Distributed Enhanced Suffix Arrays
Efficient Construction and Querying
**Pattern search / Substring search:**

- Given string $S$, and pattern string $P$
  - $n = |S|$, $m = |P|$, (generally $n \gg m$)
- Find the occurrences of $P$ in $S$
- Naively: $O(mn)$

**Faster?**

1. Pre-process $P$:
   - e.g., Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore
   - KMP: pre-process $P$ in $O(m)$, then $O(n)$ search

2. Pre-process $S$:
   - String Indices: Suffix Tree, Suffix Array, FM-index, …
   - Suffix trees: pre-process $S$ in $O(n)$, then $O(m)$ search
String / Text Indexing

• **Indexing** is required for fast **substring search**

• **Major approaches:**
  1. Index all words: **inverted index**
  2. Index all fixed size substrings: **k-mer index (q-gram index)**
  3. Index all substrings: **suffix arrays, suffix trees, FM index**

• **Structured texts** are “cheap” to index
  - e.g. natural language, websites, documents, etc

• **Unstructured texts:** e.g. genomic sequences
  
```text
c tgccagtgaattacggtatatgcacactttggaactagaactaat...
```
Introduction

- **Suffix Tree (ST)**
  - compact trie of all suffixes of a string
  - fundamental and powerful indexing structure

- **Suffix Array (SA)**
  - array of sorted suffixes
  - represents leafs of ST

- **Longest Common Prefix (LCP)**
  - length of prefix match between consecutive suffixes in SA

- **Important Applications:**
  - Approximate pattern matching, finding of longest common substrings, all-pair maximal overlaps, data compression

\[
S = \text{mississippi}
\]

**Suffix Tree (ST):**

**Suffix Array (SA):**

**Longest Common Prefix (LCP):**
Distributed bulk pattern search

- Given distributed string $S$, and $q$ pattern strings $P_1, P_2, \ldots, P_q$ on $p$ PEs
- For every pattern find all occurrences in $S$

1. **Replicated index**: replicate both input and index onto all PEs
   - Distributed querying is easy: can sent any query to any PE
   - Drawback: Does not scale past the memory of single node:
     - $O(n)$ memory requirements on every PE

2. **Local index**: split the input onto PEs and build separate index over each part
   - Distributed querying does not scale: need to sent any query to every PE

3. **Global index**: build a distributed index over the distributed input
   - Distributed querying: for a given query, a single PE contains the answer
   - Memory scalable: $O(n/p)$ memory per PE
Distributed Suffix Index (Global Index)

ST:

SA:

LCP:

S:

P_0  P_1  P_2  P_3

n/p  n/p  n/p  n/p

Top-Level Index

Distribute subtrees
Main Components

1. Distribution and Top-Level Index:
   • How to distribute local-indices and subtrees
   • How to represent the Top-Level Index

2. Local Indices
   • How to represent the local indices and allow efficient querying

3. Distributed Construction
   • How to efficiently construct the components in distributed memory
     • SA and LCP: our SC’15, scalable distributed construction algorithm using \( O(n/p) \) memory per node
**Sequential (non-distributed) string indices**

- Query complexity, space usage
- Data dependencies
  - Efficient in distributed memory?

<table>
<thead>
<tr>
<th>String Index</th>
<th>Pattern Search</th>
<th>Space excl S</th>
<th>DM eff</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Tree</td>
<td>$O(m)$</td>
<td>$20\sigma n$</td>
<td>$54n$</td>
<td>Weiner (1973)</td>
</tr>
<tr>
<td></td>
<td>$O(m \ \log \sigma)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix Array</td>
<td>$O(m \ \log n)$</td>
<td>$8n$</td>
<td></td>
<td>Manber &amp; Myers (1990)</td>
</tr>
<tr>
<td>SA + LCP</td>
<td>$O(m + \log n)$</td>
<td>$12n^*$</td>
<td></td>
<td>Manber &amp; Myers (1990)</td>
</tr>
</tbody>
</table>

* Assuming 64 bit SA & 32 bit LCP array, excluding input string S

[Arroyueloa et al. 2014] explore various ways of distributing the **SA** to speed up binary search
- All methods share $\log(n)$ global communication steps for every query.

