Dealing with Massive Data: External Memory Algorithms
Parallel disks.

Time for rotation \( T \) time for seek \( T \) time to transfer data.

Amortized search time by large block transfer so that

Time for rotation \( T \) time for seek.

I/O crisis.

Magnetic Disk Drives as Secondary Memory
\[ \frac{B}{Z} = z, \ \frac{B}{\mathcal{O}} = b, \ \frac{B}{M} = m, \ \frac{B}{N} = n \]

Notational convenience (in units of blocks):

- Output size: \( Z \)
- Number of queries: \( \mathcal{O} \)
- Number of CPUs: \( P \)
- Number of independent disks: \( D \)
- Size of disk block: \( B \)
- Size of internal memory: \( M \)
- Problem data size: \( N \)

[Agarwal & Vitter 88; Vitter & Shriver 90, 94]
A "Real" Machine
\[ \text{Sorting} \cong \text{Computational Geometry} \quad \bullet \]
\[ \text{Graph problems} \cong \text{Permutation} \quad \bullet \]
\[ \ldots \quad \star \]

For other problems [CGCTLY95, AYL95],

\[ \text{Searching}: \Theta (\log n) \quad \bullet \]

Online problems: \[ \star \]

\[ \left( \left\{ u \log \frac{d}{u} \right\}_{u \in N} \min \right) \Theta = \left( \frac{d}{N} \frac{\log \frac{d}{N}}{\log d} \right) \Theta = \left( \frac{d}{N} \frac{\log \frac{d}{N}}{\log N} \right) \Theta \quad \bullet \]

\[ \left( \frac{d}{u} \right) \Theta = \left( \frac{d}{N} \right) \Theta \quad \bullet \]

Searching (touch problem): [88] \[ \star \]

Batched problems: [94] \[ \star \]

Fundamental Bounds
To get an optimal sorting algorithm, use disks independently.

\[ m \approx D \cdot \log \frac{b}{m} \cdot \log \frac{w}{m} \text{ when } D. \]

Ratio of I/O bounds \( I/O \text{ bound } \Theta \)

\[
\left( \frac{m}{N} \left( \frac{b}{a} \right) \log \frac{w}{m} \right) \Theta \text{ increases to } \Theta
\]

\[
\left( \frac{m}{N} \left( \frac{b}{a} \right) \log \frac{w}{m} \right) \Theta \text{ bound } \Theta
\]

\[ \therefore \text{ I/O bound } \Theta \]

```
<table>
<thead>
<tr>
<th>step</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>16</td>
<td>15</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>33</td>
<td>36</td>
<td>37</td>
<td>38</td>
</tr>
</tbody>
</table>
```

Disk Stripping involves using the disks in lock step.

Layout for data:

**Disk Stripping:** \( D = 5 \) disks, block size \( B = 2 \)
Outline

1. Sorting with multiple disks.
2. Techniques for solving batched dynamic problems.
   - Distribution sort.
   - Merge sort.
3. Online data structures.
   - Empirical results.
     - Red-blue orthogonal rectangle intersection.
     - Distinct boundary sweeping (via TPFF programming environment).
4. String processing, SB-trees.
   - Range searching.
5. Dynamic memory allocation and lower bounds.
   - Experiments on bulk-loading R-trees.
   - B-trees, buffer trees, R-trees, etc.
Simple Randomized Merging \cite{BKV96} •

Share Sort \cite{AGG99} & Plaxton 94 •

Greedy Sort \cite{Nodine, Vitter 91} •

Three methods:

\[ \left( \frac{u^m \log \frac{d}{u}}{O(\log N)} \right) O = \frac{\text{OS/I \#}}{\#} \leftrightarrow \]

\[ \text{OS/I} \left( \frac{d}{u} \right) \Theta = \left( \frac{\text{DD}}{N} \right) \Theta \]

If each pass uses \# ⋆

\[ I - u^m \log \frac{d}{u} = \frac{u \log \#}{\text{passes}} \]

Merge together ⋆

\[ \text{Merge initial sorted runs of length m blocks (one memory load).} \]

Merge Sort with \( \mathcal{D} \) disks ⋆
In as few I/Os as possible, read the "next" $H$ blocks. Why Merge sort with $D$ disks is hard.

$$\frac{2}{m} \approx \frac{2H}{W}$$
Balance can quickly deteriorate. Only $p/R$ parallel reads needed to load leading blocks into memory.

