Parallel Algorithms for Computing Jaccard Weights on Graphs using Linear Algebra

Elaheh Hassani*, Md Taufique Hussain† and Ariful Azad‡
* Indiana University, Bloomington, IN, USA (ehassani@iu.edu)
† Indiana University, Bloomington, IN, USA (mth@indiana.edu)
‡ Indiana University, Bloomington, IN, USA (azad@iu.edu)

***Abstract*—Jaccard similarity between a pair of vertices in a graph measures the relative overlap among their adjacent vertices. This metric is used to estimate the strength of existing edges and predict new edges between pairs of disconnected vertices. Computing Jaccard similarity for all pairs of vertices or for all edges is computationally expensive. Existing sequential and parallel algorithms are either too slow or do not scale well for large scale graphs. We present a shared-memory parallel algorithm for computing Jaccard weights. Our algorithm relies on sparse linear algebraic operations that utilize masking, semirings, vector iterators, and other GraphBLAS features for performance. Our implementation, albeit simple, outperforms recent state-of-the-art implementations by a factor of up to 20× and exhibits an average speedup of 9×.**

I. Introduction

Given two sets A and B, Jaccard Similarity or Jaccard Index is computed by the ratio of the size of their intersection and the size of their union: $|A \cap B|/|A \cup B|$. The Jaccard similarity is widely used in many applications, such as comparing DNA sequences [1] and data mining [2]. In this paper, we consider the application of Jaccard similarity to measure the similarity between two vertices within a graph. Given a pair of vertices $\{u, v\}$ in a graph, their Jaccard similarity is computed as follows:

$$J(u,v) = \frac{|N(u) \bigcap N(v)|}{|N(u) \bigcup N(v)|} \qquad (1)$$

where $N(u)$ denotes the set of neighbors of u. $J(u, v)$ takes a value between zero and one, indicating the strength of ties between u and v. Based on this similarity, we can quantify the extent to which two vertices exhibit similarity in their immediate neighborhood. Such intuition can be used to recommend friends on social networks and understand the growth of complex networks via triadic closure [3], [4].

Computing Jaccard similarities on a graph has gained attention over the last few years. It has been proposed as a potential graph benchmark [5]. IARPA used it as a benchmark to evaluate the next generation of computing architecture[1]. Considering the increasing interest in this problem, several sequential and parallel algorithms were proposed in the literature. These algorithms have various computational complexity [5] and use different implementation techniques including MapReduce [6], GPUs [7], [8], distributed-memory parallel algorithms [1].

[1]https://www.iarpa.gov/research-programs/agile

Most previous algorithms for computing Jaccard similarities on a graph are customized for a specific computing platforms. In this paper, we take an alternative route by mapping Jaccard's computation to sparse matrix operations and then using a parallel sparse matrix library to implement the algorithm. This two-step approach makes the algorithm general purpose, which can be implemented for any computing platform with the support of a sparse matrix library. We experimentally demonstrate that our linear-algebraic Jaccard computation is 3 to 4 times faster (on multicore CPUs) than other customized algorithms.

Linear-algebraic approach for graph algorithm has been popularized by the GraphBLAS standard [9]. GraphBLAS inspired many state-of-the-art graph algorithms such as connected components [10] and triangle counting [11]. We followed the footsteps of these algorithms and designed a GraphBLAS-style Jaccard algorithm. We identify two phases in the computation: (a) neighborhood intersection (the numerator in Eq. 1) that can be mapped to a sparse matrix-matrix multiplication (SpGEMM) and (b) neighborhood union (the denominator in Eq. 1) that can be mapped to several vector operations. For the neighborhood union computation, we deviate from the GraphBLAS standard by using custom iterators for faster calculations. Finally, we use simple load-balancing schemes to make the algorithm scalable for irregular graphs. These techniques together deliver a highly parallel algorithm that scales almost linearly to 64 cores on a multicore processor and runs up to an order of magnitude faster than a recent degree-aware algorithm [7] for this problem.

