 doi:10.20944/preprints202006.0324.v1

Methods for *De-novo* Genome Assembly

Arash Bayat*[1,3], Hasindu Gamaarachchi[1], Nandan P Deshpande[2], Marc R Wilkins[2], and Sri Parameswaran[1]

[1]School of Computer Science and Engineering, UNSW, Australia
[2]Systems Biology Initiative, School of Biotechnology and Biomolecular Sciences, UNSW, Australia
[3]Health and Biosecurity, CSIRO, Australia

June 25, 2020

Abstract

Despite advances in algorithms and computational platforms, *de-novo* genome assembly remains a challenging process. Due to the constant innovation in sequencing technologies (*Sanger*, *SOLiD*, *Illumina*, *454*, *PacBio* and *Oxford Nanopore*), genome assembly has evolved to respond to the changes in input data type. This paper includes a broad and comparative review of the most recent *short-read*, *long-read* and hybrid assembly techniques. In this review, we provide (1) an algorithmic description of the important processes in the workflow that introduces fundamental concepts and improvements; (2) a review of existing software that explains possible options for genome assembly; and (3) a comparison of the accuracy and the performance of existing methods executed on the same computer using the same processing capabilities and using the same set of real and synthetic datasets. Such evaluation allows a fair and precise comparison of accuracy in all aspects. As a result, this paper identifies both the strengths and weaknesses of each method. This comparative review is unique in providing a detailed comparison of a broad spectrum of cutting-edge algorithms and methods.

Availability: `https://arashbayat.github.io/asm`

*To whom correspondence should be addressed. Email: arash.bayat@csiro.au

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

1 Introduction

DNA is a giant molecular strand that is a chain of four small molecules. Sequencing is the process of reading *DNA* molecular chain into strings of `A`, `C`, `T`and `G` where each letter (called a *base*) represents one of the small molecules. Due to the available technology today, the *DNA* strand is broken into small pieces and each fragment is then sequenced separately. However, the order of *DNA* fragments (*reads*) cannot be preserved. Therefore, a process called *genome assembly* is used. This is the process by which the entire genome of an individual or species is obtained. The word *assembly* indicates that the entire genome cannot be obtained at once; rather, it must be assembled from small *DNA* fragments. The process of *de-novo* assembly has to do with assembling a species' genome for the first time. The resulting genome can then be used to facilitate genome assembly of other individuals of the same species, in a process called the *reference-guided* assembly. (To understand this, one might think of solving a jigsaw puzzle by using its cover photo for reference.) In contrast, *de-novo* assembly refers to assembling the genome without having a draft genome; this is a complex and difficult problem to solve.

Based on the purpose of the assembly, there are several expectations about the accuracy of the assembled genome and the time it takes to be assembled. Furthermore, the type and volume of sequenced data play important roles in the assembly process. It is critical to choose a pipeline that best fits the available data and fulfils expectations. Only a fair comparison of assembly pipelines can be informative as it would truly reveal the strengths and weaknesses of various methods.

The assembled genome is expected to be accurate; various metrics can be used to measure the accuracy of an assembled genome [16]. The most important metrics include genome coverage and contiguity as well as *base* calling error rate. It is not possible to find a pipeline that maximizes the accuracy of all aspects [9]. The pipeline should be chosen based on the application for which the genome is assembled.

If the assembled genome is going to be used as a *reference-genome* in a *reference-guided* assembly, high genome coverage is prioritized. If part of the genome is missed in the *reference-genome*, all subsequent assembled genomes will lack the information on that region. However, if a slight *base* calling error exists in the *reference-genome*, a *variant-calling* process shows the same variation in all individuals; this variation provides evidence of the error in the *reference-genome*. This can be fixed by updating the *reference-genome* for further analysis. On the other hand, if the genome is assembled for functional genomic studies, high *base* calling accuracy in gene areas is critical since an incorrect *base* calling could change the predicted protein structure. High genome coverage and contiguity in all regions may not be a benefit in this scenario.

There are expectations from the assembly pipeline as well. The program should trade-off between speed and accuracy. Despite algorithmic enhancements, speed is usually obtained by applying approximations. This leads to a less accurate assembled genome, which might, for example, consider only exact-match overlaps instead of all overlaps between *reads* [43]. Scalability is another expectation from an assembly pipeline [24]. Genome assembly requires a massive amount of processing, and a pipeline should be able to utilize all available hardware resources such as processors and memory with the highest efficiency. The program should also be able to deal with limitations. For instance, using a large amount of memory is common in *de-novo* assembly programs. A scalable program should be able to deal with the limited amount of available memory, with minimal impact on execution time [28].

In addition to the above expectations, the choice of an assembly pipeline should be based on the following 'data-specific' parameters.

- *The types of data that are available for assembly*: It is important to compare *short-read* and *long-read* assembly approaches to each other and to hybrid assembly techniques.
  - Ultra-short *read* sequencing such as early *SOLiD* [38] sequencing is suitable for *de-Bruijn* graph assembly but not an *Overlap-Layout-Consensus* (*OLC*) assembly approach.
  - *Next-Generation Sequencing* (*NGS*) [31] machines such as *Illumina* [4] produce *reads* up to hundreds of *bases* (*short-reads*) with high accuracy. The error rate is less than 2%; most errors are of the substitution type and are less frequently short *Indels*.
  - *paired-end* [23] and *mate-pair* [50] *reads* are sequenced from two ends of a long DNA fragment where the distance between them is approximately known. Such information can be used to improve the quality of the assembled genome.
  - *PacBio* [14] and *Oxford Nanopore* [20] are *Single-Molecule Real-Time* (*SMRT*) technologies that can sequence *reads* up to many thousands of *bases* long. However, they suffer from a high *base* calling error rate (up to 30%) and can include long *Indels*.
- *Sequencing remains an expensive process*: It is important to understand how different software programs respond to low and high *read* coverage depth when assembling a genome.
  - If only *short-read* or *long-read* are provided, the coverage depth should be considered in configuring the assembly pipeline. For high coverage data, more filtering can be applied to collect high-confidence data for more accurate assembly. On the other hand, when the coverage is low, fewer heuristics should be applied to utilise all the information in the data. For example, one should consider a more exhaustive search to find all overlaps

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

between *reads*, not just those that are easy to capture.

- For hybrid assembly where short and long reads are used together in an assembly, coverage depth has a stronger effect on the choice of assembly pipeline. For example, if high coverage *long-reads* are provided, it is possible to do a *long-read*-only assembly [24] and use *short-read* for genome polishing [51]. For lower *long-read* coverage, *long-read* should be corrected with *short-read* [17] prior to the *long-read* assembly. If *long-read* coverage is not sufficient for assembly, it is possible to use *long-read* to improve contiguity of a *short-read* assembly [2].
- For all assemblies, it should be noted that sequencing coverage depth is not fixed across the genome. Variation in coverage results in some regions having low (or no) *read* coverage that leads to a discontiguity in the resulting assembly. In this case, one should consider a pipeline that includes a scaffolding step.

It is important to use the same dataset, the same computer, and the same quality assessment method for evaluation of different assembly pipelines; otherwise, the comparison would be biased and inaccurate.

In our evaluation, all tools were given the same set of synthetic and real datasets with low and high sequencing coverage depth. We performed all tests on the same *High Performance Computing* (*HPC*) computer. We used a uniform, comprehensive evaluation framework (*QUAST* [16]) to assess all resulting assemblies. In addition to evaluating the assemblies resulting from each pipeline, we provided accurate performance measurements of execution times and peak memory usage. We also compared *read* corrections and genome polishing tools.

This paper complements previously published reviews. In a recent review [46], Sohn et al. compared several *short-read*, *long-read* and hybrid assembly pipelines. However, they did not perform their own analysis; rather, they used the results provided by the author of each method for the evaluation. Thus, their comparison metrics were limited to N50, which is provided by all authors. In [9], Bradnam et al. evaluated multiple assemblies submitted by several participating teams. Their main focus was on *short-read* assembly. Each participating team best optimised the pipeline for the given dataset. However, the error pattern varied in accordance with the sequencing technology and sample preparation (i.e., contamination). Participating teams were able to evaluate their assembly using closely related genomes prior to submission. The work in [37] is another review paper in which *long-read* sequenced data (*PacBio*) are not considered. Neither of these surveys includes the most recent *long-read* assemblers such as *Canu* [24].

2 MATERIALS AND METHODS

2.1 Algorithms and Methods

The input to the assembly process is a set of *reads* that are passed through all data preprocessing steps. There is a graph data structure in the heart of each assembly pipeline called the *assembly graph*. The goal of the assembly graph is to link small *DNA* fragments to one another to build the genome. The output of the assembly graph is a set of sequences called contigs which are usually large pieces of the genome without any information about their order in the genome. In the best scenario, there would be one contig per chromosome covering the entire chromosome. However, this is an idealistic goal for current assembly pipelines, especially when dealing with large genomes. Finally, the post-processing steps would improve the assembly by ordering these contigs, filling the gaps and fixing errors. Scaffolding is the process in which the order of the contig in the genome and the distance between them is predicted. Gap filling is the process of identifying the sequence between the scaffolds.

In this section, we first describe the data preprocessing steps that take place before graph assembly. Then we briefly explain how different graph algorithms construct contigs from the given *reads*. Next, we describe the postprocessing steps that take place after graph assembly.

Preprocessing

Data preprocessing involves one or more of the following steps; the last step is only applicable for *short-read* data.

- Filtering *reads* is to remove those that may increase assembly error rate.
- Fixing errors in the *reads* to reduce the complexity of the subsequent steps and to improve assembly quality.
- Reducing the volume of data by removing redundancy.

Read Filtering: Filtering is a commonly applied preprocessing technique for *short-read* data. The *reads* with low quality scores and/or ambiguous *bases* [44] are removed. Furthermore, duplicate *reads* can be removed [44].

short-read Error Correction: Error correction of *short-reads* is usually performed by utilising the *k-mer* frequency [39, 22]. *k-mers* are short, fixed-length subsequences of length k taken from a sequence. Even after performing *read* filtering, *reads* would still contain sequencing errors. Assuming 30X depth of sequencing coverage, *reads* are distributed over the genome evenly; considering a low error rate in *short-read*, each part of the genome is expected to be covered by nearly 30 identical *k-mers*. A rare (low frequency) *k-mer* could be the result of an error in a *read*, or it could be the result of lack of *reads* covering a specific region of the genome. Any *k-mer* with low frequency can be corrected by replacing it with the closest high-frequency

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

k-mer. However, there are difficulties in this process too; for example, regions of the genome might be repetitive or similar to other regions, resulting in uneven distribution of *reads* mapped across the genome.

long-read Error Correction using short-read: The error correction of *long-reads* requires a different approach and is more computation-intensive. In the early stage of *long-read* sequencing technology, the rate of sequencing error was very high and assembly of *long-read* data was not feasible. In addition, it was not possible to correct one *long-read* using the information on another *long-read* due to the high error rate. Since *short-reads* were quite accurate and cheap, using them to correct errors in *long-read* data became the prevailing solution [1, 41, 17, 18].

