Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

Subject Section

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 15213, PA, USA
[3]Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Third-generation sequencing technologies can sequence long reads that contain as many as 900K base pairs (bp). 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 bps in these long reads are *incorrectly* identified. These errors propagate to the assembly and affect the accuracy of genome analysis. *Assembly polishing algorithms* minimize 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, currently available assembly polishing algorithms can only polish an assembly using reads either from a certain sequencing technology or from a small genome. This technology and genome-size dependency prevents state-of-the-art assembly polishing algorithms from either (1) using all the available read sets from multiple sequencing technologies or (2) polishing large genomes (e.g., a human genome).

Results: We introduce Apollo, a *universal* assembly polishing algorithm that is scalable to polish an assembly of *any* size (i.e., both large and small genomes) with 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 1) models an assembly as a profile hidden Markov model (pHMM), 2) uses read-to-assembly alignment to train the pHMM with the Forward-Backward algorithm, and 3) decodes the trained model with the Viterbi algorithm to produce a polished assembly. Our experiments with real read sets demonstrate that 1) using reads from multiple sequencing technologies produces a more accurate assembly compared to using reads from only a single sequencing technology, and 2) Apollo is the *only* algorithm that can use reads from any sequencing technology within a single run and that can polish an assembly of any size, and 3) Apollo performs better than or comparable to the competing state-of-the-art algorithms in terms of accuracy *even when* polishing with a set of reads from a single sequencing technology.

Contacts: onur.mutlu@inf.ethz.ch, calkan@cs.bilkent.edu.tr

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

Availability: The source code is available at https://github.com/CMU-SAFARI/Apollo

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 1) read alignment, the process of aligning the reads to a *reference genome*, a genome representative of all individuals within a species, or 2) *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 into reads. Depending on the method for reconstructing the original sequence, HTS errors often cause either 1) reads aligned to an incorrect location in the reference genome, or 2) 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: (1) second-generation and (2) 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 are short (∼100-300bp) (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 (Xin *et al.*, 2013; Alser *et al.*, 2017; Kim *et al.*, 2018). Aligners must either perform an additional computation to make a deterministic choice for the matching locations or select one of the candidate locations randomly such that the read alignments may be non-reproducible (Firtina and Alkan, 2016). In de novo genome assembly, there is high computational complexity in identifying the 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 (SMRT) and Oxford Nanopore Technologies (ONT)) are capable of producing long reads (∼10Kbps on average and up to 900Kbps) at the cost of a high error rate (∼15% error rate) (Huddleston *et al.*, 2014; Jain *et al.*, 2018). 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, the 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. First, **a typical solution is to correct the errors of long reads**. Errors are corrected by using high coverage reads (e.g., ∼150X coverage) from the same sequencing technology or additional reads from more reliable second-generation sequencing technologies. There are several available *error correction* algorithms that use additional reads to locate and correct the 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 is that error correction algorithms require *more* sequenced reads from either the same or different sequencing technologies. In both cases, this means additional cost and time. While a higher-coverage data set may lead to increased read accuracy (Berlin *et al.*, 2015), the cost of producing a high-coverage data set for long reads is often prohibitively high (Rhoads and Au, 2015). For example, sequencing the human genome with ONT at only even 30X 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 modification corrects, or *polishes*, 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.*, 2018), Quiver supports *only* PacBio long reads. This makes these assembly polishing algorithms 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., ONT and PacBio reads) within a single run.

The dependency of an assembly polishing algorithm on sequencing technology is problematic because the algorithm *cannot* take advantage of all possible read sets that may be available for a single genome for assembly polishing. This leaves a significant amount of information out of the assembly. For example, if both PacBio long reads and Illumina short reads are available for the same sample, Nanopolish can use *only* ONT long reads and *cannot* use Illumina short reads for polishing. Similarly, Quiver can use only PacBio long reads and *cannot* take advantage of the available Illumina short reads.

While the technology-dependency problem of such assembly polishing algorithms could be mitigated by sequentially 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 problems associated with running assembly polishing algorithms multiple times and using polishing algorithms to polish a large genome. First, running different assembly polishing algorithms sequentially or even running the same algorithm multiple times requires additional computational resources. For example, once an assembly is polished, an aligner should generate a new index file for the polished assembly since reads need to re-align to the polished assembly to polish it again for the next round. Therefore, this re-alignment and generation of the index file each time before assembly polishing requires additional runtime, which is at least an hour and up to around 4 hours for large genomes. Second, none of the polishing algorithms can polish large genomes (e.g., a human genome) unless the coverage of the set of reads is low (e.g., less than 10X), due to

high amount of computational resources that they require. Therefore, these assembly polishing algorithms *cannot* scale well to a large genome, and they are restricted to use a low coverage set of reads of a large genome, which causes inaccuracy.

