

*External-Memory Algorithms with
Applications in GIS - (L. Arge)*

Enylton Machado

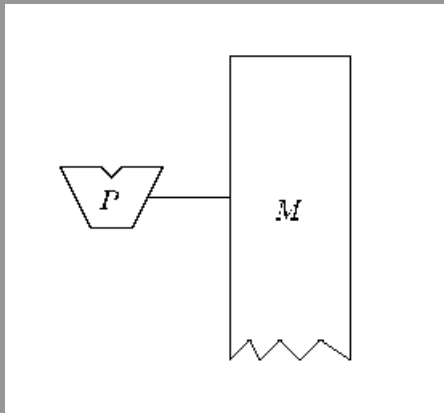
Roberto Beauclair

{ machado,tron } @ visgraf.impa.br

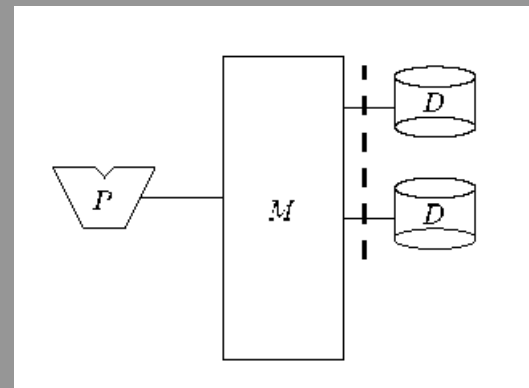


Theoretical Models

- Random Access Machine
 - Memory: Infinite Array.
 - Access cost is constant.



- External-Memory (I/O) Algorithms
 - Memory access time depends on the type of memory.



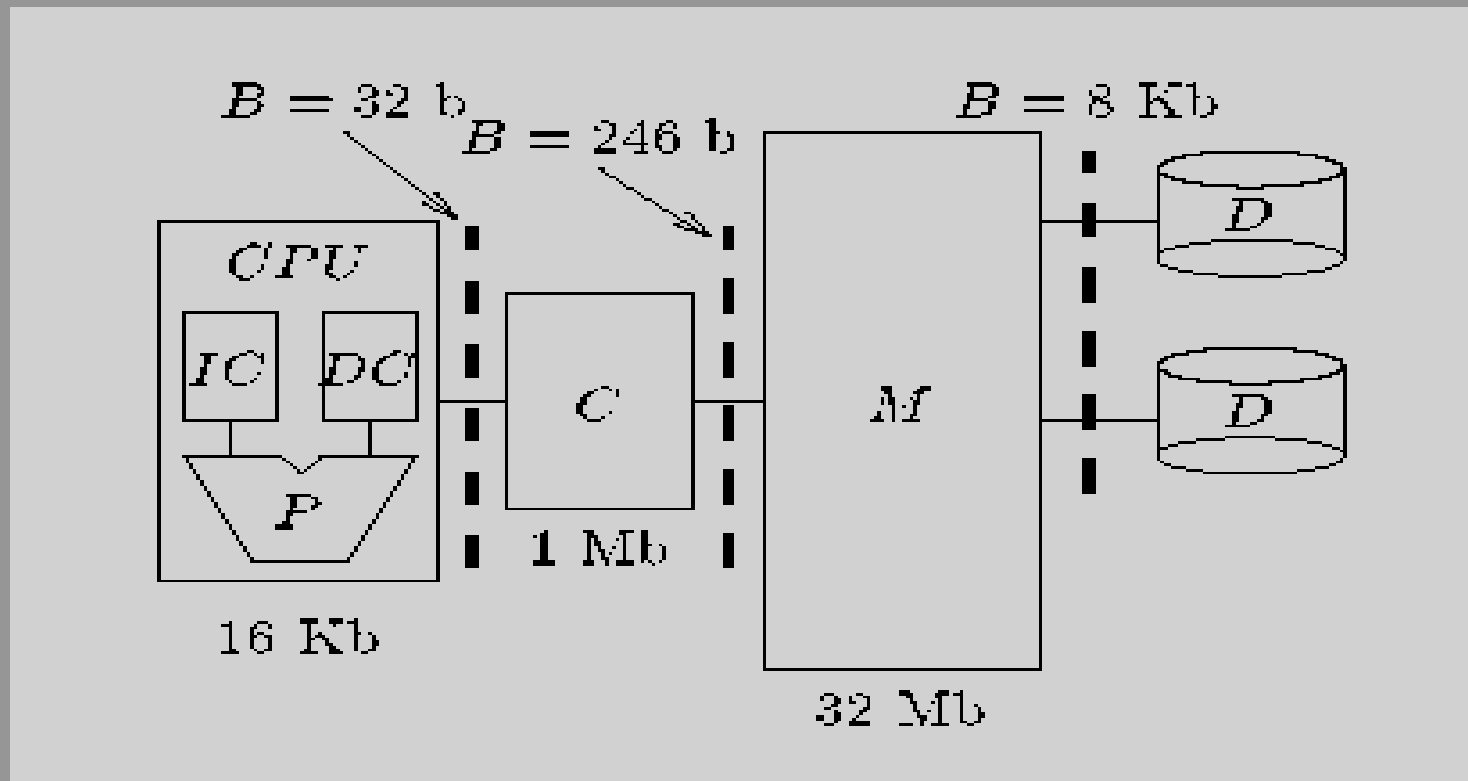
The Parallel Disk Model

- Parameters:
 - N ▶ elements in the problem instance.
 - M ▶ elements that can fit in internal memory.
 - B ▶ elements per disk block.
 - $M < N$, $1 \leq B \leq M/2$.
- I/O Operation ▶ Read/Write a disk block.
- I/O Complexity ▶ Number of I/O.



