Converting RLBWT to LZ77 in smaller space

Kyushu Institute of Technology
Masaki Shigekuni & Tomohiro I

Background

- Various lossless compression methods have been proposed to efficiently process large-scale data
- Each compression method has its advantages and disadvantages, so it would be nice to be able to convert the compression format as needed

RLBWT: Compression method suited for compressed string indexes
LZ77: Dictionary-based compression achieving a high compression rate

- A one-pass algorithm of converting from RLBWT to LZ77 in compressed space has been proposed by [Nishimoto & Tabei, 2019]
- In this study, we propose a two-pass algorithm that reduces its peak memory usage

Suffix array

- The suffix array (SA) of T is a list of text positions sorted in the lexicographic order of suffixes starting at their positions

	1	2	3	4	5	6
$T =$	a	b	c	a	b	$

Suffixes
abcab$
bcab$
cab$
ab$
b$
$

SA position	SA	Sorted suffixes
1	6	$
2	4	ab$
3	1	abcab$
4	5	b$
5	2	bcab$
6	3	cab$

Burrows-Wheeler Transform (BWT)

- BWT is a permutation of characters of T
- BWT[i] = $T[SA[i]-1]$ if SA[i] > 1, otherwise BWT[i] = $
- BWT of a highly repetitive string can be compressed greatly by run-length encoding
- Run-length encoded BWT is called RLBWT

Suffixes
abcab$
bcab$
cab$
ab$
b$
$

BWT	SA	Sorted suffixes
b	6	$
c	4	ab$
$	1	abcab$
a	5	b$
a	2	bcab$
b	3	cab$

BWT of ”abcab$”

RLBWT

“bc$aab” → ”bc$$a^2$b”

Lempel-Ziv 77 (LZ77)

- Parse a string greedily into phrases from right-to-left such that each phrase consists of the longest substring that appears to the right plus one character
- The h-th phrase is encoded by a triplet $(\pi_h, \lambda_h, \mathrm{c}_h)$, representing $\mathrm{c}_h T[\pi_h .. \pi_h + \lambda_h - 1]$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T =$	a	b	b	a	b	b	b	a	b	a	b	a	a	$

Phrases: | a b b a b | b b a b a | b a | a | $

(6, 4, a)	(9, 4, b)	(13, 1, b)	(-, 0, a)	(-, 0, $)
$(\pi_5, \lambda_5, \mathrm{c}_5)$	$(\pi_4, \lambda_4, \mathrm{c}_4)$	$(\pi_3, \lambda_3, \mathrm{c}_3)$	$(\pi_2, \lambda_2, \mathrm{c}_2)$	$(\pi_1, \lambda_1, \mathrm{c}_1)$

Backward search

- The maximum interval prefixed with a string w is called w-interval
- The process of computing cw-interval from w-interval for a character c is called backward search

SA position	SA	Sorted suffixes
1	6	$
2	4	ab$
3	1	abcab$
4	5	b$
5	2	bcab$
6	3	cab$

b-interval (rows 4–5)

Backward search

- The maximum interval prefixed with a string w is called w-interval
- The process of computing cw-interval from w-interval for a character c is called backward search

SA position	SA	Sorted suffixes
1	6	$
2	4	ab$
3	1	abcab$
4	5	b$
5	2	bcab$
6	3	cab$

