I/O-Efficient Algorithms and Data Structures

P. Sanders    R. Dementiev

Institute for Theoretical Computer Science, Algorithmics II
University of Karlsruhe, Germany

July 10, 2007
Large Data Sets

Sources of very large data volumes

- Data warehouses: enterprise data collections
- Geographic information systems: GoogleEarth, NASA’s World Wind
- Computer graphics: visualize huge scenes
- Billing systems: phone calls, traffic
- Analyze huge networks: Internet, phone call graph
- Text collections: Google, Yahoo!, Ask, etc.
Scalability of Algorithms

How to process them

- Buy a TByte main memory? ⇝ expensive or impossible
- Here: how to process very large data sets cost-efficiently
Scalability of Algorithms

How to process them

- Buy a TByte main memory? ⇝ expensive or impossible
- Here: how to process very large data sets cost-efficiently
von Neumann RAM Model

In first year course:
- Computer $\approx$ CPU + Memory
- Uniform cost model:
  - each access and each operation cost one unit of time

Impact:
- Very simple analysis
- Good estimation for first computers

BUT: Modern computers have a deep HIERARCHY of memory
Modern Computer

Increasing access time and space
Why Memory Hierarchies?

- **Why**: purely economic reasons !!!
- **faster → more expensive** → as few expensive pieces as possible.

- Hard disks can be the ultimate performance killers.

---

![Diagram of memory hierarchy]

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Relative to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0.5 ns</td>
<td>1</td>
</tr>
<tr>
<td>L1 cache</td>
<td>0.5 ns</td>
<td>1-2</td>
</tr>
<tr>
<td>L2 cache</td>
<td>3 ns</td>
<td>2-7</td>
</tr>
<tr>
<td>DRAM</td>
<td>150 ns</td>
<td>80-200</td>
</tr>
<tr>
<td>TLB</td>
<td>500+ ns</td>
<td>200-2000</td>
</tr>
<tr>
<td>Disk</td>
<td>10 ms</td>
<td>$10^7$</td>
</tr>
</tbody>
</table>
Trends

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Yearly Improvement Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Latency</td>
<td>10 %</td>
</tr>
<tr>
<td>Disk Bandwidth</td>
<td>20 %</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>55 %</td>
</tr>
<tr>
<td>RAM Bandwidth</td>
<td>40 %</td>
</tr>
<tr>
<td>RAM Capacity/Cost</td>
<td>45 %</td>
</tr>
</tbody>
</table>

- Performance gap is increasing.
- RAM Capacity doubling about every two years but users doubling data storage about every 5 months (frequently copying everything).
- Results in I/O Bottleneck.
Why are Hard Disks such slow?

Components of disk access time:

- Seek time (milliseconds, SLOW)
- Rotational latency (milliseconds, SLOW)
- Read/write access (nanoseconds, FAST): bandwidth 40–80 MByte/s

Reading many consecutive data items takes not much longer than reading a single data item

⇒ balance seek time/rot. latency with bandwidth
⇒ block size ≈ track size ≈ a few MBytes
How the Operation System tries to make up for it

Virtual Memory provides the look of the uniform model.

But not necessarily the performance !!!

Additionally:

- Disk partitioned into blocks of $\geq 512$ Bytes.
- Every disk access reads or writes a whole block.
- Read ahead.

This helps in special cases (e.g. scanning).

For most interesting algorithms this does not help at all.
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
- I/O-efficient alg. $\equiv$ External memory alg. $\equiv$ Secondary memory alg.
Main memory size $M \ll$ Problem size $N$

External memory = $D$ disks

Data is transferred in blocks of size $B$

Up to $\leq D \cdot B$ data per I/O step ($10^2$ per sec.)

Goal 1: Minimize number of I/O steps

Goal 2: Minimize number of CPU instructions

$\text{scan}(x) := \Theta\left(\frac{x}{D \cdot B}\right)$ I/Os.

$\text{sort}(x) := \Theta\left(\frac{x}{D \cdot B} \cdot \log_{M/B} \frac{x}{B}\right)$ I/Os.
How to make algorithms I/O-efficient?

Only a few golden rules:

- Avoid unstructured access patterns.
- Incorporate LOCALITY directly into the algorithm.

Tools:

- Scanning: $\text{scan}(N) = O\left(\frac{N}{DB}\right)$ I/Os.
- Sorting: $\text{sort}(N) = O\left(\frac{N}{DB}\left\lceil \log_{M/B} \frac{N}{M} \right\rceil\right)$, usually $\lceil \log_{M/B} \frac{N}{M} \rceil = 2$.
- Special I/O-efficient data structures.
- “Simulation” of parallel algorithms.
Warmup: Scanning

\[
\text{sum} = 0;
\text{for } i=1 \text{ to } N \text{ do sum := sum + A[i];}
\]

\[\text{scan}(N) = O(N/B) \text{ I/Os, optimal.}\]
**Sorting: THE tool for Reordering**

Importance of Sorting - An Example

```plaintext
int[1..N] A,B,C;
for i=1 to N do A[i]:=B[C[i]];
```

⇒ Worst case: \( \Omega(N) \) I/Os. \( N = 10^6, \ T = 10000 \ \text{sec} \approx 3 \ \text{hours} \)

Better:

```
SCAN C: (C[1]=17,1), (C[2]=5,2), ...
```

```
SORT(1st): (C[73]=1,73), (C[12]=2,12), ...
```

```
par SCAN : (B[1],73), (B[2],12), ...
```

```
SORT(2nd): (B[C[1]],1), (B[C[2]],2), ...
```

⇒ Worst case: sort(\(N\)) \(\approx O(N/DB)\) I/Os. \(B = 100\ \text{KBytes}, \ T < 1 \ \text{sec}\)
Sorting: THE tool for Reordering

Importance of Sorting - An Example

```plaintext
int[1..N] A,B,C;
for i=1 to N do A[i]:=B[C[i]];
⇒ Worst case: Ω(N) I/Os. N = 10^6, T = 10000 sec ≈ 3 hours

Better:

SCAN C: (C[1]=17,1), (C[2]=5,2), ...
SORT (1st): (C[73]=1,73), (C[12]=2,12), ...
par SCAN : (B[1],73), (B[2],12), ...
SORT (2nd): (B[C[1]],1), (B[C[2]],2), ...
⇒ Worst case: sort(N) ≈ O(N/DB) I/Os. B = 100KBytes, T < 1 sec.
```
Matrix Transposition

Problem:

$$C = A^T, \ C_{i,j} = A_{j,i}$$

Layout of matrices:

Row major  Column major  $4 \times 4$-blocked
Algorithm 1: Nested loops

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    C[j][i] = A[i][j];

- Row major
- Writing a column of $C \Rightarrow \Theta(N)$ I/Os
- Total $O(N^2)$ I/Os
Matrix Transposition: Algorithm 2

Algorithm 2: **Blocked** algorithm

Partition $A$ ($C$) into submatrices $A^{r,s}$ ($C^{r,s}$) of size $B \times B$, $B^2 = \Theta(M)$.

Transfer each submatrix $A^{r,s}$ to the internal memory $\Rightarrow B$ I/Os

Apply Algorithm 1 to $A^{r,s}$ (internally)

Transfer it to $C^{s,r} \Rightarrow B$ I/Os

\[
2 \frac{N^2}{B^2} \cdot B = O\left(\frac{N^2}{B}\right) \text{ I/Os, optimal.}
\]
Matrix Multiplication

Problem:

\[ Z = X \cdot Y, \quad z_{ij} = \sum_{k=1}^{N} x_{ik} \cdot y_{kj} \]
Matrix Multiplication

Algorithm 1: Nested loops

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

```plaintext
for $i = 1$ to $N$
  for $j = 1$ to $N$
    $z_{ij} = 0$
    for $k = 1$ to $N$
      $z_{ij} = z_{ij} + x_{ik} \cdot y_{kj}$
```

Algorithm 2: Blocked algorithm

- Partition $X$ and $Y$ into blocks of size $s \times s$, $s = \Theta(\sqrt{M})$.
- Apply Algorithm 1 to $N/s \times N/s$ matrices; elements are $s \times s$ sub-matrices.
- Use $s \times s$-blocked layout.

$O((N/s)^3 \cdot s^2/B) = O(N^3/(s \cdot B)) = O(N^3/(B \cdot \sqrt{M}))$ I/Os, optimal.
Matrix Multiplication

Algorithm 1: Nested loops

- Row major
- Reading a column of $Y$ ⇒ $N$ I/Os
- Total $O(N^3)$ I/Os

```
for $i = 1$ to $N$
  for $j = 1$ to $N$
    $z_{ij} = 0$
    for $k = 1$ to $N$
      $z_{ij} = z_{ij} + x_{ik} \cdot y_{kj}$
```

Algorithm 2: Blocked algorithm

- Partition $X$ and $Y$ into blocks of size $s \times s$, $s = \Theta(\sqrt{M})$.
- Apply Algorithm 1 to $N/s \times N/s$ matrices; elements are $s \times s$ sub-matrices.
- Use $s \times s$-blocked layout.

