BLEND: A Fast, Memory-Efficient, and Accurate Mechanism to Find Fuzzy Seed Matches

Can Firtina[1] Jisung Park[1] Mohammed Alser[1] Jeremie S. Kim[1] Damla Senol Cali[2]
Taha Shahroodi[3] Nika Mansouri-Ghiasi[1] Gagandeep Singh[1] Konstantinos Kanellopoulos[1]
Can Alkan[4] Onur Mutlu[1]

[1]*ETH Zurich* [2]*Bionano Genomics* [3]*TU Delft* [4]*Bilkent University*

Comparing genomic sequences is a fundamental yet costly step in most genomic analyses. It requires identifying similar genomic sequence pairs sharing a sufficient number of short subsequences, called seeds. The length and number of seed matches between sequences directly impact the sensitivity, performance, and memory footprint of today's read mappers. Existing attempts optimizing seed matches suffer from either 1) increasing the use of the costly sequence alignment step due to finding a large number of sequence pairs for alignment or 2) limited sensitivity when finding seed matches. Our goal is to efficiently and effectively identify potentially similar sequences using a small number of seed matches, while tolerating variations within each seed. To this end, we introduce BLEND*, a fast, memory-efficient, and accurate mechanism to find approximate (i.e., fuzzy) seed matches. BLEND 1) finds the minimizer k-mers and extends them to increase the length of seeds and 2) utilizes the SimHash technique to enable generating the same hash values for highly similar seeds, which allows matching fuzzy seeds with a single-look up.*

We show the benefits of BLEND when used in two important genomics applications: read overlapping and read mapping. For read overlapping, BLEND enables a 2.6×-63.5× *(on average* 19.5×*) faster and* 0.9×-9.7× *(on average* 3.6×*) more memory-efficient implementation than the state-of-the-art tool, Minimap2. We observe that BLEND finds better quality overlaps that lead to more accurate* de novo *assemblies compared to Minimap2. For read mapping, BLEND provides* 0.7×-3.7× *(on average* 1.7×*) speedup compared to Minimap2. Source code is available at* `https://github.com/CMU-SAFARI/BLEND`.

1. Introduction

High-throughput sequencing (HTS) technologies have revolutionized the field of genomics due to their ability to produce millions of nucleotide sequences at relatively low cost [58]. Although HTS technologies are key enablers of almost *all* genomics studies [2, 5, 18, 31, 38, 41], HTS technology-provided data comes with two key shortcomings. First, HTS technologies can only provide many short (e.g., one hundred up to a million bases depending on the technology [58]) sequences (i.e., *reads*) within a full genome [19] (e.g., billions of bases for the human genome). Second, HTS technologies can misinterpret signals during sequencing and thus provide reads that contain errors (i.e., sequencing errors [19]). The average frequency of sequencing errors in a read highly varies (from 0.1% up to 15%) depending on the HTS technology [22, 34, 56, 59, 66]. To address the shortcomings of HTS technologies, various computational approaches must be taken to process the reads into meaningful information. These include 1) read mapping [3, 8], 2) *de novo* assembly [12, 16, 51], 3) read classification in metagenomic studies [27, 42, 63], and 4) correcting sequencing errors [17, 32, 61].

At the core of these computational approaches, similarities between sequences must be identified to overcome the fundamental limitations of HTS technologies. However, identifying the similarities across *all* pairs of sequences is not practical due to the costly algorithms used to calculate the distance between two sequences. To practically identify similarities, it is essential to avoid calculating the distance between dissimilar sequence pairs. A common heuristic is to efficiently find matching *short* subsequences between pairs of sequences to filter out dissimilar sequence pairs that have no or a few matching short subsequences. Matching each short subsequence is mainly found with a *single lookup* using either the entire or a portion of these short subsequences known as *seeds*. The use of seeds drastically reduces the search space from all possible sequence pairs to the sequence pairs that are likely to be similar to facilitate efficient distance calculations over a large number of sequence pairs [6, 50, 54].

There are three main approaches when using seeds to find short subsequence matches between genomic sequences. The first direction is to use *k-mers* (i.e., subsequences of fixed length k) as seeds to find *exact-matching* k-mers between sequence pairs. The existing works such as minimap2 [29], MinHash [7], Winnowmap2 [23, 24], re$M_u$val [15], and CAS [65] sample the k-mers of an entire sequence to use fewer k-mers for matching. For example, minimap2 uses only the k-mers with the *minimum* hash value in a window of w consecutive k-mers, known as the *minimizer* k-mers [50]. Such a sampling approach guarantees that one k-mer is sampled in each window to provide a fixed ratio in sampling that can be tuned to increase the probability of finding matching k-mers between sequence pairs. Alternatively, MinHash uses several hash functions to generate many hash values from each k-mer. For each hash function, only the k-mer with the minimum hash value in the entire sequence is used for matching with no windowing guarantee. To increase the probability that seeds are generated from many different regions of the entire sequence, MinHash uses these many hash values. MinHash is mainly effective for matching sequences that have similar lengths since the number of hash functions are fixed for all sequences whereas it can generate too many seeds for shorter sequences when the sequence lengths vary greatly [28]. While these k-mer selection approaches reduce the overall number of seeds to use, all of these existing works find *only* exact-matching k-mers as they use hash functions with low collision rates to generate the hash values of these k-mers. The exact-matching requirement imposes challenges when determining the k-mer length. Longer k-mer lengths significantly decrease the probability of finding exact-matching k-mers between sequences due to genetic variations and sequencing errors. Short k-mer lengths (e.g., 8-21 bases) result in matching a large number of k-mers due to both the repetitive nature of most genomes and the high probability of finding the same short k-mer frequently in a long sequence of DNA letters [64]. Although k-mers are commonly used as seeds, a seed is a more general concept that can allow substitutions, insertions and deletions (indels) when matching short subsequences between sequence pairs.

The second direction is to allow substitutions by *masking* (i.e., ignoring) some characters of short subsequences and us-

ing the masked sequences as seeds. Such seeds are known as *spaced seeds* [33]. Approximate (i.e., *fuzzy*) matches of short subsequences can be found with a single lookup if spaced seeds mask *all* the mismatching characters between subsequence pairs. The tools such as ZOOM! [30] and SHRiMP2 [14] use spaced seeds to improve the sensitivity when mapping short reads (i.e., Illumina paired-end reads). S-conLSH [9, 10] uses different masking patterns to generate multiple spaced seeds from the same short subsequence for mapping long reads (e.g., PacBio CLR). There have been recent improvements in determining the masking patterns to improve the sensitivity of spaced seeds [35, 46]. However, none of these works can enable finding *arbitrary* fuzzy matches as the patterns still require exact matches at fixed positions of these matching short subsequences, which is a key limitation in improving the sensitivity with spaced seeds.

The third direction links a few selected k-mers of a sequence to use these linked k-mers as seeds such as paired-minimizers [13] and strobemers [52, 53]. These approaches can allow both substitutions and indels when matching short subsequences by ignoring the gaps between linked k-mers. For example, the strobemer technique concatenates a subset of k-mers of a sequence to generate the strobemer sequence, which is used as a seed. The gaps between these concatenated k-mers in a short subsequence are ignored when using strobemers to allow finding fuzzy matches between these short subsequences. Strobemers enable masking some characters of short subsequences without requiring a fixed pattern as in spaced k-mers (e.g., the number of masked characters may differ in each seed depending on the gap between concatenated k-mers). This makes strobemers a more sensitive approach than using spaced seeds as matching short subsequences can contain indels with varying lengths as well as substitutions. However, the nature of the hash function used in strobemers requires exact matches of *all* concatenated k-mers in a strobemer sequence to find fuzzy matches of short subsequences. Such an exact match requirement for certain k-mers of a seed introduces challenges for further improving the sensitivity of strobemers when finding fuzzy seed matches.

To our knowledge, there is no work that can *efficiently* find fuzzy matches of seeds *without* requiring 1) *exact matches* of all k-mers (i.e., any k-mer can mismatch) and 2) imposing a high performance and memory space overheads. In this work, we observe that existing works (e.g., minimap2, spaced seeds) have such a limitation mainly because they employ hash functions with low collision rates when generating the hash values of seeds. Although it is important to reduce the collision rate for assigning different hash values for dissimilar seeds for accuracy and performance reasons, the choice of hash functions also makes it unlikely for assigning the same hash value for similar seeds. Thus, seeds *must* exactly match to find matches between short subsequences with a single lookup. Mitigating such a requirement so that similar seeds can have the same hash value has a potential to further improve the performance and sensitivity of the applications that use seeds with their ability to allow substitutions and indels at any arbitrary positions between matching short subsequences. **Our goal** in this work is to enable finding *fuzzy* matches of seeds as well as exact-matching seeds between sequences (e.g., reads) with a single lookup of hash values of these seeds. To this end, we propose *BLEND*, a fast, memory-efficient, and accurate mechanism that can find fuzzy seed matches. The **key idea** in BLEND is to build a new, highly-efficient representation of seeds using the SimHash technique [11, 36]. We exploit the key characteristic of SimHash that can generate the *same* hash value for highly similar seeds. This provides us with two key benefits. First, BLEND can generate the same hash value for highly similar seeds *without* imposing exact matches of seeds, unlike existing seeding mechanisms that use hash functions with low collision rates. Second, BLEND generates seeds with a window guarantee by using minimizer k-mers to avoid high performance and memory overheads when finding fuzzy seed matches between sequence of varying lengths rather than using many hash functions without a window guarantee (e.g., MHAP). These two ideas ensure that we can *increase* the collision rate for similar seeds but still keep the low collision rate for dissimilar seeds while using fewer seeds. These two main benefits enable BLEND to efficiently find fuzzy seed matches between sequence pairs with high sensitivity.

Using erroneous (ONT and PacBio CLR), highly accurate (PacBio HiFi) and short (Illumina) reads, we experimentally show the benefits of BLEND on two important applications in genomics: 1) read overlapping and 2) read mapping by using either 1) immediately overlapping consecutive k-mers (`BLEND-I`) and 2) strobemers (`BLEND-S`) as seeds. First, read overlapping aims to find overlaps between all pairs of reads based on seed matches. These overlapping reads are mainly useful to generate an assembly of the sequenced genome [28, 47]. We compare BLEND with minimap2 and MinHash (i.e., MHAP) by finding overlapping reads. We then generate the assemblies from the overlapping reads to compare the qualities of these assemblies. Second, read mapping uses seeds to find similar portions between a reference genome and a read before performing the read alignment. Aligning a read to a reference genome shows the edit operations (i.e., match, substitution, insertion, and deletions) to make the read identical to the portion of the reference genome, which is useful for downstream analysis (e.g., variant calling [40]). We compare BLEND with minimap2, LRA [49], Winnowmap2, and S-conLSH by mapping long and paired-end short reads to their reference genomes. This paper makes the following **key contributions**:

- We introduce BLEND, the *first* mechanism that can quickly and efficiently find *fuzzy* seed matches between sequences with a single lookup.
- We evaluate two different seeding strategy with the SimHash hashing mechanism when finding fuzzy seed matches: `BLEND-I` and `BLEND-S`. We show that `BLEND-S` provides better speedup and accuracy compared to `BLEND-I` when using PacBio HiFi reads for read overlapping and read mapping. When using ONT, PacBio CLR and short reads, `BLEND-I` provides significantly better accuracy than `BLEND-S` with similar performance.
- For read overlapping, we show that BLEND provides speedup compared to minimap2 and MHAP by 2.6×-63.5× (on average 19.5×), 22.2×-6317.1× (on average 1386.8×), while reducing the memory overhead by 0.9×-9.7× (on average 3.6×), 35.9×-238.1× (on average 130.5×), respectively.
- We show that BLEND usually finds *longer* overlaps between reads while using *fewer* seed matches than other tools, which improves the performance and memory space efficiency for read overlapping.
- We find that we can construct more accurate assemblies with similar contiguity by using the the overlapping reads that BLEND finds compared to those that minimap2 finds.
- For read mapping, we show that BLEND provides speedup compared to minimap2, LRA, Winnowmap2, and S-conLSH by 0.7×-3.7× (on average 1.7×), 1.7×-26.5× (on average 12.5×), 1.8×-19.3× (on average 7.5×), 6.8×-58.1× (on average 21.8×) while maintaining a similar memory overhead by 0.5×-1.2× (on average 0.9×), 0.4×-1.2× (on average 0.8×),

0.9×-2.7× (on average 1.7×), 0.5×-5.6× (on average 1.9×), respectively.
- We show that BLEND provides a similar read mapping accuracy compared to minimap2, and Winnowmap2 usually provides the best read mapping accuracy.
- We open source our BLEND implementation as integrated into minimap2.
- We provide the open source SIMD implementation of the SimHash technique that BLEND employs.

2. Methods

We propose ***BLEND***, a mechanism that enables finding fuzzy (i.e., approximate) seed matches with a single lookup between two sets of sequences (i.e., target and query sequences), which can be used for 1) read overlapping and 2) read mapping with high accuracy, performance, and low memory overhead. To find fuzzy seed matches, BLEND introduces a new hashing mechanism that enables generating the same hash values for highly similar seeds. By combining this hashing mechanism with any seeding approach (e.g., minimizer k-mers or strobemers), BLEND can find fuzzy seed matches between target and query sequences with a single hash table lookup.

Figure 1 shows the overview of steps to find fuzzy seed matches with a single lookup in four steps. First, BLEND generates the same hash values for highly similar seeds. To this end, BLEND uses the SimHash technique [11, 36] to generate the hash values of seeds. The SimHash technique can generate the same hash value for highly similar seeds that have many k-mers in common. BLEND can apply any existing seeding mechanism for hashing with BLEND to find the seeds of *target sequences*. In this paper, we use minimizer k-mers and strobemers as seeding approaches with BLEND, and we call them `BLEND-I` and `BLEND-S`, respectively. Second, BLEND stores the resulting hash values of all seeds of target sequences as *keys* in a hash table to enable searching for these seeds with a single lookup. Querying the hash table with a hash value returns the list of fuzzy seed matches with a single lookup. Third, BLEND generates the hash values of seeds of query sequences by following the same steps for generating seeds of target sequences and their hash values. Fourth, BLEND can then query the hash table with hash values of seeds of *query sequences* to find fuzzy seed matches between the query and target sequences with a single lookup.

In Sections 2.1 - 2.3, we explain how BLEND generates the hash values of seeds of target sequences to build a hash table. In Section 2.4, we explain how BLEND queries the hash values of seeds of query sequences in the hash table to find fuzzy seed matches between target and query sequences.

2.1. Generating the Hash Values of Seeds

Our goal is enable efficient comparisons of equivalence or high similarity between seeds with a single lookup by generating the same hash value for highly similar or equivalent seeds. We aim generating such a hash value for a seed to enable mismatches in both 1) seeds and 2) short subsequences that BLEND uses to generate seeds. To enable generating the same hash value for highly similar or equivalent seeds, BLEND uses the SimHash technique [11]. The SimHash technique takes the hash values of all k-mers of a seed and performs a bitwise summation on all hash values to generate the hash value for a seed such that highly similar or equivalent seeds can have the same hash value. BLEND employs the SimHash technique mainly in three steps: 1) encoding the hash values of k-mers in a vector, 2) performing vector additions, and 3) decoding the vector to generate the hash value for a seed, as Figure 2 shows.

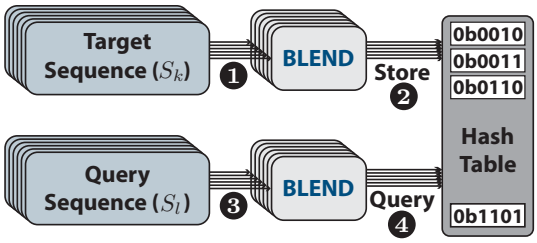

Figure 1: Overview of BLEND. ❶ To enable generating the same hash values for highly similar seeds, BLEND finds the seeds of each target sequence S_k and hashes these seeds using the SimHash technique. ❷ To enable the efficient queries of seeds with a single lookup, BLEND stores these hash values in a hash table. ❸ BLEND generates the hash values of the seeds of each query sequence S_l using the same seeding and hashing techniques as applied in ❶. ❹ To find fuzzy seed matches between target and query sequences, BLEND uses the hash values from ❸ to query the hash table that BLEND builds in ❷.

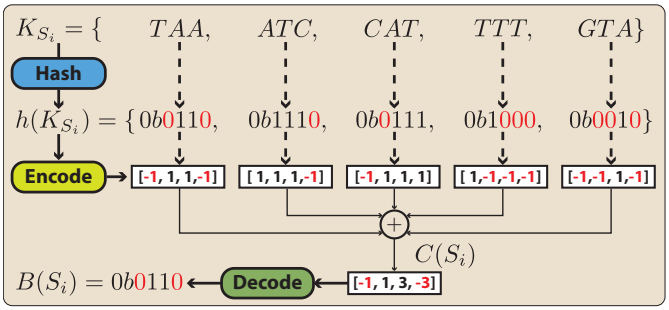

Figure 2: BLEND generates the hash value of seeds S_i. `BLEND-I` or `BLEND-S` determines the content of k-mers that a seed includes in K_{S_i}. Any hash function can be used to generate the hash values of these k-mers as we show an example set of hash values for each k-mer in $h(K_{S_i})$ using their binary representations. BLEND encodes these hash values into their vector representations. Each 0 and 1 bit in the binary representation of a hash value is encoded with values −1 (indicated by red text color) and 1 in their corresponding vectors, respectively. BLEND performs vector additions to calculate the counter vector $C(S_i)$. BLEND uses the counter vector to decode the final hash value of seed S_i denoted as $B(S_i)$.

