BIOINFORMATICS ORIGINAL PAPER

Vol. 29 no. 3 2013, pages 308–315
doi:10.1093/bioinformatics/bts690

Genome analysis

Advance Access publication November 29, 2012

Musket: a multistage *k*-mer spectrum-based error corrector for Illumina sequence data

Yongchao Liu[1,*], Jan Schröder[2] and Bertil Schmidt[1,*]

[1]Institut für Informatik, Johannes Gutenberg Universität Mainz, Mainz 55099, Germany and [2]Department of Computing and Information Systems, The University of Melbourne, Parkville 3010, Australia

Associate Editor: Martin Bishop

ABSTRACT

Motivation: The imperfect sequence data produced by next-generation sequencing technologies have motivated the development of a number of short-read error correctors in recent years. The majority of methods focus on the correction of substitution errors, which are the dominant error source in data produced by Illumina sequencing technology. Existing tools either score high in terms of recall or precision but not consistently high in terms of both measures.

Results: In this article, we present Musket, an efficient multistage *k*-mer-based corrector for Illumina short-read data. We use the *k*-mer spectrum approach and introduce three correction techniques in a multistage workflow: two-sided conservative correction, one-sided aggressive correction and voting-based refinement. Our performance evaluation results, in terms of correction quality and *de novo* genome assembly measures, reveal that Musket is consistently one of the top performing correctors. In addition, Musket is multi-threaded using a master–slave model and demonstrates superior parallel scalability compared with all other evaluated correctors as well as a highly competitive overall execution time.

Availability: Musket is available at http://musket.sourceforge.net.

Contact: liuy@uni-mainz.de or bertil.schmidt@uni-mainz.de

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 20, 2012; revised on October 24, 2012; accepted on November 26, 2012

1 INTRODUCTION

The emergence and rapid progress of next-generation sequencing (NGS) technologies has enabled the high-throughput production of short DNA sequences (reads) at low cost. The ever increasing throughput and decreasing cost has significantly altered the landscape of whole-genome sequencing, providing an opportunity for scientists to initiate whole-genome sequencing projects for almost any organism, including those whose genomes span billions of base pairs, such as the giant panda (Li *et al.*, 2010a) and humans (Li *et al.*, 2010b). Many NGS sequencing technologies have been developed, among which Illumina is most widely used. However, reads produced from NGS platforms are never perfect and can contain various types of sequencing errors, including substitutions and indels (insertions or deletions). These sequencing errors complicate data processing for many biological applications such as *de novo* genome assembly (Salzberg *et al.*, 2012) and short-read mapping (Liu *et al.*, 2012; Salmela and Schröder, 2011).

To improve data quality, one frequently used approach is to trim reads from the more error-prone 3′–ends, which unfortunately results in loss of information (Yang *et al.*, 2012). This has ignited the interest of researchers in conceiving more sophisticated algorithms to detect and correct sequencing errors in NGS data, for example, Schröder *et al.* (2009). As substitution is the dominant error type for data produced by Illumina sequencing technology, most approaches focus on correcting this type of errors (Yang *et al.*, 2012). The core of substitution-error-based methods is to compute consensus bases using the highly redundant coverage information. When a sequencing error occurs in a read that originated from a certain position on the genome, all reads covering the erroneous position could be piled up to compute the consensus base. Considering that sequencing errors are generally random and infrequent, this consensus base is likely to be correct. However, as we assume that the source genome is unknown beforehand, we can neither determine the read locations on the genome nor the correctness of reads directly. Instead, reads that cover overlapping genomic positions can be inferred by assuming that they typically share common substrings. Furthermore, we can approximate the source genome using a *k*-mer spectrum, which was first introduced by Pevzner *et al.* (2001). Given a dataset with sufficient coverage of a genome, the *k*-mer spectrum is defined as the set of all *k*-mers in the dataset, where the *k*-mers whose multiplicity exceeds a coverage *cut-off* are deemed to be trusted and otherwise, untrusted. The first *k*-mer spectrum-based corrector was proposed by Pevzner *et al.* (2001), using an iterative spectral alignment problem approach. CUDA-EC (Shi *et al.*, 2010) and DecGPU (Liu *et al.*, 2011a) accelerated this spectral alignment problem approach using graphics processing unit computing. To further improve correcting quality, Chaisson *et al.* (2004) introduced a dynamic programming approach that corrects errors by minimizing edit distances. The SOAP corrector (Li *et al.*, 2010b) adopted a similar approach. Quake (Kelley *et al.*, 2010) introduced a probabilistic model derived from base quality scores of reads. For a *k*-mer, Quake accumulates the correctness probability of all its occurrences and defines this sum as the multiplicity of the *k*-mer, instead of just the number of occurrences. Reptile (Yang *et al.*, 2010) relies on a Hamming graph to resolve ambiguities for a tile-based correction, whereas Hammer (Medvedev *et al.*, 2011) combines the Hamming graph with a probabilistic model to deal with datasets with non-uniform coverage. HiTEC (Ilie *et al.*, 2011) supports multiple *k*-mer sizes in a single launch

*To whom correspondence should be addressed.

and relies on witness clusters that are constructed from suffix arrays. SGA (Simpson and Durbin, 2012) uses a memory-efficient Burrows Wheeler transform (Burrows and Wheeler, 1994) and an FM-index (Ferragina and Manzini, 2005) to represent the *k*-mer spectrum. In addition to *k*-mer spectrum-based approaches, some correctors based on other techniques have been developed. SHREC (Schröder *et al*., 2009) uses a generalized suffix trie to detect and correct substitution errors, which is further extended by Hybrid SHREC (Salmela, 2010) to additionally correct indels. Coral (Salmela and Schröder, 2011) and ECHO (Kao *et al*., 2011) are two correctors that use the concept of multiple sequence alignment, using *k*-mers as seeds. Coral corrects errors by constructing consensus sequences from multiple alignments, and ECHO by computing consensus bases using a maximum-a-posterior estimate over position specific substitution matrices.