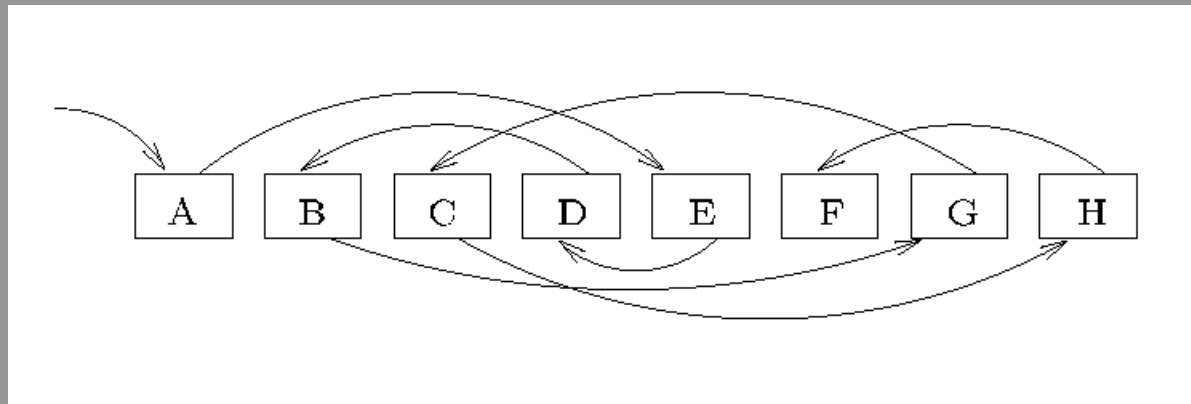
“Real” Machine

- Several Levels of Memory.



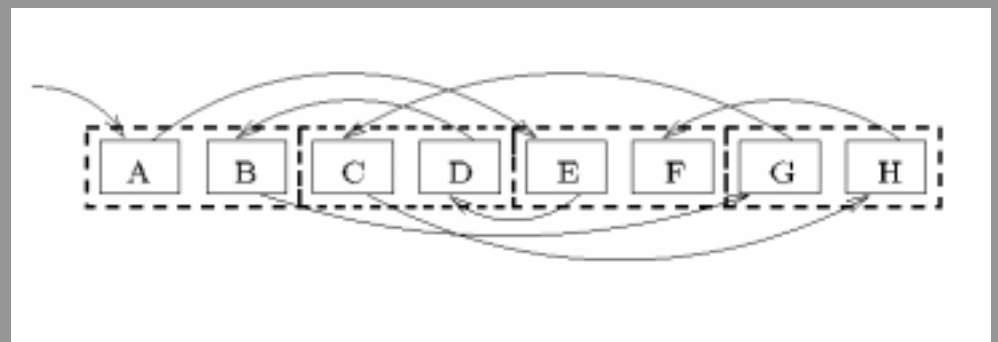
The List Ranking Problem

- Input:
 - Linked list of unsorted vertices.
- Problem:
 - Determine number of links to the end.



The List Ranking Problem

- RAM-Complexity
 - Traverse the list ranking the vertices
N-1, N-2, ...
 - $O(N)$ time.
- I/O Complexity
 - Same algorithm.
 - Internal mem: 2 blocks
 - Block: 2 data elem.
 - Paging policy: LRU



External Memory Bounds

- $n = N/B$ ▶ # of I/O to read entire input.
- $m = M/B$ ▶ # blocks to fit in memory.
- Linear # of I/Os: $O(n)$



Typical Bounds

- Scanning:

$$\Theta(N/B) = \Theta(n) \quad (\text{Linear})$$

- Sorting:

$$\Theta\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right) = \Theta(n \log_m n)$$



Paradigms for Designing I/O-Efficient Algorithms

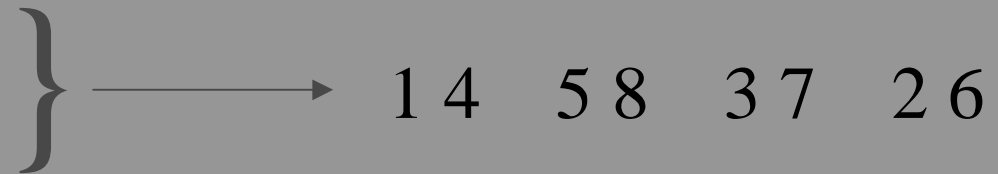
- Merging
- Distribution
- I/O versions of data structures



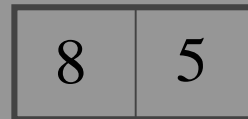
Merge Sort

- Run formation phase.
- Merge runs.

4 1 8 5 7 3 6 2



1 4 5 8 2 3 6 7



1 2 3 4 5 6 7 8



Distribution Sort

- Distribute elements in a “bucket” into m smaller ones.
- Elements in the first smaller than the ones in the second and so on.

4 1 8 5 7 3 6 2

Pivot: 1 4 8

4 1 3 2

8 5 7 6

Pivot: 1 2 4

5 6 8

1 2

4 3

5 6

8 7

1 2 3 4 5 6 7 8



Buffer Tree Sort

- “Lazy” operations:

- Collect blocks in buffers.

- *Buffer-emptying process:*

- When runs are full.
- Load to main memory, sort elements and flush to next level.

4 1 8 5 7 3 6 2

6	2
---	---

12 34 56 78



Buffer Tree Sort

- To report elements in sorted order:
 - **Empty all buffers.**
 - All elements in leaves.
 - Simply scan



External Memory Priority Queue

- Use a search tree structure (smallest element in the leftmost leaf).
- To **deletemin**: empty all buffers.
- To **insert**: check against elements in memory.
 - Hold the smaller ones and insert the largest.



Sorting on Parallel Disks

- Disk Striping ► synchronous heads.
- Sorting is not optimal due to internal memory size.

$$\Theta\left(\frac{n}{D} \log_{\frac{m}{D}} n\right) \text{ instead of } \Theta(n \log_m n)$$



Optimal Sorting on Parallel Disks

- Merge
 - *forecasting.*
- Distribution
 - blocks of the same bucket on different disks.
- Buffer tree
 - *buffer emptying = distribution.*



External-Memory Computational Geometry Algorithms

- GIS
 - Thematic Maps stored as layers
 - geometric: roads, cities, etc.
 - abstract: population density, land utilization, pollution level
 - Fundamental Operation: Map overlaying
 - Agricultural land: land utilization X pollution levels
- Many important problems from computational geometry are abstractions of important GIS operation.