Flick & Aluru (2019)

* Efficient querying in distributed representation?
Suffix Array Search [Manber & Myers ‘90]

• Binary search the SA for pattern P

\[
[l, r] \leftarrow [0, n - 1]
\]

\[
\text{while } r - l > 1 \text{ do}
\]

\[
\text{mid} \leftarrow \frac{l + r}{2}
\]

\[
\text{if } P \leq_{\text{lex}} S[SA[mid] \ldots] \text{ then}
\]

\[
r \leftarrow \text{mid}
\]

\[
\text{else}
\]

\[
l \leftarrow \text{mid}
\]

\[
\text{end}
\]

\[
\text{end}
\]

• What’s the problem?

\[
S = \text{mississippi}$
\]

\[
P = \text{issis}
\]

\[
\text{issis} \leq_{\text{lex}} \text{mississippi}$ ✔
\]

\[
\text{issis} \leq_{\text{lex}} \text{ippi}$ X
\]

\[
\text{issis} \leq_{\text{lex}} \text{issippi}$ X
\]

\[
\text{issis} \leq_{\text{lex}} \text{ississippi}$ ✔
\]

Accesses the string in arbitrary locations

-> Random accesses into $O(n)$ memory
String Indices

**Sequential (non-distributed) string indices**
- Query complexity, space usage
- Data dependencies
  - Efficient in distributed memory?

<table>
<thead>
<tr>
<th>String Index</th>
<th>Pattern Search</th>
<th>Space excl S</th>
<th>DM eff</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Tree</td>
<td>$O(m)$</td>
<td>$20\sigma n$</td>
<td>✔</td>
<td>Weiner (1973)</td>
</tr>
<tr>
<td></td>
<td>$O(m \log \sigma)$</td>
<td>$54n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix Array</td>
<td>$O(m \log n)$</td>
<td>$8n$</td>
<td>✘</td>
<td>Manber &amp; Myers (1990)</td>
</tr>
<tr>
<td>SA + LCP</td>
<td>$O(m + \log n)$</td>
<td>$12n^*$</td>
<td>✘</td>
<td>Manber &amp; Myers (1990)</td>
</tr>
<tr>
<td>ESA (SA + LCP + child-table)</td>
<td>$O(\sigma m)$</td>
<td>$28n$</td>
<td></td>
<td>Abouelhoda et al. (2004)</td>
</tr>
<tr>
<td>ESA² (SA + LCP + RMQ(LCP))</td>
<td>$O(\sigma m)$</td>
<td>$14n+o(n)$</td>
<td></td>
<td>Fischer &amp; Heun (2008)</td>
</tr>
<tr>
<td>DESA (SA + LCP + RMQ(LCP) + $L_c$)</td>
<td>$O(\sigma m)$</td>
<td>$15n+o(n)$</td>
<td></td>
<td>Flick &amp; Aluru (2019)</td>
</tr>
</tbody>
</table>

* Assuming 64 bit SA & 32 bit LCP array, excluding input string $S$
Efficient distributed querying

• Requirements:
  • Subtree of size $k$ ($k$ leaves) should be representable using $O(k)$ memory
  • Querying within a subtree optimally shouldn’t access data outside of its data range

• Problem?
  • The entire string $S$ lies on a root to leaf path
  • Existing methods (randomly) access $O(n)$ sized arrays
ESA access patterns

- Querying a size $k$ subtree
  - “random” reads of string $S$
    at every step ($O(m)$ many times)

- Similar for SA search and backward-search methods

Prohibitively expensive in distributed memory!
DESA access pattern

**ESA**
- $O(m)$ local accesses into LCP and SA
- $O(m)$ “random” accesses into string $S$

**DESA**
- $O(m)$ local accesses into LCP and $L_c$
- one local read of SA
- one string comparison into $S$
**Claim:**
For every \( i \), and every reasonable \( l < i \):

\[
S[SA[l] + LCP[i]] = S[SA[i - 1] + LCP[i]]
\]

We can pre-compute this for every \( i \).
Introducing array $L_c$:

$$L_c[i] = S[SA[i] + LCP[i + 1]]$$

- Contains the left-branching character for every LCP position
- If inner node has $d$ children, this contains the 1st character on the edge of the first $d - 1$ children.
DESA query example

1. Traverse using only left branching characters
   - uses LCP, RMQ(LCP), Lc
   - false-positives possible
   - similar to blind-search in Patricia Tries

2. Rule-out false-positives by single string comparison
   - uses SA and S
   - check: S[SA[l] ... ] == P ?
ESA vs DESA summary

