RESEARCH **Open Access**

Maptcha: an efficient parallel workflow for hybrid genome scaffolding

Oieswarya Bhowmik[1*], Tazin Rahman[1] and Ananth Kalyanaraman[1]

*Correspondence:
oieswarya.bhowmik@wsu.edu

[1] School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA

Abstract

Background: Genome assembly, which involves reconstructing a target genome, relies on scaffolding methods to organize and link partially assembled fragments. The rapid evolution of long read sequencing technologies toward more accurate long reads, coupled with the continued use of short read technologies, has created a unique need for hybrid assembly workflows. The construction of accurate genomic scaffolds in hybrid workflows is complicated due to scale, sequencing technology diversity (e.g., short vs. long reads, contigs or partial assemblies), and repetitive regions within a target genome.

Results: In this paper, we present a new parallel workflow for hybrid genome scaffolding that would allow combining pre-constructed partial assemblies with newly sequenced long reads toward an improved assembly. More specifically, the workflow, called `Maptcha`, is aimed at generating long scaffolds of a target genome, from two sets of input sequences—an already constructed partial assembly of contigs, and a set of newly sequenced long reads. Our scaffolding approach internally uses an alignment-free mapping step to build a ⟨contig,contig⟩ graph using long reads as linking information. Subsequently, this graph is used to generate scaffolds. We present and evaluate a graph-theoretic "wiring" heuristic to perform this scaffolding step. To enable efficient workload management in a parallel setting, we use a batching technique that partitions the scaffolding tasks so that the more expensive alignment-based assembly step at the end can be efficiently parallelized. This step also allows the use of any standalone assembler for generating the final scaffolds.

Conclusions: Our experiments with `Maptcha` on a variety of input genomes, and comparison against two state-of-the-art hybrid scaffolders demonstrate that `Maptcha` is able to generate longer and more accurate scaffolds substantially faster. In almost all cases, the scaffolds produced by `Maptcha` are at least an order of magnitude longer (in some cases two orders) than the scaffolds produced by state-of-the-art tools. `Maptcha` runs significantly faster too, reducing time-to-solution from hours to minutes for most input cases. We also performed a coverage experiment by varying the sequencing coverage depth for long reads, which demonstrated the potential of `Maptcha` to generate significantly longer scaffolds in low coverage settings (1×–10×).

Keywords: Genome assembly, Hybrid scaffolding, Long read mapping, Sketching

Background

Advancements in sequencing technologies, and in particular, the ongoing evolution from high throughput short read to long read technologies, have revolutionized biological sequence analysis. The first generation of long read technologies such as PacBio SMRT [40] and Oxford Nanopore Technologies (ONT) [16] sequencing platforms, were able to break the 10 Kbp barrier for read lengths. However, these technologies also carry a higher cost per base than short read (e.g., Illumina) platforms, and they also have much higher per-base error rate (5–15%) [19, 29, 32, 33, 60]. Recent long read technologies such as PacBio HiFi (High Fidelity) [24, 57, 60] have significantly improved accuracy (99.9%).

Genome assembly, irrespective of the sequencing approach employed, strives to accomplish three fundamental objectives. Firstly, it aims to reconstruct an entire target genome in as few pieces or "contigs" (i.e., contiguous sequences) as possible. Secondly, the goal is to ensure the highest accuracy at the base level. Lastly, the process seeks to minimize the utilization of computational resources. Short read assemblers effectively address the second and third objectives [10, 28, 59], while long read assemblers excel in achieving the first goal [12, 31].

An important aspect of genome assembly is to to maintain correctness in genome reconstruction [3, 53], including composition, continuity, and contiguity. Compositional correctness refers to the correctness of the sequence captured in the output contigs, and is typically measured by the number of misassemblies. Continuity is primarily assessed using metrics such as the N50 value and related measures that show to the extent long stretches of the genome are captured in the contigs correctly, or alternatively how fragmented is an output assembly. In addition to continuity, contiguity across contigs (i.e., the order and orientation of contigs along the unknown genome) is also an important factor, particularly for scaffolding methods.

In the realm of contemporary genome assembly, long read assemblers have adopted the Overlapping-Layout-Consensus (OLC) paradigm [12, 30, 31, 48, 52, 56] and de Bruijn graph approaches [39, 54, 61]. These assemblers utilize advanced algorithms that greatly accelerate the comparison of all-versus-all reads. Many long read assembly tools also perform error correction by representing long reads through condensed and specialized k-mers, such as minimizers [47] and minhashes [52]. This refined representation expedites the identification of overlaps exceeding 2 kb. The most recent long read assemblers are now progressing toward reducing computational resources [8, 9, 41]. However, the assembly of uncorrected long reads introduces challenges, necessitating additional efforts in the form of consensus polishing [11, 36, 58]. Genome assembly polishing is a process aimed at enhancing the base accuracy of assembled contig sequences. Typically, long read assemblers undergo a singular round of long read polishing, followed by multiple rounds of polishing involving both long and short reads using third-party tools [31, 35, 58].

The rapid progress in sequencing technologies is providing extensive quantities of raw genomic data. However, the reconstruction of a complete and accurate genome from these fragmented sequences remains a challenge due to inherent complexities, repetitive regions, and limitations of individual sequencing techniques. Genome assembly heavily relies on scaffolding methods to arrange and link these fragments. In other words, as the conventional assembly step focuses on generating *contigs* that represent contiguous stretches of the target genome, *scaffolding* focuses on ordering and orienting those contigs, as well as filling the gaps between adjacent contigs using any information that is contained in the raw reads. Relying on a single sequencing technology for scaffolding could still result in incomplete or fragmented assemblies [2].

This limitation necessitates hybrid scaffolding approaches that are capable of integrating sequences from multiple sources—sequencing technologies and/or prior constructed draft assemblies.

Hybrid scaffolding: The integration of contigs and long read information for scaffolding purposes can be a promising approach to improve existing genome assemblies [38]. Assemblies generated from short reads are known for their high accuracy, but are often limited by shorter contig lengths, as measured by N50 or NG50. On the other hand, long read sequencing technologies can span larger sections of the genome but are often hindered by higher costs which limit their sequencing coverage depths (to typically under $20\times$ vs. $100\times$ for short reads), and higher error rates compared to short read sequencing complicating *de novo* assembly. Hybrid scaffolding workflows can overcome these limitations by integrating the fragmented assemblies of contigs obtained from short reads and utilizing the long reads to order and orient contigs into longer scaffolds.

In this paper, we visit the *hybrid scaffolding* problem. Given an input set of contigs ($\mathcal{C}$) generated from short reads, and a set of long reads ($\mathcal{L}$), hybrid scaffolders aim to order and orient the contigs in $\mathcal{C}$ using linking information inferred from the long reads $\mathcal{L}$. Such an approach has the advantage of reusing and building on existing assemblies to create improved versions of assemblies incrementally, as more and more long read sequencing data sets are available for a target genome. This workflow can also be easily adapted to scenarios where short reads are available (in place of contigs). In such cases, the short reads can be assembled into contigs prior to the application of our hybrid scaffolder.

Related work: While the treatment of the hybrid scaffolding problem is more recent, there are several tools that incorporate long read information for extending contigs into scaffolds. The concept of genome scaffolding initially emerged in the realm of classical *de novo* genome assembly, as introduced by Huson et al. [27]. This pioneering work aimed to arrange and align contigs utilizing paired-end read information alongside inferred distance constraints. Of the two steps in scaffolding, the alignment step is not only computationally expensive, but it can also lead to loss in recall using traditional mapping techniques. On the other hand, the second step of detecting the true linking information between contig pairs can be prone to false merges, impacting precision—particularly for repetitive genomes.

Over subsequent years, a suite of tools emerged within this classical framework, each striving to refine scaffolding methodologies [2, 15, 18, 20, 37, 43, 49, 50]. For an exhaustive exploration of these methods, refer to the comprehensive review by Luo et al. [38]. Most of these tools utilize alignments of long reads to the contigs of a draft assembly to infer joins between the contig sequences. The alignment information is subsequently used to link pairs of contigs that form successive regions of a scaffold. SSPACE-LongRead produces final scaffolds in a single iteration and has shown to be faster than some of the other scaffolders for small eukaryaotic genomes; but it takes very long runtimes on larger genomes. For instance, SSPACE-LongRead takes more than 475 h to assemble *Z.mays* and for the *Human CHR*1, it takes more than a month. OPERA-LG [21] provides an exact algorithm for large and repeat-rich genomes. It requires significant mate-pair information to constrain the scaffold graph and yield an optimised result. OPERA-LG is not directly designed for the PacBio and ONT data. To construct scaffold edges and link contigs into scaffolds, OPERA-LG needs to simulate and group mate-pair relationship information from long reads.

LRScaf [44] is one of the most recent long read scaffolding tools which utilizes alignment tools like BLASR [6] or Minimap2 [34] to align the long reads against the contigs, and generates alignment information. These alignments form the basis for establishing links between contigs. Subsequently, a scaffold graph is constructed, wherein vertices represent contig ends, and edges signify connections between these ends and associated long reads. This graph also encapsulates information regarding contig orientation and long read identifiers. To mitigate errors and complexities arising from repeated regions and high error rates, LRScaf meticulously refines the scaffold graph. This refinement process involves retaining edges associated with a minimal number of long reads and ensuring the exclusion of edges connecting nodes already present within the graph. The subsequent stage involves navigating linear stretches of the scaffold graph. LRScaf traverses the graph, systematically identifying linear paths until encountering a divergent node, signifying a branching point. At this juncture, the traversal direction is reversed, ensuring exhaustive exploration of unvisited and distinct nodes within the graph. This iterative process continues until all unique nodes are visited, resulting in a complete set of scaffolds from the linear paths within the graph. As can be expected, this rigorous process can be time-consuming, taking hours of compute time even on medium sized genomes (as shown later in the Results).

Another recent tool, ntLink [14] utilizes mapping information from draft assemblies (i.e., contigs) and long reads for scaffolding. This tool employs a minimizer-based approach to first identify the mapped pairs of long reads and contigs, and then uses the mapping information to bridge contigs. However, in their minimizer selection method, non-unique minimizers are discarded. This is done so that repetitive portions within the contigs do not cause false merges in scaffolds. This scheme however limits the lengths of the scaffolds that could be generated by this method (as will be shown in our comparisons).

Contributions

We present a new scalable algorithmic workflow, Maptcha, for hybrid scaffolding on parallel machines using contigs ($\mathcal{C}$) and high-fidelity long reads ($\mathcal{L}$). Figure 1 illustrates the major phases of the Maptcha workflow. Our graph-theoretic approach constructs a contig graph from the mapping information between long reads and contigs, then uses this graph to generate scaffolds. The key ideas of the approach include: a) a sketching-based, alignment-free *mapping* step to build and refine the graph; b) a vertex-centric heuristic called *wiring* to generate ordered walks of contigs as partial scaffolds and c) a final *linking* step to bridge the partial scaffolds and create the final set of scaffolds.