A *universal technology-independent assembly polishing algorithm* that can use reads regardless of both the sequencing technology used to produce them and the size of a genome provides the potential to use all available reads for a more accurate assembly compared to using reads from a single sequencing technology and to polish an assembly of *any* size. 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 within a single run and that is scalable to polish an assembly of any size (i.e., both small and large genome assemblies). **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: (1) training and (2) decoding the profile hidden Markov model (pHMM) of an assembly. First, Apollo uses the Forward-Backward algorithm (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), a state-of-the-art algorithm 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 the data sets that are sequenced with different technologies: Escherichia coli K-12 (MinION), Escherichia coli O157 (PacBio and Illumina), a human hydatidiform mole CHM1 cell line (PacBio) (Steinberg *et al.*, 2014), and the human Ashkenazim trio sample (HG002, PacBio and Illumina). We use highly accurate and finished genome assemblies of the corresponding samples to determine the accuracy of the various assembly polishing algorithms.

Using the data sets from different sequencing technologies, we first show that Apollo is the *only* algorithm that can polish assemblies of both large and small genomes using moderate and high coverage reads, respectively. Second, it is the *only* algorithm that can use reads from *multiple* sequencing technologies in a *hybrid* manner (e.g., using both long ONT and short Illumina reads in a single run). Because of this, Apollo scales 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 using a hybrid set of reads produces more accurate assemblies than using only a single type of reads. Fourth, when we compare Apollo to other competing algorithms, our experiments show that Apollo is *more* accurate than Pilon using only short reads. For the PacBio and ONT data sets, Apollo produces either slightly more or slightly less accurate assemblies than the competing algorithms: Nanopolish, Racon, and Quiver. These comparisons show that Apollo can polish an assembly using reads from multiple sequencing technologies and it still generates an assembly with comparable accuracy to the competing algorithms. Fourth, we use moderate long read coverage data sets (e.g., 30X) to 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 1) is scalable to polish the assemblies of both large and small genomes, and 2) 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, an 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 is scalable to polish assemblies of both large and small genomes, and that has support for assembly correction using a hybrid set of reads to polish an assembly. For example, one can align both Illumina and PacBio reads to the same assembly and Apollo can use the resulting set of alignments to polish the assembly.
- We show that using both long and short reads in a hybrid manner to polish the assembly produces a more accurate assembly compared to using a single type of reads (e.g., using only Illumina reads).
- We demonstrate that Apollo produces more accurate assemblies than Pilon and has comparable accuracy to the other assembly polishing algorithms (Nanopolish, Racon, and Quiver) when Apollo uses only the reads that the competing algorithms support.
- We show that the competing polishing algorithms fail to polish assemblies of large genomes due to high computational resources that they require.
- We provide an open source implementation of Apollo (https://github.com/CMU-SAFARI/Apollo)

2 Methods

Apollo builds, trains, and decodes a profile hidden Markov model graph (pHMM-graph) to polish an assembly (i.e., correcting 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 is comprised of states, transitions between states, and probabilities associated with both states and transitions to account for all possible error types. There are three types of errors that a sequencing technology can introduce into a read: insertion, deletion, and substitution errors. Therefore, correction of these errors can be accomplished by deleting, inserting, or substituting the corresponding base pair, respectively. Apollo identifies a path in the pMHM-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 algorithm. 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 in order to merge the overlapped regions into larger fragments called contigs. An assembler usually reports contigs in the FASTA format (Pearson and Lipman, 1988) where each element is comprised of an ID and the full sequence of the contig. The entire collection of contigs represents the whole assembly. Apollo requires the assembly to

