Vol. 00 no. 00 2017
Pages 1–8

MapReduce for accurate error correction of next-generation sequencing data

Liang Zhao[1,2], Qingfeng Chen[1], Wencui Li[2], Peng Jiang[1], Limsoon Wong[3, *], Jinyan Li[4, *]
[1]School of Computing and Electronic Information, Guangxi University, Nanning 530004, China.
[2]Taihe Hospital, Hubei University of Medicine, Hubei 442000, China.
[3]School 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 $s^i_{1\dots k}$ and $s^j_{1\dots k}$ are said to overlap if $s^i_{2\dots k} = s^j_{1\dots(k-1)}$ or $s^j_{2\dots k} = s^i_{1\dots(k-1)}$. In the former case, there is an edge pointing from s^i to s^j, while in the latter the edge is pointing from s^j to s^i. 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.*,

*to whom correspondence should be addressed

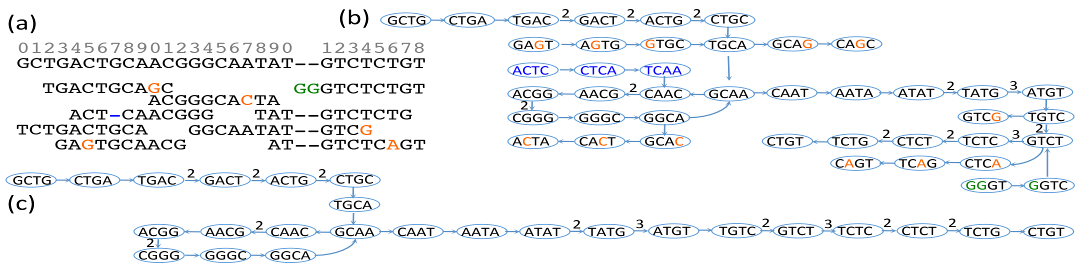

Fig. 1. An example of NGS data and its de Bruijn graph. The short stretches of sequences in Panel (a) are the reads generated from an NGS platform, while the long sequence is the reference. The reference is often unknown but, for ease of illustration, it is shown here to demonstrate substitutions (coloured in orange), insertions (green) or deletions (light blue) errors. There is no "-" in the real-life reference and sequenced reads, but it is shown here also for better understanding. Panel (b) is the de Bruijn graph constructed from all the short sequences in Panel (a) with a k-mer size of 4. Panel (c) is the simplified error-corrected version of the de Bruijn graph of Panel (b). The numbers along the edges represent their multiplicities.

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 (Sheikhizadeh and de Ridder, 2015). These methods only differ in terms of 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 κ be a k-mer and r_j be the jth read in R. An occurrence of κ at the ith position of r_j is denoted as (κ, j, i). We consider single-end reads here to simplify exposition; paired-end reads are discussed in the next section. The group corresponding to κ is denoted as G_κ, and κ is called the key of the group. The set of k-mer occurrences assigned to G_κ is denoted by $G(\kappa)$. And for every κ, i, and j, it is the case that $(\kappa, j, i) \in G(\kappa)$ if and only if κ occurs at the ith 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 alignment$_1$, alignment$_2$ and alignment$_3$ of the raw reads in Figure 2. For single-end reads, a simple ungapped alignment is

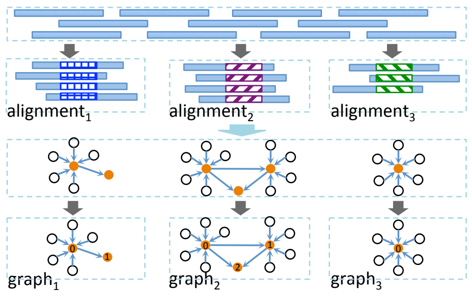

Fig. 2. The two-layered MapReduce framework for correcting errors in NGS data. In the first layer, all raw reads (light-blue bars) are mapped to groups according to the groups' k-mer keys (crosshatched short bars). The reads in each group are aligned and the discordant bases are determined. Subsequently, the discordant bases at each position are merged and passed to the second MapReduce layer for building supporting graphs. In the second layer, these graphs are decomposed into small connected graphs for error correction. For each connected graph, the orange nodes are the candidate erroneous bases to be corrected, while the unfilled ones are considered as correct bases.

