# Communication-Efficient Jaccard similarity for High-Performance Distributed Genome Comparisons

Maciej Besta[1†], Raghavendra Kanakagiri[5†], Harun Mustafa[1,3,4], Mikhail Karasikov[1,3,4],
Gunnar Rätsch[1,3,4], Torsten Hoefler[1], Edgar Solomonik[2]
[1]*Department of Computer Science, ETH Zurich*
[2]*Department of Computer Science, University of Illinois at Urbana-Champaign*
[3]*University Hospital Zurich, Biomedical Informatics Research*
[4]*SIB Swiss Institute of Bioinformatics*
[5]*Department of Computer Science, Indian Institute of Technology Tirupati*
[†]Both authors contributed equally to this work.

***Abstract*—The Jaccard similarity index is an important measure of the overlap of two sets, widely used in machine learning, computational genomics, information retrieval, and many other areas. We design and implement SimilarityAtScale, the first communication-efficient distributed algorithm for computing the Jaccard similarity among pairs of large datasets. Our algorithm provides an efficient encoding of this problem into a multiplication of sparse matrices. Both the encoding and sparse matrix product are performed in a way that minimizes data movement in terms of communication and synchronization costs. We apply our algorithm to obtain similarity among all pairs of a set of large samples of genomes. This task is a key part of modern metagenomics analysis and an evergrowing need due to the increasing availability of high-throughput DNA sequencing data. The resulting scheme is the first to enable accurate Jaccard distance derivations for massive datasets, using large-scale distributed-memory systems. We package our routines in a tool, called GenomeAtScale, that combines the proposed algorithm with tools for processing input sequences. Our evaluation on real data illustrates that one can use GenomeAtScale to effectively employ tens of thousands of processors to reach new frontiers in large-scale genomic and metagenomic analysis. While GenomeAtScale can be used to foster DNA research, the more general underlying SimilarityAtScale algorithm may be used for high-performance distributed similarity computations in other data analytics application domains.**

***Index Terms*—Distributed Jaccard Distance, Distributed Jaccard similarity, Genome Sequence Distance, Metagenome Sequence Distance, High-Performance Genome Processing, k-Mers, Matrix-Matrix Multiplication, Cyclops Tensor Framework**

**Code and data:** https://github.com/cyclops-community/jaccard-ctf

## I. Introduction

The concept of *similarity* has gained much attention in different areas of data analytics [59]. A popular method of assessing a similarity of two entities is based on computing their Jaccard *similarity* index $J(A,B)$ [29], [38]. $J(A,B)$ is a statistic that assesses the overlap of two sets $A$ and $B$. It is defined as the ratio between the size of the intersection of $A$ and $B$ and the size of the union of $A$ and $B$: $J(A,B) = |A \cap B|/|A \cup B|$. A closely related notion, the Jaccard *distance* $d_J = 1 - J$, measures the *dissimilarity* of sets. The general nature and simplicity of the Jaccard similarity and distance has allowed for their wide use in numerous domains, for example computational genomics (comparing DNA sequencing data sets), machine learning (clustering, object recognition), information retrieval (plagiarism detection), and many others [60], [21], [44], [47], [49], [70], [58].

We focus on the application of the Jaccard similarity index and distance to genetic distances between high-throughput DNA sequencing samples, a problem of high importance for different areas of computational biology [50], [70], [18]. Genetic distances are frequently used to approximate the evolutionary distances between the species or populations represented in sequence sets in a computationally intractable manner [50], [42], [70], [18]. This enables or facilitates analyzing the evolutionary history of populations and species [42], the investigation of the biodiversity of a sample, as well as other applications [70], [43]. However, the enormous sizes of today's high-throughput sequencing datasets often make it infeasible to compute the exact values of $J$ or $d_J$ [57]. Recent works (such as Mash [42]) propose approximations, for example using the MinHash representation of $J(A,B)$, which is the primary locality-sensitive hashing (LSH) scheme used for genetic comparisons [37]. Yet, these approximations often lead to inaccurate approximations of $d_J$ for highly similar pairs of sequence sets, and tend to be ineffective for computation of a distance between highly dissimilar sets unless very large sketch sizes are used [42], as noted even by the Mash authors [42]. Thus, developing a *scalable*, *high-performance*, and *accurate* scheme for computing the Jaccard similarity index is an open problem of considerable relevance for genomics computations and numerous other applications in general data analytics. **R4**

Yet, there is little research on high-performance distributed derivations of either $J$ or $d_J$. Existing works target very small datasets (e.g., 16MB of raw input text [19]), only provide simulated evaluation for experimental hardware [36], focus on inefficient MapReduce solutions [16], [7], [64] that need asymptotically more communication due to using the allreduce collective communication pattern over reducers [27], are limited to a single server [22], [45], [48], use an approach based on deriving the Cartesian product, which requires quadratic space and is infeasible for large input datasets considered in **R4** **R1**

this work [1], [28], or do not target parallelism or distribution at all [8]. Most works target novel use cases of $J$ and $d_J$ [10], [12], [61], [33], or mathematical foundations of these measures [34], [41]. We attribute this to two factors. First, many domains (for example genomics) only recently discovered the usefulness of Jaccard measures [70]. Second, the rapid growth of the availability of high-throughput sequencing data and its increasing relevance to genetic analysis have meant that previous complex genetic analysis methods are falling out of favor due to their poor scaling properties [70]. *To the best of our knowledge, no work provides a scalable high-performance solution for computing $J$ or $d_J$.*

In this work, we design and implement **SimilarityAtScale**: the first algorithm for distributed, fast, and scalable derivation of the Jaccard Similarity index and Jaccard distance. We follow [24], devising an algorithm based on an algebraic formulation of Jaccard similarity matrix computation as sparse matrix multiplication. Our main contribution is a *communication-avoiding algorithm* for both preprocessing the sparse matrices as well as computing Jaccard similarity in parallel via sparse matrix multiplication. We use our algorithm as a core element of **GenomeAtScale**, a tool that we develop to facilitate high-performance genetic distance computation. GenomeAtScale enables *the first massively-parallel and provably accurate calculations of Jaccard distances between genomes*. By maintaining compatibility with standard bioinformatics data formats, we allow for GenomeAtScale to be seamlessly integrated into existing analysis pipelines. Our performance analysis illustrates the scalability of our solutions. In addition, we benchmark GenomeAtScale on both large (2,580 human RNASeq experiments) and massive (almost all public bacterial and viral whole-genome sequencing experiments) scales, and we make this data publicly available to foster high-performance distributed genomics research. Despite our focus on genomics data, the algebraic formulation and the implementation of our routines for deriving Jaccard measures are generic and can be directly used in other settings.

Specifically, our work makes the following contributions.

- We design **SimilarityAtScale**, the first communication-efficient distributed algorithm to compute the Jaccard similarity index and distance.
- We use our algorithm as a backend to **GenomeAtScale**, the first tool that enables fast, scalable, accurate, and large-scale derivations of Jaccard distances between high-throughput whole-genome sequencing samples.
- We ensure that SimilarityAtScale is generic and can be reused in any other related problem. We overview the relevance of Jaccard measures in data mining applications.
- We support our algorithm with a theoretical analysis of the communication costs and parallel scaling efficiency.
- We evaluate GenomeAtScale on real genomic datasets, showing that it enables large-scale genomic analysis. We scale our runs to up to 1024 compute nodes, which is the largest scale that we know of for genetic distance computations [42], [18].

*The whole implementation of GenomeAtScale, as well as the analysis outcomes for established real genome datasets, are publicly available in order to enable interpretability and reproducibility, and to facilitate its integration into current and future genomics analysis pipelines.*

## II. Jaccard Measures: Definitions and Importance

We start with defining the Jaccard measures and with discussing their applications in different domains. The most important symbols used in this work are listed in Table I. While we focus on high-performance computations of distances between genome sequences, our design and implementation are generic and applicable to any other use case.

| | |
|---|---|
| **General Jaccard measures:** | |
| $J(X,Y)$ | The Jaccard similarity index of sets $X$ and $Y$. |
| $d_J(X,Y)$ | The Jaccard distance between sets $X$ and $Y$; $d_J = 1 - J$. |
| **Algebraic Jaccard measures (*SimilarityAtScale*), details in III-A:** | |
| $m, n$ | The number of possible data values (attributes) and data samples One sample contains zero, one, or more values (attributes). |
| $\otimes, \odot$ | Matrix-vector (MV) and vector dot products. |
| $\boldsymbol{A}$ | The *indicator matrix* (it determines the presence of data values in data samples), $\boldsymbol{A} \in \mathbb{B}^{m \times n}, \mathbb{B} = \{0,1\}$. |
| $\boldsymbol{S}, \boldsymbol{D}$ | The *similarity* and *distance* matrices with the values of Jaccard measures between all pairs of data samples; $\boldsymbol{S}, \boldsymbol{D} \in \mathbb{R}^{n \times n}$. |
| $\boldsymbol{B}, \boldsymbol{C}$ | Intermediate matrices used to compute $\boldsymbol{S}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$. |
| **Genomics and metagenomics computations (*GenomeAtScale*):** | |
| $k$ | The number of single nucleotides in a genome subsequence. |
| $m, n$ | The number of analyzed $k$-mers and genome data samples. |

TABLE I: The most important symbols used in the paper.

