Optimizing the Accuracy of Randomized Embedding for Sequence Alignment

Yiqing Yan, Nimisha Chaturvedi, Raja Appuswamy

Optimizing the Accuracy of Randomized Embedding for Sequence Alignment

1st Yiqing Yan
Data Science Department
EURECOM
yan@eurecom.fr

2nd Nimisha Chaturvedi
ACCELOM
nimisha.chaturvedi@accelom.com

3rd Raja Appuswamy
Data Science Department
EURECOM
raja.appuswamy@eurecom.fr

***Abstract*—Gapped alignment of sequenced data to a reference genome has traditionally been a computationally-intensive task due to the use of edit distance for dealing with indels and mismatches introduced by sequencing. In prior work, we developed Accel-Align [1], a Seed–Embed–Extend (SEE) sequence aligner that uses randomized embedding algorithms to quickly identify optimal candidate locations using Hamming distance rather than edit distance. While Accel-Align provides up to an order of magnitude improvement over state-of-the-art aligners, the randomized nature of embedding can lead to alignment errors resulting in lower precision and recall with downstream variant callers. In this work, we propose several techniques for improving the accuracy of randomized embedding-based sequence alignment. We provide an efficient implementation of these techniques in Accel-Align, and use it to present a comparative evaluation that demonstrates that the accuracy improvements can be achieved without sacrificing performance. Code is accessible in github.com/raja-appuswamy/accel-align-release.**

***Index Terms*—alignment, mapping, embedding**

I. Introduction

Often the first step, and the most time consuming one, in analyzing genomic datasets is sequence alignment-—the process of determining the location in the reference genome of each sequencing read. Despite the fact that modern gapped read aligners like BWA-MEM [2], Bowtie2 [3], Minimap2 [4], SNAP [5] and HISAT2 [6] can map thousands of reads to a reference genome per second, the sheer size of modern short-read sequencing datasets often makes sequence alignment one of the most time consuming steps in genomic data analysis.

The key bottleneck that makes sequence alignment slow is the need to use edit distance for comparing sequenced reads with multiple candidate locations in the the reference genome, combined with the use of affine-gap penalty scoring and soft clipping to identify an alignment with the best score. The high computation complexity of such an edit-distance-based alignment makes exhaustive search of all possible candidate locations infeasible. Instead, modern aligners typically use a Seed-Filter-Extend (SFE) strategy for performing alignment. In SFE, seeds, the subsequences of strings extracted from a read, are used to look up candidate locations in an indexed reference. The candidates are then filtered to eliminate unlikely matches, and the surviving candidates are passed along for edit-distance-based extension.

The filtering step in SFE plays a crucial role in balancing the accuracy and performance of a sequence aligner. Most sequence aligners use elimination-based filtering techniques, like count filtering [7], adjacency filtering [8], or shifted hamming distance [9], that conservatively eliminate candidates without significantly increasing the probability of misalignment due to accidental elimination of a true match. In contrast, in prior work [1], we introduced a new selection-based filtering scheme based on randomized embedding algorithms [10] that can embed strings such that edit distance between original strings can be approximated by the Hamming distance between embedded strings. Using such embedding algorithms, we developed Accel-Align [1], a Seed-Embed-Extend (SEE) sequence aligner that uses the Hamming distance between embedded reference and embedded read to rank and select, rather than eliminate, candidates with a high rank.

While we showed that Accel-Align provides 3–10× improvement over state-of-the-art aligners at comparable accuracy for short reads, we observed that the randomized nature of the embedding algorithm can result in Accel-Align mapping reads to the wrong location in some cases. This, in turn, leads to a drop in accuracy of downstream variant calling using Accel-Align mapped reads. In this paper, we propose several techniques for improving the accuracy of the embedding-based filtering step in SEE-based sequence alignment. We implement these techniques in Accel-Align, and using a comparative evaluation demonstrate that the accuracy improvements can be realized while maintaining the performance advantage over contemporary aligners.

II. Background

In this section, we first provide a brief overview of Accel-Align. Accel-Align processes reads in three stages, namely, seeding, embedding, and extension. During seeding, Accel-Align extracts non-overlapping k-mers from each read and converts to hash values based on a simple modulo-based hash function. By default, Accel-Align sets the k-mer size to 32 to enable a k-mer to fit in a single 64-bit integer, and the hash value to fit in a 32-bit integer. The generated hash value is used to lookup a hash table to identify candidate locations. Once Accel-Align gets the candidate locations by seeding, it identifies optimal candidates based on the theory of low distortion embedding. The goal of embedding is to transform both the reference strings at candidate locations and the read string to embedded forms such that edit distance between

the original strings can be approximated using the Hamming distance of embedded strings. Accel-Align extracts strings of length equal to the read length from the reference genome at each candidate location, transforms these reference strings and the query string (which is the read) into embedded strings, and calculates the Hamming distance between embedded strings. Finally, Accel-Align picks the top two candidate locations with the least Hamming distance as the best candidates and passes them to the extension stage, where a full dynamic-programming-based edit computation is performed to determine the alignment score and CIGAR.

