Bioinformatics, 36(12), 2020, 3669–3679
doi: 10.1093/bioinformatics/btaa179
Advance Access Publication Date: 13 March 2020
Original Paper

Sequence analysis

Apollo: a sequencing-technology-independent, scalable and accurate assembly polishing algorithm

Can Firtina [1], Jeremie S. Kim[1,2], Mohammed Alser[1], Damla Senol Cali[2], A. Ercument Cicek [3], Can Alkan[3,*] and Onur Mutlu[1,2,3,*]

[1]Department of Computer Science, ETH Zurich, Zurich 8092, Switzerland, [2]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA and [3]Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey

*To whom correspondence should be addressed.

Associate Editor: Lenore Cowen

Received on February 13, 2019; revised on December 16, 2019; editorial decision on February 14, 2020; accepted on March 11, 2020

Abstract

Motivation: Third-generation sequencing technologies can sequence long reads that contain as many as 2 million base pairs. These long reads are used to construct an assembly (i.e. the subject's genome), which is further used in downstream genome analysis. Unfortunately, third-generation sequencing technologies have *high* sequencing error rates and a *large* proportion of base pairs in these long reads is *incorrectly* identified. These errors propagate to the assembly and affect the accuracy of genome analysis. *Assembly polishing algorithms* minimize such error propagation by polishing or fixing errors in the assembly by using information from alignments between reads and the assembly (i.e. read-to-assembly alignment information). However, current assembly polishing algorithms can only polish an assembly using reads from either a certain sequencing technology or a small assembly. Such technology-dependency and assembly-size dependency require researchers to (i) run multiple polishing algorithms and (ii) use small chunks of a large genome to use all available readsets and polish large genomes, respectively.

Results: We introduce Apollo, a *universal* assembly polishing algorithm that scales well to polish an assembly of *any* size (i.e. both large and small genomes) using reads from *all* sequencing technologies (i.e. second- and third-generation). Our goal is to provide a single algorithm that uses read sets from all available sequencing technologies to improve the accuracy of assembly polishing and that can polish large genomes. Apollo (i) models an assembly as a profile hidden Markov model (pHMM), (ii) uses read-to-assembly alignment to train the pHMM with the Forward–Backward algorithm and (iii) decodes the trained model with the Viterbi algorithm to produce a polished assembly. Our experiments with real readsets demonstrate that Apollo is the *only* algorithm that (i) uses reads from any sequencing technology within a single run and (ii) scales well to polish large assemblies without splitting the assembly into multiple parts.

Availability and implementation: Source code is available at https://github.com/CMU-SAFARI/Apollo.

Contact: calkan@cs.bilkent.edu.tr or onur.mutlu@inf.ethz.ch

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

1 Introduction

High-throughput sequencing (HTS) technologies are being widely used in genomics due to their ability to produce a large amount of sequencing data at a relatively low cost compared to first-generation sequencing methods (Sanger *et al.*, 1977). Despite these advantages, HTS technologies have two significant limitations. The first limitation is that HTS technologies can only sequence fragments of the genome (i.e. *reads*). This results in the need to reconstruct the original full sequence by either using (i) read alignment, the process of aligning the reads to a *reference genome*, a genome representative of all individuals within a species or (ii) *de novo genome assembly*, the process of aligning all reads against each other to construct larger fragments called *contigs*, by identifying reads that overlap and combining them. The second limitation of HTS technologies is that they introduce non-negligible insertion, deletion and substitution errors (i.e. ~10–15% error rate) into reads. Depending on the method for reconstructing the original sequence, HTS errors often cause either (i) reads aligned to an incorrect location in the reference genome or (ii) erroneously constructed assemblies. These two limitations of HTS technologies are partially mitigated with computationally expensive algorithms such as *alignment* and *assembly construction*. Despite the wide availability of these algorithms, imperfect sequencing technologies still affect the reliability of downstream analysis in the genome analysis pipeline (e.g. variant calling).

Based on the average read length and the error profile of their reads, HTS technologies are roughly categorized into two types: (i) second- and (ii) third-generation sequencing technologies. Second-generation sequencing technologies (e.g. Illumina) generate the most accurate reads (∼99.9% accuracy). However, the length of their reads is short (∼100–300 bp) (Glenn, 2011). This introduces challenges in both read alignment and *de novo* genome assembly. In read alignment, a short read can align to multiple candidate locations in a reference equally well (Alser *et al.*, 2017, 2019a, b; Kim *et al.*, 2018; Xin *et al.*, 2013). Aligners must either deterministically select a matching location, which requires additional computation, or randomly select one of the candidate locations, which results in non-reproducible read alignments (Firtina and Alkan, 2016). In *de novo* genome assembly, high computational complexity is required to identify overlaps between reads. Even after completing *de novo* genome assembly, there are often multiple gaps in an assembly (Meltz Steinberg *et al.*, 2017). This means an assembly is composed of many smaller contigs rather than a few long contigs, or in the ideal case, a single genome-sized contig.

Third-generation sequencing technologies [i.e. PacBio's Single Molecule Real-Time and Oxford Nanopore Technologies (ONT)] are capable of producing long reads (∼10 kb on average and up to 2 Mb) at the cost of high error rates (∼10% to 15% error rate) (Huddleston *et al.*, 2014; Jain *et al.*, 2018; Payne *et al.*, 2018). Different third-generation sequencing technologies result in different error profiles. For example, PacBio reads tend to have more insertion errors than other error types whereas insertion errors are the least common errors for ONT reads (Weirather *et al.*, 2017). Long reads make it more likely to find longer overlaps between the reads in *de novo* genome assembly. As a result, there are usually fewer long contigs (Alkan *et al.*, 2011; Chaisson *et al.*, 2015; Meltz Steinberg *et al.*, 2017). Despite this, error-prone reads often result in a highly erroneous assembly, which may not be representative of the subject's actual genome. As a consequence, any analysis using the erroneous assembly (e.g. identifying variations/mutations in a subject's genome to determine proclivity for diseases) is often unreliable.

Existing solutions that try to overcome the problem of error-prone assemblies when using *de novo* genome assembly can be categorized into two types. First, a typical solution is to correct the errors of long reads. Errors are corrected by using high coverage reads (e.g. ∼100× coverage) from the same sequencing technology (i.e. self-correction) or additional reads from more reliable second-generation sequencing technologies (i.e. hybrid correction). There are several available *error correction* algorithms that use additional reads to locate and correct errors in long reads [e.g. Hercules (Firtina *et al.*, 2018), LoRDEC (Salmela and Rivals, 2014), LSC (Au *et al.*, 2012) and LoRMA (Salmela *et al.*, 2016)]. The main disadvantage of error correction algorithms is that they require *more* sequenced reads from either the same or different sequencing technologies. For example, LoRMA, a self-correction tool, uses reads to build a de Bruijn graph for error correction. The reads corrected using a de Bruijn graph method cannot span even half of the entire genome, if the coverage is lower than 100× (Salmela *et al.*, 2016). When the coverage is low, the connections in a de Bruijn graph can be weak. These weak regions can be treated as bulges and tips, and can be removed from the graph (Chaisson *et al.*, 2004), which may fail to create a reliable consensus of the entire genome for error correction. Although hybrid correction tools [e.g. PBcR (Koren *et al.*, 2012)] can use low coverage short reads (e.g. 25×) to correct the long reads that can span 95% of the genome after correction, these hybrid correction tools require *additional* short reads. Therefore, in both cases (i.e. hybrid and self-correction), generating additional reads (i.e. additional either short reads or high coverage long reads) requires additional cost and time. While a higher-coverage dataset may lead to higher read accuracy (Berlin *et al.*, 2015), the cost of producing a high-coverage dataset for long reads is often prohibitively high (Rhoads and Au, 2015). For example, sequencing the human genome with ONT at only 30× coverage costs around $36 000 (Jain *et al.*, 2018). Unless there exist sufficient resources for multiple sequencing technologies or high coverage, error correction algorithms may not be a viable option to generate accurate assemblies.

The second method for removing errors in an assembly is called *assembly polishing*. An assembly polishing process attempts to correct the errors of the assembly using the alignments of *either* long or short reads to the assembly. The *read-to-assembly* alignment, which is the alignment of the reads to the assembly, allows an assembly polishing algorithm to decide whether the assembly should be *polished* based on the similarity of the base pairs between the alignments of the reads and their corresponding locations in the assembly. If the assembly polishing algorithm finds a dissimilarity, the algorithm modifies the assembly to make it more similar to the aligned reads as it assumes that the alignment information is a more reliable source. In other words, the dissimilarity is attributed to errors in the assembly. Assembly polishing algorithms assume that such modifications correct, or polish, the errors of an assembly.

