An Ultra-fast Approach to Align Longer Short Reads onto Human Genome

Arup Ghosh and Gi-Nam Wang
Unified Digital Manufacturing Lab
Department of Industrial Engineering, Ajou University
San 5, Woncheon-dong, Yeongtong-gu, Suwon 443-749, South Korea
E-mail: {arupghosh, gnwang}@ajou.ac.kr

Satchidananda Dehuri,
Department of Systems Engineering,
Ajou University, San 5, Woncheon-dong, Yeongtong-gu, Suwon 443-749, South Korea
E-mail: satchi@ajou.ac.kr

Keywords: DNA, sequence alignment, second-generation sequencing (SGS), substring matching, BWT

Received: April 13, 2013

With the advent of second-generation sequencing (SGS) technologies, deoxyribonucleic acid (DNA) sequencing machines have started to produce reads, named as "longer short reads", which are much longer than previous generation reads, the so called "short reads". Unfortunately, most of the existing read aligners do not scale well for those second-generation longer short reads. Moreover, many of the existing aligners are limited only to the short reads of previous generation. In this paper, we have proposed a new approach to solve this essential read alignment problem for current generation longer short reads. Our ultra-fast approach uses a hash-based indexing and searching scheme to find exact matching for second-generation longer short reads within reference genome. The experimental study shows that the proposed ultra-fast approach can accurately find matching of millions of reads against human genome within few seconds and it is an order of magnitude faster than Burrows-Wheeler Transform (BWT) based methods such as BowTie and Burrows-Wheeler Aligner (BWA) for a wide range of read length.

Povzetek: Metoda omogoča izredno pohitritev iskanja daljših vzorcev v človeškem genomu.

1 Introduction

The rapid advances in DNA sequencing technology have dramatically accelerated the biomedical and biotechnology research [2, 6, 28]. Thereby opportunities have been created for data mining researchers to analyze a gamut of data. With the advent of second-generation sequencing (SGS) technologies, there is an increasing pressing need of an approach that can align large collections of reads (possibly millions) onto the reference genome rapidly. The main motivation behind this read alignment is to discover commonalities and connections between newly sequenced molecules with respect to existing reference genomes [16].

Currently, DNA sequencing machines are capable of generating millions of reads in a single run when a DNA sample is given as an input [9, 16, 27]. The DNA sequencing machines take the DNA sample as input and break it into a number of short pieces, which then are again broken into equal-length fragments called reads [25]. The 'read alignment problem' is to find matching of those reads onto a reference genome. From the computer science point of view, a genome can be considered as a long string of characters/bases (human genome contains nearly 6 billion characters/bases), and reads can be regarded as a set of equal-length small strings of characters/bases. Now, read alignment task is to map those reads (small string of characters) onto genome (long string of characters). Simply, we can think of it as a common substring matching problem [25]. The main challenge of this read alignment problem is to efficiently build the reference genome index thus reads (usually millions) can be mapped rapidly. This read alignment task has many potential applications in biomedical and bioinformatics fields, for example: 'to detect genetic variations' [4, 21] which will indeed help to identify 'disease genome' [21], 'to map DNA-protein interactions' [18], 'to profile DNA methylation patterns' [11, 13], etc.

To deal with this read alignment problem, several read alignment tools or approaches have been proposed. However, they are primarily focused on previous generation short reads which are usually of 25-70 bases long [26, 27]. Unfortunately, with the advent of SGS technologies DNA sequencing machines have started to produce reads (named as longer short reads) which are much longer than the previous short reads. Read lengths have just increased to more than 100 bases within a few years [27]. This trend of increment in read length makes the existing aligners computationally infeasible. Hence, there is an increasing need of an approach that can handle this current generation reads efficiently and also can handle future generation more long reads (by observing the trend). Here the particular importance of the longer short read alignment problem can be realized. It is theoretically and also practically difficult to avoid the overhead of processing the increased read length. However, it is needed to bind the growth rate of the processing cost efficiently. Currently, most of the read aligners are unable to achieve this scalability which makes them limited to the short reads. To this end, this paper proposes an ultra-fast method for aligning longer short reads onto human genome by combining the best attributes of hash based indexing and searching. Our approach is not bounded to a particular range of reads and can scale well for more long reads.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Our proposed approach is described in Section 3. Experimental results are presented in Section 4. Section 5 contains our conclusive remarks of the work followed by a list of relevant and state-of-the-art references.

2 Related work

The approaches proposed so far by the several research groups for read alignment problem can be broadly classified into four categories.

1) Traditional sequence mapping tools, such as Basic Local Alignment Search Tool (BLAST) [1] and BLAST-Like Alignment Tool (BLAT) [19], are unable to cope efficiently with the massive amount of reads generated by the current generation DNA sequencing machines, which make it computationally infeasible for solving the current generation read alignment problem [9, 16, 24].

2) BWT [7] based approaches, such as BowTie [20] and BWA [22], create a BWT based index and use an iterative prefix matching technique to find an alignment. A BWT-based index takes small memory footprint for example, BowTie takes less than 2 GB [30] and BWA takes less than 6 GB [29] memory to work with complete human genome. BWT based approaches have another significant feature i.e., they can handle a wide range of read lengths. For example, BowTie can handle up to 1024 bases read length [30]. So, it can easily handle current generation reads and also able to handle future generation more long reads. However, its performance degrades rapidly as the read length increases [25].

