Massive Data Algorithmics

Lecture 12: Cache-Oblivious Model

Typical Computer

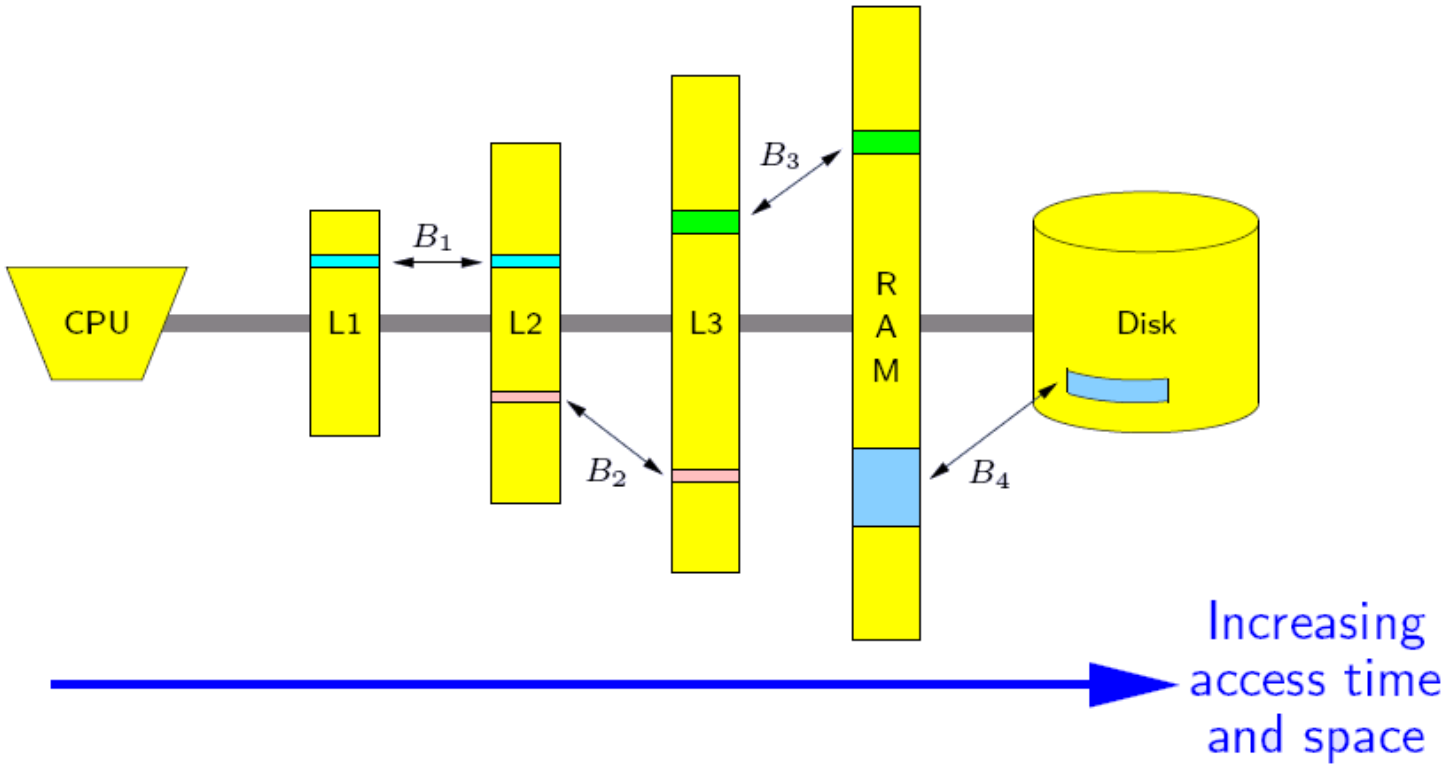
www.dell.dk

Processor speed	2.4 – 3.2 GHz
L3 cache size	0.5 – 2 MB
Memory	1/4 – 4 GB
Hard Disk	36 GB – 146 GB
	7.200 – 15.000 RPM
CD/DVD	8 – 48x

www.intel.com

L2 cache size	256 – 512 KB
L2 cache line size	128 Bytes
L1 cache line size	64 Bytes
L1 cache size	16 KB

