Information Sciences 303 (2015) 120–133

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

BulkAligner: A novel sequence alignment algorithm based on graph theory and Trinity

Junsu Lee [1], Yunku Yeu [1], Hongchan Roh, Youngmi Yoon, Sanghyun Park *

Department of Computer Science, Yonsei University, 50 Yonsei-ro, Sinchon-dong, Seodaemun-gu, Seoul 120-749, South Korea
Department of Computer Engineering, Gachon University, 1342 Sengnamdaero, Sujeong-gu, Seongnam-si, Gyeonggi-do, South Korea

ARTICLE INFO

Article history:
Received 27 March 2014
Received in revised form 4 January 2015
Accepted 11 January 2015
Available online 17 January 2015

Keywords:
Sequence alignment
Distributed system
Next generation sequencing (NGS)
Graph-based

ABSTRACT

Sequence alignment is a widely-used tool in genomics. With the development of next generation sequencing (NGS) technology, the production of sequence read data has recently increased. A number of read alignment algorithms for handling NGS data have been developed. However, these algorithms suffer from a trade-off between the throughput and alignment quality, due to the large computational costs for processing repeat reads. Conversely, alignment algorithms with distributed systems such as Hadoop and Trinity can obtain a better throughput than existing algorithms on single machine without compromising the alignment quality. In this paper, we suggest BulkAligner, a novel sequence alignment algorithm on the graph-based in-memory distributed system Trinity. We covert the original reference sequence into graph form and perform sequence alignment by finding the longest paths on the graph. Our experimental results show that BulkAligner has at least an 1.8× and up to 57× better throughput with the same, or higher quality than existing algorithms with Hadoop. We analyze the scalability and show that we can obtain a better throughput by simply adding machines.

1. Introduction

In many vital phenomena, genomes represent the most basic unit of information. Genomes convey genetic information from ancestors to descendants, and accumulated polymorphisms play important roles in the evolution of organisms. Sequence alignment is one of the most basic and widely-used tools for the analysis of genome sequences. To borrow a phrase from Mount [22], sequence alignment can be defined as follows.

> "*The sequence alignment algorithm is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences.*"

Sequence alignment algorithms are widely used in genome re-sequencing [10,15], searching for genome polymorphisms [4], and TFBS (Transcription Factor Binding Site) finding [17,25].

* Corresponding author at: Department of Computer Science, Yonsei University, 50 Yonsei-ro, Sinchon-dong, Seodaemun-gu, Seoul 120-749, South Korea. Tel.: +82 2 2123 5714; fax: +82 2 365 2579.

E-mail addresses: solidpple@cs.yonsei.ac.kr (J. Lee), yyk@cs.yonsei.ac.kr (Y. Yeu), fallsmal@gmail.com (H. Roh), ymyoon@gachon.ac.kr (Y. Yoon), Sanghyun@cs.yonsei.ac.kr (S. Park).

[1] The two authors are equally contributed.

http://dx.doi.org/10.1016/j.ins.2015.01.011
0020-0255/

NGS technology has progressed considerably in recent years. As a result, the size of input data has increased explosively [21]. Therefore, sequence alignment algorithms with a high throughput allowing polymorphisms and sequencing errors are now needed. Many of the existing sequence alignment algorithms, such as RMAP [34,35], SOAP2 [16], SOAP3 [19], BWA [13], Bowtie [12], and Bowtie2 [11] are designed for execution on a single machine. However, sequence alignment algorithms allowing for gaps and mismatches have huge computational costs. In order to process mass-produced NGS reads, the quality must be sacrificed to archive an acceptable throughput. As the computing power continues to grow and distributed systems such as Trinity [31] and Hadoop [1] appear, it is possible to obtain a better throughput without compromising the alignment quality (in comparison with existing algorithms operating on a single machine).

Various distributed systems such as Hadoop and Trinity have recently been developed. Hadoop can support parallel programming using the MapReduce [8] framework, which is widely-used in bioinformatics research [38]. However, Hadoop is not suitable for processing algorithms requiring a high throughput, as it must not only convert the original sequence data into another format that can be processed in the Map/Reduce function, but also arouse Disk I/O, since it saves the intermediate and final results to a local disk in the Map/Reduce phase. By contrast, Trinity is a new distributed system supported by Microsoft which is suitable for handling sequence alignment, making it possible to obtain a better throughput by constructing in-memory clusters and processing the graph-form data with parallel computing.

In this paper, we suggest BulkAligner using Trinity as an algorithm for solving the sequence alignment problem. First, BulkAligner converts original reference sequence data into de-bruijn [3] graph form, and builds it into the memory cluster of each slave. Second, in order to perform sequence alignment, BulkAligner extracts some parts of the reference graph to build the read graph for each read. Third, BulkAligner identifies the longest paths allowing some hops in the read graph as candidates, and chooses the best result using glocal alignment [2].

We compare BulkAligner with both CloudBurst [30] and CloudAligner [24], both of which use Hadoop. In the experimental results, BulkAligner exhibits better quality for long reads than short reads. When compared with prior algorithms, the size of the I/O data is bigger in BulkAligner, due to the FASTQ [7] and Sequence Alignment/Map (SAM) [14] formats. Nevertheless, when performing alignment for millions of reads allowing gaps and mismatches, BulkAligner shows a better throughput (1.8–57x) and quality than CloudBurst, and also shows a better throughput by at least 7.8×, and up to 24×, than CloudAligner, with the same quality. Finally, we also prove scalability, an important characteristic of distributed processing algorithms, by scaling the number of nodes in the distributed system.

This paper is organized as follows. The next section presents the existing distributed systems and previous sequence alignment algorithms. Section 3 describes the suggested basic sequence alignment algorithm model, and provides examples of our method. Section 4 explains the experimental environments and data, displays the BulkAligner results according to the parameters, and compares its performance with that of existing algorithms. The last section concludes the paper.

2. Related works

2.1. Distributed systems

There are currently two types of representative distributed processing frameworks. The most popular is the Map/Reduce framework, which was first proposed by Google and later open-sourced by the Apache Hadoop project. There are now numerous distributed processing platforms, such as impala [6], Tajo [5], and Hive [37], as well as others based on the Hadoop Map/Reduce and Hadoop Distributed File System (HDFS) [33]. However, all have common limitations, including expensive costs for conversion to the Map/Reduce framework and disk-based I/O, due to the common core components of Hadoop Map/Reduce and HDFS. Second, following the advent of the big data era prompted by Hadoop ecosystems, Google announced a distributed processing platform suitable for large-scale graph processing, called Pregel [20]. Pregel supports native graph processing operations, such as graph exploration (i.e., message passing) and aggregation, replacing the Map and Reduce operations in Hadoop. Moreover, Pregel can process an entire graph using in-memory clusters, excluding disk I/O. This framework is much more efficient for graph processing than the Map/Reduce framework, given that graph processing inherently requires frequent random accesses from vertice to vertice, creating numerous disk accesses. Map/Reduce consists of repeated sort-and-merge operations that are suitable for the batch processing of large chunks of data such as web pages, but not for exploring distant nodes and edges in a large graph. Other graph distributed processing platforms inspired by Pregel, such as Trinity, GraphLab [18], and PowerGraph [9], have recently been proposed.

In terms of sequence alignment, it is much less expensive to convert an original sequence into a graph operation than to convert it into a Map/Reduce operation. Map/Reduce is so different from the traditional programming paradigm that many existing algorithms would need to be re-written. Moreover, the size of the sequence data is manageable for the in-memory cluster systems. Typically, even in a project with a single genome, the entire sequence data is very large (more than hundreds of gigabytes), while the sequence alignment algorithms can only load dozens of gigabyte data as indexed sequences into the memory. The rest of the sequence data are processed one after another as queries. Therefore, the whole input data used in sequence alignment can be loaded into the in-memory clusters of several desktop machines.

For these reasons, we chose a graph data distributed processing platform for NGS sequence alignment offering both a better throughput and higher quality. Our algorithm can be flexibly implemented in any graph distributed processing platform. However, for now, Trinity remains the best option, as the other platforms are either under development or are not yet open-sourced. Moreover, analyses of the graph data distributed systems also support the superior efficiency of Trinity [32].

