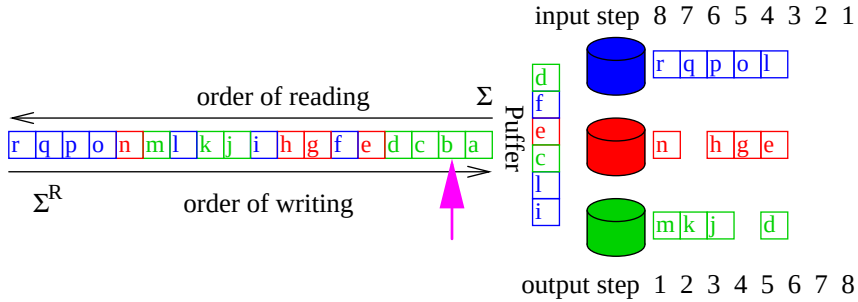
Theorem:

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

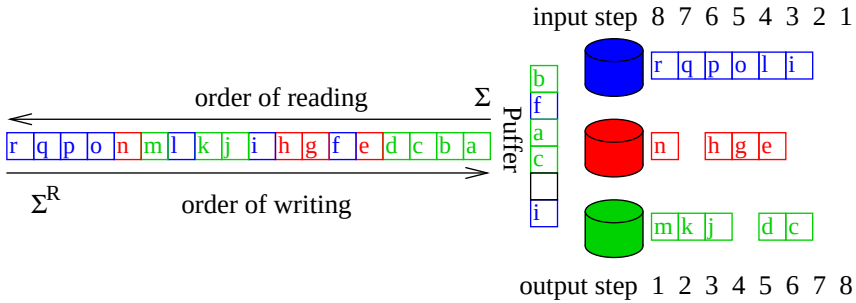
Theorem:

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

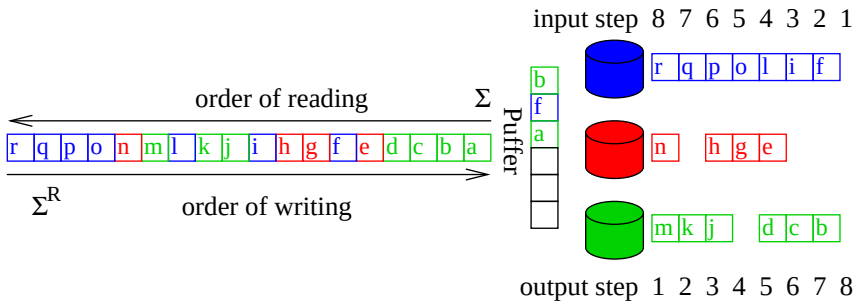
Theorem:

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

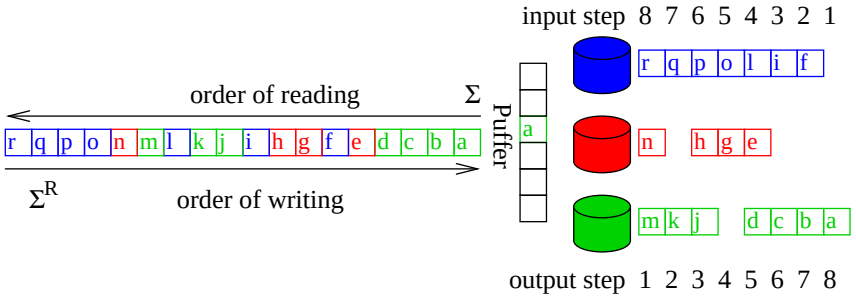
Theorem:

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 [Sanders Egner Korst SODA 2000]

+randomized cycling [Vitter Hutchinson 2001]

+optimal prefetching [Hutchinson Sanders Vitter ESA 2002]

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

$\rightsquigarrow$"answers" question in [Knuth 98]; difficulty 48 on a $1..50$ scale.

We are not done yet!

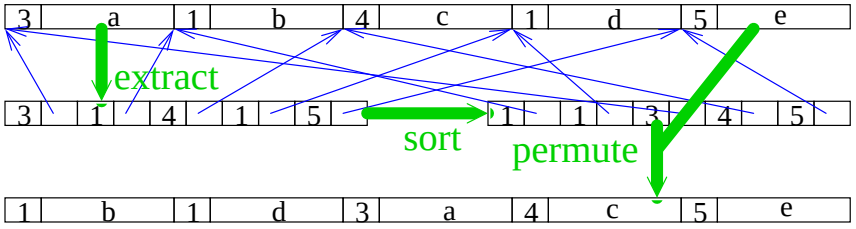
- ☐ Internal work
- ☐ Overlapping I/O and computation
- ☐ Reasonable hardware
- ☐ Interfacing with the Operating System
- ☐ Parameter Tuning
- ☐ Software engineering
- ☐ Pipelining

Key Sorting

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

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

sort key pointer pairs to save memory bandwidth during run formation



Tournament Trees for Multiway Merging

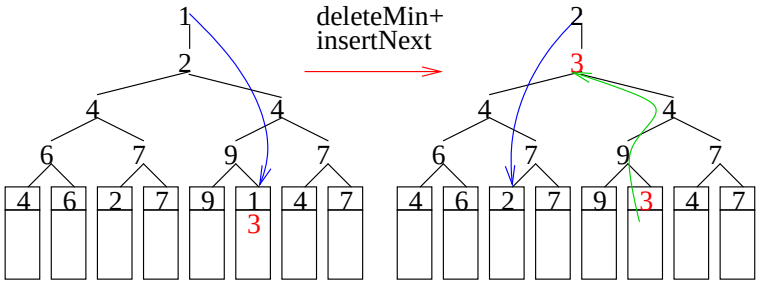
Assume $k = 2^K$ runs

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 $\rightsquigarrow$ simple index arithmetics (shifts)
- Predictable load instructions and index computations

(Unrollable) inner loop:

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


Overlapping I/O and Computation

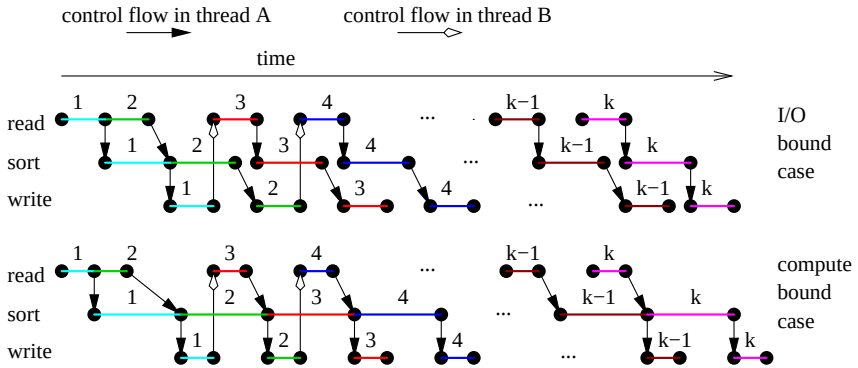
- One thread for each disk (or asynchronous I/O)
- Possibly additional threads
- Blocks filled with elements are passed by references between different buffers

Overlapping During Run Formation

First post read requests for runs 1 and 2

Thread A: Loop $\{$ wait-read i; sort i; post-write $i\}$;

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



Overlapping During Merging

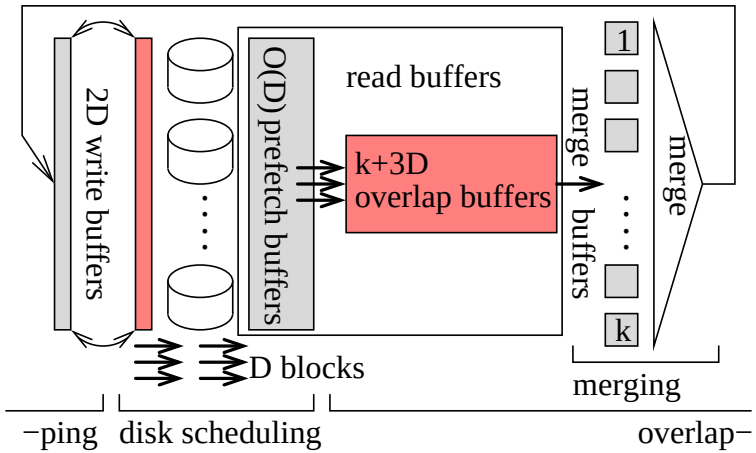
Bad example:

$1^{B-1}2\,3^{B-1}4\,5^{B-1}6\ldots$

$\ldots$

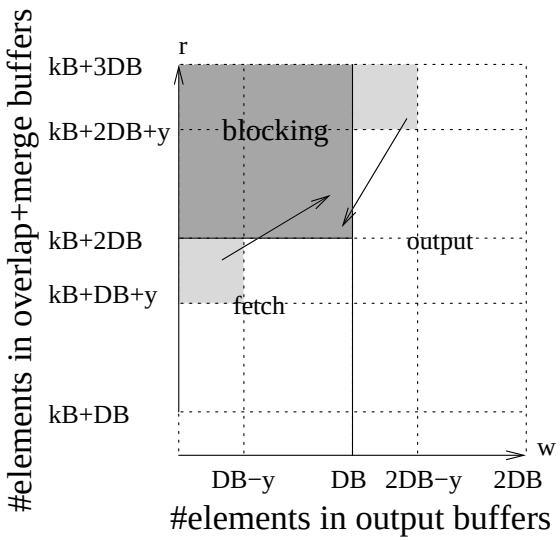
$1^{B-1}2\,3^{B-1}4\,5^{B-1}6\ldots$

Overlapping During Merging

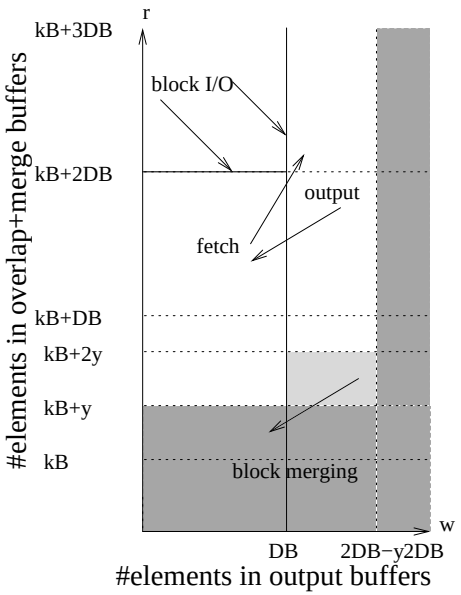


I/O Threads: Writing has priority over reading

I/O bound case: The I/O thread never blocks



Compute bound case: The merging thread never blocks



Hardware (mid 2002)