Figure 1 shows the process of mapping *short-reads* to *long-reads* which is usually done by *read* mappers such as *BWA*. First, the entire *long-read* dataset must be indexed using *FM-Index*. Then for each *short-read* several subsequences (seeds) are taken and searched to find their exact matches in the *long-reads*. Regions with enough similarities are called candidate regions. Each *short-read* is then aligned to all corresponding candidate regions. Due to high error rate in *long-reads*, sufficient number of *short-read* seeds must be searched to identify possible candidate regions [17]. Figure 2 shows the consensus process in which the target sequence (*long-read* here) is corrected when there is enough evidence of an error based on the mapped sequences (*short-reads* here). The consensus operation is used in different applications but the target sequence and the mapped sequences are different for each application.

In this method of correction, *short-reads* are mapped to the *long-reads* and then a consensus process identifies and fixes the errors in the *long-reads*. When mapping *short-reads* to *long-reads*, a low similarity is expected compared to mapping *short-reads* to a *reference-genome*. Hence, the mapper should perform a comprehensive search in order to map a *short-read* to a *long-read*.

This error correction method can be applied iteratively as many times as required (or feasible). For example, the method proposed in [17] applies three rounds of correction.

long-read Error Correction without short-read (Finding Overlaps): As high-coverage *long-read* sequencing became more affordable, tools have been developed to correct errors in *long-reads* without the need of *short-reads* [24, 49]. This approach comprises two steps: 1) finding overlaps between *long-reads*; and 2) correcting errors through a consensus operation.

Exact match overlaps are not expected for *long-reads* as the error rate is high and overlaps are usually long. Thus, any region that is sufficiently long and similar between *long-reads* is considered to be an overlap. To find similar regions, each *read* must be searched in the entire *read* dataset. Since scanning the entire dataset for each *read* is not feasible, the *read* dataset must be preprocessed and indexed to facilitate the search process. The produced index is used to quickly find identical *k-mers* between one *read* and all other *reads* in the dataset. When there are enough identical *k-mers* to support the possibility of overlap between two *reads*, further processing checks whether the overlap truly exists. Finally, *reads* are aligned in the overlap region using a pairwise alignment algorithm.

Due to the large size of the *long-read* dataset, it is not practical to index all *k-mers* that exist in the entire dataset. For large genome or high coverage sequence data, the full index cannot fit in the computer memory (RAM) [1]. There are two solutions to deal with this problem [28, 5]:

- Split the *read* dataset into N parts, such that the index for each part does not exceed available memory. This solution requires N times less memory but N times more processing, as each *read* must be searched in each part of the index separately.
- Partial indexing, in which not all *k-mers* are stored in the index. This reduces the chance of identifying common *k-mers* using the index. The lack of *k-mers* in the index is compensated for by the expected long length of overlap between *long-reads*, which increases the chance of finding common *k-mers* in the overlap region.

When the overlaps are discovered, a consensus operation corrects the errors in each *read* using all other *reads* overlapping the current *read*.

Reducing Redundancy in short-read Data: In addition to the filtering and error correction discussed above, data preprocessing in some cases can also include reducing the volume of data by removing redundancy [55] or even using a subset of all data for the assembly. However, this step is only applicable for *short-reads*.

Assembly Graphs

There are two assembly graphs [29]: *de-Bruijn* graphs [54] and overlap graphs [35].

de-Bruijn Assembly Graph: In the *de-Bruijn* assembly graph, *reads* are split into overlapping *k-mers*. Nodes of the graph represent *k-mers*. A directed edge from node N_a to node N_b indicates that N_b is next to N_a in a *read*. The number of nodes in the *de-Bruijn* graph is the total number of identical *k-mers* in the genome plus *k-mers* that do not exist in the genome but in the *reads* (due to errors). The weight on the edge indicates the number of times N_b is observed next to N_a in all *reads*. Thus, the weight of an edge indicates the possibility that two *k-mers* appear after each other in the *DNA* sequence. A path in the graph where all edges have a high weight is likely to be a part of the genome. Figure 3 illustrates an example of *de-Bruijn* graph assembly.

[1] In [53], nearly all *k-mers* of the human genome are indexed in about 40GB space. However, for 100X sequencing coverage depth, the *read* dataset is 100 times larger than the entire genome and need much larger memory which is not available.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

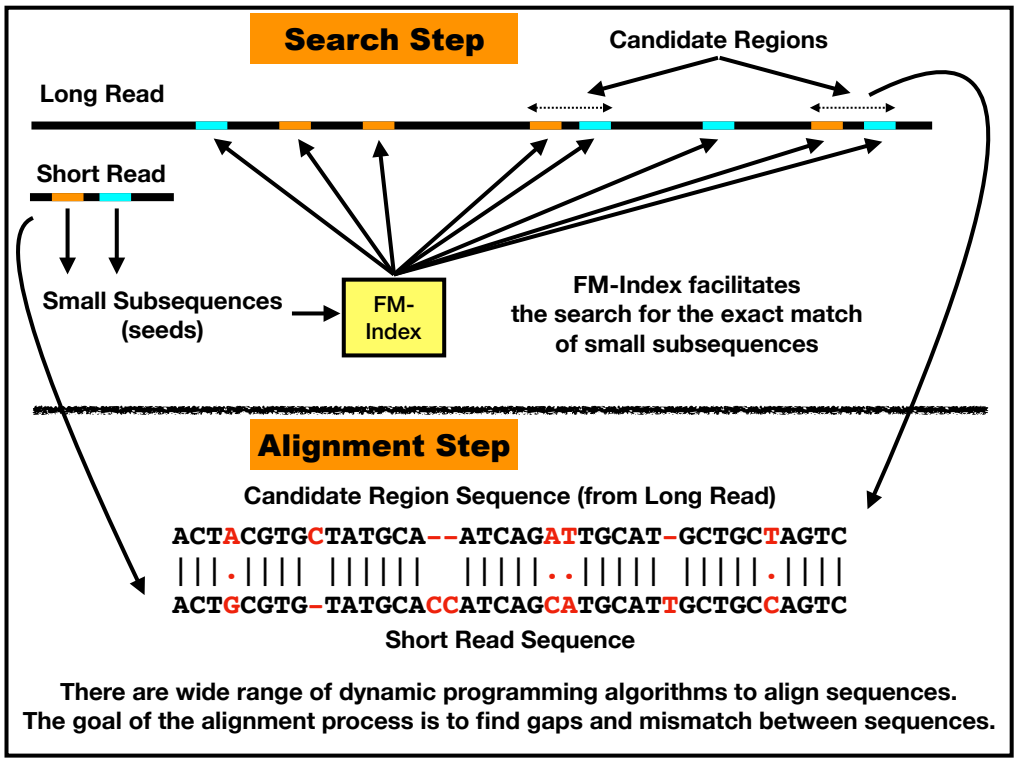

Figure 1: An example of *read* mapping. Subsequences of *short-reads* are searched in the *FM-Index* of *long-reads*. When regions with enough similarities (candidate regions) are identified, the *short-read* is aligned to each of those regions.

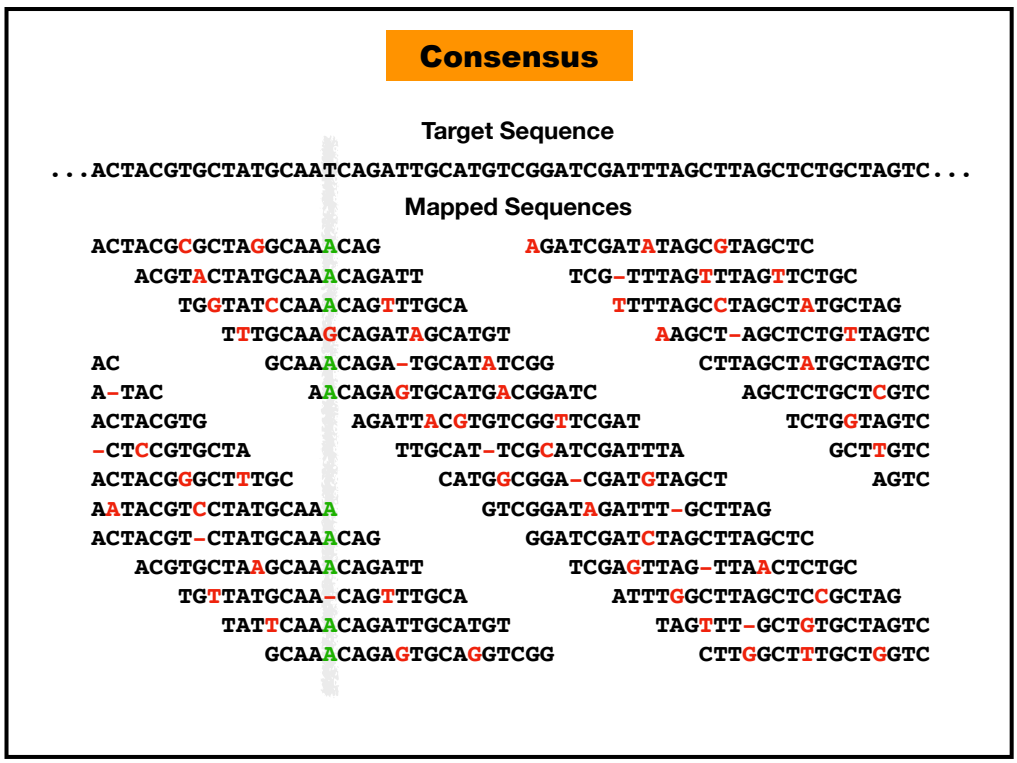

Figure 2: An example of a consensus process. The target sequence is corrected whenever there is enough evidence of error based on mapped sequences. For example the highlighted column in this figure.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

The majority of *short-read* assemblers use a *de-Bruijn* assembly graph. Due to the high error rate of *long-reads*, the *de-Bruijn* graph is not well suited for the *long-read* assembly pipeline.

OLC Assembly Graph: In *Overlap-Layout-Consensus* (*OLC*) assembly (also known as string overlap graph assembly), nodes of the graph represent the entire *read* sequence and directed edges represent the existence of overlap between *reads*. For the *OLC* graph, the number of nodes is fixed and equal to the number of input *reads*. In this graph, each path from node N_a to node N_b reveals slightly different sequences. A consensus operation constructs the most likely sequence given all paths. Figure 4 is a small example of *OLC* based assembly.

