MapReduce for accurate error correction of next-generation sequencing data
Liang Zhao1,2, Qingfeng Chen1, Wencui Li2, Peng Jiang1, Limsoon Wong3, ∗, Jinyan Li4, ∗
1School of Computing and Electronic Information, Guangxi University, Nanning 530004, China.
2Taihe Hospital, Hubei University of Medicine, Hubei 442000, China.
3School of Computing, National University of Singapore, Singapore 117417, Singapore.
4 Advanced Analytics Institute and Centre for Health Technologies, University of Technology Sydney, PO Box 123, Broadway, NSW 2007, Australia.
Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT
Motivation: Next-generation sequencing platforms have produced huge amounts of sequence data. This is revolutionizing every aspect of genetic and genomic research. However, these sequence datasets contain quite a number of machine-induced errors—e.g., errors due to substitution can be as high as 2.5%. Existing error-correction methods are still far from perfect. In fact, more errors are sometimes introduced than correct corrections, especially by the prevalent k-mer based methods. The existing methods have also made limited exploitation of on-demand cloud computing.

Results: We introduce an error-correction method named MEC, which uses a two-layered MapReduce technique to achieve high correction performance. In the first layer, all the input sequences are mapped to groups to identify candidate erroneous bases in parallel. In the second layer, the erroneous bases at the same position are linked together from all the groups for making statistically reliable corrections. Experiments on real and simulated datasets show that our method outperforms existing methods remarkably. Its per-position error rate is consistently the lowest, and the correction gain is always the highest.

Availability The source code is available at bioinformatics.gxu.edu.cn/ngs/mec.
Contact: wongls@comp.nus.edu.sg, jinyan.li@uts.edu.au

1 INTRODUCTION
The high throughput and low cost of next-generation sequencing (NGS) have turned many previously-difficult problems into easily-dissected problems, e.g., the studies on genome-wide association between single-nucleotide polymorphisms and diseases (The 1000 Genomes Project Consortium, 2010). However, NGS platforms (Mardis, 2013) are error prone, with poor single-pass sequencing read error rates (Molnar and Ilie, 2014).

Poor error rates of NGS data. NGS machines produce raw data called reads, which are short stretches of nucleotides representing some small segments of the genome being sequenced. Two types of errors widely reside in these reads (Yang et al., 2013): substitutions and indels (insertions and deletions). Figure 1(a) shows an example, where substitutions are coloured in orange and indels are coloured in light blue or green for insertion into or deletion from the reference sequence (the long sequence in black). For Illumina machines, the majority errors are substitutions, with a rate from 0.5% to 2.5% (Kelley et al., 2010); while the indel error rate is negligible. On the other hand, for PacBio technology, the major error type is indel. The error rate can be as high as 30% in some extreme cases (Mardis, 2013). As Illumina machines dominate the market, we focus on substitution-error correction in this work.

Complexity introduced by the errors. A major step in the analysis of NGS data is to assemble these short reads into the true long genome sequence as accurately as possible. Current genome-assembly methods are mainly based on de Bruijn graph (Compeau et al., 2011). A k-dimensional de Bruijn graph G of a sequence S is a directed graph representing the overlaps of all the substrings of length k (k-mers) of S. The vertices are the k-mers, and the edges are the overlaps between these k-mers. Two k-mers s1...k and s′1...k are said to overlap if s′2...k = s1...k−1 or s′2...k = s1...k−1. In the former case, there is an edge pointing from s1 to s1′, while in the latter the edge is pointing from s1 to s1′. Sequence errors can complicate a de Bruijn graph and the derivation of the true genome sequence from it. Figure 1(b) shows the de Bruijn graph constructed from the error-containing reads in Figure 1(a), for k = 4. For comparison, Figure 1(c) shows the de Bruijn graph constructed from the reads after error correction. The error-containing de Bruijn graph is much more complicated than the error-free graph.

Limits of existing error-correction methods. Error correction for NGS data has attracted intensive research. Many methods tackle the problem from an “algorithm and data structure” point of view. Yang et al. (2013) further categorized these methods based on whether they use (i) k-mers, (ii) suffix array/tree, or (iii) multiple-sequence alignment. The key idea of the k-mers methods is to decompose the raw reads into k-mers, then correct untrusted k-mers to the nearest trusted ones. Representative methods include Quake (Kelley et al., 2010), Reptile (Yang et al., 2010), Hammer (Medvedev et al., 2011), BLESS (Heo et al.,...
2014), DecGPU (Liu et al., 2011), Euler (Pevzner et al., 2001), Musket (Liu et al., 2013), RACER (Ilie and Molnar, 2013), BFC (Li, 2015), and ACE (Sheikhzadeh and de Ridder, 2015). These methods only differ in how they count the frequency of k-mers, how they determine the trusted k-mers, and most critically what they use as the value of k and what they use as the threshold of k-mer. The suffix array/tree methods, e.g. SHREC (Schröder et al., 2009), HITEC (Ilie et al., 2011) and Hybrid-SHREC (Salmela, 2010), avoid these issues to some extent. By these methods, the value of k is flexible and only the threshold of the coverage matters in determining the erroneous bases of the reads. This idea partially mitigates the impact of k on performance. However, the huge memory requirement by these methods restricts their applicability a lot. The methods based on the idea of multiple-sequence alignment, e.g. Coral (Salmela and Schröder, 2011), ECHO (Kao et al., 2011) and Karect (Allam et al., 2015), make the k parameter almost free from consideration. These methods use a k-mer as a seed to align reads that share the same k-mer, to determine the consensus sequence of the aligned sequences. Those bases of reads differing from the consensus sequence are corrected. The major bottleneck of these methods is the multiple-sequence alignment, which is too time- and memory-consuming to handle large datasets.

**Our MapReduce approach.** We introduce a two-layered MapReduce-based error corrector (MEC) for correcting substitution errors contained in Illumina NGS datasets. In the first layer, all possible k-mers of the input data are used as keys to map raw reads into groups such that all the reads in each group contain the same k-mer. The prospective erroneous positions are then identified by conducting multiple alignments of the reads within each group. In the second layer, the erroneous bases are identified and corrected based on the aligned prospective erroneous positions from all the groups.

The main computational steps of MEC—mapping raw reads into groups, identifying erroneous bases, and correcting erroneous bases—can be all carried out in parallel. Another point of novelty of MEC is that it uses the complete list of reads that cover an erroneous position. In contrast, many existing k-mer methods are unable to use all the reads that cover an erroneous position to correct the erroneous bases in the reads at this position.

2. **MEC: FRAMEWORK**

In recent years, MapReduce (Dean and Ghemawat, 2008) has become a popular model for parallel and distributed processing of large datasets on large computing clusters. The model contains two procedures: “Map” and “Reduce”. “Map” accounts for filtering and sorting operations. “Reduce” is a summary operation on the results of Map. Reduce is typically commutative and associative—i.e. its outcome is independent of the order of arrival of the results of Map—enabling Map to be massively and easily data parallelized while providing correctness guarantee. We take the essential idea of MapReduce, but not the programming model itself, to develop the two-layered MapReduce framework; cf. Figure 2.

In the first layer, we map the set R of raw reads into a number of groups. The groups are in one-to-one correspondence with the set of all k-mers occurring in the raw reads. Each k-mer occurrence is assigned to the group associated with that k-mer (and the corresponding read of that k-mer occurrence is also mapped to that group). More precisely, let \( \kappa \) be a k-mer and \( r \) be the \( j \)th read in \( R \). An occurrence of \( \kappa \) at the \( i \)th position of \( r \) is denoted as \((\kappa, j, i)\). We consider single-end reads here to simplify exposition; paired-end reads are discussed in the next section. The group corresponding to \( \kappa \) is denoted as \( G_\kappa \), and \( \kappa \) is called the key of the group. The set of k-mer occurrences assigned to \( G_\kappa \) is denoted by \( G(\kappa) \). And for every \( \kappa, i, j, \) and \( i = j \) is the case that \((\kappa, j, i) \in G(\kappa) \) and only if \( \kappa \) occurs at the \( i \)th position of \( r_j \in R \). As an example, suppose the coloured short bars (blue, purple or green) in Figure 2 are the distinct keys (i.e. k-mers). Then all the long bars (raw reads) are mapped into subgroups by checking whether they contain the corresponding keys.

By definition—in particular, the “if and only if” condition above—every raw read is mapped to multiple groups (one group for each distinct k-mer that occurs in the read), and the read can be mapped to the same group multiple times (when the k-mer corresponding to that group occurs in multiple positions of that read). This guarantees that no raw read is missed during error correction. Existing error-correction methods generally do not have such a completeness-of-coverage guarantee.

After mapping the reads into groups, the reads within each group are aligned into one, or very likely multiple, alignments to identify erroneous bases (i.e. the substitution errors in a read).