Good balance initially:

Each run is striped, but the starting disk of the runs are staggered.

$R = 8$ runs on $D = 4$ disks.
Cannot be generalized to $R > 2$.

Reduce necessary buffer space by half.

Run 2: D C B A (striped in reverse order)

Run 1: A B C D

Can achieve perfect balance for merging two runs, $R = 2$.
Output the smallest $B$ items of the 2B items.

Read the two blocks with smallest and smallest maximum items

Merge procedure for each disk:

1. Do approximate merge
2. Interleave the „sorted“ runs. independently on each disk.
3. Use Columnsort to convert the approximate sorted output into a totally ordered output run.

Overall structure of each merge pass:
Forecast and Flush buffer management policy.

Uniformly random.

At any time, the disk containing the leading block of any run is

Each run is striped starting at a randomly chosen disk.
\[
\frac{d}{r} \log d \leq \frac{d}{r} \iff \frac{d}{r} \cdot \frac{c}{\ln d} \sim [\text{Classical Max Occupancy}]
\]

Classical Maximum Bucket Occupancy

(a: disks) Max Occupancy = 3.

Independent uniform distribution

(balls: blocks)
Starting bin of each chain is uniformly random.

Conjecture:

\[ \forall \text{chains} R, \text{bins} d \]

\[ \text{Max Occupancy} = 2 \]

\[ \forall \text{chains} R, \text{bins} d \]

\[ \text{Max Occupancy} \geq \text{Max Occupancy} \]

\[ \text{Classical Max Occupancy} \]

\[ \forall \text{chains} R, \text{bins} d \]

\[ (\text{disks}) \]

\[ 1, 2, 3, 4 \]

\[ \text{Center for Geometric Computing} \]

[BCV97, Knuth98]
I/O Performance of SRM, $R = \frac{m}{2}$

\[
\begin{align*}
E[\# \text{ reads}] & \sim \left\{ \begin{array}{l}
\frac{\ln D}{k \ln \ln D} \cdot \frac{n}{D} \\
\frac{\ln D}{k \ln \ln D} \cdot \frac{n}{D} \\
\frac{n}{D} \\
\frac{n}{D} \\
\end{array} \right. \\
\end{align*}
\]

Simulation of I/O performance ratio

\[
\frac{\text{IO}_{\text{SRM}}}{\text{IO}_{\text{DSM}}} \quad \text{for } m \approx (2k + 4)D:
\]

<table>
<thead>
<tr>
<th>$D$</th>
<th>$k = 5$</th>
<th>$k = 10$</th>
<th>$k = 50$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.56</td>
<td>0.47</td>
<td>0.37</td>
</tr>
<tr>
<td>10</td>
<td>0.60</td>
<td>0.52</td>
<td>0.40</td>
</tr>
<tr>
<td>50</td>
<td>0.71</td>
<td>0.63</td>
<td>0.51</td>
</tr>
</tbody>
</table>

SRM is better than striping!
Given that the buckets are stored in an online manner,

Difficulty is to store each bucket evenly across the disks,

\[
\left(u \log \left( \frac{d}{u} \right) \right)_O = O/I \neq \left( \frac{d}{u} \right)_O
\]

If each level of recursion uses \( \Theta \) disks

The number of levels of recursion is \( \log n \)

Append together the sorted buckets.

Sort the buckets recursively.

Divide the Hilbert space evenly into buckets.

Select \( S \) or \( (w) \Theta \) or \( \Theta \) partitioning elements that

Distribution sort with \( D \) disks

\[ D = \Theta \]
Max Occupancy = 3.

Hash (independent uniform distribution)

S balls (blocks) total per bucket.

S simultaneous load balancing problems (one per bucket).

Hash assignments load balancing problems. If \( N \) is large, or \( \frac{DB}{W} \) is large, then random assignment to disks works well.

Bucket Sort [V94]
shuffle (of the $S$ buckets onto the $D$ disks).

Output each memroyload by a round-robin placement (perfect)

and then extracting a part from each of several memroyloads.

