Subject Section

**Apollo: A Sequencing-Technology-Independent, Scalable, and Accurate Assembly Polishing Algorithm**

Can Firtina¹, Jeremie S. Kim¹,², Mohammed Alser¹, Damla Senol Cali², A. Ercument Cicek³, Can Alkan³, *, and Onur Mutlu¹,²,³, *

¹Department of Computer Science, ETH Zurich, Zurich 8092, Switzerland
²Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh 15213, PA, USA
³Department 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) (Gleim, 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 (Firinha et al., 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 (Firinha et al., 2018), LoDREC (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 only use 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 (MimIO), Escherichia coli O157 (PacBio and Illumina), a human hydatidiform mole CHM1 cell line (PacBio) (Steinberg et al., 2014), and the human Ashkenazi trios 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 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 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
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 see 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 “GCCC” 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 contains insertion states for each base pair 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 updates 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.

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 correct position), or when there is a substitution error (i.e., emitting a different base pair for the correct 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 900kbp) 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 (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 2.2 and 2.3 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 $20X$.

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 Snakemake library (Doring 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 (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 `-v` 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 `pblalign` tool (https://github.com/PacificBiosciences/pblalign) 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 \(b = 50000\), 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 the 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 FDAAARGOS_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 zebrafish (GRCh11) 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 (~35X coverage) and Illumina (~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 phalnig 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., ~22X and ~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 polished 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∗                                    |
| E.coli K12 - Ground Truth |                  | GenBank NC_000913                             |
| E.coli O157 - PacBio      |                  | SRA SRR5431248                               |
| E.coli O157 - Illumina    |                  | GenBank JNE02000001                           |
| Human CHM1 - PacBio       |                  | SRA SRR1304331-5                             |
| Human CHM1 - Ground Truth |                  | GCA_000306695-5                             |
| Human HG002 - PacBio      |                  | SRA SRR20236394-471, SRR20236654-9           |
| Human HG002 - Illumina    |                  | SRA SRR176644-59                            |
| Human HG002 - Ground Truth |                | GCA_001542345-1                             |

∗ We list the data sets we use in our experiments. The data can be accessed through NCBI using the accession number.

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 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.

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 (Vasert 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 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 (~30X) and high coverage (151X) PacBio reads 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 reads (0.99999 with the polishing score of 0.99902). 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 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 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          |
| Human HG002               | PacBio                        | 96.99             | 0.9636   | 0.9346          |
| Human HG002               | Illumina                      | 96.06             | 0.9784   | 0.9396          |
| Human HG002               | 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 Illumina reads, to compare Apollo, Racon, and Nanopolish. We also investigate the performance of Apollo using only the ONT reads (E. coli K-12) and 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
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 (~30X), Apollo is also still in the comparable range as its accuracy only differs by ~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 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 (v), maximum deletion length per transition (d), transition probability to a match state (tm), transition probability to an insertion state (di). 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 in 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 phalgun 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 phalgun 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 TUBITAK [TUBITAK-1001-215E172 to C.A.].

References
Alkan, C., Sajjadjan, S., and Eichler, E. E. (2011). Limitations of next-generation genome sequence assembly. Nat. Methods, 8(1), 61–65.
Aler, M., Hassan, H., Xin, H., Ergin, O., Miuo, 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(1), 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., Alexandre, D. H., Marko, P., Khamra, A. A., Drake, J., Weiner, C., Chum, A., Copeland, A., Hagglin, A., Eichler, E. E., Turner, S. W., and Korf, L. (2013). Nobsybird, finished microbial genome assemblies from long-read SMRT sequencing data. Nat. Methods, 10(6), 563–569.
Ding, A., Wesse, 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), e1001925.
Firtina, C. and Alkan, C. (2016). On genomic repeats and reproducibility. Bioinformatics, 32(15), 2243–2247.
1 Constructing a profile hidden Markov model

A pHMM-graph includes states and directed transitions from a state to another. There are two type of probabilities that the graph contains: (1) emission and (2) transition probabilities. First, each state has emission probabilities for emitting the certain characters where each is associated with a probability in range $[0, 1]$. Second, each transition is associated with a probability in range $[0, 1]$. Now, we will explain the structure of the graph in detail. For an assembly contig $C$, let us define the pHMM-graph that represents the contig $C$ as $G(V, E)$. Let us also define the length of the contig $C$ as $n = |C|$. A basepair $C[t]$ has one of the letters in the alphabet set $\Sigma = \{A, C, G, T\}$. Thus, a state emits one of the characters in $\Sigma$ with a certain probability. For a state $i$, We denote the emission probability of a basepair $c \in \Sigma$ as $e_i(c) \in [0, 1]$ where $\sum_{c \in \Sigma} e_i(c) = 1$. We denote the transition probability from a state, $i$, to another state, $j$, as $\alpha_{ij} \in [0, 1]$. For the set of the states that the state $i$ has an outgoing transition to, $V_i$, we have $\sum_{j \in V_i} \alpha_{ij} = 1$. Now let us define in four steps how Apollo constructs the states and the transitions of the graph $G(V, E)$:

First, Apollo constructs a start state, $v_{\text{start}} \in V$, and an end state $v_{\text{end}} \in V$. Second, for each basepair $C[t]$ where $1 \leq t \leq n$, Apollo constructs a match state as follows (Figure S1):

- A match state that we denote as $M_t$ for the basepair $C[t]$ where $M = C[t]$ s.t. $C[t] \in \Sigma$ and $M_t \in V$ (i.e., if the $t$th basepair of the contig $C$ is $G$, then the corresponding match state is $G_t$). For the following steps, let us assume $i = M_t$
- A match emission with the probability $\beta$, for the basepair $C[t]$ s.t. $e_i(C[t]) = \beta$. $\beta$ is a parameter to Apollo.
- A substitution emission with the probability $\delta$, for each basepair $c \in \Sigma$ and $c \neq C[t]$ s.t. $e_i(c) = \delta$ (Note that $\delta + 3\delta = 1$). $\delta$ is a parameter to Apollo.
- A match transition with the probability $\alpha_M$, from the match state $M_t = i$ to the next match state $M_{t+1} = j$ s.t. $\alpha_{ij} = \alpha_M$. $\alpha_M$ is a parameter to Apollo.

Third, for each basepair $C[t]$ where $1 \leq t \leq n$, Apollo constructs the insertion states as follows (Figure S2):

- There are $l$ many insertion states, $I^i_1, I^i_2, \ldots, I^i_l$, where $I^i_t \in V$, $1 \leq i \leq l$ and $l$ is a parameter to Apollo.
- The match state, $M_t = i$, has an insertion transition to $I^i_1 = j$, with the probability $\alpha_I$ s.t. $\alpha_{ij} = \alpha_I$
- For each $i$ where $1 \leq i < l$, the insertion state $I^i_k = j$ has an insertion transition to the next insertion state $I^i_{k+1} = j$ with the probability $\alpha_I$ s.t. $\alpha_{kj} = \alpha_I$
- For each $i$ where $1 \leq i < l$, the insertion state $I^i_k = j$ has a match transition to the match state of the next basepair $M_{t+1} = j$ with the probability $\alpha_M$ s.t. $\alpha_{kj} = \alpha_M$
- The last insertion state, $I^i_l$, has no further insertion transitions. Instead, it has a transition to the match state of the next basepair $M_{t+1} = j$ with the probability $\alpha_M + \alpha_I$ s.t. $\alpha_{kj} = \alpha_M + \alpha_I$
Figure S1: Two match states. Here, the contig includes the basepairs $G$ and $A$ at the locations $t$ and $t + 1$, respectively. The corresponding match states are labeled with the basepairs that they correspond to (i.e., the match state $G_t$ represents the basepair $G$ at the location $t$). Each match state has a match transition to the next match state with the initial probability $\alpha_M$. A match state has a match emission probability, $\beta$, for the basepair it is labeled with. The remaining three basepairs have equal substitution emission probability $\delta$. The figure is taken from Hercules [1].

- For each $i$ where $1 \leq i \leq l$, each basepair $c \in \Sigma$ and $c \neq C[t + 1]$ has an insertion emission probability $1/3 \approx 0.33$ for the insertion state $I_i = k$ s.t. $e_k(c) = 0.33$ and $e_k(C[t + 1]) = 0$. Note that $\sum_{c \in \Sigma} e_k(c) = 1$. (i.e., if the basepair at the location $t + 1$ is T, then $e_k(A) = 0.33$, $e_k(T) = 0$, $e_k(G) = 0.33$, and $e_k(C) = 0.33$).

Fourth step for finalizing the complete structure of the pHMM graph, for each state $i \in V$, Apollo constructs the deletion transitions as follows (Figure S3):

- Let us define $\alpha_{del} = 1 - (\alpha_M - \alpha_I)$, which is the overall deletion transition probability.
- There are $k$ many deletion transitions from the state $i$, to the further match states. $k$ is a parameter to Apollo.
- We assume that a transition deletes the basepairs if it skips the corresponding match states of the basepairs. We denote the transition probability of a deletion transition as $\alpha_D^k$ s.t. $1 \leq x \leq k$, if it deletes $x$ many basepairs in a row in one transition. Apollo calculates the deletion transition probability $\alpha_D^k$ using the normalized version of a polynomial distribution where $f \in [0, \infty)$ is a factor value for the equation:

$$\alpha_D^k = \frac{f^{k-x} \alpha_{del}}{\sum_{j=0}^{k-1} f^j} \quad 1 \leq x \leq k \tag{S1}$$

- If the $f$ value is set to 1, then the each deletion transition is equally likely (i.e., $\alpha_D^1 = \alpha_{del}^{10}$ if $k \geq 10$). As the $f$ value increases, the probability of deleting more basepairs in one transition decreases accordingly (i.e., $\alpha_D^1 \gg \alpha_{del}^{10}$ if $k \geq 10$). $f$ is a parameter to Apollo.

