Compression of genomic sequencing reads via hash-based reordering: Supplementary Data

Shubham Chandak, Kedar Tatwawadi and Tsachy Weissman

August 25, 2017

In this document, some instructions for downloading datasets, installing and running the compression tools and some additional results are provided. The proposed algorithm HARC and related instructions can be found at `https://github.com/shubhamchandak94/HARC`.

1 Datasets

For all the datasets listed below, the FASTQ files from the provided links were downloaded, decompressed and concatenated into a single FASTQ file. For some datasets, only one file out of the pair was used because the current implementation of HARC supports only fixed-length reads.

***P. aeruginosa* - SRR554369**

`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR554/SRR554369/SRR554369_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR554/SRR554369/SRR554369_2.fastq.gz`

***S. cerevisiae* - SRR327342_1**

`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR327/SRR327342/SRR327342_1.fastq.gz`

***T. cacao* - SRR870667_2**

`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR870/SRR870667/SRR8700667_2.fastq.gz`

***H. sapiens* 1 - ERR174310**

`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174310/ERR174310_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174310/ERR174310_2.fastq.gz`

***H. sapiens* 2 - ERR194146**

`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR194/ERR194146/ERR194146.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR194/ERR194146/ERR194146_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR194/ERR194146/ERR194146_2.fastq.gz`

This dataset is part of the SAM datasets in the MPEG benchmarking dataset. Thus, the reads were shuffled to obtain random ordering in the FASTQ file.

***H. sapiens* 2 - ERP001775_1**

`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174324/ERR174324_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174325/ERR174325_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174326/ERR174326_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174327/ERR174327_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174328/ERR174328_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174329/ERR174329_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174330/ERR174330_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174331/ERR174331_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174332/ERR174332_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174333/ERR174333_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174334/ERR174334_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174335/ERR174335_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174336/ERR174336_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174337/ERR174337_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174338/ERR174338_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174339/ERR174339_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174340/ERR174340_1.fastq.gz`
`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174341/ERR174341_1.fastq.gz`

1.1 Generating Simulated reads

To generate the simulated reads used in the paper, first download the human genome from

`ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/human_g1k_v37.fasta.gz`

Then separate the particular chromosome (e.g. Chromosome 1 to chrom1.fasta).
Change folder to
`HARC/util/gen_fastq_noRC`
Then run `make`. To generate 35000000 noiseless reads of length 100 from chrom1.fasta without any reverse complementation, run
`./gen_fastq_noRC 35000000 100 PATH/chrom1.fasta PATH/chrom1_reads.fastq`
For reads with 1% uniform noise, add a `-e` at the end (not used in this work).

2 Installing and running tools

2.1 Installation

HARC

```
git clone https://github.com/shubhamchandak94/HARC.git
cd HARC
./install.sh
```

Orcom

Boost should already be installed.

```
git clone https://github.com/lrog/orcom.git
cd orcom
make boost
```

Leon

```
git clone https://github.com/GATB/leon.git
cd leon
sh install
```

BSC

```
git clone https://github.com/shubhamchandak94/libbsc.git
cd libbsc
make
```

pigz

```
git clone https://github.com/madler/pigz.git
cd pigz
make
```

7-zip

```
sudo apt-get install p7zip
```

2.2 Running compression algorithms

HARC

Change directory to HARC/

```
Compression - compresses FASTQ reads. Output written to .harc file
./harc -c FASTQ_file  [-p] [-t num_threads] [-q]
-p = Preserve order of reads (compression ratio 2-4x worse if order preserved)
```

```
-t = num_threads - default 8
-q = Write quality values to .quality file and read identifiers to .id file.
Quality values and read IDs are appropriately reordered if -p is not specified.

Decompression - decompresses reads. Output written to .dna.d file
./harc -d HARC_file [-p] [-t num_threads] [-m max_memory]
-p = Get reads in original order (slower). Only applicable if -p was used during
compression.
-t = num_threads - default 8
-m max_memory - Controls memory-time tradeoff for decompression with -p.
Specify max memory in GB (minimum 3 GB for 8 threads). e.g. -m 10 for 10 GB maximum memory.
Default: 7 GB (note: less than 3 GB memory required if -p not specified)

Help (this message)
./harc -h
```

Orcom

Change directory to orcom/bin/
Compression:

```
./orcom_bin e -iPATH/file.fastq -oPATH/file.bin
./orcom_pack e -iPATH/file.bin -oPATH/file.orcom
```