In this article, we present Musket (multistage *K*-mer spectrum-based corrector), an efficient substitution-error-based corrector for Illumina sequence data based on a *k*-mer spectrum approach (Pevzner *et al*., 2001). We introduce three techniques, namely, two-sided conservative correction, one-sided aggressive correction and voting-based refinement, to form a multistage correction workflow. The performance of Musket is evaluated using both simulated and real datasets for short-read data originating from small-sized (*Escherichia coli*), medium-sized (human chromosome 21) and large-sized (human chromosome 1) genomes. In terms of correction quality, Musket is consistently one of the top performing correctors compared with HiTEC, SGA, SHREC, Coral, Quake, Reptile and DecGPU. In terms of *de novo* genome assembly using the SGA assembler (Simpson and Durbin, 2012), Musket yields better performance than all other evaluated correctors with respect to some commonly used metrics. In addition, Musket provides support for multi-threading using a master–slave model and demonstrates superior scalability on a shared-memory multi-CPU workstation, as well as highly competitive overall execution speed.

2 METHODS

Musket consists of two stages: *k*-mer spectrum construction and error correction. For *k*-mer spectrum construction, Musket counts the number of occurrences of all non-unique *k*-mers using a combination of a Bloom filter (Bloom, 1970) and a hash table. This *k*-mer counting approach has been introduced by Melsted and Pritchard (2011) and has been shown to be able to significantly reduce the memory footprint for large datasets. Musket automatically estimates the coverage cut-off from the coverage histogram of all non-unique *k*-mers. For error correction, Musket introduces three techniques, namely, two-sided conservative correction, one-sided aggressive correction and voting-based refinement. In addition, Musket has been parallelized using multi-threading to benefit from the compute power of common multi-CPU systems.

2.1 *K*-mer spectrum construction

2.1.1 Parallelized k*-mer counting* *K*-mer counting based on a Bloom filter generally comprises three steps. Step 1 filters out as many unique *k*-mers as possible using a Bloom filter, storing all non-unique *k*-mers in a hash table. Because of the false-positive probability of a Bloom filter, some unique *k*-mers are likely to exist in the hash table. Step 2 computes the multiplicity of each *k*-mer in the hash table to determine the unique *k*-mers that are occasionally stored in the hash table.

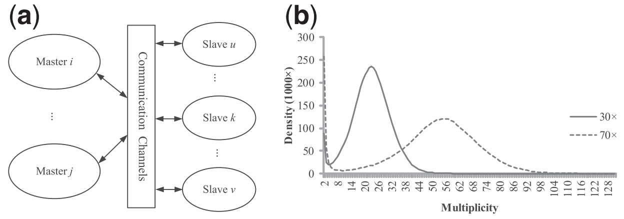

Fig. 1. (**a**) The master–slave model used in Musket ($0 \leq I < j <$ #masters and $0 \leq u < k < v <$ #slaves). (**b**) The *k*-mer coverage histograms of two datasets with different coverage

Step 3 removes all unique *k*-mers from the hash table, leaving all non-unique *k*-mers in the hash table. We have parallelized this *k*-mer counting using multi-threading to leverage the compute power of multiple CPU cores.

Our parallelization strategy uses the master–slave model, a typical parallelization paradigm in which masters are dedicated to task distribution, and slaves are assigned to work on individual tasks. Typically, there is only one master in many applications using the master–slave model. Instead, we have used multiple masters in Musket (Fig. 1a) to avoid the case in which the task distribution becomes a new bottleneck as the number of slaves grows larger. In general, the model is configured to have more slaves than masters. However, to maximize performance, the slave-to-master ratio must be carefully chosen. Our experiments indicate that a 3:1 ratio is good for Musket. In the following, we describe in detail the tasks that the masters and slaves perform in each step.

In Step 1, the masters fetch reads in parallel from the input file, where mutually exclusive accesses to the file are guaranteed by locks, and then distribute all *k*-mers in the reads to the slaves. The distribution of a *k*-mer requires the destination slave to be determined deterministically and efficiently from the *k*-mer itself. Thus, we compute the destination slave from the canonical *k*-mer, which is defined as the smaller numerical representation of the *k*-mer and its reverse complement. Once the destination slave is determined, the canonical *k*-mer is transferred to the slave through the corresponding communication channels. All slaves listen to their corresponding communication channels and start processing *k*-mers once they arrive. Each slave holds a local Bloom filter and a local hash table to capture all non-unique *k*-mers as well as filter out as many unique *k*-mers. When a *k*-mer arrives, the slave performs membership lookup in the local Bloom filter for the *k*-mer. If the *k*-mer is queried to exist in the Bloom filter, the slave inserts it into the local hash table because it is likely to have more than one occurrence and into the Bloom filter, otherwise. After completing this step, all non-unique *k*-mers are definitely stored in all local hash tables of all slaves, and the number of unique *k*-mers occasionally existing in all hash tables depends on the false-positive probability rate of all local Bloom filters. Because of the independence of the local Bloom filters and hash tables, this step does not require any synchronization between the slave threads, which greatly benefits efficiency.