To enhance scalability, we implemented a parallel batching technique for scaffold generation, enabling any standalone assembler to run in a distributed parallel manner while

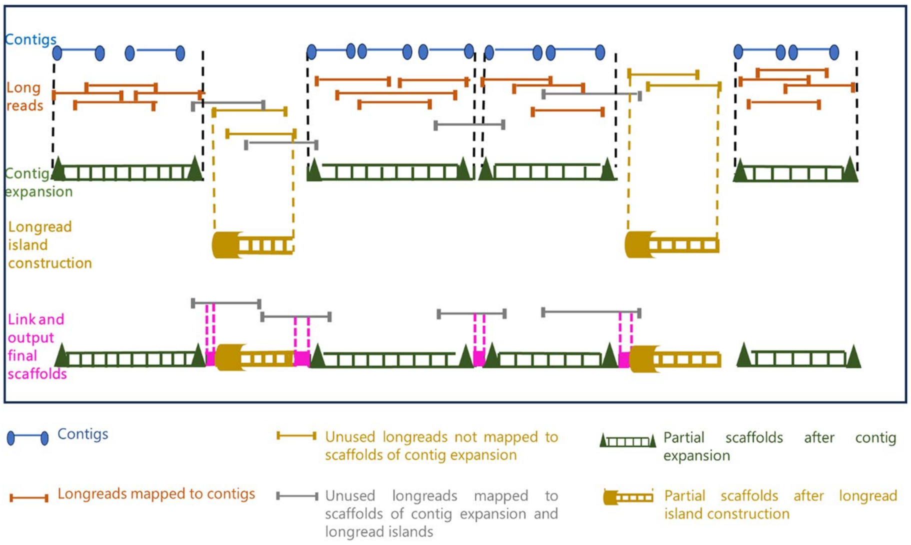
Contigs
Long reads
Contig expansion
Longread island construction
Link and output final scaffolds
Contigs
Longreads mapped to contigs
Unused longreads not mapped to scaffolds of contig expansion
Unused longreads mapped to scaffolds of contig expansion and longread islands
Partial scaffolds after contig expansion
Partial scaffolds after longread island construction

Fig. 1 A schematic illustration of the major phases of the proposed `Maptcha` approach

generating high-quality scaffolds. We use `Hifiasm` [8] as the standalone assembler and `JEM-mapper` [45, 46] for the mapping step.

Our experiments show that `Maptcha` generates longer and more accurate scaffolds than the state-of-the-art hybrid scaffolders `LRScaf` and `ntLink`, while substantially reducing time to solution. For example, the scaffolds produced on the test input *Human chr 7*, the NGA50 of `Maptcha` is around 18× and 330× larger compared to that of `LRScaf` and `ntLink` respectively. `Maptcha` is also significantly faster, reducing time-to-solution from hours to minutes in most cases. Furthermore, comparing `Maptcha` with a standalone long read assembler highlights the benefits of integrating contigs with long reads, resulting in longer scaffolds, fewer misassemblies, and faster runtimes. Coverage experiments (done by varying the sequencing coverage depth for long reads) demonstrated the potential of `Maptcha` to generate considerably longer scaffolds even in low coverage settings (1× to 10×).

The `Maptcha` software is available as open source for download and testing at https://github.com/Oieswarya/Maptcha.git.

Methods

In this section, we describe in detail all the steps of our `Maptcha` algorithmic framework for hybrid scaffolding. Let $\mathcal{C} = \{c_1, c_2, \ldots c_n\}$ denote a set of n input contigs (from prior assemblies). Let $\mathcal{L} = \{r_1, r_2, \ldots r_m\}$ denote a set of m input long reads. Let $|s|$ denote length of any string s. We use $N = \Sigma_{i=1}^{n}|c_i|$ and $M = \Sigma_{i=1}^{m}|r_i|$. Furthermore, for contig c, let $\bar{c}$ denote its reverse complement.

Problem statement: Given $\mathcal{C}$ and $\mathcal{L}$, the goal of our hybrid scaffolding problem is to generate a set of scaffolds $\mathcal{S}$ such that a) each scaffold $S \in \mathcal{S}$ represents a subset of $\mathcal{C}$ such that no two subsets intersect (i.e., $S_i \cap S_j = \emptyset$); and b) each scaffold $S \in \mathcal{S}$ is an ordered sequence of contigs $[c_1, c_2, \ldots]$, with each contig participating in either its direct form c or its reverse complemented form $\bar{c}$. Here, each successive pair of contigs in a scaffold is expected to be linked by one or more long reads $r \in \mathcal{L}$. Intuitively, there are two objectives: i) maximize recall—i.e., to generate as few scaffolds as possible, *and* ii) maximize precision—i.e.,the relative ordering and orientation of the contigs within each scaffold matches the true (but unknown) ordering and orientation of those contigs along the target genome.

Algorithm: The design of the `Maptcha` scaffolding algorithmic framework is broken down into three major phases.

- `contig expansion`: In the first phase, using the contigs as seeds, we aim to extend them on either end using long reads that align with those contigs. This extension step is also designed to detect and connect successive pairs of contigs with direct long read links. This yields the first generation of our partial scaffolds.
- `longread island construction`: Note that not all long reads may have contributed to these partial scaffolds, in particular those long reads which fall in the gap regions of the target genome between successive scaffolds. Therefore, in the next phase, we detect the long reads that do not map to any of the first generation partial scaffolds, and use them to build partial scaffolds corresponding to these long read island regions. This new set of partial scaffolds corresponds to the second generation of partial scaffolds.
- `link scaffolds with bridges`: Finally, in the last phase, we aim to link the first and second generation scaffolds using long reads that serve as bridges between them. This step outputs the final set of scaffolds.

This three phase approach has the following *advantages*. First, it provides a systematic way to progressively combine the sequence information available from the input contigs (which typically tend to be more accurate albeit fragmented, if generated from short reads) to the input long reads (which may be significantly larger in number), in an incremental fashion. Next, this incremental approach also could reduce the main computational workload within each phase that is required for mapping long reads. More specifically, we choose to align long reads either to the contigs or to the generated partial scaffolds wherever possible, and in the process restrict the more time consuming long read to long read alignments only to the gap regions not covered by any of the contigs or partial scaffolds. In this paper, we use the `JEM-mapper`, which is a recently developed fast (parallel) and accurate sketch-based alignment-free long read mapping tool suited for hybrid settings [43, 45]. Finally, by decoupling the contig ordering and orientation step (which is a graph-theoretic problem) from the scaffold generation step (which is an assembly problem), we are able to efficiently parallelize the scaffold generation step. This is achieved through a batching step that splits the input sequences into separate batches to allow the use of any existing standalone long read assembler to generate the final sequence scaffolds. Our framework is capable of leveraging any off-the-shelf long read mapping tool. In this paper, we use `Hifiasm` [8], which is one of the most widely used state-of-the-art long read assembly tool, as our standalone assembler.

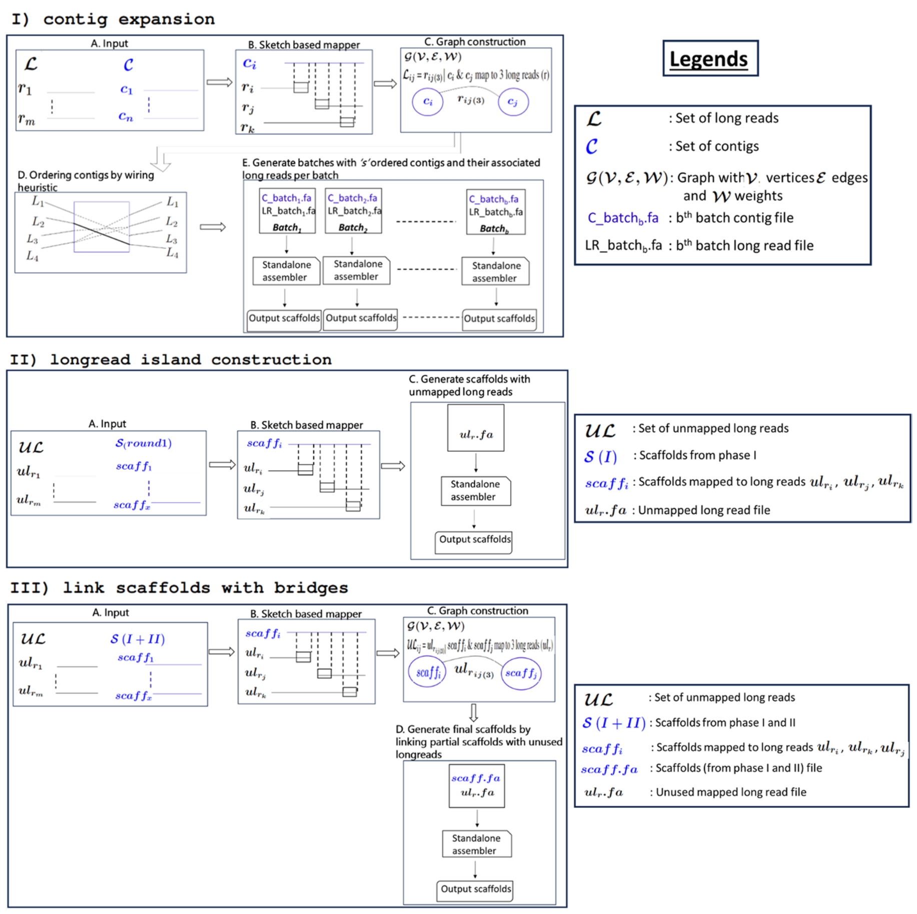
I) contig expansion
A. Input
B. Sketch based mapper
C. Graph construction
D. Ordering contigs by wiring heuristic
E. Generate batches with 's' ordered contigs and their associated long reads per batch
Standalone assembler
Output scaffolds
II) longread island construction
C. Generate scaffolds with unmapped long reads
III) link scaffolds with bridges
D. Generate final scaffolds by linking partial scaffolds with unused longreads
Legends
: Set of long reads
: Set of contigs
: Graph with vertices edges and weights
: bth batch contig file
: bth batch long read file
: Set of unmapped long reads
: Scaffolds from phase I
: Scaffolds mapped to long reads
: Unmapped long read file
: Scaffolds from phase I and II
: Scaffolds (from phase I and II) file
: Unused mapped long read file

Fig. 2 A detailed illustration of the `Maptcha` pipeline showing the different phases and their steps

In what follows, we describe the details of our algorithm for each of the three major phases of our approach. Figure 2 provides an illustration of all the main steps within each of these three phases.

Phase: `contig expansion`The goal of this phase is to enhance contigs by incorporating long reads that have been aligned with them. This process allows for the extension of contigs by connecting multiple ones into a scaffold using the long reads aligned to them, thereby increasing the overall length of the contigs. This is achieved by first mapping the long reads to contigs to detect those long reads that map to contigs, and then use that information to link contigs and extend them into our first generation of partial scaffolds (panel I in Fig. 2).