First, the goal of vector encoding is to transform hash values of k-mers of a seed into vectors so that BLEND can efficiently perform the bitwise arithmetic operations that the SimHash technique requires. Let us define the list of all k-mers of a seed as follows: $K_{S_i} = \{k_1, k_2, \ldots, k_n\}$ where k_t is the t^{th} k-mer of seed S_i. Then, the hash values of these k-mers that we can generate using any hash function h can be demonstrated as $h(K_{S_i}) = \{h(k_1), h(k_2), \ldots, h(k_n)\}$. SimHash uses these bitwise operations because it checks 1 and 0 bits at each position of a hash value to increment or decrement values in a counter vector by one, respectively. To efficiently perform these simple bitwise increments and decrements, the vector transformation step generates a vector for each hash value in $h(K_{S_i})$ where 1 and 0 bits of a hash value are represented as +1 and −1 in the vector, respectively, as shown in Figure 2 after the encoding step. Vector transformation provides BLEND the opportunity to parallelize each bitwise operation as these operations are independent from each other, which makes it mainly suitable for certain processors such as SIMD processors and GPUs.

Second, the goal of the vector addition operation is to determine for each bit position if the *majority* of bits of all hash values in $h(K_{S_i})$ is either 1 or otherwise. The key idea in determining the majority of bits for each position is that the majority results for each bit position are likely to be the similar between two seeds, S_i and S_k, if they are highly similar to each other

such that $h(K_{S_i}) \simeq h(K_{S_k})$. To efficiently determine the majority of bits at each position, BLEND counts the number of 1 and 0 bits at a position by using the vectors it generates in the vector transformation step as shown with the addition step in Figure 2. The vector additions perform simple additions of +1 or −1 values between each element in the vector and stores the result in a counter vector of S_i, $C(S_i)$. The values in $C(S_i)$ show the majority voting of bits at every position of hash values. Since BLEND assigns −1 for 0 bits and +1 for 1 bits, the majority of bits at position t among all hash values in $h(K_{S_i})$ is either 1) 1 if the corresponding value in $C(S_i)$ is greater than 0 (i.e., $C(S_i)[t] > 0$) or 2) otherwise if $C(S_i)[t] \leq 0$.

Third, to generate the hash value of a seed S_i, BLEND uses the majority of bits that it determines by calculating the counter vector $C(S_i)$. To this end, BLEND decodes the counter vector $C(S_i)$ into a hash value in its binary form, $B(S_i)$, as shown in Figure 2 with the decoding step. The decoding operation is a simple conditional operation where each bit of the final hash value, $B(S_i)$, is determined based on the corresponding value in $C(S_i)$. BLEND assigns the bit at position t of $B(S_i)$ either 1) 1 if the value at the corresponding position of $C(S_i)$ is greater than 0 (i.e., $B(S_i)[t] = 1$ if $C(S_i)[t] > 0$), or 2) 0 if otherwise (i.e., $B(S_i)[t] = 0$ if $C(S_i)[t] \leq 0$). Thus, each bit of $B(S_i)$ shows the majority voting result of multiple hash values of k-mers of a seed in a single hash value. We use $B(S_i)$ as the hash value for a seed because highly similar seeds have many k-mers in common while any *arbitrary* k-mers may still be different, which essentially leads to the *similar* majority voting results in $C(S_i)$ and, thus, likely the same $B(S_i)$. This enables BLEND to find fuzzy seed matches with a single lookup using these hash values. We explain the details of the reasoning why two similar or same seeds may have the same hash value, $B(S_i)$, in Supplementary Section S1 and provide a real example of generating the hash values for two different seeds using different parameter settings in Supplementary Section S1.1 and Supplementary Tables S1- S8.

2.2. Determining the K-mers of Seeds

Our goal is to identify the k-mers of seeds so that BLEND can use these k-mers for generating the hash values of seeds as we explain in Section 2.1. BLEND enables generating the hash values of seeds using any seeding mechanism as it needs to identify the k-mers of these seeds and use their hash values with the SimHash technique. We provide the mechanism for identifying the k-mers for two different seeding approaches 1) minimizer k-mers and 2) strobemers, which we call `BLEND-I` and `BLEND-S`.

The goal of the first mechanism, `BLEND-I`, is to reduce the amount of seeds that BLEND uses per sequence so that BLEND can reduce the storage overhead for a sequence with minimal information loss [24]. To this end, `BLEND-I` finds all the minimizer k-mers of a target sequence and uses all of these minimizer k-mers as seeds as shown in Figure 3. These minimizer k-mers are defined as the k-mers that have the smallest hash value in a window of consecutive k-mers [29]. Minimizer k-mers can reduce the storage requirement per sequence because using these k-mers avoids storing all k-mers of a sequence and instead a single k-mer is used in a window of consecutive k-mers while providing theoretical guarantees for achieving minimal information loss when using the minimizer k-mers [50].

To identify these minimizer k-mers, BLEND generates the hash values of all seed-sized subsequences of a sequence and finds the minimizer k-mers using these hash values in three steps. First, BLEND generates the hash values of *all overlapping* l-mers of a genomic sequence using any hash function h. Second, it uses each consecutive n-many l-mer to generate the hash value of a k-mer using the SimHash technique such that $l + n - 1 = k$ as explained in Section 2.1. Third, BLEND uses the hash values of these k-mers when finding the minimizer k-mers with a minimum hash value in a window of w consecutive k-mers, which we identify as the seed. Such an approach avoids requiring any certain l-mer between two minimizer k-mers (i.e., $l < k$) to exactly match to find fuzzy seed matches. Figure 3 shows the example l-mers that BLEND identifies for generating the hash values of seeds (e.g., K_{S_1}).

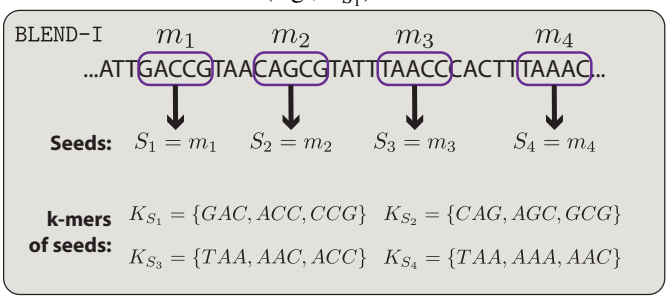

Figure 3: BLEND-I generates the seed denoted as S_i from a given sequence directly from the i^{th} minimizer, m_i. `BLEND-I` uses the hash values of n-many ($n = 3$ in this example) immediately overlapping k-mers (e.g., $k = 3$) when generating the hash values for seeds. These k-mers are included in the list K_{S_i}.

The goal of the second mechanism, `BLEND-S`, is to allow indels when matching short subsequences with a single lookup while reducing the amount of seeds so that BLEND can find fuzzy matches of short subsequences using these seeds. To achieve this, `BLEND-S` finds all the minimizer k-mers of a target sequence and links each n-many consecutive minimizer k-mers as a strobemer seed [52] as shown in Figure 4. Since minimizer k-mers may come from either of the strands of DNA, `BLEND-S` ensures the correctness of strobemer seeds by linking only the minimizer k-mers from the same strand. Strobemer seeds can find matching short subsequences with indels as the sequences between linked minimizer k-mers are ignored (i.e., masked) in seeds. Strobemer sequences are mainly useful if sequences are likely to have indels or a large portions of short subsequences are likely to match such that linking many minimizer k-mers increase the length of the short subsequences without reducing the sensitivity.

BLEND generates the hash values of seeds it finds using the `BLEND-S` mechanism in two steps. First, BLEND hashes all overlapping k-mers of a sequence using any hash function h and uses these hash values to find all the minimizer k-mers of that sequence given the window parameter w. We can define the list of all minimizer k-mers of a sequence S_G as $K_{S_G} = \{k_1, k_2, \ldots, k_m\}$ where k_t is the t^{th} minimizer k-mer in the sequence as we show four such minimizers in Figure 4. For simplicity, let us assume that all the k-mers in K_{S_G} are from the same strand. Then, `BLEND-S` generates a strobemer seed, S_i, that links the following list of k-mers $K_{S_i} = \{k_i, k_{i+1}, \ldots, k_{i+n-1}\}\, \forall i \in [1, m-n+1]$ where n determines the number of consecutive minimizer k-mers from the same strand (e.g., S_1 in Figure 4). BLEND uses the hash values of the k-mers in K_{S_i} with the SimHash technique to generate the hash value of the strobemer seed S_i as explained in Section 2.1. Such a strategy allows BLEND to link consecutive minimizer k-mers to generate a strobemer sequence while enabling any of these minimizer k-mers to mismatch when finding fuzzy seed matches.

2.3. Storing the Hash Values

Our goal is to enable efficient lookup of the hash values of seeds of target sequences. To this end, BLEND uses a hash table to store the hash values of all the seeds of target sequences,

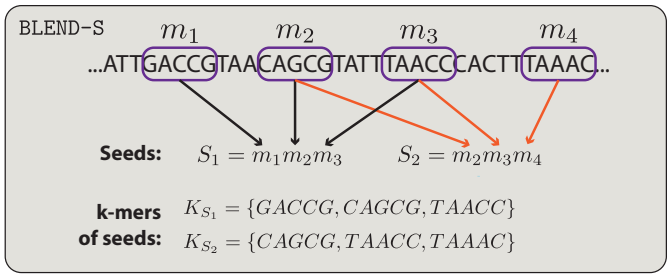

Figure 4: `BLEND-S` generates the seed denoted as S_i from a given sequence by linking n-many (e.g., $n = 3$) minimizer k-mers (e.g., $k = 5$), $m_i, m_{i+1}, \ldots m_{i+n-1}$. `BLEND-S` uses the hash values of linked minimizer k-mers when generating the hash values for seeds. These k-mers are included in the list K_{S_i}

which is usually known as the *indexing* step. Keys of the hash table are hash values of seeds, and the value that a key returns is a *list* of metadata information (i.e., seed length, position in the target sequence, and the unique name of the target sequence). BLEND keeps minimal metadata information for each seed sufficient to locate seeds in target sequences. Since similar or equivalent seeds can share the same hash value, BLEND stores these seeds using the same hash value in the hash table. Thus, a query to the hash table returns all fuzzy seed matches with the same hash value.

2.4. Querying the Hash Values

The primary goal of BLEND is to efficiently find fuzzy seed matches between target and query sequences with high sensitivity (i.e., any arbitrary character of two seeds can mismatch). To achieve this, BLEND iterates over all query sequences and uses the hash table from the indexing step to find fuzzy seed matches between query and target sequences in two steps. First, to generate the hash values of seeds of query sequences, BLEND follows the same steps for generating a hash value for a seed as explained in Sections 2.2 and 2.1. BLEND follows the same steps so that it can generate the same hash values for highly similar seeds between target and query sequences. Second, our goal is to efficiently find fuzzy seed matches between query and target sequences. To achieve this, BLEND uses the hash table it generates in the indexing step to query the hash values of seeds of query sequences. The query to the hash table returns the list of seeds of the target sequences that have the same hash value with the seed of a query sequence. Thus, the list of seeds that the hash table returns is the list of fuzzy seed matches for a seed of a query sequence as they share the same hash value. BLEND can find fuzzy seed matches with a single lookup using the hash values that it generates for the seeds from both query and target sequences.

BLEND finds fuzzy seed matches mainly for two important genomics applications: read overlapping and read mapping. For these applications, BLEND stores all the list of fuzzy seed matches between query and target sequences to perform *chaining* among fuzzy seed matches that fall in the same target sequence (overlapping reads) optionally followed by alignment (read mapping) as described in minimap2 [29].

3. Evaluation

3.1. Evaluation Methodology

We replace the mechanism in minimap2 that generates hash values for seeds with BLEND to find fuzzy seed matches when performing end-to-end read overlapping and read mapping. We also incorporate the `BLEND-I` and `BLEND-S` seeding mechanisms in the implementation and provide the user to choose either of these seeding techniques when using BLEND. We provide a set of default parameters that we optimize based on sequencing technology and the application to perform (e.g., read overlapping). We explain the details regarding the parameters in Supplementary Tables S17 and S18. We determine these default parameters empirically by testing the performance and accuracy of BLEND with different values for some parameters (i.e., k-mer length, number of k-mers to include in a seed, and the window length) as shown in Supplementary Table S12. We show the trade-offs between the seeding mechanisms `BLEND-I` and `BLEND-S` in Supplementary Figures S1 and S2 and Supplementary Tables S9 - S11 in terms of their performance and accuracy.

For our evaluation, we use real and simulated read datasets as well as their corresponding reference genomes. We list the details of these datasets in Table 1. To evaluate BLEND in several common scenarios in read overlapping and read mapping, we classify our datasets into three categories: 1) highly accurate long reads (i.e., PacBio HiFi), 2) erroneous long reads (i.e., PacBio CLR and Oxford Nanopore Technologies), and 3) short reads (i.e., Illumina). We use PBSIM2 [44] to simulate erroneous reads from both PacBio CLR and Oxford Nanopore Technologies (ONT). HiFi and Illumina reads are from real datasets. To use realistic depth of coverage, we use SeqKit [57] to down-sample the original *E. coli* and *D. ananassae* reads to 100× and 50× sequencing depth of coverage, respectively. For *D. ananassae* and *E. coli* genomes, the reference genomes are the high-quality assemblies generated using the same read sets we use for these genomes [60].

Table 1: Details of datasets used in evaluation.

Organism	Library	Reads (#)	Seq. Depth	SRA Accession	Reference Genome
Human CHM13	PacBio HiFi	3,167,477	16	SRR11292122-3	GCA_009914755.3
D. ananassae	PacBio HiFi	1,195,370	50	SRR11442117	[60]
E. coli	PacBio HiFi	38,703	100	SRR11434954	[60]
Yeast	PacBio CLR*	270,849	200	Simulated P6-C4	GCA_000146045.2
	Oxford Nanopore Tech.*	135,296	100	Simulated R9.5	GCA_000146045.2
	Illumina MiSeq	3,318,467	80	ERR1938683	

* We use PBSIM2 to generate the simulated PacBio and ONT reads from the Yeast genome.
We include the simulated chemistry under SRA Accession.

We evaluate BLEND based on two use cases: 1) read overlapping and 2) read mapping to a reference genome. For read overlapping, we perform *all-vs-all overlapping* to find all pairs of overlapping reads within the same dataset (i.e., the target and query sequences are the same set of sequences). To evaluate the efficiency of overlaps, we calculate the average length of overlaps and the number of seed matches per overlap to calculate the overlap statistics. To evaluate the quality of overlapping reads based on the accuracy of the assemblies that we generate from overlaps, we use miniasm [28]. We use miniasm because it does not perform error correction when generating *de novo* assemblies, which allows us to directly assess the quality of overlaps without using additional approaches for improving the accuracy of assemblies. We use `mhap2paf.pl` package as provided by miniasm to convert the output of MHAP to the format that miniasm requires (i.e., PAF). We use QUAST [21] to measure statistics related to the contiguity, length, and the accuracy of *de novo* assemblies such as the overall assembly length, largest contig, NG50 and NGA50 statistics (i.e., statistics related the length of the shortest contig at the half of the overall reference genome length), k-mer completeness (i.e., amount of shared k-mers between the reference genome and an assembly), the ratio of the misassembly length to the actual assembly length, and GC content (i.e., the ratio of G and C bases to A and T bases in an assembly). We use dnadiff [39] to measure the accuracy of *de novo* assemblies based on 1) the average identity of an assembly when compared to its reference genome and 2) the fraction of overall bases in a reference genome that align to a

given assembly (i.e., genome fraction). We compare BLEND with minimap2 [29] and MHAP [7] for read overlapping. For the Human CHM13 genome, MHAP generates a large output such that we cannot generate the assembly as miniasm exceeds the memory space we have in our system (i.e., 1TB).

For read mapping, we map all reads in a dataset (i.e., query sequences) to their corresponding reference genome (i.e., target sequence). To evaluate the coverage of read mapping, we use BEDTools [48] and Mosdepth [45] to calculate the breadth of coverage (i.e., percentage of bases in a reference genome covered by at least one read) and mean depth of coverage (i.e., the average number of read alignments per base in a reference genome), respectively. To evaluate the quality of read mapping, we use BAMUtil [25] to calculate mapping rate (i.e., number of aligned reads) and rate of properly paired reads for paired-end mapping. To evaluate the accuracy of read mapping, we measure 1) the overall number of true mappings with very high mapping quality (i.e., 60 mapping quality score) and 2) overall error rate in read mapping. To generate these results, we use the `paftools mapeval` tool provided by minimap2 that takes the reads from PBSIM2 such that the read IDs are annotated with their true mapping information, and finds incorrect mappings in the SAM files provided by read mapping tools. We compare BLEND with minimap2, LRA [49], Winnowmap2 [23, 24], and S-conLSH [9, 10] when performing read mapping. We do not evaluate 1) LRA, Winnowmap2, and S-conLSH when mapping paired-end short reads as these tools do not support mapping paired-end short reads and 2) S-conLSH for the *D. ananassae* genome as S-conLSH crashes due to a segmentation fault when mapping reads to the *D. ananassae* reference genome.

For both use cases, we use the `time` command in Linux to evaluate the performance and peak memory footprints. We use the default parameters of all the tools suggested for certain use cases and sequencing technologies, when applicable (e.g., mapping HiFi reads in minimap2). Since minimap2 and MHAP do not provide default parameters for read overlapping using HiFi reads, we use the parameters that HiCanu [43] uses for overlapping HiFi reads with minimap2 and MHAP. We provide the details regarding the parameters and versions we use for each tool in Supplementary Tables S17, S18, and S19. To show BLEND can provide better accuracy with the same set of parameters, we use same parameters that BLEND uses in minimap2 and show the performance and accuracy results in Supplementary Figures S3 and S4, and Supplementary Tables S13 - S15.

