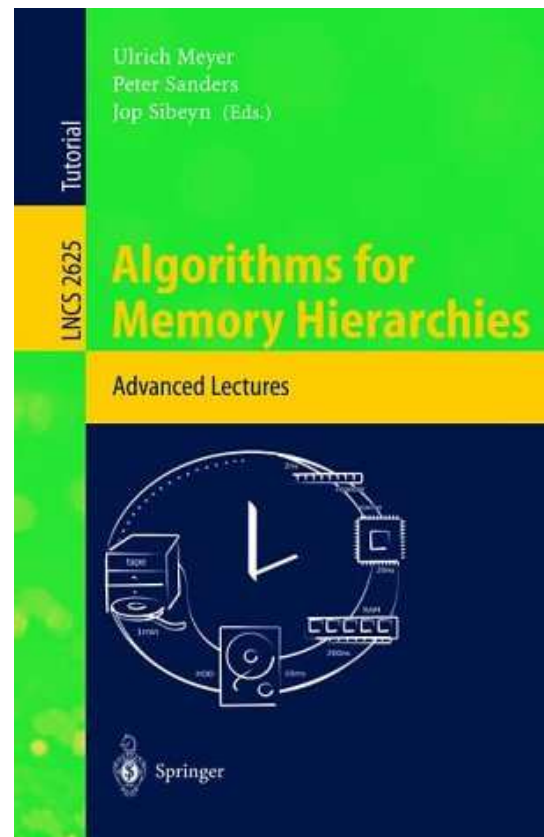


Algorithms for Memory-Hierarchies

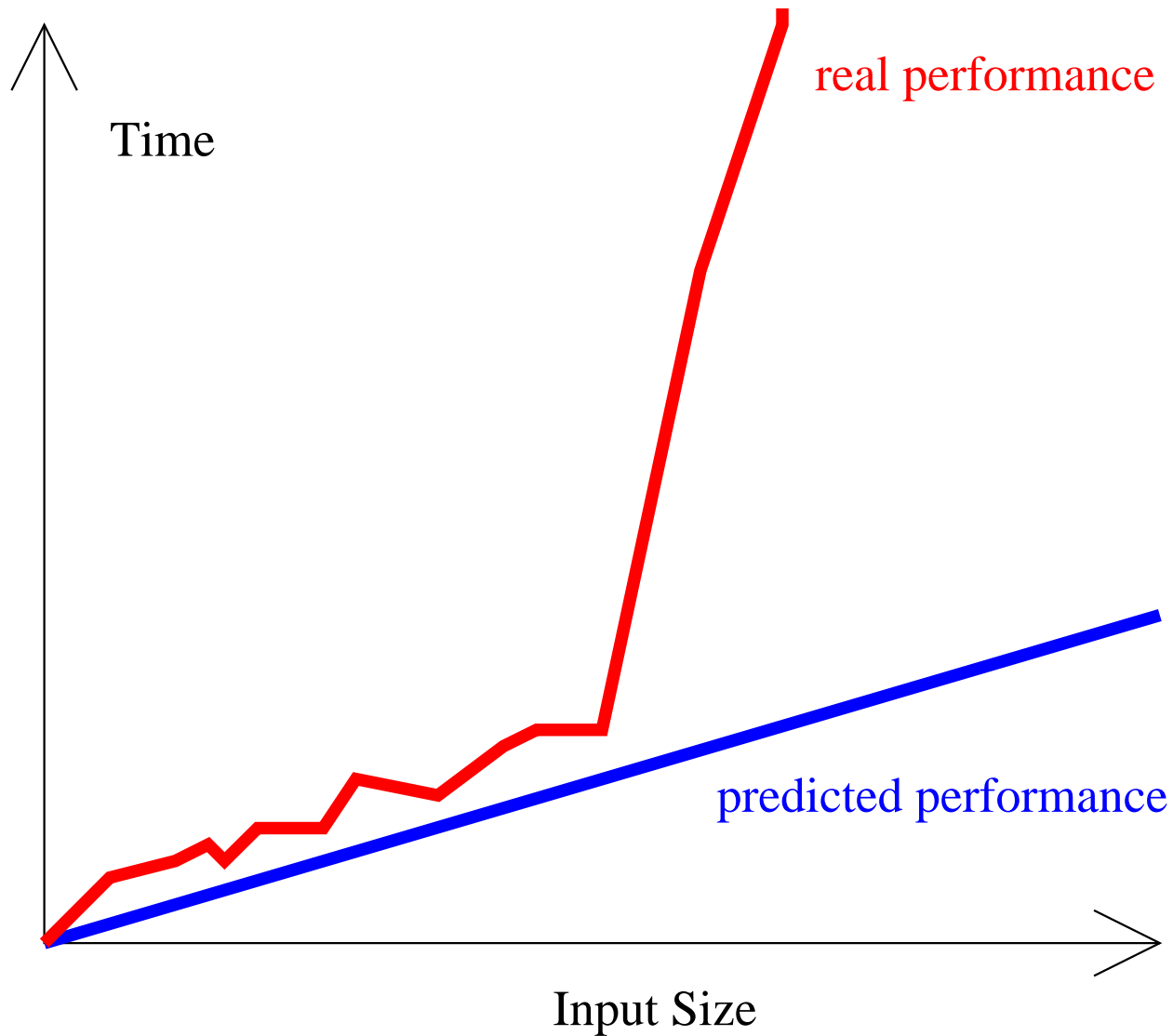
Ulrich Meyer

Max-Planck-Institut für Informatik

www.uli-meyer.de



The Problem: RAM Model vs. Real Computer



RAM Model

In first year course:

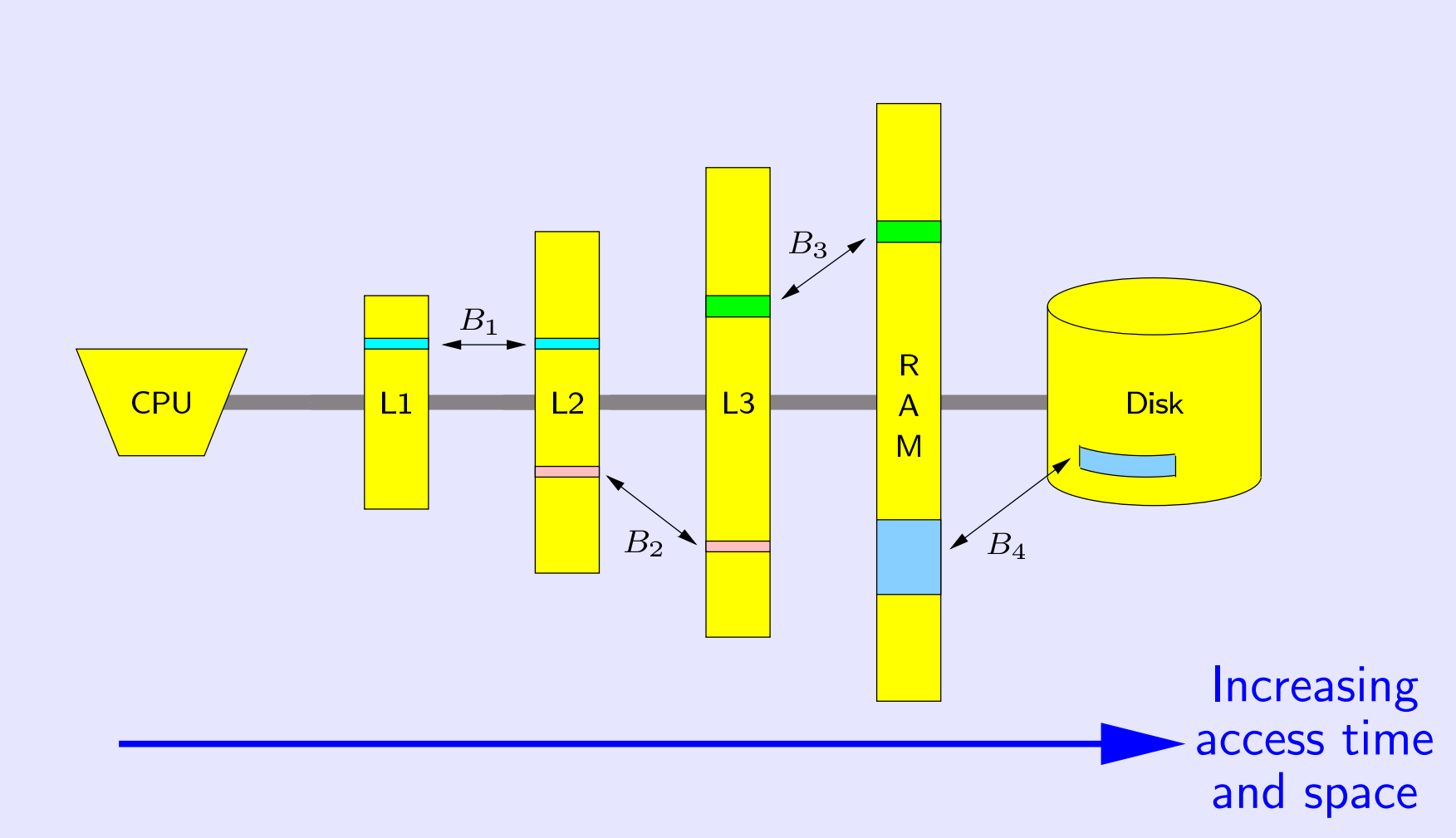
- ▶ Computer \approx CPU + Memory
- ▶ Uniform cost model: each access and each operation one unit of time

Impact:

- ▶ Programming languages based on it, convenient
- ▶ Crucial for success of computer industry

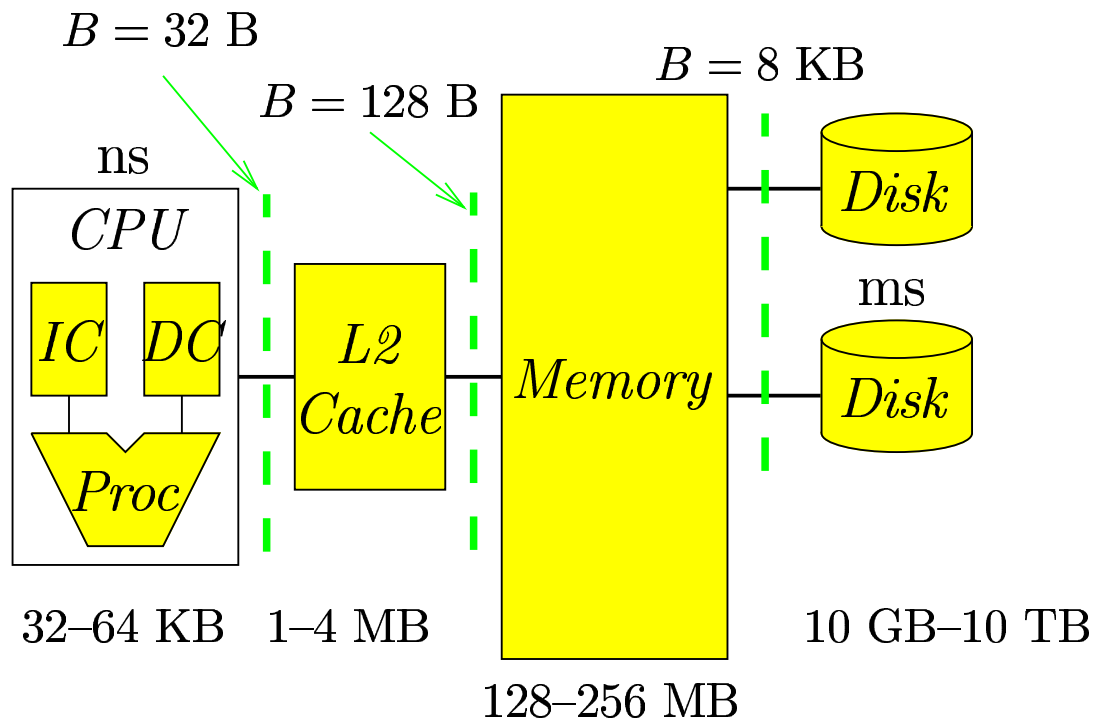
BUT: Modern computers have a **HIERARCHY** of memory.

Modern Computer



Why Memory Hierarchies?

- ▶ Why: **purely economic reasons !!!**
- ▶ faster \sim more expensive \rightarrow as few expensive pieces as possible.



