DiVerG: Scalable Distance Index for Validation of Paired-End Alignments in Sequence Graphs

Ali Ghaffaari
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Alexander Schönhuth
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Tobias Marschall
Institute for Medical Biometry and Bioinformatics, Medical Faculty, Heinrich Heine University, Düsseldorf, Germany
Center for Digital Medicine, Heinrich Heine University, Düsseldorf, Germany

Abstract

Determining the distance between two loci within a genomic region is a recurrent operation in various tasks in computational genomics. A notable example of this task arises in paired-end read mapping as a form of validation of distances between multiple alignments. While straightforward for a single genome, graph-based reference structures render the operation considerably more involved. Given the sheer number of such queries in a typical read mapping experiment, an efficient algorithm for answering distance queries is crucial. In this paper, we introduce DiVerG, a compact data structure as well as a fast and scalable algorithm, for constructing distance indexes for general sequence graphs on multi-core CPU and many-core GPU architectures. DiVerG is based on PairG [27], but overcomes the limitations of PairG by exploiting the extensive potential for improvements in terms of scalability and space efficiency. As a consequence, DiVerG can process substantially larger datasets, such as whole human genomes, which are unmanageable by PairG. DiVerG offers faster index construction time and consistently faster query time with gains proportional to the size of the underlying compact data structure. We demonstrate that our method performs favorably on multiple real datasets at various scales. DiVerG achieves superior performance over PairG; e.g. resulting to 2.5–4x speed-up in query time, 44–340x smaller index size, and 3–50x faster construction time for the genome graph of the MHC region, as a particularly variable region of the human genome.

The implementation is available at: `https://github.com/cartoonist/diverg`

2012 ACM Subject Classification Mathematics of computing → Paths and connectivity problems; Applied computing → Computational genomics

Keywords and phrases Sequence graph, distance index, read mapping, sparse matrix

Digital Object Identifier 10.4230/LIPIcs.WABI.2025.10

Related Version *Extended Version*: `https://doi.org/10.1101/2025.02.12.637964`

Supplementary Material *Software (Source Code)*: `https://github.com/cartoonist/diverg` [21]
archived at `swh:1:rel:362b7551950ee369c01ada82d05b6c8345a3048d`

Funding This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 956229.

Acknowledgements This work was supported by the de.NBI Cloud within the German Network for Bioinformatics Infrastructure (de.NBI) and ELIXIR-DE (Forschungszentrum Jülich and W-de.NBI-001, W-de.NBI-004, W-de.NBI-008, W-de.NBI-010, W-de.NBI-013, W-de.NBI-014, W-de.NBI-016, W-de.NBI-022).

1 Introduction

Many genomic studies use read mapping as a central step in order to place the donor sequence reads into context relative to a reference. Several studies have repeatedly shown that the conventional reference assemblies, i.e. consensus genomes or genomes of individuals, do not capture the genomic diversity of the population and introduce biases [37, 22, 6, 2, 12]. Furthermore, the absence of alternative alleles in a linear reference can penalize correct alignments and lead to decreased accuracy in downstream analysis. To address this issue, augmented references, often in the form of *pangenome graphs*, have been developed by incorporating genomic variants observed in a population [16].

On the other hand, shifting from a linear structure to a graph introduces a variety of theoretical challenges. Recent progress in *computational pangenomics* has paved the way for using such references in practice as established algorithms and data structures for linear references cannot be seamlessly applied to their graphical counterparts [9, 16, 36]. One significant measure affected by this transition is the concept of *distance* between two genomic loci. Defining and computing distance relative to a single genome is inherently straightforward. However, in genome graphs, distance cannot be uniquely defined due to the existence of multiple paths between two loci, with the number of paths being theoretically exponential relative to the number of variants between them.

Determining genomic distances between two loci emerges in several genomic workflows notably paired-end short-read mappings to sequence graphs. In paired-end sequencing, DNA fragments are sequenced from both their ends, where the unsequenced part in between introduces a gap between the sequenced ends. Read mapping is the process of determining the origin of each read relative to a reference genome through sequence alignment. There are several sources of ambiguity in finding the correct alignments. The distance between two pairs plays a crucial role in resolving alignment ambiguities, for example in repetitive regions [5]. An accurate alignment of one end can rescue the other end's alignment in case of ambiguities. Therefore, it is important to determine whether the distance between two reference loci, where two ends of a read could be placed, falls in a particular, statistically well motivated range $[d_1, d_2]$. This problem, referred to as the *Distance Validation Problem (DVP)*, is first formally defined in [27].

The distance between all pairs of candidate alignments corresponding to a paired-end reads should be validated in order to find the correct pairs. Additionally, the candidate alignments in pangenome graphs are often more abundant compared to a linear sequence due to the increased ambiguity in a pangnome reference caused by added variations. This fact, combined with the large number of reads in a typical read mapping workflow, necessitates efficient methods to answer the DVP, often by preprocessing the graph and constructing an index data structure. Therefore, the efficiency of the involved operation is crucial for rendering short-read-to-graph mapping practically feasible.

Long, third-generation sequencing reads (TGS) have spurred enormous enthusiasm in various domains of application, which may explain that the majority of read-to-graph mapping approaches focuses on such long reads. However, still, long TGS reads are considerably more expensive to produce than short reads.

Short next-generation sequencing reads are highly accurate, cheap, and available to nearly every sequencing laboratory today. This provides substantial motivation for delivering approaches that render short-read-to-graph mapping a viable option; in fact, this would free the way for using graph-based reference systems in many laboratories worldwide and enable their use for existing biobank-scale cohorts where short reads have already been produced.

Related Work

The distance between two nodes in a graph is determined by the paths that connect them and is typically associated with their shortest path. Identifying the shortest paths between two points in a graph is a prevalent problem across various application domains and is an extensively studied topic in computer science [14, 18, 19, 24, 26, 38, 15]. However, simply knowing the shortest path length is not always sufficient for addressing the DVP: when the shortest path length is less than d_1, it cannot determine whether a longer path of length d exists satisfying $d_1 \leq d \leq d_2$. On the other hand, the decision version of *longest path problem* and *exact-path length problem*, which respectively answer whether there is a path of at least length d or exactly length d in a graph, are shown to be NP-complete [35, 30].

In the context of sequence graphs, most sequence-to-graph aligners use heuristic approaches to estimate distances [20, 33]. Chang et al. [7] propose an exact method and indexing scheme to determine the minimum distance between any two positions in a sequence graph with a focus on seed clustering. Although their method is shown to be efficient in seed clustering, it cannot address the DVP as it only provides the minimum distance between two positions.

Jain et al. [27] propose PairG, a distance indexing method which, to the best of our knowledge, is the first method directly addressing the DVP. This method, similar to existing approaches, involves preprocessing the graph to create an index that answers distance queries efficiently. In PairG, the index data structure is a sparse Boolean matrix constructed from powers of the adjacency matrix of the graph. Once constructed, it can indicate in near-constant time whether a path exists between two nodes that meet the distance criteria. It benefits from general sparse storage format and employs standard sparse matrix algorithms to reduce space requirement and accelerate the index construction. However, the method cannot scale to handle whole genome graphs for large sequences. PairG, although applicable for small graphs, also does not run on many-core architectures such as GPU for sparse matrix computations due to its intense memory requirements.

Contributions

In this work, we propose DiVerG, an indexing scheme that offers fast exact solutions for the DVP in sequence graphs with significantly lower memory footprint and faster query and construction time than existing methods.

DiVerG enhances PairG to overcome the limitations of existing approaches. Our first contribution is new *dynamic* compressed formats, namely *rCRS* formats, for storing sparse Boolean matrices. They are *dynamic* in the sense that sparse matrix operations can be conducted directly on matrices in this format without decompression. These formats, although simple by design, provide considerably greater compression potential for sparse matrices representing the adjacency matrix of sequence graphs, or the powers thereof. The specific incorporation of the logic that supports the shape of adjacency matrices of sequence graphs facilitates significantly more compact representations than what can be achieved by merely utilizing the sparsity of the matrix in standard sparse formats.

Secondly, we propose algorithms for Boolean sparse matrix-matrix multiplication and matrix addition for matrices in rCRS formats. Our multiplication algorithm achieves enhanced performance through the bit-level parallelism that the encoded format immediately provides, while the addition algorithm runs in time proportional to the compressed size.

We tailored both algorithms and their implementations to be particularly powerful on massively parallel architectures, such as GPUs. The faster algorithms and more compacted encodings enable indexing of drastically larger graphs, previously infeasible with the prior state of the art. Our experiments show that DiVerG scales favorably with large sequence graphs at low memory footprint and significant compression ratio. Most importantly, DiVerG responds to distance queries in constant time in practice. Despite the prior work, DiVerG's query performance does not grow by the distance constraints parameters.

2 Problem Definition

2.1 Sequence Graphs

Sequence graphs provide compressed representations of sets of similar – evolutionary or environmentally related – genomic sequences [25]. They are typically defined as node-labelled *bidirected* graphs where each node contains a string label. Bidirectionality allows traversal in both forward and reverse directions, which accounts for the complementary structure of genomes. For theoretical analysis, we use an equivalent representation called a *character graph*, defined as $G(V, E, \lambda)$ where V and E are the node and edge sets, respectively, and $\lambda : V \to \Sigma$ assigns a *single* character from alphabet Σ to each node. Character graphs and general sequence graphs can be converted to each other in time linear in the size of the character graph [27]. By definition, in a character graph, the length of a sequence spelt out by a walk of length k is exactly k.

Lastly, a *chain graph* is a character graph that represents a single sequence which is equivalent to the sequential chaining of all (single-letter) nodes. A sequence graph G is called *sparse* if the average node degree in G is close to 1.

2.2 Distance Validation Problem

Fragments in sequencing libraries typically vary in size depending both on the sequencing technology in use and library preparation procedures. In our study, the distance between paired reads in a library is modelled by an interval indicating the expected lower (d_1) and upper bounds (d_2). This interval, which is referred to as *distance constraints* and denoted as (d_1, d_2), are assumed to be provided as input parameters. In Section A, we discussed how the distance constraints can be determined for a library.

As stated before, aligning short reads to a reference genome, whether linear or graph, can be ambiguous due to several factors such as repetitive regions in the genome, sequencing errors, and the genetic differences between the donor genome and the reference. In such cases, reads might have multiple candidate alignments. Utilizing the distance information in paired-end sequencing data can help identify the correct alignments (Figure 6b in Section A).

Two alignments are considered as paired if the the distance between them is consistent with the inferred fragment model. In addition to the distance, the orientation of two paired reads should also be as expected depending on the sequencing technology in use; e.g. one of the pair should be aligned on the forward strand while the other is on the reverse strand.

▶ **Problem 1** (Distance Validation). *Let r_1 and r_2 be two paired reads, and a_1, and a_2 their alignments against a sequence graph $G(V, E, \lambda)$. Having a_1 mapped to the forward strand implies a_2 to be on the reverse strand. Let v_1 and v_2 be the first nodes in the paths to which a_1 and a_2 are aligned, respectively (Figure 1). Assuming the fragment model is described by distance constraints (d_1, d_2), the Distance Validation problem is determining whether there exists a path from v_1 to v_2 of length $d \in [d_1, d_2]$. These two alignments are considered as* paired *if this condition is met.*

Due to the sheer number of queries in a typical read mapping experiment, the algorithm solving Problem 1 needs to be efficient.