We note that the start state $v_{start}$ also has a match transition to $M_1$ and deletion transitions as defined previously. There are $l$ many insertion states, $I_0, I_1, \ldots, I_{l_0}$, between the start state and the first match state $M_1$. The transitions of these insertion states are also identical to what we described before. We would also like to note that the end state $v_{end}$ has no outgoing transition. The prior states consider $v_{end}$ as a match state and connect to it accordingly. The start and end states have no emission probabilities.

2 The Forward-Backward Algorithm

Apollo trains the pHMM of the contig per each read that aligns to the contig. It uses the alignment location and the sequence of the read in order to train the pHMM-graph. First, per each aligned read sequence $r$, Apollo extracts the sub-graph $G_s(V_s, E_s)$ that corresponds to the aligned region of the contig where we have $v_{start}, v_{end}$, match and insertion states, and the transitions as described in the Supplementary Section 1. Each transition from state $i \in V_s$ to state $j \in V_s$, $E_{ij} \in E_s$, is associated with a transition probability $\alpha_{ij}$. For every pair of states, $i \in V_s$ and $j \in V_s$, the transition probability $\alpha_{ij} = 0$.
Figure S2: \( l \) many insertion states for the basepair at location \( t \). Here, the contig includes the basepairs \( C \) and \( T \) at the locations \( t \) and \( t + 1 \), respectively. The corresponding match states are labeled with the basepairs that they correspond to. Each insertion state has an insertion transition to the next insertion state with the initial probability \( \alpha_I \) and a match transition to the next match state at the location \( t + 1 \) with the initial probability \( \alpha_M \). However, the last insertion state, \( I_{lt} \), does not have a transition to the next insertion state as it is the last one. Instead, it has a match transition to the next match state \( T_{t+1} \) with the probability \( \alpha_M + \alpha_I \). The emission probability of the basepair \( T \) is 0 as it appears in the next position (\( t + 1 \)) of the contig. The figure is taken from Hercules [1].

Figure S3: Deletion transitions of the match and each insertion states at location \( t \). For the match and insertion states at location \( t \), we show only the deletion transitions (red). Note that a deletion transition from the position \( t \) to the match state of the position \( t + x + 1 \) removes \( x \) many basepairs with the probability \( \alpha^x_D \) as it skips \( x \) many match states where \( 1 \leq x \leq k \). The figure is taken from Hercules [1].
if $E_{ij} \not\in E_s$. Let us define the length of the aligned read, $r$, as $m = |r|$. Second, it calculates the forward and backward probabilities of each state based on the aligned read, $r$.

Let us assume that the forward probability of a state $j$ that observes $t^{th}$ basepair of the aligned read, $r[t]$, is $F_t(j)$. For the forward probability, observing the $t^{th}$ basepair at the state $j$ means that all the previous basepairs ($r[1] \ldots r[t-1]$ and $1 < t \leq m$) have been observed by following a path starting from the start state to the state $j$ and $j$ observes the next basepair, $r[t]$. All possible transitions that lead to state $j$ to observe the basepair $r[t]$ contribute to the probability with (1) the forward probability of the origin state $i$ calculated with the $(t-1)^{th}$ basepair of $r$, $F_{t-1}(i)$, (2) multiplied by the probability of the transition from $i$ to $j$, $\alpha_{ij}$, (3) multiplied by the probability of emitting the basepair $r[t]$ at state $j$, $e_j(r[t])$.