As the focus of this work is on improving the accuracy of embedding, we refer the reader to our prior publication [1] for further details about indexing and seeding which remain unchanged for this work. In the rest of this section, we will provide a detailed description of the randomized embedding algorithm used by Accel-Align, and illustrate how the randomized nature of embedding could potentially contribute to a loss in accuracy. In particular, we have implemented two randomized embedding algorithms in Accel-Align, namely, 3N-Embedding (3NE) that was proposed by Chakraborthy et al. [10], and 2N-Embedding (2NE) which was a modified version of 3NE that we proposed concurrently with Zhang et al. [11]. As Zhang et al. [11] have shown that 2NE and 3NE have similar worst case bounds, and as we have already demonstrated that 2NE, which is the default embedding algorithm in Accel-Align, is faster than 3NE and works well for sequence alignment [1], we only provide a description of 2NE.

Algorithm 1 $2N$-embedding

Input: A string $S \in \{A, C, G, T\}^N$, and 4 random strings $r_A, r_C, r_G, r_T \in \{0, 1\}^{2N}$
Output: The embedded string $S^{'} \in \{A, C, G, T\}^{2N}$

$j \leftarrow 0$
for $i = 0 \rightarrow N - 1$ **do**
    $S^{'}_j \leftarrow S_i$
    $j++$
    if $r_{s[i]j} = 1$ **then**
        $S^{'}_{j+1} \leftarrow S_i$
        $j++$
    end if
end for
for $j = j + 1 \rightarrow 2N - 1$ **do**
    $S^{'}_j \leftarrow P$
end for
return $S^{'}$

Algorithm 1 presents the pseudocode for the 2NE algorithm. The input string is a string of length N consisting of four possible characters (A,C,G,T). The output is a string of length $2N$ with the same alphabets, and potential multiple repeats of a pad character (P). In each iteration, the algorithm appends a character from the input string once or twice to the output string depending on a random binary bit string associated with that character. The net effect of this is that some input characters appear uniquely in the output string, while others are repeated. Chakraborthy et al. showed that given two strings x, y of length N such that $d_E(x, y)$, the edit distance between x and y, is less than K, there exists an embedding function f, such that the distortion $D(x, y) = d_H(f(x), f(y))/d_E(x, y)$ lies in $[1, O(K)]$ with at least 0.99 probability, where $d_H(x, y)$ is the Hamming distance between the embedded strings. In other words, the Hamming distance of embedded strings produced by their 3NE algorithm is at most square of the edit distance between original strings for small values of K. Zhang et al. showed that the worst case behavior of 2NE is similar to 3NE.

Accel-Align relies on the Hamming distances between embedded candidates and the embedded read generated by 2NE to accurately approximate their corresponding edit distances. However, in practice, a "bad" random bit sequence could, in some cases, lead to a large distortion. To illustrate this, let us consider an example with two strings, "CTGACTGA" (#1) and "CTCACTGA" (#2). The two strings have an edit distance of 1. Given below are three different random sequences, and the embedded versions of these two strings for each random sequence. Although the edit distance of the original strings is 1, the Hamming distance between embedded strings can vary dramatically and even be inflated to 11 as shown in example 3.

Example 1: 1 mismatch
Random seq for A: 1110001000101000
Random seq for C: 0010111101000001
Random seq for G: 0010000001110110
Random seq for T: 0110000110101111
Embedded #1: CTTGACCTTGGAPPPP
Embedded #2: CTTCACCTTGGAPPPP

Example 2: 4 mismatches
Random seq for A: 0010111101100100
Random seq for C: 0111010011001100
Random seq for G: 0110001101101001
Random seq for T: 1100100101100000
Embedded #1: CTTGAACTTGGAPPPP
Embedded #2: CTTCCAAACTGGAPPPP

Example 3: 11 mismatches
Random seq for A: 1001011101001111
Random seq for C: 1010110111011000
Random seq for G: 1001100101001100
Random seq for T: 1101111101101110
Embedded #1: CCTGGAACCTTGAAPP
Embedded #2: CCTCACCTTGGAPPPP

Similarly, mismatches or indels at the beginning of strings will also lead to higher distortion than those at the end of a string. For instance, let us consider the string "CTGACTGC". Compared to #1, it differs only in its last character. Thus, the edit distance between them is 1. When the two strings are embedded, the embedded strings will be be identical for the initial set of characters except the last one. When the embedding

algorithm reaches the last character, depending on whether the random bit is 0 or 1, the embedded strings will differ by 1 or at most 2. However, if we consider "ATGACTGA", which also has an edit distance of 1, but differs from #1 in the first character, the Hamming distance of their embedded strings will depend entirely on the random string. For instance, it will be embedded to "AATTGACCTTGGAAPP" using the random string in Example 1, with a Hamming distance of 10, or "ATTGAACTTGGAPPPP" using the random string in Example 2, with a distance of 1.