Linux

(2 × 2GHz Xeon × 2 Threads)

Several 66 MHz PCI-buses

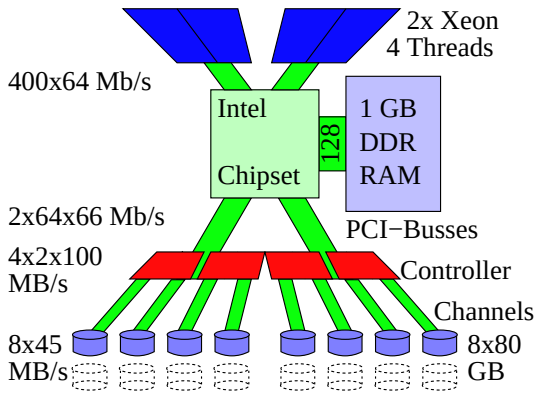
(SuperMicro P4DPE3)

Several fast IDE controllers

(4 × Promise Ultra100 TX2)

Many fast IDE disks

(8× IBM IC35L080AVVA07)

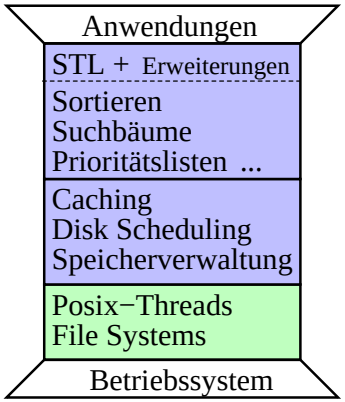


cost effective I/O-bandwidth (real 360 MB/s for ≈ 3000 €)

Software Interface

Goals: efficient + simple + compatible

Basis: C++/STL (`iterator, vector, queue, deque, stack, map, set, stream, string, priority_queue, sort, find, ...`)



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

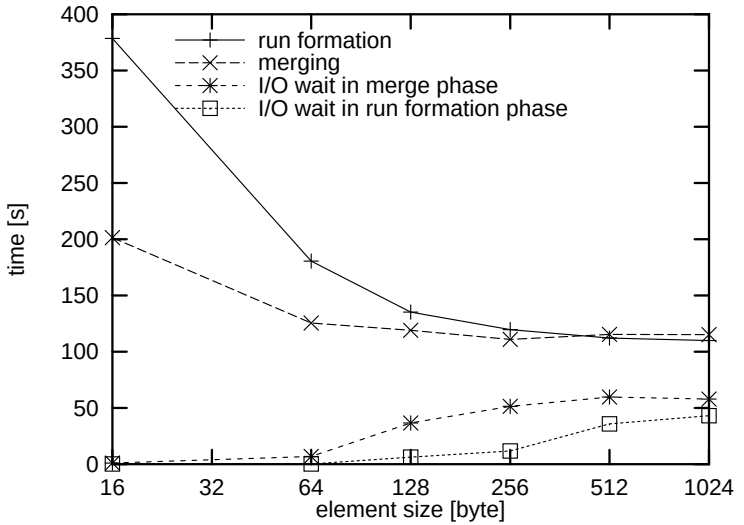
Block size B: 2 MByte

Compiler: g++ 3.2 -O6

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

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

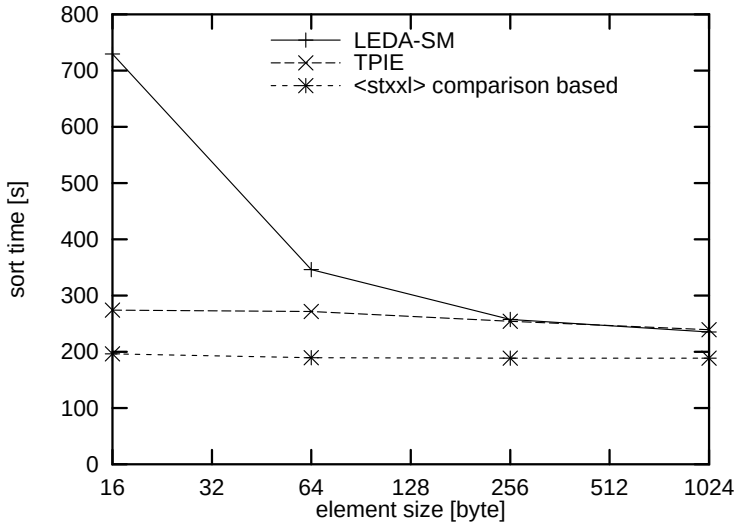
Element sizes (16 GByte, 8 disks)



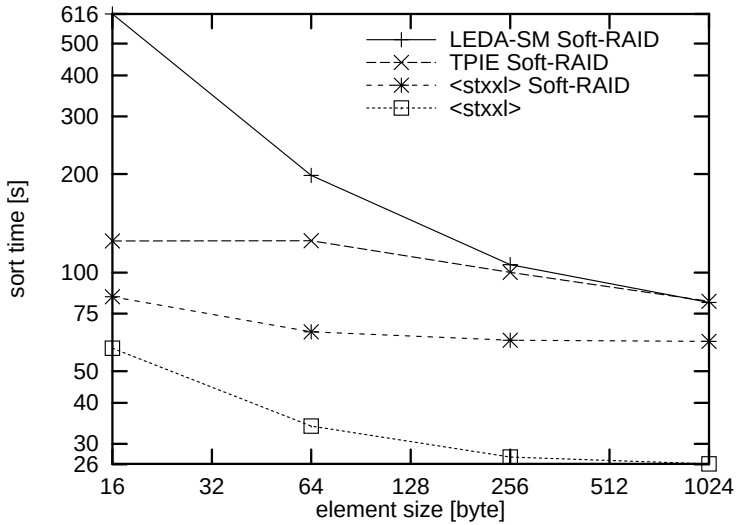
parallel disks ⇝ bandwidth “for free” ⇝ internal work, overlapping are relevent

Earlier Academic Implementations: Single Disk

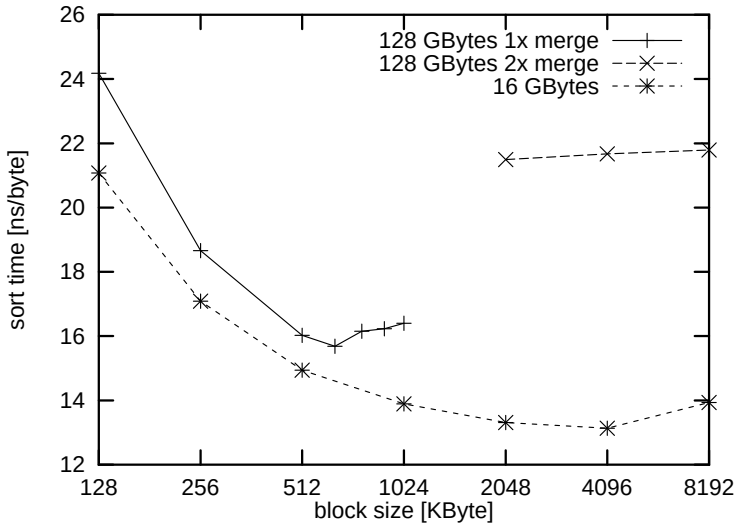
at most 2 GByte, old measurements use artificial M



Earlier Academic Implementations: Multiple Disks



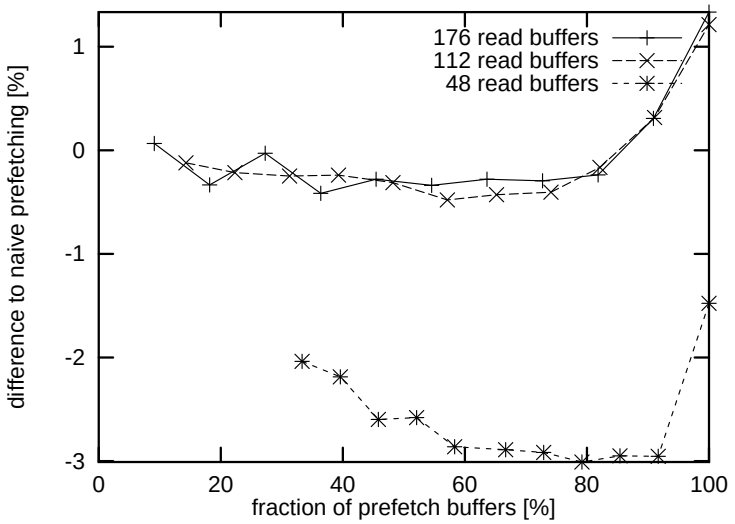
What are good block sizes (8 disks)?



