High Throughput Short Read Alignment via Bi-directional BWT

T.W. Lam*, Ruiqiang Li†, Alan Tam*, Simon Wong*, Edward Wu*, S.M. Yiu*

*Dept of Computer Science, University of Hong Kong, Hong Kong. {twlam,sltam,ckwong3,mkewu,smyiu}@cs.hku.hk
†Beijing Genomics Institute at Shenzhen, Shenzhen 518083, China. lirq@genomics.org.cn

***Abstract*—The advancement of sequencing technologies has made it feasible for researchers to consider many high-throughput biological applications. A core step of these applications is to align an enormous amount of short reads to a reference genome. For example, to resequence a human genome, billions of reads of 35 bp are produced in 1-2 weeks, putting a lot of pressure of faster software for alignment. Based on existing indexing and pattern matching technologies, several short read alignment software have been developed recently. Yet this is still strong need to further improve the speed. In this paper, we show a new indexing data structure called bi-directional BWT, which allows us to build the fastest software for aligning short reads. When compared with existing software (Bowtie is the best), our software is at least 3 times faster for finding unique best alignments, and 25 times faster for finding all possible alignments. We believe that bi-directional BWT is an interesting data structure on its own and could be applied to other pattern matching problems.**

Availability: http://www.bio5.cs.hku.hk:8080/P2BWT, where two human genomes are indexed for alignment.

***Index Terms*—short read alignment; BWT; indexing data structure**

I. Introduction

The advancement of sequencing technologies (e.g. Illumina Genome Analyzer, Roche 454, and Applied Biosystems SOLiD) has made it feasible to produce tens of millions of reads with length 35-50bp (short reads) in a single run. Generating reads for a mammalian genome can now be done in just 1-2 weeks at a very low cost. This provides an excellent platform for many exciting high-throughput biological applications, such as genome resequencing [1], [2], [3], [4], mapping DNA-protein interactions [5], whole-transcriptome sequencing [6], and whole-genome expression profiling [7].

A core and the first step of these applications is to align the short reads to a reference genome. As the reads are so short and modern sequencing technologies have very low error rate, existing applications usually require alignments allowing at most two mismatches. What is special about short read alignment is its enormous volume, often in the range of millions to billions. For example, to resequence a human genome with $15\times$ coverage, one might face 1.5 billions reads of length 35. A practical alignment software must be extremely fast.

Existing software: Prior to our work, there are four major tools for aligning short reads to a reference genome, namely Maq [8], SOAP [9], ZOOM [10], and Bowtie [11]. We have evaluated the tools using three sets of human short reads (obtained from NCBI), each containing one million reads of length 35bp. The reference sequence is the human genome. We consider alignments with up to two mismatches. Bowtie is the fastest; in our experiments (see Section V for details), Bowtie's average rate is about 200 seconds for one million reads if we only want to obtain the unique best alignment (see Section II for definition) while the average rate increases substantially to more than 8,000 seconds if we need to locate all occurrences of each read. For 1.5G reads, it would take a few days and more than 100 days for Bowtie to find the unique best hit and all occurrences, respectively. With the launch of 1000 human genome project, more and more such reads will be produced in a short period of time. A more efficient alignment tool is always desired even if we can use clusters to speed up the alignment in parallel.

Our contributions: In short read alignment applications, millions or billions of reads are aligned with the same reference sequence. Indexing the reference sequence is a common technique to speed up the alignment process. All the above existing tools are built on well-developed indexing and pattern matching technologies. SOAP is based on hashing, ZOOM on spaced seed, and Bowtie on BWT (Burrows-Wheeler transform) indexing. It has been known that in the context of DNA, BWT is the most space-and-time efficient indexing data structure for exact pattern matching [12], [13]. For example, it takes about 30 seconds to find all the exact occurrences of one million short reads in the human genome. However, BWT is not designed for approximate pattern matching, and there is no efficient way to use BWT directly for approximate pattern matching.