3.2. Use Case 1: Read Overlapping

3.2.1. Performance. Figure 5 shows the CPU time and peak memory footprint comparisons for read overlapping. We make the following five observations. First, BLEND provides speedup by 2.6×-63.5× (on average 19.5×) and 22.2×-6317.1× (on average 1386.8×) while reducing the memory footprint by 0.9×-9.7× (on average 3.6×) and 35.9×-238.1× (on average 130.5×) compared to minimap2 and MHAP, respectively. BLEND is significantly more performant and provides less memory overheads than MHAP because MHAP generates many hash values for seeds regardless of the length of the sequences while BLEND provides windowing guarantees when generating seeds, which allows sampling the number of seeds based on the sequence length. Second, when considering only HiFi reads, BLEND provides even higher speedup, on average, by 30.8× and 2294.4× while reducing the memory footprint by 5.4× and 190.4× compared to minimap2 and MHAP, respectively. HiFi reads allow BLEND to increase the window length (i.e., $w = 200$) when finding the minimizer k-mer of a seed, which improves the performance and reduces the memory overhead without reducing the accuracy. This is possible mainly because BLEND can find *both* fuzzy and exact seed matches, which enables BLEND to find *unique* fuzzy seed matches that minimap2 *cannot* find due to its exact-matching seed requirement. Third, we find that BLEND requires less than 16GB of memory space for *all* the datasets, making it largely possible to find overlapping reads even with a personal computer with relatively small memory space. BLEND has a lower memory footprint because 1) BLEND uses as many seeds as the number of minimizer k-mers per sequence to benefit from the reduced storage requirements that minimizer k-mers provide and 2) the window length is larger than minimap2 as BLEND can tolerate increasing this window length with the fuzzy seed matches without reducing the accuracy. Fourth, when using erroneous reads (i.e., PacBio CLR and ONT), BLENDprovides better performance but similar memory overhead to minimap2. The set of parameters we use for erroneous reads prevents BLEND from using large windows (i.e., $w = 10$ instead of $w = 200$) without reducing the accuracy of read overlapping. This causes BLEND to use many seeds, which requires higher memory than when using erroneous reads. Fifth, we use the same set of parameters (i.e., the seed length and the window length) with minimap2 that BLEND uses to observe the benefits that BLEND provides with PacBio CLR and ONT datasets. We cannot perform the same experiment for the HiFi datasets because BLEND uses seeds of length 31, which minimap2 cannot support due to the maximum seed length limitation in its implementation (i.e., max. 28). We call this version of minimap2, minimap2-Eq. We show in Supplementary Figure S3 that minimap2-Eq performs, on average, $\sim$ 10% better than BLEND when using the same set of parameters while providing worse accuracy than BLEND when generating the assemblies as shown in Supplementary Table S13. Performing worse than minimap2-Eq is expected because we apply the same sampling rate and k-mer size and use much simpler hashing mechanism than BLEND, which leads to using similar number of seeds while processing them faster with cheaper hashing. The main benefit of BLEND is to provide overall higher accuracy than both the baseline minimap2 and Minimap-Eq, which we can achieve by finding unique fuzzy seed matches that minimap2 cannot find. We conclude that BLEND is significantly more memory-efficient and faster than other tools to find overlaps, especially when using HiFi reads with its ability to sample many seeds using high values of w without reducing the accuracy.

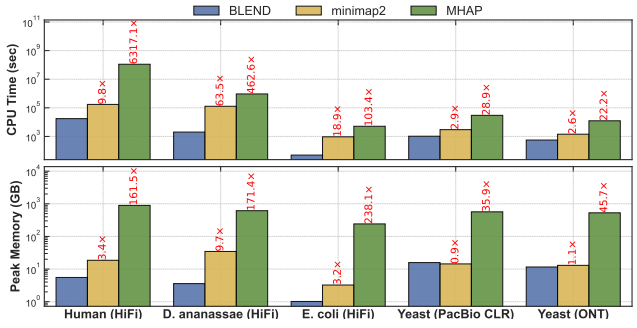

Figure 5: CPU time and peak memory footprint comparisons of read overlapping.

3.2.2. Overlap Statistics. Figure 6 shows the average length of overlaps and the average number of seed matches that each tool finds to identify the overlaps between reads. We make the following two key observations. First, we observe that BLEND finds overlaps longer than minimap2 and MHAP can find in most cases. BLEND can 1) uniquely find the fuzzy seed matches that the exact-matching-based tools cannot find and

2) perform chaining on these fuzzy seed matches to increase the length of overlap using many fuzzy seed matches that are relatively close to each other. Finding more distinct seeds and chaining these seeds enable BLEND to find longer overlaps than other tools. Second, BLEND uses significantly fewer seed matches than other tools by up to 27.34× to find these longer overlaps. This is mainly because BLEND needs much fewer seeds as it uses 1) higher window lengths than minimap2 and 2) provides windowing guarantees unlike MHAP. We conclude that the performance and memory-efficiency improvements in read overlapping we explain in Section 3.2.1 are proportional to the reduction in the seed matches that BLEND uses to find overlapping reads. Thus, reducing the number of seed matches helps improve the performance and memory space efficiency of BLEND.

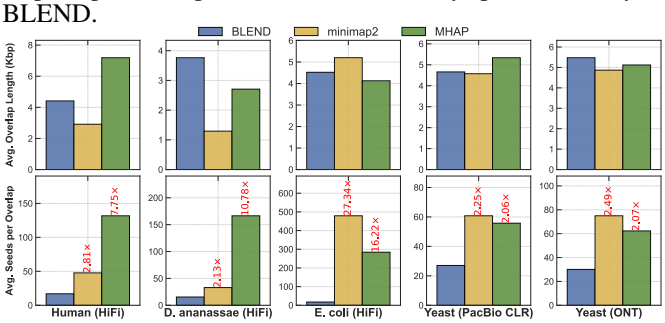

Figure 6: Average length of overlaps and average number of seeds used to find a single overlap.

3.2.3. Assembly Quality Assessment. Our goal is to assess the quality of assemblies generated using the overlapping reads found by BLEND, minimap2, and MHAP. Table 2 shows the statistics related to the accuracy of assemblies (i.e., the 7 statistics on the left-most part of the table), and the statistics related to assembly length and contiguity (i.e., the 3 statistics on the right-most part of the table) when compared to their respective reference genomes. We make the following five key observations based on the accuracy results of assemblies. First, we observe that we can construct more accurate assemblies in terms of average identity and k-mer completeness when we use the overlapping reads that BLEND finds than those minimap2 and MHAP find. These results show that the assemblies we generate using the BLEND overlaps are more similar to their corresponding reference genome. BLEND can find unique and accurate overlaps using fuzzy seed matches that lead to more accurate *de novo* assemblies than the minimap2 and MHAP overlaps due to their lack of support for fuzzy seed matching. Second, we observe that assemblies generated using BLEND overlaps usually cover a higher fraction of the reference genome than minimap2 and MHAP overlaps. Third, although the average identity and genome fraction results seem mixed for the PacBio CLR and ONT reads such that BLEND is best in terms of either average identity or genome fraction, we believe these two statistics should be considered together (e.g., by multiplying both results). This is because, an highly accurate but much smaller fraction of the assembly can align to a reference genome, giving the best results for the average identity. We observe that this is the case for the *D. ananassae* and *Yeast* (PacBio CLR) genomes such that MHAP provides very high average identity only for the much smaller fraction of the assemblies than the assemblies generated using BLEND and minimap2 overlaps. Thus, when we combine average identity and genome fraction results, we observe that BLEND consistently provides the best results for all the datasets. Fourth, BLEND and minimap2 usually provides the best results when the assemblies are aligned to the reference genome using QUAST in terms of aligned length, misassembly ratio, and NGA50 statistics. In most cases, QUAST cannot generate these statistics for the MHAP results as usually a small portion of the assemblies align the reference genome when the MHAP overlaps are used. Fifth, we find that assemblies generated from BLEND overlaps are less biased than minimap2 and MHAP overlaps. Our observation is based on the average GC content in Table 2 and the GC distributions in Supplementary Figure S5 that are mostly closer to the reference genomes. We conclude that BLEND overlaps yield assemblies with higher accuracy and less bias compared to the assemblies that the minimap2 and MHAP overlaps generate in most cases.

Table 2 shows the results related to assembly length and contiguity on its right-most part, from which we make the following two observations. First, we show that BLEND yields assemblies with better contiguity when using HiFi reads due to mostly 1) higher NG50 and 2) the largest contigs that BLEND generates are longer than that of minimap2 with the exception for the human genome. BLEND finds unique overlaps that minimap2 cannot find because of BLEND's ability to detect fuzzy seed matches. We believe such unique overlaps *fill the gap* that minimap2 cannot, which improves the contiguity in most cases. Second, we observe that BLEND and minimap2 result in assemblies where the overall length is mostly closer to the reference genome assembly. We conclude that BLEND enables generating assemblies with contiguity to minimap2. BLEND achieves both high accuracy and high contiguity by using 1) fuzzy seed matches and 2) fewer seeds to find overlapping reads. Fewer overlaps with fuzzy seed matches are likely to provide the assembly graph with fewer edges to remove in transitive reduction and bubbles to collapse in the genome assembly step, which results in more effective overlap construction when using miniasm to generate *de novo* assemblies [28].

3.3. Use Case 2: Read Mapping

3.3.1. Performance. Figure 7 shows the CPU time and the peak memory footprint comparisons when performing read mapping to the corresponding reference genomes. We make the following two key observations. First, we observe that BLEND provides speedup by 0.7×-3.7× (on average 1.7×), 1.7×-26.5× (on average 12.5×), 1.8×-19.3× (on average 7.5×), and 6.8×-58.1× (on average 21.8×) compared to minimap2, LRA, Winnowmap2, and S-conLSH, respectively. Although BLEND performs better than most of these tools, the speedup ratio is usually lower than what we observe in read overlapping discussed in Section 3.2.1. Read mapping includes an additional computationally costly step that read overlapping skips, which is the read alignment. The extra overhead of read alignment slightly hinders the benefits that BLEND provides that we observe in read overlapping. Second, we find that BLEND provides slightly more memory overhead than minimap2 and LRA (on average 0.9× and 0.8×), and lower than Winnowmap2 and S-conLSH by 0.9×-2.7× (on average 1.7×), 0.5×-5.6× (on average 1.9×), respectively. BLEND cannot provide the similar reductions in the memory overhead that we observe in read overlapping due to the more narrow window length ($w = 50$ instead of $w = 200$) it uses to find the minimizer k-mers for HiFi reads with high accuracy. Using a narrow window length generates more seeds to store in a hash table than using a high window length, which proportionally increases the peak memory space requirements. Third, we use the same parameters that BLEND uses with minimap2 to assess the accuracy benefits of finding fuzzy seed matches rather than just using exact-matching seeds in read mapping. We call the version of minimap2 that uses the same set of parameters minimap2-Eq. We observe that BLEND performs, on average, 1.3× better than minimap2-Eq as shown in Supplementary Figure S4 while mostly providing higher

Table 2: Assembly quality comparisons.

Dataset	Tool	Average Identity (%)	Genome Fraction (%)	K-mer Compl. (%)	Aligned Length (Mbp)	Misassembly Ratio (%)	NGA50 (Kbp)	Average GC (%)	Assembly Length (Mbp)	Largest Contig (Mbp)	NG50 (Kbp)
Human CHM13	BLEND	**99.8526**	**98.4847**	**90.15**	3,092.59	**0.0108**	5,442	**40.78**	**3,095.210**	22.840	5,442
	minimap2	99.7421	97.1493	83.05	**3,095.49**	0.0503	**7,133**	40.71	3,100.974	**47.139**	**7,134**
	MHAP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Reference	100	100	100	3,054.83	0	154,260	40.85	3,054.832	248.387	154,260
D. ananassae	BLEND	**99.7856**	**97.2308**	**86.43**	240.39	0.1230	**792**	**41.75**	**247.153**	**6.233**	**799**
	minimap2	99.7044	96.3190	72.33	**289.45**	**0.1013**	273	41.68	298.280	4.434	273
	MHAP	99.5551	0.7276	0.21	2.29	0.2197	N/A	42.07	2.350	0.286	N/A
	Reference	100	100	100	213.81	0	26,427	41.81	213.818	30.673	26,427
E. coli	BLEND	**99.8320**	**99.8801**	**87.91**	**5.12**	**0.0344**	**3,417**	**50.53**	5.122	**3.417**	**3,417**
	minimap2	99.7064	99.8748	79.27	5.09	0.3068	3,087	50.47	**5.042**	3.089	3,089
	MHAP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	5.094	N/A	N/A
	Reference	100	100	100	5.05	0	4,945	50.52	5.046	4.946	4,945
Yeast (PacBio)	BLEND	89.1582	**97.6297**	**11.13**	**0.227**	N/A	N/A	**38.80**	13.679	1.105	551
	minimap2	88.9002	96.9709	9.74	0.195	N/A	N/A	38.85	**12.333**	**1.561**	**828**
	MHAP	**89.2182**	88.5928	9.5	0.186	N/A	N/A	38.81	10.990	1.024	436
	Reference	100	100	100	12.16	0	924	38.15	12.157	1.532	924
Yeast (ONT)	BLEND	**89.7622**	99.2982	**13.68**	**0.377**	N/A	N/A	**38.66**	**12.164**	1.554	825
	minimap2	88.9393	**99.6878**	12.06	0.328	N/A	N/A	38.74	12.373	**1.560**	**942**
	MHAP	89.1970	89.2785	11.35	0.297	N/A	N/A	38.84	10.920	1.443	619
	Reference	100	100	100	12.16	0	924	38.15	12.157	1.532	924

Best results are highlighted with **bold** text. For most metrics, the best results are the ones closest to the corresponding value of the reference genome.
The best results for *Aligned Length* are determined by the highest number within each dataset. We do not highlight the reference results as the best results.
N/A indicates that we could not generate the corresponding result because either the tool failed or QUAST failed to generate the statistic.

read mapping quality and accuracy as shown in Supplementary Tables S14 and S15. We conclude that BLEND, on average, performs better than all tools without reducing the accuracy and provides better memory footprint than Winnowmap2 and S-conLSH.

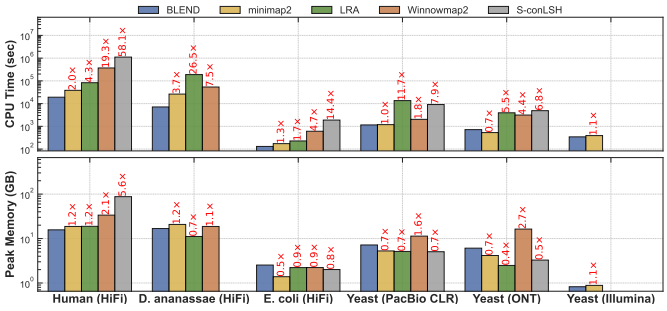

Figure 7: CPU time and peak memory footprint comparisons of read mapping.

3.3.2. Read Mapping Accuracy. To assess the accuracy of read mapping, we map two simulated erroneous read sets (i.e., PacBio CLR and ONT) to their reference genomes and compare the read mapping results with their true mapping results as generated by PBSIM2. To this end, we use paftools to assess the read mapping accuracy and show these results in Supplementary Table S16. We make two observations, First, we observe that, BLEND provides the most number of high quality true mappings for the PacBio CLR reads and the lowest overall error rate for the ONT reads. For the cases where BLEND does not provide the best results, minimap2 provides most number of high quality true mappings for the ONT reads, and Winnowmap2 provides the best error rate for the PacBio CLR reads. Second, LRA and S-conLSH map most of the reads to incorrect locations, making these tools inaccurate for mapping erroneous reads when compared BLEND, minimap2, and Winnowmap2. We conclude that although the results are mixed, BLEND is the only tool that always performs best in either of these two metrics, providing the overall best accuracy results as generated by the paftools.

3.3.3. Read Mapping Quality. Our goal is to assess the quality of read mappings in terms of four metrics: mean depth of coverage, breadth of coverage, number of aligned reads, and the ratio of the paired-end reads that are properly paired in mapping. Table 3 shows the quality of read mappings based on these metrics when using BLEND, minimap2, LRA, and Winnowmap2. We exclude S-conLSH from the read mapping quality comparisons as we cannot convert its SAM output to BAM format to properly index the BAM file due to the issues with its SAM output format. We make three observations. First, BLEND, minimap2, and Winnowmap2 cover most portion of the reference genomes after read mapping than LRA does. This result shows that these tools are less biased in mapping reads to particular regions (e.g., repetitive regions) than LRA. Second, both BLEND and minimap2 map almost complete set of reads to the reference genome for all the datasets, while Winnowmap2 suffers from the slightly lower number of aligned reads when mapping erroneous PacBio CLR and ONT reads. This result shows that although Winnowmap2 generates lower error rates for the PacBio CLR reads as we discuss in Section 3.3.2, the higher portion of the unmapped reads are not counted towards the error rate. This suggest us Winnowmap2 provides a better precision while BLEND and minimap2 provide a better recall in terms of the read mapping accuracy. Third, we find that all the tools generate read mappings with a depth of coverage that is significantly close to their sequencing depth of coverage. This shows that almost all reads map to the reference genome evenly. We conclude that read mapping qualities of BLEND, minimap2, and Winnowmap2 are highly similar, while LRA provides slightly worse results. It is worth noting that BLEND can achieve comparable accuracy while using larger window lengths when finding the minimizer k-mers, which usually has an affect on the accuracy. BLEND does this by finding unique fuzzy seed matches that the other tools cannot find due to their exact-matching seed requirements.