3) Hash table based approaches have got more and more popularity nowadays. Some of them create hash table based index for reads e.g., Efficient Large-Scale Alignment of Nucleotide Databases (ELAND) [10], Mapping and Assembly with Quality (MAQ) [23], Short Read Mapping Package (SHRiMP) [26] etc. Other approaches use hash table for the reference genome indexing e.g., Wisconsin's High- throughput Alignment Method (WHAM) [25], Periodic Seed Mapping (PerM) [9], Short Oligonucleotide Alignment Program (SOAP) [24], etc. However, only Q-Pick [16] uses hash table for both read and reference genome indexing. Hash table based approaches are in general significantly faster. However, those hash table based approaches or their software implementation have some significant drawbacks. WHAM and Q-Pick create reference genome index for a specific length of the read, which cannot be used for the different length reads (means, if WHAM and Q-Pick create index to align X bases length reads then that index cannot be used for alignment of N bases length reads where $X \neq N$). This is a significant issue because we have to create index for each of the read length. This will cause a significant overhead with respect to the index building time and disk space consumption because, nowadays most of the genome sequence mining companies have large number of databases of varied read lengths. The most significant problem with the above approaches is that, they are primarily focused on short reads. Thus, these approaches or their software implementations are limited to a specific read length which does not cover the read length of the current generation (for example, currently Illumina can produce read length up to 250 base long [31]) and there is no straight forward way to extend it to handle current generation longer short reads (or future generation more long reads). For example, ELAND can handle up to 32 bases [24, 33], MAQ can handle up to 127 bases [20], Shrimp can handle up to 70 bases [26], WHAM can handle up to 128 bases [36], PerM can handle up to 64 bases [34], SOAP can handle up to 60 bases [35] which are significantly lower length than the current generation read length.

4) Sorted Index File based approach such as fetchGWI and tagger [17] index either the reference genome or the query set and perform an efficient mapping of those two set of sorted entries (one for reference genome and another for query set) to find matches. However, this approach is also limited to 30 bases read length [17].

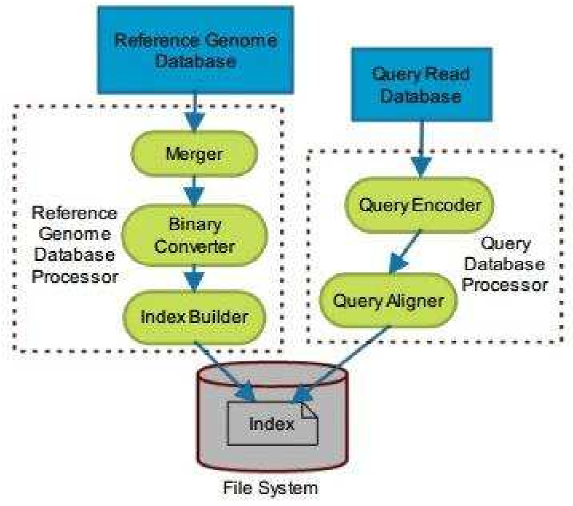

Figure 1: System Architecture.

Taking into account the limitations presented above, it can be summarized that, there is a significant lack of approaches or tools that can handle longer short reads efficiently. We propose to use BowTie and BWA approaches to solve this problem. From the trend of increment in read lengths, it does not seem that it will become infeasible in near future (because of its wide range of read length acceptance). So, we take BowTie and BWA as our base methods to experiment with longer short reads. As memory became cheap nowadays, we have no need to keep such unnecessary tight memory restriction in our approach as maintained by BowTie and BWA.

3 Proposed approach

This section is divided into two Subsections. In subsection 3.1, the statement of the problem is defined. The system architecture and working procedure of the proposed approach are described in subsection 3.2.

3.1 Problem Statement

A complete genome sequence is a set of all its chromosomal sequences. A chromosomal sequence is a series of characters. Each character (nucleic acid) is represented by the symbols A, G, C, or T (stands for adenine, guanine, cytosine and thymine respectively) or an unknown/ambiguous character, named N. The unknown character, N, represents that there is an uncertainty about the nucleotide in that position or there is a repetitive junk region in the genome and thus, all nucleotides in that region are converted into N's [25]. In the genome sequencing task, it has no biological sense to match reads onto those repetitive junk regions [25]. For simplicity, we can think N indicates error while matching [30].

The read alignment task is to efficiently build an index of the reference genome thus a fast and exhaustive mapping of a large collection of equal length query reads is possible while maintaining the accuracy in alignment. Query read database usually contains millions of reads and while mapping, read aligner has to report the matching position/s in the chromosomal sequence (if any matching occurs).

3.2 System Architecture of Proposed Approach

System architecture of our proposed approach is given in Figure 1. It has two main components: i) reference genome database processor and ii) query database processor. We will discuss them separately to present our approach in greater detail.

3.2.1 Reference Genome Database Processor

Reference genome database processor takes the complete genome sequence database as an input and creates an index for that genome into the file system (Figure 1). Complete genome sequence contains full set of chromosomal sequences. Note that though we are interested in mapping the query reads on both the forward and reverse strands of each chromosome, we will build index only for forward strand of each chromosome. We will compensate this while processing the query database (detail in subsection 3.2.2). We have selected this technique to reduce the index size because with this technique, we have to process only the half of the original genome sequence which will indeed provide us with speed gain while query read is searching.