Get a "typical" memroyload by permuting each memroyload

(and is therefore well-balanced among the $S$ buckets).

a "typical" memroyload contains more than $S \log S$ blocks

so that random assignment is not "balanced".

If $N$ (and $S$) are small and

$W \approx BD$
for each bucket $q$, \[
\frac{D}{\text{num}_n} \geq (p)^{\text{num}_q} \iff \]

- differ by at most 1.
- Maintain invariant that the largest values of
  \[
  \left\lfloor \frac{D}{A} \right\rfloor \times (A)^{\text{num}_n} = (p)^{\text{num}_q}
  \]
- \(A > p \geq 1\)
- \(\text{num}_n\) items in bucket $q$ written to disk $q$, \# items in bucket $q$ processed so far.
- Let \(\text{num}_q\) items in bucket $q$ processed on disks.

Online tracking of bucket distribution on disks.

Center for Geometric Computing

Jeff Vitter

External Memory Algorithms

BalanceSort (NV93)
Let $A$ be an $N$-processor PRAM algorithm such that
- $A$ reduces a problem of size $N$ to one of size $\alpha N$ in constant time.
- Parallel running time of $A$ is $\Theta(\log N)$.
- For each PRAM statement, sort the $N$ operands so that they are contiguous.

Simulate $N$ operations via a linear pass through the data.

I/O Complexity for $D = 1$:

$T(N) = O(sort(N)) + T(\alpha N) = O(sort(N))$.

Gives optimal EM algorithms for list ranking, Euler tours, expression tree evaluation, connected components of sparse graph.

Sometimes the sorting can be done in $O(N)$ I/Os because of constraints and assumptions $\text{[DDH97, SK97]}$.

Some problems like topological sorting, BFS, DFS are hard.
Random incremental construction.

* Batched filtering.

* Distribution sweep.

We improve this to use of virtual memory

\[ \frac{z + u^{\log (b + u)}}{O} \]

\[ (Z + N^{\log (O + N)}) \leq \]

* Triangulation.

* Batched planar point location.

* Trapezoidal decomposition.

* Batched range queries.

* 2-D and 3-D convex hulls.

* All nearest neighbors.

* General line segment intersection.

* Red-blue line segment intersection.

* Orthogonal rectangle intersection.

\[ [\text{CLTV93}, \text{AVV95}, \text{APRSv98a}, \text{APRSv98b}, \text{CFMR98}] \]
Red-blue line segment intersection important subproblem.

Find all farmland with level of pollution over certain threshold.
We consider the step: We focus on the case where input is unordered (not indexed). Occurs e.g. when input is:

- **Intermediate result.**
- We consider the step: We focus on the case where input is unordered (not indexed).

Rehearsal step: Validate intersections.
- Different maps (red-blue rectangle intersection).
- Region and compute intersections between rectangles from region and compute minimal bounding rectangles for each.

Filter step: Compute minimal bounding rectangles for each.
- In database literature often solved in two steps:

---

**Map Overlay Spatial Join**
Red-Blue Rectangle Intersection

Red rectangles are handled similarly.

(i) Remove rectangle from blue active list.

When bottom of blue rectangle is reached:
(ii) Find intersections with rectangles in red active list.

(i) Insert blue rectangle in blue active list.

When top of blue rectangle is reached:

rectangles intersecting vertical sweep line [BW80]:

Sweep plane while maintaining two active lists of red and blue
Key point: Updates and queries are batched.

Sequence of operations $a_1, a_2, \ldots, a_N$ known beforehand.

Using general method for solving batched dynamic problems.

Solved in optimal $O(I/\text{OS})$ steps.

If size of active lists $W < M$, algorithm performs badly.

Red-Blue Rectangle Intersection
I. Divide plane into \( \lceil \sqrt{m/N} \rceil \) slabs, each with \( u \lceil \log \log u \rceil \) endpoints.

2. Find \( Z \) intersections involving the part of a rectangle

3. Recursively solve problem in each slab.

Performing Step 2 in \( O(\left( \frac{B}{L} + u \right)O \) *Levels of recursion*.

\( (u \lceil \log \log u \rceil )O = (u \lceil \log \log u \rceil )O \star \)

Sketch of External Solution [APRSV98]:
Maintain an active list for each multislab.