There are various assembly polishing algorithms that use various methods for discovering dissimilarities and modifying the assembly [e.g. Nanopolish (Loman *et al.*, 2015), Racon (Vaser *et al.*, 2017), Quiver (Chin *et al.*, 2013) and Pilon (Walker *et al.*, 2014)]. However, the primary limitation of many of these assembly polishing algorithms is that they work only with reads from a limited set of sequencing technologies. For example, Nanopolish can use *only* ONT long reads (Senol Cali *et al.*, 2019), while Quiver supports *only* PacBio long reads. Thus, these assembly polishing algorithms are sequencing-technology-dependent. Even though Pilon can use long reads as it does not impose a hard restriction not to use them, Pilon does not suggest using long reads and it is well tuned for using short reads. Therefore, we consider Pilon as only a partially sequencing-technology-independent algorithm as it neither prevents nor truly supports using long reads. Even though Racon can use either short or long reads to polish an assembly, it can use only a single set of reads *within a single run* (e.g. only a set of PacBio reads). This requires an assembly to be polished in multiple runs with Racon to use all the available set of reads from multiple sequencing technologies (i.e. a *hybrid set of reads*). There is currently no single assembly polishing algorithm that can polish an assembly with an *arbitrary* set of reads from various sequencing technologies (e.g. both ONT and PacBio reads) within a single run.

While the technology-dependency problem of such assembly polishing algorithms could be mitigated by consecutively using either different algorithms (e.g. Quiver and Pilon) or the same algorithm multiple times (e.g. running Racon twice to use both PacBio and Illumina reads), there are scalability problems associated with using polishing algorithms to polish a large genome and, therefore, running assembly polishing algorithms multiple times for two reasons. First, none of the polishing algorithms can scale well to polish large genomes within a single run as they require large computational resources (e.g. polishing a human genome requires more than 192 GB of available memory) unless the coverage of a set of reads is low (e.g. <10×). Therefore, these assembly polishing algorithms *cannot* polish large genomes in a single run if the available computational resources are not tremendous, and they are restricted to polish smaller parts (e.g. contigs) of a large genome. Second, dividing a large genome into smaller contigs and running polishing algorithms multiple times requires extra effort to collect and merge the multiple results to produce the polished large genome assembly as a whole.

A *universal technology-independent assembly polishing algorithm* that can use reads regardless of both (i) the sequencing technology used to produce them and (ii) the size of the genome, enables the usage of all available reads for a more accurate assembly compared to using reads from a single sequencing technology. Such a universal assembly polishing algorithm would also not require running assembly polishing multiple times to take advantage of all available reads. Unfortunately, such an assembly polishing algorithm does not exist.

Our goal in this paper is to propose a *technology-independent* assembly polishing algorithm that enables all available reads to contribute to assembly polishing and that scales well to polish an assembly of any size (e.g. both small and large genome assemblies) within a single run. To this end, we propose a machine learning-based *universal technology-independent assembly polishing* algorithm, Apollo, that corrects errors in an assembly by using read-to-assembly alignment regardless of the sequencing technology used to generate reads. Apollo is the first *universal technology-independent* assembly polishing algorithm. Apollo's machine learning algorithm is based on two key steps: (i) training and (ii) decoding the profile hidden Markov model (pHMM) of an assembly. First, Apollo uses the Forward–Backward and Baum–Welch algorithms (Baum, 1972) to train the pHMM by calculating the probability of the errors based on aligned reads. Error probabilities in the pHMM reveal how reads and the assembly that the reads align to are similar to each other without making any assumptions on the sequencing technology used to produce the reads. This is the *key* feature that makes Apollo sequencing-technology-independent. Second, Apollo uses the Viterbi algorithm (Viterbi, 1967) to decode the trained pHMM to correct the errors of an assembly. Apollo employs a recent pHMM design (Firtina *et al.*, 2018), as this design addresses the computational problems that make pHMMs otherwise impractical to use for training in machine learning. The design of the pHMM enables flexibility in adapting the pHMM based on the error profile of the underlying sequencing technology of an assembly. Therefore, Apollo can additionally apply the known error profile of a sequencing technology to improve upon its error probability calculations.

We compare Apollo with Nanopolish, Racon, Quiver and Pilon using datasets that are sequenced with different technologies: *Escherichia coli* K-12 MG1655 (MinION and Illumina), *E.coli* O157 (PacBio and Illumina), *E.coli* O157:H7 (PacBio and Illumina), Yeast S288C (PacBio and Illumina) and the human Ashkenazim trio sample (HG002, PacBio and Illumina). We compare our polished assemblies against highly accurate and finished genome assemblies of the corresponding samples to determine the accuracy of the various assembly polishing algorithms.

Using the datasets from different sequencing technologies, we first show that Apollo scales better than other polishing algorithms in polishing assemblies of large genomes using moderate and high coverage reads. Second, Apollo is the *only* algorithm that can use reads from *multiple* sequencing technologies in a *hybrid* manner (e.g. using both ONT and Illumina reads in a single run). Because of this, Apollo scales well to polish an assembly of any size within a *single* run using *any* set of reads, which makes Apollo a universal, sequencing-technology-independent assembly polishing algorithm. Third, we show that when Apollo uses a hybrid set of reads (i.e. both PacBio and Illumina reads), it polishes assemblies generated by Canu (Koren *et al.*, 2017) (i.e. Canu-generated assemblies) more accurately than any other polishing algorithm. Fourth, for all other remaining cases, when we compare Apollo to other competing algorithms, our experiments show that Apollo usually produces assemblies of similar accuracy to competing algorithms: Nanopolish, Pilon, Racon and Quiver. However, when using long readsets to polish Miniasm-generated *E.coli* O157:H7, *E.coli* K-12 and Yeast S288C assemblies, Apollo produces assemblies with less accuracy than that of Racon and Quiver. These experiments are based on (i) a ground truth (i.e. reference-dependent comparison), (ii) *k*-mer similarity calculation [i.e. Jaccard similarity (Niwattanakul *et al.*, 2013)] between an Illumina set of reads and a polished assembly and (iii) the quality assessment of the assembly from mapped short reads (i.e. reference-independent comparison). These comparisons show that Apollo can polish an assembly using reads from any sequencing technology while still generating an assembly with accuracy usually comparable to the competing algorithms. Fifth, we use moderate long read coverage datasets (e.g. 30×) and show that Apollo can produce accurate assemblies even with a moderate read coverage. We conclude that Apollo is the *first* universal assembly polishing algorithm that (i) scales well to polish assemblies of both large and small genomes and (ii) can use both long and short reads as well as a hybrid set of reads from various sequencing technologies.

This paper makes the following contributions:

- We introduce Apollo, a new assembly polishing algorithm that can make use of reads sequenced by *any* sequencing technology (e.g. PacBio, ONT, Illumina reads). Apollo is the *first* assembly polishing algorithm that (i) is scalable such that it can polish assemblies of both large and small genomes and (ii) can polish an assembly with a hybrid set of reads within a single run.
- We show that using both long and short reads in a hybrid manner to polish a Canu-generated assembly enables the construction of assemblies more accurate than those constructed by running other polishing tools multiple times.
- We show that four competing polishing algorithms cannot scale well to polish assemblies of large genomes within a single run due to large computational resources that they require.
- We provide an open source implementation of Apollo
- (https://github.com/CMU-SAFARI/Apollo).

2 Materials and methods

Apollo builds, trains and decodes a profile hidden Markov model graph (pHMM-graph) to polish an assembly (i.e. to correct the errors of an assembly). Apollo performs assembly polishing using two input preparation steps that are external to Apollo (pre-processing) and three internal steps, as shown in Figure 1. The first two pre-processing steps involve the use of external tools such as an *assembler* and an *aligner* to generate inputs for Apollo. First, an assembler uses reads (e.g. long reads) to generate assembly contigs (i.e. larger sequence fragments of the assembly). Second, an aligner aligns the reads used in the first step and any additional reads (e.g. short reads) of the same sample to the contigs to generate read-to-assembly alignment. Third, Apollo uses the assembly generated in the first step to construct a pHMM-graph per contig. A pHMM-graph comprised states, transitions between states and probabilities that are associated with both states and transitions to account for all possible error types. Examples of errors that a sequencing technology can introduce into a read are insertion, deletion and substitution errors (which we handle in this work) and chimeric errors (which we do not handle). Therefore, correction of these errors can be accomplished by deleting, inserting or substituting the corresponding base pair, respectively. Apollo identifies a path in the pHMM-graph such that the states that make the contig erroneous are excluded. Fourth, Apollo uses the read-to-assembly alignment to update, or train, the initial (*prior*) probabilities of the pHMM-graph with the Forward–Backward and Baum–Welch algorithms. During training, the Forward–Backward algorithm uses each read alignment to change the prior probabilities of the graph based on the similarity between a read and the aligned region in the assembly. Fifth, Apollo implements the Viterbi algorithm to find the path in the pHMM-graph with the minimum error probability (i.e. decoding), which corresponds to the polished version of the corresponding contig.

2.1 Assembly construction

An assembler takes a set of reads as input and identifies the overlaps between the reads to merge the overlapped regions into larger fragments called contigs. An assembler usually reports contigs in FASTA format (Pearson and Lipman, 1988) where each element comprised an ID and the full sequence of the contig. The entire collection of contigs represents the whole assembly. Apollo requires the assembly to be constructed to correct the errors in each contig of the assembly. Thus, assembly generation is an external step to the assembly polishing pipeline of Apollo Fig 1 Step 1). Apollo supports the use of any assembler that can produce the assembly in FASTA format (Pearson and Lipman, 1988), such as Canu (Koren *et al.*, 2017) and Miniasm (Li, 2016).