Main idea behind our approach is to store in index all possible substrings of length L of every chromosomal sequence (only forward strand) with its position information. We set the length L value to 32. Note that as we are going to index each possible substring in a hash

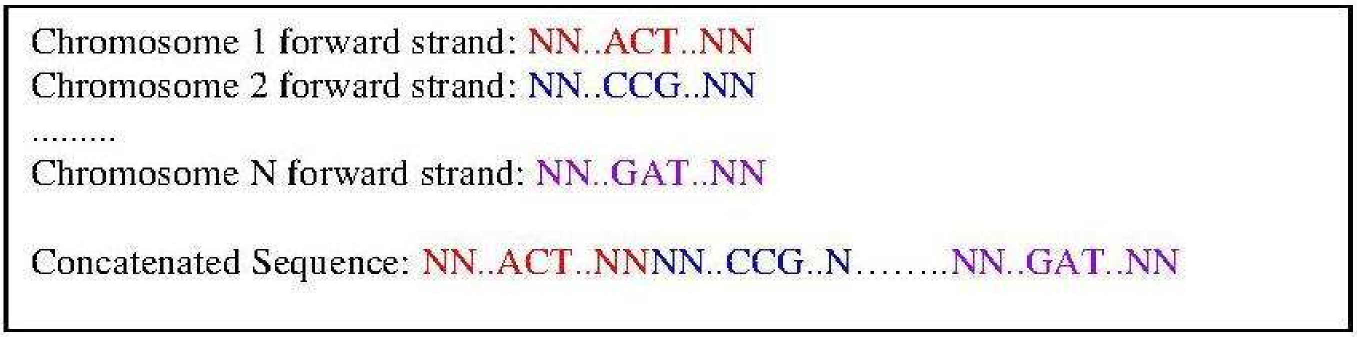
Chromosome 1 forward strand: NN..ACT..NN
Chromosome 2 forward strand: NN..CCG..NN
....
Chromosome N forward strand: NN..GAT..NN
Concatenated Sequence: NN..ACT..NNNN..CCG..N........NN..GAT..NN

Figure 2: Concatenated Chromosomal Sequence.

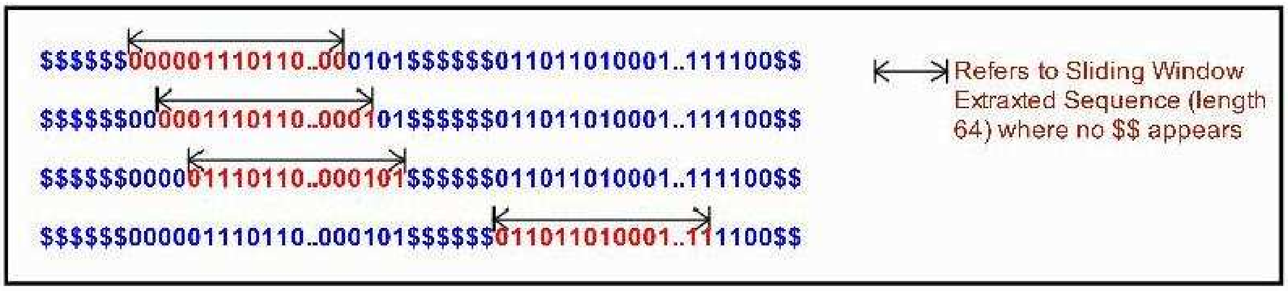

Figure 3: Sliding Window Extraction Protocol.

table, there are 4^{32} possible values (recall that, sequence can contain four bases i.e. A, C, G, or T), which will dramatically mitigate the possibility of hash collision.

Reference genome database processor starts working by concatenating (or merging) the forward strands of each chromosome (Figure 2). This is done by merger part of reference genome database processor (Figure 1). Note that with this concatenation we will lose information about "in which chromosome (or in which position of that chromosome) the subsequence of the concatenated sequence originally belongs to?" This information will be required when we find a match on a specific position on the concatenated sequence. Note that chromosome number (or position value in that chromosome) can be easily calculated by noting down the length of each chromosome. Motivation behind concatenation is to reduce the index space because with this technique, we have no need to store the chromosome number as we are going to calculate it during query read processing phase (detail in subsection 3.2.2).

Binary converter takes the concatenated sequence from merger (Figure 1) and converts each A/C/G/T character into two bit binary representation. A, C, G, T will be binary represented by 00, 01, 10, 11 respectively. Note that, as we are going to index each possible substring of L = 32, this representation will allow us to pack each of them into one computer word in the 64 bit computer architecture. Actually, we have set the L value to 32 thus our method can take advantage of current day's 64 bit computer architecture. Also note that, we are not going to index the subsequences in which N occurs (because N indicates 'error in matching'). So, if N occurs in the concatenated sequence, we will simply replace it by any two special characters (say with '$$'). By doing so, we can identify if N has occurred in the sequence. For example, if the concatenated sequence is 'NGACTN', Binary Converter will encode it as '$$10000111$$'.

Index builder takes the binary converter outputted sequence and creates an index which can be used by query database processor (Figure 1). Index builder moves a sliding window of length 64 over the input sequence and extracts the subsequence within it, and then moves two positions (Figure 3). Recall that, by doing so, it is originally extracting all possible subsequences of length L = 32 from the concatenated chromosomal sequence. Sliding window will extract the subsequences only if no $$ ($$ refers to N which means error in matching) appears in that window (Figure 3). Here, we have to keep in mind that sliding window should not extract any subsequences which do not belong to the original chromosomal sequences. This can happen while extracting subsequences from the position of concatenation of chromosomal sequence n and chromosomal sequence (n + 1) [here, n = 1, 2… up to (maximum chromosome number – 1)] (depicted in Figure 4). This can be easily avoided by keeping in mind the length of chromosomal sequences.

