bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

Under submission
DOI
Advance Access Publication Date: Day Month Year
Manuscript Category

Sequence analysis

NanoSpring: reference-free lossless compression of nanopore sequencing reads using an approximate assembly approach

Qingxi Meng [†,*], Shubham Chandak [†,*], Yifan Zhu [*] and Tsachy Weissman

Department of Electrical Engineering, Stanford University, Stanford, CA 94305, USA

[†] To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: The amount of data produced by genome sequencing experiments has been growing rapidly over the past several years, making compression important for efficient storage, transfer and analysis of the data. In recent years, nanopore sequencing technologies have seen increasing adoption since they are portable, real-time and provide long reads. However, there has been limited progress on compression of nanopore sequencing reads obtained in FASTQ files. Previous work ENANO focuses mostly on quality score compression and does not achieve significant gains for the compression of read sequences over general-purpose compressors. RENANO achieves significantly better compression for read sequences but is limited to aligned data with a reference available.
Results: We present NanoSpring, a reference-free compressor for nanopore sequencing reads, relying on an approximate assembly approach. We evaluate NanoSpring on a variety of datasets including bacterial, metagenomic, plant, animal, and human whole genome data. For recently basecalled high quality nanopore datasets, NanoSpring achieves close to 3x improvement in compression over state-of-the-art reference-free compressors. The computational requirements of NanoSpring are practical, although it uses more time and memory during compression than previous tools to achieve the compression gains.
Availability: NanoSpring is available on GitHub at `https://github.com/qm2/NanoSpring`.
Contact: qingxi@stanford.edu, schandak@stanford.edu
Supplementary information: Supplementary data are available on BioRxiv.

1 Introduction

The rapid decrease in the cost of genome sequencing has led to an explosion in the amount of data produced by these experiments, with the raw data usually requiring the most space for storage. The raw sequencing data is obtained in the form of reads with sequencing depth/coverage often being 30x or higher. A typical human whole genome sequencing experiment can produce 100s of GBs of data in FASTQ files. Given the high sequencing depth, there is much redundancy to be exploited in the reads, and several specialized compressors like SPRING (Chandak *et al.*, 2019) and PgRC (Kowalski and Grabowski, 2019) have been developed for this data. The typical approach used by these compressors is to efficiently build an approximate assembly using the reads and then store this assembly along with the encoding of the reads with respect to the assembly.

While the existing compressors have been mostly built for short-read sequencers such as Illumina, in recent years, nanopore sequencing, specifically using Oxford Nanopore Technologies (ONT) sequencers (Jain *et al.*, 2018), has seen increasing adoption since it is portable, real-time and provides long reads. However, there has been limited progress on compression of nanopore sequencing reads. Most existing works like SPRING and PgRC operate under the assumption that the reads are short ($\sim$100s of bases) and low-error (with most errors being substitutions). On the other hand, nanopore reads are much longer (often over hundreds of thousands of bases long), and have a much higher error rate, including substitution, insertion, and deletion errors from the basecalling process that converts the raw current signal to the read sequences (Wick *et al.*, 2019). However, the error rate has fallen dramatically in the recent years

[*] These authors contributed equally to the work.

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

with the advent of deep learning based basecallers which achieve median error rate close to 5% or better (Chandak *et al.*, 2020), suggesting that a similar approximate assembly approach with some adaptations can be applied to nanopore sequencing reads.

There have been a couple of works on compression of nanopore FASTQ data. ENANO (Dufort y Álvarez *et al.*, 2020) focuses mostly on quality score compression, and uses a context-based model followed by arithmetic coding for read sequences. Note that while quality scores occupy a significant amount of space even after compression, we focus on read sequences due to the lack of research in this area, and since quality scores are often ignored by downstream tools like minimap2 (Li, 2018) and have been compressed lossily for previous technologies without an impact on the downstream performance (Yu *et al.*, 2015; Ochoa *et al.*, 2016). RENANO (Dufort y Álvarez *et al.*, 2021) is a recent reference-based compressor that achieves significantly better compression for read sequences, but is limited to aligned data with a reference available.

In this work, we present NanoSpring, which is a lossless reference-free compressor for nanopore sequencing reads. NanoSpring uses an approximate assembly approach partly inspired by existing assembly algorithms but adapted for significantly better performance, especially for the recent higher quality datasets. On recent human whole genome datasets, NanoSpring achieves close to 3x improvement in compression as compared to ENANO. NanoSpring is competitive in terms of decompression time but requires higher compression time and memory to obtain the benefits in compression ratio. NanoSpring is available as an open-source tool on GitHub, requires only a FASTQ file as input for compression, and does not compress read identifiers or quality values.

2 Methods

In this section, we describe the NanoSpring algorithm which is a lossless compressor for nanopore read sequences and does not require an external reference for compression. NanoSpring relies on an approximate-assembly approach, where we first assemble the reads into contigs, obtain the consensus sequence for each contig, and finally store the consensus sequence and encode the reads with respect to the consensus sequence. Parts of the algorithm were inspired by the MinHash-based assembler presented in Berlin *et al.* (2015), with suitable adjustments to the parameters to achieve orders of magnitude speedup over the assembler while still obtaining a sufficiently accurate assembly for compression purposes.

We first give a high-level overview of the algorithm before describing the various stages and parameters in more detail. NanoSpring first loads the reads into memory using an efficient 2 bits/base representation (ignoring read identifiers and quality values in the FASTQ file). Next, NanoSpring indexes the reads using MinHash which enables efficient lookup of reads overlapping a given sequence, effectively handling substitution, insertion, and deletion errors. Once the index is constructed, NanoSpring attempts to build contigs consisting of overlapping reads. The contigs are represented using consensus graphs with each read corresponding to a path on the graph. The contig is built by greedily searching the MinHash index for reads that overlap with the current consensus sequence of the graph, and adding the candidate reads to the graph using minimap2 (Li, 2018) alignment. Finally, the consensus sequence and the errors in the reads with respect to the consensus sequence are written to separate streams and compressed using general-purpose compressors.

