Asynchronous Distributed Actor-based Approach to Jaccard Similarity for Genome Comparisons

Youssef Elmougy
Georgia Institute of Technology
Atlanta, GA USA
yelmougy3@gatech.edu

Akihiro Hayashi
Georgia Institute of Technology
Atlanta, GA USA
ahayashi@gatech.edu

Vivek Sarkar
Georgia Institute of Technology
Atlanta, GA USA
vsarkar@gatech.edu

***Abstract*—The computation of genome similarity is important in computational biology applications, and is assessed by calculating the Jaccard similarity of DNA sequencing sets. However, it's challenging to find solutions that can compute Jaccard similarity with the efficiency and scalability needed to fully utilize capabilities of modern HPC hardware. We introduce a novel approach for computing Jaccard similarity for genome comparisons, founded on an actor-based programming model. Our algorithm takes advantage of fine-grained asynchronous computations, distributed/shared memory, and the Fine-grained Asynchronous Bulk-Synchronous Parallelism execution model. Our performance results on the NERSC Perlmutter supercomputer demonstrate that this approach scales to $16,384$ cores, showing an average of $4.94\times$ and $5.5\times$ improvement in execution time at the largest scale and relevant hardware performance monitors at medium scale compared to a state-of-the-art baseline. Our approach is also able to process much larger scale genomic datasets than this baseline. We make our source code publicly available[1].**

***Index Terms*—Distributed Jaccard Similarity, Distributed Jaccard Distance, High-Performance Genome Processing, Actors, Selectors, Shared Memory, Distributed Systems, PGAS, OpenSHMEM**

I. Introduction

The computation of genome similarity and genetic distances is highly important in different applications of computational biology and comparative genomics, such as in assemblers [1], [2], metagenomic profiling [3], [4], clustering [5], [6], and retrieval of sequencing samples [7]. Specifically, computing the genome similarity among two DNA sequencing data sets is popularly assessed by computing their Jaccard similarity [8], [9]. The Jaccard similarity, $\mathcal{J}(X,Y)$, computes the overlap of two data sets X and Y by taking the ratio between the cardinality of their intersection by the cardinality of their union ($\frac{|X\cap Y|}{|X\cup Y|}$). In particular for genome similarity, each data set is a multiset of substrings of k DNA bases, referred to as k-mers, where the Jaccard similarity calculates the similarity among multisets with respect to the fraction of k-mers shared between them. Similarly, the Jaccard distance, $d_{\mathcal{J}} = 1 - \mathcal{J}$, is used in assessing genetic distance by calculating the dissimilarity between multisets.

It's important to note that due to their simple and intuitive conceptual characteristics, Jaccard similarity and distance have been employed extensively to a wide range of fields across several application types. The use cases include computational biology [10], [5], [11], pattern recognition [12], data mining [13], [14], clustering [15], [16], [17], recommendation systems [18], and machine learning [19], [20].

Although Jaccard similarity and distance have seen wide adoption in numerous applications, current solutions are not necessarily efficient for scalable and distributed computation. Prior approaches focus on sequential or single server solutions [21], [22], [23], or use inefficient MapReduce implementations [24], [25]. Particularly in the computational biology and comparative genomics field, there have been recently published solutions that attempt to tackle some of these issues [5], [10], [11]. However, due to the continuous growth in the size of sequencing datasets and the increasing number of genomes, the computation of genome similarity and genetic distances is inherently a computationally challenging problem, which motivates the need for an efficient, fast, and scalable solution that can process datasets of growing magnitudes in different domains.

This paper has three main contributions:

- **We design a distributed, scalable, and asynchronous algorithm for computing both $\mathcal{J}$ and $d_{\mathcal{J}}$**: The implementation is based on the Actor-based programming system introduced in [26], taking advantage of its fine-grained asynchronous execution, automatic message aggregation, distributed and shared memory approach, and the underlying Fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP) execution model. Our approach is the first solution to result in just two computation supersteps, reducing the need for global barrier synchronizations.
- **We apply our algorithm to high-scale computations of Jaccard similarity for genome comparisons and genetic distances and achieve $4.94\times$ performance (on a synthetic large-scale $= 20$ dataset) compared to the state-of-the-art algorithm**: This extends the use of these algorithms within the computational biology field as well as others previously mentioned. To the best of our knowledge, [10] represents the current state-of-the-art (SOTA) algorithm of Jaccard similarity for genome comparisons; thus, our work bases its evaluation on comparisons with this SOTA baseline. We present weak scaling and strong scaling results, scaling up to $16,384$ cores of the Perlmutter supercomputer at the National

[1]https://github.com/youssefelmougy/jaccard-selector/

Energy Research Scientific Computing Center (NERSC), showing an average of $4.94\times$ increased performance compared to the SOTA on both synthetic and real world data. Moreover, we are able to run much larger dataset sizes than the SOTA without memory or performance issues.

- **We perform a deep dive hardware performance counter study to realize the performance benefits of our approach**: We present execution time and HardWare Performance Counter (HWPC) results on a medium-scale ($= 14$) synthetic dataset, showing an average of $3.6\times$ and $5.5\times$ increased performance for execution time and HWPC results compared to the SOTA. Our evaluations are based on analysis of the load imbalance, L1/L2 caches, TLBs, branches, network messages, and batch sizes.

II. Background

This section defines Jaccard similarity, including its calculation in graph applications and genome similarity, provides an overview of the SOTA algorithm, and discusses the Actor-based programming system used to build the algorithms.

A. Jaccard Similarity

The Jaccard similarity, $\mathcal{J}(X,Y)$, is an operation that computes the overlap of two data sets X and Y by calculating the ratio between the cardinality of their set intersection and the cardinality of their set union:

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

Similarly, the Jaccard distance, $d_{\mathcal{J}}$, calculates the dissimilarity between sets and is calculated by $d_{\mathcal{J}}(X,Y) = 1 - \mathcal{J}(X,Y)$. If both sets are empty or contain the exact same elements, then $\mathcal{J}(X,Y) = 1$, consequently their distance would be $d_{\mathcal{J}} = 0$. These statistics can be applied to both graph-based similarly and genome-based similarity (the scope of this paper).

In graph-based similarity, the Jaccard similarity represents the similarity of the neighborhoods of two vertices connected by an edge. Specifically, for any two vertices u, v in a graph G, the Jaccard similarity $\mathcal{J}_{u,v}$ is calculated by looking at the union and intersection of the neighborhoods U (for vertex u) and V (for vertex v):

$$\mathcal{J}_{u,v} = \frac{|U \cap V|}{|U \cup V|} = \frac{\gamma_{u,v}}{d_u + d_v - \gamma_{u,v}}\ ,\ \ \forall (u,v) \in G$$

Where $\gamma_{u,v}$ is the number of common neighbors between vertices u and v, d_u is the out-degree of vertex u, and d_v is the out-degree of vertex v.

