(Some) Techniques for Handling Massive Data Sets

Jan Vahrenhold
Faculty of Computer Science
Technische Universität Dortmund, Germany

Several of the results discussed have been obtained jointly with:
Lars Arge, Joachim Gudmundsson, Thomas Hazel,
Klaus Hinrichs, Laura Toma, and Rajiv Wickremesinghe

Overview

Disclaimer:
- The opinions expressed in this talk are those of the speaker ;-) 
- The following things will be left out on purpose:
  - Detailed descriptions.
  - Proofs.
  - Several other techniques, e.g. parallel, distributed, or quantum computing.
- For the audience’s speaker’s convenience, all examples will be taken from one application area.

Acquiring Massive Data Sets

LIDAR: LIght Detection And Ranging.
- Data acquisition by low-altitude aircraft.
- Resolution: approx. 30 cm²/pixel.

Comparison with "conventional" remote sensing:

Resolution: 900 m²/pixel
Continental U.S.: 56 GB

Resolution: 30 cm²/pixel
Continental U.S.: 560,000 GB = 16,8 · 10⁶ GB

Processing Massive Data Sets

Obvious suggestions:
- “Buy more RAM”.
- “Use a super-computer.”

Obvious questions:
- “Who is going to sign the check?”
  - GISs used by, e.g., public land administration offices...
  - Quest for “poor man’s algorithms” (J. Abello).
- “What can be done today?”
  - “I would rather have
    today’s algorithms on yesterday’s computers
    than vice versa”
  - Efficient algorithms for massive data sets!

The “I/O-Modell” [Aggarwal & Vitter, 1988]

- Expensive input/output operations (I/Os).
- Complexity: Number of I/Os, disk space.
  \[ N = \frac{M}{\Theta} \text{ Objects in problem instance.} \]
  \[ M = \frac{\Theta}{\frac{N}{B}} \text{ Objects that fit into main memory.} \]
  \[ B = \frac{\Theta}{\frac{N}{M}} \text{ Objects that fit into a disk block.} \]
  \[ N = \frac{\Theta}{\log_B N} \text{ Parallel disks, P parallel CPUs}. \]

Fundamental complexities:
- Scanning \( N \) objects: \( \Theta(N/B) \text{ I/Os ("linear")}. \)
- Searching among \( N \) objects: \( \Theta(\log_B N) \text{ I/Os}. \)
- Sorting \( N \) objects: \( \Theta(N/B \cdot \log_M N/M) \text{ I/Os}. \)
  \[ \frac{B}{\Theta} \approx 10^4 \Rightarrow \frac{N}{B} < \frac{\log_B N}{\Theta} \approx N \log_B N < N \cdot \log_B N \]
Example: External Merge-Sort for a Set $S$

Approach:
- If $|S| \leq M$:
  - Sort $S$ in main memory.
- Else:
  - Subdivide $S$ into $\Theta(M/B)$ subsets of roughly equal size.
  - Sort each subset recursively.
  - Merge sorted subsets.

Analysis:
- Depth of recursion: $\Theta(\log_{M/B} N/M)$.
- Merging: $\Theta(N/B)$ I/Os/level.

I/O-complexity of sorting: $\Theta( N/B \cdot \log_{M/B} N/M )$ I/Os.

Aside:
- There is a (mathematically) much nicer model; see Mark’s talk.

A Note on the Model

Advantage:
- Abstract away hardware parameters; model first-order effects.
- Applicability to multi-level hierarchical memory.

Implications of the I/O-Model

A lot of rules tools change:
- One I/O transfers $B$ items—whether you like it or not.

| |
| Sorting $N$ items | $O(N \log_{B} N)$ | $O(\frac{1}{B} \log_{M/B} N)$ |
| Dictionary access | $O(\log_{B} N)$ | $O(\frac{1}{B} \log_{M/B} N)$ |
| Priority queue access | $O(\log_{B} N)$ | $O(\frac{1}{B} \log_{M/B} N)$ |
| Scanning an array | $O(N)$ | $O(\frac{1}{B} \log_{M/B} N)$ |
| List ranking* | $O(N)$ | $O(\frac{1}{B} \log_{M/B} N)$ |

*: Traversing a list from head to tail.

Aside:
- Round up the usual suspects.
  - Captain Renault, Casablanca.

Tools: Buffer Trees–I

Goal: Amortized cost of $O(\frac{1}{B} \log_{M/B} N)$ I/Os per operation.

Implies $O(\frac{1}{B} \log_{M/B} N)$ height”-constraint.

Play it again, Sam:
- $B$-tree: Group $\Theta(B)$ edges to get $O(\log_{B} N)$ height.
- Now: Group $\Theta(M/B)$ edges to get $O(\log_{M/B} N)$ height.

