An optimized FM-index library for nucleotide and amino acid search

Tim Anderson[1], Travis J Wheeler[1*],

1 Department of Computer Science , University of Montana, Missoula, MT, USA

* travis.wheeler@umontana.edu

Abstract

Pattern matching is a key step in a variety of biological sequence analysis pipelines. The FM-index is a compressed data structure for pattern matching, with search run time that is independent of the length of the database text. We present AvxWindowedFMindex (AWFM-index), an open-source, thread-parallel FM-index library written in C that is highly optimized for indexing nucleotide and amino acid sequences. AWFM-index is easy to incorporate into bioinformatics software and is able to perform exact match count and locate queries approximately 4x faster than Seqan3's FM-index implementation for nucleotide search, and approximately 8x faster for amino acid search in a single-threaded context. This performance is due to (i) a new approach to storing FM-index data in a strided bit-vector format that enables extremely efficient computation of the FM-index occurrence function via AVX2 bitwise instructions, and (ii) inclusion of a cache-efficient lookup table for partial k-mer searches. AWFM-index also trivially parallelizes to multiple threads, and scales well in multithreaded contexts. The open-source library is available for download at https://github.com/TravisWheelerLab/AvxWindowFmIndex.

Author summary

AvxWindowedFMIndex is a fast, open-source library implementation of the FM-index algorithm. This library takes advantage of powerful 'single-instruction, multiple data' (SIMD) CPU instructions to quickly perform the most difficult part of the algorithm, counting the number of occurrences of a given letter in a block of text. Algorithms like FM-index are widely used many places in bioinformatics like biosequence database searching, taxonomic classification, and sequencing error correction. Using the AvxWindowedFMIndex library will ease the burden of including the FM index into bioinformatic software, thus enabling faster pattern matching and overall faster software in practice.

Introduction

String pattern matching is the problem of counting or locating occurrences of a query text pattern p within a large database text T. While not limited to the analysis of biological sequences, string pattern matching is a problem integral to many tasks in bioinformatics, including mapping sequences of reads to a reference genome [1, 2], taxonomic classification [3, 4], sequencing error correction [5], and seeding for sequence alignments [6–8].

The need for high-throughput pattern matching in bioinformatics has motivated myriad approaches including hashing, lookup tables, suffix arrays [9], and compressed suffix array data structures such as the FM-index [10]. Use of the FM-index across bioinformatic applications is due to its fast performance and low memory footprint. Unfortunately, its adoption is likely limited by the lack of an optimized and easy-to-use FM-index library; the only robust, currently maintained FM-index implementation we are aware of is found in the Seqan3 library [11]. Here, we present a lightweight, open-source library called AvxWindowedFMindex (hereafter shortened to AWFM-index), which enables highly optimized search string pattern matching over nucleotide or amino acid sequence datasets that demonstrates significantly faster performance than Seqan3's library.

AWFM-index achieves significant performance gains through multiple algorithmic and data structure changes over a traditional FM-index implementation. Rather than storing the database text T in ascii symbols or as a range of integral values representing the symbols in T, AWFM-index stores bit compressed symbols strided over 256-bit (AVX2) vectors that can be efficiently reduced with a low number of bitwise SIMD instructions. A table of k-mer seed ranges makes it possible to skip an early portion of the search computation for every query. Collections of multiple k-mers are queried in a thread-parallel manner, with good parallel scaling performance. AWFM-index is an open-source library written in C, with a simple API to facilitate easy integration into bioinformatics tools.

Data Structure Background:

Suffix Array

The suffix array [9] is a classic data structure that supports efficient determination of the count and locations of all occurrences of a query pattern, P, within a database sequence T. Given a text T that ends with a special sentinel symbol '$' (defined as a symbol in the text's alphabet Σ that otherwise does not occur in T, and is the smallest symbol in Σ), a suffix array SA is a permutation of integers [0..|T-1|], such that the suffix of T beginning at position SA[i] is lexicographically smaller than the suffix denoted by SA[j] if and only if $i < j$.

Because a suffix array lexicographically orders the suffixes of T, all indices of a given substring of T can be found in a contiguous range of elements in the suffix array. This fact is the key to the suffix array's fast search, as it enables counting in O(P log|T|) time through binary search across the suffix array and locating in O(P log|T| + k) time for k instances of the pattern. Without any data compression techniques, suffix arrays generally require 4 bytes of per symbol for sequences < 4GB long, or 8 bytes per symbol for sequences *geq* 4GB.

Numerous highly-efficient algorithms have been devised to quickly construct a suffix array from text T. The optimal asymptotic performance for suffix array construction is O(|T|) [12], but the O(|T| log |T|) complexity divsufsort [13] is commonly used because of its excellent speed as an in-memory suffix sorter for genome-scale inputs; AWFM-index utilizes libdivsufsort [14] for suffix array construction.

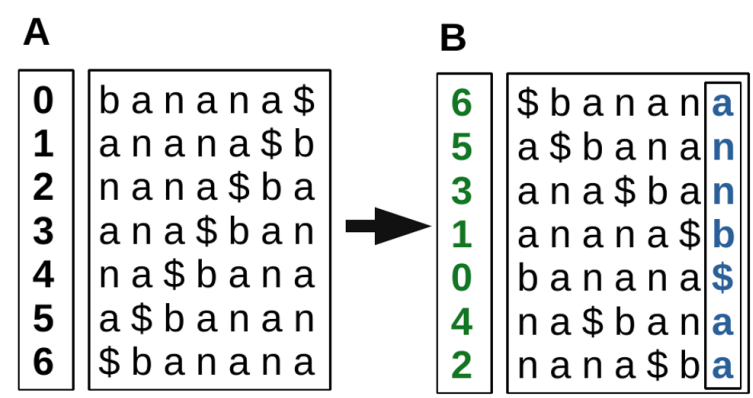
Burrows-Wheeler Transform (BWT)

The BWT is a reversible text transform that was originally proposed for lossless data compression [15]. Given a text T and a associated suffix array SA, a BWT is defined as the transformation:

$$\mathrm{BWT}(i) = \begin{cases} \mathrm{T}[\mathrm{SA}[i]-1] & \text{if } \ i \neq 0 \\ \$ & \text{otherwise} \end{cases} \tag{1}$$

In other words, each element in the BWT holds the symbol directly preceding the suffix denoted at that element's index in the suffix array. This is effectively the last column in a table of sorted rotations of T (see Fig 1), and is easily computed from a suffix array on T.