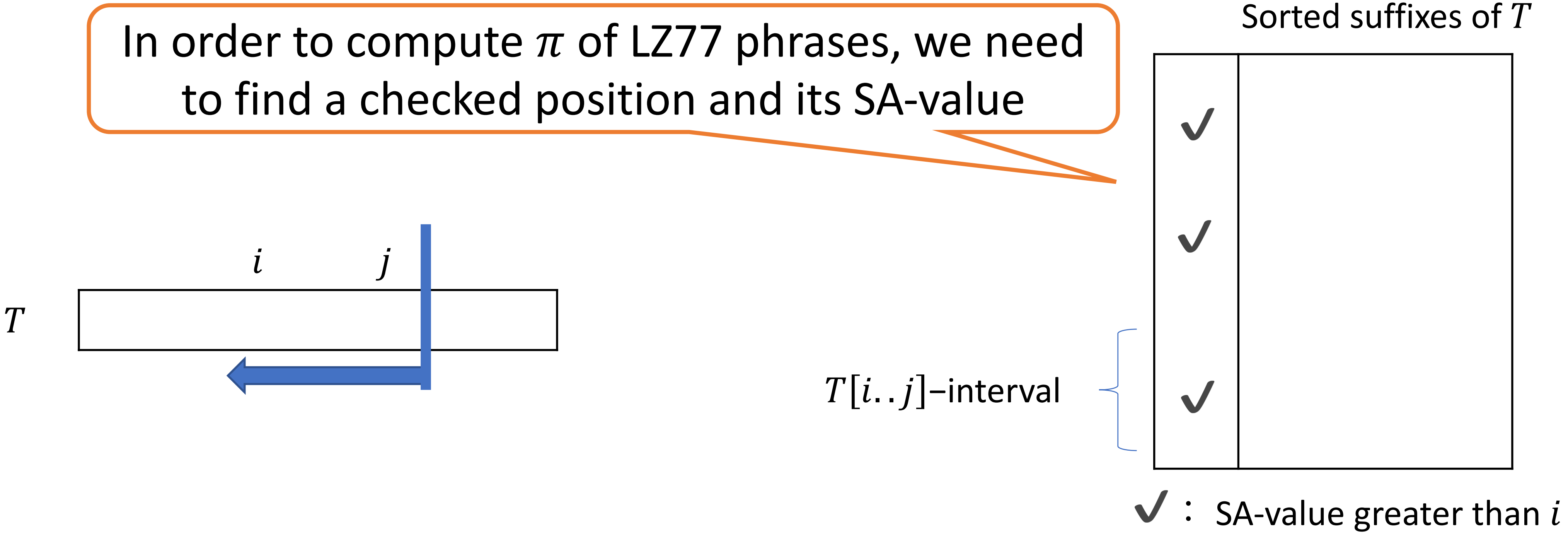
ab-interval (SA positions 2–3)

b-interval (SA positions 4–5)

- Can be conducted in $O(r)$ space （r is the number of the runs of BWT）

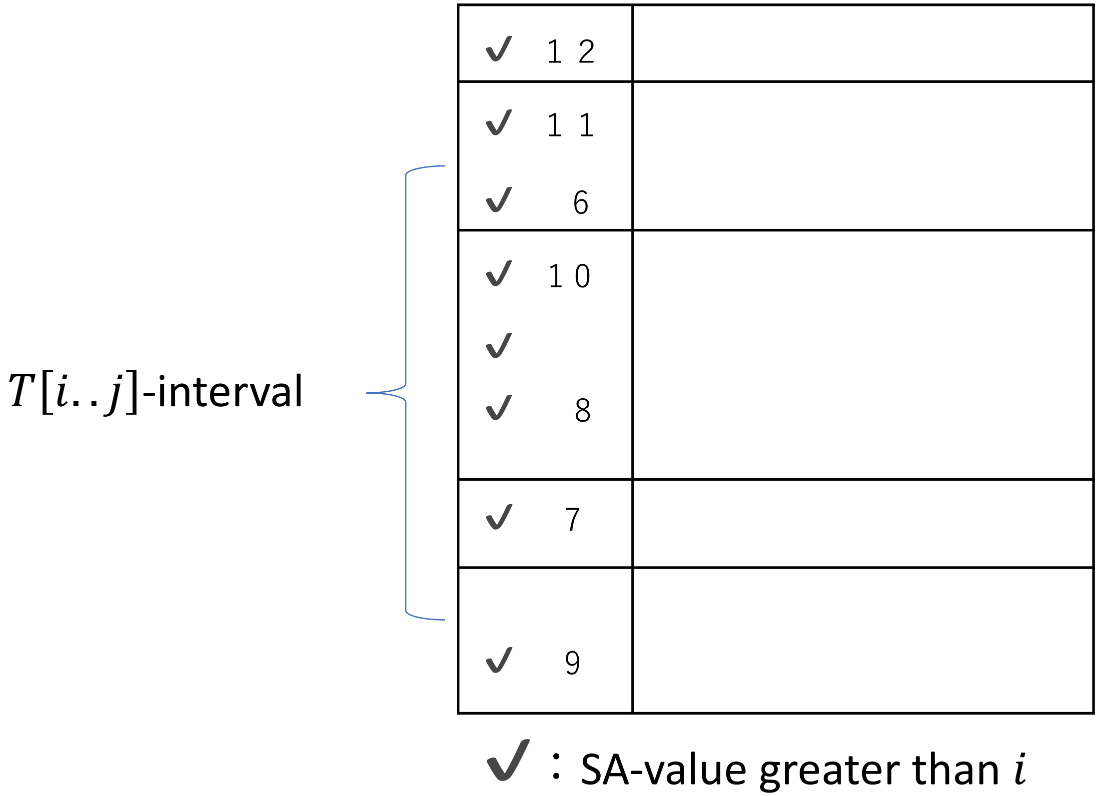
Compute LZ77 phrases using backward search on RLBWT of T

