Supplemental Material for

Fast sequence alignment for centromere with RaMA

Pinglu Zhang[1,2], Yanming Wei[2,3], Qinzhong Tian[1,2], Quan Zou[1,2], Yansu Wang[1,2,*]

1. Institute of Fundamental and Frontier Sciences, University of Electronic Science and Technology of China, Chengdu, China
2. Yangtze Delta Region Institute (Quzhou), University of Electronic Science and Technology of China, Quzhou, China
3. School of Computer Science and Technology, Xidian University, Xi'an, Shaanxi, China

*Corresponding author: wangyansu@uestc.edu.cn

Supplemental Method

Parameter Settings for Different Alignment Methods

We utilized four methods—minimap2 [1], wavefront alignment (WFA) [2], UniAligner [3], and RaMA—to align the centromere sequences. Below, we provide a detailed explanation of the parameter settings for each method.

For the first method minimap2 (git commit hash 0cc3cdca27f050fb80a19c90d25ecc6ab0b0907b), which is available at https://github.com/lh3/minimap2, we selected the same parameter settings used in previous studies:

```
minimap2 -a -I 15G -K 8G -t thread_num -ax asm20 --secondary=no --eqx -s 2500
ref.fasta query.fasta > output.sam
```

The authors of previous study[4] selected specific minimap2 parameters to optimize alignment for repetitive and structurally divergent regions in diploid human genomes. They used `-I 15G` for additional memory and `-K 8G` to allow loading 8 Gb of sequence at once, accommodating the full genome and preventing alignment bottlenecks. The `-ax asm20` option was chosen to align sequences with up to 20% divergence, suitable for variable α-satellite HOR structures. They used `--secondary=no` to prevent multi-mapping, ensuring each query aligns only once, and `--eqx` to parse CIGAR strings for calculating mean sequence identity. Finally, they set `-s 2500` as the minimal alignment score to avoid spurious alignments while retaining accurate centromere alignments, following tests of various values. After obtaining the SAM alignment results, we used the following samtools[5] command to retain only the primary alignments:

```
samtools view -h -F 256 -F 2048 output.sam -o output_primary.sam
```

The wavefront alignment algorithm (git commit hash cf3eb92dd0aa9bf067d5488a606d8c91173e74eb), which is available at https://github.com/smarco/WFA2-lib, is a recently proposed tool for pairwise sequence alignment that operates in $O(ns)$ time, where n is the read length and s is the alignment score. By leveraging homologous regions between sequences, it accelerates the alignment process, making it significantly faster than traditional dynamic programming methods, particularly for long and noisy reads. We used a 2-piece affine gap cost with the following scoring settings: match = 0, mismatch = 3, gap_open1 = 4, gap_extension1 = 2, gap_open2 = 12, and gap_extension2 = 1, aiming to minimize the total score. Here, gap_open1 and gap_extension1 are the penalties for short gaps, while gap_open2 and gap_extension2 are the penalties for long gaps. WFA is a deterministic algorithm, and we did not employ any optimization strategies, ensuring that the solution obtained is the optimal solution of dynamic programming. In practical use, since RaMA incorporates WFA, we configured RaMA to bypass anchor finding and directly use WFA for centromere alignment, obtaining the WFA results directly.

For wfmash [6], we performed the alignment using its default parameters, with the following command:

```
wfmash ref.fasta query.fasta > output.paf
```

For UniAligner [3], a parameter-free sequence alignment framework, we used the default settings. The alignment command is as follows:

```
tandem_aligner --first first.fasta --second second.fasta -o output_dir
```

The code for UniAligner can be found at https://github.com/seryrzu/unialigner, with the git commit hash c5a1eecab7bd17485a0fe3422684409c3e884f31.

Finally, for our work RaMA, the source code is available at https://github.com/malabz/RaMA, with the used version's git commit hash being 8661cde3ea0e0a4bc22d850c17321878e28f6948. RaMA was run with default parameters, using the following alignment command:

```
RaMA -r /path/to/ref.fasta -q /path/to/query.fasta -o /path/to/output_dir
```

When invoking WFA [2], RaMA used the same 2-piece affine gap cost with the following parameters: match = 0, mismatch = 3, gap open1 = 4, gap extension1 = 2, gap open2 = 12, and gap extension2 = 1, with the aim of minimizing the total score. In this scheme, gap open1 and gap extension1 are the penalties for short gaps, while gap open2 and gap extension2 are the penalties for long gaps.

Creating Simulated Data with Regions Removed from Template

To demonstrate the capability of different methods in capturing the genetic evolution of centromeres, we generated a set of simulated data. We used the X chromosome centromere of CHM13 as the template. The X chromosome sequence of CHM13 can be obtained from https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/assemblies/analysis_set/chm13v2.0.fa.gz, with the centromere located at positions 57,819,763 to 60,927,195. To extract the centromere region of the X chromosome, we used the following command:

```
samtools faidx chm13v2.0.fa chrX:57819763-60927195 > chrX_cen.fa
```

Next, we used HORmon to annotate the extracted centromere sequence. HORmon requires monomers as a basis for annotation, which can be inferred using HORmon's built-in monomer inference program:

```
monomer_inference -seq chrX_cen.fa -mon test_data/AlphaSat.fa -o chrX
```

We directly use the monomers inferred in the HORmon[7], which can be obtained from https://figshare.com/articles/dataset/HORmon/16755097/2. To start the annotation, we use the following command:

```
HORmon --seq chrX_cen.fa --mon cenX_monomers.fa --cen-id X -o chrX_res -t 12
```

The final HOR annotation results are recorded in the `HORdecomposition.tsv` file located in the `chrX_res` folder. We selected two regions: 182368-547935 as

Region 1, and 2417322-2773488 as Region 2. These regions each represent complete HOR blocks. By removing these two regions, we obtained two sequences for the simulated dataset. Therefore, the correct alignment CIGAR result for these two sequences is `182368=365568I1869390=356167D333939=`.

Generating Non-Repetitive Sequences of Different Similarities Using INDELible

In our study, we simulated non-repetitive sequences with varying levels of similarity using INDELible [8]. A script was developed to generate sequences with similarity ranges from 70% to 99%, with divergence times corresponding to each similarity level. The control files for INDELible were configured using the TVM nucleotide substitution model, specified state frequencies, insertion and deletion rates, and a Lavalette indel length distribution. Specifically, the substitution model parameters were set as b=0.01, c=0.01, d=0.04, e=0.04, and a=f=1, with state frequencies of T=0.25, C=0.31, A=0.31, and G=0.13. Rate parameters included pinv=0.84, alpha=1.03, and ngamcat=4. The indel model used a Lavalette distribution with a=5 and M=50, with an insertion rate of 0.01 and a deletion rate of 0.1 relative to a substitution rate of 1. Each sequence was simulated to a length of 1,000,000 nucleotides. This simulation approach generated a robust dataset to evaluate alignment methods, covering a broad spectrum of sequence similarities.

Generating Hybrid Sequence of Tandem Repeats and Non-Tandem Repeats

To explore the performance of RaMA on tandem repeat and non-tandem repeat sequences, we used INDELible to simulate two non-repetitive sequences of 1,000,000 in length with 95% similarity, and then inserted the centromeres from chromosomes 16 and 20 of CHM13 and CHM1 into them. For the parameters used in INDELible, please refer to the section 'Generating Non-Repetitive Sequences of Different Similarities Using INDELible'. We inserted the centromeres from chromosomes 16 and 20 of CHM13 and CHM1 at positions 300,000 and 800,000, respectively. The insertion of these two centromeres divided the entire sequence into five segments. The positions and lengths of these five segments in the two sequences are shown in the Table S6. Segment 1 spans 0–300000 in Seq1 and 0–299983 in Seq2. Centromere 1 covers 300000–1938824 in Seq1 and 299983–1868018 in Seq2. Segment 2 runs from 2238824–2738824 in Seq1 and 2168001–2668036 in Seq2. Centromere 2 extends from 2738824–4912627 in Seq1 and 2668036–5431726 in Seq2. Segment 3 spans 4912627–5106497 in Seq1 and 5431726–5625639 in Seq2.

Comparison of RaMA and Other Methods on Non-Repetitive Sequences

One significant limitation of UniAligner is its suitability only for repetitive sequences, as it does not align the remaining regions after rare-alignment. RaMA addresses this issue by aligning these regions as well. We generated simulated datasets of 1,000,000 bp with sequence similarities ranging from 75% to 99% using INDELible [8] (see Supplemental Method). Alignments were performed using RaMA, UniAligner, and WFA, with affine gap penalties set for WFA (match = 0, mismatch = 2, gap open = 3, gap extension = 1). Q scores of the alignment results are shown in Fig S10. When sequence similarity is low, WFA provides the highest alignment quality, followed by RaMA, with UniAligner performing the worst. As similarity exceeds 83%, RaMA's alignment quality becomes consistent with or surpasses WFA. UniAligner consistently underperforms across all similarity levels. Notably, at 90% similarity, the Q scores of all methods approach 1, likely due to favorable sequence characteristics. These findings suggest RaMA excels in aligning non-repetitive sequences with moderate to high similarity, while UniAligner does not.

We investigated UniAligner's poor alignment quality on non-repetitive sequences by selecting sequences with 80% similarity, where the quality difference was most pronounced. We analyzed the gaps and mismatches in the alignment results. As shown in Fig S11, WFA's gaps and mismatches closely match the true alignment, whereas UniAligner shows significantly more gaps and fewer mismatches. This discrepancy arises from UniAligner's alignment algorithm: it aligns indel-runs between anchor points, resulting in mismatches if the lengths are equal, and producing deletion-runs and insertion-runs, leading to many gaps if the lengths are unequal.

We further investigated the behavior of sparse match anchors across different sequence similarity levels. The simulated sequences had a length of 1 million base pairs. The relationship between sequence similarity and both the number of sparse match anchors, as well as the total length of these anchors, is presented in Fig S12 and Fig S13. Interestingly, as shown in Fig S12 and Fig S13, for non-repetitive sequences, the number of anchors and the total length of anchors exhibit opposite trends. We define anchor coverage as the ratio of the total anchor length to the total sequence length. The variation in anchor coverage with respect to sequence similarity is shown in Fig S14. We also performed alignments for each chromosome's centromere using RaMA, with CHM13 as the reference and CHM1 as the query. The number of anchors, total anchor length, and anchor coverage are summarized in Table S7. Across the centromeres of 23 chromosomes, the average coverage is 32%, with a maximum of 76% on chromosome 19. In contrast, the minimum coverage for non-repetitive sequences at 70% similarity is 88%. This indicates that rare match anchors are significantly fewer in tandem repeat sequences compared to non-repetitive sequences.

Comparison of RaMA and Other Methods on Hybrid Sequences

In chromosomes, extra tandem repeat sequences often appear interspersed with non-tandem repeat sequences. WFA enables RaMA to handle long sequence alignments effectively. Therefore, in this section, we explore the performance of RaMA and UniAligner on hybrid sequences. We used RaMA and UniAligner to align hybrid sequences, and their alignment paths are shown in Fig S15. As shown in the Fig S15, both methods demonstrate strong boundary distinction capabilities for the regions. For a given region pair, the match bases refer to the number of bases in the query region that align to the reference in the alignment result. Coverage is defined as the ratio of match bases to the length of the reference. We present the alignment coverage of RaMA and UniAligner for the five regions in the Table S8. The results show that RaMA achieves slightly higher alignment quality than UniAligner across all five regions. This experiment demonstrates RaMA's potential for accurately aligning hybrid sequences. In conclusion, although this experiment does not fully showcase RaMA's performance on hybrid sequences, it offers valuable insights into RaMA's potential for accurate alignment of these sequences.