The original BWT allows searching of a pattern in one direction, namely, from right to left (*backward search*). However, for efficient approximate pattern searching, we need a data structure that can support searching in both directions (forward and backward search) and allow us to *switch from forward to backward search or vice versa in the course of aligning the pattern*. This would enable us to start matching a pattern from the middle, then extend the search in either directions and switch directions in the alignment process. A seemingly trivial solution is to keep two BWTs, one for the original sequence and the other for the reversal. Then forward search can be done via a backward search on the reversal of the pattern based on the latter BWT.

However, this solution cannot allow interleaving the backward and forward search. And it deals with two separate searching space, demanding a lot more memory (see Section III for details). We develop a new indexing data structure, called bi-directional BWT, to solve this problem. Based on bi-directional BWT, we develop a new short read alignment tool 2BWT, which is faster than all existing tools.

We evaluated 2BWT on aligning reads up to two mismatches using the same three data sets mentioned earlier. For one million reads, the average times for finding the unique best alignment and all possible alignments are 63 seconds and 245 seconds, respectively, which are about 3 times and 25 times faster than Bowtie.

We have extended 2BWT to handle insertions and deletions, as well as longer reads and more errors. Yet we have to admit that 2BWT is only a first attempt to exploit the power of the bi-directional BWT, which is an interesting data structure on its own and would be useful to other pattern matching problems. We also note that there is another tool, BWA [14], just published which is also based on the original BWT. According to [14], BWA is about 20% faster than Bowtie on experiments based on real data sets.

II. Problem Definitions and Indexing Basics

The short read alignment problem is defined as follows. Let $T[1..n]$ be a reference sequence, which is usually very long (e.g., the human genome is of length 3 billion). Given a large number (millions to billions) of short patterns (reads) P_i, we want to locate all substrings of T that is at most at a Hamming (or edit) distance of δ from each P_i. Each such substring is said to be an *occurrence* of P_i. We assume that characters in T and each P_i are chosen from the alphabet $\Sigma = \{A, C, G, T\}$, each pattern is of length in the range 35–100, and $\delta \leq 2$. Note that Hamming distance counts the number of mismatches (or modifications), while edit distance also allows insertion and deletion.

In general, there are four different types of output, required by different applications. (1) **Unique best hit:** For each read, we consider the occurrences with the smallest error. If the occurrence is unique (that is, exactly one), report this occurrence. Otherwise, report null. (2) **Arbitrary hit:** For each read, we just want to know whether it has an occurrence in T with distance at most δ. If so, just report any one of such occurrences. (3) **All valid hits:** This refers to the case of reporting all occurrences of each read. (4) **All best hits:** Consider any read P_i, define e to be the smallest possible distance between P_i and an occurrence of P_i in T. If $e \leq \delta$, report all occurrences who distance from the read is e.

Next, we give a brief review of two indexing data structures, namely, suffix array Burrows-Wheeler Transform (BWT) [15].

A. *Suffix array*

Given a text $T[1..n]$, we first define the suffix array of T, denoted $SA[1..n]$, as follows. $SA[i] = j$ if the suffix $T[j..n]$ is lexicographically the i-th smallest suffix among all suffixes of T (and we say that the *rank* of the suffix $T[j..n]$ is i). In other words, SA stores the starting positions of all suffixes of T in lexicographical order.

For any pattern P, suppose P appears in T. We define the *the SA range* of P with respect to T as $[s, e]$ such that s and e are respectively the rank of the lexicographically-smallest and largest suffix of T that contains P as a prefix. To find all occurrences of a pattern P in T, we can first compute the SA range of P (using $O(m \log n)$ time [16]), afterwards the occurrences of P can be retrieved from the suffix array directly one by one in constant time.

The suffix array of a text with n characters requires $n \log n$ bits of memory in addition to the text. For example, to store the suffix array of a human genome, it requires 12G memory. Instead of using the suffix array, we consider the compressed indexing data structure Burrows-Wheeler Transform (BWT).

