BINSEQ: A Family of High-Performance Binary Formats for Nucleotide Sequences

Noam Teyssier
noam.teyssier@arcinstitute.org
Arc Institute

Alexander Dobin
alexander.dobin@arcinstitute.org
Arc Institute

Abstract

Modern genomics produces billions of sequencing records per run, which are typically stored as gzip-compressed FASTQ files. While this format is widely used, it is not optimal for high-throughput processing due to its reliance on single-threaded decompression and sequential parsing of irregularly sized records. This limitation is particularly problematic for applications that would benefit from parallel processing, such as read mapping, variant calling, and de novo assembly. Here, we present BINSEQ, a family of simple binary formats that enable high-throughput parallel processing of sequencing data. The BINSEQ family consists of two complementary implementations: BQ, optimized for fixed-length reads using a two-bit or four-bit encoding scheme with true random record access capability, and VBQ, designed for variable-length sequences with optional quality scores and block-based compression. We demonstrate that BINSEQ files are up to 90x faster than compressed FASTQ for parallel processing and can reduce analysis time from hours to minutes for large-scale genome and transcriptome analyses, particularly for resource-intensive applications like alignment, mapping, and de novo assembly. To facilitate adoption we provide high-performance libraries for reading and writing BINSEQ formats, native parallelization strategies with convenient APIs, and a command-line tool for conversion to and from traditional formats.

Keywords Bioinformatics · Genomics · NGS · Compression

Author Summary

Modern sequencing technologies routinely generate billions of reads per experiment, yet the methods for storing and accessing this data have not kept pace. Sequencing reads remain predominantly stored in FASTQ, a text-based format designed for far smaller datasets. FASTQ's sequential parsing requirements and practical need for compression create a fundamental mismatch with modern multi-core architectures, where data access rather than computation has become the primary bottleneck. We address this problem with BINSEQ, a family of binary formats engineered for random access and native parallelization. Systematic benchmarking across applications of varying computational complexity demonstrates that BINSEQ achieves 90-fold improvements in data access and maintains substantial advantages in compute-intensive tasks such as genome alignment, reducing runtimes from hours to minutes. We present two complementary implementations: BQ, optimized for simplicity and maximal throughput, and VBQ, designed for flexibility while maintaining high performance. By reconsidering the relationship between storage architecture and parallel processing capabilities, BINSEQ provides a practical solution to a critical infrastructure challenge in high-throughput genomics.

1 Introduction

Modern genomics routinely generates billions of sequencing records per experimental run, with data predominantly stored in gzip-compressed FASTQ[1]. While this format has remained the *de facto* standard for sequencing data storage due to its simplicity and widespread tool support, its fundamental design choices present significant operational constraints. The format's minimal specification, while facilitating broad adoption, introduces implementation variability and edge cases that compromise both compatibility and performance. A central limitation stems from its variable-length record structure, which necessitates sequential parsing even when the underlying nucleotide sequences are of fixed length. This architectural constraint, coupled with the practical requirement for data compression, creates an inherent tension between storage efficiency and processing speed.

These limitations become particularly acute in computational workflows that would naturally benefit from parallel processing, such as alignment[2,3], pseudoalignment[4,5], and de novo assembly[6,7]. These are embarassingly parallel operations and the sequential nature of FASTQ parsing and decompression creates an unnecessary processing bottleneck that prevents effective utilization of modern multi-core architectures[8]. This inefficiency is most pronounced in, but not limited to, applications with low per-record computational complexity where I/O operations within decompression and parsing dominate the total execution time. The fundamental limitation of FASTQ, and other sequentially parsed formats, is that they are I/O-bound rather than CPU-bound. Consequently, application performance becomes bounded by storage medium throughput rather than available computational resources.

This misalignment between modern sequencing characteristics and legacy storage formats presents a clear opportunity for optimization. There is no standard for unaligned sequencing data and researchers have used many formats such as unaligned-SAM[9], BAM[9], and CRAM[10] - though the *de facto* standard is GZIP-compressed FASTQ[1]. There have also been many novel compression algorithms[11–13] and storage formats such as the Nucleotide Archive Format (NAF)[14] proposed as alternatives to general purpose compression algorithms. However, many of these approaches focus on efficient compression and decompression without specifically considering accession properties of the underlying data which remains the limiting factor to computational efficiency.

Here, we present the BINSEQ family of formats, specifically engineered to exploit the properties of contemporary sequencing data and built for high-throughput parallel access. We introduce two complementary implementations: BQ, optimized for fixed-length reads, and VBQ, designed for variable-length sequences while maintaining high-performance characteristics. Unlike FASTQ, both formats natively support single- and paired-end reads unambiguously, eliminating the need to keep multiple synchronized files.

BQ introduces two key innovations in sequence data storage. First, it enforces fixed-size records for all sequences, enabling deterministic random access to any record without sequential parsing. Second, it employs a two-bit or four-bit encoding scheme for nucleotide representation, achieving a four-fold or two-fold improvement in space efficiency compared to the ASCII-based representation used in plain-text formats. This combination provides inherent compression without requiring additional compression algorithms, while the fixed-size record structure enables true parallel processing. Furthermore, by eliminating the storage of quality scores and sequence identifiers — elements frequently ignored by modern bioinformatics tools — BQ achieves additional space and processing efficiencies.

VBQ extends these core principles to accommodate variable-length sequences and quality score information when needed. Rather than enforcing fixed-size records for the entire dataset, VBQ organizes data into fixed-length and independent record blocks. Within these record blocks, sequences can be variable length and optionally include quality scores. Each block is optionally ZSTD compressed, and the format supports indexing for parallel access to record blocks. While VBQ sacrifices direct record random access, it maintains exceptionally high performance for parallel processing and offers greater flexibility at the record level than BQ.

In this paper, we present a comprehensive description of the BINSEQ family designs and implementations, accompanied by extensive performance evaluations across bioinformatics applications of varying complexities. Our results demonstrate significant improvements in both processing speed and storage efficiency, establishing BQ and VBQ as compelling alternatives to traditional sequence storage formats for modern genomics workflows.