$\mathcal{O}((N/s)^3 \cdot s^2 / B) = \mathcal{O}(N^3 / (s \cdot B)) = \mathcal{O}(N^3 / (B \cdot \sqrt{M}))$ I/Os, optimal.
Simple I/O-Efficient Data Structures

Stack (LIFO Order – Last In First Out):

- Maintain an combined input/output buffer of size $2 \cdot B$ in memory.

- **Push**: Insert new element into buffer; if buffer now full, write bottom $B$ elements to disk.

- **Pop**: remove top element from buffer; if buffer now empty, read next block from disk.

I/O-complexity of Push/Pop:

- Best-Case: 0 I/Os
- Worst-Case: 1 I/O
- Amortized: $1/B$ I/Os

Obs: After an I/O, the buffer contains exactly $B$ elements.
Question: Why do need **TWO** blocks in internal memory?
FIFO-Queue

First In First Out Queue

- Maintain an **input buffer** and an **output buffer** (each of size $B$) in memory.
- **Insert:** put new element into input buffer; if buffer now full, write to disk.
- **Remove:** take element from output buffer (if empty from input buffer); if buffer now empty, read next block from disk.

I/O-complexity of Insert/Remove:

- Best-Case: 0 I/Os
- Worst-Case: 1 I/O
- Amortized: $1/B$ I/Os
Lists

- Direct implementation: 1 I/O for when following a link, $\Theta(N)$ I/Os for traversing $N$ elements

- Faster $O(sort(N))$ traversal: list ranking preprocessing, later
Lists cont.

First attempt:

- Use locality: store $B$ consecutive elements together

⇒ Traversal: $N/B = O(\text{scan}(N))$ I/Os

- An insertion or deletion can cost $\Theta(N/B)$ I/Os
Second attempt:

- Relax the invariant: $\geq \frac{2}{3}B$ elements in every pair of consecutive blocks.

- Traversal: $\leq 3N/B = O(\text{scan}(N))$ I/Os.

- Insertion into block $i$:
  - block $i$ is space: 1 I/O
  - block $i$ is full:
    - a neighbor has space: push an element to it, $O(1)$ I/Os
    - both neighbors are full: split block $i$ into 2 blocks of $\approx B/2$ elements, $O(1)$ I/Os ($\geq B/6$ deletions needed to violate the invariant)
  - Deletion from block $i$:
    - if blocks $i$ and $i + 1$ or blocks $i$ and $i - 1$ have $\leq 2B/3$ elements
      $\Rightarrow$ merge the two blocks, $O(1)$ I/Os
    $\Rightarrow O(1)$ I/Os per update (the best for lists)
The STXXL Library
I/O-Efficient Software Libraries

**Advantages**

- Abstract away the technical details of I/O
- Provide implementation of basic I/O-eff. algorithms and data structures

⇒ Boost algorithm engineering

**Existing Libraries**

- TPIE: many (geometric) search data structures
- LEDA-SM: extension of LEDA (discontinued)

+ Good demonstrations of the external memory concepts
- Do not implement many features that speed up I/O-efficient algorithms
# I/O-Efficient Software Libraries

## Advantages
- Abstract away the technical details of I/O
- Provide implementation of basic I/O-eff. algorithms and data structures
  ⇒ Boost algorithm engineering

## Existing Libraries
- TPIE: many (geometric) search data structures
- LEDA-SM: extension of LEDA (discontinued)
  + Good demonstrations of the external memory concepts
  – Do not implement many features that speed up I/O-efficient algorithms
The STXXL Library

- **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)

  - Compatible with STL
  - Performance–oriented
STXXL Features

- Transparent **parallel** disk support
- Handles very large problems (up to **petabytes**)
- **Pipelining** saves many I/Os
- 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. selection alg.)
Challenges

- Cheap case for $\geq 8$ hard disks
- Many fast PCI slots for ATA controllers (no bus bottlenecks)
- Wide Parallel ATA cables worsen airflow (later system use Serial ATA)
- File system scalability: very large files

$\Rightarrow$ 375 MB/s ($\approx 98\%$ of the peak) for about 3000 Euro in 2002
$\Rightarrow$ Other systems: 10 disks $= 640$ MB/s, 4 disks $= 214$ MB/s
STXXL Design

Applications

STL-user layer
- Containers: vector, stack, set, priority_queue, map
- Algorithms: sort, for_each, merge

Streaming layer
- Pipelined sorting, zero-I/O scanning

Block management (BM) layer
- Typed block, block manager, buffered streams, block prefetcher, buffered block writer

Asynchronous I/O primitives (AIO) layer
- Files, I/O requests, disk queues, completion handlers

Operating System
STXXL Design: AIO Layer

- Hides details of async. I/O (portability, user-friendly)
- Implementations for Linux/MacOSX/BSD/Solaris and Windows systems
- Asynchrony provided by POSIX threads or Boost Threads
- Unbuffered I/O support: more control over I/O
Block abstraction
Parallel disk model
(Randomized) striping and cycling
Parallel disk buffered writing and optimal prefetching

[Hutchinson&Sanders&Vitter01]
**STXXL User Layers**

- STL-user layer: compatible with STL, vector, stack, queue, deque, priority queue, map, sorting, scanning
- Streaming layer: programming with pipelining
Some STXXL Containers

Stacks:
- Few variants
  - Classic, has 2 blocks
  - Grow-shrink, does **prefetching/buffering** (own buffer pool)
  - Grow-shrink 2, does prefetching/buffering (shared buffer pool)

Queue:
- the same buffering techniques as `stxxl::stack`

Vector – ST(XX)L dynamic array:
- caches some blocks (LRU)
- $O(N/DB)$ I/Os scanning

Deque: double ended queue
- push/pop from/to the **both ends** in $O(1/DB)$ I/Os
- implemented as adapter of `stxxl::vector` (**circular wrapping**)

P. Sanders, R. Dementiev (ITI)
I/O-Efficient Algorithms and Data Structures
July 10, 2007 38/162
Some **STXXL** Containers

Stacks:
- Few variants
  - Classic, has 2 blocks
  - Grow-shrink, does **prefetching/buffering** (own buffer pool)
  - Grow-shrink 2, does prefetching/buffering (shared buffer pool)

Queue:
- **the same buffering techniques as** `stxxl::stack`

Vector – ST(XX)L dynamic array:
- caches some blocks (**LRU**)
- $O(N/DB)$ I/Os scanning

Deque: double ended queue
- push/pop from/to the both ends in $O(1/DB)$ I/Os
- implemented as adapter of `stxxl::vector` (**circular** wrapping)
Some STXXL Containers

Stacks:
- Few variants
  - Classic, has 2 blocks
  - Grow-shrink, does **prefetching/buffering** (own buffer pool)
  - Grow-shrink 2, does prefetching/buffering (shared buffer pool)

Queue:
- the same buffering techniques as `stxxl::stack`

Vector – ST(XX)L dynamic array:
- caches some blocks (**LRU**)
- $O(N/DB)$ I/Os scanning

Deque: double ended queue
- push/pop from/to the both ends in $O(1/DB)$ I/Os
- implemented as adapter of `stxxl::vector` (**circular wrapping**)
Some STXXL Containers

Stacks:
- Few variants
  - Classic, has 2 blocks
  - Grow-shrink, does prefetching/buffering (own buffer pool)
  - Grow-shrink 2, does prefetching/buffering (shared buffer pool)

Queue:
- the same buffering techniques as stxxl::stack

Vector – ST(XX)L dynamic array:
- caches some blocks (LRU)
- $O(N/DB)$ I/Os scanning

Deque: double ended queue
- push/pop from/to the both ends in $O(1/DB)$ I/Os
- implemented as adapter of stxxl::vector (circular wrapping)
Experiments with Stacks: Insertion

gs2 = grow-shrink (overlapping) stacks  n = normal/classic stacks (2B buffer)

4-byte elements

32-byte elements
Experiments with Stacks: Deletion

\(gs2\) = grow-shrink (overlapping) stacks  \(n\) = normal/classic stacks (2B buffer)

![Graphs showing performance comparison between different stack implementations for 4-byte and 32-byte elements.](image-url)
Experiments with Stacks: Multiple Disks Insertion

\( gs2 = \text{grow-shrink (overlapping) stacks} \)

\[ \text{4/32-byte elements} \]
Some STXXL containers

Map (search tree):
- implemented as $B^+$-tree, later
- caches some nodes and leaves in internal memory (LRU)
- $O(\log_B N)$ I/Os for LOCATE query
- supports iterators: $O(N/B)$ I/Os for range scanning

Priority queue:
- implemented as sequence heap, later
- non-addressable
- $\approx O\left(\frac{1}{N}\text{sort}(N)\right)$ I/Os for DELETEMIN, INSERT
- overlapping, prefetching, buffering
Some STXXL containers

Map (search tree):
- implemented as $B^+$-tree, later
- caches some nodes and leaves in internal memory (LRU)
- $O(\log_B N)$ I/Os for LOCATE query
- supports iterators: $O(N/B)$ I/Os for range scanning

Priority queue:
- implemented as sequence heap, later
- non-addressable
- $\approx O\left(\frac{1}{N} \text{sort}(N)\right)$ I/Os for DELETERMIN, INSERT
- overlapping, prefetching, buffering
Generate Random Graph with STXXL

1. `stxxl::vector<edge> Edges(10000000000ULL);`
2. `std::generate(Edges.begin(), Edges.end(), random_edge());`
3. `stxxl::sort(Edges.begin(), Edges.end(), edge_cmp(), 512*1024*1024);`
4. `stxxl::vector<edge>::iterator NewEnd =`
5. `std::unique(Edges.begin(), Edges.end());`
6. `Edges.resize(NewEnd - Edges.begin());`
Streaming Layer and Pipelining

- EM algorithm ⇒ data flow through a DAG
- Feed output data stream directly to the consumer algorithm
- A new iterator-like interface for EM algorithms
- Basic pipelined implementations (file, sorting nodes, etc.) provided by STXXL
- Saves many I/Os (factor 2–3) in many EM algorithms
Maximal Independent Set (+input generation)

- An independent set $I$ is a set of nodes on a graph $G$ such that no edge in $G$ joins two nodes in $I$. A maximal independent set is an independent set such that adding any other node would cause the set not to be independent anymore.