We use the following definition of a partial scaffold in our algorithm: A *partial scaffold* corresponds to an ordered and oriented sequence of an arbitrary number of contigs $[c_i, c_j, c_k, \ldots]$ such that every consecutive pair of contigs along the sequence are linked by one or more long reads.

Step: Mapping long reads to contigs: For mapping, we use an alignment-free, distributed memory parallel mapping tool, `JEM-mapper` because it is both fast and accurate [45, 46]. `JEM-mapper` employs a sketch-based alignment-free approach that computes a minimizer-based Jaccard estimator (`JEM`) sketch between a subject sequence and a query sequence. More specifically, in a preprocessing step, the algorithm generates a list of minimizing k-mers [47, 51] from each subject (i.e., each contig) and then from that list computes minhash sketches [4] over T random trials (we use $T = 30$ for our experiments). Subsequently, `JEM` sketches are generated from query long reads. Based on these sketches, for each query the tool reports the subject to whom it is most similar. For further details on the methodology, refer to the original paper by Rahman et al. [45].

One challenge of using a mapping tool is that the subject (contigs) and query (long reads) sequences may be of variable lengths, thereby resulting possibly in vastly different sized ground sets of minimizers from which to draw the sketches. However, it is the minimizers from the *aligning region* between the subject and query that should be ideally considered for mapping purposes. To circumvent this challenge, in our implementation we generate sketches only from the two ends of a long read. In other words, our mapping step maps each long read to at most two contigs, one corresponding to each end of that long read. Note that this implies a contig may potentially appear in the mapped set for multiple long reads (depending on the sequencing coverage depth). In our implementation, we used a length of ℓ base pairs ($\ell = 2Kbp$ used in our experiments) from either end of a long read for this purpose. The intuitive rationale is that since we are interested in a scaffolding application, this approach of involving the ends of long reads (and their respective alignment with contigs) provides a way to link two distantly located contigs (along the genome) through long read bridges.

Using this approach in our preliminary experiments, we compared `JEM-mapper` with `Minimap2` and found that `JEM-mapper` yielded better quality results for our test inputs (results summarized in the supplementary section Figure S2).

Step: Graph construction: Let $\mathcal{M}$ denote the mapping output, which can be expressed as the set of 2-tuples of the form $\langle c, r\rangle$—where long read r maps to a contig c—output by the mapper. We use $L_c \subseteq \mathcal{L}$ to denote the set of all long reads that map to contig c, i.e., $L_c = \{r \mid \langle c, r\rangle \in \mathcal{M}\}$. Informally, we refer to L_c as the *long read set corresponding to contig c*.

Using the information in $\mathcal{M}$, and in L_c for all $c \in \mathcal{C}$, we construct an undirected graph $G(V, E)$, where:

- V is the vertex set such that there is one vertex for every contig $c \in \mathcal{C}$; and
- E is the set of all edges of the form (c_i, c_j), such that there exists at least one long read r that maps to both contigs c_i and c_j (i.e., $L_{c_i} \cap L_{c_j} \neq \emptyset$).

Intuitively, each edge is the result of two contigs sharing one or more long reads in their mapping sets. In our implementation, we store the set of long read IDs corresponding to each edge. More specifically, along an edge $(c_i, c_j) \in E$, we also store its long read set $L_{i,j}$ given by the set $L_{c_i} \cap L_{c_j}$. The cardinality of set $L_{i,j}$ is referred to as the "support value" for the edge between these two contigs. Since the vertices of G correspond to contigs, we refer to G as a *contig graph*.

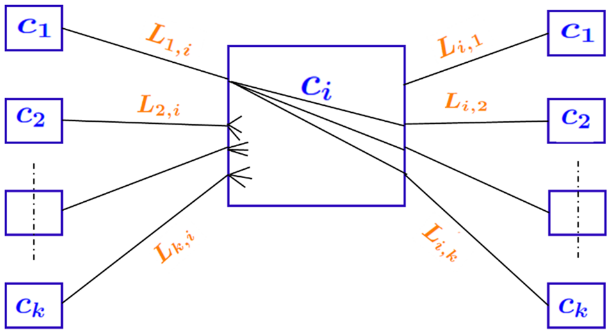

Fig. 3 Illustration of the wiring heuristic, shown centered at a contig vertex c_i. On either side of c_i are shown other contigs (c_1 through c_k) that each have at least one long read common with c_i. These long read sets shared between any contig (say j) and c_i are denoted by $L_{i,j}$ (same as $L_{j,i}$). Out of all possible pairwise connections between the incident edges, the wiring heuristic will select only one edge pair

Next, the graph G along with all of its auxiliary edge information as described above, are used to generate partial scaffolds. We perform this in two steps: a) first enumerate paths in the contig graph that are likely to correspond to different partial scaffolds (this is achieved by our wiring algorithm that is described next); and b) subsequently, generate contiguous assembled sequences for each partial scaffold by traversing the paths from the previous step (this is achieved by using a batch assembly step described subsequently).

Step: Wiring heuristic: Recall that our goal is to enumerate partial scaffolds, where each partial scaffold is a maximal sequence of contiguously placed (non-overlapping) contigs along the target genome. In order to enumerate this set of partial scaffolds, we make the following observation about paths generated from the contig graph $G(V, E)$. A partial scaffold $[c_i, c_{i+1}, \ldots, c_j]$ can be expected to be represented in the form of a path in $G(V, E)$. However, it is important to note that not all graph paths may correspond to a partial scaffold. For instance, consider a branching scenario where a path has to go through a branching node where there are more than one viable path out of that node (contig). If a wrong decision is taken to form paths out of branching nodes, the resulting paths could end up having chimeric merges (where contigs from unrelated parts of the genome are collapsed into one scaffold). While there is no way to check during assembly for such correctness, we present a technique we call *wiring*, as described below, to compute partial scaffolds that reduce the chance of false merges.

The wiring algorithm's objective is one of enumerating maximal acyclic paths in G—i.e., maximality to ensure longest possible extension of the output scaffolds, and acyclic to reduce the chance of generating chimeric errors due to repetitive regions in the genome (as illustrated in Fig. 5). This problem is trivial if each vertex in V has at most two neighbors in G, as it becomes akin to a linked list of contigs, each with one *predecessor* contig and one *successor* contig. However, in practice, we expect several branching vertices that have a degree of more than two (indicating potential presence of repeats). Therefore, finding a successor and/or a predecessor vertex becomes one of a non-trivial path enumeration problem that carefully resolves around branching nodes.

Algorithm: Our wiring algorithm is a linear time algorithm that first computes a "wiring" internal to each vertex, between edges incident on each vertex, and then uses that wired information to generate paths. First, we describe the wiring heuristic.

Step 1: Wiring of vertices: For each vertex $c \in V$ that has at least degree two, the algorithm selects a subset of two edges incident on that vertex to be "wired", i.e., to be connected to form a path through that vertex, as shown in Fig. 3. The two edges so wired determine the vertices adjacent on either side of the current vertex c.

To determine which pair of edges to connect, we use the following heuristic. Let L_i denote the set of long read IDs associated with edge e_i. We then *(hard) wire* two distinct edges e_i and e_j incident on a vertex c, if $L_i \cap L_j \neq \phi$ and it is maximized over all possible pairs of edges incident on c, i.e., $\arg\max_{e_i, e_j \in \mathcal{E}(c)} |L_i \cap L_j|$, where $\mathcal{E}(c)$ denotes all edges incident on c.

The simple intuition is to look for a pair of edges that allows maximum long read-based connectivity in the path flowing through that vertex (contig). This path has the largest *support* by the long read set and is therefore most likely to stay true to the connectivity between contigs along the target genome. All other possible paths through that vertex are ignored. The resulting wired pair of edges $\langle e_i, e_j \rangle$ generated from each vertex c is added in the form of wired edge 3-tuple $\langle c_i, c_j, c \rangle$. We denote the resulting set as $\mathcal{W}$.

There are two special cases to consider here. First, if no pair of edges incident on a vertex c have long reads in common (i.e., $L_i \cap L_j = \phi$ for all pairs of edges incident), then there is no evidence of a link between any pair of edges on that contig. Therefore, our algorithm would *not* wire any pair of edges for that contig. In other words, if a walk (step 2) should reach this vertex (contig), such a walk would terminate at this contig.

As another special case, if a vertex c has degree one, then the wiring task is trivial as there exists only one choice to extend a path out of that contig, c_e, along the edge e attached to that vertex. We treat this as a special case of wiring by introducing a dummy contig c_d to each such vertex with degree one, and adding the tuple $\langle c_d, c_e, c \rangle$ to $\mathcal{W}$.

Note that by this procedure, each vertex c has at most one entry in $\mathcal{W}$. To implement this wiring algorithm, note that all we need is to store the set of long read *IDs* along each edge. A further advantage of this approach is that this is an independent decision made at each vertex, and therefore this step easily parallelizes into a distributed algorithm that works with a partitioning of the input graph.

Step 2: Path enumeration: In the next step, we enumerate edge-disjoint acyclic paths using all the wired information from $\mathcal{W}$. The rationale behind the edge-disjoint property is to reduce the chance of genomic duplication in the output scaffolds. The rationale for avoiding cycles in paths is two-fold—both to reduce genomic duplication due to repetitive contigs, as well as to reduce the chance of creating chimeric scaffolds.

The path enumeration algorithm (illustrated through an example in Fig. 4) works as follows.

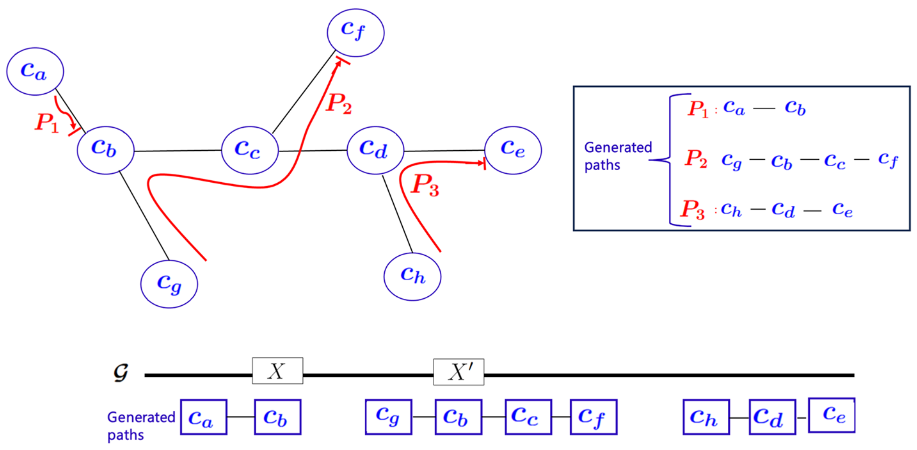

Fig. 4 Edge-disjoint acyclic paths generated from walking the *contig-contig* graph. Also shown below are the likely alignments of the individual paths to the (unknown) target genome $\mathcal{G}$. Here, since the contig c_b appears in two paths, it is likely to be contained in a repetitive region (X, X') as highlighted

(i) Initialize a *visit* flag at all vertices and set them to *unvisited*.

