An Efficient Parallel Sketch-based Algorithmic Workflow for Mapping Long Reads

Tazin Rahman, Oieswarya Bhowmik, Ananth Kalyanaraman
Washington State University, Pullman, WA, USA

***Abstract*—Long read technologies are continuing to evolve at a rapid pace, with the latest of the high fidelity technologies delivering reads over 10Kbp with high accuracy (99.9%). Classical long read assemblers produce assemblies directly from long reads. Hybrid assembly workflows provide a way to combine partially constructed assemblies (or contigs) with newly sequenced long reads in order to generate improved and near-complete genomic scaffolds. Under either setting, the main computational bottleneck is the step of mapping the long reads—against other long reads or pre-constructed contigs. While many tools implement the mapping step through alignments and overlap computations, alignment-free approaches have the benefit of scaling in performance. Designing a scalable alignment-free mapping tool while maintaining the accuracy of mapping (precision and recall) is a significant challenge. In this paper, we visit the generic problem of mapping long reads to a database of subject sequences, in a fast and accurate manner. More specifically, we present an efficient parallel algorithmic workflow, called `JEM-mapper`, that uses a new minimizer-based Jaccard estimator (or JEM) sketch to perform alignment-free mapping of long reads. For implementation and evaluation, we consider two application settings: (i) the hybrid scaffolding setting, where the goal is to map a large collection of long reads to a large collection of partially constructed assemblies or contigs; and (ii) the classical long read assembly setting, where the goal is to map long reads to one another to identify overlapping long reads. Our algorithms and implementations are designed for execution on distributed memory parallel machines. We also implemented an MPI+OpenMP version of `JEM-mapper` to enable parallelism at both distributed- and shared-memory layers. Experimental evaluation shows that our parallel algorithm is highly effective in producing high-quality mapping while significantly improving the time to solution compared to state-of-the-art mapping tools. For instance, in the hybrid setting for a large genome *Betta splendens* ($\approx 350Mbp$ genome) with $429K$ HiFi long reads and $98K$ contigs, `JEM-mapper` produces a mapping with 99.41% precision and 97.91% recall, while yielding $6.9\times$ speedup over a state-of-the-art mapper.**

***Index Terms*—hybrid assembly, long read mapping, sketching, MinHashing, parallel algorithms, alignment-free**

I. Introduction

Over the last two decades, numerous genomes have been assembled using short read sequencing technologies. These technologies continue to present a cost-effective and high-throughput solution to sequencing. While short reads are accurate ($< 1\%$ error) the challenge is in their lengths (100 to 250 bp), which causes fragmented assemblies of contigs ($\approx 10^3 - 10^4$ bp) that are several orders of magnitude shorter than their target genomes ($\approx 10^6 - 10^9$). Recent emergence in long read sequencing technologies represents a significant advance in addressing this challenge. The first generation of long read technologies (e.g., PacBio SMRT [1] or Oxford Nanopore ONT [2]) produce reads that are over 10 Kbp but also have a larger error rate (between $11\%-14\%$ [3]). The more recent generation of technologies such as PacBio HiFi (High Fidelity) [4] have highly improved accuracy (99.9%). There are also several long read error correction tools [5]. Given these, the prospect of assembling long contiguous portions of the genome has dramatically improved [6].

Broadly speaking, two classes of approaches exist for using long reads—standalone and hybrid. *Standalone* long read assemblers [6]–[8] produce a *de novo* assembly from long reads using the Overlap-Layout-Consensus (OLC) paradigm [9]. HiFi long reads significantly facilitate the task of *standalone* assemblers and with higher accuracy in the reads, the overlap graph becomes smaller as compared to the overlap graphs generated from error-prone long reads. However, the OLC paradigm requires pairwise alignments of long reads which is the major computational bottleneck of *standalone* assemblers. The computational burden is exacerbated by the fact that more sequencing coverage is needed for *de novo* sequencing. For instance, this step took about 95% of the time while assembling *D. melanogaster* using [10].

Hybrid assemblers [11], [12], on the other hand, offer the benefit of combining long and short reads, or alternatively, combining long reads with prior constructed assemblies from short reads (i.e., contigs). The use of prior constructed contigs (in lieu of short reads) can improve scalability since the number of contigs tends to be orders of magnitude fewer than the number of short reads. By combining long reads with contigs we aim to extend the contigs into longer scaffolds through contig-to-contig linking information that may be available in the long reads. Depending on the genomic fraction covered by the contigs, this may also imply that it is possible to produce long scaffolds with a decreased sequencing coverage (and cost) in long read sequencing (compared to a *de novo* pipeline). In order to implement this hybrid strategy, however, we need a way to efficiently map the long reads to the contigs.

These two cases of mapping long reads motivate the development of the mapping strategy proposed in this paper. Fig. 1 illustrates these two mapping cases. Fig. 1(a) shows the case of mapping long reads for the hybrid setting, where the goal is to map long reads (shown as queries $\mathcal{Q}$) to partially constructed assemblies or contigs (considered subjects $\mathcal{S}$). The information computed by mapping can be used by a downstream scaffolder to fill in the assembly gaps between adjacent contigs and

Corresponding author(s). E-mail(s): {tazin.rahman, oieswarya.bhowmik, ananth}@wsu.edu.

(a) L2C usecase: Mapping a long read to a contig

(b) L2L use-case: Mapping a long read to long reads

Fig. 1: Our target use-cases: a) L2C mapping use-case: Subjects are a set of contigs and queries are a set of long reads. Each end of a long read can be expected to map to a single contig (assuming a non-redundant contig set). b) L2L mapping use-case: Subjects and queries are a set of long reads. For each long read l_q, the goal is to find its set of overlapping long reads. The reference genome ($\mathcal{G}$) is assumed to be unknown, and is only shown to display the mapping coordinates.

generate longer scaffolds. Fig. 1(b) shows the case of mapping long reads for the *de novo* long read assembly setting, where the goal is to map the long reads against themselves so as to detect overlapping long reads. Note that in both cases the reference genome $\mathcal{G}$ is assumed to be unknown. As a convention henceforth, we refer to the mapping application for the hybrid setting as L2C and the mapping application for the *de novo* assembly setting as L2L.

While there are a number of sequence mapping approaches (as reviewed in §II), scalability of these tools and in addition their ability to work with different types of sequences (contigs, long reads) are limitations. The key to performance scalability is to reduce alignment computations between long reads and the corresponding subjects. However, alignment-free approaches could suffer low precision and/or recall (see §II).

Contributions: In this paper, we present a new parallel algorithmic workflow for fast and accurate mapping of long reads, under both L2C and L2L use-cases. More generically, the inputs are a set of queries $\mathcal{Q}$ and a set of subjects $\mathcal{S}$. The output is a mapping for each $q \in \mathcal{Q}$ (as formally defined in §III). The main contributions of the paper are as follows.

- Methods: We present a new sketching-based method for alignment-free mapping of long reads. As part of our approach, we propose a minimizer-based Jaccard estimator (or JEM) sketch, that is a variant of the classical MinHashing (§III-B).
- Algorithmic workflow: We present an efficient and scalable parallel algorithmic workflow that uses the JEM sketch to perform mapping of long reads on distributed memory parallel machines. We adapt this workflow to provide two parallel implementations, one for L2C and another for L2L.
- Evaluation: We conduct a thorough empirical evaluation of the proposed sketching-based implementations for both of the use-cases. Results show that our method is able to match the mapping quality of a state-of-the-art mapping tool, while providing significant speedups over the state-of-the-art. In particular, for the L2C use-case, our distributed memory implementation running on 64 processes achieves speedups between $6.9\times$ to $13\times$ compared to the state-of-the-art multithreaded execution on 64 threads. Similarly, for the L2L use-case, for the complex genomes, our implementation running on 64 processes achieves speedups between $7.6\times$ to $15\times$ compared to the state-of-the-art multithreaded execution on 64 threads.

We refer to the newly proposed algorithmic workflow as JEM-mapper, and the source code is available as open source in https://github.com/TazinRahman1105050/JEM-Mapper for testing and application. A preliminary version of our paper appeared in [13].

With mapping applications increasingly becoming more heterogeneous in their data sources, including in hybrid scaffolding/assembly workflows or reference-guided assembly workflows [14], the techniques described in this paper have broad applicability. In what follows, we provide a review of the relevant sequence mapping literature (§II), our parallel approach and algorithmic workflow (§III), and the experimental results and evaluation (§IV).

II. Related work

Sequence mapping is a classical problem in bioinformatics. It can be abstracted as one of mapping a set of *query* sequences (e.g., reads) to a set of *subject* sequences. Traditional sequencing mapping tools (e.g., [15], [16]) focus on aligning short reads (queries) against a reference genome (subject). The hybrid setting differs from this classical setting in a couple of different ways. First, *in lieu* of the reference (which is typically a handful of very long subject sequences), the input subjects consist of a set of contigs which represent a highly fragmented view of the reference genome. Consequently, the contig sets can have tens to hundreds of thousands of sequences, and may also widely vary in their sequence lengths (10^3-10^5 bp).

As for the query set, long reads are significantly longer than the short reads used in conventional reference mapping. In the absence of more scalable tools, the current batch of hybrid assemblers [11], [12], [17] rely on a classical mapping tool to implement their mapping step. For instance, Haslr [12] first assembles the short reads using Minia [18], and then maps the contigs to the set of long reads using Minimap2 [16]. Similarly, SAMBA [17] maps the long reads to the set of contigs using Minimap2 [16]. We show in the results section (§IV) that using a generic mapping tool such as Minimap2 for L2C or L2L applications results could result in a loss in precision for larger inputs.

The step to compute overlapping pairs of long reads within long read assemblers can also be considered another form of mapping. State-of-the-art long read assemblers depend on either alignment tools (e.g., DALIGNER [20], BLASR [11], MECAT [8]) or overlap candidate detection (alignment-free) tools (for example, Minimap2 [16], or MHAP [6]).

Mapping tool	Problem scope			Approach		
	Supports reads to reference genome mapping	Supports reads to contig mapping (L2C)	Supportslong read mapping (L2L)	Alignment-based	Sketching-based	Type of parallelism
BLASR [11]	✔	✔	✔	✔	✖	shared-memory
MHAP [6]	✖	✖	✔	✖	✔	shared-memory
MECAT [8]	✔	✖	✔	✔	✖	shared-memory
Mashmap [19]	✔	✔	✖	✖	✔	shared-memory
Minimap2 [16]	✔	✔	✔	✖	✔	shared-memory
JEM-mapper (this paper)	✖	✔	✔	✖	✔	distributed-memory

TABLE I: State-of-the-art comparison of relevant sequence mapping tools. The first three columns indicate the type of mapping targeted with the long reads (i.e., the problem scope). The remaining columns on the right refer to the type of approach used.

