

PSI-RA: A Parallel Sparse Index for Read Alignment on Genomes

M. Oğuzhan Külekci*, Wing-Kai Hon†, Rahul Shah‡, Jeffrey Scott Vitter§ and Bojian Xu§

*National Research Institute of Electronics & Cryptology, Turkey. kulekci@uekae.tubitak.gov.tr

†Dept. of Computer Science, National Tsing Hua University, Taiwan. wkhon@cs.nthu.edu.tw

‡Dept. of Computer Science, Louisiana State University, USA. rahul@csc.lsu.edu

§Information and Telecommunication Technology Center, University of Kansas, USA. {jsv, bojianxu}@ku.edu

Abstract—We concentrate on indexing DNA sequences via sparse suffix arrays (SSAs) and propose a new short read aligner named PSI-RA (parallel sparse index read aligner). The motivation in using SSAs is the ability to trade memory against time. It is possible to tune the space consumption of the index based on the available memory of the machine and the minimum length of the arriving pattern queries. Although SSAs have been studied before on exact matching of short reads, an elegant way of approximate matching capability was missing. We provide this by defining the right-most mismatch criteria that prioritizes the errors towards the end of the reads since it is known that the errors are more probable at that area. PSI-RA supports any number of mismatches in aligning reads. We give comparisons with some of the well known short read aligners, and show that indexing genome with SSA is a good alternative to Burrows-Wheeler transform or seed based solutions.

Keywords—short read alignment, genome indexing, sparse suffix array

I. INTRODUCTION

Last decade witnessed a rapid development in DNA sequencing by the introduction of next generation high-throughput DNA sequencing [1] technologies. The equipments based on that new technology produce billions of reads in a single day per machine [2]. The most important two problems regarding the DNA sequencing are alignment and assembly [3]. If the target specie has not been sequenced before, a *de novo* DNA assembly [4], which requires concatenation of the reads in an optimum way, has to be performed. Otherwise, reads are mapped against a reference genome, which is the result of a previous sequencing effort of the same specie. With the advent of the next-generation sequencing, various short read aligners such as Bowtie [5], PerM [6], SOAP [7], mrsFast [8], and many others have been proposed in the last three years. In a recent study, Li and Homer [9] surveyed short read aligners in general.

Many strategies have been applied to perform the alignment process fast and accurate. While some of the aligners index the reference genome, some others rely on hash tables based on q-grams or spaced seeds to perform a

quick scan. Although hash based solutions are more flexible in detecting approximate matches, indexing solutions are faster. The dominant solution in genome indexing is the Burrows-Wheeler transform (BWT) [10] of the reference sequence (e.g., [11], [5]) that the reads are searched with the backwards search algorithm introduced in FM-index of Ferragina and Manzini [12].

We concentrate on indexing the genome via sparse suffix arrays (SSAs). Lexicographic ordering of all the suffixes of the text forms the suffix array [13], which is a well-studied data structure initially proposed to lower the space requirement of the suffix tree. Although suffix arrays are much more space preserving than suffix trees, they still require large memory space on indexing massive data such as the whole genome of a human. The space consumption of an ordinary suffix array for a given text of n characters is $O(n \cdot \log n)$. The occurrences of a pattern of length m characters can be found in $O(m \cdot \log n)$ time by searching the pattern on the suffix array by binary search procedure. This bound can further be improved to $O(m + \log n)$ time by using auxiliary data structures [13].

It is believed for a long time that compressing a suffix array is not feasible since it is mainly a permutation of the numbers from 1 to n . Grossi and Vitter [14] broke that belief and showed that suffix arrays are compressible. Following that work, compressed data structures in text indexing gained great focus from the community [15], [16]. These efforts resulted that an index for a text can occupy space proportional to the compressed size of the text itself. However, if it is not possible to significantly compress the input data, as is the case for DNA sequences, these methods do not provide a considerable advantage.