4. Discussion

We demonstrate that there are usually too many redundant *short* and *exact-matching* seeds used to find overlaps between sequences, as shown in Figure 6. These redundant seeds usually exacerbate the performance and peak memory space require-

Table 3: Read mapping quality comparisons.

Dataset	Tool	Mean Depth of Cov. (×)	Breadth of Coverage (%)	Aligned Reads (#)	Properly Paired (%)
Human CHM13	BLEND	**16.58**	**99.99**	3,171,916	NA
	minimap2	**16.58**	**99.99**	**3,172,261**	NA
	LRA	16.37	99.06	3,137,631	NA
	Winnowmap2	**16.58**	**99.99**	3,171,313	NA
D. ananassae	BLEND	57.37	99.66	1,223,388	NA
	minimap2	**57.57**	**99.67**	1,245,931	NA
	LRA	57.06	99.60	1,235,098	NA
	Winnowmap2	57.40	99.66	**1,249,575**	NA
E. coli	BLEND	**99.14**	99.90	39,048	NA
	minimap2	**99.14**	99.90	**39,065**	NA
	LRA	99.10	99.90	39,063	NA
	Winnowmap2	**99.14**	99.90	39,036	NA
Yeast (PacBio)	BLEND	195.84	**99.98**	269,804	NA
	minimap2	**195.86**	**99.98**	**269,935**	NA
	LRA	194.65	99.97	267,399	NA
	Winnowmap2	192.35	**99.98**	259,073	NA
Yeast (ONT)	BLEND	97.86	**99.97**	134,721	NA
	minimap2	**97.88**	99.96	**134,885**	NA
	LRA	97.25	99.95	132,862	NA
	Winnowmap2	97.04	99.96	130,978	NA
Yeast (Illumina)	BLEND	**79.93**	99.97	**6,494,489**	95.89
	minimap2	79.91	99.97	6,492,994	95.89

Best results are highlighted with **bold** text.

Properly paired rate is only available for paired-end Illumina reads.

ment problems that read overlapping and read mapping suffer from as the number of chaining and alignment operations proportionally increases with the number of seed matches between sequences [29]. Such redundant computations have been one of the main limitations against developing population-scale genomics analysis due to the high runtime of a single high coverage genome analysis.

There has been a clear interest in using long or fuzzy seed matches because of their potential to find similarities between target and query sequences efficiently and accurately [4]. To achieve this, earlier works mainly focus on either 1) chaining the exact k-mer matches by tolerating the gaps between them to increase the seed region or 2) linking multiple consecutive minimizer k-mers such as strobemer seeds. Chaining algorithms are becoming a bottleneck in read mappers as the complexity of chaining is determined by the number of seed matches [20]. Linking multiple minimizer k-mers enable tolerating indels when finding the matches of short subsequences between genomic sequence pairs, but these seeds (e.g., strobemer seeds) should still exactly match due to the nature of the hash functions used to generate the hash values of seeds. This requires the seeding techniques to generate exactly the same seed to find either exact-matching or approximate matches of short subsequences. We state that any arbitrary k-mer in the seeds should be tolerated to mismatch to improve the sensitivity of any seeding technique, which has a potential for finding more matching regions while using fewer seeds. Thus, we believe BLEND solves the main limitation of earlier works such that it can generates the same hash value for highly similar seeds to find fuzzy seed matches with a single lookup while improving the performance, memory overhead and the accuracy of the applications that use seeds.

We hope that BLEND advances the field and prospers future work in several ways, which we list some of them next. First, we observe that BLEND is *most effective* when using high coverage and highly accurate long reads. Thus, we believe that BLEND is already ready to scale for longer and more accurate sequencing reads. Second, we believe the vector operations are suitable for hardware acceleration to improve the performance of BLEND further. Such an acceleration is mainly useful when a massive amount of k-mers in a seed are used to generate the hash value for a seed, as these calculations can be done in parallel. We already provide the SIMD implementation to calculate the hash values in `BLEND-I` and `BLEND-S`. We encourage implementing the hashing strategy for the applications that use seeds with suitable architectures such as processing-in-memory [26, 55], GPUs [1], and SSDs [37] to exploit the massive amount of embarrassingly parallel bitwise operations in BLEND. Third, we believe it is possible to apply the hashing technique we use in BLEND for many seeding techniques with a proper design. We already show we can apply SimHash in regular minimizer k-mers or strobemers that are based on minimizer k-mers in `BLEND-I` and `BLEND-S`, respectively. Strobemers can be generated using k-mer sampling strategies other than minimizer k-mers, which are based on syncmers and random selection of k-mers (i.e., randstrobes) [53]. It is worth exploring and rethinking the hash functions used in these seeding techniques. Fourth, we believe potential machine learning applications can be used to generate more sensitive hash values, similar to learning-to-hash approaches [62], while using the hash values that BLEND generates as a starting point.

5. Conclusion

We propose BLEND, a mechanism that can efficiently find fuzzy seed matches between sequences to significantly improve the performance, memory space efficiency, and accuracy of two important applications: 1) read overlapping and 2) read mapping. Based on the experiments we perform using real and simulated datasets, we make five key observations. First, BLEND provides significant speedup for read overlapping by 2.6×-63.5× (on average 19.5×) and 22.2×-6317.1× (on average 1386.8×), while reducing the peak memory footprint by 0.9×-9.7× (on average 3.6×) and 35.9×-238.1× (on average 130.5×) compared to minimap2 and MHAP. Second, we observe that BLEND finds longer overlaps, in general, while using significantly fewer seed matches by up to 27.34× to find these overlaps. Third, we find that we can usually generate more *accurate* assemblies when using the overlaps that BLEND finds than those found by minimap2 and MHAP. Fourth, for read mapping, we find that BLEND, on average, provides speedup by 1) 1.7×, 12.5×, 7.5×, and 21.8× compared to minimap2, LRA, Winnowmap2, and S-conLSH, respectively. Fifth, we observe that BLEND, minimap2, and Winnowmap2 provide both high quality and better accuracy in read mapping in all datasets while the accuracy of LRA and S-conLSH is low when using erroneous long reads. We conclude that BLEND can use fewer fuzzy seed matches to significantly improve the performance and reduce the memory overhead of read overlapping without losing from the accuracy, while BLEND, on average, provides better performance and similar memory footprint in read mapping, without reducing the read mapping quality and accuracy.

6. Acknowledgements

We thank the anonymous reviewers of the Bioinformatics journal and the SAFARI members for feedback. We are grateful for the detailed comments that Kristoffer Sahlin provided, which improved our mechanism and the manuscript greatly.

7. Funding

This work is supported by gifts from Intel [to O.M.]; VMware [to O.M.]; and a Semiconductor Research Corporation grant [to O.M.].

References

[1] A. Zeni *et al.*, "LOGAN: High-Performance GPU-Based X-Drop Long-Read Alignment," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 462–471.

[2] C. Alkan *et al.*, "Genome structural variation discovery and genotyping," *Nature Reviews Genetics*, vol. 12, no. 5, pp. 363–376, May 2011.

[3] M. Alser *et al.*, "Accelerating Genome Analysis: A Primer on an Ongoing Journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, Oct. 2020.

[4] M. Alser *et al.*, "Technology dictates algorithms: recent developments in read alignment," *Genome Biology*, vol. 22, no. 1, p. 249, Aug. 2021.

[5] M.-M. Aynaud *et al.*, "A multiplexed, next generation sequencing platform for high-throughput detection of SARS-CoV-2," *Nature Communications*, vol. 12, no. 1, p. 1405, Mar. 2021.

[6] S. Baichoo and C. A. Ouzounis, "Computational complexity of algorithms for sequence comparison, short-read assembly and genome alignment," *Biosystems*, vol. 156-157, pp. 72–85, Jun. 2017.

[7] K. Berlin *et al.*, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature Biotechnology*, vol. 33, no. 6, pp. 623–630, Jun. 2015.

[8] S. Canzar and S. L. Salzberg, "Short Read Mapping: An Algorithmic Tour," *Proceedings of the IEEE*, vol. 105, no. 3, pp. 436–458, Mar. 2017.

[9] A. Chakraborty and S. Bandyopadhyay, "conLSH: Context based Locality Sensitive Hashing for mapping of noisy SMRT reads," *Computational Biology and Chemistry*, vol. 85, p. 107206, Apr. 2020.

[10] A. Chakraborty *et al.*, "S-conLSH: alignment-free gapped mapping of noisy long reads," *BMC Bioinformatics*, vol. 22, no. 1, p. 64, Feb. 2021.

[11] M. S. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 380–388.

[12] H. Cheng *et al.*, "Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm," *Nature Methods*, vol. 18, no. 2, pp. 170–175, Feb. 2021.

[13] C.-S. Chin and A. Khalak, "Human Genome Assembly in 100 Minutes," *bioRxiv*, p. 705616, Jan. 2019.

[14] M. David *et al.*, "SHRiMP2: Sensitive yet Practical Short Read Mapping," *Bioinformatics*, vol. 27, no. 7, pp. 1011–1012, Apr. 2011.

[15] D. DeBlasio *et al.*, "Practical Universal K-Mer Sets for Minimizer Schemes," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, ser. BCB '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 167–176.

[16] B. Ekim *et al.*, "Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer," *Cell Systems*, vol. 12, no. 10, pp. 958–968.e6, Oct. 2021.

[17] C. Firtina *et al.*, "Apollo: a sequencing-technology-independent, scalable and accurate assembly polishing algorithm," *Bioinformatics*, vol. 36, no. 12, pp. 3669–3679, Jun. 2020.

[18] J. M. Friedman *et al.*, "Genome-wide sequencing in acutely ill infants: genomic medicine's critical application?" *Genetics in Medicine*, vol. 21, no. 2, pp. 498–504, Feb. 2019.

[19] S. Goodwin *et al.*, "Coming of age: ten years of next-generation sequencing technologies," *Nature Reviews Genetics*, vol. 17, pp. 333–351, may 2016.

[20] L. Guo *et al.*, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 127–135.

[21] A. Gurevich *et al.*, "QUAST: quality assessment tool for genome assemblies," *Bioinformatics*, vol. 29, no. 8, pp. 1072–1075, Apr. 2013.

[22] T. Hon *et al.*, "Highly accurate long-read HiFi sequencing data for five complex genomes," *Scientific Data*, vol. 7, no. 1, p. 399, Nov. 2020.

[23] C. Jain *et al.*, "Long-read mapping to repetitive reference sequences using Winnowmap2," *Nature Methods*, Apr. 2022.

[24] C. Jain *et al.*, "Weighted minimizer sampling improves long read mapping," *Bioinformatics*, vol. 36, no. Supplement_1, pp. i111–i118, Jul. 2020.

[25] G. Jun *et al.*, "An efficient and scalable analysis framework for variant extraction and refinement from population scale DNA sequence data," *Genome Research*, Apr. 2015.

[26] J. S. Kim *et al.*, "GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, May 2018.

[27] N. LaPierre *et al.*, "Metalign: efficient alignment-based metagenomic profiling via containment min hash," *Genome biology*, vol. 21, no. 1, pp. 1–15, 2020.

[28] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, Jul. 2016.

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

[30] H. Lin *et al.*, "ZOOM! Zillions of oligos mapped," *Bioinformatics*, vol. 24, no. 21, pp. 2431–2437, Nov. 2008.

[31] G. A. Logsdon *et al.*, "Long-read human genome sequencing and its applications," *Nature Reviews Genetics*, vol. 21, no. 10, pp. 597–614, Oct. 2020.

[32] N. J. Loman *et al.*, "A complete bacterial genome assembled de novo using only nanopore sequencing data," *Nature Methods*, vol. 12, no. 8, pp. 733–735, Aug. 2015.

[33] B. Ma *et al.*, "PatternHunter: faster and more sensitive homology search," *Bioinformatics*, vol. 18, no. 3, pp. 440–445, Mar. 2002.

[34] X. Ma *et al.*, "Analysis of error profiles in deep next-generation sequencing data," *Genome Biology*, vol. 20, no. 1, p. 50, Mar. 2019.

[35] A. Mallik and L. Ilie, "ALeS: adaptive-length spaced-seed design," *Bioinformatics*, vol. 37, no. 9, pp. 1206–1210, May 2021.

[36] G. S. Manku *et al.*, "Detecting Near-Duplicates for Web Crawling," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 141–150.

[37] N. Mansouri Ghiasi *et al.*, "GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 635–654.

[38] T. Mantere *et al.*, "Long-Read Sequencing Emerging in Medical Genetics," *Frontiers in Genetics*, vol. 10, p. 426, 2019.

[39] G. Marçais *et al.*, "MUMmer4: A fast and versatile genome alignment system," *PLOS Computational Biology*, vol. 14, no. 1, p. e1005944, Jan. 2018.

[40] A. McKenna *et al.*, "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, Sep. 2010.

[41] J. D. Merker *et al.*, "Long-read genome sequencing identifies causal structural variation in a Mendelian disease," *Genetics in Medicine*, vol. 20, no. 1, pp. 159–163, Jan. 2018.

[42] F. Meyer *et al.*, "Critical Assessment of Metagenome Interpretation-the second round of challenges," *bioRxiv*, 2021.

[43] S. Nurk *et al.*, "HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads," *bioRxiv*, p. 2020.03.14.992248, Jan. 2020.

[44] Y. Ono *et al.*, "PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores," *Bioinformatics*, vol. 37, no. 5, pp. 589–595, Mar. 2021.

[45] B. S. Pedersen and A. R. Quinlan, "Mosdepth: quick coverage calculation for genomes and exomes," *Bioinformatics*, vol. 34, no. 5, pp. 867–868, Mar. 2018.

[46] E. Petrucci *et al.*, "Iterative Spaced Seed Hashing: Closing the Gap Between Spaced Seed Hashing and k-mer Hashing," *Journal of Computational Biology*, vol. 27, no. 2, pp. 223–233, Feb. 2020.

[47] M. Pop *et al.*, "Comparative genome assembly," *Briefings in Bioinformatics*, vol. 5, no. 3, pp. 237–248, Sep. 2004.

[48] A. R. Quinlan and I. M. Hall, "BEDTools: a flexible suite of utilities for comparing genomic features," *Bioinformatics*, vol. 26, no. 6, pp. 841–842, Mar. 2010.

[49] J. Ren and M. J. P. Chaisson, "lra: A long read aligner for sequences and contigs," *PLOS Computational Biology*, vol. 17, no. 6, p. e1009078, Jun. 2021.
[50] M. Roberts *et al.*, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, Dec. 2004.
[51] G. Robertson *et al.*, "De novo assembly and analysis of RNA-seq data," *Nature Methods*, vol. 7, no. 11, pp. 909–912, Nov. 2010.
[52] K. Sahlin, "Effective sequence similarity detection with strobemers," *Genome Research*, vol. 31, no. 11, pp. 2080–2094, Nov. 2021.
[53] K. Sahlin, "Faster short-read mapping with strobemer seeds constructed from syncmers," *bioRxiv*, p. 2021.06.18.449070, Jan. 2021.
[54] S. Schleimer *et al.*, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.
[55] D. Senol Cali *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 951–966.
[56] D. Senol Cali *et al.*, "Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions," *Briefings in Bioinformatics*, vol. 20, no. 4, pp. 1542–1559, Jul. 2019.
[57] W. Shen *et al.*, "SeqKit: A Cross-Platform and Ultrafast Toolkit for FASTA/Q File Manipulation," *PLOS ONE*, vol. 11, no. 10, p. e0163962, Oct. 2016.
[58] J. Shendure *et al.*, "DNA sequencing at 40: past, present and future," *Nature*, vol. 550, no. 7676, pp. 345–353, Oct. 2017.
[59] N. Stoler and A. Nekrutenko, "Sequencing error profiles of Illumina sequencing instruments," *NAR Genomics and Bioinformatics*, vol. 3, no. 1, Mar. 2021.
[60] E. S. Tvedte *et al.*, "Comparison of long-read sequencing technologies in interrogating bacteria and fly genomes," *G3 Genes|Genomes|Genetics*, vol. 11, no. 6, Jun. 2021.
[61] R. Vaser *et al.*, "Fast and accurate de novo genome assembly from long uncorrected reads," *Genome Research*, vol. 27, no. 5, pp. 737–746, May 2017.
[62] J. Wang *et al.*, "A survey on learning to hash," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 769–790, 2018.
[63] D. E. Wood *et al.*, "Improved metagenomic analysis with Kraken 2," *Genome Biology*, vol. 20, no. 1, p. 257, Nov. 2019.
[64] H. Xin *et al.*, "Accelerating read mapping with FastHASH," *BMC Genomics*, vol. 14, no. 1, p. S13, Jan. 2013.
[65] H. Xin *et al.*, "Context-aware seeds for read mapping," *Algorithms for Molecular Biology*, vol. 15, no. 1, p. 10, May 2020.
[66] H. Zhang *et al.*, "A comprehensive evaluation of long read error correction methods," *BMC Genomics*, vol. 21, no. 6, p. 889, Dec. 2020.

Supplementary Material for
BLEND: A Fast, Memory-Efficient, and Accurate Mechanism to Find Fuzzy Seed Matches

S1. Generating the Same Hash Value for Similar Seeds

Our goal is to enable generating the same hash values for highly similar seeds. To this end, BLEND uses the SimHash technique [4, 8] to generate a hash value for a seed S_i using its k-mers K_{S_i}, as explained in Section 2.2 in the main paper. Here, we explain in three steps how two hash values, $B(S_k)$ and $B(S_l)$, that BLEND generates respectively for two seeds S_k and S_l can have the same value (i.e., $B(S_k) = B(S_l)$), although S_k and S_l may not necessarily be the same seed sequences (i.e., $S_k \neq S_l$) as stated by the following Theorem S1.

