Scrible: Ultra-Accurate Error-Correction of Pooled Sequenced Reads

Denise Duma[1], Francesca Cordero[4], Marco Beccuti[4], Gianfranco Ciardo[5], Timothy J. Close[2], and Stefano Lonardi[2]

[1] Baylor College of Medicine, Houston, TX 77030, USA
[2] Dept. of Computer Science and Eng., University of California, Riverside, CA 92521
[3] Dept. of Botany & Plant Sciences, University of California, Riverside, CA 92521
[4] Department of Computer Science, Università di Torino, 10149 Torino, Italy
[5] Department of Computer Science, Iowa State University, Ames, IA 50011, USA

Abstract. We recently proposed a novel clone-by-clone protocol for *de novo* genome sequencing that leverages combinatorial pooling design to overcome the limitations of DNA barcoding when multiplexing a large number of samples on second-generation sequencing instruments. Here we address the problem of correcting the short reads obtained from our sequencing protocol. We introduce a novel algorithm called Scrible that exploits properties of the pooling design to accurately identify/correct sequencing errors and minimize the chance of "over-correcting". Experimental results on synthetic data on the rice genome demonstrate that our method has much higher accuracy in correcting short reads compared to state-of-the-art error-correcting methods. On real data on the barley genome we show that Scrible significantly improves the decoding accuracy of short reads to individual BACs.

1 Introduction

We have recently demonstrated how to take advantage of combinatorial pooling (also known as *group testing*) for clone-by-clone *de-novo* genome sequencing [1, 8, 9]. In our sequencing protocol, subsets of non-redundant genome-tiling BACs are chosen to form intersecting pools, then groups of pools are sequenced on an Illumina sequencing instrument via standard multiplexing (DNA barcoding). Sequenced reads can be assigned to specific BACs by relying on the structure of the pooling design: since the identity of each BAC is encoded within the pooling pattern, the identity of each read is similarly encoded within the pattern of pools in which it occurs. Finally, BACs are assembled individually, simplifying the problem of resolving genome-wide repetitive sequences.

An unforeseen advantage of our sequencing protocol is the potential to correct sequencing errors more effectively than if DNA samples were not pooled. This paper investigates to what extent our protocol enables such error correction. Due to obvious needs in applications of high-throughput sequencing technology, including *de-novo* assembly, the problem of correcting sequencing errors in short reads has been the object of intense research. Below, we briefly review some

of these efforts, noting that our approach is substantially different, due to its original use of the pooling design.

Most error correction methods take advantage of the high sequencing depth provided by second-generation sequencing technology to detect erroneous base calls. For instance, SHREC [12] carries out error correction by building a generalized weighted suffix tree on the input reads, where the weight of each tree node depends on its coverage depth. If the weight of a node deviates significantly from the expectation the substring corresponding to that node is corrected to one of its siblings. HITEC [4] builds a suffix array of the set of reads and uses the longest common prefix information to count how many times short substrings are present in the input. These counts are used to decide the correct nucleotide following each substring. HITEC was recently superseded by RACER [5], from the same research group, which improves the time- and space-efficiency by using a hash table instead of a suffix array.

Several other error-correction methods are based on k-mer decomposition of the reads, e.g., SGA [13], REPTILE [16] and QUAKE [6]. SGA uses a simple frequency threshold to separate "trusted" k-mers from "untrusted" ones, then performs base changes until untrusted k-mers can become trusted. REPTILE builds a k-mer tiling across reads, then corrects erroneous k-mers based on contextual information provided by their trusted neighbor in the tiling. QUAKE uses a coverage-based cutoff to determine erroneous k-mers, then corrects the errors by applying the set of corrections that maximizes a likelihood function. The likelihood of a set of corrections is defined by taking into account the error model of the sequencing instrument and the specific genome under study. Other methods are based on multiple sequence alignments. For example, CORAL [11] builds a multiple alignment for clusters of short reads, then corrects errors by majority voting.

2 Preliminaries

Our algorithm corrects *reads*, short strings over the alphabet $\Sigma = \{A, C, G, T\}$. With $r[i]$ we denote the i-th symbol in read r. A k-mer α is any substring of a read r such that $|\alpha| = k$. A *BAC (clone)* is a 100–150kb fragment of the target genome replicated in a *E. coli* cell.

