High-Throughput Compression of FASTQ Data with SeqDB

Mark Howison

Abstract—Compression has become a critical step in storing Next-Generation Sequencing data sets because of both the increasing size and decreasing costs of such data. Recent research into efficiently compressing sequence data has focused largely on improving compression ratios. Yet, the throughputs of current methods now lag far behind the I/O bandwidths of modern storage systems. As biologists move their analyses to high-performance systems with greater I/O bandwidth, low-throughput compression becomes a limiting factor. To address this gap, we present a new storage model called SeqDB, which offers high-throughput compression of sequence data with minimal sacrifice in compression ratio. It achieves this by combining the existing multi-threaded Blosc compressor with a new data-parallel byte-packing scheme, called SeqPack, which interleaves sequence data and quality scores.

Index Terms—Compression, data storage, Next-Generation Sequencing, FASTQ

1 Introduction

The *de facto* standard for storing raw data from Next-Generation Sequencing (NGS) platforms is the FASTQ format [1], which stores sequence data with corresponding quality scores. FASTQ, and the related FASTA format, have also become a standard for data interchange among bioinformatics tools. Unfortunately, both FASTQ and FASTA suffer from a limitation inherent to most ASCII-based formats: poor space efficiency when storing binary data. Furthermore, their use of variable-length records inhibits random access.

SeqDB is a file format, compressor and storage tool for NGS data sets that addresses both of these issues, reducing storage requirements by as much as 62% and enabling efficient random access, but without breaking backward compatibility. SeqDB files can be mounted as FASTQ files and read by existing tools that expect FASTQ input.

Many compression methods and file formats for sequence data also address one or both of these issues. However, they favor compression ratio over throughput, and fail to take advantage of the parallelism available on modern multi-core CPUs. SeqDB, on the other hand, achieves similar compression ratios to `zlib`, but with considerably better throughput thanks to its use of threaded parallelism, a byte-packing scheme called SeqPack, and the high-performance Blosc compression library.

The poor throughput of existing compression methods may not be apparent on laptops or workstations where I/O bandwidth is already lower than the compresion's throughput. However, on systems where more I/O bandwidth is available – such as through a RAID array, SSD drive, or high-performance parallel file system – the low throughput of existing compression tools is a limiting factor. As more high-performance systems become available to biologists through avenues like cloud computing and scientific grid computing, high-throughput storage models like SeqDB will become increasingly necessary.

SeqDB is available for non-commercial use under an open-source license from:

https://bitbucket.org/mhowison/seqdb

2 Related Work

SeqDB uses Blosc [2], a high-performance compressor that was originally developed as part of the PyTables and `carray` projects to accelerate memory-bound computations for algebraic calculations in Python [3]. Blosc can provide faster-than-memory access to compressed binary data thanks to two optimization approaches: cache-aware blocking and parallelization through SIMD vectorization and multi-threading [4]. Blosc borrows heavily from the FastLZ compression library [5].

SeqDB abstracts the storage layer and can be extended to use any file container or I/O library that provides block access to arrays of records. Initially, we have implemented a storage backend with the Hierarchical Data Format v5 (HDF5) [6], a general-purpose and high-performance I/O library for scientific data. HDF5 takes care of many low-level storage details, like laying out and chunking 2D arrays in the 1D file space and correcting for byte endianness across different architectures.

Motivated by these same benefits of building on top of an existing I/O library, Geospiza and The HDF Group [7] created a general-purpose genomics format called BioHDF based on the HDF5 library and compressed with

• *The author is with the Center for Computation and Visualization, Brown University, Providence, RI, 02912.*
E-mail: mhowison@brown.edu

zlib. It is designed to support a wide range of genomics data, including variable-length sequences, alignments and annotations. In contrast, SeqDB's goals are much narrower, targetting only fixed-length sequence data. Internally, BioHDF represents data as either variable-length 1D concatenated strings or fixed-length 2D arrays of characters, whereas SeqDB only uses fixed-length 2D arrays. In our results, we show that modifying BioHDF to use Blosc compression improves throughput, but that BioHDF still ranks below SeqDB in both compression ratio and throughput for storing fixed-length NGS data.

Several projects have improved the compression ratio beyond what zlib can achieve for sequence data, typically by leveraging additional assumptions or features specific to nucleotide sequences. Compression ratio is an important metric of sequence storage methods because for large repositories, like the Sequence Read Archive [8], the growth in submitted NGS data is outpacing the growth in storage capacity.

Wan and Asai [9] showed that pre-sorting sequences using radix or quicksort can improve compression ratios for gzip and bzip2 compression, while also reducing compression time. These methods, however, were designed for sequence data where the quality scores have been removed, and therefore do not suport FASTQ-formatted data.