Tools: Buffer Trees–II

Problem:
- Traversal cost: “$\frac{1}{B}$(Nodes) I/Os per node”

\[
\begin{array}{c|c|c}
\text{Node size} & \text{Tree height} & \text{Traversal cost} \\
\Theta(B) & \Theta(\frac{1}{B} \log_{B} N) & \Theta(\frac{1}{B} \log_{B} N) \\
\Theta(M/B) & \Theta(\frac{1}{B} \log_{M/B} N) & \Theta(\frac{1}{B} \log_{M/B} N) \\
\end{array}
\]
Tools: Buffer Trees–III

In a nutshell:
- Emulate mergesort’s data flow upside down.

A closer look:
- One block reserved per leaf.
  - Collects (“blocks”) outgoing elements.
  - Buffer attached to root of subtree.
  - Collects (“blocks”) incoming elements.

Using buffers [Arge, 2003]:
- Collect $\Theta(M)$ elements in main memory.
- Processing $\Theta(M)$ elements:
  - Load subtree $(\Theta(M/B))$ “leaves” with an $\Theta(B)$-sized buffer from disk.
  - Route elements to buffer(s).
  - Move output to input buffers of subtrees rooted at the “leaves”.
- Recurse, i.e. use buffers on every level.

Tools: Buffer Trees–IV

Smallest-priority elements on leftmost spine of the tree.
- Maintain out-buffer for $\Theta(M)$ smallest elements.
  - Keep in memory, refill “occasionally” by deleting $\Theta(M)$ smallest elements.

A whole lot of hand-waving: Reorganization does not hurt!
- All the details (and more): [Arge, 2003].

Amortized(!) cost: $\Theta\left(\frac{1}{B} \log\frac{M}{B}\right)$ I/Os per insert/delete.
- [Arge, 2003]: Inserts/deletes do not interfere with each other!

Tools: External Priority Queues

Implications of the I/O-Model

A lot of rules change:
- One I/O transfers $B$ items—whether you like it or not.

Task:
- For each $v \in V$, compute the number of predecessors in $L$.
- Benefit: Can traverse sorted list in $\Omega(\text{sort}(|V|))$ I/Os.

Theorem 2.1 ([Chiang et al., 1995])
List-ranking has a lower bound of $\Omega(\text{sort}(|V|))$ I/Os.

Tools: List-Ranking–I

Define shorthand: $O(\text{sort}(X)) = O\left(\frac{N}{B} \log\frac{M}{B}\right)$.
- List $L$ with nodes $V$, stored in an array.

- $*$: Amortized cost.
- Worst-case: $O\left(\frac{N}{\log B}\right)$

Tools: List-Ranking–II

Goal: Find a “large” independent subset of nodes.

Alternatingly color “forward” lists using “red” and “blue”.
  - Scan all “forward” sublists simultaneously.
  - Use priority queue to send “forbidden” color “forward in time”.
- Alternatingly color “backward” lists using “green” and “blue”.
  - Choose the $\geq |V|/3$ nodes with most frequent color.
  - Remove (“bridge out”) and recurse; re-insert later.

Dominating cost (also for recursion):
- $\Theta(|V|)$ priority queue operations: $O(\text{sort}(|V|))$ I/Os (worst-case).

Implications of the I/O-Model

A lot of rules change:
- One I/O transfers $B$ items—whether you like it or not.
Definition 3.3 ([Callahan & Kosaraju, 1995])

Let $S$ be a set of points in $\mathbb{R}^d$, $d \leq 3$, and let $s > 0$. A well-separated pair decomposition (WSPD) for $S$ with respect to $s$ is a sequence $\{A_i, B_i\}_{i=1}^m$, of pairs of non-empty subsets of $S$, such that

1. $A_i \cap B_i = \emptyset$, $i = 1, \ldots, m$,
2. $A_i$ and $B_i$ are well-separated with respect to $s$, i.e. for all $i = 1, \ldots, m$:

   $s \cdot \|x-y\| \leq \|A_i - B_i\| \leq s \cdot \|x-y\|$ for all $x \in A_i$ and $y \in B_i$.
3. For any two points $p \neq q$ in $S$, there exists exactly one pair $\{A_i, B_i\}$ such that:
   (i) $p \in A_i$ and $q \in B_i$, or
   (ii) $q \in A_i$ and $p \in B_i$.

One can represent a WSPD as a split tree in $O(|S| \log |S|)$ time.

Pruning Dense Spanners

Theorem 3.4 ([Gudmundsson et al., 2002])

Given: $t$-spanner $G = (S, E)$ (for a constant $t > 1$) and constant $\epsilon > 0$. Can compute in internal memory in time $O(|E| \log_2 |S|)$:

A $(1+\epsilon)$-spanner $G'$ of $G$ with $O(|S|)$ edges.