In genome-based similarity, each data set is a multiset of substrings of k DNA bases, referred to as k-mers. The Jaccard similarity represents the similarity between all pairs of multisets with respect to the fraction of k-mers shared between them. For a set of samples $X = \{X_1, ..., X_n\}$ (which may be from multiple genomes), where a sample X_i comprises of m positive integers (where m can range to $m = 4^{30}$), the Jaccard similarity for each pair of multisets is calculated by:

$$\mathcal{J}(X_u, X_v) = \frac{|X_u \cap X_v|}{|X_u \cup X_v|} = \frac{\text{common k-mers}}{\text{total present k-mers}}\ , {}_{u,v \in \{1,...,n\}}$$

B. State-of-the-Art Similarity Algorithm

The SOTA Jaccard similarity algorithm for genome comparisons, GenomeAtScale, is introduced in [10]. It leverages the distributed memory numerical library Cyclops Tensor Framework (CTF) [27]. CTF provides routines that allow for contraction and summation of sparse and dense tensors, provides primitives for sparse input and transformation of data, and enables element-wise operations on distributed data types.

Before discussing the GenomeAtScale algorithm, it is important to realize the algebraic formulation of the Jaccard similarity. An indicator matrix $\mathbf{A}^{m \times n}$ is defined, where $a_{ij} = 1$ if $i \in X_j$ or otherwise $a_{ij} = 0$. The similarity matrix $\mathbf{S} \in \mathbb{R}^{n \times n}$, which is the Jaccard similarity $\mathcal{J}$ of all pairs of samples, is calculated by $s_{ij} = \mathcal{J}(X_i, X_j) = b_{ij}/c_{ij}$, where $\mathbf{B}, \mathbf{C} \in \mathbb{N}^{n \times n}$ are the cardinalities of the intersections and unions of pairs of samples respectively calculated by $b_{ij} = \sum_k a_{ki} a_{kj}$ and $c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$. The GenomeAtScale algorithm then calculates $\mathbf{S}$ using the following five steps: (1) divide $\mathbf{A}$'s rows into batches with $\hat{m}$ rows and loop through each batch; (2) remove zero rows within the batch using a distributed sparse vector; (3) compress row segments with bitmasking; (4) compute and accumulate partial scores into $\mathbf{B}$ and $\mathbf{C}$ matrices; (5) derive the final similarity scores $\mathbf{S}$ based on intermediate $\mathbf{B}$ and $\mathbf{C}$ matrices. The reader is directed to [10] for more detailed algorithmic descriptions.

C. Actor-Based Programming System

The algorithmic contributions of this paper are built using the FA-BSP model introduced in [26], and its accompanying C++ runtime implementation in [28]. FA-BSP is a productive, lightweight, and asynchronous computation model suitable for high-throughput DNA sequencing data computations. It provides fine-grained asynchronous point-to-point actor messages, automatic message aggregation (through the Conveyors library [29]), distributed and shared memory approach, and an underlying FA-BSP execution model. These actors are regarded as computation primitives, sharing no mutable state due to inherent isolation. It as well avoids data races and synchronization by executing messages sequentially within its mailbox, avoiding concurrent contention to local data accesses. The terms "Actor" and "Selector", the latter of which is "Actors with multiple mailboxes" [30], will be used interchangeably for the remainder of the paper. It's important to note that a unit of computation, which is a processing element (PE) or an Actor, has a one-to-one relation to a physical CPU core, where each PE/Actor is spawned and scheduled on a dedicated core to work in parallel with other PEs/Actors and communicate using asynchronous message schemes. The reader is directed to [26], [31] for more detailed system descriptions and evaluation on graph-based algorithms

and to [32] for evaluation of massive resource and dataset scalability.

III. Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

It's important to note that Jaccard similarity and distance can be employed across several application types. Although we focus on genome comparisons, the input data samples forming the indicator matrix can be uniquely identified to adapt to different computational problems. For example, rows and columns of the matrix can be formed from vertex neighborhoods to calculate the similarity across vertices in a graph; a matrix formed from rows of vertices and columns of clusters enables calculation of similarity of clusters; and, a matrix formed from rows of k-mers and columns of data samples enables the calculation of similarity of genomes. Moreover, to avoid redundant computation and reduce memory utilization, our application computes on and stores a lower triangular matrix (where $n = m$ to create an $m \times m$ matrix, reducing space from m^2 elements to $\frac{m^2}{2} - m$ elements and avoiding a factor of $O(m^2)$ communication) depending on the use case. Such cases include similarity across vertices in a graph, where a lower triangular matrix will store the undirected graph edges for computation. Our approach is the first solution to $\mathcal{J}$ and $d_{\mathcal{J}}$ that employs a novel distributed and asynchronous algorithm resulting in just two computation supersteps compared to other synchronous BSP-based approaches utilizing a large number of barriers. This section discusses the distributed algorithm for computing both $\mathcal{J}$ and $d_{\mathcal{J}}$ (Section III-A) and its application to genome comparisons (Section III-B).

A. General Jaccard Measures

This algorithm calculates $\mathcal{J}$ and $d_{\mathcal{J}}$ using a distributed and asynchronous approach based on definitions in Section II-A. The $\mathbf{A}^{m \times n}$ matrix ($m \geq n$) is highly sparse, depending on the application domain. We use the CSR format to efficiently store nonzeros in a sparse matrix representation, thereby avoiding the memory overhead of storing zero values/rows. The high-level C++ code for the core computation step per Actor in our algorithm is shown in Listing 1. It follows the FA-BSP model in [26], [31] to evaluate $\mathcal{J}$ using the following steps:

- J1 Iterate through nonzero entries in locally stored rows. **Lines: 20 - 24.**
- J2 Calculate the intersection of local elements and each nonzero within their data samples. An asynchronous message (containing the nonzero value of interest) is sent in Line 34 to the owner PE to access its (possibly remote) data sample and check for element-wise similarity using binary search. If an intersection is found, the counter will be incremented[2]. **Lines: 30 - 40.**
- J3 Calculate the final $\mathcal{J}$ values using the graph-based similarity approach discussed in Section II-A. This is a fully local operation that is performed after the barrier operation in Line 44 ensures that all asynchronous messages have been fully processed in accordance with the FA-BSP model (the `done` call in Line 43 ensures that all messages are processed before all PEs can proceed through the barrier). **Lines: 47 - 53.**

[2]For readability, `intersection` is abstracted in Listing 1 while it has the same sparse format as matrix `A` in the actual implementation.