2.1 Pooling Design

After the selection of the BACs to be sequenced (see [8, 9] for more details), we *pool* them according to a scheme that allows us to *decode* (assign) sequenced reads back to their corresponding BACs.

The design of a pooling scheme reduces to the problem of building a *disjunctive* matrix $\mathbf{\Phi}$ where columns correspond to BACs to be pooled and rows correspond to pools. Let w be a subset of the columns (BACs) of the design matrix $\mathbf{\Phi}$ and $p(w)$ be the set of rows (pools) that contain at least one BAC in w: the matrix $\mathbf{\Phi}$ is said to be *d-disjunct* (or *d-decodable*) if, for any choice of

w_1 and w_2 with $|w_1| = 1$, $|w_2| = d$, and $w_1 \not\subset w_2$, we have that $p(w_1) \not\subseteq p(w_2)$. Intuitively, d represents the maximum number of positives that are guaranteed to be identified by the pooling design.

We pool BACs using a combinatorial pooling scheme called *Shifted Transversal Design* (STD) [15]. STD is a *layered* design: the rows of the design matrix $\mathbf{\Phi}$ are organized into multiple redundant layers such that each pooled variable (BAC) appears exactly once in each layer, that is, a layer is a partition of the set of variables. STD is defined by parameters (q, L, Γ) where L is the number of layers, q is a prime number equal to the number of rows (pools) in each layer and Γ is the *compression level* of the design. To pool n variables, STD uses $m = q \times L$ pools. The set of L pools defines a unique pooling pattern for each variable, and can be used to retrieve its identity. We call this set the *signature* of the variable. The compression level Γ is defined to be the smallest integer such that $q^{\Gamma+1} \geq n$. STD has the desirable property that any two variables co-occur in at most Γ pools, therefore by choosing a small value for Γ one can make STD pooling very robust to decoding errors. The parameter Γ is also related to the decodability of the design through the equation $d = \lfloor (L-1)/\Gamma \rfloor$. Therefore, Γ can be seen as a trade-off parameter: the larger it is, the more items can be tested, up to $q^{\Gamma+1}$, but fewer positives can be reliably identified, up to $\lfloor (L-1)/\Gamma \rfloor$. For more details on this pooling scheme and its properties, see [15].

2.2 Read decoding

As the read decoding problem is presented elsewhere [1, 8, 9], here we only provide a brief overview to motivate the necessity of correcting reads before decoding them. Given a set of pools $\mathcal{P}$ and a set of BACs $\mathcal{B}$, the *signature* for a BAC $b \in \mathcal{B}$ defines the subset $\mathcal{A} \subset \mathcal{P}$ of pools ($|\mathcal{A}| = L$) in which BAC b is assigned to. Given a set $\mathcal{R}_p$ of reads for each pool $p \in \mathcal{P}$, and the set of all BAC signatures the *read decoding* problem is to determine, for each read $r \in \mathcal{R}_p$, the BAC(s) from which r originated. In [8] we solved the read decoding problem with a combinatorial algorithm, while in [1] we proposed a compressed sensing approach. In [9] we further improved the decoding by a "data slicing" approach. In all cases, we first decompose reads into their constituent k-mers and compute, for each k-mer, the number of times it occurs in each pool (the k-mer *frequency vector*).

The problems of decoding and error-correction are mutually dependent: correcting sequencing errors will improve the accuracy of decoding; a more accurate decoding can help correcting the reads more effectively (as it will become clear later).

3 Methods

3.1 Indexing k-mers

We first preprocess all reads r ($|r| \geq k$) in a pool $p \in \mathcal{P}$ by sliding a window of size k on each read $r \in \mathcal{R}_p$ to produce $|r| - k + 1$ k-mers. The identity of these

k-mers is encoded in a function *poolcount*: $\Sigma^k \times \mathcal{P} \to \mathbf{N}$, where *poolcount*$(\alpha, p)$ is the number of times k-mer α (or its reverse complement) appears in pool p. We also define three additional functions, namely (i) *pools*: $\Sigma^k \to 2^{\mathcal{P}}$ where *pools*(α) is the set of pools where k-mer α appears at least once, (ii) *count*: $\Sigma^k \to \mathbf{N}$ where *count*(α) is the total number of times α appears in any of the pools, and (iii) *bacs*: $\Sigma^k \to 2^{\mathcal{B}}$ where *bacs*(α) is the set of BACs corresponding to *pools*(α), i.e., the BACs whose signature is included in *pools*(α). Observe that *pools*$(\alpha) = \{p \in \mathcal{P} : \textit{poolcount}(\alpha, p) > 0\}$, *count*$(\alpha) = \sum_{p \in \mathcal{P}} \textit{poolcount}(\alpha, p)$, and that *bacs*$(\alpha)$ can be determined by matching *pools*(α) against the set of all BAC signatures. As a consequence, we only need to explicitly store *poolcount* into a hash table.