Theorem S1 *Let S_k and S_l be two seed sequences, and $B(S_k)$ and $B(S_l)$ be their hash values that BLEND generates, respectively. Then, $B(S_k) = B(S_l)$ if $K_{S_k} \simeq K_{S_l}$.*

First, we assume that the following conditions are true:

1. BLEND uses a *deterministic* hash function to *always* generate the same set of hash values, $h(K_{S_i})$, given a set of k-mers K_{S_i} of a seed S_i.
2. BLEND *always* generates the hash value $B(S_i)$ for the given set of hash values of k-mers, $h(K_{S_i})$.
3. All the other relevant parameters used in BLEND are the same when generating seeds to ensure $h(K_{S_k}) = h(K_{S_l})$ if $S_k = S_l$. These parameters mainly are k-mer size k, window length w, hash function h, number of neighbor k-mers included in both sets n, and the seeding mechanism (i.e., `BLEND-I` or `BLEND-S`).

Then, we state the following lemma:

Lemma S1.1 *$B(S_k) = B(S_l)$ if $S_k = S_l$*

To show that Lemma S1.1 is correct, we should show BLEND *always* generates the hash value $B(S_k)$ for the given set of hash values of k-mers, $h(K_{S_k})$. The proof is trivial because the vector transformation step (explained in Section 2.2 in the main paper) generates the same vectors for each hash value in $h(K_{S_k})$, and the vector additions are deterministic such that the addition always generates the same counter vector $C(S_k)$ given $h(K_{S_k})$. The decoding step is also simple and deterministic conversion from a vector to the binary representation of a hash value as explained in Section 2.2 in the main paper. Thus, BLEND always generates the same $B(S_k)$ from $h(K_{S_k})$.

Lemma S1.1 is true because BLEND always generates the same hash value for S_k and if $S_k = S_l$, then $h(K_{S_k}) = h(K_{S_l})$. $B(S_k) = B(S_l)$ if $h(K_{S_k}) = h(K_{S_l})$.

Second, we state the following lemma:

Lemma S1.2 *$B(S_k)$ and $B(S_l)$ can be equal (i.e., $B(S_k) = B(S_l)$) even if two seeds are highly similar but not equal (i.e., $S_k \sim S_l$).*

To prove Lemma S1.2, let us start with a single seed, S_k, its k-mers, K_{S_k}, and its hash value after applying the SimHash technique, $B(S_k)$. We state the following lemma:

Lemma S1.3 *Each value in the counter vector (i.e., $C(S_k)$) used to generate the hash value $B(S_k)$ can only change by +2 or –2 if we replace a* single *k-mer in K_{S_k} with another k-mer (i.e., removing one k-mer* and *adding another k-mer).*

Lemma S1.3 can easily be proven because a single k-mer is encoded into a single vector as shown in Figure 4 from the main paper, and each position of a vector can only have the value either 1 or –1. A value in the counter vector $C(S_k)$ at any position i (i.e., $C(S_k)[i]$) can only change by +1 or –1 after adding *or* removing a single vector, respectively. Thus, Lemma S1.3 is true because making two changes (i.e., replacing a k-mer) can only change the values in the counter vector by *at most* +2 or –2.

Given that Lemma S1.3 is true, let us replace a single k-mer in K_{S_k} with another k-mer to make the seed S_k a different seed: S_l. Then, we know that each value in the counter vectors of these seeds, $C(S_k)$ and $C(S_l)$, can only differ by *at most* +2 or –2 from Lemma S1.3.

We will now generalize Lemma S1.3 to changing a single character in S_k rather than changing a single k-mer in K_{S_k}. Although it is not relevant to consider the character changes in seeds when using the `BLEND-S` seeding technique as it links multiple gapped k-mers (i.e., a single character change is unlikely to affect the other linked k-mer), it is more important for the `BLEND-I` seeding technique as it uses immediately overlapping n-many k-mers to generate a seed. Let us change a single character in S_k to make a different seed S_l. Since S_k is generated from immediately overlapping k-mers, a single character change in S_k can change *at most* k-many k-mers in K_{S_k} and, thus, k-many hash values in $h(K_{S_k})$. Since we know that a single k-mer change can cause at most +2 or –2 difference between the values in $C(S_k)$ and $C(S_l)$ at the same positions, there can be at most $2k$ *difference* between the these values when we change k-many k-mers. Then, the hash values $B(S_k)$ and $B(S_l)$ *must* be equal if the following condition holds true:

- All the values in $C(S_k)$ are either greater than $2k$ or less than $-2k+1$ (i.e., $C(S_k)[i] > 2k$ or $C(S_k)[i] \leq -2k\ \forall i$)

Given the above condition, we will prove $B(S_k) = B(S_l)$. Let us assume that $B(S_k) \neq B(S_l)$. This means that *at least one* position in both $C(S_k)$ and $C(S_l)$ should be one of the following as these are the only conditions that BLEND checks when setting the bits in $B(S_k)$ and $B(S_k)$:

1. $C(S_k)[i] > 0$ and $C(S_l)[i] \leq 0$
2. $C(S_k)[i] \leq 0$ and $C(S_l)[i] > 0$

However, neither of the conditions above can hold true because:

1. If $C(S_k)[i] > 0$ then $C(S_k)[i] > 2k$ and $C(S_k)[i] - 2k > 0$ thus $C(S_l)[i] > 0$ (because there can be at most $2k$ difference between $C(S_k)[i]$ and $C(S_l)[i]$ when we change a single character).
2. If $C(S_k)[i] \leq 0$ then $C(S_k)[i] \leq -2k$ and $C(S_k)[i] + 2k \leq 0$ thus $C(S_l)[i] \leq 0$.

Thus, $B(S_k) = B(S_l)$ by contradiction.

We note that we set our conditions (e.g., $C(S_k)$ are either greater than $2k$ or less than $-2k+1$) for the *most extreme cases* of a single character change because we assume the following most extreme cases:

1. We use `BLEND-I`.
2. K_{S_k} and K_{S_l} differ by k-many k-mers due to a single character change between S_k and S_l.
3. If we replace k-many k-mers of seed S_k due to the single character change between S_k and S_l, the XNOR operation between the hash values of these replaced k-mers returns 0 such that for any replaced k-mer $k_i \in K_{S_k}$ and $k_i \notin K_{S_l}$ there is a replacing k-mer $k_n \in K_{S_l}$ and $k_n \notin K_{S_k}$ that provides $h(k_i) \odot h(k_n) = 0$. This implies that the difference between two encoded vectors of these hash values are *always* 2 at every position because of the opposite bits that these hash values have at each position.

Case 1 above assumes that we use the `BLEND-I` technique that may have a bigger impact in the final value of the counter vector compared to using the `BLEND-S` technique when there is a single character change in seed. Case 2 assumes the worst case in the number of k-mers that need to be replaced with a single character change in a seed. This may not necessarily be true if the change of character is at the beginning or towards the end of the seed such that k-many k-mers do not cover this change. Case 2 also becomes less impactful when seeds include many k-mers such that changing k-many k-mers have slightly minor effect in the counter vector. Case 3 above is the most extreme case because it is unlikely that the replacing hash values can *always* satisfy the condition $h(k_i) \odot h(k_n) = 0$. Since Case 3 is unlikely to always hold true, we can also claim that BLEND is likely to generate the same hash values such that $B(S_k) = B(S_l)$ if S_k and S_l differs by a single character and *some* of the counter vectors in $C(S_k)$ are *close to* either $2k$ or $-2k+1$.

Given that Lemma S1.1 and Lemma S1.2 are true, then Theorem S1 is true. We conclude that BLEND finds *all* exactly matching seeds while it can generate the same hash value for highly similar seeds.

S1.1. A Real Example for Generating the Hash Values of Seeds S_k and S_l

Our goal is to show how the k-mer length k and the number of k-mers to include in a seed, n, affect the final hash value. To this end, we use the following two seeds as found in the Yeast reference genome: S_k : `CGGATGCTACAGTATATACCA` and S_l : `ATGCTACAGTATATACCATCT`. Both seeds are 21-character long. We use two different parameter settings when generating the hash values of these seeds. The first setting uses $k = 7$ as the k-mer length and $n = 15$ as the number of immediately overlapping k-mers to include in a seed so that we can generate the 21-character long seeds S_l and S_k. The second setting uses $k = 15$ as the k-mer length and $n = 7$ as the number of k-mers to include in a seed. We use the `hash64` hash function as provided in the minimap2 implementation to generate the hash values of the k-mers of seeds.

In Supplementary Tables S1 - S8 we show k-mers, the hash values of the k-mers in their binary form, and the gradual change in the counter vectors used to calculate the hash values for seeds S_k and S_l. We update the counter vectors based on the bits in the hash values of each k-mer. Finally, we show the hash values of S_k and S_l in the last rows of each table. In Supplementary Tables S1- S4, we use $k = 7$ as the k-mer length and $n = 15$ as the number of immediately overlapping k-mers to include in a seed. In Supplementary Tables S5- S8, we use $k = 15$ as the k-mer length and $n = 7$ as the number of k-mers to include in a seed.

We make two key observations. First, we observe that the hash values of S_k and S_l are equal ($B(S_k) = B(S_l)$ =`0b11000100 01101100 11101001 10110100`) when we use a short k-mer with high number of neighbors even though these two seeds differ by 3 k-mers. Second, the hash values of these two seeds are not equal when we use fewer neighbors with larger k-mers. For S_k we find the hash value $B(S_k)$ =`0b01101000 01000001 01110100 11000000` and for S_l we find $B(S_l)$ =`0b00101101 10110000 01111100 01010011`. We note that the bit positions with large values in their corresponding counter vectors are less likely to differ between two seeds when the seeds have a large number of k-mers in common. This motivates as to design more intelligent hash functions that are aware of the values in the counter vectors to increase the chance of generating the hash value for similar seeds.

Table S1: Hash Values of the k-mers of seed S_k: `CGGATGCTACAGTATATACCA` for $k = 7$ and $n = 15$. We show the most significant 16 bits of the counter vector $C(S_k)$. Last row shows the most significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[31]	C[30]	C[29]	C[28]	C[27]	C[26]	C[25]	C[24]	C[23]	C[22]	C[21]	C[20]	C[19]	C[18]	C[17]	C[16]
CGGATGC	0b 10100000 01111111 10000110 10110101	1	-1	1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1
GGATGCT	0b 10101101 11110000 01110100 11010000	2	-2	2	-2	0	0	-2	0	0	2	2	2	0	0	0	0
GATGCTA	0b 01000010 01001011 11011001 10011011	1	-1	1	-3	-1	-1	-1	-1	-1	3	1	1	1	-1	1	1
ATGCTAC	0b 11001100 01110101 01010011 00100110	2	0	0	-4	0	0	-2	-2	-2	4	2	2	0	0	0	2
TGCTACA	0b 10110001 01101010 10101001 10100111	3	-1	1	-3	-1	-1	-3	-1	-3	5	3	1	1	-1	1	1
GCTACAG	0b 11000101 11010111 11010101 00100101	4	0	0	-4	-2	0	-4	0	-2	6	2	2	0	0	2	2
CTACAGT	0b 11001001 01111101 01001010 10110101	5	1	-1	-5	-1	-1	-5	1	-3	7	3	3	1	1	1	3
TACAGTA	0b 00101011 01101111 11111000 11111000	4	0	0	-6	0	-2	-4	2	-4	8	4	2	2	2	2	4
ACAGTAT	0b 11100100 01001110 01110101 00011010	5	1	1	-7	-1	-1	-5	1	-5	9	3	1	3	3	3	3
CAGTATA	0b 10010010 00001101 00100011 10110100	6	0	0	-6	-2	-2	-4	0	-6	8	2	0	4	4	2	4
AGTATAT	0b 01110100 00110000 10101100 00000000	5	1	1	-5	-3	-1	-5	-1	-7	7	3	1	3	3	1	3
GTATATA	0b 11001111 10110000 11001001 10010110	6	2	0	-6	-2	0	-4	0	-6	6	4	2	2	2	0	2
TATATAC	0b 10000001 00001000 00101111 01111111	7	1	-1	-7	-3	-1	-5	1	-7	5	3	1	3	1	-1	1
ATATACC	0b 11001100 11100000 00101000 11011010	8	2	-2	-8	-2	0	-6	0	-6	6	4	0	2	0	-2	0
TATACCA	0b 00110100 00000100 11110100 10010100	7	1	-1	-7	-3	1	-7	-1	-7	5	3	-1	1	1	-3	-1
		B[31]	B[30]	B[29]	B[28]	B[27]	B[26]	B[25]	B[24]	B[23]	B[22]	B[21]	B[20]	B[19]	B[18]	B[17]	B[16]
		1	1	0	0	0	1	0	0	0	1	1	0	1	1	0	0

Best results are highlighted with **bold** text.

Table S2: Hash Values of the k-mers of seed S_k: `CGGATGCTACAGTATATACCA` for $k = 7$ and $n = 15$. We show the least significant 16 bits of the counter vector $C(S_k)$. Last row shows the least significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[15]	C[14]	C[13]	C[12]	C[11]	C[10]	C[9]	C[8]	C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]
CGGATGC	0b 10100000 01111111 10000110 10110101	1	-1	-1	-1	-1	1	1	-1	1	-1	1	1	-1	1	-1	1
GGATGCT	0b 10101101 11110000 01110100 11010000	0	0	0	0	-2	2	0	-2	2	0	0	2	-2	0	-2	0
GATGCTA	0b 01000010 01001011 11011001 10011011	1	1	-1	1	-1	1	-1	-1	3	-1	-1	3	-1	-1	-1	1
ATGCTAC	0b 11001100 01110101 01010011 00100110	0	2	-2	2	-2	0	0	0	2	-2	0	2	-2	0	0	0
TGCTACA	0b 10110001 01101010 10101001 10100111	1	1	-1	1	-1	-1	-1	1	3	-3	1	1	-3	1	1	1
GCTACAG	0b 11000101 11010111 11010101 00100101	2	2	-2	2	-2	0	-2	2	2	-4	2	0	-4	2	0	2
CTACAGT	0b 11001001 01111101 01001010 10110101	1	3	-3	1	-1	-1	-1	1	3	-5	3	1	-5	3	-1	3
TACAGTA	0b 00101011 01101111 11111000 11111000	2	4	-2	2	0	-2	-2	0	4	-4	4	2	-4	2	-2	2
ACAGTAT	0b 11100100 01001110 01110101 00011010	1	5	-1	3	-1	-1	-3	1	3	-5	3	3	-3	1	-1	1
CAGTATA	0b 10010010 00001101 00100011 10110100	0	4	0	2	-2	-2	-2	2	4	-6	4	4	-4	2	-2	0
AGTATAT	0b 01110100 00110000 10101100 00000000	1	3	1	1	-1	-1	-3	1	3	-7	3	3	-5	1	-3	-1
GTATATA	0b 11001111 10110000 11001001 10010110	2	4	0	0	0	-2	-4	2	4	-8	2	4	-6	2	-2	-2
TATATAC	0b 10000001 00001000 00101111 01111111	1	3	1	-1	1	-1	-3	3	3	-7	3	5	-5	3	-1	-1
ATATACC	0b 11001100 11100000 00101000 11011010	0	2	2	-2	2	-2	-4	2	4	-6	2	6	-4	2	0	-2
TATACCA	0b 00110100 00000100 11110100 10010100	1	3	3	-1	1	-1	-5	1	5	-7	1	7	-5	3	-1	-3
		B[15]	B[14]	B[13]	B[12]	B[11]	B[10]	B[9]	B[8]	B[7]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]
		1	1	1	0	1	0	0	1	1	0	1	1	0	1	0	0

Best results are highlighted with **bold** text.

Table S3: Hash Values of the k-mers of seed S_l: `ATGCTACAGTATATACCATCT` for $k = 7$ and $n = 15$. We show the most significant 16 bits of the counter vector $C(S_l)$. Last row shows the most significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[31]	C[30]	C[29]	C[28]	C[27]	C[26]	C[25]	C[24]	C[23]	C[22]	C[21]	C[20]	C[19]	C[18]	C[17]	C[16]
ATGCTAC	0b 11001100 01110101 01010011 00100110	1	1	-1	-1	1	1	-1	-1	-1	1	1	1	-1	1	-1	1
TGCTACA	0b 10110001 01101010 10101001 10100111	2	0	0	0	0	0	-2	0	-2	2	2	0	0	0	0	0
GCTACAG	0b 11000101 11010111 11010101 00100101	3	1	-1	-1	-1	1	-3	1	-1	3	1	1	-1	1	1	1
CTACAGT	0b 11001001 01111101 01001010 10110101	4	2	-2	-2	0	0	-4	2	-2	4	2	2	0	2	0	2
TACAGTA	0b 00101011 01101111 11111000 11111000	3	1	-1	-3	1	-1	-3	3	-3	5	3	1	1	3	1	3
ACAGTAT	0b 11100100 01001110 01110101 00011010	4	2	0	-4	0	0	-4	2	-4	6	2	0	2	4	2	2
CAGTATA	0b 10010010 00001101 00100011 10110100	5	1	-1	-3	-1	-1	-3	1	-5	5	1	-1	3	5	1	3
AGTATAT	0b 01110100 00110000 10101100 00000000	4	2	0	-2	-2	0	-4	0	-6	4	2	0	2	4	0	2
GTATATA	0b 11001111 10110000 11001001 10010110	5	3	-1	-3	-1	1	-3	1	-5	3	3	1	1	3	-1	1
TATATAC	0b 10000001 00001000 00101111 01111111	6	2	-2	-4	-2	0	-4	2	-6	2	2	0	2	2	-2	0
ATATACC	0b 11001100 11100000 00101000 11011010	7	3	-3	-5	-1	1	-5	1	-5	3	3	-1	1	1	-3	-1
TATACCA	0b 00110100 00000100 11110100 10010100	6	2	-2	-4	-2	2	-6	0	-6	2	2	-2	0	2	-4	-2
ATACCAT	0b 00000111 10111111 11010101 01001100	5	1	-3	-5	-3	3	-5	1	-5	1	3	-1	1	3	-3	-1
TACCATC	0b 01010110 11100111 00100010 11001101	4	2	-4	-4	-4	4	-4	0	-4	2	4	-2	0	4	-2	0
ACCATCT	0b 00010010 11001000 11001010 11100111	3	1	-5	-3	-5	3	-3	-1	-3	3	3	-3	1	3	-3	-1
		B[31]	B[30]	B[29]	B[28]	B[27]	B[26]	B[25]	B[24]	B[23]	B[22]	B[21]	B[20]	B[19]	B[18]	B[17]	B[16]
		1	1	0	0	0	1	0	0	0	1	1	0	1	1	0	0

Best results are highlighted with **bold** text.