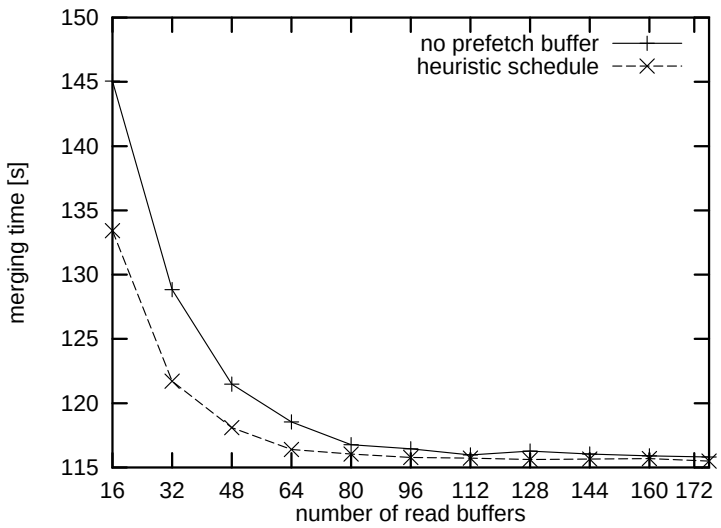
B is not a technology constant

Optimal Versus Naive Prefetching

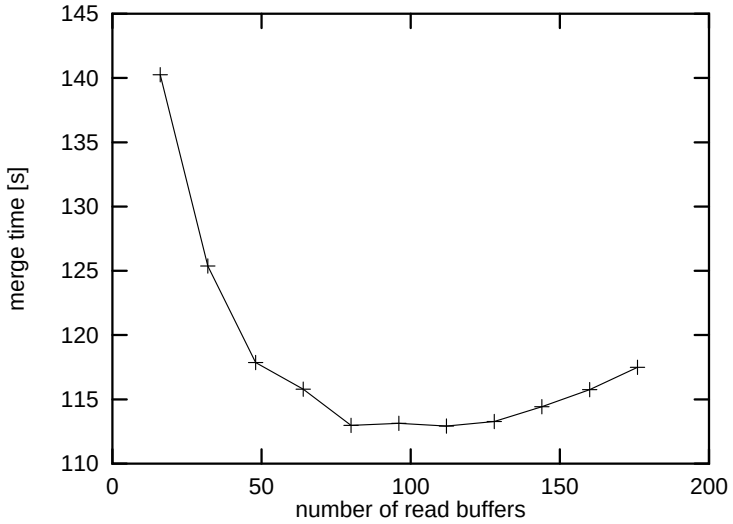
Total merge time



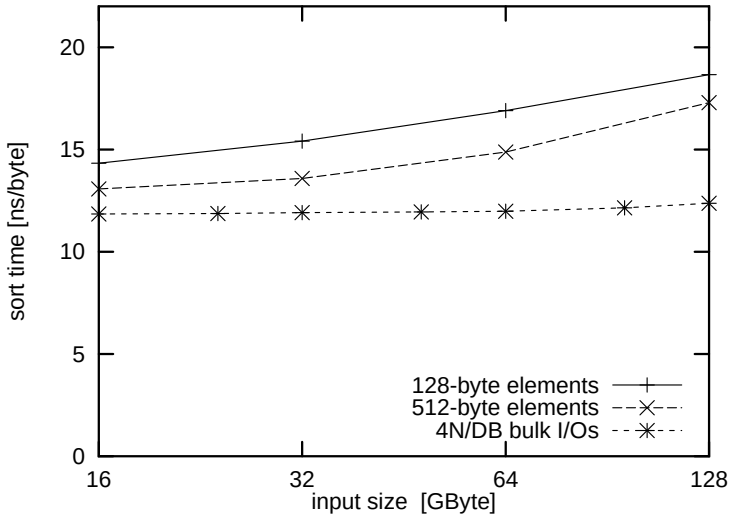
Impact of Prefetch and Overlap Buffers



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 academeic 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 negliect 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
$\rightsquigarrow$ only half the I/Os
$\rightsquigarrow$ often no I/Os for scanning done

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.?

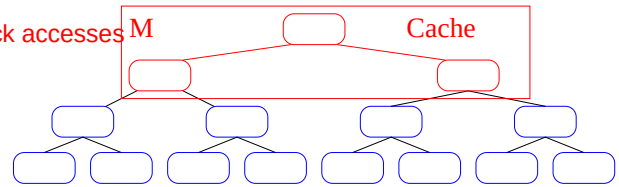
Inplace Sorting: Could also be faster

Determinism: A practical and theoretically efficient algorithm?

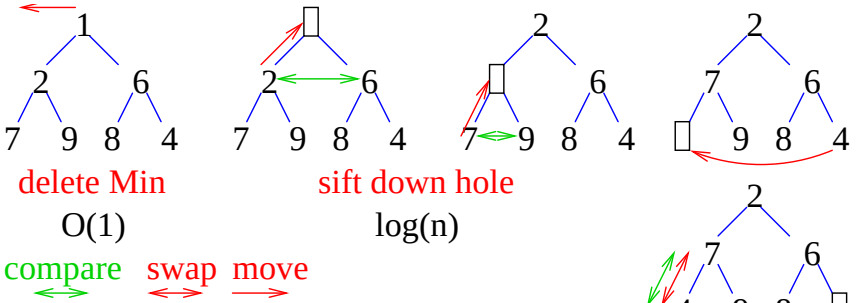
3 Priority Queues (insert, deleteMin)

Binary Heaps best comparison base "flat memory" algorithm

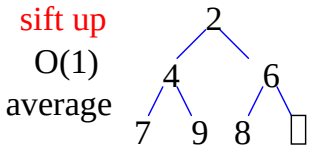
+ On average constant time for insertion
+ On average $\log n + \mathcal{O}(1)$ key comparisons per delete-Min using the "bottom-up" heuristics [Wegener 93].
− $\approx 2\log(n/M)$ block accesses per delete-Min



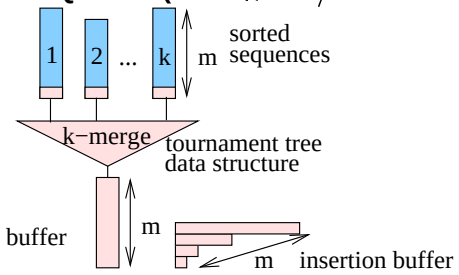
Bottom Up Heuristics



Factor between 1.2 and 2 faster
than (more or less) naive implementation



Medium Size Queues ($km \ll M^2/B$ Insertions)

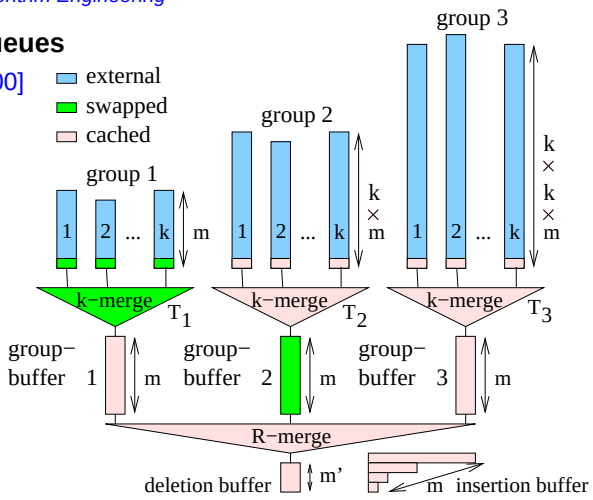


Insert: Initially into insertion buffer. Overflow $\longrightarrow$
sort; merge with deletion buffer; write out largest elements.

Delete-Min: Take minimum of insertion and deletion buffer.
Refill deletion buffer if necessary.

Large Queues

[Sanders 00]



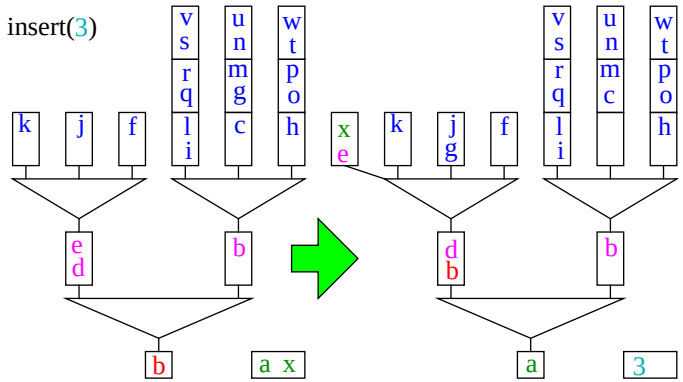
insert: group full $\longrightarrow$ merge group; shift into next group.

merge invalid group buffers and move them into group 1.