3.2 Identification and correction of sequencing errors

Taking advantage of the pooling design, we can assume that any k-mer α such that $|\textit{pools}(\alpha)| < L$ (i.e., α occurs in a few pools, less than the expected L) is considered erroneous. This assumption holds when the sequencing depth is sufficiently high, so that each genomic location is covered by several correct k-mers possibly mixed with a few corrupted k-mers. In practice, the depth of sequencing can vary significantly along the genome. When it is particularly low, it is possible (although unlikely) for a correct k-mer to appear in fewer than L pools.

These low-frequency k-mers are also responsible for the large majority of the entries in the hash table for *poolcount*. To save memory, we do not store a k-mer in the hash table during the pre-processing phase if it appears in fewer than l pools, where $1 \leq l < L$ is a user-defined parameter. At the other end, a k-mer α is deemed *repetitive* if $|\textit{pools}(\alpha)| > h$, where h is another user-defined parameter such that $dL < h \leq qL$.

After building the hash table for *poolcount*, we process the reads one by one. If a k-mer α in read r is absent from the hash table, it is assumed to be incorrect. Our algorithm attempts to correct α by changing the nucleotide at either its first or its last position into the other three possible nucleotides (we discuss below how to determine the position). The three variants are searched in the hash table: if only one is present, then it is the correct version of α, assuming that α contains only one error. If multiple variants of α are found in the hash table, the algorithm analyzes the read r to which α belongs. For any correct k-mer β in r, we expect *pools*(β) to equal one of the BAC signatures or the union of up to d BAC signatures. Furthermore, any other correct k-mer γ in r either satisfies *pools*$(\gamma) =$ *pools*(β), or *pools*$(\gamma) \cap$ *pools*(β) is equal to a BAC signature. The second condition takes into account the case when a portion of r originates from the overlapping region between BACs.

Given a read r, let $\mathcal{C} = \alpha_1, \alpha_2, \ldots, \alpha_{|r|-k+1}$ be the set of its k-mers in left to right order, i.e., $\alpha_1 = r[1, k]$, $\alpha_2 = r[2, k+1]$, etc. We define a *correct set* (or *c-set*) $\mathcal{C}'$ as a maximal contiguous subset of $\mathcal{C}$ such that all k-mers in $\mathcal{C}'$ are either repetitive or all share the same BAC signature or the union of up to d BAC signatures.

If r has only one non-empty c-set $\mathcal{C}'$, we can use $\mathcal{C}'$ as a starting point to correct the remaining k-mers. Any c-set $\mathcal{C}'$ contains at least one *border* k-mer α_i such that $\alpha_{i-1} \notin \mathcal{C}'$ when $i > 1$ (or $\alpha_{i+1} \notin \mathcal{C}'$ when $i < |r| - k + 1$). Without loss of generality, assume $\alpha_{i-1} \notin \mathcal{C}'$. The c-set is "interrupted" at position i because $r[i-1]$ is a sequencing error. Thus, we can attempt to change $r[i-1]$ to any of the other three nucleotides, and search the variant k-mer α'_{i-1} in the hash table. If $bacs(\alpha'_{i-1})$ match the shared signature in the c-set $\mathcal{C}'$, we have found the right correction for nucleotide $r[i-1]$ so we add α'_{i-1} to $\mathcal{C}'$ and let it become the new border k-mer. We then repeat the process of extending $\mathcal{C}'$ by correcting the k-mer preceding its new border k-mer (of course, the one correction we just made might actually suffice to correct multiple k-mers, up to k of them, in fact). This iterative process continues until $\mathcal{C}'$ has been extended to encompass the whole read r. Note that when correcting a read from pool p, we also update the hash table: for each k-mer α corrected into α', we increase $poolcount(\alpha', p)$ by one and decrease $poolcount(\alpha, p)$ by one.