Accel-Align embeds each candidate reference only once. In the general case, the difference in edit distance between a good candidate location and a bad one is very large. As a result, despite this randomness, the embedded string obtained from a good candidate will typically have a lower Hamming distance than a bad candidate. However, in cases where the difference in edit distance is not large, or in cases with an unfavorable random bit string, this randomness can inflate the Hamming distance of a true match and result in Accel-Align picking a wrong candidate.

III. Methods

In this section, we describe the changes we made to Accel-Align to minimize the accuracy impact of randomization in 2NE without adversely affecting performance.

A. Multiple embedding

A simple strategy for dealing with distortion caused by embedding is to perform embedding multiple times with the goal that a high distortion produced by a "bad" random string will be overridden by a low distortion outcome from another random string. In the context of Accel-Align, this translates into the following per read operations: (i) $C \times R$ embedding operations for embedding C candidate locations R times, (ii) embed the query read itself R times, and (iii) $C \times R$ Hamming distance computations to identify the best candidate. A naive implementation of multiple embedding will also require $((C+1) \times 2N \times R)$ bytes of memory per read, where N is the length of read (each of the C candidates and the read itself have to be embedded R times, with each embedding producing a string of length 2N).

During experimentation, we found that the computational and memory requirements of multiple rounds of embedding were high. Thus, we implemented a pipelined version of multiple embedding which works as follows. First, we embed the read R times. Then, we process each candidate one at a time by embedding it using a random string and computing the edit distance from the embedded read based on the same random string. After computing R Hamming distances, we only keep the minimum Hamming distance per candidate, and use this to identify the candidate with the lowest overall minimum. We adopted this approach as it integrates seamlessly with two lower-level optimizations already performed by Accel-Align.

First, the embedding algorithm in Accel-Align does not generate the entire embedded string for each candidate. Rather, given a candidate location and random string, it generates one embedded character at a time, compares it with the corresponding character in the embedded read, updates the Hamming distance, and discards the character. This results in a CPU-cache-efficient embedding implementation. Second, the embedding algorithm is parameterized with a threshold so that it stops embedding as soon the threshold is exceeded. Instead of storing the embedded distance of all candidates, Accel-Align already dynamically tracks the lowest and second-lowest distances, and uses the latter as the threshold parameter. Our pipelined implementation of multiple embedding exploits both these optimizations to efficiently track the minimum embedding distance for each candidate.

Algorithm 2 Embedding

Input: A reference string $R \in \{A,C,G,T\}^m$, a querying read string $Q \in \{A,C,G,T\}^n$, a normalized candidate start position s_r with $|M|$ matches between the reference and read whose corresponding start and end indexes are s_{ri}, e_{ri} and s_{qi}, e_{qi}, and N times to embed

Output: The candidate's embedded Hamming distance d

$d_{min} = MAX$
for $l = 0 \leftarrow N-1$ **do**
  $j = 0$, string $\hat{Q}$, $\hat{R}$
  for $k = 0 \rightarrow n-1$ **do**
    if $k \notin [s_{qi}, e_{qi}), \forall i \in [0, |M|)$ **then**
      $\hat{Q}_j = \hat{Q}_k$
      $\hat{R}_j = \hat{R}_{s_r+k}$
      $++j$
    end if
  end for
  string $\hat{Q}', \hat{R}' \leftarrow$ the embedded string of $\hat{Q}$, $\hat{R}$
  $d \leftarrow$ the Hamming distance between $\hat{Q}'$ and $\hat{R}'$
  $d_{min} = min(d, d_{min})$
  if $d_{min} == 0$ or $d_{min} == 1$ **then**
    return d_{min}
  end if
end for
return d_{min}

We further optimize embedding by doing an early-stop for a candidate as soon as we find a random string under which the Hamming distance is computed to be less than or equal to 1. If the embedded Hamming distance is 0, the two embedded strings must be the same, so the original strings are same and edit distance is 0. If the embedded Hamming distance is 1, there is 1 bit different in the embedded strings, same for the original strings, and the edit distance is 1. In either case, there is no need to do an additional round of embedding with a different random string as we have already found the minimum distance.

B. Chain embedding

Let us consider the string x to represent a read, and string y to represent a candidate in the reference genome. Originally, Accel-Align embedded the entire read and an entire candidate string of length equal to the read. However, any candidate y identified by Accel-Align must have at least one k-mer that

produced an exact match between the reference and the read which led to this candidate being identified as a potential match during seeding. If two strings are identical, their edit distance, and hence their embedded Hamming distance, will be zero. Thus, the embedded Hamming distance of all exact matching k-mers would already be zero. This implies that we only need to embed the non-matching parts of the read and the reference. We refer to such an approach as *chain embedding*, as it is reminiscent of the way aligners like Minimap2 use chaining to align gaps between exact matching regions.