used: For each pair of occurrences (κ, j, i) and (κ, 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. graph$_3$ in Figure 2) with exceptions caused by repeats (e.g., graph$_2$ in Figure 2) and multiple errors at the same position (e.g. graph$_1$ 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 κ_x 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, κ_x is expected to be correctly read $d * (1 - e)^k$ 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 κ_y be the k-mer immediately after y in the genome. Then κ_y is expected to be correctly read $d * (1 - e)^k$ times. Since the read length is l, and $k < l$, we expect $(l-1)/l$ of these $d*(1-e)^k$ 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 $n = ((l - 1)/l) * d * (1 - e)^k$ 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 $((l-1)/l)*d*(1-e)^{k+1}$ correct bases that can be used to build a consensus to correct the erroneous 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 $((l - 1)/l) * d * (1 - e)^{k+1}$ 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 $((l-1)/l)*d*(1-e)^{k+1}$ 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 jth paired-end read in the input set R. Given a k-mer κ, we denote an occurrence of κ starting at the ith position of the read at the up-stream end of r_j by $(\kappa, j, i, 0)$. Similarly we denote an occurrence of κ 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 κ to a group G_κ, which corresponds to a node in the computing cluster, thus sending all tuples (κ, j, i, ι) that denote occurrences of κ in the paired-end reads to the group G_κ.

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_κ m times if there are m occurrences of κ in that paired-end read. Similarly, we denote the set of tuples in group G_κ by $G(\kappa)$. We also say a paired-end read r_j is mapped to group G_κ if $(\kappa, j, i, \iota) \in G(\kappa)$ for some i and ι.

Some groups $G(\kappa)$ are much larger than other groups. In such a case, the k-mer κ 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 (κ, j, i, ι) of some k-mer κ and a list of occurrences $(\kappa, j_1, i_1, \iota)$, ..., $(\kappa, j_n, i_n, \iota)$ of the k-mer κ, 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 $(\kappa, j_1, i_1, \iota)$, ..., $(\kappa, j_n, i_n, \iota)$ are aligned to the occurrence (κ, j, i, ι) of the k-mer κ. Note that $j \notin \{j_1, ..., j_n\}$; i.e. we do not align two occurrences of κ in the same read. Such an anchor alignment induces a multiple alignment A of the paired-end reads $r_j, r_{j_1}, ..., r_{j_n}$ 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_g} in A if and only if the position x in r_j is aligned to the position y in r_{j_g} in A_g, for any x, y, and g. A set of positions $\{x$ in r_j, x_1 in r_{j_1}, ..., and 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_κ, one or more anchored alignments of the paired-end reads mapped to the group are generated based on the occurrences of the k-mer κ. The procedure is as follows, after initializing the set of anchored alignments to empty:

1. Randomly select a tuple (κ, j, i, ι) in $G(\kappa)$ that has not yet been assigned to any anchored alignment.
2. For each anchor (κ, j', i', ι') of the current set of anchored alignments, where $\iota = \iota'$, 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 (κ, j, i, ι) and (κ, j', i', ι'). Let the anchor (κ, j', i', ι') 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, (κ, j, i, ι) along with this pairwise alignment is assigned to the anchored alignment anchored by (κ, j', i', ι'). Otherwise, (κ, j, i, ι) 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 κ 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 κ 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 multipe 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 column as prospective erroneous bases. A prospective erroneous base is represented as a tuple $(g, x, 0)$ if it is at position x of the read at the up-stream end of the paired-end read r_g. It is represented as $(g, x, 1)$ if it is at position x of the read at the down-stream end of the paired-end read r_g. In our implementation, an identifier for the prospective erroneous position is also created and attached to the tuple. This identifier serves as a simple key that links all the prospective erroneous bases for this position together; it is also used for distributing the tuples for this position to the same computing node. For convenience, we denote the identifier attached to a prospective erroneous base (g, x, ι)—and thus the corresponding prospective erroneous position—by $\varphi(g, x, \iota)$.

Step 3: Mapping erroneous bases to candidate corrections

The core idea of using k-mers to correct errors by existing methods (Heo *et al.*, 2014; Ilie and Molnar, 2013; Li, 2015) is to link the erroneous k-mers (i.e. those having low frequency) to the closest solid k-mer (i.e. those having high frequency), and correcting the former based on the latter. However, this way of using k-mers does not guarantee the full inclusion of reads for error correction, due to both sequenced errors and the choice of k-mers. The sequenced errors give rise to mutated k-mers. Thus, the full inclusion of reads containing the errors cannot be guaranteed. 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. We present the detailed explanation in the supplementary Figure S2.

Our method, MEC, is less sensitive to these issues. In particular, in the second-layer MapReduce, all paired-end reads covering each prospective erroneous position are considered. From the previous step, if $\varphi(g, x, \iota) = \varphi(g', x', \iota)$ we know (g, x, ι) and (g', x', ι) are prospective erroneous bases for the same prospective erroneous position, even when (g, x, ι) is from a group G_κ and (g', x', ι) is from a different group $G_{\kappa'}$. Thus, the prospective erroneous bases identified by different groups for the same prospective erroneous position can be easily collected together.