	Latency	Relative to CPU
Register	0.5 ns	1
L1 cache	0.5 ns	1-2
L2 cache	3 ns	2-7
DRAM	150 ns	80-200
TLB	500+ ns	200-2000
Disk	10 ms	10^7

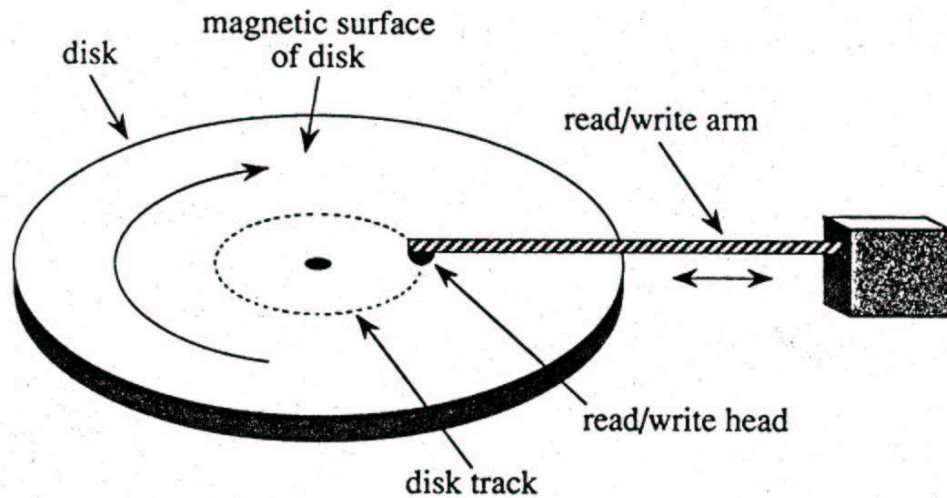
- ▶ **Hard disks can be the ultimate performance killers.**

Trends

Parameter	Yearly Improvement Rate
Disk Latency	10 %
Disk Bandwidth	20 %
Processor Speed	55 %
RAM Bandwidth	40 %
RAM Capacity/Cost	45 %

- ▶ Performance gap is increasing.
- ▶ RAM Capacity doubling about every two years but users doubling data storage about every 5 months (frequently copying everything).
- ▶ Results in I/O Bottleneck.

Why are Hard Disks such slow?



Components of disk access time:

- ▶ Seek time (milliseconds, SLOW)
- ▶ Rotational latency (milliseconds, SLOW)
- ▶ Read access (nanoseconds, FAST)

Reading many consecutive data items takes not much longer than reading a single data item.

How the Operation System tries to make up for it.

Virtual Memory provides the look of the uniform model.

But not necessarily the performance !!!

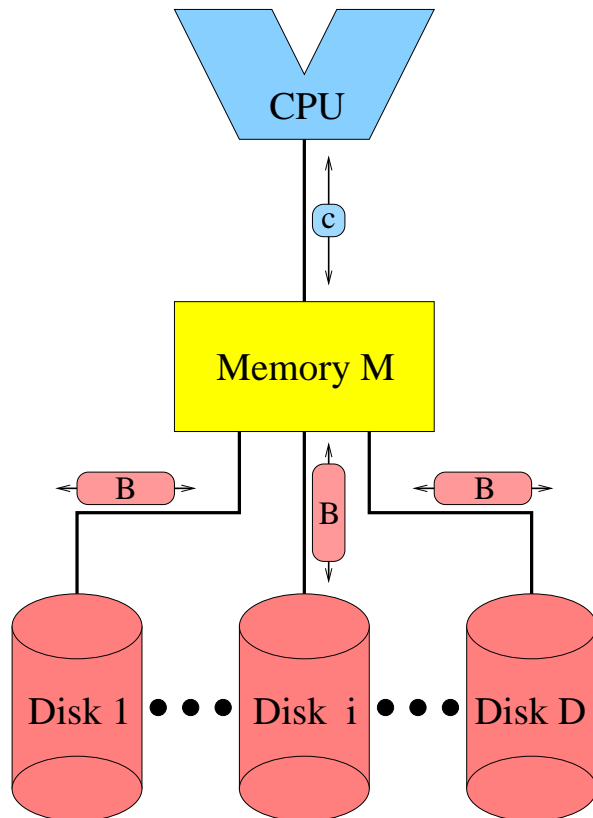
Additionally:

- ▶ Disk partitioned into blocks of ≥ 512 Bytes.
- ▶ Every disk access reads or writes a whole block.
- ▶ Read ahead.

This helps in special cases.

For most interesting algorithms this does not help at all.

External Memory (EM) Modell [VitterShriver94]



- ▶ Main memory size $M \ll$ Problem size N
- ▶ External memory = D disks (here $D=1$)
- ▶ Data is transferred in blocks of size B ($\sim 10^5$)
- ▶ Up to $\leq D \cdot B$ data per I/O step ($\sim 10^2$ per sec.)
- ▶ **Goal: Minimize number of I/O steps**
- ▶ $\text{scan}(x) := \mathcal{O}\left(\frac{x}{D \cdot B}\right)$ I/Os.
- ▶ $\text{sort}(x) := \mathcal{O}\left(\frac{x}{D \cdot B} \cdot \log_{M/B} \frac{x}{B}\right)$ I/Os.

- ▶ **Cache-Oblivious (CO) Modell:** M, B unknown (to the algorithm).
- ▶ Good on **one** memory level \Rightarrow good on **all** memory levels.
- ▶ **One** algorithm for **all** platforms ?!

Discussion of the EM-Model

POSITIVE:

- ▶ simple
- ▶ generally accepted
- ▶ concentrates on key difficulties
- ▶ has already led to some good implementations

NEGATIVE:

- ▶ concentration on I/O may yield compute-bound algorithms
- ▶ no distinction between structured I/O and unstructured I/O
- ▶ M and B must be known
- ▶ assumes full control over I/O

How to make algorithms I/O-efficient?

Only a few golden rules:

- ▶ Avoid unstructured access patterns.
- ▶ Incorporate LOCALITY directly into the algorithm.

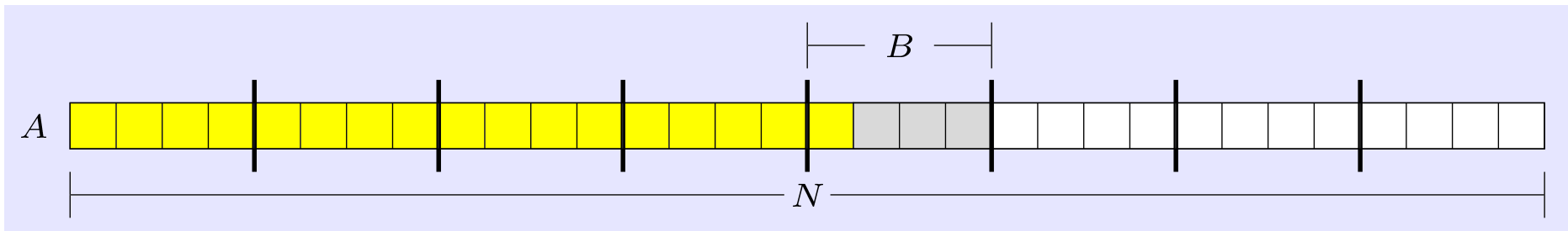
Tools:

- ▶ Scanning.
- ▶ Sorting.
- ▶ Special I/O-efficient data structures.
- ▶ "Simulation" of parallel algorithms.

Warmup: Scanning

```
sum = 0;  
for i=1 to N do sum := sum + A[i];
```

$\text{scan}(N) = \mathcal{O}(N/B)$ I/Os, optimal.



Remarks:

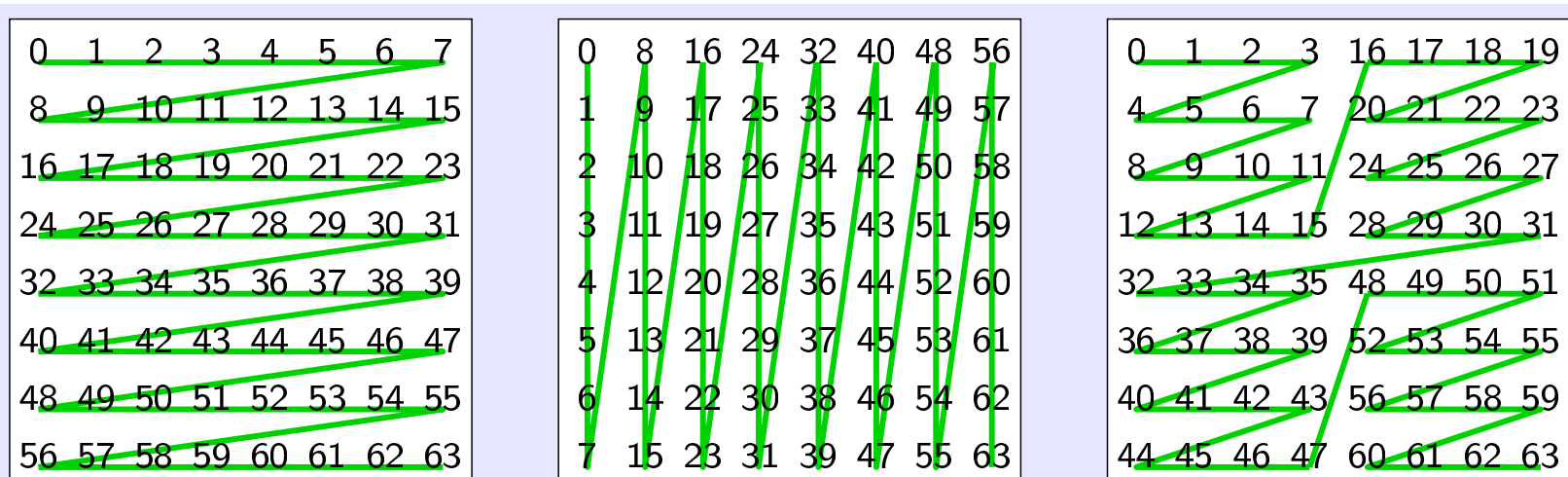
- ▶ No need to know B here.
- ▶ Scanning backwards would be slower in practice.

Matrix Multiplication

Problem:

$$Z = X \cdot Y, \quad z_{ij} = \sum_{k=1}^N x_{ik} \cdot y_{kj}$$

Layout of matrices:



Row major Column major 4 × 4-blocked

Matrix Multiplication

Algorithm 1: Nested loops

- ▶ Row major
- ▶ Reading a **column of Y** $\Rightarrow N$ I/Os
- ▶ Total $\mathcal{O}(N^3)$ I/Os

```
for  $i = 1$  to  $N$ 
  for  $j = 1$  to  $N$ 
     $z_{ij} = 0$ 
    for  $k = 1$  to  $N$ 
       $z_{ij} = z_{ij} + x_{ik} \cdot y_{kj}$ 
```

Algorithm 2: Blocked algorithm

- ▶ Partition X and Y into blocks of size $s \times s$, $s = \Theta(\sqrt{M})$.
- ▶ Apply Algorithm 1 to $N/s \times N/s$ matrices; elements are $s \times s$ sub-matrices.
- ▶ Use $s \times s$ -blocked layout.

$\mathcal{O}((N/s)^3 \cdot s^2 / B) = \mathcal{O}(N^3 / (s \cdot B)) = \mathcal{O}(N^3 / (B \cdot \sqrt{M}))$ I/Os, optimal.

Simple I/O-Efficient Data Structures (1)

FIFO-Queue:

- ▶ Maintain an **input buffer** and an **output buffer** (each of size B) in memory.
- ▶ **Insert**: put new element into input buffer;
if buffer now full, write to disk.
- ▶ **Remove**: take element from output buffer (if empty from input buffer);
if buffer now empty, read next block from disk.

