Part 2: External Memory and Cache Oblivious Algorithms

CR10: Data Aware Algorithms

September 25, 2019
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures
- External Memory Model
- Merge Sort
- Lower Bound on Sorting
- Permuting
- Searching and B-Trees
- Matrix-Matrix Multiplication
Ideal Cache Model

Properties of real cache:

- Memory/cache divided into blocks (or lines) of size $B$
- Limited associativity:
  - each block of memory belongs to a cluster (usually computed as $address \% M$)
  - at most $c$ blocks of a cluster can be stored in cache at once ($c$-way associative)
  - Trade-off between hit rate and time for searching the cache
- Block replacement policy: LRU (also LFU or FIFO)

Ideal cache model:

- Fully associative
  - $c = \infty$, blocks can be store everywhere in the cache
- Optimal replacement policy
  - Belady’s rule: evict block whose next access is furthest
- Tall cache: $M/B \gg B$ ($M = \Theta(B^2)$)
Ideal Cache Model

Properties of real cache:

- Memory/cache divided into blocks (or lines) of size $B$
- Limited associativity:
  - each block of memory belongs to a cluster (usually computed as $address \% M$)
  - at most $c$ blocks of a cluster can be stored in cache at once ($c$-way associative)
  - Trade-off between hit rate and time for searching the cache
- Block replacement policy: LRU (also LFU or FIFO)

Ideal cache model:

- Fully associative
  - $c = \infty$, blocks can be stored everywhere in the cache
- Optimal replacement policy
  - Belady’s rule: evict block whose next access is furthest
- Tall cache: $M/B \gg B$  ($M = \Theta(B^2)$)
**Lemma (Sleator and Tarjan, 1985).**

For any sequence $s$:

$$T_{LRU}(s) \leq \frac{k_{LRU}}{k_{LRU} + 1 - k_{OPT}} T_{OPT}(s) + k_{OPT}$$

- $T_A(s)$: nb of cache miss for the optimal replacement policy $A$ with cache size $k_A$
- OPT: optimal (offline) replacement policy (Belady’s rule)
- LRU, A: online algorithms (no knowledge on future requests)
- $k_A, k_{LRU} \leq k_{OPT}$

**Theorem (Bound on competitive ratio).**

Assume there exists $a$ and $b$ such that $T_A(s) \leq aT_{OPT}(s) + b$ for all $s$, then $a \geq k_A/(k_A + 1 - k_{OPT})$. 
**LRU vs. Optimal Replacement Policy**

**Lemma (Sleator and Tarjan, 1985).**
For any sequence $s$:

$$T_{LRU}(s) \leq \frac{k_{LRU}}{k_{LRU} + 1 - k_{OPT}} T_{OPT}(s) + k_{OPT}$$

- $T_A(s)$: nb of cache miss for the optimal replacement policy $A$ with cache size $k_A$
- OPT: optimal (offline) replacement policy (Belady’s rule)
- LRU, A: online algorithms (no knowledge on future requests)
- $k_A, k_{LRU} \leq k_{OPT}$

**Theorem (Bound on competitive ratio).**
Assume there exists $a$ and $b$ such that $T_A(s) \leq a T_{OPT}(s) + b$ for all $s$, then $a \geq k_A/(k_A + 1 - k_{OPT})$. 
LRU competitive ratio – Proof

- Consider any subsequence $t$ of $s$, such that $C_{LRU}(t) \leq k_{LRU}$ ($t$ should not include first request)
- Let $p$ be the block request right after $t$ in $s$
- If LRU loads twice the same block in $s$, then $C_{LRU}(t) \geq k_{LRU} + 1$ (contradiction)
- Same if LRU loads $p$ during $t$
- Thus on $t$, LRU loads $C_{LRU}(t)$ different blocks, different from $p$
- When starting $t$, OPT has $p$ in cache
- On $t$, OPT must load at least $C_{LRU}(t) - k_{OPT} + 1$
- Partition $s$ into $s_0, s_1, \ldots, s_n$ s.t.
  \[ C_{LRU}(s_0) \leq k_{LRU} \quad \text{and} \quad C_{LRU}(s_i) = k_{LRU} \quad \text{for} \ i > 1 \]
- On $s_0$, $C_{OPT}(s_0) \geq C_{LRU}(s_0) - k_{OPT}$
- In total for LRU: $C_{LRU} = C_{LRU}(s_0) + nk_{LRU}$
- In total for OPT: $C_{OPT} \geq C_{LRU}(s_0) - k_{OPT} + n(k_{LRU} - k_{OPT} + 1)$
Let $S_A^{\text{init}}$ (resp. $S_{\text{OPT}}^{\text{init}}$) the set of blocks initially in A’s cache (resp. OPT’s cache).