2.2. Existing sequence alignment algorithms

A number of single machine-based algorithms such as RMAP, SOAP3, BWA and Bowtie2, have been developed for NGS sequence alignment. Due to polymorphisms and repeats in the genome, the latter compromise the alignment quality in order to achieve an acceptable throughput. A toolkit was proposed to execute those algorithms in a Hadoop cluster [26]; however, this does not offer a fundamental solution for utilizing the power of distributed systems. The recent advances in distributed systems have provided an opportunity for the improvement of both the alignment quality and throughput.

The existing sequence alignment algorithms with distributed systems are CloudBurst and CloudAligner. Both use seed-and-extend alignment techniques like RMAP to handle the sequence alignment. CloudBurst is the first sequence alignment algorithm with Hadoop, while CloudAligner is designed to process longer reads, and demonstrates better quality and throughput than CloudBurst. However, both algorithms use Map/Reduce, in which the map function performs filtering and sorting, and the reduce function performs summary operation. In this process, a lot of Disk I/O arises as the intermediate results are saved to the local disk and the final result to the HDFS in the process of the mapping and reduction functions, respectively. As a result, the two algorithms are unable to ensure a fast throughput. Although Hadoop using MapReduce is designed to process big data sets, it may not be suitable for algorithms requiring a high throughput.

3. Graph-based sequence alignment

Fig. 1 presents an overview of the BulkAligner algorithm suggested in this paper. BulkAligner comprises two steps. In the first step, each slave transforms the original reference sequence data into graph-form data, and builds the graph-form data into its own memory (Section 3.1). In the second step, the client sends read queries to each slave and the latter perform sequence alignment according to the read queries (Section 3.2). Finally, these results are aggregated at the client (proxy) level.

3.1. Building the reference graph

To build a reference graph for each slave, we convert the original reference sequence data into a graph representation data format similar to the de-bruijn graph, as shown in Fig. 2. We trim the reference sequence $k-1$ times (where k = the length of the sequence fragment), and divide each trimmed sequence into k-mer sequence fragments in order to obtain the graph-form data. The nodes indicate the k-mer sequence fragments and are indexed so as to be accessed quickly in each slave. An edge is created between two nodes when their k-mers are adjacent in the reference sequence. (In other words, an edge represents a concatenation of two k-mers.) Each edge includes offset information which indicates the position of the adjacent k-mers in the reference. For example, in Fig. 2, the AAC → CTT edge has offsets 1, 10. Offset 1 means the [0–5] position in

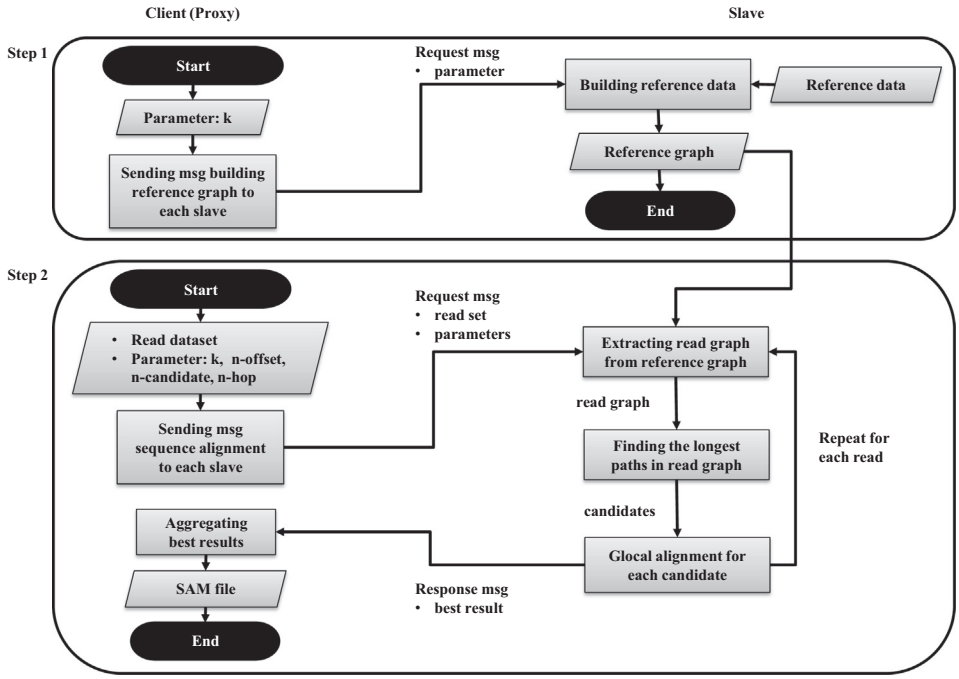

Fig. 1. Overview of BulkAligner. The reference graph-building step is summarized in step 1, and the sequence alignment step is summarized in step 2.

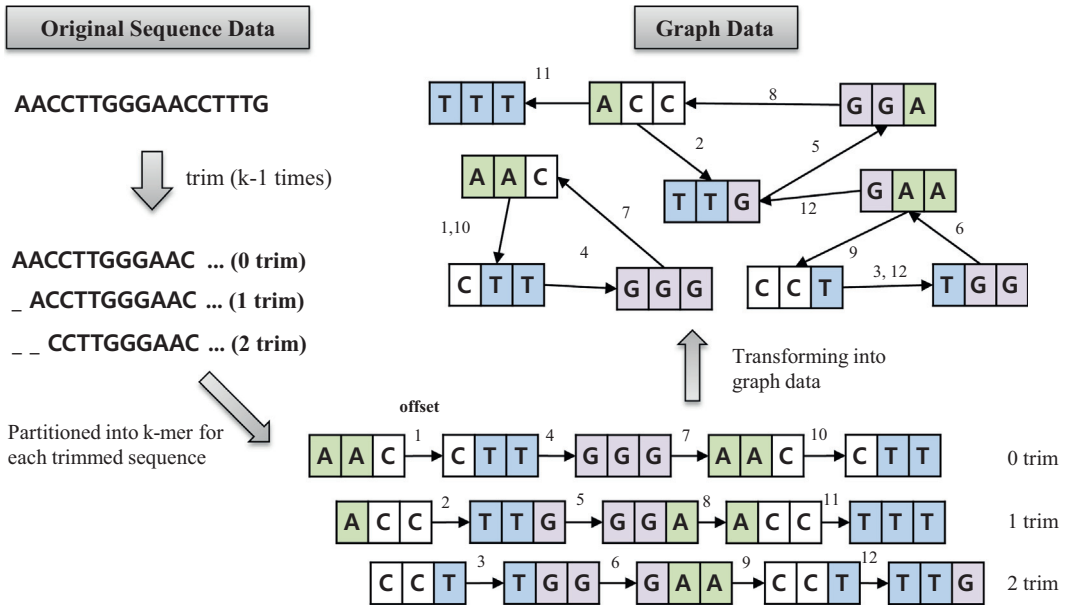

Fig. 2. Transforming the original sequence data into graph data. The original reference sequence is transformed into graph-form data ($k = 3$). The reference sequence is trimmed three times, and each trimmed sequence is divided into a k-mer. Finally, the results are transformed into graph-form data.

the reference sequence, and offset 10 represents the [9–14] position in the reference sequence. Note that 12 offsets and 11 edges are generated from the 18 bp reference sequence. The numbers of generated edges and offsets can differ from one another if more than one offset corresponds to the same edge.

When the length of the reference genome is L_G, the maximum numbers of nodes and edges in the reference graph are 4^k and $min\,(4^{2k}, L_G - 2k)$, respectively. For example, if k is set to 7, a reference graph for the human genome ($3G$ bp) will contain at most 4^7 (= 32,768) nodes and 4^{14} (= 256 mega) edges. The $3G$ offsets are generated from the reference genome and each edge contains an average of 12 offsets. When k is 8, there are at most $L_G - 2k$ edges ($\because 4^{16} > 3G - 2k$), and all edges contain only one offset if the number of edges is $L_G - 2k$. The space and time complexity of the reference graph construction is $O(L_G)$. The k trims can be processed by sliding a window of $2k$ size.