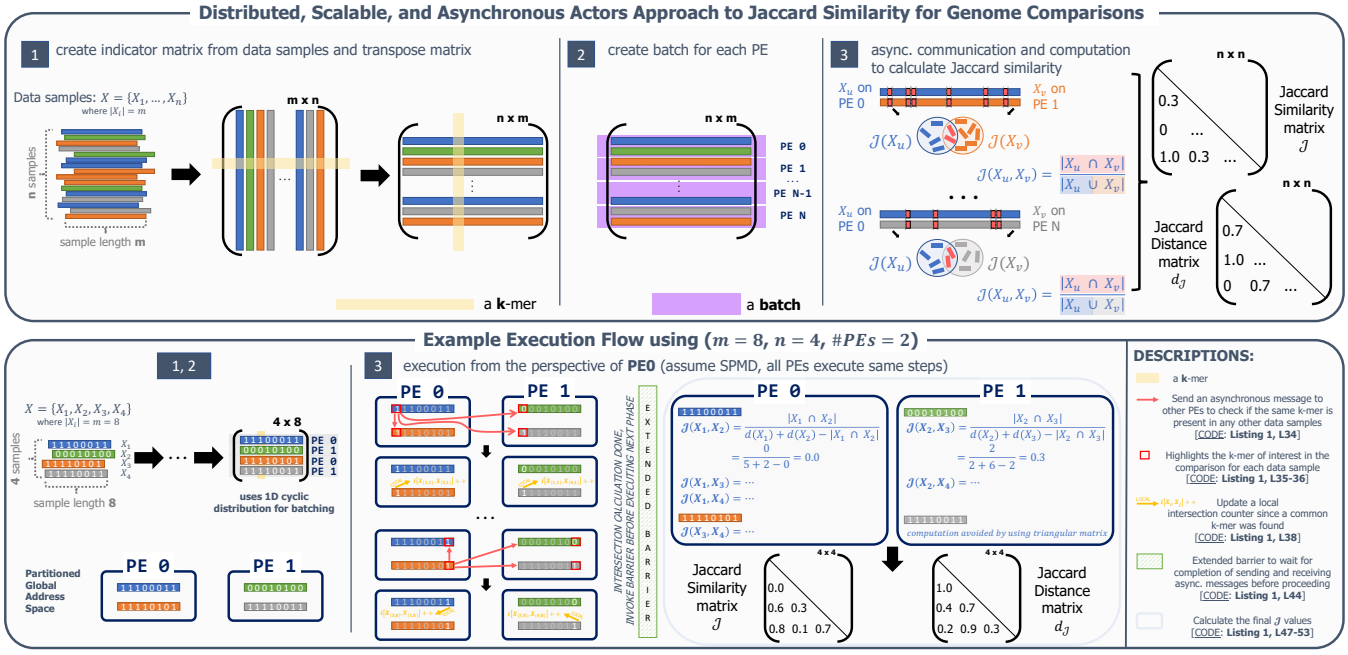
Listing 1: (SPMD) Actor-based Jaccard Similarity algorithm.

```
1  /* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
          in A->offset and nonzeros stored in A->nonzero
2   * Output: (shared) (n x n) matrix J with jaccard similarities */
3
4  /* Utility functions
5     * my_pe() : returns PE number of calling PE
6     * n_pes() : returns number of PEs running in the application
7     * get_remote_pe(x) : returns owner PE for element x
8     * get_local_index(x) : returns local index in shared array for element x
9     * get_global_id(x) : returns global ID for element x
10    * binary_search(arr[:], x) : returns array index for element x if found
11    * barrier() : blocks until async local/remote ops are completed on all PEs
12    * d(x) : returns degree of element x (number of nonzeros)
13 */
14
15 // initialization and start of the Selector instance
16 Selector<1> jac_selector;
17
18 int sender_PE = my_pe();
19 int n_local_rows = n / n_pes();
20 for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
21   int v_global = get_global_id(v); // get global ID for v (local operation)
22   // get offset indices for the row
23   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
24     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
25     /* calculate intersection, the loop in L30 & L49 is application-dependant:
26        * for general jaccard measures, the nonzeros of v are looped through
27            for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
28        * for genome-based similarity, all pairs of samples are looped through
29            for(int next_sample = v_id; next_sample < n; next_sample++){...} */
30     for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
31       int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
32       int remote_PE = get_remote_pe(u);
33       // asynchronous msg sent to loop through shared array on remote_PE
34       jac_selector.send(0, remote_PE, [=]() {
35         if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
36                           A->offset[get_local_index(u)+1]], v_nonzero))
37           // found common element, update local counter
38           intersection[get_local_index(v_global), get_local_index(u)]++;
39       });
40 } } }
41
42 // automatic termination of the Selector instance
43 jac_selector.done(0); // the mailbox will be terminated through the runtime's
       automatic termination protocol after guaranteeing all its messages have
       been received and executed
44 barrier(); // FA-BSP model, ensure all async messages have been executed before
       all PEs move forward
45
46 // calculate jaccard similarity J
47 for (int v = 0; v < n_local_rows; v++) {
48   // get offset indices for the row
49   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
50     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
51     // J = intersection / (d(x) + d(y) - intersection)
52     J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
53 } }
```

B. Jaccard Similarity for Genome Comparisons

The distributed algorithm discussed in the previous section is adapted into the field of computational biology and comparative genomics to compute the Jaccard similarity for genome comparisons and genetic distances. Figure 1 shows a high-level description of the algorithm and an example execution flow applying a matrix with $m = 8$, $n = 4$, and $\#\text{PEs} = 2$.

Fig. 1: A high-level overview of the Jaccard similarity algorithm adapted for genome comparisons (top) and an example execution flow using $m = 8$, $n = 4$, and $\#PEs = 2$ (bottom). The data samples are distributed across PEs then 3 steps are performed (matrix creation and transpose, batching, computation) to calculate $\mathcal{J}$ and $d_{\mathcal{J}}$.



The goal of the example execution flow is to assist the reader in understanding the distributed and asynchronous nature of the algorithm as well as the intricacy of the Jaccard similarity calculations. The algorithm follows two preprocessing steps (*step 1, step 2*) followed by one computation step[3] (*step 3*). The preprocessing steps are performed to enhance runtime efficiency in the computation step: (1) *step 1* creates an indicator matrix from the data samples and transposes this matrix since the actor runtime that we use [28] stores sparse matrices in a row-major CSR format. Given that computation is with respect to data samples, storing these data samples row-wise guarantees that traversal is always a local operation, hence avoiding the cost of remote communication; (2) *step 2* creates batches for each PE, given the transposed matrix as input. This allows a close-to equal distribution of data elements across PEs. However, if some processors are more specialized or have a larger per-processor-memory, then batches allow for a heavier load to be placed on the more specialized processors by adjusting the batch size for a particular set. The computation step (*step 3*) is adapted from the general Jaccard measures algorithm to allow for similarity with respect to pairs of data samples while evaluating J2. Lines 25 - 29 in Listing 1 explain the necessary changes to apply it to genome comparisons and also provides a sample code (Line 29).

IV. Evaluation

This section describes the experimental setup and architecture for the testing environment and discusses the performance results collected.

[3]Only the computation step was shown in Listing 1.

A. Experimental Setup and Architecture

We use the CPU nodes of the Perlmutter supercomputer at NERSC, which is an HPE Cray EX supercomputer. Each node has 2 AMD EPYC 7763 (Milan) CPUs with 64 physical cores per CPU, 512 GB of DDR4 memory, and 1 network card connected to the HPE Cray Slingshot 11 network. There is 32KB of L1 data cache, 32KB of L1 instruction cache, 512KB of L2 cache, and 2560 4K pages of TLB per physical CPU core. We use 32 PEs per node as our configuration and 32 MPI processes per node for the GenomeAtScale algorithm to be consistent with the configuration used in their paper [10]. To show the performance and scalability of our approach, we perform weak scaling experiments using a synthetic dataset of scale[4] = 12 to 20 (medium-scale to large-scale) and strong scaling experiments using a synthetic dataset of much larger scale than the SOTA (scale = 25) and two real world datasets (*E. coli genome*[5] with 2M sequences and *SARS CoV2 genome*[6] with 10K sequences, both with sequence reads of length 150). We ran our experiments 5 times, taking their average for each data point. The reader is directed to our GitHub repository for more information on the software stack and setup.