Chain embedding improves both performance and accuracy. It improves performance as it reduces the length of the string that needs to be embedded. On the accuracy front, as mentioned earlier, the distortion of the randomized embedding algorithm depends on the edit distance value K. For any read x and a reference candidate y, let x_i represents the i-th non-matching substring in the read, y_i represents the corresponding non-matching part in the reference. These substrings are the parts that are found outside or between exact matching k-mers. The edit distance between them $d_E(x_i, y_i)$ is K_i, and $\sum d_E(x_i, y_i) = d_E(x, y) = K$. Original Accel-Align embeds x and y as a whole. Thus, the overall distortion is bounded by $[K, O(K^2)]$. Our modified Accel-Align with chain embedding, in contrast, embeds each substring separately. As each K_i is smaller than K, this should lower the distortion for each chain embedding, thereby improving accuracy. Putting together multiple and chain embedding techniques, Algorithm 2 shows the pseudo-code for the improved embedding algorithm.

C. K-mer Selection

Originally, Accel-Align used non-overlapping k-mers from a read as seeds. All candidates identified by seeding would be forwarded to the embedding stage. While this approach provided competitive performance when we performed embedding only once, preliminary evaluation revealed that multiple embedding emerged as a computational overhead in the updated Accel-Align, particularly for single-end reads. On further examination, we found that a small fraction of reads that contain k-mers that mapped to several thousands of candidates disproportionately affected the performance in the single-end case but not in the paired-end case.

For single-end reads with such k-mers, Accel-Align did not apply any filtering and thus ends up embedding each candidate location multiple times. For paired-end reads, however, Accel-Align already does pair-wise filtering by identifying candidates from one read that have a matching pair within the specified distance in the other read. Unlike other aligners, Accel-Align does this pair-wise filtering in a very efficient fashion before the embedding stage. Typically, at the end of seeding, we get two sorted list of candidates, one per read in the pair. Thus, Accel-Align scans the two lists looking for candidates that lie within a configurable distance threshold of each other. Due to the sorted nature of the lists, this operation is done by a quick search for lower and upper bounds as shown in Algorithm 3. This pair-wise filtering eliminated several needless embedding steps resulting in multiple embedding not being an overhead

Algorithm 3 Paired-end filtering

Input: A reference string $R \in \{A, C, G, T\}^m$, a querying pair-end read with mates $Q_1, Q_2 \in \{A, C, G, T\}^n$, and the maximum allowed distance between mates $dist$
Output: A list of normalized candidate position

$\{pos_{1i}\} \leftarrow$ the candidate positions of Q_1 gets by single-end seeding
$\{pos_{2i}\} \leftarrow$ the candidate positions of Q_2 gets by single-end seeding
list l_1, l_2
for pos_{1i} in $\{pos_{1i}\}$ **do**
    $start \leftarrow$ lower bound of pos_{1i} that is not less than $pos_{1i} - dist$
    $end \leftarrow$ upper bound of pos_{1i} that is greater than $pos_{1i} - dist$
    if $start \neq end$ **then**
        add pos_{1i} into list l_1
        add positions between $start$ and end into list l_2
    end if
end for
return l_1, l_2

Algorithm 4 Single-end filtering

Input: A reference string $R \in \{A, C, G, T\}^m$, a querying read string $Q \in \{A, C, G, T\}^n$
Output: A list of normalized candidate positions

$S_1, S_2, ..., S_k \leftarrow$ get non-overlapping seeds from Q
$cnt = 0$
for $i = 0 \leftarrow k - 1$ **do**
    $C_i \leftarrow$ count the number of normalized candidate position of seed S_i
    $pos_{i1}, pos_{i2}, ..., pos_{iC_i} \leftarrow$ get normalized candidate position of seed S_i
    if $C_i > 1000$ **then**
        $++cnt$
    end if
end for
if $cnt > k/2$ **then**
    $\{pos_{ij}\} \leftarrow$ all the positions mapped by $S_1, S_2, ..., S_k$
else
    $\{pos_{ij}\} \leftarrow$ the positions mapped by the seeds whose $C_i < 1000$
end if
if exist a position occurs more than once in $\{pos_{ij}\}$ **then**
    return positions occur more than once in $\{pos_{ij}\}$
else
    return $\{pos_{ij}\}$
end if

for paired-end reads. Interestingly, this approach, also referred to as fuzzy set matching, has also been implemented recently as an effective filter by SNAP [12] and URMAP [13].

In order to optimize the performance for single-end reads also, we modify Accel-Align to adopt judicious k-mer selection similar to FastHASH [8], BWA-MEM and Minimap2.