Recently, Chien *et al.* [17] proposed *sparsification* of the suffix array as an alternative method to compress it. The key idea is instead of sorting all the suffixes beginning from each position 1 to n of the text, keeping a sorted list of the suffixes than begin at positions p , such that $p = 0 \bmod D$, for all $1 \leq p \leq n$, where D denotes the sparsification factor. As a result, we have a list of n/D numbers in the sparse suffix array rather than n numbers in the original suffix array. We can think that we interpret the text with a new alphabet by combining the D subsequent original characters

This research was supported in part by US NSF grant CCF0621457. Part of the work was done while the first, fourth, and fifth authors were with Texas A&M University.

into a meta-character. The length of the text is now n/D meta-characters. Thus, the space consumption decreases to $O((n/D) \cdot \log(n/D))$.

The main two drawbacks of the sparse suffix arrays are i) the necessity to run the search procedure D times, and ii) complicated search procedure when the queried pattern length is less than D . The necessity to run search procedure D times comes from the fact that the queried pattern may begin at some position s , where $s = \{0, 1, \dots, (D-1)\} \bmod D$. Thus, appropriate alignment positions should be checked one by one (see section 2 for more detailed explanation).

When the pattern length is shorter than the sparsification factor D , then the meta-characters, which are of D ordinary characters in length, including the queried pattern must be investigated first. Specified meta-characters should then be located on the text to finalize the actual search (see [17] for detailed analysis of the case), which is not so elegant.

On the other hand, we have two advantages to overcome these drawbacks, in particular for the short read alignment problem. First, today's multi-core processor architecture enables parallelism more than ever. We can benefit from multi-core architecture to decrease the overhead in repeating the search procedure D times. Second, next-generation sequencing machines have a lower bound on the length of the reads. Short reads are in range from 25bp to 100bp, and it is foreseen that lengths will be longer in a couple of years. Thus, if we choose the sparsification factor D less than the minimum possible length of the input reads, we do not need to deal with short pattern case anymore.

Recently, Khan *et al.* [18] proposed using sparse suffix arrays for finding maximal matches in large sequence data along with an application on short read alignment. Their study considers only exact matching of the reads. Although it is argued in some studies [19] that exact matching would be enough for short read alignment, an approximate alignment of a queried pattern would help on reducing the total number of required reads (*coverage*) for whole sequencing. That is because the percentage of aligned reads will be lower if we neglect approximate matches.

Based on the fact that errors are more probable towards the end of the reads, we extend exact matching with sparse suffix arrays to include any number of mismatches by defining the right-most mismatch criteria. We prioritize the errors on the right hand side of the reads, and detect the k -mismatch alignments sorted according to their right-most occurrences in an elegant way. When equipped with mismatch detection capability, sparse suffix arrays serve as a good alternative to BWT type genome indexing. We show that by experimentally comparing the proposed sparse suffix arrays against BWT type genome indexing.

Section 2 describes the the building blocks of the proposed scheme. Section 3 focuses on the implementation details of the PSI-RA aligner. Section 4 includes some experimental comparisons and discusses the results. We finalize the paper

```

      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
G = A G G T C G A T T C G G G A C C $
A = 13 0 6 15 14 4 9 12 5 11 10 1 2 3 8 7
SSA = 0 4 12 8

```

Figure 1. An example DNA sequence G , suffix array of G , and sparse suffix array of G by assuming sparsification factor $D = 4$.

with concluding remarks.

II. METHODOLOGY

Let an n -base long DNA sequence be $G = g_0g_1g_2 \dots g_{n-1}\$$, where each base g_i is from alphabet $\Sigma = \{A, C, G, T\}$ and the end of the sequence is marked with a special character $\$$ that is lexicographically smaller than all the characters in Σ . If we denote the n suffixes of such a given sequence by s_0 to s_{n-1} , then the i^{th} suffix s_i will correspond to $s_i = g_i g_{i+1} \dots g_{n-1}\$$. The lexicographical sorting of those suffixes creates the suffix array $A = a_0 a_1 \dots a_{n-1}$ such that a_i is the beginning position of the i^{th} smallest suffix of G for $0 \leq i < n$.