B. Performance Analysis

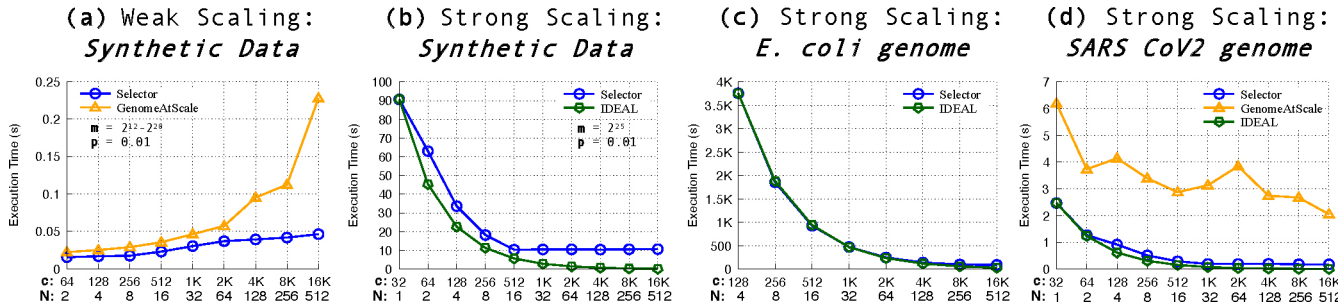
Figure 2 displays the performance and scalability of our proposed approach compared to the SOTA (GenomeAtScale)

[4]`scale` refers to the number of data sample rows in the dataset as a power of two, popularly used in Graph500 and Genomics Communities (Reference scales - `small-scale`: 2^{10}, `medium-scale`: 2^{14}, `large-scale`: 2^{20}, `extra-large-scale`: 2^{25}).

[5]https://www.ncbi.nlm.nih.gov/nuccore/NZ_CP027599.1?report=fasta

[6]https://www.ncbi.nlm.nih.gov/nuccore/NC_045512.2?report=fasta

Fig. 2: Performance analysis of our proposed approach: (a) weak scaling using synthetic datasets of scale = 12 to 20, (b) strong scaling using a synthetic dataset of scale = 25, (c) strong scaling using the *E. coli* genome ($2M$ sequences, $k = 7$, 150 bp), and (d) strong scaling using the *SARS CoV2* genome ($10K$ sequences, $k = 5$, 150 bp).

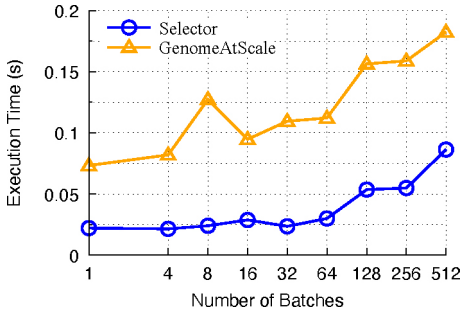


using large-scale synthetic datasets and large-scale real genomic datasets. The overhead due to startup, synchronization, and IO is negligible but included (less than 1% of the execution time for small-medium datasets).

Scalability Performance on Synthetic Data - Figure 2(a) displays the weak scaling (scale = 12 to 20) and Figure 2(b) displays the strong scaling (scale = 25) performance of our proposed approach. In the case of weak scaling, we observe good scaling with only an increase of $2.7\times$ in execution time when we increase the resources by $256\times$ and dataset by $1024\times$; in contrast, the SOTA incurs an increase of $10.5\times$ in execution time with the same increases. We observe a spike phenomena starting from $2K$ cores for the SOTA, whereas our proposed approach scales well until $16K$ cores, outperforming the SOTA approach by $4.94\times$ at the largest scale. In the case of strong scaling, *the SOTA approach is unable to process datasets of large scale without memory issues*, using a minimum of 512 nodes ($16K$ cores) to process the scale = 20 dataset while *our approach is able to process and evaluate a much higher scale dataset (scale = 25) using only a minimum of* 1 *node (*32 *cores)*. We observe good scaling in this case, executing the scale = 25 dataset in 91 seconds using 1 node, and achieving a speedup of $10.1\times$ until the point of plateau (512 cores, $16\times$ resource increase), whereas comparable approaches are unable to run this dataset size.

Scalability Performance on Real Data - Figure 2(c) displays the performance analysis of our approach on the *E. coli genome* ($2M$ seq.) and Figure 2(d) on the *SARS CoV2 genome* ($10K$ seq.). It is important to note that these two datasets are utilized due to their different data sparsities and variability, as well as to display the real world application of our proposed approach to the field of genomics. It can be seen that our algorithm undergoes excellent scaling for both genomes, even up to $16K$ cores (there is a stagnant phenomena for the *SARS CoV2 genome* from 512 cores, as the number of cores becomes greater than the number of columns of the indicator matrix, though the performance does not degrade thereafter). At the point of plateau for the *SARS CoV2 genome* (512 cores), we still achieve a $9.8\times$ speed up relative to the smallest node count and a $9.9\times$ speed up compared to the SOTA. At the highest node count, we achieve a $42.8\times$ and $13.8\times$ speed up relative to the smallest node count respectively for both genomes (and $11.4\times$ better performance compared to the SOTA for the *SARS CoV2 genome*). In fact, we are able to run the *E. coli genome* in 3759 seconds using only 4 nodes (128 cores), scaling to less than 90 seconds at the largest scale. *Notably, the SOTA approach is unable to process the large E. coli genome without memory issues.*

Fig. 3: Sensitivity to batches on algorithm performance.



Sensitivity to Batches - Figure 3 shows the effect on performance given the number of batches per core. Batches allow for variable load allocation across processors. For instance, during initialization and distribution, the user can assign larger data batches to specific processors with greater memory. Although, one must be careful with choosing the batch size since it has a direct correlation with the amount of communication and hence execution time. It can be seen that as the number of batches increases, the execution time increases with direct proportion (excluding obvious outliers). This is attributed to the fact that larger number of batches mean that per-core slices of the computation matrix will be lower, inducing the need for increased remote communication for continued computation. Given that increased communication has a direct effect on the number of remote atomics and collective operations, the execution time will in-turn be affected. Nevertheless, our proposed algorithm still scales efficiently as well as performs better than the SOTA by $3.6\times$ attributing to its asynchronous execution and underlying FA-BSP model. We note that the optimal batch size is a perfectly even distribution of data samples across all workers, hence n/N, in turn increasing per-core slices while reducing communication.

TABLE I: Hardware performance counters and descriptions.

