

PSI-RA (Ψ -RA): A parallel Sparse Index for Read Alignment on Genomes

M. Oğuzhan Külekci¹ Wing-Kai Hon² Rahul Shah³
Jeffrey Scott Vitter⁴ Bojian Xu⁴

¹TÜBİTAK-UEKAE, Turkey kulekci@uekae.tubitak.gov.tr

²National Tsing Hua University, Taiwan wkhon@cs.nthu.edu.tw

³Louisiana State University, USA rahul@csc.lsu.edu

⁴The University of Kansas, USA [{jsv, bojianxu}@ku.edu">{jsv, bojianxu}@ku.edu](mailto)

The IEEE International Conference on Bioinformatics and
Biomedicine, Hong Kong, 2010

Outline

- 1 The Problem**
 - Genome Indexing
 - Read Alignment
- 2 Sparse Suffix Arrays**
 - Construction
 - Search on Sparse-SA
 - Approximate Matching via Sparse Suffix Arrays
- 3 Implementation & Results**
 - Implementation
 - Results
 - Conclusions

Genome Indexing

DNA sequences of complex (?) organisms are not short.

- ≈ 3 billion ($2^{31.53}$) base pairs in the whole genome of a human being.
- 2.8 GB in raw text format, 700 MB in 2-bit format.

Fast search on massive data requires indexing.

- Word-level indexing (**inverted index**) is not appropriate since word notion is (*at least yet*) not meaningful on biological sequences.
- We need **full-text** indexing!

Genome Indexing

Major methods in genome indexing.

- Suffix trees and arrays (most space-consuming!)
- Burrows–Wheeler Transform (most popular)
- q-grams and spaced seeds (most accurate)

Two main problem in genome indexing!

- Space-efficient index
(should fit into the main memory of a PC)
- **approximate** search capability.

Read Alignment

DNA sequencing

- Sequencing machines produce millions of **reads**, which are small arbitrary fragments of the whole DNA
 - *De novo* sequencing → Assembly problem
 - Re-sequencing → **Read alignment problem**

Read Alignment Problem

- Map possible occurrences of some millions of **reads** onto a reference genome sequence given *a priori*.
- Should be fast and accurate! (*however, in practice there is always a trade off between speed and accuracy.*)
- Mainly a multiple string matching problem.

Read Alignment

Several choices...

- Indexing versus serial scan over reference genome
- Index the reference genome or the reads
- Find only exact matches or consider also the approximate matches
- Machine requirements (design for a cluster or a single personal computer)

Some of the Previous Read Aligners

- BWT based: Bowtie, BWA, SOAP2,
- q-gram indexing: mrsFast
- Spaced seed based: ZOOM, SHRiMP...
- Limited to exact matching: MPScan

This study...

Aim

- Index genome via **sparse** suffix arrays as an alternative to other solutions.
- Provide approximate matching in sparse suffix array indexing and apply this to read alignment problem.

Motivation

- A flexible index scheme that lets the user to tune according to available memory, trade memory versus time.
- Capability to be parallelized since it is the time of multi-core processors.

Suffix arrays

$T[1 \dots 32] = \text{CGGATTCGATTAAAGCTCGATAGGAATTCGAA}$

$\text{suffix}_i = T[i \dots n]$	i
CGGATTCGATTAAAGCTCGATAGGAATTCGAA	1
GGATTCGATTAAAGCTCGATAGGAATTCGAA	2
GATTCGATTAAAGCTCGATAGGAATTCGAA	3
.....	...
GAA	30
AA	31
A	32

Suffix Arrays

(Lexicographically Sorted) $suffix_i = T[i \dots n]$	i
A	32
AA	31
AAAGCTCGATAGGAATTCGAA	12
AAGCTCGATAGGAATTCGAA	13
AATTCGAA	25
.....	...
TCGATTAAAGCTCGATAGGAATTCGAA	6
TTAAAGCTCGATAGGAATTCGAA	10
TTCGAA	27
TTCGATTAAAGCTCGATAGGAATTCGAA	5