All extracted subsequences, which are basically 64 bit integer numbers, are hashed and their hash value provides the hash table bucket number. We have used Thomas Wang's hash function [14] to uniformly distribute values over the hash table. Thomas Wang's hash function has been widely used by many approaches for various purposes [8, 12, 15]. This is well suited for our purpose because it is fast to compute and has very high avalanche effect [3, 14]. Hash table values are the position values of the corresponding subsequences (represented by 32 bit integer numbers) in the concatenated chromosomal sequence. All those key-value pairs are inserted into the hash table, whose structures are depicted in Figure 5. Our hash table

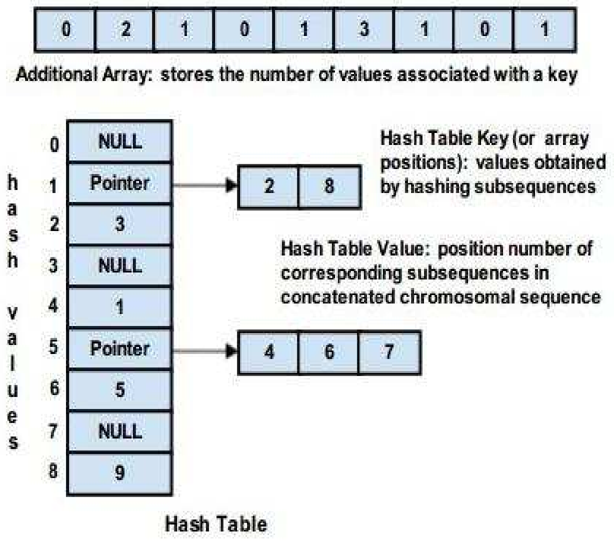
0 2 1 0 1 3 1 0 1
Additional Array: stores the number of values associated with a key
hash values
0 NULL
1 Pointer
2 3
3 NULL
4 1
5 Pointer
6 5
7 NULL
8 9
2 8
4 6 7
Hash Table Key (or array positions): values obtained by hashing subsequences
Hash Table Value: position number of corresponding subsequences in concatenated chromosomal sequence
Hash Table

Figure 5: Hash Table Structure.

structure is basically a long array, initially filled with NULL values and when we have to insert a key-value pair, we just insert that value in the corresponding array position (array position is found by hashing the key). Note that if corresponding array position is filled, then it will be replaced by a pointer to an array and the old value (or values) and the new value will be inserted into that

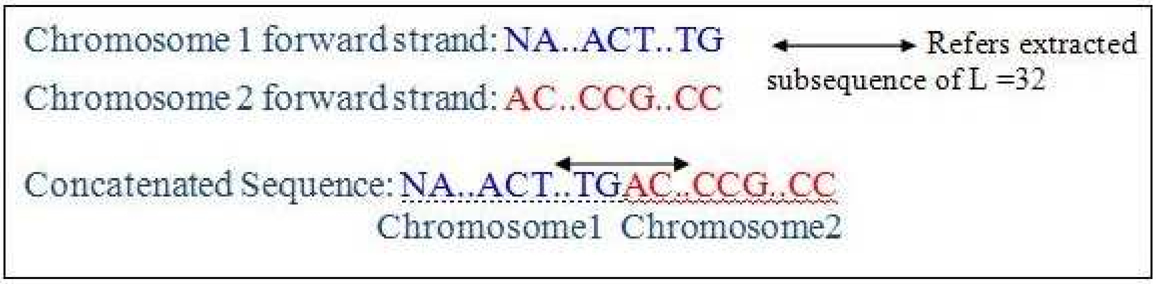
Chromosome 1 forward strand: NA..ACT..TG
Chromosome 2 forward strand: AC..CCG..CC
Refers extracted subsequence of L =32
Concatenated Sequence: NA..ACT..TGAC..CCG..CC
Chromosome1 Chromosome2

Figure 4: Error in Subsequence Extraction.

array. We have used this type of hash table structure in place of traditional hash table structure (which is usually two dimensional linked lists or an array of linked lists) to reduce the hash table space requirement. This kind of hash table structure efficiently reduces the requirement of pointers with the cost of an additional array which stores the number of values associated with the corresponding key (easily represented by 8 bit integer number - recall that we have used subsequence of length L = 32 to mitigate the possibility of collision). By doing this, we can dramatically reduce the hash table space requirement (realized through experiments also) because the size of a pointer in current day's system architecture is much longer than the 8 bit integer and many subsequences may appear only one time in the concatenated chromosomal sequence.

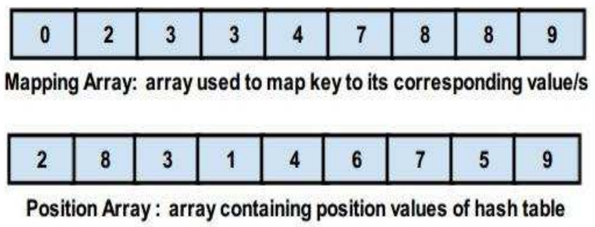
0 2 3 3 4 7 8 8 9
Mapping Array: array used to map key to its corresponding value/s
2 8 3 1 4 6 7 5 9
Position Array : array containing position values of hash table

Figure 6: Converted Sequential Structure Hash Table.