GIS

- Map overlaying problems:
 - *Line-Breaking*: computing the intersections between the line segments making up the maps.
 - *Red/Blue intersection*: given two set of non-intersecting segments, red and blue, compute all intersection red-blue segment pairs.



GIS x Computational Geometry

- Many important problems from computational geometry are abstractions of important GIS operation:
 - *range searching*: finding all objects inside a region;
 - *planar point location*: locating the region of a given city lies in;
 - *region decomposition*: trapezoid decomposition, triangulation, and convex hull computation.
- GIS systems frequently store and manipulate enormous amounts of data, and they are thus a rich source of problems that require good use of external-memory techniques.



External-Memory Algorithms

- Additional parameters:

$K \rightarrow$ number of queries in the problem instance

$T \rightarrow$ number of elements in the problem solution

$k = K/B \rightarrow$ number of query blocks

$t = T/B \rightarrow$ number of solution element blocks



External-Memory Algorithms

- $O(N \log_2 N + T) \Rightarrow O(n \log_m n + t)$
- *plane sweeping* \Rightarrow *distribution sweep*
 - *batched construction of persistent B-trees*
 - *batched filtering*
 - *external fractional cascading*



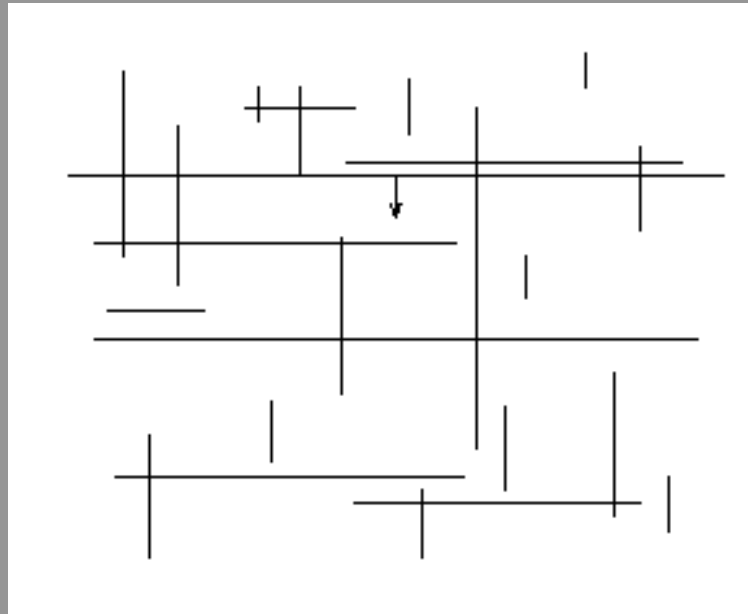
External-Memory Algorithms

- Orthogonal line segment intersection
 - *distribution sweep*
- Batched range searching
 - *external segment tree data structure*
- Red/Blue line segment intersection
 - *external fractional cascading*
- Other important external-memory computational geometry results

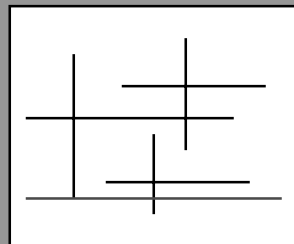
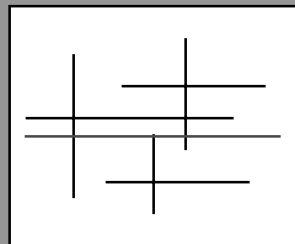
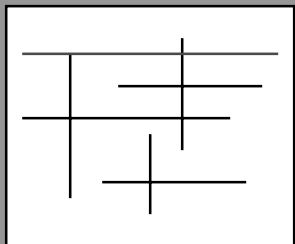
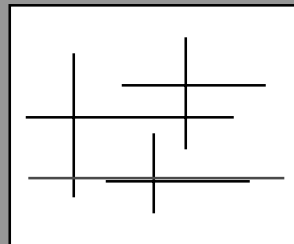
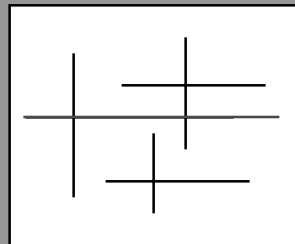
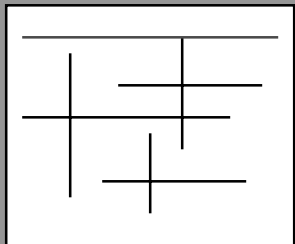
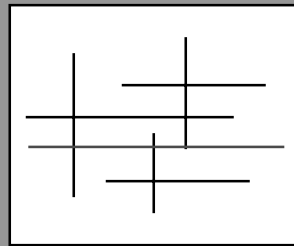
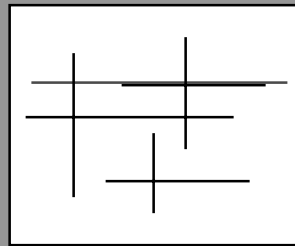
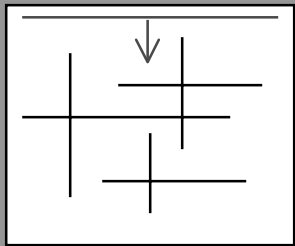


Orthogonal Line Segment Intersection

Reporting all intersecting orthogonal pairs in a set of N line segment in the plane parallel to the axis.