In Step 2, the masters follow the same procedure as in Step 1, whereas all slaves listen to their corresponding communication channels and wait for the arrival of *k*-mers. Once a *k*-mer arrives, a slave queries the existence of the *k*-mer in the local hash table. If present, the multiplicity of the *k*-mer is increased by one. After completing this step, each slave holds the multiplicity of each *k*-mer stored in its hash table, which is used to determine the uniqueness of a *k*-mer. In Step 3, the masters are idle, and each slave deletes all unique *k*-mers from its hash table. After finishing this step, each slave holds a partition of the set of all non-unique *k*-mers in the input reads. For Steps 1 and 2, a hybrid combination of *Pthreads* and OpenMP parallel programming models is used to implement the master–slave model. For Step 3, only OpenMP model is used.

2.1.2 Coverage cut-off estimation *K*-mer coverage histograms are frequently used to determine the coverage *cut-off* for a *k*-mer spectrum. A *k*-mer coverage histogram illustrates a mixture of two distributions: one for the coverage of likely correct *k*-mers and the other for spurious *k*-mers. In theory, the coverage of true *k*-mers follows a Poisson distribution, but the biases in Illumina sequencing add variance (Dohm *et al.*, 2008). Hence, the coverage of true *k*-mers can be modelled as a normal distribution or a mixture of multiple distributions (e.g. Quake models it by mixing a normal distribution and a Zeta distribution). The coverage of spurious *k*-mers can be modelled as a Poisson distribution (Chaisson *et al.*, 2009) or as a Gamma distribution (Kelley *et al.*, 2010).

Figure 1b shows two *k*-mer coverage histograms constructed from all non-unique 21mers in two real datasets, which are generated by randomly sampling reads from a high-coverage real dataset sequenced from an *E.coli K12 MG1665* sample, which has accession number ERR022075 in the NCBI sequence read archive (SRA), to form a 30× and 70× coverage of the *E.coli* genome, respectively. The two curves share similar trends. As the multiplicity increases, the density (i.e. the number of *k*-mers with the same multiplicity) goes down sharply and then climbs up and down following a bell shape, thus forming a valley at the beginning of each curve. The area around the valley is generally considered as the watershed to separate the true *k*-mers from the spurious ones. The *k*-mers distributed at the right of the valley are supposed to be true *k*-mers (trusted) and the ones on the opposite side to be spurious (untrusted). Some correctors have proposed methods to automatically estimate the coverage *cut-off*. HiTEC determines the *cut-off* by computing the minimal support that makes the expected number of correct witness pairs greater than that of incorrect ones for a specific witness. Quake calculates it by using the Broyden–Fletcher–Goldfarb–Shanno method implemented in R to compute the likelihood ratio of trusted *k*-mers to untrusted ones for various coverage values. However, the curve of a *k*-mer coverage distribution is not always smooth enough, and thus sometimes causes Quake to fail. After examining histograms from some datasets, we found that the multiplicity corresponding to the smallest density around the valley is an appropriate estimation of the *cut-off*. Hence, Musket chooses that multiplicity as the coverage *cut-off*, by default. In addition, Musket provides a parameter to allow users to specify the *cut-off*.

2.2 Error correction

The error correction stage adopts a multistage workflow (Fig. 2). The core of the workflow relies on three techniques: two-sided conservative correction, one-sided aggressive correction and voting-based refinement.

2.2.1 Two-sided conservative correction Our two-sided correction starts with the classification of trusted and untrusted bases for a read. If a base is covered by any trusted *k*-mer, the base is deemed to be trusted and untrusted, otherwise. The untrusted bases are considered as potential sequencing errors (only considering substitutions). The time complexity of this classification procedure is $O(L)$ for a L-length read, assuming the querying of a *k*-mer in the hash table takes constant time. For a sequencing error occurring at position i of a read of l bases, it causes up to $\min\{k, i, l-i\}$ erroneous *k*-mers. In this context, our two-sided correction conservatively assumes that there is at most one substitution error in any *k*-mer of a read—an assumption that is relaxed in later stages of the algorithm. Under this assumption, our two-sided correction aims to find a unique alternative base that makes all *k*-mers that cover position i trusted. In this case, we might need to examine up to k *k*-mers while correcting a single untrusted base, resulting in high computational overhead. Hence, Musket chooses to only evaluate both the leftmost and the rightmost *k*-mers that cover position i on the read, significantly improving speed. For a single base, if more than one alternative is found to make both the leftmost and the rightmost *k*-mers trusted, the base will keep unchanged as a result of ambiguity. For a read, the

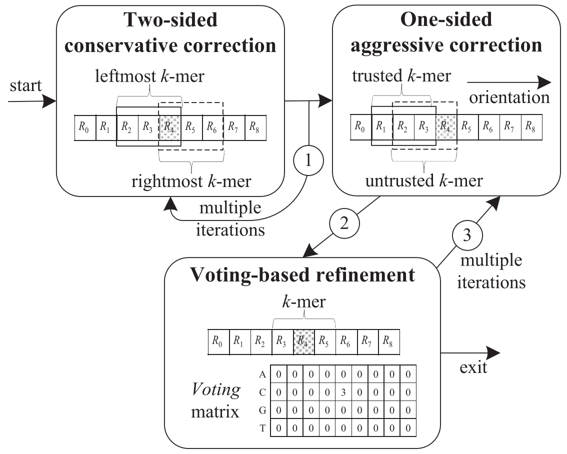

Fig. 2. Error correction workflow: (i) two-sided conservative correction is performed using multiple iterations; (ii) one-sided aggressive correction is directly followed by voting-based refinement; and (iii) the combination of one-sided correction and voting-based refinement is conducted in multiple iterations

two-sided correction will be executed for a fixed number of iterations (default = 2) or until no base change is made. As a constant number of *k*-mer membership queries are conducted for an untrusted base, the time complexity of the overall two-sided correction can be inferred as $O(L)$.