G-SQZ, a compression scheme based on Huffman coding and designed specifically for sequence data by Tembe et al. [10], also achieves high compression ratios, but requires interpretation of sequence IDs, and only supports those from the SOLEXA and SOLiD platforms. In contrast, SeqDB stores unmodified IDs, and can accomodate arbitrary data as long as it is in FASTQ format and has a fixed sequence length.

The DSRC compressor by Deorowicz and Grabowski [11] is the closest to SeqDB in terms of both goals and features. Like SeqDB, DSRC uses blocked storage of compressed sequences to enable fast random access, supports arbitrary IDs, and combines sequence and quality score data prior to compression. In their experiments, Deorowicz and Grabowski found that no compression methods could surpass gzip in decompression throughput. As we will present in our results, SeqDB does surpass gzip (and DSRC) in both compression and decompression throughput, by a wide margin.

Although SeqDB cannot achieve compression ratios as low as DSRC, this loss is offset by a much larger gain in throughput. In some sense, SeqDB and DSRC have taken orthogonal approaches to improving gzip compression of sequence data: whereas DSRC prioritizes low compression ratios, SeqDB prioritizes high throughput.

3 Design

SeqDB is made up of the following utility programs, all accessible through a common wrapper script:

- `seqdb-compress` parses an existing FASTQ file, compresses its data with SeqPack and Blosc, and stores it to a SeqDB file.
- `seqdb-extract` decompresses a SeqDB file and outputs it in FASTQ format to a pipe or file.
- `seqdb-profile` parses a FASTQ file to generate a histogram of sequence and ID lengths. These values are required prior to conversion from FASTQ, since SeqDB uses fixed dimensions to store sequences and IDs as arrays.

A C++ API is also available for application developers who wish to read or write SeqDB files natively.

3.1 Compression with SeqPack and Blosc

SeqDB uses a two-pass compression scheme of SeqPack followed by Blosc. SeqPack is a byte-packing method that interleaves nucleotide sequences with their corresponding quality scores. FASTQ represents raw sequence data inefficiently because it stores each sequence value as two ASCII bytes: one byte stores 5 possible bases (`N`, `A`, `T`, `C`, `G`) and the other stores a quality score with up to 41 possible values in most cases. Through SeqPack, this data is instead represented as a 2D array with dimensions 5×51, requiring only one byte of storage. SeqPack uses a pair of lookup tables to efficienty convert between the two-byte FASTQ representation and one-byte SeqPack representation.

Two encoding tables, `enc_base[128]` and `enc_qual[128]`, are indexed by ASCII code and provide the (x, y) coordinates into the 2D byte array. These tables are constructed so that bases (`N`, `A`, `T`, `C`, `G`) map to x-coordinates $(0..4)$, and the quality scores map to y-coordinates $(0..50)$. For example, a base `T` with score 29 (ASCII character `'>'` in Phred+33 encoding) is encoded as:

```
enc_base['T']*51 + enc_qual['>'] = 0x83
```

Two decoding tables, `dec_base[256]` and `dec_qual[256]`, are indexed by the SeqPack byte and return the base and the quality score, respectively. For example, the SeqPack byte `0x83` decodes as:

```
dec_base[0x83] = 'T'
dec_qual[0x83] = '>'
```

The lookup tables are pre-computed when SeqPack is initialized, and are also stored as an attribute in the HDF5 container, to provide provenance for the compression operation and recoverability for the data. That is, even in the abscence of the SeqDB or SeqPack implementation, the lookup tables stored in the HDF5 file contain sufficient information to decode the interleaved sequences and quality scores.

3.2 Block Storage and Buffering

For efficiency, SeqDB maintains an internal buffer so that compression and decompression can be performed on an entire block, rather than on individual records. The block size can be decreased to allow for more efficient random access (since an entire block must be loaded

even to read a single sequence), or increased to allow for more buffering, which may improve contiguous access. The block size defaults to 16,384 records, but can be overridden by the user. We chose this default value because it is the block size that achieved the peak decompression rate for the SeqPack/Blosc compression scheme in the memory bandwidth test presented below (see Figure 1).

Any storage backend that can provide block access to a 2D array can be used with SeqDB. Currently, we use the HDF5 library with chunked 2D arrays, where the chunk size matches the SeqDB block size. We store IDs and sequences in separate arrays, which each have a fixed dimension in the fastest moving direction of the maximum ID length or maximum sequence length, and an unlimited dimension in the slower direction.