Delete-Min: Refill. $m' \ll m$. nothing else

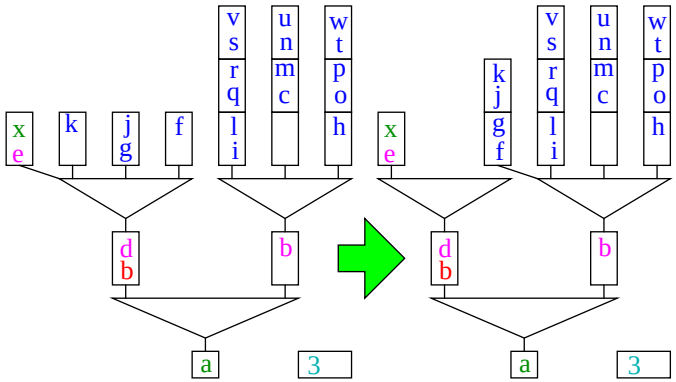
Example

Merge insertion buffer, deletion buffer, and leftmost group buffer



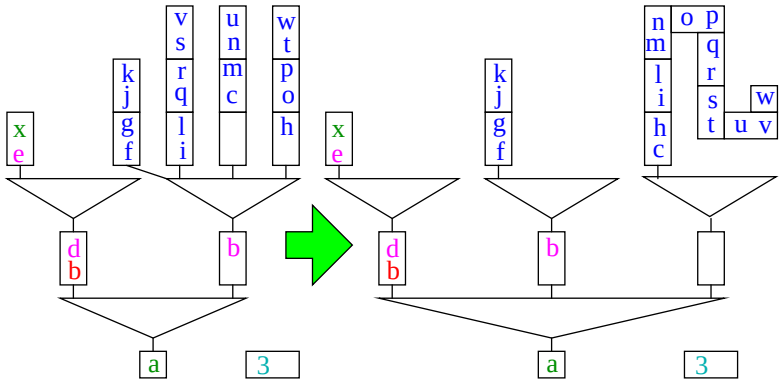
Example

Merge group 1



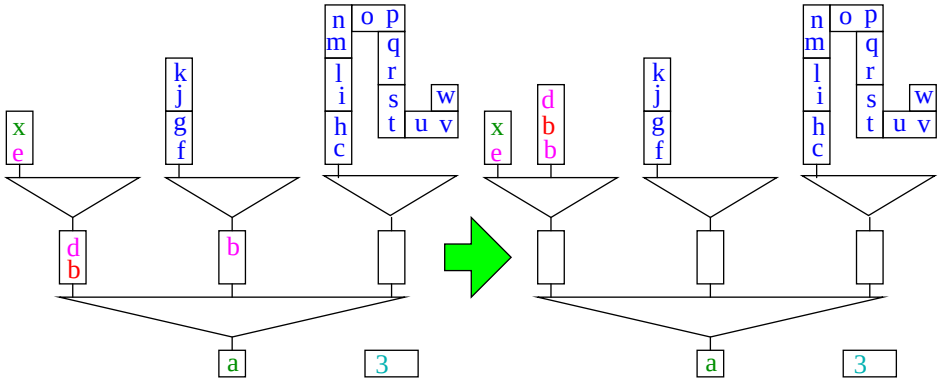
Example

Merge group 2



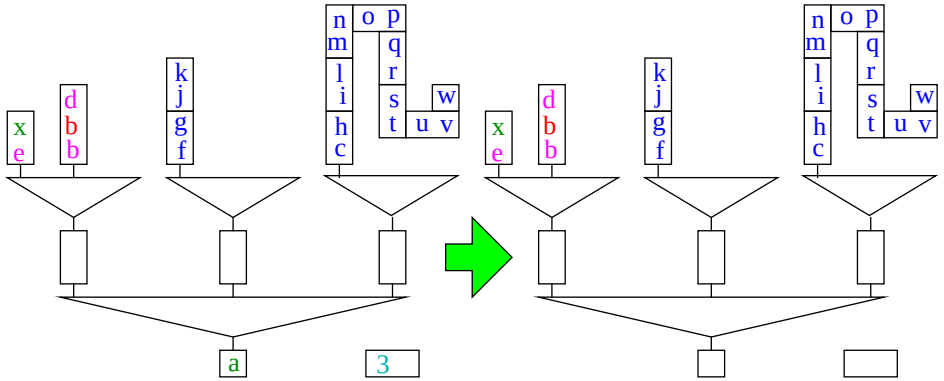
Example

Merge group buffers



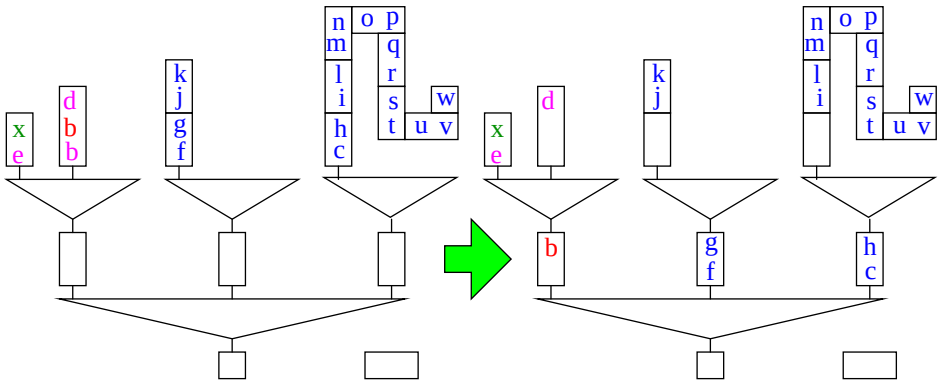
Example

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



Example

DeleteMin $\rightsquigarrow$ b



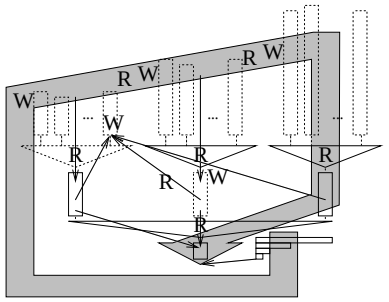
Analysis

- 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 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

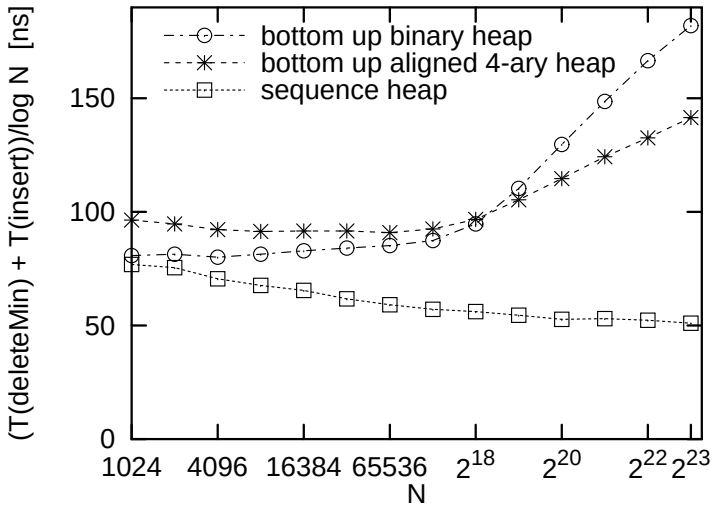
Operation Sequence:

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

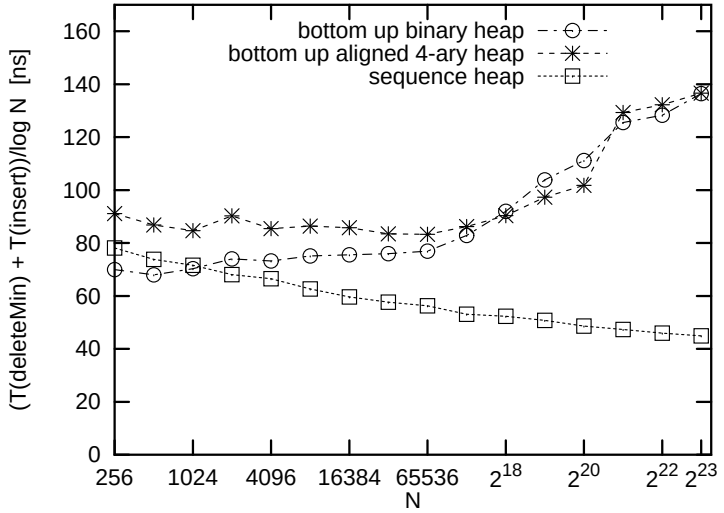
Near optimal performance on all machines tried!

Near optimal : performance on all machines tried!