Fig 1. Example of generating a Burrows-Wheeler Transform for a given text. (A) All rotations of the input text 'banana', with concatenated sentinel '$' symbol. The position of each rotation is given in the left column. (B) After sorting the rotations, the left column retains the original position of each rotation, and is thus the suffix array of the text. The final column of this sorted rotation matrix is the BWT. Note: the actual rotation matrix need not be stored, or even computed; it is represented here as a visual aid.



FM-index

While the BWT can be viewed as a data-product of a suffix array, it can also be used as an alternative method for identifying pattern matches when used in conjunction with a suffix array. Using both data structures, Ferragina and Manzini introduced the FM-index [10]. A FM-index constructed from a given text T of alphabet Σ is comprised of the following: a suffix array SA, a Burrows-Wheeler Transform B, and auxiliary data that supports efficient counting of the number of occurrences of each symbol preceding each position in the BWT (see below for details). Using these data structures, an FM-index can perform two key query functions called Count() and Locate(). The Count() function returns the number of occurrences of the query pattern P in $O(|P|)$ time. The Locate() function returns the position in T of all k instances of P, in expected time $O(|P| + k)$.

Exact pattern matching with FM-index

Search for a pattern P in text T is performed one character at a time, beginning with the final character of the pattern and moving backwards. To begin, the search process establishes a start- and end-pointer [SP..EP] that correspond to the range in SA pointing to all occurrences of the final letter of P (Alg 1). In each successive step, the preceding character in P is prepended to the searched string P', and the range [SP..EP] is updated (via Alg 2) to correspond to all positions in the text T that match the growing suffix, P'. This continues until each symbol in P has been processed. SP and EP are updated each

time a new prefix symbol is added to the query using a function called occ() (short for occurrence). The occ(s, p) function takes as parameters a symbol $s \in \Sigma - \{\$\}$ and a position p where $0 \leq i < |T|$ and returns the number of occurrences of s in B before position p. To avoid unnecessary counting over large ranges of B, a milestone table is used to store the count S[s, p'] of symbol s preceding regularly sampled positions p'. When computing occ(s, p), the closest milestone position p' < p is identified, and the value S[s, p'] is added to the count of symbol s between p' and p. If the interval between milestones is r, the milestones table will require $(\Sigma \cdot \lfloor |B|/r \rfloor \cdot 8)$ bytes, assuming 64-bit integers are used to store the symbol counts. After the conclusion of Alg 3, the range [SP..EP] of the query represents the range of positions in the suffix array that represent suffixes that begin with pattern P (i.e., the locations of P). If at any point of the process SP > EP, P is not a substring of T, and the backward search is halted.

Algorithm 1: Constructing the initial [SP..EP] range for pattern P.

```
createInitialRange(P, C):
    last = P[ | P | -1]
    SP ← C[last]
    EP ← C[last+ 1] -1
    return [SP, EP]
```

Algorithm 2: FM-Index backward search extending a BWT range [SP, EP] with prefix symbol s.

```
backwardsExtendQuery(C, s, SP, EP):
    SP ← C[s] + occ(s, SP-1) + 1
    EP ← C[s] + occ(s, EP)
    return [SP, EP]
```

Algorithm 3: FM-index backwards search.

```
backwardSearch(C, P):
    [SP, EP] ← createInitialRange(P, C)
    k ← |P| − 2
    while k ≥ 0 and SP ≤ EP:
        s ← P[k]
        [SP, EP] ← BackwardsExtendQuery(C, s, SP, EP)
        k ← k - 1
    return [SP, EP]
```

Reducing space requirements by sampling the suffix array.

While locating query sequences using an FM-index has better complexity scaling than using only a suffix array for the same task (O(|P|) for FM-index as opposed to O(|P| log |T|) for suffix arrays), a naive FM-index requires more memory than suffix array alone, since it includes the BWT and milestone counts. By down-sampling the

suffix array [16], the memory footprint of the suffix array inside an FM-index can be dramatically reduced at the cost of a modest performance hit. A common SA sampling strategy is called subscript sampling [17], in which a sampling ratio r is chosen, and the sampled SA' is generated from all SA values at positions p where $p \equiv 0 \pmod{r}$.

At the conclusion of backwardSearch, each position in the [SP..EP] range corresponds to a position in the full SA, which itself indicates the location of an instance of P in T. Under SA down-sampling, only 1/r of the positions in [SP..EP] are expected to be present in SA' (i.e., only positions $p \equiv 0 \pmod{r}$ for p in [SP..EP]). For positions that are not sampled, the backtracePosition() function steps backwards through positions in the original text until a position sampled in SA' is reached, then returns the correct position by adding the number of steps that were taken to this SA' value.

A position p in B references some position B[p] in T. The backtrace step seeks to walk back in T until finding a position sampled by SA'; by construction, this is the character found at B[p]. Thus, to take one step back in T from a current position p, the symbol at p in B is found and the symbol is used in the occurrence function to find the BWT position of the previous symbol in the original text T (Alg 4).

Algorithm 4: Backtracing a position p to the nearest previous position sampled in SA'; this supports identification of the position in T corresponding to position p in SA.

```
backtracePosition(B, SA, p, s):
    offset ← 0
    while not p ≡ 0 (mod s):
        c ← B[p]
        p ← C[c] + occ(c, p)
        offset ← offset + 1
    return SA[p / s] + offset
```

Methods

The primary contribution represented by this work is an optimized FM-index library that is lightweight, easy-to-incorporate, and provides clients with FM-index functionality at both a high-level (count or locate all instances of a query string) and low-level (step-wise control of the reverse-search process). Here, we describe the various strategies that contribute to the library's fast text indexing performance. The key innovation is the development of a representation of BWT sequence data with a strided bit-compressed vector format interleaved with the milestone data. This format supports efficient computation of the most expensive aspect of FM-index calculations: the occurrence function.