Decompression:

```
./orcom_pack d -iPATH/file.orcom -oPATH/file_orcom.dna
```

Leon

Change directory to leon.
Compression (sequence only mode):

```
./leon -file PATH/file.fastq -seq-only -c
```

Decompression:

```
./leon -file PATH/file.leon -d
```

BSC

Change directory to libbsc.
Compression:

```
./bsc e file_to_compress file_to_compress.bsc -b64p -t<num_thr>
```

Deompression:

```
./bsc d file_to_compress.bsc file_to_compress -t<num_thr>
```

pigz

Change directory to pigz.
Compression:

```
./pigz -f -k -p <num_thr> file_to_compress
```

Deompression:

```
./pigz -d -f -k -p <num_thr> file_to_compress.gz
```

7-zip

7-zip is used as part of HARC stage III and in Table 8 of the paper.
Compression:

```
7z a file_to_compress.7z file_to_compress -mmt=<num_thr>
```

Deompression:

```
7z e file_to_compress.7z -o.
```

3 Computation of the noise entropy upper bound

As noted in [7] and [6], the quality values and the noise in the reads in a FASTQ file follow a complex distribution. Most reads are relatively noiseless, while a few reads are extremely noisy. In many cases, the tail of the read has consecutive bases with very low quality values. The quality value compressor QVZ [6] uses a first order Markov model for the quality values. Even if quality values truly followed a first-order model, the noise in the reads need not follow such a model. This is because noise is a (random) function of the quality values and thus is a hidden Markov process which need not be a Markov process of finite order. QVZ also uses the idea of clustering quality values to capture the fact that even the quality values are not actually first-order Markov. They initially cluster reads with similar quality values into bins and then apply the first-order model separately to each bin.

Based on these insights, we use a second-order Markov model for noise along with the clustering idea from QVZ. For simplicity, we cluster the quality values into four bins based on their average quality value (0-10, 10-20, 20-30 and 30-40). Within each cluster, we obtain the counts (up to 3-mers) for the quality values at each position and then use these to calculate the noise entropy upper bound assuming that the noise follows a second-order Markov model. We also make use of the fact that probability of one base being wrongly called as another is not equally likely, e.g., C is much more likely to be called as an A than a G or a T (Figure 6(b) in [7]). We found that more complex models than this provide negligible improvement in the upper bound but lead to large increase in computational costs so we limited the model complexity to the current model.

Notation: Let len be the read length. Let $N = (N_1, \ldots, N_{len})$ denote the random variable representing noise in a read, where N_i represents noise at the i^{th} base. We are interested in computing $H(N)$, an upper bound on the entropy of noise.

First, we bound $H(N)$ with respect to the noise entropies for the four clusters. Let $p_i,\ i = 1, 2, 3, 4$ denote the empirical probability for the four clusters

and $H(N|i),\ i = 1, 2, 3, 4$ be the entropies of the noise in the four clusters. Then, using the chain rule for entropy,

$$H(N) \leq \sum_{i=1}^{4} -p_i \log p_i + \sum_{i=1}^{4} p_i H(N|i) \tag{1}$$

Next, we focus on the computation of the terms $H(N|i)$, which we denote by $H(N)$ for simplicity (we'll now focus on a single cluster). Note that,

$$H(N) = H(N_1, \ldots, N_{len}) = \sum_{i=1}^{len} H(N_i|N_1, \ldots, N_{i-1}) \leq \sum_{i=1}^{len} H(N_i|N_{i-2}, N_{i-1}) \tag{2}$$

by using the fact that conditioning reduces entropy. The term $H(N_i|N_{i-2}, N_{i-1})$ can be computed using its definition once we obtain the probabilities $p(N_i|N_{i-2}, N_{i-1})$. Now, denote the quality value at position i by q_i and the corresponding random variable by Q_i. Our generation model for noise suggests that noise at neighboring bases should be independent given the corresponding quality values and that the noise at a particular base should be independent of the quality values at neighboring bases given the quality value at that base. Using these independence relations, we get,

$$p(N_i|N_{i-2}, N_{i-1}) = \sum_{q_{i-2}, q_{i-1}, q_i} p(N_i, q_{i-2}, q_{i-1}, q_i|N_{i-2}, N_{i-1}) \tag{3}$$

