Cache-Oblivious Index for Approximate String Matching

CPM 2007

Siu-Lung Tam
The University of Hong Kong

Joint work with:
Wing-Kai Hon Tak-Wah Lam
Rahul Shah Jeffrey Scott Vitter

2007-07-09 11:35 -0400

Indexing for approximate string matching

Problem

Index a text $T[1..n]$ over an alphabet Σ w.r.t. a constant k.
For each pattern $P[1..m]$, search for all i such that $T[i..j]$ is a k-error match, i.e. $\exists\, j \geq i \;\; dist(P, T[i..j]) \leq k$.

Indexing for approximate string matching

Problem

Index a text $T[1..n]$ over an alphabet Σ w.r.t. a constant k. For each pattern $P[1..m]$, search for all i such that $T[i..j]$ is a k-error match, i.e. $\exists\, j \geq i \;\; dist(P, T[i..j]) \leq k$.

- In the classical setting, performance is measured by
  - (i) index size, and
  - (ii) searching time, which can depend on
    - text size n,
    - alphabet size $|\Sigma|$,
    - number of errors k,
    - pattern length m,
    - number of matches occ
- In this paper, we consider the cache-oblivious setting
  - which considers "I/O complexity" instead of time complexity

External memory model of computation

- Characteristics of external memory (EM)
  - Extremely long access time and seek time
  - More efficient to access contiguous blocks of data at a time
- The memory hierarchy consists of main memory and external memory
- Disk page: contiguous block of B words in external memory
- One “I/O” allows
  - Reading a disk page to main memory
  - Writing from main memory to a disk page

External memory model of computation

- Characteristics of external memory (EM)
  - Extremely long access time and seek time
  - More efficient to access contiguous blocks of data at a time
- The memory hierarchy consists of main memory and external memory
- Disk page: contiguous block of B words in external memory
- One "I/O" allows
  - Reading a disk page to main memory
  - Writing from main memory to a disk page
- Cache oblivious data structure
  - The data structure does not know the value of B
  - B is only used during analysis of the algorithms
  - This implies the data structure is EM-efficient for all B

EM data structures to classical problems

- 1-D range searching for n ordered items
  - Balanced binary search tree: $O(\log n + occ)$ time
  - B-tree: $O(\log_B n + \frac{occ}{B})$ I/Os
  - Cache Oblivious B-tree [FOCS00]: $O(\log_B n + \frac{occ}{B})$ I/Os

EM data structures to classical problems

- 1-D range searching for n ordered items
  - Balanced binary search tree: $O(\log n + occ)$ time
  - B-tree: $O(\log_B n + \frac{occ}{B})$ I/Os
  - Cache Oblivious B-tree [FOCS00]: $O(\log_B n + \frac{occ}{B})$ I/Os
- Predecessor query for k integers less than n in $O(k)$ space
  - *y*-fast trie: $O(\log \log n)$ time
  - *y*-fast trie in van Emde Boas (vEB) layout: $O(\log \log_B n)$ I/Os

EM data structures to classical problems

- 1-D range searching for n ordered items
  - Balanced binary search tree: $O(\log n + occ)$ time
  - B-tree: $O(\log_B n + \frac{occ}{B})$ I/Os
  - Cache Oblivious B-tree [FOCS00]: $O(\log_B n + \frac{occ}{B})$ I/Os
- Predecessor query for k integers less than n in $O(k)$ space
  - *y*-fast trie: $O(\log \log n)$ time
  - *y*-fast trie in van Emde Boas (vEB) layout: $O(\log \log_B n)$ I/Os
- Exact substring matching for a text of length n
  - Suffix tree: $O(|P| + occ)$ time
  - Suffix array: $O(|P| + occ + \log n)$ time
  - String B-tree [STOC95]: $O(\frac{|P|+occ}{B} + \log_B n)$ I/Os
  - Cache-oblivious string dictionary [SODA07]: $O(\frac{|P|+occ}{B} + \log_B n)$ I/Os