Consider the block request sequence made of two steps:

- $S_1$: $k_A - k_{\text{OPT}} + 1$ (new) blocks not in $S_A^{\text{init}} \cup S_{\text{OPT}}^{\text{init}}$
- $S_2$: $k_{\text{OPT}} - 1$ blocks s.t. then next block is always in $(S^{\text{init}}_{\text{OPT}} \cup S_1) \setminus S_A$

NB: step 2 is possible since $|S_{\text{OPT}}^{\text{init}} \cup S_1| = k_A + 1$

- A loads one block for each request of both steps: $k_A$ loads
- OPT loads one block only in $S_1$: $k_A - k_{\text{OPT}} + 1$ loads
Justification of the Ideal Cache Model

Theorem (Frigo et al, 1999).
If an algorithm makes $T$ memory transfers with a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ transfers with cache size $M$ with LRU.

Definition (Regularity condition).
Let $T(M)$ be the number of memory transfers for an algorithm with cache of size $M$ and an optimal replacement policy. The regularity condition of the algorithm writes

$$T(M) = O(T(M/2))$$

Corollary
If an algorithm follows the regularity condition and makes $T(M)$ transfers with cache size $M$ and an optimal replacement policy, it makes $\Theta(T(M))$ memory transfers with LRU.
Justification of the Ideal Cache Model

Theorem (Frigo et al, 1999).
If an algorithm makes $T$ memory transfers with a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ transfers with cache size $M$ with LRU.

Definition (Regularity condition).
Let $T(M)$ be the number of memory transfers for an algorithm with cache of size $M$ and an optimal replacement policy. The regularity condition of the algorithm writes

$$T(M) = O(T(M/2))$$

Corollary
If an algorithm follows the regularity condition and makes $T(M)$ transfers with cache size $M$ and an optimal replacement policy, it makes $\Theta(T(M))$ memory transfers with LRU.
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures

- External Memory Model
- Merge Sort
- Lower Bound on Sorting
- Permuting
- Searching and B-Trees
- Matrix-Matrix Multiplication
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures
  External Memory Model
  Merge Sort
  Lower Bound on Sorting
  Permuting
  Searching and B-Trees
  Matrix-Matrix Multiplication
External Memory Model

Model:
- **External** Memory (or disk): storage
- **Internal** Memory (or cache): for computations, size $M$
- Ideal cache model for transfers: blocks of size $B$
- Input size: $N$
- Lower-case letters: in number of blocks
  \[ n = \frac{N}{B}, \quad m = \frac{M}{B} \]

**Theorem.**
Scanning $N$ elements stored in a contiguous segment of memory costs at most $\lceil \frac{N}{B} \rceil + 1$ memory transfers.
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures

- External Memory Model
- Merge Sort
- Lower Bound on Sorting
- Permuting
- Searching and B-Trees
- Matrix-Matrix Multiplication
Merge Sort in External Memory

Standard Merge Sort: Divide and Conquer

1. Recursively split the array (size $N$) in two, until reaching size 1
2. Merge two sorted arrays of size $L$ into one of size $2L$
   requires $2L$ comparisons

In total: log $N$ levels, $N$ comparisons in each level

Adaptation for External Memory: Phase 1

- Partition the array in $N/M$ chunks of size $M$
- Sort each chunks independently (runs)
- Block transfers: $2M/B$ per chunk, $2N/B$ in total
- Number of comparisons: $M \log M$ per chunk, $N \log M$ in total
Merge Sort in External Memory

Standard Merge Sort: Divide and Conquer
1. Recursively split the array (size $N$) in two, until reaching size 1
2. Merge two sorted arrays of size $L$ into one of size $2L$
   requires $2L$ comparisons
In total: $\log N$ levels, $N$ comparisons in each level

Adaptation for External Memory: Phase 1
- Partition the array in $N/M$ chunks of size $M$
- Sort each chunks independently (→ runs)
- Block transfers: $2M/B$ per chunk, $2N/B$ in total
- Number of comparisons: $M \log M$ per chunk, $N \log M$ in total
Two-Way Merge in External Memory

Phase 2:
Merge two runs $R$ and $S$ of size $L \rightarrow$ one run $T$ of size $2L$

1. Load first blocks $\hat{R}$ (and $\hat{S}$) of $R$ (and $S$)
2. Allocate first block $\hat{T}$ of $T$
3. While $R$ and $S$ both not exhausted
   (a) Merge as much $\hat{R}$ and $\hat{S}$ into $\hat{T}$ as possible
   (b) If $\hat{R}$ (or $\hat{S}$) gets empty, load new block of $R$ (or $S$)
   (c) If $\hat{T}$ gets full, flush it into $T$
4. Transfer remaining items of $R$ (or $S$) in $T$