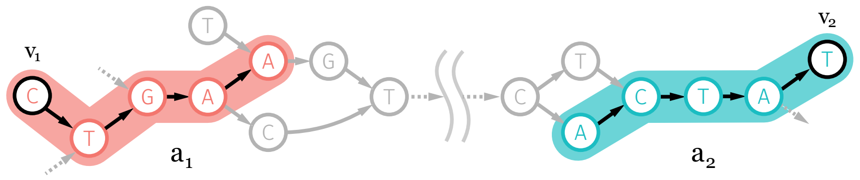


Figure 1 Alignments a_1 and a_2 correspond to paired reads r_1 = `CTGAA` and r_2 = `ATAGT` on a sequence graph (partially represented). Alignment a_1 starts from node v_1 and extends along the forward strand, while alignment a_2 starts from v_2 and extends along the reverse complement strand.

3 *k*-Walk Matrix as Distance Index

Let $\mathbf{A} = (a_{ij})$ denote the Boolean adjacency matrix of graph $G(V, E, \lambda)$, defined by $a_{ij} = 1$ if and only if $(i, j) \in E$. The $k-$th power of $\mathbf{A}$ has a special property: $\mathbf{A}^k = (a^k_{uv})$ determines the number of walks of length k between nodes u and v in G. The Boolean equivalent of $\mathbf{A}^k$, which can be achieved by replacing each non-zero entry in $\mathbf{A}^k$ with 1, can answer *existence* queries on walks of length k instead.

For the remainder of the paper, any mentioned adjacency matrices or their k-th powers implicitly refer to Boolean matrices.

▶ **Definition 2** (Boolean Matrix Operations). *Given two Boolean square matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *of order* n*, standard Boolean matrix operations are defined as follows:*

- *Addition:* $\mathbf{A} \vee \mathbf{B} = a_{ij} \vee b_{ij}$*,*
- *Multiplication:* $\mathbf{A} \cdot \mathbf{B} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$*,*
- *Power:* $\mathbf{A}^k = \mathbf{A} \cdot \mathbf{A}^{k-1}$ $(k > 0)$*, and* $\mathbf{A}^0 = \mathbf{I}$*,*

where $\vee$ *and* $\wedge$ *are Boolean disjunction (OR) and conjunction (AND) operators.*

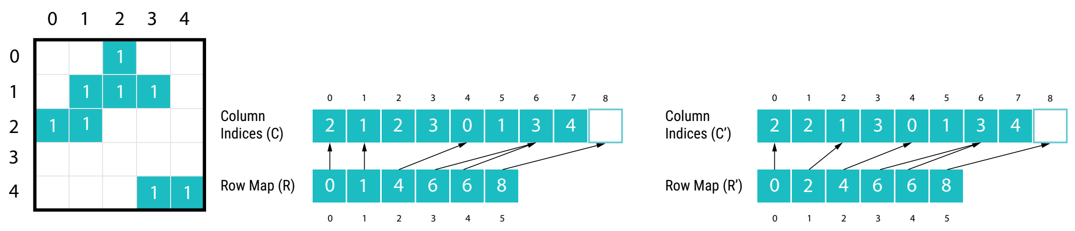
▶ **Definition 3** (Distance Index). *Given distance constraints* (d_1, d_2)*, a Boolean matrix* $\mathcal{T}$ *is called* distance index *relative to* (d_1, d_2) *if defined as:*

$$\mathcal{T} = \mathbf{A}^{d_1} \cdot (\mathbf{A} \vee \mathbf{I})^{d_2 - d_1}\,, \tag{1}$$

where I is the identity matrix, $\vee$ and $\cdot$ denote Boolean matrix addition and matrix-matrix multiplication, respectively, defined in Definition 2. Jain et al. [27] show that $\mathcal{T} = (\tau_{ij})$ can efficiently solve the *Distance Validation Problem* defined in Problem 1, i.e. if $\tau_{uv} = 1$, there exists at least one walk of length $d \in [d_1, d_2]$ from node u to v in the graph.

Generally, the diameter of sequence graphs is very large in practice, and the number of edges is on the order of the number of nodes, i.e. $|E| \sim |V|$. This implies that most sequence graphs exhibit sparse adjacency matrices. Therefore, given that distance constraints are considerably smaller than the graph, i.e. $(d_2 - d_1) \ll |V|$, one can expect $\mathcal{T}$ to be sparse. The common approaches for computing Equation (1) leverage the sparsity of the matrices, aiming to operate at time and space complexities proportional to the number of non-zero elements. Sparse matrices are typically stored in sparse formats such as the *Compressed Row Storage* (CRS) [4] and the calculation of Equation (1) relies on the sparse matrix-matrix multiplication (SpGEMM) and sparse matrix addition (SpAdd) algorithms [4].

▶ **Definition 4** (Compressed Row Storage). *Given Boolean squared matrix* $\mathbf{A}$ *with* n *rows and* $nnz(\mathbf{A})$ *number of non-zero elements, the* Compressed Row Storage *or* Compressed Sparse Row *format of* $\mathbf{A}$ *is a row-based representation consisting of two one-dimensional arrays* (C, R)*; where*



(a) Compressed Row Storage (CRS). **(b)** Range CRS (rCRS).

Figure 2 An example Boolean matrix represented in CRS and rCRS format.

- C (column indices), *of size* $nnz(\mathbf{A})$, *stores the column indices of non-zero elements in* $\mathbf{A}$ *in row-wise order;*
- R (row map), *of length* $n+1$, *is defined such that, for each row index* $i \in [0, n-1]$, $R(i)$ *gives the index in* C *where the column indices for row* i *begin. The final entry in* R *is set to* $|C|$, *i.e.* $R(n) = nnz(\mathbf{A})$.

▶ Remark 5. Row map array R defines the boundaries of each row in the column index array C: the column indices of non-zero entries of row i are located in the half-open interval $[R(i), R(i+1))$ of C. This partitions C into consecutive subsequences $C = C_0C_1 \cdots C_{n-1}$, referred to as the *row decomposition* of C, where $C_i = [C(R(i)) \cdots C(R(i+1)-1)]$.

By definition, array C does not need to be sorted within each partition C_i in CRS format. When the entries of each row in C are sorted, we refer to this as *sorted CRS*. Figure 2a demonstrate a toy example of a Boolean matrix in (sorted) CRS format. The required space for storing sparse matrix $\mathbf{A}_{n\times m}$ in CRS format is $\Theta(n + \text{nnz}(\mathbf{A}))$.

Accessing element a_{ij} in $\mathbf{A}$ using CRS representation can be reduced to searching for column index j in the part of array C corresponding to row i, as specified by row map array R. That is, one searches for j in C_i, the i-th partition of the row decomposition of C. This search can be facilitated in sorted CRS via binary search, instead of inspecting all column indices in the row. With z being the maximum number of non-zero values in any row in $\mathbf{A}$, searching for j in sorted CRS takes $O(\log(z))$, due to the binary search performed on the sorted entries. As mentioned before, given the sparsity of matrix $\mathbf{A}$, which implies that z is very small, accessing elements in sorted-CRS amounts to requiring constant time in practice.

To date, many algorithms have been proposed for parallel sparse matrix-matrix multiplication in formats such as CRS or other similar variants [32, 3]. Since standard matrix operations establish fundamental components of numerous applications, most of them are optimized for various hardware architectures, in particular for architectures that support massive parallelization, such as GPUs. Although the CRS format is often sufficient for general sparse matrix storage and relevant operations, it is computationally prohibitive when it is used for computing the k-th power of adjacency matrices as well as the resulting matrix from Equation (1) for large sequence graphs. On the other hand, sequence graph adjacency matrices offer optimization opportunities that are not typically found in general sparse matrices.

3.1 Observations

The distance index $\mathcal{T}$ computed by Equation (1) quickly becomes infeasible for large graphs due to its space requirements. For example, consider a human pangenome graph constructed using the autosomes of the GRCh37 reference genome and incorporating variants from the

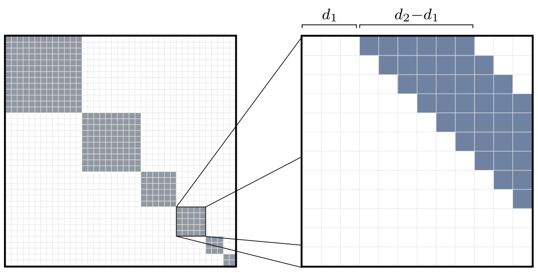


Figure 3 Schematic illustration of the structure of matrix $\mathcal{T}$ with distance constraints (d_1, d_2) for a chain graph with multiple components.

1000 Genome Project. Constructing the distance index $\mathcal{T}$ for this graph with distance criteria $d_1 = 150$ and $d_2 = 450$ results in a matrix of order 2.8B with approximately 870B non-zero values. Assuming that each non-zero value consumes 4 bytes, the total space required for the final distance matrix in sorted CRS would be about 3.5 TB.

Even for smaller graphs, storage requirements are impractical for GPU architectures as they have much smaller memory than the accessible main memory in a CPU. Such many-core architectures are beneficial for faster computation of SpGEMM, which is the computational bottleneck for constructing the distance index.

A key observation for identifying the compression potential is the structure of the distance matrix. Genome graphs usually consist of multiple connected components corresponding to each genomic region or chromosome. If the nodes are indexed such that their indices are *localized* for each region, the distance index $\mathcal{T}$ is a *block-diagonal matrix*, where each block corresponds to a different genomic region or chromosome.

Further examinations reveal that in each row, non-zero values are also localized within a certain range – often grouped into a few clusters of consecutive columns – if nodes are indexed in a "near-topological order". This is because almost all edges in sequence graphs are local, and it is reflected in the matrix as long as the ordering that defines node indices mostly preserves this locality – i.e. adjacent nodes appear together in the ordering. This observation can be seen clearly in the distance index constructed for a chain graph with distance constraints (d_1, d_2) as shown in Figure 3.

4 Method

4.1 Total Order on Node Set

The nodes of a graph are indexed along the rows of its adjacency matrix with the same order governing the columns as well. This induces a total order on the set of nodes in the graph, assigning each node a unique *index*. Different total orders yield different adjacency matrices. As will be shown in Section 4.2, our method greatly benefits if the total order on the node set maximizes the *locality* of the indices for adjacent nodes; i.e. adjacent nodes are assigned nearby indices by such ordering. It can be shown that this problem is equivalent to finding a sparse matrix with minimum bandwidth by permuting its rows and columns. The problem of finding such ordering has been proven to be NP-Complete [31].

Several heuristic algorithms have been proposed to minimize the bandwidth of sparse matrices by reordering rows and columns [10, 11, 8]. However, applying these methods to our problem presents some key limitations. First, although our method logically views sequence graphs as character graphs, in practice it works with string-labelled graphs for space efficiency. This requires preserving the sequential ordering of bases within nodes of a string-labelled graph, a constraint that existing heuristics do not necessarily meet when applied to the character graph representation. Breaking this intra-node ordering introduces memory overhead and fragmentation, impacting graph traversal performance in downstream tasks through reduced memory locality and increased cache misses. Second, when applied directly to string-labelled graphs while preserving node ordering, these methods are less effective at reducing bandwidth compared to their performance on character graphs.

