OXFORD

Briefings in Bioinformatics, 20(4), 2019, 1542–1559

doi: 10.1093/bib/bby017
Advance Access Publication Date: 2 April 2018
Review Article

Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions

Damla Senol Cali, Jeremie S. Kim, Saugata Ghose, Can Alkan and Onur Mutlu

Corresponding author: Can Alkan, Department of Computer Engineering, Bilkent University, Engineering Building, EA-509, Bilkent, 06800 Ankara, Turkey. Email: calkan@cs.bilkent.edu.tr; Onur Mutlu, Systems Group, Department of Computer Science (D-INFK), ETH Zürich, CAB F 74.2, Universitätstrasse 6, 8092 Zürich, Switzerland. Email: onur.mutlu@inf.ethz.ch

Abstract

Nanopore sequencing technology has the potential to render other sequencing technologies obsolete with its ability to generate long reads and provide portability. However, high error rates of the technology pose a challenge while generating accurate genome assemblies. The tools used for nanopore sequence analysis are of critical importance, as they should overcome the high error rates of the technology. Our goal in this work is to comprehensively analyze current publicly available tools for nanopore sequence analysis to understand their advantages, disadvantages and performance bottlenecks. It is important to understand where the current tools do not perform well to develop better tools. To this end, we (1) analyze the multiple steps and the associated tools in the genome assembly pipeline using nanopore sequence data, and (2) provide guidelines for determining the appropriate tools for each step. Based on our analyses, we make four key observations: (1) the choice of the tool for basecalling plays a critical role in overcoming the high error rates of nanopore sequencing technology. (2) Read-to-read overlap finding tools, GraphMap and Minimap, perform similarly in terms of accuracy. However, Minimap has a lower memory usage, and it is faster than GraphMap. (3) There is a trade-off between accuracy and performance when deciding on the appropriate tool for the assembly step. The fast but less accurate assembler Miniasm can be used for quick initial assembly, and further polishing can be applied on top of it to increase the accuracy, which leads to faster overall assembly. (4) The state-of-the-art polishing tool, Racon, generates high-quality consensus sequences while providing a significant speedup over another polishing tool, Nanopolish. We analyze various combinations of different tools and expose the trade-offs between accuracy, performance, memory usage and scalability. We conclude that our observations can guide researchers and practitioners in making conscious and effective choices for each step of the genome assembly pipeline using nanopore sequence data. Also, with the help of bottlenecks we have found, developers can improve the current tools or build new ones that are both accurate and fast, to overcome the high error rates of the nanopore sequencing technology.

Key words: nanopore sequencing; genome sequencing; genome analysis; assembly; mapping

Damla Senol Cali is a PhD student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. Her research interests are in computational methods for the analysis of NGS and nanopore sequencing data, and computer architecture.
Jeremie S. Kim is a PhD student in the Department of Electrical and Computer Engineering at Carnegie Mellon University and in the Department of Computer Science at ETH Zürich. His research interests are in computer architecture and hardware accelerators for bioinformatics applications.
Saugata Ghose is a Systems Scientist in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests are in several aspects of computer architecture, with a significant focus on designing architecture-aware and systems-aware memory and storage.
Can Alkan is an Assistant Professor in the Department of Computer Engineering at Bilkent University. His research interests are in combinatorial algorithms for bioinformatics and computational biology.
Onur Mutlu is a Professor in the Department of Computer Science at ETH Zürich. He is also an Adjunct Professor in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests are in computer architecture, systems, security and bioinformatics.

Submitted: 20 November 2017; **Received (in revised form):** 6 February 2018

Introduction

Next-generation sequencing (NGS) technologies have revolutionized and dominated the genome sequencing market since 2005, because of their ability to generate massive amounts of data at a faster speed and lower cost [1–3]. The existence of successful computational tools that can process and analyze such large amounts of data quickly and accurately is critically important to take advantage of NGS technologies in science, medicine and technology.

As the whole genome of most organisms cannot be sequenced all at once, the genome is broken into smaller fragments. After each fragment is sequenced, small pieces of DNA sequences (i.e. reads) are generated. These reads can then be analyzed following two different approaches: read mapping and *de novo* assembly. Read mapping is the process of aligning the reads against the reference genome to detect variations in the sequenced genome. *De novo* assembly is the method of combining the reads to construct the original sequence when a reference genome does not exist [4]. Owing to the repetitive regions in the genome, the short-read length of the most dominant NGS technologies (e.g. 100–150 bp reads) causes errors and ambiguities for read mapping [5, 6], and poses computational challenges and accuracy problems to *de novo* assembly [7]. Repetitive sequences are usually longer than the length of a short read, and an entire repetitive sequence cannot be spanned by a single short read. Thus, short reads lead to highly fragmented, incomplete assemblies [7–9]. However, a long read can span an entire repetitive sequence and enable continuous and complete assemblies. The demand for sequencing technologies that can produce longer reads has resulted in the emergence of even newer alternative sequencing technologies.

Nanopore sequencing technology [10] is one example of such technologies that can produce long read lengths. Nanopore sequencing is an emerging and a promising single-molecule DNA sequencing technology, which exhibits many attractive qualities, and in time, it could potentially surpass current sequencing technologies. Nanopore sequencing promises high sequencing throughput, low cost and longer read length, and it does not require an amplification step before the sequencing process [11–14].

Using biological nanopores for DNA sequencing was first proposed in the 1990s [15], but the first nanopore sequencing device, MinION [16], was only recently (in May 2014) made commercially available by Oxford Nanopore Technologies (ONT). MinION is an inexpensive, pocket-sized, portable, high-throughput sequencing apparatus that produces data in real time. These properties enable new potential applications of genome sequencing, such as rapid surveillance of Ebola, Zika or other epidemics [17], near-patient testing [18] and other applications that require real-time data analysis. In addition, the MinION technology has two major advantages. First, it is capable of generating ultra-long reads (e.g. 882 kilobase pairs or longer [19, 20]). MinION's long reads greatly simplify the genome assembly process by decreasing the computational requirements [8, 21]. Second, it is small and portable. MinION is named as the first DNA sequencing device used in outer space to help the detection of life elsewhere in the universe with the help of its size and portability [22]. With the help of continuous updates to the MinION device and the nanopore chemistry, the first nanopore human reference genome was generated by using only MinION devices [19].

Nanopores are suitable for sequencing because they:

- do not require any labeling of the DNA or nucleotide for detection during sequencing,
- rely on the electronic or chemical structure of the different nucleotides for identification,
- allow sequencing long reads and
- provide portability, low cost and high throughput.

Despite all these advantageous characteristics, nanopore sequencing has one major drawback: high error rates. In May 2016, ONT released a new version of MinION with a new nanopore chemistry called R9 [23], to provide higher accuracy and higher speed, which replaced the previous version R7. Although the R9 chemistry improves the data accuracy, the improvements are not enough for cutting-edge applications. Thus, nanopore sequence analysis tools have a critical role to overcome high error rates and to take better advantage of the technology. Also, faster tools are critically needed to (1) take better advantage of the real-time data production capability of MinION and (2) enable real-time data analysis.

Our goal in this work is to comprehensively analyze current publicly available tools for nanopore sequence analysis to understand their advantages, disadvantages and bottlenecks. It is important to understand where the current tools do not perform well, to develop better tools. To this end, we analyze the tools associated with the multiple steps in the genome assembly pipeline using nanopore sequence data in terms of accuracy, speed, memory efficiency and scalability.

We note that our manuscript presents a checkpoint of the state-of-the-art tools at the time the manuscript was submitted. This is a fast moving field, but we hope that our analysis is useful, and we expect that the fundamental conclusions and recommendations we make are independent of the exact versions of the tools.

Genome assembly pipeline using nanopore sequence data

We evaluate the genome assembly pipeline using nanopore sequence data. Figure 1 shows each step of the pipeline and lists the associated existing tools for each step that we analyze.

The output of MinION is raw signal data that represents changes in electric current when a DNA strand passes through nanopore. Thus, the pipeline starts with the raw signal data. The first step, basecalling, translates this raw signal output of MinION

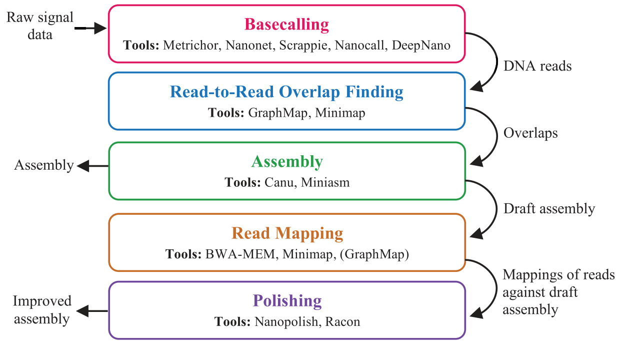

Figure 1. The analyzed genome assembly pipeline using nanopore sequence data, with its five steps and the associated tools for each step.

into bases (A, C, G, T) to generate DNA reads. The second step computes all pairwise read alignments or suffix–prefix matches between each pair of reads, called read-to-read overlaps. Overlap-layout-consensus (OLC) algorithms are used for the assembly of nanopore sequencing reads, as OLC-algorithms perform better with longer error-prone reads [24]. OLC-based assembly algorithms generate an overlap graph, where each node denotes a read, and each edge represents the suffix–prefix match between the corresponding two nodes. The third pipeline step, genome assembly, traverses this overlap graph, producing the layout of the reads and then constructing a draft assembly. To increase the accuracy of the assembly, further polishing, i.e. postassembly error correction, may be required. The fourth step of the pipeline is mapping the original basecalled reads to the generated draft assembly from the previous step (i.e. read mapping). The fifth and final step of the pipeline is polishing the assembly with the help of mappings from the previous step.

We next introduce the state-of-the-art tools used for each step.

Basecalling

When a strand of DNA passes through the nanopore (which is called the translocation of the strand through the nanopore), it causes drops in the electric current passing between the walls of the pore. The amount of change in the current depends on the type of base passing through the pore. Basecalling, the initial step of the entire pipeline, translates the raw signal output of the nanopore sequencer into bases (A, C, G, T) to generate DNA reads. Most of the current basecallers divide the raw current signal into discrete blocks, which are called events. After event-detection, each event is decoded into a most-likely set of bases. In the ideal case, each consecutive event should differ by one base. However, in practice, this is not the case because of the non-stable speed of the translocation. Also, determining the correct length of the homopolymers (i.e. repeating stretches of one kind of base, e.g. AAAAAAA) is challenging. Both of these problems make deletions the dominant error of nanopore sequencing [25, 26]. Thus, basecalling is the most important step of the pipeline that plays a critical role in decreasing the error rate.

We analyze five state-of-the-art basecalling tools in this article (Table 1). For a detailed comparison of these and other basecallers (including Albacore [32], which is not freely available, and Chiron [33]), we refer the reader to an ongoing basecaller comparison study [34]. Note that this ongoing study does not capture the accuracy and performance of the entire genome assembly pipeline using nanopore sequence data.

Metrichor

Metrichor [27] is ONT's cloud-based basecaller, and its source code is not publicly available. Before the R9 update, Metrichor was using hidden Markov models (HMMs) [35] for basecalling [23]. After the R9 update, it started using recurrent neural networks (RNN) [36, 37] for basecalling [23].

Nanonet

Nanonet [28] has also been developed by ONT, and it is available on Github [38]. As Metrichor requires an Internet connection and its source code is not available, Nanonet is an offline and open-source alternative for Metrichor. Nanonet is implemented in Python. It also uses RNN for basecalling [28]. The tool supports multithreading by sharing the computation needed to call each single read between concurrent threads. In other words, only one read is called at a time.

Scrappie

Scrappie [29] is the newest proprietary basecaller developed by ONT. It is named as the first basecaller that explicitly addresses basecalling errors in homopolymer regions. To determine the correct length of homopolymers, Scrappie performs transducer-based basecalling [25]. For versions R9.4 and R9.5, Scrappie can perform basecalling with the raw current signal, without requiring event detection. It is a C-based local basecaller and is still under development [25].