Plane-Sweep Paradigm



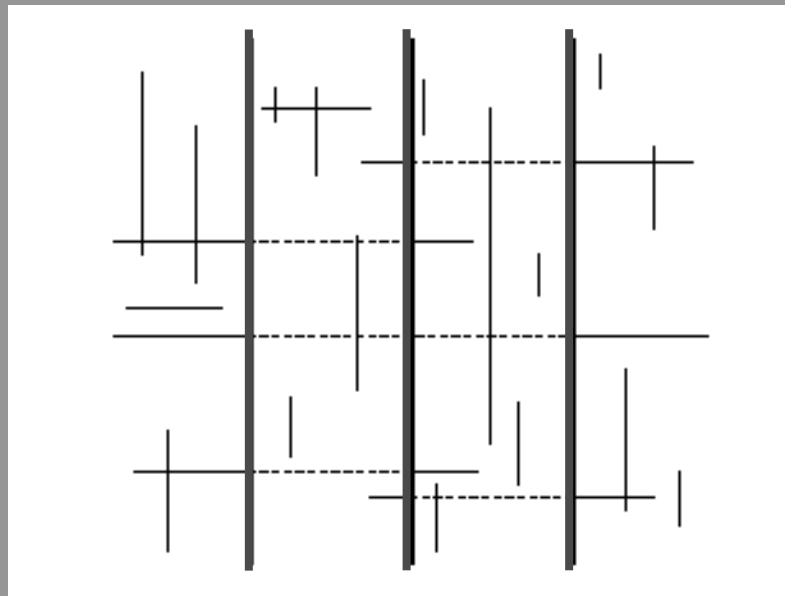
- 1) sort all endpoints by y-coordinate
- 2) when the top endpoint of vertical segment is reached, it is inserted in the search tree
- 3) when the bottom endpoint segment is reached, it is removed from the tree
- 4) when a horizontal segment is reached, a range query is made on the search tree

This way the tree at all times contains the segments intersection the sweep line.



Orthogonal Line Segment Intersection

Distribution Sweeping: combining the distribution and the plane-sweeping paradigms.



Distribution Sweeping

- 1) divide the plane into m vertical *slabs* and sweep from top to bottom
- 2) when a top endpoint of vertical segment is reached, we insert the segment in an *active list* associated with the *slab* containing the segment
- 3) when a horizontal segment is reached we scan all through all the *active lists* associated with the *slabs* it completely spans

The process finds all intersections except those between vertical and horizontal that do not completely span vertical slabs. These are found after distributing segments to the slabs, when the problem is solved recursively for each slab.

- 4) to find components of the solution between segments in the same *slab*, the problem is then solved recursively in each *slab*, to ensure that one block of data from each *slab* fits in main memory.



Orthogonal Line Segment Intersection

Using the Buffer Tree: extend the basic buffer tree with a range search operation.

- A range search operation on the *buffer tree* is done in the same way as insertions and deletions;
- When we want to perform a range search we create a special element which is pushed down the tree in a lazy way during buffer-empty processes, just as all other elements.



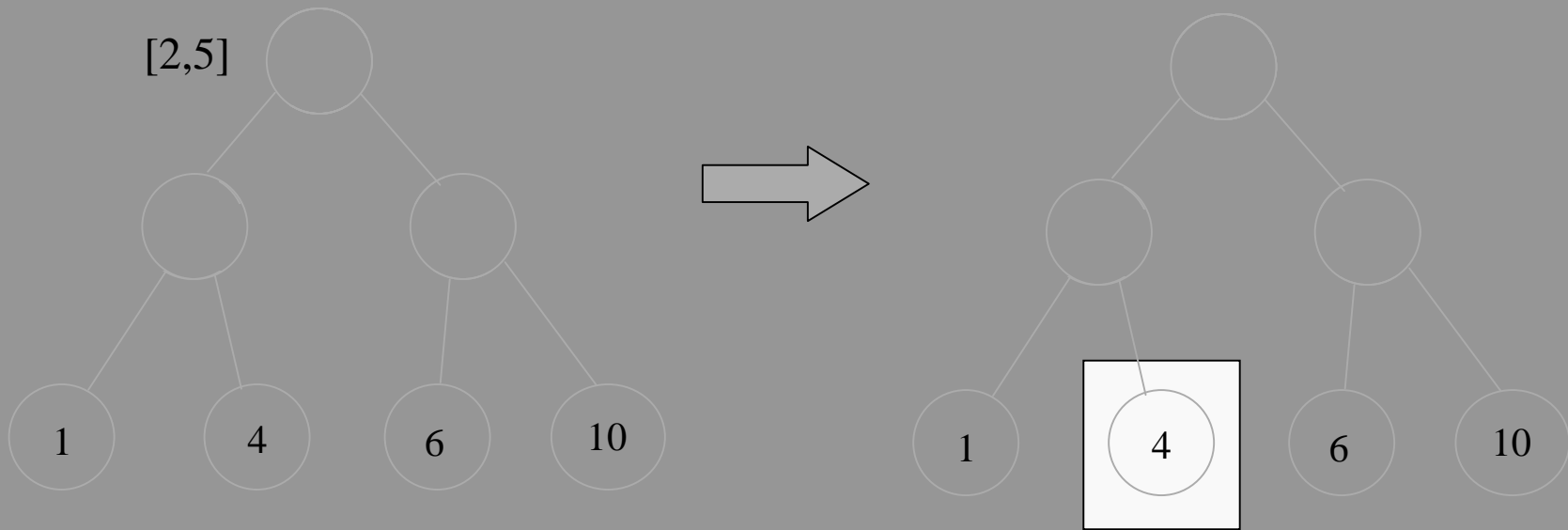
Extensions in the Buffer Tree

- 1) when we meet a range search element in a buffer-emptying process, instructing us to report elements in the tree between x_1 and x_2 , we first determine whether x_1 and x_2 are contained in the same subtree among the subtrees rooted at the children of the node in question.
- 2) if this is the case, we just insert the range search element in the corresponding buffer. Otherwise, we “split” the element in two, one for x_1 and one for x_2 , and report the elements in those subtrees where *all* elements are contained in the interval $[x_1, x_2]$. The splitting only occurs once and after that the range search element is treated like inserts and deletes in buffer-emptying process, except that we report the elements in the sub-trees for which all elements are contained in the interval.