Given a prospective erroneous position and the prospective erroneous bases at this position, the value that is most common among these prospective erroneous bases is identified as the reference value (i.e. the reference genome is hypothesized to have this value at this position). If there are more than one most-common value, one is picked arbitrarily as the reference value. The prospective erroneous bases that have a value different from the reference value are 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 $\hat{j}$ 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:

$$L_{x/x_0} = \log \frac{\prod^j I(\hat{j} = x) * p_j + I(\hat{j} \neq x) * (1 - p_j)/3}{\prod^j I(\hat{j} = x_0) * p_j + I(\hat{j} \neq x_0) * (1 - p_j)/3}$$

Take Figure 2 as an example. The erroneous bases "0" and "1" in graph$_1$ will be replaced by the reference value. The base "2" in graph$_2$, but not "0" and "1", will be replaced by the reference value. And the base "0" in graph$_3$ 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

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

Data set	Genome name	Genome size (mbp†)	Read length (bp‡)	Coverage	Number of paired-end reads	Per base error rate (%)	Per position error rate (%)
R1	S. aueus	2.8	101	46.3×	1,294,104	1.17	28.1
R2	R. sphaeroides	4.6	101	33.6×	766,646	1.28	30.0
R3	H. sapiens 14	88.3∗	101	38.3×	16,757,120	0.86	22.5
R4	B. impatiens	249.2	124	150.8×	303,118,594	0.96	27.4
D1	E.Coli	4.6	101	30.0×	689,927	1.35	34.2
D2	S.cerevisiae	12.4	101	60.2×	3,599,533	1.53	58.8
D3	H.sapiens 22	41.8∗	101	30.0×	6,209,209	1.47	36.1
D4			150	60.0×	8,361,240	1.59	64.7

† million base pairs; ‡ base pair; ∗ the genome regions marked as "N" are excluded.

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* (reca), 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_b^e/N_b, and per-position error rate is defined as N_p^e/N_p, where N_b^e is the number of erroneous bases, N_b is the number of all bases, N_p^e is the number of erroneous positions (i.e. the number of positions that are covered by at least one erroenous base), and N_p is the number of all positions. N_b is equal to $2 * N * L$ 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 (Salmela 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 is 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 2. Error-correction performance comparison between MEC, Coral, Racer, BLESS, BFC and SGA.

data	corrector	reca	prec	gain	pber(%)	pper(%)
R1	MEC	**0.893**	0.924	**0.874**	0.103	2.3
	Coral	0.803	0.858	0.728	0.210	12.6
	Racer	0.822	**0.929**	0.760	0.190	3.4
	BLESS	0.409	0.650	0.189	0.879	13.7
	BFC	0.817	0.927	0.753	0.196	8.3
	SGA	0.815	0.922	0.746	0.196	11.2
R2	MEC	**0.944**	0.963	**0.894**	0.120	2.9
	Coral	0.663	0.970	0.642	0.460	12.0
	Racer	0.921	0.949	0.872	0.150	4.0
	BLESS	0.722	0.989	0.714	0.340	9.0
	BFC	0.726	**0.990**	0.716	0.323	8.9
	SGA	0.641	0.985	0.631	0.460	12.0
R3	MEC	**0.874**	0.937	**0.814**	0.260	2.1
	Coral	0.690	0.779	0.495	0.430	8.0
	Racer	0.797	0.890	0.699	0.230	4.3
	BLESS	0.558	0.965	0.538	0.390	9.9
	BFC	0.641	0.966	0.613	0.319	7.1
	SGA	0.663	**0.967**	0.641	0.310	6.9
R4	MEC	**0.836**	**0.885**	**0.746**	0.271	3.1
	Coral	-	-	-	-	-
	Racer	0.541	0.703	0.313	0.484	8.3
	BLESS	0.018	0.003	-0.517	0.862	11.2
	BFC	0.457	0.636	0.195	0.607	9.1
	SGA	0.690	0.823	0.542	0.289	6.5
D1	MEC	**0.999**	0.996	**0.995**	0.007	0.075
	Coral	0.998	0.986	0.984	0.024	0.23
	Racer	0.998	0.981	0.980	0.032	0.66
	BLESS	0.980	0.998	0.979	0.033	0.89
	BFC	0.991	**0.999**	0.990	0.016	0.41
	SGA	0.990	**0.999**	0.989	0.016	0.41
D2	MEC	**0.997**	0.984	**0.983**	0.031	0.35
	Coral	0.995	0.954	0.947	0.081	1.1
	Racer	0.994	0.957	0.949	0.077	2.7
	BLESS	0.984	0.997	0.981	0.029	1.5
	BFC	0.970	0.997	0.968	0.043	2.2
	SGA	0.958	**0.998**	0.956	0.067	2.8
D3	MEC	**0.987**	0.915	**0.896**	0.340	1.7
	Coral	0.973	0.762	0.670	0.510	5.0
	Racer	0.880	0.466	-0.130	1.800	24.0
	BLESS	0.881	0.892	0.774	0.350	8.0
	BFC	0.883	0.907	0.792	0.340	7.9
	SGA	0.854	**0.957**	0.815	0.290	6.7
D4	MEC	**0.996**	0.939	**0.928**	0.281	1.6
	Coral	0.971	0.783	0.702	0.467	5.0
	Racer	0.883	0.487	-0.017	1.610	31.2
	BLESS	0.909	0.897	0.795	0.328	7.6
	BFC	0.891	0.889	0.819	0.316	8.9
	SGA	0.846	**0.965**	0.807	0.297	12.1

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.