$$= \sum_{q_{i-2}, q_{i-1}, q_i} p(q_{i-2}, q_{i-1}, q_i) p(N_i|q_{i-2}, q_{i-1}, q_i, N_{i-2}, N_{i-1}) \tag{4}$$

$$= \sum_{q_{i-2}, q_{i-1}, q_i} p(q_{i-2}, q_{i-1}, q_i) p(N_i|q_i) \tag{5}$$

The first term in the sum can be replaced by the empirical probabilities obtained from the 3-mer counts for the quality values at position i. For the second term $p(N_i|q_i)$, note that N_i can take four values - one value corresponding to no noise, and the other three for the three possible substitutions (we ignore bases marked 'N' as they comprise a very small fraction of the FASTQ file). To account for the higher probability of certain substitutions, we assume that each noisy base has 0.7 probability of being the most likely substitution and 0.15 probability of being the other two possibilities (based on Table 6(b) in [7]). This means that (for quality value of q_i), $P(A|A) = (1 - 10^{-q_i})$, $P(C|A) = 10^{-q_i} \times 0.7$, $P(C|A) = 10^{-q_i} \times 0.15$ and $P(C|A) = 10^{-q_i} \times 0.15$. The computation is robust to the exact figures used (0.7 and 0.15) and slight changes to these have negligible impact on the overall result. In our computations, we assume that the random N_i can take values 0, 1, 2 and 3, where 0 represents no error, 1 represents the most likely substitution and 2 and 3 represent the other two substitutions.

The HARC github repository contains code for the computation of the quality value counts and the noise entropy computation. This code is present

quality_counts.cpp and noise_entropy.py in the util/ directory (usage details in README). Table 1 details the noise entropy along with the other components for the real datasets used for the experiments.

4 Additional results

This section contains some additional data related to the experiments referenced in the paper.

4.1 Entropy computation

Table 1 contains detailed breakdown of the entropy computation for Tables 2 and 3 in the paper. The multinomial term is included in the entropy without preserving order, while the order entropy is included in the entropy with order preserved (L_{FASTA}, rev. comp. and noise entropy are common to both). We observe that the noise component is a major contributor in the order non-preserving case, and it is quite high for the first and the last dataset, where HARC performs slightly better than the entropy upper bound. The order entropy is the largest component in the order-preserving case and is usually more than thrice of all other terms combined.

Dataset	No. of reads (m)	Genome length (n)	L_{FASTA}	Multinomial $\log_2 \binom{n+m-1}{m}$	Rev. Comp.	Noise	Order entropy $m \log_2(n)$
P. aeruginosa	3.3	6	1.43	1.09	0.42	7.19	9.34
S. cerevisiae	15	12.1	2.81	3.36	1.88	9.98	58.9
T. cacao	69	350	62.3	33.9	8.65	47.1	364
H. sapiens 1	415	3,234	599	233	51.9	436	2,726
H. sapiens 2	1,635	3,234	599	560	204	838	8,097
H. sapiens 3	3,422	3,234	599	831	428	2,284	16,822

Table 1: Individual components of the entropy computation. No. of reads and genome length are in Million bases. Rest of the components are in MB.

4.2 Unmatched reads and singletons

Table 2 contains the number of reads which remain unmatched and singletons after stage I (recall that singleton reads are subset of unmatched reads which correspond to contigs of size 1). The second and third columns contain the number of reads in the original dataset without 'N' and with 'N' respectively. The table also contains the percentage of singletons and reads with 'N' that are realigned in stage II. Recall that the reads with 'N' are not considered in stage I and are directly considered in the realignment stage. From the table, we observe that around 3-6% of reads remain unmatched after stage I and a significant fraction (60-90%) of unmatched reads are singletons. We also observe that the

realignment stage is quite effective and is able to recover 50-90% singletons. Most of the reads with 'N' also get realigned (these are typically reads with only a few bases as 'N' and most other bases noiseless). Adding the realignment stage to the algorithm provided us with 15-20% improvement in compression for most datasets.