To improve scalability of mapping, there has been a growing interest in alignment-free approaches [21]–[25], and in particular sketching—e.g., minimizers [25], [26] and MinHashing [27]. Sketching is a class of techniques that use samples derived from the input sets (or sequences) to be compared in order to approximate similarity. Introduced originally for document clustering [27], sketching and its relatives like minimizers [26] have found extensive use in bioinformatics. Minimizer sketch forms the basis of the widely used mapping tool Minimap2 [16], [28]. While these techniques have shown significant promise for mapping in the classical setting, they have not yet been fully demonstrated for the use-cases targeted in this paper (L2C and L2L). Among the alignment-free approaches for overlap detection in standalone assemblers, MHAP [6] uses Jaccard similarity between long reads to estimate overlap between sequences. However, this approach achieves a very low F1 score [9]. So, the balance between precision and recall is not maintained for complex genomes. Mashmap [19], [29] uses another sketching technique named the minimizer Jaccard estimate.

Table I provides a high-level summary of the different mapping tools that can handle long reads. The mapping tool presented in this paper (JEM-mapper) is shown as the last row. It can be used for both L2C and L2L use-cases, and is the only tool with support for distributed memory parallelism.

III. Methods

In this paper, we consider two closely related mapping problem abstractions, motivated by two specific use-cases, as described below. Figure 1 illustrates the two use-cases. Let $\mathcal{Q}$ denote a set of query sequences, and $\mathcal{S}$ denote a set of subject sequences. The sequences use an alphabet Σ (DNA: $\Sigma = \{\mathsf{a}, \mathsf{c}, \mathsf{g}, \mathsf{t}\}$); therefore, $\mathcal{Q} \subseteq \Sigma^*$ and $\mathcal{S} \subseteq \Sigma^*$. Let $\psi : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}_{\geq 0}$ be a mapping function to map a query q to a subject s.

Definition 1. The L2C mapping use-case: *Given $\mathcal{Q}$ and $\mathcal{S}$, find for each query $q \in \mathcal{Q}$ a best mapping subject $s_q^* \in \mathcal{S}$, given by,*

$$s_q^* = \arg\max_s \psi(q, s)$$

Setting $\mathcal{Q}$ to a set of input long reads and $\mathcal{S}$ to a set of input contigs (hence the term, L2C) would make the results of mapping useful in a hybrid scaffolding workflow.

Definition 2. The L2L mapping use-case: *Given $\mathcal{Q}$ and $\mathcal{S}$, and given a mapping quality threshold τ,* find *for each query $q \in \mathcal{Q}$ the set of subject sequences $A_q \subseteq \mathcal{S}$ such that:*

$$A_q = \{s \mid s \in \mathcal{S}, \psi(q, s) \geq \tau\}$$

Setting $\mathcal{Q}$ and $\mathcal{S}$, both to a set of input long reads (hence the term, L2L), would help identify overlapping pairs of reads for a long read assembler.

While the function ψ can be implemented as a sequence alignment, computing alignments at scale can be expensive. Therefore, alignment-free approaches are more desirable in practice. Our approach uses an alignment-free sketch to reduce the search space as described below.

A. Preliminaries and notation

a) MinHash preliminaries: The MinHash sketching scheme was introduced by Broder in 1997 [27], originally to compute resemblance or Jaccard similarity between documents. Given two sets A and B, the Jaccard similarity between the sets is given by: $\mathcal{J}(A, B) = \frac{|A \cap B|}{|A \cup B|}$. In this seminal work, Broder showed that there exists a family of permutations ($\pi : [n] \rightarrow [n]$) called the minwise independent permutations, which can be used to generate fixed size sketches from the sets A and B. It then suffices to compare the sketches instead of explicitly computing the $\mathcal{J}(A, B)$, i.e.,

$$Pr(min\{\pi(A)\} = min\{\pi(B)\}) = \mathcal{J}(A, B)$$

In other words, higher the Jaccard similarity, higher the probability that the sketches obtained A and B will match. To improve the chance that a random sketch is found, a fixed number of random trials (T) is executed. This is achieved by choosing T random minwise independently permutations: $\{\pi_1, \pi_2, \ldots, \pi_T\}$, and using them to generate the MinHash *sketches* for sets A and B, denoted by $\overline{A}$ and $\overline{B}$ respectively:

$$\begin{aligned} \overline{A} &= [\min\{\pi_1(A)\}, \ldots, \min\{\pi_T(A)\}]; \\ \overline{B} &= [\min\{\pi_1(B)\}, \ldots, \min\{\pi_T(B)\}] \end{aligned}$$

Subsequently, if any of the trials produce the same minimum between $\overline{A}$ and $\overline{B}$ then we conclude A is *similar* to B. In practice, a value around 100 to 200 is used for T [27]. We refer to the MinHash sketches (e.g., $\overline{A}$, $\overline{B}$) sometimes as just "MinHashes" for the underlying sets.

b) String notation: Let s denote an arbitrary string over alphabet Σ, and let $|s|$ denote its length. We use the terms strings and sequences interchangeably. A k-mer is a (sub)string of length k. Given Σ and k, let $\mathcal{K}$ denote the set of all k-mers that can be built using Σ. Note that $|\mathcal{K}|$=$|\Sigma|^k$. We use the term *canonical ordering* of k-mers, denoted by Π_k^*, to refer to the lexicographical ordering of the k-mers in $\mathcal{K}$. For instance, if k=2, the canonical ordering of $\mathcal{K}$ is given by: Π_k^* = [aa,ac,ag,at,ca,cc,cg,ct,ga,gc,gg,gt,ta,tc,tg,tt]. Given a string $s \in \Sigma^*$ and a choice of k, the notation s_k is used to denote the set of all k-mers present in s.

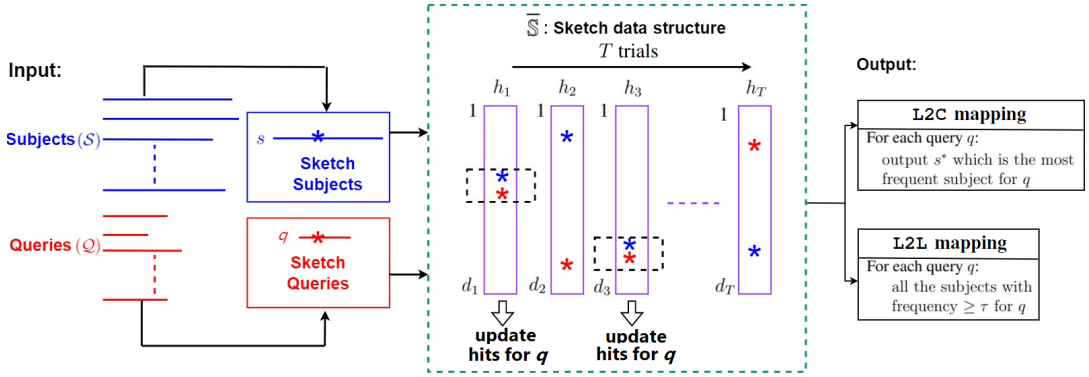


Fig. 2: `JEM-mapper`: Illustration of the major steps of our sketch-based mapping algorithm, `JEM-mapper`. The workflow shown is generic to work for both `L2C` and `L2L` use-cases.

B. Computing `mapping` using a minimizer-based Jaccard estimator sketch

In the mapping applications, we have two sets of strings—$\mathcal{Q}$ containing queries, and $\mathcal{S}$ containing subjects. For a string s, its `MinHash` sketch can be constructed from the set of all k-mers (s_k) in s—i.e., during trial t, apply a hash function $h_t(.)$ on each k-mer in s_k and then select the k-mer with the minimum value as part of the sketch.

Using this idea, a simple way to apply the MinHashing scheme for `mapping` is as follows. First enumerate the MinHashes (or the sketches) for each subject (one minimum for each random trial $t \in [1, T]$) and insert those into a sketch data structure $\overline{\mathbb{S}}$. Subsequently, during querying time, sketches are also generated from each query. The more sketches a query generates in common with a subject, the higher the likelihood that it shares a high sequence level similarity. Therefore, we can simply track the frequency of the subjects that "hit" with a given query, and report the set of top matching subjects (if any) as the mapped output hit to that query. This simple algorithmic workflow is illustrated in Fig. 2.

While this workflow can be efficiently implemented, we make several changes, as there are a few key challenges with a direct application of MinHashing as described above.

First, note that in the `mapping` applications targeted, subjects and queries could have significantly differing lengths. If a query q is significantly longer (say $10Kbp$) than a corresponding mapped subject s (say $3Kbp$), then even if s has significant identity within q, MinHashing may select k-mers that may lie outside the region of the overlap. This could mean missing out on a true mapped (affects recall). Similarly, if a subject s is significantly longer (say, $20Kbp$) compared to a query q (say $10Kbp$), recall could again be affected as the sketches from s could lie outside the region of true overlap. Either way, the qualitative efficacy of `MinHash` for mapping could be negatively impacted.

To overcome this limitation, we use two ideas: a) to map only segments of queries; and b) to compute a minimizer-based Jaccard estimator (instead of the classical `MinHash` form).

1) Using the segments *of a query q:* Instead of extracting sketches from the entire length of a query q, our approach uses only several regions (aka. "segments") of q. Specifically, we define a fixed *segment length* ℓ. We then map only λ *segments* of query q to subjects and report respective subjects with hits. The main idea is to focus on regions of high overlaps between the query and subject. This not only improves quality, but also reduces work, making the algorithm faster. In both of our applications (`L2C` and `L2L`), we found a value of $\lambda = 2$ to be sufficient in our experiments. Henceforth, we revise the set of queries $\mathcal{Q}$ to include the two segments of each query—i.e., if there are m queries, then $\mathcal{Q}$ consists of $2m$ sequences of length ℓ each. The heuristic to pick *segments* for `L2C` or `L2L` is described in (§III-C).Fig. 1 shows the end segments of queries mapped to subjects.

2) Sketching using minimizer Jaccard estimate: Segments help constrain the regions where sketches are extracted from the queries. However, subjects can be very long and it is possible that the region of overlap between a query and a subject can span anywhere in its length. Therefore, we follow a two-pronged idea by: a) reducing the base set of k-mers for Jaccard similarity computation to a set of *minimizers* [26] obtained from subjects, and b) then using a sliding interval of length ℓ bp over the list of those minimizers to select one `MinHash` per interval for a trial t. The list of minhashes so extracted becomes our version of the minimizer-based Jaccard estimator sketch (abbreviated as "JEM" henceforth) of the subject for that trial $t \in [1, T]$. Fig. 3 illustrates this procedure using a conceptual example. The detailed algorithm is as follows.