Linear Range Minimum Query Strategy

To accelerate the range minimum query on the LCP array, we employed a linear range minimum query algorithm based on block sparse table. In this section, we provide a detailed explanation of this algorithm. For range minimum queries, the sparse table [9] is a widely used algorithm, with a time complexity of *O(n log n)* for its construction. However, this can be relatively slow compared to the *O(n)* complexity of directly building an enhanced suffix array [10]. Therefore, we aim to improve the sparse table to achieve *O(n)* construction time as well. A simple and straightforward approach is to divide the sequence into blocks of length *log n*, and use a sparse table to manage the minimum value of each block. This reduces the construction time of the sparse table to *O(n)*. Of course, this alone is insufficient, as it only allows queries at the block level. To address this, additional auxiliary data structures are needed to fully refine this approach.

For query crossing multiple blocks, we utilize two auxiliary arrays: the prefix minimum (Pre) and suffix minimum (Sub). The prefix minimum for each element stores the minimum value from the start of the block up to that element, while the suffix minimum stores the minimum from the element to the end of the block. These allow us to handle any intra-block query efficiently by simply looking up the precomputed values in $O(1)$ time. So when the query range spans across multiple blocks, as shown in Fig S16, the strategy involves dividing the query into three parts: the portion within the starting block, handled by the suffix minimum; the portion within the ending block, handled by the prefix minimum; and the portion spanning entire blocks, managed by the sparse table. The result is the minimum of these three values, ensuring that cross-block queries are answered efficiently in constant time after preprocessing.

To handle query within one block efficiently, we use a monotonic stack combined with state compression, as shown in the example from Fig S16. In this example, we are working with the array $A = [3, 2, 5, 4, 7]$ and performing a query to find the minimum value in the range $[l2 = 1, r2 = 2]$ within the same block. We use a monotonic stack to track the minimum values. we precompute an F array, where $F[i]$ stores the minimum value from $A[0]$ to $A[i]$ using a monotonic stack. As we traverse the block, starting with $A[1] = 2$, it is pushed onto the stack because it is smaller than $A[0] = 3$, which was previously on top. Larger elements, like $A[0] = 3$, are popped out. The remaining elements are all larger than 2, so they are not pushed onto the stack. To further optimize, we use state compression by encoding the stack's status into a bitmask. This means we can store the entire monotonic stack using just a 64-bit integer. Naturally, this also implies that the length of each block cannot exceed 64, and the length of array A cannot exceed 264. For example, after processing $A[0,2]$, the bitmask $F[2] = 0b000010$ indicates that only $A[1] = 2$ remains in the stack at this point. To efficiently perform the query for $[1,2]$, we right-shift the bitmask by 1 bit (since l_2=1), resulting in $0b000001$. We then find the position of the first '1' in the shifted bitmask, which corresponds to the index of the minimum value within the queried range. In this case, the first '1' appears at position 0, meaning the minimum value is at index 1+0=1. Thus, the minimum value in the range [1,2] is $A[1] = 2$. This process allows us to find the minimum value in constant time $O(1)$, leveraging both the monotonic stack and efficient bitwise operations.

In the construction process of the algorithm, four key arrays are involved: the prefix minimum array Pre, the suffix minimum array Sub, the sparse table S, and the compressed state array F. Both Pre and Sub store the prefix and suffix minimum values within each block, and they are computed in $O(n)$ time through a single linear pass over the array, with each requiring $O(n)$ space. The sparse table S is built for efficient cross-block queries, which takes $O(n)$ time for preprocessing as it operates on the block-level minimums and stores results in $O(n)$ space. The compressed state array F represents the status of the monotonic stack within each block and allows intra-block queries to be resolved in constant time using bitwise operations. Constructing F involves a linear scan, giving it a time complexity of $O(n)$ and requiring only $O(n)$ space. Therefore, considering the construction of all these arrays involves only linear operations relative to the size of the input, the total time complexity of the algorithm is $O(n)$, and the space complexity remains $O(n)$ as well. In the next section, "Construction of the Enhanced Suffix Array for Subsequences," we experimentally demonstrate that the constant factor in the construction process of this algorithm is smaller than that of the suffix array construction algorithm.

Construction of the Enhanced Suffix Array for Subsequences

RaMA identifies rare matches through a recursive process. Initially, an enhanced suffix array is constructed for the two input sequences, and the LCP array is used to efficiently locate rare matches. These rare matches are then used to partition the sequences into subsequences, which undergo the same process iteratively until no further rare matches are found. Thus, a key challenge in algorithm optimization lies in how to quickly construct the enhanced suffix array for the subsequences. The most straightforward approach is to directly construct the enhanced suffix array for each subsequence, as done in UniAligner [3]. In practice, with the suffix array of the original sequence, we can efficiently construct the suffix array for subsequences in a single pass using the Inverse Suffix Array (ISA). To construct the suffix array for a subsequence using the Inverse Suffix Array

(ISA), we leverage the fact that the ISA maps the position of a suffix in the original sequence to its rank in the suffix array. For a subsequence, we can use the ISA of the original sequence to quickly determine the rank of each suffix within the subsequence. Specifically, for each suffix in the subsequence, we find its rank in the original sequence using the ISA. By sorting these ranks, we effectively build the suffix array for the subsequence. This approach is efficient because it avoids the need to recompute the suffix array from scratch, instead utilizing the existing structure of the original sequence's ISA.

To construct the LCP array, each LCP value requires a range minimum query (RMQ) on the original LCP sequence. Notably, both the construction of the suffix array using the ISA and the RMQ-based LCP array construction can be parallelized using multithreading to enhance performance. We implemented both the sparse table [9] and the block sparse table and compared their construction times with that of the enhanced suffix array [10]. We performed tests on data sizes ranging from 1 million to 20 million, measuring the construction time ten times for each size and taking the average. The results are presented in the Fig S17, it can be seen that the construction time of the enhanced suffix array is slightly faster than that of the sparse table, while the block sparse table is significantly faster than the enhanced suffix array. For instance, when the data size reaches 20 million, the construction time for the sparse table is 10.5 seconds, the enhanced suffix array takes 9.7 seconds, and the block sparse table finishes in just 2.16 seconds. This demonstrates that the constant factor for the block sparse table is much smaller than that of the enhanced suffix array.

Using the same settings, we also compared the query times of the sparse table and block sparse table across different data sizes with the construction time of the enhanced suffix array. The results are shown in the Fig S18. For example, when the data size is 20 million, the query times for the sparse table and block sparse table are similar, at 5.34 seconds and 5.15 seconds, respectively, both significantly faster than the 9.74 seconds required to construct the enhanced suffix array for a sequence of the same length. A key advantage of querying, compared to the construction of the enhanced suffix array, is that it can be parallelized, whereas the latter cannot. Using a data size of 10 million as an example, the Fig S19 shows how the total query time changes as the number of threads increases. It clearly demonstrates that multithreading significantly optimizes the querying process.

In RaMA, the process recursively searches for rare matches to split the sequences, followed by constructing enhanced suffix arrays for the subsequences. However, the length of the subsequences requiring an enhanced suffix array is unpredictable, as the same segment may be processed multiple times. To analyze this, we used RaMA to compare the centromeres of different chromosomes from CHM13 and CHM1, recording the ratio of subsequence length to the original sequence length. The results, shown in the Fig S20, indicate that the average ratio is 1.7. We used the 1.7 ratio to compare the time required by three different strategies for constructing the enhanced suffix array for subsequences. For an input size of N, all three strategies involve constructing the enhanced suffix array for a sequence of length N. In the subsequent steps, an enhanced suffix array needs to be constructed for a subsequence of length $1.7N$. The three strategies are as follows: (1) Strategy 1 directly constructs the suffix array for the subsequence. (2) Strategy 2 constructs the block sparse table for the input data of size N, performs $1.7N$ queries to build the LCP array, and uses the ISA to quickly obtain the suffix array for the subsequence. (3) Strategy 3 is the same as Strategy 2 but utilizes 16 threads to parallelize the queries. The results are shown in the Fig S21. As observed, the single-threaded block sparse table strategy is slightly faster than the direct construction of the enhanced suffix array, while the parallelized block sparse table strategy is significantly faster than both. The speed of strategy 3 is approximately twice that of strategy 1. This indicates that the block sparse table optimization effectively utilizes modern processors to accelerate the search for rare matches.

Supplemental Figures

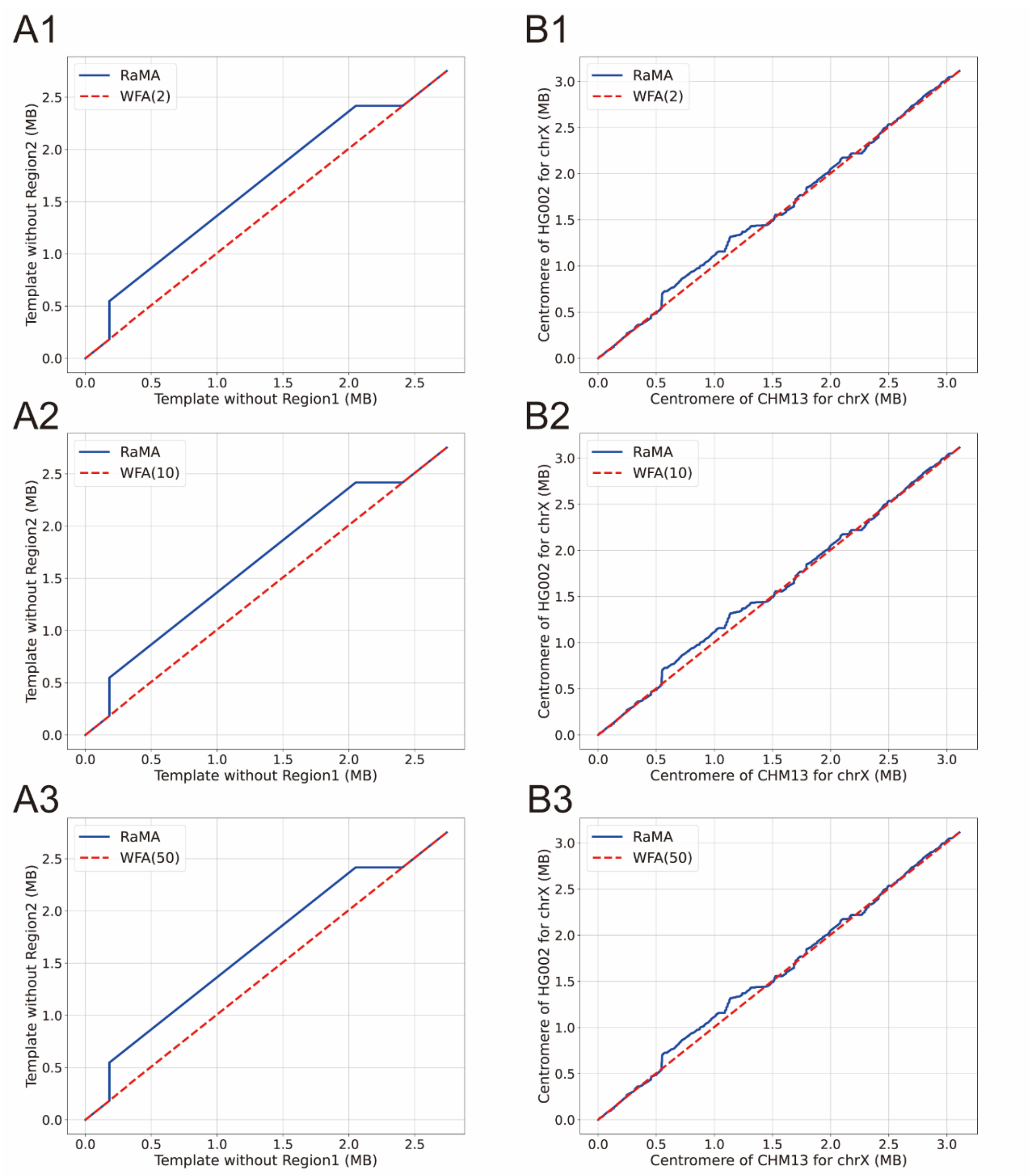


Fig S1 Comparison of alignment paths between RaMA and other methods on real and simulated centromere sequences. Series A uses a template with region1 removed as the reference and a template with region2 removed as the query. Series B shows the X chromosome centromere of CHM13 as the reference and HG002 as the query. Labels 1, 2, and 3 correspond to RaMA compared with different parameters for WFA. The WFA penalty settings are configured as follows: the match penalty is set to 0, mismatch penalty to 4, short insertion opening penalty to 6, and long insertion opening penalty to 12. The extension penalty for long insertions is 1, while the extension penalties for short insertions labeled as 1, 2, and 3 are 2, 10, and 50, respectively.