2 Methods

2.1 File Format Specification

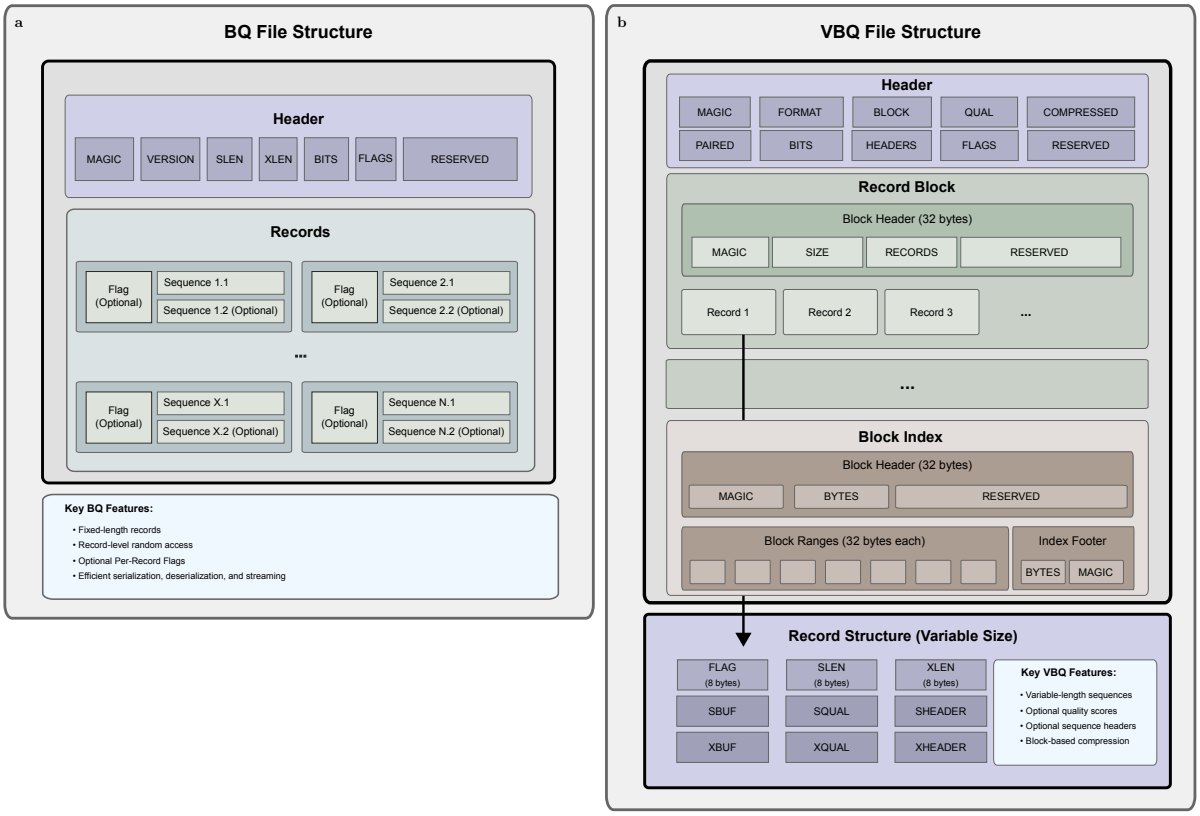

Figure 1: BINSEQ file format family. (a) The BQ file format. The header provides sequence-level descriptors (Table 5). Each record is fixed length and is composed of a flag section and a sequence section. Records are optionally paired sequences and sequence lengths are described in the header. (b) The VBQ file format. The header provides which optional features are present in the file (Table 6). Each record can be variably sized, include quality scores, include sequence headers, and be paired.

2.1.1 BQ

BQ is implemented as a binary file format (.bq) specifically engineered for the efficient storage and retrieval of fixed-length DNA sequences (Figure 1a).

The format employs a two-bit or four-bit nucleotide encoding scheme (Table 4) and maintains a consistent record structure to enable direct random access. Each BQ file consists of two primary components: a fixed-size header (Table 5) and a data section containing sequence records.

The data section follows immediately after the header and consists of fixed-size records arranged sequentially. Each record comprises a flag field and sequence data, with the record size determined by the sequence length. The flag field, implemented as a 64-bit unsigned integer, precedes the sequence data and can accommodate implementation-specific metadata, filtering criteria, or quality metrics.

Sequence data storage employs a dense two-bit or four-bit encoding scheme defined by a simple lookup table (Table 4). Invalid nucleotides are controlled via a configurable policy, allowing users to skip sequences with invalid nucleotides, replace invalids with random nucleotides, or replace them with a predetermined nucleotide. However, `N` nucleotides can be encoded using the four-bit encoding scheme. All nucleotide encodings are represented using little-endian byte order, with subsequences packed into 64-bit integers. If a sequence length is not a multiple of 32 or 16, the final integer is padded with zeros to maintain alignment.

There are two variants of the sequence data storage: primary and extended. The primary sequence data is meant to represent the first read pair or single-end reads, while the extended sequence data is used for the second read pair in paired-end sequencing. The extended sequence is optional and can be omitted if not required (and is determined by the *xlen* field in the header).

For a sequence of length N, the total record size S_r is given by:

$$S_r = F + W\left(\lceil \frac{N}{B} \rceil\right) \tag{1}$$

For an extended sequence pair with lengths N and M, the total record size S_p is given by:

$$S_p = F + W\left(\lceil \frac{N}{B} \rceil + \lceil \frac{M}{B} \rceil\right) \tag{2}$$

where:

- F is the flag field size (0 or 8 bytes depending on whether the FLAG is set to true in the header)
- N is the sequence length in nucleotides
- W is the word size (8 bytes)
- B is the bits per word (e.g. 16 for four-bit and 32 for two-bit encodings)
- $\lceil x \rceil$ denotes the ceiling function

The position P of any record i in the file can be calculated as:

$$P_i = H + S(i) \tag{3}$$

where:

- H is the header size (32 bytes)
- i is the zero-based record index
- S is a record size $\{S_r, S_p\}$ calculated in Equation 1 or Equation 2

2.1.2 VBQ

VBQ is implemented as a binary file format (.vbq) and is designed to increase flexiblity in record storage but still allow for high parallel processing capabilities (Figure 1b).

At its core, VBQ is an extension of BQ with two notable differences: variable length records and fixed-size record blocks. Each VBQ file consistest of four primary components: a fixed-size header (Table 6), block headers(Table 7), records (Table 8), and a block index.

Each block has the same virtual memory size, which is specified by the file header, and packs as many complete records as possible in the block. These blocks are then optionally ZSTD compressed, and the compressed size with the number of records in the block are stored in a preceding block header. This effectively keeps each block independent and able to be processed in parallel.

Because VBQ is optionally compressed, the block header locations cannot be known absolutely, but can be easily indexed after creation to allow for parallel decompression and access. The index is a simple ZSTD-compressed binary section that annotates the start and end locations of compressed blocks in the file as well as the the cumulative number of records at that block start. It begins with an uncompressed header that designates the file format as well as a quick-check on the number of bytes in the associated VBQ file to quickly check if the files are accurately paired (Table 9). The remaining bytes are compressed and are composed of repeating ranges (Table 10) which describe each block in the file. It finishes with the total number of bytes in the index and with a magic number to verify the integrity of the index. This

index is kept at the end of the file as it is defined after write, but is loaded first when reading to allow for parallel decompression and access.

2.2 Implementation

2.2.1 Nucleotide Encoding

BINSEQ employs a highly-efficient bit encoding library for nucleotide sequences. The library, *bitnuc*, provides a simple API for encoding and decoding nucleotide sequences represented as a vector of bytes, into a vector of 64-bit integers. The encoding scheme is based on a lookup table that efficiently maps nucleotide characters into two-bit or four-bit representations and packs them into 64-bit integers with efficient bitwise operations.

The implementation heavily makes use of SIMD instructions to accelerate encoding and decoding operations and is optimized for modern architectures. These instructions are available on x86 and ARM architectures and provide significant performance improvements over scalar implementations, which are provided as fallbacks for compatibility. The library is designed to be thread-safe and can be used in parallel processing environments without additional synchronization overhead.