To address these limitations, we propose a new heuristic algorithm inspired by the Cuthill–McKee (CM) algorithm [10, 11]. The new algorithm directly operates on string-labelled sequence graphs. In essence, the algorithm traverses the graph in a breadth-first search similar to the CM. However, instead of prioritizing visiting nodes by their degrees, it prioritizes nodes that have a predecessor with a lower *topological sort order* (Algorithm 1). Note that, in practice, sequence graphs, particularly variation graphs, are DAGs in the majority of relevant cases. If input sequence graphs are not acyclic, e.g. in de Bruijn graphs, we resort to a "semi-topological ordering" established by "dagifying" the graph, i.e. by running topological sort algorithm on the graph while the traversal algorithm ignores any edges forming a cycle.

Our experimental results (in Section 5) demonstrate that the ordering achieved by Algorithm 1 offers a very effective heuristic while assigning sequential indices to bases within each node. This can be justified by the fact that sequence graphs generally preserve the linear structure of the sequences from which they are constructed, and the variations in these sequences are often only local in the genomic coordinates, thereby affecting the graph topology locally. As a result, prioritizing sibling nodes for index assignment while following topological ordering tends to localize indices of adjacent nodes, thereby minimizing bandwidth.

Algorithm 1 A heuristic for defining a total order on the node set that minimizes bandwidth.

```
Require: Sequence graph $G = (V, E)$ (string-labelled)
 1: function MinBandwidthTotalOrder($G$)
 2:     $P \leftarrow []$                                              ▷ Output permutation
 3:     $R \leftarrow \emptyset$                                              ▷ Visited nodes
 4:     $T \leftarrow$ SemiTopologicalSortOrder($G$)
 5:     $V_{\text{sorted}} \leftarrow$ Sort $V$ according to $T$
 6:     $Q \leftarrow$ priority queue with key$(v) = \min(\{T[v]\} \cup \{T[w] : w \in G.\text{predecessor}(v)\})$
 7:     for each source node $s \in V_{\text{sorted}}$ with $G.\text{indegree}(s) = 0$ do
 8:         if $s \notin R$ then
 9:             $Q.\text{enqueue}(s)$                    ▷ Enqueue source node if not already visited
10:             $R \leftarrow R \cup \{s\}$                                  ▷ Mark s as visited
11:         while $Q$ is not empty do       ▷ Process each connected component starting from s
12:             $v \leftarrow Q.\text{dequeue}()$                 ▷ Minimum key value has highest priority
13:             Append $v$ to $P$
14:             for $u \in G.\text{successor}(v)$ and $u \notin R$ do
15:                 $Q.\text{enqueue}(u)$                      ▷ Enqueue unvisited successors of v
16:                 $R \leftarrow R \cup \{u\}$                              ▷ Mark u as visited
17:     report $P$
```

4.2 Range Compressed Row Storage (rCRS)

DiVerG aims to exploit particular properties of the distance index matrix $\mathcal{T}$ mentioned in Section 3.1, as well as its sparsity, to reduce space and time requirements. Our method introduces a new sparse storage format, which will be explained subsequently. This format relies on transforming the standard CRS by replacing column index ranges with their lower and upper bounds to achieve high compression rate. Later, we will propose tailored algorithms to compute sparse matrix operations directly on this compressed data structure without incurring additional overhead for decompression.

▶ **Definition 6** (Minimum Range Sequence)**.** *Let A be a* sorted *sequence of distinct integers. A* "range sequence" *of A is another sequence, A_r, constructed from A in which any disjoint subsequence of consecutive integers is replaced by its first and last values. When such subsequence contains only one integer i, it will be represented as $[i, i]$ in A_r. The minimum sized A_r is defined as* minimum range sequence *of A, denoted as $\rho(A)$.*

▶ **Definition 7** (Range Compressed Row Storage)**.** *For a sparse matrix $\mathbf{A}$ and its sorted CRS representation (C, R), we define Range Compressed Row Storage or rCRS$(\mathbf{A})$ as two 1D arrays (C', R') which are called* column indices *and* row map *arrays respectively:*

- *$C' = \rho(C_0) \cdots \rho(C_{n-1})$, where C_i is partition i in the row decomposition of C;*
- *R' is an array of length $n + 1$, in which $R'(i)$ specifies the start index of $\rho(C_i)$. The last value is defined as $R'(n) = |C'|$.*

It can be easily shown that rCRS can be constructed from sorted CRS in linear time with respect to the number of non-zero values. Accessing element a_{ij} in $\mathbf{A}$ represented in rCRS reduces to determining whether j lies within any range in row i. Since the ranges are disjoint and sorted, this operation can be performed using binary search which requires $O(\log(\hat{z}))$ time in the worst case, where $\hat{z}$ is the maximum size of C'_i across all rows i. Note that, unlike the CRS format, the size of C' is no longer equal to $\mathrm{nnz}(\mathbf{A})$.

The space complexity of the rCRS representation decisively depends on the distribution of non-zero values in the rows of the matrix. This representation can substantially compress the column indices array C – as low as $O(n)$ compared to $O(nnz(A))$ in CRS – if the array contains long distinct ranges of consecutive integers. The limitation of this representation is that it can be twice as large when there is no stretch of consecutive indices in C. It is important to note that the compression rate can directly influence query time, as it is proportional to the compressed representation of non-zero values.

To mitigate this issue, we define a variant of rCRS, which is referred to as *Asymmetrical Range CRS* (*aCRS*). In the worst case scenario, aCRS format takes as much space as the standard CRS while keeping the same time complexity $O(\log(\hat{z}))$ for the query operation. We elaborate on the aCRS definition and related analysis in Section B.

Although aCRS theoretically provides a more compact representation compared to rCRS, the experimental results (in Section 5) show that it occupies as much space as rCRS when storing distance index $\mathcal{T}$ for sequence graphs in practice. This is due to the fact that the ordering defined in Section 4.1, together with the topology of sequence graphs, rarely results in isolated column indices. Consequently, aCRS holds two integers per range, similar to rCRS. Considering this fact and the relative ease of implementing efficient sparse matrix algorithms with rCRS compared to aCRS, particularly on GPU, we base our implementation on the rCRS format. This can be observed in the index constructed for a chain graph (Figure 3), which often resembles the general structure of sequence graphs.

4.3 Range Sparse Boolean Matrix Multiplication (rSpGEMM)

Given sparse matrices $\mathbf{A}_{m\times n}$ and $\mathbf{B}_{n\times s}$ where m, n, and s are positive integers, SpGEMM is an algorithm that computes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ using sparse representations of two input matrices. In this section, we introduce a new SpGEMM algorithm, called *Range SpGEMM* or *rSpGEMM* for short, computing matrix-matrix multiplication when input and output matrices are in the rCRS format. The algorithm is designed to scale well on both multi-core architectures (CPUs) as well as many-core architectures (GPUs).

Similar to most parallel SpGEMM algorithms, our rSpGEMM follows Gustavson's algorithm [23]. Algorithm 2 illustrates its general structure, in which $A(i,:)$ refers to row i of matrix $\mathbf{A}$. This notation can be generalized to $B(j,:)$ and $C(i,:)$ in Algorithm 2 specifying row j and i in B and C, respectively. Note that Algorithm 2 only iterates over non-zero entries of $\mathbf{A}$ and $\mathbf{B}$.

The algorithm computes $\mathbf{C}$ one row at a time. To compute $C(i,:)$, it iterates over non-zero values in row $A(i,:)$ and computes its contribution to $C(i,:)$ by multiplying it to non-zero values in row B_j; i.e. the term $a_{ij} \cdot B(j,:)$ in Line 3. This contribution is then *accumulated* with the partial results computed from previous iterations. The row accumulation can be simplified to $C(i,:) \leftarrow C(i,:) + B(j,:)$ in Boolean matrices as a_{ij} is 1.

Since computing each row of the final matrix is independent of the others, $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ can be seen as multiple independent vector-matrix multiplication $C(i,:) = A(i,:) \cdot \mathbf{B}$. For simplicity, we will only focus on the equivalent vector-matrix multiplication.

Algorithm 2 Gustavson's algorithm.

```
Require: Sparse matrices $\mathbf{A}_{m \times n}$ and $\mathbf{B}_{n \times s}$
1: for $i := 0 \to m-1$ do
2:     for $a_{ij} \in A(i,:)$ do
3:         $C(i,:) \leftarrow C(i,:) + a_{ij} \cdot B(j,:)$
```

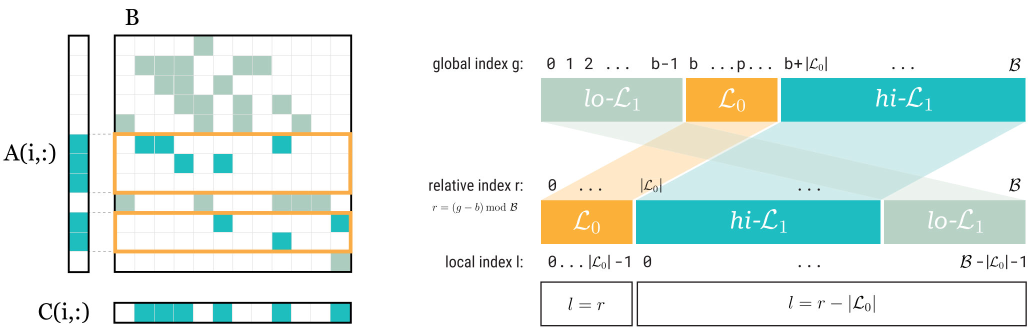
Given that the output matrix is also in the sparse format, knowing the number of non-zero values in each row of $\mathbf{C}$ is essential before storing the calculated C_i in the final matrix. This issue is usually tackled using a two-phase approach: firstly, the *symbolic* phase, delineates the structure of $\mathbf{C}$, primarily corresponding to the computation of its row map array. Secondly, the *numeric* phase, carries out the actual computation of $\mathbf{C}$.

Similar to SpGEMM algorithms based on Gustavson's, the fundamental aspect of rSpGEMM resides in three anchor points: the data structure employed for row accumulation, memory access pattern to minimize data transfer latency, and distribution of work among threads in hierarchical parallelism. In the following, we address each of these points.

4.3.1 Bi-Level Banded Bitvector as Accumulator

The row accumulation in Boolean SpGEMM, explained in Section 4.3, can be seen as the union of all column indices in $B(j,:)$ for all j where $a_{ij} \neq 0$. Figure 4a schematically depicts the row accumulation in computing a row of the final matrix. It is important to highlight that row accumulation in rSpGEMM is carried out on ranges of indices that are, by definition, disjoint and sorted.

Our method employs a dense bitvector, namely *Bi-level Banded Bitvector* (BBB), as an accumulator in rSpGEMM. From a high-level point of view, this bitvector supports two main operations: `scatter` and `gather`. The `scatter` operation stores a range of column indices at once, which essentially involves setting a range of sequential bits in the bitvector to one. Conversely, the `gather` operation retrieves the union of column indices stored by the `scatter` method in the form of the minimum range sequence (Definition 6). Computing $C(i,:)$ is essentially a combination of `scatter`ing partial results and `gather`ing the final accumulated entries using BBB.


B
A(i,:)
C(i,:)
global index g: 0 1 2 ... b-1 b ...p... b+|$\mathcal{L}_0$| ... $\mathcal{B}$
lo-$\mathcal{L}_1$
$\mathcal{L}_0$
hi-$\mathcal{L}_1$
relative index r: 0 ... |$\mathcal{L}_0$| ... $\mathcal{B}$
$r = (g - b) \bmod \mathcal{B}$
$\mathcal{L}_0$
hi-$\mathcal{L}_1$
lo-$\mathcal{L}_1$
local index l: 0...|$\mathcal{L}_0$|-1 0 ... $\mathcal{B}$-|$\mathcal{L}_0$|-1
$l = r$
$l = r - |\mathcal{L}_0|$

(a) **(b)**

Figure 4 (a) Row accumulation in Boolean matrices can be viewed as the union of all non-zero values in rows of **B** that are specified by the non-zero values in A_i. (b) BBB hierarchical structure and indexing.