The decompression process is quite simple: the decompressor first obtains the consensus sequence and error streams using the general-purpose decompressor. Then it applies the errors to the appropriate parts of the consensus sequence to obtain the reads.

2.1 MinHash indexing

We use MinHash (Broder, 1997) for indexing the reads allowing for efficient lookup of reads overlapping with any given sequence. During the construction of the index, we first extract substrings of length k (k-mers) from a read and compute n pseudorandom hash functions of the k-mers (the MinHash *sketch* of the read). For each hash function, we find the k-mer with the minimum hash value, referred to as the MinHash of the read. The basic theoretical property underlying MinHash is that the fraction of shared MinHash values (out of n) between two sequences is a good estimator for the fraction of shared k-mers between the sequences, and the estimator accuracy increases with increasing n. Since we can expect overlapping sequences, potentially with substitution/insertion/deletion errors, to have common k-mers (for sufficiently small k), MinHash provides us a way to efficiently estimate the similarity of sequences and to rapidly look up overlapping reads (as described next).

The MinHash index consists of n tables, one for each pseudorandom hash function. Each table maps MinHash values for the corresponding hash function to the list of reads with that MinHash value. During lookup, we are given a sequence and first compute the n MinHash values for that sequence (the MinHash sketch). Then we use the index to find and return reads matching the sketch for at least t out of n hash functions, where t is a threshold parameter. In our implementation, the k-mers (with $k \leq 32$) and hash values are represented as 64-bit integers, and the pseudorandom hash functions are simply computed as $hash(kmer \oplus r_i)$ where $hash$ is a standard hash function, and r_i for $i = 1, \ldots, n$ are pseudorandom integers. To minimize the memory usage, the hash tables are built using BBHash (Limasset *et al.*, 2017) which is a specialized data structure designed for hash tables that are not modified after construction, which applies in our case.

There are three key parameters for MinHash indexing, the k-mer length (k), the number of hash functions (n) and the threshold for lookup (t). Increasing n improves the accuracy of MinHash and allows for more reliable estimation of sequence similarity but leads to increased computational overhead. The ratio t/n determines the threshold for similarity of reads during lookup. Higher values of this ratio can lead to missed potentially matching reads, whereas smaller values can increase the number of false positives and hence the computational overhead. In addition, very small values can lead to low quality matches which can adversely affect the compression. Finally, the value of k should be chosen depending on the genome size, the error rate, and the computational requirements. At higher values of k, the k-mer is more likely to have an error and finding exact matches becomes unlikely. If k is too small, then we can get many spurious matches for large genomes leading to poor computational performance. By default, we set $k = 23, n = 60, t = 6$ (see Section 3.1.3 for a detailed analysis). We note that the parameters chosen in this work are different from the parameters used in the assembler from Berlin *et al.* (2015). Specifically, we tuned the parameters for better performance since we only require an approximate assembly in our application. We also take into account the fact that the nanopore error rates have reduced significantly over the past years making the use of these parameters reasonable.

2.2 Construction of contigs

NanoSpring uses the following pseudocode for constructing contigs of overlapping reads:

1. Initialization: Pick an arbitrary read not yet added to a contig, and construct the consensus graph with a single read.
2. Repeat the following until no matching reads are found:

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

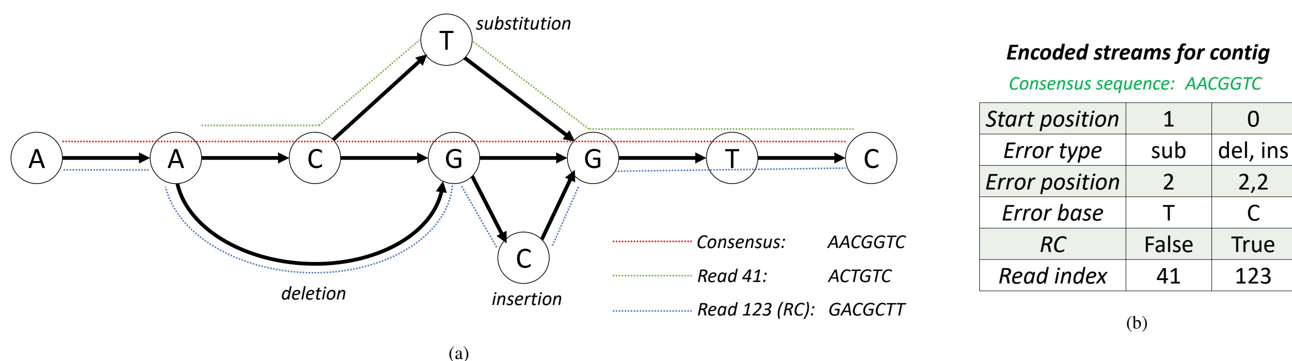

Fig. 1: (a) Consensus graph for a contig showing the consensus sequence (path shown in red) and two reads. The read with the path shown in green has index 41 in the FASTQ file, and has a substitution compared to the consensus sequence. The read with the path shown in blue has index 123 in the FASTQ file, is reverse complemented with respect to the consensus sequence, and has a deletion and an insertion. (b) The encoding of the contig into multiple streams. Note that the error position is 0-indexed and delta coded, and the error base needs to be stored only for insertions and substitutions.

a. Obtain substring of the current consensus sequence (explained below).
b. Find overlapping reads to the substring using MinHash index.
c. For each potential read from previous step (that has not yet been added to a contig):

(1) Align read to current consensus sequence using minimap2.
(2) If alignment succeeds, add read to the consensus graph and recalculate the consensus sequence (explained below).

