The Journal of Supercomputing (2025) 81:355
https://doi.org/10.1007/s11227-024-06789-0

SeqMatcher: efficient genome sequence matching with AVX-512 extensions

Elena Espinosa[1] · Ricardo Quislant[1] · Rafael Larrosa[1,2] · Oscar Plata[1]

Accepted: 27 November 2024
© The Author(s) 2024

Abstract
The recent emergence of long-read sequencing technologies has enabled substantial improvements in accuracy and reduced computational costs. Nonetheless, pairwise sequence alignment remains a time-consuming step in common bioinformatics pipelines, becoming a bottleneck in de novo whole-genome assembly. Speeding up this step requires heuristics and the development of memory-frugal and efficient implementations. A promising candidate for all of the above is Myers' algorithm. However, the state-of-the-art implementations face scalability challenges when dealing with longer reads and large datasets. To address these challenges, we propose *SeqMatcher*, a fast and memory-frugal genomics sequence aligner. By leveraging the long registers of AVX-512, *SeqMatcher* reduces the data movement and memory footprint. In a comprehensive performance evaluation, *SeqMatcher* achieves speedups of up to 12.32x for the unbanded version and 26.70x for the banded version compared to the non-vectorized implementation, along with energy footprint reductions of up to 2.59x. It also outperforms state-of-the-art implementations by factors of up to 29.21x, 17.56x, 13.47x, 9.12x, and 8.81x compared to *Edlib*, *WFA2-lib*, *SeqAn*, *BSAlign*, and *QuickEd*, while improving energy consumption with reductions of up to 6.78x.

Keywords Approximate string matching · AVX-512 · Genome assembly · Hyyrö algorithm · Myers algorithm · SIMD

1 Introduction

Whole-genome sequencing allows us to know the complete DNA sequence of an organism's genome in a single process from small random fragments known as reads. It has fostered numerous scientific advances in several fields such as personalized medicine [1–6], evolutionary theory [7, 8], and forensics [9–11] and

Ricardo Quislant, Rafael Larrosa, Oscar Plata have contributed equally to this work.

Extended author information available on the last page of the article

Published online: 31 December 2024

has become even more relevant with the advent of third-generation sequencing (also known as long-read sequencing) [12], such as PacBio [13–16] and Nanopore [5, 16–19]. These technologies obtain reads of more than 10,000 base pairs (bp) in contrast with typically reaching up to 2x300 bp with Illumina (short reads) [20, 21], and they exhibit an impressive accuracy rate of 99.9%, achieved through various advancements, including the introduction of PacBio HiFi reads [5, 22] and improvements in Nanopore technology.

Advances in this field become increasingly relevant for the reconstruction of the complete genome, overcoming the challenges of short-read sequencing. In this context, algorithms based on overlap–layout–consensus (OLC) [23, 24] have established a prominent niche for advancement of long-read assembly due to their ability to handle repetitive sequences and complex genomics regions. However, the lack of a reference as a guide continues to pose significant computational challenges [25] and becomes more noticeable when the length and the complexity of the genome increase. In particular, the overlap step has been identified as the major bottleneck. This step involves all-versus-all pairwise alignment to detect overlapping regions among reads and leads to an unapproachable resource consumption as the complexity of the genome increases.

Pairwise alignment can be formulated as a string matching (SM) problem, typically solved using dynamic programming (DP) algorithms. Notable examples include the Levenshtein distance [26], Needleman–Wunsch [27], Smith–Waterman [28], and Smith–Waterman–Gotoh (SWG) [29]. However, these algorithms exhibit quadratic time and space complexity (i.e., $O(m \times n)$ between two sequences with lengths m and n). The Hirschberg algorithm [30] enhances the space efficiency of the Needleman–Wunsch algorithm through a divide-and-conquer approach, which becomes especially relevant for long-read alignment. Despite this, it still results in quadratic time complexity. One candidate to replace classical dynamic programming algorithms is the bit-parallel Myers (BPM) algorithm [31] for approximate string matching (ASM), as well as the enhanced version proposed by Hyyrö [32]. Both approaches achieve linear time complexity by employing bit-parallel techniques and efficient data structures.

The complexity of ASM has generated different proposals to exploit the inherent parallelism from different perspectives based on CPUs, GPUs, and FPGAs. However, despite these advancements, pairwise sequence alignment remains a great challenge in OLC algorithms [25, 33]. We acknowledge the computational cost imposed by memory accesses in sequence comparison, especially when handling massive sequencing datasets. This challenge becomes more significant as the volume of data grows. To tackle this issue, we develop *SeqMatcher*, an efficient vectorized sequence matching algorithm that takes full advantage of high data-level parallelism with Intel's AVX-512 extensions, enabling rapid comparison of millions of sequences.

Our main contributions can be summarized as follows:

1. *SeqMatcher* implements both the Myers' and Hyyrö's algorithm for efficient computation of the Levenshtein distance using AVX-512 instructions. Our implementation outperforms the former enabling faster and more efficient analysis of sequence similarity.
2. *SeqMatcher* enhances Myers' algorithm to efficiently handle short and long reads while introducing parallelism within each ASM operation. This modification enables improved performance and scalability for sequence analysis tasks.
3. *SeqMatcher* expedites two specific use cases of ASM in genome sequence analysis, namely edit distance calculation and path alignment. By enhancing the performance of these operations, we enable faster and more efficient sequence analysis workflows.
4. *SeqMatcher* implements a fast two-bit mapping proposal that achieves both (i) a reduction in memory footprint and (ii) accelerated sequence comparisons.

2 Background

2.1 The read matching problem

The objective of approximate string matching (ASM) applied to genomics is to identify the differences and similarities between two sequences. Let the query sequence $Q = q_1, q_2..q_m$ and the target sequence $P = p_1, p_2..p_m$ be two strings of elements from the genetic alphabet-adenine (A), guanine (G), cytosine (C), and thymine (T) with lengths $|Q| = m$ and $|P| = n$, and a positive threshold $k \geq 0$. Further, let $d(A, B)$ be the unit cost edit distance between strings A and B. Formally, ASM problem tries to find all positions j in P such that there is a suffix of $P[1...j]$ matching Q with k-or-fewer differences, that is, j such that $min_g \delta(Q, P[g..j]) \leq k$.

These differences, denoted by edits, can be classified as substitutions, deletions, or insertions in one or both sequences, and the cumulative cost of these edits represents the edit distance. Figure 1 shows an example of edits.

The Levenshtein distance and the Hamming distance are two widely used edit distance metrics for comparing the similarity between two sequences. The Hamming distance measures the number of symbol substitutions needed to transform one string into

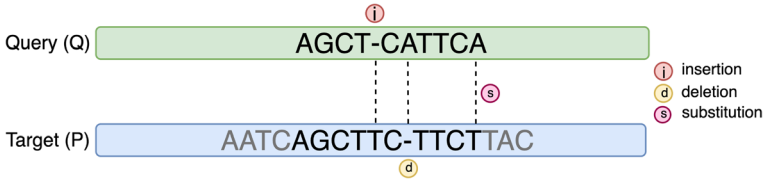


Fig. 1 Three types of errors (i.e., edits)

another, considering only strings of equal length. On the other hand, the Levenshtein distance encompasses not only substitutions but also insertions and deletions, allowing for comparisons between strings of different lengths. Additionally, the Damerau–Levenshtein distance adds the transposition of two adjacent characters to the Levenshtein distance. Determining the Hamming distance is a relatively easy task and has been effectively addressed. However, computing the Levenshtein and Damerau–Levenshtein distances efficiently remains a challenge, and it is the focus of this article.

ASM is necessary not only for determining the minimum number of edits between two genome sequences, but also for identifying the location and type of each edit. Since there can be numerous possible arrangements of edits and matches resulting in different alignments, ASM typically includes a trace-back step to find the path to the optimal alignment. The alignment score for each path is estimated as the sum of edit penalties and match scores along the path, based on a user-defined scoring function. The optimal alignment will be given by the path with the highest alignment score.

Levenshtein [26], Smith–Waterman [28], and Needleman–Wunsch [27] propose common dynamic programming (DP) implementations for ASM, but they all exhibit quadratic time and space complexity with respect to the length of the sequence pair, $O(mn)$, which raises the need to find lower complexity implementations in order to efficiently support long-read sequencing.

2.2 Myers' and Hyyrö's algorithm

Myers' algorithm [31] represents an improvement over the common DP-based ASM implementations. The Myers algorithm addresses the task of calculating the minimum edit distance between a target or reference sequence and a query sequence, allowing for a maximum of k errors. It efficiently identifies matches and mismatches between the target and the query, taking into account the specified error threshold.

Myers relied on the observation of Ukkonen [34] who noticed that adjacent values in a DP matrix can differ by ± 1 at most. Based on it, Myers re-encoded the DP scoring matrix by accounting only for the vertical and horizontal delta values among the adjacent cells.

As a result, each column j in the matrix could be represented by four bit-vectors which represent four states: horizontal positive (HP_j), horizontal negative (HN_j), vertical positive (VP_j), and vertical negative (VN_j). Myers conducted a comprehensive analysis of the connection between an individual cell and the entire column presented as bit-vectors, identifying the specific bit operations needed to compute a column based on the bit-vectors of the preceding column.

Thus, the complete scoring matrix could be obtained by sequentially calculating the bit-vector states and determining the scores at the bottom row. These properties allow us to represent the DP matrix D from the following bit-vectors:

$$\begin{aligned}
VP_j[i] = 1 \quad &\text{if and only if} \quad D[i,j] - D[i-1,j] = 1,\\
VN_j[i] = 1 \quad &\text{if and only if} \quad D[i,j] - D[i-1,j] = -1,\\
HP_j[i] = 1 \quad &\text{if and only if} \quad D[i,j] - D[i,j-1] = 1,\\
HN_j[i] = 1 \quad &\text{if and only if} \quad D[i,j] - D[i,j-1] = -1,\\
D0_j[i] = 1 \quad &\text{if and only if} \quad D[i,j] = D[i-1,j-1].
\end{aligned}$$

Notably, the edit distance scores in the last row exhibit only ± 1 changes or remain unchanged. The algorithm starts with maximum distance in the lower left cell. Changes are determined by the last bits of HP_j and HN_j: +1 if HP_j is set, -1 if HN_j is set, with j in 1...n. Both bits cannot be set simultaneously, and neither can indicate no change. Algorithm 1 shows Myers' proposal.

From a query of length m and a target of length n, initially, both VP_0 and VN_0 are set based on the initial conditions for the cells $D[i, 0]$. Specifically, VP_0[i] is assigned the value 1, and $VN_0[i]$ is set to 0 for $i \in 1...m$. Additionally, the cell $D[m, 1]$ is initialized with the value m. Subsequently, transitioning from column $j-1$ to column j encompasses the execution of the following four steps:

1. *Peq* encodes the positions of a specific nucleotide within the query sequence. For example, given the query sequence AATC, the Peq_A bit-vector is 1100 and Peq_T is 0010. Subsequently, *Peq* is used to determine where each nucleotide from the target sequence matches with the query sequence (line 12 of Algorithm 1).
2. The diagonal vector $D0_j$ is computed from Peq_j, VP_{j-1}, and VN_{j-1}.
3. The horizontal vectors HP_j and HN_j are computed from $D0_j$, VP_{j-1}, and VN_{j-1}.
4. The value $D[m, j]$ is calculated from $D[m, j-1]$ and the horizontal delta values $HP_j[m]$ and $HN_j[m]$.
5. The vertical vectors VP_j and VN_j are computed from $D0_j$, HP_j, and HN_j.

An approximate occurrence of the query concludes at text position j whenever $D[m, j] \leq k$ during the target scan.

Consequently, it results in a time requirement of $O(m)$ to precompute the *Peq* vector, with a total runtime of $O(m + n\lceil m/w \rceil)$, where w represents the word size of the machine (e.g., 32 or 64 bits). By enabling the packaging of queries in units of up to 64 bits, such as unsigned long long int, the overall time requirements are diminished to $O(m + n)$. However, for whole-genome sequencing, even short reads usually encompass 150 nucleotides.

Algorithm 1 Myers' bit-vector proposal [31]

```
1: function MYERS(Q, P, m, n, k) ▷ P and Q: target and query respectively; n and
   m: size of the target and query respectively; k: defined threshold
2:     for σ ∈ all characters do          ▷ For genome sequencing, all characters =
   {A,T,C,G}
3:         Peq_σ ← 0;
4:     end for
5:     for i from 1 to m do                                          ▷ preprocessing
6:         Peq_{Q[i]} ← Peq_{Q[i]} | 0^{m−1}10^{i−1};
7:     end for
8:     VP ← 1^m;
9:     VN ← 0^m;
10:    score ← m;
11:    for j from 1 to n do                                          ▷ Searching
12:        X ← Peq_{P[j]} | VN;
13:        D0 ← ((VP + (X & VP)) ^ VP) | X;          ▷ Diagonal zero delta vector
14:        HN ← VP & D0;                          ▷ Horizontal negative delta vector
15:        HP ← VN | ~(VP | D0);                  ▷ horizontal positive delta vector
16:        X ← HP << 1;                    ▷ Shifting HP for the next iteration
17:        VN ← X & D0;                             ▷ Vertical negative delta vector
18:        VP ← (HN << 1) | ~(X | D0);              ▷ Vertical positive delta vector
19:        if HP & 10^{m−1} then                                         ▷ Scoring
20:            score ← score + 1;
21:        else if HN & 10^{m−1} then
22:            score ← score − 1;
23:        end if
24:        if score ≤ k then
25:            Report a match ending at P_j;
26:        end if
27:    end for
28:    return score;
29: end function
```

Hyyrö [35] improved Myers' algorithm by reducing the number of nucleotides processed in each column and adopted a *banded* strategy to compute the matrix. The main concept behind this approach is to limit the calculations to a specific region called a *band* by introducing an error factor. This idea is similar to Ukkonen's proposal. By setting the value of k, which represents the maximum number of errors allowed, we can define the width of the *band*. Therefore, instead of computing the entire matrix, we only need to calculate the nucleotides that lie within the defined band, which limits the number of nucleotides processed in each column. This approach significantly reduces the memory required for the bit-vectors.