2.2.2 One-sided aggressive correction When two or more sequencing errors occur in a single *k*-mer, it is impossible to correct these errors using the two-sided correction. In this context, we propose a one-sided correction to aggressively correct errors in the case of more than one error occurring in a single *k*-mer. The core idea is as follows. Given a read R, define R_i to denote the base at position i of R, and $R_{i,k}$ to denote the *k*-mer starting at position i. If $R_{i,k}$ is trusted, but $R_{i+1,k}$ is untrusted, the base R_{i+k} is likely to be a sequencing error. Our one-sided correction aims to find an alternative of R_{i+k} that yields a trusted $R_{i+1,k}$. Unlike the two-sided correction, this correction selects the alternative, which makes the resulting trusted *k*-mer $R_{i+1,k}$ have the largest multiplicity, if more than one alternative is found.

Our one-sided correction begins with the location of trusted regions for a read. A trusted region means that every base in this region is trusted, thus having a minimum length of k. For each trusted region, error corrections are conducted towards each of the two orientations on the read. For each orientation, the error correction process does not stop until it either reaches a neighbouring trusted region or fails to correct the current base. This correction approach is effective but has the drawback that it strictly relies on the correctness of *k*-mer $R_{i,k}$. If $R_{i,k}$ does contain sequencing errors but is deemed to be trusted, this one-sided correction is possible to cause cumulative incorrect corrections, mutating a series of correct bases to incorrect ones. To reduce this negative effect, we have introduced two tricks: look-ahead validation and voting-based refinement. The look-ahead validation assesses the trustiness of a predefined maximal number (default = 2) of neighbouring *k*-mers that cover the base position at which a sequencing error likely occurs. If all evaluated *k*-mers are trusted for a certain alternative on that position, this alternative is reserved as one potential correction. The voting-based refinement is used after completing one-sided correction and will be described in more details in the following.

In addition, we use a constraint on the number N_c (default ≤ 4) of corrections that are allowed in any *k*-mer of a read. During the correcting process, we track the number of corrections that have been made in any *k*-mer. For a *k*-mer, once N_c exceeds the constraint, all corrections made in the k-mer will be disregarded. In Musket, the one-sided correction is conducted for each integer value from 1 to the maximal allowable number of corrections to confine the number of false-positive errors. The time complexity can also be inferred as $O(L)$, same as for the two-sided correction.

2.2.3 Voting-based refinement The voting-based refinement uses the same voting algorithm as used in DecGPU with the difference that Musket only considers the unique alternative base with the highest votes at a certain position. More details about the voting-algorithm can be obtained from Liu *et al.* (2011). This voting-based refinement is used because it introduces the fewest new errors, even though it does not correct as many errors as other correctors (see Section 3). Through our experiments, this approach does facilitate the reduction of the number of new errors from the one-sided correction. As the voting algorithm checks all possible base substitutions in any k-mer, the time complexity of this approach can be calculated as $O(k \cdot L)$.

3 RESULTS

We have evaluated the performance of Musket using both simulated and real short-read datasets from the following three perspectives: (i) correction quality; (ii) impact on *de novo* genome assembly quality; and (iii) speed, parallel scalability and memory consumption. Performance is compared with several publicly available correctors: HiTEC (v1.0.2), SGA (v0.9.18), SHREC (v2.2), Coral (v1.4), Quake (v0.3.1), Reptile (v1.1) and DecGPU (v1.0.6). For all correctors, we have used the default settings and disabled read trimming/discarding. For Coral, we have disabled the correction of indels. As Quake sometimes exceptionally exits, we have manually run each step of Quake in this article. All tests are conducted on a workstation with 2 six-core Intel Xeon X5650 2.67 GHz CPUs and 96 GB random access memory, running Linux (Ubuntu 12.04 LTS).

3.1 Correction quality assessment

Several previous publications (Ilie *et al.*, 2011; Liu *et al.*, 2011a; Schröder *et al.*, 2009; Yang *et al.*, 2010) assessed correction performance based on a binary read classifier that differentiates correct reads from incorrect ones. We have decided not to use this metric in our study because it does not reflect the ability to detect and correct individual erroneous bases or the risk of introducing new errors by wrongly mutating correct bases. Instead, we have evaluated the performance using a binary base classifier that takes into account how many erroneous bases are successfully corrected as well as how many new base errors are introduced.

Both simulated and real reads are used for assessment. For simulated reads, they are simulated from three references of varying complexity and length: the *E.coli K12 MG1665 strain* (NC_000913) of length 4 639 675 bases, human chromosome 21 (*Chr21*) of length 48 129 895 bases and human chromosome 1 (*Chr1*) of length 249 250 621 bases. As Quake relies on base quality scores, we further mimic the real quality scores for the simulated reads by extracting quality scores from related real datasets, where the quality scores of one read in a real dataset are entirely used for one read in a simulated dataset. For a simulated dataset with a uniform base error rate *err*, we define Q to denote the lowest quality score in the simulated dataset, where Q is calculated from *err* (deemed to be the probability that a base call is incorrect) following the PHRED definition (Ewing and Green, 1998). If a real quality score is smaller than Q, we replace the real one with Q. All simulated reads have the same length of 100 bases and contain only substitution errors. For real reads, two real datasets from *E.coli* are used (see the Supplementary Data).