As described in the pseudocode, the contig construction maintains a consensus graph (also referred to as an assembly graph) and greedily adds reads to the graph. The graph is directed and acyclic with the nodes representing the bases and the edges storing the information about the reads passing through that edge. The *weight* of an edge is the number of reads passing through the edge. Initially the graph is just a line graph consisting of a single read. As reads get added to the graph, these reads lead to branching out from the line graph due to presence of errors. The consensus sequence represents the path with the highest weight (where the weight of a path is the sum of weights of the edges on the path). Figure 1a illustrates the consensus graph with the reads and the consensus sequence, and Figure 2 shows the overall contig generation procedure. In our implementation, we use a greedy algorithm for computing the consensus path: starting at the leftmost node and picking the highest weight edge at every step. We use the greedy algorithm instead of the optimal dynamic programming-based algorithm due to its simplicity and similar performance in practice.

At every step in the contig generation algorithm, we first pick a substring of the consensus sequence, which is used to search for matching reads using the MinHash index. The substring has length equal to the average read length of the dataset or the length of the consensus sequence, whichever is smaller. After each iteration, we obtain a shifted substring by changing the start position by a quarter of the average read length (shift length chosen based on experiments). We first shift the substring to the right until we reach the end, and then shift it left till we reach the beginning. This allows us to capture reads overlapping with any section of the consensus sequence. Note that the consensus sequence itself is constantly updated, and hence we maintain the position of the first read on the consensus sequence as a reference zero coordinate for tracking the substring location. We work with the consensus sequence at the current iteration (instead of an individual read) for MinHash lookup and alignment since we expect the consensus sequence to have a lower error rate leading to a more efficient and accurate process. To handle reverse complemented reads, we search for overlapping reads to both the substring and its reverse

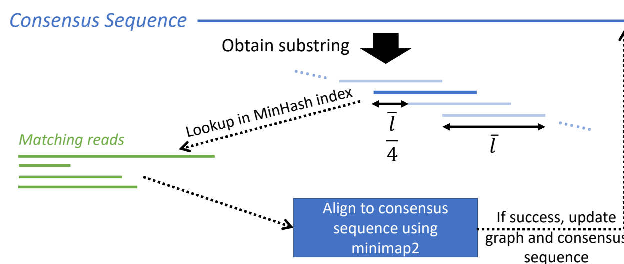

Fig. 2: Contig generation process. A substring of the current consensus sequence is used to find matching reads from the MinHash index. Each potential match is aligned to the consensus sequence using minimap2 and added to the graph if the alignment succeeds. The substring has length $\bar{l}$ (average read length) and is shifted along the consensus sequence at each step to capture all potential matches.

complement in step 2b and store a flag denoting reverse complemented reads.

For each potentially matching read obtained using MinHash, we attempt to align it to the consensus sequence using the minimap2 aligner (Li, 2018). We found that previous works on assembly like Berlin *et al.* (2015) used their own implementation of the optimal Myers aligner (Myers, 1986), but we found that the widely used minimap2 aligner was simpler to use and significantly faster. The minimap2 aligner first indexes the reference sequence (the consensus sequence in our case) based on minimizers (lexicographically smallest k-mers) of length k in each window of length w. Then it attempts to locate these minimizers in the query string (the reads returned by MinHash), followed by more accurate alignment in the regions between these minimizers. When alignment succeeds, minimap2 returns the CIGAR string consisting of the errors in the query string with respect to the reference (we restrict ourselves to the top-scoring alignment returned by minimap2). We use this information to add the read to the graph, with soft clips treated as a sequence of insertions at the beginning or end of the read. We modified the default parameters for minimap2 in order to improve the computational performance, setting $k = 20, w = 50$ and reducing the `max-chain-iter` parameter controlling the complexity of the chaining step in minimap2 to 400 from the default value of 5000. See Section 3.1.4 for more discussion on the impact of these parameters.

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

2.3 Encoding and compression of streams

The reads in each contig are encoded into multiple streams after the contig generation is done. We store the consensus sequence of the contig and then store the representation of each read with respect to the consensus. Specifically, we store the start position of the read on the consensus sequence, the errors in the read, the reverse complement flag, and the read index representing the index of the read in the original FASTQ file. The errors are themselves represented using three streams: (i) the error type (insertion/deletion/substitution), (ii) the error position on the read (delta coded), and (iii) the erroneous base (for insertions and substitutions). These streams are illustrated in Figure 1b. We observed that in many cases, we get multiple insertions at the beginning and end of the reads possibly due to adapter sequences (soft clipped in the alignment). For such cases, we simply encode the number of insertions and the inserted bases rather than encoding each insertion separately. The current implementation combines the start position and error position streams since they share the same datatype and the start position stream is typically a negligible contributor to the total size.

In our experiments, we found that the contig generation led to several contigs with only a single read due to the greedy procedure and the variation in read quality (with high error reads missed by MinHash). For such contigs, we directly write the read sequence to a separate stream for these *lone* reads.

Finally, these streams are compressed with general-purpose compressors and combined into a single file using the tar utility on UNIX. We use two general-purpose compressors which provide improvements over Gzip while being computationally efficient. For compressing the stream with the erroneous bases, we use LZMA2 (`https://github.com/conor42/fast-lzma2`) which relies on Lempel-Ziv compression (Ziv and Lempel, 1977) and arithmetic coding (Witten *et al.*, 1987). For the remaining streams we use BSC (`https://github.com/IlyaGrebnov/libbsc`) which relies on the Burrows-Wheeler transform (BWT, Burrows and Wheeler (1994)) and arithmetic coding. As discussed in Section 3.1.5, we found this combination of compressors led to the best compression ratios.