Dataset	#Reads without 'N'	#Reads with 'N'	Stage I unmatched	Stage I singletons	% singletons realigned	% reads with 'N' realigned
P. aeruginosa	3.26	0.057	0.24	0.20	70	79
S. cerevisiae	14.4	0.62	0.61	0.54	78	95
T. cacao	68.8	0.39	5.34	3.51	52	46
H. sapiens 1	410	5.10	26.8	17.3	55	91
H. sapiens 2	1,616	19.1	60.5	52.3	61	93
H. sapiens 3	3,415	6.4	92.2	80.1	68	90

Table 2: Total reads without and with 'N', number of unmatched and singleton reads after stage I and the percentage of these reads realigned in stage II. All numbers are in Millions

4.3 Number of threads

We tested HARC (compression) on various number of threads and the results are shown in Table 3. These results were obtained for the largest dataset *H. sapiens* 3 in the order non-preserving mode. The compression time improves with the number of threads. The improvement is not proportional to the number of threads because of certain serial segments of the algorithm, especially those involving disk read/write. The compression ratio degrades very gradually with increase in the number of threads and the compression RAM remains nearly constant.

#threads	Compression Time	Compression RAM (GB)	Compressed Size (MB)
4	8h33m	163	4,065
8	5h22m	163	4,072
16	4h21m	163	4,079
32	3h26m	163	4,082

Table 3: Compression performance vs. #threads for *H. sapiens* 3 dataset.

4.4 Decompression memory-time tradeoff

While HARC is very efficient in terms of both decompression time and memory requirements, it is possible to reduce one at the expense of the other. In the

order non-preserving mode, the major contributor towards the memory requirement is BSC decompression, which requires around 350 MB/thread. Thus, the decompression memory can be reduced by reducing the number of threads. In the order preserving mode, another contributor towards the memory requirement is the order restoration step, where the reordered reads are read to the memory in chunks and then written back in their original order. HARC provides a switch (-m) to control the size of this memory to tradeoff between compression time and memory. Table 4 contains the decompression time and memory used for 5 different parameters used on *H. sapiens* 3 dataset (order-preserving mode). The decompression time reduces for higher memories, but the improvement is not significant for larger memory sizes.

Switch used	Decompression time	Decompression Memory
-m 3	3h3m	3.02
no switch (-m 7)	2h38m	7.03
-m 10	2h34m	10.05
-m 20	2h23m	20.09
-m 40	2h17m	40.18

Table 4: Decompression time vs. memory for order preserving decompression for *H. sapiens* 3 dataset. Memory is in GBs.

4.5 Tests on other compressors

Apart from Orcom, we considered two more compressors for in the order non-preserving category: DARRC [4] and Mince [9]. The results for these compressors for the first three datasets are shown in Table 5. DARRC is designed for extremely large coverages (they test up to 50,000x in their paper) and is not competitive with Orcom for the range of coverages considered in this work ($\approx 100x$ or lower). Mince achieves compression closer to Orcom for these datasets but is much slower (10 times slower according to [8]) and requires large amounts of RAM (around 20x more than Orcom according to [8]). Therefore, we decided to use Orcom as the reference compressor for the order non-preserving mode.

Dataset	Orcom	Mince	DARRC
P. aeruginosa	15.84	15.35	20.44
S. cerevisiae	35.97	36.88	52.47
T. cacao	315.13	349.3	481.53

Table 5: Compressed sizes (in MB) for DARRC and Mince along with Orcom on the three smaller datasets used in the paper.

5 HARC parameter selection

In the paper, we provide experimental justification for the choice of $thresh$, $maxshift$ and the stage III compressor. In this section, we show detailed results of how the other parameter choices affect the performance of HARC. These experiments are all run on *H. sapiens* 1 dataset which consists of 415 Million reads of length 101 each. These experiments were run with 16 threads.

5.1 Stage I substring indices

The central idea behind HARC is to index reads using subtrings in the middle of the read. By default, HARC uses two substrings of length 32 to index the reads. Table 6 shows compressed sizes for the default and two other choices of these substrings. The first row contains the default choice of indices. Using only one substring leads to marked deterioration in the compression ratio as seen in the second row. Using shorter indices improves compression slightly because the probability of having a noisy base in the index reduces. However, the compression time is affected significantly by shorter substrings (around 1.4x worse than default for this experiment). Thus, we went with 2 dictionaries of length 32 as the default setting. Note that 32 bases translate to 64 bits which is the size of uint64_t data type used

Substring indices	Compressed size
18-49, 50-81	1,501
18-49	1,997
30-49, 50-69	1,484

Table 6: Compressed sizes (in MB) for HARC running on *H.sapiens* 1 dataset for different choices of substring indices in stage I. First row is the default choice.

