Processing Huge Data Sets

Roman Dementiev

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

March 26, 2007
Outline

1. Introduction
2. Experimental Parallel Disk System
3. The STXXL Library
4. Engineering Algorithms for Large Graphs
5. Engineering Large Suffix Array Construction
6. Porting Algorithms to External Memory
7. Summary
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.com, etc.
Scalability of Algorithms

How to process them
- Buy a TByte main memory? \(\sim\) 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? \(\Rightarrow\) expensive or impossible

- Here: how to process very large data sets cost-efficiently
A straightforward solution

- Use small main memory
  - keep data on cheap disks, "unlimited" virtual memory
- Theory: should work (von Neumann (RAM) model)
  - Unit cost memory access
  - No locality of reference
- Practice: terrible performance
  - Random hard disk accesses are $10^6$ slower than main memory accesses
  - Strong locality of reference

$\Rightarrow$ I/O is the bottleneck
The Parallel Disk Model (PDM)

[Vitter & Shriver’1994]

**Parameters**
- Input size $N$
- Memory size $M \ll N$
- Block size $B$
- The number of disks $D$

**Performance**
- Minimize the number of I/O steps
- In an I/O step try transfer $D$ blocks
- Minimize the number of CPU instructions

I/O-efficient algorithms ≡ external memory algorithms
Engineering Parallel Disk Systems

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; cluster at DIMACS Center for analysis of graph with billions of edges (US Homeland Security project)
The STXXL Library
Advantages

- Abstract away the technical details of I/O
- Provide implementation of basic I/O-efficient 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
  - [http://stxxl.sourceforge.net](http://stxxl.sourceforge.net)
  - 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

Dementiev, Kettner, Sanders

STXXL: The Standard Template Library for Extra Large Data Sets.

*ESA 2005, 13th Annual European Symposium on Algorithms*

- 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.)
STXXL Design

Applications

STL–user layer
- Containers: vector, stack, set
- Algorithms: priority_queue, map, 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

Roman Dementiev (Uni Karlsruhe)
**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
STXXL Design: BM Layer

- 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
Example: Generate Random Graph with STXXL

```cpp
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 = std::unique(Edges.begin(), Edges.end());
5 Edges.resize(NewEnd - Edges.begin());
```
Streaming Layer and Pipelining

- EM algorithm $\Rightarrow$ 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
Parallel Disk Sorting: an Important Part of STXXL

Sorting is the core routine of almost every I/O-eff. algorithm.

We engineer a parallel disk sorting algorithm and implementation which guarantees:

1. Optimal I/O volume \( \text{sort}(N) = O\left(\frac{N}{DB} \log \frac{M}{B} \frac{N}{B}\right) \)
2. Almost perfect overlapping of I/O and computation

Dementiev and Sanders
Asynchronous Parallel Disk Sorting.
SPAA 2003, 15th ACM Symposium on Parallelism in Algorithms and Architectures
**STXXL Performance: Sorting**

2GHz Xeon, 1GByte RAM, 2 GByte input, 32-bit keys
 runs of size 256 MByte, g++ 3.2

![Graph showing sorting performance](image)

- **Single disk**
  - I/O bandwidth of 45.4 MB/s
  - = 95% of peak bandwidth of the disk

- **Eight disks**
  - I/O bandwidth of 315 MB/s

Roman Dementiev (Uni Karlsruhe)  Processing Huge Data Sets  March 26, 2007  19/35
Sorting Performance: I/O Bottleneck Disappears

Changing element size

- 16 GByte input, 32-bit keys, $D = 8$, runs of size 256 MByte, g++ 3.2
- elem. size $\geq 64 \Rightarrow$ merging is I/O bound
- elem. size $\geq 128 \Rightarrow$ run formation is I/O bound
- For small elements I/O is not the bottleneck

Roman Dementiev (Uni Karlsruhe)
Processing Huge Data Sets
March 26, 2007 20/35
**STXXL Performance**

- All other STXXL algorithms and data structures are benchmarked in my PhD thesis against LEDA-SM, TPIE and Berkley DB (B-tree)
- STXXL implementations outperform or compete with the opponents
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)
Engineering Algorithms for Large Graphs
The core of many other practical graph algorithms
E.g.: Visualization of query graph structure at Ask.com

Challenge
Can we compute Minimum Spanning Trees (Forests) for really huge graphs?

Dementiev, Sanders, Schultes, and Sibeyn
Engineering an External Memory Minimum Spanning Tree Algorithm.
TCS 2004: 3rd IFIP International Conference on Theoretical Computer Science
A Practical Approach

Sketch of the algorithm

1. Reduce the node set $V$ merging nodes and finding some MST edges until $|V| = O(M)$

2. Run Kruskal’s algorithm keeping forests in internal memory (Union-Find)

Two implementation variants

- Node reduction using `stxxl::priority_queue`: very simple, only 12 lines of C++/STXXL code, CPU-bound
- Bucket version: based on `stxxl::stacks`, linear internal work

Results

- Computed MSTs for 100 GByte graphs in 8 hours on a PC
- Only 2–5 times slower than a good internal algorithm
A Practical Approach

Sketch of the algorithm

1. Reduce the node set $V$ merging nodes and finding some MST edges until $|V| = O(M)$

2. Run Kruskal’s algorithm keeping forests in internal memory (Union-Find)

Two implementation variants

- **Node reduction using** `stxxl::priority_queue`: very simple, only 12 lines of C++/STXXL code, CPU-bound
- **Bucket version**: based on `stxxl::stacks`, linear internal work

Results

- Computed MSTs for 100 GByte graphs in 8 hours on a PC
- Only 2–5 times slower than a good internal algorithm
A Practical Approach