Prior to the assembly graph traversal to produce contig sequences, the graph should be cleaned. Cleaning involves resolving loops and removing edges that may be false positives. In addition, there is a process for the string overlap graph called *transitive edge reduction* [42, 34] which aims to reduce redundant edges from the graph prior to the cleaning step.

The assembly graph is also able to identify the possible order of some of the contigs in the genome [27]. Such information is reported along with contigs and can be utilized to increase the accuracy of the scaffolding step.

Very few *short-read* assemblers use *OLC* assembly graphs. However, to the best of our knowledge, all *long-read* assembly pipelines use the *OLC* approach. For the same sequencing coverage depth of the same genome, there are fewer *long-reads* compared to *short-reads*; this results in a smaller assembly graph and subsequently faster graph processing in the assembly pipeline.

Postprocessing

Postprocessing involves approaches that further improve the quality of the assembly. Three commonly used approaches are:

- Scaffolding
- Gap filling
- Genome polishing

Scaffolding and gap filling: When contigs are constructed, scaffolding [8] and gap filling [7] are two processes that join contigs to build larger contigs called scaffolds (also known as *unitigs*). In both scaffolding and gap filling, *paired-end* and/or *mate-pair reads* are mapped to contigs. Since the approximate distance between paired *reads* is known, it is possible to identify the order of contigs and the approximate distance between them in the genome. Gaps can be filled when a *read* is mapped to a contig and the paired *read* is located in the gap area. It is also possible to use *long-reads* for scaffolding [8] and gap filling [21]. Figure 5 is an example showing scaffolding and gap filling process.

Polishing: Genome polishing is a process that can be applied to contigs and/or unitigs iteratively as many times as needed. Genome polishing fixes errors in the genome by aligning *reads* to the genome. If possible, utilizing data which are not employed in the assembly process would result in better polishing as the assembled genome is currently biased toward the *reads* used in the assembly process. In this process, additional *reads* are mapped to the assembled genome and then a consensus operation identifies errors. When some of the errors are fixed, there is a higher chance for *reads* to be mapped to the erroneous region of genome correctly; subsequently, there is a higher chance of fixing more errors [17].

Hybrid Assembly

Hybrid assembly refers to the use of multiple types of data in the assembly process. We have already pointed out some of the hybrid assembly methods above such as correction of *long-read* using *short-read* or polishing assembly using *paired-end* reads. However, there are several other ways in which we can utilize different types of data in the assembly process. Here we list known patterns for hybrid assembly methods. We also explain how correction and polishing processes differ from each other.

- Using *short-reads* to correct errors in *long-reads* [1, 1, 17, 41].
- Using *short-reads* to polish genomes assembled from *long-reads* [51].
- Using *short-reads* to scaffold contigs from a *long-read* assembly pipeline.
- Using *long-read* to scaffold contigs from a *short-read* assembly pipeline [2].
- Assemble the genome with *short-reads*, *long-reads* and a hybrid pipeline, and then merge the assemblies to produce a more accurate assembly [52].
- Using contigs produced by *short-reads* to correct errors in *long-reads* [3].

Correction and polishing are perhaps the most common approaches in hybrid assembly pipelines. The main difference between correction and polishing is the expected error rate in the target sequence. This results in different parameters during mapping as well as different algorithms for identifying errors. In both correction and polishing, *short-reads* are mapped to the target sequence and a consensus process identifies and fixes the errors in the target sequence. The target sequence is usually a *long-read* for the correction process or a contig for the polishing process.

When mapping *short-reads* to *long-reads*, the expected similarity is low and the mapper should do a more comprehensive search in order to map a *short-read* to a *long-read*. In addition, it is less likely to map both ends of a *paired-end read* correctly as the mapping rate is usually low. In general, *short-reads* are treated as *single-end reads* during the *long-read* correction process. Mapping *short-reads* to contigs for polishing is a less difficult problem. In fact, one can

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

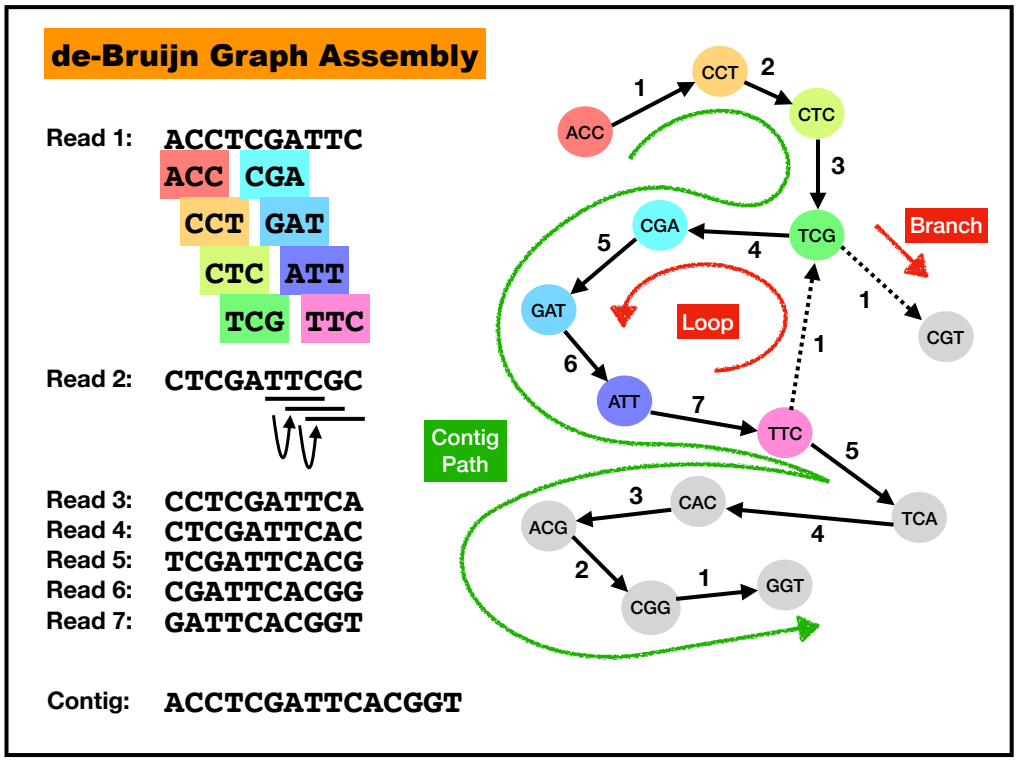

Figure 3: A tiny *de-Bruijn* graph example with *k-mers* of length 3. All overlapping *k-mers* taken from *Read* 1 are highlighted with distinct colours. The placement of these *k-mers* in the graph is also highlighted with the same colours. All *k-mers* form *Read* 2 to *Read* 7 are added to the graph in the same fashion. The weight on each edge of the graph represents how many times two *k-mers* are seen to be adjacent in the *reads*. The green path in the graph follows the heaviest edges and represents a contig. Due to errors in *Read* 2, a loop and a branch are created in the graph which is shown with the red lines. In reality, the graph is enormous with a large number of loops and many branches.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

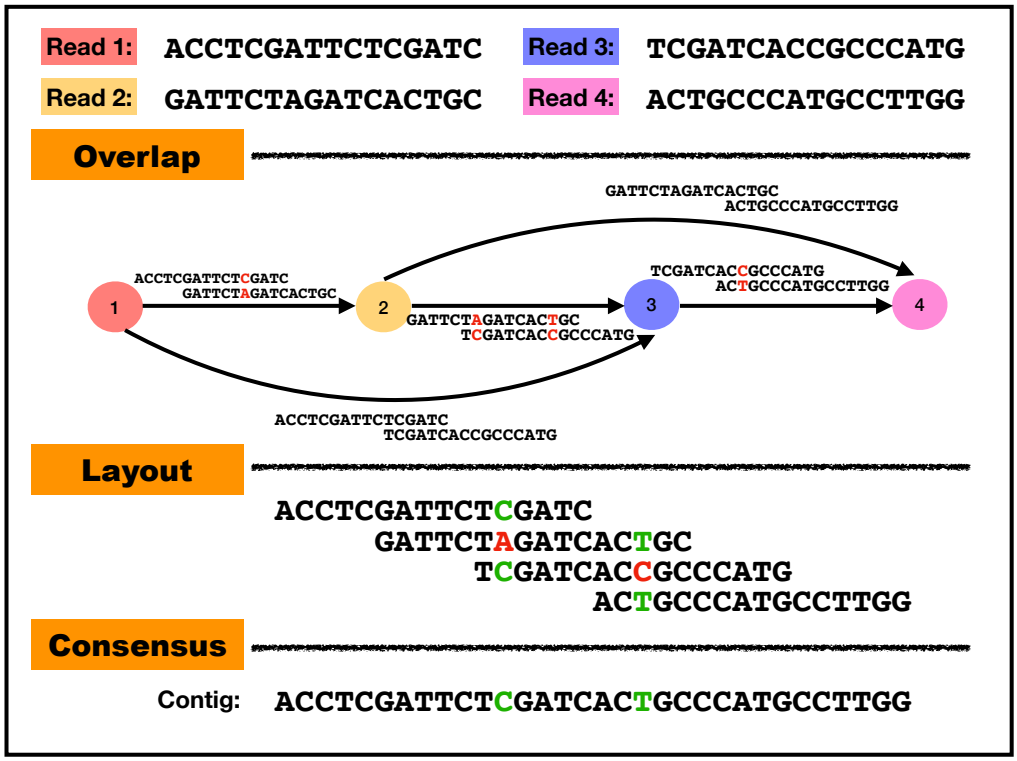

Figure 4: A tiny *Overlap-Layout-Consensus* graph example. In the overlap step, all overlaps between *reads* are listed in a graph structure. The layout step parses the graph and put the sequences in order. Finally, the consensus step generates the contig. Similar to the *de-Bruijn* graph, there might be loops and branches in the overlap graph which is not presented here for simplicity.