Table S4: Hash Values of the k-mers of seed S_l: `ATGCTACAGTATATACCATCT` for $k = 7$ and $n = 15$. We show the least significant 16 bits of the counter vector $C(S_l)$. Last row shows the least significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[15]	C[14]	C[13]	C[12]	C[11]	C[10]	C[9]	C[8]	C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]
ATGCTAC	0b 11001100 01110101 01010011 00100110	-1	1	-1	1	-1	-1	1	1	-1	-1	1	-1	-1	1	1	-1
TGCTACA	0b 10110001 01101010 10101001 10100111	0	0	0	0	0	-2	0	2	0	-2	2	-2	-2	2	2	0
GCTACAG	0b 11000101 11010111 11010101 00100101	1	1	-1	1	-1	-1	-1	3	-1	-3	3	-3	-3	3	1	1
CTACAGT	0b 11001001 01111101 01001010 10110101	0	2	-2	0	0	-2	0	2	0	-4	4	-2	-4	4	0	2
TACAGTA	0b 00101011 01101111 11111000 11111000	1	3	-1	1	1	-3	-1	1	1	-3	5	-1	-3	3	-1	1
ACAGTAT	0b 11100100 01001110 01110101 00011010	0	4	0	2	0	-2	-2	2	0	-4	4	0	-2	2	0	0
CAGTATA	0b 10010010 00001101 00100011 10110100	-1	3	1	1	-1	-3	-1	3	1	-5	5	1	-3	3	-1	-1
AGTATAT	0b 01110100 00110000 10101100 00000000	0	2	2	0	0	-2	-2	2	0	-6	4	0	-4	2	-2	-2
GTATATA	0b 11001111 10110000 11001001 10010110	1	3	1	-1	1	-3	-3	3	1	-7	3	1	-5	3	-1	-3
TATATAC	0b 10000001 00001000 00101111 01111111	0	2	2	-2	2	-2	-2	4	0	-6	4	2	-4	4	0	-2
ATATACC	0b 11001100 11100000 00101000 11011010	-1	1	3	-3	3	-3	-3	3	1	-5	3	3	-3	3	1	-3
TATACCA	0b 00110100 00000100 11110100 10010100	0	2	4	-2	2	-2	-4	2	2	-6	2	4	-4	4	0	-4
ATACCAT	0b 00000111 10111111 11010101 01001100	1	3	3	-1	1	-1	-5	3	1	-5	1	3	-3	5	-1	-5
TACCATC	0b 01010110 11100111 00100010 11001101	0	2	4	-2	0	-2	-4	2	2	-4	0	2	-2	6	-2	-4
ACCATCT	0b 00010010 11001000 11001010 11100111	1	3	3	-3	1	-3	-3	1	3	-3	1	1	-3	7	-1	-3
		B[15]	B[14]	B[13]	B[12]	B[11]	B[10]	B[9]	B[8]	B[7]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]
		1	1	1	0	1	0	0	1	1	0	1	1	0	1	0	0

Best results are highlighted with **bold** text.

Table S5: Hash Values of the k-mers of seed S_k: `CGGATGCTACAGTATATACCA` for $k = 15$ and $n = 7$. We show the most significant 16 bits of the counter vector $C(S_k)$. Last row shows the most significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[31]	C[30]	C[29]	C[28]	C[27]	C[26]	C[25]	C[24]	C[23]	C[22]	C[21]	C[20]	C[19]	C[18]	C[17]	C[16]
CGGATGCTACAGTAT	0b 01001010 11101011 00100110 11001101	-1	1	-1	-1	1	-1	1	-1	1	1	1	-1	1	-1	1	1
GGATGCTACAGTATA	0b 01101100 01000011 11111000 11000000	-2	2	0	-2	2	0	0	-2	0	2	0	-2	0	-2	2	2
GATGCTACAGTATAT	0b 01011000 01000101 00110001 11011000	-3	3	-1	-1	3	-1	-1	-3	-1	3	-1	-3	-1	-1	1	3
ATGCTACAGTATATA	0b 11100001 01110100 01100010 01000010	-2	4	0	-2	2	-2	-2	-2	-2	4	0	-2	-2	0	0	2
TGCTACAGTATATAC	0b 10111100 10011010 00111111 01011011	-1	3	1	-1	3	-1	-3	-3	-1	3	-1	-1	-1	-1	1	1
GCTACAGTATATACC	0b 11101010 01001100 01000100 11100001	0	2	2	-2	4	-2	-2	-4	-2	4	-2	-2	0	0	0	0
CTACAGTATATACCA	0b 00101001 10010001 11111100 01010000	-1	1	3	-3	5	-3	-3	-3	-1	3	-3	-1	-1	-1	-1	1
Seed CGGATGCTACAGTATATACCA		B[31]	B[30]	B[29]	B[28]	B[27]	B[26]	B[25]	B[24]	B[23]	B[22]	B[21]	B[20]	B[19]	B[18]	B[17]	B[16]
		0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	1

Best results are highlighted with **bold** text.

Table S6: Hash Values of the k-mers of seed S_k: `CGGATGCTACAGTATATACCA` for $k = 15$ and $n = 7$. We show the least significant 16 bits of the counter vector $C(S_k)$. Last row shows the least significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[15]	C[14]	C[13]	C[12]	C[11]	C[10]	C[9]	C[8]	C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]
CGGATGCTACAGTAT	0b 01001010 11101011 00100110 11001101	-1	-1	1	-1	-1	1	1	-1	1	1	-1	-1	1	1	-1	1
GGATGCTACAGTATA	0b 01101100 01000011 11111000 11000000	0	0	2	0	0	0	0	-2	2	2	-2	-2	0	0	-2	0
GATGCTACAGTATAT	0b 01011000 01000101 00110001 11011000	-1	-1	3	1	-1	-1	-1	-1	3	3	-3	-1	1	-1	-3	-1
ATGCTACAGTATATA	0b 11100001 01110100 01100010 01000010	-2	0	4	0	-2	-2	0	-2	2	4	-4	-2	0	-2	-2	-2
TGCTACAGTATATAC	0b 10111100 10011010 00111111 01011011	-3	-1	5	1	-1	-1	1	-1	1	5	-5	-1	1	-3	-1	-1
GCTACAGTATATACC	0b 11101010 01001100 01000100 11100001	-4	0	4	0	-2	0	0	-2	2	6	-4	-2	0	-4	-2	0
CTACAGTATATACCA	0b 00101001 10010001 11111100 01010000	-3	1	5	1	-1	1	-1	-3	1	7	-5	-1	-1	-5	-3	-1
Seed CGGATGCTACAGTATATACCA		B[15]	B[14]	B[13]	B[12]	B[11]	B[10]	B[9]	B[8]	B[7]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]
		0	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0

Best results are highlighted with **bold** text.

Table S7: Hash Values of the k-mers of seed S_l: `ATGCTACAGTATATACCATCT` for $k = 15$ and $n = 7$. We show the most significant 16 bits of the counter vector $C(S_l)$. Last row shows the most significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[31]	C[30]	C[29]	C[28]	C[27]	C[26]	C[25]	C[24]	C[23]	C[22]	C[21]	C[20]	C[19]	C[18]	C[17]	C[16]
ATGCTACAGTATATA	0b 11100001 01110100 01100010 01000010	1	1	1	-1	-1	-1	-1	1	-1	1	1	1	-1	1	-1	-1
TGCTACAGTATATAC	0b 10111100 10011010 00111111 01011011	2	0	2	0	0	0	-2	0	0	0	0	2	0	0	0	-2
GCTACAGTATATACC	0b 11101010 01001100 01000100 11100001	3	1	3	-1	1	-1	-1	-1	-1	1	-1	1	1	1	-1	-3
CTACAGTATATACCA	0b 00101001 10010001 11111100 01010000	2	0	4	-2	2	-2	-2	0	0	0	-2	2	0	0	-2	-2
TACAGTATATACCAT	0b 00001110 00100000 11011100 11110110	1	-1	3	-3	3	-1	-1	-1	-1	-1	-1	1	-1	-1	-3	-3
ACAGTATATACCATC	0b 00101111 10111010 00010000 11011111	0	-2	4	-4	4	0	0	0	0	-2	0	2	0	-2	-2	-4
CAGTATATACCATCT	0b 01100101 11100111 10111011 00111011	-1	-1	5	-5	5	1	-1	1	1	-1	1	1	-1	-1	-1	-3
Seed ATGCTACAGTATATACCATCT		B[31]	B[30]	B[29]	B[28]	B[27]	B[26]	B[25]	B[24]	B[23]	B[22]	B[21]	B[20]	B[19]	B[18]	B[17]	B[16]
		0	0	1	0	1	1	0	1	1	0	1	1	0	0	0	0

Best results are highlighted with **bold** text.

Table S8: Hash Values of the k-mers of seed S_l: `ATGCTACAGTATATACCATCT` for $k = 15$ and $n = 7$. We show the least significant 16 bits of the counter vector $C(S_l)$. Last row shows the least significant 16 bits of the hash value of the seed.

K-mer	Hash Value	C[15]	C[14]	C[13]	C[12]	C[11]	C[10]	C[9]	C[8]	C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]
ATGCTACAGTATATA	0b 11100001 01110100 01100010 01000010	-1	1	1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	-1	1	-1
TGCTACAGTATATAC	0b 10111100 10011010 00111111 01011011	-2	0	2	0	0	0	2	0	-2	2	-2	0	0	-2	2	0
GCTACAGTATATACC	0b 11101010 01001100 01000100 11100001	-3	1	1	-1	-1	1	1	-1	-1	3	-1	-1	-1	-3	1	1
CTACAGTATATACCA	0b 00101001 10010001 11111100 01010000	-2	2	2	0	0	2	0	-2	-2	4	-2	0	-2	-4	0	0
TACAGTATATACCAT	0b 00001110 00100000 11011100 11110110	-1	3	1	1	1	3	-1	-3	-1	5	-1	1	-3	-3	1	-1
ACAGTATATACCATC	0b 00101111 10111010 00010000 11011111	-2	2	0	2	0	2	-2	-4	0	6	-2	2	-2	-2	2	0
CAGTATATACCATCT	0b 01100101 11100111 10111011 00111011	-1	1	1	3	1	1	-1	-3	-1	5	-1	3	-1	-3	3	1
Seed ATGCTACAGTATATACCATCT		B[15]	B[14]	B[13]	B[12]	B[11]	B[10]	B[9]	B[8]	B[7]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]
		0	1	1	1	1	1	0	0	0	1	0	1	0	0	1	1

Best results are highlighted with **bold** text.

S2. Parameter Exploration

S2.1. The trade-off between `BLEND-I` and `BLEND-S`

Our goal is to show the performance and accuracy trade-offs between the seeding techniques that BLEND supports: `BLEND-I` and `BLEND-S`. In Supplementary Figures S1 and S2, we show the performance and peak memory usage comparisons when using `BLEND-I` and `BLEND-S` as the seeding technique by keeping all the other relevant parameters identical (e.g., number of k-mers to include in a seed n, window length w). In Supplementary Table S9 we show the assembly quality comparisons in terms of the accuracy and contiguity of the assemblies that we generate using the overlaps that `BLEND-I` and `BLEND-S` find. In Supplementary Tables S10 and S11 we show the read mapping quality and accuracy results using these two seeding techniques, respectively.

We also show the values for different parameters we test with BLEND in Supplementary Table S12. We determine the default parameters of BLEND emprically based on the combination of best performance, memory overhead, and accuracy results.

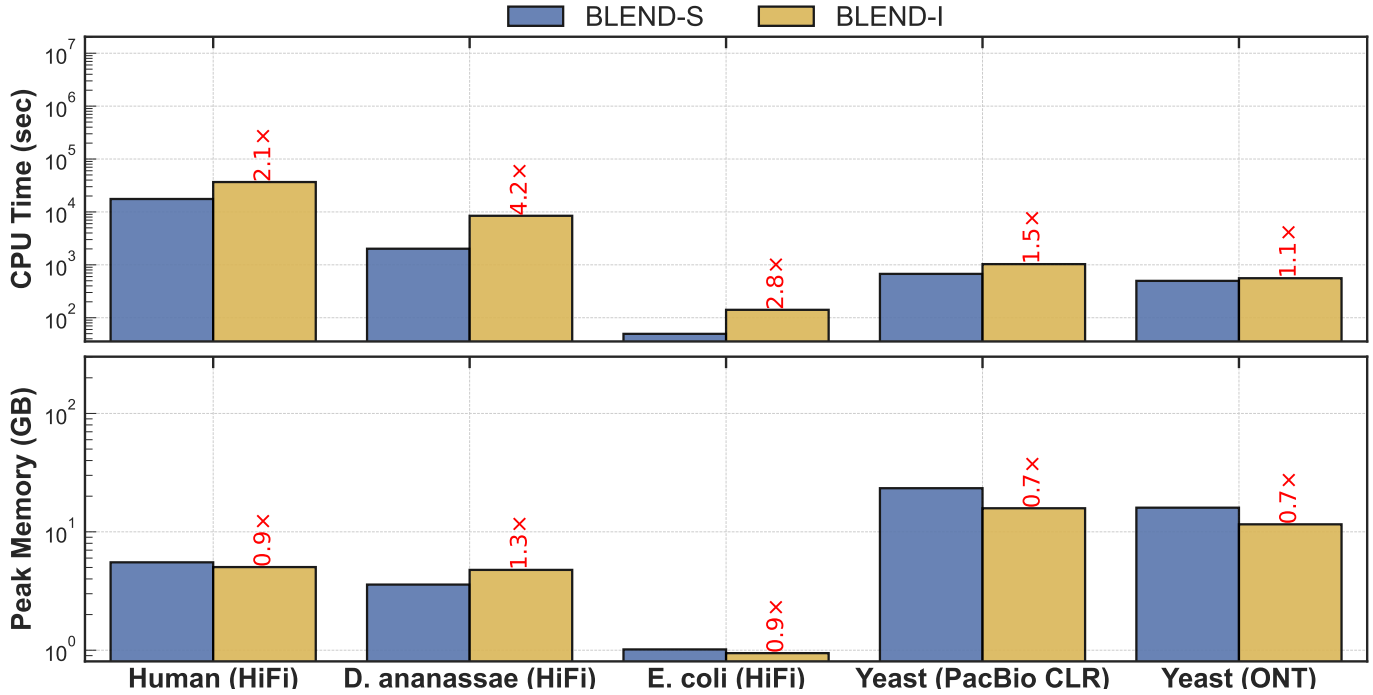

Figure S1: CPU time and peak memory footprint comparisons of read overlapping.

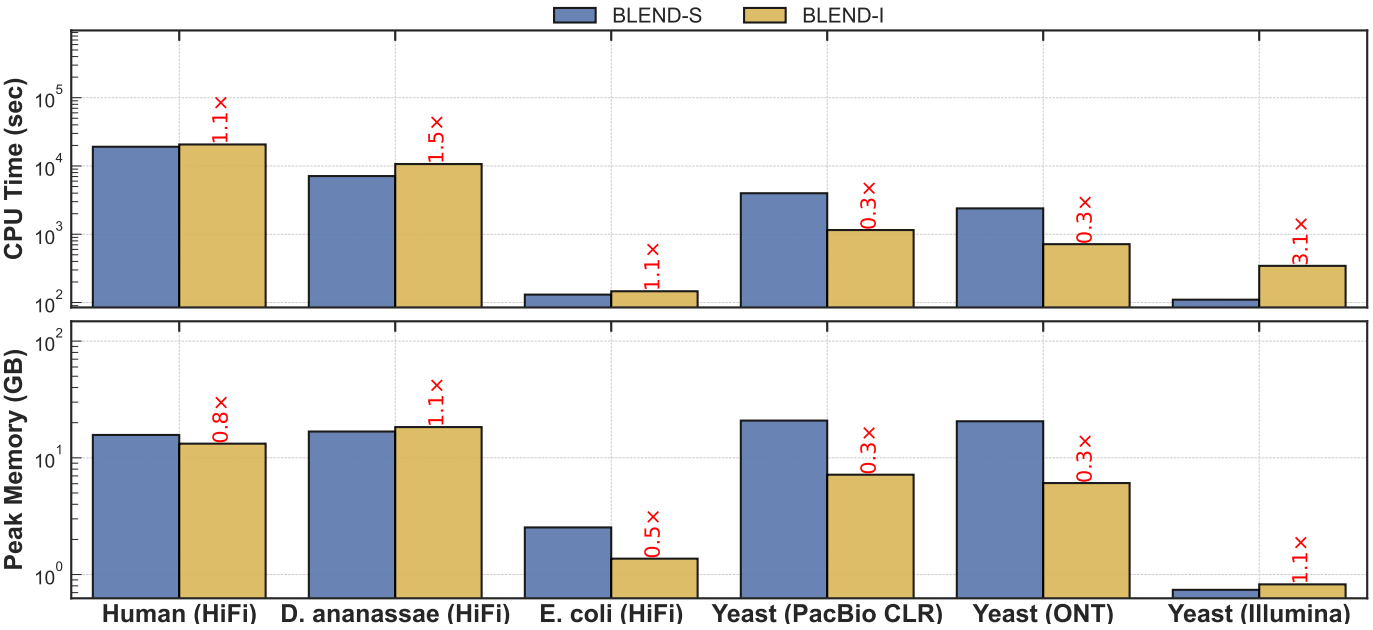

Figure S2: CPU time and peak memory footprint comparisons of read mapping.