2.2 Read-to-assembly alignment

After assembly construction, the second external step is to generate the read-to-assembly alignment using (i) the reads that the assembler used to construct the assembly and (ii) any additional reads sequenced from the same sample (Fig. 1 Step 2). It is possible to use *any* aligner that can produce the read-to-assembly alignment in SAM/BAM format (Li *et al.*, 2009) such as Minimap2 (Li, 2018) or

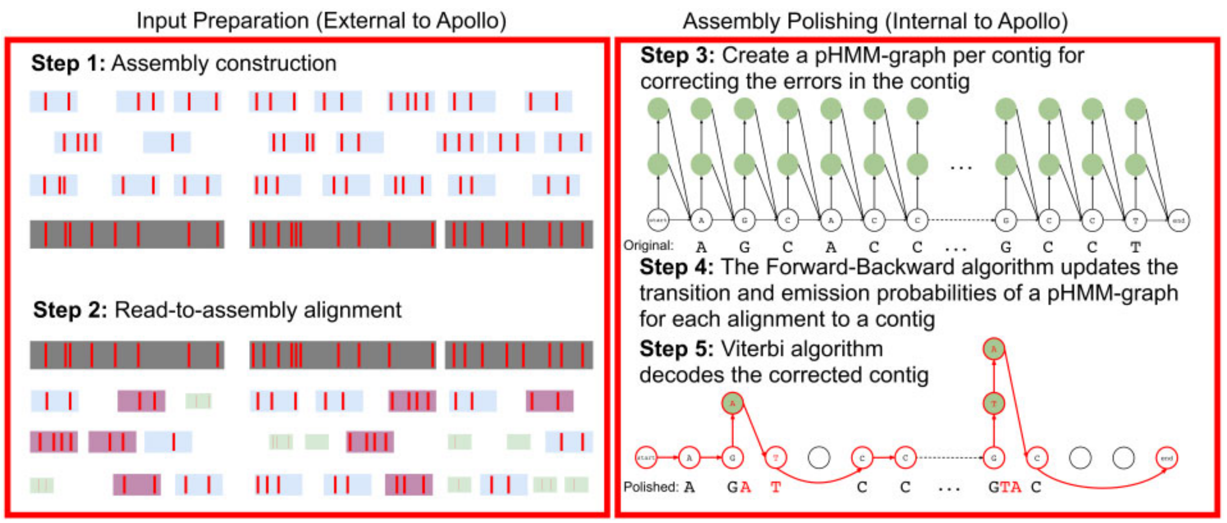
Input Preparation (External to Apollo)
Assembly Polishing (Internal to Apollo)
Step 1: Assembly construction
Step 2: Read-to-assembly alignment
Step 3: Create a pHMM-graph per contig for correcting the errors in the contig
Original: A G C A C C ... G C C T
Step 4: The Forward-Backward algorithm updates the transition and emission probabilities of a pHMM-graph for each alignment to a contig
Step 5: Viterbi algorithm decodes the corrected contig
Polished: A GA T C C ... GTA C

Fig. 1. Input preparation and the pipeline of Apollo algorithm in five steps. The first two steps refer to the use of external tools to generate the input for Apollo and are called *input preparation steps* (left side). (Step 1) An assembler generates the assembly (dark grey, large rectangles) using erroneous reads (light blue rectangles). Here, the errors are labelled with the red bars inside the rectangles. (Step 2) An aligner aligns the reads used in the first step as well as additional reads to the assembly. Here, we show the reads sequenced using different sequencing technologies in different colours and sizes (e.g. a short rectangle indicates a short read) since it is possible to use any available read within a single run with Apollo. The rest of the three steps constitute the new Apollo algorithm and are called *Internal to Apollo* (right side). (Step 3) Apollo creates a pHMM-graph per assembly contig. Here, we show an example for the pHMM-graph generated for the contig that starts with 'AGCACC' and ends with 'GCCT' as we show the original sequence below the states labelled with a base pair. Each base pair in a contig is represented by a state labelled with the corresponding base pair (i.e. match state). A pHMM-graph also consists of insertion states for each base pair labelled with green colour as well as start and end states that do not correspond to any base pair in a contig. In this example, the maximum insertion that can be made between each base pair is two as we have two insertion states per match state. Each transition or emission of a base pair from a state has a probability associated with it. For simplicity, we omit deletion transitions from this graph. (Step 4) The Forward–Backward algorithm trains the pHMM-graph and updates the transition and emission probabilities based on read-to-assembly alignments. (Step 5) Using the updated probabilities, the Viterbi algorithm decodes the most likely path in the pHMM-graph and takes the path marked with the red transitions and states, which corresponds to the polished assembly. We also show the corresponding corrections in red text colour below the states. For each contig, the output of Apollo is the sequence of base pairs associated with the states in the most likely path. (Color version of this figure is available at *Bioinformatics* online.)

BWA-MEM (Li and Durbin, 2009). In the case where reads from multiple sequencing technologies are available for a given sample, an aligner aligns all reads to the assembly. Apollo assumes that the alignment file is coordinate sorted and indexed.

Apollo uses the assembly and the read-to-assembly alignment generated in the first two pre-processing steps in its assembly polishing steps. The next three steps (Steps 3–5) are the assembly polishing steps and implemented within Apollo.

2.3 Creating a pHMM-graph per contig

The pHMM-graph that Apollo employs includes states that emit certain characters, directed transitions that connect a state to other states, and probabilities associated with character emissions and state transitions. The state transition probability represents the likelihood of following a path from a state to another state using the transitions connecting the states, and the character emission probability represents the likelihood for a state to emit a certain base pair when the state is visited. These pHMM-graph elements enable a pHMM-graph to provide the probability of generating a certain sequence when a certain path of states is followed using the directed transitions between the states.

This probabilistic behaviour of pHMM-graphs makes them a good candidate to resolve errors of an assembly. Apollo represents each contig of an assembly as a pHMM-graph. The complete structure of a pHMM-graph allows Apollo to handle three major types of errors: substitution, deletion and insertion errors. First, Apollo represents *each* base pair of a contig as a state, called the *match state*. The pHMM-graph preserves the sequence order of the contig by inserting a directed *match transition* from the previous match state of a base pair to the next one. The match state of a certain base pair has a predefined (*prior*) *match emission probability* for the corresponding base pair, and *mismatch emission probability* for the three remaining possible base pairs (i.e. a substitution error). A match state handles the cases when there is no error in the corresponding base pair (i.e. emitting the base pair that already exists in the certain position), or when there is a *substitution error* (i.e. emitting a different base pair for the certain position). Second, there are *l* many *insertion states* for each base pair in the contig, where *l* is a parameter to Apollo, which defines the maximum number of additional base pairs that can be inserted between two base pairs (i.e. two match states). An insertion state inserts a single base pair in the location it corresponds to (e.g. visiting two subsequent insertion states after a match state inserts two base pairs between the two match states) to handle a *deletion error*. Last, each match and insertion state has *k* many *deletion transitions*, where *k* is also a parameter to Apollo, which defines the maximum number of contiguous base pairs that can be deleted with a single transition. If there is an *insertion error*, a deletion transition skips the match states between a state (e.g. an insertion or a match state) and a match state to delete the corresponding base pairs of the skipped match states. Further details of the pHMM-graph can be found in Supplementary Material, Section S1.

The pHMM-graph structure that Apollo uses is identical to the one proposed in Hercules (Firtina *et al.*, 2018), a recently proposed error correction algorithm that uses pHMM-graphs. The key difference is that Apollo creates a graph *for each contig* whereas Hercules creates a graph for each *read*. As such, the pHMM-graph size in Apollo is usually larger than that in Hercules since contigs are typically longer than reads. Therefore, Apollo uses *additional techniques* to handle large pHMM-graphs (e.g. dividing pHMM-graphs into smaller graphs without compromising correction accuracy) during both training and decoding steps, which has certain trade-offs with respect to implementation, as we explain in Sections 2.4, 2.5 and 3.1.

2.4 Training with the Forward–Backward algorithm

The training step of Apollo uses each read-to-assembly alignment to update transition and emission probabilities of a contig's pHMM-graph. The purpose of the training step is to make specific transitions and emissions more probable in a *sub-graph* of the pHMM-graph such that it will be more likely to emit the entire read sequence for the region that the read aligns to. A sub-graph contains a subset of the states of a pHMM-graph and the transitions connecting these states. Each difference between a contig and the aligned read updates the probabilities so that it will be more likely to reflect the *difference* observed in the read. The calculations during training do *not* make assumptions about the sequencing technology of the read but only reflect the differences and similarities in the pHMM-graph. Thus, Apollo can update the sub-graph with *any* read aligned to the contig. This makes Apollo a sequencing-technology-independent algorithm.

For each alignment to a contig, Apollo identifies the sub-graph that the read aligns to in the pHMM-graph to update (train) the emission and transition probabilities in the sub-graph. Apollo locates the start and end states of the sub-graph to define its boundaries in the pHMM-graph. First, Apollo identifies the start location of a read's alignment in the contig and marks the match state of the *previous* base pair as the *start state*. Second, Apollo estimates the location of the *end state* such that the number of match states between the *start state* and the *end state* is longer than the length of the aligned read (i.e. up to 33.3% longer). This is to account for the case where there are more insertion errors than deletion errors. The Backward calculation uses the end state as the initial point to calculate the probabilities from backward as we explain later in this section. An accurate estimation of the end state is crucial as an inaccurate initial point for the Backward calculation may lead to inaccurate training. The insertion and the match states between the start and the end states as well as the transitions connecting these states constitute the sub-graph of the aligned region.