2.2.2 File I/O

BINSEQ provides a high-performance library for reading and writing BINSEQ files. This library is implemented in the Rust programming language and leverages the language's safety guarantees and performance characteristics to provide a robust and efficient implementation. The library is designed to be thread-safe and can be used in parallel processing environments without additional synchronization overhead. We have also developed both C and C++ bindings to the Rust library for multi-language support. Notably, C/C++ bindings are available for memory-mapped file access and do not currently support streaming at the time of writing though they are planned for future releases.

The compact binary format and fixed-size record structure allows for either sequential or random access to records, providing multiple access patterns for different applications. Both formats allow for direct zero-copy memory mapping, enabling very efficient access patterns.

2.2.3 Parallel Processing

BINSEQ provides a simple hook-based interface for parallel processing (Table 11). This interface follows a classic map-reduce pattern, allowing users to define custom map and reduce functions that operate on individual records and aggregate results on a regular batch size. This allows for efficient asynchronous processing of records in parallel during mapping but allows users to decide when best to synchronize threads for Reduce operations.

2.2.4 Command-line Tool

Alongside the core library, we have developed a command-line tool for converting between BINSEQ and FASTQ formats, bqtools. bqtools provides a simple interface for converting between formats, allowing users to integrate BINSEQ into existing workflows.

2.3 Performance Evaluation

To evaluate the performance of BINSEQ, we developed 3 different benchmarking scenarios to assess the format's efficiency across a range of bioinformatics applications. These scenarios are not meant to be exhaustive but rather to provide a representative sample of the performance improvements that BINSEQ can offer. The scenarios are as follows:

Sequence Access: This is the simplest possible scenario, where we measure the time taken to read all nucleotide sequences from a file. This is meant to replicate the raw parsing performance of the format and provide a baseline for comparison.

k-mer Counting: This is a common low-complexity bioinformatics task where we count the number of unique k-mers of incoming records. We select a low k to measure the performance of a low-complexity application.

Alignment: This is a more complex scenario where we align incoming records to a reference genome. This is meant to simulate a more computationally intensive operation on each record and provide a measure of the format's performance in a high-complexity application. This scenario is more indicative of the performance improvements that BINSEQ can offer in applications like read mapping, variant calling, and de novo assembly.

Sequence access benchmarks were performed locally on a Macbook Pro M3. k-mer counting and alignment benchmarks were performed on a high-performance computing cluster using a Dual Intel(R) Xeon(R) Platinum 8468 CPU with NVMe solid-state drives for IO.

2.3.1 Sequence Access

To evaluate sequence access performance, we measured the time taken to read and potentially decode all nucleotide sequences from a file. This scenario is meant to replicate the raw parsing performance of the format and provide a baseline for comparison. To encapsulate a fundamental trade-off of modern genomics we sought not only to evaluate the speed of the format but also its storage requirements.

To represent this trade-off quantitatively we developed a normalized composite metric, m_i (Equation 4), that combines the normalized time taken to read the sequences, $\bar{t}_t$ (Equation 5), and the normalized storage requirements, $\bar{s}_i$ (Equation 6), for each format i.

$$m_i = \frac{\bar{t}_i + \bar{s}_i}{2} \tag{4}$$

$$\bar{t}_i = \frac{t_i - \min(T)}{\max(T) - \min(T)} \tag{5}$$

$$\bar{s}_i = \frac{s_i - \min(S)}{\max(S) - \min(S)} \tag{6}$$

Each normalized metric $\{\bar{t}, \bar{s}\}$ is a value bounded by $[0, 1]$ where 0 represents the best performance and 1 represents the worst performance of that metric across all formats tested. The composite metric m is the arithmetic mean of the normalized time and storage requirements, providing a single value that represents the trade-off between the two metrics for each format. This treats both time and storage requirements as equally important factors in evaluating the performance of a format, but this metric could also be adjusted to reflect different priorities.

To attempt a fair comparison between formats we evaluated multiple formats and compression strategies (Table 1). For each format we evaluated the performance of both single-threaded and parallel parsing strategies where available.

Table 1: Sequence Access Formats, Compressions, and Processing Strategies

File Format	Additional Compression	Parallelism	Base Encoding
FASTA[15]	{None, Gzip, Zstd, Lz4}	No	None
FASTQ[1]	{None, Gzip, Zstd, Lz4}	No	None
SAM[9]	None	{No, Yes}	None
BAM[9]	None	{No, Yes}	4-bit
CRAM[10]	None	{No, Yes}	4-bit
NAF[14]	None	No	4-bit
BQ	None	{No, Yes}	2-bit, 4-bit
VBQ	None	{No, Yes}	2-bit, 4-bit

The FASTA and FASTQ parsers were implemented using the the Rust *seq_io* library[16], which offers zero-copy implementations. The SAM, BAM, and CRAM parsers were implemented using the Rust *rust-htslib* library[17], which offers FFI bindings to the C *htslib* library[18], with parallel parsing strategies. The BQ and VBQ parsers were implemented using the Rust *binseq* library, which offer zero-copy implementations with parallel parsing strategies. The Nucleotide Archive Format (NAF) parsers were implemented by wrapping the *unnaf* binary[14] and parsing sequences from stdout.

Where relevant, the Gzip, Zstd, and Lz4 decoding implementations were provided respectively by the Rust *flate*2, *zstd*, and *lz*4 libraries. All libraries used were the latest stable versions available at the time of writing, and selected in good faith to provide the best performance possible for each format and compression strategy.

For FASTA, FASTQ, and NAF, the parsing strategy was limited to a single-threaded implementation. We note that while the *seq_io* library does provide parallel processing of records, the underlying byte-stream parsing is inherently sequential.

The input format for this experiment was generated using *nucgen*, a Rust library for generating uniform random nucleotide sequences. The nucleotide sequences were generated with a uniform distribution of A, C, G, and T nucleotides. Quality scores were globally "?", simplified to a single value for each sequence. This simplification will result in better performance for existing compression formats, but does not make assumptions about the distribution of quality scores. Each experiment was performed with 10 Million records. For single-end reads each record had 100bp and for paired-end reads the primary sequence had 50bp and the secondary had 150bp.

2.3.2 K-mer Mapping

The k-mer mapping scenario is accomplished with a naive parallel implementation of k-mer counting. Briefly, each thread manages a thread-specific k-mer count table, implemented as a hashmap, and at regular batch intervals a global k-mer count table is locked, updated by the local table, and the local table counts are cleared. The k-mers are counted by sliding a window of size k through the record sequence and updating the local thread table via an $O(1)$ update.

The input data for this experiment was generated using wgsim[9], using the HGChr38 Chromosome 1 as reference and with increasing fixed length sequences of 100, 200, and 300 basepairs.

2.3.3 Alignment

The alignment scenario is a more complex scenario where we align incoming records to a reference genome. To accomplish this, we adapted *minimap*2[19], a flexible and high-performance read mapper, to accept BINSEQ as input.

We make use of the Rust library *minimap*2 which provides FFI bindings to the underlying C *minimap*2 library. We then created a simple command-line tool, *mmr*, that accepts BINSEQ or FASTQ as input, performs the alignment operation using *minimap*2, and outputs the resulting PAF file.