2.4 Additional implementation details

We made some modifications to the procedure presented above to improve the performance on real datasets.

- The MinHash indexing, contig generation, stream compression, and the decompression are parallelized to improve the wall-clock performance. During contig generation, different threads work on different contigs, and we ensure that there are no conflicts using locks. The impact of multithreading is discussed in Section 3.1.6.
- We found that for certain human datasets, minimap2 took a very long time for aligning highly repetitive sequences, usually with tandem repeats (such as $GTGTGT\ldots$). Therefore, we check the reads for short tandem repeats before the contig generation stage and repetitive reads are directly written to the lone read stream. We also write reads with length ≤ 32 directly to the lone read stream since they have too few k-mers to obtain matches using MinHash.
- To limit the memory usage of the consensus graphs during contig generation, especially when working with multiple threads, we impose a limit on the number of edges in the graph. Beyond this limit, the contig generation is stopped and we proceed with a new contig. We found that this simple strategy drastically reduces the peak memory consumption while having minimal impact on the compression ratio (see Section 3.1.7). We also use other low-level optimizations such as periodically calling `malloc_trim()` to further reduce the memory usage.

3 Results and discussion

We tested Nanospring on several real datasets that cover a variety of organisms with different genome lengths sequenced at varying depths of coverage. We compare NanoSpring to the current state-of-the-art reference-free compressor for nanopore FASTQ files, ENANO (Dufort y Álvarez *et al.*, 2020) and to pigz (`https://zlib.net/pigz/`) which is a multithreaded version of the general-purpose compressor Gzip. While ENANO compresses the entire FASTQ file including the read sequences, quality values and read identifiers, we only focus on the compressed size for the read sequences. We note that ENANO supports multiple compression levels, and we use the default one since it is significantly faster than the maximum compression level with minimal difference in read sequence compression ($\lesssim 1\%$). All experiments were run on an Ubuntu 18.04.5 server with 40 Intel Xeon processors (2.2 GHz) and 260 GB RAM. The tools were run with 20 threads unless specified otherwise. Details on installing and running the various tools are provided in Supplementary data.

Datasets

The datasets used for experiments are listed in Table 1. These include bacterial, metagenomic, animal, plant, and human datasets. Further details on obtaining these datasets are provided in the Supplementary data. We included some standard datasets including the NA12878 dataset (*hs1*) and the Zymo microbial standard on R10.3 pore (*zymo*), but we largely focused on datasets basecalled with more recent tools that provided much higher quality and better scope for compression. To further test the impact of the basecaller on the compression rate, we basecalled the *S. aureus* dataset using three modes and compare the results in Section 3.1.8. We also looked into the impact of coverage on the compression ratio for the *M. acuminata* (*banana*) and CHM13 datasets (*hs3*, *hs4*) which were available at a high initial coverage. The complete results for the different subsampled versions are shown in Section 3.1.2.

Compression results

Table 2 shows the compression results for Gzip, ENANO and NanoSpring on the datasets. We observe that Gzip and ENANO perform consistently across the datasets, achieving around 2.2 and 1.9 bits/base, respectively (with $\sim$2 bits/base being achievable with a fixed-length encoding). NanoSpring provides much better compression, getting below 0.7 bits/base for the human datasets (*hs2*, *hs3*, *hs4*), which is around 3x better than Gzip and ENANO. In absolute terms, NanoSpring compresses the 84 GB *hs2* dataset to less than 7 GB. For most other datasets, NanoSpring achieves close to 2x improvement over the other tools. Note that the compression results for the *hs1* dataset are significantly worse, although NanoSpring still outperforms Gzip and ENANO. This can be explained by the fact that this dataset was obtained using an older basecaller with appreciably higher error rates. Given the steady improvement in basecaller quality over the years (Wick *et al.*, 2019), we can expect the performance of NanoSpring to improve further in the future (cf. Section 3.1.8).

Time and memory usage

Table 3 shows the time and peak memory usage for the compression and decompression using the three tools, all running on 20 threads. We note that ENANO does not provide a mode for compressing only the read sequences, and so the time and memory usage numbers include the compression of quality scores and read identifiers. Despite this, we see that ENANO significantly outperforms NanoSpring in terms of compression time/memory and decompression memory. Gzip also uses much less time and memory than NanoSpring. We note that NanoSpring is quite competitive in decompression speed, requiring less than 10 minutes for a 26x human dataset (*hs2*).

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

Dataset Name	Species	Sample	Genome Size (Mbp)	Coverage	Number of Reads (M)	Average Read Length (Kb)	N50 (Kb)	Uncompressed Size (GB)	Source
sa	*S. aureus*	CAS28_02	2.9	84x	0.01	21.99	24.8	0.24	Wick *et al.* (2019)
zymo	*Metagenomic*	Zymo (R10.3)	-	-	1.16	3.99	25.2	4.6	Nicholls *et al.* (2019)
snail	*C. squamiferum*	SAMN10963494	356	139x	7.45	6.65	7.3	49.5	Sun *et al.* (2021)
banana	*M. acuminata*	SAMEA6104609	523	177x	5.19	17.9	31.6	92.7	Belser *et al.* (2021)
hs1	*H. sapiens*	NA12878	3,200	42x	15.7	8.48	13.6	132.9	Jain *et al.* (2018)
hs2	*H. sapiens*	GM24385	3,200	26x	3.44	24.5	46.5	84.2	See caption
hs3	*H. sapiens*	CHM13	3,200	23x	5.90	12.6	58.7	74.2	Nurk *et al.* (2021)
hs4	*H. sapiens*	CHM13	3,200	46x	11.8	12.6	58.8	148.4	Nurk *et al.* (2021)

Table 1. Datasets used for experiments. The uncompressed size refers to the file size obtained by removing the quality scores and sequence identifiers from the FASTQ files. N50 is a robust measure for read lengths, with the reads with length above the N50 metric capturing 50% of the data. The $hs2$ dataset was obtained from `https://labs.epi2me.io/gm24385_2020.09/` provided as part of ONT Open Datasets.