The sparse suffix array (SSA) of G includes only those a_i numbers in suffix array A , which are multiples of a specific number D as $a_i = 0 \bmod D$. In other words, rather than sorting all the suffixes of G , we sort only those suffixes s_i , where $i = 0 \bmod D$ and keep their starting positions in SSA. We refer D as the sparsification factor. Figure 1 depicts the suffix array and the sparse suffix array of an example sequence AGGTCGATTCGGGACC. The first element of the suffix array is $a_0 = 13$ as $s_{13} = ACG\$$ is the smallest among all suffixes of G . When we want to sparsify that suffix array with a factor of $D = 4$, we take only those points from A , where $a_i = 0 \bmod 4$. We preserve their order of appearance in the original suffix array. Thus, the sparse suffix array of given G is $SSA = \{0, 4, 12, 8\}$.

The space required by a sparse suffix array of a sequence of length n characters is $O((n/D) \cdot \log(n/D))$. When $D = 1$, this complexity converges to the ordinary suffix array. The decrease in the size of the index comes with a cost on the search complexity.

The time complexity to locate all possible occurrences of queried pattern P on a sequence of length n via ordinary suffix array is $O(|P| \cdot \log n)$. However, when we have the sparse suffix array, the same procedure only lets us detect those occurrences starting at positions that are multiples of D as SSA includes the sorted list of the suffixes beginning only on those locations. We also need to consider the cases that P starts at some position p which is not divisible by D .

Assuming we are given a pattern $P = p_0 p_1 \dots p_{m-1}$, let $head_i^P = p_0 p_1 \dots p_{i-1}$ denote the initial i characters and $tail_i^P = p_i p_{i+1} \dots p_{m-1}$ show the rest. If we search $tail_i^P$ on G via the SSA for each $0 \leq i < D$, and then verify whether G has the corresponding $head_i^P$ preceding

the positions, where $tail_i^P$ appears, we become able to locate all possible occurrences. Algorithm 1 describes this procedure.

Algorithm 1 ExactMatch(P, G, SSA, D)

Require: $D \leq |P|$

- 1: **for all** $0 \leq i < D$ **do**
- 2: $head_i^P \leftarrow p_0 p_1 \dots p_{i-1}$;
- 3: $tail_i^P \leftarrow p_i p_{i+1} \dots p_{m-1}$;
- 4: $R \leftarrow \text{SuffixRangeSearch}(SSA, tail_i^P)$; $\{R$ holds the positions on G matching with $tail_i^P$, as $R = \{k_1 \cdot D, k_2 \cdot D, \dots, k_\ell \cdot D\}$
- 5: **for** $j = 1$; $j \leq \ell$; $j++$ **do**
- 6: **if** $head_i^P = g_{k_j \cdot D - i} \dots g_{k_j \cdot D - 2g_{k_j \cdot D - 1}}$ **then**
- 7: Pattern detected at position $k_j \cdot D - i$;

The time complexity of exact match via SSA is $O(m \cdot \log(n/D) + (m-1) \cdot \log(n/D) + o_1 \cdot 1 + (m-2) \cdot \log(n/D) + o_2 \cdot 2 + \dots + (m-D-1) \cdot \log(n/D) + o_{D-1} \cdot (D-1))$, where o_i is the number of occurrences of $tail_i^P = p_i p_{i+1} \dots p_{m-1}$ on G . The terms beginning with o_i in the summation corresponds to the verification cost. Thus, total time may be approximated as $O(D \cdot m \cdot \log(n/D) + Verification)$. Note that increasing sparsification factor D decreases the space complexity while increasing the time complexity. This gives us the flexibility to tune D according to the available processor and memory.