However, though we have reduced the space requirement with that hash table structure (as presented in Figure 5), there are some issues with that data structure also. In such data structure, index loading will be quite time consuming as we are not able to bulk load that data structure after saving it into the file system. For a significant number of hash table keys (as many subsequences can appear more than once), every time we have to access the values (a key can have many corresponding values) associated with it through a pointer access. This will make it relatively slow in comparison with the data structure where we can access the values directly. Thus, we convert our hash table structure (Figure 5) into two sequential structures shown in Figure 6. This will make bulk loading possible and with that data structure we can directly access the value/s through its corresponding key. In Figure 6, we have depicted the conversion of the hash table structure shown in Figure 5 to the sequential structures. The conversion algorithm is simple. The position array (as in Figure 6) contains all the hash table values (or position numbers - as in Figure 5) inserted into it, one by one according to hash bucket number, started from n = 0 to (number of hash bucket -1). To calculate the mapping array value at position i, we have to just add Additional Array value of position i and mapping array value at position (i -1) (see Figure 5 and 6). Please note that, for i = 0, this is not true (as array position can't be negative). So, we have to check every time whether i = 0 or not. We just remove this checking requirement by making hash function to produce hash values greater than 0 and setting 0th index of mapping array to 0 (see Figure 5 and 6). This will help us to avoid checking (thus speed gain) while performing query read mapping. Now, from mapping array, we can easily map keys to its corresponding value/s. For example, suppose the hash value of a key is i where i can be any value ranging from 1 to (number of hash bucket - 1). Now, from Figure 6, we can easily find that, (mapping array [i] - mapping array [i-1]) provides the number of value/s associated with that key (for example, if i = 1, then the key has two values associated with it, also see Figure 5). To find that value/s, we have to just run a loop, collecting value/s from position array starting from position number mapping array [i-1] (for example, if i = 1, we have to collect two values from position array starting from position number 0). The mapping process is presented in Figures 5 and 6.

3.2.2 Query Database Processor

Query database processor takes a query read database (possibly contains millions of equal length reads) and the saved index (index saved into file system by reference genome database processor (Figure 1)) as inputs and outputs 'query read matching information' into file system for each such matched reads. The 'query read matching information' contains information about query read alignment region (at what position in which chromosome the matching occurs), number of other alignments etc.

Our query database processor starts working by bulk loading the index (index refers to mapping array and position array as in Figure 6). This bulk loading (which will save significant amount of time) is possible only because we have converted our index into two sequential structures. After loading the index into memory, our query database processor takes each query read from the query database and searches into the index for matching in the following manner.

Query database processor will process each query read following the same procedure as done in section 3.2.1, except, it will not process the query reads in which N (or error in matching) occurs. Query read encoding (dividing the read into subsequence of L = 32 and converting them into binary) is performed by the Query Encoder and Query Aligner is responsible for matching task (Figure 1). Please remember that we have indexed all possible subsequence of length L = 32 of the genomic sequence. Hence, Query Encoder will first divide each query read into the subsequence of length L = 32. For example, if the query read is of 100 bases, Query Encoder will divide it into four subsequence of length L = 32. The first subsequence will be from base 1 to 32, the second subsequence will be from base 33 to 64, the third subsequence will be from base 65 to 96, and the fourth subsequence will be from base 69 to 100. Note that, the last subsequence will be taken from the end of the read and overlapping in subsequence may happen. Query Aligner searches the index for each such subsequence (after binary converted by Query Encoder) of the query read by hashing and mapping them into the hash table (following the same procedure as stated in section 3.2.1). Returned matching position/s is stored into arrays. If any of the subsequence of that read is failed to align, then we can easily conclude that the read is failed to align. The worst case time complexity to find it is $O(2n)$ where n =

number of subsequence of that read. However, reverse is not true because, if searching of all query read segments is successful, it only means that all the query read segments appear in the concatenated chromosomal sequence and not necessarily mean that the whole read appeared in the concatenated chromosomal sequence.

To check if the read is aligned or not, Query Aligner has to perform some additional task i.e., it has to check whether the returned positions are the consecutive segment positions or not. Take example of 100 bases read length. Suppose, all the four subsequence are able to align and returned position value/s are stored in the arrays named A1 = {200, 415}, A2 = {232, 327, 1215}, A3 = {264, 416, 917, 971} and A4 = {268} consecutively (values inside the curly brackets are the returned position value/s). Now, to check whether the read is aligned or not (or in which position/s), Query Aligner has to search for (A1[i] + 32) in A[2], (A1[i] + 64) in A[3], (A1[i] + 68) in A[4] means for every value of i i.e., from 0 to (size of A1 – 1). By doing so, we are only checking whether the segments are consecutive segments in the concatenated chromosomal sequence or not. If searching in all the arrays is successful, then only we can conclude that the read is aligned at position A1[i] in the concatenated chromosomal sequence (for above example, the query read matches only in position 200). Here, we should mention that all arrays that store the returned matching positions are the sorted array (easy to see). Thus, Query Encoder will perform an efficient linear search in the sorted array to find a match, instead of other searching procedures (for example binary searching). This will help us to gain speed over other searching procedures because the length of the array is typically very small due to very high indexed substring length i.e., 32 [5, 32]. With this linear search we can find all the matches by only one pass through the array (means with worst case time complexity $O(n)$ where n = very small array length). Another point to note that, with the above procedure we can only identify in which position of the concatenated chromosomal sequence the match occurs. Now, Query Aligner finds the original matching position (means chromosome number and the position value in that chromosome) by using the following procedure. First it finds the previous chromosome ending position in the concatenated chromosomal sequence (so, chromosome number is found) and then deducts that position value from the matched position value (except that matched position is not within the first chromosome ending position) to find real position in that chromosome. The previous chromosome ending position is found by performing an efficient binary search on a sorted array which contains the ending position of each chromosome in concatenated chromosomal sequence (reported by Reference Genome Database Processor).