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 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.

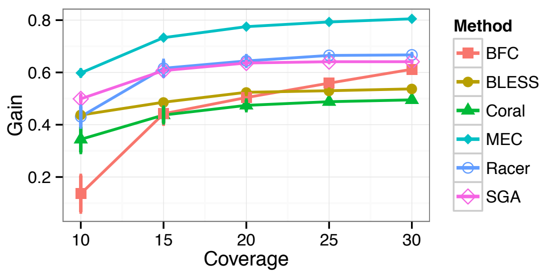

Fig. 3. The *gain* of MEC, Coral, Racer, BLESS, BFC, and SGA on subset reads of R3 having various coverage.

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.

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 X5690 3.47GHz CPUs and 96G 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.

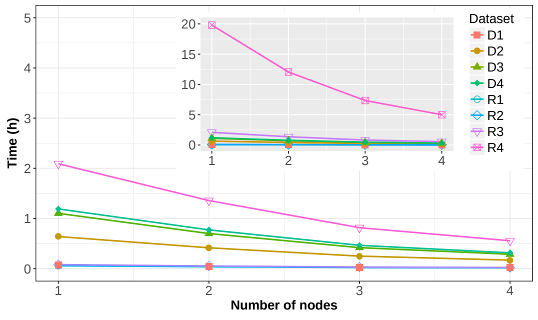

Fig. 4. The execution time of MEC for the eight datasets with various number of worker nodes.

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

Allam,A., Kalnis,P. and Solovyev,V. (2015) Karect: Accurate correction of substitution, insertion and deletion errors for next-generation sequencing data. *Bioinformatics,* **31** (21), 3421–3428.

Compeau,Phillip,E.C., Pevzner,P.A. and Tesler,G. (2011) How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology,* **29** (11), 987–91.

Dean,J. and Ghemawat,S. (2008) MapReduce: Simplified data processing on large clusters. *Communications of the ACM,* **51** (1), 107–113.

Fujimoto,M.S., Bodily,P.M., Okuda,N., Clement,M.J. and Snell,Q. (2014) Effects of error-correction of heterozygous next-generation sequencing data. *BMC Bioinformatics,* **15** (7), S3.

Heo,Y., Wu,X.L., Chen,D., Ma,J. and Hwu,W.M. (2014) BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics,* **30**

(10), 1354–1362.

Ilie,L., Fazayeli,F. and Ilie,S. (2011) HiTEC: Accurate error correction in high-throughput sequencing data. *Bioinformatics,* **27** (3), 295–302.

Ilie,L. and Molnar,M. (2013) Racer: Rapid and accurate correction of errors in reads. *Bioinformatics,* **29** (19), 2490–2493.

International Human Genome Sequencing Consortium (2004) Finishing the euchromatic sequence of the human genome. *Nature,* **431**, 931–945.

Kao,W.C., Chan,A.H. and Song,Y.S. (2011) ECHO: A reference-free short-read error correction algorithm. *Genome Research,* **21**, 1181–1192.

Kelley,D.R., Schatz,M.C. and Salzberg,S.L. (2010) Quake: Quality-aware detection and correction of sequencing errors. *Genome Biology,* **11** (11), R116.

Li,H. (2015) BFC: Correcting Illumina sequencing errors. *Bioinformatics,* **31** (17), 2885–2887.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics,* **25** (14), 1754–1760.

Liu,Y., Schmidt,B. and Maskell,D.L. (2011) DecGPU: Distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics,* **12**, 85.

Liu,Y., Schröder,J. and Schmidt,B. (2013) Musket: A multistage k-mer spectrum-based error corrector for Illumina sequence data. *Bioinformatics,* **29** (3), 308–315.

Mardis,E.R. (2013) Next-generation sequencing platforms. *Annual Review of Analytical Chemistry,* **6**, 287–303.

McElroy,K.E., Luciani,F. and Thomas,T. (2012) GemSIM: General, error-model based simulator of next-generation sequencing data. *BMC Genomics,* **13**, 74.

Medvedev,P., Scott,E., Kakaradov,B. and Pevzner,P. (2011) Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics,* **27** (13), i137–i141.

Molnar,M. and Ilie,L. (2014) Correcting Illumina data. *Briefings in Bioinformatics,* **16** (4), 588–599.