B. *Burrows-Wheeler Transform (BWT)*

Given a text $T[1..n]$, the BWT data structure, denoted $BWT[1..n]$, is defined as $BWT[i] = T[j-1]$ where $j = SA[i]$ for $SA[i] \neq 1$, otherwise, set $BWT[i] = \$$, where \$ is a special character not in Σ and assumed to be lexicographically smaller than all other characters. That is, $BWT[i]$ stores the character immediately before the i-th smallest suffix. Note that BWT requires only the same amount of memory as for storing the text. Using BWT and some auxiliary functions, we can compute efficiently the SA range of a given pattern in a backward manner efficiently *backward search*. The idea is as follows.

For any character x, let $Count(x)$ be the total number of characters in the text T that are smaller than x. And let $Precede(i, c)$ be the number of character c that occurs in $BWT[1..i-1]$. With these two functions, given the SA range $[s, e]$ of a pattern P, computing the SA range $[s', e']$ for the pattern cP for any character c can be computed based on the following lemma [17], [13].

Lemma 1: Let P be a pattern and $[s, e]$ be its SA range. Let c be any character, the SA range of cP can be computed as $[Count(c)+Precede(s, c)+1, Count(c)+Precede(e+1, c)]$.

We can precompute and store up the values of $Count$ and $Precede$ using only $o(n)$ bits, while allowing constant time retrieval of any value. Precisely, for $Count$, we store the values using an array of $|\Sigma|$ entries with a total size of $|\Sigma| \log n$ bits. For $Precede$, we use a rank and select data structure [18], which requires only $o(n)$ bits. Given any $P[1..m]$, the SA range of $P[m..m]$ is simply $[Count(P[m]) + 1, Count(c)]$, where c is the character just lexicographically larger than $P[m]$. The lemma below summarizes the above discussion. Note that computing the SA range of a pattern using BWT is even faster than using an suffix array.

Lemma 2: Using BWT and the auxiliary functions, $Count$ and $Precede$, computing the SA range of any pattern P with length m can be done in $O(m)$ time.

To retrieve the positions of an SA range, we only store part of the suffix array, called *sampled suffix array*. Intuitively, we store one SA value for every α entries for some constant α. More precisely, we only store the $SA[i]$ value for $i = k\alpha$ for $0 \leq k \leq \lceil \frac{n}{\alpha} \rceil$. That is, we only store the $SA[i]$ value if the rank of the suffix $T[SA[i]..n]$ is a multiple of α. Retriving the value for $SA[i]$ where i is not a multiple of α can be done by searching repeatedly the BWT data structure itself [17]. For the memory requirement, using human genome as an example, the BWT array requires about 0.75G, the auxiliary data structure for $Count$ and $Precede$ requires less than 0.1G, while the sampled suffix array (for $\alpha = 4$) occupies 3G. The total memory consumption is about 4G only.

BWT has been tested empirically, it is indeed extremely efficient for exact pattern matching in DNA sequences[12]. Matching a pattern of a few hundred characters with the human genome takes a few microseconds only. However, BWT is not designed for approximate pattern matching. Existing BWT-based solutions (e.g. in Bowtie) for short read alignment mainly use a brute-force approach to handle the errors; the idea is simply to generate all possible erroneous patterns from a given read and perform exact match of all these patterns using BWT. Obviously, the efficiency of BWT deteriorates very rapidly even if a very small number of errors (say, 2 mismatches) has to be dealt with.

In the next section, we propose a new indexing data structure, called the bi-directional BWT, which can be exploited to perform approximate matching for small errors efficiently, while keeping the amount of memory reasonably small.

III. Bi-directional BWT

Recall that the original BWT can support the following operation efficiently (Lemma 2). To search for the occurrences of a pattern $P[1..m]$, we start from the last character $P[m]$, then search backward character by character. This is referred to as the *backward search*.

> Given a pattern P and its SA range, for any character c, return the SA range for cP.