(ii) Initialize a work queue Q of all vertices with degree one (e.g., c_a, c_e, c_f, c_g and c_h in Fig. 4).

(iii) For each vertex $c \in Q$, if c is still unvisited, dequeue c, start a new path at c (denoted by P_c), and grow the path as follows. The edge e incident on c connects c to another vertex, say c_1. Then c_1 is said to be the *successor* of c in the path and is appended to P_c. We now mark the vertex c as visited. Subsequently, the algorithm iteratively extends the path by simply following the wired pairing of edges at each vertex visited along the way—marking each such vertex as visited and stitching together the path—until we arrive at one of the following termination conditions:

Case a) Arrive at a vertex which has chosen a different predecessor vertex: See for example path P_1 truncated at c_b because the wiring at c_b has chosen a different pair of neighbors other than c_a based on long read support, i.e., $\mathcal{W}$ contains $\langle c_g, c_c, c_b\rangle$. In this case, we add the vertex c_b at the end of the current path P_1 and terminate that path.

Case b) Arrive at a vertex that is already visited: This again implies that no extension beyond this vertex is possible without causing duplication between paths, and so the case is handled the same way as Case *a* by adding the visited vertex as the last vertex in the path and the path terminated.

Case c) Arrive at a degree one vertex: This implies that the path has reached its end at the corresponding degree one contig and the path is terminated at this contig.

More examples of paths are shown in Fig. 4.

Provable properties of the algorithm

The above wiring and path enumeration algorithms have several key properties.

Prop1 *Edge disjoint paths: No two paths enumerated by the wiring algorithm can intersect in edges.*

Proof This is guaranteed by the wiring algorithm (step 1), where each vertex chooses only two of its incident edges to be wired to build a path. More formally, by contradiction let us assume there exists an edge e that is covered by two distinct paths P_1 and P_2. Then this would imply that both paths have to pass through at least one branching vertex c such that there exist $\langle e_1, e, c\rangle \in \mathcal{W}$ and $\langle e_2, e, c\rangle \in \mathcal{W}$ (for some $e_1 \neq e_2 \neq e$ all incident on c). However, by construction of the wiring algorithm (step 1) this is *not* possible. □

Prop2 *Acyclic paths: There can be no cycles in any of the paths enumerated.*

Proof This is guaranteed by the path enumeration algorithm described above (step 2). More specifically, the termination conditions represented by the Cases (a) and (b) clip any path before it forms a cycle. By not allowing for cycles, our algorithm prevents including the same contig more than once along a scaffold. This is done so as to prevent chimeric misassemblies of a repetitive contig (for example, repetitive regions X and X' illustrated in Fig. 4. □

Prop3 *Deterministic routing: The path enumeration algorithm is deterministic and generates the same output set of paths for a given $\mathcal{W}$ regardless of the order in which paths are generated.*

Proof This result follows from the fact that the wiring heuristic at each vertex is itself deterministic as well as by the conditions represented by Cases (a) and (b) to terminate a path in the path enumeration algorithm. More specifically, note that each vertex contributes at most one hard-wired edge pair into $\mathcal{W}$ and none of the other edge pair combinations incident on that vertex could lead to paths. Given this, consider the example shown in Fig. 4, of two paths P_1 and P_2 converging onto vertex c_b. Note that in this example, $\langle c_g, c_c, c_b\rangle \in \mathcal{W}$. The question here is if it matters whether we start enumerating P_1 first or P_2 first. The answer is no. In particular, if P_1 is the first to get enumerated, then termination condition Case (a) would apply to terminate the path to end at c_b. Therefore, when P_2 starts, it will still be able to go through c_b. On the other hand, if P_2 is the first path to get enumerated, then c_b will get visited and therefore termination condition Case (b) would apply to terminate P_1 at c_b again. So either way, the output paths are the same. A more detailed example for this order agnostic behavior is shown in S3. This order

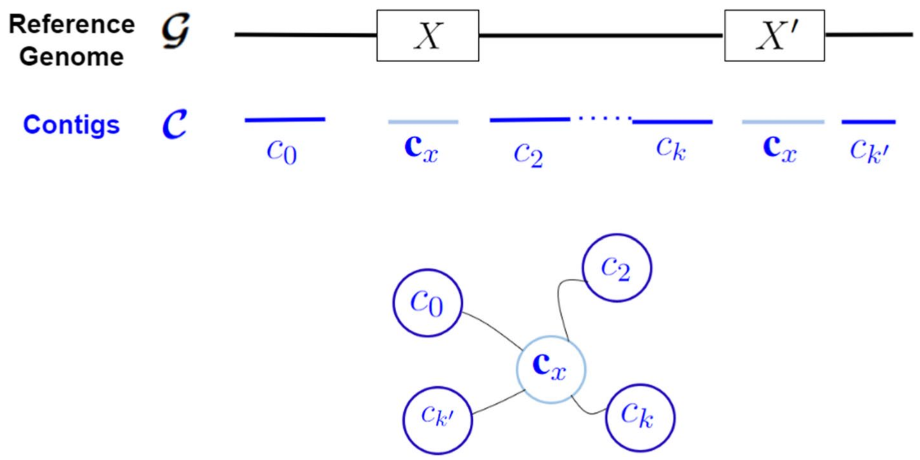

Fig. 5 A case of repeats (X, X') causing cycles branching around contigs

agnostic property allows us to parallelize the path enumeration process without having to synchronize among paths. □

As a corollary to Prop1 (on edge disjoint paths) and Prop2 (on acyclic paths), we now show an important property about the contigs from repetitive regions of the genome and how the wiring algorithm handles those contigs carefully so as to reduce the chances of generating chimeric scaffolds.

Corollary 1 *Let c_x be a contig that is completely contained within a repetitive region. Then this contig can appear as a non-terminal vertex*[1] *in at most one path output by the wiring algorithm.*

Proof Consider the illustrative example in Fig. 5, which shows a contig c_x that maps to a repeat X and its copy X'. In particular, if there is a trail of long reads linking the two repeat copies (from $[c_x, c_2, \ldots c_k, c_x]$), then it could generate a cycle in the graph G. However, based on Prop2, the cycle is broken by the path enumeration algorithm and therefore c_x is allowed to appear as a non-terminal vertex only in at most one of the paths that goes through it. Even if there is no trail of long reads connecting the two repeat regions, the same result holds because of the edge disjoint property of Prop1. □

An important implication of this corollary is that our algorithm is careful in using contigs that fall inside repetitive regions. In other words, if a contig appears as a non-terminal vertex along a path, then its predecessor and successor contigs are those to which this contig exhibits maximum support in terms of its long read based links. While it is not possible to guarantee full correctness, the wiring algorithm uses long read information in order to reduce the chances of repetitive regions causing chimeric scaffolds.

[1] A vertex is said to be *non-terminal* along a path if it appears neither at the start nor the end of that path.

Algorithm 1 Wiring Heuristic

```
Require: $G(V, E)$: Graph
Ensure: Set of wired vertices $\mathcal{W} \subseteq V$
function WiringHeuristic($G$)
    $\mathcal{W} \leftarrow \{\}$
    for all vertices $c \in V$ do
        if degree($c$) $\geq 2$ then
            Let $E(c) \leftarrow$ the set of edges incident on $c$
            Let $L_i \leftarrow$ the set of long read IDs associated with edge $e_i \in E(c)$
            Wiring: Choose 2 edges, $e_i$ and $e_j$ ($c_i \neq c_j$) such that $|L_i \cap L_j| \geq 1$ is maximized over all possible pairs $e_i, e_j \in E(c)$
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{(c_i, c_j, c)\}$
        else if degree($c$) $= 1$ then
            Let $c_d$ be a dummy contig attached to vertex $c$
            Let $c'$ be the vertex connecting to $c$
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{(c_d, c', c)\}$
        end if
    end for
    return $\mathcal{W}$
end function
```

Step: Parallelized contig batch assembly:

As the next step to wiring and path enumeration, we use the paths enumerated to build the output sequence (partial) scaffolds from this phase. To implement this step in a scalable manner, we make a simple observation that the paths enumerated all represent a disjoint set of partial scaffolds. Therefore, we use a partitioning strategy to partition the set of paths into fixed size batches (each containing *s* contigs), so that these independent batches can be fed in a parallel way, into a standalone assembler that can use both the contigs and long reads of a batch to build the sequences corresponding to the partial scaffolds. We refer to this parallel distributed approach as contig batch assembly.

The assembly of each batch is performed in parallel using any standalone assembler of choice. We used `Hifiasm` [8] for all our experiments. By executing contig-long read pairs in parallel batches, this methodology yields one or more scaffolds per batch, contributing to enhanced scalability in assembly processes. Furthermore, the selective utilization of long reads mapped to specific contig batches significantly reduces memory overhead, mitigating the risk of misassemblies that might arise from using the entire long read set which is evident in the results.

This strategy not only reduces memory utilization but also minimizes the potential for misassembly errors that could occur when unrelated sequences are combined.

Phase: `longread island construction`

The first phase of contig expansion, only focuses on expanding contigs using long reads that map on either side. This can be thought of a seed-and-extend strategy, where contigs are seeds and extensions happen with the long reads. However, there could be regions of the genome that are not covered by this contig expansion step. Therefore, in this phase, we focus on constructing "longread islands" to cover these gap regions. See Fig. 1 for an ilustration of these long read islands. This is achieved in two steps:

9a) First we detect all long reads that do not map to any of the first generation partial scaffolds (generated from the contig expansion step). More specifically, we give as input to `JEM-mapper` the set of all unused long reads (i.e., unused in the partial scaffolds) and the set of partial scaffolds output by the contig expansion phase. Any long read that maps to previous partial scaffolds are not considered for this phase. Only those that remain unmapped correspond to long reads that fall in the gap regions between the partial scaffolds.

(b) Next, we use the resulting set of unmapped long reads to build partial scaffolds. This is achieved by inputing the unmapped long reads to `Hifiasm`. The output of this phase represent the second generation of partial scaffolds, each corresponding to a long read island.

Phase: `link scaffolds with bridges`

In the last phase, we now link the first and second generations of partial scaffolds using any long reads that have been left unused so far. The objective is to bridge these two generations into longer scaffolds if there is sufficient information in the long reads to link them. Note that from an implementation standpoint this is same as for contig expansion, where the union of first and generation partial scaffolds serve as the "contigs" and the rest of the unused long reads serve as the long read set.

Complexity analysis

Recall that m denotes the number of input long reads in $\mathcal{L}$, and n is the number of input contigs in $\mathcal{C}$. Let p denote the number of processes used by our parallel program.