The sub-graphs that Apollo trains usually vary in size since the length of long reads (i.e. reads sequenced by the third-generation sequencing technologies) can fluctuate dramatically (e.g. from 15 bp to 2 Mb) whereas the length of short reads is usually fixed (e.g. 100 bp). As Apollo polishes the assembly using both short and long reads, the broad range of read lengths requires Apollo to be flexible in terms of defining the *length of the sub-graph* (i.e. the number of match states that the sub-graph includes) to train. This is a key difference in requirements between Apollo and Hercules (Firtina *et al.*, 2018). Hercules defines the number of match states to include in a sub-graph with a *fixed* ratio as the aligned reads are *always* short reads. However, Apollo is more flexible in the selection of the region that a sub-graph covers since Apollo can use reads of any length. Apollo decides whether the aligned read is *short* or *long* based on the read length, of which we set the threshold at 500 bp (i.e. if a read is longer than 500 bp, it is considered as a long read). If the aligned read length is *short* (i.e. shorter than 500 bp), the sub-graph is 33.3% longer than the length of the short read. Otherwise, the sub-graph is 5% longer than the length of the aligned long read (empirically chosen).

Apollo uses the Forward–Backward and the Baum–Welch algorithms (Baum, 1972) to train the sub-graph that a read aligns to. The Forward–Backward algorithm takes the aligned read as an observation and updates the emission and transition probabilities of the states in the sub-graph. There are three steps in the Forward–Backward algorithm: (i) Forward calculation, (ii) Backward calculation and (iii) training by updating the probabilities (i.e. the expectation–maximization step using the Baum–Welch algorithm). First, Forward calculation visits each possible path from the start state up to but not including the end state until each visited state emits a single base pair from the read starting from the first (i.e. leftmost) base pair. Therefore, the number of visited states is equal to the length of the aligned read. Second, similar to Forward calculation, Backward calculation visits each possible path in a backward fashion (i.e. from the last base pair to the first base pair) starting with the state that the Forward calculation determines to be the most likely until the start state. Third, the Forward–Backward algorithm updates the transitions and emission probabilities based on how likely it is to take a certain transition or a state to emit a certain character. We refer to the updated probabilities as *posterior probabilities*. In theory, the training step known as the Baum–Welch algorithm (Baum, 1972) is separated from the Forward–Backward calculations, as described in Supplementary Material, Section S3. However, for the sake of simplicity, we assume that the Forward–Backward step includes both the Forward–Backward calculations and the training step when we refer to it in the remaining part of this paper. Apollo trains each sub-graph (i.e. each read alignment) independently even though the states and the transitions may overlap between the aligned reads. For overlaps, Apollo takes the average of the posterior transition and emission probabilities of the overlapping regions. Once Apollo trains each pHMM sub-graph using all the alignments to a contig, it completes the training phase for that contig. The trained pHMM-graph represents the polished version of the contig. Supplementary Material, Sections S2 and S3 describe in detail how Apollo locates a sub-graph per read alignment and the training phase of the Forward–Backward algorithm.

2.5 Decoding with the Viterbi algorithm

The last step in Apollo's assembly polishing mechanism is the decoding of the trained pHMM-graph to extract the path with the highest probability from the start of the graph to the end of the graph. Finding the path with the highest probability reveals the consensus of the aligned reads to correct the contig. To identify this path, Apollo uses the Viterbi algorithm (Viterbi, 1967) on the trained pHMM-graph (Fig. 1, Step 5). The Viterbi algorithm is a dynamic programming algorithm that finds the most likely *backtrace* from a certain state to the start state in a given graph. Each Viterbi value represents how likely it is to be in a certain state at a time t (i.e. position in the contig) and is stored in the corresponding cell in a table called a dynamic programming table (DP table). Thus, a complete DP table reveals the most likely path of the entire pHMM-graph by backtracking the most likely path from the end state to the start state.

The Viterbi algorithm computes each entry of the DP table using the Viterbi values of the previously visited states. This data dependency makes the Viterbi algorithm less suitable for multi-threading support, as it prevents calculating the Viterbi values of the entire graph in parallel. Apollo overcomes this issue by dividing the pHMM-graph into sub-graphs (i.e. chunks), each of which includes a certain number of states. The Viterbi algorithm decodes each sub-graph (i.e. finds the optimal path in a graph) and merges the decoding results into one piece again. Since the Viterbi algorithm can decode each sub-graph independently, this allows Apollo to parallelize the Viterbi algorithm. We find that our parallelization greatly speeds up the Viterbi algorithm, by $\sim 20\times$.

Apollo follows a slightly different approach than the actual Viterbi algorithm when decoding a graph. The actual Viterbi algorithm uses an observation provided as input (i.e. a sequence of base pairs) to calculate the Viterbi values of states in the graph. For Apollo, there is no observation provided as input. Apollo uses the base pair with the highest emission probability of a state as observation when calculating the Viterbi value of that state. For each state in the decoded path, Apollo outputs the base pair with the highest probability, which corresponds to the polished contig. Apollo reports each polished contig as a read in FASTA format. Details of the Viterbi algorithm are in Supplementary Material, Section S4.

Note that Apollo can only polish contigs to which at least a single read aligns. Thus, Apollo reports an unpolished version of a contig, if there is no read aligned to it. In such cases, Apollo also reports the issue as output by informing that a certain contig cannot be polished because there is no read aligned to the contig. After raising the issue, Apollo continues polishing the remaining contigs, if any. We expect that such a case happens rarely. For example, a low coverage set of short reads may not be able to align to a too small and erroneous contig constructed using long reads, which would leave the contig with no read aligned to it. Another example would be having very similar regions (i.e. repetitive regions) in multiple contigs such that reads can be assigned to only one of the contigs sharing a similar region. Such a case may leave a contig without any read aligned to it since these reads may already be aligned to the similar regions in other contigs.

3 Results

3.1 Experimental setup

We implemented Apollo in C++ using the SeqAn library (Döring *et al.*, 2008). The source code is available at https://github.com/CMU-SAFARI/Apollo. Apollo supports multi-threading.

Our evaluation criteria include three different methods to assess the quality of the assemblies. First, we use the *dnadiff* tool provided under MUMmer package (Kurtz *et al.*, 2004) to calculate the accuracy of polished assemblies by comparing them with the highly accurate reference genomes (i.e. ground truth genomes). We report the percentage of bases of an assembly that align to its reference (i.e. *Aligned Bases*), the fraction of identical portions between the aligned bases of an assembly and the reference (i.e. *Accuracy*), a score value that is the product of *accuracy* and number of *Aligned Bases* (as a fraction), which we call the *Polishing Score*. *Accuracy* value provides the accuracy of only the aligned portions of the polished assembly, not the entire assembly. However, *polishing score* is a more comprehensive measure compared to *accuracy*, as it normalizes the accuracy of the aligned portions of the polished assembly to the entire length of the assembly. Second, we use sourmash (Brown and Irber, 2016) to calculate the *k*-mer similarity between filtered Illumina reads and an assembly. Third, we use QUAST (Gurevich *et al.*, 2013) to report a further quality assessment of assemblies based on the mapping of filtered Illumina reads to assemblies. Both *k*-mer similarity and QUAST provide a reference-independent evaluation of assemblies.

Based on our evaluation criteria, we compare Apollo to four state-of-the-art assembly polishing algorithms: Nanopolish (Loman *et al.*, 2015), Racon (Vaser *et al.*, 2017), Quiver (Chin *et al.*, 2013) and Pilon (Walker *et al.*, 2014). If an assembly polishing algorithm does not support a certain dataset, we do not run the algorithm on that dataset. For example, we use Nanopolish only for the ONT dataset and Quiver only for PacBio datasets, and Pilon only for the Illumina dataset. We use Pilon with a PacBio dataset only once to show its capability to polish an assembly using long reads, albeit very inefficiently. We include Apollo and Racon in every comparison as they support a set of reads from any sequencing technology. For each dataset, we compare the algorithms that polish an assembly using the same set of reads. We run each assembly polishing algorithm with its default parameters.

We run all the tools (i.e. assemblers, read mappers and assembly polishing algorithms) on a server with 24 cores (2 threads per core, Intel® Xeon® Gold 5118 CPU @ 2.30 GHz), and 192 GB of main memory. We assign 45 threads to all the tools we use and collect their runtime and memory usage using the *time* command in Linux with the $-vp$ options. We report runtime and peak memory usage of the assembly polishing algorithms based on these configurations.

We use state-of-the-art tools to construct an assembly and to generate a read-to-assembly alignment before running Apollo, which correspond to the input preparation steps. We use Canu (Koren *et al.*, 2017) and Miniasm (Li, 2016) tools to construct assemblies of each set of long reads. For read-to-assembly alignment, we use Minimap2 and BWA-MEM to align long and short reads to an assembly. Quiver cannot work with alignment results that Minimap2 and BWA-MEM produce, but requires a certain type of aligner to align PacBio reads to an assembly. Thus, we use the *pbalign* tool (https://github.com/PacificBiosciences/pbalign) that uses BLASR (Chaisson and Tesler, 2012) to align PacBio reads to an assembly to generate a read-to-assembly alignment in the format that Quiver requires. We sort and index the resulting SAM/BAM read-to-assembly alignments using the SAMtools' sort and index commands (Li *et al.*, 2009), respectively.