Let us denote the start state $v_{\text{start}}$ with the index value of 0 (i.e., $v_{\text{start}} = 0$). For each state $j \in V_s$, we calculate the forward probability, $F_t(j)$, as follows where $F_1(j)$ is the initialization step:

\[
F_1(j) = \alpha_{0j}e_j(r[1]) \quad \text{s.t.} \quad j \in V_s, \quad E_{0j} \in E_s
\] (S2.1)

\[
F_t(j) = \sum_{i \in V_s} F_{t-1}(i)\alpha_{ij}e_j(r[t]) \quad j \in V_s, \quad 1 < t \leq m
\] (S2.2)

Let us assume that the backward probability of a state $i$ that observes $t^{th}$ basepair of the aligned read, $r[t]$, is $B_t(i)$. For the backward probability, observing the $t^{th}$ basepair at the state $i$ means that all the further basepairs ($r[t+1] \ldots r[m]$ and $1 \leq t < m$) have been observed by following a path starting from the end state to the state $i$ (backwards) and $i$ observes the previous basepair, $r[t]$. All possible transitions that lead to state $i$ to observe the basepair $r[t]$ contribute to the probability with (1) the backward probability of the next state $j$ calculated with the $(t+1)^{th}$ basepair of $r$, $B_{t+1}(j)$, (2) multiplied by the probability of the transition from $i$ to $j$, $\alpha_{ij}$, (3) multiplied by the probability of emitting the basepair $r[t+1]$ at state $j$, $e_j(r[t+1])$.

Let us denote the end state $v_{\text{end}}$ with the index value of $m + 1$ (i.e., $v_{\text{end}} = m + 1$). For each state $j \in V_s$, we calculate the backward probability, $B_t(i)$, as follows where $B_m(i)$ is the initialization step:

\[
B_m(i) = \alpha_{ij(m+1)} \quad i \in V_s, \quad E_{i(m+1)} \in E_s
\] (S3.1)

\[
B_t(i) = \sum_{j \in V_s} \alpha_{ij}e_j(r[t+1])B_{t+1}(j) \quad j \in V_s, \quad 1 \leq t < m
\] (S3.2)

After calculation of the forward and backward probabilities, Apollo calculates the posterior transition and the emission probabilities of the sub-graph, $G_s$, as shown in equations S4 and S5, respectively.

\[
e_t^*(X) = \frac{\sum_{i=1}^{m} F_t(i)B_t(i)(r[t] \underline{\quad} X)}{\sum_{i=1}^{m} F_t(i)B_t(i)} \quad \forall X \in \Sigma, \forall i \in V_s
\] (S4)

\[
\alpha_{ij}^* = \frac{\sum_{i=1}^{m-1} \sum_{j \in V_s} \alpha_{ij}e_j(r[t+1])F_t(i)B_{t+1}(j)}{\sum_{i=1}^{m-1} \sum_{j \in V_s} \alpha_{ij}e_j(r[t+1])F_t(i)B_{t+1}(x)} \quad \forall E_{ij} \in E_s
\] (S5)

3 Joining Posterior Probabilities

As we explain in the Supplementary Section 2, for each read that aligns to the contig, Apollo extracts a sub-graph $G_s$ and uses the Forward-Backward algorithm to train the sub-graph. It is possible that there can be overlaps between two or many sub-graphs such that the sub-graphs can include the same states and the transitions. However, the updates on the overlapping states and the transitions are exclusive between the sub-graphs such that no two update in separate graphs affect each other while calculating the Forward or the Backward probabilities. Each sub-graph uses the initial probabilities to calculate the posterior probabilities. In order to handle training of the overlapping states and the transitions, Apollo takes the average of the posterior probabilities and reports the average probability as the final posterior probability for the entire pHMM-graph.
Let us assume that the set of sub-graphs $S$ includes the same state $i \in V$. For each $G_s$ in $S$, we obtain a $e^*_i(X)$, where $\forall X \in \Sigma$, which denotes the posterior emission probability as we explain in the Supplementary Section 2. We denote $e^*_i(X)$ that belongs to $G_s$ as $e^*_i(G_s)(X)$. Then, Apollo finds the final emission value $\hat{e}_i(X)$ as follows:

$$\hat{e}_i(X) = \frac{\sum_{G_s \in S} e^*_i(G_s)(X)}{|S|} \quad \forall X \in \Sigma$$  \hfill (S6)

Similarly, let us assume that the set of sub-graphs $S$ includes the same transition edge $E_{ij} \in E$. For each $G_s$ in $S$, we obtain an $\alpha^*_ij$ that denotes the posterior transition value. We define $\alpha^*_ij$ that belongs to $G_s$ as $\alpha^*_ij(G_s)$. Apollo finds the final transition value $\hat{\alpha}_{ij}$ as follows:

$$\hat{\alpha}_{ij} = \frac{\sum_{G_s \in S} \alpha^*_ij(G_s)}{|S|}$$  \hfill (S7)

If a state in $V$ or an edge in $E$ is not covered by a read then Apollo retains the initial emission and transition probabilities and uses as posterior probabilities, respectively.