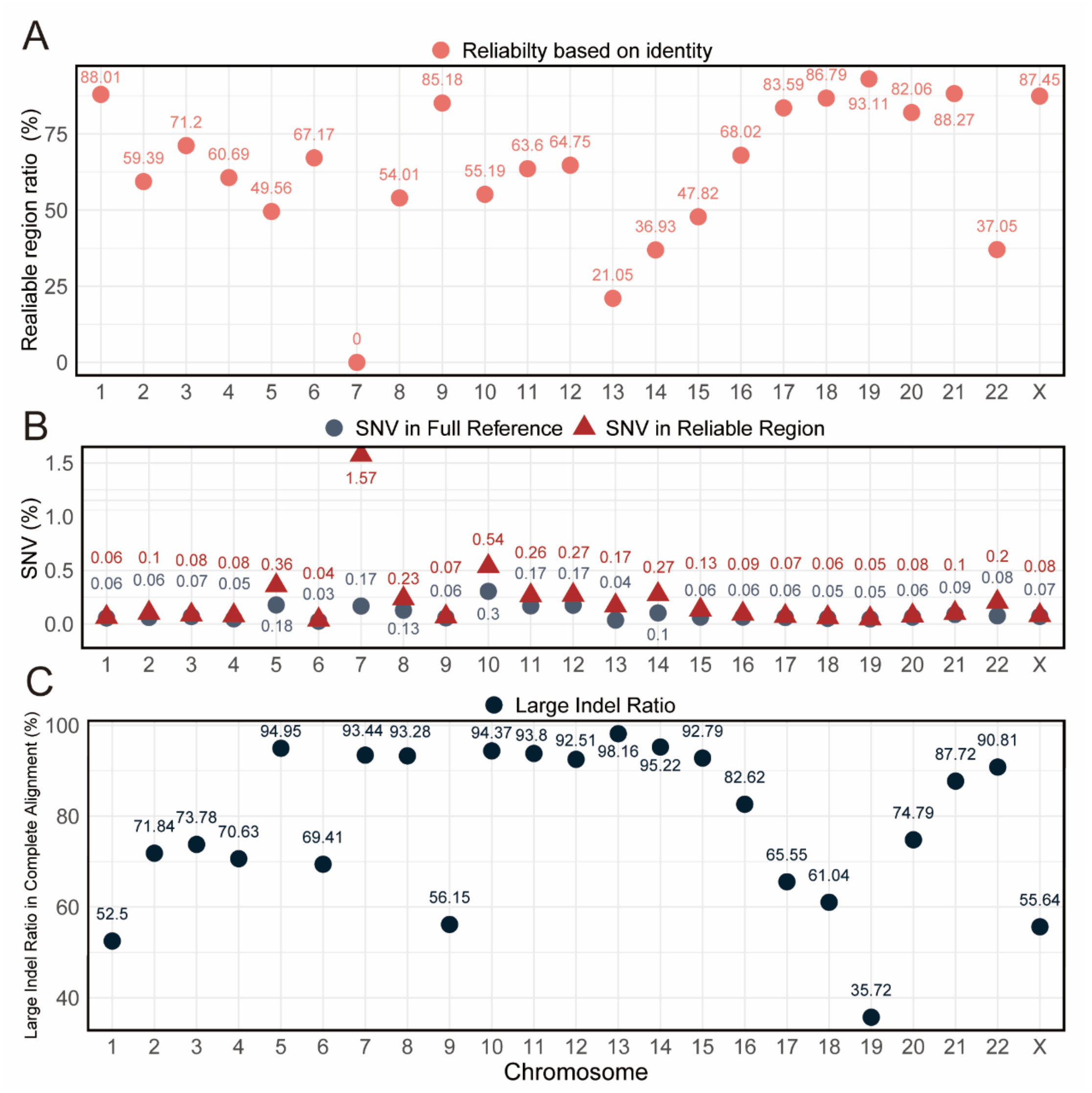


Fig S2 Statistical analysis of centromere alignment results between CHM13 and CHM1 using UniAligner. (A) Proportion of identity-based reliable regions across different chromosomes relative to the reference sequence length. (B) Comparison of the single nucleotide variant (SNV) rates between the entire reference sequence region and identity-based reliable regions. (C) Proportion of large indels across the complete reference sequence region.

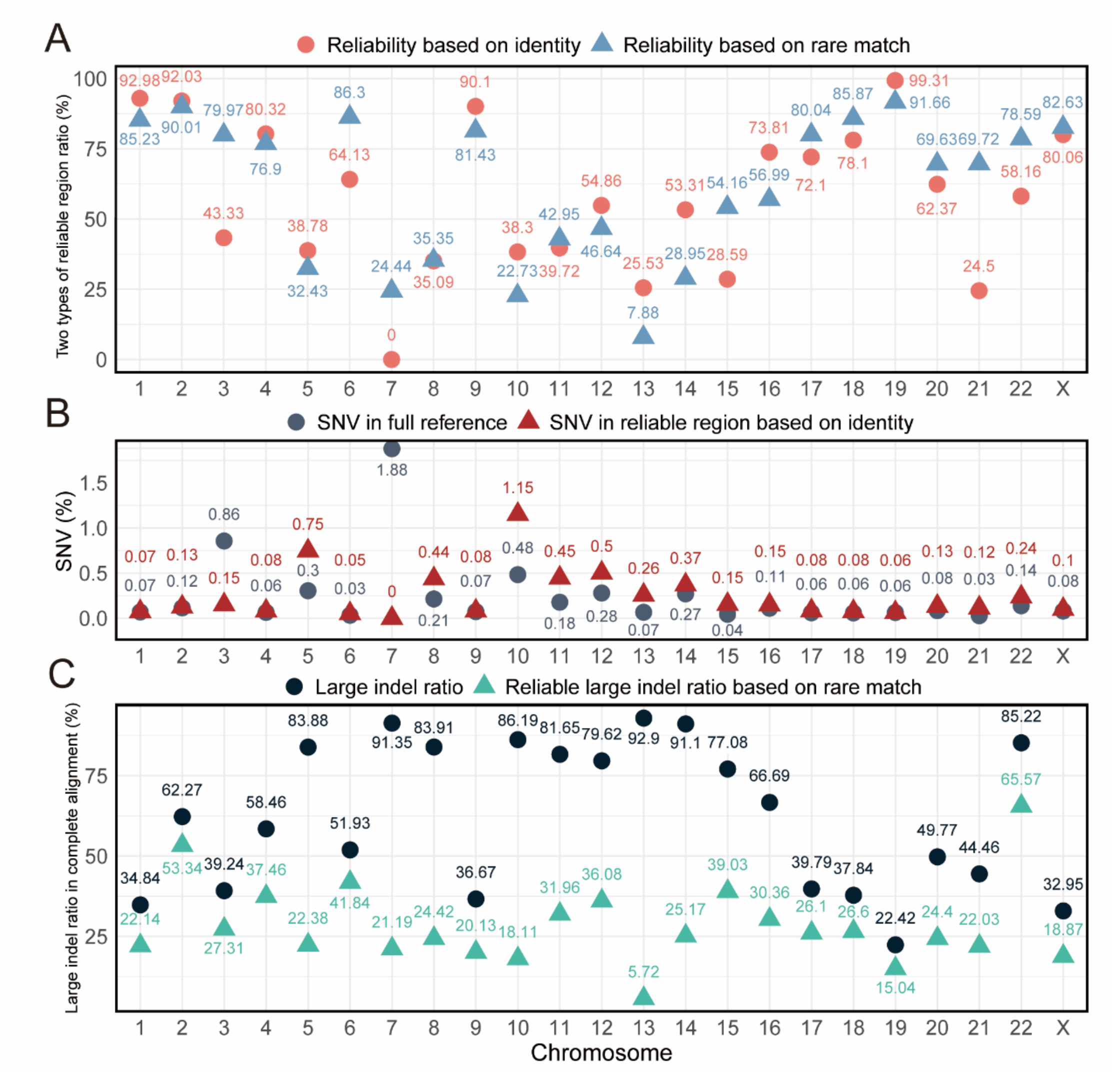


Fig S3 Statistical analysis of centromere alignment results between CHM1 and CHM13 using RaMA. (A) Proportion of two types of reliable regions, based on identity and rare matches, across different chromosomes relative to the reference sequence length. (B) Comparison of the single nucleotide variant (SNV) rates between the entire reference sequence region and identity-based reliable regions. (C) Proportion of large indels across the complete reference sequence region versus within rare-match-based reliable regions.

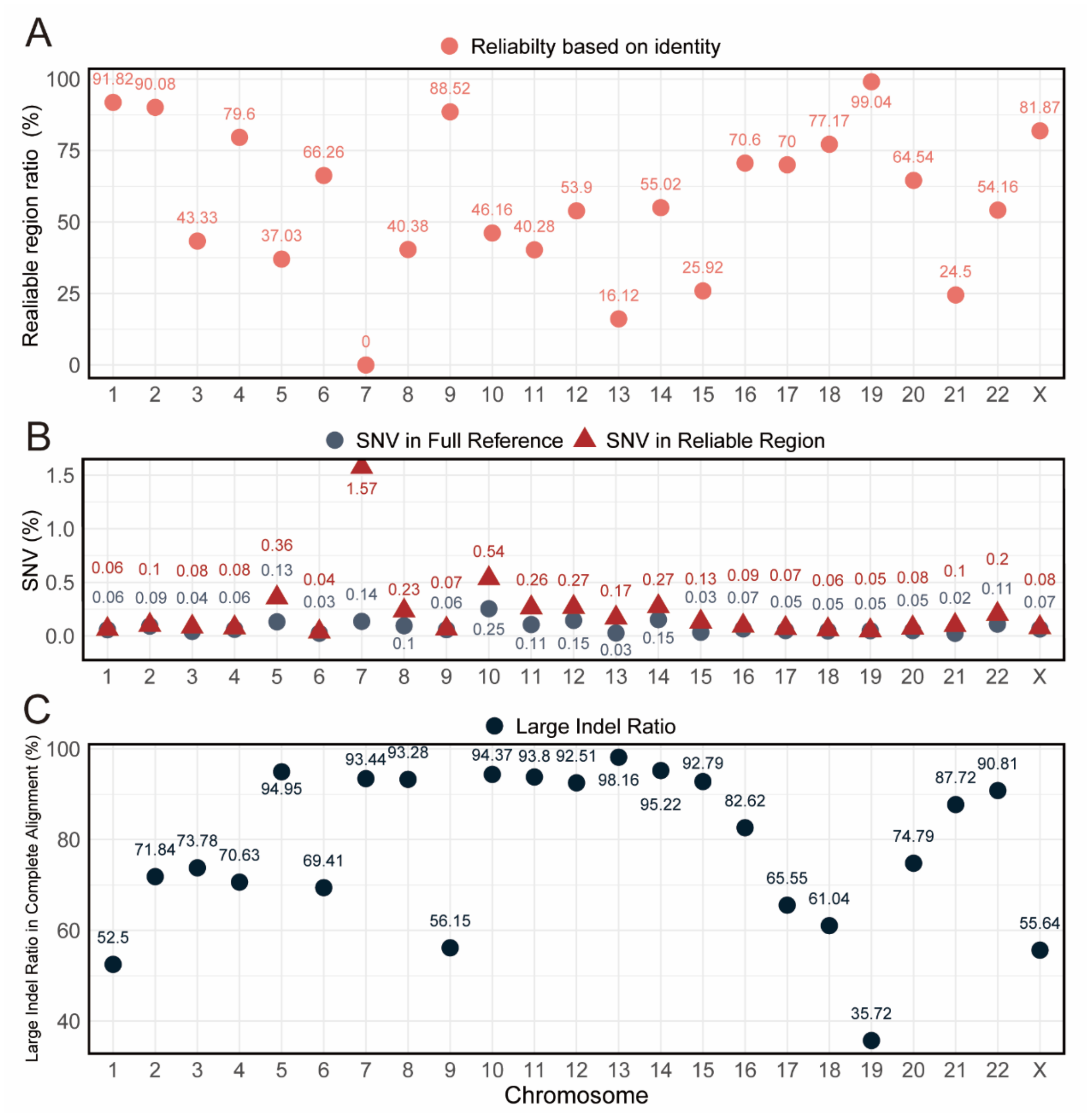


Fig S4 Statistical analysis of centromere alignment results between CHM1 and CHM13 using UniAligner. (A) Proportion of identity-based reliable regions across different chromosomes relative to the reference sequence length. (B) Comparison of the single nucleotide variant (SNV) rates between the entire reference sequence region and identity-based reliable regions. (C) Proportion of large indels across the complete reference sequence region.