The minimizer-based Jaccard estimate calculates the Jaccard similarity between two sequences using the minimizer sketches between them. Given a sequence s, a window size w, and a pre-defined total ordering of all k-mers π, a *minimizer* is the smallest k-mer of the window. Typical choices for ordering k-mers include lexicographical order, frequency-based order, random order, and others [30]. We use the lexicographically smallest k-mer as our hash function, consistent with previous works [31], [32]. The *minimizer sketch* of s (denoted by $M(s, w)$) is the set of all such minimizers in s. Hence the minimizer Jaccard estimate between sequences A and B is:

$$\mathcal{J}_m(A, B; w) = \mathcal{J}(M(A, w), M(B, w))$$

In other words, the minimizer Jaccard estimate allows us to collect and compare sketches from the list of minimizers of a

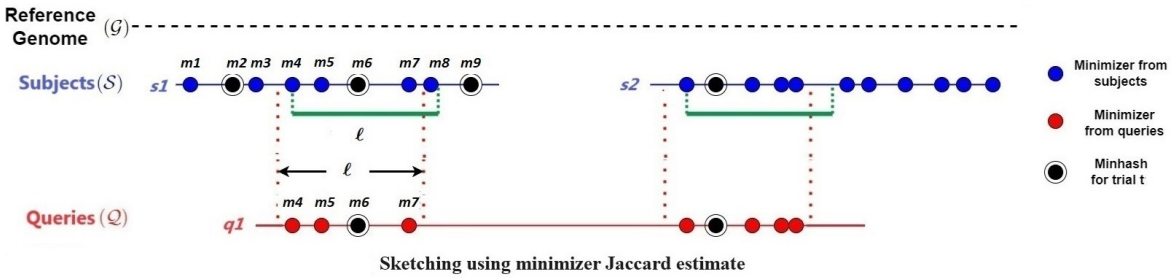


Fig. 3: An example to illustrate the way JEM-mapper works. At the subject processing time, the list of minimizer tuples $M_o(s,w)$ is generated for each subject (shown as blue circles). We then slide an interval of length ℓ over the set of minimizers based on their positions. On $s1$, this is shown as the list $[m_4 \dots m_8]$. For each such interval, we generate T minhashes for T trials. The black concentric circle shows the minhash that was randomly selected for trial t in that interval (i.e., the sketch). At query processing time, for long read segment, we generate a similar set of minimizers (denoted by the red circles). We then pick T JEM sketches in a similar fashion and look for hits in the sketch table before detecting the number of hits between a subject and a query.

sequence (rather than all the k-mers). This reduces work and also provides a certain degree of qualitative robustness against noisy k-mers.

Another type of a minimizer based Jaccard estimate has been used prior in the mapping tool Mashmap [21], [29]. Our algorithm is different in the way these sketches are computed. In Mashmap, for each minimizer, a list of all positions it is present in the subject is maintained. Later, during mapping time, if a query shares a minimizer with multiple positions, then the region where the query has maximal local intersection on the subject is detected and reported at query time. This approach entails at first, short-listing positional candidates and then eliminating those that do not have sufficient concentration of query minimizers in their vicinity.

By contrast, our approach directly applies the segment length ℓ of the query as the interval length over the subject, and tracks the MinHash for each such interval of the subject—as shown in Fig. 3. This guarantees that the sketches are generated at the resolution of the segment length, both for the subjects and queries, thereby obviating the need to check for any distance constraints later.

More specifically, let $M_o(s,w)$ represent the set of all minimizers and their corresponding positions from a string s. This is maintained as a set of tuples of the form $\langle k_i, p_i \rangle$, where k_i denotes the minimizer at position p_i on s. The set $M_o(s,w)$ is kept sorted based on the minimizer positions. For a given interval length ℓ, let us define M_i to be the set of consecutive minimizers in $M_o(s,w)$ such that $M_i = \{\langle k_j, p_j \rangle : p_i \leq p_j \leq p_i + \ell\}$. In other words, starting from each minimizer (i) in $M_o(s,w)$, we select that subset of minimizers that originate within the ℓ-characters that follow i. This is shown in the example of Fig. 3 for subject $s1$ and for the ℓ-window starting at minimizer $m4$, the following minimizers $[m4, m5, m6, m7, m8]$ are selected as that window's ground set M_4. Subsequently, T minhashes are computed from each such M_i to yield the window's JEM sketch. Fig. 4 shows a specific example of how to generate the JEM sketch corresponding to a specific ℓ-window sequence.

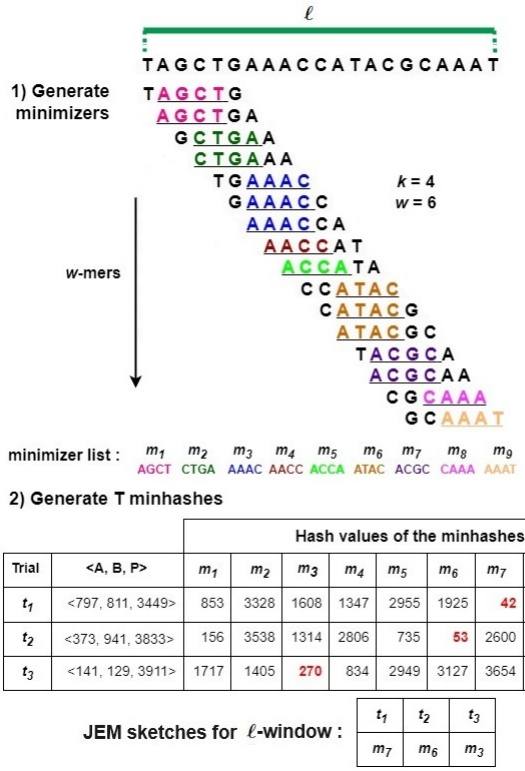
Algorithm 1 shows how sketches are extracted using our approach. Algorithm 2 shows the overall JEM-mapper algorithm. The algorithm first generates sketches (using Algorithm 1) for all the subjects, and populates them into a sketch data structure $\overline{\mathbb{S}}$. Subsequently, each query is processed by first generating its sketches and looking them up in $\overline{\mathbb{S}}$. The hits are accumulated with every query minhash colliding with a subject minhash, across the T trials. For L2C, the subject that has the largest number of hits with a query is reported as the mapped output. For L2L, the union of all subject sequences that generated a hit with the query is reported. In our implementation, we use a threshold τ for filtering low hit subjects (see the implementation remarks in Section III-D).

In our implementation, we have used lexicographic ordering of k-mers to extract minimizers from window w. For a substring s' of length greater than k, a canonical minimizer is the smallest k-mer of s' and its reverse complement $\overline{s'}$ based on lexicographic ordering. To generate the T minhashes for each interval, we assign each minimizer of that interval its

Algorithm 1: *Sketch_byJEM*

Input: s: input sequence
ℓ: segment length
$\mathcal{H}$: set of T hash functions $\{h_1, h_2, ..., h_T\}$
Output: sketches generated by for s

1: $Sketch \leftarrow []$
2: Let $s_k \leftarrow$ the set of all k-mers in s
3: $M_o(s,w) \leftarrow Generate\_Minimizers(s_k, w)$
/* returns a list of minimizer tuples $\langle k_i, p_i \rangle$, sorted by position index p_i */
4: **for** each tuple $\langle k_i, p_i \rangle \in M_o(s,w)$ **do**
5: $M_i \leftarrow Generate\_Interval(m_k, i, \ell)$ /* returns the set of minimizers $\{\langle k_j, p_j \rangle : p_i \leq p_j \leq p_{i+\ell}\}$ */
6: **for** each trial $t \in [1, T]$ **do**
7: $sketch \leftarrow \arg\min_{x \in M_i} h_t(x)$
8: $Sketch[t]$.insert(key: $sketch$, value: s)
9: **end for**
10: **end for**
11: **return** $Sketch$



		Hash values of the minhashes								
Trial	<A, B, P>	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9
t_1	<797, 811, 3449>	853	3328	1608	1347	2955	1925	**42**	84	3202
t_2	<373, 941, 3833>	156	3538	1314	2806	735	**53**	2600	1815	2060
t_3	<141, 129, 3911>	1717	1405	**270**	834	2949	3127	3654	1331	552

t_1	t_2	t_3
m_7	m_6	m_3

Fig. 4: An example to illustrate how a JEM sketch is generated for a window ℓ using $w = 6$ and $k = 4$ as the minimizer parameters, and $T = 3$ trials as the minhash parameter. The ℓ-window sequence is shown at the top. The first step shows how to detect all minimizers that are k-mers from each window of length w of the sequence. For this example, this step generates a list of nine minimizers, labelled m_1 through m_9. The second step uses these minimizers as the ground set, and computes $T = 3$ minhashes over the three trials. The resulting set of three of minhashes is the JEM sketch for this ℓ-window.

k-mer rank x (i.e., as per its canonical ordering in $\mathcal{K}$), and then use T random hash functions of the linear congruential form: $h_t(x) = ((A_t \cdot x + B_t) \mod P_t)$. Subsequently, the k-mer corresponding to the smallest hashed value becomes the sketch for that string for trial $t \in [1, T]$. Here the triplet $\langle A_t, B_t, P_t \rangle$ are randomly generated constants associated with the trial t. We generate T triplets, one for each trial, *a priori*, and use the same T triplets to sketch all sequences.

C. Additional implementation details

a) Masking of repeats in complex input genomes: For complex genomes, particularly for eukaryotic organisms, a significant portion of the whole genome is repetitive. For instance, more than 50% of the human genome is repetitive, with higher fractions for several plant genomes [33]–[37]. Although long-read sequencing technologies have improved the ability to resolve repeats in genomes, presence of short repetitive stretches can still confound the mapping process. For numerous genomes, repeat information is already available. To take advantage of such available information, we mask the repetitive regions of the input sequences using the Repeat-Masker tool [38]. Masked regions appear as 'N's. In both `L2C` and `L2L` use-cases, we have used masked input genomes.