A. Aligning with mismatches

Reads may contain some errors. Although today’s sequencing machines produce the quality values that represent the confidence of the individual bases in each read, these values vary greatly and require a fine tuning step to be integrated in the alignment process, and using these quality values also decreases the speed of the alignment [9]. In general the errors are more probable towards the end of the reads. Hence, we define the right-most criteria to prioritize these positions and find the k -mismatch approximate matchings of a given pattern.

Let a given pattern $P = p_0 p_1 \dots p_{m-1}$ be aligned with text segment $g_i g_{i+1} \dots g_{i+m-1}$ for some $0 \leq i \leq (n-m)$, and $B = b_0 b_1 \dots b_{m-1}$ be an m -bit binary number such that, for all j , $0 \leq j < m$, if $p_j = g_{i+j}$, then $b_j = 1$, else $b_j = 0$. The total number of 0-bits in B is the total number of mismatches between P and $G[i \dots i+m-1]$. Among the possible k -mismatch alignments of P onto text G , the ones having the highest B number are defined as the right-most k -mismatch alignments of P . Figure 2 depicts this definition.

When we search a pattern by using the suffix array, the process returns a range $R = (sp, ep)$ that means P starts at text positions $g_{A[sp]}$, $g_{A[sp+1]}$, \dots , $g_{A[ep]}$. If there are no occurrences of P on G , then ep is one less than sp

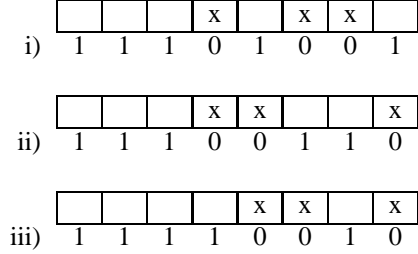


Figure 2. Positions marked with ‘x’ indicate mismatch. When sorted according to the rightmost mismatch criteria, iii) has the first priority, followed by i) and ii) respectively.

(see suffix range search procedure on [16] for more detail). In that case, either the $G[A[sp] \dots A[sp] + m - 1]$ or $G[A[ep] \dots A[ep] + m - 1]$ have the longest match with P . That is because the suffix array is a sorted list of the suffixes, and since we are running a binary search on the suffix array, the final position is the closest point to the pattern, if not itself exactly. We compare $P[0 \dots m - 1]$ against both, and find for each the number of matching nucleotides from the left side until we encounter the first mismatch. We denote the maximum of these two numbers as L meaning the length of the longest matching prefix of P . That indicates $P[0 \dots j]$ does not occur in the text for all $j \geq L$, while the prefixes $P[0 \dots i]$ exist for all $i < L$.

For example, let’s assume sample pattern AGGTCGATTCGGGACC does not occur in the reference genome, and we detect that the length of the longest matching prefix is $L = 10$, which indicates the prefix AGGTCGATTC exists in the text, but the longer prefixes AGGTCGATTCG, AGGTCGATTCGG, \dots , AGGTCGATTCGGGACC do not. If we attempt to alter the last base ‘C’ to any of the ‘A’, ‘G’, or ‘T’, none of them will report a match since we know the non-existence of its prefix AGGTCGATTCGGGAC. Similarly, changing the previous base ‘C’ also does not make sense as AGGTCGATTCGGGA is absent in the reference. The first base that has a chance to match when altered is $p_{10} = 'G'$ preceded by the prefix AGGTCGATTC that occurs in the text. Thus, when looking for the 1-mismatch alignment of the sample read, we don’t need to spend time to check possible alternatives of the last five bases.

Remembering the fact that k -mismatch alignments are 1-mismatch apart from the $(k-1)$ -alignments, we can generate the 2-mismatch patterns from the 1-mismatch cases, and keep going up to k -mismatch alignments.

