# External Memory BWT and LCP Computation for Sequence Collections with Applications

Lavinia Egidi
University of Eastern Piedmont, Italy

Felipe A. Louza
University of São Paulo, Ribeirão Preto, Brazil

Giovanni Manzini
University of Eastern Piedmont, Italy
IIT, National Research Council, Pisa, Italy

Guilherme P. Telles
University of Campinas, Campinas, Brazil

WABI 2018, Helsinki

# The problem

Given a collection of strings $S_1$ , $S_2$ , $S_3$ ... $S_k$ compute the BWT and LCP array of the collection.

Mainly interested in the case of many (relatively) short strings, but no limitations on the number of strings and their lengths

# Single string BWT/LCP

$S_1$ = abcab$

| LCP | BWT | suffixes |
|---|---|---|
| | b | $ |
| 0 | c | ab$ |
| 2 | $ | abcab$ |
| 0 | a | b$ |
| 1 | a | bcab$ |
| 0 | b | cab$ |

$S_2$ = aabcabc#

| LCP | BWT | suffixes |
|---|---|---|
| | c | # |
| 0 | # | aabcabc |
| 1 | c | abc# |
| 3 | a | abcabc# |
| 0 | a | bc# |
| 2 | a | bcabc# |
| 0 | b | c# |
| 1 | b | cabc# |

# Multi-String BWT/LCP

$S_1$ = abcab$

$S_2$ = aabcabc#

| LCP | BWT | suffixes |
|---|---|---|
| | b | $ |
| 0 | c | # |
| 0 | # | aabcabc |
| 1 | c | ab$ |
| 2 | c | abc# |
| 3 | $ | abcab$ |
| 5 | a | abcabc# |
| 0 | a | b$ |
| 1 | a | bc# |
| 2 | a | bcab$ |
| 4 | a | bcabc# |
| 0 | b | c# |
| 1 | b | cab$ |
| 3 | b | cabc# |

# The Terminator problem

We cannot use distinct terminators otherwise the alphabet size would blow up. We use the same symbol and keep track of its origin.

Eg: $\$_1 < \$_2 < \$_3 \cdots$

# State of the art

- gSACA+Φ optimal O(n) RAM algorithm: builds Suffix Array, LCP array, uses only 10KB working space [Louza et al 2017]
- BCR+LCP external memory algorithm from BEETL library. Disk I/Os: O(n Maxlen/(B log n)) [Cox et al 2011-2016].

# Our contribution

External memory algorithm taking $O(n \text{ MaxLCP}/(B \log n))$ I/Os

For DNA reads the actual running time is $O(n \text{ AveLCP}/(B \log n))$. In our tests we are faster than BCR+LCP for average length $\geq 300$

# Applications

- Multi-string BWT are used to build a compressed Suffix Array supporting counting queries and more..
- LCP values are used to emulate a Suffix Tree and solve a host of interesting problems:
  - ✔ Mutation detection (previous talk)
  - ✔ Maximal repeats
  - ✔ Suffix-Prefix Overlaps
  - ✔ Many others...

We propose external memory algorithms for computing Maximal repeats, All pairs suffix-prefix overlaps, and succinct De Bruijn graphs.

The challenge was to transform internal memory Suffix Tree algorithms into external memory algorithms working on BWT and LCP arrays.

# Algorithm outline

- Split the input into chunks that fit in RAM and compute BWT of each chunk using Louza et al. optimal algorithm
- Merge the BWTs in external memory and compute the LCP values using a modified H&M algorithm.
- Sort the LCP values in external memory using standard multiway merge

# Merging BWTs

| BWT | suffixes |
|---|---|
| b | $ |
| c | ab$ |
| $ | abcab$ |
| a | b$ |
| a | bcab$ |
| b | cab$ |

$S_1$ = abcab$

+

| BWT | suffixes |
|---|---|
| c | # |
| # | aabcabc |
| c | abc# |
| a | abcabc# |
| a | bc# |
| a | bcabc# |
| b | c# |
| b | cabc# |