After assembly generation, we divide the long reads into smaller *chunks* of size 1000 bp (i.e. we perform *chunking*). We do this because long reads cause high memory demand during the assembly polishing step, especially for large genomes (e.g. a human genome). This bottleneck exists not only for Apollo but also for other assembly polishing algorithms (e.g. Racon). For Apollo, dividing long reads into chunks prevents possible memory overflows due to the memory-demanding calculation of the Forward–Backward algorithm. Even though it is still possible to use long reads without chunking, we suggest using the resulting reads *after chunking* if the available memory is not sufficient to run Apollo. We show that chunking results in producing more accurate assemblies (Supplementary Table S18).

Default parameters of Apollo are as follows: minimum mapping quality ($q = 0$), maximum number of states that Forward–Backward ($f = 100$) and the Viterbi algorithms ($v = 5$) evaluate for the next time step, the number of insertion states per base pair ($i = 3$), the number of base pairs decoded per sub-graph by Viterbi ($b = 5000$), maximum deletions per transition ($d = 10$), transition probability to a match state ($tm = 0.85$), transition probability to an insertion state ($ti = 0.1$), factor for the polynomial distribution to calculate each deletion transition ($df = 2.5$) and match emission probability ($em = 0.97$).

3.2 Datasets

In our experiments, we use DNA-seq datasets from five different samples sequenced by multiple sequencing technologies, as we show in Table 1.

We use a dataset from a large genome (i.e. a human genome) to demonstrate the scalability of polishing algorithms. For this purpose, we use the human genome sample from the Ashkenazim trio (HG002, Son) to compare the computational resources (i.e. time and maximum memory usage) that each polishing algorithm requires. We filter out the PacBio reads that have a length of <200 before calculating coverage and average read length.

Table 1. Details of our datasets

Dataset	Accession number	Details
E.coli K-12—ONT	Loman Lab[a]	164 472 reads (avg. 9010 bp, 319× coverage)
E.coli K-12—Illumina	SRA SRR1030394	2 720 956 paired-end reads (avg. 243 bp each, 285× coverage)
E.coli K-12—Ground Truth	GenBank NC_000913	Strain MG1655 (4641 kb)
E.coli O157—PacBio	SRA SRR5413248	177 458 reads (avg. 4724 bp, 151× coverage)
E.coli O157—Illumina	SRA SRR5413247	11 856 506 paired-end reads (150 bp each, 643× coverage)
E.coli O157—Ground Truth	GenBank NJEX02000001	Strain FDAARGOS_292 (5566 kb)
E.coli O157:H7—PacBio	SRA SRR1509640	76 279 reads (avg. 8270 bp, 112× coverage)
E.coli O157:H7—Illumina	SRA SRR1509643	2 978 835 paired-end reads (250 bp each, 265× coverage)
E.coli O157:H7—Ground Truth	GCA_000732965	Strain EDL933 (5639 kb)
Yeast S288C—PacBio	SRA ERR165511(8-9), ERR1655125	296 485 reads (avg. 5735 bp, 140× coverage)
Yeast S288C—Illumina	SRA ERR1938683	3 318 467 paired-end reads (150 bp each, 82× coverage)
Yeast S288C—Ground Truth	GCA_000146055.2	Strain S288C (12 157 kb)
Human HG002—PacBio	SRA SRR2036(394-471), SRR203665(4-9)	15 892 517 reads (avg. 6550 bp, 35× coverage)
Human HG002—Illumina	SRA SRR17664(42-59)	222 925 733 paired-end reads (148 bp each, 22× coverage)
Human HG002—Ground Truth	GCA_001542345.1	Ashkenazim trio—Son (2.99 GB)

Note: The datasets we use in our experiments. These data can be accessed through NCBI using the accession number.
[a]The ONT datasets are available at http://lab.loman.net/2016/07/30/nanopore-r9-data-release/

Table 2. Applicability, runtime and memory requirements of four assembly polishing tools on a *complete human genome assembly*

Aligner	Sequencing tech. of the reads	Polishing algorithm	Runtime	Memory (GB)
Minimap2	PacBio (35×)	Apollo	228 h 43 m 13 s	62.91
BWA-MEM	PacBio (35×)	Apollo	200 h 13 m 06 s	58.60
Minimap2	PacBio (35×)	Racon	N/A	N/A
BWA-MEM	PacBio (35×)	Racon	N/A	N/A
pbalign	PacBio (35×)	Quiver	N/A	N/A
Minimap2	PacBio (8.9×)	Apollo	56 h 21 m 56 s	44.99
BWA-MEM	PacBio (8.9×)	Apollo	42 h 19 m 09 s	45.00
Minimap2	PacBio (8.9×)	Racon	3 h 31 m 37 s	54.13
BWA-MEM	PacBio (8.9×)	Racon	2 h 17 m 21 s	51.55
pbalign	PacBio (8.9×)	Quiver	N/A	N/A
Minimap2	Illumina (22×)	Apollo	98 h 07 m 05 s	101.12
BWA-MEM	Illumina (22×)	Apollo	105 h 15 m 05 s	107.06
Minimap2	Illumina (22×)	Racon	N/A	N/A
BWA-MEM	Illumina (22×)	Racon	N/A	N/A
Minimap2	Illumina (22×)	Pilon	N/A	N/A
Minimap2	Illumina (22×)	Pilon	N/A	N/A
Minimap2	PacBio (35×)	Apollo[a]	230 h 37 m 58 s	25.23
pbalign	PacBio (35×)	Quiver[a]	104 h 42 m 35 s	29.92
Minimap2	PacBio (35×)	Racon[a]	6 h 48 m 17 s	132.51
Minimap2	Illumina (22×)	Apollo[a]	103 h 27 m 45 s	39.35
BWA-MEM	Illumina (22×)	Apollo[a]	111 h 35 m 15 s	39.35
Minimap2	Illumina (22×)	Pilon[a]	13 h 59m 32 s	66.67
BWA-MEM	Illumina (22×)	Pilon[a]	21h 15 m 57 s	49.93

Note: We polished the assembly of the Ashkenazim trio sample (HG002, Son) for different combinations of sequencing technology, aligner and polishing algorithm. We report the runtime and the memory requirements of the assembly polishing tools (i.e. Aligner + Polishing). We report *Runtime* and *Memory* as N/A, if a polishing algorithm fails to polish the assembly.

[a]We polish the assembly *contig by contig* in these runs and collect the results once all of the contigs are polished separately.

We use the *E.coli* O157 (Strain FDAARGOS_292), *E.coli* O157:H7, *E.coli* K-12 MG1655 and Yeast S288C datasets to evaluate the polishing accuracy of Apollo and other state-of-the-art polishing algorithms in four ways. First, we evaluate whether using a hybrid set of reads with Apollo results in more accurate assemblies compared to polishing an assembly twice using a combination of other polishing tools (e.g. Racon + Pilon). Second, we measure the performance of the polishing algorithms when they polish the assemblies only once. Third, we subsample the *E.coli* O157 and *E.coli* K-12 datasets into 30× coverage to compare the performance of algorithms when long read coverage is moderate. Fourth, we additionally use the Human HG002 dataset to measure the *k*-mer distance and quality assessment of the assemblies using sourmash and QUAST, respectively.

3.3 Applicability of polishing algorithms to large genomes

We use the polishing algorithms to polish a large genome assembly (e.g. a human genome) to observe (i) whether the polishing algorithms can polish these large assemblies without exceeding the limitations of the computational resources we use to conduct our experiments and (ii) the overall computational resources required to polish a large genome assembly (i.e. alignment and polishing). For this purpose, we use the PacBio and Illumina reads from the human genome sample of the Ashkenazim trio (HG002, Son) to polish a *finished assembly* of the same Ashkenazim trio sample. The finished assembly was released by the Genome in a Bottle (GIAB) consortium (genomeinabottle.org). GIAB used (i) Celera Assembler with PBcR (v. 8.3rc2) (Koren *et al.*, 2012) to assemble the PacBio reads from the HG002 sample and (ii) Quiver to polish the assembly (Wenger *et al.*, 2019). Based on our experiments that we report in Table 2, we make four key observations. First, Pilon, Quiver and Racon *cannot* polish the assembly using the whole sets of PacBio (∼35× coverage) and Illumina (∼22× coverage) reads due to high computational resources that they require. Racon and Pilon exceed the memory limitations while using either the PacBio or Illumina reads to polish the human genome assembly. Quiver cannot start polishing the assembly as the required aligner (i.e. BLASR from the pbalign tool) cannot produce the alignment result due to memory limitations. Apollo can polish an assembly using *both* PacBio and Illumina reads using at most nearly half of the available memory. Second, we reduce the coverage of the PacBio reads to 8.9× (SRA SRR2036394–SRR2036422) to observe whether Racon and Quiver can polish the large genome using a low coverage set of PacBio reads. We find that Racon is able to polish a human genome assembly using low coverage set of reads whereas BLASR *cannot* produce the alignment results that Quiver requires due to memory limitations even when using a low coverage set of reads. Third, we split read-to-assembly alignment into multiple alignment files such that all reads mapped to each contig are represented in a separate alignment file (i.e. *read-to-contig* alignment) to evaluate whether Pilon, Quiver and Racon can polish the entire human genome using read-to-contig alignments. We observe that Pilon, Quiver and Racon can polish contigs of a large genome, as Table 2 shows. We note that when using pbalign, we align small batches of PacBio datasets (e.g. 1× coverage each) and later merge the alignments of these small batches. We also note that both the size of the longest contig (i.e. 35.2 Mb) and the number of short read alignments to the longest contig (i.e. 5 313 903) are ∼85× smaller than that of the entire assembly. When contigs longer than 35 Mb are available, we expect Pilon and Racon to require more memory for polishing longer contigs since these tools cannot scale well with contig size. Fourth, Apollo requires less memory than any polishing algorithm when polishing the human genome assembly contig by contig. We conclude that Apollo is the *only algorithm* that scales well (i.e. memory requirements do not increase dramatically as the genome size increases) in polishing large genomes using a set of both PacBio and Illumina reads without reducing the coverage of the readset or splitting the readset or the alignment file into smaller batches. Pilon, Quiver and

Racon can polish a large genome assembly without reducing the coverage of a readset only if they polish the entire assembly contig-by-contig or split the readset into smaller batches before alignment.