Nanocall

Nanocall [30] uses HMMs for basecalling, and it is independently developed by a research group. It was released before the R9 update when Metrichor was also using an HMM-based approach for basecalling, to provide the first offline and open-source alternative for Metrichor. However, after the R9 update, when Metrichor started to perform basecalling with a more powerful RNN-based approach, Nanocall's accuracy fell short of Metrichor's accuracy [39]. Thus, although Nanocall supports R9 and upper versions of nanopore data, its usefulness is limited [39]. Nanocall is a C++-based command-line tool. It supports multithreading where each thread performs basecalling for different groups of raw reads.

DeepNano

DeepNano [31] is also independently developed by a research group before the R9 update. It uses an RNN-based approach to perform basecalling. Thus, it is considered to be the first RNN-based basecaller. DeepNano is implemented in Python. It does not have multithreading support.

Read-to-read overlap finding

Previous genome assembly methods designed for accurate and short reads (i.e. de Bruijn graph approach [40, 41]) are not suitable for nanopore reads because of the high error rates of the current nanopore sequencing devices [9, 26, 42, 43]. Instead, OLC algorithms [44] are used for nanopore sequencing reads, as they perform better with longer, error-prone reads. OLC-based assembly algorithms start with finding the read-to-read overlaps, which is the second step of the pipeline. Read-to-read overlap is defined to be a common sequence between two reads [43]. GraphMap [45] and Minimap [46] are the commonly used state-of-the-art tools for this step (Table 2).

Table 1. State-of-the-art nanopore basecalling tools

Tool	Strategy	Multithreading support	Source	Reference
Metrichor	RNN	(Cloud-based)	https://metrichor.com/	[27]
Nanonet	RNN	With -jobs parameter	https://github.com/nanoporetech/nanonet	[28]
Scrappie	RNN	With export OMP_NUM_THREADS command	https://github.com/nanoporetech/scrappie	[29]
Nanocall	HMM	With –threads parameter	https://github.com/mateidavid/nanocall	[30]
DeepNano	RNN	No support; split data set and run it in parallel	https://bitbucket.org/vboza/deepnano	[31]

Table 2. State-of-the-art read-to-read overlap finding tools

Tool	Strategy	Multithreading support	Source	Reference
GraphMap	*k*-mer similarity	With –threads parameter	https://github.com/isovic/graphmap	[45]
Minimap	Minimizer similarity	With -t parameter	https://github.com/lh3/minimap	[46]

Note: Both GraphMap and Minimap also have read mapping functionality.

Table 3. State-of-the-art assembly tools

Tool	Strategy	Multithreading support	Source	Reference
Canu	OLC with error correction	Auto-configuration	https://github.com/marbl/canu	[48]
Miniasm	OLC without error correction	No support	https://github.com/lh3/miniasm	[46]

GraphMap

GraphMap first partitions the entire read data set into *k*-length substrings (i.e. *k*-mers), and then creates a hash table. GraphMap uses gapped *k*-mers, i.e. k-mers that can contain insertions or deletions (indels) [45, 47]. In the hash table, for each *k*-mer entry, three pieces of information are stored: (1) *k*-mer string, (2) the index of the read and (3) the position in the read where the corresponding *k*-mer comes from. GraphMap detects the overlaps by finding the *k*-mer similarity between any two given reads. Owing to this design, GraphMap is a highly sensitive and accurate tool for error-prone long reads. It is a command-line tool written in C++. GraphMap is used for both (1) read-to-read overlap finding with the graphmap owler command and (2) read mapping with the graphmap align command.

Minimap

Minimap also partitions the entire read data set into *k*-mers, but instead of creating a hash table for the full set of *k*-mers, it finds the minimum representative set of *k*-mers, called minimizers, and creates a hash table with only these minimizers. Minimap finds the overlaps between two reads by finding minimizer similarity. The goals of using minimizers are to (1) reduce the storage requirement of the tool by storing fewer *k*-mers and (2) accelerate the overlap finding process by reducing the search span. Minimap also sorts *k*-mers for cache efficiency. Minimap is fast and cache-efficient, and it does not lose any sensitivity by storing minimizers, as the chosen minimizers can represent the whole set of *k*-mers. Minimap is a command-line tool written in C. Like GraphMap, it can both (1) find overlaps between two read sets and (2) map a set of reads to a full genome.

Genome assembly

After finding the read-to-read overlaps, OLC-based assembly algorithms generate an overlap graph. Genome assembly is performed by traversing this graph, producing the layout of the reads and then constructing a draft assembly. Canu [48] and Miniasm [46] are the commonly used error-prone long-read assemblers (Table 3).

Canu

Canu performs error-correction as the initial step of its own pipeline. It finds the overlaps of the raw uncorrected reads and uses them for the error-correction. The purpose of error-correction is to improve the accuracy of the bases in the reads [48, 49]. After the error-correction step, Canu finds overlaps between corrected reads and constructs a draft assembly after an additional trimming step. However, error-correction is a computationally expensive step. In its own pipeline, Canu implements its own read-to-read overlap finding tool such that the users do not need to perform that step explicitly before running Canu. Most of the steps in the Canu pipeline are multi-threaded. Canu detects the resources that are available in the computer before starting its pipeline and automatically assigns number of threads, number of processes and amount of memory based on the available resources and the assembled genome's estimated size.

Miniasm

Miniasm skips the error-correction step, as it is computationally expensive. It constructs a draft assembly from the uncorrected read overlaps computed in the previous step. Although Miniasm lowers computational cost and thus accelerates and simplifies assembly by doing so, the accuracy of the draft assembly depends directly on the accuracy of the uncorrected basecalled reads. Thus, further polishing may be necessary for these draft assemblies. Miniasm does not support multithreading.

Read mapping and polishing

To increase the accuracy of the assembly, especially for the rapid assembly methods like Miniasm, which do not have the error-correction step, further polishing may be required. Polishing, i.e. postassembly error-correction, improves the accuracy of the draft assembly by mapping the reads to the assembly and changing the assembly to increase local similarity with the reads [26, 50, 51]. The first step of polishing is mapping the basecalled reads to the generated draft assembly from the previous step. One of the most commonly used long-read mappers for nanopore data is BWA-MEM [52]. Read-to-read overlap finding tools, GraphMap and Minimap ('Read-to-read overlap finding' section), can also be used for this step, as they also have a read mapping mode (Table 4).

After aligning the basecalled reads to the draft assembly, the final polishing of the assembly can be performed with Nanopolish [50] or Racon [51] (Table 5).

Nanopolish

Nanopolish uses the raw signal data of reads along with the mappings from the previous step to improve the assembly base quality by evaluating and maximizing the probabilities for each base with a HMM-based approach [50]. It can increase the accuracy of the draft assembly by correcting the homopolymer-rich parts of the genome. Although this approach can increase the

Table 4. State-of-the-art read mapping tools

Tool	Strategy	Multithreading support	Source	Reference
BWA-MEM	Burrows–Wheeler Transform	With -t parameter	http://bio-bwa.sourceforge.net	[52]
GraphMap	*k*-mer similarity	With –threads parameter	https://github.com/isovic/graphmap	[45]
Minimap	Minimizer similarity	With -t parameter	https://github.com/lh3/minimap	[46]

Table 5. State-of-the-art polishing tools

Tool	Strategy	Multithreading support	Source	Reference
Nanopolish	HMM	With –threads and -P parameters	https://github.com/jts/nanopolish	[50]
Racon	Partial order alignment graph	With –threads parameter	https://github.com/isovic/racon	[51]

Table 6. Specifications of evaluation systems

Name	Model	CPU specifications	Main memory specifications	NUMA* specifications
System 1	40-core Intel® Xeon® E5-2630 v4 CPU @ 2.20GHz	20 physical cores 2 threads per core 40 logical cores with hyper-threading**	128GB DDR4 2 channels, 2 ranks/channel Speed: 2400MHz	2 NUMA nodes, each with 10 physical cores, 64 GB of memory and an 25 MB of LLC
System 2 (desktop)	8-core Intel® Core i7-2600 CPU @ 3.40GHz	4 physical cores 2 threads per core 8 logical cores with hyper-threading**	16GB DDR3 2 channels, 2 ranks/channel Speed: 1333MHz	1 NUMA node, with 4 physical cores, 16 GB of memory and an 8 MB of LLC
System 3 (big-mem)	80-core Intel® Xeon® E7-4850 CPU @ 2.00GHz	40 physical cores 2 threads per core 80 logical cores with hyper-threading**	1TB DDR3 8 channels, 4 ranks/channel Speed: 1066MHz	4 NUMA nodes, each with 10 physical cores, 256 GB of memory and an 24 MB of LLC

*NUMA (Non-Uniform Memory Access) is a computer memory design, where a processor accesses its local memory faster (i.e. with lower latency) than a nonlocal memory (i.e. memory local to another processor in another NUMA node). A NUMA node is composed of the local memory and the CPU cores (see Observation 6 in Section 4.1 for detail).

**Hyper-threading is Intel's SMT implementation (See Observation 5 in Section 4.1 for detail).

accuracy significantly, it is computationally expensive, and thus time-consuming. Nanopolish developers recommend BWA-MEM as the read mapper before running Nanopolish [53].

Racon

Racon constructs partial order alignment graphs [51, 54] to find a consensus sequence between the reads and the draft assembly. After dividing the sequence into segments, Racon tries to find the best alignment to increase the accuracy of the draft assembly. Racon is a fast polishing tool, but it does not promise a high increase in accuracy as Nanopolish promises. However, multiple iterations of Racon runs or a combination of Racon and Nanopolish runs can improve accuracy significantly. Racon developers recommend Minimap as the read mapper to use before running Racon, as Minimap is both fast and sensitive [51].

Experimental methodology

Data set

In this work, we use *Escherichia coli* genome data as the test case, sequenced using the MinION with an R9 flowcell [55].

MinION sequencing has two types of workflows. In the 1D workflow, only the template strand of the double-stranded DNA is sequenced. In contrast, in the 2D workflow, with the help of a hairpin ligation, both the template and complement strands pass through the pore and are sequenced. After the release of R9 chemistry, 1D data became usable in contrast to previous chemistries. Thus, we perform the analysis of the tools on 1D data.

MinION outputs one file in the fast5 format for each read. The fast5 file format is a hierarchical data format, capable of storing both raw signal data and basecalled data returned by Metrichor. This data set includes 164 472 reads, i.e. fast5 files. As all these files include both raw signal data and basecalled reads, we can use this data set for both (1) using the local basecallers to convert raw signal data into the basecalled reads and (2) using the already basecalled reads by Metrichor.

Evaluation systems

In this work, for accuracy and performance evaluations of different tools, we use three separate systems with different specifications. We use the first computer in the first part of the analysis, accuracy analysis. We use the second and third computers in the second part of the analysis, performance analysis, to compare the scalability of the analyzed tools in the two machines with different specifications (Table 6).

We choose the first system for evaluation, as it has a larger memory capacity than a usual server, and with the help of a large number of cores, the tasks can be parallelized easily to get the output data quickly. We choose the second system, called

Table 7. Accuracy metrics

Metric name	Definition	Preferred values
Number of bases	Total number of bases in the assembly	≃ Length of reference genome
Number of contigs	Total number of segments in the assembly	Lower (≃1)
Average identity	Percentage similarity between the assembly and the reference genome	Higher (≃100%)
Coverage	Ratio of the number of aligned bases in the reference genome to the length of reference genome	Higher (≃100%)
Number of mismatches	Total number of single-base differences between the assembly and the reference genome	Lower (≃0)
Number of indels	Total number of insertions and deletions between the assembly and the reference genome	Lower (≃0)

Table 8. Performance metrics

Metric name	Definition	Preferred values
Wall clock time	Elapsed time from the start of a program to the end	Lower
CPU time	Total amount of time the CPU spends in user mode (i.e. to run the program's code) and kernel mode (i.e. to execute system calls made by the program)*	Lower
Peak memory usage	Maximum amount of memory used by a program during its whole lifetime	Lower
Parallel speedup	Ratio of the time to run a program with one thread to the time to run it with *N* threads	Higher

*If wall clock time < CPU time for a specific program, it means that the program runs in parallel.

desktop, as it represents a commonly used desktop server. We choose the third system, called *big-mem*, because of its large memory capacity. This big-mem system can be useful for those who would like to get results more quickly.