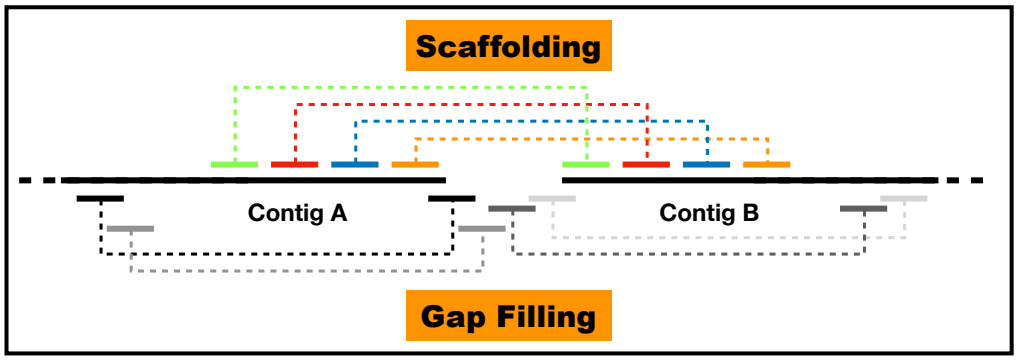

Figure 5: A small example of scaffolding and gap filling process. For the scaffolding, those *paired-end* and *mate-pair reads* are chosen that each read in a pair is mapped to the edge of a different contig. For the gap filling, one read of the pair is mapped to the edge of contig and the other is located in the gap area.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

use the same parameters as those used to map *reads* to the *reference-genome*.

For error identification, polishing tools are more conservative and require much more evidence to call an error. In addition, they are less likely to trust *reads* with low-quality mapping. On the other hand, the correction process tends to rely on all mapping information and is less conservative in calling errors.

Both the correction and the polishing process can be applied iteratively as many times as required (or feasible). In every iteration, more errors are corrected in the target sequence and the chance of mapping *short-reads* to erroneous regions is increased for the next iteration. Thus, more errors will be corrected in the next iteration. However, in reality, the number of corrected errors in each iteration drops dramatically. Furthermore, not all errors can be corrected due to difficulties in the mapping process. As a consequence, after several iterations, the number of errors remains almost constant, with slight fluctuations.

2.2 Assembly Software and Pipelines

short-read Assembly

To construct and traverse the assembly graph in the *short-read* assembly process, *short-reads* are usually treated as *single-end reads*. *paired-end* and *mate-pair* information are mainly used for scaffolding, gap filling and polishing. We elaborate three software/pipelines for *short-read* assembly below.

SGA: String Overlap Graph (*SGA*) [43] is a complete assembly pipeline that uses the *OLC* approach. *SGA* offers several tools for preprocessing and checking the quality of input data prior to assembly. Although not very common, *SGA* also offers error correction for *short-reads* to improve the quality of assembly.

In order to identify overlaps between *reads*, *SGA* uses *FM-Index* [15]. *SGA* limits its search to the overlaps that exactly match between *reads*. *SGA* also allows searching for inexact overlaps, which have only a few differences. This increases the chance of finding more overlaps and possibly generating longer contigs. However, the associated computational cost for inexact searches are high and this option is disabled by default. Since the number of *short-reads* and the overlaps between them is huge, *SGA* removes redundant overlaps using a transitive edge reduction algorithm [42] when overlaps are identified, before constructing the graph.

Since *reads* are randomly distributed across genomes, the chance of having an overlap of different lengths is equal. However, shorter overlaps are likely to be false-positive overlaps. As a consequence, *SGA* only considers overlap longer than 45 *base-pairs* in the process. This threshold is difficult to tune as reducing it increases the number of false-positive overlaps and subsequently introduces more errors in the assembly. Conversely, the chance of having errors in longer overlaps is high and considering only exact matches results in missing many true overlaps.

There are tools in the *SGA* pipeline for scaffolding and gap filling as well. Since *SGA* takes a conservative assembly approach, the resulting contigs are shorter compared to *Spades*; instead, *SGA* delivers the highest *base* calling accuracy.

Spades: Spades [2] is a *de-Bruijn* graph-based assembly pipeline that packs all processing steps into a single command. In addition to *short-reads*, *Spades* offers integration of other sources of data, including contigs produced by other pipelines as well as *long-reads*. Spades uses these additional data to produce even longer contigs. *Spades* allows integrating *long-reads* in a *short-read* assembly. *long-reads* are mainly used to scaffold contigs from a *short-read* assembly. Integration of *long-reads* in the *Spades* pipeline is discussed later in the hybrid assembly section.

SOAPdenovo2: SOAPdenovo2 [30] is an optimized *de-Bruijn* graph assembly pipeline that is faster than *Spades* and *SGA*. *SOAPdenovo2* implements a sparse graph assembly algorithm that significantly reduces the memory footprint of the graph (the maximum memory that the program uses). It builds the graph using small *k-mers* first and corrects sequencing error. Then a large *k-mer* is used to build the graph for assembly. The larger *k-mer* better resolves loops in the graph.

long-read Assembly

The advent of long-read sequencers, from *PacBio* and *Oxford Nanopore* platforms, has made it possible to assemble genomes using long reads. These may be up to tens of kilobases in length, as compared to the very short reads (100 bases in length) from *Illumina* platforms. A number of *long-read* assemblers have been developed to work with the *long-read* data.

HGAP and Falcon: HGAP [11] and Falcon [12] are early long read assembly which are widely used. HGAP uses the longest *reads* as seeds in a directed acyclic graphbased consensus procedure to construct a highly accurate preassembled *reads*. Then it follows with the assembly using off-the-shelf long-read assemblers. Falcon is another genome assembly with the focus of producing a phased diploid assembly.

Canu: One of the most recent *long-read*-only assembly pipelines is *Canu*, which was developed based on the *Celera* [13] assembler. *Canu* is a scalable software program that can automatically identify available resources and configure all parameters such that resource utilization is maximised without exceeding the available memory.

The *Canu* assembly pipeline uses *MHAP* [5] to find overlaps. *MHAP* uses *MinHash* [10], which was originally developed to determine the similarity of web pages. *MinHash* uses a multiple hash function, which increases the chance of identifying similarities between sequences. *Canu* uses Myers' O(ND) algorithm [33] for the alignment and a modified version of the "falcon_sens" algorithm [12] for the correction. The *Canu* assembly pipeline uses the modified version of "Best Overlap Graph" [32], an *OLC* assembly algorithm for graph assembly.

MiniC and Mini: Heng Li introduced *Minimap* and *Miniasm* in [27]. Later, the *Minimap2* was introduced in [28] with several advantages compared to *Minimap*. *Minimap2* is an overlap finding software designed for

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

long-reads. *Miniasm* is software that can construct and traverse the assembly graph from the overlap produced by *Minimap2*. Since *Miniasm* does not have the consensus module, *Racon* [49] has been proposed to be used as a consensus module for *Miniasm*. *Racon* also provides the ability to correct *long-read* errors using *long-read* overlaps. However, it is claimed that *Racon* can assemble genomes without the need for *read* correction. We consider this recent long-read-only assembly pipeline using *Minimap2*, *Miniasm* and *Racon* in our evaluation with and without the long-read correction step. We refer these pipelines as *MiniC* and *Mini* respectively. These two pipelines are described in detail in [28, 27, 49].

In the *MiniC* pipeline:

1. *Minimap2* finds overlaps between *long-reads*.
2. *Racon* corrects and trims *long-reads* based on the discovered overlap.
3. *Minimap2* finds the overlap between corrected *long-reads*.
4. *Miniasm* assembles the genome using overlaps between corrected *long-reads*.
5. *Minimap2* finds the overlap between assembled contigs and original *long-reads*.
6. *Racon* corrects errors in the contigs.

In the *Mini* pipeline:

1. *Minimap2* finds the overlap between *long-reads*.
2. *Miniasm* assembles the genome using the overlaps between *long-reads*.
3. *Minimap2* finds the overlap between assembled contigs and *long-reads*.
4. *Racon* corrects errors in the contigs.

Although *long-reads* are corrected prior to assembly in the *MiniC* pipeline, contigs are much longer than individual *long-reads*; an overlap between a *long-read* and a contig is more reliable than the overlap between original *long-reads*. Thus, the contig correction in the last two steps can further improve the quality of the assembled genome.

Minimap2 uses *Minimizers* [40] to reduce the number of *k-mers* in the dataset to be indexed. In a window of length $k + w$ where k is the length of *k-mer*, there are w overlapping *k-mers*. For each window, the *k-mer* that minimizes a specific hash function is called *minimizer* and is stored in the index. *Minimap2* stores minimizers of all overlapping windows in the input *reads*. Since there is a strong chance that the same *k-mer* is the minimizer of multiple overlapping windows, the number of *k-mers* to be indexed drops. However, the most efficient set of *k-mers* is selected to be indexed. In fact, *MinHash* used by *Canu* is a generalized version of *Minimizer*.

With smaller w, there would be less *k-mer* to be indexed; at the same time, however, there would be less chance to capture overlaps between *reads*. The small value of k results in *k-mers* that are repetitive and not usable for searches. On the other hand, a large value of k reduces the chance of identifying overlaps since the chance of having large identical *k-mers* is low in erroneous *long-reads*. An appropriate value of k should be chosen with care.

Racon is error correction software for *long-reads*. First, the overlaps between *long-reads* should be discovered using *Minimap2*. Then the *reads* must be aligned in the overlap region prior to correction. Both *Minimap2* and *Racon* can do the alignment step. *Minimap2* uses the Smith-Waterman [45] algorithm implemented in *K-lib* [25]. *Racon* uses Myers' bit-vector [36] algorithm implemented in *Edlib* [47].

Miniasm constructs and traverses the assembly graph. Since there are false-positive overlaps, *Miniasm* cleans up the graph in multiple iterations. Unlike *SGA*, *Miniasm* removes transitive edges after constructing the graph in the memory using a modified version of the algorithm provided in [34]. *Miniasm* produces contig graphs, which are contigs with their possible order in the genome. However, most of the subsequent processes accept contigs in a *Fasta* file and do not take into account the additional information *Miniasm* produces.

Hybrid Assembly

Colormap: Hybrid *long-read* error correction tools do not use *k-mer* indexing at all. Instead, they use *short-read* mappers such a *BWA* [26] and *Bowtie* to map *short-reads* to *long-reads*. These tools use *FM-Index* to index target sequences. *FM-Index* is a full index and its memory footprint is much smaller than *k-mer* indexing. A common *FM-Index* for a human genome only takes about 4GB of memory. However, both index generation and search time is slower compared to the *k-mer* indexing method.

Colormap [17] is a three-iteration correction process that uses *BWA* to map *short-reads* to *long-reads* and two correction algorithms named *spCorr* and *OEA*. In the first two iterations, *spCorr* identifies and corrects errors. In the third iteration, *OEA* is used for correction. The third iteration can be considered a polishing step after the first two rounds of correction.

HALC: HALC [3] is another *long-read* correction tool that requires already assembled contigs. It first aligns *long-reads* to contigs using *Blasr*. It then corrects errors in the *long-reads* but not the contigs. *HALC* allows inputting *short-reads* as well, but the input contigs are mandatory. This raises the following questions about this approach:

- If the genome is already assembled, what is the purpose of correcting *long-reads*?
- If the correction process is biased by the current assembly, how can the corrected *long-reads* enhance the quality of the current assembly?
- Only *long-reads* that overlap with available contigs can be corrected. Those *long-reads* that be-

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

long to undiscovered part of the genome remain untouched.