Using only backward search, it is not trivial how to perform approximate matching efficiently. On the other hand, if the index can also support *forward search* (i.e., search the pattern from left to right), as well as switching from backward search to forward search, or vice versa, then one can start matching the middle part of a pattern first, and extend to either direction. As to be shown in Section IV, this would allow tremendous speed up of approximate matching with small errors.

At first glance, one might consider using two BWTs as a trivial approach to solve the problem. That is, one BWT built for the text T, and another BWT built for T^R (the reversal of T). Denote these BWTs as B and B', respectively. Given any pattern P, to perform forward search, we can perform backward search on P^R using B'. However, this solution has two problems. First, the SA range computed using B' is the SA range with respect to T^R, so to retrieve the occurrence positions, we also need to store the (sampled) suffix array with respect to T^R. This increases the memory consumption a lot. More importantly, it is not trivial to integrate the an SA range based on T and and SA range based on T^R, and to use the two BWTs to interleave forward and backward search.

Below we introduce the bi-directional BWT which can support the backward search, forward search, and switching of search direction. We store B and B', but we are able to maintain the SA ranges w.r.t. T even when a forward search is conducted, and represent all occurrences using the SA ranges based on T only. Thus, at the end we avoid dealing with SA ranges based on T^R and do not need the extra memory for storing the costly (sampled) suffix array for T^R. The memory requirement for B' is not much; for the case of human genome, this requires 0.75 GB. More specifically, our bi-directional BWT can support the following operations. Given a pattern P, we refer the SA range of P w.r.t. T and the SA range of P^R w.r.t. T^R as the SA range and SA' range of P, respectively.

> Given a pattern P, its SA range and SA' range, for any character c, return the SA range and SA' range of cP, as well as the SA range and SA' range of Pc.

Bi-directional BWT forward search: Given a pattern P, its SA range and SA' range, and any character c, we show how to compute the SA range and the SA' range of Pc. Computing the SA' range can be done by exploiting B' to conduct backward searching of $(Pc)^R$ w.r.t. T^R. The non-trivial issue is how to compute the SA range of Pc.

Let $[s, e]$ be the SA range of P. It is obvious that the SA range of Pc is a sub-range in $[s, e]$ as suffixes with prefix

Pc are also suffixes with prefix P. Let x be the number of suffixes in T that start with Pd where d is lexicographically smaller than c and y be the number of suffixes in T that have Pc as prefix. Then, the SA range of Pc can be computed as $[s+x, s+x+y-1]$. Also, for any pattern X, the SA range of X and the SA' range of X^R must have the same size. To compute x, we can compute, for each d smaller than c, the SA range of $(Pd)^R$ w.r.t. T^R using B' and Lemma 2. Similarly, y can be obtained by computing the SA range of $(Pc)^R$ w.r.t. T^R. Thus, we have the following lemma.

Lemma 3: Given the SA range and SA' range of a pattern P, we can compute, for any character c, the SA range and SA' range of Pc in $O(|\Sigma|)$ time.

Bi-directional BWT backward search: Similarly to the forward search, we have the following lemma.

Lemma 4: Given the SA range and SA' range of a pattern P, we can compute, for any character c, the SA range and SA' range of cP in $O(|\Sigma|)$ time.

Note that we keep track the SA range of the pattern no matter whether we align the pattern from left to right or from right to left. Thus, to retrieve the positions of the occurrences, we only need to store the sampled suffix array of T. In practice, performing a backward search using bi-directional BWT requires almost the same time as using the original BWT. For forward search, the time required using bi-directional BWT is only about 1.1 times that required by the backward search using the original BWT. The overhead is insignificant.