Main memory k-error indexing in literature

Space	Searching Time	Authors
$O(n)$	$O((cm)^k \log n + occ)$	Huynh et al. [CPM04]
$O(n)$ *bits*	$O(((cm)^k \log n + occ) \log n)$	Huynh et al. [CPM04]
$O(n)$	$O((cm)^k \log\log n + occ)$	Lam et al. [ISAAC05]
$O(n)$ *bits*	$O(((cm)^k \log\log n + occ) \log^{\epsilon} n)$	Lam et al. [ISAAC05]

Main memory k-error indexing in literature

Space	Searching Time	Authors
$O(\frac{c^k}{k!} n \log^k n)$	$O(m + \frac{c^k}{k!} \log^k n \log\log n + occ)$	Cole et al. [STOC04]
$O(n)$	$O((cm)^k \log n + occ)$	Huynh et al. [CPM04]
$O(n)$ *bits*	$O(((cm)^k \log n + occ) \log n)$	Huynh et al. [CPM04]
$O(n)$	$O((cm)^k \log\log n + occ)$	Lam et al. [ISAAC05]
$O(n)$ *bits*	$O(((cm)^k \log\log n + occ) \log^\epsilon n)$	Lam et al. [ISAAC05]

Main memory k-error indexing in literature

Space	Searching Time	Authors
$O(\frac{c^k}{k!} n \log^k n)$	$O(m + \frac{c^k}{k!} \log^k n \log\log n + occ)$	Cole et al. [STOC04]
$O(n)$	$O((cm)^k \log n + occ)$	Huynh et al. [CPM04]
$O(n)$ *bits*	$O(((cm)^k \log n + occ) \log n)$	Huynh et al. [CPM04]
$O(n)$	$O((cm)^k \log\log n + occ)$	Lam et al. [ISAAC05]
$O(n)$ *bits*	$O(((cm)^k \log\log n + occ) \log^\epsilon n)$	Lam et al. [ISAAC05]
$O(n)$	$O(m + (c \log n)^{k(k+1)} \log\log n + occ)$	Chan et al. [CPM06]
$O(n)$ *bits*	$O((m + (c \log n)^{k(k+2)} \log\log n + occ) \log^\epsilon n)$	Chan et al. [CPM06]

Our Results

Space (words)	Searching	Authors
$O(\frac{c^k}{k!} n \log^k n)$	$O(m + occ + \frac{c^k}{k!} \log^k n \log\log n)$ time	Cole et al. [STOC04]
$O(\frac{c^k}{k!} n \log^k n)$	$O(\frac{m+occ}{B} + \frac{c^k}{k!} \log^k n \log\log_B n)$ I/O	Our Result
$O(n)$	$O(m + occ + (c \log n)^{k(k+1)} \log\log n)$	Chan et al. [CPM06]
$O(n \log n)$	$O(\frac{m+occ}{B} + (c \log n)^{k(k+1)} \log\log_B n)$	Our Result

Review of Cole et al.'s data structure

- We give a highly simplified review of Cole et al.'s index
  - Refer to their paper for details
- Chan et al.'s index is similar
  - They use an additional Tree Cross Product data structure
  - See the paper for details on making it cache-oblivious

Review of Cole et al.'s data structure

- We give a highly simplified review of Cole et al.'s index
  - Refer to their paper for details
- Chan et al.'s index is similar
  - They use an additional Tree Cross Product data structure
  - See the paper for details on making it cache-oblivious
- The index consists of numerous compact tries
- The searching algorithm may navigate a compact trie in two ways
  - Navigate along tree edges
  - Jump to another location of the trie by an "LCP query"

LCP query

- LCP query for a string X on a node u of some compact trie asks for a "location" down from u by traversing along the longest possible prefix of X
- For a query pattern $P[1..m]$, X must be in the form $P[i..m]$