There are two variants of the Jaccard similarity computations that have been discussed in the literature. $J(u, v)$ is called the *Jaccard weight* when $\{u, v\}$ is an edge in the graph. By contrast, the *Jaccard similarity* is computed between any pair of vertices that may or may not be connected by an edge. We maintain this distinction throughout the paper. In particular, Jaccard weights can be computed faster by using a masked SpGEMM where the input graph works as a mask to eliminate unnecessary computations. On the other hand, Jaccard similarity computation cannot use any masking because it needs to compute similarities for all pairs of vertices. Even then the Jaccard similarity matrix is expected to be a sparse matrix for most real-world graphs. We develop algorithms for both Jaccard weight and Jaccard similarity computations.

The contributions of the paper are as follows:

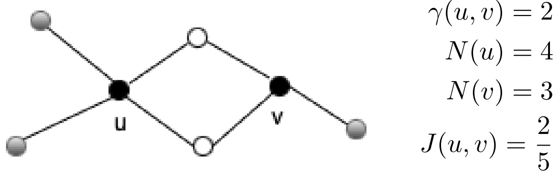
1) We develop algorithms for both Jaccard weight and Jaccard similarity computations using sparse linear algebra operations.
2) Based on the nature of Jaccard computations, we customize linear algebra operations utilizing symmetry, masking, semirings, and vector iterators.
3) Our implementation based on Suitesparse GraphBLAS is up to an order of magnitude faster than Jaccard-ML [7], a state-of-the-art algorithm for computing Jaccard weights.

II. Background and Related Work

To simplify Eq. 1, let $\gamma(u,v)$ represent the number of common neighbors of a node pair $\{u,v\}$ and $d(u)$, $d(v)$ represent the number of neighbors of u, v respectively. Then, Eq. 1 can be written as

$$J(u,v) = \frac{\gamma(u,v)}{d(u)+d(v)-\gamma(u,v)} \tag{2}$$

Figure 1 shows an example of computing the Jaccard similarity score for a pair vertices $\{u,v\}$.



$$\gamma(u,v) = 2$$
$$N(u) = 4$$
$$N(v) = 3$$
$$J(u,v) = \frac{2}{5}$$

Fig. 1: Calculation of the Jaccard similarity between a pair vertices $\{u,v\}$. White vertices are common neighbors between u and v.

Calculating the Jaccard similarity for all pairs of nodes in a graph can be computationally demanding because it involves computing set intersection for all pairs. It can become especially demanding for large graphs, for which parallelization might be necessary to utilize the power of modern computational resources. Like many other graph algorithms, getting good parallel performance is challenging due to irregular memory accesses and possible load imbalance across processes. In this paper, we translate the computation of Eq. 2 for all pairs of nodes to sparse linear algebraic operations so that we can utilize the computational techniques employed in the sparse linear algebra domain.

A. Related work

There have been several algorithms proposed to efficiently compute Jaccard similarity and Jaccard weight. For the calculation of Jaccard distances between genomes, Besta *et al.* [1] proposed a Jaccard similarity computation algorithm. Krawezik *et al.* [12] presents a parallel implementation of Jaccard similarity computation on the Emu architecture. Aljundi *et al.* [7] introduced a Jaccard Weight algorithm with an ML-based work distribution technique using GPUs. Fender *et al.* [13] developed an efficient parallel Jaccard Weight algorithm based on binary search using GPUs.

Algorithm 1 Algorithm for Jaccard similarity

Input and Output: Input adjacency matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ in CSR format and output Jaccard similarity matrix $\mathbf{J} \in \mathbb{R}^{n\times n}$ where $\mathbf{J}[u,v]$ represents value of Eqn. 2 for the vertex pair $\{u,v\}$

```
1:  procedure JACCARDSIMILARITY(A)
2:      Let t = 0.
3:      A ← SYMMETRICISE(A)
4:      SR ← SEMIRING(+, ∧)
5:      J ← SPGEMM(A, A, SR)          ▷ Intersection count
6:      e ← 𝟙^{n×1}                   ▷ Dense vector of all 1s
7:      d ← SPMV(A, e, SR)             ▷ Neighbor count
8:      J ← UPPERTRIANGLE(J)     ▷ Extract upper triangle section
9:      for (u, v) ∈ NONZEROES(J) do
10:         J[u, v] ← J[u,v] / (d[u]+d[v]−J[u,v])
11:     return J
```

III. Jaccard-LA: Parallel algorithms for computing Jaccard weights in linear algebra

A. Linear algebraic formulation of Jaccard similarity

We design the Jaccard similarity algorithm using two major steps. The outline of our algorithm is described in the Algorithm 1.