Performance is measured in terms of recall, precision, F-score and Gain. We define TP (true positive) as the number of erroneous bases that are successfully corrected, FP (false positive) as the number of newly introduced errors (i.e. the number of correct bases that are changed to be erroneous) and FN (false negative) as the number of erroneous bases that failed to be successfully corrected, including the erroneous bases that keep unchanged and those that are changed to incorrect ones. Recall is calculated as $TP/(TP+FN)$, precision as $TP/(TP+FP)$, F-score as $2 \times \text{precision} \times \text{recall}/(\text{precision}+\text{recall})$ and Gain as $(TP-FP)/(TP+FN)$. The used definitions of TP, FP and FN are identical to Yang *et al.* (2012). They can also be derived from the four measures introduced in Liu *et al.* (2011a): correct corrections (CC), incorrect corrections (IC), errors unchanged (EU) and errors introduced (EI), where $TP=CC$, $FP=EI$ and $FN=IC+EU$. All recall, precision, F-score and Gain values in related tables have been multiplied by 100, and all best values have been highlighted in bold.

3.1.1 Evaluation using E.coli genome We have first evaluated all correctors using reads simulated from the small-sized *E.coli* genome. Three uniform base error rates (1, 2 and 3) and three coverage values (30, 70 and 100×) are used. The base quality scores are extracted from the high-coverage ERR022075 real dataset (aforementioned).

Supplementary Tables S1–S4 show the correction quality for simulated datasets from the *E.coli* genome in terms of recall, precision, F-score and Gain, respectively. In terms of recall, Reptile is the worst for all 30×- and 70×-coverage datasets, and DecGPU is the worst for all 100×-coverage datasets. Among the nine tested datasets, Musket achieves the best performance in four cases, HiTEC is best in four cases and Coral is best in two cases (there is a tie between Musket and HiTEC for the 70×-coverage dataset with 2% error rate). In terms of precision, DecGPU consistently outperforms all other correctors, and Musket is consistently second best. Coral performs worst in five cases, and Reptile performs worst in four cases. This is because Coral (Reptile) produces significantly more false–positive results, for example, up to 87× (32×) more than Musket. In terms of F-score, Musket is superior to all other correctors for all datasets except for the 30×-coverage dataset with 3% error rate, for which HiTEC is better. HiTEC is the second best. Reptile is the worst for all 30×- and 70×-coverage datasets, and DecGPU is the worst for all 100×-coverage datasets. Ranking in terms of Gain is the same as for F-score. On average, Musket has the best recall, F-score and Gain, whereas DecGPU has the best precision. In addition, Supplementary Table S6 shows the correction quality using the two real datasets from *E.coli.*

3.1.2 Evaluation using human chromosomes Repeat regions generally occur more frequently in large genomic sequences than in small ones. While performing correction on a certain base position, these genomic repeats may result in multiple choices making error correction more challenging. In this evaluation, simulated reads are produced from the *Chr21* and *Chr1* sequences using three uniform base error rates (1, 2 and 3%) and two coverage values (30 and 70%). The base quality scores are extracted from a real dataset sequenced from a human individual (SRR189815 in NCBI SRA). In this test, we only evaluate the performance of Musket, SGA and Quake, excluding the other four correctors. Both HiTEC and SHREC are not evaluated because of excessive memory requirements (>96 GB random access memory), Coral because of too many false–positive results and excessive runtime and both Reptile and DecGPU because of their poor performance in terms of recall, *F*-score and Gain for *E.coli*.

Supplementary Table S5 shows the performance of all evaluated correctors using simulated reads from the medium-sized *Chr21* sequence. In terms of recall, Quake is consistently the worst. For the datasets with 2 and 3% error rates, Musket yields the highest performance, and SGA is second. For the other two datasets, SGA performs best, whereas Musket is second. In terms of precision, Quake is the best for all datasets, whereas Musket and SGA show comparable performance. In terms of *F*-score, Quake is worst for all datasets. Musket performs best for the datasets with 2 and 3% error rates. SGA is best for remaining two datasets. Gain has the same ranking with *F*-score for each case. For a specific coverage value, the performance of both SGA and Quake degrades quickly as the error rate grows in terms of recall, *F*-score and Gain, whereas the performance of Musket only shows small fluctuations. For a specific error rate, both Musket and Quake show improved performance as the coverage increases in terms of recall, *F*-score and Gain. However, SGA does not always follow this trend. All correctors keep relatively consistent precision for different coverage and error rates. On average, Musket is superior to SGA and Quake for recall, *F*-score and Gain measures, whereas Quake has the highest precision.

Table 1 shows the performance evaluation using reads simulated form the large-sized *Chr1* sequence. In terms of recall, Quake is still worst for all datasets. Musket achieves the best performance for the 30×-coverage datasets with 1 and 3% error rates and the 70×-coverage datasets with 2 and 3% error rates. SGA performs best for the two remaining datasets. In terms of precision, among the six tested datasets, Musket yields the highest performance in one case, SGA is best in two cases and Quake is best in three cases. Ranking in terms of either *F*-score or Gain is the same as for recall. For a specific coverage value, the performance of Quake degrades as the error rate increases for all datasets and in terms of all measures. The performance of both Musket and SGA varies as the error rate changes, but does not show any regular trend. For a specific error rate, none of the correctors does consistently improve its performance as the coverage increases. On average, Musket is superior to both SGA and Quake in terms of all four measures.