For our FASTQ parsing implementation, we use the *paraseq* library, which provides a minimal-copy parser for FASTQ files with a focus on parallel processing. We matched the default parameters of the standard *minimap*2 command-line tool and manage presets through the C interface.

We explored three different alignment scenarios, short-read whole-genome sequencing, long-read whole-genome sequencing, and long-read spliced sequencing. Notably, we excluded short-read spliced sequencing from this benchmark as *minimap*2 only recently announced support for this use-case and STAR is one of the most common short-read spliced aligners used in the field.

To simulate short-read sequences we used the wgsim[9] tool, using the HGChr38 Chromosome 1 as reference and with increasing fixed length sequences of 100, 200, and 300 basepairs. To simulate long-read sequences we used the pbsim3[20] tool. For the whole-genome sequences we used the "wgs" strategy, the qshmm error model (QSHMM-RSII), GHChr38 Chromosome 1 as reference, and increased mean length sizes from 1000

to 9000 base pairs (sequence lengths are variable). For the spliced sequences we used the "trans" strategy, the qshmm error model (QSHMM-RSII), and the GHChr38 Chromosome 1 cDNA library as reference.

For the short-read sequence evaluation we evaluate FASTQ and BINSEQ. For the long-read sequence evaluations we evaluate FASTQ and VBQ, excluding BQ as the sequences are of variable length. For all evaluations, VBQ is compressed and including quality scores to capture the worst-case.

To evaluate the performance of short-read spliced sequences we modified the STAR aligner[3] to accept BQ as an input format via C-bindings. This implementation minimally modified the STAR aligner and as such it is not fully optimized to take advantage of the BQ format. However, it provides a proof of concept that BQ can be used as an input format for other non-rust-based bioinformatics tools.

For this task we realigned the short-read single-cell sequencing data from the Replogle et al. study[21] using either BINSEQ or gzip compressed FASTQ as input. Notably, each file was processed independently and does not make use of any file-level parallelism.

3 Results

Our comprehensive evaluation demonstrates that the BINSEQ family of formats delivers substantial performance improvements across all tested bioinformatics workflows. BQ and VBQ achieved up to 90x faster processing than compressed FASTQ while maintaining comparable or reduced storage requirements. These improvements were most pronounced in parallel processing scenarios, where BINSEQ formats continued to scale efficiently with increasing thread counts while traditional formats quickly reached performance plateaus.

3.1 Sequence Access

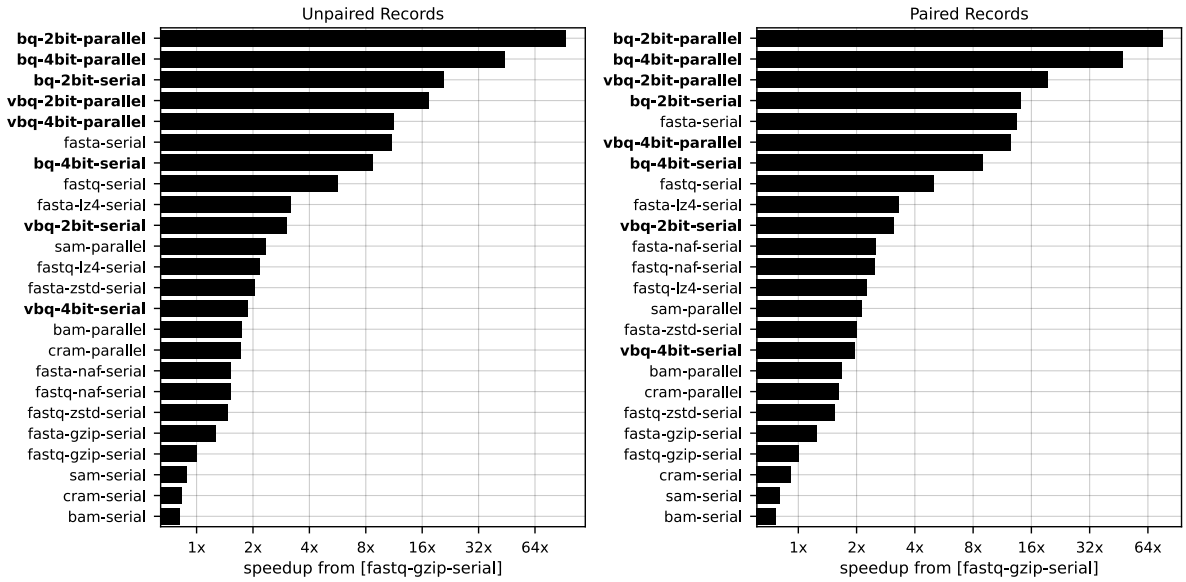

Figure 2: Speedup of file formats over FASTQ.GZ for unpaired (a) and paired (b) records. The speedup is measured as the ratio of the total elapsed time to process the record set between the test and reference condition (fastq.gz). Parallel implementations are run using 8 threads.

BINSEQ formats significantly outperform traditional sequence formats in both processing speed and storage efficiency. As shown in Table 2, multi-threaded BQ demonstrates the fastest record throughput among all tested formats, while multi-threaded VBQ achieves the best combined score for processing speed and file size.

When compared to the de facto standard gzip-compressed FASTQ, multi-threaded BQ achieves more than 90x higher throughput for both unpaired records (Figure 2). Multi-threaded VBQ similarly excels

with greater than 16x faster throughput. This dramatic performance improvement derives from BQ's fixed-size record structure and generally BINSEQ's elimination of sequential decompression bottlenecks that limit traditional formats.

Storage efficiency analysis reveals a competitive landscape among specialized formats. BQ (305.18 MB) maintains comparable file sizes to gzip-compressed FASTA (324.81 MB) and BAM (301.92 MB), while requiring less space than gzip-compressed FASTQ (347.43 MB). VBQ further reduces storage requirements (248.29 MB) to levels similar to CRAM (238.85 MB) and NAF (242.44 MB for FASTA, 242.47 MB for FASTQ). Notably, while CRAM and NAF achieve excellent compression efficiency, BQ and VBQ maintain superior decoding performance, especially in parallel processing paradigms.

The choice between two-bit and four-bit nucleotide encoding presents a clear performance-fidelity trade-off. Two-bit encoding provides maximum throughput and minimal storage, achieving the 90x speedups and 305.18 MB file sizes reported above for BQ. Four-bit encoding preserves ambiguous bases (N) at the cost of doubled storage (534.06 MB) and reduced throughput, though still maintaining substantial advantages over traditional formats—42x faster than gzip-compressed FASTQ for parallel BQ (Table 2). For VBQ, four-bit encoding (271.48 MB) requires approximately 9% more storage than two-bit (248.29 MB) while maintaining greater than 10x speedup over FASTQ. The selection between encoding schemes should be guided by application requirements: two-bit encoding is optimal for high-quality fixed-length data where ambiguous bases are rare (e.g., Illumina single-cell RNA-seq), while four-bit encoding is appropriate when base ambiguity information must be preserved (e.g., low-coverage sequencing, variant analysis, or archival storage).