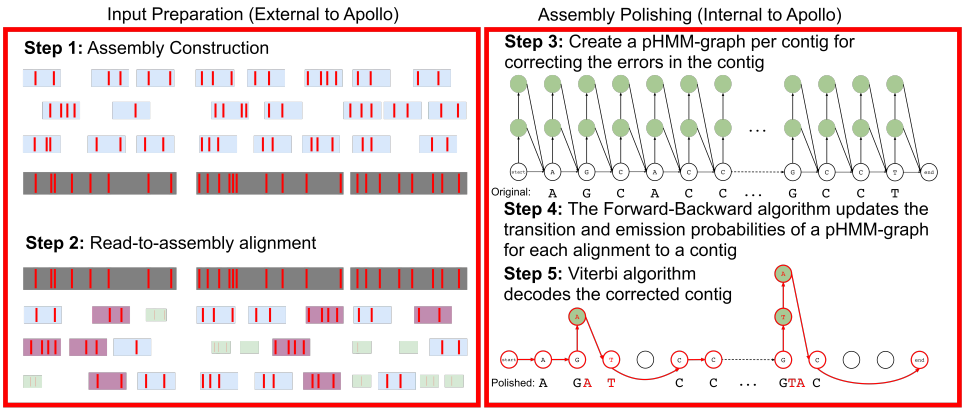

Fig. 1. Input preparation and the pipeline of Apollo algorithm in five steps. The first two steps refer to the use of external tools to generate the input for Apollo and are called input preparation steps (left side). (Step 1) An assembler generates the assembly (dark gray, large rectangles) using erroneous reads (light blue rectangles). Here the errors are labeled 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 colors 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 profile hidden Markov model graph (pHMM-graph) per assembly contig. Here, we show the pHMM-graph generated for the contig that starts with "$AGCACC$" and ends with "$GCCT$" as we show the original sequence below the states labeled with a base pair. Each base pair in a contig is represented by a state labeled with the corresponding base pair. A pHMM graph also consists of insertion states for each basepair labeled with green color as well as start and end states that do not correspond to any base pair in a contig. 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 color 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.

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 (Figure 1 step 1). Apollo supports the use of any assembler that can produce the assembly in FAST[A,Q] 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 (1) the reads that the assembler used to construct the assembly and (2) any additional reads sequenced from the same sample (Figure 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 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 in SAM/BAM format is coordinate sorted.

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 behavior 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, and the complete structure of a pHMM-graph allows Apollo to handle all possible error types: 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 number of *insertion states* for each base pair in the contig where l is a parameter to Apollo, and it 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) in order to handle a *deletion error*. Last, each match and insertion state has k number of *deletion transitions* where k is also a parameter to Apollo and 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 from a state to a match state skips the match states between the two states in order to delete the corresponding base pairs of the skipped match states. Further details of the pHMM-graph can be found in Supplementary Materials (Section 1).

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. However, importantly, 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 generally longer than reads. Therefore,

Apollo uses *additional methodologies* 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 regarding implementation as we explain in Section 2.4, Section 2.5, and in Section 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 the *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. 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. 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. This is to account for the case where there are more insertion errors than deletion errors. 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 can fluctuate dramatically (e.g., from 15bps to 900Kbps) whereas the length of short reads is usually fixed (e.g., 100bps). 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 all lengths. Apollo decides whether the aligned read is *short* or *long* based on the read length, of which we set the threshold at 500bps (i.e., if a read is longer than 500bps, it is considered as a long read). If the aligned read length is *short* (i.e., shorter than 500bps), 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 algorithm (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: (1) Forward calculation, (2) Backward calculation, and (3) updating the probabilities (i.e., the expectation-maximization step). 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*.

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 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. Sections 2 and 3 in the Supplementary Materials describe in detail how Apollo locates a sub-graph per read alignment as well as 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 in order 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 (Figure 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. Thus, the complete dynamic 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 (i.e., decoding).

The Viterbi algorithm computes each entry of the dynamic table using the Viterbi value 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 including a certain number of states. The Viterbi algorithm decodes each sub-graph and merges decoding results into one piece again, leading to a sub-optimal solution of the pHMM-graph. Since the Viterbi algorithm can decode each sub-graph independently, this allows Apollo parallelize the Viterbi algorithm. We find that our parallelization greatly speeds up the Viterbi algorithm, by $\sim 20X$.

For each state in the identified path, Apollo outputs the base pair with the highest probability. The sequence of the outputs corresponds to the polished contig, and Apollo reports each polished contig as a read in FASTA format. Details of the Viterbi algorithm can be found in Supplementary Materials (Section 4).

Note that Apollo can only polish contigs that at least a single read aligns to. Thus, Apollo reports an unpolished version of a contig, if there is no read aligning to it.