I/O-complexity of Insert/Remove:

- ▶ **Best-Case**: O I/Os
- ▶ **Worst-Case**: 1 I/O
- ▶ **Amortized**: $1/B$ I/Os

Simple I/O-Efficient Data Structures (2)

Stack:

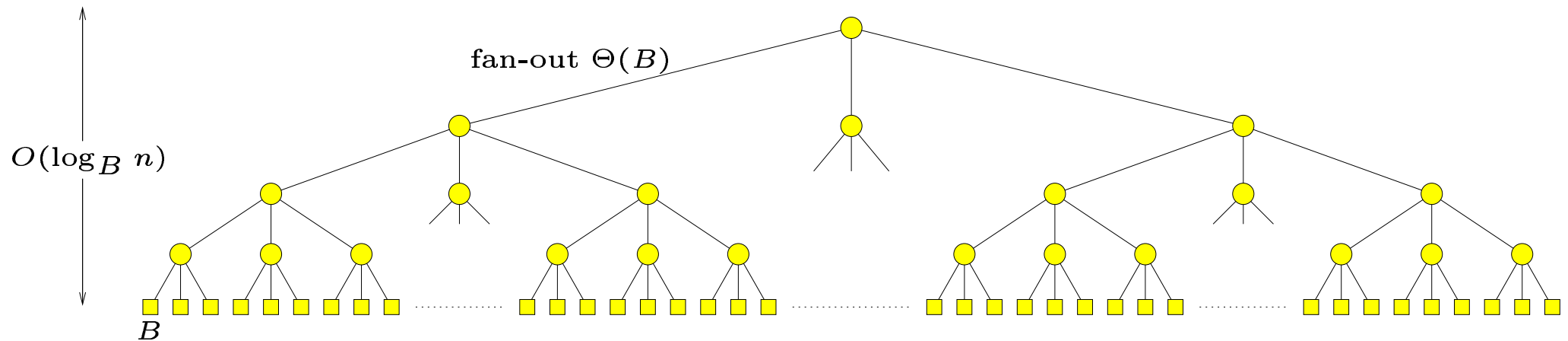
- ▶ Maintain an **combined input/output buffer** of size $2 \cdot B$ in memory.
- ▶ **Push**: Insert new element into buffer;
if buffer now full, write **bottom B elements** to disk.
- ▶ **Pop**: remove top element from buffer;
if buffer now empty, read next block from disk.

I/O-complexity of Push/Pop:

- ▶ **Best-Case**: O I/Os
- ▶ **Worst-Case**: 1 I/O
- ▶ **Amortized**: $1/B$ I/Os

Obs: After an I/O, the buffer contains exactly B elements.

Searching: BTree



Each B -tree node fits in a disk block:

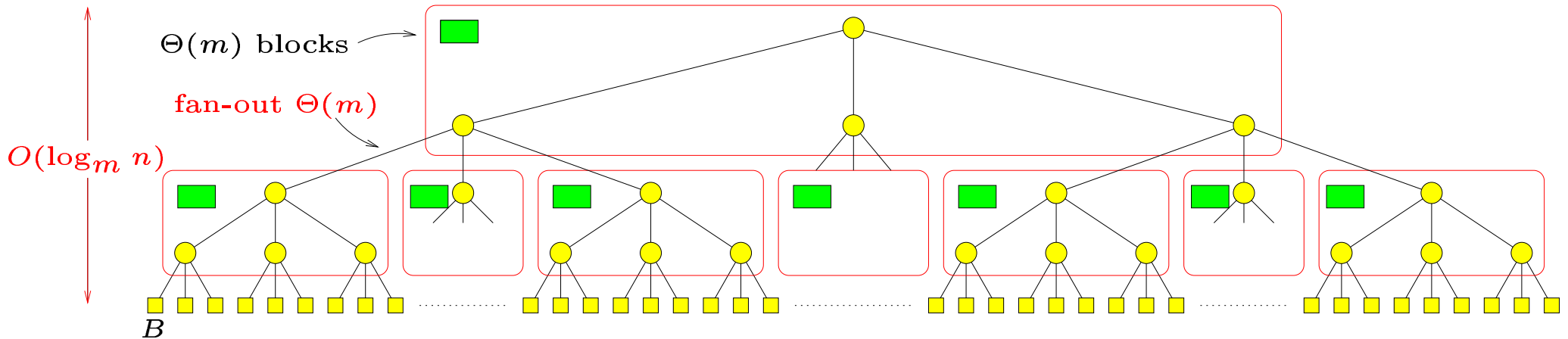
⇒ Insert, Delete, DeleteMin, Query in $\mathcal{O}(\log_B N)$ I/Os.

Optimal if each operation is handled individually.

Building B -tree:

- Repeated insertion ⇒ $\mathcal{O}(N \cdot \log_B N)$ -I/O algorithm
- Can be done more efficiently ...

Buffer Tree [Arge95]



Main idea: Logically group nodes together and add buffers

- ▶ Insertions are done "lazily" – items inserted into buffers.
- ▶ When a buffer runs full, its items are pushed one level down.
- ▶ Buffer-emptying in $m = \mathcal{O}(M/B)$ I/Os.

⇒ every block touched $\mathcal{O}(1)$ times on each level.

⇒ inserting N items in $\mathcal{O}(N/B \cdot \log_{M/B} N/B)$ I/Os.

Can be modified to obtain I/O-efficient DeleteMin, too.

I/O-efficient DecreaseKey operations?

Emulate DecreaseKey by Delete and (Re-)Insert ??

Must know old priority; often $\Omega(1)$ I/Os to learn this info.

Alternative: I/O-Efficient Tournament Trees (TTs) [KS96]:

$\leq n$ entries $\langle x_i, k_i \rangle$, $x_i \neq x_j \in \{1, \dots, n\}$, n fixed.

(a) *deletemin()*:

extract the element $\langle x, k \rangle$ with smallest priority k ;
replace it by the new entry $\langle x, \infty \rangle$.

(b) *delete(x)*:

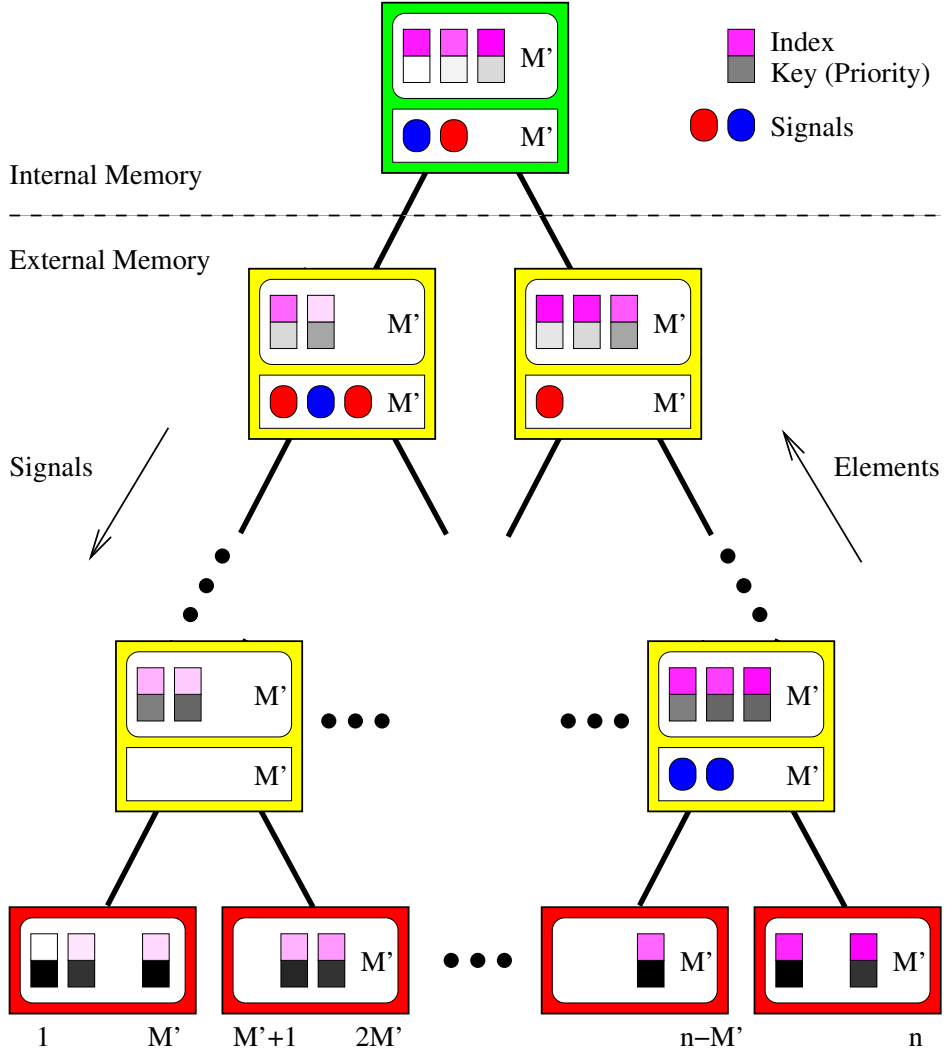
replace $\langle x, oldpriority \rangle$ by $\langle x, \infty \rangle$.

(c) *update(x, newpriority)*:

replace $\langle x, oldpriority \rangle$ by $\langle x, newpriority \rangle$ if $newpriority < oldpriority$.

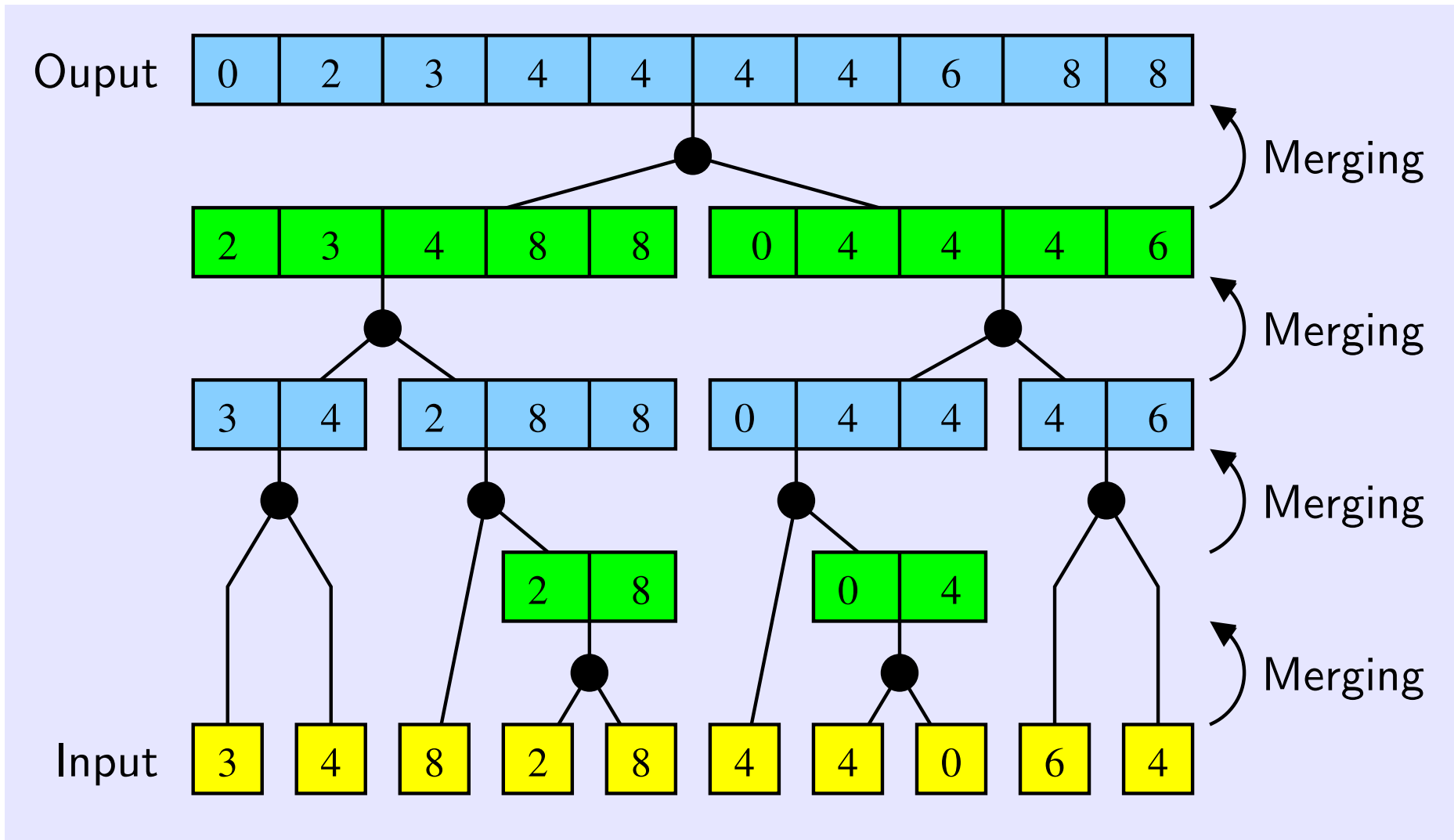
\rightsquigarrow No need to know *oldpriority* for (b) & (c) !