As mentioned above, only the forward strand of each chromosome is processed by the Reference Genome Database Processor (subsection 3.2.1). This will be addressed in details in this section. Two strands of chromosome are of complementary nature i.e., A always pairs with T, and C always pairs with G (vice versa). So, for each query read in the query read database, Query Database Processor will not only search for that query read but also search for the reverse complement of it. For example, suppose, a query read is 'ACCTGGA'. Query Database Processor will first reverse it i.e., 'AGGTCCA' and then will take complement of it i.e., 'TCCAGGT' and then search into the index for matching following the same procedure as stated above.

From the above, it is easy to see that our approach has no upper limit restriction on the read length like many other approaches. In the next section, we will provide empirical evaluation of our approach for a wide length of reads.

4 Experimental study, results, and discussion

We ran our experiments on a desktop computer with 3.70 GHz Intel Xeon dual-cores CPU and 32 GB of DDR3 main memory, running 64 bit Ubuntu (kernel 3.5.0) as operating system. All our algorithms were implemented in C++, and compiled using g++ 4.7.2. We had followed the similar comparison strategy as performed in [25]. We have taken repeat-masked NCBI build 36 human genome as our reference genome and all the approaches obliged to report all the valid matches (as done in [25]). Our

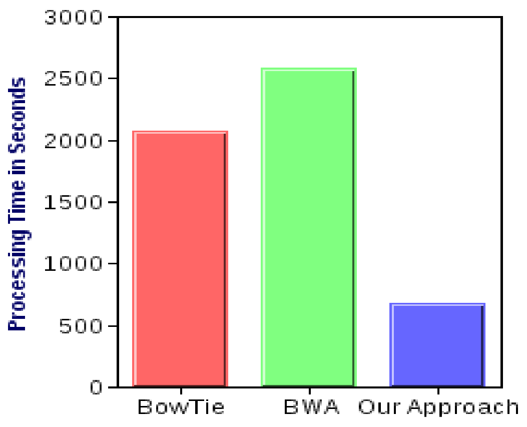

Figure 7: Comparison of Index Building Time.

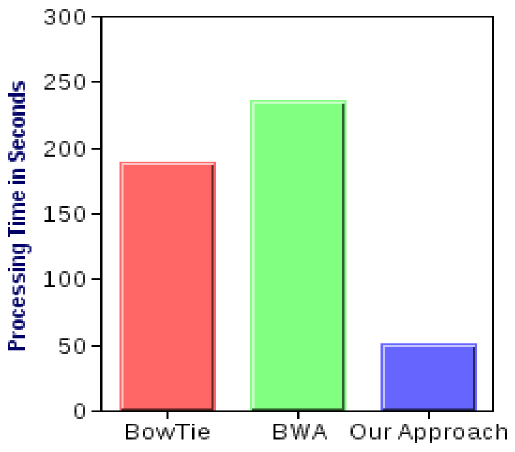

Figure 8: Comparison of Query Read Database Aligning Time for Read Length of 100 bases.

approach followed the same default output format of BowTie and all the approaches ran on single thread.

We have performed experiments with various length reads i.e. with 100, 150, 200, 250 bases read length (note that 250 bases read length is the currently maximum read length which Illumina can produce [31]). We selected four sets of query read database for experimenting with each of the read lengths. Our experimental results show comparison of our approach with BowTie [20, 30] and BWA [22, 29] by averaging results over those four sets of query read database. Each of our read databases contains 10 million of query reads. We have set the hash table bucket number to 1.5 billion throughout our experiments. We have also created four synthetic query read databases, one for each 100, 150, 200, 250 bases read lengths (each contains 10 million query reads with randomly inserted errors) to measure accuracy of our approach.

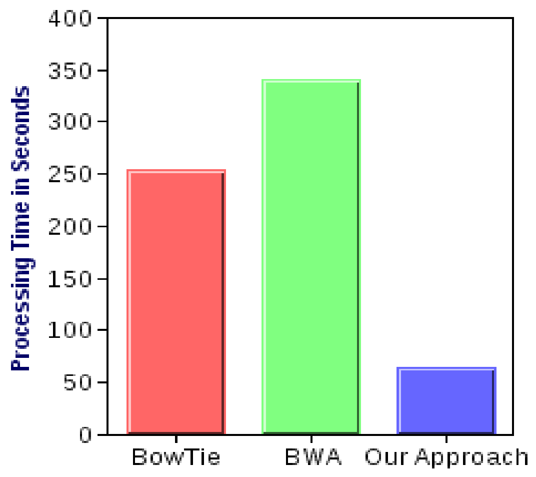

Figure 9: Comparison of Query Read Database Aligning Time for Read Length of 150 bases

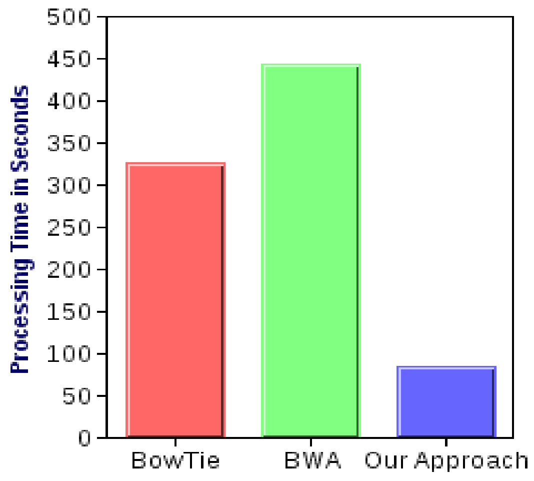

Figure 10: Comparison of Query Read Database Aligning Time for Read Length of 200 bases