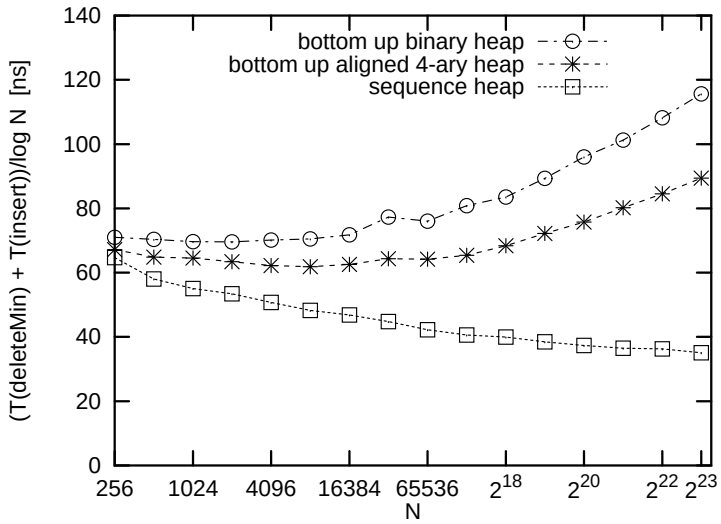
MIPS R10000, 180 MHz



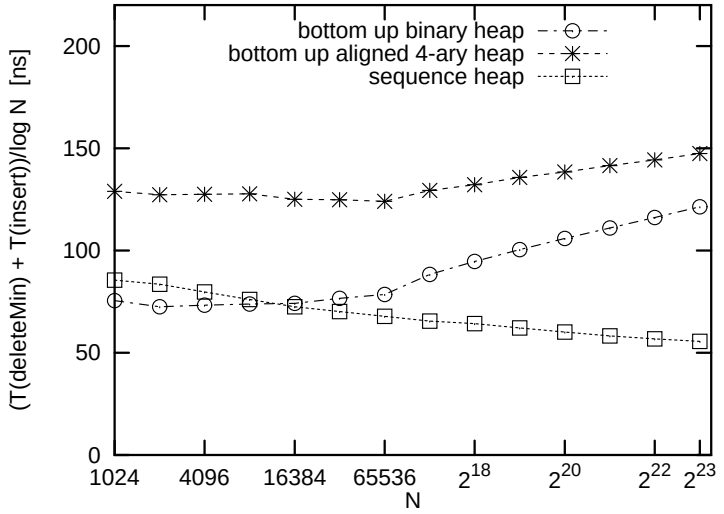
Ultra-SparcIIi, 300 MHz



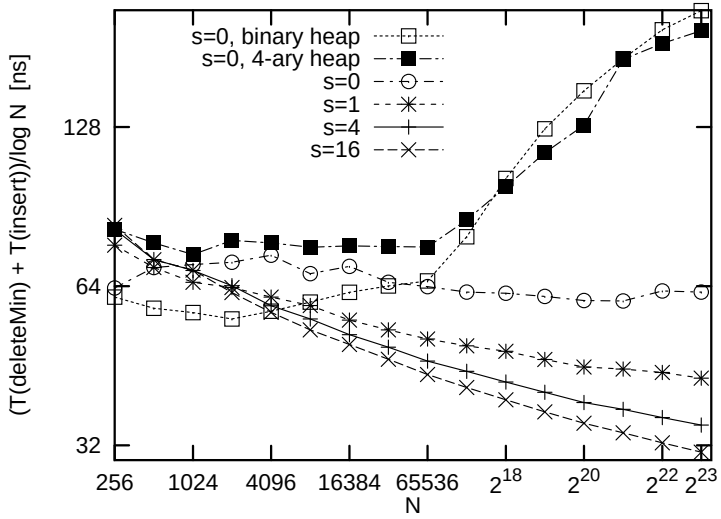
Alpha-21164, 533 MHz



Pentium II, 300 MHz



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



Methodological Lessons

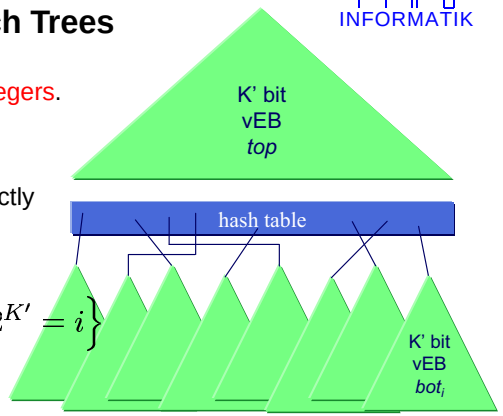
If you want to compare small constant factors in execution time:

- Reproducability 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

- Integrate into STL
- Dependence on size rather than number of insertions
- Parallel disks
- Space efficient implementation
- Multi-level cache aware or cache-oblivious variants

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
- $K' := K/2$
- $M_i := \left\{ x \bmod 2^{K'} : x \operatorname{\mathbf{div}} 2^{K'} = i \right\}$
- root points to nonempty M_i-s
- top $t = \{i : M_i \neq \emptyset\}$
- insert, delete, search in $\mathcal{O}(\log K)$ time

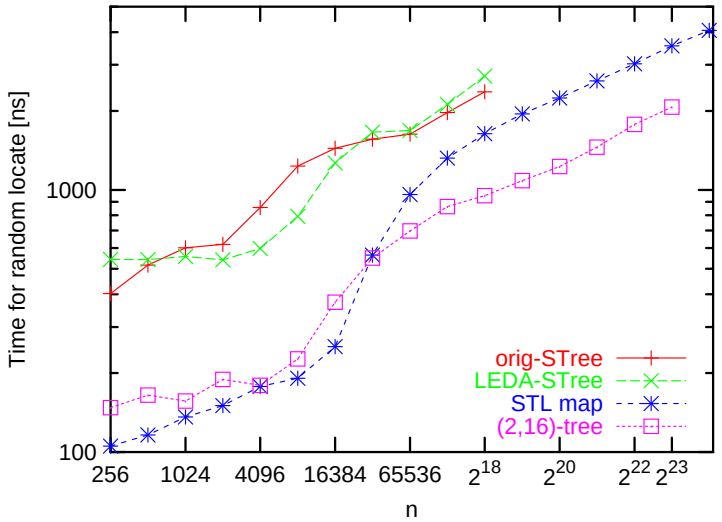
Comparison with Comparison Based Search Trees

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

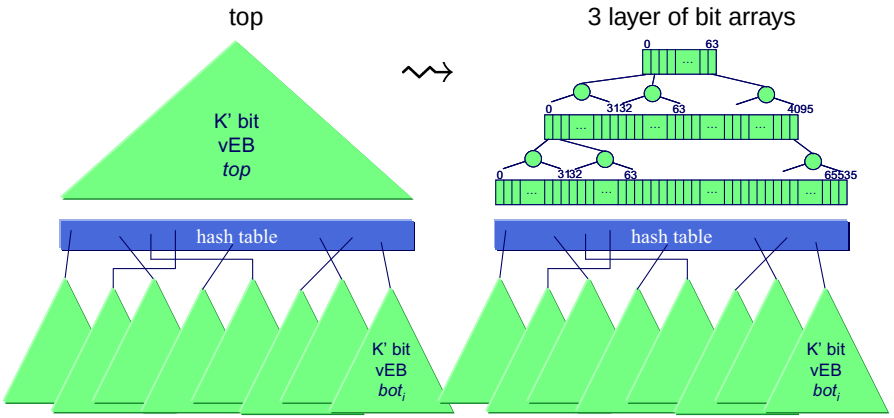
Problems:

Many special case tests

High space consumption

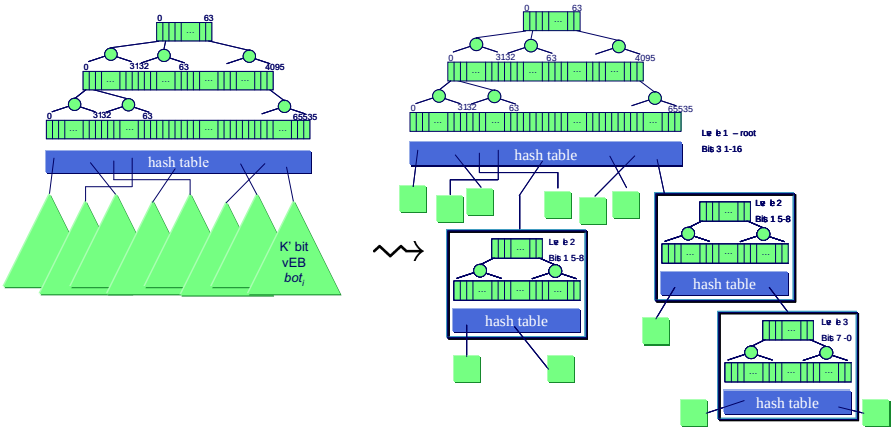


Efficient 32 bit Implementation



Efficient 32 bit Implementation

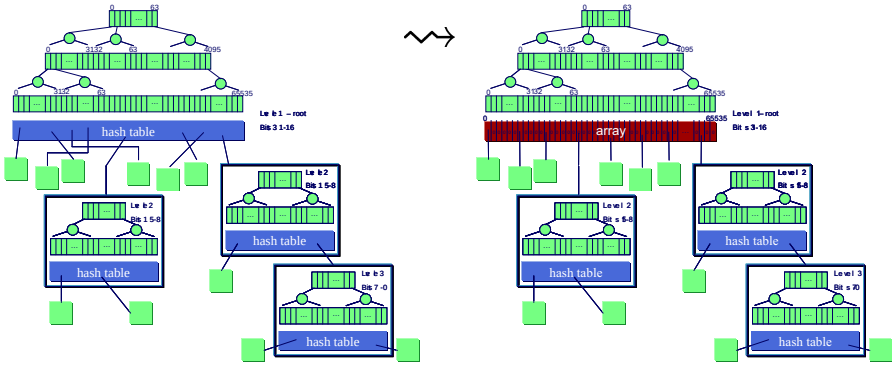
Break recursion after 3 layers



Efficient 32 bit Implementation

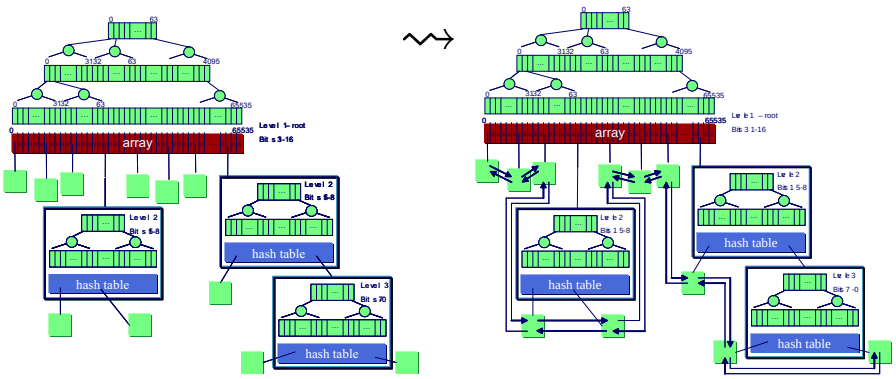
root hash table

root array

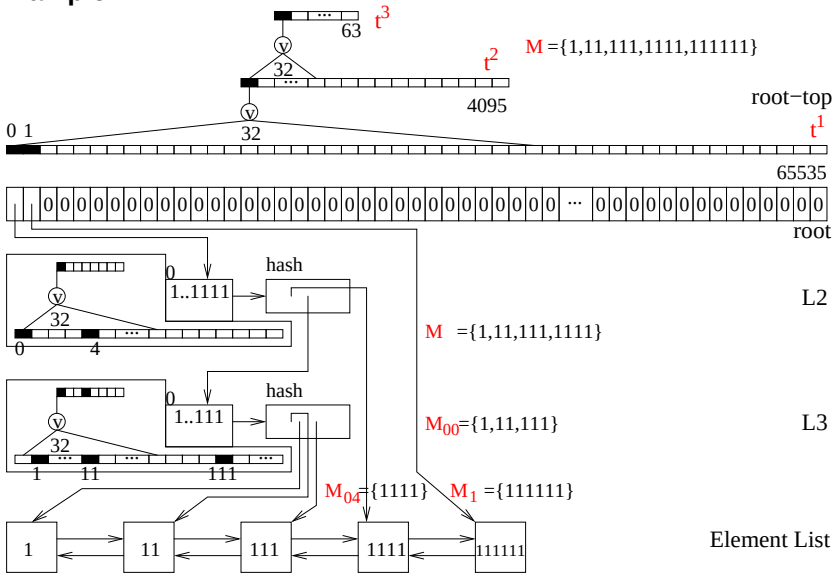


Efficient 32 bit Implementation

Sorted doubly linked lists for associated information and range queries



Example



Locate High Level

(* return handle of $\min x \in M : y \leq x$ *)

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

if $y > \max M$ **then return** ∞

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

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

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

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

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

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

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

$j := i$ **div** n -- which word?