Hierarchical Memory Basics

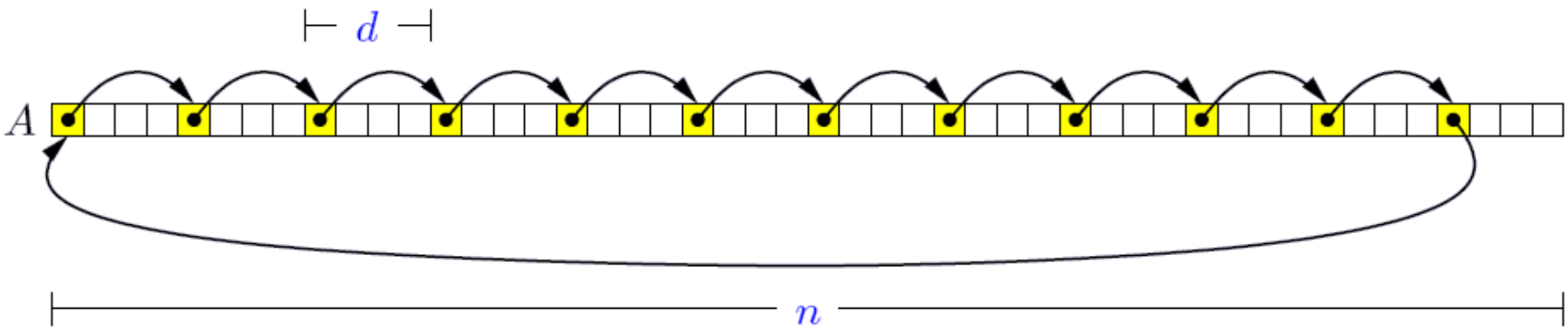


- Data moved between adjacent memory level in blocks

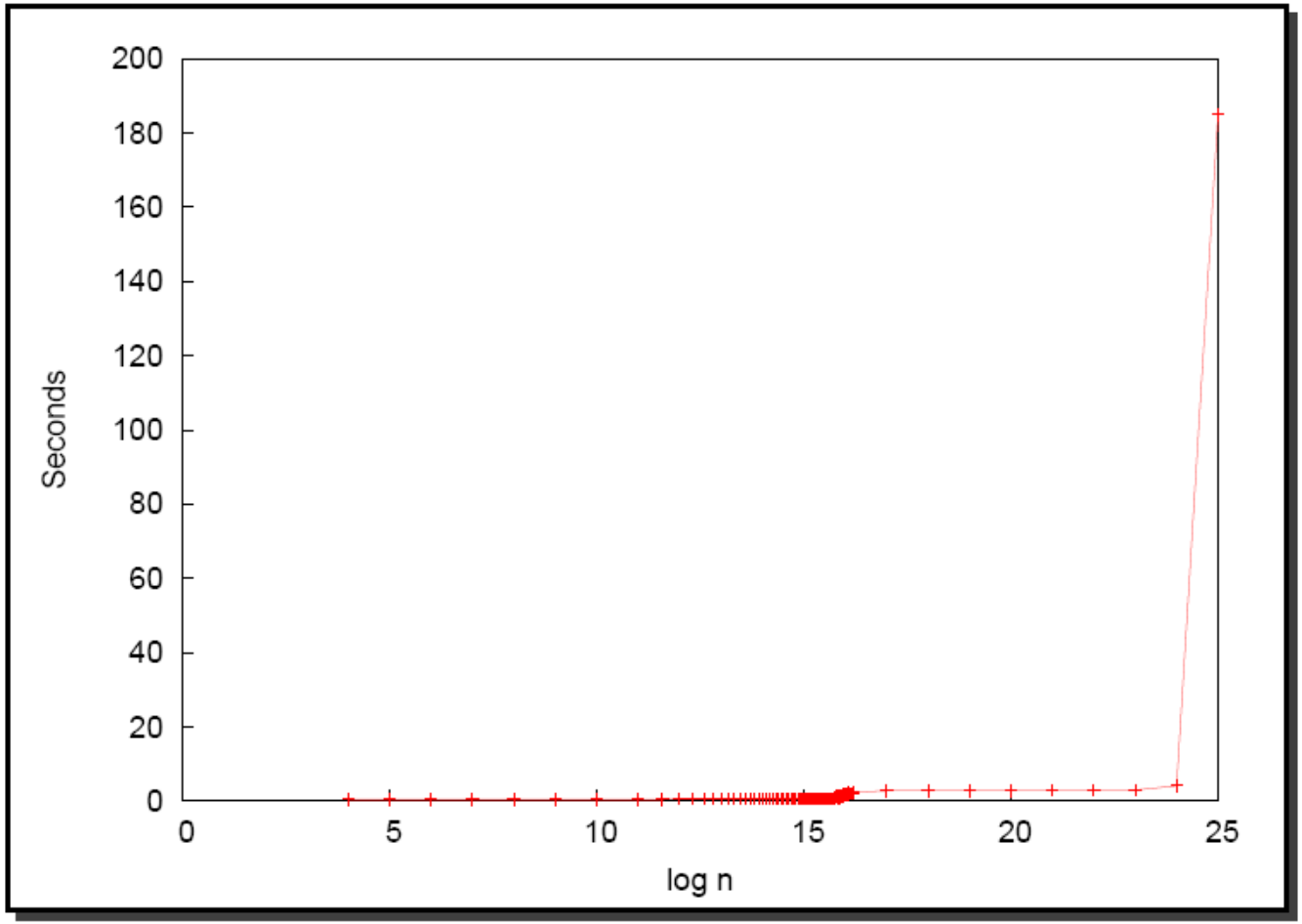
A Trivial Program

```
for (i=0; i+d<n; i+=d) A[i]=i+d;
A[i]=0;

for (i=0, j=0; j<8*1024*1024; j++) i=A[i];
```