3.2 *De novo* genome assembly

De novo genome assembly has been a difficult and challenging proposition in genomics. Considerable progress has been made in just the past few years with the successful establishment of several *de novo* assemblers. The most popular approach for *de novo* assembly is the use of de Bruijn graphs. Corresponding assemblers include Velvet (Zerbino and Birney, 2008), ALLPATHS (Butler *et al.*, 2008), ABySS (Simpson *et al.*, 2009), ALLPATHS-LG (Gnerre *et al.*, 2010), SOAPdenovo (Li *et al.*, 2010b) and PASHA (Liu *et al.*, 2011b). Furthermore, some recent assemblers, such as SGA (Simpson and Durbin, 2012) and Fermi (Li, 2012), have used string graphs. To improve assembly quality, pre-assembly data cleaning has become one of the most important steps in *de novo* assembly from NGS reads. Short-read error correction has been frequently used to improve data quality and is often the most time-consuming part of the assembly pipeline. In addition to many individual correctors, some assemblers like ALLPATHS-LG, SGA and Fermi have their built-in correctors. Hence, it is valuable to evaluate the impact of different correctors on *de novo* assembly quality.

We have used three real Illumina datasets (Table 2) to assess the impact on *de novo* assembly of three correctors: Musket, SGA and Quake. The *E.coli* dataset has a coverage of ~30, produced by randomly sampling reads from the real ERR022075 dataset aforementioned. The *Caenorhabditis elegans* dataset (SRR065390 in NCBI SRA) has a coverage of ~67, and the *Chr14* dataset has a coverage of ~34, given in Salzberg *et al.* (2012). The SGA assembler is used to perform genome assembly for all tests. For the SGA assembler, the same set of parameters has been used for each case, where we set the minimal read overlap length to 50 and used default settings for all other parameters. As the SGA assembler requires every base being known,

Table 1. Performance evaluation using simulated reads from *Chr1*

Mean coverage	30×			70×		
Error rate	1%	2%	3%	1%	2%	3%
Recall						
Musket	**86.96**	87.92	**87.85**	87.17	**88.28**	**88.49**
SGA	86.93	**88.63**	78.72	**89.56**	87.79	84.81
Quake	54.63	31.11	19.14	55.91	32.98	20.78
Precision						
Musket	97.73	**97.11**	96.54	97.71	97.14	96.65
SGA	97.74	96.67	**96.85**	96.76	97.18	**97.47**
Quake	**98.19**	96.98	94.90	**98.22**	**97.21**	95.10
F-score						
Musket	**92.03**	92.29	**91.99**	92.14	**92.49**	**92.39**
SGA	92.02	**92.47**	86.85	**93.02**	92.24	90.70
Quake	70.20	47.10	31.85	71.26	49.25	34.11
Gain						
Musket	**84.94**	85.30	**84.69**	85.13	**85.67**	**85.42**
SGA	84.93	**85.58**	76.16	**86.56**	85.23	82.61
Quake	53.62	30.14	18.11	54.90	32.03	19.71

Table 2. Information of the three utilized real Illumina datasets

Name	Read length	No. of reads	Coverage	Genome length
E.coli	100	1 391 904	30×	4 639 675
C.elegans	100	67 617 092	67×	100 286 070
Chr14	100	36 504 800	34×	107 349 540

Table 3. Assembly results for all three real datasets

Dataset	Corrector	N50	N50 corrector	Errors	Coverage (%)
E.coli	Original	13 580	13 580	1	97.94
	Musket	**21 563**	**21 563**	**1**	**98.36**
	SGA	20 018	20 018	1	98.19
	Quake	18 493	18 345	1	98.20
C.elegans	Original	7987	7570	95	**96.75**
	Musket	**9360**	**8078**	**91**	96.29
	SGA	8968	7643	93	96.14
	Quake	8720	7882	92	96.74
Chr14	Original	2273	2232	822	76.93
	Musket	**2562**	**2515**	**793**	**77.52**
	SGA	2282	2252	826	77.39
	Quake	2440	2393	822	77.42

we have randomly converted all unknown bases in each dataset into known ones.

The assembly quality is measured in terms of the following metrics: N50 *contig* size, error-corrected N50 *contig* size, number of *contig* errors and genome coverage. The calculation of N50 *contig* size starts from sorting all assembled *contig*s in the descendent order of length and then accumulates the lengths from the largest to the smallest until the summed lengths are not <50% of the reference genome size. The N50 *contig* size is the size of the smallest *contig* that stops the length summation. The number of *contig* errors is calculated by summing up the number of misjoins and the number of indel errors of length >5 in all *contig*s. The genome coverage is calculated from the error-corrected *contig*s, reflecting the number of bases covered by correct *contig*s in the reference sequence. The GAGE scripts from Salzberg *et al.* (2012) are used to analyse the resulting assemblies before and after error correction for each dataset. In this evaluation, we do not use the paired-end information and only consider *contig*s whose lengths are ≥200.

Table 3 shows the assembly results. Each corrector is able to improve both the N50 *contig* size and the error-corrected N50 *contig* size for each dataset. In terms of N50 *contig* size and error-corrected N50 *contig* size, Musket is superior to all other correctors for each dataset. SGA yields greater N50 *contig* sizes than Quake for both the *E.coli* and *C.elegans* datasets, whereas the latter performs better for the *Chr14* dataset. However, in terms of error-corrected N50 *contig* size, except for the *E.coli* dataset, Quake outperforms SGA for the other two datasets. The number of *contig* errors is identical for all correctors for the *E.coli* dataset. However, Musket is superior for the other two datasets. Furthermore, all correctors are able to reduce the number of errors for both the *C.elegans* and *Chr14* datasets compared with when not using error correction. All correctors show improved genome coverage for both the *E.coli* and *Chr14* datasets. Interestingly, the genome coverage decreases after correction for the *C.elegans* dataset. This suggests that pre-assembly error correction is not always favourable with respect to some measures. In terms of genome coverage, Musket performs best for both the *E.coli* and *Chr14* datasets and Quake for the *C.elegans* dataset. SGA consistently yields the lowest coverage.