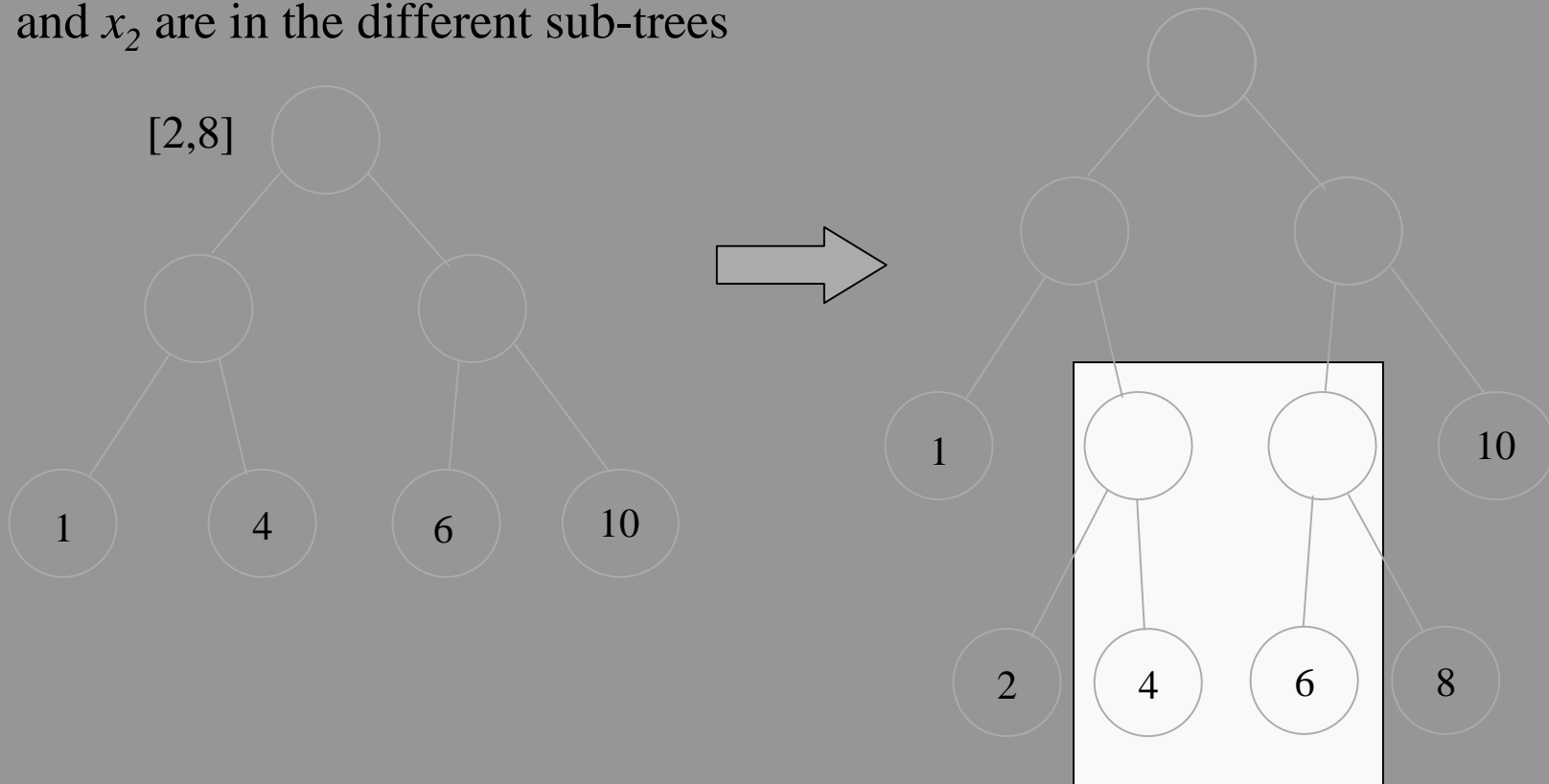
Extensions in the Buffer Tree

x_1 and x_2 are in the same sub-tree



Extensions in the Buffer Tree

x_1 and x_2 are in the different sub-trees

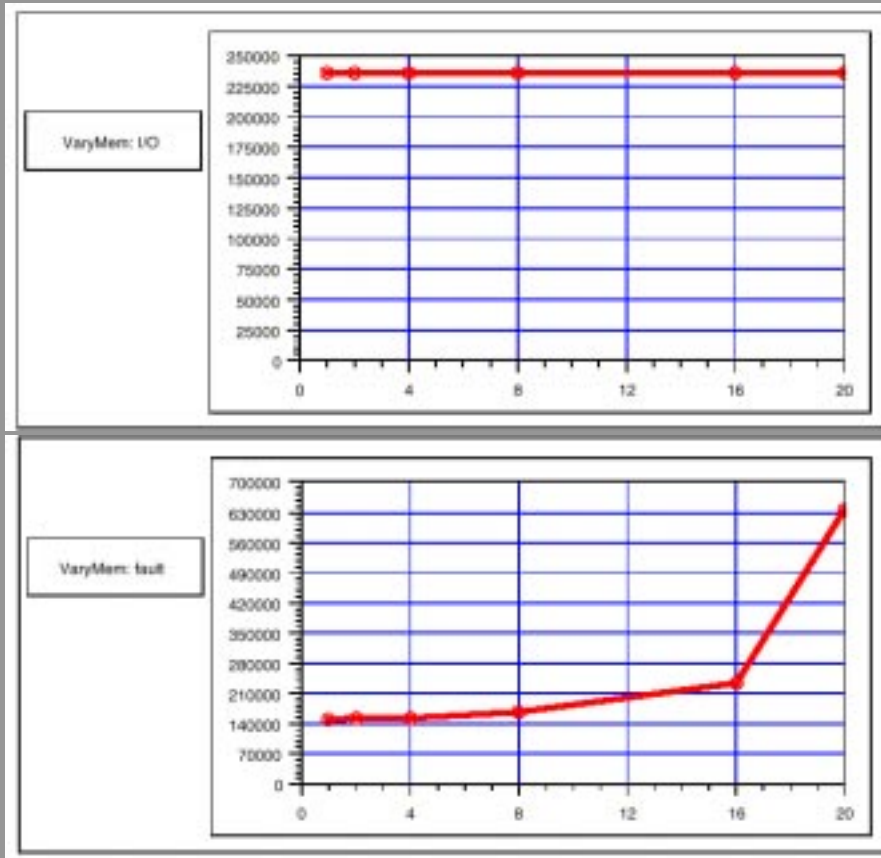


Experimental Results of Orthogonal Line Segment Intersection

- Chiang considered four algorithms:
 - *Distribution (sweeping algorithm)*
- Variants of plane-sweep:
 - *B-Tree*: external memory sort and B-Tree
 - *234-Tree*: external memory sort and 234-Tree (generic search structures tree equivalent to a red-black Tree)
 - *234-Tree-Core*: internal merge sort and 234-Tree (the OS handle the page faults)
- Sun SPARC-10 with 32 Mb (page size = 4 Kb)



Experimental Results of Orthogonal Line Segment Intersection