Tournament Trees



Sequence of z *delete(min)/update* operations: $\mathcal{O}\left(\frac{z}{B} \cdot \log_2 \frac{n}{B}\right)$ I/Os amortized.

SORTING: Internal-Memory Mergesort



I/O-Efficient Mergesort

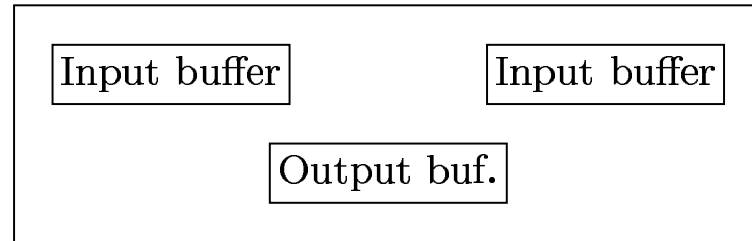
First input stream

Second input stream

(a)

2	4	7	8	12	16	19	27	37	44	48	61
---	---	---	---	----	----	----	----	----	----	----	----

1	3	5	11	17	21	22	35	40	55	57	62
---	---	---	----	----	----	----	----	----	----	----	----

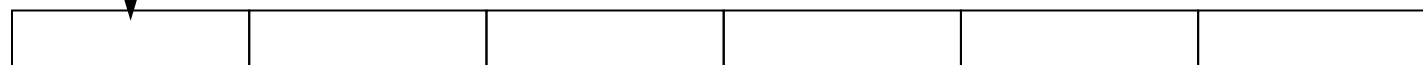
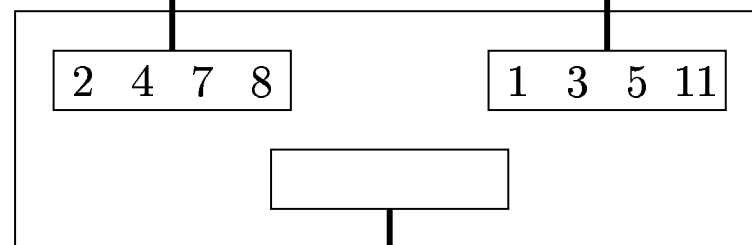


Output stream

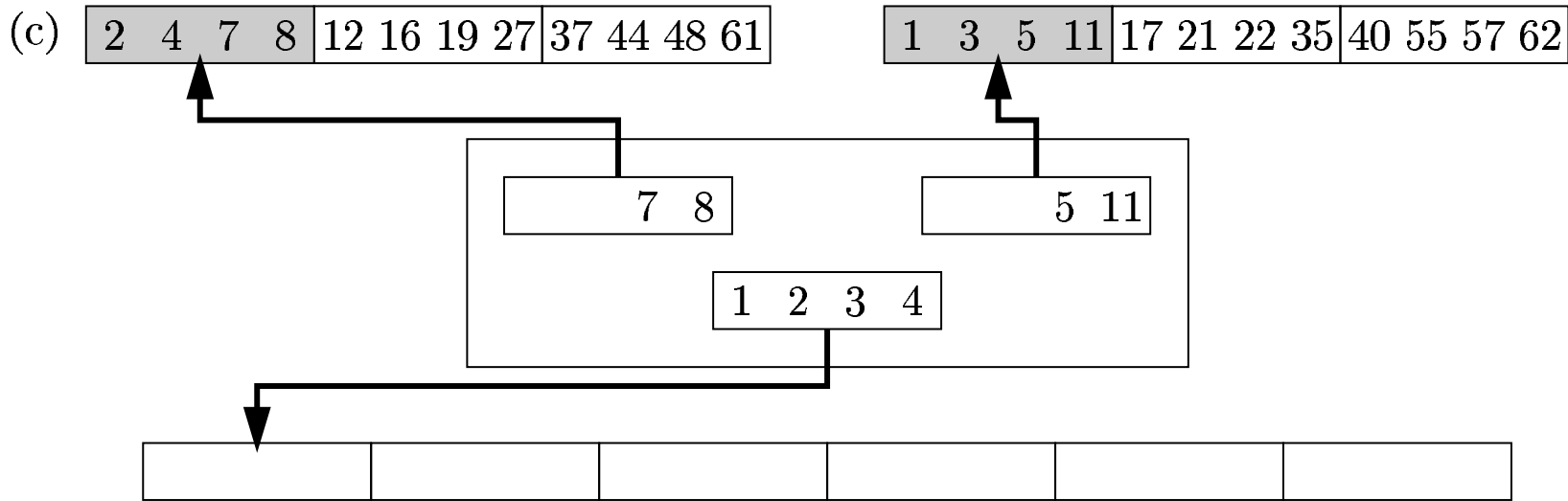
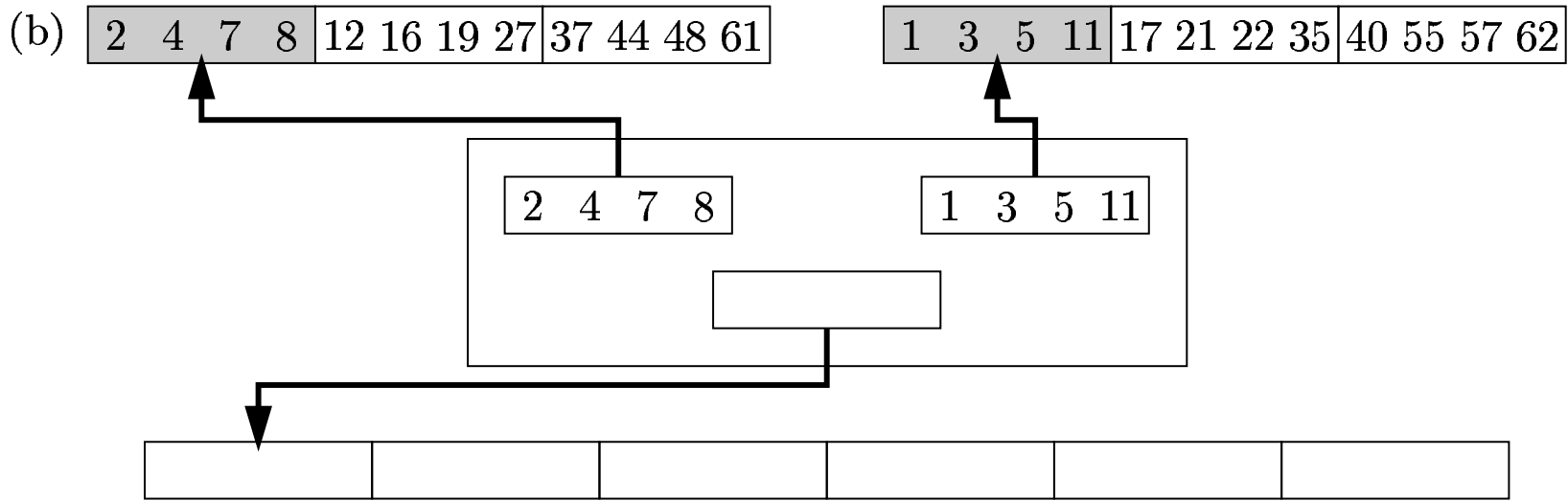
(b)

2	4	7	8	12	16	19	27	37	44	48	61
---	---	---	---	----	----	----	----	----	----	----	----

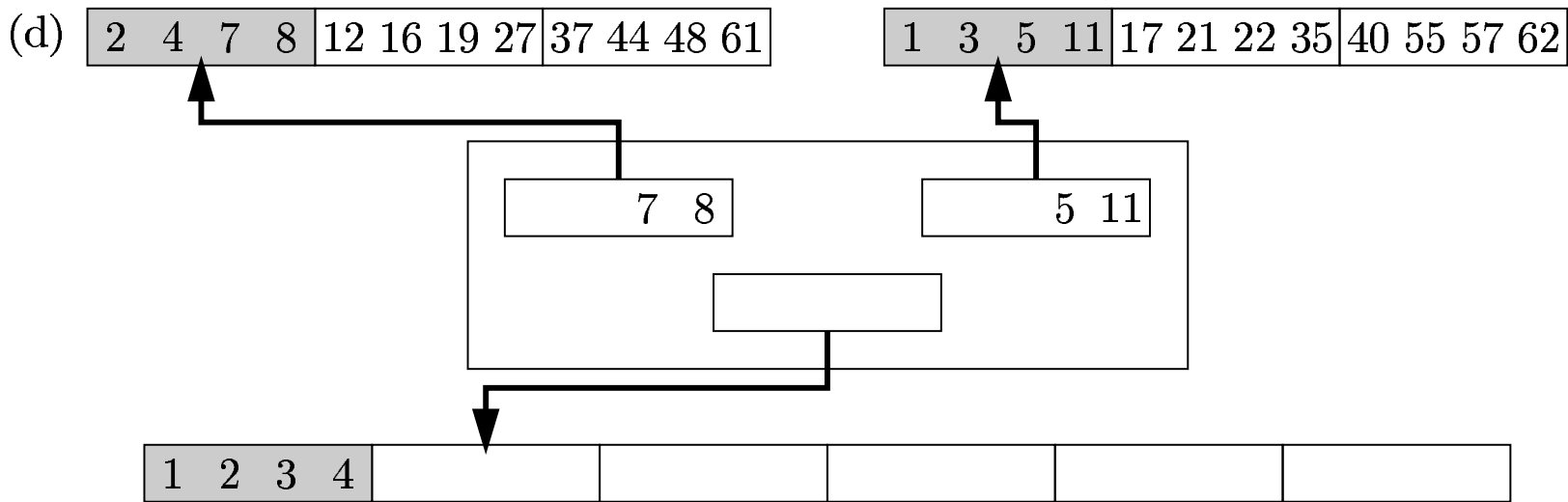
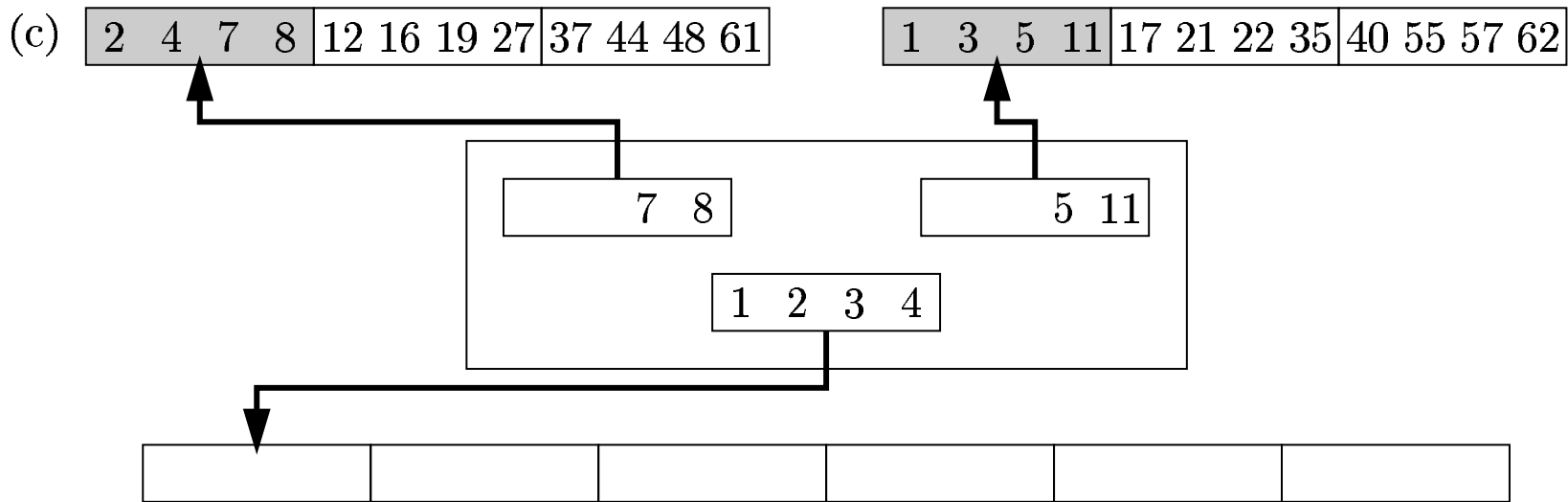
1	3	5	11	17	21	22	35	40	55	57	62
---	---	---	----	----	----	----	----	----	----	----	----



