External Memory Suffix Array Construction

Roman Dementiev
Juha Kärkkäinen
Jens Mehnert
Peter Sanders

MPI Informatik, U. Karlsruhe, U. Helsinki
Suffix Arrays

sort suffixes $T[i..n]$ of string $T[0..n]$ 
over alphabet $\{1..n\}$.

Applications

□ Full text search
□ Burrows-Wheeler text compression
□ Bioinformatics,…

Big interest in BIG inputs \(\sim\)

External memory

$$\text{scan}(n) = \frac{n}{B}, \quad \text{sort}(n) = \frac{2n}{B} \left[ \log_{M/B} \frac{n}{M} \right] \text{ machine words}$$
Related Work

Incremental: $\mathcal{O}\left(\frac{n}{M} \text{scan}(n)\right)$ I/Os [Gonnet/Baeza-Yates/Snider 92]

[CF 97] not very scalable, a lot of internal work

Doubling: Sort by first $2^i$ characters in iteration $i$ [Manber/Myers 93]

$\leadsto \mathcal{O}(\text{sort}(n) \log \text{maxlcp})$ I/Os [AFGV 97]

Doubling+Discarding: Avoid sorting suffixes known to be unique [Crauser/Ferragina 97]

Best scalable algorithm in study. $> 6h$ for 26 MByte.

$\leadsto$ External construction not practical?

via Suffix-Tree: $\mathcal{O}(\text{sort}(n))$ I/Os [Farach/Ferragina/Muthukrishnan 00]

very complicated

DC3: Simple, linear time, $\mathcal{O}(\text{sort}(n))$ I/Os [KS 03].

Practical? Better than improved doubling?
Pipelined Doubling with Bit Shuffling

\[ \text{name}(T[i..i + k]) \in \{1..n\} \text{ preserves order of } k\text{-substrings } \forall i \]

\((T[j], T[j+1], j)\)

\((\text{name}(T[j..j+2^i]), \text{name}(T[j+2^i..j+2^{i+1}], j))\)  \(i := i+1\)

\(3n \text{ words}\)

\text{sort} \quad \text{runs} \quad \text{merge} \quad \text{name} \quad \text{i bits} \quad \text{sort} \quad \text{pair}

\text{total I/O complexity: } \text{sort}(5n) \log \text{maxlcp} + \mathcal{O}(\text{sort}(n))\)
Improved Discarding

- Scan all unique suffixes [CF 97]
  - Scan new unique suffixes [Kärkkäinen 03]

- Triples → pairs

\[
\text{sort}(5N) + \Theta(\text{sort}(n)) \text{ I/Os where } N = \sum_{i} \log \text{distPrefixSize}(T[i..n])
\]
\( \alpha \text{-Tupling} \)

Sort by first \( \alpha^i \) characters in iteration \( i \)

**Constant Factor in I/Os**

<table>
<thead>
<tr>
<th>( a )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a + 3) / \log a)</td>
<td>5.00</td>
<td>3.78</td>
<td>3.50</td>
<td>3.45</td>
<td>3.48</td>
<td>3.56</td>
</tr>
</tbody>
</table>
Difference Cover 3 (DC3) Algorithm

1. sort $T[i..n]$ for $i \mod 3 \in \{1, 2\}$
   - sort and name triples
   - recurse

2. sort $T[i..n]$ for $i \mod 3 \in \{0\}$
   - sort pairs $(T[3i], \text{name}(T[3i + 1..n]))$

3. merge using difference cover property of $\{1, 2\}$
   - $T[3i..n] \leq T[3j + 1..n]$ iff
     - $(T[3i], \text{name}(T[3i + 1..n])) \leq (T[3j + 1], \text{name}(T[3j + 2..n]))$
   - $T[3i..n] \leq T[3j + 2..n]$ iff
     - $(T[3i], T[3i + 1], \text{name}(T[3i + 2..n])) \leq (T[3j + 2], T[3j + 3], \text{name}(T[3j + 4..n]))$
Pipelined DC3

\[ \text{sort}(30n) + \text{scan}(6n) \text{ I/Os} \]
Experimental Setup

\texttt{g++3.2.3 -O2}

\texttt{STXXL} library [Dementiev 03] with new iterator-like pipelining feature

\textbf{Genome:} Human Genome

\textbf{Gutenberg:} \(\approx 3\)GByte English text from Gutenberg project

\textbf{HTML:} \(\approx 3\)GByte text from a crawl of .gov

\textbf{Source:} \(\approx 0.5\)GByte Linux sources

\textbf{Random2:} \(T \circ T\) with \(T := \text{randChar}^{n/2}\)
Gutenberg I/Os

- Doubling
- Quadrupling
- Discarding
- Quad-Discarding
- DC3

I/O Volume [byte] / n

n

2^{24} 2^{26} 2^{28} 2^{30} 2^{32}
Gutenberg Time

I/O bound even for a single disk
Comparison with Previous Implementations

- 5× less I/O volume than [CF 97]
- 7–8× less clock cycles than [CF 97] (including BGS algorithm)
- 2.4× faster than internal compressed Genome [LSSSY 02]
- 1.2× slower than internal Genome on 64 GByte super computer
  [Sadakane Shibuya 01]
- Faster than linear time internal LCP computation on MPII’s SUN Starfire 15000
Conclusion

- External DC3 is practical
- Better than pipelined, shuffled 4-tupling with improved discarding
- STXXL makes pipelining easy. Saves factor 2–3 in I/O volume.

Future Work

- Tune pipelined sorters
- Go parallel
- Larger difference covers for first iteration?
- Will discarding help for DC algorithms?

Terabytes over night?
Random2 I/Os

![Graph showing I/O Volume vs. n for different methods: Doubling, Discarding, Quadrupling, Quad-Discarding, and Skew. The graph plots I/O Volume [byte] / n on the y-axis and n on the x-axis, with various lines indicating different methods and markers for nonpipelined.](image-url)
Random2 Time

Random2: Time [\mu s] / n

n

nonpipelined
Doubling
Discarding
Quadrupling
Quad-Discarding
DC3
Genome I/Os

![Graph showing Genome I/Os with different operations: Doubling, Quadrupling, Discarding, Quad-Discarding, and Skew. The x-axis represents n, and the y-axis represents I/O Volume [byte] / n. The graph includes markers for each operation, showing varying trends as n increases from $2^{24}$ to $2^{32}$.](image-url)
Genome Time

- Doubling
- Quadrupling
- Discarding
- Quad-Discarding
- Skew
Graphs showing I/O volume and source time for Quadrupling, Quad-Discarding, and Skew.