3 Results

3.1 Experimental Setup

We implement 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 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*. An *accuracy* value provides the accuracy of only the aligned portions of the polished assembly, not the entire assembly. However, the polishing score is a more comprehensive measure compared to the accuracy, as it normalizes the accuracy of the aligned portions of the polished assembly for the entire length of the assembly. We also report runtime and memory usage of the assembly polishing algorithms. The operating system we use (Debian GNU/Linux 9) provides runtime (wall clock time) and peak memory usage. Based on our evaluation criteria, we compare Apollo to the 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 data set, we do not run the algorithm on this data set. For example, we use Nanopolish only for the ONT data set and Quiver only for PacBio data sets, and Pilon only for the Illumina data set. We use Pilon with a PacBio data set 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 data set, we compare the algorithms that polish an assembly using the same set of reads.

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 @ 2.30GHz), and 192GB of the memory. We assign 45 threads to all the tools we use and collect their runtime and memory usage using *time* command in Linux with $-vp$ options.

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 a read-to-assembly alignment, we use Minimap2 to align long and short reads to an assembly and BWA-MEM for only short reads. However, 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 in order to generate a read-to-assembly alignment in the format that Quiver requires. We sort the resulting SAM/BAM read-to-assembly alignments using the SAMtools' sort command (Li *et al.*, 2009).

After assembly generation, we divide the long reads into smaller *chunks* of size 1000bps (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 also show that chunking results in producing more accurate assemblies (Supplementary Table S6).

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 basepairs 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 ($em = 0.97$).

We use the *dnadiff* tool provided under MUMmer package (Kurtz *et al.*, 2004) to calculate the accuracy of resulting assemblies by comparing them with the highly-accurate reference genomes. We also use BLASR with `bestn=1` and `noSplitSubreads` options to calculate the accuracy of a human genome assembly, since dnadiff cannot scale to a large genome. We run each assembly polishing algorithm with its default parameters.

3.2 Data Sets

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

We use a data set 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 *only* the computational resources that each polishing algorithm requires (i.e., time and maximum memory usage). We filtered out the PacBio reads that have a length of less than 200 before calculating coverage and the average read length.

We evaluate the polishing accuracy of Apollo and other state-of-the-art polishing algorithms in three ways. First, we use the E.coli O157 and E.coli O157:H7 data sets to demonstrate whether using a hybrid set of reads with Apollo results in more accurate assemblies compared to using a non-hybrid set of reads (e.g., only PacBio reads). Second, we use the E.coli K12, E.coli O157 (Strain FDAARGOS_292), and Human CHM1 cell line data sets *to compare Apollo with the state-of-the-art polishing algorithms*. Third, we subsample the E.coli K-12 and E.coli O157 data sets into 30X coverage *to compare the performance of algorithms when long read coverage is moderate*.

We use the assembly of the human CHM1 cell line, and reference genomes for human (GRCh38) and zebra fish (GRCz11) to calculate the time required to construct their index files with BWA (Li and Durbin, 2009) and Bowtie2 (Langmead and Salzberg, 2012). We evaluate if indexing an assembly increases the overall runtime to polish an assembly multiple times.

3.3 Applicability of the Polishing Algorithms to Large Genomes

We use the polishing algorithms to polish a large genome assembly (e.g., a human genome) to observe (1) whether the polishing algorithms can polish these large assemblies without exceeding the limitations of the computational resources we use to conduct our experiments and (2) the overall computational resources required to polish a large genome assembly. 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. Based on our experiments that we report in Supplementary Table S3, we make the two key observations. First, we observe that Racon, Pilon, and Quiver *cannot* polish the assembly using the sets of PacBio ($\sim 35X$ coverage) and Illumina ($\sim 22X$ 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 the memory limitations. Apollo can polish an assembly using *both* PacBio and Illumina reads using nearly at most half of the available memory. Second, we reduce the coverage of the PacBio reads to 8.9X (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 *only* using low coverage set of reads whereas BLASR *cannot* produce the alignment results that Quiver requires due to the memory limitations even when using a low coverage set of reads. We conclude that Apollo is the *only algorithm* that scales to polish large genomes using set of both PacBio and Illumina reads even when the coverage is moderate (i.e., $\sim 22X$ and $\sim 35X$). Racon can only polish a large genome assembly if the coverage of PacBio reads is low (e.g, 8.9X). Pilon and Quiver fail to scale their assembly polishing algorithms to a large genome.