Typically, it is not necessary to consider more than 10% to 20% of the query length as an acceptable difference rate for finding similarities. Additionally, it is not practical to measure the edit distance for more than $k = 32$ differences when comparing texts or sequences. By limiting the maximum number of errors to

$k = 32$, the size of the bit-vectors will be small enough to fit into a single machine register, leading to a time complexity of $O(n)$.

Thus, this algorithm is separated in two phases, the diagonal and the horizontal phase. In the diagonal phase, the calculations progress incrementally downwards in the matrix until reaching the lower boundary. Figure 2 illustrates the *banded* approach. In this example, we employ a band length of four nucleotides. We represent the diagonal phase using a solid color and the horizontal phase with a zig-zag line pattern. Hyyrö's proposal is shown in Algorithm 2, specifically illustrating the computation of the diagonal phase for column j based on the known values of *VP* and *VN* from column $j - 1$. The score is determined by verifying the matrix cell value at the lower boundary of the current position, increasing by 1 only in the case of a symbol mismatch. Later, the original Myers' algorithm (described in Algorithm 1) can be used to compute the horizontal phase. After completing the diagonal phase, the estimated score can be utilized to check if a cutoff point has been reached, allowing for an early termination before proceeding to the horizontal processing phase.

Algorithm 2 Hyyr¨o bit-vector algorithm [35] for Levenshtein distance

Computing j column from $j-1$ column
$X \leftarrow Peq_{P[j]} \mid VN_{j-1}$;
$D0 \leftarrow ((VP_{j-1} + (X \,\&\, VP_{j-1})) \hat{} VP_{j-1}) \mid X \mid VN_{j-1}$;
$HP \leftarrow VN_{j-1} \mid (D0 \mid VP_{j-1})$;
$HN \leftarrow D0_j \,\&\, VP_{j-1}$;
$VP \leftarrow HN_j \mid \tilde{} ((D0_j >> 1) \mid HP_j)$;
$VN \leftarrow (D0_j >> 1) \,\&\, HP_j$;

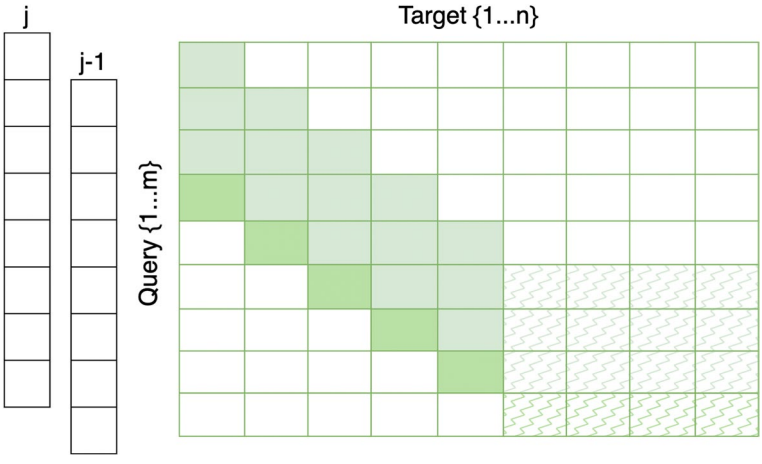


Fig. 2 *Banded* algorithm scheme

However, despite that Hyyrö's strategy reduces the complexity of the algorithm, it presents two main disadvantages. First, the search space in the target sequence is constrained, so if the alignment regions are located at distant positions between the two sequences, accurate identification could be hindered. It involves that a filtering model must be adopted beforehand. Second, from a vectorization standpoint, the linear complexity is compromised with long reads when the length of the *band* surpasses the size of CPU registers, as the number of nucleotides to be processed in each iteration or in each column surpasses 32 or 64 bits. For example, by setting a threshold of 4% or 5% of the sequence length, as in *BitMapper* [36], a band of 600 nucleotides would be required for a sequence length of 15,000 bp. It hinders the widespread adoption of either approach and emphasizes the need for further advancements in alignment refinement algorithms.

2.3 Vectorization extensions: AVX-512

Intel AVX-512 is a significant advancement in vector computing which builds upon the previous Intel AVX instruction set extension. It enables a single instruction to operate on multiple data values simultaneously, just like its predecessors, but with an expanded capacity of up to 512 bits. This extension greatly enhances the computational power and efficiency of processors by enabling larger and more complex data processing operations, which makes it suitable for processing long queries in ASM.

The new masking feature of AVX-512 allows for efficient vectorization of conditional code. It enables an algorithm to selectively operate on relevant elements of the vectors faster than AVX and AVX2. The embedded broadcast feature facilitates the direct use of scalar values in calculations by means of prefixes, without requiring an extra instruction. The embedded rounding control provides precise control over rounding or exceptions for specific instructions, ensuring accurate calculations without modifying the control register. Moreover, the availability of new instructions simplifies complex calculations that previously required multiple instructions, leading to streamlined computation and improved performance.

These advanced capabilities of Intel AVX-512 greatly contribute to take advantage of the high amount of bit parallelism available in the Myers' algorithm without the need for any additional hardware implementation.

3 Related work

Various improvements and optimizations have been proposed in the literature for the Myers' algorithm and the *banded* version of Hyyrö, particularly in the context of short-read sequence alignment. These enhancements aim to accelerate the alignment process and improve computational efficiency in specific environments.

In the realm of CPUs, *Edlib*, used by *Medaka* [37] and *Dysgu* [38], represents the forefront implementation of Myers' bit-vector algorithm. It uses Ukkonen's banded algorithm to reduce the space of search and extends Myers's algorithm to support the global alignment method and the semi-global alignment. Moreover, libraries, such as *SeqAn* [39], also include an implementation of Myer's algorithm using SIMD instructions. Additionally, there are several state-of-the-art CPU implementations available for performing sequence alignment. From a SIMD perspective, *Parasail* [40] and *KSW2* [41] (component of minimap2 [42]) present notable SIMD C (C99) pairwise sequence alignment libraries. Also, *BSAlign* [43] is also introduced as a new tool that utilizes an active F-loop for striped vectorization and a striped move for banded dynamic programming (DP). Other libraries, such as *WFA2-lib* [44], implement the wavefront alignment (WFA) algorithm. It employs simple computational patterns that can automatically vectorize for different architectures without requiring code adaptation. Finally, we can add *QuickEd* [45], a high-performance exact sequence alignment based on the bound-and-align paradigm.

Furthermore, from a hardware approach GPUs have been widely adopted as hardware accelerators for pairwise sequence alignment. Thus, GPU implementations, such as Chacón [46], provide a CUDA version of Myers' algorithm using a thread cooperative approach. Also, *GASAL2* [47] provide efficient implementations of other sequence alignment algorithms, including *Needleman–Wunsch* and *Smith–Waterman*. Others, such as *WFA-GPU* [48], present a CPU-GPU co-design capable of performing inter-sequence and intra-sequence parallel sequence alignment. Similarly, advancements in FPGA architectures have enabled parallel approaches to sequence alignment, with Cai [49], D.P. Bautista [50], and Rufas [51] who provide FPGA-based implementations of Myers' algorithm.

Thus, FPGA-based implementations demonstrate significant performance improvements in terms of sequence pair comparisons per second. Specifically, D.P. Bautista's implementation achieves 0.3 million comparisons per second for sequences pair of 112×128, while Cai reports an impressive 70 million comparisons per second using the Kintex KCU1500 FPGA, and Rufas achieves 91.5 million and 47.4 million comparisons per second using the D5005 and HARPv2 FPGAs, respectively, for sequence pairs of sizes 48×48, 100×120, and 100×120. However, despite these improvements, these results only assess the performance using short reads. For example, in the case of Rufas, the longest sequence length is 300×360, achieving performance rates of 16.6 and 47.0 million comparisons per second for the D5005 and HARPv2 FPGAs, respectively. Furthermore, these implementations rely only on the banded version of Myers' algorithm and do not compute the entire matrix. Additionally, Cai and Rufas implementation does not support trace-back. Finally, FPGA-based accelerators require specialized programming and hardware, making them less accessible to the scientific community.

On the other hand, CPU- and GPU-based implementations provide a more complete alignment with trace-back, although they exhibit lower performance. Recent GPU implementations, such as *WFA-GPU*, are capable of processing 0.5 million sequences per second. While it outperforms other GPU-based approaches, such as *GASAL2*, its performance remains below the results reported by Cai and Rufas. Additionally, the CPU version (WFA) outperforms tools such as *Parasail*, *KSW2*, *Edlib*, and *SeqAn*, but achieves a lower throughput of 0.47 million sequences per second using 10 threads.

We recognize these limitations and aims to increase the performance of CPU approaches with a CPU accelerator that is accessible and user-friendly for the scientific community. Thus, we introduce a CPU-based pairwise aligner that leverages the extended AVX-512 registers to perform pairwise alignment using both unbanded and banded approaches based on the Myers and Hyyrö algorithms. *SeqMatcher* (i) supports both short and long reads, (ii) includes trace-back functionality to retrieve the complete alignment path, and (iii) scales efficiently for large genomes.

4 SeqMatcher implementation

4.1 Algorithm overview

The primary purpose of *SeqMatcher* is to accelerate pairwise sequence alignment through a memory-frugal and efficient algorithm. *SeqMatcher* addresses ASM problem with the Levenshtein distance [26] as the cost metric, which aims to find the minimum number of single-letter substitutions, insertions, and deletions to convert the target into the query. Thus, based on Myers' [31] and Hyyrö's [32] bit-vector algorithms described in Sect. 2.2, *SeqMatcher* introduce a fast and low-power CPU accelerator capable of (i) supporting trace-back, (ii) harnessing the extended registers of AVX-512 for processing long reads, and iii) enabling data-level parallelism with AVX-512 and harnessing the thread-level capabilities of multi-core systems with OpenMP to handle a large number of iterations efficiently. Thus, by exploiting the ultra-wide registers of AVX-512, *SeqMatcher* processes long sequences reducing memory overhead, improving computational speed, and enhancing energy efficiency.

We describe the implementation of *SeqMatcher* in Sects. 4.2 and 4.3. It comprises three steps: preprocessing, matching, and score estimation. In the first step, we read the file with the sequences (in *FASTQ* or *FASTA* formats) and we map the sequences into two bits, 00 for adenine (A), 01 for cytosine (C), 10 for thymine (T), and lastly, 11 for guanine (G). Second, we proceed column by column through the DP matrix and calculate each bit-vector using Myers' approach for the *unbanded* version and Hyyrö's approach for the *banded* version. Finally, we estimate the score for each comparison.

Algorithms 4 and 9 show a high level overview of *unbanded* and *banded* versions of *SeqMatcher*, respectively.

4.2 Preprocessing

DNA and RNA sequences are made up of only four nucleotide bases: adenine (A), guanine (G), cytosine (C), and thymine (T) in DNA, and uracil (U) instead of thymine in RNA. Consequently, two bits are enough to represent each symbol of a sequence instead of the one-byte ASCII representation.

This involves a fourfold reduction in space and, more importantly, it enables the processing of a larger number of nucleotides simultaneously in a single register. In fact, leveraging AVX-512 long registers we can preprocess sequences in chunks of 64 nucleotides to end up with 512 nucleotides codified into two 512-bit registers. This can be effectively tackled with SIMD instructions, enabling us to: (i) avoid expensive branches, (ii) minimize data dependencies, and (iii) get the maximum efficiency from vectorization instructions, in contrast with memory-centric approaches like *memcpy* (*std::ptr::copy_nonoverlapping* in Rust).

In the 8-bit ASCII code representation, the second and the third least significant bits align with the two-bit encoding of characters in the DNA alphabet (A, C, G, T), where 00 represents A, 01 is C, 10 represents T, and 11 corresponds to G.

From a SIMD perspective, we resolve the conversion from ASCII to our two-bit-per-nucleotide representation by applying only two operations to each 8-bit element in both, query and target sequences: (i) right shifting the ASCII character once and (ii) extracting the least significant bit of it to get the first bit of the nucleotide. Then, we repeat the previous sequence of operations to get the second bit of the nucleotide. Since we can process 64 ASCII-coded nucleotides in each 512-bit register, this results into two 64-bit unsigned integers, which are stored in two separate 64-bit unsigned integer arrays for the query, called *query.bit*1 and *query.bit*2, and likewise for the target, called *target.bit*1 and *target.bit*2. This separation enables us to perform bitwise operations more easily compared to storing the two bits in the same array. Then, this process is repeated in chunks of 64 nucleotides until the entire sequence pair is translated.