Algorithm 2: Mapping by `JEM-mapper` $(\mathcal{Q}, \mathcal{S})$

Input: $\mathcal{Q}$: long read segments, $\mathcal{S}$: contigs, T: no. trials
Output: $\Phi : \mathcal{Q} \rightarrow \mathcal{S}$

Initialize sketch table: $\overline{\mathbb{S}}[t] \leftarrow \phi$, where $t \in [1, T]$
$\overline{\mathbb{S}}$.insert(Sketch_byJEM(s)), $\forall s \in \mathcal{S}$
for each $q \in \mathcal{Q}$ **do**
  $\overline{\mathbb{S}}$.lookup(Sketch_byJEM(q))
  for $t \in [1, T]$ **do**
    Let $Hits_q[t] \leftarrow \{s | q$ and s collide in $\overline{\mathbb{S}}[t]\}$
  end for
  if `L2C` **then**
    $\Phi(q) = s^*$, where s^* is the most frequent subject in $Hits_q$
  else if `L2L` **then**
    $\Phi(q) = \bigcup_t Hits_q[t]$,
  end if
end for
return Φ

b) Segments selection: Since our mapping uses sketches derived from segments, the selection of regions to extract segments could impact the mapping quality. In the case of `L2C`, long reads are mapped to contigs (prior constructed assemblies). Therefore, our implementation extracts segments from the ending regions of a long read—that way, the approach is suited to provide linking information between contigs, i.e., the farthest separated pair of contigs linked by a query long read (as illustrated in Fig. 3). This information can be used by a hybrid scaffolder to increase the span of its scaffold. In our `L2C` experiments, we used a value of $\ell = 1000$ bp.

For the `L2L` mapping use-case, note that each long read query can map to an arbitrary number of other long read subjects, and the regions of overlap between query and subject are not constrained to the ends of a long read. Therefore, our segment selection scheme does *not* constrain the selection to the ends of a long read. Instead, we select the top two non-overlapping segments (each of length up to ℓ bp) that have the least 'N' content. The rationale for choosing a segment with the least 'N' content is to focus on non-repetitive portions that overlap between the query and the subject. We use $\ell = 2000$ bp as our segment length for `L2L`, based on prior works that have suggested similar lengths [9], [28].

c) Frequency-based heuristics for unmasked inputs: As mentioned earlier, our default scheme uses lexicographic ordering of k-mers to extract minimizers from window w. However, for genomes where repeat information is not available for masking *a priori*, we use an alternate scheme to pick minimizers. More specifically, we count the frequency of k-mers in the subjects, and subsequently use a frequency-based heuristic that selects a least frequent k-mer as the minimizer. This idea of frequency-based minimizer sampling was originally used in the DBGFM data structure [39]. The idea exploits the simple expectation that k-mers from repetitive portions of the genome are likely to be more frequent and by selecting

a least frequent k-mer as a minimizer, we are preferring a candidate that is likely to uniquely identify that genomic region. For a substring s' of length w, the least frequent k-mer of s' and its reverse complement $\overline{s'}$ is considered as the minimizer. Note that erroneous k-mers also tend to have a low frequency and hence could become potential candidates for minimizers in this frequency-based scheme. To alleviate this issue, one can also use a minimum cutoff frequency. However, for high-fidelity (HiFi) long reads, this is less likely to be an issue—as corroborated in our experiments—because the HiFi long reads have high accuracy (99.9%).

D. Parallelization

We now present a distributed memory parallel algorithm for our `JEM-mapper` algorithm. The Algorithm 2 is well suited for parallelization on a distributed memory as described below. Fig. 5 illustrates the major steps of the parallel algorithm. Our implementation is in C/C++ and MPI (for communication). The beta version of this software is available on https://github.com/TazinRahman1105050/JEM-Mapper.

We use the following notation: $m = |\mathcal{Q}|$; $M = \Sigma_{q\in\mathcal{Q}}|q|$; $n = |\mathcal{S}|$; $N = \Sigma_{s\in\mathcal{S}}|s|$; and p to denote the number of processes. The major parallel steps are as follows.

S1) *(load input)* The processes load the input $\mathcal{Q}$ and $\mathcal{S}$ in a distributed manner, such that each process gets approximately $\mathcal{O}(\frac{M}{p})$ query bases and $\mathcal{O}(\frac{N}{p})$ subject bases. Let $\mathcal{Q}_{local}$ and $\mathcal{S}_{local}$ denote the sets of local queries and subjects respectively, held by any given process.

S2) *(sketch subjects)* Each process generates the sketches from $\mathcal{S}_{local}$, and inserts them into $\overline{\mathbb{S}}_{local}$, which holds all the sketches generated from that process.

S3) *(gather sketch)* In a global communication step that uses the `MPI_Allgatherv` primitive, we perform a union of all the $\overline{\mathbb{S}}_{local}$ into a single $\overline{\mathbb{S}}_{global}$ that is stored at each process. Note that $\overline{\mathbb{S}}_{global}$ consists of T lists, one for each trial, as shown in Fig. 5.

S4) *(map queries)* Each process then processes its local query set $\mathcal{Q}_{local}$. The mapping stage for each query $q \in \mathcal{Q}_{local}$ comprises of three steps:

Step a) slide window and generate its JEM sketches;

Step b) lookup the subject hits in $\overline{\mathbb{S}}_{global}$, as shown in Fig. 5; and

Step c) report mapping for the (or a) best hit.

As shown in Fig. 2, hits are located within $\overline{\mathbb{S}}_{global}$ by the corresponding trial numbers (step b). Subsequently, a reporting step scans the bins (or the list of trials) to generate the mapping output pairing queries to subjects (step c).

Additional remarks on our parallel implementation: The output for `L2C` mapping is the best hit as shown in Fig. 2. For step (c) above, we implemented a lazy update strategy to support a fast tracking of subjects and their hit rates *across queries*. More specifically, we initialize an array $A[1, n]$ of tuples of the form $\langle u, v\rangle$, where u is an integer counter initialized to 0, and v is the query id (initialized to -1). Whenever a query j generates a hit with a subject i, we check if $A[i].v$ is equal to j. If it is, then we simply increment counter $A[i].u$. But if it is not, then we first set $A[i].v$ to j, reset counter $A[i].u$ to 0, and then increment that counter (to 1). This lazy strategy avoids the cost of resetting the counters for all subjects whenever a new query is processed. Note that at each process, queries in $\mathcal{Q}_{local}$ are processed one by one.

For `L2L`, for a query long-read l_q, all the mapped subject reads with a certain frequency $\geq \tau$ are reported. The parameter τ is the minimum number of trials (out of T) during which a query has to generate a hit with a given subject, for it to be considered a successful map. Furthermore, note that for `L2L`, we do not need to process the query set and subject set separately. In the input loading step (S1), we just load the input set of long-reads $\mathcal{S}$ once, in a distributed manner. Let $\mathcal{S}_{local}$ denote the set of local long-reads. When we sketch the subjects (S2), we keep an additional boolean flag, a *tag*, to indicate if the *sketch* has been generated from a *segment* (described in §III-C) or not with each *sketch*. The *gather sketch* step (S3) stays the same. In the *map queries* step (S4), $\mathcal{S}_{local}$ is treated as the local query set. The *tag* for each sketch in $\overline{\mathbb{S}}_{global}$ helps to indicate sketches generated from a *segment* of a query long-read l_q (so duplicate work can be avoided). The reporting step scans the bins (or the list of trials) to generate the mapping output pairing long-reads.

E. Complexity analysis

The runtime complexity analysis of our parallel algorithm is as follows. The input loading step (S1) costs $\mathcal{O}(\frac{N+M}{p})$ time. Sketching the subjects (S2) can be achieved in $\mathcal{O}(\frac{n\ell_s T}{p})$ time, where ℓ_s is the average length of a subject. Similarly, sketching the queries (S4) can be achieved in $\mathcal{O}(\frac{m\ell_q T}{p})$ time, where ℓ_q is the average length of a query. The gather step (S3) involves communicating each $\overline{\mathbb{S}}_{local}$ to all processes, and can be achieved in $\mathcal{O}(\lambda \log p + \mu n T)$ time, where λ is the cost of network latency and μ is the reciprocal of network bandwidth (i.e., number of seconds per byte transferred). The parameters λ and μ are network constants in the Hockney model for parallel performance [40], and they can be empirically determined. While the worst-case size of $\overline{\mathbb{S}}_{global}$ is $\mathcal{O}(n\ell_s T)$, in practice we expect significantly fewer minhashes because we are selecting from the list of minimizers $M_o(s, w)$ (and not all k-mers). Finally, the query mapping step (S4) is a local step processing each query $r \in \mathcal{Q}_{local}$. The initialization of counters for the subjects takes $\mathcal{O}(n)$ time and after that each query is mapped through a linear scan of its sequence with T minhash computations at all its minimizers. Linear scan is sufficient because of the lazy counter update strategy described above. Consequently, step S4 takes $\mathcal{O}(n + \frac{m\ell_q T}{p})$ time. For `L2C`, since the number of long reads (m) can be expected to be significantly more than the number of contigs (n) due to sequencing coverage, we expect $\frac{m\ell_q T}{p}$ to dominate over n in practice. For `L2L`, we are not loading queries or sliding windows over queries seperately as subjects and queries are both long reads. We expect the generating sketches for subjects ($\frac{n\ell_q T}{p}$) to be the time consuming step for `L2L`.

The space complexity of our approach is dominated by the size of $\overline{\mathbb{S}}_{global}$. Let m_s denote the average number of minimizers generated per subject. Since we enumerate fixed-size intervals and store one minhash per interval, a subject

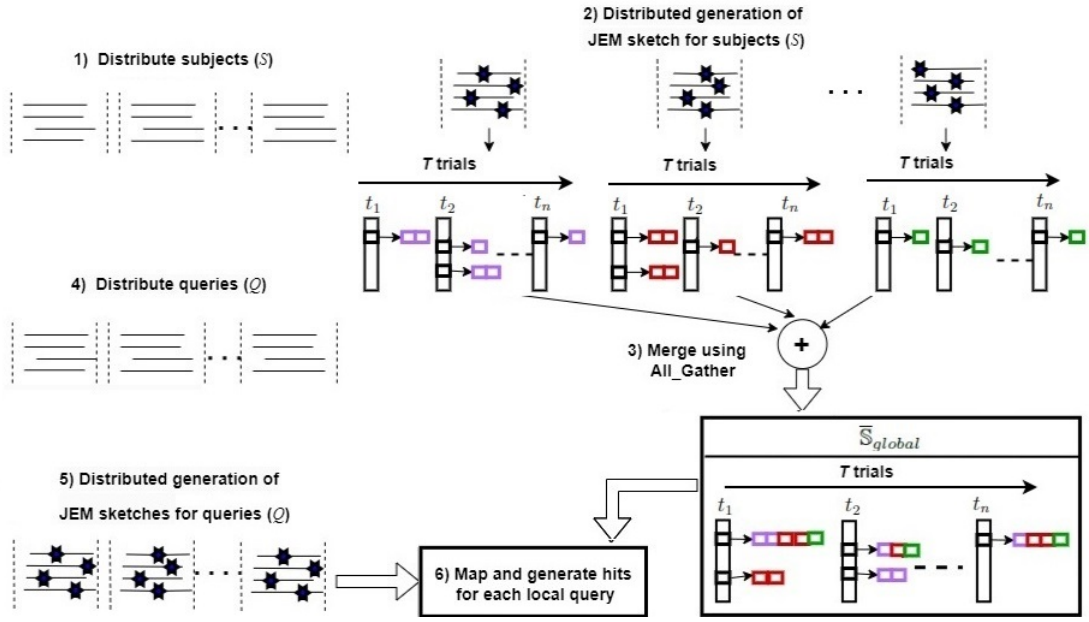


Fig. 5: Schematic illustration of all the major steps of our `JEM-mapper` distributed memory parallel workflow: 1) The input subjects ($\mathcal{S}$) are loaded in a distributed manner. 2) Each process generates the JEM sketches from its local subjects. 3) The global communication step merges the local sketches to generate $\overline{\mathbb{S}}_{global}$. 4) Queries ($\mathcal{Q}$) are loaded in a distributed manner. 5) Each process generates the JEM sketches from its local queries. 6) Each process maps the JEM sketches of queries to $\overline{\mathbb{S}}_{global}$ and generates the hits for each query. Note that in our parallel implementation, steps (2), (5), and (6) are also multi-threading enabled.

s can be expected to contribute up to $\mathcal{O}(m_s T)$ minhashes into the sketch. Therefore, the space complexity per process is $\mathcal{O}(nm_s T)$.