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 using a set of reads from only a single sequencing technology (e.g., only PacBio reads). Second, we evaluate assembly polishing algorithms and compare them to each other given different options with respect to (1) the sequencing technology that produces long reads, (2) the assembler that constructs an assembly using long reads, (3) the aligner that generates read-to-assembly alignment, (4) 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 our evaluation criteria that we explained in Section 3. We also compare the tools based on their performance given

Table 1. Details of the Data Sets

Data Set	Accession Number	Details
E.coli K12 - ONT	Loman Lab*	164,472 reads (avg. 9,010bps, 319X coverage) via Metrichor
E.coli K12 - Ground Truth	GenBank NC_000913	Strain MG1655 (4,641Kbps)
E.coli O157 - PacBio	SRA SRR5413248	177,458 reads (avg. 4,724bps, 151X coverage)
E.coli O157 - Illumina	SRA SRR5413247	11,856,506 paired-end reads (150bps each, 643X coverage)
E.coli O157 - Ground Truth	GenBank NJEX02000001	Strain FDAARGOS_292 (5,566Kbps)
E.coli O157:H7 - PacBio	SRA SRR1509640	76,279 reads (avg. 8,270bps, 112X coverage)
E.coli O157:H7 - Illumina	SRA SRR1509643	2,978,835 paired-end reads (250bps each, 265X coverage)
E.coli O157:H7 - Ground Truth	GCA_000732965	Strain EDL933 (5,639Kbps)
Human CHM1 - PacBio	SRA SRR130433(1-5)	912,421 reads (avg. 8,646bps, 2.6X coverage)
Human CHM1 - Ground Truth	GCA_000306695.2	3.04Gbps
Human HG002 - PacBio	SRA SRR2036(394-471), SRR203665(4-9)	15,892,517 reads (avg. 6,550bps, 35X coverage)
Human HG002 - Illumina	SRA SRR17664(42-59)	222,925,733 paired-end reads (148bps each, 22X coverage)
Human HG002 - Ground Truth	GCA_001542345.1	Ashkenazim trio - Son (2.99Gbps)

We list the data sets we use in our experiments. The data can be accessed through NCBI using the accession number. * The ONT data sets are available at http://lab.loman.net/2016/07/30/nanopore-r9-data-release/

moderate (e.g., $\sim 30X$) and low long read coverage (e.g., 2.6X, CHM1 data set).

Polishing using a hybrid set of reads generates more accurate assemblies than using a single set of reads. In Table 2, we investigate the benefits of using a hybrid set of reads (e.g., PacBio + Illumina) within a single polishing run to polish an assembly over using a set of reads from only a single sequencing technology (e.g., only PacBio or only Illumina). To this end, we evaluate the performance of Apollo in terms of the accuracy of polished assemblies using a hybrid and a non-hybrid (e.g., only Illumina) set of reads. We use long (PacBio) and short (Illumina) reads from E.coli O157 and E.coli O157:H7 data sets. Based on Table 2, we make two key observations. First, we observe that Apollo produces more accurate assemblies (see the *Accuracy* column) when reads are in a hybrid form over using a single type of reads to polish an assembly. Second, a larger portion of the assemblies that Apollo polishes using a hybrid set of reads aligns to the ground truth (see the *Aligned Bases* column) than that of the assemblies that Apollo polishes using a single type of reads. As a result, the polishing score of Apollo is always the highest when it uses a hybrid set of reads. We conclude that the use of a hybrid set of reads contributes to the assembly polishing in a way that it enables the construction of more accurate assemblies over using a single type of reads.

Table 2. Advantage of using a hybrid set of reads from different technologies

Data Set	Sequencing Tech. of the Reads	Aligned Bases (%)	Accuracy	Polishing Score
E.Coli O157	PacBio	98.49	0.9798	0.9650
E.Coli O157	Illumina	97.61	0.9816	0.9581
E.Coli O157	PacBio + Illumina	**98.70**	**0.9866**	**0.9738**
E.Coli O157:H7	PacBio	96.99	0.9636	0.9346
E.Coli O157:H7	Illumina	96.06	0.9781	0.9396
E.Coli O157:H7	PacBio + Illumina	**97.53**	**0.9804**	**0.9562**

We use the long reads of E.coli O157 and E.Coli O157:H7 data sets that are sequenced from PacBio (151X and 112X coverages, respectively) to generate their assemblies with Miniasm. Here, the reads specified under *Sequencing Tech. of the Reads* are sequenced by the specified sequencing technology and are aligned to an assembly using Minimap2. PacBio + Illumina constitute the hybrid set of reads. We report the performance of Apollo in terms of 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*) as calculated by dnadiff, and *Polishing Score* value that is the product of *Accuracy* and *Aligned Bases* (as a fraction). We show the best result in each performance metric in **bold** text.