Pilon: Pilon [51] is a set of polishing tools using *short-reads*, which can fix *base* calling errors as well as some misassemblies. We polished the contigs produced by *long-read* assembly pipelines including *Mini* and *Canu*. We also examined the effects of applying multiple rounds of polishing.

Metassembler: Metassembler [52] is a program that allows merging multiple *de-novo* assemblies in order to produce a more accurate assembly. It can be considered a hybrid assembly pipeline if assemblies with *short-reads* and *long-reads* are merged together.

3 RESULTS AND DISCUSSION

In order to evaluate existing *de-novo* assembly tools, we use three real and two synthetic datasets. Real datasets are sequenced data along with high quality assembled genome from three strains of *Klebsiella pneumoniae* provided by *Bioplatforms Australia* [6]. For each real dataset, the provided high-quality assembled genome is used as a ground truth genome. More details on the real datasets we used are available in [6]. There are considerations regarding the use of both real and synthetic datasets.

- Synthetic datasets do not reflect the full complexity existing in a real dataset. For example, a real dataset includes contamination and structural error where simulators can rarely model them accurately. However, for an accurate comparison, the ground truth genome is required, especially when the error rate of evaluated assembly pipelines is low. Using a high-quality assembled genome (in a real dataset) as a ground truth might not distinguish accurate pipelines precisely.

- The complexity that exists in real datasets greatly depends on the sample preparation method (including contamination and sample quality) and sequencing technology (the type of errors). This complexity can vary between real datasets. Thus, evaluation based on one real dataset may not be the same as the comparison based on another real dataset.

- Real datasets used in this evaluation are limited to a small bacteria genome. Using a synthetic dataset, we are able to evaluate existing assembly methods when dealing with larger genomes.

For each genome, we prepared *short-reads* and *long-reads* with high and low *read* coverage depth (40X and 20X respectively) to evaluate the effect of *read* coverage on assembly pipelines. For the synthetic dataset, we used *ART* [19] to simulate *short-reads* and *SimLoRD* [48] to simulate *long-reads*. Table 1 represents datasets used in our evaluation. The smaller synthetic genome (SIM5M) was generated by taking the first five million *base-pairs* of the *reference-genome* used for AJ055 evaluation. The larger synthetic genome (SIM25M) was 25 million *base-pairs* taken from part of human *reference-genome* (hg19) chromosome 1 (excluding `N` in the sequence). We were not able to process the larger genome as some pipelines exceeded our computational resources to process data. All real and synthetic *short-reads* were of length 150 *base-pair*.

All of our tests were run on a Linux machine with 16 processors and 64GB of memory. Except for *Canu*, which automatically identifies and utilizes resources, we set the parallelisation factor to 16 for all tools where applicable. In all our evaluations, we ensured there was sufficient RAM in the system and neither of our tests reached system swap memory. If this happens for a process, it will noticeably slow down the process and the measured execution time would not be appropriate for comparison.

In the rest of this section, we compare a number of *de-novo* genome assembly pipelines using the datasets and machine described above. To ensure a fair comparison across the assembly pipelines, there are some important considerations which are also highlighted in [9].

- We did not observe one pipeline to be best in all aspects. One pipeline was fast, one was scalable, one delivered the highest *base* calling accuracy, one resulted in the highest genome coverage, and one produced the longest contigs.

- The application for which the genome is assembled defines which pipeline is the most appropriate. In order to compare assembly pipelines, one should know what the expectations are regarding the accuracy of the assembled genome (genome coverage, contig length, *base* calling accuracy) and what the expectations are regarding computational costs (scalability, speed and maximum memory usage).

- Each pipeline accepts several parameters that can significantly affect its performance and accuracy. In our evaluation, we mostly ran pipelines with their default parameters unless otherwise mentioned. If the same dataset is processed with the same pipeline but different parameters, it is possible to get different speeds and accuracy.

The runtime and the memory usage for all pipeline processing all dataset are provided in Figure 6 and Figure 7 respectively. We group accuracy metrics both by the dataset and by the pipeline. For example, the tabular data in Figure 8 shows accuracy metrics for dataset AJ055 processed by all pipelines and Figure 9 shows all accuracy metrics for all datasets processed by *MiniC* pipeline. The complete evaluated metric and the *QUAST* interactive report (including charts) in *HTML* format, along with script we used to prepare synthetic data and execute each pipeline are available on `https://arashbayat.github.io/asm`. **Note that the evaluation results discussed in the paper are mainly related to the AJ055 dataset.** Also, note that *QUAST* does not consider contigs shorter than 500 *base-pair* in the evaluation.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

Table 1: Datasets used in out evaluation. Dataset with **AJ** prefix are real datasets and datasets with **SIM** prefix are synthetic datasets.

Dataset	Genome Size
AJ055	~5.5 Mbp
AJ218	~5.5 Mbp
AJ292	~5.5 Mbp
SIM5M	5 Mbp
SIM25M	25 Mbp

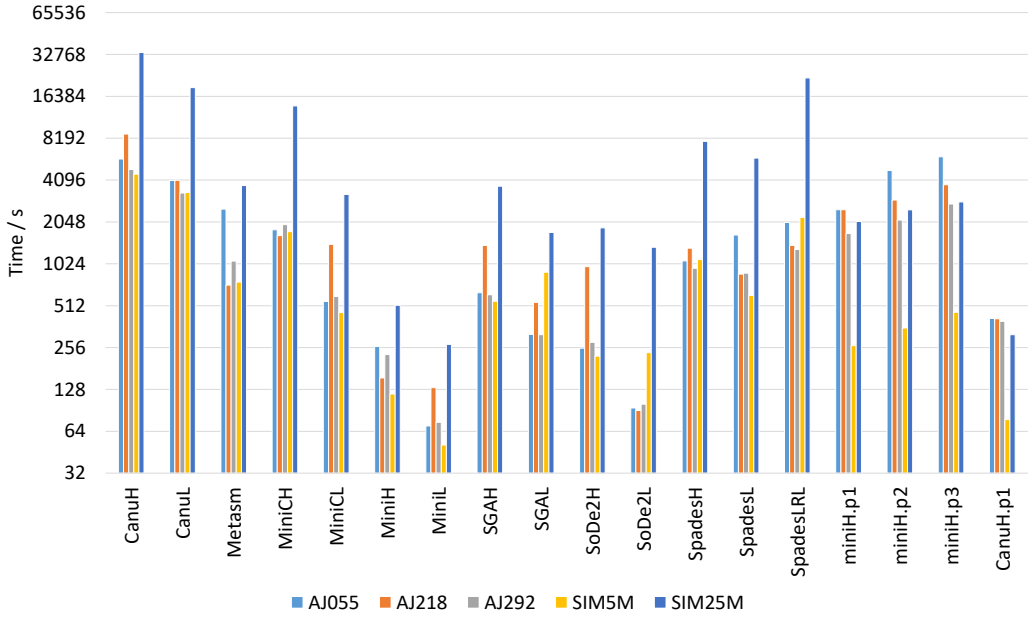

Figure 6: Colour coded runtime for all pipelines processing all datasets. For the details of assembly pipeline names refer to the first 5 rows of Figure 8. Green is better.

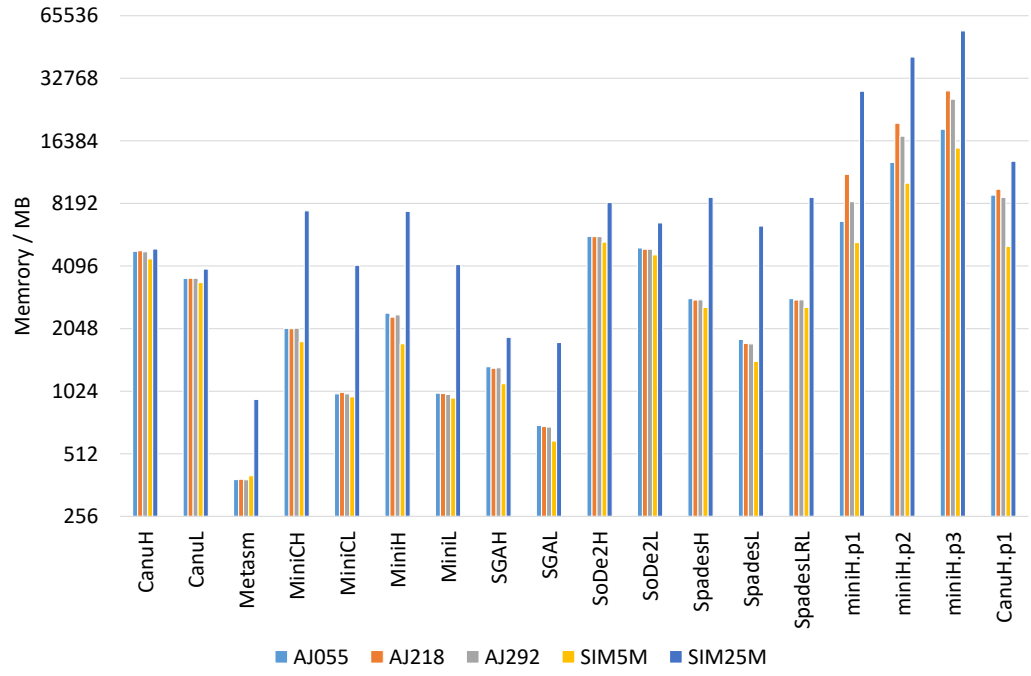

Figure 7: Colour coded Pick memory usage for all pipelines processing all datasets. For the details of assembly pipeline names refer to the first 5 rows of Figure 8. Green is better.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