	Abr.	Counter	Description
L1 data cache	L1DA	PAPI_L1_DCA	L1 data cache accesses
	L1DM	PAPI_L1_DCM	L1 data cache misses
L1 instruction cache	L1IA	perf::PERF_COUNT_HW_CACHE_L1I:ACCESS	L1 instruction cache accesses
	L1IM	perf::PERF_COUNT_HW_CACHE_L1I:MISS	L1 instruction cache misses
L2 data cache	L2DR	PAPI_L2_DCR	L2 data cache reads
	L2DM	PAPI_L2_DCM	L2 data cache misses
L2 instruction cache	L2IR	PAPI_L2_ICR	L2 instruction cache reads
	L2IM	PAPI_L2_ICM	L2 instruction cache misses
TLBs	TLBDM	PAPI_TLB_DM	Data translation lookaside buffer misses
	TLBIM	PAPI_TLB_IM	Instruction translation lookaside buffer misses
Branches	BRINS	PAPI_BR_INS	Branch instructions
	BRMSP	PAPI_BR_MSP	Conditional branch instructions mispredicted
Network Messages	MSGCNT	MSG_COUNT	Number of point-to-point communication across PEs
	MSGBYT	MSG_BYTES	Size (bytes) of point-to-point communication across PEs
Cycles	CYC	PAPI_TOT_CYC	Total cycles
Instructions	INS	PAPI_TOT_INS	Instructions completed

V. Deeper Dive into HWPC: Realizing the Performance Benefits

Our proposed approach achieves strong performance results in the weak and strong scaling experiments on the synthetic and real datasets compared to the SOTA, hence it is important to realize the performance and understand the sources of this benefits. To this front, we do a deeper dive into HardWare Performance Counters (HWPC) values, communication messages and sizes, and execution times using strong scaling experiments of a synthetic dataset of medium-scale (scale $= 14$). Given the large number of data points needed for our analysis, it was not feasible to perform our evaluation on larger scale values due to resource allocation and availability limitations on the Perlmutter system. This section states the HWPC used to evaluate the proposed and SOTA algorithms and discusses the performance results collected.

A. Hardware Metrics

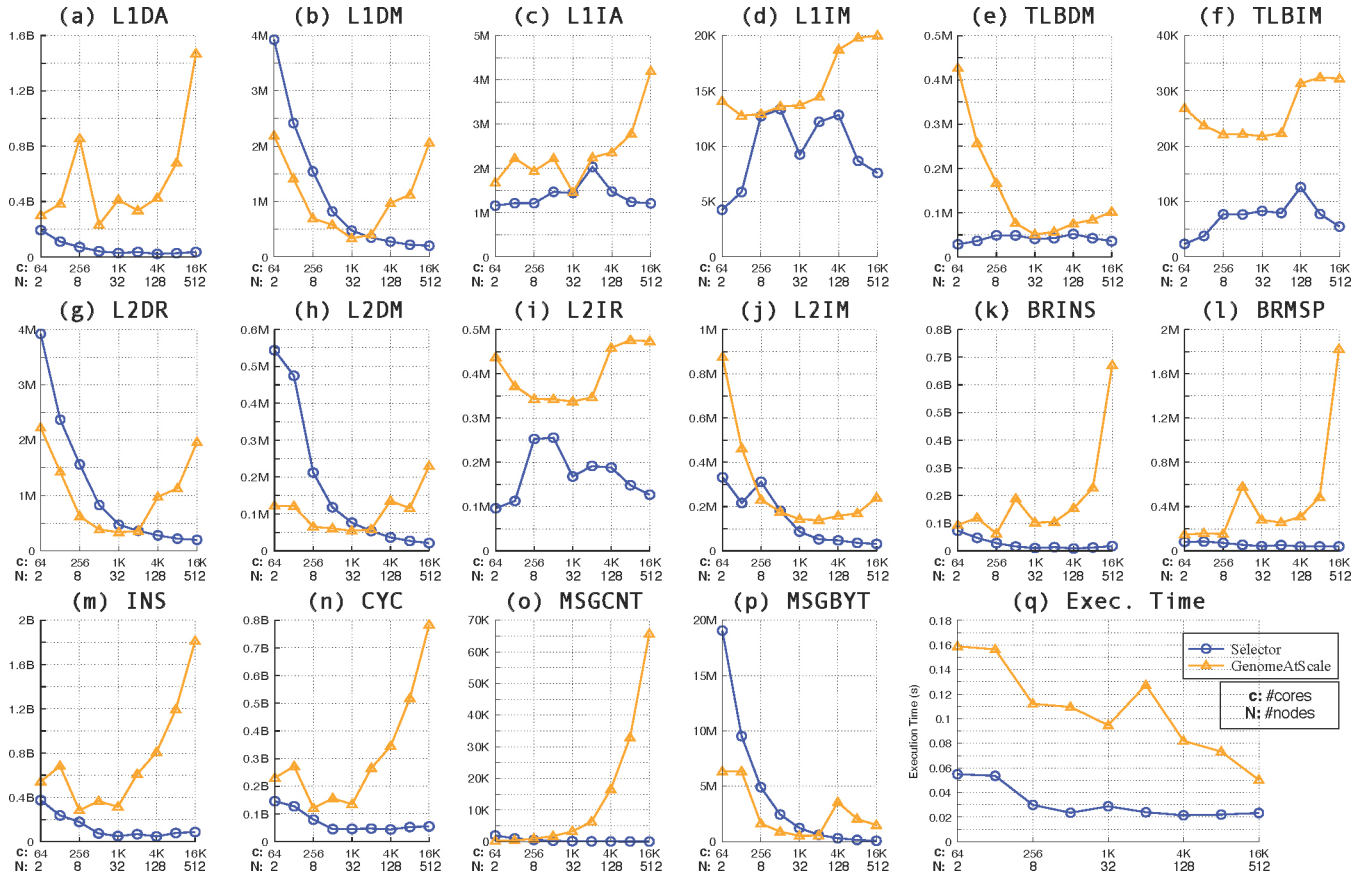
Through the analysis of hardware counters, we aim to display and explain the significant performance benefits of the algorithmic choices undertaken by our approach. We collect a total of sixteen performance counters, ranging from analysis of L1/L2 data/instruction caches, data/instruction TLBs, branches, network messages, cycles, and instructions. Memory-bound applications, such as Jaccard similarity, emphasize high importance to cache and TLB misses [33] and to memory bandwidth. Branches are studied to analyze the frequency of mispredictions [34], [35]. It's also important to study instruction cache performance since effective instruction cache is key to support high Instruction-Per-Cycle per core and high I-fetch bandwidth [36]. Network congestion and bandwidth are important since they capture the impact of remote atomics and collective operations. Table I shows the HWPC measurements collected and their descriptions. We use the `PAPI` library (v7.0.0.1) and the `perf` tool (v5.14.21) as part of the CrayPat performance analysis tool [37] to access these performance counters. For graphical analysis and visualizations, such as those for load imbalance and call graphs, we use Cray Apprentice 2 [37].

B. Performance Analysis

Figure 4 shows visually the strong-scaling performance results of the proposed Actor-based approach as compared to the SOTA (GenomeAtScale). The reader is referred to Table II for the numerical value of each data point in Figure 4 for easier comprehension. We investigate the performance based on analysis of the execution time, load imbalance, L1/L2 caches, TLBs, branches, network messages, and batch sizes, and discuss each in the order of their importance and impact.