Although SeqDB isn't designed to store variable-length sequences, we note that it will work correctly when using the maximum sequence length as the fixed dimension in the 2D array. SeqDB will parse and print the FASTQ representation with a newline character terminating the sequence line, and will store missing bases and quality scores as null characters. Although this is not the most efficient design for storing variable-length sequences, we will show below that for modest ranges in sequence length it works reasonably well in practice.

3.3 FASTQ Compability

SeqDB uses named pipes to provide backward compability with tools that expect FASTQ input. Named pipes are part of the POSIX standard and can be created on most modern UNIX operating systems, like Linux and Mac OS X, using the `mkfifo` command[1].

For ease of use, SeqDB provides a wrapper script that lets users "mount" a SeqDB file to a path. The mount script creates a named pipe at the path and spawns a `fastq-extract` process in the background to stream the contents of the SeqDB file in FASTQ format to the pipe. If the background process ends (for instance, if a program reading the pipe reaches the end-of-file), it prints a warning message to the user that the path has been "unmounted" and removes the named pipe. The user can also manually unmount the path using the script.

4 Results

We tested the efficiency of SeqDB and SeqPack against existing compression methods using a collection of publicly available NGS data sets[2], summarized in Table 1. These data sets were chosen to cover a variety of sequencing technologies, read lengths, ID lengths, and total data sizes (a span of two orders of magnitude).

1. http://pubs.opengroup.org/onlinepubs/009695399/functions/mkfifo.html

2. All data were downloaded in SRA format and converted to FASTQ using the command `fastq-dump --defline-qual +` from the SRA Toolkit.

Our tests were designed to answer the following three questions:

1) What is the maximum, in-memory throughput of SeqPack compared to other compressors?
2) How does the compression ratio, throughput and random-access latency of SeqDB compare to other storage models?
3) For real applications, does SeqDB's backward compatibility add any overhead versus reading in a FASTQ file directly?

4.1 Test System and I/O Bandwidth

All tests were conducted on IBM iDataPlex nodes at the Brown University Center for Computation and Visualization. The nodes feature dual-socket 6-core Intel Xeon X5650 2.66 Ghz processors and 96GB of RAM; their operating system is CentOS 6.3. All programs were compiled with the Intel compiler suite version 12.1.5 using the flags `-O3 -msse4.2`. For all tests, we had exclusive use of the test node (aside from system processes).

Although these nodes are serviced by a high-performance GPFS file system, that file system is also shared by nearly 300 other nodes at the Center, and is therefore subject to large variations in bandwith because of contention with other users' jobs. This makes any kind of I/O-sensitive timing difficult. Because of the large RAM footprint of the nodes, we decided to instead conduct all I/O out of the node's `/dev/shm` RAM disk. Using repeated calls to the `cat` command to redirect each of the data sets to a new file, we measured bidirectional bandwidths ranging from 1542MB/s to 1728MB/s.

These bandwidths are on par with the typical bandwidths we see on the Center's GPFS file system, which have ranged between 1GB/s to 2GB/s (unidirectional) on daily benchmarks run over the past two years. Bandwidths on the order of GB/s are also representative of the peak bandwidths available on high-performance parallel file systems at major supercomputing centers, for example 35 GB/s on Hopper at the National Energy Research Scientific Computing Center[3] or 15-20 GB/s on Intrepid at the Argonne Leadership Computing Facility[4].

4.2 Threading and Block Size

Both Blosc and SeqPack use threaded parallelism. Blosc explicitly manage a pool of worker threads with the `pthreads` library, and SeqPack implicitly parallelizes its inner packing and unpacking loops with OpenMP `parallel for` directives. For both, we allocated 12 threads to match the 12 cores available on the test nodes.

We pinned OpenMP threads to cores using the `KMP_AFFINITY` environment variable provided by the Intel OpenMP library, and found that while this did not change the peak performance, it led to less variability in

3. http://www.nersc.gov/users/computational-systems/hopper/file-storage-and-i-o/

4. http://www.alcf.anl.gov/resource-guides/intrepid-file-systems

TABLE 1
Test Data Sets

#	Project	Species	SRA ID	Sequencer	Read Count	Read Length	ID Lengths	FASTQ Size
1	Foodborne Pathgen Survey	Salmonella	SRR493328	Illumina MiSeq	2,031,674	302 bp	64–72	1,317 MB
2	1000 Genomes	Human	ERR000018	Illumina GA	9,612,363	36 bp	49–60	1,235 MB
3	1000 Genomes Pilot	Human	SRR003177	LS 454 GS FLX Titanium	1,504,571	45–4398 bp	38–45	1,701 MB
4	Ion Torrent Bias Experiment	*E. coli*	SRR611141	Ion Torrent PGM	4,853,655	6–406 bp	24–37	1,637 MB
5	Maize HapMap II	*Z. mays*	SRR448020	Illumina GA IIx	19,030,772	172 bp	48–62	7,412 MB
6	1000 Genomes	Human	SRR493233	Illumina HiSeq 2000	43,280,168	200 bp	60–70	19,557 MB