The performance gap between BINSEQ formats and traditional approaches becomes more pronounced in parallel processing scenarios. While traditional formats like FASTQ, BAM, and CRAM show modest throughput improvements with parallel processing, they quickly reach saturation, limiting their effective utilization of modern multi-core architectures. In contrast, both BQ and VBQ scale efficiently with increasing thread counts, maintaining near-linear throughput improvements.

Our composite score (Equation 4), which balances processing speed and storage efficiency, ranks BQ and VBQ as optimal formats with scores of 0.017 and 0.0233 respectively, an order of magnitude better than other formats, with NAF (0.264 for FASTA, 0.267 for FASTQ) also showing competitive combined performance. While NAF excels in storage efficiency, BINSEQ formats maintain superior processing speed, particularly for parallel workloads.

Additional analysis of compression algorithms indicates that zstd compression achieves similar compression ratios to gzip but with significantly faster decompression rates for both FASTA and FASTQ. This finding provides additional evidence supporting the replacement of gzip as the de facto compression algorithm in bioinformatics pipelines.

Table 2: Comparison of different compression methods and their performance metrics across various file formats (single-end records). Time is reported in seconds, size in megabytes. Score is a combined metric of normalized time and size (Equation 4). The "Bit Size" column indicates the number of bits used to represent each nucleotide in the format. The "Parallel" column indicates whether the method was performed in parallel (8 threads). The "Bandwidth" column indicates the Giga-bases per second (Gbp/s). The "Speedup" column indicates the runtime speedup over FASTQ.GZ.

Format	Variant	Bit Size	Parallel (t=8)	Time (s)	Storage (MB)	Score	Bandwidth [Gbp/s]	Speedup [fastq.gz]
BINSEQ	bq	two	parallel	**0.0286**	305.18	**0.017**	**34.914**	**91.97x**
BINSEQ	vbq	two	parallel	0.1586	248.29	0.0233	6.304	16.61x
BINSEQ	bq	two	serial	0.122	305.18	0.0319	8.196	21.59x
BINSEQ	vbq	four	parallel	0.2463	271.48	0.0433	4.061	10.7x
BINSEQ	bq	four	parallel	0.0624	534.06	0.0809	16.03	42.23x
BINSEQ	bq	four	serial	0.2933	534.06	0.1179	3.41	8.98x
BINSEQ	vbq	two	serial	0.8615	248.29	0.1355	1.161	3.06x
FASTA	lz4	eight	serial	0.8109	588.39	0.215	1.233	3.25x
BINSEQ	vbq	four	serial	1.3593	271.48	0.222	0.736	1.94x
FASTA	zstd	eight	serial	1.2854	349.43	0.2258	0.778	2.05x
HTSlib	cram	four	parallel	1.4438	**238.85**	0.2272	0.693	1.82x
HTSlib	bam	four	parallel	1.4275	301.92	0.2407	0.701	1.85x
FASTA	uncompressed	eight	serial	0.2359	1086.13	0.2501	4.239	11.17x
FASTA	naf	eight	serial	1.677	242.44	0.2639	0.596	1.57x
FASTQ	naf	eight	serial	1.6876	242.47	0.2665	0.593	1.56x
FASTQ	lz4	eight	serial	1.1994	599.43	0.2792	0.834	2.2x
FASTQ	zstd	eight	serial	1.7356	367.84	0.3113	0.576	1.52x
FASTA	gzip	eight	serial	2.0766	324.81	0.3508	0.482	1.27x
FASTQ	gzip	eight	serial	2.6342	347.43	0.446	0.38	1x
HTSlib	cram	four	serial	2.8813	238.85	0.4574	0.347	0.91x
HTSlib	bam	four	serial	3.1463	301.92	0.5161	0.318	0.84x
FASTQ	uncompressed	eight	serial	0.4711	2068.41	0.5387	2.123	5.59x
HTSlib	sam	eight	parallel	1.0815	2192.39	0.6693	0.925	2.44x
HTSlib	sam	eight	serial	2.8056	2192.39	0.944	0.356	0.94x

3.2 K-mer Counting

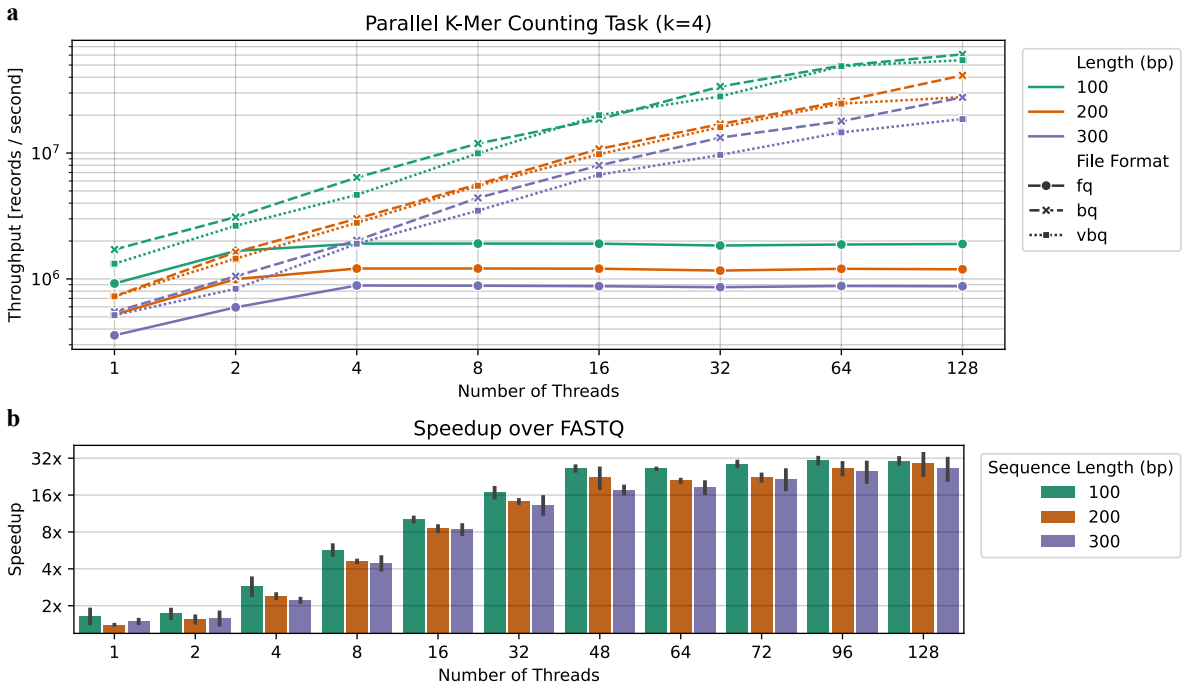

Figure 3: Runtime characteristics of **kmer-count** *as a function of the number of threads and format: FASTQ (fq), BQ (bq), VBQ (vbq). (a) Record processing throughput as a function of the number of threads used. (b) Speedup (measured as the ratio of the record throughput per second) of BINSEQ over FASTQ as a function of number of threads.*

The BINSEQ formats deliver substantial performance advantages for k-mer counting, a fundamental operation in many bioinformatics algorithms. As illustrated in Figure 3a, both BQ and VBQ maintain near-linear scaling of record throughput as thread count increases, continuing to improve even at 128 threads. In stark contrast, FASTQ reaches performance saturation at just 2-4 threads, depending on sequence length, gaining no additional throughput despite allocated computational resources.