If the read contains multiple c-sets with conflicting signatures, we first assume that the first c-set is correct and try to correct the entire read accordingly. If we succeed, we have identified the correct c-set. Otherwise we assume that the second c-set is correct, and so on. If none of the c-sets leads to a successful correction of the entire read, we do not correct the read. Figure 1 illustrates an example with two c-sets.

To deal with an arbitrary number of c-sets, we employ an *iterative deepening depth first search* (IDDFS) heuristic strategy [10]. For each read, IDDFS searches for the correction path with the smallest number of nucleotide changes by starting with a small search depth (maximum number of base changes allowed at the current iteration) and by iteratively increasing the depth until either a solution is found or we reach the maximum number of base changes allowed.

Our proposed k-mer based error correction is sketched as Correction (Algorithm 1). For each read, Correction starts by determining the set of all repetitive and non-repetitive *csets* which will be extended one by one from left to right until all the k-mers of the read are considered. If conflicting c-sets are detected, they are removed from *csets* one at a time when attempting correction. For a given *cset*, we denote by *begin* and *end* its left and right border k-mers respectively. Also, we denote by *bacs* the BACs whose signature is shared by all the k-mers in *cset*. Starting at line 7, we iteratively call the recursive function IDDFSearch with the maximum number of corrections *corr* allowed at the current iteration. If we can correct the entire read with exactly *corr* base changes, we output the corrected read and stop. Otherwise we increment *corr* and repeat the search.

Algorithm 2 sketches the recursive function IDDFSearch. When the entire read is covered by a single c-set or by several non-conflicting repetitive and non-repetitive c-sets, the corrected read is produced and the algorithm stops (lines 2-5). Otherwise, the algorithm tries to extend the current *cset* either to the left (line 9) or to the right (line 13). If *cset* is extended to the left, the algorithm needs to correct the read position *errPos* corresponding to the first nucleotide

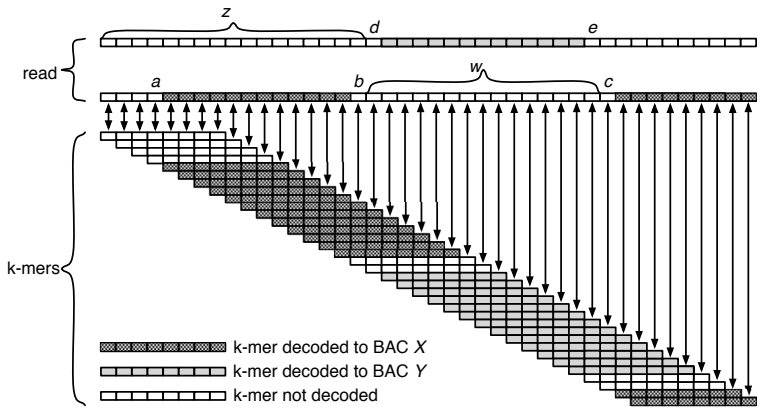

Fig. 1: An illustration of our error-correction strategy. C-sets are colored with dark and light gray. First, we assume that the read belongs to BAC X; in this case, the read positions corresponding to the first nucleotide of the k-mer starting at a, the last nucleotide of the k-mer starting at b, the first nucleotide of the k-mer starting at c and at least one more position in the portion w of the read (because the length of w is between k and $2k-1$) must be corrected. If we assume that the read belongs to BAC Y, the read positions corresponding to the first nucleotide of the k-mer starting at d, the last nucleotide of the k-mer starting at e and at least two more positions in the portion z of the read (because the length of z is between $2k$ an $3k-1$) must be corrected.