The `scatter` operation is a bit-parallel operation that sets bits corresponding to index range $[s, f]$ in a series of W-bit operations. Each set bit in the bitvector indicates absence/-presence of the corresponding column in the final result. The pseudocode of the `scatter` operation can be found in Algorithm 3, in Section C.

Each cluster of set bits in the final bitvector represents an interval in the final row $C(i,:)$, with each interval corresponding to an entry pair in the rCRS format. In the symbolic phase, the `gather` operation counts the number of entries in rCRS with regards to non-zeros in row $C(i,:)$. In the numeric phase, it constructs the resulting row as a minimum range sequence. Furthermore, in order to avoid scanning all words for set bits, the absolute minimum ($\jmath_{\min}$) and maximum ($\jmath_{\max}$) word indices are tracked and the final scan is limited to $[\jmath_{\min}, \jmath_{\max}]$. Since each row in **B** is sorted, $\jmath_{\min}$ and $\jmath_{\max}$ are updated once per row. The key advantage of the `gather` operation is that it relies solely on trivial bitwise functions, which – similar to `scatter` – exploit bit-level parallelism to convert clusters of set bits into their corresponding start and end indices. Algorithm 4 in Section C demonstrates the pseudocodes of the `gather` operations in the symbolic and numeric phrase. Additional information regarding these operations can be found in Section C.1.

The bitvector is designed to perform well in scenarios where non-zeros in row $C(i,:)$ are localized, which reflects a practically common scenario. To this end, two design decisions have been made. Firstly, the size of the bitvector is bounded by the bandwidth observed in $C(i,:)$ which is calculated before the symbolic phase. Secondly, the bitvector is partitioned into two levels $\mathcal{L}_0$ and $\mathcal{L}_1$ in order to minimize the memory latency by utilizing the locality of non-zero values in $C(i,:)$. The first level ($\mathcal{L}_0$) is allocated on the fast (low-latency) memory if there is hardware support (e.g. shared memory in GPUs). The rest of the bit fields is mapped to the second level $\mathcal{L}_1$ and allocated on the larger memory but with higher latency (e.g. global memory on GPU). On hardware where software-managed memory hierarchy is not available (like CPUs), this model promotes cache-friendly memory accesses.

The rationale behind this design is that the first level $\mathcal{L}_0$ spans the range of bits that are more likely to be accessed during row accumulations. If all non-zero values in a row are bound to $\mathcal{L}_0$ index range, no words in $\mathcal{L}_1$ are fetched or modified. Therefore, all `scatter` and `gather` operations mentioned earlier are performed on words in the fastest memory. The relative position of the first level within a row is specified by the index of its center bit p, referred as *pivot bit*. For example, the center of the band in each row of C is a viable option for p. In this work, we chose $p = i$ to set the $\mathcal{L}_0$ region around the diagonal as node i is more likely to be adjacent to nodes with ranks close to i in adjacency matrices.

The arrangement of bits in two levels necessitates calculating internal (local) indices within levels from column (global) indices. Local indices in each level are calculated from corresponding global indices relative to p and using modular arithmetic (see Figure 4b).

4.4 Range Sparse Boolean Matrix Addition (rSpAdd)

In this section, we shift our focus to calculating matrix-matrix addition in rCRS format which is required for computing the distance index. We propose the *Range Boolean Sparse Add* (or *rSpAdd*) algorithm which computes matrix $\mathbf{C} = \mathbf{A} \vee \mathbf{B}$ where all matrices are in rCRS format and $\vee$ is Boolean matrix addition defined in Definition 2.

Considering that non-zero values in each row of rCRS matrices are represented by ranges, each with two integers, and these integers are sorted and disjoint, row $C(i,:)$ is equivalent to the sorted combination of ranges in both rows $A(i,:)$ and $B(i,:)$, where overlapping ranges are collapsed into one. For each row in $\mathbf{C}$, this is achieved by using two pointers, each initially pointing to the first element of the respective rows. The algorithm peeks at the pairs in $A(i,:)$ and $B(i,:)$ indicated by the pointers. If two ranges are disjoint, it inserts the one with smaller bounds in $C(i,:)$ and increments the pointer indicating the inserted pair by two. If two ranges overlap, they will be merged by inserting the minimum of lower bounds and the maximum of upper bounds of the two pointed elements into $C(i,:)$ in that order. Then, both pointers are incremented to indicate the next pairs. In case either pointer reaches the end of the row, the remaining elements from the other row are directly appended to $C(i,:)$.

The time complexity of this algorithm is bound by compressed representation of input matrices in rCRS format. Since the ranges in rCRS are disjoint and sorted, merging two rows $A(i,:)$ and $B(i,:)$ can be done in $\Theta(|A(i,:)| + |B(i,:)|)$; in which $|A(i,:)|$ and $|B(i,:)|$ are the sizes of $A(i,:)$ and $B(i,:)$ in rCRS format, respectively. rSpAdd not only requires smaller working space compared to SpAdd but is arguably faster due to compressed representation of the matrices.

4.5 Distance Index

As stated in Section 3.1, the distance index constructed by Equation (1) is a block-diagonal matrix. For this reason, DiVerG builds the distance index incrementally for each block; i.e. computing the distance matrix for each component of the sequence graph individually. The adjacency matrix for each component can be constructed in rCRS format one row at a time. Therefore, it is not required to store the whole matrix in sorted CRS format to be able to convert it to rCRS. Once matrices $\mathbf{A}$ and $\mathbf{I}$ are in rCRS format, $\mathcal{T}$ can be computed using rSpAdd and rSpGEMM. The powers of $\mathbf{A}$ and $\mathbf{A} \vee \mathbf{I}$ are computed using "exponentiation by squaring" [28]. Ultimately, matrix $\mathcal{T}$ is further compressed by encoding both the column indices and row map arrays using Elias-δ encoding [17].

DiVerG provides an exact solution to the DVP through its compressed format for storing both final and intermediate matrices when computing the distance index $\mathcal{T}$. The compression ratio directly impacts the efficiency of the matrix multiplication operations, with higher compression resulting in faster construction. Since compression efficiency depends on the topology of the sequence graph and the structure of its adjacency matrix, establishing tight theoretical bounds on space and time complexities is challenging. However, we can derive meaningful lower bounds to facilitate comparison with previous approaches.

It can be demonstrated that for a sequence graph $G = (V, E)$, indexing cost is lower bounded by that of the chain graph $G_c = (V', E')$ representing the longest path in G [27]. This analysis applies to our rSpGEMM algorithm as well. For a chain graph, the index $\mathcal{T}$ has a particularly simple structure (Figure 3): in each row i, all columns from d_1 to d_2 is 1. This structure offers two advantages in the rCRS format:

- **Space complexity:** Each row requires only two integers regardless of the distance range, resulting in $\Omega(|V|)$ space complexity – independent of distance parameters d_1 and d_2, unlike PairG's $\Omega(|V| \cdot d)$ where $d = d_2 - d_1$. Note that Elias encoding of the row map and column indices arrays compresses the final index even further in practice.
- **Time complexity:** While rSpGEMM performs the same operations as SpGEMM, bit-parallelism reduces the computation time by a factor of the word size W. Jain et al. [27] showed that computing $\mathcal{T}$ for G using SpGEMM requires $\Omega\left(|V'|\left(d^2 + \log d_1\right)\right)$; therefore, DiVerG's rSpGEMM requires $\Omega\left(|V'|\left(d^2/W + \log d_1\right)\right)$. Note that, this improvement is particularly significant when computing $(\mathbf{A} \vee \mathbf{I})^{d_2 - d_1}$, where multiple non-zero values per row enable effective bit-parallelism.

Given that sequence graphs often closely resemble their linear structure, these bounds provide a reasonable approximation of actual performance while highlighting the fundamental advantages of our approach.

5 Experimental Results

5.1 Implementation

DiVerG is implemented in C++17. While we develop our theory using character graphs, our implementation handles general string-labelled sequence graphs without loss of generality. The matrix $\mathcal{T}$, in the final stage, is converted to *encoded rCRS* format (described in Section 4.5) whose size is reported as *index size* in Section 5.3. Our implementation relies on Kokkos [13] for performance portability across different architectures, and we specifically focused on two execution spaces: OpenMP (CPU), and CUDA (GPU).

We compared the performance of our algorithm with PairG which uses the `kkSpGEMM` meta-algorithm implemented in Kokkos [13] for computing sparse matrix multiplication. Kokkos offers several algorithms for computing SpGEMM. The `kkSpGEMM` algorithm attempts to choose the best algorithm based on the inputs. As the PairG implementation was in practice not runnable on GPU, we re-implemented the corresponding part of PairG to support GPU and used this implementation for comparison.

All CPU experiments were conducted on an instance on de.NBI cloud with a 28-core (one thread per core) Intel® Xeon® Processor running Ubuntu 22.4. Distance index construction on GPU were carried out on an instance with an NVIDIA A40 GPU running Ubuntu 22.4.

5.2 Datasets

DiVerG is assessed using seven different graphs. Four of these graphs were previously employed by PairG in their evaluation, while three additional graphs have been introduced to address significantly larger and more complex scenarios. Together, these graphs span over a spectrum of scales, complexities, and construction methods.

Three of these graphs are variation graphs of different regions of the human genome, constructed from small variants in the 1000 Genomes Project [1] based on GRCh37 reference assembly: the mitochondrial DNA (mtDNA) graph, the BRCA1 gene graph (BRC), and the killer cell immunoglobulin-like receptors (LRC_KIR) graph. One graph is a de Bruijn graph constructed using whole-genome sequences of 20 strains of B. *anthracis*. We added three new variation graphs to this ensemble: the MHC region graph constructed from alternate loci released with GRCh38 reference assembly [34], and the HPRC CHM13 graph [29] constructed by minigraph. Table 1 presents some statistics of the sequence graphs used in our evaluation.

Table 1 Statistics of the sequence graphs used for performance evaluation.

Graphs	Nodes[1)]	Edges
mtDNA (GRCh37–1KGP)	21,037	26,842
BRCA1 (GRCh37–1KGP)	83,267	85,422
LRC_KIR (GRCh37–1KGP)	1,108,768	1,154,049
MHC-minigraph (GRCh38–alt)	10,987,753	11,078,552
B. *anthracis* (20 strains)	11,237,088	11,253,454
HPRC CHM13 chr22	63,520,037	63,527,923
HPRC CHM13 chr1	261,344,589	261,360,990

1) Note that all graphs are *character graphs*.

5.3 Results

We evaluate the *index size* (`size`), *construction time* (`ctime`), and high water mark memory footprint during index construction (`mem`) for different distance constraints using the graphs explained in Section 5.2. Moreover, the *query time* (`qtime`) is measured for the distance index by averaging the time spend for querying 1M randomly sampled paired positions throughout each graph. Our method is evaluated using two separate sets of distance constraints.