Dataset Name	Coverage	Uncompressed size (GB)	Compressed size in bits/base			Improvement over ENANO
			Gzip	ENANO	NanoSpring	
sa	84x	0.24	2.29	1.89	0.50	3.78x
zymo	-	4.6	2.32	1.96	0.84	2.33x
snail	139x	49.5	2.14	1.80	1.05	1.71x
banana	177x	92.7	2.28	1.93	1.06	1.82x
hs1	42x	132.9	2.24	1.89	1.45	1.30x
hs2	26x	84.2	2.20	1.87	0.66	2.83x
hs3	23x	74.2	2.18	1.86	0.68	2.74x
hs4	46x	148.4	2.18	1.86	0.60	3.10x

Table 2. Compression results for read sequences using Gzip, ENANO and NanoSpring.

The high resource usage for NanoSpring during compression is due to the approximate assembly process which provides the gains in compression, and is on a similar scale as previous works following this approach for short reads (Chandak *et al.*, 2019; Kowalski and Grabowski, 2019). The memory usage consists of the reads, the MinHash index and the consensus graph. We typically observed that the time and memory usage scaled linearly with the dataset size, although there is some variability in the contig generation stage. For the 26x human dataset *hs2*, NanoSpring requires ∼3 hours (∼70 CPU hours) and 39 GB memory, which is an order of magnitude smaller than the requirements for genome assembly. For example, wtdbg2 (Ruan and Li, 2020), a recent efficient assembler requires over 1000 CPU hours and 200 GB memory for nanopore human datasets with ∼35x coverage. Finally, the high decompression memory usage for NanoSpring is an artifact of the current implementation that decodes the reads out of order and loads them in memory before writing them to disk in the correct order. We believe that this can be reduced by modifying the encoded streams to enable a streaming decompression process, and we plan to implement this as part of future work.

3.1 Parameter analysis and discussion

To provide further insight into the NanoSpring algorithm, we discuss various parameters, experiments and related observations. While testing the individual parameters, we keep all other parameters constant. We note that the time measurements show some experimental variation due to I/O and multithreading. Therefore, we focus on the overall trend rather than these minor variations. More details on these experiments are available in the Supplementary data and on GitHub.

3.1.1 Contribution of streams to compressed size

Figure 3 shows the contribution of the various streams to the total compressed size for the datasets from Table 1. We focus on the consensus sequence, the error streams (position, type, erroneous base) and the lone reads (i.e., reads for which no matches were found). The remaining streams contribute less than 1% to the total size and are omitted here for clarity. We first note that the error streams take up close to 0.5 bits/base for most datasets, while the contribution of the consensus sequence is smaller. The contribution of the lone read stream varies a lot between datasets and is quite high for the *snail*, *banana* and *hs1* datasets where NanoSpring

Dataset Name	Compression time			Peak compression memory (GB)			Decompression time			Peak decompression memory (GB)		
	Gzip	ENANO*	NanoSpring	Gzip	ENANO*	NanoSpring	Gzip	ENANO*	NanoSpring	Gzip	ENANO*	NanoSpring
sa	2.2s	5.9s	28.4s	0.015	0.39	7.09	0.9s	7.4s	1.3s	0.0026	0.43	0.37
zymo	38.5s	38.5s	12m23s	0.015	0.43	16.5	16.3s	58.9s	24.1s	0.0026	0.55	3.00
snail	6m48s	5m25s	2h12m	0.015	0.43	33.7	3m15s	10m04s	5m33s	0.0026	0.54	26.7
banana	14m29s	18m27s	3h19m	0.015	0.43	43.7	6m52s	22m54s	11m16s	0.0025	0.54	25.5
hs1	19m17s	26m06s	5h07m	0.015	0.46	60.3	10m56s	36m10s	22m15s	0.0025	0.56	36.6
hs2	12m	10m45s	3h19m	0.015	0.43	39	7m05s	22m25s	6m57s	0.0026	0.55	22.4
hs3	10m08s	7m24s	3h32m	0.015	0.37	39	5m50s	13m55s	6m45s	0.0025	0.49	20.2
hs4	20m06s	25m21s	7h53m	0.015	0.37	97.7	12m48s	29m57s	15m03s	0.0025	0.49	39.1

Table 3. Time and peak memory requirements for compression and decompression. *ENANO figures include the time/memory usage to compress/decompress the entire FASTQ file including read sequences, quality values and read identifiers, while Gzip and NanoSpring only compressed the read sequences.