5.2 Choice of hash map used

HARC uses a minimal perfect hash library, BBHash [5] for the hashing the substring indices. In the current implementation with BBHash, HARC needs 9 bytes/read/dictionary of RAM usage. We need to store the read ids (4 bytes) and a mapping from the hash value to the index in an array containing the read ids (4 bytes). An additional byte stores a Boolean variable indicating that the bin is empty. We do not store the value of the index (i.e. the substring which needs 8 bytes/read/dictionary) because we can check the index from the reads in the corresponding hash bin.

Earlier we tried out some standard unordered maps in C++ (e.g. std:: unordered_map, sparsepp [10], Google's sparsehash [3]). First of all, these store the substring index increasing the memory consumption by 8 bytes/read/dictionary. This alone increases the memory consumption for the H. sapiens 3 dataset from 163 GB to around 218 GB. It might be possible to overcome this by a

technique similar to that used for BBHash above, but we decided to go with the simplicity offered by BBHash. Also, sparsepp has an additional overhead of 5-10 bytes/entry (leading to around 250 GB RAM for H. sapiens 3) and Google's sparsehashmap is very slow (see benchmark [1]). Thus, considering the time/memory tradeoff, we decided to go with BBHash.

5.3 Maximum search limit in hash table

As mentioned in the paper, we put a limit of 1000 on the maximum number of searches in the hash table at each step. This makes sure that the uneven distribution of reads in the hash table bins does not affect the compression speed of HARC. Without this limit, the compression time increases quadratically for the larger datasets, which is unacceptable. We also use this limit in the second stage while realigning singletons and reads with 'N'. Table 7 shows compressed sizes for the default and two other choices of this limit for the *H. sapiens* 1 dataset. On this dataset, the compression times are almost the same for all three values. We also see that the limit has very little effect on the compressed size and we chose 1000 as the default to provide good compression for the larger datasets while maintaining good compression speed.

Maximum search limit	Compressed size
500	1501.32
1000	1501.03
2000	1500.68

Table 7: Compressed sizes (in MB) for HARC running on *H. sapiens* 1 dataset for different choices of maximum search limit in hash table bins. Second row is the default choice.

5.4 Stage II substring indices

Similar to the experiments on the substrings for stage I, we also tested various options for stage II substrings which are used for the realignment of singletons and reads with 'N'. Table 8 shows the results. Using only one index leads to fewer reads getting realigned and poorer compression (second row). Using shorter substrings as indices or using three indices slightly benefits compression but increases the time required for the realignment stage. Thus, we decided to use two substrings of size 21 each. In this stage, the bases are represented using 3 bits per base representation (to handle reads with 'N') and 21*3 = 63 is largest possible size less than 64, which is the size of uint64_t datatype used for indexing.

Stage II substring indices	Compressed size	% singletons realigned
0-20, 21-41	1501.03	55.2
0-20	1576.33	46.9
0-14, 15-29	1496.83	58.4
0-20, 21-41, 42-62	1493.38	59.0

Table 8: Compressed sizes (in MB) and percentage of singletons realigned for different choices of stage II substring indices for HARC running on *H. sapiens* 1 dataset. First row is the default choice.

5.5 *thresh* for stage II realignment

The singleton reads considered in stage II are typically more noisy than the rest of the reads. Therefore, while aligning these reads, we need to keep the Hamming distance threshold higher than that used in stage I ($thresh = 4$ in stage I). Table 9 shows the compressed sizes and percentage of singletons aligned for a range of thresholds. As the threshold increases, more singletons get realigned. However these singletons tend to be more noisy and hence the compression starts getting worse for thresholds higher than 24, which is the default choice.

Realignment threshold	Compressed size	% singletons realigned
16	1505.44	48.3
20	1501.25	52.0
24	1501.03	55.2
28	1502.66	58.3
32	1505.18	61.0

Table 9: Compressed sizes (in MB) and percentage of singletons realigned for different choices of stage II threshold for HARC running on *H. sapiens* 1 dataset. 24 is the default choice.