Apollo performs better than Pilon and comparable to Racon and Quiver when polishing an assembly using only a set of PacBio or a set of Illumina reads. In Table 3, we use E.Coli O157 PacBio and Illumina data sets 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 data sets, we make five observations. First, we use a set of short reads (i.e., Illumina reads) to compare Apollo with Pilon. We use Minimap2 and BWA-MEM to align short reads to the Miniasm- and Canu-generated assemblies. Overall, Apollo outperforms Pilon (see the *Polishing Score* column) using a set of short reads. Second, we notice that Apollo, Racon, and Quiver show significant improvement over the original Miniasm assembly in terms of the accuracy. Third, we observe that Quiver and Racon polish the Miniasm-generated assembly more accurately than Apollo (see the *Accuracy* and the *Polishing Score* columns). Fourth, we report that Apollo produces more accurate assemblies than the assemblies polished by Racon when we use moderate ($\sim 30X$) and high coverage (151X) PacBio read sets to polish Canu-generated assemblies. However, we note that both algorithms generate assemblies with a lower accuracy than the accuracy of the original Canu-generated assembly when we use high coverage read sets (0.9998 with the polishing score of 0.9992). Based on this 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 original set of long reads as we report in Table 3, and Supplementary Tables S1 and S2. We find that the alignment of the 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 S4. 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 performance 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 a comparable performance 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 1) Apollo performs better when using short reads as it outperforms Pilon and Racon for almost all the data sets and 2) Apollo's performance is either marginally better or marginally worse than Racon and Quiver when using only PacBio reads to polish an assembly.

Apollo performs comparable to Racon and Nanopolish when polishing an assembly using only a set of ONT reads. We also investigate the performance of Apollo given the ONT data set (E.coli K-12), compared to Nanopolish and Racon. We make two key observations based on the

Table 3. Assembly polishing performance of the tools for the E.Coli O157 data set

Sequencing Tech. of the Assembly	Assembler	Aligner	Sequencing Tech. of the Reads	Polishing Algorithm	Aligned Bases (%)	Accuracy	Polishing Score	Runtime	Memory (GB)
PacBio	Miniasm	-	-	-	94.93	0.9000	0.8544	1m 48s	10.03
PacBio	Miniasm	Minimap2	PacBio	Apollo	98.49	0.9798	0.9650	2h 27m 49s	7.07
PacBio	Miniasm	Minimap2	PacBio	Pilon	96.43	0.9528	0.9188	1h 31m 32s	17.68
PacBio	Miniasm	Minimap2	PacBio	Racon	99.35	0.9951	**0.9886**	**2m 13s**	**2.44**
PacBio	Miniasm	pbalign	PacBio	Quiver	99.80	0.9993	**0.9973**	**7m 31s**	**0.51**
PacBio	Miniasm	Minimap2	Illumina	Apollo	97.61	0.9816	**0.9581**	4h 25m 17s	**9.22**
PacBio	Miniasm	Minimap2	Illumina	Pilon	96.52	0.9775	0.9435	32m 48s	18.60
PacBio	Miniasm	Minimap2	Illumina	Racon	96.45	0.9876	0.9525	**14m 90s**	21.57
PacBio	Miniasm	BWA-MEM	Illumina	Apollo	96.62	0.9738	0.9409	3h 32m 45s	**9.21**
PacBio	Miniasm	BWA-MEM	Illumina	Pilon	96.13	0.9693	0.9318	31m 21s	18.45
PacBio	Miniasm	BWA-MEM	Illumina	Racon	96.90	0.9813	**0.9509**	**12m 05s**	20.85
PacBio	Canu	-	-	-	99.94	0.9998	**0.9992**	43m 53s	3.79
PacBio	Canu	Minimap2	PacBio	Apollo	99.94	0.9997	0.9991	3h 42m 03s	8.82
PacBio	Canu	Minimap2	PacBio	Racon	99.94	0.9986	0.9980	**2m 17s**	**2.34**
PacBio	Canu	pbalign	PacBio	Quiver	99.94	0.9998	**0.9992**	**7m 06s**	**0.20**
PacBio	Canu	BWA-MEM	Illumina	Apollo	99.94	0.9999	**0.9993**	4h 49m 15s	**11.05**
PacBio	Canu	BWA-MEM	Illumina	Pilon	99.94	0.9998	0.9992	**2m 05s**	11.40
PacBio	Canu	BWA-MEM	Illumina	Racon	99.94	0.9999	**0.9993**	14m 58s	21.04
PacBio (30X)	Miniasm*	-	-	-	-	-	-	-	-
PacBio (30X)	Canu	-	-	-	99.98	0.9981	0.9979	21m 03s	3.70
PacBio (30X)	Canu	Minimap2	PacBio (30X)	Apollo	99.98	0.9982	**0.9980**	43m 32s	8.00
PacBio (30X)	Canu	Minimap2	PacBio (30X)	Racon	99.98	0.9980	0.9978	**15s**	**0.59**
PacBio (30X)	Canu	Minimap2	PacBio (30X, Corr.)	Apollo	99.97	0.9976	0.9973	46m 10s	7.99
PacBio (30X)	Canu	Minimap2	PacBio (30X, Corr.)	Racon	99.98	0.9983	**0.9981**	**7s**	**0.37**
PacBio (30X)	Canu	BWA-MEM	Illumina	Apollo	99.98	0.9997	0.9995	4h 48m 31s	10.35
PacBio (30X)	Canu	BWA-MEM	Illumina	Pilon	99.98	0.9998	**0.9996**	**3m 03s**	**8.52**
PacBio (30X)	Canu	BWA-MEM	Illumina	Racon	99.98	0.9997	0.9995	14m 42s	21.04