IV. Experimental Results

In this section, we present a thorough experimental evaluation of our sketch-based mapping algorithm, `JEM-mapper` (§ III-B)). We study both `L2C` and `L2L` use-cases, analyzing the method's quality and performance and comparing against respective state-of-the-art tools, and using both simulated and real-world data sets.

A. Experimental setup

Test inputs: We used two sets of long read inputs in our experiments (see Table II):

- *PacBio HiFi simulated long reads:* These are read generated using the PBSIM3 PacBio read simulator [41]. PBSIM3 generates SAM format data for CLR reads, which is then converted to BAM files and, finally using bioconda [42] pacakge *pbccs*, CCS (Circular Consensus Sequencing) HiFi reads are generated. Simulations were run with a low coverage of 10× and a long read median length 10Kbp; and
- *PacBio HiFi real long reads:* These is a collection of real-world PacBio HiFi reads for *Oryza sativa* (chr 8), downloaded from the PacBio repository [43].

The simulated read data sets allow us to evaluate using a ground-truth (using the coordinate information from SAM file), while the real-world data set is aimed at a real-world application. Simulated reads were generated from real-world whole genomes, downloaded from NCBI GenBank [44], for eight different organisms ranging from bacterial to eukaryotic species (listed in Table II). For a subset of these genomes (with the exception of *P. aeurginosa* and *B. splendens*), repeat information was available, and so we masked those inputs using the RepeatMasker tool [38]. For the *P. aeruginosa* and *B. splendens* genomes, since repeat information was not available, we used unmasked inputs.

We used the following two steps to construct the contigs for all the inputs: use the ART sequencing simulator [45] to generate 100bp Illumina short reads; and assemble the short reads using the Minia assembler [18] into contigs.

Test platform: All experiments were conducted on a distributed memory cluster with 9 compute nodes, each with 64 AMD Opteron™ (2.3GHz) cores and 128 GB DRAM. The nodes are interconnected using 10Gbps Ethernet and share 190TB of ZFS storage. The cluster supports OpenMPI (for distributed memory MPI codes) and OpenMP (for shared memory multithreaded codes).

Software configuration: All runs of our software `JEM-mapper` was performed using the following set of parameters as default: $k = 16$ bp; no. trials $T = 30$ (choice explained in Fig. S1 in supplementary section); and a window size of $w = 100$ bp to generate minimizer sketches. In other words, we select a k-mer (of size 16 bp) from a consecutive stretch of w (100) k-mers to be the minimizer (as explained in Section III-B). These minimizers are then added to the corresponding set $M_o(s, w)$ only if they change or if the current minimizer goes out of bounds. Subsequently, to

Input genome		$\mathcal{S}$: Subject statistics (Minia contigs)			$\mathcal{Q}$: Query statistics (HiFi long reads)		
Input	Genome length (in bp)	No. contigs ($\geq$ 500bp)	Total subject size in bp (M)	Contig length (avg.±std.dev)	No. long reads (n)	Total query size in bp (N)	Read length (avg.±std.dev)
E. coli	4,641,652	375	4,510,510	12,398 ± 13,010	4,010	46,005,093	10,205 ± 3,418
P. aeruginosa (unmasked)	6,264,404	461	6,105,989	13,782 ± 19,218	6,122	62,504,041	10,221 ± 3,363
C. elegans	100,286,401	32,940	76,721,376	2,330 ± 2,962	97,103	1,000,011,602	10,205 ± 3,400
D. busckii	118,492,362	39,352	106,278,105	2,713 ± 3,174	103,781	1,058,889,285	10,168 ± 3,412
Human chr 7	159,345,973	46,798	56,547,853	1,209 ± 834	136,357	1,393,462,533	9,612 ± 2,988
Human chr 8	145,138,636	45,470	53,060,920	1,167 ± 876	141,102	1,440,205,836	10,200 ± 3,402
B. splendens (unmasked)	339,050,970	98,160	339,804,114	3,462 ± 4,181	429,520	4,371,221,619	10,177 ± 3,403
O. sativa	373,878,990	111,788	175,228,307	1,568 ± 1,446	309,615	3,012,444,385	8,989 ± 62
Z. mays chr 1	307,014,717	32,411	50,255,597	1,550 ± 1,332	301,607	2,703,145,070	8,995 ± 35
O. sativa chr 8 (real)	28,443,022	9,945	18,416,389	1,851 ± 2,067	532,667	10,458,872,536	19,642 ± 4,246

TABLE II: Input data sets used in our experiments. All contigs were produced by running Minia assembler [18] on simulated short reads. The long reads are either simulated (default) or real (*O. sativa chr 8 (real)*).

generate the JEM sketches (Algorithm 1), we set the interval length same as the segment ℓ bp (for `L2C` 1,000 bp and for `L2L` 2,000 bp) for long reads. The smaller ℓ length for `L2C` is to help capture overlaps with shorter contigs. Setting $\ell = 2,000$ bp for the `L2L` is consistent with other long read mappers [8], [16]. For `L2L`, the minimum number of trials needed to generate a hit (i.e., τ) was set to 15 (out of $T = 30$ trials), to represent a 50% hit rate with the random trials.

B. Evaluation for `L2C` mapping

Once the long reads are generated, we pulled out the two end segments (prefix and suffix) of length $\ell = 1,000$ bp and added them to the query set $\mathcal{Q}$. For comparative evaluation, the two state-of-the-art reference genome mappers that support `L2C` (see Table I) are `Mashmap` [21] and `Minimap2` [16]. Of these two, `Mashmap` tool [21] is a fast reference genome mapper that also uses sketching, and from its implementation we can easily extract the top hit for a query, making it possible to directly compare it with `JEM-mapper`. As for `Minimap2` [16], it follows a more classical seed-and-extend alignment-based approach, but it also benefits from the use of minimizers internally for the seeding step. However, it was not possible to make a direct comparison with its output because it reports multiple hits for each query. Therefore, we focus our comparative evaluation on `Mashmap`. In addition to `Mashmap`, we also implemented the classical `MinHash` scheme (Section III-A0a) by modifying `JEM-mapper` implementation. This allowed us to compare the efficacy of the Jaccard Estimator MinHash scheme over the classical `MinHash` scheme. In all cases, the *same* inputs ($\mathcal{Q}$,$\mathcal{S}$) were provided to both programs—i.e., mapping the end segments of long reads to contigs.

Evaluation methodology: For quality evaluation, we constructed a *benchmark* for all simulated data sets using the coordinate information of the contigs ($\mathcal{S}$) and long reads ($\mathcal{Q}$) mapped back to the full-length reference genome ($\mathcal{G}$). This is illustrated in Fig. 6. More specifically, to determine the ⟨start,end⟩ coordinates of each contig, we mapped the set of contigs to the reference $\mathcal{G}$ using Minimap2 [16]. The coordinates of the long reads are readily available from SAM files (generated by PBSIM3). A given end segment of a long read $e \in \mathcal{Q}$ is said to *map* to a contig $c \in \mathcal{S}$ if and only if its respective ⟨start,end⟩ coordinates overlap over at least k base pairs of the reference genome—as shown in Fig. 6.

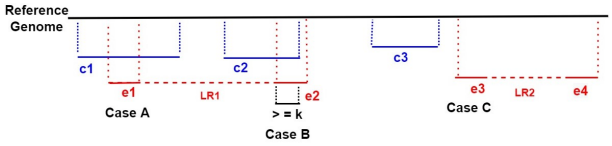


Fig. 6: `L2C`: *Benchmark* cases for when an ending segment of a long read is said to successfully map (Cases A and B) or *not* map (Case C) to a contig. In the figure, two long reads are shown, one with end segments $\langle e1, e2\rangle$ and another with end segments $\langle e3, e4\rangle$.

Let Bench denote the set of all true ⟨read end,contig⟩ mappings. Let Test denote the set of output ⟨read end,contig⟩ mappings produced by one of the test implementations. Then, we classify each ⟨read end e,contig c⟩ pair as:

- **True Positive (TP):** if $\langle e, c\rangle \in$ Test and $\langle e, c\rangle \in$ Bench
- **False Positive (FP):** if $\langle e, c\rangle \in$ Test and $\langle e, c\rangle \notin$ Bench
- **False Negative (FN):** if $\langle e, c\rangle \notin$ Test and $\langle e, c\rangle \in$ Bench
- **True Negative (TN):** if $\langle e, c\rangle \notin$ Test and $\langle e, c\rangle \notin$ Bench

Based on the above four measures, we calculate *precision* as $\frac{TP}{TP+FP}$ and *recall* as $\frac{TP}{TP+FN}$. Note that if an output mapping is a false positive, then by implication it is also a false negative (since there is room for only one best hit in the `L2C` case). But there can be additional false negatives. Therefore the recall values are upper bounded by the precision values in this evaluation scheme.

Qualitative evaluation: Fig. 7 shows the precision (left) and recall (right) values for `JEM-mapper` and `Mashmap`. These results are for the PacBio HiFi simulated long reads. The results show that by and large, our sketch-based implementation is competitive and show comparable quality compared to `Mashmap` in all cases, with both tools producing well over 98% precision for all inputs tested. For the smaller/less complex genomes (*E. coli*, *P. aeruginosa (unmasked)*) `JEM-mapper` produces similar precision values as `Mashmap`. Our scheme produces slightly better precision for all the larger (more complex) inputs. As mentioned earlier, for *B. splendens*, we have used unmasked input and we can see from the Fig. 7 (left), that `JEM-mapper` is outperforming `Mashmap`. Eukaryotic inputs have more repetitive content that may lead to reduced precision and the results show that the strategy to select the best candidate from multiple random trials makes our sketch-based scheme more precise for these more complex inputs.