Pevzner,P.A., Tang,H. and Waterman,M.S. (2001) An Eulerian path approach to DNA fragment assembly. *Proceedings of the Natational Academy of Sciences of the United States of America,* **98**, 9748–9753.

Ross,M.G., Russ,C., Costello,M., Hollinger,A., Lennon,N.J., Hegarty,R., Nusbaum,C. and Jaffe,D.B. (2013) Characterizing and measuring bias in sequence data. *Genome Biology,* **14** (5), R51.

Salmela,L. (2010) Correction of sequencing errors in a mixed set of reads. *Bioinformatics,* **26** (12), 1284–1290.

Salmela,L. and Schröder,J. (2011) Correcting errors in short reads by multiple alignments. *Bioinformatics,* **27** (11), 1455–1461.

Salzberg,S.L., Phillippy,A.M., Zimin,A., Puiu,D., Magoc,T., Koren,S., Treangen,T.J., Schatz,M.C., Delcher,A.L., Roberts,M., Marcais,G., Pop,M. and Yorke,J.A. (2011) GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research,* **22** (3), 557–567.

Sameith,K., Roscito,J.G. and Hiller,M. (2016) Iterative error correction of long sequencing reads maximizes accuracy and improves contig assembly. *Briefings in Bioinformatics,* .

Schröder,J., Schröder,H., Puglisi,S.J., Sinha,R. and Schmidt,B. (2009) SHREC: A short-read error correction method. *Bioinformatics,* **25** (17), 2157–2163.

Sheikhizadeh,S. and de Ridder,D. (2015) ACE: Accurate correction of errors using K-mer tries. *Bioinformatics,* **31** (19), 3216–3218.

Simpson,J.T. and Durbin,R. (2012) Efficient de novo assembly of large genomes using compressed data structures. *Genome Research,* **22** (3), 549–556.

The 1000 Genomes Project Consortium (2010) A map of human genome variation from population-scale sequencing. *Nature,* **467**, 1061–1073.

Yang,X., Chockalingam,S.P. and Aluru,S. (2013) A survey of error-correction methods for next-generation sequencing. *Briefings in Bioinformtics,* **14** (1), 56–66.

Yang,X., Dorman,K.S. and Aluru,S. (2010) Reptile: Representative tiling for short read error correction. *Bioinformatics,* **26**, 2526–2533.

Zaharia,M., Chowdhury,M., Franklin,M.J., Shenker,S. and Stoica,I. (2010) Spark: cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* HotCloud'10 USENIX Association, Berkeley, CA, USA.

Zerbino,D.R. and Birney,E. (2008) Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research,* **18**, 821–829.

SUPPLEMENTARY NOTES

Vol. 00 no. 00 2016
Pages 1–7

MapReduce for accurate error correction of next-generation sequencing data

Liang Zhao[1,2], Qingfeng Chen[1], Wencui Li[2], Peng Jiang[1], Limsoon Wong[3, *], Jinyan Li[4, *]

[1]School of Computing and Electronic Information, Guangxi University, Nanning 530004, China.
[2]Taihe Hospital, Hubei University of Medicine, Hubei 442000, China.
[3]School 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.

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 (κ, j, i, ι) to denote an occurrence of the k-mer, κ, starting at the ith position of the paired-end read r_j having mate index ι. The indices i and j are 0-based. For example, suppose the input set of paired-end reads is $R = \{$AGTTCG$\cdots$GGCTCA, TTCGAC$\cdots$GAGGAC, TCTCAG$\cdots$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 κ to a group G_κ, which corresponds to a node in the computing cluster. The set of tuples in group G_κ is denoted by $G(\kappa)$. Some groups $G(\kappa)$ are much larger than other groups. In such a case, the k-mer κ 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. It remains to describe the detail of the alignment algorithm mentioned in the main context. Let (κ, i, a, ι) and (κ, j, b, ι), $i \neq j$, be two occurrences of the k-mer κ in the group G_κ. An alignment between r_i and r_j respecting these two occurrences of κ 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 $r_i = R_1^i \cdots R_2^i$ and $r_j = R_1^j \cdots R_2^j$ be two paired-end reads that we want to align with respect to a k-mer κ (shown in purple in the figure) that occurs at position a in R_1^i (i.e. at position a of the read at the up-stream end of r_i) and position b in R_1^j (i.e. at position b of the read at the up-stream end of r_j).