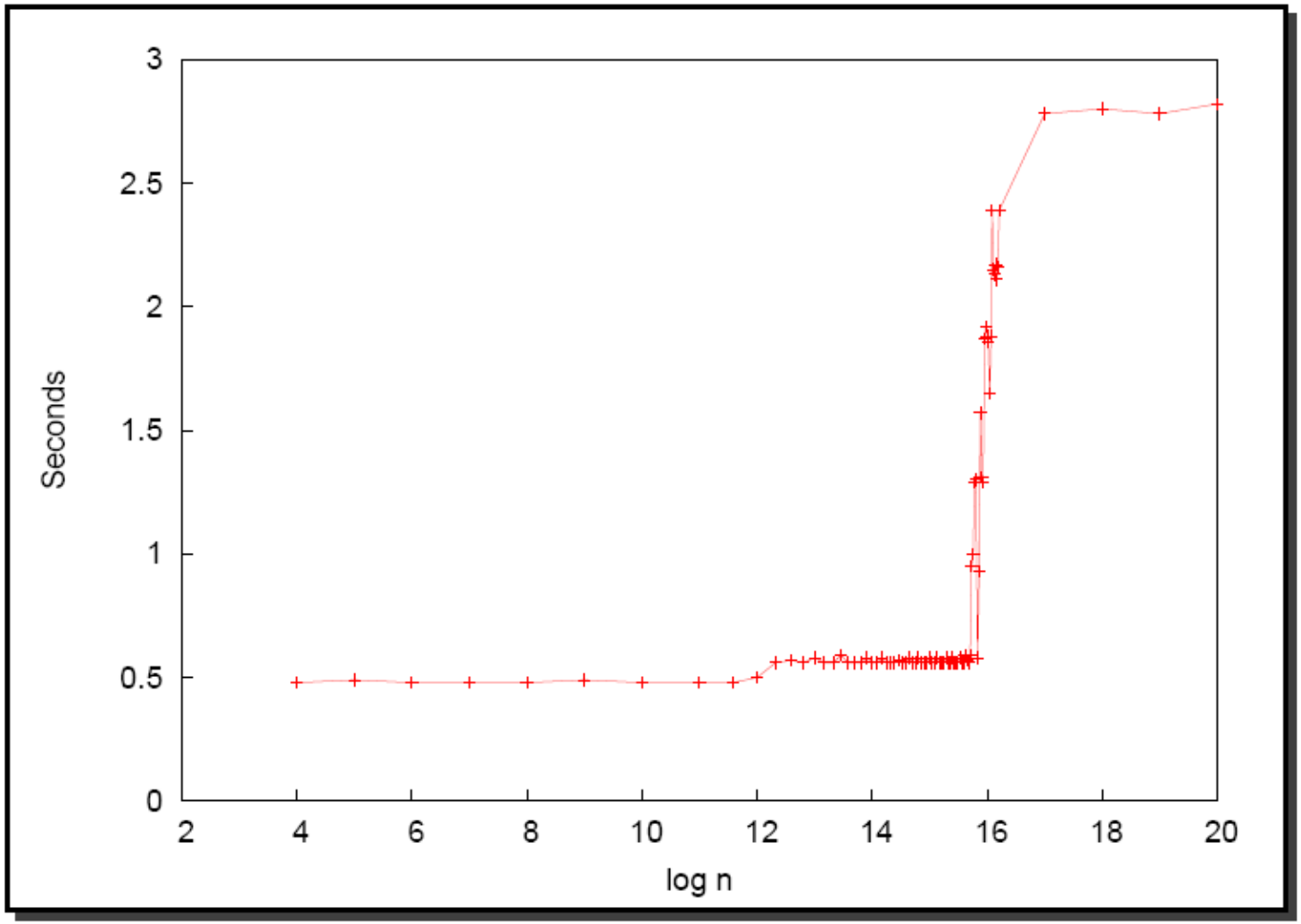


A Trivial Program: $d = 1$



RAM : $n \approx 2^{25} \equiv 128\,MB$

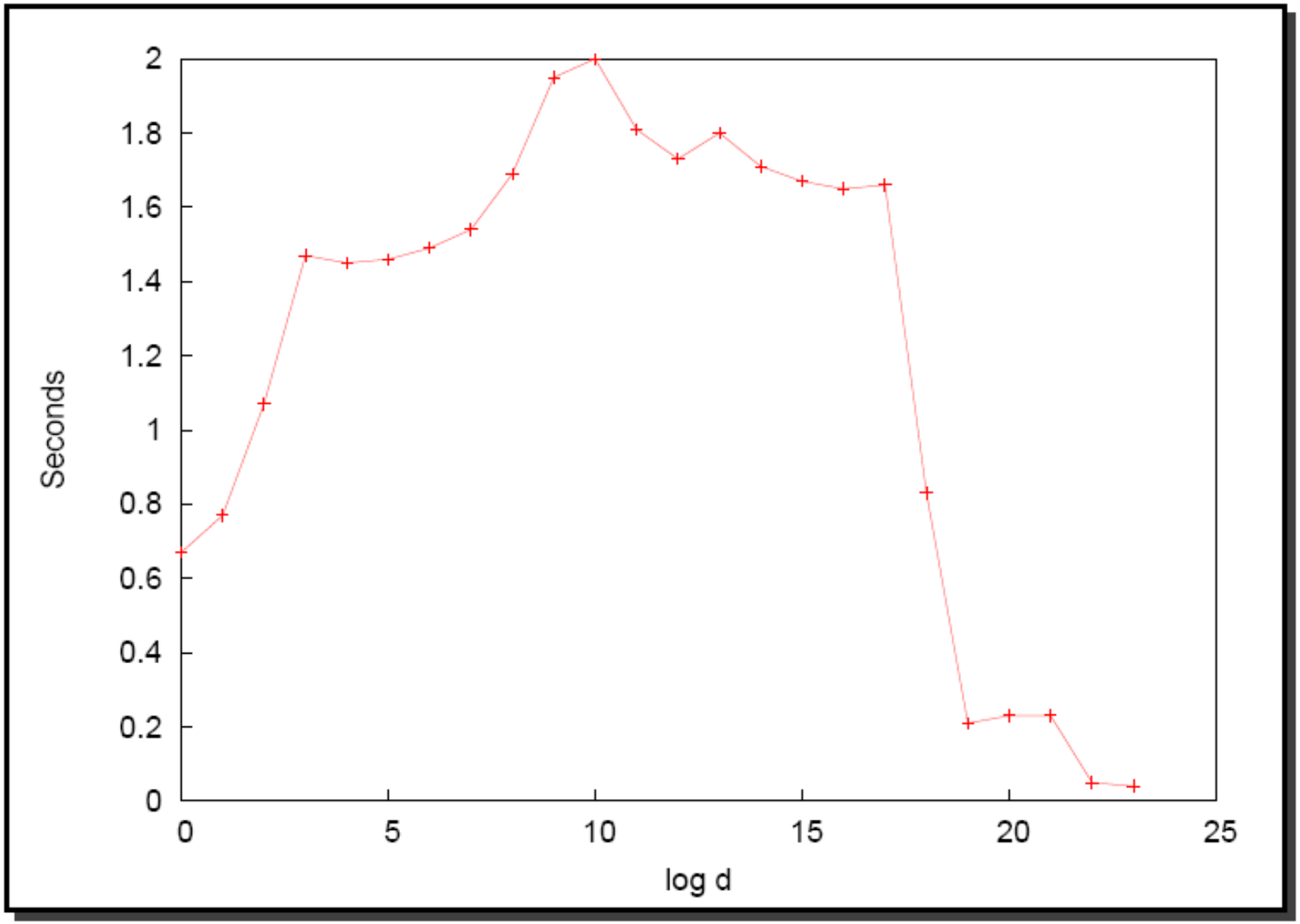
A Trivial Program: $d = 1$



L1 : $n \approx 2^{12} \equiv 16\,KB$

L2 : $n \approx 2^{16} \equiv 256\,KB$

A Trivial Program: $n = 2^{24}$



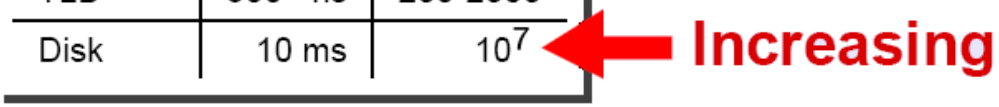
A Trivial Program: Hardware Spec

- Experiments were performed on a DELL 8000, Pentium III, 850 MHz, 128MB RAM, running Linux 2.4.2, and using gcc version 2.96 with optimization -O3
- L1 instruction and data caches
  - 4-way set associative, 32-byte line size
  - 16 KB instruction cache and 16KB write-back data cache
- L2 level cache
  - 8-way set associative, 32-byte line size
  - 256KB

Algorithmic Problems

- Memory hierarchy has become a fact of life
- Accessing non-local storage may take a very long time
- Good locality is important for achieving high performance

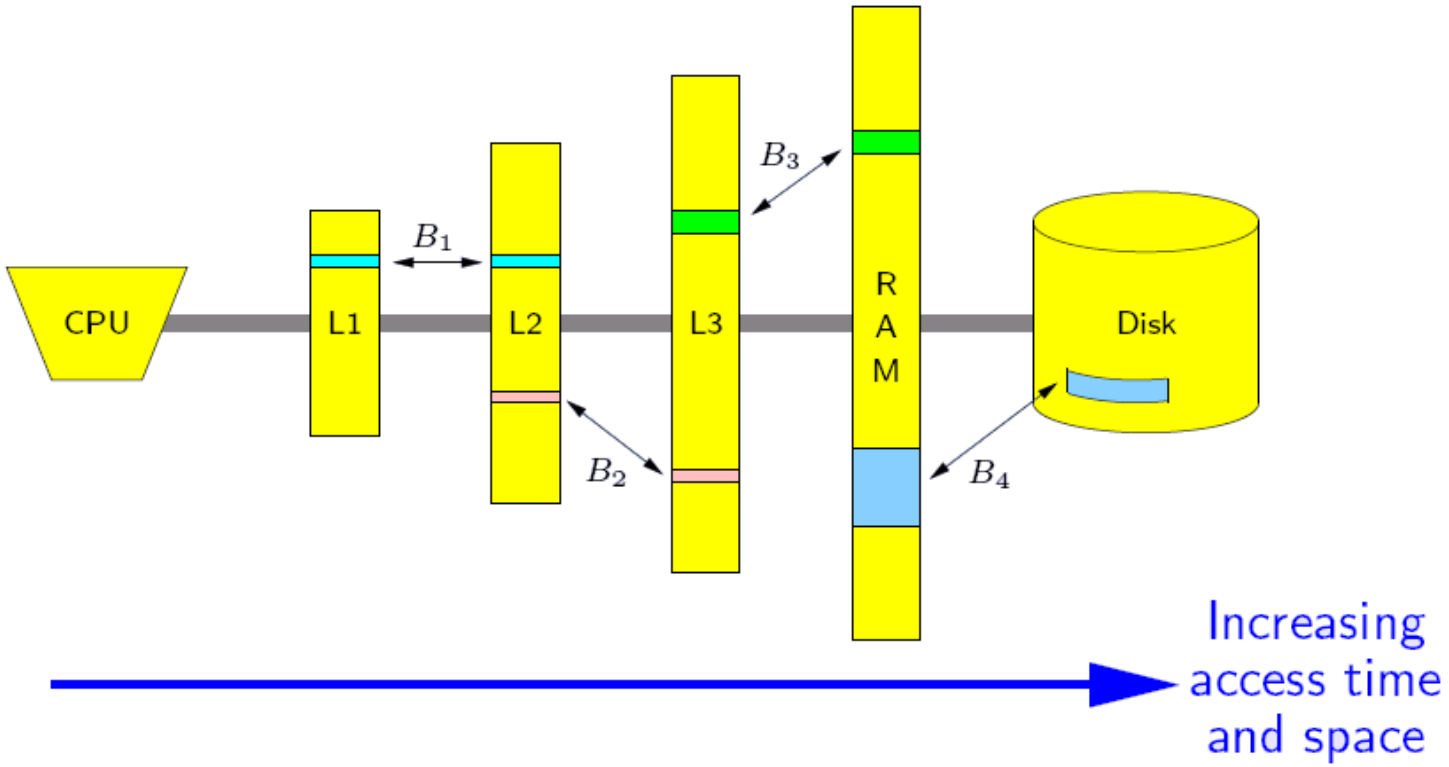
	Latency	Relative to CPU
Register	0.5 ns	1
L1 cache	0.5 ns	1-2
L2 cache	3 ns	2-7
DRAM	150 ns	80-200
TLB	500+ ns	200-2000
Disk	10 ms	10^7



Algorithmic Problems

- Modern hardware is not uniform many different parameters
  - Number of memory levels
  - Cache sizes
  - Cache line/disk block sizes
  - Cache associativity
  - Cache replacement strategy
  - CPU/BUS/memory speed
- Programs should ideally run for many different parameters
  - by knowing many of the parameters at runtime
  - by knowing few essential parameters
  - ignoring the memory hierarchies