I/O optimal algorithm [ZehPhd]: time-forward processing, scanning, sorting, priority queue
Given a labelling $\phi$, compute a labelling $\psi$ so that $\psi(v)$ is computed from $\phi(v)$ and $\psi(u_1), \ldots, \psi(u_r)$, where $u_1, \ldots, u_r$ are $v$’s in-neighbors.
Time-Forward Processing

- Assume nodes are given in topologically sorted order.
  ⇒ Use priority queue Q to send data along the edges.
- Node ID ≡ PQ priority
- Send message $x$ from $u$ to $v$ ≡ $\text{INSERT}(v, x)$
- Receive message $x$ at node $v$ ≡ $(v, x) : = \text{DELETEMIN}()$
Time-Forward Processing

Analysis:

- Vertex set + adjacency lists scanned
  \[ \Rightarrow O(\text{scan}(|V| + |E|)) \] I/Os
- Priority queue:
  - Every edge inserted into and deleted from Q exactly once
  \[ \Rightarrow O(|E|) \] priority queue operations (each costs \[ O\left(\frac{1}{|E|} \text{sort}(|E|)\right) \] I/Os)
  \[ \Rightarrow \text{Total: } O(\text{sort}(|E|)) \] I/Os
MIS: Pseudocode

**GreedyMIS:**

I := 0  
for every vertex v in G do  
    if no neighbor of v is in I then  
        Add v to I  
    end if  
end for
MIS: STXXL Code

edges: sorted outgoing edges (adjacency lists)
depend: event queue

\( v \in \text{depend} \) if \( \exists u: (u, v) \in E \land u \in \text{MIS} \) (i.e. \( v \) cannot be included into MIS)

\begin{verbatim}
1  pq_type depend(PQ_PPOOL_MEM, PQ_WPOOL_MEM);
2  stxxl::vector<node_type> MIS; // output
3  for (; !edges.empty(); ++edges) {
4      while (!depend.empty() && edges->src > depend.top())
5          depend.pop(); // delete old events
6      if (depend.empty() || edges->src != depend.top()) {
7          if (MIS.empty() || MIS.back() != edges->src)
8              MIS.push_back(edges->src);
9          depend.push(edges->dst);
10      }
11  }
\end{verbatim}
MIS: Running Times

- Debian Linux, g++ -O3
- 2× Xeon 2GHz
- single disk
- \( N = 2000 \) MBytes
- \( M = 512 \) MBytes
- TPIE: only graph gen.
- \texttt{STXXL} PQ is 3 times faster
MIS: Larger Inputs

- Only graph generation
- single disk
- $N = 16$ GBytes
- $M = 512$ MBytes
- Scales well
MIS: More Disks

- 2,4 disks
- $N = 2000$ MBytes
- $M = 512$ MBytes
- Pipel. – CPU bound
- I/O-wait counters
MIS: The Largest Graph

- The largest graph:
- $4.3 \cdot 10^9$ nodes, $13.4 \cdot 10^9$ edges = 100 GBytes
- Working space takes 4 hard disks
- Computation on an Opteron system took 3h 7min
Active STXXL Users We Know About

1. University of Karlsruhe, Germany (text processing, graph algorithms, practical courses)
2. Max-Planck-Institut für Informatik, Germany (bio-informatics, graph algorithms)
3. DIMACS Center, Rutgers University, USA (graph analysis, data mining)
4. University of Rome “La Sapienza”, Italy (connected components)
5. University of Texas at Austin, USA (Gaussian elimination)
6. Bitplane AG, Switzerland (visualization and analysis of 3D and 4D microscopic images)
7. Philips Research, The Netherlands (differential cryptographic analysis)
8. Dalhousie University, Canada (N-gram extraction)
9. Florida State University, USA (construction of Voronoi diagrams)
10. Montefiore Institute, Belgium (big sparse matrices)
11. University of British Columbia, Canada (topology analysis of large networks)
12. Bayes Forecast, Spain (statistics and time series analysis)
13. Indian Institute of Science in Bangalore, India (suffix array construction)
14. Rensselaer Polytechnic University, USA (suffix array construction)
15. Institut français du pèrole, France (analysis of seismic files)
16. Northumbria University, UK (search trees)
17. University of Trento, Italy (text compression)
18. Norwegian University of Science and Technology in Trondheim, Norway (suffix array construction)
STXXL Priority queue

Based on sequence heaps:

+ prefetching
+ buffered writing
Insert-All-Delete-All Time

3 Ghz Pentium 4, $M = 1$ GByte, 1 SATA disk, random input
I/O-volume 2 – 5.5 times less than [Brengel at al.](LEDA-SM)!
Searching I/O-efficiently

In internal memory:

- `std::binary_search` (static search)
- `std::map` (dynamic binary red-black tree)

⇒ I/O-inefficient $O(\log_2 N)$ I/Os

Implement STXXL searching (`stxxl::map`) as a $B^+$-tree
Implement STXXL searching (stxxl::map) as a $B^+$-tree:

- generalization of binary trees: up to $B$ children per node
- $O(\log_B N)$ I/Os for LOCATE, INSERT, DELETE
- very practical: used in relational databases, NTFS, ReiserFS, XFS, ...
Implementation of `stxxl::map`

- **root** as `std::map` with size limit
- LRU cache for internal nodes
- LRU cache for leaves
- full support of STL iterators, $N/B$ I/O scanning with prefetching
Experiments with \texttt{stxxl::map}

Dual-Core Opteron 2GHz, M=1GByte, D=1
32-bit random keys, 32-bit data field
Competitors: TPIE, Berkeley DB (used e.g. in MySQL)

Bulk construction:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{chart.png}
\caption{Comparison of BDB btree, TPIE btree, and \texttt{stxxl::map}}
\end{figure}
Insert 100,000 random records

100 % fill factor: load 2 leaves and save 3 leaves per insertion = 25 ms
BDB compresses key prefixes, less I/Os, tuned splitting heuristics
Locate 100,000 random records

load random leaf = 10-13 ms
Berkeley DB Interfaces

```c
struct my_key { char keybuf[KEY_SIZE]; }
struct my_data { char databuf[DATA_SIZE]; }

Dbc *cursorp; /* data base cursor
// db is the BDB B-tree object
db.cursor(NULL, &cursorp, 0); /* initialize cursor

for (int64 i = 0; i < n_locates; ++i)
{
    rand_key(key_storage); /* generate random key
    // initialize BDB key object for storing the result key
    Dbt keyx(key_storage.keybuf,KEY_SIZE);
    // initialize BDB key object for storing the result data
    Dbt datax(data_storage.databuf,DATA_SIZE);
    cursorp->get(&keyx, &datax,DB_SET_RANGE); /* perform locate
}
```

C-like, no templates
### STXXL Interfaces

```cpp
struct my_key { char keybuf[KEY_SIZE]; };
struct my_data { char databuf[DATA_SIZE]; };

std::pair<my_key, my_data> element; // key-data pair

for (i = 0; i < n_locates; ++i)
{
    rand_key(i, element.first); // generate random key
    // perform locate, CMap is a constant reference to a map object
    map_type::const_iterator result = CMap.lower_bound(element.first);
}
```

simple, stronger typing
Algorithm Engineering for Large Data Sets

Engineering from the **bottom to the top**:

- many disks $\leadsto$ CPU-bound $\leadsto$ look at internal algorithms, RAID-0 $\leadsto$ suboptimal
- Pipelining to save I/Os, overlap I/O and computation, easy to use library, abstraction, rapid prototyping
- Controlled unbuffered asynchronous I/O, scalable file systems
- **Bottleneck-free** hardware, I/O-subsystem with parallel disks
Algorithm Engineering for Large Data Sets

**Engineering from the bottom to the top:**

- many disks $\leadsto$ CPU-bound $\leadsto$ look at internal algorithms,
- RAID-0 $\leadsto$ suboptimal
- Pipelining to save I/Os, overlap I/O and computation, easy to use library, abstraction, rapid prototyping
- **Controlled** unbuffered asynchronous I/O, scalable file systems
- Bottleneck-free hardware I/O-subsystem with parallel disks
Engineering from **the bottom to the top:**