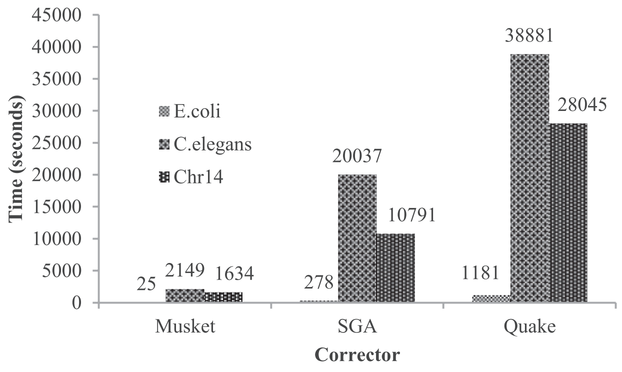

Fig. 3. Runtimes for real reads

3.3 Speed, scalability and memory consumption

In addition to correction quality, execution speed is an important factor that must be taken into account, especially for large-scale datasets. We have assessed the speed of different correctors using both simulated and real datasets. For simulated datasets, we have organized the datasets into groups as per the genomic sequences from which they are generated, and we have averaged the runtimes of all datasets in each group. For the simulated datasets from the *E.coli* genome, all evaluated correctors, with the exception of both HiTEC and Reptile (which do not support multi-threading), run two threads because at least two threads are required for Musket. For the simulated datasets from the *Chr21* and *Chr1* sequences, and the real datasets, all evaluated correctors run 12 threads. Runtimes are measured in wall clock time.

Supplementary Figure S1 shows the average runtimes of all evaluated correctors for the simulated datasets. For *E.coli*, DecGPU is the fastest, and Musket outruns the other correctors. SHREC, SGA and Coral are slowest. On average, Musket is about 2.1× faster than HiTEC, 2.5× faster than SGA, 3.3× faster than SHREC, 3.5× faster than Coral, 1.1× faster than Quake and 1.7× faster than Reptile. For *Chr21* and *Chr1*, Musket is superior to both SGA and Quake. SGA is slightly faster than Quake for all simulated *Chr21* datasets, whereas Quake runs slightly faster than SGA for all simulated *Chr1* datasets. On average, Musket runs ~4.6× (4.8×) faster than SGA and ~5.0× (4.5×) faster than Quake for all simulated datasets from sequence *Chr21* (*Chr1*).

Figure 3 shows the runtimes of all evaluated correctors for the three real datasets in Table 2 when using 12 threads. Musket is superior to both SGA and Quake for all datasets. For the *E.coli* dataset, Musket runs ~11.0× faster than SGA and ~46.7× faster than Quake. For the *C.elegans* dataset, Musket achieves a speedup of ~9.3 over SGA and ~18.1 over Quake. Finally, for the *Chr14* dataset, Musket outruns SGA by a factor of ~6.6 and Quake by a factor of ~17.2. In addition, SGA performs significantly better than Quake for each dataset.

In addition to speed, we have evaluated the parallel scalability of Musket, SGA and Quake in terms of the number of CPU threads using the real dataset *Chr14*. Figure 4 illustrates the speedups using different number of threads. For Musket, the speedups are calculated against the runtime obtained using two threads because of the use of the master–slave model

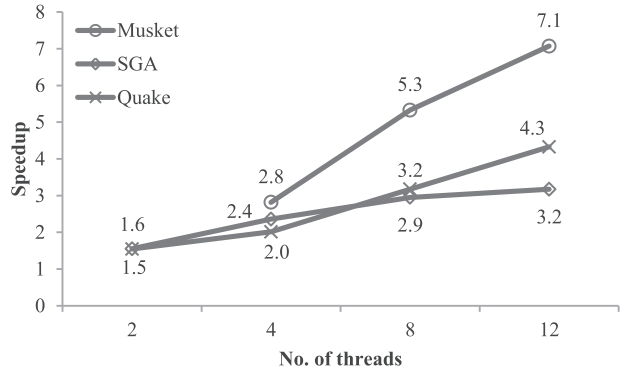

Fig. 4. Speedups using different number of threads

(see Section 2). Musket demonstrates the best scalability, and SGA is second. These results suggest superior utilization of compute power by Musket. The peak resident memory consumption for each evaluated corrector is shown in Supplementary Figures S2 and S3.

4 DISCUSSION

We have presented Musket, an efficient substitution-error-based error corrector for short DNA reads produced by Illumina sequencing technology. This reference-free error corrector uses the *k*-mer spectrum approach and aims at correcting as many errors as possible while introducing few new errors. Three correction techniques, including two-sided conservative correction, one-sided aggressive correction and voting-based refinement, have been introduced to form a multistage correction workflow.