In the next steps of *SeqMatcher*, we only have to load eight elements from the *bit*1 array to a 512-bit register and other eight elements from the *bit*2 array to another 512-bit register, having 512 nucleotides codified into two 512-bit registers.

4.3 Matching and scoring

After mapping all query and target sequences to our two-bit representation, *SeqMatcher* compares each pair of sequences in parallel, following a thread-level

approach. We distribute the pairs among threads using a static scheduling, since the processing time of each sequence pair is similar (read length exhibits low variability). The thread work function is shown in Algorithm 3.

To better understand the different kind of variables used in the algorithms, we define the following variable types:

- u(#) is a #-bit unsigned integer scalar. We use both u(32) and u(64) denoting 32-bit and 64-bit unsigned integers. We also use u(1) as a boolean.
- s(#) is a #-bit signed integer scalar.
- V(#) is an array of elements of type #. We use V(u(64)) as 64-bit unsigned integers allocated in memory, e.g., V(u(64)) bit1 = {u(64) a, u(64) b, u(64) c,...}. Arrays are accessed with indexes starting at 1.
- S(#) is a struct of type #. In this case, we define two different struct types: S(seq) comprises two 64-bit unsigned integer arrays and a 32-bit scalar, S(seq) { V(u(64)) *bit*1; V(u(64)) *bit*2; u(32) *length*}, which is used to encode a sequence of nucleotides (in our two-bit-per-nucleotide format) and its length; S(align) is a struct comprising two 32-bit unsigned integer scalars, S(align) {u(32) pos; s(32) score};
- R(512) is a 512-bit register.

We also use the ternary conditional operator with the syntax: $condition\,?\,true\_expr\,:\,false\_expr$, where *condition* can be a logical expression or a single variable of types u(#), s(#) or R(512). The single variable evaluates to false if it is equal to 0, and true otherwise.

Algorithm 3 outlines SeqMatcher implementation without trace-back. On processing each pair of sequences, if the query sequence is larger than 512 nucleotides, it is broken down into chunks. The computation of each chunk is performed in a banded or unbanded vectorized manner.

We describe both *banded* and *unbanded* strategies in Sects. 4.3.1 and 4.3.2. The final alignment position is that whose score is minimum (line 14). The less than or equal operator is used to get the longest possible alignment. In case the query sequence is shorter than 512 (line 18), splitting the query is not necessary.

Additionally, *SeqMatcher* computes the alignment with trace-back (not detailed in the algorithms). Similar to the alignment without trace-back, if the query sequence exceeds 512 nucleotides, *SeqMatcher* divides the sequence into chunks and, in this case, selects the longest alignment based on the start and end positions.

Algorithm 3 SeqMatcher algorithm without path trace-back computation

```
1: function SeqMatcher(S(seq) query, S(seq) target, u(1) unbanded)
2:     u(32) qChunks ← ⌊(query.length + 512 − 1)/512⌋;
3:     u(32) rest ← query.length mod 512;
4:     S(align) alignment ← {0, 2^32 − 1};   ▷ Holds position and score of the final
       alignment. Initialization: pos to 0, score to max
5:     if qChunks > 1 then
6:         for i from 1 to qChunks do ▷ Iterations assigned to threads, e.g., omp for
7:             u(32) sz ← (i = qChunks ∧ rest ≠ 0) ? rest : 512;
8:             u(32) pq ← (i − 1) ∗ 8 + 1;              ▷ Current query.bit# position
9:             if unbanded then
10:                tmpAlign ← SeqMatcher_unbanded(query, target, pq, sz);
11:            else
12:                tmpAlign ← SeqMatcher_banded(query, target, pq, sz);
13:            end if
14:            if tmpAlign.score ≤ alignment.score; then
15:                alignment ← tmpAlign;
16:            end if
17:        end for
18:    else
19:        if unbanded then
20:            alignment ← SeqMatcher_unbanded(query, target, 1, rest);
21:        else
22:            alignment ← SeqMatcher_banded(query, target, 1, rest);
23:        end if
24:    end if
25:    return alignment
26: end function
```

4.3.1 Unbanded algorithm

We develop our *unbanded* vectorized algorithm, outlined in Algorithm 4, by leveraging Myers' algorithm principles. It proceeds column by column through the DP matrix and returns (i) the end position in the target sequence where the query aligns with the target, (ii) the edit distance also known as score, and (iii) the start and end locations in the target of the optimal matching path. The first two outcomes involve alignments without trace-back, while the third outcome denotes an alignment with trace-back. For the sake of conciseness, Algorithms 3 and 4 show the approach without trace-back.

Algorithm 4 *Unbanded* version of SeqMatcher algorithm for the estimation of the final position in the target

```
1: function SeqMatcher_unbanded(S(seq) query, S(seq) target, u(32) pq, u(32) sz)  ▷ query and target: structs with two 64-bit unsigned int arrays storing preprocessed reads and their lengths; pq: query position to load from; sz: length of the query sequence chunk
2:     u(32) tChunks ← ⌊(target.length + 512 − 1)/512⌋, pt ← 1;
3:     u(32) rest ← target.length mod 512;
4:     s(32) score ← 0;
5:     S(align) alignment ← {0, 2^31 − 1}  ▷ alignment.score gets max signed integer
6:     R(512) VP ← 2^512 − 1, VN ← 0, jMask ← 1;
7:     R(512) bit1q ← load_64(query.bit1[pq]);  ▷ Load 512 nucleotides of ...
8:     R(512) bit2q ← load_64(query.bit2[pq]);  ▷ the query in 8 packs of 64
9:     for t from 1 to tChunks do
10:        R(512) bit1t ← load_64(target.bit1[pt]);  ▷ Load 512 nucleotides of ...
11:        R(512) bit2t ← load_64(target.bit2[pt]);  ▷ the target in 8 packs of 64
12:        pt ← pt + 8;  ▷ Current target.bit# position
13:        for j from 1 to (t = tChunks ∧ rest ≠ 0) ? rest : 512 do
14:            u(1) bit1tj ← bit_and(bit1t, jMask) ? 1 : 0;  ▷ Get target nucleotide ...
15:            u(1) bit2tj ← bit_and(bit2t, jMask) ? 1 : 0;  ▷ in column j
16:            jMask ← ROL(jMask);  ▷ Rotate 512-bit reg left
17:            R(512) peq ← BUILD_PEQ(bit1q, bit2q, bit1tj, bit2tj);
18:            R(512) X ← bit_or(peq, VN);
19:            R(512) D0 ← bit_or(bit_xor(VP, SUM(VP, bit_and(X, VP))), X);
20:            R(512) HN ← bit_and(VP, D0);
21:            R(512) HP ← bit_or(VN, not(bit_or(VP, D0)));
22:            score ← score+(bit_and(HP, 2^(sz−1)) ? 1 : 0)−bit_and(HN, 2^(sz−1)) ? 1 : 0;
23:            if score < alignment.score then
24:                alignment.score ← score;
25:                alignment.pos ← j + (t − 1) ∗ 512;
26:            end if
27:            X ← SLLI(HP);
28:            VN ← bit_and(X, D0);
29:            VP ← bit_or(SLLI(HN), not(bit_or(X, D0)));
30:        end for
31:    end for
32:    return alignment;
33: end function
```

First, we load the query into two 512-bit registers (lines 7 and 8, Algorithm 4) and calculate each bit-vector column by column. If the query is larger than 512 nucleotides, *SeqMatcher* splits it into 512-nucleotide chunks for processing (see Algorithm 3). Conversely, we process the target in 512-nucleotide chunks by accessing each position using a bitwise *and* operation, *bit_and*(), in lines 14 and 15. This operation employs a mask to extract the initial desired bit (set to 1), followed by a left rotation of the mask (line 16), to access the next nucleotide in

the next iteration. It considerably reduces the continuous memory accesses inherent when storing the target sequence in memory, reducing access latency.

Since Intel does not provide specific intrinsics for performing one-bit left rotations of the entire 512 register and only provides these intrinsics for operations on 16/32/64-bit elements, we propose this operation in Algorithm 5 using intrinsics that operate on packed 64-bit elements and performing bitwise operations. Moreover, Fig. 3 illustrates a high-level example of the left rotation operation. First, we perform a one-bit left rotation on each 64-bit element so that the *msb* becomes the *lsb*. Second, we create two masks: (i) one, *lsb*0*Mask*, with all ones except for the *lsb* of each packed 64-bit integer, and (ii) another, *lsb*1*Mask*, with all bits set to zero except for the *lsb*. We then apply these masks to the previously rotated register. The result of masking with *lsb*0*Mask*, a process we call *lsb* clearing, goes to the 4^{th} step. The other one, as a result of what we call the *lsb* extracting process, is processed in the 3^{rd} step, where it is concatenated with itself, shifted right 7 64-bit elements and then truncated to the low 8 elements. This shuffles the *lsb's* properly so that the 8^{th} element *lsb* ends up in the 1^{st} element, and the rest shift left one element. Finally, in the 4^{th} step, we apply an bitwise or of the two resulting registers.

Algorithm 5 Rotate 512-bit left

function ROL($R(512)$ reg) ▷ Single-bit left rotation given a 512-bit register reg
  $R(512)$ $lsb0Mask \leftarrow set\_64(2^{64}-2)$, $lsb1Mask \leftarrow set\_64(1)$;
  $R(512)$ $regRotated \leftarrow rol\_64(reg, 1)$; ▷ 1-bit left rotation per 64-bit element
  $R(512)$ $lsbExtracted \leftarrow bit\_and(regRotated, lsb1Mask)$;
  $R(512)$ $lsbCleared \leftarrow bit\_and(regRotated, lsb0Mask)$;
  $R(512)$ $lsbAligned \leftarrow alignr\_64(lsbExtracted, lsbExtracted, 7)$; ▷ Concatenate and shift the result right 7 64-bit elements, and store the low 8 elements
  return $bit\_or(lsbAligned, lsbCleared)$; ▷ Replace and return
end function

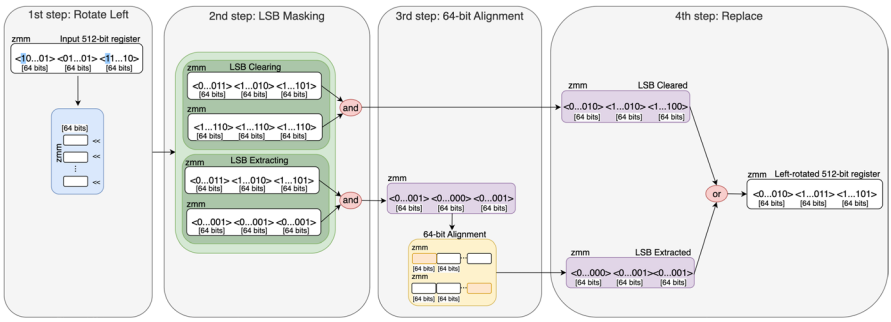


Fig. 3 Implementation of the one-bit left rotation operation

In Myers' algorithm, *Peq* is precalculated for each query sequence (see Algorithm 1, lines 2-7), which involves breaking down the query into 4 bit-vectors, one per nucleotide, and setting the corresponding element of each bit-vector to 1 if the nucleotide is present in the given position of the query (e.g., Q = ACTAGC leads to $Peq_A = 100100$, $Peq_C = 010001$,...). *Peq* is then used to determine match positions with a target nucleotide easily.

However, *SeqMatcher* does not precompute *Peq* for the query in advance, because this approach leads to inefficiencies from a SIMD perspective, where the goal is to minimize memory accesses, which substantially impact performance with long genomes. Even with cache optimizations, precomputing *Peq* results in at least four main memory accesses per query sequence (one per nucleotide) and *m* additional cache accesses for each column of the target sequence. Thus, and provided that the cost of computing *Peq* to determine the matches with the target nucleotide is negligible, we calculate it in each iteration (line 17 in Algorithm 4).

Consequently, once the query is loaded in 512-bit registers and the target nucleotide in the current column is known, *SeqMatcher* calculates the *Peq* bit-vector for the given target nucleotide using only two bitwise operations on the two registers containing the query (see Algorithm 6): (i) two *xor* operations to determine the presence of each bit of the target nucleotide in the query and (ii) a *nor* operation to determine the presence of the two bits of the target nucleotide in the query (implemented with an *or* and an *xor* because of the absence of nor intrinsics). We illustrate a straightforward example in Fig. 4 which finds the positions in the query where the nucleotide adenine (A), represented by the two bits 00, appears.