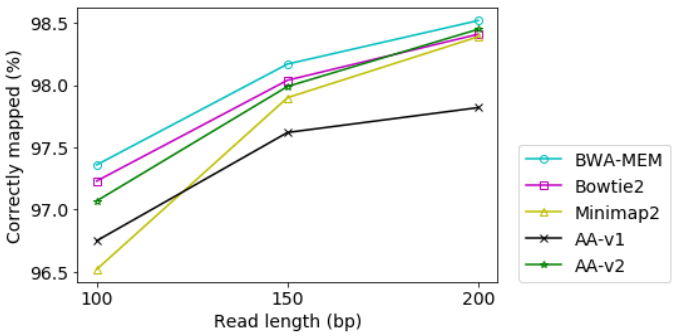

Fig. 1: Accuracy of single-end simulated datasets.

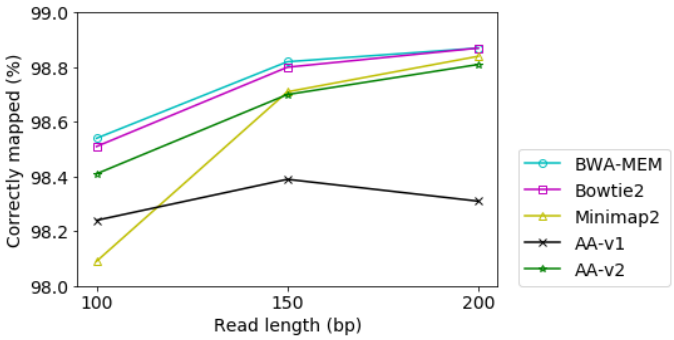

Fig. 2: Accuracy of pair-end simulated datasets.

During seeding, Accel-Align uses the number of candidates generated per seed to classify a read. In particular, if more than half of seeds in the read have more than a threshold count of candidates (default 1000 similar to Minimap2), we flag it as potentially belonging to a highly repetitive region. For such reads, we do not apply any filtering, and forward all the candidates to the embedding stage. However, for reads that are not flagged, we apply k-mer selection by only taking into account k-mers whose candidate count falls below the threshold. The overall algorithm in given in Algorithm 4.

D. Extension and MAPQ

Accel-Align can be configured to run in alignment-free mapping mode where only the identified candidate location is reported, or full-alignment mode where base-by-base extension is performed and the CIGAR string is reported. For the mapping mode, we pick the best candidate, which is the one with the least embedding distance, as the target. Then, we embed the first seed of the read and the k-mers in reference genome at multiple positions around the final candidate position, and pick the position with the least embedding distance. This is done to take into account indels in the first few characters of a read.

For the full-alignment mode, originally Accel-Align did a global alignment between the read and the substring of same length in the reference's candidate position. But we found the lack of soft clipping to adversely affect accuracy of downstream variant calling. So the updated Accel-Align now extracts a substring of length longer than read length and performs "glocal" alignment using lib-ksw [14] on either

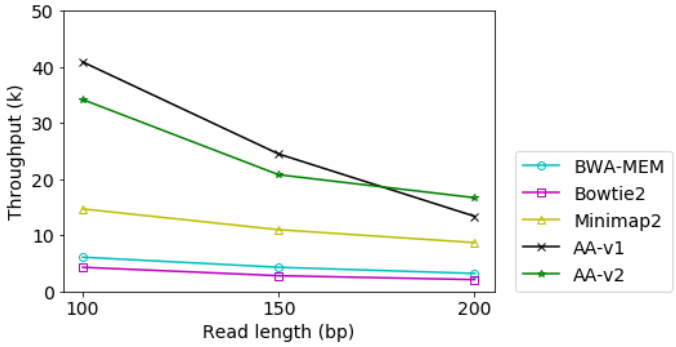

Fig. 3: Throughput for single-end simulated reads.

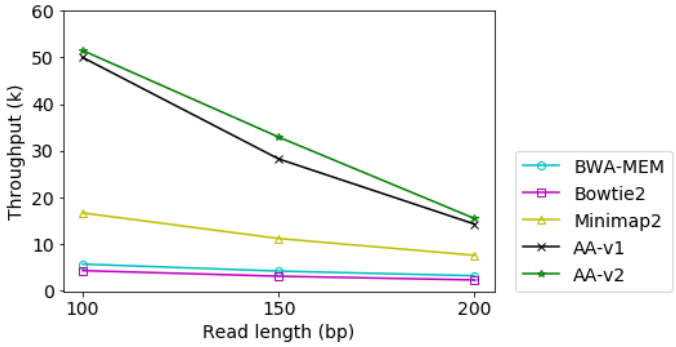

Fig. 4: Throughput for pair-end simulated reads.

end to support soft clipping. The matching score is set to 2, mismatching, gap-open and gap-extension penalty are set to 8, 12, 2, to compute the alignment score and CIGAR.