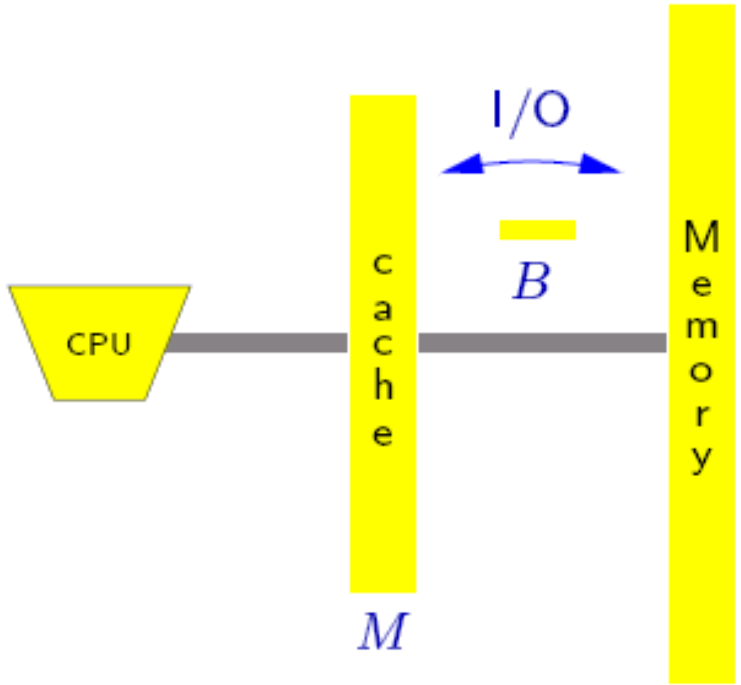
Hierarchical Memory Model—many parameters



- Limited success since model to complicated

I/O Model—two parameters

- Measure number of block transfers between two memory levels
- Very successful (simplicity)

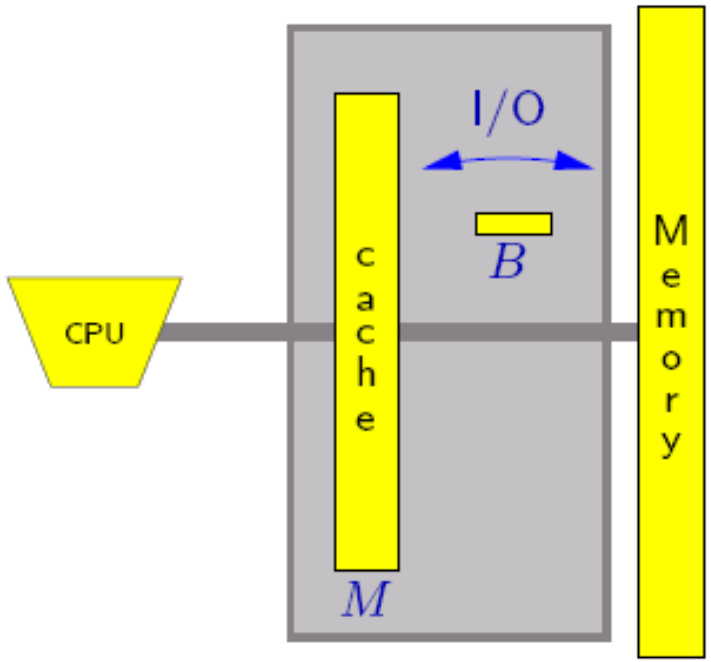


Limitations

- Parameters B and M must be known
- Does not handle multiple memory levels
- Does not handle dynamic M

Ideal Cache Model—no parameters!?

- Program with only one memory
- Analyze in the I/O model for
- Optimal off-line cache replacement strategy arbitrary B and M



Advantages

- Optimal on arbitrary level $\Rightarrow$ optimal on all levels
- Portability, B and M not hard-wired into algorithm
- Dynamic changing parameters

Justification of the Ideal Cache Model

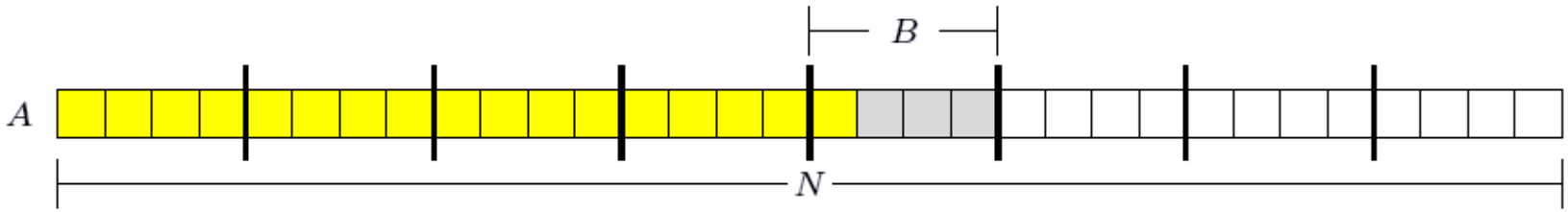
- Optimal replacement: LRU + $2\times$cache size $\Rightarrow$ at most $2\times$ cache misses
- Fully associativity cache: Simulation using hashing
- Tall-cache assumption: height is bigger than width $\Rightarrow M/B \geq B$

Scan

- Write data in a contiguous segment of memory

$sum = 0$
for $i = 1$ to N do $sum = sum + A[i]$

$$O\left(\frac{N}{B}\right) \text{ I/Os}$$



Median

- Conceptually partition the array into $N/5$ quintuplets of ve adjacent elements each.
- Compute the median of each quintuplet using O(1) comparisons.
- Recursively compute the median of these medians
- Partition the elements of the array into two groups, according to whether they are at most or strictly greater than this median.
- Count the number of elements in each group, and recurse into the group that contains the element of the desired rank.

Median

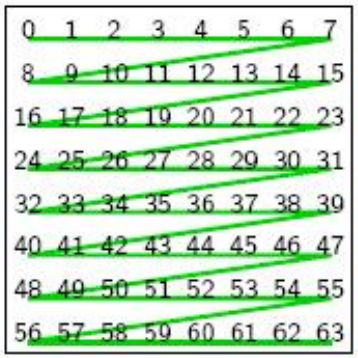
- Each step can be done with at most 3 parallel scans.
- $T(N) = T(N/5) + T(7N/10) + O(N/B)$
- $T(O(1)) = O(1) \Rightarrow T(N) = \Omega(N^c)$ where $(1/5)^c + (7/10)^c = 1$ $(c = 0.839)$
- $T(N) = \Omega(N^c)$ is larger than N/B when N is larger than B and smaller than BN^c
- But $T(O(B)) = O(1) \Rightarrow (N/B)^c$ leaves in the recursion tree.
- $O((N/B)^c) = o(N/B)$ memory transfer
- Cost per level decrease geometrically
- Total cost: $O(N/B)$

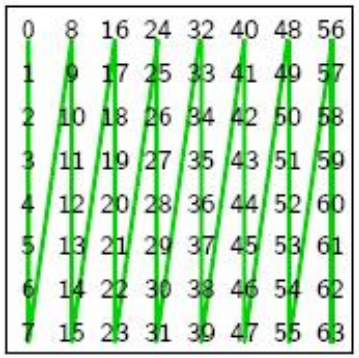
Matrix Multiplication

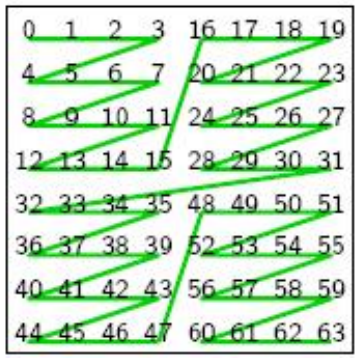
- Problem

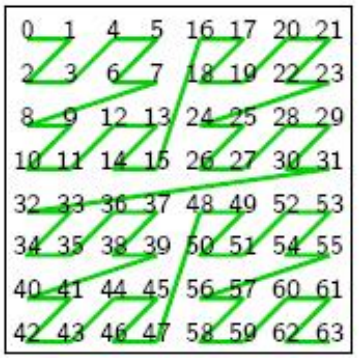
$$Z = X.Y \qquad z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