$S_2$ = aabcabc#

=

| BWT | suffixes |
|---|---|
| b | $ |
| c | # |
| # | aabcabc |
| c | ab$ |
| c | abc# |
| $ | abcab$ |
| a | abcabc# |
| a | b$ |
| a | bc# |
| a | bcab$ |
| a | bcabc# |
| b | c# |
| b | cab$ |
| b | cabc# |

# The H&M algorithm

Holt and McMillan [Bioinformatics 2014] proposed a simple and elegant algorithm to merge BWTs in small space

The idea is to compute the bounded context BWT for context size 0,1,2,…

# Context size: 0

| | |
|---|---|
| b | $ |
| c | ab$ |
| $ | abcab$ |
| a | b$ |
| a | bcab$ |
| b | cab$ |

+

| | |
|---|---|
| c | # |
| # | aabcabc |
| c | abc# |
| a | abcabc# |
| a | bc# |
| a | bcabc# |
| b | c# |
| b | cabc# |

=

| Z | BWT | |
|---|---|---|
| 0 | b | $ |
| 0 | c | ab$ |
| 0 | $ | abcab$ |
| 0 | a | b$ |
| 0 | a | bcab$ |
| 0 | b | cab$ |
| 1 | c | # |
| 1 | # | aabcabc |
| 1 | c | aabc# |
| 1 | a | abcabc# |
| 1 | a | bc# |
| 1 | a | bcabc# |
| 1 | b | c# |
| 1 | b | cabc# |

We represent the merged BWT
with the array Z

# Context size: 1

| | |
|---|---|
| b | $ |
| c | ab$ |
| $ | abcab$ |
| a | b$ |
| a | bcab$ |
| b | cab$ |

+

| | |
|---|---|
| c | # |
| # | aabcabc |
| c | abc# |
| a | abcabc# |
| a | bc# |
| a | bcabc# |
| b | c# |
| b | cabc# |

=

| Z | BWT | |
|---|---|---|
| 0 | b | $ |
| 1 | c | # |
| 0 | c | ab$ |
| 0 | $ | abcab$ |
| 1 | # | aabcabc |
| 1 | c | aabc# |
| 1 | a | abcabc# |
| 0 | a | b$ |
| 0 | a | bcab$ |
| 1 | a | bc# |
| 1 | a | bcabc# |
| 0 | b | cab$ |
| 1 | b | c# |
| 1 | b | cabc# |

We represent the merged BWT
with the array Z

# Context size: 2

| | |
|---|---|
| b | $ |
| c | ab$ |
| $ | abcab$ |
| a | b$ |
| a | bcab$ |
| b | cab$ |

+

| | |
|---|---|
| c | # |
| # | aabcabc |
| c | abc# |
| a | abcabc# |
| a | bc# |
| a | bcabc# |
| b | c# |
| b | cabc# |

=

| Z | BWT | |
|---|---|---|
| 0 | b | $ |
| 1 | c | # |
| 1 | # | aabcabc |
| 1 | c | aabc# |
| 0 | c | ab$ |
| 0 | $ | abcab$ |
| 1 | a | abcabc# |
| 0 | a | b$ |
| 0 | a | bcab$ |
| 1 | a | bc# |
| 1 | a | bcabc# |
| 1 | b | c# |
| 0 | b | cab$ |
| 1 | b | cabc# |

We represent the merged BWT with the array Z

# Context size: 2

| | |
|---|---|
| b | $ |
| c | ab$ |
| $ | abcab$ |
| a | b$ |
| a | bcab$ |
| b | cab$ |