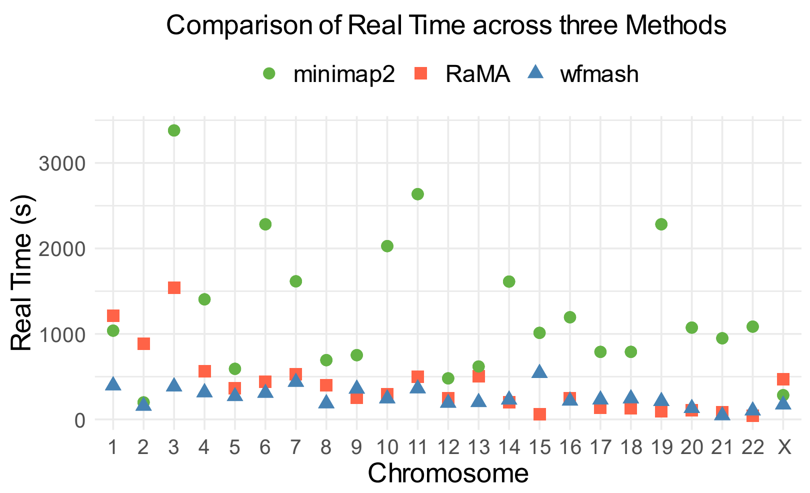


Fig S5 Comparison of runtime across three sequence alignment methods on different chromosomes. This figure shows the real runtime of three pairwise sequence alignment methods—RaMA, minimap2, and wfmash with 32 threads —on different chromosomes from CHM13 and CHM1 genomes. Each method is represented by different colors and shapes: green circles for RaMA, red squares for minimap2, and blue triangles for wfmash. The y-axis represents the runtime in seconds, while the x-axis shows the chromosome numbers. It can be observed that RaMA exhibits higher runtimes on several chromosomes, especially on larger ones like chr1 and chr3, while minimap2 and wfmash have relatively shorter runtimes, with wfmash being particularly fast on smaller chromosomes.

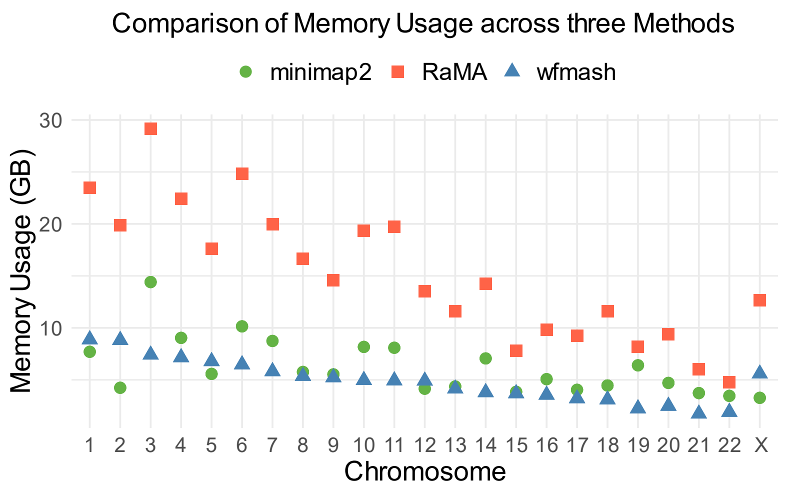


Fig S6 Comparison of memory usage across three sequence alignment methods on different chromosomes. This figure shows the memory usage of three pairwise sequence alignment methods with 32 threads—RaMA, minimap2, and wfmash—on different chromosomes from CHM13 and CHM1 genomes. Each method is represented by different colors and shapes: green circles for RaMA, red squares for minimap2, and blue triangles for wfmash. The y-axis represents the memory usage in gigabytes (GB), while the x-axis shows the chromosome numbers. It can be observed that RaMA generally consumes more memory on most chromosomes, while minimap2 and wfmash show lower memory usage, with wfmash being the most memory-efficient method across most chromosomes.

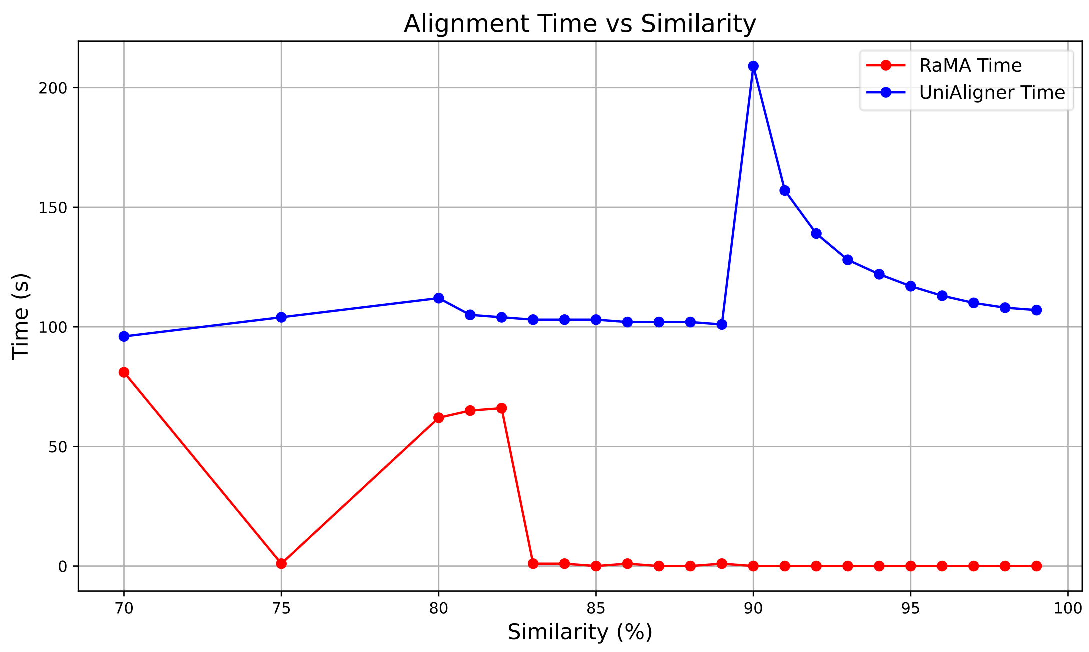


Fig S7 Alignment Time of RaMA and UniAligner on Datasets with Different Similarities

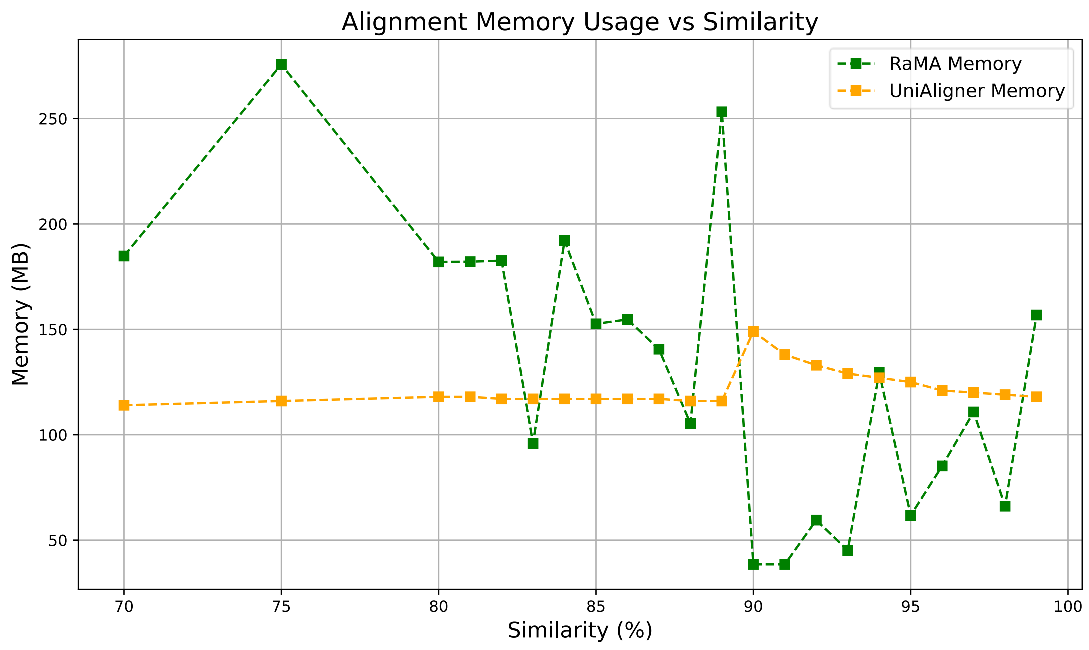


Fig S8 Alignment Max Memory of RaMA and UniAligner on Datasets with Different Similarities

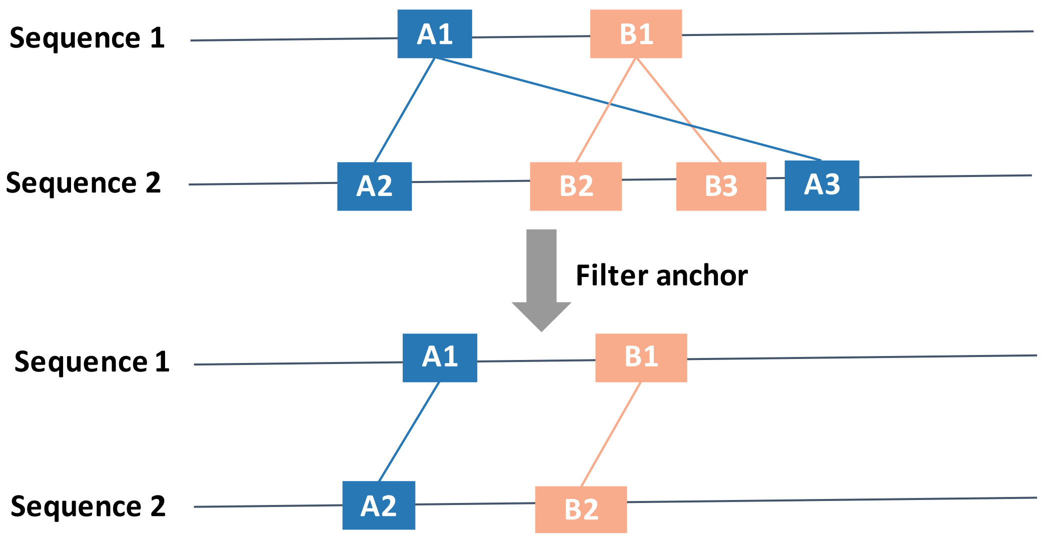


Fig S9 Anchor Filtering: Initially, the sequence contains two types of rare matches: A and B, each with three matches, resulting in four pairs of anchors: A1A2, A1A3, B1B2 and B1B3. Following dynamic programming, only A1A2 and B1B2 are saved. A1A3 were removed because they did not meet the requirement for colinearity, and B1B3 were removed due to their excessive gap cost.