LCP query

- LCP query for a string X on a node u of some compact trie asks for a "location" down from u by traversing along the longest possible prefix of X
- For a query pattern $P[1..m]$, X must be in the form $P[i..m]$
- To support efficient LCP queries, we precompute some information of $P[1..m]$ w.r.t. $T[1..n]$
- LCP query takes $O(\log \log n)$ time, mainly due to $O(1)$ "Predecessor" and "Weighted Measured Ancestor" queries

LCP query

- LCP query for a string X on a node u of some compact trie asks for a "location" down from u by traversing along the longest possible prefix of X
- For a query pattern $P[1..m]$, X must be in the form $P[i..m]$
- To support efficient LCP queries, we precompute some information of $P[1..m]$ w.r.t. $T[1..n]$
- LCP query takes $O(\log \log n)$ time, mainly due to $O(1)$ "Predecessor" and "Weighted Measured Ancestor" queries
- Let $LCP(P[i..m], T)$ be the output of LCP query for $P[i..m]$ at the root of suffix tree of T
- Cole et al. uses $O(m)$ time (random access on the index) to precompute $LCP(P[i..m], T)$ for all $1 \leq i \leq m$

Challenges and solutions

- Reporting occurrences using $O(\text{searching} + \frac{occ}{B})$ I/Os
  - For each compact trie, store an array of its leaves
  - Reporting x descending leaves of a tree node takes $O(1 + \frac{x}{B})$ I/Os
- Predecessor and WLA queries using $O(\log \log_B n)$ I/Os
  - Predecessor query has known solution
  - We give a novel data structure for cache-oblivious WLA
  - Refer to the full paper for details

Challenges and solutions (cont'd)

- Need for $LCP(P[i..m], T)$ for all $1 \leq i \leq m$
  - We are allowed to use $O(\frac{m}{B} + \text{poly-}\log n)$ I/Os, but not $O(m)$
  - We show that only $O(k)$ of them are sufficient for searching (the others can be derived from them)
- Each $LCP()$ call take $O(\frac{m}{B} + \log_B n)$ I/Os to compute
  - Our $O(k)$ $LCP()$ calls are somehow special
  - We give an algorithm to find them using $O(\frac{m}{B} + k \log_B n)$ I/Os

Why O(k) is enough?

- Suppose there are positions $c_1, c_2, \ldots, c_{k-1}$ such that $P[1..c_1-2], P[c_1..c_2-2], \ldots, P[c_{k-1}..m]$ are all substrings of T
  - Is it possible that such c_j's do not exist?
  - This occurs only when there is no k-error match

Why O(k) is enough?

- Suppose there are positions $c_1, c_2, \ldots, c_{k-1}$ such that $P[1..c_1–2], P[c_1..c_2–2], \ldots, P[c_{k-1}..m]$ are all substrings of T
  - Is it possible that such c_j's do not exist?
  - This occurs only when there is no k-error match
- Compute $LCP(P[c_j..m], T)$ and $LCP(P[c_j..c_{j+1}–2], T)$ for all j
- LCP query for $P[i..m]$ where $c_{j-1} \leq i \leq c_j–2$ by
  - First, perform LCP query for $P[i..c_j–2]$
  - Then, traverse down the trie for the character $P[c_j–1]$
  - Finally, perform LCP query for $P[c_j..m]$

  (validations for success of the previous LCP query omitted)

Why O(k) is enough?

- Suppose there are positions $c_1, c_2, \ldots, c_{k-1}$ such that $P[1..c_1–2], P[c_1..c_2–2], \ldots, P[c_{k-1}..m]$ are all substrings of T
  - Is it possible that such c_j's do not exist?
  - This occurs only when there is no k-error match
- Compute $LCP(P[c_j..m], T)$ and $LCP(P[c_j..c_{j+1}–2], T)$ for all j
- LCP query for $P[i..m]$ where $c_{j-1} \leq i \leq c_j–2$ by
  - First, perform LCP query for $P[i..c_j–2]$
  - Then, traverse down the trie for the character $P[c_j–1]$
  - Finally, perform LCP query for $P[c_j..m]$