+

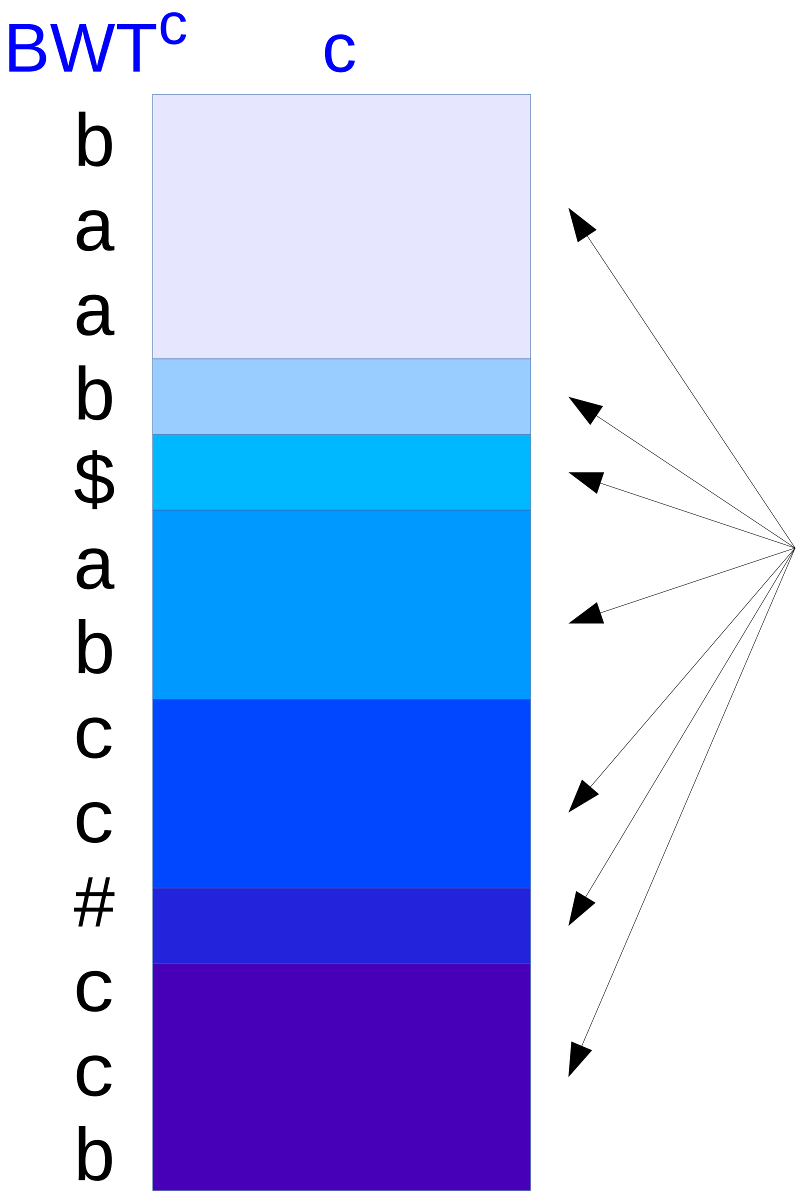
| | |
|---|---|
| c | # |
| # | aabcabc |
| c | abc# |
| a | abcabc# |
| a | bc# |
| a | bcabc# |
| b | c# |
| b | cabc# |

=

| Z | BWT | |
|---|---|---|
| 0 | b | $ |
| 1 | c | # |
| 1 | # | aabcabc |
| 1 | c | aabc# |
| 0 | c | ab$ |
| 0 | $ | abcab$ |
| 1 | a | abcabc# |
| 0 | a | b$ |
| 0 | a | bcab$ |
| 1 | a | bc# |
| 1 | a | bcabc# |
| 1 | b | c# |
| 0 | b | cab$ |
| 1 | b | cabc# |