### 4 Decoding with the Viterbi Algorithm

Apollo uses the Viterbi algorithm to reveal the polished assembly by finding the most likely path starting from the start state $v_{\text{start}}$ of the graph $G$ to the end state $v_{\text{end}}$. For each state $j$, the Viterbi algorithm calculates $v_t(j)$, which is the maximum marginal forward probability $j$ obtained from following a path starting from the start state when emitting the $t$th basepair of the polished contig. It also keeps a back pointer, $b_t(j)$, which keeps track of the predecessor state $i$ that yields the $v_t(j)$ value.

Let $X' \in \Sigma$ be the basepair that has the highest posterior emission probability for the state $j$ and $T$ be the length of the decoded sequence, which is initially unknown. The algorithm recursively calculates $v$ values for each position $t$ of a decoded sequence as described in the equations S8.1 and S8.3. The algorithm stops at iteration $T^*$ such that for the last $iter$ iterations, the maximum value we have observed for $v(end)$ cannot be improved and $iter$ is a parameter and set to 100 by default. $T$ is then set to $t^*$ such that $v_{t^*}(end)$ is the maximum among all iterations $1 \leq t \leq T^*$.

1. **Initialization**

   $$v_1(j) = \hat{\alpha}_{\text{start} \rightarrow j} \hat{e}_j(X') \quad \forall j \in V$$  \hfill (S8.1)

   $$b_1(j) = \text{start} \quad \forall j \in V$$  \hfill (S8.2)

2. **Recursion**

   $$v_t(j) = \max_{i \in V} v_{t-1}(i) \hat{\alpha}_{ij} \hat{e}_j(X') \quad \forall j \in V, 1 < t \leq T$$  \hfill (S8.3)

   $$b_t(j) = \arg\max_{i \in V} v_{t-1}(i) \hat{\alpha}_{ij} \hat{e}_j(X') \quad \forall j \in V, 1 < t \leq T$$  \hfill (S8.4)

3. **Termination**

   $$v_T(end) = \max_{i \in V} v_T(i) \hat{\alpha}_{i \rightarrow \text{end}}$$  \hfill (S8.5)

   $$b_T(end) = \arg\max_{i \in V} v_T(i) \hat{\alpha}_{i \rightarrow \text{end}}$$  \hfill (S8.6)
5 Performance of the Assembly Polishing Algorithms

In Tables S1, S2, and S3 we compare the assembly polishing performance of Apollo to the competing algorithms.

### Table S1: Assembly polishing performance of the tools for E.Coli K-12 data set

| Sequencing Tech. of the Assembly | Assembler | Aligner | Sequencing Tech. of the Reads | Polishing Algorithm | Aligned Bases (%) | Accuracy | Polishing Score |
|----------------------------------|-----------|---------|--------------------------------|---------------------|-------------------|----------|-----------------|
| OTT                              | Miniasm   | Minimap2| OTT                            | Apollo              | 97.24             | 0.9228   | 0.8973          |
| OTT                              | Miniasm   | Minimap2| OTT                            | Nanopolish          | 87.35             | 0.8512   | 0.7435          |
| OTT                              | Miniasm   | Minimap2| OTT                            | Racon               | 99.22             | 0.9766   | 0.9680          |
| OTT                              | Cann      | Minimap2| OTT                            | Apollo              | 99.99             | 0.9803   | 0.9802          |
| OTT                              | Cann      | Minimap2| OTT                            | Nanopolish          | 99.98             | 0.9925   | 0.9923          |
| OTT                              | Cann      | Minimap2| OTT                            | Racon               | 100.00            | 0.9840   | 0.9840          |
| OTT (30X)                        | Miniasm   | -       | -                              | -                   | 99.98             | 0.9794   | 0.9792          |
| OTT (30X)                        | Cann      | -       | -                              | -                   | 99.98             | 0.9752   | 0.9750          |
| OTT (30X)                        | Cann      | Minimap2| OTT (30X)                      | Apollo              | 99.99             | 0.9857   | 0.9856          |
| OTT (30X)                        | Cann      | Minimap2| OTT (30X)                      | Nanopolish          | 99.99             | 0.9825   | 0.9825          |
| OTT (30X)                        | Cann      | Minimap2| OTT (30X)                      | Racon               | 100.00            | 0.9825   | 0.9825          |
| OTT (30X)                        | Cann      | Minimap2| OTT (30X, Corr)                | Apollo              | 99.96             | 0.9755   | 0.9754          |
| OTT (30X)                        | Cann      | Minimap2| OTT (30X, Corr)                | Racon               | 100.00            | 0.9799   | 0.9799          |