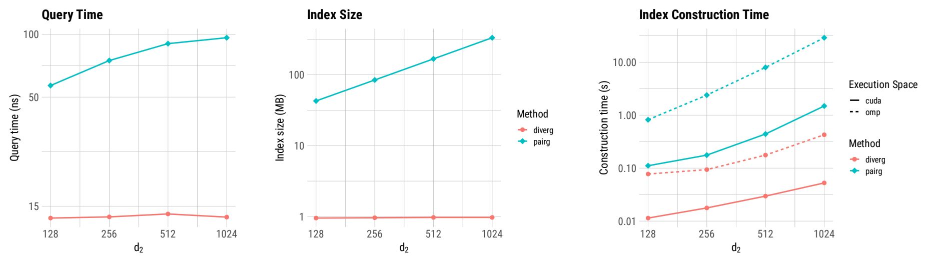
Performance Evaluation with Increasing Distance Constraints Intervals

In the first experiment, performance metrics are computed for two graphs, BRCA1 (small) and MHC (medium-sized), with distance constraints: $(0, 128)$, $(0, 256)$, $(0, 512)$, and $(0, 1024)$. This experiment reveals the behaviour of our algorithm as the distance range increases and specifically highlights the performance of rSpGEMM in computing the term $(\mathbf{A} \vee \mathbf{I})^{d_2-d_1}$ in Equation (1), which is a bottleneck in calculating $\mathcal{T}$. The performance measures are computed on both CPU (OpenMP) and GPU (CUDA).

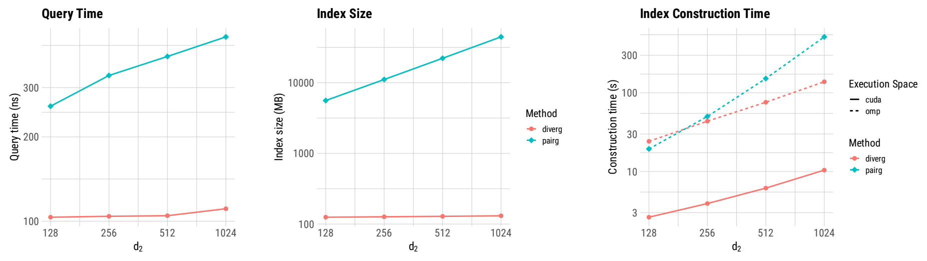
Figure 5a illustrates the performance measures and their comparison with PairG for the BRCA1 graph. As can be seen, DiVerG is about 4–7x faster in query time and 45–172x smaller in size in comparison to standard sparse matrix format and operations. Furthermore, the results show that DiVerG is approximately 9–28x faster than PairG in construction time on GPU and 10–68x on CPU. Moreover, the query time and index size in DiVerG do not change as the range of the distance constraints increases. This is because the number of non-zero values, i.e. ranges, in the rCRS format saturates quickly and stays constant.

Our evaluations for the MHC region graph conducted with the same distance intervals are illustrated in Figure 5b. Although the MHC graph is much larger than the BRCA1 graph, the trends in the results remain consistent. Specifically, DiVerG achieves about 2.5–4x speed-up in accessing the matrix; the index in the rCRS format is approximately 44–340x smaller compared to the classical CRS format; and the query time and index size do not change with an increase in the distance interval. As the standard SpGEMM in PairG fails to construct the index on GPU due to the graph size and limited memory available on the device, we only report the runtime on CPU for this graph.

As shown in Figure 5b, DiVerG is faster that PairG in indexing the graph on CPU, except for the distance interval [0, 128]. However, as the distance intervals grow larger, DiVerG becomes up to 3x faster in index construction. This speedup increases to 7–49.5x when DiVerG is executed on GPU: 7x for the distance constraint $(0, 128)$ and 49.5x for $([0, 1024)$.



(a)



(b)

Figure 5 Log-scale performance of DiVerG for BRCA1 (a) and MHC graphs (b).

Performance Evaluation with Offset Distance Constraints Intervals

In the second experiment, DiVerG is assessed for more realistic distance constraints $(0, 250)$, $(150, 450)$, and $(350, 650)$, resembling the inner distance model for paired-end reads with fragment sizes 300 bp, 500 bp, and 700 bp, respectively. The evaluation is conducted on all graphs described in Section 5.2, including chr1 and chr22 of the HPRC CHM13 graph to demonstrate the scalability of DiVerG to large genomes. All benchmarks are executed on an NVIDIA A40 GPU which has 48 GB of global memory. For certain datasets and parameter configurations, PairG failed to construct the index, either due to excessive memory requirements or because it did not complete within a 6-hour time frame. In those cases, we only report the performance of DiVerG, and the size of the PairG index which is computed via direct conversion from rCRS to CRS.

Table 2 shows the superior performance of DiVerG in all metrics, being about 170–420x smaller, and, in instances where PairG successfully completes, the indices are created 3–6x faster with considerably lower memory requirements than PairG for all datasets. The average number of column indices in each row in the rCRS format is about 2.1 for all variation graphs and all distance intervals. This number is approximately 4.8, 5.8, and 8.2 for the de Bruijn graph with distance intervals $(0, 250)$, $(150, 450)$, and $(350, 650)$, respectively, while the average number of non-zero values per row in the equivalent CRS format is close to 300, i.e. $d_2 - d_1$. This explains the high compression ratio of the DiVerG distance index. Additionally, query time shows a 3–5x speedup across all pangenome graphs. Our method can handle significantly larger graphs that are beyond the capabilities of PairG as it only requires a few tens of megabytes *auxiliary space* in memory per active thread.

6 Conclusion

In this work, we have proposed DiVerG, a compressed representation combined with fast algorithms for computing a distance index that exploits the structure of adjacency matrices of sequence graphs to significantly improve space and runtime efficiency. In this way, DiVerG

Table 2 Performance of DiVerG compared to PairG on an NVIDIA A40 GPU in terms of index size (`size`), construction time (`ctime`), high water mark memory footprint (`mem`), and query time (`qtime`), along with the number of non-zero values in the final index (`nnz`).

			DiVerG				PairG			
	d_1–d_2	`nnz`	`size`	`ctime`	`mem`	`qtime`	`size`	`ctime`	`mem`	`qtime`
mtDNA	0–250	7M	**287**KB	**37**ms	**1**MB	**12**ns	52MB	215ms	2.17GB	36ns
	150–450	8M	**245**KB	**39**ms	**1**MB	**11**ns	60MB	240ms	2.18GB	37ns
	350–650	8M	**243**KB	**39**ms	**1**MB	**11**ns	58MB	239ms	2.18GB	36ns
BRCA1	0–250	21M	**1**MB	**35**ms	**4**MB	**13**ns	164MB	113ms	542MB	69ns
	150–450	26M	**1**MB	**41**ms	**5**MB	**13**ns	197MB	173ms	500MB	75ns
	350–650	26M	**1**MB	**44**ms	**5**MB	**13**ns	197MB	176ms	500MB	75ns
LRC	0–250	290M	**13**MB	**320**ms	**52**MB	**25**ns	2.2GB	1.1s	4.5GB	113ns
	150–450	349M	**13**MB	**374**ms	**70**MB	**25**ns	2.7GB	1.9s	5.4GB	120ns
	350–650	350M	**13**MB	**394**ms	**73**MB	**22**ns	2.7GB	1.9s	5.5GB	117ns
MHC	0–250	3B	**125**MB	**8**s	**41**GB	**53**ns	21GB	11s	44GB	164ns
	150–450	3B	**126**MB	**11**s	**41**GB	**57**ns	26GB	–	–	–
	350–650	3B	**127**MB	**13**s	**41**GB	**55**ns	26GB	–	–	–
B.Anth.	0–250	5B	**178**MB	**4**m**22**s	**43**GB	**59**ns	34GB	–	–	–
	150–450	9B	**221**MB	**11**m**18**s	**43**GB	**59**ns	66GB	–	–	–
	350–650	14B	**254**MB	**23**m**49**s	**43**GB	**60**ns	104GB	–	–	–
chr22	0–250	16B	**723**MB	**37**s	**34**GB	**60**ns	121GB	–	–	–
	150–450	20B	**737**MB	**52**s	**33**GB	**57**ns	149GB	–	–	–
	350–650	21B	**743**MB	**58**s	**34**GB	**57**ns	154GB	–	–	–
chr1	0–250	66B	**2.96**GB	**14**m**18**s	**43**GB	**63**ns	495GB	–	–	–
	150–450	80B	**3**GB	**45**m**35**s	**43**GB	**62**ns	601GB	–	–	–
	350–650	82B	**3**GB	**1**h**04**m	**43**GB	**62**ns	612GB	–	–	–

provides a fast solution for the prominent DVP in paired-end read mapping to pangenome graphs. Our extensive experiments showed that DiVerG facilitates the computation of distance indexes, making it possible to solve the DVP when working with large graphs. We have demonstrated how to optimize algorithms with respect to hardware architecture considerations, which has been crucial for processing graphs that capture genomes at the scale of various eukaryotic genomes. We developed DiVerG with a particular focus on aligning paired-end short-reads to graphs, recognizing that the DVP can be a computational bottleneck. In the future, we plan to employ DiVerG in the context of sequence-to-graph alignments.

References

1 A. Auton, G. R. Abecasis, D. M. Altshuler, R. M. Durbin, D. R. Bentley, A. Chakravarti, A. G. Clark, P. Donnelly, E. E. Eichler, P. Flicek, S. B. Gabriel, R. A. Gibbs, E. D. Green, M. E. Hurles, B. M. Knoppers, J. O. Korbel, E. S. Lander, C. Lee, H. Lehrach, E. R. Mardis, G. T. Marth, G. A. McVean, D. A. Nickerson, J. P. Schmidt, S. T. Sherry, J. Wang, R. K. Wilson, E. Boerwinkle, H. Doddapaneni, Y. Han, V. Korchina, C. Kovar, S. Lee, D. Muzny, J. G. Reid, Y. Zhu, Y. Chang, Q. Feng, X. Fang, X. Guo, M. Jian, H. Jiang, X. Jin, T. Lan, G. Li, J. Li, Y. Li, S. Liu, X. Liu, Y. Lu, X. Ma, M. Tang, B. Wang, G. Wang, H. Wu, R. Wu, X. Xu, Y. Yin, D. Zhang, W. Zhang, J. Zhao, M. Zhao, X. Zheng, N. Gupta, N. Gharani,