3.4 Polishing accuracy

We first examine whether the use of a hybrid set of reads (e.g. long and short reads) within a single polishing run provides benefit over polishing an assembly twice using a set of reads from only a single sequencing technology (e.g. only PacBio reads) in each run. Second, we evaluate assembly polishing algorithms and compare them to each other given different options with respect to (i) the sequencing technology that produces long reads, (ii) the assembler that constructs an assembly using long reads, (iii) the aligner that generates read-to-assembly alignment and (iv) the set of reads that align to an assembly. We report the accuracy of unpolished assemblies as well as the performance of assembly polishing algorithms based on the evaluation criteria we explained in Section 3. We also compare the tools based on their performance given moderate (e.g. ~30×) and low (e.g. 2.6×) long read coverage.

Apollo is either more accurate than or as accurate as running Pilon twice using a hybrid set of reads. Apollo also polishes Canu-generated assemblies more accurately for a species with PacBio reads than running other polishing tools multiple times. In Table 3 (complete results in Supplementary Table S1) and Supplementary Table S2, we highlight the benefits of using a hybrid set of reads (e.g. PacBio + Illumina) within a single polishing run compared to polishing an assembly in multiple runs by using a set of reads from only a single sequencing technology (e.g. only PacBio or only Illumina) in each run. To this end, we compare the accuracy of polished assemblies using Apollo with that of the polished assemblies using other polishing tools (Nanopolish, Pilon, Quiver and Racon) that we run multiple times. We use long (PacBio or ONT) and short (Illumina) reads from *E.coli* O157, *E.coli* O157:H7, *E.coli* K-12 MG1655 and Yeast S288C datasets to polish Canu- and Miniasm-generated assemblies. For the *first run*, we use the polishing algorithms to polish Canu- and Miniasm-generated assemblies. For the *second run*, we provide Nanopolish, Pilon, Quiver and Racon with the polished assembly from the first run and run these tools for the second time (i.e. *Second Run*). Based on Supplementary Tables S1 and S2, we make three key observations. First, Apollo and Pilon are the only algorithms that *always* polish a Canu-generated assembly with a *polishing score* either equal to or better than that of the original Canu-generated assembly. Second, running other polishing tools multiple times to polish a Miniasm-generated assembly usually results in assemblies with higher polishing scores (e.g. by at most 3.79% for PacBio and 7.57% for ONT readsets) than using Apollo with a hybrid set of reads. Third, Apollo performs better when it uses PacBio reads in the hybrid set than using ONT reads. We conclude that the use of Apollo once with a hybrid set of reads that includes PacBio reads and a Canu-generated assembly is the best pipeline (i.e. one can construct the *most* accurate assemblies for a species versus running other polishing tools multiple times).

Apollo performs better than Pilon and comparable to Racon and Quiver when polishing a Canu-generated assembly using only a high coverage set of PacBio or Illumina reads. In Supplementary Tables S3, S6 and S12, we use PacBio and Illumina datasets to compare the performance of Apollo with Racon (Vaser *et al.*, 2017), Quiver (Chin *et al.*, 2013) and Pilon (Walker *et al.*, 2014). Based on these datasets, we make five observations. First, Apollo usually outperforms Pilon (i.e. 4 out of 7, see the *Polishing Score* column) using a set of short reads. Second, Apollo, Racon and Quiver show significant improvements over the original Miniasm assembly in terms of accuracy. Third, Quiver and Racon polish the Miniasm-generated assembly more accurately than Apollo (see the *Accuracy* and the *Polishing Score* columns). Fourth, Apollo produces more accurate assemblies than the assemblies polished by Racon when we use moderate (~30×) and high coverage (151×) PacBio readsets to polish Canu-generated assemblies. However, both algorithms generate assemblies with lower accuracy than the accuracy of the original Canu-generated assembly (0.9998 with the polishing score of 0.9992) when we use high coverage readsets. Based on this

Table 3. Comparison between using a hybrid set of reads with Apollo and running other polishing tools twice to polish a Canu-generated assembly

Dataset	First run	Second run	Aligned bases (%)	Accuracy	Polishing score	Runtime	Memory (GB)
E.coli O157	—	—	99.94	0.9998	0.9992	43 m 53 s	3.79
E.coli O157	Apollo (Hybrid)	—	99.94	0.9999	**0.9993**	8 h 16 m 08 s	13.85
E.coli O157	Racon (PacBio)	Racon (Illumina)	99.94	0.9994	0.9988	21 m 44 s	22.65
E.coli O157	Pilon (Illumina)	Racon (PacBio)	99.94	0.9986	0.9980	**4 m 58 s**	11.40
E.coli O157	Quiver (PacBio)	Pilon (Illumina)	99.94	0.9998	0.9992	5 m 01 s	**7.50**
E.coli O157:H7	—	—	100.00	0.9998	0.9998	43 m 19 s	3.39
E.coli O157:H7	Apollo (Hybrid)	—	100.00	0.9999	**0.9999**	5 h 58 m 05 s	8.86
E.coli O157:H7	Racon (PacBio)	Racon (Illumina)	100.00	0.9995	0.9995	9 m 43 s	**6.56**
E.coli O157:H7	Pilon (Illumina)	Racon (PacBio)	100.00	0.9996	0.9996	**6 m 04 s**	10.75
E.coli K-12	—	—	99.98	0.9794	0.9792	34 h 21 m 46 s	5.06
E.coli K-12	Apollo (Hybrid)	—	99.99	0.9953	0.9952	9 h 09 m 50 s	9.35
E.coli K-12	Racon (ONT)	Racon (Illumina)	100.00	0.9996	**0.9996**	**11 m 05 s**	**5.10**
E.coli K-12	Pilon (Illumina)	Racon (ONT)	99.99	0.9997	**0.9996**	15 m 51 s	8.84
E.coli K-12	Nanopolish (ONT)	Pilon (Illumina)	99.99	0.9992	0.9991	9 h 45 m 01 s	18.10
Yeast S288C	—	—	99.89	0.9998	0.9987	1 h 20 m 39 s	6.24
Yeast S288C	Apollo (Hybrid)	—	99.89	0.9998	**0.9987**	11 h 08 m 41 s	**6.38**
Yeast S288C	Racon (PacBio)	Racon (Illumina)	99.89	0.9994	0.9983	38 m 21 s	6.93
Yeast S288C	Pilon (Illumina)	Racon (PacBio)	99.89	0.9960	0.9949	21 m 42 s	11.85
Yeast S288C	Quiver (PacBio)	Pilon (Illumina)	98.95	0.9998	0.9893	**12 m 47 s**	13.28

Note: We use the long reads of *E.coli* O157, *E.coli* O157:H7, *E.coli* K-12 and Yeast S288C datasets that are sequenced from PacBio and ONT (151×, 112×, 319× and 140× coverage, respectively) to generate their assemblies with *Canu*. Here, the polishing tools specified under *First Run* and *Second Run* polish the assembly using the set of reads specified in parentheses. The set of reads used in the second run is aligned to the assembly polished in the first run using Minimap2. PacBio and Illumina set of reads together constitute the hybrid set of reads (i.e. *Hybrid*). We report the performance of the polishing tools in terms of the percentage of bases of an assembly that aligns to its reference (i.e. *Aligned Bases*), the fraction of identical portions between the aligned bases of an assembly and the reference (i.e. *Accuracy*) as calculated by dnadiff, and *Polishing Score* value that is the product of *Accuracy* and *Aligned Bases* (as a fraction). We report the runtime and the memory requirements of the assembly polishing tools. We show the best result among *assembly polishing algorithms* for each performance metric in bold text.

observation, we suspect that the use of the original set of long reads (i.e. the set of reads that we use to construct an assembly) is not helpful as Canu corrects long reads before constructing an assembly. Thus, we also tried using the Canu-corrected long reads to polish a Canu-generated assembly. However, the use of corrected long reads did not consistently result in generating more accurate assemblies than the assemblies polished using the original set of long reads as we report in Supplementary Tables S3 and S9. We find that the alignment of Canu-corrected long reads to an erroneous assembly generates a smaller number of alignments than the alignment of the original long reads to the same erroneous assembly, as we show in Supplementary Table S17. We believe that the decrease in the number of alignments results in loss of information that assembly polishing algorithms use to polish an assembly, which subsequently leads to either similar or worse assembly polishing accuracy than using original set of long reads. Fifth, even though Pilon is not optimized to use long reads, we use Pilon to polish an assembly using long reads to observe if it polishes the assembly with comparable accuracy to the other polishing algorithms. We observe that Pilon significantly falls behind the other polishing algorithms in terms of our evaluation criteria. Thus, we do not use Pilon with long reads. We conclude that (i) Apollo usually performs better than Pilon when using short reads and (ii) Apollo's performance is comparable to Racon and Quiver when using long PacBio reads to polish an assembly.