6 Theoretical analysis

We present here some theoretical analysis for HARC and other compressors:

6.1 Ordering according to genome position

If there was a way to reorder reads according to their position in the genome, then we could achieve good compression by exploiting the redundancy between neighboring reads. We can represent a read in terms of its previous read, by storing the length of the match with the previous read and the new symbols in the read separately. Note that the sequence formed by the new symbols exactly corresponds to the portion of the genome covered by the reads, and can be stored

using at most L_{FASTA} bits. Storing the match lengths can be done in $H(Z|X)$ space. The offset of the current read from the previous read is distributed as a geometric random variable with mean m/n, whose entropy is close to the upper bound on the $H(Z|X)$ term. Recall that $H(Z|X) = \log_2 \binom{n+m-1}{m} \approx (m+n)\log_2(m+n) - m\log_2 m - n\log_2 n$ by Stirling's approximation. The entropy of geometric random variable with mean μ is $(\mu+1)\log_2(\mu+1) - \mu\log_2\mu$. Substituting $\mu = m/n$ and multiplying by the number of reads m, we recover the expression for $H(Z|X)$. We can use an entropy achieving prefix code for geometric random variables to achieve this entropy, e.g., Golomb codes [2]. Thus, we observe that if the reads are reordered according to their position in the genome sequence, then the upper bound on the entropy as expressed in equation 2 in the paper can be achieved.

6.2 Analysis of minimizer-based techniques

Existing reordering compressors like Orcom and Mince cluster the reads into bins based on some shared substring (e.g., the minimizer) and then the bins are encoded and compressed independently. Recall that the minimizer of length l of a read is the lexicographically smallest l-length substring in the read. We assume that the reads within each bucket are reordered in an optimal fashion. For example, Orcom orders the reads in a minimizer bin by lexicographically sorting according to portion of read after the minimizer concatenated with portion of read before the minimizer. This heuristic allows the reads overlapping in the genome to be together.

We were interested in understanding whether such techniques can achieve the entropy upper bound for compression of reads. Experiments on simulated noiseless reads (Table 6 in paper) clearly showed that the minimizer based techniques didn't achieve the entropy upper bound. To further understand this gap to entropy, we tried to analyze the problem mathematically. We provide the heuristic arguments below. Although these do not provide a rigorous proof, they are useful to understand the loss incurred by binning at an intuitive level.

Let len be the length of a read and n be length of genome. To analyze such schemes we assume that as we slide along the genome, the minimizer (of length $l \approx 10$) of len-length substrings changes after every $len/2$ positions (based on results in "Reducing storage requirements for biological sequence comparison"). This is reasonable because the reads starting at 1 and $len+1$ are non-overlapping and hence are likely to have different minimizers. Thus the minimizer has to change at some point between 1 and $len+1$. This effectively means that the reads starting at positions between 1 and $len/2$ go to one bucket and those starting at positions between $len/2+1$ and len go to another bucket. Note that reads from different portions of the genome can have the same minimizer. However for simplicity and noting that $\Sigma^{len} \gg n$, we assume that the overlap between such reads is insignificant.

Now we compute the overhead caused by the binning process. Consider the portion of the genome from $len+1$ to $3 \times len/2$. This portion is present in a read starting at position (i) $len/2-1$, (ii) $len/2+1$ and (iii) $len+1$.

As per our assumption above, these three reads have different minimizers and hence go to different bins. Thus the portion of the genome from $len + 1$ to $3 \times len/2$ is encoded three times. Thus instead of spending $n \log_2(\Sigma)$ on $H(X)$ we are effectively spending $3n \log_2(\Sigma)$. It can be shown that there is no overhead spent with respect to the $H(Z|X)$ term in this idealized setting (basically we can use the same geometric distribution entropy achieving codes as described in Section 6.1). This suggests that the gap to entropy for Orcom should stay constant with coverage. This is not observed in the results for simulated noiseless reads in the paper (Table 6), and the difference between the compressed size obtained by Orcom and the entropy increases with coverage. This might be due to oversimplified analysis above, or due to additional inefficiencies in Orcom, or due to the fact that Orcom incurs some overhead here since it is designed and optimized for real data with noise and reverse complmenetation.

6.3 Analysis of proposed algorithm for noiseless reads

If the genome has no repeats of length k, then the proposed algorithm will find the read closest to the current read on the genome with high probability (as long as read is not shifted by more than $maxshift$). The first read can be from any position in the genome and hence the algorithm will find matches till it reaches the end of genome. After that it will pick some read from the part of the genome before the first read and so on. Assuming that the genome is completely covered by the reads and that the algorithm finds the read closest to the current read on the genome for each read, we show below that on average the number of unmatched reads will be $O(\log_2(n))$ where n is the length of the genome (ignoring edge effects throughout).