Perform top-down sweep:

- \( \mathcal{B} \) rectangles per multislab in internal memory.
- \( \mathcal{O}(m) \) multislabs (continuous ranges of slabs) uses \( \mathcal{W} \) slabs.

Key Ideas
Other cases handled similarly—in one sweep:

- (i) Remove "expired" red rectangles ("lazy" deletion).
- (ii) Report intersection with "non-expired" red rectangles.
- Report intersection with red rectangles.

At blue rectangle: Scan through all relevant multislab lists of

- At blue rectangle: Scan through all relevant multislab lists.
- At red rectangle: Insert into relevant multislab list.

Interssections between red centerpieces and tops of blue rects.

Sketch of sweep
General Technique: Colorable Problems [APRSV98]

**External Memory Algorithms**

- Batched semi-dynamic planar point location:
  \( (y + u)O \) space \( \frac{O}{I} \) \((y + u) \log (y + u))O

- Batched semi-dynamic planar point location:
  \( (u)O \) space \( \frac{O}{I} \) \((t + u) \log u)O

- Dim. rectangle intersection:
  \( (u)O \) space \( \frac{O}{I} \) \((t + u) \log u)O

- Dim. batched range searching:
  \( (u)O \) space \( \frac{O}{I} \) \((t + u) \log u)O

**Performance Using Technique:**

- I/O performance using technique

**Higher-dimension problems:**

- Technique can be used recursively (by decreasing number of slabs further)

**Algorithm is special case of general technique:**

**Proven using external segment tree [ARG95]:**

- Colorable problems can be solved using extra I/O factor of \( O(\log u) \).

**Batched dynamic version of static "colorable" problems:**

Jeff Vitter
Related Results

For general line segment intersection, \( O(\log b + u) \) I/Os for general line segment intersection.

Random incremental construction [GCMNR98] to get optimal

\( O(\log\log u) \) I/Os.

Persistent B-trees [CTLV93] to solve batched point location in

Red-blue line segment intersection in \( O(\log\log u) \) I/Os.

A large number of problems with GIS applications [AVV95] and external fractional cascading to solve

External segment tree used in conjunction with batched
Access-Oriented: Under development (for index structures).

- Scanning, merging, distribution, sorting, permuting, etc.

Framework-oriented: Implements a number of high-level, portable programs.

Make implementation easy (and portable), I/O-efficient (and efficient management).

OS often provides inadequate support for I/O and internal.

Many problems can be solved using small number of paradigms.

[TPiE] originally designed by Darren Venegas [Ven94].

TPiE' Hitp://www.cs.duke.edu/TPiE/
External Memory Algorithms

Pies' Distribution Access Method
<table>
<thead>
<tr>
<th>State</th>
<th>Object Category</th>
<th>Size</th>
<th>Hydrographic</th>
<th>Roads</th>
<th>TIGER/LINE Data From U.S. Census Bureau</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhode Island (RI)</td>
<td>7,013</td>
<td>0.4 MB</td>
<td>Hydrographic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connecticut (CT)</td>
<td>688,778</td>
<td>4.3 MB</td>
<td>Hydrographic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Jersey (NJ)</td>
<td>4,14,443</td>
<td>26.5 MB</td>
<td>Hydrographic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New York (NY)</td>
<td>156,468</td>
<td>25.7 MB</td>
<td>Hydrographic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>154,127,777</td>
<td>19.5 MB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(standard benchmark data for spatial databases)
Sun Sparstation 20 (Solaris 2.5), 32MB memory (TPIPE 12MB)

Performance Comparison with PBSM [DP96]
Can we take advantage of blocking and obtain an \( O(n \log n) \) I/O algorithm. 

Repeated insertion \( O(n \log N) \) I/Os. 

Building B-tree: 

Optimal if each operation is handled individually. 

Insert, delete, determine query in \( O(n \log N) \) I/Os. 

Each B-tree node fits in a block: 

\[ \text{B-tree Construction} \]
Inserting \( n \log u \) items in \( O \left( \frac{n}{\log n} \right) \) I/Os.

Every block touched \( O(1) \) times on each level.

Buffer-emptyping in \( O \left( \frac{n}{\log n} \right) \) I/Os.