See alignment1, alignment2 and alignment3 of the raw reads in Figure 2. For single-end reads, a simple ungapped alignment is
Accurate error correction for NGS data

used: For each pair of occurrences \((\kappa, j, i)\) and \((\kappa, j', i')\) in \(G(\kappa)\),
\(j \neq j'\), an alignment of \(r_j\) to \(r_{j'}\) is generated so that position
\(h + i \in r_j\) is aligned to position \(h + i'\) in \(r_{j'}\), for all \(h\), provided
the resulting alignment has a high sequence identity. Subsequently,
prospective erroneous bases as well as their candidate corrections
are summarized as an input for the second layer of MapReduce
processing for error correction.

In the second layer, the candidate erroneous bases distributed
over many groups are mapped together. The keys are the aligned
positions and the values are the bases (contained in the reads)
covering the corresponding positions. By this technique,
the supporting reads used to correct erroneous bases are much more
complete than those used by the existing approaches.

At each position, each base with small support (viz. the number
of reads covering that position) is considered a prospective erroneous
base, and we draw direct edges from the corrections (i.e. bases
with large supports) to the erroneous base (a diagram is shown in the
lower panel of Figure 2). The graphs constructed in this way are
mostly isolated (e.g. graph3 in Figure 2) with exceptions caused by
repeats (e.g., graph2 in Figure 2) and multiple errors at the same
position (e.g. graph1 in Figure 2). The erroneous bases are then
corrected by a log-likelihood ratio statistics.

This two-layered MapReduce framework has the following
optimality guarantee. Let \(\kappa_y\) be the k-mer beginning at an arbitrary
position \(x\) in the genome. According to the given sequencing depth
\(d\) and the per-base error probability \(e\), \(\kappa_y\) is expected to be correctly
read \(d \ast (1 - e)^{\kappa_y}\) times. Now let \(y\) be a position in the genome
that is spanned by a read \(r_j \in R\) and \(r_j\) is incorrect for this position.
Let \(\kappa_y\) be the k-mer immediately after \(y\) in the genome. Then
\(\kappa_y\) is expected to be correctly read \(d \ast (1 - e)^\kappa\) times. Since the read
length is \(l\) and \(k < l\), we expect \((l - 1)/l\) of these \(d \ast (1 - e)^\kappa\)
reads to span position \(y\); this is because among reads that cover position
\(y + 1\), only those starting at position \(y + 1\) cannot cover position
\(y\). This means that for any erroneous base in a read (e.g. the base in
\(r_j\) corresponding to position \(y\) in the genome sequence), we expect
there is an alignment comprising of \(n = ((l - 1)/l) \ast d \ast (1 - e)^\kappa\)
reads that span the position in the genome corresponding to this
base. And in this alignment, \(1 - e\) of the \(n\) bases corresponding
to this position are correct. In other words, we expect the erroneous
base to be in an alignment where there are \(((l - 1)/l) \ast d \ast (1 - e)^{k+1}\)
correct bases that can be used to build a consensus to correct the
error base. For example, suppose \(d = 30\), \(l = 101\), \(k = 23\),
and \(e = 0.01\), then we expect 23 correct bases that are aligned
to each erroneous base. Thus every erroneous base in every read can
be corrected with high probability by our method. In fact, the alignment
of \(((l - 1)/l) \ast d \ast (1 - e)^{k+1}\) correct bases to each erroneous base is
the maximum achievable guarantee. That is, no method requiring
at least one conserved k-mer can guarantee an alignment which can
be expected to contain more than \(((l - 1)/l) \ast d \ast (1 - e)^{k+1}\) correct
bases that are aligned to each erroneous base. In the supplementary
notes (particularly Figure S2), the incomplete coverage taken by
the existing approaches is illustrated in detail.

3 MEC: DETAILS

NGS raw reads are paired-end reads in most cases. So our implementation
is for paired-end reads. The details are described below. There are two main
steps in each layer of our two-layered MapReduce framework. So there are
four main steps in total: (1) map the input reads into small groups associated
with the set of k-mers occurring in the reads; (2) align the reads in every
group and identify the prospective erroneous bases; (3) map the erroneous
bases to candidate corrections; and (4) replace the erroneous bases with the
suggested correction candidates.

Step 1: Mapping paired-end reads into groups

A paired-end read consists of two components, one component is a read
of the up-stream end of the reference sequence, the other is a read at the
down-stream end. The nucleotides between the two ends are not sequenced.
The distance between the two components is called insert size, which varies
depending on the library construction and technology used.

Let \(r_j\) denote the \(j\)th paired-end read in the input set \(R\). Given a k-mer
\(\kappa\), we denote an occurrence of \(\kappa\) starting at the \(i\)th position of the read at the
up-stream end of \(r_j\) by \((\kappa, j, i, 0)\). Similarly we denote an occurrence of \(\kappa\)
starting at the \(i\)th position of the down-stream end of \(r_j\) by \((\kappa, j', i, 1)\).
A hash function is used to map a k-mer \(\kappa\) to a group \(G_{\kappa}\), which corresponds
to a node in the computing cluster, thus sending all tuples \((\kappa, j, i, t)\)
that denote occurrences of \(\kappa\) in the paired-end reads to the group \(G_{\kappa}\).

For a paired-end read, a window of size \(k\) is used to slide along the read
to generate all the distinct k-mers contained in the read. Note that a paired-end
read is mapped into \(n\) groups if it contains \(n\) distinct k-mers. And it
can also be mapped to a group \(G_{\kappa}\) \(m\) times if there are \(m\) occurrences
of \(\kappa\) in that paired-end read. Similarly, we denote the set of tuples in group
\(G_{\kappa}\) by \(G_{\kappa}\). We also say a paired-end read \(r_j\) is mapped to group \(G_{\kappa}\)
if \((\kappa, j, i, t) \in G_{\kappa}\) for some \(i\) and \(t\).

Some groups \(G_{\kappa}\) are much larger than other groups. In such a case, the
k-mer \(\kappa\) is likely from repeat regions or has low sequence complexity. We
discard such groups. The rationale and the detailed instructions are shown in
the supplementary notes.

Step 2: Aligning paired-end reads and identifying errors

For this step, we need the concept of an “anchored” alignment. An anchored
alignment is made up of an anchor occurrence \((\kappa, j, i, t)\) of some k-mer \(\kappa\)
and a list of occurrences \((\kappa, j_1, i_1, t), ..., (\kappa, j_n, i_n, t)\) of the k-mer \(\kappa\), along
with pairwise alignments \((A_1, ..., A_n)\) of paired-end reads \(r_{j_1}, ..., r_{j_n}\) with
\(r_j\), such that the occurrences \((\kappa, j_1, i_1, t), ..., (\kappa, j_n, i_n, t)\) are aligned to the
occurrence \((\kappa, j, i, t)\) of the k-mer \(\kappa\). Note that \(j \notin \{j_1, ..., j_n\}\); i.e. we do not
align two occurrences of \(\kappa\) in the same read. Such an anchor alignment induces
a multiple alignment \(A\) of the paired-end reads \(r_{j_1}, r_{j_2}, ..., r_{j_n}\) by a simple stacking up
of the pairwise alignments \((A_1, ..., A_n)\). Thus, a position
x in \(r_j\) is aligned to a position \(y\) in \(r_{j'}\) in \(A\) if and only if the position \(x\) in \(r_j\) is aligned to the position \(y\) in \(r_{j'}\) in \(A_\kappa\), for any \(x, y,\) and \(g\). A set of positions \(\{x \in r_j, x_1 \in r_{j_1}, ..., x_n \in r_{j_n}\}\) is called a "column" of \(A\) if \(x_1, ..., x_n\) are aligned to \(x\) in \(A\). Also, we call \(r_j\) the anchor paired-end read of the anchored alignment and \(r_{j_1}, ..., r_{j_n}\) the member paired-end reads of the anchored alignment.