IV. The Alignment Algorithm

In this section, we describe how to make use of the bi-directional BWT to perform short read alignment up to two errors. Our algorithm is not heuristic-based, it covers all possible occurrences of the errors. The main idea is as follows. Since the number of errors is small, for each (approximate) occurrence of the pattern, there exists a sufficiently long substring (at least one third of the read for the case of two errors) that is exactly matched between the pattern and the occurrence. Thus, we always start the alignment by first matching with such a substring. Since this substring may be close to the left end, right end, or in the middle of the pattern, the original BWT data structure cannot support and take advantage this searching method. The following subsections give the details of the alignment algorithm based on the bi-directional BWT. Note that for exact matching of the pattern, we can simply apply backward search to locate the occurrences. In the following, we focus on the cases with errors. We show how to handle alignment that allows mismatches only and it can be extended to handle insert/delete.

1-mismatch alignment: Given a pattern $P[1..m]$, we want to find all substrings on the text which is equal to $P[1..i-1]eP[i+1..m]$ where $e \in \Sigma\backslash\{P[i]\}$ for $1 \le i \le m$. The mismatch e either appears in the first half or the second half of the pattern. We divide all occurrences into the following two cases and search for occurrences of each case one by one. Let $x = \lceil \frac{m}{2} \rceil$.

Case A. The mismatch occurs on the first x positions on the pattern (the first half). In this case, the second half of the pattern must occur exactly in the occurrences. So, we use backward search to obtain the SA range of $P[x+1..m]$. Then, we consider the mismatch occurs at the i-th position for $1 \le i \le x$ one by one. For $i = x, x-1, \ldots, 1$, we continue the backward search to obtain the SA range of $P[i+1..m]$. We compute the SA ranges of $eP[i+1..m]$ for all $e \in \Sigma\backslash\{P[i]\}$ in $O(|\Sigma|)$ time. With each SA range of $eP[i+1..m]$, we continue the backward search for the pattern $P[1..i-1]eP[i+1..m]$ in $O(i)$ time and report their SA ranges.

Case B. The mismatch occurs on the last $m-x$ positions (the second half). We first use the forward search of the bi-directional BWT to obtain the SA range and SA' range of $P[1..x]$. For each $i = x+1, \ldots, m$, we continue to obtain the SA range and SA' range of $P[1..i-1]$. Then, we compute the SA range and SA' range of $P[1..i-1]e$ for all $e \in \Sigma\backslash\{P[i]\}$. For each e, we continue the forward search to compute the SA range of $P[1..i-1]eP[i+1..m]$ in $O(i)$ time.

Intuitively, the main advantage of our approach is as follows. Consider all possible patterns with one mismatch with the given pattern. In most of these patterns, the search ends up with no result. We start by finding the SA range of a long (at least half of the read) substring of the pattern, the resulting SA range is usually small. With only a few search steps, the SA range quickly drops to zero if no occurrence of such 1-mismatch pattern appears in the text. Thus, the whole process can be sped up.

2-mismatch alignment: Given a pattern $P[1..m]$, we want to find all substrings on the text which is equal to $P[1..i-1]e_1P[i+1..j-1]e_2P[j+1..m]$ where $e_1 \in \Sigma\backslash\{P[i]\}$ and $e_2 \in \Sigma\backslash\{P[j]\}$ for $1 \le i < j \le m$. There are four cases depending where e_1, e_2 occur in the pattern. Basically, we divide the pattern into three parts (from left to right), each with length about one-third of the read. Case A: The mismatches occur in the first two parts. Case B: Both mismatches occur on the last part. Case C: The mismatches occur on the second and the third part respectively. Case D: The mismatches occur on the first and the last part respectively.

Let $s_1 = \lfloor \frac{m}{3} \rfloor$ and $s_2 = m - s_1$. We show how to find the SA range of the pattern for each case.

Case A. We first obtain the SA range of $P[s_2+1..m]$ using backward search. For each $j = s_2, s_2-1, \ldots, 2$, we assume that e_2 occurs at position j. We continue the backward search to obtain the SA range of $P[j+1..m]$, then we compute the SA range for $e_2P[j+1..m]$ for $e_2 \in \Sigma\backslash\{P[j]\}$. The mismatch e_1 occurs in $P[1..j-1]$, this sub-problem is in fact, a 1-mismatch alignment problem. Thus, we apply the procedure for finding 1-mismatch alignment for $P[1..j-1]$ to obtain the SA range of $P[1..i-1]e_1P[i+1..j-1]e_2P[j+1..m]$.