We begin by describing a method for representing symbols in both nucleotide and amino acid alphabets with a carefully-selected combination of bits, and describe a method for efficiently testing for a match of a query symbol against such a bit representation, using a small number of boolean operations. We then demonstrate that this symbol representation can be used to compactly store an FM index in memory blocks representing 256 symbols at a time, and that these blocks can be efficiently processed using AVX2 vector instructions. This is followed by description of other aspects of the implementation, including a partial k-mer query lookup table that allows AWFM-index to skip the first few [SP..EP] update steps for each query.

Bitwise Symbol Matching

Consider an alphabet Σ, with each symbol in the alphabet encoded using n bits. In order to count the occurrences of a query symbol s in a range of symbols in the BWT, each symbol in the range must be checked for equality to s. While nearly all cpu architectures contain instructions to directly compare two numbers, we explore solutions that exploit bitwise operations for comparing symbols. Given two symbols s_1, $s_2 \in \Sigma$, it is possible to compare s_1 against s_2 using a straightforward combination of bitwise operations. For all set bits in s_1, the corresponding bits in s_2 are ANDed together. For all clear bits in s_1, the corresponding bits in s_2 are ORed together, and then bitwise NOTed. The boolean values that result from these actions are then ANDed together. The resulting value is true iff $s_1 = s_2$, and can be computed in n bitwise operations, or n-1 operations for the case where all bits in s2 are set. By performing bitwise operations in this way, we can check for symbol equality even in situations where a direct symbol comparison operation is not possible (as is the case when performing vectorized computations, as described shortly). For the purposes of this implementation we consider the following bitwise operations on a and b: AND(a,b) = a&b, OR(a,b) = a|b, and ANDNOT(a,b) = (!a)&b to each be single bitwise operations, as they are each a single CPU instruction within our instruction set.

In AWFM-index, two alphabets are supported, one for nucleotide data, one for amino acid data. Each alphabet contains symbols for each of the possible residues (4 for nucleotides, 20 for amino acids), a sentinel symbol, and an ambiguity symbol, denoted here as X, defined to be lexicographically greater than all other symbols in Σ. The resulting alphabets are length 6 and 22 respectively, and symbols in each alphabet are represented with $\lceil \log_2(6) \rceil$=3 and $\lceil \log_2(22) \rceil$=5 bits. Note that each of these alphabets have fewer symbols than the number of possible values for each of their corresponding bit lengths. A naive approach to assigning encodings to the $|\Sigma|$ symbols in each alphabet would be to assign them to the integers $[0 \,..\, |\Sigma| - 1]$. Instead, AWFM-index assigns alphabet symbol encodings to reduce the number of bitwise operations needed to compare symbols for equality. These encodings are presented in Table 1, and explained in the next two sections.

Nucleotide alphabet symbol encodings

Nucleotide symbols are represented by two groups of unique 3-bit encodings. Group-1 encodings have 2 of the 3 bits set, while group-2 encodings have only a single set bit. With a group-1 nucleotide query symbol, equality to another symbol is determined by ANDing the 2 bits corresponding to the set bits of the query. This produces a true boolean result if the symbol matches, and precludes a true result for any other symbol: (i) any other group 1 encoding would have a different pair of set bits, so that one of the compared bits would not be set, yielding a false result from the AND operation; (ii) any group-2 symbol contains only one set bit, so again the AND operation would return false, and (iii) since no encodings have more than 2 bits set, we can be sure that no other symbol could match to our query. Group-2 encodings can be checked for equality in 2 bitwise operations by taking the ANDNOT of the set bit and one of the clear bits, then ANDNOTing the result with the last clear bit. This strictly forces each of the 3 bits to match the query symbol.

Amino acid alphabet symbol encodings

Amino acid symbol encodings are split between 3 groups of unique 5-bit encodings. Group-1 encodings are represented by all 12 possible encodings in which exactly 2 bits in [0..3] differ from bit 4. Group-2 encodings are represented by all 8 possible encodings in which exactly 1 of the bits in [0..3] differs from bit 4. Group-3 encodings are represented

Table 1. Bit encodings for all nucleotide and amino acid symbols.

Nucleotide IUPAC Code	Bit Encoding	Symbol Group	# Bitwise Ops for Comparison
A	110		
G	101	1	1
C	011		
T	001		
X	010	2	2
$	100		
Amino Acid IUPAC Code	**Bit Encoding**	**Symbol Group**	**# Bitwise Ops for Comparison**
A	01100		
D	00011		
E	00110		
G	11010		
I	11001		
K	11001	1	2
L	11100		
P	01001		
R	10011		
S	01010		
T	00101		
V	10110		
C	10111		
F	11110		
H	11011		
M	11101	2	3
N	01000		
Q	00100		
W	00001		
Y	00010		
X	11111	3	3
$	00000		

by all 5 bits being either set or cleared, and denote the ambiguity symbol and the sentinel respectively. For a group-1 amino acid query symbol, equality can be tested in 2 bitwise operations, with the required operations depending on the state of bit 4 in the query symbol. If a query symbol is in group-1 and its bit 4 is set, one of the two bits corresponding to the query's clear bits is ANDNOTed with bit 4, then the other clear bit is ANDNOTed with the result. If bit 4 is clear for a group-1 symbol, the two bits corresponding to the query set bits are ANDed together, and the result is ANDNOTed with bit 4. Both of these options return true if and only if bit 4 matches the query, and the 2 bits that are supposed to differ from bit 4 in fact do so. Further, if the result is true, it shows that the symbol cannot be a group-2 encoding, since more than 1 bit differs from bit 4. In the same way, it shows that the symbol cannot encode for group-3. If the result is true, therefore, it cannot encode for any symbols other than our query.