Accuracy metrics

We compare each draft assembly generated after the assembly step and each improved assembly generated after the polishing step with the reference genome, by using the dnadiff command under the MUMmer package [56]. We use six metrics to measure accuracy, as defined in Table 7: (1) number of bases in the assembly, (2) number of contigs, (3) average identity, (4) coverage, (5) number of mismatches and (6) number of indels.

Performance metrics

We analyze the performance of each tool by running the associated command-line of each tool with the/usr/bin/time -v command. We use four metrics to quantify performance as defined in Table 8: (1) wall clock time, (2) CPU time, (3) peak memory usage and (4) parallel speedup.

Results and analysis

In this section, we present our results obtained by analyzing the performance of different tools for each step in the genome assembly pipeline using nanopore sequence data in terms of accuracy and performance, using all the metrics we provide in Tables 7 and 8. Additionally, Table 9 shows the tool version, the executed command and the output of each analyzed tool. We divide our analysis into three main parts.

In the first part of our analysis, we examine the first three steps of the pipeline (cf. Figure 1). To this end, we first execute each basecalling tool (i.e. one of Nanonet, Scrappie, Nanocall or DeepNano). As Metrichor is a cloud-based tool and its source code is not available, we cannot execute Metrichor and get the performance metrics for it. After recording the performance metrics for each basecaller run, we execute either GraphMap or Minimap followed by Miniasm, or Canu itself, and record the performance metrics for each run. We obtain a draft assembly for each combination of these basecalling, read-to-read overlap finding and assembly tools. For each draft assembly, we assess its accuracy by comparing the resulting draft assembly with the existing reference genome. We show the accuracy results in Table 10. We show the performance results in Table 11. We will refer to these tables in sections 'Basecalling tools', 'Read-to-read overlap finding tools' and 'Assembly tools'.

In the second part of our analysis, we examine the last two steps of the pipeline (cf. Figure 1). To this end, for each obtained draft assembly, we execute each possible combination of different read mappers (i.e. BWA-MEM or Minimap) and different polishers (i.e. Nanopolish or Racon), and record the performance metrics for each step (i.e. read mapping and polishing). We obtain a polished assembly after each run, and assess its accuracy by comparing it with the existing reference genome. For these two analyses, we use the first system, which has 40 logical cores, and execute each tool using 40 threads, which is the possible maximum number of threads for that particular machine. We show the accuracy results in Table 12. We show the performance results in Table 13. We will refer to these tables in section 'Read mapping and polishing tools'.

In the third part of our analysis, we assess the scalability of all of the tools that have multithreading support. For this purpose, we use the second and third systems to compare the scalability of these tools on two different system configurations. For each tool, we change the number of threads and observe the corresponding change in speed, memory usage and parallel speedup. These results are depicted in Figures 2–6, and we will refer to them throughout 'Basecalling tools', 'Read-to-read overlap finding tools', 'Assembly tools' and 'Read mapping and polishing tools' sections.

'Basecalling tools', 'Read-to-read overlap finding tools', 'Assembly tools' and 'Read mapping and polishing tools' sections describe the major observations we make for each of the five steps of the pipeline (cf. Figure 1) based on our extensive evaluation results.

Table 9. Versions, commands to execute and outputs for each analyzed tool

	Command*	Output
Basecalling tools		
Nanonet–v2.0	nanonetcall fast5_dir/–jobs N –chemistry r9	reads.fasta
Scrappie–v1.0.1	(1) export OMP_NUM_THREADS=N (2) scrappie events –segmentation Segment_Linear: split_hairpin (2) fast5_dir/...	reads.fasta
Nanocall–v0.7.4	nanocall -t N fast5_dir/	reads.fasta
DeepNano-e8a621e	python basecall.py –directory fast5_dir/–chemistry r9	reads.fasta
Read-to-read overlap finding tools		
GraphMap–v0.5.2	graphmap owler -L paf -t N -r reads.fasta -d reads.fasta	overlaps.paf
Minimap–v0.2	minimap -Sw5 -L100 -m0 -tN reads.fasta reads.fasta	overlaps.paf
Assembly finding tools		
Canu–v1.6	canu -p ecoli -d canu-ecoli genomeSize=4.6m -nanopore-raw reads.fasta	draftAssembly.fasta
Miniasm–v0.2	miniasm -f reads.fasta overlaps.paf	draftAssembly.gfa –> draftAssembly.fasta
Read mapping tools		
BWA-MEM–0.7.15	(1) bwa index draftAssembly.fasta (2) bwa mem -x ont2d -t N draftAssembly.fasta reads.fasta	mappings.sam –> mappings.bam
Minimap–v0.2	minimap -tN draftAssembly.fasta reads.fasta	mappings.paf
Polishing tools		
Nanopolish–v0.7.1	(1) python nanopolish_makerange.py draftAssembly.fasta—parallel -P M (2) nanopolish variants –consensus polished.1.fa -w 1 (2) -r reads.fasta -b mappings.bam -g draftAssembly.fasta -t N (3) python nanopolish_merge.py polished.*.fa	polished.fasta
Racon–v0.5.0	racon (–sam) –bq -1 -t N reads.fastq mappings.paf/(mappings.sam) draftAssembly.fasta	polished.fasta

*N corresponds to the number of threads and M corresponds to the number of parallel jobs.

Basecalling tools

As we discuss in section 'Basecalling', ONT's basecallers Metrichor, Nanonet and Scrappie, and another basecaller developed by Boza *et al.* (2017), DeepNano, use RNNs for basecalling, whereas Nanocall developed by David *et al.* (2016) uses HMMs for basecalling.

Accuracy

Using RNNs is a more powerful basecalling approach than using HMMs, as an RNN 1) does not make any assumptions about sequence length [57] and (2) is not affected by the repeats in the sequence [30, 31, 57]. However, it is still challenging to determine the correct length of the homopolymers even with an RNN.

To compare the accuracy of the analyzed basecallers, we group the accuracy results by each basecalling tool and compare them according to the defined accuracy metrics.

According to this analysis and the accuracy results shown in Table 10, we make the following key observation.

Observation 1: The pipelines that start with Metrichor, Nanonet or Scrappie as the basecaller have similar identity and coverage trends among all of the evaluated scenarios (i.e. tool combinations for the first three steps), but Scrappie has a lower number of mismatches and indels. However, Nanocall and DeepNano cannot reach these three basecallers' accuracies: they have lower identity and lower coverage.

As Nanonet is the local version of Metrichor, Nanonet and Metrichor's similar accuracy trends are expected. In addition to the power of the RNN-based approach, Scrappie tries to solve the homopolymer basecalling problem. Although Scrappie is in an early stage of development, it leads to a smaller number of indels than Metrichor or Nanonet. Nanocall's poor accuracy results are because of the simple HMM-based approach it uses. Although DeepNano performs better than Nanocall with the help of its RNN-based approach, it results in a higher number of indels and a lower coverage of the reference genome.

Performance

RNN and HMM are computationally intensive algorithms. In HMM-based basecalling, the Viterbi algorithm [58] is used for decoding. The Viterbi algorithm is a sequential technique, and its computation cannot currently be parallelized with multithreading. However, in RNN-based basecalling, multiple threads can work on different sections of the neural network, and thus, RNN computation can be parallelized with multithreading.

To measure and compare the performance of the selected basecallers, we first compare the recorded wall clock time, CPU time and memory usage metrics of each scenario for the first step of the pipeline. Based on the results provided in Table 11, we make the following key observation.

Observation 2: RNN-based Nanonet and DeepNano are 2.6x and 2.3x faster than HMM-based Nanocall, respectively. Although Scrappie is also an RNN-based tool, it is 5.7x faster than Nanonet because of its C implementation as opposed to Nanonet's Python implementation.

Table 10. Accuracy analysis results using different tools for the first three steps of the pipeline

		Number of Bases	Number of Contigs	Identity (%)	Coverage (%)	Number of Mismatches	Number of Indels
1	Metrichor + — + Canu	4,609,499	1	98.05	99.92	12,378	76,990
2	Metrichor + Minimap + Miniasm	4,402,675	1	87.71	94.85	249,096	372,704
3	Metrichor + GraphMap + Miniasm	4,500,155	2	86.22	96.95	237,747	360,199
4	Nanonet + — + Canu	4,581,728	1	97.92	99.97	11,971	83,248
5	Nanonet + Minimap + Miniasm	4,350,175	1	85.50	92.76	237,518	394,852
6	Nanonet + GraphMap + Miniasm	4,272,545	1	85.36	91.16	232,748	389,968
7	Scrappie + — + Canu	4,614,149	1	98.46	99.90	6,777	63,597
8	Scrappie + Minimap + Miniasm	4,877,399	8	86.94	90.04	184,669	363,025
9	Scrappie + GraphMap + Miniasm	4,368,417	1	86.78	89.86	189,192	372,245
10	Nanocall + — + Canu	1,299,808	86	93.33	28.93	21,985	61,217
11	Nanocall + Minimap + Miniasm	4,492,964	5	80.52	42.92	177,589	221,092
12	Nanocall + GraphMap + Miniasm	4,429,390	3	80.51	41.32	171,455	213,435
13	DeepNano + — + Canu	7,151,596	106	92.75	99.16	38,803	211,551
14	DeepNano + Minimap + Miniasm	4,252,525	1	82.38	65.00	199,122	335,761
15	DeepNano + GraphMap + Miniasm	4,251,548	1	82.39	64.92	197,914	335,064

Table 11. Performance analysis results for the first three steps of the pipeline

		Step 1: Basecaller			Step 2: Read-to-Read Overlap Finder			Step 3: Assembly		
		Wall Clock Time (h:m:s)	CPU Time (h:m:s)	Memory Usage (GB)	Wall Clock Time (h:m:s)	CPU Time (h:m:s)	Memory Usage (GB)	Wall Clock Time (h:m:s)	CPU Time (h:m:s)	Memory Usage (GB)
1	Metrichor + — + Canu	—*	—*	—*	—	—	—	44:12:31	502:18:56	5.76
2	Metrichor + Minimap + Miniasm				2:15	41:37	12.30	1:09	1:09	1.96
3	Metrichor + GraphMap + Miniasm				6:14	1:52:57	56.58	1:05	1:05	1.82
4	Nanonet + — + Canu	17:52:42	714:21:45	1.89	—	—	—	11:32:40	129:07:16	5.27
5	Nanonet + Minimap + Miniasm				1:13	18:55	9.45	33	33	0.69
6	Nanonet + GraphMap + Miniasm				3:18	48:27	29.16	32	32	0.65
7	Scrappie + — + Canu	3:11:41	126:19:06	13.36	—	—	—	33:47:41	385:51:23	5.75
8	Scrappie + Minimap + Miniasm				2:52	1:10:26	12.40	1:29	1:29	1.98
9	Scrappie + GraphMap + Miniasm				7:26	2:16:02	38.31	1:23	1:23	1.87
10	Nanocall + — + Canu	47:04:53	1857:37:56	37.73	—	—	—	1:35:23	27:58:29	3.77
11	Nanocall + Minimap + Miniasm				1:15	16:08	12.19	20	20	0.47
12	Nanocall + GraphMap + Miniasm				5:14	1:09:04	56.78	16	16	0.30
13	DeepNano + — + Canu	23:54:34	811:14:29	8.38	—	—	—	1:15:48	17:31:07	3.61
14	DeepNano + Minimap + Miniasm				1:50	24:30	11.71	1:03	1:03	1.31
15	DeepNano + GraphMap + Miniasm				5:18	1:17:06	54.64	58	58	1.10

*We cannot get the performance metrics for Metrichor, as its source code is not available for us to run the tool by ourselves.