We generate the assembly for E.Coli O157 data set using the reads sequenced from PacBio (151X coverage) as specified in *Sequencing Tech. of the Assembly*. We subsample PacBio reads into 30X coverage and generate the assembly using the sub-sampled reads that we show as PacBio (30X). We use Canu and Miniasm assemblers as specified in *Assembler*. Here, the reads specified under *Sequencing Tech. of the Reads* are sequenced by the specified sequencing technology and are aligned to the assembly using the *Aligner*. Canu-corrected long reads are labeled as "Corr.". We report the performance of the tools in terms of 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*) as calculated by dnadiff, and a *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. For the rows that do not specify assembly polishing algorithms, we only report the runtime and the memory requirements of the assemblers as well as accuracy of the unpolished assembly that they construct. We show the best result in each performance metric in **bold** text. * denotes that Miniasm cannot produce an assembly given the specified set of reads.

results that we show in Supplementary Table S1. First, we observe that Racon provides the best performance in terms of the accuracy of contigs when the coverage is high (319X) 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 (319X) or moderate coverage data ($\sim 30X$), Apollo is also still in the comparable range as its accuracy only differs by $\sim 1.25\%$. We conclude that Racon performs better than the competing state-of-the-art polishing algorithms, if the coverage of a set of reads is very high (e.g., 319X). 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 data sets compared to PacBio data sets.

Apollo is robust to different parameter choices. In Supplementary Tables S7 - S9, we use the E.coli O157 data set 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 since the accuracies of the Apollo-polished assemblies differ by at most 2%.

Apollo still performs well when polishing an assembly using a low coverage set of reads. We further evaluate the performance of the algorithms given a set of low coverage long reads using the CHM1 data set, as shown in Supplementary Table S2. We make three key observations. First, we show that neither Canu nor Miniasm is able to generate a whole genome size assembly due to low coverage of long reads. Instead, Miniasm and Canu produce assemblies of length 1,581Kbps and 2,099Kbps only, respectively, whereas a human genome is around 3Gbps. Second, for the Miniasm-generated assembly, we find that Racon produces

the most accurate assembly, and Quiver and Apollo marginally improves the original assembly. Third, Apollo-polished assembly gives the best polishing score when the assembly is generated by Canu whereas other polishing algorithms produce assemblies even with lower polishing scores than the unpolished assembly. To conclude, we find that the use of the reads for an assembly that represents the small portion of a genome does not help to polish the assembly as most of the reads do not belong to the region that the assembly covers in the genome, which may result incorrect alignment of the read. Thus, we believe that assembly polishing algorithms can only improve the assembly marginally in such cases where the initial assembly is poor.

3.5 Computational Resources

We report the runtimes and the maximum memory requirements of both assemblers and assembly polishing algorithms in Table 3, and in Supplementary Tables S1 and S2. Note that the runtimes of polishing algorithms do not include the runtime of an assembler and the aligner. Based on the runtimes of *only* assembly polishing algorithms, we first find that the machine learning-based assembly polishing tools, Apollo and Nanopolish, are the most time-consuming algorithms due to their computationally expensive calculations. Second, Apollo is also a more memory computationally-demanding algorithm for small genomes, such as a bacterial genome. Racon becomes a more memory-bound algorithm as the size of the long reads increases, as shown in Supplementary Table S2. Third, we observe that Quiver requires always the least amount of memory compared to its competing algorithms.