When a buffer runs full, its items are pushed one level down.

Insertions are done “lazily”—items inserted into buffers.

Main idea: Logically group nodes together and add buffers.

The Buffer Tree [Argé 95]
Practically important: Hmsy97, vdbsw97, Ahly98

Bulk operations on R-trees

String sorting [Afgv97]

External heap [Fvk98]

Ordered Binary Decision Diagram manipulation [Arge99b]

KS96

External tournament tree (improved graph algorithms)

Improved graph algorithms (fast ranking) [Arge99]

Buffer technique has been used in a number of results:

Efficient Determination operation can be designed.

Delete, and Query operations can be handled similarly.
Internal node holds minimal bounding rectangle of each subtree.

Rebalancing: basically like in B-trees.

- Pan-out B.
- Data in leaves.

Structured like B-tree:

Structure for storing d-dimensional rectangles.
trees, ... have been proposed, surveyed in [GC9, GC98].

Several insert/split heuristics (R + -trees, R*-trees, Hildert's

Small overlap or perimeter desirable.

Minimal bounding rectangles allowed to overlap.

... containing $b$.

Visits all nodes with minimal bounding rectangle

Query with point $b$.

Querying R-trees
Handles all “bulk” operations.

- Modular design (all R-tree insert heuristics can be used).
- Conceptually simple (algorithm unchanged).
- Buffer technique immediately applies:

Questionable query performance, esp. in high dimensions.

- Can only handle construction—not e.g., “bulk updates.”

Rectangles are sorted (using space-filling curve)

[VDDBSVW97]

Bottom-up algorithms [RT85, KR93, DKLP94, LPE96]

Using repeated insertion takes \(O(N \log B \log u)\) I/Os.

Construction („bulk loading“) an R-tree
Buffers on all nodes for simplicity (buffer size $\Theta(B)$)

Building R-tree on road data (I/Os in thousands)

Naive repeated insertion:

Query with (1/10) hydro data (I/Os in thousands)

Buffer: Size of buffer $\frac{B^2}{4}$

$2B^2$
<table>
<thead>
<tr>
<th>Percentage</th>
<th>Querying</th>
<th>Building</th>
<th>Updating with 25% of the data</th>
<th>Method</th>
<th>Set Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For example, indexing constraints in constraint query languages:

- 4-sided query
- 3-sided query
- 2-sided query
- Diagonal corner query
Lower bounds: [SR95] Cannot achieve simultaneously

Halvespace queries: [AVELV98]

3-d Range queries: [96V75] ∗

- sided: [SR95] ∗

3-sided: [SR95] ∗

- sided: [RS94] ∗

- sided: [AVELV98] ∗
Every internal node is a **B-Tree (BT)**.

Need to be able to find position of string $s$ efficiently.

Pointers to $\Theta(p)$ strings associated with internal node.

\[
\text{Update possible in } O(\log |S| + K \log p) \text{ I/Os.}
\]

B-tree on set of pointers to strings (in lexicographical order).

SB-tree: a String B-tree [FG95]
Binary Trie contains characters from all $B - 1$ strings.

Binary Trie on set of $B - 1$ strings has $2B$ characters.
Blind Trie [Ajtai et al 84]

S = bcbabcba
Note: Comparison model

Use "lazy time" or SB-tree.

Total length \( n \) of short strings.

Number \( k \) of long strings.

Whether breaking string into characters is allowed.

Actual I/O bound depends on

\[
\frac{\Theta}{\Theta} \left( n + k \right) \left( \log \frac{m}{k} \right) \Theta
\]

External memory

\[
\left( N + \frac{Y}{\log Y} \right) \Theta
\]

Three-way quick sort [BS97].

Internal memory

Sorting \( k \) strings
Note: \( n_1 > n_2 \) and \( n_2 \geq N_2 \).

\[
\frac{B}{N_2} = \frac{\sum u_n}{\text{total \# of characters in long strings}}, \quad \frac{B}{N_1} = \frac{\sum u_1}{\text{total \# of characters in short strings}}.
\]

\[
(\Theta(n_1 \log n_1))^{\min(\log n_1, \log n_2)} = (\Theta(n_2 \log n_2))^{\min(\log n_1, \log n_2)}.
\]

**Model A**

String is indistinguishable

**Model B**

String is indistinguishable

External Sorting of \( K \) Strings

Center for Geometric Computing
Long strings: Look at first B characters.