Apollo performs better than Pilon and Nanopolish when polishing a Miniasm-generated assembly using only a set of Illumina and ONT reads, respectively. We also investigate the performance of Apollo given the ONT dataset (*E.coli* K-12 MG1655), compared to Nanopolish and Racon. We make two key observations based on the results we show in Supplementary Table S9. First, Racon provides the best performance in terms of the accuracy of contigs when the coverage is high (319×) and the accuracy of the original assembly is low (e.g. a Miniasm-generated assembly). In the same setup, Apollo produces a more accurate assembly than Nanopolish. Second, even though Nanopolish produces the most accurate results with Canu using either high coverage (319×) or moderate coverage (~30×) data, Apollo's polishing score differs only by at most ~1.21%. We conclude that Racon performs better than the competing state-of-the-art polishing algorithms if the coverage of a set of reads is high (e.g. 319×). Apollo outperforms Nanopolish when polishing a Miniasm-generated assembly but Nanopolish outperforms Racon and Apollo when polishing a Canu-generated assembly. Thus, we also conclude that the accuracy of the original assembly dramatically affects the overall performance of Nanopolish as there is a significant performance difference between polishing Miniasm and polishing Canu assemblies. We suspect that the default parameter settings of Apollo may be a better fit for PacBio reads rather than ONT reads, which explains why Apollo performs worse with ONT datasets compared to PacBio datasets.

Apollo is robust to different parameter choices. In Supplementary Tables S19–S21, we use the *E.coli* O157 dataset to examine if Apollo is robust to using different parameter settings. To study the change in the performance of Apollo, we change the following parameters: maximum number of states that the Forward–Backward and the Viterbi algorithms evaluate for the next time step (*f*), number of insertion states per base pair (*i*), maximum deletion length per transition (*d*), transition probability to a match state (*tm*), transition probability to an insertion state (*ti*). We conclude that Apollo's performance is robust to different parameter choices as the accuracies of the Apollo-polished assemblies differ by at most 2%.

3.5 Reference-independent quality assessment

We report both (i) the *k*-mer distance [i.e. Jaccard similarity (Niwattanakul *et al.*, 2013) or *k-mer similarity*] between *filtered* Illumina reads and assemblies and (ii) quality assessment based on mapping these *filtered* Illumina reads to assemblies to provide a reference-independent comparison between the polishing tools. We filter Illumina reads in three steps to get rid of erroneous short reads before using them. First, we remove the adapter sequences (i.e. adapter trimming). Second, we apply contaminant filtering for synthetic molecules. Third, we map the reads generated after the first three steps to the reference and filter out the reads that do not map to the reference. We use BBTools (sourceforge.net/projects/bbmap/) in these steps of filtering. To calculate *k*-mer similarity, we also use trim-low-abund (Zhang *et al.*, 2015), which applies *k*-mer abundance trimming to remove *k*-mers with abundance lower than 10 for *E.coli* and Yeast datasets, and 3 for the human genome.

In *k*-mer similarity calculations, Jaccard similarity provides how a set of *k*-mers of both Illumina reads and an assembly are similar to each other. We compare the filtered Illumina reads with both polished and original (i.e. unpolished) assemblies of the small genomes (i.e. Yeast and *E.coli*) and the large genomes (i.e. human); the results are in Supplementary Tables S4, S7, S10, S13 and S15. We show the percentage of both the *k*-mers of Illumina reads present in the assembly and the *k*-mers of the assembly present in Illumina reads. The latter helps us to identify *how accurate* the assembly is whereas the former shows the *completeness* of the assembly.

Based on our experiments on small genomes, we make three key observations. First, the tool with the highest *assembly accuracy*, estimated with *k*-mer similarity (Supplementary Tables S4, S7, S10 and S13), typically provides the highest *polishing score* in its category (Supplementary Tables S3, S6, S9 and S12), respectively. Second, Quiver usually produces more accurate assemblies than the assemblies generated by other polishing tools. Third, all polishing algorithms we evaluate dramatically increase the accuracy of the unpolished assembly generated by Miniasm. We conclude that the *k*-mer similarity results correlate with our findings in Section 3.4 and support our claims regarding how polished assemblies compare with the ground truth.

Based on the *k*-mer similarity results between the Illumina reads and the human genome assemblies, we make five key observations. First, we observe a reduction in the accuracy when polishing algorithms use raw PacBio reads as the finished assembly was generated using *corrected* PacBio reads and *already polished* by Quiver. Second, the polishing algorithms produce more accurate assemblies than the finished assembly *only when* they use short reads to polish an assembly. This is because (i) Illumina reads are more accurate than raw PacBio reads and (ii) Illumina reads have not been used when polishing the HG002 assembly, which leaves room to improve the accuracy. Third, Apollo performs better than Racon in terms of *both* the completeness and the accuracy of the polished assemblies and better than Quiver in terms of accuracy (based on 51-mer results). Fourth, Apollo performs better than Pilon when it polishes the assembly using short reads. Fifth, using a low coverage readset to polish a human genome assembly dramatically reduces both the completeness of the assembly and the accuracy of the assembly. We conclude that (i) Apollo outperforms Pilon on Illumina data, and (ii) it is not advisable to use raw PacBio reads to polish the large genome assemblies that have already been polished using more accurate reads than the raw PacBio reads (e.g. corrected PacBio reads).

We use QUAST (Gurevich *et al.*, 2013), a quality assessment tool for genome assemblies, to provide a different reference-independent assessment of the assemblies. QUAST takes paired-end *filtered* Illumina reads to generate several metrics such as percentage of (i) mapped reads, (ii) properly paired reads, (iii) average depth of coverage and (iv) bases with at least 10× coverage. It also calculates the GC content (i.e. the ratio of bases that are either G or C) of the assembly. Based on the quality assessment results that we show in Supplementary Tables S5, S8, S11, S14 and S16, we make two key observations. First, for human genome assemblies, Apollo performs better than Racon and comparable to Pilon in terms of the percentage of the mapped reads, properly paired reads and the bases with at least 10× read coverage. Second, for small genomes (i.e. Yeast and *E.coli*), Quiver usually performs best in all of the metrics. We conclude that Apollo provides better performance when polishing large genomes than Racon, and Quiver usually performs better than any other polishing algorithm for small genomes.

3.6 Computational resources

We report the runtimes and the maximum memory requirements of both assemblers and assembly polishing algorithms in

Supplementary Tables S1–S3, S6, S9 and S12. Based on the runtimes of *only* assembly polishing algorithms (i.e. Apollo, Nanopolish, Pilon, Quiver and Racon), we make three observations. First, the machine learning-based assembly polishing tools, Apollo and Nanopolish, are the most time-consuming algorithms due to their computationally expensive calculations. For example, Racon is ~75× and ~15× faster than Apollo when polishing Miniasm-generated assemblies using PacBio and ONT readsets, respectively. Second, Racon becomes more memory-bound as the overall number of long reads in a readset increases (shown in Table 2). This shows that Racon's memory requirements are directly proportional to the size of the readset (i.e. the overall number of base pairs in a readset). Third, Quiver always requires the least amount of memory for *E.coli* and Yeast genomes compared to the competing algorithms.

In Supplementary Tables S1 and S2, we evaluate the overall runtime and memory requirements of (i) polishing an assembly within a single run by using a hybrid set of reads with Apollo and (ii) polishing an assembly multiple times. We observe that the overall runtime of running polishing tools multiple times is still lower at least by an order of magnitude than running Apollo once with a hybrid set of reads. However, Apollo can provide a more accurate assembly for a species when a Canu-generated assembly is polished, as discussed in Section 3.4.

We report the runtimes, maximum memory requirements and the parameters of the aligners we evaluated in Supplementary Tables S17 and S22, respectively, to observe how the aligner affects the overall runtime of both the aligner the assembly polishing tool. Based on the runtimes of aligners, we make two observations. First, pbalign is the most time-consuming and memory-demanding alignment tool. Overall, this makes Quiver require more time and memory than Racon, since Quiver can only work with BLASR, a part of pbalign tool. Second, all evaluated polishing tools except Quiver allow using any aligner; therefore, we only compare the runtime of the polishing tools, rather than comparing runtime of the full pipeline (i.e. aligner plus polishing tool) for the non-human genome datasets. We conclude that Quiver is the only algorithm whose runtime must be considered in conjunction with the aligner, as it can only use one aligner, pbalign, which we show in Table 2.