Suffix Arrays

(Lexicographically Sorted) $suffix_i = T[i..n]$	i
A	32
AA	31
AAAGCTCGATAGGAATTCGAA	12
AAGCTCGATAGGAATTCGAA	13
AATTCGAA	25
.....	...
TCGATTAAAGCTCGATAGGAATTCGAA	6
TTAAAGCTCGATAGGAATTCGAA	10
TTCGAA	27
TTCGATTAAAGCTCGATAGGAATTCGAA	5

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$T[i]$	C	G	G	A	T	T	C	G	A	T	T	A	A	A	G	C	T	C	G	A	T	A	G	G	A	A	T	T	C	G	A	A
$SA[i]$	32	31	12	13	25	14	22	20	09	26	04	29	18	07	01	16	30	24	19	08	03	15	23	02	11	21	28	17	06	10	27	05

Basics of Suffix Arrays

Properties...

- Suffix array occupies $n \cdot \log n$ bits for text $T = t_1 \dots t_n$. Efficient construction algorithms exist to build it in linear time.
- Search for a pattern P of length m can be done in $O(m \cdot \log n)$ time by running a binary search over SA. Can further be improved to $O(m + \log n)$ time by using additional auxiliary data structures.
- Easy to use, but space consumption is not small.
- For human genome, it requires around 12GB memory when we use 4-byte integers to represent each $SA[i]$, where $1 \leq i \leq 3$ billion.

Space-efficient Suffix Arrays

Reducing the space consumption of SA in genome indexing

- How about using compressed suffix arrays?
Wrong way! Compressed indexes are meaningful only when the text itself is compressible!
- What else can we do?
SPARSIFICATION! (*Chin et al., DCC'08*)

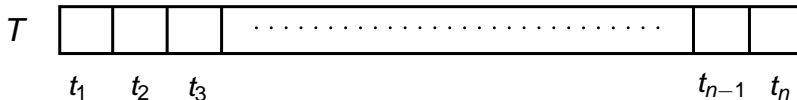
Sparse Suffix Arrays

Sparsification

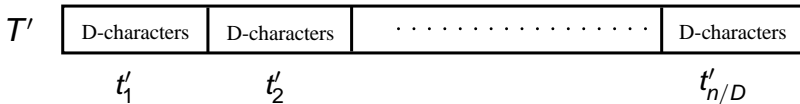
- Instead of keeping the whole suffix array, save only those positions that $SA[i] \equiv 1 \pmod{D}$, where D is the sparsification factor.
- Equivalent to creating suffix array for T' instead of T such that

$$T = t_1 t_2 \dots t_n \rightarrow T' = t'_1 t'_2 \dots t'_{n/D},$$
 where meta-character $t'_i = t_{(i-1) \cdot D + 1} \dots t_{i \cdot D}$.
- Regular suffix array of T has n numbers, where sparse suffix array of T has n/D numbers.
- Thus, space consumption decreases to $O((n/D) \cdot \log(n/D))$.
- Gain is not free! We will pay the price in search phase.

Sparse Suffix Array



$$SA(T) \rightarrow O(n \cdot \log n)$$



$$SA(T') \rightarrow O((n/D) \cdot \log(n/D))$$

Sparse Suffix Arrays

$D = 1$, original SA requires $5 \cdot 32 = 160$ bits

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$T[i]$	C	G	G	A	T	T	C	G	A	T	T	A	A	A	G	C	T	C	G	A	T	A	G	G	A	A	T	T	C	G	A	A
$SA[i]$	32	31	12	13	25	14	22	20	09	26	04	29	18	07	01	16	30	24	19	08	03	15	23	02	11	21	28	17	06	10	27	05

