Title

Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly

Authors

Guidi, Giulia
Selvitopi, Oguz
Ellis, Marquita
et al.

Publication Date

2020-10-19

Peer reviewed

Parallel String Graph Construction and Transitive Reduction for *De Novo* Genome Assembly

Giulia Guidi*†, Oguz Selvitopi†, Marquita Ellis*†, Leonid Oliker†, Katherine Yelick*†, Aydın Buluç*†
*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
†Computational Research Division, Lawrence Berkeley National Laboratory

***Abstract*—One of the most computationally intensive tasks in computational biology is *de novo* genome assembly, the decoding of the sequence of an unknown genome from redundant and erroneous short sequences. A common assembly paradigm identifies overlapping sequences, simplifies their layout, and creates consensus. Despite many algorithms developed in the literature, the efficient assembly of large genomes is still an open problem.**

In this work, we introduce new distributed-memory parallel algorithms for overlap detection and layout simplification steps of *de novo* genome assembly, and implement them in the diBELLA 2D pipeline. Our distributed memory algorithms for both overlap detection and layout simplification are based on linear-algebra operations over semirings using 2D distributed sparse matrices. Our layout step consists of performing a transitive reduction from the overlap graph to a string graph. We provide a detailed communication analysis of the main stages of our new algorithms.

diBELLA 2D achieves near linear scaling with over 80% parallel efficiency for the human genome, reducing the runtime for overlap detection by 1.2–$1.3\times$ for the human genome and 1.5–$1.9\times$ for C.elegans compared to the state-of-the-art. Our transitive reduction algorithm outperforms an existing distributed-memory implementation by 10.5–$13.3\times$ for the human genome and 18–$29\times$ for the C. elegans. Our work paves the way for efficient *de novo* assembly of large genomes using long reads in distributed memory.

I. Introduction

One of the greatest computational challenges for the analysis of high–throughput sequencing DNA fragments (namely *reads*) is *de novo* genome assembly [1]. It consists of aligning and merging redundant and incorrect DNA reads to reconstruct the original genome without any previous knowledge.

Long–read sequencing technologies [2], [3] deliver sequences with an average length of more than 10,000 base pairs (bp). The longer the sequences are read, the better. By using longer sequences, we can assemble through complex genomic repetitions to obtain more precise assemblies that were not possible with short–read technologies [4], [5]. Longer sequences come at the cost of higher error rates, which lead to higher algorithmic complexity and higher computational costs.

The Overlap–Layout–Consensus (OLC) paradigm is the most common assembly strategy for long–read data [6]. The first step (O) is to identify overlaps between reads to build an *overlap* graph. Due to the redundant sequencing and the inherent genome repetitiveness, the second step (L) simplifies the overlap graph and converts it into a *string* graph. A string graph is created from an overlap graph without contained edges and without transitive edges, where the edges represent the overlap *suffix* and not the overlap itself. Nevertheless, a string graph can be created from different source graphs depending on the application. A string graph has the desirable property of collapsing genomic repeats into a single unit [7]. This conversion makes it easier to cluster sections of the graph into *contigs*. A contig is a set of overlapping sequences that together form a consensus region of DNA. Then the consensus step (C) selects the most probable nucleotide sequence for each contig to correct errors in the data. The OLC paradigm benefits from longer reads, since significantly fewer reads are required to cover the genome, limiting the size of the overlap graph.

Our earlier work [8], [9] focused on the implementation of parallel strategies for shared and distributed memory for the *overlap* step. In this respect, BELLA [8] is designed for shared memory and is the first work formulating overlap detection for *de novo* genome assembly using sparse matrices. The distributed memory work [9], which we call diBELLA 1D, performs overlap detection using distributed hash tables.

In this work, we propose a sparse linear algebra centric approach called diBELLA 2D for distributed memory parallelization of overlap and layout phases. By using 2D distributed sparse matrices for both phases, we reduce the need for different data structures in different steps of genome assembly. For the overlap step we formulate the overlap detection as a distributed Sparse General Matrix Multiply (SpGEMM). For the layout step, we present a novel distributed memory algorithm for the *transitive reduction* of the overlap graph. This simplifies the overlap graph and makes it easier to resolve inconsistencies and create contigs.

A linear–time algorithm for the transitive reduction of an overlap graph [10] has been proposed earlier. However, that algorithm is inherently sequential. By contrast, our transitive reduction algorithm is highly parallel. Both the overlap and string graphs are represented as sparse matrices, and the entire transitive reduction algorithm is expressed as operations on sparse matrices. In this direction, our contributions include the design of custom semirings, which are integral to the correctness of the algorithm.

Our results show the scalability of our pipeline for overlap detection plus transitive reduction and show that it achieves near linear scaling with over 80% parallel efficiency for the human genome. Our transitive reduction algorithm outperforms a competing distributed memory algorithm with a speedup of up to $13.3\times$ for the human genome. Our implementation is publicly available at https://github.com/giuliaguidi/diBELLA.2D.

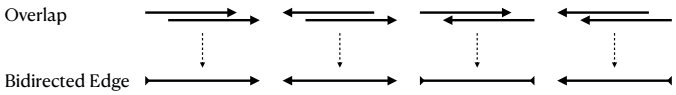

Fig. 1. Overlap to bidirected edge type mapping.

II. Background

A genome consists of one or more DNA molecules that are organized in three-dimensional space as *chromosomes*. The DNA consists of two sequences of nucleotides, called *strands*, which wind around each other and form a double helix. Each strand is a string over the alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ and has a direction. The two strands of a DNA molecule have opposite directions. On opposite strands, A always pairs with T and C with G. One strand defines the *reverse complement* of the other. If $v = \mathrm{ATTCG}$, its reserve complement is $v' = \mathrm{CGAAT}$. The *canonical form* of a DNA sequence v is the lexicographically smaller of v and its reverse-complement v'. In our example, $v = \mathrm{ATTCG}$ is the canonical form.

OLC is the most widely used assembly paradigm for long read data [6], [11]. Its first step consists of identifying overlaps between input sequences. The idea behind the search for overlaps is that two sequences that overlap may originate from adjacent positions on the genome. However, the assembly process is more complex than it might seem at first glance. This is because we cannot be sure that two overlapping sequences actually originate from adjacent positions on the genome due to the repetitiveness of the genome.

For the sequences v_1 and v_2 and their reverse-complements, v_1' and v_2' we can say that v_1 and v_2 have an overlap of length L in base pair (bp) if and only if at least one of these is true:

- the last L bp of v_1 match the first L bp of v_2;
- the last L bp of v_1 match the first L bp of v_2';
- the last L bp of v_1' match the first L bp of v_2;
- the last L bp of v_1' match the first L bp of v_2'.

Given the erroneous sequencing process, in this context an overlap indicates that the two sequences have *mostly* identical base content, so that their pairwise alignment score reaches a quality threshold defined by the overlap detection algorithm.

It is necessary to define four types of overlap, since reads can overlap in a reverse–complement manner and algorithms typically store only the canonical form of k-mers. Additionally, we can define a *contained overlap* as an overlap where the overlapping region of one read is the entire read. An overlap can be called reverse–complement if and only if one of the sequences in the overlap is used in the original direction and the other one is used in the reverse–complement direction. Therefore, an overlap that belongs to the last case is not a reverse–complement overlap, since it is equivalent to the last L bp of v_2 matching the first L bp of v_1. Since the two forward cases are equivalent in theory, we only have three different cases. However, it makes sense to keep the four categories in practice. The correct use of orientation information is crucial during the second and third stages of the OLC algorithm and is essential for the correctness of the final assembly.

Commonly, an indexing data structure, such as a k-mer (i.e., a substring of fixed length k) index table or suffix array, is used to identify an initial set of overlap candidates [11], [12], [6]. Then, as a next step, pairwise alignment is sometimes performed to discard false positives. After the overlaps have been calculated and consolidated from the reads, the next step is the *layout* step, where the goal is to create a graph that encodes how we can assemble sequences to obtain contigs.

A string graph (or matrix) is a graph $G = (V, E)$, where V is the set of sequences and E is the set of overlap *suffixes* between any two vertices. There exists an edge if and only if the respective reads overlap and the weight of this edge is the length of the suffix. For example, for the sequences $v_1 = \mathrm{TACGA}$ and $v_2 = \mathrm{ACGACC}$, their overlap suffix or *overhang* is the portion of v_2 that exceeds the overlap between v_1 and v_2, i.e. $e_{12} = \mathrm{CC}$. Given $G = (V, E)$, where $V = \{v_1, v_2, v_3\}$ and $E = \{e_{12}, e_{13}, e_{23}\}$, we can walk (a) $v_1 \rightarrow v_2 \rightarrow v_3$ using e_{12} and e_{23}, (b) or $v_1 \rightarrow v_3$ using only e_{13}. If we take the weight of the edges into account (i.e., the overhang length), we can see that one of these two paths carries less information than the other and therefore we can mark it as *transitive* and remove it from the string graph.

In *de novo* genome assembly, we want to keep as many overlapping bases as possible for any pair of sequences, so we mark the edges (belonging to a valid path) with longer suffixes (higher weight) as *transitive*. Since the string graph maximizes the overlap length, it can disambiguate short repeats [7]. In our example, e_{13} would be marked as *transitive* and removed since the path $e_{12} \rightarrow e_{23}$ encloses more overlapping bases.

Since we do not know from which strand a certain sequence originates, we want to be able to traverse our graph in both forward and reverse direction. That is, if we consider $G = (V, E)$ in our example above, we want to be able to walk both $v_1 \rightarrow v_2 \rightarrow v_3$ and $v_3' \rightarrow v_2' \rightarrow v_1'$. Using the directed graph representation requires doubling the number of nodes, because for each read we need one vertex representing its *entrance* and one representing its *exit*. Using an undirected graph can avoid this bloat, but it does not guarantee that a particular read will be used in a consistent manner at any point within a single assembly. The use of a bidirected graph (i.e., a graph with a directional head at each end of each edge) [13] solves both of these problems. Figure 1 shows the four types of bidirectional edges that result from the four overlap types described earlier.

If $G = (V, E)$ is a bidirected graph, then a valid path in G is a continuous sequence of edges where each vertex is entered by a head inward and exited by a head outward (unless it is the end of the path) or vice versa. Figure 2 shows two examples of a valid walk ($A \rightarrow B \rightarrow C \rightarrow D$ and $F \rightarrow G \rightarrow H$) and one example of an invalid walk ($E \rightarrow F \rightarrow G$). A walk through the bidirected string graph encodes the way the sequences can be consistently assembled [10].

Given two paths $v_1 \rightarrow v_2 \rightarrow v_3$ and $v_1 \rightarrow v_3$ in a bidirected string graph, the edge $v_1 \rightarrow v_3$ can only be considered to be transitive if the following conditions are satisfied: (a) $v_1 \rightarrow v_2 \rightarrow v_3$ constitutes a valid walk, (b) the two heads next to v_1 have the same orientation, and (c) the two heads next to v_3

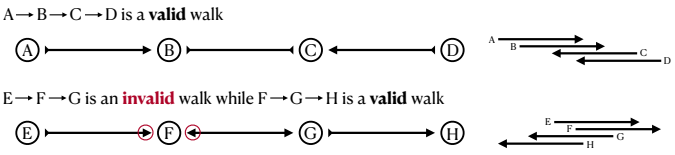

Fig. 2. An example of a valid and invalid walk in a bidirected string graph.

also have the same orientation.

By definition, a string graph can be constructed from various sources, such as an overlap graph (as we present in this paper), k-mers [14] or FM index [15], and Burrows–Wheeler transform (BWT) [7]. These approaches are not invariant to the input properties and often only consider error-free sequences. In reality, long read data with its high error rates and long lengths often make string graph construction impractical for approaches other than those based on overlap graphs.

III. Related Work

In this section we review the literature on overlap detection and transitive reduction, and describe works related to ours.

Myers' transitive reduction algorithm consists of iterating over each node v in the source graph and examining nodes up to two edges away from v to identify all transitive edges that leave or enter v [10]. These edges are then marked for removal, and they are removed after all nodes have been considered.

Li [16], [11] uses a seed–based approach to find overlaps and then uses the Myers algorithm to transitively reduce the string graph. In particular, Li uses *minimizers* (i.e., reduced k-mer representations) for overlap detection and does not perform explicit pairwise alignment on sequences.

Simpson and Durbin [7] use the Ferragina–Manzini index (FM–index) [17] derived from the Burrows–Wheeler transform [18] for overlap detection and transitive reduction. Their algorithm for transitive reduction is similar to that of Myers'. Bonizzoni et al. [15] propose a similar approach using only the FM–index of the input sequence to create a string graph and perform a transitive reduction with a different but equivalent formulation than Myers' [19].

BELLA [8] is the first work proposing to use sparse matrices for overlap detection. A sparse matrix $\mathbf{A}$ is used to indicate the occurrence of k-mers in sequences, and the multiplication of this matrix by its transpose, i.e. $\mathbf{AA}^\mathsf{T}$, is used to detect the overlaps. Given the similarity of our distributed memory design to BELLA, we give more details about it in Section IV.

Our first distributed memory design for overlap detection, diBELLA 1D [9], uses a k-mer based approach and creates and traverses a distributed hash table to find overlapping sequences. This design resembles a 1D SpGEMM using an outer product algorithm without explicit construction of matrices. It does not perform transitive reduction.

PASTIS [20] is inspired by BELLA and computes protein homology search as distributed SpGEMM. Genome assembly and protein homology search are different problems, but both require a computationally expensive all-to-all comparison.

Algorithm 1 The matrix computation in diBELLA 2D.

```
1:  procedure diBELLA 2D
2:      reads ← FastaReader()
3:      k-mers ← KmerCounter()
4:      A  ← GenerateA(reads, k-mers)        ▷ Data matrix
5:      A^T ← Transpose(A)
6:      C ← AA^T                             ▷ Candidate overlap matrix
7:      C ← Apply(C, Alignment())            ▷ Run alignment
8:      R ← Prune(C, AlignmentScoreLessThan(t))
9:      S ← TransitiveReduction(R)           ▷ Algorithm 2
10:     return S
```

Besta et al. [21] present another approach similar to BELLA using distributed SpGEMM to calculate the Jaccard similarity between read sets of different genomes. The main difference is that their software is optimized for the case where the output $|genomes|$-by-$|genomes|$ matrix is dense because it stores the Jaccard similarity between any genome pairs.

Jackson and Aluru [14] present a parallel algorithm for constructing a bidirected string graph from a de Bruijn graph [22] in which the vertices represent k-mers and edges correspond to individual nucleotides whose two vertices have in common. A de Bruijn graph is not suitable for long read data because of the high error rates.

SORA [23] computes transitive reduction of a string graph based on an overlap graph in distributed memory using Apache Spark [24] and the GraphX library [25], which allows parallel computation on distributed graphs in Spark. To the best of our knowledge, SORA is the only other distributed algorithm that computes transitive reduction on overlap graphs, although it was designed for cloud environments.

IV. Proposed Algorithm

In this section we describe the current implementation of our pipeline, focusing primarily on the novel transitive reduction algorithm. To keep the paper self–contained, we first briefly describe the first half of the pipeline, which combines k-mer counting and overlap detection of our prior work [26], [20].

A. Overview

Our algorithm design [8] uses a k-mer–based approach and relies on parallel sparse matrix multiplication for overlap detection as the first step of the OLC paradigm. The outline of our pipeline can be seen in Algorithm 1. We form a $|sequences|$-by-$|k\text{-}mers|$ matrix $\mathbf{A}$ to detect the occurrence of k-mers in sequences, and perform $\mathbf{AA}^\mathsf{T}$ to detect overlaps, resulting in a sparse $|sequences|$-by-$|sequences|$ matrix $\mathbf{C}$. Overlap detection is followed by a computationally intensive seed–and–extend pairwise alignment for all nonzeros in $\mathbf{C}$ using SeqAn [27], a sequence analysis library. If the alignment score of a read pair does not exceed a threshold, then the overlap is discarded and the entry is removed from the matrix. We refer to this resulting output $|sequences|$-by-$|sequences|$ matrix as *overlap* matrix and denote it with $\mathbf{R}$. A transitive reduction algorithm is then run on $\mathbf{R}$ to remove redundant edges and simplify it into a string graph, $\mathbf{S}$.

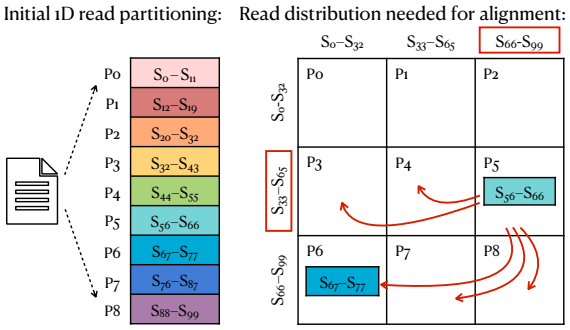

Fig. 3. Read distribution when we read the input (left) and read distribution we need to perform pairwise alignment (right).

B. Data Partitioning

The input to our program is a set of nucleotide sequences in FASTA file format. To ensure load balance, each processor reads an equal–sized independent chunk of this file via parallel MPI I/O. Immediately thereafter, processors begin communicating sequences to create a 2D grid that is consistent with the way the matrices are partitioned among processors. This approach is similar to the one chosen by PASTIS [20].

C. K-mer Selection and Counting

A k-mer based approach calculates the frequency of each k-mer in the input because not all k-mers are useful. K-mers that are usually discarded are (a) k-mers that occur only once in the input (*singletons*) and (b) high frequency k-mers. For more details on k-mer selection, see BELLA paper [8].

diBELLA 2D eliminates singletons using a Bloom filter [28] during k-mer counting and high frequency k-mers that occur at least d times, as in our first implementation. The threshold d is calculated using the approach introduced in BELLA [8], which uses dataset-specific features. In Section V we use d to calculate the communication costs of our algorithm.

Our k-mer counter is similar to that of HipMer [26] and consists of two phases. First we add k-mers to the Bloom filter and then we calculate the frequencies for the filtered k-mers. The processors extract k-mers from their local sequences, hash them, and possibly communicate them with other processors as dictated by the Bloom filter hash function. On the receiver, the incoming k-mers are added to the local Bloom filter; if they already exist, they are added to the local hash table partition. The communication requires an all-to-all exchange and is implemented via MPI Alltoall and MPI Alltoallv.

D. Overlap Detection and Alignment

The local k-mer hash table and the local sequences are used to create a distributed $|sequences|$-by-$|k\text{-}mers|$ matrix $\mathbf{A}$. A nonzero $\mathbf{A}_{ij}$ stores the position of the jth k-mer in the ith sequence. $\mathbf{A}$ is multiplied by $\mathbf{A}^\mathsf{T}$ to obtain the sparse candidate overlap matrix $\mathbf{C} = \mathbf{A}\mathbf{A}^\mathsf{T}$ of dimension $|sequences|$-by-$|sequences|$. In $\mathbf{C}$ each nonzero $\mathbf{C}_{ij}$ stores the number of common k-mers and their positions in the sequence pair i and j. The number of stored positions is a user-defined parameter. For this work we store two k-mer positions for each read pair.

To compute $\mathbf{C}$ we use the distributed SpGEMM in the CombBLAS library [29] and we overload the addition and multiplication operators in SpGEMM with a custom semiring. We overload the multiplication with an assignment by taking the positions of the respective k-mer in two sequences corresponding to $\mathbf{A}_i$ and $\mathbf{A}_j^\mathsf{T}$. We overload the addition operator by incrementing the counter of common k-mers between $\mathbf{A}_i$ and $\mathbf{A}_j^\mathsf{T}$ and storing the positions of another common k-mer in $\mathbf{C}_{ij}$ (i.e., concatenate the results of the multiplication operation) as long as it is smaller than the number of positions to be stored.

CombBLAS relies on 2D Sparse SUMMA algorithm for parallel SpGEMM [30] and it uses a hybrid hash table and heap based algorithm for local multiplication. For matrices, CombBLAS uses a 2D matrix decomposition, so both $\mathbf{A}$ and $\mathbf{C}$ are distributed over a process grid of $\sqrt{P} \times \sqrt{P}$. Observe that in such a distribution a processor may need to align a pair of sequences which it does not have in its partition (Figure 3). Such sequences need to be communicated among respective processors. In this respect, a processor has two possibilities: (a) wait until $\mathbf{C}$ is computed to find out which sequences it would need, and then begin communicating those sequences, or (b) request the full range of sequences it might need once the FASTA input file is read from disk, as described in Section IV-B. We choose the latter option because it allows for overlapping sequence exchanges with k-mer counting and matrix multiplication. This is also the approach adopted by PASTIS [20]. Once the sequences are communicated, we perform pairwise alignment for all identified pairs (i.e., the nonzeros $\mathbf{C}$) using a seed–and–extend algorithm that returns an alignment score and updated seed coordinates. If the score does not exceed the specified threshold, the read pair is discarded and the nonzero is removed from $\mathbf{C}$.

We can view these operations from a matrix point of view: (a) the pairwise alignment is an in-place element-wise operation on $\mathbf{C}$ that sets the alignment flag to true for each nonzero $\mathbf{C}_{ij}$ if the alignment score exceeds the given threshold, and to false otherwise, and (b) the removal of entries with false flags is another in-place operation on $\mathbf{C}$ that prunes nonzeros whose flags are set to false. The resulting matrix $\mathbf{R}$ (line 8 in Algorithm 1) at the end of these operations is the input for the transitive reduction algorithm. Contained overlaps, as defined in Section II, are discarded during transitive reduction regardless of their alignment scores. They may be reintroduced at later stages of the *de novo* assembly process.

E. Transitive Reduction

Our distributed memory transitive reduction algorithm takes the overlap matrix $\mathbf{R}$ as input and computes a transitively reduced version of $\mathbf{R}$, which we refer to as $\mathbf{S}$ (line 9 in Algorithm 1). Recall that each nonzero $\mathbf{R}_{ij}$ stores the number of common k-mers and their positions in the sequence pair (s_i, s_j). The transitive reduction algorithm needs two additional information for each such pair: the length of the overlap suffix and the overlap orientation. Both can be derived in-place from the alignment coordinates stored in $\mathbf{R}_{ij}$. The length of the suffix and the orientation are calculated and stored in $\mathbf{R}$

Algorithm 2 Parallel transitive reduction on $\mathbf{R}$.

```
1:  procedure TRANSITIVEREDUCTION(R)
2:      do
3:          prev ← R.NNZ
4:          N ← R^2                          ▷ Find edges two-hops away
5:          v ← R.REDUCE(Row, 0, max)
6:          v ← v.APPLY(x, add)              ▷ x is a scalar
7:          M ← R.DIMAPPLY(Row, v, return2nd)
8:          I ← M ≥ N                        ▷ Find transitive edges
9:          R ← R ∘ ¬I                       ▷ Remove transitive edges
10:         nnz ← R.NNZ
11:     while nnz ≠ prev
12:     S ← R
13:     return S
```

Algorithm 3 Custom MinPlus semiring used in $\mathbf{N} \leftarrow \mathbf{R}^2$.

```
1: struct MINPLUSSR
2:     ID( ) return ∞
3:     ADD(a, b) return MIN(a, b) ▷ Find the shortest path
4:     MULTIPLY(a, b)
5:         if ISDIROK( ) then return a + b
6:         else return ID( )
```

during the pairwise alignment, so that $\mathbf{R}$ is immediately ready for the transitive reduction phase.

Our transitive reduction algorithm is presented in Algorithm 2. The algorithm begins by discovering two-hop neighbors of each vertex in the overlap graph. This first step is the most computationally intensive stage of transitive reduction and is achieved by squaring the overlap matrix: $\mathbf{N} = \mathbf{R}^2$, where $\mathbf{N}$ is the two-hop *neighbor* matrix. In *de novo* assembly, whenever there are multiple alternative paths in the graph, we retain the one that gives us more genomic coverage in terms of nucleotides. Therefore, whenever there are multiple alternatives, the path with the shorter suffix is chosen, since a shorter suffix indicates a longer overlap between two sequences. This is achieved by using a custom MinPlus semiring during the squaring of $\mathbf{R}$. Algorithm 3 illustrates the MinPlus semiring we use, where we overload the addition operation with a minimum operation and the multiplication operation with a summation. Because of the bidirectionality of our graph, we make sure that the orientation of the edges in play conforms to the transitivity rules listed in Section II. This is ensured by checking whether the edges follow the transitivity rules during multiplication (line 5 in Algorithm 3). If not, we mark the edge as *not transitive*. In particular, we check in MinPlus semiring whether the two heads next to the intermediate node (i.e. the middle node of a three-node path) have opposite directions.

In lines 5–7 of Algorithm 2, we create the *maximal suffix* matrix $\mathbf{M}$, where each nonzero within a row is replaced by the maximum value (i.e., the longest overlap) of that row. In *de novo* assembly, read overlaps are approximate matches because sequencing errors can cause endpoint positions to shift. To make our algorithm robust to sequencing errors, we increase the value of the longest overlap per row (i.e. per read) by a scalar x. REDUCE in line 5 returns a vector $\mathbf{v}$ whose ith cell stores the maximum value of the ith row of $\mathbf{R}$. APPLY in line 6 adds x to each nonzero $\mathbf{v}_i$. For each nonzero $\mathbf{v}_i \neq 0$, DIMAPPLY in line 7 replaces the corresponding row i of $\mathbf{R}$ with $\mathbf{v}_i$ in its output. The last parameter in the functions REDUCE, APPLY and DIMAPPLY is the binary operator applied to each scalar operation.

The next step is to identify the transitive edges in $\mathbf{N}$ (line 8 in Algorithm 2). Our algorithm performs an element-wise operation between $\mathbf{M}$ and $\mathbf{N}$ to identify such edges. If $\mathbf{M}_{ij}$ is greater than or equal to $\mathbf{N}_{ij}$, the corresponding nonzero $\mathbf{I}_{ij}$ in the output matrix is set to true. In $\mathbf{N}$ we store the shortest path, so that all nonzeros with $\mathbf{M}_{ij} \geq \mathbf{N}_{ij}$ are transitive edges because $\mathbf{M}_{ij}$ is an overlap suffix longer than $\mathbf{N}_{ij}$.

The described element-wise operation is only performed for entries that are nonzero in both $\mathbf{M}$ and $\mathbf{N}$. Recall that when computing $\mathbf{N} = \mathbf{R}^2$ we checked whether a path is a valid walk or not. In this element-wise operation we also make sure that the orientation of the edges in the intersection of $\mathbf{N}$ and $\mathbf{M}$ follows the last two transitivity rules. In the element-wise operation, we check whether the two heads next to the *departure* node (i.e., the start node of a three-node path) and the two heads next to the *destination* node (i.e., the end node of a three-node path) have the same orientation.

The final operation of our transitive reduction algorithm is to prune the identified transitive edges from $\mathbf{R}$ (line 9 in Algorithm 2). This is achieved by an element-wise multiplication of $\mathbf{R}$ and the logical negation of $\mathbf{I}$, $\neg\mathbf{I}$. Any nonzero in $\mathbf{I}_{ij}$ becomes a zero in $\neg\mathbf{I}_{ij}$, therefore the nonzeros of $\mathbf{R}$ corresponding to the transitive edges (i.e. zeros) in $\neg\mathbf{I}$ are pruned. Again, only those entries that are nonzero in both $\mathbf{R}$ and $\neg\mathbf{I}$ are considered. This is equivalent to a set difference operator ($nonzeros(\mathbf{R}) \setminus nonzeros(\mathbf{I})$) in linear algebra.

In practice, we need several rounds to successfully remove all transitive edges, since we need to consider neighbors that are three, four, etc. hops away. Therefore, our algorithm iterates on $\mathbf{R}$ until the number of nonzeros remains the same (line 11 in Algorithm 2).

V. COMMUNICATION ANALYSIS

In this section we analyze the communication costs of diBELLA 2D and compare them with our 1D implementation. First, we briefly consider the communication cost of the k-mer counting phase, which is common to both implementations. Then we examine the communication costs of the overlapping phase and read exchange, which are the main differences between the two implementations. The communication costs of the transitive reduction for the current implementation follow. The communication costs are given in word count W (*bandwidth cost*) and number of messages Y (*latency cost*). The communication costs are summarized in Table I next to useful notations in Table II.

A. Communication Cost of K-mer Counting

The communication costs for this step depend on the properties of the input dataset and the settings of our algorithm,

TABLE I
COMMUNICATION COSTS OF DIBELLA 1D AND DIBELLA 2D.

Task	Bandwidth		Latency	
	diBELLA 1D	diBELLA 2D	diBELLA 1D	diBELLA 2D
K-mer Counting	$nlk/4P$	$nlk/4P$	bP	bP
Overlap Detection	a^2m/P	$am/\sqrt{P}$	P	$\sqrt{P}$
Read Exchange	cnl/P	$2nl/\sqrt{P}$	$\min\{cnl/P, P\}$	$\sqrt{P}$
Transitive Reduction	-	$rn/\sqrt{P}$	-	$t\sqrt{P}$

such as the depth d of the dataset, the genome size G (in nucleotides), the k-mer length k and the read length l.

Our total input size is $Gd \approx nl$. Each processor has $(1/P)$th of the input. Each sequence has $(l-k+1)$ k-mers and each k-mer takes $k/4$ bytes using 2–bit compression per nucleotide. Hence, the total size on each processor before communication is $n(l-k+1)k/4P$. For long–read data $l-k+1 \approx l$, since l is usually 2–3 orders of magnitude larger than k.

The hash function maps k-mers uniformly and randomly on processors, so that each processor keeps $(1/P)$th of the data for itself and communicates the rest. For large P we can assume $(P-1)/P \approx 1$ to avoid clutter. Hence, the bandwidth cost for k-mer counting per process is on average:

$$W = \frac{P-1}{P}\frac{n(l-k+1)k}{4P} \approx \frac{nlk}{4P} \tag{1}$$

Based on the available memory, we may have to perform several k-mer exchanges. So the latency cost of k-mer counting is $Y = bP$, where b is the batch count.

B. Communication Cost of Overlap Detection

In our application $\mathbf{AA}^\mathsf{T}$ is the output matrix $n \times n$, $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ are the input matrices of dimension $n \times m$ and $m \times n$ respectively. The number of nonzeros in both $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ is am, where a is the density indicating the average number of sequences containing a particular k-mer. The k-mer selection procedure that we present in [8] and that we also use in this work chooses an interval for the k-mer frequency, which in turn translates into the average number of sequences that can contain a given k-mer.

Our previous work, diBELLA 1D, computes overlap detection using distributed hash tables [9]. K-mers are distributed to processors that allow them to detect candidate overlap pairs locally. This must be followed by a global reduction. In terms of communication, this implementation is equivalent to a 1D sparse matrix multiplication using the outer product algorithm [31], [32]. The 1D outer product formulation distributes $\mathbf{A}$ in block columns where the ith block column is denoted by $\mathbf{A}_{:i}$, and $\mathbf{A}^\mathsf{T}$ in block rows where the ith block row is denoted by $\mathbf{A}^\mathsf{T}_{i:}$. $\mathbf{C}$ is distributed in block rows in diBELLA 1D. The computation can be written as $C = \sum_{i=1}^{P} \mathbf{A}_{:i}\mathbf{A}^\mathsf{T}_{i:}$.

Each k-mer exists on average in a sequences, hence the local overlap detection $\mathbf{A}_{:i}\mathbf{A}^\mathsf{T}_{i:}$ generates a^2m/P nonzeros on each processor. These nonzeros must be reduced before performing pairwise alignment, so that no read-read pair is aligned more than a few (1-2) times, depending on the algorithm parameters. This means that each processor exchanges $W_{1D} = a^2m/P$ words and the latency cost is $Y_{1D} = P$.

TABLE II
LIST OF SYMBOLS AND ANNOTATIONS IN OUR PAPER.

Symbol	Description
n	Read set cardinality
m	K-mer set cardinality
d	Depth of coverage
k	K-mer length
L	Overlap length
l	Read length
$\mathbf{A}$	Data matrix: reads–by–kmers
$\mathbf{C}$	Candidate overlap matrix: reads–by–reads
$\mathbf{R}$	Overlap matrix: reads–by–reads
$\mathbf{S}$	String matrix: reads–by–reads
a	$\mathbf{A}$ average density: $\mathrm{nnz}(\mathbf{A})/m$
c	$\mathbf{C}$ average density: $\mathrm{nnz}(\mathbf{C})/n$
r	$\mathbf{R}$ average density: $\mathrm{nnz}(\mathbf{R})/n$
s	$\mathbf{S}$ average density: $\mathrm{nnz}(\mathbf{S})/n$
P	Total number of processes
W	Bandwidth cost
Y	Latency cost

In contrast, diBELLA 2D uses a 2D sparse matrix multiplication algorithm known as Sparse SUMMA [30]. The P processors are logically organized in a $\sqrt{P} \times \sqrt{P}$ grid with row and column indexes, so that the (i,j)th processor is P_{ij}. Each processor stores a $n/\sqrt{P} \times m/\sqrt{P}$ submatrix $\mathbf{A}_{ij}$ and a $m/\sqrt{P} \times n/\sqrt{P}$ submatrix $\mathbf{A}^\mathsf{T}_{ij}$ in its local memory. Each processor calculates a product of a block row of $\mathbf{A}$ with a block column of $\mathbf{A}^\mathsf{T}$. Sparse SUMMA is an owner-computes algorithm, so we only need to consider the communication of input matrices. If we assume a good load balance, which we achieve by randomly permuting k-mers and reads, $\mathbf{A}_{ij}$ has am/P nonzeros. Each processor P_{ij} receives $2(\sqrt{P}-1)$ input blocks because $\mathbf{C}_{ij} = \sum_{i=k}^{\sqrt{P}} \mathbf{A}_{ik}\mathbf{A}^\mathsf{T}_{kj}$.

For large P we simplify $\sqrt{P}-1 \approx \sqrt{P}$ so that the number of nonzeros that a processor must collect is $W_{2D} = am/\sqrt{P}$ and the latency cost is $Y_{2D} = \sqrt{P}$.

C. Communication Cost of Read Exchange

The communication costs of the read exchange are derived from the analysis in the previous section. The sequences are distributed to the processors by parallel I/O according to the corresponding implementation decomposition, i.e. 1D for our first implementation and 2D for the present work.

To compute the communication costs, we consider the candidate overlap matrix $\mathbf{C}^{n\times n} = \mathbf{AA}^\mathsf{T}$. $\mathbf{C}$ has cn nonzeros (before computing pairwise alignment) where c is its density per row or column, which indicates the average number of overlapping sequences for each read. Each exchange costs $O(l)$. The 1D algorithm exchanges at most $W_{1D} = cnl/P$ words and sends $Y_{1D} = \min\{cnl/P, P\}$ messages, while the current 2D implementation exchanges at most $W_{2D} = 2nl/\sqrt{P}$ words and sends $Y_{2D} = \sqrt{P}$ messages.

diBELLA 1D communicates at most one read per nonzero because an alignment task is only assigned to a processor if this processor has at least one of the two sequences involved. diBELLA 2D communicates the full range of sequences that a processor may need and starts the read exchange immediately

TABLE III
LIST OF EXPERIMENTAL VALUES OF SPARSITY FOR DIBELLA 2D.

Dataset	Depth (d)	C density (c)	Inefficiency (%/2d)	R density (r)
E. coli	30	145.9	2.4	6.4
C. elegans	40	1,579.7	19.7	8.1
H. sapiens	10	1,207.7	60.4	1.3

TABLE IV
DATA SETS USED DURING EVALUATION: NAME, DEPTH, NUMBER OF SEQUENCES IN THE INPUT, AVERAGE READ LENGTH, INPUT SIZE, GENOME SIZE, AND ERROR RATE.

Label	Depth	Reads (K)	Length	Input (GB)	Size (Mb)	Error
C. elegans	40	420.7	11,241	4.8	100	0.13
H. sapiens	10	4,421.6	7,401	33.1	3,000	0.15

after the initial data partition, so that communication overlaps with computation.

The 1D algorithm has better scaling with increased concurrency, but has a large constant c, whose typical value often exceeds 1000 for large genomes, as shown in Table III. To overcome this large constant and communicate fewer words than the 2D algorithm, the 1D algorithm would require $(c^2/4)$–way parallelism. Ellis et al. [9] show that $c \approx 2d$ for a perfect overlapper. In practice, c is much larger than d and $c/2d$ can be considered the *inefficiency factor* of an overlapper.

D. Communication Cost of Transitive Reduction

The sparse matrix multiplication dominates the runtime of the transitive reduction algorithm. The communication costs for the squaring of $\mathbf{R}$ follow from the previous analysis and are $W_{2D} = rn/\sqrt{P}$, where $r \leq c$ is the sparsity of the overlap matrix $\mathbf{R}^{n \times n}$ after performing the pairwise alignment, which often leads to the discarding of nonzeros, and $Y_{2D} = \sqrt{P}$. The transitive reduction algorithm also contains some element-wise sparse routines, but these are executed in-place so that they do not contribute to communication time. While the transitive reduction loop is repeated until convergence, the number of iterations is often a small constant (denoted t in Table I) and the geometrically decreasing density after each iteration makes the total communication volume asymptotically equal to that of the first iteration.

VI. EXPERIMENTAL SETUP

Our experiments were performed on two machines: the Cray XC40 supercomputer Cori at NERSC and the IBM supercomputer Summit at Oak Ridge National Laboratory. On Cori we use the Haswell partition, while on Summit we use only IBM POWER9 CPUs. Using two architectures shows that our algorithm scales on different architectures. However, this is not intended to be a cross–platform comparison, as our algorithm is not specifically optimized for either platform.

Each Haswell node on the Cori system consists of two 2.3 GHz 16-core Intel Xeon processors and has a total memory of 128 GB. Each Summit node has two 22-core IBM POWER9 processors and 512 GB DDR4 of RAM. Because one core per Summit half-node is reserved for OS, each node has a maximum of 42 cores available for application codes. In this paper, we do not utilize the GPUs available on Summit. For details on the two architectures, see Table V.

To investigate the parallel performance of our algorithm, we use two data sets from Pacific Biosciences (CLR technology) [33] with different sizes: Caenorhabditis elegans (C. elegans) and Homo sapiens (H. sapiens). Details of the two data sets are given in Table IV. Our algorithm is also suitable for other long–read technologies such as Pacific Biosciences CCS [34] (or HiFi) and Oxford Nanopore [35]. In this paper we only run with CLR data as our parameters are tuned to it and the accuracy of our tool for CLR input is reported in the single node BELLA paper [8].

The experiments are divided into two groups: (a) parallel performance and scalability of diBELLA 2D and (b) performance comparison with related work. In the latter case, we compare the overlap detection of diBELLA 2D with diBELLA 1D [9] and minimap2 [16] written in C for shared memory, while we compare our transitive reduction algorithm with SORA [23] written in Scala on top of Apache Spark [24].

diBELLA 2D and 1D run with the same input and alignment setting, i.e. $k = 17$ and maximum k-mer frequency equal to 4, while minimap2 run with its default setting for CLR data. The results from minimap2 and SORA were only collected from Cori, as minimap2 uses SSE intrinsics that are not supported on the IBM POWER9 processor, and Summit has no support for Apache Spark. For minimap2 we only report single node performance because it does not implement distributed memory parallelism. To compare transitive reduction, we used the output of diBELLA 2D as input for SORA, which is an overlap graph consisting of 5.8M edges and 4.4M vertices for the H. sapiens data set and 4.2M edges and 0.4M vertices for C. elegans. We only compare the execution time of the transitive reduction by removing all start and shutdown times from Apache Spark and the time dedicated to I/O.

On Cori, we used `gcc-8.3.0` and the `O3` flag to compile C/C++ codes, while on Summit we used `gcc-8.1.1`. On both Cori and Summit, we used the default MPI implementation. SORA used `jdk/1.8.0_202` and `spark/2.3.0`. In the next section we report the average runtime over 10 runs for each experiment, except for the H. sapiens data set at low concurrency, where we report the average over three runs.

VII. EXPERIMENTAL RESULTS

In this section we first examine the parallel performance of diBELLA 2D and its individual components and then compare our performance with the state of the art.

A. Detailed Analysis of diBELLA 2D

Figure 4 illustrates the strong scaling of our algorithm for C. elegans on the left and for H. sapiens on the right. In this figure, the two machines run on $P = \{32, 72, 128\}$ nodes using 32 MPI ranks/node for C. elegans and $P = \{128, 200, 288, 338\}$ for H. sapiens. For C. elegans, diBELLA 2D achieves a parallel efficiency of 83% on Summit CPU, while it achieves a parallel efficiency of 68% on Cori Haswell. For H. sapiens, the parallel efficiency of both machines is over 80%

TABLE V
DETAILS OF THE MACHINES USED FOR EVALUATION: NAME, NUMBER OF PHYSICAL CORES PER NODE, PROCESSOR MAX TURBO FREQUENCY, PROCESSOR MODEL, MEMORY, NETWORK, AND L1, L2, L3 CACHES SIZES.

Platform	Cores/Node	Frequency (GHz)	Processor	Memory (GB)	Network and Topology	L1	L2	L3
Cori Haswell	32	3.6	Intel Xeon E5-2698V3	128	Aries Dragonfly	64KB	256KB	40MB
Summit CPU	42	4.0	IBM POWER9	512	InfiniBand Non-Blocking Fat Tree	32KB	512KB	10MB

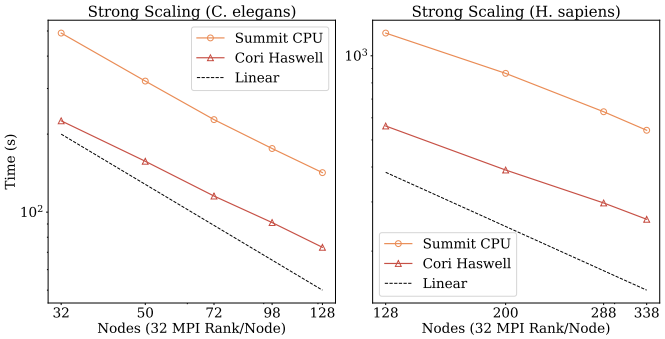

Fig. 4. diBELLA 2D strong scaling on Cori Haswell and Summit CPU using 32 MPI rank/node on C. elegans (left) and on H. sapiens (right).

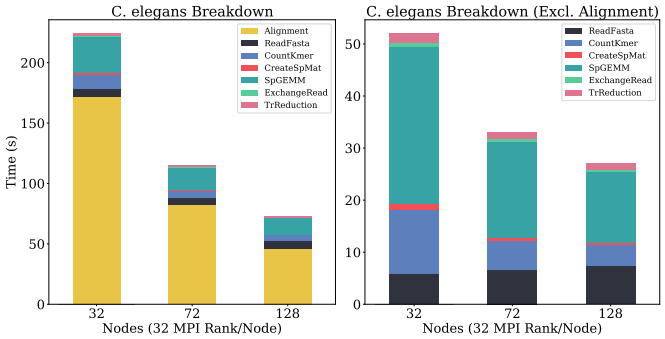

Fig. 5. diBELLA 2D runtime breakdown on Cori Haswell (C. elegans).

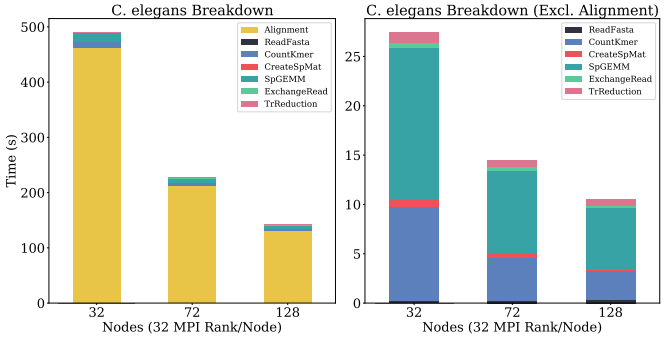

Fig. 6. diBELLA 2D runtime breakdown on Summit CPU (C. elegans).

with a peak of 92% on Summit CPU. These results show the near linear scaling behavior of our overall algorithm using a large input on two different architectures.

Figures 5–6 (C. elegans) and Figures 7–8 (H. sapiens) show the runtime breakdown of diBELLA 2D on the two machines. In each breakdown, the plot on the left shows total execution time including pairwise alignment, while the plot on the right excludes pairwise alignment. We included plots excluding pairwise alignment because alignment takes a large proportion of the runtime and makes it difficult to see the scaling of the other stages. From bottom to top the layers are ordered according to the legend. The first layer is the pairwise alignment, i.e. the time needed to align all non-zero pairs in the candidate overlap matrix $\mathbf{C}$. `ReadFastq` is the time spent reading and parsing the input file in parallel. Immediately after this step, we start exchanging sequences to overlap this communication with the subsequent computation. `CountKmer` corresponds to the k-mer counting stage, and `CreateSpMat` corresponds to the time needed to create the input matrices $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$. `SpGEMM` includes both the communication time and the computation time to create the candidate overlap matrix $\mathbf{C} = \mathbf{A}\mathbf{A}^\mathsf{T}$. `ExchangeRead` times the period from the end of `SpGEMM` until all sequence exchanges are complete. Depending on the MPI implementation, the read exchange may potentially overlap with k-mer counting and overlap detection phases. Finally, `TrReduction` is the transitive reduction time.

Figures 5–6 show diBELLA 2D performance breakdown on the C. elegans dataset using P={32, 72, 128} nodes of each machine. diBELLA 2D runs faster overall on Cori Haswell than on Summit CPU. The relative proportion of pairwise alignment in the total runtime increases on Summit CPU compared to Cori Haswell. SeqAn's pairwise alignment is probably not optimized for IBM processors, so we refrain from making architecture comparisons based solely on this data.

On the two machines, the sparse matrix computation $\mathbf{A}\mathbf{A}^\mathsf{T}$ for overlap detection after pairwise alignment has the greatest contribution to runtime. The parallel read I/O does not show any scaling and its performance deteriorates when the number of processes is increased. For C. elegans, overlap detection has a parallel efficiency of 55% for Cori Haswell and 63% for Summit CPU, while k-mer counting (consisting of first and second passes) has a parallel efficiency of 70–80% across machines. The read exchange has a parallel efficiency of 50% across machines. Despite a parallel efficiency of 38% on Cori Haswell and Summit CPU, our transitive reduction shows a significant speedup compared to a competing distributed memory implementation.

Figures 7–8 tell a similar story for H. sapiens. The size of the data set, which is about 10× of C. elegans, mitigates the contribution of I/O to the overall runtime. The parallel efficiency of $\mathbf{A}\mathbf{A}^\mathsf{T}$ increases to 65% on both machines. The parallel efficiency of k-mer counting reaches 89% on Cori Haswell. The formation of $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ has negligible impact on the total runtime, yet it scales almost linearly with a parallel efficiency of over 80% on both machines and data sets.

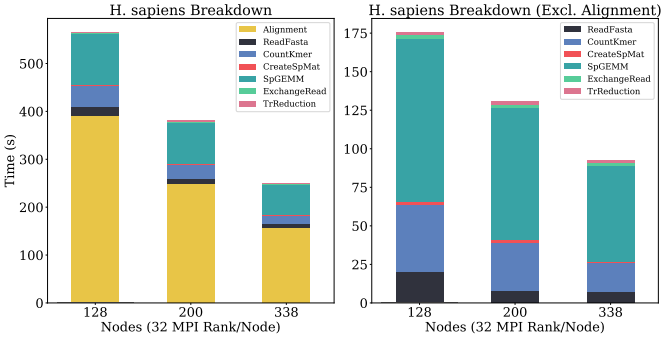

Fig. 7. diBELLA 2D runtime breakdown on Cori Haswell (H. sapiens).

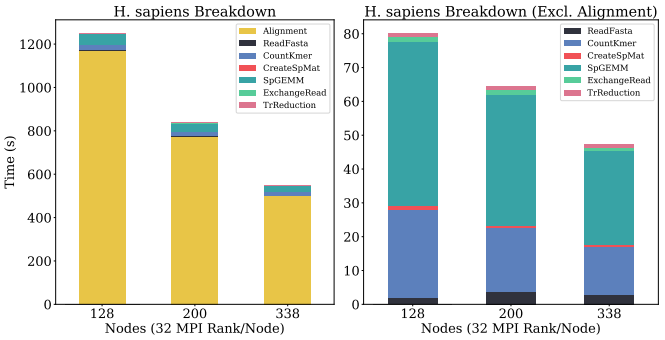

Fig. 8. diBELLA 2D runtime breakdown on Summit CPU (H. sapiens).

B. Comparison with the State of the Art

To demonstrate the competitiveness of our approach, we now compare it with prior work in the literature.

First, we compare the overall runtime and scaling of diBELLA 2D with diBELLA 1D [9], subtracting the transitive reduction time from diBELLA 2D, since the 1D version does not implement this step. This comparison was made on Summit CPU and is shown in Figure 9 for C. elegans and H. sapiens. diBELLA 2D and 1D differ mainly in the way they perform overlap detection and communicate sequences before pairwise alignment. They exhibit similar near linear scaling behavior, but diBELLA 2D consistently outperforms the 1D implementation by 1.5–1.9× (average 1.7×) for the C. elegans data set and 1.2–1.3× (average 1.2×) for H. sapiens.

For completeness, we evaluate diBELLA 2D against minimap2 [16], a popular shared memory algorithm for overlap detection. To do this, we run minimap2 on a single node using 32 OpenMP threads and compare its runtime to diBELLA 2D on a different number of nodes using 32 MPI ranks/node. It is important to note that minimap2 and diBELLA 2D perform significantly different computations. In particular, minimap2 does not perform base-level pairwise alignment and instead estimates pairwise similarity from the number of shared minimizers, making it significantly faster. Nevertheless, they are ultimately aimed at solving the same problem, which is why we provide a comparison here. For the runtimes of diBELLA 2D we refer to Figure 4. For C. elegans, minimap2 is about 2× faster than diBELLA 2D at $P = 8$, while at higher concurrency diBELLA 2D is 1.6×, 3.2×, and 5× faster than minimap2.

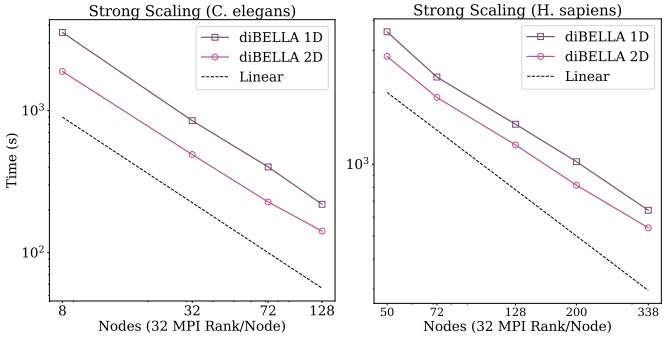

Fig. 9. Comparison of diBELLA 2D and diBELLA 1D [9] on Summit CPU.

TABLE VI
COMPARISON OF TRANSITIVE REDUCTION (IN SECONDS) BETWEEN DIBELLA 2D AND SORA [23] ON CORI HASWELL.

Dataset	Nodes	SORA	diBELLA 2D	Speed-Up
C. elegans	32	34.6	1.9	18.2×
	72	34.3	1.3	26.4×
	128	34.9	1.2	29.0×
H. sapiens	128	23.4	1.9	12.4 ×
	200	24.3	2.3	10.5 ×
	338	25.3	1.9	13.3 ×

The speedup of diBELLA 2D over minimap2 is 9.5×, 13.7×, and 20.6× at P={128, 200, 338} for the H. sapiens dataset.

Finally, we compare our transitive reduction algorithm with the transitive reduction module of SORA [23], a distributed memory implementation of transitive reduction from overlap graph to string graph based on Apache Spark and GraphX. Our transitive reduction algorithm is currently integrated into our pipeline, so for fairness reasons we do not include startup, shutdown, and I/O time for SORA. The input of SORA is the overlap matrix $\mathbf{R}$ of diBELLA 2D, therefore the two transitive reduction algorithms work with the same input. The results are summarized in Table VI. Our transitive reduction algorithm has a speedup of up to 29× for C. elegans and up to 13.3× for H. sapiens.

VIII. CONCLUSIONS

In this work, we presented a scalable distributed memory approach called diBELLA 2D for overlap detection and layout simplification in the context of *de novo* genome assembly. diBELLA 2D relies on linear algebraic operations over semirings using 2D distributed sparse matrices. Using sparse matrices for both overlap detection and transitive reduction reduces the need for different data structures normally used in genome assembly. Well-studied distributed-memory algorithms on sparse matrices, such as sparse matrix-matrix multiplication, allows diBELLA to efficiently parallelize the computation without losing expressiveness, thanks to the semiring abstraction. We provided detailed communication analysis for parallel overlap detection and transitive reduction, which were missing from previous work. In particular, our analysis shows that the new 2D overlap detection algorithm reduces communication compared to the existing 1D algorithm for commonly utilized concurrencies in the range of 100–10000 processors. This

translates into a speedup of 1.2–1.9× in our experiments. For transitive reduction, our approach is 10–29× faster than an existing distributed-memory implementation.

Our approach paves the way for high performance assembly of large genomes where it is impractical to assemble them in shared memory or on a small cluster. Fast and efficient *de novo* assembly enables the study of previously uncharacterized genomes [7]. Deciphering the nucleotide sequence is also of crucial importance when a reference genome is available. In mapping sequences to a reference, individual–specific genetic variations are often lost. But they are crucial, for example, to discover disease–causing mutations [36].

Future work includes reducing memory consumption so that diBELLA 2D can assemble large genomes at low concurrency if desired. For this purpose, we can form only a part of the candidate overlap matrix in each time step, aligning only sequences belonging to this part, and removing the spurious entries before moving on to the next region of the output matrix. Also, we plan to use GPUs in both pairwise alignment and k-mer counting to boost the performance of our pipeline.

ACKNOWLEDGMENTS

This work is supported by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Thanks to Alok Tripathy for useful suggestions and valuable discussions.

REFERENCES

[1] W. Zhang, J. Chen, Y. Yang, Y. Tang, J. Shang, and B. Shen, "A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies," *PloS One*, vol. 6, no. 3, p. e17915, 2011.

[2] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman *et al.*, "Real-time DNA sequencing from single polymerase molecules," *Science*, vol. 323, no. 5910, pp. 133–138, 2009.

[3] S. Goodwin, J. Gurtowski, S. Ethe-Sayers, P. Deshpande, M. C. Schatz, and W. R. McCombie, "Oxford nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome," *Genome Research*, vol. 25, no. 11, pp. 1750–1756, 2015.

[4] A. M. Phillippy, M. C. Schatz, and M. Pop, "Genome assembly forensics: finding the elusive mis-assembly," *Genome Biology*, vol. 9, no. 3, p. R55, 2008.

[5] N. Nagarajan and M. Pop, "Parametric complexity of sequence assembly: theory and applications to next generation sequencing," *Journal of Computational Biology*, vol. 16, no. 7, pp. 897–908, 2009.

[6] 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.

[7] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome Research*, vol. 22, no. 3, pp. 549–556, 2012.

[8] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper," *bioRxiv*, p. 464420, 2020.

[9] M. Ellis, G. Guidi, A. Buluç, L. Oliker, and K. Yelick, "diBELLA: Distributed Long Read to Long Read Alignment," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[10] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[11] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[12] 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.

[13] J. Edmonds and E. L. Johnson, "Matching: A well-solved class of integer linear programs," in *Combinatorial Optimization—Eureka, You Shrink!* Springer, 2003, pp. 27–30.

[14] B. G. Jackson and S. Aluru, "Parallel construction of bidirected string graphs for genome assembly," in *37th International Conference on Parallel Processing*. IEEE, 2008, pp. 346–353.

[15] P. Bonizzoni, G. D. Vedova, Y. Pirola, M. Previtali, and R. Rizzi, "Fsg: fast string graph construction for de novo assembly," *Journal of computational biology*, vol. 24, no. 10, pp. 953–968, 2017.

[16] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[17] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM (JACM)*, vol. 52, no. 4, pp. 552–581, 2005.

[18] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep., 1994.

[19] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, and R. Rizzi, "An external-memory algorithm for string graph construction," *Algorithmica*, vol. 78, no. 2, pp. 394–424, 2017.

[20] O. Selvitopi, S. Ekanayake, G. Guidi, G. Pavlopoulos, A. Azad, and A. Buluç, "Distributed Many-to-Many Protein Sequence Alignment Using Sparse Matrices," *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[21] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "Communication-efficient jaccard similarity for high-performance distributed genome comparisons," in *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 1122–1132.

[22] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno, "Computability of models for sequence assembly," in *International Workshop on Algorithms in Bioinformatics*. Springer, 2007, pp. 289–301.

[23] A. J. Paul, D. Lawrence, M. Song, S.-H. Lim, C. Pan, and T.-H. Ahn, "SORA: Scalable overlap-graph reduction algorithms for genome assembly using apache spark in the cloud," in *International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2018, pp. 718–723.

[24] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[25] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 599–613.

[26] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "HipMer: an extreme-scale de novo genome assembler," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2015, p. 14.

[27] A. Döring, D. Weese, T. Rausch, and K. Reinert, "SeqAn an efficient, generic C++ library for sequence analysis," *BMC Bioinformatics*, vol. 9, no. 1, p. 11, 2008.

[28] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter," *BMC Bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[29] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[30] A. Buluc and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *37th International Conference on Parallel Processing*. IEEE, 2008, pp. 503–510.

[31] ——, "On the representation and multiplication of hypersparse matrices," in *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008.
[32] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013, pp. 222–231.
[33] A. Rhoads and K. F. Au, "PacBio sequencing and its applications," *Genomics, Proteomics & Bioinformatics*, vol. 13, no. 5, pp. 278–289, 2015.
[34] A. M. Wenger, P. Peluso, W. J. Rowell, P.-C. Chang, R. J. Hall, G. T. Concepcion, J. Ebler, A. Fungtammasan, A. Kolesnikov, N. D. Olson *et al.*, "Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome," *Nature Biotechnology*, vol. 37, no. 10, pp. 1155–1162, 2019.
[35] M. Jain, H. E. Olsen, B. Paten, and M. Akeson, "The oxford nanopore minion: delivery of nanopore sequencing to the genomics community," *Genome Biology*, vol. 17, no. 1, p. 239, 2016.
[36] M. J. Chaisson, R. K. Wilson, and E. E. Eichler, "Genetic variation and the de novo assembly of human genomes," *Nature Reviews Genetics*, vol. 16, no. 11, pp. 627–640, 2015.