- many disks $\rightarrow$ CPU-bound $\rightarrow$ look at internal algorithms, RAID-0 $\rightarrow$ suboptimal

- **Pipelining** to save I/Os, overlap I/O and computation, easy to use library, abstraction, rapid prototyping

- Controlled unbuffered asynchronous I/O, scalable file systems

- **Bottleneck-free** hardware I/O-subsystem with parallel disks
Algorithm Engineering for Large Data Sets

Engineering from the bottom to the top:

- many disks \(\rightsquigarrow\) CPU-bound \(\rightsquigarrow\) look at internal algorithms,
  RAID-0 \(\rightsquigarrow\) suboptimal
- Pipelining to save I/Os, overlap I/O and computation, easy to use library, abstraction, rapid prototyping
- Controlled unbuffered asynchronous I/O, scalable file systems
- Bottleneck-free hardware I/O-subsystem with parallel disks

```
1. Radix-Sort edges
2. Compute CCs
3. Merge CCs
4. Add new edges
5. If #CC > 1 goto yes

register unsigned pReg = param;
open(file, filename, O_DIRECT);
while (--i) {
    j = l<<(+pReg);
    buffer[i-1] = in1[j] + in2[i+l];
    for(int j=pReg;j<i;++j) {
        bucket[j] = pReg + i - 1
    }
```
Some Cache Configurations

Only a few systems:

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Pentium III</th>
<th>MIPS 10000</th>
<th>AMD Athlon</th>
<th>Itanium 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock rate</td>
<td>2400 MHz</td>
<td>800 MHz</td>
<td>175 MHz</td>
<td>1333 MHz</td>
<td>1137 MHz</td>
</tr>
<tr>
<td>L1 data cache size</td>
<td>8 KB</td>
<td>16 KB</td>
<td>32 KB</td>
<td>128 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>L1 line size</td>
<td>128 B</td>
<td>32 B</td>
<td>32 B</td>
<td>64 B</td>
<td>64 B</td>
</tr>
<tr>
<td>L1 associativity</td>
<td>4-way</td>
<td>4-way</td>
<td>2-way</td>
<td>2-way</td>
<td>4-way</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>512 KB</td>
<td>256 KB</td>
<td>1024 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>L2 line size</td>
<td>128 B</td>
<td>32 B</td>
<td>32 B</td>
<td>64 B</td>
<td>128 B</td>
</tr>
<tr>
<td>L2 associativity</td>
<td>8-way</td>
<td>4-way</td>
<td>2-way</td>
<td>8-way</td>
<td>8-way</td>
</tr>
<tr>
<td>TLB entries</td>
<td>128</td>
<td>64</td>
<td>64</td>
<td>40</td>
<td>128</td>
</tr>
<tr>
<td>TLB associativity</td>
<td>full</td>
<td>4-way</td>
<td>64-way</td>
<td>4-way</td>
<td>full</td>
</tr>
<tr>
<td>RAM size</td>
<td>512 MB</td>
<td>256 MB</td>
<td>128 MB</td>
<td>512 MB</td>
<td>3072 MB</td>
</tr>
</tbody>
</table>

How can we write portable code that runs efficiently on different multilevel caching architectures?
Cache-Obliviousness

- $N$ — size of input
- $M$ — size of main/fast memory ($M \ll N$)
- $B$ — size of transfer block
- Cost measure – number of I/Os

**Cache-Oblivious (CO) Model**: $M, B$ unknown to the algorithm

$\Rightarrow$ Good on one level $\Rightarrow$ good on all memory levels

$\Rightarrow$ One algorithm for all platforms ?!
Cache-Aware (I/O Model) vs. Cache-Oblivious

Cache-aware: fixed parameters $M$ and $B$
Cache-oblivious: no parameters?! no tuning required ?!
Ideal-Cache Model

[Frigo, Leiserson, Prokop, Ramachandran 1999].

- Program with only one memory (single cache, hidden).
- Analysis like in I/O model; assumes arbitrary $M$ and $B$.
- Suppose optimal off-line cache replacement strategy for $M$ and $B$.
- Suppose fully-associative cache.

Realistic ??

- Multi-level.
- LRU (least recently used) replacement.
- Limited associativity.
Justification of the Ideal-Cache Model

**Optimal replacement:** $\text{LRU} + 2 \times \text{cache size} \Rightarrow$ at most $2 \times \text{cache misses}$ [ST85]

**Corollary:** If $T_{M,B}(N) = O(T_{2M,B}(N))$ (regularity condition)
$\Rightarrow$ # cache misses using LRU is $O(T_{M,B}(N))$.

**Two memory levels:** Optimal cache-obliv. alg. with $T_{M,B}(N) = O(T_{2M,B}(N))$
$\Rightarrow$ optimal # cache misses on each level of a multilevel LRU cache.

**Fully-associative cache:** Simulation of LRU (needs to know $M$ and $B$)
- Direct mapped cache.
- Explicit memory management.
- Dictionary (2-universal hash functions) of cache lines in memory
- Expected $O(1)$ access time of cache line in memory.
How to make algorithms cache-oblivious?

Only a few golden rules:

- Avoid unstructured access patterns.
- Incorporate LOCALITY directly into the algorithm.

Tools:

- Scanning.
- Sorting.
- Special cache-oblivious data structures and data layouts.
- "Simulation" of parallel algorithms.
- Divide and Conquer / Recursion
- Tall-Cache Assumption ($M = \Omega(B^2)$). $B = 16 - 128$ bytes!
Warmup: Scanning

already cache-oblivious!

\[
\begin{align*}
\text{sum} &= 0; \\
\text{for} \ i=1 \ \text{to} \ N \ \text{do} \ \text{sum} := \text{sum} + A[i];
\end{align*}
\]

\[\text{scan}(N) = O(N/B) \text{ I/Os, optimal.}\]

Remarks:

- No need to know \(B\) here.
- Scanning backwards would be slower in practice.
Repetition: Matrix Transposition

Problem:

\[ C = A^T, \ C_{i,j} = A_{j,i} \]

Layout of matrices:

Row major | Column major | \(4 \times 4\)-blocked
Repetition: Cache-Aware Matrix Transposition

**Algorithm 2: Blocked algorithm**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
</tr>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
<tr>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
</tr>
<tr>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
</tr>
</tbody>
</table>

Partition $A$ ($C$) into submatrices $A^{r,s}$ ($C^{r,s}$) of size $B \times B$, $B^2 = \Theta(M)$.

- Transfer each submatrix $A^{r,s}$ to the internal memory $\Rightarrow B$ I/Os
- Apply Algorithm 1 to $A^{r,s}$ (internally)
- Transfer it to $C^{s,r} \Rightarrow B$ I/Os

$$2 \frac{N^2}{B^2} \cdot B = O\left(\frac{N^2}{B}\right)$$ I/Os, optimal.
CO Matrix Transposition

\[ A = \begin{pmatrix} A1 & A2 \end{pmatrix}, \quad C = \begin{pmatrix} C1 \\ C2 \end{pmatrix} \]

\[
\text{CO\_Transpose}(A, C) \\
\{ \\
\quad \text{CO\_Transpose}(A1, C1); \\
\quad \text{CO\_Transpose}(A2, C2); \\
\}
\]

**I/O-complexity:**

Case I: \( N \leq \alpha B \) then \( Q(N) \leq N^2 / B + O(1) \) I/Os

Case II: \( N > \alpha B \) then \( Q(N) = 2Q(N/2) + O(1) \) I/Os

⇒ solves to \( O\left(1 + N^2 / B\right)\), optimal
Performance of Matrix Transposition [Chatterjee,Sen]

300 MHz UltraSPARC-II, 2 MB L2 cache, 16 KB L1 cache, page size 8 KB, 64 TLB entries

<table>
<thead>
<tr>
<th>$\log_2 N$</th>
<th>Naive</th>
<th>I/O</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.21</td>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td>11</td>
<td>0.86</td>
<td>0.49</td>
<td>0.45</td>
</tr>
<tr>
<td>12</td>
<td>3.37</td>
<td>1.63</td>
<td>2.16</td>
</tr>
<tr>
<td>13</td>
<td>13.56</td>
<td>6.38</td>
<td>6.69</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\log_2 N$</th>
<th>Naive</th>
<th>I/O</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.14</td>
<td>0.12</td>
<td>0.09</td>
</tr>
<tr>
<td>11</td>
<td>0.87</td>
<td>0.42</td>
<td>0.47</td>
</tr>
<tr>
<td>12</td>
<td>3.36</td>
<td>1.46</td>
<td>2.03</td>
</tr>
<tr>
<td>13</td>
<td>13.46</td>
<td>5.74</td>
<td>6.86</td>
</tr>
</tbody>
</table>