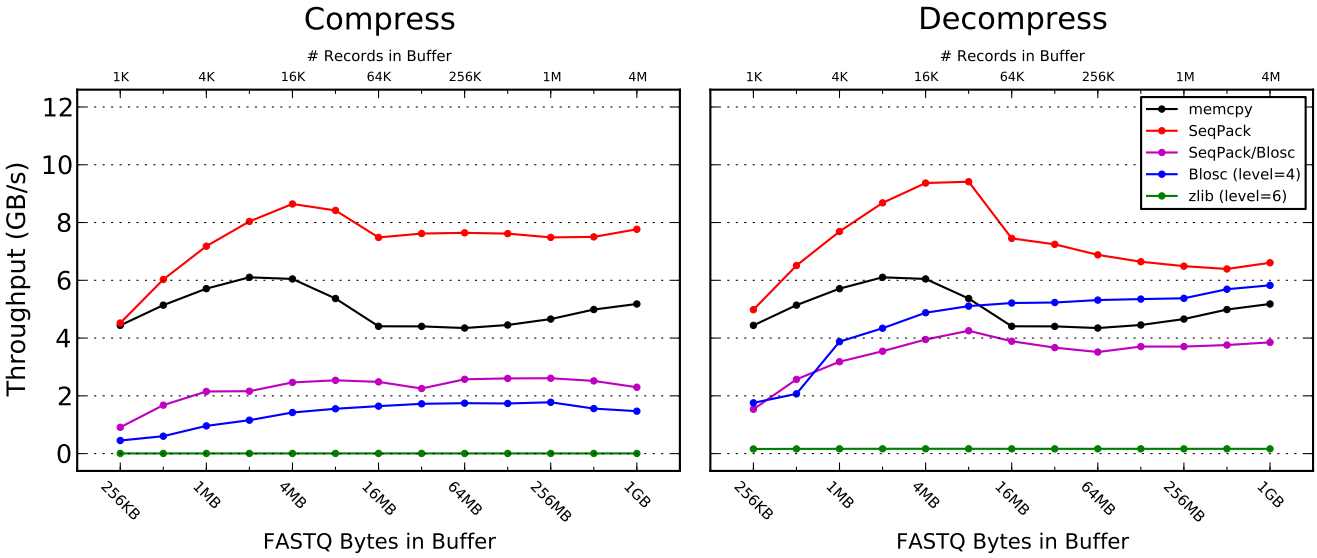

Fig. 1. In-memory throughputs for several compression schemes applied to buffers with increasing numbers of records. Each FASTQ record is 256 bytes.

measured throughputs. Therefore , we also pinned the Blosc threads by modifying the source code to include the Linux `pthread_setaffinity_np()` system call.

For the tunable SeqDB block size parameter, we used the default size of 16,384 records for the compression ratio and throughput tests. For the latency tests, we used a smaller block size of 1,024 records.

4.3 In-memory Throughput

To determine the maximum, in-memory throughput of `zlib` (version 1.2.3, with compression level of 6), Blosc (version 1.1.5, with compression level 4) and SeqPack (version 0.2.0), we loaded increasingly larger subsets of FASTQ data into a memory buffer, timed its compression into a second buffer, then timed the decompression back into the first buffer. We repeated each of these tests 100 times and kept the minimum runtime, which represents the best-case scenario when background noise from system processes was at its lowest.[5] As a baseline, we also performed a `memcpy` of the first buffer to the second, which measures the bidirectional memory bandwidth of the system. We calculated throughput as the size of the original, uncompressed block divided by runtime.

For these tests, we used a filtered version of the SRR493233 data set downloaded directly from 1000 Genomes rather than SRA. It has reads of length 100 bp, and we truncated the IDs to 56 characters so that records were stored with 256 bytes (100 bytes for sequence and quality score data, and 56 bytes for the ID header). This allows for an easier comparison between record count and FASTQ bytes, since 4 records correspond to 1 KB. We tested buffer sizes on a logarithmic scale from 256KB to 1GB (corresponding to 1K to 4M records), reading each buffer from the beginning of the input FASTQ file.