First, R_1^i and R_1^j are aligned by setting the two said occurrences of κ (shown in purple) directly on top of one another. That is, every position $h + a$ in R_1^i is aligned to the position $h + b$ in R_1^i. This causes a segment of length $b - a$ at the end of R_1^i (or R_1^j 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^i and R_2^j are aligned by sliding one on top of the other by y bases. That is, every position h in R_2^i is aligned to the position $h + y$ in R_2^j. By convention, y is negative if the start of R_2^i is aligned to a position down-stream of the start of R_2^j (i.e. a segment of length y at the start of R_2^j sticks out into the insert region of r_i); and y is positive otherwise (and a segment of length y at the start of R_2^i 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 σ_0^2 of insert size of the paired-end reads in the input set R, and define the insertion penalty as $s = c_0 * (x+y)^2/\sigma_0^2$. 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

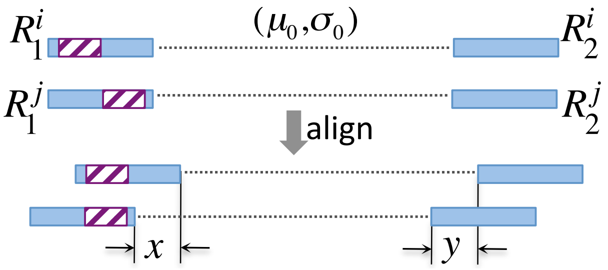

Fig. S1. An illustration of the penalty-aware alignment of paired-end reads. The mean and standard deviation of the insert size of the paired-end reads are μ_0 and σ_0 respectively.

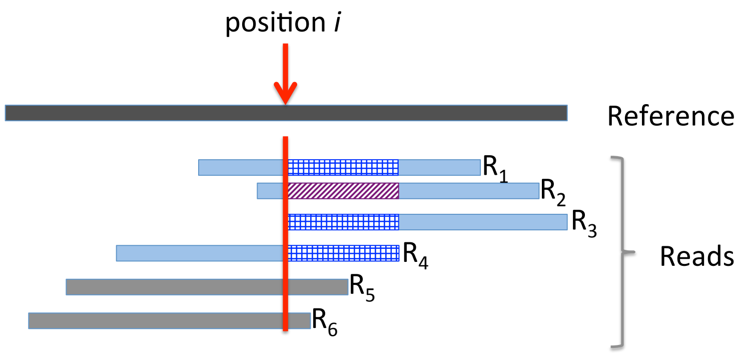

Fig. S2. An illustration of incomplete inclusion of reads by using k-mers. The k-mer of interest is in blue, while the mutated one (having errors) is in purple.

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 (≤ 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 κ is not mapped to the group G_κ if it contains an error where κ is located. E.g. if the 3-mer AGT of the paired-end read AGTTCG$\cdots$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, ι), 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, ι) is denoted by $\varphi(g, x, \iota)$. However, the value of $\varphi(g, x, \iota)$—let us call this a position reference—is generated independently on different computing nodes. Thus the same position gets a different position reference on different computing nodes. We need to map these different position references for the same position to the same value. Let D_u denote the set of entries of the form (position reference, positions) generated at computing node u, where each entry $(\varphi, \{p_1, \ldots, p_m\})$ means each positions $p_i = (g_i, x_i, \iota_i)$, $i = 1 \ldots m$, has been mapped to the position reference φ. The total size of D_1, D_2, ..., 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-joint of D_u produced by each computing node u. A self-join of D_u is defined as

$$\min\{(\varphi_1, \bigcup_{(\varphi_2, P_2) \in D_u, P_1 \cap P_2 \neq \{\}} P_2) \mid (\varphi_1, P_1) \in D_u\}.$$

Here, $(\varphi_i, P_i) \in minR$ if and only if for every $(\varphi_j, P_j) \in R$ such that $P_i = P_j$, it is the case that $\varphi_i \leq \varphi_j$. In other words, all entries in D_u 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-joint of two D_u and D_v, until only one remains. Based on the self-joint, it can be guaranteed that the intersection of any two entries of D_u or D_v is empty. Hence, it is safe to merge each entry of D_v to D_u progressively in order to achieve the binary-joint of D_u and D_v. To make it more efficient, we create indies for all the Ps against their φs, say $I(P)$. For each (φ_v, P_v) of D_v, we check the existence of P_v^i against $I(P)$. In case there exists a (φ_u, P_u), such that $P_u^j \in P_u, P_u^j = P_v^i$, we output $(\varphi_u, P_u \cup P_v)$. This process is repeated until all entries in D_v are exhausted; at this point, all remaining entries in D_u are copied to the output. If D_u 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_v.

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 allel 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. Suppose reasonably that the per base error rate of the sequencing technology is 1%, and the substitution chance of the other three wrong base values is equal. Then the chance that a given position in a read has a specific wrong value is $1/3 * 1\%$, and the chance that this same position has the same specific wrong value in k reads is $(1/3 * 1\%)^k$. Suppose the genome size is 1 billion bases, and the sequencing coverage is 30x. Then the expected number of positions having exactly three reads bearing the same specific wrong value for it is $1,000,000,000 * 30 * (1/3 * 1\%)^3 \simeq 1,100$; that of having exactly four reads bearing the same erroneous base is ~ 4; and that of having exactly five reads bearing the same erroneous base is ~ 0.01.