### A. Fundamental Definitions

The *Jaccard similarity index* [38] is a statistic used to assess the similarity of sets. It is defined as the ratio of the set intersection cardinality and the set union cardinality,

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|},$$

where $X$ and $Y$ are sample sets. If both $X$ and $Y$ are empty, the index is defined as $J(X,Y) = 1$. One can then use $J(X,Y)$ to define the *Jaccard distance* $d_J(X,Y) = 1 - J(X,Y)$, which assesses the *dissimilarity* of sets and which is a proper *metric* (on the collection of all finite sets).

We are interested in the computation of similarity between all pairs of a collection of sets (in the context of genomics, this collection contains samples from one or multiple genomes). Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be the set of *data samples*. Each data sample consists of a set of positive integers up to $m$, so $X_i \subseteq \{1, \ldots, m\}$. We seek to compute the Jaccard similarity for each pair of data samples:

$$J(X_i, X_j) = \frac{|X_i \cap X_j|}{|X_i \cup X_j|}, \quad i, j \in \{1, \ldots, n\}. \tag{1}$$

### B. Computing Genetic Distances

Due to gaps in knowledge and the general complexity of computing evolutionary distances between samples of genetic sequences, much effort has been dedicated to efficiently computing accurate proxies for genetic distance [70]. The majority

of these works fall into one of two categories: *alignment-based* and *alignment-free* methods. When comparing two (or more) sequences, alignment-based methods explicitly take into account the ordering of the characters when determining the similarity. More specifically, such methods assume a certain mutation/evolutionary model between the sequences and compute a minimal-cost sequence of edits until all compared sequences converge. This family of methods forms a mature area of research. An established alignment-based tool for deriving genome distances is BLAST [2]. While highly accurate, these methods are computationally intractable when comparing sets of high-throughput sequencing data. Contrarily, alignment-free methods do not consider the order of individual bases or amino acids while computing the distances between analyzed sequences. These methods have been recently proposed and are in general much faster than alignment-based designs [70]. Both exact and approximate variants were investigated, including the mapping of $k$-mers to common ancestors on a taxonomic tree [66], and the approximation of the Jaccard similarity index using minwise hashing [42]. To the best of our knowledge, there is no previous work that enables distributed computation of a genetic distance that is fast, accurate, and *scales to massive sets of sequencing samples*.

A *genome* is a collection of sequences defined on the alphabet of the *nucleotides* adenine (A), thymine (T), guanine (G), and cytosine (C) (each individual sequence is referred to as a *chromosome*). A $k$-mer is a subsequence of length $k$ of a given sequence. For example, in a sequence AATGTC, there are four 3-mers (AAT, ATG, TGT, GTC) and three 4-mers (AATG, ATGT, TGTC). A common task in genomics is to reconstruct the *assembly* of chromosomes from one or more species given *high-throughput sequencing* samples. These assembled sequences allow for accurate, alignment-based methods to be used when assessing the similarity between samples. Yet, the high computational cost of assembly has meant that the vast majority of available sequencing data has remained unassembled [57]. Thus, there has been a growing interest in methods that enable such comparisons to be made on representations of sequencing data without requiring prior assembly. Alignment-free methods typically represent a sequencing sample $i$ as a set $X_i$ of its respective $k$-mers. From this, one may compute the genetic distance to another sample $X_j$ via the Jaccard similarity of $X_i$ and $X_j$ [42], or map each $k$-mer in $X_i$ to a database of labels for classification [66]. The distance matrix $1-J(i,j)$ may then be used for subsequent downstream tasks, including the clustering of samples for the construction of phylogenetic trees [46], to aid the selection of related samples for joint metagenomic analysis [43], or to aid the construction of guide trees for large-scale multiple sequence alignment [3] (see Figure 1).

We briefly discuss the scalability of different alignment-free tools for deriving genetic distances, and compare them to the proposed GenomeAtScale. The considered aspects are (1) the size of processed genome data (both the input size and the number of samples in this input), (2) the amount of used compute resources, and (3) the used measure of similarity and its accuracy. A comparison is presented in Table II. GenomeAtScale delivers the largest scale of genome distance computations in the all considered aspects. We will describe the evaluation in detail in Section V.

### C. General Data Science: Clustering

As the Jaccard distance $d_J$ is formally a metric, it can be used with many clustering routines. For example, one can use it with centroid-based clustering such as the popular $k$-means algorithm when deriving distances between data points and centroids of clusters [23], [67]. It can be used together with other classes of methods, for example hierarchical clustering [60], [21]. The advantage of using $d_J$ is that it is straightforwardly applicable to categorical data that does not consist of numbers but rather attributes that may be present or absent [40], [47].

### D. General Data Science: Anomaly Detection

Another use case for the Jaccard distance is anomaly detection through proximity-based outlier detection [35]. This application is particularly useful when the analyzed data contains binary or categorical values.

### E. Machine Learning: Object Detection

In object detection, the Jaccard similarity is referred to as *Intersection over Union* and it is described as the most popular evaluation metric [44]. Here, sets $X$ and $Y$ model two bounding boxes: a ground-truth bounding box around an object to be localized in a picture and a predicted bounding box. $|X \cap Y|$ is the overlap area of these two boxes; $|X \cup Y|$ constitutes the union area. The ratio of these two values assesses how well a predicted box matches the ideal box.

### F. Graph Analytics and Graph Mining

The Jaccard similarity is also used in graph analytics and mining, to compute the similarity of any two vertices $v, u$ using only the graph adjacency information. This similarity can be defined as $|N(v) \cap N(u)|/|N(v) \cup N(u)|$, where $N(v)$ and $N(u)$ are sets with neighboring vertices of $v$ and $u$, respectively [47]. Vertex similarity is often used as a building block of more complex algorithms. One example is Jarvis-Patrick graph clustering [30], where the similarity value determines whether $v$ and $u$ are in the same cluster. Other examples include discovering missing links [17] or predicting which links will appear in dynamic graphs [69].

### G. Information Retrieval

In the information retrieval context, $J(X,Y)$ can be defined as the ratio of the counts of common and unique words in sets $X$ and $Y$ that model two documents. Here, $J(X,Y)$ assesses the similarity of two such documents. For example, text2vec, an R package for text analysis and natural language processing [49], uses the Jaccard distance for this purpose.

## III. Communication-Efficient Jaccard Measures

We now describe **SimilarityAtScale**, a distributed algorithm for deriving Jaccard measures based on sparse linear algebra. In Section IV, we apply our algorithm to genomics.

| Tool | # compute nodes | # samples | Raw input data size | Preprocessed data size | Similarity |
|---|---|---|---|---|---|
| DSM [50] | 1 | 435 | 3.3TB | N/A‡ | Jaccard |
| Mash [42] | 1 | 54, 118 | N/A† | 674GB | Jaccard (MinHash) |
| Libra [18] | 10 | 40 | 372GB | N/A‡ | Cosine |
| GenomeAtScale | 1024 | 446, 506 | 170TB | 1.8TB | Jaccard |

TABLE II: **Comparison of scales of different alignment-free tools for deriving genetic distances.** Raw input data refers to high-throughput sequencing data, while preprocessed data refers to cleaned and assembled long sequences. All sizes given refer to uncompressed FASTA files. † Mash is constructed from assembled and curated reference genomes, where in some cases the corresponding raw sequencing data files may not be available. ‡ DSM and Libra directly query raw sequencing data with no assembly step. GenomeAtScale was computed from cleaned and assembled sequences (see Section V-A2 for details). **GenomeAtScale is shown to achieve larges problem size and parallelism scales than past approaches.**