Out of the three major phases of `Maptcha`, the `contig expansion` phase is the one that works on the entire input sets ($\mathcal{L}$ and $\mathcal{C}$). The other two phases work on a reduced subset of long reads (unused by the partial scaffolds of the prior scaffolds) and the set of partial scaffolds (which represents a smaller size compared to $\mathcal{C}$). For this reason, we focus our complexity analysis on the `contig expansion` phase.

In the `contig expansion` phase we have the following steps:

(i) *Mapping long reads to contigs*: `JEM-mapper` [45] is an alignment-free distributed memory parallel implementation and hence processes load the long reads and contigs in a distributed manner. The dominant step is sketching the input sequences (long reads or contigs). Given that the number of long reads is expected to be more than the number of contigs (due to sequencing depth), the complexity can be expressed as $O(\frac{m\ell_l T}{p})$, where ℓ_l is average long read length and T denotes the number of random trials used within its minhash sketch computation.

(ii) *Graph construction*: Let the list of mapped tuples $\langle c, r\rangle$ from the previous step contain $\mathcal{T}$ tuples. These $\mathcal{T}$ tuples are used to generate the contig graph by first sorting all the tuples by their long read IDs to aggregate all contigs that map to the same ID. This can be achieved using an integer sort that scans the list of tuples linearly and inserts into a lookup table for all long read IDs—providing a runtime of $O(m + \mathcal{T})$ time. Next, this lookup table is scanned one long read ID at a time, and all contigs in its list are paired with one another to create all the edges corresponding to that long read. The runtime of this step is proportional to the output graph size ($G(V, E)$), which contains n vertices (one for each contig), and $|E|$ is the number of edges corresponding to all contig pairs detected. Our implementation performs this graph construction in a multithreaded mode.

(iii) *Wiring heuristic*: For the wiring step, each node detects a pair of edges incident on it that has the maximum intersection in the number of long read IDs. This can be achieved in time proportional to $O(d^2)$ where d is the average degree of a vertex. The subsequent step of path enumeration traverses each edge at most once. Since both these steps are parallelized, the wiring heuristic can be completed in $O(\frac{nd^2+|E|}{p})$ time.

(iv) *Contig batch assembly*: The last step is the contig batch assembly, where each of the set of enumerated paths are partitioned into b batches, and each batch is individually assembled (using `Hifiasm`). As this step is trivially parallelizable, this step takes $O\left(\frac{b\times a}{p}\right)$ time, where a is the average time taken for assembling any batch.

In our results, we show that the `contig expansion` phase dominates the overall runtime of execution (shown later in Fig. 6).

The space complexity of `Maptcha` is dominated by the size to store the input sequences and the size of the contig graph—i.e., $O(N + M + n + |E|)$.

Results

Experimental setup

Test inputs: For all our experiments, we used a set of input genomes (from various families) downloaded from the NCBI GenBank [1]. These genome data sets are summarized for their key statistics in Table 1. Using the reference for each genome, we generated a set of contigs and a set of long reads as follows. The set of test input contigs ($\mathcal{C}$) were generated by first generating and then assembling a set of Illumina short reads using the ART Illumina simulator [26], with 100× coverage and 100bp read length. The reads generated for our experiments do not have paired-end information. For short read assembly, we used the Minia [10] assembler. As for the set of test long reads ($\mathcal{L}$), we used the Sim-it PacBio HiFi simulator [17], with a 10× coverage and long read median length 10Kbp. Furthermore, note the length divergences in both $\mathcal{C}$ and $\mathcal{L}$.

As a real-world dataset, we used a draft assembly of contigs and a set of real-world long reads available for *Hesperophylax magnus* (*H. magnus*)—a caddisfly genome [42]. The corresponding data was downloaded from NCBI GenBank, as reported in Olsen et al. [42]. All GenBank accession IDs are shown in supplementary Table S2. Since the original reads used in this assembly were not available, we simulated the short reads from this assembly and assembled them into contigs using Minia. For long reads, we used the real HiFi long reads provided by Hotaling et al. [25]. This HiFi dataset consists of a median read length of 11.4 Kbp with a 22.8× coverage. The long reads were generated using the PacBio Sequel II system with SMRTcell.

Qualitative evaluation: To evaluate the quality of of the scaffold outputs produced by `Maptcha`, we used Quast [23] which internally maps the scaffolds against the target reference genome and obtains key qualitative metrics consistent with literature, such as NG50 and NGA50 lengths, largest alignment length, number of misassemblies, and genome fraction (the percentage of genome recovered by the scaffolded assembly) (Table 3). For a comparative evaluation against a state-of-the-art hybrid scaffolder,

Table 1 Input data sets used in our experiments. All inputs were downloaded from NCBI GenBank [1]. For all the inputs the contigs were generated from simulated short reads using the Minia assembler, and the long reads also were simulated (as described under Experimental setup), except for *H. magnus*. For the *Hesperophylax magnus* (*H. magnus*) genome input—a type of a caddisfly—the estimated genome size is reported to be 1.2 Gbp [42]. For this input we used real-world HiFi long reads downloaded from NCBI Genbank. All accession numbers are provided in the supplementary table Table S2

Input genome		**$\mathcal{C}$: Contig statistics (Minia contigs)**			**$\mathcal{L}$: Long read statistics (HiFi simulated reads)**		
Genome	**Genome length (in bp)**	**No. contigs (≥ 1,000bp) (n)**	**Total length in bp (N)**	**N50 in bp**	**No. long reads (m)**	**Total length in bp (M)**	**Read length (avg.±std. dev)**
E. coli	4,641,652	330	4,499,289	23,328	4,541	46,312,093	10,198 ± 3420
P. aeruginosa	6,264,404	370	6,093,817	30,162	6,122	62,511,066	10,210 ± 3,72
C. elegans	100,272,607	18,054	77,564,568	7,268	98,103	1,001,075,296	10,204 ± 3396
T. crassiceps	107,053,072	4,976	90,108,186	52,334	104,679	1,065,911,598	10,182 ± 3390
D. busckii	118,492,362	28,505	99,697,088	4,622	123,781	1,258,903,798	10,170 ± 3406
Human chr 7	159,345,173	34,921	96,494,010	3,354	156,285	1,591,064,955	10,180 ± 3390
N. polychloros	398,112,776	91,698	220,924,212	2,743	389,895	3,973,622,942	10,191 ± 3398
C. septempunctata	398,868,586	57,938	136,903,130	2,563	390,797	3,981,681,897	10,188 ± 3398
B. splendens	441,388,503	73,785	322,195,214	6,451	432,230	4,404,143,269	10,189 ± 3393
M. florea	485,103,743	86,826	218,502,680	2,740	474,914	4,836,563,328	10,184 ± 3393
H. aestivaria	501,713,186	76,767	154,096,503	2,118	491,533	5,005,317,758	10,183 ± 3397
H. magnus	(1.2 Gbp est.)	91,837	656,731,831	1713	2,436,589	28,013,062,204	11,496 ± 720

we compared the quality as well as runtime performance of `Maptcha` against that of `LRScaf` [44] and `ntLink` [13, 14].

Qualitative evaluation

Scaffold quality

First, we report on the qualitative evaluation for `Maptcha`, for its hybrid assembly quality. Table 2 shows the quality by the various assembly metrics alongside the quality values for `LRScaf` and `ntLink`—for all the inputs tested. The same inputs were provided into all the tools. Note that the assembly quality for `Maptcha` shown are for the final output set of scaffolds produced by the framework (i.e, after its `link scaffolds with bridges` phase).

We observe from Table 2 that `Maptcha` is able to produce a high quality scaffolded assembly, reaching nearly 99% genome coverage with high NG50, NGA50 and largest alignment lengths, low misassembly rate, and a near-perfect (1.0) duplication ratio, for

Table 2 Qualitative comparison of the output scaffolds generated by the different tools on the different inputs. All statistics shown are for the final output scaffolds, and were calculated using the Quast tool. Symbol — indicates that the corresponding runs did not complete within 6 h; and ∗ indicates that no NGA50 were obtained from the Quast results. Bold face values show the best results for any input

Input	Method	NG50	NGA50	Largest Alignment (bp)	Genome Coverage %	Missassemblies	Duplication Ratio
E. coli	`LRScaf`	4,499,158	3,541,973	3,541,973	97.12	**0**	1.03
	`ntLink`	**4,653,131**	4,495,406	4,495,406	96.96	1	1.04
	`Maptcha`	4,641,652	**4,641,652**	**4,641,652**	**99.87**	**0**	**1**
P. aeruginosa	`LRScaf`	3,780,771	3,703,369	3,703,369	97.96	1	1.03
	`ntLink`	4,734,796	3,640,841	3,640,841	97.21	1	1.03
	`Maptcha`	**6,264,404**	**6,264,404**	**6,102,781**	**98.69**	**0**	**1**
C. elegans	`LRScaf`	1,080,091	1,080,091	5,019,647	83.68	46	1.23
	`ntLink`	818,712	435,226	3,007,944	77.34	35	1.25
	`Maptcha`	**15,736,218**	**15,736,218**	**17,718,942**	**99.81**	**11**	**1**
T. crassiceps	`LRScaf`	171,559	128,433	1,108,210	86.86	501	1.02
	`ntLink`	564,569	478,734	3,925,088	84.26	**20**	1.05
	`Maptcha`	**4,805,993**	**2,716,011**	**12,352,928**	**98.68**	21	**1**
D. busckii	`LRScaf`	2,597,298	1,129,460	13,199,135	91.41	42	1.11
	`ntLink`	1,598,290	335,476	10,477,172	84.75	17	1.13
	`Maptcha`	**13,533,287**	**13,432,400**	**23,381,820**	**95.69**	**0**	**1.01**
Human chr 7	`LRScaf`	4,499,158	4,499,158	3,541,973	97.12	**0**	1.03
	`ntLink`	872,912	245,860	4,336,964	60.69	33	1.44
	`Maptcha`	**81,166,983**	**81,144,021**	**81,144,021**	**99.71**	2	**1**
N. polychloros	`LRScaf`	–	–	–	–	—	
	`ntLink`	449,932	66,222	2,556,669	55.50	273	1.61
	`Maptcha`	**13,933,406**	**13,338,748**	**18,337,428**	**99.92**	**15**	**1**
C. septempunctata	`LRScaf`	—	–	–	–	–	
	`ntLink`	81,127	∗	2,243,121	34.31	236	1.83
	`Maptcha`	**24,570,419**	**21,121,362**	**40,568,023**	**99.95**	**25**	**1**
B. splendens	`LRScaf`	—	—	—	–	–	
	`ntLink`	890,090	548,569	6,269,807	73.08	190	1.15
	`Maptcha`	**18,757,076**	**16,788,131**	**31,409,892**	**99.01**	**54**	**1.1**
M. florea	`LRScaf`	—	–	–	—	–	
	`ntLink`	453,387	∗	2,806,031	45.03	266	1.74
	`Maptcha`	**34,041,601**	**15,241,540**	**24,515,726**	**97.91**	**67**	**1**
H. aestivaria	`LRScaf`	—	–	—	—	–	
	`ntLink`	37,703	∗	901,904	30.73	441	2.08
	`Maptcha`	**18,646,319**	**9,640,658**	**29,935,333**	**99.83**	**34**	**1**

all the test inputs. These results are substantially better than the output quality produced by the two state-of-the-art tools `LRScaf` and `ntLink`. For smaller genomes such as *E. coli* and *P. aeruginosa*, both `LRScaf` and `ntLink` yield competitive results with `Maptcha`. However, as the genome sizes increase, the assemblies produced by `ntLink` and `LRScaf` become more fragmented. For instance, on *T. crassiceps*, the NGA50 value for `Maptcha` is about 21× and 5.6× larger compared to that of the value for `LRScaf` and `ntLink` respectively. Whereas for *Human chr 7*, the NGA50 of `Maptcha` is around 18× and 330× larger compared to that of `LRScaf` and `ntLink` respectively.