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, ι, β), where β is the corrected base. We use g as the key, and the set of (x, ι, β) 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 every group only contains 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. E.g., 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 applicble 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

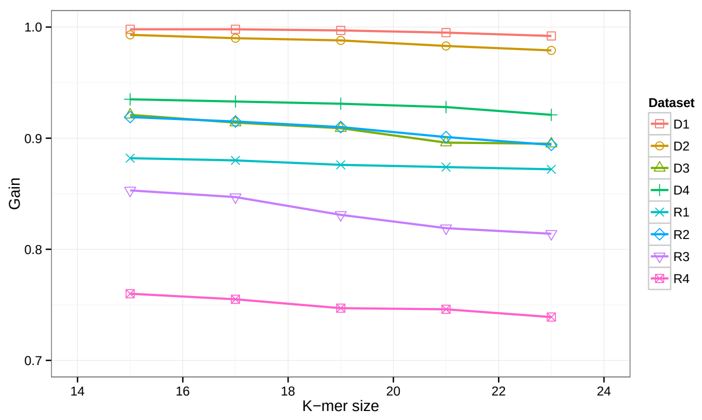

Fig. S3. The *gain* of MEC on the eight datasets with various size of k-mers.

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

k	corrector	reca				prec				gain				pber(%)				pper(%)			
		R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
15	BLESS	0.031	0.089	0.055	0.002	0.308	0.571	0.533	0.001	0.013	0.051	0.042	-0.862	1.120	1.202	0.845	0.973	27.5	29.3	21.7	27.6
	BFC	0.051	0.059	0.050	0.039	0.356	0.521	0.513	0.001	0.047	0.067	0.043	0.019	1.117	1.195	0.825	0.923	27.3	29.4	21.5	26.7
	SGA	0.057	0.025	0.047	0.053	0.241	0.253	0.300	0.241	0.008	0.012	-0.063	0.004	1.067	1.140	0.903	0.894	26.9	29.1	23.9	26.2
17	BLESS	0.120	0.241	0.177	0.006	0.443	0.701	0.695	0.002	0.063	0.205	0.165	-0.823	0.932	1.105	0.767	0.921	23.5	24.2	19.0	22.7
	BFC	0.243	0.214	0.189	0.135	0.413	0.635	0.617	0.001	0.198	0.201	0.163	0.045	0.906	1.024	0.751	0.882	22.9	24.1	19.1	23.5
	SGA	0.386	0.274	0.337	0.349	0.485	0.530	0.658	0.517	0.153	0.177	0.162	0.139	0.786	0.934	0.745	0.807	23.2	23.7	18.8	21.4
19	BLESS	0.209	0.372	0.265	0.009	0.497	0.774	0.762	0.002	0.097	0.332	0.252	-0.767	0.906	0.529	0.627	0.903	19.7	16.4	17.0	19.2
	BFC	0.446	0.376	0.307	0.214	0.789	0.854	0.786	0.487	0.337	0.325	0.297	0.076	0.534	0.591	0.632	0.476	14.9	15.4	16.1	17.3
	SGA	0.623	0.467	0.553	0.572	0.787	0.842	0.899	0.676	0.582	0.501	0.491	0.402	0.317	0.472	0.437	0.429	13.9	15.2	10.5	9.3
21	BLESS	0.317	0.614	0.437	0.012	0.517	0.815	0.793	0.002	0.127	0.526	0.417	-0.611	0.892	0.351	0.453	0.882	14.2	11.7	11.9	13.2
	BFC	0.717	0.603	0.510	0.337	0.838	0.901	0.837	0.510	0.609	0.593	0.507	0.137	0.252	0.384	0.403	0.292	9.3	10.7	9.5	11.2
	SGA	0.716	0.547	0.566	0.601	0.841	0.914	0.907	0.739	0.656	0.572	0.538	0.479	0.237	0.391	0.376	0.352	12.3	13.9	8.3	7.6
23	BLESS	0.410	0.724	0.523	0.019	0.649	0.987	0.943	0.003	0.183	0.698	0.527	-0.531	0.872	0.327	0.405	0.878	13.8	9.8	10.2	11.7
	BFC	0.793	0.705	0.637	0.454	0.912	0.986	0.957	0.627	0.739	0.705	0.601	0.188	0.207	0.332	0.327	0.208	8.7	9.2	8.0	9.9
	SGA	0.793	0.631	0.652	0.684	0.913	0.976	0.957	0.814	0.726	0.625	0.630	0.532	0.201	0.482	0.320	0.307	11.7	13.1	7.4	6.9