L. H. Toji, N. P. Gerry, A. M. Resch, J. Barker, L. Clarke, L. Gil, S. E. Hunt, G. Kelman, E. Kulesha, R. Leinonen, W. M. McLaren, R. Radhakrishnan, A. Roa, D. Smirnov, R. E. Smith, I. Streeter, A. Thormann, I. Toneva, B. Vaughan, X. Zheng-Bradley, R. Grocock, S. Humphray, T. James, Z. Kingsbury, R. Sudbrak, M. W. Albrecht, V. S. Amstislavskiy, T. A. Borodina, M. Lienhard, F. Mertes, M. Sultan, B. Timmermann, M. L. Yaspo, L. Fulton, R. Fulton, V. Ananiev, Z. Belaia, D. Beloslyudtsev, N. Bouk, C. Chen, D. Church, R. Cohen, C. Cook, J. Garner, T. Hefferon, M. Kimelman, C. Liu, J. Lopez, P. Meric, C. O'Sullivan, Y. Ostapchuk, L. Phan, S. Ponomarov, V. Schneider, E. Shekhtman, K. Sirotkin, D. Slotta, H. Zhang, S. Balasubramaniam, J. Burton, P. Danecek, T. M. Keane, A. Kolb-Kokocinski, S. McCarthy, J. Stalker, M. Quail, C. J. Davies, J. Gollub, T. Webster, B. Wong, Y. Zhan, C. L. Campbell, Y. Kong, A. Marcketta, F. Yu, L. Antunes, M. Bainbridge, A. Sabo, Z. Huang, L. J. Coin, L. Fang, Q. Li, Z. Li, H. Lin, B. Liu, R. Luo, H. Shao, Y. Xie, C. Ye, C. Yu, F. Zhang, H. Zheng, H. Zhu, C. Alkan, E. Dal, F. Kahveci, E. P. Garrison, D. Kural, W. P. Lee, W. F. Leong, M. Stromberg, A. N. Ward, J. Wu, M. Zhang, M. J. Daly, M. A. DePristo, R. E. Handsaker, E. Banks, G. Bhatia, Angel G. Del, G. Genovese, H. Li, S. Kashin, S. A. McCarroll, J. C. Nemesh, R. E. Poplin, S. C. Yoon, J. Lihm, V. Makarov, S. Gottipati, A. Keinan, J. L. Rodriguez-Flores, T. Rausch, M. H. Fritz, A. M. Stütz, K. Beal, A. Datta, J. Herrero, G. R. Ritchie, D. Zerbino, P. C. Sabeti, I. Shlyakhter, S. F. Schaffner, J. Vitti, D. N. Cooper, E. V. Ball, P. D. Stenson, B. Barnes, M. Bauer, R. K. Cheetham, A. Cox, M. Eberle, S. Kahn, L. Murray, J. Peden, R. Shaw, E. E. Kenny, M. A. Batzer, M. K. Konkel, J. A. Walker, D. G. MacArthur, M. Lek, R. Herwig, L. Ding, D. C. Koboldt, D. Larson, K. Ye, S. Gravel, A. Swaroop, E. Chew, T. Lappalainen, Y. Erlich, M. Gymrek, T. F. Willems, J. T. Simpson, M. D. Shriver, J. A. Rosenfeld, C. D. Bustamante, S. B. Montgomery, La Vega F. M. De, J. K. Byrnes, A. W. Carroll, M. K. DeGorter, P. Lacroute, B. K. Maples, A. R. Martin, A. Moreno-Estrada, S. S. Shringarpure, F. Zakharia, E. Halperin, Y. Baran, E. Cerveira, J. Hwang, A. Malhotra, D. Plewczynski, K. Radew, M. Romanovitch, C. Zhang, F. C. Hyland, D. W. Craig, A. Christoforides, N. Homer, T. Izatt, A. A. Kurdoglu, S. A. Sinari, K. Squire, C. Xiao, J. Sebat, D. Antaki, M. Gujral, A. Noor, E. G. Burchard, R. D. Hernandez, C. R. Gignoux, D. Haussler, S. J. Katzman, W. J. Kent, B. Howie, A. Ruiz-Linares, E. T. Dermitzakis, S. E. Devine, H. M. Kang, J. M. Kidd, T. Blackwell, S. Caron, W. Chen, S. Emery, L. Fritsche, C. Fuchsberger, G. Jun, B. Li, R. Lyons, C. Scheller, C. Sidore, S. Song, E. Sliwerska, D. Taliun, A. Tan, R. Welch, M. K. Wing, X. Zhan, P. Awadalla, A. Hodgkinson, X. Shi, A. Quitadamo, G. Lunter, J. L. Marchini, S. Myers, C. Churchhouse, O. Delaneau, A. Gupta-Hinch, W. Kretzschmar, Z. Iqbal, I. Mathieson, A. Menelaou, A. Rimmer, D. K. Xifara, T. K. Oleksyk, Y. Fu, M. Xiong, L. Jorde, D. Witherspoon, J. Xing, B. L. Browning, S. R. Browning, F. Hormozdiari, P. H. Sudmant, E. Khurana, C. Tyler-Smith, C. A. Albers, Q. Ayub, Y. Chen, V. Colonna, L. Jostins, K. Walter, Y. Xue, M. B. Gerstein, A. Abyzov, S. Balasubramanian, J. Chen, D. Clarke, A. O. Harmanci, M. Jin, D. Lee, J. Liu, X. J. Mu, J. Zhang, Y. Zhang, C. Hartl, K. Shakir, J. Degenhardt, S. Meiers, B. Raeder, F. P. Casale, O. Stegle, E. W. Lameijer, I. Hall, V. Bafna, J. Michaelson, E. J. Gardner, R. E. Mills, G. Dayama, K. Chen, X. Fan, Z. Chong, T. Chen, M. J. Chaisson, J. Huddleston, M. Malig, B. J. Nelson, N. F. Parrish, B. Blackburne, S. J. Lindsay, Z. Ning, H. Lam, C. Sisu, D. Challis, U. S. Evani, J. Lu, U. Nagaswamy, J. Yu, W. Li, Kang H. Min, L. Habegger, H. Yu, F. Cunningham, I. Dunham, K. Lage, J. B. Jespersen, H. Horn, D. Kim, R. Desalle, A. Narechania, M. A. Sayres, F. L. Mendez, G. D. Poznik, P. A. Underhill, L. Coin, D. Mittelman, R. Banerjee, M. Cerezo, T. W. Fitzgerald, S. Louzada, A. Massaia, F. Yang, D. Kalra, W. Hale, X. Dan, K. C. Barnes, C. Beiswanger, H. Cai, H. Cao, B. Henn, D. Jones, J. S. Kaye, A. Kent, A. Kerasidou, R. Mathias, P. N. Ossorio, M. Parker, C. N. Rotimi, C. D. Royal, K. Sandoval, Y. Su, Z. Tian, S. Tishkoff, M. Via, Y. Wang, H. Yang, L. Yang, J. Zhu, W. Bodmer, G. Bedoya, Z. Cai, Y. Gao, J. Chu, L. Peltonen, A. Garcia-Montero, A. Orfao, J. Dutil, J. C. Martinez-Cruzado, R. A. Mathias, A. Hennis, H. Watson, C. McKenzie, F. Qadri, R. LaRocque, X. Deng, D. Asogun, O. Folarin, C. Happi, O. Omoniwa, M. Stremlau, R. Tariyal, M. Jallow, F. S. Joof,

T. Corrah, K. Rockett, D. Kwiatkowski, J. Kooner, T. T. Hiên, S. J. Dunstan, N. T. Hang, R. Fonnie, R. Garry, L. Kanneh, L. Moses, J. Schieffelin, D. S. Grant, C. Gallo, G. Poletti, D. Saleheen, A. Rasheed, L. D. Brooks, A. L. Felsenfeld, J. E. McEwen, Y. Vaydylevich, A. Duncanson, M. Dunn, and J. A. Schloss. A global reference for human genetic variation. *Nature*, 526(7571):68–74, October 2015. doi:10.1038/nature15393.

2 Débora Y. C. Brandt, Vitor R. C. Aguiar, Bárbara D. Bitarello, Kelly Nunes, Jérôme Goudet, and Diogo Meyer. Mapping Bias Overestimates Reference Allele Frequencies at the HLA Genes in the 1000 Genomes Project Phase I Data. *G3: Genes, Genomes, Genetics*, 5(5):931–941, March 2015. doi:10.1534/g3.114.015784.

3 Aydin Buluç and John R. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, January 2012. doi:10.1137/110848244.

4 Aydın Buluç, John Gilbert, and Viral B. Shah. *Implementing Sparse Matrices for Graph Algorithms*, pages 287–313. Society for Industrial and Applied Mathematics (SIAM), 2011. doi:10.1137/1.9780898719918.ch13.

5 Stefan Canzar and Steven L. Salzberg. Short Read Mapping: An Algorithmic Tour. *Proceedings of the IEEE*, 105(3):436–458, March 2017. doi:10.1109/jproc.2015.2455551.

6 Mark J. P. Chaisson, Ashley D. Sanders, Xuefang Zhao, Ankit Malhotra, David Porubsky, Tobias Rausch, Eugene J. Gardner, Oscar L. Rodriguez, Li Guo, Ryan L. Collins, Xian Fan, Jia Wen, Robert E. Handsaker, Susan Fairley, Zev N. Kronenberg, Xiangmeng Kong, Fereydoun Hormozdiari, Dillon Lee, Aaron M. Wenger, Alex R. Hastie, Danny Antaki, Thomas Anantharaman, Peter A. Audano, Harrison Brand, Stuart Cantsilieris, Han Cao, Eliza Cerveira, Chong Chen, Xintong Chen, Chen-Shan Chin, Zechen Chong, Nelson T. Chuang, Christine C. Lambert, Deanna M. Church, Laura Clarke, Andrew Farrell, Joey Flores, Timur Galeev, David U. Gorkin, Madhusudan Gujral, Victor Guryev, William Haynes Heaton, Jonas Korlach, Sushant Kumar, Jee Young Kwon, Ernest T. Lam, Jong Eun Lee, Joyce Lee, Wan-Ping Lee, Sau Peng Lee, Shantao Li, Patrick Marks, Karine Viaud-Martinez, Sascha Meiers, Katherine M. Munson, Fabio C. P. Navarro, Bradley J. Nelson, Conor Nodzak, Amina Noor, Sofia Kyriazopoulou-Panagiotopoulou, Andy W. C. Pang, Yunjiang Qiu, Gabriel Rosanio, Mallory Ryan, Adrian Stütz, Diana C. J. Spierings, Alistair Ward, AnneMarie E. Welch, Ming Xiao, Wei Xu, Chengsheng Zhang, Qihui Zhu, Xiangqun Zheng-Bradley, Ernesto Lowy, Sergei Yakneen, Steven McCarroll, Goo Jun, Li Ding, Chong Lek Koh, Bing Ren, Paul Flicek, Ken Chen, Mark B. Gerstein, Pui-Yan Kwok, Peter M. Lansdorp, Gabor T. Marth, Jonathan Sebat, Xinghua Shi, Ali Bashir, Kai Ye, Scott E. Devine, Michael E. Talkowski, Ryan E. Mills, Tobias Marschall, Jan O. Korbel, Evan E. Eichler, and Charles Lee. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nature Communications*, 10(1), April 2019. doi:10.1038/s41467-018-08148-z.

7 Xian Chang, Jordan Eizenga, Adam M. Novak, Jouni Sirén, and Benedict Paten. Distance Indexing and Seed Clustering in Sequence Graphs. *Bioinformatics*, 36(Supplement 1):i146–i153, July 2020. doi:10.1093/bioinformatics/btaa446.

8 K. Y. Cheng. Minimizing the bandwidth of sparse symmetric matrices. *Computing*, 11(2):103–110, 1973. doi:10.1007/BF02252900.

9 Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, January 2018. doi:10.1093/bib/bbw089.

10 E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. Association for Computing Machinery. doi:10.1145/800195.805928.

11 Elizabeth Cuthill. *Several Strategies for Reducing the Bandwidth of Matrices*, pages 157–166. Springer US, Boston, MA, 1972. doi:10.1007/978-1-4615-8675-3_14.

12 Jacob F. Degner, John C. Marioni, Athma A. Pai, Joseph K. Pickrell, Everlyne Nkadori, Yoav Gilad, and Jonathan K. Pritchard. Effect of read-mapping biases on detecting allele-specific expression from RNA-sequencing data. *Bioinformatics*, 25(24):3207–3212, October 2009. doi:10.1093/bioinformatics/btp579.

13 Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Multi-threaded Sparse Matrix-Matrix Multiplication for Many-Core and GPU Architectures, 2018. `doi:10.48550/arXiv.1801.03065`.

14 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. `doi:10.1007/bf01386390`.