We report the runtimes, maximum memory requirements, and the parameters of the aligners we used in Supplementary Tables S4 and S10, respectively, to observe how the aligner used affects the overall runtime of both the aligner the assembly polishing tool. Based on the runtimes of aligners, we make two observations. First, we observe that pbalign is the most time-consuming and memory-demanding alignment tool. Overall, this makes Quiver the more time-consuming and memory-demanding polishing algorithm than Racon as Quiver can only work with BLASR, a part of pbalign tool. Second, for the polishing tools other than Quiver, the runtimes of aligners do not make any difference to compare the overall runtimes of assembly polishing pipelines (i.e., overall runtime of both an aligner and an assembly polishing tool) to each other as any of these polishing algorithms can use any aligner that the other polishing algorithms can use. We conclude that Quiver is the only algorithm that is affected by the runtime of the aligner that it requires as it cannot use any other aligners.

In Supplementary Table S5, we evaluate whether running assembly polishing algorithms multiple times cause additional runtimes. We observe that we need to generate the index file of an assembly each time it is polished. We suspect that generating index files for large genomes may require additional significant amount of time. To show the additional runtime that an indexing step requires, we use BWA and Bowtie2 to generate the index files for the human reference genome (GRCh38), zebra fish genome (GRCz11), and finished assembly of CHM1. We show that indexing takes at least an hour for large genomes, as we report in Supplementary Table S5. We conclude that using a hybrid set of reads *within a single run* eliminates the additional runtime that is required to generate the index files of assemblies when polishing an assembly multiple times (e.g., running Racon multiple times to use both PacBio and Illumina set of reads) using either the same tool or multiple polishing tools.

3.6 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, 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. However, we believe that these runtimes for Apollo are still reasonable for two reasons. First, Apollo is the *only* algorithm that can scale itself to polish a large genome assembly using moderate coverage (e.g., up to $\sim 35X$) set of reads even though it requires more than a week to polish a large genome assembly. Second, assembly polishing is a one-time task for the assembly that is usually used many times and even made publicly available to the community. Therefore, we believe that long runtimes are still reasonable given that genomic data will probably be used many times after it is generated. In addition, it is possible to accelerate the calculation of the Forward-Backward algorithm and the Viterbi algorithm using Tensor cores, SIMD and GPUs (Murakami, 2017; Eddy, 2011; Liu, 2009; Yu *et al.*, 2014), which we leave as future work.

4 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 is scalable 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. In this paper, we show three key results. First, we show that using a hybrid set of reads results in more accurate assemblies compared to using a single set of reads. Second, Quiver, Racon, and Pilon fail to polish a large genome assembly using a moderate coverage set of reads (i.e., $\sim 22X$ and $\sim 35X$) whereas Apollo can use these reads to polish a large genome. Apollo is the *only* algorithm that can use a moderate coverage set of long and short reads to polish an assembly of a large genome. Third, Apollo polishes assemblies with comparable accuracy to the accuracy of assemblies that state-of-the-art assembly polishing algorithms produce when only a single set of reads are used. 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, achieving high accuracy.

Funding

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

References

Alkan, C., Sajjadian, S., and Eichler, E. E. (2011). Limitations of next-generation genome sequence assembly. *Nat. Methods*, **8**(1), 61–65.

Alser, M., Hassan, H., Xin, H., Ergin, O., Mutlu, O., and Alkan, C. (2017). GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics*, **33**(21), 3355–3363.

Au, K. F., Underwood, J. G., Lee, L., and Wong, W. H. (2012). Improving PacBio Long Read Accuracy by Short Read Alignment. *PLoS One*, **7**(10), 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., Koren, S., Chin, C.-S., Drake, J. P., Landolin, J. M., and Phillippy, A. M. (2015). Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.*, **33**(6), 623–630.

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**(1), 238.

Chaisson, M. J. P., Wilson, R. K., and Eichler, E. E. (2015). Genetic variation and the de novo assembly of human genomes. *Nat. Rev. Genet.*, **16**(11), 627–640.

Chin, C.-S., Alexander, D. H., Marks, P., Klammer, A. A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E. E., Turner, S. W., and Korlach, J. (2013). Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat. Methods*, **10**(6), 563–569.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**(1), 11.

Eddy, S. R. (2011). Accelerated Profile HMM Searches. *PLoS Comput. Biol.*, **7**(10), e1002195.

Firtina, C. and Alkan, C. (2016). On genomic repeats and reproducibility. *Bioinformatics*, **32**(15), 2243–2247.