of the k-mer starting at $cset.begin - 1$ (lines 10-11). If *cset* is extended to the right, the algorithm needs to correct the read position *errPos* corresponding to the last nucleotide of the k-mer starting at $cset.end + 1$ (lines 14-15). Line 17 calls the function KmerCorrect (Algorithm 3) with the *kmer* which is currently being corrected. KmerCorrect searches for *kmer* in the hash table (line 2) and if found, verifies that the BAC(s) associated with it, *bacs*(*kmer*), agree with the shared BAC signature(s) in *cset* (line 11). If this is the case, *kmer* is assumed to be correct and *read* is updated with the current base change (line 16 or line 21, depending on the direction). The variable *cset* is also extended by one position either to the left or to the right (line 18 or line 23). Line 17 in Algorithm 2 checks the same k-mer as found in *read* without changing it. This is necessary after every successful correction, so that, when extending the current c-set, we know that the k-mers we add contain no further errors. When all k-mers not initially covered by c-sets are checked and all c-sets are extended without detecting further errors, we have corrected the entire read, and we return from the recursive call and produce the solution. If instead we detect additional errors by checking the k-mers not initially covered by c-sets, the algorithm tries all three alternative nucleotides at the erroneous position detected, *errPos* (line 23 in Algorithm 2) and calls KmerCorrect with the modified k-mer. Upon a successful return from KmerCorrect, we recursively call IDDFSearch (line

ALGORITHM 1: CORRECTION (*read-set* R, hashtable H)

```
1  for each read in R do
2      Determine csets, the set of all repetitive and non-repetitive c-sets
3      if conflicts detected then
4          Remove conflicting c-sets from csets one at a time when attempting correction
5      corr ← 0
6      cset ← leftmost element of csets
7      while corr ≤ MaxCorrections do
8          numCorrections ← 0
9          IDDFSEARCH (corr, numCorrections, read, cset)
10         if numCorrections = corr then
11             break
12         corr ← corr + 1
13     output read
14     update hashtable H by decreasing the counts of erroneous k-mers and increasing the counts of corrected
       k-mers
```

31). The nucleotide changes at *errPos* and the calls to the two functions are only made if the total number of corrections so far does not exceed *corr*, the maximum number of corrections allowed at the current iteration (line 21).

4 Experimental Results

We present an experimental evaluation of our method on short reads derived from BACs belonging to a *Minimum Tiling Path* (MTP) of the rice and barley genomes. As its name suggests, an MTP is a set of BACs which cover the genome with minimum redundancy. The construction of an MTP for a given genome requires a physical map but we do not discuss either procedure here (see, e.g., [3, 8] for details).

The use of an MTP allows us to assume that at most two (or three, to account for imperfections) BACs overlap each other. This assumption leads to a 3-decodable pooling design. To achieve $d = 3$ for STD [15], we choose parameters $L = 7$, $\Gamma = 2$ and $q = 13$, so that $d = \lfloor (L-1)/\Gamma \rfloor = 3$, $m = qL = 91$, and $n = q^{\Gamma+1} = 2,197$. With this parameter choices, we can handle up to 2,197 BACs using 91 pools organized in 7 redundant layers of 13 pools each. Since each layer is a partition of the set of pooled BACs, each BAC is pooled in exactly 7 pools (which is its *signature*). In addition, pools are well-balanced, as each pool contains exactly $q^{\Gamma} = 169$ BACs. By the properties of this pooling design, any two BAC signatures share at most $\Gamma = 2$ pools and any three BAC signatures share at most $3\Gamma = 6$ pools [15].

Once the set of MTP BACs has been pooled, we sequenced the resulting pools and used the read decoding algorithm HASHFILTER [8, 9] to assign the reads back to their source BACs, and finally assemble each BAC individually. Error correction is applied prior to read decoding. All experiments were carried out on an Intel Xeon X5660 2.8GHz server with 12 CPU cores and 192GB of RAM.

For our correction algorithm SCRIBLE, we used parameters $k = 31$, $l = 3$, $h = 45$, and a maximum of 4 corrections per read, unless otherwise noted. An

ALGORITHM 2: IDDFSEARCH(corr, numCorrections, read, cset)

```
1  while true do
2      while cset joins its neighbors do
3          cset ← next c-set
4      if exhausted all c-sets then
5          return
6      prevEnd ← end of previous c-set or 0 if none
7      nextBegin ← begin of next c-set or (|r| − |k| + 1) if none
8      if cset.begin − 1 ≠ prevEnd then
9          direction ← left
10         errKmer ← cset.begin − 1
11         errPos ← errKmer
12     else
13         direction ← right
14         errKmer ← cset.end + 1
15         errPos ← errKmer + kmerSize − 1
16     kmer ← k-mer starting at position errKmer in read
17     corrected ← KMERCORRECT (read, cset, direction, kmer, errPos)
18     if not corrected then
19         break
20 numCorrections ← numCorrections + 1
21 if numCorrections > corr then
22     return
23 toTry ← the three alternative nucleotides at errPos
24 for nt ∈ toTry do
25     if direction = left then
26         kmer[1] ← nt
27     else
28         kmer[kmerSize] ← nt
29     corrected ← KMERCORRECT (read, cset, direction, kmer, errPos)
30     if corrected then
31         IDDFSEARCH (corr, numCorrections, read, cset)
32         if numCorrections = corr then
33             break
```

analysis of other choices of k is carried out later in Figure 2. The other methods corrected all the reads in the 91 pools together (not pool-by-pool).