Given a group \(G_\kappa\), one or more anchored alignments of the paired-end reads mapped to the group are generated based on the occurrences of the k-mer \(\kappa\). The procedure is as follows, after initializing the set of anchored alignments to empty:

1. Randomly select a tuple \((\kappa, j, i, \ell)\) in \(G(\kappa)\) that has not yet been assigned to any anchored alignment.
2. For each anchor \((\kappa, j', i', \ell')\) of the current set of anchored alignments, where \(\ell = \ell'\), apply the penalty-aware pairwise alignment algorithm (described in the supplementary notes) to generate the pairwise alignment of \(r_j\) and \(r_{j'}\) that respects \((\kappa, j, i, \ell)\). Let \((\kappa, j', i', \ell')\) be the one that yields the pairwise alignment with the highest score. If the identity between \(r_j\) and \(r_{j'}\) according to this pairwise alignment is at least 95% of the alignment length, \((\kappa, j, i, \ell)\) along with this pairwise alignment is assigned to the anchored alignment anchored by \((\kappa, j', i', \ell')\). Otherwise, \((\kappa, j, i, \ell)\) is made an anchor of a new (empty) anchored alignment.
3. Repeat the two steps above until all tuples in \(G(\kappa)\) have been assigned to some anchored alignments.
4. Some cleaning up of the anchor alignments is performed if there is more than one occurrence of the k-mer \(\kappa\) in any paired-end read \(r_j\). In such a case, only the occurrence associated with the highest pairwise alignment score among all occurrences of \(\kappa\) in \(r_j\) is kept.
5. The prospective erroneous positions are identified from the multiple alignments induced by the anchor alignments. This is done by looking down a column in an induced multiple alignment; if not all the bases have the same value, then the position corresponding to this column is regarded as a prospective erroneous position and the bases of this position together; it is also used for distributing the tuples for this prospective erroneous bases is identified as the erroneous bases. Considering the subclones of a cancer genome might result in multiple correct references at a specific position, we only correct the candidate bases having frequency not larger than three. This restriction results in the same, even slightly better, correction accuracy. The rationale is explained in the supplementary notes.

A mapping between the erroneous bases and the reference bases is then created (cf. the graphs in Figure 2). As each prospective erroneous position is independent and only reads covering that position are involved, the mapping of erroneous bases and the reference bases can be handled separately in parallel for each position. This mapping gives the candidate corrections for the erroneous bases, whereby the values of the reference bases are proposed as the correct values for the respective erroneous bases.

**Step 4: Correcting errors**

Given a candidate correction that proposes to correct an erroneous base (having value \(x\)) by a reference value \(x_0\) at a prospective erroneous position, a log likelihood ratio is calculated and the correction is made only when the ratio is very small.

Let \(j\) range over all the bases (both erroneous and reference bases) at the given prospective erroneous position. Let \(j\) denote the value of the base \(j\). Let \(I(true) = 1\) and \(I(false) = 0\) be the indicator function. Let \(p_j\) be the probability that the base \(j\) is called correctly by the sequencing machine—\(p_j\) can be calculated based on the Phred score \(q\) of the base as \(p_j = 1 - 10^{-q/10}\). Then the log likelihood ratio is calculated as:

\[
L_{x/x_0} = \log \frac{\prod_{j \neq x_0} I(j = x_0) \cdot p_j + I(j \neq x_0) \cdot (1 - p_j)/3}{\prod_{j \neq x} I(j = x) \cdot p_j + I(j \neq x) \cdot (1 - p_j)/3}
\]

Take Figure 2 as an example. The erroneous bases "0" and "1" in graph1 will be replaced by the reference value. The base "2" in graph2, but not "0" and "1", will be replaced by the reference value. And the base "0" in graph3 will be replaced by the reference value.

**4 PERFORMANCE EVALUATION**

Four real datasets and four simulated datasets are collected for evaluating the performance of MEC. The simulated datasets are used because of three reasons. Firstly, the absolute ground truth of a genome is usually unknown. Secondly, some very complicated regions in a genome cannot be sequenced, or the sequenced data cannot be matched correctly to the reference. Thirdly, the ground truth of simulated genome sequences is known and can be controlled. As the simulated data not only can mimic the real sequencing data but also can track where the sequences come from, a solid comparison between the corrected sequences and their raw sequences can be conducted. An error model-aware simulator, GemSim (McElroy et al., 2012), is employed to produce the simulated NGS data. Four simulated datasets are constructed in this study from three genomes: Escherichia coli (E.coli), Saccharomyces cerevisiae (S.cerevisiae) and chromosome 22 of Homo sapiens (H.sapiens 22). These datasets have different levels of complexity. Some other details of these datasets are listed in Table 1. The four real datasets have been used in the renowned
NGS-data assembly project, GAGE (Salzberg et al., 2011). They are named Staphylococcus aureus (S. aureus), Rhodobacter sphaeroides (R. sphaeroides), chromosome 14 of Homo sapiens (H. sapiens 14) and Bombus impatiens (B. impatiens). A pre-process was applied before our evaluation procedure: (1) removed sequences containing undecided base(s); (2) mapped all the remaining sequences to the reference genome using BWA (Li and Durbin, 2009) and excluded those which are unaligned or aligned to multiple places.

4.1 Evaluation metrics

The performance of an error-correction method is assessed using gain (gain), recall (recall), and precision (prec). The measure gain is defined as \((TP - FP)/(TP + FN)\), recall is defined as \(TP/(TP + FN)\), and precision is defined as \(TP/(TP + FP)\), where \(TP\) is the number of errors that are correctly corrected, \(FN\) is the number of errors that are not corrected and \(FP\) is the number of errors introduced. These definitions have been widely used by the literature error-correction methods (Yang et al., 2013), and the most important performance measure is gain.

Per-base error rate (pber) and per-position error rate (pper) are calculated as additional measurements. Per-base error rate is defined as \(N_e/N_s\), and per-position error rate is defined as \(N_{ep}/N_p\), where \(N_e\) is the number of erroneous bases, \(N_s\) is the number of all bases, \(N_{ep}\) is the number of erroneous positions (i.e. the number of positions that are covered by at least one erroneous base), and \(N_p\) is the number of all positions. \(N_e\) is equal to \(2 \times N \ast L\) and \(N_p\) is equal to the length of the genome. \(N\) and \(L\) are the number of paired-end reads and read length, respectively.

4.2 Error-correction quality

The results of our experiments are reported in Table 2. Our method MEC has a remarkable correction performance for substitutional sequencing errors. The average per-base error rate for the eight datasets before correction is 1.28%, which is reduced to 0.17% after correction by MEC, making a 8-fold decrease. The per-position error rate is also reduced significantly after error correction by MEC. E.g., this error rate is 64.7% for dataset D4, which is reduced to 1.6% after error correction by MEC.

MEC’s performance is better on datasets R1, R2, D1, D2, D3, and D4 than on the datasets R3 and R4. This may be attributed to the complexities of the genomes: the genomes of Homo sapiens and Bombus impatiens are much more complicated than the genomes of Escherichia coli, Saccharomyces cerevisiae and Rhodobacter sphaeroides, particularly at the rich repeat regions in the genome of Homo sapiens (International Human Genome Sequencing Consortium, 2004). It is also noteworthy that MEC’s performance on the simulated datasets D1 to D4 are much better than that on the real datasets R1 to R4, indicating that the real datasets are more complicated than the simulated data, even when only the mapped sequences are considered.

Five representative error-correction methods are included to benchmark the performance of MEC. These representative methods are Coral (Salmea and Schröder, 2011), Racer (Ilie and Molnar, 2013), BLESS (Heo et al., 2014), BFC (Li, 2015) and SGA (Simpson and Durbin, 2012). They are chosen as they use different strategies, and all of them have claimed to perform very well. Coral is a multiple sequence alignment-based error corrector. Racer uses \(k\)-mers to correct errors. BLESS corrects errors using Bloom filter. BFC uses the blocked Bloom filter. SGA employs overlap graphs to correct errors. The default parameters for each method are used.

As shown in Table 2, the performance of these five existing error-correction methods on datasets D1 and D2 are excellent and very close. Our method MEC is only slightly better than them. On datasets R1 to R4, D3 and D4, the performance of these five methods varies significantly. E.g., on the simulated datasets D3 and D4, Racer performs much worse than other approaches. In particular, the gains of Racer on these two datasets are even negative, indicating more errors are introduced than corrected. However, on the four real datasets R1 to R4, Racer produces competitive results. For our method MEC, its performance is superior to all these five methods in terms of gain on all the datasets, especially on the more complicated datasets. The superior performance of MEC is attributed to the following reasons. Firstly, as explained in Section 2, all the reads are guaranteed to be used; thus no position is missed out. Secondly, also explained in Section 2, more reads originating from the same position are included; thus more trustworthy results can be obtained. In addition, the paired-end information is considered which, to a certain extent, can reduce the false alignment of repeat reads coming from different regions. We also attempted to use the suffix tree/array-based error corrector, HiTEC, to test on the eight datasets. However, segmentation faults occurred when the program was executed.