Algorithm:

1. Compute a WSPD for $S$ with $O(|S|)$ pairs; separation ratio:

   $s = 4(1 + (1 + \epsilon)!) \epsilon$

2. Consider each pair $\{A_i, B_i\}$ and keep at most one edge $(x_i, y_i)$ of $E$ with $x_i \in A_i$ and $y_i \in B_i$, provided such an edge exists.
3. Do the math: Resulting graph is a $(1+\epsilon)$-spanner of $G$.
4. Tool: Query split tree in $O(\log_2 |S|)$ time per query.
Anatomy of a Server:
- Some CPUs.
- A lot of RAM.
- A whole lot of disk space.
  - Redundant array of inexpensive disks (RAID).
  - "Glue together" several disks; redundancy, if needed.

Practical considerations:
- Disks (and OSs) optimize sequential scans.
- A lot of attention has been (successfully!) paid to optimize large-scale sorting.
  - PennySort- and MinuteSort-Competitions; lots of (theoretical and practical) results...

Solving a problem by sorting and scanning should be effective!
A Sketch of the Algorithm

- Compute the grid’s shape: Sequential scan.
- Store “cells” in row-major order.
- Assign segment to cell(s): Sequential scan.
- Group segments by cells: Sort.

Single-shot point location:

- Compute (number of) cell containing q.
- Load segments: Sequential scan of “a few” disk blocks.

Multiple (“batched”) point location:

- Compute (number of) cells containing points: Sequential scan.
- Group points by cells: Sort.
- Load segments and points: Synchronized sequential scans.

Use Dijkstra’s algorithm.

Complexity:

- Scan: For each cell: cost of “least-cost path” to “nearest” source.
- Cost: sources
- Elevation model
- Discrete cost function
- Set
- Sources

Output: “Least-cost path surface”

A Closer Look. . .

Related concept: Voronoi diagram

- Partitioning of the plane into equivalence classes.
- Main difference: cost given by a continuous(!) metric(!!).

Cost given by a discrete cost function.

“Point-to-point”-queries, high preprocessing complexity.

For a single(!) source: \( \Omega \left( \frac{M}{M/B} \log_{M/B} \left( \frac{|N|}{M} \right) \right) \) I/Os.

Experimental Evaluation

- External algorithm performs at least one I/O per single query.
- Compare against fastest internal algorithm [Edahiro et al., 1984].
  - Time- and space-optimal; hand-tuned to avoid cache-faults.
- Can internal algorithm amortize potential page faults?

Cache/page/. . . faults on all levels of the memory hierarchy!

Levels of the memory hierarchy

Overview

Part V:

Algorithm Engineering:
Computing Least-Cost Surfaces

There ain’t nothing so complicated as the inside of a torpedo

Algorithms for Digital Elevation Models

TerraCost [Hazel et al., 2008]:

- Input:
  - Digital (raster) elevation model \( T \).
  - Discrete cost function \( \text{cost} : T \to \mathbb{R} \).
  - Set \( S \) of “sources”.
- Output: “Least-cost path surface”
  - For each cell: cost of “least-cost path” to “nearest” source.

Example: Spreading of wildfires

Sources: Set \( S \) of “focal” points.

Shortest paths on a cost grid

Input data:

- Digital elevation model, given as grid with \( \sqrt{N} \times \sqrt{N} \) cells.
- Store cost for traversing each cell.

Cost function:

- Moving from \( c_1 \) to adjacent cell \( c_2 \):
  \[
  \text{cost}(c_1, c_2) = \text{cost}(c_1) + \text{cost}(c_2) - \frac{1}{2} \text{cost}(c_1 \rightarrow c_2)
  \]
  \[
  \text{cost}(c_1 \rightarrow c_2) = \frac{1}{\sqrt{2}} \left( c_1 c_2 \right)_{\text{axis-parallel}}
  \]
  \[
  \text{cost}(c_1 \rightarrow c_2) = \frac{1}{\sqrt{2}} (c_1 c_2)_{\text{diagonal}}
  \]
- Find cost-minimal path.

Overview

Single-source algorithm [Arge et al., 2001]:

1. Partition grid into tiles of size \( R \).
2. Shortest paths “source—boundary”/tile.
5. Shortest paths “boundary—cells”/tile.
Choose \( R \in O(M) \Rightarrow O(\text{sort}(N)) \) I/Os.

[Hazel et al., 2008]: Multiple sources & Algorithm Engineering.
Applications

Where's the data?
National Center for Ecological Analysis and Synthesis (USA):
Simulate water-borne agents in a marine environment at a global scale.

Experimenting with massive data sets:
- It’s not a short-time diversion.
  - Experiments are likely to take more than a day per run.
- It’s very rewarding (if it works).
- High motivation for students:
  - Contribute to open source project.