- Suppose the next LZ77 phrase ends with $T[\,j\,]$
- Compute $T[i..j]$–interval in decreasing order of $i = j, j-1, \ldots$
- If $T[i..j]$–interval contains a SA-value greater than i, the phrase will continue to grow
- If not, encode the LZ77 phrase



One-pass Algorithm [Nishimoto & Tabei, 2019]

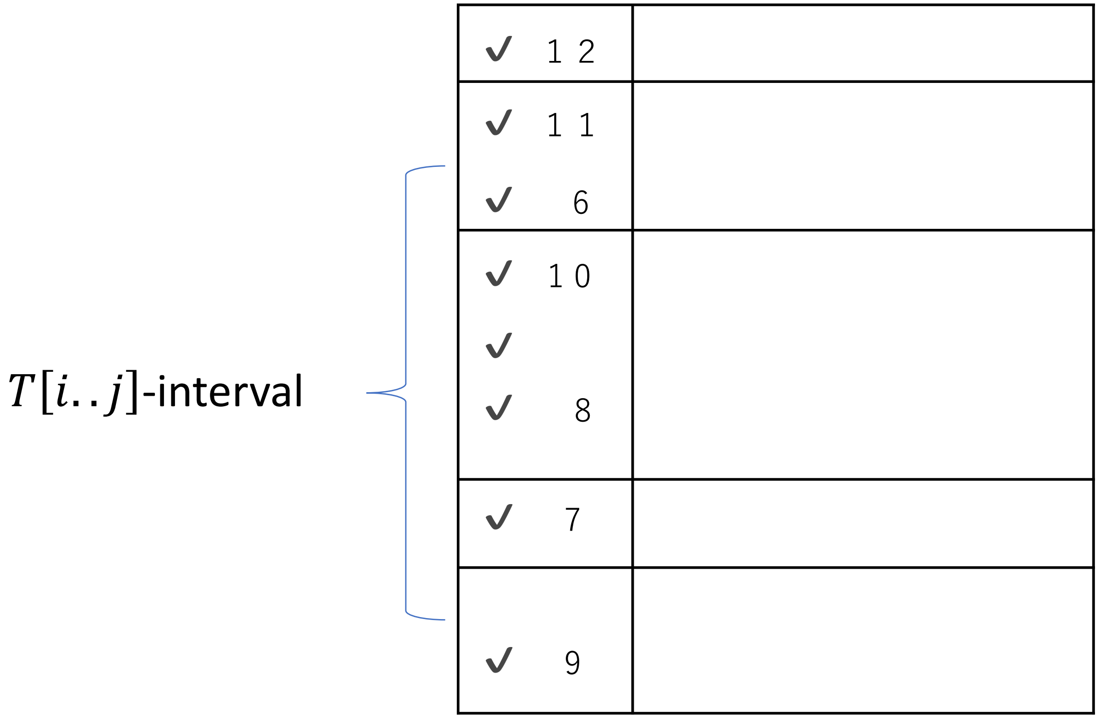
- One-pass algorithm can compute the text position of LZ77 phrase in a single pass while keeping track of two SA-positions and their SA-values per run



One-pass Algorithm [Nishimoto & Tabei, 2019]

- One-pass algorithm can compute the text position of LZ77 phrase in a single pass while keeping track of two SA-positions and their SA-values per run