For a group-2 amino acid query symbol, equality can be tested in 3 bitwise operations. If bit 4 is set, the bit corresponding to the query's singular clear bit is ANDNOTed with one of the 3 set bits in [0..3]. The other 2 set bits in [0..3] are ANDed together, and the result is ANDed with the result of the first operation. If bit 4 is clear, one of the 3 clear bits in [0..3] is ANDNOTed with the singular set bit. The remaining 2

clear bits are ORed together, and the result is ANDNOTed with the result of the first operation. Note that we did not check bit 4 in any way; if the result is true we can infer the state of bit 4 because no encodings exist with 3 bits that differ from bit 4. Therefore the state of bit 4 must be the opposite of the bit with the unique state. We also know that the symbol cannot encode for a group-1 or group-3 symbol, because we have shown that exactly 1 bit differs from bit 4. Therefore, the result is true if and only if the symbol matches the query.

Group-3 comparisons are straightforward. The ambiguity symbol is encoded with 5 set bits, and can be compared by ANDing together bits 0 and 1, ANDing bits 2 and 3, and ANDing the two results. Again, bit 4 does not need to be checked because no symbols are represented by all bits in [0..3] differing from bit 4. Comparison against the sentinel symbol is similar, however this is never necessary in practice since query strings cannot contain the sentinel.

As group-1 encodings require 1 less instruction to reduce, this group is used to encode for the 12 most frequent amino acids in the UniProtKB/Swiss-Prot database [18] and group-2 encodings represent the 8 least frequent amino acids. The reason is that the more-common amino acids will likely be queried more often, and therefore should be represented by encodings that require the fewest instructions.

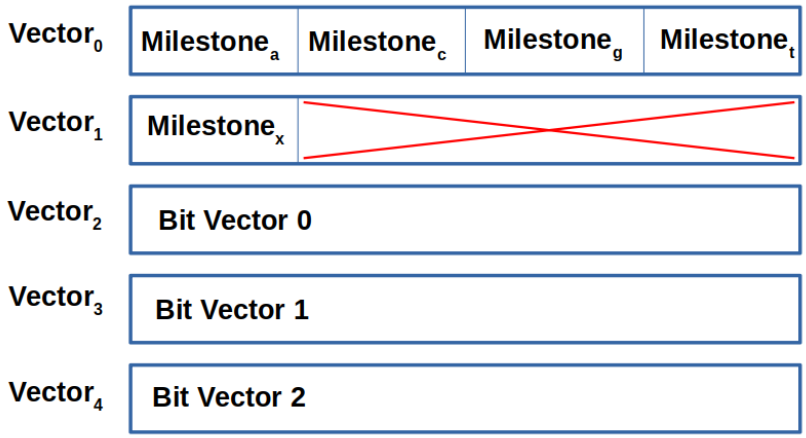
Strided Bit Vector Data Format

Counting the number of occurrences of a symbol s in the BWT prefix, B[0..p], can be achieved in two stages: (i) identify the largest milestone position p' $\leq$ p, and retrieve the corresponding count of symbol s in B[0..p'], then (ii) count the number of occurrences of s in B[p'+1..p] (assuming p'$\neq$p). Naively, the second stage can be performed by iterating over every symbol in B[p'+1..p], and comparing to the query symbol, incrementing a counter for each match. The previous section described a bitwise method for comparing a single symbol encoding against a query symbol. AWFM-index employs this bitwise comparison strategy in the context of Single Instruction, Multiple Data (SIMD) parallelization. Specifically, AWFM-index uses AVX2 instructions to perform the bitwise operations on vectors of 256 symbols in parallel, effectively comparing up to 256 symbols from the BWT to a single query symbol in the same number of bitwise instructions as comparing a single symbol. The AVX2 instruction set is an extension of the x86 instruction set that performs operations on vectors of 256 bits with a single instruction. AWFM-index uses 3 AVX2 intrinsic instructions (_mm256_and_si256, _mm256_or_si256, and _mm256_andnot_si256) to implement the bitwise operations for comparing symbol encodings, as described earlier. Through precise, interleaved layout of the BWT and milestone data, AWFM-index finds all matches to a query symbol in $\leq$4 instructions, and computes the occurrence function with a few extra steps.

In the AWFM-index, the BWT sequence is broken up into windows of 256 symbols; each window represents a range [i..j] (with j=i+255), and is comprised of 3 sections: (i) an array of 8-byte milestone occurrence counts containing the count of symbol s in B[0..i-1], for each symbol in the alphabet except the sentinel, (ii) a padding section to ensure that all other sections align to 32-byte boundaries necessary for AVX2 SIMD instructions, and (iii) multiple contiguous 256-bit AVX2 vectors that store the 256 symbols in a strided bit-vector format (see Fig 2, and text below). By including the milestone counts adjacent to the BWT windows, both the milestone count and the symbol bit vectors can be brought into cache in the same memory request. The milestone section of the window contains five (5) 64-bit values for nucleotide windows, or 21 64-bit values for amino acid windows. Since the milestone sections are aligned to a 32 byte (256 bit) memory boundary, a 24 byte padding (for both nucleotide and amino acid alphabets) ensures that the strided bit-vector section also aligns to a 32 byte boundary.