First, we calculate the numerator of the Eq. 1 by computing the square of the adjacency matrix using sparse matrix multiplication (SpGEMM) with $(+,\wedge)$ semiring (Line 5, Algorithm 1. $\wedge$ represents the logical AND operation). In most practical graphs, the resulting matrix is expected to contain more nonzeros than the original matrix. An example of this computation on a toy graph is shown in Fig. 2.

Second, we calculate the denominator of Eqn 1 and divide the numerator to get the similarity score. Because the denominator represents the cardinality of the union of neighborhood sets corresponding to a vertex pair, this operation can also be computed by performing SpGEMM on the same operand as the previous step except with $(+,\vee)$ semiring. Using SpGEMM to compute the union of neighborhoods would give us a proper GraphBLAS solution because it relies on operations defined in the GraphBLAS standard. However, using SpGEMM for computing the union of neighborhoods could perform a lot of redundant computations that are not needed to compute Jaccard similarity. This redundancy originated from the fact that Jaccard similarity defined in Eq. 1 needs to compute $\{N(u) \cup N(v)\}$ only when $\{N(u) \cap N(v)\}$ is nonempty. Hence, instead of using SpGEMM, we compute the degree of each vertex (Line 6-7, Alg. 1) and then for each nonzero intersection, we use the degree information to compute the similarity using Eqn 2(Line 9-10, Alg. 1). Because Jaccard similarity is a symmetric relationship, to further reduce the computation we perform the second step only on the upper triangular part of the intersection count matrix (Line 8, Alg. 1). Note that the latter part of Alg. 1 does not use standard GraphBLAS operations.

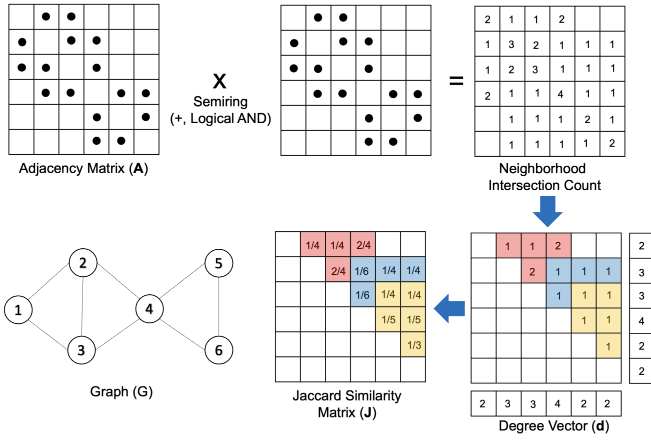


Fig. 2: Steps of our Jaccard similarity algorithm (Alg. 1) on a toy graph of six vertices. The top row shows the neighborhood intersection count computation. In the bottom row, different colors represent different threads involved in computing the neighborhood union count in parallel when three threads are employed (when sparse matrices are stored in CSR format).

B. Computational Complexity

We analyze the computational complexity of Alg. 1 by considering the adjacency matrix to be an Erdős-Rényi matrix. If the adjacency matrix has n rows/columns and an average number of nonzeros of d, then computing the intersection count with SpGEMM would take $O(nd^2)$ time [14]. For each possible n^2 non-zero entry in the intersection count matrix probability of the existence of the non-zero is $n \times \frac{d^2}{n^2} = \frac{d^2}{n}$. So, the expected total number of non-zeros is nd^2. In the last step, we iterate over these non-zeros to compute the Jaccard similarity coefficient. Hence the total time complexity of our algorithm is the same as the SpGEMM operation $O(nd^2)$.