Algorithm 6 Find matches of one target nucleotide in a 512-nucleotide query chunk

1: **function** BUILD_PEQ($R(512)$ bit1q, $R(512)$ bit2q, s(1) bit1tj, s(1) bit2tj)
2: $R(512)\ r1 \leftarrow bit\_xor(bit1q, bit1tj\ ?\ 2^{512} - 1 : 0)$;
3: $R(512)\ r2 \leftarrow bit\_xor(bit2q, bit2tj\ ?\ 2^{512} - 1 : 0)$;
4: **return** $bit\_xor(bit\_or(r1, r2), 2^{512} - 1)$;
5: **end function**

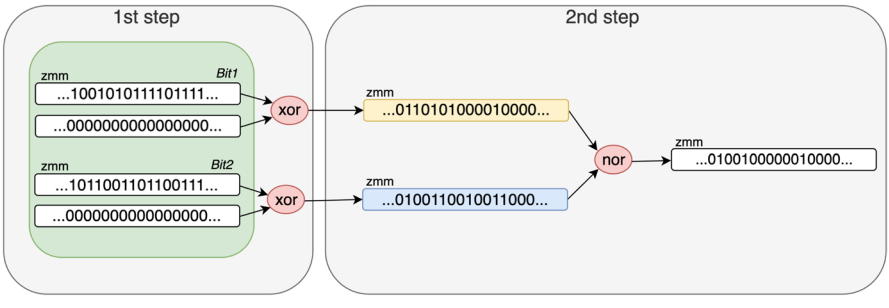


Fig. 4 Example of the *Peq* bit-vector calculation for the nucleotide adenine, represented by 00

Once we obtain the value of the *Peq* bit-vector, calculating each bit-vector of the scoring matrix is similar to Myers' approach (see Algorithm 1). However, since Intel does not provide intrinsics for neither full 512-bit register addition (line 19, Algorithm 4) nor 512-bit register left shifting (lines 27 and 29), which is similar to left rotation, we propose these operations in Algorithms 7 and 8, respectively.

In Algorithm 7, we implement left shifting by first performing the left rotation in Algorithm 5 and then applying a bitwise and operation with a 512-bit register that has all bits to one except for the *lsb*.

In Algorithm 8, we perform a full 512-bit addition with carry propagation across 64-bit elements inspired by the hardware Kogge-Stone Adder [52]. First, we add registers a and b in packs of 64-bit elements, storing the result in s and ignoring overflow. We then generate a bit-mask c to identify elements that generated a carry (those where $s < a$), and a bit-mask m for elements that reach the maximum possible value and will generate a carry if the carry-in to them is 1. After that, we sum them (with c shifted left by 1) and then xor the result to obtain the final bit-mask m holding which elements of s should be added 1. Finally, we perform the operation $s - max\_word$ $(= s - (-1))$ on every 64-bit element of s whose corresponding m bit is set to 1, leaving the rest intact (line 8).

Unlike Hamming distance verification, which relies on matches where the difference between the start and final positions equals the read length, Myers' algorithm exclusively provides the final position of a match. Specifically, Myers' algorithm determines the minimal number of errors (score) for a fixed final position and a free start position. However, to reveal the alignment path, determining the start position of the solution in the target is essential. To achieve this, *SeqMatcher* searches the target backward from the final position using Myers' algorithm on the reverse query and target sequences (not shown in the algorithms).

Algorithm 7 Shift 512-bit left

```
1: function SLLI(R(512) reg)        ▷ Single-bit left shift given a 512-bit register reg
2:     R(512) lsb0Mask ← 2^512 − 2;                          ▷ All 1s but the lsb
3:     return bit_and(ROL(reg), lsb0Mask);                   ▷ Set lsb to 0 and return
4: end function
```

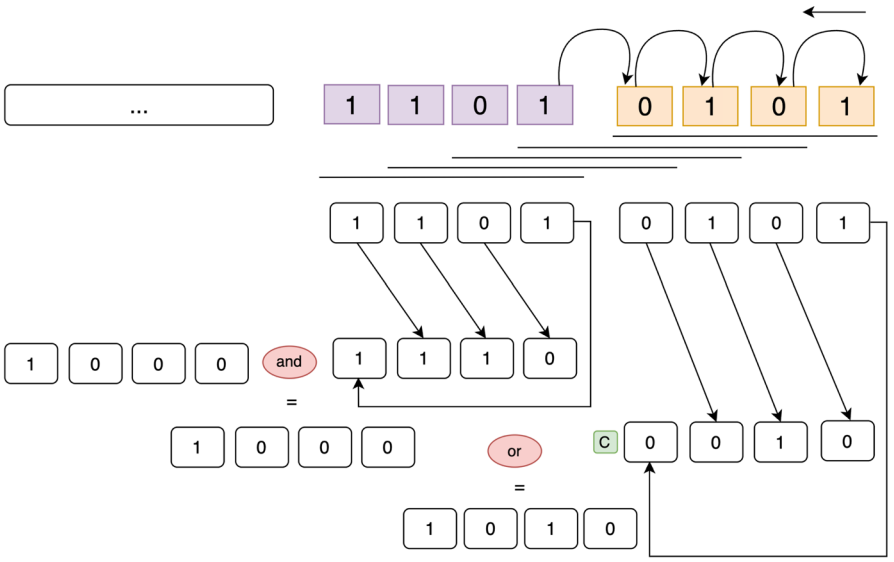


Fig. 5 Example of band displacement. Using a 4-bit band, we apply a rotate tight (ROR) instruction to shift the current bit-vector (0101), moving the new bit (1) from the *lsb* position in the next bit-vector to the *msb* position in the current bit-vector

Algorithm 8 Little-Endian 512-bit Full Adder

function SUM($R(512)$ a, $R(512)$ b) ▷ Sum of two 512-bit registers a y b
  $R(512)$ $max\_word \leftarrow 2^{512} - 1$;
  $R(512)$ $s \leftarrow add\_64(a, b)$; ▷ Add registers in packs of 64-bit elements
  $s(8)$ $c \leftarrow cmp\_lt\_64(s, a)$; ▷ Compare 64-bit elements for less-than, and store the results in an 8-bit mask
  $s(8)$ $m \leftarrow cmp\_eq\_64(s, max\_word)$; ▷ Compare for equal to max_word
  $s(32)$ $carry \leftarrow m + c * 2$; ▷ Shift c left by 1
  $m \leftarrow m \hat{} carry$; ▷ Bitwise xor
  return $mask\_sub\_64(s, m, s, max\_word)$; ▷ Subtract packed 64-bit integers in max_word from packed 64-bit integers in s, using writemask m. Elements in s whose corresponding m bit is not set are left intact
end function

Algorithm 9 *Banded* version of *SeqMatcher* algorithm for the estimation of the final position in the target

```
1: function SeqMatcher_banded(S(seq) query, S(seq) target, u(32) pq, u(32) sz)
   ▷ Same parameters as SeqMatcher_unbanded
2:     R(512) VP ← 2^512 − 1, VN ← 0, jMask ← 1;
3:     s(32) score ← 0;
4:     S(align) alignment ← {0, 2^31 − 1} ▷ alignment.score gets max signed integer
5:     u(32) tChunks ← ⌊(target.length + 512 − 1)/512⌋, pt ← 1, j ← 1, count ← 1;
6:     u(32) column ← query.length, k ← 512;
7:     R(512) bit1q ← load_64(query.bit1[pq]);          ▷ Load 512 nucleotides of ...
8:     R(512) bit2q ← load_64(query.bit2[pq]);          ▷ the query in 8 packs of 64
9:     R(512) bit1qn ← load_64(query.bit1[pq + 8]);          ▷ Load next 512 ...
10:    R(512) bit2qn ← load_64(query.bit2[pq + 8]);      ▷ nucleotides of the query
11:    for t from 1 to tChunks do
12:        R(512) bit1t ← load_64(target.bit1[pt]);      ▷ Load 512 nucleotides of ...
13:        R(512) bit2t ← load_64(target.bit2[pt]);      ▷ the target in 8 packs of 64
14:        pt ← pt + 8;                                  ▷ Current target.bit# position
15:        while column > k do
16:            u(1) bit1tj ← bit_and(bit1t, jMask) ? 1 : 0;    ▷ Get target nucleotide...
17:            u(1) bit2tj ← bit_and(bit2t, jMask) ? 1 : 0;          ▷ in column j
18:            jMask ← rol(jMask);
19:            R(512) peq ← build_peq(bit1q, bit2q, bit1tj, bit2tj);
20:            R(512) X ← bit_or(peq, VN);
21:            R(512) D0 ← bit_or(bit_xor(VP, sum(VP, bit_and(X, VP))), X);
22:            D0 ← bit_or(D0, VN);
23:            score ← score + 1 − bit_and(D0, 2^(sz−1)) ? 1 : 0;
24:            if score < alignment.score then
25:                alignment.score ← score;
26:                alignment.pos ← j;
27:            end if
28:            R(512) HP ← bit_or(VN, not(bit_or(D0, VP));
29:            R(512) HN ← bit_and(D0, VP);
30:            X ← srli(D0);
31:            VP ← bit_or(HN, not(bit_or(X, HP));
32:            VN ← bit_and(X, HP);
33:            bit1qn ← ror(bit1qn);
34:            bit2qn ← ror(bit2qn);
35:            bit1q ← bit_or(srli(bit1q), bit_and(bit1qn, 2^511));
36:            bit2q ← bit_or(srli(bit2q), bit_and(bit2qn, 2^511));
37:            if count = 512 then      ▷ Load next 512 nucleotides from the query
38:                count ← 0;
39:                pq ← pq + 8;
40:                bit1qn ← load_64(query.bit1[pq + 8]);
41:                bit2qn ← load_64(query.bit2[pq + 8]);
42:            end if
43:            count ← count + 1;
44:            column ← column − 1;
45:            j ← j + 1;
46:        end while
47:    end for
48:    return alignment;
49: end function
```

4.3.2 Banded algorithm

Based on Hyyrö's proposal, *SeqMatcher* employs a vectorized *banded* variant of Myers' algorithm that is described in Algorithm 9, which shows the implementation of the diagonal step. Like the unbanded variant, it yields two key outputs: (i) the optimal end location for each query in the target and (ii) the edit distance, or score.

Similar to Hyyrö's proposal, *SeqMatcher* consists of two phases: (i) diagonal, where it utilizes a band of 512 nucleotides, and (ii) horizontal, where it employs the Myers' approach to process the remaining nucleotides. With a value k as threshold for the differences set at 3–5% of the length of the target sequence, *SeqMatcher* can handle sequences of up to 17,000 nucleotides. For longer sequences requiring a band larger than 512 nucleotides, we use a similar strategy to that outlined in Sect. 4.3.1 for alignment without trace-back.

The loading and reading of the target remain similar to the previous *unbanded* algorithm (see Algorithm 4); however, the query loading process involves additional steps. To efficiently shift the band along the query, *SeqMatcher* performs a right shift operation and incorporates the new bit on the left in each iteration (see lines 35 and 36). Since the nucleotides are stored in the register from *lsb* to *msb*, we incorporate the new bits at the *msb* position. Figure 5 provides a high-level overview of this operation. First, *SeqMatcher* performs a right rotation on the next register (from which we can extract the next bit to incorporate) and a right shift on the current register, respectively, using Algorithms 10 and 11. Next, it extracts the *msb* from the rotated register using an *and* operation and incorporates it into the shifted register with an *or* operation.

The computation of each bit-vector is similar to Hyyrö's algorithm and can be performed using AVX-512 intrinsics, except for the addition and right shift operations (see lines 30 and 21), which are implemented by Algorithms 8 and 11, respectively, as previously discussed.

Finally, the score is estimated by verifying the matrix cell value at the lower boundary of the current position, increasing by 1 only in the case of a symbol mismatch. This value can be identified by checking the last bit (2^{xz-1}) state of $D0$. The diagonal state is 0 if the distance increases, and 1 if it does not.

When the lowest boundary is reached, the horizontal phase should be computed by implementing the *unbanded* version described in Algorithm 4, but it is restricted inside the width of the band. Similar to the *unbanded* version (described in Sect. 4.3.1), we employ here a sequential approach intra-sequence.

Algorithm 10 Rotate 512-bit right

function ROR($R(512)$ reg) ▷ Single-bit right rotation given a 512-bit register reg
  $R(512)$ $msb0Mask \leftarrow set\_64(2^{63}-1)$, $msb1Mask \leftarrow set\_64(2^{63})$;
  $R(512)$ $regRotated \leftarrow ror\_64(reg, 1)$; ▷ 1-bit right rotation per 64-bit element
  $R(512)$ $msbExtracted \leftarrow bit\_and(regRotated, msb1Mask)$; ▷ Extract the msb from each 64-bit element
  $R(512)$ $msbCleared \leftarrow bit\_and(regRotated, msb0Mask)$; ▷ Clear the msb bit on each 64-bit element
  $R(512)$ $msbAligned \leftarrow alignr\_64(msbExtracted, msbExtracted, 1)$; ▷ Concatenate and shift the result right 1 64-bit element, and store the low 8 elements
  return $bit\_or(msbAligned, msbcleared)$ ▷ Replace and return
end function

Algorithm 11 Right shift a 512-bit register

function SRLI($R(512)$ reg) ▷ Single-bit left right given a 512-bit register reg
  $R(512)$ $msb0Mask \leftarrow 2^{511}-1$; ▷ All 1s but the msb
  return $bit\_and(\text{ROR}(reg), msb0Mask)$; ▷ Set msb to 0 and return
end function

5 Experimental evaluation

5.1 Methodology

5.1.1 Data input

The experiments are conducted with simulated datasets of different lengths. Specifically, we use the *Pan troglodytes* [53] reference genome to generate random target sequences of varying lengths with *SimLord* [54]. Subsequently, we obtain

Table 1 Summary of dataset characteristics

Read set		Target set	
Length (bp)	Number	Length (bp)	Number
300	1,000,000	320	1,000,000
300	1,000,000	360	1,000,000
512	1,000,000	620	1,000,000
512	1,000,000	660	1,000,000
5,000	1,000,000	10,000	1,000,000
5,000	1,000,000	12,000	1,000,000
5,000	1,000,000	15,000	1,000,000

query sequences by extracting sub-sequences of a specified length from each target sequence, selecting random start and end positions with an error range of 0.5–1%, which corresponds to the typical error rate observed in newer sequencing technologies [16, 25]. We provide a summary of different sets of sequences used in Table 1.