- Lay out









Matrix Multiplication

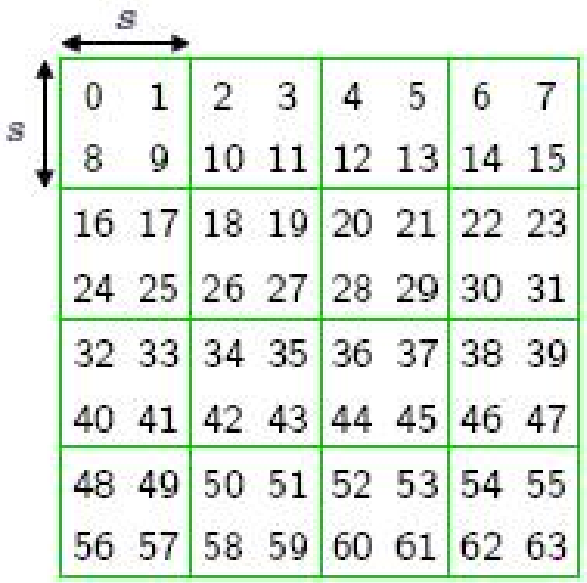
Algorithm 1: Nested Loops

- Row major
- Reading a column of Y, N I/Os
- Total $O(N^3)$ I/Os
- if Y is columns major $\Rightarrow$ $O(N^3/B)$ I/Os

- for $i = 1$ to N
- for $j = 1$ to N
- $z_{ij} = 0$
- for $k = 1$ to N
- $z_{ij} = z_{ij} + x_{ik}.ykj$

Algorithm 2: cache aware

- Partition into $s \times s$ blocks
- $s = O(\sqrt{M})$
- Apply algorithm 1 to $N/s \times N/s$ matrices where elements are $s \times s$ matrices
- Row major and $M = O(B^2)$
- $O((N/s)^3.s^2/B) = O(N^3/(B\sqrt{M}))$ I/Os



Matrix Multiplication

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{pmatrix}$$

- 8 recursive $\frac{N}{2} \times \frac{N}{2}$ multiplications + 4 $\frac{N}{2} \times \frac{N}{2}$ matrix sums
- # I/Os if row major and $M = \Omega(B^2)$

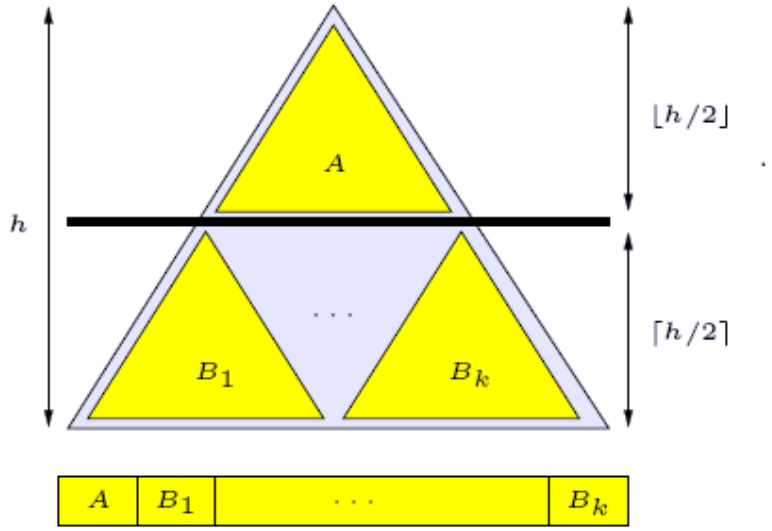
$$T(N) \leq \begin{cases} O(\frac{N^2}{B}) & \text{if } N \leq \varepsilon\sqrt{M} \\ 8 \cdot T\left(\frac{N}{2}\right) + O\left(\frac{N^2}{B}\right) & \text{otherwise} \end{cases}$$

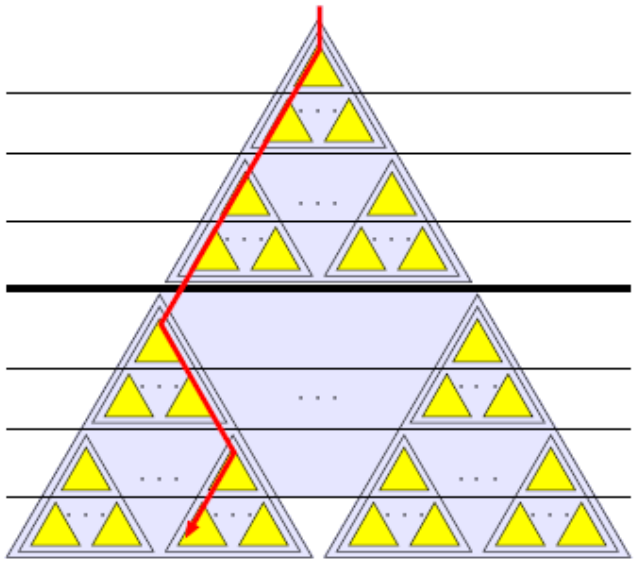
$$T(N) \leq O\left(\frac{N^3}{B\sqrt{M}}\right)$$

Static Search Tree

- Sorted array
- $T(N) = T(N/2) + 1$
- $T(B) = O(1)$
- $T(N) = \log N - \log B >> \log_B N$

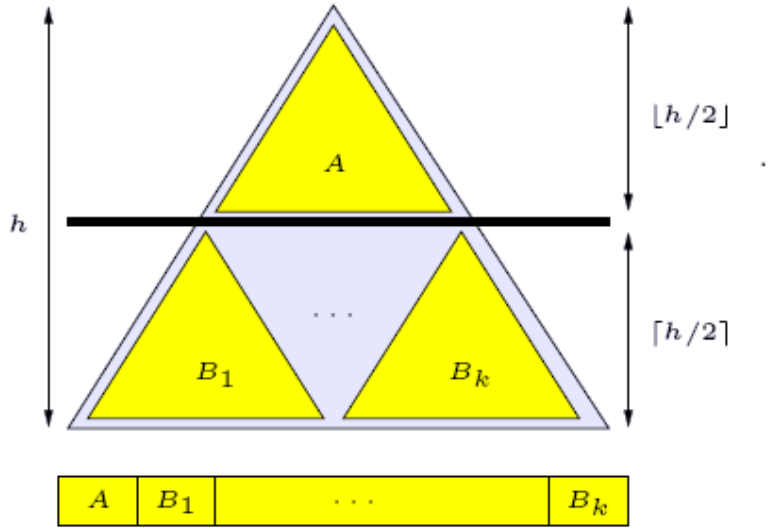
Static Search Tree

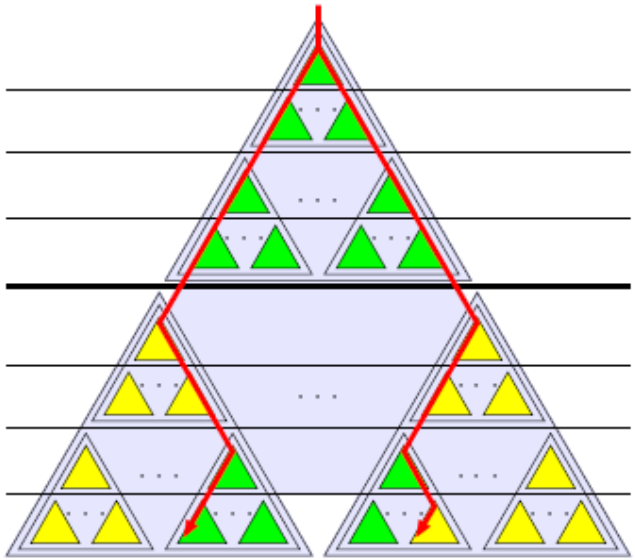




Searches use $O(\log_B N)$ I/Os

Static Search Tree





Searches use $O(\log_B N)$ I/Os

Range reportings use

$O\left(\log_B N + \frac{K}{B}\right)$ I/Os

Ordered File

- Maintaining a sequence of elements in order in memory, with constant size gaps, subject to N insertions and deletions of elements in the middle of the order
- Two extremes of trivial (inefficient) solutions
  - Avoid gaps: $O(N/B)$ memory transfers
  - Allocate 2^N memory, and the new element is stored in midway between the two given elements: O(1) memory transfers

