Processing Huge Graphs with STXXL

joint work with


Roman Dementiev
Algorithmik II, Fakultät für Informatik, University of Karlsruhe
dementiev@ira.uka.de
Where do we have HUGE graph instances?

Challenges:

■ Network Analysis
  ▶ **World Wide Web**: clusters, cliques, transitivity coefficient
  ▶ Phone calls: terrorist networks (**CIA**)!
  ▶ . . .

■ Route planning with **PDAs**

■ Single agent search, action planning, state exploration in **Artificial Intelligence**, BDD reduction

■ Approximation and heuristic algorithms for very large optimization problems

■ . . .
The Problem: RAM Model vs. Real Computer

- Where do we have HUGE graph instances?
- The Problem: RAM Model vs. Real Computer
- I/O Model

Motivation

Input Size vs. Time

- Real performance
- Predicted performance

University of Karlsruhe, Roman Dementiev
I/O Model

- Aggarwal–Vitter I/O model
  - $N$ — size of input
  - $M$ — size of main memory ($M \ll N$)
  - $B$ — size of transfer block (128KB .. 2 MB)
  - Cost measure – number of I/Os

- Add disks to improve bandwidth:
  Parallel Disk Model (PDM) by Vitter–Shriver
  - $D$ independent disks
  - In an I/O step try transfer $D$ blocks between main memory and disks
  - Goal: minimize number of I/O steps
I/O-Efficient Tools

- **Scanning**: $\operatorname{scan}(N) = O\left(\frac{N}{DB}\right)$ I/Os.

- **Sorting**: $\operatorname{sort}(N) = O\left(\frac{N}{DB \lceil \log_{M/B} \frac{N}{M} \rceil}\right), \approx \frac{2N}{DB}$.

- **Special I/O-efficient data structures**:
  - priority queues – $O\left(\frac{1}{N} \operatorname{sort}(N)\right)$ per operation
  - stacks, FIFO queues – $O\left(\frac{1}{DB}\right)$ per operation
  - search trees – $O\left(\log_{M/B} \frac{N}{M}\right)$ per operation
  - ...
**STXXL**: a Collection of I/O-Efficient Tools

- STL – C++ Standard Template Library, implements basic containers (maps, sets, priority queues, etc.) and algorithms (quicksort, mergesort, selection, etc.)

- **STXXL**: Standard Template Library for XXL Data Sets


  containers and algorithms that can process huge volumes of data that only fit on disks (I/O-efficient tools)
  - Compatible with STL
  - **Performance**–oriented
**STXXL Features**

- **Parallel** disk support
- Handles very large problems (up to terabytes)
- **Pipelining** saves many I/Os
  - Feed output of I/O-efficient alg. to input of another I/O-efficient alg. directly
- Explicitly **overlaps** I/O and computation
- Avoids superfluous **copying**
  - in OS I/O subsystem and the library itself
- **Compatible with STL** – C++ Standard Template Library
  - Short development times
  - **Reuse** of STL code (e.g. STL selection algorithm)
Applications: state exploration, shortest paths, crawling WWW, ...
BFS: RAM algorithm

Internal memory algorithm:

Q: FIFO queue of nodes
Q.push(s)
while Q.notEmpty()
    u := Q.pop();
    visit u
    foreach unmarked neighbor v
        mark v
    Q.push(v)
BFS: RAM algorithm

Internal memory algorithm:
Q: FIFO queue of nodes
Q.push(s)
while Q.notEmpty()
  u := Q.pop();
  visit u
  foreach unmarked neighbor v
    mark v
    Q.push(v)

- Marking nodes: $O(m)$ I/Os
- Finding neighbors (adj. lists): $O(n)$ I/Os
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. $N(L(t - 1)) = \text{all neighbours of } L(t - 1)$ \quad $\mathcal{O}(|L(t - 1)| + \frac{|N(L(t - 1))|}{D \cdot B})$ I/Os.
2. eliminate duplicates in $N(L(t - 1))$ by sorting \quad $\mathcal{O}(\text{sort}(|N(L(t - 1))|))$ I/Os.
3. eliminate nodes already in $L(t - 1)$ by scanning \quad $\mathcal{O}(\text{scan}(|L(t - 1)|))$ I/Os.
4. eliminate nodes already in $L(t - 2)$ by scanning \quad $\mathcal{O}(\text{scan}(|L(t - 2)|))$ I/Os.