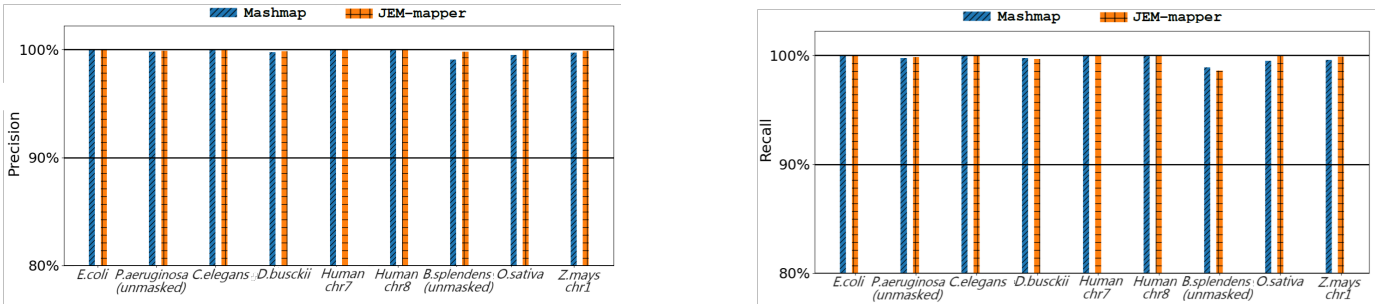


Fig. 7: L2C: Comparative quality evaluation (precision and recall), comparing JEM-mapper and Mashmap, on various PacBio HiFi simulated long reads inputs.

Input	JEM-mapper					Mashmap
	p=4	p=8	p=16	p=32	p=64	t=64
C. elegans	125	54	41	27	24	227
D. busckii	155	85	57	36	33	251
Human chr 7	128	75	39	27	23	260
Human chr 8	119	63	39	25	22	257
B. splendens (unmasked)	519	280	180	145	127	876
O. sativa	557	216	130	83	69	677
Z. mays chr 1	283	164	99	54	44	528
O. sativa chr 8 (real)	420	218	122	91	69	605

TABLE III: L2C: Strong scaling results for JEM-mapper: Shown are the parallel runtimes (in sec) for JEM-mapper as function of the number of processes (p) on various inputs. Also shown are the Mashmap runtimes, which was run on 64 threads (t), as the tool supports only shared memory parallelism.

For all the input datasets, JEM-mapper produces similar recall as compared to Mashmap. The difference between the two tools is marginal in all cases (except for *B. splendens (unmasked)*). Again, both tools produce recall values that are 98% or more for most inputs. We also note that the recall values are very close to the precision values, implying that most of the loss in recall can be attributed to false positive mapping in the top hit. Note that if we are to extend our method to report a fixed number, say top x hits per read, then several of the missing contig hits could possibly be recovered and recall improved.

We also studied the effect of varying the number of trials T on quality results. The key observation from this study is that JEM-mapper can achieve above 97% precision and recall, using fewer number of trials compared to classical MinHash. The results are shown in Supplementary Section A.

Performance evaluation: Next, we evaluate the runtime and parallel performance of JEM-mapper and compare it with state-of-the-art tools. First, we studied the strong scaling behavior of our parallel implementation for JEM-mapper, by varying p from 4 through 64. Table III shows the parallel runtimes for the larger inputs tested. Overall, the parallel runtime reduces with increase in p, demonstrating improving speedups. For instance, on *B. splendens (unmasked)*, the relative speedup (relative to $p = 4$) increases from 1.9× on $p = 8$, to 2.9× on $p = 16$, 3.5× on $p = 32$, and 4.1× on $p = 64$. As the number of processes increases, the work per process also reduces leading to parallel overheads slowly starting to dominate. We have compared our distributed memory implementation results with Mashmap runtimes. Mashmap only supports shared memory parallelism using multithreading. Table III shows the Mashmap runtimes for where the number of threads is set to 64. The results show that JEM-mapper is significantly faster than the Mashmap implementations. In all the input cases, JEM-mapper running in distributed memory mode with $p = 64$ yields higher speedup (ranging from 5.6× to 13×) over MashMap running on the same number of processors (no. threads = 64). Note that in parallel processing, distributed memory setting is expected to have more overheads due to network communication.

Fig. 8a (left) shows the parallel runtime broken down by the individual steps of the JEM-mapper implementation for $p = 16$. It is evident that the dominant step is the query processing time, which includes the time to sketch the queries and search in the hash table and report the hits.

We also closely analyzed the query processing time from the perspective of querying throughput, defined as the number of queries processed per unit time (sec). To calculate this, we included the times for sliding windows on the queries, sketching the queries, and search in $\overline{\mathbb{S}}_{global}$ and report step. Fig. 8b (right) shows the querying throughput for our JEM-mapper implementation, for the larger inputs tested. We observe that this querying throughput scales almost linearly.

Fig. 9 shows the total computation versus communication time for *Human chr 7* and *B. splendens (unmasked)* varying the number of processors from $p = 4$ to $p = 64$. The total computation time includes the I/O time, subject processing time, generating the $\overline{\mathbb{S}}_{global}$ time, and the query processing and search time. As expected, increasing the number of processors increases the total communication overhead, but the overhead stays well under 25% for up to $p = 64$.

Multi-threaded version: The base version of JEM-mapper presented so far uses only distributed memory (MPI) parallelism. To generate further parallelism at the process-level, we also implemented an MPI+OpenMP version of JEM-mapper that uses a two-level parallelism: MPI for distributed memory parallelism across processes, and OpenMP multi-threading [46] for shared memory parallelism within each process using multi-threading. More specifically, we parallelize the steps of generating sketches from the subjects, generating sketches from the queries, and mapping

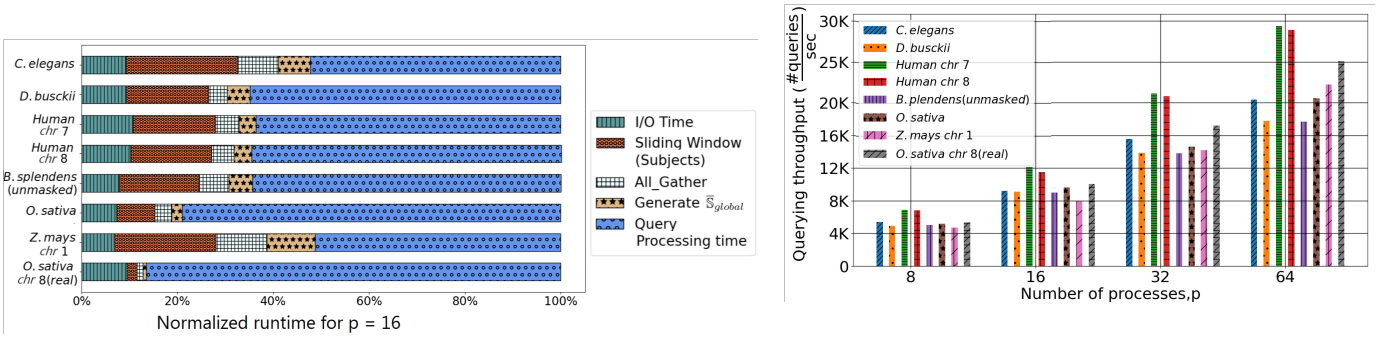


Fig. 8: L2C: (a) Runtime breakdown by steps for JEM-mapper; (b) Querying throughput for JEM-mapper.

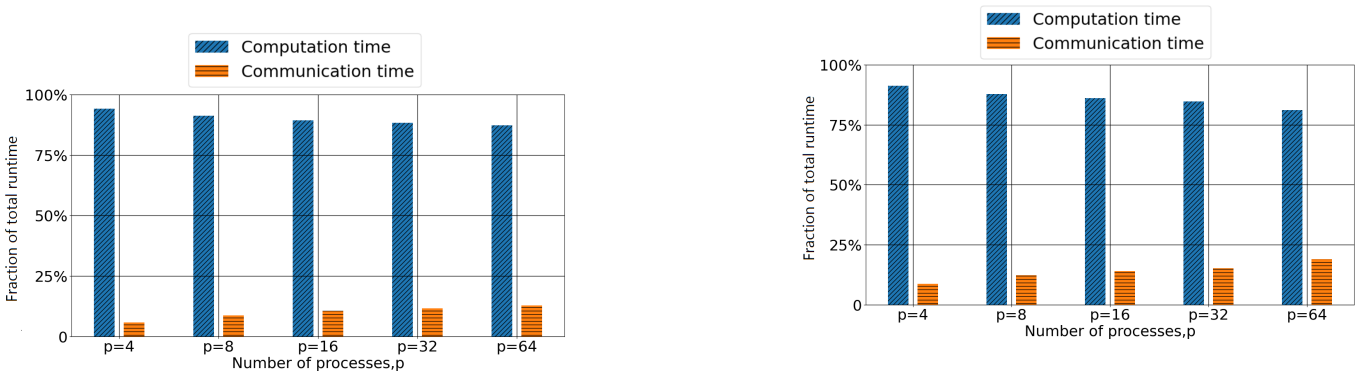


Fig. 9: L2C: The fractions of the total runtime spent in computation (blue) versus communication (orange) for (a) *Human chr 7*; and (b) *B. splendens (unmasked)*. For a scalable parallel execution, computation time should dominate over communication.

them to the global sketch to generate the final hits in the JEM-mapper algorithm (see Section III-D; and Fig. 5).

Table IV shows the run-time results comparing the MPI-only version for 64 processes, against the MPI+OpenMP multi-threaded version for varying number of threads. The results show that the run-times improve with the introduction of multi-threading. For most inputs, we see a strong scaling trend where the parallel multi-threaded run-time keeps halving from the t=1 thread setting to roughly up to 16 threads. For instance, on *O. sativa* the parallel run-time reduces from 557 sec on 1 thread to 32 sec on 16 threads (i.e., roughly corresponding to a 17$\times$ speedup). Linearity in speedup is expected to stop beyond a certain number of threads due to Amdahl's law [47] as the sequential workload starts to dominate. A second observation from the Table IV comes from the columns MPI-only running using p=64 processes, versus the column for (p=4, t=16) also using 64 cores. We see that the multi-threading enabled MPI+OpenMP version consistently outperforming the MPI-only version.