ESA
• $O(m)$ local accesses into LCP and SA
• $O(m)$ “random” accesses into string $S$

DESA
• $O(m)$ local accesses into LCP and $L_c$
• one local read of SA
• one string comparison into $S$
First results

- Benchmarking a sequential (non distributed) implementation
- ESA vs DESA vs SDLS indices
- *Pizza & Chili* Benchmark

### Query Time in μs per query

<table>
<thead>
<tr>
<th></th>
<th>dna</th>
<th>proteins</th>
<th>english</th>
<th>sources</th>
<th>dplb</th>
</tr>
</thead>
<tbody>
<tr>
<td>esa_index</td>
<td>7.6</td>
<td>11.2</td>
<td>28.0</td>
<td>30.4</td>
<td>26.6</td>
</tr>
<tr>
<td>desa_index</td>
<td>6.4</td>
<td>9.7</td>
<td>19.1</td>
<td>20.0</td>
<td>15.7</td>
</tr>
<tr>
<td>desa_tl_index</td>
<td>6.0</td>
<td>5.8</td>
<td>14.5</td>
<td>14.7</td>
<td>10.1</td>
</tr>
<tr>
<td>sdsl::csa_wt</td>
<td>6.3</td>
<td>13.7</td>
<td>15.1</td>
<td>19.9</td>
<td>18.8</td>
</tr>
<tr>
<td>sdsl::csa_sada</td>
<td>74.9</td>
<td>72.2</td>
<td>65.9</td>
<td>94.1</td>
<td>97.9</td>
</tr>
</tbody>
</table>
Main Components

1. Distribution and Top-Level Index:
   - How to distribute local-indices and subtrees
   - How to represent the Top-Level Index

2. Local Indices
   - DESA \((SA + LCP + RMQ(LCP) + L_c)\)
   - Fast local search of subtrees

3. Distributed Construction
   - How to efficiently construct the components in distributed memory
Distribution during construction:
- Equally block distributed

Distribution for DESA querying:
- Distribution of subtrees

(1) Create TLI
(2) 1D-partition
(3) Re-distribute
Top Level Index

• Duplicated on each processor
• Models the first levels of the ST
• Given a pattern $P$, the TLI can be locally queried for the location (processor) of the subtree containing $P$

Distribution

1. *Fixed $n/p$ per processor*
   • Top-Level Index indexes all suffixes on processor boundaries
   • Problem:
     • Potentially very large common prefix on boundary
   • [Fischer et al. 2017] use constant, fixed first $q = 30$ characters of each boundary suffix
     • $\implies$ No support for queries $|P| > 30$

2. *Redistribution of sub-trees*
   • Find better split points between sub-trees
   • Support arbitrary query length
   • data-dependent
   • Comparing two approaches:
     • *Top-Level Lookup Table (TLLT)*
     • *Top-Level Dynamic Trie (TLDT)*
Top-Level Index

Top-Level Lookup Table (TLLT)
• $k$-mer table
  • for each possible size $k$ string contains the offset into SA
  • Size $|\Sigma|^k$
• Easy to implement
• Not good for large load imbalance

Top-Level Dynamic Trie (TLDT)
• Dynamically compute the sub-trees based on target load-imbalance factor
• Constructed using a type of All-Nearer-Smallest-Values algorithm that keeps just ranges larger than some threshold
• $O(n/p)$ time ANSV-like followed by allgather of $LCP$, $L_c$ and index ranges
• Details in Paper
- Distribution and Top-Level index comparison
  - Imbalance vs size of TLI
  - Distribution for \( p = 1024 \) processors for *Pizza & Chili* corpus
  - Imbalance = \( \frac{\text{max size} - \text{avg size}}{\text{avg size}} \)
  - Clear winner:
    - *Top-Level Dynamic Trie (TLDT)*