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

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

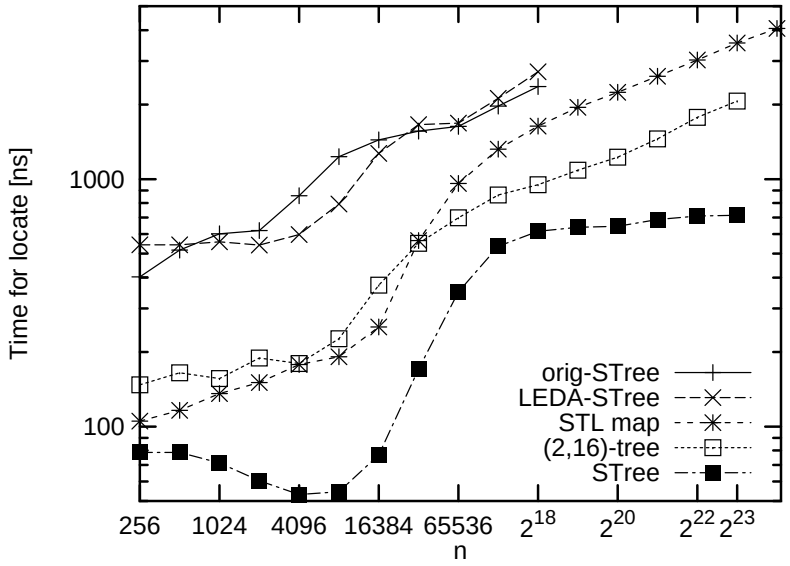
if $a = 0$ **then**

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

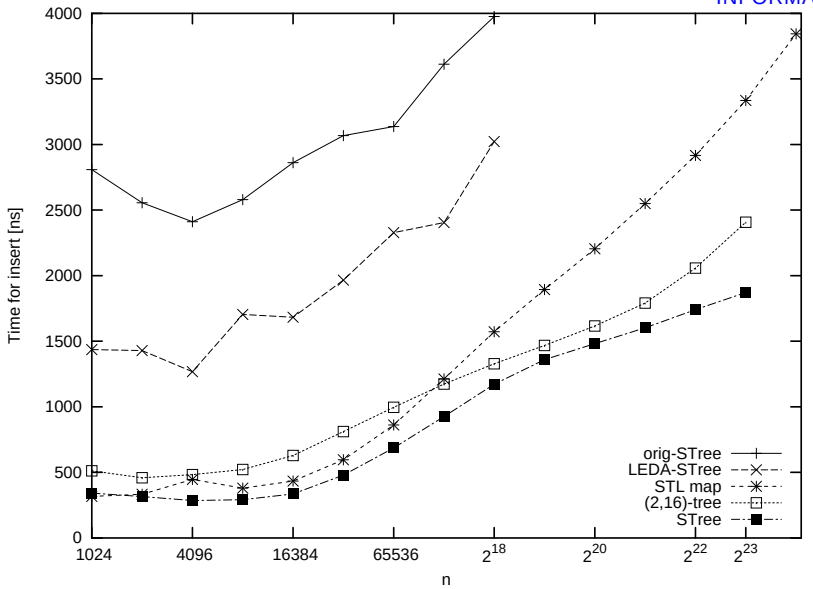
$a := t^k[nj..nj + n - 1]$

return $nj +$ msbPos(a) -- e.g. floating point conversion

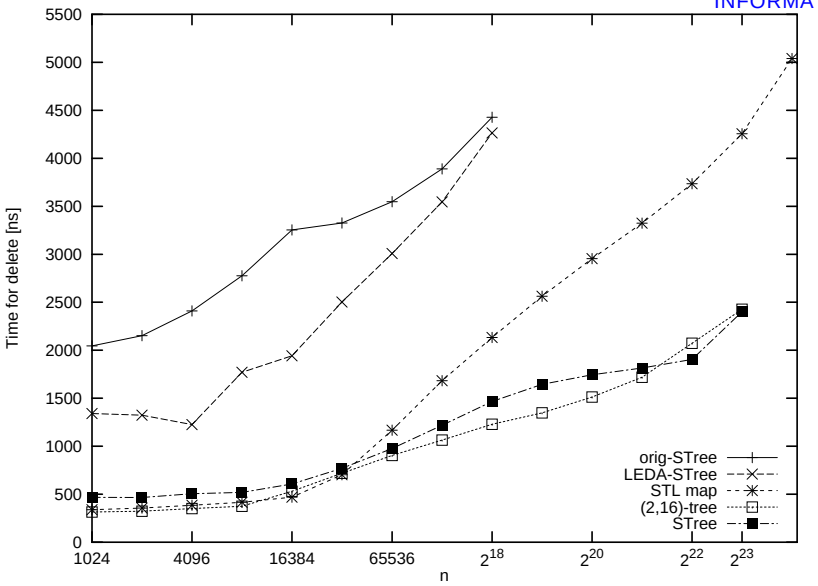
Random Locate



Random Insert



Delete Random Elements



Open Problems

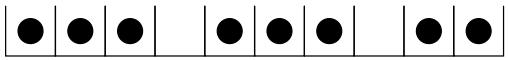
- Measurement for "worst case" inputs
- Measure Performance for realistic inputs
  - IP lookup etc.
  - Best first heuristics like, e.g., bin packing
- More space efficient implementation
- (A few) more bits

5 Space Efficient Hashing with Worst Case Constant Access Time

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

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

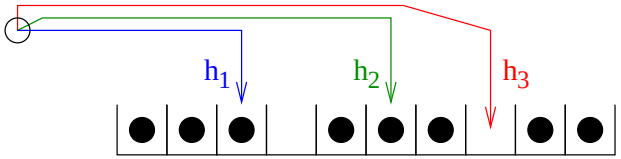
Assume a truly random hash function h



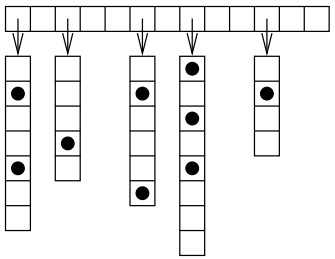
Related Work

Uniform hashing:

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



Dynamic Perfect Hashing,
[Dietzfelbinger et al. 94]
Worst case constant time
for lookup but ϵ is not small.



Approaching the Information Theoretic Lower Bound:
[Brodnik Munro 99,Raman Rao 02]
Space $(1+o(1))\times$lower bound without associated information
[Pagh 01] static case.

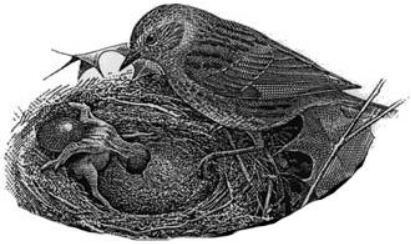
Cuckoo Hashing

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

Two choices for each element.

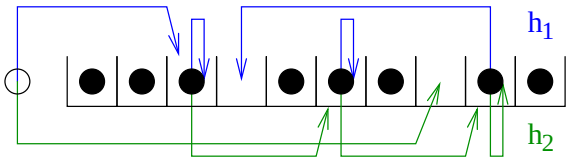
Insert moves elements;

rebuild if necessary.



Very fast lookup and insert.

Expected constant insertion time.



d-ary Cuckoo Hashing

d choices for each element.

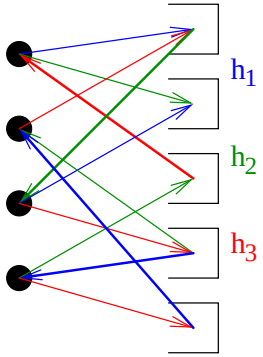
Worst case d probes for delete and lookup.

Task: maintain perfect matching

in the bipartite graph

$(L = \text{Elements}, R = \text{Cells}, E = \text{Choic}^{\text{es}})$,

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

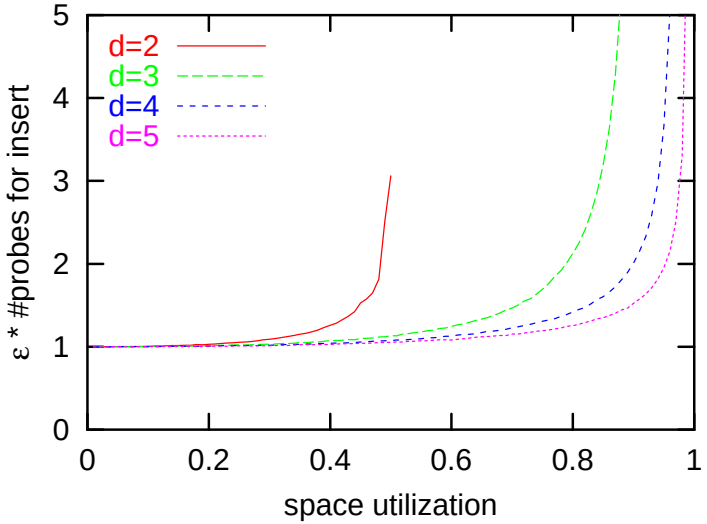


Tradeoff: Space $\leftrightarrow$ Lookup/Deletion Time

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

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

Experiments



Open Questions and Further Results

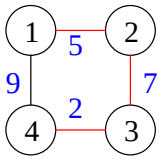
- Tight analysis of insertion
- Two choices with d slots each [Dietzfelbinger et al.]
  $\rightsquigarrow$ cache efficiency

Good Implementation?