### A. Algebraic Formulation of Jaccard Measures

We first provide an algebraic formulation of the Jaccard measures (the following description uses definitions from II-A). We define an *indicator matrix* $\boldsymbol{A} \in \mathbb{B}^{m \times n}$, where $\mathbb{B} = \{0, 1\}$. We have

$$a_{ij} = \begin{cases} 1 & : i \in X_j \\ 0 & : \text{otherwise} \end{cases}.$$

The matrix $\boldsymbol{A}$ determines the presence of data values in data samples. We seek to obtain the *similarity matrix*, $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ defined to obtain the similarities described in (1),

$$s_{ij} = J(X_i, X_j) = \frac{|X_i \cap X_j|}{|X_i \cup X_j|}.$$

To compute $\boldsymbol{S}$, it suffices to form matrices $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$, which describe the cardinalities of the intersections and unions of each pair of data samples, respectively. The computation of $\boldsymbol{B}$ is most critical, and can be described as a sparse matrix–matrix product,

$$b_{ij} = |X_i \cap X_j| = \sum_k a_{ki} a_{kj}, \text{ so } \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}.$$

The matrix $\boldsymbol{C}$ can subsequently be obtained via $\boldsymbol{B}$ (we use $\hat{\boldsymbol{a}} = (\hat{a}_1, ..., \hat{a}_n)^T$ to simplify notation):

$$\begin{aligned} c_{ij} &= |X_i \cup X_j| = |X_i| + |X_j| - |X_i \cap X_j| \\ &= \hat{a}_i + \hat{a}_j - b_{ij}, \text{ where } \hat{a}_i = \sum_k a_{ki}. \end{aligned}$$

The similarity and the distance matrices $\boldsymbol{S}$, $\boldsymbol{D}$ are given by

$$s_{ij} = b_{ij}/c_{ij}, d_{ij} = 1 - s_{ij}, \quad i, j \in \{1, ..., n\}, \tag{2}$$

The formulation and the algorithm are generic and can be used in any setting where compared data samples are *categorical* (i.e., they consist of *attributes* that may or may not be present in a given sample). For example, a given genome data sample usually contains some number of $k$-mers that form a subset of all possible $k$-mers. Still, most numerical data can be transformed into the categorical form. For example, the neighborhood $N(v)$ of a given graph vertex $v$ usually contains integer vertex IDs ($N(v) \subset \mathbb{N}$). Hence, one can model all neighborhoods with the adjacency matrix [9].

### B. Algorithm Description

The formulation of Jaccard similarity computation via sparse linear algebra derived in Section III-A is succinct, but retains some algorithmic challenges. First, the data samples (nonzeros contained in indicator matrix $\boldsymbol{A}$) may not simultaneously fit in the combined memory of all nodes on a distributed-memory system. Second, the similarity matrix $\boldsymbol{S}$ may not fit in the memory of a single node, but should generally fit in the combined memory of the parallel system. *Further, the most significant computational challenge is that the indicator matrix $\boldsymbol{A}$ is incredibly sparse for genomic similarity problems.* The range of $k$-mers generally extends to $m = 4^{30}$. This means that $\boldsymbol{A}$ is very hypersparse [13], i.e., the overwhelming majority of its rows are entirely zero. To resolve this challenge, the SimilarityAtScale algorithm makes use of three techniques:

1) subdividing $\boldsymbol{A}$'s rows into batches, each with $\tilde{m}$ rows,
2) filtering zero rows within each batch using a distributed sparse vector,
3) masking row segments into bit vectors.

We now describe in detail how these techniques are deployed. A high-level pseudocode can be found in Listing 1 while more details are provided in Listing 2.

```
//For any details of specific structures or operations, see Listing 2
//Below, we refer to respective equations in the text

for each batch of the input matrix A { //Batches are defined in Eq.(3).
  Read the next l-th batch A^(l) of A
  Remove zero rows from A^(l) using a filter f^(l), the result is Ā^(l)
  //The filter f^(l) and the matrix Ā^(l) are defined in Eq.(5) and (6).
  Compress Ā^(l) with bitmasking, the result is Â^(l)
  //The matrix Â^(l) and bitmasking are defined between Eq.(5) and (6).
  Compute the partial scores S^(l) = A^(l)T A^(l) and â^(l)
  //The partial scores are defined implicitly in Eq.(4) and (7).
  Accumulate the partial scores into intermediate matrices B and â
  //Intermediate matrices B and â are defined in Eq.(4).
  //Recall that B describes the cardinalities of intersections of
  //pairs of data samples, and - together with â - can be used
  //to describe the cardinalities of unions of pairs of data samples.
Derive the final similarity scores S based on B and â
//Batches S^(l) of S are defined in Eq.(7), S is defined in Eq.(2).
```

Listing 1: **High-level pseudocode** of the SimilarityAtScale algorithm. All mathematical symbols are defined in Section III-A and listed in Table I.

We subdivide the indicator into batches $r = m/\tilde{m}$ (to simplify the presentation we assume $m$ divides into $\tilde{m}$) batches:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}^{(1)} \\ \vdots \\ \boldsymbol{A}^{(r)} \end{bmatrix}, \text{ where } \boldsymbol{A}^{(l)} \in \mathbb{B}^{\tilde{m} \times n}, \forall l \in \{1, \ldots, r\}. \tag{3}$$

```
/*              Notation remarks, description of input and output              */
/* For simplicity, we use the same symbols for the variables that
 * correspond to the introduced mathematical objects: matrices A, B,
 * C, S; a number of all data samples n and attributes m.
 * Input: ``files'': an array of pointers to n files, where one file
 * contains data values from one data sample X_i (i ∈ {1, ..., n}).
 * ``batch_cnt'': the number of batches into which we partition
 * the derivation and processing of matrices A, B, and C.
 * ``max_val'': the maximum value across all samples X_i.
 * ``bitmask'': a bitmask used to compress the (boolean) input data
 * and reduce the memory footprint in A. ``comm'' is an object with
 * details of the distributed execution (e.g., process count).
 * ``Value'' is an opaque object that represents an arbitrary value
 * possibly contained in input data (e.g., a number, a letter).
 * Output: The Jaccard similarity matrix S (it can be trivially
 * transformed into the Jaccard distance matrix D). */

/*  A part that combines all elements used to derive the Jaccard measures   */
typedef vector<pair<index, data>> Vector;

// Derive and return the Jaccard similarity matrix.
Matrix* SimilarityAtScale(File* files, int n, int m,
    int bitmask, Comm* comm, int batch_cnt) {
  int batch_cnt_tot = batch_cnt + (m % batch_cnt) > 0;
  //"DMatrix"/"DVector" indicate a distributed matrix/vector
  DMatrix A, B, C; // Declare opaque matrix objects.
  DVector f; // Declare a temporary data structure for input.

  for(int i = 0; i < batch_cnt_tot; i++) {
    readFiles(files, n, comm, &f); // Read input in batches.
    // Compress each input batch and remove zero rows.
    preprocessInput(n, m, batch_cnt, bitmask, &f, &A);
    jaccardAccumulate(&A, &B, &C); // Construct matrices A, B, C.
  }
  C["ij"] -= B["ij"];
  S["ij"] += B["ij"] / C["ij"]; // Derive the similarity matrix.
  return &S; // Return the Jaccard similarity matrix.
}

/*                          Loading input files                          */
void readFiles(File* files, int n, Comm* comm, DVector* f) {
  // This function is executed by each process in parallel.
  Vector data_sample;
  for(int i = comm->my_rank; i < n; i += comm->num_procs) {
    // One file line contains one data value.
    Value val = files[i]->read_file_line();
    // Store the value in a tuple; <index=val,data=1>.
    data_sample.push(val, 1);
  }
  // Bulk update a structure with loaded data (f[data_value.index]=1).
  // We use a write function on the vector in which
  // all processes update the loaded data in parallel.
  f.write(data_sample);
}

/*      Preprocessing (removing zero rows, compression using the bitmask)     */
void preprocessInput(int n, int m, int batch_cnt, int bitmask,
                     Vector* R, Matrix* A) {
  int len_bm = length(bitmask); // Get the bitmask length.
  // This function is executed by each process in parallel.
  // Get the non-zero data indices.
  Vector* nonzero_data = get_all_nonzero_pairs(R);
  // ``nonzero_data'' effectively represents the non-zero rows.
  for(int i = comm->my_rank; i < n; i += comm->num_procs) {
    int j = 0; int mask = 0;
    Vector masked_data_sample;
    while (j < m) {
      Value val = data_sample[j++].first;
      int l = 0;
      while (j < nonzero_data.size && l++ < len_bm) {
        if (nonzero_data.index == (val % (m / batch_cnt))) {
          // The iteration follows the column-major order
          // (this reflects our implementation).
          mask |= ((bitmask)1) << ((j % (m / batch_cnt))) % len_bm);
      } }
      if (mask) masked_data_sample.push(mask_index, mask);
  } }
  write(A, masked_data_sample); // Bulk update A.
}

/*              Deriving intermediate matrices B and C in batches              */
void jaccardAccumulate(Matrix* A, Matrix* B, Matrix* C) {
  // ``popcount'' counts the number of ``ones'' in a given row/column.
  // The operations below follow the specification in III-A.
  B["ij"] = popcount(A["ki"] & A["kj"]);
  v["i"] += popcount(A["ki"]);
  C["ij"] += v["i"] + v["j"];
}
```

Listing 2: The details of the SimilarityAtScale algorithm. All mathematical symbols are defined in Section III-A and listed in Table I.

