Algorithms for dealing with massive data

Luciana S. Buriol and Marcus Ritt

Computer Science Department – Federal University of Rio Grande do Sul
Porto Alegre, Brazil
Outline of the talk

- Algorithms models for dealing with massive datasets
  - External Memory Algorithms: Motivation, Model, Analysis, Examples.
  - Data Stream Algorithms: Motivation.
Introduction
Dealing with Massive Datasets
External Memory Algorithms
Cache-Efficient Algorithms
Data Stream Algorithms
Computing with preprocessing
Concluding Remarks

Massive Datasets

- The speed of computers and the size of memories are not growing at the same rate than the amount of data to be processed
- Large datasets appear in all kind of areas and applications
  - satellite images (Google Earth and Microsoft TerraServer)
  - gene expression in bioinformatics
  - webgraph mining
  - monitoring internet traffic
  - NASA’s Earth Observing System project produces petabytes of data per year

\(^1\)http://eospso.gsfc.nasa.gov/
The speed of computers and the size of memories are not growing at the same rate than the amount of data to be processed.

Large datasets appear in all kind of areas and applications:
- satellite images (Google Earth and Microsoft TerraServer)
- gene expression in bioinformatics
- webgraph mining
- monitoring internet traffic
- NASA’s Earth Observing System project produces petabytes of data per year\(^1\)

If the data cannot be processed, it is useless!

\(^1\)http://eospso.gsfc.nasa.gov/
Memory capacities and transfer rate

- 60000 MB/s
- 7500 MB/s
- 3000 MB/s
- 80 MB/s

- 1 KB
- 16 MB
- 16 GB
- 100 GB
Memory Hierarchy

- Registers: multiple ports/several accesses in parallel
- First Level L1 Cache (32KB)
- Second Level L2 Cache: communication with L1 via block sizes of 16-32 bytes (4096 KB)
- Third Level L3 Cache: static RAM - SRAM fast/costly (recent use)
- Main memory: dynamic RAM cells
- External Memory: disks have cheap and non volatile memory
Algorithm Models for Dealing with Massive Datasets

- Data Stream Algorithms
- External Memory Algorithms
- Cache-Efficient Algorithms
Consider applications where
- the data does not fit into main memory; but
- the algorithm can process the whole data off-line
Consider applications where
- the data \textit{does not fit} into main memory; but
- the algorithm can process the whole data \textit{off-line}

In this case \textbf{External Memory Algorithms} are a suitable choice

Since accessing an external device is much more time demanding than accessing main memory, External Memory Algorithms optimize the use of I/Os
A disk access is up to 1,000,000 slower than a RAM access
A disk transfer rate is about 50-100 MB/s
A disk access is up to 1,000,000 slower than a RAM access
A disk transfer rate is about 50-100 MB/s
Clearly it makes sense to process data in large chunks
External Memory Algorithms: The Two-Level I/O Model

Parallel disk model (Vitter, Shriver 1994)

- $P$ processors share $D$ disks
- Internal memory of $M$ items ($M/P$ per processor)
- Each block has $B$ items
- Problem size of $N$ items
The I/O complexity of an algorithm is the number of blocks transferred between disk and memory. The goal is to exploit locality in order to reduce the I/O costs. The read step moves B elements from external to internal memory, and the write step moves B elements from internal to external memory. Block size: at least 512 bytes (imposed by hardware), but usually it is used at least 8KB.
External Memory Algorithms (EMA)

- The I/O complexity of an algorithm is the number of blocks transferred between disk and memory.
- The goal is to exploit locality in order to reduce the I/O costs.
- Read step: moves B elements from external to internal memory.
- Write step: moves B elements from internal to external memory.
- Block size: at least 512 bytes (imposed by hardware), but usually it is used at least 8KB.
- Performance measures:
  - I/O complexity: number of I/Os the algorithm executes.
  - Space complexity: the maximum disk space (no. of blocks) active at any one time.
  - Time Complexity: internal processing time.
Principles

- **Time complexity**: the time complexity should be comparable with the best internal memory algorithms
- **Spatial locality**: when a block is accessed, it must contain as much useful data as possible
- **Temporal locality**: once data is in main memory, as much as possible should be processed
Some fundamental I/O operations and bounds (considering D=1):

- **Scan** $N$ items: $\text{scan}(N) = \Theta\left(\frac{N}{B}\right)$
- **Search** one in $N$ items: $\text{search}(N) = \Omega\left(\log_B \frac{N}{M}\right)$
- **Sort** $N$ items: $\text{sort}(N) = \Theta\left(\frac{N}{B} \log\frac{M}{B}(\frac{N}{B})\right)$
External Memory Algorithms: Basic Data Structures

- Stacks (last-in-first-out): $\frac{1}{B}$ I/Os for insertions and deletions using two buffers of size $B$
- Queue (first-in-first-out): $\frac{1}{B}$ I/Os for insertions and deletions using two buffers of size $B$
- Linked Lists:
  - One possibility: $\frac{N}{B}$ for traversing, insertion and deletion
  - A better possibility: $3/2 \frac{N}{B}$ for traversing, but insertions and deletions are constant in amortized time
I/O Complexity for Other Problems

- Matrix multiplication of a KxK matrix: $\Theta\left(\frac{K^3}{\min(K, \sqrt{MB})}\right)$
- Counting triangles in graphs: $O(m\text{Scan}(\sqrt{m}) + \text{Sort}(m))$
Connected Component and Minimum Spanning Tree algorithms

- **Connected components**
  - K. Munagala and A. Ranade, 1999: $O(\text{sort}(E) \cdot \log \log \frac{VB}{E})$
  - Abello et al., 2002: $O(\text{sort}(E) + \frac{E}{V} \text{sort}(V) \log_2 \frac{V}{M})$
  - J. Sibeyn and U. Meyer, 2004: $O(\text{sort}(m) \log \frac{n}{M})$