  (validations for success of the previous LCP query omitted)
- $LCP(P[i..c_j–2], T)$ can be derived from $LCP(P[c_{j-1}..c_j–2], T)$ with help of inverse suffix array (which takes $O(1)$ I/Os)

Computing the k positions

- How do we compute the positions?
  - Greedy
  - Let c_1 be largest s.t. $P[1..c_1-2]$ is a substring of T
  - Let c_2 be largest s.t. $P[c_1..c_2-2]$ is a substring of T
  - and so on ...
  - Each of them can be computed in $O(\frac{m}{B} + \log_B n)$ I/Os using Cache-oblivious string dictionary
- Easy to see the algorithm succeeds iff such positions exist

Computing the k positions

- How do we compute the positions?
  - Greedy
  - Let c_1 be largest s.t. $P[1..c_1–2]$ is a substring of T
  - Let c_2 be largest s.t. $P[c_1..c_2–2]$ is a substring of T
  - and so on ...
  - Each of them can be computed in $O(\frac{m}{B} + \log_B n)$ I/Os using Cache-oblivious string dictionary
- Easy to see the algorithm succeeds iff such positions exist
- The complete pre-computation
  - Compute $c_1, c_2, \ldots c_k$ as above, stop if P already exhausted
  - If $c_k - 2 < m$, report no k-error occurrence for P
  - Compute $LCP(P[c_{j-1}..c_j–2], T)$
  - Compute $LCP(P[c_{j-1}..m], T)$ in decreasing order of j

Avoiding O(km/B) I/Os

- We use the Cache-oblivious string dictionary like this
  - $lookup(P[1..m], D) \rightarrow c_1$
  - $lookup(P[c_1..m], D) \rightarrow c_2$
  - $\ldots$
  - $lookup(P[c_{k-1}..m], D) \rightarrow c_k$

Avoiding O(km/B) I/Os

- We use the Cache-oblivious string dictionary like this
  - $lookup(P[1..m], D) \rightarrow c_1$
  - $lookup(P[c_1..m], D) \rightarrow c_2$
  - $\ldots$
  - $lookup(P[c_{k-1}..m], D) \rightarrow c_k$
- If c_j's are small, strings queried have total length $O(km)$
  - Reason: querying $P[1..m]$ is wasteful if c_1 is small
  - Instead, we begin with $r = \lceil m/k \rceil$ and we query $P[1..r]$
  - If $P[1..r]$ is exhausted, we double r and try again
  - Otherwise, we proceed with finding c_2 by resetting r

Avoiding O(km/B) I/Os

- We use the Cache-oblivious string dictionary like this
  - $lookup(P[1..m], D) \rightarrow c_1$
  - $lookup(P[c_1..m], D) \rightarrow c_2$
  - $\ldots$
  - $lookup(P[c_{k-1}..m], D) \rightarrow c_k$
- If c_j's are small, strings queried have total length $O(km)$
  - Reason: querying $P[1..m]$ is wasteful if c_1 is small
  - Instead, we begin with $r = \lceil m/k \rceil$ and we query $P[1..r]$
  - If $P[1..r]$ is exhausted, we double r and try again
  - Otherwise, we proceed with finding c_2 by resetting r
- We still make $O(k)$ queries in total due to doubling
- Total length of strings queried is $O(m)$ since the last (failed) attempt for each c_j is bounded by $\max\{\lceil m/k \rceil, 2(c_j - c_{j-1})\}$

Conclusion

- We transformed the two main memory indexes into cache-oblivious ones
  - k-partitionability is a necessary condition for existence of k-error matches
  - k-partitioning is useful for computing longest common prefix of every suffix of P against the suffix tree
  - k-partitioning can be sped up by a doubling technique
  - We gave a cache-oblivious weighted level ancestor data structure, which may have independent interest
- It is open whether we can further reduce the $\log^k n$ term by at least a factor of $\log B$, such as $O(\frac{m+occ}{B} + \text{poly-}\log_B n)$ I/Os for $k = 1$