### Table 1. Datasets used for evaluating the performance of error-correction methods.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Genome name</th>
<th>Genome size (mbp)†</th>
<th>Read length (bp)‡</th>
<th>Coverage</th>
<th>Number of paired-end reads</th>
<th>Per base error rate (%)</th>
<th>Per position error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>S. aureus</td>
<td>2.8</td>
<td>101</td>
<td>46.3×</td>
<td>1,294,104</td>
<td>1.17</td>
<td>28.1</td>
</tr>
<tr>
<td>R2</td>
<td>R. sphaeroides</td>
<td>4.6</td>
<td>101</td>
<td>33.6×</td>
<td>766,646</td>
<td>1.28</td>
<td>30.0</td>
</tr>
<tr>
<td>R3</td>
<td>H. sapiens 14</td>
<td>88.3*</td>
<td>101</td>
<td>38.3×</td>
<td>16,757,120</td>
<td>0.86</td>
<td>22.5</td>
</tr>
<tr>
<td>R4</td>
<td>B. impatiens</td>
<td>249.2</td>
<td>124</td>
<td>150.8×</td>
<td>303,118,594</td>
<td>0.96</td>
<td>27.4</td>
</tr>
<tr>
<td>D1</td>
<td>E.Coli</td>
<td>4.6</td>
<td>101</td>
<td>30.0×</td>
<td>689,927</td>
<td>1.35</td>
<td>34.2</td>
</tr>
<tr>
<td>D2</td>
<td>S.cerevisiae</td>
<td>12.4</td>
<td>101</td>
<td>60.2×</td>
<td>3,599,533</td>
<td>1.53</td>
<td>58.8</td>
</tr>
<tr>
<td>D3</td>
<td>D3</td>
<td>41.8*</td>
<td>150</td>
<td>60.0×</td>
<td>8,361,240</td>
<td>1.59</td>
<td>64.7</td>
</tr>
</tbody>
</table>

† million base pairs; ‡ base pair; * the genome regions marked as “N” are excluded.
The smaller \( k \) is, the larger \( n \) is, the larger the number of reads is contained in a group. The group size can have an impact on MEC’s error-correction performance and running speed. As MEC makes corrections based on all alignments, the value of \( k \) has only small effect on its error-correction performance. Figure S4 shows that an increasing k-mer size only slightly decreases MEC’s error-correction performance. The extreme situation is: every group contains only one read when \( k \) increases to a big number. Under this extreme situation, none of the groups can be used to correct any erroneous read.

To understand the impact of k-mer size on other error-correction methods, the performance of BLESS, BFC and SGA on the real datasets R1 to R4 with various k-mer sizes is also studied. The results are shown in Table S1. It can be observed that the size of \( k \) has a significant impact on the error-correction performance of these k-mer methods. E.g., for R3, when \( k = 15 \) the gain of BLESS is only 0.042, and SGA even introduces more false-positive corrections. This evaluation is not applicable to Racer and Coral, because Racer learns the optimal size of \( k \) with a built-in procedure, while Coral does not use k-mers.

### 4.4 Effect of coverage on error correction

Coverage is an important factor that affects error-correction performance. The coverage of existing NGS data is reasonably high, usually around 30× but can be up to 200×. However, the coverage is not uniformly distributed over the genome, and can be very low in both GC-rich and GC-poor regions (Ross et al., 2013).

To understand the effect of coverage on the performance of error correction, we sample subsets of the reads from R3 to obtain datasets with various coverage. The datasets having coverage of 10, 15, 20, 25 and 30 are randomly sampled, and the six error correctors are applied on these datasets. This process is repeated five times to understand the variability of the approaches. The results are shown in Figure 3. It is clear that MEC consistently outperforms over other approaches, particularly on the datasets with a low coverage. For instance, the average gain of MEC on datasets with the coverage of 10 is 0.598, while this gain is only 0.136 for BFC. Figure 3 also shows that all the correctors have increasingly better performance given a higher coverage. Interestingly, a coverage of \( \geq 15 \) seems necessary for human chromosome 14—the performance of all methods drops markedly below this coverage level.

### 4.3 Effect of k-mer size on error correction

Instead of using k-mers to determine the erroneous bases, MEC uses k-mers as the key to map the raw reads into groups. Thus, the k-mer factor does not play the same central role that it has in the existing k-mer methods. Nonetheless, the size of the groups mapped by k-mers is determined by the value of \( k \). The smaller \( k \) is, the larger the size of all methods drops markedly below this coverage level.

### Table 2. Error-correction performance comparison between MEC, Coral, Racer, BLESS, BFC and SGA.

<table>
<thead>
<tr>
<th>data</th>
<th>corrector</th>
<th>reca</th>
<th>prec</th>
<th>gain</th>
<th>pber(%)</th>
<th>pper(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>MEC</td>
<td>0.893</td>
<td>0.924</td>
<td>0.874</td>
<td>0.103</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.803</td>
<td>0.858</td>
<td>0.728</td>
<td>0.210</td>
<td>12.6</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.822</td>
<td>0.929</td>
<td>0.760</td>
<td>0.190</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.409</td>
<td>0.650</td>
<td>0.189</td>
<td>0.879</td>
<td>13.7</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.817</td>
<td>0.927</td>
<td>0.733</td>
<td>0.196</td>
<td>8.3</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.815</td>
<td>0.922</td>
<td>0.746</td>
<td>0.196</td>
<td>11.2</td>
</tr>
<tr>
<td>R2</td>
<td>MEC</td>
<td>0.944</td>
<td>0.963</td>
<td>0.894</td>
<td>0.120</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.663</td>
<td>0.970</td>
<td>0.642</td>
<td>0.460</td>
<td>12.0</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.921</td>
<td>0.949</td>
<td>0.872</td>
<td>0.150</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.722</td>
<td>0.989</td>
<td>0.714</td>
<td>0.340</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.726</td>
<td>0.990</td>
<td>0.716</td>
<td>0.323</td>
<td>8.9</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.641</td>
<td>0.985</td>
<td>0.631</td>
<td>0.460</td>
<td>12.0</td>
</tr>
<tr>
<td>R3</td>
<td>MEC</td>
<td>0.874</td>
<td>0.937</td>
<td>0.814</td>
<td>0.260</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.690</td>
<td>0.779</td>
<td>0.495</td>
<td>0.430</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.797</td>
<td>0.890</td>
<td>0.699</td>
<td>0.230</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.558</td>
<td>0.965</td>
<td>0.538</td>
<td>0.390</td>
<td>9.9</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.641</td>
<td>0.966</td>
<td>0.613</td>
<td>0.319</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.663</td>
<td>0.967</td>
<td>0.641</td>
<td>0.310</td>
<td>6.9</td>
</tr>
<tr>
<td>R4</td>
<td>MEC</td>
<td>0.836</td>
<td>0.885</td>
<td>0.746</td>
<td>0.271</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.541</td>
<td>0.703</td>
<td>0.313</td>
<td>0.484</td>
<td>8.3</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.998</td>
<td>0.986</td>
<td>0.984</td>
<td>0.024</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.018</td>
<td>0.003</td>
<td>0.517</td>
<td>0.862</td>
<td>11.2</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.457</td>
<td>0.636</td>
<td>0.195</td>
<td>0.607</td>
<td>9.1</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.690</td>
<td>0.823</td>
<td>0.542</td>
<td>0.289</td>
<td>6.5</td>
</tr>
<tr>
<td>D1</td>
<td>MEC</td>
<td>0.999</td>
<td>0.996</td>
<td>0.995</td>
<td>0.007</td>
<td>0.075</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.998</td>
<td>0.986</td>
<td>0.984</td>
<td>0.032</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.998</td>
<td>0.981</td>
<td>0.980</td>
<td>0.032</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.980</td>
<td>0.979</td>
<td>0.979</td>
<td>0.033</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.991</td>
<td>0.999</td>
<td>0.990</td>
<td>0.016</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.990</td>
<td>0.999</td>
<td>0.989</td>
<td>0.016</td>
<td>0.41</td>
</tr>
<tr>
<td>D2</td>
<td>MEC</td>
<td>0.997</td>
<td>0.984</td>
<td>0.983</td>
<td>0.031</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.995</td>
<td>0.954</td>
<td>0.947</td>
<td>0.081</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.994</td>
<td>0.957</td>
<td>0.949</td>
<td>0.077</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.984</td>
<td>0.997</td>
<td>0.981</td>
<td>0.029</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.970</td>
<td>0.997</td>
<td>0.968</td>
<td>0.043</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.958</td>
<td>0.998</td>
<td>0.956</td>
<td>0.067</td>
<td>2.8</td>
</tr>
<tr>
<td>D3</td>
<td>MEC</td>
<td>0.987</td>
<td>0.915</td>
<td>0.896</td>
<td>0.340</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.973</td>
<td>0.762</td>
<td>0.670</td>
<td>0.510</td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.880</td>
<td>0.466</td>
<td>0.130</td>
<td>1.800</td>
<td>24.0</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.881</td>
<td>0.892</td>
<td>0.774</td>
<td>0.350</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.883</td>
<td>0.907</td>
<td>0.792</td>
<td>0.340</td>
<td>7.9</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.854</td>
<td>0.957</td>
<td>0.815</td>
<td>0.290</td>
<td>6.7</td>
</tr>
<tr>
<td>D4</td>
<td>MEC</td>
<td>0.996</td>
<td>0.939</td>
<td>0.928</td>
<td>0.281</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>Coral</td>
<td>0.971</td>
<td>0.783</td>
<td>0.702</td>
<td>0.467</td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>Racer</td>
<td>0.883</td>
<td>0.487</td>
<td>0.017</td>
<td>1.610</td>
<td>31.2</td>
</tr>
<tr>
<td></td>
<td>BLESS</td>
<td>0.909</td>
<td>0.897</td>
<td>0.795</td>
<td>0.328</td>
<td>7.6</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.891</td>
<td>0.889</td>
<td>0.819</td>
<td>0.316</td>
<td>8.9</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.846</td>
<td>0.965</td>
<td>0.807</td>
<td>0.297</td>
<td>12.1</td>
</tr>
</tbody>
</table>