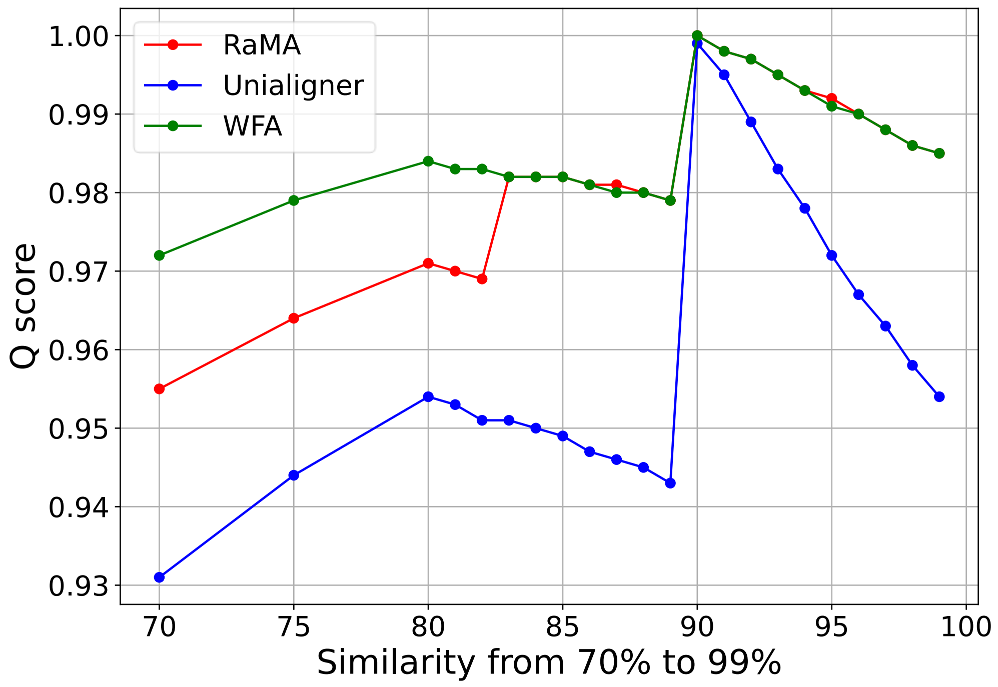


Fig S10 Q-score comparison of alignment results for simulated non-repetitive sequences with different similarities using RaMA, UniAligner and WFA.

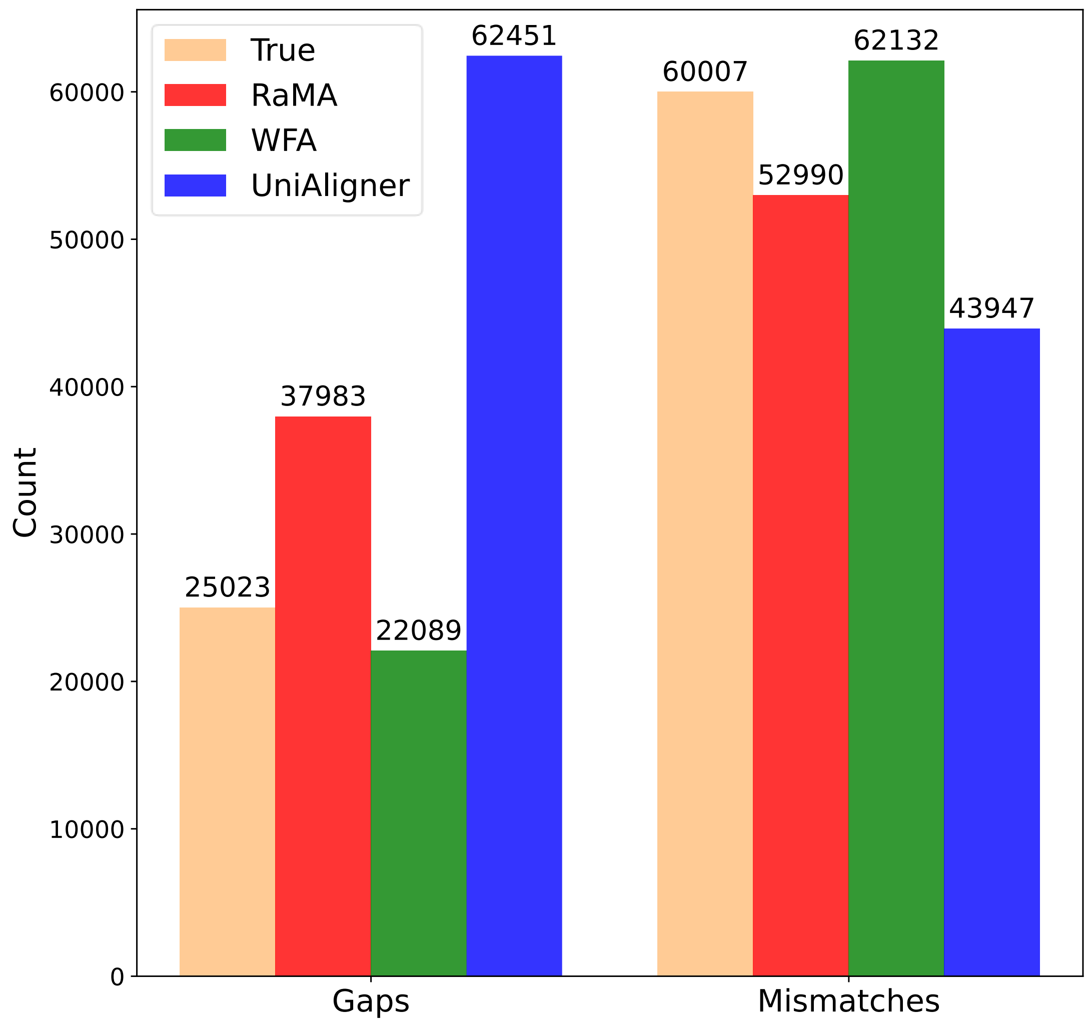


Fig S11 Statistical analysis of bases count for gaps and mismatches in alignment results for sequences with 80% similarity using three methods.

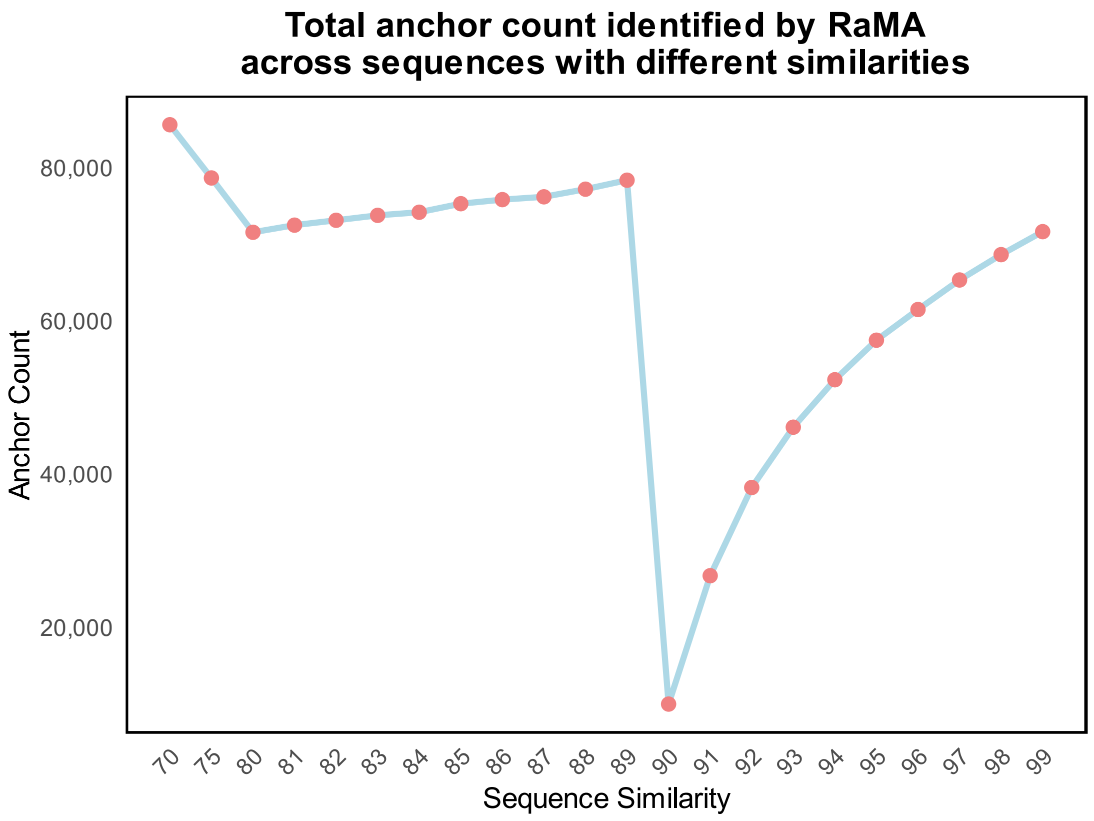


Fig S12 Number of rare match anchors identified by RaMA for non-repetitive sequences at different similarity levels

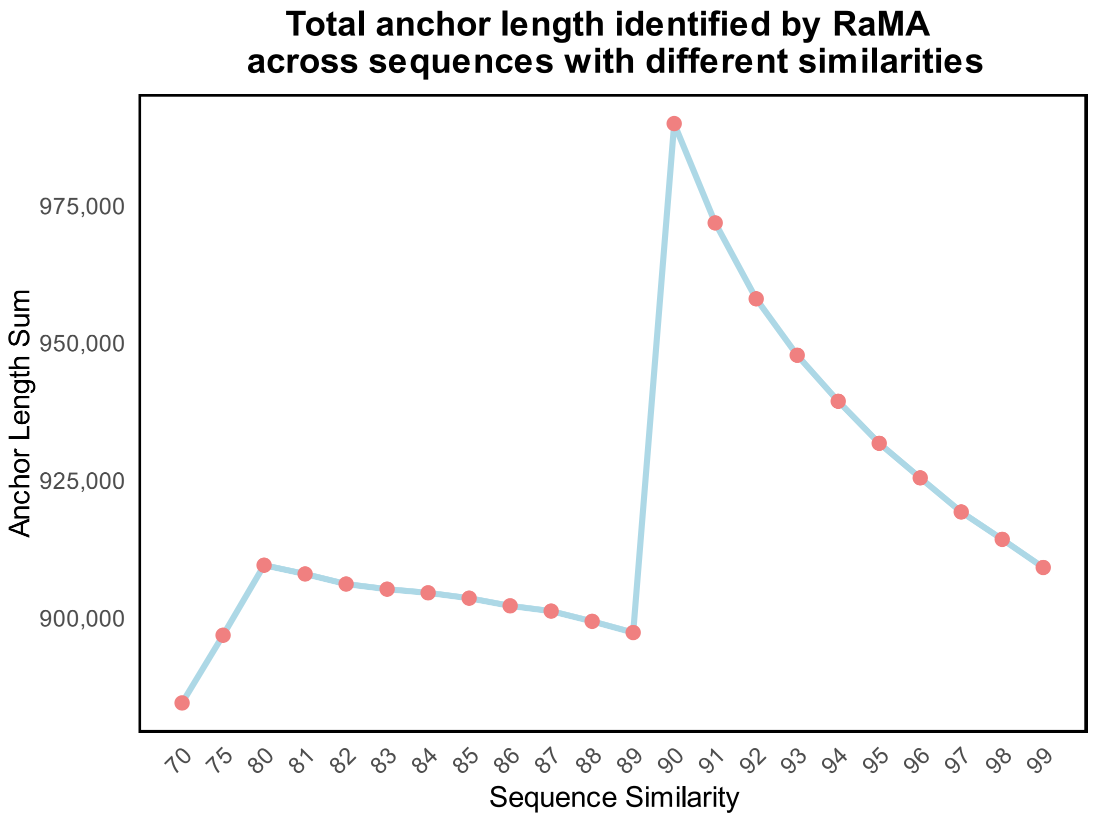


Fig S13 Total length of rare match anchors identified by RaMA in 1-million-length non-repetitive sequences at different similarity levels.