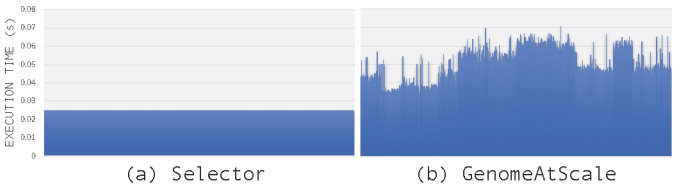
Execution Time - [Figure 4 (q)] We can see that our algorithm achieves significantly lower execution times compared to the SOTA for all node counts. This is largely attributed to the exploitation of asynchronous communication/computation and actor-level parallelism, as well as the benefits from the forthcoming memory-bound discussions. Besides reduction in execution time, the total instructions (Figure 4 (m)) and cycles (Figure 4 (n)) for program execution are as well significantly reduced ($5.5\times$ - $8.9\times$). However, we also note that for this medium-scale input size, the execution time plateaus after 1024 cores with strong scaling, but does not degrade (not experienced when running larger scale datasets). This occurs because the number of cores begins to surpass the number of rows. The optimizations in Section VI are proposed to help reduce the impact of the bottlenecks contributing to this plateauing phenomenon.

Fig. 4: Strong scaling (scale = 14) HWPC, network statistics, and execution time performance results of the proposed Jaccard similarity algorithm for genome comparisons (blue) compared to the SOTA (orange). All numbers are per-PE averages.



Load Imbalance - Figure 5 displays the workload per core, where the x-axis is the cores (512 core experiment) and the y-axis is the execution time. It can be seen that our algorithm reduces the total workload and achieves a near-perfect equal distribution of load per core, as compared to the SOTA where the workload is much higher with a heavy imbalance per core. This is clearly reflected with the Relative Load Imbalance (RLI) statistic [38], measuring the relative difference in workload among processing units, where our approach outperforms the SOTA by approximately $592\times$ in terms of RLI. Refer to the following section to understand the effect of collectives on the load imbalance.

Fig. 5: Workload per core compared to the SOTA. Numbers collected using CrayPat for 512 cores.



Network Messages - [Figure 4 (o-p)] Remote atomics and collective operations induce heavy effects on network congestion and bandwidth, hence the importance of their reduction. Our algorithm is observed to reduce both message counts and bytes as the number of nodes is scaled, with message counts observing super-scalar speedups. The increased performance can be attributed to two factors: (1) our algorithm executes in only two computation phases (intersection calculation, union + similarity calculation) and takes advantage of the FA-BSP execution model [26], reducing the total number of collective calls to 2 per application run. Conversely, the SOTA algorithm is profiled to utilize an average of 98% of network messages in `MPI_Alltoall()` calls and invoke synchronization barriers at each collective call (largely due to topology changes and data re-distribution), with an increasing number as the nodes are scaled. Moreover, our algorithm has scalable performance, though the SOTA sees spikes in performance due to calls to collective variants (`MPI_Bcast`, `MPI_reduce`); (2) our algorithm takes advantage of fine-grained message aggregation from the Actor-based system, allowing a significant reduction to network traffic and message counts. Though the aggregation increases the total bytes per network message, an optimal spot is achieved at $\approx 2K$ cores. It's important to note that while 95% of the messages sent in the SOTA are 16B, 100% of

TABLE II: Numerical performance results as displayed in Figure 4. Our algorithm is "Selector" and GenomeAtScale is "SOTA".

Cores:	64 cores		128 cores		256 cores		512 cores		1024 cores		2048 cores		4096 cores		8192 cores		16384 cores	
Event	Selector	SOTA	Selector	SOTA	Selector	SOTA	Selector	SOTA	Selector	SOTA	Selector	SOTA	Selector	SOTA	Selector	SOTA	Selector	SOTA
L1DA	193M	301M	110M	384M	71M	855M	41M	232M	28M	412M	35M	335M	21M	425M	27M	679M	35M	1.4B
L1DM	3.9M	2.2M	2.4M	1.4M	1.6M	0.6M	0.8M	0.4M	0.5M	0.3M	0.4M	0.4M	0.3M	1.0M	0.2M	1.2M	0.2M	2.0M
L1IA	1.2M	1.7M	1.2M	2.2M	1.2M	1.9M	1.5M	2.2M	1.4M	1.5M	2.0M	2.2M	1.5M	2.4M	1.2M	2.8M	1.2M	4.2M
L1IM	4.3K	14.1K	5.9K	12.8K	12.7K	12.9K	13.3K	13.6K	9.2K	13.7K	12.2K	14.5K	12.8K	18.7K	8.7K	19.8K	7.6K	20.0K
L2DR	3.9M	2.2M	2.4M	1.4M	1.5M	0.7M	0.8M	0.6M	0.5M	0.3M	0.3M	0.4M	0.3M	1.0M	0.2M	1.1M	0.2M	2.1M
L2DM	544K	122K	474K	121K	212K	65K	118K	61K	76K	54K	54K	58K	36K	136K	27K	115K	21K	230K
L2IR	96K	437K	113K	372K	253K	343K	256K	342K	168K	337K	192K	347K	188K	458K	148K	476K	127K	473K
L2IM	332K	875K	217K	463K	312K	231K	182K	176K	87K	143K	52K	129K	47K	158K	36K	169K	31K	240K
BRINS	72.8M	93.1M	47.0M	120M	27.8M	61.9M	15.8M	189M	10.1M	102M	13.4M	105M	8.5M	154M	12.2M	228M	17.3M	671M
BRMSP	82K	147K	85K	158K	72K	152K	55K	580K	43K	280K	52K	256K	39K	308K	40K	483K	40K	1.8M
TLBDM	29K	426K	36K	257K	49K	167K	49K	76K	41K	51K	42K	57K	52K	75K	42K	84K	36K	102K
TLBIM	2.4K	26.9K	3.8K	23.8K	7.7K	22.1K	7.7K	22.2K	8.3K	21.8K	7.9K	22.4K	23.6K	31.4K	7.8K	33.4K	5.5K	33.2K
MSGCNT	1.9K	247	1.0K	377	540	927	281	1.7K	140	3.3K	97	6.3K	69	16.6K	35	32.9K	16	65.7K
MSGBYT	19.1M	6.3M	9.5M	6.3M	4.9M	1.6M	2.5M	0.9M	1.2M	0.5M	0.6M	0.5M	0.3M	3.6M	0.2M	2.1M	77K	1.5M
CYC	146M	230M	128M	272M	79M	121M	46M	157M	26M	135M	47M	265M	44M	345M	53M	518M	55M	783M
INS	374M	542M	237M	683M	178M	282M	73.6M	366M	50.6M	315M	66.9M	610M	47.0M	808M	77.9M	1.2B	86.2M	1.8B
Exec. Time	0.05s	0.16s	0.05s	0.16s	0.03s	0.11s	0.02s	0.11s	0.03s	0.09s	0.02s	0.13s	0.02s	0.08s	0.02s	0.07s	0.02s	0.05s

the messages sent in our algorithm are $\approx$ 10KB (= size of our aggregation buffer), improving the network bandwidth. An average performance increase of $595.2\times$ (max: $4065.9\times$) and $5.3\times$ (max: $19.4\times$) is observed for message counts and bytes respectively.