For a closer look at the effect of multi-threading, we examined the total run-time broken down by the different steps. We observed that the time to compute the two major steps—namely, sliding window of subjects, and queries—are significantly reduced in the MPI+OpenMP execution. These results are presented in Supplementary Section B.

We also evaluated JEM-mapper on real-world PacBio HiFi long read data (*Oryza sativa* chromosome 8) and observed that the percent identity of long read ends and the corresponding contigs falls between 95%-100% (more details are presented in Supplementary Section C).

C. Evaluation for L2L mapping

In the L2L use-case, the input consists of only a set of long reads. We have used the same synthetic and real-world data as mentioned in (Section IV-A). For comparative evaluation, we compared against two state-of-the-art long read overlap detection tools, namely Minimap2 [16] and MECAT [8]—as noted in Table I. As mentioned earlier for Minimap2 [16], it follows a classical seed and extend-based approach, but it also benefits from the use of minimizers internally for the seeding step. MECAT [8] is an alignment-free approach that relies on k-mers to detect overlapping candidates and uses a pseudo-linear alignment scoring algorithm to discard false overlap candidates. In all cases, the *same* inputs ($\mathcal{S}$) were provided to all the tools.

Evaluation methodology: For quality evaluation in the L2L use-case, we constructed a *benchmark* using the genomic coordinate information of the long reads. First, all the input long reads ($\mathcal{S}$) were positioned, using the coordinate information from PBSIM3, along the reference genome $\mathcal{G}$. A query long read l_q is said to *map* to a long read $l \in \mathcal{S}$ if and only if their respective <start,end> coordinates overlap in at least 2 Kbp positions of the reference genome—as shown in Fig. 10. This overlap cutoff is derived from the state-of-the-art tools [9], [28].

Let Bench denote the set of all true mappings generated as above. Let Test denote the set of all test mappings generated by our implementation. Then, we can place each distinct <l_q,

Input	(MPI-only)	JEM-mapper running in MPI + multi-threaded mode					
	p=64	p=4, t=1	p=4, t=2	p=4, t=4	p=4, t=8	p=4, t=16	p=4, t=32
C. elegans	24	125	50	28	21	14	13
D. busckii	33	168	73	53	32	20	15
Human chr 7	23	128	72	28	18	15	13
Human chr 8	22	119	53	27	22	14	13
B. splendens (unmasked)	127	519	301	178	101	83	73
O. sativa	69	557	210	120	75	32	30
Z. mays chr 1	44	283	152	89	51	31	28

TABLE IV: L2C: Parallel runtimes (in seconds) for JEM-mapper running in MPI-only mode (with p=64 processes) versus JEM-mapper running with both MPI and multithreading enabled. For the multi-threaded runs, we keep the number of MPI processes fixed at p=4 and vary the number of threads t per process from 1 through 32.

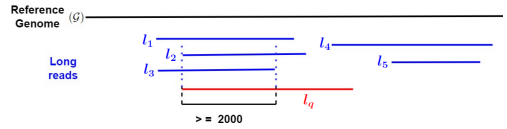


Fig. 10: L2L: In our *benchmark* each query long read l_q is said to map to a subset of long reads in $\mathcal{S}$ ($\{l_1, l_2, l_3\}$in the example) if the overlap is more than 2 Kbp. Note that, based on this criterion, l_q does *not* map to l_4 and l_5.

l> pair into one of the following categories: TP, FP, FN, and TN, and subsequently calculate precision and recall—consistent with our definitions in Section IV-B. Note that since this L2L allows each query long read to map to one or more long reads, a false positive does not necessarily imply a false negative (as it did for L2C).

Qualitative evaluation: Fig. 11 shows the precision (left) and recall (right) values for JEM-mapper, with comparisons to Minimap2 and MECAT. These results are for the PacBio HiFi simulated long reads. The results indicate that, for the most part, JEM-mapper exhibits comparable quality to whichever tool is the best between Minimap2 and MECAT for each input. From Fig. 11 (left) we observe that all precision values are generally well above 88% for masked inputs using JEM-mapper. For all the input data sets, the recall values are between 85% to 99.9% using JEM-mapper. For the smaller/less complex genomes (*E. coli*, *P. aeruginosa (unmasked)*), JEM-mapper produces similar recall values as MECAT.

As for recall, Fig. 11 (right) shows that all three tools perform comparably, with MECAT yielding better recall values for some of the inputs. However, its precision is also lower than the other two tools for many of the inputs. In contrast, JEM-mapper and Minimap2 perform consistently (precision and recall-wise) for the masked inputs. Note that for unmasked input *B. splendens (unmasked)*, Minimap2 attain a very low precision (near 20%) since the repetitive regions of the genome can possibly produce many false-positive overlaps. JEM-mapper overcomes this issue by using the frequency-based heuristic (described in (Section III-C)).

Performance Evaluation: We studied the strong scaling behavior of our parallel implementation for JEM-mapper, by varying p from 4 through 64. Table V shows the parallel runtimes for the larger inputs tested. As with our L2C study, the results show good parallel scaling behavior for JEM-mapper. As a concrete example, for the *O. sativa* input, the relative speedup (relative to $p = 4$) increases from 1.9× on $p = 8$ to 6.3× on $p = 64$.

Table V also compares the parallel runtimes achieved by JEM-mapper to the corresponding multithreaded parallel runtimes achieved by Minimap2 and MECAT. For the smaller inputs, Minimap2 shows faster runtimes than JEM-mapper. However, for the larger inputs, it is evident that the runtimes for Minimap2 significantly increases. Whereas JEM-mapper outperforms both Minimap2 and MECAT on these larger inputs. For instance, for *O. sativa*, JEM-mapper running on distributed memory with $p = 64$ yields 9.9× and 7.2× speedups over Minimap2 and MECAT running on 64 threads, respectively.

Furthermore, comparing Tables III and V, we observe that the L2L runtimes are larger than the L2C runtimes for the corresponding inputs. This is to be expected because the entire long read set is treated as both the subject set and query set for L2L, implying more work.

Fig. 12 shows the parallel runtime broken down by the individual steps of the JEM-mapper implementation for $p = 16$. It is evident that the dominant step is the sliding window of the subjects, which includes the time to sketch the subjects. Note that the fraction of total runtime spent in query processing is significantly smaller compared to the corresponding fractions observed in the L2C case (Fig. 8 (left)). This is because our parallel implementation is optimized to avoid duplicated effort in the sketching of query processing, since $\mathcal{S} = \mathcal{Q}$ for L2L (as noted in Section III-D).

We also studied computation versus communication time and as we have seen earlier for L2C, increasing the number of processors increases the total communication overhead, but the overhead stays well under 10% for even up to $p = 64$ (results summarized in Supplementary Section D).

V. Conclusions

In this paper, we presented a minimizer-based Jaccard estimator sketch-based algorithm for mapping long reads to different types of biological sequences. Our mapping algorithm can be used to map long reads to either a set of partially assembled contigs (from a previous short read assembly), or to the set of long reads themselves. The former application can be used to extend the reach of previously constructed short read assemblies (using long reads), while the latter can be used in *de novo* long read assembly workflows. All our results indicate that we are able to meet the quality in both precision and recall, compared to the corresponding state-of-the-art mapping tools, while providing significant advantages in performance

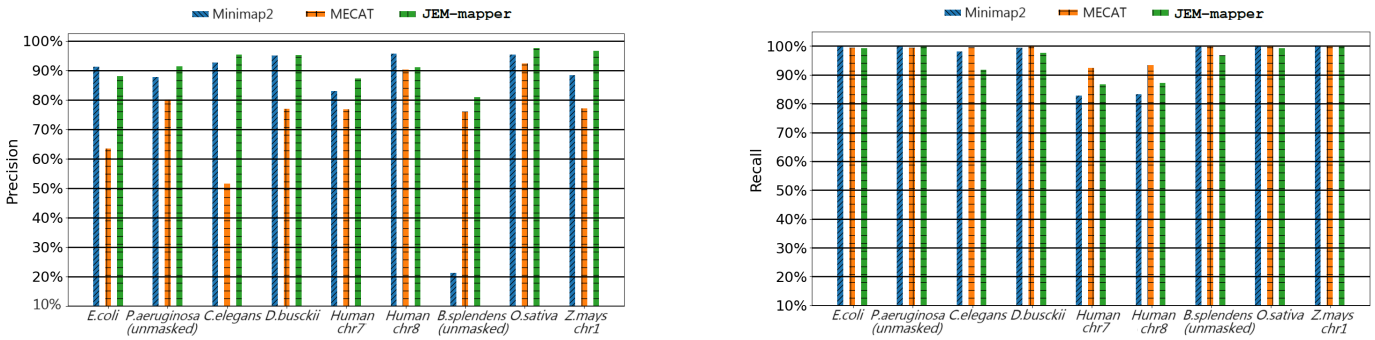


Fig. 11: L2L: Quality results (precision and recall) using PacBio HiFi simulated long reads for different long-read overlappers

Input	JEM-mapper					Minimap2	MECAT
	p=4	p=8	p=16	p=32	p=64	t=64	t=64
C. elegans	485	264	154	101	79	73	498
D. busckii	589	318	218	160	102	74	595
Human chr 7	605	323	205	137	115	60	470
Human chr 8	707	295	178	119	103	71	439
B. splendens (unmasked)	2,521	1,378	760	494	388	2,886	2,931
O. sativa	1,902	1,030	579	376	302	3,008	2,184
Z. mays chr 1	1,894	945	528	350	281	4,123	2,670
O. sativa chr 8 (real)	3,171	1,659	947	608	494	7,479	3,418

TABLE V: L2L: Strong scaling results for JEM-mapper: Shown are the parallel runtimes (in sec) for JEM-mapper as function of the number of processes (p) on various inputs. Also Minimap2 and MECAT runtimes are shown. These tools were run on 64 threads (t), as the tool supports only shared memory parallelism.

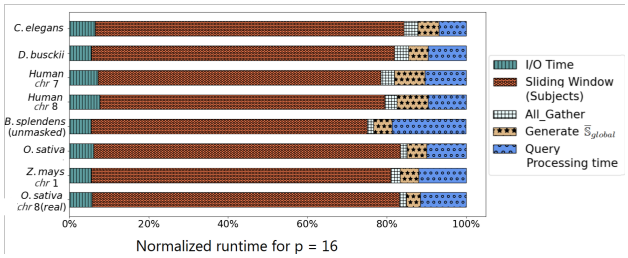


Fig. 12: L2L: Normalized runtime breakdown by steps for JEM-mapper implementation for $p = 16$

speedup. Our open source parallel implementations can be executed on distributed memory platforms (i.e., clusters) which make them suited to scale to large inputs.