As a BWT window denotes a contiguous range of 256 symbols from the BWT, a

Fig 2. The 5 AVX2 vectors that comprise a nucleotide BWT window. The milestone counts for A, C, G, and T are stored in vector 1. Vector 2 contains the milestone for the ambiguity symbol 'X' and a 24 byte padding section to align the bit vectors to the 32 byte alignment necessary for AVX2 instructions



straightforward approach to storing symbols would be to represent each symbol with one byte as raw ASCII character values. For small alphabets, modern FM-index implementations prefer to use some form of bit-compressed representation, such as representing nucleotide symbols with 2 bits [1] (though this approach does not support ambiguity symbols, and special handling is required for the sentinel symbol). AWFM-index adjusts this bit-compression strategy to better leverage SIMD parallelization in computing the occurrence function. The symbols in the BWT are strided over the window's bit-vectors, with one vector for bit-0 of all 256 symbols, another vector for bit-1, and so on (generally: bit n of bit-vector m of a given window represents bit the m-th bit of the n-th symbol in the window). Including the milestone values and the padding, nucleotide data takes up 5 AVX2 vectors for each 256 symbol window, and thus requires 5 bits per symbol in the original text. Similarly, amino acid windows take up 11 AVX2 vectors, and so require 11 bits per symbol.

SIMD Occurrence Calculation

To compute the occurrence function occ(s, p) for symbol s and position p, the milestone occurrence count is taken from the appropriate section in the BWT window. Then, positions in the BWT window matching symbol s are identified and captured into a 256-bit vector such that bit n is set if and only if the n-th position in the window represents symbol s, here called an occurrence vector. This occurrence vector is generated by using AVX2 SIMD instructions to implement a bitwise comparison across all 256 symbols in the window, with bitwise instructions described in the previous section (see Fig 3). Once the occurrence vector has been generated, a bitmask is applied to clear all bits after position p; this ensures that no positions after p are counted in the final occurrence count. The set bits in each of the 4 quad-words in the occurrence vector are then counted with _popcnt64() intrinsic instructions, and the results are summed to compute the final occurrence count. Multiple strategies of generating a population count of the occurrence vector were tested, and summing the results of the 4 _popcnt64() instructions was found to outperform other SIMD vector popcounting techniques (e.g. [19]).

Fig 3. Examples of creating an occurrence bit vector from the strided BWT bit vectors. (a) An example of the bit vectors in a BWT window. Each bit vector is 256 bits wide representing 256 symbols, but only the first 8 positions are shown for brevity. By performing bitwise operations on these bit vectors an occurrence vector can be generated where a set bit indicates the presence of the queried symbol at the bit's position in the window. (b) Creating an occurrence bit vector for Group 1 amino acid "A" in 2 SIMD operations. (c) Creating an occurrence bit vector for Group 2 amino acid "H" in 3 SIMD operations.

(a).

	E	H	X	L	A	$	A	G	
Bit Vector$_4$	0	1	1	1	0	0	0	1	...
Bit Vector$_3$	0	1	1	1	1	0	1	1	...
Bit Vector$_2$	1	0	1	1	1	0	1	0	...
Bit Vector$_1$	1	1	1	0	0	0	0	1	...
Bit Vector$_0$	0	1	1	0	0	0	0	0	...

(b). Occ('A')

'A' = 0b01100

V_1 = Andnot (BV_4, BV_2)

= ~(0b0111 0001) & 0b1011 1010

= 0b**1000 1010**

Res = And (BV_3, V_1)

= 0b0111 1011 & 0b**1000 1010**

= 0b**0000 1010**

(c). Occ('H')

'H' = 0b11011

V_1 = And (BV_1, BV_0)

= 0b1110 0001 & 0b0110 0000

= 0b**0110 0000**

V_2 = AndNot (BV_2, BV_3)

= ~(0b1011 1010) & 0b0111 1011

= 0b**0100 0001**

Res = And (V_1, V_2)

= 0b**0110 0000** & 0b**0100 0001**

= 0b**0100 0000**

Manual Prefetching

While the computation to update the BWT range is minimal, the unpredictable nature of each subsequent position p given to the occurrence function creates a performance bottleneck in reading data from memory. Given a sufficiently large BWT, every occurrence call will result in a memory read request for cache lines that will almost certainly not be in cache, except for pathological case queries like 'AAAAAAA', or if SP and EP land in the same BWT window. To ease the performance hit caused by this random access, AWFM-index employs manual data prefetching using the

_mm_prefetch() SSE instruction as soon as the location of the memory address for the next occurrence function is known. The updates to SP and EP are also staggered such that after SP is updated, a prefetch request is generated and the update to EP begins while the SP memory prefetch request is being serviced.

Accelerated Search with a K-mer Lookup Table

Traditionally, searching an FM-index involves updating the [SP,EP] pair for every symbol in the given query. AWFM-index uses a pre-computed lookup table with a modest memory footprint to skip a sizable portion these [SP,EP] updates. When the AWFM-index is built, a parameter k is selected, and a table is allocated to store the [SP,EP] pair for every length-k string over alphabet Σ (except the ambiguity and sentinel symbols, which are excluded because neither are found in query strings). This table of k-mer ranges consumes $16 \cdot (|\Sigma| - 2)^k$ bytes in memory, and memoizing prefix ranges enables $O((|\Sigma| - 2)^k)$ construction time. If a query string q is at least length k, the [SP,EP] range for the length k suffix of q is found in the k-mer lookup table, effectively skipping the first k updates to the [SP,EP] range. Then, the query proceeds as normal, querying for symbols until the entire query has been completed, or the range is invalid. If the query string length is less than k, the range is resolved without using the k-mer table. The recommended values of k are 12 for nucleotide indices (268MB table size) and 5 for amino indices (51MB table size) as they strike a balance between memory footprint and performance benefit. Other values may be selected by the library client depending on expected factors such as query lengths and available system memory.

API and Thread-Parallel Search

The core API for AWFM-index includes the locate() and count() functions, which each accept as arguments (i) the AWFM-index data structure, (ii) a collection of query sequences, and (iii) a number of threads used to parallelize search. Parallelization is achieved using simple OpenMP pragmas, as each query in the collection is data-independent with respect to the other queries in the collection. Given a collection Q of query sequences, the collection is implicitly divided into bins of 4 queries, resulting in a total of $\lceil |Q| \ / \ 4 \rceil$ bins. The user-specified number of threads are then used to parallelize the search across the collection of bins. When a thread begins to compute the results for a bin of queries, it begins by finding [SP,EP] range in the k-mer seed table that represents the final k symbol suffix for each of the 4 queries. Then, each query in the bin is extended until either the SA range of the query has been fully resolved, or has failed due to SP > EP. If the parallel locate() function is used, the location of each instance of each query string is found via the SA backtracking method described earlier.