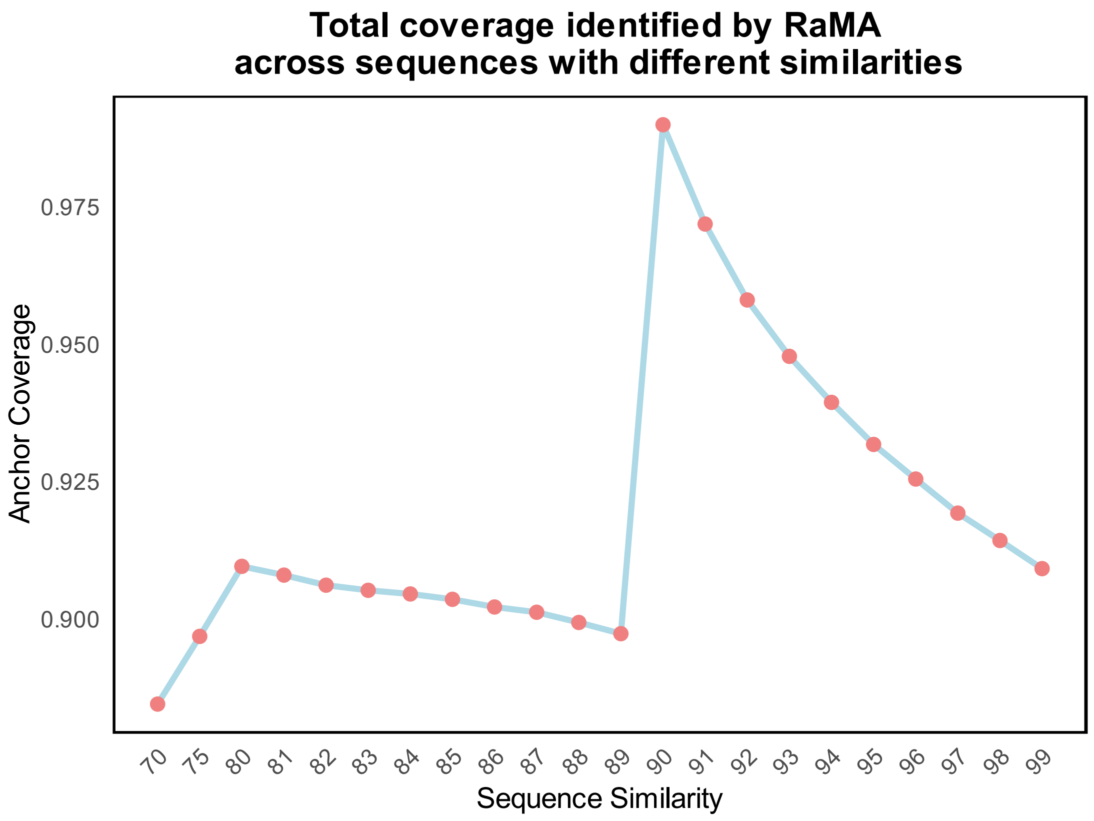


Fig S14 Coverage of rare match anchors identified by RaMA in 1-million-length non-repetitive sequences at different similarity levels. Coverage is defined as the ratio of the total length of rare match anchors to the total sequence length.

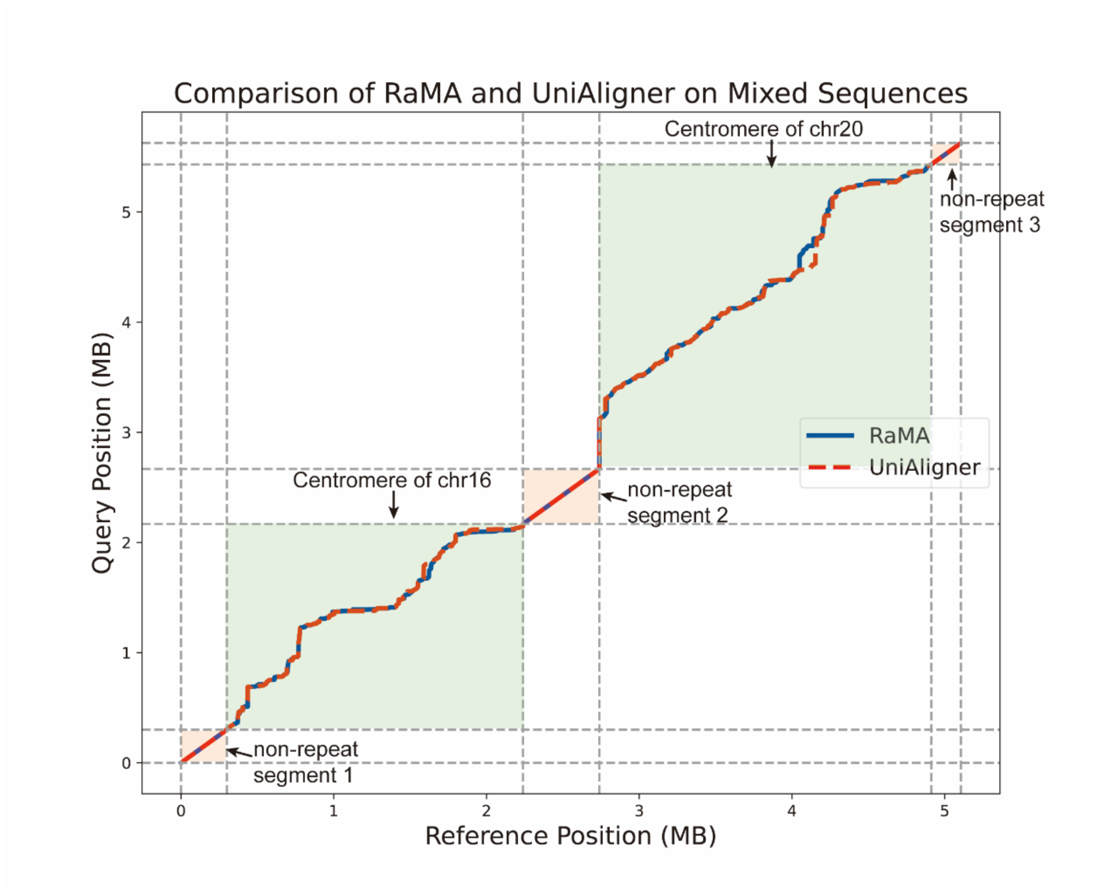
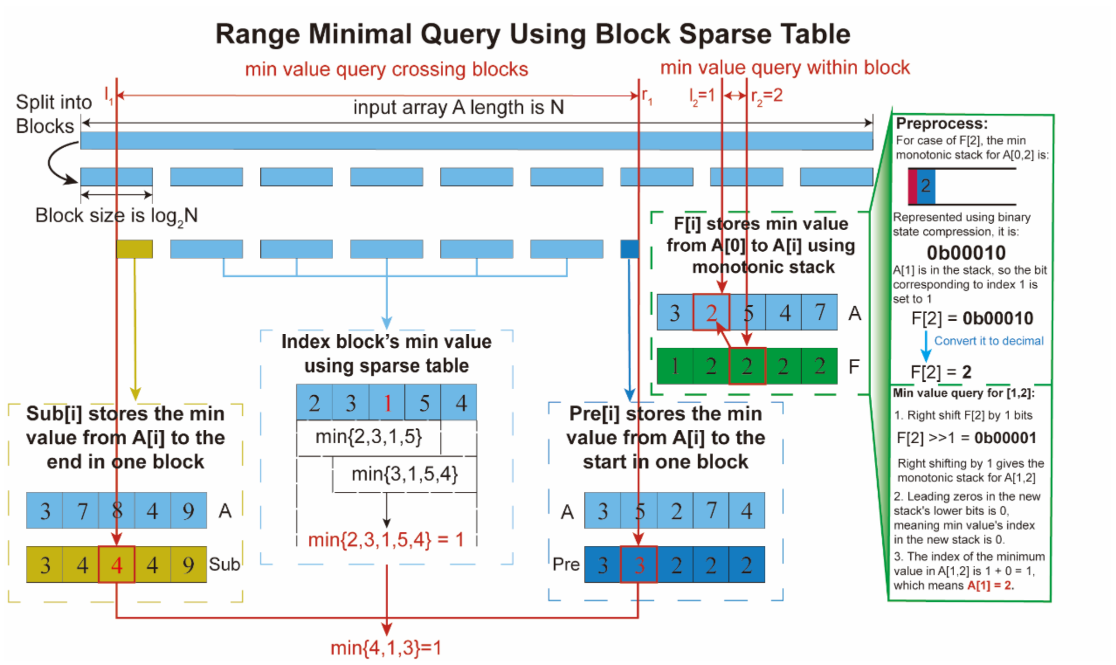


Fig S15 Comparison of alignment paths between RaMA and UniAligner on hybrid sequences composed of tandem repeat and non-tandem repeat regions. We simulated two non-repetitive sequences of length 1,000,000 with 95% similarity, then inserted the centromeres from chromosomes 16 and 20 of CHM13 and CHM1 into them. This figure demonstrates that both RaMA and UniAligner have the ability to distinguish the boundaries of different regions.


Range Minimal Query Using Block Sparse Table
min value query crossing blocks
min value query within block
Split into Blocks
l_1
r_1
$l_2=1$
$r_2=2$
input array A length is N
Block size is $\log_2 N$
F[i] stores min value from A[0] to A[i] using monotonic stack
3 2 5 4 7 A
1 2 2 2 2 F
Index block's min value using sparse table
2 3 1 5 4
min{2,3,1,5}
min{3,1,5,4}
min{2,3,1,5,4} = 1
Sub[i] stores the min value from A[i] to the end in one block
3 7 8 4 9 A
3 4 4 4 9 Sub
Pre[i] stores the min value from A[i] to the start in one block
A 3 5 2 7 4
Pre 3 3 2 2 2
min{4,1,3}=1

Preprocess:

For case of F[2], the min monotonic stack for A[0,2] is:

Represented using binary state compression, it is:

0b00010

A[1] is in the stack, so the bit corresponding to index 1 is set to 1

F[2] = **0b00010**

Convert it to decimal

F[2] = **2**

Min value query for [1,2]:

1. Right shift F[2] by 1 bits

F[2] >>1 = **0b00001**

Right shifting by 1 gives the monotonic stack for A[1,2]

2. Leading zeros in the new stack's lower bits is 0, meaning min value's index in the new stack is 0.
3. The index of the minimum value in A[1,2] is 1 + 0 = 1, which means **A[1] = 2.**

Fig S16 Workflow of the Range Minimal Query algorithm using Block Sparse Table. The input array A[1…N] is divided into blocks of length log₂N), with each block's maximum value precomputed, and inter-block minimal values handled by sparse table. For range minimum queries across blocks, it compares the edge parts with the full blocks in between. The sparse table quickly provides the minimum for the full blocks in the middle. For the incomplete head and tail blocks, we precompute the Pre and Sub arrays. Pre[i] stores the minimum from A[0] to the block's start, with Sub following the same logic, allowing the minimum value of the incomplete blocks to be obtained with a single query. For query within block, we precompute an F array, where F[i] stores the minimum value from A[0] to A[i] using a monotonic stack. The monotonic stack is compressed into a single integer, and bitwise operations allow us to retrieve the stack for different ranges to obtain the minimum value within that range. A detailed example of an intra-block query can be seen in the green box on the right side of the figure.