To obtain the similarity matrix $\mathbf{S}$, we need to obtain $\hat{\boldsymbol{a}}$ and $\boldsymbol{B}$, which can be combined by accumulation of contributions from each batch,

$$\boldsymbol{B} = \sum_{l=1}^{r} \boldsymbol{A}^{(l)T}\boldsymbol{A}^{(l)}, \quad \hat{\boldsymbol{a}} = \sum_{l=1}^{r} \hat{\boldsymbol{a}}^{(l)}, \text{ where } \hat{a}_i^{(l)} = \sum_k a_{ki}^{(l)}. \quad (4)$$

To filter out nonzero rows in a batch, we construct a sparse vector $\boldsymbol{f}^{(l)} \in \mathbb{B}^{\tilde{m}}$ that acts as a filter,

$$f_k^{(l)} = \begin{cases} 1 & : \exists i \in \{1, \ldots, n\}, a_{ki}^{(l)} \neq 0 \\ 0 & : \text{otherwise} \end{cases}. \quad (5)$$

Given the prefix sum $\boldsymbol{p}^{(l)}$ of $\boldsymbol{f}^{(l)}$, the batch of the indicator matrix $\boldsymbol{A}^{(l)}$ can be reduced to a matrix $\bar{\boldsymbol{A}}^{(l)}$ that contains only nonzero rows,

$$\bar{a}_{p_k^{(l)} i}^{(l)} = a_{ki}^{(l)}. \quad (6)$$

Subsequently, it suffices to work with $\bar{\boldsymbol{A}}^{(l)}$ since

$$\boldsymbol{A}^{(l)T}\boldsymbol{A}^{(l)} = \bar{\boldsymbol{A}}^{(l)T}\bar{\boldsymbol{A}}^{(l)}.$$

Even after removal of nonzero rows in each batch of the indicator matrix, it helps to further reduce the number of rows in each $\bar{\boldsymbol{A}}^{(l)}$. The meta-data in both COO and CSR formats necessary to store each nonzero corresponds to a 32-bit or 64-bit integer. In the CSR layout, the same amount of meta-data is necessary to store each "row start" count. We reduce the latter overhead by leveraging the fact that each binary value only requires one bit of data. In particular, we encode segments of $b$ elements of each column of $\bar{\boldsymbol{A}}^{(l)}$ in a $b$-bit bitmask. A natural choice is $b = 32$ or $b = 64$, which increases the storage necessary for each nonzero by no more than $2{-}3\times$, while reducing the number of rows (and consequently row-start counts in the CSR representation) by $b$, as well as potentially reducing the number of actual nonzeros stored. The resulting matrix $\hat{\boldsymbol{A}}^{(l)} \in \mathbb{S}^{(\tilde{m}/b)\times n}$ where $\mathbb{S} = \{0, \ldots, 2^b - 1\}$, can be used effectively to compute the similarity matrix. In particular, for $\boldsymbol{S}^{(l)} = \boldsymbol{A}^{(l)T}\boldsymbol{A}^{(l)}$, we have

$$s_{ij}^{(l)} = \sum_k \text{popcount}(\hat{a}_{ki}^{(l)} \wedge \hat{a}_{kj}^{(l)}), \quad (7)$$

where $\text{popcount}(x)$ counts the number of set bits in $x$.

### C. Parallelization and Analysis

We assume without loss of generality that the rows and columns of $\boldsymbol{A}$ are randomly ordered, which can be enforced via a random reordering for any other datasets. Consequently, the cost of computation of each batch is roughly the same. Let $h = \tilde{m}/b$ so that $\boldsymbol{R} = \hat{\boldsymbol{A}}^{(l)} \in \mathbb{S}^{h\times n}$, and let $z$ be the number of nonzeros in $\boldsymbol{R}$. We assume $b = O(1)$ so that the cost of popcount is $O(1)$ arithmetic operations. We first analyze the cost of the sparse matrix–matrix product, then quantify the overheads of the initial compression steps. We consider a $p$ processor Bulk Synchronous Parallel (BSP) model [62], [51], where each processor can store $M$ words of data, the cost of a superstep (global synchronization) is $\alpha$, the per byte bandwidth cost is $\beta$, and each arithmetic operation has cost $\gamma$. We assume $\alpha \geq \beta \geq \gamma$.

Our parallelization and analysis of the sparse matrix–matrix product follows known communication-avoiding techniques for (sparse) matrix–matrix multiplication [54], [53], [4], [32], [5], [25], [15]. Given enough memory to store $1 \leq c \leq p$ copies of $\boldsymbol{B}$, i.e., $M = \Omega(cn^2/p)$, we define a $\sqrt{p/c} \times \sqrt{p/c} \times c$ processor grid. On each $\sqrt{p/c} \times \sqrt{p/c}$ subgrid, we compute $1/c$th of the contributions to $\boldsymbol{B}$ from a given batch of the indicator matrix, $\boldsymbol{R}$. Each processor then needs to compute $\boldsymbol{R}^{(s,t)T}\boldsymbol{R}^{(s,v)}$, where each $\boldsymbol{R}^{(i,j)}$ is a $h/c \times n\sqrt{c/p}$ block of $\boldsymbol{R}$. For a sufficiently large $z, c, p$, w.h.p., #nonzeros in each $\boldsymbol{R}^{(i,j)}$ is $O(z/\sqrt{cp})$. Using a SUMMA [25], [15], [63] algorithm, computing $\boldsymbol{R}^{(s,\star)T}\boldsymbol{R}^{(s,\star)}$ on the $s$th layer of the processor grid takes $O(1 + z/(M\sqrt{cp}))$ BSP supersteps. Finally, assuming $c > 1$, one needs a reduction to sum the contributions to $\boldsymbol{C}$ for each layer, which requires $O(1)$ supersteps, where each processor sends/receives at most $O(cn^2/p)$ data. The overall BSP communication cost of our algorithm is

$$O\left(\left(1 + \frac{z}{M\sqrt{cp}}\right) \cdot \alpha + \left(\frac{z}{\sqrt{cp}} + \frac{cn^2}{p}\right) \cdot \beta\right).$$

The number of arithmetic operations depends on the sparsity structure (i.e., if some rows of $\boldsymbol{R}$ are dense and others mostly zero more operations are required per nonzero than if the number of nonzeros per row is constant). For a sufficiently large $z, c, p$ these total number of arithmetic operations, $F$, will be evenly distributed among processors, yielding a BSP arithmetic cost of $O((F/p) \cdot \gamma)$.

We allow each processor to read an independent set of data samples from disk for each batch. Assuming $n \gg p$ and that no row contains more than $O(z/(p \log p))$ nonzeros (the average is $O(n/z)$), each processor reads $O(z/p)$ nonzeros. This assumption may be violated if columns of $\boldsymbol{A}$ have highly variable density and $p$ approaches $n$, in which case the resulting load imbalance would restrict the batch size and create some overhead. To filter out nonzero rows, we require the computation of $\boldsymbol{f}^{(l)}$ and its prefix sum $\boldsymbol{p}^{(l)}$. We perform a transposition of the initial data, so that each process collects all data in $m/(rp)$ of the $m/r$ rows in the batch, and combines it locally to identify nonzero rows. If the number of nonzeros $z$ and total rows $bh$ is sufficiently large, i.e., $z, bh \gg p$, each processor receives $O(z/p)$ entries and ends up with $O(bh/p)$ of then nonzero rows of $\boldsymbol{f}^{(l)}$. Subsequently, a prefix sum of the nonzero entries of $\boldsymbol{f}^{(l)}$ can be done with BSP cost,

$$O(\alpha + p \cdot \beta),$$

assuming $p = O(M)$. The nonzero entries received by each processor can then be mapped to the appropriate row in the $bh \times n$ matrix $\bar{\boldsymbol{A}}^{(l)}$ and sent back to the originating processor. At that point, each processor can compress the columns of $\bar{\boldsymbol{A}}^{(l)}$ by a factor of $b$ to produce the $h \times n$ matrix $\boldsymbol{R}$.