⇒ Tuned cache-aware algorithm is faster than CO algorithm
⇒ CO algorithm is much faster than naive algorithm
Cache Simulator Results

\[ N = 2^{13}, \quad B = 2^6 \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Data refs</th>
<th>L1 misses</th>
<th>TLB misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>134 mln</td>
<td>38 mln</td>
<td>34 mln</td>
</tr>
<tr>
<td>I/O</td>
<td>403 mln</td>
<td>37 mln</td>
<td>0.3 mln</td>
</tr>
<tr>
<td>CO</td>
<td>134 mln</td>
<td>56 mln</td>
<td>2 mln</td>
</tr>
</tbody>
</table>

- Naive alg.: few data accesses but many non-local
- CO alg.: too deep recursion \( \Rightarrow \) many L1 and TLB misses
- CO alg.: breaking recursion earlier gives better performance
Cache-Oblivious Searching

- Binary search: $\Theta(\log_2 N)$ I/Os, suboptimal
- B-tree (B-way search): $\Theta(\log_B N)$ I/Os, but needs to know $B$
- Cache-oblivious search with $\Theta(\log_2 N)$ I/Os, possible?
Recursive memory layout of a perfect binary tree (van Emde Boas layout):

Observation: if a subtree fits in a block its height is $\geq \frac{\log B}{2}$

$\Rightarrow$ a search crosses $O\left(\frac{\log n}{\log B}\right)$
subtrees

$\Rightarrow$ $O(\log_B n)$ I/Os

Memory layout:

```
top tree  bot tree 1  bot tree 2  bot tree 3  ...  bot tree $\sqrt{n}$
```

$\sqrt{n}$  $\sqrt{n}$  $\sqrt{n}$  $\sqrt{n}$  $\sqrt{n}$
Experiments [Kumar2002]

Itanium processor, 2 GByte RAM, 48 byte elements, random input

General search $\equiv$ searching with pre-order layout
Cache-oblivious $\equiv$ searching with vEB layout
Cache-Oblivious Sorting

Simple recursive Mergesort: \( Q(n) = 2Q(n/2) + \Theta(n/B) \) 
\[ \Rightarrow Q(n) = \Theta\left(\frac{N}{B} \log_2 \frac{N}{B}\right) \text{ I/Os, suboptimal} \]

How to increase log base to \( M/B \) without knowing \( M \) and \( B \)?

Solution: a recursive merger (k-funnel) [Frigo et al. 1999]
**K-funnel**

- **Input:** $k$ sorted sequences
- **Outputs** $k^3$ elements

**vEB layout:**

<table>
<thead>
<tr>
<th>$R$</th>
<th>$B_1$</th>
<th>$L_1$</th>
<th>$B_2$</th>
<th>$L_2$</th>
<th>$B_3$</th>
<th>$L_3$</th>
<th>...</th>
<th>$B_{\sqrt{k}}$</th>
<th>$L_{\sqrt{k}}$</th>
</tr>
</thead>
</table>

Merging $k^3$ elements takes $\Theta\left(\frac{k^3}{B} \log_{M/B} \left(\frac{k^3}{B}\right) + k\right)$ I/Os and $\Theta(\log_2 n)$ work.
Lazy Funnelsort [BroFag2002]

1. Split input into $k = n^{1/3}$ contiguous segments each of size $n/k = n^{2/3}$
2. Recursively sort each segment
3. Apply the $k$-funnel to merge the sorted sequences.

I/O-complexity: $Q(n) = n^{1/3} Q(n^{2/3}) + O\left(\frac{n}{B} \log_{M/B}(n/B) + n^{1/3}\right)$
solves to $Q(n) = \frac{n}{B} \log_{M/B}(n/B)$

Practical?
**k-funnel structure:**

- **recursive** implementation is faster than iterative (+ special allocator):
  Pentium 4 caches return instruction address

- navigation with **pointers is faster** than implicit layouts:
  too expensive CPU computation
Tested mergers of degree $z = 2..9$: $z = 4$ is the optimum

- small $z$: less instructions
- large $z$: less levels, elements movements $1/\log(z)$, navigations
Other Optimizations

- Compute repeating merger configurations only once: 3-5 % speedup
- Base case: for < 100 elements use `std::sort`
- Tuning constants $\alpha, d$ for buffer lengths ($\alpha k^d$)
  \[\Rightarrow \alpha = 16, \quad d = 2.5\]
Quicksort Implementations

**GCC ≡ std::sort** GCC C++ Version 3.2 implementation

**Dink ≡ std::sort** Dinkumware incl. in Intel C++ compiler 7.0, 3-way

**Mix ≡ own implementtion of [BentleyMcIlroy93]**, 3-way partitioning

**Sedge ≡ implementation by Sedgewick (book)**

![Graph showing performance comparisons between GCC, Dink, Mix, and Sedge for Uniform pairs on Pentium 4 and AMD Athlon processors.](image)
In-RAM Experiments

\textbf{msort-*} \equiv \text{cache-aware implementations from [Xiaoao et al. 2000]}

\textbf{Rmerge} \equiv \text{cache-aware algorithm from [Arge et al. 2002]}

Pentium III, Itanium 2: similar behavior
In-RAM Experiments on MIPS 10000

CPU cycles are costly
many registers
External Memory Experiments

Funnelsort run on inputs larger than $M$:

- use memory mapping (virtual memory):

```c
int * array;
mmap(&array, "filename");
// algorithm begins

// reading memory:
var = array[i]; // file → OS cache (RAM) → processor caches → CPU
// if cache is full, flush data: CPU cache → OS cache (RAM) → file

// writing memory:
array[j] = var2;
// checks caches, loads page from slower level if needed
// flush data if cache full

// algorithm finishes
munmap(array);
```
External Memory Experiments cont.

[Ajwani et al. 2007]:

<table>
<thead>
<tr>
<th>$n$</th>
<th>Funnelsort</th>
<th>stxxl:::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>$256 \times 10^6$</td>
<td>21 min</td>
<td>8 min</td>
</tr>
<tr>
<td>$512 \times 10^6$</td>
<td>46 min</td>
<td>13 min</td>
</tr>
<tr>
<td>$1024 \times 10^6$</td>
<td>96 min</td>
<td>25 min</td>
</tr>
</tbody>
</table>

⇒ I/O-efficient/aware algorithms tuned to external memory perform better (smaller constant factors, overlapping of I/O and computation)
Suffix Sorting

- Sort suffixes $T[i..n]$ of string $T[0..n]$.
- The result is **Suffix Array**: $SA[i]$ stores the position of $i$th smallest suffix
  - Powerful full-text search
  - Burrows-Wheeler text compression (UNIX bzip2)
  - Bioinformatics

Big interest in **BIG inputs** but no **PRACTICAL** I/O-efficient implementations existed!
Doubling

Lexicographic names

Choose integer name IDs such that:

\[ T[i, i + 2^k] \leq T[j, j + 2^k] \iff \text{name}(T[i, i + 2^k]) \leq \text{name}(T[j, j + 2^k]) \]

Doubling Algorithm

\[
\text{for } k := 1 \text{ to } \lceil \log n \rceil \text{ do} \\
\quad \text{find lexicographic names for } T[i, i + 2^k] \\
\quad \text{if the names are unique then} \\
\quad \quad \text{return suffix array}
\]
How to generate names for the next iteration?

Idea: $T[i, i + 2 \cdot 2^k] \leq T[j, j + 2 \cdot 2^k]$ 

iff 

$(\text{name}(T[i, i + 2^k]), \text{name}(T[i + 2^k, i + 2 \cdot 2^k])) \leq (\text{name}(T[j, j + 2^k]), \text{name}(T[j + 2^k, j + 2 \cdot 2^k]))$ 

⇒ 

**Name the pairs** $(\text{name}(T[i, i + 2^k]), \text{name}(T[i + 2^k, i + 2 \cdot 2^k]))$ 

to get $\text{name}(T[i, i + 2^{k+1}])$
Doubling Algorithm: Pseudocode

Function doubling\( (T) \)
\[ S := \langle ((T[i], T[i + 1]), i) : i \in [0, n] \rangle \]
for \( k := 1 \) to \( \lceil \log n \rceil \) do
  sort \( S \)
  \( P := \text{name}(S) \)
  invariant \( \forall (c, i) \in P : \)
  \hfill \( c \) is a lexicographic name for \( T[i, i + 2^k] \)
  if the names in \( P \) are unique then
    return \( \langle i : (c, i) \in P \rangle \)
  sort \( P \) by \( (i \mod 2^k, i \div 2^k) \)
S\( := \langle ((c, c'), i) : j \in [0, n), \)
\( (c, i) = P[j], (c', i + 2^k) = P[j + 1] \rangle \)
Lexicographical Naming: Pseudocode

**Function** name($S :\text{Sequence of Pair}$)

$q := r := 0; \quad (\ell, \ell') := (\$, \$)$

result := ⟨⟩

foreach $((c, c'), i) \in S$ do

$q++$

if $(c, c') \neq (\ell, \ell')$ then $r := q; \quad (\ell, \ell') := (c, c')$

append $(r, i)$ to result

return result
Bit Shuffling

**Problem**: distance between $\text{name}(T[i, i + 2^k])$ and $\text{name}(T[i + 2^k, i + 2 \cdot 2^k])$ in $P$ is $2^k$

$\Rightarrow$ need **two** read pointers ($\times 2$ I/Os in the last iterations)