Our test program is included with the SeqDB source code, and was instrumented with the Performance Application Programming Interface (PAPI, version 4.1.3.0)[6] to measure elapsed runtime and performance counter data for L3 cache utilization. We placed the timers so as to exclude the cost of thread creation, and also wrote 32 MB of random data on each thread in between every trial to flush the L3 cache on both CPU sockets. All buffers were allocated on NUMA node 0 using the `numa_alloc_onnode()` call from the Linux `libnuma`

5. The `zlib` condition was so much slower that its runtime had little variability, and we only repeated it 4 times.

6. http://icl.cs.utk.edu/papi/

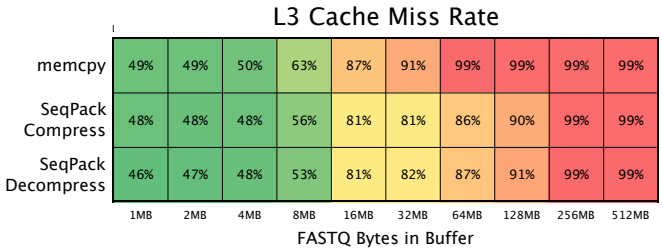

Fig. 2. For `memcpy` and SeqPack, cache misses transition from around 50% to 99% between the 4 MB and 32 MB buffer sizes, as the buffers become too large to fit in L3 cache.

library, to mitigate any variability from non-uniform memory access across the sockets.

Figure 1 shows how throughput varies with block size for both SeqPack and Blosc, and how both were able to achieve higher throughput than `memcpy` in many cases, thanks to their compression of the buffer. In contrast, `zlib` displayed a constant throughput across block sizes that never rose above 6.4 MB/s for compression or 166.8 MB/s for decompression.

Blosc attained a maximum throughput of 5.8 GB/s for decompression, but its highest compression throughput was only 1.8 GB/s. These results are comparable to those reported in synthetic benchmarks for Blosc for similar Intel processors.[7]

SeqPack's throughputs were higher than those of both Blosc and `memcpy` at all buffer sizes, reaching 8.6 GB/s for compression and 9.4 GB/s for decompression. The peaks at the 4 MB and 8 MB buffers seen for both SeqPack and `memcpy` correspond to an inflection point in the percentage of L3 cache misses (see Figure 2), so we suspect this is an artifact of the memory hierarchy. Each X5650 CPU has a shared L3 cache of 12 MB, so the 8 MB buffer is probably the last buffer size where both the read and write buffers can fit in the caches simaultaneously.

Finally, we also tested a SeqPack/Blosc condition in which we compressed the buffer with SeqPack followed by Blosc, then decompressed with Blosc followed by SeqPack. Not surprisingly, with the extra round of compression and decompression, this condition performed worse than either SeqPack or Blosc alone. Yet, it still outperformed `zlib` while yielding similar compression ratios, as we will report in more detail in the next section.

4.4 Compression Ratio and Throughput

Compression methods face a trade-off between throughput and compression ratio. Additional processing, and hence lower throughput, can improve compression ratio. Some methods, like Blosc and `zlib`, have a tunable parameter for controlling this trade-off, called the compression level.

7. http://blosc.pytables.org/trac/wiki/SyntheticBenchmarks

We tested SeqDB (version 0.2.0) with SeqPack/Blosc compression against four alternative methods for compressing NGS data sets. The most commonly used method is probably `gzip` (which uses the same compression algorithm as `zlib`). We ran `gzip` at compression level 6 since this is the default level if none is specified on the command line, and we suspect this is the most common usage. Next, we tested BioHDF (version 0.4a) with its default HDF5 `zlib` compression filter, and a modified version in which we replaced this with the Blosc filter. Details on how to register the Blosc filter with HDF5 are provided in the Blosc documentation, and the changes required modifying less than 10 lines of code in BioHDF. For all Blosc-based methods, we used a compression level of 4, which we found to offer the best compromise between throughput and compression ratio after conducting a parameter sweep on the ERR000018 data set. Finally, we tested DSRC (version 1.02), which is designed specifically for compressing FASTQ data.

For each compression methods, we loaded each of the data sets from Table 1 into the RAM disk, compressed it, and decompressed the output of the compression (to validate correctness, we also ran a `diff` on the original file and the output of the decompression). We measured the times to compress and decompress and divided by the original file size to calculate throughputs. We measured the size of the compressed output and divided by the original file size to calculate compression ratio.

Figure 3 shows the results of the comparison. Although SeqDB did not achieve the best compression ratio (DSRC wins on ratio alone), it does strike the best compromise between throughput and ratio. In compression, SeqDB was $25\times$ to $89\times$ faster than `gzip`, yet achieved within 69% to 85% of the compression ratio. Although DSRC achieved the best compression ratio on every data set, it did so at a cost of $2.4\times$ to $10.8\times$ longer runtimes than SeqDB.