Sketch of the algorithm

1. Reduce the node set $V$ merging nodes and finding some MST edges until $|V| = O(M)$

2. Run Kruskal’s algorithm keeping forests in internal memory (Union-Find)

Two implementation variants

- Node reduction using `stxxl::priority_queue`: very simple, only 12 lines of C++/STXXL code, CPU-bound
- Bucket version: based on `stxxl::stacks`, linear internal work

Results

- Computed MSTs for 100 GByte graphs in 8 hours on a PC
- Only 2–5 times slower than a good internal algorithm
Study two I/O-efficient BFS algorithms (MunagalaRanade and MehlhornMeyer)

Use STXXL pipelining

Results

- BFS of a real huge WWW crawl graph (130 \cdot 10^6 nodes, 1.4 \cdot 10^9 edges) in about 2 hours on a PC
Study two I/O-efficient BFS algorithms (MunagalaRanade and MehlhornMeyer)

Use STXXL pipelining

Results

- BFS of a real huge WWW crawl graph (130 \cdot 10^6 nodes, 1.4 \cdot 10^9 edges) in about 2 hours on a PC
I/O-Efficient Breadth First Search

Ajwani, Dementiev and Meyer
A Computational Study of External-Memory BFS Algorithms.

\textit{SODA 2006, ACM Symposium on Discrete Algorithms}

Study two I/O-efficient BFS algorithms (MunagalaRanade and MehlhornMeyer)

Use \textsc{STXXL} pipelining

Results

- BFS of a real huge \textit{WWW crawl} graph (130 $\cdot$ 10\(^6\) nodes, 1.4 $\cdot$ 10\(^9\) edges)
  in about 2 hours on a PC
Engineering Algorithms for Large Graphs

In my PhD thesis

- Maximal Independent Set
- Connected Components and Spanning Trees
- Listing All Triangles
- (Heuristics for) Graph Coloring
Engineering Large Suffix Array Construction
Engineering Large Suffix Array Construction

Suffix Array: \( SA[i] \) is the starting pos of the \( i \)-th smallest suffix of input \( S \)

Example:
\[
S = [b, a, n, a, n, a]
SA = [5, 3, 1, 0, 4, 2]
\]

Applications: full-text index, bzip2 compression

Our Work
- Design, implement, evaluate several new I/O-efficient algorithms
- Apply pipelining to external memory suffix array construction

Dementiev, Kärkkäinen, Mehnert, Sanders
Better External Memory Suffix Array Construction.

\textit{JEA and ALENEX05: Algorithm Engineering and Experiments}
Engineering Large Suffix Array Construction

Suffix Array: $SA[i]$ is the starting pos of the $i$-th smallest suffix of input $S$

Example:
$S = [b, a, n, a, n, a]$
$SA = [5, 3, 1, 0, 4, 2]$

Applications: full-text index, bzip2 compression

Our Work

- Design, implement, evaluate several new I/O-efficient algorithms
- Apply pipelining to external memory suffix array construction

Dementiev, Kärkkäinen, Mehnert, Sanders
Better External Memory Suffix Array Construction.

*JEA and ALENEX05: Algorithm Engineering and Experiments*
### Considered algorithms

- Doubling, \( a \)-Tupling (quadrupling), doubling+discarding, quadrupling+discarding, difference cover (DC3 aka Skew, I/O-optimal)

### Input instances: random and real world

- Concatenation of a random string (make heuristics look bad)
- Gutenberg text collection (\( \approx 3 \) GBytes)
- Human genome (small alphabet, \( \approx 3 \) GBytes)
- HTML (text + tags, \( \approx 4 \) GBytes)
- (C++) source code (\( \approx 500 \) MBytes)
Results

- Pipelining saves a factor of 3 in I/O volume
  ⇒ a speedup of 1.9-2.4 for $D = 1$
- **Optimal** DC3 outperforms all opponents on all inputs
- Suffix array of a 4 GByte input can be computed in a few hours on a PC with a small main memory
  ⇒ Very price-efficient
Porting Algorithms to External Memory

Replace few underlying non-I/O-efficient algorithms by corresponding I/O-efficient versions

We have applied this technique obtaining algorithms for

- Bipartiteness test (aka 2-coloring): $O(sort(|E| + |V|))$ I/Os
- 5-Coloring Planar Graphs: $O(sort(|E| + |V|))$ I/Os
- Finding 1/2-Approximation of Maximum Weighted Matching: $O(sort(|E| + |V|))$ I/Os
- Finding Perfect Matchings in Bipartite Multigraphs: $O(sort(|E| + |V|) \log_2(|E|))$ I/Os

⇒ Can be easily implemented with STXXL
Summary

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
Summary

**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
Summary

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
Summary

Engineering from the bottom to the top:

- many disks ➞ CPU-bound ➞ look at internal algorithms,
  RAID-0 ➞ 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
Conclusion

The STXXL library

- high-performance (parallel disks, pipelining, overlapping of I/O and computation)
- easy to use (STL-compatible)

STXXL applications: solve very large problem instances externally using a low cost hardware in record time

⇒ Price-efficient

Outlook

- Possible efficiency improvements:
  - pipelining+overlapping
  - parallel processing (Multi-Core STL)
  - pipelining+task-based parallelism

- Submit to the BOOST libraries
Conclusion

The STXXL library

- **high-performance** (parallel disks, pipelining, overlapping of I/O and computation)
- easy to use (STL-compatible)

**STXXL applications:** solve **very large** problem instances externally using a **low cost** hardware in record time

⇒ Price-efficient

Outlook

- **Possible efficiency improvements:**
  - pipelining+overlapping
  - parallel processing (Multi-Core STL)
  - pipelining+task-based parallelism
- Submit to the **BOOST libraries**
Thank you for your attention!