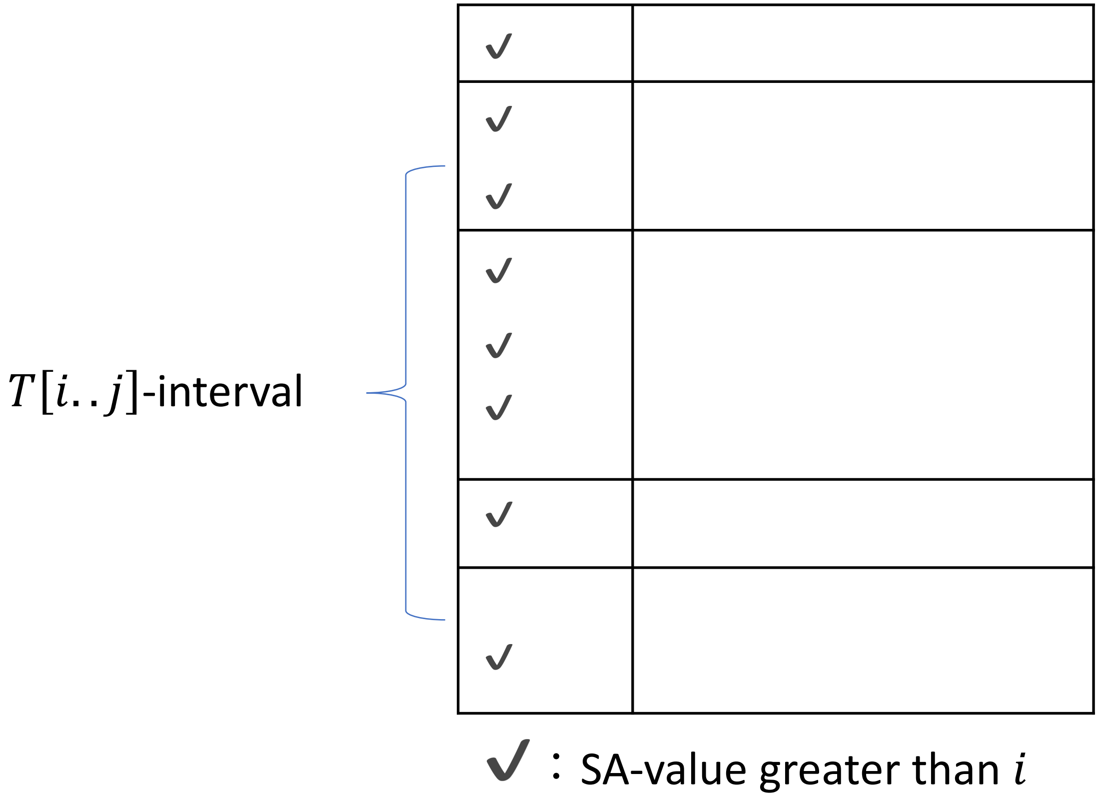
Spend $2r \lg n$ bits
n is the length of T



✓ : SA-value greater than i

Two-pass Algorithm (First pass)