Denote the expected number of unmatched reads in the process described above by U_n, a function of the genome length. If the first read is from position n' in the genome (n' is uniformly distributed in 1 to n), then the expected number of unmatched reads is $1 + U_{n'-1}$ (1 for the first read and $U_{n'-1}$ for the similar process on the portion of genome before the first read). Thus, we get the recurrence

$$U_n = 1 + \frac{1}{n} \sum_{n'=1}^{n} U_{n'-1}$$

with $U_0 = 0$. Changing the summation index,

$$U_n = 1 + \frac{1}{n} \sum_{n'=0}^{n-1} U_{n'}$$

Writing the recurrence for U_{n-1}

$$U_{n-1} = 1 + \frac{1}{n-1} \sum_{n'=0}^{n-2} U_{n'}$$

Combining the two equations above,

$$U_n = \frac{1}{n} + U_{n-1}$$

Thus,

$$U_n = \sum_{n'=1}^{n} \frac{1}{n'}$$

which is $O(\log_2(n))$ [11].

7 Extensions

We describe some possible extensions to HARC which we plan to add in the future.

7.1 Handling variable-length short reads

The current implementation of HARC does not support variable-length reads (here variable-length reads refer to variable-length short reads and not to 3rd generation sequencing reads). This is tied to the implementation and we plan to add support to such datasets in the future. To handle these reads, certain changes need to be made in the algorithm. In the Hamming distance computation step, the distance should be computed after truncating the longer read so that the read lengths match. The dictionary indices can be chosen in the middle of the read for all the read lengths - thus the exact indices will differ from read to read. Also, the read lengths need to be stored separately in the encoding stage as well.

Another way to handle such datasets is to run the algorithm separately on the portions having the different read lengths. However this is not optimal because the effective coverage for each of the files is lower than the actual coverage.

7.2 Paired-end reads

Currently, HARC works on a single FASTQ file. To compress a file containing paired-end reads while preserving the pairing information, one needs to concatenate the two files and run HARC in the order-preserving mode. However, in a lot of applications, only the read pairing information is relevant and the complete order of the reads is not required. In such cases, it is possible to reduce the (approximately) $n \log_2 n$ bits used for storing the order to $(n \log_2 n)/2$ bits, where n is the number of reads. This can be done by only storing the position of the pairing read for the first read in the pair. Since the order information is the largest component of the total compressed size (50-70%, depending on coverage), storing only the pairing information can further boost compression while retaining the relevant information for downstream applications.

References

[1] Hash table benchmark. `https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html`, 2016.

[2] R. Gallager and D. van Voorhis. Optimal source codes for geometrically distributed integer alphabets (Corresp.). *IEEE Transactions on Information Theory*, 21(2):228–230, March 1975.

[3] Google. sparsehash. `https://github.com/sparsehash/sparsehash`, 2017.

[4] Guillaume Holley, Roland Wittler, Jens Stoye, and Faraz Hach. *Dynamic Alignment-Free and Reference-Free Read Compression*, pages 50–65. Springer International Publishing, Cham, 2017.

[5] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. *CoRR*, abs/1702.03154, 2017.

[6] Greg Malysa, Mikel Hernaez, Idoia Ochoa, Milind Rao, Karthik Ganesan, and Tsachy Weissman. QVZ: lossy compression of quality values. *Bioinformatics*, 31(19):3122–3129, 2015.

[7] André E. Minoche, Juliane C. Dohm, and Heinz Himmelbauer. Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and Genome Analyzer systems. *Genome Biology*, 12(11):R112, 2011.

[8] Ibrahim Numanagic, James K. Bonfield, Faraz Hach, Jan Voges, Jorn Ostermann, Claudio Alberti, Marco Mattavelli, and S. Cenk Sahinalp. Comparison of high-throughput sequencing data compression tools. *Nat. Methods*, 13(12):1005–1008, Dec 2016. Brief Communication.

[9] Rob Patro and Carl Kingsford. Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, 31(17):2770–2777, 2015.

[10] Gregory Popovitch. sparsepp. `https://github.com/greg7mdp/sparsepp`, 2017.

[11] Wikipedia. Harmonic number — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 14-August-2017].