Algorithm 2 depicts the k -mismatch idea in pseudo-code. We keep possible alterations of original P in a list (α -list and β -list in lines 7 and 16) consisting of three attributes. The first one is the altered pattern. The second attribute is the list of the altered positions that are used in sorting the alterations according to rightmost mismatch criteria. Note that this inner list contains k numbers for k -mismatch alterations as a k -

mismatch occurrence requires to change k positions. The third one is the offset indicating the length of the *head* of P as it is required in sparse suffix search process. After initializing the α -list for the 1-mismatch case within the first `for` loop, we generate the possible k -mismatch alterations of P sorted according to the rightmost mismatch criteria by the second `for` loop, and then search the exact occurrences of those altered patterns on G by using the proposed *SSA* index.

Algorithm 2 k -Mismatch(P, G, SSA, D, k)

```

1: for all  $0 \leq i < D$  do
2:    $L \leftarrow \text{LongestPrefix}(\text{tail}_i^P, SSA, G)$ ;
3:   for  $j = L$  down to  $0$  do
4:     for  $a = 0; a < 3; a++$  do
5:        $\text{newP} \leftarrow P$ ;
6:        $\text{newP}[j] \leftarrow q_a$ ;  $\{q_0, q_1, \text{ and } q_2 \text{ are the three}$ 
7:          $\text{bases other than } P[j]\}$ 
8:       add  $\langle \text{newP}, j, i \rangle$  to the  $\alpha$ -list
9:   for  $mm = 2; mm \leq k; mm++$  do
10:    for all  $\text{item } i$  in  $\alpha$ -list do
11:       $\langle \text{pat}, \text{mmposlist}, \text{offset} \rangle \leftarrow \alpha\text{-list}[i]$ ;
12:       $Z \leftarrow \text{LongestPrefix}(\text{tail}_{\text{offset}}^{\text{pat}}, SSA, G)$ ;
13:      for  $j = Z$  down to  $\text{mmposlist}[mm - 1]$  do
14:        for  $a = 0; a < 3; a++$  do
15:           $\text{newP} \leftarrow \text{pat}$ ;
16:           $\text{newP}[j] \leftarrow q_a$ ;
17:          add  $\langle \text{newP}, (\text{mmposlist}, j), \text{offset} \rangle$  to the
18:             $\beta$ -list
19:       $\alpha\text{-list} \leftarrow \beta\text{-list}$ ;
20:    Sort  $\alpha$ -list according to rightmost criteria;
21:    for all  $\text{item } i$  in  $\alpha$ -list do
22:       $\langle \text{pat}, \text{mmposlist}, \text{offset} \rangle \leftarrow \alpha\text{-list}[i]$ ;
23:       $R \leftarrow \text{SuffixRangeSearch}(SSA, \text{tail}_{\text{offset}}^{\text{pat}})$ ;  $\{R =$ 
24:         $\{k_1 \cdot D, k_2 \cdot D, \dots, k_\ell \cdot D\}\}$ 
25:      for  $j = 1; j \leq \ell; j++$  do
26:        if  $\text{head}_{\text{offset}}^{\text{pat}} = g_{k_j \cdot D - \text{offset}} \dots g_{k_j \cdot D - 2g_{k_j \cdot D - 1}}$ 
27:          then
28:             $P$  detected position  $k_j \cdot D - \text{offset}$  with  $k$ -
29:              mismatches;

```

III. IMPLEMENTATION

We have implemented the PSI-RA (a.k.a. Ψ -RA) read aligner based on the proposed method. The index of a reference genome in Ψ -RA has two parts as the sequence itself coded in 2-bit format, and the sparse suffix array of the sequence with a given sparsification factor. We use the Yuta Mori's implementation (sites.google.com/site/yuta256/sais) of the SAIS [20] algorithm for constructing the sparse suffix array. Initially we run SAIS on the 2-bit coded sequence that actually produces sparse suffix array with $D = 4$ as each byte is composed of 4 bases. According to the input D parameter, we extract the required SSA from the one with

$D = 4$. The input D parameter should be a multiple of 4 in the current implementation.