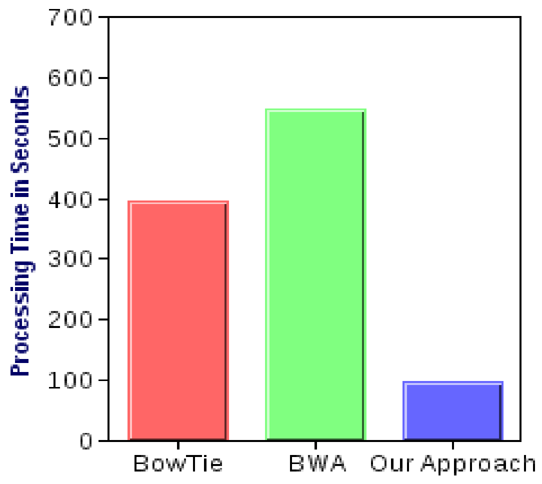

Figure 11: Comparison of Query Read Database Aligning Time for Read Length of 250 bases

Comparison of index building time of our approach with BowTie and BWA is given in Figure 7. Our experimental result show that our approach is significantly faster than BowTie (3X faster) and BWA (3.8X faster) as presented in Figure 7. Please note we have to build our index just only one time for a genome and we can use it repeatedly for searching various length reads (easy to see from section 3) unlike many other approaches. Experimental results for comparison of our approach with BowTie and BWA for various length reads i.e. of 100, 150, 200, 250 bases read length are given in Figures 8-11 respectively. From those experimental results it can be easily seen that, our approach is significantly faster for query read alignment than BowTie (3.7X, 3.9X, 3.7X, 4X faster for 100, 150, 200, 250 bases read length respectively) and BWA (4.6X, 5.2X, 5.1X, 5.6X faster for 100, 150, 200, 250 bases read length respectively). By significantly reducing the index building and query read searching time over BowTie and BWA, our approach is able to fulfill its primary motivation. To measure how much accurate our approach is, we ran it on four synthetic databases one for each 100, 150, 200, 250 bases read length, where it was previously known a number of query reads that provide an alignment. Our approach is able to align exactly the same number of query reads within these databases. In addition, during the previous experiments with BowTie for various length read databases (i.e. four sets of databases for each of 100, 150, 200, 250 bases read length, as stated early of this section), we have found that for all the databases of all the read length, our approach is able to align exactly the same number of query reads as aligned by BowTie (which is one of the most accurate read aligner as can be found from the experimental results of [16]). Actually, the accuracy of our approach can be outlined as follows:

- We have indexed subsequence of length L = 32 and used Thomas Wang's hash function (which uniformly distributes the key values) to mitigate the possibility of collision.
- We have used large number of hash table buckets i.e. 1.5 billion during our experiments which will also dramatically mitigate the possibility of collision. During index building time, we have found that our approach has extracted around 1.25 billion subsequences of

length L = 32 from NCBI human genome (build 36) which is quite lower value than 1.5 billion.

- Our approach is primarily targeted for current generation reads (or future generation more long reads) which is > 100 bases. So, query reads will be divided into >= 4 fragments and our approach will provide false positive match only if all the fragments gives collision (easy to see) which is quite unlikely to occur.

From the above discussion, we can conclude that, our approach is significantly faster than other methods presented and discussed above for comparison in all their aspects without compromising the accuracy. Moreover, from Figures 8 and 11, we can see that our approach becomes 1.91X slower (for BowTie, it is 2.09X and for BWA, it is 2.32X) by increasing the read length from 100 bases to 250 bases (note that the read length is increased 2.5X). This performance degradation rate is not completely accurate because of the difference in query reads in the databases (thus will give different processing execution). However, we can use this to get a rough idea about the growth rate of the performance degradation (as the database contains same number of reads and have to perform same kind of task). As we have argued previously, it is practically impossible to avoid the processing cost of the increased read length. However, we can summarise that our approach is able to bind it efficiently. By observing this bounded growth rate of performance degradation over BowTie and BWA, we can draw a conclusion that our approach will scale well for more long reads of future generation as well.

5 Conclusion

With the advent of second-generation sequencing technology, there is an increasing need of a fast and accurate read alignment method that can deal with longer short reads. In this paper, we address that need. Our experimental section shows that, for the longer short read of the current generation, our approach is an order of magnitude faster than BowTie and BWA in all aspects and this is done by keeping the accuracy intact. It can also be seen from the results that our approach can handle current generation's longer short read efficiently and also scale well for future generation's more longer short reads (by observing the bounded growth rate of performance degradation) and hence, will not become infeasible in near future (by observing the trend of increment in read length). Moreover, our approach has no upper bound in the read length like many other approaches.

References

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J. (1990) "Basic local alignment search tool", *Journal of Molecular Biology*, Vol. 215, No. 3, pp. 403-410.

[2] Ansorge, W.J. (2009) "Next-generation DNA sequencing techniques", *New Biotechnology*, Vol. 25, No. 4, pp. 195–203.

[3] Aydin, F. and Dogan, G. (2013) "Development of a new integer hash function with variable length using prime number set", *Balkan Journal of Electrical & Computer Engineering*, Vol. 1, No. 1, pp. 10-14.

[4] Bentley, D.R., Balasubramanian, S., et al. (2008) "Accurate whole human genome sequencing using reversible terminator chemistry", *Nature*, Vol. 456, No. 7218, pp 53-59.

[5] Bentley, J.L. and McGeoch, C.C. (1985) "Amortized analyses of self-organizing sequential search heuristics", *Communications of the ACM*, Vol. 28, pp 404-411.