In decompression, `gzip` performed noticeably better, and was slower than SeqDB by only $1.8\times$ to $7.0\times$. In contrast, DSRC was slowest in decompression, and SeqDB outperformed its throughput by $3.2\times$ to $14.2\times$. We would argue that the decompression comparison is the more important of the two, since an NGS data set will likely be compressed once when archived, but decompressed many times as it is reused in subsequent analyses.

BioHDF faired worst of all the methods, both in terms of compression ratio and throughput. The Blosc-modified BioHDF showed some improvement in throughput, but at the cost of poorer compression ratio.

SeqDB's throughput was lowest for the data sets with variable-length sequences. Its compression ratio faired well in these cases, though, considering it is backed by a fixed-length array. For the shorter sequences, the space wasted by the fixed-length represention was likely mitigated by Blosc's efficient compression of all the repeated null characters.

Overall, SeqDB delivered compression on par with

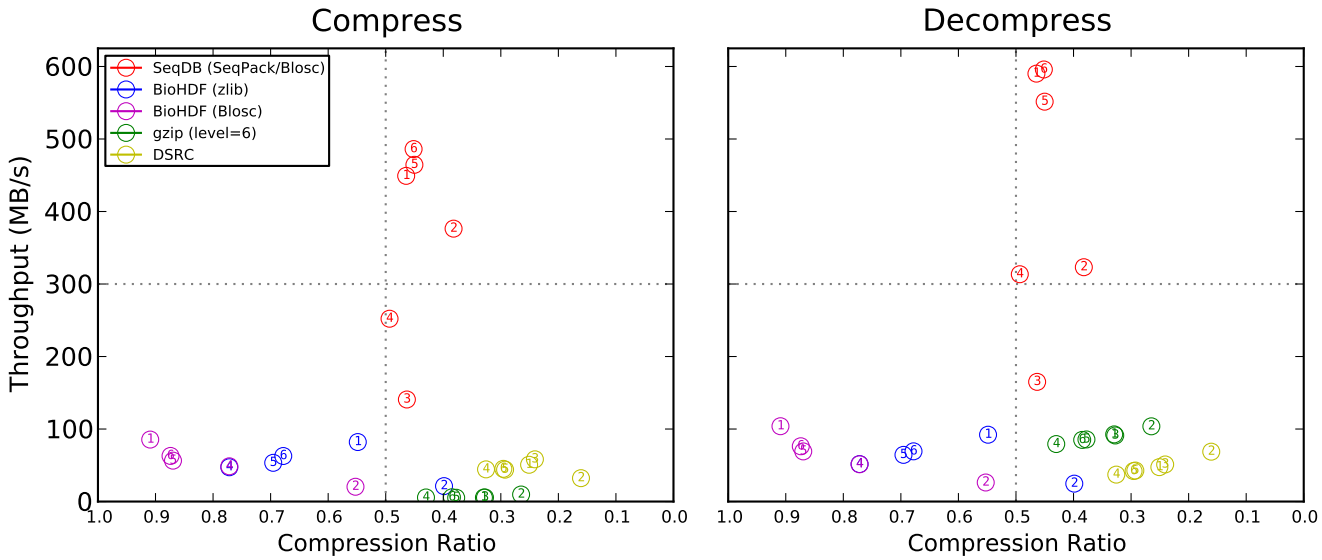

Fig. 3. Plotting compression ratio against throughput shows the trade-off inherent to compression methods: longer processing times (less throughput) often yield better ratios. SeqDB, however, negotiates this trade-off better than other methods, achieving higher throughputs relative to its compression ratio, and occupying the upper-right quadrant of the scatter plot. The methods in the lower-right quadrant favor ratio over throughput, and those in the lower-left quadrant are poor at both. Points are labeled by data set number (see Table 1). We were unable to compress the SRR003177 data set (#3) with BioHDF due to an error.

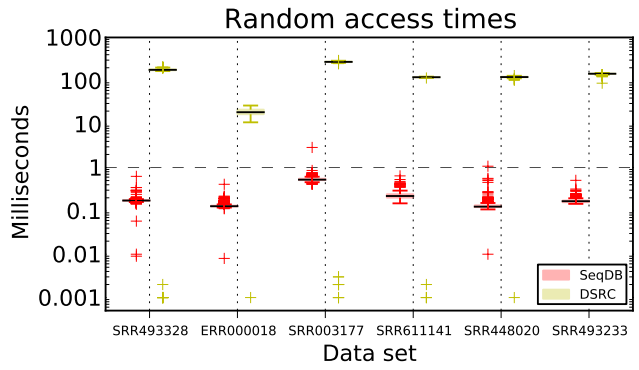

Fig. 4. Box-and-whisker plots show the distribution of latencies for reading 1000 individual records at random from each data set stored in SeqDB and DSRC formats. The median access time (shown as a black bar) for SeqDB is under 1ms for all data sets.