Case B. We first obtain the SA range and SA' range of $P[1..s_2]$ using forward search. For each $i = s_2+1,\ldots,m-1$, we assume that e_1 occurs at position i. We continue the forward search to obtain the SA range and the SA' range of $P[1..i-1]$. For each possible e_1, we obtain the SA range of $P[1..i-1]e_1$ and the SA' range. For each $j = i+1,\ldots,m$, we assume that e_2 occurs at position j. We continue the forward search to obtain the SA range of $P[1..i-1]e_1P[i+1..j-1]$ and the SA' range. Consider all possible cases of $e_2 \in \Sigma\backslash\{P[j]\}$ and continue the forward search, we can compute the SA range for $P[1..i-1]e_1P[i+1..j-1]e_2P[j+1..m]$.

Case C. We first obtain the SA range of $P[1..s_1]$ and the SA' range using forward search. For each $i = s_1+1,\ldots,s_2$, we assume that the mismatch e_1 occurs in position i. We continue the forward search to obtain the SA range of $P[1..i-1]e_1P[i+1..s2]$ and the SA' range for all $e_1 \in \Sigma\backslash\{P[i]\}$. The second mismatch should be on $P[s_2+1..m]$. Similarly, continue the forward search, we can compute the SA range of $P[1..i-1]e_1P[i+1..j-1]e_2P[j+1..m]$ for all possible e_2 with $j-1 \geq s_2$.

Case D. This case shows the full power of bi-directional BWT as we have to start our search in the middle of the pattern. We first obtain the the SA range of $P[s_1+1..s_2]$ and then the SA' range using forward search. For each $i = s_1,\ldots,1$, we apply backward search to compute the SA range of $P[1..i-1]e_1P[i..s_2]$. Then, for each $j = s_2+1,\ldots,m$ (e_2 occurs in one of the possible j-th positions), we apply forward search to compute the SA range of $P[1..i-1]e_1P[i..j-1]e_2P[j..m]$ for all possible e_2.

The above methodology can be generalized to handle three or more mismatches as well as insert/delete.

V. Experimental Results

We have implemented our short read alignment solution, called 2BWT, based on the bi-directional BWT. We compared the performance of 2BWT with SOAP (v1.11, released on July 2008), Bowtie (the memory intensive version v0.9.7.1, released on November 2008), Maq (v0.7.1, released on September 2008), and ZOOM (v.1.2.4, released on June 2008) using real short read data sets and the human reference genome obtained from NCBI. Note that in all our experiments, we align both strands of the genome.

Testing environment and data. All experiments are run on a single computer equipped with Intel(R) Xeon(R) CPU E5420 @ 2.50GHz / 6144 KB Cache / 24 GB RAM. The reference sequence is obtained from NCBI, Human genome [NCBI; Build 36.3]. We construct three short read datasets obtained from NCBI: SRR001113, SRR001114, and SRR001258.[1] For each source, we randomly select one million reads to form a testing dataset. These sources contain a small percentage of reads that are comprised entirely of the same nucleotide (in particular, "A"). We exclude them from the three datasets. Furthermore, we trim each read to 35bp so as to be compatible with the testings reported in the literature, which use length-35 reads (Bowtie [11]) or length-36 reads (ZOOM [10])).

Alignment speed: We compare the speed of Maq, SOAP, Bowtie, ZOOM and 2BWT according to four types of output: (1) unique best hit; (2) arbitrary hit; (3) all valid hits; and (4) all best hits. We focus on alignment with at most two mismatches. Table I shows the results on the four types of output. All the timing figures include both the computation and disk IO time. For aligning one million reads, 2BWT on average takes 59 seconds to 245 seconds, depending on the output required. In all cases, 2BWT is the fastest, followed by Bowtie. For unique best hit and arbitrary hit, 2BWT is about 3 times faster than Bowtie. For all valid hits and all best hits, 2BWT is about 25 times faster than Bowtie.