In terms of misassemblies, all three tools produce misassemblies, however to varying degrees, with Maptcha in general producing the fewest number of misassemblies over nearly all the inputs. Misassembly rates are influenced by multiple factors, including the genomic repeat complexity, baseline contiguity, genome fraction, and duplication ratio. In particular, repetitive sequences can significantly impact assembly accuracy and increase misassemblies [5, 7, 22, 55]. While the number of misassemblies produced by ntLink and Maptcha are comparable for inputs such as *P. aeruginosa* and *T. crassiceps*, as the genome size and complexity increase, there is a notable rise in the number of misassemblies with ntLink. As for duplication ratio as well, Maptcha produces scaffolds which have almost no duplication (i.e., ratio is close to 1) in nearly all inputs, while the other tools show varying degrees of duplication. Maptcha also shows the best performance when it comes to genome fraction, capturing almost 99% or more fraction for all the inputs. In general, these results clearly show that Maptcha is able to outperform both LRScaf and ntLink in all the quality metrics reported.

We further examined the growth of contigs and incremental improvement in assembly quality through the different scaffolding phases of Maptcha. Table 4 shows these results, using NG50 lengths output from these different phases as the basis for this improvement assessment. Supplementary Figure S4 shows the increase across all three phases in log-scale.

As can be seen from Table 4, the initial set of Minia-assembled contigs for larger genomes have NG50 measurements ranging from 1 to 3 Kbp. After the contig expansion phase of Maptcha, a substantial increase in NG50 is observed, often exceeding 200-fold. For instance, inputs such as *C. septempunctata*, *M. florea*, and *H. aestivaria* show a notable increase in NG50 values from around 2 Kbp for the initial contigs to over 400 Kbp post-contig expansion phase. This substantial increase is attributed to the long reads acting as connectors between the shorter contigs, resulting in longer partial scaffolds.

In the subsequent longread island construction phase, there is a modest increase in NG50. However, the primary contribution of this phase is to provide more comprehensive genome coverage in regions not covered by contigs. This phase ensures that gaps left by contigs are filled, thereby enhancing the overall assembly.

The final phase of linking partial scaffolds with remaining long reads in Maptcha results in a noteworthy surge in NG50, up to 1,000× for larger genomes. This phase, similar to the contig expansion phase, shows the greatest increase in NG50 among all phases. The average length of these partial scaffolds is considerably longer, which contributes to this dramatic improvement.

Performance evaluation

Next, we report on the runtime and memory performance of Maptcha and compare that with LRScaf and ntLink. Table 3 shows these comparative results for all inputs tested. All runs with Maptcha were obtained by running it on the distributed memory cluster using $p = 64$ processes—more specifically on 4 compute nodes, each running 16 processes. For both LRScaf and ntLink, we ran them in their multithreaded mode on 64 threads on a single node of the cluster. Note that in parallel computing, distributed memory systems support larger aggregate memory but at the

Table 3 Performance evaluation for our test inputs. Symbol — indicates that these results could not be collected in time on the same system i.e 6 h

Input	Method	Time Taken (in secs)	Peak Memory (in GB)
E. coli	`LRScaf`	**0.13**	**10.87**
	`ntLink`	0.32	16.01
	`Maptcha`	0.33	12.33
P. aeruginosa	`LRScaf`	**0.18**	**11.02**
	`ntLink`	0.22	18.01
	`Maptcha`	0.37	12.11
C. elegans	`LRScaf`	126.18	18.54
	`ntLink`	2.41	19.79
	`Maptcha`	**1.84**	**12.48**
T. crassiceps	`LRScaf`	131.68	18.45
	`ntLink`	3.98	19.79
	`Maptcha`	**2.93**	**12.19**
D. busckii	`LRScaf`	232.58	20.22
	`ntLink`	7.05	19.79
	`Maptcha`	**3.32**	**14.57**
Human chr 7	`LRScaf`	355.8	21.03
	`ntLink`	10.43	19.88
	`Maptcha`	**3.5**	**14.33**
N. polychloros	`LRScaf`	—	—
	`ntLink`	14.3	20.05
	`Maptcha`	**5.8**	**16.01**
C. septempunctata	`LRScaf`	—	—
	`ntLink`	19.01	20.01
	`Maptcha`	**10.45**	**16.23**
B. splendens	`LRScaf`	—	—
	`ntLink`	25.85	21.19
	`Maptcha`	**17.52**	**18.96**
M. florea	`LRScaf`	—	—
	`ntLink`	30.1	22.07
	`Maptcha`	**20.55**	**18.57**
H. aestivaria	`LRScaf`	—	—
	`ntLink`	45.52	22.16
	`Maptcha`	**40.68**	**18.71**

Bold face values show the best results for any input

expense of incurring communication (network) overheads, which do not appear in multithreaded systems running on a single node. However to enable a fair comparison on equivalent number of resources, we tested both on the same number (p) of processes, with `Maptcha` running in distributed memory mode while `LRScaf` and `ntLink` running on shared memory. For all runs reported for the performance evaluation, we ran `Maptcha` with a batch size of 8,192 in the batch assembly step.

The results in Table 3 demonstrate that `Maptcha` outperforms both `LRScaf` and `ntLink` in terms of run-time performance. For instance, on medium-sized inputs such as *C. elegans*, `Maptcha` completes nearly 70× faster than `LRScaf`, reducing the time to solution from over 2 h (`LRScaf`) to 1.8 min (`Maptcha`), whereas `ntLink`

Table 4 The increases in the values of NG50 achieved through the `Maptcha` phases starting from the input contigs to the final scaffolds. For *H. magnus*, N50 values are shown instead

Input genome		**NG50 after each phase of `Maptcha` (in bp)**		
Genome	**Contigs NG50 (in bp)**	`contig expansion`	`longread island construction`	`link scaffolds with bridges`
E. coli	22,175	348,034	357,799	4,448,034
P. aeruginosa	29,539	310,601	329,562	6,101,601
C. elegans	4,481	294,365	294,400	15,736,218
T. crassiceps	40,139	351,834	358,177	4,805,993
D. busckii	3,678	303,316	311,207	3,533,287
Human chr 7	1,667	384,512	493,519	81,116,983
N. polychloros	1,260	325,208	474,756	13,933,406
C. septempunctata	2,563	457,265	599,471	24,570,419
B. splendens	3,998	410,300	411,111	18,757,076
M. florea	2,740	483,063	587,311	44,041,601
H. aestivaria	2,118	439,243	547,441	18,646,319
H. magnus	1,713	681,784	734,318	10,010,993

Table 5 Real-world long read input analysis: Quality and performance comparison of the output scaffolds generated by the different tools on an real-world input *Hesperophylax magnus* (*H. magnus*), a type of a caddisfly. Symbol — indicates that the corresponding runs did not complete within 6 h. Bold face values show the best results for the given input

Input	**Method**	**N50(bp)**	**Largest contig (bp)**	**Total assembly length (bp)**	**# N's (per 100 kbp)**	**Time Taken (in mins)**	**Peak Memory (in GB)**
H. magnus	`LRScaf`	–	–	–	–	–	–
	`ntLink`	83,830	1,056,097	956,804,493	56,074.63	**52.21**	35.32
	`Maptcha`	**10,010,993**	**30,653,099**	**1,208,865,085**	**0**	61.54	**26.35**

takes 2.42 min. For larger genomes like *N. polychloros* and *M. florea*, `Maptcha` is still the fastest. Even though `ntLink` runs in comparable times for some of the inputs, the quality of the scaffolds generated by `Maptcha` is considerably better than that of `ntLink` (as shown in Table 2). For the five largest inputs (out of the 11 simulated test inputs), we could not obtain performance results for `LRScaf` as those runs did not complete within the allotted 6-hour limit of the cluster.

Table 3 also shows the memory used by the three tools for all the inputs. For `Maptcha`, recall that the memory is primarily dictated by the memory needed to produce the batch assemblies (which are partitioned into batches). Due to batching, even though the input genome size is increased, the number of contigs that anchor a batch is kept about the same, ensuring a way to control the memory needed to run large assemblies in a scalable fashion. This is the reason why despite growing input sizes, the peak memory used by `Maptcha` stays approximately steady (under 20 GB).

For the real-world long read dataset used in case of the caddisfly genome input, *H. magnus*, the quality of the scaffolds generated by `Maptcha` surpasses both state-of-the-art tools, as shown in Table 5. `LRScaf` was unable to complete its run within 6 h, and thus its results are not included. `Maptcha` outperforms `ntLink` by producing scaffolds that are 119 times

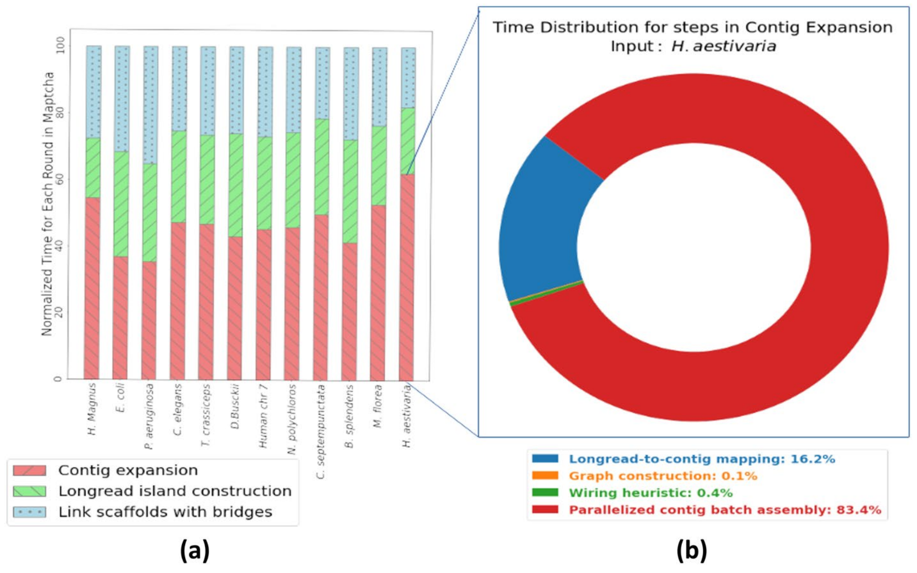

Fig. 6 (**a**) Normalized runtime breakdown for the different rounds of `Maptcha` pipeline for $p = 64$. (b) Normalized runtime breakdown for different steps in the contig expansion round for input *H. aestivaria*.

larger in N50 and 29 times longer in the largest contig metric compared to `ntLink`. Additionally, `Maptcha` generated scaffolds with no gaps, whereas `ntLink` had more than 56k gaps per 100 kbp. Although `ntLink` finished faster, with a runtime of approximately 52 min compared to `Maptcha`'s 61 min, the difference in runtime is marginal when considering the substantial improvement in scaffold quality.