Table S9: Assembly quality comparisons between `BLEND-I` and `BLEND-S`.

Dataset	Tool	Average Identity (%)	Genome Fraction (%)	K-mer Compl. (%)	Aligned Length (Mbp)	Misassembly Ratio (%)	NGA50 (Kbp)	Average GC (%)	Assembly Length (Mbp)	Largest Contig (Mbp)	NG50 (Kbp)
Human CHM13	BLEND-S	**99.8526**	**98.4847**	**90.15**	**3,092.59**	**0.0108**	5,442	40.78	**3,095.210**	22.840	5,442
	BLEND-I	99.7578	96.4306	84.07	2,990.41	0.0503	**8,802**	**40.84**	2,994.976	**41.834**	**9,005**
	Reference	100	100	100	3,054.83	0	154,260	40.85	3,054.832	248.387	154,260
D. ananassae	BLEND-S	**99.7856**	**97.2308**	**86.43**	240.39	0.1230	**792**	41.75	**247.153**	**6.233**	**799**
	BLEND-I	99.7088	97.0399	79.43	**248.99**	**0.1020**	363	**41.86**	258.483	4.439	363
	Reference	100	100	100	213.81	0	26,427	41.81	213.818	30.673	26,427
E. coli	BLEND-S	**99.8320**	99.8801	**87.91**	**5.12**	**0.0344**	3,417	50.53	5.122	3.417	3,417
	BLEND-I	99.7166	**99.8824**	80.37	5.04	0.9810	**4,025**	**50.52**	**5.043**	**4.946**	**4,946**
	Reference	100	100	100	5.05	0	4,945	50.52	5.046	4.946	4,945
Yeast (PacBio)	BLEND-S	**90.3347**	83.8814	**18.47**	**0.558**	N/A	N/A	**38.71**	22.952	0.265	116
	BLEND-I	89.1582	**97.6297**	11.13	0.227	N/A	N/A	38.80	**13.679**	**1.105**	**551**
	Reference	100	100	100	12.16	0	924	38.15	12.157	1.532	924
Yeast (ONT)	BLEND-S	**91.0865**	7.9798	1.57	0.066	N/A	N/A	**38.35**	0.900	0.043	N/A
	BLEND-I	89.7622	**99.2982**	**13.68**	**0.377**	N/A	N/A	38.66	**12.164**	**1.554**	**825**
	Reference	100	100	100	12.16	0	924	38.15	12.157	1.532	924

Best results are highlighted with **bold** text. For most metrics best results are the ones closest to the corresponding value of the reference genome.

The best results for *Aligned Length* are determined by the highest number within each dataset. We do not highlight the reference results as the best results.

N/A indicates that we could not generate the corresponding result because either the tool failed or QUAST failed to generate the statistic.

Table S10: Read mapping quality comparisons between `BLEND-I` and `BLEND-S`.

Dataset	Tool	Mean Depth of Cov. (×)	Breadth of Coverage (%)	Aligned Reads (#)	Properly Paired (%)
Human CHM13	BLEND-S	16.58	99.99	3,171,916	NA
	BLEND-I	16.58	99.99	**3,172,313**	NA
D. ananassae	BLEND-S	57.37	**99.66**	1,223,388	NA
	BLEND-I	**57.50**	99.65	**1,249,731**	NA
E. coli	BLEND-S	99.14	99.90	39,048	NA
	BLEND-I	99.14	99.90	**39,065**	NA
Yeast (PacBio)	BLEND-S	170.43	99.98	217,975	NA
	BLEND-I	**195.84**	99.98	**269,804**	NA
Yeast (ONT)	BLEND-S	75.74	99.93	95,847	NA
	BLEND-I	**97.86**	**99.97**	**134,721**	NA
Yeast (Illumina)	BLEND-S	79.63	99.97	6,466,782	95.28
	BLEND-I	**79.93**	99.97	**6,494,489**	**95.89**

Best results are highlighted with **bold** text.

Properly paired rate is only available for paired-end Illumina reads.

Table S11: Read mapping accuracy comparisons between `BLEND-I` and `BLEND-S`.

Dataset	Tool	Error Rate (%)	High Quality True Mappings (#)
Yeast (PacBio)	BLEND-S	0.4027985	206,243
	BLEND-I	**0.2524054**	**266,382**
Yeast (ONT)	BLEND-S	0.3328221	85,081
	BLEND-I	**0.2404970**	**133,228**

Best results are highlighted with **bold** text.

Table S12: Performance, memory, and accuracy comparisons using different parameter settings in BLEND.

Tool	K-mer Length (k)	# of k-mers in a Seed (n)	Window Length (w)	CPU Time (seconds)	Peak Memory (KB)	Average Identity (%)	Genome Fraction (%)
BLEND	9	11	200	62.38	1,115,384	99.7255	99.8502
BLEND	9	13	200	58.13	994,120	99.7294	99.7808
BLEND	9	15	200	49.79	1,030,148	99.7411	99.7619
BLEND	9	17	200	45.03	960,080	99.7302	99.7460
BLEND	9	21	200	36.84	976,456	99.7257	99.6640
BLEND	15	5	200	83.05	1,168,612	99.6735	99.7625
BLEND	15	7	200	74.93	1,137,360	99.7009	99.5874
BLEND	15	11	200	58.09	1,051,912	99.7149	99.1166
BLEND	19	5	200	77.16	1,130,604	99.7312	99.8802
BLEND	19	7	200	50.50	1,078,596	99.7880	99.8424
BLEND	19	11	200	46.26	977,060	99.8078	99.6438
BLEND	21	5	200	67.85	1,116,684	99.7472	99.8835
BLEND	21	7	200	61.63	1,042,724	99.7969	99.8605
BLEND	21	11	200	42.35	969,184	99.8340	99.7515
BLEND	25	5	200	65.61	1,057,804	99.7769	99.8818
BLEND	25	7	200	54.88	1,029,888	99.8320	99.8801
BLEND	25	11	200	37.01	936,260	99.8646	99.8001
BLEND	25	15	200	29.83	866,208	99.8838	99.7307
BLEND	25	17	200	29.59	826,456	99.8784	99.7521
BLEND	25	21	200	26.09	791,736	99.8774	99.6955
BLEND	9	11	50	263.82	1,786,516	99.7013	99.8612
BLEND	9	13	50	411.24	1,805,800	99.6995	99.8573
BLEND	9	15	50	271.00	1,729,784	99.6798	99.8517
BLEND	9	17	50	238.52	1,690,912	99.6690	99.8083
BLEND	9	21	50	206.76	1,725,168	99.6496	99.8150
BLEND	15	5	50	330.84	1,785,456	99.6634	99.8604
BLEND	15	7	50	337.95	1,812,052	99.6280	99.8177
BLEND	15	11	50	236.82	1,803,816	99.5831	99.6893
BLEND	19	5	50	328.67	1,692,248	99.7077	99.8794
BLEND	19	7	50	295.57	1,713,940	99.7188	99.8579
BLEND	19	11	50	201.79	1,700,412	99.7015	99.8578
BLEND	21	5	50	378.58	1,625,388	99.7120	99.8832
BLEND	21	7	50	278.56	1,695,476	99.7333	99.8832
BLEND	21	11	50	189.33	1,694,820	99.7623	99.8594
BLEND	25	5	50	323.69	1,685,304	99.7272	99.8831
BLEND	25	7	50	211.78	1,647,984	99.7722	99.8831
BLEND	25	11	50	170.60	1,683,736	99.8094	99.8866
BLEND	25	15	50	142.42	1,622,452	99.8170	99.8576
BLEND	25	17	50	103.96	1,590,776	99.8073	99.8206
BLEND	25	21	50	109.62	1,548,228	99.7792	99.7880
BLEND	9	11	20	837.50	2,769,552	99.6916	99.8784
BLEND	9	13	20	813.50	2,765,480	99.6834	99.8785
BLEND	9	15	20	764.91	2,795,848	99.6797	99.8756
BLEND	9	17	20	739.52	2,823,188	99.6801	99.8802

We use the *E.coli* dataset for all these runs

S2.2. The trade-off between BLEND and minimap2

Our goal is to compare BLEND and minimap2 using the same set of parameters that BLEND uses when generating its results. To achieve this, we control the following two conditions. First, we ensure that we use the same seeding technique that minimap2 uses. To this end, we use the `BLEND-I` seeding technique, which finds minimizer k-mers and uses them as seeds, similar to how minimap2 generates the seeds. We should note that `BLEND-I` does not always provide the best results in terms of performance or accuracy for the HiFi reads as the default seeding technique is `BLEND-S` for HiFi datasets in BLEND.

Second, we use the same seed length when we compare BLEND with minimap2. In minimap2, the seed length equals to the k-mer length as minimap2 directly hashes these k-mers and finds the minimizer k-mers from these hash values. However, the seed length in `BLEND-I` is determined by both the k-mer length and the number of k-mers that we include in a seed. For example, BLEND uses the `BLEND-I` seeding technique with the k-mer length $k = 7$ and the number of neighbors $n = 15$ for the PacBio CLR reads. Combining immediately overlapping 15-many 7-mers generates seeds with length $15+7-1 = 21$. Thus, `BLEND-I` uses seeds of length 21 while allowing fuzzy seeds to match with its hashing technique. We calculate the seed lengths that `BLEND-I` uses with its default parameters for each dataset: PacBio HiFi, PacBio CLR, ONT, and Illumina short reads in read overlapping and read mapping. We apply the same seed length and the window length that BLEND uses to minimap2 using the k and w parameters. We show these parameters in Supplementary Tables S17 and S18 in the Minimap-Eq rows. In the results we show below, Minimap-Eq indicates the runs of minimap2 when using the same set of parameters that BLEND uses with the `BLEND-I` seeding technique. We should note that BLEND uses seeds of length 31 (i.e., $k = 25$ and $n = 7$) for HiFi datasets in read overlapping while it is not possible to set a seed length larger than 28 in minimap2 due to the limitations in its implementation. Thus, for these datasets, we cannot perform the same experiment with minimap2.

In Supplementary Figures S3 and S4, we show the performance and peak memory usage comparisons when using BLEND with the `BLEND-I` seeding technique, minimap2, and minimap2-Eq. In Supplementary Table S13 we show the assembly quality comparisons in terms of the accuracy and contiguity of the assemblies that we generate using the overlaps that each tool finds. In Supplementary Tables S14 and S15 we show the read mapping quality and accuracy results, respectively.

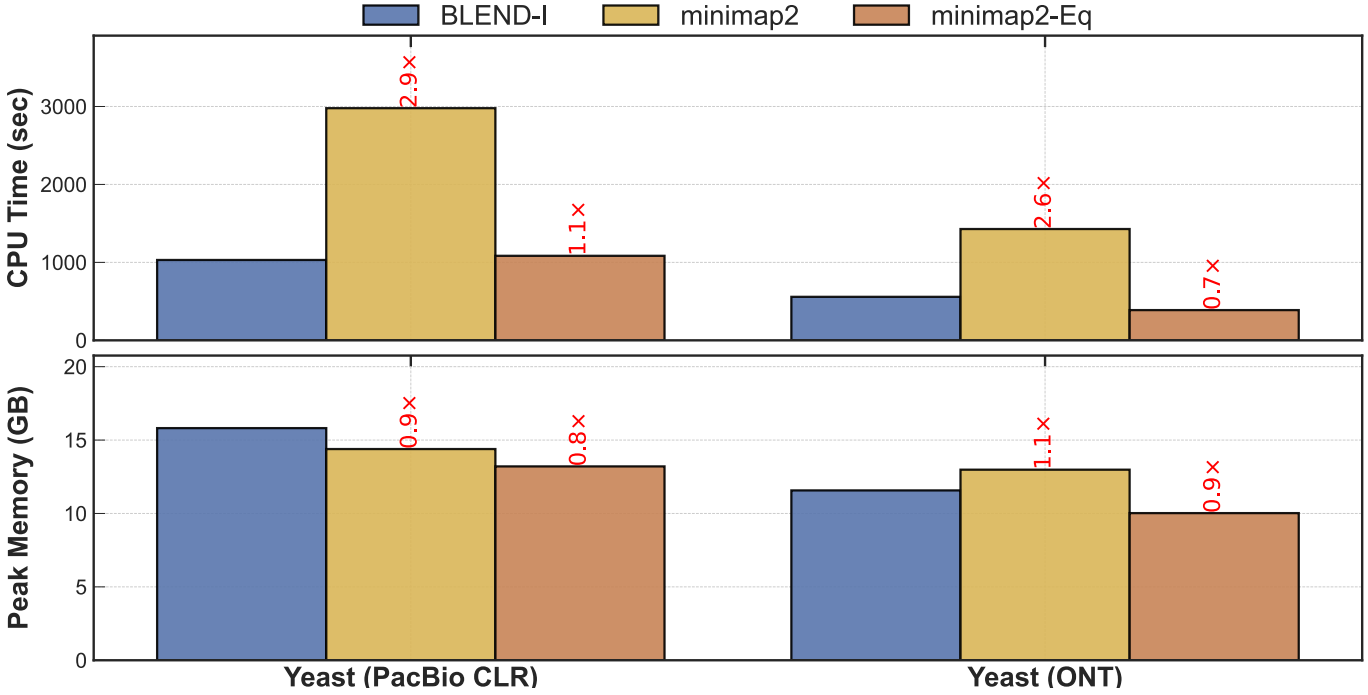

Figure S3: CPU time and peak memory footprint comparisons of read overlapping.

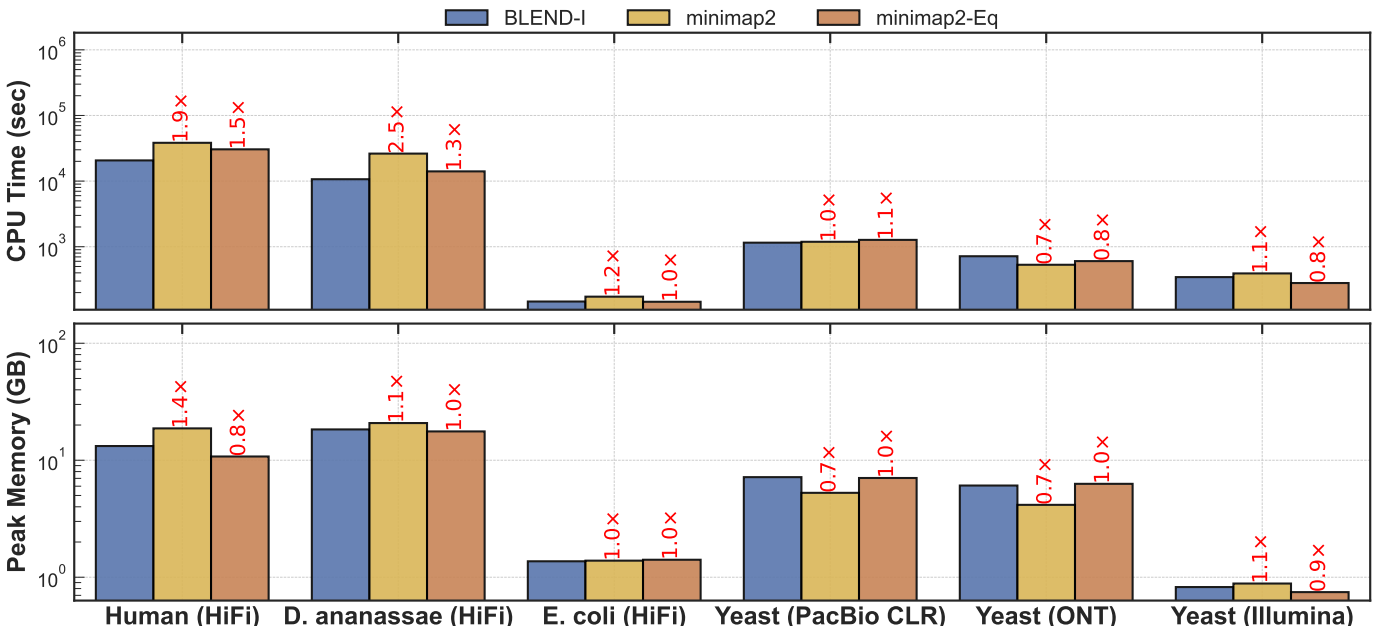

Figure S4: CPU time and peak memory footprint comparisons of read mapping.

Table S13: Assembly quality comparisons when using the parameters equivalent to `BLEND-I`.