As an example, the size of the whole human genome, which has approximately three billion bases, is about $700MB$ in 2-bit format. If $D = 8$, SSA requires to store one eighth of the three billion positions, which is roughly $325K$ numbers. Since we store each number as a 32-bit integer, SSA for $D = 8$ needs approximately $1.5GB$ of memory. Thus, the total size of the index becomes $2.2GB$ including the original sequence.

The backbone of Ψ -RA is the binary search over the SSA. We use a trick to speed up search process on suffix array. We *partition* the suffix array according to initial $K = 8$ bases. That is, we store the range of each 8-gram on the SSA as a separate table that has $4^8 = 64K$ rows, where each row includes the starting and ending positions of the corresponding item on the SSA. When we search the pattern `ACGTTGCA` on the SSA as an example, we first fetch the range of the first eight bases `ACGTTGCA` from the table, and then run the ordinary binary search on that interval instead of the whole array. This trick decreases the time complexity $O(\log n)$ of binary search to $O(\log(n/4^K))$ along with the additional cost of storing the table of size $O(4^K)$.

Ψ -RA supports k -mismatch search for any k value, but having larger k values requires longer times as in other aligners. The difference is Ψ -RA returns the queried number of alignments sorted in right-most mismatch criteria, and guarantees to find them as oppose to some aligners sacrificing accuracy for speed. The implementation of the k -mismatch search is an optimized version of the algorithm depicted in Alg.2. Ψ -RA software can be downloaded from www.busillis.com/o_kulekci/PSIRA.zip for academic or non-commercial purposes.

IV. EXPERIMENTAL RESULTS

The reads used in all experiments are collected from SRR003078 experiment that is available from the sequence read archive at <http://www.ncbi.nlm.nih.gov/sra>. We mapped randomly selected $100K$ reads of various lengths from SRR003078 onto complete human genome GRCh37 that is available at the site of Genome Reference Consortium (<http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/index.shtml>).

We created the sparse suffix array based indexes of the reference genome with sparsification factors of 4, 8, 12, and 16 according to the proposed methodology. The index sizes of Ψ -RA(4), Ψ -RA(8), Ψ -RA(12), and Ψ -RA(16), where Ψ -RA(D) refers Ψ -RA with a sparsification factor of D , are $3.4GB$, $2.0GB$, $1.6GB$, and $1.3GB$ respectively. Note that these values all include the $700MB$ complete human genome in 2-bit format.

SSAs give us the opportunity to tune the size of the index according to available memory by defining the sparsification