The AWFM-index API also includes non-parallelized functions for initializing a SA range, extending queries with additional prefix symbols, backtracing to the most recently sampled SA position, and looking up the original position using the suffix array. These functions allow a user to implement custom FM-index applications such as inexact pattern matching.

Suffix Array Sampling, on-memory or on-disk

The suffix array component of the FM-index is often down-sampled to reduce memory requirements. Suffix arrays that are sparsely sampled have a smaller memory footprint, but require more backtrace steps to deduce the actual sequence position during the locate() function. The AWFM-index library currently supports suffix array sampling ratios r that are $1 \leq \ r < 256$ and utilizes a subscript sampling strategy such that every

r^{th} entry is sampled. Since each suffix array value is stored as an 8 byte integer, a suffix array from text T under sampling ratio r will be $8 \cdot \lfloor (|T|+1)/r \rfloor$ bytes in size. AWFM-index provides the option to either load the suffix array into memory with the rest of the index (default) or leave the suffix array on disk and read directly from disk the values necessary to resolve the final sequence positions after all SA range elements have been backtraced to a sampled SA position. While disk access is significantly slower than memory access, on-disk suffix array storage allows for denser sampling, even on systems with limited memory. For example, in the context of the locate() function, using an AWFM-index with a suffix array sampling ratio of 1 where the suffix array is left on disk results in a single random disk read for each found substring, which may be preferable to the large number of sequential cache misses necessary to backtrace to the nearest sampled suffix array position for each found substring. This functionality aims to make high performance indexing accessible to a wider range of users on personal computers or laptops with limited memory.

Results

We performed numerous tests to assess the performance of AWFM-index relative to SeqAn's FM-index implementation, and to demonstrate the impact of AWFM-index parameterization. All tests were run on a dedicated system with two Intel Xeon Gold 6240 processors (18-core Cascade Lake, 2.60 GHz).

Speed Comparison Between AWFM-index and Seqan3

We created a simple benchmark dataset to evaluate the performance of AWFM-index relative to the FM-index implementation in the Seqan3 library [11]. A 1 billion base pair nucleotide sequence and a 200 million amino acid sequence were generated with the easel sequence analysis library [20]. FM-index files were generated from these sequences for both Seqan3 and AWFM-index using the native construction method for each tool. The index build times were similar for the two libraries. A partial lookup table was pre-computed for all length-12 (nucleotide) and length-5 (amino acid) k-mers. The suffix array was compressed by a ratio of 8, and kept in memory. A collection of 10 million queries of varying lengths were sampled from the original text, and run times for locate() and count() functions were captured in Tables 2 and 3.

Table 2. Run time for nucleotide locate() and count() functions. Target consists of 1 billion nucleotide-long simulated sequence. Query consists of 10 million nucleotide query sequences sampled from the target, at each of several query lengths.

Nucleotide search		Count() Time (seconds)			Locate() Time (seconds)		
Query Length	**Hits/ Query**	**Seqan3**	**AWFM**	**Speed-up**	**Seqan3**	**AWFM**	**Speed-up**
20	1.3	36.6	8.2	4.5	80.7	19.1	4.2
18	1.5	29.0	6.7	4.3	82.6	21.9	3.8
16	2.0	24.2	5.1	4.8	89.5	20.0	4.5
14	5.9	19.9	3.5	5.7	215.0	59.2	3.6
12	62.0	13.8	1.5	9.0	2040.4	472.8	4.3
11	240.9	11.7	6.2	1.9	7509.1	1774.7	4.2

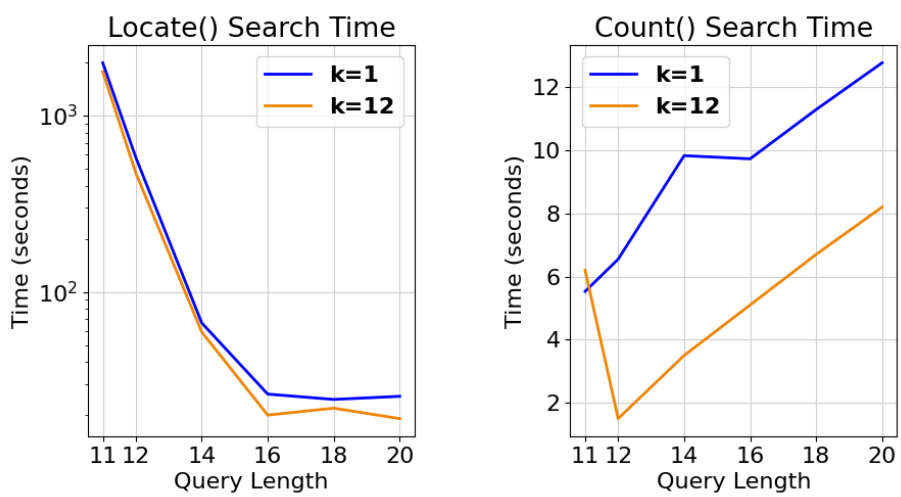
Table 3. Run time for amino acid locate() and count() functions. Target consists of 200 million character-long simulated amino acid sequence. Query consists of 10 million amino acid query sequences sampled from the target at each of several query lengths.