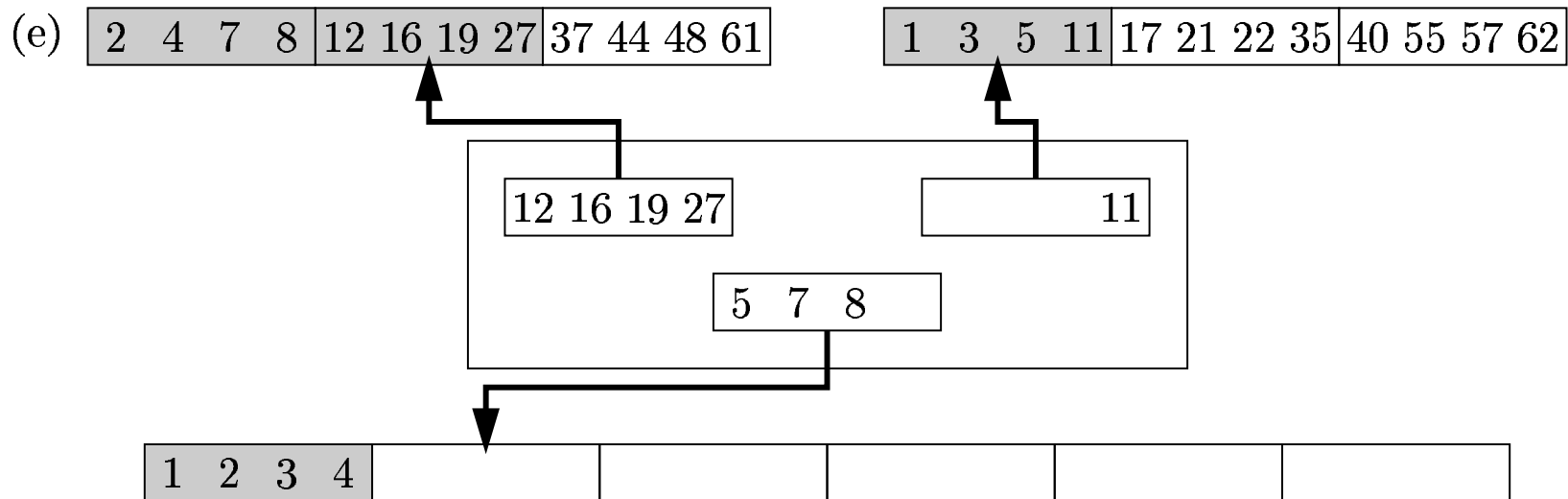
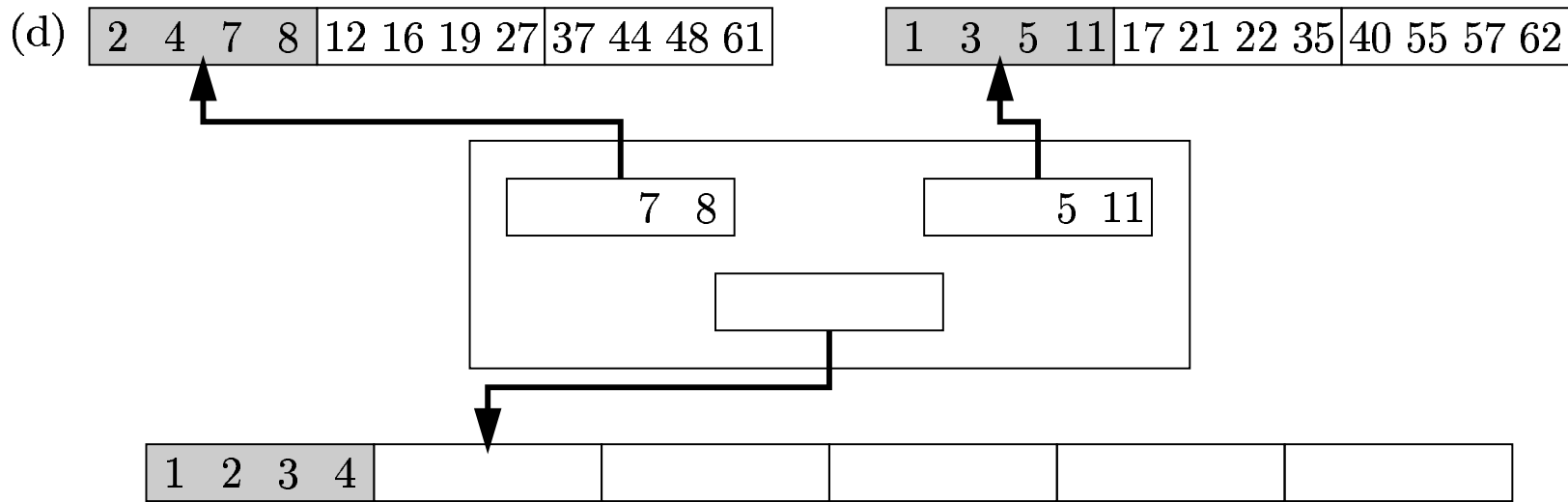
I/O-Efficient Mergesort



I/O-Efficient Mergesort



I/O-Efficient Mergesort



I/O Performance of Mergesort

Simple Mergesort: $\mathcal{O}(N + N/B \cdot \log_2(N/B))$ I/Os.

How do get rid of the N ? \Rightarrow Internal Memory Run Formation.

With initial runs of size $\Theta(M)$: $\mathcal{O}(N/B \cdot \log_2(N/B))$ I/Os.

Even faster? $\Rightarrow \Theta(M/B)$ -way merge.

Then: $\mathcal{O}(N/B \cdot \log_{M/B}(N/B))$ I/Os. Optimal.

Many alternatives, extensions:

- Distribution sort \approx I/O-eff. Quicksort
- Sorting with an I/O-Eff. Priority-Queue
- Parallel Disks
- Overlapping I/O and Computation
- Sorting is central part of most I/O Libraries (TPIE, STXXL, LEDA-SM)
- ...

Importance of Sorting - An Example

```
int[1..n] A,B,C;  
for i=1 to n do A[i]:=B[C[i]];
```

► Worst case: $\Omega(n)$ I/Os.

Better:

```
SCAN C:      (C[1]=17,1), (C[2]=5,2), ...  
SORT(1st):  (C[73]=1,73), (C[12]=2,12), ...  
par SCAN :  (B[1],73), (B[2],12), ...  
SORT(2nd):  (B[C[1]],1), (B[C[2]],2), ...
```

► Worst case: $\text{sort}(n)$ I/Os.

Graph Algorithms

- ▶ Many results, many open questions.
- ▶ Undirected case often easier than directed case.
- ▶ Dense graphs often easier than sparse graphs.
- ▶ Special graph classes often easier.
- ▶ General Methods: Time-Forward Processing, PRAM Simulation, Graphreduction,
- ▶ Efficient solutions: CC, MST, Listranking, . . .
- ▶ Still difficult: BFS, DFS, Shortest Paths, . . .

Parallel Simulation Paradigm

Let A be an N -processor PRAM algorithm s.t.

- ▶ A reduces a problem of size N to αN in constant time.
- ▶ Parallel running time of A is $\mathcal{O}(\log N)$.

For each PRAM statement, sort the $\leq N$ operands.

Simulate N operations via a linear pass through the data.

$$T(N) = \mathcal{O}(\text{sort}(N) + T(\alpha N)) = \mathcal{O}(\text{sort}(N)).$$

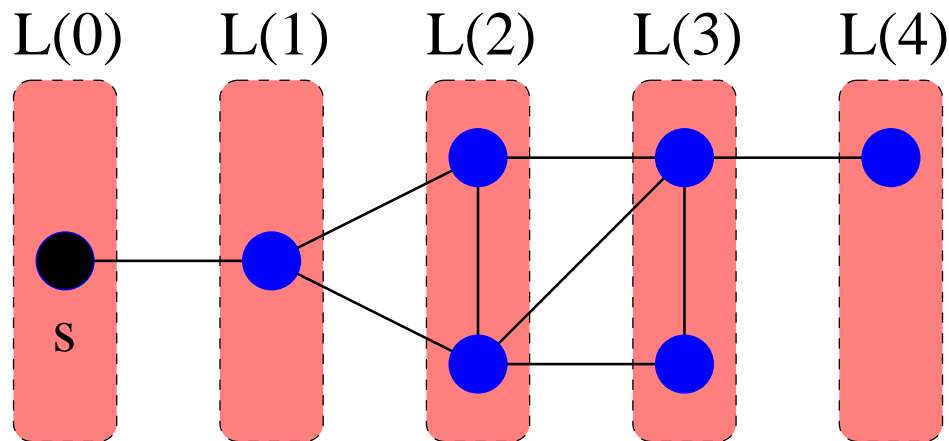
Yields e.g. optimal algorithm for list-ranking.

However, direct implementations usually have much better constants.

Breadth-First Search (BFS)

One of the most basic graph-traversal methods.

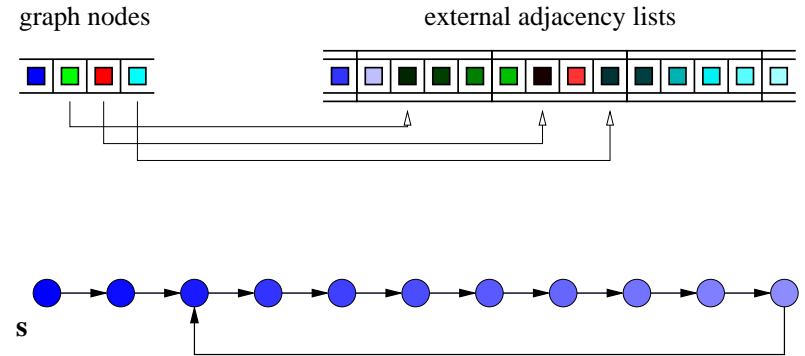
- ▶ **input:** $G = (V, E)$, undirected, $|V| = n$, $|E| = m$
- ▶ **one starting point:** s
- ▶ **compute:** BFS-levels $L(i)$, where $L(i) =$ nodes with dist. i (# edges) from s .



Standard implementation for internal memory: $\mathcal{O}(n + m)$ time.

Key Difficulties in External-Memory BFS

- (1) nodes visited in **unpredictable order**:
 unstructured access to adjacency lists
 seems to need at least one I/O per node.
- (2) **remembering settled nodes** requires extra data structures / algorithmic changes.

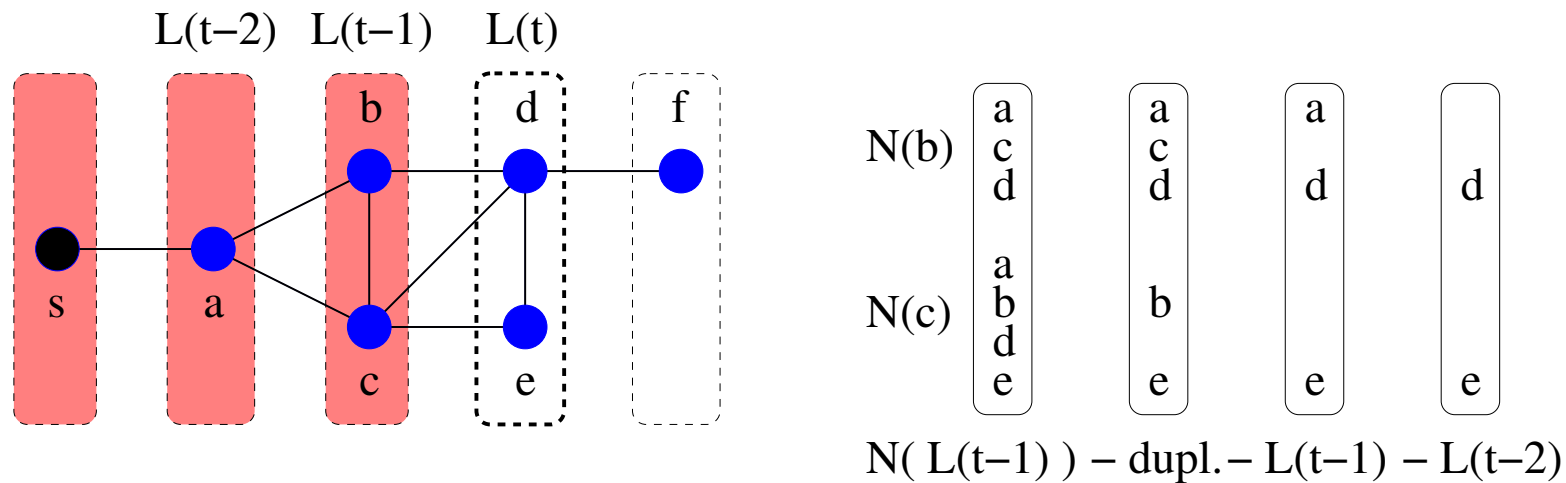


- ▶ Munagala/Ranade ('99) only address (2): $\mathcal{O}(n + \text{sort}(n + m))$ I/Os.
- ▶ we also address (1): $\mathcal{O}\left(\sqrt{nm/B} + \text{sort}(n + m)\right)$ I/Os.
- ▶ for sparse graphs ($m = \mathcal{O}(n)$) this is: $\mathcal{O}\left(n/\sqrt{B} + \text{sort}(n)\right)$ I/Os.
- ▶ **caveat**: both algs work only for undirected graphs.