3.2. Sequence alignment

The sequence alignment for the read queries is divided into three sub-steps. In the first sub-step, the read graph is extracted from the reference graph (Section 3.2.1). In the second sub-step, we identify the longest paths allowing n-hops in the obtained read graph as candidates for the final alignment (Section 3.2.2). In the last sub-step, we perform glocal alignment for the candidates in order to find the best results (Section 3.2.3). The process is described in detail below.

3.2.1. Extracting the read graph from the reference graph

We decompose the read query into k-mer units in order to generate a read graph like the reference graph. Each k-mer sequence then becomes a node in the read graph and is used to find the corresponding sequence in the reference graph. After extracting the k-mer nodes, the edge structure is extracted from the reference graph. Fig. 3 illustrates the read graph extraction process. In this example, the read sequence has a substitution and deletion at the 7th and 16th bases, respectively. The substitution and deletion create "missing edges" in the extracted graph. These missing edges will be addressed in the longest path-finding step (Section 3.2.2).

The eukaryote genome features repeats, characterized as multiple copies with very similar genomic content. Some repeat sequences include more than hundreds of thousand matches in the reference sequence. When extracting the read graph from the reference graph, if the entire offset information is copied from the reference graph, some of the edges from the repeat regions can contain a large number of offsets. These repeats exponentially increase the number of combinations in the longest path-finding step. For this reason, we remove the offsets exceeding a certain threshold (*n-offset*).

3.2.2. Finding candidates using the longest paths

After extracting the read graph from the reference graph, BulkAligner identifies the *longest paths* in the read graph. A *path* is defined as a sequence of edges, and the length of a path is defined as the distance from the start position of the first node to the last position of the last node in the path. Each edge in the initial read graph can be regarded as a path of $2k$ length.

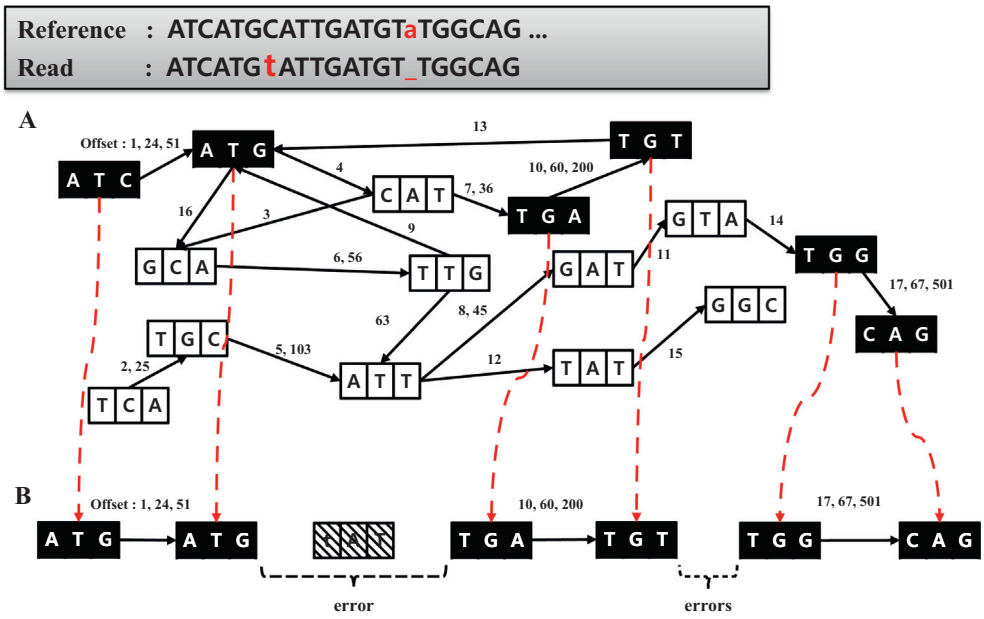

Fig. 3. Extracting the read graph from the reference graph. A is the reference graph; B is the extracted read graph. The black arrows indicate the edges which connect one sequence fragment to another and contain the offset information. Polymorphisms such as insertions, deletions, and substitutions are denoted in red. The black sequence fragments in the reference graph indicate the nodes which are transferred from the reference graph to the read graph, and the red dotted arrows indicate the extracted nodes from the reference graph. The graph below shows the read graph, and the sequence fragments that are disconnected indicate a gap or mismatch. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Given a read graph $G = (N, E)$, a set of nodes $N = \{n_o, n_1, n_2 \ldots, n_{j-1}\}$ (with j: number of nodes in the read graph G), a set of edges $E = \{e_1{:}n_0n_1, e_2{:}n_1n_2 \ldots, e_{j-1}{:}\ n_{j-2}n_{j-1}\}$, and each edge contains a set of offsets.

BulkAligner identifies the longest paths starting from each edge in E. A new path is created from a single edge. The *closest connectable* edge from the contained edges in the path is then searched for, and is added to the path repeatedly. If o_x and o_y represent the offsets of edges e_i and e_j, respectively, then both e_i and e_j are *connectable* if the following two rules are satisfied:

Connectivity rule (1) $\{(e_i{\cdot}o_x, e_j{\cdot}o_y)\ |e_i{\cdot}o_x + k \leqslant e_j{\cdot}o_y\} \neq \emptyset$ (k: length of the sequence fragment k-mer).
Connectivity rule (2) $\{(e_i{\cdot}o_x, e_j{\cdot}o_y)\ |e_j{\cdot}o_y - e_i{\cdot}o_x < L_R\} \neq \emptyset$ (L_R: length of the read).

Rule 1 means that at least one offset of the former edge e_i should be less than one offset of the latter edge e_j. The addition of k enables the calculation of the end offsets of the former edge. Rule 2 means that at least one pair of offsets between the edges has a smaller distance than the length of the read.

By applying the *connectivity* concept, we substitute the alignment problem with that of constructing the longest path by concatenating *connectable* edges. Moreover, in order to identify a path covering most of the read graph, we adopt a "hop" concept as our search scope for finding the next edge. As mentioned in Section 3.2.1, polymorphisms in the read can cause discontinuous offsets and a fragmented read graph. Therefore, if two edges in the read graph are not *connectable*, we jump to the next edge with a jump size of $1 \sim n$ (with n: the threshold value of an n-hop). If an edge can be visited from the last edge of a path under n hops, the path and the edge are *connectable*, and we add a virtual edge between the last edge of the path and the newly-found edge. When an edge is inserted into the path, only the offsets satisfying the *connectivity* rules survive.

Table 1 describes the above process in detail. In order to obtain candidates, we create a path list object (Table 1, line 1) before finding the longest path for each edge in the read graph. We create a path object to store both the list of edges and the polymorphism information (line 4). We identify the *connectivity* between the current edge and the target edge that is less than n-hops from the current edge (line 10). If there is a *connectable* edge, we add the target edge to the path object and set the current edge to the added target edge. If a polymorphism exists between edges, we store the information of the polymorphism into the path object. If there is no possibility of a connection, we add one to the hop. We repeat the above process until the value of *hop* equals *n-hop* (the threshold value), or there are no more target edges (lines 9–18).

When the value of the threshold is 3, we compare the possibility of a connection below 3 hops from e_1. In Fig. 4, because both e_2 and e_3 are not *connectable* to e_1, we compare e_1 with e_4, the next edge from e_3 (the offsets of e_1 and e_4 are 1, 51 and 10, 60, respectively). Since e_4 is a node less than 3 hops from e_1 and the two edges are *connectable*, we add a virtual edge v_1 from the node ATG of e_1 to the node TGA of e_4, and only the corresponding offsets are included in the path. In the next step, e_5 is a missing edge and e_6 is an edge below 2 hops from e_4. Therefore, we compare the possibility of a connection between e_4 and e_6

Table 1
Pseudo-code for finding the longest paths.