For a deeper understanding of these tools' advantages, disadvantages and bottlenecks, we also perform a scalability analysis for each basecaller by running it on the desktop server and the big-mem server separately, with 1, 2, 4, 8 (maximum for the desktop server), 16, 32, 40, 64 and 80 (maximum for the big-mem server) threads, and measuring the performance metrics for each configuration. Metrichor and DeepNano are not included in this analysis because Metrichor is a cloud-based tool and its source code is not available for us to change its number of threads, and DeepNano does not support multithreading. Figure 2 shows the speed, memory usage and parallel speedup results of our evaluations. We make four observations.

Observation 3: When we compare desktop's and big-mem's single-thread performance, we observe that desktop is approximately 2x faster than big-mem (cf. Figure 2A and B). This is mainly because of desktop's higher CPU frequency (see Table 6). It is an indication that all of these three tools are computationally expensive. Larger memory capacity or larger Last-Level Cache (LLC) capacity of big-mem cannot make up for the higher CPU speed in desktop when there is only one thread.

Observation 4: Scrappie and Nanocall have a linear increase in memory usage when number of threads increases. In contrast, Nanonet has a constant memory usage for all evaluated thread units (cf. Figure 2C and D).

In Scrappie and Nanocall, each thread performs the basecalling for different groups of raw reads. Thus, each thread allocates its own memory space for the corresponding data. This causes the linear increase in memory usage when the level of parallelism increases. In Nanonet, all of the threads share the computation of each read, and thus, memory usage is not affected by the amount of thread parallelism.

Observation 5: When the number of threads exceeds the number of physical cores, the simultaneous multithreading (SMT) overhead prevents continued linear speedup of Nanonet, Scrappie and Nanocall (cf. Figure 2E and F).

SMT (i.e. running more than one thread per physical core [59–66]), or more specifically Intel's hyper-threading (i.e. as we use Intel's hyper-threading enabled machines (see Table 6)) helps to decrease the total runtime, but it does not provide a linear speedup with the number of threads because of the CPU-intensive workload of Scrappie, Nanocall and Nanonet. If the threads executed are CPU-bound and do not wait for the memory or I/O requests, hyper-threading does not provide linear speedup because of the contention it causes in the shared resources for the computation. This phenomenon has been analyzed extensively in other application domains [59–61].

Observation 6: Data sharing between threads degrades the parallel speedup of Nanonet when cores from multiple NUMA nodes take role in the computation (cf. Figure 2F).

In Nanonet, data are shared between threads, and each thread performs different computations on the same data. There are four NUMA nodes in big-mem (cf. Table 6), and when data are shared between multiple NUMA nodes, this negatively affects the speedup of Nanonet because accessing the data located in another node (i.e. non-local memory) require longer latency than accessing the data located in local memory. When multiple NUMA nodes start taking role in the computation, Nanocall performs better in terms of scalability, as it does not require data sharing between different threads.

Summary. Based on the observations we make about the analyzed basecalling tools, we conclude that the choice of the tool for this step plays an important role to overcome the high error rates of nanopore sequencing technology. Basecalling with RNNs (e.g. Metrichor, Nanonet, Scrappie) provides higher accuracy and higher speed than basecalling with HMMs, and the newest basecaller of ONT, Scrappie, also has the potential to overcome the homopolymer basecalling problem.

Read-to-read overlap finding tools

As we discuss in 'Read-to-read overlap finding' section, GraphMap and Minimap are the commonly used tools for this step. GraphMap finds the overlaps using *k*-mer similarity, whereas Minimap finds them by using minimizers instead of the full set of *k*-mers.

Accuracy

As done in GraphMap, finding the overlaps with the help of full set of *k*-mers is a highly sensitive and accurate approach. However, it is also resource-intensive. For this reason, instead of the full set of *k*-mers, Minimap uses a minimum representative set of *k*-mers, which are called minimizers, as an alternative approach for finding the overlaps.

To compare the accuracy of these two approaches, we categorize the results in Table 10 based on read-to-read overlap

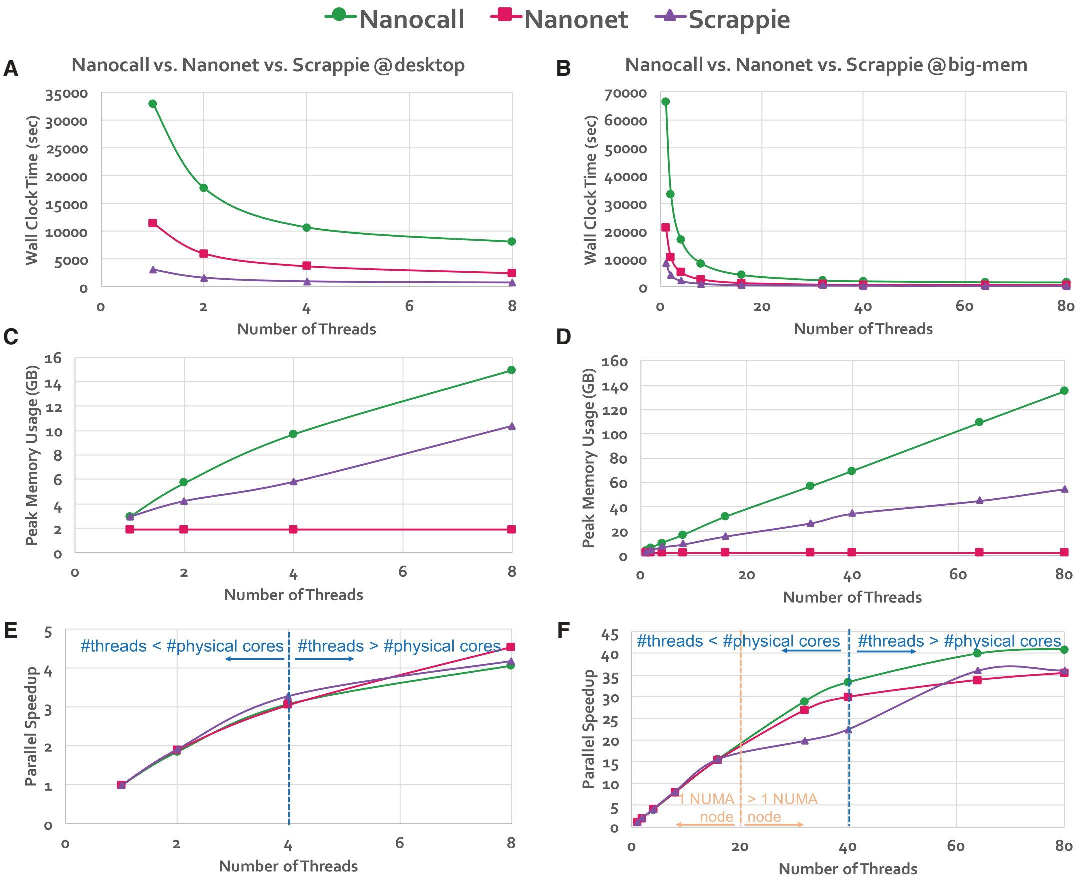

Figure 2. Scalability results of Nanocall, Nanonet and Scrappie. Wall clock time (**A**, **B**), peak memory usage (**C**, **D**) and parallel speedup (**E**, **F**) results obtained on the desktop and big-mem systems. The left column (**A**, **C**, **E**) shows the results from the desktop system, and the right column (**B**, **D**, **F**) shows the results from the big-mem system.

finding tools. In other words, we look at the rows with the same basecaller (i.e. red-labeled tools) and same assembler (i.e. green-labeled tools) but different read-to-read overlap finder (i.e. blue-labeled tools). After that, we compare them according to the defined accuracy metrics. We make the following major observation.

Observation 7: Pipelines with GraphMap or Minimap end up with similar values for identity, coverage, number of indels and mismatches. Thus, either of these read-to-read overlap finding tools can be used in the second step of the nanopore sequencing assembly pipeline to achieve similar accuracy.

Minimap and GraphMap do not have a significantly different effect on the accuracy of the generated draft assemblies. This is because Minimap does not lose any sensitivity by storing minimizers instead of the full set of *k*-mers.

Performance

To compare the performance of GraphMap and Minimap, we categorize the results in Table 11 based on read-to-read overlap finding tools, in a similar way we describe the results in Table 10 for the accuracy analysis. We also perform a scalability analysis for each of these tools by running them on the big-mem server with 1, 2, 4, 8, 16, 32, 40, 64 and 80 threads, and measuring the performance metrics. Because of the high memory usage of GraphMap, data necessary for the tool do not fit in the memory of the desktop server, and the GraphMap job exits because of a bad memory allocation exception. Thus, we could not perform the scalability analysis of GraphMap in the desktop server.

Figure 3 depicts the speed, memory usage and parallel speedup results of the scalability analysis for GraphMap and Minimap. We make the following three observations according to the results from Table 11 and Figure 3.

Observation 8: The memory usage of both GraphMap and Minimap is dependent on the hash table size but independent of number of threads. Minimap requires 4.6x less memory than GraphMap, on average.

This is mainly because Minimap stores only minimizers instead of all *k*-mers. Storing the full set of *k*-mers in GraphMap requires a larger hash table, and thus higher memory usage than Minimap. The high amount of memory requirement causes GraphMap to not run on our desktop system for none of the selected number of thread units.

Observation 9: Minimap is 2.5x faster than GraphMap, on average, across different scenarios in Table 11.

As GraphMap stores all *k*-mers, GraphMap needs to scan its large data set while finding the overlaps between two reads. However, in Minimap, the size of data set that needs to be scanned is greatly shrunk by storing minimizers, as we describe in Observation 8. Thus, Minimap performs much less computation, leading to its 2.5x speedup. Another indication of the different memory usage and its effect on the speed of computation is the LLC miss rates of these two tools. The LLC miss rate of Minimap is 36%, whereas the LLC miss rate of GraphMap is 55%. As the size of data needed by GraphMap is much larger than the LLC size, GraphMap experiences LLC misses more frequently. As a result, GraphMap stalls for longer, waiting for data accesses from main memory, which negatively affects the speed of the tool.

Observation 10: Minimap is more scalable than GraphMap. However, after 32 threads, there is a decrease in the parallel speedup of Minimap (cf. Figure 3C).

Because of its lower computational workload and lower memory usage, we find that Minimap is more scalable than GraphMap. However, in Minimap, threads that finish their work wait for the other active threads to finish their workloads, before starting new work, to prevent higher memory usage. Because of this, when the number of threads reaches a high number (i.e. 32 in Figure 3C), synchronization overhead greatly increases, causing the parallel speedup to reduce. GraphMap does not suffer from such a synchronization bottleneck and hence does not experience a decrease in speedup. However, GraphMap's speedup saturates when the number of threads reaches a high number because of data sharing between threads.