`gzip`, but at much higher throughputs. In addition, SeqDB is a structured, array representation of the sequence data, which enables fast random access (even in decompression, since the file is stored in blocks). BioHDF and DSRC provide this same benefit, but their throughputs were far below SeqDB's.

4.5 Random-Access Latency

One advantage of storing sequence data as a fixed-dimension array is that random access becomes an $O(1)$ operation (rather than $O(n)$ for variable-length storage). To measure this constant cost for random access, or the *latency*, we generated a list of 1000 record indices chosen from a uniform random distribution (without replacement), and loaded each record from both SeqDB and DSRC files for each data set. The test program is included in the SeqDB source code and we used the Linux `gettimeofday()` system call to measure elapsed time.

Although DSRC has exhibited random access times of less than 10ms [11], we were unable to reproduce this result, and the lowest latency we measured for DSRC was instead 11.0ms. This discrepancy could be caused by several factors, such as differences in the data sets tested, in the test hardware, or in the compilers and flags used.

Figure 4 shows the distribution of latencies across the 1000 trials for each data set. For all data sets but the largest (SRR493233), some of the random indices were close enough that the same block was re-read, leading to the outlier latencies as low as 1μs for DSRC. This did not occur for the largest data set, however, where the probability of a re-read is smaller since there are more indices, but the same block size. Across all data sets, the median access time for SeqDB was under 1ms and as low 0.13ms. For DSRC, the median access times ranged from 18.9–273.3ms.

Since both SeqDB and DSRC must read and decompress an entire block to access a single record, the latency correlates with the sequence length, with the worst

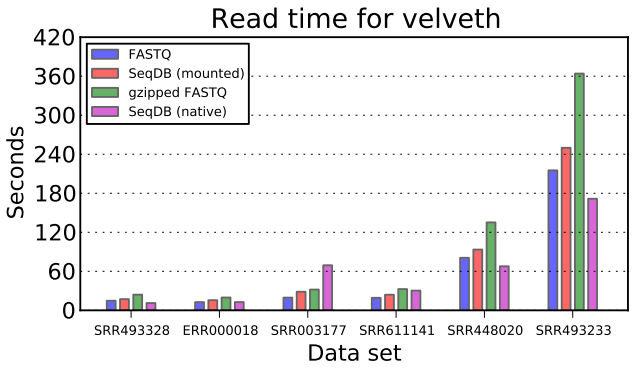

Fig. 5. Comparison of input methods for `velveth`, the read stage of the Velvet assembler. Reading a mounted SeqDB file introduces a 16% to 46% overhead versus reading FASTQ directly, while gzipped FASTQ introduces a 56% to 70% overhead. Mounted SeqDB is faster than gzipped FASTQ for all data sets. Native SeqDB reads were faster than uncompressed FASTQ for all data sets except SRR003177.

latencies occuring for the LS 454 data set (SRR003177).

4.6 Overhead of FASTQ Compatibility

One impediment to creating a new storage model for sequence data is that it won't initially be supported by the large ecosystem of bioinformatics tools that already exist. SeqDB addresses this by presenting a virtual representation of its contents in FASTQ format through a named pipe, as described in Section 3.3. To validate this compatibility mechanism with a real-world application, and to measure its overhead, we ran the first stage of the Velvet *de novo* genome assembler [12] on FASTQ, mounted SeqDB, and gzipped FASTQ versions of each of the test data sets. We also modified the Velvet source code to add support for reading SeqDB files natively by calling the SeqDB API.

The first stage of Velvet, a program called `velveth`, constructs the full de Bruijn graph used in the assembly. To do this, it reads in all of the input sequences (and reports the total read time), hashes them, and stores them to an internal graph representation.

Figure 5 shows the measured read times for each input format and data set. SeqDB's compatibility mode introduces a 16% to 24% overhead for the Illumina and Ion Torrent data sets, and a 49% overhead for the LS 454 data set. When compared to directly reading a FASTQ version of the same data set, SeqDB is faster than `gzip` for all data sets, by as much as 46% for the largest data set, SRR493233. Natively reading the SeqDB file is even faster than reading the uncompressed FASTQ file for all data sets but SRR003177.

5 Conclusion

SeqDB provides an efficient storage model for the raw data produced by NGS platforms like the Illumina HiSeq 2000 and MiSeq and Life Technologies Ion Torrent PGM. Through its backward compatibility mode, it can be integrated with existing bioinformatics tools and pipelines that expect FASTQ input. In performance comparisons against other compression methods, SeqDB is the clear winner in terms of throughput, and it offers favorable compression ratios as well. Thus, it is an ideal storage model for archiving raw NGS data.