Algorithm of Munagala and Ranade

Creating $L(t)$: all reached neighbors of nodes in $L(t - 1)$ belong to $L(t - 2)$ or $L(t - 1)$.

1. $N(L(t - 1)) =$ all neighbours of nodes in $L(t - 1)$ $\mathcal{O}(|L(t - 1)| + \frac{|N(L(t-1))|}{D \cdot B})$ I/Os.
2. eliminate duplicates in $N(L(t - 1))$ by sorting $\mathcal{O}(\text{sort}(|N(L(t - 1))|))$ I/Os.
3. eliminate nodes already in $L(t - 1)$ by sorting $\mathcal{O}(\text{sort}(|L(t - 1)|))$ I/Os.
4. eliminate nodes already in $L(t - 2)$ by sorting $\mathcal{O}(\text{sort}(|L(t - 2)|))$ I/Os.



$$\sum_i |N(L(i))| \leq 2 \cdot m \text{ and } \sum_i |L(i)| \leq n \Rightarrow \mathcal{O}(n + \text{sort}(n + m)) \text{ I/Os in total.}$$

Improved BFS Algorithm [MM02]

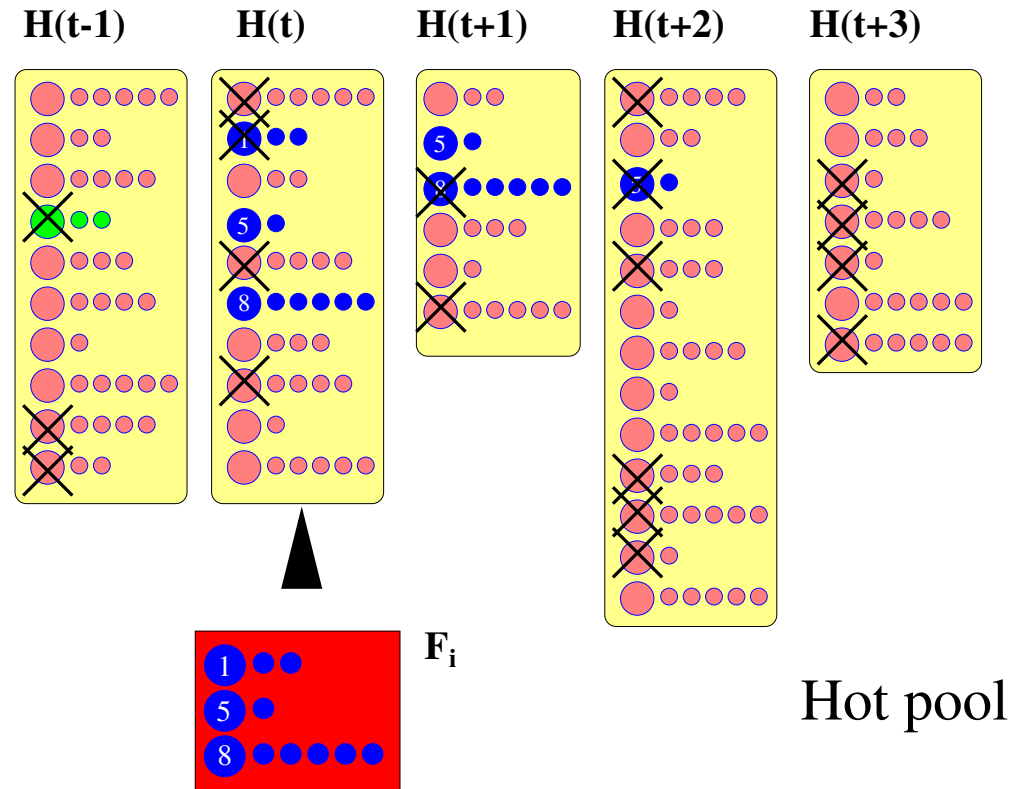
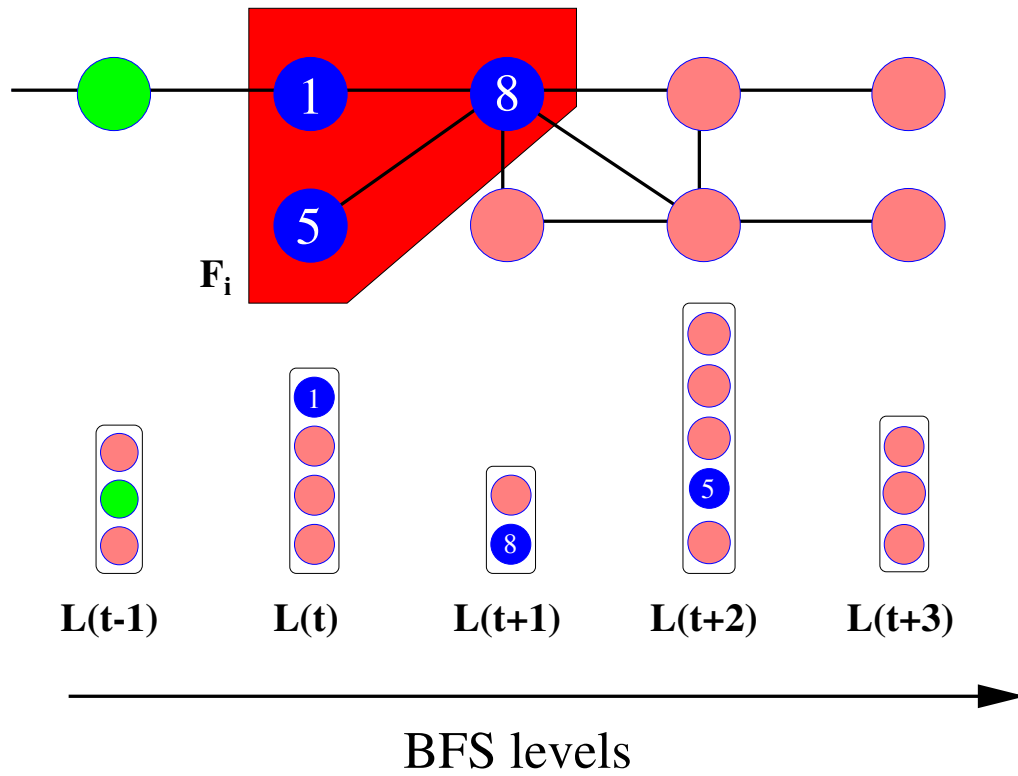
Preprocessing: $\mathcal{O}(\text{sort}(n + m))$ I/Os

- ▶ partition vertices into $\mathcal{O}(n/\mu)$ **subsets (clusters)** C_i s.t. any two nodes in same C_i have distance at most μ in G .
- ▶ store adjacency lists of nodes in the same C_i consecutively $\rightsquigarrow F_i$.

BFS Phase: Refined Algorithm of Mungala-Ranade ('99)

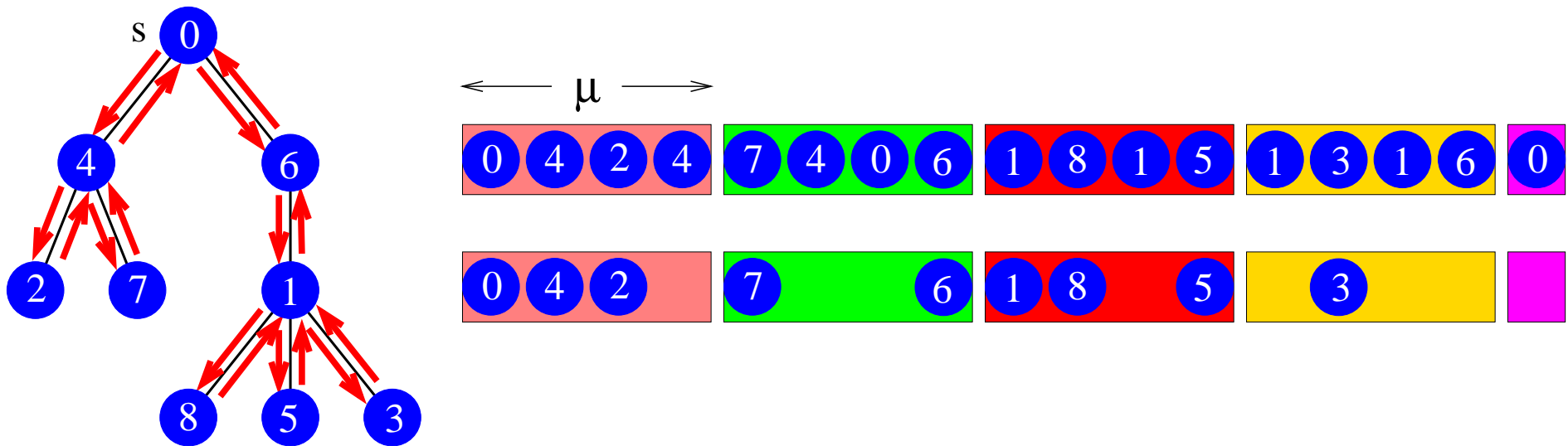
- ▶ extract adjacency lists by scanning sorted external data structure \mathcal{H} (**hot pool**).
- ▶ if first node in C_i is reached, add **all** adjacency lists of C_i to \mathcal{H} .
- ▶ each adjacency list stays in \mathcal{H} for at most μ **iterations**.
- ▶ $\mathcal{O}(n/\mu + \mu \cdot \text{scan}(n + m) + \text{sort}(n + m))$ I/Os.
- ▶ **Balancing:** $\mathcal{O}\left(\sqrt{nm/B} + \text{sort}(n + m)\right)$ I/Os.

BFS - An Illustration



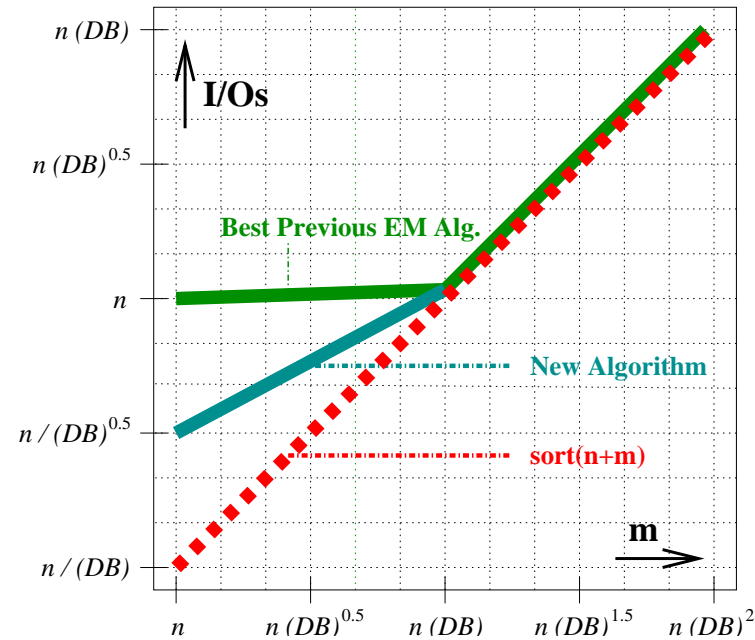
Simple Clustering for BFS

1. Build a spanning tree: $\mathcal{O}(\text{sort}(n + m))$ I/Os (randomized).
2. Obtain Euler-tour (length $2n$): $\mathcal{O}(\text{sort}(n))$ I/Os.
3. Chop Euler-tour into $2n/\mu$ pieces.
4. Eliminate duplicates: $\mathcal{O}(\text{sort}(n))$ I/Os.



$\Rightarrow \mathcal{O}(\text{sort}(n + m))$ I/Os for partitioning.

Summary BFS



number of I/Os as a function of m for fixed n .