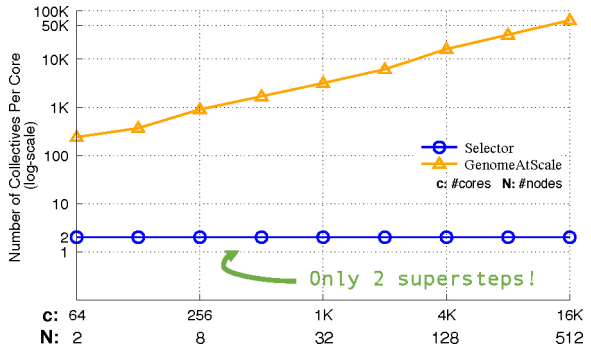
L1,L2 Caches - [Figure 4 (a-d) and (g-j)] For dcache, our proposed algorithm achieves efficient parallel scaling, as seen by the reduction in accesses and misses to both L1 and L2 dcache as a factor of the increasing number of nodes. Moreover, an optimal spot is achieved at $\approx 2K$ cores where henceforth (1) the proposed algorithm outperforms the SOTA and (2) the SOTA shows an increase in the HWPC numbers due to topology changes and data re-distribution. This can be attributed to the data distribution mechanisms (and both preprocessing steps) employed in our algorithm, increasing the possibility of reusing data stored within the same PE. An average performance increase of $8.7\times$ (max: $41.7\times$) and $2.5\times$ (max: $10.9\times$) is observed for L1 and L2 dcache respectively. For icache, the proposed algorithm consistently outperforms the SOTA (with the exception of obvious outliers) on accesses and misses to both L1 and L2 icache. It can be seen that as the number of nodes is increased, the icache misses are either stable or decreased in the proposed algorithm, as compared to an inefficient increasing pattern in the SOTA, which in turn decreases the instruction fetch latency. Specifically, the proposed algorithm efficiently decreases the cycles per instruction (CPI) due to its sensitivity to L2 cache misses. These effects can be attributed to: (1) the reusing of the same remote message handler code across all memory requests; and (2) the simplicity (in terms of binary file code size) of our user code and backend runtime system (as compared to the SOTA CTF-based backend as well as their utilization of several external libraries). An average performance increase of $1.7\times$ (max: $3.5\times$) and $2.8\times$ (max: $7.7\times$) is observed for L1 and L2 icache respectively. In general, our approach takes 98 nodes to fit all the data into the L1 dcache and only 6 nodes to fit all the data in the L1 and L2 dcache. This is evident by the graph crossover with the SOTA as well as the rapid decrease in accesses and misses.

TLB - [Figure 4 (e-f)] TLB misses have a positive correlation with instruction fetch latency/stalls and CPI, hence it's important to decrease their occurrence. On a TLB miss (no TLB entry is present for the virtual page number of the accessed page), the processor has to perform a page walk (lookup on the page table) which is an expensive operation [39]. Our algorithm is observed to consistently reduce latency and page walks at a much higher magnitude as compared to the SOTA for all node counts. This is extremely important in memory-bound applications. Moreover, the memory bandwidth is increased in our algorithm due to the reduced frequency of TLB misses. An average performance increase of $4.0\times$ (max: $15.0\times$) and $4.6\times$ (max: $11.4\times$) is observed for data and instruction TLBs respectively.

Branches - [Figure 4 (k-l)] Besides observing much lower overall branch instructions, our algorithm is seen to achieve efficient parallel scaling as the number of nodes is increased for both branch statistics. On a branch misprediction, an instruction flush and an instruction fetch occurs in the pipeline, incurring cycle overheads which have a dire effect on pipeline performance. Our algorithm is observed to consistently decrease the cycle penalty incurred from mispredictions at a much higher magnitude as compared to the SOTA for all node counts. This can be attributed to the simple (predictable) pattern among the majority of branch instructions, allowing the Branch Target Buffer to predict the branch jump accurately. An average performance increase of $12.4\times$ (max: $38.8\times$) and $10.4\times$ (max: $45.9\times$) is observed for branch instructions and mispredictions respectively.

Performance Benefit from Algorithmic Choices - Figure 6 displays the number of collectives/barriers invoked per-core throughout the execution run on the log-scale. Our proposed approach invokes only two barriers throughout the full execution, which is consistent with our algorithmic choice of a *two superstep execution process*. The results show that as the number of cores and nodes are scaled, our proposed approach consistently only invokes a collective/barrier per superstep (2) while the SOTA approach drastically increases the number of collectives as the application is scaled.

Fig. 6: Per-core number of collectives throughout the execution run (log-scale).



VI. Optimizations and Future Work

In an effort to understand bottlenecks and possible optimizations, as well as engage future work using our proposed distributed Jaccard similarity algorithm for genome comparisons, we explore several possible algorithmic-level and compiler-/runtime-level optimizations and discuss their potential performance benefits.

Row compression using bitmasking - The aim is to reduce the overhead of storing nonzeros by storing a sequence of nonzeros as a binary value which takes only one bit of data compared to a 32-/64-bit integer nonzero. Specifically, an x-bit bitmask is used to encode x elements of each column of the indicator matrix, compressing the number of rows by a factor of x and potentially the number of nonzeros stored [10].

Matrix distribution - The aim is to reduce the need for remote atomics by distributing data in a communication-aware approach. This can be achieved by exploring different data distribution strategies, such as cyclic distribution, XOR distribution, or circular hash distribution, in effect improving communication by reducing the amount of remote atomics needed to access data. Our approach employs the 1D-cyclic distribution which is effective on synthetic and real data, though our algorithm is flexible enough to support different distributions when encountering a load-imbalance problem.

Message buffer size - In the runtime system used to realize the proposed algorithm, a harmony of asynchronous fire-and-forget messages and a capped message buffer exist. During program execution, a PE will execute program code and send asynchronous messages to other PEs (as seen in Listing 1). When the message buffer of a PE is full, a context switch occurs to yield the processing element to flush and execute the messages from the buffer. Therefore, from the analysis done in Sections IV and V, the small amount of messages with a large message size sent across the execution of the application can be attributed to a large message buffer size. Although, there may be more optimal configurations to induce increased memory bandwidth per core.

GPU acceleration - GPUs can be leveraged to accelerate the execution of our fine-grained computations. Although GPUs have not been exploited in SOTA approaches for this application domain, it has benefited many Sparse-Sparse and fine-grained operations, thus, this acceleration can be applied to large-scale datasets to reduce the CPU resources utilized and provide a comparable execution time speed up.

VII. Related Work

There are four well-known approaches to genome and set similarity: Mash, Dashing, BELLA, and GenomeAtScale. Mash [5] is a genome similarity software that extends MinHash sketches to calculate the Jaccard similarity for all pairs of multisets. It allows for genome distance estimation and clustering through Mash distance. Dashing [11] is a software for estimating genome similarities using HyperLogLog sketches and is presented as a faster alternative to Mash. It uses cardinality estimation methods highly specialized for set union and set intersection. BELLA [40] is an overlap detection and alignment algorithm that defines genome similarity calculations using sparse matrix-matrix multiplication. Similarly, GenomeAtScale [10], the SOTA algorithm, is a matrix-matrix multiplication approach to genome similarity that allows for parallel data compression and batch computation.