Algorithm: Finding the longest paths of BulkAligner

```
Input: n-hop, n-candidate, read graph G
Output: candidates
1.  create a path list object PL
2.  repeat each edge E in G
3.    curEdge ← E
4.    create a new path object P
5.    add curEdge to P
6.    while (curEdge != last edge of G)
7.      hop ← 1
8.      targetEdge ← curEdge + hop    // jump hop times from curEdge
9.      while (hop <= n-hop AND target edge ! = null)
10.       if (connectable (curEdge, targetEdge))
11.         add targetEdge to P
12.         curEdge ← targetEdge;
13.         break
14.       else
15.         hop ← hop + 1
16.         targetEdge ← curEdge + hop
17.       end if
18.     end while
19.     curEdge ← curEdge + 1
20.   end while
21.   add P to path list PL
22. end for
23. call sortLongestPaths (PL, G)
24. candidates = getCandidate (PL, n-candidate)
25. return candidates
```

(offsets of 10, 60 and 17, 67, respectively). Another virtual edge, v_2, is added from node TGT to node TGG. Finally, a path containing $\{e_1, v_1, e_2, v_2, e_6\}$ is stored as one of the candidates.

After identifying the longest path for each node in the read graph, we order them by path length and choose the top *n-candidates* from the longest paths list (lines 23–25).

We now discuss the time complexity of the longest paths-finding algorithm. The longest paths-finding algorithm is an important stage of our method, and consumes most of the processing time. The numbers of offsets in the edges and in the paths represent a major complexity factor for the algorithm. Let us assume that two paths or edges contain m and n offsets, respectively. Keeping the offsets of each path or edge in ascending order allows us to finish the connectivity test for the two paths (or edges) with $(m + n)$ comparisons instead of $(m \times n)$. Note that the offsets are already sorted when the edges are created from the referenced sequence using the sliding window. In addition, when a new path is created by including connectable edges, only the continuous offsets survive. Thus, the cost of the connectivity test between the lengthened path and remaining edges gradually decreases as the length of the path increases.

Before analyzing the complexity of the longest path-finding algorithm, a few terms need to be defined. Assuming that the length of the reference genome is L_G, and the length of the read is L_R, the number of nodes in the read graph will be L_R/k, and the number of edges will be $(L_R/k) - 1$. The average number of offsets per edge (corresponding to a $2k$ sequence) will be $L_G/4^{2k}$.

We first perform an analysis for a best-case scenario in which every pair of adjacent edges is connectable. In the first connectivity test between the first edge and the second edge, $(L_G/4^{2k} + L_G/4^{2k})$ comparisons are performed. In the next test, a newly-created path with $3k$ length and a third edge are examined, and $(L_G/4^{3k} + L_G/4^{2k})$ comparisons are undertaken. The total number of connectivity tests is $(L_R/k) - 1$, and the cost of the best-case scenario is therefore as follows.

$$Cost_{best}(L_G, L_R, k) = \left(\frac{L_G}{4^{2k}} + \frac{L_G}{4^{2k}}\right) + \left(\frac{L_G}{4^{3k}} + \frac{L_G}{4^{2k}}\right) + \left(\frac{L_G}{4^{4k}} + \frac{L_G}{4^{2k}}\right) + \cdots + \left(\frac{L_G}{4^{(\frac{L_R}{k}-1)k}} + \frac{L_G}{4^{2k}}\right) = \sum_{i=2}^{\frac{L_R}{k}-1} \cdot \frac{L_G}{4^{ik}} + \left(\frac{L_R}{k} - 1\right) \cdot \frac{L_G}{4^{2k}}$$

$$= \sum_{i=2}^{\frac{L_R}{k}-1} \frac{L_G}{4^{ik}} - \frac{L_G}{4^{2k}} + \frac{L_R}{k} \cdot \frac{L_G}{4^{2k}} = \sum_{i=3}^{\frac{L_R}{k}-1} \frac{L_G}{4^{ik}} + \frac{L_R}{k} \cdot \frac{L_G}{4^{2k}}$$

The above cost function has two terms, and the right term is a dominant factor in a usual case. If k is greater than 6 in the human reference genome, the denominator for the left term ($4^{18} = 64G$) will become much larger than L_G ($3G$), and the effect of the left term will be negligible.

Second, we consider a worst-case scenario, in which no edges are connectable. In this case, multiple connectivity tests are performed without a decrease in the number of offsets. Each test requires $(L_G/4^{2k} + L_G/4^{2k})$ comparisons. Assuming that the algorithm always hops to the last edges in the worst-case scenario, it therefore performs the connectivity test

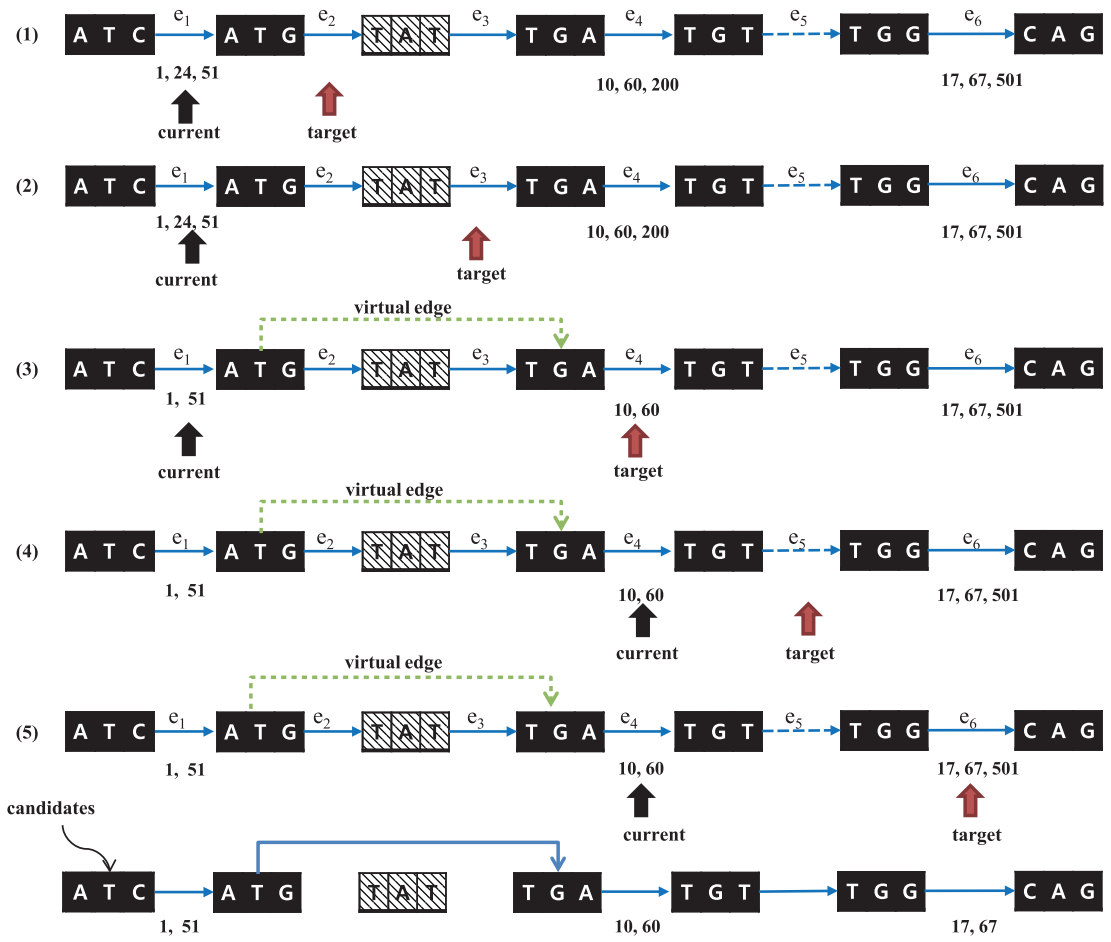

Fig. 4. Finding the longest paths in the read graph. This figure shows the addition of new virtual paths into the extracted initial graph, allowing errors below n-hops, as well as the obtained candidates. The figure describes a 3-hop case. The blue line indicates the relation between the sequence fragments, and the blue dotted line and diagonal rectangles represent gaps or mismatches. The green dotted line is the range of the hop allowed in the node. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