**Solution**: sort $P$ by $(i \mod 2^k, i \div 2^k)$ instead of $i$
Doubling: Example

T = banana -> pair
<(ba,0), (an,1), (na,2), (an,3), (na,4), (a0,5)>
-> sort by pairs
<(a0,5), (an,1), (an,3), (ba,0), (na,2), (na,4)>
-> name
<( 1,5), ( 2,1), ( 2,3), ( 4,0), ( 5,2), ( 5,4)>
-> shuffle by pos (mod 0, mod 1)
455 221 -> pair
<(45,0), (55,2), (50,4), (22,1), (21,3), (10,5)>
-> sort by pairs
<(10,5), (21,3), (22,1), (45,0), (50,4), (55,2)>
-> name
<( 1,5), ( 2,3), ( 3,1), ( 4,0), ( 5,4), ( 6,2)>
unique! -> project -> 531042
Pipelined Doubling

\[(T[j], T[j+1], j)\]

\[(\text{name}(T[j..j+2^i), \text{name}(T[j+2^i..j+2^{i+1}), j)\]

\[i := i+1\]

3n words

sort

form runs

runs merge

name

2n words

sort

i bits

pair

total I/O complexity: \(\text{sort}(5n) \log \text{maxlcp} + O(\text{sort}(n))\)
Discarding

Denote $c_i^k = \text{name}(T[i, i + 2^k])$ (iteration $k$)

What if particular $c_i^k$ is already unique?
⇒ Exclude suffix $i$ from later iterations
⇒ Reduces I/O volume

The names are stable, i.e. if $c_i^k$ is unique then $c_i^k = c_i^h$ for all $h > k$:

- Fully discard from $S$ all tuples $((c, c'), i)$ where $c$ is unique
  ▶ Previous approaches [CF 97] scanned all discarded suffixes in all iterations
- Can not discard $((c, c'), i)$ if $c'$ is unique but $c$ is not
  ▶ Partially discard $(c, i)$ (keep in a separate EM array – only scanned in later iterations)
Pipelined Improved Discarding

- Scan all unique suffixes [CF 97]⇝
  Scan new unique suffixes
- Triples [Kärkkäinen 03]⇝ pairs

\[
\text{sort}(5N) + O(\text{sort}(n)) \text{ I/Os where } N = \sum_i \log \text{distPrefixSize}(T[i..n])
\]
a-Tupling

Sort by first $a^i$ characters in iteration $i$

- large $a$: few iterations, but need to sort long tuples
- small $a$: many iterations, sorting short tuples

Constant Factor in I/O Volume

\[
\begin{array}{ccccccc}
 a & 2 & 3 & 4 & 5 & 6 & 7 \\
(a + 3)/\log a & 5.00 & 3.78 & 3.50 & 3.45 & 3.48 & 3.56 \\
\end{array}
\]

CPU computations for $a = 4$ are cheaper than for $a = 5$, I/O volume differs by only 1.5%
Difference Cover 3 (DC3, Skew) Algorithm

1. sort $T[i..n]$ for $i \mod 3 \in \{1, 2\}$
   sort and name triples
   recurse

2. sort $T[i..n]$ for $i \mod 3 \in \{0\}$
   sort pairs $(T[3i], \text{name}(T[3i+1..n]))$

3. merge using difference cover property of $\{1, 2\}$
   $T[3i..n] \leq T[3j+1..n]$ iff
   $(T[3i], \text{name}(T[3i+1..n])) \leq$
   $(T[3j+1], \text{name}(T[3j+2..n]))$
   $T[3i..n] \leq T[3j+2..n]$ iff
   $(T[3i..3i+1], \text{name}(T[3i+2..n])) \leq$
   $(T[3j+2..3j+3], \text{name}(T[3j+4..n]))$
Pipelined DC3

\[ \text{sort}(30n) + \text{scan}(6n) \text{ I/Os} \]
Experimental Setup

Pipelining + STL-user layer from STXXL

Experiments on a faster machine (Opteron 1.8 GHz, SCSI Seagate 15,000 RPM disks) have shown similar results.

all computations took 30 days, 40 TBytes data moved

Inputs:

- **Genome**: Human Genome ($\approx 4$GByte)
- **Gutenberg**: $\approx 3$GByte English text from Gutenberg project
- **HTML**: $\approx 3$GByte text from a crawl of .gov
- **Source**: $\approx 0.5$GByte Linux sources
- **Random2**: $T \circ T$ with $T := \text{randChar}^{n/2}$
Quadrupling is faster than doubling
Discarding variants are faster (except special inputs)
Random2 I/Os

Non-pipelined doubling implementation has much larger I/O
$D = 4 \Rightarrow$ fast I/O, less CPU work pays off:
non-pipelined doubling is close to pipelined doubling (for $D = 1$, speedup $\approx 2$)
quadrupling with discarding loses quadrupling (complex CPU comput., difficult input)

DC3 compares only pairs or triples vs. quadruples
Genome I/Os

I/O Volume [byte] / n

Doubling
Quadrupling
Discarding
Quad-Discarding
Skew

n

P. Sanders, R. Dementiev (ITI)
I/O-Efficient Algorithms and Data Structures
July 10, 2007 123/162
Genome Time

Graph showing the genome time in microseconds per n for different operations:
- Doubling
- Quadrupling
- Discarding
- Quad-Discarding
- Skew

The x-axis represents n, and the y-axis represents Genome Time in microseconds per n. The graph compares the performance of these operations with respect to the input size n.
P. Sanders, R. Dementiev (ITI)
I/O-Efficient Algorithms and Data Structures
July 10, 2007 125/162
Comparison with Previous Implementations

- $5 \times$ less I/Os than [CF 97]
- $7 - 8 \times$ less clock cycles than [CF 97] (including BGS algorithm)
- $2.4 \times$ faster than internal compressed Genome [LSSSY 02]
- $1.2 \times$ slower than internal Genome on 64 GByte super computer [Sadakane Shibuya 01]
- Faster than linear time internal LCP computation on MPII’s SUN Starfire 15000
### Other Hardware Configurations

#### DC3 with many disks

<table>
<thead>
<tr>
<th>$D$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\mu s/\text{byte}}$</td>
<td>13.96</td>
<td>9.88</td>
<td>8.81</td>
<td>8.65</td>
<td>8.52</td>
</tr>
</tbody>
</table>

$\Rightarrow$ CPU-bound $\iff$

#### A Faster 64-bit Opteron with SCSI disks
- Implementations are 1.7 – 2.4 times faster
- Relative performance is the same
Conclusion

Results

- **STXXL** makes **pipelining** easy. Saves factor 2–3 in I/O volume.
- External DC3 is **practical**
- And better than pipelined 4-tupling with improved discarding

Future Work

- Tune pipelined **sorters**
- Go **parallel**
- Will **discarding** help for **DC** algorithms?
  
Terabytes over night?
DCX algorithm

choose suffixes starting at $I_X = \{i \mid i \mod X \in C_X\}$ (for DC3 $X = 3, C_3 = \{1, 2\}$)

for given $X$ minimize $C_X$ s.t. the order of the remaining suffixes can be reconstructed $\Rightarrow C_X = \{j \mid X - j - 1 \in C'_X\}$, where $C'_X$ is minimum difference cover [Haanpää 2004]

<table>
<thead>
<tr>
<th>$X$</th>
<th>$C'_X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>${0, 1}$</td>
</tr>
<tr>
<td>7</td>
<td>${0, 1, 3}$</td>
</tr>
<tr>
<td>13</td>
<td>${0, 1, 3, 9}$</td>
</tr>
<tr>
<td>21</td>
<td>${0, 1, 6, 8, 18}$</td>
</tr>
<tr>
<td>31</td>
<td>${0, 1, 3, 8, 12, 18}$</td>
</tr>
<tr>
<td>39</td>
<td>${0, 1, 16, 20, 22, 27, 30}$</td>
</tr>
<tr>
<td>57</td>
<td>${0, 1, 9, 11, 14, 35, 39, 51}$</td>
</tr>
<tr>
<td>73</td>
<td>${0, 1, 3, 7, 15, 31, 36, 54, 63}$</td>
</tr>
<tr>
<td>91</td>
<td>${0, 1, 7, 16, 27, 56, 60, 68, 70, 73}$</td>
</tr>
<tr>
<td>95</td>
<td>${0, 1, 5, 8, 18, 20, 29, 31, 45, 61, 67}$</td>
</tr>
<tr>
<td>133</td>
<td>${0, 1, 32, 42, 44, 48, 51, 59, 72, 77, 97, 111}$</td>
</tr>
</tbody>
</table>
I/O-Volume Estimation of DCX