	Pipeline info	Long read only assembly						Short read only assembly						Hybrid assembly						
		Canu		MiniC		Mini		SGA		SOAPdenovo2		Spades		Spades	Colormap	Canu+pilon	Mini+pilon			Metasm
	Coverage	40X	20X	40X	20X	Mini	20X	40X	20X	40X	20X	40X	20X	40X long 20X short		40X long reads 40X short reads				
	Assembly	CanuH	CanuL	MiniCH	MiniCL	MiniH	MiniL	SGAH	SGAL	SoDe2H	SoDe2L	SpadesH	SpadesL	SpadesLRL	Colormap	CanuH.p1	MiniH.p1	MiniH.p2	MiniH.p3	Metasm
Statistics with reference	Genome fraction (%)	99.877	99.785	99.999	98.626	99.934	97.962	97.757	97.119	27.577	55.537	99.131	98.932	99.882	97.625	99.877	99.939	99.939	99.939	99.877
	Duplication ratio	1	1.003	1.005	1.007	1.005	1.009	1.024	1.015	1.011	1.005	1.001	1.001	1	1.094	1	1	1	1	1
	Largest alignment	5496967	3169756	5529367	1138217	5529303	833675	37194	33813	3463	4124	585792	342668	1587076	504450	5497400	5500863	5501187	5501107	5496967
	Total aligned length	5498061	5506645	5530466	5467427	5530402	5435975	5508142	5427729	1521547	3069319	5458002	5447750	5498433	5871337	5498494	5501955	5502281	5502201	5498061
	NG50	5496970	3169885	5529667	470381	5534010	455694	6594	7167		589	226024	164005	1571477	305685	5497403	5505273	5505584	5505476	5496970
	NG75	5496970	1809348	5529667	298434	5534010	353027	3728	4097			163120	89646	552169	132047	5497403	5505273	5505584	5505476	5496970
	NA50	5496967	3169756	5529367	470380	5529303	455694	6096	6988	714	875	226024	164005	1571477	218907	5497400	5500863	5501187	5501107	5496967
	NA75	5496967	688559	5529367	298434	5529303	243400	3240	3760	575	645	163120	86634	552169	81683	5497400	5500863	5501187	5501107	5496967
	NGA50	5496967	3169756	5529367	470380	5529303	455694	6296	7092		568	226024	164005	1571477	305685	5497400	5500863	5501187	5501107	5496967
	NGA75	5496967	1120789	5529367	298434	5529303	351127	3474	3879			163120	89646	552169	130178	5497400	5500863	5501187	5501107	5496967
	LG50		1		1	4	1	5	249	237		2801	9	12	2	7	1	1	1	1	1
	LG75		1	2	1	7	1	8	524	490			16	23	4	15	1	1	1	1	1
	LA50		1	1	1	4	1	5	272	247	813	1240	9	12	2	8	1	1	1	1	1
	LA75		1	3	1	7	1	9	587	519	1447	2305	16	24	4	19	1	1	1	1	1
	LGA50		1	1	1	4	1	5	258	241		2900	9	12	2	7	1	1	1	1	1
	LGA75		1	2	1	7	1	8	550	503			16	23	4	15	1	1	1	1	1
Misassemblies	No of local misassemblies		0	1	1	2	5	4	0	0	548	197	11	17	1	12	0	2	2	1	0
	No of unaligned mis. contigs		0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Mismatches	No of mismatches per 100 kbp		0.27	6.28	15.03	34.91	29.62	49.96	0.06	0.24	1.71	2.78	5.22	7.09	5.57	24.7	0	1.04	0.69	0.8	0.27
	No of indels per 100 kbp		34.76	165.41	330.97	639.88	466.43	726.07	0.59	0.47	1.91	1.01	1.45	1.4	1.75	8.04	2.58	55.57	8.05	7.82	34.76
	No of N's per 100 kbp		0	0	0	0	0	0	0	0	578.49	95.3	17.33	25.66	1.77	0	0	0	0	0	0
Statistics without reference	No of contigs		4	9	2	16	2	18	1370	1228	2124	3607	64	103	16	73	4	2	2	2	4
	Largest contig		5496970	3169885	5529667	1138302	5534010	833810	37194	33813	3463	4124	585792	342678	1845647	509223	5497403	5505273	5505584	5505476	5496970
	Total length		5727073	5728738	5696122	5634250	5700578	5602098	5670252	5585990	1618182	3190953	5621063	5611391	1845647	5663299	6030828	5727525	5670982	5671342	5671230	5727073
	N50		5496970	3169885	5529667	470381	5534010	455694	6394	7111	751	907	226024	164005	1571477	218907	5497403	5505273	5505584	5505476	5505476	5496970
	N75		5496970	1809348	5529667	298434	5534010	243471	3500	3969	603	672	163120	86650	552169	104347	5497403	5505273	5505584	5505476	5505476	5496970
	L50		1	1	1	4	1	5	262	243	779	1204	9	12	2	8	1	1	1	1	1	1
	L75		1	2	1	7	1	9	559	505	1384	2230	16	24	4	18	1	1	1	1	1	1
	GC (%)		57.18	57.14	57.21	57.13	57.19	57.17	57.39	57.5	54.53	56.21	57.28	57.32	57.25	57.27	57.19	57.24	57.23	57.23	57.18

Figure 8: Accuracy metrics of all pipelines processing AJ055 dataset. All cells of the table are colour coded such that green, yellow and red represent best, moderate and worst respectively.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

	40X coverage long reads						20X coverage long reads				
Assembly	MiniCH						MiniCL				
Dataset	AJ055	AJ218	AJ292	sim5M	sim25M		AJ055	AJ218	AJ292	sim5M	sim25M
Genome fraction (%)	99.999	99.977	99.999	99.896	99.937		98.626	99.273	98.916	99.556	99.536
Duplication ratio	1.005	1.006	1.004	1.007	1.011		1.007	1.009	1.005	1.021	1.012
Largest alignment	5529367	5049176	5463805	3537045	4236176		1138217	1163455	1660035	1341548	2343337
Total aligned length	5530466	5496411	5463805	5029349	25259712		5467427	5472727	5410760	5077176	25170834
NG50	5529667	5485519	5464361	3537045	2347005		470381	615749	1327492	704173	1073552
NG75	5529667	5485519	5464361	1235134	1588793		298434	459808	720527	413350	761653
NA50	5529367	5049176	5463805	3537045	2347005		470380	610591	1327480	704169	1073551
NA75	5529367	5049176	5463805	1235133	1189240		298434	459808	638853	413350	758216
NGA50	5529367	5049176	5463805	3537045	2347005		470380	610591	1327480	704169	1073551
NGA75	5529367	5049176	5463805	1235133	1189240		298434	459808	638853	413350	758216
LG50	1	1	1	1	4		4	3	2	3	8
LG75	1	1	1	2	7		7	6	4	5	15
LA50	1	1	1	1	4		4	3	2	3	8
LA75	1	1	1	2	8		7	6	4	5	15
LGA50	1	1	1	1	4		4	3	2	3	8
LGA75	1	1	1	2	8		7	6	4	5	15
No of local misassemblies	1	4	2	0	0		2	10	1	0	6
No of unaligned mis. contigs	0	1	0	0	0		0	1	0	0	0
No of mismatches per 100 kbp	15.03	9.44	9.64	1.28	2.77		34.91	20.6	17.79	5.75	8.92
No of indels per 100 kbp	330.97	248.34	247.97	165.99	166.48		639.88	455.2	409.22	207.88	198.9
No of N's per 100 kbp	0	0	0	0	0		0	0	0	0	0
No of contigs	2	2	1	3	23		16	14	6	13	43
Largest contig	5529667	5485519	5464361	3537045	4487765		1138302	1168567	1660035	1348395	2343337
Total length	5696122	5688657	5464361	5029350	25259994		5634250	5675913	5414604	5087756	25176816
N50	5529667	5485519	5464361	3537045	2347005		470381	615749	1327492	704173	1073552
N75	5529667	5485519	5464361	1235134	1189240		298434	459808	720527	413350	761653
L50	1	1	1	1	4		4	3	2	3	8
L75	1	1	1	2	8		7	6	4	5	15

Figure 9: Accuracy metrics of all datasets processed by *MiniC* pipeline. All cells of the table are colour coded such that green, yellow and red represent best, moderate and worst respectively.

3.1 *short-read* Assembly

Table 2 compares the results for three *short-read* assembly pipelines. *SOAPdenovo2* is faster than other pipelines but results in poor assembly quality. *SOAPdenovo2* requires more memory compared to other assembly pipelines. *Spades* is the slowest pipeline but results in much longer contigs and higher genome coverage compared to the others. *Spades* memory consumption is not as low as *SGA* and not as high as *SOAPdenovo2*. *SGA* results in the highest *base* calling accuracy and requires the lowest amount of memory. However, the contiguity of the genome assembled by *SGA* is low.

Comparing high and low coverage data reveals an interesting point about each of these programs. In all programs and for all datasets, the processing time and peak memory usage is less for low coverage data compared to high coverage data. For *SGA* pipelines, the execution time and peak memory usage seem to have a linear relationship with the *read* coverage. For *SOAPdenovo2*, the execution time is 2.7 times higher than for twice the amount of input, which indicates *SOAPdenovo2* runtime is more sensitive to the number of input *reads*. On the other hand, *Spades* seems to be less sensitive to the size of input data and is only 1.6 times slower for twice as many *reads*. The peak memory usage of *Spades* and *SOAPdenovo2* do not seem to be linear either.

In terms of contiguity, *Spades* utilized higher *read* coverage well and produced longer contigs. On the other hand, the evaluation shows that *SGA* and *SOAPdenovo2* produce longer contigs for 20X data compared to 40X data. While this is unexpected, it is true for all other genomes. Note that *QUAST* only considered contigs longer than 500 *base-pairs* for evaluation. Since there were not enough contigs longer than 500 *base-pairs*, the NGA50 is not computed for *SOAPdenovo2* when processing 40X data. The NGA50 is the N50 for the portion of contigs that are aligned to the ground truth genome, excluding overlaps. The decrease in contiguity when a higher depth of *read* coverage was used could be the result of algorithmic weaknesses when dealing with a larger and more complex assembly graph.

For *Spades* and *SGA*, using 40X data resulted in a slight improvement in genome coverage compared to using 20X data. When the sequencing cost matters, one should consider whether this slight improvement is worth spending more on sequencing. The genome coverage was low for *SOAPdenovo2* when processing a real dataset. Also, the genome coverage of 40X data was lower than genome coverage of 20X data. The difference is significant for real data. In addition to the weakness of *SOAPdenovo2* in genome assembly, one reason for the lack of genome coverage is the fact that *QUAST* throws out the contigs shorter than 500 *base-pairs* prior to evaluation.

The error rate of all programs are generally low (being less than 3 errors per 100 kbp), except for mismatch errors of *Spades*. Note that to improve contiguity, the program should be less conservative and allow low coverage regions to be assembled as well. This can be a reason for the increase in the *base* calling error rate. Since the *base* calling accuracy was high in the *short-read* assembly approach, we did not study the effect of genome polishing on the resulting assembly.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

Table 2: Comparison of *short-read* assembly pipeline for the assembly of the AJ055 genome.