4.1 Results on Synthetic Reads for the Rice Genome

We tested our error correction method on short reads from the rice genome (*Oryza sativa*) which is a fully sequenced 390Mb genome. We started from an MTP of 3,827 BACs selected from a real physical map library of 22,474 BACs. The average BAC length in the MTP was ≈ 150kB. Overall, the BACs in the MTP spanned 91% of the rice genome. We pooled a subset of 2,197 of these BACs into 91 pools according to the parameters defined above.

The resulting BAC pools were "sequenced" *in silico* using SIMSEQ, which is a high-quality short read simulator used to generate the synthetic data for Assemblathon [2]. SIMSEQ uses error profiles derived from real Illumina data to inject realistic substitution errors. We used SIMSEQ to generate ≈1M paired-end reads per pool with a read length of 100 bases and an average insert size of 300 bases. A total of ≈ 200M bases gave an expected ≈ 8× sequencing depth for a BAC in a pool. Since each BAC is present in 7 pools, this is an expected ≈ 56× combined coverage before decoding. The average error distribution for the first read in a paired-end read is: 48.42% reads with no error, 34.82% with 1 error,

ALGORITHM 3: KMERCORRECT(read, cset, direction, kmer, errPos)

```
1  correct ← false
2  if kmer ∉ hashtable then
3      return false
4  pools ← pools(kmer) // from hashtable
5  if SIZE (pools) > h then
6      correct ← true
7  if SIZE (pools) ≥ l and SIZE (pools) ≤ h then
8      bacs ← bacs(kmer)
9      if SIZE (bacs) = 0 then
10         return false
11     if bacs agree with cset.bacs then
12         correct ← true
13 if correct then
14     if direction = left then
15         // update read with base change
16         read[errPos] ← kmer[1]
17         // extend cset left
18         cset.begin ← cset.begin − 1;
19     else
20         // update read with base change
21         read[errPos] ← kmer[kmerSize]
22         // extend cset right
23         cset.end ← cset.end + 1;
24     return true
25 return false
```

12.96% with 2 errors, 3.14% with 3 errors, 0.57% with 4 errors, 0.08% with 5 errors and 0.01% with 6 errors. For the second read, the error distribution is: 32.85% no errors, 35.71% with 1 error, 20.75 with 2 errors, 7.91% with 3 errors, 2.20% with 4 errors, 0.48% with 5 errors, 0.09% with 6 errors and 0.01% with 7 errors.

We compared the performance of SCRIBLE against the state-of-the-art error-correction method RACER [5]. The authors of RACER performed extensive experimental evaluations and determined that RACER is superior in all aspects (performance, space, and execution time) to HiTEC, SHREC, REPTILE, QUAKE, and CORAL. We also compare SCRIBLE against SGA [13], as it was not evaluated in [5] but we had evidence it provides good error correction. SGA provides two correction methods, read overlap based and k-mer based, but we only consider the latter.

	0 mm	*1 mm*	*2 mm*	*3 mm*	*execution time* (min)	*space* (GB)
Original reads	14.56%	55.98%	85.43%	96.36%	N/A	N/A
SGA	62.68%	69.58%	72.31%	73.23%	228.73 + 0.87	**6**
RACER	87.10%	93.25%	96.00%	97.10%	**19.76 + 0.11**	120
SCRIBLE	**95.00%**	**97.77%**	**98.70%**	**99.11%**	600 + 2.76	50

Table 1: Percentage of error-corrected reads that map to the rice genome for increasing number of allowed mismatches; execution times (preprocessing + correction) are *per* 1M reads; boldface values highlight the best result in each column