C. Parallel Jaccard Similarity Algorithm

At first, we discuss a parallel algorithm for Jaccard similarity for each pair of vertices with at least one common neighbor. The pair of vertices may or may not be connected by an edge. To design a parallel algorithm for Jaccard similarity, we make use of parallel linear algebraic modules. The outline of our parallel Jaccard similarity algorithm is described in Alg. 2. The major steps are the same as the Alg. 1 except we use parallel SpGEMM and sparse matrix-vector multiplication (SpMV) to compute the intersection count matrix and degree vector (Line 5 and 7, Alg. 2). Because the last step of our algorithm is as expensive as the SpGEMM operation and we don't make use of any linear algebraic operation, we parallelize it explicitly. We assign an equal workload to each thread by dividing the non-zero values into chunks of consecutive regions of the sparse matrix (Line 9-10, Alg. 2). Fig. 2 shows how the division takes place in a toy example when we use CSR data structures for sparse matrices. Parallelizing in this way does not ensure that the reads from the degree vector would always be consecutive - which creates some potential for cache misses. However, the size of the degree vector is much smaller compared to the non-zero array(n vs nd^2), and memory operation on the degree vector is only read while memory operations on the non-zero array are both read and write. So, cache misses on the non-zero array play a more significant role in the parallel performance, hence we optimize that. This parallelization part can be optimized even more by tiling the upper triangular portion of the intersection count matrix into appropriate chunks so that memory operations on both arrays are optimized. However, we did not attempt it as we did not notice any evidence of performance degradation due to this issue in our experiments.

Algorithm 2 Parallel Algorithm for Jaccard similarity

Input and Output: Input adjacency matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ in CSR format and output Jaccard similarity matrix $\mathbf{J} \in \mathbb{R}^{n\times n}$ where $\mathbf{J}[u, v]$ represents value of Eqn. 2 for the vertex pair $\{u, v\}$

1: **procedure** ParallelJaccardSimilarity($\mathbf{A}$)
2: Let $t = 0$.
3: $\mathbf{A} \leftarrow$ Symmetricise($\mathbf{A}$)
4: SR $\leftarrow$ Semiring($+$, $\wedge$)
5: $\mathbf{J} \leftarrow$ Parallel-SpGEMM($\mathbf{A}$, $\mathbf{A}$, SR) ▷ Intersection count
6: $\mathbf{e} \leftarrow \mathbb{1}^{n\times 1}$ ▷ Dense vector of all 1s
7: $\mathbf{d} \leftarrow$ Parallel-SpMV($\mathbf{A}$, $\mathbf{e}$, SR) ▷ Neighbor count
8: $\mathbf{J} \leftarrow$ UpperTriangle($\mathbf{J}$) ▷ Extract upper triangle section
9: $T \leftarrow$ ThreadCount
10: **for** $(u, v) \in$ NonZeros($\mathbf{J}$)$[t \times \frac{nnz}{T} : (t+1) \times \frac{nnz}{T}]$ in parallel **do** ▷ t is thread id
11: $\mathbf{J}[u, v] \leftarrow \frac{\mathbf{J}[u,v]}{\mathbf{d}[u]+\mathbf{d}[v]-\mathbf{J}[u,v]}$
12: **return J**

Algorithm 3 Parallel Algorithm for Jaccard weight

Input and Output: Input adjacency matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ in CSR format and output Jaccard weight matrix $\mathbf{J} \in \mathbb{R}^{n\times n}$ where $\mathbf{J}[u, v]$ represents value of Eqn. 2 for the vertex pair $\{u, v\}$

1: **procedure** ParallelJaccardWeight($\mathbf{A}$)
2: Let $t = 0$.
3: $\mathbf{A} \leftarrow$ Symmetricise($\mathbf{A}$)
4: SR $\leftarrow$ Semiring($+$, $\wedge$)
5: $\mathbf{J} \leftarrow$ Parallel-Masked-SpGEMM($\mathbf{A}$, $\mathbf{A}$, SR, mask=$\mathbf{A}$) ▷ Intersection count
6: $\mathbf{e} \leftarrow \mathbb{1}^{n\times 1}$ ▷ Dense vector of all 1s
7: $\mathbf{d} \leftarrow$ Parallel-SpMV($\mathbf{A}$, $\mathbf{e}$, SR) ▷ Neighbor count
8: $\mathbf{J} \leftarrow$ UpperTriangle($\mathbf{J}$) ▷ Extract upper triangle section
9: $T \leftarrow$ ThreadCount
10: **for** $(u, v) \in$ NonZeros($\mathbf{J}$)$[t \times \frac{nnz}{T} : (t+1) \times \frac{nnz}{T}]$ in parallel **do** ▷ t is thread id
11: $\mathbf{J}[u, v] \leftarrow \frac{\mathbf{J}[u,v]}{\mathbf{d}[u]+\mathbf{d}[v]-\mathbf{J}[u,v]}$
12: **return J**