\[
\sum_{i} |N(L(i))| \leq 2 \cdot m \quad \text{and} \quad \sum_{i} |L(i)| \leq n \quad \Rightarrow \quad \mathcal{O}(n + \text{sort}(n + m)) \text{ I/Os in total.}
\]
Algorithm of Mehlhorn and Meyer

**Preprocessing:** $O(sort(n + m))$ I/Os
- partition nodes into $O(n/\mu)$ subsets (clusters) s.t. any two nodes in same cluster have distance at most $\mu$ in $G$.
- store adjacency lists of nodes in the same cluster consecutively

**BFS Phase: Refined Algorithm of Munagala-Ranade**
- extract neighbors of $L(t)$ by scanning sorted external data structure $H$ (hot pool) – prevents the $O(n)$ accesses.
- if first node in a cluster is reached, add all adjacency lists of the cluster to $H$.
- each adjacency list stays in $H$ for at most $\mu$ iterations.
- $O(n/\mu + \mu \cdot scan(n + m) + sort(n + m))$ I/Os.
- Balancing: $O\left(\sqrt{nm/B} + sort(n + m)\right)$ I/Os.
BFS Experiments: Random Graphs $m = 4n$

- **Pipelined** STXXL implementations of MR-BFS and MM-BFS
- Linux, 2GHz Xeon, 1 GB RAM, Seagate Barracuda hard disks (65 MB/s max, 9 ms seek time)

![Graph showing time vs. n for different BFS algorithms](image)
BFS Experiments

MM-BFS versus MR-BFS

- Random graphs: MR-BFS 3.8 times faster — small diameter
- Grid graphs: MM-BFS 87 times faster — large diameter
BFS Experiments

MM-BFS versus MR-BFS
- Random graphs: MR-BFS 3.8 times faster — small diameter
- Grid graphs: MM-BFS 87 times faster — large diameter

Parallel disks ($D = 4$):
- Speedup is about two
- Become more CPU bound: may benefit from parallel processing in STXXL sorting
BFS Experiments

MM-BFS versus MR-BFS
- Random graphs: MR-BFS 3.8 times faster — small diameter
- Grid graphs: MM-BFS 87 times faster — large diameter

Parallel disks ($D = 4$):
- Speedup is about two
- Become more CPU bound: may benefit from parallel processing in STXXL sorting

Real graph instance:
Web crawl with $n=130$ million, $m=1.4$ billion
- Core: 10–12 levels, the rest: thousands of levels with 2-3 nodes per level
- MR-BFS (2.3 hours) beats MM-BFS (4.5 hours)
- External BFS computation is now feasible!
Minimum Spanning Trees and Forests

- Applications: network design, clustering, subroutine in many algorithms for optimization problems, approximation of TSP
- Challenge: Can we compute MSTs for really huge graphs?
Semiexternal MST

Kruskal’s Algorithm [1956]

Idea: grow a forest

\[
T := \emptyset \quad / \quad \text{subforest of the MST}
\]

\[
\text{foreach } (u, v) \in E \text{ in ascending order of weight do}
\]

\[
\text{if } u \text{ and } v \text{ are in different subtrees of } T \text{ then}
\]

\[
T := T \cup \{(u, v)\} \quad / \quad \text{Join two subtrees}
\]

\[
\text{return } T
\]

Problem: Random accesses to a union-find data structure

Semiexternal MST [Abello Buchsbaum Westbrook 02]

Assume \( n \leq M - 2B \):

run Kruskal’s algorithm using I/O-efficient \texttt{stxxl::sort}
**Fast Node Reduction** \( n \rightarrow n' \)

**Motivation**

**STXXL Library**

**BFS**

**MST**
- Minimum Spanning Trees and Forests
- Semiexternal MST
- Fast Node Reduction
- External Implementation: Sweeping
- Relinking Using Priority Queues
- Sweeping with linear internal work
- \( m \approx 2n \)
- MST Experiments

**Coloring**

---

**for** \( i := n \) **downto** \( n' + 1 \) **do**

- pick a random node \( v \)
- find the **lightest** edge \((u, v)\) out of \( v \) and output it
- **contract** \((u, v)\)

---

**#edges processed:** \( 2m \ln \frac{n}{n'} \)
\( \pi : \) random permutation \( V \rightarrow V \)

sort edges \((u, v)\) by \( \min(\pi(u), \pi(v)) \)

for \( i := n \) downto \( n' + 1 \) do

pick the node \( v \) with \( \pi(v) = i \)

find the lightest edge \((u, v)\) out of \( v \) and output it

contract \((u, v)\)