We generate the assembly for E.Coli K-12 data set using the reads sequenced from Oxford Nanopore Technologies (ONT) (319X coverage) as specified in **Sequencing Tech. of the Assembly**. We subsample ONT reads into 30X coverage and generate the assembly using the sub-sampled reads that we show as ONT (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.
Table S2: Assembly polishing performance of the tools for CHM1 (Homo Sapiens) data set

| Sequencing Tech. of the Assembly | Assembler | Aligner | Sequencing Tech. of the Reads | Polishing Algorithm | Aligned Bases (%) | Accuracy | Polishing Score | Runtime | Memory |
|---------------------------------|-----------|---------|-------------------------------|---------------------|-------------------|----------|----------------|---------|--------|
| PacBio                          | Miniasm   | -       | -                            | -                   | 71.62             | 0.8108   | 0.5807         | 2h 47m 16s | 68.23 |
| PacBio                          | Miniasm   | Minimap2| PacBio                       | Apollo              | 81.15             | 0.8236   | 0.6684         | 1h 50m 22s | 4.90  |
| PacBio                          | Miniasm   | Minimap2| PacBio                       | Racon               | 83.02             | 0.8381   | 0.6958         | 17m 35s   | 13.15 |
| PacBio                          | Canu      | pbalign | PacBio                       | Quiver              | 81.12             | 0.8108   | 0.6742         | 1m 26s    | 9.52  |
| PacBio                          | Canu      | -       | -                            | -                   | 92.27             | 0.8442   | 0.5311         | 1h 25m 16s | 4.50  |
| PacBio                          | Canu      | Minimap2| PacBio                       | Apollo              | 93.11             | 0.8236   | 0.6684         | 23h 59m 12s | 12.99 |
| PacBio                          | Canu      | Minimap2| PacBio (Cor.)                | Racon               | 93.11             | 0.8902   | 0.8173         | 3h 10m 05s | 4.50  |
| PacBio                          | Canu      | pbalign | PacBio                       | Quiver              | 89.48             | 0.8925   | 0.7266         | 41s      | 0.51  |

We generate the assembly for CHM1 data set using the reads sequenced from PacBio (2.6X coverage) as specified in Sequencing Tech. of the Assembly. 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.

Table S3: Assembly polishing performance of the tools for the Ashkenazim trio (HG002, Son) data set

| Aligner | Sequencing Tech. of the Reads | Polishing Algorithm | Runtime | Memory |
|---------|-------------------------------|---------------------|---------|--------|
| Minimap2| PacBio (35X)                  | Apollo              | 227h 12m 15s | 62.91  |
| BWA-MEM | PacBio (35X)                  | Apollo              | 198h 41m 15s | 58.60  |
| Minimap2| PacBio (35X)                  | Racon               | N/A     | N/A    |
| BWA-MEM | PacBio (35X)                  | Racon               | N/A     | N/A    |
| pbalign | PacBio (35X)                  | Quiver              | N/A     | N/A    |
| Minimap2| PacBio (8.9X)                 | Apollo              | 55h 38m 44s | 44.99  |
| BWA-MEM | PacBio (8.9X)                 | Apollo              | 41h 38m 27s | 45.00  |
| Minimap2| PacBio (8.9X)                 | Racon               | 2h 48m 25s | 54.13  |
| BWA-MEM | PacBio (8.9X)                 | Racon               | 1h 36m 39s | 51.55  |
| pbalign | PacBio (8.9X)                 | Quiver              | N/A     | N/A    |
| Minimap2| Illumina (22X)                | Apollo              | 96h 22m 16s | 101.12 |
| BWA-MEM | Illumina (22X)                | Apollo              | 102h 01m 57s | 107.06 |
| Minimap2| Illumina (22X)                | Racon               | N/A     | N/A    |
| BWA-MEM | Illumina (22X)                | Racon               | N/A     | N/A    |
| Minimap2| Illumina (22X)                | Pilon               | N/A     | N/A    |
| BWA-MEM | Illumina (22X)                | Pilon               | N/A     | N/A    |

We use the Polishing Algorithms to polish the finished assembly of the Ashkenazim trio (HG002, Son) sample. Here, the reads from the Ashkenazim trio sample and that is specified under Sequencing Tech. of the Reads are sequenced by the specified sequencing technology and are aligned to the assembly using the Aligner. We report the runtime and the memory requirements of the assembly polishing tools. We report the Runtime and the Memory as N/A, if a Polishing Algorithm fails to polish the assembly.
6 Performance of the Aligners

Here in Table S4, we show the performances of the aligners in terms of number of alignments that the aligners generate given the assembly and the reads to align, runtime (wall clock), and the memory requirement. In Table S5, we show the the running times that BWA and Bowtie2 require to construct the indexing files for the human reference genome (GRCh38), the zebra fish reference genome (GRCz11), and the complete assembly of the human CHM1 cell line.