Memory consumption. Among all tools, ZOOM uses the least amount of memory (only 0.9G) while SOAP requires the largest amount of memory (14.3G). Bowtie and 2BWT use 11.2G and 13.2G memory respectively. Note that Bowtie has a lightweight version using less memory (about 2.7G) which is slower than the memory intensive version. 2BWT also has a lightweight version which uses 6.7G memory (nowadays the memory capacity of an ordinary personal computer is 8G) This lightweight version is about 10% to 20% slower than the memory intensive version.

We remark that SOAP, ZOOM, Bowtie, and 2BWT report the same set of occurrences up to a few differences on the boundary cases. On the other hand, Maq only guarantees at most two mismatches in the first 28bp of the read, thus reports more occurrences.

VI. Conclusions

In this paper we have shown how to extend the BWT index to support interleaving forward and backward search. The new data structure allows us to implement a simple and the fastest software for aligning short reads. Like other tools, in addition to mismatches, we also consider insert/delete and allow longer reads (e.g. 100bp) with three or more mismatches. Roughly speaking, the running time of 2BWT doubles when errors are modeled as insert/delete instead of mismatches. Also, the speed of 2BWT does not deteriorate at all when aligning

[1]Both SRR001113 and SRR001114 are from the project of 1000 Genome Whole Genome Shotgun Fragment, while SRR001258 is from the project 1000 Genomes Project Pilot 2.

		SRR001113		SRR001114		SRR001258		**Average**
Unique best hit	Parameters	Occurrences	Time(s)	Occurrences	Time(s)	Occurrences	Time(s)	**Time**(s)
SOAP	-s 12 -r 0	543,491	4,505	510,188	4,315	367,238	4,452	4,424
ZOOM	-mm 2 -mk 1	544,638	1,908	513,404	1,836	368,759	1,632	1,792
Bowtie	-t -f -v 2 -best -m 1	543,678	175	512,243	202	369,015	207	195
2BWT	-m 2 -h 0	563,551	53	531,796	61	381,561	74	63

Note: Maq does not support unique best hit.

Arbitrary hit	parameters							
Maq	-n 2 -C 1	756,954	7,607	744,316	6,894	605,091	6,023	6,842
SOAP	-s 12 -r 1	654,639	4,555	614,987	4,315	440,878	4,456	4,442
ZOOM	-mm 2 -mk 1	655,997	1,908	618,572	1,836	442,592	1,632	1,792
Bowtie	-t -f -v 2 -best	655,998	153	618,572	179	442,592	193	175
2BWT	-m 2 -h 1	655,999	49	618,574	55	442,592	71	59

All valid hits	parameters							
Maq	-n 2 -C 1	6,730,321,743	10,763	5,685,547,975	9,279	5,667,065,017	7,951	9,331
ZOOM	-mm 2 -mk 474784	1,229,392,005	10,094	923,181,419	9,061	606,920,304	6,842	8,666
Bowtie	-t -f -v 2 -a -nostrata	1,229,392,265	9,795	923,181,676	9,297	606,920,476	6,153	8,415
2BWT	-m 2 -h 2	1,229,392,200	284	923,181,598	247	606,920,401	204	245

Note: SOAP does not support all valid hits.

All best hits	parameters							
SOAP	-s 12 -r 2	58,440,851	4595	40,499,409	4388	30,696,832	4504	4496
Bowtie	-t -f -v 2 -a	64,513,685	1,159	44,516,608	920	33,248,152	697	925
2BWT	-m 2 -h 3	64,513,709	63	44,516,628	73	33,248,152	84	73

Note: Maq and ZOOM do not support all best hits.