Ordered File

- Fix N: whenever N grows or shrinks by a constant factor (2 for instance), rebuild the entire the data structure
- Conceptually divide the array of size N into subranges of size $O(\log N)$
- Conceptually construct a complete binary tree over subranges: height $h = \log N - \log\log N$
- Density of a node: the number of elements below that node divided by the total capacity of that node
- Density constraint to each node: for nodes at depth d density must be between $\frac{1}{2} - \frac{1}{4}d/h (\in [1/4, 1/2])$ and $\frac{3}{4} + \frac{1}{4}d/h (\in [3/4, 1])$

Ordered File: Updates

- Insertion:
  - If there is space in the relevant leaf subrange, we can accommodate the new element by possibly moving $O(\log N)$ moves
  - Otherwise, we walk up the tree by scanning elements until we find an ancestor within threshold.
  - We rebalance this ancestor by redistributing all of its element uniformly throughout the constituent leafs $\Rightarrow$ every descendant will be within threshold as density constraint increase walking down the tree.
- Deletion: In a similar way

Ordered File: Analysis

- The difference in density threshold of two adjacent levels is $O(\frac{1}{4}h) = O(1/\log N)$
- If the node has capacity K, $\Theta(K/\log N)$ elements should be inserted or deleted in order to fall outside the threshold again.
- Amortized cost of inserting and deleting below a particular node is $\Theta(\log N)$
- Each element falls below $h = \Theta(\log N)$ nodes $\Rightarrow$ total amortized cost: $\Theta(\log^2 N)$

$\Rightarrow O(\log^2 N)$ time and $O((\log^2 N)/B)$ memory transfers

B-trees

- A combination of two structures
  - An ordered file
  - A static search tree with size N
- We also maintain a fix one-to-one correspondence bidirectional pointers between cell in ordered files and leafs in the tree
- Each node of the tree store the maximum (non-blank) key of its two children

B-trees: search

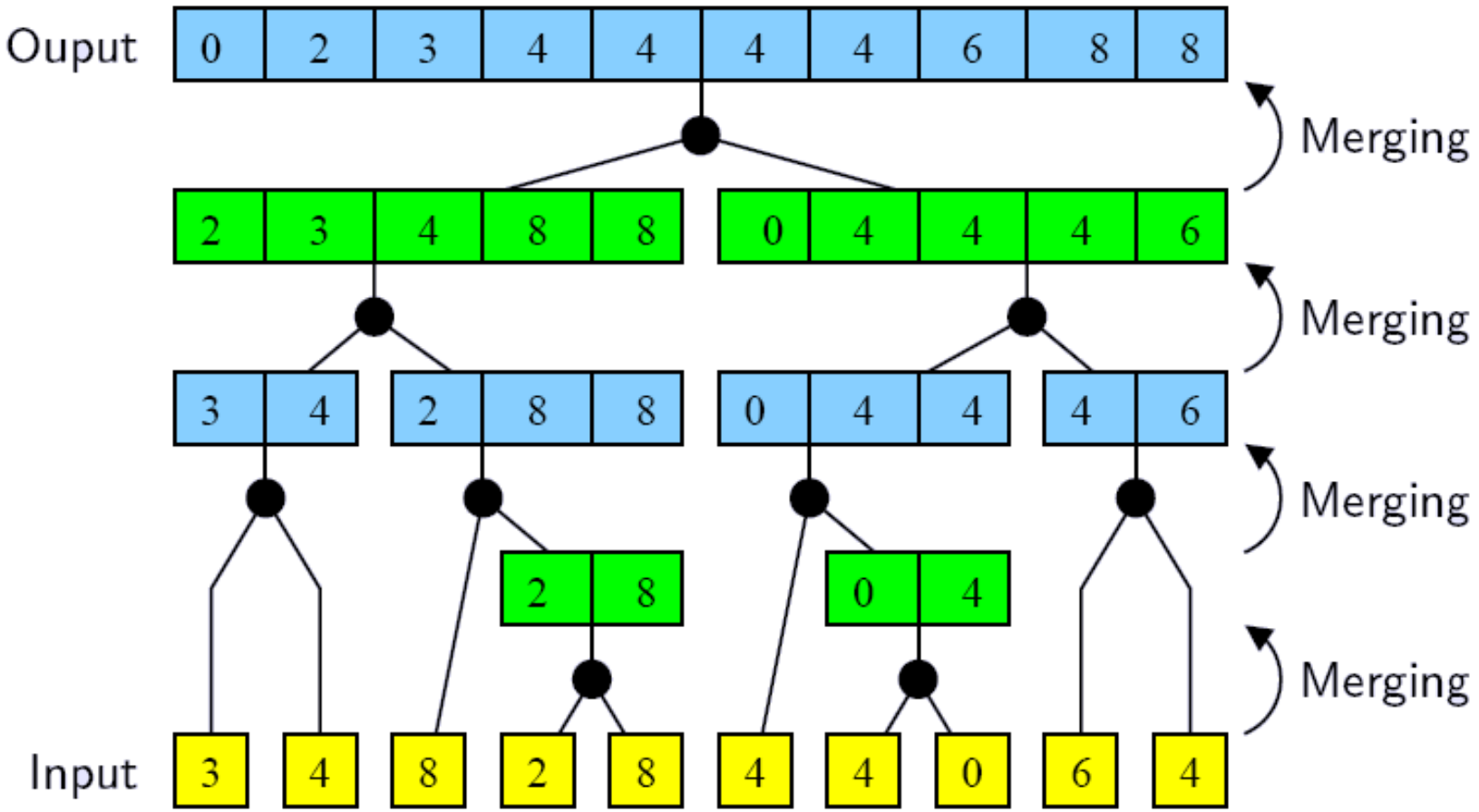
- Based on the maximum key stored in the left child we can decide go to left or right
- Since the tree is stored in Van Emde Boas layout, search needs $O(\log_B N)$ memory transfers

B-trees: Update

- Search in the tree for the location of given element
- Insert in the ordered file
- Let K be the number of movements in ordered file (K amortized is $O(\log^2 N)$)
- Leaves of tree corresponding to the affected K cells in ordered file must be updated using bidirectional pointers: $O(K/B)$ memory transfers
- The key changes are propagated up the tree (using post-order traversal) to all ancestors to updates maximum keys stored in internal nodes: $O(K/B + \log_B N)$ memory transfers

$\Rightarrow$ Updates: $O(\log_B N + (\log^2 N)/B)$ memory transfers

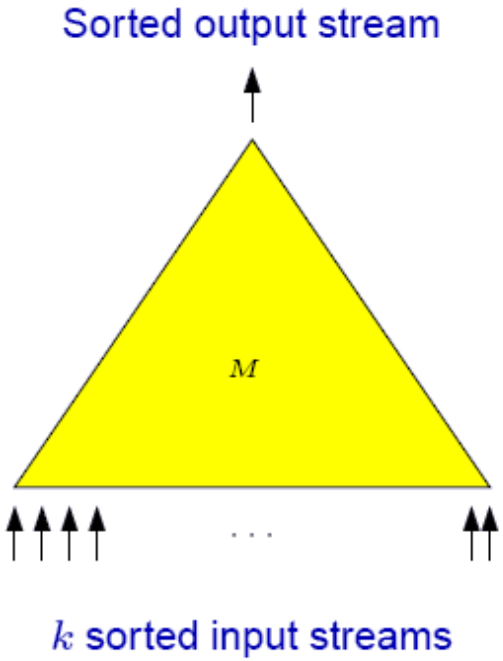
Merge Sort


Ouput
0 2 3 4 4 4 4 6 8 8
Merging
2 3 4 8 8
0 4 4 4 6
Merging
3 4
2 8 8
0 4 4
4 6
Merging
2 8
0 4
Merging
Input
3 4 8 2 8 4 4 0 6 4