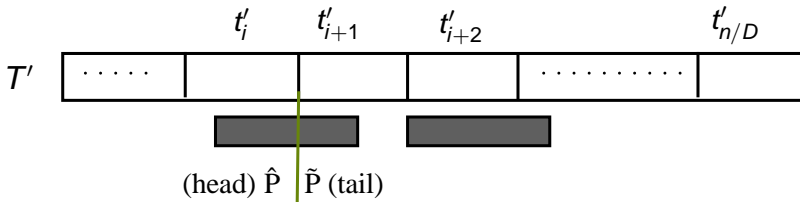
$D = 2$, sparse SA requires $4 \cdot 16 = 64$ bits

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T'[i]$	CG	GA	TT	CG	AT	TA	AA	GC	TC	GA	TA	GG	AA	TT	CG	AA
$SA[i]$	16	7	13	5	15	4	1	10	2	8	12	6	11	9	14	3
	(31)	(13)	(25)	(9)	(29)	(07)	(01)	(19)	(03)	(15)	(23)	(11)	(21)	(17)	(27)	(05)

$D = 4$, sparse SA requires $3 \cdot 8 = 24$ bits

i	1	2	3	4	5	6	7	8
$T'[i]$	CGGA	TTCG	ATTA	AAGC	TCGA	TAGG	AATT	CGAA
$SA[i]$	4	7	3	8	1	6	5	2
	(13)	(25)	(09)	(29)	(01)	(21)	(17)	(05)

Exact Matching via Sparse Suffix Arrays



We need to run the search procedure D times!

- We can only find the occurrences of a given pattern P beginning from meta-character boundaries.
 If P begins at position s such that $s \equiv 1 \pmod{D}$, it can be detected by standard range search over sparse-SA.
- What to do not to miss the other possible occurrences?
- Split the pattern, and run the range search D times.

Exact Matching via Sparse Suffix Arrays

Search TCGA on our sample text T that is indexed with $D = 4$.

i	1	5	9	13	17	21	25	29
$T[i]$	CGGA	TTCG	ATTA	AAGC	TCGA	TAGG	AATT	CGAA
$SA[i]$	13	25	09	29	01	21	17	05

\hat{P}	\tilde{P}	Search \tilde{P}	Verify \hat{P}	Match
	TCGA	$T[17]$		✓
T	CGA	$T[29]$	$T[28]=T$	✓
TC	GA	-		x
TCG	A	$T[9]$	$T[6-8]=TCG$	✓
		$T[13]$	$T[10-12]=TTA$	x
		$T[25]$	$T[22-24]=AGG$	x

Search Complexity on Sparse SA

Cost of having small index

- Sparsifying suffix array with a degree of D gives us $O((n/D) \cdot \log(n/D))$ space index.
- Good, but nothing is free! Now we need to run the range search over sparse SA D times.
- At each step, we perform the verification process to see if the preceding characters match with the head part \hat{P} of pattern P .
- The search time complexity becomes $O(D \cdot m \cdot \log(n/D) + \textit{Verification})$.
- Good news is each step can be run independently. Thus, gives us a very good basis for parallelization in multi-core architectures. (*...will visit later*)

Short Pattern Problem

What to do if $m < D$?

- Search procedure works if the occurrence of P passes through a meta-character boundary.
- Select D larger than the least possible length.
- For shorter length patterns, there are ways to handle (please refer Chin et al, DCC'08).
- No need to worry in read alignment problem since we can set a reasonable minimum length easily.

Approximate Matching on Sparse Suffix Arrays

Approximate Matching

- Approximation is always needed in DNA sequence search.
- Although we have many effective indexing schemes, approximate search is not easy.
- We will define approximate search on suffix arrays based on prioritizing the errors towards the end of the pattern.
- In general errors are more frequent towards the end.

Rightmost Mismatch Precedence in Approximate Search

- Approximate occurrences that have same number of mismatches are sorted according to the position of the mismatches.
- The occurrences having the rightmost errors are higher importance.

i)

			x		x	x	
--	--	--	---	--	---	---	--

 1 1 1 0 1 0 0 1 → 233

ii)

			x	x			x
--	--	--	---	---	--	--	---

 1 1 1 0 0 1 1 0 → 230

iii)

				x	x		x
--	--	--	--	---	---	--	---

 1 1 1 1 0 0 1 0 → 241

Precedence is **iii** > **i** > **ii** according to RMM criteria.