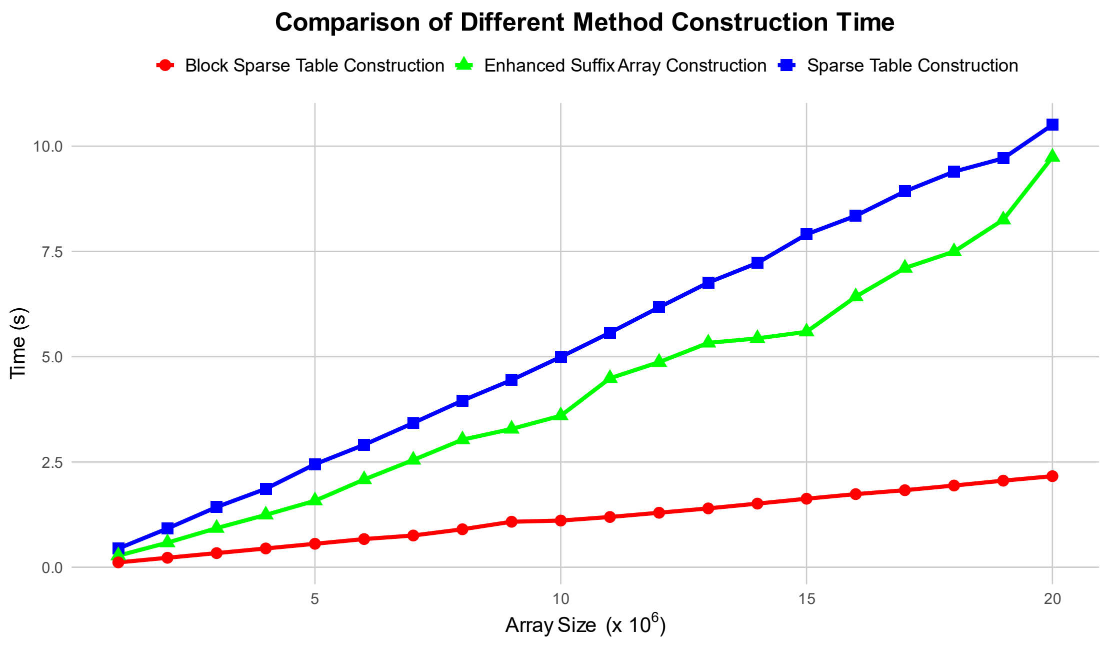


Fig S17 Comparison of construction time for Block sparse Table, Regular Sparse Table, and Enhanced Suffix Array across different array sizes, averaged over ten experiments.

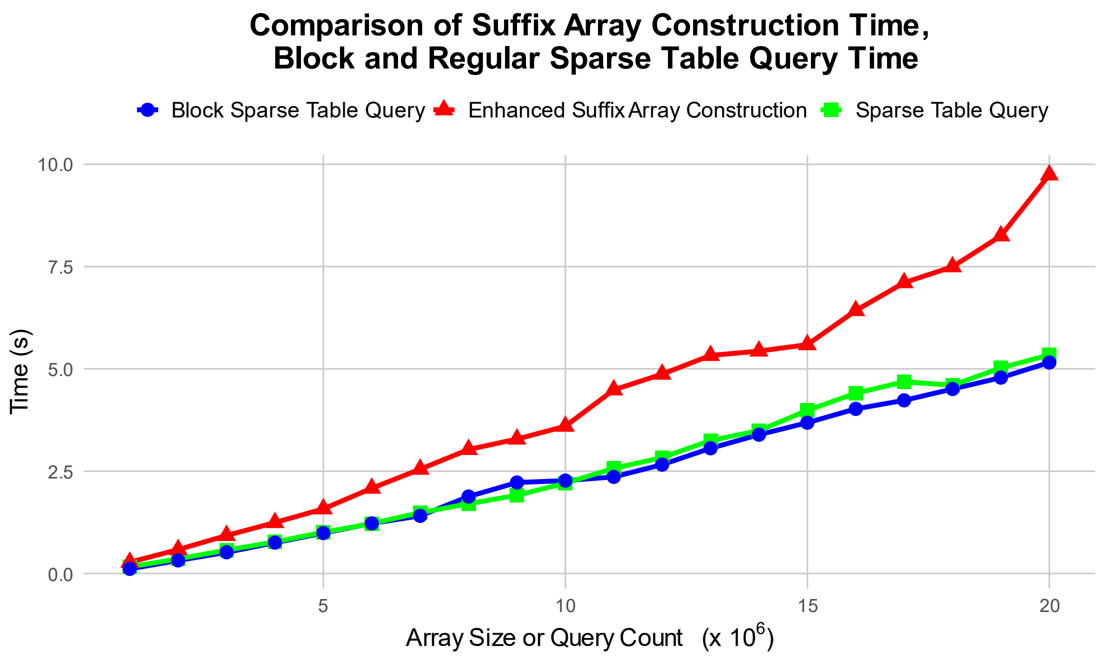


Fig S18 Comparison of construction time for Suffix Array and query times for Regular Sparse Table and Block sparse Table across different array sizes or query counts, averaged over ten experiments.

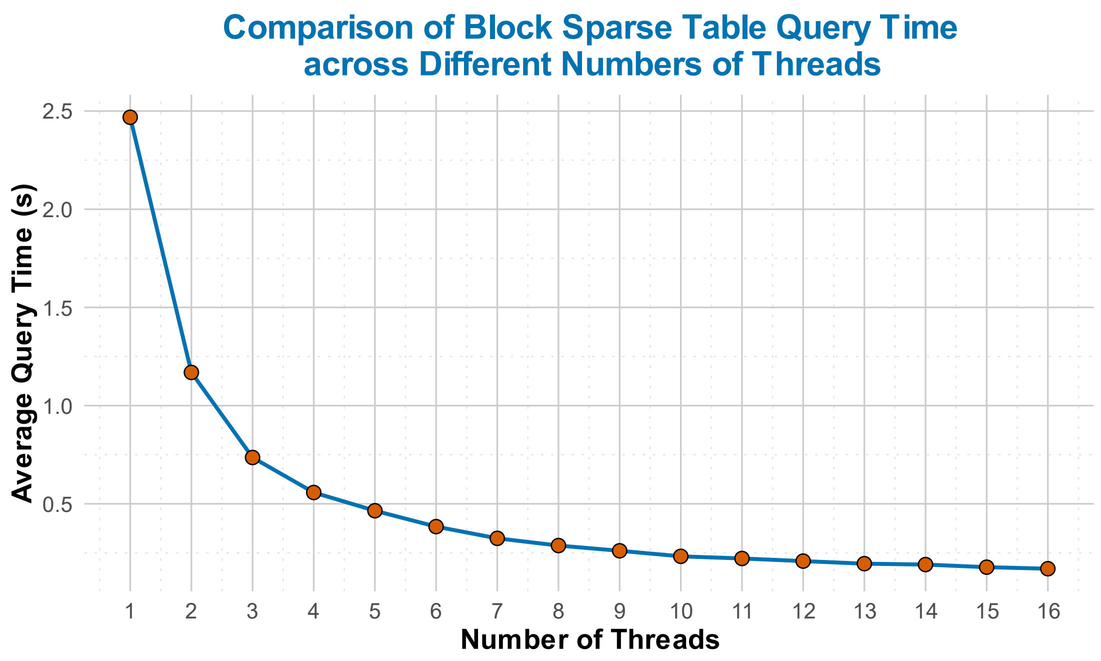


Fig S19 Variation in parallel query time for the Block sparse Table with respect to the number of threads, for N=1,000,000 queries, averaged over ten experiments.

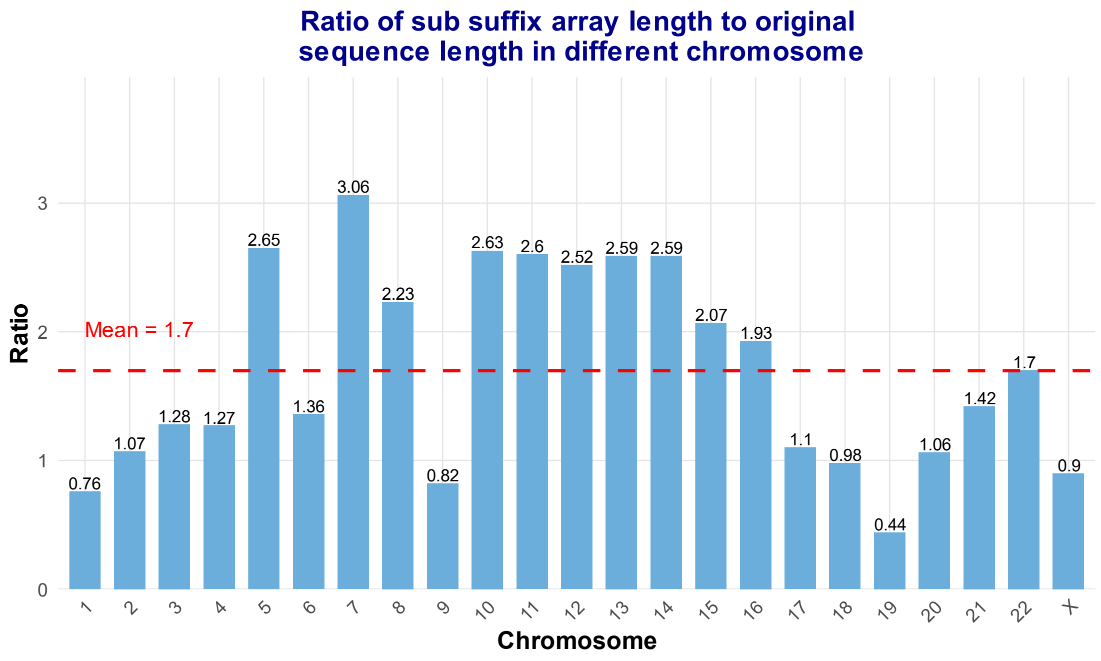


Fig S20 Ratio of sub suffix array length to the original sequence length across different chromosomes, based on the alignment results of CHM13 and CHM1 centromeres using RaMA.

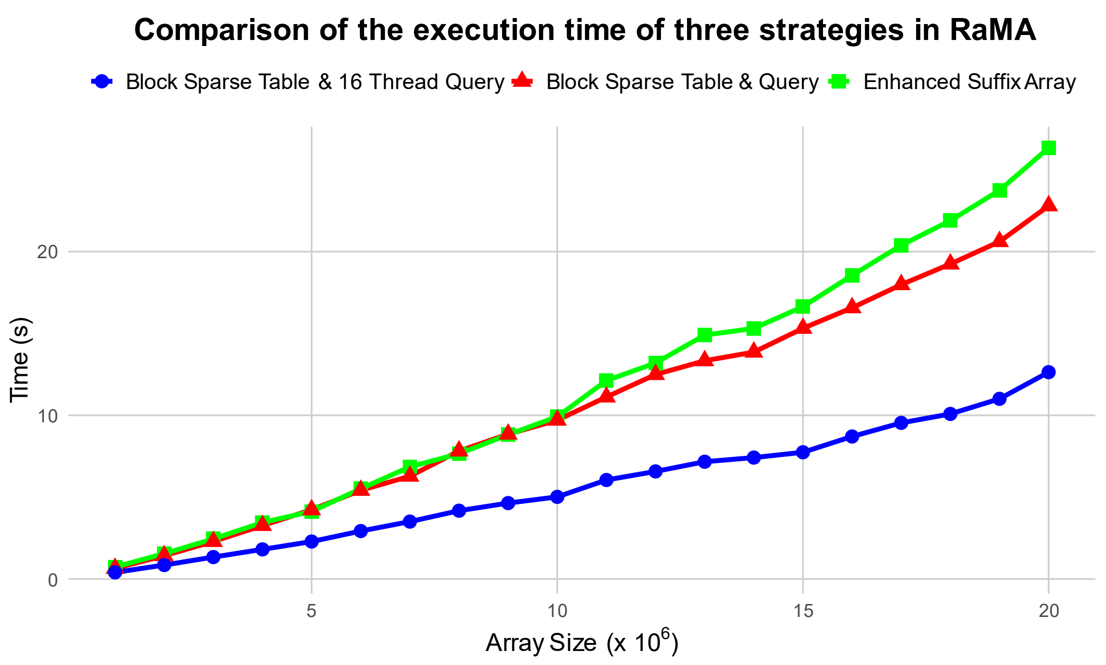


Fig S21 Time comparison of three RaMA strategies across different array sizes averaged over ten experiments. For input data size *N*, all three strategies require constructing an enhanced suffix array for a sequence of length *N*. For subsequent steps, an enhanced suffix array needs to be constructed for a sub-sequence of length 1.7*N*. The three strategies differ in their approach: (1) Strategy 1 directly constructs the suffix array for the sub-sequence. (2) Strategy 2 constructs the Block sparse Table for the input data of size *N* and performs 1.7*N* queries. (3) Strategy 3 is the same as Strategy 2, but uses 16 threads for parallel querying.