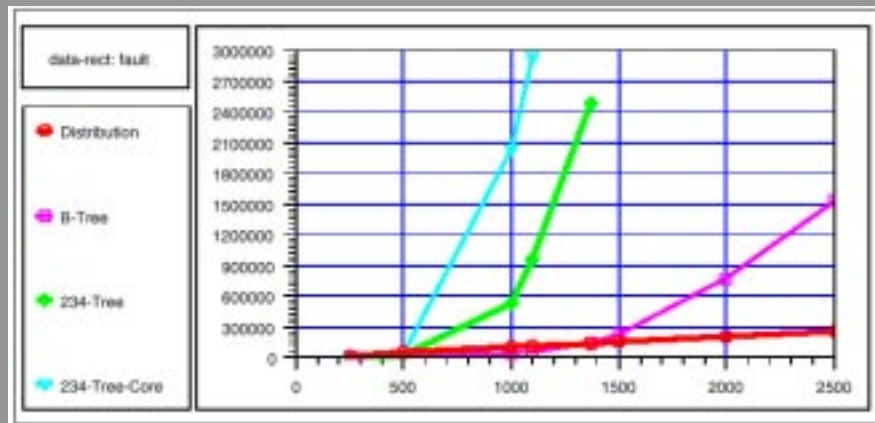
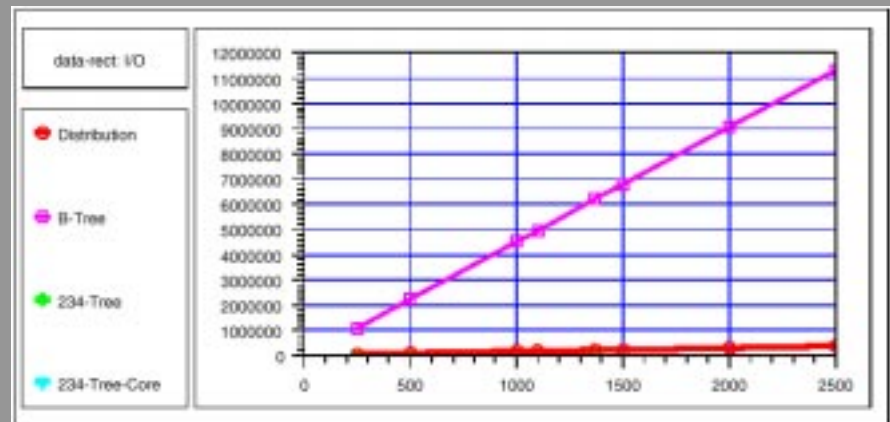
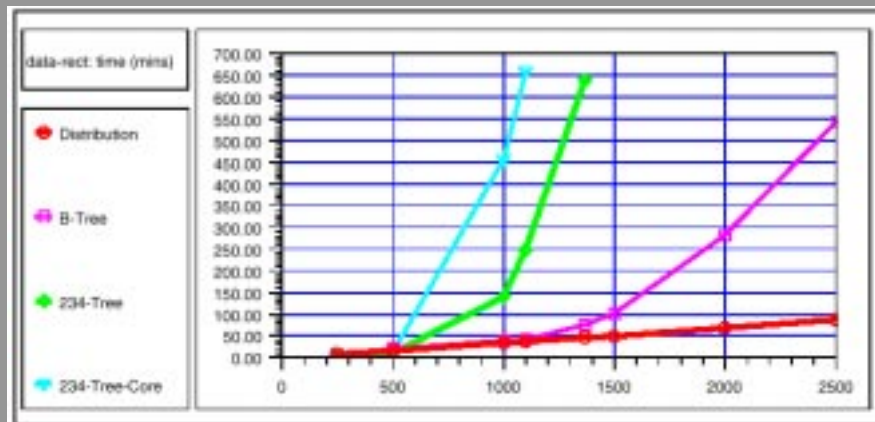


* X-axis: size of the main memory used (Mb)

The main memory available for use is typically much smaller than would be expected.

The reason is that a lot of daemons are also taking up memory.

Experimental Results of Orthogonal Line Segment Intersection



* X-axis: # segments (x 1000)

- *234-Tree-Core* performs the best for very small inputs, but as input size grows the performance quickly becomes worse;
- *234-Tree* always runs slowest and *Distribution* always the fastest, because the search tree structure is not small enough to fit into internal memory;