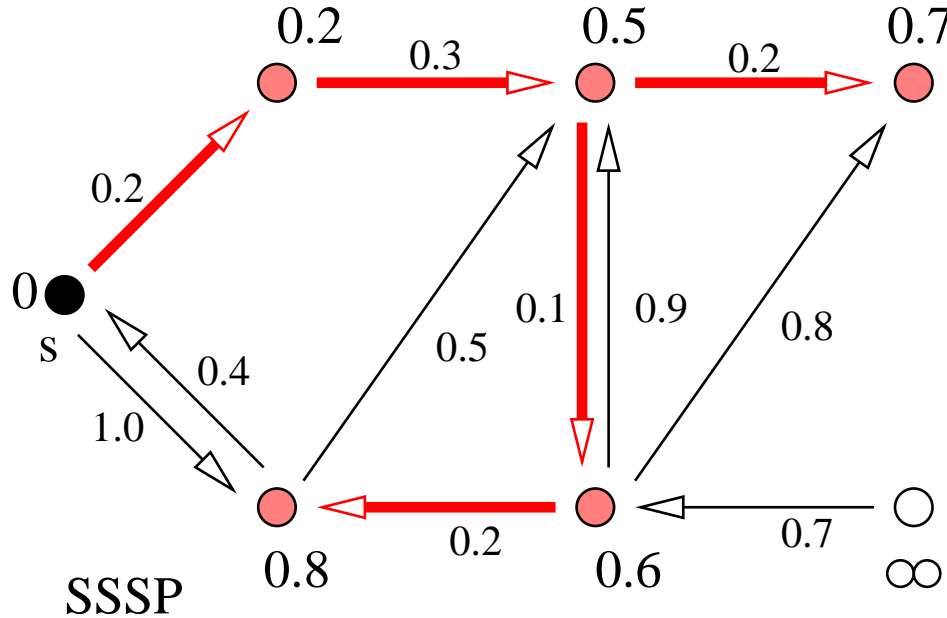
Current state:

- ▶ New BFS Alg much better on general sparse graphs.
- ▶ Still large gap to lower bound.
- ▶ Directed graphs ???

Single-Source Shortest-Paths (SSSP)

Classical optimization problem with numerous applications.

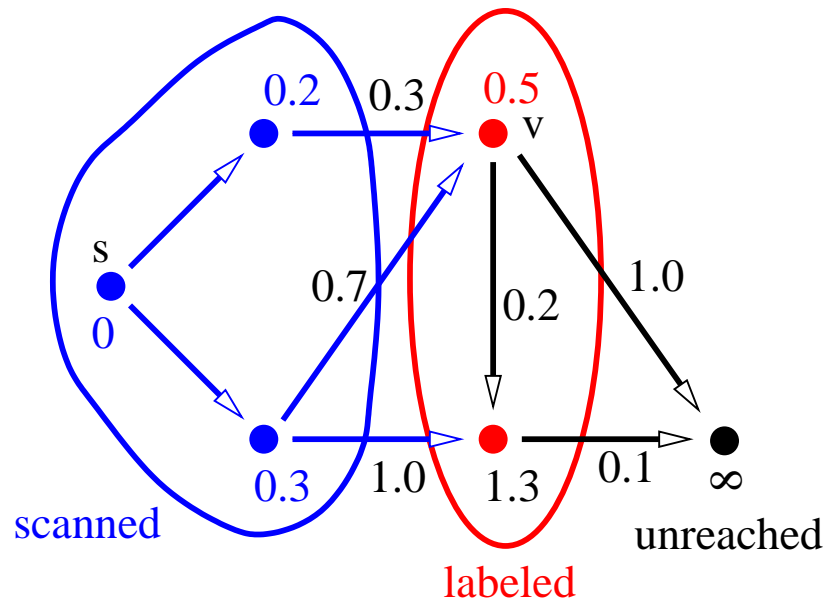
- ▶ **input:** $G = (V, E)$, undirected, $|V| = n$, $|E| = m$
- ▶ **one starting point:** s
- ▶ **nonnegative edge weights:** $0 \leq c_{\min} \leq c(e) \leq c_{\max}$
- ▶ **compute:** $\text{dist}(v) = \min\{c(p) ; p \text{ path from } s \text{ to } v\}$



Standard Approach: Dijkstra's Algorithm

- ▶ subdivides vertices into “settled”, “labeled” and “unreached”
- ▶ **labeled**: tentative distances $\text{tent}(v)$ in *priority queue* Q
- ▶ considers vertices one-by-one according to priorities:

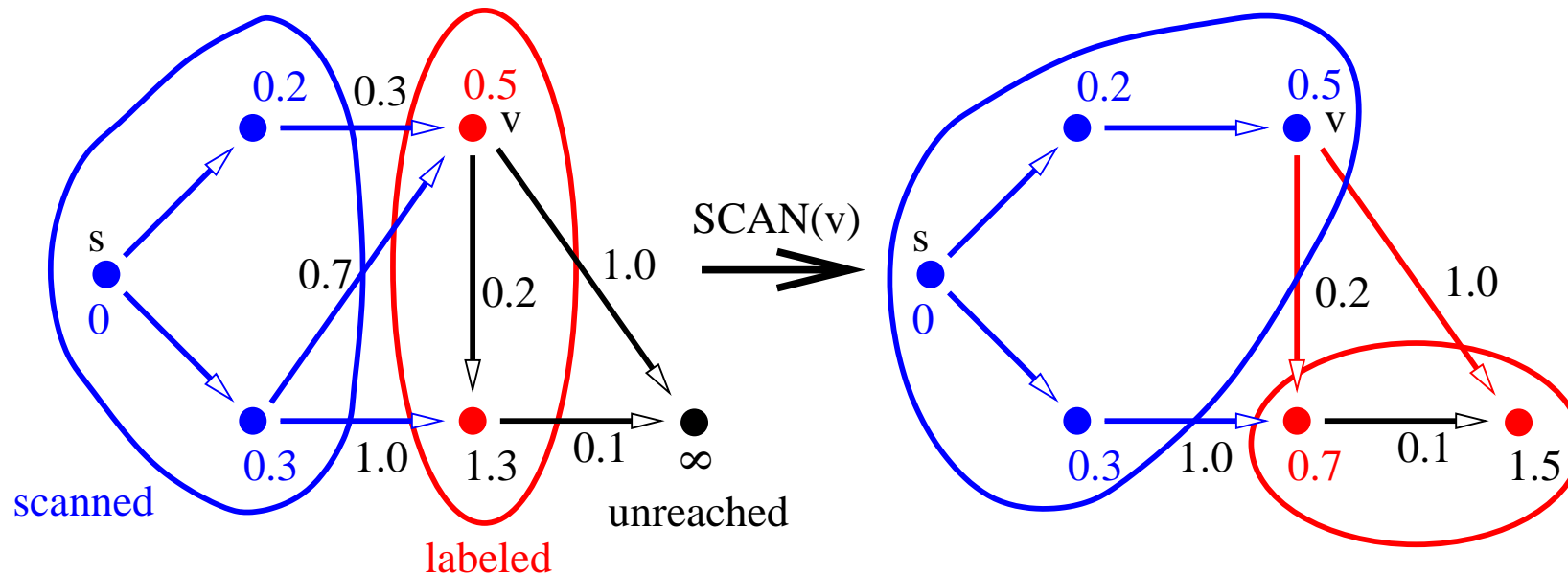
v removed from Q (v settled) $\Rightarrow \text{tent}(v) = \text{dist}(v)$



Standard Approach: Dijkstra's Algorithm

- ▶ subdivides vertices into “settled”, “labeled” and “unreached”
- ▶ **labeled**: tentative distances $\text{tent}(v)$ in *priority queue* Q
- ▶ considers vertices one-by-one according to priorities:

v removed from Q (v settled) $\Rightarrow \text{tent}(v) = \text{dist}(v)$



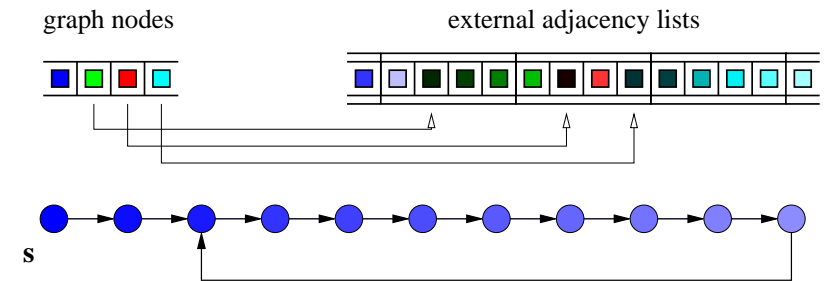
- ▶ $\text{SETTLE}(v)$ relaxes adjacent edges to non-settled nodes
- ▶ running time $\mathcal{O}(n \cdot \log n + m) \rightsquigarrow \mathcal{O}(\text{sort}(n \cdot \log n + m))$ I/Os ???

Key Difficulties in External-Memory SSSP

(1) Nodes visited in **unpredictable order**

(2) **Remembering settled nodes**

(3) Priority-Queues **without Decrease_Key**



▶ Kumar/Schwabe ('96) only address (2) & (3): $\mathcal{O}(n + (m/B) \log(n/B))$ I/Os.

▶ we also tackle (1): $\mathcal{O}\left(\sqrt{(nm/B) \log(c_{\max}/c_{\min})} + \text{sort}(n + m)\right)$ I/Os.

▶ for sparse graphs ($m = \mathcal{O}(n)$) this is: $\mathcal{O}\left(n \sqrt{\frac{\log(c_{\max}/c_{\min})}{B}} + \text{sort}(n)\right)$ I/Os.

▶ **caveats:**

- both algs work only for undirected graphs.
- performance of our alg depends on c_{\max}/c_{\min} .

BFS goes SSSP (1): Integers in $\{1, \dots, W\}$ [MM02]

BFS-like algo with $\leq nW$ levels (out of which $\leq n$ are nonempty).

- ▶ Oblivious preprocessing: max. cluster diameter $\mu \rightarrow \mu W$.
- ▶ PQ: Parallel scanning of $\Theta(W)$ buckets ($BW < M$).
- ▶ Performance: $\mathcal{O}\left(\sqrt{nmW/B} + W_{\text{sort}}(n + m)\right)$ I/Os.
- ▶ only efficient for $W \ll B$!!!

The Central Problem:

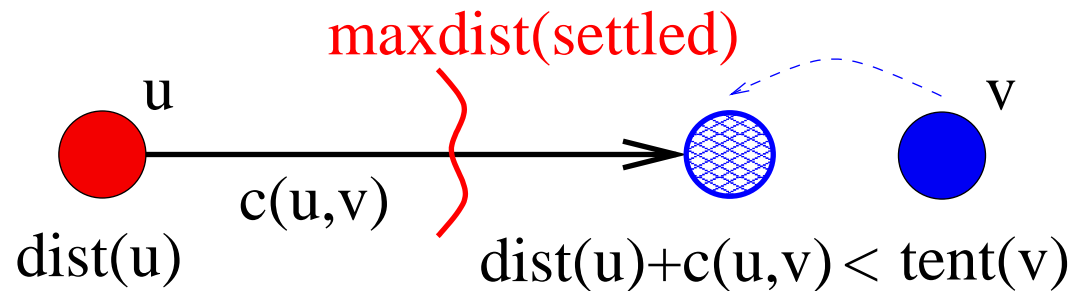
Edges may be scanned too often in the hot pool !

Solution:

Delayed edge relaxations / non-uniform hot pool scanning.

Delayed Edge Relaxations

- ▶ Relaxation of edge (u, v) can be delayed until nodes of distance $\text{dist}(u) + c(u, v)$ are to be settled:



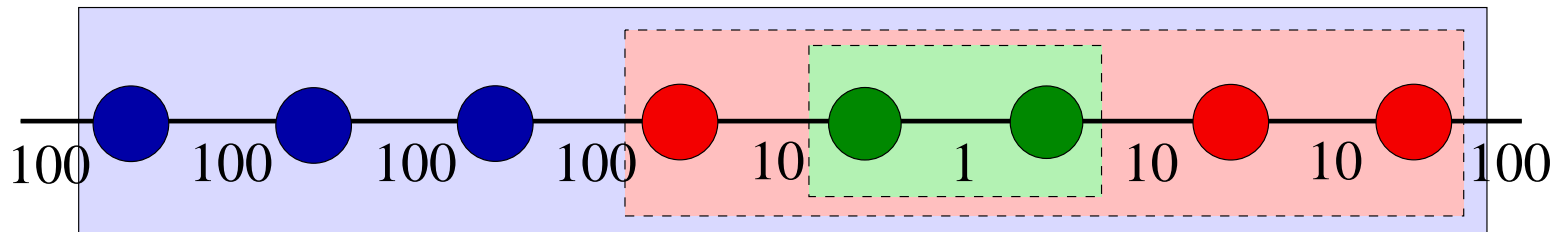
- ▶ idea has been used for many *internal memory* SSSP algos (in order to avoid implicit sorting).
- ▶ none of them alone yields improved I/O-bounds.
- ▶ delayed relaxation needs to be combined with hot pool scanning !