Supplemental Tables

Table S1 Indel Statistics for RaMA Alignment Results of HOR Arrays Across Chromosomes in CHM13 and CHM1

Chromosome	Total Indels	Insertions	Deletions	Short Indels	Long Indels	Short Insertions	Short Deletions	Long Insertions	Long Deletions	Total Short Indel Length
chr1	787	364	423	162	625	73	89	291	334	278
chr2	571	288	283	163	408	82	81	206	202	289
chr3	8802	4630	4172	6827	1975	3592	3235	1038	937	13173
chr4	648	286	362	115	533	55	60	231	302	179
chr5	2593	1423	1170	899	1694	477	422	946	748	1560
chr6	324	175	149	62	262	33	29	142	120	112

chr7	33529	17035	16494	16984	16545	8364	8620	8671	7874	38302
chr8	1274	684	590	505	769	263	242	421	348	982
chr9	405	199	206	75	330	36	39	163	167	124
chr10	1955	1119	836	831	1124	477	354	642	482	1502
chr11	2421	1377	1044	494	1927	237	257	1140	787	785
chr12	2193	1095	1098	790	1403	424	366	671	732	1403
chr13	347	177	170	73	274	38	35	139	135	131
chr14	598	287	311	130	468	69	61	218	250	241
chr15	256	168	88	47	209	27	20	141	68	75
chr16	533	274	259	113	420	67	46	207	213	221
chr17	594	309	285	78	516	37	41	272	244	119
chr18	934	486	448	185	749	93	92	393	356	302
chr19	635	301	334	109	526	53	56	248	278	169
chr20	373	198	175	65	308	35	30	163	145	106
chr21	103	45	58	16	87	7	9	38	49	29
chr22	678	292	386	184	494	97	87	195	299	324
chrX	530	314	216	124	406	65	59	249	157	195

Table S2 Indel Statistics for RaMA Alignment Results of HOR Arrays Across Chromosomes in CHM13 and CHM1

Chromosome	Total Indels	Insertions	Deletions	Short Indels	Long Indels	Short Insertions	Short Deletions	Long Insertions	Long Deletions	Total Short Indel Length
chr1	1154	545	609	344	810	167	177	378	432	613
chr2	745	371	374	278	467	146	132	225	242	446
chr3	11208	5770	5438	5996	5212	3018	2978	2752	2460	12327
chr4	1108	536	572	359	749	192	167	344	405	660
chr5	4819	2487	2332	2409	2410	1164	1245	1323	1087	4359
chr6	481	234	247	147	334	76	71	158	176	280

chr7	113807	58563	55244	64723	49084	31270	33453	27293	21791	135311
chr8	2665	1332	1333	1174	1491	589	585	743	748	2250
chr9	582	281	301	195	387	92	103	189	198	321
chr10	7027	3658	3369	3712	3315	1964	1748	1694	1621	6839
chr11	5206	2793	2413	1878	3328	887	991	1906	1422	3506
chr12	4179	2093	2086	1905	2274	993	912	1100	1174	3659
chr13	866	487	379	383	483	194	189	293	190	751
chr14	2272	1085	1187	1063	1209	536	527	549	660	2013
chr15	520	271	249	198	322	105	93	166	156	316
chr16	791	375	416	224	567	115	109	260	307	387
chr17	1020	525	495	300	720	157	143	368	352	489
chr18	1183	603	580	359	824	177	182	426	398	537
chr19	881	438	443	264	617	129	135	309	308	432
chr20	634	306	328	209	425	102	107	204	221	363
chr21	223	107	116	90	133	48	42	59	74	179
chr22	1190	599	591	512	678	262	250	337	341	902
chrX	879	466	413	315	564	153	162	313	251	578

Table S3 Coordinates of centromeres on each chromosome in CHM13 assembly v2.0

Chromosome	Centromere Start	Centromere End
chr1	121796048	126300487
chr2	92333543	94673023
chr3	91738002	96415026
chr4	49705154	55199795
chr5	47039134	49596625
chr6	58286706	61058390
chr7	60414372	63714499
chr8	44215832	46325080
chr9	44951775	47582595

chr10	39633793	41664589
chr11	51061948	54413484
chr12	34620838	37202490
chr13	15547593	17498291
chr14	10092112	12708411
chr15	16678794	17694466
chr16	35854528	37793352
chr17	23892419	27486939
chr18	15971633	20740248
chr19	25832447	29749519
chr20	26925852	29099655
chr21	10962853	11303831
chr22	12788180	15711065
chrX	57819763	60927195

Table S4 Coordinates of centromeres on each chromosome in CHM1 assembly v1.0

Chromosome	Centromere Start	Centromere End
chr1	69846251	74163648
chr2	2280725	3823209
chr3	90996577	98681590
chr4	49812514	54001726
chr5	29395586	32818667
chr6	58497481	61307082
chr7	58944638	62985385
chr8	41216354	44037473
chr9	45051953	47583457
chr10	39520501	41948392
chr11	50937313	56228467
chr12	34592723	37694387
chr13	6192909	8741416

chr14	5595461	7351743
chr15	6497612	8371049
chr16	35822802	37690820
chr17	23728691	28020815
chr18	16002565	21365394
chr19	25770284	29452617
chr20	26344814	29108504
chr21	5735844	6964260
chr22	6404040	8403387
chrX	9013916	12333297

Table S5 Coordinates of centromeres on each chromosome in HG002 assembly v1.0 for maternal and paternal

Chromosome	Centromere Start	Centromere End
chr1_MATERNAL	122027438	125955688
chr2_MATERNAL	92151246	94103074
chr3_MATERNAL	91302525	96202269
chr4_MATERNAL	50067869	53484024
chr5_MATERNAL	46798262	50386372
chr6_MATERNAL	58406826	63810043
chr7_MATERNAL	60365620	63823418
chr8_MATERNAL	43874099	46711122
chr9_MATERNAL	45056210	47410738
chr10_MATERNAL	39744707	42316517
chr11_MATERNAL	50999815	54520890
chr12_MATERNAL	34645712	37413891
chr13_MATERNAL	15945009	17237358
chr14_MATERNAL	16333134	18227245
chr15_MATERNAL	17565932	18803961
chr16_MATERNAL	36084593	38030775

chr17_MATERNAL	23434616	26974283
chr18_MATERNAL	15892634	19438086
chr19_MATERNAL	25955163	29401419
chr20_MATERNAL	26800597	29114249
chr21_MATERNAL	12802086	13492624
chr22_MATERNAL	15170886	17537395
chrX_MATERNAL	57866532	60979089
chr1_PATERNAL	122098079	127818069
chr2_PATERNAL	91976043	93860057
chr3_PATERNAL	91752227	96708256
chr4_PATERNAL	49905202	54056008
chr5_PATERNAL	46811597	56296672
chr6_PATERNAL	58484688	63416222
chr7_PATERNAL	60475720	62980001
chr8_PATERNAL	44141137	46832173
chr9_PATERNAL	43149766	45356409
chr10_PATERNAL	39736349	42039460
chr11_PATERNAL	50977296	53400916
chr12_PATERNAL	34645797	37414611
chr13_PATERNAL	11766683	13098132
chr14_PATERNAL	14415236	16746986
chr15_PATERNAL	14196988	15332200
chr16_PATERNAL	34883482	37233542
chr17_PATERNAL	23369094	27776826
chr18_PATERNAL	15911083	21158049
chr19_PATERNAL	25478659	29550562
chr20_PATERNAL	27138158	29974389
chr21_PATERNAL	9212650	10459921
chr22_PATERNAL	11150940	13678387

chrY_PATERNAL	10561582	10878917

Table S6 The positions and lengths of the five segments in the hybrid sequence.

Segment	Seq1 Start index	Length in Seq1	Seq2 Start index	Length in Seq2
Segment 1	0	300000	0	299983
Centromere 1	300000	1938824	299983	1868018
Segment 2	2238824	500000	2168001	500035
Centromere 2	2738824	2173803	2668036	2763690
Segment 3	4912627	193870	5431726	193913

Table S7 Statistics of total anchor length, anchor count, and coverage in RaMA alignment results for different chromosomes of CHM13 and CHM1. Coverage is defined as the ratio of the total anchor length to the total sequence length. This table summarizes the total length of anchors, the number of anchors, and the coverage across various chromosomes in CHM13 and CHM1.

Chromosome	Anchor Length Sum	Anchor Count	Coverage
chr1	2839728	3146	0.630428784
chr2	856477	1922	0.366097167
chr3	2460704	3006	0.526126015
chr4	2164858	2708	0.393994439
chr5	252468	6248	0.098717063
chr6	1231592	902	0.444347913
chr7	99035	25362	0.030009451
chr8	228398	3007	0.10828409
chr9	1611356	1897	0.612491923
chr10	91294	3811	0.044954786
chr11	364467	5687	0.108746258
chr12	269378	4928	0.104343265
chr13	41923	537	0.021491282
chr14	97974	1513	0.037447555
chr15	153407	554	0.151039903
chr16	515491	1283	0.265878182
chr17	1937038	2423	0.538886416

chr18	2823736	3304	0.592150132
chr19	2999472	2451	0.765743392
chr20	982130	1452	0.45180267
chr21	162450	244	0.476423699
chr22	378753	2431	0.129581903
chrX	1979063	2526	0.63688055

Table S8 The number of correct matches and the coverage of correct matches in the five regions of the hybrid sequence for RaMA and UniAligner. For a given region pair, the match bases refer to the number of bases in the query region that align to the reference in the alignment result. Coverage is defined as the ratio of match bases to the length of the reference.

Region Start	Region End	Region Length	RaMA Matched Bases	RaMA Coverage	UniAligner Matched Bases	UniAligner Coverage
0	299999	300000	293885	0.979617	290206	0.967353
300000	2238823	1938824	639272	0.329722	563272	0.290523
2238824	2738823	500000	482241	0.964482	473135	0.94627
2738824	4912626	2173803	1099443	0.505769	1004062	0.461892
4912627	5102181	189555	129310	0.682177	138254	0.729361

Reference

1. Li, H., *Minimap2: pairwise alignment for nucleotide sequences.* Bioinformatics, 2018. **34**(18): p. 3094-3100.
2. Marco-Sola, S., et al., *Fast gap-affine pairwise alignment using the wavefront algorithm.* Bioinformatics, 2021. **37**(4): p. 456-463.
3. Bzikadze, A.V. and P.A. Pevzner, *UniAligner: a parameter-free framework for fast sequence alignment.* Nature Methods, 2023. **20**(9): p. 1346-1354.
4. Logsdon, G.A., et al., *The variation and evolution of complete human centromeres.* Nature, 2024: p. 1-10.
5. Li, H., et al., *The sequence alignment/map format and SAMtools.* bioinformatics, 2009. **25**(16): p. 2078-2079.
6. Guarracino, A., et al. *wfmash: a pangenome-scale aligner.* 2021; Available from: https://github.com/waveygang/wfmash.
7. Kunyavskaya, O., et al., *Automated annotation of human centromeres with HORmon.* Genome Research, 2022. **32**(6): p. 1137-1151.

8. Fletcher, W. and Z. Yang, *INDELible: A Flexible Simulator of Biological Sequence Evolution.* Molecular Biology and Evolution, 2009. **26**(8): p. 1879-1888.

9. Bender, M.A. and M. Farach-Colton. *The LCA problem revisited*. in *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000 Proceedings 4*. 2000. Springer.

10. Louza, F.A., et al., *gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections.* Algorithms for Molecular Biology, 2020. **15**: p. 1-5.