Amino acid search		Count() Time (seconds)			Locate() Time (seconds)		
Query Length	**Hits/ Query**	**Seqan3**	**AWFM**	**Speed-up**	**Seqan3**	**AWFM**	**Speed-up**
10	6.8	33.8	5.6	6.1	346.1	38.4	9.0
9	7.3	32.8	3.9	8.4	349.5	38.6	9.0
8	7.9	24.3	3.3	7.5	363.1	39.9	9.1
7	8.8	19.7	2.6	7.6	419.7	45.1	9.3
6	16.9	14.7	1.9	7.6	964.1	132.8	7.3
5	145.9	11.7	0.5	23.3	9078.5	1453.2	6.2

Effect of K-mer Lookup Table on Speed

To gauge the performance gains due to the partial k-mer lookup table, the previous nucleotide benchmark was used to compare AWFM-index locate() performance with a minimum size lookup table (k=1) with the default recommended size partial k-mer look table (k=12). Figure 4 shows that count() performance is significantly boosted by avoiding the first 12 steps of Alg 3. Meanwhile the impact on locate() is modest, as large numbers of backtrace operations cause runtime to be dominated by occurrence calculations.

Fig 4. Timings of 10 million nucleotide queries using a partial k-mer table of length 1 (blue), and of length 12 (orange). The performance benefits of the partial k-mer table are most obvious in the count() function, whereas the performance benefit in the locate() function is most notable for longer queries that generate fewer hits.



Suffix Array Sampling

We find that densely sampled in-memory suffix arrays are faster than densely sampled on-disk suffix arrays by up to a factor of 10 (Table 4). On-disk suffix arrays only outperform in-memory suffix arrays when the on-disk SA is uncompressed or densely sampled and the in-memory SA is very sparsely sampled. On-disk suffix arrays should only be considered in extremely memory-constrained systems, as even a SA with sampling ratio of 64 (125MB/billion positions) outperform all on-disk suffix arrays.

Table 4. Impact of suffix array compression. Suffix array memory requirements for various suffix array compression ratios, and the time taken to locate() 10 million length 14 nucleotide queries for in-memory and on-disk suffix arrays. While unsampled on-disk suffix arrays can outperform sparsely sampled in-memory suffix arrays, moderately sampled in-memory suffix arrays consume relatively little memory and outperform unsampled suffix arrays left on-disk. On-disk suffix arrays should be considered only for extremely memory-constrained systems with ample disk storage.

Suffix Array		**Locate() time (seconds)**	
Compression ratio	**in-memory size**	**On-disk SA**	**In-memory SA**
1	8 GB	20.7	2.4
4	2 GB	22.1	3.3
8	1 GB	18.5	4.7
16	500 MB	21.1	6.5
32	250 MB	23.1	9.5
64	125 MB	20.9	15.3
128	75.5 MB	29.7	26.1
200	40 MB	42.4	40.7

Thread-Parallel Performance

We evaluated the speed gains achieved with multi-threading using the nucleotide benchmark described above (1 billion simulated nucleotides), with 10 million length-14 query strings. As seen in Fig 5, performance scales effectively up to 20 threads.

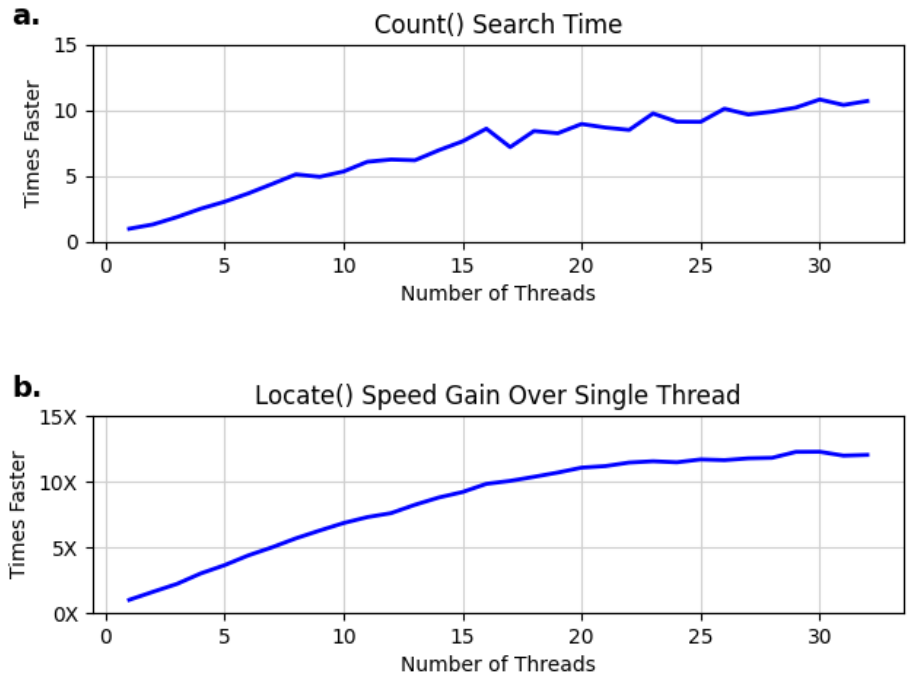
Prefetch Directives

The efficacy of data prefetch directives was analyzed by timing nucleotide locate() functions with each prefetch hint, and with prefetching directives disabled. Prefetch hint directives are used to tell the CPU which levels of cache to store the data in. All prefetching hints were shown to improve overall performance by a small amount, but non-temporal prefetching (_MM_HINT_NTA) was shown to be fastest over multiple trials at a performance gain of 1.4%. Since the performance difference is minimal, prefetching is not a major contributor to AWFM-index's excellent performance.

Discussion

We developed AWFM-index to be a lightweight, easy-to-incorporate library that simplifies the inclusion of fast pattern matching into bioinformatics software. Our implementation leverages a custom data layout and SIMD vectorized character comparison instructions to produce highly efficient symbol counting for nucleotide and amino acid alphabets. Combined with a pre-computed k-mer lookup table and out-of-the-box parallelism, the result is a library that provides very fast locate() and count() queries with very little development effort in the client. In addition to

Fig 5. AWFM-index search times parallelized with varying numbers of threads, from single-threaded search up to search using 32 threads. (a) Increase in speed for the count() command, relative to single-threaded search. (b) Increase in speed for the locate() command, relative to single-threaded search.



single-command search for full query sequences, the AWFM-index API also exposes stepwise iterative search functionality, so that clients can exert fine-grained control over FM index search steps, for example in support of back-tracking for inexact search as used in [1, 2].