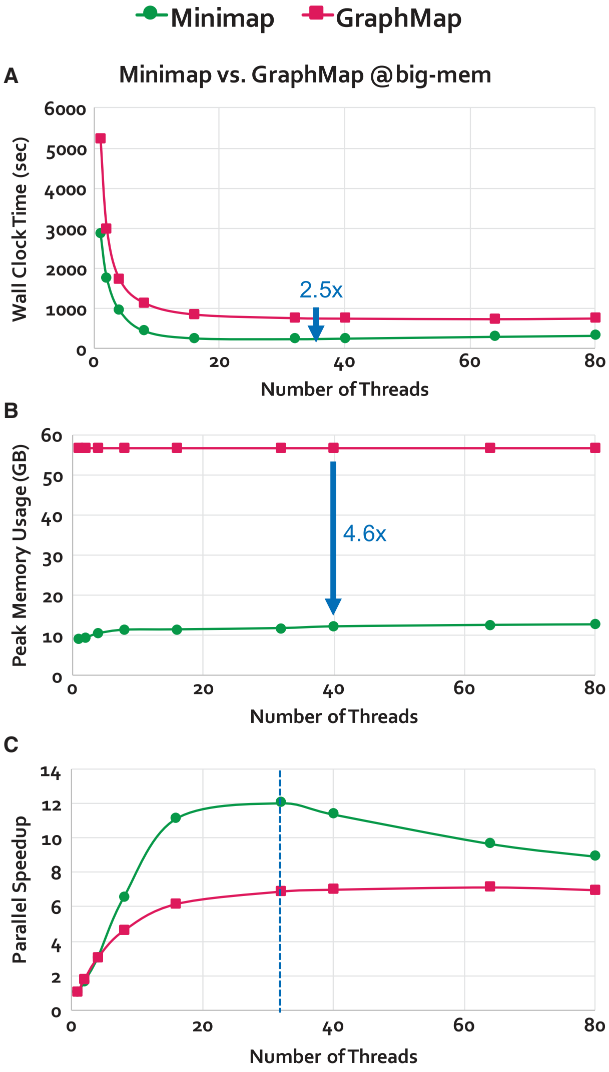

Figure 3. Scalability results of Minimap and GraphMap. Wall clock time (A), peak memory usage (B) and parallel speedup (C) results obtained on the big-mem system.

Summary. According to the observations we make about GraphMap and Minimap, we conclude that storing minimizers instead of all *k*-mers, as done by Minimap, does not affect the overall accuracy of the first three steps of the pipeline. Moreover, by storing minimizers, Minimap has a much lower memory usage and thus much higher performance than GraphMap.

Assembly tools

As we discuss in section 'Genome assembly', Canu and Miniasm are the commonly used tools for this step (In addition, we attempted to evaluate MECAT [67], another assembler. We were unable to draw any meaningful conclusions from MECAT, as its memory usage exceeds the 1 TB available in our big-mem system.).

Accuracy

To compare the accuracy of these two tools, we categorize the results in Table 10 based on assembly tools. We make the following observation.

Observation 11: Canu provides higher accuracy than Miniasm, with the help of the error-correction step that is present in its own pipeline.

Performance

To compare the performance of Canu and Miniasm, we categorize the results in Table 11 based on assembly tools, in a way similar to what we did in Table 10 for the accuracy analysis. We could not perform a scalability analysis for these tools, as Canu has an auto-configuration mechanism for each sub-step of its own pipeline, which does not allow us to change the number of threads, and Miniasm does not support multithreading. We make the following observation according to the results in Table 11.

Observation 12: Canu is much more computationally intensive and greatly (i.e. by 1096.3x) slower than Miniasm because of its expensive error-correction step.

Summary. According to the observations we make about Canu and Miniasm, there is a trade-off between accuracy and performance when deciding on the appropriate tool for this step. Canu produces highly accurate assemblies, but it is resource intensive and slow. In contrast, Miniasm is a fast assembler, but it cannot produce as accurate draft assemblies as Canu. We suggest that Miniasm can potentially be used for fast initial analysis and then further polishing can be applied in the next step to produce higher-quality assemblies.

Read mapping and polishing tools

As we discuss in section 'Read mapping and polishing', further polishing may be required for improving the accuracy of the low-quality draft assemblies. For this purpose, after aligning the reads to the generated draft assembly with BWA-MEM or Minimap (We do not discuss these tools in great detail here, as they perform read mapping, which is commonly analyzed and relatively well understood [68–86]), one can use Nanopolish or Racon to perform polishing and obtain improved assemblies (i.e. consensus sequences).

Nanopolish accepts mappings only in sequence alignment/map (SAM) format [88], and it works only with draft assemblies generated with the Metrichor-basecalled reads. On the other hand, Racon accepts both pairwise mapping format (PAF) mappings [46] and SAM-format mappings, but it requires the input reads and draft assembly files to be in fastq format [89], which includes quality scores. However, by using the -bq -1 parameter, it is possible to disable the filtering used in Racon, which requires quality scores. As our basecalled reads are in fasta format [90], in our experiments, we convert these fasta files into fastq files and disable the filtering with the corresponding parameter.

BWA-MEM generates mappings in SAM format, whereas Minimap generates mappings in PAF format. As Nanopolish requires SAM format input, we generate the mappings only with BWA-MEM and use them for Nanopolish polishing, in our analysis. On the other hand, as Racon accepts both formats, we generate the mappings and the overlaps with both BWA-MEM and Minimap, respectively, and use them for Racon polishing, in our analysis.

Accuracy

Table 12 presents the accuracy metrics results for Nanopolish (i.e. Rows 1–3) and Racon (i.e. Rows 4–23) pipelines. Based on these results, we make two observations.

Table 12. Accuracy analysis results for the full pipeline with a focus on the last two steps

										Number of Bases	Number of Contigs	Identity (%)	Coverage (%)	Number of Mismatches	Number of Indels
1	Metrichor	+	—	+	Canu	+	BWA-MEM	+	Nanopolish	4,683,072	1	99.48	99.93	8,198	15,581
2	Metrichor	+	Minimap	+	Miniasm	+	BWA-MEM	+	Nanopolish	4,540,352	1	92.33	96.31	162,884	182,965
3	Metrichor	+	GraphMap	+	Miniasm	+	BWA-MEM	+	Nanopolish	4,637,916	2	92.38	95.80	159,206	180,603
4	Metrichor	+	—	+	Canu	+	BWA-MEM	+	Racon	4,650,502	1	98.46	100.00	18,036	51,842
5	Metrichor	+	—	+	Canu	+	Minimap	+	Racon	4,648,710	1	98.45	100.00	17,906	52,168
6	Metrichor	+	Minimap	+	Miniasm	+	BWA-MEM	+	Racon	4,598,267	1	97.70	99.91	24,014	82,906
7	Metrichor	+	Minimap	+	Miniasm	+	Minimap	+	Racon	4,600,109	1	97.78	100.00	23,339	79,721
8	Nanonet	+	—	+	Canu	+	BWA-MEM	+	Racon	4,622,285	1	98.48	100.00	16,872	52,509
9	Nanonet	+	—	+	Canu	+	Minimap	+	Racon	4,620,597	1	98.49	100.00	16,874	52,232
10	Nanonet	+	Minimap	+	Miniasm	+	BWA-MEM	+	Racon	4,593,402	1	98.01	99.97	20,322	72,284
11	Nanonet	+	Minimap	+	Miniasm	+	Minimap	+	Racon	4,592,907	1	98.04	100.00	20,170	70,705
12	Scrappie	+	—	+	Canu	+	BWA-MEM	+	Racon	4,673,871	1	98.40	99.98	13,583	60,612
13	Scrappie	+	—	+	Canu	+	Minimap	+	Racon	4,673,606	1	98.40	99.98	13,798	60,423
14	Scrappie	+	Minimap	+	Miniasm	+	BWA-MEM	+	Racon	5,157,041	8	97.87	99.80	18,085	78,492
15	Scrappie	+	Minimap	+	Miniasm	+	Minimap	+	Racon	5,156,375	8	97.87	99.94	17,922	77,807
16	Nanocall	+	—	+	Canu	+	BWA-MEM	+	Racon	1,383,851	86	93.49	28.82	19,057	65,244
17	Nanocall	+	—	+	Canu	+	Minimap	+	Racon	1,367,834	86	94.43	28.74	15,610	55,275
18	Nanocall	+	Minimap	+	Miniasm	+	BWA-MEM	+	Racon	4,707,961	5	90.75	97.11	91,502	347,005
19	Nanocall	+	Minimap	+	Miniasm	+	Minimap	+	Racon	4,673,069	5	92.23	97.10	72,646	291,918
20	DeepNano	+	—	+	Canu	+	BWA-MEM	+	Racon	7,429,290	106	96.46	99.24	27,811	102,682
21	DeepNano	+	—	+	Canu	+	Minimap	+	Racon	7,404,454	106	96.03	99.21	34,023	110,640
22	DeepNano	+	Minimap	+	Miniasm	+	BWA-MEM	+	Racon	4,566,253	1	96.76	99.86	25,791	125,386
23	DeepNano	+	Minimap	+	Miniasm	+	Minimap	+	Racon	4,571,810	1	96.90	99.97	24,994	119,519

Observation 13: Both Nanopolish and Racon significantly increase the accuracy of the draft assemblies.

For example, Nanopolish increases the identity and coverage of the draft assembly generated with the Metrichor + Minimap + Miniasm pipeline from 87.71 and 94.85% (Row 2 of Table 10), respectively, to 92.33 and 96.31% (Row 2 of Table 12). Similarly, Racon increases them to 97.70 and 99.91% (Rows 6–7 of Table 12), respectively.

Observation 14: For Racon, the choice of read mapper does not affect the accuracy of the polishing step.

We observe that using BWA-MEM or Minimap to generate the mappings for Racon results in almost identical accuracy metric results. For example, when we use BWA-MEM before Racon for the draft assembly generated with the Metrichor + Canu pipeline (Row 4 of Table 12), Racon results with 98.46% identity, 100.00% coverage, 18 036 mismatches and 51 482 indels. When we use Minimap, instead (Row 5 of Table 12), Racon results with 98.45% identity, 100.00% coverage, 17 096 mismatches and 52 168 indels, which is almost identical to the BWA-MEM results.

Performance

In the first part of the performance analysis for Nanopolish, we divide the draft assemblies into 50 kb segments and polish 4 of these segments in parallel with 10 threads for each segment. For Racon, each draft assembly is polished using 40 threads, but the tool, by default, divides the input sequence into windows of 20 kb length. Table 13 presents the performance results for Nanopolish (i.e. Rows 1–3) and Racon (i.e. Rows 4–23) pipelines. Based on these results, we make the following two observations.

Observation 15: Nanopolish is computationally much more intensive and thus greatly slower than Racon.

Nanopolish runs take days to complete, whereas Racon runs take minutes. This is mainly because Nanopolish works on each base individually, whereas Racon works on the windows. As each window is much longer (i.e. 20 kb) than a single base, the computational workload is greatly smaller in Racon. Also, Racon only uses the mappings/overlaps for polishing, whereas Nanopolish uses raw signal data and an HMM-based approach to generate the consensus sequence, which is computationally more expensive.

Observation 16: BWA-MEM is computationally more expensive than Minimap.

Although the choice of BWA-MEM and Minimap for the read mapping step does not affect the accuracy of the polishing step, these two tools have a significant difference in performance (Minimap2 [87] is a recently released successor to Minimap. We compare Minimap2 to BWAMEM and to Minimap, and make two observations. First, Minimap2 significantly outperforms BWA-MEM. As Minimap2 can produce SAM alignments (which BWA-MEM produces), we can replace BWA-MEM with Minimap2 in future genome assembly pipelines. Second, Minimap2 has similar accuracy and performance compared with Minimap. This is because Minimap2 and Minimap use similar indexing and seeding algorithms [87], and the new features of Minimap2 (more accurate chaining, base-level alignment, support for spliced alignment) are not used in the pipeline we analyze. As a result, our findings for Minimap generally remain the same for Minimap2.).