BFS goes SSSP (2): random integers $\{1, \dots, W\}$

- ▶ Oblivious Preprocessing: $\mathcal{O}(n/\mu)$ clusters, max. cluster diameter μW !
- ▶ One hot pool $\mathcal{H} \rightarrow r = \lceil \log W \rceil$ hot pools \mathcal{H}_i :
 - \mathcal{H}_i handles edges of weight $\{2^{i-1}, \dots, 2^i - 1\}$
 - \mathcal{H}_i is scanned whenever $\text{maxdist}(\text{settled})$ has increased by $\geq 2^{i-1}$
- ▶ On average $m/2^{r+1-i}$ edges in \mathcal{H}_i , each scanned $\leq 2^{r+1-i}\mu$ times.
- ▶ $\mathcal{O}(n/\mu + \mu \log W \text{scan}(n + m) + \text{sort}(n + m))$ I/Os on av.
- ▶ Balancing: $\mathcal{O}\left(\sqrt{nm \log W / B} + \text{sort}(n + m)\right)$ I/Os on av.
- ▶ What about non-random edge weights?

Moving Edges between Hot Pools

- ▶ **So far:** fixed assignment of edges to pools:
high weight \sim low scan frequency, low weight \sim high scan frequency
- ▶ **Now:** adaptive assignment:
start with low scan frequency, increase frequency as late as possible.



Result: $\mathcal{O}\left(\sqrt{nm \log W / B} + \text{sort}(n + m)\right)$ I/Os; exponential improvement!

(Previously: $\mathcal{O}\left(\sqrt{nmW / B} + W \text{sort}(n + m)\right)$ I/Os.)

EM All-Pairs BFS/SP [ArgMeyTom04]

We consider **sparse graphs**:

- ▶ Internal memory: All-Pairs = $n \cdot$ Single-Source (sequential).
- ▶ External memory: **Quasi-parallel** execution of all n sub-problems.
- ▶ n rounds with **joint adjacency list accesses**.
- ▶ n^2 **unstructured I/Os** $\rightsquigarrow \mathcal{O}(n)$ **sort & scan rounds**: $\mathcal{O}(\text{sort}(n \cdot m))$ I/Os.
- ▶ AP-BFS: I/O-optimal, APSP: Factor- \sqrt{B} improvement.
- ▶ **Trouble**: $\Omega(n^2)$ **working space**.

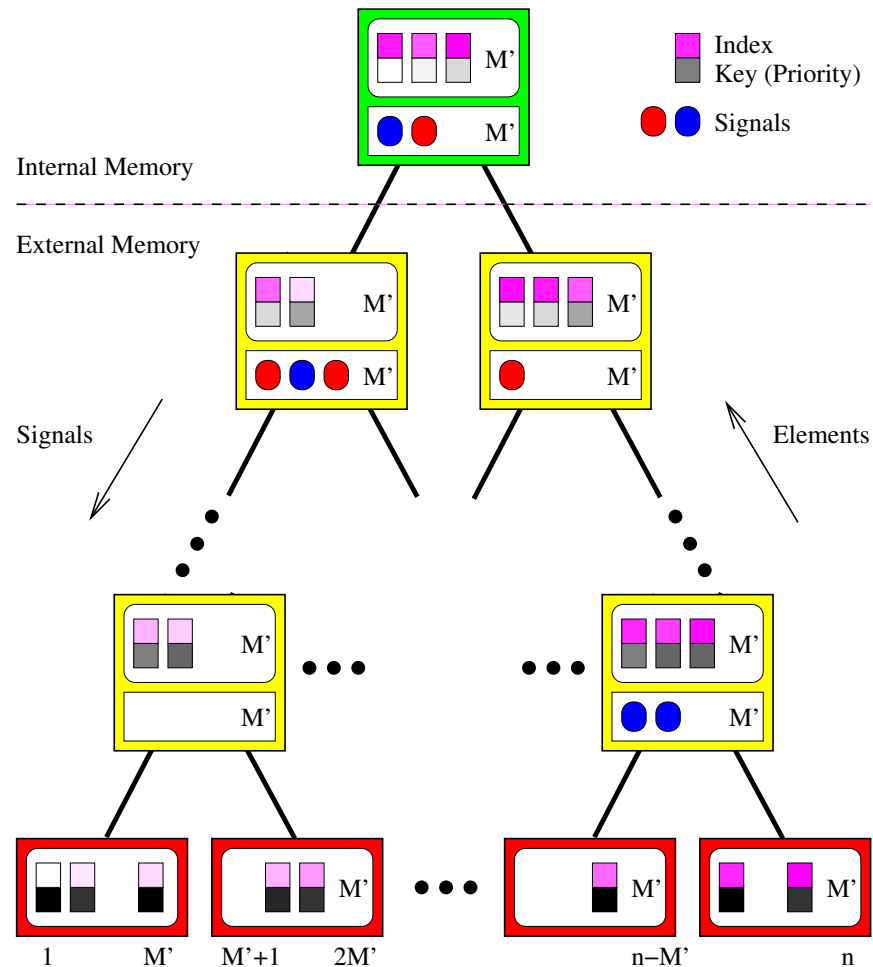
Idea for I/O-efficient APSP

- ▶ Quasi-parallel execution of all n SSSP problems.
- ▶ n rounds with **joint adjacency list accesses**.
- ▶ n^2 **unstructured I/Os** $\rightsquigarrow \mathcal{O}(n)$ **sorting & scanning steps**: $\mathcal{O}(\text{sort}(n \cdot m))$ I/Os.

s_i, v	swap →	sort →	scan →	sort →	$s_i, \text{adj}(v)$
0,5	5,0	0,7	7,adj(0)	0,adj(5)	
1,3	3,1	1,6	6,adj(1)	1,adj(3)	
2,7	7,2	2,8	8,adj(2)	2,adj(7)	
3,4	4,3	3,1	1,adj(3)	3,adj(4)	
4,7	7,4	3,5	5,adj(3)	4,adj(7)	
5,3	3,5	4,3	3,adj(4)	5,adj(3)	
6,1	1,6	5,0	0,adj(5)	6,adj(1)	
7,0	0,7	5,9	9,adj(5)	7,adj(0)	
8,2	2,8	7,2	2,adj(7)	8,adj(2)	
9,5	5,9	7,4	4,adj(7)	9,adj(5)	

Trouble: Switching between Subproblems

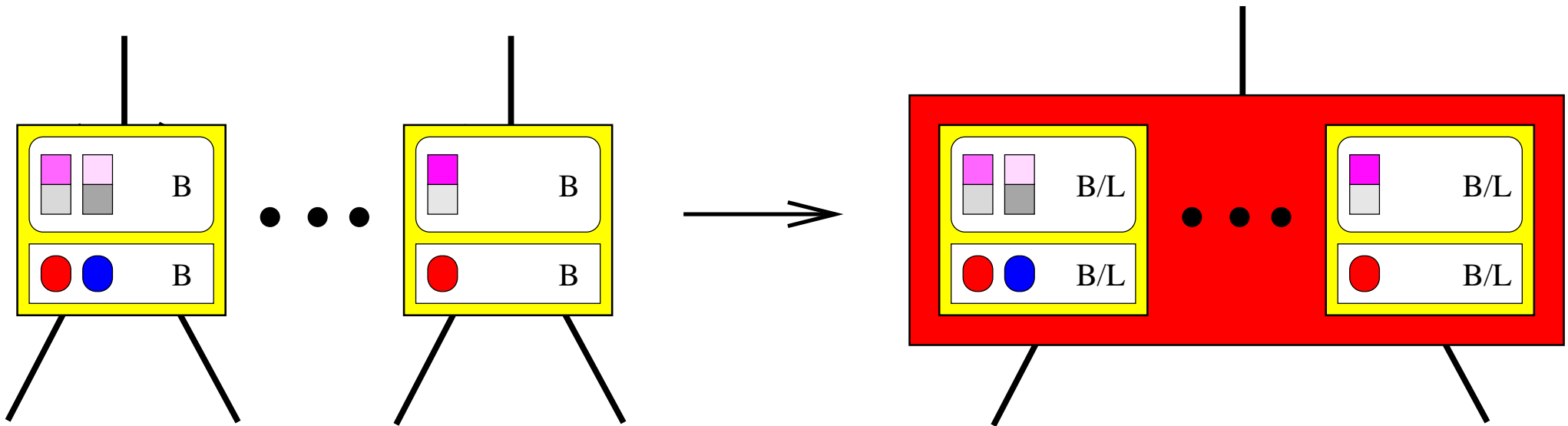
- ▶ Kumar/Schwabe SSSP uses I/O-Tournament-Tree: $\Theta(1)$ blocks in IM.
- ▶ n^2 Problem switches: $\Omega(n^2)$ I/Os !!!



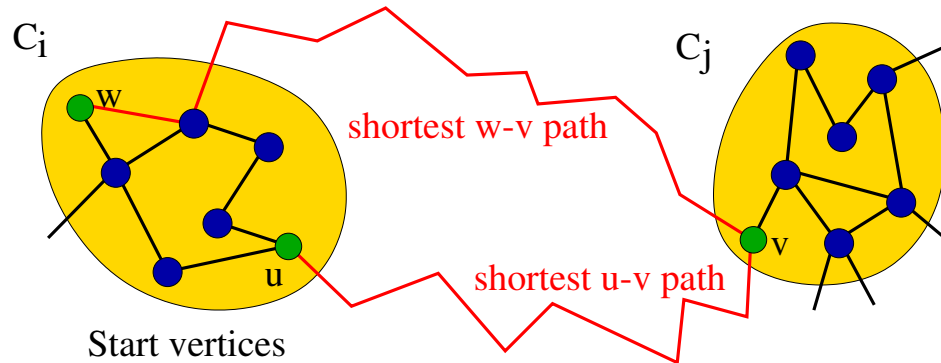
Solution: Multi-Tournament-Trees

- ▶ Bundle $L > 1$ I/O-Tournament-Trees into one data structure.
- ▶ Operations on single I/O-TTs will cause L times more I/O.
- ▶ Switching costs reduce to $\mathcal{O}(n^2/L)$ I/Os.
- ▶ Sparse graphs: $L = \sqrt{B/\log n} \Rightarrow$

APSP using $\mathcal{O}(n^2 \cdot \sqrt{\frac{\log n}{B}} + \text{sort}(n^2))$ I/Os



Improving the Space for AP-BFS



$$u, w \in C_i : | \text{BFS_level}(v, \text{source } u) - \text{BFS_level}(v, \text{source } w) | \leq \mu$$

Idea: Modify the [MehMey02] BFS-Alg as follows:

- ▶ Restrict parallelism to sources within the same cluster.
- ▶ Keep adjacency lists in \mathcal{H} at most twice as long.
- ⇒ Can amortize unstructured I/O over all sources of the start cluster.
- ▶ I/O-optimal and $\mathcal{O}(n \cdot \sqrt{B})$ working space.
- ▶ I/O-Space Trade-off: Linear space with $\log B$ -I/O loss is possible, too.
- ▶ Similar results: Chowdhury/Ramachandran, SODA 05.

EM-BFS Implementation (Prelim. Results)

- ▶ Implemented RAM-BFS, [MunRan99], and [MehMey02]
- ▶ Build on top of STXXL (sorting, vectors, iterator, parallel disks).
- ▶ Algorithms use extensive pipelining.
- ▶ Still in fine-tuning phase.

First results:

- ▶ **RAM-BFS clearly worst** on most external instances.
- ▶ [MunRan99] better than [MehMey02] on well-behaved instances (typ. 0.5 hours vs. 2.5 hours).
- ▶ [MehMey02] much better than [MunRan99] on difficult instances (typ. 1.2 days vs. 23 days).
- ▶ [MunRan99] more I/O bound than [MehMey02].

Conclusions

done:

- ▶ I/O Model
- ▶ Scanning, Sorting
- ▶ Elementary Data Structures
- ▶ Parallel Simulation
- ▶ Some Graph Algorithms

next time:

- ▶ Cache-Oblivious Algs. & Data Structures