- Internal memory usage: 3 blocks
- Block transfers: $2L/B$ reads + $2L/B$ writes = $4L/B$
- Number of comparisons: $2L$
Total complexity of Two-Way Merge Sort

Analysis at each level:
► At level $k$: runs of size $2^k M$ (nb: $N / (2^k M)$)
► Merge to reach levels $k = 1 \ldots \log_2 N / M$
► Block transfers at level $k$: $2^{k+1} M / B \times N / (2^k M) = 2N / B$
► Number of comparisons: $N$

Total complexity of phases 1+2:
► Block transfers: $2N / B (1 + \log_2 N / B) = O(N / B \log_2 N / B)$
► Number of comparisons: $N \log M + N \log_2 N / M = N \log N$

but we use only 3 blocks of internal memory 😞
Optimization: $K$-Way Merge Sort

- Consider $K$ input runs at each merge step
- Efficient merging, e.g.: MinHeap data structure
  insert, extract: $O(\log K)$
- Complexity of merging $K$ runs of length $L$: $KL \log K$
- Block transfers: no change ($2KL/B$)

Total complexity of merging:

- Block transfers: $\log_K N/M$ steps $\rightarrow 2N/B \log_K N/M$
- Computations: $N \log K$ per step $\rightarrow N \log K \times \log_K N/M$
  $= N \log_2 N/M$ (id.)

Maximize $K$ to reduce transfers:

- $(K + 1)B = M$ ($K$ input blocks + 1 output block)
- Block transfers: $O \left( \frac{N}{B} \log_{M/B} \frac{N}{M} \right)$
- NB: $\log_{M/B} N/M = \log_{M/B} N/B - 1$
- Block transfers: $O \left( \frac{N}{B} \log_{M/B} \frac{N}{B} \right) = O(n \log_m n)$
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures

- External Memory Model
- Merge Sort
- Lower Bound on Sorting
- Permuting
- Searching and B-Trees
- Matrix-Matrix Multiplication
Lower Bound on Sorting

- Comparison based model: elements compared when in internal memory
- Inputs of new blocks give new information (but not outputs)
- $S_t$: number of permutations consistent with knowledge after reading $t$ blocks of inputs
- At the beginning: $S_0 = N!$ possible orderings (no information)
- After reading one block: new information (answer) *how the elements read are ordered among themselves and among the $M$ elements in memory?*
- Assume $X$ possible answers after one read, then
  \[ S_{t+1} \geq S_t / X \]

Proof:
- Partition of the $S_t$ orderings into $X$ parts
- There exists a part of size at least $S_t / X$, that is an answer with at least $S_t / X$ compatible orderings
Lower Bound on Sorting

Bound the number of possible orderings:

(i) When reading a block already seen: $X = \binom{M}{B}

(ii) When reading a new block (never seen): $X = \binom{M}{B} B!

NB: at most $N/B$ new blocks (case (i))

From $S_0 = N!$ and $S_{t+1} \geq S_t / X$, we get:

$$S_t \geq \frac{N!}{\binom{M}{B}^t (B!)^{N/B}}$$

$S_t = 1$ for final step

Stirling’s formula gives: $\log x! \approx x \log x$ and $\log \binom{x}{y} \approx x \log x/y$

$$t = \Omega \left( \frac{N}{B} \log \frac{M}{B} \frac{N}{B} \right)$$
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures

  External Memory Model
  Merge Sort
  Lower Bound on Sorting
  Permuting
  Searching and B-Trees
  Matrix-Matrix Multiplication
Permuting

Inputs:
- $N$ elements together with their final position:
  (a,3) (b,2) (c,1) (d,4) $\rightarrow$ c,b,a,d

Two simple strategies:
- Place each element at its final position, one after the other
  \( \text{I/O cost: } \Theta(N) \quad (\text{cmp cost: } O(N)) \)
- Sort elements based on final position
  \( \text{I/O cost: } \Theta(SORT(N)) = \Theta(N/B \log_{M/B} N/B) \)
  \( (\text{cmp cost: } O(N \log N)) \)

Lower-bound:
- Using similar argument, one may prove that the
  \( \text{I/O complexity is bounded by } \Theta(\min(SORT(N), N)) \)
- NB: generally, \( SORT(N) \ll N \)
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures
  External Memory Model
  Merge Sort
  Lower Bound on Sorting
  Permuting

Searching and B-Trees
Matrix-Matrix Multiplication
Problem: Search for a particular element in a huge dataset
Solution: Search tree with large degree ($\approx B$)

**Definition (B-tree with minimum degree $d$).**
Search tree such that:
- Each node (except the root) has at least $d$ children
- Each node has at most $2d$ children
- Node with $k$ children has $k - 1$ keys separating the children
- All leaves have the same depth

Proposed by Bayer and McCreigh (1972)
Search and Insertion in B-Trees

Usually, we require that $d = O(B)$

**Lemma.**

Searching in a B-Tree requires $O(\log_d N)$ I/Os.