VIII. Conclusion

Jaccard similarity and distance are highly popular in assessing the genome similarity of DNA sequencing sets in applications of computational biology and comparative genomics, but have also seen adoption in applications of pattern recognition, data mining, clustering, machine learning, and much more. However, current solutions are not suitable or efficient for scalable computation. This paper makes three contributions. The first contribution is a distributed, scalable, and asynchronous algorithm for computing Jaccard similarity and distance for applications assessing general Jaccard measures. This is based on an Actor programming system [26], taking advantage of fine-grained asynchronous execution, distributed/shared memory approach, and the FA-BSP model. We believe this distributed Actors approach to deriving Jaccard similarity and distance can be adapted to applications in other domains as well.

The algorithm for the derivation of Jaccard similarity and distance is then applied to genome comparisons and genetic distances of DNA sequencing sets, extending the use of this algorithm within the computational biology field, as the second and third contributions. This application follows a communication-aware and distributed/shared-memory approach to computation. Thorough analysis on the performance of the proposed algorithm compared to the state-of-the-art algorithm [10] was done using hardware performance metrics and execution times. We scale to $16,384$ cores of NERSC Perlmutter supercomputer, achieving $3.6\times$ and $5.5\times$ increased performance in execution time and hardware counters on a medium-scale synthetic dataset compared to SOTA baseline, and reducing L1/L2 cache accesses/misses, data/instruction TLB misses, branch instructions/mispredictions, and network statistics with efficient parallel scaling. We also show a $4.94\times$ increase in performance on a large-scale synthetic dataset, and we are able to efficiently run much larger scale synthetic and

real datasets without memory issues unlike the SOTA. Moreover, our algorithm allows for an equal workload distribution per core throughout program execution. The efficacy of our algorithm is clearly shown when larger datasets (which are more popular in genomics) are employed, where we decrease the execution time from 1.1 hours to less than 90 seconds using the *E. coli genome* dataset for example. In future work, we plan to explore possible acceleration strategies on both the algorithm level (bitmasking and matrix distribution) as well as the compiler/runtime level (message buffer size and GPU acceleration).

ACKNOWLEDGEMENT

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

REFERENCES

[1] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, vol. 33, no. 6, pp. 623–630, 2015.

[2] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation," *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.

[3] D. Moi, L. Kilchoer, P. S. Aguilar, and C. Dessimoz, "Scalable phylogenetic profiling using minhash uncovers likely eukaryotic sexual reproduction genes," *PLoS computational biology*, vol. 16, no. 7, p. e1007553, 2020.

[4] A. Criscuolo, "On the transformation of minhash-based uncorrected distances into proper evolutionary distances for phylogenetic inference," *F1000Research*, vol. 9, p. 1309, 2020.

[5] 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*, vol. 17, no. 1, pp. 1–14, 2016.

[6] S. Behera, J. S. Deogun, and E. N. Moriyama, "Minisoclust: Isoform clustering using minhash and locality sensitive hashing," in *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2020, pp. 1–7.

[7] S. Seth, N. Välimäki, S. Kaski, and A. Honkela, "Exploration and retrieval of whole-metagenome sequencing samples," *Bioinformatics*, vol. 30, no. 17, pp. 2471–2479, 2014.

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

[9] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, vol. 234, no. 5323, pp. 34–35, 1971.

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

[11] D. N. Baker and B. Langmead, "Dashing: fast and accurate genomic distances with hyperloglog," *Genome biology*, vol. 20, pp. 1–12, 2019.

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

[13] S. Park and D.-Y. Kim, "Assessing language discrepancies between travelers and online travel recommendation systems: Application of the jaccard distance score to web data mining," *Technological Forecasting and Social Change*, vol. 123, pp. 381–388, 2017.

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

[15] 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*. IEEE, 2009, pp. 1–6.

[16] 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)*, vol. 58, 2000, p. 64.

[17] S. E. Schaeffer, "Graph clustering," *Computer science review*, vol. 1, no. 1, pp. 27–64, 2007.

[18] S. Bag, S. K. Kumar, and M. K. Tiwari, "An efficient recommendation generation using relevant jaccard similarity," *Information Sciences*, vol. 483, pp. 53–64, 2019.

[19] Y. Yuan, M. Chao, and Y.-C. Lo, "Automatic skin lesion segmentation using deep fully convolutional networks with jaccard distance," *IEEE transactions on medical imaging*, vol. 36, no. 9, pp. 1876–1886, 2017.

[20] 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/CVF conference on computer vision and pattern recognition*, 2019, pp. 658–666.

[21] 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*, 2007, pp. 131–140.

[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*, 2017, pp. 1–8.

[23] V. Sachdeva, D. M. Freimuth, and C. Mueller, "Evaluating the jaccard-tanimoto index on multi-core architectures." in *ICCS (1)*, 2009, pp. 944–953.

[24] J. Bank and B. Cole, "Calculating the jaccard similarity coefficient with map reduce for entity pairs in wikipedia," *Wikipedia Similarity Team*, vol. 1, p. 94, 2008.

[25] 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*, 2010, pp. 495–506.

[26] S. R. Paul, A. Hayashi, K. Chen, and V. Sarkar, "A productive and scalable actor-based programming system for pgas applications," in *Computational Science–ICCS 2022: 22nd International Conference, London, UK, June 21–23, 2022, Proceedings, Part I*. Springer, 2022, pp. 233–247.

[27] 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*, vol. 74, no. 12, pp. 3176–3190, 2014.

[28] "Hclib-actor documentation," https://hclib-actor.com, 2022.

[29] F. M. Maley and J. G. DeVinney, "Conveyors for streaming many-to-many communication," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2019, pp. 1–8.

[30] S. M. Imam and V. Sarkar, "Selectors: Actors with multiple guarded mailboxes," in *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, 2014, pp. 1–14.

[31] S. R. Paul, A. Hayashi, K. Chen, Y. Elmougy, and V. Sarkar, "A fine-grained asynchronous bulk synchronous parallelism model for pgas applications," *Journal of Computational Science*, vol. 69, p. 102014, 2023.

[32] Y. Elmougy, A. Hayashi, and V. Sarkar, "Highly scalable large-scale asynchronous graph processing using actors," in *2023 23rd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023.

[33] S. Beamer, K. Asanović, and D. Patterson, "Gail: The graph algorithm iron law," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015, pp. 1–4.

[34] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 56–65.

[35] O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 212–223.
[36] D. Bortolotti, F. Paterna, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini, "Exploring instruction caching strategies for tightly-coupled shared-memory clusters," in *2011 International Symposium on System on Chip (SoC)*. IEEE, 2011, pp. 34–41.
[37] S. Kaufmann and B. Homer, "Craypat-cray x1 performance analysis tool," *Cray User Group (May 2003)*, 2003.
[38] R. Sakellariou and J. R. Gurd, "Compile-time minimisation of load imbalance in loop nests," in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 277–284.
[39] Z. Jia, J. Zhan, L. Wang, C. Luo, W. Gao, Y. Jin, R. Han, and L. Zhang, "Understanding big data analytics workloads on modern processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1797–1810, 2016.
[40] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "Bella: Berkeley efficient long-read to long-read aligner and overlapper," in *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 2021, pp. 123–134.