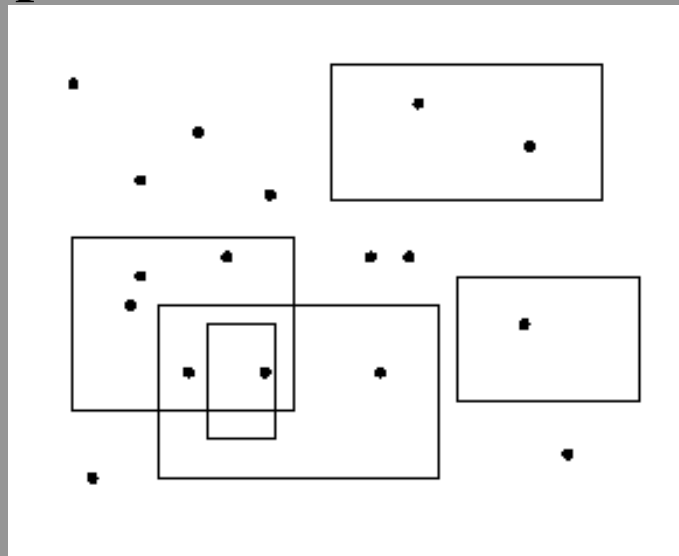
Conclusions of the Experimental Results

- The cost of I/O to solving a problem can be very important;
- Algorithms developed for main memory would not be able to solve problems in practice;
- The *buffer emptying* algorithm for buffers containing search elements is quite complicated, so a worse performance could be expected of the *buffer tree* algorithm compared to the *distribution sweep* algorithm.



Batched Range Searching

Given N point and N (axis-parallel) rectangles in the plane, the problem consists of reporting for each rectangle all points that lie inside it.



Batched Range Searching

Internal-memory

– *plane-sweep and segment-tree as data structure*

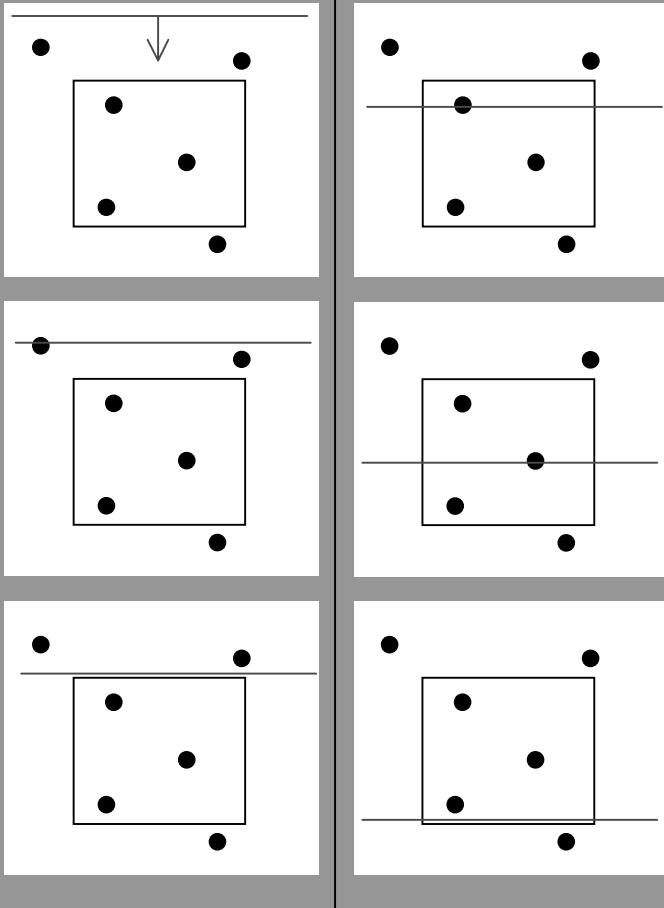
“*segment-tree* is a dynamic data structure used to store a set of N segments in one dimension, such that given a query point all segments containing the point can be found in $O(\log_2 N + T)$ time - *stabbing query*”

External-memory

– *external segment tree*



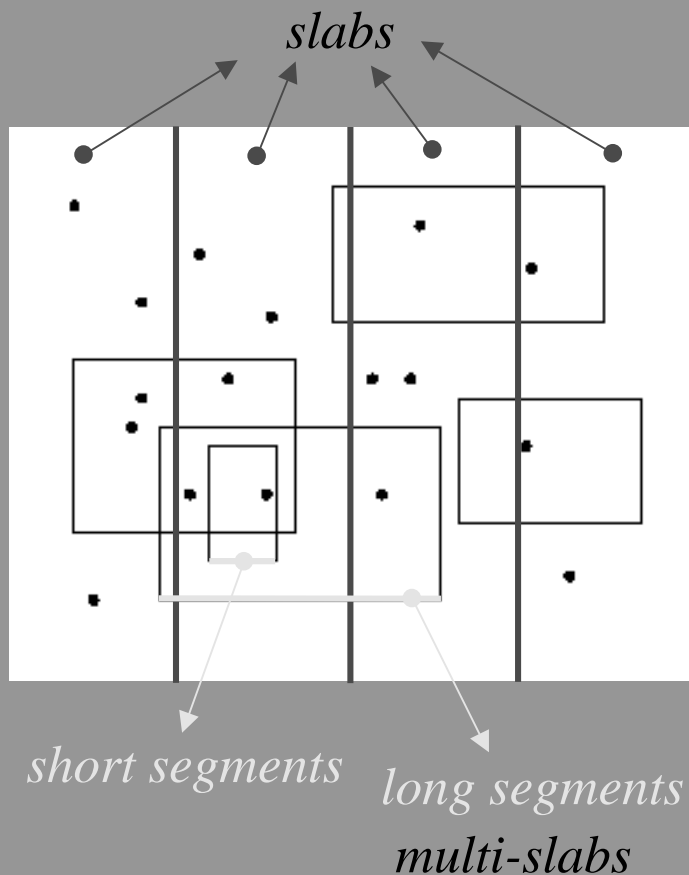
Plane-Sweep and Segment Tree



1. A vertical sweep with a horizontal line is made
2. When a top horizontal of a rectangle is reached, it is inserted in a *segment tree*
3. The *segment tree* is deleted when the corresponding bottom segment is reached
4. When a point is reached in the sweep, a *stabbing query* is performed with it on the *segment tree* and all rectangles containing the points are found.



External Segment Tree



Given an *external segment tree* with “empty buffers”, a *stabbing query* can be answered by searching down the tree for the query value, and at every node encountered report all the *long segments* associated with each of the *multi-slabs* that spans the query value.