Algorithm Engineering–I

Engineering: Choice of tile size $B \leq R \leq M$
- Number of boundary nodes: $O(\sqrt{R})$.
- Internal-memory (Steps 2, 3, and 5): $O(\sqrt{R}\log_2 R)$ time.
- Number of edges: $O(N)$ (stored in a blocked layout).
- External-memory (Step 4): $O(\sqrt{R} + N/B)$ I/Os.
- Global extra cost: $O(sort(N))$ I/Os.

Algorithm Engineering–II

Observation:
- Trade-off between internal- and external-memory cost:
  $O(\sqrt{R}\log_2 R)$ time $\rightarrow O(\sqrt{R} + N/B)$ I/Os.

Algorithm Engineering–III

Surprising(?) fact:
- Theory: Best performance, if $R \in \Theta(M)$.
- Practice: Best performance, if $R \approx$ size of L2-cache.
- Interpretation: Skip one level in the memory hierarchy.

Algorithm Engineering–IV

Efficient internal data structures:
- Array: Row-based versus column-based access (cache-lines).
- Priority queue: In-place binary heaps.
- Using large tiles:
  - Internal-memory shortest path computations dominate running time.
  - External-memory shortest path computations relatively fast.

Algorithm Engineering–V

More observations:
- Using large tiles:
  - Internal-memory shortest path computations dominate running time.
  - External-memory shortest path computations relatively fast.
- Internal-memory computations only local to tile.
- Using cluster-connected resources: almost linear speed-up.

Experimental evaluation

- Comparison with GRASS module r.cost (open source GIS).
- Comparison with “hand-tuned” version of Dijkstra.

Smaller memory simulates larger data set:
- Our largest data set: 32 hrs. running time, $256$ GB temp. data.
- Near-linear scaling...also beyond the “main memory barrier”.

Overview

Part VI:

Conclusions

“The best goodbyes are short. Adieu.”
Kasper Gutman, The Maltese Falcon.
Techniques for handling massive data sets:
- Know the model and its limitations.
- Use efficient "black boxes" (there are lots of them).
- "Keep it simple, stupid": Use sorting and scanning.
- Think "Algorithm Engineering!"


Further reading:
- [Vitter, 2001]: ACM Comp. Surv., external algorithms.
- [Arge, 2002]: Handbook chapter, external data structures.
- [Meyer et al., 2003]: Dagstuhl volume "Alg. for Memory Hierarchies".

Further listening (and open problems):
- The two talks coming up after the coffee break.

Further reading:
- [Arge, 2002]: Handbook chapter, external data structures.

Techniques for handling massive data sets:

---

Bibliography


---

Summary

Techniques for handling massive data sets:

- Know the model and its limitations.
- Use efficient "black boxes".
- "Keep it simple, stupid": Use sorting and scanning.
- Think "Algorithm Engineering!"


Further reading:
- [Vitter, 2001]: ACM Comp. Surv., external algorithms.
- [Arge, 2002]: Handbook chapter, external data structures.
- [Meyer et al., 2003]: Dagstuhl volume "Alg. for Memory Hierarchies".

Further listening (and open problems):
- The two talks coming up after the coffee break.

---

Techniques for handling massive data sets:

---

Bibliography


---

Summary

Techniques for handling massive data sets:

- Know the model and its limitations.
- Use efficient "black boxes".
- "Keep it simple, stupid": Use sorting and scanning.
- Think "Algorithm Engineering!"


Further reading:
- [Vitter, 2001]: ACM Comp. Surv., external algorithms.
- [Arge, 2002]: Handbook chapter, external data structures.
- [Meyer et al., 2003]: Dagstuhl volume "Alg. for Memory Hierarchies".

Further listening (and open problems):
- The two talks coming up after the coffee break.

---

Techniques for handling massive data sets:

---

Bibliography


---

Summary

Techniques for handling massive data sets:

- Know the model and its limitations.
- Use efficient "black boxes".
- "Keep it simple, stupid": Use sorting and scanning.
- Think "Algorithm Engineering!"


Further reading:
- [Vitter, 2001]: ACM Comp. Surv., external algorithms.
- [Arge, 2002]: Handbook chapter, external data structures.
- [Meyer et al., 2003]: Dagstuhl volume "Alg. for Memory Hierarchies".

Further listening (and open problems):
- The two talks coming up after the coffee break.

---

Techniques for handling massive data sets:

- Know the model and its limitations.
- Use efficient "black boxes".
- "Keep it simple, stupid": Use sorting and scanning.
- Think "Algorithm Engineering!"


Further reading:
- [Vitter, 2001]: ACM Comp. Surv., external algorithms.
- [Arge, 2002]: Handbook chapter, external data structures.
- [Meyer et al., 2003]: Dagstuhl volume "Alg. for Memory Hierarchies".

Further listening (and open problems):
- The two talks coming up after the coffee break.