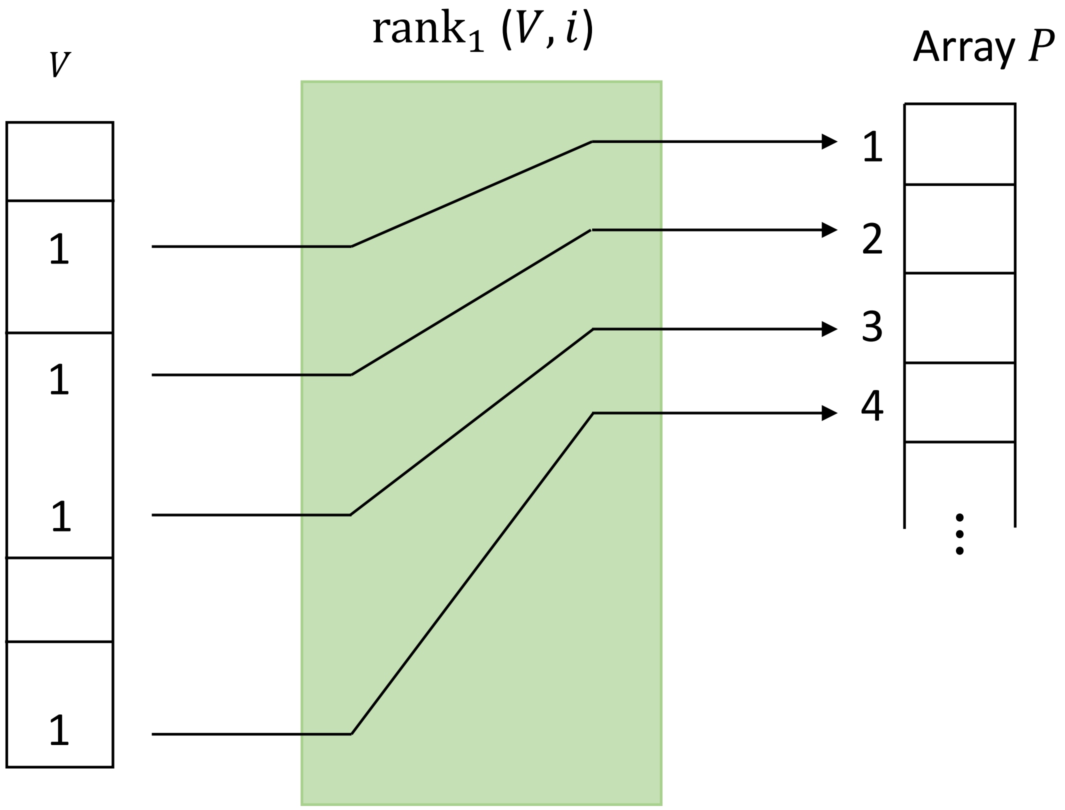
- In the first pass, we keep track of only SA-positions, which are enough to compute λ_h and c_h of LZ77 phrase (π_h, λ_h, c_h), and also compute the sequence $k_1, k_2, \ldots, k_z$ of SA-positions such that SA[k_h] = π_h.
- By discarding SA-values, we can reduce the space by $2r \lg n$ bits



Two-pass Algorithm (Second pass)

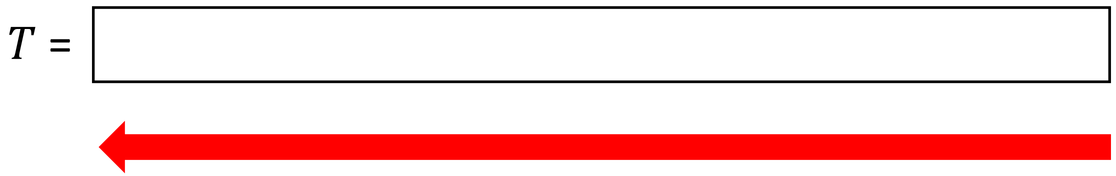
- Build the rank data structure for a bit vector V marking SA-positions
- While visiting SA-positions in the decreasing order of their SA-values, we check if the current SA-position is marked, and if so store the current SA-value (text position) at $P[\mathrm{rank}_1\,(V, i)]$
- Finally, we scan the sequence $\mathrm{k}_1, \mathrm{k}_2, \ldots, \mathrm{k}_z$ and output $\pi_h = P[\mathrm{rank}_1\,(V, \mathrm{k}_h)]$

Return the number of 1's in $V[1..i]$



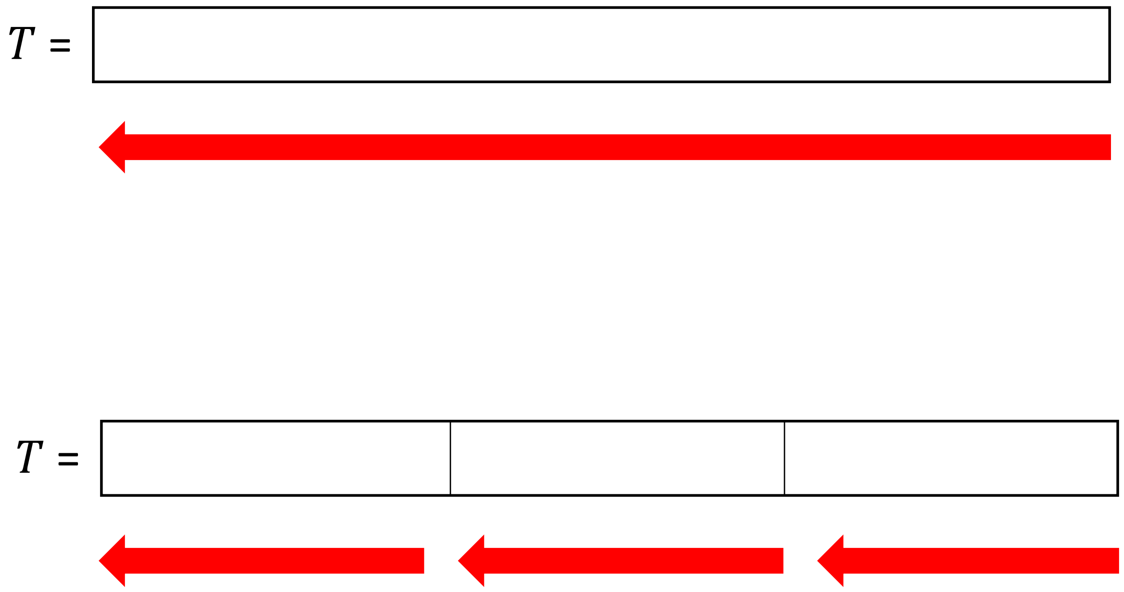
Two-pass Algorithm (Second pass)