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

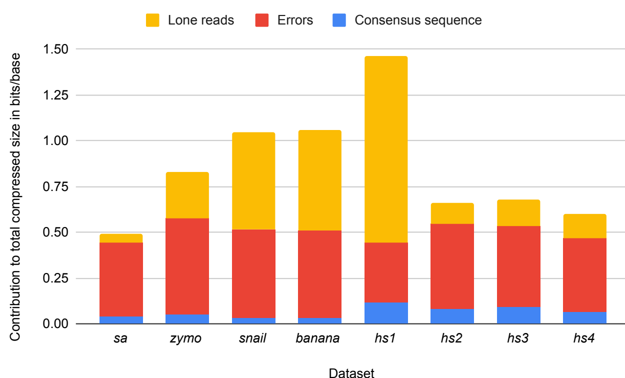

Fig. 3: Contribution of consensus sequence, errors and lone reads to the compressed size across datasets.

has relatively worse compression performance. We believe that this is associated with the data quality since we see a drastic reduction in the lone reads for datasets basecalled with the latest basecallers (*sa*, *hs2*, *hs3* and *hs4*). Finally, we note that datasets with higher coverage have a smaller contribution from the consensus sequence (e.g., compare *hs3* and *hs4*). This is expected theoretically since the genome size is a fixed constant while the error streams grow linearly as we get more reads.

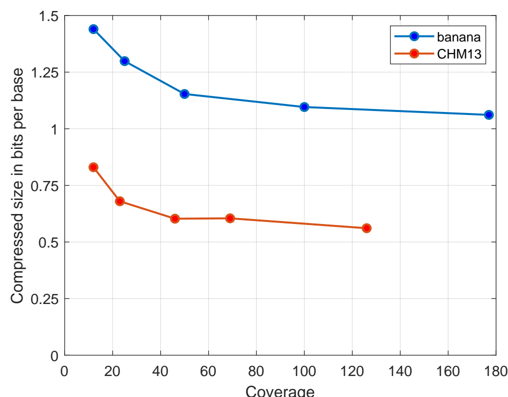

Fig. 4: Compressed size (bits/base) vs. coverage for subsampled *banana* and CHM13 datasets.

3.1.2 Coverage

To understand the impact of sequencing coverage on the performance of NanoSpring, we tested it on two datasets subsampled to multiple coverage values. We used the *banana* dataset with 177x coverage, and the CHM13 dataset (used to obtain *hs3*, *hs4* in Table 1) with 126x coverage. The compressed sizes are shown in Figure 4, where we see that the compression improves with coverage for both datasets even though the two datasets have significantly different compression levels (likely due to the differing basecalling qualities). This is expected since higher coverage datasets have more redundancy. For the CHM13 dataset, we get compressed sizes of 0.83, 0.68, 0.60, 0.60 and 0.56 bits/base for coverage values of 12x, 23x, 46x, 69x and 126x, respectively, with diminishing returns as the coverage increases (note that slight deviations from the trend can be explained by the random subsampling process and experimental variation). We observe that even at the low coverage of 12x, NanoSpring achieves more than two times better compression ratio than ENANO or Gzip for the high quality CHM13 dataset.

k	Compressed size		Compression time		Peak memory usage	
	hs2	*zymo*	*hs2*	*zymo*	*hs2*	*zymo*
10	-	0.782	-	33m03s	-	16.4
12	0.827	0.780	4h06m	23m51s	39.2	16.1
15	0.674	0.794	3h16m	13m33s	38.4	16.3
18	0.654	0.808	2h54m	13m08s	39.0	17.0
20	0.663	0.816	2h46m	12m54s	39.0	17.6
23	0.656	0.830	2h44m	12m02s	39.1	17.5
25	0.660	0.842	2h31m	11m36s	38.9	15.6
30	0.685	0.876	2h18m	10m58s	38.6	15.8

Table 4. Performance of NanoSpring with different MinHash k-mer sizes for the $hs2$ and $zymo$ datasets. Compressed size is in bits/base and the peak memory usage is in GB. The $hs2$ dataset was tested with $k = 10$.

n	t	t/n	Compressed size		Compression time		Peak memory usage	
			hs2	*zymo*	*hs2*	*zymo*	*hs2*	*zymo*
50	4	0.08	0.680	0.821	3h05m	14m19s	38.3	16.7
50	5	0.10	0.664	0.835	2h52m	12m21s	38.2	15.9
60	6	0.10	0.669	0.832	2h50m	12m34s	38.8	15.3
70	7	0.10	0.662	0.830	2h47m	11m52s	39.2	15.9
50	6	0.12	0.688	0.847	2h32m	10m59s	38.3	16.2

Table 5. Performance of NanoSpring with varying MinHash t and n parameters for the $hs2$ and $zymo$ datasets. Compressed size is in bits/base and the peak memory usage is in GB.

3.1.3 MinHash parameters

NanoSpring uses MinHash to index the reads and find overlapping reads during contig generation, with parameters k (MinHash k-mer length), n (number of hash functions) and t (threshold number of matches for successful lookup). In Table 4, we look at the performance for the *hs2* and *zymo* datasets with varying k (default is $k = 23$). The impact of k is dependent on the genome size since for a larger genome we expect to get more false positive matches as k decreases. For the *hs2* dataset, we see that the compression time increases as k decreases. The compression first improves with decreasing k since smaller k-mers are less likely to have errors leading to fewer missed matches. But as k decreases further, the compression worsens due to increasing false positive matches. For the *zymo* dataset which consists of bacterial and yeast genomes, we see that the compression improves until $k = 12$. Thus, the value of k should be chosen based on the genome size and error rate, with the default value of $k = 23$ chosen to give a reasonable tradeoff for most human datasets.

Table 5 shows the performance for the same two datasets as the parameters n and t are varied (default values $n = 60, t = 6$). As the ratio t/n increases, we miss more and more matches, while very small values lead to false positive matches and increased computation time for minimap2 alignment. While the best value of this ratio depends on the dataset and the error rate, we found the ratio of 0.10 to work consistently well. For a fixed ratio t/n, we expect higher n to give better results (Berlin *et al.*, 2015) at the cost of more computation. In practice we found that the dependence of compression ratio on n was not significant within a range of values, and we chose $n = 60$ based on experiments across datasets. Finally, we note that the memory usage stays roughly constant with the three MinHash parameters.