While we expect AWFM-index to be immediately applicable in its current form, we note two potential improvements that will improve the future value of the library. The first of these is support for bi-directional FM-index search [21]. The bi-directional FM-index supports updates to the range of matching substrings by extending an existing substring [SP..EP] range with either a suffix or prefix symbol, and achieves this by supplementing the data structure with a single additional BWT over the reversed sequence T. Adding bi-directional search functionality to the library will improve it's applicability to some special-case pattern matching applications such as [22].

The second improvement will extend the performance benefit of the k-mer lookup table. As described above, the BWT range of a query with the same length as the k-mer seed table (e.g. nucleotide search for a length-12 query) is identified with a single memory access. Conversely, search for a slightly shorter k-mer (e.g. length 11 nucleotide query) does not use the k-mer seed table, and thus receives no search shortcut (see Table 2, Fig 4). Future work on AWFM-index will aim to leverage the k-mer seed table for queries shorter than k. For instance, consider the length-5 nucleotide query "CGTAG", and a lookup table storing all length-7 nucleotide suffixes. Since all suffixes

in the BWT are sorted, suffixes that begin with "CGTAG" will be found between (i) the start of the range for the k-mer extended with lowest rank non-sentinel symbol (here, "CGTAGAA"), inclusively, and (ii) the start of the range for the k-mer lexicographically one higher, extended with the lowest rank non-sentinel symbol (here, "CGTATAA"), exclusively. However, use of these longer strings as proxies during the identification of the BWT range fails to account for the possibility of a sentinel symbol, which may introduce a non-matching string into the proxy range. Since a BWT is guaranteed to only contain a single sentinel symbol at the end of the sequence, the last few symbols of the original text T can be kept along with the k-mer seed table, and used to remove this matches from a range list. A more thorny problem arises when the short query k-mer ends with a symbol of the highest (non-ambiguity) rank (nucleotide T or amino acid Y), as the lookup table does not have a higher-rank symbol to use in selecting the top end of the range. One way to resolve this is to store all ranges in the k-mer table, including those that contain ambiguity symbols; however, including the ambiguity symbol X increases the table size appreciably, e.g. a table of all length-12 nucleotide k-mers takes $16 \cdot 4^{12} = 268$MB, while the same table that also stores ambiguity characters takes $16 \cdot 5^{12} = 3.9$GB. Workarounds include creation of partial k-mer seed range tables, and conditional fallback to the traditional backwards search method. We aim to address both of these issues in a future library release.

Availability of Data and Materials

The scripts used to generate tables and figures in this study are available at `http://wheelerlab.org/publications/2021-AWFM-Anderson/Anderson_suppl.tar.gz`.

Acknowledgments

We thank Robert Hubley for beta testing AWFM-index and suggesting improvements to the library's build process, as well as George Lesica for his help in improving the library API. We also gratefully acknowledge the computational resources provided by the University of Montana's Griz Shared Computing Cluster (GSCC).

Funding

This work was supported by NIH grant P20 GM103546 (NIGMS) and DOE grant DE-SC0021216.

Competing interests

The authors declare that they have no competing interests.

References

1. Li H, Durbin R. Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics. 2009;25(14):1754–1760.

2. Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology. 2009;10(3):R25.

3. Kim D, Song L, Breitwieser FP, Salzberg SL. Centrifuge: rapid and sensitive classification of metagenomic sequences. Genome Research. 2016;26(12):1721–1729.

4. Menzel P, Ng KL, Krogh A. Fast and sensitive taxonomic classification for metagenomics with Kaiju. Nature Communications. 2016;7(1):1–9.

5. Huang YT, Huang YW. An efficient error correction algorithm using FM-index. BMC Bioinformatics. 2017;18(1):524.

6. Buchfink B, Xie C, Huson DH. Fast and sensitive protein alignment using DIAMOND. Nature Methods. 2015;12(1):59–60.

7. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. Journal of Molecular Biology. 1990;215(3):403–410.

8. Steinegger M, Söding J. MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. Nature Biotechnology. 2017;35(11):1026–1028.

9. Manber U, Myers G. Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing. 1993;22(5):935–948.

10. Ferragina P, Manzini G. Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science. IEEE; 2000. p. 390–398.

11. Reinert K, Dadi TH, Ehrhardt M, Hauswedell H, Mehringer S, Rahn R, et al. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. Journal of Biotechnology. 2017;261:157–168.

12. Ko P, Aluru S. Space efficient linear time construction of suffix arrays. In: Annual Symposium on Combinatorial Pattern Matching. Springer; 2003. p. 200–210.

13. Fischer J, Kurpicz F. Dismantling divsufsort. arXiv preprint arXiv:171001896. 2017;.

14. Mori Y. libdivsufsort;. Available from: `https://github.com/y-256/libdivsufsort`.

15. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation;.

16. Ferragina P, González R, Navarro G, Venturini R. Compressed text indexes: From theory to practice. Journal of Experimental Algorithmics. 2009;13:1–12.

17. Cheng H, Wu M, Xu Y. FMtree: A fast locating algorithm of FM-indexes for genomic data. Bioinformatics. 2018;34(3):416–424.

18. Consortium U, et al. UniProt: the universal protein knowledgebase in 2021. Nucleic Acids Research; p. gkaa1100.

19. Muła W, Kurz N, Lemire D. Faster Population Counts Using AVX2 Instructions. The Computer Journal. 2017;61(1):111–120.

20. Eddy S. Easel - a C library for biological sequence analysis;. Available from: `http://bioeasel.org`.

21. Lam TW, Li R, Tam A, Wong S, Wu E, Yiu SM. High throughput short read alignment via bi-directional BWT. In: IEEE International Conference on Bioinformatics and Biomedicine. IEEE; 2009. p. 31–36.

22. Simpson JT, Durbin R. Efficient construction of an assembly string graph using the FM-index. Bioinformatics. 2010;26(12):i367–i373.