- Automatic rehash
- Always correct

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

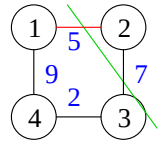
- Basics: Edge property and cycle property
- Jarník-Prim Algorithm
- Kruskals Algorithm
- Some tricks and comparison
- Advanced algorithms using the cycle property
- External MST

Applications: Clustering; subroutine in combinatorial optimization, e.g.,
Held-Karp lower bound for TSP. Challenging real world instances???
Anyway: almost ideal "fruit fly" problem

Selecting and Discarding MST Edges

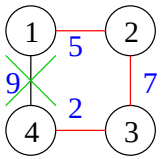
The Cut Property

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 a C can be used in an MST.



The Cycle Property

The heaviest edge on a cycleis not needed for an MST



The Jarník-Prim Algorithm [Jarník 1930, Prim 1957]

Idea: grow a tree

T:= $\emptyset$

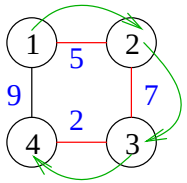
S:= $\{s\}$ for arbitrary start node s

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

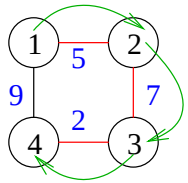
foreach $(u,v) \in E$ **do**

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

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

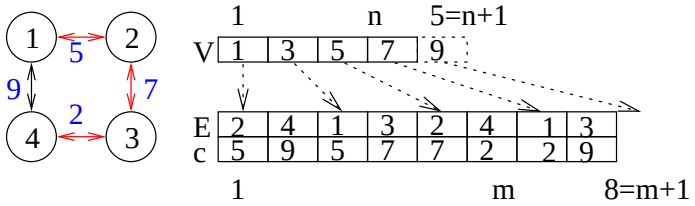
if $v \in q$ **then** q.decreaseKey(v) **else** q.insert(v)

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

The Union-Find Data Structure

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

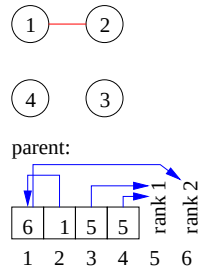
    assert i and j are leaders of different subsets

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

    elsif 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.find(u)

    $v' :=$ T.find(v)

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

      output (u,v)

      T.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

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

Kruskal versus Jarník-Prim I

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

Better Version For Dense Graphs ?

Procedure quickKruskal(E : Sequence **of** Edge)
  if $m \leq \beta n$ **then** kruskal(E) -- for some constant β
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq E \rangle$ -- partitioning a la
    $E_{>} := \langle e \in E : e > E \rangle$ -- quicksort
    quickKruskal($E_{\leq}$)
    $E'_{>} :=$ filter($E_{>}$)
    quickKruskal($E'_{>}$)

Function filter(E)
  make sure that leader$[i]$ gives the leader of node i -- $\mathcal{O}(n)$!
  return $\langle (u,v) \in E$: leader$[u] \neq$ leader$[v] \rangle$

6.1 Attempted Average-Case Analysis

Assume different random edge weights, arbitrary graphs

Assume pivot p has median weight

Let $T(m)$ denote the expected execution time for m edges

$m \leq \beta n$: $\mathcal{O}(n \log n)$

Partitioning, Filtering: $\mathcal{O}(m+n)$

$m > \beta n$: $T(m) = \Omega(m) + T(m/2) + T(2n)$ [Chan 98]

Solves to $\mathcal{O}\left(m + n\log(n) \cdot \log \frac{m}{n}\right) \leq \mathcal{O}(m + n\log(n) \cdot \log\log n)$

Open Problem: I know of no graph family with $T(n) = \omega(m + n\log(n))$

Kruskal versus Jarník-Prim II

Things are even less clear.

Kruskal may be better even for dense graphs

Experiments would be interesting.

Even for artificial graphs.

6.2 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$((L \cup F))$

6.2.1 Analysis

[Chan 98, KKK 95]

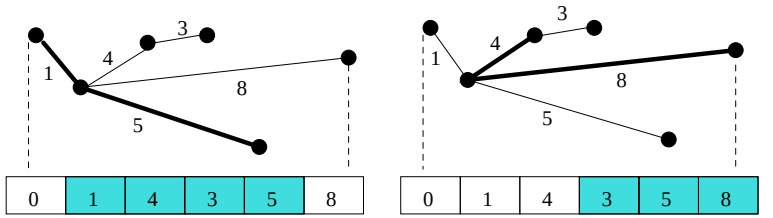
Observation: $e \in L$ only if $e \in \mathrm{MST}(R \cup \{e\})$.

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

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

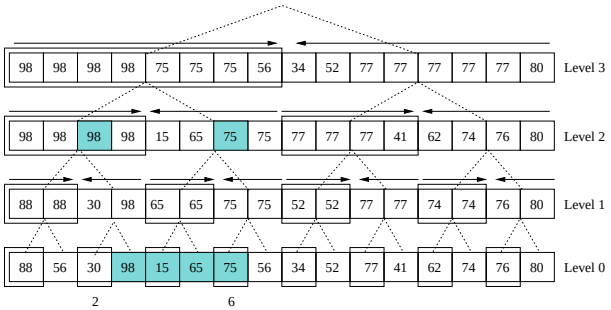
MST Verification by Interval Maxima

- 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: $2 \oplus 6 = 010 \oplus 110 = 100 > \ell = 2$

A Simple Filter Based Algorithm

Choose $r = \sqrt{mn}$.

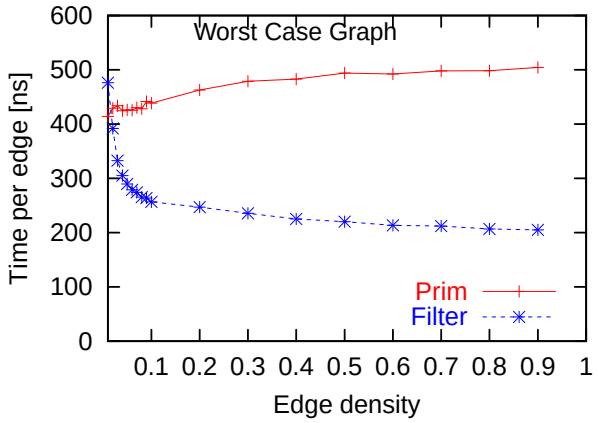
We get expected time

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

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

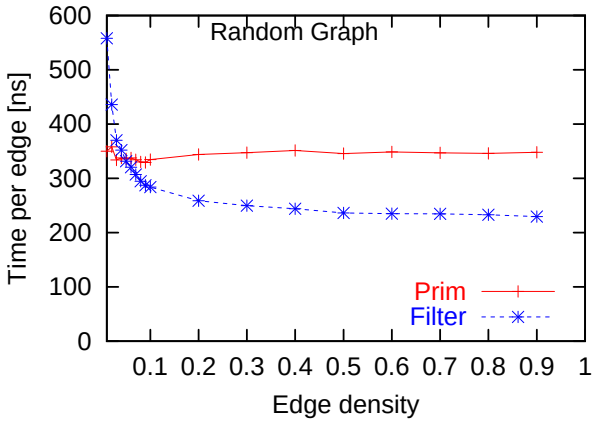
Results

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

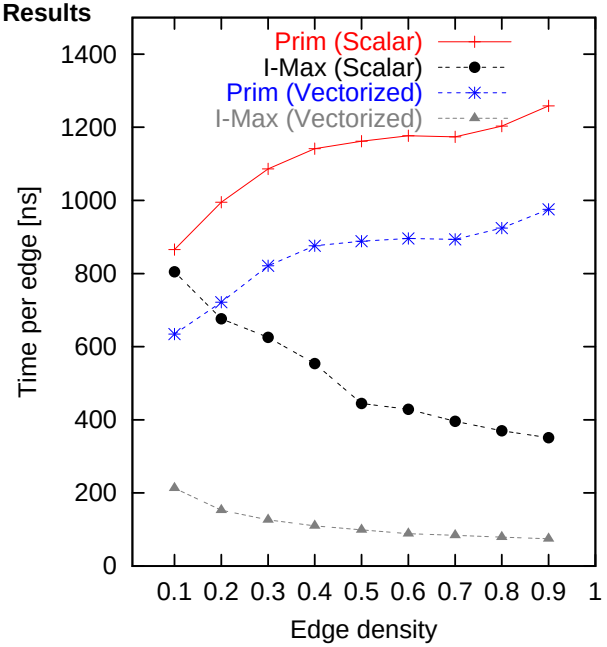


Results

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



Results



10 000 nodes,

NEC SX-5

Vector Machine

"worst case"

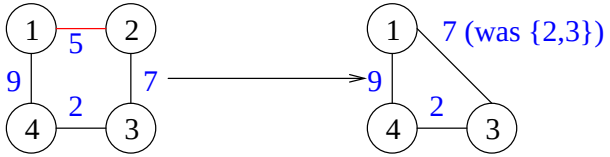
Edge Contraction

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

Eliminate v:

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

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



Boruvka's Node Reduction Algorithm

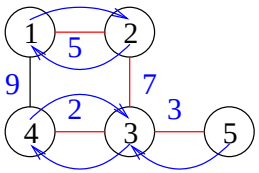
For each edge find the lightest incident edge.