This performance differential becomes more pronounced at higher thread counts. By 128 threads, BINSEQ formats achieve near 32x greater throughput per record compared to FASTQ (Figure 3b). This remarkable improvement demonstrates how BINSEQ's architecture effectively overcomes the I/O bottlenecks that constrain traditional formats, allowing full utilization of modern parallel computing resources.

The impact of sequence length on performance scaling is particularly noteworthy. While all formats show some performance reduction with longer sequences, the BINSEQ family maintains its scaling advantage across all tested sequence lengths (100, 200, and 300 bp). This consistent behavior indicates that the performance benefits of BINSEQ formats are robust across varying data characteristics.

K-mer counting, while computationally straightforward, represents a common low-complexity task that forms the foundation of many bioinformatics tools, including de Bruijn graph assemblers, taxonomic classifiers, and sequence similarity analyzers. The demonstrated performance improvements suggest that BINSEQ formats would be particularly beneficial for these applications, especially in high-throughput environments where processing efficiency directly impacts analysis turnaround time.

The ability of BINSEQ formats to maintain scaling efficiency at high thread counts indicates that they effectively shift the performance bottleneck from I/O operations to computational resources, allowing bioinformatics applications to fully leverage available CPU cores. This characteristic is especially valuable in shared computing environments and cloud platforms where computational resources must be efficiently utilized.

3.3 Alignment

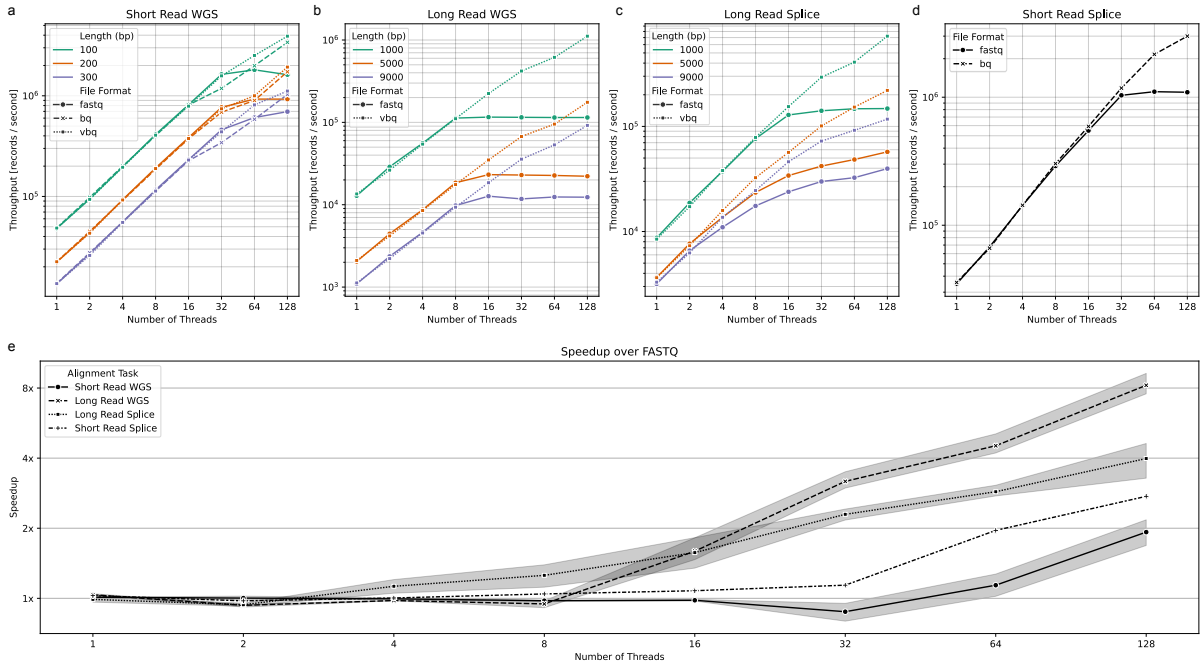

Figure 4: Runtime characteristics of **mmr** *(a-c) and* **STAR** *(d) as a function of the number of threads, format (FASTQ or VBQ), and alignment task. (a) Short read whole genome sequencing (preset = "sr") (b) Long read whole genome sequencing (preset = "map-pb") (c) Long read splicing (preset = "splice") (d) Short read splicing (STAR) (e) Speedup of VBQ or BQ over FASTQ at each thread count for each of the alignment tasks.*

Sequence alignment represents a more computationally intensive operation than previous benchmarks, yet VBQ still demonstrates significant performance advantages over FASTQ at higher thread counts (Figure 4). This performance difference becomes more pronounced as parallelization increases, showing that BINSEQ formats maintain their advantages even in complex analytical workflows.

Our analysis reveals that the performance divergence point between VBQ and FASTQ depends on both the specific alignment task and the average read length. For short-read whole-genome sequencing, FASTQ and VBQ show comparable performance up to 32 threads across all tested read lengths (Figure 4a), after which VBQ begins to pull ahead. In contrast, for long-read whole-genome sequencing, the performance separation occurs earlier, at approximately 16 threads for all tested read lengths (Figure 4b).

The most striking results appear in the long-read splicing alignment task, where separation points vary with sequence length (Figure 4c). For reads averaging 1000bp, performance diverges at 16 threads, while for 9000bp reads, the separation occurs as early as 4 threads. Counterintuitively, this complex spliced alignment task shows that VBQ's advantage becomes more pronounced with longer read lengths, where one might expect computational demands to overshadow I/O limitations. This pattern suggests that VBQ's performance advantage stems not merely from more efficient thread utilization but from fundamentally faster record loading to worker threads. Even in this computation-heavy task with 9000bp reads, FASTQ fails to achieve thread saturation at 128 threads, indicating that the format itself, rather than computational resources, remains a limiting factor.

We also observe a significant divergence in performance between BQ and FASTQ formats for short-read spliced alignments using STAR, where BQ achieves greater than 2x speedup over FASTQ at 128 threads (Figure 4d). This is significant in the case of STAR where the core mapping algorithm was built around streaming records and remains fairly unchanged in this context, so even greater gains can be expected in the future.

The relative speedup of VBQ over FASTQ varies across alignment tasks (Figure 4e), with the most substantial improvements observed in short-read whole-genome and long-read whole-genome sequencing

at higher thread counts. All tested scenarios show increasing performance advantages as thread count rises, with no observed plateau for VBQ even at 128 threads.

These results demonstrate that VBQ's benefits extend beyond simple I/O-bound tasks to complex analytical workflows central to genomic research. The format's ability to maintain scaling efficiency with increasing parallelization makes it particularly valuable for computationally intensive applications running on high-performance computing infrastructure.