- In model B use "permutation count pattern argument".
  - Use count pattern argument [AV88].

\[ \frac{N}{N} \]

Consider "equal length case" (all strings of length N). (1)

Short strings:

Lower Bounds:

- Model B: Using buffer-free technique on SB-tree [FC95].

Upper Bounds:

- Model A: Merge sort with "lazy" tree in internal memory.

External Sorting of K Strings
Dynamic memory model considered in [ZI99] is not general.

Merge sort based on [PCGI99] is general but nonoptimal.

more than a constant number of times during $\alpha$.

no other algorithm $\mathcal{A}$ can solve problem $\mathcal{P}$ is dynamically optimal for $\mathcal{P}$

Suppose that $\mathcal{A}$ solves problem $\mathcal{P}$ during $\alpha$.

\[ \alpha = m_1, m_2, \ldots \]

Algorithm chooses allocation sequence $\alpha = m_1, m_2, \ldots$ of allocation phases.

EM algorithm is allocated memory in an

ith phase: Algorithm owns $m_i$ blocks of memory for $2m_i$ I/O's.

Dynamic Memory Allocation Model

Barve & Vitter 98
\[(n \log n)^2 = \log \log n \quad \iff \quad iN < \left(\frac{B}{W}\right)^{B/N}(iB)\]

otherwise,

\[
\begin{align*}
\left(\frac{B}{W}\right) 
\times (iB)
\end{align*}
\]

\(=\) to comparisons per I/O possible outcomes

\# Internal memory

\(\text{Traditional I/O lower bound for sorting}\)
\[
\left(\frac{n}{\log n}\right) \leq \sum_{i=1}^{\frac{\log n}{2m}} \left( \left( \frac{B}{W} \right)^{\frac{i}{m}} \right) = \binom{\frac{n}{\log n}}{1}
\]

otherwise:

\[
\left( \left( \frac{B}{W} \right)^{\frac{i}{m}} \right) \times iB
\]

\[
O/I \to \text{I/O comparisons per I/O outcome}
\]

\[
B \frac{B}{W} = \frac{m}{w}
\]

\[
B\_{\text{blocks}} \frac{B}{W} = \frac{m}{w}
\]

Internal memory

ith phase:
execute a $\text{m-way merge operation}$ from $[\text{m}, \text{m}]$.

If the allocation size $m'$ is in $[\text{m}, \text{m}]$, we

in terms of $\text{m-way merge operations}$.

$[\text{m}, \text{m}]$ is an organization of an $\text{m-way merge}$

Data structure $[\text{m}, \text{m}]$.

$\text{m-way merge}$

$m$-way merge $U(m)$
Allocation size $|\text{VM}|$, m’s active merge

$\Omega \left( \frac{m}{w^m} \right) \text{ is the active merge.}$

$\Omega \left( \frac{m}{w^m} \right) \text{ is the active merge.}$

$\Omega \left( \frac{m}{w^m} \right) \text{ is the active merge.}$

$\Omega \left( \frac{m}{w^m} \right) \text{ is the active merge.}$
In response, transform the active merge $n_2$ into $[\frac{w}{n}, \frac{1}{4} w]$.

Data structure for allocation sizes $[\frac{w}{n}, \frac{1}{4} w]$.

Allocate size drops from $\lceil mw \rceil$ to $\lceil \frac{w}{4} \rceil$.
Conclusions and Open Problems

Dynamic memory allocation.

String processing.

Fundamental structures: contour line structure, point location.

Support of indexing/data structures (e.g., implementation of
(e.g., practical red-blue line segment intersection, spatial join).

GIS applications

(e.g., dynamic point location, range-searching).

Online dynamic data structures

(e.g., topological sorting, BFS, DFS, connectivity).

Fundamental graph problems

Large graphs (Don’t use square root trick).

Handling many disks, large merge orders, many partitioning elements,

TPFE, see http://www.cs.cmu.edu/~TPFE/

... New models, clusters of workstations, memory hierarchies,

Indistinguishability assumption.