4 Discussion

We show that there is a dramatic difference between non-machine learning-based algorithms and the machine learning-based ones in terms of runtime. Apollo and Nanopolish usually require several hours to complete the polishing. Racon, Quiver and Pilon usually require less than an hour (Supplementary Tables S1–S3, S6, S9 and S12), which may suggest that Racon and Pilon can use a hybrid set of reads to polish an assembly in multiple runs instead of using Apollo in a single run. Indeed, we confirm that running Racon, Pilon or Quiver multiple times still takes a much shorter time than running Apollo once using a hybrid set of reads within a single run. However, assembly polishing is a one-time task performed for an assembly that is usually used *many* times and even made publicly available to the community. Therefore, we believe that long runtimes could still be acceptable given that genomic data produced by Apollo will probably be used *many* times after it is generated. Hence, Apollo's runtime cost is paid only once but benefits are reaped many times. Note that this observation is not restricted to Apollo and applies to any polishing tool that has a long runtime. In addition, it is possible to accelerate the calculation of the Forward–Backward algorithm and the Viterbi algorithm using Tensor cores, SIMD and GPUs (Eddy, 2011; Liu, 2009; Murakami, 2017; Yu *et al.*, 2014), which we leave to future work.

Despite these slower runtimes of Apollo compared to other polishing tools, Apollo is new, unique and useful because it provides two major functionalities that are not possible with prior tools. First, Apollo is the *only* algorithm that can scale itself well to polish a large genome assembly using a readset with moderate coverage (e.g. up to ~35×) set of reads. Therefore, it is possible to polish a large genome with a relatively small amount of memory (i.e. <110 GB) only with Apollo. Second, Apollo can construct more reliable Canu-generated assemblies compared to running other polishing tools multiple times when both PacBio and Illumina reads are used (i.e. a hybrid set of reads). These two advantages are *only* possible if Apollo is used for assembly polishing.

5 Conclusion

In this paper, we present a universal, sequencing-technology-independent assembly polishing algorithm, Apollo. Apollo uses all available reads to polish an assembly and removes the dependency of the polishing tool on sequencing technology. Apollo is the first polishing algorithm that scales well to use any arbitrary hybrid set of reads *within a single run* to polish both large and small genomes. Apollo also removes the requirement of using assembly polishing algorithms multiple times to polish an assembly as it allows using a hybrid set of reads.

We show three key results. First, three state-of-the-art polishing algorithms, Quiver, Racon and Pilon, cannot scale well to polish large genome assemblies without splitting the assembly into its contigs or readsets into smaller batches whereas Apollo scales well to polish large genomes. Second, using a hybrid set of reads with Apollo usually results in constructing Canu-generated assemblies more accurate than those generated when running other polishing tools multiple times. Third, Apollo usually polishes assemblies with comparable accuracy to state-of-the-art assembly polishing algorithms with a few exceptions that occur when long reads are used to polish Miniasm-generated assemblies. We conclude that Apollo is the first universal, sequencing-technology-independent assembly polishing algorithm that can use a hybrid set of reads within a single run to polish both large and small assemblies, while achieving high accuracy.

Funding

This work was supported by gifts from Intel [O.M.]; VMware [O.M.]; and the funding from TÜBİTAK [TÜBİTAK-1001-215E172 to C.A.].

Conflict of Interest

None declared.

References

`Alkan,C. *et al.* (2011) Limitations of next-generation genome sequence assembly. *Nat. Methods*, **8**, 61–65.

Alser,M. *et al.* (2017) GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics*, **33**, 3355–3363.

Alser,M. *et al.* (2019a) Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, **35**, 4255–4263.

Alser,M. *et al.* (2019b) *SneakySnake: A Fast and Accurate Universal Genome Pre-alignment Filter for CPUs, GPUs, and FPGAs*. arXiv e-prints, arXiv: 1910.09020.

Au,K.F. *et al.* (2012) Improving PacBio long read accuracy by short read alignment. *PLoS One*, **7**, e46679.

Baum,L.E. (1972) An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, **3**, 1–8.

Berlin,K. *et al.* (2015) Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.*, **33**, 623–630.

Brown,CT. and Irber,L. (2016) sourmash: a library for MinHash sketching of DNA. *J. Open Source Softw.*, **1**, 27.

Chaisson,M. *et al.* (2004) Fragment assembly with short reads. *Bioinformatics*, **20**, 2067–2074.

Chaisson,M.J. and Tesler,G. (2012) Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics*, **13**, 238.

Chaisson,M.J.P. *et al.* (2015) Genetic variation and the *de novo* assembly of human genomes. *Nat. Rev. Genet.*, **16**, 627–640.

Chin,C.-S. *et al.* (2013) Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat. Methods*, **10**, 563–569.

Döring,A. *et al.* (2008) SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**, 11.

Eddy,S.R. (2011) Accelerated profile HMM searches. *PLoS Comput. Biol.*, **7**, e1002195.

Firtina,C. and Alkan,C. (2016) On genomic repeats and reproducibility. *Bioinformatics*, **32**, 2243–2247.

Firtina,C. *et al.* (2018) Hercules: a profile HMM-based hybrid error correction algorithm for long reads. *Nucleic Acids Res.*, **46**, e125.

Glenn,T.C. (2011) Field guide to next-generation DNA sequencers. *Mol. Ecol. Resour.*, **11**, 759–769.

Gurevich,A. *et al.* (2013) QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, **29**, 1072–1075.

Huddleston,J. *et al.* (2014) Reconstructing complex regions of genomes using long-read sequencing technology. *Genome Res.*, **24**, 688–696.

Jain,M. *et al.* (2018) Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nat. Biotechnol.*, **36**, 338–345.

Kim,J.S. *et al.* (2018) GRIM-Filter: fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics*, **19**, 89.

Koren,S. *et al.* (2012) Hybrid error correction and *de novo* assembly of single-molecule sequencing reads. *Nat. Biotechnol.*, **30**, 693–700.

Koren,S. *et al.* (2017) Canu: scalable and accurate long-read assembly via adaptive *k* -mer weighting and repeat separation. *Genome Res.*, **27**, 722–736.

Kurtz,S. *et al.* (2004) Versatile and open software for comparing large genomes. *Genome Biol.*, **5**, R12.

Li,H. (2016) Minimap and miniasm: fast mapping and *de novo* assembly for noisy long sequences. *Bioinformatics*, **32**, 2103–2110.

Li,H. (2018) Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**, 3094–3100.

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

Li,H. *et al.* (2009) The sequence alignment/map format and SAMtools. *Bioinformatics*, **25**, 2078–2079.

Liu,C. (2009) cuHMM: a CUDA implementation of hidden Markov Model training and classification. *Chron. High. Educ.*, 1–13.

Loman,N.J. *et al.* (2015) A complete bacterial genome assembled *de novo* using only nanopore sequencing data. *Nat. Methods*, **12**, 733–735.

Meltz Steinberg,K. *et al.* (2017) Building and improving reference genome assemblies. *Proc. IEEE*, **105**, 1–14.

Murakami,T. (2017) Expectation–maximization tensor factorization for practical location privacy attacks. *Proc. Privacy Enhancing Technol.*, **2017**, 138–155.

Niwattanakul,S. *et al.* (2013) Using of Jaccard coefficient for keywords similarity. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists.* Newswood Limited, Hong Kong, Vol. **1**, pp. 380–384.

Payne,A. *et al.* (2018) BulkVis: a graphical viewer for Oxford nanopore bulk FAST5 files. *Bioinformatics*, **35**, 2193–2198.

Pearson,W.R. and Lipman,D.J. (1988) Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, **85**, 2444–2448.

Rhoads,A. and Au,K.F. (2015) PacBio sequencing and its applications. *Genomics Proteomics Bioinform.*, **13**, 278–289.

Salmela,L. and Rivals,E. (2014) LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, **30**, 3506–3514.

Salmela,L. *et al.* (2016) Accurate self-correction of errors in long reads using de Bruijn graphs. *Bioinformatics*, **33**, 799–806.

Sanger,F. *et al.* (1977) DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci.*, **74**, 5463–5467.

Senol Cali,D. *et al.* (2019) Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions. *Brief. Bioinform.*, **20**, 1542–1559.

Vaser,R. *et al.* (2017) Fast and accurate *de novo* genome assembly from long uncorrected reads. *Genome Res.*, **27**, 737–746.

Viterbi,A. (1967) Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inf. Theory*, **13**, 260–269.

Walker,B.J. *et al.* (2014) Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PLoS One*, **9**, e112963.

Weirather,J.L. *et al.* (2017) Comprehensive comparison of Pacific Biosciences and Oxford Nanopore Technologies and their applications to transcriptome analysis. *F1000Research*, **6**, 100.

Wenger,A.M. *et al.* (2019) Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nat. Biotechnol.*, **37**, 1155–1162.

Xin,H. *et al.* (2013) Accelerating read mapping with FastHASH. *BMC Genomics*, **14**, S13.

Yu,L. *et al.* (2014) GPU-accelerated HMM for speech recognition. In: *2014 43rd International Conference on Parallel Processing Workshops.* IEEE, Minneapolis, MN, USA, pp. 395–402.

Zhang,Q. *et al.* (2015) Crossing the streams: a framework for streaming analysis of short DNA sequencing reads. *PeerJ PrePrints*, **3**, e890v1.