$(L_R/k)-2, (L_R/k)-3, (L_R/k)-4, \ldots, 1, 0$ times for the 1st, 2nd, 3rd, $\ldots$, $(L_R/k)-2$th, $(L_R/k)-1$th edges, respectively. The cost of the worst case is therefore as follows.

$$Cost_{worst}(L_G, L_R, k) = \frac{2L_G}{4^{2k}} \cdot \left(\frac{L_R}{k} - 2\right) + \frac{2L_G}{4^{2k}} \cdot \left(\frac{L_R}{k} - 3\right) + \frac{2L_G}{4^{2k}} \cdot \left(\frac{L_R}{k} - 4\right) + \cdots + \frac{2L_G}{4^{2k}} = \frac{\left(\frac{L_R}{k} - 2\right) \cdot \left(\frac{2L_G}{4^{2k}} + \frac{2L_G}{4^{2k}} \cdot \left(\frac{L_R}{k} - 2\right)\right)}{2}$$
$$= \frac{\left(\frac{L_R}{k} - 1\right) \cdot \left(\frac{L_R}{k} - 2\right) \cdot \frac{2L_G}{4^{2k}}}{2} = \left(\frac{L_R}{k} - 1\right) \cdot \left(\frac{L_R}{k} - 2\right) \cdot \frac{L_G}{4^{2k}}$$

As demonstrated above, the complexity of the longest paths-finding algorithm decreases exponentially as k increases. The analysis of the computational costs for BulkAligner is described in Section 3.3.

3.2.3. Glocal alignment

Glocal alignment is employed to identify the best candidate with the highest score among the candidates obtained from the previous process. Glocal alignment is an algorithm combining global alignment [23] and local alignment [36]. This algorithm constructs an $m \times n$ matrix (where m and n are the lengths of the two sequences, respectively), and performs sequence alignment using dynamic programming.

Fig. 5 provides an example of glocal alignment. Glocal alignment performs local alignment and global alignment depending on the situation. Global alignment is used for alignment between the seeds (exact matched sequences), while local alignment is used for the front of the seeds at the head and the rear of the seeds at the tail. Referring to the example in Fig. 4, e_1, e_4, and e_6 are the seeds, and v_1 and v_2 are the middle areas of the seeds.

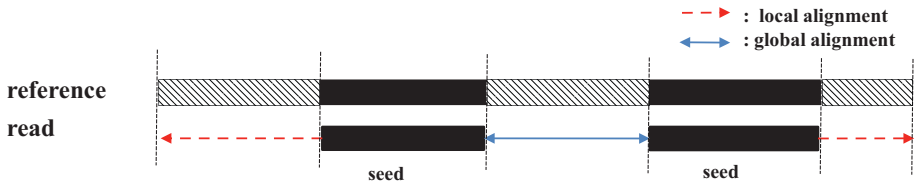

Fig. 5. Glocal alignment for polymorphisms including insertions, deletions, and substitutions. The figure provides an example of glocal alignment according to the error position. "Seed" indicates the sequence fragments connected correctly, without polymorphisms. The diagonal rectangles are unmatched regions due to gaps or mismatches, and the dotted red arrows and double-headed blue arrows are parts of the local alignment and global alignment, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.3. Time complexity analysis of BulkAligner and existing methods

In this section, we discuss the time complexity of BulkAligner and the existing methods exploiting suffix array as an index structure.

Suffix array can be used as an index structure for the exact matching of k-mer with $O(k)$ time complexity. In order to align the reads allowing indels (insertions and deletions) and substitutions, BWA and Bowtie2 exploit the seed and extend paradigm. They find the seed(s) with suffix array first, and extend the seed(s) using local alignment.

BWA consists of three different algorithms: BWA-backtrack, BWA-SW, and BWA-MEM. BWA-MEM is the latest algorithm, and its detailed algorithm has not been published yet. We therefore base our complexity comparison on BWA-backtrack. BWA-backtrack extracts a seed from the front of the read and maps the seed, allowing indels and substitutions. During the seed mapping, it traverses 9 cases for every position of the seed (insertion of A, T, G, C at current position, deletion at current position, and substitution at current position). As it usually allows two differences in the seed at most, its seed-mapping cost represents a maximum of $9^2 \times k$ (seed length). After seed mapping, it converts the SA (suffix array) range of the seed into real offsets and extends each offset using local alignment.

Bowtie2 extracts seeds with regular intervals and maps all seeds in an exact manner. It then determines priority according to the SA range of the seeds. Finally, part of the SA ranges are converted into real offsets and extended using local alignment. The seed-mapping cost of Bowtie2 is $O(k)$.

The read mapping algorithms using suffix array extend individual seeds rather than connect them, meaning that a relatively high number of offsets are included at the extension stage. Particularly, the seeds in BWA-backtrack contain large sets of offsets as they allow indels and substitutions into the seed mapping. Due to these larger numbers of offsets, more local alignment is executed. The cost of the local alignment is $O({L_R}^2)$.

On the other hand, our proposed method maps each seed in $O(1)$ time, using a hash index. Although it involves additional costs for creating the paths from the seeds, the number of offsets finally included and the number of executions of local alignment are minimized. SA-based algorithms such as Bowtie2 typically do not connect the seeds, as the offsets converted from the SA range are not in sorted order. If the offsets are not sorted, the cost of the connectivity test increases to $O(mn)$, or an additional sorting process is needed. Our method maintains the offsets in ascending order and performs the connectivity test in $O(m + n)$.

Table 2 compares the costs of the essential steps in the three algorithms. n_k and n_p represent the number of k-mers and the number of paths, respectively. O_k, O_p and O_{gap} represent the average number of offsets for a k-mer, a path, and a seed in BWA-backtrack, respectively. Direct comparison of the costs is difficult, as it is hard to estimate n_p, O_p, O_{gap} and the exact length of the sequence for local alignment.

4. Experimental results and discussion

4.1. Experimental environments

A number of experiments were conducted to measure the accuracy and throughput of BulkAligner, CloudBurst, and CloudAligner. To this end, a Trinity cluster with 6 nodes and one client, and a Hadoop cluster with 6 nodes, were built.

Table 2
Compared algorithm costs.

	BulkAligner	BWA-backtrack	Bowtie2
Seed length (k, default)	7 bp	32 bp	22 bp
Number of k-mers (seeds)	n_k	1	n_k
Cost for seed mappings	$n_k \times 1$ (Hashing)	$1 \times 9^2 k$ (Gapped mapping using SA)	$n_k \times k$ (Exact mapping using SA)
Cost for create path	${n_k}^2 \times O_k$ (Worst case)	–	–
Number of local alignment executions	$n_p \times O_p$ ($n_p < n_k$, $O_p < O_k$)	$1 \times O_{gap}$ ($O_k < O_{gap}$)	$n_k \times O_k$
Length of sequence for local alignment	Part of read	Whole read	Whole read

All of the experiments were performed on the same machines. Each machine had a 1 TB HDD, 32 GB RAM, and a 3.40 GHz Intel R i73770 processor. The operating systems were 64-bit Windows Server 2008 R2 Enterprise for Trinity, and Ubuntu 12.04.2 for Hadoop.

4.2. Data and measurement descriptions

We simulated reads for chromosome 1 of the human genome (hg18) using the dwgsim program included in the SAM tools package. The format was FASTQ. The lengths of the read sequences were 40, 60, 80, 100, and 120 bp ("base pair", i.e., length of the sequence). For the 120 bp read, we built 6 data sets: 0.5 million read sequences, 1 million read sequences, 1.5 million read sequences, 2 million read sequences, 2.5 million read sequences, and 3 million read sequences.

We employed the elapsed time and quality to measure the performance of the algorithms. The elapsed time was used to assess the throughput. The elapsed time was computed as the total run-time value obtained by adding the alignment time and the Network and Disk I/O time. In order to calculate the quality, we used recall and precision, which were defined as follows:

$$Recall = \frac{mc}{mc + mi + mu} \tag{1}$$

$$Precision = \frac{mc}{mc + mu} \tag{2}$$

The quantity *mc* represents the number of reads mapped correctly to the place where they should be mapped. *mi* is the number of reads mapped incorrectly to a place where they should not be mapped. *mu* is the number of reads unmapped to the place where they should be mapped.