Coral is unable to run data set R4 on our computer due to memory limitation. reca: recall; prec: precision; pber: per-base error rate; pper: per-position error rate.
5 DISCUSSION

The performance of error correction can be evaluated in many perspectives. Besides gain, recall, precision, per-base error rate, and per-position error rate that have been used above, speed and memory usage are another two important ones. To compare the speed and memory usage of the five algorithms with our method MEC, we carried out experiments on one computer, which has two six-core Intel Xeon E5-2620 CPUs and 64GB random-access memory, running the Red Hat 6.5 operating system. The experimental results are shown in the supplementary Figure S5 and S6. For every dataset, BLESS and BFC run the fastest, Coral the slowest. MEC and SGA have similar speed. Under the same configuration, BFC uses the least memory, but Coral always consumes the largest amount of memory. MEC and Racer usually take similar amount of memory. BLESS always consumes similar amounts of memory for different datasets due to the parameter settings. Although the time and space complexity of MEC are not ranked the best, it is straightforward to deploy it on large clusters to overcome these limitations.

To demonstrate the scalability of MEC in large computing clusters, we have carried out experiments on a cluster having five physical nodes. Each node has two six-core Intel Xeon E5-2620 CPUs and 64GB random-access memory, running the Ubuntu 12.04 operating system. Taking these resources, one master node and four worker nodes are created each having 12 cores and 60GB RAM. The parallel computing processes are managed by using the Apache Spark (Zaharia et al., 2010). For each dataset, we run MEC on the cluster having the worker nodes varying from one to four. This procedure is applied to all the datasets. Figure 4 shows the execution time of MEC for the eight datasets on the cluster. It is obvious that, increasing the number of worker nodes can significantly reduce the execution time. For instance, by using one worker node, the running time for R4 is 19.8 hours, while this time is reduced to 5.0 hours when all the four worker nodes are used. It is noteworthy that the execution time of R4 is longer than the expected time, as the RDDS (resilient distributed datasets) are unable to fit the available RAM provided by the single worker node. For this situation, the RDD’s StorageLevel is set to MEMORY_ONLY_SER, while the default parallelism level is increased to 3 per core. Obviously, the most effective approach to handle the big memory usage is to increase the number of worker nodes. However, in the situation where the resource is limited, we can tune the Spark configurations to solve the problem.

De novo sequence assembly results can be also used to evaluate the performance of error correction. It has been reported that the error correction can significantly improve the quality of sequence assembly (Sameith et al., 2016) as well as downstream analysis (Fujimoto et al., 2014; Yang et al., 2013). To assess the impact of MEC as well as the representative error correctors on sequence assembly, we run Velvet (Zerbino and Birney, 2008) on the raw reads as well as on the corrected reads. Unlike other assemblers, Velvet is released without built-in error corrector. Hence, it is appropriate to use Velvet to examine the impact of the error correctors.

The experiments are conducted on the eight datasets and the corresponding corrected datasets. The result reveals that the N50 size increases significantly after error-correction by MEC, with an improvement ranging from 2 folds to 4 folds. The maximum length of the assembled sequences also increases markedly. E.g., the maximum assembled sequence length of D2 increases from 74,526 to 316,460. Besides assessing the error-correction effectiveness for MEC, other error-correction methods are also examined. The comparative experiments carried out on the R1 and R2 demonstrate that all error correctors improve the quality of sequence assembly with MEC and Racer considerably outperforming the rest. Although Racer produces longer contigs, its lower NGA50 value suggests that these might be mis-assemblies. More results are presented in the supplementary Table S3.

6 CONCLUDING REMARKS

Error correction plays an important role in next-generation sequencing data analysis. Existing error-correction methods have serious drawbacks. In this study, we have proposed a two-layered MapReduce method, named MEC, to correct the substitution errors in NGS datasets. Given an input data set, MEC maps the raw reads into groups using all of the distinct k-mers, and then it aligns the reads within each group to identify erroneous bases. At the second layer, the erroneous bases are mapped to their corrections and then corrected using a statistics technique. MEC is superior to five existing methods in correction accuracy.

Funding: This work is collectively supported by the National Natural Science Foundation for Young Scientists of China (Grant No. 31501070), the Natural Science Foundation of Guangxi Province, China (Grant No. 2016GXNSFCA380006), and the Scientific Research Foundation of GuangXi University (Grant No. XGZ150316).

REFERENCES


MapReduce for accurate error correction of next-generation sequencing data
Liang Zhao\textsuperscript{1,2}, Qingfeng Chen\textsuperscript{1}, Wencui Li\textsuperscript{2}, Peng Jiang\textsuperscript{1}, Limsoon Wong\textsuperscript{3}, *, Jinyan Li\textsuperscript{4}, *
\textsuperscript{1}School of Computing and Electronic Information, Guangxi University, Nanning 530004, China.
\textsuperscript{2}Taihe Hospital, Hubei University of Medicine, Hubei 442000, China.
\textsuperscript{3}School of Computing, National University of Singapore, Singapore 117417, Singapore.
\textsuperscript{4}Advanced Analytics Institute and Centre for Health Technologies, University of Technology Sydney, PO Box 123, Broadway, NSW 2007, Australia.

1 MEC: DETAILS
Due to the page limitation, we present more detail of the constituent steps of MEC as follows.

Step 1: Mapping paired-end reads into groups
We use the tuple \((\kappa, j, i, \iota)\) to denote an occurrence of the k-mer, \(\kappa\), starting at the \(i\)th position of the paired-end read \(r_j\) having mate index \(\iota\). The indices \(i\) and \(j\) are 0-based. For example, suppose the input set of paired-end reads is \(R = \{\text{AGTTCG} \cdots \text{GGCTCA}, \text{TTCGAC} \cdots \text{GAGGAC}, \text{TCTCAG} \cdots \text{TTAGGC}\}\). Then the tuples representing occurrences of the 3-mer TCA are \((TCA, 0, 3, 1)\) and \((TCA, 2, 2, 0)\).

A hash function is used to map a k-mer \(\kappa\) to a group \(G_\kappa\), which corresponds to a node in the computing cluster. The set of tuples in group \(G_\kappa\) is denoted by \(G(\kappa)\). Some groups \(G(\kappa)\) are much larger than others. In such a case, the k-mer \(\kappa\) is likely from repeat regions or has low sequence complexity. We discard such groups. The threshold on group size for the decision to discard a group is determined as follows.

The mean and standard deviation of the number of paired-end reads of each group are estimated from a subset containing 1000 groups. When the number of paired-end reads in a group is more than 5 standard deviations higher than the mean, the group is discarded. This threshold is dataset-dependent, and is mostly around 200 in our study. Similar to the reads from repeat regions, the reads mapping to multiple places causing huge \(G(\kappa)\) are discarded as well.

Step 2: Aligning paired-end reads and identifying errors
We have described the whole procedure of identifying prospective erroneous base by using penalty-aware multiple sequence alignment. Let \(\kappa, i, a, \iota\) and \((\kappa, j, b, \iota), \iota \neq j\), be two occurrences of the k-mer \(\kappa\) in the group \(G_\kappa\). An alignment between \(r_i\) and \(r_j\) respecting these two occurrences of \(\kappa\) is generated using a penalty-aware sliding algorithm. We use Figure S1 as an example to help us describe the penalty-aware sliding algorithm. Let \(r_i = R_1 \cdots R_2\) and \(r_j = R'_{1} \cdots R'_{2}\) be two paired-end reads that we want to align with respect to a k-mer \(\kappa\) (shown in purple in the figure) that occurs at position \(a\) in \(R_1\) (i.e. at position \(a\) of the read at the up-stream end of \(r_i\)) and position \(b\) in \(R'_{1}\) (i.e. at position \(b\) of the read at the up-stream end of \(r_j\)).

First, \(R_1\) and \(R'_{1}\) are aligned by setting the two said occurrences of \(\kappa\) (shown in purple) directly on top of one another. That is, every position \(h + a\) in \(R_1\) is aligned to the position \(h + b\) in \(R'_{1}\). This causes a segment of length \(b - a\) at the end of \(R_1\) (or \(R'_{1}\) if \(a > b\)) to stick out into the insert region of \(r_j\) (or \(r_i\) if \(a > b\)). Let \(x = b - a\).