Longest Matching Prefix of P

- Search on suffix array gives us a range $R = \langle sp, ep \rangle$ indicating pattern P begins at positions $t_{SA[sp]}$, $t_{SA[sp]+1}$, \dots , $t_{SA[ep]}$.
- If there are no matches, then $ep = sp - 1$.
- The closest match of P on T begins at either $t_{SA[sp]}$ or $t_{SA[ep]}$ (by definition of binary search on sorted suffixes).
- The **longest matching prefix** of P is obtained by comparing P against $t_{SA[sp]}$ and $t_{SA[ep]}$.

1-mismatch Case

When SA range search for AGGTCGATTCGGGACC gave empty interval...

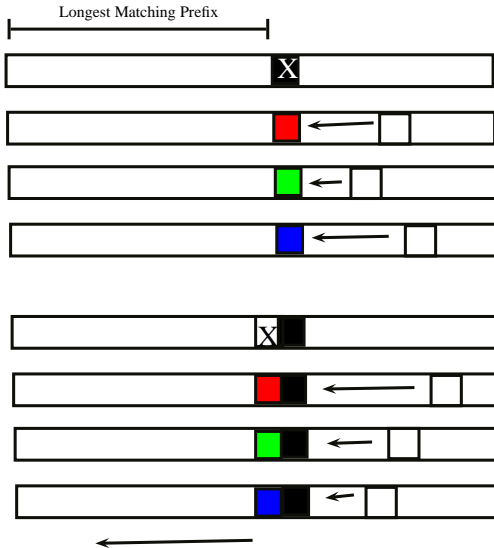
- Assume LMP is AGGTCGATTC ($|LMP| = L = 10$), which means AGGTCGATTC**G**, AGGTCGATTC**GG**, AGGTCGATTC**GGG**, AGGTCGATTC**GGGA**, AGGTCGATTC**GGGAC**, AGGTCGATTC**GGGACC** do not exist in the sequence.
- No change after position $L + 1 = 11$ can report a match, e.g. AGGTCGATTCGGTACC cannot appear since we know AGGTCGATTCGG is missing.
- The first meaningful alteration on the read is to be done on the eleventh position resulting the possibilities AGGTCGATTC**A**GGACC, AGGTCGATTC**C**GGACC, AGGTCGATTC**T**GGACC.
- For 1-mismatch alignment, we test the alterations at position $L + 1, L$, down to 1 according to rightmost mismatch criteria.

1-mismatch Case

$$P = p_0 p_1 \dots p_{m-1}, \text{tail}_i^P = p_0 \dots p_{m-1}$$

- 1: **for all** $0 \leq i < D$ **do**
- 2: $L \leftarrow \text{LongestPrefix}(\text{tail}_i^P, \text{SSA}, G)$;
- 3: **for** $j = L$ down to 0 **do**
- 4: **for** $a = 0$; $a < 3$; $a++$ **do**
- 5: $\text{newP} \leftarrow P$;
- 6: $\text{newP}[j] \leftarrow q_a$; $\{q_0, q_1, \text{ and } q_2 \text{ are the three bases other than } P[j]\}$
- 7: add $\langle \text{newP}, j, i \rangle$ to the α -list

2-mismatch Case



2-mismatch Case

- We have collected 1-mismatch possibilities in α -list.
- We can generate possible 2-mismatch alignments from α -list.

for all item i in α -list **do**

$\langle pat, mmposlist, offset \rangle \leftarrow \alpha\text{-list}[i];$

$Z \leftarrow \text{LongestPrefix}(tail_{offset}^{pat}, SSA);$

for $j = Z$ down to $mmposlist[mm - 1]$ **do**

for $a = 0; a < 3; a++$ **do**

$newP \leftarrow pat;$

$newP[j] \leftarrow q_a;$

add $\langle newP, (mmposlist, j), offset \rangle$ to the β -list

Extending to k -mismatch Case

k -mismatch case is the 1-mismatch of $(k - 1)$ -mismatches.