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

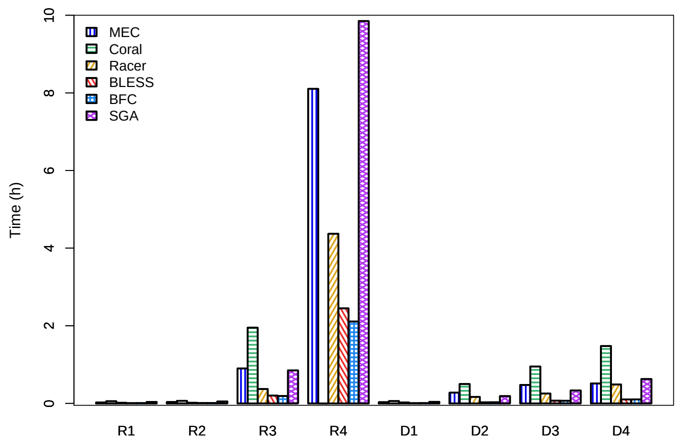

Fig. S4. Speed comparison between the error-correction methods. The result of Coral is not obtained for some datasets because the amount of RAM required is beyond the available limit.

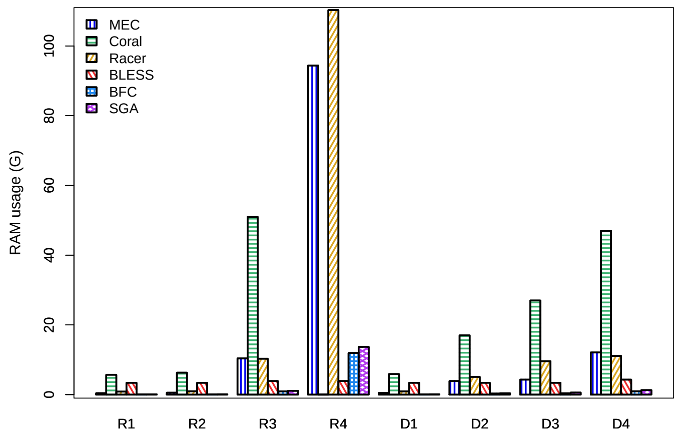

Fig. S5. Memory usage comparison between the error-correction methods. The result of Coral is not obtained for some datasets because the amount of RAM required is beyond the available limit.

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.

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

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.

Corrector	N50		NGA50		max length	
	R1	R2	R1	R2	R1	R2
MEC	2,236	2,865	2,997	3,287	39,972	43,594
Coral	613	547	1,295	1,175	19,183	22,029
Racer	3,356	3,979	1,479	1,656	31,278	35,129
BLESS	886	979	1,378	1,437	30,037	29,305
BFC	996	1,271	1,464	1,773	28,634	30,217
SGA	745	674	1,736	1,275	23,821	21,924

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. aueus: `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`;
- B. impatiens: `http://gage.cbcb.umd.edu/data/Bombus_impatiens`.

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
R1	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
	sga 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
R2	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	

	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
R4	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.fastq -read2 R4_2.fastq -prefix R4_bless -kmerlength 23 bfc -s 250m -t12 R4_1_2.fastq > R4_bfc_1_2.fastq 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
D1	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 4600000 bless -read1 D1_1.fastq -read2 D1_2.fastq -prefix D1_bless -kmerlength 23 bfc -s 5m -t12 D1_1_2.fastq > D1_bfc_1_2.fastq 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
D2	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 bfc -s 13m -t12 D2_1_2.fastq > D2_bfc_1_2.fastq 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
D3	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 bfc -s 42m -t12 D3_1_2.fastq > D3_bfc_1_2.fastq 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
D4	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 bfc -s 42m -t12 D4_1_2.fastq > D4_bfc_1_2.fastq 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

Fujimoto, M. S., Bodily, P. M., Okuda, N., Clement, M. J., and Snell, Q. (2014). Effects of error-correction of heterozygous next-generation sequencing data. *BMC Bioinformatics*, **15**(7), S3.

Gurevich, A., Saveliev, V., Vyahhi, N., and Tesler, G. (2013). QUAST: Quality assessment tool for genome assemblies. *Bioinformatics*, **29**(8), 1072–1075.

Kelley, D. R., Schatz, M. C., and Salzberg, S. L. (2010). Quake: Quality-aware detection and correction of sequencing errors. *Genome Biology*, **11**(11), R116.

Li, R., Zhu, H., Ruan, J., Qian, W., *et al.* (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, **20**, 265–272.

Miller, J. R., Koren, S., and Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. *Genomics*, **95**(6), 315–27.

Salzberg, S. L., Phillippy, A. M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T. J., Schatz, M. C., Delcher, A. L., Roberts, M., Marcais, G., Pop, M., and Yorke, J. A. (2011). GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, **22**(3), 557–567.

Yang, X., Chockalingam, S. P., and Aluru, S. (2013). A survey of error-correction methods for next-generation sequencing. *Briefings in Bioinformtics*, **14**(1), 56–66.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA. USENIX Association.

Zerbino, D. R. and Birney, E. (2008). Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, **18**, 821–829.