Both *recall* and *precision* are important when comparing the quality of the sequence alignment. We converted both the *precision* and *recall* into one single measure, the *f1-score* [28], which was defined as follows:

$$f1 = 2 \times \frac{Recall \times Precision}{\text{Recall} + \text{Precision}} \tag{3}$$

4.3. BulkAligner performance

The experiments in this section are divided into two parts, so as to analyze the performance changes depending on the value of the parameters of the BulkAligner. The first part evaluates the performance of the algorithm at the reference graph-building stage. The second part evaluates its performance at the sequence alignment stage.

The first part involves a comparison of the quality according to the k values. We set the value of k to 5, 6, 7, and 8. The value of k indicates the size of the sequence fragment when generating the read graph and the reference graph. The more k increases, the more the number of nodes and the number of edges per node in the entire graph increase. However, the number of offsets connecting the sequence fragments diminishes, and the number of offsets per edge eventually decreases. For example, when $k = 5$ (8), the number of nodes goes up to 4^5 (4^8). As a result, when $k = 8$, the number of offsets per edge is smaller than when $k = 5$. Therefore, the more k increases, the less processing speed is available for identification of the longest paths. On the other hand, as k decreases, the number of nodes is reduced and the number of offsets per edge is increased. As a result, when looking for the longest paths, the number of combinations between the offsets is increased.

We set the value of the repeat threshold in the experiment to 200, and evaluate the quality according to the read length and to k. In Fig. 6(a), when k is 5, the results show low quality as the number of nodes is small, and the edges for which the number of offsets are bigger than the repeat threshold are removed. When k is 6, the results demonstrate better quality than when k is equal to 8 at 40 bp. Thus, it can be observed that the results show higher quality when the read is shorter. This is explained by the fact that as k increases, the number of nodes in the read graph decreases. In that case, removing just one node because of polymorphism causes a large loss of information. On the other hand, it is possible to obtain better quality when both k and the length of the read are long. This is due to the fact that the number of nodes in the read graph increases. Therefore, if a node is lost to polymorphism, the information loss is small. In addition, because the number of offsets per edge is small, the discarded information is reduced. In conclusion, as k increases, we obtain better quality for long reads. However, as the number of nodes increases exponentially when the k of the reference graph increases, the size of the reference graph increases while the quality decreases for very short reads.

Next, we evaluate the performance of BulkAligner at the sequence alignment stage. In this step, the quality is compared using a variety of parameters including the offset threshold, the number of candidates, and the interval count.

First, with the read graph extracted from the reference graph, we perform experiments with threshold values (*n-offset*) of 5, 10, 50, 100, 200, 300, 500, and 1000. Generally, the alignment quality is degraded when the *n-offset* is too small or too large. When the value of the offset threshold is too small, a large amount of information for the sequence alignment is removed. Conversely, when the value of the offset threshold is too large, the edges from the repeat reads contain huge numbers of offsets, which cause many ambiguous alignments. Moreover, the large number of offsets generates excessive offset combinations, which increases the computation time for identification of the longest paths.

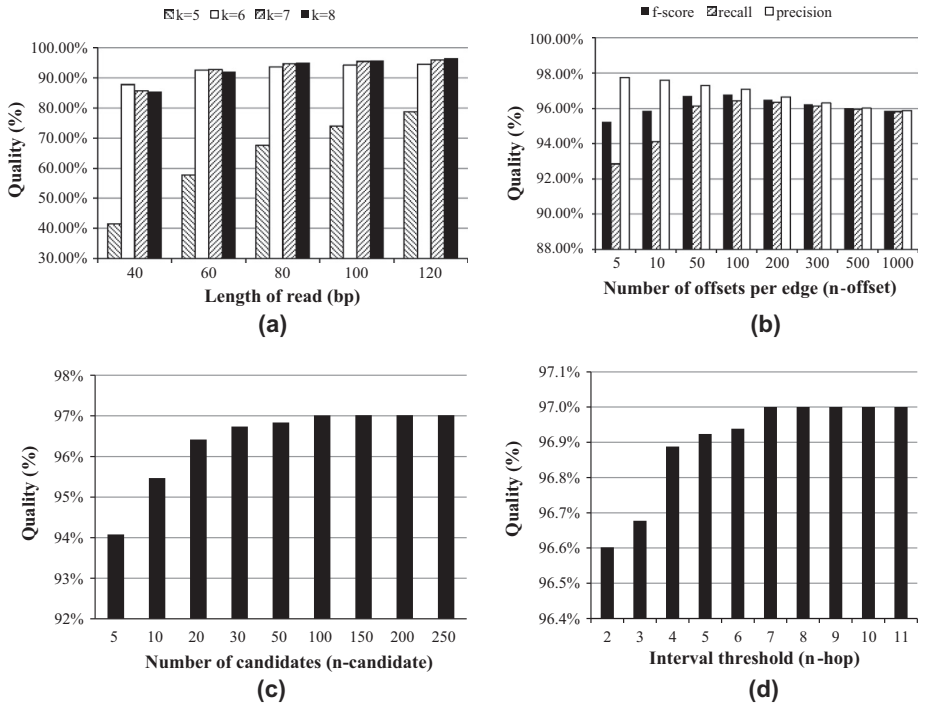

Fig. 6. Evaluation of the BulkAligner quality according to the *f*1-score. (a) *f*1-score according to the *k*-mer fragment size and read length. (b) *f*1-score according to the number of offsets per edge (*n-offset*). (c) *f*1-score according to the number of candidates (*n*-candidate). (d) *f*1-score according to the interval threshold (*n-hop*).

After building the read graph, we compare the performance quality according to the *n*-candidate value obtained at the longest path stage (Fig. 6(c)). The alignment quality is compared with candidate threshold values (*n*-candidate) of 5, 10, 20, 30, 50, 100, 150, 200, and 250. The interval count (*n-hop*) is fixed to 7. As a result, as the number of candidates increases, the quality shows a nearly constant value of 0.97. As a large number of candidates also increases the total execution time, we use a candidate threshold of 100 in the subsequent experiments.

Finally, we test the interval count indicating the scope of the hops. Using a fixed value for the number of candidates (100), we vary the value of the interval count (*n-hop*) between 2 and 11 (Fig. 6(d)). When the interval count is h, any two nodes $h-1$ hops apart are examined in order to find the longest paths. In other words, these two nodes can be connected even though the $h-1$ nodes between them have errors. Therefore, the alignment quality is proportional to the interval count. We show that the quality reaches a maximum value of 0.97 when the interval count is 7.

Lastly, BulkAligner occupies 2.7 GB of memory in each slave.

The values of the parameters used in the performance comparison are summarized in Table 3.

4.4. Comparison with existing methods

First of all, we evaluate BulkAligner and state-of-the-art, stand-alone algorithms with the same simulated read data (100 bp, 1 M). BWA-backtrack and Bowtie2 are selected as the stand-alone algorithms. We compare the recall, precision, and *f*1-score of each algorithm for the dataset.

Table 4 shows the evaluation results for BulkAligner and the stand-alone algorithms. BulkAligner demonstrates a better performance than BWA-backtrack and Bowtie2.

Table 3
Parameters and values used in the BulkAligner comparative experiment.

Step	Parameter	Description	Value
Reference graph building	*k*	Sequence fragment unit	8
Sequence alignment of read queries	Offset threshold	Number of offsets per edge(*n-offset*)	100
	Candidate threshold	Number of candidates for glocal alignment (*n-candidate*)	100
	Interval threshold	Value of allowed interval (*n-hop*)	7

Table 4
Performance comparison between BulkAligner and existing stand-alone algorithms (100 bp simulated read).

Algorithm	Recall	Precision	*f*1-score
BulkAligner	96.85	97.02	96.94
BWA-backtrack	94.24	98.05	96.11
Bowtie2	96.34	97.02	96.68

Table 5
Parameters and values used in the CloudBurst and CloudAligner comparative experiments.