We also compared our scaffolding results with the scaffolds reported in Olsen et al. [42]. We note that the underlying raw reads used in these two studies were different, as the raw reads used in [42] were not available in public as of this writing. In their original work, they report an N50 of 768 Kbp for performing a hybrid assembly using their Illumina (49×) and Nanopore (26×) data. In comparison, our `Maptcha` scaffolder produces a scaffold set with an N50 of 10 Mbp. This represents a significant improvement in scaffolding length—showing promise that when applied to real-world data our `Maptcha` scaffolder is likely to yield longer scaffolds. However further study is needed to validate and compare assembly quality, and also perhaps experimenting with different choices of HiFi long read assemblers.

We also studied the runtime breakdown of `Maptcha` across its different phases. This breakdown is shown normalized for each input in Fig. 6a (left), all running on $p = 64$ processes. It can be observed that the `contig expansion` phase is generally the most time consuming phase, occupying anywhere between 40% to 60% of the runtime, with the other two phases roughly evenly sharing the remainder of the runtime. Figure 6b (right) further shows how the run-time is distributed within the `contig expansion` phase. As can be noted, more than 80% of the time is spent in the batch assembly step, while the remainder of the run-time is spent mostly on mapping.

Effect of batch size on NG50 and run-time: Fig. 7 shows the impact of varying batch sizes on NG50 and processing time, using the *H. aestivaria* genome as an example. Recall that the batch size is the number of contigs that are used to anchor each

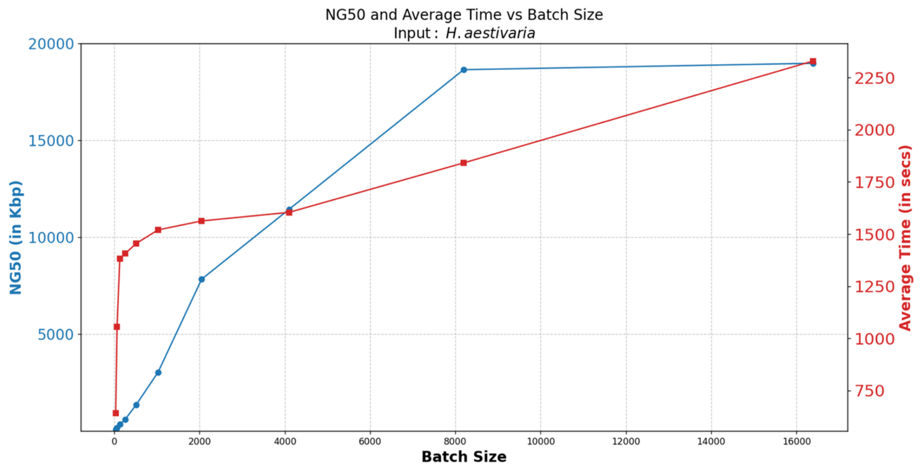

Fig. 7 Effect of batch size on NG50 and average time taken for input *H. aestivaria*

batch along with their respective long reads that map to those contigs. Subsequently, each batch is provided to a standalone assembly (using Hifiasm) to produce the assemblies for the final scaffolds. We experimented with a wide range of batch size, starting from 32, and until 16K. As anticipated, smaller batch sizes exhibit reduced processing times due to the smaller assembly workload per batch. However, if a batch is too small then the resulting assembly quality is highly fragmented (resulting in small NG50 values) as can be observed. Conversely, larger batch sizes necessitate longer processing times (e.g., batch size 32 requiring approximately 280 s, while 8K batch size requires 1,841 s). But the NG50 metric substantially improves—e.g., NG50 size improvement from 93Kbp to 1.8Mbp from a batch size of 32 to 8K.

We found that increasing the batch size from 8K to 16K resulted in a slight increase in NG50 (1.86Mbp to 1.89Mbp), but also a substantial increase in processing time (1,841 s to 2,329 s). Since the increase in NG50 was not significant enough to justify the longer processing time, we decided to use the batch size of 8K for all our tests.

Coverage experiment with Maptcha (hybrid) and Hifiasm (only-LR)

One of the main features of a hybrid scaffolding workflow is that it has the potential to build incrementally on prior constructed assemblies using newly sequenced long reads. This raises two questions: a) how does the quality of a hybrid workflow compare to a standalone long read-only workflow? b) can the information in contigs (or prior constructed assemblies) be used to offset for lower coverage sequencing depth in long reads?

To answer these two questions, we compared the Maptcha scaffolds to an assembly produced directly by running a standalone long read assembler but just using the long reads. For the latter, we used Hifiasm and denote the corresponding runs with the label Hifiasm (only-LR) (to distinguish it from the hybrid configuration in Maptcha). Analysis was performed using different coverages (1x, 2x, 3x, 4x, 8x, and 10x) for the long read data set, for the *H. aestivaria* input, and focusing on performance metrics of NG50, execution time, and peak memory utilization.

The results shown in Table 6 for this experiment, revealed that at lower coverages (1x and 2x), Hifiasm (only-LR) and Maptcha demonstrated relatively comparable

Table 6 Quality and performance evaluation of running `Hifiasm (only-LR)` and `Maptcha` with different coverages of longread on input *H. aestivaria*

Coverage of LR	Method	NG50 (in bp)	NGA50 (in bp)	Misassemblies	Time Taken in mins)	Peak Memory (in GB)
1×	`Hifiasm (only-LR)`	32,586	**32,586**	**9**	18.37	16.36
	`Maptcha`	**33,353**	32,416	16	**5**	**11.51**
2×	`Hifiasm (only-LR)`	30,106	30,106	**11**	21.82	20.54
	`Maptcha`	**39,990**	**32,964**	17	**9.32**	**12.33**
3×	`Hifiasm (only-LR)`	560,317	554,390	**160**	28.03	20.56
	`Maptcha`	**724,586**	**698,713**	189	**14.85**	**12.97**
4×	`Hifiasm (only-LR)`	943,059	917,859	**158**	31.62	20.64
	`Maptcha`	**9,060,428**	**7,881,934**	211	**17.01**	**15.19**
8×	`Hifiasm (only-LR)`	11,122,586	11,122,586	42	59.53	25.68
	`Maptcha`	**11,602,876**	**11,602,876**	**33**	**25.57**	**16.46**
10×	`Hifiasm (only-LR)`	16,455,206	**9,998,605**	39	81.98	29.67
	`Maptcha`	**18,646,319**	9,640,658	**34**	**30.8**	**18.71**

The bold values highlight superior results

performance. However, as the long read coverage increased, `Maptcha` exhibited better NG50 quality over `Hifiasm (only-LR)`, demonstrating the value of adding the contigs in growing the scaffold length. For instance, at 4x coverage, `Maptcha` yielded a considerably longer NG50 (ten-fold increase). The assembly quality becomes comparable for higher coverage settings. These results demonstrate that the addition of prior constructed assemblies can increase the scaffold length compared to long read-only assemblies. However, this value in growing the scaffold length tends to diminish for higher coverage settings—showing that the addition of contigs can be used to offset reduced coverage settings.

Table 6 also shows that a run-time and memory advantage of `Maptcha` over `Hifiasm (only-LR)`. For instance, `Maptcha` was generally between two and four times faster than `Hifiasm (only-LR)` (e.g., on the 10x input, `Maptcha` takes 30 min compared to 81 min taken by `Hifiasm (only-LR)`). Note that internally, `Maptcha` also is using the standalone version of `Hifiasm` to compute its final assembly product. These results show that the `Maptcha` approach of enumerating paths to generate partial scaffolds and distributing those into batches, reduces the overall assembly workload for the final assembly step, without compromising on the quality.

Conclusions

Genome assembly remains a challenging task, particularly in resolving repetitive regions, given its inherently time-intensive nature. In this study, we present `Maptcha`, a novel hybrid scaffolding pipeline designed to combine previously constructed assemblies with newly sequenced high fidelity long reads. As demonstrated, the `Maptcha` framework is able to increase the scaffold lengths substantially, with the NG50 lengths growing by more than four orders of magnitude relative to the initial input contigs. This represents a substantial improvement in genomic reconstruction that comes without any compromise in the accuracy of the genome. Furthermore, our method is able to highlight the value added by prior constructed genome assemblies toward potentially reducing the required coverage depth for downstream long read sequencing. In terms of performance, the `Maptcha` software is a parallel implementation that is able to take advantage of distributed memory machines to reduce time-to-solution of scaffolding. The software is available as open source for testing and application at https://github.com/Oieswarya/Maptcha.git.

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s12859-024-05878-4.

Supplementary Material 1.

Acknowledgements
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.

Author Contributions
Oieswarya Bhowmik was the lead student author who developed the methods, implemented the tool, and conducted experimental evaluation. Tazin Rahman assisted in experimental set up as well as contributed to the incorporation of the mapping step of the approach. Ananth Kalyanaraman conceptualized the project and contributed to algorithm design. All authors contributed to the writing of the manuscript and/or proof reading of the manuscript.

Funding
This research was funded in parts by NSF grants OAC 1910213, CCF 1919122, and CCF 2316160.

Availibility of data and materials
All inputs were downloaded from NCBI GenBank [1]. All accession numbers are provided in the supplementary table Table S2. Our software is available as open source for testing and application at https://github.com/Oieswarya/Maptcha.git.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 1 March 2024 Accepted: 22 July 2024
Published online: 08 August 2024

References
1. Benson DA, Cavanaugh M, Clark K, Karsch-Mizrachi I, Lipman DJ, Ostell J, Sayers EW. Genbank. Nucleic Acids Res. 2012;41(D1):D36–42.
2. Boetzer M, Henkel CV, Jansen HJ, Butler D, Pirovano W. Scaffolding pre-assembled contigs using sspace. Bioinformatics. 2011;27(4):578–9.
3. Bradnam KR, Fass JN, Alexandrov A, Baranay P, Bechner M, Birol I, Boisvert S, Chapman JA, Chapuis G, Chikhi R, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. Gigascience. 2013;2(1):2047-217X.
4. Broder AZ. On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), pages 1997;21–29. IEEE.
5. Cechova M. Probably correct: rescuing repeats with short and long reads. Genes. 2020;12(1):48.
6. Chaisson MJ, Tesler G. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. BMC Bioinform. 2012;13(1):1–18.
7. Chakravarty S, Logsdon G, Lonardi S. Rambler: de novo genome assembly of complex repetitive regions. bioRxiv, pages 2023;2023–05.

8. Cheng H, Concepcion GT, Feng X, Zhang H, Li H. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. Nat Methods. 2021;18(2):170–5.
9. Cheng H, Jarvis ED, Fedrigo O, Koepfli K-P, Urban L, Gemmell NJ, Li H. Haplotype-resolved assembly of diploid genomes without parental data. Nat Biotechnol. 2022;40(9):1332–5.
10. Chikhi R, Rizk G. Space-efficient and exact de bruijn graph representation based on a bloom filter. Algorithms Mol Biol. 2013;8(1):1–9.
11. Chin C-S, Alexander DH, Marks P, Klammer AA, Drake J, Heiner C, Clum A, Copeland A, Huddleston J, Eichler EE, et al. Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data. Nat Methods. 2013;10(6):563–9.
12. Chin C-S, Peluso P, Sedlazeck FJ, Nattestad M, Concepcion GT, Clum A, Dunn C, O'Malley R, Figueroa-Balderas R, Morales-Cruz A, et al. Phased diploid genome assembly with single-molecule real-time sequencing. Nat Methods. 2016;13(12):1050–4.
13. Coombe L, Li JX, Lo T, Wong J, Nikolic V, Warren RL, Birol I. Longstitch: high-quality genome assembly correction and scaffolding using long reads. BMC Bioinform. 2021;22:1–13.
14. Coombe L, Warren RL, Wong J, Nikolic V, Birol I. ntlink: a toolkit for de novo genome assembly scaffolding and mapping using long reads. Curr Protocols. 2023;3(4): e733.
15. Dayarian A, Michael TP, Sengupta AM. Sopra: Scaffolding algorithm for paired reads via statistical optimization. BMC Bioinform. 2010;11(1):1–21.
16. Deamer D, Akeson M, Branton D. Three decades of nanopore sequencing. Nat Biotechnol. 2016;34(5):518–24.
17. Dierckxsens N, Li T, Vermeesch JR, Xie Z. A benchmark of structural variation detection by long reads through a realistic simulated model. Genome Biol. 2021;22(1):1–16.
18. Donmez N, Brudno M. Scarpa: scaffolding reads with practical algorithms. Bioinformatics. 2013;29(4):428–34.
19. Fu S, Wang A, Au KF. A comparative evaluation of hybrid error correction methods for error-prone long reads. Genome Biol. 2019;20:1–17.
20. Gao S, Sung W-K, Nagarajan N. Opera: reconstructing optimal genomic scaffolds with high-throughput paired-end sequences. J Comput Biol. 2011;18(11):1681–91.
21. Gao S, Bertrand D, Chia BK, Nagarajan N. Opera-lg: efficient and exact scaffolding of large, repeat-rich eukaryotic genomes with performance guarantees. Genome Biol. 2016;17(1):1–16.
22. Guo R, Li Y-R, He S, Ou-Yang L, Sun Y, Zhu Z. Replong: de novo repeat identification using long read sequencing data. Bioinformatics. 2018;34(7):1099–107.
23. Gurevich A, Saveliev V, Vyahhi N, Tesler G. Quast: quality assessment tool for genome assemblies. Bioinformatics. 2013;29(8):1072–5.
24. Hon T, Mars K, Young G, Tsai Y-C, Karalius JW, Landolin JM, Maurer N, Kudrna D, Hardigan MA, Steiner CC, et al. Highly accurate long-read hifi sequencing data for five complex genomes. Scientific data. 2020;7(1):1–11.
25. Hotaling S, Wilcox ER, Heckenhauer J, Stewart RJ, Frandsen PB. Highly accurate long reads are crucial for realizing the potential of biodiversity genomics. BMC Genomics. 2023;24(1):117.
26. Huang W, Li L, Myers JR, Marth GT. Art: a next-generation sequencing read simulator. Bioinformatics. 2012;28(4):593–4.
27. Huson DH, Reinert K, Myers EW. The greedy path-merging algorithm for contig scaffolding. J ACM. 2002;49(5):603–15.
28. Jackman SD, Vandervalk BP, Mohamadi H, Chu J, Yeo S, Hammond SA, Jahesh G, Khan H, Coombe L, Warren RL, et al. Abyss 2.0 resource-efficient assembly of large genomes using a bloom: filter. Genome Res. 2017;27(5):768–77.
29. Jain M, Koren S, Miga KH, Quick J, Rand AC, Sasani TA, Tyson JR, Beggs AD, Dilthey AT, Fiddes IT, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. Nat Biotechnol. 2018;36(4):338–45.
30. Kolmogorov M, Yuan J, Lin Y, Pevzner PA. Assembly of long, error-prone reads using repeat graphs. Nat Biotechnol. 2019;37(5):540–6.
31. Koren S, Walenz BP, Berlin K, Miller JR, Bergman NH, Phillippy AM. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome Res. 2017;27(5):722–36.
32. Korlach J, Biosciences P. Understanding accuracy in smrt sequencing. Pac Biosci. 2013;1–9:2013.
33. Laver T, Harrison J, Oneill P, Moore K, Farbos A, Paszkiewicz K, Studholme DJ. Assessing the performance of the oxford nanopore technologies minion. Biomol Detect Quantif. 2015;3:1–8.
34. Li H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. 2018;34(18):3094–100.
35. Lin Y, Yuan J, Kolmogorov M, Shen MW, Chaisson M, Pevzner PA. Assembly of long error-prone reads using de bruijn graphs. Proc Natl Acad Sci. 2016;113(52):E8396–405.
36. 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.
37. Luo J, Wang J, Zhang Z, Li M, Wu F-X. Boss: a novel scaffolding algorithm based on an optimized scaffold graph. Bioinformatics. 2017;33(2):169–76.
38. Luo J, Wei Y, Lyu M, Wu Z, Liu X, Luo H, Yan C. A comprehensive review of scaffolding methods in genome assembly. Brief Bioinform. 2021;22(5):033.
39. Luo R, Liu B, Xie Y, Li Z, Huang W, Yuan J, He G, Chen Y, Pan Q, Liu Y, et al. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience. 2012;1(1):2047-217X.
40. Mason CE, Elemento O. Faster sequencers, larger datasets, new challenges. 2012.
41. Nurk S, Walenz BP, Rhie A, Vollger MR, Logsdon GA, Grothe R, Miga KH, Eichler EE, Phillippy AM, Koren S. Hicanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. Genome Res. 2020;30(9):1291–305.
42. Olsen LK, Heckenhauer J, Sproul JS, Dikow RB, Gonzalez VL, Kweskin MP, Taylor AM, Wilson SB, Stewart RJ, Zhou X, et al. Draft genome assemblies and annotations of agrypnia vestita walker, and hesperophylax magnus banks reveal substantial repetitive element expansion in tube case-making caddisflies (insecta: Trichoptera). Genome Biol Evol. 2021;13(3):evab013.
43. Pop M, Kosack DS, Salzberg SL. Hierarchical scaffolding with bambus. Genome Res. 2004;14(1):149–59.

44. Qin M, Wu S, Li A, Zhao F, Feng H, Ding L, Ruan J. LRScaf: Improving draft genomes using long noisy reads. BMC Genom. 2019;20(1):1–12.
45. Rahman T, Bhowmik O, Kalyanaraman A. An efficient parallel sketch-based algorithm for mapping long reads to contigs. In 2023 IEEE International parallel and distributed processing symposium workshops (IPDPSW), pages 157–166. IEEE, 2023a.
46. Rahman T, Bhowmik O, Kalyanaraman A. An efficient parallel sketch-based algorithmic workflow for mapping long reads. bioRxiv, pages 2023–11, 2023b.
47. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA. Reducing storage requirements for biological sequence comparison. Bioinformatics. 2004;20(18):3363–9.
48. Ruan J, Li H. Fast and accurate long-read assembly with wtdbg2. Nat Methods. 2020;17(2):155–8.
49. Sahlin K, Vezzi F, Nystedt B, Lundeberg J, Arvestad L. Besst-efficient scaffolding of large fragmented assemblies. BMC Bioinformatics. 2014;15(1):1–11.
50. Salmela L, Mäkinen V, Välimäki N, Ylinen J, Ukkonen E. Fast scaffolding with small independent mixed integer programs. Bioinformatics. 2011;27(23):3259–65.
51. Schleimer S, Wilkerson DS, Aiken A. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 76–85, 2003.
52. Shafin K, Pesout T, Lorig-Roach R, Haukness M, Olsen HE, Bosworth C, Armstrong J, Tigyi K, Maurer N, Koren S, et al. Nanopore sequencing and the shasta toolkit enable efficient de novo assembly of eleven human genomes. Nat Biotechnol. 2020;38(9):1044–53.
53. Simao FA, Waterhouse RM, Ioannidis P, Kriventseva EV, Zdobnov EM. Busco: assessing genome assembly and annotation completeness with single-copy orthologs. Bioinformatics. 2015;31(19):3210–2.
54. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol I. Abyss: a parallel assembler for short read sequence data. Genome Res. 2009;19(6):1117–23.
55. Tørresen OK, Star B, Jentoft S, Reinar WB, Grove H, Miller JR, Walenz BP, Knight J, Ekholm JM, Peluso P, et al. An improved genome assembly uncovers prolific tandem repeats in atlantic cod. BMC Genomics. 2017;18:1–23.
56. Vaser R, Sović I, Nagarajan N, Šikić M. Fast and accurate de novo genome assembly from long uncorrected reads. Genome Res. 2017;27(5):737–46.
57. Vollger MR, Logsdon GA, Audano PA, Sulovari A, Porubsky D, Peluso P, Wenger AM, Concepcion GT, Kronenberg ZN, Munson KM, et al. Improved assembly and variant detection of a haploid human genome using single-molecule, high-fidelity long reads. Ann Hum Genet. 2020;84(2):125–40.
58. Walker BJ, Abeel T, Shea T, Priest M, Abouelliel A, Sakthikumar S, Cuomo CA, Zeng Q, Wortman J, Young SK, et al. Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. PLoS ONE. 2014;9(11): e112963.
59. Weisenfeld NI, Yin S, Sharpe T, Lau B, Hegarty R, Holmes L, Sogoloff B, Tabbaa D, Williams L, Russ C, et al. Comprehensive variation discovery in single human genomes. Nat Genet. 2014;46(12):1350–5.
60. Wenger AM, Peluso P, Rowell WJ, Chang P-C, Hall RJ, Concepcion GT, Ebler J, Fungtammasan A, Kolesnikov A, Olson ND, et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. Nat Biotechnol. 2019;37(10):1155–62.
61. Zerbino DR, Birney E. Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome Res. 2008;18(5):821–9.