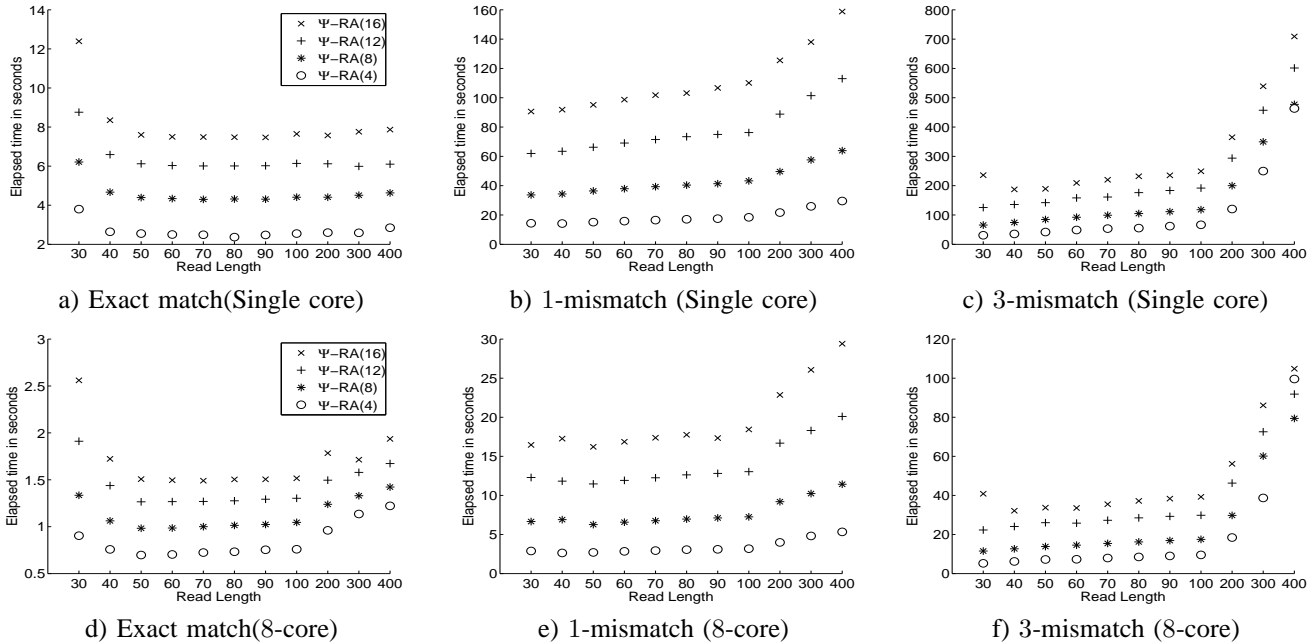


Figure 3. The effect of sparsification factor D on single thread versus eight-thread executions.

factor D . A larger D value results in a smaller index size, but the computation cost increases as we need to consider offsets from 0 to $D - 1$. Figure 3 exhibits this trade off. Same queries are executed both with single thread and with eight threads. It is observed that the gain in space is directly reflected to the computation time as expected. The increase in time can be compensated with the parallel execution of the software. Since today’s processors in general have multiple cores, we can work with smaller-sized indexes, and receive the same performance as if we are using a larger one. Sparse suffix arrays with larger sparsification factors catch the performance of the smaller ones by benefiting from multi-core architecture. For example on all cases, eight-core running of Ψ -RA(16) is faster than single processor running of Ψ -RA(4) index.

Short read aligners have different strategies and hence different parameters to perform matching. Although it is very difficult to make a really fair benchmark, we compare the speed of Ψ -RA against Bowtie, which is one of the fastest aligners, to give a clue on the performance. Figure 4 shows the results of the comparison. On exact matching and 3-mismatch Ψ -RA(4) represents a better performance, where Bowtie is faster on 1 and 2 mismatch cases. We should note that although Bowtie does not guarantee the results for the 2 and 3 mismatch cases, Ψ -RA finds all occurrences.

Two widely used techniques in genome indexing are Burrows-Wheeler transform or seed based hash tables. We compared Ψ -RA with *Bowtie* [5] and *SOAP2* [7], as the two successful representatives of BWT genre, and with *PerM* [6] and *mrsFast* [8], which are based on spaced

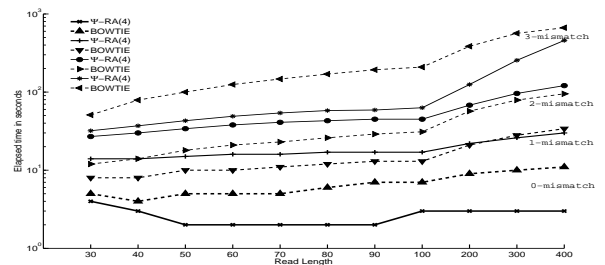


Figure 4. Performance comparison of Ψ -RA versus Bowtie on exact matching, 1-mismatch, 2-mismatch, and 3-mismatch alignments.

seeds and ordinary q -grams respectively. The exact matching performances of the tested aligners on mapping 100K reads against complete human genome is shown in Figure 5. The speed of the *mrsFast* was not competitive with the other aligners and hence it is not plotted. The index sizes of those aligners are 2.3GB (*Bowtie*), 6.1GB (*SOAP2*), 12.4GB (*PerM*), and 19.5GB (*mrsFast*).