Problem: how to implement relinking?
Relinking Using Priority Queues

Q: STXXL priority queue  // Order: max node, then min edge weight

`foreach (\{u, v\}, c) \in E do Q.insert((\{\pi(u), \pi(v)\}, c, \{u, v\}))`

current := \(n + 1\)

```
loop
    (\{u, v\}, c, \{u_0, v_0\}) := Q.deleteMin()
    if current \neq \max \{u, v\} then
        if current = M + 1 then return
        output \{u_0, v_0\}, c
        current := \max \{u, v\}
        connect := \min \{u, v\}
    else Q.insert((\{\min \{u, v\}, connect\}, c, \{u_0, v_0\}))
```

\[\approx sort(10m \ln \frac{n}{M})\] I/Os with STXXL priority queues  [Sanders 00]

Problem: Superfluous internal work overhead
⇒ for parallel disks CPU bound!
Sweeping with linear internal work

- Assume $m = \mathcal{O}(M^2/B)$
- $k = \Theta(M/B)$ external buckets with $n/k$ nodes each
- $M$ nodes for last “semie external” bucket

Assume $m = \mathcal{O}(M^2/B)$

$k = \Theta(M/B)$ external buckets with $n/k$ nodes each

$M$ nodes for last “semie external” bucket
$m \approx 2n$

Motivation

STXXL Library

BFS

MST

- Minimum Spanning Trees and Forests
- Semiexternal MST
- Fast Node Reduction
- $n \rightarrow n'$
- External Implementation:
  - Sweeping
- Relinking Using Priority Queues
- Sweeping with linear internal work

$m \approx 2n$

MST Experiments

Coloring

University of Karlsruhe, Roman Dementiev
MST Experiments

Huge inputs:
- grid graph with $n = 2^{32}$ (input = 96 GBytes) $\Rightarrow$ only 8h 40min

Random graphs with $n = 320 \cdot 10^6$ and $m = 640 \cdot 10^6$ (time per edge):

<table>
<thead>
<tr>
<th></th>
<th>1 disk</th>
<th>4 disks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucket implementation</td>
<td>6.7 $\mu$s</td>
<td>4.3 $\mu$s</td>
</tr>
<tr>
<td>priority queue implementation</td>
<td>11.0 $\mu$s</td>
<td>8.9 $\mu$s</td>
</tr>
</tbody>
</table>

- Bucket version: going from one to four disks brings speedup 1.56 ($\Rightarrow$ not I/O bound)
- PQ version is CPU-bound but performs reasonably well
  - handles also graphs with high degree ($> M$) nodes

$\Rightarrow$ External MST computation is feasible!!
Graph Coloring

- Applications: solving sparse linear systems of equations, resource allocation, scheduling, the construction and testing of VLSI circuits
- NP-complete problem
- Heuristics do not guarantee the quality, but very fast
- I/O-efficient coloring heuristics?
Greedy Coloring [Zeh]

Algorithm:
- Process nodes in arbitrary order
- When a node $v$ is visited, it is assigned the smallest color, which has not been assigned to any of already visited neighbors of $v$.

I/O-Efficient priority queue needed ($O(\text{sort}(m))$ I/Os):

Function $\text{GreedyColoring}(E)$

```plaintext
ExternalPriorityQueue: $Q$         // stores neighbor colors

while $E \neq \emptyset \lor Q \neq \emptyset$ do
    $v := \min\{E.\text{front}().\text{src}, Q.\text{min}().\text{node}\}$
    $U := \emptyset$      // used colors
    while $v = Q.\text{min}().\text{node}$ do
        $U.\text{append}(Q.\text{delMin}().\text{color})$ od
    sort $U$
    scan $U$ and assign node $v$ color $C := \min_{c \in U} \{c\}$
    while $v = E.\text{front}().\text{src}$ do
        $Q.\text{insert}((E.\text{popFront}().\text{dst}, C))$ od
```

(I)
Highest Degree First (HDF)

Visit and color the nodes in the order of their degree (difficult nodes first):

1. Rename nodes in $E$ such that $\text{deg}(v) > \text{deg}(u) \Rightarrow \text{newName}(v) < \text{newName}(u)$.
2. Call $\text{GreedyColoring}(E)$ computing a coloring.
3. Restore the old names in the coloring.

- Steps (1) and (3) can be implemented using sorting.
- **Non-pipelined** implementation max. I/O volume: $224m + 120n$ bytes
- **STXXL pipelined** implementation transfers at most: $112m + 72n$ bytes
Smallest Degree Last (SDL)

- (Phase 1) recursively remove a node with the smallest degree from the graph.
- (Phase 2) greedily color the nodes in the \textit{backward} removal order.