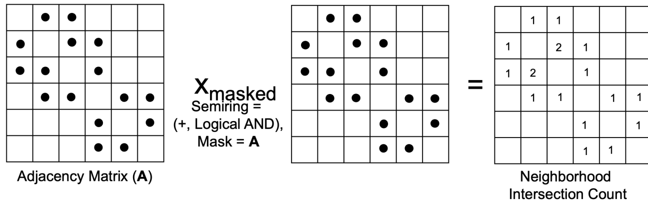


Fig. 3: Neightborhood intersection count for Jaccard Weight using Masked-SpGEMM (Alg. 3) on a toy graph of six vertices.

D. Parallel Jaccard Weight Algorithm

The Jaccard Weight algorithm is a special case of the Jaccard Similarity algorithm, where we compute the Jaccard similarity between two endpoints of each edge in the graph. Because it only needs to compute for a subset of pairs involved in Jaccard Similarity, we further optimize the algorithm by computing the intersection count of only relevant pairs. We do it by employing Masked-SpGEMM instead of general SpGEMM operation with the adjacency matrix as the mask and the same semiring as the Jaccard Similarity algorithm(Line 5, Alg. 3). Given a mask, a Masked-SpGEMM operation computes non-zero entries of the resulting SpGEMM operation only at the location of the given mask. Fig. 3 shows the neighborhood intersection count computation of Alg. 3 using Masked-SpGEMM operation.

IV. Results

A. Platform

We evaluate the performance of parallel Jaccard algorithms on Big Red 200, an HPE Cray EX supercomputer at IU. Each compute node is equipped with 256 GB Memory and two 64-core 2.25 GHz AMD EPYC 7742. We used Suitesparse GraphBLAS [15] (version 7.3.3) in C to implement our linear-algebraic algorithm for computing Jaccard weights.

B. Dataset

Table I describes a representative set of graphs from the SuiteSparse matrix collection [16]. In the preprocessing step, we symmetrize each input matrix and remove diagonal entries (self-loops) from them. The preprocessed matrix A is considered as the adjacency matrix of an undirected graph and passed to the Parallel Jaccard algorithm as input. Here, the "#Jaccard Weights" and "#Jaccard Similarities" indicate the number of nonzero Jaccard values computed in the Jaccard Weight algorithm and Jaccard Similarity algorithm, respectively.

C. Relative performance of algorithm

We compare the performance of the Jaccard weights algorithm with the state-of-the-art algorithm Jaccard-ML [7], which is a degree-aware CPU kernel utilizing machine learning for tuning the work distribution on the CPU. We also compared our parallel Jaccard weight and Jaccard similarity algorithm with the naive-Jaccard GraphBLAS algorithm. With naive-Jaccard, which uses only GraphBLAS operations, computing the intersection matrix (the numerator in Eq. 1) is identical to our parallel Jaccard algorithm. Then, we computed neighborhood union (the denominator in Eq. 1) by first building a matrix S where every entry s_{ij} denotes $s_{ij} = d_i + d_j$ where d_i and d_j represent the degree of nodes i and j in the input adjacency matrix A. To compute the output Jaccard matrix J, we used the intersection matrix B and an ElementWise Matrix operation $J = J/S - J$. Building Matrix S requires extracting nonzero positions from intersection matrix B and generating a vector containing values of d_i and d_j for every entry of S. These operations are time and memory-consuming compared to our Jaccard algorithm. The comparison of runtimes for the calculation of Jaccard similarity and Jaccard weights using different Algorithms is shown in Table II and III, respectively.