The H&M algorithm only uses $BWT_0$ $BWT_1$ and Z

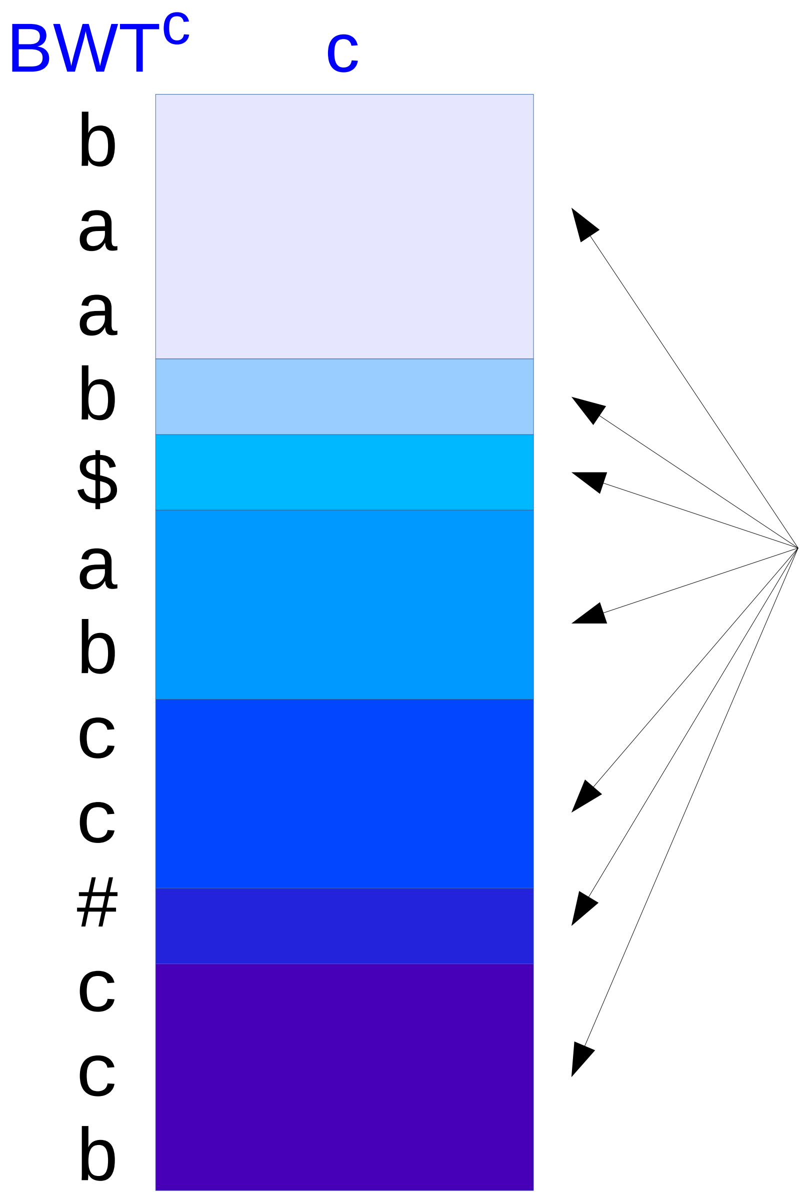
# Computing LCP values




At iteration c a block is a set of suffixes sharing a length-c prefix

Each iteration reorders suffixes within a block and creates new blocks splitting old ones