3.1.4 Minimap2 parameters

NanoSpring uses the minimap2 aligner (Li, 2018) to align candidate reads to the consensus sequence and add them to the graph during contig generation. We tuned the default minimap2 parameters to improve the performance of the algorithm, focusing on three main parameters: the minimizer k-mer size, the minimizer window size w and the maximum

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

k	w	mci	Compressed size		Compression time		Peak memory usage	
			hs2	*hs3*	*hs2*	*hs3*	*hs2*	*hs3*
15	10	5000	0.646	0.670	6h18m	12h45m	38.7	55.3
15	10	400	0.661	0.686	4h01m	5h16m	39.3	57.4
20	50	5000	0.649	0.688	2h54m	5h39m	38.6	46.2
20	50	400	0.665	0.662	2h59m	3h38m	39.9	38.7

Table 6. Performance of NanoSpring with varying minimap2 k, w and mci (max-chain-iter) parameters for the $hs2$ and $hs3$ datasets. Compressed size is in bits/base and the peak memory usage is in GB.

number of iterations during the chaining step (max-chain-iter or mci). We changed the default minimap2 setting of ($k = 15, w = 10, mci = 5000$) to ($k = 20, w = 50, mci = 400$) in order to get the best tradeoff between compressed size and compression time. Table 6 shows the results for four parameter settings for the *hs2* and *hs3* datasets. We observe that the parameters have a significant effect on compression time, while the impact on compressed size is much smaller. The choice of the appropriate parameters was especially crucial for the *hs3* dataset, which requires 3 times more time with the default minimap2 setting as compared to the setting used in NanoSpring (the memory usage also improves for the chosen parameters). Based on some tests, we believe that this effect is due to the presence of more repetitive sequences in the *hs3* dataset which leads to extremely slow alignment unless the max-chain-iter parameter is reduced.

Compressor	Compressed size (bits/base)
Gzip	0.873
BSC	0.685
LZMA2	0.683
BSC+LZMA2	0.658

Table 7. Effect of stream compressor on compressed size for the $hs2$ dataset. In the last row which corresponds to the default setting for NanoSpring, the stream of erroneous bases is compressed using LZMA2 and the other streams are compressed using BSC.

3.1.5 Stream compressor

Table 7 shows the effect of the stream compressor on the overall compression ratio. NanoSpring uses LZMA2 for compressing the stream of erroneous bases and BSC for the other streams. We see that Gzip performs around 30% worse than the default setting. But NanoSpring with Gzip still has 2x better compression than ENANO suggesting that the advantage of NanoSpring is mostly due to the approximate assembly process. Between BSC and LZMA2, we found that BSC was better for most streams, but LZMA2 was around 10% better for the erroneous base stream which is a major contributor to the overall compressed size for certain datasets. Thus, we use a combination of the two for best results. We note that the computational requirements for stream compression are relatively small as compared to the approximate assembly process and were not a major factor in the choice of the compressor.

3.1.6 Threads

Table 8 shows the compression performance of NanoSpring with different number of threads on the *hs2* dataset (where we used 20 threads by default for the main experiments). The time required for compression reduces as we use more threads, while the peak memory usage increases because the threads work simultaneously on different contigs. We observe that the compressed size is roughly constant with the increasing thread count, apart from minor experimental variation. For very high number of threads, the disk I/O and memory allocations become the bottleneck leading to diminishing returns.

Number of threads	Compressed size (bits/base)	Compression time	Peak memory usage (GB)
5	0.651	8h34m	28.2
10	0.653	4h59m	31.8
20	0.658	3h19m	39.0
40	0.646	2h12m	51.2

Table 8. Performance of NanoSpring with different number of threads for the $hs2$ dataset.

Edge threshold	Compressed size (bits/base)	Compression time	Peak memory usage (GB)
1M	0.692	2h11m	28.8
4M	0.665	2h59m	39.9
16M	0.651	3h12m	80.6
∞	0.648	3h41m	113.8

Table 9. Performance of NanoSpring with different edge thresholds for the $hs2$ dataset.

3.1.7 Edge threshold

To reduce the peak memory usage during compression, NanoSpring imposes a limit on the number of edges in a consensus graph during contig generation (4 million by default). Table 9 shows the compression results for the *hs2* dataset for different values of the threshold. The last row shows the result without any edge threshold, and we see the memory usage is more than 100 GB as compared to roughly 40 GB for the default threshold. In general, we see that decreasing the threshold significantly reduces the memory usage and slightly reduces the time but leads to more fragmented contigs and worse compression. We selected 4M as the default threshold to achieve a practical tradeoff between these factors.

Basecaller mode	Mean error rate	Compressed size (bits/base)	Compression time	Peak memory usage (GB)
fast	7.01%	0.698	27.5s	5.8
hac	4.59%	0.504	29.6s	6.2
sup	3.68%	0.415	29.3s	7.2

Table 10. Performance of NanoSpring for the $S. aureus$ dataset under different basecaller modes.

3.1.8 Impact of basecalling error rate

To understand the impact of the basecaller error rate on the performance of NanoSpring, we basecalled the *S. aureus* dataset with three modes of the Guppy basecaller (version 5.0.7) by ONT: *fast*, *hac* (high-accuracy) and *sup* (super-accurate) (where we used the *hac* mode to obtain the *sa* dataset in Table 1). As shown in the Table 10, there is significant impact of the basecaller mode on the mean error rate and the compressed size. The compression time stays roughly constant while the memory usage increases for higher accuracies possibly because we get larger contigs in that case. For the *sup* mode with mean error rate below 4%, the compressed size is close to 0.4 bits/base, showing the potential for better compression as the basecaller quality continues to improve in the near future.

bioRxiv preprint doi: https://doi.org/10.1101/2021.06.09.447198; this version posted June 10, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY 4.0 International license.