Next, \(R_2\) and \(R'_{2}\) are aligned by sliding one on top of the other by \(y\) bases. That is, every position \(h + y\) in \(R_2\) is aligned to the position \(h + y\) in \(R'_{2}\). By convention, \(y\) is negative if the start of \(R_2\) is aligned to a position down-stream of the start of \(R'_{2}\) (i.e. a segment of length \(y\) at the start of \(R_2\) sticks out into the insert region of \(r_j\)); and \(y\) is positive otherwise (and a segment of length \(y\) at the start of \(R'_{2}\) sticks out into the insert region of \(r_j\)).

The alignment is given a score which is defined as the number of matched bases minus the number of mismatched bases and an insertion penalty. The sum \(x + y\) represents the amount of insertions made by the alignment, if every paired-end read has the same insert size. However, the insert size of paired-end reads is not a constant. So we moderate \(x + y\) by the variance \(\sigma_0^2\) of insert size of the paired-end reads in the input set \(R\), and define the insertion penalty as \(s = c_0 \times (x + y)^2 / \sigma_0^2\). Here \(c_0\) is a multiplicative factor to make it easier to distinguish tandem repeats. In this study, the average size of tandem repeats is assumed to be 5, and we set \(c_0 > \sigma_0^2 / 5^2\).

*to whom correspondence should be addressed
Accurate error correction for NGS

Fig. S1. An illustration of the penalty-aware alignment of paired-end reads. The mean and standard deviation of the insert size of the paired-end reads are $\mu_0$ and $\sigma_0$ respectively.

Fig. S2. An illustration of incomplete inclusion of reads by using k-mers. The k-mer of interest is in blue, while the mutated one (having errors) is in purple.

The final pairwise alignment is the one having the maximum score among all possible alignments (by trying different values for y). A “bandwidth” search is used to accelerate the alignment. That is, for each possible value of y, a few positions ($\leq 5$) are tested at the start of the alignment, and the alignment is continued only if the partial alignment is highly matched.

Step 3: Mapping erroneous bases to candidate corrections

Existing approaches which use k-mers to correct errors cannot guarantee the full inclusion of reads covering the erroneous position; see Figure S2 for an example. On one hand, the sequences are error prone. As a result, a read that comes from a sequence containing the k-mer $\kappa$ is not mapped to the group $G_\kappa$, if it contains an error where $\kappa$ is located. E.g. if the 3-mer AGT of the paired-end read AGTTCG...GGCTCA mutates to ACT, then this paired-end read does not get included in the group with 3-mer AGT as the key. As illustrated by Figure S2, the read $R_2$ is not included by the k-mer colored in blue as the original k-mer has mutated to another one. On the other hand, the k-mer itself most likely also causes an incomplete inclusion of paired-end reads due to the partial coverage of k-mer at the end of each read. E.g. if the paired-end read shown above shifts one position to the left (i.e. the read at the up-stream end starts with “GTT”), then this paired-end read gets excluded when the 3-mer AGT is used for the mapping. Take Figure S2 for an example, the reads $R_5$ and $R_6$ are not included by the k-mer in blue color as only partial sequence of them overlaps with the k-mer. We can also see that the performance of existing methods is sensitive to the size of k-mers.

We have described how to fish out the prospective erroneous bases denoted by $(g, x, i)$, in the main text. Now, it remains to cluster all the bases, including the erroneous bases and supportive bases, to correct the erroneous bases.

Per the main text, the position of $(g, x, i)$ is denoted by $\varphi(g, x, i)$. However, the value of $\varphi(g, x, i)$—let us call this a position reference—is generated independently on different computing nodes. Thus the same position gets a different position reference for the same position to the same value. Let $D_u$ denote the set of entries of the form $(\text{position reference}, \text{positions})$ generated at computing node $u$, where each entry $(\varphi, \{p_1, \ldots, p_m\})$ means each positions $p_i = (g_i, x_i, i)$, $i = 1 \ldots m$, has been mapped to the position reference $\varphi$. The total size of $D_1, D_2, \ldots, D_n$ (for all n computing nodes) can be too huge to be processed by in-memory algorithms. In this study, we proposed the external-memory algorithm. It is composed of two steps:

1. Conduct the self-join of $D_u$ produced by each computing node $u$. A self-join of $D_u$ is defined as

$$\min\{\varphi_1, \bigcup_{(\varphi_2, P_2) \in D_u, P_1 \cap P_2 \neq \emptyset} P_2 \mid (\varphi_1, P_1) \in D_u\}.$$
Here, \((\varphi, P) \in \min R\) if and only if for every \((\varphi, P') \in R\) such that \(P_1 = P_1\), it is the case that \(\varphi_1 \leq \varphi_2\). In other words, all entries in \(D_\alpha\) that share at least one position are merged into a unique entry. Note that the above self-join is only a mathematical definition; it can be implemented efficiently using disjoint set.

2. Recursively do the binary-join of two \(D_\alpha\) and \(D_\beta\), until only one remains. Based on the self-join, it can be guaranteed that the intersection of any two entries of \(D_\alpha\) or \(D_\beta\) is empty. Hence, it is safe to merge each entry of \(D_\alpha\) to \(D_\beta\) progressively in order to achieve the binary-join of \(D_\alpha\) and \(D_\beta\). To make it more efficient, we create indices for all the \(P_\alpha\) against their \(\varphi_\alpha\), say \(I(P)\). For each \((\varphi_\alpha, P_\alpha)\) of \(D_\alpha\), we check the existence of \(P_\beta' = P_\beta\) such that \(P_\alpha \in P_\beta\). If in case there exists a \((\varphi_\alpha, P_\alpha)\) such that \((\varphi_\alpha, P_\alpha) \in I(P)\), we output \((\varphi_\alpha, P_\alpha) \cup P_\beta\). This process is repeated until all entries in \(D_\alpha\) are exhausted; at this point, all remaining entries in \(D_\alpha\) are copied to the output. If \(D_\alpha\) is too large to be loaded into the main memory, we split it into small chunks and apply the above process to each chunk and \(D_\beta\).

Per the above processes, we can collect all the reads covering prospective erroneous positions. Now it remains to figure out the reference base value as well as the prospective erroneous base values. Clearly, but not perfectly, taking the most common base value as the reference is optimal. However, considering all the rest base values as erroneous is not very ideal. Several scenarios can demonstrate the exceptions. Firstly, the subclones of cancer genomes may result in multiple correct reference values at the same prospective erroneous position. Secondly, the position containing errors in the heterozygous allele of a diploid genome has exactly two correct reference values. Thirdly, the reads from imperfect repeat regions will produce multiple illusive reference values. To diminish the confusion, we further restrict the prospective erroneous base as the one having frequency no larger than three. The rationale is as follows.

### Step 4: Correcting errors

Based on the results produced from the Step 3, the corrected bases are recorded. Per the main text, it is denoted as \((g, x, \iota, \beta)\), where \(\beta\) is the corrected base. We use \(g\) as the key, and the set of \((x, \iota, \beta)\) as the value to map all the corrections of each paired-end read together. Later, they are distributed to different computing nodes according to the key \(g\). Finally, all the paired-end reads are distributed to the computing nodes the same way as distributing \(g\), and the erroneous bases are replaced by the correct ones according to the values of \(g\).

## 2 EXPERIMENTAL RESULTS

### 2.1 Effect of k-mer size on error correction

Instead of using k-mers to determine the erroneous bases, MEC uses k-mers as the key to map the raw reads into groups. Thus, the k-mer factor does not play the same central role that it has in existing k-mer-based methods. Nonetheless, the size of the groups mapped by k-mers is determined by the value of \(k\). The smaller \(k\) is, the larger number of reads contained in a group. And group size can have an impact on MEC’s error-correction performance and running speed.

However, since MEC makes corrections based on all alignments, the value of \(k\) only has small effect on its error-correction performance. The experimental results carried out on the eight data sets are shown in Figure S4. It shows that increasing k-mer size only slightly decreases MEC’s error-correction performance. The extreme situation is when only one read when \(k\) increases to a big number. Under this extreme situation, none of the groups can be used to correct any erroneous read.

To understand the impact of k-mer size on other error-correction methods, the performance of BLESS, BFC and SGA on the real datasets R1 to R4 with various k-mer sizes is studied. The results are shown in Table S1. It can be observed that the size of \(k\) has a significant impact on the error-correction performance of these k-mer-based methods. For example, when \(k = 15\), the gain of BLESS on R3 is only 0.042, and SGA even introduces more false-positive corrections. This evaluation is not applicable to Racer and Coral, because Racer learns the optimal size of \(k\) with a built-in procedure, while Coral does not use k-mers.

### 2.2 Speed and memory usage

Speed and memory usage are another two important factors which we have assessed, as real-world datasets are huge (sometimes up to hundreds of gigabytes, even terabytes). To compare the speed and memory usage of the five algorithms and our method MEC, we carried out all the experiments on the same computer, which has two six-core Intel Xeon X5690 3.47GHz CPUs and 96G random-access memory, installed with the Red Hat 6.5 operating system.