The overall BSP cost per batch assuming $b = O(1)$ is

$$T(z, n, M, c, p) = O\left(\left(1 + \frac{z}{M\sqrt{cp}}\right) \cdot \alpha + \left(\frac{z}{\sqrt{cp}} + \frac{cn^2}{p} + p\right) \cdot \beta + \frac{F}{p}\gamma\right).$$

Generally, we pick the batch size to use all available memory, so $z = \Theta(Mp)$, and replicate $\boldsymbol{B}$ in so far as possible, so $c = \Theta(\min(p, Mp/n^2))$. Given this, and assuming $p = O(M)$ and $M \leq n^2$, which is the "memory-bound regime", which is critical in all of our experiments, the above cost simplifies to

$$\tilde{T}(n, M, p) = O\left(\frac{n}{\sqrt{M}} \cdot \alpha + n\sqrt{M} \cdot \beta + \frac{F}{p}\gamma\right).$$

For a problem with $m$ rows and $Z$ nonzeros overall, requiring $G$ arithmetic operations overall, maximizing the batch size gives the total cost,

$$\frac{Z}{Mp}\tilde{T}(n, M, p) = O\left(\frac{nZ}{pM^{3/2}} \cdot \alpha + \frac{nZ}{\sqrt{M}p} \cdot \beta + \frac{G}{p}\gamma\right).$$

These costs are comparable to the ideal cost achieved by parallel dense matrix–matrix multiplication [6] in the memory-dependent regime, where $Z = n^2$ and $G = n^3$.

Given a problem where the similarity matrix fits in memory with $p_0$ processors, i.e., $M = n^2/p_0$, we can consider strong scaling where batch size is increased along with the number of processors, until the entire problem fits in one batch. The parallel efficiency is then given by the ratio of BSP cost for computing a batch with $z_0 = O(n^2)$ nonzeros and $h_0$ nonzero rows with $p_0$ processors to computing a larger batch with $z = O(n^2 \cdot p/p_0)$ nonzeros and $h = h_0 \cdot p/p_0$ nonzero rows using up to $p = O(\min(M, n))$ processors,

$$E_p = \frac{T(z_0, n, n^2/p_0, 1, p_0)}{T(pz_0/p_0, n, n^2/p_0, p/p_0, p)} = O(1).$$

Thus, our algorithm can achieve perfect strong scalability so long as the load balance assumptions are maintained. These assumptions hold given either balanced density among data samples or a sufficiently large number of data samples, and so long as the number of processors does not exceed the local memory or the dimension $n$.

### D. Algebraic Jaccard for Different Problems

We briefly explain how to use the algebraic formulation of the Jaccard measures in selected problems from Section II. The key part is to properly identify and encode data values and data samples within the indicator matrix $\mathbf{A}$. We illustrate this for selected problems in Table III.

| Computational problem | One row of $\mathbf{A}$ | One column of A |
|---|---|---|
| Distance of genomes | One $k$-mer | One genome data sample |
| Similarity of vertices | Neighbors of one vertex | Neighbors of one vertex |
| Similarity of documents | One word | One document |
| Similarity of clusters | One vertex | One cluster |

TABLE III: Framing of the SimilarityAtScale algorithm for different computational problems. $\mathbf{A} \in \{0,1\}^{m \times n}$ is the indicator matrix that determines the presence of data values in the compared data samples, detailed in Section III-A.

## IV. Implementation

We package the SimilarityAtScale algorithm as part of **GenomeAtScale**, a tool for fast distributed genetic distance computation. Figure 1 illustrates the integration of GenomeAtScale with general genomics and metagenomics

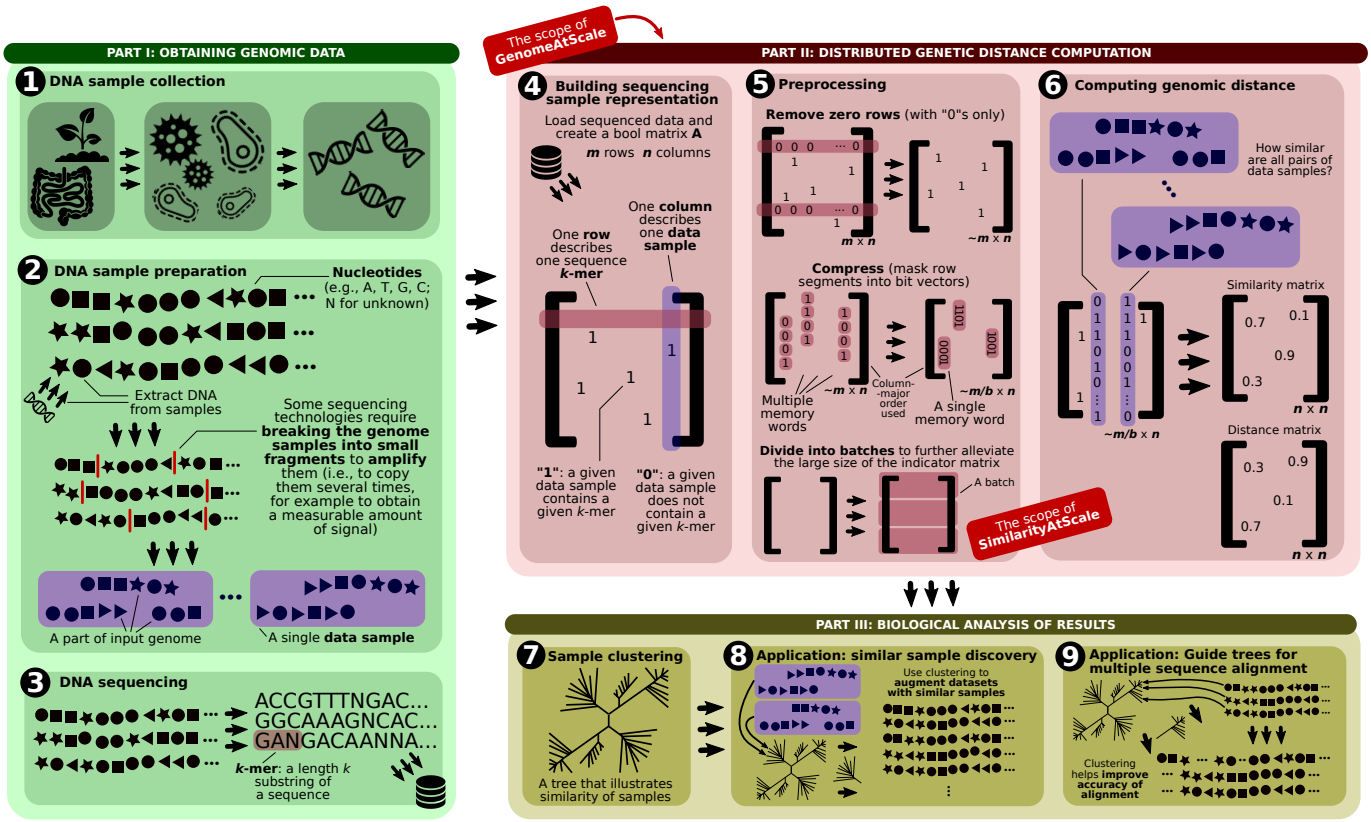



Fig. 1: **The scope of the SimilarityAtScale algorithm and GenomeAtScale tool within a metagenomics project.** DNA is sequenced and preprocessed before being deposited into a database (❶–❸). Given sequence data from several samples, a binary matrix $\boldsymbol{A}$ is constructed indicating which $k$-mers are present in which samples (❹). The matrix is divided into batches and pair-wise Jaccard similarities are computed (❺–❻). The results may then be used for downstream genomics analysis (❼–❾).

projects. GenomeAtScale includes infrastructure to produce files with a sorted numerical representation for each data sample. Each processor is responsible for reading in a subset of these files, scanning through one batch at a time. Once the data is read-in, the SimilarityAtScale implementation performs preprocessing and parallel sparse matrix mutliplication.