In addition to the CIGAR, Accel-Align also reports a mapping quality (MAPQ) that represents the degree of confidence in the alignment for each read. Previously, Accel-Align used Bowtie2's MAPQ estimation procedure [3] adjusted to use the embedded Hamming distance of the top two candidates in order to produce a MAPQ value between 0 and 42. However, using embedded Hamming distance for MAPQ makes the aligner more conservative in estimating MAPQ which led slightly lower accuracy during downstream variant calling. Further, with the addition of chain embedding to Accel-Align, we no longer compute the end-to-end Hamming distance. Thus, we have simplified and updated the MAPQ estimation procedure. Accel-Align now uses the cumulative Hamming distance obtained from chain embedding for identifying the top two candidates. If d_1 is the least embedded Hamming distance and d_2 is the second least, the MAPQ is computed as $MAPQ = 60 * (1 - d_1/d_2)^2$.

IV. Results

In this section, we compare the accuracy and performance of the updated Accel-Align (v2), with both the older version of Accel-Align (v1), and three other state-of-the-art short-read aligners, namely, BWA-MEM (v0.7.17), Bowtie2 (v2.3.5), Minimap2 (v2.17). All experiments are performed on a server equipped with a quad-core Intel(R) Core(TM) i5-7500 CPU clocked at 3.40GHz, 32GB RAM, and a 256GB SATA SSD.

We run each aligner five times, ignored the first "cold" run and report the average time of last four "warm" runs. We also tried to evaluate SNAP (v2.0) and BWA-MEM2 (v2.2.1) [15], but we do not report the results here as they failed due to their peak memory usage exceeding our server capacity.

A. Simulated data.

We used Mason2 [16] to simulate a VCF file with variants with default SNP and indel rates from the hg37 reference genome. We simulated multiple 10M Illumina single-end and paired-end read datasets of length 100bp, 150bp and 200bp.

1) Alignment evaluation: Fig 1 and Fig 2 report the accuracy of various aligners in terms of fraction of reads correctly mapped for various single-end and paired-end datasets. We consider a read to be correctly mapped if the reported alignment and the Mason-provided alignment overlap by at least 90%. From the figures we can make two observations. First, Accel-Align v2 clearly improves accuracy over Accel-Align v1, especially at longer read lengths. Second, Accel-Align v2 provides accuracy better than, or comparable to, Minimap2 under all datasets while lagging behind BWA-MEM or Bowtie2 by less than 0.2%.

We show the throughput, measured as the number of reads processed per second per thread, for all aligners in Fig 3 and Fig 4. We can see that Accel-Align v2 is slightly slower than v1 for 100bp and 150bp single-end datasets. But in the paired-end datasets, Accel-Align v2 provides performance comparable to Accel-Align v1. Analyzing the 10M 100bp pair-end dataset further, we found that there are 630M candidate positions before pair filtering, and only 74M positions after filtering. Thus, as the pair filter eliminates nearly 88% of candidates, it helps to minimize the overhead of multiple embedding. This is not the case for single-end dataset leading to a slow down of Accel-Align v2 over v1. More importantly, comparing Accel-Align v2 with other aligners, we see that it is 4∼9× faster than BWA-MEM, 6∼12× faster than Bowtie2 and 2∼3× faster than Minimap2 similar to Accel-Align v1.

2) Alignment-free mapping evaluation: Accel-Align and Minimap2 provide an alignment-free mapping mode which does not report the CIGAR. Such mapping is useful in the applications that only need the position while not requiring a base-by-base alignment. Hence, we compare the performance of two aligners in alignment-free mode. Fig 5 shows the throughput per second per thread for 100bp, 150bp and 200bp pair-end read. Comparing Fig 5 and Fig 4, we see that the alignment-free mode provides a further 1.15∼1.36× improvement in throughput over full alignment mode. Second, Accel-Align is 2.1∼3.2× faster than Minimap2 at all read lengths (Fig 5) with alignment-free mapping.

3) Multiple embedding microbenchmark: As a key technique to improving the accuracy is multiple embedding, we show the relationship between multiple rounds of embedding (x-axis) and execution time/accuracy (y-axis) in Fig 6 for the 10M 200bp single-end read dataset. We use the alignment-free mode to highlight the impact of embedding. As expected, the execution time increases proportionate to the number of rounds

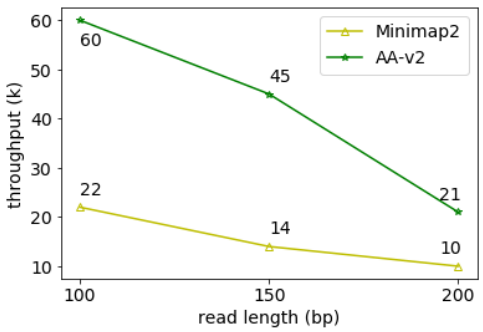

Fig. 5: Alignment-free mapping throughput for pair-end reads.