| Data Set for the Assembly | Assembler | Aligner | Platform of the Aligned Reads | Number of Alignments | Runtime | Memory (GB) |
|---------------------------|-----------|---------|-------------------------------|----------------------|---------|-------------|
| E.Coli K-12 - ONT         | Miniasm   | Minimap2| ONT                           | 8,095,856            | 3m 30s  | 4.88        |
| E.Coli K-12 - ONT         | Canu      | Minimap2| ONT                           | 1,662,306            | 39s     | 2.10        |
| E.Coli K-12 - ONT (30X)   | Canu      | Minimap2| ONT (30X)                     | 170,910              | 6s      | 0.60        |
| E.Coli O157 - PacBio      | Miniasm   | Minimap2| PacBio                        | 732,397              | 25s     | 1.79        |
| E.Coli O157 - PacBio      | Canu      | Minimap2| Illumina                      | 21,933,051           | 1m 35s  | 3.16        |
| E.Coli O157 - PacBio (30X)| Canu      | Minimap2| PacBio (30X)                  | 741,343              | 22s     | 1.80        |
| E.Coli O157 - PacBio (30X)| Canu      | Minimap2| PacBio (30X, Corr)            | 148,241              | 5s      | 0.67        |
| E.Coli O157 - PacBio      | Miniasm   | BWA-MEM | Illumina                      | 137,620              | 3s      | 0.47        |
| E.Coli O157 - PacBio      | Canu      | BWA-MEM | Illumina                      | 19,799,002           | 2m 34s  | 3.17        |
| E.Coli O157 - PacBio      | Canu      | BWA-MEM | Illumina                      | 23,328,379           | 1m 16s  | 2.89        |
| E.Coli O157 - PacBio      | Canu      | BWA-MEM | Illumina                      | 23,326,292           | 1m 20s  | 2.96        |
| E.Coli O157 - PacBio      | Miniasm   | pbalign | PacBio                        | 49,561               | 12m 55s | 6.36        |
| E.Coli O157 - PacBio      | Canu      | pbalign | PacBio                        | 51,994               | 11m 29s | 6.28        |
| CHM1 - PacBio             | Miniasm   | Minimap2| PacBio                        | 3,209,313            | 20m 24s | 1.62        |
| CHM1 - PacBio             | Canu      | Minimap2| PacBio                        | 3,309,401            | 8m 11s  | 1.63        |

We generate the assembly using the reads specified under Data Set for the Assembly. We use Canu [2] and Miniasm [3] assemblers as specified in Assembler. Here, the reads specified under Platform of the Aligned Reads are aligned to the assembly using the Aligner. We use Minimap2 [4] aligner for aligning both long and short reads to the assembly and BWA-MEM [5] aligner to align the short reads to the assembly. We report the performance of the aligners in terms of the number of the aligners (Number of Alignments), the runtime (Runtime), and the maximum memory requirement Memory.

| Tool | GRCh38     | GRCz11     | CHM1       |
|------|------------|------------|------------|
| BWA | 1h 13m 35s | 1h 01m 17s | 1h 16m 35s |
| Bowtie2 | 3h 18m 06s | 2h 30m 22s | 3h 46m 12s |

We show the running times required to generate the index files of the references/assembleys specified in each column using indexing tools of BWA and Bowtie2.
7 Robustness of Apollo

Here in Tables S6, S7, S8, S9, we show the robustness of Apollo based on the parameters that has a direct affect on the machine learning algorithm. In each of the tables we show that Apollo is robust to different set of parameters.

Table S6: Apollo’s robustness based on the chunk size of the long read and the contig

| Long Read Chunk Size | Contig Chunk Size | Aligned Bases | Aligned Bases (%) | Accuracy |
|----------------------|-------------------|---------------|-------------------|----------|
| 1000                 | Original          | 5,708,747     | 98.49             | 0.9798   |
| 1000                 | 25000             | 5,487,736     | 94.46             | 0.9733   |
| 1000                 | 50000             | 5,689,120     | 97.95             | 0.9728   |
| 1000                 | 100000            | 5,493,663     | 94.52             | 0.9727   |
| 5000                 | 25000             | 5,430,700     | 93.06             | 0.8974   |
| 5000                 | 50000             | 5,411,163     | 92.68             | 0.8971   |
| 5000                 | 100000            | 5,516,599     | 94.49             | 0.8970   |
| 10000                | 25000             | 5,415,333     | 92.65             | 0.8918   |
| 10000                | 50000             | 5,423,340     | 92.75             | 0.8914   |
| 10000                | 100000            | 5,474,159     | 93.61             | 0.8914   |

Here we divide the long reads and the assembly into smaller chunks. We use E.coli O157 data set where Miniasm generates the assembly. We divide long reads into smaller reads with lengths 1000, 5000, and 10000. Similarly, we divide the assembly contigs into smaller contigs with lengths 25000, 50000, and 100000. We align each chunked read to each chunked contig. We report the performance of Apollo given the chunked assembly and chunked reads.