5.1.2 Performance assessment

Measurement metrics We evaluate the performance of *SeqMatcher* on two levels: (i) validating the potential of our vectorization approach and (ii) assessing the performance of thread-level parallelism. Moreover, we evaluate the performance by comparing our approach against state-of-the-art methods by means of: (1) the alignment speed, measured in sequence pairs comparisons per second, or pairs comparisons per second (PCPS), for simplicity, (2) energy consumption, measured in Joules (J), and (3) RAM peak. For the energy consumption analysis, we use the *perf* profiler, specifically measuring the energy consumed by the processor package using the `energy-pkg` event.

Baseline implementations

We compare *SeqMatcher* with a non-vectorized baseline implementation (both non-band-based and band-based), as well as with two state-of-the-art implementations of Myers' algorithm: (i) the *Edlib* [55] and *SeqAn* [56] libraries and (ii) other aligners, including *BSAlign* [43], *QuickEd* [45], and *WFA2-lib* [44].

For the *Edlib* library, we use the C++ API, and we compute the edit distance and the alignment's end position (without trace-back), as well as the start and end positions (with trace-back) using the options *EDLIB_TASK_DISTANCE* and *EDLIB_TASK_LOC*, respectively. Additionally, we utilize the three available modes to analyze the impact on performance: (i) the default global *Needleman–Wunsch (NW)*, (ii) *Infix (HW)*, and (iii) *Prefix (SHW)* by the options: *EDLIB_MODE_NW*, *EDLIB_MODE_HW*, and *EDLIB_MODE_SHW*, respectively. Although *Edlib* uses a band, we evaluated it with both the banded and non-banded versions to show the performance improvements of *SeqMatcher*

Similarly, for *QuickEd*, *BSAlign* and *WFA2-lib*, we utilize their C API and execute them in both banded and unbanded modes.

Finally, in the case of *SeqAn*, we leverage the test suite provided by the library developers [57]. We employ the AVX-512 implementation of *SeqAn* with a score size of 32 bits. Our evaluation utilizes the *align_bench_par* and *align_bench_par_trace* tools (which perform alignments in parallel using OpenMP) to estimate the alignment without trace-back and with trace-back, respectively.

Results variability To mitigate the variability of the results, we perform 15 executions for each dataset and algorithm implementation and calculate the average of all runs.

Table 2 Memory footprint of datasets encoded in ASCII and our two-bit proposal

Query set		Target set		ASCII (GB)	2-bits (GB)
Number	Length (bp)	Number	Length (bp)		
1,000,000	300	1,000,000	320	0.58	0,14
1,000,000	300	1,000,000	360	0.61	0.157
1,000,000	512	1,000,000	620	1.05	0.26
1,000,000	512	1,000,000	660	1,09	0.27
1,000,000	5,000	1,000,000	10,000	13.97	3.49
1,000,000	5,000	1,000,000	12,000	15.83	3.96
1,000,000	5,000	1,000,000	15,000	18.63	4.66

Table 3 Data packing time for each dataset

Target set			Query set		
Number	Length (bp)	Time (sec)	Number	Length (bp)	Time (sec)
1,000,000	300	0.362	1,000,000	320	0.356
1,000,000	300	0.362	1,000,000	360	0.356
1,000,000	512	0.361	1,000,000	620	0.369
1,000,000	512	0.363	1,000,000	660	0.368
1,000,000	5,000	0.497	1,000,000	10,000	0.455
1,000,000	5,000	0.523	1,000,000	12,000	0.471
1,000,000	5,000	0.531	1,000,000	15,000	0.476

5.1.3 Resources and configuration

Our evaluation is performed in an Intel Xeon Gold 6230R server featuring AVX-512 vectorization extensions. It is equipped with 2 processors, each featuring 26 cores, resulting in a total of 52 cores. Hyperthreading is disabled. The processors operate at a base frequency of 2.10 GHz per core, with the ability to reach up to 4 GHz. Additionally, the nodes are equipped with 192 GB of RAM. Each node also has access to 950 GB of local scratch storage, providing ample space for temporary data storage and processing.

Compiling SeqMatcher We utilize Intel `icpx` compiler version 2024.1 for the compilation of *SeqMatcher*. We incorporate the -mavx512f flag to enable the use of AVX-512 instructions and the -mfma flag to activate Fused Multiply-Add (FMA) instructions.

Compiling Baseline Reference We compile the scalar implementation and *Edlib* code using the Intel `icpx` compiler version 2024.1. However, for the *SeqAn* test suite, we used GCC `g++` version 13.2.0, as recommended by the developers for AVX-512 architecture instructions. Similarly, we compiled *BSAlign*, *WFA2-lib*, and *QuickEd* using GCC `gcc` version 13.2.0.

5.2 Data packing results

We present here the outcomes of the performance evaluation and analysis for our novel approach of packing sequences on the CPU using AVX-512. It allows us to (i) reduce the memory footprint and (ii) enhance the number of nucleotides that can be simultaneously processed.

Table 2 shows an estimation of the memory footprints resulting from encoding target and query sequences using ASCII and our two-bit proposal, specifically for the datasets described in Table 1 in Sect. 5.1.1 of this article.

Our method leverages the power of AVX-512 instructions to significantly reduce data packing time. Table 3 presents the time that *SeqMatcher* employs for packing each sequence on the CPU using AVX-512 across different datasets. Since only a few seconds are required to pack the sequences, the data packing time is completely eliminated.

5.3 Performance results

5.3.1 Single threading results

In this section, we compare *SeqMatcher* with our scalar implementation (in both non-banded and banded versions) of Myer's algorithm to identify the significant speed improvements resulting from the AVX-512 vectorization. Throughout this section, we will refer to this implementation as the scalar version.

Moreover, we conduct a comparative analysis with state-of-the-art implementations of Myers' algorithm, namely: (i)*Edlib* and *SeqAn*, as well as other aligners such as *WFA2-lib*, *BSAlign*, and *QuickEd*. We evaluate the read alignment throughput in terms of PCPS in both, unbanded and banded versions of

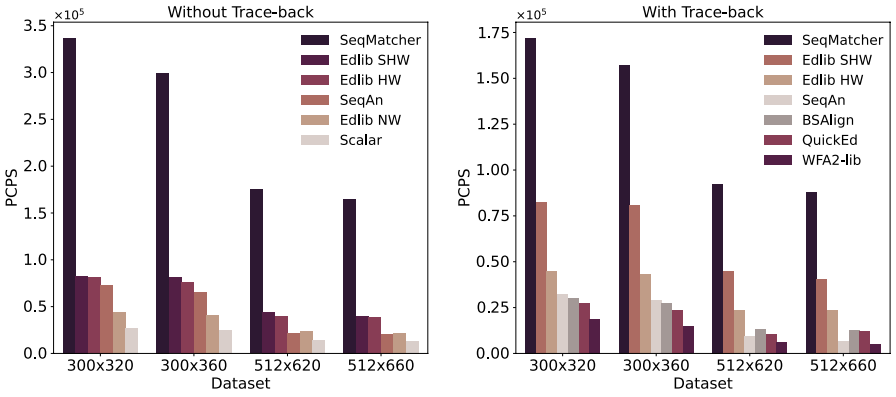


Fig. 6 Throughput (PCPS) comparison of the unbanded version of *SeqMatcher*. On the left, we compare *SeqMatcher's* performance without trace-back computation against the scalar implementation, *SeqAn*, and *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix). On the right, we compare *SeqMatcher* with trace-back computation to *SeqAn*, *Edlib* across the SHW and HW alignment modes, and the aligners: *WFA2-lib*, *BSAlign*, and *QuickEd*. *The x-axis labels represent the lengths of the query and target sequences, respectively

SeqMatcher. Since this evaluation uses a single thread, we only employ the read sets of 5,000x10,000, 5,000x12,000, and 5,000x15,000 for the banded evaluation.

For the evaluation of the unbanded version of *SeqMatcher*, we compare it with the unbanded scalar version. Additionally, we compare it with *Edlib* (which uses a banded approach by default), *SeqAn*, *QuickEd*, *WFA2-lib*, and *BSalign*. For *Edlib*, we compute alignments both with and without trace-back using infix (HW) and the prefix (SHW) methods, while for the global (NW) method, we compute alignments without trace-back only. For *QuickEd*, *WFA2-lib*, and *BSalign*, we compute alignments with trace-back using their default modes. Figure 6 shows the number of PCPS for the estimation of the alignment without trace-back (left) and with trace-back (right). In general, *SeqMatcher* demonstrates the best overall performance.

In this analysis, *SeqMatcher* shows a performance improvement of 12.32x compared to the scalar version. Myers' fast bit-vector algorithm requires a $O(m)$ to precompute the *peq* vector for one query sequence, resulting in an overall runtime of $O(m + n\lceil m/w \rceil)$, where w represents the word length. Thus, we expect an ideal acceleration close to eightfold, which is overcome by the two-bit mapping, vectorization capabilities, and the enhanced optimizations of *SeqMatcher*.

Moreover, despite *Edlib* leveraging Ukkonen's banded algorithm to constrain the search space and the parallelization capabilities of Myers' algorithm, *SeqMatcher* achieves speedups of up to 7.59x, 4.35x, and 4.07x over *Edlib* in NW, HW, and SHW modes, respectively, for alignments without trace-back. For alignments with trace-back, the performance improves but decreases slightly, still maintaining accelerations of up to 3.88x and 2.16x in HW and SHW modes, respectively. The strategies for computing trace-back differ between *SeqMatcher* and *Edlib*. While *SeqMatcher* applies Myers' algorithm again, *Edlib* uses Hirschberg's algorithm.

In addition, *SeqMatcher* achieves speedups of up to 7.89x over *SeqAn* for alignments without trace-back, and up to 13.47x, 17.56x, 8.81x, and 7.02x over *SeqAn*, *WFA2-lib*, *QuickEd*, and *BSAlign*, respectively, for alignments with trace-back.

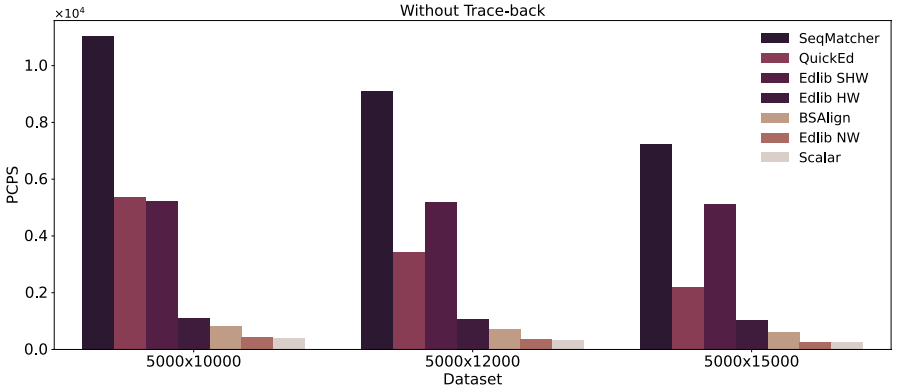


Fig. 7 Throughput (PCPS) comparison of the banded version of *SeqMatcher* with the scalar implementation, *BSAlign, QuickEd*, and *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix). *The x-axis labels represent the lengths of the query and target sequences, respectively

In our analysis of the poor performance of *WFA2-lib*, we found that its performance improves significantly when using a read set of the same length but with highly similar sequences. For example, with a read set of 300x320 where the query aligns perfectly with the target, *WFA2-lib* outperforms *SeqMatcher*, achieving a 2.21-fold increase in speed with short reads. However, as the read set length increases, its performance drops significantly, and this decline becomes more notable when both the sequence length and similarity are increased. Thus, we find that the performance of *WFA2-lib* is highly sensitive to both sequence length and, particularly, to the number of discrepancies.

This is because the use of bit-vectors in *Myers' algorithm* results in a time complexity of $O(m + n\lceil m/w \rceil)$, and $O(n\lceil m/w \rceil)$ for *SeqMatcher*, which does not precompute the *peq* bit-vector, where w is the machine word size. Also, the degree of similarity between sequences does not affect the performance of *SeqMatcher*. In contrast, the complexity of the WFA2-lib implementation is $O(ms + s^2)$, where s is the optimal alignment score. Thus, although *WFA2-lib* benefits from automatic vectorization, the increasing of divergent regions directly impact its performance, which also increase with increasing sequence size. On the contrary, *SeqMatcher* achieves linear time complexity when the sequence length is less than or equal to 512 nucleotides, enabling greater scalability as the sequence length increases.

On the other hand, *QuickEd* and *BSAlign* exhibit similar performances, both lower than *Edlib*. Although the length and number of discrepancies affect *QuickEd*'s performance, the efficient implementation of the bound-and-align paradigm enables better scalability for longer and noisier sequences compared to *WFA2-lib*. Finally, *SeqAn* exhibits the poorest performance for alignments without trace-back.