4 Discussion

The development and evaluation of the BINSEQ family of formats demonstrates that significant performance improvements in sequence data processing can be achieved through careful consideration of modern genomics workflows and their computational requirements. By focusing on the core requirements of many bioinformatics applications — efficient access to nucleotide sequences — BQ and VBQ achieve substantial gains in both processing speed and storage efficiency.

The simplicity of BQ's design is one of its key strengths. The format specification is concise and straightforward, consisting of a minimal header structure and a uniform record format that leverages established bit-encoding techniques. This simplicity not only makes the format easy to implement and validate but also contributes to its robust performance characteristics. The use of fixed-size records and two-bit or four-bit nucleotide encodings provide inherent advantages that manifest across various usage scenarios, from basic sequence access to complex analytical workflows. Its definition allows for truly random access and enables tools that were previously bound to a single thread to fully utilize modern multi-core processors.

VBQ, while more complex in implementation, balances flexibility with performance. Although its block-based approach sacrifices the true random access capability of BQ, it compensates by supporting variable-length sequences, quality scores, and optional compression, all while maintaining impressive parallel processing performance. Compared to the de facto standard of gzip-compressed FASTQ, VBQ shows significant improvements with minimal data loss, primarily omitting only sequence headers and invalid nucleotides.

Our performance evaluations demonstrate that both formats translate their design advantages into substantial practical benefits. They consistently outperform traditional approaches across different operational contexts, showing particular advantages in parallel processing scenarios. The ability to scale effectively to high thread counts—achieving up to 90x improvement over compressed FASTQ for sequence access and up to 32x improvement for k-mer based operations—represents a significant advancement for high-throughput genomics applications. This scalability is particularly relevant as the field continues to move toward larger datasets, more computationally intensive analyses, and increased computational availability.

The parallelization advantages of BINSEQ formats operate at the file format level and complement other common parallelization strategies. Large experiments distributed across multiple files can employ file-level parallelism (processing each file independently), while BINSEQ enables efficient within-file parallelism regardless of data organization. These approaches are composable, workflows can process multiple BINSEQ files in parallel while each file is internally multi-threaded—maximizing utilization of available computational resources across diverse computing environments.

Importantly, we found that the performance advantages of BINSEQ formats extend beyond simple data retrieval tasks. Even in computationally intensive operations like sequence alignment, where the processing time per record is substantially higher, both formats demonstrated improved thread utilization and record throughput compared to traditional formats. This suggests that the bottleneck created by sequential parsing and decompression in FASTQ affects not just I/O-bound applications but also more complex analytical workflows.

The integration of BINSEQ formats with existing tools, demonstrated through our adaptation of minimap2, shows that the formats can be readily incorporated into established bioinformatics pipelines. The provided library and command-line tools offer a framework that simplifies both the adoption of BINSEQ formats in existing applications and the development of new tools that can take full advantage of their performance characteristics. The hook-based parallel processing interface, in particular, provides a flexible foundation for building high-performance genomic analysis tools.

For practitioners, the choice between BQ and VBQ presents a clear trade-off between performance and flexibility. BQ is optimized for maximum throughput and minimal storage with fixed-length sequencing data, making it ideal for platforms like Illumina that produce standardized reads. VBQ offers greater versatility for variable-length sequences and applications where quality scores remain important, such as long-read sequencing from PacBio or Oxford Nanopore technologies, with only a modest performance decrease compared to BQ but still substantial improvements over traditional formats.

The trade-offs inherent in the BINSEQ family's design reflect a broader consideration in bioinformatics tool development: the balance between generality and optimization for common use cases. While FASTQ's flexibility has contributed to its widespread adoption, our results suggest that there is significant value in optimizing for the specific requirements of modern high-throughput sequencing applications. One important consideration is the handling of ambiguous nucleotides (N bases), which we address through configurable policies (skip, random replacement, fixed replacement, or preservation via four-bit encoding) rather than prescribing a single approach. The optimal policy is application-dependent—ambiguous bases in single-cell barcodes may require different treatment than those in genomic reads for variant calling. This same consideration applies to quality scores, which can be optionally preserved or discarded depending on the specific needs of the application. While this consideration applies to all sequence formats, BINSEQ provides explicit policy choices at the format level while maintaining high performance regardless of policy selected. The performance improvements demonstrated by BQ and VBQ indicate that the bioinformatics community might benefit from reconsidering the one-size-fits-all approach to sequence data storage, particularly as dataset sizes continue to grow and computational efficiency becomes increasingly critical.

Looking forward, the BINSEQ architecture provides a foundation for further optimization and extension. The reserved space in both formats' headers and the flexible flag fields offer clear paths for adding features while maintaining backward compatibility. Future development could explore additional compression schemes, integration with cloud-native storage, handling of higher-order segments per record, and structured data provenance, all while preserving the core performance benefits of the current implementation.

In conclusion, the BINSEQ family represents a significant step forward in sequence data storage and processing efficiency. Their combination of simplicity, performance, and practicality makes them valuable additions to the bioinformatics toolkit, particularly for applications where computational efficiency is paramount. While not complete replacements for existing formats in all scenarios, BQ and VBQ demonstrate that substantial improvements in processing efficiency are achievable through careful consideration of modern genomics workflows and their computational requirements.

5 Code Availability

All software listed is available free and open-source.

Name	Description	URL
binseq	Library for BINSEQ IO	https://github.com/arcinstitute/binseq
bitnuc	Library for SIMD two-bit and four-bit operations	https://github.com/noamteyssier/bitnuc
binseq-bindings	Multi-language bindings for BINSEQ	https://github.com/arcinstitute/binseq-bindings
bsb	Parsing implementations for benchmarking	https://github.com/noamteyssier/binseq_benchmark
kmer_count	Naive parallel k-mer counting for benchmarking	https://github.com/arcinstitute/kmer-count
mmr	Minimap2 bindings with BINSEQ input support	https://github.com/arcinstitute/mmr
STAR	STAR branch with BINSEQ input	https://github.com/alexdobin/STAR/tree/binseq

6 Acknowledgments

The authors would like to thank David Holtz for his help on improving BINSEQ and its ecosystem, as well as Hani Goodarzi, Nicholas D. Youngblut, Yusuf H. Roohani, Abhinav Adduri, David P. Burke, and the rest of the Arc Institute computational team for their helpful suggestions and feedback.

Bibliography

[1] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants," *Nucleic Acids Research*, vol. 38, no. 6, pp. 1767–1771, Apr. 2010, doi: 10.1093/nar/gkp1137.

[2] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009, doi: 10.1093/bioinformatics/btp324.

[3] A. Dobin *et al.*, "STAR: ultrafast universal RNA-seq aligner," *Bioinformatics*, vol. 29, no. 1, pp. 15–21, Jan. 2013, doi: 10.1093/bioinformatics/bts635.

[4] R. Patro, G. Duggal, M. I. Love, R. A. Irizarry, and C. Kingsford, "Salmon provides fast and bias-aware quantification of transcript expression," *Nat Methods*, vol. 14, no. 4, pp. 417–419, Apr. 2017, doi: 10.1038/nmeth.4197.

[5] N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter, "Near-optimal probabilistic RNA-seq quantification," *Nat Biotechnol*, vol. 34, no. 5, pp. 525–527, May 2016, doi: 10.1038/nbt.3519.