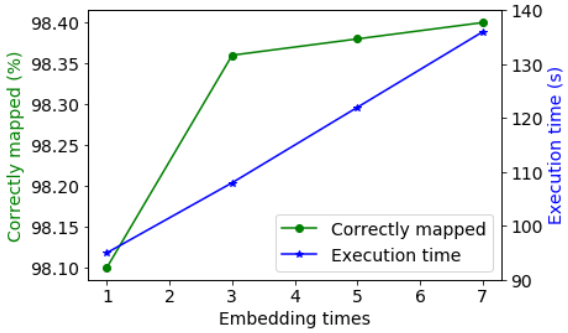

Fig. 6: Performance and accuracy with multiple embedding

of embedding. However, while accuracy improves 0.25% from embedding once to embedding three times, the improvement for higher rounds of embedding is too low to justify the increase in execution time. This is the reason why Accel-Align v2 performs three rounds of embedding by default.

B. Real data.

To evaluate Accel-Align v2's performance on real data, we use a human whole-exome sequencing dataset that has 150M paired-end reads of length 151bp. We align the reads to the hg37 reference genome and use GATK HaplotypeCaller (v4.1.0) [17] for variant calling. We capture variant locations using the SureSelect Human All Exon v2 target capture kit bed file (ELID:S0293689), and validate with high confidence variant calls (v2.19) from Genome in a Bottle (GiaB) containing 23,686 SNVs and 1,258 INDELs.

TABLE I: Evaluation on real data

	BWA-MEM	Bowtie2	Minimap2	AA-v1	AA-v2
Exec. time HH:MM	7:43	7:31	4:43	1:21	1:21
Ti/Tv	2.84	2.86	2.85	2.62	2.85
Overall F-score	0.990	0.992	0.992	0.991	0.992
TP(SNP)	23521	23457	23521	23447	23508
FP(SNP)	235	79	163	872	146
FN(SNP)	165	229	165	239	178
F-score(SNP)	0.991	0.993	0.993	0.993	0.993
TP(InDels)	1223	1213	1223	1194	1223
FP(InDels)	49	30	27	105	30
FN(InDels)	35	45	35	64	35
F-score(InDels)	0.966	0.970	0.975	0.959	0.974
% Mapped	99.4%	97.1%	99.3%	95.5%	95.4%

Table I presents a comparison of various aligners under several metrics. The wall-clock time to align 150M paired-end reads (300M total reads) is reported in the field *Exec. time HH:MM*. Similar to the performance with simulated dataset, Accel-Align v2 is the fastest aligner, providing a 7× speedup over BWA-MEM and Bowtie2, and 3× speedup over Minimap2. The second metric is the transition-to-transversion ratio (Ti/Tv), which should fall in the range of 2.6∼3.3 for this dataset. All aligners satisfied this requirement. We found that Accel-Align v2's result is more consistent than Accel-Align v1's compared to the other three aligners according to the Ti/Tv. As it is illustrated in Table I, the Ti/Tv difference between Accel-Align v2 and other 3 aligners is 0∼0.1, while Accel-Align v1's difference is 0.22∼0.24.

Looking at the overall F-score across SNP and Indel variants, we see that all aligners are comparable. To classify the variants found by each aligner, we list true positive (TP) variants found in both GiaB and by the variant calling pipeline, false positive (FP) variants determined by pipeline but not validated by GiaB, false negative (FN) variants which are validated by GiaB but not determined by pipeline, for SNP and InDels separately. The F-score is computed as 2*TP/(2*TP+FP+TN).

Comparing Accel-Align v1 and v2, we can see that the new version with accuracy improvements is able to capture substantially more SNP and Indel variants. Comparing all aligners, we see that Accel-Align v2 detects the same number of TP InDels as BWA-MEM and Minimap2. In terms of TP SNP variants, Accel-Align v2 detects only 13 fewer variants than BWA-MEM and Minimap2, while several more than Bowtie2. To better understand the difference between detected variants, we show a Venn diagram of variants detected by various aligners in Fig 7 and Fig 8. Majority of SNPs and Indels captured by other aligners are also captured by Accel-Align. There are 10 SNPs and 1 Indel that are concordantly detected by other aligners but not Accel-Align v2. Upon further inspection, we found that the missing variants are mainly due to the use of large k-mer length and non-overlapping seeding which led to 4.6% of reads being not aligned as shown by the % Mapped metric. We plan to optimize the seeding stage as a part of future work.