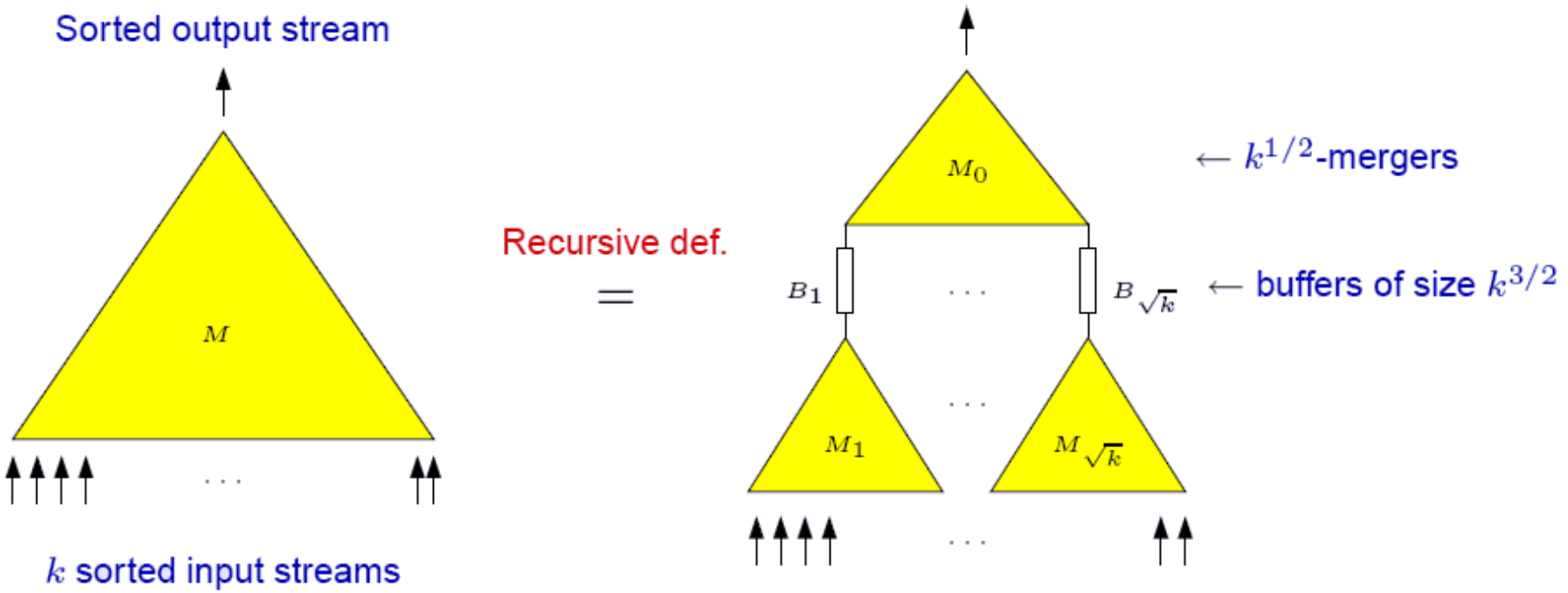
Merge Sort

Degree	I/O
2	$O\left(\frac{N}{B}\log_2\frac{N}{M}\right)$
d $(d \leq \frac{M}{B}-1)$	$O\left(\frac{N}{B}\log_d\frac{N}{M}\right)$
$\Theta\left(\frac{M}{B}\right)$	$O\left(\frac{N}{B}\log_{M/B}\frac{N}{M}\right) = O(\mathrm{Sort}_{M,B}(N))$

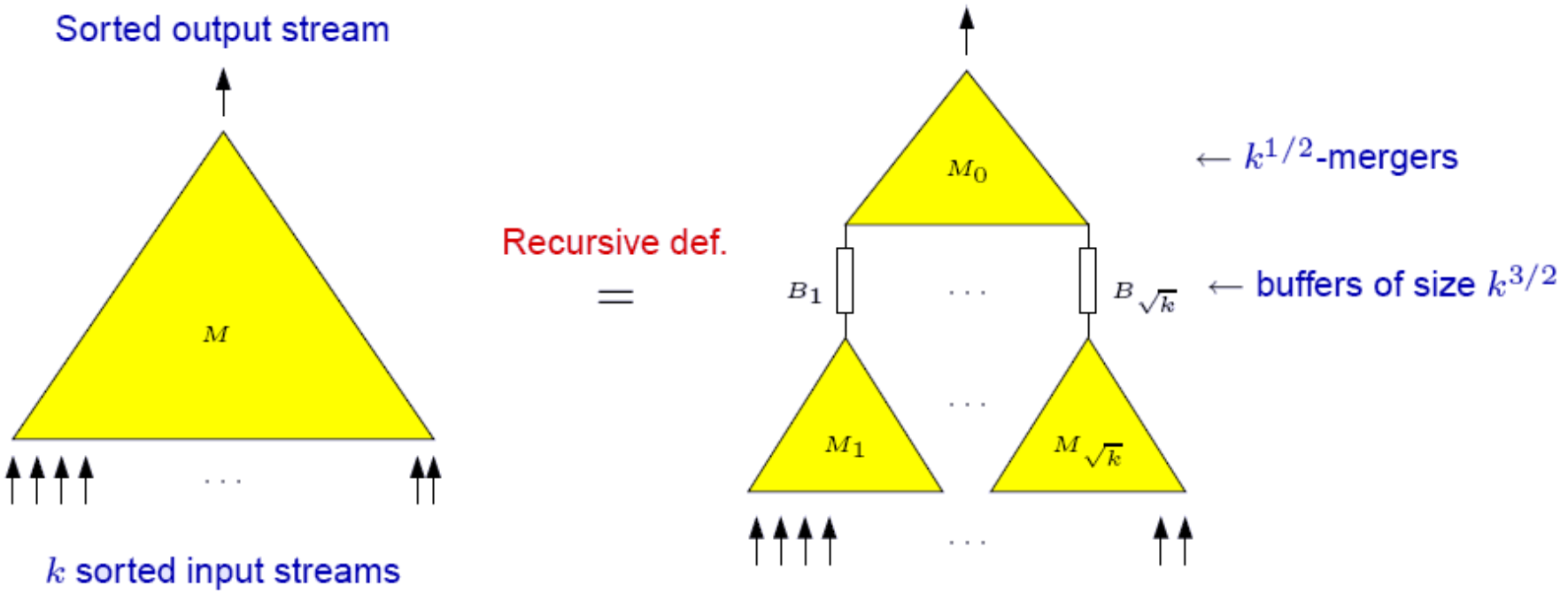
K-Merger

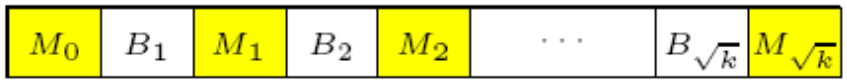


K-Merger


Sorted output stream
M
k sorted input streams
Recursive def.
=
M_0
$\leftarrow k^{1/2}$-mergers
B_1
$B_{\sqrt{k}}$
$\leftarrow$ buffers of size $k^{3/2}$
M_1
$M_{\sqrt{k}}$