- We can proceed up to any desired value of k in the same way.

for $mm = 2; mm \leq k; mm++$ **do**

for all item i in α -list **do**

$\langle pat, mmposlist, offset \rangle \leftarrow \alpha\text{-list}[i];$

$Z \leftarrow \text{LongestPrefix}(tail_{offset}^{pat}, SSA);$

for $j = Z$ down to $mmposlist[mm - 1]$ **do**

for $a = 0; a < 3; a++$ **do**

$newP \leftarrow pat;$

$newP[j] \leftarrow q_a;$

add $\langle newP, (mmposlist, j), offset \rangle$ to the β -list

$\alpha\text{-list} \leftarrow \beta\text{-list};$

Final Search for k -mismatch Alignments

Sort α -list according to rightmost criteria;

for all item i in *alpha*-list **do**

$\langle pat, mmposlist, offset \rangle \leftarrow \alpha\text{-list}[i];$

$R \leftarrow \text{SuffixRangeSearch}(SSA, tail_{offset}^{pat});$

$\{R = \{k_1 \cdot D, k_2 \cdot D, \dots, k_\ell \cdot D\}\}$

for $j = 1 ; j \leq \ell ; j++$ **do**

if $head_{offset}^{pat} = t_{k_j \cdot D - offset} \dots t_{k_j \cdot D - 2} t_{k_j \cdot D - 1}$ **then**

report match at position $k_j \cdot D - offset;$

Implementation

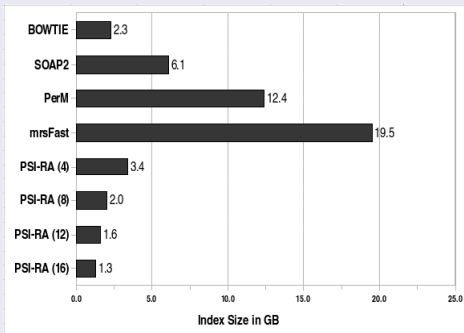
- Sparse SA construction via Yuta Mori's implementation of SAIS algorithm
(sites.google.com/site/yuta256/sais).
- D is a multiple of 4 to make everything byte operable.
- Actual index values in sparse-SA are stored as 32-bit integers (as oppose to $\log(n/D)$ bits in theory.)
- We **partition** the suffix array to speed up the range search on SA. The starting and ending positions of individual 8-grams are kept in a table ($4^8 = 64K$ rows). So the binary range search is done in $O(\log(n/4^8))$ instead of $O(\log n)$ complexity.
- Algorithm is implemented in an optimized way (a bit different then the representation).
- The aligner is available at
www.busillis.com/o_kulekci/PSIRA.zip.

Experimental Setup

- Target genome is the complete human genome of GRCh37.
- GRCh37 is indexed via proposed indexing method with sparsification factors of 4, 8, 12, and 16.
- Randomly selected 100K reads of length 30 to 400 from SRR003078 are used in experiments.
- Each measurement is done by repeating the same procedure several times, and the average values are reported.
- Q-gram or spaced-seed based indexing are powerful in approximate matching, but their speed is relatively slow when compared with BWT based aligners. Thus, given comparisons are mainly against BWT based indexing solutions.

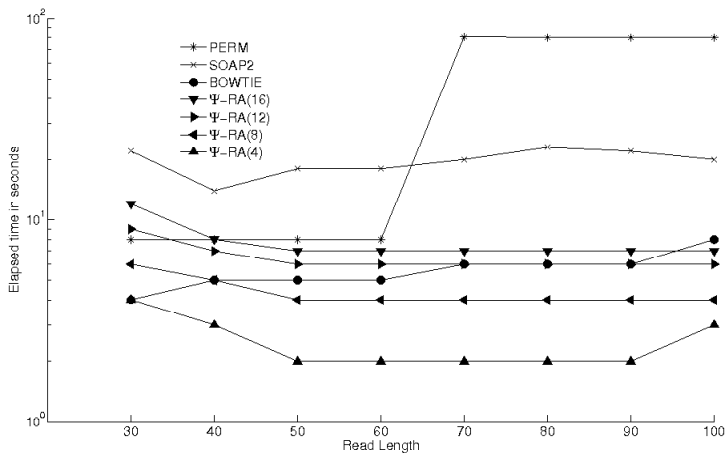
Index Size Comparison

Indexing human reference genome GRCh37 via various aligners.