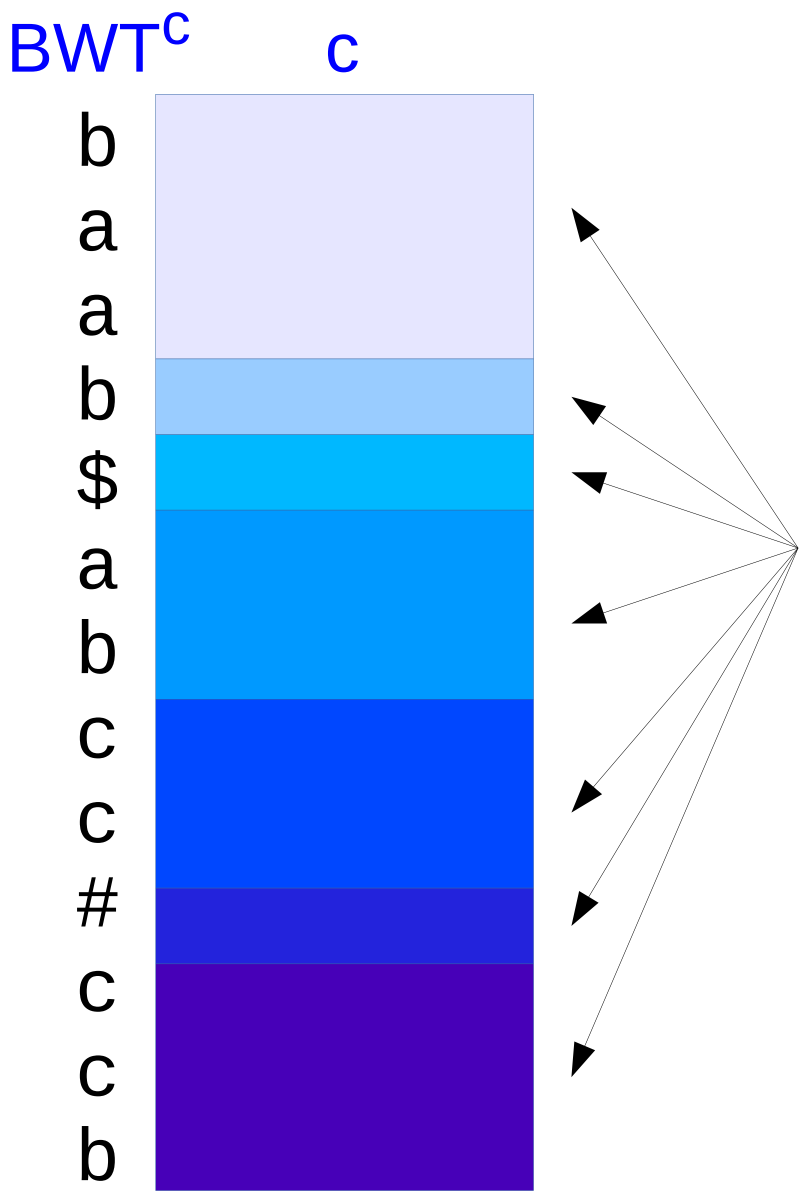
# Computing LCP values




We can prove that if we split a block at iteration c+1 the LCP value for the position where the splitting occurs is c.

- During the merging we “discover” larger and larger LCP values
- Each time a LCP value is discovered we write to file the pair <position, LCP>
- When the merging is done, we sort the pairs by position and save the resulting LCP values

# When to stop?




At iteration c a block is a set of suffixes sharing a length-c prefix

Each iteration reorders suffixes within a block and creates new blocks splitting old ones

# Running time

- When all the blocks have size 1 the merging is done and all LCP values have been computed
- The resulting algorithm takes O(n MaxLCP) time and O(n MaxLCP/(B log n)) I/Os
- A simple heuristic (skipping ranges of size-1 blocks) makes the I/O complexity in practice closer to O(n AveLCP/(B log n)) I/Os

# Experimental results

- External memory algorithms should be tested using RAM much smaller than input size
- The RAM should be limited at boot time, otherwise the OS will use the extra RAM to avoid disk transfers
- We have compared our algorithm (eGap) with the state of the art for external memory (BCR+LCP)

| Name | Size GB | $\sigma$ | N. of strings | Max Len | Ave Len | Max LCP | Ave LCP |
|---|---|---|---|---|---|---|---|
| shortreads | 8.0 | 6 | 85,899,345 | 100 | 100 | 99 | 27.90 |
| longreads | 8.0 | 5 | 28,633,115 | 300 | 300 | 299 | 90.28 |
| pacbio.1000 | 8.0 | 5 | 8,589,934 | 1,000 | 1,000 | 876 | 18.05 |
| pacbio | 8.0 | 5 | 942,248 | 71,561 | 9,116 | 3,084 | 18.32 |

Collections

| Name | eGap | | | BCR+LCP | | |
|---|---|---|---|---|---|---|
| | 1GB | 8GB | 32GB | 1GB | 8GB | 32GB |
| shortreads | 17.19 | 3.76 | 2.87 | $\times$ | 5.65 | 3.96 |
| longreads | 52.39 | 9.75 | 6.76 | 18.54 | 16.01 | 10.88 |
| pacbio.1000 | 24.88 | 3.54 | 1.81 | $\times$ | 54.00 | 36.96 |
| pacbio | 23.43 | 3.42 | 1.82 | $> 70$ | $> 50$ | $> 50$ |

Running time in µsecs x symb

# Observations

- eGap is faster for datasets with longer reads
- eGap running time appears to be related to AveLCP rather than MaxLCP
- eGap is better at exploiting all the available RAM
- More experiments required ...

# Summing Up

- We propose an external memory algorithm which is faster than the state of the art for long reads
- In addition to BWT and LCP our algorithm can produce the Document Array and the useful xlcp bit array (see paper)
- Using these additional arrays we can compute maximal repeats, suffix-prefix overlaps and compact DB-graphs efficiently in external memory