[6] Berglund, E.C., Kiialainen, A., and Syvanen, A.C. (2011) "Next-generation sequencing technologies and applications for human genetic history and forensics", *Investigative Genetics*, Vol. 2, No. 1, pp. 23.

[7] Burrows, M. and Wheeler, D. (1994) *A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation.

[8] Chavarria-Miranda, D., Márquez, A., Nieplocha, J., Maschhoff, K., and Scherrer, C. (2008) "Early experience with out-of-core applications on the cray XMT", *IEEE International Symposium on parallel and Distributed Processing (IPDPS 2008)*, pp. 1-8.

[9] Chen, Y., Souaiaia, T., and Chen, T. (2009) "PerM: Efficient mapping of short sequencing reads with periodic full sensitive spaced seeds", *Bioinformatics*, Vol. 25, No. 19, pp. 2514-2521.

[10] Cox, A. J. (2007) *ELAND: efficient large-scale alignment of nucleotide databases*, Illumina, San Diego, USA.

[11] Deng, J., Shoemaker, R., et al. (2009) "Targeted bisulfite sequencing reveals changes in DNA methylation associated with nuclear reprogramming", *Nature Biotechnology*, Vol. 27, No. 4, pp 353–360.

[12] Devarakonda, K., Ziavras, S.G., and Rojas-Cessa, R. (2007) "Measuring Network Parameters with Hardware Support", *Third International Conference on Networking and Services (ICNS'07)*, pp. 2-2.

[13] Down, T.A., Rakyan, V.K. et al. (2008) "A Bayesian deconvolution strategy for immunoprecipitation-based DNA methylome analysis", *Nature Biotechnology*, Vol. 26, No. 7, pp 779-785.

[14] Golubitsky, O. and Maslov, D. (2012) "A study of optimal 4-bit reversible Toffoli circuits and their synthesis", *IEEE Transactions on Computers*, Vol. 61, No. 9, pp. 1341–1353.

[15] Greuter, S., Parker, J., Stewart, N. and Leach, G. (2003) "Real-time procedural generation of 'pseudo infinite' cities", *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '03)*, pp. 87–94.

[16] Huynh, T., Vlachos, M. and Rigoutsos, I. (2010) "Anchoring millions of distinct reads on the human genome within seconds", *Proceedings of the 13th*

International Conference on Extending Database Technology, pp. 252-262.

[17] Iseli, C., Ambrosini, G., Bucher, P. and Jongeneel, C. (2007) "Indexing Strategies for Rapid Searches of Short Words in Genome Sequences", *PLoS ONE*, Vol. 2, No. 6, Article e579.

[18] Johnson, D.S., Mortazavi, A., Myers, R.M. and Wold, B. (2007) "Genome-wide mapping of in vivo protein-DNA interactions", *Science*, Vol. 316, No. 5830, pp. 1497-1502.

[19] Kent, W. J. (2002) "BLAT–the BLAST-like alignment tool", *Genome Research*, Vol. 12, No. 4, pp. 656–664.

[20] Langmead, B., Trapnell, C., Pop, M. and Salzberg, S.L. (2009), "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome", *Genome Biology*, Vol. 10, No. 3, Article R25.

[21] Ley, T.J., Mardis, E.R. et al. (2008) "DNA sequencing of a cytogenetically normal acute myeloid leukaemia genome", *Nature*, Vol. 456, No. 7218, pp. 66–72.

[22] Li, H. and Durbin, R. (2009) "Fast and accurate short read alignment with Burrows-Wheeler transform", *Bioinformatics*, Vol. 25, No. 14, pp. 1754–1760.

[23] Li, H., Ruan, J. and Durbin, R. (2008) "Mapping short DNA sequencing reads and calling variants using mapping quality scores", *Genome Research*, Vol. 18, No. 11, pp. 1851–1858.

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

[25] Li, Y., Terrell, A. and Patel, J.M. (2011) "WHAM: A High-throughput Sequence Alignment Method", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 445–456.

[26] Rumble, S.M., Lacroute, P., Dalca, A.V., Fiume, M., Sidow, A. and Brudno, M. (2009) "SHRiMP Accurate Mapping of Short Color-space Reads", *PLoS Computational Biology*, Vol. 5, No. 5, Article e1000386.

[27] Schatz, M.C., Delcher, A.L. and Salzberg, S.L. (2010) "Assembly of large genomes using second-generation sequencing", *Genome Research*, Vol. 20, No. 9, pp. 1165-1173.

[28] Shendure, J. and Ji, H. (2008) "Next-generation DNA sequencing", *Nature Biotechnology*, Vol. 26, No. 10, pp. 1135–1145.

[29] BWA Software, available from: http://bio-bwa.sourceforge.net/ (last visited: 15 December 2012).

[30] BowTie Software, available from: http://BowTie-bio.sourceforge.net/index.shtml (last visited: 15 December 2012).

[31] Illumina Sequencing Systems, available from: http://www.illumina.com/systems/sequencing.ilmn (last visited: 17March 2013).

[32] Wikipedia - Linear Search, available from: https://en.wikipedia.org/wiki/Linear_search (last visited: 17 March 2013).

[33] NGS Alignment Programs, available from: http://lh3lh3.users.sourceforge.net/NGSalign.shtml (last visited: 15 February 2013).

[34] PerM Software, available from: http://code.google.com/p/perm (last visited: 1 February 2013).

[35] SOAP Software, available from: http://soap.genomics.org.cn/soap1/ (last visited: 1 February 2013).

[36] WHAM Software, available from: http://research.cs.wisc.edu/wham/ (last visited: 15 December 2012).