For the evaluation of the banded version, we compare *SeqMatcher* with the banded scalar version, as well as with *Edlib*, *BSAlign*, and *QuickEd*. For *Edlib*, similar to unbanded version, we employ different alignment methods: global (NW), infix (HW), and prefix (SHW). For *BSAlign* and *QuickEd*, we use their banded modes. Figure 7 shows the results for the estimation of the alignment without trace-back. Since the memory consumption of *QuickEd* with long reads exceeds the capacity of our system, we reduce the number of comparisons from 1,000,000 to 10,000. Furthermore, we exclude *WFA2-lib* from this benchmark due to its excessive execution time.

In a similar fashion to the unbanded evaluation, *SeqMatcher* demonstrates superior performance, achieving a speedup of up to 26.70x compared to the scalar version. As in the unbanded evaluation, we expected an acceleration of approximately eightfold, which is surpassed due to the optimizations implemented in *SeqMatcher*'s algorithm, and the processing of the target sequence. Similarly, *SeqMatcher* overcome *Edlib* with a acceleration of 29.21x and 10.03x in NW and HW modes. This acceleration decreases to 2.11x with respect to *Edlib* in SHW mode. This difference is due to *Edlib* SHW aligning only a specific part (the prefix), which has a more noticeable impact on performance with long target sequences. Therefore, since the processing of the target sequence is sequential, the importance of efficient reference management becomes especially evident with very long references. *SeqMatcher* searches the query sequence across the entire target sequence, similar to both *Edlib*

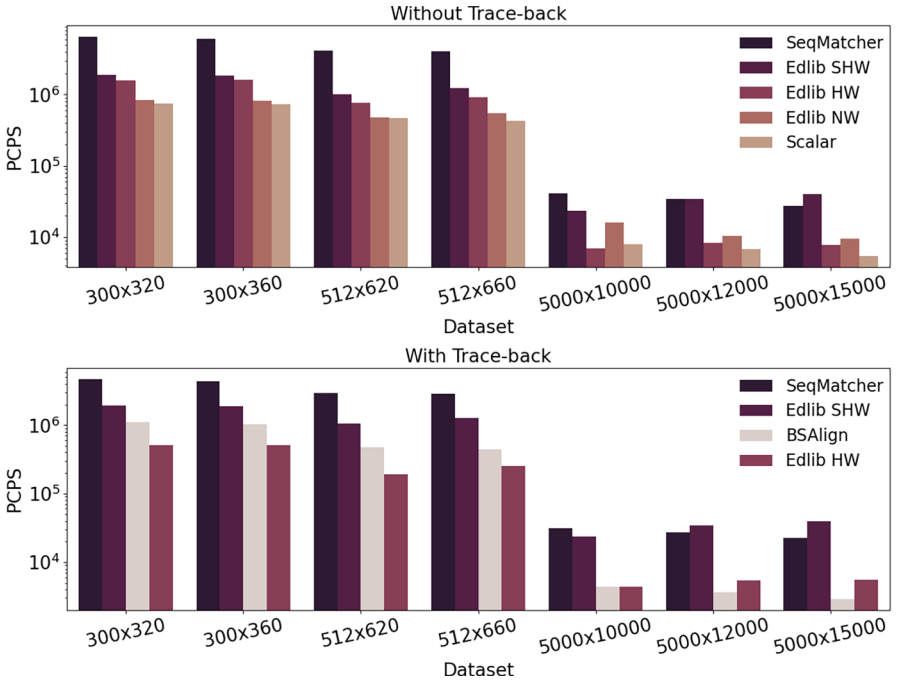


Fig. 8 Throughput (PCPS) comparison of the unbanded version of *SeqMatcher* using both short and long reads with 52 threads. Above, we compare *SeqMatcher's* performance without trace-back computation against the scalar implementation, and *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix). Below, we compare *SeqMatcher* with trace-back computation to *Edlib* across the SHW and HW alignment modes, and *BSAlign*.*The x-axis labels represent the lengths of the query and target sequences, respectively. The y-axis is displayed on a logarithmic scale

in HW mode and the scalar version, but with AVX-512 optimization on the reference, significantly enhancing its performance.

In addition, when benchmarked against *BSAlign*, *SeqMatcher* achieves speedups of up to 9.12x and delivers similar performance improvements relative to *QuickEd* as it does to *Edlib* in SHW mode.

Finally, we analyze the accuracy of both the banded and unbanded versions of *SeqMatcher* across all datasets in determining the final position. We find that, in our dataset, the accuracy of the banded version is reduced by 3–10% compared to the unbanded version. In this regard, the use of the banded approach should be carefully considered based on the specific application.

5.3.2 Multi-threading results

In this section, we conduct a comprehensive evaluation on a multi-core system with 52 cores using the fork-join model within the OpenMP API to efficiently distribute loop iterations among available threads.

Fig. 9 Speedup comparison across datasets as the number of threads increases. On the left, we compare ▶ the speedup of *SeqMatcher* without trace-back computation against the scalar implementation, *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix), and *SeqAn*, relative to the single-thread execution of *SeqMatcher*. On the right, we compare the speedup of *SeqMatcher* with trace-back computation against *SeqAn*, *Edlib* in the SHW and HW alignment modes, as well as *WFA2-lib*, *BSAlign*, and *QuickEd*, all relative to the single-thread execution of *SeqMatcher*

We evaluate the performance of *SeqMatcher* across various datasets in both banded and unbanded configurations within a multi-core environment.

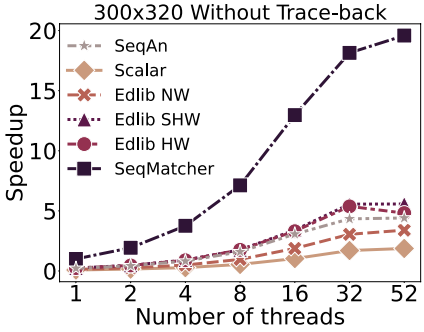
For the unbanded version, we evaluate the speedup across varying numbers of cores and assess the performance when using 52 cores to align both short and long reads, with and without trace-back.

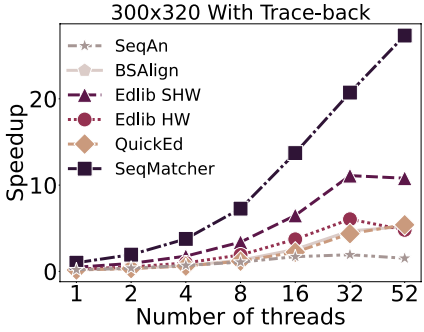
In the 52-core performance analysis, we compare *SeqMatcher* against the scalar implementation and *Edlib*. We exclude *QuickEd* from this analysis of short and long reads due to its excessive memory consumption with long reads, as discussed in the previous section (5.3.1). Additionally, we exclude *WFA2-lib* and *SeqAn* from the analysis due to their extended execution times.

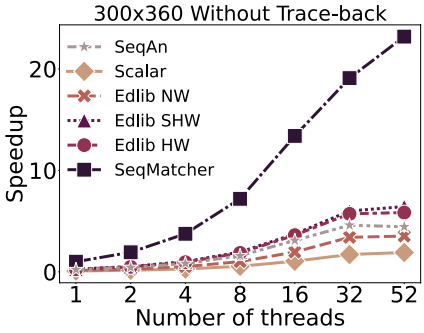
Figure 8 shows the number of PCPS for the alignment estimation without trace-back (above) and with trace-back (below). Similar to the results obtained using a single thread, *SeqMatcher* demonstrates superior performance in aligning short reads. For long-read alignments, while *Edlib* in HW mode underperforms compared to *SeqMatcher*, *Edlib* in SHW mode yields better results. This is due to two main reasons. First, *SeqMatcher* computes the entire matrix, whereas *Edlib* leverages Ukkonen's banded algorithm to narrow the search space. Additionally, as observed in the unbanded analysis, *Edlib* in HW mode searches the entire target sequence, while in SHW mode, it restricts the alignment by treating the query as a prefix of the target. Nevertheless, *SeqMatcher*'s 512-nucleotide sequence chunk-splitting strategy demonstrates better performance than *Edlib* in HW mode. Moreover, similar to the single-thread analysis, *SeqMatcher* outperforms *BSAlign* even when no banded strategy is applied.

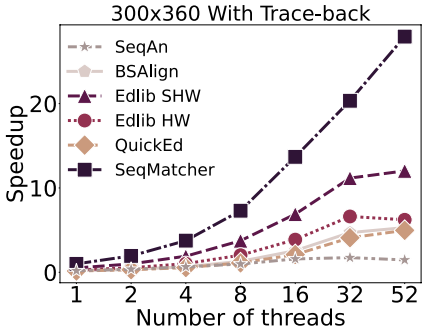
In the speedup analysis, we calculate the speedup of the different implementations as the number of threads increases. Since the execution time of some tools in the unbanded version with long reads is excessively high, we do not include speedup results for the longest reads in this analysis. Thus, Fig. 9 shows the speedup of the different implementations as the number of threads increases with respect to the execution with a single-thread with *SeqMatcher*, for both alignment without trace-back (left) and with trace-back (right).

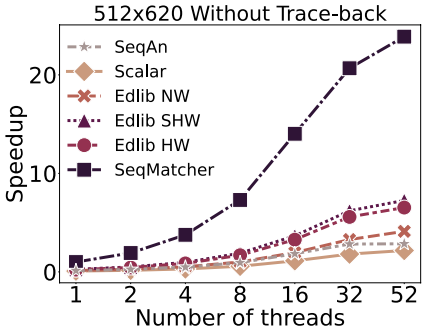
We observe that the speedup remains relatively consistent across all thread counts, with only a slight decrease. Compared to the scalar version, the speedup is almost identical (12.31x with a single thread vs. 12.14x with 52 threads). Moreover, for alignment without trace-back, the speedup of *SeqMatcher* decreases only slightly to 6.57x, 4.0467x, and 3.605x with 52 cores compared to *Edlib* in NW, HW, and SHW modes, respectively. This reduction is expected, as using AVX-512 instructions increases power consumption (see Sect. 5.5). As a result, the processor lowers its clock speed and/or voltage to manage the higher energy usage and

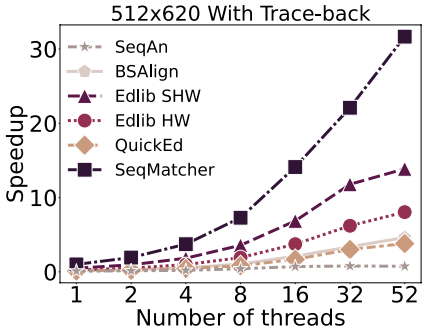

300x320 Without Trace-back
SeqAn
Scalar
Edlib NW
Edlib SHW
Edlib HW
SeqMatcher
Speedup
Number of threads

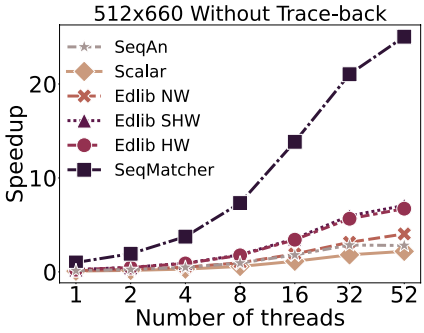

300x320 With Trace-back
SeqAn
BSAlign
Edlib SHW
Edlib HW
QuickEd
SeqMatcher
Speedup
Number of threads

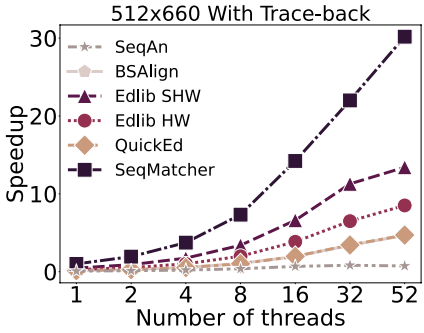

300x360 Without Trace-back
SeqAn
Scalar
Edlib NW
Edlib SHW
Edlib HW
SeqMatcher
Speedup
Number of threads


300x360 With Trace-back
SeqAn
BSAlign
Edlib SHW
Edlib HW
QuickEd
SeqMatcher
Speedup
Number of threads


512x620 Without Trace-back
SeqAn
Scalar
Edlib NW
Edlib SHW
Edlib HW
SeqMatcher
Speedup
Number of threads


512x620 With Trace-back
SeqAn
BSAlign
Edlib SHW
Edlib HW
QuickEd
SeqMatcher
Speedup
Number of threads


512x660 Without Trace-back
SeqAn
Scalar
Edlib NW
Edlib SHW
Edlib HW
SeqMatcher
Speedup
Number of threads


512x660 With Trace-back
SeqAn
BSAlign
Edlib SHW
Edlib HW
QuickEd
SeqMatcher
Speedup
Number of threads