As shown in Table II, our Jaccard Similarity algorithm is $3\times$ faster than the Naive GraphBLAS algorithm. This happens due to the load-balancing of our parallel iteration in the neighborhood union calculation part. Plus, computing Jaccard Similarities using Naive GraphBLAS Jaccard algorithm failed to be completed to big graphs due to memory consumption for building matrix D. For Jaccard Weights computation, our Parallel algorithm is $2.5\times$ faster than the Naive GraphBLAS algorithm and $9\times$ faster than Jaccard-ML. This is because SpGEMM benefits from the sparsity of matrices in matrix multiplication. As shown in Table III, the number of calculated Jaccard Weights is normally much smaller than the number of Jaccard Similarities for the same graph. Especially in big social graphs Parallel Jaccard Weights algorithm is much faster due to the sparsity of the output matrix. Table III clearly shows the benefit of linear algebraic approaches where even naive implementation is faster than Jaccard-ML.

D. Scalability

Table IV shows runtimes the Jaccard similarity and Jaccard weight algorithms on single core and 64 cores of Big Red 200. This table also shows the speedups for these algorithms relative to their runtimes on a single core. Jaccard Similarity could not be computed for big social graphs such as `webbase-2001` and `uk-2002` because the intersection matrices do not fit in memory. On average, Jaccard Similarity and Jaccard Weight algorithms are $31.23\times$ and $20.53\times$ faster on 64 cores, respectively. These results show that our algorithms scale very well on the multicore processor used in our experiments. For smaller graphs, Jaccard weight is not as scalable as Jaccard similarity. Figure 4 shows the scalability of (a) parallel Jaccard similarity and (b) parallel Jaccard weight algorithms on Big Red 200 for several graphs. The average speedup on 128 cores relative to the same algorithm on a single core is 32.4 with stdev=6.4 for the Jaccard Similarity algorithm and 30.4 with stdev=21.4 for the Jaccard Weight algorithm. There is a high standard deviation in the speedup of the Jaccard weight algorithm due to the sparsity of the output Jaccard matrix for some graphs. For example in a road network like europe-osm, we can not benefit from a high number of cores because of the small number of Jaccard weight values we compute.

TABLE I: Test problems for evaluating Jaccard algorithms.

Graph	#Vertices	#Edges	Max Degree	Avg. Degree	#Jaccard weights	#Jaccard Similarities
com-Youtube	1,134,890	2,987,624	28,754	5.25	1,397,278	1,259,704,147
com-Orkut	3,072,441	234,370,166	33,313	76.28	101,196,543	
soc-LiveJournal1	4,847,571	68,993,773	20,333	17.68	38,694,506	4,200,885,672
cage15	5,154,859	99,199,551	94	19.24	47,022,346	461,934,194
wb-edu	9,845,725	57,156,537	25,781	9.39	42,747,719	4,244,285,350
uk-2002	18,520,486	298,113,762	194,955	16.096	261,200,630	
webbase-2001	118,142,155	1,019,903,190	816,127	8.633	816,367,013	
europe_osm	50,912,018	108,109,320	13	2.12	183,231	65,645,839

TABLE II: Runtime comparison of Jaccard Similarity algorithms on 128 cores of Big Red 200.

Graph	Naive Jaccard	Parallel Jaccard Similarity
com-Youtube	23.27	6.23
soc-LiveJournal	OOM	17.94
cage15	6.05	1.633
wb-edu	OOM	11.20
europe_osm	2.28	1.28

TABLE III: Comparison of Jaccard Weight algorithms runtime on 128 cores of Big Red 200 in sec.

Graph	Jaccard-ML	Naive Jaccard	Parallel Jaccard Weight
com-Youtube	0.76	0.41	0.24
com-Orkut	119.97	34.91	8.45
soc-LiveJournal	34.05	5.25	1.64
cage15	2.24	1.00	1.64
wb-edu	6.20	1.86	0.61
uk-2002	51.13	9.02	4.84
webbase-2001	185.03	27.75	14.22
europe_osm	4.28	1.28	0.27

TABLE IV: Speedup of Parallel Jaccard-LA Algorithms. Jaccard Similarity could not be computed for big social graphs such as `webbase-2001` and `uk-2002` because intersection matrices are dense and go out of memory.