However, because of the size of the nodes and the auxiliary *multi-slab* data, the *external segment tree* is inefficient for answering single queries. A *buffered query* can be used to solve this problem.



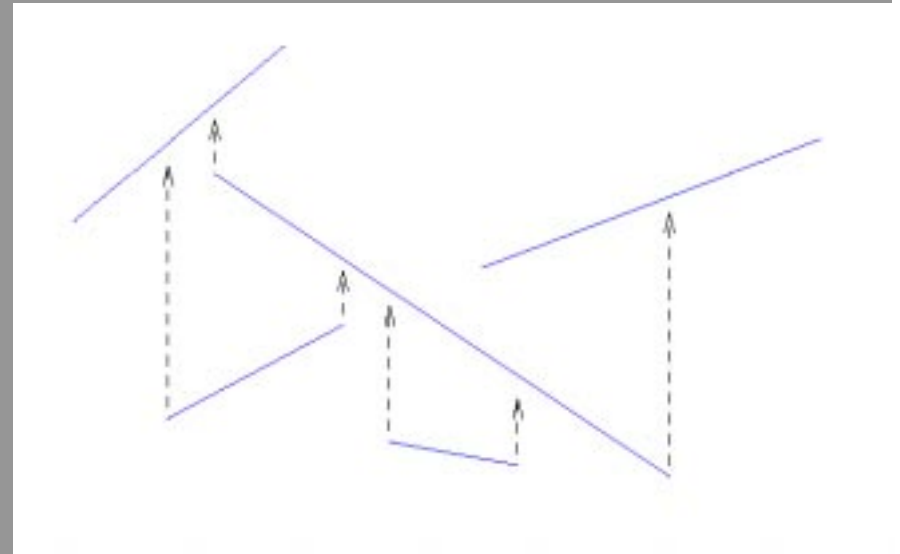
Red/Blue Line Segment Intersection

- Problem:
 - Two sets of internally non-intersecting segments.
- Task:
 - Report all intersections between segments.
- *Distribution sweep* and *buffer trees* not suitable.



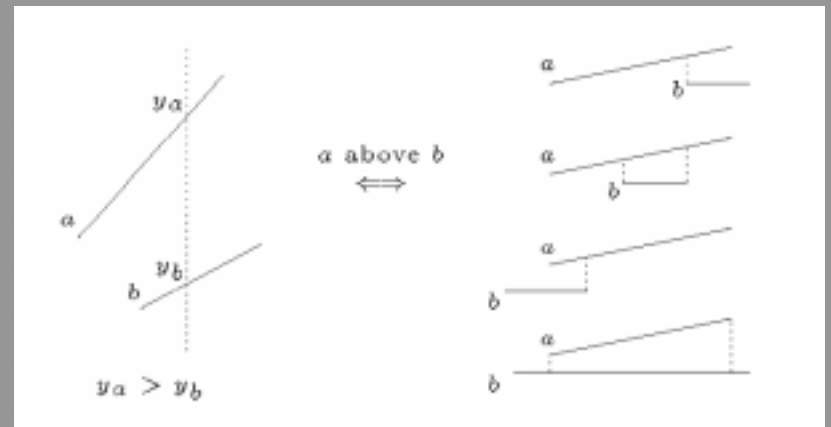
Endpoint Dominance Problem

- Can be used to sort non-intersecting segments in the plane.
- Problem:
 - N non-intersecting segments.
- Task:
 - To find segment above each endpoint.



Endpoint Dominance Problem

- a above b if y_a above y_b .
- If two segments are comparable, it is sufficient to consider only the four endpoints.



Endpoint Dominance Problem

- **Internal Memory**

- *Plane sweep:*
 - *Sweep from left to right*
 - *Insert in a search tree when reach left endpoint.*
 - *Remove when reach right endpoint.*
 - *Perform a search in the tree for every endpoint.*
- only segments which cross the sweep lines are stored in the tree.

- **External Memory**

- *Distribution sweep and buffer trees not optimal.*
- *Buffer tree*
 - old segments due to laziness.
- *Distribution sweep*
 - Need to compute total order and EDP is used to do so optimally.



So What?

- Construct ext. mem. segment tree of the projections of segments onto **X**-axis.
- Find segment directly above (in **y**) for each node -- **dominating segment**.
- Find **global dominating segment** by doing so to all nodes in the path from root to leaf.



External Red/Blue Line Segment Intersection Algorithm.

- Use algorithm to EDP (i.e. the ability to sort non-intersecting segments) .
- Two Steps:
 - Presort both *blue* and *red* segments (not endpoints).
 - Use a variant of the *distribution sweeping*.
- Step 1: Construct sets T_r and T_b
 - $T_r \Rightarrow$ union of red segments and blue endpoints.
 - $T_b \Rightarrow$ union of blue segments and red endpoints.



External Red/Blue Line Segment Intersection Algorithm.

- Step 2: Plane sweep
 - Task: report intersection between long segments of one color and long or short segments of the other color. (done twice)
 - For T_r :
 - Scan T_r from top to bottom.
 - Insert small blue segment in active list of corresponding slab when its top endpoint found.
 - Scan such list when long red encountered.



Other External-Memory Computational Geometry Algorithms

- Goodrich -
 - *pairwise intersection of N rectangles,*
 - *all nearest neighbors for N points in the plane,*
 - *Area of union of N rectangles in the plane,*
 - *Convex hull (extending Graham's scan)...*
- L. Arge -
 - *Extension of some to d dimensions.*



Transparent Parallel I/O Environment

- C++ set of template classes to allow programmers to abstract away I/O.
- 3 Components:
 - Block Transfer Engine (BTE).
 - Moves data blocks to and from disk
 - Memory Manager (MM).
 - Runs on top of BTE managing memory resources.
 - Access Method Interface (AMI).
 - Provides high-level interface to programmers.



Transparent Parallel I/O Environment

- Tested for *convex hull* and *list ranking*.
- Allows a large degree of overlapping between computation and I/O.
- L. Arge recently reported good results for the *pairwise rectangle intersection problem*.