To realize both preprocessing and sparse matrix multiplication, we use the Cyclops library [56]. Cyclops is a distributed-memory library that delivers routines for contraction and summation of sparse and dense tensors. The library also provides primitives for sparse input and transformation of data. Importantly, Cyclops enables the user to work with distributed vectors/matrices/tensors with arbitrary fixed-size element data-types, and to perform arbitrary elementwise operations on these data-types. This generality is supported via C++ templating, lambda functions, and constructs for algebraic structures such as monoids and semirings [53], [55]. Cyclops relieves the user of having to manually determine the matrix data distribution. Cyclops automatically distributes matrices over all processors using a processor grid. Each routine searches for an optimal processor grid with respect to communication costs and any additional overheads, such as data redistribution.

Listing 2 provides the pseudocode for our overall approach. We describe details of how preprocessing and similarity calculation are done with Cyclops below.

### A. Implementation of Preprocessing

After obtaining genome data (Part I), we load the input sequence files (using the established FASTA format [39]) and construct a sparse representation of the indicator matrix $\mathbf{A}$. The `readFiles()` function then processes the input data, and writes into a Cyclops sparse vector $\mathbf{f}$, i.e., each $k$-mer is treated as an index to update $\mathbf{f}$ with 1. To do this, we leverage the Cyclops `write()` function, which collects arbitrary inputs from all processes, combining them by accumulation, as specified by the algebraic structure associated with the tensor. We make use of the $(\max, \times)$ semiring for $\mathbf{f}$ so that each vector entry is 1 if *any* processor writes 1 to it.

Our implementation then proceeds by collecting the sparse vector $\mathbf{f}$ on all processors, and performing a local prefix sum to determine appropriate nonzero rows for each data item. This approach has been observed to be most efficient for the scale of problems that we consider in the experimental section. The use of the Cyclops `read()` function in place of replication would yield an implementation that matches the algorithm description and communication cost. The function `preprocessInput()` then proceeds to map the locally stored entries to nonzero rows as prescribed by the prefix sum of the filter and to apply the masking. The distributed Cyclops matrix storing the batch of the indicator matrix $\bar{\boldsymbol{A}}^{(l)}$ is then created by a call to `write()` from each processor with the nonzero entries it generates.

### B. Implementation of Semiring Sparse Matrix Multiplication

Given the generated sparse matrix $\bar{\boldsymbol{A}}^{(l)}$, which corresponds to $\mathbf{A}$ in our pseudocode, the function `jaccardAccumulate()` proceeds to compute the contribution to $\boldsymbol{B}$. To do this with Cyclops, we define a dense distributed matrix $\mathbf{B}$, and use the Einstein summation syntax provided by Cyclops to specify the

matrix-multiplication. The use of the appropriate elementwise operation is specified via a Cyclops `Kernel` construct, which accepts an elementwise function for multiplication (for us `popcount`, which counts number of set bits via a hardware-supported routine) and another function for addition, which in our case is simply the addition of 64-bit integers. The Einstein summation notation with this kernel is used as follows

```
Jaccard_Kernel(A["ki"],A["kj"],B["ij"]);
```

Aside from this sparse matrix multiplication, it suffices to compute a column-wise summation of the matrix $\mathbf{A}$, which is done using similar Cyclops constructs.

The resulting implementation of sparse matrix multiplication is fully parallel, and can leverage 3D sparse matrix multiplication algorithms. The $\sqrt{p/c} \times \sqrt{p/c} \times c$ processor grid proposed in Section III-C is within the space of processor grids considered by Cyclops. Our results (almost ideal scaling) confirm that the desired scaling is achieved for the considered parameters' spectrum.

R1

## V. Evaluation

We now analyze the performance of our implementation of SimilarityAtScale for real and synthetic datasets.

### A. Methodology and Setup

We first provide information that is required for interpretability and reproducibility [26].

*1) Experimental Setup:* We use the Stampede2 supercomputer. Each node has a Intel Xeon Phi 7250 CPU ("Knights Landing") with 68 cores, 96GB of DDR4 RAM, and 16GB of high-speed on-chip MCDRAM memory (which operates as 16GB direct-mapped L3). There is also 2KB of L1 data cache per core and 1MB of L2 per two-core tile. There are 4,200 compute nodes in total. The network is a fat tree with six core switches, with the 100 Gb/sec Intel Omni-Path architecture.

In our experiments, we consistently use 32 MPI processes per node. Using fewer processes per node enables larger batch sizes as our implementation replicates the filter vector on each processor. We also find that this configuration outperforms those with 64 processes per node for representative experiments, as the on-node computational kernels are generally memory-bandwidth bound.

To maximize fair evaluation, we include the I/O time (loading data from disk) in the reported runtimes (the I/O time is ≈1% of the total runtime).

R3

*2) Considered Real Datasets:* We evaluate our design on datasets of differing sequence variability to demonstrate its scalability in different settings. As a **low-variability set**, we use the public BBB/Kingsford dataset consisting of 2,580 RNASeq experiments sequenced from human blood, brain, and breast samples [52] with sequencing reads of length at least 20. We consider all such experiments that were publicly available at the time of study. The raw sequences were preprocessed to remove rare (considered noise) $k$-mers. Minimum $k$-mer count thresholds were set based on the total sizes of the raw sequencing read sets [52]. We used the $k$-mer size of 19 (unlike the value of 20 used in [52]) to avoid the possibility of $k$-mers being equal to their reverse complements. As a **high-variability set**, we use all bacterial and viral whole-genome sequencing data used in the BIGSI database [11], representing almost every such experiment available as of its release, totaling 446,506 samples (composed overwhelmingly of Illumina short-read sequencing experiments). In the same fashion as the BIGSI, these data were preprocessed by considering longer contiguous stretches of $k$-mers to determine $k$-mer count thresholds [11]. We used the same $k$-mer size as the BIGSI paper ($k = 31$).

The considered real datasets enable analyzing the performance of our schemes for different data sparsities. The indicator matrix $\boldsymbol{A}$ in the Kingsford dataset has a density of $\approx 1.5 \cdot 10^{-4}$, and in the BIGSI dataset its density is $\approx 4 \cdot 10^{-12}$. All input data is provided in the FASTA format [39].

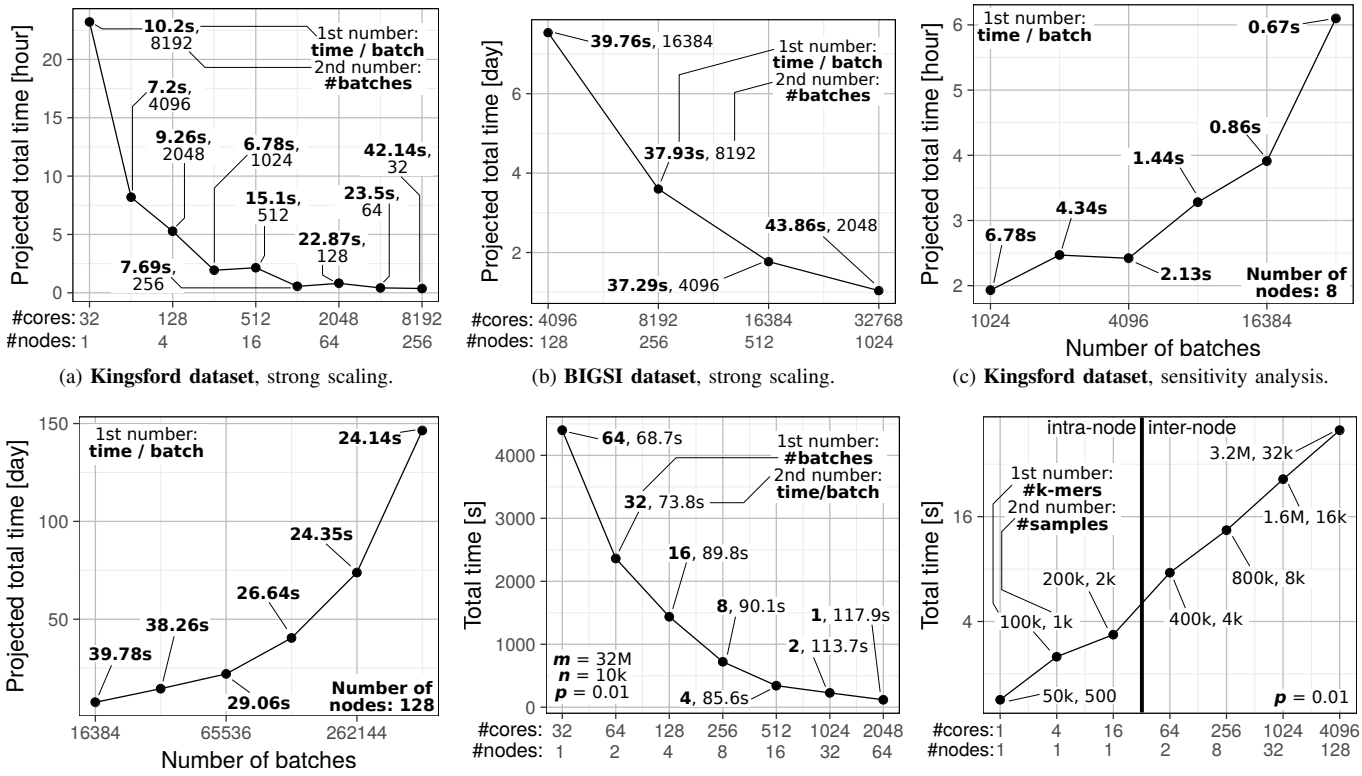
*3) Considered Synthetic Datasets:* We also use synthetic datasets where each element of the indicator matrix $\boldsymbol{A}$ is present with a specified probability $p$ (which corresponds to density), independently for all elements. This enables a systematic analysis of the impact of data sparsity on the performance of our schemes.