For a deeper performance analysis of these read mapping and polishing tools, we perform a scalability analysis for each

Table 13. Performance analysis results for the full pipeline with a focus on the last two steps

		Step 4: Read Mapper			**Step 5:** Polisher		
		Wall Clock Time (h:m:s)	CPU Time (h:m:s)	Memory Usage (GB)	Wall Clock Time (h:m:s)	CPU Time (h:m:s)	Memory Usage (GB)
1	Metrichor + — + Canu + BWA-MEM + Nanopolish	24:43	15:47:21	5.26	5:51:00	191:18:52	13.38
2	Metrichor + Minimap + Miniasm + BWA-MEM + Nanopolish	12:33	7:50:54	3.75	122:52:00	4458:36:10	31.36
3	Metrichor + GraphMap + Miniasm + BWA-MEM + Nanopolish	12:47	7:57:58	3.60	129:46:00	4799:03:51	31.31
4	Metrichor + — + Canu + BWA-MEM + Racon	24:20	15:43:40	6.60	14:44	9:09:22	8.11
5	Metrichor + — + Canu + Minimap + Racon	3	1:35	0.26	15:12	9:45:33	14.55
6	Metrichor + Minimap + Miniasm + BWA-MEM + Racon	12:10	7:48:10	5.19	15:43	9:33:39	9.98
7	Metrichor + Minimap + Miniasm + Minimap + Racon	3	1:24	0.26	20:28	8:57:40	18.24
8	Nanonet + — + Canu + BWA-MEM + Racon	9:08	5:53:18	4.84	6:33	4:02:10	4.47
9	Nanonet + — + Canu + Minimap + Racon	2	54	0.26	6:45	4:17:26	7.93
10	Nanonet + Minimap + Miniasm + BWA-MEM + Racon	4:40	2:58:02	3.88	7:08	4:19:30	5.35
11	Nanonet + Minimap + Miniasm + Minimap + Racon	2	46	0.26	7:01	4:18:48	9.53
12	Scrappie + — + Canu + BWA-MEM + Racon	33:41	21:11:06	8.66	13:32	8:24:44	7.58
13	Scrappie + — + Canu + Minimap + Racon	3	1:39	0.27	18:45	7:43:17	13.20
14	Scrappie + Minimap + Miniasm + BWA-MEM + Racon	22:41	14:31:00	6.08	14:37	8:53:59	9.50
15	Scrappie + Minimap + Miniasm + Minimap + Racon	3	1:27	0.27	15:10	9:02:45	12.72
16	Nanocall + — + Canu + BWA-MEM + Racon	4:52	3:01:15	3.80	11:07	3:26:52	5.63
17	Nanocall + — + Canu + Minimap + Racon	3	1:16	0.22	7:28	2:50:35	3.62
18	Nanocall + Minimap + Miniasm + BWA-MEM + Racon	16:06	10:27:20	5.06	18:56	11:32:45	11.47
19	Nanocall + Minimap + Miniasm + Minimap + Racon	4	1:18	0.26	11:49	7:08:59	10.98
20	DeepNano + — + Canu + BWA-MEM + Racon	17:36	11:30:20	4.43	12:48	7:13:04	8.88
21	DeepNano + — + Canu + Minimap + Racon	3	1:24	0.28	11:39	6:55:01	3.73
22	DeepNano + Minimap + Miniasm + BWA-MEM + Racon	8:15	5:22:29	4.11	14:16	8:34:32	10.30
23	DeepNano + Minimap + Miniasm + Minimap + Racon	3	1:10	0.26	12:29	7:55:32	17.11

read mapper and each polisher by running them on the desktop system and the big-mem system separately, with 1, 2, 4, 8 (maximum for desktop server), 16, 32, 40, 64 and 80 (maximum for big-mem server) threads, and measuring the performance metrics. Figure 4 shows the the speed, memory usage and parallel speedup of BWA-MEM and Minimap. We make two observations.

Observation 17: On both systems, Minimap is greatly faster than BWA-MEM (cf. Figure 4A and B). However, when the number of threads reaches high value, Minimap's performance degrades because of the synchronization overhead between its threads (cf. Figure 4F).

On the desktop system, Minimap is 332.0x faster than BWA-MEM, on average (see Figure 4A). On the big-mem system, Minimap is 294.6x and 179.6x faster than BWA-MEM, on average, when the number of threads is <32 and >32, respectively. This is because of the synchronization overhead that increases with the number of threads used in Minimap (see Observation 10). As we also show in Figure 4F, Minimap's speedup reduces when the number of threads exceeds 32, which is another indication of the synchronization overhead that causes Minimap to slow down.

Observation 18: Minimap's memory usage is independent of the number of threads and stays constant. In contrast, BWA-MEM's memory usage increases linearly with the number of threads (cf. Figure 4C and D).

In Minimap, memory usage is dependent on the hash table size and is independent of number of threads (see Observation 8). In contrast, in BWA-MEM, each thread separately performs computation for different groups of reads (as in Scrappie and Nanocall, see Observation 4). This causes the linear increase in memory usage of BWA-MEM when the number of threads increases.

Figure 5 shows the scalability results for Racon on the big-mem system. We obtain the results on both of the systems. However, we only show the results for the big-mem system, as the results for both of the systems are similar. We separately test the tool by using PAF mappings and SAM mappings. Based on the results, we make the following observation.

Observation 19: Racon's memory usage is independent of the number of threads for both PAF mode and SAM mode. However, the memory usage of PAF mode is 1.86x higher than the memory usage of SAM mode, on average (cf. Figure 5B).

The memory usage of Racon depends on the number of mappings received from the fourth step, as Racon performs polishing by using these mappings. Racon's memory usage is higher for the PAF mode because the number of mappings stored in the PAF files is greater than the number of mappings stored in the SAM files (i.e. 1.4x). However, using PAF mappings or SAM, mappings do not significantly affect the speed (see Figure 5A) and the parallel speedup (see Figure 5C) of Racon.

Figure 6 shows the scalability results for Nanopolish. We test the tool by separately using a 25 kb and a 50 kb segment length to assess the scalability of the tool with respect to the segment length, in addition to the scalability with respect to the number of threads. We measure the performance metrics. We only show the results for the big-mem system, as the results for both of the systems are similar. Based on the results, we make the following observation.

Observation 20: Nanopolish's memory usage is independent of the number of threads. However, its memory usage in dependent on the segment length (cf. Figure 6B).

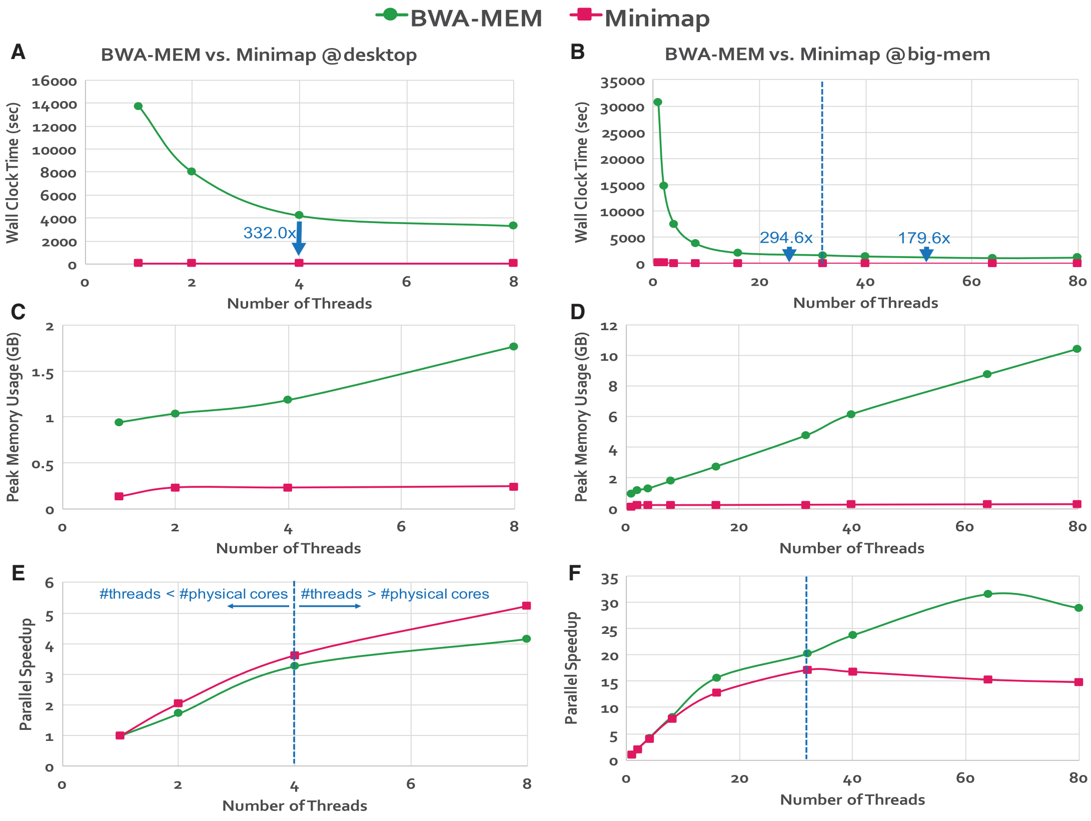

Figure 4. Scalability results of BWA-MEM and Minimap. Wall clock time (**A**, **B**), peak memory usage (**C**, **D**) and parallel speedup (**E**, **F**) results obtained on the desktop and big-mem systems. The left column (**A**, **C**, **E**) shows the results from the desktop system, and the right column (**B**, **D**, **F**) shows the results from the big-mem system.

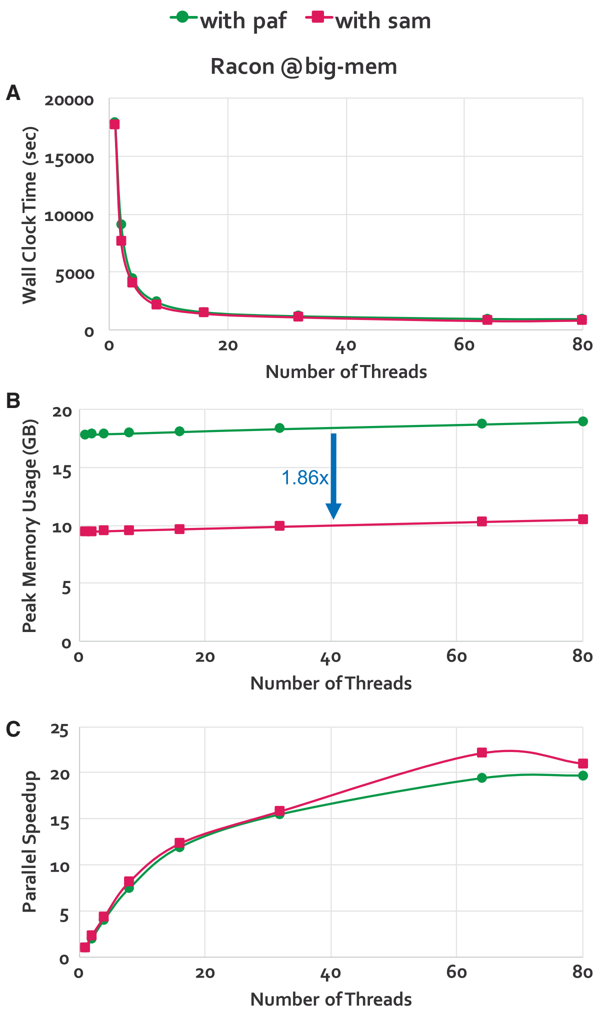

Figure 5. Scalability results of Racon. Wall clock time (**A**), peak memory usage (**B**) and parallel speedup (**C**) results obtained on the big-mem system.

The memory usage of Nanopolish is not affected by the number of threads. However, it is dependent on the segment length. Nanopolish uses more memory for longer segments. When the segment length is doubled from 25 to 50 kb, the increase in the memory usage (i.e. 2.7x) is >2.0x. This is because the memory usage of Nanopolish depends both on the length of the segment and the number of read mappings that map to this segment. For both of the segments, the memory usage also affects the speed. The Nanopolish run for the 25 kb segment is 2.7x faster than the run for the 50 kb segment (see Figure 6A).

Observation 21: Nanopolish's performance greatly degrades when the number of threads exceeds the number of physical cores (cf. Figure 6C).