V. Conclusion

In this work, we presented several strategies to improve the accuracy of randomized embedding in the context of sequence alignment. We implemented these strategies in Accel-Align and showed that it is up to 3× faster than Minimap2, 9× times faster than BWA-MEM, and 12× times faster than Bowie2, while providing comparable accuracy. An area of future work that we are investigating is the utility of Accel-Align for long-read alignment with a higher error rate. Another interesting avenue of future work is combining embedding with more accurate, effective strategy based on spaced seeds [18], minimizers [19], or strobemer [20], to be able to map more reads and thus, be able to identify the missing SNP variants.

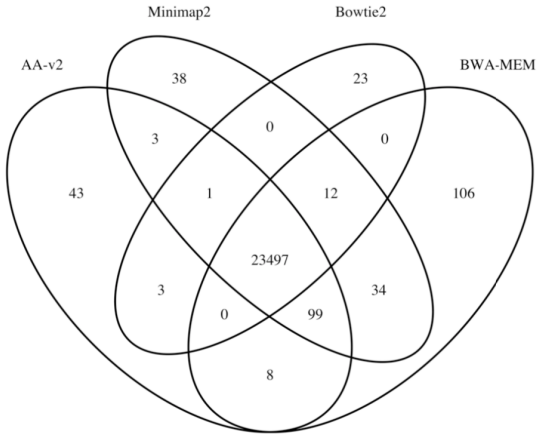

Fig. 7: Venn diagram of SNPs detected by various aligners.

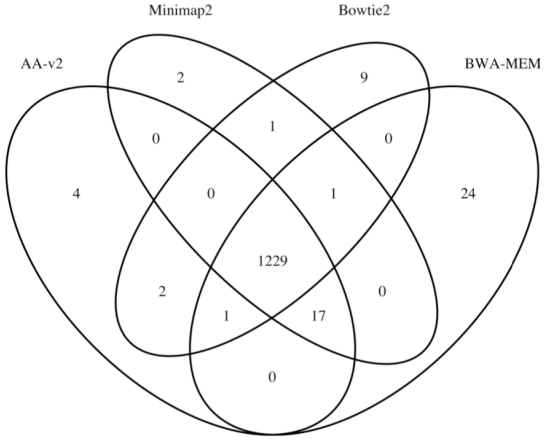

Fig. 8: Venn diagram of InDels detected by various aligners.

VI. Acknowledgements

We are grateful to Matthew Holt for for providing valuable feedback and data for comparative evaluation. This work was partially funded by the European Union's Horizon 2020 research and innovation programme, project OligoArchive, under Grant agreement No. 863320.

References

[1] Y. Yan, N. Chaturvedi, and R. Appuswamy, "Accel-align: a fast sequence mapper and aligner based on the seed–embed–extend method," *BMC bioinformatics*, vol. 22, no. 1, pp. 1–20, 2021.

[2] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[3] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[4] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[5] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with snap," *arXiv preprint arXiv:1111.5572*, 2011.

[6] D. Kim, J. M. Paggi, C. Park, C. Bennett, and S. L. Salzberg, "Graph-based genome alignment and genotyping with hisat2 and hisat-genotype," *Nature biotechnology*, vol. 37, no. 8, pp. 907–915, 2019.

[7] Y. Liao, G. K. Smyth, and W. Shi, "The Subread aligner: fast, accurate and scalable read mapping by seed-and-vote," *Nucleic Acids Research*, vol. 41, no. 10, 2013.

[8] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, no. 1. Springer, 2013, pp. 1–13.

[9] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping," *Bioinformatics*, vol. 31, no. 10, pp. 1553–1560, 2015.

[10] D. Chakraborty, E. Goldenberg, and M. Koucký, "Streaming algorithms for embedding and computing edit distance in the low distance regime," in *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 2016, pp. 712–725.

[11] X. Zhang, Y. Yuan, and P. Indyk, "Neural embeddings for nearest neighbor search under edit distance," 2019.

[12] W. J. Bolosky, A. Subramaniyan, M. Zaharia, R. Pandya, T. Sittler, and D. Patterson, "Fuzzy set intersection based paired-end short-read alignment," *bioRxiv*, 2021.

[13] R. Edgar, "Urmap, an ultra-fast read mapper," *PeerJ*, vol. 8, p. e9338, 2020.

[14] H. Suzuki and M. Kasahara, "Introducing difference recurrence relations for faster semi-global alignment of long sequences," *BMC bioinformatics*, vol. 19, no. 1, pp. 33–47, 2018.

[15] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 314–324.

[16] M. Holtgrewe, "Mason: a read simulator for second generation sequencing data," 2010.

[17] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna *et al.*, "A framework for variation discovery and genotyping using next-generation dna sequencing data," *Nature genetics*, vol. 43, no. 5, pp. 491–498, 2011.

[18] B. Ma, J. Tromp, and M. Li, "Patternhunter: faster and more sensitive homology search," vol. 18, no. 3.

[19] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[20] K. Sahlin, "Effective sequence similarity detection with strobemers," *Genome Research*, vol. 31, no. 11, pp. 2080–2094, Nov. 2021.