*4) Considered Scaling Scenarios:* To illustrate the versatility of our design, we consider (1) strong scaling for a small dataset (fixed indicator matrix ($\boldsymbol{A}$) size, increasing batch size and core count), (2) strong scaling for a large dataset (same as above), (3) weak scaling (increasing the $\boldsymbol{A}$ size with core count, increasing batch size with core count), (4) batch size sensitivity (fixed node count, increasing batch size).

### B. Performance Analysis for Real Data

The results for the Kingsford and the BIGSI datasets are presented in Figure 2a and 2b, respectively. The BIGSI dataset as noted has $n = 446,506$ columns. This requires us to distribute three matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ among the processes for the similarity calculation. We find it necessary to use 64 nodes to have enough memory to store these matrices. Hence, we report performance numbers for 128, 256, 512, and 1024 nodes. As we double the number of nodes, we also double the batch size, utilizing all available memory. We also use not more than 256 nodes for preprocessing, i.e., the sparse vector is constructed using not more than 256 nodes, but is used by all the participating nodes in the later stages of the pipeline. We find less variability in performance with this variant.

In Figure 2b, we show the average batch time (averaged across eight batches, not considering the first three batches to account for startup cost). Per batch time across nodes, remains the same (the batch size though is doubled according to the node size). The y-axis shows the predicted completion time for running the entire dataset to calculate the similarity matrix. We note that we are able to calculate the similarity matrix for BIGSI benchmark in a day (24.95 hours) using 1024 nodes. We note that despite high-variability of density across different columns in the BIGSI dataset, SimilarityAtScale achieves a good parallel efficiency even when using 32K cores.




(a) **Kingsford dataset**, strong scaling. (b) **BIGSI dataset**, strong scaling. (c) **Kingsford dataset**, sensitivity analysis.

(d) **BIGSI dataset**, sensitivity analysis. (e) **Synthetic dataset**, strong scaling. (f) **Synthetic dataset**, weak scaling.

Fig. 2: **Performance analysis of GenomeAtScale**. We calculate 95% confidence intervals for the reported mean values. **The derived confidence intervals are very tight around the means, and we exclude them from the plot to ensure clarity.** The confidence values for the BIGSI dataset are 0.12 (for 128 nodes), 0.16 (for 256 nodes), 0.38 (for 512 nodes), and 0.40 (for 1024 nodes).

R1

Similarly, in Figure 2a, we show the results for the Kingsford dataset which is denser than BIGSI. The performance behavior observed for this smaller dataset is a bit less consistent. We observe both superscalar speed-ups and some slowdowns when increasing node count. On 32 nodes, we note a sweet-spot, achieving a 42.2× speed-up relative to the single node performance. Thus, we are able to construct the similarity matrix in less than an hour using 32 nodes. For larger node counts, the number of MPI processes (2048, 4096, 8192) starts to exceed the number of columns in the matrix (2,580), leading to load imbalance and deteriorating performance.

To verify the projected execution times, we fully process Kingsford for 128 nodes and 64 batches (we cannot derive data for *all* parameter values due to budget constraints). The total runtime takes 0.38h, the corresponding projection is 0.42h.

R1

In Figures 2c and 2d, we show the sensitivity of the datasets for the size of the batches. For both datasets we observe a general trend that the execution time does not scale with batch size, despite the work scaling linearly with batch size. This behavior is expected, as a larger batch size has a lesser overhead in synchronization/latency and bandwidth costs, enabling a higher rate of performance. Thus, in both the datasets the overall projected time for the similarity matrix calculation reduces with the increase in batch size.

### C. Performance Analysis for Synthetic Data

In Figure 2e, we present the strong scaling results (with the increasing batch size) for synthetic data. The total time decreases in proportion to the node count, although the time per batch slightly increases, yielding good overall parallel efficiency, as predicted in our theoretical analysis. In Figure 2f, we show weak scaling by increasing the $\boldsymbol{A}$ matrix size *and* the batch size with the core count. In this weak scaling regime, the amount of work per processor is increasing with the node count. From 1 core to 4096 cores, the amount of work per processor increases by 64×, while the execution time increases by 35.3×, corresponding to a 1.81× efficiency improvement.

We also show how the performance for synthetic datasets changes with data sparsity expressed with the probability $p$ of the occurrence of a particular $k$-mer. The results are in Figure 3 . We enable nearly ideal scaling of the *total runtime* with the decreasing data sparsity (i.e., with more data to process).

R1

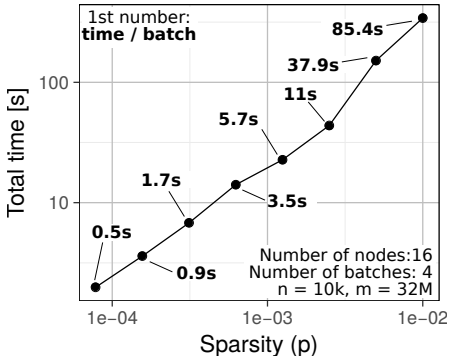



Fig. 3: The impact of data sparsity on the performance of GenomeAtScale.

### D. Impact from Fast Cache

We also test our design *without using MCDRAM as L3 cache*, but instead as an additional memory storage. The resulting performance patterns are negligibly worse than the ones in which MCDRAM serves as L3. For example, time per batch with MCDRAM as L3 for Kingsford dataset on 4 nodes and 32 nodes is 9.26s and 7.69s, respectively. Then, without MCDRAM L3 cache, it is 9.33s and 8.01s, respectively.

R3

## VI. Discussion

The algebraic formulation of our schemes is generic and can be implemented with other frameworks such as CombBLAS [14], [31]. However, we selected Cyclops which is – to the best of our knowledge – the only library that offers high-performance routines where *the input matrices are sparse but the outcome of the matrix-matrix multiplication is dense*. CombBLAS targets primarily graph processing and, to the best of our knowledge, does not provide a fast implementation of matrix-matrix product with a dense output. Thus, CombBLAS would result in suboptimal performance when combined with SimilarityAtScale. One could also use MapReduce [20] and engines such as Spark [68] or Hadoop [65]. However, they are communication-intensive and their limited expressiveness often necessitates multiple communication rounds, *resulting in inherent overheads when compared to communication-avoiding and expressive algebraic routines provided by Cyclops*.

## VII. Conclusion

We introduce **SimilarityAtScale**, the first high-performance distributed algorithm for computing the Jaccard similarity, which is widely used in data analytics. SimilarityAtScale is based on an algebraic formulation that uses (1) linear algebra routines that are provably communication-efficient, (2) compression based on bitmasking, (3) batched computation to alleviate large input sizes, and (4) theoretical analysis that illustrates scalability in communication cost and parallel efficiency. The result is a generic high-performance algorithm that can be applied to any problem in data analytics that relies on Jaccard measures. To facilitate the utilization of SimilarityAtScale in different domains, we provide a comprehensive overview of problems that could be accelerated and scaled with our design.