Algorithm	Parameter	Description	Value
CloudBurst	*k*	Number of mismatch differences to allow (higher numbers require more time)	8
	Allowdifferences	0: mismatches only, 1: indels as well	1
	Filteralignments	0: all alignments, 1: only report unambiguous best alignment	0 or 1
	#mappers	Number of mappers to use (suggested: #processor-cores $*$ 10)	240
	#reducer	Number of reducers to use (suggested: #processor-cores $*$ 2)	48
	#fmappers	Number of mappers for filtration alg (suggested:#processor-cores)	24
	#freducers	Number of reducers for filtration alg (suggested:#processor-cores)	24
	Blocksize	Number of queries and reference tuples to consider at a time in the reduce phase (suggested: 128)	128
	Redundancy	Number of copies of low complexity seeds to use (suggested: # processor cores)	24
CloudAligner	Maxmismatches	Number of mismatches	8
	Seed num	Number of seeds (suggested : 11)	11
	Seed weight	Weight of seeds (suggested : 3)	3
	Map mode	0 : map, 1: bisulfite, 2: pair end	0

We then evaluate the BulkAligner performance against that of two existing algorithms, CloudBurst v1.1.0 and CloudAligner v1.9. First, we conduct preprocessing (described later in this section) for the input data. Next, we find the optimal parameters for CloudBurst and CloudAligner. Finally, we compare the quality and throughput of each algorithm using the same dataset in the same experimental environment.

We perform pre/post-processing in order to match the input and output formats of each algorithm. CloudBurst and CloudAligner use FASTA [27] for the input, while BulkAligner uses FASTQ. Therefore, we convert the data in FASTQ format to FASTA format. When FASTQ is transformed into FASTA, the quality value of the sequence in the FASTQ format is deleted, and the size of the data is approximately reduced to 60% (e.g., 1,284 MB → FASTA: 769 MB). Note that the reduced size of the input data affects the total network transfer time.

BulkAligner uses the SAM format for the output, as the latter contains more sequence alignment information than the Browser Extensible Data (BED) [29] format. However, due to this additional information, the size of the SAM format is about 10 times greater than that of the BED format. We directly convert the BED format used by both algorithms by adding the default values of the SAM format that are not contained in the BED format. Finally, SAM is used to compare the final results using Dwgsim-eval.

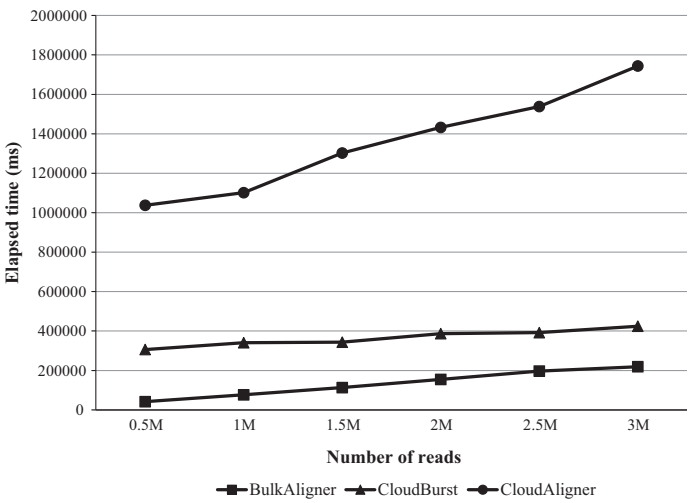

Fig. 7. Comparison of the elapsed time in BulkAligner, CloudBurst, and CloudAligner according to the number of reads.

Table 6
Quality and throughput of each algorithm.

Algorithm	Quality (f1-score)	Throughput (elapsed time, 2 M, ms)
BulkAligner	0.97	154,542
CloudBurst (filteralignments: 0)	0.73	8,590,843
CloudBurst (filteralignments: 1)	0.66	386,767
CloudAligner	0.97	1,432,351

In the pre/post-processing, the size of the input/output data of BulkAligner is larger than that of both CloudBurst and CloudAligner. This feature of BulkAligner causes more Network I/O and Disk I/O than the other algorithms. However, considering future extensibility, BulkAligner chooses formats that contain more information (FASTQ and SAM).

We adjust the parameters in order to determine the optimal parameters for CloudAligner and CloudBurst. The parameters and values used for CloudBurst and CloudAligner are described in Table 5. The values of the parameters are obtained by evaluating the quality and throughput at different numbers of mismatches, with 120-bp reads. CloudBurst and CloudAligner show a tendency toward increased quality and a decreased throughput as the number of mismatches increases. Therefore, we perform experiments changing the value of the number of mismatches from 0 to 9 in order to obtain a high throughput with high quality. At 0.73, CloudBurst shows the highest quality when the value of filteralignments is 0 and the number of mismatches is 8. However, it shows a very low throughput as it reports all the alignment results. For example, an input of 1 (2) million reads generates an output of 56 (113) million alignments. A value of filteralignments set to 1 produces the highest quality (0.66), allowing for 4 mismatches. This also gives a low recall and a high throughput, as only the unambiguous, best alignments are reported. Conversely, CloudAligner achieves the highest quality (0.97) when the number of mismatches is 8 or 9. We therefore set the mismatch value to 8, as it generates a better throughput with the same quality.

In order to evaluate the quality and throughput of the three algorithms, BulkAligner uses the values of the parameters in Table 3, while CloudBurst and CloudAligner use the values of the parameters in Table 5. We select simulation read data numbers of 0.5 M, 1 M, 1.5 M, 2 M, 2.5 M and 3 M for 6 nodes showing the same performance. The experimental results are summarized in Fig. 7 and Table 6. In Fig. 7, the value of the throughput of CloudBurst (filteralignments: 0) is not considered, as CloudBurst (filteralignments: 0) shows a significantly lower throughput than BulkAligner, CloudBurst (filteralignments: 1), and CloudAligner. The experimental results demonstrate that BulkAligner obtains at least a 0.24 (and up to 0.31) higher quality than CloudBurst. Moreover, BulkAligner shows a 1.8–51 times better throughput than CloudBurst. As compared with CloudAligner, BulkAligner achieves a 7.8–24 times better throughput, with a same quality of 0.97. In summary, BulkAligner produces a better quality and throughput than the other algorithms.

Finally, in order to verify the scalability – an important feature of distributed system-based algorithms, we perform experiments for 120 bp, 2 M reads by increasing the number of slaves from 3 to 7. These experiments show that the elapsed time decreases by simply adding machines. From this result, we can conclude that BulkAligner is scalable (Fig. 8). The CloudAligner throughput is not considered in the graph in Fig. 8, as its elapsed time shows a large difference with those of BulkAligner and CloudBurst (filteralignments: 1).

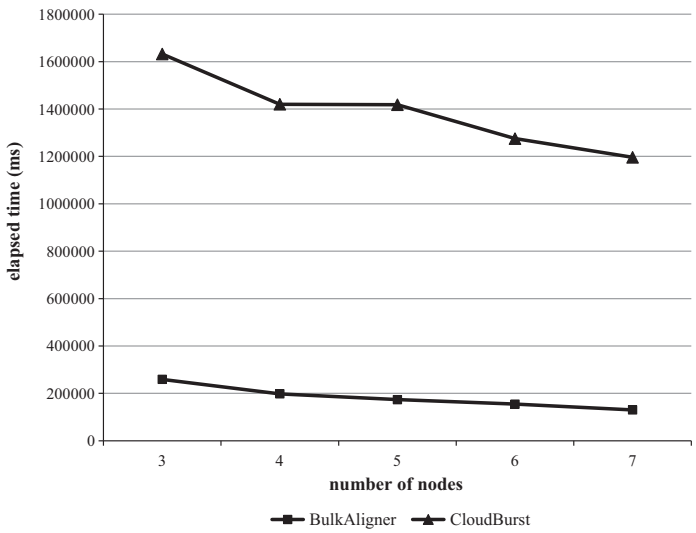

Fig. 8. Comparison between BulkAligner and CloudBurst elapsed time according to the number of machines.

5. Conclusion

With the advent of NGS technology, the amount of available sequence data has grown exponentially. Due to the vast amounts of data and the repeat issue, existing single machine-based sequence alignment algorithms have sacrificed the alignment quality in order to achieve an acceptable throughput. While this trade-off is unavoidable, it can be mitigated by exploiting the high computing power available in distributed systems.