Hyper-threading causes a slowdown for Nanopolish because of the CPU-intensive workload of Nanopolish and the resulting high contention in the shared resources between the threads executing on the same core, as we discuss in Observation 5.

Summary. Based on the observations we make about tools for the optional last two steps of the pipeline, we conclude that further polishing can significantly increase the accuracy of the assemblies. As BWA-MEM and Nanopolish are more resource-intensive than Minimap and Racon, pipelines with Minimap and Racon can provide a significant speedup compared with the pipelines with BWA-MEM and Nanopolish while resulting with high-quality consensus sequences.

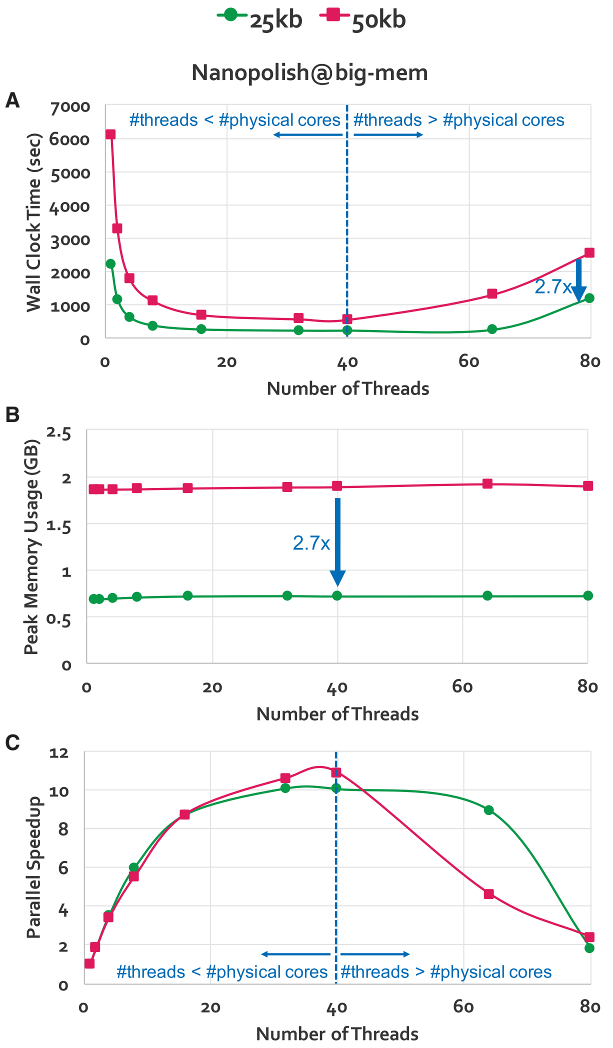

Figure 6. Scalability results of Nanopolish. Wall clock time (**A**), peak memory usage (**B**) and parallel speedup (**C**) results obtained on the big-mem system.

Recommendations

Recommendations for tool users

Based on the results we have collected and observations we have made for each step of the genome assembly pipeline using nanopore sequence data and the associated tools, we make the following major recommendations for the current and future tool users.

- ONT's basecalling tools, Metrichor, Nanonet and Scrappie, are the best choices for the basecalling step in terms of both accuracy and performance. Among these tools, Scrappie is the newest, fastest and most accurate basecaller. Thus, we recommend using Scrappie for the basecalling step (see analysis in section 'Basecalling tools').
- For the read-to-read overlap finding step, Minimap is faster than GraphMap, and it requires low memory. Also, it has similar accuracy to GraphMap. Thus, we recommend Minimap for the read-to-read overlap finding step (see analysis in section 'Read-to-read overlap finding tools').
- For the assembly step, if execution time is not an important concern, we recommend using Canu, as it produces much more accurate assemblies. However, for a fast initial analysis, we recommend using Miniasm, as it is fast and its accuracy can be increased with an additional polishing step. If Miniasm is used

for assembly, we definitely recommend further polishing to increase the accuracy of the final assembly (see analysis in section 'Assembly tools'). Even though polishing takes a similar amount of time if we use Miniasm or Canu, the accuracy improvements are much smaller for a genome assembled using Canu. We hope that future work can improve the performance of polishing when the assembled genome already has high accuracy, to reduce the execution time of the overall assembly pipeline.

- For the polishing step, we recommend using Racon, as it is much faster than Nanopolish. Racon also produces highly-accurate assemblies (see analysis in section 'Read mapping and polishing tools').
- In the future, laptops may become a popular platform for running genome assembly tools, as the portability of a laptop makes it a good fit for in-field analysis. Compared with the desktop and server platforms that we use to test our pipelines, a laptop has even greater memory constraints and lower computational power, and we must factor in limited battery life when evaluating the tools. Based on the scalability studies we perform using our desktop and server platforms, we would likely recommend using Minimap followed by Miniasm for the assembly step, and Minimap followed by Racon for the polishing step, when performing assembly on a laptop. These three tools use relatively low amounts of memory, and execute quickly, which we expect would make the tools a good fit for the various constraints of a laptop. Despite their low memory usage and fast execution, our recommended pipeline can produce high-quality assemblies that are suitable for fast initial in-field analyses. We leave it to future work to quantitatively study the genome assembly pipeline using nanopore sequence data on laptops and other mobile devices.

Recommendations for tool developers

Based on our analyses, we make the following recommendations for the tool developers.

- The choice of language to implement the tool plays a crucial role regarding the overall performance of the tool. For example, although the basecallers Scrappie and Nanonet belong to the same family (i.e. they both use the more accurate RNNs for basecalling), Scrappie is significantly faster than Nanonet, as Scrappie is implemented in C whereas Nanonet is implemented in Python (see analysis in section 'Basecalling tools').
- Memory usage is an important factor that greatly affects the performance and the usability of the tool. While developing new tools or improving the current ones, the developers should be aware of the memory hierarchy. Data structure choices that can minimize the memory requirements and cache-efficient algorithms have a positive impact on the overall performance of the tools. Keeping memory usage in check with the number of threads can enable not only a usable (i.e. runnable on machines with relatively small memories) tool but also a fast one. For example, we find that GraphMap cannot even run with a single-thread in our desktop machine because of excessively high memory usage (see analyses in 'Basecalling tools', 'Read-to-read overlap finding tools', 'Assembly tools' and 'Read mapping and polishing tools' sections).
- Scalability of the tool with the number of cores/threads is an important requirement. It is important to make the tool efficiently parallelized to decrease the overall runtime. Design choices should be made wisely while considering the possible overheads that parallelization can add. For example, we find that the parallel speedup of Minimap reduces when the number of threads reaches a high number because of a large increase in the overhead of synchronization between threads (see analyses in 'Basecalling tools', 'Read-to-read overlap finding tools', 'Assembly tools' and 'Read mapping and polishing tools' sections).
- As parallelizing the tool can increase the memory usage, dividing the input data into batches, or limiting the memory usage of each thread, or dividing the computation instead of dividing the data set between simultaneous threads can prevent large increases in memory usage, while providing performance benefits from parallelization. For example, in Nanonet, all of the threads share the computation of each read, and thus, memory usage is not affected by the amount of thread parallelism. As a result, Nanonet's usability is not limited to machines with relatively larger memories (see analyses in 'Basecalling tools', 'Read-to-read overlap finding tools', 'Assembly tools' and 'Read mapping and polishing tools' sections).

Conclusion

We analyze the multiple steps and the associated state-of-the-art tools in the genome assembly pipeline using nanopore sequence data in terms of accuracy, speed, memory efficiency and scalability (We leave it to future work to quantitatively study tools for different applications of nanopore sequencing, such as variant calling, detection of base modifications (i.e. methylation studies [91]) and pathogen detection.). We make four major conclusions based on our experimental analyses of the whole pipeline. First, the basecalling tools with higher accuracy and performance, like Scrappie, can overcome the major drawback of nanopore sequencing technology, i.e. high error rates. Second, the read-to-read overlap finding tools, Minimap and GraphMap, perform similarly in terms of accuracy. However, Minimap performs better than GraphMap in terms of speed and memory usage by storing only minimizers instead of all *k*-mers, and GraphMap is not scalable when running on machines with relatively small memories. Third, the fast but less accurate assembler Miniasm can be used for a fast initial assembly, and further polishing can be applied on top of it to increase the accuracy of the final assembly. Fourth, a state-of-the-art polishing tool, Racon, generates high-quality consensus sequences while providing a significant speedup over another polishing tool, Nanopolish.

We hope and believe that our observations and analyses will guide researchers and practitioners to make conscious and effective choices while deciding between different tools for each step of the genome assembly pipeline using nanopore sequence data. We also hope that the bottlenecks or the effects of design choices we have found and exposed can help developers in building new tools or improving the current ones.

Key Points

To our knowledge, this is the first work that analyzes state-of-the-art tools associated with each step of the genome assembly pipeline using sequence data generated with nanopore sequencing, a promising new sequencing technology.

The key contributions are:

1. We analyze the tools in multiple dimensions that are important for both developers and users/practitioners: accuracy, performance, memory usage and scalability.

2. We reveal new bottlenecks and trade-offs that different combinations of tools lead to, based on our extensive experimental analyses.
3. We provide guidelines for both practitioners, such that they can determine the appropriate tools and tool combinations that can satisfy their goals, and tool developers, such that they can make design choices to improve current and future tools.
4. We show that tools that are aware of the memory hierarchy have a better overall performance and scalability, and they are more usable than the tools that do not keep memory usage in check with the number of threads.
5. We show that basecalling is the most important step of the pipeline to overcome the high error rates of nanopore sequencing technology.
6. We show that there is a trade-off between accuracy and performance when choosing the tool for the assembly step. Miniasm, coupled with an additional polishing step, can lead to faster overall assembly than using Canu itself while producing high-quality assemblies.

Acknowledgments

The authors thank Jared Simpson and David Matei for their feedback and help with the questions about the tools. Posters describing earlier stages of the work in this article were presented at PSB 2017 and ISMB-ECCB 2017. The authors thank the poster session attendees for their feedback on the works. The authors especially thank Adam M. Phillippy and Mile Šikić for their feedback during the poster sessions. The authors also thank developers of Nanonet and Racon for answering our questions on GitHub.

Funding

This work was supported by a grant from the National Institutes of Health to O.M. and C.A. (grant number HG006004); an installation grant from the European Molecular Biology Organization to C.A. (grant number EMBO-IG 2521); and gifts from Google, Intel, Samsung and VMware.