We have assessed the performance of Musket in comparison with several established error correctors: HiTEC, SGA, SHREC, Coral, Quake, Reptile and DecGPU. The assessment is conducted using both simulated and real reads in terms of correction quality and *de novo* genome assembly measures. In terms of correction quality, Musket is consistently one of the top performing correctors for reads simulated from the small-sized *E.coli* genome as well as large-sized human chromosomes. The best performance of Musket is obtained using simulated reads from the human chromosomes, which suggests that Musket can perform well on genomic sequences with complex repeat structures. In terms of *de novo* genome assembly, Musket is superior to all other evaluated correctors in terms of the following metrics: N50 *contig* size, error-corrected N50 *contig* size, number of *contig* errors and genome coverage. Through this study, we have found that pre-assembly error correction does not always guarantee to yield better assembly quality in terms of all metrics, but have demonstrated the capability of improving *contig* contiguities and reducing mis-assemblies. Besides *de novo* genome assembly, short-read error correctors can also be used before short-read alignment. As many state-of-the-art short-read aligners tolerate multiple sequencing errors in the full-read length, substitution-error-based error correctors might not be able to make significant impact on the improvement of alignment performance. Hence, the use of Musket might be more valuable before *de novo* genome assembly than short-read alignment. In addition, Musket uses multi-threading, based on a master–slave model, to leverage the compute power of common shared-memory multi-CPU platforms. This approach results in superior parallel scalability compared with all other evaluated correctors in terms of the number of CPU threads as well as in overall execution time.

All existing standalone correctors target haploid genome sequencing (Yang *et al.*, 2012). Kelley *et al.* (2010) showed that error correction can benefit single nucleotide polymorphism calling in haploid genomes. However, the effect of error correction on diploid genomes has not been explored yet. A heterozygous site results in a percentage of *k*-mers supporting alternative alleles. In a moderate or high coverage dataset, we would expect these *k*-mers to be within the distribution of trusted *k*-mers. However, this might not be the case when the coverage is low or allelic distribution is imbalanced. In such cases, diploid single nucleotide polymorphisms could be lost as a result of false-positive corrections.

Up to date, almost all correctors based on *k*-mer spectrum use a single coverage *cut-off* to differentiate trusted *k*-mers from the untrusted ones. This static coverage *cut-off* ignores the confidence difference in the correctness of bases that is represented as base quality scores in NGS reads. Hence, a dynamic coverage *cut-off* as in Coral (Salmela and Schröder, 2011) might be advantageous to further improve correction quality of correctors based on *k*-mer spectrums. A possible solution might be the use of a position specific *cut-off* matrix derived from base quality scores for a single read.

Conflicts of Interest: none declared.

REFERENCES

Bloom,B.H. (1970) Space/time trade-offs in hash coding with allowable errors. *Commu. ACM*, **13**, 422–426.
Burrows,M. and Wheeler,D.J. (1994) A block-sorting lossless data compression algorithm. *Technical Report 124 Palo Alto, CA.*, Digital Equipment Corporation.
Butler,J. *et al.* (2008) ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome Res.*, **18**, 810–820.
Chaisson,M. *et al.* (2004) Fragment assembly with short reads. *Bioinformatics*, **20**, 2067–2074.
Chaisson,M. *et al.* (2009) De novo fragment assembly with short mate-paired reads: does the read length matter? *Genome Res.*, **19**, 336–346.
Dohm,J.C. *et al.* (2008) Substantial biases in ultra-short read data sets from high-throughput DNA sequencing. *Nucleic Acids Res.*, **36**, e105.
Ewing,B. and Green,P. (1998) Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Res.*, **8**, 186–194.
Ferragina,P. and Manzini,G. (2005) Indexing compressed text. *J. ACM*, **52**, 4.
Gnerre,S. *et al.* (2010) High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proc. Natl Acad. Sci. USA*, **108**, 1513–1518.
Ilie,L. *et al.* (2011) HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, **27**, 295–302.
Kao,W.C. *et al.* (2011) ECHO: a reference-free short-read error correction algorithm. *Genome Res.*, **21**, 1181–1192.
Kelley,D.R. *et al.* (2010) Quake: quality-aware detection and correction of sequencing errors. *Genome Biol.*, **11**, R116.
Li,H. (2012) Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, **28**, 1838–1844.
Li,R. *et al.* (2010a) The sequence and de novo assembly of the giant panda genome. *Nature*, **463**, 311–317.
Li,R. *et al.* (2010b) De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, **20**, 265–272.
Liu,Y. *et al.* (2011a) DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, **12**, 85.

Liu,Y. *et al.* (2011b) Parallelized short read assembly of large genomes using de Bruijn graphs. *BMC Bioinformatics*, **12**, 354.

Liu,Y. *et al.* (2012) CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, **28**, 1830–1837.

Medvedev,P. *et al.* (2011) Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, **27**, i137–i141.

Melsted,P. and Pritchard,J.K. (2011) Efficient counting of *k*-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, **12**, 333.

Pevzner,P.A. *et al.* (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA*, **98**, 9748–9753.

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

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

Salzberg,S.L. *et al.* (2012) GAGE: a critical evaluation of genome assemblies and assembly algorithms. *Genome Res.*, **22**, 557–567.

Schröder,J. *et al.* (2009) SHREC: a short-read error correction method. *Bioinformatics*, **25**, 2157–2163.

Shi,H. *et al.* (2010) A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware. *J. Comput. Biol.*, **17**, 603–615.

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

Simpson,J.T. *et al.* (2009) ABySS: a parallel assembler for short read sequence data. *Genome Res.*, **19**, 1117–1123.

Yang,X. *et al.* (2010) Reptile: representative tiling for short read error correction. *Bioinformatics*, **26**, 2526–2533.

Yang,X. *et al.* (2012) A survey of error-correction methods for next-generation sequencing. *Brief. Bioinform.*, [Epub ahead of print, doi:10.1093/bib/bbs015, April 6, 2012].

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