In this study, we surveyed the existing distributed systems in order to identify appropriate systems for the sequence alignment problem. Hadoop, a widely-used distributed system, makes it difficult to obtain a high throughput in our domain as it exploits the MapReduce platform, which presents the disadvantages of saving intermediate results to its own local disk and converting data into a format that can be processed by the Map/Reduce function. Conversely, Trinity can obtain a better throughput than Hadoop, as it is based on an in-memory cluster and can transform/process network-form data directly. For these reasons, we solved the sequence alignment problem using Trinity. To our knowledge, no prior study has tried to perform sequence alignment algorithms on graph-based, in-memory distributed systems.

We suggested a BulkAligner algorithm capable of dealing with next-generation sequence data in a parallel manner on Trinity. BulkAligner converts original reference sequence data into de-bruijn-based graph-form, and extracts the read graph from the reference graph using a reads sequence. BulkAligner then identifies the longest paths allowing hops in the read graph, and chooses the top n longest paths as candidates. Finally, the best results are selected among the candidates through glocal alignment.

In this study's experiments, BulkAligner demonstrated an effective sequence alignment performance allowing polymorphisms for longer reads. We compared BulkAligner with two Hadoop-based sequence alignment algorithms, using the criteria of quality and throughput. Despite the disk and network I/O loss caused by the large size of the input/output format in BulkAligner, the latter showed a dramatic improvement on the throughput of existing algorithms, with a similar or higher quality. The scalability was also proven according to the number of worker nodes in Trinity.

Acknowledgements

I would like to thank Bin Shao, Haixun Wang for introducing me to Trinity. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (2012R1A2A1A01010775). Sanghyun Park is corresponding author of this paper.

References

[1] Apache Hadoop. <http://hadoop.apache.org/> (accessed 30.12.14).
[2] M. Brudno, S. Malde, A. Poliakov, C.B. Do, O. Couronne, I. Dubchak, S. Batzoglou, Glocal alignment: finding rearrangements during alignment, Bioinformatics 19 (S1) (2003) i54–i62.
[3] N.G. Bruijn, P. Erdos, A combinatorial problem, K. Ned. Akad. v. Wet. 49 (1946) 758–764.
[4] W.S. Bush, J.H. Moore, Genome-wide association studies, PLoS Comput. Biol. 8 (2012) 12.
[5] H. Choi, J. Son, H. Yang, H. Ryu, B. Lim, S. Kim, Y.D. Chung, Tajo: a distributed data warehouse system on large clusters, in: S.J. Christian, K. Rao, Proc. 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, 2013, pp. 1320–1323.
[6] Cloudera Impala. <http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html/> (accessed 30.11.14).
[7] P.J.A. Cock, J.F. Christopher, G. Naohisa, L.H. Michael, M.R. Peter, The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants, Nucl. Acids Res. 38 (6) (2010) 1767–1771.
[8] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[9] J.E. Gonzlez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: distributed graph-parallel computation on natural graphs, in: V. Amin, T. Chandu, Proc. 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, Hollywood, CA, USA, 2012, p. 2.
[10] X. Huang et al, High-throughput genotyping by whole-genome resequencing, Genome Res. 19 (6) (2009) 1068–1076.
[11] B. Langmead, S.L. Salzberg, Fast gapped-read alignment with Bowtie 2, Nat. Methods 9 (4) (2012) 357–359.
[12] B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biol. 10 (2009) R25.
[13] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[14] H. Li et al, The sequence alignment/map format and SAMtools, Bioinformatics 25 (16) (2009) 2078–2079.
[15] R. Li, Y. Li, X. Fang, H. Yang, J. Wang, K. Kristiansen, J. Wang, SNP detection for massively parallel whole-genome resequencing, Genome Res. 19 (6) (2009) 1124–1132.
[16] R. Li, C. Yu, Y. Li, T.W. Lam, S.M. Yiu, K. Kristiansen, J. Wang, SOAP2: an improved ultrafast tool for short read alignment, Bioinformatics 25 (15) (2009) 1966–1967.
[17] G.G. Loots, I. Ovcharenko, L. Pachter, I. Dubchak, E.M. Rubin, RVista for comparative sequence-based discovery of functional transcription factor binding sites, Genome Res. 12 (5) (2002) 832–839.
[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, GraphLab: a new parallel framework for machine learning, in: P. Grünwald, P. Spirtes, Proc. the 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010, Catalina Island, CA, USA, 2010.
[19] R. Luo et al, SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner, PloS One 8.5 (2013).
[20] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: E. Ahmed, A. Divyakant, Proc. 2010 ACM SIGMOD International Conference on Management of Data, ACM SIGMOD '10, Indianpolis, Indiana, USA, 2010, pp. 135–146.
[21] V. Marx, Biology: the big challenges of big data, Nature 498 (2013) 255–260.
[22] D. Mount, Bioinformatics: Sequence and Genome Analysis, second ed., Cold Spring harbor Laboratory Press, Cold Spring Harbor, NY, 2004.
[23] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (3) (1970) 443–453.
[24] T. Nguyen, W. Shi, D. Ruden, CloudAligner: a fast and full-featured MapReduce based tool for sequence mapping, BMC Res. Notes 4 (1) (2011) 171.
[25] I. Ovcharenko, G.G. Loots, B.M. Giardine, M. How, J. Ma, R.C. Hardison, L. Stubbs, W. Miller, Mulan: multiple-sequence local alignment and visualization for studying function and evolution, Genome Res. 15 (1) (2005) 184–194.
[26] R.V. Pandey, C. Schlötterer, DistMap: a toolkit for distributed short read mapping on a Hadoop cluster, PloS One 8 (2013).

[27] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proc. Nat. Acad. Sci. 85 (8) (1988) 2444–2448.
[28] D.M.W. Powers, Evaluation: from precision, recall and F-measure to ROC, informedness, markedness & correlation, J. Mach. Learn. Technol. 2 (1) (2011) 37–63.
[29] A.R. Quinlan, M.H. Ira, BEDTools: a flexible suite of utilities for comparing genomic features, Bioinformatics 26 (6) (2010) 841–842.
[30] M.C. Schatz, CloudBurst: highly sensitive read mapping with MapReduce, Bioinformatics 25 (11) (2009) 1363–1369.
[31] B. Shao, H. Wang, Y. Li, The trinity graph engine, Microsoft Res. (2012) 54
[32] B. Shao, H. Wang, Y. Li, Trinity: a distributed graph engine on a memory cloud, in: R. Kenneth, S. Divesh, P. Dimitris, P. Stavros, Proc. the 2013 ACM SIGMOD International Conference on Management of Data, ACM SIGMOD '13, New York, NY, 2013, pp. 505–516.
[33] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: M.G. Khatib, X. He, M. Factor, Proc. 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2010, Lake Tahoe, Nevada, USA, 2010, pp. 1–10.
[34] A.D. Smith, W.Y. Chung, E. Hodges, J. Kendall, G. Hannon, J. Hicks, Z. Xuan, M.Q. Zhang, Updates to the RMAP short-read mapping software, Bioinformatics 25.21 (2009) 2841–2842.
[35] A.D. Smith, Z. Xuan, M.Q. Zhang, Using quality scores and longer reads improves accuracy of Solexa read mapping, BMC Bioinformatics 9 (2008) 128.
[36] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, J. Mol. Biol. 147 (1) (1981) 195–197.
[37] A. Thusoo, J.S. Sarma, N. Jain, S. Zheng, C. Prasad, Z. Ning, A. Suresh, L. Hao, M. Raghotham, Hive – a petabyte scale data warehouse using hadoop, in: G. Shahram, D. Umeshwar, Proc. 26th IEEE International Conference on Data Engineering, ICDE 2010, Long Beach, CA, USA, 2010, pp. 996–1005.
[38] Q. Zou, X.B. Li, W.R. Jiang, Z.Y. Lin, G.L. Li, K. Chen, Survey of MapReduce frame operation in bioinformatics, Briefings Bioinformatics 15 (4) (2014) 637–647.