Pipeline Name:	SGA		SOAPdenovo2		Spades	
Read Coverage & Type:	40X SR	20X SR	40X SR	20X SR	40X SR	20X SR
Genome Coverage (%)	97.757	97.119	27.577	55.537	99.131	98.932
# Contigs	1370	1228	2124	3607	64	103
N50	6394	7111	751	907	226024	164005
NGA50	6296	7092	-	568	226024	164005
# Mismatches per 100 kbp	0.06	0.24	1.71	2.78	5.22	7.09
# Indels per 100 kbp	0.59	0.47	1.91	1.01	1.45	1.4
# Local Misassemblies	0	0	548	197	11	17
Execution Time (sec)	633	318	253	94	985	641
Memory Usage (MB)	1376	717	5824	5125	2925	1859

We also evaluated these assembly pipelines with a synthetic genome nearly five times longer to see the effect of genome size on the speed and accuracy of the assembly. The evaluation result for the longer genome (SIM25M). The evaluation showed that *SGA* genome coverage declined to about 90% for the longer genome, while the *Spades* genome coverage remained above 98%. The execution time and peak memory usage were also affected by the genome size, as shown in Table 3. While the execution time is linearly increased with the genome size for the *SGA*, *SOAPdenovo2* and *Spades* show a higher sensitivity to the genome size. The genome size had less effect on peak memory usage and its effect varied between tools and differing *read* coverage.

3.2 *long-read* Assembly

Error Correction

In order to distinguish between different correction algorithms, we corrected both low coverage (20X) and high coverage (40X) *long-read* datasets using a *Canu* correction module and a combination of *Minimap2* and *Racon* (alignment was done by *Racon*). We evaluated the corrected *read* using *QUAST*. Since the entire set of corrected *reads* is a large amount of data and takes a long time to be evaluated using *QUAST*, we only evaluated the first 2000 *reads* from each dataset with no specific order. Table 7 represents error rates in corrected *reads* and the correction execution time where *MiniC*.cr refers to the combination of the *Minimap2* and *Racon* correction methods, and *Canu*.cr refers to *reads* corrected by *Canu*. While *MiniC*.cr correction is faster, *Canu* result in a lower error rate in corrected *reads*.

The Effect of Overlap Finding Parameters

As discussed before, the number of *k-mers* to be indexed during the *long-read* overlap finding step affects the number of discovered overlaps as well as the execution time and peak memory usage. In order to show this effect, we changed the parameters of *Minimap2* and processed the same dataset. Table 4 shows the effect of varying k and w on the size of the index (peak memory usage), the number of overlaps found and the time it took to find overlaps. For this evaluation, the 40X coverage *long-read* data of AJ055 dataset was used.

long-read assembly

Table 5 compares the most recent *long-read* assembly pipelines which have not been considered in previous reviews. The *Mini* pipeline was considerably faster than *MiniC* and *Canu* as it did not include the most time-consuming step in *long-read* assembly (*read* error correction). However, the *Mini* pipeline achieved high genome coverage and its contiguity is similar to that of *MiniC*. The *Mini* pipeline mainly suffered from a high *base* calling error rate as well as misassemblies. The *MiniC* pipeline was slower than the *Mini* pipeline but led to a more accurate *base* calling in the assembled genome. *Canu* was the slowest *long-read* pipeline but resulted in the highest *base* calling accuracy as well as higher continuity. In addition, *Canu* requires more memory compared to the *Mini* pipeline. The memory usage of *MiniC* is smaller than that of *Mini*, which is possibly due to fewer false positive overlaps between corrected *long-reads* (the assembly graph would be smaller). Compared to the *short-read* assembler (*SGA* and *Spades*) as shown in Table 2, *long-read* assemblies have higher contiguity but at the same time much higher *base* calling error rates (for *Canu* only *Indels*).

Considering the depth of *read* coverage, the execution time and peak memory usage for all datasets and all programs were lower for 20X data compared to 40X data. *Canu* reached almost 100% genome coverage even with 20X data. Processing 40X data with *Canu* resulted in higher contiguity and fewer *Indel* errors in the assembly. For *Mini* and *MiniC*, using higher *read* coverage improved the genome quality in all aspects. The execution time of *Mini* and *MiniC* pipelines were 3.7 and 3.1 times faster (respectively) when processing 20X data compared to 40X data. This ratio is 1.4 for *Canu*, which indicates *Canu* is less sensitive to the *read* coverage. The memory usage of *Mini* and *MiniC* was almost twice as high for 40X data compared to 20X data. This relation is expected as the number of nodes in the overlap graph was equal to the number of reads. *Canu* memory usage showed less sensitivity to the *read* coverage.

The effect of genome size on execution time and memory usage of the *long-read* assembly pipeline is shown in Table 6. Apart from the *Mini* pipeline, which does not include read correction, the execution time of other pipelines almost linearly increased for the longer genome. The memory usage of *Canu* remained the same for the longer genome. The *Mini* and *MiniC* pipelines showed significant increases in the amount of memory used to process the longer genome.

Using a longer genome (SIM25M compared to AJ055) also affect the accuracy of the *long-read* only pipeline. Important changes are briefly highlighted here (for the complete evaluation metrics for SIM25M refer to `https://arashbayat.github.io/asm`). The genome coverage of *Canu* processing low coverage data declined from

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

Table 3: Comparing execution time of *short-read* assembly pipelines when processing different genome size

		SGA		*SOAPdenovo2*		*Spades*	
	Dataset	40X SR	20X SR	40X SR	20X SR	40X SR	20X SR
Execution Time (sec)	SIM25M	3698	1718	1856	1347	7775	5829
	AJ055	633	318	253	94	985	614
Ratio		*5.8*	*5.4*	*7.3*	*14.3*	*7.9*	*9.5*
Memory Usage (MB)	SIM25M	1905	1799	8474	6760	8977	6538
	AJ055	1376	717	5824	5125	2925	1859
Ratio		*1.4*	*2.5*	*1.5*	*1.3*	*3.1*	*3.5*

Table 4: The effect of k and w on overlap finding using *Minimap2*

k	w	Number of Overlaps	Execution Time (sec)	Memory (MB)
10	**10**	7750985	3453	10711
15	**10**	229058	13.38	630
20	**10**	181270	12.83	542
25	**10**	163344	12.07	532
20	**20**	162514	9.05	363
20	**30**	145905	9.33	337

Table 5: Comparison of *long-read* assembly pipeline for the assembly of the AJ055 genome.

Pipeline Name:	Mini		MiniC		Canu	
Read Coverage & Type:	20X LR	40X LR	20X LR	40X LR	20X LR	40X LR
Genome Coverage (%)	97.962	99.934	98.626	99.999	99.785	99.877
# Contigs	18	2	16	2	9	4
N50	455694	5534010	470381	5529667	3169885	5496970
NGA50	455694	5529303	470380	5529367	3169756	5496967
# Mismatches per 100 kbp	49.96	29.62	34.91	15.03	6.28	0.27
# Indels per 100 kbp	726.07	466.43	639.88	330.97	165.41	34.76
# Local Misassemblies	4	5	2	1	1	0
Execution Time (sec)	70	260	548	1800	4070	5808
Memory Usage (MB)	1026	2488	1019	2103	3656	4994

Table 6: The effect of genome size on execution time and memory usage of *long-read* assembly pipeline

		Mini		*MiniC*		*Canu*	
		20X LR	40X LR	20X LR	40X LR	20X LR	40X LR
Execution Time (sec)	SIM25M	269	561	3227	13984	18906	33873
	AJ055	70	260	548	1800	4070	5808
Ratio		*3.8*	*2.2*	*5.9*	*7.8*	*4.6*	*5.8*
Memory Usage (MB)	SIM25M	4265	7690	4233	7725	4052	5067
	AJ055	1026	2488	1019	2103	3656	4994
Ratio		*4.2*	*3.1*	*4.2*	*3.7*	*1.1*	*1.0*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

99.8% (for AJ055) to about 85.6% (for SIM25M). Processing low and high coverage data respectively, the N50 of *Canu* dropped from 3169885 and 5496970 (for AJ055) to 109830 and 556356 (for SIM25M). The N50 of *MiniC* pipeline drops from 5529667 (for AJ055) to 2347005 (for SIM25M) processing high coverage data and increased from 470381 (for AJ055) to 1073552 (for SIM25M) when processing low coverage data.

3.3 Hybrid assembly

long-read error correction using short-read

Table 7 compares the *Color-Map* hybrid correction method with other *long-read*-only correction methods. *Color-Map* was much better in correcting *Indels* mainly due to the low rate of *Indels* in high-confidence *short-reads*. However, the correction execution time was significantly larger than a complete *long-read*-only assembly pipeline. We also assembled a genome using *Canu* from corrected *reads* produced by *Color-Map*. In this case, we inputted corrected *reads* as *pacbio-corrected* to *Canu*, such that *Canu* did not execute the correction step for the input *reads*. As shown in Table 8, while the resulting genome has a lower *Indel* error rate compared to the *Canu* pipeline, it lacks contiguity and genome coverage possibly due to the timing applied by *Color-Map*.

Polishing long-read Assembly using short-read

We evaluated the effect of the *Pilon* genome polisher on the output of the *Canu* and the *Mini* pipelines. We also considered the evaluation of iterative genome polishing using *Pilon*. In Table 9 p1, p2 and p3 respectively refer to the first, second and third round of polishing with *Pilon*. The result shows that genome polishing with *Pilon* can improve *base* calling accuracy and reduce misassemblies. For the *Mini* pipeline, which has a higher *base* calling error rate, the second round of polishing further improved the genome quality. However, the third round of polishing seemed to be less effective and did not provide much improvement. This table provides execution times for one round of polishing with *Pilon* and does not include genome assembly time. Polishing high-quality assemblies such as *Canu* takes less time since *Pilon* exerts less effort in correcting genomes. The execution time of the second and third rounds of polishing of the *Mini* pipeline also suggest that the time taken for polishing depends on the quality of the input genome.

Other Hybrid Assembly Approach

Table 10 compares the *Spades* pipeline with and without using *long-reads*. *long-reads* are mainly used to scaffold contigs from a *short-read* assembly. The use of *long-read* in the *Spades* pipeline significantly increased contig size while having no effect on the error rate. However, integration of *long-read* in the *Spades* pipeline increased execution time. Using *long-read* also reduced the number of misassemblies.

We used *Metassembler* to merge three of the best assemblies we obtained, as shown in Table 11. In our evaluation, *Metassembler* does not lead to an improvement and its output is as accurate as one of the input assemblies (the one obtained with *Canu*). We did not test *Metassembler* with low-quality assemblies or large than tree assemblies. Perhaps *Metassembler* better fit such cases.