We then use SimilarityAtScale as a backend to develop **GenomeAtScale**, the first tool for accurate large-scale calculations of distances between high-throughput whole-genome sequencing samples on distributed-memory systems. To foster DNA research, we use real established datasets in our evaluation, illustrating that – for example – GenomeAtScale enables analyzing all the bacterial and viral whole-genome sequencing data used in the BIGSI database in less than a day. We maintain compatibility with standard bioinformatics data formats, enabling seamless integration of GenomeAtScale with existing biological analysis pipelines. We use GenomeAtScale to deliver largest-scale exact computations of Jaccard genetic distances so far. Our publicly available design and implementation can be used to foster further research into DNA and general data analysis.

## References


[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] J. Armstrong, G. Hickey, M. Diekhans, A. Deran, Q. Fang, D. Xie, S. Feng, J. Stiller, D. Genereux, J. Johnson, V. D. Marinescu, D. Haussler, J. Alföldi, K. Lindblad-Toh, E. Karlsson, G. Zhang, and B. Paten. Progressive alignment with Cactus: a multiple-genome aligner for the thousand-genome era. *bioRxiv*, 2019.

[4] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[5] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 222–231, New York, NY, USA, 2013. ACM.

[6] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in linear algebra. *SIAM J. Mat. Anal. Appl.*, 32(3), 2011.

[7] J. Bank and B. Cole. Calculating the Jaccard similarity coefficient with map reduce for entity pairs in wikipedia. *Wikipedia Similarity Team*, pages 1–18, 2008.

[8] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007.

[9] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–41. IEEE, 2017.

[10] I. Binanto, H. L. H. S. Warnars, B. S. Abbas, Y. Heryadi, N. F. Sianipar, and H. E. P. Sanchez. Comparison of similarity coefficients on morphological rodent tuber. In *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*, pages 104–107. IEEE, 2018.

[11] P. Bradley, H. C. den Bakker, E. P. Rocha, G. McVean, and Z. Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology*, 37(2):152, 2019.

[12] B. Brekhna, C. Zhang, and Y. Zhou. An experimental approach for evaluating superpixel's consistency over 2D Gaussian blur and impulse noise using Jaccard similarity coefficient. *International Journal of Computer Science and Security (IJCSS)*, 13(3):53, 2019.

[13] A. Buluç and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[14] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[15] A. Buluç and J. R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[16] P. Burkhardt. Asking hard graph questions. *US National Security Agency Technical report NSA-RD-2014-050001v1*, 2014.

[17] H.-H. Chen, L. Gou, X. L. Zhang, and C. L. Giles. Discovering missing links in networks using vertex similarity measures. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 138–143. ACM, 2012.

[18] I. Choi, A. J. Ponsero, M. Bomhoff, K. Youens-Clark, J. H. Hartman, and B. L. Hurwitz. Libra: scalable k-mer–based tool for massive all-vs-all metagenome comparisons. *GigaScience*, 8(2):giy165, 2018.

[19] M. Cosulschi, M. Gabroveanu, F. Slabu, and A. Sbircea. Scaling up a distributed computing of similarity coefficient with mapreduce. *IJCSA*, 12(2):81–98, 2015.

[20] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] Y. Dong, Y. Zhuang, K. Chen, and X. Tai. A hierarchical clustering algorithm based on fuzzy graph connectedness. *Fuzzy Sets and Systems*, 157(13):1760–1774, 2006.

[22] A. Fender, N. Emad, S. Petiton, J. Eaton, and M. Naumov. Parallel Jaccard and related graph clustering techniques. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, page 4. ACM, 2017.

[23] R. Ferdous et al. An efficient k-means algorithm integrated with Jaccard distance measure for document clustering. In *2009 First Asian Himalayas International Conference on Internet*, pages 1–6. IEEE, 2009.

[24] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç. BELLA: Berkeley efficient long-read to long-read aligner and overlapper. *bioRxiv*, page 464420, 2019.

[25] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[26] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.

[27] T. Hoefler and D. Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations*, 1(2):58–75, 2014.

[28] X. Hu, K. Yi, and Y. Tao. Output-optimal massively parallel algorithms for similarity joins. *ACM Transactions on Database Systems (TODS)*, 44(2):6, 2019.

[29] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[30] R. A. Jarvis and E. A. Patrick. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on computers*, 100(11):1025–1034, 1973.

[31] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[32] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, and K. Yelick. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 842–853. IEEE, 2016.

[33] P. M. Kogge. Jaccard coefficients as a potential graph benchmark. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 921–928. IEEE, 2016.

[34] S. Kosub. A note on the triangle inequality for the Jaccard distance. *Pattern Recognition Letters*, 120:36–38, 2019.

[35] V. Kotu and B. Deshpande. *Data Science: Concepts and Practice*. Morgan Kaufmann, 2018.

[36] G. P. Krawezik, P. M. Kogge, T. J. Dysart, S. K. Kuntz, and J. O. McMahon. Implementing the Jaccard index on the migratory memory-side processing Emu architecture. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[37] G. Kucherov. Evolution of biosequence search algorithms: a brief survey. *Bioinformatics*, 35(19):3547–3552, 2019.

[38] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234(5323):34, 1971.

[39] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[40] T. Morzy, M. Wojciechowski, and M. Zakrzewicz. Scalable hierarchical clustering method for sequences of categorical values. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 282–293. Springer, 2001.

[41] R. Moulton and Y. Jiang. Maximally consistent sampling and the Jaccard index of probability distributions. *arXiv preprint arXiv:1809.04052*, 2018.

[42] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome biology*, 17(1):132, 2016.

[43] V. Popic, V. Kuleshov, M. Snyder, and S. Batzoglou. Fast metagenomic binning via hashing and bayesian clustering. *Journal of Computational Biology*, 25(7):677–688, 2018.

[44] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 658–666, 2019.

[45] V. Sachdeva, D. M. Freimuth, and C. Mueller. Evaluating the Jaccard-Tanimoto index on multi-core architectures. In *International Conference on Computational Science*, pages 944–953. Springer, 2009.

[46] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–425, 1987.

[47] S. E. Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[48] J. Scripps and C. Trefftz. Parallelizing an algorithm to find communities using the Jaccard metric. In *2015 IEEE International Conference on Electro/Information Technology (EIT)*, pages 370–372. IEEE, 2015.

[49] D. Selivanov and Q. Wang. text2vec: Modern text mining framework for r. *Computer software manual (R package version 0.4. 0)*, 2016.

[50] S. Seth, N. Välimäki, S. Kaski, and A. Honkela. Exploration and retrieval of whole-metagenome sequencing samples. *Bioinformatics*, 30(17):2471–2479, 2014.

[51] D. B. Skillicorn, J. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[52] B. Solomon and C. Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 34(3):300, 2016.

[53] E. Solomonik, M. Besta, F. Vella, and T. Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 47:1–47:14, 2017.

[54] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 90–109. 2011.

[55] E. Solomonik and T. Hoefler. Sparse tensor algebra as a parallel programming model. *arXiv preprint arXiv:1512.00066*, 2015.

[56] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[57] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLoS Biology*, 2015.

[58] A. Strehl, J. Ghosh, and R. Mooney. Impact of similarity measures on web-page clustering. In *Workshop on artificial intelligence for web search (AAAI 2000)*, volume 58, page 64, 2000.

[59] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar. Introduction to data mining, 2018.

[60] S. Theodoridis and K. Koutroumbas. Pattern recognition. 2003. *Elsevier Inc*, 2009.

[61] E. Valari and A. N. Papadopoulos. Continuous similarity computation over streaming graphs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 638–653. Springer, 2013.

[62] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[63] R. A. Van De Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[64] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[65] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[66] D. E. Wood and S. L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):R46, 2014.

[67] Z. Wu. *Service Computing: Concept, Method and Technology*. Academic Press, 2014.

[68] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[69] K. Zhou, T. P. Michalak, M. Waniek, T. Rahwan, and Y. Vorobeychik. Attacking similarity-based link prediction in social networks. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 305–313. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[70] A. Zielezinski, S. Vinga, J. Almeida, and W. M. Karlowski. Alignment-free sequence comparison: benefits, applications, and tools. *Genome biology*, 18(1):186, 2017.