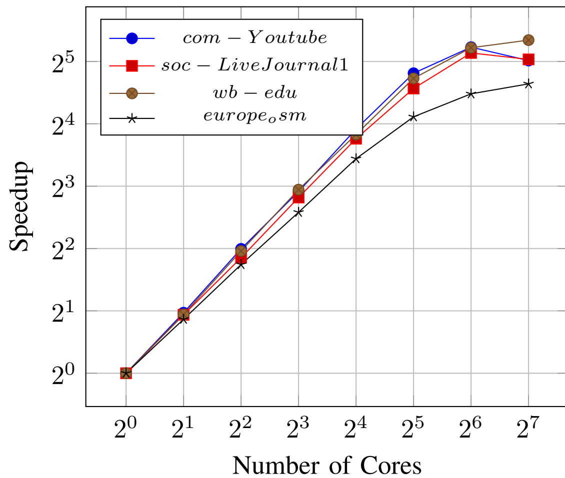
	#Cores=1		#Cores=64		Speedup	
Graph	Jaccard Weights(sec)	Jaccard Similarity(sec)	Jaccard Weights(sec)	Jaccard Similarity	Jaccard Weights	Jaccard Similarity
com-Youtube	3.31	237.26	0.25	6.99	13.39	33.94
com-Orkut	499.57	OOM	29.61	OOM	16.87	-
soc-LiveJournal1	84.15	1,108.98	4.13	30.89	20.37	35.90
cage15	14.69	98.46	0.62	2.98	23.66	33.00
wb-edu	27.55	794.14	1.11	23.74	24.74	33.46
uk-2002	336.48	OOM	11.88	OOM	28.32	-
webbase-2001	856.56	OOM	38.81	OOM	22.07	-
europe_osm	3.80	17.16	0.26	0.76	14.83	22.57

E. Breakdown of runtime

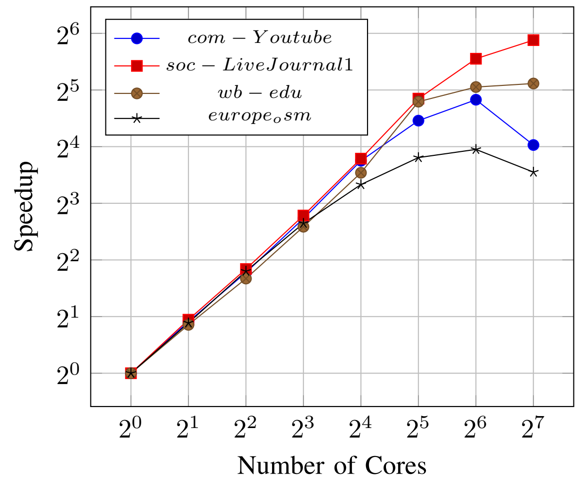
The breakdown of runtimes of (a) Jaccard similarity and (b) Jaccard weight algorithms for different input graphs on 128 cores of Big Red 200 is given in Figure 5. Here, “Degree” denotes the time to compute the degree of all vertices, “Intersection” denotes the time to compute the intersection of all pairs of nodes, “Masked Intersection” is the time to calculate the intersection of adjacent nodes, “Selection” shows the time to select the upper triangle of intersection matrix, and “Union” indicates the computation time to iterate over intersection matrix and compute Jaccard in parallel. Figure 5(a) shows that the Jaccard similarity algorithm spends around 50% of the runtime on calculating the intersection matrix and selecting the upper triangle of it. This is because the intersection matrix is usually much denser than the input matrix. For the Jaccard weight algorithm, “Masked Intersection” took nearly 80% of the runtime. Due to the sparsity of the intersection matrix, the “Union” and “Selection” parts are quite fast. As shown in Table I, the number of Jaccard similarities calculated is more than 100x larger than the Jaccard weights to be calculated. The parallel computation of the union is faster on most graphs after we have the intersection matrix, as it is load-balanced.