<table>
<thead>
<tr>
<th>$X$</th>
<th>3</th>
<th>7</th>
<th>13</th>
<th>21</th>
<th>31</th>
<th>39</th>
<th>57</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>C_X</td>
<td>$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>sort[$N$]</td>
<td>30</td>
<td>24.75</td>
<td>30.11</td>
<td>38.56</td>
<td>50.12</td>
<td>60.65</td>
<td>79.02</td>
</tr>
<tr>
<td>scan[$N$]</td>
<td>6</td>
<td>3.50</td>
<td>2.89</td>
<td>2.63</td>
<td>2.48</td>
<td>2.39</td>
<td>2.33</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>66</strong></td>
<td><strong>53</strong></td>
<td><strong>63.11</strong></td>
<td><strong>79.75</strong></td>
<td><strong>102.72</strong></td>
<td><strong>123.75</strong></td>
<td><strong>160.37</strong></td>
</tr>
</tbody>
</table>

DC7 has **20 % less I/O-volume** than DC3
I/O-Volume Estimation of DCX with alphabet compression

Genome data (4-character alphabet): pack 16 characters in a 32-bit word

<table>
<thead>
<tr>
<th>$X$</th>
<th>3</th>
<th>7</th>
<th>13</th>
<th>21</th>
<th>31</th>
<th>39</th>
<th>57</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>C_X</td>
<td>\ $</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>sort[$N$]</td>
<td>24.50</td>
<td>18.17</td>
<td>15.46</td>
<td>15.23</td>
<td>15.14</td>
<td>16.57</td>
<td>17.43</td>
</tr>
<tr>
<td>scan[$N$]</td>
<td>2.46</td>
<td>1.63</td>
<td>1.20</td>
<td>0.96</td>
<td>0.80</td>
<td>0.75</td>
<td>0.61</td>
</tr>
<tr>
<td>Total</td>
<td><strong>51.49</strong></td>
<td><strong>37.99</strong></td>
<td><strong>32.13</strong></td>
<td><strong>31.41</strong></td>
<td><strong>31.09</strong></td>
<td><strong>33.89</strong></td>
<td><strong>35.49</strong></td>
</tr>
</tbody>
</table>

more CPU work: practical?
I/O-Efficient Spanning Trees

Simplify the MSF implementation [Schultes 2003]

- no “weight” component in tuples $\Rightarrow$ less I/Os
- base case simplified: no sorting needed for Kruskal’s alg.
- delete node $i$:
  - output the lightest edge $(i, w) \Rightarrow$ output $(i, v)$ with $v = \min \{u : (i, u) \in E\}$
  - $\Rightarrow$ postpone work to later iteration, faster reduction
EM Connected Components

Problem: for each node $v$ find representative node $r(v)$ s.t. $r(v) = r(u)$ iff $u$ and $v$ are in the same component ($\exists$ path between $u$ and $v$)

Adapt the MSF implementation using ideas from [Sibeyn and Meyer]

Preliminaries:

- “question” $(v, u)$ is a preliminary assignment $u = r(v)$
- “answer” $(v, u)$ is the ultimate assignment $u = r(v)$
- assignment of nodes to buckets $b : V \rightarrow \{0..k – 1\}$
During the processing of bucket $i$

if list of $v$ is empty then $r(v) := v$ else $r(v) :=$ smallest entry in the list of $v$;

After the processing of bucket $i$

for $v := u_{i-1} + 1$ to $u_i$ do
   if $r(v) \leq u_{i-1}$ then
      add $(v, r(v))$ to Questions[$b(r(v))$];
   else
      $r(v) := r(r(v))$;
      if $r(v) \leq u_{i-1}$ then
         add $(v, r(v))$ to Questions[$b(r(v))$];
      else
         add $(v, r(v))$ to Answers[$b(v)$];
EM Connected Components: Post-Processing

Post-Processing (An additional pass)

\[
\text{for } i := 1 \text{ to } b \text{ do} \\
\quad \text{read } \text{Answers}[i]; \\
\quad \text{foreach } (v, r(v)) \in \text{Questions}[i] \text{ do} \\
\qquad r(v) := r(r(v)); \\
\qquad \text{add } (v, r(v)) \text{ to } \text{Answers}[b(v)]; \\
\quad \text{write } \text{Answers}[i] \text{ to result;}
\]
Measurements: Speedup over the MSF implementation

<table>
<thead>
<tr>
<th>type</th>
<th>(n/10^6)</th>
<th>(m/10^6)</th>
<th>SF</th>
<th>CC</th>
<th>SF&amp;CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>80</td>
<td>160</td>
<td>7.1</td>
<td>5.8</td>
<td>5.8</td>
</tr>
<tr>
<td>grid</td>
<td>1280</td>
<td>2560</td>
<td>1.8</td>
<td>1.8</td>
<td>1.5</td>
</tr>
<tr>
<td>random</td>
<td>80</td>
<td>160</td>
<td>2.1</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>random</td>
<td>1280</td>
<td>2560</td>
<td>2.1</td>
<td>2.3</td>
<td>1.9</td>
</tr>
<tr>
<td>random</td>
<td>40</td>
<td>320</td>
<td>2.5</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>random</td>
<td>320</td>
<td>2560</td>
<td>2.1</td>
<td>2.5</td>
<td>2.0</td>
</tr>
<tr>
<td>geometric</td>
<td>80</td>
<td>149</td>
<td>2.8</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>geometric</td>
<td>640</td>
<td>1190</td>
<td>1.7</td>
<td>1.6</td>
<td>1.4</td>
</tr>
<tr>
<td>geometric</td>
<td>40</td>
<td>270</td>
<td>3.6</td>
<td>3.4</td>
<td>3.5</td>
</tr>
<tr>
<td>geometric</td>
<td>160</td>
<td>1080</td>
<td>3.3</td>
<td>3.2</td>
<td>3.2</td>
</tr>
</tbody>
</table>

- SF&CC is faster than MSF (factor \(\geq 1.4\))
- CC (without SF) does not carry original node ids
- SF is faster than CC: no third pass is needed, simple CPU work
List Ranking

A fundamental graph problem: compute the **distance** to the list tail/head for each node in a list

**Input (succ links):**

**Output (node_id,dist):**

Internal memory algorithm (just follow links): $\Omega(n)$ I/Os
External Memory List Ranking
External Memory List Ranking

\[ \text{dist}[i] := 1, \ 1 \leq i \leq n \]
External Memory List Ranking

\[ \text{find a Maximal Independent Set } S \Rightarrow O(\text{sort}(n)) \text{ I/Os} \]
delete $S$ and update $\text{dist}$:

$$\text{dist}[\text{pred}[i]] + = \text{dist}[i], \ i \in S$$

$\implies O(\text{sort}(n))$ I/Os
External Memory List Ranking

Find a MIS $S$
delete $S$ and update $\text{dist}$:
\[
\text{dist}[\text{pred}[i]] + = \text{dist}[i], \ i \in S
\]
External Memory List Ranking

Find a MIS $S$
delete $S$ and update $\text{dist}$:

$$\text{dist}[	ext{pred}[i]] + = \text{dist}[i], \ i \in S$$
External Memory List Ranking

restore $S$ and update $dist$: $dist[i] + = dist[succ[i]]$; $i \in S$
restore $S$ and update $dist$:
\[ dist[i] += dist[\text{succ}[i]]; \quad i \in S \]
restore $S$ and update $dist$: $\text{dist}[i] \leftarrow \text{dist}[\text{succ}[i]]; \ i \in S$
EM List Ranking: Analysis

- Recursion step: $O(\text{sort}(n))$ I/Os
- MIS on a list: $|S| \geq n/3$

$$\Rightarrow Q(n) \leq O(\text{sort}(n)) + Q(2n/3) = O(\text{sort}(n)) \text{ I/Os}$$

Not really practical, large I/O volume
A More Practical Algorithm [Sibeyn2004]

The idea

- Similar to Connected Components Alg. [Sibeyn Meyer]
- Each node $v$ keeps a (adjacency) list of entries $(u, l)$: distance between $v$ and $u$ is $l$ (can be negative)
Fast List Ranking: Pseudocode

```plaintext
foreach edge (u, v) ∈ L // Step 1
    if u < v then add (u, −1) to the list of v
    else add (v, 1) to the list of u
for u := n − 1 downto 1 do // Step 2
    if u has list entries (v, l_v) and (w, l_w), v < w < u then
        add (v, l_v − l_w) to the list of w
    else // u has a single entry (w, l_w), w < u
        add (−∞, −l_w) to the list of w
    ref_u := w
    δ_u := l_w
// node 0 has list entries (−∞, l_{head}), l_{head} < 0 and/or (−∞, l_{tail}), l_{tail} > 0
d(0) := l_{last} or 0 if node 0 is the tail // distance from the last node
for u := 1 to n − 1 do // Step 3
    d(u) := d(ref_u) + δ_u
```

Fast List Ranking: Example (Step 2)

(the list nodes are ordered for the simplicity)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1,1)</td>
<td>(2,1)</td>
<td></td>
<td>(1,-1)</td>
<td>(5,-1)</td>
<td>(4,1)</td>
</tr>
<tr>
<td></td>
<td>(1,1)</td>
<td>(2,1)</td>
<td></td>
<td>(1,-1)</td>
<td>(4,2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1,1)</td>
<td>(2,1)</td>
<td></td>
<td>(1,-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(−∞,3)</td>
<td>(1,1)</td>
<td>(2,1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(−∞,3)</td>
<td>(1,1)</td>
<td>(−∞,-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(−∞,3)</td>
<td>(−∞,-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fast List Ranking: Analysis