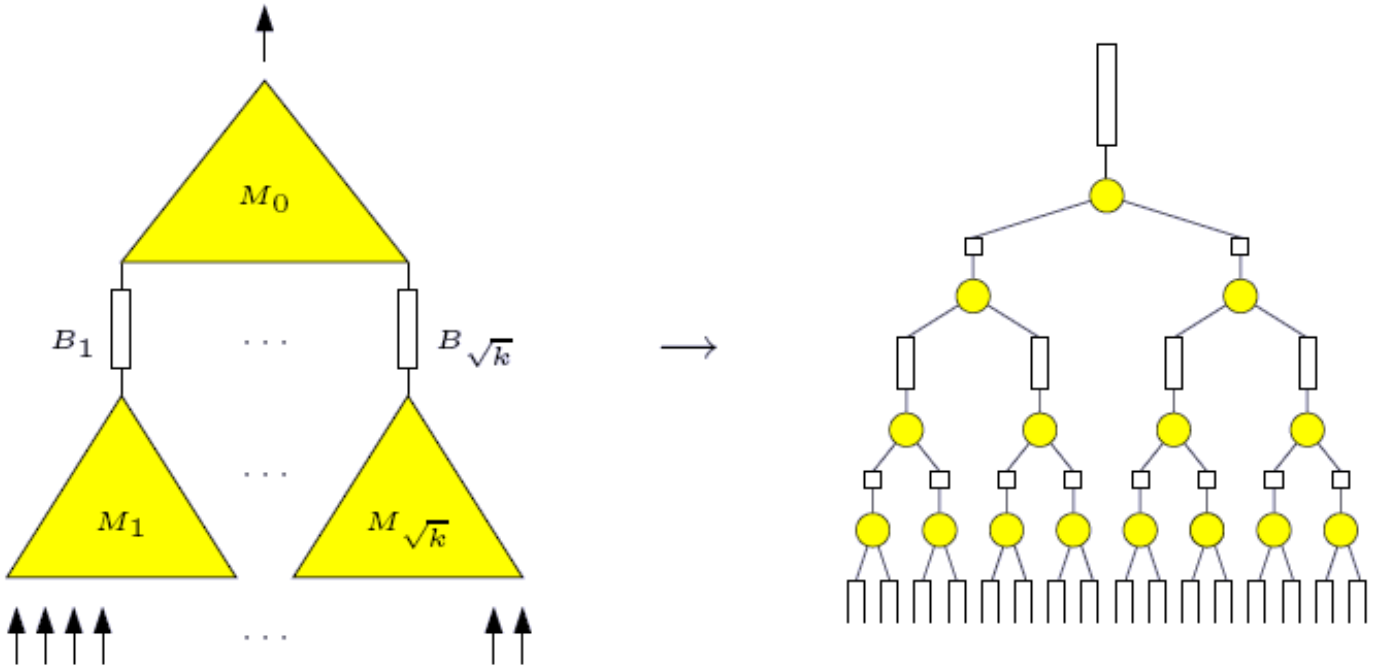
K-Merger



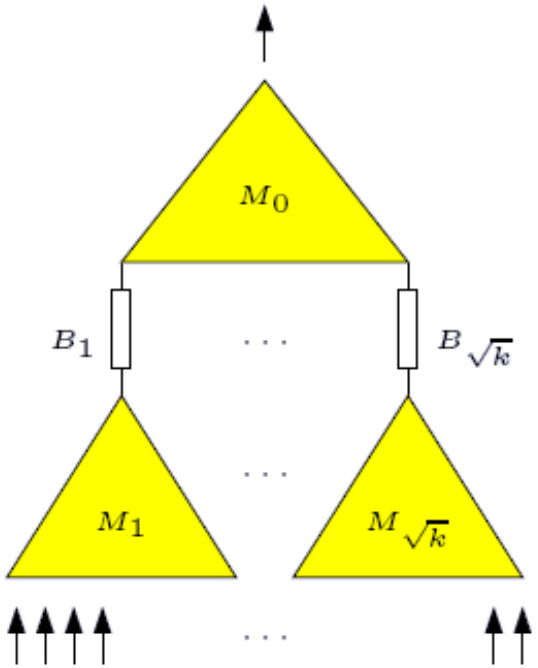


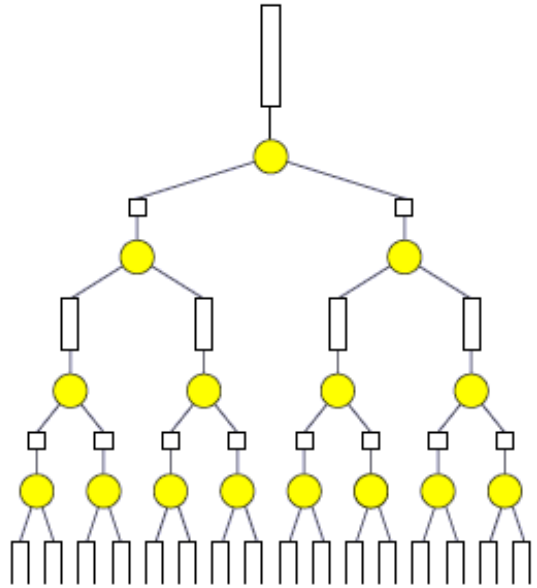
Recursive Layout

K-Merger



K-Merger





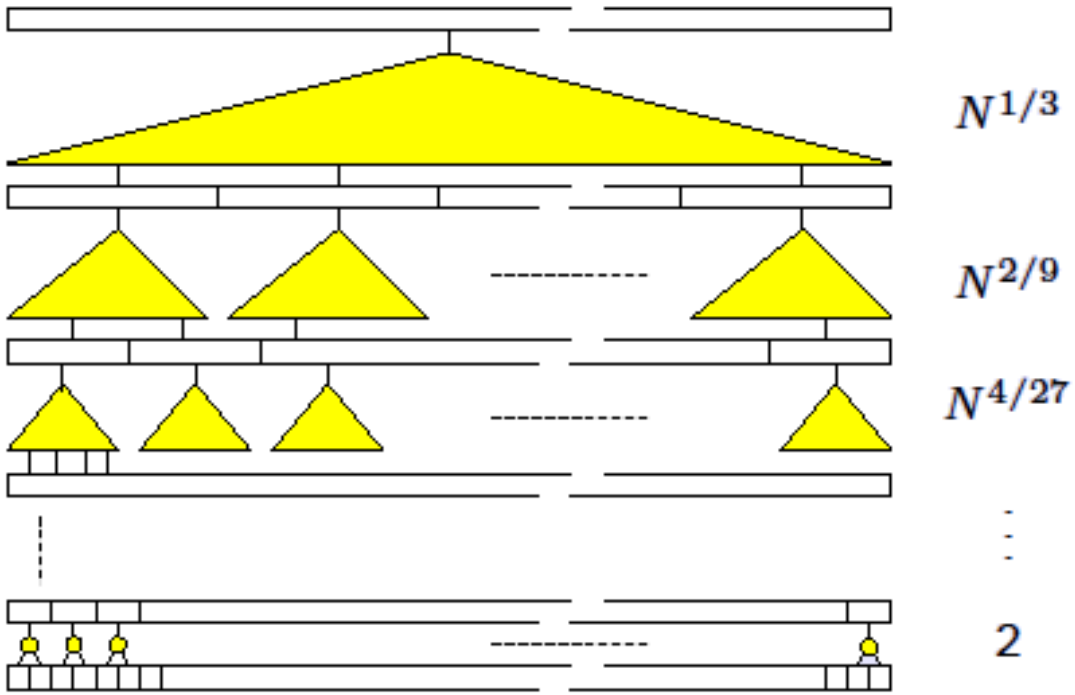
Procedure **Fill**(v)
  while out-buffer not full
    if left in-buffer empty
      Fill(left child)
    if right in-buffer empty
      Fill(right child)
    perform one merge step

Lemma

If $M \geq B^2$ and output buffer has size k^3 then $O(\frac{k^3}{B}\log_M(k^3) + k)$ I/Os are done during an invocation of **Fill**(root)

Funnel Sort

- Divide input in $N^{1/3}$ segments of size N^{2^3}
- Recursively Funnel-Sort each segment
- Merge sorted segments by an $N^{1/3}$-merger



- $T(N) = N^{1/3}T(N^{2/3}) + O(N/B \log_{M/B} N/B + N^{1/3})$ and $T(B^2) = O(B)$ $\Rightarrow T(N) = O(\text{sort}(N))$

References

- **Cache oblivious algorithms and data structures**
  Lecture notes by Erik D. Demaine.