Table 1 reports correction accuracy as well as time and space used by each method. As it was done in [5], correction accuracy is determined by mapping the corrected reads to the reference using Bowtie [7] in stringent mode (paired-end, end-to-end alignment). Columns 2, 3, 4, and 5 report the fraction of reads mapped when 0, 1, 2, and 3 mismatches were allowed, respectively. The second row of Table 1 reports the mapping results for the original set of uncorrected reads. The last two columns report time (pre-processing + correction) and memory requirements for each method. The pre-processing time is method dependent: in our method it is the time to build the hash table (which currently is not multi-threaded), for Sga it is the time to build the FM-index, and, for Racer it is the time to compute witnesses and counters. For Scrible and Sga we chose a $k = 31$ because (1) our method performs better with larger k-mer sizes and (2) this is the default choice for Sga. Racer determines the best k-mer size from the data.

Table 1 shows that Scrible is by far the most accurate. The difference between Scrible and state-of-the-art Racer on a small number of allowed mismatches (which is what matters here) is very significant. In this application domain, accuracy is much more important than time- and space-complexity (as long as time and space are reasonable). The bottleneck in our method is the preprocessing stage (building the hash table), which is currently single-threaded. Figure 2 shows correction accuracy (percentage of mapped reads) for Sga and our method for other choices of k-mer size. As the k-mer size increases, our method corrects more reads, whereas the opposite is true for Sga.

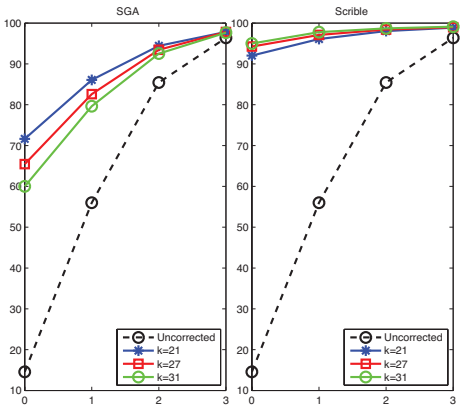

Fig. 2: Correction accuracy for different k-mer sizes for SGA and Scrible (x-axis: number allowed mismatches for mapping, y-axis: percentage of reads mapped)

An alternative way to assess the performance of error correction is to assemble the corrected reads, after decoding the reads with HashFilter. For this purpose, we used Velvet [17] with hash parameter $k = 49$ (chosen based on our extensive experience on this dataset). Table 2 reports the averaged results of 2,197 BACs in each of the four datasets. The table shows average coverage, the percentage of reads used by Velvet in the assembly, the number of contigs with length of at least 100 bases, the n50 value, the ratio of the sum of all contigs sizes over their BAC length, and the coverage of the BAC sequence by the assembly. The slightly different coverages depends on the ability of HashFilter to decode reads for each dataset. Despite the fact that Velvet employs its own error-correction algorithm, assemblies corrected by Scrible have better statistics (except for the number of contigs) than those obtained using the outputs of the other tools.

	coverage	*reads used*	*# of contigs*	*n50* (bp)	*sum/size*	*BAC coverage*
Original reads	88.66x	92.40%	52.27	35,846	89.05%	82.22%
SGA	88.51x	97.46%	51.90	36,603	90.33%	82.88%
RACER	85.48x	98.22%	**46.29**	29,770	88.59%	81.82%
SCRIBLE	**88.76x**	**98.43%**	53.92	**39,422**	**91.50%**	**82.97%**

Table 2: Assembly statistics for rice BACs using VELVET on datasets of reads uncorrected and corrected via various methods; all values are an average of 2,197 BACs; boldface values highlight the best result in each column

		0 mm	*1 mm*	*2 mm*	*3 mm*
Hv4	HASHFILTER	88.41%	91.66%	92.54%	93.25%
Hv4	SCRIBLE	**92.04%**	**93.67%**	**94.62%**	**95.23%**
Hv5	HASHFILTER	87.37%	90.45%	91.61%	92.63%
Hv5	SCRIBLE	**93.26%**	**94.66%**	**95.56%**	**96.14%**
Hv6	HASHFILTER	84.72%	88.46%	90.05%	91.38%
Hv6	SCRIBLE	**92.13%**	**93.48%**	**94.23%**	**94.73%**

Table 3: Percentage of real barley reads that map to 454-based high-quality BAC assemblies for increasing number of allowed mismatches; boldface values highlight the best result for each Hv set