4 CONCLUSION

As mentioned in Section 1 and based on the results provided in this review, it is difficult to name a pipeline that can be considered best for all sort and volume of data and to improve all aspect of accuracy and performance. Based on the type and volume of data as well as the purpose of assembly, one should choose the pipeline that best suits the application. This review provides guidelines that can be summarised as follows:

- If high coverage *long-read* data are available, assembling genomes using *long-read* pipelines seems to be a reasonable method. Polishing genomes once or twice with *short-read* additionally improves the quality.
- If low coverage *long-reads* are available, they can be used for the *Spades* hybrid pipeline to improve contig length in the *short-read* assembly.
- If only *short-reads* are available, *Spades* results in higher genome coverage and contiguity while *SGA* results in an accurate *base* calling.
- In case speed is matter one can use *SOAPdenovo2* for *short-reads* and *Mini* for *long-reads*. Also, the correction of *long-read* using *short-read* is not recommended as it is quite time-consuming.

5 ACKNOWLEDGEMENTS

Marc R Wilkins acknowledges funding from Bioplatforms Australia under the federal government NCRIS scheme. Marc R Wilkins also acknowledges funding from the New South Wales State Government RAAP scheme.

References

[1] Kin Fai Au, Jason G Underwood, Lawrence Lee, and Wing Hung Wong. Improving pacbio long read accuracy by short read alignment. *PloS one*, 2012.

[2] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 2012.

[3] Ergude Bao and Lingxiao Lan. Halc: High throughput algorithm for long read error correction. *BMC bioinformatics*, 2017.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 June 2020 doi:10.20944/preprints202006.0324.v1

Table 7: Comparison of correction algorithm for the assembly of the AJ055 genome.

Pipeline Name: Read Coverage & Type:	MiniC.cr 20X LR	MiniC.cr 40X LR	Canu.cr 20X LR	Canu.cr 40X LR	Colormap.cr 20X LR + 40X SR
# Mismatches per 100 kbp	134.01	109.27	95.67	49.78	90.27
# Indels per 100 kbp	1254.47	1032.45	705.65	371.26	98.9
Execution Time (sec)	402	1630	1804	3218	21239

Table 8: Comparison of assembly with hybrid and non-hybrid correction for the assembly of the AJ055 genome.

Pipeline Name: Read Coverage & Type:	**Canu 20X LR**	**Colormap 20X LR + 40X SR**
Genome Coverage (%)	99.785	97.625
# Contigs	9	73
N50	3169885	218907
NGA50	3169756	305685
# Mismatches per 100 kbp	6.28	24.7
# Indels per 100 kbp	165.41	8.04
# Local Misassemblies	1	12
Execution Time (sec)	4070	21279
Memory Usage (MB)	3656	5649

Table 9: The effect of (iterative) polishing for the assembly of the AJ055 genome.

Pipeline Name: Read Coverage & Type:	Canu 40X LR	Canu.p1 40X LR + 40X SR	Mini 40X LR	Mini.p1 40X LR + 40X SR	Mini.p2 40X LR + 40X SR	Mini.p3 40X LR + 40X SR
Genome Coverage (%)	99.877	99.877	99.934	99.939	99.939	99.939
# Contigs	4	4	2	2	2	2
N50	5496970	5497403	5534010	5505273	5505584	5505476
NGA50	5496967	5497400	5529303	5500863	5501187	5501107
# Mismatches per 100 kbp	0.27	0	29.62	1.04	0.69	0.8
# Indels per 100 kbp	34.76	2.58	466.43	55.57	8.05	7.82
# Local Misassemblies	0	0	5	2	2	1
Execution Time (sec)	5808	414	260	2506	2301	1215
Memory Usage (MB)	4944	9206	2488	6874	6350	5859

Table 10: Comparison of *Spades* with and without *long-reads* for the assembly of the AJ055 genome.

Pipeline Name: Read Coverage & Type:	**Spades 40X SR**	**SpadesHybrid 40X SR + 20X LR**
Genome Coverage (%)	99.131	99.882
# Contigs	64	16
N50	226024	1571477
NGA50	226024	1571477
# Mismatches per 100 kbp	5.22	5.57
# Indels per 100 kbp	1.45	1.75
# Local Misassemblies	11	1
Execution Time (sec)	1074	2033
Memory Usage (MB)	2925	2925

Table 11: Merging three high quality genome with *Metassembler* for the assembly of the AJ055 genome.

Pipeline Name: Read Coverage & Type:	SGA 40X SR	Canu 40X LR	SpadesHybrid 40X SR + 20X LR	Metassembler 40X SR + 40X LR
Genome Coverage (%)	97.757	99.877	99.882	99.877
# Contigs	1370	4	16	4
N50	6394	5496970	1571477	5496970
NGA50	6296	5496967	1571477	5496967
# Mismatches per 100 kbp	0.06	0.27	5.57	0.27
# Indels per 100 kbp	0.59	34.76	1.75	34.76
# Local Misassemblies	0	0	1	0
Execution Time (sec)	663	5808	2033	20535
Memory Usage (MB)	1376	4944	2925	394

[4] David R Bentley, Shankar Balasubramanian, Harold P Swerdlow, Geoffrey P Smith, John Milton, Clive G Brown, Kevin P Hall, Dirk J Evers, Colin L Barnes, Helen R Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *nature*, 2008.

[5] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology*, 2015.

[6] bioplatforms. Antibiotic resistant sepsis pathogens. *data.bioplatforms.com/organization/about/bpa-sepsis*, 2018.

[7] Marten Boetzer and Walter Pirovano. Toward almost closed genomes with gapfiller. *Genome biology*, 2012.

[8] Marten Boetzer and Walter Pirovano. Sspace-longread: scaffolding bacterial draft genomes using long read sequence information. *BMC bioinformatics*, 2014.

[9] Keith R Bradnam, Joseph N Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inanç Birol, Sébastien Boisvert, Jarrod A Chapman, Guillaume Chapuis, Rayan Chikhi, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2013.

[10] Andrei Z Broder. On the resemblance and containment of documents. *Compression and Complexity of Sequences 1997. Proceedings*, 1997.

[11] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, et al. Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data. *Nature methods*, 2013.

[12] Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, Maria Nattestad, Gregory T Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, et al. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature methods*, 2016.

[13] Gennady Denisov, Brian Walenz, Aaron L Halpern, Jason Miller, Nelson Axelrod, Samuel Levy, and Granger Sutton. Consensus generation and variant detection by celera assembler. *Bioinformatics*, 2008.

[14] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, et al. Real-time dna sequencing from single polymerase molecules. *Science*, 2009.

[15] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, 2001.

[16] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 2013.

[17] Ehsan Haghshenas, Faraz Hach, S Cenk Sahinalp, and Cedric Chauve. Colormap: Correcting long reads by mapping short reads. *Bioinformatics*, 2016.

[18] Ruifeng Hu, Guibo Sun, and Xiaobo Sun. Lscplus: a fast solution for improving long read accuracy by short read alignment. *BMC bioinformatics*, 2016.

[19] Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth. ART: a next-generation sequencing read simulator. *Bioinformatics (Oxford, England)*, 2012.

[20] Miten Jain, Hugh E Olsen, Benedict Paten, and Mark Akeson. The oxford nanopore minion: delivery of nanopore sequencing to the genomics community. *Genome biology*, 2016.

[21] Juhana I Kammonen, Olli-Pekka Smolander, Lars Paulin, Pedro AB Pereira, Pia Laine, Patrik Koskinen, Jukka Jernvall, and Petri Auvinen. gapfinisher: a reliable gap filling pipeline for sspace-longread scaffolder output. *PeerJ PrePrints*, 2017.

[22] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 2010.

[23] Jan O Korbel, Alexander Eckehart Urban, Jason P Affourtit, Brian Godwin, Fabian Grubert, Jan Fredrik Simons, Philip M Kim, Dean Palejev, Nicholas J Carriero, Lei Du, et al. Paired-end mapping reveals extensive structural variation in the human genome. *Science*, 2007.

[24] 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 research*, 2017.

[25] Heng Li. Klib library. *attractivechaos.github.io/klib*, 2011.

[26] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[27] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 2016.

[28] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 2018.

[29] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, Bicheng Yang, and Wei Fan. Comparison of the two major classes of assembly algorithms: overlap layout consensus and de bruijn graph. *Briefings in Functional Genomics*, 2012.

[30] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, et al. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience*, 2012.

[31] Michael L Metzker. Sequencing technologiesthe next generation. *Nature reviews genetics*, 2010.

[32] Jason R Miller, Arthur L Delcher, Sergey Koren, Eli Venter, Brian P Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 2008.

[33] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1986.

[34] Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 2005.

[35] Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 2000.

[36] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 1999.

[37] Giuseppe Narzisi and Bud Mishra. Comparing de novo genome assembly: the long and short of it. *PloS one*, 2011.

[38] Vicki Pandey, Robert C Nutter, and Ellen Prediger. Applied biosystems solid system: ligation-based sequencing. *Next Generation Genome Sequencing: Towards Personalized Medicine*, 2008.

[39] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 2001.

[40] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 2004.

[41] Leena Salmela and Eric Rivals. Lordec: accurate and efficient long read error correction. *Bioinformatics*, 2014.

[42] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 2010.

[43] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 2012.

[44] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 2012.

[45] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 1981.

[46] Jang-il Sohn and Jin-Wu Nam. The present and future of de novo whole-genome assembly. *Briefings in bioinformatics*, 2016.

[47] Martin Sosic and Mile Sikic. Edlib: A C/C++ library for fast, exact sequence alignment using edit distance. *bioRxiv*, 2016.

[48] Bianca K Stöcker, Johannes Köster, and Sven Rahmann. Simlord: Simulation of long read data. *Bioinformatics*, 2016.

[49] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome research*, 2017.

[50] Sarah Vergult, Ellen Van Binsbergen, Tom Sante, Silke Nowak, Olivier Vanakker, Kathleen Claes, Bruce Poppe, Nathalie Van der Aa, Markus J Van Roosmalen, Karen Duran, et al. Mate pair sequencing for the detection of chromosomal aberrations in patients with intellectual disability and congenital malformations. *European Journal of Human Genetics*, 2014.

[51] Bruce J Walker, Thomas Abeel, Terrance Shea, Margaret Priest, Amr Abouelliel, Sharadha Sakthikumar, Christina A Cuomo, Qiandong Zeng, Jennifer Wortman, Sarah K Young, et al. Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PloS one*, 2014.

[52] Alejandro Hernandez Wences and Michael C Schatz. Metassembler: merging and optimizing de novo genome assemblies. *Genome biology*, 2015.

[53] Matei Zaharia, William J Bolosky, Kristal Curtis, Armando Fox, David Patterson, Scott Shenker, Ion Stoica, Richard M Karp, and Taylor Sittler. Faster and More Accurate Sequence Alignment with SNAP. *arXiv*, 2011.

[54] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 2008.

[55] Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The masurca genome assembler. *Bioinformatics*, 2013.