Include them into the MST (cut property)

contract these edges,

Time $\mathcal{O}(m)$

At least halves the number of remaining nodes



Simpler and Faster Node Reduction

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

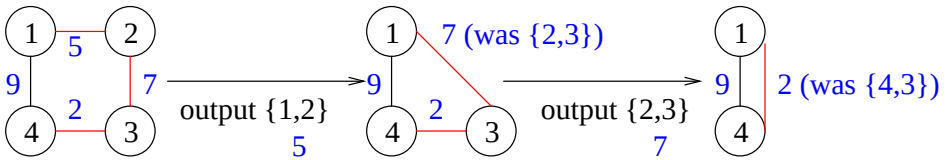
pick a random node v

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

contract (u,v)

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

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



6.3 Randomized Linear Time Algorithm

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

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

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:

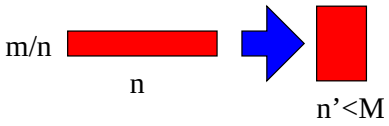
T:= $\emptyset$ -- current approximation of MST
while there are any unprocessed edges **do**
  load any $\Theta(M)$ unprocessed edges E'
  T:= MST$(T \cup E')$ -- for any internal MST alg.

Corollary: we can do it with linear expected internal work

Disadvantages to Kruskal:
Slower in practice
Smaller max. n

General External MST

while $n > M - 2B$ **do**
  perform some node reduction
use semi-external Kruskal



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

External Implementation I: Sweeping

π : random permutation $V \to V$

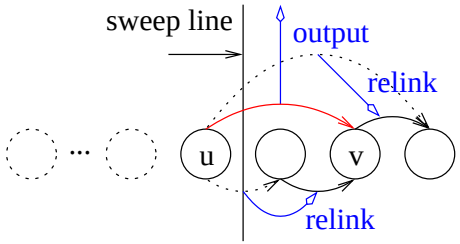
sort edges (u,v) by $\min(\pi(u),\pi(v))$

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

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

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

contract (u,v)



Problem: how to implement relinking?

Relinking Using Priority Queues

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

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

current := $n+1$

loop

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

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

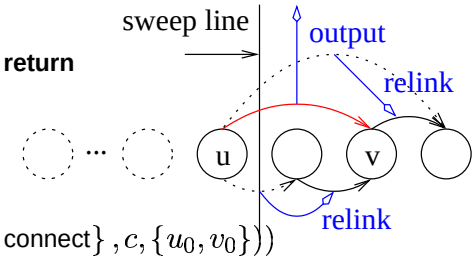
if current$= M+1$ **then return**

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

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

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

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

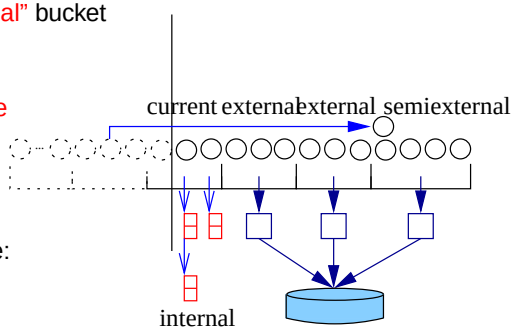


$\approx$ sort$(10m \ln \frac{n}{M})$ I/Os with opt. priority queues [Sanders 00]

Problem: Compute bound

Sweeping with linear internal work

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



Sweeping:

Scan current internal bucket twice:

1. Find minimum
2. Relink

New external bucket: scan and put in internal buckets

Large degree nodes: move to semiexternal bucket

Experiments

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

- sparse random graphs
- random geometric graphs
- grids

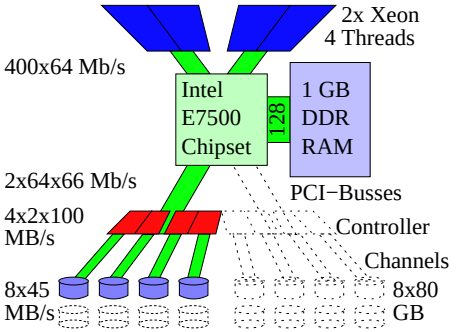
  $\mathcal{O}(\text{sort}(m))$ I/Os

  for planar graphs by

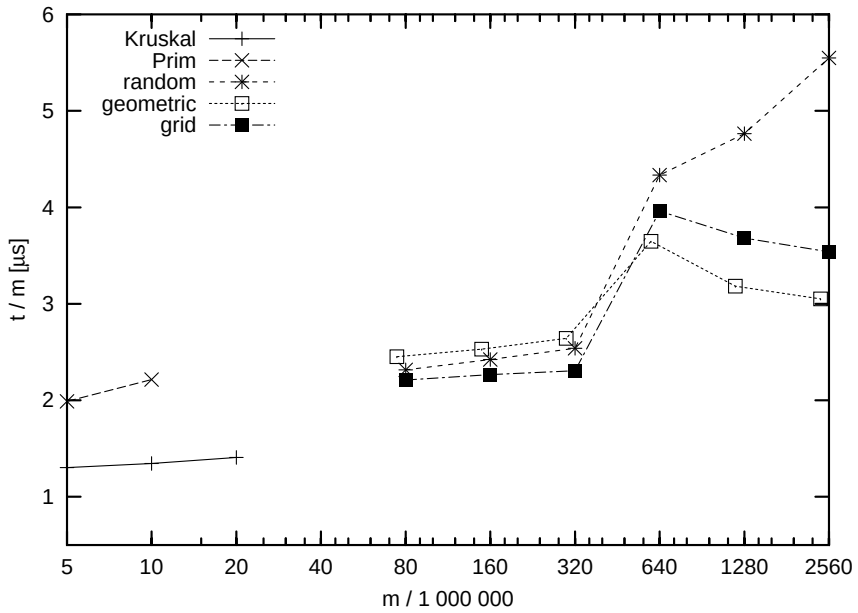
  removing parallel edges!

Other instances are rather dense

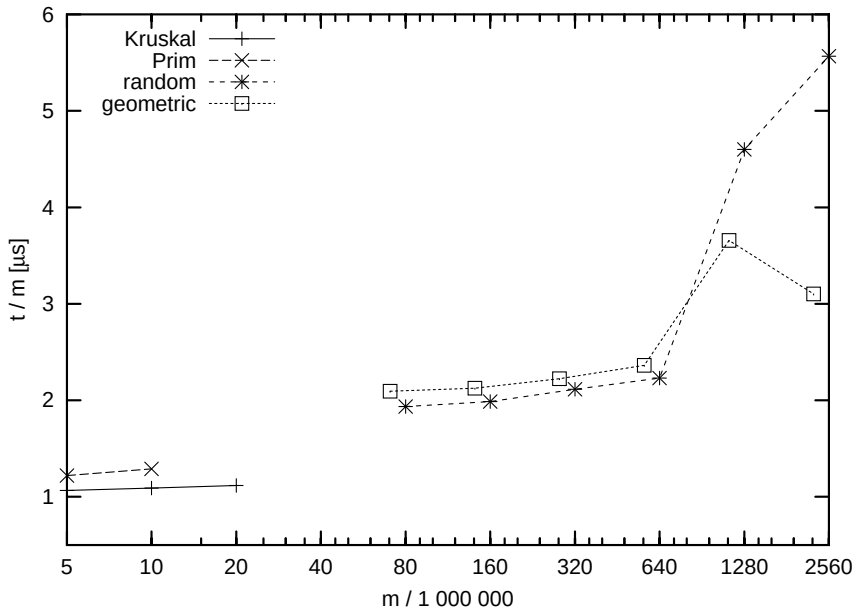
or designed to fool specific algorithms.



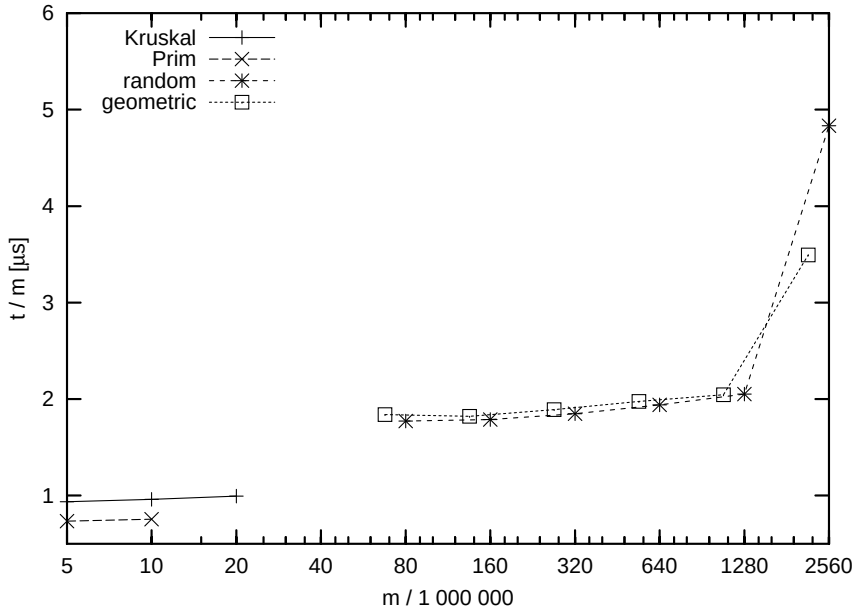
$m \approx 2n$



$m \approx 4n$



$m \approx 8n$



MST Summary

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

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

External MST is feasible

Open Problems

- New experiments for (improved) Kruskal versus Jarník-Prim
- Realistic (huge) inputs
- Parallel external algorithms
- Implementations for other graph problems

Conclusions

- Even fundamental, “simple” algorithmic problems still raise interestin 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

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 Tr"aff WAE 98]:

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

Ergebnis

- Einfach extern implementierbar
- $n' = M \rightsquigarrow$ semiexterner Kruskal Algorithmus
- Insgesamt $\mathcal{O}\left(\text{sort}(m \ln \frac{n}{m})\right)$ erwartete I/Os
- F"ur realistische Eingaben mindestens $4\times$ bisher als bisher bekannte Algorithmen
- Implementierung in `<stxxl>` mit bis zu 96 GByte grossen Graphen l"auft ""uber Nacht"