4.2 Results on Real Reads from the Barley Genome

We also tested our method on real sequencing data from the barley (*Hordeum vulgare*) genome, which is about 5,300 Mb and not yet fully sequenced [14]. We started from an MTP of about 15,000 BACs selected from a subset of nearly 84,000 gene-enriched BACs for barley (see [8, 9] for more details). We divided the set of MTP BACs into seven sets of 2,197 BACs and pooled each set using the STD parameters previously defined. We assessed the performance of error correction on three of these data sets, namely Hv4, Hv5 and Hv6 (with an average BAC length of about 116 kb). Each of the 91 pools in Hv4, Hv5 and Hv6 were sequenced on a flowcell the Illumina HiSeq2000 by multiplexing 13 pools on each lane. After each sample was demultiplexed, we quality-trimmed and cleaned the reads of spurious sequencing adapters and vectors. We ended up with high quality reads of about 87–89 bases on average. The number of reads in a pool ranged from 3.37M to 16.79M (total of 706M) in Hv4, from 2.32M to 15.65M (total of 500M) in Hv5 and from 1.3M to 6.33M (total of 280M) in Hv6.

In this experiment, we compared SCRIBLE's ability to decode reads against HASHFILTER's. In other words, while HASHFILTER decodes erroneous reads, SCRIBLE corrects and decode the reads. Table 3 presents the results of this comparison. Once the reads were decoded to individual BACs by HASHFILTER and SCRIBLE, we mapped them using BOWTIE [7] in stringent mode (paired-end, end-to-end alignment) to the subset of BAC for which a high-quality 454-based

assembly was available (151 high-quality assemblies for HV4, 141 for HV5 and 121 for HV6). Table 3 shows that SCRIBLE achieves significantly higher mapping percentages than HASHFILTER. Higher mapping percentages indicate more error-free reads, and higher decoding accuracy.

References

1. DUMA, D., ET AL. Accurate decoding of pooled sequenced data using compressed sensing. In *Proceedings of WABI* (2013), pp. 70–84.
2. EARL, D., ET AL. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research 21*, 12 (Dec. 2011), 2224–2241.
3. ENGLER, F., ET AL. Locating sequence on FPC maps and selecting a minimal tiling path. *Genome Research 13*, 9 (2003), 2152–2163.
4. ILIE, L., ET AL. HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics 27*, 3 (2011), 295–302.
5. ILIE, L., AND MOLNAR, M. RACER: Rapid and accurate correction of errors in reads. *Bioinformatics* (2013).
6. KELLEY, D. R., ET AL. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol 11*, 11 (2010), R116.
7. LANGMEAD, B., ET AL. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology 10*, 3 (2009), R25.
8. LONARDI, S., ET AL. Combinatorial pooling enables selective sequencing of the barley gene space. *PLoS Comput Biol 9*, 4 (04 2013), e1003010.
9. LONARDI, S., ET AL. When less is more: "slicing" sequencing data improves read decoding accuracy and *De Novo* assembly quality. *Bioinformatics* (2015), to appear, also in BioRxiv, http://dx.doi.org/10.1101/013425.
10. RUSSELL, S., NORVIG, P., ET AL. *Artificial Intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996, ch. 3.
11. SALMELA, L., AND SCHRODER, J. Correcting errors in short reads by multiple alignments. *Bioinformatics 27*, 11 (2011), 1455–1461.
12. SCHRODER, J., ET AL. SHREC: a short-read error correction method. *Bioinformatics 25* (2009), 2157–2163.
13. SIMPSON, J. T., AND DURBIN, R. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research 22*, 3 (2012), 549–556.
14. THE INTERNATIONAL BARLEY GENOME SEQUENCING CONSORTIUM. A physical, genetic and functional sequence assembly of the barley genome. *Nature advance online publication* (10 2012), in press.
15. THIERRY-MIEG, N. A new pooling strategy for high-throughput screening: the shifted transversal design. *BMC Bioinformatics 7*, 28 (2006).
16. YANG, X., ET AL. Reptile: representative tiling for short read error correction. *Bioinformatics 26*, 20 (2010), 2526–2533.
17. ZERBINO, D., AND BIRNEY, E. Velvet: Algorithms for *de novo* short read assembly using de Bruijn graphs. *Genome Research 8*, 5 (2008), 821–9.