The speed of these six methods running with 12 threads is shown in Figure S4. For every dataset, BLESS and BFC run the fastest, Coral the slowest, while MEC and SGA have similar speed. It is clear that the sequence alignment-based approaches are slower than the k-mer-based methods.

Under the same configuration, the memory usage for all six methods are shown in Figure S5. For all the datasets, BFC uses the least memory, while BLESS consistently consumes similar amounts of memory due to the parameter settings. Coral always consumes the largest
Accurate error correction for NGS

Fig. S3. The gain of MEC on the eight datasets with various size of k-mers.

Table S1. Error-correction performances of BLESS, BFC and SGA on dataset R1 to R4 with various size of k.

<table>
<thead>
<tr>
<th>k</th>
<th>Corrector</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>BLESS</td>
<td>0.031</td>
<td>0.089</td>
<td>0.055</td>
<td>0.002</td>
<td>0.308</td>
<td>0.571</td>
<td>0.533</td>
<td>0.001</td>
<td>0.013</td>
<td>0.051</td>
<td>0.042</td>
<td>-0.862</td>
<td>1.120</td>
<td>1.202</td>
<td>0.845</td>
<td>0.973</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.051</td>
<td>0.059</td>
<td>0.050</td>
<td>0.039</td>
<td>0.356</td>
<td>0.521</td>
<td>0.513</td>
<td>0.001</td>
<td>0.047</td>
<td>0.067</td>
<td>0.043</td>
<td>0.019</td>
<td>1.117</td>
<td>1.195</td>
<td>0.825</td>
<td>0.923</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.057</td>
<td>0.025</td>
<td>0.047</td>
<td>0.053</td>
<td>0.241</td>
<td>0.253</td>
<td>0.300</td>
<td>0.241</td>
<td>0.008</td>
<td>0.012</td>
<td>-0.063</td>
<td>0.004</td>
<td>1.067</td>
<td>1.140</td>
<td>0.903</td>
<td>0.894</td>
</tr>
<tr>
<td>17</td>
<td>BLESS</td>
<td>0.120</td>
<td>0.241</td>
<td>0.177</td>
<td>0.006</td>
<td>0.443</td>
<td>0.701</td>
<td>0.695</td>
<td>0.002</td>
<td>0.063</td>
<td>0.205</td>
<td>0.165</td>
<td>-0.823</td>
<td>0.932</td>
<td>1.105</td>
<td>0.767</td>
<td>0.921</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.243</td>
<td>0.214</td>
<td>0.189</td>
<td>0.135</td>
<td>0.413</td>
<td>0.635</td>
<td>0.617</td>
<td>0.001</td>
<td>0.198</td>
<td>0.201</td>
<td>0.163</td>
<td>0.045</td>
<td>0.096</td>
<td>1.024</td>
<td>0.751</td>
<td>0.882</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.386</td>
<td>0.274</td>
<td>0.337</td>
<td>0.349</td>
<td>0.485</td>
<td>0.530</td>
<td>0.517</td>
<td>0.153</td>
<td>0.177</td>
<td>0.162</td>
<td>0.139</td>
<td>0.786</td>
<td>0.934</td>
<td>0.745</td>
<td>0.807</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>BLESS</td>
<td>0.209</td>
<td>0.372</td>
<td>0.265</td>
<td>0.009</td>
<td>0.497</td>
<td>0.774</td>
<td>0.762</td>
<td>0.002</td>
<td>0.097</td>
<td>0.332</td>
<td>0.252</td>
<td>-0.767</td>
<td>0.906</td>
<td>0.529</td>
<td>0.627</td>
<td>0.903</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.446</td>
<td>0.376</td>
<td>0.307</td>
<td>0.214</td>
<td>0.789</td>
<td>0.854</td>
<td>0.786</td>
<td>0.487</td>
<td>0.337</td>
<td>0.325</td>
<td>0.297</td>
<td>0.076</td>
<td>0.534</td>
<td>0.591</td>
<td>0.632</td>
<td>0.476</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.623</td>
<td>0.467</td>
<td>0.553</td>
<td>0.572</td>
<td>0.787</td>
<td>0.842</td>
<td>0.899</td>
<td>0.676</td>
<td>0.582</td>
<td>0.501</td>
<td>0.491</td>
<td>0.402</td>
<td>0.317</td>
<td>0.472</td>
<td>0.437</td>
<td>0.429</td>
</tr>
<tr>
<td>21</td>
<td>BLESS</td>
<td>0.317</td>
<td>0.614</td>
<td>0.437</td>
<td>0.012</td>
<td>0.517</td>
<td>0.815</td>
<td>0.793</td>
<td>0.002</td>
<td>0.127</td>
<td>0.526</td>
<td>0.417</td>
<td>-0.611</td>
<td>0.892</td>
<td>0.351</td>
<td>0.453</td>
<td>0.882</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.717</td>
<td>0.603</td>
<td>0.510</td>
<td>0.337</td>
<td>0.838</td>
<td>0.901</td>
<td>0.837</td>
<td>0.510</td>
<td>0.609</td>
<td>0.593</td>
<td>0.507</td>
<td>0.137</td>
<td>0.252</td>
<td>0.384</td>
<td>0.403</td>
<td>0.292</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.716</td>
<td>0.547</td>
<td>0.566</td>
<td>0.601</td>
<td>0.841</td>
<td>0.914</td>
<td>0.907</td>
<td>0.739</td>
<td>0.656</td>
<td>0.572</td>
<td>0.538</td>
<td>0.479</td>
<td>0.237</td>
<td>0.391</td>
<td>0.376</td>
<td>0.352</td>
</tr>
<tr>
<td>23</td>
<td>BLESS</td>
<td>0.410</td>
<td>0.724</td>
<td>0.523</td>
<td>0.019</td>
<td>0.649</td>
<td>0.987</td>
<td>0.943</td>
<td>0.003</td>
<td>0.183</td>
<td>0.698</td>
<td>0.527</td>
<td>-0.531</td>
<td>0.872</td>
<td>0.327</td>
<td>0.405</td>
<td>0.878</td>
</tr>
<tr>
<td></td>
<td>BFC</td>
<td>0.793</td>
<td>0.705</td>
<td>0.637</td>
<td>0.454</td>
<td>0.912</td>
<td>0.986</td>
<td>0.956</td>
<td>0.627</td>
<td>0.739</td>
<td>0.705</td>
<td>0.601</td>
<td>0.188</td>
<td>0.207</td>
<td>0.332</td>
<td>0.327</td>
<td>0.208</td>
</tr>
<tr>
<td></td>
<td>SGA</td>
<td>0.793</td>
<td>0.631</td>
<td>0.652</td>
<td>0.684</td>
<td>0.913</td>
<td>0.976</td>
<td>0.957</td>
<td>0.814</td>
<td>0.726</td>
<td>0.625</td>
<td>0.630</td>
<td>0.532</td>
<td>0.201</td>
<td>0.482</td>
<td>0.320</td>
<td>0.307</td>
</tr>
</tbody>
</table>

reca: recall; prec: precision; pber: per-base error rate; pper: per-position error rate.

amount of memory. Regarding MEC and Racer, they usually take similar amount of memory. The memory usage of MEC is related to raw-read volume, coverage and k-mer size. Thus its space consumption increases along with expansion of data size.

Although the time and space complexity of MEC are not ranked the best, naturally it is straightforward to deploy it on large clusters to overcome the limitations. Experimental results carried out on a cluster having five nodes show that MEC has good scalability in terms of execution time and memory usage. The detailed results are shown in the discussion section of the main manuscript. For instance, using one worker node, the running time for R4 is 19.8 hours, while this value reduces to 5.0 hours when all the four worker nodes are used. In case the RAM usage exceeds the limit of the worker nodes, we can tune the StorageLevel of RDDs (resilient distributed datasets) as well as the default parallelism level of Spark (Zaharia et al., 2010).

2.3 Error correction improves genome assembly

Error correction can significantly improve the quality of sequence assembly, as well as downstream analysis (Fujimoto et al., 2014; Yang et al., 2013). Thus, we conduct error correction using MEC, and then assemble the short paired-end reads into long contigs using...
Accurate error correction for NGS

Fig. S4. Speed comparison between the error-correction methods. The result of Coral is not obtained for some datasets because the amount of RAM required is beyond the available limit.

Fig. S5. Memory usage comparison between the error-correction methods. The result of Coral is not obtained for some datasets because the amount of RAM required is beyond the available limit.

Velvet (Zerbino and Birney, 2008). Unlike other assemblers that are released with either standalone or built-in error corrector, e.g., Quake (Kelley et al., 2010) and SOAPdenovo (Li et al., 2010), Velvet does not come with an error corrector. Hence, it is appropriate to use Velvet to examine the impact of MEC.