Table S7: Apollo’s robustness based on the maximum deletion and filter size parameters

| Max Deletion (-d) | Filter Size (-f) | Aligned Bases | Aligned Bases (%) | Accuracy |
|-------------------|------------------|---------------|-------------------|----------|
| 3                 | 100              | 5,699,182     | 97.91             | 0.9739   |
| 5                 | 100              | 5,696,138     | 97.93             | 0.9735   |
| 15                | 100              | 5,678,838     | 97.90             | 0.9731   |
| 3                 | 200              | 5,705,130     | 98.12             | 0.9751   |
| 5                 | 200              | 5,704,582     | 98.12             | 0.9750   |
| 15                | 200              | 5,702,478     | 98.14             | 0.9751   |

Performance of Apollo with respect to the parameter that defines the maximum number of deletion in one transition \( (d = 3, d = 5, d = 15) \). We also adjust the filter size \( (f = 100, f = 200) \)
Table S8: Apollo’s robustness based on the maximum insertion and filter size parameters

| Max Insertion (-s) | Filter Size (-f) | Aligned Bases | Aligned Bases (%) | Accuracy |
|-------------------|-----------------|---------------|-------------------|----------|
| 1                 | 100             | 5,685,635     | 97.89             | 0.9660   |
| 5                 | 100             | 5,638,585     | 97.62             | 0.9696   |
| 10                | 100             | 5,365,978     | 95.54             | 0.9531   |
| 1                 | 200             | 5,685,040     | 98.02             | 0.9668   |
| 5                 | 200             | 5,692,813     | 98.07             | 0.9740   |
| 10                | 200             | 5,623,736     | 97.62             | 0.9701   |

Performance of Apollo with respect to the parameter that defines the maximum number of insertion states for each basepair ($i = 1, i = 5, i = 10$). We also adjust the filter size ($f = 100, f = 200$)

Table S9: Apollo’s robustness based on the match transition, insertion transition probabilities, and the filter size parameters

| Match Transition Probability (-tm) | Insertion Transition Probability (-ti) | Filter Size (-f) | Aligned Bases | Aligned Bases (%) | Accuracy |
|-----------------------------------|---------------------------------------|-----------------|---------------|-------------------|----------|
| 0.60                              | 0.25                                  | 100             | 5,670,852     | 97.95             | 0.9625   |
| 0.60                              | 0.30                                  | 100             | 5,660,957     | 97.90             | 0.9596   |
| 0.80                              | 0.10                                  | 100             | 5,699,660     | 98.02             | 0.9788   |
| 0.90                              | 0.05                                  | 100             | 5,685,770     | 97.89             | 0.9774   |
| 0.60                              | 0.25                                  | 200             | 5,682,512     | 98.10             | 0.9644   |
| 0.60                              | 0.30                                  | 200             | 5,681,993     | 98.13             | 0.9618   |
| 0.80                              | 0.10                                  | 200             | 5,707,293     | 98.16             | 0.9803   |
| 0.90                              | 0.05                                  | 200             | 5,695,902     | 98.05             | 0.9789   |

Performance of Apollo with respect to the parameters that define the match and insertion transition probabilities ($tm = 60 - ti = 0.25, tm = 60 - ti = 0.30, tm = 80 - ti = 0.10, tm = 90 - ti = 0.05$). We also adjust the filter size ($f = 100, f = 200$)
8 Parameters

We show the parameter settings of the aligners that we used to align the reads to the assembly in Table S10.

Table S10: List of the parameters that are used to align the reads to the assemblies

| Aligner                              | Parameters                                      |
|--------------------------------------|-------------------------------------------------|
| BWA-MEM                              | -t 45                                           |
| Minimap2 (for PacBio)                | -x map-pb -a -k15 -w5 -t 45 -p 0.6              |
| Minimap2 (for ONT)                   | -x map-ont -a -k15 -w5 -t 45 -p 0.6             |
| Minimap2 (for Illumina)              | -a -k15 -w3 –sr –frag=yes –heap-sort=yes -t 45 -p 0.6 |
| pbalign                              | –nproc 45                                       |
References

[1] Can Firtina, Ziv Bar-Joseph, Can Alkan, and A Ercument Cicek. Hercules: a profile HMM-based hybrid error correction algorithm for long reads. *Nucleic Acids Res.*, page gky724, aug 2018.

[2] Sergey Koren, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Res.*, 27(5):722–736, may 2017.

[3] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, jul 2016.

[4] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, page bty191, may 2018.

[5] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, jul 2009.