SeqDB could, however, achieve better compression ratios by extracting only the information in the header that changes between reads. For example, the Illumina CASAVA 1.8 headers for a given lane only vary by flow cell coordinates, forward/reverse orientation, and mask. Unfortunately, in initial testing with SeqPack, we found that the cost of parsing the header on packing, and formatting the header on unpacking, led to considerable deterioration in throughput. Also, supporting this kind of extraction would require writing a different extraction method for each NGS platforms' header format. However, storing the extracted information instead of a header string may benefit analysis tools that make use of the flow cell coordinates, since those tools would already need to perform the extraction.

Although it isn't designed for variable-length sequences, we showed that SeqDB can compress data sets with small variations in sequence length reasonably well. This scenario is common with Illumina HiSeq data, where there is a bias toward low quality at the end of the reads, and the last 1 to 10 base pairs (1% to 10% of the sequence) may be trimmed during a quality control phase. The variable-length sequences we tested (from LS 454 and Ion Torrent PGM data sets) had even more extreme ranges than this, so we expect that quality-trimmed HiSeq data would compress well with SeqDB. However, we leave the testing of quality-trimmed data to future work, since we plan to incorporate SeqDB into analysis pipelines that conduct such filtering on HiSeq data, in particular through the BioLite [13] framework that we are actively developing. For large variations in length, such as in the contig output of a *de novo* assembly, SeqDB is clearly a poor choice, but such data sets typically lack quality scores, are much smaller than sequence read data sets, and lie outside the scope of SeqDB.

Finally, SeqDB's modular storage backend is designed to extend to I/O libraries other than HDF5. We plan to investigate how an indexed database, such as SQLite, could enable efficient ID querying in SeqDB, and whether a database backend could provide the same high-throughput for sequential access as HDF5 provides.

Acknowledgments

The author would like to thank Quincey Koziol and Dana Robinson of the HDF Group and Fransesc Alted for their feedback. This research was conducted using computational resources and services at the Center for Computation and Visualization, Brown University.

REFERENCES

[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.

[2] F. Alted, "BLOSC," 2009. [Online]. Available: http://blosc.pytables.org/

[3] F. Alted, I. Vilata *et al.*, "PyTables: Hierarchical Datasets in Python," 2002. [Online]. Available: http://www.pytables.org/

[4] F. Alted, "Why modern CPUs are starving and what can be done about it," *Computing in Science & Engineering*, vol. 12, no. 2, pp. 68–71, 2010.

[5] A. Hidayat, "FastLZ - lightning-fast compression library," 2007. [Online]. Available: http://fastlz.org

[6] The HDF Group, "Hierarchical Data Format version 5," 2000. [Online]. Available: http://www.hdfgroup.org/HDF5/

[7] C. E. Mason, P. Zumbo, S. Sanders, M. Folk, D. Robinson, R. Aydt, M. Gollery, M. Welsh, N. E. Olson, and T. M. Smith, "Standardizing the Next Generation of Bioinformatics Software Development with BioHDF (HDF5)," in *Advances in Computational Biology*, H. R. Arabnia, Ed. Springer New York, 2010, vol. 680, pp. 693–700.

[8] R. Leinonen, H. Sugawara, and M. Shumway, "The Sequence Read Archive," *Nucleic Acids Research*, vol. 39, pp. D19–D21, Jan. 2011.

[9] R. Wan and K. Asai, "Sorting next generation sequencing data improves compression effectiveness," in *Proceedings of the 2010 IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW)*, 2010, pp. 567–572.

[10] W. Tembe, J. Lowey, and E. Suh, "G-SQZ: compact encoding of genomic sequence and quality data," *Bioinformatics*, vol. 26, no. 17, pp. 2192 –2194, 2010.

[11] S. Deorowicz and S. Grabowski, "Compression of DNA sequence reads in FASTQ format," *Bioinformatics*, vol. 27, no. 6, pp. 860–862, Mar. 2011.

[12] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[13] M. Howison, N. A. Sinnott-Armstrong, and C. W. Dunn, "BioLite, a lightweight bioinformatics framework with automated tracking of diagnostics and provenance," in *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP '12)*, Jun. 2012.

Mark Howison received the master's degree in Computer Science from the University of California, Berkeley in 2009. He has worked as a Computer Systems Engineer for Lawrence Berkeley National Laboratory's Visualization Group, and currently as an Application Scientist at Brown University's Center for Computation and Visualization. His research interests include computational biology, scientific and high-performance computing, visualization, performance tuning and parallel I/O.