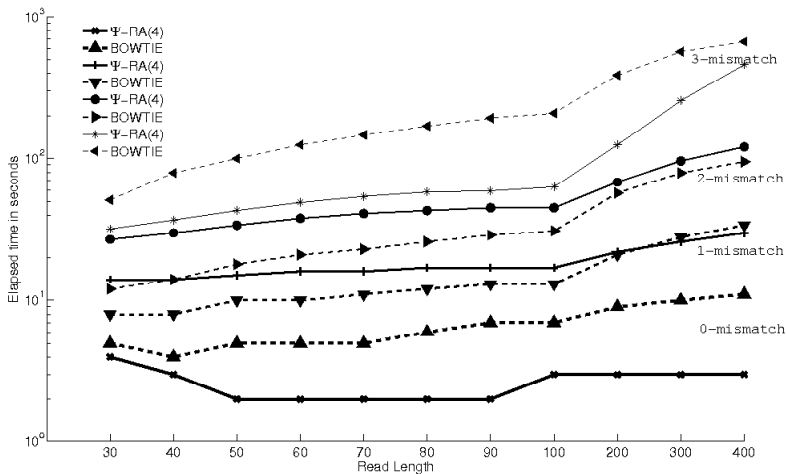


Note that ≈ 700 MB of the index is the 2-bit encoded sequence in $\Psi - RA$ values.

Exact Match Performance



Approximate Match Performance



Effect of Parallelization

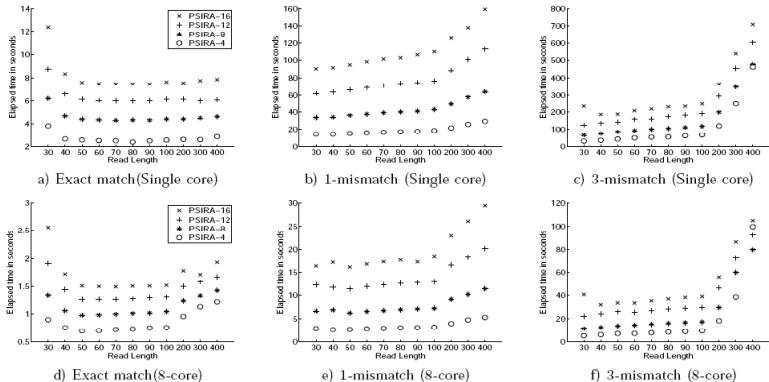


Figure 3. The effect of sparsification factor D on single thread versus eight thread executions.

BWT based indexing versus (sparse) SA indexing

- BWT is not cache friendly!
- Thus, although the theoretic time complexity of search on SA is larger than that of backwards search on BWT, it is experimentally shown that exact matching performance is much better via (even sparse) SA.
- Being cache oblivious is a great advantage in practice.

Contributions of the study

Flexible indexing

- Size of the index can be tuned according to the available memory by changing the sparsification factor.
- Trade memory against time

Contributions of the study

Parallelization

- Data level parallelism is always possible by distributing the queries to processors (multi-threaded execution option is included in nearly all aligners).
- The search procedure on a sparse SA is composed of D **independent** pieces each of which can be assigned to a processor without any prerequisite.
- Thus, it is possible to parallelize even a single query in a multi-core architecture. Although that does not make much sense in read alignment since we have millions of reads, it is more meaningful in single search operations.

Contributions of the study

Approximate search capability on sparse SA

- Approximate matching on sparse suffix arrays is defined.
- The mismatches are returned in order of their precedence according to right-most mismatch criteria.

Thank you!

Questions?