TABLE I

THE TIME RQUIRED FOR FINDING (1) THE UNIQUE BEST HIT, (2) AN ARBITRARY HIT, (3) ALL VALID HITS, AND (4) ALL BEST HITS OF ONE MILLION READS (MEASURED IN SECONDS). THE EXPERIMENT IS BASED ON 3 DATASETS EACH CONTAINING ONE MILLION LENGTH-35 READS; THE REFERENCE SEQUENCE IS THE HUMAN GENOME; A HIT CONTAINS AT MOST TWO MISMATCHES.

longer reads with more errors. We have tested reads with 75 bp (both simulated data and real data NCBI SRR013647), the average time to find all valid hits with up to 3 mismatches is about 110 seconds for one million of reads. The time increases to 479 seconds when up to 4 mismatches are allowed.

In conclusion, 2BWT is the first alignment software that exploits the power of the bi-directional BWT index, We believe the latter would be useful to building more sophisticated short read alignment software, as well as other pattern matching tools.

ACKNOWLEDGEMENT

The project is partially supported by Hong Kong RGC Grant HKU7140/064.

REFERENCES

[1] D. Bentley, "Whole-genome re-sequencing," *Curr. Opin. Genet. Dev.*, vol. 16, pp. 545–552, 2006.

[2] L. Hillier, G. Marth, A. R. Quinlan, and D. D. et al., "Whole-genome sequencing and variant discovery in *C. elegans*," *Nature Methods*, vol. 5, pp. 183–188, 2008.

[3] D. R. Bentley, S. Balasubramanian, and H. P. S. et al., "Accurate whole human genome sequencing using reversible terminator chemistry," *Nature*, vol. 456, no. 7218, pp. 53–59, 2008.

[4] J. Wang, W. Wang, and R. L. et al., "The diploid genome sequence of an Asian individual," *Nature*, vol. 456, no. 7218, pp. 60–65, 2008.

[5] D. Johnson, A. Mortazavi, R. Myers, and B. Wold, "Genome-wide mapping of in vivo protein-DNA interactions," *Science*, vol. 316, no. 5830, pp. 1497–1502, 2007.

[6] T. Jarvie and T. Harkins, "Transcriptome sequencing with the Genome Sequencer FLX system," *Nature Methods*, vol. 5, 2008.

[7] G. Robertson, M. Hirst, M. Bainbridge, M. Bilenky, Y. Zhao, T. Zeng, G. Euskirchen, B. Bernier, R. Varhol, A. Delaney, N. Thiessen, O. Griffith, A. He, M. Marra, M. Snyder, and S. Jones, "Genome-wide profiles of STAT1 DNA association using chromatin immunoprecipitation and massively parallel sequencing," *Nature Methods*, vol. 4, pp. 651–657, 2007.

[8] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Research*, vol. 18, pp. 1851–1858, 2008.

[9] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, pp. 713–714, 2008.

[10] H. Lin, Z. Zhang, M. Zhang, B. Ma, and M. Li, "ZOOM! Zillions of oligos mapped," *Bioinformatics*, vol. 24, no. 21, pp. 2431–2437, 2008.

[11] 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. R25, 2009.

[12] R. A. Lippert, "Space-efficient whole genome comparisons with Burrows-Wheeler transforms," *Journal of Computational Biology*, vol. 12, no. 4, pp. 407–415, 2005.

[13] T. Lam, W. Sung, S. Tam, C. Wong, and S. Yiu, "Compressed indexing and local alignment of DNA," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, 2008.

[14] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, 2009.

[15] M. Burrow and D. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, California, Tech. Rep. 124, 1994.

[16] D. Gusfield, *Algorithms on strings, trees, and sequences*. Cambridge University Press, 1997.

[17] M.-Y. Kao, Ed., *Encyclopedia of Algorithms*. Springer, 2008.

[18] V. Mäkinen and G. Navarro, "Rank and select revisited and extended," *Theoretical Computer Science*, vol. 387, no. 3, pp. 332–347, 2007.