References

1. Van Dijk EL, Auger H, Jaszczyszyn Y. Ten years of next-generation sequencing technology. *Trends Genet* 2014;**30**(9):418–26.
2. Hongyi X, Donghyuk L, Farhad H, *et al*. Accelerating read mapping with FastHASH. *BMC Genomics* 2013;**14(Suppl 1)**:S13.
3. Shendure J, Balasubramanian S, Church GM, *et al*. DNA sequencing at 40: past, present and future. *Nature* 2017;**550**(7676):345–53.
4. Steinberg KM, Schneider VA, Alkan C, *et al*. Building and improving reference genome assemblies. *Proc IEEE* 2017;**105**(3):422–35.
5. Treangen TJ, Salzberg SL. Repetitive DNA and next-generation sequencing: computational challenges and solutions. *Nat Rev Genet* 2011;**13**(1):36–46.
6. Firtina C, Alkan C. On genomic repeats and reproducibility. *Bioinformatics* 2016;**32**(15):2243–7.
7. Alkan C, Sajjadian S, Eichler EE. Limitations of next-generation genome sequence assembly. *Nat Methods* 2011;**8**(1):61–5.
8. Lu H, Giordano F, Ning Z. Oxford Nanopore MinION sequencing and genome assembly. *Genomics Proteomics Bioinformatics* 2016;**14**(5):265–79.
9. Magi A, Semeraro R, Mingrino A, *et al*. Nanopore sequencing data analysis: state of the art, applications and challenges. *Brief Bioinform* 2017, in press.
10. Clarke J, Wu HC, Jayasinghe L, *et al*. Continuous base identification for single-molecule nanopore DNA sequencing. *Nat Nanotechnol* 2009;**4**(4):265–70.
11. Marx V. Nanopores: a sequencer in your backpack. *Nat Methods* 2015;**12**(11):1015–18.
12. Branton D, Deamer DW, Marziali A, *et al*. The potential and challenges of nanopore sequencing. *Nat Biotechnol* 2008;**26**(10):1146–53.
13. Laver T, Harrison J, O'neill PA, *et al*. Assessing the performance of the Oxford Nanopore Technologies MinION. *Biomol Detect Quantif* 2015;**3**:1–8.
14. Ip CLC, Loose M, Tyson JR, *et al*. MinION analysis and reference consortium: phase 1 data release and analysis. *F1000Res* 2015;**4**:1075.
15. Kasianowicz JJ, Brandin E, Branton D, *et al*. Characterization of individual polynucleotide molecules using a membrane channel. *Proc Natl Acad Sci USA* 1996;**93**(24):13770–3.
16. MinION, Oxford Nanopore Technologies. 2017 https://nanoporetech.com/products/minion.
17. Quick J, Loman NJ, Duraffour S, *et al*. Real-time, portable genome sequencing for Ebola surveillance. *Nature* 2016;**530**(7589):228–232.
18. Quick J, Quinlan AR, Loman NJ. A reference bacterial genome dataset generated on the MinION^{TM} portable single-molecule nanopore sequencer. *Gigascience* 2014;**3**(1):22.
19. Jain M, Koren S, Miga KH, *et al*. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nat Biotechnol* 2018, in press.
20. Loman NJ. Thar she blows! Ultra long read method for nanopore sequencing. 2017. http://lab.loman.net/2017/03/09/ultra reads-for-nanopore/.
21. Madoui MA, Engelen S, Cruaud C, *et al*. Genome assembly using Nanopore-guided long and error-free DNA reads. *BMC Genomics* 2015;**16**(1):327.
22. First DNA sequencing in space a game changer. 2017. https://www.nasa.gov/mission_pages/station/research/news/dna_sequencing.
23. Update: New R9 nanopore for faster, more accurate sequencing, and new ten minute preparation kit. 2017. https://nanoporetech.com/about-us/news/update-new-r9-nanopore-faster-more-accurate-sequencing-and-new-ten-minute-preparation.
24. Pop M. Genome assembly reborn: recent computational challenges. *Brief Bioinform* 2009;**10**(4):354–66.
25. Clive Brown Technical Update: GridION X5—The Sequel. 2017. https://nanoporetech.com/resource-centre/videos/gridion-x5-sequel.
26. de Lannoy C, de Ridder D, Risse J. A sequencer coming of age: de novo genome assembly using MinION reads. *F1000Res* 2017;**6**:1283.
27. Metrichor. Oxford Nanopore Technologies. 2017. https://nanoporetech.com/products/metrichor.
28. Nanonet. Oxford Nanopore Technologies. 2017. https://github.com/nanoporetech/nanonet.

29. Scrappie. Oxford Nanopore Technologies. 2017. https://github.com/nanoporetech/scrappie.
30. David M, Dursi LJ, Yao D, *et al*. Nanocall: an open source basecaller for Oxford Nanopore sequencing data. *Bioinformatics* 2017;**33**(1):49–55.
31. Boža V, Brejová B, Vinař T. DeepNano: deep recurrent neural networks for base calling in MinION nanopore reads. *PLoS One* 2017;**12**(6):e0178751.
32. New basecaller now performs 'raw basecalling', for improved sequencing accuracy. 2017. https://nanoporetech.com/about-us/news/new-basecaller-now-performs-raw-basecalling-improved-sequencing-accuracy.
33. Teng H, Hall MB, Duarte T, *et al*. Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning. *bioRxiv* 2017;179531.
34. Wick RR, Judd LM, Holt KE. Comparison of Oxford Nanopore basecalling tools. 2017. https://github.com/rrwick/Basecalling-comparison.
35. Eddy SR. Hidden markov models. *Curr Opin Struct Biol* 1996; **6**(3):361–5.
36. Schuster M, Paliwal KK. Bidirectional recurrent neural networks. *IEEE Trans Signal Process* 1997;**45**(11):2673–81.
37. Pearlmutter BA. Learning state space trajectories in recurrent neural networks. *Neural Computation* 1989;**1**(2):263–69.
38. Nanonet: First Generation RNN Basecaller. https://github.com/nanoporetech/nanonet.
39. Nanocall: An Oxford Nanopore Basecaller. 2017. https://github.com/mateidavid/nanocall.
40. Pevzner PA, Tang H, Waterman MS. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci USA* 2001;**98**(17):9748–53.
41. Compeau PEC, Pevzner PA, Tesler G. How to apply de Bruijn graphs to genome assembly. *Nat Biotechnol* 2011;**29**(11): 987–91.
42. Koren S, Harhay GP, Smith TPL, *et al*. Reducing assembly complexity of microbial genomes with single-molecule sequencing. *Genome Biol* 2013;**14**(9):R101.
43. Chu J, Mohamadi H, Warren RL, *et al*. Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art. *Bioinformatics* 2017;**33**(8):1261–70.
44. Li Z, Chen Y, Mu D, *et al*. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Brief Funct Genomics* 2012;**11**(1):25–37.
45. Sović I, Šikić M, Wilm A, *et al*. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nat Commun* 2016;**7**:11307.
46. Li H. Minimap and Miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics* 2016;**32**(14): 2103–10.
47. Burkhardt S, Kärkkäinen J. Better filtering with gapped q-grams. *Fundam Inform* 2003;**56**(1–2):51–70.
48. Koren S, Walenz BP, Berlin K, *et al*. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Res* 2017;**27**(5):722–36.
49. Canu Tutorial. 2017. http://canu.readthedocs.io/en/latest/tutorial.html.
50. Loman NJ, Quick J, Simpson JT. A complete bacterial genome assembled *de novo* using only nanopore sequencing data. *Nat Methods* 2015;**12**(8):733–5.
51. Vaser R, Sović I, Nagarajan N, *et al*. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Res* 2017;**27**(5):737–46.
52. Heng L. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv Preprint arXiv 1303.3997*, 2013.
53. Nanopolish. https://github.com/jts/nanopolish.
54. Lee C, Grasso C, Sharlow MF. Multiple sequence alignment using partial order graphs. *Bioinformatics* 2002;**18**(3):452–64.
55. Loman NJ. Nanopore R9 rapid run data release. 2017. http://lab.loman.net/2016/07/30/nanopore-r9-data-release/.
56. MUMmer 3.x. 2017. https://github.com/garviz/MUMmer.
57. Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. In: *Proceedings of the Advances in Neural Information Processing Systems*. Neural Information Processing Systems Foundation, La Jolla, CA, 2014, 3104–12.
58. Forney GD. The Viterbi algorithm. *Proc IEEE* 1973;**61**(3):268–78.
59. Marr D, Binns F, Hill D. Hyper-threading technology in the NetBurst® microarchitecture. In: *Proceedings of the 14th Hot Chips Symposium*, 2002.
60. Magro W, Petersen P, Shah S. Hyper-threading technology: impact on compute-intensive workloads. *Intel Technol J* 2002; **6**(1):1–9.
61. Tuck N, Tullsen DM. Initial observations of the simultaneous multithreading Pentium 4 processor. In: *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, PACT*. IEEE Computer Society, Washington, DC, 2003.
62. Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: Maximizing on-chip parallelism. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA*. ACM, New York, NY, 1995.
63. Eggers SJ, Emer JS, Levy HM, *et al*. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro* 1997;**17**(5):12–19.
64. Tullsen DM, Eggers SJ, Emer JS, *et al*. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: *Proceedings of the 23rd Annual International Symposium on Computer Architecture, ISCA*. ACM, New York, NY, 1996, 191–202.
65. Yamamoto W, Nemirovsky M. Increasing superscalar performance through multistreaming. In: *Proceedings of the Working Conference on Parallel Architectures and Compilation Techniques, PACT*. IFIP Working Group on Algol, Manchester, UK, 1995, 49–58.
66. Hirata H, Kimura K, Nagamine S. *et al*. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture, ISCA*. ACM, New York, NY, 1992, 136–45.
67. Xiao CL, Chen Y, Xie SQ, *et al*. MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads. *Nat Methods* 2017;**14**(11):1072–74.
68. Li H, Durbin R. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 2009;**25**(14): 1754–60.
69. Langmead B, Trapnell C, Pop M, *et al*. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol* 2009;**10**(3):R25.
70. Alkan C, Kidd JM, Marques-Bonet T, *et al*. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nat Genet* 2009;**41**(10):1061–7.
71. Hach F, Hormozdiari F, Alkan C, *et al*. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nat Methods* 2010;**7**(8):576–7.
72. Schatz MC. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics* 2009;**25**(11):1363–9.
73. Li H, Ruan J, Durbin R. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res* 2008;**18**(11):1851–8.

74. Kim JS, Senol Cali D, Xin H, *et al*. GRIM-Filter: Fast seed location filtering in DNA read mapping using Processing-in-Memory technologies. *BMC Genomics* 2018, in press.
75. Xin H, Greth J, Emmons J, *et al*. Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics* 2015;**31**(10):1553–60.
76. Alser M, Hassan H, Xin H, *et al*. GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics* 2017;**33**(21):3355–63.
77. Alser M, Mutlu O, Alkan C. MAGNET: understanding and improving the accuracy of genome pre-alignment filtering. *IPSI Trans Internet Res* 2017;**13**(2):33–42.
78. Weese D, Emde AK, Rausch T, *et al*. RazerS-fast read mapping with sensitivity control. *Genome Res* 2009;**19**(9):1646–54.
79. Lee WP, Stromberg MP, Ward A, *et al*. MOSAIK: a hash-based algorithm for accurate next-generation sequencing short-read mapping. *PLoS One* 2014;**9**(3):e90581.
80. Rumble SM, Lacroute P, Dalca AV, *et al*. SHRiMP: accurate mapping of short color-space reads. *PLoS Comput Biol* 2009; **5**(5):e1000386.
81. David M, Dzamba M, Lister D, *et al*. SHRiMP2: sensitive yet practical short read mapping. *Bioinformatics* 2011;**27**(7): 1011–12.
82. Hatem A, Bozdağ D, Toland AE, *et al*. Benchmarking short sequence mapping tools. *BMC Bioinformatics* 2013;**14**(1):184.
83. Olson CB, Kim M, Clauson C, *et al*. Hardware acceleration of short read mapping. In: *Proceedings of the 20th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM*. IEEE Computer Society, Washington, DC, 2012, 161–8.
84. Fonseca NA, Rung J, Brazma A, *et al*. Tools for mapping high-throughput sequencing data. *Bioinformatics* 2012;**28**(24): 3169–77.
85. Li H, Durbin R. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 2010;**26**(5):589–95.
86. Siragusa E, Weese D, Reinert K. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Res* 2013;**41**(7):e78.
87. Li H. Minimap2: fast pairwise alignment for long DNA sequences.arXiv:1708.01492, 2017.
88. Li H, Handsaker B, Wysoker A, *et al*. The sequence alignment/ map format and SAMtools. *Bioinformatics* 2009;**25**(16):2078–9.
89. Cock PJA, Fields CJ, Goto N, *et al*. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res* 2010;**38**(6):1767–71.
90. Pearson WR, Lipman DJ. Improved tools for biological sequence comparison. *Proc Natl Acad Sci USA* 1988;**85**(8): 2444–8.
91. Simpson JT, Workman RE, Zuzarte PC, *et al*. Detecting DNA cytosine methylation using nanopore sequencing. *Nat Methods* 2017;**14**(4):407–10.