- **Minimum Spanning Tree**
  - Chiang et al. 1995: $O(\text{sort}(E) \cdot \log \frac{V}{M})$
  - V. Kumar and E. Schwabe, 1996:
    $O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$
  - Lars Arge, Gerth Brodal and Laura Toma, 2000: $O(V + \text{sort}(E))$
  - R. Dementiev, P. Sanders, D. Schultes 2004: $O(\frac{m'}{m} \text{sort}(m))$
External Memory Algorithms (EMA)

- **Functional** EMA: Once the output data is written, it remains unchanged.
- External vs. **Semi-External** Memory Algorithms.
  - In graph problems when $V \leq M$ but $E > M$
  - Always faster than EMA!
References and hints

- **STXXL** ([http://stxxl.sourceforge.net](http://stxxl.sourceforge.net)): Library with the main data structures and common algorithms implemented.
- To count the number of I/Os: STXXL or `iostat`.

Cache-Efficient Algorithms

- External Memory Algorithms process data off-line, but the data does not fit into main memory.
- Consider applications where:
  - the data *fits* into main memory; and
  - the algorithm can be speedup if the use of cache is optimized.

Luciana S. Buriol and Marcus Ritt
Algorithms for dealing with massive data
Cache-Efficient Algorithms

- External Memory Algorithms process data off-line, but the data does not fit into main memory.
- Consider applications where
  - the data fits into main memory; and
  - the algorithm can be speedup if the use of cache is optimized.
- In this case Cache-Efficient Algorithms are a suitable choice.
- Cache-Efficient algorithms minimize the number of cache misses.
Cache-Efficient Algorithms: Model

- Ideal-cache data model (Prokop 1999)

- Fully associative cache of $Z$ words
- Each cache line contains $L$ words
- Cache usually tall, i.e. $Z = \Omega(L^2)$.
- Optimal replacement strategy (evict line with latest future reference)
Cache-Efficient Algorithms

- Corresponds to two-level external memory I/O model with $M = Z$ and $B = L$.

- A *cache-oblivious* algorithm does not use knowledge of $Z$ and $L$; otherwise its *cache-conscious*.

- A cache-oblivious algorithm is portable, and adapts to all levels of a multi-level memory hierarchy.

- In a cache-oblivious algorithm, whenever a block is brought into cache it contains as much useful data as possible.

- Performance measures:
  - number of cache misses
  - Time complexity
Longest Common Subsequence - LCS

- Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$
- We can find the longest common subsequence by dynamic programming

$$c_{ij} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
   c_{i-1,j-1} + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\
   \max(c_{i-1,j}, c_{i,j-1}) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j
\end{cases}$$

- Example: The LCS of $X = AC TGCA TGC$ and $Y = ATGC TA$ is $Z = ATGCA$
Longest Common Subsequence - LCS

- Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$
- Find the longest common subsequence
- Example: The LCS of $X = ACTGCA TGC$ and $Y = ATGCTA$ is $Z = ATGCA$
Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$

We can find the longest common subsequence by dynamic programming

$$c_{ij} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
 c_{i-1,j-1} + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\
 \max(c_{i-1,j}, c_{i,j-1}) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j
\end{cases}$$

Example: $X = \text{ACTGCATGC}$ and $Y = \emptyset$
Longest Common Subsequence - LCS

- Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$
- We can find the longest common subsequence by dynamic programming

$$c_{ij} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\ c_{i-1,j-1} + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\
\ max(c_{i-1,j}, c_{i,j-1}) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j
\end{cases}$$

- Example: $X = ACTGCATG C$ and $Y = CTAGCTA C$
Longest Common Subsequence - LCS

- Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$
- We can find the longest common subsequence by dynamic programming

$$c_{ij} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max(c_{i-1,j-1}, c_{i,j-1}) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j \\
c_{i-1,j-1} + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j 
\end{cases}$$

- Example: $X = ACTGCATG$ and $Y = CTAGCT$
Longest Common Subsequence: LCS

- Given sequences \( X = x_1 \cdots x_n \) and \( Y = y_1 \cdots y_m \)
- We can find the longest common subsequence by dynamic programming

\[
c_{ij} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max(c_{i-1,j}, c_{i,j-1}) + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\
\max(c_{i-1,j}, c_{i,j-1}) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j 
\end{cases}
\]

- Straightforward implementation: time and memory \( O(mn) \) and \( O(mn/L) \) cache misses.
- Hirschberg’s algorithm (1975) reduces memory to \( O(\min(m, n)) \)
Cache-Oblivious Algorithms: Longest Common Subsequence

- A cache-oblivious solution can reduce cache misses to $O(\frac{mn}{LZ})$ (Chowdhury, Ramanchandran, 2006)

- Given any submatrix, we can propagate input to output boundary; by divide-and-conquer, tiles fit into cache

- 2-6 times faster

- To recover the subsequence, trace it recursively
Our Project in Cache-Efficient Algorithms

- Longest Common Subsequence (Chowdhury, Ramanchandran, 2006): 2x faster
- Floyd-Warshall Algorithm (J. Park, M. Penner and V. Prasanna, 2002): 10x faster
Our Project in Cache-Efficient Algorithms

- Longest Common Subsequence (Chowdhury, Ramanchandran, 2006): 2x faster
- Floyd-Warshall Algorithm (J. Park, M. Penner and V. Prasanna, 2002): 10x faster
- Cache-oblivious algorithm for the Knapsack problem
- Cache-oblivious algorithm for matrix multiplication
References and hints

- To count the number of cache misses: *cache-grind's* Valgrind

References:
