A Work-Optimal Parallel Algorithm for Aligning Sequences to Genome Graphs

Aranya Banerjee*, Daniel Gibney†, Helen Xu*, and Srinivas Aluru*
*School of Computational Science and Engineering, Georgia Institute of Technology
aranyabanerjee@gatech.edu, aluru@cc.gatech.edu, hxu15@gatech.edu
†Department of Computer Science, University of Texas at Dallas
daniel.gibney@utdallas.edu

***Abstract*—Representing genetic variations among a population of individuals in the form of a genome graph, and using the graph as a reference instead of the genome of a single individual, are central techniques in the fast-emerging area of pangenomics. A fundamental problem in pangenomics is to align a DNA sequence simultaneously against all reference genomes through sequence-to-graph alignment. The sequential approach to the problem uses dynamic programming and takes $O(m|E|)$ time, where m is the length of the DNA sequence that is aligned and $|E|$ is the number of edges in the genome graph.**

In this paper, we present ParSGA, the first parallel algorithm for the sequence-to-graph alignment problem. Prior works on parallelization only addressed the embarrassingly parallel problem of mapping numerous DNA sequences independently, but each sequentially, to the genome graph. In contrast, ParSGA aligns a single sequence in parallel, which is required in cases involving long or ultra-long DNA sequences, or in applications involving successively mapping and incorporating DNA sequences into an evolving graph, or to improve available parallelism for multiple sequence-to-graph alignments even further. ParSGA is work-optimal, and its design provides a high degree of parallelism. On a 128-core AMD Epyc processor, ParSGA achieves $81\times$ speedup compared to serial execution of itself, and $43\times$ speedup compared to the sequential algorithm, when aligning 250bp reads to human Chromosome 1 variation graph. In the popularly used billion cell updates per second (GCUPS) metric, ParSGA achieves 6.14 GCUPS. The C++ implementation of ParSGA is available at https://github.com/ParBLiSS/ParSGA.

***Index Terms*—Alignment, genome graph, variation graph, pangenomics, dynamic programming.**

I. Introduction

Advances in rapid and inexpensive DNA sequencing have led to the sequencing of genomes of numerous individuals in a population, particularly for humans. Due to the low frequency of variations compared to the size of the genome, genetic variations among individuals are succinctly captured in the form of a labeled graph, often referred to as the genome graph or pangenome [1]–[6]. The deciphering of human pangenomes is the subject of several major international projects [7], [8].

The ***sequence-to-graph alignment***[1] problem is a fundamental problem in utilizing genome graphs: it compares a DNA fragment from a newly sequenced individual against the graph to uncover known and potentially unknown genetic variations. By comparing query sequences with a genome graph, sequence-to-graph alignment avoids a painstaking comparison of the sequence against each of the individual reference genomes upon which the genome graph is built. Sequence-to-graph alignment also extends to other scenarios involving labeled graphs beyond variation graphs. For example, such graphs can be built for any set of related genetic data, and are used in applications involving microbial metagenomes, sequencing reads instead of genomes, and genome assembly [10], [11].

The objective of sequence-to-graph alignment is to find a walk in a genome graph G such that its corresponding sequence minimizes the edit distance from an input query sequence P. More formally, the sequence-to-graph alignment problem takes as input a genome graph $G = (V, E)$ with $|V|$ nodes and $|E|$ edges and a query sequence P of length m.

With edits only allowed in the query sequence, the problem can be solved in $O(m|E|)$ time using a dynamic-programming algorithm [12], which is optimal under the Strong Exponential Time Hypothesis [13]. If edits are allowed in the graph, either standalone or in conjunction with edits in the query, the problem is NP-complete [14].

Sequence-to-graph alignment has been implemented in several software tools designed to work with genome graphs [15]–[18]. While some tools are direct implementations of the optimal dynamic programming algorithm, others utilize heuristics such as seed-and-extend techniques to obtain good computational efficiency in practice. Nevertheless, the worst-case complexity for optimal sequence-to-graph alignment remains unaltered [13].

Prior works on accelerating sequence-to-graph alignment primarily focus on using parallel resources to simultaneously align multiple query sequences to a genome graph. Each query sequence is aligned independently using the serial algorithm, but multiple alignment tasks are scheduled together using SIMD instructions or multithreading [15], [16], [19]–[24].

The query sequences, or ***reads***, to be aligned are typically the output of next-generation sequencers. While short reads of length 150bp-250bp produced by Illumina sequencers are often aligned in bulk, the parallelization of a single instance of sequence-to-graph alignment is required for long (PacBio, 15-20 Kbp) and ultra-long (Oxford Nanopore, >1 Mbp) sequences. Particularly for Nanopore sequencers, there

[1]Also referred to as Pattern Matching on Labeled Graphs (PMLG) [9]

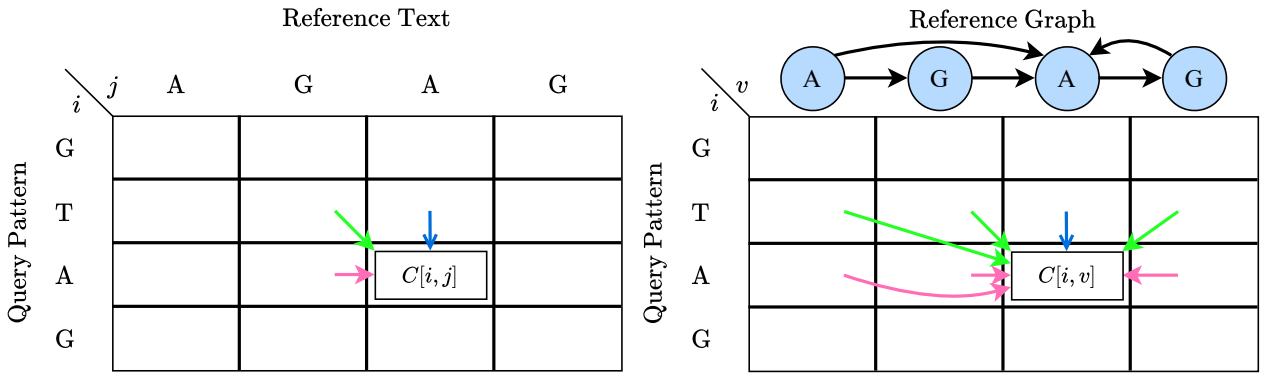

Fig. 1. Linear versus graph reference while computing a cell in the dynamic-programming table. Blue edges represent deletion operations, green edges represent substitution operations, and pink edges represent insertion operations.

is value in quickly identifying the mapping region of the query sequence after only sequencing a portion of it, to abort the effort if the mapping region offers little value. There are also applications involving successively aligning and then incorporating DNA sequences into an evolving graph, where the only means to parallelization is at the single query level. Note that a parallel algorithm for sequence-to-graph alignment can be combined with parallel algorithms for handling multiple reads to further enhance the scaling even for applications involving bulk sequence-to-graph alignments.

To date, there is no existing sequence-to-graph alignment algorithm that uses multiple parallel cores, and only one tool accelerates a single sequence-to-graph alignment with bit-level parallelism: GraphAligner [20], [25] uses bit vector operations to process multiple DNA bases, since only two bits are sufficient to represent a base. Given a machine word size w, GraphAligner achieves $O(|V|+\frac{m}{w}|E|\log w)$ time on directed acyclic graphs (DAGs) and $O(|V|+m|E|\log w)$ time on arbitrary graphs. However, it does not make use of the many cores available in today's shared-memory multicore machines as it is limited to bit-level parallelism. Note that the run-time complexity improves by a factor of $O\left(\frac{w}{\log w}\right)$ for DAGs but actually degrades by a factor of $O(\log w)$ for arbitrary graphs. Nevertheless, empirical studies show practical performance gains of $3\times$ to $20\times$ depending on the genome graph.

Parallelizing sequence-to-graph alignment with multiple cores. This paper introduces ParSGA, the first parallel algorithm that aligns a query sequence to a genome graph. ParSGA directly parallelizes the sequential dynamic programming (DP) algorithm for sequence-to-graph alignment that we will detail in Section II. At a high level, the DP algorithm constructs an $m \times |V|$ DP table of cells, where each row has $|V|$ cells, each corresponding to a node in the genome graph. The ParSGA algorithm proceeds row-by-row (i.e., character-by-character in the query sequence) sequentially, but parallelizes within a row across the graph.

The primary challenge in updating cells in parallel are

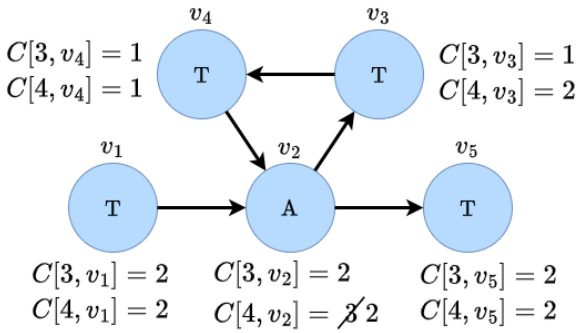

Fig. 2. An example from Navarro [26] demonstrating how insertion propagation modifies the C values (as defined in Equation 1) for the pattern 'TTTT'. The value $C[4, v_2] = 2$ shows the insertion update that occurs when considering vertex v_4 again (after updating the loop).

dependencies between neighboring nodes in the graph and the corresponding cells in both the current and previous rows, as shown in Figure 1. Insertions, which are the appending of characters to the end of the pattern, are particularly challenging, as their effect can propagate along sequences of edges connecting cells within the same row. Figure 2 reproduces an example by Navarro [26] that demonstrates these difficulties. Within the cycle pictured, for example, any vertex may possibly improve any other vertex, so that there is no clear order of updates on scores over which parallelization should occur.

The main insight behind ParSGA is that edges corresponding to the next character in the query effectively terminate insertion propagation, and ParSGA uses this observation to break the genome graph into numerous small and independent components over which the sequential algorithm can be readily parallelized. The other types of character operations (match/mismatch, the alignment of equal or unequal characters on the pattern and reference, and deletion, the removal of the last pattern character) do not depend on the graph edges within a row, and can be easily parallelized as there are no

dependencies between the relevant cells in the previous row.

Results summary. On 128 cores, ParSGA achieves up to $43\times$ speedup compared to the sequential algorithm on the CHR1[2] graph, which has 2.63×10^8 edges. Furthermore, when running on all 128 cores on the CHR1 graph, ParSGA achieves 6.14 billion cell updates per second (GCUPS), which is computed by dividing the number of DP table cells ($m|V|$) by the time in seconds. Section IV contains details about the full evaluation, which includes scalability analysis and experiments on genome graphs from multiple human chromosomes and other organisms.

Contributions. The main contributions of this paper are as follows:

- The construction and analysis of ParSGA, a work-optimal shared-memory parallel algorithm for sequence-to-graph alignment. That is, ParSGA performs asymtotically the same amount of operations as the optimal serial dynamic-programming algorithm ($O(m|E|)$).
- An implementation of ParSGA in C++ and OpenMP [27].
- An empirical evaluation of ParSGA on large real-world variation graphs that demonstrates that it achieves good efficiency and parallel speedup in practice.

II. Preliminaries

This section presents notation necessary to describe the sequence-to-graph alignment problem and its solutions. It then formalizes the problem and reviews the classical dynamic-programming solution. Finally, it presents the classical Compressed Sparse Row (CSR) graph representation and the work-span analysis model used to prove bounds in later sections.

A. Notation

We model genome graphs as vertex-labeled directed graphs, although the same techniques can be applied to edge-labeled graphs and/or undirected graphs with slight modifications. Let $G = (V, E, \sigma : V \to \Sigma)$ denote a vertex-labeled graph over the alphabet Σ, where $\sigma(v)$ denotes the character labeling vertex v. We assume that the length m query sequence, referred to as pattern $P[1, m]$ from hereonwards, also consists of only characters from Σ. A length $k-1$ *walk* in G is a sequence of k vertices $v_1, v_2, ..., v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. Note that walks are allowed to repeat vertices. For notation purposes in algorithms and proofs, let $N^-(v)$ and $N^+(v)$ be the sets of inward and outward neighbors of vertex v, and let $deg^-(v) = |N^-(v)|$ and $deg^+(v) = |N^+(v)|$ be the indegree and outdegree of vertex v. Let $d(u, v)$ be the number of edges on the shortest path from vertex u to vertex v. We assume that G is at least weakly connected so that $|E| \geq |V|-1$. For a 2D array A, we will use $A[i]$ to denote the i^{th} row of A.

[2]We generated genome variation graphs from human chromosome variation data from the 1000 Genomes Project (Phase 2) [8].

B. Sequence-to-graph Alignment

We formally define the problem as follows:

Problem 1 (Sequence-to-Graph Alignment)**.** *Given a vertex-labeled graph $G = (V, E, \sigma : V \to \Sigma)$ over alphabet Σ and a sequence $P[1, m] \in \Sigma^+$, determine over all walks in G the minimum cost of alignment with P. In the alignment, Δ_{del} is the cost of deleting a character from P, Δ_{ins} is the cost of inserting a character into P, and $\Delta_{i,v}$ is the cost of aligning the character at position i in P to the character at vertex v in G.*

Several interesting theoretical results have been proven regarding sequence alignment on labeled graphs. This problem was first addressed for the application of pattern matching in hypertext [28]. In this early work, an algorithm was presented that runs in time $\tilde{O}(|E|m)$[3] on a graph $G = (V, E)$, where the pattern P is of length m. This was subsequently improved to $O(|E|m)$ time by Navarro [26]. Amir et al. [28] also demonstrated that the problem is NP-complete when edits are allowed to the graph. This hardness result was later strengthened to show NP-hardness holds even for the cases where the graph is labeled with a binary alphabet [14] and when the graph is a de Bruijn graph [29]. Importantly, the $O(|E|m)$-time algorithm is provably optimal under popular computational complexity assumptions. In particular, it was proven that an algorithm running in time $O(|E|^{1-\varepsilon}m + |E|m^{1-\varepsilon})$ for any constant $\varepsilon > 0$ would violate the Strong Exponential Time Hypothesis (SETH) [13]. This lower bound was later strengthened in several ways, including conditioning it on likely weaker assumptions in complexity theory [30], and proving that even with polynomial-time preprocessing of G, an algorithm that can match a pattern to the graph in time $O(|E|^{\delta}m^{\beta})$ where $\delta < 1$ or $\beta < 1$ would violate SETH [31].

C. Dynamic-Programming Approach

The standard dynamic-programming approach makes use of the following recurrence:

$$C[0, v] = 0$$
$$C[i, v] = \min_{u \in N^-(v)} \begin{cases} C[i-1, v] + \Delta_{del} \\ C[i-1, u] + \Delta_{i,v} \\ C[i, u] + \Delta_{ins} \end{cases} \quad (1)$$

The approach proceeds in m stages, with stage i aligning the character at position i in P to G. At each stage, array C of size $|V|$ records the best alignment score of walks ending at each vertex $v \in V$. The operation with cost $\Delta i, v$ of aligning the character at position i in P to the character at vertex v in G will be referred to as a match with cost Δ_{match} when $\sigma(v) = P[i]$, or as a mismatch with cost $\Delta_{mismatch}$ when $\sigma(v) \neq P[i]$, or as a substitution for both cases collectively. As shown in Figure 1, deletions are calculated by adding deletion cost to the cost on the vertex in the previous row, substitutions

[3]$\tilde{O}(\cdot)$ suppresses polylogarithmic factors.

are calculated by adding substitution costs to the cost on the incoming neighbors in the previous row, and insertion costs are calculated by adding insertion costs to the cost of the final cost on the incoming neighbors in the current row. The process of computing the updated cost using values from the current row and incorporating insertions is called *insertion propagation*. This is most simply implemented using a recursive procedure, as shown in Algorithm 3, but applied to the entire graph G.

D. Compressed Sparse Row (CSR) Representation

We make use of the Compressed Sparse Row (CSR) format [32], a classical method for storing sparse graphs. CSR stores vertex neighborhoods contiguously in an edge array of size $|E|$. An offset array of size $|V|$ stores the start point of each vertex v's neighbors in the edge array. Iterating through the neighbor list for a vertex v in CSR requires an $O(1)$ lookup in the vertex array for the start offset, and then $O(\deg^+(v))$ steps to traverse the edges in the edge array. Finally, a label array of size $|V|$ stores the DNA label $\sigma(v)$ associated with each vertex v.

E. Shared-memory Analysis Model

We use the ***work-span model*** with binary forking [33] to analyze algorithms in this paper. The ***work*** W is the total number of operations required in a computation (i.e., the time to run on one core), and the ***span*** T_∞ is the longest sequence of serial operations in a computation (i.e., the time on infinitely many cores). Under binary forking, tasks can fork into two child tasks, so spawning n tasks takes $O(\log n)$ time. That is, a parallel-for loop with n iterations and $O(1)$ work per iteration has $O(\log n)$ span. The ***parallelism***, or maximum possible speedup of an algorithm on infinitely many processors, is the work divided by the span.

To analyze concurrent writes to the same memory location, we use a CREW shared-memory model, where all processors can concurrently read and exclusively write to the same memory location. In the case of multiple concurrent writes having different values, we pessimistically assume that parallel write conflicts to the same memory location are resolved by serializing the operations to that location. The resolution of values can be done with priority updates [34], which compare the results of compiler-supported compare-and-swap operations until the minimum value has gone into the target memory address.

A parallel algorithm is ***work-optimal*** if it performs no more than a constant factor of operations in total compared to the best serial algorithm for the problem.

III. Algorithm

We begin this section by proving the key result on how to avoid data dependencies between insertions in sequence-to-graph alignment using "character graphs," which facilitate breaking up a row computation in the DP table into independent subproblems. As mentioned in the introduction, the main challenge to parallelization is dependencies between insertions in the genome graph. Next, we will show how to use character graphs for parallelization in ParSGA, and conclude with theoretical analysis that shows that ParSGA is work-optimal and has a relatively low span.

A. Insertion parallelization via character graphs

First, we will show that *insertion operations are not required for alignment correctness on edges into vertices that match the pattern character* $P[i]$ when calculating the i-th row in the DP table. That is, when aligning character $P[i]$, every edge (u, v) such that $\sigma(v) = P[i]$ can be ignored.

This observation gives rise to character graphs that are the same as the original graph but with all the edges corresponding to a given character removed.

Definition 1. *Given a character $x \in \Sigma$ and a genome graph $G = (V, E)$, the corresponding* ***character graph*** *$G_x = (V_x, E_x)$ where $V_x = V$ and $E_x = \{(u, v) \in E \mid \sigma(v) \neq x\}$.*

In DNA variation graphs, $\Sigma = \{T, C, G, A\}$, with $|\Sigma| = 4$. Therefore, given an input genome graph G, each nucleotide has a corresponding character graph G_T, G_C, G_G, or G_A as defined in Definition 1. An example of a variation graph and its resulting character graphs is presented in Figure 3.

When computing the i-th row of the DP table (corresponding to the pattern character $P[i]$), ParSGA uses the *corresponding character graph* $G_{P[i]}$ as the input graph rather than the original input graph G and parallelizes over the *connected components* in $G_{P[i]}$. In the context of directed graphs, connected components are taken to mean weakly connected components. To enable efficient parallelization, the character graphs should ideally contain many independent connected components without data dependencies between

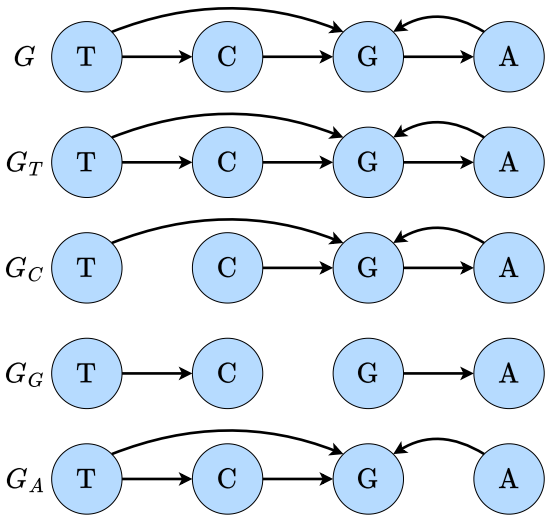

Fig. 3. Input Graph G and Character Graphs G_T, G_C, G_G, G_A: When aligning a character to the input graph, the corresponding character graph can be used for insertion computations. A character graph omits the input graph's edges into vertices labeled with the corresponding character.

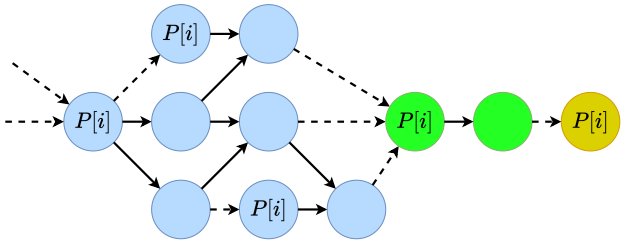

Fig. 4. Insertion operations are localized to each of character graph $G_{P[i]}$'s connected components. Dotted edges indicate the edges into $P[i]$-labeled vertices that are cut to disconnect G into $G_{P[i]}$'s distinctly colored connected components.

them. Figure 4 illustrates how ParSGA parallelizes over the components while calculating insertions serially within each connected component.

To demonstrate the parallelism resulting from the character-graph technique, Figure 6 shows that in human genome graphs, the maximum connected component sizes in character graphs are several orders of magnitude smaller than the original graph sizes. Section IV details how we generated the input datasets. In practice, genome graphs contain a near-linear graph topology and a well-distributed alphabet over the vertex labels. As a result, removing the edges for a particular nucleotide to generate a character graph disconnects the original genome graph into numerous significantly smaller connected components.

Given the genome graph in Compressed Sparse Row (CSR) graph representation (defined in Section II), character-graph construction takes linear time in the number of edges, which is a small-order term compared to the DP algorithm for sequence-to-graph alignment. To generate the character graphs in linear time, we first count character labels that do not match the pattern character using parallel prefix, and then allocate and populate the resulting smaller offset and adjacency arrays.

B. Correctness of Character Graph Decomposition

With the objective of constructing character graphs in mind, we formalize and prove our claim that if a vertex v (in the genome graph) has the same label $\sigma(v)$ matching pattern character $P[i]$ (in the query sequence), then v's score in the i-th row of the DP table will not be improved by insertions. Intuitively, this holds for the following reason: if the insertion propagation uses some path on the i^{th} stage and traverses an edge e going into a vertex v with label $P[i]$, then we can instead use the same path on the $(i-1)^{th}$ stage up until the start of e. From there, we use e to obtain the same value on the i^{th} iteration. This modification will never increase the score for a given vertex.

Note that $\Delta_{match} \leq \Delta_{mismatch}, \Delta_{del}, \Delta_{ins}$ in the scoring function of Recurrence 1. In fact, Δ_{match} is typically 0 as matching character imposes no edit distance, while all other quantities are positive numbers. We make no assumption on

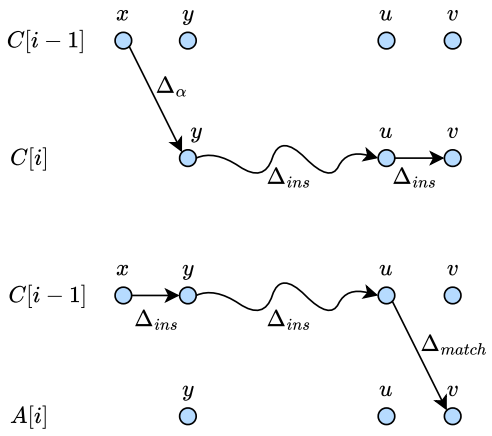

Fig. 5. The vertices indicated in the proof of Lemma 1 shown with pre-insertion A scores and post-insertion C scores on stages $i-1$ and i. The alignment represented on top can be replaced with the alignment represented on the bottom without increasing the cost.

the topology of G, so our claim also holds true for arbitrary cyclic graphs.

Formulation and analysis of this claim benefit from additional notation. Let $A[i, v]$ be the pre-insertion score for $P[i]$ on vertex v.

$$A[i,v] = \min_{u \in N^-(v)} \begin{cases} C[i-1,v] + \Delta_{del} \\ C[i-1,u] + \Delta_{i,v} \end{cases} \tag{2}$$

The array C remains defined as it was in Equation 1, and can also be referred to as the post-insertion score.

Lemma 1. *For all $v \in V$, if $\sigma(v) = P[i]$, then $A[i,v] = C[i,v]$, i.e., the i^{th} pre-insertion score for v is equal to the i^{th} post-insertion score for v,*

Proof. If $i = 0$, then $C[0,v] = A[0,v] = 0$ for all $v \in V$. Assume $i \geq 1$. By definition $A[i,v] \geq C[i,v]$. For the other side of the inequality, we have that

$$\begin{aligned} C[i,v] &= A[i,y] + \Delta_{ins} \cdot d(y,v) \\ &= C[i-1,x] + \Delta_\alpha + \Delta_{ins} \cdot d(y,v) \end{aligned}$$

for some $x, y \in V$, where $\Delta_\alpha \in \{\Delta_{i,y}, \Delta_{del}\}$. See Figure 5. Note that x and y need not be distinct. Let us further assume that $d(y,v) \geq 1$ (If $d(y,v) = 0$, then $C[i,v] = A[i,y]$ already). Since $d(y,v) \geq 1$, we can write

$$C[i,v] = C[i-1,x] + \Delta_\alpha + \Delta_{ins} \cdot d(y,u) + \Delta_{ins}$$

for some $u \in N^-(v)$. Note that if $d(y,v) = 1$, then $y = u$.

Next, we have

$$\begin{aligned} A[i,v] &\leq C[i-1,u] + \Delta_{match} \\ &\leq C[i-1,x] + \Delta_{ins} \cdot d(x,u) + \Delta_{match} \\ &\leq C[i-1,x] + \Delta_{ins} + \Delta_{ins} \cdot d(y,u) + \Delta_{match} \\ &\leq C[i-1,x] + \Delta_{ins} + \Delta_{ins} \cdot d(y,u) + \Delta_{\alpha} \\ &= C[i,v]. \end{aligned}$$

□

Lemma 3 allows us to perform insertion propagation in parallel on subgraphs of G. Note that Lemma 3 can also be extended to apply to affine gap penalties. Performing insertion propagation for a single character $P[i]$ in the character graph $G_{P[i]}$ takes $|E_{P[i]}| \leq |E|$ work. Taking the sum over the entire m rows of the DP table, the total work of computing insertions using the character graphs for each row is $\sum_{i=1}^{m} |E_{P[i]}| = O(m|E|)$.

We also define here the maximum component size $cc_{max} = \max_{\sigma \in \Sigma} \max_{cc \in G_\sigma} |E_{cc}|$, where $cc \in G_\sigma$ represents a connected component in G_σ, and $|E_{cc}|$ is the number of edges in the connected component cc.

C. ParSGA Algorithm

ParSGA, shown in Algorithm 1, parallelizes the dynamic programming used in past methods by splitting deletion work over vertices, substitution work over edges, and insertion work over character-graph components. In preprocessing up to Line 6, ParSGA constructs G_T, G_C, G_G, G_A required for insertions. In Line 8, ParSGA initializes the first row of the dynamic programming table per the base case in 1. After initialization, Line 10 loops over the characters in P. For each position i in P, the loops at Lines 11, 13, 15 compute the deletions, substitutions, and insertions from 1. Note that the deletion and substitution loops at Lines 11 and 13 can be done in parallel at the same time to compute pre-insertion A scores before the insertion loop.

The insertion loop at Line 15 makes use of insertion parallelization over $G_{P[i]}$. On each connected component in the character graph $G_{P[i]}$, the serial insertion algorithm [26] for insertion propagation on arbitrary graphs is applied, as reproduced in Algorithms 2 and 3. Deletions and substitutions use G instead of $G_{P[i]}$ at each iteration i.

D. Work and Span Analysis

In the remainder of this section, we will analyze the work and span (and therefore parallelism) of ParSGA. First, we show that ParSGA is work-optimal.

Lemma 2. *ParSGA is work-optimal with $W = O(m|E|)$.*

Proof. Let W_{del}, W_{sub}, and W_{ins} be the work used for deletion, substitutions, and insertions, respectively. We have that the total work $W = O(W_{del} + W_{sub} + W_{ins})$. Noting that $W_{del} = O(m|V|)$, $W_{sub} = O(m|E|)$, and $W_{ins} = O(m \cdot \max_{\sigma \in \Sigma} |E_\sigma|) = O(m|E|)$, it follows that $W = O(m|E|)$. □

Algorithm 1: ParSGA

Data: Graph $G = (V, E, \sigma : V \mapsto \Sigma)$ and pattern $P \in \Sigma^m$
Result: Alignment score array $C[m]$

1 **parallel for** $\sigma \in \Sigma$ **do**
2 $E_\sigma \leftarrow \emptyset$;
3 **parallel for** $(u, v) \in E$ **do**
4 **parallel for** $\sigma \in \Sigma \setminus \sigma(v)$ **do**
5 $E_\sigma \leftarrow (u, v)$;
6 **parallel for** $\sigma \in \Sigma$ **do**
7 $G_\sigma \leftarrow (V, E_\sigma, \sigma : V \mapsto \Sigma)$;
8 **parallel for** $v \in V$ **do**
9 $C[0, v] \leftarrow 0$;
10 **for** $i \in 1..m$ **do**
11 **parallel for** $v \in V$ **do**
12 $C[i, v] \leftarrow C[i-1, v] + \Delta_{del}$;
13 **parallel for** $(u, v) \in E$ **do**
14 $C[i, v] \leftarrow \min\{C[i, v], C[i-1, u] + \Delta_{i,v}\}$;
15 **parallel for** $cc \in G_{P[i]}$ **do**
16 $Insert(cc, C[i])$;
17 **return** $C[m]$

Algorithm 2: Insert

Data: Connected component $cc = (V, E, \sigma : V \mapsto \Sigma) \subseteq G_{P[i]}$ and array $C[i]$
Result: Updated array $C[i]$

1 **for** $(u, v) \in E$ **do**
2 Propagate$((u, v), C[i])$

Next we show that $ParSGA$ has relatively low span. To this end, we first prove the following lemma bounding the maximum in-degree by cc_{max} whenever $|\Sigma| \geq 2$ [4].

Lemma 3. *For $|\Sigma| \geq 2$, we have $cc_{max} \geq \max_{v \in V} \deg^-(v)$.*

Proof. Let $u = \arg\max_{v \in V} \deg^-(v)$. Observe that for any $\alpha \neq \sigma(u)$, that the edges (w, u) for $w \in N^-(u)$ are in the same connected component of G_α. Hence, the component of G_α containing u has size at least $\deg^-(u)$. □

Next, we apply Lemma 3 to analyze the span of ParSGA.

Lemma 4. *ParSGA has span*

$$T_\infty = O(m \cdot (\log|E| + cc_{max})).$$

Proof. Let $T_{\infty_{del}}$, $T_{\infty_{sub}}$, and $T_{\infty_{ins}}$ be the span of deletion, substitutions, and insertions, respectively. We have that the span is their maximum, that is, $T_\infty = O(T_{\infty_{del}} + T_{\infty_{sub}} + T_{\infty_{ins}})$. First, $T_{\infty_{del}} = O(m \cdot \log|V|)$ because the time required at each iteration over P to spawn $|V|$

[4] The case where $|\Sigma| = 1$ can be handled trivially.

Algorithm 3: Propagate

Data: Connected component $cc = (V, E, \sigma : V \mapsto \Sigma) \subseteq G_{P[i]}$ and array $C[i]$
Result: Updated array $C[i]$

```
if C[i,v] > Δ_ins + C[i,u] then
    C[i,v] ← Δ_ins + C[i,u];
    for w ∈ N⁺(v) do
        Propagate((v,w), C[i])
```

threads is $O(\log |V|)$ under the binary forking model, as defined in Section II-E. The other terms are defined similarly, but with additional contributors to the longest sequential path of operations. For $T_{\infty_{sub}}$, observe that the worst case serialization of concurrent writes to the same location is bounded by the maximum indegree in the G, hence $T_{\infty_{sub}} = O(m \cdot (\log |E| + \max_{v \in V} \deg^-(v)))$. We have $T_{\infty_{ins}} = O(m \cdot (\log \max_{\sigma \in \Sigma} |cc(G_\sigma)| + cc_{max}))$ as this sums the cost of parallelizing over the largest number of connected components over the character graphs plus the cost of serializing over the maximum connected component size over the character graphs. Applying that $\max_{v \in V} \deg^-(v) \leq cc_{max}$ from Lemma 3, we conclude that

$$\begin{aligned} T_\infty &= O\left(m \cdot (\log |V| + \log |E| + \max_{v \in V} \deg^-(v) \right. \\ &\quad \left. + \log \max_{\sigma \in \Sigma} |cc(G_\sigma)| + cc_{max})\right) \\ &= O(m \cdot (\log |E| + cc_{max})). \end{aligned}$$

□

As shown in Figure 6 and Table I, in practice, cc_{max} is orders of magnitude smaller than $|E|$, so ParSGA achieves good parallelism and should achieve significant parallel speedup, as we will confirm in Section IV.

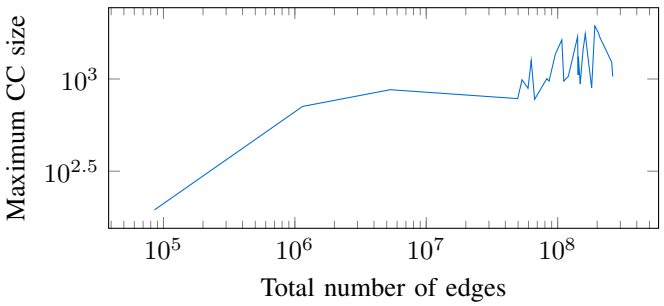

Fig. 6. The relationship between the maximum connected component (CC) size over the character graphs versus the number of edges on human genome variation graphs.

IV. Experimental Evaluation

This section contains an empirical evaluation of the runtime and parallel scalability of ParSGA using short, long, and ultra-long reads on multiple genome graphs drawn from different species.

Graph	$\lvert V \rvert$	$\lvert E \rvert$	cc_{max}
BRCA1	83,268	85,422	196
LRC	1,099,857	1,144,498	738
MHC1	5,138,363	5,318,019	897
E. coli	16,793,055	17,055,833	3,100
CHR21	48,868,640	49,650,499	783
CHR22	52,423,214	53,608,349	991
CHRY	59,433,754	59,495,356	891
CHR19	60,980,865	62,948,541	1,259
CHR20	64,853,076	66,783,153	776
CHR18	80,357,609	82,765,902	1,005
CHR17	83,545,051	86,031,268	974
CHR16	93,072,814	95,962,958	1,361
CHR15	104,968,213	107,547,065	1,630
CHR14	109,347,425	111,460,891	975
CHR13	117,838,149	120,654,511	1,035
CHR12	137,742,382	141,850,415	1,698
CHR11	139,073,853	143,375,375	1,053
CHR10	139,552,787	143,806,599	1,319
CHR9	144,794,207	148,580,237	941
Arabidopsis	133,882,945	151,557,201	1,405
CHR8	150,986,670	155,886,499	1,424
CHRX	158,738,836	162,393,701	1,759
CHR7	163,880,289	168,897,337	1,369
CHR6	176,169,656	181,509,018	897
CHR5	186,204,492	191,791,400	1,937
CHR4	196,912,162	203,000,099	1,800
CHR3	203,882,100	210,066,755	1,677
CHR2	250,312,065	257,829,316	1,234
CHR1	255,754,180	262,621,670	1,031

TABLE I
GENOME GRAPHS USED IN OUR STUDY, ORDERED BY THEIR SIZES, AND THE MAXIMUM COMPONENT SIZES OF THEIR CORRESPONDING CHARACTER GRAPHS. BRCA1, LRC, AND MHC1 CORRESPOND TO SMALLER GRAPHS FROM SPECIFIC REGIONS OF HUMAN CHROMOSOMES.

A. Experimental Setup

We implemented ParSGA in C++ using GCC 10.3.0 and parallelized using OpenMP 4.5. We enabled compiler flags `-O3 -march=native`. We ran experiments on a machine with dual AMD EPYC 9534 2.45 GHZ CPUs with 128 cores, 3TB DDR5 DRAM, 4TB NVMe, and 100 Gbps InfiniBand NICs.

Datasets and genome graphs. We constructed variation graphs for all the human chromosomes using VG toolkit [16] on reference genomes and variant files from the 1000 Genomes Project (Phase 2) [8]. We also used graphs for targeted chromosomal regions known to be rich in variants (BRCA1, LRC, and MHC1), because they were used in many earlier studies [19], [22], [23]. To extend our tests to other species, we chose the microbial organism *E. Coli* and the model plant *Arabidopsis thaliana*. We used the graph-loading component of PaSGAL [19] for reading variation graph toolkit format [16] files. Table I details the graphs used in this evaluation.

Connected component analysis. For each genome graph, the largest connected component size is calculated by first generating the four corresponding character graphs G_T, G_G, G_C, G_A, and taking the size of the largest component in any of these graphs. Ambiguous nucleotides, in which the true characters were unknown, were treated as matches. ParSGA

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	845.08	38.47	482.23	324.38	0.08
2	440.78	19.26	251.10	170.42	0.15
4	226.17	9.71	129.35	87.10	0.28
8	114.72	4.87	65.66	44.10	0.56
16	57.68	2.44	33.02	22.12	1.11
32	29.10	1.22	16.67	11.13	2.20
64	15.29	0.62	8.85	5.75	4.18
128	10.41	0.42	5.99	3.88	6.14

TABLE II
RUNTIME (S) VS CORES ON CHROMOSOME 1 GRAPH WITH READS OF LENGTH 250.

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	503.84	23.10	288.13	192.61	0.08
2	262.81	11.65	149.93	101.17	0.15
4	134.72	5.83	77.26	51.59	0.28
8	68.30	2.92	39.23	26.11	0.56
16	34.32	1.46	19.72	13.10	1.12
32	17.34	0.74	9.97	6.59	2.21
64	9.12	0.37	5.31	3.41	4.20
128	6.24	0.25	3.61	2.28	6.15

TABLE III
RUNTIME (S) VS CORES ON CHROMOSOME 1 GRAPH WITH READS OF LENGTH 150.

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	142.47	7.35	81.85	53.27	0.09
2	86.07	3.68	49.19	33.20	0.14
4	43.83	1.86	25.14	16.82	0.28
8	22.21	0.93	12.77	8.48	0.55
16	11.18	0.47	6.44	4.26	1.09
32	5.63	0.23	3.24	2.14	2.17
64	2.88	0.12	1.66	1.09	4.24
128	1.80	0.07	1.04	0.67	6.80

TABLE IV
RUNTIME (S) VS CORES ON CHROMOSOME 21 GRAPH WITH READS OF LENGTH 250.

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	85.03	4.41	48.81	31.81	0.09
2	51.42	2.23	29.32	19.85	0.14
4	26.16	1.12	14.99	10.04	0.28
8	13.25	0.56	7.61	5.06	0.55
16	6.67	0.28	3.84	2.54	1.10
32	3.36	0.14	1.93	1.28	2.18
64	1.72	0.07	0.99	0.65	4.25
128	1.07	0.04	0.62	0.40	6.82

TABLE V
RUNTIME (S) VS CORES ON CHROMOSOME 21 GRAPH WITH READS OF LENGTH 150.

uses code from the GAP Benchmark [35] for computing connected components, and from Parlaylib [36] for parallel sorting to store connected components contiguously. Note that the character graphs do not change between iterations matching successive characters of the pattern, nor do they change for different patterns. Thus, their construction can be considered pre-processing cost when the genome graph remains fixed. On 128 cores, the times spent (in seconds) in preprocessing to construct the character graphs and label their components were 23.4 for Chromosome 1, 8.8 for Arabidopsis, 3.1 for Chromosome 21, and 1.3 for E. coli. The file sizes of the input graphs versus ParSGA's additional files for character graphs were 5.1GB vs. 34GB for Chromosome 1, 2.7GB vs. 17.6GB for Arabidopsis, 0.9GB vs. 5.6GB for Chromosome 21, and 0.3GB vs. 2.0GB for E. coli.

The sizes of individual genome graphs, and the corresponding largest connected component sizes (whose distribution is also illustrated in Figure 6), are shown in Table I. It is readily observable that for all but the smallest graph BRCA1, the largest connected component size falls within a tight range that is uncorrelated to the genome graph size. This size, in relation to the corresponding genome graph size, determines the effectiveness and scalability of our insertion parallelization strategy. In fact, this ratio between $|E|$ and cc_{max} determines the degree of parallelism available for insertion. As can be observed, $|E|$ is three orders of magnitude larger than cc_{max} for a graph with a million edges, and five orders of magnitude larger for large graphs exceeding 100 million edges.

B. Aligning Short Reads

For short reads which tend to have fixed lengths, we generated 150bp and 250bp simulated reads with Mason2 software [37] to reflect the current state-of-the-art Illumina sequencers. To smooth out variations, We report mean alignment time per read, averaged over aligning five reads.

The run-time of ParSGA and its individual components for 250bp and 150bp reads when aligning to variation graphs for human Chromosomes 1 and 21, respectively, are shown in Tables II, III, IV, and V. The tables also contain the breakdown of the run-time spent in deletion, substitution, and insertion phases over all the bases (characters) in the read (pattern). The components parallelize well, and ParSGA achieves a speedup for 250bp of $\sim 55\times$ on 64 cores and $\sim 81\times$ on 128 cores on Chromosome 1. For Chromosome 21, which is more than five times smaller, the speedups reduce to $\sim 49\times$ on 64 cores and $\sim 79\times$ on 128 cores. Runtime and speedup plots for 250bp on Chromosome 1, the larger of the two short read lengths and the largest of the human chromosome graphs, are shown in Figures 7 and 8. In the legends, deletions, substitutions, and insertions are abbreviated as "del.", "subst.", and "ins.", respectively. On this, and also all other datasets, the reduced speedup achieved when going from 64 to 128 cores is due to the communication overhead of spanning dual 64-core CPUs in two sockets, rather than any scaling limitation of the algorithm itself.

Figure 11 shows that the sizes of the connected components in character graphs provide a good distribution. As each core computes over several connected components, their cumulative sizes and underlying structure can cause load imbalance. The figure contains plot of such load imbalance, computed as the ratio of the longest runtime on a core divided by the average runtime over all cores. The imbalance slowly grows with an increase in the number of cores, and is a modest $1.12\times$ even on 128 cores.

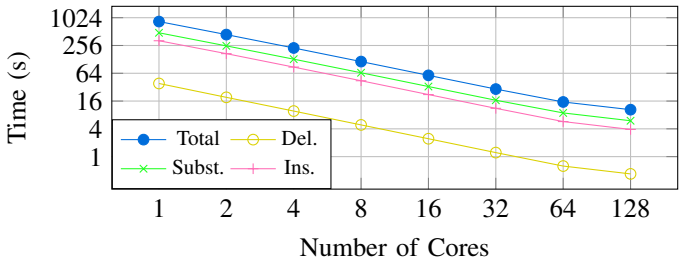

Fig. 7. Runtime (s) vs Cores on CHR1 Graph with Reads of Length 250.

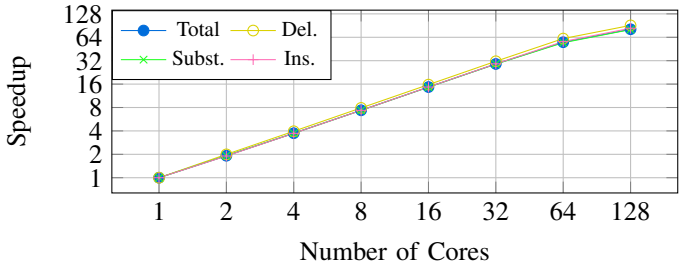

Fig. 8. Speedup vs Cores on CHR1 Graph with Reads of Length 250.

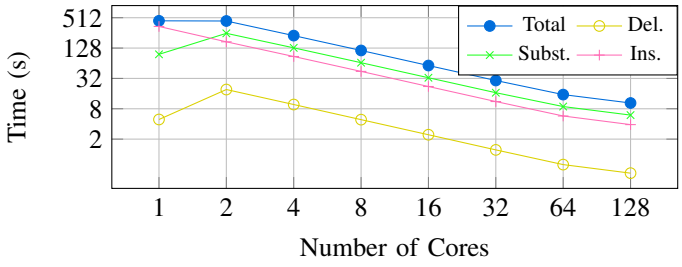

Fig. 9. Runtime (s) vs Cores on Chromosome 1 Graph with Reads of Length 250. Single-core is serial algorithm.

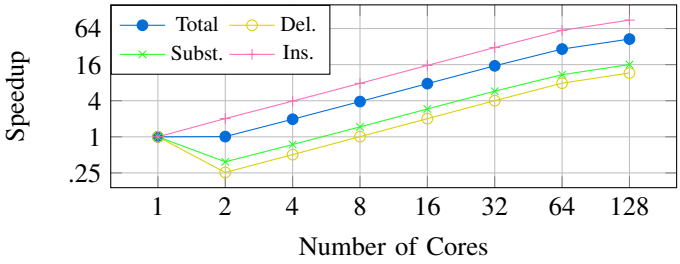

Fig. 10. Speedup vs Cores on Chromosome 1 Graph with Reads of Length 250. Single-core is serial algorithm.

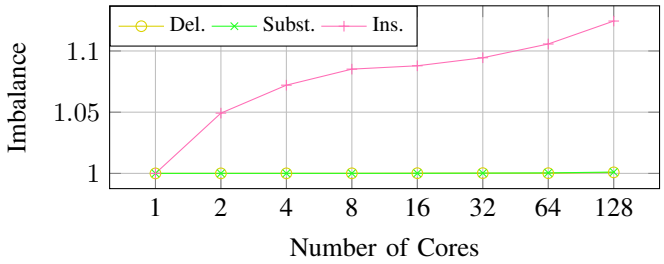

Fig. 11. Imbalance ($\text{Time}_{max}/\text{Time}_{avg}$) vs Cores on CHR1 Graph with Reads of Length 250.

While ParSGA demonstrates good scaling, it is important to compare its performance against the serial dynamic-programming algorithm to measure real speedups achieved when transitioning from serial code to the parallel code. Figures 9 and 10 show such a comparison on human Chromosome 1 when aligning 250bp reads. Here, the serial algorithm is used when running on one core and ParSGA is used in all other cases. At two cores, ParSGA still (narrowly) underperforms the serial algorithm, limiting the speedup achieved on 128 cores to $\sim 43\times$.

Further analysis of the different operations contributing to the runtime reveals interesting observations (Figures 9 and 10). Insertions parallelize well compared to the serial algorithm, providing speedups from the get-go. Both substitutions and deletions are slower on two cores when compared to the serial algorithm. Though the parallel runtime of these operations shows near-perfect scaling, it takes 6 cores for substitutions and 8 cores for deletions to surpass the runtime of the serial algorithm. Recall that novel parallelization of insertions is the main algorithmic contribution in this paper, while both substitution and deletion phases lend themselves to trivial parallelization, albeit with the unavoidable overhead. The overhead in parallel deletion has marginal effect on dragging overall performance. On the Chromosome 1 graph with 250bp reads, deletions take up only 4.9 seconds out of a total runtime of 444.6 seconds required for the serial algorithm, a mere 1.1%. Similar results are observed for all the graphs studied in Table I.

From a practical standpoint, it makes sense not to parallelize substitutions and deletions when running on fewer cores, and instead run the serial algorithm for them in conjunction with parallel algorithm for the insertion phase. This will further improve run-times and speedups of ParSGA on a smaller number of cores. However, the exact number of cores at which the transition of switching from serial execution of substitutions and deletions to their corresponding parallel execution depends on architectural features of the system on which the code is run. This can be measured through diagnostic runs and ParSGA can be auto-tuned to the underlying architecture to bring out such savings.

C. Aligning Long Reads

Unlike short-read sequencers, long-read sequencers produce variable-length reads with much higher error rates. For this study, we simulated PacBio reads with pbsim2 software [38]. On Chromosome 1, the reads simulated have a mean length of 16,872.8 bp. On Chromosome 21, the reads simulated have a mean length of 16,796.6 bp. As before, alignment times are reported per read, averaged over five reads.

ParSGA aligns long reads to large chromosome-scale variation graphs in reasonable time. When running on 128 cores, ParSGA took ∼12 minutes to align a long read to the Chromosome 1 variation graph (Table VI), and ∼2 minutes to align a long read to the Chromosome 21 variation graph (Table VII). The parallel algorithm is particularly useful in aligning long sequences, where the serial algorithm takes several hours to completion. All three operations - substitutions, insertions, and deletions - scale uniformly well, as shown in Figure 12.

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	60,050.22	2,597.63	34,171.32	23,281.19	0.07
2	31,327.07	1,299.34	17,789.96	12,237.66	0.14
4	16,085.61	655.77	9,179.69	6,250.03	0.27
8	8,139.81	328.37	4,649.38	3,153.20	0.53
16	4,092.18	164.72	2,337.51	1,582.73	1.05
32	2,068.13	82.46	1,184.06	795.16	2.09
64	1,082.98	42.00	626.08	410.73	3.98
128	734.02	27.71	420.26	275.22	5.88

TABLE VI
RUNTIME (S) VS CORES ON CHR1 GRAPH WITH READS OF AVERAGE LENGTH 16,872.8.

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	10,060.20	494.00	5,759.97	3,806.18	0.08
2	6,117.74	249.34	3,477.70	2,388.57	0.13
4	3,107.21	124.85	1,772.87	1,206.88	0.26
8	1,572.48	62.53	900.10	608.06	0.52
16	791.67	31.38	453.98	304.62	1.04
32	398.82	15.70	228.83	153.01	2.06
64	203.59	7.92	117.35	77.50	4.03
128	126.34	4.88	72.39	47.58	6.50

TABLE VII
RUNTIME (S) VS CORES ON CHROMOSOME 21 GRAPH WITH READS OF AVERAGE LENGTH 16,796.6.

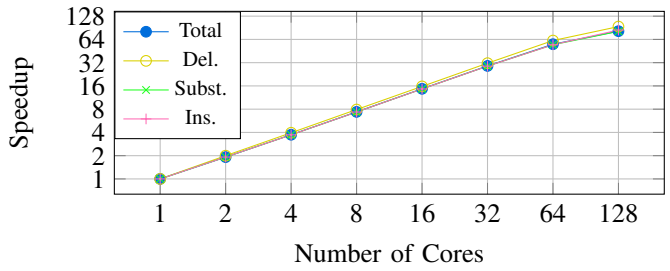

Fig. 12. Speedup vs Cores on Chromosome 1 Graph with Reads of Average Length 16,872.8.

To provide experiments on ultra-long reads, we simulated 1 million bp reads, representative of Oxford Nanopore sequencers. When running on 128 cores, ParSGA took 12 hours, measuring at 5.92 GCUPS, to align one read to the Chromosome 1 graph, and 2hrs. and 12 mins., measuring at 6.15 GCUPS, to align one read to the Chromosome 21 graph.

An important observation to make is that the runtime of ParSGA increases linearly with an increase in pattern size. The parallelization is on a per-character basis, and thus its scaling is unaffected by the length of the sequence being aligned. For this reason, the GCUPS metric is similar for Chromosome 1 across four different read lengths when examining Tables VI, II, III, and its ultra-long read runtime, and for Chromosome 21 across four different read lengths when examining Tables VII, IV, V, and its ultra-long read runtime.

No implementations were available for a direct comparison on ParSGA's intended use case, the parallel optimal alignment of long reads to graphs. For this purpose, we ran the batch-parallel optimal sequence-to-graph aligner PaSGAL [19] on Chromosome 21 on PacBio reads on 128 cores. PaSGAL took 1hr. 30 mins. per read, compared to ParSGA taking 2 mins. PaSGAL solves local alignment, a recurrence with identical time complexity as our semiglobal alignment recurrence (Equation 1), so ParSGA's 55× speedup over PaSGAL is in line with ParSGA's 43× speedup over our own serial code.

D. Evaluation on Variantion Graphs for Other Species

Most sequence-to-graph alignment works only target human genetic variation data [15]–[18], which by itself makes a compelling use case. Though the experimental analysis presented so far is also similarly focused, here we present additional results to illustrate the effectiveness of ParSGA for variation graphs from other species. For this purpose, we chose the widely studied microbial organism *E. Coli* and the model plant *Arabidopsis thaliana*. We used the E. coli genome graph from Garrison et. al. [39]. We constructed an Arabidopsis variation graph using data from the Ensembl Genomes resource [40]. The respective maximum connected component sizes of the character graphs derived from them are included in Table I, alongside the human chromosome graphs.

Table VIII and Figure 13 show short read runtimes and speedups on the E. Coli and Arabidopsis genome graphs, compared with human Chromosome 1 and 21 graphs. ParSGA performs equally well, achieving maximum speedups of 81×, 81×, 79×, and 88× on Chromosome 1, Arabidopsis, Chromosome 21, and E. coli, respectively. Compared to the serial algorithm on the same read data, ParSGA achieved similar maximum speedups of 43×, 45×, 40×, and 46× on Chromosome 1, Arabidopsis, Chromosome 21, and E. coli, respectively. Experimental data for long reads on E. Coli and Arabidopsis are shown in Tables IX and X. As indicated by their lower maximum GCUPS metrics of 4.95 and 5.40, both E. coli and Arabidopsis require longer runtimes for alignment relative to their graph sizes than human Chromosome 1 and 21, which have maximum GCUPS of 5.88 and 6.50 respectively. This is likely due to the graph structures present in the variations of such species.

V. RELATED WORK

To date, practical work on parallelizing algorithms for sequence-to-graph alignment focused on parallelizing multiple reads simultaneously. For example, the PaSGAL algorithm [19] uses Single Instruction Multiple Data (SIMD) operations and inter-task parallelism across reads to accelerate sequence-to-graph alignment on directed acyclic graphs (DAGs). Addi-

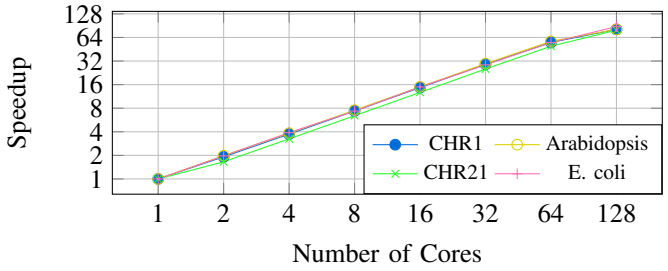

Fig. 13. Speedup vs Cores by Variation Graphs with Reads of Length 250.

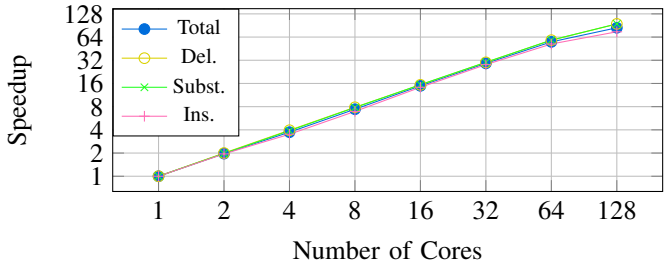

Fig. 14. Speedup vs Cores on E. Coli Graph with Reads of Average Length 18,011.0.

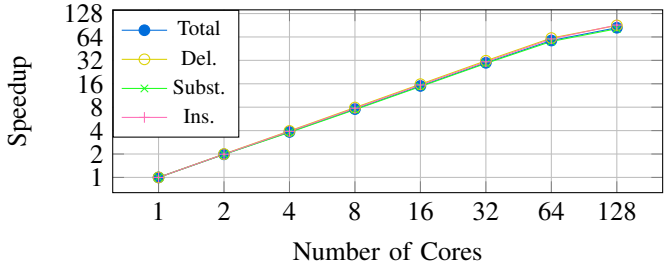

Fig. 15. Speedup vs Cores on Arabidopsis Graph with Reads of Average Length 16,569.6.

tional research effort has been devoted to parallelizing multiple reads using SIMD operations, again for multiple reads [15]. Another prior work focused on parallelizing alignment of multiple reads for a GPU-based architecture [22].

On the theoretical side, researchers proposed an algorithm for the exact matching problem that takes poly-logarithmic time but $O(|V|^3 m)$ work [41]. The algorithm uses prefix-sum techniques on the matrix multiplications of the transition matrices, which correspond to the different symbols in the given sequence. However, this algorithm is not work-optimal because the DP solution takes $O(|E|m)$ time.

Cores	CHR1	Arabidopsis	CHR21	E. coli
$\|E\|$	2.6×10^8	1.5×10^8	5.0×10^7	1.7×10^7
1	845.08	542.17	142.47	62.02
2	440.78	273.61	86.07	31.45
4	226.17	141.26	43.83	15.90
8	114.72	71.75	22.21	8.44
16	57.68	36.23	11.18	4.15
32	29.10	18.27	5.63	2.17
64	15.29	9.48	2.88	1.13
128	10.41	6.68	1.80	0.70

TABLE VIII
RUNTIME (S) VS CORES FOR DIFFERENT VARIATION GRAPHS WITH READS OF LENGTH 250.

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	4731.88	182.21	2592.45	1957.18	0.06
2	2406.07	91.13	1311.11	1003.77	0.13
4	1268.36	46.08	664.33	557.75	0.24
8	638.62	23.21	332.57	282.12	0.47
16	317.52	11.87	168.06	137.15	0.95
32	162.16	6.06	86.09	69.70	1.87
64	85.35	3.12	44.39	37.60	3.54
128	56.02	1.92	27.34	26.12	5.40

TABLE IX
RUNTIME (S) VS CORES ON E. COLI GRAPH WITH READS OF AVERAGE LENGTH 18,011.0

Cores	Total	Deletions	Substitutions	Insertions	GCUPS
1	37,698.51	1,334.15	22,326.27	14,038.03	0.06
2	19,027.33	667.27	11,298.57	7,061.42	0.12
4	9,805.64	336.72	5,880.54	3,588.12	0.23
8	4,984.52	168.51	3,001.12	1,810.34	0.45
16	2,511.92	84.34	1,515.36	908.57	0.88
32	1,266.40	42.18	763.52	457.45	1.75
64	654.63	21.55	398.51	232.47	3.39
128	448.57	14.81	273.62	154.28	4.95

TABLE X
RUNTIME (S) VS CORES ON ARABIDOPSIS GRAPH WITH READS OF AVERAGE LENGTH 16,569.6.

VI. CONCLUSIONS

We presented ParSGA, the first parallel algorithm and its multicore implementation for sequence alignment to a genome graph. ParSGA relies on a novel parallelization of insertion propagation, which works by breaking the genome graph into numerous significantly smaller components with the crucial property that an insertion could not propagate from one component to another. ParSGA outputs optimal alignments using high-end multiprocessors in several minutes for long reads, and is the only viable option for aligning ultra-long Oxford Nanopore reads measuring 1 Mbp and up.

Besides long reads, ParSGA is useful for dynamically evolving graphs. In this scenario, a read is first mapped to the current graph to determine its mapping location, following which the variants present in the read are incorporated into the graph before aligning the next read. Algorithms that require multiple reads to align at once are rendered useless in this scenario.

ParSGA's intra-task parallelization could be used in combination with existing inter-task parallelization methods to further increase the number of processors that can be utilized for aligning large sequence datasets against genome-scale graphs. As alignment capabilities scale with complex, rapidly growing datasets, graphical pangenomics will better be able to support further advances in biological research.

REFERENCES

[1] A. Dilthey, C. Cox, Z. Iqbal, M. R. Nelson, and G. McVean, "Improved genome inference in the mhc using a population reference graph," *Nature genetics*, vol. 47, no. 6, pp. 682–688, 2015.

[2] J. M. Eizenga, A. M. Novak, J. A. Sibbesen, S. Heumos, A. Ghaffaari, G. Hickey, X. Chang, J. D. Seaman, R. Rounthwaite, J. Ebler *et al.*, "Pangenome graphs," *Annual review of genomics and human genetics*, vol. 21, pp. 139–162, 2020.

[3] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison, "Genome graphs and the evolution of genome inference," *Genome research*, vol. 27, no. 5, pp. 665–676, 2017.

[4] S. Chen, P. Krusche, E. Dolzhenko, R. M. Sherman, R. Petrovski, F. Schlesinger, M. Kirsche, D. R. Bentley, M. C. Schatz, F. J. Sedlazeck *et al.*, "Paragraph: a graph-based structural variant genotyper for short-read sequence data," *Genome biology*, vol. 20, no. 1, pp. 1–13, 2019.

[5] H. P. Eggertsson, S. Kristmundsdottir, D. Beyter, H. Jonsson, A. Skuladottir, M. T. Hardarson, D. F. Gudbjartsson, K. Stefansson, B. V. Halldorsson, and P. Melsted, "Graphtyper2 enables population-scale genotyping of structural variation using pangenome graphs," *Nature communications*, vol. 10, no. 1, pp. 1–8, 2019.

[6] G. Hickey, D. Heller, J. Monlong, J. A. Sibbesen, J. Sirén, J. Eizenga, E. T. Dawson, E. Garrison, A. M. Novak, and B. Paten, "Genotyping structural variants in pangenome graphs using the vg toolkit," *Genome biology*, vol. 21, no. 1, pp. 1–17, 2020.

[7] "Pangaia," Nov 2020. [Online]. Available: https://www.pangenome.eu/

[8] . G. P. Consortium *et al.*, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[9] D. Gibney, G. Hoppenworth, and S. V. Thankachan, "Simple reductions from formula-sat to pattern matching on labeled graphs and subtree isomorphism," in *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, H. V. Le and V. King, Eds. SIAM, 2021, pp. 232–242. [Online]. Available: https://doi.org/10.1137/1.9781611976496.26

[10] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de bruijn graphs to genome assembly," *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[11] S. Garg, M. Rautiainen, A. M. Novak, E. Garrison, R. Durbin, and T. Marschall, "A graph-based approach to diploid genome assembly," *Bioinform.*, vol. 34, no. 13, pp. i105–i114, 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty279

[12] A. Amir, M. Lewenstein, and N. Lewenstein, "Pattern matching in hypertext," *J. Algorithms*, vol. 35, no. 1, pp. 82–99, 2000. [Online]. Available: https://doi.org/10.1006/jagm.1999.1063

[13] M. Equi, R. Grossi, V. Mäkinen, A. Tomescu *et al.*, "On the complexity of string matching for graphs," in *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[14] C. Jain, H. Zhang, Y. Gao, and S. Aluru, "On the complexity of sequence-to-graph alignment," *J. Comput. Biol.*, vol. 27, no. 4, pp. 640–654, 2020. [Online]. Available: https://doi.org/10.1089/cmb.2019.0066

[15] C. A. Darby, R. Gaddipati, M. C. Schatz, and B. Langmead, "Vargas: heuristic-free alignment for assessing linear and graph read aligners," *Bioinformatics*, vol. 36, no. 12, pp. 3712–3718, 2020.

[16] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin *et al.*, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature biotechnology*, vol. 36, no. 9, pp. 875–879, 2018.

[17] D. Kim, J. M. Paggi, C. Park, C. Bennett, and S. L. Salzberg, "Graph-based genome alignment and genotyping with hisat2 and hisat-genotype," *Nature biotechnology*, vol. 37, no. 8, pp. 907–915, 2019.

[18] P. Ivanov, B. Bichsel, and M. Vechev, "Fast and optimal sequence-to-graph alignment guided by seeds," *bioRxiv*, 2021.

[19] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru, "Accelerating sequence alignment to graphs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 451–461.

[20] M. Rautiainen and T. Marschall, "Graphaligner: rapid and versatile sequence-to-graph alignment," *Genome biology*, vol. 21, no. 1, pp. 1–28, 2020.

[21] H. Li, X. Feng, and C. Chu, "The design and construction of reference pangenome graphs with minigraph," *Genome biology*, vol. 21, no. 1, pp. 1–19, 2020.

[22] Z. Feng and Q. Luo, "Accelerating sequence-to-graph alignment on heterogeneous processors," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[23] J. Ma, M. Cáceres, L. Salmela, V. Mäkinen, and A. I. Tomescu, "Chaining for accurate alignment of erroneous long reads to acyclic variation graphs," *Bioinformatics*, vol. 39, no. 8, p. btad460, 2023.

[24] G. Chandra and C. Jain, "Sequence to graph alignment using gap-sensitive co-linear chaining," in *International Conference on Research in Computational Molecular Biology*. Springer, 2023, pp. 58–73.

[25] M. Rautiainen, V. Mäkinen, and T. Marschall, "Bit-parallel sequence-to-graph alignment," *Bioinformatics*, vol. 35, no. 19, pp. 3599–3607, 2019.

[26] G. Navarro, "Improved approximate pattern matching on hypertext," *Theor. Comput. Sci.*, vol. 237, no. 1-2, pp. 455–463, 2000. [Online]. Available: https://doi.org/10.1016/S0304-3975(99)00333-3

[27] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," May 2008. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[28] A. Amir, M. Lewenstein, and N. Lewenstein, "Pattern matching in hypertext," *Journal of Algorithms*, vol. 35, no. 1, pp. 82–99, 2000.

[29] D. Gibney, S. V. Thankachan, and S. Aluru, "The complexity of approximate pattern matching on de bruijn graphs," *(Accepted) RECOMB2022*, vol. abs/2201.12454, 2022. [Online]. Available: https://arxiv.org/abs/2201.12454

[30] D. Gibney, G. Hoppenworth, and S. V. Thankachan, "Simple reductions from formula-sat to pattern matching on labeled graphs and subtree isomorphism," in *Symposium on Simplicity in Algorithms (SOSA)*. SIAM, 2021, pp. 232–242.

[31] M. Equi, V. Mäkinen, and A. I. Tomescu, "Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless seth fails," in *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 2021, pp. 608–622.

[32] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[34] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons, "Reducing contention through priority updates," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, 2013, pp. 152–163.

[35] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[36] G. E. Blelloch, D. Anderson, and L. Dhulipala, "Parlaylib-a toolkit for parallel algorithms on shared-memory multicore machines," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 507–509.

[37] M. Holtgrewe, "Mason–a read simulator for second generation sequencing data," *Technical Report FU Berlin*, 2010.

[38] Y. Ono, K. Asai, and M. Hamada, "Pbsim2: a simulator for long-read sequencers with a novel generative model of quality scores," *Bioinformatics*, vol. 37, no. 5, pp. 589–595, 2021.

[39] E. Garrison, A. Guarracino, S. Heumos, F. Villani, Z. Bao, L. Tattini, J. Hagmann, S. Vorbrugg, S. Marco-Sola, C. Kubica *et al.*, "Building pangenome graphs," *Nature Methods*, pp. 1–5, 2024.

[40] A. D. Yates, J. Allen, R. M. Amode, A. G. Azov, M. Barba, A. Becerra, J. Bhai, L. I. Campbell, M. Carbajo Martinez, M. Chakiachvili *et al.*, "Ensembl genomes 2022: an expanding genome resource for non-vertebrates," *Nucleic acids research*, vol. 50, no. D1, pp. D996–D1003, 2022.

[41] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, 1980. [Online]. Available: https://doi.org/10.1145/322217.322232