[6] A. Bankevich *et al.*, "SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, May 2012, doi: 10.1089/cmb.2012.0021.

[7] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences." Accessed: Mar. 18, 2025. [Online]. Available: http://arxiv.org/abs/1512.01801

[8] B. Langmead, C. Wilks, V. Antonescu, and R. Charles, "Scaling read aligners to hundreds of threads on general-purpose processors," *Bioinformatics*, vol. 35, no. 3, pp. 421–432, Feb. 2019, doi: 10.1093/bioinformatics/bty648.

[9] H. Li *et al.*, "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, Aug. 2009, doi: 10.1093/bioinformatics/btp352.

[10] G. Cochrane *et al.*, "Facing growth in the European Nucleotide Archive," *Nucleic Acids Research*, vol. 41, no. D1, pp. D30–D35, Nov. 2012, doi: 10.1093/nar/gks1175.

[11] A. Al-Okaily, B. Almarri, S. Al Yami, and C.-H. Huang, "Toward a Better Compression for DNA Sequences Using Huffman Encoding," *Journal of Computational Biology*, vol. 24, no. 4, pp. 280–288, Apr. 2017, doi: 10.1089/cmb.2016.0151.

[12] M. H. Mohammed, A. Dutta, T. Bose, S. Chadaram, and S. S. Mande, "DELIMINATE—a fast and efficient method for loss-less compression of genomic sequences," *Bioinformatics*, vol. 28, no. 19, pp. 2527–2529, Oct. 2012, doi: 10.1093/bioinformatics/bts467.

[13] G. Benoit *et al.*, "Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph," *BMC Bioinformatics*, vol. 16, no. 1, p. 288, Dec. 2015, doi: 10.1186/s12859-015-0709-7.

[14] K. Kryukov, M. T. Ueda, S. Nakagawa, and T. Imanishi, "Nucleotide Archival Format (NAF) enables efficient lossless reference-free compression of DNA sequences," *Bioinformatics*, vol. 35, no. 19, pp. 3826–3828, Oct. 2019, doi: 10.1093/bioinformatics/btz144.

[15] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison.," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 85, no. 8, pp. 2444–2448, Apr. 1988, doi: 10.1073/pnas.85.8.2444.

[16] M. Schlegel, "seq_io." [Online]. Available: https://github.com/markschl/seq_io

[17] J. Köster, "rust_htslib." [Online]. Available: https://github.com/rust-bio/rust-htslib

[18] J. K. Bonfield *et al.*, "HTSlib: C library for reading/writing high-throughput sequencing data," *GigaScience*, vol. 10, no. 2, 2021, doi: 10.1093/gigascience/giab007.

[19] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, Sep. 2018, doi: 10.1093/bioinformatics/bty191.

[20] Y. Ono, M. Hamada, and K. Asai, "PBSIM3: a simulator for all types of PacBio and ONT long reads," *NAR Genomics and Bioinformatics*, vol. 4, no. 4, p. lqac92, Oct. 2022, doi: 10.1093/nargab/lqac092.

[21] J. M. Replogle *et al.*, "Maximizing CRISPRi efficacy and accessibility with dual-sgRNA libraries and optimal effectors," *eLife*, vol. 11, p. e81856, Dec. 2022, doi: 10.7554/eLife.81856.

1 Supplementary Materials and Data

1.1 Format Specifications

1.1.1 Shared Specification

Table 4: Nucleotide Encoding. Standard nucleotides use two-bit encoding. In four-bit mode, these values are preserved in the least significant bits with leading zeros, while N requires the full four-bit representation.

Nucleotide	two-bit	four-bit
A	00	0000
C	01	0001
G	10	0010
T	11	0011
N	—	1111

1.1.2 BQ

Table 5: BQ Header Structure (32 bytes)

Offset	Size (bytes)	Field	Type	Description
0	4	magic	uint32	Magic number (0x42534551)
4	1	version	uint8	Format version (currently 2)
5	4	slen	uint32	Sequence length (primary)
9	4	xlen	uint32	Sequence length (secondary)
13	1	bits	uint8	Number of bits per nucleotide (2, 4)
14	1	flags	bool	Records are prefixed by a flag uint64
15	17	reserved	[uint8]	Reserved for future extensions

1.1.3 VBQ

Table 6: VBQ Header Structure (32 bytes)

Offset	Size (bytes)	Field	Type	Description
0	4	magic	uint32	Magic number (0x51455356)
4	1	format	uint8	Format version (currently 1)
5	8	block	uint64	Virtual size of all blocks in bytes
13	1	qual	bool	Records include quality scores
14	1	compressed	bool	Blocks are compressed
15	1	paired	bool	Records consist of sequence pairs
16	1	bits	uint8	Number of bits per nucleotide (2, 4)
17	1	headers	bool	Records include sequence headers
18	1	flags	bool	Records are prefixed by a flag uint64
19	13	reserved	[uint8]	Reserved bytes for future extensions

Table 7: VBQ Block Header (32 bytes)

Offset	Size (bytes)	Field	Type	Description
0	8	magic	uint64	Magic number (0x5145534B434F4C42)
8	8	size	uint64	True size of record block (can vary if compressed)
16	4	records	uint32	Number of records in block
20	12	reserved	[uint8]	Reserved bytes for future extensions

Table 8: VBQ Record. B represents the number of bits per word and is 32 for two-bit encodings or 16 for four-bit.

Field	Type	Size (bytes)	Description
flag	uint64	8	Binary flag for the record
slen	uint64	8	Primary sequence length
xlen	uint64	8	Extended sequence length
sbuf	[uint64]	$\lceil \frac{\text{slen}}{B} \rceil$	Encoded primary sequence
squal	[uint8]	slen if paired else 0	Primary sequence quality scores
sheader	[uint8]	8 + Variable	Sequence Header. The first uint64 (8 bytes) defines the length in bytes of the header. The remaining bytes contain the header data.
xbuf	[uint64]	$\lceil \frac{\text{xlen}}{B} \rceil$	Encoded extended sequence
xqual	[uint8]	xlen if paired else 0	Extended sequence quality scores
xheader	[uint8]	8 + Variable	Sequence Header. The first uint64 (8 bytes) defines the length in bytes of the header. The remaining bytes contain the header data.

Table 9: VBQ Index Header (32 bytes)

Offset	Size (bytes)	Field	Type	Description
0	8	magic	uint64	Magic number (0x5845444e49514256)
8	8	bytes	uint64	Number of bytes in the file pair)
16	16	reserved	[uint8]	Reserved bytes for future extensions

Table 10: VBQ Index Range (32 bytes)

Offset	Size (bytes)	Field	Type	Description
0	8	offset	uint64	File offset of block start (bytes)
8	8	len	uint64	Size of the block (bytes)
16	4	records	uint64	Number of records in block
20	4	cumulative	uint64	Number of records before block
24	8	reserved	[uint8]	Reserved bytes for future extensions

1.2 Processing

Table 11: Parallel Processing Hooks

Hook	Type	Description
process_record	map	Process a single record (primary / extended)
on_batch_complete	reduce	Called when a batch of records has completed processing