Random accesses to the adjacency lists during node removal \(\Rightarrow \Omega(m)\) I/Os.
Batched Smallest Degree Last (BSDL)

- Remove many nodes (batch) in each recursion, each recursion costs $O(\text{sort}(m))$ I/Os
- Take nodes with degrees at most $\delta(V, E)$, where $\delta$ maps an integer in range $[\min_{v \in V} \{\text{deg}(v)\}, \infty)$
- Possible maximum batch degree ($\delta$) functions:
  - average degree $\delta(V, E) = \lceil 2m/n \rceil$
  - median degree
  - node $c$-quantile $\delta(V, E) = \min \{ k : |V_{1..k}| \geq cn \}$
  - edge $c$-quantile
    $\delta(V, E) = \min \{ k : \sum_{v \in V_{1..k}} \text{deg}(v) \geq 2mc \}$
On planar graphs BSDL with $\delta(V, E) = \lceil 2m/n \rceil$ reduces a constant fraction of edges in each recursion, guarantees:
- Needs at most $\mathcal{O}(\text{sort}(m))$ I/Os
- Since $\lceil 2m/n \rceil \leq 6 \Rightarrow$ BSDL produces a 7-coloring

To reduce more nodes (faster) we can set $\delta(V, E) = 6$ and obtain the same guarantees (EM7 algorithm)
Experiments: Small Inputs

- Linux, Opteron 2GHz, 1-4 IDE Disks
- **SDL**: an implementation from the **BOOST** library
- Random and real-world graphs
- Difference in the number of colors: < 1.3%
Experiments: Large Inputs

- SDL can not run
- Random graphs: HDF is slightly faster
- Difference in the number of colors: < 1.3%

**Minutes/#colors**

<table>
<thead>
<tr>
<th>Instance</th>
<th>$n/10^6$</th>
<th>$m/10^6$</th>
<th>HDF</th>
<th>BSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand.planar1</td>
<td>8.4</td>
<td>16.8</td>
<td>0.55/6.0</td>
<td>0.57/5.1</td>
</tr>
<tr>
<td>rand.planar2</td>
<td>8.4</td>
<td>25.2</td>
<td>0.72/7.0</td>
<td>0.77/6.1</td>
</tr>
<tr>
<td>delaunay1</td>
<td>8.5</td>
<td>25.5</td>
<td>0.69/7</td>
<td>0.67/7</td>
</tr>
<tr>
<td>delaunay2</td>
<td>84.7</td>
<td>254.0</td>
<td>6.95/7</td>
<td>6.57/7</td>
</tr>
<tr>
<td>delaunay3</td>
<td>503.0</td>
<td>1509.0</td>
<td>49.87/7</td>
<td>43.19/7</td>
</tr>
<tr>
<td>webgraph</td>
<td>135.0</td>
<td>1079.9</td>
<td>35.86/246</td>
<td>43.57/246</td>
</tr>
</tbody>
</table>

- BSDL is slightly faster than HDF on delaunay graphs (they have many low-degree nodes)
- BSDL produces better colorings
## Experiments: EM7 on Huge Planar Graphs

<table>
<thead>
<tr>
<th>Instance</th>
<th>n/10^6</th>
<th>m/10^6</th>
<th>C</th>
<th>minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand.planar1</td>
<td>8.4</td>
<td>16.8</td>
<td>5.5</td>
<td>0.49</td>
</tr>
<tr>
<td>rand.planar2</td>
<td>8.4</td>
<td>25.2</td>
<td>6.05</td>
<td>0.71</td>
</tr>
<tr>
<td>delaunay1</td>
<td>8.5</td>
<td>25.5</td>
<td>7</td>
<td>0.62</td>
</tr>
<tr>
<td>delaunay2</td>
<td>84.7</td>
<td>254</td>
<td>7</td>
<td>5.80</td>
</tr>
<tr>
<td>delaunay3</td>
<td>503</td>
<td>1509</td>
<td>7</td>
<td>42.1</td>
</tr>
</tbody>
</table>

- **EM7 outperforms BSDL and HDF (can reduce more nodes)**

  ⇒ Huge real-world graphs can be colored in **an hour** on a PC!
Summary

STXXL graph implementations not covered here:

- Triangle counting (clustering coefficient computation)

Open problems:

- Shortest paths
- Graph diameter (BFS-related)
- Better I/O-efficient coloring heuristics