V. CONCLUSION

Indexing DNA sequences require large memories. It is not possible to benefit much from the compressed indexes because of the random structure of the DNA sequences. As an alternative method of generating small size indexes of biological data, we focused on sparse suffix arrays, and developed an aligner named Ψ -RA based on this notion.

Ψ -RA is very fast on exact matching because of its cache

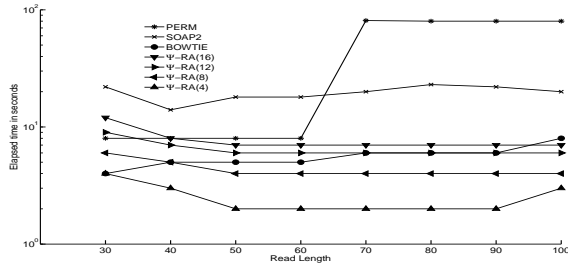


Figure 5. The comparison of some aligners on exact matching.

friendly structure. In addition to the speed caught in exact matching, we also integrated an elegant k -mismatch alignment capability by defining the rightmost mismatch criteria. The k -mismatch alignments of a queried read is reported according to the position of the mismatches, where being on the right is prioritized since the sequencing machines tend to generate erroneous bases especially towards the end of the reads.

The size of sparse suffix array changes with the used sparsification factor D , such that larger values result in smaller size indexes. On the other hand, large sparsification factors make the time complexity of the processing worse. We showed experimentally that by applying parallelism in contemporary processor architectures, small size indexes having large D catch the performance of the large size indexes that have smaller D values. Ψ -RA gives users the flexibility to tune the size of the index according to the available resources. The index will fit in the main memory with a cost of increased computation time. That increase can be avoided up to a level by benefiting from multi-core processors in practice.

REFERENCES

- [1] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nature Biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.
- [2] M. Metzker, "Sequencing technologies – the next generation," *Nature Reviews Genetics*, vol. 11, pp. 31–46, 2010.
- [3] P. Flicek and E. Birney, "Sense from sequence reads: methods for alignment and assembly," *Nature Methods*, vol. 6, no. 11, pp. S6–S12, October 2009.
- [4] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, pp. 315–327, 2010.
- [5] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, 2009.
- [6] Y. Chen, T. Souaiaia, and T. Chen, "Perm: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds," *Bioinformatics*, vol. 25, no. 19, pp. 2514–2521, 2009.
- [7] R. Li, Y. Li, K. Kristiansen, and J. Wang, "Soap: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, pp. 713–714, 2008.
- [8] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler, "Personalized copy number and segmental duplication maps using next-generation sequencing," *Nature Genetics*, vol. 41, no. 10, pp. 1061–1067, October 2009.
- [9] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, 2010.
- [10] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.
- [11] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, July 2009.
- [12] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000, pp. 390–398.
- [13] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1990, pp. 319–327.
- [14] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, pp. 378–407, 2005.
- [15] W.-K. Hon, R. Shah, and J. S. Vitter, "Compression, indexing, and retrieval for massive string data," in *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2010, pp. 260–274.
- [16] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, 2007.
- [17] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter, "Geometric burrows-wheeler transform: Linking range searching and text indexing," in *Proceedings of the Data Compression Conference (DCC)*, 2008, pp. 252–261.
- [18] Z. Khan, J. Bloom, L. Kruglyak, and M. Singh, "A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays," *Bioinformatics*, vol. 25, no. 13, July 2009.
- [19] E. Rivals, L. Salmela, P. Kiskinen, P. Kalsi, and J. Tarhio, "Mpscan: Fast localisation of multiple reads in genomes," in *Proceedings of the 9th Workshop on Algorithms in Bioinformatics (WABI)*, 2009, pp. 246–260.
- [20] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Proceedings of Data Compression Conference (DCC)*, 2009, pp. 193–202.