V. Conclusion and Future Works

Research on Jaccard similarity computation on a graph has gained significant attention due to its role in many important

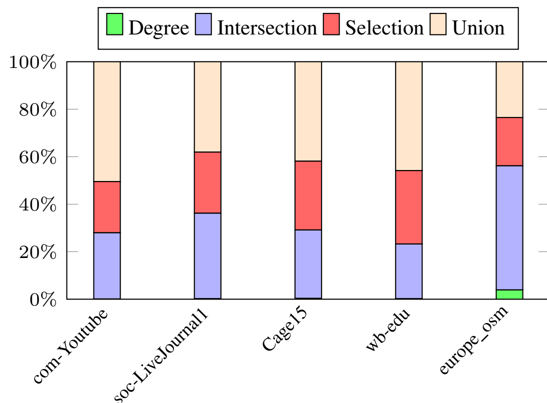


(a) Scalability of Jaccard Similarity algorithm.

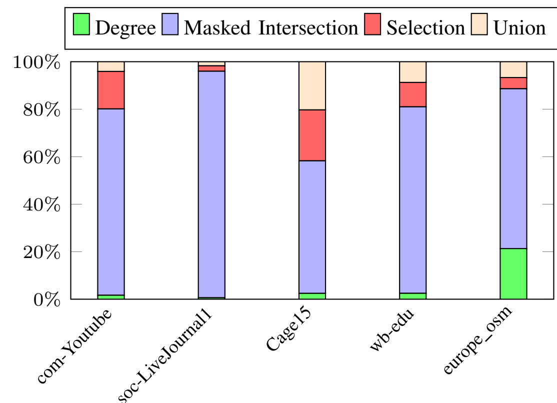


(b) Scalability of Jaccard Weight algorithm.

Fig. 4: Scalability of Jaccard Similarity and Jaccard Weight algorithm.



(a) Jaccard Similarity.



(b) Jaccard Weights.

Fig. 5: Breakdown of runtimes for Jaccard Similarity and Jaccard Weights algorithms.

applications. In this work, we propose a linear algebraic (LA) formulation of Jaccard similarity computation. Such a formulation is especially helpful when graphs are large and parallel computing is necessary. While non-LA formulation requires many different manual tuning to get a good performance, LA formulation has the potential to give good parallel performance out of the box by utilizing all the optimization techniques developed in the sparse linear algebra domain. We showed experimental evidence that our algorithm could make one variant of Jaccard similarity computation significantly faster than the state-of-the-art method. We hope our exploration opens up opportunities for new scientific discoveries.

This work opens up several possible scopes for future exploration. The neighborhood intersection count matrix becomes significantly denser than the input adjacency matrix. We reduce the density in half by selecting the upper triangular part. However, it might be possible to reduce the memory footprint and computation even further by only generating the upper triangular portion during the SpGEMM process. We saw in our experiment that our algorithm was unable to compute the Jaccard similarity for large graphs due to the memory requirement being too high. Distributed memory parallel method may overcome the issue but that would bring additional complexity of communication and load-balance between processes. That is another scope for future studies.

VI. Acknowledgements

This research is partially supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contracts numbered DE-SC0022098 and DE-SC0023349.

References

[1] 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)*. IEEE, 2020, pp. 1122–1132.

[2] 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)*. IEEE, 2015, pp. 370–372.

[3] D. Easley, J. Kleinberg *et al.*, *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge university press Cambridge, 2010, vol. 1.

[4] M. S. Granovetter, "The strength of weak ties," *American journal of sociology*, vol. 78, no. 6, pp. 1360–1380, 1973.

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

[6] 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.

[7] A. A. Aljundi, T. A. Akyildiz, and K. Kaya, "Degree-aware kernels for computing jaccard weights on gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 897–907.

[8] H. Anzt and J. Dongarra, "A jaccard weights kernel leveraging independent thread scheduling on gpus," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 229–232.

[9] 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)*. IEEE, 2016, pp. 1–9.

[10] Y. Zhang, A. Azad, and Z. Hu, "Fastsv: A distributed-memory connected component algorithm with fast convergence," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 46–57.

[11] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 804–811.

[12] 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)*, 2018, pp. 1–6.

[13] 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.

[14] G. Ballard, A. Buluc, 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*, 2013, pp. 222–231.

[15] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.