	$hs2$		sa	
	RENANO	NanoSpring	RENANO	NanoSpring
Compressed size	0.501	0.656	0.377	0.504
Alignment time	4h55m	-	7.7s	-
Alignment memory usage	25.7	-	1.7	-
Compression time	18m43s*	2h44m	6.3s*	29.6s
Compression memory usage	6.0*	39.1	0.4*	6.2

Table 11. Performance of NanoSpring and RENANO on $hs2$ and sa datasets. Compressed size is in bits/base (only includes read sequences for RENANO) and the alignment memory usage and compression memory usage are in GB. *RENANO time/memory figures include the time/memory to compress the entire FASTQ file including read sequences, quality values and read identifiers.

3.1.9 Comparison with RENANO

To better understand the limits of nanopore read compression, we compared NanoSpring with RENANO (Dufort y Álvarez *et al.*, 2021). RENANO is a recently released reference-based compressor for FASTQ files, and for a fairer comparison we use the mode which produces a self-contained compressed file that does not need the reference during decompression. We evaluated RENANO on two datasets: *hs2* (using the GRCh38 reference) and *sa* (using a high quality assembly for the specific sample, obtained from Wick *et al.* (2019)). For using RENANO, we first aligned the FASTQ file to the reference and then ran RENANO as recommended in their documentation. All tools were run with 20 threads. Table 11 shows the compressed size, as well as the time and memory usage both for minimap2 alignment and compression. We see that the compressed size for RENANO is around 25% smaller than that for NanoSpring, suggesting scope for further improvement in the NanoSpring algorithm. In terms of time and memory usage, we see that RENANO is significantly less resource-intensive as compared to NanoSpring. However, the overall process for RENANO is much slower for the human dataset because of the minimap2 alignment. Overall, we see that the reference-free NanoSpring is only around 20-30% worse in compression than the state-of-the-art reference-based compressor while being competitive in terms of overall time for processing and compression.

4 Conclusion

We present NanoSpring, a specialized reference-free compressor for nanopore sequencing reads, relying on an approximate assembly approach to achieve close to 3x improvement in compression over previous compressors for recent high quality datasets. With rapidly improving basecalling quality in the past few years, we can expect the advantages of such an approach to grow further. NanoSpring offers fast decompression and practical computational requirements during compression, although it uses more computational resources than previous compressors. NanoSpring is open-source and available on GitHub at `https://github.com/qm2/NanoSpring`. Future work includes improvement in the computational requirements and research on the assembly algorithm to get closer to the limits. Another important direction is to incorporate NanoSpring into a full-fledged FASTQ compressor capable of handling quality scores and read identifiers, possibly by combining the best aspects of ENANO and NanoSpring.

Acknowledgements

We thank the Stanford EE REU program for providing the opportunity to initiate this project.

Funding

The authors acknowledge funding from Philips.

References

Belser, C. *et al.* (2021). Telomere-to-telomere gapless chromosomes of banana using nanopore sequencing. *bioRxiv*.

Berlin, K. *et al.* (2015). Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat Biotechnol*, **33**(6), 623–630.

Broder, A. (1997). On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29.

Burrows, M. and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer.

Chandak, S. *et al.* (2019). SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, **35**(15), 2674–2676.

Chandak, S. *et al.* (2020). Impact of lossy compression of nanopore raw signal data on basecalling and consensus accuracy. *Bioinformatics*, **36**(22-23), 5313–5321.

Dufort y Álvarez, G. *et al.* (2020). ENANO: Encoder for NANOpore FASTQ files. *Bioinformatics*, **36**(16), 4506–4507.

Dufort y Álvarez, G. *et al.* (2021). RENANO: a REference-based compressor for NANOpore FASTQ files. *bioRxiv*.

Jain, M. *et al.* (2018). Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, **36**(4), 338–345.

Kowalski, T. M. and Grabowski, S. (2019). PgRC: pseudogenome-based read compressor. *Bioinformatics*, **36**(7), 2082–2089.

Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**(18), 3094–3100.

Limasset, A. *et al.* (2017). Fast and scalable minimal perfect hashing for massive key sets. *ArXiv e-prints*.

Myers, E. W. (1986). An O(ND) difference algorithm and its variations. *Algorithmica*, **1**(1-4), 251–266.

Nicholls, S. M. *et al.* (2019). Ultra-deep, long-read nanopore sequencing of mock microbial community standards. *GigaScience*, **8**(5). giz043.

Nurk, S. *et al.* (2021). The complete sequence of a human genome. *bioRxiv*.

Ochoa, I. *et al.* (2016). Effect of lossy compression of quality scores on variant calling. *Briefings in Bioinformatics*, **18**(2), 183–194.

Ruan, J. and Li, H. (2020). Fast and accurate long-read assembly with wtdbg2. *Nature methods*, **17**(2), 155–158.

Sun, J. *et al.* (2021). Benchmarking oxford nanopore read assemblers for high-quality molluscan genomes. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, **376**(1825), 20200160.

Wick, R. R. *et al.* (2019). Performance of neural network basecalling tools for Oxford Nanopore sequencing. *Genome biology*, **20**(1), 1–10.

Witten, I. H. *et al.* (1987). Arithmetic coding for data compression. *Communications of the ACM*, **30**(6), 520–540.

Yu, Y. W. *et al.* (2015). Quality score compression improves genotyping accuracy. *Nature biotechnology*, **33**(3), 240–243.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, **23**(3), 337–343.