The experiments are conducted on the eight datasets and the corresponding corrected datasets. The assembly is evaluated using N50 value, NGA50 value and maximum length of assembled sequences. N50 is defined as the maximum length of the sequences at which the collection of those sequences not shorter than this length contains at least half of the sum of the lengths of all the sequences (Miller et al., 2010). NGA50 is introduced by QUAST (Gurevich et al., 2013), which is an aligned N50 with respect to the size of reference genome instead of the assembled length. Both the N50 and NGA50 are computed based on the contigs produced by Velvet having length greater than 100bp.

The results for datasets R1 to R2 and D1 to D2 are shown in Table S2, while the results are not obtained for datasets R3, R4, D3 and D4 as Velvet cannot handle such large data on our computers. It can be seen that the N50 size increases significantly after error-correction by MEC, with the improvement ranging from 2 to 4 folds. Regarding the maximum length of the assembled sequences, it is also increased markedly. E.g., the N50 of D2 is 17,357 before error correction, but this value is increased to 69,954 after error correction, and the maximum assembled sequence length is increased from 74,526 to 316,460. These results confirm that MEC is effective for correcting errors in next-generation sequencing data.

Besides assessing the error-correction effectiveness for MEC, the other error-correction methods are also examined. The comparative experiments are carried out on the real datasets R1 and R2 for all the methods. The results are presented in Table S3. On the two datasets, MEC and Racer considerably outperform the other methods. Although Racer produces longer contigs, its lower NGA50 value suggests that these tend to be mis-assemblies. Datasets D1 and D2 are not used in this comparison as the performance of all methods are very good, indicating almost all the errors are corrected perfectly. Thus, very similar assembly results would be produced on D1 and D2. Regarding datasets R3, R4, D3 and D4, the results are unable to obtained due to memory requirement of Velvet is beyond the limitation of our computer.
Table S2. Impact of MEC on sequence assembly.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Before correction</th>
<th>After correction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N50 max length</td>
<td>N50 max length</td>
</tr>
<tr>
<td>D1</td>
<td>45,236 173,805</td>
<td>87,665 269,712</td>
</tr>
<tr>
<td>D2</td>
<td>17,357 74,526</td>
<td>69,954 316,460</td>
</tr>
<tr>
<td>R1</td>
<td>974 13,081</td>
<td>2,483 35,964</td>
</tr>
<tr>
<td>R2</td>
<td>1,029 17,184</td>
<td>2,865 43,594</td>
</tr>
</tbody>
</table>

Velvet is unable to run on the datasets R3, R4, D3 and D4 on our computer with 96G random access memory. The results are obtained with contig length greater than 100bp.

Table S3. Error-correction performance comparison with respect to sequence assembly on datasets R1 and R2.

<table>
<thead>
<tr>
<th>Corrector</th>
<th>R1</th>
<th>R2</th>
<th>R1</th>
<th>R2</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEC</td>
<td>2,236</td>
<td>2,865</td>
<td>2,997</td>
<td>3,287</td>
<td>39,972</td>
<td>43,594</td>
</tr>
<tr>
<td>Coral</td>
<td>613</td>
<td>547</td>
<td>1,295</td>
<td>1,175</td>
<td>19,183</td>
<td>22,029</td>
</tr>
<tr>
<td>Racer</td>
<td>3,356</td>
<td>3,979</td>
<td>1,479</td>
<td>1,656</td>
<td>31,278</td>
<td>35,129</td>
</tr>
<tr>
<td>BLESS</td>
<td>866</td>
<td>979</td>
<td>1,378</td>
<td>1,437</td>
<td>30,037</td>
<td>29,305</td>
</tr>
<tr>
<td>BFC</td>
<td>996</td>
<td>1,271</td>
<td>1,464</td>
<td>1,773</td>
<td>28,634</td>
<td>30,217</td>
</tr>
<tr>
<td>SGA</td>
<td>745</td>
<td>674</td>
<td>1,736</td>
<td>1,275</td>
<td>23,821</td>
<td>21,924</td>
</tr>
</tbody>
</table>

The results are obtained with contig length greater than 100bp.

3 MATERIALS
The four real sequencing data sets are downloaded from the GAGE (Salzberg et al., 2011) website:

- S. aureus: http://gage.cbcb.umd.edu/data/Staphylococcus_aureus;
- R. sphaeroides: http://gage.cbcb.umd.edu/data/Rhodobacter_sphaeroides;
- H. Chromosome 14: http://gage.cbcb.umd.edu/data/Hg_chr14;

4 COMMANDS
The commands used for running Coral, Racer, BLESS, BFC, and SGA on the eight data sets are as follows.

```
coral -fq R1_1_2.fastq -o R1_coral_1_2.fastq -illumina -p 12
racer R1_1_2.fastq R1_racer_1_2.fastq 2800000
bless -read1 R1_1.fastq -read2 R1_2.fastq -prefix R1_bless -kmerlength 23
bfc -s 3m -t12 R1_1_2.fastq > R1_bfc_1_2.fastq
sga preprocess -pe-mode 1 -permute-ambiguous -no-primer-check -o tmp.fq R1_1.fastq R1_2.fastq
go index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o R1_sga_1_2.fastq tmp.fq
```

```
coral -fq R2_1_2.fastq -o R2_coral_1_2.fastq -illumina -p 12
racer R2_1_2.fastq R2_racer_1_2.fastq 4600000
bless -read1 R2_1.fastq -read2 R2_2.fastq -prefix R2_bless -kmerlength 23
bfc -s 5m -t12 R2_1_2.fastq > R2_bfc_1_2.fastq
sga preprocess -pe-mode 1 -permute-ambiguous -no-primer-check -o tmp.fq R2_1.fastq R2_2.fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o R2_sga_1_2.fastq tmp.fq
```

```
coral -fq R3_1_2.fastq -o R3_coral_1_2.fastq -illumina -p 12
racer R3_1_2.fastq R3_racer_1_2.fastq 88300000
bless -read1 R3_1.fastq -read2 R3_2.fastq -prefix R3_bless -kmerlength 23
```

R3
Accurate error correction for NGS

bfc -s 88m -t12 R3_1,2.fastq > R3_bfc_1,2.fastq
sga preprocess –pe-mode 1 –permute-ambiguous –no-primer-check -o tmp.fq R3_1, fastq R3_2, fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o R3_sga_1,2.fastq tmp.fq

coral -fq R4_1,2.fastq -o R4_coral_1,2.fastq -illumina -p 12
racer R4_1,2.fastq R4_racer_1,2.fastq 249200000
bless -read1 R4_1,2.fastq -read2 R4_2,fastq -prefix R4_bless -kmerlength 23
sga preprocess –pe-mode 1 –permute-ambiguous –no-primer-check -o tmp.fq R4_1,fastq R4_2,fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o R4_sga_1,2.fastq tmp.fq

coral -fq D1_1,2.fastq -o D1_coral_1,2,fastq -illumina -p 12
racer D1_1,2.fastq D1_racer_1,2.fastq 46000000
bless -read1 D1_1,fastq -read2 D1_2,fastq -prefix D1_bless -kmerlength 23
sga preprocess –pe-mode 1 –permute-ambiguous –no-primer-check -o tmp.fq D1_1,fastq D1_2,fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o D1_sga_1,2.fastq tmp.fq

coral -fq D2_1,2.fastq -o D2_coral_1,2,fastq -illumina -p 12
racer D2_1,2.fastq D2_racer_1,2.fastq 12400000
bless -read1 D2_1,fastq -read2 D2_2,fastq -prefix D2_bless -kmerlength 23
sga preprocess –pe-mode 1 –permute-ambiguous –no-primer-check -o tmp.fq D2_1,fastq D2_2,fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o D2_sga_1,2.fastq tmp.fq

coral -fq D3_1,2.fastq -o D3_coral_1,2,fastq -illumina -p 12
racer D3_1,2.fastq D3_racer_1,2.fastq 41800000
bless -read1 D3_1,fastq -read2 D3_2,fastq -prefix D3_bless -kmerlength 23
sga preprocess –pe-mode 1 –permute-ambiguous –no-primer-check -o tmp.fq D3_1,fastq D3_2,fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o D3_sga_1,2.fastq tmp.fq

coral -fq D4_1,2.fastq -o D4_coral_1,2,fastq -illumina -p 12
racer D4_1,2.fastq D4_racer_1,2.fastq 41800000
bless -read1 D4_1,fastq -read2 D4_2,fastq -prefix D4_bless -kmerlength 23
sga preprocess –pe-mode 1 –permute-ambiguous –no-primer-check -o tmp.fq D4_1,fastq D4_2,fastq
sga index -a ropebwt -t 12 tmp.fq
sga correct -t 12 -o D4_sga_1,2.fastq tmp.fq

REFERENCES