<table>
<thead>
<tr>
<th>Input</th>
<th>sigma</th>
<th>k</th>
<th>Size (MB)</th>
<th>imbalance</th>
<th>Size (MB)</th>
<th>imbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr1</td>
<td>4</td>
<td>8</td>
<td>131</td>
<td>71.3%</td>
<td>6</td>
<td>0.7%</td>
</tr>
<tr>
<td>dna</td>
<td>16</td>
<td>5</td>
<td>262</td>
<td>582%</td>
<td>6</td>
<td>0.5%</td>
</tr>
<tr>
<td>proteins</td>
<td>27</td>
<td>5</td>
<td>262</td>
<td>4%</td>
<td>19</td>
<td>0.5%</td>
</tr>
<tr>
<td>english</td>
<td>239</td>
<td>3</td>
<td>131</td>
<td>1639%</td>
<td>29</td>
<td>0.5%</td>
</tr>
<tr>
<td>sources</td>
<td>230</td>
<td>3</td>
<td>131</td>
<td>3060%</td>
<td>39</td>
<td>1.9%</td>
</tr>
<tr>
<td>dblp.xml</td>
<td>97</td>
<td>3</td>
<td>16</td>
<td>2577%</td>
<td>22</td>
<td>0.4%</td>
</tr>
</tbody>
</table>
Distributed Querying

1. Step 1: Processor P₁
   i. Pattern arrives at processor P₁
   ii. P₁ queries the TLI
   iii. P₁ sends the pattern to P₂

2. Step 2: Processor P₂
   i. P₂ queries the local DESA subtree for the pattern
   ii. Read \( SA[l] \): the string position of one of the possible matches
   iii. Send pattern to processor containing the string segment \( SA[l] \)

3. Step 3: Processor P₃
   i. Compare pattern to string
Main Components

1. Distribution and Top-Level Index:
   - Top-Level Dynamic Tries (TLDT) replicated on all processors

2. Local Indices
   - DESA \((SA + LCP + RMQ(LCP) + L_c)\)
   - Fast local search of subtrees

3. Distributed Construction
   - How to efficiently construct the components in distributed memory
Efficient distributed construction of $L_c$

- During our SC15 distributed SA & LCP construction algorithm
- Given all suffixes are sorted by their $h$-prefix
- Prefix doubling sorts by $2h$-prefix
- New boundaries of $2h$-groups: Set the $LCP[i]$ to:

$$LCP[i] = h + LCP[RMQ(a, b)]$$

where
- $a = \text{rank of suffix } SA[i - 1] + h$
- $b = \text{rank of suffix } SA[i] + h$

- In same step, we set:
  - $L_c[i] = L_c[RMQ(a, b)]$

- We proof, that:
  - $L_c[RMQ(a, b)] = S[SA[i - 1] + LCP[i]]$

- Near zero overhead.
- Does not affect experimental runtime of SA & LCP construction
- Effectively, we’re getting $L_c$ for free
1. Distribution and Top-Level Index:
   - **Top-Level Dynamic Tries (TLDT)** replicated on all processors

2. Local Indices
   - **DESA** (**SA + LCP + RMQ(LCP) + L_c**)
   - Fast local search of subtrees

3. Distributed Construction
   - **SA + LCP**: [Flick & Aluru SC’15]
   - **L_c**: “free” during LCP construction
   - **TLDT**: $O(n/p)$ time ANSV-like followed by allgather of LCP, $L_c$ and index ranges
Results: Construction

- Small overhead over SA and LCP construction (5-8%)
- $L_c$ construction has no measurable impact on runtime
Results: Construction

Construction time comparison

- Distributed Patricia Trie (DPT) [Fischer et al. 2017]
  - [github.com/kurpicz/dpt](https://github.com/kurpicz/dpt)
  - Constructed using our SA + LCP [SC’15]
  - Static decomposition, limited query size to small constant (q=30)
  - Local index: pointer based patricia tries
  - Space: > 48n
    - ours 15n + o(n)

Construction Results:

DESA 14.1 - 25.7 × faster given SA and LCP
- Total of 2.0 - 2.6 × faster
Results: Querying

- 10M queries of length 30 on chr1
- DESA 13-29% faster than DPT (despite doing more work)
- Single node 36.4x speedup (48 cores)
- 1 -> 8 nodes: 7.4x speedup (total 270.9x on 384 cores)
Results: Weak Scaling

- Weak scaling
  - Entire human genome
  - 10M queries per core

System: Stampede2
Summary

Distributed Enhanced Suffix Array (DESA)

• Distributed string index data structure
• Efficient querying with constant communication steps per query
• Cheap construction:
  • Small overhead over constructing just SA & LCP
• Sequentially outperforms ESAs
• Efficiently scales to > 1.5k cores

Questions?

Contact
Patrick Flick
patrick.flick@gatech.edu
patflick.github.io

Srinivas Aluru
aluru@cc.gatech.edu

Funding