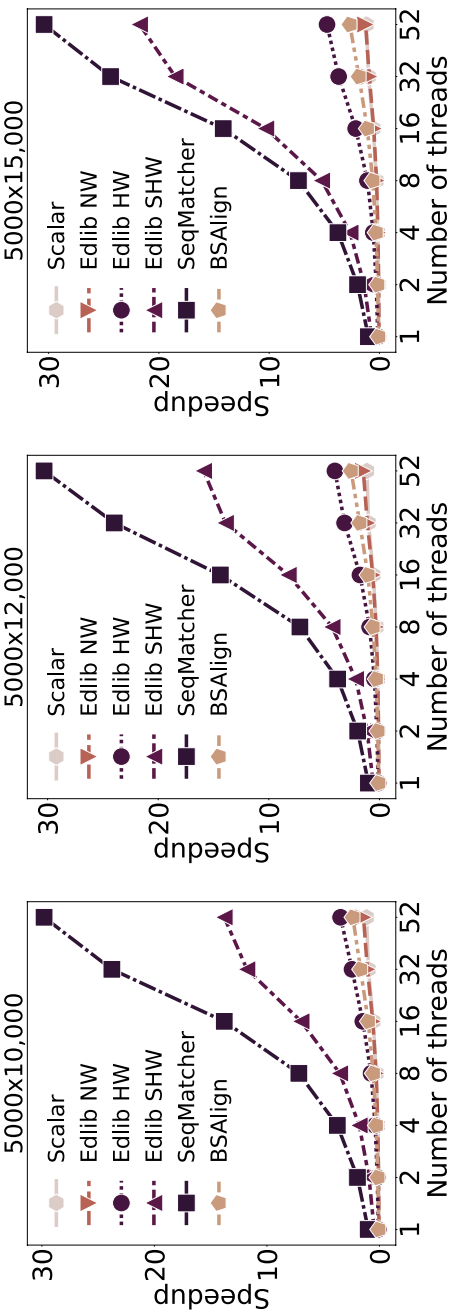


Fig. 10 Comparison of speedup across datasets as the number of threads increases for *SeqMatcher* against the scalar implementation, *BSAlign*, and *Edlib* in three alignment modes: NW (global), SHW (prefix), and HW (infix), relative to the execution time with a single thread in *SeqMatcher*

Table 4 Memory consumption of pairwise alignment algorithms for unbanded approach

Algorithm	s320	s360	s620	s660	s10000	s12000	s15000
SeqMatcher	0.943	0.989	1.422	1.466	16.608	17.995	20.095
Baseline	8.212	8.242	8.720	8.766	18.781	20.555	20.555
Edlib NW	8.252	8.275	8.750	8.805	18.815	20.590	20.588
Edlib HW	8.346	8.372	8.887	8.903	18.863	20.637	20.636
Edlib SHW	8.350	8.373	8.887	8.903	18.862	20.634	20.635
WFA2	0.739	0.792	1.236	1.284	22.384	24.667	28.725
BSAlign	0.634	0.680	1.114	1.146	14.759	16.664	19.728

Dataset names have been abbreviated with compact labels: s320 corresponds to "300 × 320," s360 to "300 × 360," s620 to "512 × 620," s660 to "512 × 660," s10000 to "5000 × 10000," s12000 to "5000 × 12000," and s15000 to "5000 × 15000"

avoid overheating. In the case of *SeqAn*, the speedup increases to 8.91x, which is expected since *SeqAn* also utilizes SIMD instructions.

In the case of alignment with trace-back, *SeqMatcher* shows similar results, with a slight decrease to 6.87x and 8.32x with 52 cores compared to *BSAlign* and *QuickEd*, respectively.

For the banded version, we evaluate the speedup across varying numbers of cores. In this analysis, we compare *SeqMatcher* against the scalar implementation, as well as the libraries *Edlib*, and *BSAlign*. Figure 10 presents the speedup for the banded version, using the single-thread execution of *SeqMatcher* as a reference for alignment without trace-back. As in the unbanded evaluation, the speedup remains relatively consistent across all thread counts. The acceleration compared to the scalar version is almost identical (26.70x with a single thread vs. 26.76x with 52 threads). In comparison with *BSAlign* and *Edlib*, the speedup only decreases slightly, reaching 12.60x with respect to *BSAlign*, and 24.22x and 8.62x with respect to *Edlib* in NW and HW modes, respectively.

5.4 Memory footprint

Table 4 presents the peak RAM usage of the evaluated algorithms. Since the RAM peak is slightly higher between the unbanded with respect to banded versions for the long reads evaluated, we only report the RAM peak for the alignment computation with trace-back using the unbanded approach. The exception is made for the scalar version and *Edlib* in NW mode, where only the alignment without trace-back is considered.

These results experimentally confirm the significantly reduced memory footprint of the AVX512-accelerated implementation of Myers' algorithm compared to both the scalar implementation and *Edlib* when aligning the read sets *s320*, *s360*, *s620*, and *s660*. This improvement is due to *SeqMatcher*'s use of AVX-512 vectorization, which processes intermediate vectors (e.g., *VP*, *VN*, *HP*, *HN*) directly in 512-bit SIMD registers, avoiding the need to store them in RAM as required by the scalar

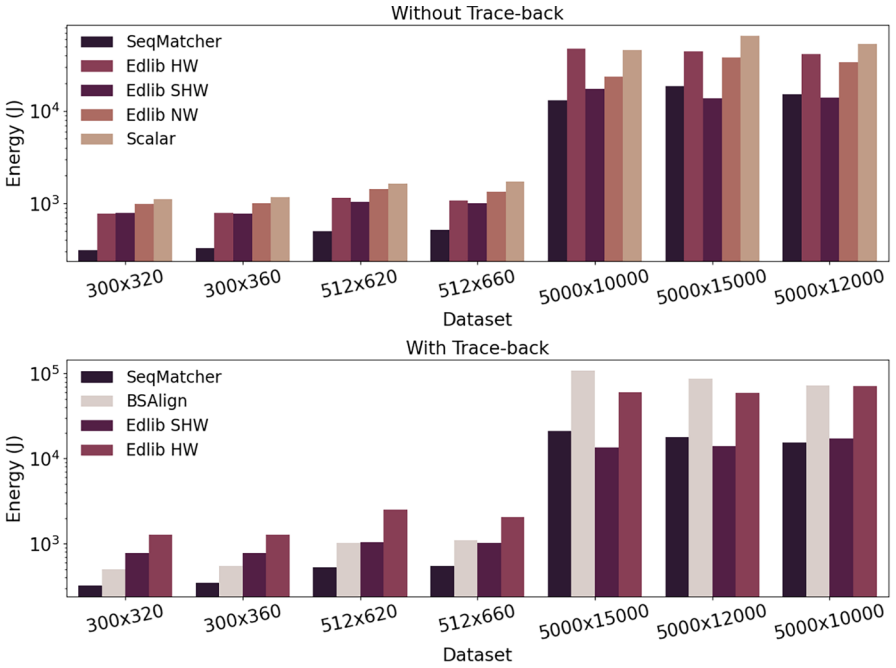


Fig. 11 Processor energy usage of the unbanded version of *SeqMatcher* with both short and long reads using 52 threads. Above, we compare the energy usage of *SeqMatcher* without trace-back computation against the scalar implementation and *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix). Below, we compare *SeqMatcher* with trace-back computation to *Edlib* across the SHW and HW alignment modes, as well as to *BSAlign*. *The x-axis labels represent the lengths of the query and target sequences, respectively. The y-axis in the figure is presented on a logarithmic scale

version and *Edlib*. However, as sequence lengths increase, the difference in peak RAM usage narrows (20.095 MB vs. 20.555 MB for the 5,000x15,000 read set).

In contrast, *WFA2-lib* demonstrates a smaller memory footprint than *SeqMatcher* for short reads. However, as the sequence length increases, the alignment score s grows, resulting in increased memory usage due to *WFA2-lib*'s $O(s^2)$ space complexity.

Finally, *BSAlign* achieves the best performance, with a memory footprint slightly smaller than that of *SeqMatcher*.

5.5 Energy consumption results

We assess the energy footprint of *SeqMatcher* and compare it with the scalar version and for the four evaluated software reference tools: *Edlib*, and *BSAlign*. We exclude *SeqAn* from direct evaluation because we evaluate the performance of *SeqAn* on an existing benchmark. Similar to the previous sections, where we used long reads, we exclude *WFA2-lib* and *QuickEd* from our analysis.

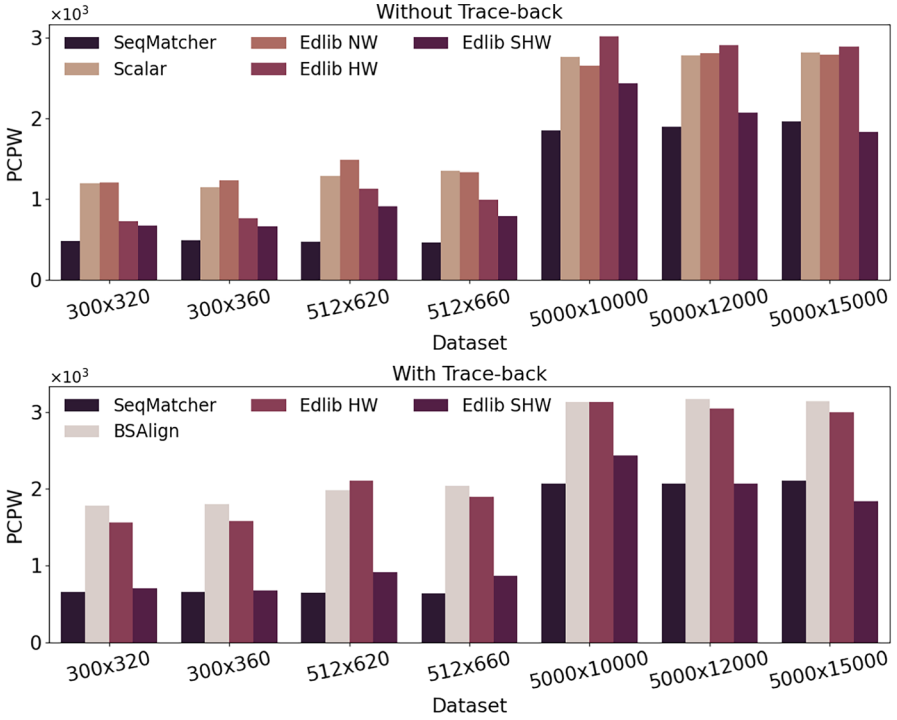


Fig. 12 Power usage of the unbanded version of *SeqMatcher* with both short and long reads using 52 threads. Above we compare the power consumption of *SeqMatcher* without trace-back computation against the scalar implementation and *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix). Below, we compare the power usage of *SeqMatcher* with trace-back computation to *Edlib* across the SHW and HW alignment modes, as well as to *BSAlign*. *The x-axis labels represent the lengths of the query and target sequences, respectively

We evaluate the energy consumption of the processor package and calculate the total energy in joules (J) and power in watts (W), for both banded and unbanded approaches.

For the unbanded version, we calculate the energy consumption for both the alignment without trace-back and the alignment with trace-back. Figures 11 and 12 display the total energy consumption and power for the alignment without trace-back (above) and with trace-back (below), respectively.

We observe that *SeqMatcher* achieves a significant reduction in energy consumption, both for alignments with and without trace-back. Specifically, for the alignment without trace-back, *SeqMatcher* reduces energy usage by up to 2.59x, 2.17x, 2.65x, and 1.53x compared to the scalar implementation and *Edlib* in NW, HW, and SHW modes, respectively. Furthermore, for the alignment with trace-back, *SeqMatcher* outperforms *BSAlign*, and *Edlib* in HW and SHW modes, achieving reductions of up to 4.13x, 3.74x, and 1.41x, respectively.

However, while the overall energy consumption of *SeqMatcher* is lower compared to other programs-likely due to its significantly reduced computation time-in

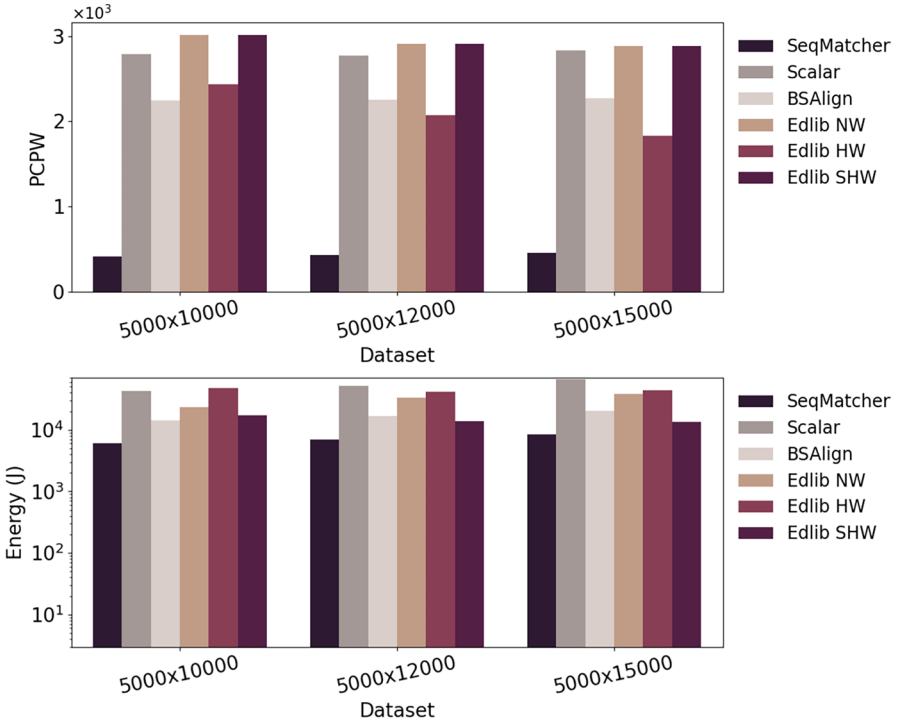


Fig. 13 Comparison of energy usage and power consumption for the banded version of *SeqMatcher* using 52 threads, against the scalar implementation, *BSAlign*, and *Edlib* across three alignment modes: NW (global), SHW (prefix), and HW (infix). *The x-axis labels represent the lengths of the query and target sequences, respectively. The y-axis above is displayed on a logarithmic scale

terms of energy consumption per unit of time, *SeqMatcher* exhibits higher power usage. This is primarily attributed to the use of AVX-512 instructions, which involve more complex operations and result in higher power demands during execution, despite the shorter runtime.