- $\pi_h = \mathrm{SA}[\mathrm{k}_h]$ can be computed in the second pass of retrieving SA-positions for suffixes of T from right to left order



Two-pass Algorithm (Second pass)

- $\pi_h = \mathrm{SA}[\mathrm{k}_h]$ can be computed in the second pass of retrieving SA-positions for suffixes of T from right to left order
- The second pass can be conducted in parallel



We compared one-pass and two-pass algorithms on some highly-compressible texts

dataset	Alphabet size	Length of the string	#runs in the BWT	#LZ77 phrases	r-index [KiB]
einstein	139	467,626,544	290,239	75,700	1,084
kernel	160	257,961,616	2,791,368	708,336	8,124
para	5	222,953,928	15,636,740	1,886,379	29,191
chr19x50	6	2,956,259,455	33,139,327	3,798,554	73,499

- r-index : RLBWT + Data structures for backward search

Results Peak memory usage [MiB]

23% to 37% less space

dataset	One-pass	Two-pass	$\frac{\text{Two–pass}}{\text{One–pass}}$
einstein	4.62	3.60	0.779
kernel	38.05	27.53	0.723
para	191.39	119.91	0.626
chr19x50	463.76	289.95	0.625

Results　Computational time [sec]

dataset	one-pass	Two-pass				$\frac{\text{Two–pass}}{\text{One–pass}}$
		#threads	total	first-pass	second-pass	
einstein	853	1	1207	855	352	1.414
		2	1041	860	180	1.219
		4	951	858	93	1.114
		8	936	872	63	1.097
kernel	566	1	778	548	230	1.373
		2	665	546	119	1.174
		4	605	544	60	1.068
		8	584	544	40	1.032
para	924	1	1302	933	368	1.408
		2	1118	930	188	1.209
		4	1039	939	99	1.124
		8	985	923	62	1.066
chr19x50	6930	1	9916	6965	2950	1.430
		2	8533	7044	1488	1.231
		4	7532	6773	758	1.086
		8	7353	6855	498	1.061

Results Computational ti

The overheads of the two-pass algorithm are within 10% when we use 8 threads.

dataset	one-pass	Two-pass				Two−pass / One−pass
		#threads	total	first-pass	second-pass	
einstein	853	1	1207	855	352	1.414
		2	1041	860	180	1.219
		4	951	858	93	1.114
		8	936	872	63	1.097
kernel	566	1	778	548	230	1.373
		2	665	546	119	1.174
		4	605	544	60	1.068
		8	584	544	40	1.032
para	924	1	1302	933	368	1.408
		2	1118	930	188	1.209
		4	1039	939	99	1.124
		8	985	923	62	1.066
chr19x50	6930	1	9916	6965	2950	1.430
		2	8533	7044	1488	1.231
		4	7532	6773	758	1.086
		8	7353	6855	498	1.061

Conclusion

- We propose a two-pass algorithm and show by experiments that it works in 23% to 37% less space with up to 10% increase of computational time when we use 8 threads.

- The reduced space may be used to employ a space-consuming but faster r-index to improve the throughput if processing 3GB text in 2 hours is too slow.