This work has opened up multiple avenues for future research, including (but not limited to): i) integration into end-to-end hybrid assembly and scaffolding workflows; and ii) large-scale studies targeting more complex eurakyotic genomes; iii) extension to reference-guided assembly pipelines [14], where either reads are mapped against the reference genome or alternatively contigs or scaffolds are aligned against the reference genome. These use-cases can further enhance the utility of this standalone mapping tool, and help in better harnessing the power of long read sequencing into existing assembly and sequencing workflows.

ACKNOWLEDGMENTS

This research was supported in parts by NSF grants OAC 1910213, CCF 1919122, and CCF 2316160. We thank Dr. Priyanka Ghosh for several discussions during the early stages of the project.

REFERENCES

[1] C. E. Mason and O. Elemento, "Faster sequencers, larger datasets, new challenges," 2012.
[2] D. Deamer, M. Akeson, and D. Branton, "Three decades of nanopore sequencing," *Nature biotechnology*, vol. 34, no. 5, pp. 518–524, 2016.
[3] M. O. Pollard, D. Gurdasani, A. J. Mentzer, T. Porter, and M. S. Sandhu, "Long reads: their purpose and place," *Human molecular genetics*, vol. 27, no. R2, pp. R234–R241, 2018.
[4] T. Hon, K. Mars, G. Young, Y.-C. Tsai, J. W. Karalius, J. M. Landolin, N. Maurer, D. Kudrna, M. A. Hardigan, C. C. Steiner, *et al.*, "Highly accurate long-read hifi sequencing data for five complex genomes," *Scientific data*, vol. 7, no. 1, pp. 1–11, 2020.
[5] P. Morisse, T. LeCroq, and A. LeFeBVre, "Long-read error correction: a survey and qualitative comparison," *BioRxiv*, pp. 2020–03, 2021.
[6] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation," *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.
[7] G. M. Kamath, I. Shomorony, F. Xia, T. A. Courtade, and N. T. David, "HINGE: long-read assembly achieves optimal repeat resolution," *Genome research*, vol. 27, no. 5, pp. 747–756, 2017.
[8] C.-L. Xiao, Y. Chen, S.-Q. Xie, K.-N. Chen, Y. Wang, Y. Han, F. Luo, and Z. Xie, "MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads," *Nature methods*, vol. 14, no. 11, pp. 1072–1074, 2017.
[9] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "BELLA: Berkeley efficient long-read to long-read aligner and overlapper," in *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*, pp. 123–134, SIAM, 2021.
[10] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, vol. 33, no. 6, pp. 623–630, 2015.
[11] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, "hybridSPAdes: an algorithm for hybrid assembly of short and long reads," *Bioinformatics*, vol. 32, no. 7, pp. 1009–1015, 2016.
[12] E. Haghshenas, H. Asghari, J. Stoye, C. Chauve, and F. Hach, "Haslr: Fast hybrid assembly of long reads," *Iscience*, vol. 23, no. 8, p. 101389, 2020.
[13] T. Rahman, O. Bhowmik, and A. Kalyanaraman, "An Efficient Parallel Sketch-based Algorithm for Mapping Long Reads to Contigs," in *2023*

IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 157–166, IEEE, 2023.
[14] H. E. Lischer and K. K. Shimizu, "Reference-guided de novo assembly approach improves genome reconstruction for related species," *BMC bioinformatics*, vol. 18, no. 1, pp. 1–12, 2017.
[15] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, pp. 1–10, 2009.
[16] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.
[17] A. V. Zimin and S. L. Salzberg, "The SAMBA tool uses long reads to improve the contiguity of genome assemblies," *PLoS computational biology*, vol. 18, no. 2, p. e1009860, 2022.
[18] R. Chikhi and G. Rizk, "Space-efficient and exact de bruijn graph representation based on a bloom filter," *Algorithms for Molecular Biology*, vol. 8, no. 1, pp. 1–9, 2013.
[19] C. Jain, S. Koren, A. Dilthey, A. M. Phillippy, and S. Aluru, "A fast adaptive algorithm for computing whole-genome homology maps," *Bioinformatics*, vol. 34, no. 17, pp. i748–i756, 2018.
[20] G. Myers, "Efficient local alignment discovery amongst noisy long reads," in *International Workshop on Algorithms in Bioinformatics*, pp. 52–67, Springer, 2014.
[21] C. Jain, A. Dilthey, S. Koren, S. Aluru, and A. M. Phillippy, "A fast approximate algorithm for mapping long reads to large reference databases," in *International Conference on Research in Computational Molecular Biology*, pp. 66–81, Springer, 2017.
[22] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, "Mash: fast genome and metagenome distance estimation using minhash," *Genome biology*, vol. 17, no. 1, pp. 1–14, 2016.
[23] B. D. Ondov, G. J. Starrett, A. Sappington, A. Kostic, S. Koren, C. B. Buck, and A. M. Phillippy, "Mash screen: high-throughput sequence containment estimation for genome discovery," *Genome biology*, vol. 20, no. 1, pp. 1–13, 2019.
[24] G. Marçais, D. DeBlasio, P. Pandey, and C. Kingsford, "Locality-sensitive hashing for the edit distance," *Bioinformatics*, vol. 35, no. 14, pp. i127–i135, 2019.
[25] L. Coombe, J. X. Li, T. Lo, J. Wong, V. Nikolic, R. L. Warren, and I. Birol, "LongStitch: High-quality genome assembly correction and scaffolding using long reads," *BMC bioinformatics*, vol. 22, pp. 1–13, 2021.
[26] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.
[27] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pp. 21–29, IEEE, 1997.
[28] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.
[29] M. Belbasi, A. Blanca, R. S. Harris, D. Koslicki, and P. Medvedev, "The minimizer jaccard estimator is biased and inconsistent," *bioRxiv*, 2022.
[30] H. Zheng, G. Marçais, and C. Kingsford, "Creating and using minimizer sketches in computational genomics," *Journal of Computational Biology*, vol. 30, no. 12, pp. 1251–1276, 2023.
[31] G. Holley, R. Wittler, J. Stoye, and F. Hach, "Dynamic alignment-free and reference-free read compression," *Journal of Computational Biology*, vol. 25, no. 7, pp. 825–836, 2018.
[32] G. Marçais, D. Pellow, D. Bork, Y. Orenstein, R. Shamir, and C. Kingsford, "Improving the performance of minimizers and winnowing schemes," *Bioinformatics*, vol. 33, no. 14, pp. i110–i117, 2017.
[33] M. Cechova, "Probably correct: rescuing repeats with short and long reads," *Genes*, vol. 12, no. 1, p. 48, 2020.
[34] M. J. Chaisson and G. Tesler, "Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory," *BMC bioinformatics*, vol. 13, no. 1, pp. 1–18, 2012.
[35] R. Guo, Y.-R. Li, S. He, L. Ou-Yang, Y. Sun, and Z. Zhu, "Replong: de novo repeat identification using long read sequencing data," *Bioinformatics*, vol. 34, no. 7, pp. 1099–1107, 2018.
[36] P. S. Schnable, D. Ware, R. S. Fulton, J. C. Stein, F. Wei, S. Pasternak, C. Liang, J. Zhang, L. Fulton, T. A. Graves, *et al.*, "The B73 maize genome: complexity, diversity, and dynamics," *Science*, vol. 326, no. 5956, pp. 1112–1115, 2009.
[37] B. Sosinski, V. Shulaev, A. Dhingra, A. Kalyanaraman, R. Bumgarner, D. Rokhsar, I. Verde, R. Velasco, and A. G. Abbott, "Rosaceaous genome sequencing: perspectives and progress," *Genetics and genomics of Rosaceae*, pp. 601–615, 2009.
[38] N. Chen, "Using repeat masker to identify repetitive elements in genomic sequences," *Current protocols in bioinformatics*, vol. 5, no. 1, pp. 4–10, 2004.
[39] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, "On the representation of de bruijn graphs," in *Research in Computational Molecular Biology: 18th Annual International Conference, RECOMB 2014, Pittsburgh, PA, USA, April 2-5, 2014, Proceedings 18*, pp. 35–55, Springer, 2014.
[40] R. W. Hockney, "Parametrization of computer performance," *Parallel Computing*, vol. 5, no. 1-2, pp. 97–103, 1987.
[41] N. Dierckxsens, T. Li, J. R. Vermeesch, and Z. Xie, "A benchmark of structural variation detection by long reads through a realistic simulated model," *Genome biology*, vol. 22, no. 1, pp. 1–16, 2021.
[42] B. Grüning, R. Dale, A. Sjödin, B. A. Chapman, J. Rowe, C. H. Tomkins-Tinch, R. Valieris, J. Köster, and B. Team, "Bioconda: sustainable and comprehensive software distribution for the life sciences," *Nature methods*, vol. 15, no. 7, pp. 475–476, 2018.
[43] P. Biosciences, "PacBio Real-world HiFi long reads for *O. sativa*." https://downloads.pacbcloud.com/public/dataset/Sequel-IIe-202104/rice/, 2021 (last date accessed: Aug 2022).
[44] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "Genbank," *Nucleic acids research*, vol. 41, no. D1, pp. D36–D42, 2012.
[45] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "Art: a next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, pp. 593–594, 2012.
[46] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.
[47] G. M. Amdahl, "Computer architecture and amdahl's law," *Computer*, vol. 46, no. 12, pp. 38–46, 2013.

Tazin Rahman received a B.S. in computer science and engineering from Bangladesh University of Engineering and Technology, Bangladesh. Currently, Tazin is a Ph.D. candidate in computer science at Washington State University, Pullman, Washington. Her research interests include bioinformatics & computational biology, high-performance computing, and string algorithms.

Oieswarya Bhowmik received a B. Tech in computer science & engineering from West Bengal University of Technology, Kolkata, India; and a M. Tech in computer science & engineering from SRM University, Chennai, India. Currently, Oieswarya is a Ph.D. candidate in computer science at Washington State University, Pullman, Washington. Her research interests include bioinformatics, high performance computing, and graph algorithms.

Ananth Kalyanaraman (Senior Member, IEEE) received his MS and PhD degrees from Iowa State University, Ames, Iowa, in 2002 and 2006, respectively. Ananth is a professor and Boeing Centennial Chair in computer science, at the School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington. He is the Director of the AgAID AI Institute, and holds a joint appointment with Pacific Northwest National Laboratory. His research focuses on developing parallel algorithms and software for data-intensive problems originating in the areas of computational biology and graph-theoretic applications. Ananth serves as an Associate Editor-In-Chief of Journal of Parallel and Distributed Computing, and in the editorial board of IEEE/ACM Transactions on Computational Biology and Bioinformatics, and Parallel Computing.