Similarly, for the *banded* version, we evaluate the energy consumption of *SeqMatcher* against the scalar implementation, *BSAlign*, and *Edlib*. Figure 13 depicts the power consumption (top) and total energy consumption (bottom). As observed, *SeqMatcher* achieves energy savings of 1.37x, 3.77x, 6.78x, and 1.83x compared to *BSAlign* and *Edlib* in NW, HW, and SHW modes, respectively. However, as in the unbanded analysis, *SeqMatcher* exhibits the highest power consumption.

6 Conclusions and future work

De novo genome assembly represents a crucial step in any bioinformatics workflow. Furthermore, with the continuous reduction in sequencing costs and the advent of new long-read sequencing technologies, its widespread adoption has become

prevalent within the scientific community. However, pairwise alignment algorithms still face crucial challenges in reducing execution time and memory consumption.

In this work, we present *SeqMatcher*, designed to efficiently handle large-scale genome pairwise sequence alignment. Thus, *SeqMatcher* provides a fast bit-vector algorithm to compute the Levenshtein distance between pairs of sequences using AVX-512 intrinsics.

We evaluate *SeqMatcher* across multiple datasets and compare it to its scalar version, other implementations of Myer's algorithm, and additional novel aligners. Our findings reveal that, despite an increase in power consumption, *SeqMatcher* achieves superior performance in aligning both long- and short-read datasets compared to state-of-the-art methods.

However, a key limitation of our work is the lack of certain intrinsics for performing addition, rotation, and shift operations on full registers, rather than on 16-, 32-, or 64-bit packets. This has necessitated the implementation of these operations using multiple intrinsics, which introduces additional complexity and may reduce the overall efficiency of the system, particularly when processing large datasets. As future work, we will explore the inclusion of new vector instructions for accelerating full add, shift, and rotate operations in the open RISC-V standard.

Additionally, by using a 2-bit mapping, the 'N' values are ignored, which could impact performance. We plan to study the effect of this limitation in future work.

Acknowledgements The authors thank the computer resources, technical expertise, and assistance provided by the Supercomputing and Bioinnovation Center (SCBI) of the University of Malaga. Also, this work has been partially supported by the Spanish MINECO PID2019-105396RB-I00 and PID2022-136575OB-I00 projects.

Author contributions E.E conceptualized the study, wrote the main manuscript text, conducted the experiments, analyzed the data, and prepared Figures. R.Q wrote the main manuscript text, provided critical insights, and assisted in the manuscript revision. R.L provided critical insights and reviewed the final manuscript. O.P provided critical insights and reviewed the final manuscript.

Funding Funding for open access publishing: Universidad Málaga/CBUA.

Declarations

Conflict of interest Not applicable

Ethical approval Not applicable

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Ginsburg GS, Willard HF (2009) Genomic and personalized medicine: foundations and applications. Transl Res 154(6):277–287
2. Chin L, Andersen JN, Futreal PA (2011) Cancer genomics: from discovery science to personalized medicine. Nat Med 17(3):297–303
3. Flores M, Glusman G, Brogaard K, Price ND, Hood L (2013) P4 medicine: how systems medicine will transform the healthcare sector and society. Pers Med 10(6):565–576
4. Ashley EA (2016) Towards precision medicine. Nat Rev Genet 17(9):507–522
5. Wenger AM, Peluso P, Rowell WJ, Chang P-C, Hall RJ, Concepcion GT, Ebler J, Fungtammasan A, Kolesnikov A, Olson ND et al (2019) Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. Nat Biotechnol 37(10):1155–1162
6. Espinosa Garcia E, Arroyo Varela M, Larrosa Jimenez R, Gomez-Maldonado J, Cobo Dols MA, Claros MG, Bautista Moreno R (2023) Construction of miRNA-mRNA networks for the identification of lung cancer biomarkers in liquid biopsies. Clin Transl Oncol 25(3):643–652
7. Prado-Martinez J, Sudmant PH, Kidd JM, Li H, Kelley JL, Lorente-Galdos B, Veeramah KR, Woerner AE, O'connor TD, Santpere G et al (2013) Great ape genetic diversity and population history. Nature 499(7459):471–475
8. Ellegren H (2014) Genome sequencing and population genomics in non-model organisms. Trends Ecol Evolut 29(1):51–63
9. Alvarez-Cubero MJ, Saiz M, Martínez-García B, Sayalero SM, Entrala C, Lorente JA, Martinez-Gonzalez LJ (2017) Next generation sequencing: an application in forensic sciences? Ann Hum Biol 44(7):581–592
10. Børsting C, Morling N (2015) Next generation sequencing and its applications in forensic genetics. Forensic Sci Int Genet 18:78–89
11. Yang Y, Xie B, Yan J (2014) Application of next-generation sequencing technology in forensic science. Genom Proteom Bioinform 12(5):190–197
12. Hu T, Chitnis N, Monos D, Dinh A (2021) Next-generation sequencing technologies: an overview. Hum Immunol 82(11):801–811
13. Biosciences P. Pacific Biosciences. https://www.pacb.com/
14. Rhoads A, Au KF (2015) PacBio sequencing and its applications. Genom Proteom Bioinform 13(5):278–289
15. Korlach J, Gedman G, Kingan SB, Chin C-S, Howard JT, Audet J-N, Cantin L, Jarvis ED (2017) De novo PacBio long-read and phased avian genome assemblies correct and add to reference genes generated with intermediate and short reads. Gigascience 6(10):085
16. Espinosa E, Bautista R, Larrosa R, Plata O (2024) Advancements in long-read genome sequencing technologies and algorithms. Genomics 116:110842
17. Nanopore O. Oxford Nanopore. https://nanoporetech.com/
18. Sereika M, Kirkegaard RH, Karst SM, Michaelsen TY, Sørensen EA, Wollenberg RD, Albertsen M (2022) Oxford Nanopore R10. 4 long-read sequencing enables the generation of near-finished bacterial genomes from pure cultures and metagenomes without short-read or reference polishing. Nature methods 19(7):823–826
19. Jain M, Koren S, Miga KH, Quick J, Rand AC, Sasani TA, Tyson JR, Beggs AD, Dilthey AT, Fiddes IT et al (2018) Nanopore sequencing and assembly of a human genome with ultra-long reads. Nat Biotechnol 36(4):338–345
20. Illumina: Illumina. https://www.illumina.com/
21. Stoler N, Nekrutenko A (2021) Sequencing error profiles of Illumina sequencing instruments. NAR Genom Bioinform 3(1):019
22. PacBio: HIFI SEQUENCING. https://www.pacb.com/technology/hifi-sequencing/
23. Li Z, Chen Y, Mu D, Yuan J, Shi Y, Zhang H, Gan J, Li N, Hu X, Liu B et al (2012) Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph. Brief Funct Genomics 11(1):25–37
24. Rizzi R, Beretta S, Patterson M, Pirola Y, Previtali M, Della Vedova G, Bonizzoni P (2019) Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era. Quantitat Biol 7:278–292

25. Espinosa E, Bautista R, Fernandez I, Larrosa R, Zapata EL, Plata O (2023) Comparing assembly strategies for third-generation sequencing technologies across different genomes. Genomics 115:110700
26. Levenshtein VI et al (1966) Binary codes capable of correcting deletions, insertions, and reversals. Sov Phys Dokl 10:707–710
27. Needleman SB, Wunsch CD (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol 48(3):443–453
28. Smith TF, Waterman MS et al (1981) Identification of common molecular subsequences. J Mol Biol 147(1):195–197
29. Gotoh O (1982) An improved algorithm for matching biological sequences. J Mol Biol 162(3):705–708
30. Hirschberg DS (1975) A linear space algorithm for computing maximal common subsequences. Commun ACM 18(6):341–343
31. Myers G (1999) A fast bit-vector algorithm for approximate string matching based on dynamic programming. J ACM (JACM) 46(3):395–415
32. Hyyrö H (2003) A bit-vector algorithm for computing Levenshtein and Damerau edit distances. Nord J Comput 10(1):29–39
33. Jayakumar V, Sakakibara Y (2019) Comprehensive evaluation of non-hybrid genome assembly tools for third-generation PacBio long-read sequence data. Brief Bioinform 20(3):866–876
34. Ukkonen E (1985) Finding approximate patterns in strings. J Algorithms 6(1):132–137
35. Hyyrö H, Navarro G (2002) Faster bit-parallel approximate string matching. In Combinatorial Pattern Matching: 13th Annual Symposium, CPM 2002 Fukuoka, Japan, July 3–5, 2002 Proceedings 13, pp 203–224. Springer
36. Cheng H, Jiang H, Yang J, Xu Y, Shang Y (2015) BitMapper: an efficient all-mapper based on bit-vector computing. BMC Bioinform 16(1):1–16
37. Medaka: nanoporetech/medaka: Sequence correction provided by ONT. https://github.com/nanoporetech/medaka/
38. Cleal K, Baird DM (2022) Dysgu: efficient structural variant calling using short or long reads. Nucleic Acids Res 50(9):53–53
39. Döring A, Weese D, Rausch T, Reinert K (2008) SeqAn an efficient, generic C++ library for sequence analysis. BMC Bioinform 9:1–9
40. Daily J (2016) Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. BMC Bioinform 17(1):1–11
41. Suzuki H, Kasahara M (2018) Introducing difference recurrence relations for faster semi-global alignment of long sequences. BMC Bioinform 19(1):33–47
42. Li H (2018) Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics 34(18):3094–3100
43. Shao H, Ruan J (2024) Bsalign: a library for nucleotide sequence alignment. Genom Proteom Bioinform 22:25
44. Marco-Sola S, Moure JC, Moreto M, Espinosa A (2021) Fast gap-affine pairwise alignment using the wavefront algorithm. Bioinformatics 37(4):456–463
45. Doblas M, Lostes-Cazorla O, Aguado-Puig Q, Iñiguez C, Moretó M, Marco-Sola S (2024) Quicked: high-performance exact sequence alignment based on bound-and-align. bioRxiv, 2024–09
46. Chacón A, Marco-Sola S, Espinosa A, Ribeca P, Moure JC (2014) Thread-cooperative, bit-parallel computation of levenshtein distance on GPU. In Proceedings of the 28th ACM International Conference on Supercomputing, pp 103–112
47. Ahmed N, Lévy J, Ren S, Mushtaq H, Bertels K, Al-Ars Z (2019) Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data. BMC Bioinform 20:1–20
48. Aguado-Puig Q, Doblas M, Matzoros C, Espinosa A, Moure JC, Marco-Sola S, Moreto M (2023) Wfa-gpu: gap-affine pairwise read-alignment using gpus. Bioinformatics 39(12):701
49. Cai L, Wu Q, Tang T, Zhou Z, Xu Y (2019) A design of FPGA acceleration system for myers bit-vector based on openCL. In 2019 International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS), pp. 305–312. IEEE
50. Bautista DP, Aguilera RC, Acevedo FA, Badillo IA (2021) Bit-vector-based hardware accelerator for dna alignment tools. J Circ Syst Comput 30(05):2150087
51. Castells-Rufas D, Marco-Sola S, Aguado-Puig Q, Espinosa-Morales A, Moure JC, Alvarez L, Moretó M (2021) OpenCL-based FPGA accelerator for semi-global approximate string matching

using diagonal bit-vectors. In 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pp 174–178. IEEE
52. Yee A Kogge-Stone Parallel Addition. http://www.numberworld.org/y-cruncher/internals/addition.html#ks_add. Accessed: nov 2024
53. Chimpanzee Sequencing and Analysis Consortium: The Chimpanzee Genome (Pan troglodytes). https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_028858775.2. Genome Reference Consortium, GCF_028858775.2, Pan_tro_3.0 (2024)
54. Stöcker BK, Köster J, Rahmann S (2016) SimLoRD: simulation of long read data. Bioinformatics 32(17):2704–2706
55. Šošić M, Šikić M (2017) Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. Bioinformatics 33(9):1394–1395
56. Rahn R, Budach S, Costanza P, Ehrhardt M, Hancox J, Reinert K (2018) Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading. Bioinformatics 34(20):3437–3445
57. R R.: DP Bench - A benchmark tool for SeqAn's alignment engine

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Elena Espinosa[1] · Ricardo Quislant[1] · Rafael Larrosa[1,2] · Oscar Plata[1]

✉ Elena Espinosa
elenamesga@uma.es

Ricardo Quislant
quislant@uma.es

Rafael Larrosa
rlarrosa@uma.es

Oscar Plata
oplata@uma.es

[1] Department of Computer Architecture, University of Malaga, Bulevar Louis Pasteur, 35, 29071 Málaga, Spain

[2] Supercomputing and Bioinnovation Center (SCBI), University of Malaga, C.Severo Ochoa 34, 29590 Malaga, Spain