- Message sending and receiving: an I/O-efficient priority queue
- Each step has \( n \) iterations
- Each iteration step performs \( O(1) \) PQ operations

\[ \Rightarrow \text{Total: } O(\text{sort}(n)) \text{ I/Os} \]

Possible Implementation

- Instead of PQ can use buckets (linear CPU work)
- Estimation from [Sibeyn2004]: \( \times 4 \) faster than the MIS-based algorithm
Euler Tour

Input:

(2, 4), (4, 1), (4, 6), (3, 1), (5, 1)

Output:

[(2, 4), (4, 6)], [(6, 4), (4, 1)], [(4, 6), (6, 4)],
[(4, 1), (1, 3)], [(3, 1), (1, 5)], [(5, 1), (1, 4)],
[(1, 3), (3, 1)], [(1, 4), (4, 2)], [(1, 5), (5, 1)]

no specific order!!
EM Euler Tour Algorithm

- Let \((v, w_1), \ldots, (v, w_k)\) are incident edges of \(v\)
- \(\text{succ}((w_i, v)) = (v, w_{i+1})\) for \(1 \leq i < k\) and \(\text{succ}((w_k)) = (v, w_1)\)

**Algorithm**

1. Scan \(E\) to replace \((v, w)\) with \((v, w)\) and \((w, v)\)
2. Sort the result by target node ids (groups incoming edges together)
3. Scan the result to compute \([((v, w), \text{succ}((v, w)))]\) pairs

\(\Rightarrow O(\text{sort}(n))\) I/Os
Euler Tour Technique

Many applications: e.g. tree rooting (direct each edge from parent to the child)

1. **Compute Euler Tour**
   - $[(2, 4), (4, 6)], [(6, 4), (4, 1)], [(4, 6), (6, 4)],
   - $[(4, 1), (1, 3)], [(3, 1), (1, 5)], [(5, 1), (1, 4)],
   - $[(1, 3), (3, 1)], [(1, 4), (4, 2)], [(1, 5), (5, 1)]$

2. **Run list ranking**
   - $[(2, 4), 10], [(4, 6), 9], [(6, 4), 8], [(4, 1), 7],[(1, 3), 6], [(3, 1), 5], [(1, 5), 4], [(5, 1), 3],
   - $[(1, 4), 2], [(4, 2), 1]$

3. **Sort edges** $[(u, v), d]$ by
   - $(\min(u, v), \max(u, v))$
   and for opposite edges $(u, v)$ and $(v, u)$
   take ones with smaller rank
   - $[(1, 3), 6], [(3, 1), 5], [(1, 4), 2], [(4, 1), 7],
   - $[(1, 5), 4], [(5, 1), 3], [(2, 4), 10], [(4, 2), 1],
   - $[(4, 6), 9], [(6, 4), 8]$

$\Rightarrow O(sort(n))$ I/Os
Other Tree Algorithms using Euler Tour/List Ranking

- Subtree size
- Distance to root
- Preorder numbering
- Postorder numbering
- ...
Breadth First Search

Applications: state exploration, shortest paths, crawling WWW, ...
BFS: 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: 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: $\Theta(m)$ I/Os
- Finding neighbors (adj. lists): $\Theta(n)$ I/Os
Algorithm of Munagala and Ranade

Creating BFS level $t$ (node set $L(t)$):
all reached neighbors of nodes in $L(t-1)$ belong to $L(t-2)$ or $L(t-1)$.

1. $N(L(t-1)) = \text{all neighbours of } L(t-1)$ \hspace{1cm} $\mathcal{O}(|L(t-1)| + \frac{|N(L(t-1))|}{D\cdot B})$ I/Os.
2. eliminate duplicates in $N(L(t-1))$ by sorting \hspace{1cm} $\mathcal{O}(\text{sort}(|N(L(t-1))|))$ I/Os.
3. eliminate nodes already in $L(t-1)$ by scanning \hspace{1cm} $\mathcal{O}(\text{scan}(|L(t-1)|))$ I/Os.
4. eliminate nodes already in $L(t-2)$ by scanning \hspace{1cm} $\mathcal{O}(\text{scan}(|L(t-2)|))$ I/Os.

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

Preprocessing: $O(\text{sort}(n + m))$ 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 $\mathcal{H}$ (hot pool) – prevents the $O(n)$ accesses.
- if first node in a cluster is reached, add all adjacency lists of the cluster to $\mathcal{H}$.
- each adjacency list stays in $\mathcal{H}$ for at most $\mu$ iterations.
- $O\left(\frac{n}{\mu} + \mu \cdot \text{scan}(n + m) + \text{sort}(n + m)\right)$ I/Os.
- **Balancing:** $O\left(\sqrt{nm/B} + \text{sort}(n + m)\right)$ I/Os.
BFS Phase: Example
Randomized Clustering

- choose $n/\mu$ random master nodes.
- grow subgraphs $S_i$ around master nodes in parallel: Label unvisited neighbor nodes & discard them from the representation of $G$.
- any node is labeled after $\mathcal{O}(\mu)$ phases on average.

I/Os per phase ($F = \text{fringe} = \text{active nodes}$):

$$\mathcal{O}(\text{sort}(|F| + |N(F)|) + \text{scan}(|G_{\text{unvisited}}|))$$

$$\Rightarrow \mathcal{O}(\mu \cdot \text{scan}(n + m) + \text{sort}(n + m))$$

expected I/Os for partitioning.

$$\Rightarrow \forall u, v \in S_i : \text{dist}(u, v) \text{ in } G = \mathcal{O}(\mu \cdot \log n) \text{ whp.}$$

P. Sanders, R. Dementiev (ITI)

I/O-Efficient Algorithms and Data Structures
Deterministic Clustering

1. Build a spanning tree: $\Theta(sort(n + m))$ I/Os (randomized).
2. Obtain Euler-tour (length $2n$) and do list ranking: $\Theta(sort(n))$ I/Os.
3. Chop Euler-tour into $2n/\mu$ pieces.
4. Eliminate duplicates: $\Theta(sort(n))$ I/Os.

$\Rightarrow \Theta(sort(n + m))$ I/Os for partitioning.
Experiments

2.0 GHz Opteron, 1 GByte RAM, 250 GByte PATA Seagate disk (65 MByte/s, 9 ms seek time)

IM BFS vs. pipelined MR BFS and MM BFS ($STXXL$)
Tuning MM BFS [Ajwani et al. 2006]

- Choose fast EM list ranking, CC, MSF subroutines (earlier).
- Tune: # clusters, block sizes etc.
- Efficient implementation of internal-memory pools and cluster prefetching.

“Randomize” the shape of underlying spanning trees (det. variant).

\[ \Rightarrow \text{Smaller cluster diameters} \]
## A few numbers

### Total running times in hours:

<table>
<thead>
<tr>
<th>Graph class</th>
<th>n</th>
<th>MunRan</th>
<th>MehMey_Rand</th>
<th>MehMey_Det</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random, $m = 4n$</td>
<td>$2^{28}$</td>
<td>1.4 h</td>
<td>7×</td>
<td>6×</td>
</tr>
<tr>
<td>Webgraph $m \simeq 8n$</td>
<td>$2^{27}$</td>
<td>2.6 h</td>
<td>3.5×</td>
<td>2×</td>
</tr>
<tr>
<td>Random Grid ($2^{14} \times 2^{14}$)</td>
<td>$2^{28}$</td>
<td>2.5×</td>
<td>1.25×</td>
<td>21 h</td>
</tr>
<tr>
<td>Random Grid ($2^{21} \times 2^{7}$)</td>
<td>$2^{28}$</td>
<td>&gt; 100×</td>
<td>&gt; 10×</td>
<td>4.0 h</td>
</tr>
<tr>
<td>Random Grid ($2^{27} \times 2$)</td>
<td>$2^{28}$</td>
<td>&gt; 500×</td>
<td>&gt; 25×</td>
<td>3.8 h</td>
</tr>
<tr>
<td>Random Line</td>
<td>$2^{28}$</td>
<td>&gt; 1000×</td>
<td>&gt; 25×</td>
<td>3.7 h</td>
</tr>
<tr>
<td>Simple Line</td>
<td>$2^{28}$</td>
<td>0.4 h</td>
<td>7×</td>
<td>7×</td>
</tr>
</tbody>
</table>

**Max.** ~ 1/2 year ~ 1 week ~ 1 day
Other Experiments

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

Cache Oblivious Implementation [Christiani]
- Uses CO sorting, CO list ranking, CO MST
- Factor 14-20 slower than EM implementation
EM BFS: Conclusion

- IM-BFS clearly worst on most external instances.
- \textit{[MunRan99]} better than \textit{[MehMey02]} on well-behaved instances (typ. 1 hour vs. 5 hours).
- \textit{[MehMey02} _Det\textit{]} much better than \textit{[MunRan99]} on difficult instances (typ. 4 hours vs. 1/2 year).
- \textit{[MehMey02} _Det\textit{]} proved to be the most robust choice.
- Undirected EM-BFS becomes feasible.

The big challenge for the future:

Directed EM-BFS.