Dataset	Tool	Average Identity (%)	Genome Fraction (%)	K-mer Compl. (%)	Aligned Length (Mbp)	Misassembly Ratio (%)	NGA50 (Kbp)	Average GC (%)	Assembly Length (Mbp)	Largest Contig (Mbp)	NG50 (Kbp)
Yeast (PacBio)	`BLEND-I`	89.1582	**97.6297**	**11.13**	**0.227**	N/A	N/A	**38.80**	13.679	1.105	551
	minimap2	88.9002	96.9709	9.74	0.195	N/A	N/A	38.85	**12.333**	**1.561**	**828**
	minimap2-Eq	**89.2166**	97.2674	10.48	0.215	N/A	N/A	38.82	12.424	1.534	781
	Reference	100	100	100	12.16	0	924	38.15	12.157	1.532	924
Yeast (ONT)	`BLEND-I`	89.7622	99.2982	**13.68**	**0.377**	N/A	N/A	38.66	**12.164**	1.554	825
	minimap2	88.9393	**99.6878**	12.06	0.328	N/A	N/A	38.74	12.373	**1.560**	**942**
	minimap2-Eq	**89.7653**	97.3273	13.39	0.371	N/A	N/A	**38.64**	11.828	1.073	677
	Reference	100	100	100	12.16	0	924	38.15	12.157	1.532	924

Best results are highlighted with **bold** text. For most metrics best results are the ones closest to the corresponding value of the reference genome.

The best results for *Aligned Length* are determined by the highest number within each dataset. We do not highlight the reference results as the best results.

N/A indicates that we could not generate the corresponding result because either the tool failed or QUAST failed to generate the statistic.

Table S14: Read mapping quality comparisons when using the parameters equivalent to `BLEND-I`.

Dataset	Tool	Mean Depth of Cov. (×)	Breadth of Coverage (%)	Aligned Reads (#)	Properly Paired (%)
Human CHM13	BLEND-I	16.58	99.99	**3,172,313**	NA
	minimap2	16.58	99.99	3,172,261	NA
	minimap2-Eq	16.58	99.99	3,172,312	NA
D. ananassae	BLEND-I	57.50	99.65	1,249,731	NA
	minimap2	**57.57**	**99.67**	1,245,931	NA
	minimap2-Eq	57.49	99.65	**1,250,816**	NA
E. coli	BLEND-I	99.14	99.90	39,065	NA
	minimap2	99.14	99.90	39,065	NA
	minimap2-Eq	99.14	99.90	39,065	NA
Yeast (PacBio)	BLEND-I	195.84	99.98	269,804	NA
	minimap2	**195.86**	99.98	**269,935**	NA
	minimap2-Eq	195.81	99.98	269,493	NA
Yeast (ONT)	BLEND-I	97.86	**99.97**	134,721	NA
	minimap2	**97.88**	99.96	**134,885**	NA
	minimap2-Eq	97.82	99.96	134,578	NA
Yeast (Illumina)	BLEND-I	**79.93**	99.97	**6,494,489**	**95.89**
	minimap2	79.91	99.97	6,492,994	**95.89**
	minimap2-Eq	79.83	99.97	6,485,540	95.72

Best results are highlighted with **bold** text.

Properly paired rate is only available for paired-end Illumina reads.

Table S15: Read mapping accuracy comparisons when using the parameters equivalent to `BLEND-I`.

Dataset	Tool	Error Rate (%)	High Quality True Mappings (#)
Yeast (PacBio)	BLEND-I	0.2524054	**266,382**
	minimap2	0.2504307	265,709
	Minimap-Eq	**0.2293195**	266,355
Yeast (ONT)	BLEND-I	**0.2404970**	133,228
	minimap2	0.2468770	**133,784**
	Minimap-Eq	0.2489263	132,641

Best results are highlighted with **bold** text.

S3. GC Content Distribution for Assembly Quality Assessment

In Figure S5, we show the GC content distribution of assemblies from each dataset that we generate using the overlapping reads from BLEND, minimap2 [7], and MHAP [1].

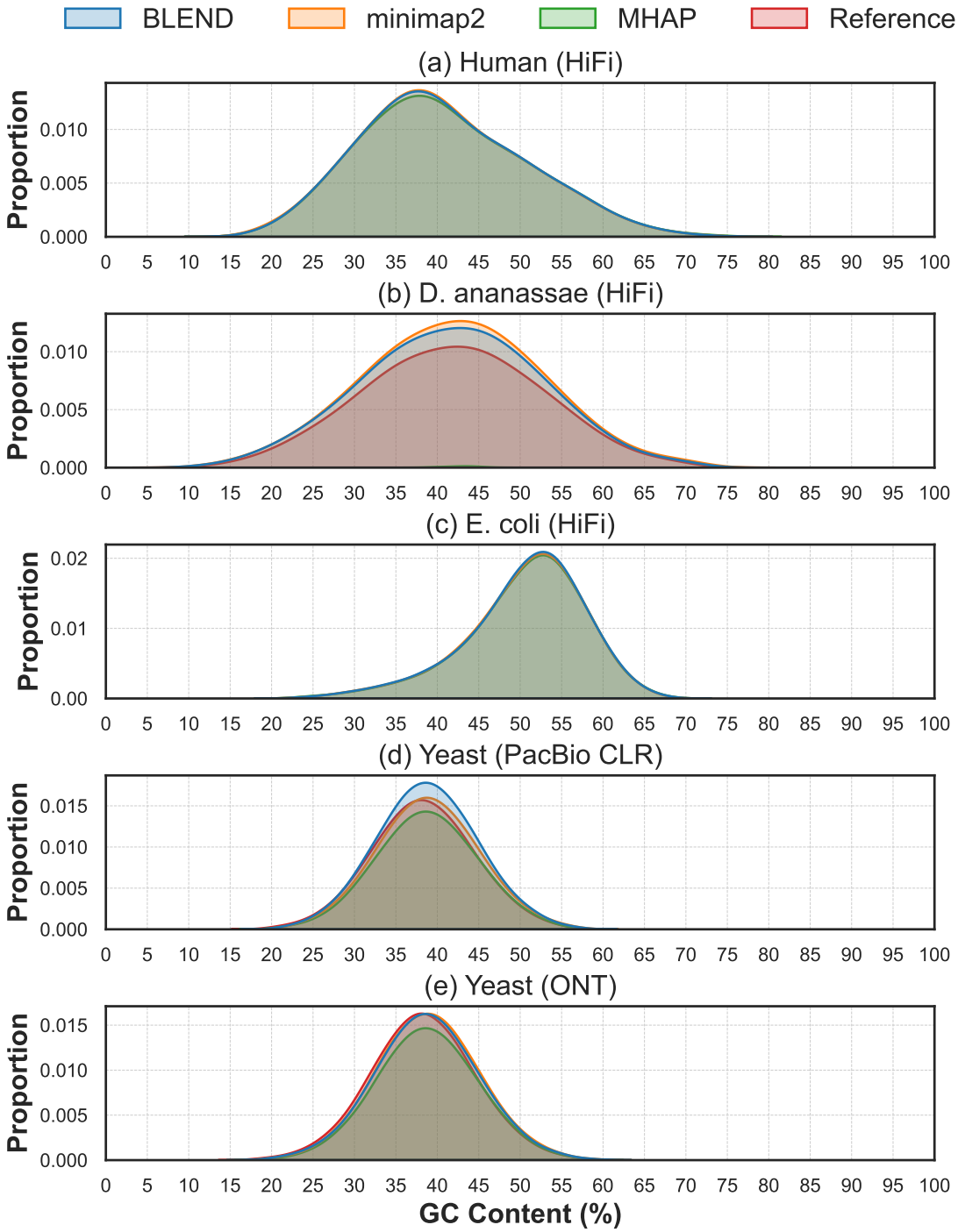

Figure S5: GC content distribution of assemblies.

S4. Read Mapping Accuracy

In Supplementary Table S16, we show the accuracy results of each read mapping tool as provided by the `paftools mapeval`, which is a script included in the GitHub page minimap2 [7]: `https://github.com/lh3/minimap2/tree/master/misc`. In order to generate these accuracy results, we first use PBSIM2 [10] to generate the simulated reads along with their true mapping locations. Then, the `paftools pbsim2fq` tool takes the true mapping information from PBSIM2 and generates the reads such that the read IDs in the generated FASTA file are annotated with the true mapping information. Then, `paftools mapeval` uses this annotated information to measure the read mapping accuracy by comparing the mapping result that each tools generates with the true mapping location. For more detailed information we refer to the GitHub page of paftools provided above.

Table S16: Read mapping accuracy comparisons.

Dataset	Tool	Error Rate (%)	High Quality True Mappings (#)
Yeast (PacBio)	BLEND	0.2524054	**266,382**
	minimap2	0.2504307	265,709
	LRA	99.9958863	11
	Winnowmap2	**0.2474206**	247,024
	S-conLSH	97.2181306	3,822
Yeast (ONT)	BLEND	**0.2404970**	133,228
	minimap2	0.2468770	**133,784**
	LRA	99.9954840	5
	Winnowmap2	0.2534777	126,899
	S-conLSH	96.6753628	2,361

Best results are highlighted with **bold** text.

S5. Parameters and Versions

In Supplementary Table S17, we show the parameters we use with BLEND, minimap2, and MHAP [1] for read overlapping. Since there are no default parameters for minimap2 and MHAP when using the HiFi reads, we used the parameters as suggested by the HiCanu tool [9]. We found these parameters in the source code of Canu. For minimap2 and MHAP, the HiFi parameters are found in the GitHub pages[1,2], respectively. In Supplementary Table S18, we show the parameters we use with BLEND, minimap2 [7], LRA [11], Winnowmap2 [5, 6], and S-conLSH [2, 3] for read mapping. In both Supplementary Tables S17 and S18, *minimap-Eq* shows the parameters that are equivalent to the parameters we use with BLEND without the fuzzy seed matching capability. In Supplementary Table S19, we show the version numbers of each tool we use. When calculating the performance and peak memory usage of each tool we use the time command from Linux and append the following command to the beginning of each of our runs: `/usr/bin/time -vp`.

[1] `https://github.com/marbl/canu/blob/404540a944664cfab00617f4f4fa37be451b34e0/src/pipelines/canu/OverlapMMap.pm#L63-L65`

[2] `https://github.com/marbl/canu/blob/404540a944664cfab00617f4f4fa37be451b34e0/src/pipelines/canu/OverlapMhap.pm#L100-L131`

Table S17: Parameters we use in our evaluation for each tool and dataset in read overlapping.

Tool	Dataset	Parameters
BLEND	*Human CHM13*	-x ava-hifi -t 32 (ava-hifi: –strobemers -k25 -w200 –neighbors 7)
BLEND	*D. ananassae*	-x ava-hifi -t 32
BLEND	*E. coli*	-x ava-hifi -t 32
BLEND	*Yeast PacBio CLR*	-x ava-pb -t 32 (ava-pb: -Hk19 -w10 –neighbors 5)
BLEND	*Yeast PacBio ONT*	-x ava-ont -t 32 (ava-ont: -k15 -w10 –neighbors 5)
minimap2	*Human CHM13*	-x ava-pb -Hk21 -w14 -t 32
minimap2	*D. ananassae*	-x ava-pb -Hk21 -w14 -t 32
minimap2	*E. coli*	-x ava-pb -Hk21 -w14 -t 32
minimap2	*Yeast PacBio CLR*	-x ava-pb -t 32
minimap2	*Yeast PacBio ONT*	-x ava-ont -t 32
minimap2-Eq	*Yeast PacBio CLR*	-x ava-pb -k23 -w10 -t 32
minimap2-Eq	*Yeast PacBio ONT*	-x ava-ont -k19 -w10 -t 32
MHAP	*Human CHM13*	–store-full-id –ordered-kmer-size 18 –num-hashes 128 –num-min-matches 5 –ordered-sketch-size 1000 –threshold 0.95 –num-threads 32
MHAP	*D. ananassae*	–store-full-id –ordered-kmer-size 18 –num-hashes 128 –num-min-matches 5 –ordered-sketch-size 1000 –threshold 0.95 –num-threads 32
MHAP	*E. coli*	–store-full-id –ordered-kmer-size 18 –num-hashes 128 –num-min-matches 5 –ordered-sketch-size 1000 –threshold 0.95 –num-threads 32
MHAP	*Yeast PacBio CLR*	–store-full-id –num-threads 32
MHAP	*Yeast PacBio ONT*	–store-full-id –num-threads 32

Table S18: Parameters we use in our evaluation for each tool and dataset in read mapping.

Tool	Dataset	Parameters
BLEND	*Human CHM13*	-ax map-hifi -t 32 –secondary=no (map-hifi: –strobemers -k19 -w50 –neighbors 5)
BLEND	*D. ananassae*	-ax map-hifi -t 32 –secondary=no
BLEND	*E. coli*	-ax map-hifi -t 32 –secondary=no
BLEND	*Yeast PacBio CLR*	-ax map-pb -t 32 –secondary=no (map-pb: -Hk7 -w10 –neighbors 15)
BLEND	*Yeast PacBio ONT*	-ax map-ont -t 32 –secondary=no (map-ont: -k7 -w10 –neighbors 11)
BLEND	*Yeast Illumina*	-ax sr -t 32
minimap2	*Human CHM13*	-ax map-hifi -t 32 –secondary=no
minimap2	*D. ananassae*	-ax map-hifi -t 32 –secondary=no
minimap2	*E. coli*	-ax map-hifi -t 32 –secondary=no
minimap2	*Yeast PacBio CLR*	-ax map-pb -t 32 –secondary=no
minimap2	*Yeast PacBio ONT*	-ax map-ont -t 32 –secondary=no
minimap2	*Yeast Illumina*	-ax sr -t 32
minimap2-Eq	*Human CHM13*	-ax map-hifi -k23 -w50 -t 32 –secondary=no
minimap2-Eq	*D. ananassae*	-ax map-hifi -k23 -w50 -t 32 –secondary=no
minimap2-Eq	*E. coli*	-ax map-hifi -k23 -w50 -t 32 –secondary=no
minimap2-Eq	*Yeast PacBio CLR*	-ax map-pb -k21 -w10 -t 32 –secondary=no
minimap2-Eq	*Yeast PacBio ONT*	-ax map-ont -k17 -w10 -t 32 –secondary=no
minimap2-Eq	*Yeast Illumina*	-ax sr -k25 -w11 -t 32
Winnowmap2	*Human CHM13*	meryl count k=15 meryl print greater-than distinct=0.9998 -ax map-pb -t 32
Winnowmap2	*D. ananassae*	meryl count k=15 meryl print greater-than distinct=0.9998 -ax map-pb -t 32
Winnowmap2	*E. coli*	meryl count k=15 meryl print greater-than distinct=0.9998 -ax map-pb -t 32
Winnowmap2	*Yeast PacBio CLR*	meryl count k=15 meryl print greater-than distinct=0.9998 -ax map-pb-clr -t 32
Winnowmap2	*Yeast PacBio ONT*	meryl count k=15 meryl print greater-than distinct=0.9998 -ax map-ont -t 32
LRA	*Human CHM13*	align -CCS -t 32 -p s
LRA	*D. ananassae*	align -CCS -t 32 -p s
LRA	*E. coli*	align -CCS -t 32 -p s
LRA	*Yeast PacBio CLR*	align -CLR -t 32 -p s
LRA	*Yeast PacBio ONT*	align -ONT -t 32 -p s
S-conLSH	*Human CHM13*	–threads 32 –align 1
S-conLSH	*E. coli*	–threads 32 –align 1
S-conLSH	*Yeast PacBio CLR*	–threads 32 –align 1
S-conLSH	*Yeast PacBio ONT*	–threads 32 –align 1

Table S19: Versions of each tool.

Tool	Version	GitHub or Conda Link to the Version
BLEND	1.0	https://github.com/CMU-SAFARI/BLEND
minimap2	2.24	https://github.com/lh3/minimap2/releases/tag/v2.24
MHAP	2.1.3	https://anaconda.org/bioconda/mhap/2.1.3/download/noarch/mhap-2.1.3-hdfd78af_1.tar.bz2
LRA	1.3.2	https://anaconda.org/bioconda/lra/1.3.2/download/linux-64/lra-1.3.2-ha140323_0.tar.bz2
Winnowmap2	2.03	https://anaconda.org/bioconda/Winnowmap/2.03/download/linux-64/Winnowmap2-2.03-h2e03b76_0.tar.bz2
S-conLSH	2.0	https://github.com/anganachakraborty/S-conLSH-2.0/tree/292fbe0405f10b3ab63fc3a86cba2807597b582e

Supplementary References

[1] K. Berlin *et al.*, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature Biotechnology*, vol. 33, no. 6, pp. 623–630, Jun. 2015.

[2] A. Chakraborty and S. Bandyopadhyay, "conLSH: Context based Locality Sensitive Hashing for mapping of noisy SMRT reads," *Computational Biology and Chemistry*, vol. 85, p. 107206, Apr. 2020.

[3] A. Chakraborty *et al.*, "S-conLSH: alignment-free gapped mapping of noisy long reads," *BMC Bioinformatics*, vol. 22, no. 1, p. 64, Feb. 2021.

[4] M. S. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 380–388.

[5] C. Jain *et al.*, "Long-read mapping to repetitive reference sequences using Winnowmap2," *Nature Methods*, Apr. 2022.

[6] C. Jain *et al.*, "Weighted minimizer sampling improves long read mapping," *Bioinformatics*, vol. 36, no. Supplement_1, pp. i111–i118, Jul. 2020.

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

[8] G. S. Manku *et al.*, "Detecting Near-Duplicates for Web Crawling," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 141–150.

[9] S. Nurk *et al.*, "HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads," *bioRxiv*, p. 2020.03.14.992248, Jan. 2020.

[10] Y. Ono *et al.*, "PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores," *Bioinformatics*, vol. 37, no. 5, pp. 589–595, Mar. 2021.

[11] J. Ren and M. J. P. Chaisson, "lra: A long read aligner for sequences and contigs," *PLOS Computational Biology*, vol. 17, no. 6, p. e1009078, Jun. 2021.