Insertion algorithm:

1. If root node is full ($2d$ children), split it:
   (a) Find median key, send it to the father $f$
       (if any, otherwise it becomes the new root)
   (b) Keys and subtrees $<\text{median key}$ → new left subtree of $f$
   (c) Keys and subtrees $>\text{median key}$ → new right subtree $f$

2. If root node = leaf, insert new key

3. Otherwise, find correct subtree $s$, insert recursively in $s$

NB: height changes only when root is split → balanced tree

Number of transfers: $O(h)$
Suppression in B-Trees

Suppression algorithm of $k$ from a tree with at least $d$ keys:

- If tree=leaf, straightforward
- If $k =$ key of root node:
  - If subtree $s$ immediately left of $k$ has $d$ keys, remove maximum element $k'$ of $s$, replace $k$ by $k'$
  - Same on right subtree (with minimum element)
  - Otherwise (both neighbor subtrees have $d-1$ keys): remove $k$ and merge these neighbor subtrees
- If $k$ is in a subtree, find the correct subtree $T$
- If $T$ has only $d-1$ keys:
  - Try to steal one key from a neighbor of $T$ with at least $d$ keys
  - Otherwise merge $T$ with one of its neighbors
- Call recursively on the correct subtree

Number of block transfers: $O(h)$
Usage of B-Trees

Widely used in large database and filesystems (SQL, ext4, Apple File System, NTFS)

Variants:

- **B+ Trees**: store data only on leaves
  increase degree → reduce height
  add pointer from leaf to next one to speedup sequential access

- **B* Trees**: better balance of internal node
  (max size: \(2b \rightarrow 3b/2\), nodes at least 2/3 full)
  - When 2 siblings full: split into 3 nodes
  - Pospone splitting: shift keys to neighbors if possible
Searching Lower Bound

Theorem.
Searching for an element among \( N \) elements in external memory requires \( \Theta(\log_{B+1} N) \) block transfers.

Proof:
- Adversary argument
- Total order of \( N \) elements known to the algorithm
- Let \( C_t \) be the number of candidates after \( t \) reads \( (C_0 = N) \)
- When a block of size \( B \) is read, the \( C_t - B \) remaining elements are distributed into \( B + 1 \) parts, one of them has at least \( (C_t - B)/(B + 1) \) elements.
- By induction, \( C_t \geq N/(B + 1)^t - (B + 1)/B \)

If memory initially full, \( C_0 = (N - M)/(M + 1) \), lower bound: \( \Theta(\log_{B+1} N/M) \)
Outline

Ideal Cache Model

External Memory Algorithms and Data Structures
  External Memory Model
  Merge Sort
  Lower Bound on Sorting
  Permuting
  Searching and B-Trees
  Matrix-Matrix Multiplication
Matrix-Matrix Multiplication

The I/O bound on matrix multiplication seen previously is extended:

**Theorem.**
The number of block transfers for multiplying two $N \times N$ matrices is $\Theta(N^3/(B\sqrt{M}))$ when $M < N^2$.

Blocked algorithms naturally reduces block transfers.
## Summary: External Memory Bounds

<table>
<thead>
<tr>
<th></th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanning</td>
<td>$N$</td>
<td>$N/B$</td>
</tr>
<tr>
<td>Sorting</td>
<td>$N \log_2 N$</td>
<td>$N/B \log_{M/B} N/B$</td>
</tr>
<tr>
<td>Permuting</td>
<td>$N$</td>
<td>$\min(N, N/B \log_{M/B} N/B)$</td>
</tr>
<tr>
<td>Searching</td>
<td>$\log_2 N$</td>
<td>$\log_B N$</td>
</tr>
<tr>
<td>Matrix Mult.</td>
<td>$N^3$</td>
<td>$N^3/(B\sqrt{M})$</td>
</tr>
</tbody>
</table>

**Notes:**
- Linear I/O: $O(N/B)$
- Permuting is not linear
- $B$ is an important factor: $N/B < N/B \log_{M/B} N/B \ll N$
- Search tree cannot lead to optimal sort