15 Ran Duan, Jiayi Mao, Xiao Mao, Xinkai Shu, and Longhui Yin. Breaking the Sorting Barrier for Directed Single-Source Shortest Paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, STOC '25, pages 36–44, New York, NY, USA, 2025. Association for Computing Machinery. `doi:10.1145/3717823.3718179`.

16 Jordan M. Eizenga, Adam M. Novak, Jonas A. Sibbesen, Simon Heumos, Ali Ghaffaari, Glenn Hickey, Xian Chang, Josiah D. Seaman, Robin Rounthwaite, Jana Ebler, Mikko Rautiainen, Shilpa Garg, Benedict Paten, Tobias Marschall, Jouni Sirén, and Erik Garrison. Pangenome Graphs. *Annual Review of Genomics and Human Genetics*, 21(1), May 2020. `doi:10.1146/annurev-genom-120219-080406`.

17 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. `doi:10.1109/tit.1975.1055349`.

18 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. `doi:10.1145/28869.28874`.

19 Giorgio Gallo and Stefano Pallottino. Shortest path algorithms. *Annals of Operations Research*, 13(1):1–79, December 1988. `doi:10.1007/bf02288320`.

20 Erik Garrison, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F. Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, August 2018. `doi:10.1038/nbt.4227`.

21 Ali Ghaffaari. cartoonist/diverg. Software, version 1.0., EU-Grant-956229, swhId: `swh:1:rel:362b7551950ee369c01ada82d05b6c8345a3048d` (visited on 2025-06-27). URL: `https://github.com/cartoonist/diverg`, `doi:10.4230/artifacts.23650`.

22 Cristian Groza, Tony Kwan, Nicole Soranzo, Tomi Pastinen, and Guillaume Bourque. Personalized and graph genomes reveal missing signal in epigenomic data. *Genome Biology*, 21(1):124, May 2020. `doi:10.1186/s13059-020-02038-8`.

23 Fred G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, September 1978. `doi:10.1145/355791.355796`.

24 Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968. `doi:10.1109/tssc.1968.300136`.

25 J. Hein. A new method that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when the phylogeny is given. *Molecular Biology and Evolution*, 6(6):649–668, November 1989. `doi:10.1093/oxfordjournals.molbev.a040577`.

26 Moritz Hilger, Ekkehard Köhler, Rolf Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 41–72. DIMACS/AMS, July 2006. `doi:10.1090/dimacs/074/03`.

27 Chirag Jain, Haowen Zhang, Alexander Dilthey, and Srinivas Aluru. Validating Paired-End Read Alignments in Sequence Graphs. In Katharina T. Huber and Dan Gusfield, editors, *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.WABI.2019.17`.

28 Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2, chapter 4, pages 194–477. Addison-Wesley Longman Publishing Co., Inc., USA, third edition, 1997.

29 Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K. Lucas, Jean Monlong, Haley J. Abel, Silvia Buonaiuto, Xian H. Chang, Haoyu Cheng, Justin Chu, Vincenza Colonna, Jordan M. Eizenga, Xiaowen Feng, Christian Fischer, Robert S. Fulton, Shilpa Garg, Cristian Groza, Andrea Guarracino, William T. Harvey, Simon Heumos, Kerstin Howe, Miten Jain, Tsung-Yu Lu, Charles Markello, Fergal J. Martin, Matthew W. Mitchell, Katherine M. Munson, Moses Njagi Mwaniki, Adam M. Novak, Hugh E. Olsen, Trevor Pesout, David Porubsky, Pjotr Prins, Jonas A. Sibbesen, Jouni Sirén, Chad Tomlinson, Flavia Villani, Mitchell R. Vollger, Lucinda L. Antonacci-Fulton, Gunjan Baid, Carl A. Baker, Anastasiya Belyaeva, Konstantinos Billis, Andrew Carroll, Pi-Chuan Chang, Sarah Cody, Daniel E. Cook, Robert M. Cook-Deegan, Omar E. Cornejo, Mark Diekhans, Peter Ebert, Susan Fairley, Olivier Fedrigo, Adam L. Felsenfeld, Giulio Formenti, Adam Frankish, Yan Gao, Nanibaa' A. Garrison, Carlos Garcia Giron, Richard E. Green, Leanne Haggerty, Kendra Hoekzema, Thibaut Hourlier, Hanlee P. Ji, Eimear E. Kenny, Barbara A. Koenig, Alexey Kolesnikov, Jan O. Korbel, Jennifer Kordosky, Sergey Koren, HoJoon Lee, Alexandra P. Lewis, Hugo Magalhães, Santiago Marco-Sola, Pierre Marijon, Ann McCartney, Jennifer McDaniel, Jacquelyn Mountcastle, Maria Nattestad, Sergey Nurk, Nathan D. Olson, Alice B. Popejoy, Daniela Puiu, Mikko Rautiainen, Allison A. Regier, Arang Rhie, Samuel Sacco, Ashley D. Sanders, Valerie A. Schneider, Baergen I. Schultz, Kishwar Shafin, Michael W. Smith, Heidi J. Sofia, Ahmad N. Abou Tayoun, Françoise Thibaud-Nissen, Francesca Floriana Tricomi, Justin Wagner, Brian Walenz, Jonathan M. D. Wood, Aleksey V. Zimin, Guillaume Bourque, Mark J. P. Chaisson, Paul Flicek, Adam M. Phillippy, Justin M. Zook, Evan E. Eichler, David Haussler, Ting Wang, Erich D. Jarvis, Karen H. Miga, Erik Garrison, Tobias Marschall, Ira M. Hall, Heng Li, and Benedict Paten. A draft human pangenome reference. *Nature*, 617(7960):312–324, May 2023. doi:10.1038/s41586-023-05896-x.

30 Matti Nykänen and Esko Ukkonen. The Exact Path Length Problem. *Journal of Algorithms*, 42(1):41–53, January 2002. doi:10.1006/jagm.2001.1201.

31 Ch. H. Papadimitriou. The NP-Completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, September 1976. doi:10.1007/bf02280884.

32 Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G. Pudov, Vadim O. Pirogov, and Pradeep Dubey. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, volume 9137, pages 48–57, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-20119-1_4.

33 Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(1):253, September 2020. doi:10.1186/s13059-020-02157-2.

34 Valerie A. Schneider, Tina Graves-Lindsay, Kerstin Howe, Nathan Bouk, Hsiu-Chuan Chen, Paul A. Kitts, Terence D. Murphy, Kim D. Pruitt, Françoise Thibaud-Nissen, Derek Albracht, Robert S. Fulton, Milinn Kremitzki, Vincent Magrini, Chris Markovic, Sean McGrath, Karyn Meltz Steinberg, Kate Auger, William Chow, Joanna Collins, Glenn Harden, Timothy Hubbard, Sarah Pelan, Jared T. Simpson, Glen Threadgold, James Torrance, Jonathan M. Wood, Laura Clarke, Sergey Koren, Matthew Boitano, Paul Peluso, Heng Li, Chen-Shan Chin, Adam M. Phillippy, Richard Durbin, Richard K. Wilson, Paul Flicek, Evan E. Eichler, and Deanna M. Church. Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. *Genome Research*, 27(5):849–864, April 2017. doi:10.1101/gr.213611.116.

35 Alexander Schrijver. *Combinatorial optimization.* Number 24 in Algorithms and combinatorics. Springer, Berlin, 2003.

36 Ting Wang, Lucinda Antonacci-Fulton, Kerstin Howe, Heather A. Lawson, Julian K. Lucas, Adam M. Phillippy, Alice B. Popejoy, Mobin Asri, Caryn Carson, Mark J. P. Chaisson, Xian Chang, Robert Cook-Deegan, Adam L. Felsenfeld, Robert S. Fulton, Erik P. Garrison, Nanibaa' A. Garrison, Tina A. Graves-Lindsay, Hanlee Ji, Eimear E. Kenny, Barbara A. Koenig, Daofeng Li, Tobias Marschall, Joshua F. McMichael, Adam M. Novak, Deepak Purushotham, Valerie A. Schneider, Baergen I. Schultz, Michael W. Smith, Heidi J. Sofia, Tsachy Weissman, Paul Flicek, Heng Li, Karen H. Miga, Benedict Paten, Erich D. Jarvis, Ira M. Hall, Evan E. Eichler, and David Haussler. The Human Pangenome Project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, April 2022. `doi:10.1038/s41586-022-04601-8`.

37 Xuefang Zhao, Ryan L. Collins, Wan-Ping Lee, Alexandra M. Weber, Yukyung Jun, Qihui Zhu, Ben Weisburd, Yongqing Huang, Peter A. Audano, Harold Wang, Mark Walker, Chelsea Lowther, Jack Fu, Mark B. Gerstein, Scott E. Devine, Tobias Marschall, Jan O. Korbel, Evan E. Eichler, Mark J. P. Chaisson, Charles Lee, Ryan E. Mills, Harrison Brand, and Michael E. Talkowski. Expectations and blind spots for structural variation detection from long-read assemblies and short-read genome sequencing technologies. *The American Journal of Human Genetics*, 108(5):919–928, May 2021. `doi:10.1016/j.ajhg.2021.03.014`.

38 Uri Zwick. Exact and Approximate Distances in Graphs — A Survey. In Friedhelm Meyer auf der Heide, editor, *Algorithms — ESA 2001*, pages 33–48, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. `doi:10.1007/3-540-44676-1_3`.

A Fragment Model in Paired-End Sequencing

The fragment model of a read library describes the probability distribution of the insert sizes. This model is typically denoted by a normal distribution, characterized by the empirical mean and variance of the insert sizes.

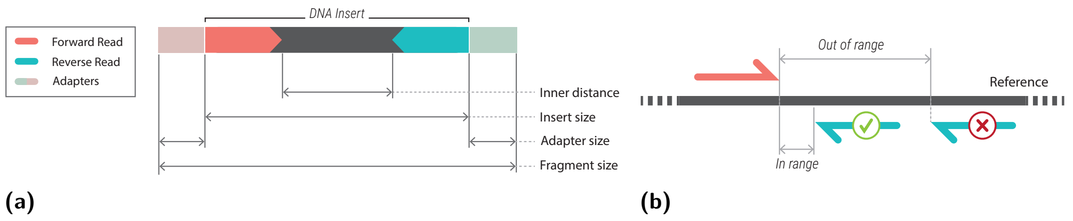


Figure 6 (a) A fragment in paired-end sequencing: a DNA insert with two adapters attached at each end enabling sequencing from both ends. (b) A schematic example of resolving alignment ambiguities using distance validation. The forward read has an unique alignment (in red) which can determine the correct alignment of the other end (blue).

In our study, the fragment model is represented by the lower and upper bounds of the expected insert size instead of a probability distribution. These bounds are still probabilistic, as they refer to reasonably chosen cut-offs on the tails of the insert size distribution, e.g. capturing 99.7% of the observed lengths or defined as $\mu \pm 3\sigma$ in which μ and σ are mean and variance in the fragment model.

There are two ways to define the distance between two paired reads. The *inner distance* between the ends of two paired reads which is determined by the size of unsequenced part of the DNA fragment. It can be calculated by subtracting the sum of the read lengths from the insert size (as shown in Figure 6a). The *outer distance* is defined by the distance between the starts of two paired reads which is equivalent to the *insert size*. Given the variability in both insert sizes and read lengths, and the possibility of overlapping reads in some datasets, using the outer distance can simplify the process of distance validation. Therefore, validating outer distance is more practical compared to the inner distance. Nevertheless, the method is neutral with respect to either definition of distance.

B Asymmetrical Range CRS

Asymmetrical Range CRS (aCRS) is defined similar to the rCRS format with slight tweaks to avoid redundancy in the column indices array when there is an isolated column index x, which is otherwise stored as subsequence $[x, x]$ in rCRS.

▶ **Definition 8** (Minimum Asymmetrical Range Sequence)**.** *Let A be a* sorted *sequence of distinct positive integers, and A_s a sequence constructed by replacing any disjoint subsequence of consecutive integers $S = s_f \cdots s_l$ in A by s_f and $-s_l$ if $|S| > 1$; otherwise S is replaced by the single element s_f. The sequence of minimum size constructed this way is defined as the* minimum asymmetrical range sequence *of A and is denoted by $\hat{\rho}(A)$.*

▶ **Example 9.** Let $A = [10, 11, 12, 13, 23, 29, 30]$ from previous example. The minimum asymmetrical range sequence of A is defined as $\hat{\rho}(A) = [10, -13, 23, 29, -30]$.

▶ **Lemma 10.** *The minimum asymmetrical range sequence of a non-empty set of integers A can be constructed from the sorted array representing A in linear time.*

Proof. It can be trivially constructed by starting from $\hat{\rho}(A) = \emptyset$ and iterating over elements in A in ascending order. ◀

▶ **Lemma 11.** *Given A as a non-empty set of positive integers, the size of the minimum asymmetrical range sequence of A is at most equal to the cardinality of A; i.e. $|\hat{\rho}(A)| \leq |A|$.*

Proof. This follows directly from the fact that any asymmetrical range sequence of A defines a partition of A, denoted as P, where each block in P represents a contiguous range of integers in A and contributes either one element to $\hat{\rho}(A)$ (if $|P| = 1$) or two elements (the endpoints if larger). In either case, each block contributes at most as many elements to $\hat{\rho}(A)$ as it contains from A. Summing over all blocks shows that $|\hat{\rho}(A)| \leq |A|$. ◀

▶ **Definition 12** (Asymmetrical Range Compressed Row Storage)**.** *For a sparse matrix $\mathbf{A}$ and its sorted CRS representation (C, R), Asymmetrical Range Compressed Row Storage or aCRS($\mathbf{A}$) is defined as two arrays (C'', R''); where*

- *$C'' = \hat{\rho}(C_0)\hat{\rho}(C_1) \cdots \hat{\rho}(C_{n-1})$, in which C_i is the ith partition in row decomposition of C;*
- *R'' is an array of length $n+1$ in which $R'(i)$ specifies the start index in C'' that corresponds to row i; i.e. the start index of $\hat{\rho}(C_i)$. The last value is defined as $R''(n) = |C'|$.*

▶ Remark 13. Since column indices are non-negative, negative values in aCRS distinctly indicate the upper bound of a range.

▶ **Theorem 14.** *aCRS can be constructed from sorted CRS in linear time with respect to the number of non-zero values.*

Proof. Directly followed by Lemma 10 and Definition 12. ◀

▶ **Theorem 15.** *For a sparse matrix $\mathbf{A}$, aCRS($\mathbf{A}$) always requires space that is equal to or smaller than that of CRS($\mathbf{A}$) as well as rCRS($\mathbf{A}$).*

Proof. This can be readily demonstrated using Lemma 11 and Definition 12. ◀

▶ Remark 16. The elements of the column indices array in aCRS, C'', are sorted in each partition of its row decomposition and the ranges indicated by pairs or isolated elements in C''_i are disjoint.

▶ **Theorem 17.** *Let $\hat{z}$ be the maximum size of the minimum asymmetrical range sequence representation over all rows of sparse matrix* $\mathbf{A}$*. Accessing an element in* $\mathbf{A}$ *represented by aCRS format takes* $O(\log(\hat{z}))$ *time in the worst case.*

Proof. Similar to rCRS, each row i in aCRS format is represented by a subsequence C_i'' of signed (paired or singleton) integers denoting a series of ranges. These elements encode intervals of column indices where non-zero entries occur, and their signs indicate range boundaries (see Remark 13). To query a_{ij}, we perform a binary search on the *absolute values* of C_i'' to find the largest index p such that $\mathbf{abs}(C_i''[p]) \leq j$. The value at position p identifies either the lower or upper bound of a range containing j, depending on its sign. By construction, every such range is disjoint and the sequence is sorted in terms of absolute values (Remark 16), ensuring that at most one range can contain j.

The identification step is constant-time, and since binary search is used over C_i'' of size at most $\hat{z}$, the total time per lookup is $O(\log \hat{z})$. ◀

C Bi-Level Banded Bitvector

C.1 `scatter` and `gather` Operations

In this section, we cover some details regarding the `scatter` and `gather` operations (Algorithm 3 and Algorithm 4).

The `gather` operation functions using four trivial bitwise functions: `cnt`, `map01`, `map10`, and `sel`. The `cnt` function simply counts the number of set bits in a word and is performed by a single machine instruction (`POPCNT`).

The `map01` and `map10` functions identify boundaries of consecutive set bits (runs of 1s) in a word through specific bit pattern transformations. The `map01` function detects `01` patterns, mapping them to `01` (i.e. itself) while mapping all other two-bit patterns (`00`, `10`, and `11`) to `00`. Since a `01` pattern occurs when a 0 is followed by a 1, this function effectively marks the starting position of each run of consecutive 1s.

Conversely, the `map10` function detects `10` patterns, mapping them to `01` and all other patterns to `00`. Since a `10` pattern occurs when a 1 is followed by a 0, this function marks the position immediately after the end of each run of consecutive 1s. Both `map01` and `map10` can be calculated using basic bitwise operations[1] in constant time independent of the word size. The border cases, when `01` or `10` occur at the boundaries between two words, are handled by peeking at the immediate bit on the left of the bit that is being processed. In other words, the most significant bit (MSB) of the previous word is appended to the current word before applying `map*` functions. This bit, which is called *carried* bit, is zero for the first word.

Finally, the `sel`(x, i) function gives the position of i-th set bit in word x relying on native machine instructions (e.g. `__fns` intrinsic in CUDA and `PDEP` and `TZCNT` instructions on CPUs). The $\textsc{shiftLeft}(w, l)$ and $\textsc{shiftRight}(w, l)$ function calls in Algorithm 3 and Algorithm 4 denote bitwise left-shift and right-shift operations of word w by l bits, respectively.

In the symbolic phase, `gather` counts the number of entries in rCRS with regards to non-zeros in row $C(i, :)$. This count is equivalent to twice the number of `01` occurrences in the bitvector and can be formulated as `cnt`(`map01`(w)) for all modified words w in the bitvector. In order to convert the set bit stretches to integer intervals in numeric phase, the first and the last bit of each stretch are marked by applying `map01` and `map10` functions on all words. Finally, calling `sel` on each set bits in (`map01`(w) OR `map10`(w)) gets the final minimum range sequence of the accumulated indices.

[1] `http://www-graphics.stanford.edu/~seander/bithacks.html`

Algorithm 3 Bi-level Banded Bitvector `scatter` operation.

```
Require: Bi-level bitvector $\mathcal{B}$, start and finish column indices $s$ and $f$, and word size $W$
 1: function SCATTER($\mathcal{B}$, $s$, $f$)
 2:     $i \leftarrow \lfloor s/W \rfloor$                    ▷ Word index of the bit at $s$ in $\mathcal{B}$
 3:     $j \leftarrow \lfloor f/W \rfloor$                    ▷ Word index of the bit at $f$ in $\mathcal{B}$
 4:     $o \leftarrow s \bmod W$                              ▷ Offset of $s$ within word $i$
 5:     $p \leftarrow f \bmod W$                              ▷ Offset of $f$ within word $j$
 6:     if $i = j$ then
 7:         mask $\leftarrow$ SHIFTLEFT(SHIFTLEFT$(1, p - o + 1) - 1, o$)   ▷ $mask = 0\ldots0\overbrace{1\ldots1}^{p \leftarrow o}0\ldots0$
 8:         $\mathcal{B}[i] \leftarrow \mathcal{B}[i]$ OR mask
 9:     else
10:         mask $\leftarrow$ SHIFTLEFT$(2^W - 1, o)$            ▷ $mask = \overbrace{1\ \ldots\ 11}^{W-1 \leftarrow o}0\ldots0$
11:         $\mathcal{B}[i] \leftarrow \mathcal{B}[i]$ OR mask
12:         for $k := i + 1 \rightarrow j - 1$ do               ▷ Set all bits of word $k \in [i + 1, j - 1]$
13:             $\mathcal{B}[k] = 2^W - 1$
14:         mask $\leftarrow$ SHIFTRIGHT$(2^W - 1, W - p - 1)$    ▷ $mask = 0\ldots0\overbrace{1\ldots1}^{p \leftarrow 0}$
15:         $\mathcal{B}[j] \leftarrow \mathcal{B}[j]$ OR mask
```

Algorithm 4 Bi-level Banded Bitvector `gather` operation.

```
Require: Word $w$ and carried bit $c$
 1: function MAP10($w$, $c$)
 2:     $x \leftarrow$ SHIFTLEFT$(w, 1)$ OR $c$               ▷ appending bit $c$ to $w$ and left-shift by 1
 3:     report $x$ AND NOT$(w)$
 4: function MAP01($w$, $c$)
 5:     $x \leftarrow$ SHIFTLEFT$(w, 1)$ OR $c$               ▷ appending bit $c$ to $w$ and left-shift by 1
 6:     report $(w$ XOR $x)$ AND $w$

Require: Column indices subarray $C'_r$, bi-level bitvector $\mathcal{B}$, minimum and maximum column
         indices $j_{min}$ and $j_{max}$ in the row, and word size $W$
 7: function GATHERSYMBOLIC($\mathcal{B}$, $j_{min}$, $j_{max}$)
 8:     $b \leftarrow \lfloor j_{min}/W \rfloor$              ▷ Word index of the bit at $j_{min}$ in $\mathcal{B}$
 9:     $e \leftarrow \lfloor j_{max}/W \rfloor$              ▷ Word index of the bit at $j_{max}$ in $\mathcal{B}$
10:     $z \leftarrow 0,\ c \leftarrow 0$
11:     for $i := b \rightarrow e$ do
12:         $z \leftarrow z + 2 *$ CNT(MAP01$(\mathcal{B}[i], c)$)       ▷ Count each runs of 1s twice
13:         $c \leftarrow$ RIGHTSHIFT$(\mathcal{B}[i], W - 1)$   ▷ $c$ is assigned to the most significant bit of $\mathcal{B}[i]$
14:     report $z$
15: function GATHERNUMERIC($C'_r$, $\mathcal{B}$, $j_{min}$, $j_{max}$)
16:     $b \leftarrow \lfloor j_{min}/W \rfloor$              ▷ Word index of the bit at $j_{min}$ in $\mathcal{B}$
17:     $e \leftarrow \lfloor j_{max}/W \rfloor$              ▷ Word index of the bit at $j_{max}$ in $\mathcal{B}$
18:     $k \leftarrow 0$
19:     for $i := b \rightarrow e$ do
20:         $w \leftarrow$ MAP01$(\mathcal{B}[i])$ OR MAP10$(\mathcal{B}[i])$   ▷ Identify boundaries of consecutive set bits
21:         for $j := 0 \rightarrow$ CNT$(w)$ do
22:             $C'_r[k] = i \cdot W +$ SEL$(w, j + 1)$
23:             $k \leftarrow k + 1$
```